Tutorial: Using Thymeleaf (ja)

Japanese translation contributed by: Mitsuyuki Shiiba (@bufferings)
Document version: 20141222 - 22 December 2014
Project version: 2.1.4.RELEASE

Project web site: http://www.thymeleaf.org

1 Thymeleafの□介

1.1 Thymeleafって何?

Thymeleafは Javaのテンプレートエンジンライブラリです。XML/XHTML/HTML5で□かれたテンプレートを□□して、アプリケーションのデータやテキストを表示することができます。

ウェブアプリケーション内のXHTML/HTML5を□う方が得意ですが、どんなXMLファイルでも□理できますし、ウェブアプリケーションでもスタンドアローンアプリケーションでも使用可能です。

Thymeleafのメインゴールは、テンプレート作成のための□雅で整形式の方法を提供することです。そのため、テンプレート内にロジックを□述する方法ではなく、事前定□されたロジックの□行を *DOM(Document Object Model)* 上でXMLタグ属性によって指定する方法を基本としています。

このアーキテクチャのおかげで、パースしたファイルを□くキャッシュして□行□のI/O□理を最小限に抑えることができるので、テンプレートを高速に□理することが可能となっています。

さらに、Thymeleafは最初からXMLとウェブ□□を念□に置いてデザインされているので、必要に□じて完全にバリデーションされた状□のテンプレートを作成することもできます。

1.2 Thymeleafはどんな□□のテンプレート を□理できるの?

Thymeleafは6□□のテンプレートを□理することができます。これをテンプレートモードと呼びます:

- XML
- Valid XML
- XHTML
- Valid XHTML
- HTML5
- Legacy HTML5

Legacy HTML5 以外は整形式XMLです。*Legacy HTML5* モードでは□じていないタグ□がない属性・引用符で□まれていない属性が□容されていますが、Thymeleafはこのモードのファイルを最初に整形式XMLに□□します。それでもHTML5としては正しい状□です(そして□□こちらがHTML5を□くのに推□されている方法です) [1]。

また、バリデーションはXMLとXHTMLのみで使用可能なことに注意してください。

ただし、Thymeleafが□理できるテンプレートのタイプはこれだけではありません。テンプレートを「パースする方法」と□果を「□き□む方法」を指定することで、ユーザーは独自 のモードを定□することができます。Thymeleafは、DOMツリーとして表□することができるものであれば何でも(XMLかどうか□□なく)テンプレートとして□率よく□理することができます。

1.3 ダイアレクト :スタンダ□ド ダイアレクト

Thymeleafは非常に□□性の高いテンプレートエンジンです(□□「テンプレートエンジンフレームワーク」と呼んだほうがいいかもしれません)。Thymeleafでは、□理□象のDOMノードと、そのDOMノードをどのように□理するかを完全に定□することができます。

DOMノードにロジックを□用するものを「プロセッサ」と呼びます。そして、プロセッサ一式 — と、いくつかの特□な生成物 — のことをダイアレクトと呼びます。Thymeleafでは「スタンダードダイアレクト」というそのまますぐに使えるコアライブラリを提供していて、大半のユーザーにとってはこれで十分です。

このチュートリアルでカバーしているのはスタンダードダイアレクトです。以降のページで学ぶ全ての属性や文法は特に明□していなくても、このダイアレクトに定□してあります。

もちろん、ライブラリの□□□能を利用して独自 の□理ロジックを定□したい、など(スタンダードダイアレクトを□□することも含めて)独自 のダイアレクトを作りたい□合があるかもしれません。テンプレートエンジンは□数のダイアレクトを同□に使用できます。

> 公式のthymeleaf-spring3とthymeleaf-spring4□携□ケージはどちらも「Springスタンダードダイアレクト」と呼□れるダイアレクトを定□しています。これは、ほぼスタンダードダイアレクトと同じで、そこにSpring Framework用の便利□能を少しき□用しています(例えば、Thymeleaf□□のOGNLの代わりにSpring式言□を使用するなど)。ですので、Spring MVCを使用するような□合でも□□の□□にはなりません。ここで学ぶことは全て、Springアプリケーションを作成する□にも役立つでしょう。

Thymeleafのスタンダードダイアレクトはどのテンプレートモードでも使用できますが、特にウェブ向けのテンプレートモード□□しています(XHTMLとHTML5モード)。HTML5の他に具体的には以下のXHTML仕□をサポート・□作□□しています: "XHTML 1.0 Transitional"、"XHTML 1.0 Strict"、"XHTML 1.0 Frameset"、そして"XHTML 1.1"です。

スタンダードダイアレクトの大半の□□セッサは□属性□□セッサ□です。属性□□セッサを使用すると、XHTML/HTML5テンプレートファイルは□理前であってもブラウザで正しく表示することができます。□□にその属性が□□されるからです。例えば、タグライブラリを使用したJSPだとブラウザで直接表示できない□合がありますが:

```
<form:inputText name="userName" value="${user.name}" />
```

Thymeleafスタンダードダイアレクトでは同□の□能をこのように□□します:

```
<input type="text" name="userName" value="James Carrot" th:value="${user.name}" />
```

ブラウザで正しく表示できるだけでなく、(任意ですが)value属性を指定することもできます(この□合の"James Carrot"の部分です)。ホット・タイプを静的にブラウザで□いた□合にはこの□が表示され、Thymeleafでテンプレートを□理した□合には `${user.name}` の□□□果□で置き□えられます。

必要な□合には、全く同じファイルをデザイナーとデベロッパーが触ることができるので、静的なホット・タイプをテンプレートに□□する□力を□らすことができます。この□能のことを"ナチュラルテンプレーティング"と呼びます。

1.4 全体のア□キテクチャ

Thymeleafのコアは□□理エンジンです。具体的にいうと—□□のDOM API□はなく—高性能の独自DOM□装によってテンプレートのインメモリー表□を生成します。その後、そのインメモリー上でノードを走□して□□セッサを□行しDOMを□更します。DOMの□更は□在の□定や、テンプレートに渡されるコンテキストと呼□れるデータセットに□います。

ウェブドキュメントはオブジェクトツリーとして表□されることが本当によくあるので、DOMテンプレート表□の使用はウェブアプリケーション□にとても□しています(□□にブラウザはDOMツリーによってメモリ上でウェブページを表□します)。また、多くのウェブアプリケーションで、使用するテンプレート数は数十□程度である、そのテンプレートが大きなサイズではない、アプリケーションの□行中に通常は□更されない、という考えに基づいてThymeleafはテンプレートのDOMツリーのインメモリキャッシュを利用しています。これによって多くのテンプレート□理で(必要だとしても)ほんの少しのI/Oしか必要なくなるので、本番□境での□行を速くすることができます。

> このチュートリアルの後ろの方にキャッシュについて、高速な□理のために、Thymeleafがどのようにメモリとリソースを最□化しているかについて□明した章がありますので□□はそちらを参照してください。

しかし、制□もあります: このアーキテクチャではテンプレート□理に他のアプローチよりも多くのメモリスペースが必要になります。つまり、(ウェブドキュメントとは□照的な)大きなサイズのデータXMLの作成には使わない方が良いということです。大まかには(といってもJVMのメモリサイズによりますが)1テンプレートを□理するのに数十メガバイトが必要になるXMLファイルを□理する□合は、おそらくThymeleafを使わない方が良いでしょう。

> ここで、データXML□□してきたこの制□について考えているのは、ウェブのXHTML/HTML5□□しては、そんなに大きなサイズのドキュメントは作成しないからです。ブラウザもDOMツリーを生成するので、そんなことをすると固まってしまいますもんね。

1.5 次に□む前に□むことをお□めします...

Thymeleafは特にウェブアプリケーション□□しています。そしてウェブアプリケーション□には□□というものがあります。みんながこの□□についてよく知っているべきなのですが、ほとんどの人が知りません。たとえウェブアプリケーションの仕事を何年もやっている人であってもです。

HTML5の出□によって、今日のウェブ□□はかつてないほどに混乱しています...「XHTMLからHTMLに□るの?」「XMLシンタックスはなくなるの?」「XHTML2.0はどこにいったの?」

ということでこのチュートリアルでは先に□む前に Thymeleafのウェブサイトの次の□事を□むことを□くおすすめします: *"From HTML to HTML (via HTML)"* http://www.thymeleaf.org/fromhtmltohtmlviahtml.html

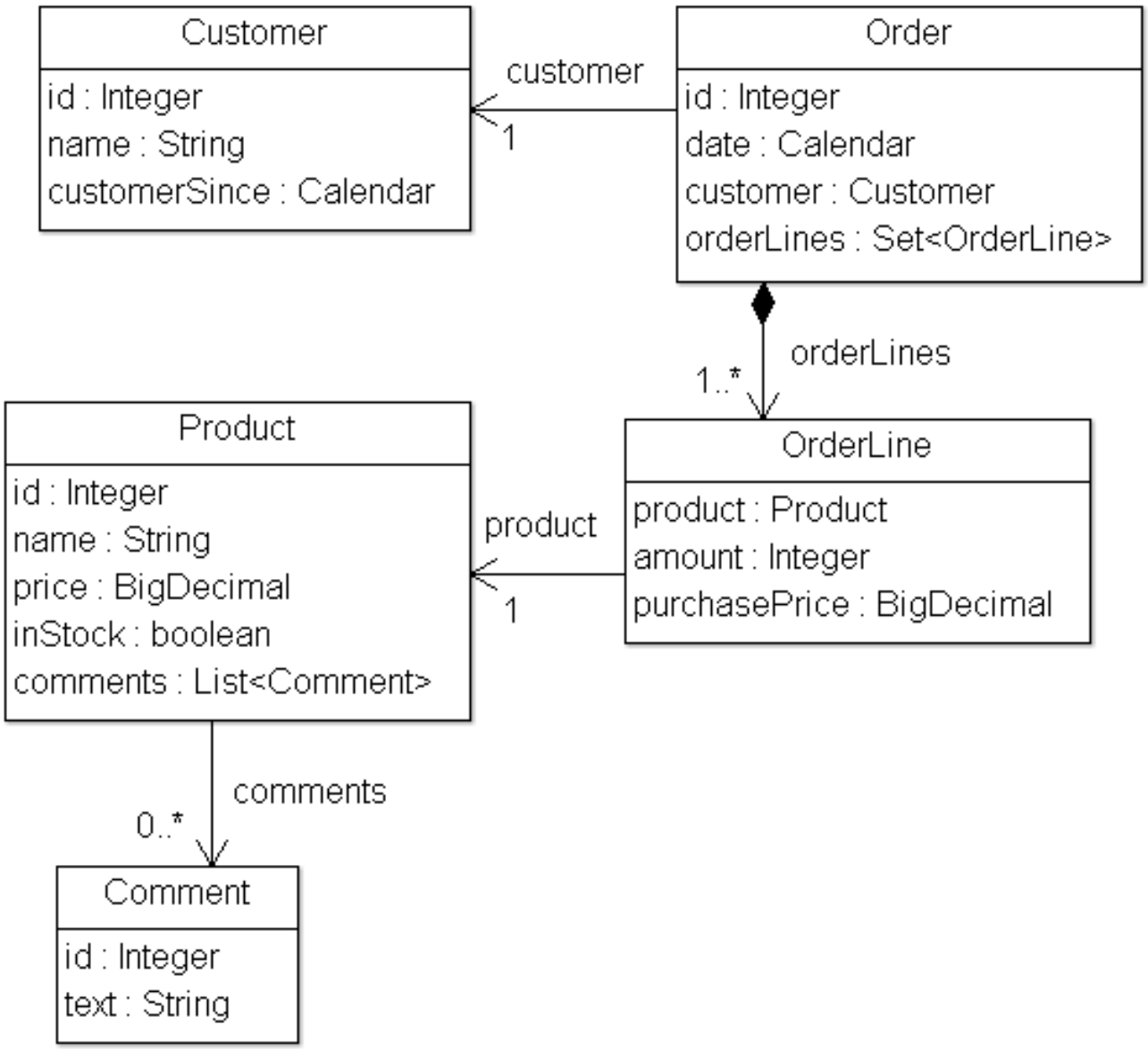
2 The Good Thymes Virtual Grocery(グッドタイムス□想食料品店)

2.1 食料品店用のウエブサイト

Thymeleafのテンプレート□理のエッセンスを分かりやすく□明するために、このチュートリアルでは□アプリケーションを使用します。□アプリケーションはオブジェクトのウェブサイトからダウンロードできます。

このアプリケーションは架空の□想食料品店のウェブサイトで、□々なThymeleafの□能の例をお□せするのに十分なデータが用意されています。

アプリケーションにはとてもシンプルなモデルエンティティが必要でしょう: `Products` は `Orders` を作成することによって `Customers` に□□されます。さらにこの `Products` について `Comments` も管理しましょう:



Example application model

とてもシンプルなサービスレイヤも作りましょう。次のようなメソッドを持つ `Service` オブジェクトです:

```
public class ProductService {
```

```
    ...

    public List<Product> findAll() {
        return ProductRepository.getInstance().findAll();
    }

    public Product findById(Integer id) {
        return ProductRepository.getInstance().findById(id);
    }

}
```

最後にリクエストURLに応じてThymeleafに処理を委譲するフィルタをウェブレイヤに作成しましょう:

```
private boolean process(HttpServletRequest request, HttpServletResponse response)
        throws ServletException {

    try {

        /*
         * Query controller/URL mapping and obtain the controller
         * that will process the request. If no controller is available,
         * return false and let other filters/servlets process the request.
         */
        IGTVGController controller = GTVGApplication.resolveControllerForRequest(request);
        if (controller == null) {
            return false;
        }
        /*
         * Obtain the TemplateEngine instance.
         */
        TemplateEngine templateEngine = GTVGApplication.getTemplateEngine();

        /*
         * Write the response headers
         */
        response.setContentType("text/html;charset=UTF-8");
        response.setHeader("Pragma", "no-cache");
        response.setHeader("Cache-Control", "no-cache");
        response.setDateHeader("Expires", 0);

        /*
         * Execute the controller and process view template,
         * writing the results to the response writer.
         */
        controller.process(
                request, response, this.servletContext, templateEngine);

        return true;

    } catch (Exception e) {
        throw new ServletException(e);
    }

}
```

`IGTVGController` インターフェイスは次のようになります:

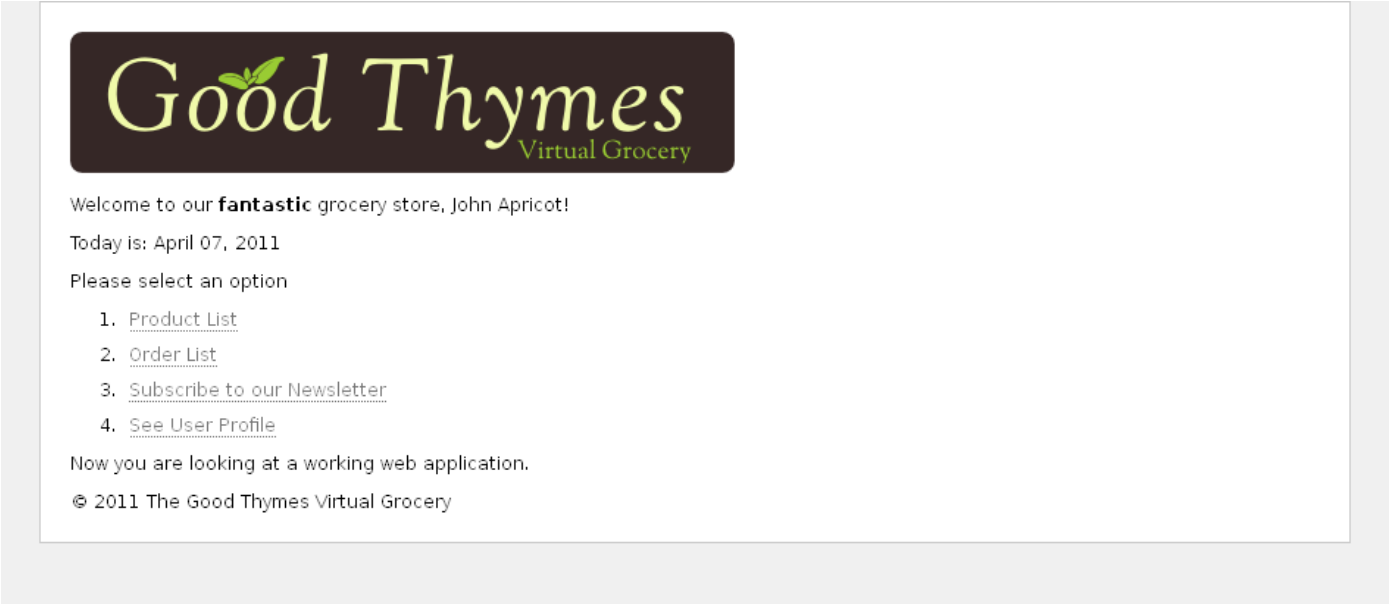
```
public interface IGTVGController {

    public void process(
            HttpServletRequest request, HttpServletResponse response,
            ServletContext servletContext, TemplateEngine templateEngine);

}
```

これで `IGTVGController` の実装を作成すれば良いだけです。データをサービスから受け取って `TemplateEngine` オブジェクトを使用してテンプレートを処理します。

最終的にはこのようになりますが:



Example application home page

まずはテンプレートエンジンの初期化について□てみましょう。

2.2 テンプレ□ト エンジンの作成と□定

フィルタの *process(...)* メソッドの中に次のような文があります:

```
TemplateEngine templateEngine = GTVGApplication.getTemplateEngine();
```

