ANDRÉS ALMIRAY

FXMLLoaderでFXMLのカスタム動作を定義する

FXMLを用いた、JavaFXアプリケーションへのカスタム動作の注入

本シリーズの1回目の記事では、FXMLフォーマットの基本のほか、FXMLLoaderという便利なユーティリティについて説明しました。名前からもわかるように、FXMLLoaderはFXMLリソースを読み込み、その内容を解析し、FXMLファイルに含まれる定義に基づいてSceneGraphを構築します。しかし、FXMLLoaderでできることはそれだけにとどまりません。特に、最近のJava開発で一般的に利用される技術である依存性注入（Dependency Injection /DI）と組み合わせれば、その可能性は大きく広がります。FXMLLoaderになじみがないという方は、本記事の内容が理解しづらい可能性があるため、1回目の記事をご覧になることをお勧めします。

設定の中には、プログラム的なアプローチで調整したいと開発者が考えるものもあります。FXMLフォーマットの設計者は、そのことを認識していました。たとえば、UIが作成された直後、ユーザーがアプリケーションを操作できるようになる前に、ListViewやTableViewに一連の動的な項目を挿入したい場合がその例です。このようなタスクを実行するためには、アプリケーションの初期化時に特定のタイミングで呼び出されるカスタム動作をJavaFX開発者が定義できることが必要です。また、カスタマイズの対象となるターゲット・ウィジェット（ListViewなど）を見つける方法も必要です。この動作は、最近のDIフレームワークが提供するものに非常によく似ています。FXMLの設計者が実現しているのは、まさにその機能です。ここからは、その仕組みについて見ていきます。

コントローラ

1回目の記事で紹介したJavaFXビューには、アプリケーション固有の動作はありませんでした。しかし、役割が分離されているという意味で、これはよいことだと言えます。ビューの要素はUIの外見を定義すべきですが、動作を定義すべきではありません。動作を定義するのは、コントローラの役割です。FXMLの用語では、DIに関与し、関連付けられたビューで定義されたUI要素がトリガーするイベントに反応できるオブジェクトをコントローラと呼びます。JavaFXアプリケーションのメイン・エントリ・ポイント

は、特定のクラス（javafx.application.Application）を拡張する必要があります。それとは違い、コントローラ・クラスは、どんな型でも構いません。また、実装する必要があるマーカー・インタフェースもありません。そのため、必要に応じて自由に型やコントローラ階層を作ることができます。

まずは、ボタンとListViewからなるUIを備えた単純なアプリケーションから始めます。リストの内容はプログラムで初期化し、ボタンをクリックするたびに新しい項目が追加されるものとします。このUIのFXML定義は次のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.ListView?>
<?import javafx.scene.layout.VBox?>
<VBox xmlns="http://javafx.com/javafx"
   xmlns:fx="http://javafx.com/fxml"
   fx:controller="sample.AppController"
   prefHeight="300.0" prefWidth="100.0">
   <Button text="Add item" onAction="#addItem"/>
   <ListView fx:id="theList"/>
</VBox>
```

このファイルでは、本シリーズの記事で初めて扱ういくつかの機能が使われています。まずは、UIのルート要素に適用されているfx:controller属性です。この属性は、指定されたクラスのオブジェクトのインスタンスを作成し、そのインスタンスをこのビューのコントローラに設定するようにFXMLLoaderに指示するものです。このようにインスタンス化されるオブジェクトのクラスには、引数なしのpublicなコンストラクタが必要です。

2つ目の機能は、ボタンの定義で使われているonActionという名前の属性です。この値のフォーマットに注目してください。文字#の後に識別子が記述されています。これは、コントローラ・クラスに定義されているメソッドです。

最後に、ListViewでのfx:idの使い方を確認してください。これは、属性名を変数名として用いてListViewへの内部参照を保持するようにFXMLLoaderに指示するものです。この変数は、コントローラにListViewへの参照を注入する際に使われます。次に、コントローラを見てみます。

```
// sample/AppController.java
package sample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;

public class AppController {
   @FXML private ListView<String> theList;

   public void initialize() {
      for (int i = 0; i < 5; i++) {
         theList.getItems().add("Item " + i);
      }
   }

