Spring MVC

～Spring Framework 3.0/3.1～

クックブック

【2013.01.06.02 版】

Tatsuo TSUCHIE　　著

Project *Green-Day*

【変更履歴】

No.	版	変更箇所	変更内容
1	2013.01.06	5 章	REST サービスの作り方の章を作成。
2			
3			

【目次】

1. Spring3.1 の変更点

Spring3.0 から 3.1（2011 年 11 月）へバージョンアップされたことにより、Spring MVC も細かな修正が入り、より使いやすくなりました。特に、フラッシュスコープ（リダイレクト先もパラメータが有効）が正式に組み込まれました。

本書では、Spring3.1 で追加された機能で、特に Spring MVC にも触れていきます。

1.1. Servlet 3 に対応

Servlet3.0 の web.xml の書式に対応しました。Servlet 3.0 を使用する場合、Tomcat7 を使用する必要があります。また、servlet-api 3.0 を使用するために、「2.4.1.1　SpringMVC の依存ファイル」の pom.xml の内容を変更する必要があります。

1.1.1. Sevrlet3.0 を使用するための web.xml の記述の変更

web.xml で Servlet3.0 から追加になった新しい記述方法を利用する場合、XML Schema の定義の変更をする必要があります。単純にバージョンを 2.5⇒3.0 に変更するだけです。

【web.xml：Servlet2.5 の場合】

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
・・・省略
</web-app>
```

【web.xml：Servlet3.0 の場合】

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">
・・・省略
</web-app>
```

1.1.2. TLD（タグライブラリ定義）ファイルの配置場所の変更

- 今まで、「META-INF/」や、「WEB-INF/lib/」に配置していた場合は、「WEB-INF」の直下に変更する必要があります。
  - 従来のように web.xml に<taglib>タグを使用し配置場所を直接記述しても読み込むことができます。
- 「WEB-INF/lib/」以下の jar ファイルに含まれる TLD ファイルは今までと同様読み込むことができます。

【web.xml：任意の場所に配置した tld ファイルを読み込む記述】

```
<jsp-config>
    <taglib>
        <taglib-uri>http://www.example.co.jp/authorize/tags</taglib-uri>
        <taglib-location>/WEB-INF/lib/authorize.tld</taglib-location>
    </taglib>
</jsp-config>
```

1.1.3. Servlet3.0 を使用するための pom.xml の記述の変更

Servlet3.0 を使用するためには、pom.xml の記述を変更します。JSP のバージョンも変更されているので注意してください。

【pom.xml：Tomcat6 (Servlet 2.5/JSP 2.1)】

```
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.1</version>
            <scope>provided</scope>
        </dependency>
```

【pom.xml：Tomcat7(Servlet 3.0/JSP2.2)】

```
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.2</version>
            <scope>provided</scope>
        </dependency>
```

<artifactId>の値が変更されているので注意してください。

1.1.4. Spring の定義情報読み込み方法の追加

今まで web.xml で記述していた Spring の Context の設定をプログラムで記述することが可能な「org.springframework.web.WebApplicationInitializer」が追加されました。

【Spring 3.0 の web.xml の記述】

```
<web-app>
・・・省略
 <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
       <param-name>contextConfigLocation</param-name>
       <param-value>/WEB-INF/spring/dispatcher-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
 </servlet-mapping>
・・・省略
</web-app>
```

【Spring 3.1 の記述】

```
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;

public class MyWebAppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
       XmlWebApplicationContext appContext = new XmlWebApplicationContext();
       appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

       ServletRegistration.Dynamic dispatcher =
          container.addServlet("dispatcher", new DispatcherServlet(appContext));
       dispatcher.setLoadOnStartup(1);
       dispatcher.addMapping("/");
    }

 }
```

1.1.5. Servlet3.0 のマルチパート（ファイルアップロード）に対応

Servlet 3 から、ファイルアップロードのためのマルチパートが組み込まれました。今まで、別 API の「Commons FileUpload」を使用していましたが、外部の API が不要になりました。

「org.springframework.web.multipart.MultiPartResolver」の実装クラスとして、「org.springframework.web.multipart.support.StandardServletMultipartResolver」が追加されました。

Servlet3 のマルチパートを使用する場合、アップロードファイルのサイズの上限値の設定などを「web.xml」の<multipart-config>要素で設定します。Servlet 3 になり、マルチパートの設定を、@MultipartConfig を使用し、Servlet クラスのリクエストごとに設定可能になりましたが、Spring MVC では Serlvet クラスは 1 つであるため、web.xml で設定します。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans">
・・・省略
    <!-- use file upload -->
    <!--  ↓Commons FileUpload の API を使用する場合  -->
    <!--<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <property name="maxUploadSize" value="${app.fileupload.maxSize}"/>
    </bean> -->

    <!--  ↓Servlet 3 のマルチパート API を使用する場合  -->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.support.StandardServletMultipartResolver">
    </bean>
</beans>
```

Commons FileUpload の代わりに使用します。

図 1.1　Serlvet3 のファイルアップロードのための multipartResolver の追加 servlet-context.xml 記述

```
<web-app>
・・・省略
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/servlet-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <multipart-config>
            <max-file-size>1000000</max-file-size>
            <max-request-size>1000000</max-request-size>
            <file-size-threshold>1000000</file-size-threshold>
        </multipart-config>
    </servlet>
・・・省略
</web-app>
```

・SpringMVC の共通 Servlet で、マルチパートの設定を行います。
・サイズの単位は、byte です。

図 1.2　マルチパートのパラメータ設定（web.xml）

【Commons FileUpload と Servlet3.0 のマルチパートの違い】

- Spring MVC 経由で使用する限りでは、web.xml と servlet-context.xml を修正するだけで、**Controller や JSP 側では変更しないで利用**できます。
- アップロード可能なファイルの上限値を超えた場合にスローされる例外が変わります（「表 1.1」）。
  - Commons FileUpload の場合は、ライブラリの例外として扱われるが、Servlet3.0 の場合はサーバの例外として扱われるので、障害時の原因切り分けなどの際に注意する必要があります。

表 1.1　アップロード可能な上限値を超えた場合にスローされる例外

項目	Commons FileUpload	Servlet3.0 のマルチパート
スローされる例外	org.springframework.web.multipart.MaxUploadSizeExceededException ※MultipartException のサブクラス	org.springframework.web.multipart.MultipartException
ラップされる例外	org.apache.commons.fileupload.FileUploadBase$SizeLimitExceededException	java.lang.IllegalStateException さらに次の例外がラップされている。 「org.apache.tomcat.util.http.fileupload.FileUploadBase$FileSizeLimitExceededException」。

1.2. @RequestPart によるマルチパートデータの処理の追加

RESTful サービスにおいて、WEB ブラウザ以外のクライアントからファイルアップロードのようなマルチパートデータのリクエストを送信する場合に Controller の引数に「@RequestPart」を指定します。

- JSON などのデータを受け取る「@RequestBody」は、“multipart/form-data”に対応していなので、そのようなときに、@RequestPart で取得します。

【クライアントから送信されるデータの例】

- 送信するデータは 2 つあります。
  - 1 つは、“meta-data”という JSON 形式のデータ。
  - 2 つ目は、“file-data”というファイルデータ。

```
POST /someUrl
Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="meta-data"
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit

{
   "name": "value"
}
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="file-data"; filename="file.properties"
Content-Type: text/xml
Content-Transfer-Encoding: 8bit
... File Data ...
```

【Controller の例】

- 引数に@RequestPart を指定する。
- 1 つめの引数の“meta-data”という名前のデータは、JSON 形式で送信されるため、MetaData という JavaBean を指定します。
  - @RequesstBody のように HttpMessageConverter で JSON から Java オブジェクトに変換されます。
- 2 つ目の引数の“file-data”という名前のファイルデータの場合は、通常のファイルアップロードと同様に、クラスとして「MultipartFile」を指定します。

```
@RequestMapping(value="/someUrl", method = RequestMethod.POST)
public String onSubmit(@RequestPart("meta-data") MetaData metadata,
                       @RequestPart("file-data") MultipartFile file) {
    // ...

}
```

1.3. アノテーションンを処理する各種 HandlerMethod の変更

この変更により、既存の Flash Scope の実装（フラッシュスコープの実装）や、認証処理の実装方法（独自実装のアノテーションを利用した権限チェック）に変更がありました。

No.	変更後(Spring 3.0)	変更前(Spring 3.1)
1	RequestMappingHandlerMapping	DefaultAnnotationHandlerMapping
2	RequestMappingHandlerAdapter	AnnotationMethodHandlerAdapter
3	ExceptionHandlerExceptionResolver	AnnotationMethodHandlerExceptionResolver

Spring3.1 から、HandlerIntercepter の実装である「HandlerIntercepterAdapter」でも、@RequestMapping のメソッドごとの情報が取得できるようになり、AspectJ を使用する必要がなくなりました。
HandlerIntercepterAdapter を利用した認証・認可チェックの例は、「9.3　独自実装のアノテーションを使用した権限チェック（Spring MVC 3.1）　」を参照してください。

1.4. アノテーション「@RequestMapping」の改善

- HTTP ヘッダーの「Content-Type」「Accept」などで設定されているメディアタイプを指定する属性「consumes」と「produces」が追加されました。詳細は、「3.3.1　アノテーション「@RequestMapping」の仕様」を参照してください。
- 既存の属性「headers」で HTTP ヘッダーは指定可能でしたが、よく使う 2 つのパターンを簡単に指定しできるようにして、可読性を高めるために追加されたものになります。
- クラスとメソッドの@RequestMapping で指定されている場合、メソッドに指定された値が優先されます。

1.4.1. リクエストのメディアタイプを指定する「consumes」属性

- サーバが受け付ける（ブラウザ、クライアントから送信された）メディアタイプ（Content-Type）を指定する場合、「consumes」を使用します。
  - 従来は、「headers="ContentType=application/json"」と指定していました。

```
@Controller
@RequestMapping(value = "/pets", method = RequestMethod.POST, consumes="application/json")
public void addPet(@RequestBody Pet pet, Model model) {
    // implementation omitted
}
```

1.4.2. レスポンスのメディアタイプを指定する「produces」属性

- サーバがブラウザに返す（ブラウザ、クライアントが受信する）レスポンスのメディアタイプ（Accept）を指定する場合、「produces」を使用します。
  - 従来は、「headers="ContentType=application/json"」と指定していました。

```
@Controller
@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, produces="application/json")
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model) {
    // implementation omitted
}
```

1.5. Flash Attribute(フラッシュ属性、フラッシュスコープ)の実装

フラッシュスコープを使用する場合、「RedirectAttributes」をコントローラのメソッドに追加します。

- 「RedirectAttributes#addFlashAttribute(…)」は、リダイレクト先で有効となります。
- 「ResirectAttributes#addAttribute(…)」は、通常通り、リダイレクト先では無効です。
- 詳細は、「3.4.4　フラッシュ属性を使用した redirect による URL 転送（Spring MVC 3.1）」を参照してください。

```
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String register(@Valid RegisterForm form, BindingResult result, RedirectAttributes attrs) {
    if (result.hasErrors()) {
        return "register/input";
    }
    attrs
        .addAttribute("id", form.getName())
        .addFlashAttribute("message", "登録されました.");
    return "redirect:/register";
}
```

1.6. URITemplate のパス変数の改良

@RequestMapping の URI の記述方法である URI Template 形式の変数（パス変数、例えば「/app/customers/{userCd}」であれば、{userCd}）が使える箇所が増えました。

- メソッドの引数の@ModelAttribute に、パス変数をバインドすることができるようになりました。
  - 詳細は、「1.6.1　URI Template 中パス変数の@ModelAttribute へのバインド」を参照してください。
- 引数に Model（ModelMap）を指定した場合、パス変数に記述した値が、引数の Model に自動的に登録されるようになりました。
  - 詳細は、「1.6.2　URI Template 中のパス変数の Model への自動登録」を参照してください。
- リダイレクト先の URI にもプレースホルダーとして、URI Template の記述ができるようになりました。
  - 例えば、「redirect:/blog/{year}/{month}」。
  - 詳細は、「1.6.3　リダイレクト先の URL に URI Template を指定する」を参照してください
- URI template のパス変数を@ModelAttribute にバインドできるようになったため、InitBinder などによる値の細かな変換もできるようになりました。
  - 詳細は、「1.6.4　URI Template のパス変数の@MdelAttribute へのバインド時のカスタマイズ」を参照してください。

1.6.1. URI Template 中パス変数の@ModelAttribute へのバインド

- URI Template のパス変数の@ModelAttribute を付加したオブジェクトにバインドできるようになりました。
- 従来では、パス変数の値は@PathVariable でしかバインドできなかったため、パス変数の個数分だけ引数が増えていました。
  - ➢ メトリクスの指針である引数の個数が多い場合を改善できます。

【従来の Spring3.0 での記述方法】

```
@RequestMapping("/spring30/sample01/{firstName}/{lastName}/SSN")
public ModelAndView search01(@PathVariable String firstName,
                             @PathVariable String lastName) {

    Person person = new Person();
    person.setFirstName(firstName);
    person.setLastName(lastName);

    ModelAndView mav = new ModelAndView("/spring31/helloView");
    mav.addObject("person", person);
}
```

【Spring3.1 での記述方法】

```
@RequestMapping(value="/spring31/sample01/{firstName}/{lastName}/SSN")
public ModelAndView search01(@ModelAttribute Person person) {

    System.out.printf("firstName=%s¥n", person.getFirstName());
    System.out.printf("lastName=%s¥n", person.getLastName());

    ModelAndView mav = new ModelAndView("/spring31/helloView");
    mav.addObject("person", person);

    return mav;
}
```

1.6.2. URI Template 中のパス変数の Model への自動登録

- 引数に Model（ModelMap）を指定した場合、パス変数に記述した値が、引数の Model に自動的に登録されるようになりました。
  - 値を次の画面へ引き継ぐようなときに、コード量を減らせます。
- ただし、引数には、@PathVariable または、@ModelAttribute でバインド先を指定しておく必要があります。
  - そのため、正しくは、「パス変数を引数@PathVariable または@ModelAttribute にバインドした結果の値を Model へ自動登録する」という動作になります。

【従来の Spring3.0 での記述方法】

```
@RequestMapping(value="/spring30/sample02/{firstName}/{lastName}/SSN")
public String search02(@PathVariable String firstName, @PathVariable String lastName, ModelMap model) {

    model.addAttribute("firstName", firstName);
    model.addAttribute("lastName", lastName);

    return "/spring30/helloView";
}
```

【Spring3.1 での記述方法（@PathVariable の場合）】

- Model への登録名は、@PathVariable のバインド先名（変数名）です。

```
@RequestMapping(value="/spring30/sample02/{firstName}/{lastName}/SSN")
public String search02(@PathVariable String firstName, @PathVariable String lastName, ModelMap model) {

    System.out.printf("firstName=%s¥n", model.get("firstName"));
    System.out.printf("lastName=%s¥n", model.get("lastName"));

    return "/spring30/helloView";
}
```

【Spring3.1 での記述方法（@ModelAttribute の場合）】

- 「1.6.1」で説明した改良により、@ModelAttribute へバインドできるようになったため、その値も Model へ登録されます。
  - ただし、Model への登録名は、@ModelAttribute のバインド先名（変数名）です。

```
@RequestMapping(value="/spring31/sample02/{firstName}/{lastName}/SSN")
public String search02(@ModelAttribute Person person, ModelMap model) {

    System.out.printf("person=%s¥n", model.get("person"));

    return "/spring31/helloView";
}
```

1.6.3. リダイレクト先の URL に URI Template を指定する

- リダイレクト先の URL に URI Template の形式を指定できるようになりました。
  - パス変数の値は、Model に格納します。

```
@RequestMapping(value="/spring31/sample03_2")
public ModelAndView search032() {

    ModelAndView mav = new ModelAndView("redirect:/spring31/sample01/{firstName}/{lastName}/SSN.html");

    // model へパス変数の値を格納する
    mav.addObject("firstName", "Hanako");
    mav.addObject("lastName", "Hayasi");

    return mav;
}
```

1.6.4. URI Template のパス変数の@MdelAttribute へのバインド時のカスタマイズ

- 「1.6.1」で説明した改良により、@ModelAttribute へバインドできるようになったため、InitBinder で Converter や PropertyEditor による細かな制御ができるようになりました。

【従来の Spring3.0 での記述方法（アノテーションによるフォーマットして）】

```
@RequestMapping(value="/spring31/sample04_2/{firstName}/{birthday}/SSN")
public ModelAndView search042(@PathVariable String firstName,
        @PathVariable @DateTimeFormat(pattern="yyyy-MM-dd") Date birthday) {

    Person person = new Person();
    person.setFirstName(firstName);
    person.setBirthday(birthday);

    System.out.println(person);

    ModelAndView mav = new ModelAndView("/spring31/helloView");
    mav.addObject("person", person);

    return mav;
}
```

【Spring3.1 での記述方法】

```
// InitBinder によるマッピングのカスタマイズ
@InitBinder
public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyy-MM-dd");
    binder.registerCustomEditor(Date.class, "birthday", new CustomDateEditor(dateFormat, true));
}

@RequestMapping(value="/spring31/sample04/{firstName}/{birthday}/SSN")
public ModelAndView search04(@ModelAttribute Person person) {
    ModelAndView mav = new ModelAndView("/spring31/helloView");
    mav.addObject("person", person);
    return mav;
}
```

1.7. @RequestBody 時に指定した@Valid アノテーションの動作

JSON や XML 形式のデータを受信する場合、リクエストのメソッドの引数 Command に@ResponseBody を付与します。Command に対して Bean Validation などを使用して自動で入力値チェックを行いたい場合、Spring3.0 では動作しませんでした。Spring3.1 からは正常に動作するようになりました。

- **値が不正な場合は、HTTP ステータスコード「400（Bad Request）」を返す**ようになりました。
- 詳細は、「5.5　REST サービスによるエラー処理」を参照してください。

1.8. URI を組み立てるための UriComponentsBuilder の追加

- RFC 6570（http://tools.ietf.org/html/rfc6570）の URI Template の仕様の実装である、「UriTemplate」クラスをより柔軟に実装したものになります。
- メソッドチェーンを利用することによって流れるようなインタフェースで、REST サービスにアクセスする URI が組み立てる URI ことができます。
- 詳細は、「5.6.3　URI の組み立て（UriTemplate、UriComponents／UriComponentsBuilder）」を参照してください。

```
// 従来の Spring3.0 の URI Template の実装「UriTemplate」
UriTemplate uriTemplate = new UriTemplate("http://example.com/hotels/{hotel}/bookings/{booking}");
URI uri = uriTemplate.expand("42", "21");

// Spring3.1 の URI Template の実装「UriComponentsBuilder/UriComponents」
String hostname = "example.com";
UriComponents uriComponents = UriComponentsBuilder.newInstance()
                              .scheme("http").host(hostname)
                              .path("/hotels/{hotel}/")
                              .path("/bookings/{booking}").build()
                              .normalize()
                              .expand("42", "21")
                              .encode();
URI uri = uriComponents.toUri;
```

URI を部品に分けて組み立てることができるようになったため、接続先が動的に変わるようなときなど対処しやすくなった。

2. Spring MVC による開発

2.1. はじめに

Spring Framework の Core プロジェクトの Web フレームワーク「Spring MVC」について解説します。前提とする Spring Framework のバージョンは“3.0”または、”3.1”とし、バージョン“2.5”から導入されたアノテーションを使用した方法「@MVC」を基本とし説明していきます。

環境は次のものを使用しています。

- APP サーバ「Tomcat6」
- Java「JDK6」

2.2. Spring MVC の処理フロー

Spring MVC は、他の多くのフレームワーク(Struts、Click、Wicket)と同様、リクエストドリブンの「フロントコントローラパターン」を採用しています。フロントコントローラとは、１つのハンドラオブジェクトを介してリクエストを振り分け、リクエストに対して統一的に処理を記述できるようにするデザインパターンです。

ブラウザから送信されたリクエストは、図 2.1 に示すように、ブラウザから送信されたリクエストは、SpringMVC が提供する“DispatcherServlet”クラスが全て管理します。すなわち、“DispatcherServlet”がフロントコントローラになります。

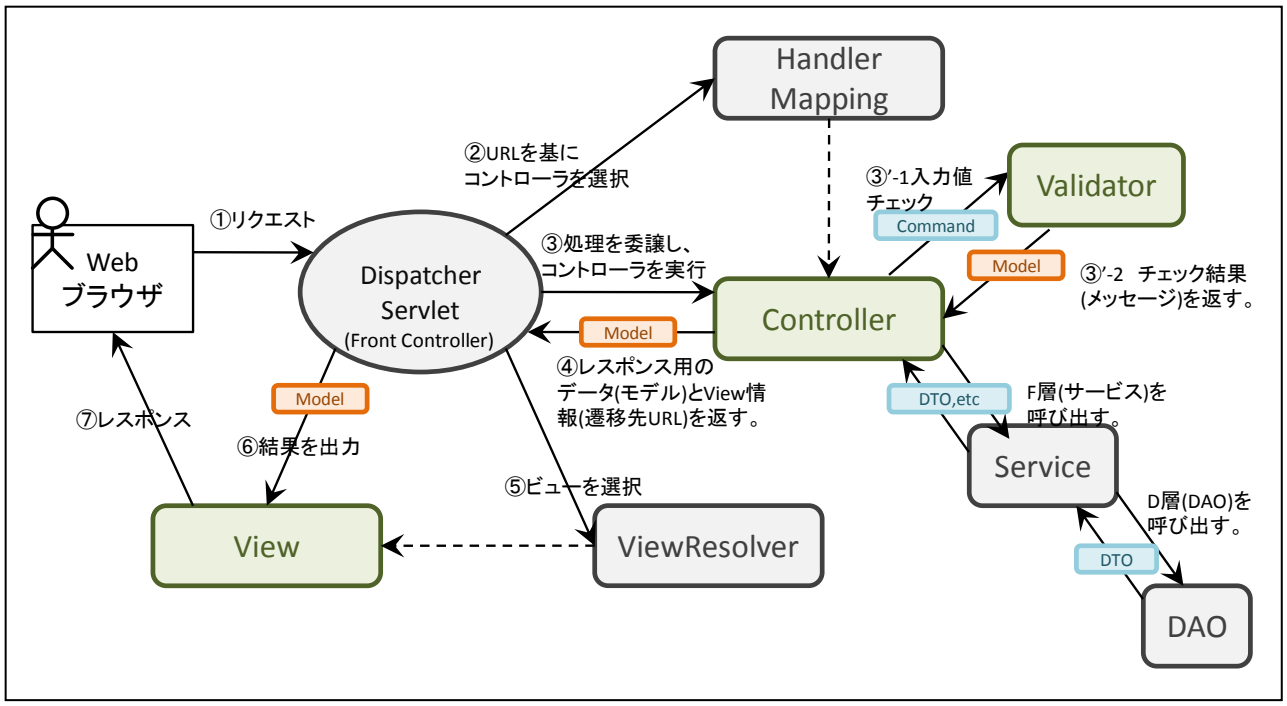


図 2.1　SpringMVC の処理フロー

表 2.1 SpringMVC のオブジェクト

No.	オブジェクト名称	説明	Struts との対応
1	DispatcherServlet	・HandlerMapping を介し、リクエスト(URL)に対して、Controller への振り分けを行う。 ・インスタンスはアプリケーションに 1 つのみ生成される。	ActionServlet
2	HandlerMapping	・リクエスト URL と Controller とのマッピングを管理する。 ・SpringMVC では、アノテーション「@RequestMapping」で定義する。	RequestProcessor
3	ViewResolver	・Controller が返したビュー名から遷移先となる View を決定する。	RequestProcessor
4	Model	・データを保持するオブジェクト。 ・Controller と View 間にて、データを受け渡すために使用する。 ・form の値を格納したものを Form オブジェクトと呼ばれる。 ・JavaBean の形式で抽出したものを、Spring では Command と呼ばれる。	ActionForm
5	View	・プレゼンテーション層への出力データを設定する。 ・実体は JSP、Velocity などの他に、JSON、Excel、PDF などがある。	JSP
6	Controller	・ビジネスロジックを呼び出し、処理の結果である Model や View を返す。	Action
7	Validator	・入力値チェックを行い、エラーメッセージを Model として返す。 ・アノテーションベースと独自実装の 2 種類がある。	Validator、ValidateForm
8	Command	・送受信した Model のデータを JavaBean に格納した形式のオブジェクト。	ActionForm

開発を行う際には、主に「View(JSP)」「Controller」「Validator」「Command」を作成していきます。

2.3. ファイル構成

Maven 形式のファイル構成を図 2.2 に示します。ファイル構成は、プロジェクトごとにこの限りではありません。

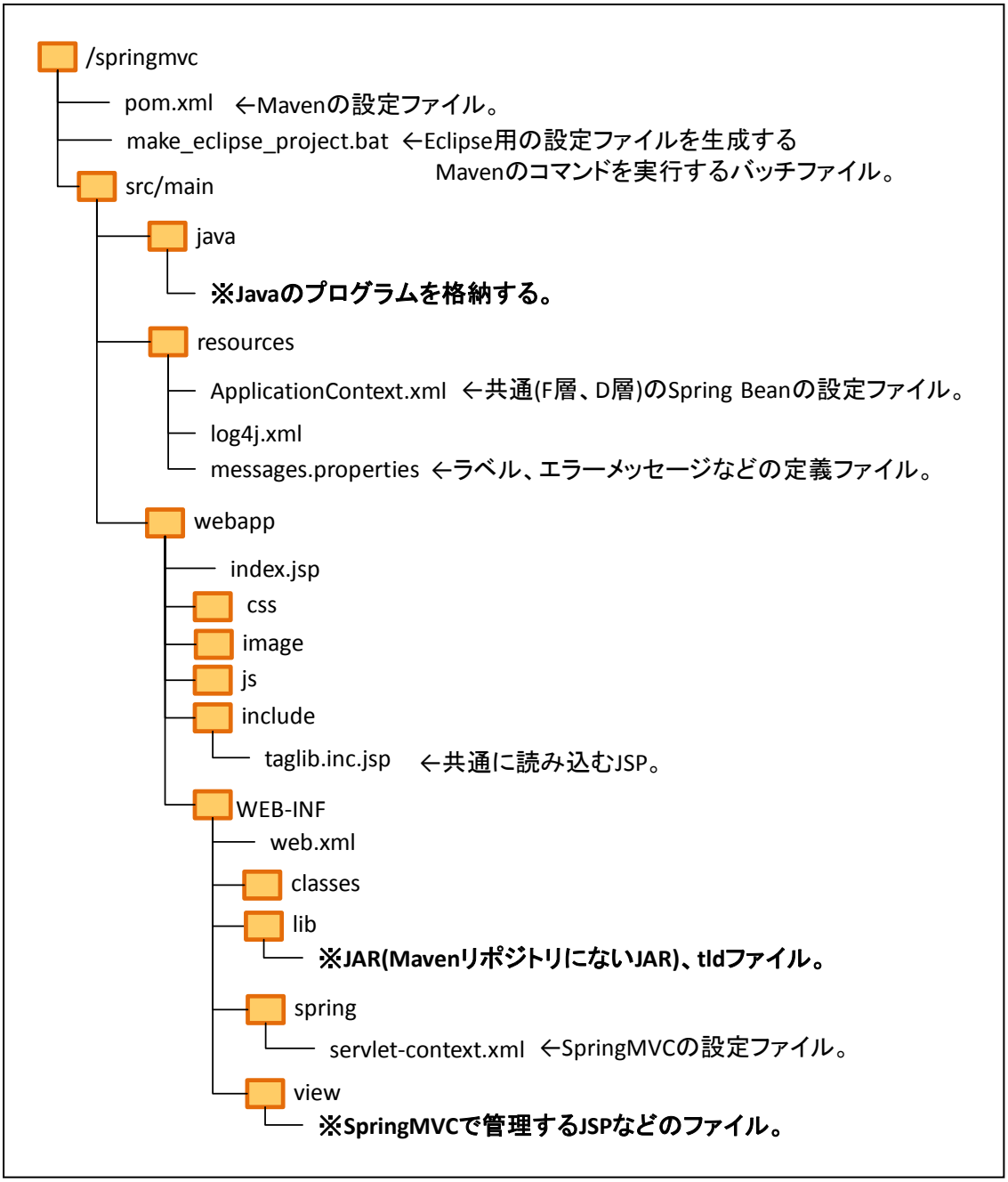


図 2.2　フォルダ構成

2.4. 設定ファイルの準備

2.4.1. pom.xml

2.4.1.1. SpringMVC の依存ファイル

①Web 開発に必要なライブラリ

```
<dependencies>
      <!-- Web Library -->
    <dependency>
          <groupId>javax.servlet</groupId>
          <artifactId>servlet-api</artifactId>
          <version>2.5</version>
          <scope>provided</scope>
    </dependency>
    <dependency>
          <groupId>javax.servlet.jsp</groupId>
          <artifactId>jsp-api</artifactId>
          <version>2.1</version>
          <scope>provided</scope>
    </dependency>
    <!-- Tag Library -->
    <dependency>
          <groupId>taglibs</groupId>
          <artifactId>standard</artifactId>
          <version>1.1.2</version>
    </dependency>
    <dependency>
          <groupId>javax.servlet</groupId>
          <artifactId>jstl</artifactId>
          <version>1.2</version>
    </dependency>
    <!-- Logger Library -->
    <dependency>
          <groupId>org.slf4j</groupId>
          <artifactId>jcl-over-slf4j</artifactId>
          <version>1.5.8</version>
    </dependency>
    <dependency>
          <groupId>org.slf4j</groupId>
          <artifactId>slf4j-api</artifactId>
          <version>1.5.8</version>
    </dependency>
    <dependency>
          <groupId>org.slf4j</groupId>
          <artifactId>slf4j-log4j12</artifactId>
          <version>1.4.2</version>
    </dependency>
    <dependency>
          <groupId>log4j</groupId>
          <artifactId>log4j</artifactId>
          <version>1.2.14</version>
    </dependency>
</dependencies>
```

Servler3.0 を使用したい場合、artifactId を「java.servlet-api」、version を「3.0」と設定します。

Servlet API。APP サーバで提供されるので、スコープを“provided”に設定します。

タグライブラリ。
JSP を記述する場合に必要になります。

ログ出力するためのライブラリ。
Log4j を SLF4J 経由で使用します。

図 2.3　Web 開発に必要なライブラリ

②Spring Framework のライブラリ

```
<dependencies>
    <!-- Spring Framework -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>${aspectJ.version}</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
        <version>${aspectJ.version}</version>
    </dependency>
    <dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib-nodep</artifactId>
        <version>2.2.2</version>
    </dependency>
</dependencies>

<properties>
    <spring.version>3.0.5.RELEASE</spring.version>
    <slf4j.version>1.5.6</slf4j.version>
    <aspectJ.version>1.6.2</aspectJ.version>
</properties>
```

Spring Framework のライブラリ。
Logger は、Log4j を使用する。

AOP のライブラリ。

Spring3.1 を使用したい場合、「3.1.2.RELEASE」と設定します。

図 2.4　Spring MVC の依存ファイル

③JSON 通信に必要なライブラリ

Spring MVC は、JSON 通信をする場合「Jackson(http://jackson.codehaus.org/)」を使用し、Java オブジェクトを JSON 形式に変換します。

```
<dependencies>
    <!-- JSON Library -->
    <dependency>
        <groupId>org.codehaus.jackson</groupId>
        <artifactId>jackson-core-asl</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>org.codehaus.jackson</groupId>
        <artifactId>jackson-mapper-asl</artifactId>
        <version>${jackson.version}</version>
    </dependency>
</dependencies>

<properties>
    <jackson.version>1.8.2</jackson.version>
</properties>
```

JSON ライブラリ。
必要なときのみ追加します。

図 2.5　JSON 通信に必要なライブラリ

2.4.1.2.　Maven リポジトリにない JAR ファイルの追加

Maven のリポジトリにない JAR ファイルや商用のライブラリは「/src/main/webapp/WEB-INF/lib/」以下に直接格納します。その場合 Maven のコマンド「eclipse:eclipse」を実行しただけでは、格納した JAR ファイルにクラスパスが設定されません。

そのため、pom.xml に「<systemPath>[JAR ファイルのパス]</systemPath>」に直接パスを設定します(図 2.6)。<systemPath>を指定した場合、「<groupId>、<artifactId>」は任意の値でかまいません。

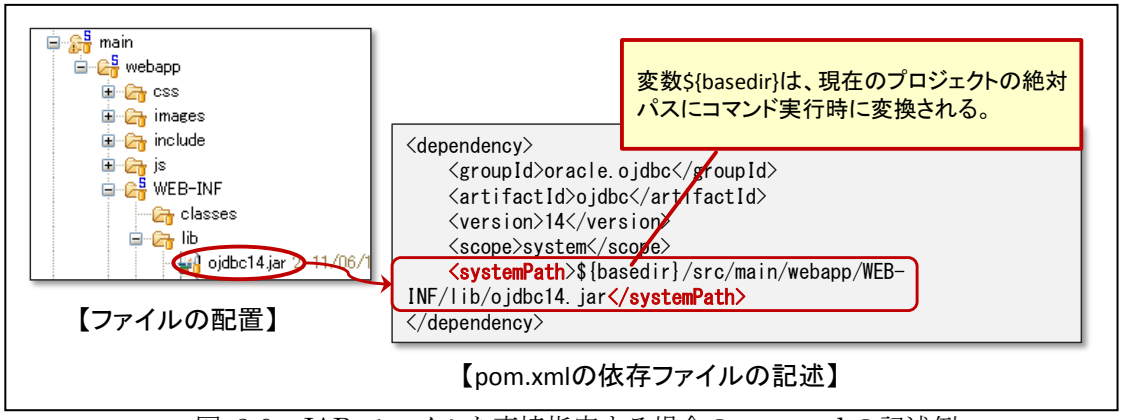


図 2.6　JAR ファイルを直接指定する場合の pom.xml の記述例

2.4.1.3.　War ファイル名の固定

パッケージを作成する Maven のコマンド「mvn package」を実行した際、デフォルトでは「[arfifactId]-[version].war」となる。War ファイルの場合バージョンが付加されていると、不便であるため、ファイル名を固定にする必要性がある。その場合、maven-war-plugin を使用し名前を設定することができる。

```
<project>
    <build>
        <pluginManagement>
            <plugins>
                <!-- war の名前を固定する -->
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-war-plugin</artifactId>
                    <configuration>
                        <warName>hoge</warName>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

パッケージを作成した際に、「hoge.war」となる。

図 2.7　war ファイルの名称を固定にする設定

2.4.1.4. Eclipse の設定ファイルの作成

Maven のコマンド「mvn eclipse:eclipse」を使用し、pom.xml から、Eclipse 用のプロジェクトファイルを作成します。オプションを付けたりするため、バッチファイル「make_eclipse_project.bat」等を用意しておくと便利です。

```
@echo off

%~d0
cd %~p0

call mvn eclipse:eclipse -DdownloadSources=true -Declipse.useProjectReferences=false
```

図 2.8 Eclipse のプロジェクトを作成するバッチファイル

【オプションの説明】

- 「downloadSources」

依存するライブラリの JAR の他にソースもダウンロードします。Eclipse のエディタ上で使用しているクラスに対するソースを参照することができます。

- 「eclipse.useProjectReferences」

依存している JAR ライブラリを Eclipse のプロジェクトとして読み込んでいる場合、Class Path を Maven のローカルリポジトリではなく、Eclipse のプロジェクトの方へ張ります。しかし、ただしくパスが張られないため、機能を無効にするため「false」を設定します。

pom.xml を変更した場合、必ずバッチファイルを再実行し、Eclipse 上でプロジェクトの更新（リロード）を行ってください。

2.4.2. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <!-- Encoder Filter -->
    <filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!-- Spring の共通 Bean の読み込み-->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/classes/ApplicationContext.xml</param-value>
    </context-param>

    <!-- Spring の Web スコープを利用するための設定 -->
    <listener>
        <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
    </listener>
    <!-- Spring MVC の Bean の読み込み -->
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/servlet-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
</web-app>
```

①文字のエンコードフィルタ。
SpringMVC 専用に用意されている。

②Spring の共通 Bean 設定ファイルの読み込み。

③Spring の Web スコープの利用設定

④SpringMVC の Bnean 設定ファイルの読み込み。

図 2.9 SpringMVC を利用するための“web.xml”のサンプル

表 2.2　SpringMVC を利用するための“web.xml”の説明

No.	説明
①	入力データを送信した際のエンコードを行います。
②	・Spring の ApplicationContext の設定ファイルを読み込みます。 ・Spring の F 層、D 層の設定がされているファイルです。 ・Servlet 内で直接呼び出すときには、「WebApplicationContext appContext = WebApplicationContextUtils.getWebApplicationContext(servletContext);」とします。
③	Spring Bean の Web スコープ(request、session、global session)を使用できるようにします。 詳細は、「8.3.3　SpringBean を使用したセッション管理」を参照してください。
④	・SpringMVC の DispacheServlet の設定を行います。 ・SpringMVC 用の Spring の Bean ファイルを読み込みます。 読み込みパスを設定しない場合、デフォルトでは WEB-INF フォルダ以下の「XXX-context.xml」を読み込みます。 ・関連付ける URL のパターンは、サンプルのように「*.html」と指定することで、利用者には静的な HTML のファイルのアクセスのように見せかけ、アプリケーション内部の仕組みを利用者から隠蔽することができます。 ・必ず②の設定の後に記述します。DispacheServlet は、②のオブジェクトが必要になります。 SpringMVC の記述は②のファイルに定義してもかまいませんが、P 層と F・D 層と機能を分けるために別定義します。

2.4.3. アプリケーション用(共通の)Spring Bean ファイル

Spring Bean アプリケーション用のファイルは、通常 classpath のトップに作成します。例えば、「/src/main/resources/ApplicationContext.xml」に配置します。SpringBean の登録は XML でもかまいませんが、ここでは Spring MVC に合わせ、アノテーションによる依存関係の自動設定(Autowiring)で記述する方法を紹介します。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
    ">

    <!-- Autowired の Scan 有効定義 -->
    <context:annotation-config />
    <!-- 設定値 -->
    <bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:database.properties</value>
                <value>classpath:app.properties</value>
            </list>
        </property>
    </bean>
    <!-- 共通のメッセージファイル -->
    <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>classpath:message/message</value>
                <value>classpath:message/label</value>
            </list>
        </property>
    </bean>

    <!-- Scan 対象のパッケージの定義
        Scan 対象のアノテーション：@Component, @Repository, @Service, or @Controller. -->
    <context:component-scan base-package="sample.core.service" />
    <context:component-scan base-package="sample.core.component " />
</beans>
```

①名前空間“context”を利用する際の定義。

②Autowire 機能の有効化の宣言。

③設定値を外部化する。
SpringBean の中では、${キー}でアクセス可能。

④共通メッセージの定義。
JSP のカスタムタグ<spring:message>により呼び出すことができる。

⑤Autowire に自動設定の対象となるパッケージ名の定義。

図 2.10　共通の SpringBean の設定ファイル“ApplicationContext.xml”のサンプル

表 2.3　共通の SpringBean の設定ファイル“ApplicationContext.xml”の説明

No.	説明
①	・XML ファイル内で名前空間<context:XXX>を使用可能にするための定義です。 ・XMLSchema の定義に沿って記述します。
②	・アノテーションによる依存関係の自動的に設定する Autowiring 機能を有効にします。 ・必ず④の記述よりも前に定義します。
③	・Spring Bean の設定ファイルの中で参照可能な設定値をプロパティファイルで定義です。 ・プロパティファイルの値は、${プロパティキー}で参照可能です。 ・通常は、運用によって変わる可能性がある値を定義します。 例)DB の設定値、APP の設定値、メールサーバの設定値など。
④	・共通のメッセージを定義します。 ・この定義により、Spring では、MessageSource クラス経由でプロパティの値を取得できます。他に、SpringMVC では、Spring 専用の JSP のカスタムタグ<spring:message>により、値を呼び出すことができる(12.1 節参照)。
⑤	・Autowiring 機能が有効なパッケージを定義します。 ・アノテーションを定義したクラスをこのパッケージに格納する必要があります。 ・複数定義できます。 ・Spring が生成するインスタンス(今までは<bean>タグで定義していた)をステレオタイプと呼びます。ステレオタイプとして有効なアノテーションは、「@Aspect(AOP の部品)」「@Component(部品)」「@Service(F 層用)」「@Repository(D 層用)」「@Controller(P 層用)」です。 ・共通の設定ファイルなので、通常は@Controller 以外のアノテーション設定されたクラスが格納されているパッケージを対象とします。

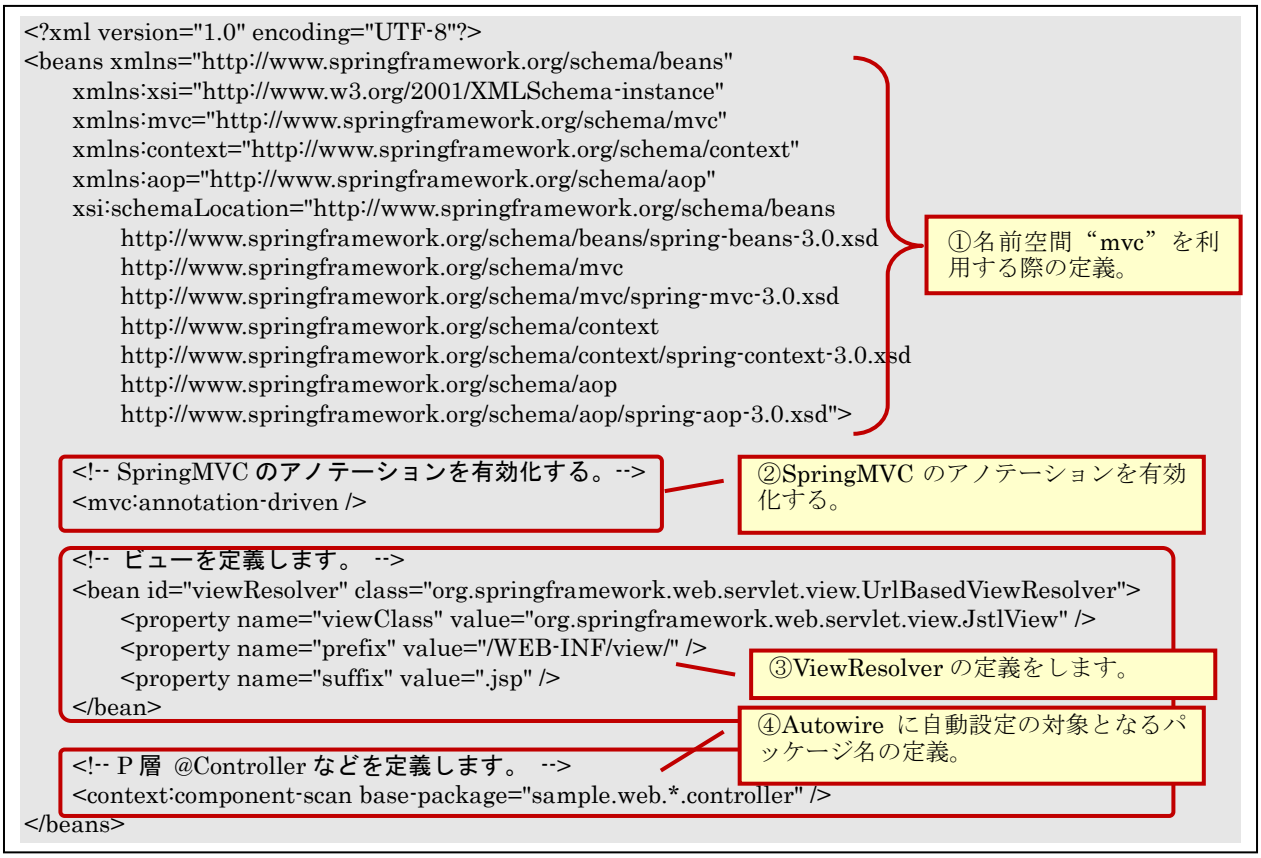
Autowiring の場合、テストで使用するスタブなど対象外としたい場合があります。図 2.11 に示すようにフィルタリングすることができます。

```
<beans>

    <context:component-scan base-package="org.example">
        <context:include-filter type="regex" expression=".*Stub.*Repository"/>
        <context:exclude-filter type="annotation"
                                expression="org.springframework.stereotype.Repository"/>
    </context:component-scan>

</beans>
```

図 2.11　Autowring の対象パッケージのフィルタ設定

2.4.4. Spring MVC 用の設定ファイル

SpringMVC の設定ファイルは、“/WEB-INF/”以下に作成します。例えば、「/src/main/webapp/WEB-INF/spring/servlet-context.xml」に配置します。このファイルには、SpringMVC でのみ利用する設定を記述していきます。




```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:mvc="http://www.springframework.org/schema/mvc"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/mvc
            http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

    <!-- SpringMVC のアノテーションを有効化する。-->
    <mvc:annotation-driven />

    <!-- ビューを定義します。 -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
          <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
          <property name="prefix" value="/WEB-INF/view/" />
          <property name="suffix" value=".jsp" />
    </bean>

    <!-- P 層 @Controller などを定義します。 -->
    <context:component-scan base-package="sample.web.*.controller" />
</beans>
```

図 2.12　SpringMVC の設定ファイル“servlet-context.xml”のサンプル

表 2.4　SpringMVC の設定ファイル“servlet-context.xml”の説明

No.	説明
①	・XML ファイル内で名前空間<mvc:XXX>を使用可能にするための定義です。 ・XMLSchema の定義に沿って記述します。
②	・SpringMVC の設定をアノテーションを使用し指定するということを宣言します。 ・必ず④の記述よりも前に定義します。
③	・View の設定を行います(3.5 節を参照)。
④	・Autowiring 機能が有効なパッケージを定義します。 ・アノテーションを定義したクラスをこのパッケージに格納する必要があります。

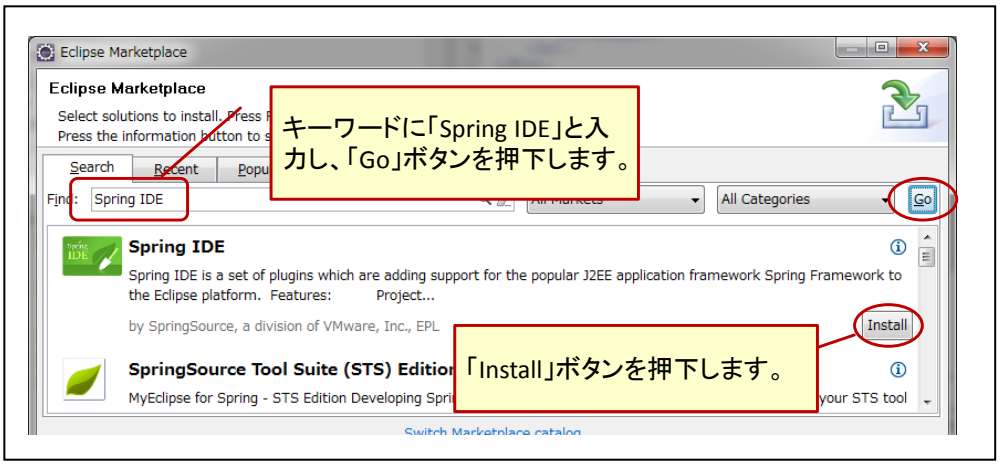
2.4.5. Eclipse のプラグイン SpringIDE

Spring IDE をインストールすると、アノテーションが正しく適用されているか、または AOP が正しく適用されているかどうか、ビジュアル化され設定ミスを防ぐことができます。

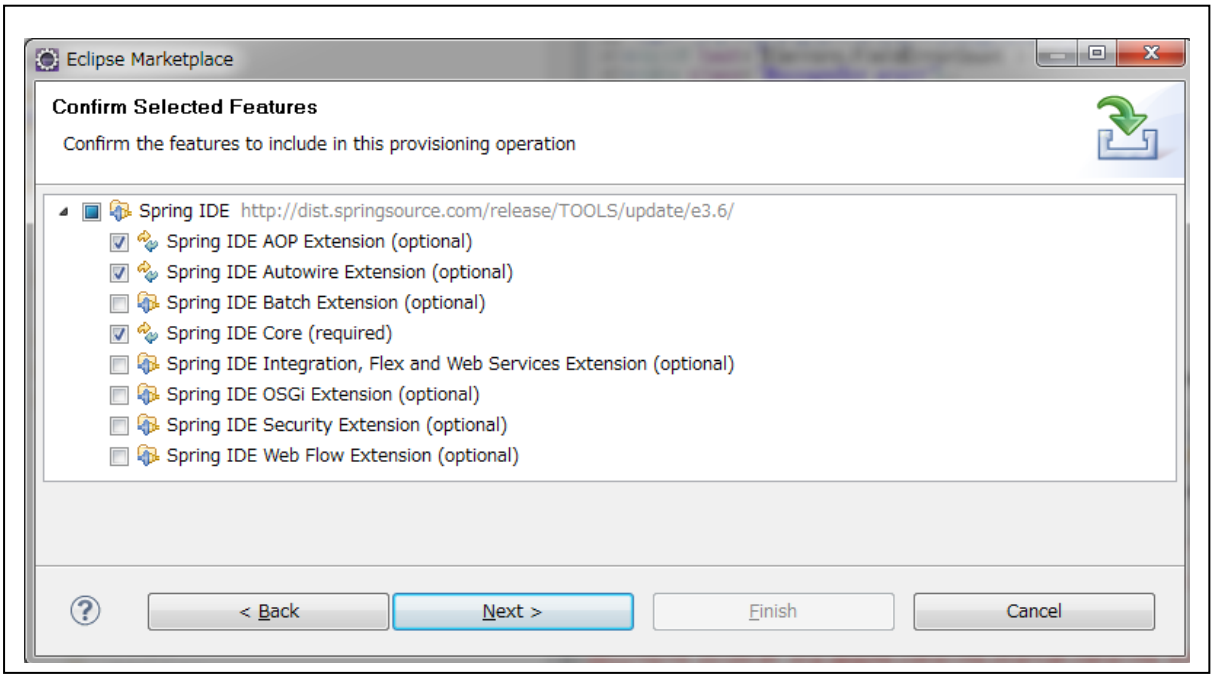
2.4.5.1. Spring IDE のインストール

Eclipse3.6 以降は、Marketplace が導入されたので、そちらからインストールします。Eclipse のメニュー「Help」－「Eclipse Marketplace」を選択します。

(1) 検索キーワードに「Spring IDE」と入力し、「Go」ボタンを押下します。
その後、Spring IDE の項目の「Install」ボタンを押下します。（画面は、Eclipse3.6 のものです）。



(2) プラグインを選択します。基本的に「Core」「AOP Extension」「Autowire Extension」で十分です。
必要ならば全て選択してください。その後、「Next」を押下し、インストールします。

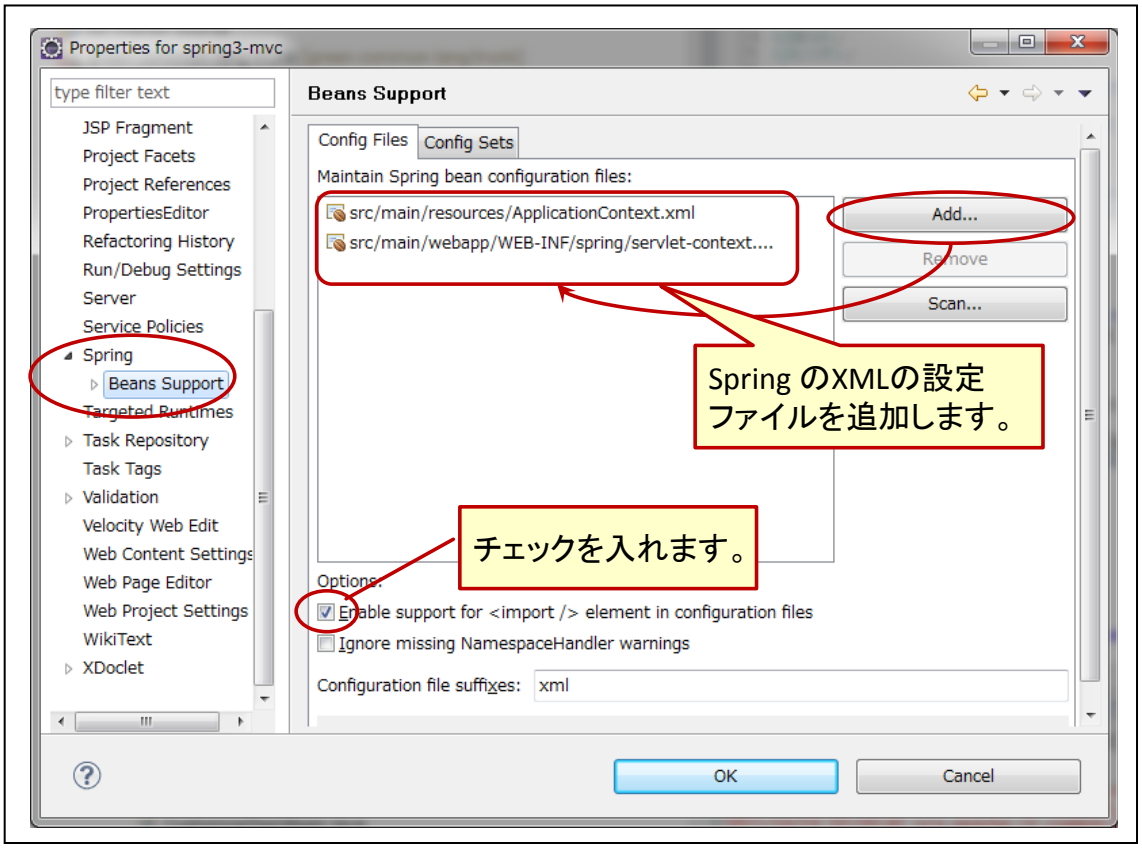


2.4.5.2. Spring IDE の設定

(1) Spring IDE を適用するプロジェクト上で右クリックし、メニュー「Spring Tools」－「Add Srping Project Nature」を選択します。正しく選択されると、プロジェクトのアイコン上に「S」が表示されます。

(2) 再び、プロジェクト上で右クリックし、メニュー「Properties」を選択し、プロパティを開きます。

(3) 左ペインの「Spring」－「Beans Support」を選択します。
その後、タブ「Config Files」を選択し、「Add」ボタンで、Spring の設定ファイルを追加します。
追加するのは、「ApplicationContext.xml」と SpringMV 用の「servlet-context.xml」です。
また、チェックボックス「Enable support for <import/> element in configuration files」を選択します。



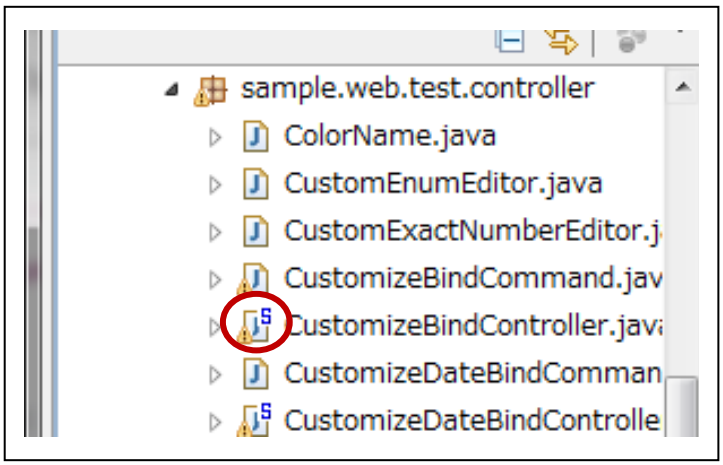
(4) 設定完了後、「OK」ボタンを押下します。

> Spring IDE は動作が重いため、環境によっては PC の動作を遅くします。その場合は、Spring IDE による設定の確認をが終わったら、プラグインの適用を外すことをお勧めします。
>
> プロジェクト上で右クリックし、メニュー「Spring Tools」－「Remove Srping Project Nature」を選択すると設定が外れます。

2.4.5.3.　Spring IDE を利用した設定ファイルが正しく適用されているかどうかの確認

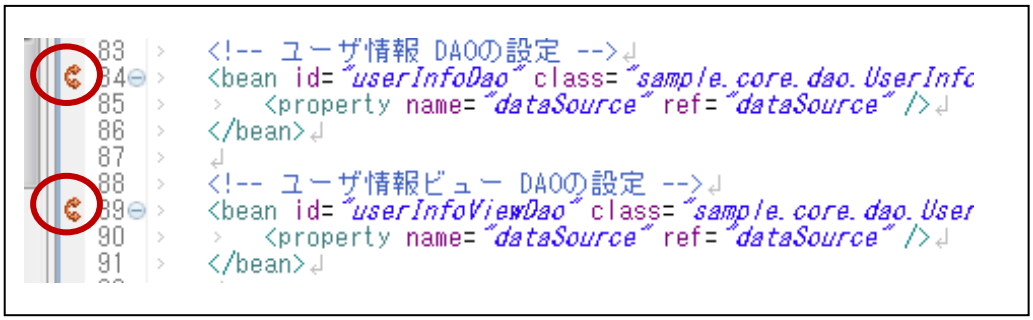
【Spring Beans の確認】

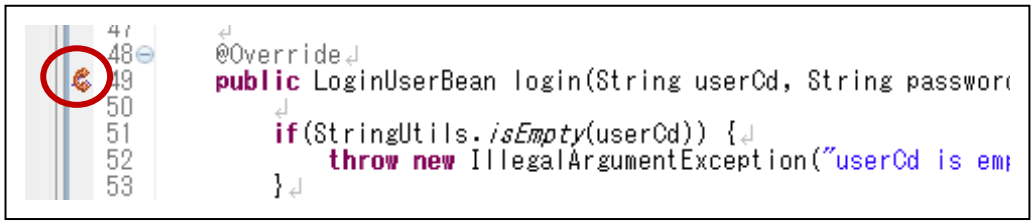
- Spring Bean として登録しているものには、ソース上「S」マークがアイコンに付与されます。
  アノテーション「@Service」「@Component」「@Controller」「@Configuration」などで登録しているソースも含みます。



【AOP の適用先の確認】

- トランザクション用の AOP などを使用している場合、適用先の Spring Bean（設定ファイル）、メソッドのソース上に「⇔」が表示されます。





3. コントローラの作成

3.1. 簡単なコントローラの作成

単純なコントローラの例を図 3.1 に示します。これは、メッセージを次の画面に渡し画面遷移し、受け取ったメッセージの文字列を JSP で表示するものです。基本的に、SpringMVC のアノテーションを使用した「@MVC」の形式を使用します。

```
import java.util.Date;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

/**
 * 単純なコントローラ。
 */
@Controller
public class HelloWorldController {

    @RequestMapping("/hello")
    public ModelAndView helloWorld() {

        ModelAndView mav = new ModelAndView("/helloView");

        // 直接モデルに、メッセージを設定する。
        mav.addObject("message1", " Hello World, <strong>Srping MVC 3.0!</strong> ");

        // モデルを取得して、メッセージを設定する。
        mav.getModelMap().put("message2", "メッセージ 2。");

        mav.addObject("currentDate", new Date());

        return mav;
    }
}
```

モデル（値を保持するクラス）の作成。

図 3.1　簡単なコントローラ

【コントローラの定義】

- コントローラのクラスには、「@Controller」を付与します。これにより、このクラスは“コントローラ”として定義されていることを Spring として定義したことになります。
- リクエストを受け取る(URL にアクセスした際に処理を委譲される)メソッドに対して、「@RequestMapping」を付与します。引数には、アクセスする際の URL を定義します。定義方法により様々な処理を実現できます。詳細は「3.2　@RequestMapping による様々な URL の処理」を参照してください。

- この例では、「http://<サーバ名>:8080/<APP 名>/hello.html」とブラウザからアクセスした際の処理を定義します。拡張子「.html」は自動に付与され、この設定は web.xml で定義します。詳細は、「2.4.2 web.xml」を参照してください。

【リクエストの処理】

- リクエストを処理するメソッドは、通常は戻り値として「ModelAndView」を使用します。このクラスは、遷移先に渡すデータを保持する（Model）と遷移先（View）を設定できます。戻り値として、View のみ、Model のみなど設定できますが、コーディングの統一（規約）として画面遷移を伴う処理には、ModelAndView を使用することをお勧めします。また、戻り値として他には、JSON 形式など様々なデータ型を取ることができ、詳細は「3.2　コントローラの引数と戻り値」を参照してください。
- 遷移先の URL は、ModelAndView のコンストラクタに渡すか、または、「ModelAndView#setViewName()」で設定します。ここの例では、「/helloView.jsp」に画面遷移します。注意点として、“/WEB-INF/view/”以下のリソースには、ブラウザからは直接アクセスはできません。必ず、View オブジェクト経由で表示することになります。
- 遷移先の画面にデータを渡す場合、「ModelAndView#addObject(“キー名”, データ)」を使用し渡します。基本的に、Model のデータは“リクエストスコープ”として処理され、次の画面まで有効となります。
- Model は Java のクラス「java.util.Map」形式になっており、「ModelAndView#getModelMapMap()」で、直接 Model を操作できたりします。

3.1.1. 簡単な JSP の定義

遷移先の画面の JSP を定義します。Controller で Model に設定したメッセージなどを出力します。

```
<%@ page language="java" trimDirectiveWhitespaces="true" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%-- ▼ taglib --%>
<%@ include file="/include/taglibs.inc.jsp"%>
<%-- ▲ taglib --%>
<html>
<head>
<title>Spring 3.0 MVC：Hello World</title>
</head>
<body>
<p>モデルの値の呼び出し。</p>
<ul>
    <li>message1=${message1}</li>
    <li>message1(カスタムタグから)=<c:out value="${message1}" /></li>
    <li>message2=${message2}</li>
    <li>currentDate(フォーマットする)=<fmt:formatDate value="${currentDate}" pattern="yyyy 年 MM 月 dd 日" /></li>
</ul>
</body>
</html>
```

図 3.2　簡単な JSP の定義「helloView.jsp」

【JSP の定義】

- ページディレクティブに「trimDirectiveWhitespace=”true”」とすると、JSP タグの定義などによる余分な空白を自動削除されます。これは、JSP2.1（Tomcat では ver6 以降）から追加された機能です。
- 同様にページディレクディブに「contentType=”text/html; charset=UTF-8”」を定義することで、HTTP の Content-Type ヘッダーに指定され、<meta>タグによる指定をしなくてもよくなります。Include する JSP の場合、必要ありません。
- JSP ファイルの文字コードとして、「pageEncoding=”UTF=8”」を定義します。インクルードする文字コードは必ず定義します。
- カスタムタグなど共通の定義が記述している JSP を<%@include%>で読み込みます。カスタムタグなどを使用している場合など便利です。

```
<%@ page language="java" trimDirectiveWhitespaces="true" pageEncoding="UTF-8"%>

<%-- JSTL の定義 --%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>

<%-- Spring のカスタムタグの定義 --%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
```

図 3.3　共通で読み込む JSP「taglibs.inc.jsp」

【モデルに格納したデータの出力】

- モデルに格納したデータは、「${キー名}」で参照します。これは、JSP2.0（Tomcat では ver5.5 以降）から導入された「EL 式」と呼ばれる形式です（詳細は、「6　EL 式(Expression Language)」を参照。）。また、カスタムタグの属性値の中でも使用可能ですが、TLD の定義で<rtexprepvalue>true</rtexprepvalue>となっている場合に限ります。また、${...}で参照する場合、HTML エスケープされません。
- エスケープした状態で出力する場合には、JSTL の<c:out>を使用します。
- Date 型などオブジェクトの場合、JSTL の<fmt:formatDate>を使用し、書式を指定し使用します。

3.1.2. ファイルの配置

SpringMVC で管理する JSP ファイルの配置場所は決まっています（図 3.4）。SpringMVC 経由でアクセスする JSP（View）ファイルは、「WEB-INF/view/」以下に配置します。配置する場所は変更することができ、「2.4.4　Spring MVC 用の設定ファイル」で示したように、「servlet-context.xml」で定義します。

- WEB-INF 以下以外の include フォルダなどは、通常のファイルとしてアクセス可能です。
- 「WEB-INF/view/」以下のファイル構成と、コントローラ内で指定した画面遷移の URL の構造は一致している必要があります。今回の例では、コントローラで「new ModelAndView(“/helloView”)」遷移先を定義しているので、「WEB-INF/view/helloView.jsp」に JSP ファイルを作成します。

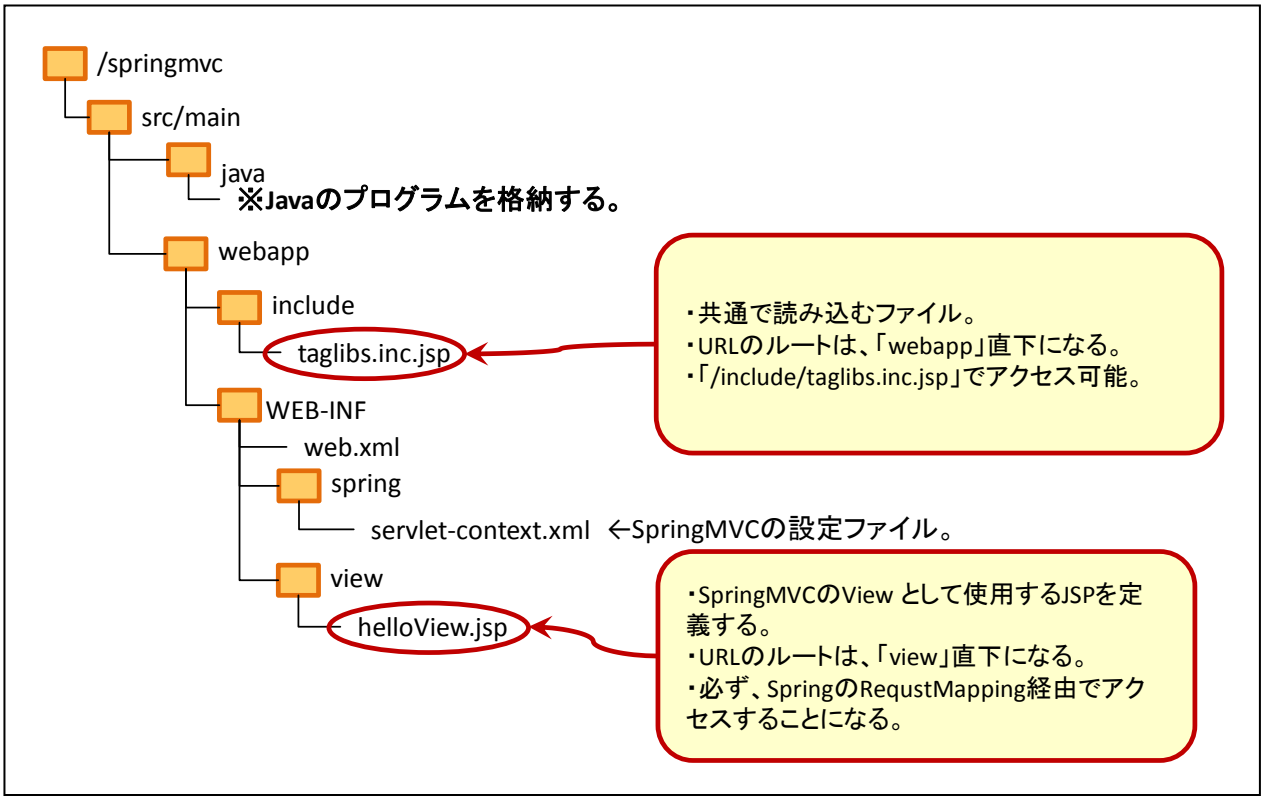


図 3.4　JSP ファイルの配置

3.1.3. Web ブラウザからアクセスする

作成したプログラムをブラウザからアクセスする場合、「http://<サーバ名>:8080/<APP 名>/hello.html」からアクセスします。

図 3.5　Web ブラウザでアクセスした際の表示例

【SpringMVC での処理フロー】

作成したプログラムがどのように SpringMVC で処理されるかを説明したものを次に示します。

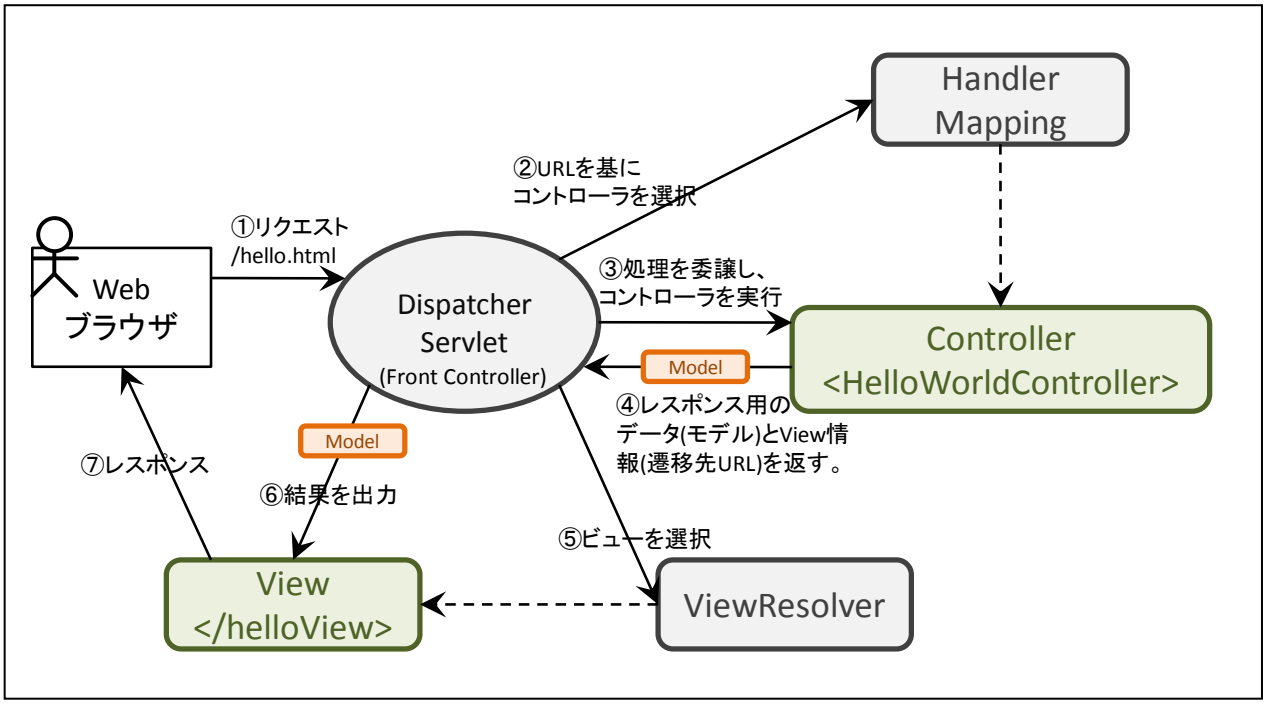


図 3.6　作成したプログラムの SpringMVC での処理フロー

3.2. コントローラの引数と戻り値

アノテーション「@RequestMapping」が付与されたメソッドは「リクエストを処理する」メソッドとして呼び出されます。Spring MVC の場合、このメソッドの引数は非常に柔軟に設定でき、また種類も多数あるため、場面によって組合せて利用します。プロジェクトで利用する際には、場面ごとにコーディング規約として統一した組み合わせで使用することをお勧めします。

3.2.1. コントローラの引数一覧

コントローラのメソッドとして設定可能な引数の一覧を表 3.1 に示します。

表 3.1　コントローラの引数一覧

No.	Java 型	説明	I/O
1	ServletRequest /HttpServletRequest	Servlet API のリクエスト。 通常は、HttpServletRequest を利用します。	I
2	ServletResponse /HttpServletResponse	Servlet API のレスポンス。 通常は、HttpServletResponse を利用します。	O
3	HttpSession	Serlvet API のセッション。NULL になることはなく、Spring MVC 側でセッションが生成される。	I/O
4	org.springframework.web.context.request.WebRequest /org.springframework.web.context.request.NativeWebRequest	Session、Request 情報など取得／設定する際に、Servlet API の代わりに利用します。 スコープを指定して#getAttribute()、#setAttribute()など操作できます。	I/O
5	java.util.Locale	現在のロケール情報を取得できます。 「LocaleResolver」で環境の変更を行うことができます。	I
6	java.io.InputStream /java.io.Reader	リクエストされたコンテンツの情報の入力ストリームで、Servlet API から取得した値です。	I
7	java.io.OutputStream /java.io.Writer	レスポンスするコンテンツの情報を出力ストリームで、ServletAPI から取得した値です。	O
8	java.security.Principal	現在の認証済みのユーザ情報が格納されます。 通常は使用しません。	I
9	@PathVariable が付加された変数	モデルに格納されたリクエストのパラメータを抽出したものです。 ・名前がない場合、「@RequestParamString userId」 ・名前付きの場合。複数パラメータがある場合は、名前付きを使用します。「@RequestParam("userId")	I

		String userId」	
10	@RequestHeader が付加された変数	特定の Request HTTP Header を取得します。	I
11	@RequestBody が付加された変数	HTTP のリクエスト本体を取得します。 JSON や XML 形式のデータ受信する場合に、引数の変数に付与します。HttpMessageConverters を使用し自動的に JavaObject に変換されます。	I
13	HttpEntity<T>	HTTP のヘッダーを直接指定する場合に使用します。 メソッドの中に HttpEntity<T>を自動的に注入することができます。注入する際には、HttpMessageConverters を使用し値を変換します。	I
14	java.util.Map /org.springframework.ui.Model /org.springframework.ui.ModelMap	View に渡す暗黙的なモデルを取得します。 戻り値が ModelAndView を使用しない、View(or String)の場合のときに使用します。	O
15	@ModelAttribute が付加された変数（※1）	Form の値を JavaBean にバインドした変数に付与します。Spring MVC ではコマンド（Command）と呼ばれています。 ・@InitBinder を付加したメソッドにより、バインド方法を細かく定義することができます。 ・@ModelAttribute で名前を設定しない場合、Command の初期値名は、「command」となります。 ・クラスに@SessionAttributes を付与すると、Form の値（ModelAttribute）の値を、セッションスコープで保持することができます。	I
16	org.springframework.validation.Errors /org.springframework.validation.BindingResult（※1）	Form オブジェクトの validation 結果が格納されます。 バインディングに失敗した場合のエラーメッセージなどが格納されます。	I
17	org.springframework.web.bind.support.SessionStatus	セッション情報をクリアするなど補助的なことを行うクラスです。 @SessionAttriutes で定義されている名前を持つセッション上の情報を、「SessionStatus#setComplete()」により一括でクリアします。	I

※1　BindResult と@ModelAttribute の両方を記述する場合は、順番として「@ModelAttribute, BindResult」と連続して記述します。

Model も含む場合は、○：「@ModelAttribute, BindResult, Model」、×：「@ModelAttribute, Model, BindResult」とします。

3.2.2. コントローラの戻り値の一覧

表 3.2　コントローラの戻り値の一覧

No.	Java 型	説明
1	ModelAndView	View で指定した URL に、Model(データ)を渡す際に利用します。
2	Model	View を暗黙的に決めて、Model を設定します。 View の URL は RequestToViewNameTranslator で決まり、通常は現在の URL と変わりません。
3	java.util.Map	Model と同じです。 ModelMap は、Map を実装しているため、ModelMap のインスタンスを返しても問題ありません。
4	View	View で指定した URL に遷移します。 Model は暗黙的に決まり、引数に Model、@ModelAttribute がある場合は、その値を Model とします。 View の実装クラスには様々なものが用意されており、種類により JSON 型や PDF、Excel など様々なタイプを View として扱うことができます。 詳細は、「3.5　ViewResolver」を参照してください。
5	String	View の URL を直接記述します。View と同様です。 Model は暗黙的に、View の場合と同様に決まります。
6	void	自身にレスポンスを返します。View を省略した場合と同様に URL は RequestToViewNameTranslator により決まります。
7	void (引数に@ResponseBody がある場合)	引数で指定した ResponseBody を返します。HttpMessageConverter で値が変換されます。JSON、XML など通常の JSP 以外を返す場合に利用します。
8	メソッドに @ResponseBody が付加されている場合	戻り値を HTTP のレスポンスのボディとして返します。値は、HttpMessageConverter により変換されます。 JSON 型を返す場合などに利用します。
9	HttpEntity<T> /@ResponseEntity<?>	Servlet の HTTP ヘッダーに直接値を設定します。引数で RequestEntity がある場合、HttpMessageConverter で値が変換されます。
10	任意のオブジェクト	値を 1 つだけ設定した Model として扱います。 引数に@ModelAttribute が設定されており、戻り値とクラス型が一致する場合、@ModelAttribute の Model として返します。

3.2.3. よくある処理ごとの引数と戻り値の組合せ

3.2.3.1. View(JSP)に遷移を行う場合

単に、View(JSP)に遷移する場合。

- 引数はありません。
- 戻り値として、String 型を指定し、View のパスを直接指定します。
  拡張子は必要ありません。
- JSP の格納先が「/WEB-INF/view/commons/sample.jsp」ならば、「/commons/sample」のようにディレクトリも指定します。

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public String sample() {

        return "/toView";
    }
}
```

3.2.3.2. View(JSP)に遷移を行う場合（Model あり）

View(JSP)に Model を渡す場合。

- 引数はありません。
- 戻り値として、ModelAndView を使用します。

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public ModelAndView sample() {

        // View を設定する
        ModelAndView mav = new ModelAndView("/toView");

        // モデルを取得して、メッセージを設定する。
        mav.addObject("message1", "メッセージ 1。");

        return mav;
    }
}
```

3.2.3.3.　他の URL に転送を行う場合

他の URL に転送を行う場合、forward、redirect を使用します。

- 戻り値の View の URL に対して、接頭語「forward:」または、「redirect:」を付加します。
- Servlet API の forward と同様、転送元のリクエストが POST メソッドで実行されている場合、転送先も POST メソッドになります。
- 詳細は、「3.4　URL への転送方法」を参照してください。

```
@Controller
public class SampleController {
	// フォワードする
	@RequestMapping("/sample1")
	public String forward() {
		return "forward:/hello.html";
	}

	// リダイレクトする
	@RequestMapping("/sample2")
	public String refirect() {
		return "redirect:/hello.html";
	}
}
```

3.2.3.4.　URL のクエリストリングから値を受け取る場合

URL のクエリストリングでパラメータを指定した場合。

- 引数として、クエリストリングのパラメータを「@RequestParam」で指定した値を取得します。
- @ModelAttribute がなく、@RequestParam のみの場合、引数として、BindingResult を指定することはできません。
- 戻り値として、ModelAndView を使用します。

```
@Controller
public class SampleController {

	@RequestMapping(value="/sample", method={RequestMethod.GET})
	public ModelAndView sample(@RequestParam(value="userCd", required=true) Integer id, @RequestParam
String token) {

		// バインド時のエラー処置
		if(bindingResult.hasErrors()) {
			・・・省略
		}

		ModelAndView mav = new ModelAndView("/toView");
		return mav;
	}
}
```

3.2.3.5. Form から値を取得する場合

Form から POST されたデータを処理する場合。

- 引数として、POST された値を JavaBean にマッピングする先を「@ModelAttribute」で指定した値を指定します。
- 引数として、BindingResult を指定します。パラメータを受け取る際は、必ず設定します。
- 戻り値として、ModelAndView を使用します。

```
@Controller
public class SampleController {

    @RequestMapping(value="/sample", method={RequestMethod.POST})
    public ModelAndView sample(@ModelAttribute LoginCommand command, BindingResult bindingResult) {

        // バインド時のエラー処置
        if(bindingResult.hasErrors()) {
            ・・・省略
        }
        ModelAndView mav = new ModelAndView("/toView");
        return mav;
    }
}
```

3.2.3.6. Form から値を取得し、セッションに格納する場合

Form から POST されたデータを処理し、セッションに格納する場合。

- 引数として、WebRequest を指定します。「WebRequest#setAttribute()」を使用し、セッションにデータを格納します。
- 引数として、POST された値を JavaBean にマッピングする先として「@ModelAttribute」で指定した指定します。マッピング先の名称はデフォルトでは変数名となります。
- 引数として、BindingResult を指定します。@ModelAttribute でパラメータを受け取る際は、必ず設定します。
- 戻り値として、ModelAndView を使用します。

```
@Controller
public class SampleController {

    @RequestMapping(value="/sample", method={RequestMethod.POST})
    public ModelAndView sample(WebRequest request, @ModelAttribute LoginCommand command, BindingResult
bindingResult) {
        // バインド時のエラー処置
        if(bindingResult.hasErrors()) {
            ・・・省略
        }
        // セッションにデータを格納する
        request.setAttribute("loginUser", "hogehoge", RequestAttributes.SCOPE_SESSION);

        ModelAndView mav = new ModelAndView("/toView");
        return mav;
    }
}
```

3.2.3.7.　ファイルをダウンロードする場合（HttpServletResponse を使用する場合）

帳票などを作成し、それをレスポンスに書き込みダウンロードさせる場合。

- 引数として、**HttpServletResponse** を指定します。
  ServletAPI を直接使用するため、Content-Type など指定可能です。
- 戻り値は、void で必要ありません。

```
@Controller
public class DownloadController {

    // HttpServletResponse を使用する場合
    @RequestMapping(value="/downloadFile1", method = {RequestMethod.GET} )
    public void downloadFile1(HttpServletResponse response) throws IOException {

        // HTTP ヘッダーの指定
        response.setContentType("application/octet-stream");
        response.setHeader("Content-Disposition", String.format("filename=¥"%s¥"", "sample.txt"));

        PrintWriter writer = response.getWriter();
        writer.append("HelloWorld");
    }
}
```

3.2.3.8.　ファイルダウンロードする場合（ResponseEntity を使用する場合）

HttpServletResponse と同様に、HTTP ヘッダーを細かく指定することができます。直接 ServletAPI に依存しない方式を取る場合は、こちらの方法を利用します。

- 戻り値として、**ResponseEntity** を使用します。
  - ➢　レスポンスに書き込むデータ型を、Generics で指定します。
- メディアタイプなどの HTTP ヘッダーを、「HttpHeaders」を使用して指定します。

```
@Controller
public class DownloadController {

    // ResponseEntity を使用する場合
    @RequestMapping(value="/downloadFile2", method = {RequestMethod.GET} )
    public ResponseEntity<String> downloadFile2() throws IOException {

        // HTTP ヘッダーの指定
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(new MediaType("application", "octet-stream"));
        headers.set("Content-Disposition", String.format("filename=¥"%s¥"", "sample.txt"));

        String data = "HelloWorld!";
        return new ResponseEntity<String>(data, headers, HttpStatus.OK);

    }
}
```

3.2.3.9.　ファイルをダウンロードする場合（Writer/OutputStream を使用する場合）

- 引数として、Writer または、OutputStream を指定します。これは、HttpServletResponse#getWriter/getOutputStream から取得したものになります。
  ContentType や Header など指定できないため、「3.2.3.7　ファイルをダウンロードする場合（HttpServletResponse を使用する場合）」「3.2.3.8　ファイルダウンロードする場合（ResponseEntity を使用する場合）」を使用するのが一般的です。
- 戻り値は、void で必要ありません。

```
@Controller
public class DownloadController {
	// Writer を使用する場合
	@RequestMapping(value="/downloadFile3", method = {RequestMethod.GET} )
	public void downloadFile3(Writer writer) throws IOException {

		writer.append("HelloWorld");
	}
}
```

3.2.3.10. JSON 形式を取得する場合

Ajax でよく使用する JSON 形式でデータを取得する方式を説明します。

- 引数として、パラメータを受け取る場合は指定します。
- 戻り値は、オブジェクトを返します。MessageConverter で変換されるため、変換可能なものならば何でも構いません。
- メソッドに、アノテーション「@ResponseBody」を付加します。自動的に、HTTP ヘッダーの Accept が「application/json」になります。
  - レスポンス時のメディアタイプを指定したい場合は、@RequestMapping(…,headers="Accept=application/json"))で指定します。
  - または、Spring3.1 で追加になった「@RequestMapping(…,produces="application/json"))」で指定します。
- JSON を使用する場合、別途ライブラリ「Jackson」が必要となります。Maven の設定は、「2.4.1.1　図 2.5　JSON 通信に必要なライブラリ」を参照してください。
- JSON などによるデータの送受信の詳細は、「5　REST サービスの作成」を参照してください。

【Contoller 側】

```
@Controller
public class JsonController {

    @RequestMapping(value="/ajax/jsonOut1", method={RequestMethod.GET, RequestMethod.POST})
    @ResponseBody
    public List<UserInfoViewDto> jsonOut (@RequestParam String cd) {

        if(StringUtils.isEmpty(departmentCd)) {
            return new ArrayList<UserInfoViewDto>();
        }

        List<UserInfoDto> list = /* Servlet/DAO から取得する  */;
        return list;

    }
}
```

【クライアント側（Web ブラウザ側）】

- レスポンス時のデータが JSON や XML の場合、通常は jQuery などで取得します。その場合、レスポンス時のメディアタイプを指定します。

```
<script type="text/javascript">
$(document).ready(function(){
	$('#jsonOut1').click(function(){
		$.ajax({
			type: "POST",
			url : "${appUrl}/ajax/jsonOut1.html",
			data : {"cd": "syasin"},
			// 受信時のメディアタイプ
			dataType: "json",
			success:function(data){
				//TODO:
				alert(data);
			}
		});
	});
});
</script>
<ol>
	<li> <a href="${appUrl}/ajax/jsonOut1.html?cd=aaa">JSON 形式を取得(GET で取得)</a></li>
	<li> <span id="jsonOut1">JSON 形式を取得(jQuery 経由で取得)</span></li>
</ol>
```

3.2.3.11. 遷移先を省略した場合の挙動（メソッドの戻り値が void）

@RequstMapping を付加したメソッドは、戻り値として void を許可していますが、その場合、注意が必要になります。

- 戻り値が void の場合、自画面遷移します。
  View のパスは、@RequestMapping で設定した同じパスになります。
- JSP にモデルを渡したい場合は、引数に ModelMap(or Model)を取るか、または戻り値として返します。
- 遷移先の URL が変わらないため、HTML の設定やブラウザの設定によっては、HTML が更新されない場合があります（View のパスを指定し、リクエストを同じ URL を指定しても同様）。
  その場合、HTML のキャッシュを行わないよう<meta>タグなどに記述しておく必要があります。
  また、この対策として、「3.4.3　フラッシュスコープ（Flash Scope）を使用した redirect による URL 転送」を利用します。

```
@Controller
public class AppointmentsController

    @RequestMapping("/sample")
    public void doAction(ModelMap model) {
        model.addAttribute("message1", "aaaa");
    }
}
```

3.2.3.12. 遷移先を省略した場合の挙動（View のパスが空）

ModelMap を返却する場合、View のパスを空に設定できますが、その場合、注意が必要になります。
前節の「3.2.3.11　遷移先を省略した場合の挙動（メソッドの戻り値が void）」と同じ挙動をします。

- View のパスが空の場合、自画面遷移します。
  View のパスは、@RequestMapping で設定した同じパスになります。

```
@Controller
public class AppointmentsController

    @RequestMapping("/sample")
    public ModelAndView doAction() {

        ModelAndView mav = new ModelAndView();
        model.addObject("message1", "aaaa");
        return mav;
    }
}
```

3.3. @RequestMapping による様々な URL の処理

Controller にどのような URL を関連付けるかどうかは、アノテーション「@RequestMapping」で定義します。Spring MVC は非常に柔軟にできており、１つのコントローラに対して、複数の URL を定義することもできます。以下の図 3.7 に例を示します。

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

    @Autowired
    public AppointmentsController(AppointmentBook appoint
        this.appointmentBook = appointmentBook;
    }

    @RequestMapping(method = RequestMethod.GET)
    public Map<String, Appointment> get() {
        return appointmentBook.getAppointmentsForToday();
    }

    @RequestMapping(value="/{day}", method = RequestMethod.GET)
    public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day,
Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    @RequestMapping(value="/new", method = RequestMethod.GET)
    public AppointmentForm getNewForm() {
        return new AppointmentForm();
    }

    @RequestMapping(method = RequestMethod.POST)
    public String add(@Valid AppointmentForm appointment, BindingResult result) {
        if (result.hasErrors()) {
            return "appointments/new";
        }
        appointmentBook.addAppointment(appointment);
        return "redirect:/appointments";
    }
}
```

クラスに付加することで、ベースとなる URL を定義します。」
クラスに付加する場合は、通常は名前のみ定義します。

URL「/approintments」でかつ、GET メソッドでアクセスした際に呼び出されます。

URL「/approintments/yyyy-MM-dd」の形式で、かつ GET メソッドでアクセスした際に呼び出されます。
{day}の部分は動的に変わります。

URL「/approintments/new」の形式で、かつ GET メソッドでアクセスした際に呼び出されます。

URL「/approintments」の形式で、かつ POST メソッドでアクセスした際に呼び出されます。

図 3.7　RequestMapping の定義サンプル

3.3.1. アノテーション「@RequestMapping」の仕様

- Spring 3.1 から、引数に「consumes」と「produces」が追加されました。

表 3.3　@RequestMapping の定義可能場所

No.	定義場所	説明
1	クラス	・メソッド付加した際のベースとなる設定を定義します。 ・通常は、基本となる URL を定義する際に使用します。 ・省略可能です。
2	メソッド	・URL、HTTP の受理するメソッドなど様々なものを定義します。 ・省略できません。

表 3.4　@RequestMapping の引数

No.	型 引数名	説明
1	String[] value	・URL を定義します。 ・初期値は、{}で、省略可能です。 ・URL のみ定義する場合は、引数名を省略できます。
2	RequestMethod[] method	・受理するリクエストを HTTP のリクエストメソッドを制限するために使用します。省略するとすべてのメソッドを受理します。 ・初期値は、{}で、省略可能です。 ・GET、POST、HEAD、OPTIONS、PUT、DELETE、TRACE が指定できます。
3	String[] params	・受理するリクエストをパラメータにより制限するために使用します。 ・初期値は、{}で、省略可能です。 ・具体的な使用方法は「3.3.8　複数の submit ボタンにより処理を振り分ける」を参照のこと。 ・値には演算子「=、!=」を使用できます。
4	String[] headers	・受理するリクエスト、出力するレスポンスを任意の HTTP ヘッダーにより制限するために使用します。 ・HTTP ヘッダー「Content-Type」「Accept」をそれぞれ指定したい場合は、Spring 3.1 から追加になった「consumes」「produces」を使用します。 ・初期値は、{}で、省略可能です。 ・値には演算子「=、!=」やワイルドカード「*」を使用できます。 ・例)

		headers="Accept=application/xml, application/json" headers = "content-type=text/*" headers = "content-type!=text/*" headers={"Content-Type=text/xml", "Accept=application/xml"}
5	String[] consumes	・サーバが受理するリクエストを HTTP リクエストヘッダーのメディアタイプ（Content-Type）を制限するために使用します。 ・Spring 3.1 より追加されました。 ・初期値は、{}で、省略可能です。 ・メソッドに指定した@RequestMapping で指定した値が優先されます。 ・値には否定演算子「!」やワイルドカード「*」を使用できます。 ・例) consumes="text/plain" consumes="application/json" consumes="application/*" consumes="!text/plain"
6	String[] produces	・サーバが出力するレスポンスのメディアタイプの HTTP レスポンスヘッダー（Accept）を指定するために使用します。 ・Spring 3.1 より追加されました。 ・初期値は、{}で、省略可能です。 ・メソッドに指定した@RequestMapping で指定した値が優先されます。 ・値には否定演算子「!」やワイルドカード「*」を使用できます。 ・例) produces="text/plain" produces="application/json" produces="application/*" produces="!text/plain"

3.3.2. サンプル「クラスに定義する」

【ケース】

- クラスに定義し、その際に基本となる URL を定義します。
- メソッド「doAction()」には、受理対象の HTTP メソッド GET を定義しています。
  インスタンスは、列挙型「org.springframework.web.bind.annotation.RequestMethod」です。
- メソッドの側の@RequestMapping には、URL の指定がないため、クラスに定義した URL「/sample」がそのまま使用されます。

【リクエスト URL】

- 「/<APP 名>/sample.html」
- GET メソッドのみ受理します。

```
@Controller
@RequestMapping("/sample")
public class AppointmentsController

    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView doAction() {
        ・・・
    }
}
```

3.3.3. サンプル「メソッドのみに定義する」

【ケース】

- メソッドのみに、@RequestMapping を定義します。
- クラスには@RequestMapping の定義がなく、URL の指定もないため、メソッドに定義した URL「/sample」がそのまま使用されます。

【リクエスト URL】

- 「/<APP 名>/sample.html」
- HTTP メソッドの制限はありません。

```
@Controller
public class AppointmentsController

    @RequestMapping("/sample")
    public ModelAndView doAction() {
        ・・・
    }
}
```

3.3.4. サンプル「URL をクラスとメソッドの両方に定義する」

【ケース】

- クラスに定義し、URL を”/sample”と定義すると、メソッドに対しては、URL として”/sample/～”となります。
- メソッド new()の@RequestMapping に対しては、URL”/new”が定義されているため、クラス側に定義した”/sample”と結合され”/sample/new”として処理されます。

【リクエスト URL：メソッド doAction()の場合】

- 「/<APP 名>/sample.html」
- HTTP の GET メソッドのみ受理します。

【リクエスト URL：メソッド new()の場合】

- 「/<APP 名>/new.html」
- HTTP の GET または POST のみ受理します。。

```
@Controller
@RequestMapping("/sample")
public class AppointmentsController
    // 処理①
    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView doAction() {
        ・・・
    }

    // 処理②
    @RequestMapping(value="/new", method = {RequestMethod.GET, RequestMethod.POST})
    public ModelAndView new() {
        ・・・
    }
}
```

3.3.5. サンプル「Welcome 用の URL を定義する」

【ケース】

- @RequestMapping の URL に、「””(空)」「/」と定義します。
- 遷移先として、”/index”としておき、index.jsp などに遷移するようにしておきます。

【リクエスト URL】

- 「/<APP 名>/」
- HTTP メソッドの制限はありません。

```
@Controller
public class IndexController

    @RequestMapping(value={"","/"})
    public ModelAndView doAction() {
        ModelAndView mav = new ModelAndView("/index);

        return mav;
    }
}
```

3.3.6. サンプル「URL の一部が動的に変化する URL を定義する」

【ケース】

- @RequestMapping の URL に、”{変数名}”を定義します。引数には、変数名と対応した「@PathVariable(“変数名”)」を付けます。
- RESTful な Web サービスを作成する際に利用します。詳細は、「5　REST サービスの作成」参照してください。

【リクエスト URL】

- 「/<APP 名>/owners/user1」。（user1 は動的に変わります）

```
@Controller
public class AppointmentsController {

    @RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
    public String findOwner(@PathVariable("ownerId") String ownerId, Model model) {
        // implementation omitted
    }
}
```

3.3.7. 同じ URL に対して HTTP メソッドにより処理を振り分ける

【ケース】

- アクセスの仕方により処理を振り分けます。
- クラスの@RequestMapping に URL を記述しておき、メソッドの@RequestMapping に対しては、リクエストメソッドを定義します。
- GET メソッド（ブラウザの URL を直打ち）でアクセスしてきた場合、init()メソッドが呼ばれ、POST メソッド（form からサブミット）でアクセスしてきた場合、onSubmit()メソッドが呼ばれる。

【リクエスト URL】

- 「/<APP 名>/common/login」。

```
@Controller
@RequestMapping("/common/login")
public class LoginController {

    @RequestMapping(method={RequestMethod.GET})
    public ModelAndView init(ModelMap model) {
        ・・・
    }

   @RequestMapping(method={RequestMethod.POST})
    public ModelAndView onSubmit(ModelMap model) {
        ・・・
    }
}
```

3.3.8. 複数の submit ボタンにより処理を振り分ける

【説明】

- 1 つの form の中にボタンが複数あり、メソッドをボタンごとに振り分けます。
- 振り分けを行う判断基準は、ボタンの名前(<input type="submit" name="ボタンの名前" />)です。
- Controller では、同じ URL を設定し、その際に、パラメータ名（@RequstMapping(params="ボタンの名前")）と対応付きます。

```
<p>ボタンにより振り分ける</p>
<form action="${appUrl}/test/confirm.html" method="get">
    <p><input type="text" name="name" /></p>
    <p><input type="text" name="age" /></p>
    <p>
        <input type="submit" name="confirm" value="確認" />
        <input type="submit" name="cancel" value="キャンセル" />
    </p>
</form>
```

図 3.8　ボタンによる振り分けを行う JSP の例

```
@Controller
@RequestMapping("/test ")
public class DispatchButtonController {

    // 確認ボタンを押下した際の処理
    @RequestMapping(value="/confirm", params="confirm")
    public ModelAndView doConfirm() {

        System.out.println("[確認]ボタンが押下されました。");
        ModelAndView mav = new ModelAndView("/test/complete");
        return mav;
    }

    // キャンセルボタンを押下した際の処理
    @RequestMapping(value="/confirm", params="cancel")
    public ModelAndView doCancel() {

        System.out.println("[キャンセル]ボタンが押下されました。");
        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

・JSP のボタン名と一致した場合、処理が実行される。
・実際には、「指定したパラメータ名を持つリクエストのみを受理する」という意味。

図 3.9　ボタンによる振り分けを行う Controller の例

3.3.8.1.　リンクにより処理を振り分ける

リンク形式（<a href="">）による、Controller の処理を振り分けることも可能です。

- クエリストリングに、@RequestMapping の「params」で指定した名前のパラメータを追加します。その際に、値は特に何でもよく、空でも構いません。
- Submit ボタンは、フォームの値として「<input type="hidden" name="ボタン名">」と同じ意味です。

```
<p>リンクサブミットにより振り分ける</p>
<ol>
    <li><a href="${appUrl}/test/confirm.html?confirm=">確認</a></li>
    <li><a href="${appUrl}/test/confirm.html?cancel=">キャンセル</a></li>
    <li>
        <spring:url value="/test/confirm.html" var="confirmUrl">
            <spring:param name="confirm" value="" />
        </spring:url>
        <a href="${confirmUrl}">確認（カスタムタグによる URL の組み立て）</a>
    </li>
    <li>
        <spring:url value="/test/confirm.html" var="cancelUrl">
            <spring:param name="cancel" value="" />
        </spring:url>
        <a href="${cancelUrl}">キャンセル（カスタムタグによる URL の組み立て）</a>
    </li>
</ol>
```

図 3.10　リンクによる振り分けを行う JSP の例

3.3.8.2. サブミットリンクにより処理を振り分ける

リンクをクリックした際に、form をサブミットする場合。jQuery で例を示します。

- リンクをクリックした際のイベントを追加します。イベントは form オブジェクトを取得し、submit() 関数を実行します。
- jQuery で、<input type=”hidden” name=”ボタン名”>のタグを form に追加します。
  <input>タグを追加する方式として、文字列でタグを記述する方式と、jQuery の関数でタグを組み立てる方式があります。
  複雑なタグを組み立てる場合や、属性名が引数によって変わるような場合は、jQuery を使用し組み立てる方がわかりやすくなります。
- この方式は、DOM を書き換えるため、Ajax のような画面全体を再読み込みしない場合は、使用できないので注意してください。

```
<script type="text/javascript" src="${appUrl}/js/lib/jquery.min.js"></script>
<script type="text/javascript">
    $(function() {

        // 確認リンクのサブミット
        $('a#confirmLink').click(function(){
            var formObj = $('form[name="confirmForm"]');
            // jQuery によるタグの組み立て
            $(formObj).append($('<input/>').attr({type:'hidden', name:'confirm'}));
            //文字列によるタグの記述
            //$(formObj).append('<input type="hidden" name="confirm"/>');
            $(formObj).submit();
        });

        // キャンセルのリンクのサブミット
        $('a#cancelLink').click(function(){
            var formObj = $('form[name="confirmForm"]');
            $(formObj).append($('<input/>').attr({type:'hidden', name:'cancel'}));
            //$(formObj).append('<input type="hidden" name="cancel"/>');
            $(formObj).submit();
        });
    });
</script>

<h4>複数の submit を振り分ける。</h4>

<!-- submit 対象の form -->
<form    name="confirmForm"    action="${appUrl}/test/confirm.html" method="post">
    <p><input type="text" name="name" /></p>
    <p><input type="text" name="age" /></p>
</form>

<p>サブミットリンク<p>
<ol>
    <li><a id="confirmLink" href="#">確認</a></li>
    <li><a id="cancelLink" href="#">キャンセル</a></li>
</ol>
```

3.3.8.3. パラメータで絞り込む処理の方法

「@RequestMapping(params="")」は、どのようなパラメータを持つかどうかでリクエストを絞り込むということがわかったと思います。params の値には演算子「=」「!=」「!」を使用でき、複雑な条件で絞り込むことができます。ここでは、様々な条件による絞り込みの例を説明します。

表 3.5　パラメータで絞り込む場合の例

No.	params の記述例	説明
1	@RequestMapping(params="confirm")	・パラメータ名「confirm」を持つリクエストのみ受理します。 ・値についは任意してください。
2	@RequestMapping(params="confirm=確認")	・パラメータ名「confirm」を持つリクエストのみ受理します。 ・値は、「確認」と等しい場合のみ受理します。
3	@RequestMapping(params="confirm!=確認")	・パラメータ名「confirm」を持ち、かつ値が「確認」とは異なる場合のみ受理します。 ・パラメータ「confirm」を持っていることが前提です。
4	@RequestMapping(params="!confirm ")	・パラメータ名「confirm」を持っていないリクエストのみ受理します。

3.3.9. HTTP ヘッダーによりリクエスト／レスポンスを制限する(TODO)

3.4. URL への転送方法

3.4.1. Forward による URL 転送

- 他の URL に対して転送を行う場合、forward を使用します。ログイン画面後に、メインメニューに遷移する場合、メインメニューの初期表示用の Controller に遷移する際などに利用します。
- 同じ WebApp 内の URL に対して転送可能です。外部の「http://～」のような URL に対しては転送できません。その場合は、Redirect を使用してください。
- Forward した際の Web ブラウザ上の URL は、転送元の URL になり、転送先とは一致しません。

3.4.1.1. 単純な forward

- 戻り値の View の URL に対して、接頭語「forward:」を付加します。
- Servlet API の forward と同様、転送元のリクエストが POST メソッドで実行されている場合、転送先も POST メソッドになります。

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public String sample() {

        return "forward:/hello.html";
    }
}
```

3.4.1.2. パラメータを渡して forward する

他の URL（Controller）に転送し、その際にパラメータを渡す場合を説明します。

- 戻り値の View の URL に対して、接頭語「forward:」を付加します。
- 戻り値の URL に対して、クエリストリングの形式でパラメータを設定します。
  Model に値を設定しても転送されません。
  URL の長さにはブラウザに上限値があるため、あまり長いものは渡せません。
- パラメータに関しては、エンコードなど特に必要ありません。

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public String sample() {

        return "forward:/hello.html?msg1=aaa&msg2=bbb";
    }
}
```

