Spring MVC

～Spring Framework 4.0～

変更点

【2013.12.13.01 版】

Tatsuo TSUCHIE 著

Project *Green-Day*

【変更履歴】

No.	版	変更箇所	変更内容
1	2013.12.13	－	新規作成。
2			
3			

【目次】

1. Spring4.0 の変更点

Spring4.0 が、2013 年 12 月にリリースされました。2009 年の Spring3.0 以来のメジャーバージョンアップです。

今回の変更は、Java8 のサポート、Spring MVC 関連では、HTML5/WebSocket 対応が主です。また、Spring3.2 に引き続き、Servlet3 の機能対応や、Ajax、REST サービス向けの機能が追加、改善されています。

Spring4.0 が、**Java 8 の完全サポート**の初めてのメジャーリリースとなります。また、前提として、**Java 6 以上が必要**となります。Java5 は開発終了に伴い非サポートとなりました。

1.1. Spring4.0 を利用するに当たり

削除されたパッケージ、非推奨となった前提 PP があるため、それぞれ対応する機能に変更してください。パッケージやライブラリなどの変更内容は、「1.2　機能追加／前提 PP 変更に伴うパッケージ／メソッドの」を参照してください。

1.1.1. ドキュメントなどの公開先

【ドキュメント】

- 1 ページの HTML にまとめられたドキュメント

http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/

- 章・節単位に HTML ファイルが分割されたドキュメント

http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/html/

- PDF 形式のドキュメント

http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf

【Javadoc（API 仕様）】

http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/index.html

【変更されたクラス／パッケージ一覧】

http://docs.spring.io/spring-framework/docs/3.2.4.RELEASE_to_4.0.0.RELEASE/

1.1.2. リリース予定

- Spring4.0 については、基本的に 2 か月間隔にバグフィックス版がリリースされる予定です。情報は、JIRA の SpringFramework のサイトを参照してください。
- 不良情報が記載されているため、現在使用しているバージョンで気になる動作をしている部分があれば、参照してみるのもよいと思います。
- Java8（Open JDK）の正式リリースが 2014 年 3 月 18 日のため、Java8 正式対応は Spring 4.0.2 からとなります。

【JIRA による Spring のバグトラッカー】

- 未リリースの情報
  https://jira.springsource.org/browse/SPR

- 全バージョンの情報
  https://jira.springsource.org/browse/SPR?selectedTab=com.atlassian.jira.plugin.system.project:versions-panel&subset=-1

表 1.1　Spring4.x のリリース予定

No.	バージョン	リリース予定日	備考
1	4.0.0	2013 年 12 月 12 日	メジャーバージョンアップ版
2	4.0.1	2014 年 1 月 23 日	バグフィックス版
3	4.0.2	2014 年 3 月 20 日	バグフィックス版 （Open JDK 8 GA サポート版）
5	4.0.3	2014 年 5 月 27 日	バグフィックス版
6	4.1.0（仮）	2014 年 12 月	リビジョンアップ版

1.1.3. ライブラリの変更（:TODO）

Spring4.0 を利用する際の pom.xml や定義ファイルの記述の移行法方法を説明します。

TODO: ライブラリ一覧にも追加する。

- Spring 本体の各種のライブラリのバージョンを「4.0.0.RELEASE」に設定します。

TODO：対応ライブラリが比較的新しいバージョンのみに対応なったことを明記する。
WebSocket の場合は、モジュールが別になったことを明記する。

```
<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-web</artifactId>
            <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${spring.version}</version>
        <scope>test</scope>
    </dependency>

    <!-- spring-asm の定義は必要ありません
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>${spring.version}</version>
    </dependency>
    -->

    <!-- AspectJ -->
    <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>
    <!-- CGLIB は必要ありません。
    <dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib-nodep</artifactId>
        <version>2.2.2</version>
```

・「3.2.1.RELEASE」を利用する場合、spring-web を今まで記述していない場合は、追加します。
・「3.2.2.RELEASE」以降は記述しなくても問題ありません。

「spring-asm」は、「spring-core」に取り込まれたため、必要ありません。

CGLIB は、CGLIB3.0 として「spring-core」に取り込まれたため、必要ありません。

```
    </dependency>
    -->
    <!-- JUnit -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
        <scope>test</scope>
    </dependency>

</dependencies>
<properties>
    <spring.version>4.0.0.RELEASE</spring.version>
    <aspectJ.version>1.7.1</aspectJ.version>
    <junit.version>4.11</junit.version>
</properties>
```

AspectJ は、既存の 1.6.x でも構いません。

1.1.4. 定義ファイルの変更（:TODO）

- SpringBean の XML の定義ファイル内のスキーマの指定を変更します。
  - 「spring-xxx-4.0.xsd」というようにバージョンを指定します。
- XML の Spring4.0 の新しい記述を利用しない場合は、Spring3.x のままでも問題ありません。

```
<?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-4.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

・・・（省略）・・・
</beans>
```

1.2. 機能追加／前提 PP 変更に伴うパッケージ／メソッドの変更

Spring 3.x で非推奨（deprecated）となったパッケージや、メソッドが削除されました。また、機能追加に伴いパッケージも追加されました。詳細は下記 URL を参照してください。

- API の変更一覧（Spring4.0 情報）
  http://docs.spring.io/spring-framework/docs/3.2.4.RELEASE_to_4.0.0.RELEASE/
- 非推奨パッケージ情報（Spring3.2 情報）
  http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/migration-3.2.html#migration-3.2-removals-and-deprecations

1.2.1. 追加されたパッケージ

基本的に、Java8 サポート、WebSocket サポートのために追加されたものになります。

表 1.2　パッケージの変更点の概要（追加パッケージ）

No.	パッケージ	変更内容
1	org.springframework. beans.factory.groovy	Groovy 言語による、SpringBean 定義の読み込みをサポートのために追加されました。 ・詳細は、「1.5　Groovy 言語による Spring Bean 定義のサポート」を参照してください。
2	org.springframework. cache.guava	キャッシュ抽象機能（Cache Abstraction）に、「Google Guava」の Cache クラスに対応しました。
3	org.springframework. format.datetime.standard	Java8 の「Date And Time API」対応のために追加されました。 ・パーサ、フォーマッターが追加されています。 ・詳細は、「3.2　Date And Time API（JSR-310）」を参照してください。
4	org.springframework. messaging	WebSocket のためのメッセージング機能のために追加されました。
5	org.springframework. util.concurrent	Java5 から追加された「java.util.concurrent.Future」を拡張した機能のために追加されました。 ・非同期の REST クライアント「AsyncRestTemplate」などで使用します。詳細は、「1.7.3　非同期の REST クライアント（AsyncRestTemplate）のサポート」を参照してください。
6	org.springframework. web.socket	WebSocket 関連の機能が追加されました。 ・詳細は、「2　WebSocket」を参照してください。

1.2.2. 削除されたパッケージ

基本的に、古いバージョンの前提 PP が非サポートとなったため削除されました。また、Java 自体へ追加された機能を利用することで、使用されなくなった機能を削除されました。

