イベントの自動実装

# 概要

C#の機能の多くは、元となるC#コードに対して、コンパイラーが生成するILコード(あるいはさらにそのコンパイル結果となるCPU命令)が非常に多いものがいくつかあります(要は、複雑な処理をコンパイラーが肩代わりしてくれてる)。代表的なものでは、C# 2.0で導入された[イテレーター](http://ufcpp.net/study/csharp/sp2_iterator.html)、3.0の[ラムダ式](http://ufcpp.net/study/csharp/sp3_lambda.html)、5.0の[非同期メソッド](http://ufcpp.net/study/csharp/sp5_async.html)などです。

そして、C# 1.0の頃からあるものの中ではダントツで複雑なのが[イベント](http://ufcpp.net/study/csharp/sp_event.html#event-keyword)構文(特に、add/removeの自動実装)です。ここでは、このイベント構文がやっていることについて説明していきます。

## ポイント

* イベント構文 ＝ イベントを起こす側と受け取る側の分離
* イベント ハンドラーの登録口を自動生成してる
* イベントの自動実装ではスレッド安全性も保障

# イベント構文

おさらいしておくと、イベント構文(eventキーワード)が提供するのは以下のような保証です。

* デリゲート呼び出しはクラス内部からのみ可能。
* 外部からはデリゲートの追加/削除のみが可能。
  + 別の場所で、別の受け手がデリゲートを追加できる
  + 追加した本人だけが削除できる

要するに、イベントを起こす側と受け取る側の分離です。

イベントを起こす側では、以下のように、イベント構文を使って、イベント ハンドラーの登録口を作ります。

// イベントを起こす側  
class EventSource  
{  
    // イベント ハンドラーの登録口  
    public event EventHandler<XArgs> X;  
   
    // イベントが起こったことを通知したいときに呼ぶ  
    private void OnX(XArgs args)  
    {  
        var x = X;  
        if (x != null)  
            x(this, args);  
    }  
}

イベントを受け取る側では、以下のように、そのイベントに対して += でハンドラー登録、-=で登録解除を行います。

// イベントを受け取る側  
class EventConsumer : IDisposable  
{  
    private EventSource \_source;  
   
    public EventConsumer(EventSource source)  
    {  
        \_source = source;  
   
        // イベント ハンドラーの登録  
        source.X += HandelX;  
   
        //source.X(source, new XArgs()); ← エラーになる。外からは呼び出せない  
    }  
   
    public void Dispose()  
    {  
        // Dispose 時に登録解除(自分が登録したものだけを解除)  
        \_source.X -= HandelX;  
    }  
   
    private void HandelX(object sender, XArgs e)  
    {  
        // イベントが起きた時にやりたい処理  
    }  
}

勝手にイベントを起こすことはできず(source.Xを普通のデリゲートのように呼び出すことは外からはできない)、他者が登録したイベント ハンドラーを勝手に外すこともできません(+=で渡したデリゲートを、-=に対しても渡さないといけない)。

# 自動実装イベントがやっていること

イベント構文がやっていることを理解するために、まず、自動実装の部分を自前で実装しなおしてみましょう(add/removeアクセサーを明示的に書く)。

(注意: 本当は、ここで示すようなコードに加えて、マルチスレッドで使っても大丈夫なように、同期処理(スレッド安全性の保証)を加える必要があります。この点については後述。)

先ほどのEventSourceクラスは以下のように書き変わります(EventConsumer側は何も変わりません)。

class EventSource  
{  
    // イベント ハンドラーの登録口(自前実装)  
    public event EventHandler<XArgs> X  
    {  
        add { \_x += value; }  
        remove { \_x -= value; }  
    }  
   
    // 登録されたイベント ハンドラーを保存しておくためのフィールド  
    // 実際には、コンパイラーが生成するフィールドの名前は X (イベント名と同じ)  
    // C# では認めてないけども、IL 的にはイベントと同名のフィールドを作れる  
    private EventHandler<XArgs> \_x;  
   
    private void OnX(XArgs args)  
    {  
        var x = \_x; // 呼び出してたのは実はフィールドの方  
        if (x != null)  
            x(this, args);  
    }  
}