3.4.2. Redirect による URL 転送

- 外部の URL に対して転送する場合などに、redirect を使用します。
- リダイレクトをした場合の Web ブラウザ上の URL は、転送先の URL になり、転送先と一致します。

3.4.2.1. 単純な redirect

- 戻り値の View の URL に対して、接頭語「redirect:」を付加します。
- リダイレクトの場合、外部 URL からのリクエストとして処理されるため、遷移先のコントローラでは GET メソッドでアクセスすることになります。

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public String sample() {
        // WebApp 内部の URL に対してリダイレクト
        return "redirect:/hello.html";

        // WebAapp 外部の URL に対してリダイレクト
        return "redirect:http://www.google.co.jp/";
    }
}
```

3.4.2.2. パラメータを渡して redirect する

他の URL（Controller）に転送し、その際にパラメータを渡す場合。

- 戻り値の View の URL に対して、接頭語「redirect:」を付加します。
- パラメータは Model に設定します。その場合 URL エンコードなど必要ありません。
  Model に設定した値は、自動的にクエリストリングとして URL に付加し処理されるため、クエリストリングとして渡せる簡単なもののみ設定します。JavaBean などの複雑なオブジェクトは設定できません。
- Forward 時と同様に、URL にクエリストリングとして直接追加しても問題ないですが、その際には URL エンコードが必要となります。「java.net.URLEncoder.encode()」を使用しエンコードしてください。

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public ModelAndView sample() {
        ModelAndView mav = new ModelAndView("redirect:/hello.html");

        mav.addObject("msg1", "aaa");
        mav.addObject("msg2", "bbb");
        return mav;
    }
}
```

3.4.3. フラッシュスコープ（Flash Scope）を使用した redirect による URL 転送（独自実装）

フラッシュスコープは、Ruby On Rails で有名になった機能で、redirect する際に、Model オブジェクトを完全に保持し渡すことができます。すなわち、リダイレクト先のページにパラメータを渡すことができます。Spring MVC3.0 では未実装であるため独自に実装します。また、次期バージョン Spring MVC 3.1 において実装される予定です。
実装方法は、「8.5　フラッシュスコープの実装」を参照してください。

【特徴】

- Redirect のように、ブラウザ上の URL は、遷移先の URL と一致します。
  Forward は遷移元の URL だったため、ブラウザをリロードすると再度実行されることとなり、2 重送信されてしまうのでチェック機構が必要でした。
- 遷移先の URL（View）に対して、Model オブジェクトを渡すことができます。
  Redirect、Forward のようにパラメータで渡す場合、複雑な JavaBean のようなデータは渡せませんでした。また、URL の長さにも上限があるため、データサイズにも注意が必要でした。
- Forward と同様に、WebApp 内の URL に対してのみ遷移可能です。
  Model オブジェクトを、一端セッションに格納するため、異なる Web コンテナへの転送には利用できません。

【記述方法】

- 遷移元では、フラッシュスコープを利用するには、遷移先の View に対して接頭語「redirect_with_flash:」を付加します。
- 渡したデータは、遷移先の JSP で直接利用可能です。

```
@Controller
public class SampleController {

    @RequestMapping("/sample")
    public ModelAndView sample() {
        ModelAndView mav = new ModelAndView("redirect_with_flash:/hello.html");

        mav.addObject("msg1", "aaa");
        mav.addObject("msg2", "bbb");
        return mav;
    }
}
```

3.4.4. フラッシュ属性を使用した redirect による URL 転送（Spring MVC 3.1）

Spring MVC 3.1 からフラッシュスコープが正式にサポートされました。正式名称は、「Flassh Attribute（フラッシュ属性）」と呼びます。フラッシュ属性を利用することで、をリダイレクト先のページにパラメータを渡すことができます。

【送信元】

- コントローラの引数に「RedirectAttributes」を指定します。
- フラッシュスコープで送りたい値は、「addFlashAttribute()」メソッドで指定します。
  - メソッド「addFlashAttribute()」で設定した値は、URL エンコードなどは必要ありません。
  - メソッド「addAttribute()」は、通常の Redirect 時と同様に送られるので URL エンコードが必要となり注意が必要です。

```
@Controller
public class FromController {
	@RequestMapping(value = "/from", method = RequestMethod.GET)
	public String from(RedirectAttributes attributes) {
		//メッセージ hogehoge を渡す
		attributes.addFlashAttribute("message","hogehoge");
		return "redirect:/to";
	}
}
```

【リダイレクト先（転送先）】

- フラッシュ属性で送られた値は、リダイレクト先の Model として設定されるため、コントローラの引数に Model（または、相当する Map、ModelMap）を設定し、値を取得します。

```
@Controller
public class ToController {
	@RequestMapping(value = "/to", method = RequestMethod.GET)
	public View to(Model model) {
		// フラッシュで送ったメッセージ
		String message = model.get("message);
		ModelAndView mav = new ModelAndView("view_message");
		return mav;
	}
}
```

- 引数に Model を指定しない場合、遷移先の JSP に渡されるため、フラッシュ属性で送られた値は JSP でそのまま使用できます。

```
@Controller
public class ToController {
	@RequestMapping(value = "/to", method = RequestMethod.GET)
	public String to() {
		return "view_message";
	}
}
```

3.5. ViewResolver(:TODO)

PDF、XSLT、Excel、Feed、XML

3.5.1. Apache Tiles を使用する(:TODO)

4. Form データの送受信

4.1. 基本的なデータの送受信

Form からのデータの受け取り方法、初期の設定方法を説明します。

4.1.1. @RequestParam によるデータの送受信

【Controller のサンプル】

Form によるデータを受け取る場合、Controller は一般的に 2 種類の@ReqeuestMapping を設定します。

- 1 つ目は「form の初期値などを設定」するメソッドです。URL に直接アクセスした場合、HTTP では GET メソッドで処理されるので、RequestMethod.GET で振り分けます。
  その際に、Model に form の各値を設定します。
- 2 つ目は「form の値を受け取り処理する」メソッドです。Form から POST で送信されてくる（=データをサブミットされた）データを処理します。
- 引数には、受け取るデータをバインドするための「@RequestParam」を定義します。POST で submit されたデータは、このアノテーションが付加された引数に値が設定されます。設定値により様々な動作を定義ことができます（「4.1.1.1　アノテーション「@RequestParam」の仕様」を参照）。

```
@Controller
@RequestMapping("/test/form1")
public class Form1Controller {

    // 初期値の設定(get でアクセス)
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {
        model.addAttribute("name", "");
        model.addAttribute("mail", "");
        model.addAttribute("age", "0");
        model.addAttribute("isConfirmed", true);
    }

    // post で送られた場合
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(@RequestParam String name,
            @RequestParam(value="mail", defaultValue="default@example.com") String mailAddress,
            @RequestParam(defaultValue="0") Integer age) {
            @RequestParam(defaultValue="false") Long confirmed

        ・・・処理を行う。
        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

・ブラウザから直接アクセス(=GET)した場合に呼ばれるメソッド。
・動作として“初期化”のために使用するのに向いている

・Form の各項目の名前を一致させる必要があります。
・value 属性を省略した場合は、変数名が名前となります。

図 4.1　@RequestParam にデータを受け取る Controller のサンプル

【JSP のサンプル】

- Controller 側でデータを送信する場合などは、JSP 側はカスタムタグを利用しない素の HTML で記述します。
- HTML の name 属性は、Controller 側の@RequestParam で受け取る名前と一致させる必要があります。
- 初期値は、Model から呼び出します。

```
<form action="${appUrl}/test/form1.html" method="post">
    <p>
        <label for="name">名前</label>
        <input id="name" type="text" name="name" value="${name}" />
    </p>
    <p>
        <label for="mail">メールアドレス</label>
        <input id="mail" type="text" name="mail" value="${mail}"/>
    </p>
    <p>
        <label for="age">年齢</label>
        <input id="age" type="text" name="age" value="${age}" />
    </p>
    <p>
        <label for="checked">チェック</label>
        <input id="checked" type="checkbox" name="confirmed " value="1 "
 <c:if test="${isConfirmed}">checked='checked'</c:if> />
    </p>
    <input type="submit"/>
</form>
```

@RequestParam にバインドする名前と一致させる必要があります。

Model から初期値を取得します。

チェックボックスの場合、初期値は属性「checked=”checked”」を設定するかどうかを設定します

図 4.2　Form によるデータを送信する JSP のサンプル

【注意点】

- @RequestParam のみの場合、引数 Errors／BindingResult を使用することはできません。その場合、システムエラーが発生します。
  Errors／BindingResult によるバインドエラーを検知したい場合は、@ModelAttribute を使用してデータを受け取るようにしてください（「4.1.2　Command(@ModelAttribute)によるデータの送受信」）。
- バインドが発生するようなケースでは、@RequestParam では、「required=false」とします。バインドエラーが発生した場合、「400 エラー　The request sent by the client was syntactically incorrect ().」や「java.util.IllegalFormatConversionException」が発生します。
  このエラーは、バインド先が見つからない場合や型か異なる場合に発生します。
- チェックボックスなどの値を受け取る場合、@RequestParam では、必ず「required=false」と設定します。チェックボックスにチェックが入っていない場合、データ自体送信されずバインドできないため、システムエラーが発生します。
  型は必ず Long 型を設定します。（Boolean 型、String 型では変換できないためエラーが起きます。）

4.1.1.1. アノテーション「@RequestParam」の仕様

- アノテーション「@RequestParam」は、@RequestMapping が定義されているメソッドの引数にのみ定義可能です。

表 4.1　@RequestParam の引数

No.	型 引数名	説明
1	String value	・パラメータの名称を定義します。 Form で定義した項目の名前（例. <input type=”name”/>）と一致させる必要があります。 ・初期値は、{}で、省略可能です。 ・省略した場合は、Java の変数名が名称になります。
2	boolean required	・パラメータが必須かどうか定義します。 ・初期値は、true です。 ・defaultValue が設定されると、暗黙的にこの値は false になります。
3	String defaultValue	・パラメータのデフォルト値を定義します。 ・初期値は、「¥n¥t¥t¥n¥t¥t¥n¥uE000¥uE001¥uE002¥n¥t¥t¥t¥t¥n」です。 ・required=false でかつ、対応する名称の form のパラメータがない場合に設定される値です。

4.1.2. Command(@ModelAttribute)によるデータの送受信

Command(=ModelAttribute)とは、JavaBean に Form をバインドさせる方式です。
JSP の Form の各項目と Command オブジェクトをバインドさせるために、JSP では SpringMVC 専用のカスタムタグを使用します。

【Command クラスの作成】

- JavaBean 形式のように、各プロパティに対して settter、getter を定義します。
- Command をセッションに格納することを考慮し Serializable を実装します。
- デバッグしやすいように、toString()を実装します。その際に、Commons-Lang の「ToStringBuilder」を使用すると便利です。
  Spring にも似たようなクラス「ToStringCreator」がありますが、こちらは、リフレクションによる組み立てができないなど使用するには不便です。
- Command の各プロパティは、数値型などプリミティブ型（float、double、int、long、boolean、char）を使用しないことをお勧めします。その代わり、プリミティブ型のラッパークラスを使用します。
  理由として、プリミティブ型の場合、HTML で値が空(= NULL)を表現できないからです。

```
import java.io.Serializable;

import org.apache.commons.lang.builder.ToStringBuilder;


public class SampleCommand implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private String name;

    private String mail;

    private Integer age;

    private Boolean confirmed;

    public SampleCommand() {}

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }

    // getter,setter は省略
}
```

リフレクションによる組み立てを利用すると、よりコード量が減ります。

【Controller クラスの作成】

Command を使用したデータを受信する場合、最低 3 つのメソッドを用意します。

- １つ目として、「Command オブジェクトの初期データを取得」するメソッドです。メソッドに対して、アノテーション「@ModelAttribute」を付与します。属性には、Command の名称を定義します。
  メソッドに付与した場合、リクエストを処理し、データをバインドする Command のインスタンスを生成する際に呼ばれます。
  このメソッドは通常は定義しなくても Controller として動作しますが、バインドエラー時などにも呼ばれるので、定義が無いとシステムエラーとなるので、作成しておくことをお勧めします。
- 2 つめとして、「form の初期値などを設定」するメソッドです。URL に直接アクセスした場合、HTTP では GET メソッドで処理されるので、RequestMethod.GET で振り分けます。
  初期値を設定したい場合、Command オブジェクトを Model に設定します。Model に設定しない場合は、1 つめのメソッドから取得した値が、Command オブジェクトの初期値となります。
- 3 つめとして、「form の値を受け取り処理する」メソッドです。Form から POST で送信されてくる（=データをサブミットされた）データを処理します。
  引数には、Command クラスを定義し、「@ModelAttribute」を付与します。
- 引数 BindingResult（または、Errors）を定義します。これには、バインドエラー時や、入力エラー時のエラーメッセージが格納されます。
  入力エラーがあった場合、BindingResult#rejectXXX()を使用します。Validator クラスを使用する方法などの詳細は、「7　入力値検証」を参照してください。

```
@Controller
@RequestMapping("/test/form2")
public class Form2Controller {

    // ①command の初期オブジェクトの取得
    @ModelAttribute("sampleCommand")
    public SampleCommand createInitCommand() {
        SampleCommand command = new SampleCommand();
        return command;
    }

    // ②初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        SampleCommand command = createInitCommand();
        command.setAge(1);
        model.addAttribute("sampleCommand", command);

    }

    // ③post で送られた場合
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(@ModelAttribute("sampleCommand") SampleCommand command,
BindingResult bindingResult) {
```

Command の名前は、Controller 側と JSP 側と一致させる必要があります。Controller の定数としてしておいてもよいかもしれません。

```
        System.out.println(command);

        // command の各項目のエラーチェック
        if(StringUtils.isEmpty(command.getName())) {
            bindingResult.rejectValue("name", "error.required");
        }

        //  共通のエラーチェック
        if(bindingResult.hasErrors()) {
            bindingResult.reject("error.message");
        }

        //エラーがある場合（もとの画面へ戻る）
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());
            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;

    }
}
```

入力値のチェックを行います。

何かしらのエラーがあった場合、エラー内容を Model に移し換え自画面へ遷移します。

【JSP のサンプル】

- Spring 用のカスタムタグの定義を宣言します。Spring には 2 種類のカスタムタグがあり、<spring:XXX>と<form:XXX>があります。詳細は、「12　カスタムタグ」を参照してください。
- form を定義するには、<form:form>を使用します。属性 modelAttribute には、Controller で定義した Command の名前を設定します。HTML では、「id=”modelAttribute の名前”」になります。
- テキストフィールドを表示するには、カスタムタグ<form:input path="Command のプロパティ名">を使用します。HTML の各入力項目に対応するカスタムタグがあります。
  詳細は、「12.2　Spring MVC 用のカスタムタグ 2（<form:XXX>）」を参照してください。
- 項目ごとに固有のエラーを表示するには、<form:errors path="Command のプロパティ名" cssClass="class 属性">を使用します。エラーがある場合には、「メッセージ 」の形式の HTML として出力されます。

```
<%-- Spring のカスタムタグの定義 --%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>

<spring:hasBindErrors name="sampleCommand">
<div>
    <font color="red"><spring:message code="error.input" /></font>
    <%-- 共通のエラーメッセージの表示 --%>
    <font color="red"><c:forEach items="${errors.globalErrors}" var="error">
        <spring:message message="${error}" /><br/>
    </c:forEach></font>
</div>
</spring:hasBindErrors>

<form:form modelAttribute="sampleCommand" action="${appUrl}/test/form2.html" method="post">
    <p>
        <form:label path="name">名前</form:label>
        <form:input path="name" />
        <form:errors path="name" cssClass="errors" />
    </p>
    <p>
        <form:label path="mail">メールアドレス</form:label>
        <form:input path="mail" />
        <form:errors path="mail" cssClass="errors" />
    </p>
    <p>
        <form:label path="age">年齢</form:label>
        <form:input path="age" />
        <form:errors path="age" cssClass="errors" />
    </p>
    <p>
        <form:label path="confirmed">確認</form:label>
        <form:checkbox path="confirmed" />
        <form:errors path="confirmed" cssClass="errors" />
    </p>
    <input type="submit"/>
</form:form>
```

何かしらのエラーがある場合に表示させる共通のメッセージです。

プロパティに関連付いていないエラーメッセージの表示。

使用するコマンド名を定義します。
Controller 側の@ModelAttribute で定義した名前と一致させます。

属性「path」は、Command から見たプロパティの位置を示します。
詳細は、「4.5.1　プロパティの位置(=path)の表現」を参照。

【エラーメッセージの定義】

Spring Bean の“messageSource”として定義しているプロパティファイルにエラーメッセージを定義します。通常 2 種類存在します。

- 1 つめは、データのバインドに失敗した際に表示されるメッセージです。これは、DefaultMessageCondesResolver により自動的に設定されます。
  詳細は「4.2　データバインドエラー（型ミスマッチ）処理」を参照してください。
- 2 つめは、入力値チェックのエラー時に表示されるメッセージです。エラーチェックの方法により、3 つに分かれます。
  - 1 つ目は、JSP 側の<spring:message>で直接呼び出す場合です。
  - 2 つ目は、Command の属性に対してではなく、共通のエラーに使用します。例えば、項目間チェックなどのエラーメッセージに使用します。
    BindingResult#reject("エラーコード")により設定します
  - 3 つめは、Command の属性に対して使用します。単項目チェックに使用します。
    Bean Validator や BindingResult#rejectValue("属性名","エラーコード")により設定します。

```
## 型が合わない場合のメッセージ
typeMismatch=入力形式が不正です。
typeMismatch.int=整数で入力してください。
typeMismatch.java.lang.Integer=整数で入力してください。

## 入力値エラーのメッセージ
# (1)JSP から直接呼び出すメッセージ
error.input=入力内容を確認してください。

# (2)共通のエラー用、項目間のエラー用
error.message=共通のエラーメッセージ。

# (3)単項目エラー用
error.required=必須です。
```

【生成された HTML のサンプル】

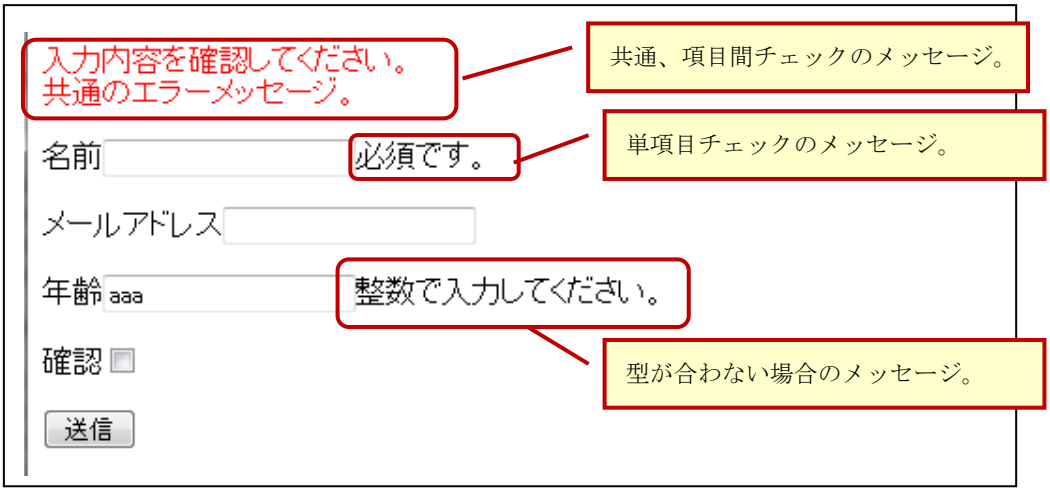


図 4.3　Web ブラウザでの表示

```
<!-- エラーがあるかどうかの判定 -->
<div>
    <font color="red">入力内容を確認してください。</font><br/>
    <font color="red">共通のエラーメッセージ。<br/>
    </font>
</div>
<form id="sampleCommand" action="/spring3-mvc/test/form2.html" method="post">
    <p>
        <label for="name">名前</label>
        <input id="name" name="name" type="text" value=""/>
        <span id="name.errors" class="errors">必須です。</span>
    </p>
    <p>
        <label for="mail">メールアドレス</label>
        <input id="mail" name="mail" type="text" value=""/>
    </p>
    <p>
        <label for="age">年齢</label>
        <input id="age" name="age" type="text" value="aaa"/>
        <span id="age.errors" class="errors">整数で入力してください。</span>
    </p>
    <p>
        <label for="confirmed1">確認</label>
        <input id="confirmed1" name="confirmed" type="checkbox" value="true"/>
        <input type="hidden" name="_confirmed" value="on"/>
    </p>

    <input type="submit"/>
</form>
```

共通、項目間チェックのメッセージ。

modelAttribute 属性は、id 属性に変換されます。

単項目チェックのメッセージ。

型が合わない場合のメッセージ。

図 4.4　HTML のソース

4.2. データバインドエラー（型ミスマッチ）処理

Spring MVC では POST などされたデータを、Command などの各プロパティ（=フィールド）にバインドする際に、型変換を行います。その際に、Validator でチェックする前に実行され、その結果は、BindingResult、Errors クラスに格納されます。

バインド時に型変換エラーが起こると、次のようなエラーが出力されます。開発者は理解できるかもしれませんがユーザにとっては意味不明な内容です。

```
Failed to convert property value of type [java.lang.String] to required type [{型名}] for property '{プロパティ名}'; nested exception is ...
```

型変換エラーは、専用のメッセージを用意することで、入力項目に対してエラーメッセージとして表示することができます。

- メッセージは、Spring の messageSource として読み込むプロパティファイルに定義します。
  定義方法などは、「2.4.3　アプリケーション用(共通の)Spring Bean ファイル」を参照してください。
- メッセージコードは、形式が決まっており「typeMismatch.{キー名 }」とします。
  これらのメッセージは、「org.springframework.validation. DefaultMessageCodesResolver」で処理されます。
  キー名の指定方法により、型に対してのメッセージや、プロパティ名（=フィールド名）に対するメッセージなど優先順位を決めて指定することができます（表 4.2)。また、プロパティファイルでの定義順は関係なく、メッセージコードの形式により一致します。。

表 4.2　型変換のメッセージコードと優先度

優先度	メッセージコードの形式	説明
1	typeMismatch.[Command 名].[フィールド名]	特定の Command のフィールド名に一致する場合のメッセージです。 あまり使用する機会はないと思います。
2	typeMismatch.[フィールド名]	フィールド名（プロパティ名）と一致する場合のメッセージです。 日付型など特定のフォーマットの場合に使用します。
3	typeMismatch.[形名]	フィールドのクラス型と一致する場合のメッセージです。 通常はこの形式を使用します。
4	typeMismatch	優先度の高いメッセージコードに該当するものがない場合に一致します。 必ず記述しておく必要があります。

※DefaultMessageCodesResolver の Javadoc にも詳しく記載されています。

4.2.1. データバインドのエラーメッセージのサンプル

エラーメッセージのサンプルを示します。

```
## オブジェクト名に対するエラーメッセージ
typeMismatch.userCommand.name=ユーザ名の形式で入力してください。

## フィールド名に対するエラーメッセージ
typeMismatch.dateTime=日時の形式（yyyy/MM/dd HH:mm）で入力してください。
typeMismatch.date=日付の形式（yyyy/MM/dd）で入力してください。
typeMismatch.time=時刻の形式（HH:mm）で入力してください。


## 型名に対するエラーメッセージ（プリミティブ型）
typeMismatch.short=整数で入力してくだし。
typeMismatch.java.lang.Short=整数で入力してください。
typeMismatch.int=整数で入力してください。
typeMismatch.java.lang.Integer=整数で入力してください。
typeMismatch.long=整数で入力してください。
typeMismatch.java.lang.Long =整数で入力してください。
typeMismatch.float=小数で入力してください。
typeMismatch.java.lang.Float=小数で入力してください。
typeMismatch.double=小数で入力してください。
typeMismatch.java.lang.Double=小数で入力してください。
typeMismatch.boolean=ブール値(true、false)で入力してください。
typeMismatch.java.lang.Boolean=ブール値(true、false)で入力してください。

## フィールドの型に対するエラーメッセージ（その他の型）
typeMismatch.java.sql.Timestamp=日時の形式（yyyy-MM-dd HH:mm:ss.SSS）で入力してください。

## 型変換のエラーメッセージ
typeMismatch=入力形式が不正です。
```

4.2.2. List 型、Map 型のバインド時のエラーメッセージ

List 型、Map 型などの場合も指定することができます。例を下記に示します。これは、コマンド“userCommand”に対して、リスト形式のフィールド“List<Group>”を持ち、Group の中にプロパティ“name”を持つ場合に一致するエラーメッセージを示したものです。

例をみるとわかると思いますが、JSP で Command のプロパティをバインドする path の指定形式と同じであることがわかります（「4.5.1　プロパティの位置(=path)の表現」）。

```
## List,Map 型の形式に
typeMismatch.userCommand.groups[0].name=優先度 1
typeMismatch.userCommand.groups.name=優先度 2
typeMismatch.groups[0].name=優先度 3
typeMismatch.groups.name=優先度 4
typeMismatch.name=優先度 5
typeMismatch.java.lang.String=優先度 6
typeMismatch=優先度 7
```

図 4.5　List、Map 形式に対するデータバインド時のエラーメッセージ

4.2.3. 項目名を埋め込む

- エラーメッセージの**置換文字{0}**は、フィールド名（=プロパティ名）が自動的に設定されます。しかし、フィールド名は変数名なので、画面上は英数字が表示され日本語サイトでは不格好です。
- プロパティファイルにフィールド名も定義すると、メッセージの置換文字を入れ替えることができます。

```
# バインドエラーのメッセージ定義
## 型名に対するエラーメッセージ（プリミティブ型）
typeMismatch.int={0}は、整数で入力してください。
typeMismatch.java.lang.Integer={0}は、整数で入力してください。

## 型変換のエラーメッセージ
typeMismatch={0}は、入力形式が不正です。

# フィールド名の定義
age=年齢
```

フィールド名の定義。

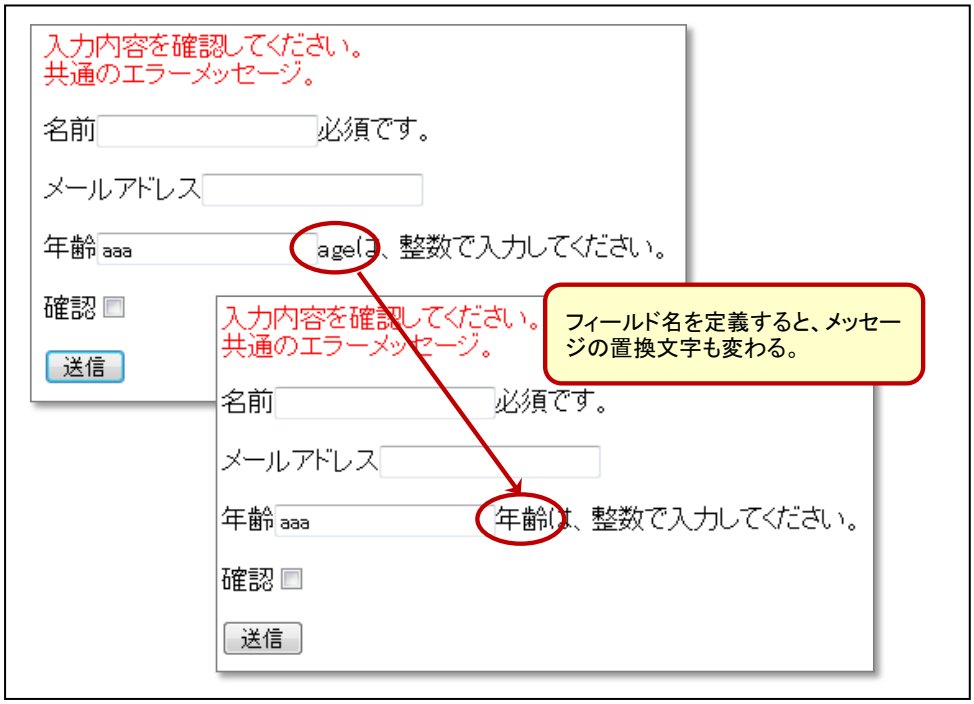


図 4.6　フィールド名を定義した場合のエラーメッセージ

Validator によるチェック（「7.1　Errors クラスを使用した入力値検証」）によるフィールドエラーのメッセージ形式は、**バインドエラーとは異なり置換文字{0}にフィールド名は保持ません**。

フィールド名は、カスタムタグ<form:error path="フィールド名">を記述する時点で分かっているので、メッセージの書式を統一するため、**<spring:message code="フィールド名">により項目名を出力する**ことをお勧めします。

4.3. 独自のデータ型のバインド（@InitBinder）

Date 型へのバインドを行う場合、フォーマットとして「yyyy/MM/dd」なのか、「yyyy-MM-dd」なのか、運用によって様々あります。Sprin MVC では、データのバインド方法を容易にカスタマイズできます。

4.3.1. 日付型のバインド（CustomDateEditor）

【Command の作成】

- Command に、「java.util.Date」クラスのプロパティを定義します。通常の Command と変わりません。

```
import java.io.Serializable;
import java.util.Date;

import org.apache.commons.lang.builder.ToStringBuilder;

public class CustomizeDateBindCommand implements Serializable {

	private static final long serialVersionUID = 1L;

	private Date startDate;

	private Date endDate;

	public CustomizeBindCommand() {
	}

	// getter、setter は省略

	@Override
	public String toString() {
		return ToStringBuilder.reflectionToString(this);
	}
}
```

【Controller の作成】

- @InitBinder を付加したメソッド、initBinder()を定義します。Command と JSP の入力項目のバインド方法をカスタマイズするメソッドです。
  引数として、「org.springframework.web.bind.WebDataBinder」を取ります。
- WebDataBinder#registerCustomEditor()で、Command のプロパティを操作する PropertyEditor の 1 種である、CustomDateEditor を追加します。関連付け方法は、クラス指定とプロパティ名を指定した方法があります。
- 他は通常と変わりません。データバインド時のエラーを検出するために、「@ModelAttribute」をメソッドに付与したものを用意します。詳細は、「4.2　データバインドエラー（型ミスマッチ）処理」を参照してくださし。

```
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping(value="/test/customizeDateBind")
public class CustomizeDateBindController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {

        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
        dateFormat.setLenient(false);

        // 型を指定した Bind 設定
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));
        // プロパティ名を指定した Bind 設定
        binder.registerCustomEditor(Date.class, "endDate", new CustomDateEditor(dateFormat, true, 10));
    }

    @ModelAttribute("command")
    public CustomizeDateBindCommand createInitCommand() {
        CustomizeBindCommand command = new CustomizeDateBindCommand();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {
        CustomizeDateBindCommand command = createInitCommand();
        // 開始日付の初期値は現在の日付
        command.setStartDate(new Date());

        model.addAttribute("command", command);
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(@ModelAttribute("command") CustomizeDateBindCommand command,
            BindingResult bindingResult) {

        System.out.println(command.toString());

        // バインドエラーの場合
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());
            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

データバインドの型チェックを行うための SimpleDateFormat を生成する。

CustomEditor の 1 種である、日付用のクラスを追加する。

データバインド時のエラー検出のために、必ず必要。

開始日付の初期値を現在の日付とする。

【JSP の作成】

- 通常の項目定義方法と変わりません。

```
<h4>バインド方法をカスタマイズする</h4>
<form:form modelAttribute="command" action="${appUrl}/test/customizeDateBind.html" method="post">

    <p>
        <form:label path="startDate">開始日付</form:label>
        <form:input path="startDate" />
        <form:errors path="startDate" cssClass="errors" />
    </p>

    <p>
        <form:label path="endDate">終了日付</form:label>
        <form:input path="endDate" />
        <form:errors path="endDate" cssClass="errors" />
    </p>

    <input type="submit"/>
</form:form>
```

【エラーメッセージの定義】

データバインド時のエラーを定義します。詳細は、「4.2　データバインドエラー（型ミスマッチ）処理」を参照してください。

- フィール名指定、クラス指定を定義します。
  今回の場合、開始日付（=startDate）は、フィールド名指定のエラーメッセージになります。終了日付（=endDate）の場合はクラス名指定のエラーメッセージなります。
- それぞれのフィールド名も定義します。

```
# フィールド名に対するエラーメッセージ
typeMismatch.startDate={0}は、日付の形式（yyyy/MM/dd）で入力してください。

# フィールドの型に対するエラーメッセージ（その他の型）
typeMismatch.java.util.Date={0}は、日付の形式が不正です。

## 型変換のエラーメッセージ
typeMismatch={0}は、入力形式が不正です。

## フィールド名の定義
startDate=開始日付
endDate=終了日付
```

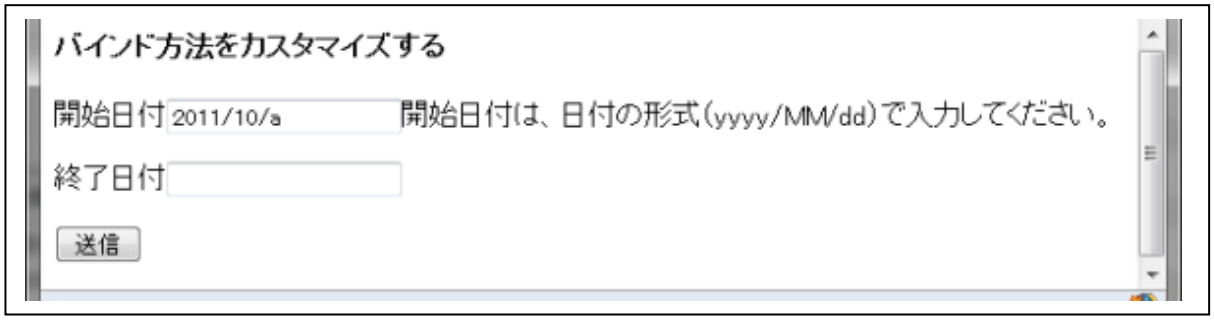
【注意】

- CustomEditor は、Command だけでなく、@RequestParam で取得した値にも適用可能です。

【ブラウザでの表示】

(1)初期表示。初期値として、現在の日付がフォーマットされ入っています。

(2)開始日付に間違った日付を入力すると、データバインド時のエラーメッセージが表示されます。また、存在しない日付「2011/13/01」のような場合、今回は、「SimpleDateFormat#setLenient(false)」としているため、エラーとなります。「SimpleDateFormat#setLenient(true)」と設定すると、「2011/13/01」の場合エラーとならず、日付が自動的に繰り越しなり、「2012/01/01」として処理されます。

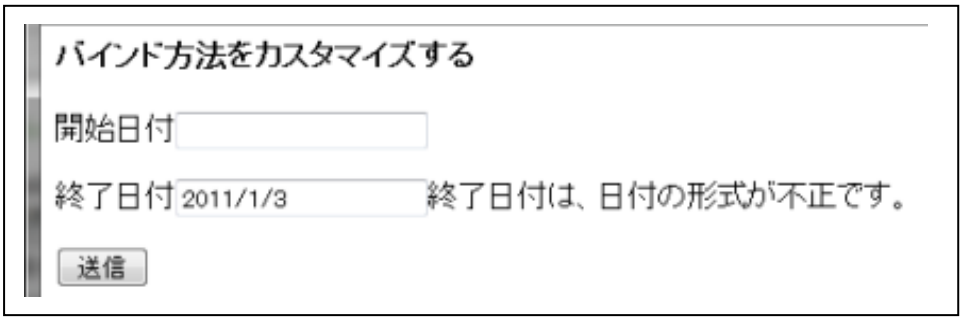


(3)開始日付に、1 桁の月、または日を入力した場合、自動的に“0”が補間され、送信されます。

・Controller 側で、CustomDateBindCommand#toString()の値をコンソールで出力した結果。

```
sample.web.test.controller.CustomizeDateBindCommand@62b59d[
startDate=Mon Jan 03 00:00:00 JST 2011,endDate=<null>]
```

(4)終了日付に、1 桁の月、または日を入力した場合、データバインドエラーとなります。これは、CutomDateEditor の引数に、入力値の文字数を 10 桁(「new CustomDateEditor(dateFormat, true, 10)」)に指定しているためです。
また、エラーメッセージは、クラス名で定義されたメッセージキー「typeMismatch.java.util.Date」に一致します。



4.3.2. 数値型のバインド（CustomNumberEditor）

CutsomNumberEditor を使用すると、数値型（Long、Double など）の特定のフォーマットを受け取ることができます。金額表示などの際に、3 桁ごとにカンマ「,」を入れる際などに使用します。

また、初期表示なども自動的にフォーマットされるため、Struts では ActionForm のプロパティにおいて String 型で受け取っていたものを、Spring MVC では数値型で受け取ることができます。

【Command の作成】

- Long、Double 型のプロパティを定義します。
  Integer、Float、BigDecimal でも定義できます。

```
public class CustomizeNumberBindCommand implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private Long amount;

    private Double average;

    public CustomizeNumberBindCommand() {

    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }

    // setter, getter は省略

}
```

【Controller の作成】

- CustomDateFomat と同様に、Formatter のインスタンスを WebDataBinder#registerCustomEditor() に渡します。
- CustomNumberEditor にもクラス型を指定する必要があり、その際に WebDataBinder に設定したクラス型を一致させる必要があります。

```
import org.springframework.beans.propertyeditors.CustomNumberEditor;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;

@Controller
@RequestMapping("/test/customizeNumberBind")
public class CustomizeNumberBindController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {

        // 名前を指定した Bind 設定
        NumberFormat amountFormat = NumberFormat.getInstance();
        binder.registerCustomEditor(Long.class, "amount",
                new CustomNumberEditor(Long.class, amountFormat, true));

        // 型を指定した Bind 設定
        DecimalFormat doubleFormat = new DecimalFormat("###,###.###");
        binder.registerCustomEditor(Double.class,
                new CustomNumberEditor(Double.class, doubleFormat, true));

    }

    // @RequestMapping などは省略

}
```

DataBinder に渡す数値のクラス型と CustomNumberEditor に渡す数値のクラスを一致させる必要があるので注意してください。

DecimalFormat も使用可能です。

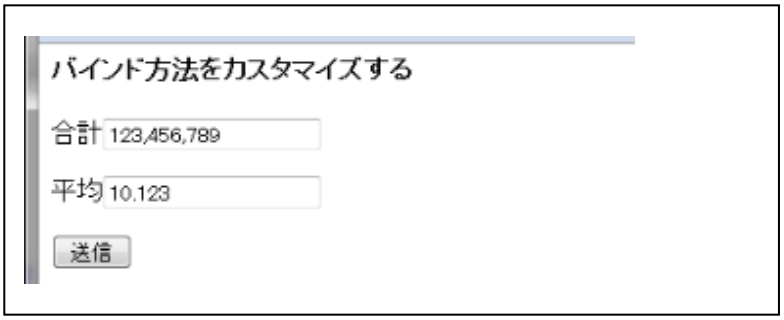
【JSP の作成】

- JSP は通常と同じようにプロパティを出力します。

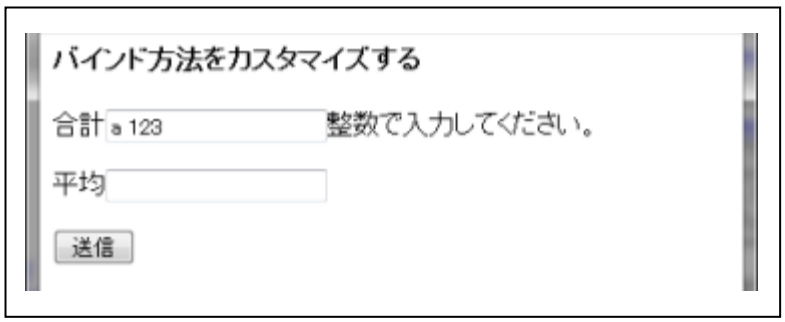
```
<h4>バインド方法をカスタマイズする</h4>
<form:form modelAttribute="command" action="${appUrl}/test/customizeNumberBind.html" method="post">

    <p>
        <form:label path="amount">合計</form:label>
        <form:input path="amount" />
        <form:errors path="amount" cssClass="errors" />
    </p>

    <p>
        <form:label path="average">平均</form:label>
        <form:input path="average" />
        <form:errors path="average" cssClass="errors" />
    </p>

    <input type="submit"/>
</form:form>
```

【ブラウザでの表示】

(1)初期表示。値が設定されている場合、フォーマットされ表示されます。



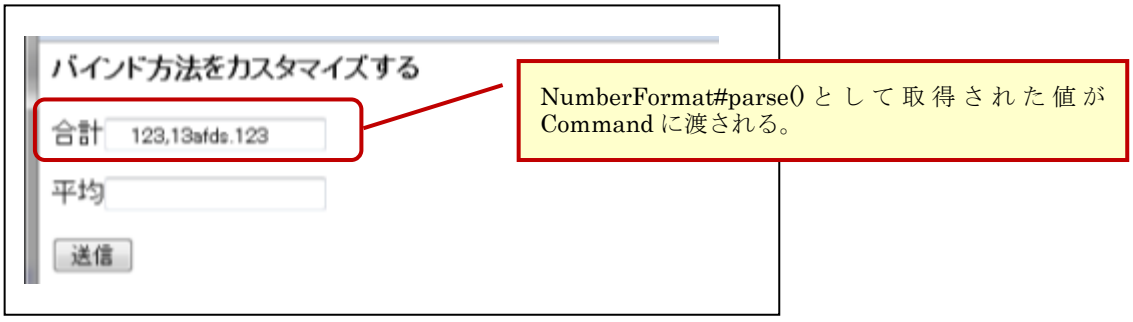
(2)不正な値を入力した場合。バインドエラーとなりメッセージが表示されます。



(3)先頭が空白で、途中に不正な英字文字が含まれる場合、バインドエラーとなりません。その場合、先頭の空白は無視され、途中の英字がある部分だけ送信されます。

これは、NumberFomat#parse( )の仕様として、「文字の先頭から解析し、解析できない文字があった場合、途中までの結果を返し、例外はスローしない」ためです。詳細は、Javadoc を参照ください。

このようなケースをエラーとしたい場合、専用の PropertyEditor を作成しバインドします。詳細は、「4.3.2.1　正確なフォーマットの数値のみをバインドする」を参照ください。



・Controller 側で、CustomDateBindCommand#toString()の値をコンソールで出力した結果。

途中の数値部部だけ送信されていることがわかります。

```
sample.web.test.controller.CustomizeNumberBindCommand@f56dc8[amount=12313,average=<null>]
```

4.3.2.1. 正確なフォーマットの数値のみをバインドする

CustomNumberEditor は、NumberFormat#parse()が解析可能なものを変換するため、不正な英字が入っているものはエラーとならず取得できてしまいます。ここでは、このような不正な文字が渡された場合、バインドエラーとして処理する方法を説明します。方法は 2 つあります。

① 「NumberFormat を拡張し、不正な文字がわたってきた場合、ParseException をスローする。」

② 「CustomNumberEditor を拡張し、不正な文字がわたってきた場合、バインドエラーとする。」

ここでは、影響範囲が少ない②「CustomNumberEditor の拡張した方法」を紹介します。

【CustomExactNumberEditor.java の作成】

- メソッド「setAsText」をオーバーライドし、桁数、または正規表現でのチェックを追加します。
- 初期設定では、正規表現によるチェックを行います。数字、空白、特定の記号を許可します。
  正規表現を変更したい場合は、メソッド「setExactNumberPattern()」で設定します。利便性を考慮しメソッドチェーンに形式にします。
- プロパティ「allowEmty」に関しては、継承元の CustomNumberEditor では private 修飾となっているため、このクラスで別途保持するようにします。

```
import java.text.NumberFormat;
import java.util.regex.Pattern;

import org.springframework.beans.propertyeditors.CustomNumberEditor;
import org.springframework.util.StringUtils;

/**
 * 数値型を処理するプロパティエディタ。
 * <p>正規表現、桁数などを元に、正確なチェックを行う。
 *
 */
public class CustomExactNumberEditor extends CustomNumberEditor {

    /** 「数字(半角、全角)」、空白、記号(.-E;%) */
    protected static final String EXACT_NUMBER_PATTERN = "[¥¥d０-９¥¥s#¥¥.¥¥-,E;%]*";

    /** 空白を許可するか */
    protected boolean allowEmpty;

    /** フォーマットチェック用の正規表現 */
    protected Pattern exactNumberPattern = Pattern.compile(EXACT_NUMBER_PATTERN);

    /** 桁数（0 以下の場合は、桁数チェックは行わない） */
    protected int exactNumberLength;

    @SuppressWarnings("rawtypes")
    public CustomExactNumberEditor(Class numberClass, boolean allowEmpty)
            throws IllegalArgumentException {
        super(numberClass, allowEmpty);
        this.allowEmpty = allowEmpty;
    }

    @SuppressWarnings("rawtypes")
```

```
    public CustomExactNumberEditor(Class numberClass, NumberFormat numberFormat, boolean allowEmpty)
            throws IllegalArgumentException {
        super(numberClass, numberFormat, allowEmpty);
        this.allowEmpty = allowEmpty;
    }

    /**
     * 文字列から数値オブジェクトに変換する。
     */
    @Override
    public void setAsText(final String text) throws IllegalArgumentException {

        if(this.allowEmpty && !StringUtils.hasText(text)) {
            setValue(null);
        } else if(text != null
                && this.exactNumberLength >= 0
                && text.length() != this.exactNumberLength) {
            throw new IllegalArgumentException(
                    "Could not parse number : it is not exactly length " + this.exactNumberLength);
        } else if(text != null
                && this.exactNumberPattern != null
                && !exactNumberPattern.matcher(text).matches()) {
            throw new IllegalArgumentException(
                    "Could not parse number : it is not mutch pattern " + this.exactNumberPattern.pattern());
        }

        super.setAsText(text);

    }

    public CustomExactNumberEditor setExactNumberLength(int exactNumberLength) {
        this.exactNumberLength = exactNumberLength;
        return this;
    }

    public CustomExactNumberEditor setExactNumberPattern(Pattern exactNumberPattern) {
        this.exactNumberPattern = exactNumberPattern;
        return this;
    }

    public CustomExactNumberEditor setExactNumberPattern(String exactNumberPattern) {
        final Pattern pattern = exactNumberPattern == null ? null : Pattern.compile(exactNumberPattern);
        return setExactNumberPattern(pattern);
    }
}
```

桁数によるチェックを行います。
CustomDateEditor を参考にしています。

正規表現によるチェックを行います。

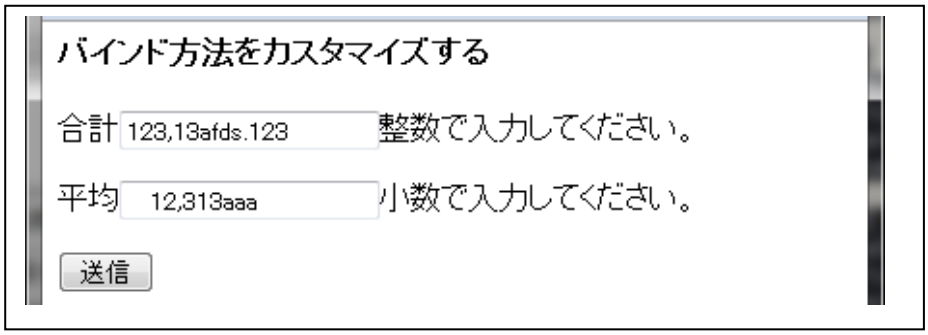
【使い方】

- コンストラクタのインタフェースは変更していないため、CustomNumberEditor から使い方は変わっていません。
- チェックパターンの正規表現を変更したい場合は、「setExactNumberPattern()」で変更します。

```
@InitBinder
protected void initBinder(WebDataBinder binder) {

    // 名前を指定した Bind 設定
    NumberFormat amountFormat = NumberFormat.getInstance();
    binder.registerCustomEditor(Long.class, "amount",
            new CustomExactNumberEditor(Long.class, amountFormat, true));

    // 型を指定した Bind 設定
    DecimalFormat doubleFormat = new DecimalFormat("###,###.###");
    binder.registerCustomEditor(Double.class,
            new CustomExactNumberEditor(Double.class, doubleFormat, true)
                .setExactNumberPattern("[¥¥d,¥¥.]*"));


}
```

【ブラウザの表示例】

途中に英数字を含んでいた場合、バインドエラーとなるようになりました。



4.3.3. CustomEditor の名前による関連付け方法（path の指定方法）

WebDataBinder は、Command のプロパティ名（=フィールド名）を指定し設定するメソッド「registerCustomEditor(Class requiredType, String field, PropertyEditor propertyEditor)」があります。

バインドエラーメッセージのように、Command の中にリストを持つような場合でも指定することができます。また、一致する優先度が決まっています。

- プロパティ名とクラス型を比較した場合、「表 4.3」に示すように、プロパティ名による関連付けが優先されます。

表 4.3　CustomEditor の適用される優先度

優先度	バインドの関連付け方法	説明
1	プロパティ名による指定	特定の Command のフィールドに一致する場合。 階層が、「Command.getA().getB().getC()」で取得するようなプロパティの場合、「A.B.C」と指定する。 パスは絶対パスで定義し、「B.C」のように先頭を省略できない。
2	クラスの型による指定	フィールドのクラス型と一致する場合。

4.3.3.1. CustomEditor の関連付けのサンプル

【Command】

- List<CustomizeItemBean>のように JavaBean のリスト形式のプロパティ「items」を持ちます。

```
public class CustomizeBindCommand implements Serializable {
	// プロパティ
	private Double average;
	private Date startDate;
	private List<CustomizeItemBean> items;

	@SuppressWarnings("rawtypes")
	public CustomizeBindCommand() {
		items = ListUtils.lazyList(
				new ArrayList(),
				FactoryUtils.instantiateFactory(CustomizeItemBean.class));
	}
	// getter, setter は省略
}

public class CustomizeItemBean implements Serializable {
	// プロパティ
	private Double processTime;
	private Date updateTime;
	public String toString() {
	}
	// getter, setter は省略
}
```

【JSP】

```
<h4>バインド方法をカスタマイズする</h4>
<form:form modelAttribute="command" action="${appUrl}/test/customizeBind.html" method="post">
    <p>
        <form:label path="startDate">開始日付</form:label>
        <form:input path="startDate" />
        <form:errors path="startDate" cssClass="errors" />
    </p>
    <p>
        <form:label path="average">平均</form:label>
        <form:input path="average" />
        <form:errors path="average" cssClass="errors" />
    </p>

    <table border="1">
        <tr>
            <th>処理時間</th><th>更新日時</th>
        </tr>
        <c:forEach items="${command.items}" var="item" varStatus="itemStatus">
        <tr>
            <td>
                <form:input path="items[${itemStatus.index}].processTime" />
                <form:errors path="items[${itemStatus.index}].processTime" cssClass="errors" />
            </td>
            <td>
                <form:input path="items[${itemStatus.index}].updateTime" />
                <form:errors path="items[${itemStatus.index}].updateTime" cssClass="errors" />
            </td>
        </tr>
        </c:forEach>
    </table>

    <input type="submit"/>
</form:form>
```

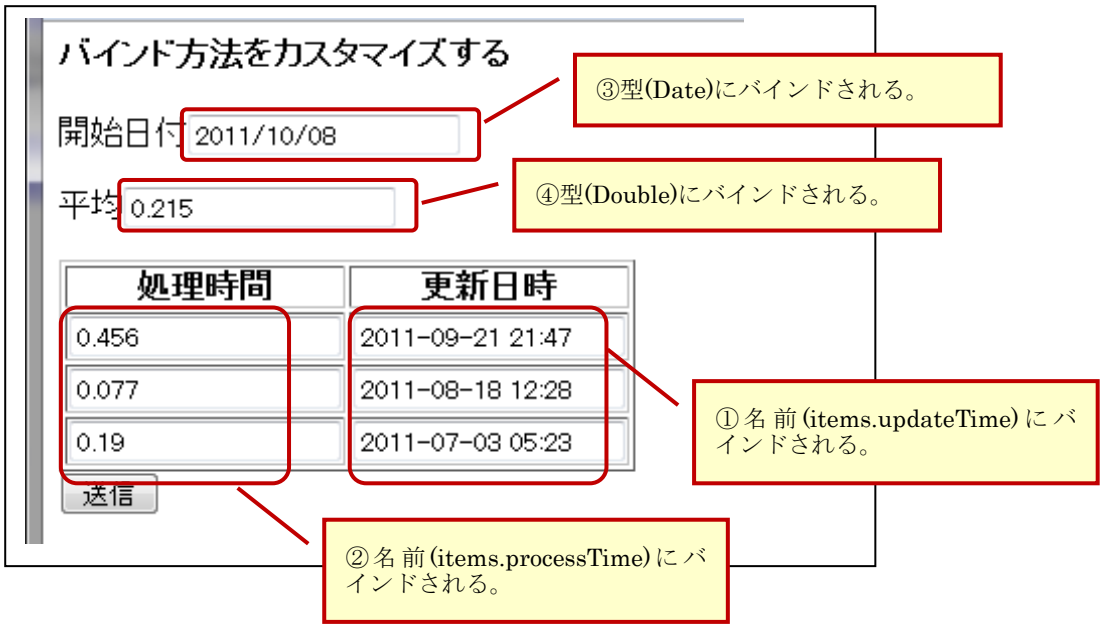
【Controller(@initBinder)】

```
@Controller
@RequestMapping("/test/customizeBind")
public class CustomizeBindController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        // ①名前(items.updateTime)を指定した Bind 設定
        SimpleDateFormat updateTimeFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        updateTimeFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, "items.updateTime",
                new CustomDateEditor(updateTimeFormat, true));

        // ②名前(items.processTime)を指定した Bind 設定
        DecimalFormat processTimeFormat = new DecimalFormat("0.000");
        binder.registerCustomEditor(Double.class, "items.processTime",
                new CustomNumberEditor(Double.class, processTimeFormat, true));

        // ③型(Date)を指定した Bind 設定
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));

        // ④型(Double)を指定した Bind 設定
        NumberFormat doubleFormat = NumberFormat.getInstance();
        binder.registerCustomEditor(Double.class, new CustomNumberEditor(Double.class, doubleFormat, true));

    }

    // 他のメソッドは、省略
}
```

「updateTime」など items を省略することはできない。

エラーコードのように、「items[0].updateTime」も指定できるが、画面表示時はバインドされるが、データ受信時はバインドされない。

【ブラウザでの表示】



4.3.4. システム全体のバインドの設定

Date 型などシステムで統一した書式を持つ場合、各 Controller で毎回設定するのは面倒です。そこで、システムで予め CustomEditor を登録しておくこともできます。流れは以下の通りです。

① インタフェース「WebBindingInitializer」の実装クラスを作成します。
　実装クラスの中で、CustomEditor を登録します。

② 作成した WebBindingInitializer の実装クラスを、Spring Bean に登録するため、「servlet-context.xml」に設定を記述します。

【WebBindingInitializer の実装クラスの作成】

- @InitBinder と同様に、引数に WebDataBinder を持つので、CutsomEditor を登録します。
　通常は、クラスの型による指定を追加し、プロパティ名による指定は追加しません。

```
package sample.web;

import java.text.SimpleDateFormat;
import java.util.Date;

import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebBindingInitializer;
import org.springframework.web.context.request.WebRequest;

public class GlobalBindingInitializer implements WebBindingInitializer {

    @Override
    public void initBinder(WebDataBinder binder, WebRequest request) {

        //型(Date)を指定した Bind 設定
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));
    }
}
```

【servlet-context.xml の編集】

- AnnotationMethodHandlerAdapter のプロパティ webBindingInitializer に作成したクラスを登録します。

```
<beans>
・・・省略
    <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
        <property name="cacheSeconds" value="0" />
        <property name="webBindingInitializer">
            <bean class="sample.web.GlobalBindingInitializer" />
        </property>
    </bean>

    <mvc:annotation-driven />
・・・省略
</beans>
```

必ず、<mvc:annotation-driven/>より前に記述します。
後に記述した場合動作しません。

4.3.5. 様々な CustomEditor（PropertyEditor）

Spring には、予め CustomEditor が用意されており、「org.springframework.beans.propertyeditors」パッケージに格納されています。もともとは、JavaBean のプロパティの値を変換するための機構であるため、「java.beans.ProperyEditor」クラスをインタフェースに持ちます。
ここでは、Spring MVC でよく使用するものを「表 4.4」に示します。

表 4.4　SpringFramework で用意されている CustomEditor(PropertyEditor)

No.	クラス名	登録済（※1）	説明
1	CustomBooleanEeditor	○	Boolen 型を処理する。 デフォルトで、「"true"/"on"/"yes"/"1"」「"false"/"off"/"no"/"0"」を処理する。 コンストラクタでカスタマイズ可能。 「CustomBooleanEditor(String trueString, String falseString, boolean allowEmpty)」
2	CustomDateEditor	×	java.util.Date 型を処理する。 詳細は、「4.3.1　日付型のバインド（CustomDateEditor）」を参照。
3	CustomNumberEditor	○	Integer、Long、Float、Double、BigDecimal のような Number のサブクラスを処理可能。 詳細は、「4.3.2　数値型のバインド（CustomNumberEditor）」を参照。
4	StringArrayPropertyEditor	○	カンマで区切られた文字列型を String[]へと処理する。 コンストラクタで区切り文字が変更可能。
5	StringTrimmerEditor	×	文字列をトリミングするプロパティエディタ。コンストラクタの指定で、空文字を null に変換することも可能。
5	URLEditor	○	URL の文字列表現を実際の URL へと処理する。

※1 デフォルトで登録されているかどうか。

その他、「LocaleEditor」「FileEditor」「CustomMapEditor」「CustomCollectionEditor」があります。PropertyEditor の説明については「http://andore.com/money/trans/spring_ref_p6_ja.html」を参照してください。

4.3.6. 列挙型のバインド

- 列挙型の場合、Enum#valueOf()によって文字列⇒列挙型に変換します。
- 空白を許可するかどうかなど、コンストラクタでオプションとして指定します。

【CustomEnumEditor.java】

```
package sample.web.test.controller;

import java.beans.PropertyEditor;
import java.beans.PropertyEditorSupport;
import java.util.EnumSet;
import java.util.Set;

import org.springframework.util.StringUtils;


/**
 * 列挙型の PropertyEditor
 * @see http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6219769
 */
public class CustomEnumEditor<E extends Enum<E>> extends PropertyEditorSupport implements PropertyEditor
{

	protected Class<E> clazz;
	protected String[] tags;
	protected E value;

	/** 空白の場合を許可するかどうか */
	protected boolean allowEmpty;
	/** 列挙型に含まない場合を考慮するかどうか */
	protected boolean allowInvalidValue;

	public CustomEnumEditor(Class<E> clazz, boolean allowEmpty) {
		this(clazz, allowEmpty, false);
	}

	public CustomEnumEditor(Class<E> clazz, boolean allowEmpty, boolean allowInvalidValue) {
		this.clazz = clazz;
		this.allowEmpty = allowEmpty;
		this.allowInvalidValue = allowInvalidValue;

		Set<E> allEnumValues = EnumSet.allOf(clazz);
		tags = new String[allEnumValues.size()];
		int i = 0;
		for (E enumValue : allEnumValues) {
			tags[i++] = enumValue.name();
		}
	}

	@SuppressWarnings("unchecked")
	@Override
	public E getValue() {
		return (E) super.getValue();
	}

	public void setValue(E value) {
		super.setValue(value);
```

```
    }

    @Override
    public void setAsText(final String text) throws IllegalArgumentException {

        if (this.allowEmpty && !StringUtils.hasText(text)) {
            setValue(null);
        } else {

            try {
                setValue(Enum.valueOf(clazz, text));
            } catch (IllegalArgumentException e) {
                if(this.allowInvalidValue) {
                    setValue(null);
                } else {
                    throw e;
                }
            }
        }
    }

    @Override
    public String getAsText() {
        if (getValue() == null) {
            return null;
        }
        return getValue().name();
    }

    @Override
    public String[] getTags() {
        return tags;
    }
}
```

空白を許可するかどうかの判定。

valueOf でオブジェクトに変換する。
失敗する場合は例外が発生する。

【Controller の例】

```
// 列挙型の定義
public enum ColorName {
    RED, ORANGE, BLUE;
}

// Controller
@Controller
@RequestMapping("/test/customizeBind")
public class CustomizeBindController {

    @InitBinder()
    protected void initBinder(WebDataBinder binder) {

        //列挙型
        binder.registerCustomEditor(ColorName.class,
                new CustomEnumEditor<ColorName>(ColorName.class, true));

    }
}
```

4.3.7. アノテーションを使用したデータバインド(:TODO)

「org.springframework.format.annotation」の@DateTimeFormat、@NumberFormat を紹介する。

DateBind というより、Spring Format API。

4.3.7.1. アノテーションの自作(:TODO)

4.4. ファイルアップロード

ファイルアップロードの方法として、「Commons FileUpload」と「COS」を使用する 2 つの方式があります。また、Servlet3.0(Tomcat7)を利用する場合は、外部の API が必要なくなりました。ここでは、利用されている数やライセンスの使い勝手の良い Commons FileUpload、Servlet3.0 を使った方式を説明します。

4.4.1. ファイルアップロードの準備（Commons FileUpload）

① ライブラリ「Commons FileUpload」を準備します。pom.xml にライブラリを追加します。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <dependencies>
        ・・・省略

        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
            <version>1.2.2</version>
        </dependency>

        ・・・省略
    </dependencies>
</project>
```

図 4.7　ファイルアップロード用のライブラリの追加（pom.xml）

② Spring MVC の設定ファイル「servlet-context.xml」にマルチパートを解釈する設定を追加します。
その際にアップロード可能な最大サイズを設定し、この値は外部ファイルで定義しておくと便利です（「2.4.3　アプリケーション用(共通の)Spring Bean ファイル」で示した”propertyConfigurer”を参照)。
このサイズは一度にアップロード可能な最大サイズであるため、複数ファイルをアップロードする場合は注意してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans">
・・・省略

    <!-- use file upload -->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!-- アップロード可能な最大ファイルサイズ（単位はバイト） -->
        <property name="maxUploadSize" value="${app.fileupload.maxSize}"/>
    </bean>

</beans>
```

設定値を外部ファイルに定義しておくと、運用の際に便利です。
単位は、byte で指定します。

図 4.8　ファイルアップロードのための multiparrtResolver の追加（servlet-context.xml）

4.4.2. ファイルアップロードの準備（Spring MVC 3.1＋Servlet3.0 のマルチパート機能）

Spring3.1 以上でかつ Servlet3.0(Tomcat7)のマルチパート機能を利用する場合の説明をします。Spring3.1 でも Commons FileUpload を利用することができます。その場合は、「4.4.1　ファイルアップロードの準備（Commons FileUpload）」を参照してください。

① Servlet3.0 が利用可能な環境設定をします。

- 「1.1　Servlet 3 に対応」を参照してください。

② 「web.xml」の Spring の DispatcherServlet に<multipart-config>の記述を追加します。

- Servlet3.0 のマルチパートの設定は、サーブレットごとに行います。Spring MVC の DispacherServlet も Servlet の一種のため、設定可能です。
- <multipart-config>を設定するには、Servlet 3.0 のアノテーション「@MultipartConfig」でも設定可能ですが、DispacherServlet はフレームワークで準備されているクラスなので、web.xml のタグを利用するしかありません。

```
<web-app>
・・・省略
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/servlet-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <multipart-config>
            <max-file-size>1000000</max-file-size>
            <max-request-size>1000000</max-request-size>
            <file-size-threshold>1000000</file-size-threshold>
        </multipart-config>
    </servlet>
・・・省略
</web-app>
```

・SpringMVC の共通 Servlet で、マルチパートの設定を行います。
・サイズの単位は、byte です。

図 4.9　ファイルアップロードのためのマルチパートの設定値の追加（web.xml）

表 4.5　<multipart-config>で設定可能な子要素

No.	項目	説明
1	<location>String</location>	アップロードしたファイルの一時的な配置場所を指定します。 ・省略可能です。空文字を設定と同義です。 ・省略した場合、デフォルトの場合、APP サーバの一時フォルダに格納されます。Tomcat の場合は、「${CATALINA_HOME}/temp/」が一時フォルダとなります。。
2	<max-filse-size>long<max-file-size>	アップロード可能なファイルサイズ。 ・単位はバイト。 ・デフォルト“-1”で、この設定すると無制限になる。
3	<max-request-size>long</max-request-size>	「multipart/form-data」としてリクエスト可能な最大サイズ。 ・単位はバイト。 ・デフォルト“-1”で、この値を設定すると無制限になる
4	<file-size-threshold>int<<file-size-threshold>	<file-size-threshold>より大きいサイズのファイルがアップロードされた場合、一時ファイルがこの値で分割され処理されます。 ・アップロードリクエストが完了すると、一時ファイルは削除されます。

③　Spring MVC の設定ファイル「servlet-context.xml」にマルチパートを解釈する設定を追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans">
・・・省略
    <!--  ↓Servlet 3 のマルチパート API を使用する場合  -->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.support.StandardServletMultipartResolver">
    </bean>
</beans>
```

図 4.10　ファイルアップロードのための multiparrtResolver の追加（servlet-context.xml）（Servlet3.0）

4.4.3. 単純なファイルアップロード

ファイルを 1 つアップロードする方法を説明します。

【JSP】

- form の属性に「enctype="multipart/form-data"」を追加します。
- ファイルアップロードの項目<input type="file" name="プロパティ名" />を追加します。
  name 属性は、Command や@RequestParam の名称と一致させます。
- Spring 用のカスタムタグを使用したい場合は、テキストフィールド用に属性 type="file"を追加します。
  <form:input path="file" type="file" />
- 初期値は設定できません。これは、セキュリティ上問題があるためです。初期値を設定しても、ブラウザ自体で制限がかかるため無効になります。

```
<h4>ファイルを 1 つアップロードする</h4>
<form action="${appUrl}/test/fileupload/single.html" method="post" enctype="multipart/form-data">

    <input type="file" name="file" />

    <input type="submit"/>
</form>
```

【Controller】

- アップロードしたファイルは、「org.springframework.web.multipart.MultipartFile」で受け取ります。
  byte[]でも受け取れますが、メモリ上にすべて読み込むため性能に問題があります。また、MultipartFile の方が、予めメソッドがそろっており扱うには便利です（表 4.6　MultipartFile クラスの仕様）。
- Commmons FileUpload を使用している場合、MultiPartFile の実装クラスは、
  「org.springframework.web.multipart.commons.CommonsMultipartFile」です。
  Servlet API 3.0 を使用している場合、MultiPartFile の実装クラスは、「javax.servlet.http.Part」です。
- 図 4.7 の servlet-context.xml にて設定した上限値を超えてサイズをアップロードした場合、例外「org.springframework.web.multipart.MaxUploadSizeExceededException」がスローされます。
  Servlet 3.0 の場合は、「org.springframework.web.multipart.MultipartException」がスローされます。

```
@Controller
@RequestMapping("/test/fileupload")
public class FileuploadController {

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

    }

    // 1 つのファイルをアップロードする。
    @RequestMapping(value="single", method=RequestMethod.POST)
    public ModelAndView doAction(@RequestParam("file") MultipartFile file)
throws IllegalStateException, IOException {
```

空のサイズのファイルどうかの判定は、「 !file.getOriginalFilename().isEmpty() && file.isEmpty()」で行います。

```
        if(!file.getOriginalFilename().isEmpty() && !file.isEmpty()) {
            File uploadFile = new File("d:/upload/", file.getOriginalFilename());
            file.transferTo(uploadFile);

            ModelAndView mav = new ModelAndView("/test/complete");
            mav.addObject("filename", file.getOriginalFilename());
            mav.addObject("filesize", FileUtils.byteCountToDisplaySize(file.getSize()));
            return mav;

        } else {
            ModelAndView mav = new ModelAndView("/test/fail");
            return mav;
        }
    }
}
```

用意されているメソッド用いてサーバの別の場所に保存します。

表 4.6　MultipartFile クラスの仕様

No.	戻り値 メソッド	説明
1	byte[] getBytes( )	ファイルをバイト配列で取得します。 サイズが大きなファイルを取得する際には、メモリ上にロードされるため注意が必要です。
2	String getContentType( )	ContentType を取得します。 画像などの場合、「image/jpeg」を返します。
3	InputStream getInputStream( )	ファイルをストリームとして取得します。 Close 処理を実装する必要があります。
4	String getName( )	バインドした際のプロパティの名を取得します。 タグの<input type=”file” name=”プロパティ名”/>の属性 name の値です。
5	String getOriginalFilename( )	アップロードしたファイル名を取得します。 クライアント側でアップロードしたファイル名そのものです。 ファイルを選択していない場合、空文字が設定されます。
6	long getSize( )	ファイルサイズ（単位は byte）を取得します。 メソッド「isEmpty()」が true を返す場合、サイズは 0 となります。 サイズのをフォーマット表示する場合、Commons IO の FileUtils#byteCountToDisplaySize()が便利です。
7	void transferTo(File dest)	ファイルをサーバ上の指定した任意の場所へ移動します。 一度移動すると、移動元のファイルは削除されます。
8	boolean isEmpty()	ファイルサイズが空かどうか判定します。 true を返す場合、ファイルサイズが空の場合以外に、ファイルを選択していない場合があります

4.4.4. リスト形式によるファイルアップロード

リストにする場合は、要素のインスタンスが必要となります。しかし、MultipartFile はインタフェースで、また、その実装クラス「CommonsMultipartFile（Servlet3.0 の場合は Part）」のコンストラクタを作成する際には、引数が必要であるため直接インスタンスを作ることはできません。

方法としては、List 型の要素には、MultipartFile をプロパティとして持つ JavaBeans を持たせます。

【Commad の作成】

- List の要素に、MultipartFile を直接格納するのではなく、JavaBeans を介して定義します。
- List のインスタンスは、画面で項目数が動的に増えることを考慮し、LazyList を使用します。

```
public class FileuploadCommand implements Serializable {
    private static final long serialVersionUID = 1L;

    private List<FileuploadItem> fileuploadItems;

    public FileuploadCommand() {
        fileuploadItems = ListUtils.lazyList(
                new ArrayList(),
                FactoryUtils.instantiateFactory(FileuploadItem.class));
    }

    public List<FileuploadItem> getFileuploadItems() {
        return fileuploadItems;
    }

    public void setFileuploadItems(List<FileuploadItem> fileuploadItems) {
        this.fileuploadItems = fileuploadItems;
    }
}
```

MultipartFile をプロパティに持つ、JavaBean のリスト。

動的に増えた場合を考慮し、Comons-Collectio の LazyList を使用しインスタンスを作成します。

図 4.11　ファイルアップロード用 JavaBen の List 型を持つ Command クラス

```
public class FileuploadItem implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;

    private MultipartFile file;

    public FileuploadItem() {
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }

    // 省略（setter、getter）
}
```

MultipartFile の定義

図 4.12　ファイルアップロードの List の要素のクラス

【JSP の作成】

- リスト型なので、JSTL の<c:forEach>～</c:forEarch>で要素を取り出します。
- テキストフィールドやファイルアップロードのフィールドは、Command のから見た位置を属性 path で表現します。リストの場合、インデックスを[ ]で表現します。
  リスト型によるデータのやり取りの詳細は、「4.5.2　リストによるデータの送受信」を参照してください。

```
<h4>ファイルを複数アップロードする</h4>
<form:form　modelAttribute="fileuploadCommand"　action="${appUrl}/test/fileupload/multi.html"　method="post"
enctype="multipart/form-data">

    <c:forEach items="${fileuploadCommand.fileuploadItems}" var="item" varStatus="itemStatus">
    <p>
        <form:input path="fileuploadItems[${itemStatus.index}].name" />
        <form:errors path="fileuploadItems[${itemStatus.index}].name" cssClass="errors" />

        <form:input path="fileuploadItems[${itemStatus.index}].file" type="file" />
        <form:errors path="fileuploadItems[${itemStatus.index}].file" cssClass="errors" />
    </p>
    </c:forEach>

    <input type="submit"/>
</form:form>
```

【Controller の作成】

```
@Controller
@RequestMapping("/test/fileupload")
public class FileuploadController {

    // command の初期オブジェクトの取得
    @ModelAttribute("fileuploadCommand")
    public FileuploadCommand createInitCommand() {
        FileuploadCommand command = new FileuploadCommand();
        return command;
    }

    //　初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        FileuploadCommand command = createInitCommand();
        // リストの初期サイズを 3 とする
        for(int i=0; i < 3; i++) {
            command.getFileuploadItems().add(new FileuploadItem());
        }
        model.addAttribute("fileuploadCommand", command);

    }

    // 複数のファイルをアップロードする。
    @RequestMapping(value="multi", method=RequestMethod.POST)
    public ModelAndView doAction2(@ModelAttribute("fileuploadCommand") FileuploadCommand command,
            BindingResult bindingResult) throws IllegalStateException, IOException {
```

Command の初期値を取得するメソッドです。

画面の初期表示用の Command を作成します。
3 つ要素を作りたい場合は、リストの要素を 3 つ作り、追加ます。

```
        // バインドエラーの場合
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());
            return mav;
        }

        for(FileuploadItem item : command.getFileuploadItems()) {

            MultipartFile file = item.getFile();
            if(file.isEmpty()) {
                continue;
            }

            File uploadFile = new File("d:/upload/", file.getOriginalFilename());
            file.transferTo(uploadFile);
        }

        ModelAndView mav = new ModelAndView("/test/complete");
        return mav;

    }

}
```

リストから各要素を取得し、処理を行います。

【ブラウザでの表示】

- HTML のソース「図 4.14」を見るとわかりますが、リストの要素 FileuploadItem ごとにインデックスが埋め込まれていることがわかります。

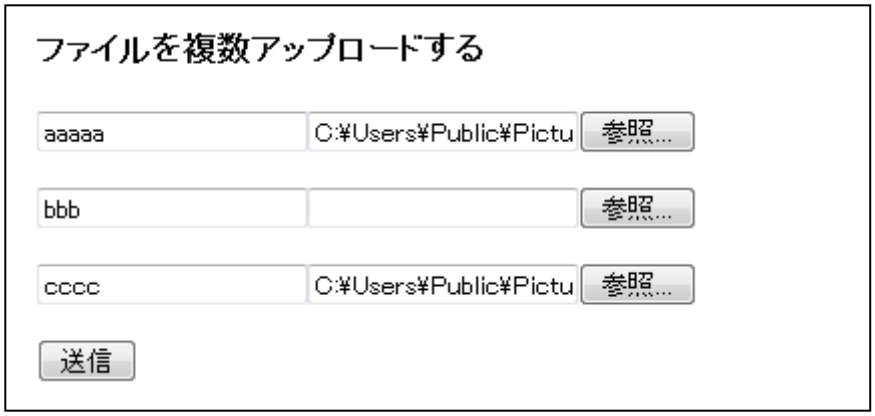


図 4.13　リスト形式のファイルアップロードのブラウザでの表示

```
<h4>ファイルを複数アップロードする</h4>

<form id="fileuploadCommand" action="/spring3-mvc/test/fileupload/multi.html" method="post" enctype="multipart/form-data">
    <p>
        <input id="fileuploadItems0.name" name="fileuploadItems[0].name" type="text" value=""/>
        <input id="fileuploadItems0.file" name="fileuploadItems[0].file" type="file" type="text" value=""/>
    </p>
    <p>
        <input id="fileuploadItems1.name" name="fileuploadItems[1].name" type="text" value=""/>
        <input id="fileuploadItems1.file" name="fileuploadItems[1].file" type="file" type="text" value=""/>
    </p>
    <p>
        <input id="fileuploadItems2.name" name="fileuploadItems[2].name" type="text" value=""/>
        <input id="fileuploadItems2.file" name="fileuploadItems[2].file" type="file" type="text" value=""/>
    </p>
    <input type="submit"/>
</form>
```

図 4.14　リスト形式のファイルアップロードの HTML のソース

4.4.5. ファイルサイズの上限値のを超えてアップロードした場合の処理

- ファイルの上限値を超えてアップロードした場合、「表 4.7　ファイルアップロードした際にスローされる例外」に示す例外が発生します。
- アップロード時の例外は、例外処理用アノテーション「@ExceptionHandler」ではキャッチできません。「@ExceptionHandler」はリクエストが Controller 内で発生した例外のみキャッチ可能です。
- アップロード時の例外を処理するには、共通の例外処理クラス「HandlerExceptionResolver」で処理します。詳細は、「11.2　システム全体での例外ハンドリング」を参照してください。

表 4.7　ファイルアップロードした際にスローされる例外

項目	Commons File Upload	Servlet3.0 のマルチパート
スローされる例外	org.springframework.web.multipart.MaxUploadSizeExceededException ※MultipartException のサブクラス	org.springframework.web.multipart.MultipartException
ラップされる例外	org.apache.commons.fileupload.FileUploadBase$SizeLimitExceededException	java.lang.IllegalStateException さらに次の例外がラップされている。 「org.apache.tomcat.util.http.fileupload.FileUploadBase$FileSizeLimitExceededException」。

4.5. リスト、マップなどの複雑なデータ構造を送受信する

Spring MVC では、Command の各プロパティにデータをバインドします。バインドしたプロパティにアクセスする際に、Spring では統一された表現を使用しています。

4.5.1. プロパティの位置(=path)の表現

表 4.8　プロパティの位置の表現

No.	プロパティの位置の表現	説明	参照
1	name	Java Bean のプロパティ「name」と一致します。 Getter(=getName())、Setter(=setName(…))経由でアクセスします。	―
2	account.name account.card.number	Java Bean のプロパティ「account」は、さらに Java Bean「Account」でネストしており、Account のプロパティ「name」を示します。 半角ピリオド「.」で区切ります。	―
3	account[2] account[2].card account[2][0].name	Java Bean のプロパティがリスト(=List<Account>)、配列(=Account[])の場合、各要素をインデックスで指定します。 0 以上の数値を指定します。 リストの中で、さらにネストしたり、2 次元リスト、配列を指定することもできます。	
4	account[APPLE] account[APPLE].title account[APPLE][0].name	Java Bean のプロパティがマップ(=Map<String,Account>)の場合、マップのキーを指定します。 キーは文字列(半角英数字)を使用します。 マップの中で、さらにネストしたりすることもできます。	

表 4.9　プロパティの位置表現を使用する箇所

No.	位置表現の仕様箇所	参照先
1	JSP において、form の各フィールドへのバインド指定。 カスタムタグ<spring:bind>、<form:XXX>を使用する。	本節 12.1.2 12.2
2	データバインド時のエラーメッセージ。	4.2
3	Command の各プロパティ固有のバインド方法の設定。	4.3
4	入力値検証時のエラーオブジェクトの作成。エラーメッセージの指定。	7.2

4.5.2. リストによるデータの送受信

リストにデータをバインドする際に、ライブラリ「commons-collections」のクラス「org.apache.commons.collections.list.LazyList」をインスタンスとして利用します。

- ブラウザの画面上で、JavaScript を使用し動的に項目を増やした場合に、サーバ側で処理するときに確保してあるリストサイズを超える項目数があると「ArrayOutOfBoundException」が発生します。
- LazyList は確保してあるサイズを超えてアクセスした場合、動的にサイズを増やすことで、例外が発生しなくなります。
- LazyList はリストの項目が JavaBean の場合のときに、インスタンスの生成方法を指定することができるため、複雑な Command を表現することができます。
- 配列型の場合はリスト型でプロパティを作成します。取り出す際に toArray()メソッドを使用し配列に変換します。。

4.5.2.1. リストの要素がプリミティブ型の場合

【Command の作成】

- LazyList のインスタンスは、ListUtils.layzList(第 1 引数, 第 2 引数)で取得します。
  - ➢ メソッドの第 1 引数には、リストのインスタンス「ArrayList」を指定します。
  - ➢ メソッドの第 2 引数には、リストの要素となるオブジェクトの Factory クラスを指定します。
    - ✧ 引数なしのデフォルトコンストラクタがある場合は、FactoryUtils.instantiateFactory()を使用します。
- Getter、Setter は通常のものを用意します。
  - ➢ Struts の ActionForm のように、各要素へのアクセス用のメソッド(setBooks(int i, String book)、getBooks(int i))は必要ありません。

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.apache.commons.collections.FactoryUtils;
import org.apache.commons.collections.ListUtils;
import org.apache.commons.lang.builder.ToStringBuilder;

public class Sample3Command implements Serializable {

    private List<String> books;

    @SuppressWarnings("unchecked")
    public Sample3Command() {

        books = ListUtils.lazyList(
                new ArrayList<String>(),
                FactoryUtils.instantiateFactory(String.class));
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
```

LazyList のインスタンスを取得します。

リストの要素のインスタンスを生成する Factory を指定します。
引数なしのコンストラクタを呼び作成する場合に使用します。

```
    }

    public List<String> getBooks() {
        return books;
    }

    public void setBooks(List<String> books) {
        this.books = books;
    }

}
```

setter、getter は通常のものを用意します。

【JSP の作成】

- JSTL の<c:forEach>を使用し、リストのプロパティ「books」を取り出します。
  - 属性「varStatus」にて、<c:forEach>の繰り返し情報を取得することができます。
  - リストのインデックス情報を「index」で取得することができるため、リストの位置表現の書式「プロパティ名[インデックス]」を作成します。
- 初期表示・再表示の際には、カスタムタグ<form:input>が自動的に Command(Java Bean)のプロパティにアクセスし値を取得します。
  - <c:forEach>は、リストのインデックス情報を取得するためだけに使用し、プロパティの値を取得する必要はありません。

```
<h4>入力値検証：リスト形式のデータ</h4>
<form:form modelAttribute="sample3Command" action="${appUrl}/test/sample3.html" method="post">
    <ul>
        <c:forEach items="${sample3Command.books}" var="book" varStatus="bookStatus">
        <li>
            <form:label path="books[${bookStatus.index}]">本（${bookStatus.index+1}）</form:label>
            <form:input path="books[${bookStatus.index}]" />
            <form:errors path="books[${bookStatus.index}]" cssClass="errors" />
        </li>
        </c:forEach>
    </ul>

    <input type="submit"/>
</form:form>
```

【Controller の作成】

- データバインド時に呼び出される Command のインスタンスを作成するためのメソッドを準備します。
  - メソッドに「@ModelAttribute("コマンド名")」を付与します。
- 初期表示(GET メソッドアクセス)の際の Command のインスタンスを作成するメソッドを準備します。
  - リストサイズが 0 だど、入力フィールドは作成されないので、最低限 1 以上の個数を作成します。

```
@Controller
@RequestMapping("/test/sample3")
public class Sample3Constroller {

    @Resource
    private Sample3Validator sample3Validator;

    @InitBinder("sample3Command")
    protected void initBinder(WebDataBinder binder) {
```

```
        binder.setValidator(sample3Validator);
    }

    @ModelAttribute("sample3Command")
    public Sample3Command createInitCommand() {
        Sample3Command command = new Sample3Command();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        Sample3Command command = createInitCommand();
        for(int i=0 ; i < 3; i++) {
            command.getBooks().add("");
        }
        model.addAttribute("sample3Command", command);
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample3Command") @Valid Sample3Command command,
            BindingResult bindingResult) {

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

データバインド時にデータを格納するためのインスタンスを取得するためのメソッド。

初期表示の際に、1 以上の空の項目を用意しておきます。
データがないと、入力フィールドが表示されません。

【HTML の表示：初期表示】

```
<h4>入力値検証：リスト形式のデータ</h4>
<form id="sample3Command" action="/spring3-mvc/test/sample3.html" method="post"><p>
    <ul>
        <li>
            <label for="books0">本（1）</label>
            <input id="books0" name="books[0]" type="text" value=""/>
        </li>
        <li>
            <label for="books1">本（2）</label>
            <input id="books1" name="books[1]" type="text" value=""/>
        </li>
        <li>
            <label for="books2">本（3）</label>
            <input id="books2" name="books[2]" type="text" value=""/>
        </li>
    </ul>

    <input type="submit"/>
</form>
```

4.5.2.2.　リストの要素が JavaBean（ネストしている）の場合

【Command の作成】

- リストの要素がプリミティブ型の時と同様、LazyList のインスタンスを、ListUtils.lazyList()で取得します。
- デフォルトコンストラクタで要素を作成する場合、FactoryUtils.instantiateFactory()を使用します。
- デフォルトコンストラクタがない場合や、要素の初期値を独自に設定したい場合は、「Factory」インタフェースのメソッド「create( )」を実装します。
  - ➢　初期値を設定する場合は、Controller の初期表示用メソッドで設定することをお勧めします。
- Getter、Setter は通常のものを用意します。

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.apache.commons.collections.Factory;
import org.apache.commons.collections.FactoryUtils;
import org.apache.commons.collections.ListUtils;
import org.apache.commons.lang.builder.ToStringBuilder;

public class Sample5Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private List<BookBean> books;

    @SuppressWarnings("unchecked")
    public Sample5Command() {
        // デフォルトコンストラクタでリストの要素のインスタンスを作成する場合
        books = ListUtils.lazyList(
                new ArrayList<BookBean>(),
                FactoryUtils.instantiateFactory(BookBean.class));

        // 独自の Factory を指定する場合
        books = ListUtils.lazyList(
                new ArrayList<BookBean>(),
                new Factory() {

                    @Override
                    public Object create() {
                        BookBean book = new BookBean();
                        // Bean の初期値の設定
                        book.setPrice(0);
                        return book;
                    }

                });
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
```

デフォルトコンストラクタを使用する場合。

独自の Factory を指定する場合。

```
    }
    public List<BookBean> getBooks() {
        return books;
    }

    public void setBooks(List<BookBean> books) {
        this.books = books;
    }
}
```

【リストの用の Java Bean の作成】

- プロパティ「authors」はさらにリスト構造を持ちます。
  - インスタンスには、LazyList を使用します。

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.apache.commons.collections.FactoryUtils;
import org.apache.commons.collections.ListUtils;
import org.apache.commons.lang.builder.ToStringBuilder;

public class BookBean implements Serializable{

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    protected String title;

    protected Integer price;

    protected List<String> authors;

    @SuppressWarnings("unchecked")
    public BookBean() {
        authors = ListUtils.lazyList(
                new ArrayList<String>(),
                FactoryUtils.instantiateFactory(String.class));
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
    // getter、setter は省略
}
```

【JSP の作成】

- JSTL の<c:forEach>を使用し、リストのプロパティ「books」を取り出します。
  - 属性「varStatus」にて、<c:forEach>の繰り返し情報を取得することができます。
  - リストのインデックス情報を「index」で取得することができるため、リストの位置表現の書式「プロパティ名[インデックス]」を作成します。
  - リストの要素が JavaBean とネストしているため、「books[インデックス情報].title」とプロパティをさらに記述します。
- 初期表示・再表示の際には、カスタムタグ<form:input>が自動的に Command(JavaBean)のプロパティにアクセスし値を取得します。
  - <c:forEarch>はリストのインデックス情報を取得するためだけに使用し、プロパティの値を取得する必要はありません。

```
<h4>入力値検証：階層を持つデータ</h4>
<form:form modelAttribute="sample5Command" action="${appUrl}/test/sample5.html" method="post">
    <h5>購入した本の情報</h5>
    <table border="1">
        <tr>
            <th>No.</th>
            <th>題名</th>
            <th>価格</th>
            <th>著者</th>
        </tr>
        <c:forEach items="${sample5Command.books}" var="bookItem" varStatus="booksStatus">
        <tr>
            <td>${booksStatus.index + 1}</td>
            <td>
                <form:input path="books[${booksStatus.index}].title" />
                <form:errors path="books[${booksStatus.index}].title" cssClass="errors" />
            </td>
            <td>
                <form:input path="books[${booksStatus.index}].price" />
                <form:errors path="books[${booksStatus.index}].price" cssClass="errors" />
            </td>
            <td>
                <c:forEach items="${bookItem.authors}" var="authorItem" varStatus="authorsStatus">
                    <form:input path="books[${booksStatus.index}].authors[${authorsStatus.index}]" />
                    <form:errors path="books[${booksStatus.index}].authors[${authorsStatus.index}]"
cssClass="errors" />
                </c:forEach>
            </td>
        </tr>
        </c:forEach>
    </table>

    <input type="submit"/>
</form:form>
```

JavaBean の中で、さらにリスト型のプロパティ「authors」を表示します。

【Controller の作成】

- 内容は「4.5.2.1　リストの要素がプリミティブ型の場合」の Controller とほぼ同じです。
- 初期表示用のメソッド内で、Command のインスタンスを作成する際に、リストは 1 以上の値を設定しておきます。0 個だと、ブラウザ上で入力フィールドが表示されません。

```
@Controller
@RequestMapping("/test/sample5")
public class Sample5Constroller {

    @Resource
    private Sample5Validator sample5Validator;

    @InitBinder("sample5Command")
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(sample5Validator);
    }

    @ModelAttribute("sample5Command")
    public Sample5Command createInitCommand() {
        Sample5Command command = new Sample5Command();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        Sample5Command command = createInitCommand();

        for(int i=0; i < 3; i++) {
            // 必ず 2 つの空のリストを作る
            BookBean book = new BookBean();
            book.getAuthors().add("");
            book.getAuthors().add("");
            command.getBooks().add(book);
        }

        model.addAttribute("sample5Command", command);
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample5Command") @Valid Sample5Command command,
            BindingResult bindingResult) {

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());
            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

データバインド時に使用する Command のインスタンスを取得するためのメソッド。

初期表示の際に、1 以上の空の項目を用意しておきます。
データがないと、入力項目が表示されません。

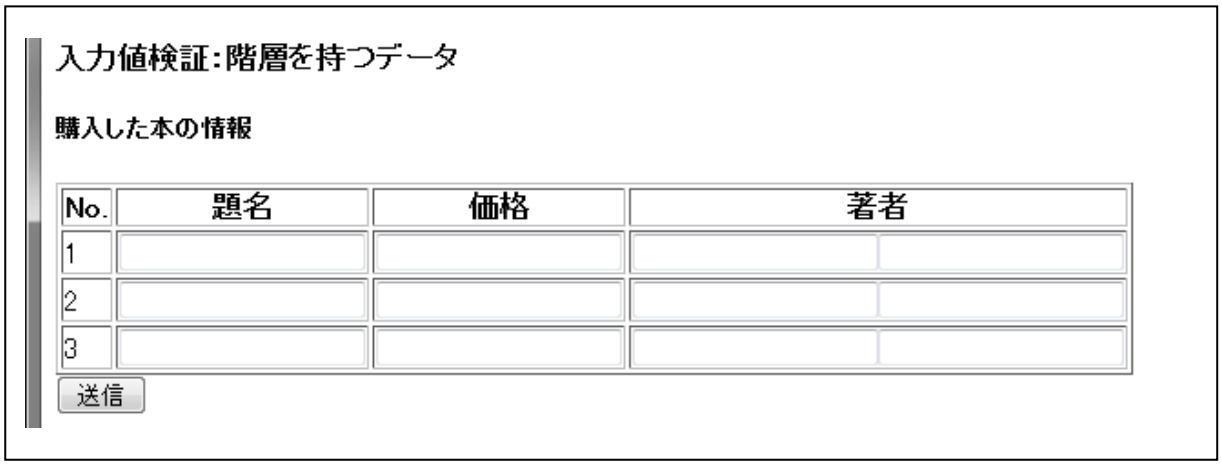
【HTML の表示：初期表示】

```
<h4>入力値検証：階層を持つデータ</h4>
<form id="sample5Command" action="/spring3-mvc/test/sample5.html" method="post">
    <h5>購入した本の情報</h5>
    <table border="1">
        <tr><th>No.</th><th>題名</th><th>価格</th><th>著者</th></tr>
        <tr>
            <td>1</td>
            <td><input id="books0.title" name="books[0].title" type="text" value=""/></td>
            <td><input id="books0.price" name="books[0].price" type="text" value=""/></td>
            <td>
                <input id="books0.authors0" name="books[0].authors[0]" type="text" value=""/>
                <input id="books0.authors1" name="books[0].authors[1]" type="text" value=""/>
            </td>
        </tr>
        <tr>
            <td>2</td>
            <td><input id="books1.title" name="books[1].title" type="text" value=""/></td>
            <td>
                <input id="books1.price" name="books[1].price" type="text" value=""/></td>
            <td>
                <input id="books1.authors0" name="books[1].authors[0]" type="text" value=""/>
                <input id="books1.authors1" name="books[1].authors[1]" type="text" value=""/>
            </td>
        </tr>
        <tr>
            <td>3</td>
            <td><input id="books2.title" name="books[2].title" type="text" value=""/></td>
            <td><input id="books2.price" name="books[2].price" type="text" value=""/></td>
            <td>
                <input id="books2.authors0" name="books[2].authors[0]" type="text" value=""/>
                <input id="books2.authors1" name="books[2].authors[1]" type="text" value=""/>
            </td>
        </tr>
    </table>
    <input type="submit"/>
</form>
```

JavaBean を要素に持つリストのプロパティ「title」。

JavaBean の中で、さらにリスト型のプロパティ「authors」。

【ブラウザの表示：初期表示】



4.5.3. マップによるデータの送受信

マップのデータをバインドする際に、ライブラリ「commons-collections」のクラス「org.apache.commons.collections.map.LazyMap」をインスタンスとして利用します。

- リスト型の LazyList の場合と同様、ブラウザの画面上で、JavaScript を使用し動的に項目を増やした場合に、サーバ側で処理するときにデータ領域を確保していないマップのキーがある場合、「NullPointerException」が発生します。
- LazyMap は、存在しないマップのキーを指定した場合、動的にキーに対する値を作成することで例外が発生しなくなります。

4.5.3.1. マップの値がプリミティブ型の場合

【Command の作成】

- LazyMap のインスタンスは、MapUtils.lazyMap(第 1 引数, 第 2 引数)で取得します。
  - ➢ メソッドの第 1 引数には、マップのインスタンス「LinkedHashMap」を指定します。
    - ✧ データを格納した順を保持する LinkedHashMap を使用する理由として、画面のフィールドの表示順とエラーメッセージの表示順を合わせるためです。
      Validator にて値をチェックする際に、Map#entrySet()にてマップデータを順に取り出し、エラーメッセージオブジェクトを画面に表示されている順に作成し、リストに格納します。
  - ➢ メソッドの第 2 引数には、マップの値となるオブジェクトの Factory クラスを指定します。
    - ✧ 引数なしのデフォルトコンストラクタがある場合は、FactoryUtils.instantiateFactory()を使用します。
- Getter、Setter は通常のものを用意します。

```
import java.io.Serializable;
import java.util.LinkedHashMap;
import java.util.Map;

import org.apache.commons.collections.FactoryUtils;
import org.apache.commons.collections.MapUtils;
import org.apache.commons.lang.builder.ToStringBuilder;

public class Sample4Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private Map<String, String> family;

    @SuppressWarnings("unchecked")
    public Sample4Command() {
        family = MapUtils.lazyMap(
                new LinkedHashMap<String, String>(),
                FactoryUtils.instantiateFactory(String.class));
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
```

```
    }

    public Map<String, String> getFamily() {
        return family;
    }

    public void setFamily(Map<String, String> family) {
        this.family = family;
    }
}
```

setter、getter は通常のものを用意します。

【JSP の作成】

- マップのキーとなるマスターデータ「familyType」をセッションから取得し、それをもとに位置表現の path 属性を組み立てます。
  - キーのマスターデータは、Controller で予め、Model やセッションスコープに登録しておきます。
  - マップの位置表現の書式は、「プロパティ名[キー]」であるため、「family[キー]」とします。

```
<h4>入力値検証：マップ形式のデータ</h4>
<form:form modelAttribute="sample4Command" action="${appUrl}/test/sample4.html" method="post">

    <ul>
        <c:forEach items="${familyType}" var="type" varStatus="familyStatus">
        <li>
            <form:label path="family[${type}]">${type.localeName}</form:label>
            <form:input path="family[${type.name}]" />
            <form:errors path="family[${type}]" cssClass="errors" />
        </li>
        </c:forEach>
    </ul>

    <input type="submit"/>
</form:form>
```

マップのキーとなるデータをセッションから取得します。

【Controller の作成】

- マップのキーとなるリスト(配列)の情報を Model に登録します。
  - 今回は、列挙型を使用し、一覧を values()メソッドで配列として取得します。
  - マップのキーのマスターデータは、プロジェクトによっては DB から取得することもあると思います。

```
@Controller
@RequestMapping("/test/sample4")
public class Sample4Constroller {

    @Resource
    private Sample4Validator sample4Validator;

    @InitBinder("sample4Command")
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(sample4Validator);
    }

    @ModelAttribute("sample4Command")
    public Sample4Command createInitCommand() {
        Sample4Command command = new Sample4Command();
        return command;
```

```
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        Sample4Command command = createInitCommand();
        model.addAttribute("sample4Command", command);

        // マップから値を取得するためキーのリスト
        model.addAttribute("familyType", Family.values());

    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample4Command") @Valid Sample4Command command,
            BindingResult bindingResult) {

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            // マップから値を取得するためキーのリスト
            mav.addObject("familyType", Family.values());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

【マップのキーとなる列挙型の作成】

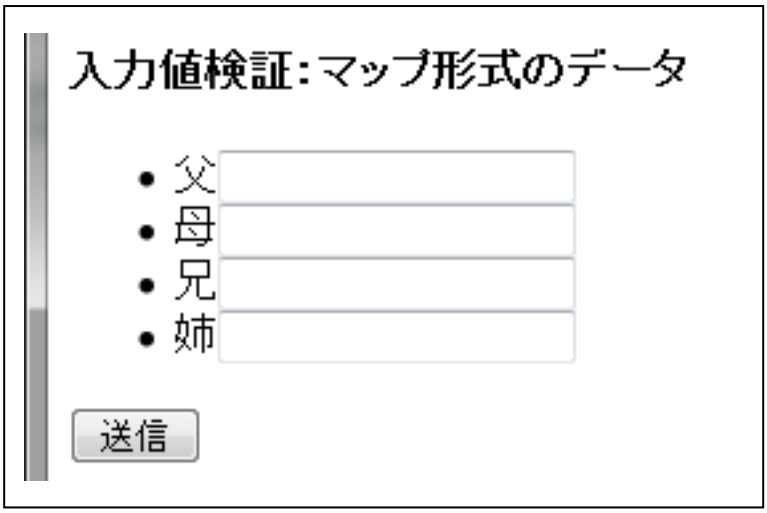
- 列挙型名前をマップのキーとします。
  - プロパティ name に対する Getter が存在しないため、メソッド「getName()」を作成します。

```
public enum Family {
    FATHER("父"), MOTHER("母"), BROTHER("兄"), SISTER("姉");

    private String localeName;

    private Family(String localeName) {
        this.localeName = localeName;
    }

    public String getLocaleName() {
        return localeName;
    }
    //マップのキーの getter
    public String getName() {
        return name();
    }
}
```

【HTML の表示：初期表示】

```
<h4>入力値検証：マップ形式のデータ</h4>
<form id="sample4Command" action="/spring3-mvc/test/sample4.html" method="post"><p>
    <ul>
        <li>
            <label for="familyFATHER">父</label>
            <input id="familyFATHER" name="family[FATHER]" type="text" value=""/>
        </li>
        <li>
            <label for="familyMOTHER">母</label>
            <input id="familyMOTHER" name="family[MOTHER]" type="text" value=""/>
        </li>
        <li>
            <label for="familyBROTHER">兄</label>
            <input id="familyBROTHER" name="family[BROTHER]" type="text" value=""/>
        </li>
        <li>
            <label for="familySISTER">姉</label>
            <input id="familySISTER" name="family[SISTER]" type="text" value=""/>
        </li>
    </ul>
    <input type="submit"/>
</form>
```

【ブラウザでの表示：初期表示】



4.5.3.2.　マップの値が JavaBean（ネストしている）の場合

【Command の作成】

- マップの値がプリミティブ型の時と同様、LazyMap のインスタンスを、MapUtils.lazyMap()で取得します。
- デフォルトコンストラクタでマップの値を作成する場合、FactoryUtils.instantiateFactory()を使用します。
  - ➢　マップのインスタンスには、データを格納した順を保持する LinkedHashMap を指定します。
- デフォルトコンストラクタがない場合や、マップの値の初期値を独自に設定したい場合は、「Factory」インタフェースのメソッド「create()」を実装します。
  - ➢　初期値を設定する場合は、Controller の初期表示のメソッドで設定することをお勧めします。
- Getter、Setter は通常のものを用意します。

```
import java.io.Serializable;
import java.util.LinkedHashMap;
import java.util.Map;

import org.apache.commons.collections.Factory;
import org.apache.commons.collections.FactoryUtils;
import org.apache.commons.collections.MapUtils;
import org.apache.commons.lang.builder.ToStringBuilder;


public class Sample5Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private Map<String, PersonBean> family;

    @SuppressWarnings("unchecked")
    public Sample5Command() {

        family = MapUtils.lazyMap(
                new LinkedHashMap<String, PersonBean>(),
                FactoryUtils.instantiateFactory(PersonBean.class));

        // 独自の Factory を指定する場合
        family = MapUtils.lazyMap(
                new LinkedHashMap<String, PersonBean>(),
                new Factory() {

                    @Override
                    public Object create() {
                        PersonBean person = new PersonBean();
                        // Bean の初期値の設定
                        person.setAge(0);
                        return person;
                    }
                });
    }
```

デフォルトコンストラクタを使用する場合。

独自の Factory を指定する場合。

```
    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }

    public Map<String, PersonBean> getFamily() {
        return family;
    }

    public void setFamily(Map<String, PersonBean> family) {
        this.family = family;
    }
}
```

【JSP の作成】

- マップのキーとなるマスターデータ「familyType」をセッションから取得し、それをもとに位置表現の path 属性を組み立てます。
  - ➢ キーのマスターデータは、Controller で予め Model やセッションスコープに登録しておきます。

```
<h4>入力値検証：階層を持つデータ</h4>
<form:form modelAttribute="sample5Command" action="${appUrl}/test/sample5.html" method="post">

<h5>家族構成</h5>
    <table border="1">
        <tr>
            <th>続柄</th>
            <th>名前</th>
            <th>年齢</th>
        </tr>
        <c:forEach items="${familyType}" var="type" varStatus="familyStatus">
        <tr>
            <td><c:out value="${type.localeName}"/></td>
            <td>
                <form:input path="family[${type.name}].name" />
                <form:errors path="family[${type.name}].name" cssClass="errors" />
            </td>
            <td>
                <form:input path="family[${type.name}].age" />
                <form:errors path="family[${type.name}].age" cssClass="errors" />
            </td>
        </tr>
        </c:forEach>
    </table>

    <input type="submit"/>
</form:form>
```

マップのキーとなるデータをセッションから取得します。

【Controller の作成】

- マップのキーとなるリスト（配列）の情報を Model に登録します。
  - 今回は、列挙型を使用し、一覧を values()メソッドで配列として取得します。

```
@Controller
@RequestMapping("/test/sample5")
public class Sample5Constroller {

    @Resource
    private Sample5Validator sample5Validator;

    @InitBinder("sample5Command")
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(sample5Validator);
    }

    @ModelAttribute("sample5Command")
    public Sample5Command createInitCommand() {
        Sample5Command command = new Sample5Command();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        Sample5Command command = createInitCommand();
        model.addAttribute("sample5Command", command);

