WPF アクション

[WPF][.Net][Silverlight][Universal Windows Platform][C#]

・WPF にはアクションを扱うための共通の方法が 3 つある
　・イベント、コマンド、およびトリガ

アクションの原則

・3 つのメカニズム (イベント、コマンド、トリガ) すべてに当てはまる原則

3 つの原則に基づいて機能

・要素の合成
　・表示ツリーを構築する方法にアクションを対応させる
・疎結合
　・イベントのソースとイベントを処理するコードが密接に結合されていると問題が生じる
・宣言型アクション
　・宣言型のプログラミングをシステムのすべての側面で実現する必要がある

要素の合成

・ボタンは実際には複数の要素から構成されるため、イベント処理で問題が生じる
・コントロールモデルの原則は、要素の合成、リッチコンテンツ、簡単なプログラミングモデル

簡単なプル蔵民具モデルを実現するには

・クリックイベントをリッスンするコードを開発者に親しみやすいものにする必要がある
・Click イベントにハンドラを添付するだけのものであることが望まれる

```
Button b = new Button();
b.Content = "Click";
b.Click +=
    delegete { MessageBox.Show("Clicked!"); }
```

このコードはうまく機能するように見えるが、クリックされているのはボタンそのものではなく、ボタンの表示を構成する要素

・WPF ではこれをシームレスに機能させるためにルーティングイベントという概念を導入

■ルーティングイベント

・コンテンツとしてボタンを含むように変更する

```
Button b = new Button();
b.Content = new Button();
b.Click +=
    delegete { MessageBox.Show("Clicked!"); }
```

・コンテンツとしてボタンを含むように変更した場合、内側もしくは外側のいずれかのボタンをクリックするとイベントが発生する

イベントの合成は、イベントだけでなく、アクション処理のすべての側面に影響を与える

疎結合

・Button のイベントを見ると、直接的なマウスイベント (MouseUp,MouseDown など) と Click イベントの両方をサポートすることが分かる
・Click はマウスイベントよりも上位に位置する抽象概念 (フォーカスがある状態でスペースキーや規定ボタンである場合 Enter キー押下で発生する)
・Click はセマンティック (意味的) イベントで、マウスイベントは、フィジカル (物理的) イベント

Click イベントに対するコードを記述することは、特定の入力に縛られない、ボタンに縛られない (クリック可能なコンポーネントにのみ依存する) という利点がある

・ただし、イベント自体はメソッドの実装を特定のシグネチャにすることが要求される

Button.Click のデリゲート

```
public delegate void RoutedEventHandler(object sender,RoutedEventArgs e);
```

WPF の目標の一つは密接に結合されたフィジカルイベントから完全にセマンティックな通知までの幅広いアクションを許容すること

■コマンド

・疎結合を許容することでコントロールを劇的に変化させるテンプレートを記述することが可能となる

Close コマンドに結び付けられたボタンを追加する

・ウィンドウを閉じるためのクロムを追加するウィンドウ用テンプレートを記述できる

```
<ControlTemplate TargetType="{x:Type Window}">
  <DocPanel>
    <StatusBar DockPanel.Dock="Buttom">
      <StatusBarItem>
        <Button Command="{x:Static ApplicationCommands.Close}">
        閉じる
        </Button>
      </StatusBarItem>
    </StatusBar>
  </DocPanel>
</ControlTemplate>
```

・その後、ウィンドウにコマンドバインディングを追加することで、任意のコンポーネントが Close コマンドを発行したときにウィンドウが閉じるようにすることが可能

```
public MyWindow() {
  InitializeComponent();

  CommandBindings.Add(
    new CommandBinding(ApplicationCommands.Close, CloseExecuted)
  );
}
void CloseExecuted(object sender, ExecuteRouteEventArgs e) {
  this.Close();
}
```

・コマンドは WPF における最も疎結合なアクションモデルを表す
・コマンドはアクションのソース (ボタンなど) とアクションのハンドラからの完全な抽象化を提供

アプリケーションを破壊せずに全く異なるコントロールを使用するようにスタイルの変更が可能となる

宣言型アクション

・コマンドと疎結合が導入されたことで、ソフトウェアが意図を宣言するモデルへと向かうことがわかる
  ・「このボタンがクリックされたら、Window.Close() を呼び出す」から「このコマンドが実行されたらウィンドウを閉じます」へと向かっている
・WPF の主要な基盤は宣言型プログラミングという考え方
・視覚要素、UI レイアウトに加え、アプリケーションロジックの大半をマークアップで指定することが可能

