勉強会 リアクティブプログラミングについて その2

2020-12-25

株式会社キングプリンターズ システム課

この資料で説明すること

- 前回 Java の Reactor Core を紹介しましたが、消化不良気味でしたので今回は別の観点から**リアクティブプログラミング**について調べたことを発表します。

そもそもリアクティブプログラミングとは？

- リアクティブプログラミングは、CPUのマルチコア化やクラウドの活用が進む中で、 **非同期通信に基づいたアプリケーションの設計に適している**ため、 これからのソフトウェア開発における重要な技術として注目されています。

リアクティブプログラミングはこれまでの手法と何が違うのか？

以下の疑似コードを見てください。

```
a = 1
b = 2 * a
a = 2
print b
```

これまでの命令型プログラミングの観点では以下の出力を得ます

```
a = 1
b = 2 * a
a = 2
print b
```

出力結果: 2

対して、リアクティブプログラミングの観点では以下の出力を得ます

```
a = 1
b = 2 * a
a = 2
print b
```

出力結果: 4

リアクティブプログラミングの観点

```
a = 1
b = 2 * a
a = 2
print b
```

出力結果: 4

- リアクティブプログラミングの観点では、2行目の「b = 2 * a」というコードは「b を a の2倍として定義する」という意味で解釈されます。
- つまり、a と b の関係性を定義した後は、「 a への代入」というイベントへのリアクションとして 「 b の再計算」を常にバックグラウンドで行います。
- このようなパラダイムのことを**リアクティブプログラミング**と呼びます。

リアクティブプログラミングのメリット

これまでの命令型プログラミングでは、特に非同期の並行処理において様々な問題がありました。
例えば以下のような問題です。

- コードが複雑化する
- 命令の実行順序を管理することができない
- 幾重にもネストされたコールバック(いわゆるコールバック地獄)

リアクティブプログラミングを活用すると、**今までとは異なるプログラミングパラダイム**で上記の問題を解決するコードを記述することが可能になります。

とは言われるものの...🤔

一般的には先ほどのように**非同期処理と絡めて**リアクティブプログラミングの利点が紹介されるケースが多いですが、正直自分は全然実感がわきませんでした。
そこで、ここでは**自分が納得いく方法で**説明を試みたいと思います。

今までとは異なるプログラミングパラダイムって？🤔

これは個人的な意見で他のサイトの受け売りではないのですが
Rx系のライブラリを利用すると

- for, foreach などの**ループ処理**を書かないようになる
- if, else if などの**分岐処理**を書かないようになる

というコードに落ち着くかと思います。
何故そうなるかを、これから**自作のRx実装**😎を開発しながら説明していきたいと思います。

まずリアクティブプログラミングを実現する方法は？

一般的にリアクティブプログラミングの為に使用されるライブラリは、その根幹に**オブザーバーパターン**という考え方を持っています。

GoF のオブザーバーパターン (Observer pattern)

- オブザーバーパターンは 1995年に GoF によって提案された23のデザインパターンの一つであり、プログラムのインタフェースとしては古典的なものです。
- **Subject(監視対象)** と **Observer(観察者)** からなるパラダイムで、利用シーンとしては、GUIの設計においてデータベースの変更を検知して表示を自動更新したい場合などが代表的です。

オブザーバーパターンについては

- ここでは詳しく解説しません、別の一般的な資料を参照することをお勧めします。
- 理由は Rx では表向きオブザーバーパターンのインターフェイスを利用しますが、どうやら内部的にはそれに準拠しない実装で実現していると推測されるからです。(※その部分は後述します)

※個人的な感想なので判断はお任せ致します

まず C# でシンプルなオブザーバーパターンを実装してみましょう

幸いなことに .NET Framework では以下のインターフェイスが定義されています

- IObserver<T>インターフェイス：
  - 「値を受け取る側」が実装するインターフェイス
    - void OnNext(T value);
    - void OnError(Exception e);
    - void OnCompleted();
- IObservable<T>インターフェイス：
  - 「値を発行する側」が実装するインターフェイス
    - IDisposable Subscribe(IObserver<T> observer);

最初にシンプルな IObserver 実装クラスを作成します

```
public class HogeObserver : IObserver<string> {
    public void OnCompleted() {
        throw new NotImplementedException(); // ※実装省略
    }
    public void OnError(Exception error) {
        throw new NotImplementedException(); // ※実装省略
    }
    // 変更通知を受け取るメソッド
    public void OnNext(string value) {
        Console.WriteLine("OnNext: " + value); // ここではコンソールに表示します
    }
}
```