        // マップから値を取得するためキーのリスト
        model.addAttribute("familyType", Family.values());
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample5Command") @Valid Sample5Command command,
            BindingResult bindingResult) {

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            // マップから値を取得するためキーのリスト
            mav.addObject("familyType", Family.values());
            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

【マップのキーとなる列挙型の作成】

- 列挙型名前をマップのキーとします。
  - プロパティ name に対する Getter が存在しないため、メソッド「getName()」を作成します。

```
public enum Family {
	FATHER("父"), MOTHER("母"), BROTHER("兄"), SISTER("姉");

	private String localeName;

	private Family(String localeName) {
		this.localeName = localeName;
	}

	public String getLocaleName() {
		return localeName;
	}
	//マップのキーの getter
	public String getName() {
		return name();
	}
}
```

【HTML の表示：初期表示】

```
<h4>入力値検証：階層を持つデータ</h4>
<form id="sample5Command" action="/spring3-mvc/test/sample5.html" method="post">
	<h5>家族構成</h5>
	<table border="1">
		<tr>
			<th>続柄</th>
			<th>名前</th>
			<th>年齢</th>
		</tr>
		<tr>
			<td>父</td>
			<td><input id="familyFATHER.name" name="family[FATHER].name" type="text" value=""/></td>
			<td><input id="familyFATHER.age" name="family[FATHER].age" type="text" value=""/></td>
		</tr>
		<tr>
			<td>母</td>
			<td><input id="familyMOTHER.name" name="family[MOTHER].name" type="text" value=""/></td>
			<td><input id="familyMOTHER.age" name="family[MOTHER].age" type="text" value=""/></td>
		</tr>
		<tr>
			<td>兄</td>
			<td><input id="familyBROTHER.name" name="family[BROTHER].name" type="text" value=""/></td>
			<td><input id="familyBROTHER.age" name="family[BROTHER].age" type="text" value=""/></td>
		</tr>
		<tr>
			<td>姉</td>
			<td><input id="familySISTER.name" name="family[SISTER].name" type="text" value=""/></td>
			<td><input id="familySISTER.age" name="family[SISTER].age" type="text" value=""/></td>
		</tr>
		</table>
	<input type="submit"/>
</form>
```

【ブラウザでの表示：初期表示】

4.5.4. マップとリスト組み合わせたデータの送受信

- マップとリストを組み合わせることで、複雑なデータ構造を表現することができます。
  - インスタンスには、それぞれ、LazyMap、LazyList を使用します。
- 複雑なデータ構造を表現することは、送受信するデータ量も非常に多くなり、以下の点に注意する必要があります。その場合は、画面を分割しデータ量を減らすことをお勧めします。
  - ネットワークの環境、クライアントのスペックが低いことにより、単にデータを送受信するだけでも負荷がかかります。
  - JavaScript でフォーマットやチェックなどの処理を追加している場合、DOM の走査だけで時間がかかります。特に、Internet Explorer 6,7 などの古いブラウザを使用していると動作が非常に遅くなります。

【Command の作成】

- マップの値がリストのデータ構造を定義します。
- リストのインスタンスは、Factory インタフェースの実装メソッド「create()」の中で作成します。

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import org.apache.commons.collections.Factory;
import org.apache.commons.collections.FactoryUtils;
import org.apache.commons.collections.ListUtils;
import org.apache.commons.collections.MapUtils;
import org.apache.commons.lang.builder.ToStringBuilder;

public class Sample8Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private Map<String, List<TeamBean>> categoryEntries;

    @SuppressWarnings("unchecked")
    public Sample8Command() {

        categoryEntries = MapUtils.lazyMap(new LinkedHashMap<String, List<TeamBean>>(),
                new Factory() {

                    @Override
                    public Object create() {

                        // 初期サイズ 1 で作成する
                        List<TeamBean> list = new ArrayList<TeamBean>();
                        list.add(new TeamBean());

                        return ListUtils.lazyList(list,
                                FactoryUtils.instantiateFactory(TeamBean.class));
                    }
                });
```

マップの値のリストのインスタンスの作成。

```
    }

    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }

    public Map<String, List<TeamBean>> getCategoryEntries() {
        return categoryEntries;
    }

    public void setCategoryEntries(Map<String, List<TeamBean>> categoryEntries) {
        this.categoryEntries = categoryEntries;
    }

}
```

【リストの要素の JavaBean「TeamBean」の定義】

- プロパティ「members」は、リスト型で、さらにネストしています。

```
public class TeamBean implements Serializable {
    private String name;

    private List<PersonBean> members;

    @SuppressWarnings("unchecked")
    public TeamBean() {

        members = ListUtils.lazyList(new ArrayList<PersonBean>(),
                FactoryUtils.instantiateFactory(PersonBean.class));
    }

    // getter、setter は省略
}
```

【TeanBean のプロパティ「members」のリストの要素「PersonBean」の定義】

```
public class PersonBean implements Serializable {

    private String name;

    private Integer age;

    public PersonBean() {

    }

    // getter、setter は省略
}
```

【JSP の作成】

- マップのキーとなるマスターデータ「categoryType」をセッションから取得します。
  - Controller などで、事前に Model やセッションスコープに登録しておきます。

```
<form:form modelAttribute="sample8Command" action="${appUrl}/test/sample8.html" method="post">

<c:forEach items="${categoryType}" var="category" varStatus="categoryStatus">
    <h4>カテゴリ：<c:out value="${category.localeName}"/></h4>

    <table border="1">
        <tr>
            <th>No.</th>
            <th>チーム名</th>
            <th>チーム情報</th>
        </tr>

        <%-- プロパティ「categoryEntries」のマップの値「List<TeamBean>」の取得 --%>
        <c:forEach items="${sample8Command.categoryEntries[category.name]}" var="team"
varStatus="teamStatus">
        <tr>
            <td>${teamStatus.index + 1}</td>
            <td>
                <form:input path="categoryEntries[${category.name}][${teamStatus.index}].name" />
                <form:errors path="categoryEntries[${category.name}][${teamStatus.index}].name"
cssClass="errors" />
            </td>
            <td>
                <ol>
                    <%-- TeamBean のプロパティ「members」の値「List<PersonBean>」の取得 --%>
                    <c:forEach items="${team.members}" var="person" varStatus="personStatus">
                    <li>
                        名前：
                        <form:input
path="categoryEntries[${category.name}][${teamStatus.index}].members[${personStatus.index}].name" />
                        <form:errors
path="categoryEntries[${category.name}][${teamStatus.index}].members[${personStatus.index}].name"
cssClass="errors" />
                        <br/>
                        年齢：
                        <form:input
path="categoryEntries[${category.name}][${teamStatus.index}].members[${personStatus.index}].age" />
                        <form:errors
path="categoryEntries[${category.name}][${teamStatus.index}].members[${personStatus.index}].age"
cssClass="errors" />
                    </li>
                    </c:forEach>
                </ol>
            </td>
        </tr>
        </c:forEach>
    </table>

</c:forEach>

    <input type="submit" name="check1"/>
</form:form>
```

TeamBean のプロパティ「name」の入力フィールド。

PersonBean のプロパティ「name」の入力フィールド。

【マップのキーとなる列挙型の作成】

```
public enum RaceCategory {

    PROFESSIONAL("プロフェッショナル"), EXPERT("エキスパート"), BEGINNER("初心者");

    private String localeName;

    private RaceCategory(String localeName) {
        this.localeName = localeName;
    }

    public String getLocaleName() {
        return localeName;
    }

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

}
```

【Controller の作成】

```
@Controller
@RequestMapping("/test/sample8")
public class Sample8Constroller {

    @ModelAttribute("sample8Command")
    public Sample8Command createInitCommand() {
        Sample8Command command = new Sample8Command();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        Sample8Command command = createInitCommand();

        // 初期データの作成
        for(RaceCategory category :  RaceCategory.values()) {
            List<TeamBean> entries = command.getCategoryEntries().get(category.name());
            entries.add(new TeamBean());
            for(TeamBean team : entries) {
                for(int i=0; i < 3; i++) {
                    team.getMembers().add(new PersonBean());
                }

            }
        }

        model.addAttribute("sample8Command", command);

        // マップから値を取得するためのキーのリスト
        model.addAttribute("categoryType", RaceCategory.values());
```

初期表示用の空のデータの作成。

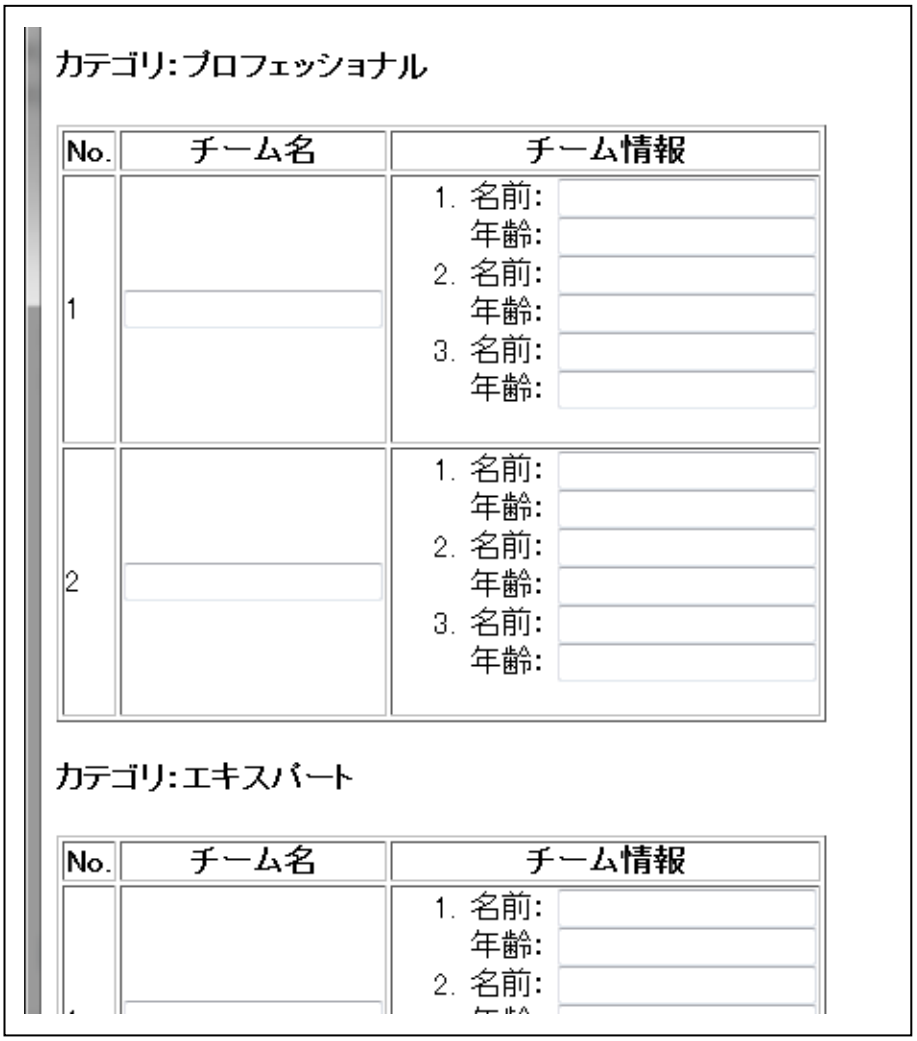
```
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample8Command") Sample8Command command,
            BindingResult bindingResult) {

        System.out.println(command.toString());

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            // マップから値を取得するためのキーのリスト
            mav.addObject("categoryType", RaceCategory.values());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;

    }

}
```

【HTML の表示：初期表示】

```
<form id="sample8Command" action="/spring3-mvc/test/sample8.html" method="post">
    <h4>カテゴリ：プロフェッショナル</h4>
    <table border="1">
        <tr>
            <th>No.</th>
            <th>チーム名</th>
            <th>チーム情報</th>
        </tr>
        <tr>
            <td>1</td>
            <td>
                <input                                          id="categoryEntriesPROFESSIONAL0.name"
name="categoryEntries[PROFESSIONAL][0].name" type="text" value=""/></td>

            <td>
                <ol>
                    <li>
                        名前：
                        <input                              id="categoryEntriesPROFESSIONAL0.members0.name"
name="categoryEntries[PROFESSIONAL][0].members[0].name" type="text" value=""/><br/>
                        年齢：
                        <input                                id="categoryEntriesPROFESSIONAL0.members0.age"
name="categoryEntries[PROFESSIONAL][0].members[0].age" type="text" value=""/>
                    </li>
                    <li>
                        名前：
                        <input                              id="categoryEntriesPROFESSIONAL0.members1.name"
name="categoryEntries[PROFESSIONAL][0].members[1].name" type="text" value=""/><br/>

                        年齢：
                        <input                                id="categoryEntriesPROFESSIONAL0.members1.age"
```

```
name="categoryEntries[PROFESSIONAL][0].members[1].age" type="text" value=""/>
                        </li>
                        <li>
                            名前：
                            <input                          id="categoryEntriesPROFESSIONAL0.members2.name"
name="categoryEntries[PROFESSIONAL][0].members[2].name" type="text" value=""/><br/>
                            年齢：
                            <input                            id="categoryEntriesPROFESSIONAL0.members2.age"
name="categoryEntries[PROFESSIONAL][0].members[2].age" type="text" value=""/>
                        </li>
                    </ol>
                </td>
            </tr>

        </table>
        <!-- 省略 -->

<input type="submit" name="check1"/>
</form>
```

【ブラウザでの表示：初期表示】



5. REST サービスの作成

Spring MVC は REST サービスを非常に簡単に作成できます。

- サーバ側は、通常の Controller と同様に@RequestMapping をメソッドに付与して作成します。
  - データ形式として、XML と JSON に標準で対応しています。
- クライント側は、ブラウザ上では通常と同様に JavaScript のライブラリ「jQuery」で実装します。また、Java で直接 URL に対してデータをやり取りしたい場合、Spring のクラス「RestTemplate」を使用します。

5.1. JSON／XML データの送受信

REST サービスを実装に当たって、JSON 形式や XML データをサーバ側／クライアント側で送受信する方法を説明します。

5.1.1. 準備

5.1.1.1. JSON を利用するための準備

pom.xml に、ライブラリ「Jackson(http://jackson.codehaus.org/)」を追加します。

```
<dependencies>
    <!-- JSON Library -->
    <dependency>
        <groupId>org.codehaus.jackson</groupId>
        <artifactId>jackson-core-asl</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>org.codehaus.jackson</groupId>
        <artifactId>jackson-mapper-asl</artifactId>
        <version>${jackson.version}</version>
    </dependency>
</dependencies>

<properties>
    <jackson.version>1.8.2</jackson.version>
</properties>
```

5.1.1.2. RestTemplate を利用するための準備

- pom.xml にライブラリ、「HttpClient」を追加します。
- Spring 3.2 から利用可能になった、HTTP の PATCH メソッドを利用する場合は、version4.2 移行を利用します。

```
<dependency>
    <groupId>commons-httpclient</groupId>
    <artifactId>commons-httpclient</artifactId>
    <version>3.1</version>
</dependency>
```

5.1.2. サーバ側で JSON データを出力／クライアント側で取得する場合

REST を利用した Web サービスを作成する場合、多くがクライアント側において URL で条件を指定し、JSON や XML を取得するこのケースになると思います。

5.1.2.1. サーバ側（Controller）

JSON 形式のデータをサーバで出力する方法を説明します。

- メソッドに、アノテーション「@ResponseBody」を付加します。戻り値により自動的に判断し、HTTP ヘッダーのメディアタイプの Accept が「application/json」になります。
- レスポンス（戻り値）が XML、JSON かどうか、パッと見ではソースコード上は区別がつかないため、次の方法でメディアタイプを明示した方がよいです。
  - レスポンス時のメディアタイプを指定したい場合は、
    @RequestMapping(…,headers="Accept=application/json"))で指定します。
  - または、Spring3.1 で追加になった「@RequestMapping(…,produces="application/json"))」で指定します。
- メソッドの戻り値は、String の他に、JavaBean でも構いません。Spring の「HttpMessageConverter」により自動的に変換されます（「5.3.1　データの変換「HttpMessageConverter」）。
  - シリアライズ可能である必要がありませんが、後々使いまわしたいときに便利なので「java.io.Serializable」を実装します。

【Contoller】

```
@Controller
public class JsonController {

    @RequestMapping(value="/ajax/jsonOut1", method={RequestMethod.GET, RequestMethod.POST})
    //@RequestMapping(value="/ajax/jsonOut1", headers="Accept=application/json")
    //@RequestMapping(value="/ajax/jsonOut1", produces="application/json")
    @ResponseBody
    public List<UserInfoViewDto> jsonOut (@RequestParam String cd) {

        if(StringUtils.isEmpty(departmentCd)) {
            return new ArrayList<UserInfoViewDto>();
        }

        List<UserInfoDto> list = /* Servlet/DAO から取得する */;
        return list;

    }
}
```

・メディアタイプを明示的に指定できます。
・“produces”は、Spring3.1 から利用可能です。

戻り値の Java オブジェクトは、自動的に JSON に変換されます。

5.1.2.2. クライアント側（jQuery）

JavaScript のライブラリ「jQuery」を使用して、クライアント側で JSON 形式のデータを取得する方法を説明します。

- jQuery を利用する場合、「ajax(…)」メソッドでデータを取得します。
  - 戻り値が JSON と固定であるならば、「getJSON(…)」でも構いません。
- レスポンス時のメディアタイプを“dataType”で指定します。
  - "xml"は XML ドキュメント。
  - "html"は HTML をテキストデータとして（ただし script タグの中身は実行されます）。
  - "json"は JSON 形式のデータ。
  - "text"は通常の文字列。

```
<script type="text/javascript">
$(document).ready(function(){
	$('#jsonOut1').click(function(){
		$.ajax({
			type: "POST",
			url : "${appUrl}/ajax/jsonOut1.html",
			data : {"cd": "syasin"},
			// 受信時（レスポンス時）のメディアタイプ
			dataType: "json",
			success:function(data){
				//TODO:
				alert(data);
			}
		});
	});
});
</script>
<ol>
	<li> <a href="${appUrl}/ajax/jsonOut1.html?cd=aaa">JSON 形式を取得(GET で取得)</a></li>
	<li> <span id="jsonOut1">JSON 形式を取得(jQuery 経由で取得)</span></li>
</ol>
```

【取得できる JSON データの例】

```
[{"name":"管理者","id":"1"},{"name":"一般ユーザ","id":"2"}]
```

5.1.2.3.　クライアント側（RestTemplate）

RestTemplate を使用して、クライアント側で JSON 形式のデータを取得する方法を説明します。

- HTTP の GET メソッドでアクセスする場合、「RestTemplate#getForObject(…)」を利用します。
  - HTTP のメソッドごとに、RestTemplate のメソッドも用意されています。
- クエリストリングのように URL パラメータを指定したい場合は、変数も利用できます。
  - Controller 側の@PathVariable のように指定します。また、Map 形式でも指定できます。
  - URI を組み立てるためのユーティリティクラス「UriComponents／UriComponetsBuilder」も用意されています。使い方は、「5.6.3　URI の組み立て（UriTemplate、UriComponents／UriComponentsBuilder)」を参照してください。
- 戻り値がリスト型でその要素が JavaBean の場合（List<JavaBean>)、JavaBean のプロパティ名と値が対になり Map として返されます。
  - 今回は、List<Map<stromg, Stromg>>が戻り値となります。
  - JavaBean に戻したい場合は、別ライブラリ「Common BeanUtils」などを利用してマッピングします。
- プロキシ経由でアクセスする環境や認証が必要な URL にアクセスする場合は、HttpClient のインスタンスを「RestTemplate」に設定します。詳細は、「5.6　クライアント側「RestTemplate」を参照してください。
  - 通常は、RestTemplate のインスタンスは Spring Bean として登録して利用します。

```
public class RestTemplateClient {
    public void loadJson1() {

        try {
            RestTemplate restTemplate = new RestTemplate();

            List<Map<String, String>> responseData = restTemplate.getForObject(
                    "http://localhost:8080/spring-mvc-3.1/ajax/jsonOut1.html?cd={cd}",
                    List.class,
                    "asyin");

            // Map 形式の項目を DTO に戻す
            for(Map<String, String> item : responseData) {
                UserInfoViewDto dto = new UserInfoViewDto();
                BeanUtils.populate(dto, item);
                System.out.println(dto);
            }

        } catch(RestClientException e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

戻り値がリストの JavaBean 場合は、JavaBean は Map にプロパティ名と値が対になり格納されています。

戻り値が Bean 場合は、Map にプロパティ名と値が遂になり格納されています。

5.1.3. クライアント側で JSON データを送信／サーバ側で受信する場合

クライアントから JSON データを送信するようなケースはあまりありませんが、階層化されたデータを送信する場合に利用します。クライアントの処理が複雑になるため、できるだけこの方式はとらず、URL にデータを埋め込む方式を採用／設計することをお勧めします。

5.1.3.1. サーバ側（Controller）

JSON 形式のデータをサーバで受信する方法を説明します。

- 引数は、「@RequestBody」を付与します。
  - メソッドの変数に@RequestBody を付与した場合、自動的に、HTTP ヘッダーの Content-Type が「application/json」となります。
  - @RequestBody を付与した引数が Java Bean 形式の場合、自動的にマッピングされます。また、String 型、Integer 型など各種の形式でも問題ありません。
  - レスポンスで使用しメソッドに付与する@ResponseBody とは異なるので注意。
  - FORM のデータを受信する「@ModelAttribute」に対応します。
- リクエスト（入力値）が XML、JSON かどうか、パッと見ではソースコード上は区別がつかないため、次の方法でメディアタイプを明示した方がよいです。
  - リクエスト時のメディアタイプを指定したい場合は、@RequestMapping(…,headers="Content-Type=application/json"))で指定します。
  - または、Spring3.1 で追加になった「@RequestMapping(…,consumes ="application/json"))」で指定します。
- @RequstBody を付与した変数に、入力値チェック用のアノテーション「@Valid」を付与した場合、Spring3.0 では動作しないため手動でチェックを実行する必要があります。Spring3.1 以降は正常に動作します。
- メソッドの引数は、String、Java のオブジェクトでも構いません。Spring の「HttpMessageConverter」により自動的に変換されます（「5.3.1　データの変換「HttpMessageConverter」）。
- データバインドエラーが発生した場合は、HTTP ステータスコード「400（Bad Request）」が発生します。

【Conroller】

```
@Controller
public class JsonController {

    @RequestMapping(value="/ajax/jsonIn1")
    //@RequestMapping(value="/ajax/jsonIn1", headers="Content-Type=application/json")
    //@RequestMapping(value="/ajax/jsonIn1", consumes="application/json")
    public ModelAndView jsonIn1(@RequestBody JsonSampleCommand1 command) {

        ModelAndView mav = new ModelAndView("/ajax/sample1");
        mav.addObject("data", command);
        return mav;
```

・メディアタイプを明示的に指定できます。
・“consumes”は、Spring3.1 から利用可能です。

```
    }
}
```

5.1.3.2. クライアント側（jQuery）

JavaScript のライブラリ「jQuery」を使用して、クライアント側で JSON 形式のデータを送信する方法を説明します。

- jQuery を利用する場合、「ajax(...)」メソッドでデータを送信します。
- リクエストする送信時のメディアタイプを、「contentType」で“application/json”と指定します。
  - 指定すると、「data」で指定したデータが JSON になります。
  - contentType の値として、文字エンコーディングも明示的に指定した方が良いです。
- レスポンス時のデータが JSON や XML の場合、通常は jQuery などで取得します。その場合、リクエスト時のメディアタイプを指定します。
- データバインドエラーの HTTP ステータスコード「400（Bad Request）」が発生した場合、error メソッドでハンドリングします。その場合、status は“error”、errorThrown は“Bad Request”となります。

```
<script type="text/javascript">
$(document).ready(function(){
    $('#jsonIn1').click(function(){
        $.ajax({
            type: "POST",
            url : "${appUrl}/ajax/jsonIn1.html",
            // 送信時のメディアタイプ
            contentType: "application/json;charset=UTF-8",
            data : '{"name":"猫", "age":"2"}',
            // 受信時のメディアタイプ
            dataType: "html",
            success:function(data){
                alert(data);
            },
            error:function(XMLHttpRequest, textStatus, errorThrown){
                // エラーのタイプ（timeout, error, notmodified, parsererror）
                alert("textStatus =" + textStatus);

                // エラーメッセージ
                alert("errorThrown=" + errorThrown);
            }
        });
    });
});
</script>
<ol>
    <li> <span id="jsonIn1">JSON 形式を送信、HTML を受信</span></li>
</ol>
```

・送信時のメディアタイプを“contentType”で明示的に指定できます。
・指定すると、“data”でした送信データがJSON 形式になります。

・入力値エラーや、そのほかの通信エラーが発生した場合のエラー処理です。

5.1.3.3.　クライアント側（RestTemplate）

RestTemplate を使用して、クライアント側で JSON 形式のデータを送信する方法を説明します。

- RestTemplate は、URL ベースでデータを送信することを前提としているため、今回の場合は汎用的にアクセスできるメソッドとして、「**RestTemplate#exchange(...)**」を利用します。
- Java オブジェクトを JSON 形式へ自動的に変換して送信するために、「**HttpEntity**」で送信データである Command を設定します。
  - 送信データを指定するために「**HttpHeaders#setContentType(...)**」でメディアタイプとして「application/json」を指定します。
  - HttpHeaders でメディアタイプを指定すると、Controller 側と同様に HttpMessageConverter で JSON 形式に自動的に変換されます。
- 戻り値は、「ResponseEntity」で取得します。戻り値は、HTML(text/html)で文字列であるため、Generics のクラスタイプは String で指定します。
  - 受信したデータは、「**ResponseEntity#getBody()**」で取得できます。
  - ただし、入力値エラーなど発生した場合と区別するために、HTTP ステータスコード判定します。
- プロキシ経由でアクセスする環境や認証が必要な URL にアクセスする場合は、HttpClient のインスタンスを「RestTemplate」に設定します。詳細は、「5.6　クライアント側「RestTemplate」を参照してください。
  - 通常は、RestTemplate のインスタンスは Spring Bean として登録して利用します。

```
public class RestTemplateClient {

    public void postJson1() {

        try {
            RestTemplate restTemplate = new RestTemplate();

            // 入力データの作成
            JsonSampleCommand1 inputData = new JsonSampleCommand1();
            inputData.setName("猫");
            inputData.setAge(2);

            // リクエスト時の HTTP ヘッダーを設定（ContentType を指定）
            HttpHeaders rqeustHeaders = new HttpHeaders();
            rqeustHeaders.setContentType(MediaType.APPLICATION_JSON);

            // リクエスト時の HTTP データの作成（HTTP ヘッダーと Body(入力データ)）を指定。
            HttpEntity<JsonSampleCommand1> requestEntity =
                    new HttpEntity<JsonSampleCommand1>(inputData, rqeustHeaders);

            ResponseEntity<String> responseData = restTemplate.exchange(
                    "http://localhost:8080/spring-mvc-3.1/ajax/jsonIn1.html",
                    HttpMethod.POST,
                    requestEntity,
                    String.class);

            if(responseData.getStatusCode() == HttpStatus.OK) {
                System.out.println(responseData.getBody());
```

・送信データを Java オブジェクトで作成します。
・Controller 側の@RequestBody で指定したデータからと同じものを使用します。

・HTTP ヘッダーの ContentType を指定します。
・リクエストデータを作成します。

・サーバからのデータを取得します。

```
            } else if(responseData.getStatusCode() == HttpStatus.BAD_REQUEST) {
                // 入力データが不正な場合
                System.out.println(responseData.getBody());
            }

        } catch(RestClientException e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.1.4. サーバ／クライアント側の両方で JSON データを送受信する

「5.1.2　サーバ側で JSON データを出力／クライアント側で取得する場合」「5.1.3　クライアント側で JSON データを送信／サーバ側で受信する場合」で説明した JSON データの送受信を組み合わせた方法を説明します。

5.1.4.1. サーバ側（Controller）

JSON 形式データをサーバ側で送受信する方法を説明します。

- メソッドと引数に対してアノテーション「@ResponseBody」と「@RquestBody」を設定します。
- メディアタイプを指定したい場合は、「headers」を使用します。
  - Spring3.1 からは、「sonsumes」「produces」でも指定可能です。
- リクエストやレスポンス時のデータの送受信の動作を細かく指定したい場合は、引数として「HttpEntity」、戻り値として「ResponseEntity」を使用します。

【Controller】

・メディアタイプを明示的に指定できます。
・“consumes”、“produces”は、Spring3.1 から利用可能です。

```
@Controller
public class JsonController {

    @RequestMapping(value="/ajax/jsonIn2")
    //@RequestMapping(value="/ajax/jsonIn2",
                headers={"Content-Type=application/json", "Accept=application/json"})
    //@RequestMapping(value="/ajax/jsonIn2", consumes="application/json", produces="application/json")
    @ResponseBody
    public JsonSampleCommand1 jsonIn2(@RequestBody JsonSampleCommand1 command) {
        //TODO:
        command.setAge(command.getAge()+10);
        return command;
    }
}
```

5.1.4.2. クライアント側（jQuery）

JavaScript のライブラリ「jQuery」を使用して、クライアント側で JSON 形式のデータを送受信する方法を説明します。

- jQuery を利用する場合、「ajax(...)」メソッドでデータを取得します。
- リクエストする送信時のメディアタイプを「contentType」で“application/json”と指定します。
  - 指定すると、「data」で指定したデータが JSON になります。
- レスポンスの受信時のメディアタイプを「dataType」で“json”と指定します。

```
<script type="text/javascript">
$(document).ready(function(){
	$('#jsonIn2').click(function(){
		$.ajax({
			type: "POST",
			url : "${appUrl}/ajax/jsonIn2.html",
			// 送信時のメディアタイプ
			contentType: "application/json;charset=UTF-8",
			data : '{"name":"猫", "age":"2"}',
			// 受信時のメディアタイプ
			dataType: "json",
			success:function(data){
				alert("name=" + data.name + ", age=" + data.age);
			},
			error:function(XMLHttpRequest, textStatus, errorThrown){
				// エラーのタイプ（timeout, error, notmodified, parsererror）
				alert("textStatus=" + textStatus);

				// エラーメッセージ
				alert("errorThrown=" + errorThrown);
			}
		});
	});
});
</script>
<ol>
	<li> <span id="jsonIn2">JSON 形式を送信、JSON を受信</span></li>
</ol>
```

5.1.4.3.　クライアント側（RestTemplate）

RestTemplate を使用して、クライアント側で JSON 形式のデータを送受信する方法を説明します。

- RestTemplate は、URL ベースでデータを送信することを前提としているため、今回の場合は汎用的にアクセスできるメソッドとして、「RestTemplate#exchange(...)」を利用します。
- 送信データは、「HttpEntity」で組み立てます。
  - 送信時のデータ形式を「HttpHeaders#setContentType(...)」で指定します。
- 受信データは、JavaBean の場合、自動的にマッピングされた形で取得できます。
  - 「5.1.2.3　クライアント側（RestTemplate）」のようにリスト型の場合は、自分で変換が必要になります。

```
public class RestTemplateClient {
    public void postJson2() {
        try {
            RestTemplate restTemplate = new RestTemplate();

            // 入力データの作成
            JsonSampleCommand1 inputData = new JsonSampleCommand1();
            inputData.setName("猫");
            inputData.setAge(2);

            // リクエスト時の HTTP ヘッダーを設定（ContentType を指定）
            HttpHeaders rqeustHeaders = new HttpHeaders();
            rqeustHeaders.setContentType(MediaType.APPLICATION_JSON);

            // リクエスト時の HTTP データの作成（HTTP ヘッダーと Body(入力データ)）を指定。
            HttpEntity<JsonSampleCommand1> requestEntity =
                    new HttpEntity<JsonSampleCommand1>(inputData, rqeustHeaders);

            ResponseEntity<JsonSampleCommand1> responseData = restTemplate.exchange(
                    "http://localhost:8080/spring-mvc-3.1/ajax/jsonIn2.html",
                    HttpMethod.POST,
                    requestEntity,
                    JsonSampleCommand1.class);

            if(responseData.getStatusCode() == HttpStatus.OK) {
                // 取得データの抽出
                JsonSampleCommand1 outputData = responseData.getBody();
                System.out.println(outputData);

            } else if(responseData.getStatusCode() == HttpStatus.BAD_REQUEST) {
                // 入力データが不正な場合
                System.out.println(responseData.getBody());
            }

        } catch(RestClientException e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

・送信データを Java オブジェクトで作成します。
・Controller 側の@RequestBody で指定したデータからと同じものを使用します。

・HTTP ヘッダーの ContentType を指定します。
・リクエストデータを作成します。

・戻り値が JavaBean なので自動的に変換されているため、直接所得可能です。

5.1.5. サーバ側で XML データを出力／クライアント側で取得（JAXB）

5.1.5.1. JAXB の Java オブジェクトの作成

XML の形式が複雑になると、JAXB で Java オブジェクトを定義するのが面倒になります。また、JAXB の知識も必要になるため、JDK1.6 の標準機能である「xjc」コマンドを使用して XML スキーマから Java ソースを自動生成することをお勧めします。詳細は、「5.7　JAXB の Java ソースの自動生成」を参照してください。

- Spring MVC を使用して JAXB で XML をやり取りする場合、アノテーション「@XmlRootElement」が付与されている XML のルート要素のクラスを受け渡します。
  - 子要素のオブジェクトで@XmlRootElement が付与されていないものを直接やり取りすることはできません。

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;


@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "id",
    "value"
})
@XmlRootElement(name = "SampleJaxb1")
public class SampleJaxb1 {

    protected String id;
    protected Integer value;

    public String getId() {
        return id;
    }

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

}
```

・JAXB のプロパティの属性の定義。
・この場合、プロパティは子要素とする。

5.1.5.2.　サーバ側（Controller）

XML 形式のデータをサーバ側で出力する方法を説明します。

- JSON データの場合と同様に、メソッドにアノテーション「@ResponseBody」を付与します。戻り値により自動的に判断し、HTTP ヘッダーのメディアタイプの Accept が「application/xml」になります。
- レスポンス（戻り値）が XML、JSON かどうか、パッと見ではソースコード上は区別がつかないため、次の方法でメディアタイプを明示した方がよいです。
  - レスポンス時のメディアタイプを指定したい場合は、
    @RequestMapping(…,**headers="Accept=application/xml "**))で指定します。
  - または、Spring3.1 で追加になった「@RequestMapping(…,**produces="application/xml"**))」で指定します。
- メソッドの戻りは、必ず JAXB のアノテーション「**@XmlRootElement**」が付与されたオブジェクトを返す必要があります。

```
@Controller
public class XmlJaxbController {

    @RequestMapping(value="/ajax/xmlOut1")
//    @RequestMapping(value="/ajax/xmlOut1", headers="Accept=application/xml")
//    @RequestMapping(value="/ajax/xmlOut1", produces="application/xml")
    @ResponseBody
    public SampleJaxb1 jsonOut1(@RequestParam String cd) {

        System.out.printf("cd=%s¥n", cd);

        SampleJaxb1 outData = new SampleJaxb1();
        outData.setId("猫");
        outData.setValue(2);

        return outData;
    }

}
```

・メディアタイプを明示的に指定できます。
・“produces”は、Spring3.1 から利用可能です。

・JAXB 用のオブジェクトを返す必要があります。

5.1.5.3.　クライアント側（jQuery）

JavaScript のライブラリ「jQuery」を使用して、クライアント側で JSON 形式のデータを取得する方法を説明します。

- jQuery を利用する場合、「ajax(…)」メソッドでデータを取得します。
- 受信するレスポンス時のメディアタイプを「dataType」で指定し、“xml”と設定します。

```
<script type="text/javascript">
$(document).ready(function(){
    $('#xmlOut1').click(function(){
        $.ajax({
            type: "POST",
            url : "${appUrl}/ajax/xmlOut1.html",
            data : {"cd": "syasin"},
            // 受信時のメディアタイプ
            dataType: "xml",
```

```
                success:function(data){
                    alert(data);
                }
            });

    });
});
</script>
<ol>
    <li><a href="${appUrl}/ajax/xmlOut1.html?cd=aaa">XML 形式を取得(GET で取得)</a></li>
    <li><span id="xmlOut1">XML 形式を取得(jQuery 経由で取得)</span></li>
</ol>
```

【取得できる XML データの例】

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SampleJaxb1><id>猫</id><value>2</value></SampleJaxb1>
```

5.1.5.4. クライアント側（RestTemplate）

RestTemplate を使用して、クライアント側で XML 形式のデータを取得する方法を説明します。

- HTTP の GET メソッドでアクセスする場合、JSON の時と同じように「RestTemplate#getForObject(…)」を利用します。
- クエリストリングのように URL パラメータを指定したい場合は、変数も利用できます。
  - Controller 側の@PathVariable のように指定します。また、Map 形式でも指定できます。
- 戻り値が JAXB 形式のオブジェクトの場合、XML⇒Java オブジェクトに自動的にマッピングされます。
  - 実際には、HttpMessageConverter を利用して変換されるため、アノテーション「@XmlRootElement」が付与されている必要があります。
  - 戻り値を String 型にすると、JAXB オブジェクトではないため、テキストベースの JSON 形式のが取得できます。

```
public class RestTemplateClient {

public void loadXml1() {

        try {
            RestTemplate restTemplate = new RestTemplate();

            SampleJaxb1 responseData = restTemplate.getForObject(
                    "http://localhost:8080/spring-mvc-3.1/ajax/xmlOut1.html?cd={cd}",
                    SampleJaxb1.class,
                    "asyin");

            System.out.println(responseData);

        } catch(RestClientException e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

・戻り値が JAXB のオブジェクトの場合、自動的にマッピングされて格納されます。
・Sring 型にすると、テキストベースの JSON になります。

5.1.6. クライアント側で XML データを送信／サーバ側で受信する場合

クライアント側から XML データを送信するようなケースはあまりありませんが、複雑なデータを処理する際に利用する場合があります。ですが、XML を組み立てる処理が複雑になりため、JavaScript で処理しやすい JSON 形式のデータを利用することをお勧めします。

5.1.6.1. JAXB オブジェクトの作成

「5.1.5.1　JAXB の Java オブジェクトの作成」と同じであるため、詳細はそちらを参照してください。

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;


@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "id",
    "value"
})
@XmlRootElement(name = "SampleJaxb1")
public class SampleJaxb1 {

    protected String id;
    protected Integer value;

    public String getId() {
        return id;
    }

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

}
```

・JAXB のプロパティの属性の定義。
・この場合、プロパティは子要素とする。

5.1.6.2. サーバ側（Controller）

XML 形式のデータをサーバ側で受信する方法を説明します。

- 引数は、「5.1.6.1」で作成した JAXB 形式のオブジェクトを指定し、「**@RequestBody**」を付与します。
  - メソッドの引数に@RequestBody を付与した場合、オブジェクトの形式が JAXB の場合、自動的に HTTP ヘッダーの ContentType が「application/xml」になります。
- リクエスト（入力値）が XML、JSON かどうか、パッと見ではソースコード上は区別がつかないため、次の方法でメディアタイプを明示した方がよいです。
  - リクエスト時のメディアタイプを指定したい場合は、@RequestMapping(…,**headers="Content-Type=application/xml "**))で指定します。
  - または、Spring3.1 で追加になった「@RequestMapping(…,**consumes ="application/xml"**))」で指定します。
- データバインドエラーが発生した場合は、HTTP ステータスコード「400（Bad Request）」が発生します。
  - JAXB の場合、定義方法にもよりますが、非整形式やルート要素が見つからないような致命的なエラーの場合でしかエラーがは起きません。子要素とオプション設定にしていると、バインドできない場合は null が設定され、特にエラーは起きません。

```
@Controller
public class XmlJaxbController {

    @RequestMapping(value="/ajax/xmlIn1")
//  @RequestMapping(value="/ajax/xmlIn1", headers="ContentType=application/xml")
//  @RequestMapping(value="/ajax/xmlIn1", consumes="application/xml")
    public ModelAndView xmlIn1(@RequestBody SampleJaxb1 command) {

        System.out.printf("cd=%s¥n", command);

        ModelAndView mav = new ModelAndView("/ajax/sample1");
        mav.addObject("xmlData", command);
        return mav;

    }

}
```

・メディアタイプを明示的に指定できます。
・“consumes” は、Spring3.1 から利用可能です。

5.1.6.3.　クライアント側（jQuery）

JavaScript のライブラリ「jQuery」を使用して、クライアント側で XML 形式のデータを送信する方法を説明します。

- jQuery を利用する場合、「ajax(...)」メソッドでデータを送信します。
- リクエストする送信時のメディアタイプを、「contentType」で“application/xml”と指定します。
  - リクエストデータが XML であるため、文字列として組み立てる必要があります。
  - contentType の値として、文字エンコーディングも明示的に指定した方が良いです。
- 他は、「5.1.3.2　クライアント側（jQuery）」の JSON の場合と同様です。

```
<script type="text/javascript">
$(document).ready(function(){
    $('#xmlIn1').click(function(){
        $.ajax({
            type: "POST",
            url : "${appUrl}/ajax/xmlIn1.html",
            // 送信時のメディアタイプ
            contentType: "application/xml;charset=UTF-8",
            data : '<SampleJaxb1><id>猫</id><value>2</value></SampleJaxb1>',
            // 受信時のメディアタイプ
            dataType: "html",
            success:function(data){
                alert(data);
            },
            error:function(XMLHttpRequest, textStatus, errorThrown){

                // エラーのタイプ（timeout, error, notmodified, parsererror）
                alert("textStatus=" + textStatus);

                // エラーメッセージ
                alert("errorThrown=" + errorThrown);
            }
        });
    });
});
</script>
<ol>
    <li> <span id="jsonIn1">JSON 形式を送信、HTML を受信</span></li>
</ol>
```

・送信時のメディアタイプを“contentType”で明示的に指定できます。
・XML データを送信したい場合は、テキストベースで XML を組み立てる必要がります。

・入力値エラーや、そのほかの通信エラーが発生した場合のエラー処理です。

5.1.6.4.　クライアント側（RestTemplate）

RestTemplate を使用して、クライアント側で XML 形式のデータを送信する方法を説明します。送信するデータが XML データである以外、「5.1.3.3　クライアント側（RestTemplate）」の JSON の場合と同様です。

- RestTemplate は、URL ベースでデータを送信することを前提としているため、今回の場合は汎用的にアクセスできるメソッドとして、「**RestTemplate#exchange(...)**」を利用します。
- Java オブジェクトを XML 形式自動的に変換して送信するために、「**HttpEntity**」で送信データである JAXB の Java オブジェクトを設定します。
  - 送信データを指定するために「**HttpHeaders#setContentType(...)**」でメディアタイプとして「application/xml」を指定します。
  - HttpHeaders でメディアタイプを指定すると、Controller 側と同様に HttpMessageConverter で XML 形式に自動的に変換されます。
- 戻り値は、「ResponseEntity」で取得します。戻り値は、HTML(text/html)で文字列であるため、Generics のクラスタイプは String で指定します。
  - 受信したデータは、「**ResponseEntity#getBody()**」で取得できます。
  - ただし、入力値エラーなど発生した場合と区別するために、HTTP ステータスコード判定します。
- プロキシ経由でアクセスする環境や認証が必要な URL にアクセスする場合は、HttpClient のインスタンスを「RestTemplate」に設定します。詳細は、「5.6　クライアント側「RestTemplate」を参照してください。
  - 通常は、RestTemplate のインスタンスは Spring Bean として登録して利用します。

```
public class RestTemplateClient {

    public void postXml1() {

        try {
            RestTemplate restTemplate = new RestTemplate();

            // 入力データの作成
            SampleJaxb1 inputData = new SampleJaxb1();
            inputData.setId("猫");
            inputData.setValue(2);

            // リクエスト時の HTTP ヘッダーを設定（ContentType を指定）
            HttpHeaders rqeustHeaders = new HttpHeaders();
            rqeustHeaders.setContentType(MediaType.APPLICATION_XML);

            // リクエスト時の HTTP データの作成（HTTP ヘッダーと Body(入力データ)）を指定。
            HttpEntity<SampleJaxb1> requestEntity =
                    new HttpEntity<SampleJaxb1>(inputData, rqeustHeaders);

            ResponseEntity<String> responseData = restTemplate.exchange(
                    "http://localhost:8080/spring-mvc-3.1/ajax/xmlIn1.html",
                    HttpMethod.POST,
                    requestEntity,
                    String.class);
```

・送信データを JAXB の Java オブジェクトで作成します。
・Controller 側の@RequestBody で指定したデータからと同じものを使用します。

・HTTP ヘッダーの ContentType を指定します。
・リクエストデータを作成します。

・サーバからのデータを取得します。

```
            if(responseData.getStatusCode() == HttpStatus.OK) {
                System.out.println(responseData.getBody());

            } else if(responseData.getStatusCode() == HttpStatus.BAD_REQUEST) {
                // 入力データが不正な場合
                System.out.println(responseData.getBody());
            }

        } catch(RestClientException e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.1.7. サーバクライアント側の両方で XML データ送受信する

「5.1.5　サーバ側で XML データを出力／クライアント側で取得（JAXB）」「5.1.6　クライアント側で XML データを送信／サーバ側で受信する場合」で説明した XML データの送受信を組み合わせた方法を説明します。

5.1.7.1. JAXB オブジェクトの作成

【サーバへ送信する XML の JAXB の定義】

- 「5.1.5.1　JAXB の Java オブジェクトの作成」と同じです。

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "id",
    "value"
})
@XmlRootElement(name = "SampleJaxb1")
public class SampleJaxb1 {

    protected String id;
    protected Integer value;

    public String getId() {
        return id;
    }

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

}
```

・JAXB のプロパティの属性の定義。
・この場合、プロパティは子要素とする。

【サーバから受信する XML の JAXB の定義】

- “SampleJaxb2”をルート要素として、子要素としてリスト形式の“Result”を定義します。

```
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

/**
 * root 要素の定義
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
	"result",
	"processingTime"
})
@XmlRootElement(name = "SampleJaxb2")
public class SampleJaxb2 {

	@XmlElement(name = "Result")
	protected List<Result> result;
	protected Double processingTime;

	// getter, setter は省略
}

/**
 * 子要素の定義
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
	"id",
	"name",
	"age"
})
@XmlRootElement(name = "Result")
public static class Result {

	@XmlElement(required = true)
	protected String id;
	@XmlElement(required = true)
	protected String name;
	protected int age;

	// getter, setter は省略
}
```

5.1.7.2. サーバ側（Controller）

XML 形式データをサーバ側で送受信する方法を説明します。

- メソッドと引数に対してアノテーション「@ResponseBody」と「@RquestBody」を設定します。
- メディアタイプを指定したい場合は、「headers」を使用します。
  - Spring3.1 からは、「consumes」「produces」でも指定可能です。
- リクエストやレスポンス時のデータの送受信の動作を細かく指定したい場合は、引数として「HttpEntity」、戻り値として「ResponseEntity」を使用します。

【Controller】

・メディアタイプを明示的に指定できます。
・“consumes”、“produces”は、Spring3.1 から利用可能です。

```
@Controller
public class XmlJaxbController {

    @RequestMapping(value="/ajax/xmlInOut1")
    //@RequestMapping(value="/ajax/xmlInOut1",
            headers={"ContentType=application/xml", "Accept=application/xml"})
    //@RequestMapping(value="/ajax/xmlInOut1", consumes="application/xml", produces="application/xml")
    @ResponseBody
    public SampleJaxb2 xmlInOut1(@RequestBody SampleJaxb1 command) {

        long startTime = System.currentTimeMillis();
        System.out.printf("cd=%s¥n", command);

        // 出力用の JAXB データの作成
        SampleJaxb2 responseData = new SampleJaxb2();
        for(int i=0; i < 3; i++) {
            Result res = new Result();
            res.setId(String.format("id-%d", i));
            res.setName(String.format("name-%d", i));
            res.setAge(i);

            responseData.getResult().add(res);
        }

        long endTime = System.currentTimeMillis();
        responseData.setProcessingTime((endTime - startTime)/1000.0);

        return responseData;

    }
}
```

5.1.7.3. クライアント側（jQuery）

JavaScript のライブラリ「jQuery」を使用して、クライアント側で XML 形式のデータを送受信する方法を説明します。

- jQuery を利用する場合、「ajax(...)」メソッドでデータを取得します。
- リクエストする送信時のメディアタイプを「contentType」で“application/json”と指定します。
  - 指定すると、「data」で指定したデータが XML になります。
- レスポンスの受信時のメディアタイプを「dataType」で“xml”と指定します。

```
<script type="text/javascript">
$(document).ready(function(){
	$('#xmlInOut1').click(function(){
		$.ajax({
			type: "POST",
			url : "${appUrl}/ajax/xmlInOut1.html",
			// 送信時のメディアタイプ
			contentType: "application/xml;charset=UTF-8",
			data : '<SampleJaxb1><id>猫</id><value>1</value></SampleJaxb1>',
			// 受信時のメディアタイプ
			dataType: "xml",
			success:function(data){
				alert($(data).text());
			},
			error:function(XMLHttpRequest, textStatus, errorThrown){
				// エラーのタイプ（timeout, error, notmodified, parsererror）
				alert("textStatus=" + textStatus);

				// エラーメッセージ
				alert("errorThrown=" + errorThrown);
			}
		});
	});
});
</script>
<ol>
	<li><span id="xmlInOut1">XML 形式を送受信(jQuery 経由)</span></li>
</ol>
```

【取得した XML データの例】

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SampleJaxb2>
	<Result><id>id-0</id><name>name-0</name><age>0</age></Result>
	<Result><id>id-1</id><name>name-1</name><age>1</age></Result>
	<Result><id>id-2</id><name>name-2</name><age>2</age></Result>
	<processingTime>0.01</processingTime>
</SampleJaxb2>
```

5.1.7.4.　クライアント側（RestTemplate）

RestTemplate を使用して、クライアント側で XML 形式のデータを送受信する方法を説明します。

- RestTemplate は、URL ベースでデータを送信することを前提としているため、今回の場合は汎用的にアクセスできるメソッドとして、「RestTemplate#exchange(…)」を利用します。
- 送信データは、「HttpEntity」で組み立てます。
  - 送信時のデータ形式を「HttpHeaders#setContentType(…)」で指定します。
- 受信データは、JAXB の Java オブジェクトの場合、自動的にマッピングされた形で取得できます。
  - JSON 形式の場合ととは異なり、XML が階層化されていても、完全にマッピングされた形で取得できます。

```
public class RestTemplateClient {
    public void postXmlInOut1() {

        try {
            RestTemplate restTemplate = new RestTemplate();

            // 入力データの作成
            SampleJaxb1 inputData = new SampleJaxb1();
            inputData.setId("猫");
            inputData.setValue(2);

            // リクエスト時の HTTP ヘッダーを設定（ContentType を指定）
            HttpHeaders rqeustHeaders = new HttpHeaders();
            rqeustHeaders.setContentType(MediaType.APPLICATION_XML);

            // リクエスト時の HTTP データの作成（HTTP ヘッダーと Body(入力データ)）を指定。
            HttpEntity<SampleJaxb1> requestEntity =
                    new HttpEntity<SampleJaxb1>(inputData, rqeustHeaders);

            ResponseEntity<SampleJaxb2> responseData = restTemplate.exchange(
                    "http://localhost:8080/spring-mvc-3.1/ajax/xmlInOut1.html",
                    HttpMethod.POST,
                    requestEntity,
                    SampleJaxb2.class);

            if(responseData.getStatusCode() == HttpStatus.OK) {
                String textXml = convString(responseData.getBody());
                System.out.println(textXml);

            } else if(responseData.getStatusCode() == HttpStatus.BAD_REQUEST) {
                // 入力データが不正な場合
                System.out.println(responseData.getBody());
            }

        } catch(RestClientException e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    // JAXB オブジェクトをテキストベースの XML に変換する
    public static String convString(SampleJaxb2 obj) {
```

・送信データを JAXB の Java オブジェクトで作成します。
・Controller 側の@RequestBody で指定したデータからと同じものを使用します。

・HTTP ヘッダーの ContentType を指定します。
・リクエストデータを作成します。

・サーバからのデータを取得します。

```
        try {
            JAXBContext context = JAXBContext.newInstance(SampleJaxb2.class.getPackage().getName());
            Marshaller marshaller = context.createMarshaller();
            StringWriter out = new StringWriter();
            marshaller.marshal(obj, out);

            return out.toString();
        } catch(Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

5.2. RESTful なシステム設計

RESTful なシステムは乱暴に言えば、「HTTP プロトコルを上手に利用して、URI を通じてリソースにアクセスするもの」と言えます。以下に、RESTful サービスを作成する上でのよくある間違いやポイントを示します。

(1) POST メソッドを乱用しないで、「GET、DELETE、PUT」を場面に合わせて使う。
  (ア) POST は、データの送信、受信の両方も柔軟にできるため、全て POST で実装したくなります。しかし、HTTP のメソッドには、「GET、DELETE、PUT」などを複数のもがあり、それらは CRUD に対応しています。
  (イ) わざわざ、CRUD に対応するものがあるならばそれらを使うべきです。
  (ウ) HTTP メソッドごとの役割については、「5.2.2　REST サービスにおける HTTP メソッドの役割」を参照してください。
(2) HTTP ステータスコードでリクエストの間違いやリソースの状態を表現する。
  (ア) リクエストの URI が間違っている場合、“400（Bad Request)”を返すべきです。
  (イ) よくある間違ったケースとして、リクエストの内容が間違っていた場合、“200（OK)”を返し、レスポンスのデータにエラーコードやエラー内容を埋め込んだりすることです。
  (ウ) HTTP ステータスコードの使い分けは、「5.2.2　REST サービスにおける HTTP メソッドの役割」を参照してください。
(3) リソースに１つ１つに対して、URI はユニークであるべき。
  (ア) 例えば、「a01」という情報（name=a01）がある場合、「http://～/name/a01」いうように、URI でリソースを特定できるようにする。
  (イ) また、付随する情報「b01」は、「http://～/name/a01/info/b01」というようにアクセスすべきです。
  (ウ) URI の設計については、「5.2.1　REST サービスにおける URI の決め方」を参照してください。
(4) URI にアクションを入れない。
  (ア) 「http://～/sample?action=delete」のようなクエリストリングは用いず、また URI 自体にもアクション（動詞）は含めない。アクションを表現したい場合は、HTTP メソッドの「GET、DELTE、PUT」などを利用すべきです。
  (イ) URI の設計については、「5.2.1　REST サービスにおける URI の決め方」「5.2.2　REST サービスにおける HTTP メソッドの役割」を参照してください。

(5) ステートレスであるべきで、セッションには依存しない。

(ア) HTTP 自体がステートレス（一つ一つの通信において、以前の状態を保持しない）であるため、REST もそのようにあるべきです。

(イ) 例えば、商品をサイトのカートにおいて、ステートフルの場合は、サーバがカート情報を保持しているため、商品を追加する際に商品を 1 つずつサーバに情報に送ればよい。ステートレスの場合は、カート情報を追加する場合、毎回全ての情報をサーバへ送り、カートの情報は、サーバ側は保持しないでクライアント側で全て持ち、サーバの機能（ここではセッション機能）に依存しない。

(ウ) 動作としては、WEB ブラウザで URI をブックマークに登録しておいて、その URI にアクセスする度に同じ結果が返ってくる。すなわち Boolmarkable でもあるべきす。

(エ) セッションを使いたければ、REST は使わずに通常の WEB アプリケーションを利用するか、SOAP などを利用するべきです。

【参考】

- http://www.geocities.jp/yamamotoyohei/rest/mistakes.html
- http://yohei-y.blogspot.jp/2005/05/rest-8-rest.html

5.2.1. REST サービスにおける URI の決め方

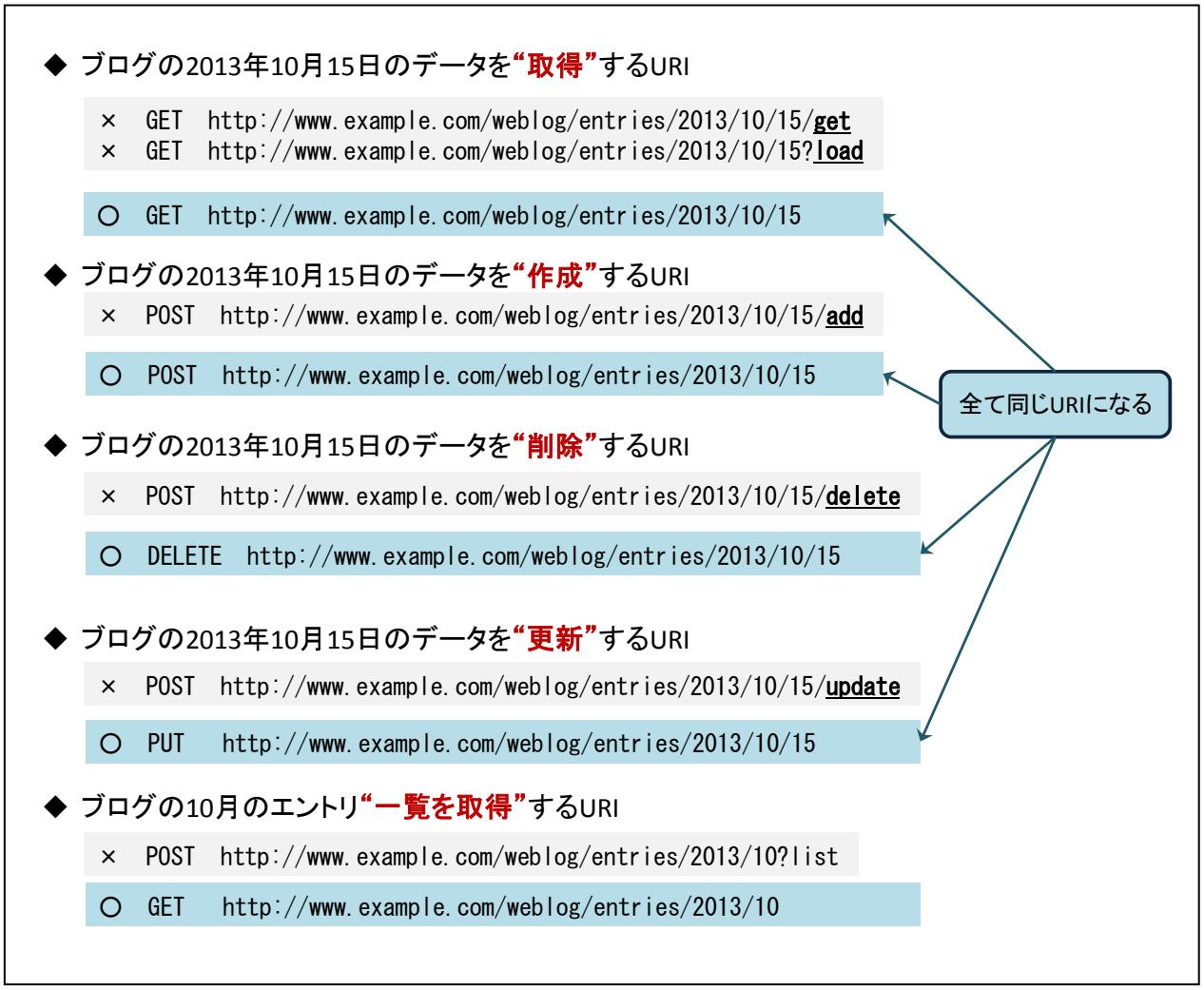
RESTfu サービスにおける「アドレス可読性（Addressability)」に当たるものです。

【参考 URL】

- http://www.slideshare.net/t_wada/restful-web-design-review

5.2.1.1. URI に動詞を含めてはならない

- URI に、「get/read」「delete/remove」「add」「update」などを使用しない。これらを表現したい場合は、HTTP メソッドの「GET」「DLETE」「PUT」で区別すべきです。
- 「confirm⇒confirmation」のようになるべく名詞にする。
  - ただし、「copy」「use」のように名詞と動詞が同じ場合は、他の語を探すか、例外扱いとする。



5.2.1.2.　URI は階層的で予測可能であった方がよい

- URI の右にいくほど情報が詳細になる、または自然な階層構造になっていく。

◆ ブログの2013年10月15日を表現するURI

○　http://www.example.com/weblog/entries/2013/10/15

抽象的 ⟷ 具体的

◆ ファイル“hoge”を{src}から{dest}へコピーする

×　http://www.example.com/files/hoge/copy/{src}/{dest}

階層構造でない

◆ 地図の座標（緯度={lat}、経度={lng}）を取得する

×　http://www.example.com/map/{lat}/{lng}

階層構造でなく並列な情報

5.2.1.3.　階層的でない情報はカンマ“,”またはセミコロン“;”を使用する

- 順序性がある場合、カンマ“,”で区切り表現する。
- 順序性がない場合、セミコロン“;”で区切り表現する。
- これらの表現は「マトリックス URI（RFC 3986）」と呼ばれいる。

◆ 地図の座標（緯度={lat}、経度={lng}）を取得する（順序性がある）

http://www.example.com/map/{lat},{lng}

順序が関係ある順列

◆ 地図の座標（緯度={lat}、経度={lng}）を取得する（順序性がない）

http://www.example.com/map;lag={lat};lng={lng}

順序は関係ないが名前がある連想配列（マップ）

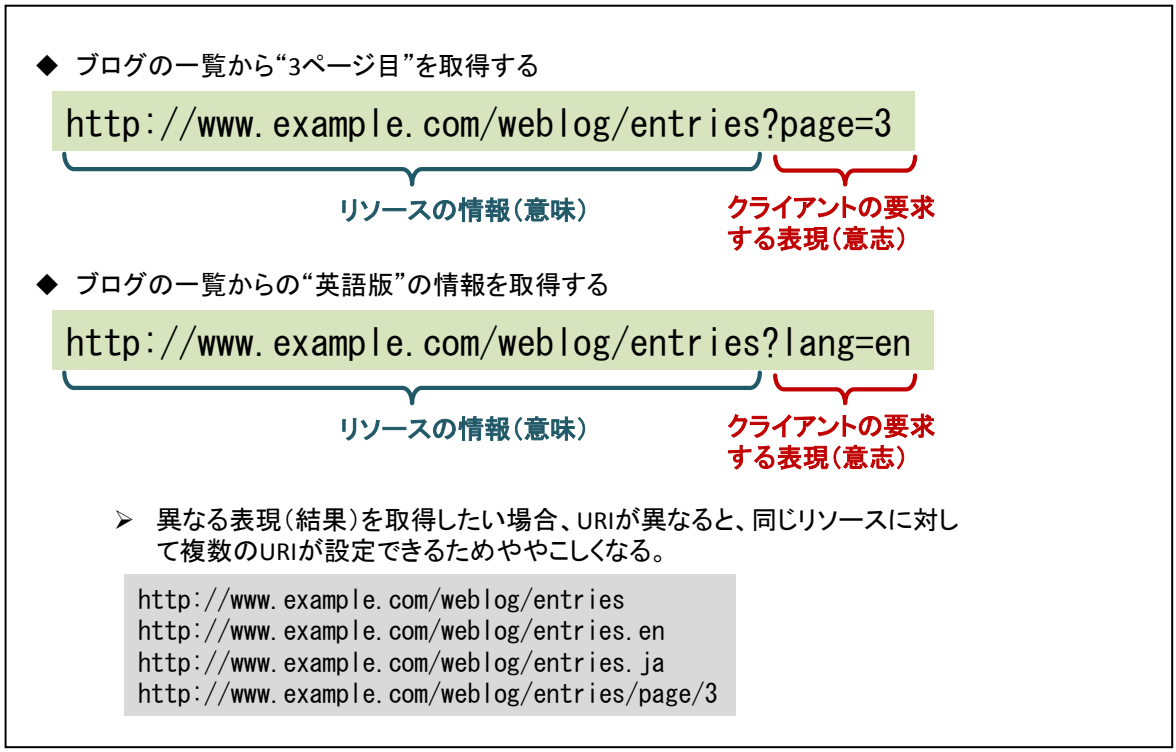
◆ 色を設定する（順序性がない）

http://www.example.com/colorpair/red;blue

順番は関係ない集合

5.2.1.4. クエリストリングを全て除去してもリソースにアクセス可能である

- GET などの読み取り専用のリソースの URI に対して、クエリストリングを取り去ってもアクセスできるリソースは同じようにすべきです。
  - これは、“表現（representation）の選択によるコンテンツネゴシエーション”と呼ばれ、クライアント側が設定すべきで、既存のリソースに対して影響を与えるようなものではありません。
- 異なる表現（結果）を取得したい場合、クエリストリングではなく URI の一部として埋め込んだ場合、同じリソースに対して複数の URI を持つことになるためややこしくなります。
- 表現の選択をする際に、クエリストリングではなく“Accept-Language”などの“HTT のリクエストヘッダ”で設定してもかまわないが、表現がパッと見で理解しづらくなるためあまりお勧めしない。



5.2.1.5. その他の URI を設計する際の考え方

- リソースを特定するにはユニークな ID を利用するが、DB などの内部的な主キーである UUID は使わず、公開されているようなコードなどを使った方がよい。
  - 例）本などの ISBN はユニークだが、公開されている一般的な値。

5.2.2. REST サービスにおける HTTP メソッドの役割

RESTful サービスの特性として、「統一インタフェース（Uniform Interface）」に当たるものです。あるリソースに対する操作を行う際に、統一されたインタフェース（または、方法）の HTTP のメソッドを利用することです。

- http://www.infoq.com/jp/articles/designing-restful-http-apps-roth

表 5.1 HTTP/1.1 の主なメソッド

No.	メソッド	利用目的	安全 (※1)	冪等 (※2)	参照
1	GET	リソースの取得。 ・GET でのアクセスはリソースの内容に影響を与えない。	○	○	5.2.2.1
2	DELETE	リソースの削除。	×	○	5.2.2.2
3	PUT	既存のリソースの更新。 ・リソースを完全に新しい情報に置換する。	×	○	5.2.2.3
4	POST	リソースの新規作成。 ・リソースを部分的に更新する。 ・PATCH が利用できない場合は、POST を利用する。 ・GET、DELETE、PUT に当てはまらない場合は、POST を利用する。	×	×	5.2.2.4
5	PATCH	リソースの部分更新 ・PUT とは異なり、一部のデータのみ更新する実験的なメソッドです。 ・Spring MVC は、v3.2 から対応しました。	×	×	—
6	HEAD	サービスが利用可能であるかどうか、HTTP ステータスコードをみて確認する際などに利用します。 ・レスポンスのボディを含まない GET と同様の動作をする。	○	○	—
7	OPTIONS	利用可能なメソッド一覧を取得する。 ・セキュリティ上、DELETE、PUT などが制限されている場合があるため、問い合わせるのに使用します。	—	—	—

※1 「REST における“安全（safe）”」：操作対象のリソースの状態を変化させず、副作用がないこと。

※2 「REST における“冪等（べきとう）（idempotent）”」：ある操作を何回行っても結果が同じである。PUT や DELETE は、同じリソースに何回発行しても、必ず同じ結果（リソースが更新、削除）が得られる。

- HTTP メソッドごとに役割があるため、インタフェースとしての送受信するデータもそれぞれに決まってきます。
  - サービスを実装する際には、「表 5.2」に示す HTTP メソッドに見合ったインタフェースで実装してください。
- GET、DELETE、PUT に見合わないような場合は、汎用的な POST を利用します。
  - 例えば、データを削除する場合、関連するデータを返してほしい場合などです。

表 5.2 HTTP メソッドごとの REST サービスにおける推奨するインタフェース

No.	メソッド	リクエストデータ	レスポンスデータ
1	GET	・なし。 ・ただし、クエリストリングで URL の一部に埋め込むのは可能。 ・リソースの選択は URL のパス変数で指定する。	・XML/JSON などのオブジェクト
2	DELETE	・なし。 ・ただし、クエリストリングで URL の一部に埋め込むのは可能。 ・リソースの選択は URL のパス変数で指定する。	・なし。 ・結果は、HTTP ステータスコードで判定すべき。
3	PUT	・XML/JSON などのオブジェクト。 ・リソースの選択は URI にパス変数などで埋め込む。 ・更新対象のデータ全体を送る。	・なし。 ・結果は、HTTP ステータスコードで判定すべき。
4	POST	・XML/JSON などのオブジェクト。 ・作成するデータ全体を送る。	・XML/JSON などのオブジェクト ・作成したデータ全体を返す。
5	PATCH	・XML/JSON などのオブジェクト。 ・更新対象のデータ一部を送る。 ・リソースの選択は URL のパス変数で指定する。	・XML/JSON などのオブジェクト ・更新したデータ全体を返す。
6	HEAD	・なし。 ・ただし、クエリストリングで URL の一部に埋め込むのは可能。	・HTTP のレスポンスヘッダー。
7	OPTIONS	・なし。 ・ただし、クエリストリングで URL の一部に埋め込むのは可能。	・利用可能な HTTP メソッドのリスト。

5.2.2.1. GET メソッド

- リソースを取得するために使用します。
- 安全かつ、冪等です。
  - アクセスしたリソースに影響を与えず、また、常に同じ結果を返します。

表 5.3 返却すべき HTTP ステータスコード（GET メソッド）

No.	HTTP ステータスコード	説明
1	200 (OK)	レスポンスとして表現が送信された。
2	204 (no content)	リソースが空の表現をもつ。
3	301 (Moved Permanently)	リソース URI が更新された。
4	303 (See Other)	ロードバランシングなど。
5	304 (not modified)	リソースが更新されていない (キャッシング)。
6	400 (bad request)	不正なリクエストを表す (間違ったパラメータなど)。
7	404 (not found)	リソースが存在しない。
8	406 (not acceptable)	サーバは要求された表現をサポートしない。
9	500 (internal server error)	汎用的なエラーレスポンス。
10	503 (Service Unavailable)	サーバが現在、リクエストを処理できない。

5.2.2.2. DELETE メソッド

- リソースを削除するために使用します。
- 安全でないが、冪等です。
  - アクセスしたリソースに影響しますが、同じリソースにアクセスする限り同じ“削除した”という結果を返す。

表 5.4 返却すべき HTTP ステータスコード（DELETE メソッド）

No.	HTTP ステータスコード	説明
1	200 (OK)	リソースが削除された。
2	301 (Moved Permanently)	リソース URI が更新された。
3	303 (See Other)	ロードバランシングなど。
4	400 (bad request)	不正なリクエストを表す (間違ったパラメータなど)。
5	404 (not found)	リソースが存在しない。
6	500 (internal server error)	汎用的なエラーレスポンス。
7	503 (Service Unavailable)	サーバが現在、リクエストを処理できない。

5.2.2.3. PUT メソッド

- リソースを更新するために使用します。
  - 既存のリソースを完全に新しい情報に置換します。
- 安全でないが、冪等です。
  - アクセスしたリソースに影響しますが、同じリソースにアクセスする限り同じ“更新した”という結果を返す。

表 5.5 返却すべき HTTP ステータスコード（PUT メソッド）

No.	HTTP ステータスコード	説明
1	200 (OK)	既存のリソースが更新された。
2	201 (created)	新しいリソースが作成された。
3	301 (Moved Permanently)	リソース URI が更新された。
4	303 (See Other)	ロードバランシングなど。
5	400 (bad request)	不正なリクエストを表す (間違ったパラメータなど)。
6	404 (not found)	リソースが存在しない。
7	406 (not acceptable)	サーバは要求された表現をサポートしない。
8	409 (conflict)	一般的な衝突（楽観的排他)。
9	412 (Precondition Failed)	条件付きの更新時の衝突など（楽観的排他)。
10	415 (unsupported media type)	受信した表現をサポートしていない。
11	500 (internal server error)	汎用的なエラーレスポンス。
12	503 (Service Unavailable)	サーバが現在、リクエストを処理できない。

5.2.2.4. POST メソッド

- リソースを新規作成するために使用します。
  - PATCH メソッドが利用できない場合は、PUT に対して、リソースを部分的に更新します。
  - GET、DELETE、PUT に当てはまらない処理の場合にも POST を利用します。
- 安全でないく、冪等でもありません。
  - アクセスしたリソースに影響し、新規作成のため、毎回異なる結果を返します。

表 5.6 返却すべき HTTP ステータスコード（POST メソッド）

No.	HTTP ステータスコード	説明
1	200 (OK)	既存のリソースが更新された（部分更新)。
2	201 (created)	新しいリソースが作成された。

3	202 (accepted)	処理が受け入れられたがまだ終了していない(非同期処理)。
4	301 (Moved Permanently)	リソース URI が更新された。
5	303 (See Other)	ロードバランシングなど。
6	400 (bad request)	不正なリクエストを表す (間違ったパラメータなど)。
7	404 (not found)	リソースが存在しない。
8	406 (not acceptable)	サーバは要求された表現をサポートしない。
9	409 (conflict)	一般的な衝突（楽観的排他）（部分更新)。
10	412 (Precondition Failed)	条件付きの更新時の衝突など（楽観的排他）（部分更新)。
11	415 (unsupported media type)	受信した表現をサポートしていない。
12	500 (internal server error)	汎用的なエラーレスポンス。
13	503 (Service Unavailable)	サーバが現在、リクエストを処理できない。

5.2.2.5. PATCH メソッド

- リソースを部分的に更新するために使用します。
  - PUT の既存のリソースを完全に新しい情報に置換に対して、部分的に更新します。
  - PATCH が利用できない場合は、POST を利用します。
- 安全でなく、冪等でもありません。
  - アクセスしたリソースに影響し、かつ同じリソースにアクセスしても、楽観的排他用のキーの更新などで、毎回“異なる”結果を返す。

表 5.7 返却すべき HTTP ステータスコード（PATCH メソッド）

No.	HTTP ステータスコード	説明
1	200 (OK)	既存のリソースが更新された。
2	301 (Moved Permanently)	リソース URI が更新された。
3	303 (See Other)	ロードバランシングなど。
4	400 (bad request)	不正なリクエストを表す (間違ったパラメータなど)。
5	404 (not found)	リソースが存在しない。
6	406 (not acceptable)	サーバは要求された表現をサポートしない。
7	409 (conflict)	一般的な衝突（楽観的排他)。
8	412 (Precondition Failed)	条件付きの更新時の衝突など（楽観的排他)。
9	415 (unsupported media type)	受信した表現をサポートしていない。
10	500 (internal server error)	汎用的なエラーレスポンス。
11	503 (Service Unavailable)	サーバが現在、リクエストを処理できない。

5.2.2.6.　PUT と POST の使い分け

基本的に PUT は更新で、POST は新規作成だが、特に PUT はリソースへ影響を与える考え方が複雑なので、その説明をする。

表 5.8　URI に対する PUT のアクション

URI	PUT		POST
	新しいリソース	既存のリソース	
/weblog（※1）	― （リソースは既に存在する）	無効	新しいブログを作成する
/weblog/myblog	このブログを作成する	このブログの設定を変更する	新しいブログエントリを作成する
/weblog/mylog/entries/1	― （このエントリは存在しない）	このブログのエントリを編集する	このブログエントリへコメントを投稿する

※1　既に“/weblog”という URI は存在している状態

5.3. Spring MVC における RESTful サービスの実現

「5.2　RESTful なシステム設計」で説明した RESTful なサービスを作成するに当たり、Spring MVC でどのように実装するかを説明します（図 5.1　REST サービスによるデータの送受信の概要）。

- Spring MVC における XML や JSON データのやり取りの中核をなすのが、「HttpMessageConverter」です。
  - Controller で受信／送信するデータは Java オブジェクトですが、WEB ブラウザなどのクライアント側では、XML や JSON 形式で処理します。Java オブジェクトとそれらのデータ形式を相互に変換する役目をするのが HttpMessageConverter です。
  - 詳細は、「5.3.1　データの変換「HttpMessageConverter」」を参照してください。
- RESTful な URI を実現するために、Controller での URI の定義において「@PathVariable」を利用します。
  - Spring 3.2 からはマトリックス URI に対応し、「@MtrixVariable」を利用します。
  - 詳細は、「5.4　RESTful な URI の」を参照してください。
-

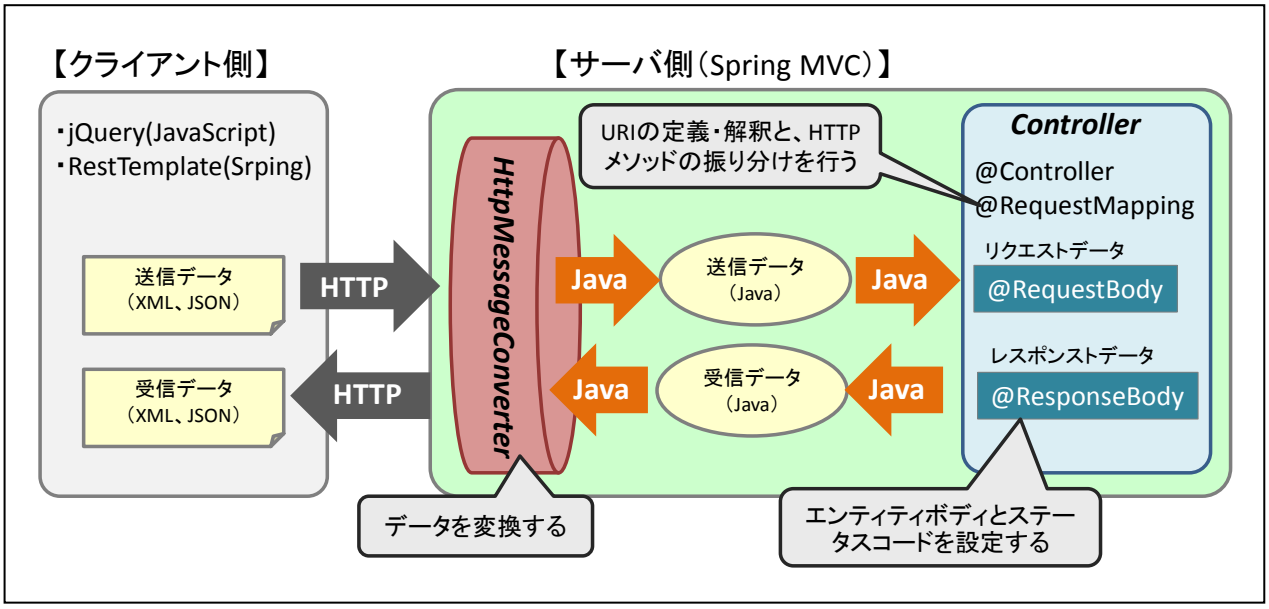


図 5.1　REST サービスによるデータの送受信の概要

5.3.1. データの変換「HttpMessageConverter」

Spring MVC の HttpMessageConverter では様々なものが用意されています（表 5.9　標準で用意されている HttpMessageConverter）。

- 利用するは、<mvc:message-converter>に登録しますが、予め初期値として登録されているため、実際には何もする必要はありません（図 5.2　HttpMessageConverter の登録）。
- XML 形式のデータの送受信では、JDK1.6 から標準採用された JAXB を使用します。
  - ➢　これは、XML⇔Java オブジェクトを相互に変換／マッピングするためのライブラリ。
- JSON 形式のデータの送受信では、ライブラリ Jackson を使用します。
  - ➢　これは、JSON⇔Java オブジェクトを相互に変換／マッピングするためのライブラリ。
- 自分の独自にの形式の Converter を作成することで自由にカスタマイズが可能で、そのときには登録が必要になります。

【参考】

- http://www.ibm.com/developerworks/jp/web/library/wa-restful/index.html

表 5.9　標準で用意されている HttpMessageConverter

No.	クラス	説明
1	StringHttpMessageConverter	・文字列で相互変換する。 ・メディアタイプは、「text/*」「text/plain」。
2	FormHttpMessageConverter	・FORM データを相互変換する。 ・メディアタイプは、「application/x-www-form-urlencoded」。
3	ByteArrayHttpMessageConverter	・byte 配列（byte[ ]）を相互変換する。 ・メディアタイプは、「*/*」、「application/octet-tream」。
4	MarshallingHttpMessageConverter	・XML を相互変換する上位クラス。 ・実際には、「Jaxb2RootElementHttpMessageConverter」を利用　する。 ・Spring の“org.springframework.oxm”パッケージの Marsheller／Unmarshaller を利用する。 ・メディアタイプは、「text/xml」「application/xml」。
5	MappingJacksonHttpMessageConverter	・JSON データを相互変換する。 ・別途ライブラリ「Jackson」必要。 ・メディアタイプは、「application/json」。
6	SourceHttpMessageConverter	・Java の「javax.xml.transform.Source」クラスで XML を相互変換する。 ・メディアタイプは、「application/xml」。

7	BufferedImageHttpMessageConverter	・Java の「java.awt.image.BufferedImage」クラスで画像を相互変換する。
8	Jaxb2RootElementHttpMessageConverter	・JAXB を利用した XML を相互変換する。 ・変換対象の Java オブジェクトには、アノテーション「XmlRootElment」「XmlType」が付与されている必要がある。 ・メディアタイプは、「text/xml」「application/xml」。
9	AtomFeedHttpMessageConverter	・フィードの一種である Atom をやり取りする。 ・別途ライブラリ「ROME」が必要。 ・メディアタイプは、「application/atom+xml」。
10	RssChannelHttpMessageConverter	・フィードの一種である RSS をやり取りする。 ・別途ライブラリ「ROME」が必要。 ・メディアタイプは、「application/rss+xml」。。

```
<bean>
・・・(省略)・・・
    <mvc:annotation-driven>
        <!-- HttpMessageConverter の登録(通常は必要ありません) -->
        <mvc:message-converters>
            <bean class="org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter"/>
        </mvc:message-converters>
    </mvc:annotation-driven>
・・・(省略)・・・
</bean>
```

図 5.2　HttpMessageConverter の登録

5.4. RESTful な URI の実装

RESTful な URI を実現するための方法を説明します。URI の一部を変えるための形式「/pet/{id}（{id}は変数名）」は、「URI Template Pattern」と呼ばれ「RFC 6570（http://tools.ietf.org/html/rfc6570）」で定義されています。URI Template の実装は、サーバ側、クライアント側それぞれ実装があるので、REST サービスを作成する際にはにそれらを利用します。

- サーバ側の“Controller”は、アノテーション「**@PathVariable**」を利用して URI を表現します。
  - Spring 3.2 からは、マトリックス URI を処理しやすくするための「**@MatrixVariable**」が導入されました。
- クライアント側の“jQuery”は、プラグイン「**jQuery Uri Templates**」を利用します。これは、既存の ajax(..)メソッドに拡張を加え、パス変数を解釈できるようにしたものです。
  - 「http://jqueryuritemplates.anthonyvanderhoorn.com/」
  - JavaScript による RFC 6570 の実装は他にもありますが、利用する環境によって使い分けてください。
- クライント側の“RestTemplate”は、「**UriComponents/UriComponentsBuilder**」を利用します。
  - 「UriComponents/UriComponentsBuilder」は Sprin3.1 で追加されたものなので、Spring3.0 では、「**UriTemplate**」を利用します。

5.4.1. URI にパス変数が１つの場合

5.4.1.1. サーバ側（Contrller）

- @RequesMapping で定義する URI の定義に、パス変数を「{ 変数名}」として、埋め込みます。
- メソッドの引数にアノテーション「@PathVariable」を付与します。
  - **メソッドの引数の変数名と、URI の定義したパス変数名は合わせる必要**があります。
  - 引数とパス変数の名前が一致しない場合は、アノテーションで変数名を指定する必要があります。「@PathVariable("変数名")」
- 下記の例では URL として「${appUrl}/rest/owners/a01.html」を許可します。

```
@Controller
public class RestUriController {

    // パス変数が１つ
    @RequestMapping(value = "/rest/owners/{userCd}",
            method=RequestMethod.GET, headers="Accept=application/json")
    @ResponseBody
    public UserInfoViewDto findUserById(@PathVariable String userCd) throws DataNotFoundException {

        //省略

    }
}
```

URI のパス変数と引数の変数名は合わせる必要があります。

5.4.1.2. クライアント側（jQuery）

- 既存の ajax(...)メソッドを使用します。
- URL にパス変数を「{変数名}」として埋め込みます。
  - URL は、Spring MVC の DispatcherServlet により拡張子「.html」が付与されるため注意が必要です。
- 引数「tokens」でパス変数に埋め込んだ変数とその値を定義します。

```
<script type="text/javascript" src="${appUrl}/js/lib/jquery-uritemplate.1.0.js"></script>
<script type="text/javascript">
$(document).ready(function(){
    // パス変数が 1 つ
    $('#uriSample1').click(function(){
        $.ajax({
            type: "GET",
            url : "${appUrl}/rest/owners/{userCd}.html",
            tokens: {userCd:'admin'},
            // 受信時のメディアタイプ
            dataType: "json",
            success:function(data){
                alert(data);
            }
        });
    });
});
</script>
<ol>
    <li><span id="uriSample1">URI Template で URI を組み立てる</span></li>
</ol>
```

プラグインを読み込みます。

5.4.1.3. クライアント側（RestTemplate）

- RestTemplate のメソッドのに直接パス変数の値を渡すこともできます。
- UriTemplate を使用する場合は、UriTemplate#expand(...)メソッドでパス変数の値を置換し、「java.net.URI」を取得します。
  - 最後に URI クラスを RestTemplate に渡します。その場合は、パス変数は既に値に変換済みなのでパス変数は必要ありません。
- UriComponentsBuilder を使用する場合は、UriComponentsBuilder#build()メソッドで、UriComponents のインスタンスを取得します。
  - そして、UriComponents#expand()でパス変数をの値を置換し、UriComponents#toURI()で java.net.URI のインスタンスを取得します。UriComponents を利用すると URL エンコードもできます。
  - 最後に URI クラスを RestTemplate に渡します。

```
public class RestTemplateClient2 {

    @Resource
    private RestTemplate restTemplate;
```

```
    public void uriSample1() {

        try {
            // RestTemplate を直接使用する
            UserInfoViewDto responseData = restTemplate.getForObject(
                    "http://localhost:8080/spring-mvc-3.1/rest/owners/{userCd}.html",
                    UserInfoViewDto.class,
                    "admin");
            System.out.println(responseData);

            // UriTemplate を使用する
            UriTemplate uriTemplate =
                    new UriTemplate("http://localhost:8080/spring-mvc-3.1/rest/owners/{userCd}.html");
            URI uri2 = uriTemplate.expand("admin");

            responseData = restTemplate.getForObject(
                    uri2,
                    UserInfoViewDto.class);
            System.out.println(responseData);

            // UriComponents/UriComponentsBuilder を使用する
            UriComponents uriComponents = UriComponentsBuilder
                    .fromUriString("http://localhost:8080/spring-mvc-3.1")
                    .path("/rest/owners/{userCd}.html")
                    .build();;
            URI uri3 = uriComponents.expand("admin").encode().toUri();
            responseData = restTemplate.getForObject(
                    uri3,
                    UserInfoViewDto.class);
            System.out.println(responseData);

        } catch(RestClientException e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

}
```

・URL を分割して組み立てたりでき、UriTemplate よりも柔軟にできます。

5.4.2. URI にパス変数が複数の場合

5.4.2.1. サーバ側（Contrller）

- パス変数を複数指定したい場合は、単に複数記述すればよいです。
  - ただし、変数名はそれぞれユニークにして重複しない名前を付けます。
- パス変数を連続して記述する場合は、「/」や「,」などで区切ってください。
  - パス変数かどうか判別できなくなるため、注意してください。

```
@Controller
public class RestUriController {

    // パス変数 が複数
    @RequestMapping(value = "/rest/infos/{year},{month},{date}/detail",
            method=RequestMethod.GET, headers="Accept=application/json")
    @ResponseBody
    public List<InformationDto> searchInfoByDate(@PathVariable("year") Integer var1,
            @PathVariable Integer month, @PathVariable Integer date) {
        // 省略
    }
}
```

パス変数を複数指定する場合は、それぞれユニークな名前にする。

URI のパス変数と引数の変数名が異なる場合は、名前を指定します。

5.4.2.2. クライアント側（jQuery）

- 既存の ajax(...)メソッドを利用します。
- サーバ側と同様に、URL にパス変数を複数指定します。
  - 別な記述方法として、変数をまとめて記述ですが、サーバ側と書式を合わせたがミスを減らせます。
- パス変数の値は、「tokens」定義し、カンマ「,」で区切りそれぞれ指定します。

```
<script type="text/javascript" src="${appUrl}/js/lib/jquery-uritemplate.1.0.js"></script>
<script type="text/javascript">
$(document).ready(function(){

    // パス変数が複数
    $('#uriSample2').click(function(){
        $.ajax({
            type: "GET",
            url : "${appUrl}/rest/infos/{year},{month},{date}/detail.html",
            // 別の方法もできます
            //url : "${appUrl}/rest/infos/{year,month,date}/detail.html",
            tokens: {year:'2010', month:'1', date:'10'},
            // 受信時のメディアタイプ
            dataType: "json",
            success:function(data){
                alert(data);
            }
        });
    });
});

<ol>
    <li><span id="uriSample2">URI Template で URI を組み立てる(複数の変数)</span></li>
</ol>
```

プラグインを読み込みます。

変数をまとめて別な方法で記述する方法もありますが、サーバ側の記述と合わせた方がよいです。

複数指定する場合は、カンマ「,」で区切り指定します。

5.4.2.3. クライアント側（RestTemplate）

- パス変数に値を Map で組み立てます。
  - 可変長引数（=配列）で RestTemplate に渡せますが、順番に依存するため、Map の方がミスを減らせます。
- RestTemplate、UriTemplate、UriComponetsBuilder などの使い方は、パス変数が 1 つの場合と同じです。

```
public class RestTemplateClient2 {

    @Resource
    private RestTemplate restTemplate;

    public void uriSample2() {
        try {
            // パス変数の値の定義
            Map<String, Object> pathVars = new HashMap<String, Object>();
            pathVars.put("year", 2010);
            pathVars.put("month", 1);
            pathVars.put("date", 10);

            // RestTemplate を直接使用する
            List<InformationDto> responseData = restTemplate.getForObject(
                    "http://localhost:8080/spring-mvc-3.1/rest/infos/{year},{month},{date}/detail.html",
                    List.class,
                    pathVars);
            System.out.println(responseData);

            // UriTemplate を使用する
            UriTemplate uriTemplate =
                    new
UriTemplate("http://localhost:8080/spring-mvc-3.1/rest/infos/{year},{month},{date}/detail.html");
            URI uri2 = uriTemplate.expand(pathVars).normalize();
            responseData = restTemplate.getForObject(
                    uri2,
                    List.class);
            System.out.println(responseData);

            // UriComponents/UriComponentsBuilder を使用する
            UriComponents uriComponents = UriComponentsBuilder
                    .fromUriString("http://localhost:8080/spring-mvc-3.1/")
                    .path("/rest/infos/{year},{month},{date}/detail.html")
                    .build()
                    .normalize();

            URI uri3 = uriComponents.expand(pathVars).encode().toUri();
            responseData = restTemplate.getForObject(
                    uri3,
                    List.class);
            System.out.println(responseData);

        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

パス変数の値を Map で組み立てます。

5.4.3. URI にパス変数と FORM データをクライアントから送信する

URI Template Pattern の URI と FORM データを送信します。

5.4.3.1. サーバ側（Contrller）

【@RequestParam で受け取る】

- @RequestParam で FORM データを受け取り、値が不正でバインドエラーが発生した場合、「400(Bad Request)」が Spring MVC で自動的に返されます。
- RESTful な URI では、クエリストリングなどは“表現の選択”をするために利用する場合があるので、「required=false」としてオプション扱いにする方が無難です。

```
@Controller
public class RestUriController {

    // パス変数と FORM データの組合せ
    @RequestMapping(value = "/rest/infos/{year}",
            method=RequestMethod.GET, headers="Accept=application/json")
    @ResponseBody
    public List<InformationDto> searchInfoByDateAndPage(@PathVariable Integer year,
            @RequestParam(required=false) Integer page) {

        // 省略
    }
}
```

【@ModelAttribute で受け取る】

- @ModelAttribute として FORM データの値をコマンドとして受け取り、値が不正でバインドエラーが発生した場合、エラー内容は BindingResult に格納されます。
  - ➢ 値が不正化どうかの処理を自分でチェックして、ステータスコード 400 を返すべきです。

```
@Controller
public class RestUriController {

    // パス変数とフォームデータの組合せ 2
    @RequestMapping(value = "/rest/infos/{year}",
            method=RequestMethod.GET, headers="Accept=application/json")
    @ResponseBody
    public List<InformationDto> searchInfoByDateAndPage2(@PathVariable Integer year,
            @ModelAttribute RestSampleCommand command, BindingResult bindingResult) {

        //省略
    }
}
```

5.4.3.2. クライアント側（jQuery）

- 既存の ajax(...)メソッドを使用します。
- パス変数の値を「tokens」で指定します。
- 送信したいデータを引数「data」で指定します。
  - JSON 形式の場合は、値を' 'などで囲み文字列としていましたが、この場合は連想配列の形式なので注意してください。
  - GET メソッドの場合、自動的にクエリストリングとして URL に付加されます。
  - POST/PUT/DELETE メソッドの場合、フォーム(application/x-www-form-urlencoded)で送信されます。

```
<script type="text/javascript" src="${appUrl}/js/lib/jquery-uritemplate.1.0.js"></script>
<script type="text/javascript">
$(document).ready(function(){
    // パス変数とフォームデータの組合せ
    $('#uriSample3').click(function(){
        $.ajax({
            type: "GET",
            url: "${appUrl}/rest/infos/{year}.html",
            tokens: {year:'2010'},
            // フォームのデータ
            data: {'page': '4'},
            // 受信時のメディアタイプ
            dataType: "json",
            success:function(data){
                alert(data);
            }
        });
    });
});
<ol>
    <li><span id="uriSample3">URI Template で URI を組み立てる(フォームデータの組合せ)</span></li>
</ol>
```

プラグインを読み込みます。

・ajax(...)メソッドが HTTP メソッドによって、自動的に判断してくれる。
・GET の場合はクエリストリング、その他の場合は FORM で送信される。

【別な記述方法】

- 別の記述として、使ってクエリストリングで記述する方法がありますが、これはあまりお勧めしません。
  - クエリストリングはフォームデータなので、パス変数と区別するために、引数 tokens はパス変数のみを指定した方が、ミスを減らせ、また理解しやすくなります。

```
<script type="text/javascript" src="${appUrl}/js/lib/jquery-uritemplate.1.0.js"></script>
<script type="text/javascript">
$(document).ready(function(){

    // パス変数とフォームデータの組合せ(別な記述)
    $('#uriSample3_').click(function(){
        $.ajax({
            type: "GET",
            url: "${appUrl}/rest/infos/{year}.html{?page}",
            tokens: {year:'2010', page:'4'},
            // 受信時のメディアタイプ
            dataType: "json",
            success:function(data){
                alert(data);
```

クエリストリングとして記述する。

```
            }
        });

    });
});
```

5.4.3.3. クライアント側（RestTemplate）

- jQuery の場合と同じようにクエリストリングに埋め込む方法と、JSON として送信する方法があります。
  - JSON(application/json)で送信されますが、Controller 側で@RequestPram でも取得可能です

```
public class RestTemplateClient2 {

    @Resource
    private RestTemplate restTemplate;

        // パス変数とフォームデータの組合せ
    public void uriSample3() {

        try {

            // クエリストリングにフォームデータを埋め込む
            List<InformationDto> responseData = restTemplate.getForObject(
                    "http://localhost:8080/spring-mvc-3.1/rest/infos/{year}.html?page={page}",
                    List.class,
                    2010, 4);
            System.out.println(responseData);

            // リクエストデータとして送る
            RestSampleCommand requestData = new RestSampleCommand();
            requestData.setPage(4);

            // リクエスト時の HTTP ヘッダーを設定（ContentType を指定）
            HttpHeaders rqeustHeaders = new HttpHeaders();
            rqeustHeaders.setContentType(MediaType.APPLICATION_JSON);

            // リクエスト時の HTTP データの作成（HTTP ヘッダーと Body(入力データ)）を指定。
            HttpEntity<RestSampleCommand> requestEntity =
                    new HttpEntity<RestSampleCommand>(requestData, rqeustHeaders);

            // UriComponents/UriComponentsBuilder を使用する
            UriComponents uriComponents = UriComponentsBuilder
                    .fromUriString("http://localhost:8080/spring-mvc-3.1/")
                    .path("/rest/infos/{year}.html")
                    .build()
                    .normalize();

            URI uri3 = uriComponents.expand(2010).encode().toUri();

            // RestTemplate の汎用的な exchange を使用する
            ResponseEntity<List> responseEntity = restTemplate.exchange(
                    uri3, HttpMethod.GET, requestEntity, List.class);
            System.out.println(responseEntity.getBody());

        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

クエリストリングとして送信する。

JSON として送信する。

5.4.4. URI にパス変数と JSON／XML データを送受信する

クライアント側で JSON 形式を送信し、XML 形式のデータを受信する方法を説明します。

- XML や JSON の扱い方の詳細は、「5.1　JSON／XML データの送受信」を参照してください。

表 5.10　REST サービスで使用される一般的な MIME タイプ

No.	MIME タイプ	コンテンツタイプ
1	JSON	application/json
2	XML	application/xml
3	XHTML	application/xhtml+xml

5.4.4.1. サーバ側（Contrller）

- JSON 形式をサーバ側で受信する場合、引数にアノテーション「@RequestBody」を付与します。
  - 今回は受信するデータが階層構造を持っているため、引数は Java Bean 形式になっています。
  - メディアタイプを“header”や“consumers”で明示的に指定します。
- XML 形式をサーバ側で送信する場合は、メソッドにアノテーション「@ResponseBody」を付与します。
  - メソッドの戻り値となるクラスは、JAXB の Java オブジェクトである必要があります。
  - JAXB の Java オブジェクトの作成については、「5.7.5　JAXB のソースを修正するツール」を参照してくだい。
  - メディアタイプを“header”や“produces”で明示的に指定します。

```
@Controller
public class RestUriController {

    // パス変数と JSON/XML の組合せ
    @RequestMapping(value = "/rest/owners/{userCd}",
            method=RequestMethod.POST,
            headers={"Content-Type=application/json", "Accept=application/xml"})
//consumes=MediaType.APPLICATION_JSON_VALUE, produces=MediaType.APPLICATION_XML_VALUE)")
    @ResponseBody
    public Person createUserInfo(@PathVariable String userCd,
            @RequestBody UserInfoDto userInfo) throws DataNotFoundException {
        // 省略
    }
}
```

メディアタイプは、MediaType クラスの定数を利用すると間違いがありません。

5.4.4.2. クライアント側（jQuery）

- 既存の ajax(…)メソッドを使用します。
- パス変数を「tokens」で指定します。
- 送信データを JSON で組み立てます。
  - JSON 形式を送信するため、「contentType」でメディアタイプを指定します。
- 受信データは XML であるため、「dataType」でメディアタイプを指定します。

```
<script type="text/javascript" src="${appUrl}/js/lib/jquery-uritemplate.1.0.js"></script>
<script type="text/javascript">
$(document).ready(function(){

    // パス変数と JSON/XML データの組合せ
    $('#uriSample5').click(function(){
        $.ajax({
            type: "POST",
            url: "${appUrl}/rest/owners/{userCd}.html",
            tokens: {userCd:'admin'},
            // 送信データ
            contentType: "application/json;charset=UTF-8",
            data: '{"name": "管理者", "phoneNumber": "0123-456-789"}',

            // 受信時のメディアタイプ
            dataType: "xml",
            success:function(data){
                //alert($(data).text());
                alert($(data).find('Person').attr('name'));
            },
            error:function(XMLHttpRequest, textStatus, errorThrown){

                // エラーのタイプ（timeout, error, notmodified, parsererror）
                alert("textStatus=" + textStatus);

                // エラーメッセージ
                alert("errorThrown=" + errorThrown);
            }
        });

    });

});
</script>
<ol>
    <li><span id="uriSample5">URI Template で URI を組み立てる(XML/JSON データを送受信する)</span></li>
</ol>
```

プラグインを読み込みます。

送信する JSON データを組み立てます。

5.4.4.3. クライアント側（RestTemplate）

【RestTemplate#exchange(…)メソッドを利用する場合】

- 送信するデータが階層構造を持つため、JavaBean の形式としてリクエストデータを作成します。
  - HttpHeaders クラスで送信するデータのメディアタイプを指定します。
  - HTTP リクエスト本体の HttpEntity クラスで作成します。
- 受信するデータは XML なので、JAXB 形式のオブジェクトを RestTemplate の戻り値の Generics のタイプとします。

```
@Component
public class RestTemplateClient2 {

    @Resource
    private RestTemplate restTemplate;
```

```
    // パス変数と XML/JSON の組合せ(exchange メソッドの利用)
    public void uriSample5() {

        try {

            // リクエストデータを JavaBean クラスで作成する
            UserInfoDto requestData = new UserInfoDto();
            requestData.setName("管理者");
            requestData.setPhoneNumber("0123-456-789");

            // リクエスト時の HTTP ヘッダーを設定（ContentType を指定）
            HttpHeaders rqeustHeaders = new HttpHeaders();
            rqeustHeaders.setContentType(MediaType.APPLICATION_JSON);

            // リクエスト時の HTTP データの作成（HTTP ヘッダーと Body(入力データ)）を指定。
            HttpEntity<UserInfoDto> requestEntity =
                    new HttpEntity<UserInfoDto>(requestData, rqeustHeaders);

            // UriComponents/UriComponentsBuilder を使用する
            UriComponents uriComponents = UriComponentsBuilder
                    .fromUriString("http://localhost:8080/spring-mvc-3.1/")
                    .path("/rest/owners/{userCd}.html")
                    .build()
                    .normalize();

            URI uri3 = uriComponents.expand("admin").encode().toUri();

            // RestTemplate の汎用的な exchange を使用する
            ResponseEntity<Person> responseEntity = restTemplate.exchange(
                    uri3, HttpMethod.POST, requestEntity, Person.class);

            Person responsedata = responseEntity.getBody();
            System.out.println(responsedata);

        } catch(Exception e) {
            e.printStackTrace();
        }
    }

}
```

リクエストデータ(JSON)の作成。

パス変数を含む URI の組み立て。

リクエストの実行とレスポンスデータの取得。

【RestTemplate#postForObject(…)メソッドを利用する場合】

- HTTP メソッドと送信データと受信データの組合せがマッチするため、「RestTemplate#postForObject(….)」を利用します。
- 送信するデータは、RestTemplate に直接渡します。
  - ➢　クラス型によって、HttpMessageConverter が判断してくれます。
- 受信するデータも、ResttTemplate から直接取得します。
- HTTP メソッド固有の RestTemplate のメソッドを利用できる場合、送受信するレスポンス、リクエストのクラス HttpEntity や ResponseEntity を介さなくても簡単に処理できることがわかります。

```
@Component
public class RestTemplateClient2 {

    @Resource
    private RestTemplate restTemplate;

    // パス変数と XML/JSON の組合せ(postForObject メソッドの利用)
    public void uriSample5_2() {

        try {

            // リクエストデータを JavaBean クラスで作成する
            UserInfoDto requestData = new UserInfoDto();
            requestData.setName("管理者");
            requestData.setPhoneNumber("0123-456-789");

            // UriComponents/UriComponentsBuilder を使用する
            UriComponents uriComponents = UriComponentsBuilder
                    .fromUriString("http://localhost:8080/spring-mvc-3.1/")
                    .path("/rest/owners/{userCd}.html")
                    .build()
                    .normalize();

            URI uri3 = uriComponents.expand("admin").encode().toUri();

            // RestTemplate の POST 用の psotForObject を使用する
            Person responseData = restTemplate.postForObject(
                    uri3, requestData, Person.class);
            System.out.println(responseData);

        } catch(Exception e) {
            e.printStackTrace();
        }
    }

}
```