表 1.3　パッケージの変更点の概要（削除パッケージ）

No.	パッケージ	変更内容
1	org.springframework. core.enums	Java5 の列挙型（Enum）がサポートされる前の列挙型の実装機能は削除されました。
2	org.springframework. ejb.support	EJB2.x 形式の機能は非サポートとなりました。 EJB3.X 以降のアノテーション形式を利用してください。
3	org.springframework. instrument.classloading.oc4j	Oracle 提供の J2EE「OC4J」は、開発終了となったため、そのためのクラスローダは削除されました。 ・OC4J の後継「WebLogic」を使用してください。
4	org.springframework. orm.ibatis	iBATIS はプロジェクトが後継の「MyBatis」に変更され、後継プロジェクトが Spring 連携用のライブラリを提供しているため、削除されました。
5	org.springframework. remoting.jaxrpc	JAX-WS 1.x は非サポートとなったため削除されました。 ・後継の「JAX-WS 2.0/2.1/2.2」用のパッケージ「org.springframework.remoting.jaxws」を使用してください。
6	org.springframework. scheduling.backportconcurrent	Java6 の標準パッケージ「java.util.concurrent」内の機能を使用するようにしたため削除されました。 ・TreadPoolTaskExecutor などは、別パッケージ「org.springframework.scheduling.concurrent」のクラスを使用してください。
7	org.springframework. scheduling.timer	Java5 の標準パッケージ「java.util.concurrent」内の機能で代替できるため削除されました。 ・「ScheduledTimerTask」「TimerTaskExecutor」は、Java5 の「TimerTask」「ExecutorService」を使用してください。
8	org.springframework. test.context.junit38	JUnit3.x は非サポートとなりました。 ・アノテーションベースの JUnit4.x 用の機能を使用してください。
9	org.springframework. web.struts	Struts との連携機能が削除されました。 ・ActionSupport クラスなど削除されたため、Struts1.x は非対応となります。Struts2.x は、Struts が提供している「Spring Plugin」を使用してください。

1.2.3. 前提 PP の変更

サードパーティ製のライブラリに依存していた機能について、サポートのバージョンが変更になりました。

【サポートバージョンの基準】

- Spring4 がサポートするのは、2010 年または、2011 年以降にリリースされたバージョンとする。
- ただし、比較的最近の Spring3 でサポートされた機能に依存するライブラリは対象外とする。
  - Spring3 でサポートされた、Bean Validation の参照実装である「Hibernate Validator4.3+」は例外でサポート継続。
  - Spring3.2 において Jackson2.x がサポートとなったため、Jackson 1.x について、しばらくはサポート継続するが、非推奨（deprecated）とする。今後は、Jackson2.x のサポートに注力する。
- 基本的に、開発終了したバージョンは非サポートとする。
  - Spring3.x までは、Java5 をサポートしていましたが、Java5 は 2009 年に開発終了したため、サポート対象外となりました。

表 1.4　依存ライブラリのバージョン

No.	ライブラリ	Spring3 以前	Spring4
1	Java	5.x、6.x、7.x	6.x、7.x、8.x
2	Hibernate（OR マッパー）	3.x、4.x ※4.x は Spring3.1 からサポート。	3.6+、4.x
3	EhCache（データキャッシュ）	－（明示なし） ※Spring3.1 からサポート。	2.1+
4	Quartz（スケジューリング）	1.x	1.8+
	Joda-Time（日時ユーティリティ）	－（明示なし）	2.0+
5	Bean Validation	1.0(JSR-303) ※Spring3.0 からサポート。	1.0(JSR-303)、1.1(JSR-349)
6	Hibernate Validator（Bean Validation の参照実装）	4.x	4.3+、5.0 ※5.0 は、Bean Validation1.1 の参照実装。
7	Jackson（JSON データ処理）	1.8、1.9、2.x ※2.x は、Spring3.2 からサポート。	1.8、1.9、2.0+ ※ただし、1.x は今後非サポート。

1.2.4. 削除された機能／クラス（Spring MVC 関連）（:TODO）

Spring MVC 関連は、Spring3.0～3.2 で毎回変更があったため、そのたびに非推奨となったクラスがあり、Spring4 のリリースに伴いそれらは削除されました。

【非推奨リスト】

詳細は、Spring3.2 の Javadoc の非推奨リストに記載されています。

- Spring3.2 の Javadoc の非推奨リスト
  http://docs.spring.io/spring-framework/docs/3.2.0.RELEASE/javadoc-api/deprecated-list.html

【削除された機能】

- Spring2.x 以前のアノテーションを使わない方式の Controller 関連のクラスは削除されました。
  - 「SimpleFormController」など、「AbstractCommandController」を継承しているクラス。
- 各種機能のアノテーションを処理する代替となるクラスが、Spring3.1 で追加されたため、古いものは削除されました。主に、アノテーション「@RequestMapping」を処理するためのクラスです。
  - 削除対象のこのクラスも、Spring2.5 時代のアノテーション機能の黎明期に追加されたクラスのため、削除されました。
  - 対応するクラスは、「表 1.5」を参照してください。

表 1.5　代替クラスが追加された SpringMVC のクラス

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

※1　パッケージは、「org.springframework.web.servlet.mvc.method.annotation」。

1.3. Java8 対応（:TODO）

Java8 への対応を行いました。

- Lambda 式や、Spring の Callback 用のインタフェースにおけるメソッド参照（Method Reference）を使用可能です。
  - 詳細は「3.3　Project Lambda（JSR-335）」を参照。
- java.time（Java Date And Time API）に対応しました。
  - 詳細は「3.2　Date And Time API（JSR-310）」を参照。
- アノテーションの重複指定可能な「@Repeatable」のようなアノテーションへの対応をしました。
  - http://d.hatena.ne.jp/tomoTaka/20130418/1366239630

Java6、Java7 への互換性は担保しています。

- Java6 については、基本的にフルサポートしていますが、Spring4 から追加された新機能については、基本的に Java7、8 をサポート対象とする方針です。

1.4. Java EE6／7（:TODO）

Java EE6

JPA2.0、Servlet3.0

1.5. Groovy 言語による Spring Bean 定義のサポート（:TODO）

1.6. Core Container の機能の改良（:TODO）

コア機能である、コンテな機能も改良されています。主に、Spring3.0 から追加された、アノテーションベースのコンテナ定義（JavaConfig）について、機能が追加されました。

- アノテーション「@Lazy」が、「@Bean」と同様に、インジェクションする箇所に定義可能となりました。
- 説明（コメント）用のアノテーション「@Description」が追加されました。アノテーションベースのコンテナ定義に使うことで、Javadoc などがなくても、Bean の説明を参照できるようになりました。
  - XML 定義の場合は、<!-- -->でコメントを書けばおいですが、アノテーションベースの場合、
- アノテーション「@Conditional」を使用することで、Spring Bean の生成のための条件を切り替えることができます。@Profile と異なるのは、切り替えの条件を自分でハードコーディングできることです。
  - 参考「http://www.javacodegeeks.com/2013/10/spring-4-conditional.html」
- CGLIB を利用した AOP ににおいて、デフォルトコンストラクタの定義は不必要になりました。
- タイムゾーンをサポートされました。LocaleContext の実装クラスが追加されました。