宣言型ロジックは、宣言形式の周囲で、ユーザーにツールを提供することによりエクスペリエンスを向上させたり、より高度なサービスをシステムで提供したりできるという点できわめて有効

アクションの処理方法によって、宣言型プログラミングのサポートレベルはことなる

アクション	内容
イベント	ターゲット関数をマークアップで宣言できるが、ハンドラはコードで実装する必要がある
コマンド	宣言型の使用のために特別に設計されており、アクションのソースと利用者の間に最良の抽象化を提供
トリガ	最もリッチな宣言型サポートを提供するが、拡張性に欠けるため複雑なタスクで使用するのは困難

イベント

・イベントは他の .NET クラスライブラリと全く同じように機能
・各オブジェクトは一覧のイベントを公開し、イベントに対しデリゲートを使用してリスナを添付できる

ルーティングイベント

・WPF はルーティングイベントに関連する追加の機能セットを備える

ルーティングイベント	内容
直接イベント	単一のソースで発生する単純なイベント。標準の .NET イベントとほぼ同じだが、WPF のルーティングイベントシステムに登録される点が異なる

バブルイベント	要素ツリーのターゲットからルートに向かって移動
トンネルイベント	要素ツリーのルートからターゲットに向かって移動

・バブルイベントとトンネルイベントは裏表の関係、通常、対になっており、トンネルバージョンには、Preview という接頭語がつく

■要素の階層を作成してイベントをリッスンする

・グループボックスと複数のボタンを含むウィンドウ

```
<Window ...
  PreviewMouseRightbuttonDown="WindowPreviewRightButtonDown"
  MouseRightButtonDown="WindowRightButtonDown"
  >
  <GroupBox
    PreviewMouseRightbuttonDown="GroupBoxPreviewRightButtonDown"
    MouseRightButtonDown="GroupBoxRightButtonDown"
    >
    <StackPanel>
      <Button>ボタン 1</Button>
      <Button
         PreviewMouseRightbuttonDown="ButtonTwoPreviewRightButtonDown"
         MouseRightButtonDown="ButtonTwoRightButtonDown"
         >
      ボタン 2
      </Button>
    </StackPanel>
  </GroupBox>
</Window>
```

・イベントハンドラでイベント名を出力

```
void ButtonTwoPreviewRightButtonDown(object sender,MouseButtonEventArgs e) {
  Debug.WriteLine("ButtonTwo PreviewRightButtonDown");
}
void ButtonTwoRightButtonDown(object sender,MouseButtonEventArgs e) {
  Debug.WriteLine("ButtonTwo RightButtonDown");
}
void GroupBoxPreviewRightButtonDown(object sender,MouseButtonEventArgs e) {
  Debug.WriteLine("GroupBox PreviewRightButtonDown");
}
void GroupBoxRightButtonDown(object sender,MouseButtonEventArgs e) {
  Debug.WriteLine("GroupBox RightButtonDown");
}
void WindowPreviewRightButtonDown(object sender,MouseButtonEventArgs e) {
  Debug.WriteLine("Window PreviewRightButtonDown");
}
void WindowRightButtonDown(object sender,MouseButtonEventArgs e) {
  Debug.WriteLine("Window RightButtonDown");
}
```

イベント順序

1.Window PreviewMouseRightButtonDown
2.GroupBox PreviewMouseRightButtonDown
3.ButtonTwo PreviewMouseRightButtonDown
4.ButtonTwo MouseRightButtonDown
5.GroupBox MouseRightButtonDown
6.Window MouseRightButtonDown

・コントロールの既定の動作は常にイベントのバブルバージョンで実装する必要がある
・プレビューイベントを使用しないパターンにより、開発者がプレビューイベントを使用してロジックにフックしたり、既定の動作をキャンセルすることが可能になる
・イベントルーティングの任意のポイントで Handled プロパティを設定し、さらなるイベントハンドラが呼び出されるのを防ぐことができる

・すべての要素がクリックされるのを防ぐ

```
public Window() {
  this.PreviewMouseRightButtonDown += WindowPreviewRightButtonDown;
}
void WindowPreviewRightButtonDown(object sender, MouseButtonEventArgs e) {
  e.Handled = true;
}
```

