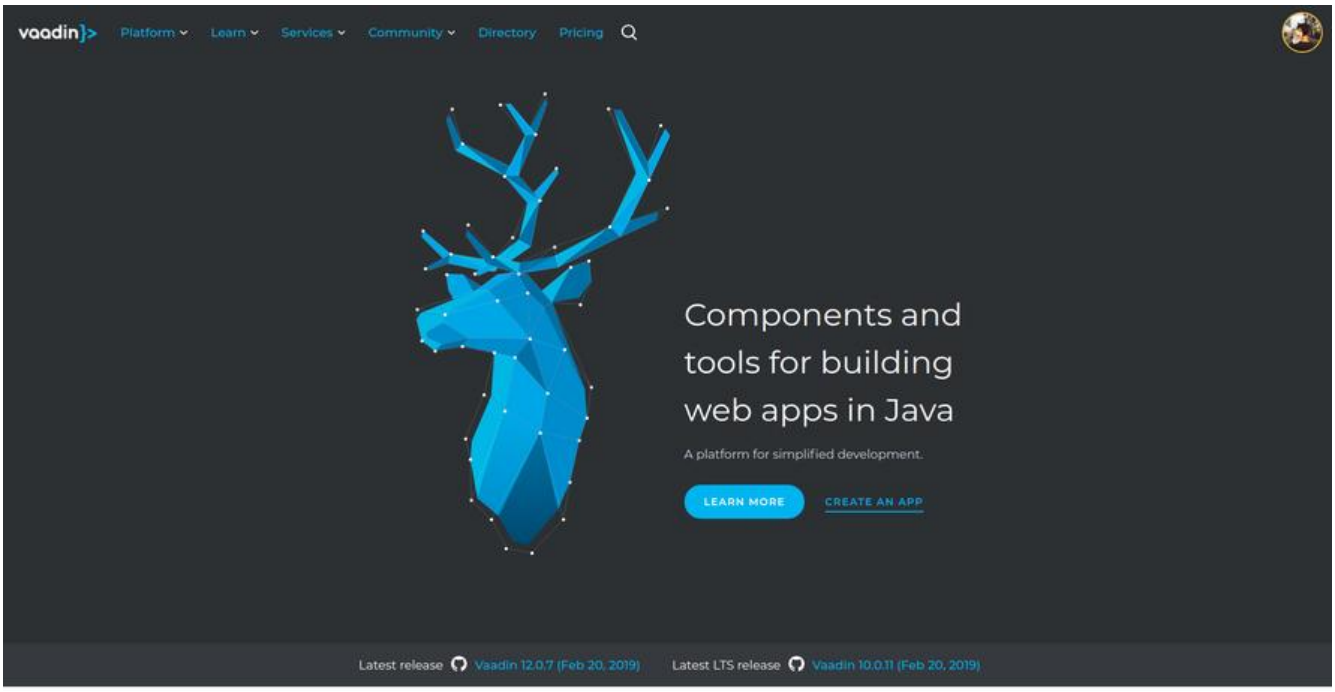
链滴

Vaadin —— Java 从后端到前端 （路由与导航）

作者：lizhongyue248

原文链接：https://ld246.com/article/1551108003098

来源网站：链滴



前段时间无聊逛 maven 仓库的时候，无意中看到一个名为 spring boot 的框架，但是却发布到了 12 的版本，我很奇怪，spring boot 最近在用，没那么高啊，然后点进去才发现是 Vaadin 12 集成 spri g boot 的库。于是一时好奇去查看了一下 Vaadin，感觉还不错，Vaadin早期叫IT Mill Toolkit，前用一种专有的Javascript实现的，开发非常复杂。2007年底，这种专有的Javascript实现就被放弃了转而拥抱GWT。2009年改名字叫Vaadin Framework。然而现在的 Vaadin 其实在也是有一席之地，