リクエストデータの作成。

パス変数を含む URI の組み立て。

リクエストの実行とレスポンスデータの取得。

5.4.5. パス変数の値を正規表現でフィルタする

- Controller 側でパス変数の書式を正規表現で制限することができます。
  - 書式は、「{変数名:<正規表現>}」です。
- この表現に合わない URI にアクセスした場合、「404(Not Found)」が返され、「400(Bad Request)」ではありません。

```
@Controller
public class RestUriController {

    // パス変数のフォーマット指定
    @RequestMapping(value = "/rest/range/{start:[¥¥d]+}-{end:[0-9]+}",
            method=RequestMethod.GET, headers="Accept=application/json")
    @ResponseBody
    public List<InformationDto> findUserInfoByDateRange(
            @PathVariable String start, @PathVariable String end) throws DataNotFoundException {


        //省略
    }
}
```

5.5. REST サービスによるエラー処理

REST では、「5.2.2　REST サービスにおける HTTP メソッドの役割」で説明したように、エラーなどの状態は HTTP のステータスコードで表現すべきです。

表 5.11　HTTP ステータスコードごとの返し方

No.	HTTP ステータスコード	処理方法
1	1xx（Informational）	・使用しません。
2	2xx（Successful）	・Controller で制御します。
3	3xx（Redirection）	・ログインする際のリダイレクト処理などで制御します。
4	4xx（Client Error）	・リクエストの値が不正などのクライアントに問題がある場合に使用します。 ・Controller で主に制御します。
5	5xx（Server Error）	・サーバに問題がある場合に利用します。 ・基本的には APP では制御しないで、Tomcat などのサーバ側や SpringMVC のフレームワークで自動的に判定して返します。

5.5.1. サーバ側

5.5.1.1. エラー時の特定の HTTP ステータスコードの返却

- 特定の HTTP ステータスを返却するには、メソッドに「@ResponseStatus」を設定します。
  - 引数には、ステータスコードの列挙型「HttpStatus」を指定します。
  - @ResponseStatus を付与しない場合は、自動的にステータス「200（OK）」になりますが、エラー時と区別するために、明示的にしていた方がよいです。
- リクエストを処理するメソッドは、@ResponseStatus を付与しているため「ステータス 200（OK）」しか返せなません。そのため、パラメータ値が不正や処理中に発生する整合性の不正時のエラーの処理は、例外をスローし、「@ExceptionHandler」が付与されたメソッドで対処します。
- @ExceptionHandler が付与されたメソッドは、@RequestMapping が付与されたリクエスト処理用のメソッド同じような引数と戻り値を取ることができます。REST の場合は、クライアント側ではエラーは基本的にステータスで判断するため、戻り値として必要なのは、“HTTP ステータスコード”と“エラーの詳細”の 2 つです。
  - @ExceptionHandler を付与したメソッドが取りえる引数、戻り値については、「11.1　Controller での例外ハンドリング「@ExceptionHandler」」を参照してください。
  - 戻り値は、文字列以外でも JSON などで渡せますが、REST はシンプルであるべきなので、文字列でも十分だと考えられます。

```
@Controller
public class RestServiceController {

    @Resource
    private IUserInfoDao userInfoDao;

    // ユーザ情報を取得する
    @RequestMapping(value="/service/users/{userCd}"
            ,method=RequestMethod.GET, produces=MediaType.APPLICATION_JSON_VALUE)
    @ResponseStatus(value=HttpStatus.OK)
    @ResponseBody
    public UserInfoDto loadUserByUserCd(@PathVariable String userCd) throws DataNotFoundException {

        SqlBuilder sql = new SqlBuilder()
            .append("userCd = :userCd", userCd);

        UserInfoDto userInfo = userInfoDao.loadFirst(sql);
        if(userInfo == null) {
            // データが見つからない場合
            throw new DataNotFoundException(UserInfoDto.TABLE_NAME, "userCd", userCd);
        }

        return userInfo;

    }

    // データが見つからない場合
    @ExceptionHandler(DataNotFoundException.class)
    @ResponseStatus(value=HttpStatus.NOT_FOUND)
    @ResponseBody
    public String handlerDataNotException(DataNotFoundException e) {

        // レスポンスのデータを設定
        // エラーの内容の詳細を返す
        String reason = String.format("%s=%s is not found.",
                e.getColumnName(), e.getColumnValue());
        return reason;
    }

}
```

・返却する HTTP ステータスを指定します。
・何も指定しない場合は、ステータス 200 になります。

リクエストを処理するメソッドで、ステータス 200(OK)以外を返したい場合、例外をスローし@ExceptionHandler が付与されたメソッドで処理します。

ハンドル（キャッチ）する例外を引数で指定します。

返すステータスコードを指定しま

クライアント側に渡すエラーの詳細を設定します。

5.5.1.2. 様々なエラーの処理方法

- @ExceptionHandler には、1 つだけでなく複数の例外が指定できる。
  - 同じステータスコードを返すような場合は、まとめてもよいかもしれません。
- 抽象度の高い例外、例えば「Exception.class」を指定した場合は、優先度が低くなります。
  - 予期しない例外をまとめて処理する際に、指定します。
  - HttpEntity をメソッドの戻り値に設定すると、ステータスコードも動的に変更することができます。
- @PathVariable を設定した引数が Integer に対して、“aaa”などの不正な値をクライアントから送信された場合、Spring MVC では、「TypeMismatchException」がスローされ“400（Bad Request)”が自動的に返されます。
  - これらの自動的にスローされる例外は、Controller 側で@ExceptionHandler にてキャッチできる場合もあります。
  - Spring MVC で発生する例外を下記の「表 5.12」に示します。

表 5.12 Spring MVC で発生する例外と HTTP ステータスコード

No.	例外クラス	HTTP ステータスコードと発生ケース
1	ConversionNotSupportedException	500 (Internal Server Error)
2	HttpMediaTypeNotAcceptableException	406 (Not Acceptable) ・クライアントへの応答するメディアタイプ(Accept)が要求されたものと一致しない場合。
3	HttpMediaTypeNotSupportedException	415 (Unsupported Media Type) ・クライアントへのから送信されたメディアタイプ(Content-Type)がサーバ側と一致しない場合。
4	HttpMessageNotReadableException	400 (Bad Request) ・クライアントから送信された JSON などをサーバ側で、HttpMessageConverter 経由で Java オブジェクトにマップする際に失敗する場合。
5	HttpMessageNotWritableException	500 (Internal Server Error) ・サーバからクライアントに JSON などを、レスポンスを返す場合、HttpMessageConverter 経由で Java オブジェクトから JSON に変換する際に失敗する場合。
6	HttpRequestMethodNotSupportedException	405 (Method Not Allowed)
7	MissingServletRequestParameterException	400 (Bad Request)

8	NoSuchRequestHandlingMethodException	404 (Not Found) ・要求されたリクエストに対して、サーバ側で処理可能な HTTP メソッド（@RequestMappinng(method=<HTTP メソッド>)）が付与された Controller が見つからない場合。
9	TypeMismatchException	400 (Bad Request) ・バインドエラーが発生した場合。

【様々な例外を処理する REST 用 Controller】

```
@Controller
public class RestServiceController {

    @Resource
    private IInformationDao informationDao;

    // 年を情報を検索する
    @RequestMapping(value="/service/infos/{year}"
            ,method=RequestMethod.GET
            ,produces=MediaType.APPLICATION_JSON_VALUE)
    @ResponseStatus(value=HttpStatus.OK)
    public List<InformationDto> searchBlogEntroies(@PathVariable Integer year) {

        if(year < 0) {
            throw new InvalidParameterException("year < 0");
        }
        return informationDao.search(new SqlBuilder());

    }

    // パラメータが不正な場合
    @ExceptionHandler({TypeMismatchException.class, InvalidParameterException.class})
    @ResponseStatus(value=HttpStatus.BAD_REQUEST)
    @ResponseBody
    public String handlerInvalidParameter(Exception ex) {

        String reason = null;
        if(ex.getClass().isAssignableFrom(TypeMismatchException.class)) {
            reason = ex.getMessage();

        } else if(ex.getClass().isAssignableFrom(InvalidParameterException.class)) {
            reason = ex.getMessage();

        } else {
            //TODO: その他
            reason = ex.getMessage();
        }

        return reason;
    }

    // その他の例外
    @ExceptionHandler(Exception.class)
    @ResponseBody
    public ResponseEntity<String> handlerException(Exception ex) {

        System.out.println("::handelrException :  : " + ex.getMessage());
```

@ExceptionHandler は、複数の例外を記述して処理することが可能。

抽象度の高い、その他の例外を指定します。

```
        // エラーメッセージとステータスの組み立て
        String message = ex.getMessage();
        ResponseEntity<String> response =
                new ResponseEntity<String>(message, HttpStatus.INTERNAL_SERVER_ERROR);
        return response;
    }
}
```

ResponseEntity を使用することで、ヘッダー情報やステータスを動的に設定できる。

5.5.1.3.　REST サービスにおける Controlelr 側の例外処理の考え方

- Spring MVC では、例外を処理する“ExceptionResolver”があり、このクラスにて Controller で発生する例外をまとめて処理します（「11.2　システム全体での例外ハンドリング」を参照)。
- しかし、REST 用の Controller と通常の画面（JSP）を処理する Controller が同じコンテナで混在していると、問題が出てきます。
  - REST の例外処理は最後には、適したステータスコードを返すべきで、画面の例外処理は専用のエラー画面を表示します。
  - この場合､例外時の処理が提供する機能によって､異なるため ExceptionResolver では適しません。
- REST サービスしか提供しない場合は、ExceptionResolver で処理してもかまいませんが、画面と併用する場合、REST 用の抽象クラスを用意し、それを継承して Controller を作成します。

【抽象クラスを利用した共通の例外処理】

```
// REST 用コントローラの抽象クラス
public abstract class AbstractRestServiceController {

      // データが見つからない場合
     @ExceptionHandler(DataNotFoundException.class)
     @ResponseStatus(value=HttpStatus.NOT_FOUND)
     @ResponseBody
     public String handlerDataNotException(DataNotFoundException e) {
          // 省略
     }

     // パラメータが不正な場合
     @ExceptionHandler({TypeMismatchException.class, InvalidParameterException.class})
     @ResponseStatus(value=HttpStatus.BAD_REQUEST)
     @ResponseBody
     public String handlerInvalidParameter(Exception ex) {
          // 省略
     }
     // その他の例外
     @ExceptionHandler(Exception.class)
     @ResponseBody
     public ResponseEntity<String> handlerException(Exception ex) {
          // 省略
     }
}

// REST 用コントローラの実装クラス
@Controller
public class RestServiceController extends AbstractRestServiceController{
     // 省略
}
```

5.5.2. クライアント側

5.5.2.1. jQuery を使用する場合

- ajax(...)を利用している場合、サーバからステータス 200(OK)以外が返された場合、error(...)メソッドでエラー処理を行います。
- ステータスコード「3xx」が返される場合は、環境によってはステータス 200(OK)として判断し、success メソッドが呼ばれ正常系として処理されることがあるので注意が必要。
  - 302(リダイレクト)の場合は、リダイレクト先にリクエストを送信して、正常終了し 200 が返ってくる。

```
<script type="text/javascript" src="${appUrl}/js/lib/jquery-uritemplate.1.0.js"></script>
<script type="text/javascript">
$(document).ready(function(){

    // データが見つからない場合
    $('#errorSample2').click(function(){
        $.ajax({
            type: "GET",
            url : "${appUrl}/service/users/{userCd}.html",
            tokens: {userCd:'hogehoge'},
            // 受信時のメディアタイプ
            dataType: "json",
            success:function(data, textStatus, xhr){
                // data = 受信データ
                // textStatus = 例) OK
                // xhr = XMLHttpRequest
                alert(data);
            },
            error:function(xhr, textStatus, errorThrown){
                // xhr = XMLHttpRequest
                // textStatus = {timeout, error, notmodified, abort, parsererror}
                // erorrThown = 例) Internal SeverError (ステータスコードに対するメッセージ名)

                // 例外時に設定したレスポンスデータ
                alert(xhr.responseText)

                if(xhr.status == 400) {
                    // BadRequest
                    alert("詳細メッセージ=" + xhr.responseText);
                }
            }
        });
    });

});
</script>
<ol>
    <li><span id="errorSample2">例外処理（データが見つからない場合）</span></li>
</ol>
```

エラー発生時（ステータス 200(OK)以外）は、error メソッドが呼ばれます。

Controller 側で設定した詳細情報（レスポンスボディ）を取得します。

ステータスコードごとに処理したい場合。

5.5.2.2.　RestTemplate を利用する場合

- ステータス 200(OK)以外がサーバから返された場合、例外「HttpClientException」が発生します。
  - ステータスの詳細を取得する場合は、そのサブクラス「**HttpStatusCodeException**」を利用します。
- サーバに接続できないような I/O エラーが発生した場合は、「**ResourceAccessException**」が発生します。
- 例外体系については、「図 5.3　RestTemplate を使用した場合の例外体系」を参照してください。
  - ステータスコード 200(OK)以外が返される場合や、通信中に例外が発生すると実際には「**ResponseErrorHandler**」で処理されます。これは、「RestTemplate#setErrorHandler(…)」で他のカスタマイズできます。
  - 初期値は、「**DefaultResponseErrorHandler**」が設定されています。

【RestTemplate での例外処理】

```
@Component
public class RestTemplateClient3 {

    @Resource
    private RestTemplate restTemplate;

    public void errorSample1() {

        try {

            UriComponents uriComponents = UriComponentsBuilder
                    .fromUriString("http://localhost:8080/spring-mvc-3.1/")
                    .path("/service/users/{userCd}.html")
                    .build()
                    .normalize();
            URI uri3 = uriComponents.expand("admin").encode().toUri();
            UserInfoDto responseData = restTemplate.getForObject(
                    uri3,
                    UserInfoDto.class);
            System.out.println(responseData);

        } catch(HttpStatusCodeException e) {
            // HTTP ステータス 200 以外の処理

            // ステータスの取得
            HttpStatus status = e.getStatusCode();
            String statusText = e.getStatusText();

            System.out.printf("statusCode=%s, statusName=%s¥n",
                    status.value(), statusText);

            // エラー詳細（レスポンスの取得）
            String errorDetail = e.getResponseBodyAsString();
            System.out.printf("エラー詳細=%s¥n", errorDetail);

        } catch(ResourceAccessException e) {
            // サーバ接続エラーの場合
            System.out.println("サーバ接続エラー");
            e.printStackTrace();

        } catch(RestClientException e) {
```

例外「HttpStatusCodeException」をキャッチして、エラーの内容に従い判定を行います。

HTTP ステータスコードの取得ができます。

Controller 側で設定した詳細情報（レスポンスボディ）を取得します。

サーバに接続できないような場合の例外。

```
            e.printStackTrace();

        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

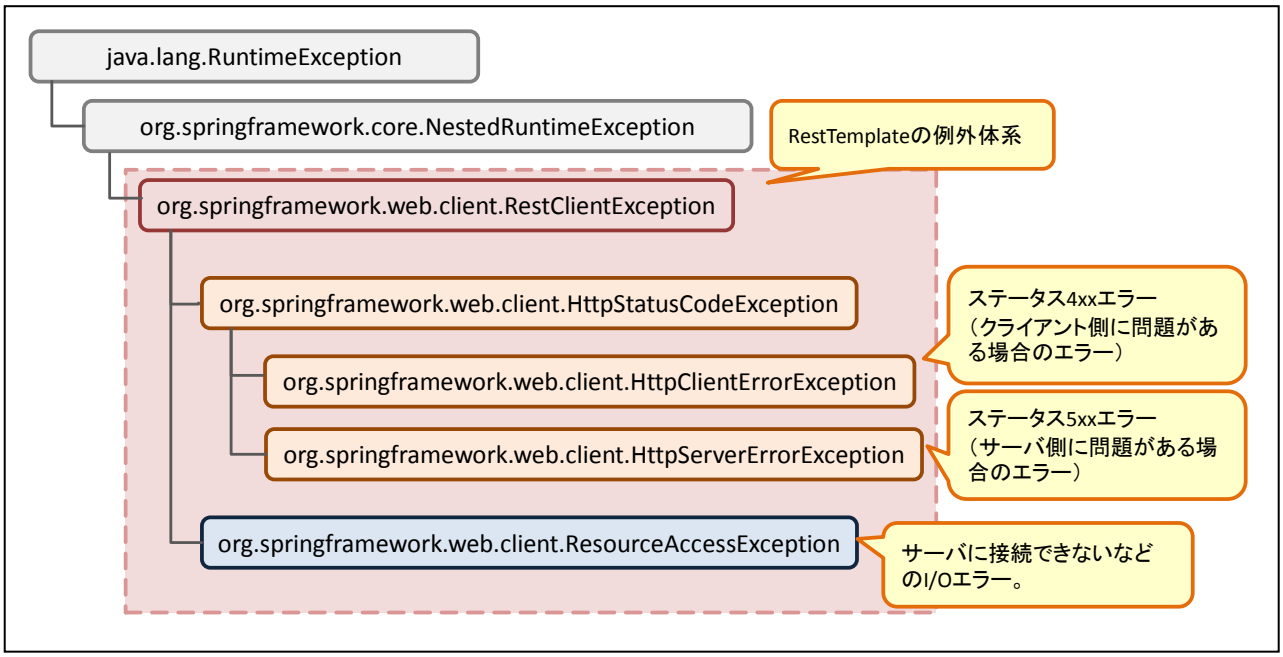


図 5.3　RestTemplate を使用した場合の例外体系

5.6. クライアント側「RestTemplate による REST サービスへのアクセス」

Spring MVC では、RESTful サービスを実現するためのクライアントとして「RestTemplate」が用意されています。これは、Spring の他のテンプレートである“JdbcTemplate”や“JmsTemplate”と似たようなコンセプトです。

- Java 側で REST サービスにアクセスするクライアントを作成する際には、ライブラリ「Commons HttpClient」を使用してわざわざ処理していたのを、RestTemplate を使用すると容易に処理できます。
  - JavaScript の「jQuery」における「ajax(...)」メソッドのような役割をします。
- 特に、JSON や XML を送受信する際に JAVA オブジェクトに自動的に変換してくれます。
  - これは、サーバ側の Controller で利用していた「HttpMessageConverter」をクライアント側でも利用することにより実現します。

5.6.1. RestTemplate を Spring Bean として定義する

RestTemplate は、インスタンスをそのまま作成しても利用できますが、環境によってはカスタマイズする必要があります。

- プロキシ環境や認証環境の場合、HttpClient をカスタマイズします。詳細は、「5.6.1.1　「Commons HttpClient」を経由してリソースにアクセスする」を参照してください。
- 送受信するデータを変換する方式をカスタマイズしたい場合は、HttpMessageConverter を変更します。通常は変更する必要はありません。詳細は、「5.3.1　データの変換「HttpMessageConverter」を参照してください。
- エラー発生時のレスポンスのステータスなどをカスタマイズできます。通常は変更する必要はありません。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
・・・(省略)・・・

    <bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
        <!-- HttpClient のカスタマイズ
        <constructor-arg>
            <bean class="org.springframework.http.client.HttpComponentsClientHttpRequestFactory">
                <property name="httpClient">
                    <bean id="restHttpClient" class="org.apache.commons.httpclient.HttpClient">
                    </bean>
                </property>
            </bean>
        </constructor-arg>
         -->

        <!-- HttpMessageConverter のカスタマイズ
        <property name="messageConverters">
            <list>
                <bean class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"/>
            </list>
        </property>
        -->
```

```
        <!-- エラーハンドリングのカスタマイズ
        <property name="errorHandler">
            <bean class="org.springframework.web.client.DefaultResponseErrorHandler"/>
        </property>
         -->
    </bean>
・・・（省略）・・・
</beans>
```

5.6.1.1.　「Commons HttpClient」を経由してリソースにアクセスする

プロキシ環境内からの外部のサービスへのアクセスする場合、「Commons HttpClient」などを利用します。

- RestTemplate のコンストラクタの引数に「org.springframework.http.client.ClientHttpRequestFactory」の実装クラスのインスタンスを渡します（表 5.13　ClientHttpRequestFactory の実装クラス）。
- Spring 3.1 では、「HttpComponentsClientHttpRequestFactory」を利用します（図 5.4　RestTemplate に HttpClient をインジェクションする例）。
  - プロキシ環境など利用する環境に設定した「HttpClient」のインスタンスをインジェクションする必要があるので注意してください。
  - プロキシ環境などネットワークはクライアントの環境に高く依存し、それに合わせ HttpClient も柔軟に設定できるため、バリエーションが多すぎてインタフェースが対応しきれないため、**Spring 側で HttpClient を作成する FactoryBean は用意されていません**。そのため、**自分で HttpClient を生成する FactoryBean を実装してください**。
  - よくある HttpClient のパターンは、Spring Modules として公開されているため、次の参考にしてください。

    「http://www.springbyexample.org/examples/spring-by-example-utils-module.html」

表 5.13　ClientHttpRequestFactory の実装クラス

No.	クラス名	説明
1	SimpleClientHttpRequestFactory	・RestTemplate のデフォルトのクラスです。 ・中身は Java 標準の「java.net.HttpURLConnection」を利用して実装されています。 ・Java 標準では、**HTTP の PATCH メソッドに非対応**であるため、その場合は、Commons HttpClient の実装を利用します。
2	CommonsClientHttpRequestFactory	・ライブラリ「Commons HttpClient」を利用して実装されています。 ・ただし、**Spring 3.1 では非推奨**となっているので「HttpComponentsClientHttpRequestFactory」を使用します。

3	HttpComponentsClientHttpRequestFactory	・ライブラリ「Commons HttpClient」を利用して実装されています。 ・Spring 3.1 から追加されたクラスです。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    ・・・(省略)・・・
    <bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
        <constructor-arg>
            <bean class="org.springframework.http.client.HttpComponentsClientHttpRequestFactory">
                <property name="httpClient">
                    <!-- TODO:認証設定した HttpClient のインスタンスをインジェクションする -->
                    <bean id="restHttpClient" class="org.apache.commons.httpclient.HttpClient">
                    </bean>
                </property>
            </bean>
        </constructor-arg>
    </bean>
    ・・・(省略)・・・
</beans>
```

図 5.4　RestTemplate に HttpClient をインジェクションする例

5.6.2. RestTemplate のメソッド(TODO)

RestTemplate には、HTTP メソッドに対応したものが用意されています（表 5.14　ResteTemplate のメソッド)。

- 単純なデータのやり取りでは、それぞれに対応したメソッドを使用するればよいですが、細かなカスタマイズをしたい場合は、汎用的な「exchange」を利用します。
- 「exchange」メソッドを利用すればどのようなデータでも送受信できますが、できるだけ固有のメソッドを利用するようにしてください。
  - 利用するサービスとインタフェースが一致しないため既存のメソッドが利用できないようなときは、それは REST サービスの設計がまずいと言えるため、設計をやり直した方がよいと思います。
    - ✧　詳細は、「5.2.2　REST サービスにおける HTTP メソッドの役割」を参照してください。
  - 例えば、ResetTemplate#delete(...)を使用する場合、レスポンスとして JSON や XML を返す場合です。実行結果は、レスポンスの HTTP ステータスコードで判断すべきです。

表 5.14　ResteTemplate のメソッド

No.	HTTP メソッド	RestTemplate のメソッド	
1	GET	・Object getForObject(...)：レスポンスの ボディ部分のみを取得します。 ・ResponseEntity getForEntity(...)：レスポンスのエンティティを取得して細かな判定を行いたい場合に利用します。	
2	DLETE	void delete(...)	

3	PUT	void put(…)	
4	POST	・Object postForObject(…) : レスポンスのボディ部分のみを取得します。 ・ResponseEntity postForEntity(…) : レスポンスのエンティティを取得して細かな判定をしたい場合に利用します。 ・URI postForLocation(…) : HTTP のヘッダーに Location を付加してリダイレクする場合に利用します。	
5	PATCH	ResponseEntity exchange(..) : 汎用的なメソッドを利用します。 ・ただし、Spring 3.2 から PATCH メソッドに対応	
6	HEAD	・HttpHEaders headForHeaders(…)	
7	OPTIONS	・Set<HttpMedhod> optionForAllow(…)	
8	—	・ResponseEntity exchange(..) : 汎用的なメソッドです。	

5.6.3. URI の組み立て（UriTemplate、UriComponents／UriComponentsBuilder）

REST サービスにアクセスする URI を組み立てる際に、変数が埋め込まれていたりすると、それだけでソースコードが複雑になります。それを補助するのが「UriComponentsBuilder」です。**Spring 3.1 から追加**になった機能です。

- 「UriComponentsBuilder」を利用して URI を組み立てて、インスタンス「UriComponents」を取得します。
- 「UriComponets」は、「java.net.URI」を拡張したようなもので、最終的に RestTemplate に渡す形式の“String 型の URL”や“java.net.URI”を生成します。
- クエリストリングで値に URL エンコードが必要な場合にも利用します。

5.6.3.1. UriTemplate

- UriTemplate は、**RestTtemplate に渡す URI を組み立てる場合は、はっきり言って使うメリットはありません**。
  - なぜならば、RestTemplate に、UriTemplate と同程度に機能があるからです。
  - UriTemplate は、RestTemplate 以外での他の機能で「UIR Template Pattern」を利用する際に使います。

【URI を文字列から取得する場合】

- URI のパス変数は、UriComponets#expand(…)メソッドで実際の値に変換します。
  - パス変数は、可変長引数（配列）で渡した場合は、URI に**記述された順番で置換**します。

```
UriTemplate uriTemplate = new UriTemplate("http://example.com/hotels/{hotel}/bookings/{booking}");
URI uri2 = uriTemplate.expand("42", "21");
```

【パス変数をマップで指定する場合】

- パス変数の値を Map で組み立てると、expand()メソッドに渡す際に順番は関係ありません。

```
UriTemplate uriTemplate = new UriTemplate("http://example.com/hotels/{hotel}/bookings/{booking}");

Map<String, Object> pathVars = new HashMap<String, Object>();
pathVars.put("hotel", 42);
pathVars.put("booking", 21);

URI uri2 = uriTemplate.expand(pathVars);
```

5.6.3.2. UriComponents/UriComponetsBuilder

- UriComponents は、immutable(インスタンス生成後変更不可)であるため、UriComponentsBuilder でインスタンスを組み立てます。
- URI のパス変数は、UriComponets#expand(…)メソッドで実際の値に変換します。
  - パス変数は、可変長引数（配列）で渡した場合は、URI に記述された順番で置換します。

【URI を文字列から取得する場合】

- URI を文字列から単に組み立てる場合、UriComponentsBuilder#fromUriString()を使用します。
- パス変数の値は、UriComponents#expand(…)で取得できます。
  - 可変長配列（配列）で渡す際には引数に指定した順に値が変換されます。
- パス変数は英字数字だけでなく記号や日本語も設定可能なので、URI エンコードをするための UriComponents#encode()を必ず使用します。

```
// URI を文字列から取得する場合
UriComponents uriComponents = UriComponentsBuilder
        .fromUriString("http://example.com/hotels/{hotel}/bookings/{booking}").build();

URI uri = uriComponents.expand("42", "21").encode().toUri();
```

【パス変数をマップで指定する場合】

- パス変数の値を Map で組み立てると、expand()メソッドに渡す際に順番は関係ありません。

```
UriComponents uriComponents =
        UriComponentsBuilder.fromUriString("http://example.com/hotels/{hotel}/bookings/{booking}").build();

Map<String, Object> pathVars = new HashMap<String, Object>();
pathVars.put("hotel", 42);
pathVars.put("booking", 21);

URI uri = uriComponents.expand(pathVars).encode().toUri();
```

【URI を部分ごとに組み立てる場合】

- プロトコル名、ホスト名、その他のパスと分けて組みたることもできます。
  - ホスト名やその他のパスが動的に変わる際など便利です。

```
public void sampleUriComponets3() {
UriComponents uriComponents =
        UriComponentsBuilder.newInstance()
            .scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}").build()
            .expand("42", "21")
            .encode();

URI uri = uriComponents.toUri();
```

【URI の正規化】

- UriComponets#normalize()を使用すると、パス中に含む相対パス「./」「../」を解釈して、絶対パスに変換します。
  - URI の区切り”/”（スラッシュ）が連続するような場合は対象外で変換されません。

```
UriComponents uriComponents =
        UriComponentsBuilder.newInstance()
            .path("http://example.com")
                .path("/hotels/{hotel}/../")
                    .path("./bookings/{booking}").build()
            .normalize()
            .expand("42", "21")
            .encode();

URI uri = uriComponents.toUri();

// normalize 後の URI
// http://example.com/hotels/bookings/42
```

5.7. JAXB の Java ソースの自動生成

JAXB による XML マッピングを行うには、JAXB のアノテーションの使用方法の知識が必要になります。また、場合によっては、XML Schema ファイルも必要になります。

XML の定義が変わるたびに、これらのファイルを修正していては、定義間違いが発生します。さらに、XML の構造が複雑になると、作業量が膨大になります。

そこで、JAXB のソースを自動生成する方法を説明します（図 5.5　JAXB のソースの自動生成の流れ）。

- JAXB のソースは、JDK 付属の“xjc”コマンドで、XML Schema から生成可能です。
  - しかし、XML Schema の仕様は膨大で、調べたりするのが大変です。
- XML のスキーマ言語の 1 つである RELAX NG（リラクシング）は、XML Schema よりも仕様が単純で簡単に覚えることができます。
  - XML スキーマ変換ツール“Trang”を利用し、「RELAX NG⇒XML Schema」に変換します。
  - RELAX NG は XML Schema と比較して「仕様が単純＝機能が少ない」ですが、JAXB のソースを生成し利用するには、機能としては十分です。
- そこで、RELAX NG を XML Schema を介して、JAXB の Java ソースを生成します。
  - “xjc”コマンド自体、RELAX NG から直接ソースを生成できますが、これは実験的な機能であり、XML Schema のデータ型の解釈ができないなど機能不足です。

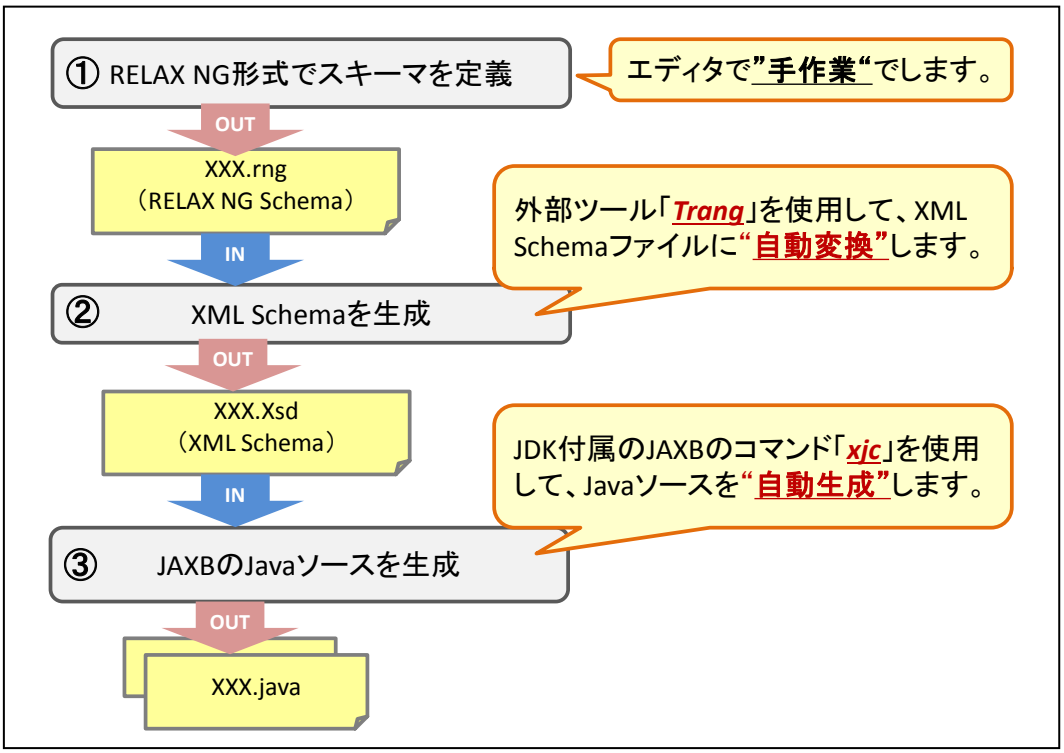


図 5.5　JAXB のソースの自動生成の流れ

5.7.1. RELAX NG ファイルの作成

RELAX NG は 2000 年代前半に出てきて、改良はされていないので新し情報はあまりありません。RELAX NG の Schema が JAXB で生成されるソース、XML にどのような影響を与えるかについては、「5.8　RELAX NG（リラクシング）について」を参照してください。

【参考 URL】

- RELAX NG 仕様書
  http://www.asahi-net.or.jp/~eb2m-mrt/relaxngjis/jaspec-20011203.html
- RLAX NG 入門（チュートリアル）
  http://www.kohsuke.org/relaxng/tutorial.ja.html
  ※ブラウザの文字エンコーディングを「Shift_JIS」に設定する必要があります。

【RELAX NG ファイルのサンプル】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
  xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

    <define name="SampleJaxb2">
        <element name="SampleJaxb2">
            <zeroOrMore>
                <ref name="Result"/>
            </zeroOrMore>
            <optional>
                <element name="processingTime">
                    <data type="double"/>
                </element>
            </optional>
        </element>
    </define>

    <define name="Result">
        <element name="Result">
            <attribute name="id"><text/></attribute>
            <element name="name"><text/></element>
            <optional>
                <element name="age"><data type="int"/></element>
                <element name="tel"><data type="string"/></element>
            </optional>
        </element>
    </define>

</grammar>
```

5.7.2. Trang による XML Schema ファイルの変換

5.7.2.1. Trang のインストール

(1)　Trang の媒体を下記の URL から、最新版の「trang-20091111.zip」をダウンロードします。

http://www.thaiopensource.com/relaxng/trang.html

(2)　ダウンロードしたファイルを任意の場所に展開し配置ます。ここでは、「c:¥」直下に配置します。

```
c:¥trang-20091111
 ├ trang.jar　・・・Trang の本体の jar ファイル。
 ・・・
```

(3)　バッチファイル「trang.bat」を作成し、パスが通った場所に配置します。

```
@echo off
REM trang の実行用バッチファイル

%JAVA_HOME%¥bin¥java -cp .; -jar C:¥trang-20091111¥trang.jar %1 %2 %3 %4 %5 %6 %7 %8 %9
```

5.7.2.2. XML Shema ファイルの変換

RELAX NG ファイル⇒XML Schema ファイルに変換するには、次のようにオプションを指定します。

```
> trang -I rng -O xsd <Relax NG スキーマファイル> <XMLSchema ファイル名>

# 例
> trang -I rng -O xsd sample.rng sample.xsd
```

5.7.3. xjc コマンドを使用した JAXB のソース生成

【参考 URL】

http://www.jagat.or.jp/sgml/xml/xmltools/trang-20030619/trang-manual-ja.html

```
・パッケージを指定しない
> %JAVA_HOME%¥bin¥xjc <スキーマファイル>

・パッケージを指定する
> %JAVA_HOME%¥bin¥xjc <スキーマファイル> -p <パッケージ名>

例）パッケージ名を指定する場合（com.sample.jaxb パッケージに作成される）
> %JAVA_HOME%¥bin¥xjc sample.xsd -p com.sample.jaxb
```

5.7.3.1.　ポイント「java.io.Serializable を実装したい」

- xjc コマンドで生成した JAXB のソースファイルに対して、「java.io.Serializable」を実装したい場合、XML Schema フィルの定義を次のように変更します。
  (1)　名前空間「xmlns:jxb="http://java.sun.com/xml/ns/jaxb"」を追加。
  (2)　名前空間「jxb:version="2.1"」を追加
  (3)　要素<jxb:serializable uid="1">の定義を追加する。
- RELAX NG スキーマファイルを修正したら、毎回 XMLSchema ファイルを編集するのが面倒なので、ツールにより Serializable の定義を自動的に追加する方法をお勧めします。詳細は、「5.7.5.2　XML Schema に Serializable の定義を追加する」を参照してください。

【XML Schema ファイル】

- シリアライズ用の UID が固定値しか設定できないので、後からユニークな値に変更することをお勧めします。
  - 「5.7.5　JAXB のソースを修正するツール」に示す、ツールを利用して一括変換する方法もあります。

```
<xs:schema version="1.0"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
          jxb:version="2.1">

 <!--- ▼Serializable の実装を追加▼ -->
 <xs:annotation>
   <xs:appinfo>
     <jxb:globalBindings>
       <jxb:serializable uid="1"/>
     </jxb:globalBindings>
   </xs:appinfo>
 </xs:annotation>
 <!--- ▲Serializable の実装を追加▲ -->

 <xs:element name="persons">
 ・・・省略
 <xs:element>
<xs:schema>
```

名前空間を追加します。

・Serializable の定義を追加します。
・固定値として「1」を設定します。

5.7.4. RELAX NG から JAXB ソースの生成

JAXB のソースを生成するのに、毎回コマンドを打っていては面倒なので、バッチファイルを作成しておくと便利です。

【JAXB ソースを生成するバッチファイルの例（generate_jaxb_sample1.bat）】

- 「sample1.xml」という RELAX NG のファイルから、「src¥main¥java¥sample¥core¥jaxb¥sample1」以下のディレクトリに JAXB のソースを生成するバッチファイルです。

```
@echo off

%~d0
cd %~p0

REM RELAX NG のスキーマごとに設定変更する
SET SCHEMA_NAME=sample1
SET PACKAGE_NAME=sample.core.jaxb.sample1

REM パッケージ名をファイルパスに変換する（.⇒¥に変換）
SET SRC_DIR=%PACKAGE_NAME:.=¥%

cd src¥main¥java

REM 初期化
mkdir %SRC_DIR%
del /Q %SCHEMA_NAME%.xsd
del /Q %SRC_DIR%

CALL trang.bat -I rng -O xsd %SCHEMA_NAME%.xml %SCHEMA_NAME%.xsd
%JAVA_HOME%¥bin¥xjc %SCHEMA_NAME%.xsd -p %PACKAGE_NAME%

cd ..¥..¥..¥

pause
```

・環境により値を変更します。

・「RELAX NG ⇒ XML Schema ⇒JAXB ソース」を生成するコマンドを実行します。

【ファイル構成】

```
<プロジェクトフォルダ>
  ├ src¥main¥java
  │      ├ sample1.xml  ← RELAX NG のスキーマファイル（手作業で作成する）
  │      ├ sample1.xsd  ← XML Schema ファイル（trang で自動作成する）
  │      ├ sample¥core¥jaxb¥sample1
  │          ├ XXX.java ← JAXB の Java ソース（xjc で自動作成する）
  │          ・・・
  │
  ├ generate_jaxb_sample1.bat ← RELAX NG⇒JAXB ソースを生成するバッチファイル
  ・・・
```

5.7.5. JAXB のソースを修正するツール

生成したソースなどに「toString()」メソッドを追加するなどの「jaxb-utils.jar」を紹介します。

5.7.5.1. 設定

(1) 以下の URL から「jaxb-utils.jar」をダウンロードします。

「https://docs.google.com/folder/d/0BzR3hjGfqNYFQU82VDJiTzkzaVE/edit」

(2) 「5.7.4　RELAX NG から JAXB ソースの生成」で紹介したバッチファイルに組み込む場合、次のようにプロジェクトフォルダ直下に格納します。

```
<プロジェクトフォルダ>
  ├ src¥main¥java
  │      ├ sample1.xml　← RELAX NG のスキーマファイル（手作業で作成する）
  │      ├ sample1.xsd　← XML Schema ファイル（trang で自動作成する）
  │      ├ sample¥core¥jaxb¥sample1
  │          ├ XXX.java ← JAXB の Java ソース（xjc で自動作成する）
  │          ・・・
  ├ generate_jaxb_sample1.bat ← RELAX NG⇒JAXB ソースを生成するバッチファイル
  ├ jaxb-utils.jar ← ダウンロードした jar ファイル
  ・・・
```

(3) バッチファイルに、必要な各処理を追加します。

```
@echo off

%~d0
cd %~p0

REM RELAX NG のスキーマごとに設定変更する
SET SCHEMA_NAME=sample1
SET PACKAGE_NAME=sample.core.jaxb.sample1

REM パッケージ名をファイルパスに変換する（.⇒¥に変換）
SET SRC_DIR=%PACKAGE_NAME:.=¥%

cd src¥main¥java

REM 初期化
mkdir %SRC_DIR%
del /Q %SCHEMA_NAME%.xsd
del /Q %SRC_DIR%

CALL trang.bat -I rng -O xsd %SCHEMA_NAME%.xml %SCHEMA_NAME%.xsd
%JAVA_HOME%¥bin¥java -cp ..¥..¥..¥jaxb-utils.jar tools.SerializableAppender %SCHEMA_NAME%.xsd UTF-8
%JAVA_HOME%¥bin¥xjc %SCHEMA_NAME%.xsd -p %PACKAGE_NAME%

%JAVA_HOME%¥bin¥java -cp ..¥..¥..¥jaxb-utils.jar tools.SerialVersionUIDReplacer .¥%SRC_DIR% UTF-8
%JAVA_HOME%¥bin¥java -cp ..¥..¥..¥jaxb-utils.jar tools.ToStringMethodAppender .¥%SRC_DIR% UTF-8
%JAVA_HOME%¥bin¥java -cp ..¥..¥..¥jaxb-utils.jar tools.EqualsMethodAppender .¥%SRC_DIR% UTF-8
%JAVA_HOME%¥bin¥java -cp ..¥..¥..¥jaxb-utils.jar tools.HashCodeMethodAppender .¥%SRC_DIR% UTF-8

cd ..¥..¥..¥

pause
```

必要な各処理を追加します。

5.7.5.2. XML Schema に Serializable の定義を追加する

- 「5.7.3.1 ポイント「java.io.Serializable を実装したい」」で紹介した、XML Schema ファイルにシリアライズの定義を追加します。
- 「SerialVersionUIDReplacer」と併用して利用します。

【書式】

```
jar -cp jaxb-utils.jar tools.SerializableAppender <XML Schema ファイル> <ファイルの文字エンコード>
例）
jar -cp jaxb-utils.jar tools.SerializableAppender sample.xsd UTF-8
```

【変更された XML Schema ファイルの例】

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
targetNamespace="urn:yahoo:jp:jlp:FuriganaService" xmlns:ns1="urn:yahoo:jp:jlp:FuriganaService"
xmlns:jxb="http://java.sun.com/xml/ns/jaxb" jxb:version="2.1">
  <xs:annotation>
    <xs:appinfo>
      <jxb:globalBindings>
        <jxb:serializable uid="1"/>
      </jxb:globalBindings>
    </xs:appinfo>
  </xs:annotation>
・・・（省略）・・・
</xs:schema>
```

5.7.5.3. JAXB の Java ソースの「serialVersionUID」の値をランダムな値に置換する

- 「5.7.5.2 XML Schema に Serializable の定義を追加する」により追加し、生成した JAXB のソースは、「serialVersionUID」フィールドの値が“1L”で固定あるため、シリアライズの正確な定義ではユニークな値である必要があります。
- ランダムな値に置換するため、ユニークにならない場合があるかもしれませんが、「SecureRandom」クラスを利用して生成した long 型の値なので、値が衝突する確率は非常に小さいです。

【書式】

```
jar -cp jaxb-utils.jar tools.SerialVersionUIDReplacer <ソースコードのディレクトリパス> <ファイルの文字エンコード
>
例）
jar -cp jaxb-utils.jar tools.SerialVersionUIDReplacer .¥src¥main¥java¥sample¥jaxb UTF-8
```

【変更された Java ソース】

```
・・・（省略）・・・
public class Result
    implements Serializable
{
    private final static long serialVersionUID = -3300303992336383190L;
    @XmlElement(name = "WordList")
    protected WordList wordList;
・・・（省略）・・・
}
```

5.7.5.4. JAXB の Java ソースに「toString()」メソッドを追加する

- 「Commons Lang」の「ToStringBuilder」を利用した「toString()」メソッドを JAXB の生成したソースに追加します。メソッドだけでなく impor 分も追加します。

【書式】

```
jar -cp jaxb-utils.jar tools.ToStringMethodAppender <ソースコードのディレクトリパス> <ファイルの文字エンコード>
例)
jar -cp jaxb-utils.jar tools.ToStringMethodAppender .¥src¥main¥java¥sample¥jaxb UTF-8
```

【変更された Java ソース】

```
・・・(省略)・・・
import org.apache.commons.lang.builder.ToStringBuilder;

public class Result
    implements Serializable
{
・・・(省略)・・・
    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
・・・(省略)・・・
}
```

5.7.5.5. JAXB の Java ソースに「equals()」メソッドを追加する

- 「Commons Lang」の「EqualsBuilder」を利用した「equals()」メソッドを JAXB の生成したソースに追加します。メソッドだけでなく impor 分も追加します。

【書式】

```
jar -cp jaxb-utils.jar tools.EqualsMethodAppender <ソースコードのディレクトリパス> <ファイルの文字エンコード>
例)
jar -cp jaxb-utils.jar tools.EqualsMethodAppender .¥src¥main¥java¥sample¥jaxb UTF-8
```

【変更された Java ソース】

```
・・・(省略)・・・
import org.apache.commons.lang.builder.EqualsBuilder;

public class Result
    implements Serializable
{
・・・(省略)・・・
    @Override
    public boolean equals(Object obj) {
        return EqualsBuilder.reflectionEquals(this, obj);
    }
・・・(省略)・・・
}
```

5.7.5.6. JAXB の Java ソースに「hashCode()」メソッドを追加する

- 「Commons Lang」の「HashCodeBuilder」を利用した「hashCode()」メソッドを JAXB の生成したソースに追加します。メソッドだけでなく impor 分も追加します。
- 「equals()」メソッドを変更した場合、「hashCode()」も変更すべきなので、「EqualsMethodAppender」と合わせて利用します。

【書式】

```
jar -cp jaxb-utils.jar tools.HashCodeMethodAppender <ソースコードのディレクトリパス> <ファイルの文字エンコード>
例)
jar -cp jaxb-utils.jar tools.HashCodeMethodAppender .¥src¥main¥java¥sample¥jaxb UTF-8
```

【変更された Java ソース】

```
・・・(省略)・・・
import org.apache.commons.lang.builder.HashCodeBuilder;

public class Result
    implements Serializable
{
・・・(省略)・・・
    @Override
    public int hashCode() {
        return HashCodeBuilder.reflectionHashCode(this);
    }
・・・(省略)・・・
}
```

5.7.6. JAXB を利用して XML を読み書きする（JAXB のライブラリを使用する）

Spring MVC では、HttpMessageConverter で自動的に Java オブジェクトにマッピングできますが、テストや XML の構造設計時などで直接読み書きしたい場合があるため、その方法を説明します。

5.7.6.1. Marshal(マーシャル)（書き込み）

- JAXBContext から、Marshller のインスタンスを取得する。
- 出力結果を整形したい場合は、Marshaller#getProperty(…)でオプションを設定する。

```
// 1. JAXB コンテキストの作成
// 引数には，パッケージ名(Java のクラス)もしくは，クラスを設定する
JAXBContext context = JAXBContext.newInstance("net.javainthebox.xml");
//　　パッケージ名ならば、JAXB の Java オブジェクトから直接取得しても問題なし
//JAXBContext context = JAXBContext.newInstance(Sample.class.getPackage().getName());

// 2. Marsaller オブジェクトの取得
Marshaller marshaller = context.createMarshaller();

//　出力結果を整形したい場合はプロパティを設定する
//marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

// 3. マーシャリング出力先
//　　出力にはストリームを使用
FileOutputStream out = new FileOutputStream("artists2.xml");

// 4. 出力するオブジェクトの作成
Sample sample = new Sample();
sample.setName("aaaa");

// 5. 書き込み
marshaller.marshal(sample, out);
```

5.7.6.2. Unmarshal(アンマーシャル)（読み込み）

- JAXBContext から、Unmarshller のインスタンスを取得する。

```
// 1. JAXB コンテキストの作成
// 引数には，パッケージ名(Java のクラス)もしくは，クラスを設定する
JAXBContext context = JAXBContext.newInstance("net.javainthebox.xml");
//　　パッケージ名ならば、JAXB の Java オブジェクトから直接取得しても問題なし
//JAXBContext context = JAXBContext.newInstance(Sample.class.getPackage().getName());

// 2. Marsaller オブジェクトの取得
Unmarshaller unmarshaller = context.createUnmarshaller();

// 3. マーシャリング出力先
//　　入力にはストリームを使用（各ストリーム，リーダのインタフェースがある）
FileInputStream in = new FileInputStream("artists2.xml");

// 4. 読み込み
Sample sample = (Sample) unmarshaller.unmarshal(in);
```

5.7.7. JAXB を利用して XML を読み書きする（Spring のライブラリを使用する）(:TODO)

Spring では、JAXB の他に DOM、SAX、XStream など様々なライブラリを使用して XML を読み書きするための機能が抽象化されて用意されている。

5.8. RELAX NG（リラクシング）について

RELAX NG（RELAX Next Generation）は、XML のスキーマ言語の 1 つです。

- RELAX NG の仕様については、次の URL を参考にしてください。
  - RELAX NG 仕様書

  http://www.asahi-net.or.jp/~eb2m-mrt/relaxngjis/jaspec-20011203.html
  - RLAX NG 入門（チュートリアル）

  http://www.kohsuke.org/relaxng/tutorial.ja.html

  （※ブラウザの文字エンコーディングを「Shift_JIS」に変更する必要があります。）
- RELAX NG は 2000 年代前半に出てきて、改良はされていないので新し情報はあまりありません。改良されていないからと言って、廃れたとかではなく、仕様として過不足しておらず十分だったとも言えます。。仕様を調べるにはチュートリアルを見た方が早いと思います。
- Trang で XML Schema に変換する際に、対応してない表現があるため注意してくだい。
  - 詳細は、「5.8.3　ポイント「使用すべきでない RELAX NG の記述」」を参照してくだい。

5.8.1. RELAX NG の基本

5.8.1.1. ファイルの拡張子

- ファイルの拡張子は、「rng」または「xml」。
  - 正確には「rng」ですが、Relaxer の Eclipse のプラグインは既にリンク切れとなってしまったので、エディタの関連付けや扱いやすさを考慮して「xml」の方が良いかもしれません。
  - 自分で、Eclipse のファイルの関連付けとして「rng」を追加して、エディタを「XML エディタ」としてもかまいません。

5.8.1.2. ルート要素（<grammer>）

- ルート要素は、**<grammer>**です。
  - 名前空間は、「http://relaxng.org/ns/structure/1.0」です。
  - データ型として XML Schema の外部定義を利用するため、

    「datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"」も追加します。

【RELAX NG の例】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
   xmlns="http://relaxng.org/ns/structure/1.0"
   datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

     <!-- ========== スキーマを定義します ========= -->

</grammer>
```

5.8.1.3. 要素の定義（<define>、<ref>、<element>）

- XML の要素を定義する場合、まず<define name="定義名">で定義して名前を付けます。
  - <define>で定義したものは、<ref name="定義名">で参照ができます。
  - <define>の子要素に、<element name="要素名">を使用して、XML の要素を定義します。
  - 可読性のために、<def>と、その直下の<element>要素の属性 name を合わせておいた方がよいです。
- JAXB の Java ソースにしたとき、<define>要素で定義した単位に Java ソースが作成されます。
  - Java ソースのクラス名は、<define>要素の直下の<element>要素の属性 name の値がクラス名になります（<element name="要素名=クラス名">）。

【RELAX NG の例】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
  xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

    <define name="SampleJaxb3">
        <element name="SampleJaxb3">
            <ref name="Result"/>
        </element>
    </define>

    <define name="Result">
        <element name="Result">
            ・・・（省略）・・・
        </element>
    </define>

</grammar>
```

・属性 name の値を揃えておくと、間違いが少なくなります。
・<element>の属性 name が Java のクラス名になります。

・<define name=”Result”>を参照します。
・複数の場所から参照できます。

要素<Result>の定義。

【XML の例】

```
<SampleJaxb3>
    <Result>
        ・・・（省略）・・・
    </Result>
</SampleJaxb3>
```

【JAXB の生成される Java ソースファイル】

```
SampleJaxb3.java
Result.java
```

5.8.1.4.　オプション扱いの要素の定義（<optional>）

- 子要素をオプション（必須でない）扱いにしたい場合、**<optional>要素**で囲みます。

【RELAX NG の例】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

    <define name="ResultInfo">
        <element name="ResultInfo">
            <optional>
                <ref name="Result"/>
            </optional>
        </element>
    </define>

    <define name="Result">
        <element name="Result">
            ・・・(省略)・・・
        </element>
    </define>

</grammar>
```

【JAXB の Java ソースの例 1（<optional>要素で囲まない）】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "result"
})
@XmlRootElement(name = "ResultInfo")
public class ResultInfo {

    @XmlElement(name = "Result", required = true)
    protected Result result;

    /** setter, getter は省略 */
}
```

・属性「requred = true」が付与され必須扱いになります。
・この要素がないと、マッピング時に例外が発生します

【JAXB の Java ソースの例 2（<optional>要素で囲む。）】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "result"
})
@XmlRootElement(name = "ResultInfo")
public class ResultInfo {

    @XmlElement(name = "Result")
    protected Result result;

    /** setter, getter は省略 */
}
```

@XmlElement の属性から「requred = true」が除去されオプション扱いになります。

5.8.1.5.　0 個以上要素が連続した場合（<zeroOrMore>）

- 0 個以上の同じ要素を連続させて、リスト化したい場合は、<zeroOrMore>で囲みます。
  - <zeroOrMore>は、0 個以上の要素があってもよく、要素が空（=null）でもよいという意味です。

【RELAX NG の例】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

    <define name="SampleJaxb3">
        <element name="SampleJaxb3">
            <zeroOrMore>
                <ref name="Result"/>
            </zeroOrMore>
        </element>
    </define>
    <define name="Result">
        <element name="Result">
        </element>
    </define>
</grammar>
```

【XML の例】

```
<!-- 要素が複数個ある場合 -->
<SampleJaxb3>
    <Result> </Result>
    <Result> </Result>
</SampleJaxb3>

<!-- 要素 0 個の場合 -->
<SampleJaxb3>
</SampleJaxb3>
```

【JAXB の Java ソースの例】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "result",
    "processingTime"
})
@XmlRootElement(name = "SampleJaxb3")
public class SampleJaxb3 {

    @XmlElement(name = "Result")
    protected List<Result> result;

    public List<Result> getResult() {
        if (result == null) {
            result = new ArrayList<Result>();
        }
        return this.result;
    }
}
```

・要素が空（=null）の場合は、リストのインスタンスが自動的に生成されます。

5.8.1.6. 1 個以上要素が連続した場合（<oneOrMore>）

- 1 個以上の同じ要素を連続させて、リスト化したい場合は、<oneOrMore>で囲みます。
  - <oneOrMore>は、1 個以上の要素があってもよく、**要素が必須（!= null）**という意味です。

【RELAX NG の例】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

    <define name="SampleJaxb3">
        <element name="SampleJaxb3">
            <oneOrMore>
                <ref name="Result"/>
            </oneOrMore>
        </element>
    </define>
    <define name="Result">
        <element name="Result">
        </element>
    </define>
</grammar>
```

【XML の例】

```
<!-- 要素が複数個ある場合 -->
<SampleJaxb3>
    <Result> </Result>
    <Result> </Result>
</SampleJaxb3>
```

【JAXB の Java ソースの例】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "result",
    "processingTime"
})
@XmlRootElement(name = "SampleJaxb3")
public class SampleJaxb3 {

    @XmlElement(name = "Result", required = true)
    protected List<Result> result;

    public List<Result> getResult() {
        if (result == null) {
            result = new ArrayList<Result>();
        }
        return this.result;
    }
}
```

・必須制約が付加され、要素数が 0 個の場合、バインディング時に例外が発生します。

5.8.1.7.　要素の値の定義（<text>、<data>）

- XML の要素の値を定義する場合、要素定義の<element>の子要素として、データ型を<text>または<data>を使用して定義します。
  - <text> : Java の String 型になります。RELAX NG の標準ではデータ型は<text>しかないので、実際には<data>を使った方が良いと思います。**要素の値を<text>で定義するとオプション扱い**になります。
  - <data type= "XML Schema のデータ型">: XML Schema のデータ型。設定可能なデータ型は「5.8.2 XML Schema のデータ型」を参照してください。
- 要素の値を持ち、かつ子要素を持つ要素は定義できません。です。理由として、インデントによる改行や空白も値として含まれるため、要素の値とインデントの区別がつかなくなるためです。
- 属性を持たない子要素を<element>に直接定義した場合、生成される JAXB の Java ソースは、親要素のフィールドとして定義され、別クラスとして分離されません。
  - 詳細は、「5.8.4　ポイント「JAXB の Java ソースのクラス分割の制御」」を参照してくだい。

【RELAX NG の例】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

    <define name="Item1">
        <element name="Item1">
            <text/>
            <attribute name="id"><data type="int"/></attribute>
        </element>
    </define>

    <define name="Item2">
        <element name="Item2">
            <element name="name"><data type="int"/></element>
            <element name="value"><text/></element>
        </element>
    </define>

</grammar>
```

<element>の直下にデータ型を定義します。

・属性を持たない子要素を直接定義した場合、Java ソースは分割されません。

・<text>を要素とするとオプション扱いになります。
・次の定義と同義です。
<optional>
<element name="value"><data type="string">
</optinal>

【XML の例】

```
<!-- 値を持つ要素 -->
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Item1 id="1">あああああ</Item1>

<!-- 属性を持たない子要素を持つ -->
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Item2>
    <name>3</name>
    <value>あああああ</value>
</Item2>
```

【JAXB の Java ソースの例 1：値を持つ要素】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
      "content"
})
@XmlRootElement(name = "Item1")
public class Item1 {

    @XmlValue
    protected String content;

    @XmlAttribute(required = true)
    protected int id;

   /** setter, getter は省略  */
}
```

要素の値は、“content”フィールドで定義されます。

【JAXB の Java ソースの例 2: 属性を持たない子要素を持つ】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
      "name",
      "value"
})
@XmlRootElement(name = "Item2")
public class Item2 {

    protected int name;

    protected String value;

    /** setter, getter は省略  */

}
```

子要素が、フィールドとして定義されます。

5.8.1.8. 属性の定義（<attribute>）

- XML の属性を定義する場合、<attribute name= "属性名">で定義します。
- 属性の値のデータ型を、<attribute>の要素として定義します。
  - <text> : Java の String 型になります。RELAX NG の標準ではデータ型は<text>しかないので、実際には<data>を使った方が良いと思います。
  - <data type= "XML Schema のデータ型">: XML Schema のデータ型。設定可能なデータ型は「5.8.2 XML Schema のデータ型」を参照してください。
- 属性をオプション（必須ではない）にしたい場合、<optional>要素で囲みます。
  - 複数の属性定義を 1 つの<optional>で囲むと、Trang で変換する際に警告がでるため、1 つずつ囲みます。
  - 数値型を<optional>で囲むと、オプション扱いになるため生成される Java ソースはプリミティブ型のラッパークラスになり、null で値がないことを表現します。

【RELAX NG の例】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
   xmlns="http://relaxng.org/ns/structure/1.0"
   datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

      <define name="Person">
            <element name="Person">
                  <attribute name="id"><text/></attribute>
                  <attribute name="name"><data type="string"/></attribute>
                  <attribute name="age"><data type="int"/></attribute>
                  <optional>
                        <attribute name="birthYear"><data type="int"/></attribute>
                  </optional>
            </element>
      </define>
</grammar>
```

データ型を文字列に指定したい場合は、2 通りあります。

オプションにしたい場合、<optional>で囲みます。

【XML の例】

```
<Person age="20" name="name-01" id="id-01"/>
```

【JAXB の Java ソースの例】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "")
@XmlRootElement(name = "Person")
public class Person {

    @XmlAttribute(required = true)
    @XmlSchemaType(name = "anySimpleType")
    protected String id;

    @XmlAttribute(required = true)
    protected String name;
```

```
    @XmlAttribute(required = true)
    protected int age;

    @XmlAttribute
    protected Integer birthYear;

    /** setter, getter は省略 */

}
```

オプションにすると、プリミティブ型のラッパークラスになります。

5.8.1.9. 名前空間の定義

- 名前空間を定義するには、属性として「ns="名前空間"」を追加します。
- **要素を定義する<element>では名前空間は親から継承される**ため、親にのみ定義すればよいです。すなわち、RELAX NG のルート要素の<grammar ns="名前空間">に追加します。
- しかし、**属性を定義する<attribute>では、名前空間は親から継承されない**ため、個別に定義する必要があります。
- 同じ RELAX NG ファイルの中で**要素に対して複数の名前空間を定義することができません**。
  - XML Schema 上は問題ないですが、JAXB のソースを生成する際にエラーが出ます。
  - ただし、**属性に対しては異なる名前空間を複数定義することができます**。

【RELAX NG ファイルの定義】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
  xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
  ns="http://sample.co.jp/sample1">

    <define name="NSSample1">
        <element name="NSSample1">
            <element name="child1"><data type="string"/></element>

            <attribute name="id">
                <data type="int"/>
            </attribute>
        </element>
    </define>

    <define name="NSSample2">
        <element name="NSSample2" ns="http://sample.co.jp/sample">
            <element name="child1"><data type="string"/></element>
            <attribute name="id" ns="http://sample.co.jp/sample2">
                <data type="int"/>
            </attribute>

            <attribute name="value">
                <data type="string"/>
            </attribute>
        </element>
    </define>

</grammar>
```

<element>の場合、親（<grammar>）の名前空間を継承する。

属性の場合、親の名前空間を継承しないので、個別に定義する必要がある。

【XML の例 1：要素のみに名前空間を付与した場合】

- RELAX NG で要素のみに名前空間を付与した場合、JAXB による生成された XML には、ルート要素に名前空間の宣言がある。
  - ➢ 名前空間のプレフィックスは付かない。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<NSSample1 id="1" xmlns="http://sample.co.jp/sample1">
    <child1>あああ</child1>
</NSSample1>
```

【XML の例 2：属性のみ名前空間を付与した場合】

- RELAX NG で属性のみに名前空間を付与した場合、JAXB には、“プレフィックス付き”の名前空間が定義される。
  - ➢ 属性と要素の名前空間は、処理上、別々に扱うため“プレフィックス付き”になる。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<NSSample1 id="1" xmlns:ns2="http://sample.co.jp/sample2">
    <child1>あああ</child1>
</NSSample1>

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<NSSample2 value="属性 02" ns2:id="1" xmlns:ns2="http://sample.co.jp/sample2">
    <child1>いいい</child1>
</NSSample2>
```

属性は、必ず“プレフィックス付き”の名前空間になる。

【XML の例 3：要素と属性に名前空間を付与した場合】

- RELAX NG で要素と属性に名前空間を付与した場合、要素も“プレフィックス付き”の名前空間になる。
  - ➢ “プレフィックス”の値は、定義した順に連番が付与され「ns<連番>」となる。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns1:NSSample1 id="1" xmlns:ns1="http://sample.co.jp/sample1" xmlns:ns2="http://sample.co.jp/sample2">
    <ns1:child1>あああ</ns1:child1>
</ns1:NSSample1>

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns1:NSSample2 value="属性 02"
 ns2:id="1" xmlns:ns1="http://sample.co.jp/sample1" xmlns:ns2="http://sample.co.jp/sample2">
    <ns1:child1>いいい</ns1:child1>
</ns1:NSSample2>
```

子要素に名前空間が継承されています。

属性には、定義を追加したものしか名前空間が付加されません。

【JAXB の Java ソースの例 1：名前空間の付与】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "child1"
})
@XmlRootElement(name = "NSSample1", namespace = "http://sample.co.jp/sample1")
public class NSSample1 {

    @XmlElement(namespace = "http://sample.co.jp/sample1", required = true)
    protected String child1;
```

親の@XmlRootElment の名前空間が、子要素に付与される。

```
    @XmlAttribute(required = true)
    protected int id;

    /** setter, getter は省略 */

}
```

【JAXB の Java ソースの例 2：属性に名前空間を付与】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "child1"
})
@XmlRootElement(name = "NSSample2", namespace = "http://sample.co.jp/sample1")
public class NSSample2 {

    @XmlElement(namespace = "http://sample.co.jp/sample1", required = true)
    protected String child1;

    @XmlAttribute(namespace = "http://sample.co.jp/sample2", required = true)
    protected int id;

    @XmlAttribute(required = true)
    protected String value;

    /** setter, getter は省略 */
}
```

追加した属性のみに名前空間が付与される。

5.8.2. XML Schema のデータ型

RELAX NG で定義可能なデータ型を示します。

- XML Schema のデータ型の場合、<data type="データ型">で指定します。
- Date などの日時のデータ型も用意されていますが、フォーマットの問題もあるので、実際には文字列や long(ミリ秒)で表現した方がよいです。
  - JAXB を上手く使えば、書式を指定し日時をマッピングできますが、今回は RELAX NG から自動で生成するという趣旨のため割愛します。
  - JAXB で自動的にフォーマットする機能を利用してもよいですが、システムに依存してしまうため、「<date format="yyyy-MM-dd">2012-01-13<date>」のように属性に書式を持たせて、値の変換自体は APP 側でする方式が良いと思います。

表 5.15　RELAX NG のデータ型と Java のデータ型の対応

No.	REELAX NG のデータ型	Java のデータ型
1	属性に<text />	java.lang.String ただし、XML Schema では、「anySimpleType」と解釈する。
2	要素に<text />	java.lang.String ただし、かならずオプション扱い（必須ではない）になります。

表 5.16 XML Schema のデータ型と Java のデータ型の対応

種類.	XML		Java	
	データ型	精度など	データ型	精度など
文字	string	—	String	—
数値	boolean	true、false、1、0	boolean/Boolean	true、false
	byte	-128～128	byte/Byte	XML と同じ。
	short	-32,768～32,767	short/Short	XML と同じ。
	int	-214,7483,648～ 214,7483,647	int/Integer	XML と同じ。
	long	-9,223,372,036,854,775,808 ～ 9,223,372,036,854,775,807	long/Long	XML と同じ。
	float	IEEE 単精度 32bit 浮動小数	floag/Float	XML と同じ。
	double	IEEE 倍精度 64bit 浮動小数	double/Double	XML と同じ。
	integer	無限制度の整数	java.math.BigInteger	—
	decimal	無限制度の 10 進数	java.math.BigDecimal	—
	unsignedByte	0～255	short/Short	—
	unsignedShort	0～65,535	int/Integer	—
	unsignedInt	0～4,294,967,295	long/Long	—
	unsignedLong	0～ 18,446,744,073,709,551,615	java.math.BigInteger	—
	positiveInteger	1 以上の無限制度の整数	java.math.BigInteger	—
	negativeInteger	-1 以下の無限制度の整数	java.math.BigInteger	—
	nonPositiveInteger	0 以下の無限制度の整数	java.math.BigInteger	—
	nonNegativeInteger	0 以上の無限制度の整数	java.math.BigInteger	—
日時	dateTime	—	javax.xml.datatype.XM LGregorianCalendar	—
	date	—		—
	time	—		—
他	base64Binary	—	byte[]	—
	hexBinary	—	byte[]	—
	duration	—	javax.xml.datatype.Du ration	—
	NOTATION	—	javax.xml.namespace.Q Name	—

5.8.3. ポイント「使用すべきでない RELAX NG の記述」

Trang で解釈できない RELANX NG の定義や、使用すべきでない定義などを説明します。

5.8.3.1. 効果がないタグ（<start>）

- ルート要素を指定する<start>タグは、XML Schema、JAXB では仕様としてないため表現できません。
  - これは、RELAX NG は再帰的な要素の定義を許可していないためです。
  - 可読性を高めるために、利用するのもよいと思います。
  - JAXB のソースでは、<define>で定義した要素が全てルート要素を取れるため効果はありません。

```
<start>
   <element name="ルート要素">
        ・・・（省略）・・・
   </element>
</start>
```

5.8.3.2. 効果がないタグ（<choice>）

- 属性の値を制限し、択一させる<choise>を設定した場合、JAXB にマッピングする際に値はチェックされません。
  - <choice>で定義した値以外を設定していてもエラーとなりません。
- XML Schema 上はきちんと表現されているため、JAXB 以外の XML の Validator などでチェックするような場合は、効果があります。

【RELAX NG の例】

```
<define name="Result">
    <element name="Result">
        <attribute name="type">
            <choice>
                <value>html</value>
                <value>xml</value>
            </choice>
        </attribute>
        <optional>
            <attribute name="int"><data type="int"/></attribute>
        </optional>
    </element>
</define>
```

値を制限し、択一させるためのタグ。

【XML Schema の例】

```
  <xs:element name="Result">
    <xs:complexType mixed="true">
      <xs:attribute name="id" use="required" type="xs:int"/>
      <xs:attribute name="type" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:enumeration value="html"/>
            <xs:enumeration value="xml"/>
          </xs:restriction>
        </xs:simpleType>
```

XML Schema では、択一が表現されている。

```
        </xs:attribute>
    </xs:complexType>
  </xs:element>
```

【JAXB の Java ソースの例】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "content"
})
@XmlRootElement(name = "Result")
public class Result {

    @XmlAttribute(required = true)
    protected int id;

    @XmlAttribute(required = true)
    @XmlJavaTypeAdapter(CollapsedStringAdapter.class);
    protected String type;

    /** setter, getter は省略 */
}
```

自分で択一の制限をチェックする Adapter を作成する必要がある。

5.8.3.3. 仕様すべきでないタグ（<interleave>）

- 異なる子要素を複数持つ場合で、定義の順序を無視するタグ<interleave>を使うと、JAXB の Java クラスが操作し辛らくなるため、使用しない方がよいです。

【RELAX NG の例】

```
<define name="Result">
    <element name="Result">
        <interleave>
            <element name="data1"><text/></element>
            <element name="data2"><text/></element>
            <element name="data3"><text/></element>
        </interleave>
    </element>
</define>
```

要素の定義順を無視し、依存しないようにするタグ。

【JAX の Java ソース】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "content"
})
@XmlRootElement(name = "Result")
public class Result {
    @XmlElementRefs({
        @XmlElementRef(name = "data3", type = JAXBElement.class),
        @XmlElementRef(name = "data1", type = JAXBElement.class),
        @XmlElementRef(name = "data2", type = JAXBElement.class)
    })
    protected List<JAXBElement<String>> data1OrData2OrData3;

    /** setter, getter は省略 */
}
```

汎用的なリストになり、オブジェクトの作成時や取り出し時に判定する必要が出てくる。

5.8.4. ポイント「JAXB の Java ソースのクラス分割の制御」

JAXB で Java オブジェクトとしてデータを取得する際や作成する際に、クラスの粒度が細かすぎると、処理が複雑になったり、冗長になったります。うまく RELAX NG のスキーマを定義することで、クラスの粒度を制御することができます。

5.8.4.1. <define>タグで定義した単位でクラス（ファイル）が作成される

- RELAX NG で<define>タグで定義した単位に最終的に JAXB の Java ソースが分割されます。
- <define>タグで定義していても、属性などがなく要素の値だけの場合、クラスは分割されません。
  - xjc コマンドで一緒に作成される「ObjectFactory.java」経由で生成することとなります。

【RELAX NG の例】

```
<grammer>
    <define name="Item1">
        <element name="Item2">
        ・・・(省略)・・・
        </element>
    </define>

    <define name="Item2">
        <element name="Item2">
        ・・・(省略)・・・
        </element>
    </define>

    <define name="Item3">
        <element name="Item3">
            <data type="string"/>
        </element>
    </define>
</grammer>
```

属性を持たない値だけの場合、ソースファイルは生成されません。

【JAXB の生成される Java ソースファイル】

```
Item1.java
Item2.java
ObjectFactory.java
```

【JAXB のソースの ObjectFactory.java】

```
@XmlRegistry
public class ObjectFactory {

    private final static QName _Item3_QNAME = new QName("", "Item3");

    @XmlElementDecl(namespace = "", name = "Item3")
    public JAXBElement<String> createItem3(String value) {
        return new JAXBElement<String>(_Item3_QNAME, String.class, null, value);
    }
}
```

ObjectFacotry.java クラスのめどっどから作成する。

5.8.4.2.　属性を持たない値をだけを持つ要素はクラス（ファイル）が生成されない

- XML では要素として扱いたいが、Java ソース上では属性と同じようにフィールドとして扱って簡単にデータを取得、設定したい場合にこの特性を利用します。
- XML では属性の個数が増えると、人から見て非常に見づらくなります。

【RELAX NG の例】

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

    <define name="Item2">
        <element name="Item2">
            <element name="name"><data type="int"/></element>
            <element name="value"><text/></element>
        </element>
    </define>

</grammar>
```

・属性を持たない子要素を直接定義した場合、Java ソースは分割されません。

【XML の例】

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Item2>
    <name>3</name>
    <value>あああああ</value>
</Item2>
```

【JAXB の Java ソースの例】

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "name",
    "value"
})
@XmlRootElement(name = "Item2")
public class Item2 {

    protected int name;
    protected String value;

    /** setter, getter は省略  */

}
```

子要素が、フィールドとして定義されます。

5.8.4.3. 属性を持つ子要素はクラス（ファイル）が作成される

- RELAX NG で、子要素として定義していても属性を持つ場合、JAXB の Java ソースが分割されます。
  - これは、XML の属性と Java クラスの属性（フィールド）が考え方が対応しているためです。
- このような場合は、<define>タグで抽出して別に定義すべきです。
  - RELAX NG の可読性が上がり、また JAXB のソース分割を意識した際に理解しやすくなります。

【RELAX NG の例：変更前】

```
<grammer>
    <define name="Item6">
        <element name="Item6">
            <attribute name="name"><text/></attribute>
            <element name="Child6">
                <attribute name="attr1"><text/></attribute>
            </element>
        </element>
    </define>
</grammer>
```

属性を持つとファイルがクラスとして分割されます。

【JAXB の生成される Java ソースファイル】

```
Item6.java
Chld6.java
```

【RELAX NG の例：変更後】

```
<grammer>
    <define name="Item6">
        <element name="Item6">
          <attribute name="name"><text/></attribute>
            <ref name="Child6"/>
        </element>
    </define>

    <define name="Child6">
        <element name="Child6">
            <attribute name="attr1"><text/></attribute>
        </element>
    </define>

</grammer>
```

<define>で別定義にする。

6. EL 式(Expression Language)

JSP2.0（Tomcat5.5）より、EL 式（Expression Language）が導入されました（実際に使用可能なのは、Tomcat6.0 から？）。EL 式は、式言語とも呼ばれ、JSP の中で「演算結果」「値の参照」「メソッドの呼び出し」など様々なことが簡単に呼び出すことができます。

6.1. EL 式の基本

【基本形】

- EL 式は、「${式}」の形式で記述し、{}で囲まれた式を計算し、計算結果を出力するものです。

```
${式}
```

【値の参照】

- session、Request(Model)、pageContext に格納されている値を出力する場合は、名称を記述します。
  通常は、複数の種類があるセッションスコープは区別されません。
- Bean の中のプロパティを呼び出す際には、ドット「.」でプロパティ名を続けで記述します。
- これらの式は、JSP の中で、基本的にどこでも記述することができます。
  例えば、タグの属性中や、<script>タグの中など。
- EL 式で出力した値は、標準では HTML エスケープされません。
  エスケープし出力したい場合、JSTL「<c:out/>」を使用するか、EL Function のライブラリ（「6.4 Amateras 「Java Standard EL Functions（JSEL）」」参照）である「${f:h(式)}」を使用します。
  他のライブラリとして、「6.3　JSTL Functions」があり、「${fn:escapeXml(式)}」を使用します。

```
<%-- 変数の呼び出し --%>
${var}

<%-- Bean のプロパティ name の呼び出し --%>
${loginUser.name}

<%-- JSTL による値の出力 --%>
<c:out value="${loginUser.name}" />

<%-- EL Function による値の出力 --%>
${fn:escapeXml(loginUser.name)}
${f:h(loginUser.name)}
```

※値が null または、存在しないオブジェクトを参照した場合、何も出力されません。

【Unified EL（遅延評価式）】

- 「#{式}」とすることで式の値を評価せずに、式そのものを渡すことができます。
  独自の JSTL などで、式の評価を JSTL 内で行う際などに使用します。
- JSF(Java Server Faces)の独自 EL 式と区別し、一緒に使用するために導入されたもので、あまり使う機会はないと思います。
- TECHSCORE Unifed EL「http://www.techscore.com/tech/Java/JavaEE/JSP/15-2/」

【値の参照（スコープの指定)】

- 通常、EL 式はスコープの区別なく、全てのスコープを参照します。
  参照する際の優先度は、スコープの狭い順が高くなっています。
- 特定のスコープのオブジェクトにアクセスすることも可能です。
  例えば、session スコープを参照する場合、「sessionScope」を式の先頭に付けます。
- 暗黙オブジェクトの詳細は、「6.1.1　EL 式の暗黙オブジェクト」を参照してください。

```
<%-- session スコープへの参照 --%>
${sessionScope.loginUser.name}

<%-- application スコープへの参照 --%>
${applicationScope.loginUser.name}
```

表 6.1　スコープの EL 式での記述

優先度	スコープの種類	EL 式での記述
1	Page スコープ	pageScope
2	Request スコープ(Model と同じです) /Flash スコープ(リダイレクト先に Model として渡される)	requestScope
3	Sesion スコープ	sessionScope
4	Application スコープ	applicationScope

【計算】

- EL 式内において、演算子を使用することができます。
  「算術演算子」「比較演算子」「論理演算子」「三項演算子」などが利用できます。
- 演算子の詳細は、「6.1.2　EL 式の演算子」を参照してください。

```
<%-- 算術演算子 --%>
${5*2}
<%-- 比較演算子 --%>
${3 > 5}
<%-- 論理演算子 --%>
${!(3==5)}
<%-- 三項演算子 --%>
${3 > 2 ? 100:200}
<%-- 空または null かどうか--%>
${empty data}
```

【リスト（配列）の参照】

- リストや配列を参照する場合は、「プロパティ名[インデックス]」のように、インデックスを数値で指定します。
- リストと配列区別はありません。

```
sample[0]
sample[1].name
```

【マップの参照】

- マップを参照する場合は、「プロパティ名[キー]」、キーを指定します。
- 他の指定方法として、「プロパティ名.キー」のようにも指定できます。
  この形式は、Spring の Validate 時のパス、エラーメッセージのパスと統一するために、使わないことをお勧めします。
- マップのリストなど、組み合わせても使用できます。

```
sample[red]
sample[red].name
sample[red][0].age
```

【リテラル】

EL 式で使用可能なリテラルを、次の「表 6.2」に示します。基本的に、Java と同じです。

表 6.2　EL 式でのリテラル

No.	種類	値
1	ブール値	true、false
2	整数	Java と同様。 例）1234
3	浮動小数点	Java と同様。 例)）0.1234、1e6
4	文字列	引用符と二重引用符。" は ¥"、' は ¥'、¥ は ¥¥ とエスケープします。 例）"あいう"、"円マーク¥"です"
5	NULL	null

6.1.1. EL 式の暗黙オブジェクト

EL 式では、JSP のように暗黙オブジェクトが準備されています。スコープの他にも、様々なものがあります。

- TECHSCORE EL 式「http://www.techscore.com/tech/Java/JavaEE/JSP/12/」
- 「http://struts.wasureppoi.com/jsp/04_el_pageContext.html」）

表 6.3　EL 式のスコープの種類

優先度(※1)	スコープの種類	EL 式での記述	JSP での記述
1	Page スコープ	pageScope	page
2	Request スコープ (Model と同じです)	requestScope	request
3	Sesion スコープ	sessionScope	session
4	Application スコープ	applicationScope	application

※1 スコープを省略して参照した場合の優先度。

表 6.4　EL 式で使用可能な暗黙オブジェクト

No.	オブジェクト名	説明
1	pageContext	JSP ファイルのコンテキスト。 context、session、request の各オブジェクトにアクセスできます。 例) ${pageContext.session.id}
2	param	リクエストパラメータと値のマップ。 例) ${param.arg1}
3	paramValues	リクエストパラメータと配列値のマップ。 例) ${paramValue.arg[0]}
4	header	ヘッダー名と値のマップ。 例) ${headerValues['user-agent']}
5	headerValues	ヘッダー名と配列のマップ。 例) ${headerValues['user-agent'][0]}
6	initParam	web.xml に定義した初期化パラメータを持つ Map オブジェクト。 例) ${initParam["param1"]}
7	cookie	Cookie 名と cookie オブジェクトを対応させた Map オブジェクト。 例) ${cookie["param1"].value}

6.1.2. EL 式の演算子

EL 式では、比較などの演算子が使用することができます。

- 基本的に、文字列に対して使用することはできません。使用した場合は例外が発生するので注意してください。
  だたし、比較演算子「empty」など一部の演算子は文字列に対して使用することはできます。
- 文字列に対して、様々な処理を行いたい場合は、EL Function を自前で作成するか、既存のライブラリを使用します。ライブラリについては、「6.3　JSTL Functions」「6.4　Amateras 「Java Standard EL Functions（JSEL)」」を参照してください。
- 参考「http://struts.wasureppoi.com/jsp/03_enzan.html」

表 6.5　EL 式で使用可能な演算子

分類	No.	演算子	別名(※1)	説明
算術演算子	1	+		加算。
	2	-		減算。
	3	*		除算。
	4	/	div	除算。 別名を使用すると、整数どうしの演算でも結果は実数(double)となる。 0 除算でも例外は発生せずに、結果として文字列'Inifinity'を返す。
	5	%	mod	剰余。
比較演算子	1	==	eq	等しい。
	2	!=	ne	等しくない。
	3	<	lt	小さい。
	4	>	gt	大きい。
	5	<=	le	以下。
	6	>=	ge	以上。
	7	empty		null または空文字。
論理演算子	1	&&	and	集合積(AND)。
	2	||	or	集合和(OR)。
	3	!	not	否定(NOT)。
三項演算子	1	a ? b : c		条件演算。 「a」の場合は「b」、「a」以外の場合は「c」

※1 別名で使用可能な演算子。

6.1.3. EL 式の演算子の優先順位

- EL 式内での演算子の優先順序は、基本的に Java と同様です。

表 6.6 EL 式の演算子の優先順位

優先度	演算子
1	[],.
2	()
3	単項の-、not、!、empty
4	*、/、div、%、mod
5	+、二項の-
6	()<,>,<=、>=、lt、gt、le、ge
7	==、!=、eq、ne
8	&、and
9	||、or

6.2. EL Function の作成(:TODO)

6.3. JSTL Functions

JSP Standard Tag library には、EL 式内で使用可能な関数が準備されています。

- TECHSCORE「http://struts.wasureppoi.com/jstl/04_function.html」
- Java の道「http://www.javaroad.jp/opensource/js_taglibs8.htm」

6.3.1. JSTL Functions の設定／使用例

【pom.xml】

- 依存ファイルに、JSTL のライブラリを追加します。

```
<project>
・・・省略
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>
    </dependencies>
・・・省略
</project>
```

【JSP の設定】

- Taglib ディレクティブを追加します。

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn"%>
<%-- エスケープし出力します。 --%>
${fn:escapeXml(message1)}
```

6.3.2. JSTL Functions の一覧

表 6.7　JSTL Funcstions の一覧

No.	EL 関数	説明
1	fn:contains	ある文字列の中に、指定された文字列が含まれるかを調べる。 [書式]boolean fn:contains(string, searchString)
2	fn:containsIgnoreCase	ある文字列の中に、指定された文字列が含まれるかを調べる。調べる際、大文字小文字の違いは区別されない。 [書式]boolean fn:containsIgnoreCase(string, searchString)
3	fn:endsWith	ある文字列の最後に、指定された文字列が含まれるかを調べる。 [書式] boolean fn:endsWith(string, suffix)
4	fn:escapeXml	XML で解釈される文字記号（<、>、&、'、"）を、HTML で表示できる文字記号(<、>、&、'、")に置き換えて出力する。 [書式] String fn:escapeXml(string)

5	fn:indexOf	ある文字列の中で、指定された文字列がはじめて合致した際、合致した場所の index 番号を返す。 １文字目は、「1」、見つからない場合は「-1」を返す。 [書式] int fn:indexOf(string, substring)
6	fn:join	配列内の要素を一連の文字列として出力する。 [書式] String fn:join(array, separator)
7	fn:length	配列、Collection オブジェクトの要素数、文字列の文字数をカウントして、その数を返す。 [書式] String fn:length(input)
8	fn:replace	引数に指定された置き換え前文字列に合致する文字列を、置き換え後文字列に変換して出力する。置き換え前文字列に合致するすべての文字列が置き換わる。 [書式] String fn:replace(inputString, beforeText, afterText)
9	fn:split	文字列を引数に指定された区切り文字でわけ、配列に変換して出力する。 [書式] String[] fn:split(string, delimiters)
10	fn:startsWith	ある文字列の最初に、指定された文字列が含まれるかを調べる。 [書式] boolean fn:startsWith(string, prefix)
11	fn:substring	index 番号を指定して、文字列内の特定文字列を抜き出す。 [書式] String fn:substring(string, beginIndex, endIndex)
12	fn:substringAfter	引数に指定する区切り文字列以降の文字列を抜き出す。 [書式] String fn:substringAfter(string, substring)
13	fn:substringBefore	引数に指定する区切り文字列以前の文字列を抜き出す。 [書式] String fn:substringBefore(string, substring)
14	fn:toLowerCase	文字列を小文字に変換して出力する。 [書式] String fn:toLowerCase(string)
15	fn:toUpperCase	文字列を大文字に変換して出力する。 [書式] String fn:toUpperCase(string)
16	fn:trim	文字列の両端の空白文字を削除して出力する。 全角の空白は削除されません。 [書式] String fn:trim(string)

6.4. Amateras 「Java Standard EL Functions（JSEL)」

プロジェクト「Amateras」にて、EL 関数のライブラリが公開されている。主に 2 つの機能があり、基本関数として、文字列処理（エスケープ、結合）とログ出力関数がある。

- JSEL のサイト「http://amateras.sourceforge.jp/cgi-bin/fswiki/wiki.cgi?page=JSEL」
- JSEL の Maven サイト「http://amateras.sourceforge.jp/site/functions/quickstart.html」

6.4.1. JSEL の設定

【pom.xml の編集】

- Amateras の Maven リポジトリサイトと、依存ファイルを追加します。

```
<project>
・・・省略
    <repositories>
        <repository>
            <id>amateras</id>
            <name>Project AMateras Maven2 Repository</name>
            <url>http://amateras.sourceforge.jp/mvn/</url>
        </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>jp.sf.amateras.functions</groupId>
            <artifactId>functions</artifactId>
            <version>1.1.2</version>
        </dependency>
    </dependencies>
    ・・・省略
</project>
```

【web.xml の編集】

- フィルタを追加します。

```
<web-app>
・・・省略
    <!-- Amateras Java Standard EL Functions -->
    <filter>
        <filter-name>functionsFilter</filter-name>
        <filter-class>jp.sf.amateras.functions.filter.FunctionsFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>functionsFilter</filter-name>
        <url-pattern>*</url-pattern>
        <dispatcher>REQUEST</dispatcher>
    </filter-mapping>
・・・省略
</web-app>
```

【functions.properties の作成】

- f:date()、f:datetime()、f:time()などの関数はデフォルトのフォーマットパターンが決められています。
- これらのフォーマットパターンを変更するにはクラスパスルートに「**functions.properties**」というプロパティファイルを作成します。
  Maven のプロジェクト形式の場合、「/src/main/resources/」直下に配置します。

```
# f:u()で URL エンコードするときの文字コード
defaultEncoding=UTF-8

# f:date()でフォーマットするパターン
datePattern=yyyy/MM/dd

# f:datetime()でフォーマットするパターン
datetimePattern=yyyy/MM/dd HH:mm:ss

# f:time()でフォーマットするパターン
timePattern=HH:mm:ss
```

6.4.2. JSEL の使用例

【基本関数の使用例】

- 基本関数用の Taglib ディレクティブを追加します。

```
<%@ taglib uri="http://amateras.sf.jp/functions" prefix="f" %>

<%-- HTML エスケープし出力します。 --%>
${f:h(message)}
```

【ログ出力関数の使用例】

- ログ出力用の Taglib ディレクティブを追加します。

```
<%@ taglib uri="http://amateras.sf.jp/log4j" prefix="log4j" %>

<%-- info ログを出力する --%>
${log:info(message)}

${log:printDebug('デバッグログが有効な場合のみ表示されます。')}

<c:if test="${log:isDebugEnabled()}>
  デバッグログが有効な場合のみ表示されます。
</c:if>
```

※使用可能な関数の一覧は、下記の URL を参照してください。

- 「http://amateras.sourceforge.jp/cgi-bin/fswiki/wiki.cgi?page=JSEL」

6.5. Spring Expression Language(SpEL)

6.5.1. SpEL の言語仕様

表 6.8 SpEL のリテラル

No.	種類	例	説明
1	文字列	'Hello World'	シングルクオートで囲みます。
2	数値（整数）	1235 -1234	Java と同じです。
3	数値（小数）	132.3123 6.02323E+23	Java と同じです。
4	16 進数	0x7FFFFFF	Java と同じです。
5	ブール値	true false	Java と同じです。
6	NULL 値	null	Java と同じです。
7	リスト	{1,2,3,4} {{'a','b'},{'x','y'}}	括弧で囲み、カンマ「,」で区切ります。
8	配列	new int[4] new int[]{1,2,3} new int[4][5]	Java と同じです。
9	変数（コンテキスト変数も含む）	#name #command.name	接頭語「#」を付けます。

表 6.9 SpEL のプロパティへのアクセス方法

No.	種類	例	説明
1	ネストしたオブジェクト	command.name command.member.name	ピリオドで繋げプロパティ名を記述します。ただし、対応した getter メソッドが必要です。
2	ネストしたオブジェクト(Null の場合の回避)	command?.name command?.member?.name	親のオブジェクトが null の場合、NullPointerException を発生させるのを防ぐことができる。 親のオブジェクトが null の場合、参照結果も null となる。
3	リスト、配列	inventions[3]	括弧でインデックス(数値)を囲みます。

		members[0].name	
4	マップ	officers['president'] officers['president'].name	括弧でキー(文字列)を囲み指定します。 キーは文字列である必要があります。
5	セレクタ（リスト）	Members.?[Nationality == 'Serbian']	「?[式]」でフィルタリングして新しいリストを取得することができる。
6	セレクタ（マップ）	map.?[value<27]	「?[式]」でフィルタリングしてマップの値を取得することができる。
7	Spring Bean	@messageSource	Spring Bean への参照の場合、bean 名の先頭に「@」を付ける。

表 6.10 SpEL のメソッド、定数、キャストの呼び出し方法

No.	種類	例	説明
1	インスタンスメソッド	'abc'.substring(2, 3) isMember('Mihajlo Pupin')	Java と同じです。
2	static メソッド	T(java.lang.Math).abs(-100) T(Math).abs(-100) T(org.apache.commons.lang.StringUtils).isEmpty(#name)	クラス名に“T”を付けます。 パッケージ「java.lang」の場合はパッケージ名を省略できます。
3	static 変数	T(java.lang.Math).PI T(java.math.RoundingMode).CEILING	Static メソッドと同様に“T”を付けます。 パッケージ「java.lang」の場合はパッケージ名を省略できます。
4	キャスト	T(String) 10 (T(java.util.Date) #obj).time	キャストしたいクラスに“T”を付けます。 パッケージ「java.lang」の場合はパッケージ名を省略できます。
5	テンプレート	random number is #{T(java.lang.Math).random()} ⇒戻り値は「random number is 0.7038186818312008」	文字列の中に式があるような場合、明確に区別するために「#{式}」を使用する。 JSP 内での EL 式の使い方と同様。

表 6.11　SpEL の演算子

分類	No.	演算子	XML(※1)	説明
算術演算子	1	+		加算。
	2	-		減算。
	3	*		除算。
	4	/	div	除算。
	5	%	mod	剰余。
	6	^		べキ乗。
比較演算子	1	==	eq	等しい。
	2	!=	ne	等しくない。
	3	<	lt	小さい。
	4	>	gt	大きい。
	5	<=	le	以下。
	6	>=	ge	以上。
論理演算子	1	and		集合積(AND)。
	2	or		集合和(OR)。
	3	!	not	否定(NOT)。
三項演算子	1	a ? b : c		条件演算。 「a」の場合は「ｂ」、「a」以外の場合は「c」
その他	1	=		代入
	2	instanceof		インスタンスの比較。 例)「'xyz' instanceof T(int)」
	3	matches		正規表現による比較 例)「'5.00' matches '^-?¥¥d+(¥¥.¥¥d{2})?$'」

※1　Spring の設定ファイルの XML 内で使用する場合の使用方法。

6.5.2. SpEL の使用例

6.5.2.1. Spring の XML の設定ファイル

- SpEL を使用する場合は、「#{式}」で囲み、記述する。

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

    <!-- other properties -->
</bean>
```

- システムプロパティを呼び出したい場合は、「systemProperties['キー']」として呼び出す。

```
<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">
    <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

    <!-- other properties -->
</bean>
```

6.5.2.2. Spring のアノテーションで使用する

- XML の場合と同様、SpEL を使用する場合は、「#{式}」で囲み、記述する。

```
public static class FieldValueTestBean

   @Value("#{ systemProperties['user.region'] }")
   private String defaultLocale;

   public void setDefaultLocale(String defaultLocale) {
      this.defaultLocale = defaultLocale;
   }

   public String getDefaultLocale() {
      return this.defaultLocale;
   }
}
```

6.5.2.3. 入力値検証（OVal）で使用する

- アノテーションベースの入力値検証の際に、条件式を記述するために使用する。
- 詳細は、「7.4.10　条件付きチェック」「7.4.15　条件付きチェックに「SpEL」を使用する」を参照してください。

```
@Assert(expr="#_value != null and #_value.length() <= 5", lang="spel", message="error.name")
private String name;

@Range(min=1, max=200, when="spel:T(org.apache.commons.lang.StringUtils).isNotEmpty(#_this.name)")
private Integer age
```

7. 入力値検証

7.1. Errors クラスを使用した入力値検証

Spring MVC では、入力値検証の結果であるエラーメッセージを「org.springframework.validation.Errors」の実装クラス「BingingResult」に格納します。入力値検証結果のメッセージは「グローバルエラー」と「フィールドエラー」の 2 つに分かれています。

表 7.1　エラーメッセージの種類と入力値の検証方法

分類	入力値の検証方法（エラーメッセージの設定方法）	参照先
グローバルエラーメッセージ	項目間チェックや、Command 全体に対するエラーメッセージを設定します。 Errors の実装クラスに、手動でメッセージを格納します。 通常は、メソッド「Errors#reject("エラーコード", ["エラー引数"], ["デフォルトメッセージ"]);」を使用します。	7.1.1
フィールドエラーメッセージ	Validator インタフェースの実装をする。 値のチェックは手動で行い、Errors の実装クラス「BindingResult」にメッセージを格納します。 通常は、メソッド「Errors#rejectValue("フィールド名", "エラーコード", ["エラー引数"], ["デフォルトメッセージ"]);」を使用します。	7.2
	Bean Validation という API を使用し、Command(=Java Beans)の各プロパティに対してアノテーションを設定し、自動的にチェックを行います。 基本的に、単項目チェックのみ可能です。 Spring Framework 標準に組み込まれています。	7.3
	OVal という外部の API を使用し、Command(=JavaBeans)の各プロパティに対してアノテーションを設定し、自動的にチェックします。 アノテーションでは単項目チェックのみ可能ですが、Bean Validation よりも機能が豊富です。	7.4
	データバインド時のエラーもフィールドエラーメッセージに含みます。 データバインドエラーは、自動的にチェックされます。	4.2

7.1.1. Errors を使用した入力値検証のサンプル

【Controller の作成】

- Errors インタフェースの実装クラス「BindingResult」を引数に取ります。
  バインドエラーがある場合は、データ受信時に既にエラーメッセージが格納されています。
  バインドエラーについての詳細は、「4.2　データバインドエラー（型ミスマッチ）処理」を参照してください。
- フィールドエラーの場合、「Errors#rejectValue( )」を使用しメッセージを設定します。
  データ受信時のバインドエラーが既に設定されている可能性があるので、「Errors#hasFieldtErrors("プロパティ名")」を使用してこれからチェックするプロパティに対してエラーがないかどうかチェックします。
- グローバルエラーの場合、「Errors#reject( )」メソッド使用し、メッセージを設定します。また、引数がある場合も同様に与えることができます。
- フィールドエラーエラーがある場合、「Errors#hasFieldErrors("プロパティ名")」でチェックを行います。
  エラーがある場合、「BindingResult#getModel()」から Model 形式でエラーメッセージを抽出し、自画面へ全て移し替えて遷移します。

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.util.StringUtils;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MaxUploadSizeExceededException;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("/test/validate1")
public class Validate1Controller {

    // command の初期オブジェクトの取得
    @ModelAttribute("sampleCommand")
    public SampleCommand createInitCommand() {
        SampleCommand command = new SampleCommand();
        return command;
    }

    //  初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        SampleCommand command = createInitCommand();
        command.setAge(1);
        model.addAttribute("sampleCommand", command);

    }

    // post で送られた場合
```

バインディングエラーなどを正しく検知するために必要。

```
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(
            @ModelAttribute("sampleCommand") SampleCommand command,
            BindingResult bindingResult) {

        System.out.println(command);

        // フィールドエラーチェック(プロパティ=name)
        if(!bindingResult.hasFieldErrors("name")) {
            if(!StringUtils.hasLength(command.getName())) {
                bindingResult.rejectValue("name", "error.required");
            }
        }

        // フィールドエラーチェック(プロパティ=age)
        if(!bindingResult.hasFieldErrors("age")) {
            if(command.getAge() != null
                    && !(0 <= command.getAge() && command.getAge() <= 200)) {
                bindingResult.rejectValue("age", "error.range", new Object[]{0, 200}, null);
            }
        }
        // グローバルメッセージ
        if(bindingResult.hasErrors()) {
            bindingResult.reject("error.message");
        }

        // エラーメッセージがある場合、自画面遷移する。
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }

}
```

既にバインドエラーのメッセージがないかどうかチェックします。

入力値が空でないかどうか（=必須かどうか）チェックします。

フィールド名（=プロパティ名）を指定し、エラーコードを設定します。

フィールド名はなく、エラーコードから指定します。

エラーメッセージに引数がある場合、配列にて設定します。
デフォルトメッセージについて、通常は null で構いません。

エラーがある場合、エラーメッセージを Model にすべて移し替え、自画面へ遷移します。

【メッセージ用のプロパティファイルの作成】

```
# バインドエラー（型変換エラー）のエラーメッセージ
typeMismatch.int=整数で入力してください。
typeMismatch.java.lang.Integer=整数で入力してください。
typeMismatch=入力形式が不正です。

# フィールドエラーメッセージの定義
error.required=必須です。
error.range={0}から{1}の間の値を入力してください。

# グローバルエラーメッセージの定義
error.message=共通のエラーメッセージ。

# フィールド名の定義
name=名前
age=年齢
```

引数は、{インデックス}の形式で参照します。

【JSP の作成】

- エラーがあるかどか、カスタムタグ<spring:hasBindErrors>を使用し、エラーが含まれるかチェックします。コマンド名を属性「name」で指定することで、コマンドの定義が複数ある場合を区別します。
- エラーオブジェクトは、EL 式${error.XXX}で処理します。
  実体は、「org.springframework.validation.Errors」クラスであり、「Errors#getXXX()」メソッドは、${error.XXX}で呼び出すことができます。詳細は、「表 7.2　エラークラス「Errors」のメソッド」を参照してください。
- グローバルエラーメッセージは、${errors.globalErrors}で取得できます。エラーメッセージは、カスタムタグ<spring:message>で出力します。引数がある場合があるので、属性「arguments」を指定します。
  実体は、「org.springframework.validation.ObjectError」クラスです。詳細は、「表 7.3　グローバルエラークラス「ObjectError」のメソッド」を参照してください。
- フィールドエラーメッセージは、カスタムタグ<form:errors>で出力します。属性 path でフィールド名(=プロパティ名)を指定します。
  入力フィールドに対して、エラー時のみに埋め込む class 属性として、「cssErrorClass」で指定することができます。
- フィールドエラーメッセージは、グローバルメッセージと同様に、EL 式「${error.FieldErrors}」で取得することも可能です。実体は、「org.springframework.validation.FieldError」クラスです。詳細は、「表 7.4　フィールドエラークラス「FieldError」のメソッド」を参照してください。
  項目名を表示するには、<spring:message code="${error.field}"/>でプロパティファイルから取得します。

```
<%-- JSTL の定義 --%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>

<%-- Spring のカスタムタグの定義 --%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>

<spring:hasBindErrors name="sampleCommand">
    <%-- グローバルエラーメッセージの出力 --%>
    <c:if test="${errors.globalErrorCount > 0}">
    <div class="MessageBox erorr">
        <h4>グローバルエラー</h4>
        <ul>
            <c:forEach items="${errors.globalErrors}" var="error">
            <li><span class="error">
                <spring:message message="${error}"/>
            </span></li>
            </c:forEach>
        </ul>
    </div>
    </c:if>

    <%-- フィールドエラーメッセージの出力(項目名あり) --%>
    <c:if test="${errors.fieldErrorCount > 0}">
    <div class="MessageBox erorr">
```

グローバルエラーが存在するかチェックします。${errors}の実体は、「Errors」クラスなので、メソッドが呼び出せます。

グローバルエラーのリストを取り出す。

メッセージを表示します。

```
        <h4>フィールドエラー(項目名の埋め込み)</h4>
        <ul>
            <c:forEach items="${errors.fieldErrors}" var="error">
            <li><span class="error"><spring:message code="${error.field}"/>は、
<spring:message message="${error}"/></span></li>
            </c:forEach>
        </ul>
    </div>
    </c:if>
</spring:hasBindErrors>