1.7. Web 機能の改善（:TODO）

1.7.1. アノテーション「@RestController」の追加

クラスに付与するコントローラのアノテーション「@Controller」の代わりに、「@RestController」を付与すると、「@RequestMapping」を設定したメソッドは自動的に「@RequstBody」を付与したことと同じ意味になります。すなわち、アノテーション「@RequestBody」を省略できます。

- Spring3.2 で追加となった、@ControllerAdvice による各処理の共通部分の判断条件となります。
  - 詳細は、「1.7.2　アノテーション「@ControllerAdvice」の改善」を参照してください。
- 通常 REST 用の機能を実装する際は、クラス単位で処理をまとめ、戻り値が通常の View と REST 用の XML/JSON 形式を返すメソッドが混在することは少ないため、アノテーションの付与漏れやアノテーション地獄に陥ることによるソースコードの可読性の低下を改善することができます。
  - ただし、@RestController を付与したクラス内に、Model、ModelAndView を返却するメソッドも混在させることができます。
- 「@RestContrller」を付与することで、そのコントローラは、REST 用と一目で理解することに役経てることができます。

【@Controller を用いた場合】

```
@Controller
public class SampleController {

    @RequstMapping("/sample/load/{id}")
    @ResponseBody
    public Access load(@PathVariable String id) {
    }

    @RequstMapping("/sample/get/{id}")
    public ModelAndView get(@PathVariable String id) {
    }
}
```

【@RestController を用いた場合】

```
@RestController
public class SampleController {

    // @ResponseBody を省略できる
    @RequstMapping("/sample/load/{id}")
    public Access load(@PathVariable String id) {
        //
    }

    // 戻り値の型が、Model、ModelAndView などの場合、自動的に判別され、JSP などの通常ページに遷移する。
    @RequstMapping("/sample/get/{id}")
    public ModelAndView get(@PathVariable String id) {
        //
    }
}
```

1.7.2. アノテーション「@ControllerAdvice」の改善

Spring3.2 で追加されたアノテーション「@ControllerAdvice」は、いわゆる Controller の特定メソッドに対して、AOP のように処理を追加できるものでした。しかし、**Spring3.2 までは全ての Controller クラスに対して作用するため今一使い勝手が悪い**ものでした。

Spring4.0 では、AOP での所謂ポイントカットの条件として、**適用するクラス、パッケージが指定**できるようになりました。

表 1.6　アノテーション「@ControllerAdvice」の要素

No.	型	要素	説明
1	Class<? extends Annotation>[]	Annotations	・指定したアノテーションを付与された Controller のみ適用範囲とする。クラスに付与されたアノテーションを対象とする。 ・「@RestController」のように限られた用途のクラスに適用したい場合に指定する。 ・独自のアノテーションも指定可能。 ・複数指定可能。オプション。
2	Class<?>[]	assinableTypes	・指定した Controller クラス（インタフェース含む）及びそのクラスを継承／実装しているクラスが対象となる。 ・Controller の基底クラス、インタフェースを指定するのが一般的。 ・複数指定可能。オプション。
3	Class<?>[]	basePackageClasses	・指定したクラスに属するパッケージと同じ Controller に対して適用する。 ・複数指定可能。オプション。
4	String[]	basePackages	・　指定したパッケージに属する Controller に対して適用する。 ・複数数指定可能。・オプション。
5	String[]	value	・要素名を指定しない場合の値。 ・要素「basePackages」を指定した場合と同じ意味。 ・複数数指定可能。・オプション。

【@ControllerAdvice による適用条件の指定】

- アノテーション「@ControllerAdvice」の要素に何も指定しないと、全ての Controller クラスに適用されます。
- 複数の要素を指定した場合、OR 条件となり、それぞれの条件に一致する Controller に対して適用されます。

```
// 全ての Controller に適用する（Spring3.2 以前）
@ControllerAdvice
public class AllAdvice {}

// アノテーション「@RestController」が付与された Controller にのみ適用する
@ControllerAdvice(annotations = RestController.class)
public class AnnotationAdvice {}

// 独自のアノテーション「@MyAnnotation」が付与された Controller にのみ適用する
@ControllerAdvice(annotations = MyAnnoattion.class)
public class AnnotationAdvice {}

// 指定したパッケージに属する全ての Controller に適用する。
@ControllerAdvice("org.example.controllers")
public class BasePackageAdvice {}

// 指定したクラス／インタフェースを継承／実装している全ての Controller に適用する。
@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.class})
public class AssignableTypesAdvice {}

// パッケージ指定とクラス指定の組合せ（それぞれの条件に一致する Controoler に適用される。）
@ControllerAdvice(basePackages={org.example.controllers"}, assignableTypes = {ControllerInterface.class, AbstractController.class})public class ComplexAdvice {}
```

1.7.2.1. アノテーション「@ControllerAdvice」とは

アノテーション「@ControllerAdvice」は、Component-Scan 対象のクラス用アノテーションです。Controller 内で定義していた「@ExceptionHandler」「@InitBinder」「@ModelAttribute」が付与されたメソッドを<u>に対して AOP のように処理を追加</u>します。

すなわち、Controller のメソッドのポイントカットを「@ExceptionHandler」「@InitBinder」「@ModelAttribute」として、そのポイントカットに対するアドバイスとしての処理を「@ControllerAdvice」が付与したクラスで処理するためのものです。

@ExceptionHandler や@InitBinder は、Controller ごとに定義していますが、共通する部分があると思います。それらを AOP の考え方により共通部分を抽出して Advice として部品化して、それぞれの Controller に適用するためのものが@ControllerAdvice になります。

```
@ControllerAdvice({"sample.web.common", "sample.web.search"})
public class ControllerAdviceHandler {

    @Resource
    private Validator validator;

    @InitBinder("sampleModel")
    protected void initBinder(WebDataBinder binder) {
        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;
    }

    // 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;
    }
}
```

Controller の@ RequestMapping を付与したメソッドの呼び出し時に呼び出される。

Controller の@RequestMapping を付与したメソッドの呼び出し時に呼び出される。

Controller の@RequestMapping を付与したメソッドの呼び出し時に例外が発生した場合に呼び出される。

1.7.2.2.　ResponseEntityExceptionHandler の使用（REST 用 Controller の例外処理）

Spring3.2 で追加された、REST 用のメソッドの例外処理用の ControllerAdvice の基底クラスとして、「ResponseEntityExceptionHandler」がありました。しかし、@ControllerAdvice が全ての Controller に適用されるという仕様のため、今までは出番がなかった思います。Spring4.0 から条件が指定可能となったため、アノテーション「@RestController」を条件に指定するとよいと思います。

既存の例外処理を行う DefaultHandlerExceptionResolver は、戻り値は“ModelAndView”であり JSP などのエラー画面を表示するために使用していました。REST サービスなどは、HTTP ステータスコードを返す必要があったため不向きでした。