相比来说不用写 js 还是不错的，不过 css 还是要写的，可以理解是 GWT 的一个超集吧。虽然说goog e工程已经使用 dart 和 angular dart 来取代GWT 写的应用了，但我还是比较想试试使用 Vaadin 如，不过可惜的是文章实在少，只有从官网上看了，英语听力差的一批，不得已只能看文章慢慢学习啦不过先体验一番，可以的话在考虑深入学习。

定义路由

@route 注释允许您将任意组件定义为给定 URL 片段的路由目标。例如：

```
@Route("")
public class HelloWorld extends Div {
  public HelloWorld() {
    setText("Hello world");
  }
}
```

这里，我们将 Helloworld 组件定义为应用程序的默认路由目标（空路由）。您可以为不同的路由定单独的组件，如下所示：

```
@Route("some/path")
public class SomePathComponent extends Div {
  public SomePathComponent() {
    setText("Hello @Route!");
  }
}
```

每当用户访问 http://yourdomain.com/some/path 时（假设应用程序是从根上下文运行的）通过击应用程序内的链接或直接在地址栏上键入地址，SomePathComponent 组件将显示在页面上。

如果省略了 @Route 的值，则路由的路径默认将从类名派生。例如，MyEditor 将变为 “myeditor，personView 将变为 “person”，mainView 将变为 “”。

PS: 很简单的一个路由注解就完成路由设置，类似于 spring boot 的 @RequestMapping

导航的生命周期

在将导航从一种状态应用到另一种状态时，将触发许多生命周期事件。事件触发将会调用到 UI 实例侦听器和实现特殊观察者接口的附加组件

BeforeLeaveEvent

在导航期间激发的第一个事件是BeforeLeaveEvent。该事件允许延迟或取消导航，或者将导航更改转到其他目的地。此事件将传递到实现 BeforeLeaveObserver 并在导航开始前附加到 UI 的任何组实例。也可以使用UI中的 addBeforeLeaveListener（beforeLeaveListener）方法为此事件注册独立侦听器。

此事件的一个典型用例是，在导航到应用程序的其他部分之前，询问用户是否要保存任何未保存的更。

Postpone 推迟导航

BeforeLeaveEvent有一个 Postpone 方法，可用于推迟当前导航转换，直到满足特定条件。

E.g. 在离开页面前请求用户确认：

```
public class SignupForm extends Div implements BeforeLeaveObserver {
    @Override
    public void beforeLeave(BeforeLeaveEvent event) {
        if (this.hasChanges()) {
            ContinueNavigationAction action = event.postpone();
            ConfirmDialog.build("Are you sure you want to leave this page?")
                    .ifAccept(action::proceed).show();
        }
    }

    private boolean hasChanges() {
        // no-op implementation
        return true;
    }
}
```

使用 postpone 方法将会暂时中断观察者和监听者，当他恢复以后，将会启用推迟的观察者之后的观者。

例如，我们假设当前页面有 A B C 三个观察者（即实现了 Observer 结尾接口），按照这些顺序通知些观察者，如果 B 调用推迟，对 C 的调用以及转换过程的其余部分将被推迟。如果 B 推迟的转换没恢复，C 将不会收到有关此事件的通知，并且转换永远不会结束。但是，如果 B 执行其 ContinueNavigationAction 以恢复转换，则从中断的位置继续。因此，A 和 B 不再被调用，但 C 被通知。

任何时候最多可以推迟一个导航事件；当前一个导航事件处于推迟状态时启动新的导航转换将取消推状态。之后，执行之前保存的 ContinueNavigationAction 将没有任何效果

PS: 其实类似于 axios 的 路由前置守卫

BeforeEnterEvent

在导航期间触发的第二个事件是 BeforeEnterEvent。它允许将导航更改为转到其他目的地。此事件常用于响应特殊情况，例如，如果没有要显示的数据或用户没有适当的权限。