これは Thymeleafを使用するアプリケーションにおいて最も重要なオブジェクトの中の一つである `TemplateEngine` インスタンスの作成と□定を *GTVGApplication* クラスが担っているということです。

ここでは `org.thymeleaf.TemplateEngine` を次のように初期化しています:

```
public class GTVGApplication {


    ...
    private static TemplateEngine templateEngine;
    ...


    static {
        ...
        initializeTemplateEngine();
        ...
    }


    private static void initializeTemplateEngine() {

        ServletContextTemplateResolver templateResolver =
            new ServletContextTemplateResolver();
        // XHTML is the default mode, but we set it anyway for better understanding of code
        templateResolver.setTemplateMode("XHTML");
        // This will convert "home" to "/WEB-INF/templates/home.html"
        templateResolver.setPrefix("/WEB-INF/templates/");
        templateResolver.setSuffix(".html");
        // Template cache TTL=1h. If not set, entries would be cached until expelled by LRU
        templateResolver.setCacheTTLMs(3600000L);

        templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(templateResolver);

    }
```

```
    ...
}
```

もちろん `TemplateEngine` オブジェクトを初期化するのには□々な方法がありますが、今はこの数行のコードで十分です。

テンプレートリゾルバ□

テンプレートリゾルバーからスタートしましょう:

```
ServletContextTemplateResolver templateResolver = new ServletContextTemplateResolver();
```

テンプレートリゾルバーはThymeleafのAPIである `org.thymeleaf.templateresolver.ITemplateResolver` を□装しています:

```
public interface ITemplateResolver {

    ...

    /*
     * 文字列名(templateProcessingParameters.getTemplateName())によってテンプレートを解□します。
     * このテンプレートリゾルバーで解□できない□合は null を返します。
     */
    public TemplateResolution resolveTemplate(
            TemplateProcessingParameters templateProcessingParameters);

}
```

テンプレートリゾルバーは、どうやってテンプレートにアクセスするかを□定する役割を担っています。GTVGアプリケーションの□合は `org.thymeleaf.templateresolver.ServletContextTemplateResolver` □装を使用して *Servlet Context* からテンプレートファイルを取得します: Javaの全てのウェブアプリケーションにはアプリケーションレベルの `javax.servlet.ServletContext` というオブジェクトが存在し、それによってウェブアプリケーションのルートをリソースベースのルートとしてリソースを解□することができます。

テンプレートリゾルバーにはいくつかのパラメータを□定することができます。まず、□□的なものとして、テンプレートモードがあります:

```
templateResolver.setTemplateMode("XHTML");
```

XHTMLは `ServletContextTemplateResolver` のデフォルトテンプレートモードですが意□を明らかにするために□いておくのは良い□□ですね。

```
templateResolver.setPrefix("/WEB-INF/templates/");
templateResolver.setSuffix(".html");
```

prefix と *suffix* は文字通り、テンプレート名から□□のリソース名を作り出すために使用されます。

この□定を使用すると *"product/list"* というテンプレート名は次の内容と同じになります:

```
servletContext.getResourceAsStream("/WEB-INF/templates/product/list.html")
```

任意ですが *cacheTTLMs* でテンプレートキャッシュの生存期□を指定することもできます:

```
templateResolver.setCacheTTLMs(3600000L);
```

もちろんTTL以内であってもキャッシュのサイズが最大□に□した□合は古いエントリーから削除されます。

> キャッシュの振る舞いやサイズは `ICacheManager` インターフェイスの□装によって定□されます。または、□□にデフォルトで□定されている `StandardCacheManager` を修正しても良いです。

テンプレートリゾルバーについてのより□□な□明は後ほど行います。今はテンプレートエンジンオブジェクトの生成について□てみましょう。

テンプレート エンジン

テンプレートエンジンオブジェクトとは *org.thymeleaf.TemplateEngine* のことです。□在の例ではこのようにエンジンを作成しています:

```
templateEngine = new TemplateEngine();
templateEngine.setTemplateResolver(templateResolver);
```

かなりシンプルですよね。インスタンスを作成してテンプレートリゾルバーをセットするだけです。

`TemplateEngine` に必□のパラメータはテンプレートリゾルバーだけです。もちろん他にも色々な□定があります(メッセージリゾルバーやキャッシュサイズなど)が、それについては後ほど□明します。今はこれだけで十分です。

これでテンプレートエンジンの□□ができました。では、Thymeleafを使用したページの作成に□みましょう。

3 テキストを使う

3.1 □数言□でウエルカム

私たちの食料品店用の最初のアプリはホームページ作成です。

最初のバージョンは非常にシンプルです: タイトルとウェルカムメッセージだけです。`/WEB-INF/templates/home.html` は以下のようになります:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <p th:text="#{home.welcome}">Welcome to our grocery store!</p>

  </body>

</html>
```

最初に□て欲しいのは、このファイルがどんなブラウザでも正しく表示できるXHTMLであるということです。理由は、XHTMLにあるアプリが使っていないからです(そしてブラウザは `th:text` のような知らない属性は□□します)。また整形式の `DOCTYPE` 宣言を持っているので互□モードではなくスタンダードモードで表示されます。

次に、このファイルは `th:text` のような属性を定□したThymeleafのDTDを指定しているので妥当なXHTMLでもあります[2]。さらにテンプレートが□理されると(全ての `th:*` 属性が取り除かれますが)、Thymeleafは自□的に `DOCTYPE` 内のDTD定□を□□的な `XHTML 1.0 Strict` のものに置き□えます(このDTD□□□能に□しては後の章で□明します)。

thymeleaf名前空□も `th:*` として定□されています。

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
```

もしテンプレートの妥当性や整形式であるかどうかを全く□にしないのであれば□□に□□の `XHTML 1.0 Strict DOCTYPE` を指定すればよく、xmlns名前空□の定□も不要であることに□をつけてください:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <p th:text="#{home.welcome}">Welcome to our grocery store!</p>

  </body>

</html>
```

...こういってもXHTMLモードのThymeleafは□□な□理することができます(IDEの警告で残念な感じになると思いますけど)。

バリデーション□□してはOKですね。ではテンプレート□□する本当に面白い部分□□みましょう: `th:text` 属性を□て行きましょう。

th:text とテキスト の外部化

テキストの外部化とは、テンプレートコードのフラグメント(断片)をテンプレートファイルの外に取り出すことです。それによって、テンプレートから切り□された□のファイル(通常は `.properties` ファイル)の中でフラグメントを管理することができ、また □□に他の言□で□かれた文字列に置き□えることができます(このことを多言□□□、または *i18n* と呼びます)。外部化されたテキストのフラグメントのことを通常は "メッセージ" と呼びます。

メッセージは、そのメッセージを特定するためのキーを持っており、Thymeleafは `#{...}` という□文を使用してテキストとメッセージの□付けを行います:

```
<p th:text="#{home.welcome}">Welcome to our grocery store!</p>
```

ここでは□□、Thymeleafスタンダードダイアレクトの2つの□なる□能を使用しています:

- `th:text` 属性: この属性は□の式を□□した□果をタグのボディに□定します。ここでは、コード内の "Welcome to our grocery store!" というテキストを置□します。
- `#{home.welcome}` 式: 「スタンダード式□文」に□定されています。ここでは、テンプレートを□理する全てのロケールで `home.welcome` キーに□□するメッセージを取得して `th:text` 属性で使用するということを意味します。

ふむ。では外部化されたテキストはどこにあるのでしょうか?

Thymeleafでは外部化テキストの□所は `org.thymeleaf.messageresolver.IMessageResolver` を□装することで自由に□定できます。通常は `.properties` ファイルを使用する□装になっていますが、独自 □装を作成することも可能です。例えばメッセージをDBから取得することも可能です。

ところで、私たちのテンプレートエンジンには初期化の□にメッセージリゾルバーを指定していません。これは `org.thymeleaf.messageresolver.StandardMessageResolver` クラスによって□装されている「スタンダードメッセージリゾルバー」を使用していますよ、ということです。

スタンダードメッセージリゾルバーは `/WEB-INF/templates/home.html` というテンプレートに□してテンプレートと同じフォルダ内で、同じ名前のファイルで□□子が .properties のファイルの中からメッセージを探します。

- `/WEB-INF/templates/home_en.properties` が英□用。
- `/WEB-INF/templates/home_es.properties` がスペイン□用。
- `/WEB-INF/templates/home_pt_BR.properties` がポルトガル□(ブラジル)用。
- `/WEB-INF/templates/home.properties` がデフォルト用(ロケールが一致しない□合)。

`home_es.properties` ファイルを□てみましょう:

```
home.welcome=¡Bienvenido a nuestra tienda de comestibles!
```

これでThymeleafのテンプレート□理に必要なことは全て□わりました。ではHomeコントローラーを作成しましょう。

コンテキスト

テンプレートを□理するために `HomeController` クラスを作成します。前述の `IGTVGController` インターフェイスを□装します:

```
public class HomeController implements IGTVGController {

    public void process(
            HttpServletRequest request, HttpServletResponse response,
            ServletContext servletContext, TemplateEngine templateEngine) {

        WebContext ctx =
            new WebContext(request, response, servletContext, request.getLocale());
        templateEngine.process("home", ctx, response.getWriter());

    }
```

```
}
```

まずはコンテキストの作成について□てみましょう。Thymeleafのコンテキストは `org.thymeleaf.context.IContext` インターフェイスを□装したオブジェクトです。コンテキストはテンプレートエンジンの□行に必要な全てのデータを□数のマップとして持ち、また外部化メッセージで使用される□ケールへの参照を持っています。

```
public interface IContext {

    public VariablesMap<String,Object> getVariables();
    public Locale getLocale();
    ...

}
```

このインターフェイスの□□として `org.thymeleaf.context.IWebContext` というインターフェイスがあります:

```
public interface IWebContext extends IContext {

    public HttpSerlvetRequest getHttpServletRequest();
    public HttpSession getHttpSession();
    public ServletContext getServletContext();

    public VariablesMap<String,String[]> getRequestParameters();
    public VariablesMap<String,Object> getRequestAttributes();
    public VariablesMap<String,Object> getSessionAttributes();
    public VariablesMap<String,Object> getApplicationAttributes();

}
```

Thymeleafのコアライブラリはそれぞれの□装を提供しています:

- `org.thymeleaf.context.Context` implements `IContext`
- `org.thymeleaf.context.WebContext` implements `IWebContext`

コントローラーのコードを□ていただければ分かるように、ここでは `WebContext` を使用しています。というか、そうしなければなりません。`ServletContextTemplateResolver` が `IWebContext` の□装を必要とするからです。

```
WebContext ctx = new WebContext(request, servletContext, request.getLocale());
```

3つの引数のうち2つまでが必□です。□ケールに何も指定しなかったらシステムのデフォルト□ケールが使用されます(□□のアプリケーションでは□□に指定した方がよいですが)。

インターフェイスの定□から `WebContext` はリクエストパラメータ、リクエスト属性、セッション属性、アプリケーション属性を取得するメソッドを持っていることが分かりますが、□□のところ `WebContext` はもう少し色んとやっています:

- 全てのリクエスト属性をコンテキスト□数マップに追加。
- 全てのリクエストパラメータを持つ`param` というコンテキスト□数を追加。
- 全てのセッション□数を持つ`session` というコンテキスト□数を追加。
- 全てのサーブレットコンテキスト属性を持つ`application` というコンテキスト□数を追加。

□行直前に全てのコンテキストオブジェクト(`IContext` の□装)に□して特□な□数が□定されます。`Context` と `WebContext` のどちらもその□象です。この□数は□行情□(`execInfo`)と呼ばれます。この□数はテンプレートで使用される2つのデータを持っています。

- テンプレート名(`${execInfo.templateName}`): エンジンの□行□に指定される名前です。これは、□理するテンプレート名と一致します。
- □在日 □(`${execInfo.now}`): テンプレートエンジンが□在のテンプレートの□理を□始した日 □を示す `Calendar` オブジェクトです。

テンプレート エンジンの□行

コンテキストオブジェクトが□□できたので、あとはテンプレートエンジンを□行するだけです。テンプレート名とコンテキストとレスポンスライターを渡してレスポンスへの□き□みを行います:

```
templateEngine.process("home", ctx, response.getWriter());
```

ではスパイン□ロケールを使用して□果を□てみましょう:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
    <link rel="stylesheet" type="text/css" media="all" href="/gtvg/css/gtvg.css" />
  </head>

  <body>

    <p>¡Bienvenido a nuestra tienda de comestibles!</p>

  </body>

</html>
```

3.2 テキスト と □数に□するその他のこと

エスケ□プ処理テキスト

私たちのホームページの最もシンプルなバージョンは□□できましたが、もしメッセージが次のようなものだったらどうしましょう...

```
home.welcome=Welcome to our <b>fantastic</b> grocery store!
```

今のままでテンプレートを□行するとこのようになります:

```
<p>Welcome to our &lt;b&gt;fantastic&lt;/b&gt; grocery store!</p>
```

これは本当に欲しい□果ではありません。 `<b>` タグがエスケープされてブラウザに表示されてしまっています。

これはth:text属性のデフォルトの振る舞いです。ThymeleafでXHTMLタグをエスケープせずに表示したいのであれば、□う属性を使用しなければなりません: `th:utext` ("unescaped text"用):

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>
This will output our message just like we wanted it:
<p>Welcome to our <b>fantastic</b> grocery store!</p>
```

□数の使用と表示

さて、私たちのホームページについてもう少し□てみましょう。例えば、ウェルカムメッセージに次のようなデータを表示したいかもしれません:

```
Welcome to our fantastic grocery store!

Today is: 12 july 2010
```

まずはじめに コントローラーを修正してコンテキスト□数に日 付を追加します:

```
public void process(
        HttpServletRequest request, HttpServletResponse response,
        ServletContext servletContext, TemplateEngine templateEngine) {

    SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");
    Calendar cal = Calendar.getInstance();

    WebContext ctx =
        new WebContext(request, response, servletContext, request.getLocale());
```

```
    ctx.setVariable("today", dateFormat.format(cal.getTime()));

    templateEngine.process("home", ctx, response.getWriter());

}
```

`String` 型のtoday□数をコンテキストに追加したので、テンプレートで表示することができるようになりました

```
<body>

  <p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

  <p>Today is: <span th:text="${today}">13 February 2011</span></p>

</body>
```

□ての通りここでも `th:text` 属性を使用しています(タグのボディを置□したいので、これで□□ありません)。ですが□文が少し□いますね。 `#{...}` 式ではなく `${...}` 式を使っています。これが□数用の式です。 *OGNL (Object-Graph Navigation Language)* と呼ばれる言□の式でコンテキスト□数マップに□して□理を行います。

この `${today}` は□□に「todayという名前の□数を取得する」という意味ですが、もっと□□なこともできます(例えば `${user.name}` は「user□数を取得してその `getName()` メソッドを呼び出す」という意味になります)。

属性には□々な□を□定することができます: メッセージ、□数式、...などなど。次の章では、どのようなものが指定できるかを全て□ていきましょう。

4 スタンダード式文

私たちの想食料品店のは少し休憩して、Thymeleafスタンダードダイアレクトの中でもっとも重要なものの一つについて学んでいきましょう: 「Thymeleafスタンダード式文」です。

この文を使って表された2タイプの属性を既にできました メッセージ式と数式です:

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

<p>Today is: <span th:text="${today}">13 february 2011</span></p>
```

ですが、まだ知らないタイプがたくさんあります。また、知っているものにももっと面白い部分があります。初めにスタンダード式の能の概要をてみましょう。

- 式:
  - 数式: `${...}`
  - 数式: `*{...}`
  - メッセージ式: `#{...}`
  - リンクURL式: `@{...}`
- リテラル
  - テキストリテラル: `'one text'`, `'Another one!'`,...
  - 数リテラル: `0`, `34`, `3.0`, `12.3`,...
  - 真リテラル: `true`, `false`
  - Nullリテラル: `null`
  - リテラルトークン: `one`, `sometext`, `main`,...
- テキスト演算子:
  - 文字列合: `+`
  - リテラル置: `|The name is ${name}|`
- 算演算子:
  - バイナリ演算子: `+`, `-`, `*`, `/`, `%`
  - マイナス符号 (演算子): `-`
- 理演算子:
  - 二演算子: `and`, `or`
  - 理否定演算子 (演算子): `!`, `not`
- 比と等:
  - 比演算子: `>`, `<`, `>=`, `<=` (`gt`, `lt`, `ge`, `le`)
  - 等演算子: `==`, `!=` (`eq`, `ne`)
- 条件演算子:
  - If-then: `(if) ? (then)`
  - If-then-else: `(if) ? (then) : (else)`
  - Default: `(value) ?: (defaultvalue)`

これら全ての能は、合したりネストしたりすることができます:

```
'User is of type ' + (${user.isAdmin()} ? 'Administrator' : (${user.type} ?: 'Unknown'))
```

4.1 メッセージ

ご存知の通り `#{...}` メッセージ式は次のようにいて:

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>
```

...これとリンクすることができます:

```
home.welcome=¡Bienvenido a nuestra tienda de comestibles!
```

でも、まだ考えていないことが一つあります: メッセージテキストが完全に静的ではない□合はどうしましょうか?例えば、アプリケーションは□がサイトにこれているかをいつでも知っているとして、その人の名前を呼んで挨拶文を出したい□合にはどのようにすればいいのでしょうか?