それに対しては、**「ResponseEntityExceptionHandler」は、戻り値は“ResponseEntity”**であり、任意の HTTP ステータスコードを返すのに向いています。また、DefaultHandlerExceptionResolver と同様に、Spring MVC がスローする例外とそれに対して返すレスポンスのステータスコードが登録されています（表 1.7）。

```
// アノテーション「@RestController」が付与された REST 用の Controller にのみ適用する
@ControllerAdvice(annotations = RestController.class)
public class RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler {

    // 新しい例外に対する処理を追加
    @ExceptionHandler(value = { IllegalArgumentException.class, IllegalStateException.class })
    protected ResponseEntity<Object> handleConflict(RuntimeException ex, WebRequest request) {
        String bodyOfResponse = "This should be application specific";

        return handleExceptionInternal(ex, bodyOfResponse,
          new HttpHeaders(), HttpStatus.CONFLICT, request);
    }

    // 既存の例外処理の振る舞いを変更する
    @Override
    protected ResponseEntity<Object> handleNoSuchRequestHandlingMethod(
NoSuchRequestHandlingMethodException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        //TODO:
        return super.handleNoSuchRequestHandlingMethod(ex, headers, status, request);
    }
}
```

表 1.7　ResponseEntityExceptionHandler に登録されている例外と返却される HTTP ステータスコード

No.	例外	HTTP ステータスコード
1	BindException	400 (Bad Request)
2	ConversionNotSupportedException	500 (Internal Server Error)
3	HttpMediaTypeNotAcceptableException	406 (Not Acceptable)
4	HttpMediaTypeNotSupportedException	415 (Unsupported Media Type)
5	HttpMessageNotReadableException	400 (Bad Request)
6	HttpMessageNotWritableException	500 (Internal Server Error)
7	HttpRequestMethodNotSupportedException	405 (Method Not Allowed)
8	MethodArgumentNotValidException	400 (Bad Request)
9	MissingServletRequestParameterException	400 (Bad Request)
10	MissingServletRequestPartException	400 (Bad Request)
11	NoSuchRequestHandlingMethodException	404 (Not Found)
12	TypeMismatchException	400 (Bad Request)

1.7.3. 非同期の REST クライアント（AsyncRestTemplate）のサポート

- AsyncRestTemplate は、RestTemplate とよく似ているが、戻り値が「ListenableFuture」オブジェクトとなる。「ListenableFuture」となるとで、処理完了時のイベント（Callback）を設定することができる。
  - RestTemplate の場合、正常／異常時の処理のイベント（Callback）は共通で 1 つのみ設定可能だが、AsyncRestTemplate の場合、1 つずつの処理それぞれに対 s 知恵（リクエスト）に対して設定可能。
- AsyncRestTemlplate の非同期アクセスのスレッドは、デフォルトでは「SimpleAsyncTaskExecutor」が使用される。スレッド処理のインスタンスは、コンストラクタで指定して切り替える。
  - 大量にリクエストを行う場合、「ThreadPoolTaskExecutor」のようにスレッドをプーリングするようなクラスを使用すべきです。理由として、同時アクセスするスレッド件数に上限値が決められ、リクエストを受付するサーバに対して負荷をかけないようにすることができる。
  - 例えば、10 秒処理がかかる処理を for 文などでループさせ 200 回実行した場合、サーバから見ると同時に 200 件アクセスされたこととなり、Servlet コンテナの標準のスレッド数の上限値 200 件に容易に達してしまう。ThreadPoolTaskExecutor を利用し最大スレッド件数を 10 件とすると、サーバに対しては最大同時に 10 件までしか実行されない。その他の 190 件は待ち行列にためられるが、多くの場合、プーリングした方が早く終わることが多い。
  - ただし、AsyncRestTemplate に設定可能な TaskExecutor は、インタフェース「AsyncListenableTaskExecutor」を実装している必要がありますが、**現状、「SimpleAsyncTaskExecutor」しか存在しない**ため、「1.7.3.1　AsyncThreadPoolTaskExecutor の作成」に示すように自作します。

【Spring3.2 までの非同期実装：RestTemplate を使用】

```
RestTemplate restTemplate = ...;
// async call
Future<ResponseEntity<String>> futureEntity = template.getForEntity(
    "http://example.com/hotels/{hotel}/bookings/{booking}", String.class, "42", "21");

// get the concrete result - synchronous call
ResponseEntity<String> entity = futureEntity.get();
```

【Spring4.0 の非同期実装：AsyncRestTemplate】

```
AsyncRestTemplate restTemplate = ...;
ListenableFuture<ResponseEntity<String>> futureEntity = template.getForEntity(
    "http://example.com/hotels/{hotel}/bookings/{booking}", String.class, "42", "21");

// register a callback
futureEntity.addCallback(new ListenableFutureCallback<ResponseEntity<String>>() {
    @Override
    public void onSuccess(ResponseEntity<String> entity) {
        //...
    }
```

```
    @Override
    public void onFailure(Throwable t) {
        //...
    }
});
```

1.7.3.1.　AsyncThreadPoolTaskExecutor の作成(:TODO)

【AsyncThreadPoolTaskExecutor の作成】

- インタフェース「AsyncListenableTaskExecutor」を実装した、「ThreadPoolTaskExecutor」を作成します。
- ThreadPoolTaskExecutor には、「ListenableFuture」を返却するメソッドのみ存在しないため、実装します。

```
import java.util.concurrent.Callable;

import org.springframework.core.task.AsyncListenableTaskExecutor;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
import org.springframework.util.concurrent.ListenableFuture;
import org.springframework.util.concurrent.ListenableFutureTask;


public class AsyncThreadPoolTaskExecutor
        extends ThreadPoolTaskExecutor implements AsyncListenableTaskExecutor {

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

    @Override
    public ListenableFuture<?> submitListenable(final Runnable task) {
        final ListenableFutureTask<Object> future = new ListenableFutureTask<Object>(task, null);
        execute(future);
        return future;
    }

    @Override
    public <T> ListenableFuture<T> submitListenable(final Callable<T> task) {
        final ListenableFutureTask<T> future = new ListenableFutureTask<T>(task);
        execute(future);
        return future;
    }

}
```

1.7.3.2. AsyncRestTemplate のカスタマイズ(:TODO)

1.7.4. TimeZone のサポート(:TODO)

2. WebSocket（:TODO）

2.1. はじめに

WebSocket は、当初 HTML5 の仕様の一部として策定が進められましたが、その後、HTML5 から切り離されて、現在は単独のプロトコルとして規格策定が進められています。WebSocket の歴史などは、次の URL が参考になると思います。「http://gihyo.jp/dev/feature/01/websocket/0001」

比較的新しい技術であるため、サーバ、クライアント環境とも新しいバージョンが必要となります。そのため、業務で利用するとなると、新規リプレース案件以外は導入するのが難しいと思います。ただし、**SockJS を利用することで古いブラウザにも対応**できます（「2.3　SockJS」を参照）。また、スマートフォンなどにおいては買い替えサイクルが短く、比較的環境が新しく WebSocket に対応しているものが多い場合は、導入しやすいと思います。