只有在通过 BeforeLeaveEvent 的任何 postpone 都已继续之后，才会激发该事件。

此事件将传递到任何实现 BeforeEnterObserver 的组件实例，该实例将在导航完成后附加到 UI。请意，在分离并到达组件或使UI 导航到的位置匹配之前，将激发该事件。也可以使用 UI 中的 addBeforEnterListener（beforeEnterListener） 方法为此事件注册独立的侦听器。

Reroute

如果需要在某些状态下显示完全不同的信息，可以使用 BeforeEnterEvent 或 BeforeLeaveEvent 动重新路由。重新路由后，不会再激发任何其他侦听器或观察器。相反，将根据新的导航目标触发新的航阶段，而事件将根据该导航触发。

E.g. 下面的例子发生在进入 BlogList 没有任何结果的时候

```
@Route("no-items")
public class NoItemsView extends Div {
    public NoItemsView() {
        setText("No items found.");
    }
}

@Route("blog")
public class BlogList extends Div implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        // implementation omitted
        Object record = getItem();

        if (record == null) {
            event.rerouteTo(NoItemsView.class);
        }
    }

    private Object getItem() {
        // no-op implementation
        return null;
    }
}
```

rerouteto 有几个重载来服务不同的用例。

AfterNavigationEvent

导航过程中的第三个也是最后一个触发事件是 AfterNavigationEvent。此事件通常用于在实际导航成后更新 UI 的各个部分。例如，调整 breadcrumb 组件的内容，并在菜单中直观地将活动条目标记 active。

事件在 beforeenterEvent 和更新附加的 UI 组件后激发。此时，可以预期当前的导航状态将实际显给用户，也就是说，不会有任何进一步的重新路由或类似的情况。

此事件将传递到完成导航后附加的实现 AfterNavigationOnBServer 的任何组件实例。也可以使用 UI 中的 AddAfterNavigationListener（AfterNavigationListener） 方法为此事件注册独立的侦听器。

```
public class SideMenu extends Div implements AfterNavigationObserver {
    Anchor blog = new Anchor("blog", "Blog");

    @Override
    public void afterNavigation(AfterNavigationEvent event) {
        boolean active = event.getLocation().getFirstSegment()
                .equals(blog.getHref());
        blog.getElement().getClassList().set("active", active);
    }
}
```

PS:其实就是路由后置守卫

路由器布局和嵌套路由器目标

RouterLayout

使用 @route（“path”） 定义路由时，默认情况下，组件将呈现在页面上的 <body> 标记内（hasElement.getElement() 返回的元素附加到 <body>）。

可以使用 route.layout() 方法定义父布局。例如，在名为 Mainlayout 的布局中呈现 CompanyComponent，代码如下：

```
@Tag("div")
@Route(value="company", layout=MainLayout.class)
public class CompanyComponent extends Component {
}
```

所有用作父布局的布局都必须实现 RouterLayout 接口。

如果有多个路由器目标组件使用相同的父布局，那么当用户在子组件之间导航时，父布局实例将保持变。

PS:类似于 HTML 的元素嵌套

具有@parentlayout的多个父布局

在某些情况下，可能需要在应用程序中为父布局提供父布局。一个例子是，我们有一个用于所有内容主布局和一个可重用为视图的菜单栏。

为此，我们可以进行以下设置：

```
public class MainLayout extends Div implements RouterLayout {
```

```
}

@ParentLayout(MainLayout.class)
public class MenuBar extends Div implements RouterLayout {
    public MenuBar() {
        addMenuElement(TutorialView.class, "Tutorial");
        addMenuElement(IconsView.class, "Icons");
    }
    private void addMenuElement(Class<? extends Component> navigationTarget,
            String name) {
        // implementation omitted
    }
}

@Route(value = "tutorial", layout = MenuBar.class)
public class TutorialView extends Div {
}

@Route(value="icons", layout = MenuBar.class)
public class IconsView extends Div {
}
```

在这种情况下，我们将拥有一个始终封装 MenuBar 的 MainLayout，而 MenuBar 又封装 TutorialView 或 IconsView，具体取决于我们导航到的位置。