<form:form modelAttribute="sampleCommand" action="${appUrl}/test/validate1.html" method="post">
    <p>
        <form:label path="name">名前</form:label>
        <form:input path="name" cssErrorClass="input_error"/>
        <form:errors path="name" cssClass="errors" />
    </p>
    <p>
        <form:label path="age">年齢</form:label>
        <form:input path="age" cssErrorClass="input_error"/>
        <form:errors path="age" cssClass="errors" />
    </p>
    <input type="submit"/>
</form:form>
```

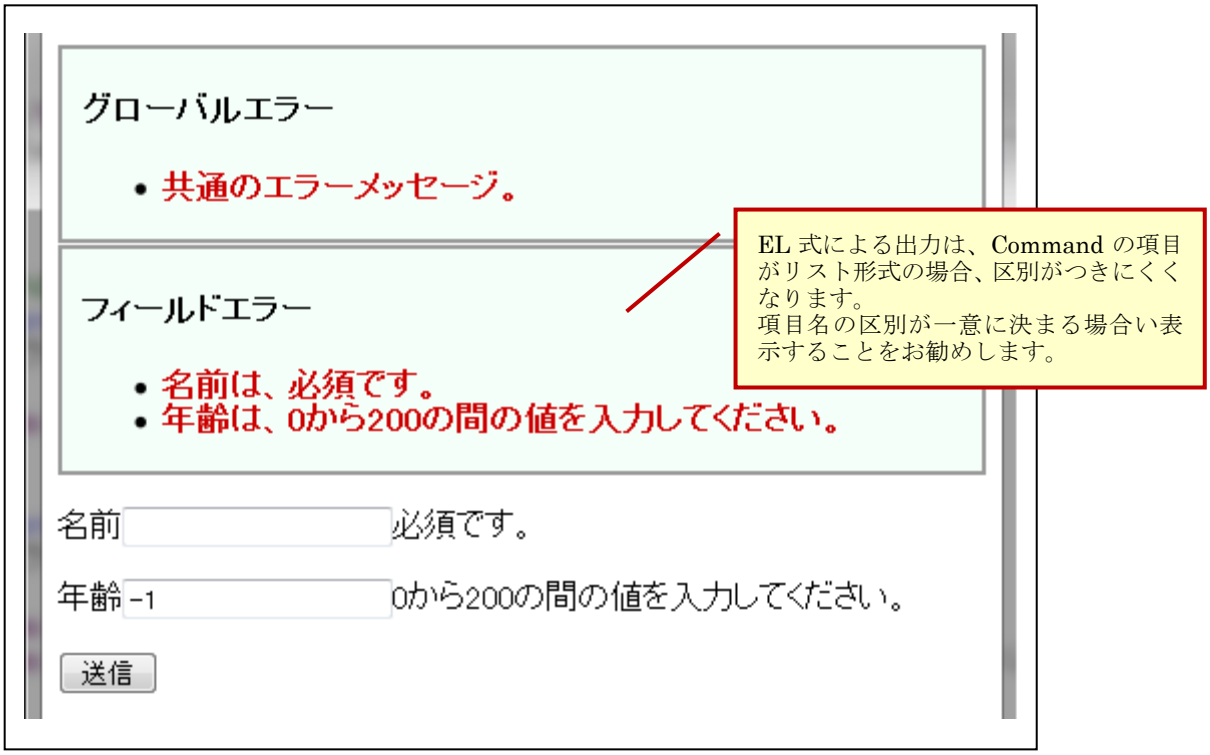
フィールドエラーを Errors クラスから呼ぶ。.

フィールドの項目名の取得。

エラー時に埋め込まれる class 属性の値です。

Spring のカスタムタグを使用し、フィールドエラーを出力します。

【ブラウザでの表示】



EL 式による出力は、Command の項目がリスト形式の場合、区別がつきにくくなります。
項目名の区別が一意に決まる場合い表示することをお勧めします。

【エラー時の HTML のソース】

- 入力フィールドの属性 class は、カスタムタグ cssErrorClass で定義した値であり、エラーがあるときのみ埋め込まれます。
- フィールドエラーのカスタムタグ<form:error>は、エラー時にはタグとして出力されます。エラーがない場合は、このタグは出力されません。

```
<div class="MessageBox erorr">
    <h4>グローバルエラー</h4>
    <ul>
        <li><span class="error">共通のエラーメッセージ。</span></li>
    </ul>
</div>

<div class="MessageBox erorr">
    <h4>フィールドエラー(項目名の埋め込み)</h4>
    <ul>
        <li><span class="error">名前は、必須です。</span></li>
        <li><span class="error">年齢は、0 から 200 の間の値を入力してください。</span></li>
    </ul>
</div>
<form id="sampleCommand" action="/spring3-mvc/test/validate1.html" method="post">
    <p>
        <label for="name">名前</label>
        <input id="name" name="name" class="input_error" type="text" value=""/>
        <span id="name.errors" class="errors">必須です。</span>
    </p>
    <p>
        <label for="age">年齢</label>
        <input id="age" name="age" class="input_error" type="text" value="-1"/>
        <span id="age.errors" class="errors">0 から 200 の間の値を入力してください。</span>
    </p>

    <input type="submit"/>
</form>
```

7.1.2. エラー時に使用するクラス

表 7.2　エラークラス「Errors」のメソッド

No.	メソッドの書式	説明
1	void reject(String erroCode)	グローバルエラーを追加します。
2	void reject(String errorCode, Object[] errorArgs, String defaultMessage)	引数を持つグローバルエラーを追加します。 errorArg、defaultMessage は null を設定可能です。
3	void reject(String errorCode, String defaultMessage)	デフォルトメッセージを持つグローバルエラーを追加します。
4	void rejectValue(String field, String errorCode)	フィールドエラーを追加します。 フィールド名を null または""(空文字)とした場合、現在のネストしているオブジェクト自身を指します。
5	void rejectValue(String filed, String errorCode, Object[] errorArgs, String defaultMessage)	引数を持つフィールドエラーを追加します。 フィールド名を null または""(空文字)とした場合、現在のネストしているオブジェクト自身を指します。 errorArg、defaultMessage は null を設定可能です。
6	void rejectValue(String filed, String errorCode, String defaultMessage)	デフォルトメッセージを持つフィールドエラーを追加します。 フィールド名を null または""(空文字)とした場合、現在のネストしているオブジェクト自身を指します。
7	void addAllErrors(Erros errors)	エラーオブジェクトを追加します。
8	boolean hasErrors( )	グローバルエラー、フィールドエラーが存在するかどうかチェックします。
9	List<ObjectError> getAllErrors( )	グローバルエラー、フィールドエラーを取得します。 [EL 式] ${errors.allErrors}
10	int getErrorCount( )	グローバルエラー、フィールドエラーの個数の合計値を取得します。 [EL 式] ${errors.errorCount}
11	ObjectError getGlocalError( )	１番初めに追加されたグローバルエラー(「表 7.3」を参照)オブジェクトを取得します。
12	boolean hasGlobalErrors( )	グローバルエラーが存在するかチェックします。
13	int GlobalErrorCount( )	グローバルエラーの個数を種痘します。 [EL 式] ${errors.globalErrorCount}
14	List<ObjectError> getGlobalErrors( )	グローバルエラー(「表 7.3」を参照)オブジェクトのリストを取得します。 [EL 式] ${errors.globalErrors}

15	boolean hasFiledErrors( )	フィールドエラーが存在するかチェックします。
16	boolean hasFieldErrors(String field)	指定したフィールド(=プロパティ名)のフィールドエラーが存在するかチェックします。
17	int getFieldErrorCount( )	フィールドエラーの個数を取得します。 [EL 式] ${errors.filedErrorCount}
18	int getFieldErrorCount(String field)	指定したフィールド(=プロパティ名)のフィールドエラーの個数を取得します。
19	FieldError getFieldError( )	１番初めに追加されたフィールドエラー（「表 7.4」を参照）オブジェクトを取得します。 [EL 式] ${errors.fieldError}
20	FieldError getFieldError(String field)	指定したフィールド名(＝プロパティ名)のフィールドエラーを取得します。
21	List<FieldError> getFieldErrors( )	フィールドエラーオブジェクト（「表 7.4」を参照）のリストを取得します。 [EL 式] ${errors.filedErrors}
22	List<FieldError>　getFieldErrors(String field)	指定したフィールド(=プロパティ名)のフィールドエラーオブジェクト（「表 7.4」を参照）のリストを取得します。
23	Class getFiledType(String field)	指定したフィールド(=プロパティ名)のクラス型を取得します。
24	Object getFiledValue(String field)	指定したフィールド(=プロパティ名)の値を取得します。
25	String getObjectName( )	ルートオブジェクト(=Command 名)を取得します。
26	String getNestedPath( )	現在のネストしているパス（=プロパティ名）を取得します。 「7.2.9　Validator による階層を持つ Command の入力値検証」を参照してください。
27	setNestedPath(String nestedPath)	現在位置からの下位階層の任意パスへ移動することができます。 「7.2.9　Validator による階層を持つ Command の入力値検証」を参照してください。
28	void pushNestedPath(String subPath)	引数で指定したパス（＝プロパティ名）にネストします。パスはスタック構造で管理し、メソッド「popNestedPath()」と合わせて使用します。 「7.2.9　Validator による階層を持つ Command の入力値検証」を参照してください。
29	void popNestedPath()	スタックで管理しているパスの１つ上の階層へ移動します。

表 7.3　グローバルエラークラス「ObjectError」のメソッド

No.	メソッドの書式	説明
1	String getCode( )	メッセージコードを取得する。 [EL 式] ${error.code}
2	Object[] getArguments( )	メッセージの引数を取得する。 設定されていない場合、null を返す。 [EL 式] ${error.arguments}
3	String getDefaultMessage( )	デフォルトメッセージを取得する。 設定されてい場合、null を返す。 [EL 式] ${error.defaultMessage}
4	String getObjectName( )	Command の名称を取得する。 [EL 式] ${error.objectName}

表 7.4　フィールドエラークラス「FieldError」のメソッド

No.	メソッドの書式	説明
1	String getField( )	フィールド名(=プロパティ名)を取得する。 [EL 式] ${error.field}
2	Object getRejectValue( )	フィールドの値を取得する。 [EL 式] ${error.rejectValue}
3	boolean isBindingFailure( )	データバインドエラーかどうか。 [EL 式] ${error.bindingFailure}
4	String getCode( )	メッセージコードを取得する。 [EL 式] ${error.code}
5	Object[] getArguments( )	メッセージの引数を取得する。 設定されていない場合、null を返す。 [EL 式] ${error.arguments}
6	String getDefaultMessage( )	デフォルトメッセージを取得する。 設定されてい場合、null を返す。 [EL 式] ${error.defaultMessage}
7	String getObjectName( )	Command の名称を取得する。 [EL 式] ${error.objectName}

※FieldError は、ObjectError（表 7.3　グローバルエラークラス「ObjectError」のメソッド）のサブクラスです。

7.2. Validator を実装した入力値検証

インタフェース「org.springframework.validation.Validator」を使用した入力値検証の方法を説明します。この方法は、Command（=JavaBeans）に対して値を検証します。チェックロジックを独自に実装することで次の長所・短所があります。

- 単項目チェックだけでなく、項目間チェックも実装できる。
- 数値の範囲チェックを行うときなど、値の範囲が外部ファイルなどに定義されている場合にも対応できる。
- ロジックを独自に実装するため、コーディング量が増えてしまう。
  このような場合は、「7.2.4　ポイント：フィールドを検証する際のユーティリティメソッド」にあるようなユーティリティメソッドを用意しコード量を減らすことができます。

7.2.1. Validator の基本

【Validator クラスの作成】

- メソッド「suppoerts()」にて、どの Command に対する Validator か定義します。
  メソッド「validate()」にて、入力値検証のロジックを実装します。
- 次の点に注意してください。
  - 一時変数などをプロパティに設定しないでください。
    Spring Bean に登録したり、ネストしている Bean に対してはインスタンスを使い回すためです。
  - どの Command に対する Validator かどうか、分かりり安い名前を付けてください。
    例）XXXCommand⇔XXXValidator

```
import org.springframework.util.StringUtils;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

public class Sample1Validator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Sample1Command.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        // Command へキャストする。
        Sample1Command command = (Sample1Command) target;

        // フィールドエラーチェック(プロパティ=name)
        if(!errors.hasFieldErrors("name")) {
            if(!StringUtils.hasLength(command.getName())) {
                errors.rejectValue("name", "error.required");
            }
        }

        // フィールドエラーチェック(プロパティ=age)
        if(!errors.hasFieldErrors("age")) {
```

どの Command に対する Validator かチェックします。

BindingResult のインタフェース。

フィールドに対して 1 つずつ独自にちぇク kしていきます。

バインドエラーなど、既にチェック対象のフィールドに対してエラーがあればスキップします。

```
            if(command.getAge() != null
                    && !(0 <= command.getAge() && command.getAge() <= 200)) {
                errors.rejectValue("age", "error.range", new Object[]{0, 200}, null);
            }
        }

    }
}
```

【チェック対象の Command】

```
import java.io.Serializable;
import org.apache.commons.lang.builder.ToStringBuilder;

public class Sample1Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private String name;

    private Integer age;

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
    // setter、getter は省略
}
```

【Controller からの呼び出し】

- 作成した Validator「Sample1Validator」のインスタンスを作成し、メソッド「validate()」を実行します。その際に、引数として BindingResult を渡します。
- Validator にてエラーが設定されているかどうか、BindingResult#hasErrors()にてチェックを行い、エラーがあれば Model に値を全て移し替え、自画面へ遷移します。

```
@Controller
@RequestMapping("/test/sample1")
public class Sample1Constroller {

    @ModelAttribute("sample1Command")
    public Sample1Command createInitCommand() {
        Sample1Command command = new Sample1Command();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        Sample1Command command = createInitCommand();
        command.setAge(0);
        model.addAttribute("sample1Command", command);

    }

    @RequestMapping(method=RequestMethod.POST)
```

```
    public ModelAndView doAction1(@ModelAttribute("sample1Command") Sample1Command command,
            BindingResult bindingResult) {

        // 入力値チェックの実行
        Sample1Validator validator = new Sample1Validator();
        validator.validate(command, bindingResult);

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

Validator を呼び出します。

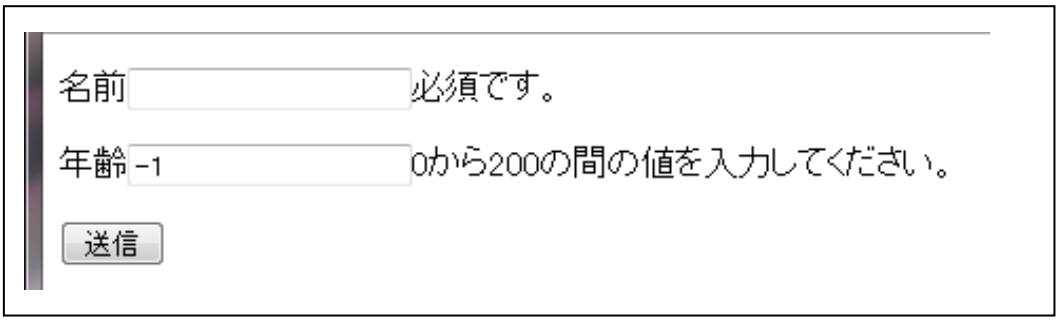
Validator にてチェックがある場合、Model にエラーデータを移し替えた後、自画面遷移します。

【エラーメッセージの定義】

- エラーコードに対するメッセージは、Spring の messageSource として読み込むプロパティファイルに定義します。
  定義方法などは、「2.4.3　アプリケーション用(共通の)Spring Bean ファイル」を参照してください。
- メッセージに置換文字を使用する場合「{インデックス}」とします。
  インデックスはエラーメッセージにの引数配列のインデックスと一致します。

```
error.required=必須です。
error.range={0}から{1}の間の値を入力してください。
```

【Web ブラウザでの表示】



7.2.2. ポイント：Validator を Spring Bean として扱う

Validator を Spring Bean として登録することで、Validator の中で F 層やプロパティファイルなど簡単に呼び出すことができます。

【Validator の作成】

- アノテーション「@Component」をクラスに付与し、Validator を Spring Bean として登録します。このアノテーションは、「@Service」と機能としては区別ありません。
- MessageSource など Spring Bean をインジェクションし使用することができます。

```
import javax.annotation.Resource;

import org.springframework.context.MessageSource;
import org.springframework.context.support.MessageSourceAccessor;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

/**
 * Spring Bean に登録する Validator
 */
@Component
public class Sample2Validator implements Validator {

    @Resource
    private MessageSource messageSource;

    @Override
    public boolean supports(Class<?> clazz) {
        return Sample1Command.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        // Command へキャストする
        Sample1Command command = (Sample1Command) target;

        // フィールドエラーチェック(プロパティ=name)
        if(!errors.hasFieldErrors("name")) {
            if(!StringUtils.hasLength(command.getName())) {
                errors.rejectValue("name", "error.required");
            }
        }

        // プロパティファイルからチェック範囲の値を取得する
        MessageSourceAccessor messageAccessor = new MessageSourceAccessor(messageSource);
        int ageMin = Integer.parseInt(messageAccessor.getMessage("app.age.min"));
        int ageMax = Integer.parseInt(messageAccessor.getMessage("app.age.max"));

        // フィールドエラーチェック(プロパティ=age)
        if(!errors.hasFieldErrors("age")) {
            if(command.getAge() != null
                    && !(ageMin <= command.getAge() && command.getAge() <= ageMax)) {
```

Spring Bean をインジェクションすることができる。

プロパティファイルから、値を取得できる。

```
                errors.rejectValue("age", "error.range", new Object[]{ageMin, ageMax}, null);
            }
        }
    }
}
```

※ポイント：Commons Configuration を使用すると、キャストなど省略することができます。

使用方などは、「14.3　ライブラリ「Commons Configuration」を使用する」を参照してください。

```
@Component
public class Sample2Validator implements Validator {

    @Resource(name="appConfiguration")
    private Configuration appConfiguration;

    @Override
    public void validate(Object target, Errors errors) {
        // Command へキャストする
        Sample1Command command = (Sample1Command) target;

        ・・・省略

        // プロパティファイルからチェック範囲の値を取得する
        //MessageSourceAccessor messageAccessor = new MessageSourceAccessor(messageSource);
        //int ageMin = Integer.parseInt(messageAccessor.getMessage("app.age.min"));
        //int ageMax = Integer.parseInt(messageAccessor.getMessage("app.age.max"));

        int ageMin = appConfiguration.getInt("app.age.min");
        int ageMax = appConfiguration.getInt("app.age.max");

        // フィールドエラーチェック(プロパティ=age)
        if(!errors.hasFieldErrors("age")) {
            if(command.getAge() != null
                    && !(ageMin <= command.getAge() && command.getAge() <= ageMax)) {
                errors.rejectValue("age", "error.range", new Object[]{ageMin, ageMax}, null);
            }
        }
    }
}
```

CommonsConfiguration 経由で取得すると、キャストを省略できる。

【Controller からの呼び出し】

- Validator は、インジェクションし使用します。

```
@Controller
@RequestMapping("/test/sample1")
public class Sample1Constroller {

    @Resource
    private Sample2Validator sample2Validator;

    ・・・省略

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(@ModelAttribute("sample1Command") Sample1Command command,
            BindingResult bindingResult) {

        // 入力値チェックの実行
        sample2Validator.validate(command, bindingResult);

        // エラーがある場合、自面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

Validator をインジェクションします。

7.2.3. ポイント：抽象クラスによりキャスト処理を省略する

Generics を使用することで、「Validator#supports()」メソッドを省略することができます。

- Gnerics を使用することで、実装クラスにおいて、Command の型チェックと、キャストを省略することができます。
- プロジェクトで決まっている共通処理などがあれば、抽象クラスにて定義しておくこともできます。

【抽象クラス】

```
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

/**
 * Validator の抽象クラス
 * @param <T>  チェック対象の Command クラス
 */
public abstract class AbstractValidator<T> implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return true;
    }

    @SuppressWarnings("unchecked")
    @Override
    public void validate(Object target, Errors errors) {
        validateCommand((T) target, errors);
    }

    protected abstract void validateCommand(T command, Errors errors);
}
```

Generics により、Command のクラスか型は判明しているので、処理を省略するために true を返しても問題なし。

Generics により、Command のクラス型が判明しているので、キャストする。

【実装クラス】

```
public class LoginValidator extends AbstractValidator<LoginCommand> {

    @Override
    protected void validateCommand(LoginCommand command, Errors errors) {
        if(errors.hasFieldErrors("userCd")) {
            // バインドエラーがある場合

        } else if(!StringUtils.hasLength(command.getUserCd())) {
            errors.rejectValue("userCd", "error.required");
        }

        if(errors.hasFieldErrors("password")) {
            // バインドエラーがある場合

        } else if(!StringUtils.hasLength(command.getPassword())) {
            errors.rejectValue("password", "error.required");
        }
    }
}
```

キャスト済みの Command で受け取ることができる。

7.2.4. ポイント：フィールドを検証する際のユーティリティメソッド

7.2.4.1. ValidationUtils を使う

Sprng には、「org.springframework.validation.ValidationUtils」値が空白かなどをチェックするクラスが予め用意されています。基本的に、Command のフィールド（=プロパティ）の値をチェックするために使用します。

表 7.5　ValidationUtils のメソッド

No.	メソッドの書式	説明
1	static void rejectIfEmpty(Errors errors, String field, String errorCode)	フィールドの値が、null、""(空)の場合エラーとします。 エラーメッセージに引数を設定する場合などケースにより使い分けます。
2	static void rejectIfEmpty(Errors errors, String field, String errorCode, Object[] errorArgs)	
3	static void rejectIfEmpty(Errors errors, String field, String errorCode, Object[] errorArgs, String defaultMessage)	
4	static void rejectIfEmpty(Errors errors, String field, String errorCode, String defaultMessage)	
5	static void rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode)	フィールドの値が、null、""(空)、空白スペースの場合にエラーとします。 エラーメッセージに引数を設定する場合などケースにより使い分けます
6	static void rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode, Object[] errorArgs)	
7	static void rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode, Object[] errorArgs, String defaultMessage)	
8	static void rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode, String defaultMessage)	

7.2.4.2.　独自のユーティリティメソッド

「7.2.4.1　ValidationUtils を使う」を参考に作っていきます。単項目チェックをユーティリティメソッドで簡単に実装・呼び出しを行うようにします。

- このようなクラスを作るくらいならば、アノテーションを利用した Bean Validation（7.3 節参照）、OVal（7.4 節参照）を使用した方がよいかと思うかもしれませんが、チェック条件が複雑になり、また Command は共通だがチェック条件が Controller ごとに異なる場合は対応が難しくなります。
- アノテーション方式の場合、チェックの種類を追加する場合、アノテーションクラスとチェッククラスを作成しなければならず、少々面倒になります。

【使用例】

- 条件演算子「&&」で複数のチェック処理を繋げて呼び出します。
  - “**定義した順**”にチェックし、どれか 1 つでもチェック結果が不正な場合、そこでチェック処理が終了します。

```
public class MainSearchValidator implements Validator {

    private ISampleDao sampleDao;

    protected void validate(final Object target, final Errors errors) {
        MainSearchCommand command = (MainSearchCommand) ta

        final String keywordName = "keyword";
        final String keywordValue = command.getKeyword();
        final boolean keywordValid = FieldValidationUtils.validateRequired(errors, keywordName, keywordValue)
                && FieldValidationUtils.validateMaxLength(errors, keywordName, keywordValue, 30)
                && FieldValidationUtils.validatePattern(errors, keywordName, keywordValue, "[a-zA-Z]*");

        final String ageName = "age";
        final long ageValue = command.getAge();
        boolean ageValid = FieldValidationUtils.validateRange(errors, ageName, ageValue, 0, 100)
                && validateXXX(errors, ageName, ageValue);

        if(keywordValid && ageValid) {
            // keyword と age にエラーがない場合
            //TODO: 項目間のチェック
        }

        /* エラーがあるかどうかの判定は同じ
        if(errors.hasFieldErrors(keywordName) && errors.hasFieldErrors(ageName)) {
            // keyword と age にエラーがない場合
            //TODO: 項目間のチェック
        }*/
    }

    // DAO などを呼び、ユーティリティメソッドではまとめられない、Command 独自のチェック
    boolean validateXXX(final Errors errors, final String field, final String value,) {
        //TODO: ユーティリティメソッドと仕様は変わません。
        return false;
    }
}
```

条件演算子「&&」で繋げていきます。
どれかのチェックでエラー（戻り値が false）となると、他のチェックは実行されません。

チェック結果を変数に取っておくことで、項目間チェック実行の判定をやりやすくします。
ただし、下記の「Errors#hasFieldErrors(...)」を利用してもあまりコーディング量は変わりません。

DAO などを呼び出したりして、共通性のない独自のチェック処理を行う場合。

【検証用クラス「FieldValidationUtils.java」の実装】

- 各メソッドは static にして簡単に呼び出せるようにします。
  - DAO を呼び出すようなチェック処理を必要とする場合は、チェック対象の Command 固有の処理のケースが多いため、Validator クラスで実装します。
- 各チェック用のメソッドの“引数”は、エラー格納用の「Errors」クラス、「フィールド名」、「コマンドから取得したフィールドの値」を共通して持ちます。
  - バインドエラーがある場合は、コマンドから取得した値は null となるので注意が必要です。
- 各チェック用のメソッドの“戻り値”は、boolean を返します。チェックが不正な場合 false を返します。
  - バインドエラーや、他のチェックにより既にエラーがある場合も、false を返します。
  - フィールドの値が null の場合、true を返し処理をスキップます。ただし、必須チェックの場合は例外です。
- チェックした際のエラーコードに対応するメッセージは、メッセージプロパティに定義しておいてください。

```
import org.springframework.util.Assert;
import org.springframework.validation.Errors;
public class FieldValidationUtils {

    /**
     * 値が入力されているかどうかチェックする。
     * <p> 文字列の場合は、半角空白の場合もチェックする。
     * <p> 既に指定したフィールドに対するエラーがある場合はスキップする（true を返す）
     * <p>エラーコード：error.required
     */
    public static boolean validateRequired(final Errors errors, final String field, final Object value) {
        Assert.notNull(errors, "Errors object must not be null");
        if(errors.hasFieldErrors(field)) {
            // 既にフィールドに対してエラーがある場合。
            return false;
        }

        if(value == null) {
            // 値がない場合
            errors.rejectValue(field, "error.required");
            return false;
        }

        if(value.getClass().isAssignableFrom(String.class)) {
            // 文字列の場合、空白のみかチェックする。
            final String str = (String) value;
            if(str.trim().isEmpty()) {
                errors.rejectValue(field, "error.required");
                return false;
            }
        }
        return true;

    }
```

バインド時や他のチェックでエラーとなっている場合がるので、その場合は処理を終了します。
ただし、エラーが既にあるということで、false を返します。

必須チェックなので、値が空の場合エラーとします。

String 型の場合、空文字かどうかもチェックします。
空白のみの場合もエラーとするのかは、プロジェクトごとに決めてください。

```
    /**
     * 文字列の長さが、指定した値よりも小さいかどうかチェックする。
     * <p>エラーコード：error.maxLength
     * @param maxLength 最大長
     * @return
     */
    public static boolean validateMaxLength(
            final Errors errors, final String field, final String value, final int maxLength) {

        Assert.notNull(errors, "Errors object must not be null");
        if(errors.hasFieldErrors(field)) {
            // 既にフィールドに対してエラーがある場合。
            return false;
        }

        if(value == null) {
            return true;
        }

        final int length = value.length();
        if(length > maxLength) {
            errors.rejectValue(field, "error.maxLength", new Object[]{maxLength},
                    "validateMaxLength:length={0}");
            return false;
        }

        return true;
    }

    /**
     * 値が指定した値の範囲内かチェックする。
     * @param min 最小値
     * @param max 最大値
     * @return
     */
    public static boolean validateRange(
            final Errors errors, final String field, final Long value, final long min, final long max) {

        Assert.notNull(errors, "Errors object must not be null");

        if(errors.hasFieldErrors(field)) {
            // 既にフィールドに対してエラーがある場合。
            return false;
        }

        if(value == null) {
            return true;
        }

        if(value.compareTo(min) > 0 || value.compareTo(max) < 0) {
            errors.rejectValue(field, "error.range", new Object[]{min, max},
                    "validateRange:min={0}, max={1}");
            return false;
        }

        return true;
    }
}
```

値がない場合は、処理をスキップします。
値が必須かどうかは他のメソッド「validateRequired(…)」で行い、また必須でない場合も正しいときもあるためです。

7.2.5. ポイント：フィールド用 Validator を作成し検証する

- フィールド用の Validator を独自に用意し、単項目チェックを簡単にできるようにします。イメージとしては、コンポーネント志向の P 層フレームワーク「Apache Wicket」のように、Model に対して Validator を複数追加できるようにします。
- このようなクラスを作るくらいならば、OVal（7.4 節参照）を使用した方がよいと思うかもしれませんが、チェック条件が複雑になった場合など、OVal の条件式が複雑になりソースの可読性が下がります。また、業務用 APP では編集条件や入力値チェック自体が複雑なパターンになり表現しきれなくなります。さらに、Command のフィールドにアノテーションで直接設定するので、Controller によって Validator そのものを切り替えたい場合に実現不可能になります。

【フィールドの検証のサンプル】

```
public class MainSearchValidator implements Validator {

    @Override
    protected void validate(final Object target, final Errors errors) {

        MainSearchCommand command = (MainSearchCommand) target;

        final Field<String> nameField = new Field<String>("name", command.getName())
            .setRequired(true)
            .add(StringValidator.maxLength(30))
            .add(new PatternValidator("[a-zA-Z¥s]"));
        nameField.validate(errors);

        final Field<Integer> ageField = new Field<String>("age", command.getAge())
            .setRequired(false)
        ageField.validate(errors);

        if(nameField.hasNotErrors(errors) && ageField.hasNotErrors(errors)) {
            //TODO: 項目間チェックの実装
        }
    }
}
```

メソッドチェーンで、どのような検証を行うかを組み立てます。

Errors クラスのメソッドを委譲し、項目間チェック実行の判定をやりやすくします。

表 7.6　フィールド検証のために作成するクラス一覧

No.	クラス名	説明	参照
1	IFieldValidator.java	フィールド用の Validator のインタフェース。	7.2.5.1
2	AbstractFieldValidator.java	フィールド用の Validator の抽象クラス。フィールド用の Validator を作成する際には、このクラスを継承します。	7.2.5.2
3	Field.java	Command 中のフィールドの値を保持し、フィールド用の Validator を実行します。	7.2.5.3
4	SringValidator.java	文字列長を検証するためのフィールド Validator のサンプル。	7.2.5.4
5	MinValidator.java	数値などを検証するためのフィールド Validator のサンプル。	7.2.5.5

7.2.5.1.　「IFieldValidator.java」の実装

- フィールドを検証する Validator のインタフェースで。
- Generics として、フィールドの値のクラスタイプを指定します。
- この実装クラスは、Field クラス内で呼ばれます。

```
import org.springframework.validation.Errors;

/**
 * フィールドバリデータのインタフェース。
 * @param <T> フィールドのタイプ
 */
public interface IFieldValidator<T> {

	/**
	 * フィールドの入力値チェックを行う。
	 * @param fieldName フィールド名
	 * @param value フィールドの値
	 * @param errors 入力値チェックした結果
	 * @return true: チェックを実行した結果、エラーがない場合。
	 */
	public boolean validate(String fieldName, T value, Errors errors);

}
```

7.2.5.2.　「AbstractFieldValidator.java」の実装

- フィールドを検証する Validator の抽象クラスで、実際の Validator を実装する際には、このクラスを継承し作成していきます。
- IFieldValidator.java と同様に、Generics のタイプは、フィールドの値のクラスタイプを指定します。

```
/**
 * フィールド Validator の抽象クラス。
 * @param <T> チェック対象のフィールドのタイプ
 */
public abstract class AbstractFieldValidator<T> implements IFieldValidator<T>{

	/**
	 * 値が null かどうか判定を行う。
	 * @param value
	 * @return
	 */
	public boolean isNullValue(T value) {
		return (value == null);
	}

	/**
	 * 値が null 出ないかどうか判定を行う。
	 * @return
	 */
	public boolean isNotNullValue(T value) {
		return !isNullValue(value);
	}

	/**
	 * 入力値エラー時のメッセージキーを取得する。
```

```
     * @return
     */
    public abstract String getMessageKey();

}
```

7.2.5.3.　「Field.java」の実装

- メソッド「validate()」にてチェックを実行します。その際、必須チェックは Field クラスで実装します。
  - 通常、このような Validator を実装する際には、型チェックを行う必要があり、“文字列⇔データ型のオブジェクト”に相互に変換する Converter が必要になります。しかし、型チェックは Command に渡す前に、Spring MVC がデータバインド時に行っているため省略できます。
- チェックを実行する場合、チェック対象のフィールドで既に他のエラーがあるときは処理をスキップします。バインドエラーの場合、Command の値は null だが Errors クラスの中にエラーメッセージが格納されているので、Errors#hasFieldErrors()メソッドで判定します。
- 各種 setter メソッドは、メソッドチェーンで使いやすくするために、自身のインスタンスを返すようにします。

```
import java.util.ArrayList;
import java.util.List;

import org.springframework.validation.Errors;

/**
 * 1 つの項目（フィールド）に対する入力値チェックをするためのクラス。
 * <p>Spring のデータバインドを併用することを前提としているため、型ミスマッチエラーなどは Spring 側で行う。
 *
 * @param <T>  チェック対象の値のタイプ
 */
public class Field<T> {

    /**  フィールド名（チェック対象のプロパティ名）  */
    final private String name;

    /**  チェック対象の値  */
    private T value;

    /**  必須かどうか  */
    private boolean required;

    /**  フィールド Validator */
    private List<IFieldValidator<T>> validators;

    /**
     *  コンストラクタ
     * @param name  フィールドの名称
     * @param value  フィールドの値
     */
    public Field(final String name, final T value) {
        this.name = name;
        setValue(value);
        this.validators = new ArrayList<IFieldValidator<T>>();
    }
```

Command のフィールドの情報。
コンストラクタで指定する。

フィールドバリデータのインスタンス。
追加された順にチェックしていく。

```
/**
 * Validator を追加する。
 * @param validator
 * @return
 */
public Field<T> add(IValidator<T> validator) {
	if(validator == null) {
		throw new IllegalArgumentException("validator is null.");
	}

	validators.add(validator);
	return this;
}

/**
 * 入力値チェックを行う。
 * <p>既に、引数の errors の中に自身に関するエラーがある場合は無視する。
 * <p>チェック順は、(1)必須チェック、(2)追加した FieldValidator の順。
 * @param errors
 */
public Field<T> validate(Errors errors) {
	if(errors == null) {
		throw new IllegalArgumentException("errors is null.");
	}

	if(errors.hasFieldErrors(getName())) {
		// 既にフィールドに対するエラーがある場合
		return this;
	}

	if(!validateAsRequired(errors)) {
		// 必須エラーの場合
		return this;
	}

	if(validators != null && !validators.isEmpty()) {
		// 各種入力値チェックを行う。
		for(IFieldValidator<T> validator : validators) {
			if(!validator.validate(getName(), getValue(), errors)) {
				// エラーがある場合
				return this;
			}
		}
	}

	return this;
}

/**
 * 必須チェックを行う。
 * @param errors
 * @return true:エラーがない場合。既にエラーがある場合。
 */
protected boolean validateAsRequired(Errors errors) {
	if(getValue() == null || getValue().toString().isEmpty()) {
		// 必須エラーチェックを行う場合
		if(isRequired()) {
			errors.rejectValue(getName(), "error.required");
			return false;
		}
```

既にフィールドに対して、他のエラーがある場合は、処理を中断します。

必須チェックを行います。
必須チェックを行うかどうかは、setRequired(boolean)で設定します。

追加された各種フィールド Validator を実行します。エラーがある時点で処理を中断します。

```
                return true;
            }

            // エラーがない場合
            return true;

        }

        public List<IFieldValidator<T>> getValidators() {
            return validators;
        }

        public Field<T> setValue(final T value) {
            this.value = value;
            return this;
        }

        public String getName() {
            return name;
        }

        public T getValue() {
            return value;
        }

        public Field<T> setRequired(final boolean required) {
            this.required = required;
            return this;
        }

        public boolean isRequired() {
            return required;
        }

        /**
         * 自身のフィールドに対してフィールドエラーを持つかどうか。
         * <p>{@link Errors#hasFieldErrors(String)}を呼び出す。
         * @param errors
         */
        public boolean hasErrors(Errors errors) {
            return errors.hasFieldErrors(getName());
        }

        /**
         * 自身のフィールドに対してエラーを持たないかどうか
         * @param errors
         */
        public boolean hasNotErrors(Errors errors) {
            return !hasErrors(errors);
        }
}
```

各種インスタンス変数に対する setter／getter。
setter メソッドは、自身のインスタンスを返すことで、メソッドチェーンによる設定を実現します。

Errors クラスのメソッドの委譲です。
フィールド名の指定を省いて指定など、スペルミスなどを防ぐことができます。

7.2.5.4.　「StringValidator.java」の実装

- 文字列に関するチェックは、最小文字長、最大文字長、範囲など通常では複数あるので、内部クラスとしてまとめます。
- Field クラス内で呼び出されるロジック処理を、「validate()」メソッドで実装します。
- 内部クラスで実装しているので、それらのインスタンスを作成するメソッドを static メソッドで作成します。この辺りの構造は、各自の好みなので、特に内部クラスで実装する必要もありません。

```
import org.springframework.validation.Errors;

public abstract class StringValidator extends AbstractFieldValidator<String>{

    /**
     * 文字列が指定した文字長以内かどうかチェックする。
     */
    public static class MaxLengthValidator extends StringValidator {

        private final int maxLength;

        public MaxLengthValidator(final int maxLength) {
            this.maxLength = maxLength;
        }

        @Override
        public boolean validate(final String fieldName, final String value, Errors errors) {
            if(isNullValue(value)) {
                return true;
            }

            if(value.length() <= getMaxLength()) {
                return true;
            }

            errors.rejectValue(fieldName, getMessageKey(), new Object[]{getMaxLength()},
                    "StringValidator.MaxLengthValidator:maxLength={0}");

            return false;
        }

        @Override
        public String getMessageKey() {
            return "error.maxLength";
        }

        public int getMaxLength() {
            return maxLength;
        }

    }

    /**
     * 文字列が指定した文字長以上かどうかチェックする。
     */
    public static class MinLengthValidator extends StringValidator {
```

文字列に関するチェックなので、Generics のタイプは、“String”にします。

コンストラクタで、最大文字長の値を設定します。

値が空の場合はスキップします。
必須チェックは Field クラスで行います。

Errors#rejectValue()メソッドを呼び出し、エラーメッセージを作成します。

```
        /** 最小文字長 */
        private final int minLength;

        public MinLengthValidator(final int minLength) {
            this.minLength = minLength;
        }

        @Override
        public boolean validate(final String fieldName, final String value, Errors errors) {
            if(isNullValue(value)) {
                return true;
            }

            if(value.length() >= getMinLength()) {
                return true;
            }

            errors.rejectValue(fieldName, getMessageKey(), new Object[]{getMinLength()},
                    "StringValidator.MaxLengthValidator:minLength={0}");

            return false;
        }

        @Override
        public String getMessageKey() {
            return "error.minLength";
        }

        public int getMinLength() {
            return minLength;
        }

    }

    /**
     * 文字列が指定した文字長の範囲内かどうかチェックする。
     */
    public static class BetweenLengthValidator extends StringValidator {

        private final int minLength;

        private final int maxLength;

        public BetweenLengthValidator(final int minLength, final int maxLength) {
            this.minLength = minLength;
            this.maxLength = maxLength;
        }

        @Override
        public boolean validate(final String fieldName, final String value, Errors errors) {
            if(isNullValue(value)) {
                return true;
            }

            final int strLength = value.length();
            if(getMinLength() <= strLength && strLength <= getMaxLength()) {
                return true;
            }

            errors.rejectValue(fieldName, getMessageKey(), new Object[]{getMinLength(), getMaxLength()},
                    "StringValidatorBetweenLengthValidator:minLength={0}, maxLength={1}");
```

```
            return false;
        }

        @Override
        public String getMessageKey() {
            return "error.betweenLength";
        }

        public int getMinLength() {
            return minLength;
        }

        public int getMaxLength() {
            return maxLength;
        }

    }

    /**
     * 文字長が指定した文字長以下かチェックする Validator を取得する。
     * @param maxLength
     * @return
     */
    public static StringValidator maxLength(final int maxLength) {
        return    new MaxLengthValidator(maxLength);
    }

    /**
     * 文字長が指定した文字長以上かチェックする Validator を取得する。
     * @param minLength
     * @return
     */
    public static StringValidator minLength(final int minLength) {
        return    new MinLengthValidator(minLength);
    }

    /**
     * 文字長が指定した文字長の範囲内かかチェックする Validator を取得する。
     * @param maxLength
     * @param minLength
     * @return
     */
    public static StringValidator betweenLength(final int minLength, final int maxLength) {
        return    new BetweenLengthValidator(minLength, maxLength);
    }
}
```

各種 validator のインスタンスを取得する static メソッド。

7.2.5.5.　「MinValidator.java」の実装

- 数値の Integer、Long などは Number クラスを親に持つため、Generics のタイプを Number にし、StringValidator のように最大値、最小値、範囲のチェックと内部クラスとしてまとめたくなります。
  - しかし、Number クラスを Generics のタイプにすると、Field#getValue()で取得する型も Number になり後々不便になります。そこで、少し汎用的に、「Comparable」を Generics のタイプとします。
  - Comparable を使用することで、Number の子クラス以外の Date クラスでも比較することができます。実際には、エラーメッセージが不自然になるため、DateValidator を別途作成した方が無難かもしれません。

```
import org.springframework.validation.Errors;

public class MinValidator<T extends Comparable<T>> extends AbstractFieldValidator<T> {

    private final T min;

    public MinValidator(T min) {
        this.min = min;
    }

    @Override
    public String getMessageKey() {
        return "error.min";
    }

    @Override
    public boolean validate(final String fieldName, final T value, Errors errors) {
        if(isNullValue(value)) {
            return true;
        }

        if(value.compareTo(getMin()) >= 0) {
            return true;
        }

        errors.rejectValue(fieldName, "error.min", new Object[]{getMin()},
                "MinValidator:min={0}");

        return false;
    }

    public T getMin() {
        return min;
    }

}
```

インタフェース“Comparable”を実装したクラスを対象にするが、実際のタイプはインスタンス生成時に決められるようにする。

Comparable#compareTo()　のメソッド使用し値をチェックします。

7.2.6. ポイント：@Valid を使用した Validator の呼び出し

アノテーション「@Valid」は、Bean Validation の API に含まれるので、設定方法は「7.3.1　Bean Validation の準備」とほぼ同じです。

【pom.xml の編集】

- 依存ライブラリとして、Bean Validation を追加します。
  Bean Validation の実装の 1 つである、Hibernate Validator も追加します。

```
<project>
    ・・・省略
    <dependencies>
        <!-- validator -->
        <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
            <version>1.0.0.GA</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
            <version>4.2.0.Final</version>
        </dependency>
    </dependencies>
    ・・・省略
</project>
```

【Controller 側の設定】

- Command と Validator を関連付けるために、WebDataBinder#setValidator()にて、Validator を登録します。
  この例では、Validator は Spring Bean として登録したものですが、「new Sample2Validator()」のように直接インスタンスを作成し格納してもかまいません。
- Command を受け取るメソッドにおいて、アノテーション「@Valid」を付加します。
  データバインド時に自動的に登録した Validator が呼び出されます。

```
import javax.annotation.Resource;
import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("/test/sample2")
public class Sample2Constroller {
```

```
    @Resource
    private Sample2Validator sample2Validator;

    @InitBinder("sample2Command")
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(sample2Validator);
    }

    @ModelAttribute("sample2Command")
    public Sample1Command createInitCommand() {
        Sample1Command command = new Sample1Command();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        Sample1Command command = createInitCommand();
        command.setAge(0);

        model.addAttribute("sample2Command", command);

    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample2Command") @Valid Sample1Command command,
            BindingResult bindingResult) {

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;

    }

}
```

WebDataBinder#setValidator()にて、Validatorのインスタンスを登録します。

Command に、「@Valid」を付与します。

7.2.7. リストを項目とする Command の入力値検証

【Command の作成】

- フィールド(=プロパティ)「books」は、List<String>形式のデータ型です。

```
public class Sample3Command implements Serializable {

	/** serialVersionUID */
	private static final long serialVersionUID = 1L;

	private String name;

	private List<String> books;

	@SuppressWarnings("unchecked")
	public Sample3Command() {
		books = ListUtils.lazyList(
				new ArrayList<String>(),
				FactoryUtils.instantiateFactory(String.class));
	}

	@Override
	public String toString() {
		return ToStringBuilder.reflectionToString(this);
	}
	// getter、setter は省略
}
```

【JSP の作成】

- プロパティ books はリスト型なので、<form:XXX path="books[インデックス]">の書式で記述します。

```
<h4>入力値検証：リスト形式のデータ</h4>
<form:form modelAttribute="sample3Command" action="${appUrl}/test/sample3.html" method="post">
	<p>
		<form:label path="name">名前</form:label>
		<form:input path="name" />
		<form:errors path="name" cssClass="errors" />
	</p>
	<ul>
		<c:forEach items="${sample3Command.books}" var="book" varStatus="bookStatus">
		<li>
			<form:label path="books[${bookStatus.index}]">本（${bookStatus.index+1}）</form:label>
			<form:input path="books[${bookStatus.index}]" />
			<form:errors path="books[${bookStatus.index}]" cssClass="errors" />
		</li>
		</c:forEach>
	</ul>
	<input type="submit"/>
</form:form>
```

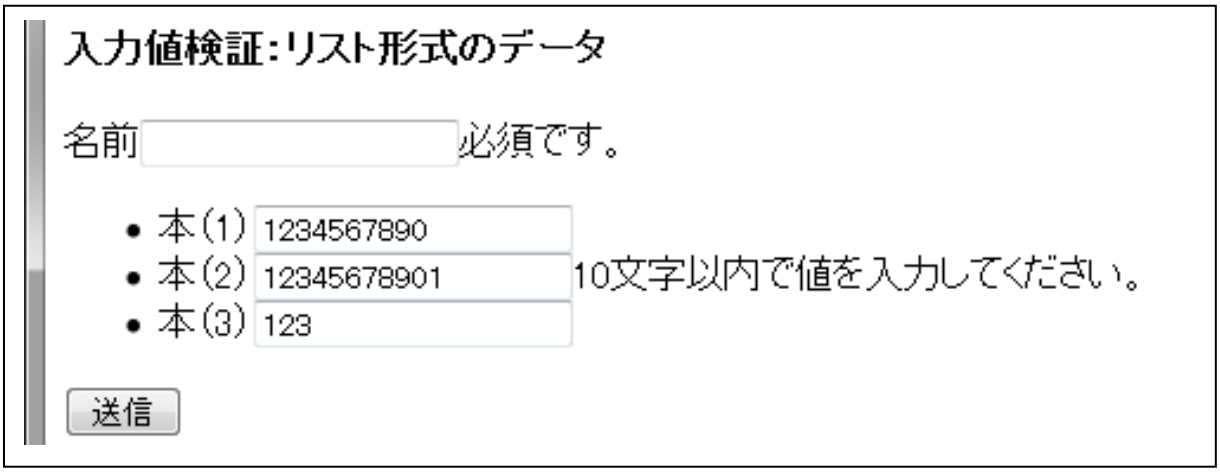
【Validator の作成】

- エラーメッセージを「Errors#rejectValue(フィールド名, エラーコード,...)」にて設定する際のフィールド名は、JSP の path 属性と同様、books[インデックス]とします。
  リスト型のプロパティに対するエラーメッセージを設定するには、フィールド名を「プロパティ[インデックス]」とします。

```
@Component
public class Sample3Validator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Sample3Command.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        // Command へキャストする
        Sample3Command command = (Sample3Command) target;

        // フィールドエラーチェック(プロパティ=name)
        if(!errors.hasFieldErrors("name")) {
            if(!StringUtils.hasLength(command.getName())) {
                errors.rejectValue("name", "error.required");
            }
        }

        for(int i=0; i < command.getBooks().size(); i++) {
            String fieldName = String.format("books[%d]", i);
            if(errors.hasFieldErrors(fieldName)) {
                continue;
            }

            String fieldValue = command.getBooks().get(i);
            if(StringUtils.hasLength(fieldValue)
                    && fieldValue.length() > 10) {
                errors.rejectValue(fieldName, "error.maxLength", new Object[]{10}, null);
            }

        }

    }
}
```

フィールド名を JSP と同じように、books[インデックス]とします。

【ブラウザでの表示】



【HTML のソース】

```
<h4>入力値検証：リスト形式のデータ</h4>
<form id="sample3Command" action="/spring3-mvc/test/sample3.html" method="post">
    <p>
        <label for="name">名前</label><input id="name" name="name" type="text" value=""/>
        <span id="name.errors" class="errors">必須です。</span>
    </p>
    <ul>
        <li>
            <label for="books0">本（1）</label>
            <input id="books0" name="books[0]" type="text" value="1234567890"/>
        </li>
        <li>
            <label for="books1">本（2）</label>
            <input id="books1" name="books[1]" type="text" value="12345678901"/>
            <span id="books1.errors" class="errors">10 文字以内で値を入力してください。</span>
        </li>
        <li>
            <label for="books2">本（3）</label>
            <input id="books2" name="books[2]" type="text" value="123"/>
        </li>
    </ul>

    <input type="submit"/>

</form>
```

7.2.8. マップを項目とするCommandの入力値検証

マップ型の場合は、リスト型の場合とほとんど同じです。フィールド名を「プロパティ名[キー名]」とすればエラーメッセージを埋め込むことができます。

【Commandの作成】

- フィールド(=プロパティ)「family」は、Map<String, String>形式のデータ型です。

```
public class Sample4Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private String name;

    private Map<String, String> family;

    @SuppressWarnings("unchecked")
    public Sample4Command() {
        family = MapUtils.lazyMap(
                new LinkedHashMap<String, String>(),
                FactoryUtils.instantiateFactory(String.class));
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
    // setter, getter は省略
}
```

【マップのキーとなるデータ】

- マップのキーとなるマスターデータとして、下記の列挙型を使用します。
  ケースにより DB から取得したリストなど様々あると思います。

```
public enum Family {
    FATHER("父"), MOTHER("母"), BROTHER("兄"), SISTER("姉");

    private String localeName;

    private Family(String localeName) {
        this.localeName = localeName;
    }

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

マップのキーとして取得するための JavaBean 形式の getter を定義する。
定義しない場合は、toString()が実行されるので特には問題ない。

【Controller の作成】

- JSP でマップのキーとなるデータを Model に格納します。今回は、列挙型なので、Enum#values()にて配列形式にしたデータを設定します。

  Model にデータを格納した場合、通常はリクエストスコープとなるため、エラー時に自画面に戻る際にはもう一度 Model に格納する必要があります。

  マップのキーとなるデータが普遍的ならば、システム起動時にアプリケーションスコープに登録するなどをお勧めします(「14.2　アプリケーションの初期化プログラムの実行」を参照)。

```
@Controller
@RequestMapping("/test/sample4")
public class Sample4Constroller {

    @Resource
    private Sample4Validator sample4Validator;

    @InitBinder("sample4Command")
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(sample4Validator);
    }

    @ModelAttribute("sample4Command")
    public Sample4Command createInitCommand() {
        Sample4Command command = new Sample4Command();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {
        Sample4Command command = createInitCommand();
        model.addAttribute("sample4Command", command);

        // マップから値を取得するためキーのリスト
        model.addAttribute("familyType", Family.values());
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample4Command") @Valid Sample4Command command,
            BindingResult bindingResult) {

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            // マップから値を取得するためキーのリスト
            mav.addObject("familyType", Family.values());
            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

画面表示のために、マップのキーを Model に格納します。

【JSP の作成】

- プロパティ family はマップ型なので、<form:XXX path=”family[キー]”>の書式で記述します。

```
<h4>入力値検証：マップ形式のデータ</h4>
<form:form modelAttribute="sample4Command" action="${appUrl}/test/sample4.html" method="post">

    <p>
        <form:label path="name">名前</form:label>
        <form:input path="name" />
        <form:errors path="name" cssClass="errors" />
    </p>
    <ul>
        <c:forEach items="${familyType}" var="type" varStatus="familyStatus">
        <li>
            <form:label path="family[${type.name}]">${type.localeName}</form:label>
            <form:input path="family[${type.name}]" />
            <form:errors path="family[${type.name }]" cssClass="errors" />
        </li>
        </c:forEach>
    </ul>

    <input type="submit"/>
</form:form>
```

Model に登録したマップのキーとなる列挙型を取り出します。

${type}としても問題なし。
その場合、列挙型の toString()が呼ばれる。

【Validator の作成】

- エラーメッセージを「Errors#rejectValue(フィールド名, エラーコード,...)」にて設定する際のフィールド名は、JSP の path 属性と同様、family[キー]とします。
  マップ型のプロパティに対するエラーメッセージを設定するには、フィールド名を「プロパティ[キー]」とします。

```
@Component
public class Sample4Validator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Sample4Command.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        // Command へキャストする
        Sample4Command command = (Sample4Command) target;

        // フィールドエラーチェック(プロパティ=name)
        if(!errors.hasFieldErrors("name")) {
            if(!StringUtils.hasLength(command.getName())) {
                errors.rejectValue("name", "error.required");
            }
        }

        for(Family family : Family.values()) {
            String fieldName = String.format("family[%s]", family.name());
            if(errors.hasFieldErrors(fieldName)) {
                continue;
            }
```

マップのキーとなるデータを列挙から取り出します。

フィールド名を JSP と同じように、family[キー]とします。

```
                String fieldValue = command.getFamily().get(family.name());
                if(StringUtils.hasLength(fieldValue)
                        && fieldValue.length() > 10) {
                    errors.rejectValue(fieldName, "error.maxLength", new Object[]{10}, null);
                }
            }

        }
}
```

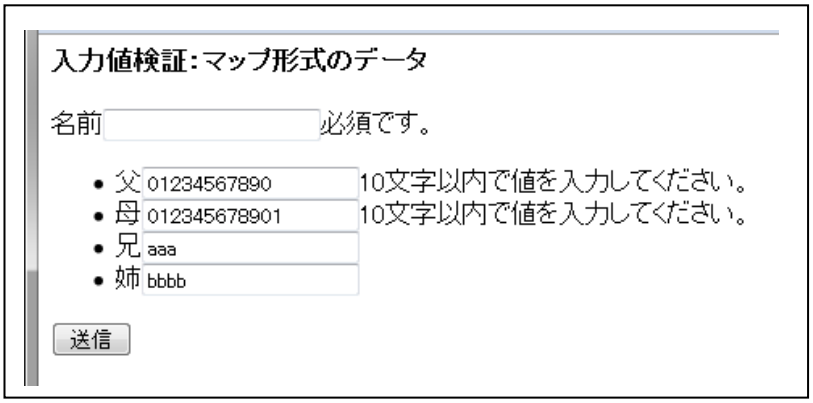
※ポイント：マップ型の場合、入力値チェックなどの際のキーの取り出し方に注意してください。

- 次の図 7.1 のように、Map.entrySet()によりデータを取り出した場合通常は問題なく動作します。
- しかし、JSP（HTML）を不正に書き換えられ、予期しないマップのキーを設定された場合、不正なキーのデータにも関わらず処理が実行される可能性があります。
- マップ型を使用する場合はキーのマスターデータを決めておき、そこから取り出すことをお勧めします。マスターデータからマップのデータを取り出すことで、不正なキーは無視され、そのまま DB に登録されるようなことを防ぐことができます。

```
// 送信されたデータをもとにキーと値を取り出す。
for(Map.Entry<String, String> entry : command.getFamily().entrySet()) {
    String fieldName = String.format("family[%s]", entry.getKey());
    if(errors.hasFieldErrors(fieldName)) {
        continue;
    }

    String fieldValue = entry.getValue();
    if(StringUtils.hasLength(fieldValue)
            && fieldValue.length() > 10) {
        errors.rejectValue(fieldName, "error.maxLength", new Object[]{10}, null);
    }


}
```

図 7.1　Map.entrySet()によるデータの取り出し

【ブラウザでの表示】



【HTML のソース】

```
<h4>入力値検証：マップ形式のデータ</h4>
<form id="sample4Command" action="/spring3-mvc/test/sample4.html" method="post">
    <p>
        <label for="name">名前</label>
        <input id="name" name="name" type="text" value=""/>
        <span id="name.errors" class="errors">必須です。</span>
    </p>

    <ul>
        <li>
            <label for="familyFATHER">父</label>
            <input id="familyFATHER" name="family[FATHER]" type="text" value="01234567890"/>
            <span id="familyFATHER.errors" class="errors">10 文字以内で値を入力してください。</span></li>
        <li>
            <label for="familyMOTHER">母</label>
            <input id="familyMOTHER" name="family[MOTHER]" type="text" value="012345678901"/>
            <span id="familyMOTHER.errors" class="errors">10 文字以内で値を入力してください。</span>
        </li>
        <li>
            <label for="familyBROTHER">兄</label>
            <input id="familyBROTHER" name="family[BROTHER]" type="text" value="aaa"/>
        </li>
        <li>
            <label for="familySISTER">姉</label>
            <input id="familySISTER" name="family[SISTER]" type="text" value="bbbb"/>
        </li>
    </ul>

    <input type="submit"/>
</form>
```

7.2.9. Validator による階層を持つ Command の入力値検証

フィールド（=プロパティ）に JavaBean を持つような Command の入力値検証を行う場合、Spring MVC では JavaBean ごとに Validator を作成し処理します（図 7.2）。

- JavaBean ごとに Validator を呼び出すために、現在の JavaBean の位置(=path)を移動してから、Validator を呼び出します。
- 現在の位置は、スタック構造で管理します。
  1 段下のネストしたプロパティに移動する「Errors#pushNestedPath(プロパティ名)」と、1 つ上の階層に戻る「Errors#popNestedPath()」を利用します。

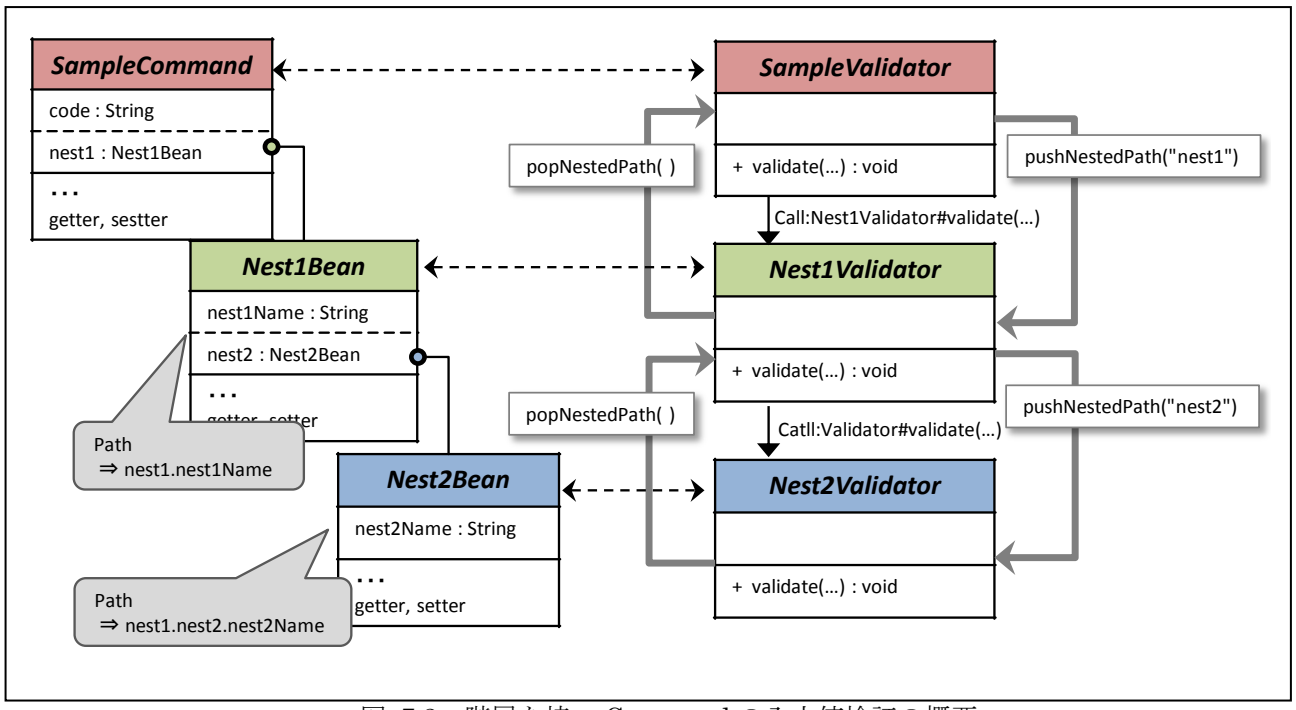


図 7.2　階層を持つ Command の入力値検証の概要

【階層構造を持つ Command】

```
public class Sample5Command implements Serializable {

	private String name;
	private MemberCardBean memberCard;
	private List<BookBean> books;

	@SuppressWarnings("unchecked")
	public Sample5Command() {

		memberCard = new MemberCardBean();

		books = ListUtils.lazyList(
				new ArrayList<String>(),
				FactoryUtils.instantiateFactory(BookBean.class));
	}
	// getter、setter、toString()は省略

}

public class MemberCardBean implements Serializable {

	protected String code;
	protected Date entryDate;

	public MemberCardBean() {

	}
	// getter、setter、toString()は省略
}

public class BookBean implements Serializable{

	protected String title;
	protected Integer price;
	protected List<String> authors;

	// getter、setter、toString()は省略
}
```

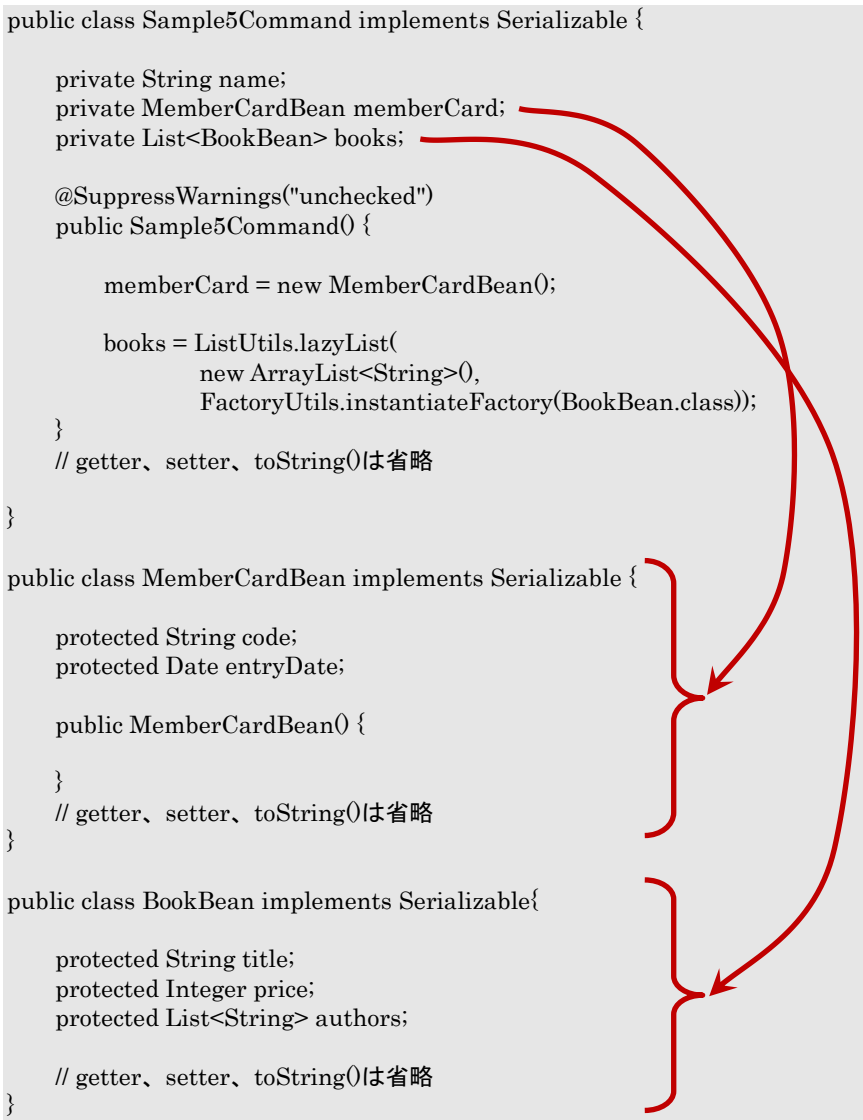


図 7.3　プロパティに JavaBean を持つ Command

【Validator の作成】

- プロパティ「memberCard」の値を検証する場合、「Errors#pushNestedPath(“memberCard”)」にて、階層を 1 つ下にネストしてから、Validator を呼び出します。
  検証完了後は、「Errors#popNestedPath()」にてパスを現在の位置に戻します。
  検証中に例外が発生した場合を考慮し、finally 句を記述し必ず元の位置に戻るようにします。
- ネストした JavaBean の Validator を呼び出すときには、「ValidationUtils.invokeValidator()」で呼び出します。メソッド内で Validator#supports()などを呼び出し、型チェックなどを行ってくれます。

- リスト型のプロパティ「books」の検証も、基本的に同じです。
  項目ごとに値の検証を行うため、「Errors#pushNestedPath(“books[インデックス]”)」にてネストした１つ下の階層に移動します。
- Spring Bean として登録しておくことで、各 JavaBean の Validator がインジェクションするだけで利用できるようになります。

```
@Component
public class Sample5Validator implements Validator {

    @Resource
    private MemberCardValidator memberCardValidator;

    @Resource
    private BookValidator bookValidator;

    @Override
    public boolean supports(Class<?> clazz) {
        return Sample5Command.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        // Command へキャストする
        Sample5Command command = (Sample5Command) target;

        // フィールドエラーチェック(プロパティ=name)
        if(!errors.hasFieldErrors("name")) {
            if(!StringUtils.hasLength(command.getName())) {
                errors.rejectValue("name", "error.required");
            }
        }

        try {
            // MembarCardBean の入力値チェック
            errors.pushNestedPath("memberCard");
            ValidationUtils.invokeValidator(memberCardValidator, command.getMemberCard(), errors);
        } finally {
            errors.popNestedPath();

        }

        // リスト型の Book の入力値チェック
        for(int i=0; i < command.getBooks().size(); i++) {
            try {
                errors.pushNestedPath(String.format("books[%d]", i));
                ValidationUtils.invokeValidator(bookValidator, command.getBooks().get(i), errors);
            } finally {
                errors.popNestedPath();
            }
        }

    }
}
```

Spring Bean として登録しておくことで、インジェクションして利用できる。

チェック対象のプロパティの位置をスタックに追加する

JavaBean 「 MemberCardBean 」 の Validator を呼び出す。

１つ上に移動し、現在の位置に戻る。

リスト型の場合は、項目１ずつに対して Validator を呼び出す。
マップ型も同様。

図 7.4　プロパティに JavaBean を持つ Command の Validator

【ネストしたJavaBeanのValidatorの作成】

- 通常のJavaBeanをプロパティに持たないValidatorと同じです。
- MemberCardのプロパティ「code」のフィールド名は、実際には「memberCar.code」となります。上位の階層（「図 7.4」のSimple5Validator）にて、Errors#pushNestedPath(“member”)としているため、フィールド名に自動的に「member.」が付加された状態となります。
- 上位の階層で設定されたパスは、「Errors#getNestedPath()」で確認することができます。

```
@Component
public class MemberCardValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return MemberCardBean.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        MemberCardBean command = (MemberCardBean) target;
        System.out.printf("MemberCardValidator,bojectName=%s, nestedPath=%s¥n",
                errors.getObjectName(), errors.getNestedPath());

        // フィールドエラーチェック(プロパティ=code)
        if(!errors.hasFieldErrors("code")) {
            if(StringUtils.hasLength(command.getCode())
                    && command.getCode().length() > 5) {
                errors.rejectValue("code", "error.maxLength",
                        new Object[]{5}, null);;
            }
        }

        // フィールドエラーチェック(プロパティ=entryDate)
        if(!errors.hasFieldErrors("entryDate")) {
            SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
            Date startRange = Timestamp.valueOf("2010-01-01 00:00:00.000");
            Date endRange = new Date();

            if(command.getEntryDate().compareTo(startRange) < 0
                    || command.getEntryDate().compareTo(endRange) > 0) {
                errors.rejectValue("entryDate", "error.dateRange",
                        new Object[]{dateFormat.format(startRange), dateFormat.format(endRange)}, null);
            }
        }
    }
}
```

```
@Component
public class BookValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return BookBean.class.isAssignableFrom(clazz);
    }

    @Override
```

```
    public void validate(Object target, Errors errors) {
        BookBean command = (BookBean) target;
        System.out.printf("bookValidator,bojectName=%s, nestedPath=%s¥n",
                errors.getObjectName(), errors.getNestedPath());

        // フィールドエラーチェック(プロパティ=price)
        if(errors.hasFieldErrors("price")) {
            if(command.getPrice() != null
                    && !(1 <= command.getPrice() && command.getPrice() <= 10000)) {
                errors.rejectValue("price", "error.range",
                        new Object[]{1, 10000}, null);
            }
        }

        // フィールドエラーチェック(プロパティ=authors)
        for(int i=0; i < command.getAuthors().size(); i++) {
            final String fieldName = String.format("authors[%d]", i);
            if(!errors.hasFieldErrors(fieldName)) {
                String fieldValue = command.getAuthors().get(i);
                if(StringUtils.hasLength(fieldValue)
                        && fieldValue.length() > 10) {
                    errors.rejectValue(fieldName,
                            "error.maxLength", new Object[]{10}, null);
                }
            }
        }
    }
}
```

【ブラウザでの表示】

【HTML のソース】

```
<h4>入力値検証：階層を持つデータ</h4>
<form id="sample5Command" action="/spring3-mvc/test/sample5.html" method="post">
    <p>
        <label for="name">名前</label><input id="name" name="name" type="text" value=""/>
        <span id="name.errors" class="errors">必須です。</span>
    </p>
    <h5>メンバーカード情報</h5>
    <table>
        <tr>
            <th>コード</th>
            <td>
                <input id="memberCard.code" name="memberCard.code" type="text" value="012345"/>
                <span id="memberCard.code.errors" class="errors">5 文字以内で値を入力してください。</span>
            </td>
        </tr>
        <tr>
            <th>入会日付</th>
            <td>
                <input id="memberCard.entryDate" name="memberCard.entryDate" type="text"
value="2011/10/17"/>
                <span id="memberCard.entryDate.errors" class="errors">2010/01/01～2011/10/16 の範囲で値を入
力してください。</span>
                </td>
        </tr>
    </table>

    <h5>購入した本の情報</h5>
    <table border="1">
        <tr>
            <th>No.</th>
            <th>題名</th>
            <th>価格</th>
            <th>著者</th>
        </tr>
        <tr>
            <td>1</td>
            <td><input id="books0.title" name="books[0].title" type="text" value=""/></td>
            <td><input id="books0.price" name="books[0].price" type="text" value=""/></td>
            <td>
                <input id="books0.authors0" name="books[0].authors[0]" type="text" value="01234567891"/>
                <span id="books0.authors0.errors" class="errors">10 文字以内で値を入力してください。</span>
                <input id="books0.authors1" name="books[0].authors[1]" type="text" value=""/>
            </td>
        </tr>
        <tr>
            <td>2</td>
            <td><input id="books1.title" name="books[1].title" type="text" value=""/></td>
            <td>
                <input id="books1.price" name="books[1].price" type="text" value="-1"/>
                <span id="books1.price.errors" class="errors">1 から 10,000 の間の値を入力してください。</span>
            </td>
            <td>
                <input id="books1.authors0" name="books[1].authors[0]" type="text" value=""/>
                <input id="books1.authors1" name="books[1].authors[1]" type="text" value=""/>
            </td>
        </tr>
        <tr>
```

```
            <td>3</td>
            <td><input id="books2.title" name="books[2].title" type="text" value=""/></td>
            <td><input id="books2.price" name="books[2].price" type="text" value=""/></td>
            <td>
                <input id="books2.authors0" name="books[2].authors[0]" type="text" value=""/>
                <input id="books2.authors1" name="books[2].authors[1]" type="text" value=""/>
            </td>
        </tr>
    </table>

    <input type="submit"/>
</form>
```

7.2.10. エラーメッセージの定義

Validator によるエラーは、専用のメッセージを用意することで、Command ごと・入力項目ごとにカスタマイズすることができます。

- メッセージは、Spring の messageSource として読み込むプロパティファイルに定義します。
  定義方法などは、「2.4.3　アプリケーション用(共通の)Spring Bean ファイル」を参照してください。
- メッセージコードは、任意に設定することができます。
  これらのメッセージは、「org.springframework.validation.DefaultMessageCodesResolver」で処理されます。
  キー名の指定方法により、プロパティ名（=フィールド名）に対するメッセージを優先順位を決めて指定することができます（表 7.7）。また、プロパティファイルでの定義順は関係なく、メッセージコードの形式により一致します。
- データバインドのエラーメッセージのコードが任意に設定になったのと変わりません。
  「4.2　データバインドエラー（型ミスマッチ）処理」参照。

表 7.7　Validator のメッセージコードと優先度

優先度	メッセージコードの形式	説明
1	エラーコード.[Command 名].[フィールド名]	特定の Command のフィールド名に一致する場合のメッセージです。 あまり使用する機会はないと思います。
2	エラーコード.[フィールド名]	フィールド名（プロパティ名）と一致する場合のメッセージです。
3	エラーコード.[形名]	フィールドのクラス型と一致する場合のメッセージです。
4	エラーコード	優先度の高いメッセージコードに該当するものがない場合に一致します。 必ず記述しておく必要があります。 通常は、この形式を使用します。

※DefaultMessageCodesResolver の Javadoc にも詳しく記載されています。

7.3. Bean Validation を利用した入力値検証

Spring MVC は、Bean Validation(JSR-303)を正式にサポートしています。実装は Hibernate の API「Hibernate Validator」を使用します。特徴として以下のことが挙げられます。

- アノテーションのみで設定し、ロジックを排除することで、コード量を減らすことができる。
- 単項目のチェックしかできず、複雑な項目間のチェックはできない。
- 最大値などの値をアノテーションにて指定するので、パラメータをプロパティファイルや DB などで外部化できない。
- 「7.2.9　Validator による階層を持つ Command の入力値検証」のようにネストした Bean のようなチェックはできない。

7.3.1. Bean Validation の準備

【pom.xml の編集】

- 依存ライブラリとして、Bean Validation を追加します。

  Bean Validation の実装の 1 つである、Hibernate Validator も追加します。

```
<project>
    ・・・省略
    <dependencies>
        <!-- validator -->
        <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
            <version>1.0.0.GA</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
            <version>4.2.0.Final</version>
        </dependency>
    </dependencies>
    ・・・省略
</project>
```

【servlet-context.xml の編集】

- Spring MVC 形式の BeanValidation の Validator を定義します。
- 基本的にはこの設定のみで動作しますが、「AnnotationMethodHandlerAdapter」の Bean を独自に設定している場合、動作しない場合があり、さらに設定が必要になります。

  詳細は、「7.3.5　こんなときは：Bean Validation がうまく動作しない場合」を参照してください。

```
<beans>
    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven/>

    <!-- Bean Validation 用の Validator -->
    <bean id="beanValidator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
</beans>
```

7.3.2. Bean Validation を使用する

【Command の作成】

- アノテーションを Command のプロパティ（=フィールド）に定義します。
- チェック順はアノテーションの定義した順番とは限りませんので注意してください。
  チェックを実行する度に変わります。
- 標準で使用可能なアノテーションは「7.3.3　Bean Validation のアノテーションの一覧」を参照してください。
- アノテーションは、プロパティの getter メソッドにも定義できますが、通常はプロパティ自身に付与します。

```
import java.io.Serializable;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import org.apache.commons.lang.builder.ToStringBuilder;
import org.hibernate.validator.constraints.Length;

public class Sample6Command implements Serializable {
    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    @NotNull
    @Pattern(regexp="[a-z]*")
    @Length(max=5)
    private String name;

    @Min(0)
    @Max(200)
    private Integer age;

    public Sample6Command() {
    }

    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
    // getter、setter は省略
}
```


【メッセージファイルの作成】

- プロパティファイル「ValidationMessages.properties」を、クラスパスのルートに配置します。
- 初期ファイルは、Hibernate Validator の jar「hibernate-validator-XXX.jar」のパッケージ「org.hibernate.validator」中にあるプロパティファイル「ValidationMessages.properties」を参照してください。
- 任意の位置にプロパティファイルを配置したい場合は、「7.3.4　こんなときは：エラーメッセージの定義場所を変更したい」を参照してください。

【ValiadtionMessages.properties の定義】

- エラーメッセージのコードは、「アノテーションのクラスパス.message」となっています。
- メッセージの置換文字として、アノテーションで設定した引数名を使用することができます。
- Command ごとにメッセージを設定したいときには、「7.3.6　こんなときは：Command ごとにメッセージを変更したい」を参照してください。

```
## Bean Validator 用のメッセージ

## JSR-303 のエラーメッセージ
javax.validation.constraints.AssertFalse.message=true を設定してください。
javax.validation.constraints.AssertTrue.message=fale を設定してください。
javax.validation.constraints.DecimalMax.message={value}より同じか小さい値を入力してください。
javax.validation.constraints.DecimalMin.message={value}より同じか大きい値を入力してください。
javax.validation.constraints.Digits.message=整数{integer}桁以内、小数{fraction}桁以内で入力してください。
javax.validation.constraints.Future.message=未来の日付を入力してください。
javax.validation.constraints.Max.message={value}より同じか小さい値を入力してください。
javax.validation.constraints.Min.message={value}より同じか大きい値を入力してください。
javax.validation.constraints.NotNull.message=値が未入力です。
javax.validation.constraints.Null.message=値は未入力でなければいけません。
javax.validation.constraints.Past.message=過去の日付を入力してください。
javax.validation.constraints.Pattern.message="{regexp}"にマッチしていません。
javax.validation.constraints.Size.message=サイズは{min}から{max}の間の値を入力してください。

## Hibernate Validator のエラーメッセージ
org.hibernate.validator.constraints.Email.message=E-mail 形式で入力してください。
org.hibernate.validator.constraints.Length.message=文字の長さは{min}から{max}の間で入力してください。
org.hibernate.validator.constraints.NotBlank.message=値が空白以外を入力してください。
org.hibernate.validator.constraints.NotEmpty.message=値が未入力です。
org.hibernate.validator.constraints.Range.message={min}から{max}の間の値を入力してください。
org.hibernate.validator.constraints.CreditCardNumber.message=不正なクレジットカードの番号です。
org.hibernate.validator.constraints.SafeHtml.message=may have unsafe html content
org.hibernate.validator.constraints.ScriptAssert.message=script expression "{script}" didn't evaluate to true
org.hibernate.validator.constraints.URL.message=不正な URL の形式です。
```

【Controller の作成】

- @RequestMapping が付与されているメソッドの Command の引数に、アノテーション「@Valid」を付与します。
- リクエスト受信時に、自動的に入力値検証が実行されます。

```
import javax.annotation.Resource;
import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.Validator;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
```

```
@Controller
@RequestMapping("/test/sample6")
public class Sample6Constroller {

	@ModelAttribute("sample6Command")
	public Sample6Command createInitCommand() {
		Sample6Command command = new Sample6Command();
		return command;
	}

	@RequestMapping(method=RequestMethod.GET)
	public void setupForm(Model model) {
		Sample6Command command = createInitCommand();
		command.setAge(0);
		model.addAttribute("sample6Command", command);
	}

	@RequestMapping(method=RequestMethod.POST)
	public ModelAndView doAction1(
			@ModelAttribute("sample6Command") @Valid Sample6Command command,
			BindingResult bindingResult) {

		// エラーがある場合、自画面遷移する
		if(bindingResult.hasErrors()) {
			ModelAndView mav = new ModelAndView();
			mav.getModel().putAll(bindingResult.getModel());
			return mav;
		}

		ModelAndView mav = new ModelAndView("forward:/hello.html");
		return mav;
	}
}
```

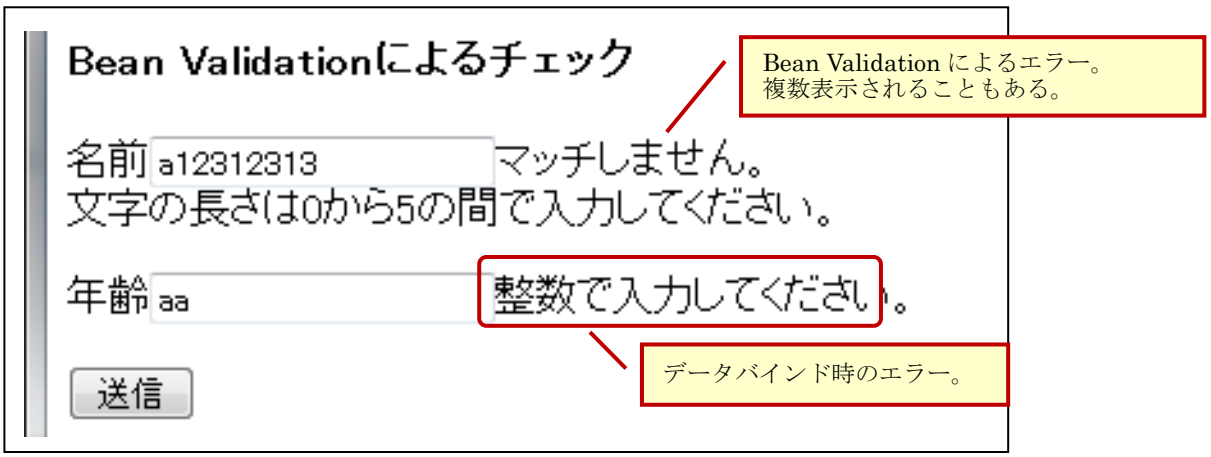
【JSP の作成】

- 通常の Validator を使用した場合と変わりません。

```
<h4>Bean Validation によるチェック</h4>
<form:form modelAttribute="sample6Command" action="${appUrl}/test/sample6.html" method="post">

	<p>
		<form:label path="name">名前</form:label>
		<form:input path="name" />
		<form:errors path="name" cssClass="errors" />
	</p>
	<p>
		<form:label path="age">年齢</form:label>
		<form:input path="age" />
		<form:errors path="age" cssClass="errors" />
	</p>

	<input type="submit" name="check1"/>
</form:form>
```

【ブラウザでの表示】

- 複数のエラーに該当する場合、エラーは複数出力されます。
  複数該当するようなケースのアノテーションの組合せを付与しないことをお勧めします。
  この例では、正規表現@Pattern と文字数@Length がエラーとなっていますが、@Length のチェックは正規表現で表現可能なので、この場合は必要ありません。
  例) @Pattern(regexp="[a-z]*") ⇒ @Pattern(regexp="[a-z]{0,5}")
- データバインド時のエラーが発生しているフィールドは、Bean Validation による検証は実行されません。



【HTML のソース】

- 複数エラーがある場合、
タグ改行して表示します。

```
<h4>Bean Validation によるチェック</h4>
<form id="sample6Command" action="/spring3-mvc/test/sample6.html" method="post">
    <p>
        <label for="name">名前</label>
        <input id="name" name="name" type="text" value="a12312313"/>
        <span id="name.errors" class="errors">マッチしません。<br/>文字の長さは 0 から 5 の間で入力してください。
</span>
    </p>
    <p>
        <label for="age">年齢</label>
        <input id="age" name="age" type="text" value="aa"/>
        <span id="age.errors" class="errors">整数で入力してください。</span>
    </p>

    <input type="submit" name="check1"/>
</form>
```

7.3.3. Bean Validation のアノテーションの一覧

7.3.3.1. Bean Validation (JSR 303)のアノテーション

- パッケージ「javax.validation.constraints」にあるアノテーションです。
- 共通の引数として次のものがあります。
  - 「String message」：エラー時のメッセージを定義します。
    実際には、「7.3.6　こんなときは：Command ごとにメッセージを変更したい」の方法をとることをお勧めします。
  - 「Class<?>[] groups」：グルーピングし、ある条件の場合に同じグループに属するアノテーションのみを処理します。
    実装により異なるため使用しないことをお勧めします。
- Javadoc は、JSR-303 の下記の URL からダウンロードできます。
  「http://jcp.org/en/jsr/detail?id=303」

表 7.8　Bean Validation(JSR-303)のアノテーション一覧

No.	アノテーション	フィールド型(※1)	説明
1	@NotNull	Object	Null かどうかチェックします。
2	@Null	Object	Null でないかどうかチェックします。
3	@AssertFalse	boolean 対応するラッパークラス	True かどうかチェックします。 値が null の場合は正常値であると判断します。
4	@AssertTrue	boolean,Boolean	Flase かどうかチェックします。 値が null の場合は正常値であると判断します。
5	@DecimalMin	String byte,short,int,long float,double, 対応するラッパークラス BigDecimal,BigInteger	指定した小数値以上かどうかチェックします。 [引数]String value : 必須。最小値を小数で設定します。 値が null の場合は正常値であると判断します。
6	@DecimalMax	String byte,short,int,long float,double, 対応するラッパークラス BigDecimal,BigInteger	指定した小数値以下かどうかチェックします。 [引数]String value : 必須。最大値を小数で設定します。 値が null の場合は正常値であると判断します。
7	@Digits	String byte,short,int,long 対応するラッパークラス BigDecimal,BigInteger	指定した桁以内かどうかチェックします。 [引数]int integer : 必須。整数部の最大桁数を設定します。 [引数]int fraction : 必須。小数部の最大桁数を設定し

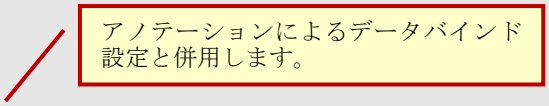
			ます。 値が null の場合は正常値であると判断します。
8	@Min	String byte,short,int,long 対応するラッパークラス BigDecimal,BigInteger	指定した整数以上かどうかチェックします。 [引数]long value：必須。最小値を設定します。 値が null の場合は正常値であると判断します。
9	@Max	String byte,short,int,long 対応するラッパークラス BigDecimal,BigInteger	指定した整数以下かどうかチェックします。 [引数]long value：必須。最大値を設定します。 値が null の場合は正常値であると判断します。
10	@Size	String java.util.Collection java.util.Map 配列	文字数、リストサイズなどが指定した範囲以内のサイズかどうかチェックします。 [引数]int min：最小値を設定します。 [引数]int max：最大値を設定します。 値が null の場合は正常値であると判断します。
11	@Pattern	String	指定した正規表現に一致するかどうかチェックします。 値が null の場合は正常値であると判断します。
12	@Past	java.util.Date java.util.Calendar	現在の日付より過去かどうかチェックします。 値が null の場合は正常値であると判断します。
13	@Futrue	java.util.Date java.util.Calendar	現在の日付より未来かどうかチェックします。 値が null の場合は正常値であると判断します。

※1 アノテーションを付与可能なフィールド型(=プロパティのクラスタイプ)です。

【Bean Valiation のサンプル】

```
public class SampleCommand {

    @NotNull
    private String objNotNull;

    @Null
    private String objNull;

    @AssertFalse
    private boolean assertFalse;

    @AssertTrue
    private boolean assertTrue;

    @DecimalMin(value="5.5")
    private BigDecimal decimalMin;

    @DecimalMax(value="5.5")
    private BigDecimal decimalMax;
```

```
    @Digits(integer=3,fraction=2)
    private BigDecimal digits;

    @Min(value=3)
    private int min;

    @Max(value=3)
    private int max;

    @Size(min=3,max=5)
    private String size;

    @Pattern(regexp="[a-z]+")
    private String pattern;

    @Past
    @DateTimeFormat(pattern="yyyy/MM/dd")
    private Date pastDate;

    @Future
    @DateTimeFormat(pattern="yyyy/MM/dd")
    private Date futureDate;
}
```



7.3.3.2.　Hibernate Validator のアノテーション

- パッケージ「org.hibernate.validator.constraints」にあるアノテーションです。
  値の検証を行う実装クラスは、パッケージ「org.hibernate.validator.constraints.impl」にあります。
- 基本的に、JSR-303 にて実用では不足しているものが追加されています。
- 共通の引数として次のものがあります。
  - 「String message」：エラー時のメッセージを定義します。
    実際には、「7.3.6　こんなときは：Command ごとにメッセージを変更したい」の方法をとることをお勧めします。
  - 「Class<?>[] groups」：グルーピングし、ある条件の場合に同じグループに属するアノテーションのみを処理します。
    実装により異なるため使用しないことをお勧めします。
- ドキュメントや Javadoc などは、次の URL からダウンロードできます。
  ソース「http://www.hibernate.org/subprojects/validator/download」
  ドキュメント「http://www.hibernate.org/subprojects/validator/docs」

表 7.9　Hibernate Validator のアノテーション一覧

No.	アノテーション	フィールド型(※1)	説明
1	@NotBlank	String	文字列が null または空白スペースでないかチェックします。 @NotEmpty と違い、半角スペースのみの場合も正常値と判断します。
2	@NotEmpty	String java.util.Collection java.util.Map 配列	文字数や Collection のサイズが 0 または null かどうかチェックします。
3	@Length	String	文字数の長さが指定した範囲内にあるかどうかチェックします。 [引数]int min：最小値を設定します。初期値 0。 [引数]int max：最小値を設定します。初期値 2147483647。 値が null の場合は正常値であると判断します。
4	@Range	byte,short,int,long 対応するラッパークラス BigDecimal,BigInteger	指定した整数の範囲内にあるかどうかチェックします。 [引数]long min : 最小値を設定します。初期値 0L。

			[引数]long max：最大値を設定します。初期値9223372036854775807L。 値が null の場合は正常値であると判断します。
5	@CreditCardNumber	String	Luhn アルゴリズムによるクレジットカードの番号として正しいかチェックします。 値が null の場合は正常値であると判断します。
6	@Email	String	RFC-2822 に従ったメールアドレスのパターンとして正しいかチェックします。 値が null の場合は正常値であると判断します。
7	@SafeHtml	String	<script>タグを含むような悪意のある HTML でない安全なものかチェックします。WISIWYG などのリッチテキストなどのチェックに使用します。 [引数]SafeHtml.WhiteListType whiteListType：ホワイトリストのタグ（安全なタグ）を設定します。予め用意されている列挙型 WhiteListType から選択します。 [引数]String[] additionalTags：予めい用意されているホワイトリストのタグの列挙型に、安全なタグを追加します。 値が null の場合は正常値であると判断します。
8	@ScriptAssert	String	JSR-223 で取り込まれた Java Script API として、正しい書式であるかどうかチェックします。 [引数]String lang：言語情報を設定します。必須です。 [引数]String script：指定したスクリプトを実行します。必須です。 値が null の場合は正常値であると判断します。
9	@URL	String	URL の形式として正しいか、または指定した条件に合う URL かどうかチェックします。 [引数]String protocol：許可するプロトコルを指定します。初期値は空文字です。 [引数]String host：許可するホスト名部分を指定します。初期値は空文字です。 [引数]int port：許可するポート番号を指定します。初期値は-1 です。80 番ポートはデフォルトで許可します。

			[引数]String regexp：許可する URL の形式を正規表現で定義します。初期値は、「.*」です。 [引数]Pattern.Flag[] flags：引数 regexp を使用した場合の正規表現のオプションを指定します。初期値は空{}です。 値が null の場合は正常値であると判断します。

※1 アノテーションを付与可能なフィールド型(=プロパティのクラスタイプ)です。

【Hibernate Validator のサンプル】

```
public class SampleCommand {

    @NotEmpty
    private String notEmpty;

    @Length(min=3,max=5)
    private String length;

    @Range(min=3,max=5)
    private BigDecimal range;

    @Email
    private String email;
}
```

7.3.4. こんなときは：エラーメッセージの定義場所を変更したい

Bean Validation 用のエラーメッセージの定義方法は、次の 2 つ方法があります。お好きな方法をご利用ください。

【標準設定】

- Hibernate Validator は標準では、クラスパスの直下（ルート）の「**ValidationMessage.properties**」という名称のプロパティファイルをメッセージファイルとして認識します。
- ロケールごとに別々に定義したい場合は、ファイル名のロケールを付け「ValidationMessage_ja.properties」のように定義します。
- サンプルは、Hibernate Validtor の jar「hibernate-validator-XXX.jar」のパッケージ「org.hibernate.validator」に言語ごとに格納されています。

【任意の場所にプロパティファイルを配置する】

- Spring Bean「messageSource」として、メッセージファイルを読み込みます。
  任意の場所に、任意のファイル名で定義することができます。
- Bean Validation 用の Validator のプロパティ「validationmMessageSource」に、定義した messageSource をインジェクションします。

```
<beans>

    <!-- 共通のメッセージファイル -->
    <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>classpath:message/message</value>
                <value>classpath:message/label</value>
                <value>classpath:message/ValidationMessages</value>
            </list>
        </property>
    </bean>

    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven/>

    <bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
        <property name="validationMessageSource"><ref bean="messageSource" /></property>
    </bean>

</beans>
```

messageSource として設定します。

プロパティに messageSource をインジェクションします。

7.3.5. こんなときは：Bean Validation がうまく動作しない場合

DataBinder をカスタマイズするために、Spring の設定ファイルに「AnnotationMethodHandlerAdapter」を定義している場合デフォルト設定が変わってしまいうまく動作しません。

その場合、次の「7.3.5.1　各 Controller で Bean Validator を関連付ける」「7.3.5.2　各 Controller で Bean Validator を関連付ける」にある方法を取ってください。

7.3.5.1. 標準の Validator として Bean Validator を登録する

【setvlet-context.xml の編集】

- AnnotationMethodHandlerAdapter のプロパティ「webBindingInitializer」に共通設定用のクラスをインジェクションします。
- 通常は、「4.3.4　システム全体のバインドの設定」にあるように、システム全体のデータバインドの設定のために使用します。

```
<beans>

    <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
        <property name="cacheSeconds" value="0" />
        <property name="webBindingInitializer">
            <bean class="sample.web.GlobalBindingInitializer" />
        </property>
    </bean>

    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven/>

    <!-- Bean Validation 用の Validator -->
    <bean id="beanValidator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>

</beans>
```

システム共通の WebBinder の定義を設定します。

【GlobalBindingInitializer の編集】

```
package sample.web;

import java.text.SimpleDateFormat;
import java.util.Date;

import javax.annotation.Resource;

import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.validation.Validator;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebBindingInitializer;
import org.springframework.web.context.request.WebRequest;

public class GlobalBindingInitializer implements WebBindingInitializer {

    @Resource
    private Validator beanValidator;

    @Override
    public void initBinder(WebDataBinder binder, WebRequest request) {
```

Bean Validation の Validator をインジェクションします。

```
        //型(Date)を指定した Bind 設定
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));

        // Bean Validation の Validator を標準 Validator として登録する
        binder.setValidator(beanValidator);
    }
}
```

システム標準の Validator として登録します。

7.3.5.2.　各 Controller で Bean Validator を関連付ける

- Controller の@InitBinder を付加したメソッドにおいて、Bean Validation の Validator を Command の Validator として登録します。
- 「7.2.6　ポイント：@Valid を使用した Validator の呼び出し」で通常の Validator を登録する場合と同様です。

```
@Controller
@RequestMapping("/test/sample6")
public class Sample6Constroller {

    @Resource
    private Validator beanValidator;

    @InitBinder("sample6Command")
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(beanValidator);

    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample6Command") @Valid Sample6Command command,
            BindingResult bindingResult) {

        // 省略
    }
}
```

Bean Validation の Validator をインジェクションします。

Command の Validator として登録します。

7.3.6. こんなときは：Command ごとにメッセージを変更したい

Command ごとにメッセージを変更したい場合、次の 2 つの方法があります。

(1) アノテーションの引数「message」を指定する。

例) @Max(value=10, message="{value}以下の値を入力してください。")

この方法は、プログラムに直接メッセージを書き込むため、メンテナンス性が悪くなります。

(2) プロパティファイルのメッセージのコードを Command ごと、プロパティごと(フィールドごと)に定義します。

通常は、この方法を使用します。

表 7.10　Bean Validation のエラーコードと優先度

優先度	メッセージコードの形式	説明
1	アノテーション名.[Command 名].[フィールド名]	特定の Command のフィールド名に一致する場合のメッセージです。
2	アノテーション名.[フィールド名]	フィールド名（プロパティ名）と一致する場合のメッセージです。
3	アノテーション名.[形名]	フィールドのクラス型と一致する場合のメッセージです。
4	アノテーション名	アノテーション名と一致する場合のメッセージです。 通常は、(4)か(5)の形式を使用し、どちらかの形式を必ず記述しておきます。
5	アノテーションのクラスパス.message	優先度の高いメッセージコードに該当するものがない場合に一致します。 通常は、(4)か(5)の形式を使用し、どちらかの形式を必ず記述しておきます。

【@Max のエラーコードの例】

```
Max.sample6Command.name=優先度 1
Max.name=優先度 2
Max.java.lang.Integer=優先度 3
Max=優先度 4
javax.validation.constraints.Max.message=優先度 5
```

7.3.7. Bean Validation のアノテーションを独自実装する

Bean Validation の独自のアノテーションを追加する場合、Hibernate Validator のアノテーションとして追加します。非常に簡単に作成できます。

- 参考
  「http://docs.redhat.com/docs/ja-JP/JBoss_Enterprise_Application_Platform/4.3/html/Hibernate_Annotations_Reference_Guide/Hibernate_Validator.html」
- Hibernate Validator のパッケージ「org.hibernate.validator.constraints.impl」以下にある Validator のソースも非常に参考になります。コード量も少なくシンプルなので容易に理解することができると思います。

表 7.11　独自の Bean Validation 用のアノテーションを作成するたに必要なもの

No.	項目	内容
1	アノテーションの定義クラス	作成するアノテーションの Java の定義ファイル。
2	Validator の実装クラス	アノテーションが付加された項目の値を検証する Validator クラス。
3	エラーメッセージファイル	アノテーションに対するエラーメッセージ。

【作成するアノテーションの仕様】
サンプルとして次の仕様のアノテーションを作成します。

- 日付が指定した範囲内にあるかどうかチェックする。
  例) @DateRange(min="2000-01-01 00:00:00", max="2050-01-01 00:00:00")
- 引数 min、max は必須。
- エラーメッセージコードは「sample.web.validator.hibernatevalidator.DateRange.message」。
- Validator の実装クラスは、「DateRangeValidator」

【アノテーションの定義クラス】

- メタアノテーション「@Constraint」にて入力値の検証を行う Validator クラスを関連付けます。
- 定義可能場所を設定するメタアノテーション「@Target」は、基本的に「FILED」のみで足りますが、ほかのアノテーションと仕様を合わせるためにほかのものも追加しておきます。
- 引数 min、max を追加します。アノテーションの引数はプリミティブ型しか定義できないため、文字列型で指定します。
  今回は min、max は必須なので default は指定しません。
- メッセージコードとして引数「message」を追加します。
  通常は使用しないので、デフォルト値を設定します。その際に、他のアノテーションと仕様を合わせるために、「{クラスパス.message}」とします。

```
package sample.web.validator.hibernatevalidator;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;


/**
 * Bean Validation のアノテーション
 * <p>日付が指定した範囲内に含まれるかどうか。
 *
 */
@Documented
@Constraint(validatedBy = DateRangeValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface DateRange {

    /** 開始時刻（書式:yyyy-MM-dd HH:mm:ss） */
    String min();

    /** 開始時刻（書式:yyyy-MM-dd HH:mm:ss） */
    String max();

    /** メッセージコード */
    String message() default "{sample.web.validator.hibernatevalidator.DateRange.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    /** Defines several {@code @DateRange} annotations on the same element. */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    public @interface List {
        DateRange[] value();
    }
}
```

Validator を関連付けます。

必須の引数を定義する。

必須の引数を定義する。

他のアノテーションと仕様を合わせるために追加します。

【Validator の作成】

- Hibernate のインタフェース「ConostraintValidator」を実装します。
  Generics として、アノテーションのクラスと、アノテーションを付加数るフィールドのクラスを設定します。
- メソッド「initialize( )」にて、アノテーションの引数を取得します。また、必要ならば整合性のチェックを行います。
- メソッド「isValid( )」にて入力値検証のロジックを定義します。
  戻り値は true の場合正常値として判定するようにします。
  他のアノテーションと仕様を合わせるために、値が null の場合は正常値と判定します。

```
package sample.web.validator.hibernatevalidator;

import java.sql.Timestamp;
import java.util.Date;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class DateRangeValidator implements ConstraintValidator<DateRange, Date> {

    private Date min;
    private Date max;

    @Override
    public void initialize(DateRange parameters) {
        min = Timestamp.valueOf(parameters.min());
        max = Timestamp.valueOf(parameters.max());
        validatePrameters();
    }

    @Override
    public boolean isValid(Date value, ConstraintValidatorContext context) {

        // 値が null の場合、正常値と判定する。
        if(value == null) {
            return true;
        }

        return (min.compareTo(value) <= 0 && max.compareTo(value) >= 0);
    }

    /**
     * アノテーションのパラメータの整合性チェック。
     */
    private void validatePrameters() {

        if(min.compareTo(max) > 0) {
            throw new IllegalArgumentException("parameters min <= max.");
        }
    }
}
```

アノテーションのクラス

アノテーションを付加するフィールドのクラスタイプ。

フィールドの定義した際に、引数で指定した値などを処理します。

実際の入力値検証のロジックを定義します。
True の場合正常値として判定するようにします。

【エラーメッセージの定義】

- ValidationMessages.properties にエラーメッセージを追加します。
- エラーコードのキーは、アノテーションの引数 message で指定したものになります。
- 置換文字として、アノテーションの引数(例えば{min}、{max}）が利用可能です。

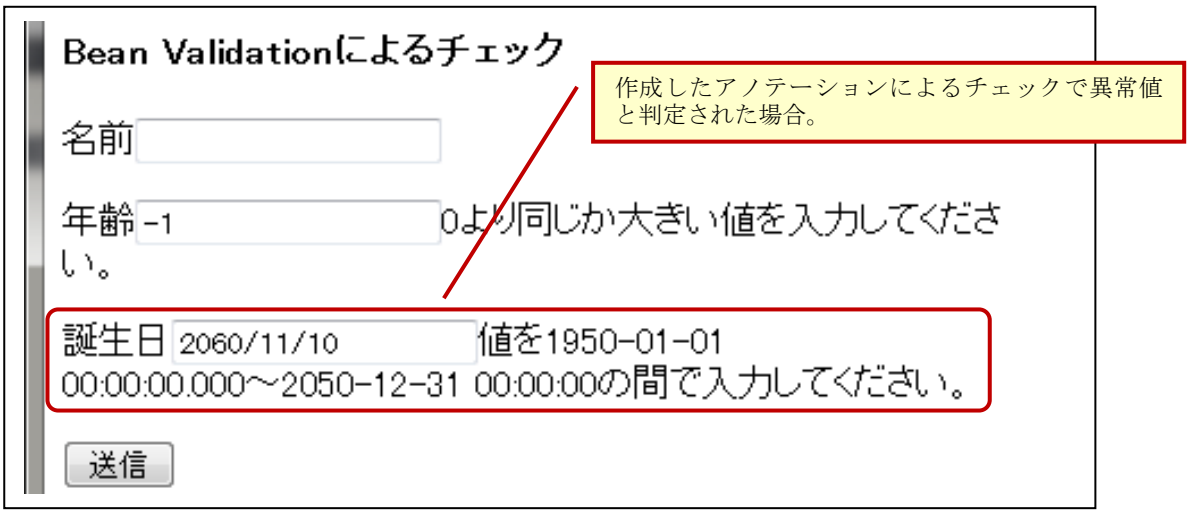
```
## 独自のアノテーション
sample.web.validator.hibernatevalidator.DateRange.message=値を{min}～{max}の間で入力してください。
```

【Command の例】

- 作成したアノテーションを Command のプロパティに付加します。