次にシンプルな IObservable 実装クラスを作成します

```
public class HogeObservable : IObservable<string> {
    // observer のリストを持ちます
    List<IObserver<string>> observers = new List<IObserver<string>>();
    // リストに observer を追加します
    public IDisposable Subscribe(IObserver<string> observer) {
        observers.Add(observer);
        return null; // ※ この実装では IDisposable には対応しません
    }
    // 登録された observer に変更内容を通知する
    public void Notify(string value) {
        // ここではリストをループ処理しています
        foreach (var observer in observers) {
            observer.OnNext(value);
        }
    }
}
```

実行してみます

```
class Program {
    static void Main(string[] args) {
        // Observable と Observer を使用
        var observable = new HogeObservable();
        observable.Subscribe(new HogeObserver()); // 登録するだけです
        observable.Notify("Hello"); // ※ Notify() は IObservable で定義されていません
        observable.Notify("World"); // ※ Notify() は IObservable で定義されていません
    }
}
```

```
# 結果
OnNext: Hello
OnNext: World
```

- 問題点
  - **Observable** と **Observer** の結びつきが強い ⇒ 密結合
  - **実行するメソッド** がインターフェイスで定義されてない！

どうするか？ ⇒ 仲介役を作成します

さらに System.Reactive(Rx) で以下のインターフェイスが定義されています

- ISubject<T>インターフェイス：
  - 「値を受け取る側」が実装するインターフェイス
    - void OnNext(T value);
    - void OnError(Exception e);
    - void OnCompleted();
  - 「値を発行する側」が実装するインターフェイス
    - IDisposable Subscribe(IObserver<T> observer);

どちらの操作も持つ **ISubject** インタフェースを使います

それではシンプルな ISubject 実装クラスを作成してみます

```
class HogeSubject : ISubject<string> {
    List<IObserver<string>> observers = new List<IObserver<string>>();
    public void OnCompleted() {
        throw new NotImplementedException(); // ※実装省略
    }
    public void OnError(Exception error) {
        throw new NotImplementedException(); // ※実装省略
    }
    // ここが呼ばれて初めて処理が走ります
    public void OnNext(string value) {
        foreach (var observer in this.observers) {
            observer.OnNext(value);
        }
    }
    // ここでは observer が登録されるだけ
    public IDisposable Subscribe(IObserver<string> observer) {
        this.observers.Add(observer);
        return null; // ※ この実装では IDisposable には対応しません
    }
}
```

実行してみます

```
class Program {
    static void Main(string[] args) {
        // Subject と Observer を使用
        var subject = new HogeSubject();
        subject.Subscribe(new HogeObserver());
        subject.OnNext("Hello");
        subject.OnNext("World");
    }
}
```

```
# 結果
OnNext: Hello
OnNext: World
```

- 現時点で
  - OnNext メソッドで処理が実行されるようになりました

どういうこと？🤔

- つまり Subject の OnNext() を**呼ぶと** Observer の OnNext() が**呼ばれ**ます

```
public class HogeObserver : IObserver<string> {
    // ※省略
    public void OnNext(string value) {
        Console.WriteLine("OnNext: " + value); // ここが呼ばれます
    }
}
```

これは Subject から Observer にイベント通知されると表現されます。イベント通知という言葉で難しく説明されますが、単純に Subject が自分で Observer のリストを持っていて、それらをループ処理して Observer のメソッドを呼んでいます。

ここまでのまとめ

- Subject が Observer を呼び出す：Pull型で情報**を通知し**ます
- Observer は Subject から呼ばれる：Push型で情報**が通知され**ます

現時点では、Subject の OnNext() メソッドを呼べば Observer の処理が走るところまで実装出来ました😋

ここから一気に Rx の実装に近づけます！😤

- 問題点
  - IObserver インタフェースを実装するのめんどくさくないですか？😫
- 解決策
  - それ**ラムダ式**で出来ます！😋

それでは C# のラムダ式を使用してみます

```
class Program {
    static void Main(string[] args) {
        // Subject とラムダ式を使用
        var subject = new HogeSubject();
        subject.Subscribe(x => Console.WriteLine("OnNext: " + x)); // ラムダ式
        subject.OnNext("Hello");
        subject.OnNext("World");
    }
}
```

```
# 結果
OnNext: Hello
OnNext: World
```