在这个示例中，我们有两个父层，但是嵌套布局的数量没有限制。

使用 @routeprefix 的 parentlayout 路由控制

在某些情况下，父布局应该通过添加到路由位置来补充导航路由。

这可以通过用 @RoutePrefix("prefix_to_add") 注解父布局来完成。

```
@Route(value = "path", layout = SomeParent.class)
public class PathComponent extends Div {
    // Implementation omitted
}

@RoutePrefix("some")
public class SomeParent extends Div implements RouterLayout {
    // Implementation omitted
}
```

在本例中，PathComponent 将接收的路由是 some/path，就像前面提到的 somePathComponent 一样。

绝对路由

有时，我们可能有一个设置，我们希望在许多部分中使用相同的父组件，但在某些情况下，不使用父中的任何 @RoutePrefix，或仅将它们用于定义的部分。

在这些情况下，我们可以将 absolute=true 添加到 @Route 或 @RoutePrefix 注释中。

因此，如果我们想在 SomeParent 布局的许多地方使用某些内容，但不想将路由前缀添加到导航路中，我们可以用以下方式构建一个类 MyContent：

```
@Route(value = "content", layout = SomeParent.class, absolute = true)
public class MyContent extends Div {
    // Implementation omitted
}
```

在这种情况下，即使完整的链路径应该是 some/content，我们实际上得到路径是 content 正如我们定义的，这应该是绝对的。

当在链的中间有绝对定义时，也可以这样做，例如：

```
@RoutePrefix(value = "framework", absolute = true)
@ParentLayout(SomeParent.class)
public class FrameworkSite extends Div implements RouterLayout {
    // Implementation omitted
}

@Route(value = "tutorial", layout = FrameworkSite.class)
public class Tutorials extends Div {
    // Implementation omitted
}
```

在这种情况下，绑定的路由将是 framework/tutorial 即使整个链接是 some/framework/tutorial

路由和 URL 参数

导航目标的 URL 参数

支持通过URL传递参数的导航目标应实现 HasUrlParameter 接口，并使用泛型定义参数类型。通过种方式，路由器 API 可以提供一种类型安全的方式来构造指向特定目标的 URL。

HasUrlParameter 定义路由器根据从 URL 提取的值调用的 setParameter 方法。该方法将始终在激导航目标之前被调用。

在下面的代码段中，我们定义了一个导航目标，它接受一个字符串参数并从中生成一个 hello 字符串然后目标将其设置为自己的导航文本内容。

```
@Route(value = "greet")
public class GreetingComponent extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event, String parameter) {
        setText(String.format("Hello, %s!", parameter));
    }
}
```

启动时，此导航目标将自动配置为格式 greet/<anything> 的每个路径，除非已将具有精确 @Route 的单独导航目标配置为匹配 greet/<some specific path> ,因为在解析URL时，精确导航目标优先。

导航目标的可选 URL 参数

可以使用 @OptionalParameter 对 URL 参数进行注释，使路由同时匹配 greet 和 greet/<anything>。

```
@Route("greet")
public class OptionalGreeting extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
            @OptionalParameter String parameter) {
        if (parameter == null) {
            setText("Welcome anonymous.");
        } else {
            setText(String.format("Welcome %s.", parameter));
        }
    }
}
```

另外，对于可选参数，特定路由将优先于参数化路由。

导航目标的通配符 URL 参数

在需要更多参数的情况下，还可以使用 @WildcardParameter 对 URL 参数进行注释，以使路由匹配候语以及之后的任何内容，例如问候语 /one/five/three。

```
@Route("greet")
public class WildcardGreeting extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
            @WildcardParameter String parameter) {
        if (parameter.isEmpty()) {
            setText("Welcome anonymous.");
        } else {
            setText(String.format("Handling parameter %s.", parameter));
        }
    }
}
```

通配符参数的参数永远不会为空。

更具体的路径将优先于通配符目标。

查询参数

也可以获取包含在 URL 中的查询参数。e.g. ?name1=value1&name2=value2.