```
public class Sample6Command implements Serializable {

    @NotNull
    @Length(max=5)
    @Pattern(regexp="[a-z]*")
    private String name;

    @Min(0)
    @Max(200)
    private Integer age;

    // 独自のアノテーション
    @DateRange(min="1950-01-01 00:00:00.000", max="2050-12-31 00:00:00")
    private Date birthDay;

   // getter、setter などは省略
}
```

【ブラウザの表示】



7.4. OVal（Object Validation framework）を利用した入力値検証

Bean Validation(JSR-303)と互換性はあり、同じアノテーションが利用できます。また、カスタマイズ性に優れており、条件付きチェック（例：A という項目が null の場合に B を検証する）が用意されています。特徴として、以下の項目が挙げられます。

- アノテーションのみで設定し、ロジックを排除することで、コード量を減らすことができる。
- Bean Validation と同様、アノテーションで記述するため、最大値など条件がプロパティファイルや DB から取得するようなことはできない。
- ロジックによるチェックも併用できケース・バイ・ケースで記述できる。Struts の ActionForm の 1 つである「ValidateForm」に近い構造を持ちます。また、アノテーションによる条件付きチェックが可能。

Bean Validation(Hibernate Validator)の欠点をカバーしたもので、Bean Validation よりも OVal を使用することをお勧めします。

7.4.1. Oval の準備

【pom.xml】

- 依存ライブラリとして、OVal を追加します。

```
<project>
    ・・・省略
    <dependencies>
        <!-- validator -->
        <dependency>
            <groupId>net.sf.oval</groupId>
            <artifactId>oval</artifactId>
            <version>1.80</version>
        </dependency>
        <!-- Bean Validation のアノテーションを使用する場合
        <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
            <version>1.0.0.GA</version>
        </dependency> -->
        <!-- JPA のアノテーションを使用する場合
        <dependency>
            <groupId>javax.persistence</groupId>
            <artifactId>persistence-api</artifactId>
            <version>1.0</version>
        </dependency> -->
        <!-- XML による設定を使用する場合
        <dependency>
            <groupId>com.thoughtworks.xstream</groupId>
            <artifactId>xstream</artifactId>
            <version>1.4.1</version>
        </dependency> -->
    </dependencies>
    ・・・省略
</project>
```

必要な場合に追加します。

【servlet-context.xml の編集】

- Spring MVC 形式の OVal の Validator「net.sf.oval.integration.spring.SpringValidator」を定義します。
- 基本的にはこの設定のみで動作しますが、「AnnotationMethodHandlerAdapter」の Bean を独自に設定している場合、動作しない場合があり、さらに対策が必要になります。
  詳細は、「7.3.5　こんなときは：Bean Validation がうまく動作しない場合」を参照してください。
  ※Bean Validator と設定は同じです。
- 検証方法の定義方法を指定します。指定するには、「net.sf.oval.configuration.Configure」インタフェースを持つクラスをインジェクションします(「表 7.12」を参照)。
  今回は OVal のアノテーションを使用するので「AnnotationConfigure」を指定します。

```
<beans>
    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven/>

    <!-- OVal 用の Validator -->
    <bean id="ovalValidator" class="net.sf.oval.integration.spring.SpringValidator">
        <property name="validator">
            <bean class="net.sf.oval.Validator">
                <constructor-arg>
                    <list>
                        <!-- OVal のアノテーションを使用する場合 -->
                        <bean class="net.sf.oval.configuration.annotation.AnnotationsConfigurer"/>

                        <!-- Bean Validation のアノテーションを使用する場合 -->
                        <!-- <bean
class="net.sf.oval.configuration.annotation.BeanValidationAnnotationsConfigurer"/> -->

                        <!-- EJB3 JPA のアノテーションを使用する場合 -->
                        <!-- <bean class="net.sf.oval.configuration.annotation.JPAAnnotationsConfigurer"/> -->

                        <!-- XML による設定を使用する場合 -->
                        <!-- <bean class="net.sf.oval.configuration.xml.XMLConfigurer">
                            <constructor-arg
type="java.io.InputStream" value="classpath:com/acme/OValConfiguration.xml" />
                        </bean> -->
                    </list>
                </constructor-arg>
            </bean>
        </property>
    </bean>
</beans>
```

Bean Validation の Validator を名称が重複しないように「ovalValidator」と付けます。

リスト形式なので、同時に複数の Configure を指定できます。

表 7.12　OVal の Validator に設定可能な Configure の種類

No.	クラス	説明	参照先
1	net.sf.oval.configuration.annotation.AnnotationsConfigurer	OVal 独自のアノテーションを使用します。 デフォルトで設定されれています。	7.4.2
2	net.sf.oval.configuration.annotation.BeanValidationAnnotationsConfigurer	Bean Validation のアノテーションを使用します。 使用するには、Bean Validation のライブラリが必要になります。 対応するアノテーションは OVal に用意されています。	7.4.4
3	net.sf.oval.configuration.annotation.JPAAnnotationsConfigurer	EJB3 JPA のアノテーションを使用します。 使用するには､JPA のライブラリが必要になります。 対応するアノテーションは OVal に用意されています。	7.4.5
4	net.sf.oval.configuration.xml.XMLConfigurer	XML による設定を行います。 使用する際には、別ライブラリ XStream が必要となります。 Struts1.X で使用されていた Commons -Validator に近いものです。	7.4.6
5	net.sf.oval.configuration.pojo.POJOConfigurer	手動による設定を行います。XML による設定を Java 記述しますが、Spring で使用するとなると結局 XML で記述することになり、XMLConfigure と変わりません。 Spring で OVal を使用する場合は、通常、POJOConfigure は使用しません。	―

7.4.2. OVal Validator を使用する

【Command の作成】

- アノテーションを Command のプロパティに付与します。
  Bean Validation と基本的な使い方は同じです。
- 使用可能なアノテーションは、「7.4.3　Oval のアノテーション一覧」を参照してください。
- OVal は Bean Validation とは異なり、引数「when」によりチェック条件を設定できます。
  詳細は、「7.4.10.2　引数「when」」を参照してください。

```
import java.io.Serializable;
import java.util.Date;

import net.sf.oval.constraint.DateRange;
import net.sf.oval.constraint.Length;
import net.sf.oval.constraint.MatchPattern;
import net.sf.oval.constraint.NotNull;
import net.sf.oval.constraint.Range;

import org.apache.commons.lang.builder.ToStringBuilder;

public class OvalSample1Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    @NotNull
    @Length(max=5)
    @MatchPattern(pattern="[a-z]*")
    private String name;

    @Range(min=1, max=200, when="groovy:_this.name != null && _this.name.length() > 0")
    private Integer age;

    @DateRange(format="yyyy/MM/dd", min="2000/01/01", max="tomorrow")
    private Date birthDay;

    public OvalSample1Command() {

    }

    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }

    // getter、setter は省略

}
```

Bean Validation と同様に、通常は Command のプロパティにアノテーションを付加します。

チェックを実行する条件を記述します。
この場合は、「プロパティ name に値が入力された場合にチェックする」。

【Controller の作成】

- Bean Validation と同様に、OVal の Spring 用の Validator を WebDataBinder に設定します。
  共通の DataBinder で設定している場合は必要ありません。
- 検証対象の Command にアノテーション「@Valid」を付与します。
  自動的に入力値の検証が実行されます。

```
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.annotation.Resource;
import javax.validation.Valid;

import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.Validator;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("/oval/sample1")
public class OvalSample1Constroller {

    @Resource(name="ovalValidator")
    private Validator validator;

    @InitBinder("sample1Command")
    protected void initBinder(WebDataBinder binder) {
        // validator の設定
        binder.setValidator(validator);

        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
        dateFormat.setLenient(false);

        // 型を指定した Bind 設定
        binder.registerCustomEditor(Date.class, "birthDay", new CustomDateEditor(dateFormat, true));

    }

    @ModelAttribute("sample1Command")
    public OvalSample1Command createInitCommand() {
        OvalSample1Command command = new OvalSample1Command();
        return command;
    }

    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        OvalSample1Command command = createInitCommand();
```

OVal 用の Spring Validator をインジェクションします。

Validator を WebDataBinder に設定します。

```
        command.setAge(0);
        command.setBirthDay(new Date());
        model.addAttribute("sample1Command", command);

    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(
            @ModelAttribute("sample1Command") @Valid OvalSample1Command command,
            BindingResult bindingResult) {

        System.out.println(command.toString());

        // エラーがある場合、自画面遷移する
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;

    }

}
```

チェック対象の Command に@Valid アノテーションを付加します。

【JSP の作成】

- 通常の JSP と同じです。

```
<h4>OVal によるチェック</h4>
<form:form modelAttribute="sample1Command" action="${appUrl}/oval/sample1.html" method="post">

    <p>
        <form:label path="name">名前</form:label>
        <form:input path="name" />
        <form:errors path="name" cssClass="errors" />
    </p>
    <p>
        <form:label path="age">年齢</form:label>
        <form:input path="age" />
        <form:errors path="age" cssClass="errors" />
    </p>
    <p>
        <form:label path="birthDay">誕生日</form:label>
        <form:input path="birthDay" />
        <form:errors path="birthDay" cssClass="errors" />
    </p>

    <input type="submit" name="check1"/>
</form:form>
```

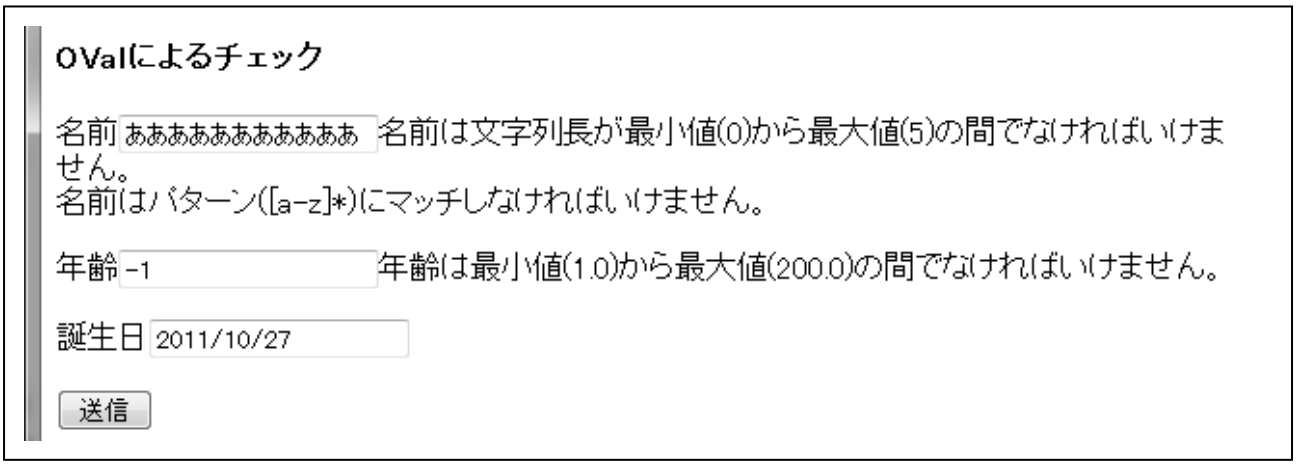
【メッセージの作成】

- OVal はライブラリの oval-xxx.jar に標準のメッセージファイルが格納されています。日本語のメッセージファイルもあり、パッケージ「**net.sf.oval**」以下の「Message_ja.properties」に格納されています。
- 他のメッセージに変更したい場合は、アノテーションの引数「message」に値を設定するか、「7.4.8　OVal のエラーメッセージのカスタマイズ」の方法を取ってください。
- メッセージのサンプルを「図 7.5」に示します。
  メッセージのキーは、「**アノテーションのクラスパス.violated**」となっています。
- メッセージ中で使用可能な共通変数として、{context}、{invalidValue}があります。
  他の変数として、アノテーションの引数で指定したものが使用できます。
  - 共通変数の{context}は、初期設定ではクラス名が入ります。変更する場合は、「7.4.8　OVal のエラーメッセージのカスタマイズ」に示す方法を取ってください。

```
## 共通で使用可能な変数
# {context} = フィールド名
# {invalidValue} = フィールドの値

## メッセージの定義
net.sf.oval.constraint.DateRange.violated={context}が最小値({min})から最大値({max})の間にありません。
net.sf.oval.constraint.Length.violated={context}は文字列長が最小値({min})から最大値({max})の間でなければいけません。
net.sf.oval.constraint.MatchPattern.violated={context}はパターン({pattern})にマッチしなければいけません。
net.sf.oval.constraint.NotNull.violated={context}はヌル値ではいけません。
net.sf.oval.constraint.Range.violated={context}は最小値({min})から最大値({max})の間でなければいけません。

## コンテキスト（フィールド名）の値
label.field.name=名称
label.field.age=年齢
label.field.birthday=誕生日
```

図 7.5　Messages_ja.properties の中身

【ブラウザの表示】



7.4.3. Oval のアノテーション一覧

OVal で使用可能なアノテーション一覧を「表 7.14　OVal のアノテーション一覧」に示します。また、共通の引数を「表 7.13　OVal のアノテーションの共通の引数」に示します。

- OVal のアノテーションは、パッケージ「net.sf.oval.constraint」に格納されています。
  検証を行うチェックロジックも同パッケージに格納されており、「XXXCheck」（XXX はアノテーション名）という名称で統一されています。

表 7.13　OVal のアノテーションの共通の引数

No.	クラス型 引数(※1)	説明
1	ConstaintTarget[] appliesTo	リスト、マップ、配列に対して、検証対象の部分を明示します。 リスト、マップ、配列以外に指定した場合、無視されます。 列挙型「ConstraintTarget」で指定し、次の値が設定できます。 ・「CONTEINER」：リスト、マップ、配列の自身を対象とします。 ・「VALUES」：リスト、マップ、配列の各要素を対象とします。 ・「KEYS」：マップの各キーを対象とします。 初期値は、アノテーションにより異なりますが、通常は、「VALUES」です。
2	String errorCode	エラーコード。内部的な処理を行う場合、どのアノテーションでチェックされたかどうか見分けるために付けます。通常は変更しません。 初期値は、アノテーションのクラスパス。 例)@Assert の場合、「net.sf.oval.constraint.Assert」。
3	String message	入力値の検証の結果、異常値と判定した場合のメッセージキー。 メッセージキー名でプロパティファイルに定義しておくと、そのメッセージが表示される。 特別なエラーメッセージを表示したい場合に変更します。 初期値は、「アノテーションのクラスパス.violated」 例)@Assert の場合、「net.sf.oval.constraint.Assert.violated」
4	String[] profiles	制約に名前を付け、項目間でグルーピングし、有効、無効を指定することができる。 初期値は「default」。複数設定できる。 Validator のメソッド「#enableProfile(プロファイル名)」「disableProfile(プロファイル名)」で一時的に無効化、有効化することができる。 Spring MVC では使う機会はないと思います。
5	int severity	重要度を定義する。 特に機能として特別な処理はなく、検証結果のエラーオブジェクト

		「ConstraintViolation」に情報として渡されて、「#getSeveiry()」で取得できる。 メッセージを表示する際の優先度として利用したりする。 初期値は、「0」。
6	String target	検証対象のフィールドが JavaBean やリストなどの場合、JabaBean のプロパティやリストの各項目に対して、検証対象を絞り込むことができる。 初期値は、空文字。 例) フィールドの変数名が“hoge”の場合。 ・「hoge」: 現在のフィールドを示す。 ・「hoge.id」: hoge 中の階層化されたプロパティ“id”を指す。 ・「hoge[0]」: hoge がリストなどの場合、“0 個目の要素”を指す。 ・「jxpath:hoge/id」: JXPath の形式で指定できる。 使用する際には、ライブラリ「JXPath」が必要になります。。
7	String when	フィールドを検証する際の条件を指定できる。詳細は、「7.4.10　条件付き」を参照。 初期値は空文字。 変数として次のものが使用できる。 ・「_this」: 検証対象の JavaBean、Command を示す。 ・「_value」: 検証対象フィールドの値を示す。 様々な言語で記述できるが、そのライブラリも必要。 例)「groovy:_this.amount > 0」

※1 引数は全てオプションです。

表 7.14　OVal のアノテーション一覧

No.	アノテーション	フィールド型(※1)	説明
1	@Assert	Object java.util.Collection java.util.Map 配列	言語を指定し、任意の条件を記述する。 [引数]String expr : 必須。正常値(true)となる条件式を記述します。 [引数]String lang : 必須。条件式の言語を指定する。「表 7.20」に示す値が使用可能です。
2	@AssertConstraintSet	Object	他に定義されている複数の制約(検証用アノテーションなど)の集合を全て満たすかチェックします。 [引数]String id : 必須。制約の集合の ID。 XMLConfigurer で定義した制約の ID を指定します。
3	@AssertTrue	boolean 対応するラッパークラス	True であることをチェックします。

4	@AssertFalse	boolean 対応するラッパークラス	False であることをチェックします。
5	@AssertNull	Object	null であることをチェックします。
6	@NotNull	Object	Null でないかどうかチェックします。
7	@EqualsToField	Object プリミティブ型	指定したフィールドと同じ値であることをチェックします。パスワード、メールなどの再入力項目などに使用されます。 値が null の場合は正常値と判定します。 [引数]String value : 必須。フィールド名を指定します。
8	@NotEqualToField	Object プリミティブ型	指定したフィールドと異なる値であることをチェックします。 値が null の場合は正常値と判定します。 [引数]String value : 必須。フィールド名を指定します。
9	@DateRange	java.util.Date	指定した範囲内に日付があるかチェックします。 値が null の場合は正常値と判定します。 [引数]String format：min,max の引数の書式を指定します。SimpleDateFormat で指定可能な値である必要があります。 [引数]String min：最小日付を、引数「format」で指定した書式で定義します。 [引数]String max：最大日付を、引数「format」で指定した書式で定義します。 ※引数 min、max の値として、次の変数名が使用可能です。「now」「today」「yesterday」「tomorrow」。
10	@Past	java.util.Date	現在の日付より過去かどうかチェックします。 値が null の場合は正常値と判定します。
11	@Future	java.util.Date	現在の日付より未来かどうかチェックします。 値が null の場合は正常値と判定します。
12	@Min	short,int,long float,double 対応するラッパークラス BigDecimal,BigInteger	指定した数値以上かチェックします。 値が null の場合は正常値と判定します。 [引数]double value：必須。下限値を指定します。
13	@Max	short,int,long float,double 対応するラッパークラス BigDecimal,BigInteger	指定した数値以下かチェックします。 値が null の場合は正常値と判定します。 [引数]double value：必須。上限値を指定します。

14	@Range	short,int,long float,double 対応するラッパークラス BigDecimal,BigInteger	指定した範囲内の数値かチェックします。 値が null の場合は正常値と判定します。 [引数]double min：下限値を指定します。 [引数]double max：上限値を指定します。
15	@NotNegative	short,int,long float,double 対応するラッパークラス BigDecimal,BigInteger	負の数でない(=0 以上)の数値であるかチェックします。 値が null の場合は正常値と判定します。
16	@Digits	float,double 対応するラッパークラス BigDecimal,BigInteger	数値の整数部、小数部が指定した桁数の範囲内にあるかチェックします。 値が null の場合は正常値と判定します。 [引数]int minInteger：整数部の最小桁数。初期値は「0」。 [引数]int maxInteger：整数部の最大桁数。初期値は「2147483647」。 [引数]int minFraction：小数部の最小桁数。初期値は「0」。 [引数]int minFraction：小数部の最大桁数。初期値は「2147483647」。
17	@NotBlank	String	文字列が空白スペースでないかチェックします。 @NotEmpty とは違い、null、半角スペースのみの場合も正常値と判断します。 値が null の場合は正常値と判定します。
18	@NotEmpty	String	文字列が空文字(=サイズが 0)かどうかチェックします。 値が null の場合は正常値と判定します。
19	@NotEqual	String	指定した文字列に一致しないかどうかチェックします。 値が null の場合は正常値と判定します。 [引数]String value：必須。比較対象の文字列。
20	@Length	String	指定した範囲以内の文字長かチェックします。 値が null の場合は正常値と判定します。 [引数]int min：文字長の下限値を指定します。初期値は「0」。 [引数]int max：文字長の上限値を指定します。初期値は「2147483647」。
21	@MinLength	String	指定した文字長以上かチェックします。

			値が null の場合は正常値と判定します。 [引数]int value：必須。文字長の下限値を指定します。
22	@MaxLength	String	指定した文字長以下かチェックします。 値が null の場合は正常値と判定します。 [引数]int value：必須。文字長の上限値を指定します。
23	@AssertURL	String	URL の形式として正しいかチェックします。 値が null の場合は正常値と判定します。 [引数] boolean connect：URL が実際に存在するかネットワーク接続し検証します。初期値は「false」。 [引数] AssertURLCheck.URIScheme[] permittedURISchemes：許可するスキーマを絞り込みます。初期値｛HTTP、HTTPS、FTP｝の 3 つです。
24	@Email	String	RFC822 のメールのパターンとして正しいかチェックします。 値が null の場合は正常値と判定します。 [引数]boolean allowPersonalName：個人名(=Personal Name、例)個人名@ホスト名)を含んでいるかチェックします。初期値は「true」。
25	@MatchPattern	String	指定した正規表現を満たすかチェックします。 値が null の場合は正常値と判定します。 [引数]String[] pattern：必須。正規表現。複数指定可能です。 [引数]boolean matchAll：複数していた場合、全ての正規表現を満たすか設定します。初期値は「true」。 [引数]int[] flags：正規表現 Pattern の設定値。複数指定可能。初期値は「0」。指定可能な値は以下の通り。 「Pattern.INSENSITIVE」 「Pattern.MULTILINE」「Pattern.DOTALL」 「Pattern.UNICODE_CASE」 「Pattern.CANON_EQ」。
26	@NotMatchPattern	String	指定した正規表現を満たさないかチェックします。 値が null の場合は正常値と判定します。 [引数]String[] pattern：必須。正規表現。複数指定可能した場合は、何れかを満たさなければ正常値と判定する。 [引数]int[] flags：正規表現 Pattern の設定値。複数指定可能。初期値は「0」。指定可能な値は

			「@MatchPattern」を参照。
27	@HasSubstring	String	指定した文字を含むどうかチェックする。 値が null の場合は正常値と判定します。 [引数]String value：必須。包含すべき文字列を指定します。 [引数]boolean ignoreCase：大文字・小文字の区別をしないかどうか設定します。初期値は「false」。
28	@MemberOf	String	指定した文字列集合の何れかに一致するかチェックします。 値が null の場合は正常値と判定します。 [引数]String[] value：必須。包含すべき文字列の集合を指定します。 [引数]boolean ignoreCase：大文字・小文字の区別をしないかどうか設定します。初期値は「false」。
29	@NotMemberOf	String	指定した文字集合の全てに一致しないかチェックします。 値が null の場合は正常値と判定します。 [引数]String[] value：必須。包含すべきではない文字列の集合を指定します。 [引数]boolean ignoreCase：大文字・小文字の区別をしないかどうか設定します。初期値は「false」。
30	@Size	java.util.Colletion java.util.Map 配列 String	リストなどのサイズが範囲内にあるかチェックします。文字列もチェック可能ですが、@Length を使用することをお勧めします。 値が null の場合は正常値と判定します。 [引数]int min：リストなどのサイズの下限値を指定します。初期値は「0」。 [引数]int max：リストなどのサイズの上限値を指定します。初期値は「2147483647」。
31	@MinSize	java.util.Colletion java.util.Map 配列 String	リストなどのサイズが指定した値以上であるかチェックします。文字列もチェック可能ですが、@MinLength を使用することをお勧めします。 値が null の場合は正常値と判定します。 [引数]int value：必須。リストなどのサイズの下限値を指定します。
32	@MaxSize	java.util.Colletion java.util.Map	リストなどのサイズが指定した値以下であるかチェックします。文字列もチェック可能ですが、@MaxLength

		配列 String	を使用することをお勧めします。 値が null の場合は正常値と判定します。 [引数]int value：必須。リストなどのサイズの上限値を指定します。
33	@InstanceOf	Object	指定したクラス／インタフェースを全て持つかかチェックします。Java の「instanceof」の効果と同じです。 値が null の場合は正常値と判定します。 [引数]Class<?>[] value：必須。実装すべきクラス／インタフェースのクラスオブジェクトを指定します。
34	@InstanceOfAny	Object	指定したクラス／インタフェースの何れかを持つかチェックします。Java の「instanceof」の効果と同じです。 値が null の場合は正常値と判定します。 [引数]Class<?>[] value：必須。実装すべきクラス／インタフェースのクラスオブジェクトを指定します。
35	@AssertValid	Object	ネストしたオブジェクト(JavaBean)をチェックします。 検証対象のオブジェクトに対して、OVal のアノテーションの設定をしている必要があります。
36	@CheckWith	Object	CheckWithCheck.SimpleCheck を実装したインナークラスにてチェックする。 詳細は、「7.4.13.1　@CheckWith」を参照。 [引数]CheckWithCheck.SimpleCheck value：必須。CheckWithCheck.SimpleCheck を実装したクラス。 [引数]boolean ignoreIfNull：フィールドの値が null の場合にスキップするかどうか。初期値は「true」。
37	@ValidateWithMethod	Object プリミティブ型	引数「methodName」で指定したメソッドによりチェックを行う。 [引数]String methodName：必須。検証処理を実装したメソッド名を指定します。 [引数]Class<?> parameterType：必須。検証処理を実装したメソッドの引数を指定します。フィールドの型名と合わせる必要があります。
38	@NoSelfReference	Object	自身のオブジェクト(=検証対象の JavaBean)を参照していないかチェックします。循環参照をチェックする際に使用します。

7.4.4. Bean Validation のアノテーションを使用する

OVal は、Hibernate Validator と同様に、「Bean Validation（JSR-303）」の実装の１つとして、アノテーションをそのまま使用できます。一部、実装が Hibernate Validator とは異なったりするので注意が必要です。

7.4.4.1. 準備

【pom.xml の編集】

- Bean Validation のライブラリのみ追加します。
  「7.3.1　Bean Validation の準備」とは異なり、Hibernate Validator は必要ありません。

```
<project>
    ・・・省略
    <dependencies>
        <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
            <version>1.0.0.GA</version>
        </dependency>
    </dependencies>
    ・・・省略
</project>
```

【servlet-context.xml】

- OVal 用の Validator に、Bean Validation 用の Configure「BeanValidationAnnotationsConfigure」を追加します。

```
<beans>
    ・・・省略
    <!-- OVal 用の Validator -->
    <bean id="ovalValidator" class="net.sf.oval.integration.spring.SpringValidator">
        <property name="validator">
            <bean class="net.sf.oval.Validator">
                <constructor-arg>
                    <list>
                        <!-- OVal のアノテーションを使用する場合 -->
                        <bean class="net.sf.oval.configuration.annotation.AnnotationsConfigurer"/>

                        <!-- Bean Validation のアノテーションを使用する場合 -->
                        <bean
class="net.sf.oval.configuration.annotation.BeanValidationAnnotationsConfigurer"/>

                    </list>
                </constructor-arg>
            </bean>
        </property>
    </bean>
    ・・・省略
</beans>
```

Bean Validation 用の Configure を追加します。

7.4.4.2.　使用する

Bean Validation のアノテーションを使用する場合、Hibernate Validator 経由で使用するときと使い方は変わりません。

- OVal 経由で Bean Validation のアノテーションを使用する場合、内部では検証ロジックとして OVal のアノテーションに対応するものを使用します(BeanValidationAnnotationsConfigure で変換されます)。（「表 7.15」を参照)。
- Hibernate Validator のアノテーションは使用できないので、「表 7.16」に示す OVal に対応するアノテーションをご使用ください。
  ただし、Hibernate Validator の Configure を独自に実装すれば使用できます。

7.4.4.3.　Bean Validation と OVal のアノテーションとの比較

Bean Validation から OVal 用のアノテーションに移行を行う際には、一部機能の違いがあるため、注意が必要です。それぞれのアノテーションの対応を「表 7.15」「表 7.16」に示します。Bean Validattion のアノテーションの詳細は、「7.3.3　Bean Validation のアノテーションの一覧」を参照してください。

表 7.15　Bean Validation と OVal のアノテーションの対応

No.	Bean Validation(※1)	OVal(※2)	違いの説明
1	@NotNull	@NotNull	基本的な機能に違いはありません。
2	@Null	@AssertNull	基本的な機能に違いはありません。
3	@AssertFalse	@AssertFalse	基本的な機能に違いはありません。
4	@AssertTrue	@AssertTrue	基本的な機能に違いはありません。
5	@DecimalMin	@Min	基本的な機能に違いはありません。
6	@DecimalMax	@Max	基本的な機能に違いはありません。
7	@Digits	@Digits	OVal の方は、整数部、小数部にそれぞれに最小桁数を指定できます。
6	@Min	@Min	基本的な機能に違いはありません。OVal の方は、小数(double など)にも使用できるため、最小値は double 型で指定します。
7	@Max	@Max	基本的な機能に違いはありません。OVal の方は、小数(double など)にも使用できるため、最大値は double 型で指定します。
8	@Size	@Size	基本的な機能に違いはありません。
9	@Pattern	@MatchPattern	基本的な機能に違いはありません。 OVal は、逆の検証を行う「@NotMatchPattern」があ

			ります。
10	@Past	@Past	基本的な機能に違いはありません。
11	@Futrue	@Future	基本的な機能に違いはありません。

※1 Bean Validation のアノテーションは、パッケージ「javax.validation.constraints」に格納されています。

※2 OVal のアノテーションは、パッケージ「net.sf.oval.constraint」に格納されています。

表 7.16　Hibernate Validator と OVal のアノテーションとの対応

No.	Hibernate Validator(※1)	OVal(※2)	違いの説明
1	@NotBlank	@NotBlank	OVal の場合、値が null の場合正常値として判定します。同じ動作をさせるためには、「@NotNull」も付与する必要があります。
2	@NotEmpty	@NotEmpty	OVal の場合、値が null の場合正常値として判定します。同じ動作をさせるためには、「@NotNull」も付与する必要があります。
3	@Length	@Length	基本的な機能に違いはありません。
4	@Range	@Range	基本的な機能に違いはありません。
5	@CreditCardNumber	—	対応するものはありません。
6	@Email	@Email	OVal の方は RFC-2822 に特に沿っているものではなく、独自の正規表現でチェックします。
7	@SafeHtml	—	対応するものはありません。
6	@ScriptAssert	—	対応するものはありません。
7	@URL	@AssertURL	OVal の方は、プロトコルやポートなど細かな条件で指定はできません。

※1 Hibernate Validator のアノテーションは、パッケージ「org.hibernate.validator.constraints」に格納されています。

※2 OVal のアノテーションは、パッケージ「net.sf.oval.constraint」に格納されています。

7.4.5. EJB3 JPA のアノテーションを使用する

OVal は、JPA のアノテーションをそのまま使用できます。しかし、一部実装に JPA とは異なったりするので注意が必要です。

7.4.5.1. 準備

【pom.xml の編集】

- JPA の Psersistence API のライブラリのみ追加します。

```
<project>
    ・・・省略
    <dependencies>
        <!-- JPA のアノテーションを使用する場合 -->
        <dependency>
            <groupId>javax.persistence</groupId>
            <artifactId>persistence-api</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>
    ・・・省略
</project>
```

【servlet-context.xml】

- OVal 用の Validator に、JPA 用の Configure「JPAAnnotationsConfigurer」を追加します。

```
<beans>
    ・・・省略
    <!-- OVal 用の Validator -->
    <bean id="ovalValidator" class="net.sf.oval.integration.spring.SpringValidator">
        <property name="validator">
            <bean class="net.sf.oval.Validator">
                <constructor-arg>
                    <list>
                        <!-- OVal のアノテーションを使用する場合 -->
                        <bean class="net.sf.oval.configuration.annotation.AnnotationsConfigurer"/>

                        <!-- EJB3 JPA のアノテーションを使用する場合 -->
                        <bean class="net.sf.oval.configuration.annotation.JPAAnnotationsConfigurer"/>

                    </list>
                </constructor-arg>
            </bean>
        </property>
    </bean>
    ・・・省略
</beans>
```

JPA 用の Configure を追加します。

7.4.5.2. JPA と OVal のアノテーションとの比較

JPA から OVal 用のアノテーションに移行を行う際には、一部機能の違いがあるため、注意が必要です。それぞれのアノテーションの対応を「表 7.17」に示します。

表 7.17　EJB3 JPA と OVal のアノテーションとの比較

No.	JPA (※1)	OVal(※2)	違いの説明
1	@Basic(optional=false)	@NotNull	基本的な機能に違いはありません。
2	@OneToOne(optional=false)	@NotNull	基本的な機能に違いはありません。
3	@OneToOne	@AssertValid	基本的な機能に違いはありません。
4	@OneToMany	@AssertValid	基本的な機能に違いはありません。
5	@ManyToOne(optional=false)	@NotNull	基本的な機能に違いはありません。
6	@ManyToOne	@AssertValid	基本的な機能に違いはありません。
7	@Column(nullable=false)	@NotNull	基本的な機能に違いはありません。
6	@Column(length=5)	@Length	基本的な機能に違いはありません。 OVal の方は最小の文字長も指定できます。

※1 JPA のアノテーションは、パッケージ「javax.persistence」に格納されています。

※2 OVal のアノテーションは、パッケージ「net.sf.oval.constraint」に格納されています。

7.4.6. XML による設定を使用する

7.4.6.1. 準備

【pom.xml の編集】

- XML を処理する XStream のライブラリを追加します。

```
<project>
    ・・・省略
    <dependencies>
        <!-- XML による設定を使用する場合 -->
        <dependency>
            <groupId>com.thoughtworks.xstream</groupId>
            <artifactId>xstream</artifactId>
            <version>1.4.1</version>
        </dependency>
    </dependencies>
    ・・・省略
</project>
```

【servlet-context.xml】

- OVal 用の Validator に、XML 用の Configure「XMLConfigurer」を追加します。
- 設定ファイルの場所も指定します。
  - クラスパスで指定したい場合は、「classpath:」をパスの前に付けます。
  - システムファイルで指定したい場合は、「file:」をパスの前に付けます。

```
<beans>
    ・・・省略
    <!-- OVal 用の Validator -->
    <bean id="ovalValidator" class="net.sf.oval.integration.spring.SpringValidator">
        <property name="validator">
            <bean class="net.sf.oval.Validator">
                <constructor-arg>
                    <list>
                        <!-- OVal のアノテーションを使用する場合 -->
                        <bean class="net.sf.oval.configuration.annotation.AnnotationsConfigurer"/>

                        <!-- XML による設定を使用する場合 -->
                        <bean class="net.sf.oval.configuration.xml.XMLConfigurer">
                            <constructor-arg type="java.io.InputStream"
value="classpath:prop/OValConfiguration.xml" />
                        </bean>
                    </list>
                </constructor-arg>
            </bean>
        </property>
    </bean>
    ・・・省略
</beans>
```

XML 用の Configure を追加します。

7.4.6.2. XMLConfigurte を使用する

【OValConfiguration.xml の作成】

- タグ<constraintSet id="制約名">で定義した制約は、XML 内でけでなく、アノテーション「@AssertConstrainSet(id="制約名")」として呼び出すことができます。
  - タグの要素に、各チェックしたい制約を定義します。アノテーションの名称に一致します。
    例) @NotNull ⇒ <notNull/>
  - XML スキーマが認識されれば、Eclipse の XML エディタにて自動的にタグを補間します。
- タグ<class type="チェック対象のオブジェクトのクラス名">にて、チェック対象のオブジェクトの制約を定義します。
  - 属性「orverwrite="true|false"」にて、継承元やアノテーションで定義されている制約を上書きするかを設定します。「false」にした場合、制約を引き継ぎ、かつ XML で定義した制約も追加することができます。
    初期値は「false」です。
- タグ<field name="フィールド名">にて、オブジェクトのフィールド（プロパティ）に対する制約を定義します。
  - 属性「orvewrite="true|false"」にて、継承元やアノテーションで定義されている制約を上書きするかを設定します。
    タグ<class overwrite="false">とし設定されている場合に、フィールドごと上書きするか設定します。初期値は「false」です。
- タグ<method name="メソッド名">にて、メソッドに対する制約を定義します。Getter メソッド場合の戻り値に対するチェックなので、さらにタグ<returnValue>を記述し、その中に制約を定義していきます。
- アノテーションと XML の入力値検証のそれぞれで異常値と判定された場合、エラーメッセージの順番（チェックされる順番）は、「servlet-context.xml」で Configure 記述した順番になります。
  例）AnnotationsConfigurer⇒XMLConfigurer の順に定義した場合、アノテーション側のチェック⇒XML 側のチェックとなります。

```
<?xml version="1.0" encoding="UTF-8"?>
<oval xmlns="http://oval.sf.net/oval-configuration"
      xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://oval.sf.net/oval-configuration http://oval.sourceforge.net/oval-configuration.xsd">

    <!-- 制約の集合を定義します。 -->
    <constraintSet id="user.name">
        <length max="5"/>
        <matchPattern matchAll="true">
            <pattern pattern="[a-z]*" flags="0"/>
        </matchPattern>
    </constraintSet>

    <!-- クラス「OvalSample6Command」に対する制約を定義します。 -->
    <class type="sample.web.oval.controller.OvalSample6Command" overwrite="false">

        <!-- フィールド「name」に対するチェック -->
        <field name="name" overwrite="false">
            <notEmpty/>
        </field>

        <field name="age" overwrite="true">
            <range min="0" max="100"/>
        </field>

        <!-- メソッド名「getAge()」に対するチェック -->
        <method name="getAge" isInvariant="true">
            <!-- 戻り値に対するチェック -->
            <returnValue>
                <notNegative/>
            </returnValue>
        </method>

        <!-- ネストしたフィールド「memberCard」に対するチェック -->
        <field name="memberCard">
            <assertValid requireValidElements="true"/>
        </field>

    </class>

</oval>
```

制約の集合の定義。アノテーション「@AssertConstraintSet」からも利用できる。

Bean、Command などのチェック対象のオブジェクトに対する制約の定義。

フィールド（プロパティ）に対する制約を定義します。

メソッドの戻り値に対するチェックです。「@IsInvariant」に対応する属性「isInvariant="true"」を設定します。

戻り値に対するチェックなので、<returnValue>の中に制約を記述します。

ネストしたフィールド(Bean)を持つ場合、@AssertValid に対応する<assertValid>を使用します。

7.4.6.3.　独自に作成したアノテーションを利用する

「7.4.14　OVal の独自アノテーションを実装する」で説明している、独自に作成したアノテーションの設定を XMLConfigure にて利用する方法を説明します。

- 「AbstractAnnotationCheck」を継承して作成したクラス、チェックロジッククラス（XXXCheck）をタグ名として設定します。
- 属性は、チェックロジックで定義した Setter メソッドが利用可能です。

```
<field name="age" overwrite="true">
    <sample.web.oval.annotation.DecimalRangeCheck min="0" max="100"/>
</field>
```

7.4.7. POJOConfigure を使用する

POJOCongirue を使用するケースでは XMLConfigure を使用するため、Spring 経由で使用することはありません。

使用例などについては、下記の URL を参照してください。

- POJOConfigure の使用例「http://bitsofwizardry.wordpress.com/tag/pojoconfigurer/」

7.4.8. OVal のエラーメッセージのカスタマイズ

OVal のメッセージ定義を変更する方法は、2 つあります。Spring からは利用し辛いので、SpringValidator を拡張する方式をお勧めします。

また、メッセージの種類として次の 3 つの種類があり、それぞれ設定する必要があります。

- エラーメッセージ。「MessageResolver」の実装したクラスで処理する。
- メッセージ中の項目名などのコンテキスト。「OValContextRenderer」の実装クラスで処理する。
  エラーメッセージ中の**置換文字{context}**が該当します。置換文字{context}をエラーメッセージ中に使用しなければ特に設定する必要はありません。
  初期値は、クラス名.フィールド名とエンドユーザからは意味不明な文字が表示されるので注意してください。

7.4.8.1. システム共通の DataBinder で指定する

Spring MVC の WebDataBinder の共通設定にて、メッセージも登録します。

【GlobalBindingInitializer の設定】

- エラーメッセージは、「net.sf.oval.localization.message.MessageResolver」を、OVal 用の Validator から取得し、ResourceBundle 経由で指定しします。
  - ResourceBundle で指定するため、プロパティファイル名には、ロケール名（例：OvalMessages_ja.properties）を付ける必要があります。
- 項目名などのコンテキストは、「ResourceBundleValidationContextRenderer」のインスタンスを設定します。
  - メッセージ用のプロパティファイルは、チェック対象のオブジェクト（Command、JavaBeans）と同じ場所、同じ名称で記述する必要があります。
  - 「sample.bean.LoginCommand.java」の場合、「sample.bean.LoginCommand_ja.properties」に作成します。
  - メッセージの書式が決まっており、「label.field.フィールド名」（例えば「labe.field.name」）とします。
- この方法は、OVal を標準の Validator として登録すると同時に、メッセージも設定する際に有用ですが、Spring MVC でしか有効ではないのが欠点です。

```
package sample.web;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.ResourceBundle;
import javax.annotation.Resource;
import net.sf.oval.integration.spring.SpringValidator;
import net.sf.oval.localization.message.ResourceBundleMessageResolver;
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.validation.Validator;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebBindingInitializer;
import org.springframework.web.context.request.WebRequest;

public class GlobalBindingInitializer implements WebBindingInitializer {

    @Resource(name="ovalValidator")
    private SpringValidator ovalSpringValiator;

    @Override
    public void initBinder(WebDataBinder binder, WebRequest request) {

        // OVal の Validator を標準 Validator として登録する
        binder.setValidator(ovalSpringValiator);

        // OVal の MessageResolver の設定
        ResourceBundleMessageResolver messageResolver =
            (ResourceBundleMessageResolver) net.sf.oval.Validator.getMessageResolver();
        messageResolver.addMessageBundle(ResourceBundle.getBundle("message/OvalMessages"));

        // OVal の ContextRenderer の設定
        net.sf.oval.Validator.setContextRenderer(ResourceBundleValidationContextRenderer.INSTANCE);

    }
}
```

OVal の Spring 用の Validator のインジェクション。

OVal の MessageResolver 経由でプロパティファイルを設定する。

OVal の MessageResolver 経由でプロパティファイルを設定する。

【servlet-context.xml の編集】

- 作成した、GlobalWebDataBinder を登録します。

  必ず、<mvc:annotation-driven>よりも前に記述します。

```
<beans>
    ・・・省略
    <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
        <property name="cacheSeconds" value="0" />
        <property name="webBindingInitializer">
            <bean class="sample.web.GlobalBindingInitializer" />
        </property>
    </bean>

    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven/>
    ・・・省略
</beans>
```

システム共通の WebBinder の定義を設定します。

7.4.8.2. Spring の MessageSource を使用する

MessageResolver、ContextRenderer に Spring の MessageSource を使用できるクラスを作成します。
詳細は、「7.4.9　SpringValidator を拡張する」を参照してください。

7.4.9. SpringValidator を拡張する

既存の SpringValidator は以下のに示す問題点があり、Spring MVC の入力値検証用 API としては使い辛いいため、拡張して使用する方法を説明します。

表 7.18　既存の SpringValidator の問題点

No.	問題点
ア	プロパティの形式がリスト、マップ、配列の場合、メッセージオブジェクトにインデックスが付与されず、リストなどの要素ごとにエラーメッセージのを表示できない。 例)エラーメッセージのキーの形式：Spring「books[1]」、OVal「books」。
イ	アノテーション「@AssertValid」を使用し、ネストしたオブジェクトを検証した際に、エラーメッセージのキーが階層化されない。 例)エラーメッセージのキーの形式：Spring「company.name」、OVal「name」。
ウ	アノテーション「@IsInvariant」を使用し、Getter メソッドに対して検証した際に、エラーメッセージがグローバルメッセージとなる。 Bean Validation と仕様を合わせるために、Getter メソッドに付与されている場合、フィールドエラーメッセージとして処理すべき。
エ	エラーメッセージ、コンテキストメッセージは、ResourceBundle から取得するようになっているため、Spring の「MessageSource」が使用できない。 また、既存の Validator のメッセージを設定するメソッドは、static メソッドであるので Spring によるインジェクションが困難。
オ	「@Assert」や引数「when」の条件付きチェックにおいて、Spring で予め用意されている Spring Expression Language(SpEL)を使用できるようにする。 SpEL が使用できることで、既に SpEL を設定にて使用している場合、他の EL の学習コスト省くことができる。また、依存するライブラリを減らすことができる。
カ	型変換時のバインドエラーが発生した場合は、値が null として OVal のチェックが実行される。 １つのフィールドに対して複数のエラーが発生している場合、全てのエラーが表示されてしまう。

表 7.19　SpringValidator を拡張するために必要なもの

No.	ファイル名	説明	問題点(※1)
1	OValValidator.java 「7.4.9.1」を参照。	既存の Validator を拡張し、プロパティの形式がリスト、マップ、配列の場合、インデックス情報を保持するよう処理します。	ア
2	IndexedFieldContext.java 「7.4.9.2」を参照。	インデックス情報を保持することができるフィールドのエラー情報。	ア
3	IndexedMethodReturnValudContext.java 「7.4.9.3」を参照。	インデックス情報を保持することができる「@IsInvariant」が付与されたメソッドの戻り値に対するエラー情報。	ア
4	OValSpringValidator.java 「7.4.9.4」を参照。	「SpringValidator」を拡張したクラス。「@AssertValid」を付与し、ネストしたエラーメッセージを処理する。 「@IsInvariant」を付与した Getter メソッドをフィールドエラーとして処理する。 型変換時のバインドエラーが発生しているフィールドに対して、OVal のエラーメッセージを無視する処理を行う。	イ、ウ、エ、カ
5	SpringMessageResolver.java 「7.4.9.5」を参照。	OVal のエラーメッセージを解決するための MessageResolver の実装。 Spring の MessageSource を使用可能にしたもの。	エ
6	SpringValidationContextRenderer.java 「7.4.9.6」を参照。	OVal の項目名などのコンテキストを解決するための OValContextRenderer の実装。 Spring の MessageSource を使用可能にしたもの。	エ
7	ExpressionLanguageSpelImpl.java 「7.4.15」を参照。	スクリプト言語で記述する条件式で「SpEL」を利用するためのクラス。 実装方法は「7.4.15　条件付きチェックに「SpEL」を使用する」を参照。	オ
8	servlet-context.xml 「7.4.9.7」を参照。	作成した Validator に MessageSource を設定する。	―
9	ApplicationContext.xml 「7.4.9.7」を参照。	MessageSource に、OVal 用のエラーメッセージを設定する。	―

※1　「表 7.18　既存の SpringValidator の問題点」に示す問題点に対応しています。

7.4.9.1. 「OValValidator.java」

- 既存の OVal の Validator を拡張子、リスト、マップ、配列などのインデックス、キーをエラー情報に含めるようにします。
- メソッド「checkConstraint(...)」をオーバライドして振る舞いを変更していますが、リスト、マップ、配列の各要素の Context をインデックス付きのクラスに変換する以外は既存のものと処理はまったく同じです。

```
package sample.web.oval;

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.Collection;
import java.util.List;
import java.util.Map;

import net.sf.oval.Check;
import net.sf.oval.ConstraintTarget;
import net.sf.oval.ConstraintViolation;
import net.sf.oval.Validator;
import net.sf.oval.configuration.Configurer;
import net.sf.oval.context.FieldContext;
import net.sf.oval.context.MethodReturnValueContext;
import net.sf.oval.context.OValContext;
import net.sf.oval.exception.OValException;
import net.sf.oval.internal.ContextCache;
import net.sf.oval.internal.util.ArrayUtils;
import net.sf.oval.ogn.ObjectGraphNavigationResult;

/**
 * リスト、マップ、配列などのインデックス情報に対応したクラス。
 *
 */
public class OValValidator extends Validator {

	public OValValidator()
	{
		super();
	}

	public OValValidator(final Collection<Configurer> configurers)
	{
	    super(configurers);
	}

	public OValValidator(final Configurer... configurers)
	{
		super(configurers);
	}

	@Override
	protected void checkConstraint(final List<ConstraintViolation> violations, final Check check,
			Object validatedObject, Object valueToValidate, OValContext context, final String[] profiles,
			final boolean isContainerValue, final boolean ignoreTarget) throws OValException
	{
		if (!isAnyProfileEnabled(check.getProfiles(), profiles)) return;
```

```
        if (!check.isActive(validatedObject, valueToValidate, this)) return;

        final ConstraintTarget[] targets = check.getAppliesTo();

        if (!ignoreTarget)
        {
            String target = check.getTarget();
            if (target != null)
            {
                target = target.trim();
                if (target.length() > 0)
                {
                    if (valueToValidate == null) return;
                    final String[] chunks = target.split(":", 2);
                    final String ognId, path;
                    if (chunks.length == 1)
                    {
                        ognId = "";
                        path = chunks[0];
                    }
                    else
                    {
                        ognId = chunks[0];
                        path = chunks[1];
                    }
                    final ObjectGraphNavigationResult result =
objectGraphNavigatorRegistry.getObjectGraphNavigator(
                            ognId).navigateTo(valueToValidate, path);
                    if (result == null) return;
                    validatedObject = result.targetParent;
                    valueToValidate = result.target;
                    if (result.targetAccessor instanceof Field)
                        context = ContextCache.getFieldContext((Field) result.targetAccessor);
                    else
                        context = ContextCache.getMethodReturnValueContext((Method) result.targetAccessor);
                }
            }
        }

        final Class< ? > compileTimeType = context.getCompileTimeType();

        final boolean isCollection = valueToValidate != null ? //
                valueToValidate instanceof Collection< ? > : //
                compileTimeType != null && Collection.class.isAssignableFrom(compileTimeType);
        final boolean isMap = !isCollection && //
                (valueToValidate != null ? //
                        valueToValidate instanceof Map< ? , ? > : //
                        compileTimeType != null && Map.class.isAssignableFrom(compileTimeType));
        final boolean isArray = !isCollection && !isMap && //
                (valueToValidate != null ? //
                        valueToValidate.getClass().isArray() : //
                        compileTimeType != null && compileTimeType.isArray());
        final boolean isContainer = isCollection || isMap || isArray;

        if (isContainer && valueToValidate != null)
            if (isCollection)
            {
                if (ArrayUtils.containsSame(targets, ConstraintTarget.VALUES)) {
                    int i=0;
                    for (final Object item : (Collection< ? >) valueToValidate) {
                        checkConstraint(violations, check, validatedObject, item, convertIndexContext(context, i),
```

インデックス付きの Context に変換します。

```
profiles, true, true);
                                i++;
                            }
                        }
                    }
                    else if (isMap)
                    {
                        if (ArrayUtils.containsSame(targets, ConstraintTarget.KEYS))
                            for (final Object item : ((Map< ? , ? >) valueToValidate).keySet())
                                checkConstraint(violations, check, validatedObject, item, convertIndexContext(context, item), profiles, true, true);

                        if (ArrayUtils.containsSame(targets, ConstraintTarget.VALUES))
                            for (final Object item : ((Map< ? , ? >) valueToValidate).values())
                                checkConstraint(violations, check, validatedObject, item, convertIndexContext(context, item), profiles, true, true);
                    }
                    else if (ArrayUtils.containsSame(targets, ConstraintTarget.VALUES)) {
                        int i=0;
                        for (final Object item : ArrayUtils.asList(valueToValidate)) {
                            checkConstraint(violations, check, validatedObject, item, convertIndexContext(context, i), profiles, true, true);
                            i++;
                        }
                    }

            super.checkConstraint(violations, check, validatedObject, valueToValidate, context, profiles, isContainerValue, ignoreTarget);
        }

        /**
         * インデックス付きのコンテキストに変換する
         * @param context
         * @param index
         * @return
         */
        protected OValContext convertIndexContext(final OValContext context, final Object index) {

            if(context instanceof FieldContext) {
                return new IndexedFieldContext((FieldContext) context, index.toString());

            } else if(context instanceof MethodReturnValueContext) {
                return new IndexedMethodReturnValueContext((MethodReturnValueContext) context, index.toString());
            }

            return context;
        }

}
```

7.4.9.2.　「IndexedFieldContext.java」

- リスト、マップ、配列のインデックス、キー情報を保持する FieldContext です。
- FIeldContext は、フィールドにアノテーションを付与し、エラーが発生した場合のフィールド情報を保持するクラスです。

```
package sample.web.oval;
```

```
import java.lang.reflect.Field;

import net.sf.oval.context.FieldContext;


/**
 * インデックス情報を保持する FieldContext。
 * <p>リスト、マップ、配列の要素の場合、この情報を持つ。
 *
 */
public class IndexedFieldContext extends FieldContext {

	/** serialVersionUID */
	private static final long serialVersionUID = 1L;

	/** インデックス情報 */
	final protected String index;

	public IndexedFieldContext(final Field field, final String index) {
		super(field);
		this.index = index;
	}

	/**
	 * @param declaringClass
	 * @param fieldName
	 */
	public IndexedFieldContext(final Class< ? > declaringClass, final String fieldName, final String index)
	{
		super(declaringClass, fieldName);

		this.index = index;
	}

	public IndexedFieldContext(final FieldContext context, final String index) {
		super(context.getField());
		this.index = index;

	}

	public String getIndex() {
		return index;
	}

}
```

7.4.9.3.　「IndexedMethodReturnValueContext.java」

- リスト、マップ、配列のインデックス、キー情報を保持する MethodReturnValueContext です。
- MethodReturnValueContext は、アノテーション「@IsInvariant」を付与し、エラーが発生した場合のメソッドの戻り値の情報を保持するクラスです。

```
package sample.web.oval;

import java.lang.reflect.Method;

import net.sf.oval.context.MethodReturnValueContext;
```

```
/**
 * インデックス情報を保持する MethodReturnValueContext。
 * <p>リスト、マップ、配列の要素の場合、この情報を持つ。
 *
 */
public class IndexedMethodReturnValueContext extends MethodReturnValueContext {

	/** serialVersionUID */
	private static final long serialVersionUID = 1L;

	/** インデックス情報 */
	final protected String index;

	public IndexedMethodReturnValueContext(final Method method, final String index) {
		super(method);
		this.index = index;
	}

	public IndexedMethodReturnValueContext(final MethodReturnValueContext context, final String index) {
		super(context.getMethod());
		this.index = index;
	}

	public String getIndex() {
		return index;
	}

}
```

7.4.9.4.　「OValSpringValidator.java」

- MessageResolver などをインジェクションで設定できるようにします。

```
package sample.web.oval;

import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.List;

import net.sf.oval.ConstraintViolation;
import net.sf.oval.Validator;
import net.sf.oval.context.FieldContext;
import net.sf.oval.context.MethodReturnValueContext;
import net.sf.oval.context.OValContext;
import net.sf.oval.exception.ValidationFailedException;
import net.sf.oval.integration.spring.SpringValidator;
import net.sf.oval.internal.Log;
import net.sf.oval.internal.util.ReflectionUtils;

import org.springframework.context.MessageSource;
import org.springframework.util.Assert;
import org.springframework.validation.Errors;
import org.springframework.validation.FieldError;

/**
 * OVal 用の Spring Validator の拡張。
 * <p>
 * 変更点は次の通り。
```

```
 * <ul>
 *   <li>アノテーション「@AssertValid」を付与した場合、
 *   <li>アノテーション「@IsInvariant」を付与した場合、フィールドエラーとして処理されるよう変更する。
 *   <li>条件式において「SpEL」を使用可能にする。
 *
 */
public class OValSpringValidator extends SpringValidator {

    private static final Log LOG = Log.getLog(OValSpringValidator.class);

    /** バインドエラーがある場合、OVal のエラーメッセージを無視する */
    private boolean ignoreOnBindingErrors;

    /** フィールドエラーに対して複数のエラーが定義されている場合、1 番目のエラーのみ抽出する */
    private boolean ignoreOnMultipleFieldErrors;

    public OValSpringValidator() {
        super(new OValValidator());
    }

    public OValSpringValidator(final OValValidator validator) {
        super(validator);
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        // SpEL の登録
        getValidator().getExpressionLanguageRegistry()
            .registerExpressionLanguage("spel", new ExpressionLanguageSpelImpl());
    }

    /**
     * {@inheritDoc}
     */
    public void validate(final Object objectToValidate, final Errors errors) {
        try {
            doValidate(getValidator().validate(objectToValidate), errors);

        } catch(final ValidationFailedException ex) {
            LOG.error("Unexpected error during validation", ex);
            errors.reject(ex.getMessage());
        }
    }

    protected void doValidate(final List<ConstraintViolation> violations, final Errors errors) {

        for(ConstraintViolation violation : violations) {

            final OValContext ctx = violation.getContext();
            final String errorCode = violation.getErrorCode();
            final String errorMessage = violation.getMessage();
            final ConstraintViolation[] causeViolateions = violation.getCauses();

            if(ctx instanceof FieldContext) {

                String fieldName = ((FieldContext) ctx).getField().getName();
                if(ctx instanceof IndexedFieldContext) {
                    fieldName += String.format("[%s]", ((IndexedFieldContext) ctx).getIndex());
                }

                if(causeViolateions == null) {
```

SpEL を使用する場合。
実装は「7.4.15　条件付きチェックに「SpEL」を使用する」を参照。

リストなどのインデックス付きのエラーメッセージオブジェクトを作成します。。

バインドエラー時などの場合、OVal のエラーを無視します。

```
                        if(hasBindingErrors(errors, fieldName) || hasFieldErrors(errors, fieldName)) {
                            continue;
                        }
                        errors.rejectValue(fieldName, errorCode, errorMessage);

                    } else {
                        // ネストした bean の処理
                        errors.pushNestedPath(fieldName);
                        try {
                            doValidate(Arrays.asList(causeViolateions), errors);
                        } finally {
                            errors.popNestedPath();
                        }
                    }

                } else if(ctx instanceof MethodReturnValueContext) {

                    final Method method = ((MethodReturnValueContext) ctx).getMethod();

                    if(!ReflectionUtils.isGetter(method)) {
                        errors.reject(errorCode, errorMessage);

                    } else {
                        // getter メソッドの場合、フィールドとして処理する
                        String fieldName = ReflectionUtils.guessFieldName(method);
                        if(ctx instanceof IndexedMethodReturnValueContext) {
                            fieldName +=
String.format("[%s]", ((IndexedMethodReturnValueContext) ctx).getIndex());
                        }

                        if(causeViolateions == null) {
                            if(hasBindingErrors(errors, fieldName) || hasFieldErrors(errors, fieldName)) {
                                continue;
                            }
                            errors.rejectValue(fieldName, errorCode, errorMessage);

                        } else {
                            // ネストした bean の処理
                            errors.pushNestedPath(fieldName);
                            try {
                                doValidate(Arrays.asList(causeViolateions), errors);
                            } finally {
                                errors.popNestedPath();
                            }
                        }

                    }

                } else {
                    errors.reject(errorCode, errorMessage);
                }

            }

        }

        /**
         * バインドエラーを持つか判定する。
         * <p>設定値「ignoreOnBindingErrors」を考慮する。
         * @param errors
         * @param fieldName
```

ネストしている場合のエラーメッセージオブジェクトを作成します。

「@IsInvariant」を付与されてた Getter メソッドのエラーを、フィールドエラーとして処理する。

```
     * @return true: バインドエラーを持つ。
     */
    protected boolean hasBindingErrors(final Errors errors, final String fieldName) {

        if(!isIgnoreOnBindingErrors()) {
            return false;
        }

        if(!errors.hasFieldErrors(fieldName)) {
            return false;
        }

        for(FieldError fieldError : errors.getFieldErrors(fieldName)) {
            if(fieldError.isBindingFailure()) {
                return true;
            }
        }

        return false;

    }

    /**
     * フィールドエラーを持つか判定する。
     * <p>設定値「IgnoreFieldErrorWithMultiple」を考慮する。
     * @param errors
     * @param fieldName
     * @return
     */
    protected boolean hasFieldErrors(final Errors errors, final String fieldName) {

        if(!isIgnoreOnMultipleFieldErrors()) {
            return false;
        }

        return errors.hasFieldErrors(fieldName);
    }

    /**
     * Validator 用のメッセージを設定する。
     * <p>MessageResolver、ContextRenderer 用のメッセージを同時に設定する。
     * @param messageSource
     */
    public void setValidationMessageSource(final MessageSource messageSource) {
        Assert.notNull(messageSource);
        Validator.setMessageResolver(new SpringMessageResolver(messageSource));
        Validator.setContextRenderer(new SpringValidationContextRenderer(messageSource));

    }

    public boolean isIgnoreOnBindingErrors() {
        return ignoreOnBindingErrors;
    }

    public void setIgnoreOnBindingErrors(boolean ignoreOnBindingErrors) {
        this.ignoreOnBindingErrors = ignoreOnBindingErrors;
    }

    public boolean isIgnoreOnMultipleFieldErrors() {
        return ignoreOnMultipleFieldErrors;
    }
```

メッセージの解決に Spring の MessageSource を利用する。

```
    public void setIgnoreOnMultipleFieldErrors(boolean ignoreOnMultipleFieldErrors) {
        this.ignoreOnMultipleFieldErrors = ignoreOnMultipleFieldErrors;
    }
}
```

7.4.9.5.　「SpringMessageResolver.java」

```
package sample.web.oval;

import net.sf.oval.internal.Log;
import net.sf.oval.localization.message.MessageResolver;

import org.springframework.context.MessageSource;
import org.springframework.context.NoSuchMessageException;
import org.springframework.context.support.MessageSourceAccessor;


/**
 * Spring の{@link MessageSource}を OVal の{@link MessageResolver}として利用可能にするクラス。
 *
 */
public class SpringMessageResolver implements MessageResolver {

    private static final Log    LOG = Log.getLog(SpringMessageResolver.class);

    protected MessageSourceAccessor messageSourceAccessor;

    public SpringMessageResolver() {

    }

    public SpringMessageResolver(final MessageSource messageSource) {
        setMessageSource(messageSource);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public String getMessage(final String key) {
        try {
            return messageSourceAccessor.getMessage(key);
        } catch(NoSuchMessageException e) {
            LOG.debug("Key {1} not found.", key, e);
        }

        return null;
    }

    public void setMessageSource(final MessageSource messageSource) {
        this.messageSourceAccessor = new MessageSourceAccessor(messageSource);
    }

    protected MessageSourceAccessor getMessageSourceAccessor() {
        return messageSourceAccessor;
    }

}
```

ロケールなどを考慮し、MessageSource は、MessageSourceAccessor 経由で利用する。

メッセージコードの解決を行う、MessageResolver のメソッド。

7.4.9.6. 「SpringValidationContextRenderer.java」

- OVal の「ResourceBundleValidationContextRenderer」を Spring 用に記述しなおしたもの。
- エラーメッセージ上の置換文字{context}（フィールド名）を解決するクラスで、フィールドにアノテーションを付与した場合、次のキーの優先度に従いプロパティファイルからメッセージを取得する。
  - ①書式：「label.field.オブジェクト名.フィールド名」。例)「label.field.LoginCommand.name」
  - ②書式：「label.field.フィールド名」。例)「label.field.name」
  - ③書式：「label.オブジェクト名.フィールド名」「label.LoginCommand.name」
  - ④書式：「label.フィールド名」。例)「label.name」

```
package sample.web.oval;

import java.util.ArrayList;
import java.util.List;

import net.sf.oval.context.ClassContext;
import net.sf.oval.context.ConstructorParameterContext;
import net.sf.oval.context.FieldContext;
import net.sf.oval.context.MethodEntryContext;
import net.sf.oval.context.MethodExitContext;
import net.sf.oval.context.MethodParameterContext;
import net.sf.oval.context.MethodReturnValueContext;
import net.sf.oval.context.OValContext;
import net.sf.oval.internal.Log;
import net.sf.oval.localization.context.OValContextRenderer;

import org.springframework.context.MessageSource;
import org.springframework.context.NoSuchMessageException;
import org.springframework.context.support.MessageSourceAccessor;
import org.springframework.util.StringUtils;


/**
 * Spring の{@link MessageSource}を OVal の{@link OValContextRenderer}として利用可能にするクラス。
 * <p>{@link net.sf.oval.localization.context.ResourceBundleValidationContextRenderer}を参考に。
 *
 * <p>
 * コンテキストのキーの形式として、次の優先順位に一致したものを返す。
 * <pre>
 * label.class=My translated name of the class name
 * label.field.firstname=My translated name of the field "firstname"
 * label.field.lastname=My translated name of the field "lastname"
 * label.parameter.amount=My translated name of a constructor/method parameter "amount"
 * label.method.increase=My translated name of the method "increase"
 * </pre>
 *
 */
public class SpringValidationContextRenderer implements OValContextRenderer {

	private static final Log	LOG = Log.getLog(SpringValidationContextRenderer.class);

	public static final String CODE_SEPARATOR = ".";

	/** メッセージの接頭語 */
```

```
protected String prefix = "label";

protected MessageSourceAccessor messageSourceAccessor;

public SpringValidationContextRenderer() {

}

public SpringValidationContextRenderer(final MessageSource messageSource) {
    setMessageSource(messageSource);
}

/**
 * キーの候補を組み立てる。
 * @param key
 * @param baseName
 * @param name
 * @return
 */
protected String[] buildCode(final String key, final String baseName, final String name) {

    List<String> codeList = new ArrayList<String>();

    final String baseKey;
    if(getPrefix().isEmpty()) {
        baseKey = key;
    } else {
        baseKey = getPrefix() + CODE_SEPARATOR + key;
    }

    String[] splitBaseName = baseName.split("¥¥.");
    final String objName = splitBaseName[splitBaseName.length - 1];

    if(name == null) {
        codeList.add(baseKey + CODE_SEPARATOR + objName);
        codeList.add(baseKey);

    } else {
        codeList.add(baseKey + CODE_SEPARATOR + objName + CODE_SEPARATOR + name);
        codeList.add(baseKey + CODE_SEPARATOR + name);

        if(prefix.isEmpty()) {
            codeList.add(objName + CODE_SEPARATOR + name);
        } else {
            codeList.add(baseKey + CODE_SEPARATOR + objName + CODE_SEPARATOR + name);
            codeList.add(baseKey + CODE_SEPARATOR + name);
            codeList.add(getPrefix() + CODE_SEPARATOR + objName + CODE_SEPARATOR + name);
            codeList.add(getPrefix() + CODE_SEPARATOR + name);
        }

        codeList.add(name);

    }

    return StringUtils.toStringArray(codeList);
}

/**
 * {@inheritDoc}
 */
@Override
```

```
public String render(final OValContext ovalContext) {

    final String baseName;
    final String[] keys;
    if (ovalContext instanceof ClassContext)
    {
        final ClassContext ctx = (ClassContext) ovalContext;
        baseName = ctx.getClazz().getName();
        keys = buildCode("class", baseName, null);
    }
    else if (ovalContext instanceof FieldContext)
    {
        final FieldContext ctx = (FieldContext) ovalContext;
        baseName = ctx.getField().getDeclaringClass().getName();
        final String fieldName = ctx.getField().getName();
        keys = buildCode("field", baseName, fieldName);
    }
    else if (ovalContext instanceof ConstructorParameterContext)
    {
        final ConstructorParameterContext ctx = (ConstructorParameterContext) ovalContext;
        baseName = ctx.getConstructor().getDeclaringClass().getName();
        keys = buildCode("parameter", baseName, ctx.getParameterName());
    }
    else if (ovalContext instanceof MethodParameterContext)
    {
        final MethodParameterContext ctx = (MethodParameterContext) ovalContext;
        baseName = ctx.getMethod().getDeclaringClass().getName();
        keys = buildCode("parameter", baseName, ctx.getParameterName());
    }
    else if (ovalContext instanceof MethodEntryContext)
    {
        final MethodEntryContext ctx = (MethodEntryContext) ovalContext;
        baseName = ctx.getMethod().getDeclaringClass().getName();
        keys = buildCode("method", baseName, ctx.getMethod().getName());
    }
    else if (ovalContext instanceof MethodExitContext)
    {
        final MethodExitContext ctx = (MethodExitContext) ovalContext;
        baseName = ctx.getMethod().getDeclaringClass().getName();
        keys = buildCode("method", baseName, ctx.getMethod().getName());
    }
    else if (ovalContext instanceof MethodReturnValueContext)
    {
        final MethodReturnValueContext ctx = (MethodReturnValueContext) ovalContext;
        baseName = ctx.getMethod().getDeclaringClass().getName();
        keys = buildCode("method", baseName, ctx.getMethod().getName());
    }
    else
        return ovalContext.toString();

    // キー候補から取得する
    for(String key : keys) {
        try {
            return messageSourceAccessor.getMessage(key);

        } catch(NoSuchMessageException e) {
            LOG.debug("Key {1} not found.", key, e);
        }
    }

    return ovalContext.toString();
```

```
    }

    public void setMessageSource(final MessageSource messageSource) {
        this.messageSourceAccessor = new MessageSourceAccessor(messageSource);
    }

    protected MessageSourceAccessor getMessageSourceAccessor() {
        return messageSourceAccessor;
    }

    protected String getPrefix() {
        return prefix;
    }

    public void setPrefix(final String prefix) {
        this.prefix = (prefix != null ? prefix : "");
    }
}
```

7.4.9.7.　設定ファイルの記述

【servlet-context.xml】

- 既存の OVal 用の Validator のクラス名を、作成したクラス「OValSpringValidator」に変更します。
- メッセージカスタマイズ用に作成したクラス「SpringMessageResolver」、「SpringValidationContextRenderer」に Spring Bean の MessageSource を設定し、インジェクションします。

```
<beans>
    ・・・省略
    <!-- OVal 用の Validator -->
    <bean id="ovalValidator" class="sample.web.oval.OValSpringValidator">
        <property name="validator">
            <bean class="sample.web.oval.OValValidator">
                <constructor-arg>
                    <list>
                        <!-- OVal のアノテーションを使用する場合 -->
                        <bean class="net.sf.oval.configuration.annotation.AnnotationsConfigurer"/>
                    </list>
                </constructor-arg>
            </bean>
        </property>
        <property name="messageResolver">
            <bean class="sample.web.oval.SpringMessageResolver">
                <constructor-arg><ref bean="messageSource"/></constructor-arg>
            </bean>
        </property>
        <property name="contextRenderer">
            <bean class="sample.web.oval.SpringValidationContextRenderer">
                <constructor-arg><ref bean="messageSource"/></constructor-arg>
            </bean>
        </property>

        <property name="ignoreOnBindingErrors" value="true"/>
        <property name="ignoreOnMultipleFieldErrors" value="true"/>
    </bean>
    ・・・省略
</beans>
```

作成した、SpringValidator に変更する。

作成した、OVal 用 Validator に変更する。

メッセージ処理に MessageSource を渡します。

バインドエラー時などに OVal のエラーを無視するか設定します。

【ApplicationContext.xml】

```
<beans>
    ・・・省略
    <!-- 共通のメッセージファイル -->
    <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>classpath:message/message</value>
                <value>classpath:message/label</value>
                <value>classpath:app</value>
                <value>classpath:message/ValidationMessages</value>
                <value>classpath:message/OvalMessages</value>
            </list>
        </property>
    </bean>
    ・・・省略
</beans>
```

OVal 用のメッセージの読み込み設定を追加します。

【messages/OValMessages.properties】

```
## 共通で使用可能な変数
# {context} = フィールド名
# {invalidValue} = フィールドの値

## メッセージの定義
net.sf.oval.constraint.Assert.violated={context}は状態({expression})を満たしません。
net.sf.oval.constraint.AssertFalse.violated={context}が偽ではありません。
net.sf.oval.constraint.AssertNull.violated={context} must be null
net.sf.oval.constraint.AssertTrue.violated={context}が真ではありません。
net.sf.oval.constraint.AssertURL.violated={context} is not a valid URL
net.sf.oval.constraint.AssertValid.violated={context} is invalid
net.sf.oval.constraint.CheckWith.violated={context}が{simpleCheck}を満たしません。
net.sf.oval.constraint.DateRange.violated={context}が最小値({min})から最大値({max})の間にありません。
net.sf.oval.constraint.Email.violated={context} is not a valid e-mail address
net.sf.oval.constraint.EqualToField.violated={context}がフィールド({fieldName})と同一ではありません。
net.sf.oval.constraint.Future.violated={context}は未来日付である必要があります。
net.sf.oval.constraint.HasSubstring.violated={context}は部分文字列'{substring}'を含む必要があります。
net.sf.oval.constraint.InstanceOf.violated={context}は{types}のインスタンスでなければいけません。
net.sf.oval.constraint.InstanceOfAny.violated={context}は{types}のインスタンスでなければいけません。
net.sf.oval.constraint.Length.violated={context}は文字列長が最小値({min})から最大値({max})の間でなければいけません。
net.sf.oval.constraint.MatchPattern.violated={context}はパターン({pattern})にマッチしなければいけません。
net.sf.oval.constraint.Max.violated={context}は最大値({max})を超えています。
net.sf.oval.constraint.MaxLength.violated={context}は長さが最大値({max})を超えています。
net.sf.oval.constraint.MaxSize.violated={context}は{max}より多く要素を持つことが出来ません。
net.sf.oval.constraint.MemberOf.violated={context}はメンバー({members})の一部でなければいけません。
net.sf.oval.constraint.Min.violated={context}は最小値({min})を下回っています。
net.sf.oval.constraint.MinLength.violated={context}は長さが最小値({min})を下回っています。
net.sf.oval.constraint.MinSize.violated={context}は最小値({min})以上の要素を持たなければいけません。
net.sf.oval.constraint.NoSelfReference.violated={context}は自己参照しています。
net.sf.oval.constraint.NotBlank.violated={context}は空白のみです。
net.sf.oval.constraint.NotEmpty.violated={context}は空文字です。
net.sf.oval.constraint.NotEqual.violated={context}は文字列({testString})と同一ではいけません。
net.sf.oval.constraint.NotEqualToField.violated={context}はフィールド({fieldName})と同一ではいけません。
net.sf.oval.constraint.NotMatchPattern.violated={context} must not match the pattern {pattern}
```

```
net.sf.oval.constraint.NotMemberOf.violated={context}はメンバー({members})の一部ではいけません。
net.sf.oval.constraint.NotNegative.violated={context}は負の数ではいけません。
net.sf.oval.constraint.NotNull.violated={context}はヌル値ではいけません。
net.sf.oval.constraint.Past.violated={context}は過去日付ではいけません。
net.sf.oval.constraint.Range.violated={context}は最小値({min})から最大値({max})の間でなければいけません。
net.sf.oval.constraint.Size.violated={context}は{min}から{max}の間の要素数をもたなければいけません。
net.sf.oval.constraint.ValidateWithMethod.violated={context} はメソッド{methodName}(パラメータ:{parameterType})の呼び出しに失敗しました。

net.sf.oval.guard.Pre.violated={context}は事前状態({expression})を満たしません。
net.sf.oval.guard.Post.violated={context}は事後状態({expression})を満たしません。

net.sf.oval.exception.AccessingFieldValueFailedException.message=フィールド({fieldName})へのアクセスに失敗しました。
net.sf.oval.exception.ConstraintSetAlreadyDefinedException.message=既に別の制約(id：{constraintSetId})が設定されています。
net.sf.oval.exception.ExpressionLanguageNotAvailableException.message=EL 言語（{languageId}）は利用できません。
net.sf.oval.exception.UndefinedConstraintSetException.message=制約(id：{constraintSetId})は設定されていません。
net.sf.oval.exception.InvokingMethodFailedException.message=メソッド(メソッド名：{methodName})の呼び出しに失敗しました。

net.sf.oval.context.ConstructorParameterContext.parameter=パラメータ
net.sf.oval.context.MethodParameterContext.parameter=パラメータ

# 項目名（コンテキスト）
label.field.name=名前
label.age=年齢
```

7.4.10. 条件付きチェック

OVal では、チェック条件を Groovy、JavaScript など様々なスクリプト言語で記述することができます。

- 各種言語を使用するには、対応したライブラリが必要になります（表 7.20）。
  また、どの言語を使用するかは、表 7.20 の項目「キー」を指定します。
- チェック条件を記述する方式として、2 つあります。
  - チェック内容を完全に独自に指定できるアノテーションを使用する。
    書式「@Assert(expr="条件式", lang="言語キー")」
    詳細は、「7.4.10.1　アノテーション「@Assert」」を参照してください。
  - チェック処理実行の条件をアノテーション共通の引数「when」で指定する。
    書式「@アノテーション(value="1.0", when="言語キー:条件式")」
    一部、使用できないアノテーションがあります。
    詳細は、「7.4.10.2　引数「when」」を参照してください。
- プロジェクトで使用する際には、言語を 1 つに絞込み使用することをお勧めします。
  使用しやすい、「Goovy」「OGNL」「SpEL」がお勧めです。

表 7.20　引数「lang」で指定可能な言語の種類

No.	言語キー	言語	必要なライブラリ(pom 形式の情報)
1	bsh、beanshell	BeanShell	GroupId=org.beanshell, ArtifactId=bsh, Version=2.0b4
2	groovy	Groovy	GroupId=org.codehaus.groovy, ArtifactId=groovy-all, Version=1.8.2
3	jexl	JEXL	GroupId=org.apache.commons , ArtifactId=commons-jexl, Version=2.0.1
4	js、javascript	JavaScript	GroupId=org.mozilla , ArtifactId=rhino, Version=1.7R3
5	mvel	MVEL	GroupId=org.mvel , ArtifactId=mvel2, Version=2.0.19
6	ognl	OGNL	GroupId=ognl , ArtifactId=ognl, Version=3.0.2
7	ruby、jruby	Ruby	GroupId=org.jruby , ArtifactId=jruby, Version=1.6.4
8	spel	SpEL	「7.4.15　条件付きチェックに「SpEL」を使用する」を参照。

表 7.21　引数「when」「expr」で使用可能な変数

No.	変数	説明
1	_value	チェック対象のプロパティ(フィールド)自身。
2	_this	チェック対象のプロパティが定義されているオブジェクト(Command)。 例）「_this.name」で、他のプロパティを参照することができます。

7.4.10.1. アノテーション「@Assert」

検証内容をスクリプト言語で記述します。

- 書式「@Assert(expr="条件式", lang="言語キー")」
- 正常値の場合、true を返す必要があります。
- エラーメッセージはたいていが固有のため、引数「message」でメッセージキーを指定します。

【Groovy の場合】

- Groovy の演算子、リテラルは、基本的に Java と同じです
- 任意のクラスの static メソッドを呼ぶ場合は、通常の Java のように呼びます。
  呼び出す際には、クラスは「クラスパス.メソッド(引数)」と絶対パスで指定します。

```
// 演算子を使用した条件の記述
@Assert(expr="_value != null && _value.length() <= 5", lang="groovy", message="error.name")

// static メソッドを利用した条件の記述
@Assert(expr="_value != null && sample.utils.ValiateUtils.isNamePattern(_value)",
 lang="groovy", message="error.name")
```

【OGNL の場合】

- OGNL の演算子、リテラルは、基本的に Java と同じです。メソッドも呼び出せます。
  言語の仕様については、下記のサイトを参考にしてください
  - 「http://commons.apache.org/ognl/language-guide.html」
  - 「http://s2container.seasar.org/2.4/ja/ognl.html」

```
// 演算子を使用した条件の記述
@Assert(expr="_value != null && _value.length() <= 5", lang="ognl ", message="error.name")

// static メソッドを利用した条件の記述
@Assert(expr="_value != null && @sample.utils.ValiateUtils@isNamePattern(_value)",
 lang="gnl ", message="error.name")
```

【SpEL の場合】

- SpEL では、変数には接頭語として「#」を付けます。
  したがって、「_this⇒#_this」「_value⇒#_value」として参照します。
- 比較演算子は Java と同じです。論演算子は「and」「or」「!」と Java とは異なります。
  詳細は、「6.5　Spring Expression Language(SpEL)」を参照してください。
- static メソッド呼ぶ場合は、「T(クラスパス).メソッド(引数)」とします。
  “T”はクラス“タイプ”の T です。

```
// 演算子を使用した条件の記述
@Assert(expr="#_value != null and #_value.length() <= 5", lang="spel", message="error.name")

// static メソッド利用した条件の記述
@Assert(expr="#_value != and and T(sample.utils.ValiateUtils).isNamePattern(#_value)",
 lang="spel", message="error.name")
```

7.4.10.2. 引数「when」

アノテーションの検証内容の実行条件をスクリプト言語で指定します。

- 書式「@アノテーション(value="1.0", when="言語キー:条件式")」
- 戻り値として「boolean」を返す式を記述する必要があります。
- 戻り値が true の場合、各アノテーションの検証処理が実行されます。

【Groovy の場合】

式の記述方法は、@Assert の場合と同様です。

```
// 演算子を使用した条件の記述
@Range(min=1, max=200, when="groovy:_this.name != null && _this.name.length() > 0")

// static メソッドを利用した条件の記述
@Range(min=1, max=200, when="groovy:org.apache.commons.lang.StringUtils.isNotEmpty(_this.name)")
```

【OGNL の場合】

式の記述方法は、@Assert の場合と同様です。

```
// 演算子を使用した条件の記述
@Range(min=1, max=200, when="ognl:_this.name != null && _this.name.length() > 0")

// static メソッド利用した条件の記述
@Range(min=1, max=200, when="ognl:@org.apache.commons.lang.StringUtils@isNotEmpty(_this.name)")
```

【SpEL の場合】

式の記述方法は、@Assert の場合と同様です。

```
// 演算子を使用した条件の記述
@Range(min=1, max=200, when="spel:#_this.name != null and #_this.name.length() > 0")

// static メソッド利用した条件の記述
@Range(min=1, max=200, when="spel:T(org.apache.commons.lang.StringUtils).isNotEmpty(#_this.name)")
```

7.4.11. ネストした Comamnd の検証を行う「@AssertValid」

- JavaBean のような、階層を持つフィールドを検証したい場合、アノテーション「@AssertValid」を使用します。
- 「@AssertValid」を付与したフィールドのクラスにも、入力値検証のアノテーションを付与する必要があります。
- OVal に付属の Spring Validator の場合、エラーメッセージのキーが正しくネストした形式にになりません。
  - 対策として改良した Validator を使用します。詳細は、「7.4.9　SpringValidator を拡張する」を参照してください。

【Command の作成】

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import net.sf.oval.constraint.AssertValid;
import net.sf.oval.constraint.NotEmpty;
import net.sf.oval.constraint.NotNull;

import org.apache.commons.collections.FactoryUtils;
import org.apache.commons.collections.ListUtils;
import org.apache.commons.lang.builder.ToStringBuilder;

public class OvalSample5Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    @NotEmpty
    private String name;

    @NotNull
    @AssertValid
    private MemberCardBean memberCard;

    @NotNull
    @AssertValid
    private List<BookBean> books;

    @SuppressWarnings("unchecked")
    public OvalSample5Command() {

        memberCard = new MemberCardBean();

        books = ListUtils.lazyList(
                new ArrayList<String>(),
                FactoryUtils.instantiateFactory(BookBean.class));
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
```

プロパティが Bean の場合、@AssertValid を付与します。
Null の場合、例外が発生するので、@NotNull を付与しておきます。

プロパティがリスト形式の場合、@AssertValid を付与します。
マップの場合も同様に付与します。

```
    }

    // getter, setter は省略
}
```

【ネストしている JavaBean の定義】

- 通常の Command のように、検証用のアノテーションを付与します。

```
public class MemberCardBean implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 880011173788283374 1L;

    @Length(max=10)
    protected String code;

    @DateRange(format="yyyy-MM-dd", min="2010-01-01 00:00:00.000", max="today")
    protected Date entryDate;

    public MemberCardBean() {

    }

    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }

    // getter, setter は省略
}
```

7.4.12. Getter メソッドにアノテーションを付与する「@IsInvariant」

アノテーションを Command のプロパティに付与でいない場合の方法を説明します。方法は 2 つあります。

- 1 つ目は、アノテーションを使用しないで外部の XML ファイルによる設定です。
  詳細は、「7.4.6　XML による設定を使用する」を参照してください。
- 2 つ目は、プロパティの Getter メソッドにアノテーションを付与する方法です。
  方法は、通常のアノテーションに加え、Getter メソッドに対して「**@IsInvariant**」を付与します。
  他のアノテーションとは異なり、パッケージ「net.sf.oval.configuration.annotation」に格納されています。

【Command の作成】

- OVal に付属の Spring Validator の場合、エラーオブジェクトはフィールドエラーではなくグローバルエラーになるため注意が必要です。
  - 対策として、改良した Validator を使用します。詳細は、「7.4.9　SpringValidator を拡張する」を参照してください。

```
package sample.web.oval.controller;

import java.io.Serializable;

import net.sf.oval.configuration.annotation.IsInvariant;
```

```
import net.sf.oval.constraint.Length;
import net.sf.oval.constraint.Range;

import org.apache.commons.lang.builder.ToStringBuilder;

public class OvalSample4Command implements Serializable {

	/** serialVersionUID */
	private static final long serialVersionUID = 1L;

	private String name;

	private Integer age;

	public OvalSample4Command() {
	}

	@Override
	public String toString() {
		return ToStringBuilder.reflectionToString(this);
	}

	@IsInvariant
	@Length(max=5)
	public String getName() {
		return name;
	}

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

	@IsInvariant
	@Range(min=1, max=200, when="groovy:_this.name != null && _this.name.length() > 0")
	public Integer getAge() {
		return age;
	}

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

}
```

通常のアノテーションに追加し、「@IsInvariant」を付与します。

7.4.13. 独自の検証用メソッドを呼ぶ

別メソッドに定義した検証ロジックを呼ぶ方法として、2 種類あります。

7.4.13.1. @CheckWith

- インタフェース「CheckWith.SimpleCheck」を実装し、チェックロジック用メソッド「#isSatisfied(…)」を定義します。このメソッドは、正常値の場合 true を返す必要があります。
- アノテーションの引数「value」にて、CheckWith.SimpleCheck を実装したクラスを指定します。「@ValidateWithMethod」と機能は同じですが、「@CheckWith」はクラスを作成するので Command クラスとは別クラスのファイルにて作成することもでき、ステップ数の多い検証を行うのに向いています。
- 引数「ignoreIfNull=false」とすることで、チェック対象のフィールドが null の場合も処理が実行されます。
- 独自ロジックのため、アノテーションの引数「message」にて、メッセージキーを指定することをお勧めします。

【Command の作成】

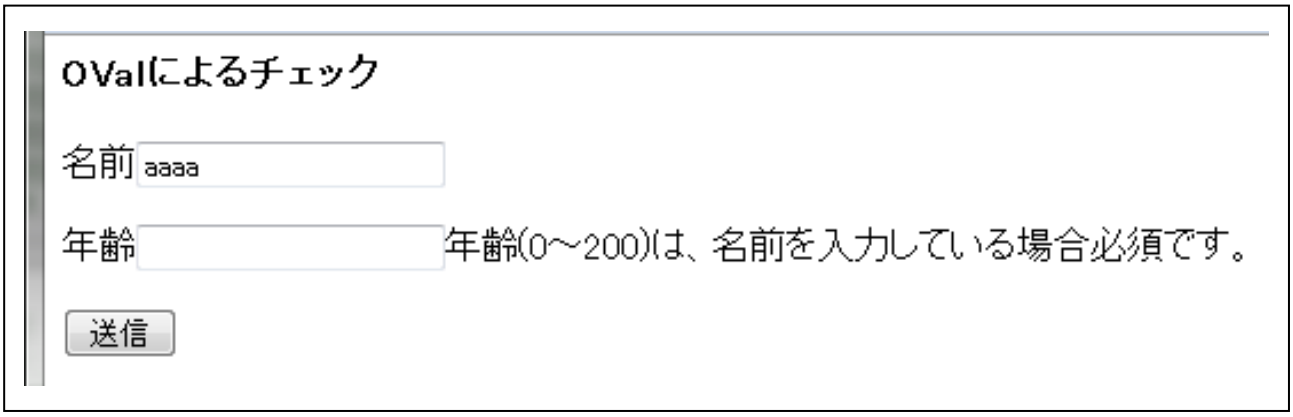
```
import java.io.Serializable;

import net.sf.oval.constraint.CheckWith;
import net.sf.oval.constraint.CheckWithCheck;
import net.sf.oval.constraint.Length;

import org.apache.commons.lang.StringUtils;
import org.apache.commons.lang.builder.ToStringBuilder;
import org.apache.commons.lang.math.IntRange;

/**
 * OVal によるチェック
 *
 */
public class OvalSample2Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    @Length(max=5)
    private String name;

    @CheckWith(value=AgeRequiredCheck.class, ignoreIfNull=false, message="error.age.required.violated")
    private Integer age;

    public OvalSample2Command() {

    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
```

@CheckWith にて、複雑なチェックを実装します。
ロジックの実装クラスを指定します。

```
    /**
     * age の必須チェック。
     */
    private static class AgeRequiredCheck implements CheckWithCheck.SimpleCheck {

        /** serialVersionUID */
        private static final long serialVersionUID = 1L;

        /** age が取りうる範囲 */
        private IntRange ageRange = new IntRange(0, 200);

        @Override
        public boolean isSatisfied(final Object validatedObject, final Object value) {

            // チェック対象のフィールド
            final Integer targetValue = (Integer) value;

            // Command 内の他のフィールド
            final String name = ((OvalSample2Command) validatedObject).name;

            // name に値が入っているとき、age は必須。
            if(StringUtils.isNotEmpty(name) && targetValue == null) {
                return false;

            // name と age の両方が空の場合
            } else if(StringUtils.isEmpty(name) && targetValue == null) {
                return true;
            }

            // age が範囲以内に無い場合
            if(ageRange.containsInteger(targetValue)) {
                return false;
            }

            return false;
        }
    }
    // getter、setter は省略
}
```

検証ロジックの実装を行います。
戻り値が true の場合、正常値と判定されます。

【ブラウザの表示】



7.4.13.2. CheckWithCheck.SimpleCheck の実装クラスで Spring Bean をインジェクションする

チェックロジックを独自実装した場合クラスの中で、Spring Bean を利用する方法を説明します。

【servlet-context.xml】

- 「SpringInjector」を Spring Bean として登録します。

```
<beans>
    ・・・省略
    <!-- OVal 用の CheckWithChewk.SimpleCheck の中で Spring Bean を使用する場合-->
    <bean class="net.sf.oval.integration.spring.SpringInjector" />

    <!-- OVal 用の Validator -->
    <bean id="ovalValidator2" class="net.sf.oval.integration.spring.SpringValidator">
        <property name="validator">
            <bean class="net.sf.oval.Validator">
                <constructor-arg>
                    <list>
                        <!-- OVal のアノテーションを使用する場合  -->
                        <bean class="net.sf.oval.configuration.annotation.AnnotationsConfigurer"/>
                    </list>
                </constructor-arg>
            </bean>
        </property>
    </bean>
    ・・・省略
</beans>
```

【AnnotationsConfigurer の設定】

- AnnotationsConfigure のインスタンスを取得し、「BeanInjectingCheckInitializationListener」を追加します。
- 「7.4.9　SpringValidator を拡張する」のように、独自の SpringValidator の拡張をしている場合は、メソッド「#afterPropertiesSet()」で設定を行います。または、
  または、「7.4.8.1　システム共通の DataBinder で指定する」のように、GlobalBindingInitializer で設定する方法もあります。

```
Validator validator = /* net.sf.oval.Validator の取得　*/;
for(Configurer configure : validator.getConfigurers()) {
    if(configure instanceof AnnotationsConfigurer) {
        AnnotationsConfigurer annotationConfigure = (AnnotationsConfigurer) configure;

annotationConfigure.addCheckInitializationListener(BeanInjectingCheckInitializationListener.INSTANCE);

    }
}
```

※独自に拡張した SpringValidator 内で設定する場合

```
public class OValSpringValidator extends SpringValidator {
    ・・・省略
    @Override
    public void afterPropertiesSet() throws Exception {
```

```
        Validator validator = getValidator();
        for(Configurer configure : getValidator().getConfigurers()) {
            if(configure instanceof AnnotationsConfigurer) {
                AnnotationsConfigurer annotationConfigure = (AnnotationsConfigurer) configure;

annotationConfigure.addCheckInitializationListener(BeanInjectingCheckInitializationListener.INSTANCE);

            }
        }
    }
    ・・・・省略
}
```

【CheckWithCheck.SimpleCheck の実装クラス】

- アノテーション「@Autowired」にてインジェクションします。（クラス型が一致する場合インジェクションします）
  - アノテーション「@Resource」は使用できないので注意してください。
    これは、登録した「SpringInjector」の実装中で、「@Autowired」でインジェクション処理を行う実装の「AutowiredAnnotationBeanPostProcessor」が使用されているためです。
    「@Resource」を使用したい場合は、「SpringInjector」を拡張し、
    「CommonAnnotationBeanPostProcessor」を使用するよう修正してください。
  - 名前によりインジェクションする Spring Bean を絞り込みたい場合、「@Qualifier」を使用します。

```
public class OvalSample6Command implements Serializable {

    ・・・省略

    @CheckWith(value=AgeRequiredCheck.class, ignoreIfNull=false, message="error.age.required.violated")
    private Integer age;

    public OvalSample6Command() {
    }

    /**
     * age の必須チェック。
     */
    private static class AgeRequiredCheck implements CheckWithCheck.SimpleCheck {

        @Autowired
        @Qualifier("appConfiguration")
        private Configuration configuration;

        @Override
        public boolean isSatisfied(final Object validatedObject, final Object value) {

            int userNameMaxLength = configuration.getInt("userNameMaxLength");

            //TODO: ロジックの実装

            return false;
        }
    }
    ・・・省略
}
```

Spring Bean を@Autowired でインジェクションします。

7.4.13.3. @ValidateWithMethod

- 引数「methodName」にてメソッド名を指定します。
  指定するメソッドの書式は、「boolean XXXX(フィールドのクラスタイプ)」で、引数は必ず１つです。
  「@CheckWith」と同じ機能ですが、Command 内のメソッドとして定義することで、他のフィールドへのアクセスが容易でき、それほど複雑でない検証ロジックに向いています。
- 引数「parameterType」にて、メソッドの引数を指定します。フィールドのタイプと合わせます。
- 引数「ignoreIfNull=false」とすることで、チェック対象のフィールドが null の場合も処理が実行されます。
- 独自ロジックのため、アノテーションの引数「message」にて、メッセージキーを指定することをお勧めします。

【Command の作成】

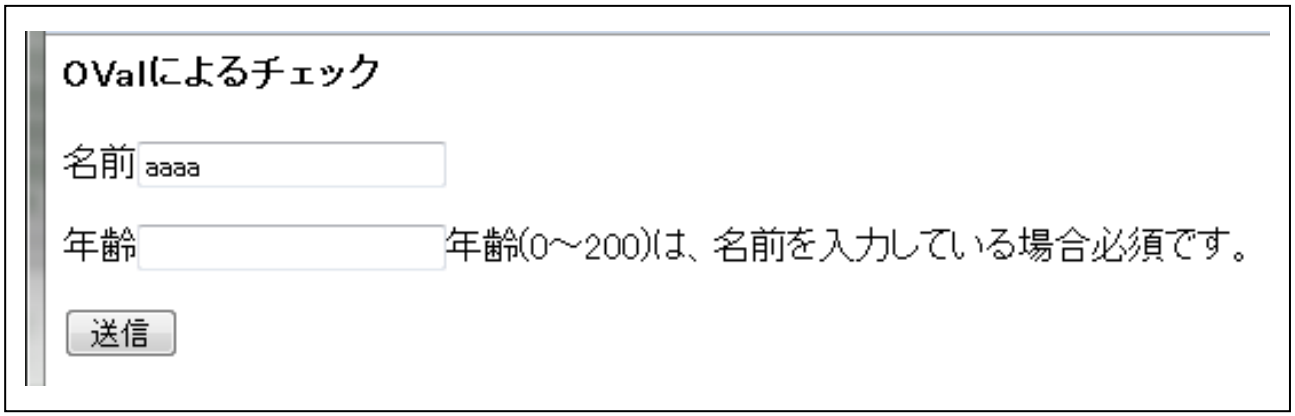
```
import java.io.Serializable;

import net.sf.oval.constraint.Length;
import net.sf.oval.constraint.ValidateWithMethod;

import org.apache.commons.lang.StringUtils;
import org.apache.commons.lang.builder.ToStringBuilder;
import org.apache.commons.lang.math.IntRange;

/**
 * OVal によるチェック
 *
 */
public class OvalSample3Command implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    @Length(max=5)
    private String name;

    @ValidateWithMethod(methodName="isValidAge", parameterType=Integer.class,
 ignoreIfNull=false, message="error.age.required.violated")
    private Integer age;

    public OvalSample3Command() {

    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }

    /**
     * age の必須チェック。
     */
    private boolean isValidAge(final Integer targetValue) {
```

@ValidateWithMethod にて、複雑なチェックを実装します。

検証ロジックの実装を行います。
戻り値が true の場合、正常値と判定されます。

```
        /** age が取りうる範囲 */
        final IntRange ageRange = new IntRange(0, 200);

        // name に値が入っているとき、age は必須。
        if(StringUtils.isNotEmpty(name) && targetValue == null) {
            return false;

        // name と age の両方が空の場合
        } else if(StringUtils.isEmpty(name) && targetValue == null) {
            return true;
        }

        // age が範囲以内に無い場合
        if(ageRange.containsInteger(targetValue)) {
            return false;
        }

        return false;

    }
    // getter, setter は省略
}
```

【ブラウザの表示】



7.4.14. OVal の独自アノテーションを実装する

独自のアノテーションを作成する方法を説明します。作成に必要なものを「表 7.22」に示します。基本k的に、「7.3.7　Bean Validation のアノテーションを独自実装する」の Bean Validattion の時と同様の種類のものを作成します。

表 7.22　独自の OVal 用のアノテーションを作成するたに必要なもの

No.	項目	内容
1	アノテーションの定義クラス	作成する検証用アノテーションの Java の定義ファイル。
2	検証ロジックの実装クラス	アノテーションが付加された項目の値を検証する Check クラス。
3	エラーメッセージファイル	アノテーションに対するエラーメッセージ。

【作成するアノテーションの仕様】

- 整数に対する範囲チェックを行うアノテーション「@DecimalRange」を作成する。
  - 既存の「@Range」は、上限値、下限値のクラス型が小数「double」であるため、エラーメッセージに小数で表示されてしまいます。
    そのため、プロパティのクラス型が整数「int」の場合も、エラーメッセージでは小数で表示され、入力型とエラーメッセージの型が不一致となっていしまいます。
- 固有の引数、下限値「min」、上限値「max」を持つ。
- 共通の引数「appliesTo」「errorCode」「message」「profiles」「severity」「when」が設定可能とします。

【アノテーションの定義：DecimalRange.java】

- メタアノテーション「@Constraint」にて、チェックロジックの実装クラスを指定します。
- 共通の引数として、「message」は必須です。他は、定義しなくてもかまいません。
- 使用する際は、特にどこかに登録する必要などはありません。

```
package sample.web.oval.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import net.sf.oval.ConstraintTarget;
import net.sf.oval.configuration.annotation.Constraint;
import net.sf.oval.configuration.annotation.Constraints;

/**
 * 整数に対して、範囲内に収まっているかをチェックする。
*
 */
@Documented
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Constraint(checkWith = DecimalRangeCheck.class)
public @interface DecimalRange {

    /** 固有の引数：下限値 */
    long min() default Long.MIN_VALUE;

    /** 固有の引数：上限値 */
    long max() default Long.MAX_VALUE;

    /** 共通の引数：検証対象の部分を明示する */
    ConstraintTarget[] appliesTo() default ConstraintTarget.VALUES;

    /** 共通の引数：エラーコード */
    String errorCode() default "sample.web.oval.annotation.DecimalRange";

    /** 共通の引数：エラーメッセージ */
    String message() default "sample.web.oval.annotation.DecimalRange.violated";

    /** 共通の引数 */
    String[] profiles() default {};

    /** 共通の引数 */
    int severity() default 0;

    /** 共通の引数 */
    String target() default "";

    /** 共通の引数：条件式 */
    String when() default "";

    @Documented
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
    @Constraints
    public @interface List {
        DecimalRange[] value();
        String when() default "";
    }

}
```

チェックロジックの実装クラスを設定します。

固有の引数の定義。

共通のメッセージとして、「message」は必須です。

【検証用ロジックの実装クラス：DecimalRangeCheck】

- 抽象クラス「net.sf.oval.configuration.annotation .AbstractAnnotationCheck」を継承します。
  Generics には、対応するアノテーション、今回は「DecimalRange」を設定します。
- クラス名の統一として、「アノテーション名 + Check」とします。
- メソッド「#isSatisfied(...)」を実装します。
  正常値の場合、true を返すようにします。
- 共通の引数「when」などは、アノテーションに定義しておくだけで特に処理を実装する必要はありません。継承元のクラス「AbstractAnnotationCheck」にて処理されます。

```
package sample.web.oval.annotation;

import static net.sf.oval.Validator.getCollectionFactory;

import java.util.Map;

import net.sf.oval.ConstraintTarget;
import net.sf.oval.Validator;
import net.sf.oval.configuration.annotation.AbstractAnnotationCheck;
import net.sf.oval.context.OValContext;

public class DecimalRangeCheck extends AbstractAnnotationCheck<DecimalRange> {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private long min = Long.MIN_VALUE;
    private long max = Long.MAX_VALUE;

    /**
     * {@inheritDoc}
     */
    @Override
    public void configure(final DecimalRange constraintAnnotation) {
        super.configure(constraintAnnotation);
        setMax(constraintAnnotation.max());
        setMin(constraintAnnotation.min());
    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected Map<String, String> createMessageVariables() {
        final Map<String, String> messageVariables = getCollectionFactory().createMap(2);
        messageVariables.put("max", Long.toString(max));
        messageVariables.put("min", Long.toString(min));
        return messageVariables;
    }

    /**
     * {@inheritDoc}
     */
    @Override
```

