

「An introduction to AS3 Signals」

本ドキュメントは DevelopRIA の[An introduction to AS3 Signals]ページで公開されている記事をヒム・カンパニー 永井勝則が自主的に記述したものです。

<http://www.himco.jp/>

[knagai@himco.jp](mailto:knagai@himco.jp)

(2011/2)

本記事の原文は、

<http://www.developria.com/2010/10/an-introduction-to-as3-signals.html>

で読むことができます。

本記事では [Robert Penner](#) による AS3 Signal ライブラリを紹介し、その使用方法を見ていきます。AS3 Signal ライブラリを利用すると、オブジェクト間のやりとりが高速になり、しかも通常の ActionScript イベントを使用するときを書くコード量よりもかなり少なくて済みます。

AS3 Signal を利用するにはまず GitHub サイトから [AS3 Signal SWC ファイル](#) をダウンロードする必要があります。お好みの IDE (わたしは FDT です) でプロジェクトを作成し、ダウンロードした SWC ファイルとリンクさせます。

訳者注:

AS3 Signal ではオンラインドキュメントが公開されていないようなので、ライブラリのソースコードから作成する必要があるかもしれません。これは Grant Skinner 謹製の [ASDocr](#) で簡単に作成できます。

AS3 Signal とは？

”Signal は Qt での C# のイベントとシグナル/スロットに着想を得た、AS3 イベントへの新しいアプローチです” (Robert Penner)。

Signal に関する Robert Penner のスローガンは”Event の範囲を超えた発想”です。このスローガンは、Signal が軽量で強い型指定のされた AS3 メッセージツールとして記述されている点に表れています。

さらに詳しく述べると、Signal は AS3 イベントをほかの要素に置き換えるフレームワークで、1つの API にイベントの考えと C# のシグナルの考えを併合した機能性を有しています。ネイティブのイベントよりも全体として高速に動作し、コードも少なくて済みます。またネイティブのイベントを利用することも可能で、4倍速く動作します。

Signal とは？

Signal は簡単にいうと、1つのイベントに特化した、リスナーの配列を持ったミニディスパッチャーです。

Signal の基本概念は、ネイティブのイベントのようにストリングベースのチャンネルを使用せず、イベント/シグナルがクラス内で具体的なメンバーとなる、ということです。これは、オブジェクト間のやりとりがもっと制御できるようになるということです。Signal ではもうストリング定数は必要ないのです。

経験を積んだ ActionScript 開発者の方なら、プロジェクトでカスタムイベントサブクラスを作成する大変さをご存じでしょう。イベントを送出するクラスでは定数を定義し、イベントを定義して、リスナーを追加、削除し、つねに EventDispatcher を拡張します。しかし Signal を知ると、みなさんの開発者生活はあっという間

に変貌します。

## 基本 Signal

では基本 Signal から見ていきましょう。みなさんは早晚カスタム AS3 イベントの置き換えに使用することになります。Signal は1つのオブジェクトで、基本 Signal はイベントを送出するクラスの public プロパティとして定義します。素晴らしいことは、イベントを送出するクラスは EventDispatcher を拡張する必要がないということです。次の AlarmClock クラスはそのシンタックス(書き方)の例です。

```
package insideria.basicSignal
{
    import org.osflash.signals.Signal;

    public class AlarmClock
    {
        public var alarm:Signal;

        public function AlarmClock()
        {
            alarm = new Signal();
        }

        public function ring():void
        {
            alarm.dispatch();
        }
    }
}
```

この AlarmClock クラスでは、Signal の変数 alarm の保持に public を使っています。コンストラクタで Signal をインスタンス化し、public メソッドの ring() が呼び出されるときに Signal を送ります。ここまでは簡単です。次はこのクラスの使用方法を見ていきましょう。これも実に簡単です。

```
package insideria.basicSignal
{
```

```
import flash.display.Sprite;

public class WakeUp extends Sprite
{
    private var alarmClock:AlarmClock;

    public function WakeUp()
    {
        alarmClock = new AlarmClock();
        alarmClock.alarm.add(onRing);
        alarmClock.ring();
    }

    private function onRing():void
    {
        trace("Wake up!");
    }
}
}
```