可以通过 Location 类的 getQueryParameters() 方法访问这些查询参数。位置类可以通过 setParamter 方法的 BeforeEvent 参数获得。

Location 对象表示由路径段和查询参数组成的相对 URL，但不用主机名，e.g. new Location("foo/ba/baz?name1=value1").

```
@Override
public void setParameter(BeforeEvent event,
        @OptionalParameter String parameter) {

    Location location = event.getLocation();
    QueryParameters queryParameters = location.getQueryParameters();

    Map<String, List<String>> parametersMap = queryParameters.getParameters();
}
```

getQueryParameters() 支持与同一个键关联的多个值。 Example: https://example.com/?one=1&wo=2&one=3 3将生成对应的映射 {"one" : [1, 3], "two": [2]}}.

URL 生成

路由器公开了获取已注册导航目标的导航 URL 的方法。

对于一个普通的导航目标，请求是一个简单的调用 Router.getUrl(Class target)。

```
@Route("path")
public class PathComponent extends Div {
  public PathComponent() {
    setText("Hello @Route!");
  }
}

public class Menu extends Div {
    public Menu() {
        String route = UI.getCurrent().getRouter()
                .getUrl(PathComponent.class);
        Anchor link = new Anchor(route, "Path");
        add(link);
    }
}
```

在这种情况下，返回的 URL 将简单地解析为 **路径**，但在我们在父布局有添加部分路径情况下，手工成路径可能不那么简单。

带参数导航目标的 URL 生成

对于具有所需参数的导航目标，参数被赋予解析器，返回的字符串将包含参数，e.g. Router.getUrl(Class target, T parameter)

```
@Route(value = "greet")
public class GreetingComponent extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
            String parameter) {
        setText(String.format("Hello, %s!", parameter));
    }
}
```

```
public class ParameterMenu extends Div {
    public ParameterMenu() {
        String route = UI.getCurrent().getRouter()
                .getUrl(GreetingComponent.class, "anonymous");
        Anchor link = new Anchor(route, "Greeting");
        add(link);
    }
}
```

在路线之间导航

您可以使用 RouterLink 组件创建链接，以引导到应用程序中的路由目标。

带或不带 url 参数的导航目标的 RouterLink 示例

```
void routerLink() {
    Div menu = new Div();
    menu.add(new RouterLink("Home", HomeView.class));
    menu.add(new RouterLink("Greeting", GreetingComponent.class, "default"));
}
```

带URL参数的 GreetingComponent 组件

```
@Route(value = "greet")
public class GreetingComponent extends Div
        implements HasUrlParameter<String> {

    @Override
    public void setParameter(BeforeEvent event,
            String parameter) {
        setText(String.format("Hello, %s!", parameter));
    }
}
```

也可以使用普通的类型链接进行导航，但这些链接会导致页面重新加载。相反，使用 RouterLink 导会获取新组件的内容，该组件在不重新加载页面的情况下就地更新。

通过向常规链接添加 router-link 属性，您可以告诉框架它应在不重新加载的情况下处理导航，e.g. <a router-link href="company">Go to the company page。

要从服务器端触发导航，请使用 UI.navigate(String)，其中 String 参数是要导航到的位置。还有，UI navigate(Class<? extends Component> navigationTarget) 或 navigate(Class<? extends C> navigationTarget, T parameter)，这样就不必手动生成路由字符串。这将触发浏览器位置的更新并添加的历史记录状态条目。单击按钮时指向 company 路线目标的示例导航：

```
NativeButton button = new NativeButton("Navigate to company");
button.addClickListener( e-> {
    button.getUI().ifPresent(ui -> ui.navigate("company"));
});
```

