

「Ten tips for building better Adobe AIR applications」の日本語訳

本ドキュメントは、Adobe サイトで公開されている記事「Ten tips for building better Adobe AIR applications」をヒム・カンパニー 永井勝則が自主的に日本語に訳したものです。

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

knagai@himco.jp

(2010/10)

本ドキュメントの原文は

http://www.adobe.com/devnet/air/articles/10_tips_building_on_air.html

です。

より良い Adone AIR アプリケーション構築のための 10 のティップス

Christian Cantrell

2010/6/25

必要な予備知識

本記事は、Adobe AIR の一般的な理解と ActionScript 3.0 に習熟している読者を前提としています。

必要な製品

Adobe AIR

ユーザーレベル

すべて

AIR 2 がリリースされた今、わたしがこの数カ月で記述してきた AIR コードを見直し、最良のコードスニペットや概念を取り上げてコミュニティと共有するにはよいタイミングです。本記事では、わたしが AIR アプリケーションのパフォーマンスやユーザビリティ(使い勝手)、セキュリティを向上させ、開発をより早く容易にするために使用した 10 のティップスについて述べていきます。

- ・メモリ使用量を低く抑える
- ・CPU の使用量を減らす
- ・慎重に扱う必要のあるデータの保持
- ・“ヘッドレス”アプリケーションの記述
- ・ドックアイコンとシステムトレイアイコンの更新
- ・ネットワーク接続の変化の処理
- ・“デバッグ”と“テスト”モードの作成
- ・ユーザーのアイドル状態の判定
- ・2つめのウィンドウの管理
- ・異なるオペレーティングシステム用のプログラミング

1. メモリ使用量を低く抑える

わたしは最近、[MailBrew](#)というメール通知アプリケーションを記述しました。これはGmailとIMAPアカウントを監視して、新着メッセージがあったときGrowlのような通知を表示して注意を喚起します。MailBrewは新しいメールの到着を知らせつづけるので、つねに起動させておく必要があり、メモリの使用

量を抑える必要があります(図1参照)。

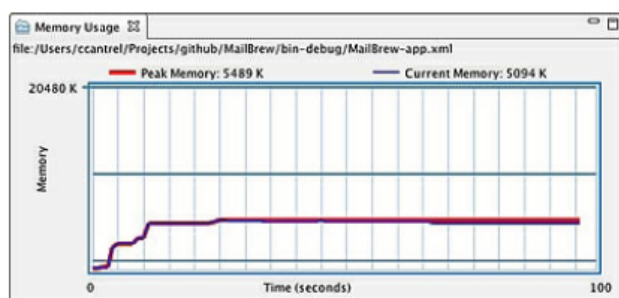


図1: MailBrew は初期化時にメモリを消費し、メール調べるときにも少し消費するが、使用量は常に低い

ランタイムは自動的にガベージコレクション処理を実行するので、AIR 開発者であるみなさんはメモリを明示的に管理する必要はありませんが、これはメモリの管理をしなくてもよいということではありません。実際には、AIR 開発者は新しいオブジェクトの作成に慎重であるべきで、特に参照を維持しているとクリーンアップできないので、これには注意深くあるべきです。以下のティップスは AIR アプリケーションのメモリ使用量を低く安定的に抑えるための参考になります。

- ・イベントリスナーはつねに削除する
- ・XML の破棄も忘れずに行う
- ・dispose()関数を自作する
- ・SQL データベースを使う
- ・アプリケーションをプロファイリングする

イベントリスナーはつねに削除する

これまでもおそらく耳にしたことがあるでしょうが、「イベントを投げるオブジェクトを使ったときには、ガベージコレクション処理されるように、イベントリスナーはすべて削除する」ことは重要です。

次の(単純化した)コードは、わたしが書いた[PluggableSearchCentral](#)というアプリケーションのコードで、イベントリスナーの適切な追加と削除を示しています。

```
private function onDownloadPlugin():void
{
    // URLLoader をローカル変数として作成
    var req:URLRequest = new URLRequest(someUrl);
    var loader:URLLoader = new URLLoader();
```

```

        loader.addEventListener(Event.COMPLETE, onRemotePluginLoaded);
        loader.addEventListener(IOErrorEvent.IO_ERROR,
                                onRemotePluginIOError);

        loader.load(req);
    }

private function onRemotePluginIOError(e:IOErrorEvent):void
{
    // URLLoader を e.target からローカル変数として取得
    var loader:URLLoader = e.target as URLLoader;
    loader.removeEventListener(Event.COMPLETE, onRemotePluginLoaded);
    loader.removeEventListener(IOErrorEvent.IO_ERROR,
                                onRemotePluginIOError);

    this.showError("Load Error", "Unable to load plugin: " +
                    e.target, "Unable to load plugin");
}

private function onRemotePluginLoaded(e:Event):void
{
    var loader:URLLoader = e.target as URLLoader;
    loader.removeEventListener(Event.COMPLETE, onRemotePluginLoaded);
    loader.removeEventListener(IOErrorEvent.IO_ERROR,
                                onRemotePluginIOError);

    this.parseZipFile(loader.data);
}

```

もう一つ、イベントリスナーが容易に削除できるように、イベントリスナー関数を保持する変数を作成するテクニックがあります。

```

public function initialize(responder:DatabaseResponder):void
{
    this.aConn = new SQLConnection();
    // リスナー関数をローカル変数で作成
    var listener:Function = function(e:SQLEvent):void
    {
        aConn.removeEventListener(SQLEvent.OPEN, listener);
    }
}