```
<p>¡Bienvenido a nuestra tienda de comestibles, John Apricot!</p>
```

つまり、メッセージにパラメータを持たせる必要があるということです。こんなふうに

```
home.welcome=¡Bienvenido a nuestra tienda de comestibles, {0}!
```

パラメータは `java.text.MessageFormat` の□□□文に□って指定します。つまり、そのクラスのAPIドキュメントにあるように、数□や日 付にフォーマットを指定することもできるということです。

HTTPセッションに持っている `user` という属性をパラメータとして指定するには次のように□述します:

```
<p th:utext="#{home.welcome(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

必要に□じて□数のパラメータをカンマ区 切りで指定することも可能です。□□のところ、メッセージ自 体も□数から取得することができます:

```
<p th:utext="#{${welcomeMsgKey}(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

4.2 □数

既に述べたように `${...}` 式は、□□にはコンテキスト内の□数マップ上で□行されるOGNL(Object-Graph Navigation Language)式です。

> OGNL□文や□能についての□□はOGNL Language Guideを参照してください: http://commons.apache.org/ognl/

OGNL□文から次のようなことが分かります。以下の内容は:

```
<p>Today is: <span th:text="${today}">13 february 2011</span>.</p>
```

...□□には次の内容と同等です:

```
ctx.getVariables().get("today");
```

ただし、OGNLではもっとパワフルな表□が可能です。こんな□に:

```
<p th:utext="#{home.welcome(${session.user.name})}">
  Welcome to our grocery store, Sebastian Pepper!
</p>
```

...これは、□□には次の□理を□行することでユーザー名を取得します:

```
((User) ctx.getVariables().get("session").get("user")).getName();
```

ですが、Getterメソッドのナビゲーションは、OGNLの□能の1つにすぎません。もっと□てみましょう:

```
/*
 * ポイント (.)を使用したプロパティへのアクセス。プロパティのGetterを呼び出すのと同
```

```
です。
 */
${person.father.name}

/*
 * プロパティへのアクセスは角括弧([])にプロパティ名を指定することでも可能です。
 * プロパティ名の指定は□数でも、シングルクオート で□まれた文字列でも可能です。
 */
${person['father']['name']}

/*
 * オブジェクトがマップの□合、ドット も括弧も同□に get(...) メソッドを呼び出します。
 */
${countriesByCode.ES}
${personsByName['Stephen Zucchini'].age}

/*
 * 配列やコレクションに□するインデックスを使用したアクセスも同□に角括弧を使用します。
 * インデックスをクオート なしで□きます。
 */
${personsArray[0].name}

/*
 * メソッド呼び出しが可能です。引数あり でも可能です。
 */
${person.createCompleteName()}
${person.createCompleteNameWithSeparator('-')}
```

式の基本オブジェクト

コンテキスト□数に□してOGNL式で□□をする□に より柔□に□述できるようにいくつかのオブジェクトを用意しています。これらのオブジェクトの参照は(OGNL □□に□って) # シンボルで始まります:

- `#ctx`: コンテキストオブジェクト。
- `#vars`: コンテキスト□数。
- `#locale`: コンテキストロケール。
- `#httpServletRequest`: (ウェブコンテキストのみ) `HttpServletRequest` オブジェクト。
- `#httpSession`: (ウェブコンテキストのみ) `HttpSession` オブジェクト。

次のようなことができます:

```
Established locale country: <span th:text="${#locale.country}">US</span>.
```

□□は Appendix A を参照して下さい。

式のユーティリティオブジェクト

基本オブジェクト以外にも、式の中の共通のタスクを手助けするためのユーティリティオブジェクトがあります。

- `#dates`: `java.util.Date` オブジェクト用のユーティリティメソッド: フォーマット、コポーネントの抽出など。
- `#calendars`: `#dates` に似ていますが `java.util.Calendar` オブジェクト用です。
- `#numbers`: 数□オブジェクト用のユーティリティメソッド。
- `#strings`: `String` オブジェクト用のユーティリティメソッド: contains, startsWith, prepending/appending, など。
- `#objects`: オブジェクト一般のユーティリティメソッド。
- `#bools`: 真□□□□用のユーティリティメソッド。
- `#arrays`: 配列用のユーティリティメソッド。
- `#lists`: リスト用のユーティリティメソッド。
- `#sets`: セット用のユーティリティメソッド。
- `#maps`: マップ用のユーティリティメソッド。
- `#aggregates`: 配列やコレクション上での集□□理用ユーティリティメソッド。
- `#messages`: #{...} と同□に、□数式内での外部化メッセージを取り□うためのユーティリティメソッド。
- `#ids`: (例えば、イテレーション□果などの)□り返し□理内でid属性を取り□うためのユーティリティメソッド。

それぞれのユーティリティオブジェクトの□□については Appendix B を参照してください。

私たちのホームページ内の日付書式フォーマット

ユーティリティオブジェクトについて学んだので、それを使って私たちのホームページ内の日付表示を□えてみましょう。次のように `HomeController` で□理する代わりに

```
SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");
Calendar cal = Calendar.getInstance();

WebContext ctx = new WebContext(request, servletContext, request.getLocale());
ctx.setVariable("today", dateFormat.format(cal.getTime()));

templateEngine.process("home", ctx, response.getWriter());
```

...次のように□いて:

```
WebContext ctx = new WebContext(request, servletContext, request.getLocale());
ctx.setVariable("today", Calendar.getInstance());

templateEngine.process("home", ctx, response.getWriter());
```

...ビュー□でフォーマットすることができます:

```
<p>
  Today is: <span th:text="${#calendars.format(today,'dd MMMM yyyy')}">13 May 2011</span>
</p>
```

4.3 □□したものに□する式(アスタリスク□文)

□数式は `${...}` だけでなく `*{...}` でも□くことができます。

重要なのは、アスタリスク□文はコンテキスト□数マップに□してではなく、□□されたオブジェクトに□して□□をする式であるということです。□□されたオブジェクトがない□合は、ダラー□文もアスタリスク□文も全く同じになります。

オブジェクトの□□とはどういうことでしょうか? `th:object` のことです。では、ユーザープロフィールページ(`userprofile.html`)で使ってみましょう:

```
  <div th:object="${session.user}">
    <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
    <p>Surname: <span th:text="*{lastName}">Pepper</span>.</p>
    <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
  </div>
```

これは次と全く同じです:

```
<div>
  <p>Name: <span th:text="${session.user.firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="${session.user.nationality}">Saturn</span>.</p>
</div>
```

もちろん、ダラー□文とアスタリスク□文は共存可能です:

```
<div th:object="${session.user}">
  <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

ダラー□文内で `#object` 式□数を使用して□□されているオブジェクトを参照することもできます:

```
<div th:object="${session.user}">
  <p>Name: <span th:text="${#object.firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

□り返しになりますが、オブジェクトが□□されていない□合は□ー□文とアスタリスク□文は全く同じ意味になります。

```
<div>
  <p>Name: <span th:text="*{session.user.name}">Sebastian</span>.</p>
  <p>Surname: <span th:text="*{session.user.surname}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{session.user.nationality}">Saturn</span>.</p>
</div>
```

4.4 リンクURL

その重要性から、URLはウェブアプリケーションテンプレートにおけるファースト・クラスのオブジェクトであり、Thymeleafスタンダード・ダイアレクト」にも特□な□文が用意されています。 `@` □文です: `@{...}`

URLにはいくつかのタイプがあります:

- □□URL: `http://www.thymeleaf.org`
- 相□URL:
  - ページ相□URL: `user/login.html`
  - コンテキスト相□URL: `/itemdetails?id=3` (サーバー内のコンテキスト名は自 □的に付与されます)
  - サーバー相□URL: `~/billing/processInvoice` (同じサーバー内の□なるコンテキスト(= application)のURLを呼び出すことができます。)
  - プロトコル相□URL: `//code.jquery.com/jquery-2.0.3.min.js`

Thymeleafでは□□URLはどんな□合でも使用できますが、相□URLを使用する□合は `IWebContext` を□装したコンテキスト・オブジェクトが必要です。そのコンテキスト・オブジェクトを使用して、相□リンクを生成するための情□をHTTPリクエスト内から取得します。

ではこの新しい□文を使ってみましょう。 `th:href` 属性で使用します:

```
<!-- Will produce 'http://localhost:8080/gtvg/order/details?orderId=3' (plus rewriting) -->
<a href="details.html"
   th:href="@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/gtvg/order/details?orderId=3' (plus rewriting) -->
<a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/gtvg/order/3/details' (plus rewriting) -->
<a href="details.html" th:href="@{/order/{orderId}/details(orderId=${o.id})}">view</a>
```

いくつか注意点:

- `th:href` は属性□更用の属性です: リンクURLを生成し `<a>` タグのhref属性にセットします。
- URLパラメータを指定することができます(`orderId=${o.id}` の部分です)。自 □的にURLエンコーディングされます。
- □数のパラメータを指定する□合はカンマ区 切りで指定できます `@{/order/process(execId=${execId},execType='FAST')}`
- URLパス内でも□数式は使用可能です `@{/order/{orderId}/details(orderId=${orderId})}`
- `/` で始まる相□URL(`/order/details`)については、自 □的にアプリケーション・コンテキスト名を前に付けます。
- クッキーが使用できない□合、またはまだ分からない□合は `";jsessionid=..."` を相□URLの最後につけてセッションをキープできるようにすることがあります。これは *URL Rewriting* と呼ばれていますが、Thymeleafでは全てのURLに□してサーブレットAPIの `response.encodeURL(...)` のメカニズムを使用して独自 リライトフィルタを追加することができます。
- `th:href` タグを使用する□合、(任意ですが)静的な `href` 属性をテンプレートに同 □に指定することができます。そうすることでプロトタイプ用途などで直接テンプレートをブラウザで開いた□合でもリンクを有 □にすることができます。

メッセージ□文(`#{...}`)のときと同 □に、URL□文でも他の式の□□□果が使用可能です。

```
<a th:href="@{${url}(orderId=${o.id})}">view</a>
<a th:href="@{'/details/'+${user.login}(orderId=${o.id})}">view</a>
```

私的なホームページ用のメニュー

リンクURLの作成方法がわかったので、ホームページ内の他のページへの小さなメニューを加えてみましょうか。

```
<p>Please select an option</p>
<ol>
  <li><a href="product/list.html" th:href="@{/product/list}">Product List</a></li>
  <li><a href="order/list.html" th:href="@{/order/list}">Order List</a></li>
  <li><a href="subscribe.html" th:href="@{/subscribe}">Subscribe to our Newsletter</a></li>
  <li><a href="userprofile.html" th:href="@{/userprofile}">See User Profile</a></li>
</ol>
```

4.5 リテラル

テキスト リテラル

テキストリテラルはシングルクォートで囲まれた文字列です。どんな文字でも大丈夫ですが、シングルクォート自体は `\'` のようにエスケープしてください。

```
<p>
  Now you are looking at a <span th:text="'working web application'">template file</span>.
</p>
```

数値リテラル

数値リテラルは数字そのままです。

```
<p>The year is <span th:text="2013">1492</span>.</p>
<p>In two years, it will be <span th:text="2013 + 2">1494</span>.</p>
```

真偽値リテラル

真偽値リテラルは `true` と `false` です:

```
<div th:if="${user.isAdmin()} == false"> ...
```

ここで注意して欲しいのは、 `== false` が括弧の外にあるということです。この場合はThymeleaf自身が処理します。もし括弧の中にある場合は、OGNL/SpringELのエンジンが処理を担当します。

```
<div th:if="${user.isAdmin() == false}"> ...
```

null値リテラル

`null` リテラルも使用可能です:

```
<div th:if="${variable.something} == null"> ...
```

リテラルトークン

数値、真偽値、null値リテラルは、いわば「リテラルトークン」の特定のケースなのです。

このリテラルトークンは、スタンダード式を少しだけシンプルにしてくれます。テキストリテラル(`'...'`)と全く同じ働きをしますが、次の文字しか使用できません: 文

字(A-Z and a-z)、数字(0-9)、括弧([と])、ドット(.)、ハイフン(-)アンダースコア(_)。ですので、空白文字やカンマ等は使用できません。

この利点は何でしょうか?それはトークンはクォートでむ必要がないという点です。ですので、次のようにく代わりに:

```
<div th:class="'content'">...</div>
```

こうくことができます:

```
<div th:class="content">...</div>
```

4.6 テキスト の追加

テキストは + 演算子で追加できます。文字列リテラルであっても、やメッセージ式の果であっても大丈夫です:

```
th:text="'The name of the user is ' + ${user.name}"
```

4.7 リテラル置

リテラル置を使用すると数の数から文字列を作成するフォーマットがになります。'...' + '...' のようにリテラルを追加する必要がありません。

リテラル置を使用する合は、棒(|)でみます:

```
<span th:text="|Welcome to our application, ${user.name}!|">
```

これは以下の内容と同じです:

```
<span th:text="'Welcome to our application, ' + ${user.name} + '!'">
```

リテラル置は他の式とみ合わせて使用することができます:

```
<span th:text="${onevar} + ' ' + |${twovar}, ${threevar}|">
```

注意点:リテラル置(|...|)内で使用可能なのは、数式(${...})だけです。他のリテラル('...')や真/数トークンや条件式などは使用できません。

4.8 算演算子

いくつかの算演算子が使用可能です: +, -, *, /, %

```
th:with="isEven=(${prodStat.count} % 2 == 0)"
```

この演算子はOGNL数式の中でも使用可能なことに注意して下さい(その合はThymeleafスタンダード式エンジンの代わりにOGNLによって算されます)。

```
th:with="isEven=${prodStat.count % 2 == 0}"
```

いくつかの演算子には文字列エイリアスもあります: div (/), mod (%)

4.9 比演算子と 等演算子

式の中の□は `>`, `<`, `>=`, `<=` シンボルで比□できます。また `==` と `!=` 演算子で等□性を□□できます。ただし、XMLの属性□には `<` と `>` を使用すべきではないと策定されていますので、代わりに `<` と `>` を使用すべきです。

```
th:if="${prodStat.count} &gt; 1"
th:text="'Execution mode is ' + ( (${execMode} == 'dev')? 'Development' : 'Production')"
```

文字列エイリアスもあります: `gt` (`>`), `lt` (`<`), `ge` (`>=`), `le` (`<=`), `not` (`!`), `eq` (`==`), `neq`/`ne` (`!=`)。

4.10 条件式

「条件式」は条件(それ自 体が□の式です)を□□した□果によって、2つのうちのどちらかの式を□□することを意味します。

例を□てみましょう(今回は `th:class` という「属性□更子」を使用しますね):

```
<tr th:class="${row.even}? 'even' : 'odd'">
  ...
</tr>
```

条件式の3つのパーツ全て(`condition`, `then` and `else`)がそれぞれ式になっています。つまり、□数(`${...}`, `*{...}`)やメッセージ(`#{...}`)や URL(`@{...}`)やリテラル(`'...'`)を使うことができるということです。

条件式は括弧で□むことでネスト可能です:

```
<tr th:class="${row.even}? (${row.first}? 'first' : 'even') : 'odd'">
  ...
</tr>
```

Else式は省略可能です。その□合、条件がfalseのときにはnull□が返されます。

```
<tr th:class="${row.even}? 'alt'">
  ...
</tr>
```

4.11 デフォルト 式 (エルビス演算子)

「デフォルト式」は then」のない特□な条件式です。Groovyなどの「エルビス演算子」と同じです。2つの式を指定して最初の式がnullを返した□合にのみ2番目 の式の□が□□されます。

□□にユーザープロフィールページを□てみましょう:

```
<div th:object="${session.user}">
  ...
  <p>Age: <span th:text="*{age}?: '(no age specified)'">27</span>.</p>
</div>
```

演算子は `?:` です。年□(`*{age}`)がnullの□合にのみ□外(今回はリテラル□)を表示します。つまり、以下の内容と同じです:

```
<p>Age: <span th:text="*{age != null}? *{age} : '(no age specified)'">27</span>.</p>
```

括弧で□むことでネスト可能です:

```
<p>
  Name:
  <span th:text="*{firstName}?: (*{admin}? 'Admin' : #{default.username})">Sebastian</span>
</p>
```

4.12 プリプロセッシング

ここまで□できた式に加えて、Thymeleafは□プリプロセッシング□式を提供します。

プリプロセッシングとはどういうことでしょうか?それは、通常の式よりも先に□□されるということです。それによって、最□的に□行される□□の式の□更をすることができます。

プリプロセッシング式は普通の式と全く同じように□くことができますが、二重のアンダースコアで□まれています(`__${expression}__`)。

i18nの `Messages_fr.properties` のエントリに言□特有のスタティックメソッドを呼び出すようなOGNL式が含まれているとしましょう:

```
article.text=@myapp.translator.Translator@translateToFrench({0})
```

... `Messages_es.properties` の□□する部分:

```
article.text=@myapp.translator.Translator@translateToSpanish({0})
```

ロケールに□じた式を□□してマークアップを作成する必要があるので、まずは□プリプロセッシング□の式を□□して、その次にThymeleafにそれを□行させます:

```
<p th:text="${__#{article.text('textVar')}__}">Some text here...</p>
```

フランス□ロケールの□合のプリプロセッシングは次と同等になります:

```
<p th:text="${@myapp.translator.Translator@translateToFrench(textVar)}">Some text here...</p>
```

プリプロセッシング用文字列 `__` は属性の中では `\_\_` とエスケープします。

5 属性を定する

この章ではThymeleafでどのようにしてマークアップタグ内の属性を定(または更)するかを明します。タグのボディの内容を定する能の次に必要な基本能かもしれません。

5.1 任意の属性にを定する

私たちのウェブサイトでニュースレターを行するとしましょう。ユーザーが□□できるようにしたいので `/WEB-INF/templates/subscribe.html` テンプレートにフォームを置します:

```
<form action="subscribe.html">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe me!" />
  </fieldset>
</form>
```

これで全然□□ないようにえます。しかし□□はこのファイルはウェブアプリケーションのテンプレートというよりは静的なXHTMLにえます。まず、action属性がこのテンプレートファイル自身への静的リンクなので、URLをきえる方法がありません。次に、submitボタンのvalue属性は英で表示されますが多言□□□したいですよね。

ということで `th:attr` 属性を使いましょう。これで、タグの中の属性を更することができます。

```
<form action="subscribe.html" th:attr="action=@{/subscribe}">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe me!" th:attr="value=#{subscribe.submit}"/>
  </fieldset>
</form>
```

コンセプトは非常に直感的です: `th:attr` には□□に属性にを代入する式をきます。□□するコントローラーやメッセージファイルを作成することによって、想定通りの理果が得られます:

```
<form action="/gtvg/subscribe">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="¡Suscríbeme!"/>
  </fieldset>
</form>
```

新しい属性の□が使用されていることに加えて `/gtvg/subscribe` のURLには、既に□明したようにアプリケーションコンテキスト名が自□的に付け加えられています。

同□に□数の属性にを定したい□合はどうしたらよいでしょうか?XMLでは1つのタグの中に同じ属性を2つ以上くことはできませんので `th:attr` にカンマ区切りのリストを指定します:

```
<img src="../../images/gtvglogo.png"
     th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />
```

メッセージファイルを用意すれば、このような出力になります:

```
<img src="/gtgv/images/gtvglogo.png" title="Logo de Good Thymes" alt="Logo de Good Thymes" />
```

5.2 特定の属性にを定する

ここまでで、次のような書き方はすごくいやぁと思っているかもしれませんね:

```
<input type="submit" value="Subscribe me!" th:attr="value=#{subscribe.submit}"/>
```

属性の中で値を設定するというのはとても実用的ではありますが、常にそうしないといけないというのは正式ではありません。

ですよね。なので実のところ `th:attr` 属性はテンプレート内ではほとんど使われません。通常は `th:*` 属性を使用します。この属性を使用すると(`th:attr` のような任意の属性ではなく)特定のタグ属性に値を設定することができます。

ではスタンダードダイアレクトでボタンの `value` 属性に値を設定するにはどのような属性を使用すればいいのでしょうか?これはかなり分かりやすいと思います。`th:value` です。では試してみましょう:

```
<input type="submit" value="Subscribe me!" th:value="#{subscribe.submit}"/>
```

この方が全然良いですよね!同様に `form` タグの `action` 属性も試してみましょう:

```
<form action="subscribe.html" th:action="@{/subscribe}">
```

`th:href` 属性を `home.html` で使用したのを覚えていますか?これも同じです:

```
<li><a href="product/list.html" th:href="@{/product/list}">Product List</a></li>
```

このような属性が非常にたくさん用意されていて、それぞれが特定のXHTMLやHTML5のタグを象徴しています:

th:abbr	th:accept	th:accept-charset
th:accesskey	th:action	th:align
th:alt	th:archive	th:audio
th:autocomplete	th:axis	th:background
th:bgcolor	th:border	th:cellpadding
th:cellspacing	th:challenge	th:charset
th:cite	th:class	th:classid
th:codebase	th:codetype	th:cols
th:colspan	th:compact	th:content
th:contenteditable	th:contextmenu	th:data
th:datetime	th:dir	th:draggable
th:dropzone	th:enctype	th:for
th:form	th:formaction	th:formenctype
th:formmethod	th:formtarget	th:frame
th:frameborder	th:headers	th:height
th:high	th:href	th:hreflang
th:hspace	th:http-equiv	th:icon
th:id	th:keytype	th:kind
th:label	th:lang	th:list
th:longdesc	th:low	th:manifest
th:marginheight	th:marginwidth	th:max
th:maxlength	th:media	th:method
th:min	th:name	th:optimum
th:pattern	th:placeholder	th:poster
th:preload	th:radiogroup	th:rel
th:rev	th:rows	th:rowspan
th:rules	th:sandbox	th:scheme
th:scope	th:scrolling	th:size

th:sizes	th:span	th:spellcheck
th:src	th:srclang	th:standby
th:start	th:step	th:style
th:summary	th:tabindex	th:target
th:title	th:type	th:usemap
th:value	th:valuetype	th:vspace
th:width	th:wrap	th:xmlbase
th:xmllang	th:xmlspace	

5.3 □数の□を同□に□定する

ここでは2つのちょっと特□な属性を□介します。`th:alt-title` と `th:lang-xmllang` です。2つの属性は同じ□を同□に指定することができます。具体的には:

- `th:alt-title` は `alt` と `title` を□定します。
- `th:lang-xmllang` は `lang` と `xml:lang` を□定します。

私たちのGTVGホームページで次のように□いている部分は:

```
<img src="../../images/gtvglogo.png"
     th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />
```

このように□くこともできますし:

```
<img src="../../images/gtvglogo.png"
     th:src="@{/images/gtvglogo.png}" th:title="#{logo}" th:alt="#{logo}" />
```

このように□くこともできます:

```
<img src="../../images/gtvglogo.png"
     th:src="@{/images/gtvglogo.png}" th:alt-title="#{logo}" />
```

5.4 前後に追加する

`th:attr` と同じように任意の属性に□して作用するものとして、Thymeleafには `th:attrappend` と `th:attrprepend` 属性があります。既存の属性□の前や後ろに□□□果を付け加えるための属性です。

例えばある物に□して、ユーザーが何をしたかによって□なるCSSクラスを追加(□定ではなく追加)したい□合が考えられます。これは□□です:

```
<input type="button" value="Do it!" class="btn" th:attrappend="class=${' ' + cssStyle}" />
```

`cssStyle` □数に `"warning"` という□を□定してテンプレートを□理すると次の□果が得られます:

```
<input type="button" value="Do it!" class="btn warning" />
```

スタンダードダイアレクトには2つの特□な属性追加用の属性があります: `th:classappend` と `th:styleappend` です。CSSクラスや *style* の一部を既存のものを上□きせずに追加します:

```
<tr th:each="prod : ${prods}" class="row" th:classappend="${prodStat.odd}? 'odd'">
```

(`th:each` 属性のことは心配しないでください。「□り返し用の属性」として後ほど□明します。)

5.5 固定ブール属性

XHTML/HTML5属性の中には、値を持つか、その属性自体が存在しないかのどちらか、という特別な属性があります。

例えば `checked` です:

```
<input type="checkbox" name="option1" checked="checked" />
<input type="checkbox" name="option2" />
```

XHTMLでは `checked` 属性には `"checked"` という値しか指定できません(HTML5では少し緩いですが)。`disabled`, `multiple`, `readonly` と `selected` も同様です。

これらの属性に対して条件の結果によって値を設定するための属性を、スタンダードダイアレクトでは提供しています。条件の評価結果がtrueの場合はその固定値が設定され、falseの場合は属性自体が設定されません:

```
<input type="checkbox" name="active" th:checked="${user.active}" />
```

スタンダードダイアレクトには次のような固定ブール属性があります:

th:async	th:autofocus	th:autoplay
th:checked	th:controls	th:declare
th:default	th:defer	th:disabled
th:formnovalidate	th:hidden	th:ismap
th:loop	th:multiple	th:novalidate
th:nowrap	th:open	th:pubdate
th:readonly	th:required	th:reversed
th:scoped	th:seamless	th:selected

5.6 HTML5フレンドリな属性や要素名のサポート

よりHTML5フレンドリーな書き方もできます。これは全く異なる構文になります。

```
<table>
    <tr data-th-each="user : ${users}">
        <td data-th-text="${user.login}">...</td>
        <td data-th-text="${user.name}">...</td>
    </tr>
</table>
```

`data-{prefix}-{name}` 構文は、`th:*` などの名前空間を使用せずに独自属性を書くためのHTML5での標準的な方法です。Thymeleafでは、(スタンダードダイアレクトだけでなく)全てのダイアレクトでこの構文を使用することができます。

`{prefix}-{name}` という形式で独自タグを指定するための構文もあります。これは *W3C Custom Elements specification* (より大きな *W3C Web Components spec* の一部です)に準拠しています。例えば `th:block` 要素(または `th-block`)で使用することができますが、これについては後述します。

重要: この構文は名前空間を使用した `th:*` に追加された機能であって、置き換えるものではありません。将来的に名前空間構文を非推奨にする意図は全くありません。

6 □り返し□理

ここまでホームページとしてユーザープロフィールページと、ニュースレター□□ページを作ってきましたですが、商品についてはどうでしょう?□□者に私たちの商品を知ってもらうための商品一□ページを作るべきではないでしょうか?ええ、明らかにYesですね。ではそうしましょう。

6.1 □り返し□理の基□

`/WEB-INF/templates/product/list.html` ページに商品一□を□□するためにテーブルが必要です。1行(`<tr>` 要素)に1商品ずつ表示したいので、テンプレートの中にはテンプレート行」(各商品がどのように表示されるかを示す行)を作って、それをThymeleafで商品ごとに□り返す必要があります。

スタンダードダイアレクトにはそのための属性があります。`th:each` です。

th:each を使用する

商品一□ページのコントローラーはサービスレイヤから商品一□を取得してテンプレートコンテキストにそれを追加します:

```
public void process(
        HttpServletRequest request, HttpServletResponse response,
        ServletContext servletContext, TemplateEngine templateEngine) {

    ProductService productService = new ProductService();
    List<Product> allProducts = productService.findAll();

    WebContext ctx = new WebContext(request, servletContext, request.getLocale());
    ctx.setVariable("prods", allProducts);

    templateEngine.process("product/list", ctx, response.getWriter());
}
```

では商品リストを□り返し□理するために `th:each` を使いましょう:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <h1>Product list</h1>

    <table>
      <tr>
        <th>NAME</th>
        <th>PRICE</th>
        <th>IN STOCK</th>
      </tr>
      <tr th:each="prod : ${prods}">
        <td th:text="${prod.name}">Onions</td>
        <td th:text="${prod.price}">2.41</td>
        <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
      </tr>
    </table>

    <p>
      <a href="../home.html" th:href="@{/}">Return to home</a>
    </p>
```

```
  </body>

</html>
```

上の `prod : ${prods}` 属性は `${prods}` の結果の各要素について、その要素をprodという変数に収めて、このテンプレートのフラグメントを繰り返し処理する」という意味になります。呼び名を決めておきましょう。

- ここでは `${prods}` のことを「被繰り返し式」または「被繰り返し変数」と呼びます。[3]
- ここでは `prod` のことを「繰り返し変数」と呼びます。

繰り返し変数 `prod` は `<tr>` 要素の内部だけで使用できることに注意してください。(`<td>` のような内部のタグでも使用可能です)

繰り返し処理が可能なもの

Thymeleafの繰り返し処理で使用可能なのは `java.util.List` だけではありません。実は `th:each` ではオブジェクト一式が「繰り返し可能」だとみなされます。

- `java.util.Iterable` を実装しているオブジェクト
- `java.util.Map` を実装しているオブジェクト。マップを繰り返し処理する場合の繰り返し変数は `java.util.Map.Entry` のクラスになります。
- 配列
- その他のオブジェクトは、そのオブジェクト自身のみを要素として持つ、1要素だけのリストのように扱われます。

6.2 繰り返しステータスの保持

`th:each` を使用する際に、繰り返し処理中のステータスを知るための便利な仕組みがThymeleafにはあります:「ステータス変数」です。

ステータス変数は `th:each` 属性の中で定義され、次の内容を保持しています:

- `index` プロパティ: 0始まりの現在の「繰り返しインデックス」
- `count` プロパティ: 1始まりの現在の「繰り返しインデックス」
- `size` プロパティ: 被繰り返し変数の全要素数
- `current` プロパティ: 繰り返し中の「繰り返し変数」
- `even/odd` 真偽プロパティ: 現在の繰り返し処理が偶数か奇数か
- `first` 真偽プロパティ: 現在の繰り返し処理が最初かどうか
- `last` 真偽プロパティ: 現在の繰り返し処理が最後かどうか

ではどのように使用するのかを前回の例で試してみましょう:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
  </tr>
  <tr th:each="prod,iterStat : ${prods}" th:class="${iterStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
  </tr>
</table>
```

ご覧のとおり `th:each` 属性の中で、繰り返し変数の後ろにカンマで区切って名前を書いてステータス変数(この例では `iterStat`)を定義します。繰り返し変数と同様、ステータス変数も `th:each` 属性を持っているタグによって定義されたフラグメントの内部でのみ使用可能です。

それでは、テンプレートの処理結果を見てみましょう:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
    <link rel="stylesheet" type="text/css" media="all" href="/gtvg/css/gtvg.css" />
  </head>

  <body>

    <h1>Product list</h1>

    <table>
      <tr>
        <th colspan="1" rowspan="1">NAME</th>
        <th colspan="1" rowspan="1">PRICE</th>
        <th colspan="1" rowspan="1">IN STOCK</th>
      </tr>
      <tr>
        <td colspan="1" rowspan="1">Fresh Sweet Basil</td>
        <td colspan="1" rowspan="1">4.99</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
      <tr class="odd">
        <td colspan="1" rowspan="1">Italian Tomato</td>
        <td colspan="1" rowspan="1">1.25</td>
        <td colspan="1" rowspan="1">no</td>
      </tr>
      <tr>
        <td colspan="1" rowspan="1">Yellow Bell Pepper</td>
        <td colspan="1" rowspan="1">2.50</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
      <tr class="odd">
        <td colspan="1" rowspan="1">Old Cheddar</td>
        <td colspan="1" rowspan="1">18.75</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
    </table>

    <p>
      <a href="/gtvg/" shape="rect">Return to home</a>
    </p>

  </body>

</html>
```

□り返しステータス□数は完璧に□いていますね。`odd` CSSクラスが奇数行のみに□用されています(行番号は0から始まります)。

> colspanとrowspan属性が `<td>` タグに追加されていますが、これは `<a>` のshape属性と同□に □□されている *XHTML 1.0 Strict* □□のDTDに□□ってThymeleafが自 □的に追加します。*XHTML 1.0 Strict* □□では、これらの□が属性のデフォルト□として策定されています(テンプレートでは□を□定していないことに注意してください)。ページの表示には影□はないので、このことを□にする必要は全然ありません。例えば、HTML5(にはDTDがありませんが)を使用していたら、この属性は□して追加されません。

ステータス□数を明示的に指定しない□合は、□り返し□数の後ろに `Stat` をつけた□数名をThymeleafはいつでも作成します:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
  </tr>
</table>
```

7 条件の□□

7.1 □□な条件: “if” と “unless”

特定の条件が□たされる□合にのみ、フラグメントを表示したい□合があるでしょう。

例えば、商品テーブルの各商品に□してコメント数を表示するカラムを用意する□合を想像してみてください。もしコメントがあれば、その商品のコメント□□ページへのリンクを□りたいです。

この□合 `th:if` 属性を使用します:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:if="${not #lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
</table>
```

□□□山のことをやっているので、重要な行にフォーカスしましょう:

```
<a href="comments.html"
   th:href="@{/product/comments(prodId=${prod.id})}"
   th:if="${not #lists.isEmpty(prod.comments)}">view</a>
```

□□、ほとんど□明することはないですね: 商品の `id` を `prodId` パラメータに□定してコメントページ(`/product/comments`)へのリンクを作成します。でもそれは商品にコメントがついている□合だけです。

では、□果のマークアップを□てみましょう(□やすくするために、デフォルト属性の `rowspan` と `colspan` は取り除いています):

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
```

```
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Old Cheddar</td>
    <td>18.75</td>
    <td>yes</td>
    <td>
      <span>1</span> comment/s
      <a href="/gtvg/product/comments?prodId=4">view</a>
    </td>
  </tr>
</table>
```

かしら! まさに欲しかったものです。

`th:if` 属性は *boolean* 条件のみを□□するわけではないことに注意して下さい。もう少し幅□いのです。次のようなルールに□って指定された式を `true` と□□します:

- □が null ではない□合:
    - boolean の `true`
    - 0以外の数□
    - 0以外の文字
    - “false” でも “off” でも “no” でもない文字列
    - 真□□でも、数□でも、文字でも文字列でもない□合
- (□が null の□合は th:if は false と□□します).

また、`th:if` には反□の意味で□になるものがあります。`th:unless` です。先ほどの例で、OGNL式の `not` を使用する代わりに、これを使用することもできます。

```
<a href="comments.html"
   th:href="@{/comments(prodId=${prod.id})}"
   th:unless="${#lists.isEmpty(prod.comments)}">view</a>
```

7.2 スイッチ文

Javaにおける *switch* □造と同じように使用して、コンテンツを条件□に表示する方法もあります: `th:switch` / `th:case` 属性のセットです。

ご想像通りの□きをします:

```
<div th:switch="${user.role}">
  <p th:case="'admin'">User is an administrator</p>
  <p th:case="#{roles.manager}">User is a manager</p>
</div>
```

一つの `th:case` 属性が `true` と□□されるとすぐに、同じスイッチコンテキスト内の他の全ての `th:case` 属性は `false` と□□されることに注意してください。

デフォルトオプションは `th:case="*"` で指定します:

```
<div th:switch="${user.role}">
  <p th:case="'admin'">User is an administrator</p>
  <p th:case="#{roles.manager}">User is a manager</p>
  <p th:case="*">User is some other thing</p>
</div>
```

8 テンプレートレイアウト

8.1 テンプレート フラグメント のインクルード

フラグメント の定義と参照

他のテンプレートのフラグメントを自分のテンプレートにインクルードしたいという場合がよくあります。よく使われるのはフッターやヘッダー、メニューなどです。

そうするためにThymeleafではインクルード可能なフラグメントを定義する必要があります。定義には `th:fragment` 属性を使用します。

私たちの食料品店の全てのページに著作的なコピーライトフッターを追加したいとしましょう。`/WEB-INF/templates/footer.html` ファイルにこのようなコードを定義します:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <body>

    <div th:fragment="copy">
      &copy; 2011 The Good Thymes Virtual Grocery
    </div>

  </body>

</html>
```

このコードは `copy` と呼ばれるフラグメントを定義しており、私たちのホームページで `th:include` または `th:replace` 属性のどちらかを使用して自分にインクルードすることができます:

```
<body>

  ...

  <div th:include="footer :: copy"></div>

</body>
```

これらのインクルード属性の構文は両方ともとても直感的です。そのフォーマットには3種類あります:

- `"templatename::domselector"` またはそれと同等の `templatename::[domselector] templatename` という名前のテンプレート内にある、DOMセレクターで指定されたフラグメントをインクルードします。
    - `domselector` はフラグメント名でも大丈夫なので上の例の `footer :: copy` のように単に `templatename::fragmentname` を指定することもできることに注意してください。

    > DOMセレクター構文はXPath表記やCSSセレクターと似ています。詳しくは Appendix C を参照してください。

- `"templatename" templatename` という名前のテンプレート全体をインクルードします。

    > `th:include`/`th:replace` タグで使用されるテンプレート名は現在テンプレートエンジンで使用されているテンプレートリゾルバによって解決可能でなければならないことに注意してください。

- `::domselector"` or `"this::domselector"` 同じテンプレート内のフラグメントをインクルードします。

上の例の `templatename` と `domselector` には両方とも式を指定することができます(条件式でも大丈夫です!):

```
<div th:include="footer :: (${user.isAdmin}? #{footer.admin} : #{footer.normaluser})"></div>
```

フラグメントにはどんな `th:*` 属性でも含めることができます。これらの属性は包含テンプレート(`th:include` / `th:replace` 属性が書かれたテンプレートのことです)にそのフラグメントがインクルードされるときに1度評価されます。フラグメント内の属性は、包含テンプレート内のコンテキスト変数を参照することができます。

> フラグメントに関するこのアプローチの大きな利点は、完全かつ妥当なXHTML構造によって、ブラウザで完全に表示できるフラグメントを書くことができるという点です。Thymeleafを使って他のテンプレートにインクルードすることができるのです。

`th:fragment` を使用せずフラグメントを参照する

さらに、DOMセレクターのパワーのおかげで、 `th:fragment` 属性を使わなくてもフラグメントをインクルードすることができます。全くThymeleafのことを知らない他のアプリケーションのマークアップコードでさえもインクルードすることができます。

```
...
<div id="copy-section">
  &copy; 2011 The Good Thymes Virtual Grocery
</div>
...
```

このフラグメントを単に `id` 属性によってCSSセレクターに似た方法で参照することができます。

```
<body>

  ...

  <div th:include="footer :: #copy-section"></div>

</body>
```

`th:include` と `th:replace` の違い

では、 `th:include` と `th:replace` の違いって何でしょうか? `th:include` はホストタグの中にフラグメントの中身をインクルードする一方で `th:replace` は実際にホストタグをフラグメントで置換します。ですので、このようなHTML5フラグメントに対して:

```
<footer th:fragment="copy">
  &copy; 2011 The Good Thymes Virtual Grocery
</footer>
```

...ホストとなる `<div>` タグを2個用いてインクルードしてみます:

```
<body>

  ...

  <div th:include="footer :: copy"></div>
  <div th:replace="footer :: copy"></div>

</body>
```

...するとこのような結果になります:

```
<body>

  ...

  <div>
    &copy; 2011 The Good Thymes Virtual Grocery
  </div>
  <footer>
    &copy; 2011 The Good Thymes Virtual Grocery
```

```
  </footer>

</body>
```

th:substituteby 属性は th:replace 属性に□するエイリアスとして使用できますが、後者を推□します。th:substituteby は将来のバージョンで非推□になるかもしれないことに注意してください。

8.2 パラメ□タ化可能なフラグメント シグネチャ

テンプレートフラグメントを、より□数のようなメカニズムで作成するために th:fragment で定□されたフラグメントは、パラメータを持つことができます:

```
<div th:fragment="frag (onevar,twovar)">
    <p th:text="${onevar} + ' - ' + ${twovar}">...</p>
</div>
```

th:include や th:replace からこのフラグメントを呼び出す□合には、以下の2つのどちらかの□文を使用します:

```
<div th:include="::frag (${value1},${value2})">...</div>
<div th:include="::frag (onevar=${value1},twovar=${value2})">...</div>
```

後者の□合は、□番が重要ではないことに注意してください:

```
<div th:include="::frag (twovar=${value2},onevar=${value1})">...</div>
```

フラグメント シグネチャなしフラグメント ロ□カル□数

シグネチャなしでフラグメントが定□されている□合でも:

```
<div th:fragment="frag">
    ...
</div>
```

上□の後者の□文を使うことができます(後者の□文だけです):

```
<div th:include="::frag (onevar=${value1},twovar=${value2})">
```

□□のところ、これは th:include と th:with を□み合わせて使ったのと同じことです:

```
<div th:include="::frag" th:with="onevar=${value1},twovar=${value2}">
```

注意 シグネチャの有□にかかわらず、フラグメントに□するローカル□数のこの仕□によってエディタが□行前に空になるというようなことはありません。この□合でもフラグメントは呼び出し元のテンプレートと同じように、全てのエディタ□数にアクセスすることができます。

テンプレ□ト 内でサ□ションの□□ th:assert

th:assert 属性に□して、全ての□□が true になるはずの式をカンマ区切りの□で指定すると、もしそうならない□合には例外を投げます。

```
<div th:assert="${onevar},(${twovar} != 43)">...</div>
```

これを使うと、フラグメントシグネチャで□□にパラメータをバリデートすることができます:

```
<header th:fragment="contentheader(title)"
th:assert="${!#strings.isEmpty(title)}">...</header>
```

8.3 テンプレート フラグメント の削除

私たちの商品リストテンプレートの最新バージョンをもう一度見てみましょう:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
</table>
```

このコードはテンプレートとしては全然問題ありませんが、静的ページ(Thymeleafで処理をせずに直接ブラウザで開いた場合)としては良いホームページではなさそうです。

なぜでしょうかブラウザで完全に表示できはしますが、テーブルにはモックデータの1行しかないからです。ホームページとしては内容にリアルさが足りません... 2つ以上の商品を表示するほうが良かったですね。複数行必要ですよね。

ということで、追加しましょう:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
  <tr class="odd">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr>
    <td>Mild Cinnamon</td>
    <td>1.99</td>
    <td>yes</td>
    <td>
      <span>3</span> comment/s
      <a href="comments.html">view</a>
    </td>
  </tr>
</table>
```

よし、これで3商品になったので、ホームページとしてはこのほうが全然良いです。でも...Thymeleafで処理したらどうなるでしょうか?:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Old Cheddar</td>
    <td>18.75</td>
    <td>yes</td>
    <td>
      <span>1</span> comment/s
      <a href="/gtvg/product/comments?prodId=4">view</a>
    </td>
  </tr>
  <tr class="odd">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr>
    <td>Mild Cinnamon</td>
    <td>1.99</td>
    <td>yes</td>
    <td>
      <span>3</span> comment/s
      <a href="comments.html">view</a>
    </td>
  </tr>
</table>
```

最後の2行がモック行です! あぁ、そう。そうです: □り返し□理は最初の行にしか□用されませんので、Thymeleafが他の2行を削除する理由がありません。

テンプレート□理をする□にこの2行を削除する手段が必要です。`th:remove` 属性を2つ目 と 3つ目 の `<tr>` に使用しましょう:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
```

```
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
  <tr class="odd" th:remove="all">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr th:remove="all">
    <td>Mild Cinnamon</td>
    <td>1.99</td>
    <td>yes</td>
    <td>
      <span>3</span> comment/s
      <a href="comments.html">view</a>
    </td>
  </tr>
</table>
```

テンプレートを□理すると、正しく□くように□りましたね:

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Old Cheddar</td>
    <td>18.75</td>
    <td>yes</td>
    <td>
      <span>1</span> comment/s
      <a href="/gtvg/product/comments?prodId=4">view</a>
    </td>
  </tr>
</table>
```

この属性に□する all という□はどうなっているのでしょう?何を意味するのでしょうか?ねぃ、□□のところ th:remove はその□によって、5つの□なる振る舞いをします:

- all:この属性を含んでいるタグとその全ての子の□方を削除します。
- body:この属性を含んでいるタグは削除せずに、全ての子を削除します。
- tag:この属性を含んでいるタグは削除しますが、子は削除しません。
- all-but-first:最初の子以外の全ての子を削除します。
- none:何もしません。この□は、□的な□□の□合に有用です。

all-but-first □は何の役に立つのでしょう?それはホロタイプにく th:remove="all" を□らしてくれます:

```
<table>
  <thead>
    <tr>
      <th>NAME</th>
      <th>PRICE</th>
      <th>IN STOCK</th>
      <th>COMMENTS</th>
    </tr>
  </thead>
  <tbody th:remove="all-but-first">
    <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
      <td th:text="${prod.name}">Onions</td>
      <td th:text="${prod.price}">2.41</td>
      <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
      <td>
        <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
        <a href="comments.html"
           th:href="@{/product/comments(prodId=${prod.id})}"
           th:unless="${#lists.isEmpty(prod.comments)}">view</a>
      </td>
    </tr>
    <tr class="odd">
      <td>Blue Lettuce</td>
      <td>9.55</td>
      <td>no</td>
      <td>
        <span>0</span> comment/s
      </td>
    </tr>
    <tr>
      <td>Mild Cinnamon</td>
      <td>1.99</td>
      <td>yes</td>
      <td>
        <span>3</span> comment/s
        <a href="comments.html">view</a>
      </td>
    </tr>
  </tbody>
</table>
```

th:remove 属性は□可された文字列□(all, tag, body, all-but-first または none)を返すのであればどんな Thymeleafスタンダード式」でも指定することができます。

つまり、削除に条件を□用することもできるということです:

```
<a href="/something" th:remove="${condition}? tag : none">Link text not to be removed</a>
```

また、th:remove は null を none と同□の□名とみなすため、次のような□合は上□の例と全く同じ□きをします:

```
<a href="/something" th:remove="${condition}? tag">Link text not to be removed</a>
```

この□合、${condition} がfalseの□合、null が返されるので、何も削除されません。

9 ローカル変数

Thymeleafではテンプレートの特定のフラグメントに対して定義され、そのフラグメント内でのみ参照可能な変数のことを「ローカル変数」と呼びます。

既に見たことのある例を挙げると、商品リストページの繰り返し変数 `prod` がそれにあたります。

```
<tr th:each="prod : ${prods}">
    ...
</tr>
```

`prod` 変数は `<tr>` タグの範囲だけで有効です。具体的には:

- そのタグの中で `th:each` より優先順位が下の `th:*` 属性全てで使用することができます(優先順位が下の属性とは `th:each` より後に評価される属性という意味です)。
- `<tr>` タグの子要素、例えば `<td>` などでも使用可能です。

Thymeleafには、繰り返し処理以外でもローカル変数を定義する方法があります。`th:with` 属性です。その構文は属性値の代入の構文に似ています:

```
<div th:with="firstPer=${persons[0]}">
  <p>
    The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.
  </p>
</div>
```

`th:with` が処理されると、`firstPer` 変数がローカル変数として作成されコンテキストの変数マップに追加されます。そして、コンテキスト内で最初から定義されている他の変数と同様にアクセス可能になります。ただし、`<div>` タグの範囲だけです。

複数の変数を同時に定義したい場合は、普通に変数の代入をする構文を使用することができます:

```
<div th:with="firstPer=${persons[0]},secondPer=${persons[1]}">
  <p>
    The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.
  </p>
  <p>
    But the name of the second person is
    <span th:text="${secondPer.name}">Marcus Antonius</span>.
  </p>
</div>
```

`th:with` 属性ではその属性内で定義された変数の再利用ができます:

```
<div th:with="company=${user.company + ' Co.'},account=${accounts[company]}">...</div>
```

それでは私たちの食料品店のホームページで使ってみましょう!日付をフォーマットして出しているコードを覚えていますか?

```
<p>
  Today is:
  <span th:text="${#calendars.format(today,'dd MMMM yyyy')}">13 february 2011</span>
</p>
```

では、ここの `"dd MMMM yyyy"` をロケールに合わせたい場合はどうしましょうか?例えば `home_en.properties` に次のようなメッセージを追加したいかもしれません:

```
date.format=MMMM dd'','' yyyy
```

そして、同様に `home_es.properties` には次のように

```
date.format=dd ''de'' MMMM'','' yyyy
```

さて、th:with を使って、ローカライズされた日 付フォーマットを□数に入れて、それを th:text 式で使ってみましょう:

```
<p th:with="df=#{date.format}">
  Today is: <span th:text="${#calendars.format(today,df)}">13 February 2011</span>
</p>
```

□□で□□ですね。□は、th:with は th:text よりも高い□先□位を持っていますので全部を span タグに□くこともできます:

```
<p>
  Today is:
  <span th:with="df=#{date.format}"
        th:text="${#calendars.format(today,df)}">13 February 2011</span>
</p>
```

□先□位?それまだ知らない!と思ったかもしれませんね。心配しないでください、次の章は□先度についてです。

10 属性の□先□位

同じタグの中に□数の `th:*` 属性を□いた□合には何が起こるのでしょうか?例えば:

```
<ul>
  <li th:each="item : ${items}" th:text="${item.description}">Item description here...</li>
</ul>
```

もちろん、期待した□果を得るためには `th:each` 属性が `th:text` より先に□行されて欲しいですよね。ですが、DOM(Document Object Model)□□ではタグの中の属性が□かれている□番には特に意味を持たせていないので、私たちの想定通り□□作することを保□するためには、属性自 身に「□先□位」というルールを持たせなければなりません。

ですので、Thymeleafの全ての属性は数□の□先□位を定□しています。その□によってタグの中で□行される□番が□まります。この□番は次の通りです:

□番	□能	属性
1	フラグメントのインクルード	`th:include` `th:replace`
2	フラグメントの□り返し	`th:each`

3	条件の□□	`th:if`
		`th:unless`
		`th:switch`
		`th:case`

4	ローカル□数の定□	`th:object` `th:with`

5	一般的な属性の□更	`th:attr`
		`th:attrprepend`
		`th:attrappend`

6	特定の属性の□更	`th:value`
		`th:href`
		`th:src`
		`...`

7	テキスト (タグ特有の□更)	`th:text` `th:utext`
8	フラグメントの定□	`th:fragment`
9	フラグメントの削除	`th:remove`

この□先□位のルールがあるので、上□の□り返しのフラグメントで属性の位置を入れ替えても全く同じ□果を得ることができます(少し□みにくくなりますけどね)。

```
<ul>
  <li th:text="${item.description}" th:each="item : ${items}">Item description here...</li>
</ul>
```

11. コメントとブロック

11.1. □□的なHTML/XMLコメント

□□的なHTML/XMLコメント `<!-- ... -->` はThymeleafテンプレート内のどこでも使用することができます。このコメントの中にあるものは全てThymeleafにもブラウザにも□理されずに、一字一句そのまま□□に□果にコピーされます:

```
<!-- User info follows -->
<div th:text="${...}">
  ...
</div>
```

11.2. Thymeleafパ□サ□レベルのコメント ブロック

パーサーレベルのコメントブロックはThymeleafがそれをパースする□にテンプレートから削除されます。こんな感じです:

```
<!--/* This code will be removed at thymeleaf parsing time! */-->
```

Thymeleafは `<!--/*` と `*/-->` の□にあるもの全てを完全に削除するので、このコメントブロックはテンプレートが静的に□かれた□合には内容を表示するという用途のために使用することもできます。Thymeleafで□理すると削除されます:

```
<!--/*-->
  <div>
    you can see me only before thymeleaf processes me!
  </div>
<!--*/-->
```

これは、例えばたくさんの `<tr>` を持ったテーブルのプロトタイプを作成する□にとても便利かもしれません:

```
<table>
  <tr th:each="x : ${xs}">
    ...
  </tr>
  <!--/*-->
  <tr>
    ...
  </tr>
  <tr>
    ...
  </tr>
  <!--*/-->
</table>
```

11.3. Thymeleafプロトタイプのみのコメント ブロック

Thymeleafにはテンプレートが静的に(例えばプロトタイプとして)□かれた□合にはコメントになり、テンプレートとして□行された□合には通常のマークアップとして□われる特□なコメントブロックがあります。

```
<span>hello!</span>
<!--/*/
  <div th:text="${...}">
    ...
  </div>
/*/-->
<span>goodbye!</span>
```

Thymeleafのパーズ方式は単に `<!--/*/` と `/*/-->` のマーカーを削除しますが、コンテンツは削除しないので、そのコンテンツがアンコメントされて残ります。ですので、テンプレートを静的に実行するときにはThymeleafからは単にこのようにも見えます:

```
<span>hello!</span>

  <div th:text="${...}">
    ...
  </div>

<span>goodbye!</span>
```

パーサーレベルコメント機能と同じ。この機能はダイアレクトからは独立した機能です。

11.4. 代替的な `th:block` タグ

`th:block` は、Thymeleafのスタンダードダイアレクトに唯一含まれている要素処理子(属性処理子ではなく)です。

`th:block` は、テンプレート作者が好きな属性を指定することができるという、ただの属性コンテナにすぎません。Thymeleafは属性を評価して、次に単にその処理子を形もなく消してしまいます。

ですので例えば、繰り返しを使用したテーブルで各要素に対して1つ以上の `<tr>` が必要な場合に有用でしょう:

```
<table>
  <th:block th:each="user : ${users}">
    <tr>
        <td th:text="${user.login}">...</td>
        <td th:text="${user.name}">...</td>
    </tr>
    <tr>
        <td colspan="2" th:text="${user.address}">...</td>
    </tr>
  </th:block>
</table>
```

そしてプロトタイプのみのコメントと組み合わせると特に有用です:

```
<table>
    <!--/*/ <th:block th:each="user : ${users}"> /*/-->
    <tr>
        <td th:text="${user.login}">...</td>
        <td th:text="${user.name}">...</td>
    </tr>
    <tr>
        <td colspan="2" th:text="${user.address}">...</td>
    </tr>
    <!--/*/ </th:block> /*/-->
</table>
```

この解決策によって、テンプレートが(`<table>` 内で禁止されている `<div>` 処理子を置く必要がなく)妥当なHTMLになっていることに注意してください。また、プロトタイプとしてブラウザで静的に開かれても問題ありません!

12 インライン処理

12.1 テキスト のインライン処理

必要なものはほぼ全てスタンダードダイアレクトのタグ属性で処理できますが、HTMLテキストの中に直接式を書きたいというシチュエーションもあります。例えば、こう書くよりは:

```
<p>Hello, <span th:text="${session.user.name}">Sebastian</span>!</p>
```

このように書きたいかも知れません:

```
<p>Hello, [[${session.user.name}]]!</p>
```

`[[...]]` の中の式はThymeleafでインライン処理される式と見なされ、`th:text` 属性で使用することができる式ならどんな種類のものでも使用できます。

インライン処理を動作させるためには `th:inline` 属性を使用してアクティブにしなければなりません。`th:inline` 属性には3つの値またはモードを指定することができます (`text` と `javascript` と `none`)。

```
<p th:inline="text">Hello, [[${session.user.name}]]!</p>
```

`th:inline` を持つタグはインライン式を含んでいるタグ自 体である必要はなく、任意のタグであっても良いません:

```
<body th:inline="text">

   ...

   <p>Hello, [[${session.user.name}]]!</p>

   ...

</body>
```

そして今こう思っているかもしれません:「どうして最初からこれをしなかったの?この方が `th:text` 属性よりコードが少なくてすむじゃない!」ああ、落ち着けてください。インライン処理はとても面白いと思ったかもしれませんが、インライン用に書かれた式は静的に書いた場合にはそのままHTMLの中に表示される、ということを覚えておいてください。つまりその場合、プロトタイプとしてはたぶんもう使えないのです!

インライン処理を使用していないテンプレートを静的にブラウザで表示した場合:

```
Hello, Sebastian!
```

そして、インライン処理を使用した場合:

```
Hello, [[${session.user.name}]]!
```

ということです。

12.2 スクリプト のインライン処理 (JavaScript と Dart)

Thymeleafのインライン処理機能にはスクリプトモードがあります。いくつかのスクリプト言語で書かれたスクリプト内にデータを組み込むことができます。

現在のスクリプトモードは `javascript` (`th:inline="javascript"`) と `dart` (`th:inline="dart"`) です。

スクリプトのインライン処理でできることの1つ目 は、スクリプト内に式の結果を書くことです:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = /*[[${session.user.name}]]*/ 'Sebastian';

    ...
/*]]>*/
</script>
```

`/*[[...]]*/` 構文は、中の式を処理するようにThymeleafに伝えます。しかし、いくつかポイントがあります:

- Javascriptコメント(`/*...*/`)になっているので、ブラウザで静的にページを開いた場合にはこの式は処理されます。
- インライン式の後ろのコード(`'Sebastian'`)は、静的にページを開いた場合には表示されます。
- Thymeleafは式を実行して結果を挿入しますが、同時にこの行のインライン式の後ろにある全てのコードを削除します(静的に開いた場合には表示される部分です)。

ですので、実行結果はこのようになります:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = 'John Apricot';

    ...
/*]]>*/
</script>
```

コメント化しなくても大丈夫なのですが、それだと静的に読み込んだ場合にスクリプトのエラーになるでしょう:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = [[${session.user.name}]];

    ...
/*]]>*/
</script>
```

この処理はいいので、文字列以外も使用できることに注意してください。Thymeleafは次の種類のオブジェクトをJavascript/Dart構文に正しく変換することができます:

- Strings
- Numbers
- Booleans
- Arrays
- Collections
- Maps
- Beans (objects with *getter* and *setter* methods)

例えば、このようなコードがあったとしましょう:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var user = /*[[${session.user}]]*/ null;

    ...
/*]]>*/
</script>
```

この `${session.user}` 式によって `User` オブジェクトが□□され、Thymeleafによって正しくJavascript□文に□□されます:

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var user = {'age':null,'firstName':'John','lastName':'Apricot',
                'name':'John Apricot','nationality':'Antarctica'};

    ...
/*]]>*/
</script>
```

コードを追加する

Javascriptインライン□理で使用できるもう1つの□能は、 `/*[+...+]*/` という特□なコメント□文で挟まれたコードをインクルードするという□能です。これを使用すると、Thymeleafはテンプレート□理□に自 □的にそのコードをアンコメントします:

```
var x = 23;

/*[+

var msg  = 'This is a working application';

+]*/

var f = function() {
    ...
```

は、このように□理されます:

```
var x = 23;

var msg  = 'This is a working application';

var f = function() {
...
```

このコメントの中には式を含めることができ、Thymeleafで□□されます:

```
var x = 23;

/*[+

var msg  = 'Hello, ' + [[${session.user.name}]];

+]*/

var f = function() {
...
```

コードを削除する

Thymeleafでは `/*[- */` と `/* -]*/` という特□なコメントの□に挟むことでコードを削除することもできます:

```
var x = 23;

/*[- */

var msg  = 'This is a non-working template';

/* -]*/

var f = function() {
...
```

13 バリデーションと Doctype

13.1 テンプレートをバリデートする

前述の通り、Thymeleafには事前にテンプレートをバリデートする2つのテンプレートモードがあります: `VALIDXML` と `VALIDXHTML` です。これらのモードの場合はテンプレートは整形式のXMLである(常にそうあるべきですが)というだけでなく、更に指定された `DTD` にも従って妥当である必要があります。

これは、以下のように `DOCTYPE` を含んでいるテンプレートに対して `VALIDXHTML` モードを使用する場合です:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

これは、バリデーションエラーになるでしょう。`th:*` タグが `DTD` に存在しないからです。当然ですよね。W3CがThymeleafの機能を勝手に入れるわけがないですよね。でも、じゃあどうしましょうか? `DTD` を更新することによって解決します。

Thymeleafにはは XHTML勧告のオリジナルをコピーした `DTD` が含まれていて、それらの `DTD` では、スタンダードダイアレクトの全ての `th:*` 属性が使用できるようになっています。そういった理由で、これまでテンプレート内で次のようにしていたのです:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
```

`SYSTEM` 句の子はThymeleafパーサーに対して、Thymeleafが用意した特別な `XHTML 1.0 Strict DTD` ファイルを解決して、テンプレートをバリデートするときにそれを使用するように指示します。`http` の部分に対しては心配しないでください、これはただの句の子であって `DTD` ファイルはThymeleafのjarファイルからローカルで読み込まれます。

> このDOCTYPE宣言は完全に妥当なので、ブラウザで静的にこのテンプレートをプロトタイプとして開いた場合は 互換モードでレンダリングされることに注意してください。

対応しているXHTML全てに対してThymeleafが用意した `DTD` 定義の一式を提供します:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-transitional-thymeleaf-4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-frameset-thymeleaf-4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml11-thymeleaf-4.dtd">
```

また、バリデートモードを使用していない場合でも、IDEが幸せになるように `th` ネームスペースを `html` タグに定義しておいておくと良いです。

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
```

13.2 Doctypeの省略

テンプレートに次のように `DOCTYPE` を持つことは良いのですが:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
```

この `DOCTYPE` を持ったXHTMLドキュメントをウェブアプリケーションからクライアントブラウザに送るのは次のような理由から良くありません:

- `PUBLIC` ではないので(`SYSTEM DOCTYPE` なので)、W3Cのバリデーターでバリデートすることができない。
- 処理後には全ての `th:*` タグはなくなるので、この宣言は不要。

そのため、ThymeleafにはDOCTYPE 省略のメカニズムがあり、自動的にThymeleaf用のXHTML `DOCTYPE` を適切な `DOCTYPE` に置換します。

例えば、テンプレートが *XHTML 1.0 Strict* で次のような場合:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    ...
</html>
```

Thymeleafでテンプレートを処理すると、結果のXHTMLは次のようになります:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
    ...
</html>
```

この2つの行については何もする必要はありません: Thymeleafが自動的に面倒をみてくれます。

14 食料品店用のページをいくつか追加

Thymeleafの使い方について、もうたくさん知っているので、注文管理のための新ページをいくつか追加することができます。

XHTMLコードにフォーカスしますが、□□するエントローラーを□てみたい□合はパドルされたソースコードをチェックしてください。

14.1 注文リスト

注文リストページを作成しましょう `/WEB-INF/templates/order/list.html`:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <head>

    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <h1>Order list</h1>

    <table>
      <tr>
        <th>DATE</th>
        <th>CUSTOMER</th>
        <th>TOTAL</th>
        <th></th>
      </tr>
      <tr th:each="o : ${orders}" th:class="${oStat.odd}? 'odd'">
        <td th:text="${#calendars.format(o.date,'dd/MMM/yyyy')}">13 jan 2011</td>
        <td th:text="${o.customer.name}">Frederic Tomato</td>
        <td th:text="${#aggregates.sum(o.orderLines.{purchasePrice * amount})}">23.32</td>
        <td>
          <a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>
        </td>
      </tr>
    </table>

    <p>
      <a href="../home.html" th:href="@{/}">Return to home</a>
    </p>

  </body>

</html>
```

□くようなことは何もありません。ちょっとしたOGNLでシンプルですね:

```
<td th:text="${#aggregates.sum(o.orderLines.{purchasePrice * amount})}">23.32</td>
```

ここでやっているのは注文の中の各注文行(`OrderLine` オブジェクト)で `purchasePrice` と `amount` 力値を□□する `getPurchasePrice()` と `getAmount()` メソッドを呼び出して□いあわせて□果を数□のリストに返し、`#aggregates.sum(...)` □数で集□して注文の合□金□を取得するという□理です。

きっとOGNLのパワーが好きになったでしょう。

14.2 注文□□

次は、注文□□ページです。アスタリスク□文を多用します:

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body th:object="${order}">

    <h1>Order details</h1>

    <div>
      <p><b>Code:</b> <span th:text="*{id}">99</span></p>
      <p>
        <b>Date:</b>
        <span th:text="*{#calendars.format(date,'dd MMM yyyy')}">13 jan 2011</span>
      </p>
    </div>

    <h2>Customer</h2>

    <div th:object="*{customer}">
      <p><b>Name:</b> <span th:text="*{name}">Frederic Tomato</span></p>
      <p>
        <b>Since:</b>
        <span th:text="*{#calendars.format(customerSince,'dd MMM yyyy')}">1 jan 2011</span>
      </p>
    </div>

    <h2>Products</h2>

    <table>
      <tr>
        <th>PRODUCT</th>
        <th>AMOUNT</th>
        <th>PURCHASE PRICE</th>
      </tr>
      <tr th:each="ol,row : *{orderLines}" th:class="${row.odd}? 'odd'">
        <td th:text="${ol.product.name}">Strawberries</td>
        <td th:text="${ol.amount}" class="number">3</td>
        <td th:text="${ol.purchasePrice}" class="number">23.32</td>
      </tr>
    </table>

    <div>
      <b>TOTAL:</b>
      <span th:text="*{#aggregates.sum(orderLines.{purchasePrice * amount})}">35.23</span>
    </div>

    <p>
      <a href="list.html" th:href="@{/order/list}">Return to order list</a>
    </p>

  </body>

</html>
```

ここでも本当に新しいことはありません。このオブジェクト□□くらいですね:

```
<body th:object="${order}">

  ...

  <div th:object="*{customer}">
```

```
    <p><b>Name:</b> <span th:text="*{name}">Frederic Tomato</span></p>
    ...
  </div>

  ...
</body>
```

この *{name} は□□は次と同等です:

```
<p><b>Name:</b> <span th:text="${order.customer.name}">Frederic Tomato</span></p>
```

15 □定についてもう少し

15.1 テンプレ□トリゾルバ□

ウェブサイト□想食料品店では `ITemplateResolver` □装の `ServletContextTemplateResolver` を□び、テンプレートをサーブレットコンテキストからリソースとして取得しました。

`ITemplateResolver` を□装して独自 のテンプレートリゾルバーを作成する以外にもThymeleafにはそのまま使用可能な□装が3つあります:

- `org.thymeleaf.templateresolver.ClassLoaderTemplateResolver` テンプレートをクラスローダーリソースとして解□します:

```
return Thread.currentThread().getContextClassLoader().getResourceAsStream(templateName);
```

- `org.thymeleaf.templateresolver.FileTemplateResolver` テンプレートをファイルシステムのファイルとして解□します:

```
return new FileInputStream(new File(templateName));
```

- `org.thymeleaf.templateresolver.UrlTemplateResolver` テンプレートをURL(ローカル以外でも大丈夫)として解□します:

```
return (new URL(templateName)).openStream();
```

最初から□ビルドされている `ITemplateResolver` □装には、全て同じ□定パラメータを指定することができます。そのパラメータには次のようなものがあります:

- Prefix と suffix の□定(もう□たことありますね):

```
templateResolver.setPrefix("/WEB-INF/templates/");
templateResolver.setSuffix(".html");
```

- テンプレートエイリアス□定。これを使用するとファイル名と一致しないテンプレート名を使用することができます。suffix/prefixとエイリアスが□方指定されている□合はエイリアスがprefix/suffixより前に□用されます:

```
templateResolver.addTemplateAlias("adminHome","profiles/admin/home");
templateResolver.setTemplateAliases(aliasesMap);
```

- テンプレートを□み□むときのエンコーディング□定:

```
templateResolver.setEncoding("UTF-8");
```

- デフォルトテンプレートモード□定と、特定のテンプレートに他のモードを指定するためのパターン□定:

```
// デフォルト は TemplateMode.XHTML
templateResolver.setTemplateMode("HTML5");
templateResolver.getXhtmlTemplateModePatternSpec().addPattern("*.xhtml");
```

- テンプレートキャッシュのデフォルトモード□定と、特定のテンプレートにキャッシュするかしないかを指定するためのパターン□定:

```
// デフォルト は true
templateResolver.setCacheable(false);
templateResolver.getCacheablePatternSpec().addPattern("/users/*");
```

- このテンプレートリゾルバーでパースされたテンプレートキャッシュエントリーのTTLをミリセカンドで□定。□定されていない□合は、キャッシュからエントリーが削除されるのはLRU(最大キャッシュサイズを超えた□に一番古いキャッシュエントリーが削除される)のみになります。

```
// デフォルト はTTL指定なし （LRUのみがエント リ ーを削除）
templateResolver.setCacheTTLMs(60000L);
```

また、テンプレートエンジンには□数のテンプレートリゾルバーを指定することもできます。その□合、テンプレート解□のために□テンプレートリゾルバーは□番付けされ、最初のリゾルバーがテンプレートを解□できない□合には、次のリゾルバーに□い合わせる、といった流れで□理を行います:

```
ClassLoaderTemplateResolver classLoaderTemplateResolver = new ClassLoaderTemplateResolver();
classLoaderTemplateResolver.setOrder(Integer.valueOf(1));

ServletContextTemplateResolver servletContextTemplateResolver = new
ServletContextTemplateResolver();
servletContextTemplateResolver.setOrder(Integer.valueOf(2));

templateEngine.addTemplateResolver(classLoaderTemplateResolver);
templateEngine.addTemplateResolver(servletContextTemplateResolver);
```

□数のテンプレートリゾルバーが□用されている□合には、それぞれのテンプレートリゾルバーにパターンを指定することをお□めします。そうすることでThymeleafはテンプレートに□して、□象外のリゾルバーを素早く□□することができるので、パフォーマンスが良くなります。必□ということではなく、最□化のための□めです:

```
ClassLoaderTemplateResolver classLoaderTemplateResolver = new ClassLoaderTemplateResolver();
classLoaderTemplateResolver.setOrder(Integer.valueOf(1));
// This classloader will not be even asked for any templates not matching these patterns
classLoaderTemplateResolver.getResolvablePatternSpec().addPattern("/layout/*.html");
classLoaderTemplateResolver.getResolvablePatternSpec().addPattern("/menu/*.html");

ServletContextTemplateResolver servletContextTemplateResolver = new
ServletContextTemplateResolver();
servletContextTemplateResolver.setOrder(Integer.valueOf(2));
```