即使会话 session 已过期，路由器链接也可以工作，因此您应该更喜欢使用这些链接，而不是处理导服务器端。

路由器异常处理

vaadin 对于导航目标有特殊的支持，因为在**导航过程**中引发了未处理的异常而激活这些目标，以便用户显示“错误视图”。

这些目标通常与常规导航目标的工作方式相同，尽管它们通常没有任何特定的 @Route，因为它们是任意 URL 显示的。

错误导航根据导航期间引发的异常类型解析为 target。

在启动时，将收集实现接口 HasErrorParameter<T extends Exception> 的所有类，以便在导航期用作异常 targets。

例如，这里是 NotFoundException 的默认目标，当给定的 URL 没有目标时，将显示该目标。

RouteNotFoundError for NotFoundException during routing

```
@Tag(Tag.DIV)
public class RouteNotFoundError extends Component
        implements HasErrorParameter<NotFoundException> {

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
            ErrorParameter<NotFoundException> parameter) {
        getElement().setText("Could not navigate to '"
                + event.getLocation().getPath() + "'");
        return HttpServletResponse.SC_NOT_FOUND;
    }
}
```

这将返回 404 的 HTTP 响应并向用户显示设置的文本。

异常匹配将首先按异常原因运行，然后按异常超类型运行。

实现的默认异常为 NotFoundException（404） 的 RouteNotFoundError ，Java.lang.Exception（00） 的 InternalServerError 。

默认的异常处理程序可以通过如下方式进行扩展来重写：

Custom route not found that is using our application layout

```
@ParentLayout(MainLayout.class)
public class CustomNotFoundTarget extends RouteNotFoundError {

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
            ErrorParameter<NotFoundException> parameter) {
        getElement().setText("My custom not found class!");
        return HttpServletResponse.SC_NOT_FOUND;
    }
}
```

作为一个更复杂的示例，我们可以有一个仪表板，它可以收集和向用户显示小部件，并且可以有不应未经身份验证的用户显示的小部件。出于某种原因，为未经身份验证的用户加载 ProtectedWidget。

集合本应捕获受保护的小部件，但出于某种原因实例化了它，但幸运的是，该小部件检查创建时的身验证，并抛出 AccessDeniedException

此未处理的异常在导航过程中传播，并由 AccessDeniedExceptionHandler 处理，该处理程序仍保主布局的菜单栏，但显示发生异常的信息。

错误加载受保护的小部件时访问被拒绝的异常示例

```
@Route(value = "dashboard", layout = MainLayout.class)
@Tag(Tag.DIV)
public class Dashboard extends Component {
    public Dashboard() {
        init();
    }

    private void init() {
        getWidgets().forEach(this::addWidget);
    }

    public void addWidget(Widget widget) {
        // Implementation omitted
    }

    private Stream<Widget> getWidgets() {
        // Implementation omitted, gets faulty state widget
        return Stream.of(new ProtectedWidget());
    }
}

public class ProtectedWidget extends Widget {
    public ProtectedWidget() {
        if (!AccessHandler.getInstance().isAuthenticated()) {
            throw new AccessDeniedException("Unauthorized widget access");
        }
        // Implementation omitted
    }
}

@Tag(Tag.DIV)
public abstract class Widget extends Component {
    public boolean isProtected() {
        // Implementation omitted
        return true;
    }
}

@Tag(Tag.DIV)
@ParentLayout(MainLayout.class)
public class AccessDeniedExceptionHandler extends Component
        implements HasErrorParameter<AccessDeniedException> {

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
            ErrorParameter<AccessDeniedException> parameter) {
```

```
        getElement().setText(
            "Tried to navigate to a view without correct access rights");
        return HttpServletResponse.SC_FORBIDDEN;
    }
}
```