AlarmClock クラスを利用するには何よりもまず、それをインスタンス化します (alarmClock)。そしてインスタンスの public プロパティである alarm 上で (これは Signal インスタンス) add() メソッドを使って、onRing という名前のリスナーを追加します。最後に実際に AlarmClock を鳴らせる ring() メソッドを呼び出します。アプリケーションを起動すると、コンソールに "Wake up!" が表示されます。

訳者注:  
add() はシグナルのリスナーを追加します。リスナーは、シグナルが送出する値クラスに一致する引数を持った関数です。(Signal コンストラクタで) 値クラスを指定しない場合でも、dispatch() は引数なしで呼び出せます。dispatch() はオブジェクトをリスナーに送出します。

### 引数の配信

もちろんこれだけではありません。みなさんは Signal にどうやって引数を渡すのだらうと思っておられるでしょう。これはいたってシンプルです。前の AlarmClock クラスを変更してみましょう。

```

package insideria.basicSignalArguments
{
    import org.osflash.signals.Signal;

    public class AlarmClock
    {
        public var alarm:Signal;

        public function AlarmClock()
        {
            alarm = new Signal(String);
        }

        public function ring():void
        {
            alarm.dispatch("9 AM");
        }
    }
}

```

Signal を通して強い型指定がなされた引数を配信するには、Signal のコンストラクタでその型を定義し、dispatch()メソッドで希望する値を送出するだけです（型指定しない引数を配信することもできます）。引数は同じ型でも別の型でも、コンマで区切ることで複数渡すことができます。

```

alarm = new Signal(String, Number, int, uint, Boolean);
alarm.dispatch("9 AM", 45.4, -10, 5, true);

```

訳者注: Signal コンストラクタにはクラス参照がいくつでも渡せます。

```
public function Signal(... valueClasses)
```

... valueClasses は dispatch()で型チェックできるクラス参照です。たとえば new Signal(String, uint)では signal.dispatch("the Answer", 42)が可能ですが、signal.dispatch(true, 42.5)や signal.dispatch()は行えません。

ではこの時刻を出力するメインアプリケーションファイルを見てみましょう。

```


```

```

package insideria.basicSignalArguments
{
    import flash.display.Sprite;

    public class WakeUp extends Sprite
    {
        private var alarmClock:AlarmClock;

        public function WakeUp()
        {
            alarmClock = new AlarmClock();
            alarmClock.alarm.add(onRing);
            alarmClock.ring();
        }

        private function onRing(time:String):void
        {
            trace("Wake up! It's already " + time);
        }
    }
}

```

Signal のリスナーにはパラメータを1つ取らせる必要があります。そうしないとコンパイラが `ArgumentError` (引数エラー) を発生させます。パラメータ名はここでは `time` にしています。このパラメータも型指定する必要があります。アプリケーションを実行すると、今度はコンソールに "Wake up! It's already 9 AM" と表示されます。

### Signal の追加と削除

実際のプログラミングでは、イベントを1度だけ発射させそのあとすぐそれを廃棄したい場合がよくあります。Signal ではこのとき、`addEventListener(なにがし)`、`removeEventListener(なにがし)` といったネイティブのイベント処理のお決まりのコードを書かずに、実に簡単に同じことが行えます。

ではこれも前の `WakeUp` クラスで行ってみましょう。Signal を1度だけキャッチするには、`add()` を `addOnce()` メソッドに置き換えるだけです。

```
alarmClock.alarm.addOnce(onRing);
```

これによりリスナーは1回だけ実行され、その後自動的に削除されます。add()メソッドの使用時にリスナーを削除したい場合には、remove(リスナー)メソッドが使用できます。特定の Signal に複数のリスナーを登録した場合には、その Signal 上で removeAll()メソッドが使用できます。これは次のように記述します。

```
alarmClock.alarm.remove(onRing);  
alarmClock.alarm.removeAll();
```

訳者注:

addOnce()は、このシグナルが1回だけ呼び出すリスナーを追加します。シグナルは1度呼び出されると、すべてのリスナーへの送付完了後、自動的に削除されます。

ここまで学んだことは、今後何度も使用することになる便利で小さな機能です。しかしみなさんはもう、Signal でもっと多くの情報を配信するにはどうするのだろうとっておられるでしょう。たとえばネイティブのイベントはターゲットに関する事柄を教えてください。無論 Signal も同様です。