デリゲート型のprivateなフィールドが生成されて、そのデリゲートに対する+=と-=が作られます。

EventConsumer側のsource.Xに対する+=と-=は、実際にはこのadd/removeアクセサーが呼ばれています。

## 余談: デリゲートの+=、-=

ちなみに、デリゲートの+=、-=は、以下のようなコードと同じ意味になります。

public event EventHandler<XArgs> X  
{  
    add  
    {  
        \_x = (EventHandler<XArgs>)Delegate.Combine(\_x, value);  
    }  
    remove  
    {  
        \_x = (EventHandler<XArgs>)Delegate.Remove(\_x, value);  
    }  
}

Delegete(System名前空間)クラスのCombine/Removeメソッドの呼び出しに展開されます。あと、2点、補足しておきましょう。

* Delegate.Remove(x, x) は null を返します
  + x -= value; の結果、xがnullになることがあります
* Delegate.Combine(null, value)はxをそのまま返します
  + xがnullの状態で x += value; としても問題は起こりません。

# イベント構文がやっていること

さらに、イベント構文やデリゲートの+=、-=がやっていることまで展開するとどうなるか見てみましょう。

(注意: これも、スレッド安全性の保証はさぼっています。後述。)

まず、EventoSource側は以下のようになります。

class EventSource  
{  
    // イベント ハンドラーの登録口  
    // Add と Remove の2つのメソッドを用意  
    public void AddXHandler(EventHandler<XArgs> value)  
    {  
        \_x.Add(value);  
    }  
    public void RemoveXHandler(EventHandler<XArgs> value)  
    {  
        \_x.Remove(value);  
    }  
   
    // 登録されたイベント ハンドラーを保存しておくためのフィールド  
    // C# のデリゲートは、それ自身がリストみたいなもの(マルチキャスト デリゲート)  
    private List<EventHandler<XArgs>> \_x = new List<EventHandler<XArgs>>();  
   
    private void OnX(XArgs args)  
    {  
        // マルチキャスト呼び出し  
        foreach (var x in \_x)  
        {  
            x(this, args);  
        }  
    }  
}

元のeventにあたる部分は、AddXHandlerとRemoveXHandlerという2つのメソッドに分かれます。ハンドラー保存用のフィールドも、(+=、-=を使わないなら)リストに置き換わります。

EventConsumer側も単純に、+=、-=を呼んでいた場所がAddXHandler/RemoveXHandlerに置き換わります。

class EventConsumer : IDisposable  
{  
    private EventSource \_source;  
   
    public EventConsumer(EventSource source)  
    {  
        \_source = source;  
   
        // イベント ハンドラーの登録。 += を AddXHandler に  
        source.AddXHandler(HandelX);  
    }  
   
    public void Dispose()  
    {  
        // Dispose 時に登録解除。 -= を RemoveXHandler に  
        \_source.RemoveXHandler(HandelX);  
    }  
   
    private void HandelX(object sender, XArgs e)  
    {  
        // イベントが起きた時にやりたい処理  
    }  
}

# 実際のイベント構文の展開結果

一応注釈はつけていましたが、実際には、イベント(ハンドラーの登録口)は、スレッド安全性を保証する必要があります(イベント駆動なプログラムは、イベントを起こす側と受け取る側を異なるスレッドで動かすことが結構あります)。

C#の言語仕様的には、イベントの自動実装はスレッド安全であるということだけ明記されています。実際の実装方法はコンパイラーのバージョンなどによって変わっていますが、現状の(C# 4.0以降の)マイクロソフト製コンパイラーは、以下のような実装になっています。

public event EventHandler<XArgs> X  
{  
    add  
    {  
        EventHandler<XArgs> e1 = \_x, e2;  
        do  
        {  
            e2 = e1;  
            var value2 = (EventHandler<XArgs>)Delegate.Combine(e2, value);  
            e1 = Interlocked.CompareExchange(ref \_x, value2, e2);  
        }  
        while (e1 != e2);  
    }  
    remove  
    {  
        EventHandler<XArgs> e1 = \_x, e2;  
        do  
        {  
            e2 = e1;  
            var value2 = (EventHandler<XArgs>)Delegate.Remove(e2, value);  
            e1 = Interlocked.CompareExchange(ref \_x, value2, e2);  
        }  
        while (e1 != e2);  
    }  
}  
private EventHandler<XArgs> \_x;

このコードでどうしてスレッド安全になるかについては、別ページで改めて説明します。