アノテーションに固有の引数を持つ場合、その値を保持する際に定義します。

エラーメッセージ内で使用可能な置換文字を定義します。通常は固有の引数を設定します。{context}、{invalidValue}は定義する必要はありません。

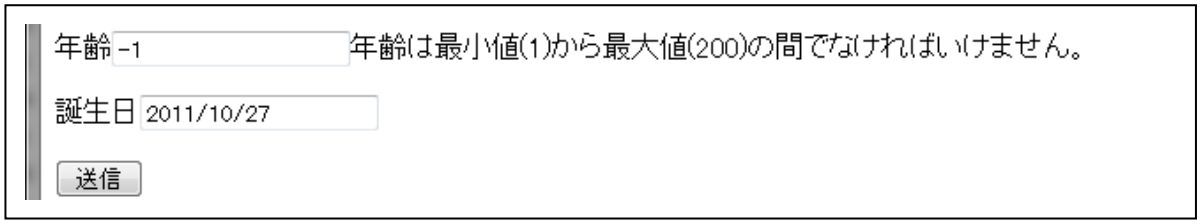
```
    protected ConstraintTarget[] getAppliesToDefault() {
        return new ConstraintTarget[] {ConstraintTarget.VALUES};
    }

    /**
     * {@inheritDoc}
     */
    public boolean isSatisfied(final Object validatedObject, final Object valueToValidate,
            final OValContext context, final Validator validator) {
        if(valueToValidate == null)
            return true;

        if(valueToValidate instanceof Number) {
            final long longValue = ((Number) valueToValidate).longValue();
            return longValue >= min && longValue <= max;
        }

        final String stringValue = valueToValidate.toString();
        try {
            final double longValue = Long.parseLong(stringValue);
            return longValue >= min && longValue <= max;
        } catch (final NumberFormatException e) {
            return false;
        }
    }

    public double getMin() {
        return min;
    }

    public void setMin(final long min) {
        this.min = min;
        requireMessageVariablesRecreation();
    }

    public double getMax() {
        return max;
    }

    public void setMax(final long max) {
        this.max = max;
        requireMessageVariablesRecreation();
    }
}
```

入力値の検証ロジックです。
True を返す場合、正常値と判断されます。

【エラーメッセージ：message/OValMessage.properties】

```
## 独自のアノテーションのエラーメッセージ
sample.web.oval.annotation.DecimalRange.violated={context}は最小値({min})から最大値({max})の間でなければいけません。
```

【ブラウザの表示】



7.4.15. 条件付きチェックに「SpEL」を使用する

「@Assert」や引数「when」などでスクリプト言語による条件式の記述において、Spring Expression Language（SpEL）を使用する方法を説明します。SpEL を利用する利点は次の通りです。

- 「SpEL」は OGNL と文法が似ており、かつ Spring 用のスクリプト言語であるため、OVal を Spring 内で使用するには非常に親和性が高い。
- 既に SpEL を使っている場合は、他の言語を覚えなおす必要がない。

【SpEL を評価するクラス（ExpressionLanguageSpelImpl.java）】

- OVal で、新たにスクリプト言語を使用するには、「net.sf.oval.expression.ExpressionLanguage」を実装する必要があります。
- SpEL の処理方法は、OGNL の実装「ExpressionLanguageOGNLImpl」とほぼ同じです。

```
package sample.web.oval;

import java.util.Map;
import java.util.Map.Entry;

import net.sf.oval.exception.ExpressionEvaluationException;
import net.sf.oval.expression.ExpressionLanguage;
import net.sf.oval.expression.ExpressionLanguageOGNLImpl;
import net.sf.oval.internal.Log;
import net.sf.oval.internal.util.ObjectCache;

import org.springframework.expression.EvaluationContext;
import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionException;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;


/**
 * OVal 用の SpEL を使用するための実装。
 * <p>参考「{@link ExpressionLanguageOGNLImpl}」
 *
 */
public class ExpressionLanguageSpelImpl implements ExpressionLanguage {

	private static final Log LOG = Log.getLog(ExpressionLanguageSpelImpl.class);

	private final ObjectCache<String, Object> expressionCache = new ObjectCache<String, Object>();

	/**
	 * {@inheritDoc}
	 */
	@Override
	public Object evaluate(final String expression, final Map<String, ?> values)
			throws ExpressionEvaluationException {

		try {
			// 変数などのコンテキストの設定
```

```
            final EvaluationContext ctx = new StandardEvaluationContext();
            for (final Entry<String, ? > entry : values.entrySet()) {
                ctx.setVariable(entry.getKey(), entry.getValue());
            }

            LOG.debug("Evaluating SpEL expression: {1}", expression);

            // 式をコンパイルし、キャッシュする
            Expression expr = (Expression) expressionCache.get(expression);
            if (expr == null) {
                ExpressionParser parser = new SpelExpressionParser();
                expr = parser.parseExpression(expression);
                expressionCache.put(expression, expr);
            }

            return expr.getValue(ctx);

        } catch (final ExpressionException ex){
            throw new ExpressionEvaluationException("Evaluating script with SpEL failed.", ex);
        }
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public boolean evaluateAsBoolean(final String expression, final Map<String, ?> values)
            throws ExpressionEvaluationException {

        final Object result = evaluate(expression, values);

        if(!(result instanceof Boolean)) {
            throw new ExpressionEvaluationException("The script must return a boolean value.");
        }

        return (Boolean) result;
    }
}
```

【SpEL を登録する】

- OVal の Validator から「ExpressionLanguageRegister」を取得し、言語キー「spel」と作成した SpEL 式を処理するクラスを登録します。
- 独自に拡張した Spring 用の Validator で登録する場合は、「7.4.9　SpringValidator を拡張する」を参照してください。

```
Validator validator = /* net.sf.oval.Validator の取得 */;
validaotr.getExpressionLanguageRegistry().registerExpressionLanguage(
                "spel", new ExpressionLanguageSpelImpl());
```

8. セッション管理

8.1. セッションスコープ

セッションの種類には複数あり、それぞれスコープごとに有効範囲があります。また、SrpingMVC には基本の Servlet API を使用したものの他に、SpringBean のインスタンスのスコープをセッションにすることなどできます。

表 8.1　セッションスコープの種類

No.	セッションスコープ	説明
①	ページ(page)	・ページ内でのみ有効なスコープ。 ・JSP 内で保存した値がこのスコープとなる。
②	リクエスト(request)	・リクエスト発生から終了まで有効。 ・次のページまで有効。ただし、リダイレクト先は無効となる。 ・Controller の中で Model に設定したオブジェクトは、この値となる。
③	フラッシュ(flash)	・リダイレクト先のページまで有効。 ・Spring3.0 で無い機能であるので、独自実装となる(3.4.3 節、8.5 節)。 ・Spring3.1 から標準実装されている(3.4.4 節)。
④	セッション(session)	・ページ間で有効。例えば、ログインユーザ情報など保存する。
⑤	アプリケーション (application)	・アプリケーション全体で有効。セッションが切れても有効。 ・共通変数などを保存する。

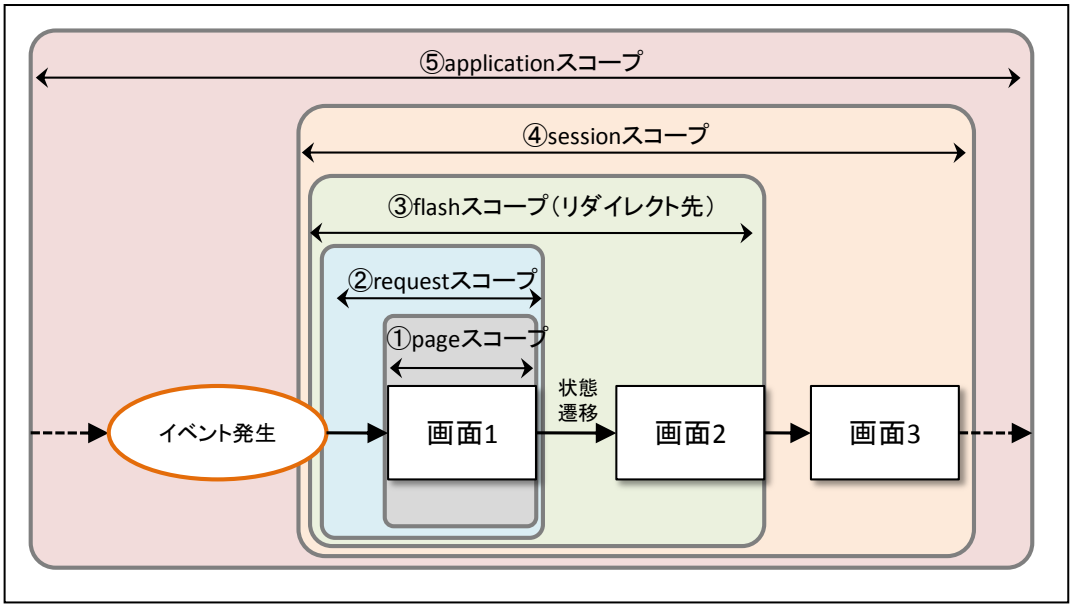


図 8.1　各スコープのライフサイクルイメージ

8.2. Servlet API を使用したセッション管理

セッションスコープごとに、「XXX#setAttribute(“名称”, データ)」、「XXX#getAttribute(“名称”)」というメソッドがあり、そのインスタンスを取得して作成します。Servlet API の場合、スコープごとに異なるクラスを使用するため不便です。Spring MVC の場合、Servlet API も使用可能ですが、1 つのクラス「WebRequest」で取得・設定可能になります(8.3 節参照)。

表 8.2　各セッションスコープに対応する ServletAPI のクラス

No.	スコープ	使用するクラス	使用するクラスの取得元
1	page	javax.servlet.jsp.PageContext	JSP(8.6 節参照) TagSupport(カスタムタグクラス)
2	request	javax.servlet.ServletRequest / javax.servlet.http.HttpServletRequest	Servlet クラスの引数
3	session	javax.servlet.http.HttpSession	HttpServletRequest#getSession()
4	flash	— (SpringMVC でのみ使用可能)	— (SpringMVC でのみ使用可能)
5	application	javax.servlet.ServletContext	HttpSession#getServletContext() ServletContextEvent

8.2.1. JSP を使用したセッション管理（:TODO）

8.3. Spring MVC のセッション管理

8.3.1. セッション上のデータ呼び出し

JSP データを呼び出す際には、EL 式の形式で呼び出すことができます。標準ではセッションスコープの種類に区別なく呼び出すことがでます。詳細は、「6.1　EL 式の基本」を参照してください。

```
<%-- 単純なセッションデータの呼び出し --%>
${data}

<%-- Bean のプロパティを指定した呼び出し
      メソッド「getName()」を定義しておく必要があります。
 --%>
${data.name}

<%-- 配列 or リストデータを指定した呼び出し --%>
<%-- 配列のインデックスは 0 から始まります。 --%>
${list[1].data.name}

<%-- マップデータのキーを指定した呼び出し --%>
${map[key].data.name}
```

図 8.2　JSP でセッションデータの呼び出し

8.3.2. WebRequest クラスを使用する場合

Spring MVC の場合、「org.springframework.web.context.request.WebRequest」クラスで、よく使う request、session スコープを代用することができます。

```
@Controller
@RequetMapping(value="/hoge")
public class SampleController {

    @RequestMapping
    public String hoge(WebRequest request, Model model) {

        // セッションから取得する
        LoginUser loginUser = (LoginUser) request.getAttribute(
                "loginUser", RequestAttributes.SCOPE_SESSION);

        // セッションに値を登録する
        String sessionData = "Session Data";
        request.setAttribute("sessionData", sessionData,   RequestAttributes.SCOPE_SESSION);

        // セッションからデータを削除する
        request.removeAttribute("message", RequestAttributes.SCOPE_SESSION);

        // リクエストに値を登録する
        String requestData = "request Data";
        request.setAttribute("requestData", requestData,   RequestAttributes.SCOPE_REQUEST);

        return "/fuga";
    }
}
```

図 8.3　WebRequest を使用したセッション情報の管理

8.3.3. SpringBean を使用したセッション管理

Spring の Bean をセッションに登録し、使用する方法を説明します。この方法を利用すると、インジェクションしインスタンスを取得できるので、わざわざ Servlet API や WebRequest 経由で取得しなくてもよくなります。

ただし、JSP からは直接呼び出すことができないため、Conrtoller で Model に登録して使うことになります。この方法を利用すると、様々な場所でセッションにデータをすることが制限され、管理がし易くなります。

①web.xml で、RequestContextListenr を登録します。

```
<web-app>
・・・省略
    <listener>
        <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
    </listener>
・・・省略
</web-app>
```

②Spring の設定ファイル servlet-context.xml に bean を登録します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- P 層(SpringMVC 用の SpringBean の設定ファイル。) -->
<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-3.0.xsd">
    <!-- session bean -->
    <bean id="loginUserBean" class="sample.web.bean.common.LoginUserBean" scope="session">
        <aop:scoped-proxy/>
    </bean>
</beans>
```

scope 属性を“session”または、“request”にして登録します。

③Controller での使い方

```
@Controller
@RequetMapping(value="/hoge")
public class SampleController {

    @Resource
    private LoginUserBean loginUserBean;

    @RequestMapping
    public ModelAndView hoge(WebRequest request) {
        // データの呼び出し
        loginUserBean.setName("admin");
        ModelAndView mav = new ModelAndView("/fuga");

        // JSP 内で使う場合は、Model に登録して使う
        model.addObject("loginUser", loginUserBean);

        return mav;
    }
}
```

Spring の Bean を通常のようにインジェクションし使用します。

JSP からは直接呼び出すことはできないため、Model に登録して使用します。

8.4. @SessionAttributes を使用したセッション管理

Command は通常、request スコープでデータを受け渡すため、次の画面までしか有効にはなりません。しかし、「@SessonAttribute」を使用すると、Model データで送受信するデータが session スコープに登録することができます。また、下記の例のように Command クラスは実際には Model 経由で送受信されるため、session スコープに登録されます。

```
@Controller
@RequestMapping(value="/search")
@SessionAttributes(value={"data","command"})
public class SearchController {

    @RequestMapping
    public ModelAndView search(@ModelAttribute ConditionCommand command){
        // 省略
        ModelAndView mav = new ModelAndVeiw("/result");
        mav.addObject("data", "Hello, Session Data");
        return mav;

    }

    @RequestMapping(value="clear")
    public ModelAndView clear(SessionStatus status) {

        //セッションのデータをクリアします。
        status.setComplete();

    }
}
```

・受信データのコマンド ConditionCommand は、session スコープに登録されます。
・Model に登録した送信データ「data」も session スコープに登録されます。

@SessionAttriutes で登録されているデータをクリアします。ここでは、名前が「command」「data」という session スコープの値が対象となります。

8.4.1. セッションが切れている場合の問題点(:TODO)

セッション(HttpSession)が切れている状態で、ブラウザなどからデータを送信されると、データを保存するセッション領域がないため例外「」が発生します。
メソッドに、@ModelAttribute を付与し、Command クラスのインスタンスを返すものを用意します。

・・・・・・・・・・・・・追加する

8.5. フラッシュスコープの実装

フラッシュスコープは、Spring3.1 で標準実装されています。Spring3.0 以前の場合でのフラッシュスコープの利用方法（実装方法）を説明します。

- ここでは、下記の URL で紹介されている方法をそのまま実装したものを紹介します。
  「http://d.hatena.ne.jp/ryoasai/20110402/1301750921」
  「https://jira.springsource.org/browse/SPR-6464」
- 使用方法について、「3.4.3　フラッシュスコープ（Flash Scope）を使用した redirect による」を参照してください。

8.5.1. フラッシュスコープの実装にあたって

フラッシュスコープを実装するために作成するファイル、編集するファイルを表 8.3 に示します。

表 8.3　フラッシュスコープの実装に必要なファイル

No.	作成/編集するファイル	説明	参照先
1	FlashMap.java	画面遷移する際の Model オブジェクトを格納するためのクラス。	8.5.2
2	FlashMapFilter.java	FlashMap に格納されたデータをリクエストスコープに転記後、セッションから自動的に削除するフィルタ。	8.5.3
3	FlashMapStoringRedirectViewResolver.java	フラッシュマップを処理するための Spring MVC の View Resolver。 View のパスに対する接頭語「redirect_with_flash:」を処理する。	8.5.4
4	web.xml	FlashMapFilter を登録する。	8.5.5
5	servlet-context.xml	Spring MVC 用の設定が書かれたファイル（「2.4.4　Spring MVC 用の設定ファイル」を参照のこと）。 FlashMapStroingRedirectViewResolver を登録する。	8.5.6

8.5.2. 「FlashMap.java」の実装

```
import java.util.HashMap;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;


/**
 * フラッシュスコープの実装。
 * <p>リクエストのデータを保持するクラス。
 *
 * @see http://d.hatena.ne.jp/ryoasai/20110402/1301750921
 *
 */
public class FlashMap implements Serializable {

	/** serialVersionUID */
	private static final long serialVersionUID = 1L;

	static final String FLASH_SCOPE_ATTRIBUTE = FlashMap.class.getName();

	public static Map<String, Object> getCurrent(HttpServletRequest request) {
		HttpSession session = request.getSession();

		synchronized (session) {
			@SuppressWarnings("unchecked")
			Map<String, Object> flash = (Map<String, Object>)
session.getAttribute(FLASH_SCOPE_ATTRIBUTE);
			if (flash == null) {
				flash = new HashMap<String, Object>();
				session.setAttribute(FLASH_SCOPE_ATTRIBUTE, flash);
			}

			return flash;
		}
	}

	private FlashMap() {

	}
}
```

8.5.3. 「FlashMapFilter.java」の実装

```
package sample.web;

import java.io.IOException;
import java.util.Map;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.springframework.web.filter.OncePerRequestFilter;


/**
 * FlashMap に格納されたデータをリクエストスコープに転記した後に、セッションから自動的に削除する。
 *
 * @see http://d.hatena.ne.jp/ryoasai/20110402/1301750921
 *
 */
public class FlashMapFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
            FilterChain filterChain) throws ServletException, IOException {

        HttpSession session = request.getSession(false);
        if (session != null) {
            synchronized (session) {
                @SuppressWarnings("unchecked")
                Map<String, ?> flash = (Map<String, ?>)
session.getAttribute(FlashMap.FLASH_SCOPE_ATTRIBUTE);
                if (flash != null) {

                    // フラッシュのデータが存在していたら、リクエストコンテキストにコピーする
                    for (Map.Entry<String, ?> entry : flash.entrySet()) {
                        Object currentValue = request.getAttribute(entry.getKey());
                        if (currentValue == null) {
                            request.setAttribute(entry.getKey(), entry.getValue());
                        }
                    }

                    // フラッシュを削除
                    session.removeAttribute(FlashMap.FLASH_SCOPE_ATTRIBUTE);
                }
            }
        }

        filterChain.doFilter(request, response);

    }
}
```

8.5.4. 「FlashMapStoringRedirectViewResolver.java」の実装

UrlBaseViewResolver のサブクラスを作成し、通常の forward、redirect 処理をオーバーライドする。

```
package sample.web;

import java.io.IOException;
import java.util.Locale;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.View;
import org.springframework.web.servlet.view.RedirectView;
import org.springframework.web.servlet.view.UrlBasedViewResolver;


/**
 * フラッシュマップを利用したリダイレクトを処理するクラス。
 * <p>フラッシュに保存するためには、プレフィックス「redirect_width_flash:」を付ける。
 *
 */
public class FlashMapStoringRedirectViewResolver extends UrlBasedViewResolver {

	public static final String REDIRECT_WITH_FLASH_URL_PREFIX = "redirect_with_flash:";

	private String redirectWithFlashUrlPrefix = REDIRECT_WITH_FLASH_URL_PREFIX;

	public FlashMapStoringRedirectViewResolver() {
		setViewClass(FlashMapStoringRedirectView.class);
	}

		// 柔軟性を高めるためプレフィックスは DI により外部で変更可能にしておく
	public String getRedirectWithFlashUrlPrefix() {
		return redirectWithFlashUrlPrefix;
	}

	public void setRedirectWithFlashUrlPrefix(String redirectWithFlashUrlPrefix) {
		this.redirectWithFlashUrlPrefix = redirectWithFlashUrlPrefix;
	}


	@Override
	protected Class<FlashMapStoringRedirectView> requiredViewClass() {
		return FlashMapStoringRedirectView.class;
	}

	@Override
	protected View createView(String viewName, Locale locale) throws Exception {
		if (!canHandle(viewName, locale)) {
			return null;
		}

		if (viewName.startsWith(getRedirectWithFlashUrlPrefix())) {
			String redirectUrl = viewName.substring(getRedirectWithFlashUrlPrefix().length());
			return new FlashMapStoringRedirectView(redirectUrl, isRedirectContextRelative(),
isRedirectHttp10Compatible());
```

```
        }

        return null; //リダイレクト時以外は後続のレゾルバーのチェーンに処理をまわす。
    }

    private static class FlashMapStoringRedirectView extends RedirectView implements View {

        public FlashMapStoringRedirectView(String redirectUrl, boolean redirectContextRelative, boolean
redirectHttp10Compatible) {
            super(redirectUrl, redirectContextRelative, redirectHttp10Compatible);
            setExposeModelAttributes(false); // リダイレクト時に URL のパラメーターとして値を埋め込む処理を無
効化する。
        }

        @Override
        protected void renderMergedOutputModel(
                Map<String, Object> model, HttpServletRequest request, HttpServletResponse response)
                throws IOException {

            // ここでフラッシュマップにモデルマップ中のデータを保存しておく
            FlashMap.getCurrent(request).putAll(model);

            super.renderMergedOutputModel(model, request, response);
        }

    }

}
```

8.5.5. 「web.xml」の編集

```
<web-app>
・・・・

    <!-- フラッシュマップフィルター -->
    <filter>
        <filter-name>flashMapFilter</filter-name>
        <filter-class>sample.web.FlashMapFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>flashMapFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

8.5.6. 「servlet-context.xml」の編集

Spring MVC では、Chain of responsibility パターンによって、ViewResolver に処理が委譲されます。その際に、order 属性の低いものから優先して処理が呼びだれるため、今回作成した FlashMapStroingRedirectViewResolver の優先度を高く設定します。

```
<beans>
・・・

    <!-- フラッシュマップによるリダイレクトを処理するリゾルバー -->
    <bean class="sample.web.FlashMapStoringRedirectViewResolver">
        <property name="order" value="1" />
    </bean>

    ・・・・

    <bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/view/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

8.6. JSP でのセッションデータの取得・設定方法(:TODO)

8.6.1. Servlet(:TODO)

8.6.2. Spring MVC(:TODO)

9. 権限チェック（認証・認可機能）

9.1. 「Spring Security」を利用した権限チェック（:TODO）

@Secured とかの紹介も行う

9.2. 独自実装のアノテーションを利用した権限チェック（Spring MVC 3.0）

Spring Sercurity は、Servlet Filter で権限チェックを実現しており、リクエストされた URL に対して権限チェックを行います。この方法は汎用性が高いが、URL に対してチェックを設定するファイルと、リクエストを処理する業務部分が別々になってしまい管理しづらい面があります。

Spring MVC は、@RequestMapping でリクエスト URL を定義するため、このアノテーションを付与したメソッドに対して、権限を定義すれば管理もしやすく、URL が変わった場合など変更する必要がありません。

9.2.1. 独自アノテーションを利用した権限チェックの実装にあたって

【作成するアノテーション／カスタムタグの仕様】

- ユーザ情報は、セッションに保持する。ユーザのセッション情報中に、ユーザの権限を持っている。
- アノテーション@Authorize(roles={権限})を付与したコントローラは、特定の権限を持っているユーザのみリクエストを処理する。
  また、セッションにユーザ情報がないようなログインしていない場合は、権限を持っていない場合と同様の動作をする。
- 権限は、属性 roles で指定する。
  権限の形式は、「処理@業務」の形式。たとえば、ユーザの検索権限は、「SEARCH@USER」。
  権限は複数指定でき、何れかの権限を持つ場合式を評価する。
- 実装するために作成するファイル、編集するファイルを表 9.1 に示します。

表 9.1 独自アノテーションによる権限チェックの実装に必要なファイル（Spring MVC 3.0）

No.	作成/編集するファイル	説明	参照先
1	LoginUserBean.java	ログイン情報を保持する JavaBeans。	9.2.2
2	Authorize.java	アノテーション@Authroize の定義ファイル。	9.2.3
3	AuthorizeHandlerMethodAspect.java	メソッドの@RequestMapping ごとに、アノテーション@Authorize を処理する Java ファイル。 AOP を使用し処理を実装する。	9.2.4
4	SessionTimeoutException.java	セッションにユーザ情報がない場合にスローする例外クラス。	9.2.5
5	InvalidRoleException.java	有効な権限を持っていない場合にスローする例外クラス。	9.2.6
6	servlet-context.xml	Spring MVC 用の設定が書かれたファイル（「2.4.4 Spring MVC 用の設定ファイル」を参照のこと）。 AuthorizeHandlerMethodAspect を登録する。 権限チェック関連の例外を処理する ExceptionResolver を登録する。	9.2.7

9.2.2. 「LoginUserBean.java」の実装

- システムにログインした際にアカウント情報を保持するクラスです。
- セッション上に登録されます。プロジェクトごとに異なります。
- メソッド「#hasRole()」にて、自身が権限を持つかチェックします。

```
import java.util.List;

import org.apache.commons.lang.StringUtils;

public class LoginUserBean extends UserInfoViewDto {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    /** サービスコードと操作コードの区切り文字 */
    public static final String SERVICE_SEPARATOR = "@";
    /** ユーザが所持している権限のリスト */
    protected List<RoleServicesViewDto> roleServiceList;

    /**
     * 権限を持つかどうかチェックする。
     * <p>書式は、以下の通り。
     * <ul>
     *    <li>完全な書式「operateCd@serviceCd」
     *    <li>操作コードを省略時「@serviceCd」
     *
     * @param role 操作権限。
     * @return true:権限を持っている時。
     */
    public boolean hasRole(final String role) {

        if(StringUtils.isEmpty(role)
                || !StringUtils.contains(role, SERVICE_SEPARATOR)
                || roleServiceList == null
                || roleServiceList.isEmpty()) {
            return false;
        }

        String split[] = role.split(SERVICE_SEPARATOR);
        if(split.length != 2) {
            // 書式が不正な場合
            return false;
        }

        final String operateCd = split[0].trim();
        final String serviceCd = split[1].trim();

        for(RoleServicesViewDto dto : roleServiceList) {

            // サービスコードの比較
            if(!serviceCd.equalsIgnoreCase(dto.getServiceCd())) {
                continue;
            }

            // サービスコードが等しく、操作コードが指定されていない
            if(operateCd.isEmpty()) {
                return true;
```

```
                }

                // 操作コードが指定されている場合
                if(operateCd.equalsIgnoreCase(dto.getOperateCd())) {
                    return true;
                }

            }

            return false;
        }

        /**
         * 権限を持つかどうかチェックする。
         */
        public boolean hasRole(String... roles) {
            for(String role : roles) {
                if(hasRole(role)) {
                    return true;
                }
            }

            return false;
        }

        // getter, setter は省略

}
```

9.2.3. 「Authorize.java」の実装

@RequestMapping は、クラスとメソッドに付与できるため、今回作成する@Authorize も同様に付与できるようにします。属性 roles の初期値は、{}とし、省略可能とします。

```
package sample.web.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import sample.core.exception.InvalidRoleException;
import sample.core.exception.SessionTimeoutException;


/**
 * アクセス制御を定義するアノテーション。
 * <p>コントローラの<code>RequestMapping</code>を定義しているメソッド、クラスに設定してください。
 * <p>以下のチェックを行う。
 *   <ol>
 *     <li>ユーザがログインしているかどうか。ログインしていない場合、{@link SessionTimeoutException}をスローする。
 *     <li>ログインユーザが指定した何れかの権限を持っているかどうか。権限を持っていない場合、{@link InvalidRoleException}をスローする。
 * <p>クラスとメソッドの両方にアノテーションが付加された場合、どちらからの権限を持っていれば認可される。
 * <p>判定処理は、{@link AuthorizeHandlerMethodAspect}で行う。
 *
 */
@Target(value={ElementType.METHOD, ElementType.TYPE})
@Retention(value=RetentionPolicy.RUNTIME)
@Documented
public @interface Authorize {

    /**
     * 権限を定義する。
     * <p>書式は<code>操作コード@サービスコード</code>。
     * <p>操作コードは省略が可能。
     * @return
     */
    String[] roles() default {};

}
```

9.2.4. 「AuthorizeHandlerMethodAspect.java」の実装

- リクエストが処理される部分を拡張するには、通常はインタフェース「**HandlerInterceptorAdapter**」を実装します。
  - しかし、このインタフェースではコントローラクラスが呼ばれたことは検知できますが、@RequestMapping が付与されたメソッドを別々に検知することはでません。
- そのため、@RequestMapping を処理しているクラス「**AnnotationMethodHandlerAdapter**」を拡張し実装を行います。
  - しかし、**実際の処理部分は private の内部クラスとなっており非常に拡張しにくいため**、AOP を利用し拡張を行います。詳細は、「http://d.hatena.ne.jp/ryoasai/comment/20110212/1297526255」を参照してください。

```
package sample.web.annotation;

import java.lang.reflect.Method;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.context.request.RequestAttributes;
import org.springframework.web.context.request.RequestContextHolder;

import sample.core.exception.InvalidRoleException;
import sample.core.exception.SessionTimeoutException;
import sample.web.common.bean.LoginUserBean;

/**
 * アノテーション{@link Authorize}を設定されたメソッドに対して、認証・認可の判定を行う。
 * @see http://d.hatena.ne.jp/ryoasai/comment/20110212/1297526255
 *
 */
@Aspect
public class AuthorizeHandlerMethodAspect {

    final static Logger logger = LoggerFactory.getLogger(AuthorizeHandlerMethodAspect.class);

    public static final String SESSION_LOGIN_USER = "secLoginUser";

    @Pointcut("execution(@org.springframework.web.bind.annotation.RequestMapping * *(..))")
    public void handlerMethod() {}

    @Before("handlerMethod()")
    public void interceptHandlerMethod(JoinPoint jp) throws Exception {

        RequestAttributes requestAttributes = RequestContextHolder.getRequestAttributes();
        MethodSignature methodSignature = (MethodSignature) jp.getSignature();

        Method method = methodSignature.getMethod();
        Class<?> clazz = method.getDeclaringClass();
```

AOP を使用し、ポイントカットを定義します。

```
        // クラスに付与されたアノテーション@Authorize を取得する。
        Authorize classAnno = clazz.getAnnotation(Authorize.class);
        if(classAnno != null && logger.isDebugEnabled()) {
            logger.debug("classAnnotation:Authorize roles={}", combineArray(classAnno.roles()));

        }

        // メソッドに付与されたアノテーション@Authorize を取得する。
        Authorize methodAnno = method.getAnnotation(Authorize.class);
        if(methodAnno != null && logger.isDebugEnabled()) {
            logger.debug("methodAnnotation:Authorize roles={}", combineArray(methodAnno.roles()));
        }

        // セッションからユーザ情報を取得する。
        LoginUserBean loginUser = (LoginUserBean) requestAttributes.getAttribute(
                SESSION_LOGIN_USER, RequestAttributes.SCOPE_SESSION);

        authorize(loginUser, classAnno, methodAnno);
    }

    /**
     * 認証認可の判定を行う。
     * @param loginUser
     * @param classAnno
     * @param methodAnno
     * @return
     * @throws SessionTimeoutException
     * @throws InvalidRoleException
     */
    private boolean authorize(final LoginUserBean loginUser, Authorize classAnno, Authorize methodAnno) throws
SessionTimeoutException, InvalidRoleException {

        // アノテーションがない場合
        if(classAnno == null && methodAnno == null) {
            return true;
        }

        // アノテーションがあり、ログインユーザがない場合
        if(loginUser == null) {
            throw new SessionTimeoutException();
        }

        // class アノテーションの判定
        if(classAnno != null && loginUser.hasRole(classAnno.roles())) {
            return true;
        }

        // method アノテーションの判定
        if(methodAnno != null && loginUser.hasRole(methodAnno.roles())) {
            return true;
        }

        // 権限が設定されていない場合、ログイン有無のみ判定する。
        if(classAnno.roles().length ==0 && methodAnno.roles().length == 0) {
            return true;
        }

        throw new InvalidRoleException();
    }
```

・ログイン情報に対して、指定した権限を持つかどうかチェックします。
・プロジェクトにより、対応するメソッドや処理に読み替えてください。

```
    private String combineArray(String[] array) {

        StringBuilder sb = new StringBuilder();
        for(int i=0; i < array.length; i++) {
            sb.append(array[i]);

            if(i < array.length-1) {
                sb.append(":");
            }
        }

        return sb.toString();
    }
}
```

9.2.5. 「SessionTimeoutException.java」の実装

- このクラスは、プロジェクトにより異なります。対応する他の例外クラスがあれば、そちらを使用してください。
- 「java.lang.RuntimeException」を継承し作成します。「java.lang.Exception」を継承した場合、Spring MVC が「java.lang.reflect.UndeclaredThrowableException」でラッピングしてしまい、exceptionResolver でうまく処理できません。

```
package sample.core.exception;

/**
 * ユーザがログインしていない場合にスローする例外。
 */
public class SessionTimeoutException extends RuntimeException {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;
}
```

9.2.6. 「InvalidRoleException.java」の実装

このクラスは、プロジェクトにより異なります。対応する他の例外クラスがあれば、そちらを使用してください。

```
package sample.core.exception;

/**
 * ログインユーザが権限を持っていない場合。
 */
public class InvalidRoleException extends RuntimeException {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    public InvalidRoleException() {

    }
}
```

9.2.7. 「servlet-context.xml」の編集

- 「AuthorizeHandlerMethodAspect」は、Proxy ベースの AOP で登録します。その際に、java のライブラリ「cglib」を使用することを宣言するために「<aop:aspectj-autoproxy proxy-target-class="true"/>」とします。
- 権限がない場合などにスローされる例外を処理する ExceptionResolver を定義します。Bean の id は、「exceptionResolver」と固定です。プロジェクトで独自に exceptionResolver を実装している場合は、そちらに処理を記述します。詳細は、「11.2　システム全体での例外ハンドリング」を参照してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

    <!-- Enables the Spring MVC @Controller programming mode
    <mvc:annotation-driven />

    <aop:aspectj-autoproxy proxy-target-class="true"/>
    <bean class="sample.web.annotation.AuthorizeHandlerMethodAspect" />

    <bean id="exceptionResolver"
        class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
        <property name="exceptionMappings">
            <props>
                <prop key="sample.core.exception.SessionTimeoutException">error/sessionTimeout</prop>
                <prop key="sample.core.exception.InvalidRoleException">error/invalidRole</prop>
                <prop key="java.lang.Exception">error/error</prop>
            </props>
        </property>
    </bean>

</beans>
```

・独自アノテーションを処理する「AuthorizeHandlerMethodAspect」を登録します。

・ExceptionResolver に権限を持っていない場合にスローされる例外の処理を追加します。

9.2.8. アノテーションを使用した権限チェックのサンプル

- メソッド「helloWorld1()」は、@Authroize は付与されていませんが、クラスに付与されている@Authorize があるので、ログインしているかどうかのみチェックします。
- メソッド「helloWorld2()」は、権限「AAA@BBBB」を持っているかどうかチェックします。
- メソッド「hwlloWorld3()」は、権限「EDIT@USER」または「SEARCH@USER」の何れかを持っているかどうかチェックします。

```
@Controller
@Authorize
public class HelloWorldAuthrizeController {
    // 権限チェック①
    @RequestMapping("/hello_authorize1")
    public ModelAndView helloWorld1() {

        String message = "Hello World Authorize,";
        return new ModelAndView("hello", "message", message);
    }

    // 権限チェック②
    @RequestMapping("/hello_authorize2")
    @Authorize(roles="AAA@BBBB")
    public ModelAndView helloWorld2() {

        String message = "Hello World Authorize,";
        return new ModelAndView("hello", "message", message);
    }

    // 権限チェック③
    @RequestMapping("/hello_authorize3")
    @Authorize(roles={"EDIT@USER","SEARCH@USER"})
    public ModelAndView helloWorld3() {

        String message = "Hello World Authorize,";
        return new ModelAndView("hello", "message", message);
    }

}
```

図 9.1　実装したアノテーション「@Authorize」の使用例

9.3. 独自実装のアノテーションを使用した権限チェック（Spring MVC 3.1）

HandlerInterceptor の実装「HandlerInterceptorAdapter」を使用した認証・認可の実装の例を説明します。実装に当たり、AspectJ の方式と共通している部分があります（「表 9.2」参照）。

- HanlderInteceptorAdapter は、Controller をリクエストされた際に呼ばれます。すなわち、@RequestMapping が付与されたクラス／メソッドが実行された際に、インターセプトし独自の処理を挟み込むことができます。
- Spring3.0 では、「HandlerInterceptorAdaptor」でリクエストをインターセプトしても、Controller のクラス情報までした取得できませんでした。@RequestMapping には、クラスとメソッドに付与できますが、実際のリクエスト URL に対する実装はメソッドであるため、メソッド情報が取得できないと非常に不便でした。
- Spring3.1 から、@RequestMapping を処理するクラスが「RequestMappingHandlerAdapter」⇒「AnnotationMethodHandlerAdapter」変更になり、**HandlerInterceptorAdapter に渡される引数のオブジェクトが「HandlerMethod」に変更になり、メソッド情報も取得できる**ようになりました。

表 9.2　独自アノテーションによる権限チェックの実装に必要なファイル（Spring MVC 3.1）

No.	作成/編集するファイル	説明	参照先
1	LoginUserBean.java	ログイン情報を保持する JavaBeans。 ※Spring MVC 3.0 の実装方法から変更ありません。	9.2.2
2	Authorize.java	アノテーション@Authroize の定義ファイル。 ※Spring MVC 3.0 から変更ありません。	9.2.3
3	AuthorizeHandlerIntercopter.java	メソッドの@RequestMapping ごとに、アノテーション@Authorize を処理する Java ファイル。 ※Spring MVC 3.1 用に新規に作成。	9.3.1
4	SessionTimeoutException.java	セッションにユーザ情報がない場合にスローする例外クラス。 ※Spring MVC 3.0 の実装方法から変更ありません。	9.2.5
5	InvalidRoleException.java	有効な権限を持っていない場合にスローする例外クラス。 ※Spring MVC 3.0 の実装方法から変更ありません。	9.2.6
6	servlet-context.xml	Spring MVC 用の設定が書かれたファイル（「2.4.4 Spring MVC 用の設定ファイル」を参照のこと）。 AuthorizeHandlerIntercepter を登録する。 権限チェック関連の例外を処理する ExceptionResolver を登録する。 ※Spring MVC 3.1 用に修正。	9.3.2

9.3.1. 「AuthorizeHandlerInterceptor.java」の実装

- HandlerIntercepter の実装である「HandlerInterceptorAdapter」を継承し、メソッド「**preHandle**」をオーバーライドします（「表 9.3　HandlerInterceptor のメソッド」参照）。
  - HandlerInterceptor を実装しても問題ありませんが、インタフェースのため 3 つのメソッドを全て実装する必要があるため、コード量が多くなります。
  - HandlerInterceptorAdapter は、HandlerInterceptor を使いすくするために存在するもので、必要なメソッドのみをオーバーライドすればよいようになっています。そのため、ソースを見るとわかりますがロジックは含まれていません。
  - 「10.4　ロケール（地域・言語）の切り替え」で紹介している同様の原理の「ThemeChangeIntercepter」も、HandlerInterceptorAdaptor を継承して作成されています。
- 引数「handler」には、Controller の@RequestMapping が定義されたメソッド情報が「HandlerMethod」のインスタンスとして渡されるため、タイプをチェックし、キャストして使用します。
  - クラス、メソッドのアノテーション「@Authorize」を取得後の処理は、「9.2.4 「AuthorizeHandlerMethodAspect.java」の実装」と同じです。
- セッション内にあるログイン情報は、引数「HttpServletRequest request」から取得します。

表 9.3　HandlerInterceptor のメソッド

No.	メソッド	説明
1	boolean preHandle(…)	・インターセプトしたメソッド呼び出しの前に実行されます。 ・戻り値 true とすると、Spring MVC の DispacheServlet による Chain 処理が続行されます。 ・AOP のアドバイス「before」に該当します。
2	void postHandle(…)	・インターセプトしたメソッドの呼び出し後に直後に実行されます。 ・ただし、View で画面を描画する前に実行されます。 ・AOP のアドバイス「after」に該当します。
3	void afterCompletion(…)	・インターセプトしたメソッドの呼び出し後に実行されます。 ・View で画面を描画した後に呼び出されます。 ・また、インターセプトしたメソッド内で例外がスローされても呼び出されます。 ・ただし、メソッド「preHandle」で true が返された場合にのみ呼びだれ、false を返し Chain が途中終了した場合は呼び出されません。 ・AOP のアドバイス「after returning」「after throwing」に害号します。

```
package sample.web.annotation;

import java.lang.reflect.Method;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.method.HandlerMethod;
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

import sample.core.exception.InvalidRoleException;
import sample.core.exception.SessionTimeoutException;
import sample.web.common.bean.LoginUserBean;

/**
 * コントローラに付与されたアノテーション「@Authorize」により、認証・認可チェックを行うインタセプター。
 */
public class AuthorizeHandlerInterceptor extends HandlerInterceptorAdapter {

    final static Logger logger = LoggerFactory.getLogger(AuthorizeHandlerIntercepter.class);

    public static final String SESSION_LOGIN_USER = "secLoginUser";

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {

        if(!(handler instanceof HandlerMethod)) {
            if(logger.isInfoEnabled()) {
                logger.info("handler type is not HandlerMethod : {}.", handler.getClass().getName());
            }
            return preHandle(request, response, handler);
        }

        final HandlerMethod handlerMethod = (HandlerMethod) handler;
        final Method method = handlerMethod.getMethod();
        final Class<?> clazz = method.getDeclaringClass();

        // クラスに付与されたアノテーション「@Authorize」を取得する。
        Authorize classAnno = clazz.getAnnotation(Authorize.class);
        if(classAnno != null) {
            logger.debug("Class Annotation Info : @Authorize roles={}", combineArray(classAnno.roles()));
        }

        // メソッドに付与されたアノテーション@Authorize を取得する。
        Authorize methodAnno = handlerMethod.getMethodAnnotation(Authorize.class);
        if(methodAnno != null && logger.isDebugEnabled()) {
            logger.debug("methodAnnotation:Authorize roles={}", combineArray(methodAnno.roles()));
        }

        // セッションからユーザ情報を取得する。
        final LoginUserBean loginUser = (LoginUserBean) request.getSession().getAttribute(
                SESSION_LOGIN_USER);

        authorize(loginUser, classAnno, methodAnno);

        return super.preHandle(request, response, handler);
    }
```

引数「handler」が、「HandlerMethod」のインスタンスかどうかチェックします。

```
    /**
     * 認証・認可の判定を行う。
     * <p>不正な場合は、例外をスローする。
     */
    protected boolean authorize(final LoginUserBean loginUser, Authorize classAnno, Authorize methodAnno)
            throws SessionTimeoutException, InvalidRoleException {

        // アノテーションがない場合
        if(classAnno == null && methodAnno == null) {
            return true;
        }

        // アノテーションがあり、ログインユーザがない場合
        if(loginUser == null) {
            throw new SessionTimeoutException();
        }

        // class アノテーションの判定
        if(classAnno != null && loginUser.hasRole(classAnno.roles())) {
            return true;
        }

        // method アノテーションの判定
        if(methodAnno != null && loginUser.hasRole(methodAnno.roles())) {
            return true;
        }

        // 権限が設定されていない場合、ログイン有無のみ判定する。
        if(classAnno.roles().length ==0 && (methodAnno == null || methodAnno.roles().length == 0)) {
            return true;
        }

        throw new InvalidRoleException();
    }

    private String combineArray(String[] array) {

        StringBuilder sb = new StringBuilder();
        for(int i=0; i < array.length; i++) {
            sb.append(array[i]);

            if(i < array.length-1) {
                sb.append(":");
            }
        }
        return sb.toString();
    }
}
```

9.3.2. 「servlet-context.xml」の編集

- 「AuthorizeHandlerInterceptor」は、インタセプター用の要素<mvc:interceptors>～</mvc:interceptor>に登録します。
- 権限がない場合などにスローされる例外を処理する ExceptionResolver を定義します。Bean の id は、「exceptionResolver」と固定です。プロジェクトで独自に exceptionResolver を実装している場合は、そちらに処理を記述します。詳細は、「11.2　システム全体での例外ハンドリング」を参照してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">

    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven/>

    <mvc:interceptors>
        <!-- コントローラに付与された@Authorize アノテーションを使用した認証・認可をチェックする。 -->
        <bean class="sample.web.annotation.AuthorizeHandlerInterceptor" />

    </mvc:interceptors>

    <bean id="exceptionResolver"
        class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
        <property name="exceptionMappings">
            <props>
                <prop key="sample.core.exception.SessionTimeoutException">forward:/common/login.html</prop>
                <prop key="sample.core.exception.InvalidRoleException">error/invalidRole</prop>
                <prop key="java.lang.Exception">error/error</prop>
            </props>
        </property>
    </bean>

</beans>
```

作成したインターセプターを登録します。

・ExceptionResolver に権限を持っていない場合にスローされる例外の処理を追加します。

9.4. 独自実装のカスタムタグを利用した権限処理

Spring Sercurity の<security:authorize>タグは、複雑な権限を持つ場合対応しきれない。例えば、業務ごとに処理の権限があり、その権限を持っているユーザのみボタンを表示するときなどです。

ここで紹介する方法は、比較的簡単に作成でき、Spring MVC 以外でも JSP を用いるものならば利用できます。

9.4.1. カスタムタグを利用した権限処理の実装にあたって

【作成するカスタムタグの仕様】

- ユーザ情報は、セッションに保持する。ユーザのセッション情報中に、ユーザの権限を持っている。
- タグ<auth:authorize roles=”権限”>～</auth:authorize>で囲まれた中は、特定の権限を持っているユーザのみ評価（表示）する。
  また、セッションにユーザ情報がないようなログインしていない場合は、権限を持っていない場合と同様の動作をする。
- 権限は、属性 roles で指定する。
  権限の形式は、「処理@業務」の形式。たとえば、ユーザの検索権限は、「SEARCH@USER」。
  権限は複数指定でき、半角カンマ「,」で区切り、何れかの権限を持つ場合式を評価する。

実装するために作成するファイル、編集するファイルを表 9.4 に示します。

表 9.4　カスタムタグを利用した権限処理の実装に必要なファイル

No.	作成/編集するファイル	説明	参照先
1	AuthorizeTag.java	カスタムタグを処理する Java の実装クラス。	
2	authorize.tld	カスタムタグの定義ファイル。	

9.4.2. 「AuthorizeTag.java」の実装

タグで囲んだボディ部分を評価するため、JSP API「javax.servlet.jsp.tagext.BodyTagSupport」を実装します。

```
import java.io.IOException;

import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.BodyTagSupport;

import org.apache.commons.lang.StringUtils;

import sample.web.common.bean.LoginUserBean;


/**
 * 権限を持つ場合，タグの中を評価するカスタムタグ
 * <p>セッション上にログイン情報がない場合、タグの中の評価は行わない。
 *
 */
public class AuthorizeTag extends BodyTagSupport {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    /** 権限コードの区切り文字 */
    private static final String SEPARATOR_CD = ",";

    public static final String SESSION_LOGIN_USER = "secLoginUser";

    /** 操作権限 */
    private String roles;

    private LoginUserBean loginUser = null;

    /**
     * 開始タグを処理する
     */
    @Override
    public int doStartTag() throws JspTagException {

        // セッションからユーザオブジェクトを取得する
        loginUser = (LoginUserBean) pageContext.getSession().getAttribute(SESSION_LOGIN_USER);
        if(loginUser == null) {
            // セッションに無い場合、ボディを評価しない。
            return SKIP_BODY;

        } else if(loginUser != null && StringUtils.isEmpty(roles)) {
            // セッションにユーザ情報があり、属性 role が設定されていない場合
            return EVAL_BODY_BUFFERED;
        }

        String[] cds = StringUtils.split(roles, SEPARATOR_CD);
        for(String cdValue : cds) {
            if(loginUser.hasRole(cdValue.trim())) {
                // ユーザが権限を持っていれば，ボディを評価する
```

・ログイン情報に対して、指定した権限を持つかどうかチェックします。
・プロジェクトにより、対応するメソッドや処理に読み替えてください。

```
                return EVAL_BODY_BUFFERED;
            }
        }

        // ユーザが権限を持っていなければ，ボディを評価しない
        return SKIP_BODY;

    }

    /**
     * タグの中身を処理
     */
    @Override
    public int doAfterBody() throws JspTagException {

        // ボディの内容を取得
        BodyContent body = getBodyContent();
        try {
            // ボディを出力
            body.writeOut(getPreviousOut());

        } catch (IOException e) {
            throw new JspTagException(e);
        }
        body.clearBody();

        return SKIP_BODY;

    }

    /**
     * 終了タグを処理
     */
    @Override
    public int doEndTag() {

        return EVAL_PAGE;
    }

    @Override
    public void release() {
        super.release();
        loginUser = null;
        roles = null;

    }

    public String getRoles() {
        return roles;
    }

    public void setRoles(String roles) {
        this.roles = roles;
    }
}
```

JSP で定義した属性「roles」は、setter 経由で値が渡されます。

9.4.3. 「authorize.tld」の実装

- tld ファイルは、クラスパス直下に配置します。通常は、「/WEB-INF/lib/」以下に配置します。
  - Servlet3.0（Tomcat7 以降）の環境では、tld ファイルは「/WEB-INF/」の直下に配置します。
- JSP2.0 から、web.xml に tld の定義を不要です。

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
    version="2.0">

    <description>トータルシステムのカスタムタグ</description>
    <tlib-version>1.1</tlib-version>
    <short-name>auth</short-name>
    <uri>http://www.example.co.jp/authorize/tags</uri>

    <tag>
        <description>権限を持つ場合、タグを評価する。</description>
        <name>authorize</name>
        <tag-class>sample.web.taglib.AuthorizeTag</tag-class>
        <body-content>JSP</body-content>

        <attribute>
            <description>サービス操作。複数指定する場合は「,」で区切る。</description>
            <name>roles</name>
            <required>false</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>

    </tag>

</taglib>
```

プロジェクトで URI が決まっている場合は、変更してください。

作成したクラスを登録します。

・属性「roles」の定義。
・値は、AuthorizeTag#setRoles()経由で設定されます。

9.4.4. カスタムタグを使用した権限チェックのサンプル

```
<%@ taglib uri="http://www.example.co.jp/authorize/tags" prefix="auth"%>

<auth:authorize>
権限設定なし。<%-- 権限に関係なく、ログインしている場合表示されます。 --%>
</auth:authorize>

<auth:authorize roles="SEARCH@USER">
権限設定あり。SEARCH@USER<br>
</auth:authorize>

<auth:authorize roles="SEARCH@USER, EDIT@USER">
権限設定あり(複数)。SEARCH@USER, EDIT@USER<br>
</auth:authorize>
```

カスタムタグの利用する際の宣言。

10. 国際化

10.1. JSP からプロパティファイルの値を呼び出す

外部に定義されたプロパティファイルの値を JSP から呼び出す方法を説明します。JSP から値を呼び出すには、カスタムタグ「<spring:message />」を使用します。引数をとることもでき、デフォルトでは半角カンマ「,」で区切り文字を指定します。

```
<div>
  <ol>
    <li>引数なし：<spring:message code="label.item" /></li>
    <li>引数あり(1 つ)：<spring:message code="label.item1" arguments="${index}" /></li>
    <li>引数あり(複数)：<spring:message code="label.item2" arguments="${index},引数 2" /></li>
  </ol>
</div>
```

図 10.1　プロパティファイルの値の呼び出し

メッセージを定義するプロパティは、Spring Bean の「messageSource」から利用できるようにします。メッセージは、F 層などからも呼び出すことがあるので、ApplicationContext.xml に定義します。

```
<beans xmlns="http://www.springframework.org/schema/beans">

    <!-- 共通のメッセージファイル -->
    <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>classpath:message/message</value>
                <value>classpath:message/label</value>
            </list>
        </property>
    </bean>

</beans>
```

・“classpath:”をパスの前に付けると、クラスパス上のリソースを検索します。
・”file:”をパスの前に付けると、システムパス上のリソースを検索します。
・クラスパスで指定した場合は、拡張子を省略し指定します。

図 10.2　SpringBean「messageSource」の定義

プロパティファイルの例を図 10.3 に示します。引数をとる場合、「{添え字}」で指定します。添字は 0 から始まります。Java5 から導入された XML 形式プロパティファイルも Spring はサポートしています(図 10.4)。

```
## メッセージの定義
label.item=引数なしのメッセージ
label.item1=引数有りのメッセージ 1。引数 1={0}
label.item2=引数有のメッセージ 2。引数 1={0}、引数 1={1}
```

図 10.3　プロパティファイル「messsage/label.properties」の例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>XML 形式のプロパティファイル</comment>

    <entry key="label.item">引数なしのメッセージ</entry>
    <entry key="label.item1">引数有りのメッセージ 1。引数 1={0}</entry>
    <entry key="label.item2">引数有のメッセージ 2。引数 1={0}、引数 1={1}</entry>

</properties>
```

図 10.4 プロパティファイル「message/label.xml」の例

表 10.1 <spring:message />タグの属性

No.	属性	必須	説明
1	arguments	×	メッセージの引数を指定します。デフォルトでは、半角カンマ「,」で区切り複数指定することができます。
2	argumentSeparator	×	メッセージの引数の区切り文字。指定しない場合、デフォルトの半角カンマ「,」が指定されます。
3	code	×	プロパティファイルのキーを指定します。 キーが見つからず、属性「text」を指定している場合、属性「text」の値が出力されます。
4	htmlEscape	×	出力する値を HTML エスケープします。“true”の場合エスケープされます。デフォルト値は“false”です。
5	javaScriptEscape	×	出力する値を JavaScript エスケープします。“true”の場合エスケープされます。デフォルト値は“false”です。
6	message	×	不明です。
7	scope	×	属性「var」を指定した際に、その変数のセッションスコープを指定します。「page、request、session、application の何れかを指定できます。
8	text	×	属性「code」で指定したキーが存在しない場合に出力するメッセージを指定します。初期値は null です。
9	var	×	セッションに登録されているオブジェクトの名前を指定します。

10.2. Controller、Service(F 層)からプロパティファイルの値を呼び出す

Controller、Service など Spring で管理しているクラスからプロパティファイルに定義した値を呼び出すには、クラス「org.springframework.context.MessageSource」を Spring Bean としてインジェクションします。

また、クラス MessageSource からメッセージを呼び出した場合、使いづらい部分があるため、「org.springframework.context.support.MessageSourceAccessor」を使用します。MessageSourceAccessor は、MessageSource をラップして使用します。

```
@Controller
@RequestMapping(value="/hoge")
public class SampleController {

    @Resource
    private MessageSource messageSource;

    @RequestMapping
    public ModelAndView hoge() {

        ModelAndView mav = new ModelAndView();

        String message = null;
        // MessageSource を使用する場合
        message = messageSource.getMessage("error.01", null, null);

        // MessageSourceAccessor を使用する場合
        MessageSourceAccessor messageAccessor = new MessageSourceAccessor(messageSource);
        message = messageAccessor.getMessage("error.01");

        mav.addObject("message", message);

        return mav;
    }
}
```

MessageSource の場合、メッセージの引数がない場合も設定しないといけないため、使い勝手が悪い。

図 10.5　Controller から MessageSource を呼び出す

10.3. テーマの設定

CSS やヘッダー画像を切り替え、ユーザが好きなデザインを選択できる方法を説明します。

10.3.1. テーマの切り替えの基本設定

【servlet-context.xml】

- インタセプターとして、テーマの変更を行う「ThemeChangeIntercepter」を登録します。
  - プロパティ「paramName」として、テーマを切り替える際のパラメータ名を設定します。デフォルトだと「theme」になります。
- テーマごとの設定ファイルを管理する「ThemeSource」の実装クラス「ResourceBundleThemeSource」を定義します。
  - プロパティ「basenamePrefix」で、設定ファイルを格納するパッケージの位置をクラスパスの形式で指定します。デフォルトはルート(/WEB-INF/classes/)です。
  - パッケージを指定する際には、「theme.」のように、ドット“.”を付けます。
- 「表 10.2　様々な ThemeResolver」にあるテーマを管理する ThemeResolver を定義します。

```
<beans>
    ・・・省略
    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven/>

    <mvc:interceptors>
        <!-- テーマの変更を実行するクラス -->
        <bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor">
            <property name="paramName" value="themeCode"/>
        </bean>

    </mvc:interceptors>

    <!-- テーマの設定ファイル。 -->
    <bean id="themeSource" class="org.springframework.ui.context.support.ResourceBundleThemeSource">
        <!-- 設定ファイルを格納するパッケージを指定する -->
        <property name="basenamePrefix" value="theme."/>
    </bean>

    <!-- テーマを制御するリゾルバ -->
    <bean id="themeResolver" class="org.springframework.web.servlet.theme.CookieThemeResolver">
        <property name="cookieName" value="themeCode"/>

        <property name="defaultThemeName" value="normal"/>
    </bean>
    ・・・省略
</bean>
```

テーマを切り替える際のパラメータ名を指定します。デフォルトは、「theme」です。

テーマごとの設定ファイルを管理するクラス。

デフォルトのテーマを設定します。

【テーマの設定ファイルの作成】

- クラスパス上に配置するので、Maven 形式のプロジェクトの場合、「src/main/resources」以下に作成します。
- パッケージに格納する場合、「servlet-contxt.xml」の「ResourceBundleThemeSource」で設定した値と合わせる必要があります。
- テーマごとに設定ファイルを作成します。ファイル名がテーマ名となります。

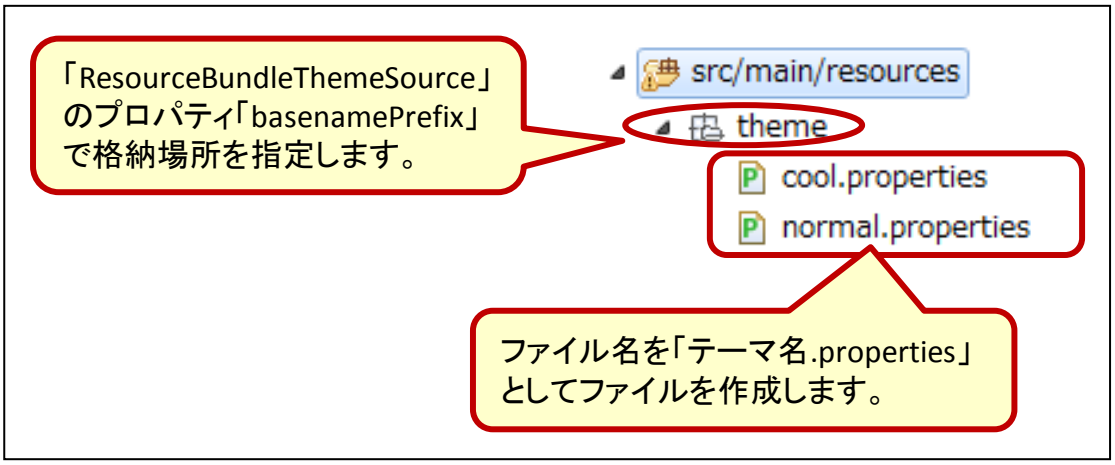


図 10.6　テーマの設定ファイルの格納場所

- 設定ファイルの中身は、通常のプロパティファイルと同じです。プロパティファイルのキーは、テーマごとに用意する必要があります。

「theme/cool.properties」の中身

```
css=/css/cool.css
background=/images/cool_image.jpg
```

「theme/normal.properties の中身」

```
css=/css/normal.css
background=/images/normal_image.jpg
```

【JSP の作成】

- テーマの設定ファイルから値を取得するためのカスタムタグとして<spring:theme>を使用します。
  - 属性「code」にて、設定ファイルで定義したキーを指定します。
  - カスタムタグの詳細な使用はを「12.1.8　カスタムタグ<spring:theme>」参照してください。
- テーマを切り替えるためのリンク「?themeCode=テーマ名」を定義します。
  - パラメータ「themeCode」は、「servlet-context.xml」の「ThemeChangeIntercepter」で定義した値になります。

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<html>
<head>
    <title>Spring MVC：Test</title>
    <link rel="stylesheet" href="<spring:theme code='css'/>" type="text/css"/>
</head>
<body>

<p>テーマの切り替え</p>
<div>
    <a href="?themeCode=normal">Normal</a><br/>
    <a href="?themeCode=cool">Cool</a>
</div>

</body>
</html>
```

テーマ用のプロパティファイルで定義した「css」という値を呼び出します。

テーマ用のプロパティファイルで定義した「css」という値を呼び出します。

【生成された HTML：テーマ“cool”を選択した場合】

```
<html>
<head>
    <title>Spring MVC：Test</title>
    <link rel="stylesheet" href="/css/cool.css" type="text/css"/>
</head>
<body>

<p>テーマの切り替え</p>
<div>
    <a href="?theme=normal">Normal</a><br/>
    <a href="?theme=cool">Cool</a>
</div>
</body>
</html>
```

プロパティファイル「cool.properties」に定義した、プロパティ「css」の値が出力されます。

10.3.2. 様々な ThemeResolver

- ThemeResolver は、テーマ情報の指定、保存方法により予め複数の実装があります(表 10.2)。
  - 「org.springframework.web.servlet.ThemeResolver」を実装している必要があります
- ThemeResolver は、sevlet-context.xml に 1 つのみ定義可能で、複数指定することはできません。

表 10.2　様々な ThemeResolver

No.	クラス名(※1)	説明
1	FixedThemeResolver	テーマをサーバ起動時に決定し、固定にします。
2	SessionThemeResolver	セッションが有効な間、切り替えたテーマ情報を保持します。
3	CookieThemeResolver	Cookie が有効な間、切り替えたテーマ情報を保持します。

※1 パッケージ「org.springframework.web.servlet.theme」に格納されています

【servlet-context.xml】

- プロパティ「defaultThemeName」でデフォルトのテーマ名を指定します。
  - 必ず指定しておきます。
- CookieThemeResolver の場合、Cookie の有効期限など様々な設定値があります。

```
<beans>
・・・省略
    <!-- テーマ情報を起動時に決定する -->
    <bean id="themeResolver" class="org.springframework.web.servlet.theme.FixedThemeResolver">
        <property name="defaultThemeName" value="normal"/>
    </bean>

    <!-- テーマ情報をセッションに保持する -->
    <bean id="themeResolver" class="org.springframework.web.servlet.theme.SessionThemeResolver">
        <property name="defaultThemeName" value="normal"/>
    </bean>

    <!-- テーマ情報を Cookie に保持する -->
    <bean id="themeResolver" class="org.springframework.web.servlet.theme.CookieThemeResolver">
        <!-- Cookie 名 -->
        <property name="cookieName" value="themeCode"/>
        <!-- Cookie 有効期限（秒）。-1 の場合、ブラウザを閉じるまで有効。指定しない場合は Cookie が削除されるまで有効。 -->
        <property name="cookieMaxAge" value="-1"/>
        <!-- SSL を使用するかどうか -->
        <property name="cookieSecure" value="false"/>

        <property name="defaultThemeName" value="normal"/>
    </bean>
・・省略
</beans>
```

10.4. ロケール（地域・言語）の切り替え

ユーザごとにメッセージなどを切り替え、１つのシステムで様々な言語に対応する方法を説明します。

- LocaleResolver で切り替えるには、「MessageSource」経由で取得しているメッセージが対象です。
- JSP のカスタムタグ<spring:message>も MessageSource 経由で取得しています。

10.4.1. ロケールの切り替えの基本設定

【ApplicationContext.xml】

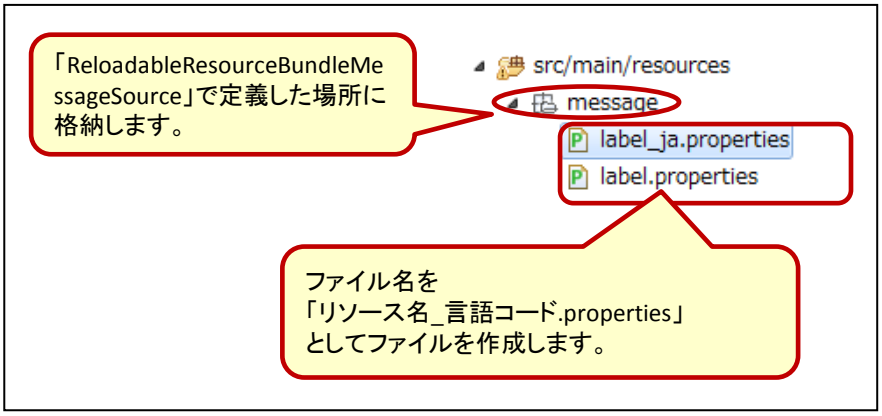
- Spring Bean「messageSource」を定義します。
- MessageSource を取得する実装は多数ありますが、今回はよく使われる「ReloadableResourceBundleMessageSource」を定義します。

```
<beans>
・・・省略
    <!-- 共通のメッセージファイル -->
    <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>classpath:message/label</value>
                <value>classpath:app</value>
            </list>
        </property>
        <property name="fallbackToSystemLocale" value="false"/>
    </bean>
・・・省略
</beans>
```

プロパティファイルの位置をクラスパス形式で指定します。
パッケージは、「/」で区切ります。
ファイルの拡張子、「.properties」「.xml」は省略します。

【メッセージファイルの作成】

- クラスパス上に配置するので、Maven 形式のプロジェクトの場合、「src/main/resources」以下に作成します。
- パッケージに格納する場合、「ApplicationContext.xml」の「ReloadableResourceBundleMessageSource」で設定した値と合わせる必要があります。
- 言語キーごとにファイルを作成します。
  - 言語キーがないデフォルトのプロパティファイルは必ず用意しておきます。該当しない言語キーやプロパティが見つからない場合に、デフォルトのファイルから値を取得します。



【servlet-context.xml】

- Spring MVC 管理下の Controller をにおいて、ロケール切り替え用の Intercepter である「LocaleChangeIntercepter」を指定します。
  - ➢ プロパティ「paramName」にて、言語を切り替える際の URL のパラメータ名を定義します。
    例えば、「?lang=ja」にリンクすると、日本語に切り替えられます。
- ロケールを処理する実装クラス「id="localeResolver"」を定義します。
  実装によりパラメータは異なります。

```
<beans>
・・・省略

    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven/>

    <mvc:interceptors>
        <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
            <property name="paramName" value="lang"/>
        </bean>
    </mvc:interceptors>

    <!-- ロケールを制御するリゾルバ -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
        <property name="cookieName" value="clientLocale"/>

        <!-- デフォルトの言語を指定する -->
        <property name="defaultLocale" value="ja"/>
    </bean>
・・・省略
</beans>
```

言語を切り替える際のパラメータ名を指定します。

【JSP】

- 言語切り替え用のリンク「?lang=言語キー」を用意します。
  - クエストリングのパラメータ名「lang」は、「servlet-context.xml」の「LocaleChangeIntercepter」で定義したものになります。
- を切り替えるためのリンク「?themeCode=テーマ名」を定義します。

```
<div>
    <a href="?lang=en">English</a><br/>
    <a href="?lang=ja">日本語</a>
</div>

<p>
    message:<spring:message code="message.hello" />
</p>
```

10.4.2. 様々な LocaleResolver

- Spring MVC には様々な方法でロケールを切りかえる方法があり、インタフェース「org.springframework.web.servlet.LocaleResolver」を実装している必要があります(表 10.3)。
- LocaleResolver は、sevlet-context.xml に 1 つのみ定義可能で、複数指定することはできません。

表 10.3　様々な LocaleResolver

No.	クラス名(※1)	説明
1	AcceptHeaderLocaleResolver	ブラウザの言語(クライアント OS の言語)設定をもとにロケールを判定します。 実際には、HTTP ヘッダーの「accept-language」をもとに判定します。
2	CookieLocaleResolver	Cookie が有効な間、切り替えたロケール情報を保持します。
3	SessionLocaleResolver	セッションが有効な間、切り替えたロケール情報を保持します。
4	FixedLocaleResolver	Java VM の言語設定をもとにロケールを判定します。 JVM オプション「-Duser.language」「-Duser.region」「-Duser.country」または、環境変数「LANG」により言語情報を設定します。

※1 パッケージ「org.springframework.web.servlet.i18n」に格納されています。

【servlet-context.xml】

- プロパティ「defaultLocale」でデフォルトのロケールを指定します。
  - 省略した場合は、JVM(システム)の設定値をもとに設定されます。
- CookieLocaleResolver の場合、Cookie の有効期限など様々な設定値があります。

```
<beans>
・・・省略
    <!-- ロケール情報をブラウザから判定する -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver">
        <!-- デフォルトのロケールを指定する。指定しない場合は JVM(システム)のロケール。 -->
        <!-- <property name="defaultLocale" value="en"/> -->
    </bean>

    <!-- ロケール情報を Cookie に保持する -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
        <!-- Cookie 名 -->
        <property name="cookieName" value="locale"/>
        <!-- Cookie 有効期限（秒）。-1 の場合、ブラウザを閉じるまで有効。指定しない場合は Cookie が削除されるまで有効。 -->
        <property name="cookieMaxAge" value="-1"/>
        <!-- SSL を使用するかどうか -->
        <property name="cookieSecure" value="false"/>

        <!-- デフォルトのロケールを指定する。指定しない場合は JVM(システム)のロケール。 -->
        <!-- <property name="defaultLocale" value="en"/> -->
    </bean>

    <!-- ロケール情報をセッションに保持する -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
        <!-- デフォルトのロケールを指定する。指定しない場合は JVM(システム)のロケール。 -->
        <!-- <property name="defaultLocale" value="en"/> -->
    </bean>

    <!--  -->
    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.FixedLocaleResolver">
        <!-- デフォルトのロケールを指定する。指定しない場合は JVM(システム)のロケール。 -->
        <!-- <property name="defaultLocale" value="en"/> -->
    </bean>
・・省略
</beans>
```

11. 例外処理

11.1. Controller での例外ハンドリング「@ExceptionHandler」

- Controller 内で発生した例外は、専用のメソッドを用意し、「@ExceptionHandler」を付与することで、処理することができます。
- アノテーションの引数に処理対象の例外クラスを指定します。複数設定することもできます。
- この方法は、Controller クラス内で共通なので、複数のリクエスト URL を処理するような場合では、単純に、try-catch 句で処理することをお勧めします。

```
@Controller
@RequestMapping("/test/sample7")
public class Sample7Controller {

    @ExceptionHandler(IOException.class)
    public ModelAndView handleIOException(IOException exception, WebRequest request) {
        ModelAndView mav = new ModelAndView("/error/error");
        mav.addObject("message", "IOException が発生しました。");
        return mav;
    }

    @ExceptionHandler({DataNotFoundException.class, TokenException.class})
    public ModelAndView handleServiceException(Exception exception, WebRequest request) {
        ModelAndView mav = new ModelAndView("/error/error");
        mav.addObject("message", "ServiceException が発生しました。");
        return mav;
    }

    @ExceptionHandler(DataNotFuondException.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public void handleIOException(DataNotFuondException exception) {
        //TODO: 例外処理
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction1(@ModelAttribute("sample7Command") Sample7Command command,
            BindingResult bindingResult) throws IOException, DataNotFoundException, TokenException {

        // TODO:入力値検証
        // 例外が発生するクラスの呼び出し。
        doServie1();
        doServie2();

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }

    // 例外「IOException」が発生するメソッド
    private void doServie1() throws IOException {
        System.out.println("doService1");

        //TODO: 処理を行う。
        throw new IOException("");
    }

    // 例外「DataNotFoundException」「TokenException」が発生するメソッド
    private void doServie2() throws DataNotFoundException, TokenException {
```

1 種類の例外を処理します。

2 種類の例外を処理します。

HTTP ステータスコードを返します。

```
        System.out.println("doService2");

        //TODO: 処理を行う。
        throw new TokenException();
    }
}
```

11.1.1. 例外処理用メソッドの引数と戻り値

- 例外処理用メソッドは、「表 11.1 例外処理用メソッドの引数一覧」「表 11.2 例外処理用メソッドの戻り値一覧」に示す様々な引数と戻り値をとることができます。
  多くが、@RequestMapping を付与した通常処理のメソッドと共通しており、使用方法も同じです（「3.2 コントローラの引数と戻り値」を参照）。
- ModelAndView を使用すると、遷移先に対してメッセージ、エラーの種類など渡すことができます。

表 11.1 例外処理用メソッドの引数一覧

No.	引数の型	説明	I/O
1	java.lang.Exception / java.lan.RuntimeException	処理したい例外を設定します。 必須です。	I
2	ServletRequest /HttpServletRequest	Servlet API のリクエスト。 通常は、HttpServletRequest を利用します。	I
3	ServletResponse /HttpServletResponse	Servlet API のレスポンス。 通常は、HttpServletResponse を利用します。	O
4	HttpSession	Serlvet API のセッション。NULL になるこ	I/O
5	org.springframework.web.context.request.WebRequest /org.springframework.web.context.request.NativeWebRequest	Session、Request 情報など取得／設定する際に、Servlet API の代わりに利用します。 スコープを指定して、#getAttribute()、#setAttribute()など操作できます。	I/O
6	java.util.Locale	現在のロケール情報を取得できます。 「LocaleResolver」で環境の変更を行うことができます。	I
7	java.io.InputStream /java.io.Reader	リクエストされたコンテンツの情報の入力ストリームで、 Servlet API から取得した値です。	I
8	java.io.OutputStream /java.io.Writer	レスポンスするコンテンツの情報を出力ストリームで、 ServletAPI から取得した値です。	O

表 11.2　例外処理用メソッドの戻り値一覧

No.	引数の型	説明
1	ModelAndView	View で指定した URL に、Model(データ)を渡す際に利用します。
2	Model	View を暗黙的に決めて、Model を設定します。 View の URL は RequestToViewNameTranslator で決まり、通常は現在の URL と変わりません。
3	java.util.Map	Model と同じです。 ModelMap は、Map を実装しているため、ModelMap のインスタンスを返しても問題ありません。
4	View	View で指定した URL に遷移します。 Model は暗黙的に決まり、引数に Model、@ModelAttribute がある場合は、その値を Model とします。 View の実装クラスには様々なものが用意されており、種類により JSON 型や PDF、Excel など様々なタイプを View として扱うことができます。詳細は、「3.5　ViewResolver」を参照してください。
5	String	View の URL を直接記述します。View と同様です。 Model は暗黙的に、View の場合と同様に決まります。
6	void	自身にレスポンスを返します。View を省略した場合と同様に URL は RequestToViewNameTranslator により決まります。 メソッドに「@ResponseStatus」が付与されている場合、指定した HTTP ステータスコードを返します。
7	void (引数に@ResponseBody がある場合)	引数で指定した ResponseBody を返します。HttpMessageConverter で値が変換されます。JSON、XML など通常の JSP 以外を返す場合に利用します。

11.1.2. AnnotationMethodHandlerExceptionResolver を明示的に定義する

アノテーション「@ExceptionHandler」は「AnnotationMethodHandlerExceptionResolver」で処理されます。servlet-context.xml に「<mvc:annotation-driven/>」が記述されている場合、標準で設定されているため、特に設定は必要ありません。

明示的に定義するには次のように定義します。

```
<beans>
・・・省略
    <!-- Spring MVC 標準の ExceptionResolver -->
    <bean name="annotationMethodHandlerExceptionResolver"
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver" />
・・・省略
</beans>
```

11.2. システム全体での例外ハンドリング

Spring MVC では、システム共通の例外は、「org.springframework.web.servlet.HandlerExceptionResolver」を実装したクラスを、Spring Bean 名「exceptionResolver」として登録し処理します。

11.2.1. SimpleMappingExceptionResolver を使用する

Spring MVC で予め用意されている、例外クラスと遷移先 URL を関連付ける方式を説明します。

- Spring Bean として、「id="exceptionResolver"」を登録します。
  クラスは、「org.springframework.web.servlet.handler.SimpleMappingExceptionResolver」を使用します。
- 例外ごとに遷移先を設定します。遷移先は、View クラスと同様の指定の仕方です。
  「forward」「redirect」を使用したい場合は、URL の接頭語として記述します。
- 例外処理の記述は、try-catch 句と同様に上から順に実行されます。
  継承関係を注意し、継承元（スーパクラス）は下方に定義します。
  例) IllegalArgumentException のスーパクラスは、RuntimeException であるため、それよりも上に定義します。
- 「java.lang.Exception」を継承している例外で、Controller 内でスローされた例外は、「java.lang.reflect.UndeclaredThrowableException」でラップされ意図したマップができません。
  そのような場合は、「java.lang.RuntimeException」を継承した例外クラスをスローするようにします。

【servlet-context.xml の編集】

```
<beans>
・・・省略
    <bean id="exceptionResolver"
        class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
        <property name="exceptionMappings">
            <props>
                <!-- forward で遷移する -->
                <prop key="sample.core.exception.SessionTimeoutException">forward:/common/login.html</prop>

                <!-- ファイルアップロード時の例外処理 -->
                <prop
key="org.springframework.web.multipart.MaxUploadSizeExceededException">error/fileupload</prop>

                <prop key="java.lang.IllegalArgumentException">error/error</prop>
                <prop key="java.lang.RuntimeException">error/error</prop>

                <!-- 上記に該当しない例外の処理 -->
                <prop key="java.lang.Exception">error/error</prop>
            </props>
        </property>
    </bean>
・・・省略
</beans>
```

継承関係に注意し、継承元は下方に記述する。

該当しないその他の例外の遷移先を設定します。

11.2.2. DefaultHandlerExceptionResolver を使用する

Spring MVC の標準の ExceptionResolver です。

- 「表 11.3　例外と HTTP ステータスコード」に示す各種例外が起きた場合に、特定の HTTP ステータスコードを返します。
- servlet-context.xml に Spring Bean として「exceptionResolver」を登録していない場合にも使用されます。明示的に指定することもできます。

表 11.3　例外と HTTP ステータスコード

No.	例外	HTTP ステータスコード
1	ConversionNotSupportedException	500 (Internal Server Error)
2	HttpMediaTypeNotAcceptableException	406 (Not Acceptable)
3	HttpMediaTypeNotSupportedException	415 (Unsupported Media Type)
4	HttpMessageNotReadableException	400 (Bad Request)
5	HttpMessageNotWritableException	500 (Internal Server Error)
6	HttpRequestMethodNotSupportedException	405 (Method Not Allowed)
7	MissingServletRequestParameterException	400 (Bad Request)
8	NoSuchRequestHandlingMethodException	404 (Not Found)
9	TypeMismatchException	400 (Bad Request)

【servlet-context.xml の編集】

```
<beans>
・・・省略
    <!-- Spring MVC 標準の ExceptionResolver -->
    <bean id="exceptionResolver"
class="org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver"/>
・・・省略
</beans>
```

11.2.3. 独自実装した HandlerExceptionResolver を使用する

SimpleMappingExceptionResolver では機能が足りない場合は、独自の例外処理を定義します。例えば、ログ出力処理や、エラーメッセージ、内容を動的に設定したい場合に使用します。

【例外処理クラスの作成】

- インタフェース「org.springframework.web.servlet.HandlerExceptionResolver」を実装し作成します。
- 戻り値として ModelAndView を取るので、Controller と同様に、遷移先やメッセージなどの Model オブジェクトを自由に設定できます。
- Controller 内でスローされた例外のうち、「java.lang.RuntimeException」を継承していないものは、「java.lang.reflect.UndeclaredThrowableException」にラップされます。
  そのため、「Exception#getCause( )」から原因となる例外を取り出し処理します。
- ログなどを出力したりなどの、自由に処理を記述できます。

```
package sample.web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import sample.core.exception.InvalidRoleException;
import sample.core.exception.SessionTimeoutException;
import sample.core.exception.TokenException;


/**
 * 独自の例外処理を行う。
 */
public class SystemExceptionResolver implements HandlerExceptionResolver {

    // logger
    static Logger logger = LoggerFactory.getLogger(SystemExceptionResolver.class);

    @Override
    public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response,
            Object handler, Exception ex) {

        ModelAndView mav = new ModelAndView();

        Throwable exception = ex;
        if((ex instanceof UndeclaredThrowableException) && (ex.getCause() != null)) {
            exception = ex.getCause();
        }

        if(exception instanceof SessionTimeoutException) {
            mav.setViewName("error/sessionTimeout");

        } else if(exception instanceof InvalidRoleException) {
```

「java.lang.Exception」を継承している例外がスローされた場合は、「UndeclaredThrowableException」にラップされるため、原因となる例外の中身を取り出します。

```
            mav.setViewName("error/invalidRole");

        } else if(exception instanceof TokenException) {
            mav.setViewName("error/token");

        } else {
            mav.setViewName("error/error");
        }

        // メッセージの設定
        mav.addObject("exceptionType", ex.getClass().getName());

        // ログ出力
        logger.error("エラーです。", exception);

        return mav;
    }

}
```

【servlet-context.xml の編集】

- Spring Bean として登録します。
  その際の Bean 名は、必ず「exceptionResolver」と設定します。

```
<beans>
・・・省略
    <bean id="exceptionResolver" class="sample.web.SystemExceptionResolver"/>
・・・省略
</beans>
```

11.3. web.xml で例外時の遷移先を定義する

- Spring MVC の例外処理でキャッチできない致命的な例外の場合、Tomcat などの APP サーバのスタックトレースが表示され、それらの情報を利用してセキュリティ攻撃を受ける場合があります。
- セキュリティを考慮し、Spring MVC と web.xml での例外処理の 2 段構えで設定することをお勧めします。

```
<web-app>
・・・省略
    <error-page>
        <error-code>404</error-code>
        <location>/WEB-INF/view/error/error.jsp</location>
    </error-page>
    <error-page>
        <exception-type>java.lang.Exception</exception-type>
        <location>/WEB-INF/view/error/error.jsp</location>
    </error-page>
・・・省略
</web-app>
```

HTTP のエラーコードを指定し、遷移先の JSP を指定します。

例外クラスを指定して、遷移先の JSP を指定します。

11.4. JSP で例外が起きた場合の遷移先の指定

- 参考 URL「http://struts.wasureppoi.com/jsp/05_errorPage.html」

【例外発生元のページの設定】

- ページディレクティブの属性「errorPage="エラーページのパス"」を指定します。

```
<%@ page language="java" contentType="text/html;charset=UTF-8" errorPage="/error/error.jsp"%>
<html>
<body>

<%
String strMsg = null;
String msg = strMsg.toString();
%>

</body>
</html>
```

【遷移先のエラー用ページの設定（/error/error.jsp）】

- ページディレクティブの属性「isErrorPage="true"」を指定します。

```
<%@ page language="java" contentType="text/html;charset=UTF-8" isErrorPage="true"%>
<html>
<body>

<%
//　暗黙オブジェクト exception からエラーメッセージを取得して出力
String str = exception.toString();
%>

エラーメッセージ：<%=str%>

</body>
</html>
```

12. カスタムタグ

Spring MVC のカスタムタグは、2 種類しか存在しません。プロパティファイルからメッセージを取得したりする<spring:XXX>の書式のものと、form のフィールド用のタグ<form:XXX>です。If-else などの制御を行いたい場合は、JSTL などを使用します。

12.1. Spring MVC 用のカスタムタグ 1（<spring:XXX>）

12.1.1. はじめに

- JSP の先頭で、カスタムタグのディレクティブ「taglib」定義を行います。

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%-- ▼ taglib --%>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%-- ▲ taglib --%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
・・・省略
</html>
```

表 12.1　カスタムタグの一覧（<spring:XXX>）

No.	タグ名	概要	参照先
1	<spring:bind>	Form の各フィールドと、Controller に渡す Command の項目をバインド設定するために使用します。 通常は、<form:XXX>のカスタムタグを使用します	12.1.2
2	<spring:escapeBody>	タグで囲んだ中身を HTML エスケープし出力します。	12.1.3
3	<spring:hasBindErrors>	Command に対してバインドエラーがある場合、タグのボディを評価（実行）します。	12.1.4
4	<spring:htmlEscape>	Spring MVC 他のカスタムタグの属性「htmlEscape」のページ内でのデフォルト値を設定します。	12.1.5
5	<spring:message>	Spring Bean の「messageSource」からメッセージを取得します。	12.1.6
6	<spring:nestedPath>	Form と Command をバインドする際の現在のパスの位置を変更します。「7.2.9」にある Errors#putNestedPath("パス名")と同じです。	12.1.7
7	<spring:theme>	テーマごとの設定値を出力します。	12.1.8
8	<spring:transform>	指定した値の型を文字列に変換します。	12.1.9
9	<spring:url>	指定した URL を、アプリケーションの URL に変換します。	12.1.10
10	<spring:eval>	SpEL（「6.5」参照）の式を評価します。	12.1.11

12.1.2. カスタムタグ<spring:bind>

Form の各フィールドと Controller に渡す、Command の項目をバインドし関連付けるために使用します。通常は、<form:XXX>のカスタムタグを使用しますが、<spring:bind>は素の HTML のタグと Command を関連付けることができます。

【タグの仕様】

表 12.2　<spring:bind>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	htmlEscape		○		“true”の場合 HTML エスケープを行います。 <spring:escapeHtml>の値が初期値となります。
2	ignoreNestedPath		○	false	ネストした(階層を持つ)オブジェクトに対するパスを無視するかどうか。
3	path	○	○		Command に対するフィールドのパスを指定します。

【タグの使用例】

- 「login」という Command のフィールド（=プロパティ）の 1 つである“account”に対してバインドの設定を行います。
  - Controller 側で、データを受信する際に、「@ModelAttribute」で指定した Command のプロパティ“account”に関連付けられます。
- 階層が深い場合は、ドット「.」でプロパティ名を繋げて記述します。
  - リスト、マップ、配列の場合、“プロパティ名[インデックス]”の形式で、インデックスやマップキーを指定します。

```
<spring:bind path="login.account">
    <input name="account" value="<c:out value="${status.value}"/>"/>
</spring:bind>

<!—ネストしたパスの場合 -->
<spring:bind path="login.books[0].title">
    <input name="books[0].title" value="<c:out value="${status.value}"/>"/>
</spring:bind>
```

12.1.3. カスタムタグ<spring:escapeBody>

タグで囲んだ中身を HTML エスケープし出力します。

【タグの仕様】

表 12.3 <spring:escapeBody>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	htmlEscape		○		“true”の場合 HTML エスケープを行います。 <spring:escapeHtml>の値が初期値となります。
2	javaScriptEscape		○	false	JavaScript のエスケープを行います。

【タグの使用例】

- JSP に直接文字を記述する際に使用します。
- プロパティファイルや、Model から取得する際には、それぞれ、<spring:message htmlEscape="true">や<c:out escapeXml="true" var="${name}"/>などを利用しエスケープするため、使いどころはあまりないかもしれません。

```
<spring:escapeBody>こんにちは<。/springEscapeBody>
<spring:escapeBody javaScriptEscape="true"><script>alert("ERROR");</script></springEscapeBody>
```

12.1.4. カスタムタグ<spring:hasBindErrors>

Command に対してエラーがある場合、タグのボディを評価（実行）します。Validator などでチェックしエラーがある場合評価します。

【タグの仕様】

表 12.4　<spring:hasBindErrors>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	htmlEscape		○		“true”の場合 HTML エスケープを行います。 <spring:escapeHtml>の値が初期値となります。
2	name	○	○		エラーが存在する Command の名前。

【タグの使用例】

- 指定した Comamnd の名前に対して、Errors オブジェクトがあるか判定します。
- エラーがある場合は、${errors.globalErrors}などでエラー内容を取得します。
- 詳細は、「7.1　Errors クラスを使用した入力値検証」を参照してください。

```
<spring:hasBindErrors name="sampleCommand">
<%-- グローバルエラーメッセージの出力 --%>
<c:if test="${errors.globalErrorCount > 0}">
<div class="MessageBox erorr">
    <h4>グローバルエラー</h4>
    <ul>
        <c:forEach items="${errors.globalErrors}" var="error">
        <li><span class="error"><spring:message message="${error}"/></span></li>
        </c:forEach>
    </ul>
</div>
</c:if>

<%-- フィールドエラーメッセージの出力 --%>
<c:if test="${errors.fieldErrorCount > 0}">
<div class="MessageBox erorr">
    <h4>フィールドエラー</h4>
    <ul>
        <c:forEach items="${errors.fieldErrors}" var="error">
        <li><span class="error"><spring:message message="${error}"/></span></li>
        </c:forEach>
    </ul>
</div>
</c:if>
</spring:hasBindErrors>
```

12.1.5. カスタムタグ<spring:htmlEscape>

Spring MVC 他のカスタムタグの属性「htmlEscape」のページ内でのデフォルト値を設定します。システム全体の設定は、web.xmlで設定することができます。

【タグの仕様】

表 12.5　<spring:htmlEscape>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	defaultHtmlEscape	○	○		Spring MVC 他のカスタムタグの属「htmlEscape」のページ内でのデフォルト値を設定します。

【タグの使用例】

```
<spring:htmlEscape defaultHtmlEscape="true" />

<!-- デフォルト値を使用する場合 -->
<spring:message code="label.copryright" />

<!-- 設定を上書きする場合 -->
<spring:message code="label.copryright" htmlEscape="false" />
```

【web.xml の設定】

- システム全体に対して設定する場合、web.xml で設定を行います。

```
<web-app>
・・・省略
    <context-param>
        <description>SpringMVC のカスタムタグの HTML エスケープの初期値設定</description>
        <param-name>defaultHtmlEscape</param-name>
        <param-value>true</param-value>
    </context-param>
・・・省略
</web-app>
```

12.1.6. カスタムタグ<spring:message>

Spring Bean の「messageSource」からメッセージを取得します。また、Validator で作成した Error オブジェクトのメッセージを処理します。

【タグの仕様】

表 12.6　<spring:message>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	arguments		○		引数の値を指定します。 複数指定する場合は、「,」で区切ります。 argumentSeprator で区切り文字は変更できます。
2	argumentSeparator		○	, (カンマ)	属性 arguments の区切り文字を指定します。
3	code		○		プロパティに定義したキーを指定します。
4	htmlEscape		○		Spring MVC 他のカスタムタグの属性「htmlEscape」のページ内でのデフォルト値を設定します。「true」「false」で指定します。
5	javaScriptEscape		○	false	JavaScript のエスケープいます。
6	message		○		MessageSourceResolvable を実装しているオブジェクトを指定すると、自動的に引数などを解釈し出力します。 Errors オブジェクトの各メッセージが該当します。
7	scope		○		属性 var を指定した際に、その変数のセッションスコープを指定します。 「page|request|session|application」の何れか。
8	text		○	null	属性 code で指定した名前のプロパティが存在しない場合に出力するデフォルト値を指定します。
9	var		○		セッションに登録されているメッセージの名前を指定して呼び出します。

【タグの使用例】

```
<%-- 引数なし --%>
<spring:message code="label.item" />

<%-- 引数あり(１つ) --%>
<spring:message code="label.item1" arguments="1" />

<%-- 引数あり(複数つ) --%>
<spring:message code="label.item2" arguments="1,引数 2" />
```

```
<%-- メッセージを直接指定(引数あり) --%>
<spring:message text="テキストメッセージ。引数={0}、引数={1}" arguments="arg1,arg2">

<%-- 入力値検証結果のメッセージの表示（グローバルエラー） --%>
<ul>
    <c:forEach items="${errors.globalErrors}" var="error">
    <li><span class="error"><spring:message message="${error}"/></span></li>
    </c:forEach>
</ul>
```

<spring:message code="${error.code}" arguments="${error.arguments}"/>
を、属性「message」のみで出力することができる。
「ObjectError」は、MessageSourceResolvable インタフェースを実装しているため。

【プロパティファイル（messageSource のプロパティファイル）の記述例】

```
label.item=引数なしのメッセージ
label.item1=引数有りのメッセージ 1。引数 1={0}
label.item2=引数有のメッセージ 2。引数 1={0}、引数 1={1}
```

12.1.7. カスタムタグ<spring:nestedPath>

Form と Command をバインドする際の現在のパスの位置を変更します。「6.2.8　Validator による階層を持つ Command の入力値検証」にある「Errors#putNestedPath("パス名")」と同じ機能を持ちます。カスタムタグ<spring:bind>と併用して使用します。

【タグの仕様】

表 12.7　<spring:nestedPath>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		ネストするパスの指定をします。 深い階層の場合は、ピリオド「.」で区切り指定します。

【タグの使用例】

- ネストするパスを指定することで、<spring:bind path=”パス”>の記述において、“login”が省略できます。

```
<!-- ネストを指定しない場合 -->
<spring:bind path="login.account">
    <input name="account" value="<c:out value="${status.value}"/>"/>
</spring:bind>

<!-- ネストを指定する場合（"login"でネスト） -->
<spring:nestedPath path="login"/>
<spring:bind path="account">
    <input name="account" value="<c:out value="${status.value}"/>"/>
</spring:bind>
```

12.1.8. カスタムタグ<spring:theme>

テーマを切り替えた際などに、それぞれのテーマの設定値を出力します。テーマを切り替える方法は、「10.3 テーマの設定」を参照してください。

【タグの仕様】

表 12.8 <spring:theme>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	arguments		○		引数の値を指定します。 複数指定する場合は、「,」で区切ります。 argumentSeprator で区切り文字は変更できます。
2	argumentSeparator		○	, (カンマ)	属性 arguments の区切り文字を指定します。
3	code		○		ThemeSource で指定したプロパティファイル中のキー名を指定します。
4	htmlEscape		○		Spring MVC 他のカスタムタグの属性「htmlEscape」のページ内でのデフォルト値を設定します。「true」「false」で指定します。
5	javaScriptEscape		○	false	JavaScript のエスケープいます。
6	message		○		MessageSourceResolvable を実装しているオブジェクトを指定すると、自動的に引数などを解釈し出力します。 Errors オブジェクトの各メッセージが該当します。
7	scope		○		属性 var を指定した際に、その変数のセッションスコープを指定します。 「page｜request｜session｜application」の何れか。
8	text		○	null	属性 code で指定した名前のプロパティが存在しない場合に出力するデフォルト値を指定します。
9	var		○		セッションに登録されているメッセージの名前を指定して呼び出します。

【タグの使用例】

- <spring:theme>は、<spring:message>のクラスを継承し作成されてりるため、同じ属性を指定できますが、実際には属性「code」を使用する下記のケースしか使用する場面はないと思います。

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
```

```
    <head>
        <link rel="stylesheet" href="<spring:theme code='styleSheet'/>" type="text/css"/>
    </head>
    <body style="background=<spring:theme code='background'/>">
        ...
    </body>
</html>
```

12.1.9. カスタムタグ<spring:transform>

指定した値の型を文字列に変換します。

【タグの仕様】

表 12.9　<spring:transform>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	htmlEscape		○		Spring MVC 他のカスタムタグの属性「htmlEscape」のページ内でのデフォルト値を設定します。「true」「false」で指定します。
2	scope		○	page	属性 var を指定した際に、その変数の保存先のセッションスコープを指定します。「page｜request｜session｜application」の何れか。
3	value	○	○		文字列に変換する値。
4	var		○		変換した値を保存する変数名。

【タグの使用例】

- 属性 var を指定しない場合は、文字列に保存した値を保存しないでそのまま出力します。

```
<%-- 変数${type}を文字列型に変換し出力します。 --%>
<spring:transform value="${type}" />

<%-- 変数${type}を文字列型に変換し、変数「typeString」に保存します。 --%>
<spring:transform value="${type}" var="typeString"/>
```

12.1.10. カスタムタグ<spring:url>

指定した URL をアプリケーション用の URL に変換します。また、パラメータを動的に組み立てることもできます。

【タグの仕様】

表 12.10　<spring:url>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	value	○	○		変換、または組み立てる URL です。
2	context		○		アプリケーションのコンテキスト名を指定します。 通常は、現在動作させているアプリケーションの名称になります。
3	var		○		URL の形式に変換した値の保存先の変数名。
4	scope		○	page	属性 var を指定した際に、その変数の保存先のセッションスコープを指定します。 「page｜request｜session｜application」 の何れか
5	htmlEncoding		○		Spring MVC 他のカスタムタグの属性「htmlEscape」のページ内でのデフォルト値を設定します。「true」「false」で指定します。
6	javascriptEncoding		○	false	JavaScript エスケープを行います。

【タグの使用例】

- クエストリングでパラメータを指定したい場合は、タグ<spring:param name="パラメータ名" value="値">を、<spring:url>の中にネストします。
- セッションを使用している場合、URL に jssessionId が付加される場合があります。

```
<ol>
    <li>URL1：<spring:url value="/dir1/sample.html"/></li>

    <li>URL2(context 指定あり)：<spring:url value="/dir1/sample.html" context="sample"/></li>

    <li>URL3（パラメータ指定あり）：
        <spring:url value="http://example.com/index.html">
            <spring:param name="name">arg1</spring:param>
            <spring:param name="code" value="arg2"/>
        </spring:url>
    </li>

    <li>URL4(変数に保存する)：
        <spring:url value="/dir2/index.html" var="url4" scope="page"/>
        ${url4}
    </li>
</ol>
```

【生成した HTML】

```
<ol>
    <li>URL1：/spring3-mvc/dir1/sample.html</li>

    <li>URL2(context 指定あり)：/sample/dir1/sample.html</li>

    <li>URL3（パラメータ指定あり）：
        http://example.com/index.html?name=arg1&amp;code=arg2</li>

    <li>URL4(変数に保存する)：
        /spring3-mvc/dir2/index.html</li>
</ol>
```

Servlet のコンテキスト名が自動的に付加されます。

12.1.11. カスタムタグ<spring:eval>

Spring Expression Language(SpEL)の式を評価し、結果を出力します。SpEL の仕様については、「6.5 Spring Expression Language(SpEL)」を参照してください。

【タグの仕様】

表 12.11　<spring:eval>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	expression	○	○		SpEL 式を記述します。
2	var		○		SpEL 式を評価した結果の保存先の変数名。
3	scope		○	page	属性 var を指定した際に、その変数の保存先のセッションスコープを指定します。 「page｜request｜session｜application」の何れか
4	htmlEncoding		○		Spring MVC 他のカスタムタグの属性「htmlEscape」のページ内でのデフォルト値を設定します。「true」「false」で指定します。
5	javascriptEncoding		○	false	JavaScript エスケープを行います。

【タグの使用例】

```
<spring:eval expression="2 div 3.0"/>
<spring:eval expression="T(org.apache.commons.lang.StringUtils).isEmpty(#sample)" var="spel2" scope="session"/>
```

12.2. Spring MVC 用のカスタムタグ 2（<form:XXX>）

Command の各プロパティと HTML の各フィールドをバインドするためのタグです。

- <spring:bind>でもバインドできますが、<form:XXX>のカスタムタグを使用すると簡単に設定できます。
- フィールドごとにエラーメッセージを出力できたり、エラーがある場合などに class 属性を変更できたりでします。

12.2.1. はじめに

- JSP の先頭で、カスタムタグのディレクティブ「taglib」定義を行います。

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%-- ▼ taglib --%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%-- ▲ taglib --%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
・・・省略
</html>
```

表 12.12　カスタムタグの一覧（<spring:XXX>）

No.	タグ名	概要	参照先
1	<form:form>	HTML のタグ<form>に相当し、Command と関連付けます。	12.2.2
2	<form:errors>	フィールドエラーのメッセージを出力します。	12.2.3
3	<form:label>	HTML のタグ<label>に相当し、フィールドごとに設定します。	12.2.4
4	<form:input>	HTML のタグ<input type="text">を出力します。	12.2.5
5	<form:hidden>	HTML のタグ<input type="hidden">を出力します。	12.2.6
6	<form:textarea>	HTML のタグ<textarea>を出力します。	12.2.7
7	<form:password>	HTML のタグ<input type="password">を出力します。	12.2.8
8	<form:checkbox>	HTML のタグ<input type="checkbox">を出力します。	12.2.9
9	<form:checkboxes>	Collection などから、タグ<input type="checkbox">を出力します。	12.2.10
10	<form:radiobutton>	HTML のタグ<input type="radio">を出力します。	12.2.11
11	<form:radiobuttons>	Collection などから、タグ<input type="radio">を出力します。	12.2.12
12	<form:select>	HTML のタグ<select>を出力します。	12.2.13
13	<form:option>	HTML のタグ<option>を出力します。	12.2.13
14	<form:options>	Collection などから、タグ<option">を出力します。	12.2.14

12.2.2. カスタムタグ<form:form>

HTML のタグ<form>に相当し、Command と関連付けます。

【タグの仕様】

表 12.13　<form:form>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	action		○		HTML と同じ。 省略した場合、現在表示されている URL に対してデータを送信します。
2	commandName		○	command	バインドする Command の名称を指定します。 Controller 側で@ModelAttribute で指定した名称を設定します。
3	method		○	POST	HTML と同じ。POST、GET、DELETE など。
4	modelAttribute		○	command	属性「commandName」と同じ。 Controller 側の@ModelAttribute と名称を合わせるために通常はこちらを使用します。
5	cssClass		○		HTML の class 属性を出力します。
6	cssStyle		○		HTML の style 属性を出力します。
7	htmlEscape				各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
8	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass」「cssStyle」を使用します。

【タグの使用例】

- Controller 側で、@ModelAttribute を指定している場合、JSP 側では属性「modelAttribute」を定義します。その際に、名前を必ず一致させるようにします。
- データの送信を指定する属性「action」は HTML のものと同じです。その際に、URL を絶対パスで指定することをお勧めします。
  - URL が階層化されている場合、相対パスだと階層により「../」など使用しないといけなく、バグのもととなるからです。
  - 絶対パスで指定する場合、アプリケーションのコンテキスト名をシステム起動時の初期化時に定義しておくと便利です。詳細は、「14.2　アプリケーションの初期化プログラムの実行」を参照してください。

```
<form:form modelAttribute="sampleCommand" action="${appUrl}/test/validate1.html" method="post">
    <p>
        <form:label path="name">名前</form:label>
        <form:input path="name" cssClass="input_field" cssErrorClass="input_error"/>
        <form:errors path="name" cssClass="errors" />
    </p>
    <p>
        <form:label path="age">年齢</form:label>
        <form:input path="age" cssClass="input_field" cssErrorClass="input_error"/>
        <form:errors path="age" cssClass="errors" />
    </p>
    <input type="submit"/>
</form:form>
```

【生成された HTML】

- JSP で指定した属性「modelAttribute」の値は、HTML では「id」属性となります。