・Handle プロパティはすべてのルーティングイベントが共有するプロパティの一つ

```
public class RoutedEventArgs : EventArgs {
  public bool Handled { get; set; }
  public object OriginalSource { get; }
  public RouteEvent RouteEvent { get; set; }
  public object Source { get; set; }
}
```

WPF はルーティングという概念をイベントに含めるように .NET イベントモデルを拡張し、それにより要素の合成を可能にする。ほかのアクション処理機構は、すべてこの基盤イベントルーティングモデル状に構築される

コマンド

・WPF のほとんどのイベントは、各コントロールの実装の詳細に結びついている
・イベントはコードの特定の部分をコントロールからの通知に結び付ける場合には便利だが、もっと抽象的に処理を行いたい場合もある

プログラムを終了する機能

■イベントでの実装例

・まず必要なのは、プログラムを終了するためのメニュー

```
<MenuItem Hader="ファイル(_F)">
  <MenuItem Header="終了(_X)" Click="ExitClicked"/>
</MenuItem>
```

・次に分離コードでイベントハンドラを実装

```
void ExitClicked(object sender, RouteEventArgs e) {
  Application.Current.Shutdown();
}
```

・この方法はうまく機能するが、アプリケーションを終了するハイパーリンクも加えてみる

```
<TextBlock>
  <Hyperlink Click="ExitClicked">終了</Hyperlink>
</TextBlock>
```

・ここでは様々なことが仮定されている

・シグネチャが、Hyperlink.Clicked と互換性があること
・実装がアプリケーションを単純に終了すること

コマンド

・これらの問題をコマンドが解決する
・コマンドを使用すると、目的のアクションを示す単一の名前を提供できる

■コマンドを使用するには

・コマンドを定義する
・コマンドの実装を定義する
・コマンドのトリガを作成する

WPF のコマンドシステムの基盤になるのは、ICommand インターフェース

```
public interface ICommand {
  event EventHandler CanExecuteChanged;
  bool CanExecute(object parameter);
  void Execute(object paramter);
}
```

・CanExecute はコマンドが利用可能かどうか
  ・使用可能の概念を共有することで同じコマンドに結び付けられている複数のコントロールが一貫した共有可能状態を持つことが可能となる
・Execute コマンドの実行をトリガする

■対応するコマンドの実装

```
protected virtual void OnClick(RoutedEventArgs e) {
  if (Command != null && Command.CanExecute(CommandParameter)) {
    Command.Execute(CommandParmeter);
  }
}
```

■新しいコマンドの定義

```
public class Exit : ICommand {
  public bool CanExecute(object parameter) {
    return true;
  }
  public void Execute(object parameter) {
    Application.Current.Shutdown();
  }
}
```

■マークアップとバインド

```
<MenuItem Hader="ファイル(_F)">
  <MenuItem Header="終了(_X)">
    <MenuItem.Command>
      <l:Exit />
    </MenuItem.Command>
  </MenuItem>
</MenuItem>
  :
<TextBlock>
  <Hyperlink>終了
    <Hyperlink.Command><l:Exit /></Hyperlink.Command>
  </Hyperlink>
</TextBlock>
```

■機能セットの発行

・コマンドは複数の場所から利用されることが一般的
・通常はコマンドインスタンスを使用して制的フィールドを作成する

```
public partial class Window1 : Window {
  public static readonly ICommand ExitCommand = new Exit();
    :
}
```

・ICommand をフィールド型とすることで、Exit の実装を完全にプライベートにできる利点
・これで Exit をプライベートクラスとしてマークし、静的フィールドにバインドするようにマークアップを変更できる

```
<MenuItem Hader="ファイル(_F)">
  <MenuItem Header="終了(_X)"
    Command="{x:Static l:Window1.ExitCommand}">
</MenuItem>
```

こうすることで、ウィンドウでコマンドを公開することで、機能セットを発行できる。

■変更

・現在 Exit はどこからでも呼び出すことができ、アプリケーションを終了する
・これを現在のウィンドウを閉じることになるように変更したいとした場合、実装を定義から分離する必要がある

分離のもっとも簡単な方法はイベントを利用する

・コマンドで新しいイベントを定義し、イベントルーティングシステムを利用してコンポーネントに通知することができる

```
class Exit : ICommand {
  public static readonly RouteEvent ExecuteEvent =
   EventManager.RegisterRouteEvent(
     "Execute",
     RoutingStrategy.Bubble,
     typeof(RoutedEventHandler),
     typeof(Exit)
   );
    :
}
```