ISubject の実装だけでここまで出来ました

再びどういうこと？

- Rx のライブラリで C# の拡張メソッドを使用した糖衣構文が用意されています

```
public static class ObservableExtensions {
    // ※省略
    public static IDisposable Subscribe<T>(this IObservable<T> source, Action<T> onNext);
    // ※省略
}
```

- **Action<T> デリゲートにラムダ式を渡す**ことで実現しています
- 実際には内部的に IObserver の実装インスタンスを作成して onNext() で Action デリゲートが実行されます

※ここではラムダ式は詳しく説明しません

ここまでのまとめ

- Rx系でよく見る subject.Subscribe(x => x への処理)) は、C# では内部的に IObserver の実装インスタンスが作成されています。
- subject.Subscribe() に渡すデリゲートは IObserver の文脈で onNext() として呼ばれます。
- Subscribe() にラムダ式を記述しただけでは処理は走りません、別途 onNext() されて初めて処理が走ります。

ちょっと待って😟

- まだ全然 Rx っぽくないじゃないか？😕

Rx(リアクティブエクステンション)っぽいコードとは？🤔

- 一般的に以下のような操作が行われるコードだと思います

```
// 処理の定義
var subject = new HogeSubject();
subject.Where(x => x.Contains("H")) // "H"が含まれていたら
.Select(x => x + "ですねん") // "ですねん" を追加する
// Subscribe(※これまでの説明により、ここでは登録されるだけ)
.Subscribe(x => Console.WriteLine("OnNext: " + x));
// 処理の実行
subject.OnNext("Hello");
subject.OnNext("World");
```

```
# 結果
OnNext: Helloですねん
```

とてもシンプルな実装ですが

- Where() : フィルタリング系オペレーター
- Select() : メッセージ変換系オペレーター

上記の動作を実現するコードを書いてみます😤

- ついでに Subscribe() の独自実装もしてみます

まずデリゲートを受ける Observer を実装します

```
public class AnonymousObserver<T> : IObserver<T> {
    Action<T> onNext = null;
    public AnonymousObserver(Action<T> onNext) { // 引数がデリゲートである
        this.onNext = onNext;
    }
    public void OnCompleted() {
        throw new NotImplementedException(); // ※実装省略
    }
    public void OnError(Exception error) {
        throw new NotImplementedException(); // ※実装省略
    }
    public void OnNext(T value) {
        this.onNext(value); // デリゲートが実行されます
    }
}
```

そしてデリゲートを登録できる Subscribe() を実装します

※上記 AnonymousObserver クラスを使用して独自実装します

```
public static IDisposable Subscribe<T>(
    this IObservable<T> self, Action<T> onNext) {
    var disposable = self.Subscribe(
        new AnonymousObserver<T>(onNext) // IObserver の実装を生成します
    );
    return disposable;
}
```

実行してみます

```
class Program {
    static void Main(string[] args) {
        // Subject とラムダ式を使用
        var subject = new HogeSubject();
        subject.Subscribe(x => Console.WriteLine("OnNext: " + x)); // ラムダ式
        subject.OnNext("Hello");
        subject.OnNext("World");
    }
}
```

```
# 結果
OnNext: Hello
OnNext: World
```

Microsoft謹製の Rx ライブラリと同じようにラムダ式で記述できました

次にデリゲートを受ける Observable を実装します

```
public class AnonymousObservable<T> : IObservable<T> {
    Func<IObserver<T>, IDisposable> subscribe = null;
    public AnonymousObservable(Func<IObserver<T>, IDisposable> subscribe) {
        this.subscribe = subscribe;
    }
    public IDisposable Subscribe(IObserver<T> observer) {
        // デリゲートが実行されます ※中にネストされた observable を持ちます
        var disposable = this.subscribe(observer);
        return disposable;
    }
}
```

ではデリゲートを登録できる Where() を実装します

```
public static IObservable<T> Where<T>(
    this IObservable<T> self, Func<T, bool> predicate) {
    var observable = new AnonymousObservable<T>(observer => {
        var disposable = self.Subscribe(new AnonymousObserver<T>(value => {
            // ここが OnNext される内容
            // 条件に一致したときだけ observer に値を流します
            if (predicate(value)) {
                observer.OnNext(value);
            }
        }));
        return disposable;
    });
    return observable; // self ではなく別の observable を返す
}
```

実行してみます

```
// Where を使用
var subject = new HogeSubject();
subject.Where(x => x.Contains("H")) // "H" が含まれる場合のみ
.Subscribe(x => Console.WriteLine("OnNext: " + x));
subject.OnNext("Hello");
subject.OnNext("World");
```

```
# 結果
OnNext: Hello
```