表 2.1　Web における非同期処理の方式

処理方式	説明
Ajax (Polling)	・クライアントの JavaScript の XMLHttpRequest を利用した処理方式。 ・**クライアントを起点**に処理を非同期で取得する処理（Pull 型）、**サーバ側からはデータを送信不可**。 ・クライアントからサーバ**定期的にリクエスト**を行い（Polling、ポーリング）、非同期処理を実現する。プロトコルとして、HTTP/HTTPS を利用する。 ・**通信回数が多く、ネットワークに負荷**がかかる。 ・Spring MVC では、@ResponseBody／@RuqestBody アノテーションを用いて、REST 処理で実現。Spring3.0 から利用可能。
Comet (Long Polling)	・Ajax の使い方を変更した処理方式で、サーバ側にリクエストして、コネクションを張ったままにしておき、サーバ側から必要となったらレスポンスを返す処理。 ・**サーバ側から、疑似的に任意のタイミングでクライアントにデータを配信**（Push 型）可能。コネクションを張ったままで処理待ちになるため、**AP サーバなどの同時処理スレッド数に上限値に達しやすく**なります。・通信するには**毎回コネクションを確立**する必要がある。 ・Spring MVC では、DefferedResult クラスを用いて実現。Spring3.2 から利用可能。
WebSocket	・Comet の疑似的なサーバプッシュ型の処理ではなく、専用のプロトコルを利用し、**サーバ側とクライアントでの相互にデータの送受信が可能**。 ・専用のプロトコルのため、一度コネクションを確立すれば、AP サーバとクライアントで何度も送受信が可能となり、**負荷が少ない**。 ・コネクションの確立には、HTTP／HTTPS を用いる。

もし、IE8 などの古いブラウザなどで WebSocket の機能を利用したい場合は、SockJS を利用してください。SockJS は、WebSocket をエミュレートする JavaScript ライブラリです。詳細は「2.3 SockJS」を参照してください。

2.1.1. WebSocket を使う場面

WebSocket を利用することで、サーバ側とクライアント側で相互データの送受信（特に、サーバ側⇒クライアント側への Push）が可能となり、完全なリアルタイム処理が実現できるようになりました。

既に、実例はありますが、環境が整っていないなどで、普及はこれからだと思います。

【例 1：サーバのリソース監視】

- サーバ側からリアルタイムにデータを配信することで、リアルタイムにアクセス監視などができるようになります。
- 今までは、数秒～数分毎に定期的にクライアントからデータをリクエストしていた処理が不必要となります。通信回数が減ることで、監視によるリソースの消費が減らせます。

【例 2：クライアント間のコミュニケーション】

- Twitter、LINE などで、サーバ側からのリアルタイム配信の他に、サーバ経由のクライアント同士で、リアルタイムにコミュニケーションが取れるようになります。
  - もっとも簡単な例が、チャットです。

【例 3：リモート操作処理】

- GUI によるリモートデスクトップや、端末エミュレータのようなコマンドでの操作処理ができます。
- 今までは、数秒～数分毎に定期的にクライアントからデータをリクエストしていた処理が不必要となります。
  - 通信回数が減ることで、監視によるリソースの消費が減らせます。
- WebSocket を利用した VNC「ThinVNC」などがあります。

【例 4：UI の振る舞い】

- 進捗情報（プログレスバー）において、サーバからプッシュすることで状況を更新することができます。
  - デザインパターンでいう Observer パターン。
- 今までは、毎秒ごとに定期的にクライアントからデータをリクエストしていた処理が不必要となります
  - 通信回数が減ることで、リソースの消費が減らせます。

2.2. WebSocket を利用するための準備

2.2.1. WebSocket を利用するための準備「サーバ環境」

2.2.1.1. ネットワーク（Firewall）の設定

- WebSocket は、HTTP と同じ 80 番ポートを使用するため、Firewall による接続許可はあまり必要ないかもしれません。
- 独自のポート番号も使うこともできるため、その場合は、Firewall などで許可設定を行います。

2.2.1.2. 前提 PP の準備「サーバ側」

- Servlet3 が利用可能な Servlet コンテナが必要です。
  - Spring が保証している各種 AP サーバのバージョンは、「Tomcat 7.0.47 以上」「GlassFish4.0 以上」「Jetty 9.0+」となります。
- Java のバージョンを 7 以上にします。
- コネクションの確立は、HTTP の GET メソッドで行うため、GET によるレクエストを許可しておく必要があります。

2.2.1.3. ライブラリの準備（pom.xml の変更）

- Spring の WebSocket の機能は、別モジュールとしてライブラリが「spring-websocket」として分離されているため、依存関係を追加します。
- WebSocket のサブプロトコルである「STOMP」を利用するには、ライブラリ「spring-message」を追加します。

pom.xml

2.2.1.4. web.xml の変更（Servlet3 対応）

Servlet3 対応のために「web.xml」ファイルも DTD など変更する必要があります。

2.2.1. WebSocket を利用するための準備「クライアント環境」

【前提 PP の準備】

- WebSocket は比較的新しい技術のため、古い Web ブラウザは対応していないものがあります。
- 詳細は、下記の URL を参照してください。各 Web ブラウザのどのバージョンから WebSocket が利用できるかわかります。
  - http://caniuse.com/websockets
- Android の場合、組み込みの Web ブラウザを利用する場合、2014 年 1 月現在、Android4.4 を搭載している機種はリリースされていないため、Firefox、Chrome などを利用する必要があります。
- iOS の場合、Safari 以外の Web ブラウザとして Chrome などがあります。しかし、Apple の規約により内部のエンジンは Safari と同じ「WebKit」しか利用できないため、動作は Safari と同様です。

表 2.2　WebSocket が利用可能な主な Web ブラウザ（完全サポート）

No.	Web ブラウザ	バージョン	OS
1	Internet Exproler(PC)	10.0+	Windows7/8
2	Mozilla Firefox(PC)	20.0+	Windows、Linux、Mac
3	Google Chrome(PC)	14.0+	Windows、Linux、Mac
4	Safari(PC)	6.0+	Windows、Mac
5	Opera(PC)	11.0+	Windows、Linux、Mac
6	Safari(iOS)	6.0+	iOS 6.0(4.2 は一部サポート)
7	Android Browser (Android の組み込み)	4.4+	Android4.4 (4.4 は 2014 年 1 月現在、未リリース)
8	Mozilla Firefox(Android)	31.0+	Android4.0+
9	Google Chrome(Android)	25.0+	Android4.0+
10	Opera Mobile(Android)	11.0+	Android4.0+
11	Internet Exproler Mobile	10.0+	Windows Phone 8+

2.3. SockJS（:TODO）

WebSocket はクライアントの環境として、新しいものしか対応していません。特に、2014 年 4 月でサポート終了となる Windows XP の IE8 は、しばらくは使われるでしょう。また、Windows Vista/7 での IE9 もしばらくは使用されるしょう。

SockJS は、古い IE6 などの Web ブラウザでも WebSocket と同様の機能を提供するためにするものです。具体的には、WebSocket の API をエミュレートする JavaScript ライブラリです。

2.3.1. SockJS の対応環境「クライアント」

https://github.com/sockjs/sockjs-client

http://www.slideshare.net/ngocdaothanh/sockjs-intro

2.3.2. SockJS を利用するに当たり

WebSocket を利用する際に、クライアント側で JavaScript を記述し、WebSocket の API を呼び出しますが、SockJS はインタフェースがほぼ同じであ（初めの Socket オブジェクトの名称が異なる）るため、新たに覚えることはほぼありません。

サーバ側の Spring 上も、SockJS を利用する際は、通常の WebSocket の利用設定に少しだけ手を加えるだけです。

3. Java 8（:TODO）

3.1. ドキュメントの公開先

Java8 は、2014 年 3 月に正式にリリースされる予定です。Javadoc などのドキュメントの公開は、プレビュー版を参照することになります。（このドキュメントは、2014 年 1 月に作成しています。）