   public void addItem(ActionEvent ignored) {
      theList.getItems().add("Item "
         + theList.getItems().size());
   }
}
```

おわかりのように、コントローラは通常のJavaクラスのように見えます。ただし、本シリーズの記事では初めて扱う要素である@FXMLアノテーションが使われています。これは、FXMLLoaderがターゲット・コントローラのインスタンスにDIを行うことができるようにするために必要なリンクです。このアノテーションでは、ListViewを参照するために定義した変数名を利用しています。フィールド名が変数名と同じであり、ListViewに含めることができる要素の型（この場合は、単純なJava文字列）が定義されている点に注目してください。FXMLLoaderは、このネーミング規則に従ってフィールド名と変数名、およびフィールドの型とウィジェットの型を照合します。そのため、フィールドがChoicebox<String>として定義されている場合は、実行時例外が発生します。お使いのIDEがFXMLの規約を

認識するものであれば、こういったエラーは避けることができます。

> JSR 330では、Javaプログラムで依存性注入を実行する際の**基本原理が定義されています。**

2つ目の特徴は、引数がないinitializeという名前のpublicメソッドが定義されている点です。このメソッドは、すべてのウィジェットの注入が行われた後、FXMLLoaderによって呼び出されます。この時点で、ウィジェットの参照が完全に解決され、注入も完了しています。そのため、データの初期化には絶好のタイミングです。今回の例では、まさにこの処理によってListViewにデフォルトのデータが提供されています。メソッドが別の名前である場合、FXMLLoaderに呼び出されることはありません。これは、FXMLLoaderの規約によって定義されている強い制約です。

最後に、少しだけアクション・ハンドラを見てみます。このハンドラは、ボタンに関連付けられているonActionハンドラの値に一致する名前のメソッドとして定義されています。FXMLファイルを見れば、おわかりいただけるでしょう。このハンドラは、ボタンのクリックに反応します。そのため、引数としてActionEventを受け取ります。この型は、コードを反応させたいイベントの種類によって変わるため、注意してください。たとえば、コードをマウスの移動に反応させたい場合は、引数の型はMouseEventにする必要があります。このイベント・ハンドラは、ボタンをクリックすると即時に呼び出され、UIスレッド内でも呼び出されます。このちょっとしたアプリケーションでは、ハンドラがリストに新しい項目を追加しているだけであるため、問題にはなりませんが、この設定には注意が必要です。通常、アプリケーションでは、UIスレッドの外側で行う必要がある、ネットワーク接続のオープン、データベースやファイルへのアクセスや、さまざまな計算を行うからです。データの準備ができたら、UIスレッド内のUI要素にデータを戻す必要があります。したがって、JavaFXが提供するスレッド構造には十分注意してください。スレッド処理を適切に行っていない場合、実行時例外、アプリケーションの無応答、データの破損や、さらに深刻な事態につながる可能性があります。

以上の内容をまとめておきます。コントローラには、FXMLビューの動作を定義する役割があります。ビューとコントローラは、FXMLビューのルート・ノードにあるfx:controller属性を使って関連付けます。FXMLLoaderは、自動的にコントローラのインスタンスを作成します。大半の場合、これが問題になることはありませんが、別の方法でコントローラのインスタンスを作成しなければならない場合はどうでしょうか。たとえば、@FXMLアノテーションが付加されたすべての要素の注入が完了する前に、追加のカスタマイズを行わなければならないような場合です。そのような場合でも、心配は無用です。これは、DIフレームワークを使って行うことができます。次のセクションでは、その方法を説明します。

JSR 330と依存性注入

JSR 330では、JavaプログラムでDIを実行する際の基本原理が定義されています。そのための実装は何種類かありますが、説明を簡潔にするため、個別のDIフレームワークの設定に関する説明は行いません。以降の例では、JSR 330で提供される標準APIを使用します。

それでは、先ほどの例で、単純なforループを使ってListViewにデータを挿入する代わりに、データ・プロバイダ・メカニズムを使ってみます。このデータ・プロバイダ・メカニズムは、次のようなインタフェースを使って定義します。

```
// sample/DataProvider.java
package sample;

import java.util.Collection;

public interface DataProvider<T> {
   Collection<T> getData();
}
```

このインタフェースは、アプリケーションでの必要に応じて自由に実装できます。たとえば、ファイルや、JDBCなどによるデータベース接続からデータを取得できます。次に、アップデートしたコントローラ・クラスを示します。

```
// sample/AppController.java
package sample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;

import javax.inject.Inject;

public class AppController {
   @FXML private ListView<String> theList;
```

```
    private final DataProvider<String> dataProvider;

    @Inject
    public AppController(DataProvider<String> dp) {
        this.dataProvider = dp;
    }

    public void initialize() {
        theList.getItems().addAll(dataProvider.getData());
    }

    public void addItem(ActionEvent ignored) {
         theList.getItems().add("Item "
           + theList.getItems().size());
        }
    }
```

コントローラに単一引数のコンストラクタが定義されており、さらに@Injectアノテーションが付加されている点に注意してください。ただし、この単純な変更は、コントローラのインスタンスを作成するためのFXMLLoaderの規約と矛盾します。FXMLLoaderとコントローラ・クラスを連携させるためには、別の方法が必要になります。幸いなことに、その方法が用意されています。FXMLLoaderにControllerFactoryを設定すると、別の戦略を使ってコントローラのインスタンスを作成することができます。そのことを知っていれば、後はアプリケーションのエントリ・ポイントに戻り、FXMLLoaderのインスタンスにその新しい情報を設定するだけです。次に例を示します。