15.2 メッセ□ジリ ゾルバ□

私たちの食料品店アプリケーションでは、明示的にメッセージリゾルバーの□装を指定していません。前述の通り、この□合は `org.thymeleaf.messageresolver.StandardMessageResolver` オブジェクトが使用されています。

この `StandardMessageResolver` は、既に□明した通りテンプレートと同じ名前のメッセージファイルを探しますが、□□のところThymeleafのコアがそのまま使えるように用意している唯一のメッセージリゾルバーです。ですがもちろん、`org.thymeleaf.messageresolver.IMessageResolver` を□装すればあなた独自 のメッセージリゾルバーを作成することができます。

> Thymeleaf + Spring □携パッケージでは `IMessageResolver` の□装が提供されていて、そのリゾルバーは□□的なSpringの方法で、`MessageSource` オブジェクトを使用して外部化されたメッセージを取得します。

テンプレートエンジンにメッセージリゾルバーを1つ(または□数)指定したい□合はどうすれば良いでしょうか?□□です:

```
// For setting only one
templateEngine.setMessageResolver(messageResolver);

// For setting more than one
templateEngine.addMessageResolver(messageResolver);
```

でも、どうして□数のメッセージリゾルバーを指定したいのでしょうか?テンプレートリゾルバーと同じ理由ですね: メッセージリゾルバーは□番付けされて、最初のリゾルバーがあるメッセージを解□できなければ、次のリゾルバーに□い合わせて、その次は3番目 に・・となります。

15.3 ロギング

Thymeleafはロギングにもかなり□を使っていて、いつでもロギングインターフェイスを通して最大限の有用な情□を提供しようとしています。

ロギングライブラリは `slf4j` を使用しています。`slf4j` は□□にアプリケーションで使用しているどんなロギング□装(例えば `log4j`)に□してもブリッジとして振る舞います。

Thymeleafのクラスは、□□レベルに合わせて TRACE と DEBUG と INFO レベルのログを出力します。そして一般的なロギングとは□に、TemplateEngine クラスに□□付けられた3つの特□なロガーがあり目 的に合わせて□□に□定をすることができます。

- org.thymeleaf.TemplateEngine.CONFIG は、ライブラリの初期化□に□定の□□を出力します。
- org.thymeleaf.TemplateEngine.TIMER は、それぞれのテンプレートを□理する□につかかった□□を出力します(ベンチマークに便利です!)。
- org.thymeleaf.TemplateEngine.cache は、キャッシュに□する特定の情□を出力するロガーのプレフィックスになっています。ユーザーがキャッシュロガーの名前を□定することができるので、名前は□わりうるのですが、デフォルトでは:
    - org.thymeleaf.TemplateEngine.cache.TEMPLATE_CACHE
    - org.thymeleaf.TemplateEngine.cache.FRAGMENT_CACHE
    - org.thymeleaf.TemplateEngine.cache.MESSAGE_CACHE
    - org.thymeleaf.TemplateEngine.cache.EXPRESSION_CACHE

Thymeleafのロギングインフラのための□定例は log4j を使用する□合、次のようになります:

```
log4j.logger.org.thymeleaf=DEBUG
log4j.logger.org.thymeleaf.TemplateEngine.CONFIG=TRACE
log4j.logger.org.thymeleaf.TemplateEngine.TIMER=TRACE
log4j.logger.org.thymeleaf.TemplateEngine.cache.TEMPLATE_CACHE=TRACE
```

16 テンプレートキャッシュ

ThymeleafはDOM処理エンジンと一連のプロセッサーロジックを使用する必要があるノードのタイプごとに1つ — のおかげで高いパフォーマンスを発揮しています。プロセッサは、ドキュメントのDOMツリーとデータをバインドすることによって期待する結果を作成するために、DOMツリーに変更を加えます。

また、 — デフォルトで — パースしたテンプレートをキャッシュする機能があります。テンプレートファイルを読み込んでパースした結果の、処理前のDOMツリーをキャッシュします。これは特に以下のようなコンセプトに基づいて作成されているウェブアプリケーションに役立ちます:

- Input/Outputが、いつでもどんなアプリケーションにとっても最も高い部分である。インメモリ処理の方が全然速い。
- インメモリのDOMツリーをクローンする方が、テンプレートファイルを読み込んでパースして新しいDOMツリーを生成するよりも断然速い。
- ウェブアプリケーションは通常数十個のテンプレートしか使わない。
- テンプレートファイルは小・中程度のサイズであって、アプリケーションの実行中には変更されない。

このことから、ウェブアプリケーションで最も使用されているテンプレートをキャッシュすることで、大量のメモリを効率に使うこともなくうまくいきそうですし、原則として変更されない少ないファイルのIO処理に費やされるたくさんの時間を節約することができると考えられます。

このキャッシュをどのようにコントロールすることができるのでしょうか？まず、テンプレートリゾルバーで有効/無効の切り替えをすることができ、特定のテンプレートにだけ適用することもできるということは学びましたね。

```
// Default is true
templateResolver.setCacheable(false);
templateResolver.getCacheablePatternSpec().addPattern("/users/*");
```

また、独自の「キャッシュマネージャー」を作成することで設定を変更することもできます。デフォルトの `StandardCacheManager` 実装のインスタンスを使用することも可能です。

```
// Default is 50
StandardCacheManager cacheManager = new StandardCacheManager();
cacheManager.setTemplateCacheMaxSize(100);
...
templateEngine.setCacheManager(cacheManager);
```

キャッシュの設定についてのより詳しい情報は `org.thymeleaf.cache.StandardCacheManager` のJavadoc APIを参照してください。

テンプレートキャッシュから手動でエントリーを削除することもできます:

```
// Clear the cache completely
templateEngine.clearTemplateCache();

// Clear a specific template from the cache
templateEngine.clearTemplateCacheFor("/users/userList");
```

17 Appendix A: Expression Basic Objects

(OGNLやSpring EL)によって行される)数式の中で、常に使用可能なオブジェクトや数マップがあります。それをてみましょう:

基本オブジェクト

- **#ctx** : コンテキストオブジェクト。境(スタンドアローンかウェブか)によって `org.thymeleaf.context.IContext` や `org.thymeleaf.context.IWebContext` の装になります。*Spring*携モジュールを使用している合は `org.thymeleaf.spring[3|4].context.SpringWebContext` のインスタンスになります。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.IContext
 * ======================================================================
 */

${#ctx.locale}
${#ctx.variables}

/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.IWebContext
 * ======================================================================
 */

${#ctx.applicationAttributes}
${#ctx.httpServletRequest}
${#ctx.httpServletResponse}
${#ctx.httpSession}
${#ctx.requestAttributes}
${#ctx.requestParameters}
${#ctx.servletContext}
${#ctx.sessionAttributes}
```

- **#locale** : 在のリクエストに付けられている `java.util.Locale` への直接アクセス。

```
${#locale}
```

- **#vars** : コンテキスト内の全ての数を持った `org.thymeleaf.context.VariablesMap` のインスタンス(通常は `#ctx.variables` に含まれている数にローカル数を加えたものです)。

  限定子がついていない式はこのオブジェクトにしてされます。のところ `${something}` は `${#vars.something}` と完全に同等です(がより美しいです)。

  `#root` はこのオブジェクトの同意です。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.VariablesMap
 * ======================================================================
 */

${#vars.get('foo')}
${#vars.containsKey('foo')}
${#vars.size()}
...
```

request/session 属性 ウェブコンテキスト ネムスペス

ウェブ境でThymeleafを使っている合、リクエストパラメータ、セッション属性、アプリケーション属性にアクセスするのにショートカットを使用することができます。

これらは「コンテキストオブジェクト」ではなく、コンテキストに□して□数として追加されたマップです。ですので # を使いません。そのため、ある意味で「名前空□」のように振る舞います。

- **param**：リクエストパラメータを取得するために使用します。`${param.foo}` は `foo` リクエストパラメータの□を持つ `String[]` です。ですので、最初の□を取得するために普通は `${param.foo[0]}` を使用します。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebRequestParamsVariablesMap
 * ======================================================================
 */

${param.foo}              // Retrieves a String[] with the values of request parameter 'foo'
${param.size()}
${param.isEmpty()}
${param.containsKey('foo')}
...
```

- **session**：セッション属性を取得するために使用します。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebSessionVariablesMap
 * ======================================================================
 */

${session.foo}                // Retrieves the session atttribute 'foo'
${session.size()}
${session.isEmpty()}
${session.containsKey('foo')}
...
```

- **application**：アプリケーション/サーブレットコンテキストを取得するために使用します。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebServletContextVariablesMap
 * ======================================================================
 */

${application.foo}             // Retrieves the ServletContext atttribute 'foo'
${application.size()}
${application.isEmpty()}
${application.containsKey('foo')}
...
```

リクエスト属性にアクセスするときは、リクエストパラメータとは対照的に、名前空□を指定する必要がないことに注意してください。なぜなら、全てのリクエスト属性は自□的にコンテキストルートの□数としてコンテキストに追加されるからです：

```
${myRequestAttribute}
```

ウェブコンテキスト オブジェクト

ウェブ□境の□合は、次のようなオブジェクトにも直接アクセスすることができます（これらはオブジェクトであって、マップや名前空□ではないことに注意して下さい）：

- **#httpServletRequest**：□在のリクエストに□□付けられた `javax.servlet.http.HttpServletRequest` オブジェクトへの直接アクセス

```
${#httpServletRequest.getAttribute('foo')}
${#httpServletRequest.getParameter('foo')}
${#httpServletRequest.getContextPath()}
${#httpServletRequest.getRequestName()}
```

```
...
```

- **#httpSession** : □在のリクエストに□□付けられた `javax.servlet.http.HttpSession` オブジェクトへの直接アクセス。

```
${#httpSession.getAttribute('foo')}
${#httpSession.id}
${#httpSession.lastAccessedTime}
...
```

Springコンテキスト オブジェクト

SpringからThymeleafを使用している□合は、これらのオブジェクトにもアクセスできます:

- **#themes** : Springの `spring:theme` JSPタグと同じ□能を提供します。

```
${#themes.code('foo')}
```

Springビ□ン

Thymeleafでは、SpringELによってSpringアプリケーションコンテキストに通常の方法で定□されて登□されたビーンに `@beanName` シンタックスを使用してアクセスすることができます。例:

```
<div th:text="${@authService.getUserName()}">...</div>
```

18 Appendix B: Expression Utility Objects

日付

- **#dates** : `java.util.Date` オブジェクトに□するユーティリティメソッド群:

```
/*
 * =====================================================================
 * See javadoc API for class org.thymeleaf.expression.Dates
 * =====================================================================
 */

/*
 * □□ロケールフォーマット で日付をフォーマット します
 * 配列、リスト 、セット にも□□し ています
 */
${#dates.format(date)}
${#dates.arrayFormat(datesArray)}
${#dates.listFormat(datesList)}
${#dates.setFormat(datesSet)}

/*
 * 指定されたパターンで日付をフォーマット します
 * 配列、リスト 、セット にも□□し ています
 */
${#dates.format(date, 'dd/MMM/yyyy HH:mm')}
${#dates.arrayFormat(datesArray, 'dd/MMM/yyyy HH:mm')}
${#dates.listFormat(datesList, 'dd/MMM/yyyy HH:mm')}
${#dates.setFormat(datesSet, 'dd/MMM/yyyy HH:mm')}

/*
 * 日付のプロパティを取得します
 * 配列、リスト 、セット にも□□し ています
 */
${#dates.day(date)}                     // also arrayDay(...), listDay(...), etc.
${#dates.month(date)}                   // also arrayMonth(...), listMonth(...), etc.
${#dates.monthName(date)}               // also arrayMonthName(...), listMonthName(...), etc.
${#dates.monthNameShort(date)}          // also arrayMonthNameShort(...),
listMonthNameShort(...), etc.
${#dates.year(date)}                    // also arrayYear(...), listYear(...), etc.
${#dates.dayOfWeek(date)}               // also arrayDayOfWeek(...), listDayOfWeek(...), etc.
${#dates.dayOfWeekName(date)}           // also arrayDayOfWeekName(...),
listDayOfWeekName(...), etc.
${#dates.dayOfWeekNameShort(date)}      // also arrayDayOfWeekNameShort(...),
listDayOfWeekNameShort(...), etc.
${#dates.hour(date)}                    // also arrayHour(...), listHour(...), etc.
${#dates.minute(date)}                  // also arrayMinute(...), listMinute(...), etc.
${#dates.second(date)}                  // also arraySecond(...), listSecond(...), etc.
${#dates.millisecond(date)}             // also arrayMillisecond(...), listMillisecond(...),
etc.

/*
 * コンポーネント を指定し て日付オブジェクト (java.util.Date)を作成し ます
 */
${#dates.create(year,month,day)}
${#dates.create(year,month,day,hour,minute)}
${#dates.create(year,month,day,hour,minute,second)}
${#dates.create(year,month,day,hour,minute,second,millisecond)}

/*
 * □在日 □の日付オブジェクト (java.util.Date)を作成し ます
 */
${#dates.createNow()}

/*
 * □在日 の日付のオブジェクト (java.util.Date)を作成し ます(□□は 00:00に□定され ます)
 */
${#dates.createToday()}
```

カレンダー

- **#calendars** : `#dates` に似ていますが、 `java.util.Calendar` オブジェクト用です:

```
/*
 * =========================================================================
 * See javadoc API for class org.thymeleaf.expression.Calendars
 * =========================================================================
 */

/*
 * ロケールフォーマット でカレンダーをフォーマット します
 * 配列、リスト 、セット にも対応 しています
 */
${#calendars.format(cal)}
${#calendars.arrayFormat(calArray)}
${#calendars.listFormat(calList)}
${#calendars.setFormat(calSet)}

/*
 * 指定されたパターンでカレンダーをフォーマット します
 * 配列、リスト 、セット にも対応 しています
 */
${#calendars.format(cal, 'dd/MMM/yyyy HH:mm')}
${#calendars.arrayFormat(calArray, 'dd/MMM/yyyy HH:mm')}
${#calendars.listFormat(calList, 'dd/MMM/yyyy HH:mm')}
${#calendars.setFormat(calSet, 'dd/MMM/yyyy HH:mm')}

/*
 * カレンダーのプロパティを取得します
 * 配列、リスト 、セット にも対応 しています
 */
${#calendars.day(date)}                // also arrayDay(...), listDay(...), etc.
${#calendars.month(date)}              // also arrayMonth(...), listMonth(...), etc.
${#calendars.monthName(date)}          // also arrayMonthName(...), listMonthName(...), etc.
${#calendars.monthNameShort(date)}     // also arrayMonthNameShort(...),
listMonthNameShort(...), etc.
${#calendars.year(date)}               // also arrayYear(...), listYear(...), etc.
${#calendars.dayOfWeek(date)}          // also arrayDayOfWeek(...), listDayOfWeek(...), etc.
${#calendars.dayOfWeekName(date)}      // also arrayDayOfWeekName(...),
listDayOfWeekName(...), etc.
${#calendars.dayOfWeekNameShort(date)} // also arrayDayOfWeekNameShort(...),
listDayOfWeekNameShort(...), etc.
${#calendars.hour(date)}               // also arrayHour(...), listHour(...), etc.
${#calendars.minute(date)}             // also arrayMinute(...), listMinute(...), etc.
${#calendars.second(date)}             // also arraySecond(...), listSecond(...), etc.
${#calendars.millisecond(date)}        // also arrayMillisecond(...), listMillisecond(...),
etc.