※Java8 の機能一覧

http://openjdk.java.net/projects/jdk8/features

文法の変更、クラスライブラリの追加他、JVM が HotSpot から、HotRockit に変更になりました。JVM の変更により、今までのパフォーマンスチューニング方法も手法が異なってきます。

3.2. Date And Time API（JSR-310）

3.2.1. 仕様

http://etc9.hatenablog.com/entry/2013/09/20/191842

3.2.2. Spring での使用

3.3. Project Lambda（JSR-335）

Lambda 式を利用することで、ソースコードの表記が簡易になります。また、Lambda 式以外の Stream 機能を組み合わせることで、さらにその効果を発揮します。

http://openjdk.java.net/projects/lambda/

・説明（日本語）

http://www.javainthebox.com/2011/12/project-lambda.html

http://waman.hatenablog.com/entry/20130401/1364778128

↓のスライドが分かりやすい

http://cco.hatenablog.jp/entry/2013/12/15/210948

3.3.1. Lambda 式（Lambda Expression）

3.3.2. メソッド参照（Method Reference）

http://etc9.hatenablog.com/entry/2013/09/15/005516

3.3.3. インタフェースのデフォルト実装（Interface Default Implementation）

http://etc9.hatenablog.com/entry/2013/09/15/005516

3.3.4. Stream

Labmda を組み合わせて使う。

http://devlights.hatenablog.com/entry/20130506/p1

3.3.1. Spring での使用

3.4. その他の変更

その他

StringJoiner

http://etc9.hatenablog.com/entry/2013/09/29/223222

https://gist.github.com/ponkotuy/5690470

Google Guava で実装されていたユーティリティ系のクラス String Joiner が

3.5. JVM「HotRockit」

Java8 から、今までの JavaSE の標準であった、JVM の「HotSpot」と、Oracle 製の Java の JVM「JRockit」が統合され、「HotRockit」になりました。

よく言われている違いは、メモリの Permanentr 領域（Perm サイズ）が、Heap 領域に統合され、Perm サイズの枯渇による OutOfMemory が発生しなくなりました。

しかし、その分、パフォーマンスチューニング用のパラメータが増えました。

http://openjdk.java.net/jeps/173

Jav8 の変更一覧（vm の項目を参照）

http://openjdk.java.net/projects/jdk8/features

↓日本語説明

http://cco.hatenablog.jp/entry/2013/12/15/210948

Java6 の HotSpot(CMS)

http://itpro.nikkeibp.co.jp/article/COLUMN/20100208/344358/

・HotRockit について

http://www.slideshare.net/OracleMiddleJP/jee-devreport

↓メモリ構造は↓がよい。

http://openjdk.java.net/projects/jdk8/features

3.6. JavaFX の組み込み

Java7 で本体に組み込まれた JavaFX ですが、実行時のライブラリ「/lib/jfxrt.jar」がオプション扱いだったため、JavaFX のクラス「javafx.application.Application」を継承したクラスを指定しても、そのまま実行できませんでした。

そのため、今まで専用の Ant プラグインでのパッケージングや、「/lib/jfxrt.jar」にクラスパスを通す必要がありました。しかし、Java8 から、そのような面倒なことはなく、単純に実行できるようになりました。

4. Bean Validation 1.1（JSR-349）

Bean Validation 1.1（JSR-349）は、2013 年 4 月 10 日に正式リリースされました。
実際に利用するには、参照実装である「Hibernate Validator 5.x」を利用します。

4.1. ドキュメント／仕様などの公開先

【JSR-349 のドキュメント／仕様書】

- JSR-349 の公式サイト
  http://beanvalidation.org/1.1/
- 仕様書（HTML）
  http://beanvalidation.org/1.1/spec/
- 仕様書（PDF）のダウンロード先
  http://jcp.org/aboutJava/communityprocess/final/jsr349/
- Hibernate Validator による JSR349 のリファレンス
  http://docs.jboss.org/hibernate/beanvalidation/tck/1.1/reference/
- Javadoc
  http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/

【Hibernate Validator のドキュメント／仕様書】

- Hibernate Validator の公式サイト
  http://hibernate.org/validator/
- Hibernate Validator5.0 のリファレンス
  http://docs.jboss.org/hibernate/validator/5.0/reference/en-US/
- Javadoc
  http://docs.jboss.org/hibernate/validator/5.0/api/

4.2. 利用に当たり

4.2.1. Maven の pom.xml の設定

- Bean Validation と Hibernate のバージョンは関係しているため、そろえる必要があります。
  - Hibernate Validator の一方のライブラリの定義のみを記述するだけで、依存関係が Maven により解決されるため通常は問題ありません。
- Bean Validation のバージョンを「ver1.0」⇒「ver1.1」へ変更します。
- Hibernate Validator は、バージョンを「ver4.x」⇒「ver5.x」へ変更します。

【pom.xml 変更前（JSR-303）】

```
<!-- Bean Validation 1.0(JSR-303)系を利用する -->
    <dependencies>
        <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>4.3.1.Final</version>
        </dependency>
    <dependencies>
```

【pom.xml 変更後（JSR-349）】

```
    <!-- Bean Validation 1.1 系を利用する -->
    <dependencies>
        <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
            <version>1.1.0.Final</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
            <version>5.0.2.Final</version>
        </dependency>
    <dependencies>
```

表 4.1　Bean Validation と Hibernate Validator のバージョンの関係

No.	Bean Validation	参照実装「Hibernate Validator」
1	Ver.1.0（JSR-303）	Ver.4.x
2	Ver.1.1（JSR-349）	Ver.5.x

4.2.2. 標準メッセージの変更

- EL 式の対応により、標準のメッセージが一部変更になっているため、「ValidationMessages.properties」などを変更します。
- EL 式については、「4.3.4　メッセージ内の EL 式のサポート」を参照してください。

```
## Bean Validation JSR-303/349 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。

## ------- 変更 -------
#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.DecimalMax.message={value}${inclusive == true ? '以下の' : 'より小さい'}値を入力してください。
javax.validation.constraints.DecimalMin.message={value}${inclusive == true ? '以上の' : 'より大きい'}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。

## -------- 追加 -----
org.hibernate.validator.constraints.br.CNPJ.message=不正な法人税金支払番号（CNPJ）の書式です。
org.hibernate.validator.constraints.br.CPF.message=不正な個人税金支払い番号（CPF）の書式です。
org.hibernate.validator.constraints.br.TituloEleitor.message=不正な ID カードの番号の書式です。
```

4.3. Bean Validation1.1 の追加／変更機能の概要

4.3.1. 依存性の注入（Dependency Injection）

- Bean Validator の各処理の「表 4.2」に示すポイントを拡張することができます。
  - ➢ 実装クラスを独自のものに切り替えることでカスタマイズができます。
- Spring から設定変更する場合、「LocalValidatorFactoryBean」の各プロパティにインジェクションします。