#### DeluxeSignal と GenericEvent

DeluxeSignal は、ネイティブの ActionScript イベントから移行するわれわれが習熟すべき次のステップです。DeluxeSignal を使用するとターゲットにアクセスできますが、とりわけ重要なのが Signal そのものにもアクセスできるようになるということです。われわれは今後その両方が操作できるようになるのです。もちろん DeluxeSignal にも基本的な Signal の全機能が備わっています。

訳者注:

DeluxeSignal のコンストラクタには、値クラスに加え、ターゲット(target)が指定できます。DeluxeSignal インスタンスは指定されたターゲットに代わって、イベントを送出します。

```
DeluxeSignal(target:Object = null, ... valueClasses)
```

GenericEvent は IEvent インターフェイスを備えるイベントクラスで、public プロパティとして bubbles や currentTarget、target のほか signal(イベントを送出したシグナル)を持っています。

では DeluxeSignal が GenericEvent との併用で使用できるように、AlarmClock をアップグレードしましょう。GenericEvent はこの過程で理解できます。

```

package insideria.deluxeSignal
{
    import org.osflash.signals.DeluxeSignal;
    import org.osflash.signals.events.GenericEvent;

    public class AlarmClock
    {
        public var alarm:DeluxeSignal;
        public var message:String;

        public function AlarmClock()
        {
            alarm = new DeluxeSignal(this);
            message = "This is a message from our AlarmClock";
        }

        public function ring():void
        {
            alarm.dispatch(new GenericEvent());
        }
    }
}

```

DeluxeSignal 用の public プロパティの alarm を作成し、AlarmClock のコンストラクタでインスタンス化するとき DeluxeSignal のコンストラクタに this を渡しています。これによって DeluxeSignal は AlarmClock を知ることになります。また後でリスナーからアクセスして確認できるように、もう1つ message という名前の public プロパティも作成しています。これは GenericEvent を使って複数の引数を渡す方法を示す単純な例です。GenericEvent は ring()メソッドで使っています。DeluxeSignal も基本的な Signal と同じように送りますが、今度は dispatch ()に GenericEvent の新しいインスタンスを渡します。この GenericEvent は、その Signal が発せられたクラスと Signal 自体の情報を持っていきます。ではメインのアプリケーションファイルでその情報へのアクセス方法を見てみましょう。

```

package insideria.deluxeSignal
{
    import org.osflash.signals.events.GenericEvent;

```

```
import flash.display.Sprite;

public class WakeUp extends Sprite
{
    private var alarmClock:AlarmClock;

    public function WakeUp()
    {
        alarmClock = new AlarmClock();
        alarmClock.alarm.add(onRing);
        alarmClock.ring();
    }

    private function onRing(event:GenericEvent):void
    {
        trace(event.target);
        trace(event.signal);
        trace(event.target.message);
    }
}
}
```

変わったのはリスナーだけです。リスナーは GenericEvent のパラメータを取っています。これは AlarmClock クラスで dispatch()に渡したものです。情報はここから得ることができます。ここではネイティブのイベントと同じように、ターゲットやクラスのメッセージなどの情報を出力することができ、さらに AlarmClock クラスから送出された Signal にアクセスすることもできます。

これで Signal を2種類学ぶことができました。最後は少し飛躍して、AS3 Signal とネイティブのイベントを合わせて使う方法を見ていきましょう。

NativeSignal

ネイティブのイベントの完全な置き換えを実現する最後のステップは、開発者がネイティブのイベントにアクセスできるようにすることです。Robert Penner はこの作業にも対処しています。

わたしはこの記事のように実際のサンプルが好きなので、実行時別の SWF ファイルをロードする方法を通し

て、NativeSignal クラスを見ていくことにします。

訳者注:

NativeSignal は IEventDispatcher 用の強い型指定がされたクラスで、特定のイベント型とイベントクラスに固定されたミニディスパッチャーです。ターゲット (IEventDispatcher の target) に代わってイベントを送出します。

```
NativeSignal(target:IEventDispatcher = null, eventType:String, eventClass:Class = null)
```

IEventDispatcher は flash.events パッケージのクラスで、リファレンスには「IEventDispatcher インターフェイスは、イベントリスナーを追加または削除するメソッドの定義、特定のタイプのイベントリスナーが登録されているかどうかのチェック、およびイベントの送出手を行う」とあります。IEventDispatcher は EventDispatcher が実装しているので、ActionScript の多くのクラスが当てはまります。これはつまり NativeSignal の target には通常の Sprite や MovieClip が使用できるということで、Sprite や MovieClip のイベントの代行者に NativeSignal が使用できるということです。