异常目标可以定义 ParentLayouts，并且在进行导航之前和之后发送的 NavigationEvent 将与正常航相同。

一个异常只能有一个异常处理程序（只允许扩展实例）。

重新路由到错误视图

可以从 BeforeEnterEvent 和 BeforeLeaveEvent 重新路由到为异常注册的错误视图。

重新路由是通过使用其中一个重载来完成的，该重载用于只将异常类重新路由到目标或添加自定义错消息。

重新路由到错误视图

```
public class AuthenticationHandler implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        Class<?> target = event.getNavigationTarget();
        if (!currentUserMayEnter(target)) {
            event.rerouteToError(AccessDeniedException.class);
        }
    }

    private boolean currentUserMayEnter(Class<?> target) {
        // implementation omitted
        return false;
    }
}
```

如果重新路由方法捕获到异常，并且需要添加自定义消息，则可以使用 rerouteToError(Exception, String) 方法设置自定义消息。

包含自定义消息的日志示例错误视图

```
@Tag(Tag.DIV)
public class BlogPost extends Component implements HasUrlParameter<Long> {

    @Override
    public void setParameter(BeforeEvent event, Long parameter) {
        removeAll();

        Optional<BlogRecord> record = getRecord(parameter);

        if (!record.isPresent()) {
            event.rerouteToError(IllegalArgumentException.class,
                    getTranslation("blog.post.not.found",
                            event.getLocation().getPath()));
        } else {
```

```
            displayRecord(record.get());
        }
    }

    private void removeAll() {
        // NO-OP
    }

    private void displayRecord(BlogRecord record) {
        // NO-OP
    }

    public Optional<BlogRecord> getRecord(Long id) {
        // Implementation omitted
        return Optional.empty();
    }
}

@Tag(Tag.DIV)
public class FaultyBlogPostHandler extends Component
        implements HasErrorParameter<IllegalArgumentException> {

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
            ErrorParameter<IllegalArgumentException> parameter) {
        Label message = new Label(parameter.getCustomMessage());
        getElement().appendChild(message.getElement());

        return HttpServletResponse.SC_NOT_FOUND;
    }
}
```

获取已注册的路由

要检索应用程序中所有已注册的路由，可以使用：

```
Router router = UI.getCurrent().getRouter();
List<RouteData> routes = router.getRoutes();
```

RouteData 对象包含有关已定义路由的所有相关信息，如 URL、参数和父布局。

按父布局获取已注册路由

要获取由父布局定义的所有路由，可以使用：

```
Router router = UI.getCurrent().getRouter();
Map<Class<? extends RouterLayout>, List<RouteData>> routesByParent = router.getRoutes
yParent();
List<RouteData> myRoutes = routesByParent.get(MyParentLayout.class);
```

更新导航页面标题

导航期间有两种更新页面标题的方法：

- 使用 @PageTitle 注解
- 实现 HasDynamicTitle

这两种方法是互斥的：在同一个类上同时使用这两种方法将在启动时导致运行时异常。

使用 @PageTitle 注解

更新页面标题的最简单方法是在组件类上使用 @PageTitle 注释。

```
@PageTitle("home")
class HomeView extends Div {

  HomeView(){
    setText("This is the home view");
  }
}
```

@PageTitle 注释仅从实际导航目标读取；不考虑其超类或其父视图。

动态设置页面标题

实现 HasDynamicTitle 接口使我们可以在运行时从 Java 更改标题：

```
@Route(value = "blog")
class BlogPost extends Component
       implements HasDynamicTitle, HasUrlParameter<Long> {
  private String title = "";

  @Override
  public String getPageTitle() {
    return title;
  }

  @Override
  public void setParameter(BeforeEvent event,
       @OptionalParameter Long parameter) {
    if (parameter != null) {
      title = "Blog Post #" + parameter;
    } else {
      title = "Blog Home";
    }
  }
}
```

结束语

路由这一块还不错，相比于 vertx 至少他的注解是很友好的（其实两者不是一性质哈哈），不过 html 组件构建方面有点麻烦，慢慢学习适应下咯只有，不过重点是 资料少！资料少！资料少！ 啊！不过喜欢（逃。。。