表 4.2　Bean Validator の拡張ポイント

No.	クラス／インタフェース（※1）	説明
1	MessageInterpolator	メッセージ中のパラメータや EL 式を解釈します。
2	TraversableResolver	値の検証時に、プロパティなどにアクセスする際に、値がキャッシュ可能かなどを判断したりします。 JPA などの DB の OR マッピングで利用する場合は、遅延読み込みの処理を行ったりします。
3	ParameterNameProvider	メッセージ中で使用可能なパラメータ名を解決する。 アノテーションを付与したメソッド、コンストラクタから情報を取得する。
4	ConstraintValidatorFactory	アノテーションに対する値の検証の実装クラス（ConstraintValidator を実装したクラス）のインスタンスのファクトリクラス。

※1　パッケージは、「javax.validation」

```
<bean>
    <!-- Spring MVC のアノテーションの有効化 -->
    <mvc:annotation-driven validator="beanValidator"/>

    <!-- Bean Validation -->
    <bean id="beanValidator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
        <!-- メッセージ定義の実体（MessageSource）を指定 -->
        <property name="validationMessageSource">...</property>

        <!-- 「MessageInterpolator」を指定 -->
        <property name="messageInterpolator">...</property>

        <!-- 「TraversableResolver」を指定 -->
        <property name="traversableResolver">...</property>

        <!-- 「constraintValidatorFactory」を指定 -->
        <property name="constraintValidatorFactory">...</property>
    </bean>
</beans>
```

4.3.2. メソッドの Validation のサポート（Method Validation）

メソッドの引数／戻り値、コンストラクタの引数において、Validation が可能（アノテーションが付与可能）となりました。Bean Validation 1.0 では、クラスのフィールドにしか付与できませんでした。

- メソッドとコンストラクタの引数にアノテーションを付与することで、前提条件(precondition）を検証できます。
  - 引数の null チェックなど、メソッド内で実装していた処理を排除でき、ロジックに専念できるようになります。
- メソッドの戻り値にアノテーションを付与することで、事後条件(postcondition)を検証できます。
  - 仕様とは異なる戻り値を返すようなロジックを制限し、その時点で検証することができます。
  - 戻り値のオブジェクトをさらに検証したい場合は、アノテーション「@Valid」を付与します。
- 引数や戻り値の仕様をアノテーション付与するだけで、メソッドのインタフェース設計／仕様を表現することができます。
  - Javadoc にもアノテーションは出力されるため、Javadoc の引数の制約の説明を省くことができ、工数の節約につなげることができます。

```
public class Access {

    @NotNull
    String street;

    @ZipCode
    String zipCode;

}

public class User {

    // フィールド
    @NotNull
    private String name;

    private List<Access> accesses;

    public User() {
    }

    // コンストラクタの引数
    public User(@NotNull String name) {
        this.name = name;
    }

    // メソッドの引数
    // @Valid を付けるとネストして検証する
    public void addAccess(@NotNull @Valid Access access) {
        this.accesses.add(access);
    }

    // メソッドの戻り値
```

```
    // @Valid を付けるとネストして検証する
    @Size(min=1, max=10)
    @Valid
    public List<Access> getAccesses {
        return accesses;
    }
}
```

【注意点】

- この機能は実装より処理が異なるため、値の検証が行われない場合があるため注意が必要です。
- Spring MVC では、次の場合のみ付与したアノテーションが有効になります。
  - フィールドに付与した場合（従来通り）。
  - フィールドに対する Getter メソッドに付与した場合（Bean Validation 1.1）。
- 次の場合は、アノテーションを付与しても値の検証は行われません。
  - コンストラクタ、メソッドの引数（Setter も含む）に付与した場合。
  - フィールドとは異なる Getter に付与した場合。

4.3.3. グループの変換（Group Conversion）

アノテーション「@ConvertGroup」を使用することで、付与済みのネストしたオブジェクトのグループを変更することができます。
この機能によって、アノテーションだけが異なる別なクラスを作る必要がなくなります。

ただし、構造が複雑になるため、グループの定義ミスによる不良が発生した場合など、調査するのが大変になる場合があります。付与するアノテーションの内容が変わる場合は、XML などで定義を切り替えた方がよいと思います。

【グループ情報の変換例】

- User クラスに対して、グループ「Default（グループ指定なし）」を指定し検証を行った場合、Address クラスに対しては、グループ「BasicPostal」として実行します。
- User クラスに対して、グループ「Complete」を指定し検証を行った場合、Address クラスに対しては、グループ「FullPostal」として実行します。

```
// グループ用のインタフェースの定義
public interface Complete extends Default {}
public interface BasicPostal {}
public interface FullPostal extends BasicPostal {}

// JavaBean の定義
public class Address {
    @NotNull(groups=BasicPostal.class)
    String street1;

    String street2;

    @ZipCode(groups=BasicPostal.class)
    String zipCode;

    @CodeChecker(groups=FullPostal.class)
    String doorCode;
}

// JavaBean の定義（グループの変換）
public class User {
    @Valid
    @ConvertGroup.List( {
        @ConvertGroup(from=Default.class, to=BasicPostal.class),
        @ConvertGroup(from=Complete.class, to=FullPostal.class)
    } )
    Set<Address> getAddresses() { [...] }
}
```

4.3.4. メッセージ内の EL 式のサポート

- エラーメッセージ内に、EL 式が定義可能となり、値によって動的にメッセージ内容を変更することができます。
- あまり、複雑な EL 式にすると、不良の原因となるため、等号、不等号などの単純なもののみ使用した方がよいと思います。
- EL 式の文法／機能は、**EL3.0（JSR-341）**を使用しています。
  - 詳細は、「5　Expression Language 3.0（JSR-341）」を参照してください。

4.3.4.1. EL 式について

- 従来のパラメータを出力したい場合は、半角中括弧“**{パラメータ}**”で記述します。
- EL 式を定義したい場合は、先頭に“$”を付けて、“**${EL 式}**”で記述します。
- 記述方式が増えたため、それらをただの文字列として出力したい場合は、バックスラッシュ（¥）でエスケープする必要があります。（表 4.3　メッセージ内のエスケープ対象の文字）

```
# パラメータの出力
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。=

# EL 式の仕様による条件判定
javax.validation.constraints.DecimalMax.message={value}${inclusive == true ? '以下の' : 'より小さい'}値を入力してください。
```

表 4.3　メッセージ内のエスケープ対象の文字

No.	文字	エスケープによる出力	Java/プロパティファイル上の表記（※1,2）
1	{	¥{	¥¥{
2	}	¥}	¥¥}
3	¥	¥¥	¥¥¥¥
4	$	¥$	¥¥$

※1　Java の properties ファイルやプログラム上では、バックスラッシュは特定の記号のため、さらにエスケープする必要があります。

※2　エスケープした“¥¥{”をメッセージ中に複数定義していると、それ以降のメッセージが表示されないようです。Hibernate Validator 5.0.2.Final の不良かもしれません。

表 4.4　メッセージ内で利用可能な共通のパラメータ／オブジェクト

No.	パラメータ	説明
1	validatedValue	設定された値のインスタンスのオブジェクト。
2	formatter	オブジェクトのフォーマッター「java.util.Formatter」のインスタンス。 詳細は、「4.3.4.2　メッセージ内のフォーマッター」を参照。