```
// sample/AppMain.java
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
```

```
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.net.URL;

public class AppMain extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        // DIフレームワーク経由で初期化する
        AppController controller = ...
        URL location = getClass().getResource("app.fxml");
        FXMLLoader fxmlLoader = new FXMLLoader(location);

        // ControllerFactoryを設定する
        fxmlLoader.setControllerFactory(param -> controller);

        VBox vbox = fxmlLoader.load();
        Scene scene = new Scene(vbox);
        primaryStage.setScene(scene);
        primaryStage.sizeToScene();
        primaryStage.show();
    }
}
```

これで、コントローラのインスタンスがDIコンテナからフェッチされます。このフェッチ操作はフレームワークによって異なるため、例示は省略していますが、簡単なControllerFactoryを定義するラムダ式を適用することもできます。AppControllerクラスがDIコンテナによって管理されるようになったことにより、メソッドに@PostConstructアノテーションを付加するなどの他の機能も利用できるため、コントローラをさらにカスタマイズできます。ただし、@PostConstructアノテーションが付加されたメソッドは、コントローラのインスタンスに@FXMLアノテーションが付加されたフィールドが注入される前に呼び出されるという点は覚えておいてくださ

い。なお、この設定の副作用として、fx:controllerの値は使われなくなります。それでも、DIコンテナによって提供されるコントローラの実際の型を指定しておくとよいでしょう。そうしておくと、@FXMLアノテーションで注入できるウィジェットの型および名前と、コントローラが提供するイベント・ハンドラの名前に関するヒントがIDEに表示されるためです。別の方法として、コントローラのプロパティをFXMLLoaderで直接設定することもできます。しかし、値の遅延計算を行う際に役立つと思われるため、今回はファクトリによるアプローチを紹介しました。

ここまでは、ビューとコントローラについて説明してきました。しかし、ご存知のように、UIアプリケーションの設計に使用できるデザイン・パターンがいくつかあります。その多くはモデル・ビュー・コントローラ（MVC）パターンから派生したもので、モデル・ビュー・プレゼンタやモデル・ビュー・ビューなどがあります。これらのパターンを使うと、役割によって各メンバーを明確に区別できるため、整然とした実装になり、アプリケーションの開発もやりやすくなります。

今回説明した例で、どこがモデルに相当するのかと疑問に思う方もいらっしゃるかもしれません。今回の例では、モデルはコントローラ内に暗黙的に存在しています。ObservableList<String>のような形ですべての項目の状態を保持する必要がある場合、このリストを配置する場所として最初に思いつくのはコントローラ自体でしょう。今回の例のような小さなアプリケーションの場合、その選択でも誤りとは言えません。しかし、アプリケーションの機能が増加し、複雑になってくると、この方式にはスケーラビリティが不足していることにすぐ気づくでしょう。そのため、MVCパターンやその派生パターンに基づいて考えることを強くお勧めします。つまり、動作はコントローラのみに配置し、データおよび状態はモデル・クラスに残しておきます。このクラスのインスタンスは、手動でもDIコンテナ経由でも作成できます。特定のアプリケーションの要件に従って自由に決めて構いません。

FXMLは、JavaFXのUIを宣言的に定義する方法を提供しますが、動作の定義に使うことはできません。

まとめ

FXMLは、JavaFXのUIを宣言的に定義する方法を提供しますが、動作の定義に使うことはできません。必要になる動作は、コントローラを指定して記述できます。コントローラは、DIを使用して、設定が必要なUIコンポーネントにアクセスすることができます。また、FXMLファイルで特殊なフォーマットを使って、UIコンポーネント内に直接フック

できるイベント・ハンドラを定義することも可能です。デフォルトの戦略を使うと、FXMLLoaderにコントローラのインスタンスを作成させることができます。しかし、別のインスタンス化メカニズムが必要な場合は、この戦略を変更することもできます。</article>

Andrés Almiray：JavaおよびGroovyの開発者で、Java Champion。17年を超えるソフトウェアの設計および開発の経験があり、Javaの初期の頃から、Webアプリケーションやデスクトップ・アプリケーションの開発に関わっている。熱心なオープン・ソースの信奉者で、Groovy、JMatter、Asciidoctorなどの有名プロジェクトに参加。Griffonフレームワークの創設メンバーであり、現在のプロジェクト・リードである。JSR 377の仕様リードも務めている。

1段組みの新しい記事レイアウトはお気に召しましたか?
ぜひ、JAVAMAG_US@ORACLE.COMまでご意見をお寄せください。

learn more

オラクルによるFXMLに関するチュートリアル