Where() オペレーターの動作を実現出来ました

はああ？😯😯😯

- 一体全体何が起こったの？😟😟

C# の動作環境でステップ実行したらメカニズムが分かります

- デリゲート経由の Observer がネストされたような構造になってました
- すみません余りにもデリゲートの絡みが複雑すぎて言葉に出来ません😫
- ただ、subject.OnNext() 後に Observer の処理が一斉に走ります

※上記コードは Unity の UniRx の作者のコードを参考によりシンプルに実装しました

- ポイントは**Where()** や **Subscribe()** では処理は走らない
- **subject.OnNext()** されて初めて評価、処理されるという点です

※ステップ実行したら面白いです魔法のようです🧙

さらにデリゲートを登録できる Select() を実装します

```
public static IObservable<TResult> Select<T, TResult>(
    this IObservable<T> self, Func<T, TResult> selector) {
    var observable = new AnonymousObservable<TResult>(observer => {
        var disposable = self.Subscribe(new AnonymousObserver<T>(value => {
            // ここが OnNext される内容
            // 渡ってきた値を加工して observer にその値を流します
            TResult result = selector(value);
            observer.OnNext(result);
        }));
        return disposable;
    });
    return observable; // self ではなく別の observable を返す
}
```

実行してみます

```
// Select を使用
var subject = new HogeSubject();
subject.Select(x => x + "ですねん") // "ですねん" を追加する
.Subscribe(x => Console.WriteLine("OnNext: " + x));
subject.OnNext("Hello");
subject.OnNext("World");
```

```
# 結果
OnNext: Helloですねん
OnNext: Worldですねん
```

Select() オペレーターの動作を実現出来ました

さらに Where と Select を組み合わせて実行してみます

```
// Where と Select を使用
var subject = new HogeSubject();
subject.Where(x => x.Contains("H")) // "H" が含まれる場合のみ
.Select(x => x + "ですねん") // "ですねん" を追加する
.Subscribe(x => Console.WriteLine("OnNext: " + x));
subject.OnNext("Hello");
subject.OnNext("World");
```

```
# 結果
OnNext: Helloですねん
```

Where()、Select() オペレーターの動作を実現出来ました

簡易 Rx 実装の完成です🎉🎉

もちろんこの実装はいろいろ足りません😭😭

- ですが、Rx(リアクティブエクステンション)の実装が内部で何をしているのかが、何となくわかって頂けたのではないでしょうか？
- 今回は C# の実装を参考にしたので他の言語では異なる実装になると思われますが、エッセンスは共通だと思います。

さて、序盤で書いた今までとは異なるプログラミングパラダイムって？🤔

- for, foreach などのループ処理を書かない
  - ⇒ そもそも Subject 内の実装でラップされる
- if, else if などの分岐処理を書かない
  - ⇒ Where などで代替えされる

結果、上記のようになったと思いませんか？😚

最後に命令型プログラミングパラダイムで実装してみます

```
// foreach や if で実装
var list = new List<string>();
list.Add("Hello");
list.Add("World");
foreach (var item in list) {
    if (item.Contains("H")) { // "H" が含まれる場合のみ
        var result = item + "ですねん"; // "ですねん" を追加する
        Console.WriteLine("result: " + result);
    }
}
```

```
# 結果
result: Helloですねん
```

全く違いますね、今感じたことがリアクティブプログラミングとは**パラダイムが異なる**という感覚だと思います。

📖最後のまとめ

- 今回の説明ではRx(リアクティブエクステンション)≠リアクティブプログラミングのエッセンスを紹介したかったので **Subject.OnNext()** を自分で呼びました。
- しかし、例えば実際の Unity ⇒ UniRx ゲームプログラミングでは自分で **OnNext()** を呼び出すケースはほぼありません。
- **OnNext()** は Rx ライブラリが提供する **Observableストリーム**を管理するライブラリ側で良きタイミングで呼び出されています。
- ゲームプログラマはライブラリから **呼び出して** もらう **ラムダ式** の部分だけコーデイングします、今回の実装例がそのエッセンスです。
- この **OnNext()っていったい誰が呼び出してるんだろう？** という点が Rx系で一番混乱する部分だと思いました。

Visual Studio で実行する場合は🛠

- System.Reactive.Linq を NuGet でインストールします
  ※ .NET Core では動きません

ご清聴ありがとうございました🙇