```
<form id="sampleCommand" action="/spring3-mvc/test/validate1.html" method="post">
    <p>
        <label for="name">名前</label>
        <input id="name" name="name" class="input_field" type="text" value=""/>
    </p>
    <p>
        <label for="age">年齢</label>
        <input id="age" name="age" class="input_field" type="text" value="1"/>
    </p>
    <input type="submit"/>
</form
```

12.2.3. カスタムタグ<form:errors>

フィールドエラーのメッセージを出力します。

- エラーメッセージの作成方法などは、「7.2　Validator を実装した入力値検証」を参照してください。

【タグの仕様】

表 12.14　<form:errrors>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssStyle		○		HTML の style 属性を出力します。
5	delimiter		○	 	エラーが複数あった際の区切り文字（タグ）を指定します。
6	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass」「cssStyle」を使用します。

【タグの使用例】

- 属性「path」は、データバインドしているフィールドの「path」と一致させます。

```
<form:input path="name" />
<form:errors path="name" cssClass="errors" />
```

【生成された HTML】

- エラーがない場合は、タグは出力されません。
- エラーメッセージはタグで囲み出力します。
- 複数メッセージがある場合、属性「delimiter」で設定された文字で区切りられます。

```
<!-- エラーがない場合 ->
<input id="name" name="name" type="text" value="abc"/>

<!-- エラーがある場合 -->
<input id="name" name="name" type="text" value="111111"/>
<span id="name.errors" class="errors">文字の長さは 0 から 5 の間で入力してください。<br/>マッチしません。</span>
```

12.2.4. カスタムタグ<form:label>

HTML のタグ<label>に相当し、フィールドごとに設定します。

【タグの仕様】

表 12.15　<form:label >のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- 属性「path」は、データバインドしているフィールドの「path」と一致させます。

```
<form:label path="name" cssClass="field_label" cssErrorClass="field_error_label">名前</form:label>
<form:input path="name" />
```

【生成された HTML】

- 属性「for」を記述していない場合、属性「path」をもとに自動的に付与されます。
- バインドしたフィールドに対してエラーがある場合、属性「cssErrorClass」で指定した値が出力されます。

```
<!-- エラーがない場合 -->
<label for="name" class="field_label">名前</label>
<input id="name" name="name" type="text" value="abc"/>

<!-- エラーがある場合 -->
<label for="name" class="field_error_label">名前</label>
<input id="name" name="name" type="text" value="111111"/>
```

12.2.5. カスタムタグ<form:input>

HTML のタグ<input type="text">を出力します。

- ファイルアップロード用のフィールドの場合、<form:input path="upload" type="file"/>」のように属性「type="file"」とします。詳細は、「4.4　ファイルアップロード」を参照してください。

【タグの仕様】

表 12.16　<form:input>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- 属性「path」にて、バインドする Command のプロパティのパスを指定します。
- 値を設定するための、属性「value」は指定する必要ありません。バインドした Command のプロパティから自動的に出力されます。

```
<form:input path="age" cssClass="input_field" cssErrorClass="input_error"/>
```

【生成された HTML】

- 属性「value」は自動的にバインド先から値が取得されます。
- 属性「id」「name」を記述していない場合、属性「path」をもとに自動的に付与されます。
- バインドしたフィールドに対してエラーがある場合、属性「cssErrorClass」で指定した値が出力されます。

```
<!-- エラーがない場合 -->
<input id="age" name="age" class="input_field " type="text" value="20"/>

<!-- エラーがある場合 -->
<input id="age" name="age" class="input_error" type="text" value="-1"/>
```

12.2.6. カスタムタグ<form:hidden>

HTML のタグ<input type="hidden">を出力します。

【タグの仕様】

表 12.17　<form:hidden>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	id		○		HTML の属性「id」と同じ。

【タグの使用例】

- 属性「path」にて、バインドする Command のプロパティのパスを指定します。

```
<form:hidden path="tokenId"/>
```

【生成された HTML】

- 属性「id」を記述していない場合、属性「path」をもとに自動的に付与されます。

```
<input id="tokenId" name="tokenId" type="hidden" value="1323124241"/>
```

12.2.7. カスタムタグ<form:textarea>

HTML のタグ<textarea>を出力します。

【タグの仕様】

表 12.18　<form:textarea >のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- 属性「path」にて、バインドする Command のプロパティのパスを指定します。

```
<form:textarea path="comment " rows="3" cols="20" cssErrorClass="textarea_error" />
```

【生成された HTML】

- 属性「id」「name」を記述していない場合、属性「path」をもとに自動的に付与されます。
- バインドしたフィールドに対してエラーがある場合、属性「cssErrorClass」で指定した値が出力されます。

```
<!-- エラーがない場合 -->
<textarea id="comment" name="comment" rows="3" cols="20">今日はいい天気です</textarea>

<!-- エラーがある場合 -->
<textarea id="comment" name="comment" rows="3" cols="20" class="textarea_error"></textarea>
```

12.2.8. カスタムタグ<form:password>

HTML のタグ<input type="password">を出力します。

【タグの仕様】

表 12.19　<form:password>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	htmlEscape		○		データバインド時の Command のプロパティのパスを指定します。
2	cssClass		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssErrorClass		○		HTML の class 属性を出力します。
4	cssStyle		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	showPassword		○	false	初期表示やエラー時のレンダリング時に、パスワードの値を保持しておくか設定します。 「false」の場合、エラー時の場合など、毎回、空文字にリセットされます。
6	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- 属性「path」にて、バインドする Command のプロパティのパスを指定します。
- 属性「showPassword="true"」を設定すると、エラー時などに自画面遷移し戻ってきた場合も、パスワードはリセットされません。

```
<!-- 通常の場合 -->
<form:password path="password" cssErrorClass="field_error"/>

<!-- 属性「showPassword="true"」 -->
<form:password path="password" showPassword="true" cssErrorClass="field_error"/>
```

【生成された HTML】

- 属性「id」「name」を記述していない場合、属性「path」をもとに自動的に付与されます。

- バインドしたフィールドに対してエラーがある場合、属性「cssErrorClass」で指定した値が出力されます。
- 属性「showPassword="true"」を設定している場合、パスワードの初期値がリセットされずに残っています。

```
<!-- エラーがない場合 -->
<input id="password" name="password" type="password" value=""/>

<!-- エラーがある場合：通常の場合 -->
<input id="password" name="password" type="password" cssErrorClass="field_error" value=""/>

<!-- エラーがある場合：属性「showPassword="true"」 -->
<input id="password" name="password" type="password" cssErrorClass="field_error" value="bbbb"/>
```

12.2.9. カスタムタグ<form:checkbox>

HTML のタグ<input type="checkbox">を出力します。

【タグの仕様】

表 12.20　<form:checkbox>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	label		○		チェックボックスの名前を指定します。 タグ<input type=”checkbox">の直後に、タグ<label>で囲まれ出力される文字列です。
7	value		○		チェックボックスの値を指定します。 Boolean 型で送受信する場合、指定する必要はありません。
8	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- ON/OFF の情報しか必要ないような、選択項目が 1 つしかない場合、value を省略して boolean 型にバインドします。
  - チェックボックスの場合、生成された HTML の属性「id」は「“path” ＋ “連番”」となるため、<form:label>を指定する場合は属性「id」を直接記述します。
- 複数の項目を選択させたい場合、文字列のリスト(List<String>)にバインドします。
  その場合、属性「path」の値は同じに設定し、属性「label」「value」はそれぞれ固有の値に設定します。
- 属性「label」を指定すると、タグ<label>が自動的に出力されます。

```
<!-- boolean 型にバインドする場合 -->
<p>
    <form:label path="confirmed">確認メールを送信する</form:label>
    <form:checkbox path="confirmed" id="confirmed"/>
    <form:errors path="confirmed" cssClass="errors" />
</p>

<!-- リスト型の文字列にバインドする場合 -->
<p>
    <form:checkbox path="favoriteSubject" label="国語" value="japanese" />
    <form:checkbox path="favoriteSubject" label="数学" value="math" />
    <form:checkbox path="favoriteSubject" label="歴史" value="history" />
    <form:checkbox path="favoriteSubject" label="理科" value="science" />
    <form:errors path="favoriteSubject" cssClass="errors" />
</p>
```

属性「id」を固定にするために、直接記述します。

リストにバインドする場合は、属性「path」は同じ値を設定します。

【Command の作成】

```
public class SampleCommand implements Serializable {

    /** serialVersionUID */
    private static final long serialVersionUID = 1L;

    private Boolean confirmed;

    private List<String> favoriteSubject;

    public SampleCommand() {
        favoriteSubject = ListUtils.lazyList(
                new ArrayList<String>(),
                FactoryUtils.instantiateFactory(String.class));
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this).toString();
    }

    // getter、setter は省略
}
```

項目が 1 つしかなく、ON/OFF の判定のみしたい場合は、Boolean 型にします。

複数項目がある場合、文字列のリスト型に指定します。
データを受信した場合、属性「value」の値が入ります。

【生成された HTML（初期表示：何も選択していない場合）】

- 属性「id」「name」を記述していない場合、属性「path」をもとに自動的に付与されます。
- JSP の属性「value」を指定しない場合、HTML の属性「value="true"」となります。
- JSP の属性「label」を指定しておくと、タグ<label>が自動的に生成されます。

```
<!-- boolean 型にバインドする場合 -->
<p>
    <label for="confirmed">確認メールを送信する</label>
    <input id="confirmed" name="confirmed" type="checkbox" value="true"/>
    <input type="hidden" name="_confirmed" value="on"/>
</p>

<!-- リスト型の文字列にバインドする場合 -->
<p>
    <input id="favoriteSubject1" name="favoriteSubject" type="checkbox" value="japanese"/>
    <label for="favoriteSubject1">国語</label>
    <input type="hidden" name="_favoriteSubject" value="on"/>

    <input id="favoriteSubject2" name="favoriteSubject" type="checkbox" value="math"/>
    <label for="favoriteSubject2">数学</label>
    <input type="hidden" name="_favoriteSubject" value="on"/>

    <input id="favoriteSubject3" name="favoriteSubject" type="checkbox" value="history"/>
    <label for="favoriteSubject3">歴史</label>
    <input type="hidden" name="_favoriteSubject" value="on"/>

    <input id="favoriteSubject4" name="favoriteSubject" type="checkbox" value="science"/>
    <label for="favoriteSubject4">理科</label>
    <input type="hidden" name="_favoriteSubject" value="on"/>
</p>
```

JSP のカスタムタグで、属性「value」を省略した場合。

バインドするための hidden が自動生成されます。

属性「id」は、「“path” + “連番”」となります。

JSP のカスタムタグで、属性「label」を設定したした場合、自動的に生成されます。

【Controller（初期値を設定したい場合）】

```
@Controller
@RequestMapping("/test/form2")
public class Form2Controller {

    // command の初期オブジェクトの取得
    @ModelAttribute("sampleCommand")
    public SampleCommand createInitCommand() {
        SampleCommand command = new SampleCommand();
        return command;
    }

    // 初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {
        SampleCommand command = createInitCommand();

        // boolean 型のチェックボックスの初期値
        command.setConfirmed(Boolean.TRUE);

        // リスト型のチェックボックスの初期値
        command.getFavoriteSubject().add("science");
        command.getFavoriteSubject().add("math");

        model.addAttribute("sampleCommand", command);
```

Boolean 型の場合のチェックボックスの初期値(ON/OFF 状態)を設定する。

リスト型のチェックボックスの初期値は、属性「value」の値を指定します。
値は**順不同に設定**しても問題ありません。

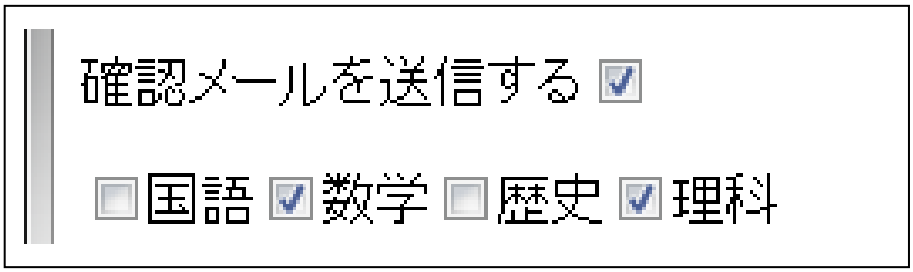
```
    }

    // post で送られた場合
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(@ModelAttribute("sampleCommand") SampleCommand command,
            BindingResult bindingResult) {
        // 省略
    }
}
```

【生成された HTML（再描画時：選択済みの状態）】

- 属性「checked="checked"」が自動的に付与され選択状態となります。

```
<!-- boolean 型にバインドする場合 -->
<p>
    <label for="confirmed">確認メールを送信する</label>
    <input id="confirmed" name="confirmed" type="checkbox" value="true" checked="checked"/>
    <input type="hidden" name="_confirmed" value="on"/>
</p>

<!-- リスト型の文字列にバインドする場合 -->
<p>
    <input id="favoriteSubject1" name="favoriteSubject" type="checkbox" value="japanese"/>
    <label for="favoriteSubject1">国語</label>
    <input type="hidden" name="_favoriteSubject" value="on"/>

    <input id="favoriteSubject2" name="favoriteSubject" type="checkbox" value="math" checked="checked"/>
    <label for="favoriteSubject2">数学</label>
    <input type="hidden" name="_favoriteSubject" value="on"/>

    <input id="favoriteSubject3" name="favoriteSubject" type="checkbox" value="history"/>
    <label for="favoriteSubject3">歴史</label>
    <input type="hidden" name="_favoriteSubject" value="on"/>

    <input id="favoriteSubject4" name="favoriteSubject" type="checkbox" value="science" checked="checked"/>
    <label for="favoriteSubject4">理科</label>
    <input type="hidden" name="_favoriteSubject" value="on"/>
</p>
```

【ブラウザでの表示】



12.2.10. カスタムタグ<form:checkboxes>

Collection、Map、配列から、タグ<input type="checkbox">を出力します。

【タグの仕様】

表 12.21 <form:checkbox>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	items	○	○		各項目のリスト（Collection、Map、配列）のオブジェクトが格納されている名前を指定します。
7	itemLabel		○		各項目のチェックボックスの名前を指定します。 属性「items」のリストの要素の「プロパティの名前」を指定します。 <input type="checkbox">タグの直後に、タグ<label>で囲まれ出力される文字列です。
8	itemValue		○		各項目のチェックボックスの値を指定します。 属性「items」のリストの要素の「プロパティの名前」を指定します。
9	delimiter		○		項目が複数あった場合、<input type="checkbox">のタグ間の区切り文字を指定します。例えば、 。
10	element		○	span	項目が複数あった場合、タグ<input>、<label>を囲むタグです、 例) <input…><label>～</label>
11	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- 各項目を Collection などから取得するために、属性「items」で指定します。
  Controller 側で、事前に Model やセッションスコープに登録しておく必要があります。
- Collection の要素が JavaBean の場合、JavaBean のプロパティの名前を指定します。

```
<!-- リスト型の文字列にバインドする場合(データを文字列のリストから取得) -->
<p>
    <form:checkboxes path="favoriteSubject" items="${favotiteSubjectList}" />
    <form:errors path="favoriteSubject" cssClass="errors" />
</p>

<!-- リスト型の文字列にバインドする場合(データを JavaBean のリストから取得) -->
<p>
    <form:checkboxes path="favoriteSubject"
 items="${favotiteSubjectList2}" itemValue="name" itemLabel="localeName"/>
    <form:errors path="favoriteSubject" cssClass="errors" />
</p>
```

Model などに登録しているデータを、EL 式で指定します。

JavaBean を要素に持つ場合、ラベルや値をどの値とするか、プロパティ名で指定します。

【Command の作成】

```
public class SampleCommand implements Serializable {

    private List<String> favoriteSubject;

    public SampleCommand() {
        favoriteSubject = ListUtils.lazyList(
                new ArrayList<String>(),
                FactoryUtils.instantiateFactory(String.class));
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this).toString();
    }

    // getter、setter は省略
}
```

【Controller の作成】

- チェックボックスの選択候補を、List 型で作成し Model などに登録します。
  - Model などの request スコープに登録している場合、エラーがあった場合の再描画時に再設定する必要があり、設定し忘れなどのバグの元にもなります。
  - 選択候補の値がシステムで普遍な場合は、システム起動時に application スコープに登録しておくことをお勧めします。
    システム起動時に設定する方法は、「14.2　アプリケーションの初期化プログラムの実行」を参照のこと。

```
@Controller
@RequestMapping("/test/form2")
public class Form2Controller {
```

```
    // command の初期オブジェクトの取得
    @ModelAttribute("sampleCommand")
    public SampleCommand createInitCommand() {
        SampleCommand command = new SampleCommand();
        return command;
    }

    // 初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {
        SampleCommand command = createInitCommand();

        // リスト型のチェックボックスの初期値
        command.getFavoriteSubject().add("science");
        command.getFavoriteSubject().add("math");
        model.addAttribute("sampleCommand", command);

        // チェックボックスのリストの選択候補の値
        model.addAttribute("favotiteSubjectList", getFavotiteSubjectData());
        model.addAttribute("favotiteSubjectList2", getFavotiteSubjectDataFromDb());
    }

    // チェックボックスの選択候補のデータの取得（文字列のリスト）
    private List<String> getFavotiteSubjectData() {

        List<String> list = new ArrayList<String>();
        list.add("国語");
        list.add("数学");
        list.add("歴史");
        list.add("理科");
        return list;
    }

    // チェックボックスの選択候補のデータの取得（JavaBean のリスト）
    private List<FavoriteSubjectDto> getFavotiteSubjectDataFromDb() {
        List<FavoriteSubjectDto> list = /** DB などから取得 */;
        return list;
    }

    // post で送られた場合
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(@ModelAttribute("sampleCommand") @Valid SampleCommand command,
            BindingResult bindingResult) {

        // エラーがある場合
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            // チェックボックスの選択候補データの再設定
            mav.addObject("favotiteSubjectList", getFavotiteSubjectData());
            mav.addObject("favotiteSubjectList2", getFavotiteSubjectDataFromDb());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

チェックボックスの選択候補の値を、Model などに登録しておきます。

Model(Request)に選択候補の値を設定している場合、再描画の際に、再設定する必要があります。

【DTO(JavaBean)の定義】

```
public class FavoriteSubjectDto {

    public FavoriteSubjectDto() {

    }

    private String name;

    private String localeName;

    // getter、setter は省略
}
```

JSP のカスタムタグの属性「itemLabel」「itemValue」でプロパティの名前ををを指定します。

【生成された HTML】

- 生成された HTML を見るとわかりますが、文字列のリストから生成した場合、属性「value」が文字列の要素値のそのままになります。
  - ➢ 国際化などの際に、言語ごとにラベルが変わる場合、値も変わってしまうため、JavaBean から取得、または Map 型から取得することをお勧めします。
    また、初期値を設定する市にも、ラベルの値をそのまま設定するためデータ処理するには不都合になります。
  - ➢ Map 型からデータを取得する場合、「Map のキー⇒value 属性」「マップの値⇒ラベル」として設定されます。順番を一定にするため、LinkedHashMap などに値を格納することをお勧めします

```
<!-- リスト型の文字列にバインドする場合(データを文字列のリストから取得) -->
<p>
    <span>
        <input id="favoriteSubject5" name="favoriteSubject" type="checkbox" value="国語"/>
        <label for="favoriteSubject5">国語</label>
    </span>
    <span>
        <input id="favoriteSubject6" name="favoriteSubject" type="checkbox" value="数学"/>
        <label for="favoriteSubject6">数学</label>
    </span>
    <span>
        <input id="favoriteSubject7" name="favoriteSubject" type="checkbox" value="歴史"/>
        <label for="favoriteSubject7">歴史</label>
    </span>
    <span>
        <input id="favoriteSubject8" name="favoriteSubject" type="checkbox" value="理科"/>
        <label for="favoriteSubject8">理科</label>
    </span>
    <input type="hidden" name="_favoriteSubject" value="on"/>
</p>

<!-- リスト型の文字列にバインドする場合(データを JavaBean のリストから取得) -->
<p>
    <span>
        <input id="favoriteSubject9" name="favoriteSubject" type="checkbox" value="japanese"/>
        <label for="favoriteSubject9">国語</label>
    </span>
    <span>
        <input id="favoriteSubject10" name="favoriteSubject" type="checkbox" value="math" checked="checked"/>
```

文字列型のリストから生成した場合、属性「value」の値もリストの要素の値となります。

JavaBean のリストから生成した場合。

```
        <label for="favoriteSubject10">数学</label>
    </span>
    <span>
        <input id="favoriteSubject11" name="favoriteSubject" type="checkbox" value="history"/>
        <label for="favoriteSubject11">歴史</label>
    </span>
    <span>
        <input id="favoriteSubject12" name="favoriteSubject" type="checkbox" value="science"
checked="checked"/>
        <label for="favoriteSubject12">理科</label>
    </span>
    <input type="hidden" name="_favoriteSubject" value="on"/>
</p>
```

12.2.11. カスタムタグ<form:radiobutton>

HTML のタグ<input type="radio">を出力します。

【タグの仕様】

表 12.22 <form:radiobutton>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	label		○		ラジオボタンの名前を指定します。 タグ<input type="radio">の直後に、タグ<label>で囲まれ出力される文字列です。
7	value		○		ラジオボタンの値を指定します。
8	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- ラジオボタンは数者択一なので、１つのグループに対して、属性「path」を同じ値にします。
- Boolean 型にバインドする場合は、属性「value」の値は「true」「false」の何れかにします。
- ラジオボタンの場合、生成された HMLT の属性「id」は、「“path” + “連番”」となるため、<form:label>を指定する場合は、属性「id」を直接記述します。
  - 属性「label」を指定すると、自動的にタグ<label>が生成されるため、通常は属性「label」を指定します。

```
<!-- ラジオボタン：boolean 型にバインドする -->
<div>
    <span>実行しますか？</span>
    <ul>
        <li>
            <form:radiobutton path="executed" value="true" id="executedTrue"/>
            <form:label path="executed" for="executedTrue">はい</form:label>
        </li>
        <li>
            <form:radiobutton path="executed" value="false" label="いいえ"/>
        </li>
    </ul>
</div>

<!-- ラジオボタン：文字列型にバインドする -->
<div>
    <span>この商品に満足しましたか？</span>
    <ul>
        <li><form:radiobutton path="answer" value="1" label="満足しました"/></li>
        <li><form:radiobutton path="answer" value="2" label="普通です"/></li>
        <li><form:radiobutton path="answer" value="3" label="不満な点があります"/></li>
    </ul>
</div>
```

Boolean 型にバインドする場合、値を「true」「false」の何れかにする。

属性「id」を固定にするために、直接記述します。

【Command の作成】

```
public class SampleCommand implements Serializable {

    // ラジオボタン（boolean 型にバインドする）
    private Boolean executed;

    // ラジオボタン（文字列型にバインドする）
    private String answer;

    public SampleCommand() {}

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this).toString();
    }

    // getter、setter は省略
}
```

属性「value」の値が全て、数字の場合の場合、「Integer」型にバインドすることもできます。

【生成された HTML（初期表示：何も選択していない場合）】

- 属性「id」「name」を記述していない場合、属性「path」をもとに自動的に付与されます。
  - 自動付与される場合、属性「id」は「“path”＋“連番”」になります。
- JSP の属性「label」を指定しておくと、タグ<label>が自動的に生成されます。

```
<!-- ラジオボタン：boolean 型にバインドする -->
<div>
    <span>実行しますか？</span>
    <ul>
        <li>
            <input id="executedTrue" name="executed" type="radio" value="true"/>
```

```
            <label for="executedTrue">はい</label>
        </li>
        <li>
            <input id="executed1" name="executed" type="radio" value="false"/>
            <label for="executed1">いいえ</label>
        </li>
    </ul>
</div>

<!-- ラジオボタン：文字列型にバインドする -->
<div>
    <span>この商品に満足しましたか？</span>
    <ul>
        <li>
            <input id="answer1" name="answer" type="radio" value="1"/>
            <label for="answer1">満足しました</label>
        </li>
        <li>
            <input id="answer2" name="answer" type="radio" value="2"/>
            <label for="answer2">普通です</label>
        </li>
        <li>
            <input id="answer3" name="answer" type="radio" value="3"/>
            <label for="answer3">不満な点があります</label>
        </li>
    </ul>
</div>
```

JSP のカスタムタグで、属性「label」を設定した場合、自動的に生成されます。

【Controller（初期値を設定したい場合）】

```
@Controller
@RequestMapping("/test/form2")
public class Form2Controller {

    // command の初期オブジェクトの取得
    @ModelAttribute("sampleCommand")
    public SampleCommand createInitCommand() {
        SampleCommand command = new SampleCommand();

        return command;
    }

    // 初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        SampleCommand command = createInitCommand();

        // ラジオボタンの初期値（boolean 型）
        command.setExecuted(Boolean.TRUE);

        // ラジオボタンの初期値（文字列型）
        command.setAnswer("2");

        model.addAttribute("sampleCommand", command);
    }

    // post で送られた場合
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(@ModelAttribute("sampleCommand") SampleCommand command,
```

Command のプロパティに対して、初期値を設定します。

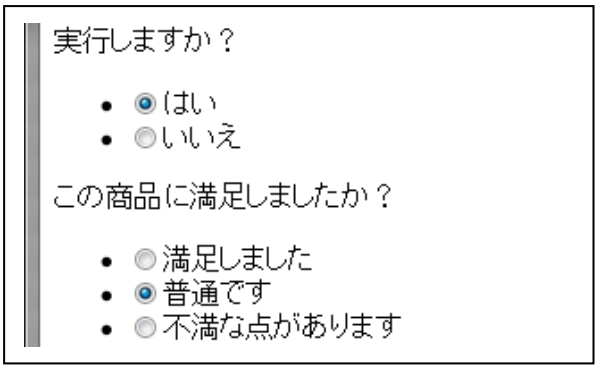
```
            BindingResult bindingResult) {

        // 省略
    }
}
```

【生成された HTML（再描画時・初期値指定時：選択済みの状態）】

- 属性「checked="checked"」が自動的に付与され、選択状態となります。

```
<!-- ラジオボタン：boolean 型にバインドする -->
<div>
    <span>実行しますか？</span>
    <ul>
        <li>
            <input id="executedTrue" name="executed" type="radio" value="true" checked="checked"/>
            <label for="executedTrue">はい</label>
        </li>
        <li>
            <input id="executed1" name="executed" type="radio" value="false"/>
            <label for="executed1">いいえ</label>
        </li>
    </ul>
</div>

<!-- ラジオボタン：文字列型にバインドする -->
<div>
    <span>この商品に満足しましたか？</span>
    <ul>
        <li>
            <input id="answer1" name="answer" type="radio" value="1"/>
            <label for="answer1">満足しました</label>
        </li>
        <li>
            <input id="answer2" name="answer" type="radio" value="2" checked="checked"/>
            <label for="answer2">普通です</label>
        </li>
        <li>
            <input id="answer3" name="answer" type="radio" value="3"/>
            <label for="answer3">不満な点があります</label>
        </li>
    </ul>
</div>
```

【ブラウザでの表示】



12.2.12. カスタムタグ<form:radiobuttons>

Collection、Map、配列から、タグ<input type="radio">を出力します。

【タグの仕様】

表 12.23 <form:radiobuttons>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	items	○	○		各項目のリスト（Collection、Map、配列）のオブジェクトが格納されている名前を指定します。
7	itemLabel		○		各項目のラジオボタンの名前を指定します。 属性「items」のリストの要素の「プロパティの名前」を指定します。 <input type="radio">タグの直後に、タグ<label>で囲まれ出力される文字列です。
8	itemValue		○		各項目のラジオボタンの値を指定します。 属性「items」のリストの要素の「プロパティの名前」を指定します。
9	delimiter		○		項目が複数あった場合、<input type="radio">のタグ間の区切り文字を指定します。例えば、 。
10	element		○	span	項目が複数あった場合、タグ<input>、<label>を囲むタグです、 例) <input…><label>～</label>
11	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- 各項目を Collection などから取得するために、属性「items」で指定します。
  Controller 側で、事前に Model やセッションスコープに登録しておく必要があります。
- Map から選択候補を取得する場合、マップのキーが属性「value」、マップの値が属性「label」に相当します。
- Colleciton の要素が JavaBean の場合、JavaBean のプロパティ名を指定します。
- タグ<ul>などのリストで表示したい場合は、タグ<li>で囲む必要があります。
  その場合、属性「element」で囲みたいタグの名称を指定します。

```
<!-- ラジオボタン：文字列型にバインドする(データをマップがから取得する) -->
<div>
    <span>この商品に満足しましたか？</span>
    <p>
        <form:radiobuttons path="answer1" items="${answerList1}"/>
    </p>
</div>

<!-- ラジオボタン：文字列型にバインドする(データを JavaBean のリストから取得する) -->

<div>
    <span>この商品に満足しましたか？</span>
    <ul>
        <form:radiobuttons path="answer2" items="${answerList2}" itemValue="code" itemLabel="label"
element="li"/>
    </ul>
</div>
```

Model などに登録しているデータを、EL 式で指定します。

JavaBean を要素に持つ場合、ラベルや値をどの値とするか、プロパティ名で指定します。

項目を囲みたいタグ名を指定します。

【Command の作成】

```
public class SampleCommand implements Serializable {
    // ラジオボタン（文字列型にバインドする）
    private String answer1;
    private String answer2;

    public SampleCommand() {}

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this).toString();
    }

    // getter、setter は省略
}
```

【Controller の作成】

- ラジオボタンの選択候補を、Map 型、JavaBean のリスト型で作成し Model などに登録します。
  - Model などの request スコープに登録している場合、エラーがあった場合の再描画時に再設定する必要があり、設定し忘れなどのバグの元にもなります。
  - 選択候補の値がシステムで普遍な場合は、システム起動時に application スコープに登録しておくことをお勧めします。
    システム起動時に設定する方法は、「14.2　アプリケーションの初期化プログラムの実行」を参照のこと。
- Map 型の場合は、順番を一定にするため、LinkedHashMap を使用します。

```
@Controller
@RequestMapping("/test/form2")
public class Form2Controller {

    // command の初期オブジェクトの取得
    @ModelAttribute("sampleCommand")
    public SampleCommand createInitCommand() {
        SampleCommand command = new SampleCommand();

        return command;
    }

    // 初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        SampleCommand command = createInitCommand();
        model.addAttribute("sampleCommand", command);

        // ラジオボタンのリストの選択候補の値
        model.addAttribute("answerList1", getAnswertDataFromMap());
        model.addAttribute("answerList2", getAnswerDataFromBean());

    }


    // ラジオボタンの選択候補のデータの取得（Map）
    private Map<String, String> getAnswertDataFromMap() {

        Map<String, String> map = new LinkedHashMap<String, String>();
        map.put("1", "満足しました");
        map.put("2", "普通です");
        map.put("3", "不満な点があります");
        return map;
    }

    // ラジオボタンの選択候補のデータの取得（JavaBean のリスト）
    private List<AnswerDto> getAnswerDataFromBean() {

        List<AnswerDto> list = /** DB などから取得 */
        return list;
    }
```

ラジオボタンの選択候補の値を、Model などに登録しておきます。

Map の場合は、順番を一定にするため、LinkedHashMap などを使用します。

```
    // post で送られた場合
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(@ModelAttribute("sampleCommand") SampleCommand command,
            BindingResult bindingResult) {

        // エラーがあるの場合
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            // ラジオボタンのリストの選択候補の再設定
            mav.addObject("answerList1", getAnswertDataFromMap());
            mav.addObject("answerList2", getAnswerDataFromBean());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;

    }
}
```

Model(Request)に選択候補の値を設定している場合、再描画の際に、再設定する必要があります。

【DTO(JavaBean)の作成】

```
public class AnswerDto {

    public AnswerDto() {}

    private String code;

    private String label;

    // getter、setter は省略

}
```

JSP のカスタムタグの属性「itemLabel」「itemValue」でプロパティの名前を指定します。

【生成された HTML】

- JSP のカスタムタグで属性「element」を指定しない場合、デフォルトでタグで囲まれます。

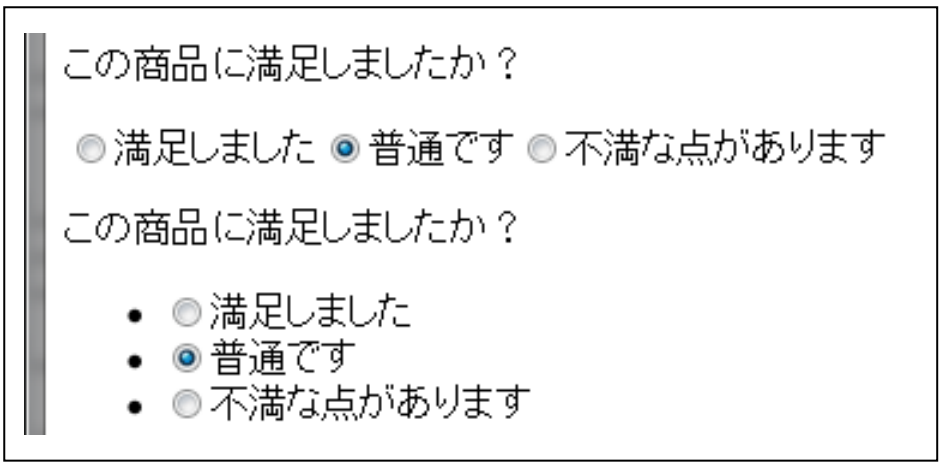
```
<!-- ラジオボタン：文字列型にバインドする(データをマップがから取得する) -->
<div>
    <span>この商品に満足しましたか？</span>
    <p>
        <span>
            <input id="answer11" name="answer1" type="radio" value="1"/>
            <label for="answer11">満足しました</label>
        </span>
        <span>
            <input id="answer12" name="answer1" type="radio" value="2" checked="checked"/>
            <label for="answer12">普通です</label>
        </span>
        <span>
            <input id="answer13" name="answer1" type="radio" value="3"/>
            <label for="answer13">不満な点があります</label>
        </span>
    </p>
</div>
```

カスタムタグの属性「element」を指定していない場合。

```
<!-- ラジオボタン：文字列型にバインドする(データを JavaBean のリストから取得する) -->
<div>
    <span>この商品に満足しましたか？</span>
    <ul>
        <li>
            <input id="answer21" name="answer2" type="radio" value="1"/>
            <label for="answer21">満足しました</label>
        </li>
        <li>
            <input id="answer22" name="answer2" type="radio" value="2" checked="checked"/>
            <label for="answer22">普通です</label>
        </li>
        <li>
            <input id="answer23" name="answer2" type="radio" value="3"/>
            <label for="answer23">不満な点があります</label>
        </li>
    </ul>
</div>
```

カスタムタグの属性「element」を指定した場合。

【ブラウザでの表示】



12.2.13. カスタムタグ<form:select>、<form:option>

HTML のタグ<select>を出力します。属性の指定により、タグ<option>も出力します。

【タグの仕様】

表 12.24 <form:select>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	items		○		<option>タグを生成するための、各項目のリスト(Collection、Map、配列)を指定します。
7	itemLabel		○		各項目の<option>タグの要素の文字列を指定します。 属性「items」のリストの要素の「プロパティの名前」を指定します。
8	itemValue		○		各項目の<option value="">の属性「value」を指定します。 属性「items」のリストの要素の「プロパティの名前」を指定します。
9	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

表 12.25 <form:option>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
2	cssClass		○		HTML の class 属性を出力します。
3	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指

					定します。
4	cssStyle		○		HTML の style 属性を出力します。
5	label		○		タグ<option>label</option>の要素の値を指定します。
6	value	○	○		タグ<option value="">～の属性「value」の値。
7	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- カスタムタグ<form:option>は、項目を 1 つずつ出力します。
  属性「label」を指定した場合、<option>属性 label の値</option>として出力されます。
- カスタムタグ<form:select>の属性「items」に、Model などに登録した Collection(リスト、マップ、配列)のオブジェクトを指定すると、複数のタグ<option>が生成されます。
  - JavaBean のリストから生成する場合は、属性「itemLabel」「itemValue」を指定します。
- カスタムタグ<form:select>の属性「multiple="multiple"」を設定すると、項目を複数選択することができます。
  - Command のプロパティは、List 型で定義する必要があります。

```
<!-- セレクトボックス -->
<div>
    <p>セレクトボックス</p>
    <form:select path="organization">
        <form:option value="">―</form:option>
        <form:option value="10" label="営業部"/>
        <form:option value="20">総務部</form:option>
        <form:option value="30">開発部</form:option>
    </form:select>
</div>
<!-- セレクトボックス：複数選択 -->
<div>
    <p>セレクトボックス(複数選択)</p>
    <form:select path="food1" items="${foodList}" multiple="multiple" size="4"></form:select>
</div>
```

バインド先のプロパティのパスを指定します。

項目名を属性「label」で指定すると、HTML 生成時に要素のに出力されます。

選択項目を 1 ずつ作成します。

Model に登録しているデータを、EL 式で指定します。

複数選択したい場合、属性「multiple」を指定します。

【Command の作成】

```
public class SampleCommand implements Serializable {

    // セレクトボックス
    private String organization;
    // セレクトボックス（複数選択：multiple）
    private List<String> food1;

    public SampleCommand() {
        food1 = ListUtils.lazyList(
                new ArrayList<String>(),
                FactoryUtils.instantiateFactory(String.class));
    }
    // getter、setter は省略
}
```

JSP で属性「multiple」を指定し、複数選択するので、リスト型で定義します。

【Controller の作成】

```
@Controller
@RequestMapping("/test/form2")
public class Form2Controller {

    ・・・省略

    // 初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        SampleCommand command = createInitCommand();

        // セレクトボックスのリストの選択候補の値
        model.addAttribute("foodList", getFoodDataFromMap());
    }

    // セレクトボックスの選択候補のデータの取得（Map）
    private Map<String, String> getFoodDataFromMap() {

        Map<String, String> map = new LinkedHashMap<String, String>();
        map.put("apple", "リンゴ");
        map.put("banana", "バナナ");
        map.put("orange", "オレンジ");

        return map;
    }

    // post で送られた場合
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(@ModelAttribute("sampleCommand") SampleCommand command,
            BindingResult bindingResult) {

        // エラーがある場合
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            // セレクトボックスのリストの選択候補の再設値
            mav.addObject("foodList", getFoodDataFromMap());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

選択候補の値を Model などに登録しておきます。

Model(Request)に選択候補の値を設定している場合、再描画の際に再設定する必要があります。

【生成された HTML（初期表示：何も選択していない場合）】

- 属性「id」「name」を記述していない場合、属性「path」をもとに自動的に付与されます。

```
<!-- セレクトボックス -->
<div>
    <p>セレクトボックス</p>
    <select id="organization" name="organization">
        <option value="">―</option>
```

```
            <option value="10">営業部</option>
            <option value="20">総務部</option>
            <option value="30">開発部</option>
        </select>
</div>
<!-- セレクトボックス：複数選択 -->
<div>
        <p>セレクトボックス(複数選択)</p>
        <select id="food1" name="food1" multiple="multiple" size="4">
            <option value="apple">リンゴ</option>
            <option value="banana">バナナ</option>
            <option value="orange">オレンジ</option>
        </select>
        <input type="hidden" name="_food1" value="1" />
</div>
```

自動的に生成されるタグです。

【生成された HTML（再描画時・初期値設定値：選択済みの場合）】

- 属性「checked="checked"」が自動的に付与された状態となります。

```
<!-- セレクトボックス -->
<div>
        <p>セレクトボックス</p>
        <select id="organization" name="organization">
            <option value="">―</option>
            <option value="10">営業部</option>
            <option value="20" selected="selected">総務部</option>
            <option value="30">開発部</option>
        </select>
</div>
<!-- セレクトボックス：複数選択 -->
<div>
        <p>セレクトボックス(複数選択)</p>
        <select id="food1" name="food1" multiple="multiple" size="4">
            <option value="apple">リンゴ</option>
            <option value="banana" selected="selected">バナナ</option>
            <option value="orange" selected="selected">オレンジ</option>
        </select>
        <input type="hidden" name="_food1" value="1" />
</div>
```

【ブラウザの表示】

12.2.14. カスタムタグ<form:options>

Collection、Map、配列から、タグ<option>を出力します。

【タグの仕様】

表 12.26 <form:options>のタグの仕様

No.	属性名	必須	EL 式	初期値	説明
1	path	○	○		データバインド時の Command のプロパティのパスを指定します。
2	htmlEscape		○		各フィールドを出力する際に、エスケープするか指定します。「true」「false」を指定します。
3	cssClass		○		HTML の class 属性を出力します。
4	cssErrorClass		○		「path」で指定したプロパティに対してフィールドエラーがある場合、出力される HTML の class 属性を指定します。
5	cssStyle		○		HTML の style 属性を出力します。
6	items	○	○		各項目のリスト（Collection、Map、配列）のオブジェクトが格納されている名前を指定します。
7	itemLabel		○		各項目のタグ<option>label</option>の要素の値を指定します。 属性「items」のリストの要素の「プロパティの名前」を指定します。
8	itemValue		○		各項目のタグ<option value="">の属性「value」を指定します。 属性「items」のリストの要素の「プロパティの名前」を指定します。
9	他、HTML の属性と同じ。		○		属性「class」「style」を指定したい場合、それぞれ「cssClass、cssErrorClass」「cssStyle」を使用します。

【タグの使用例】

- 各項目を Collection などから取得するために、属性「items」で指定します。
  Controller 側で、事前に Model やセッションスコープに登録しておく必要があります。
- Map から選択候補を取得する場合、マップのキーが属性「value」、マップの値が属性「label」に相当します。
- Colleciton の要素が JavaBean の場合、JavaBean のプロパティ名を指定します。

```
<!-- セレクトボックス：データをマップから取得する -->
<div>
    <p>セレクトボックス(マップから項目を取得)</p>
    <form:select path="food2">
        <form:option value="">—</form:option>
        <form:options items="${foodList}" />
    </form:select>
</div>

<!-- セレクトボックス：データを JavaBean から取得する -->
<div>
    <p>セレクトボックス(JavaBean から項目を取得する)</p>
    <form:select path="food3">
        <form:option value="">—</form:option>
        <optgroup label="野菜">
            <form:options items="${foodVegetableList}" itemValue="code" itemLabel="label"/>
        </optgroup>
        <optgroup label="果物">
            <form:options items="${foodFruitList}" itemValue="code" itemLabel="label"/>
        </optgroup>
    </form:select>
</div>
```

Model などに登録しているデータを EL 式で指定します。

JavaBean を要素に持つ場合、ラベルや値をどの値とするかプロパティ名で指定します。

【Command の作成】

```
public class SampleCommand implements Serializable {

    // セレクトボックス
    private String food2;

    // セレクトボックス
    private String food3;

    public SampleCommand() {
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this).toString();
    }

    // getter、setter は省略
}
```

【Controller の作成】

- セレクトボックスの選択候補を、Map 型、JavaBean のリスト型で作成し Model などに登録します。
  - Model などの request スコープに登録している場合、エラーがあった場合の再描画時に再設定する必要があり、設定し忘れなどのバグの元にもなります。
  - 選択候補の値がシステムで普遍な場合は、システム起動時に application スコープに登録しておくことをお勧めします。
    システム起動時に設定する方法は、「14.2　アプリケーションの初期化プログラムの実行」を参照のこと。
- Map 型の場合は、順番を一定にするため、LinkedHashMap を使用します。

```
@Controller
@RequestMapping("/test/form2")
public class Form2Controller {

    // command の初期オブジェクトの取得
    @ModelAttribute("sampleCommand")
    public SampleCommand createInitCommand() {
        SampleCommand command = new SampleCommand();
        return command;
    }

    // 初期値の設定
    @RequestMapping(method=RequestMethod.GET)
    public void setupForm(Model model) {

        SampleCommand command = createInitCommand();
        // セレクトボックスの初期値
        command.setFood2("apple");
        command.setFood3("tomato");
        model.addAttribute("sampleCommand", command);

        // セレクトボックスのリストの選択候補の値
        model.addAttribute("foodList", getFoodDataFromMap());
        model.addAttribute("foodFruitList", getFoodDataFromBean1());
        model.addAttribute("foodVegetableList", getFoodDataFromBean2());

    }

    // セレクトボックスの選択候補のデータの取得（Map）
    private Map<String, String> getFoodDataFromMap() {

        Map<String, String> map = new LinkedHashMap<String, String>();
        map.put("apple", "リンゴ");
        map.put("banana", "バナナ");
        map.put("orange", "オレンジ");
        return map;
    }

    // セレクトボックスの選択候補のデータの取得（JavaBean のリスト）
    private List<FoodDto> getFoodDataFromBean1() {

        List<FoodDto> list = /* DB などから取得します */
        return list;
    }
```

初期値を設定する必要がある場合、設定します。

セレクトボックスの選択候補の値を、Model などに登録します。

Map の場合は、順番を一定にするため、LinkedHashMap などを使用します。

```
    // セレクトボックスの選択候補のデータの取得（JavaBean のリスト）
    private List<FoodDto> getFoodDataFromBean2() {

        List<FoodDto> list = /* DB などから取得します */
        return list;
    }

    // post で送られた場合
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView doAction(@ModelAttribute("sampleCommand") SampleCommand command,
            BindingResult bindingResult) {

        // エラーがある場合
        if(bindingResult.hasErrors()) {
            ModelAndView mav = new ModelAndView();
            mav.getModel().putAll(bindingResult.getModel());

            // セレクトボックスのリストの選択候補の再設値
            mav.addObject("foodList", getFoodDataFromMap());
            mav.addObject("foodFruitList", getFoodDataFromBean1());
            mav.addObject("foodVegetableList", getFoodDataFromBean2());

            return mav;
        }

        ModelAndView mav = new ModelAndView("forward:/hello.html");
        return mav;
    }
}
```

Model(Request)に選択候補の値を設定している場合、再描画の際に、再設定する必要があります。

【DTO(JavaBean)の作成】

```
public class FoodDto {

    public FoodDto() {

    }

    private String code;

    private String label;

    // getter、setter は省略

}
```

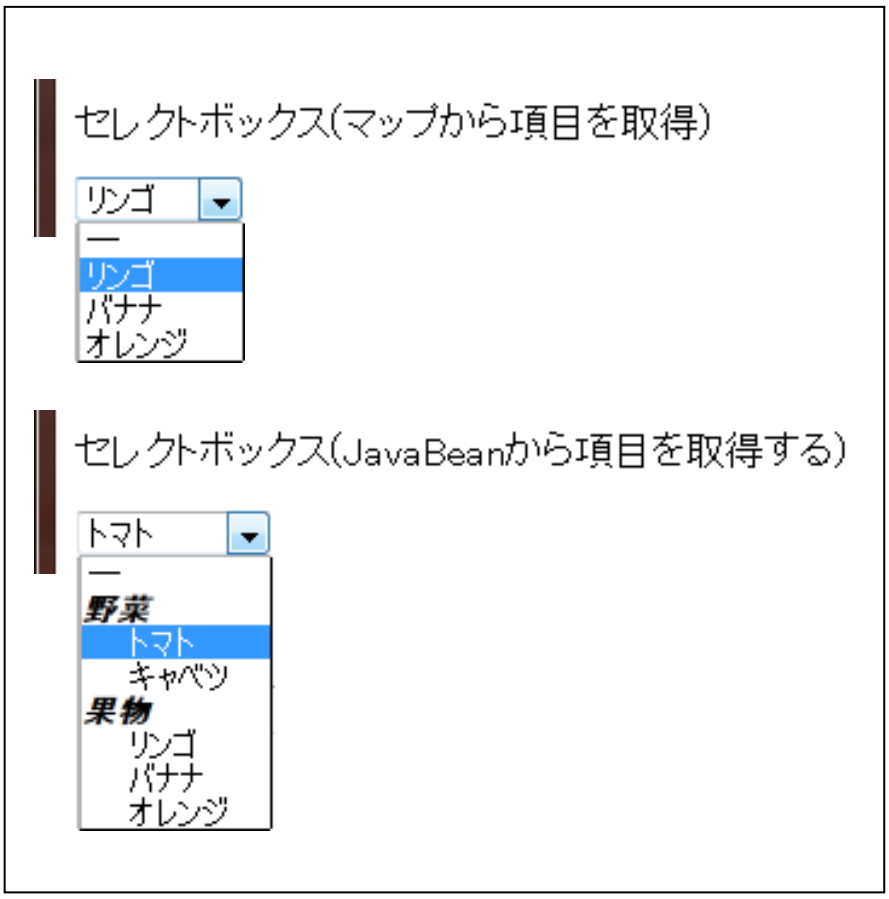
JSP のカスタムタグの属性「itemLabel」「itemValue」でプロパティの名前を指定します。

【生成された HTML】

```
<!-- セレクトボックス：データをマップから取得する -->
<div>
    <p>セレクトボックス(マップから項目を取得)</p>
    <select id="food2" name="food2">
        <option value="">―</option>
        <option value="apple">リンゴ</option>
        <option value="banana">バナナ</option>
        <option value="orange">オレンジ</option>
    </select>
</div>
```

```
<!-- セレクトボックス：データを JavaBean から取得する -->
<div>
    <p>セレクトボックス(JavaBean から項目を取得する)</p>
    <select id="food3" name="food3">
        <option value="">―</option>
        <optgroup label="野菜 ">
            <option value="tomato">トマト</option>
            <option value="cabbage">キャベツ</option>
        </optgroup>
        <optgroup label="果物">
            <option value="apple">リンゴ</option>
            <option value="banana">バナナ</option>
            <option value="orange">オレンジ</option>
        </optgroup>
    </select>
</div>
```

【ブラウザの表示】



12.3. 標準タグライブラリ JSTL

- 参考サイト「http://struts.wasureppoi.com/jstl/00_jstl.html」

表 12.27　JSTL ライブラリの種類

No.	種類	インポート方法
1	Core ライブラリ	ループ、条件分岐、変数の参照などの基本的なタグ。
2	il8n ライブラリ	数値・日付のフォーマットや国際化対応機能をサポートするタグ。
3	XML ライブラリ	XML の解析、XSLT を提供し XML を変換するタグ。 通常は使用しません。
4	Database ライブラリ	SQL を発行し、DB を操作するタグ。 通常は JSP では使用しません。DB の操作は DAO を経由します。

【pom.xml の設定】

```
<project>
    ・・・省略
    <dependencies>
        <!-- Tag Library -->
        <dependency>
            <groupId>taglibs</groupId>
            <artifactId>standard</artifactId>
            <version>1.1.2</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>
    </dependencies>
    ・・・省略
</project>
```

13. ページをタイル化する

13.1. Apache Tiles（:TODO）

Spring MVC では、View の一種として、もともと Struts のプロジェクトの 1 つとして開発されていた「Apache Tiles」をサポートしています。バージョン 1.x 系と、2.x 系の両方をサポートしていますが、パッケージが異なるので注意してください。

Apache Tiles のバージョン	Spring のパッケージ
Ver1.1+	org.springframework.web.servlet.view.tiles
Ver2.x+	org.springframework.web.servlet.view.tiles2

【参考】

http://d.hatena.ne.jp/GreenTea2010/20110529/1306684496

http://d.hatena.ne.jp/GreenTea2010/20110529/1306664637

13.2. SiteMesh

ページを部品としてタイルに分割（ヘッダー、フッター、メニュー、ボディ）するビュー用フレームワークとして、Struts で使用されている「Apache Tiles」が有名ですが、ここでは SiteMesh を使用します。

SiteMesh は、非常に拡張性が高く、カスタマイズのしやすいやり方で設計されています。

表 13.1　SiteMesh と Apache Tiles との比較

項目	SiteMesh	Apache Tiles
タイル化した際のアクセス URL	元の URL と変更はありません。	定義ファイルで設定した URL に変わる。
環境、フレームワークへの依存	依存しません。	依存します。 それぞれのフレームワークに対してプラグインなどが必要です。
レイアウト定義のファイルの種類	JSP 以外にも、Velocity、Freemaker など対応しています。 JSP ではカスタムタグを利用します。	JSP のみに対応しています。 カスタムタグを利用します。
タイル化を適用するページの修正	修正する必要がありません。	<body>タグの中身以外を残して修正する必要があります。
タイル化するページの指定方法	URL のパターンを使用し一括指で定可能です。	適用するページ 1 つずつ定義する必要があります。
動的にタイルの中身を切り替える	できません。固定になります。	カスタムタグを利用しできますが、定義ファイルで記述する必要があります。

【参考】

- SiteMesh の本家のページ

http://wiki.sitemesh.org/display/sitemesh/Home

- 書籍「現場で使える Java ライブラリ」、ISBN：4798123366

13.2.1. SiteMesh の準備

- SiteMesh には、2.4 系と 3.0 系がありますが、2012 年 10 月現在、3.0 系は alpha 版で正式版がリリースされいないので、ここでは 2.4 系について説明します。
- SiteMesh2.4 系を利用する場合、Tomcat5.x 以上（Servlet2.3+）を利用します。また、公式には Tomcat7.x ではサポートされていない（テストしていない）ということですが普通に動作します。

【pom.xml】

- 依存ライブラリとして、SiteMesh を追加します。

```
<dependencies>
    <dependency>
        <groupId>opensymphony</groupId>
        <artifactId>sitemesh</artifactId>
        <version>2.4.2</version>
    </dependency>
</dependencies>
```

表 13.2　SiteMesh を利用するに当たり作成・編集するファイル

No.	ファイル	説明	参照
1	/WEB-INF/web.xml	SiteMesh 用の ServletFilter の定義を追加します。	13.2.2
3	/WEB-INF/sitemesh.xml	SiteMesh の環境設定ファイルです。 必須ではありませんが、細かな動作を変更するのに利用します。	13.2.3
2	/WEB-INF/decorators.xml	SiteMesh 用のレイアウト定義用 XML です。 Apache Tiles の「tiles-defs.xml」ファイルに相当します。	13.2.4
4	/WEB-INF/decorators/	ベースとなる JSP やヘッダーなどのタイル用 JSP を格納します。	13.2.5

13.2.2. 「web.xml」の編集

既存の「/WEB-INF/web.xml」に、SiteMesh 用のフィルタを登録します。

```
<web-app>
    <filter>
        <filter-name>sitemesh</filter-name>
        <filter-class>com.opensymphony.sitemesh.webapp.SiteMeshFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>sitemesh</filter-name>
        <url-pattern>/*</url-pattern>
        <dispatcher>REQUEST</dispatcher>
        <dispatcher>FORWARD</dispatcher>
    </filter-mapping>
</web-app>
```

13.2.3.「sitemesh.xml」の作成

このファイルは必須ではありませんが、標準の設定を変更したいときに利用します。

- ファイル名、配置場所は必ず「/WEB-INF/sitemesh.xml」としてください。

【標準の sitemesh.xml】

SiteMesh の jar ファイルの「com.opensymphony.module.sitemesh.factory.sitemesh-default.xml」に格納されています。

「decorators.xml」の配置場所、名前を定義します。。

```
<?xml version="1.0" encoding="UTF-8"?>
<sitemesh>
    <property name="decorators-file" value="/WEB-INF/decorators.xml"/>
    <excludes file="${decorators-file}"/>

    <page-parsers>
        <parser content-type="text/html" class="com.opensymphony.module.sitemesh.parser.HTMLPageParser" />
    </page-parsers>

    <decorator-mappers>
        <mapper class="com.opensymphony.module.sitemesh.mapper.PageDecoratorMapper">
            <param name="property.1" value="meta.decorator" />
            <param name="property.2" value="decorator" />
        </mapper>
        <mapper class="com.opensymphony.module.sitemesh.mapper.FrameSetDecoratorMapper"/>
        <mapper class="com.opensymphony.module.sitemesh.mapper.PrintableDecoratorMapper">
            <param name="decorator" value="printable" />
            <param name="parameter.name" value="printable" />
            <param name="parameter.value" value="true" />
        </mapper>
        <mapper class="com.opensymphony.module.sitemesh.mapper.FileDecoratorMapper"/>
        <mapper class="com.opensymphony.module.sitemesh.mapper.ConfigDecoratorMapper">
            <param name="config" value="${decorators-file}" />
        </mapper>
    </decorator-mappers>
</sitemesh>
```

13.2.4.「decorators.xml」の作成

- 「/WEB-INF/decorators.xml」に格納してください。標準では、空もでも構いません。
  - ファイル名や配置場所を変更したい場合は、「sitemesh.xml」で変更できます。
- 設定ファイルに対する DTD、XML Schema はないので、要素名、属性名に間違いのないようにしてください。

【基本の初期状態】

- 属性「defaultdir」で、レイアウト用の JSP（タイル）を格納するディレクトリを定義します。
- 要素<decorator>で、レイアウト用 JSP と、適用する URL パターンを定義します。
  - 属性「name」で名前を付けます。他の定義と重複しなようにします。
  - 属性「page」でレイアウト用の JSP のパスを記述します。これは、「defaultdir」で定義したパスからの相対パスになります。
  - 複数定義することができます。
- 子要素<pattern>で適用する URL のパターンを定義します。
  - 値を「/*」とすると全ての URL に対して適用されてしまうため、通常は、「/edit/*」などと限定して定義します。
  - SiteMesh を適用する URL が HTML を出力する URL ならば問題ありませんが、**ファイルダウンロードや JSON などを返す URL に対して適用すると不正**となりますので、**URL 設計を行ってから適用**してください。
- URL パターン以外の適用方法で、カスタムタグ<page:applyDecorator>による方法があります。
  - 詳細は、「13.2.6.6　カスタムタグ<page:applyDecorator>」を参照してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- SiteMesh のデコエレータを定義するファイル -->
<decorators defaultdir="/WEB-INF/decorators">
    <decorator name="base" page="base.jsp">
        <pattern>/*</pattern>
    </decorator>
    <decorator name="panel" page="panel.jsp"/>
</decorators>
```

・子要素<pattern>で SiteMesh を適用する URL パターンを定義します。
・「/*」とすると、全ての URL に対して適用されます。

子要素<pattern>がない場合、カスタムタグで独自に適用することになります。
その際に、属性「name」で名前を付けることで識別します。

13.2.5. レイアウト用 JSP の作成

【概要】

- 「13.2.4 「decorators.xml」の作成」で定義したレイアウト用 JSP を作成します。
- SiteMesh 用カスタムタグ<decorator:XXX />を利用し、作成します。
  - カスタムタグの詳細は、「13.2.6　SiteMesh 用のカスタムタグライブラリ」を参照してください。
  - XPath のように、コンテンツ本体の情報が取得できるため、Apache Tiles のようにコンテンツ側に対して修正は必要ありません。
- メニューなどの共通な部分は、レイアウト用ファイルに定義してもかまいませんが、JSP 標準のカスタムタグ<jsp:include />で読み込んだりもできます。

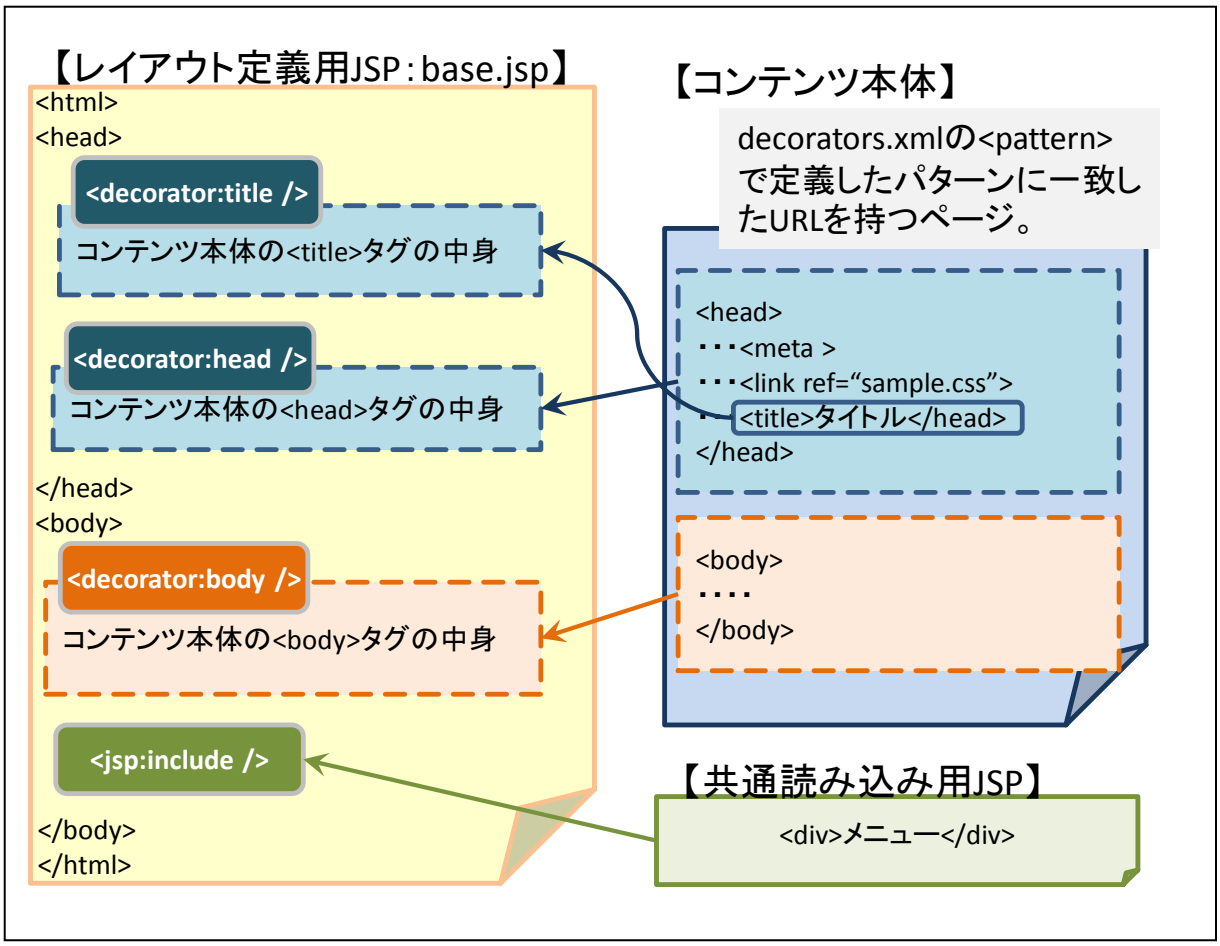


図 13.1　レイアウト用 JSP の概要

【実際のサンプル】

- SiteMesh 用のカスタムタグの定義をします。基本的なことは、<decorator:XXX>のみで足りますが、他のレイアウト用ページを呼び出すときなどには、<page:XXX>を使用します。
- タイトルは<decorator:title>タグで取得できるため、<decorator:head/>タグには含まれないので注意してください。
  - ただし、<decorator:head/>タグには、<meta>タグや、<link>タグなどの、コンテンツ本体の他の全てのタグが含まれます。
- **文字コードに関する定義は、レイアウト用とコンテンツ本体は全て統一**する必要があります。
  - ページディレクティブで定義している JSP ファイルの文字コード(<%page pageEncoding=””%>)。
  - <meta>タグで定義する HTML のコンテンツの文字コード。
    - コンテンツ本体に文字コードを<meta>タグで定義している場合は、レイアウト用ページには必要ありません。
    - レイアウト用ページとコンテンツ本体の両方に定義している場合、HTML の出力時に**定義が重複してしまいますが動作に問題はありません**。しかし、Another HTML などの**文法チェックツールではエラーとなっていしまいます**ので注意してください。

```
<%@ page language="java" contentType="text/html; charset=UTF-8" trimDirectiveWhitespaces="true"
pageEncoding="UTF-8"%>
<%-- ▼ taglib --%>
<%@ taglib uri="http://www.opensymphony.com/sitemesh/decorator" prefix="decorator" %>
<%@ taglib uri="http://www.opensymphony.com/sitemesh/page" prefix="page" %>
<%-- ▲ taglib --%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

    <title><decorator:title default="Sample"/></title>
    <decorator:head/>
</head>
<body onload='<decorator:getProperty property="body.onload" />'>

    <h1>Header</h1>
    <p><b>Navigation</b></p>
    <hr>

    <decorator:body/>

    <hr>
    <h1><b>Footer</b></h1>
</body>
</html>
```

SiteMesh 用のカスタムタグの定義。

コンテンツ本体の<title>タグの中身を取得します。

コンテンツ本体の<head>タグの中身を取得します。

コンテンツ本体の<body>タグの onload 属性の値を取得します。

コンテンツ本体の<body >タグの中身を取得します。

13.2.6. SiteMesh 用のカスタムタグライブラリ

- 本家説明ページ
  - 「http://wiki.sitemesh.org/display/sitemesh/Tag+References」

13.2.6.1. カスタムタグ<decorator:head>

- オリジナルのページの<head>～</head>タグの内容を取得し、出力します。
- ただし、<title>タグについては、別途カスタムタグ<decorator：title>（「13.2.6.2　カスタムタグ<decorator:title>」参照）が用意されているため、このタグを利用してもタイトルは取得できません。
- CSS 用や、文字コード指定の<meta>タグなど、レイアウト定義用の JSP に取得元のオリジナルページと同じ記述があると、重複して出力されます。そのため、Another HTML などで文法チェックすると警告が出る場合があります。

表 13.3　<decorator:head>の仕様

No.	項目	内容
1	タグの形式	Self-close タグで、コンテンツを持ちません。
2	属性	ありません。

【使用例】

- レイアウト用のページで利用します。

```
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <link rel="stylesheet" type="text/css" media="all" href="${appUrl}/css/base.css">
    <decorator:head />
</head>
```

13.2.6.2. カスタムタグ<decorator:title>

- オリジナルのページの<title>～</title>タグの内容を取得し、出力します。
- ページ内のどこにでも適用できるため、大見出し(<h1>)の値としても出力できます。

表 13.4　<decorator:title>の仕様

No.	項目	内容
1	タグの形式	Self-close タグで、コンテンツを持ちません。
2	属性	・「default」：初期値を設定します。オプションです。 オリジナルのページに<title>タグが定義されていない場合、定義されているが中身が空（半角空白も含む）場合に、代わりに出力されます

【使用例】

- レイアウト用ページで使用します。

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title><decorator:title default="Sample"/></title>
</head>
<body>
    <h1><decorator:title default="Sample"/></h1>
</body>
</html>
```

13.2.6.3. カスタムタグ<decorator:body>

- オリジナルのページの<body>～</body>タグの中身を取得し、出力します。
- <body>タグの中身全てが出力されるので、レイアウトをデザインする際に、オリジナルのページはシンプルな構造にし、 などで構造を深くしすぎないデザインするとことをお勧めします。

表 13.5　<decorator:body>の仕様

No.	項目	内容
1	タグの形式	Self-close タグで、コンテンツを持ちません。
2	属性	ありません。

【使用例】

- レイアウト用ページで使用します。

```
<body>
    <h1>Header</h1>
    <hr>

    <decorator:body/>

    <hr>
    <h1><b>Footer</b></h1>
</body>
```

13.2.6.4. カスタムタグ<decorator:getProperty>

- オリジナルのページの特定のタグ（表 13.7）の値、属性の値を取得することができます。

表 13.6 <decorator:getProperty >の仕様

No.	項目	内容
1	タグの形式	Self-close タグで、コンテンツを持ちません。
2	属性	・「property」: 取得する情報のプロパティ（タグ名、属性名）を指定します。必須です。 基本は、「タグ名.属性名」で指定します。属性名は省略できます。 ・「default」: プロパティが取得でない代わりに出力する値です。オプションです。 ・「writeEntireProperty」: 指定したプロパティの属性の形式全体を取得することができます。オプションです。 値は、「true」「yres」「1」の何れかを取り、全て同じ意味です。 形式は、「プロパティ名=”値”」となります。

表 13.7 <decorator:getProperty>で取得可能なタグの情報

No.	タグ	内容
1	<html>	・<html>タグの lang 属性などの全ての属性を取得できます。 ・属性名を指定する際には大文字・小文字の区別があります。 【property の指定例】 ・「lang」: <html>タグの lang 属性を取得します。 ・「xml:lang」: <html>タグの xml:lang 属性を取得します。 ・タグ名“html”は省略して属性名のみ指定します。
2	<title>	・<title>タグのコンテンツの内容を取得します。 ・<decorator:title/>と同じです。 【property の指定例】 ・「title」: <title>タグの内容を取得します。
3	<meta>	・<meta>タグの属性「content」の内容を取得します。 ・どのタグの情報を取得するかは属性「name」の値をもとにします。 【property の指定例】 ・「 meta.description 」: <meta name=”description” content=”AAA”> の 属 性

		「content」を取得します。 ・「meta.author」: <meta name=”author” content=”Hoge”>の属性「content」を取得します。 ・タグ名“meta”は省略できません。
4	<body>	・<body>タグの bgcolor、onload などの全ての属性を取得できます。 ・属性名を指定する際には<u>大文字・小文字の区別があります</u>。 【propety の指定例】 ・「body.onload」: <body>タグの onload 属性を取得します。 ・タグ名“body”は省略できません。

【使用例１】

- オリジナルページ

```
<html lang="ja" xml:lang="ja_JP">
<head>
    <title>ページのタイトル</title>
    <meta name="description" content="ページの説明 特殊文字">
    <meta name="author" content="ページの作成者&amp;特殊文字">
    ・・・
</head>
<body bgcolor="green" onload="document.someform.somefield.focus();">
    ・・・
</body>
```

- レイアウト用ページの JSP

```
<ul>
    <!-- html タグの属性 -->
    <li><decorator:getProperty property="lang"/></li>
    <li><decorator:getProperty property="xml:lang"/></li>

    <!-- title タグ -->
    <li><decorator:getProperty property="title"/></li>

    <!-- meta タグ -->
    <li><decorator:getProperty property="meta.description"/></li>
    <li><decorator:getProperty property="meta.author"/></li>

    <!-- body タグ -->
    <li><decorator:getProperty property="body.bgcolor"/></li>
    <li><decorator:getProperty property="body.onload"/></li>
</ul>
```

- レイアウト用ページのレンダリング結果（HTML）
  - タグのコンテンツとして定義可能な形式として出力されます。

```
<ul>
    <!-- html タグの属性 -->
    <li>ja</li>
    <li>ja_JP</li>
```

```
    <!-- title タグ -->
    <li>ページのタイトル</li>

    <!-- meta タグ -->
    <li>ページの説明&amp;nbsp;特殊文字</li>
    <li>ページの作成者&amp;amp;特殊文字</li>

    <!-- body タグ -->
    <li>green</li>
    <li>document.someform.somefield.focus();</li>
</ul>
```

【使用例 2：属性「writeEntireProperty=”true”」を指定する場合】

- レイアウト用のページの JSP

```
<ul>
    <!-- html タグの属性 -->
    <li><decorator:getProperty property="lang" writeEntireProperty="true" /></li>
    <li><decorator:getProperty property="xml:lang" writeEntireProperty="true" /></li>

    <!-- title タグ -->
    <li><decorator:getProperty property="title" writeEntireProperty="true" /></li>

    <!-- meta タグ -->
    <li><decorator:getProperty property="meta.description" writeEntireProperty="true" /></li>
    <li><decorator:getProperty property="meta.author" writeEntireProperty="true" /></li>

    <!-- body タグ -->
    <li><decorator:getProperty property="body.bgcolor" writeEntireProperty="true" /></li>
    <li><decorator:getProperty property="body.onload" writeEntireProperty="true" /></li>
</ul>
```

- レイアウト用ページのレンダリング結果（HTML）
  - タグの属性として定義可能な形式（属性名=”値”）として出力されます。

```
<ul>
    <!-- html タグの属性 -->
    <li> lang="ja"</li>
    <li> xml:lang="ja_JP"</li>

    <!-- title タグ -->
    <li> title="ページのタイトル"</li>

    <!-- meta タグ -->
    <li> description="ページの説明&amp;nbsp;特殊文字"</li>
    <li> author="ページの作成者&amp;amp;特殊文字"</li>

    <!-- body タグ -->
    <li> bgcolor="green"</li>
    <li> onload="document.someform.somefield.focus();"</li>
</ul>
```

13.2.6.5. カスタムタグ<decorator:usePage>

- <decorator:getProperty>のオブジェクトを JSP のページスコープの変数として保存し、カスタムタグを経由しないでページのプロパティを取得できるようになります。
- 保存されるオブジェクトの実体は、「com.opensymphony.module.sitemesh.Page」。(「表 13.9」参照)
  - 変数として保存した場合は、「Page#getProperty(…)」「Page#getTitle()」などでプロパティを呼び出せます。
  - Page クラスでプロパティを指定する際の書式は、「13.2.6.4　カスタムタグ<decorator:getProperty>」と同じです。

表 13.8　<decorator:usePage >の仕様

No.	項目	内容
1	タグの形式	self-close タグで、コンテンツを持ちません。
2	属性	・「id」: ページスコープのオブジェクトとして保存する際の変数名。**必須**です。

【使用例】

- 変数「orgPage」として保存します。
- EL 式中で呼び出すこともできます。
  - 取得した値は、アンエスケープされていないので、出力する際はエスケープする必要はありません。

```
<decorator:usePage id="orgPage"/>
<ul>
    <li>${orgPage.getProperty('lang')}</li>

    <li><c:out value="${orgPage.getProperty('meta.author')}" escapeXml="false" /></li>
</ul>

<!-- 数値として取得する -->
<%if ( orgPage.getIntProperty("rating") == 10 ) { %>
  <b>10 out of 10!</b>
<% } %>
```

EL 式からでも利用できます。

取得した値はエスケープする必要はありません。

int 型として取得できます。

表 13.9　「com.opensymphony.module.sitemesh.Page」クラスの重要なメソッド

No.	メソッド :: 戻り値	説明
1	getBody() :: String	・<body>タグの中身を全て出力します。 ・<decorator:body>と同じ動作をします。
2	getTitle() :: String	・<title>タグの中身を出力します。 ・<decorator:title>と同じ動作をします。
3	getProperty(String name) :: String	・引数でプロパティを指定し、**文字列**として取得します。
4	getIntProperty(String name) :: int	・引数でプロパティを指定し、**int 型**として取得します。

5	getLongProperty(String name) :: long	・引数でプロパティを与えて、**long 型**として取得します。
6	getBooleanProperty(String name) :: boolean	・引数でプロパティを与えて、**boolean 型**として取得します。
7	getProperties() :: Map<String, Object>	・全てのプロパティを、プロパティ名をキーとしてマップオブジェクトとして取得します。
8	getPropertyKey() :: String[]	・全てのプロパティの名前を配列として取得します。

13.2.6.6. カスタムタグ<page:applyDecorator>

- レイアウト用ページの中で、任意のデコレータをさらに適用することができます。
  - 適用するデコレータは、「decorators.xml（「13.2.4　「decorators.xml」の作成)」参照」で定義した名前付きのもに限ります。

表 13.10　<page:applyDecorator >の仕様

No.	項目	内容
1	タグの形式	・コンテンツを持ちます。 ・コンテンツの中身は、適用されるデコレータ中で**<decorator:body>で取得できる内容**となります。 ・<page:param>（「13.2.6.7　カスタムタグ<page:param>」参照）を要素として持ちます。
2	属性	・「name」: decorators.xml で定義したデコレータ名（レイアウト名）を指定します。**必須**です。 ・「page」: オプションです。 ・「title」: オプションです。

【使用例】

・「decorators.xml」

```
<decorators defaultdir="/WEB-INF/decorators">

    <decorator name="base" page="base.jsp">
        <pattern>/*</pattern>
    </decorator>

    <decorator name="panel" page="panel.jsp"/>

</decorators>
```

name 属性で名前を設定します。

- 「panel.jsp」

```
<%@ page language="java" contentType="text/html; charset=UTF-8" trimDirectiveWhitespaces="true" pageEncoding="UTF-8"%>
```

```
<%@ taglib uri="http://www.opensymphony.com/sitemesh/decorator" prefix="decorator" %>
<%@ taglib uri="http://www.opensymphony.com/sitemesh/page" prefix="page" %>

<div class="panel">
    <div class="panelTitle">
        <decorator:title default="Panel Title"/>
    </div>
    <div class="panelBody">
        <decorator:body />
    </div>
</div>
```

- 適用先のレイアウト用 JSP

```
<page:applyDecorator name="panel" title="これはパネルです。">
    <page:param name="meta.author">パネルの著者です</page:param>

    <!-- パネルの body -->
    <p>
        今日はいい天気です
    </p>
</page:applyDecorator>
```

デコレータに渡されるページのプロパティを設定できます。

コンテンツ(<body>)として、デコレータに渡される。

13.2.6.7. カスタムタグ<page:param>

- <page:applyDecorator>と合わせて利用し、タグの要素として利用します。。
- <page:applyDecorator>でデコレータを適用したレイアウトに対して、ページのプロパティ（<decorator:getProperty>、<decorateor:usePage>で取得できる Page クラス）を変更することができます。

表 13.11　<page:param>の仕様

No.	項目	内容
1	タグの形式	・コンテンツを持ちます。 ・コンテンツの内容がページプロパティの値になります。
2	属性	・「name」: <page:applyDecorator>で起用先に対するページのプロパティ名を指定します。必須です。 <decorator:getProperty>などで呼び出すことができるプロパティ名を指定します。

【使用例】

- 「13.2.6.6　カスタムタグ<page:applyDecorator>」を参照してください。

14. ユーティリティ

14.1. ログ出力

14.1.1. Log4j の設定

ログを出力するには、基本は Log4j を SLF4J 経由で使用します。設定は、プロパティファイル、XML、Java の 3 種類で可能ですが、ここでは XML による設定例を図 14.1 に示します。

設定ファイルは、クラスパスのトップに設定しておくと自動的に読み込まれます。クラスパスのトップは、「/WEB-INF/classes/log4j.xml」になります。ソース上は「/src/main/resources/log4j.xml」となります。それ以外に配置した場合、web.xml で log4j.xml ファイルのパスを指定することができます。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "/org/apache/log4j/xml/log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <!-- 全てのファイルアペンダ -->
    <appender name="RootFileAppender"
        class="org.apache.log4j.RollingFileAppender">
        <param name="File" value="${catalina.home}/logs/root.log" />
        <param name="MaxFileSize" value="20MB"/>
        <param name="MaxBackupIndex" value="5"/>
        <param name="Encoding" value="UTF-8"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="%d %5p %-21t %-70c - %m%n" />
        </layout>
    </appender>

    <!-- デバッグ用のアペンダ：標準出力する -->
    <appender name="debugAppender"
        class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d %-5p - %-26.26c{1} - %m%n" />
        </layout>
    </appender>

    <!-- 全てのログ -->
    <root>
        <level value="ERROR" />
        <appender-ref ref="debugAppender" />
        <appender-ref ref="RootFileAppender" />
    </root>
</log4j:configuration>
```

ファイルに出力する書式や世代の切り替え方法を設定します。

変数を使用することができます。
・サーバ(Java)の起動オプション「-Dxxxx」で指定したもの。
・システムプロパティに指定したもの。
・web.xml の<context-param>で指定したもの。

標準出力の書式の設定を行います。

図 14.1　logj4.xml の設定例

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
・・・省略
    <!-- Log4j の設定ファイルの指定 -->
    <context-param>
        <param-name>log4jConfigLocation</param-name>
        <param-value>/WEB-INF/classes/log4j.xml </param-value>
    </context-param>

    <!-- 設定ファイルの Listener の設定 -->
    <listener>
        <listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>
    </listener>
・・・省略
</web-app>
```

設定ファイルをクラスパスのルート以外に配置している場合に記述します。省略可能です。

Lister を登録しておくことで、サーバ初期化時に Log4j も初期化される。

図　14.2　log4j.xml の web.xml での設定

14.1.2. ログの出力

Logj4 を SLF4j 経由で使用します。ロガークラス(org.slf4j.Logger)の取得は、「org.slf4j.LoggerFactory」クラスから取得します。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Sample {
    final static Logger logger = LoggerFactory.getLogger(Sample.class);

    public void doMethod() {
        logger.debug("hello logger");
        logger.debug("id={}", "ID を引数で渡す");   // 変数を使用
    }

}
```

図　14.3　SLF4J の使用例 1

```
// for 分で何度も呼び出す場合、現在の出力レベルをチェックします。これによりパフォーマンスが向上します。
Logger log = LogFactory.getLogger();
if(log.isDebugEnable()){
  log.debug("パラメータ="+param);
}

if(log.isTraceEnable()){
  for(int i=0; i< args.length;i++){
    log.trace("パラメータ["+i+"]="+args[i]);
  }
}
```

図　14.4　SLF4J の使用例 2

14.2. アプリケーションの初期化プログラムの実行

サーバ起動時に application スコープに共通変数を登録したいなどの初期化処理の方法を説明します。Servlet API のインタフェース「javax.servlet.ServletContextListener」を実装し、web.xml に登録します。

```
import java.net.MalformedURLException;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

import org.apache.commons.lang.StringUtils;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;

/**
 * アプリケーションの起動時・終了時に呼ばれるクラス。
 */
public class WebApplicationInitializer implements ServletContextListener {

    /**
     * サーバ起動時に呼ばれるメソッド
     */
    @Override
    public void contextInitialized(ServletContextEvent sce) {

        // システムプロパティの設定
        initSystemProperty();

        // Spring の Bean の呼び出し
        ServletContext servletContext = sce.getServletContext();
        WebApplicationContext appContext =
                WebApplicationContextUtils.getWebApplicationContext(servletContext);

        try {
            String appName = getAppName(servletContext);
            servletContext.setAttribute("appName", appName);
            servletContext.setAttribute("appUrl", "/" + appName);
        } catch(MalformedURLException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * サーバ停止時に呼ばれるメソッド
     */
    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        // TODO Auto-generated method stub
    }

    /**
     * アプリケーション名を取得する
     */
    private String getAppName(ServletContext servletContext) throws MalformedURLException {

        //「/ホスト名/APP 名/」の形式
        String path = servletContext.getResource("/").getPath();
        String[] split = StringUtils.split(path, "/");
```

SpringBean を呼び出す場合、WebApplicationContextutils を使用します。

JSP 中でリンク先に使用する「APP 名」をアプリケーションスコープに登録します。

```
        return split[1];
    }

    /**
     * システムプロパティを設定する。
     */
    private void initSystemProperty() {

        // システムプロパティ catalina.home の設定
        String catalinaHome = System.getProperty("catalina.home", "");

        if(StringUtils.isEmpty(catalinaHome)) {
            try {
                // 環境変数の CATALINA_HOME を取得する
                String envCatalinaHome = System.getenv("CATALINA_HOME");
                if(StringUtils.isEmpty(envCatalinaHome)) {
                    envCatalinaHome = System.getenv("TOMCAT_HOME");
                }

                // システムプロパティへ設定する
                if(StringUtils.isNotEmpty(envCatalinaHome)) {
                    System.setProperty("catalina.home", envCatalinaHome);
                }

            } catch(SecurityException e) {
                // なにもしない。
            }
        }
    }
}
```

Log4j の出力ディレクトリなどに使用します。

図 14.5　アプリケーションの初期化用クラス

```
<web-app>
・・・省略
    <!-- Spring などの初期化 -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/classes/ApplicationContext.xml</param-value>
    </context-param>

    <listener>
        <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
    </listener>

    <!-- 初期化用クラス -->
    <listener>
        <listener-class>sample.web.WebApplicationInitializer</listener-class>
    </listener>
・・・省略
</web-app>
```

初期化用クラスの定義。SpringBean を呼び出す場合、必ず ContextLoderListener よりも後に定義します。

図 14.6　初期化用クラスの「web.xml」の設定

14.3. ライブラリ「Commons Configuration」を使用する

プロパティファイルを扱う、「Commons Configuration」を Spring から使用する方法を説明します。Commons Configuration は、プロパティファイルを扱う場合、Java 標準の Properties、ResourceBundle や、Sprinig の MessageSources よりも高機能で使い勝手がよく、様々な形式のプロパティファイルに対応しています。

Commons Configuration を Spring から呼び出す方法は、2 つあります。

- 1 つめは、Spring Modules にある「CommonsConfigurationFactoryBean」を使う方法です。
  この場合、Java のプロパティ形式のみ対応していますが、設定ファイルの定義が簡単になります。
- 2 つめは、Spring Bean で直接「Configuration」のインスタンスを定義する方法です。
  この場合、通常の Java で直接使う方法と同じ定義ができ、様々な定義方法が可能です。

【準備】

- pom.xml の依存ファイルの定義に Commons Configuration を追加します。

```
<project>
    ・・・省略
    <dependencies>

        <dependency>
            <groupId>commons-configuration</groupId>
            <artifactId>commons-configuration</artifactId>
            <version>1.7</version>
        </dependency>

    </dependencies>
    ・・・省略
</project>
```

14.3.1. 「CommonsConfigurationFactoryBean」を使用する

Spring Module は、jar がないため、ソースを自分でコンパイルして使用します。実際には、ほかの余分な機能もあるため、使用するソースのみダウンロードし、APP のパッケージに直接組み込むと楽です。

(1)Java のソース「CommonsConfigurationFactoryBean.java」を下記の URL からダウンロードします。
「 http://java.net/projects/springmodules/sources/svn/content/trunk/projects/commons/src/java/org/springmodules/commons/configuration/CommonsConfigurationFactoryBean.java?rev=2110」

(2)パッケージ「org.springmodules.commons.configuration」を作成し、ダウンロードしたソースを格納します。

(3)Spring の設定ファイル「ApplicationContext.xml」に定義を追加します。プロパティファイルのパスを指定します。

```
<beans>
・・・省略
    <bean id="configuration"
class="org.springmodules.commons.configuration.CommonsConfigurationFactoryBean">
        <property name="locations">
            <list>
                <value>classpath:prop/config-sample1.properties</value>
                <value>classpath:rome.properties</value>
            </list>
        </property>
    </bean>
    ・・・省略
</beans>
```

(4)呼び出す際には、Configuration をインジェクションして利用します。

```
import javax.annotation.Resource;

import org.apache.commons.configuration.Configuration;
import org.springframework.stereotype.Component;

@Component
public class SampleComponnet {

    @Resource
    Configuration configuration;

    public void doProperties() {

        int value = configuration.getInt("sample1.int");

    }
}
```

CommonsConfiguration のインスタンスをインジェクションする。

14.3.2. Spring Bean として「Configutation」を定義、組み立てる

【単純な定義】

1 つのプロパティファイルを読み込む方法です。

- PropertiesConfigutation のインスタンスを定義します。
- コンストラクタの引数として、プロパティファイルのパスを定義します。
  クラスパスで指定する場合は、java.net.URL をインスタンスとして、接頭語「classpath:」を付けます。
- 変更されたら、自動的に再読み込みするために、FileChangeReloadingStrategy を指定します。

```
<beans>
・・・省略
    <bean id="simpleConfiguration" class="org.apache.commons.configuration.PropertiesConfiguration">
        <constructor-arg type="java.net.URL" value="classpath:prop/config-sample1.properties" />
        <property name="reloadingStrategy">
            <bean class="org.apache.commons.configuration.reloading.FileChangedReloadingStrategy"/>
        </property>
    </bean>
・・・省略
</beans>
```

- 呼び出し方法は、「14.3.1　「CommonsConfigurationFactoryBean」を使用する」と同じです。
  Bean 名称が simpleConfigutation と名前を変更している場合、name 属性で指定します。

```
    @Resource(name="simpleConfiguration")
    Configuration configuration;

    public void doProperties() {

        int value = configuration.getInt("sample1.int");

    }
```

- Spring Bean での定義と同義な Java での定義を次に示します。

```
FileConfiguration configuration = new PropertiesConfiguration(
    new URL("classpath:prop/config-sample1.properties"));
configuration.setReloadingStrategy(new FileChangedReloadingStrategy());
```

【複数のファイルを読み込む場合】

- CompositeConfigutation のインスタンスを定義します。
- コンストラクタの引数として、リストで Configuration の定義を追加します。
- リストには、Configuation の他のインスタンスの参照や直接定義します。
- 呼び出し方法は、「14.3.1 「CommonsConfigurationFactoryBean」を使用する」と同じです。

```
<beans>
・・・省略
    <!-- 複数のファイル -->
    <bean id="compositeConfiguration" class="org.apache.commons.configuration.CompositeConfiguration">
        <constructor-arg>
            <list>
                <!-- 他定義の Configutation をインジェクションする。 -->
                <ref bean="simpleConfiguration"/>

                <!-- 独自の XML -->
                <bean class="org.apache.commons.configuration.XMLConfiguration">
                    <constructor-arg type="java.net.URL" value="classpath:prop/config-sample2.xml" />
                    <property name="reloadingStrategy">
                        <bean
class="org.apache.commons.configuration.reloading.FileChangedReloadingStrategy"/>
                    </property>
                </bean>

                <!-- Java のプロパティ XML -->
                <bean class="org.apache.commons.configuration.XMLPropertiesConfiguration">
                    <constructor-arg type="java.net.URL" value="classpath:prop/config-sample3.xml" />
                    <property name="reloadingStrategy">
                        <bean
class="org.apache.commons.configuration.reloading.FileChangedReloadingStrategy"/>
                    </property>
                </bean>

            </list>
        </constructor-arg>
    </bean>

    <!-- １つのファイル -->
    <bean id="simpleConfiguration" class="org.apache.commons.configuration.PropertiesConfiguration">
        <constructor-arg type="java.net.URL" value="classpath:prop/config-sample1.properties" />
        <property name="reloadingStrategy">
            <bean class="org.apache.commons.configuration.reloading.FileChangedReloadingStrategy"/>
        </property>
    </bean>
・・・省略
</beans>
```

- 呼び出し方法は、「14.3.1 「CommonsConfigurationFactoryBean」を使用する」と同じです。
  Bean 名称が compositeConfigutation と名前を変更している場合、name 属性で指定します。

14.4. JSP のページディレクティブの web.xml での一括指定

- JSP のページディレクティブで指定していた設定値を、web.xml 内にて、一括で指定できます。
- <url-pattern>タグにて、Spring MVC 管理の URL(@RequestMapping で指定した URL)を指定すると、動作しないため注意が必要です。
  - Spring MVC 管理の JSP に適用したい場合は、<url-pattern>を指定しないようにします。

【JSP の指定】

```
<%@ page language="java"
 isELIgnored="false"
 deferredSyntaxAllowedAsLiteral="false"
 trimDirectiveWhitespaces="true"
 pageEncoding="UTF-8"%>
```

【web.xml の指定】

```
<web-app>
・・・省略
    <jsp-config>
        <jsp-property-group>
            <!--<url-pattern>*</url-pattern>
            <url-pattern>/index.jsp</url-pattern>-->
            <el-ignored>false</el-ignored>
            <page-encoding>UTF-8</page-encoding>
            <scripting-invalid>false</scripting-invalid>
            <deferred-syntax-allowed-as-literal>false</deferred-syntax-allowed-as-literal>
            <trim-directive-whitespaces>true</trim-directive-whitespaces>
        </jsp-property-group>
    </jsp-config>
・・・省略
</web-app>
```

<url-pattern>を使用すると、Spring MVC 管理の JSP でエラーが発生するので、使用しない。

表 14.1　JSP の設定値

No.	JSP の属性値	web.xml の要素	初期値	説明
1	isELIgnored	<el-ignored>	true	EL 式（${式}）を無視するか制御します。
2	pageEncoding	<page-encoding>	—	JSP の文字コードを指定します。
3	—	<scripting-invalid>	true	スクリプト記述要素を無効にするかどうかを制御する。
4	deferredSyntaxAllowedAsLiteral	<deferred-syntax-allowed-as-literal>	false	Unified EL 式の「#{」を文字列として利用するかどうか。
5	trimDirectiveWhitespaces	<trim-directive-whitespaces>	false	空白を削除します。

15. 二重送信のチェック

15.1. JavaScript による確認ダイアログの表示

ボタンを押下した際に、確認ダイアログを表示し、ダブルクリックなどの連打を防止します。この方法は、JavaScript の確認ダイアログを利用し実装します。

また、データを送信後、ブラウザの「戻る」ボタンで画面を戻った場合、再度データを送信することができるので、「15.2　トークンによるチェック」と合わせて使用することをお勧めします。

実装例を図 15.1 に示します。JavaScript「window.confirm()」を使用しダイアログを表示します。ダイアログで「OK」を選択すると Form が submit され、「いいえ」を選択すると Form は submit されません。

```
<form:form commandName="command" action="${appUrl}/sample/complete.html" method="post">
　　・・・省略
<input type="submit" name="complete" value="登録" onClick="return window.confirm('登録してもよろしいですか？
') ? true : false;">
　　・・・省略
</form>
```

図 15.1　JavaScript による確認ダイアログの表示(JSP)

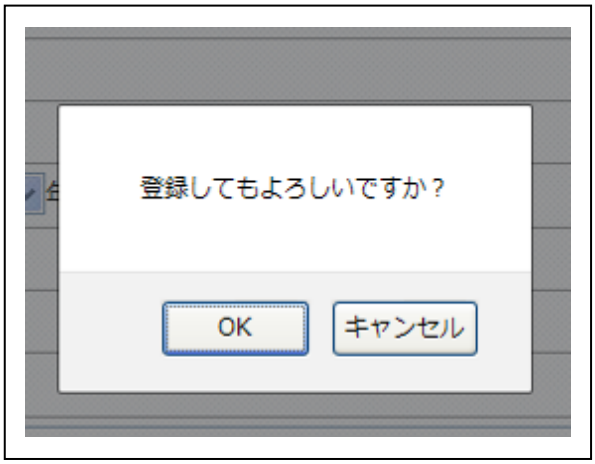


図 15.2　JavaScript による確認ダイアログの表示(Web ブラウザ)

15.2. トークンによるチェック

Struts で採用されているトークンによるチェックの方式を説明します。Spring MVC は、トークン機能はないので独自実装になります。Struts のクラス「org.apache.struts.util.TokenProcessor」と同じものを実装します。

トークン ID を HttpSession 上と画面の Form の値（<input type="hidden">）の 2 つに登録し、それを比較することでチェックします。Submit 後、ブラウザの「戻る」で画面を戻り、再度 submit した場合などのチェックが可能です。

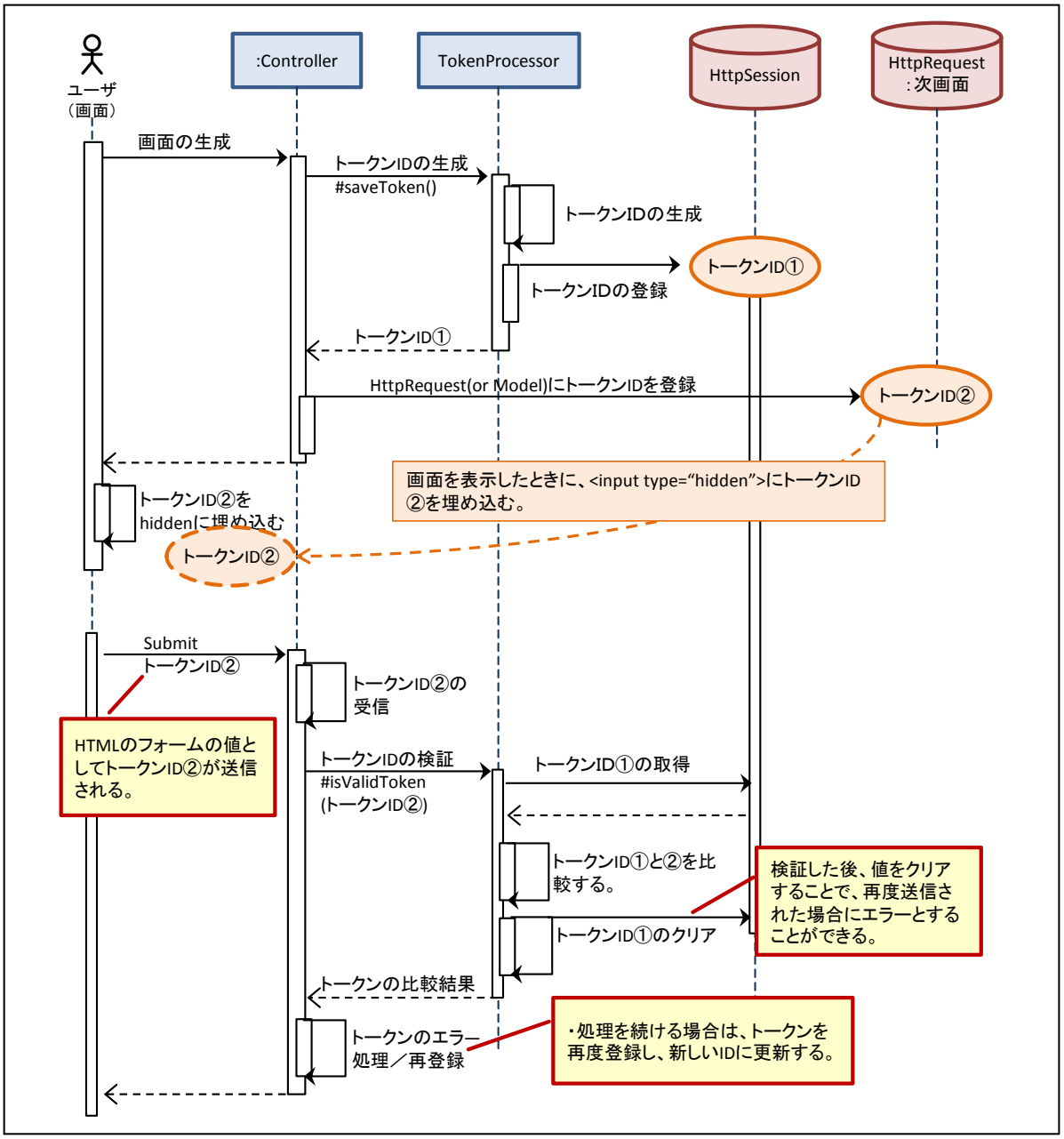


図 15.3　トークンによる 2 重送信のチェック処理の流れ

15.2.1. TokenProcessor の実装

トークンを処理するクラス TokenProcessor の例を表 15.1、図 15.4 に示す。セッションから取得する際には、Spring のクラス「org.springframework.web.context.request.WebRequest」を使用しているが、ServletAPI の「HttpSession」でも特にかまわない。

表 15.1　トークン処理クラス(TokenProcessor)の説明

No.	メソッド	説明
1	getInstance()	自身のクラスのインスタンスを取得する。同期をとるため、singleton にする。
2	saveToken()	トークン ID を新しく生成し、それをセッションに登録する。 トークン ID は、ランダムな文字列として、UUID を使用する。
3	resetToken()	セッションに登録されているトークン ID を削除する。
4	isTokenValid()	引数で渡したトークン ID とセッション上にあるトークン ID を比較する。 比較後は、セッション上のトークン ID を削除する。

```
/**
 * トークンの処理を行うクラス。
 */
public final class TokenProcessor {

    /** セッションに登録するトークンのキー名 */
    static final String SESSION_TOKEN_KEY = TokenProcessor.class.getName() + "@token";

    /** 自身のインスタンス */
    private static TokenProcessor instance = new TokenProcessor();

    public static TokenProcessor getInstance() {
        return instance;
    }

    /**
     * 現在のセッション上のトークンの値を取得する。
     * @param request
     * @return 見つからない場合は null を返す。
     */
    public String getCurrentToken(WebRequest request) {
        return (String) request.getAttribute(SESSION_TOKEN_KEY, RequestAttributes.SCOPE_SESSION);

    }

    /**
     * トークンのチェックを行う。
     * @param request
     * @param requestToken 比較対象のトークンの値
     * @param reset true: トークンのチェック後、セッションからトークンを駆除する。
     * @return true: トークンが等しい。
     */
    public synchronized boolean isTokenValid(WebRequest request, final String requestToken, boolean reset) {

        if(StringUtils.isEmpty(requestToken)) {
```

```
            return false;
        }

        final String sessionToken = getCurrentToken(request);
        if(StringUtils.isEmpty(sessionToken)) {
            return false;
        }
        boolean result = StringUtils.equals(sessionToken, requestToken);

        if(reset) {
            resetToken(request);
        }

        return result;

    }

    /**
     * トークンのチェックを行う。
     * <p>チェック語、トークンをセッションから削除する。
     * @param request
     * @param requestToken 比較対象のトークンの値
     * @return true: トークンが等しい。
     */
    public boolean isTokenValid(WebRequest request, final String requestToken) {
        return isTokenValid(request, requestToken, true);

    }

    /**
     * トークンをリセットする。
     * <p>セッションからトークン情報を削除する。
     * @param request
     */
    public void resetToken(WebRequest request) {
        request.removeAttribute(SESSION_TOKEN_KEY, RequestAttributes.SCOPE_SESSION);
    }

    /**
     * トークンを新しくセッションに保存する。
     * @param request
     * @return 新しいトークン ID
     */
    public synchronized String saveToken(WebRequest request) {

        final String value = UUID.randomUUID().toString();
        request.setAttribute(SESSION_TOKEN_KEY, value, RequestAttributes.SCOPE_SESSION);

        return value;

    }

}
```

図 15.4　トークンの処理クラス(TokenProcessor)

15.2.2. トークンによるチェック

Controller でトークンを使った 2 重送信のチェック例を図 15.5、図 15.6 に示す。Controller 側で Model に「token」という名称で登録した値を、JSP 側で hidden の値として「${token}」を設定している。

```
@Controller
@RequestMapping("/sample")
public class SampleController {

    /** 初期化 */
    @RequestMapping(method={RequestMethod.POST, RequestMethod.GET})
    public String initNew(WebRequest request, ModelMap model) {

        // トークンの設定
        TokenProcessor tokenProcessor = TokenProcessor.getInstance();
        String newToken = tokenProcessor.saveToken(request);
        model.addAttribute("token", newToken);

        return "/book/edit";
    }

    /** 編集完了 */
    @RequestMapping(method={RequestMethod.POST})
    public ModelAndView complete(WebRequest request, @RequestParam String token,
                                 @ModelAttribute SampleCommand command,
            BindingResult result) throws TokenException {

        // トークンのチェック
        TokenProcessor tokenProcessor = TokenProcessor.getInstance();
        if(!tokenProcessor.isTokenValid(request, token)) {
            throw new TokenException();
        }

        if(result.hasErrors()) {
            ModelAndView mav = new ModelAndView("/sample");
            mav.getModel().putAll(result.getModel());

            // トークンの更新
            String newToken = tokenProcessor.saveToken(request);
            mav.getModel().put("token", newToken);

            return mav;
        }

        //TODO: FS 層の呼び出し

        ModelAndView mav = new ModelAndView("forward:/sample/search.html");

        //トークンのリセット
        tokenProcessor.resetToken(request);

        return mav;
    }
}
```

・Form から submit されたトークン ID を取得するする。
・Model からも取得できるが、処理しやすいように「@RequestPraram」で取得する。

図 15.5　トークンを使用した 2 重送信のチェック(Controller 側)

```
<form:form commandName="command" action="${appUrl}/sample/complete.html" method="post">
      ・・・省略
      <%--  トークンの hidden への埋め込み  --%>
      <input type="hidden" name="token" value="${token}">
      ・・・省略
</form>
```

トークン ID の埋め込み。

図 15.6　トークンを使用した 2 重送信のチェック(JSP 側)

16. Controller で DB トランザクションを制御する(:TODO)

DB トランザクションは、通常 Service 単位で制御すべきで、Controller では制御するべきではありません。しかし、エラーメッセージの作成などは Controller 側で行うために、Service のメソッドに Presentation のソースを含めたくない場合があります。

アノテーション「@Transational」をコントローラのメソッドに付与し、明示的なトランザクションを設定っする。