/*
 * コンポーネント を指定し てカレンダーオブジェクト (java.util.Calendar)を 作成し ます
 */
${#calendars.create(year,month,day)}
${#calendars.create(year,month,day,hour,minute)}
${#calendars.create(year,month,day,hour,minute,second)}
${#calendars.create(year,month,day,hour,minute,second,millisecond)}

/*
 * 本日 のカレンダーオブジェクト (java.util.Calendar)を 作成し ます
 */
${#calendars.createNow()}

/*
 * 本日 のカレンダーのオブジェクト (java.util.Calendar)を 作成し ます (時刻は 00:00に 設定されま
す )
 */
${#calendars.createToday()}
```

数値

- **#numbers** : 数値オブジェクトに対するユーティリティメソッド群:

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.expression.Numbers
 * ======================================================================
 */

/*
 * ==========================
 * 整数のフォーマット
 * ==========================
 */

/*
 * 整数の最小桁数を指定します。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatInteger(num,3)}
${#numbers.arrayFormatInteger(numArray,3)}
${#numbers.listFormatInteger(numList,3)}
${#numbers.setFormatInteger(numSet,3)}


/*
 * 整数の最小桁数と千の位の区切り文字を指定します。
 * 'POINT', 'COMMA', 'NONE' または 'DEFAULT' (ロケールに依存)。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatInteger(num,3,'POINT')}
${#numbers.arrayFormatInteger(numArray,3,'POINT')}
${#numbers.listFormatInteger(numList,3,'POINT')}
${#numbers.setFormatInteger(numSet,3,'POINT')}


/*
 * ==========================
 * 小数のフォーマット
 * ==========================
 */

/*
 * 整数の最小桁数と小数桁数を指定します。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatDecimal(num,3,2)}
${#numbers.arrayFormatDecimal(numArray,3,2)}
${#numbers.listFormatDecimal(numList,3,2)}
${#numbers.setFormatDecimal(numSet,3,2)}

/*
 * 整数の最小桁数と小数桁数と小数点の文字を指定します。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatDecimal(num,3,2,'COMMA')}
${#numbers.arrayFormatDecimal(numArray,3,2,'COMMA')}
${#numbers.listFormatDecimal(numList,3,2,'COMMA')}
${#numbers.setFormatDecimal(numSet,3,2,'COMMA')}

/*
 * 整数の最小桁数と小数桁数と小数点の文字と千の位の区切り文字を指定します。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatDecimal(num,3,'POINT',2,'COMMA')}
${#numbers.arrayFormatDecimal(numArray,3,'POINT',2,'COMMA')}
${#numbers.listFormatDecimal(numList,3,'POINT',2,'COMMA')}
${#numbers.setFormatDecimal(numSet,3,'POINT',2,'COMMA')}



/*
 * ==========================
 * ユーティリティメソッド
 * ==========================
 */

/*
 * xからyまでの整数のシーケンス(配列)を作成します
```

```
 */
${#numbers.sequence(from,to)}
${#numbers.sequence(from,to,step)}
```

文字列

- **#strings** : String オブジェクトに□する□ーティリティメソッド群:

```
/*
 * ========================================================================
 * See javadoc API for class org.thymeleaf.expression.Strings
 * ========================================================================
 */

/*
 * Null安全な toString()
 */
${#strings.toString(obj)}                           // array*, list* and set* にも□□し ていま
す

/*
 * 文字列が空(またはnull)かどうかをチエックします。チエック前に trim() □理をします。
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.isEmpty(name)}
${#strings.arrayIsEmpty(nameArr)}
${#strings.listIsEmpty(nameList)}
${#strings.setIsEmpty(nameSet)}

/*
 * 'isEmpty()' を □行して false の□合はその文字列を返し 、 true の□合は指定されたデフォ
ルト 文字列を返します
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.defaultString(text,default)}
${#strings.arrayDefaultString(textArr,default)}
${#strings.listDefaultString(textList,default)}
${#strings.setDefaultString(textSet,default)}

/*
 * 文字列にフラグメント が含まれているかどうかをチエックします
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.contains(name,'ez')}                     // also array*, list* and set*
${#strings.containsIgnoreCase(name,'ez')}           // also array*, list* and set*

/*
 * 文字列が指定されたフラグメント で始まっているかどうかまたは□わっているかどうかを
チエックします
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.startsWith(name,'Don')}                  // also array*, list* and set*
${#strings.endsWith(name,endingFragment)}           // also array*, list* and set*

/*
 * 部分文字列□□
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.indexOf(name,frag)}                      // also array*, list* and set*
${#strings.substring(name,3,5)}                     // also array*, list* and set*
${#strings.substringAfter(name,prefix)}             // also array*, list* and set*
${#strings.substringBefore(name,suffix)}            // also array*, list* and set*
${#strings.replace(name,'las','ler')}               // also array*, list* and set*

/*
 * Append と prepend
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.prepend(str,prefix)}                     // also array*, list* and set*
${#strings.append(str,suffix)}                      // also array*, list* and set*

/*
 * 大文字小文字□□
 * 配列、リスト 、セット にも□□し ています
```

```
 */
${#strings.toUpperCase(name)}                       // also array*, list* and set*
${#strings.toLowerCase(name)}                       // also array*, list* and set*

/*
 * Split と  join
 */
${#strings.arrayJoin(namesArray,',')}
${#strings.listJoin(namesList,',')}
${#strings.setJoin(namesSet,',')}
${#strings.arraySplit(namesStr,',')}                // returns String[]
${#strings.listSplit(namesStr,',')}                 // returns List<String>
${#strings.setSplit(namesStr,',')}                  // returns Set<String>

/*
 * Trim
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.trim(str)}                               // also array*, list* and set*

/*
 * □さの□算
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.length(str)}                             // also array*, list* and set*

/*
 * 与えられたテキストが最大サイズ nになるよう 省略□理をします。
 * もしテキストがそれよりも大きい□合は、切り 取られて最後に "..." がつきます。
 * 配列、リスト 、セット にも□□し ています
 */
${#strings.abbreviate(str,10)}                      // also array*, list* and set*

/*
 * 最初の文字を 大文字に□□(と その逆)
 */
${#strings.capitalize(str)}                         // also array*, list* and set*
${#strings.unCapitalize(str)}                       // also array*, list* and set*

/*
 * □□の最初の文字を 大文字に□□
 */
${#strings.capitalizeWords(str)}                    // also array*, list* and set*
${#strings.capitalizeWords(str,delimiters)}         // also array*, list* and set*

/*
 * 文字列のエスケープ
 */
${#strings.escapeXml(str)}                          // also array*, list* and set*
${#strings.escapeJava(str)}                         // also array*, list* and set*
${#strings.escapeJavaScript(str)}                   // also array*, list* and set*
${#strings.unescapeJava(str)}                       // also array*, list* and set*
${#strings.unescapeJavaScript(str)}                 // also array*, list* and set*

/*
 * Null安全な比□と □□
 */
${#strings.equals(str)}
${#strings.equalsIgnoreCase(str)}
${#strings.concat(str)}
${#strings.concatReplaceNulls(str)}

/*
 * Random
 */
${#strings.randomAlphanumeric(count)}
```

オブジェクト

- **#objects** : 一般的なオブジェクトに□するユーティリティメソッド群

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Objects
```

```
 * =============================================================================
 */

/*
 * null でなければ obj を、null の場合は指定されたデフォルト を返します
 * 配列、リスト、セット にも対応しています
 */
${#objects.nullSafe(obj,default)}
${#objects.arrayNullSafe(objArray,default)}
${#objects.listNullSafe(objList,default)}
${#objects.setNullSafe(objSet,default)}
```

真偽

- **#bools** : 真偽判定に関するユーティリティメソッド群

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Bools
 * =============================================================================
 */

/*
 * th:if タグと同じように条件を評価します(条件の評価の章を後で参照してください)。
 * 配列、リスト、セット にも対応しています
 */
${#bools.isTrue(obj)}
${#bools.arrayIsTrue(objArray)}
${#bools.listIsTrue(objList)}
${#bools.setIsTrue(objSet)}

/*
 * 否定の評価
 * 配列、リスト、セット にも対応しています
 */
${#bools.isFalse(cond)}
${#bools.arrayIsFalse(condArray)}
${#bools.listIsFalse(condList)}
${#bools.setIsFalse(condSet)}

/*
 * 評価して AND演算子を適用
 * 配列、リスト、セット をパラメータとして受け取ります
 */
${#bools.arrayAnd(condArray)}
${#bools.listAnd(condList)}
${#bools.setAnd(condSet)}

/*
 * 評価して OR演算子を適用
 * 配列、リスト、セット をパラメータとして受け取ります
 */
${#bools.arrayOr(condArray)}
${#bools.listOr(condList)}
${#bools.setOr(condSet)}
```

配列

- **#arrays** : 配列に関するユーティリティメソッド群

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Arrays
 * =============================================================================
 */

/*
 * コンポーネントクラスを推論して配列に変換します
 * 結果の配列が空もしくは、対象オブジェクト に複数のクラスが含まれる場合 Object[] を返します。
```

```
 */
${#arrays.toArray(object)}

/*
 * コンポーネント クラスを指定して配列に□□
 */
${#arrays.toStringArray(object)}
${#arrays.toIntegerArray(object)}
${#arrays.toLongArray(object)}
${#arrays.toDoubleArray(object)}
${#arrays.toFloatArray(object)}
${#arrays.toBooleanArray(object)}

/*
 * □さを□算
 */
${#arrays.length(array)}

/*
 * 配列が空かどうかをチェック
 */
${#arrays.isEmpty(array)}

/*
 * 1つまたは□数の要素が配列に含まれているかどうかをチェック
 */
${#arrays.contains(array, element)}
${#arrays.containsAll(array, elements)}
```

リスト

- **#lists** : リストに□するユーティリティメソッド群

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Lists
 * ============================================================================
 */

/*
 * リスト に□□
 */
${#lists.toList(object)}

/*
 * サイズを□算
 */
${#lists.size(list)}

/*
 * リスト が空かどうかをチェック
 */
${#lists.isEmpty(list)}

/*
 * 1つまたは□数の要素がリスト に含まれているかどうかをチェック
 */
${#lists.contains(list, element)}
${#lists.containsAll(list, elements)}

/*
 * 与えられたリスト のコピーをソート。リスト のメンバーが comparable を□装して いるか
 * または comparator が指定されている必要があります。
 */
${#lists.sort(list)}
${#lists.sort(list, comparator)}
```

セット

- **#sets** : セットに□するユーティリティメソッド群

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Sets
 * =============================================================================
 */

/*
 * セット に□□
 */
${#sets.toSet(object)}

/*
 * サイズを□算
 */
${#sets.size(set)}

/*
 * セット が空かどうかをチエック
 */
${#sets.isEmpty(set)}

/*
 * 1つまたは□数の要素がセット に含まれているかどうかをチエック
 */
${#sets.contains(set, element)}
${#sets.containsAll(set, elements)}
```

マップ

- **#maps** : ﾏｯﾌﾟに□する□ﾃｨﾘﾃｨﾒｿｯﾄﾞ群

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Maps
 * =============================================================================
 */

/*
 * サイズを□算
 */
${#maps.size(map)}

/*
 * マップが空かどうかをチエック
 */
${#maps.isEmpty(map)}

/*
 * キーや□がマップに含まれているかどうかをチエック
 */
${#maps.containsKey(map, key)}
${#maps.containsAllKeys(map, keys)}
${#maps.containsValue(map, value)}
${#maps.containsAllValues(map, value)}
```

集□

- **#aggregates** : 配列やｺﾚｸｼｮﾝに□する集□を生成するﾕｰﾃｨﾘﾃｨﾒｿｯﾄﾞ群

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Aggregates
 * =============================================================================
 */

/*
 * 合□□を□算。配列またはコレクションが空の□合は null を返します
 */
${#aggregates.sum(array)}
${#aggregates.sum(collection)}
```

```
/*
 * 平均を□算。配列またはコレクションが空の□合は null を返します
 */
${#aggregates.avg(array)}
${#aggregates.avg(collection)}
```

メッセ□ジ

- **#messages** : □数式の中で外部化メッセージを取得するためのユーティリティメソッド群。`#{...}` □文を使用して取得するのと同じです。

```
/*
 * ========================================================================
 * See javadoc API for class org.thymeleaf.expression.Messages
 * ========================================================================
 */

/*
 * 外部化メッセージを取得します。□一のキー、□一のキーと引数
 * キーの配列/リスト/セット(この□合は外部化メッセージの配列/リスト/セットを返します)
を渡すことができます。
 * メッセージが□つからない□合は、デフォルト メッセージ( '??msgKey??' など)を返します。
 */
${#messages.msg('msgKey')}
${#messages.msg('msgKey', param1)}
${#messages.msg('msgKey', param1, param2)}
${#messages.msg('msgKey', param1, param2, param3)}
${#messages.msgWithParams('msgKey', new Object[] {param1, param2, param3, param4})}
${#messages.arrayMsg(messageKeyArray)}
${#messages.listMsg(messageKeyList)}
${#messages.setMsg(messageKeySet)}

/*
 * 外部化メッセージまたは null を取得します。指定されたキーに□するメッセージが□つからない
□合は、
 * デフォルト メッセージの代わりに null を返します。
 */
${#messages.msgOrNull('msgKey')}
${#messages.msgOrNull('msgKey', param1)}
${#messages.msgOrNull('msgKey', param1, param2)}
${#messages.msgOrNull('msgKey', param1, param2, param3)}
${#messages.msgOrNullWithParams('msgKey', new Object[] {param1, param2, param3, param4})}
${#messages.arrayMsgOrNull(messageKeyArray)}
${#messages.listMsgOrNull(messageKeyList)}
${#messages.setMsgOrNull(messageKeySet)}
```

ID

- **#ids** : (□り返し□理の中などで)□り返し登□する `id` 属性を□うためのユーティリティメソッド群

```
/*
 * ========================================================================
 * See javadoc API for class org.thymeleaf.expression.Ids
 * ========================================================================
 */

/*
 * 通常は th:id 属性に使用されます。
 * id属性□にカウンターの□を加えるので、□り返し□理の中でもユニークな□を持つことができ
ます。
 */
${#ids.seq('someId')}

/*
 * 通常は <label> タグの中の th:for 属性に使用されます。
 * #ids.seq(...) □数で生成されたidをラベルから参照することができます。
 *
 * <label> が #ids.seq(...) □数を持った要素の前にあるか後ろにあるかによって
 * "next" (ラベルが"seq"の前の□合) または "prev" (ラベルが"seq"の後の□合) □数を呼び出
```

```
します。
 */
${#ids.next('someId')}
${#ids.prev('someId')}
```

19 Appendix C: DOM Selector syntax

DOMセレクターはXPATHやCSSやjQueryの□文を参考にして、テンプレートフラグメントを特定するための□□かつパワフルな方法を提供しています。

例えば、次のセレクターはマークアップの中で `content` クラスを持った `<div>` を全て取得します:

```
<div th:include="mytemplate :: [//div[@class='content']]">...</div>
```

XPathを参考にした基本□文には次のようなものがあります:

- `/x` □在のノードの直接の子の中でxという名前を持つノード。
- `//x` □在のノードの子□の中でxという名前を持つノード。
- `x[@z="v"]` xという名前の要素で、z属性の□が"v"のもの。
- `x[@z1="v1" and @z2="v2"]` xという名前の要素で、z1,z2属性の□がそれぞれ"v1","v2"のもの。
- `x[i]` xという名前の要素の兄弟の中でi番目 のもの。
- `x[@z="v"][i]` xという名前の要素で、z属性の□が"v"の兄弟の中でi番目 のもの。

ですが、もっと□□な□文もあります:

- `x` は `//x` と全く同じ意味です(深さに□□なく `x` という名前または参照を持つ要素を探します)。
- 引数を持つ□合は要素名や参照を指定しなくても大丈夫です。ですので `[@class='oneclass']` は、class属性の□が"oneclass"の要素(ダメ)を探す、という意味の有□なセレクターになります。

高度な属性□□□能:

- `=` (equal)の他にも比□演算子が使用できます: `!=` (not equal), `^=` (starts with)と `$=` (ends with)。例: `x[@class^='section']` は `x` という名前の要素で `class` 属性の□が `section` で始まっているものを指します。
- 属性の指定は `@` で始まっていても(XPath-style)、始まっていなくても(jQuery-style)大丈夫です。ですので、 `x[z='v']` は `x[@z='v']` と同じ意味になります。
- □数属性を指定する□合は `and` でつないでも(XPath-style)、□数の修□子をつないでも(jQuery-style)大丈夫です。ですので、 `x[@z1='v1' and @z2='v2']` と `x[@z1='v1'][@z2='v2']` は同じ意味になります(`x[z1='v1'][z2='v2']` もです)。

「jQueryのような」ダイレクトセレクター:

- `x.oneclass` は `x[class='oneclass']` と同等です。
- `.oneclass` は `[class='oneclass']` と同等です。
- `x#oneid` は `x[id='oneid']` と同等です。
- `#oneid` は `[id='oneid']` と同等です。
- `x%oneref` は、xという名前を持った-要素だけではなく-ノードの中で、指定された `DOMSelector.INodeReferenceChecker` □装に□って *oneref* という参照に一致するもの指します。
- `%oneref` は、名前に□□なく-要素だけではなく-ノードの中で、指定された `DOMSelector.INodeReferenceChecker` □装に□って *oneref* という参照に一致するもの指します。参照は要素名の代わりに使用されるので、□□は□に `oneref` と同等であることに注意してください。
- ダイレクトセレクターと属性セレクターは混ぜることができます: `a.external[@href^='https']`.

上□のDOMセレクター式は:

```
<div th:include="mytemplate :: [//div[@class='content']]">...</div>
```

このようにも書くことができます:

```
<div th:include="mytemplate :: [div.content]">...</div>
```

複数の名前を持つclassのマッチング

DOMセレクターは複数の名前を持つclass属性にも対応しているので、要素がいくつかのclassを持っている場合でも、セレクターを利用することができます。

例えば、`div[class='two']` は `<div class="one two three" />` にマッチします。

任意の括弧

フラグメントインクルード属性の文は全てのフラグメント指定をDOM評価に変換するので、括弧 `[...]` はなくても大丈夫です(あってもいいですが)。

なので、次のように括弧を付けなくても、上の括弧をつけたものと同等になります:

```
<div th:include="mytemplate :: div.content">...</div>
```

ですので、まとめると:

```
<div th:replace="mytemplate :: myfrag">...</div>
```

これは `th:fragment="myfrag"` フラグメントシグネチャを探します。しかし、(HTMLには存在しませんが)もし存在するならば `myfrag` という名前のタグも探します。次のように.をつけてください:

```
<div th:replace="mytemplate :: .myfrag">...</div>
```

この場合は `class="myfrag"` の要素を探しますが、 `th:fragment` シグネチャについては気にしません。

---

1. application/xhtml+xmlコンテンツタイプで取り扱われるXML整形式のHTML5はXHTML5と呼ばれるので、ThymeleafはXHTML5をサポートしていると言ってもいいかもしれません。
2. このテンプレートは妥当なXHTMLですが、テンプレートモードとしては "VALID XHTML" ではなく、"XHTML" を選んでいます。ですので今のところ、バリデーションを有効にしなくても問題ないのですが、そうはいってもIDEにごちゃごちゃ指摘されるのも嫌ですよね。
3. 注: iterated expression の適当なのが分かりませんでした..