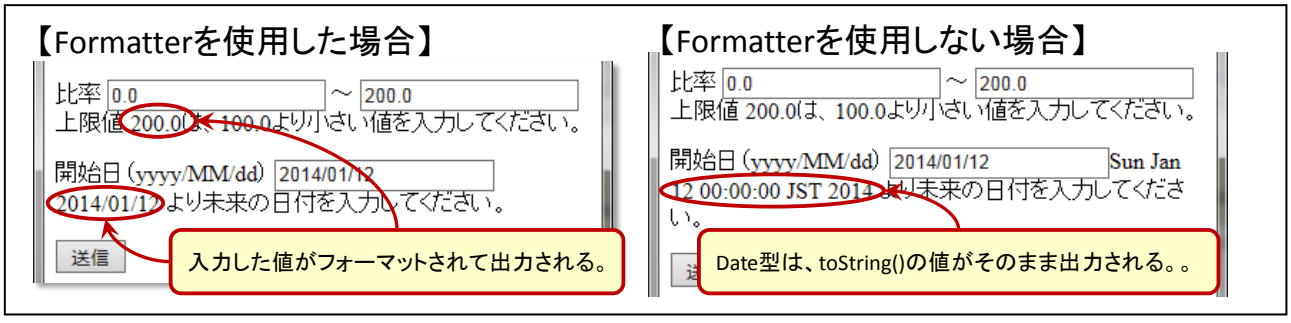
4.3.4.2.　メッセージ内のフォーマッター

- EL 式サポートされたことにより、オブジェクトやそのメソッドが呼び出し可能となりました。
- メソッドが呼び出し可能となったため、メッセージ中で数値型、日付型などの値を、フォーマットして出力可能となりました。
- 入力された値をメッセージ中に出力する場合、小数（double）、日付（java.util.Date）の場合、Java オブジェクトにマッピングされた後のため、もともとの書式がどのようなものか判断できないため、今までは出力しても嬉しくありませんでした。
- EL 式内では、標準オブジェクトとして、${formatter.format(…)}が利用できます。
  - 実体は、「java.util.Formatter.format(String format, Object... args)」です。
  - 書式は、Java5 から追加された printf 形式になります。
- 指定可能な書式は、下記を参照してください。
  - Javadoc「http://docs.oracle.com/javase/jp/7/api/java/util/Formatter.html」
  - ひしだま's 技術メモページ「http://www.ne.jp/asahi/hishidama/home/tech/java/formatter.html」

【メッセージの定義例】

```
public class SampleBean {

    @DecimalMin(inclusive=true, value = "0.0")
    private Double startRatio;

    @DecimalMax(inclusive=false, value = "100.0",
            message = "上限値 ${formatter.format('%1$.1f', validatedValue)}は、{value}${inclusive == true ? '以下の' : 'より小さい'}値を入力してください。")
    private Double endRatio;

    @Future(message="  ${formatter.format('%1$tY/%1$tm/%1$td%n',  validatedValue)}  より未来の日付を入力してください。")
    private Date startDate;

    // getter/setter は省略

}
```

【画面の表示】



4.3.5. 追加／変更された Bean Validation のアノテーション

4.3.5.1. Bean Validation のアノテーションの変更点

- EL 式がサポートされたことにより、@DecimalMax、@DecimalMin の数値の大小関係の比較において、不等号「>、≧、<、≦」が表現できるようになりました。
- 追加されたアノテーションの要素「inclusive」のデフォルト値は“true”のため、ver.1.0 からメッセージやアノテーションを変更しないでそのまま利用できます。
  - ただし、ver.1.1 の機能を利用したいならば、メッセージも EL 式による判定で記述すべきです。

表 4.5 Bean Validation の変更されたアノテーション

No.	ｱﾉﾃｰｼｮﾝ（※1）	変更
1	@DecimalMax	・アノテーションの要素「inclusive::boolean」が追加されました。 これは、判定条件にその値を含むかどうかを指定するフラグです。 ・デフォルト値は“true”です。 ・“true”の場合、要素「maximum」で指定した値以下のとき正常値と判定する。 ・“false”の場合、要素「maximum」で指定した値より小さいとき正常値と判定する。
2	@DecimalMin	・アノテーションの要素「inclusive::boolean」が追加されました。 これは、判定条件にその値を含むかどうかを指定するフラグです。 ・デフォルト値は“true”です。 ・“true”の場合、要素「minimum」で指定した値以上のとき正常値と判定する。 ・“false”の場合、要素「minimum」で指定した値より大きいとき正常値と判定する。

※1 アノテーションのパッケージは、「javax.validation.constraints」です。

```
## Bean Validation 1.0 のメッセージ
javax.validation.constraints.DecimalMin.message={value}以上の値を入力してください。
javax.validation.constraints.DecimalMax.message={value}以下の値を入力してください。

## Bean Validation 1.1 のメッセージ
javax.validation.constraints.DecimalMin.message={value}${inclusive == true ? '以上の' : 'より大きい'}値を入力してください。
javax.validation.constraints.DecimalMax.message={value}${inclusive == true ? '以下の' : 'より小さい'}値を入力してください。
```

図 4.1 Bean Validation のバージョンによるメッセージの違い

4.3.5.2. Hibernate Validator のアノテーションの変更

- 新たなアノテーション「@CNPJ」「@CPF」「@TituloEleitor」が追加されました。
- これは、ブラジル内で使用する特殊な書式チェックのためのもので、アノテーション「@Pattern」を拡張したものであり、特に使用はしないと思います。

表 4.6 Hibernate Validator のアノテーション

No.	ｱﾉﾃｰｼｮﾝ（※1）	変更
1	@CNPJ	・ブラジル国内の会社（法人）の税金支払いのための登録番号のパターンです。 ・String 型に対して付与可能です。
2	@CPF	・ブラジル国内の個人の税金支払いのための登録番号のパターンです。 ・String 型に対して付与可能です。
3	@TituloEleitor	・ブラジル国内の投票用のためのカード番号の ID のパターンです。 ・String 型に対して付与可能です。

※1 アノテーションのパッケージは、「org.hibernate.validator.constraints.br」です。

5. Expression Language 3.0（JSR-341）(:TODO)

EL 式はもともと、JSP 内で使用することを想定したものだったため、Java EE と同時に策定されていましたが、EL3.0 からは、独立した仕様になりました。

EL3.0 から文法の拡張も行われ、式というより、プログラミング言語に近くなってきました。

- EL3.0 の仕様書は、下記からダウンロードできます。
  http://download.oracle.com/otndocs/jcp/el-3_0-fr-eval-spec/index.html

サーバ側の JSP で EL3.0 を使用したい場合は、Tomcat8 以上を利用します。ただし、Tomcat8 は現在リリース候補版で、正式リリースには至っていません。
ここでは、Bean Validation 1.1 中で利用することを前提に説明します。

5.1. 演算子

5.2. EL3.0 の追加機能

5.2.1. Collection オブジェクトの操作

EL 式上で、Collection（List、Map）、配列について、今まで参照のみでしたが生成できるようになりました。

【インスタンスの作成】

【コレクションの操作】
Java8 の Stream を利用し、操作可能です。

http://kikutaro777.hatenablog.com/entry/2013/10/05/125052

【コレクションの参照】
既存通り、インデックスやキーを指定して参照。

※Bean Validation 中のメッセージだと、複雑な演算子など使用できない。