ロードする SWF ファイルの作成

まず行うのは、後でロードできる SWF の作成です。次のクラスでは ADDED\_TO\_STAGE イベントを使用するので、NativeSignal はまずここで使います。

```
package insideria.nativeSignals
{
    import org.osflash.signals.natives.NativeSignal;

    import flash.display.Sprite;
    import flash.events.Event;

    public class AddToStage extends Sprite
    {
        public function AddToStage()
        {
            var added:NativeSignal = new NativeSignal(this, Event.ADDED_TO_STAGE, Event);
            added.addOnce(onAdded);
        }
    }
}
```

```

private function onAdded(event:Event):void
{
    graphics.beginFill(0xCCCCCC);
    graphics.drawRect(0, 0, 100, 100);
    graphics.endFill();
}
}
}

```

ご覧のようにこのクラスではローカルの NativeSignal (added) を作成するとき、1つめの target パラメータに this (このクラスのインスタンス) を、2つめの eventType パラメータに使用したいイベントクラスと定数を、3つめのパラメータ eventClass にイベントクラスを渡しています。基本 Signal で学んだように、Signal に1度だけ応答させたいときには addOnce() メソッドを使用します。Signal は1度しか発射されず、その始末を気にする必要はありません。

つづいてネイティブのイベントを扱っているときと同じように、イベントハンドラの onAdded を作成しています。このハンドラは Event 型のパラメータ event を受け取ります。この event パラメータでは、NativeSignal をインスタンス化したときに使ったイベントクラスで型指定しているかどうかを必ずチェックしてください。このイベントハンドラでは、ただ適当なグラフィックを描画して、別のファイルが確かにロードされたということが分かるようにしています。

これで SWF ファイルを作成するクラスができました。

#### ロードするクラスの作成

Sprite を拡張するクラスを作成すると、そこに可視的な要素が追加できます。次のクラスでは NativeSignal を保持する private のプロパティが3つ要ります。これらにはこのクラスの外からアクセスしないので、基本 Signal と異なり private に設定します。

```

private var loadedSignal:NativeSignal;
private var progressSignal:NativeSignal;
private var faultSignal:NativeSignal;

```

次に必要なのはバイナリファイルをロードする基本的なセットアップです。作成するのは URLRequest と Loader、Signal のターゲット (シグナルが代行するイベントの送出者) の IEventDispatcher です。

```
var request:URLRequest = new URLRequest("AddToStage.swf");
var loader:Loader = new Loader();
var signalTarget:IEventDispatcher = loader.contentLoaderInfo;
```

ロード完了イベントと進行イベント、失敗イベント用のネイティブの Signal の作成には NativeSignal クラスを使用します。Loader インスタンスの contentLoaderInfo プロパティは安全を期して IEventDispatcher で型指定します。

```
loadedSignal = new NativeSignal(signalTarget, Event.COMPLETE, Event);
loadedSignal.addOnce(onLoaded);

progressSignal = new NativeSignal(signalTarget, ProgressEvent.PROGRESS, ProgressEvent);
progressSignal.add(onProgress);

faultSignal = new NativeSignal(signalTarget, IOErrorEvent.IO_ERROR, IOErrorEvent);
faultSignal.addOnce(onFault);
```

ネイティブの Signal を作成するときのシンタックスを思い出してください。1つめのパラメータはターゲットで、2つめはイベント定数付きのイベントクラス、3つめは再度イベントクラスです。またイベントハンドラのパラメータは、ネイティブの Signal インスタンスで使用したものと必ず同じイベントクラスに型指定します。コンストラクタで残る最後の処理はローダーのロード動作の開始です。

```
loader.load(request);
```

つづいて Signal を処理するイベントハンドラを作成します。

```
private function onFault(event:IOErrorEvent):void
{
    trace("Something went wrong!");
}

private function onProgress(event:ProgressEvent):void
{
    trace((event.bytesLoaded / event.bytesTotal) * 100);
}
```

```
private function onLoad(event:Event):void
{
    progressSignal.removeAll();
    addChild(event.target.content);
}
```