・このイベントはバブル方式のため、イベントソースからバブルアップする

イベントを発行するには

・現在の要素を検索するように Execute を変更し、適切なイベントを発行する

```
public void Execute(object parameter) {
  RoutedEventArgs e =
    new RoutedEventArgs(Exit.ExecuteEvent, Keyboard.FocusedElement);
  Keyboard.FocusedElement.RaiseEvent(e);
}
```

ここからコマンドバインディングという概念が生まれる

■コマンドバインディング

・コマンドバインディングとは、コマンドの実装を、そのコマンドの身元から切り離す機能

Window1 の実装に戻り、コマンドの実装を追加する

```
public Window1 : Window {
 :
  public Window1() {
    InitializeComponent();

    AddHandler(Exit.ExecuteEvent, ExitExecuted);
  }
  void ExitExecuted(object sender, RoutedEventArgs e) {
    this.Close();
  }
}
```

このケースでは、Exit コマンドは要素ツリーで Execute イベントを発生させ、Window がそのイベントをリッスンして対応できるようにする

1. メニュー項目がクリックされる
2.MenuItem がコマンドで Execute を呼び出す
3.Exit の実装がフォーカスが設定されているオブジェクト (この場合 MenuItem) で、Exit.Execute イベントを発生させる
4. そのイベントが要素ツリーをバブルアップ
5. ウィンドウが Exit.Execute(ウィンドウを閉じる) を実行する

■ RoutedCommand

・この方法で、入力バインディング、パラメータ、およびその他の機能をサポートするように基盤の ICommand モデルを拡張することも可能

ただしフレームワークには、これらのほとんどを処理する組み込みのユーティリティクラス、RoutedCommand がすでに含まれている

・ルーティングコマンドは、コマンドの実装をコマンドの身元から完全に分離する
・コマンドの定義は静的つまり、コマンドの定義は、コマンドの身元を提供する一種のトークンに過ぎない

```
public partial class Window1 : Window {
  public static readonly ICommand ExitCommand =
    new RoutedCommand("Exit", typeof(Window1));
  :
}
```

■処理コードをバインド

・コマンドが実行されたときに何を行うかは、処理コードをコマンドにバインドする必要がある
・ルーティングコマンドはイベントと同じようにバブルアップする
  ・ルートウィンドウにコマンドバインディングを追加し、すべてのコマンドを確認することができる
  ・コマンドバインディングはリッスンするコマンドと、トリガされたときに実行するコードを受け取る

```
public Window1() {
  InitializeComponent();
```

```
  CommandBindings.Add(new CommandBinding(ExitCommand, ExitExecuted));
}
void ExitExecuted(object sender, ExecutedRoutedEventArgs e) {
  this.Close();
}
```

・コマンドバインディングを使用すると、コマンドを使用可能にするか否かを判断するためのロジックを使用できる

■ InputBindings プロパティ

・InputBindings プロパティを使用して入力ジェスチャをマップできる

```
<Window x:Class="..."
  :>
  <Window.InputBindings>
    <KeyBinding Key="A" Modifires="Contorol"
      Command="{x:Static l:Window1.ExitCommand}" />
  </Window.InputBindings>
</Window>
```

■セキュアコマンド

・切り取り、コピー、貼り付けなどの一部コマンドはセキュリティに影響を与える
・システムがこれらの操作をユーザーによって要求された場合のみに実行することを保証するために、RoutedCommand は、ユーザーによって開始されたかどうかを追跡可能

一般にアプリケーションロジックはイベントハンドラではなく、コマンドの観点で実装することが推奨される。イベントハンドラが必要なケースの多くはトリガーを使用したほうがうまく処理できる。

コマンドとデータバインディング

・コマンドを使用する場合のもっとも強力な機能はデータバインディングの統合
・Command と CommandParameter はいずれも要素のプロパティのため、データをバインドすることができる
・コマンドを使用することでデータ駆動型のロジックも可能となる

■ドライブ内のファイルをリストするアプリケーション例

```
<Window x:Class="..."
  :>
  <ListBox Margin="2" Name="_files">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding Path=Name}" />
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
</Window>
```

・分離コードで、ItemSource プロパティにファイルのリストを代入

```
public partial class DataAndCommands : Window {
  public DataAndCommands() {
    InitializeComponent();
    FileInfo[] fileList = new DirectoryInfo("c:\\").GetFiles("*.*");
    _files.ItemSource = fileList;
  }
}
```