このコードでロード中の失敗に対応し、ロードの進行状況も表示できます。ロードしたファイルの内容を表示リストに追加するとともに、もう不要になった進行状況の Signal 用リスナーも削除できます。add()と addOnce()の概念が頭に入っていれば、失敗と完了にaddOnce()を使った理由も自ずと明らかでしょう。失敗と完了は1回の発射で十分で、進行状況はロードが成功するまで複数回発射されるからです。

アプリケーションを実行すると、100 x 100 ピクセルの矩形が左上隅に表示されます。ロードに失敗している場合には、このクラスにハードコーディングした通り、コンソールに“Something went wrong!”が表示されます。その場合には URLRequest インスタンスのリンクがロードしたい SWF ファイルをちゃんと指しているか確認してください。

以下がロードするクラスの完成コードです。

```
package insideria.nativeSignals
{
    import org.osflash.signals.natives.NativeSignal;

    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.IEventDispatcher;
    import flash.events.IOErrorEvent;
    import flash.events.ProgressEvent;
    import flash.net.URLRequest;

    public class Loading extends Sprite
    {
        private var loadedSignal:NativeSignal;
        private var progressSignal:NativeSignal;
        private var faultSignal:NativeSignal;
```

```

public function Loading()
{
    var request:URLRequest = new URLRequest("AddToStage.swf");
    var loader:Loader = new Loader();
    var signalTarget:IEventDispatcher = loader.contentLoaderInfo;

    loadedSignal = new NativeSignal(signalTarget, Event.COMPLETE, Event);
    loadedSignal.addOnce(onLoaded);

    progressSignal =
        new NativeSignal(signalTarget, ProgressEvent.PROGRESS, ProgressEvent);
    progressSignal.add(onProgress);

    faultSignal =
        new NativeSignal(signalTarget, IOErrorEvent.IO_ERROR, IOErrorEvent);
    faultSignal.addOnce(onFault);

    loader.load(request);
}

private function onFault(event:IOErrorEvent):void
{
    trace("Something went wrong!");
}

private function onProgress(event:ProgressEvent):void
{
    trace((event.bytesLoaded / event.bytesTotal) * 100);
}

private function onLoaded(event:Event):void
{
    progressSignal.removeAll();
    addChild(event.target.content);
}
}

```

```
}
```

前に述べたようにわたしは実際のサンプルが大好きなので、サンプルをもう1つお見せしましょう。今度はクリックを処理します。AddToStage クラスに次の2つのプロパティを追加します。

```
private var clicked:NativeSignal;  
private var box:Sprite;
```

つづいて Sprite の box に適当なグラフィックを描画し、box を表示リストに追加します。

```
box = new Sprite();  
box.graphics.beginFill(0xCCCCCC);  
box.graphics.drawRect(0, 0, 100, 100);  
box.graphics.endFill();  
addChild(box);
```

最後にネイティブのクリックシグナル(clicked)とそれに対応するイベントハンドラを追加します。MouseEvent を使うので import ステートメントに追加します。

```
import flash.events.MouseEvent;  
  
clicked = new NativeSignal(box, MouseEvent.CLICK, MouseEvent);  
clicked.add(onClicked);  
  
private function onClicked(event:MouseEvent):void  
{  
    trace(" clicked");  
}
```

このファイルを SWF にコンパイルし、SWF をロードします。box をクリックするたびに、コンソールに" clicked"が表示されます。

この例では AS3 Signal を使った別のファイルのロードに加え、クリックリスナーの登録を行っています。ActionScript の通常のネイティブイベントを組み込むにはこのようにします。

まとめ

本記事では、Robert Penner の AS3 Signal を使うとオブジェクト間のやりとりが実に簡単に行えることを述べてきました。みなさんは3つの異なるタイプの Signal とそれぞれの使い方を学びました。わたしは本記事によってみなさんが AS3 Signal に目覚め、実際に使用されるようになることを願います。またコミュニティの人々と同じように AS3 Signal を好きになっていただければ実にうれしく思います。AS3 Signal が大好きなコミュニティの人々はみなさんとリソースを共有したいと思っています。

- [AS3 Signals on GitHub](#)
- [Community examples](#)
- [Templates for FDT](#)

大事なことを忘れていましたが、わたしが作成したサンプルファイルです。 [Signals.zip](#)

PS: AS3 Signal は [Robotlegs](#) でも使用できます。