```

```

        aConn.removeListener(SQLErrorEvent.ERROR, errorListener);
        var dbe:DatabaseEvent =
            new DatabaseEvent(DatabaseEvent.RESULT_EVENT);
        responder.dispatchEvent(dbe);
    };
    // リスナー関数をローカル変数で作成
    var errorListener:Function = function(ee:SQLErrorEvent):void
    {
        aConn.removeListener(SQLEvent.OPEN, listener);
        aConn.removeListener(SQLErrorEvent.ERROR, errorListener);
        dbFile.deleteFile();
        initialize(responder);
    };
    this.aConn.addEventListener(SQLEvent.OPEN, listener);
    this.aConn.addEventListener(SQLErrorEvent.ERROR, errorListener);
    this.aConn.openAsync(dbFile, SQLMode.CREATE, null,
        false, 1024, this.encryptionKey); }

```

XML の破棄も忘れずに行う

Flash Player 10.1 と AIR 1.5.2 では、System クラスの静的関数として disposeXML() が追加されました。これは XML オブジェクトのすべてのノードを逆参照し、ただちにガベージコレクション処理を可能にします。XML を解析するアプリケーションでは、用済みになった XML オブジェクトは必ずこの関数を呼び出すようにします。System.disposeXML() を使用しないと、XML オブジェクトはずっとガベージコレクション処理されない、循環参照となる可能性があります。

次のコードは、Gmail によって生成された XML フィードを解析するコードを単純化したものです。

```

var ul:URLLoader = e.target as URLLoader;
var response:XML = new XML(ul.data);
var unseenEmails:Vector.<EmailHeader> = new Vector.<EmailHeader>();
for each (var email:XML in response.PURL::entry)
{
    var emailHeader:EmailHeader = new EmailHeader();
    emailHeader.from = email.PURL::author.PURL::name;
    emailHeader.subject = email.PURL::title;
    emailHeader.url = email.PURL::link.@href;
}

```

```
        unseenEmails.push(emailHeader);
    }
    var unseenEvent:EmailEvent = new EmailEvent(EmailEvent.UNSEEN_EMAILS);
    unseenEvent.data = unseenEmails;
    this.dispatchEvent(unseenEvent);
    System.disposeXML(response);
```

dispose()関数を自作する

多くのクラスを使って中規模から大規模のアプリケーションを記述するときには、習慣的に“dispose(破棄する)”関数を追加するようにします。実際、この習慣づけを実行しようとする、IDisposable という名前のインターフェイスを作成したくなるでしょう。dispose()関数の目的は、オブジェクトをガベージコレクション処理させなくする可能性のあるすべての参照を、オブジェクトに確実に保持させなくすることです。dispose()では少なくとも、クラスレベルの変数をすべて null に設定します。IDisposable を使ったコードがある箇所では、その処理が終わったら dispose()関数を呼び出します。ほとんどの場合、これらの参照は通常ガベージコレクション処理されるので(コードにバグがないと仮定した場合)、これは必須ではありませんが、参照を明示的に null に設定し、明示的に dispose()関数を呼び出すことで、つぎの2つの重要なメリットが生まれます。

- ・半ば強制的にメモリの配分を考えるようになる。すべてのクラスで dispose()関数を記述すると、オブジェクトのクリーンアップを妨げるインスタンスへの参照をうっかり保持しつづけることがなくなります(メモリリークの原因になります)。
- ・ガベージコレクターの作業を楽にする。すべての参照が null に設定されると、メモリを回収するガベージコレクターの作業が効率的になります。アプリケーションが予測可能な間隔で肥大化するときには(異なるアカウントからの新しいメッセージを調べる MailBrew のように)、その作業の完了後に System.gc() (ガベージコレクション処理を強制的に実行。Flash Player デバッグ版と AIR アプリケーションでのみ有効)を呼び出した方がよいかもしれません。

以下は、MailBrew の明示的なメモリ管理を行うコードを単純化したコードです。

```
private function finishCheckingAccount():void
{
    this.disposeEmailService();
    this.accountData = null;
    this.currentAccount = null;
    this.newUnseenEmails = null;
    this.oldUnseenEmails = null;
```

```
        System.gc();
    }

    private function disposeEmailService():void
    {
        this.emailService.removeEventListener(
            EmailEvent.AUTHENTICATION_FAILED, onAuthenticationFailed);
        this.emailService.removeEventListener(
            EmailEvent.CONNECTION_FAILED, onConnectionFailed);
        this.emailService.removeEventListener(
            EmailEvent.UNSEEN_EMAILS, onUnseenEmails);
        this.emailService.removeEventListener(
            EmailEvent.PROTOCOL_ERROR, onProtocolError);
        this.emailService.dispose();
        this.emailService = null;
    }
}
```

SQL データベースの使用

以下は、AIR アプリケーションでデータを残存させる方法です。

- ・フラットファイル(通常のプレーンテキストのファイル)
- ・ローカル共有オブジェクト
- ・EncryptedLocalStore
- ・オブジェクトの直列化
- ・SQL データベース

これらの方法にはそれぞれメリットとデメリットがあります(その解説は本記事の範囲を超えます)。SQL データベースを使用するメリットの1つは、ファイルから多くのデータをメモリにロードしないので、アプリケーションのメモリ使用量を抑える助けになるということです。たとえば、アプリケーションのデータをデータベースに保持すると、必要なものを必要なときに選び、その処理が終わったら、メモリからそのデータをすぐに削除することができます。

その良い例が MP3 プレイヤーのアプリケーションです。もし全曲のデータを XML ファイルで保持したものの、ユーザーは特定のアーティストの1曲か特定のジャンルの1曲しか見たくなかったという場合には、すべての曲をメモリに一度にロードしたにもかかわらず、ユーザーに見せるのはその一部です。SQL データベースを使用すると、ユーザーが今すぐ見たいと思っているデータを正確に選択し、メモリ使用量を最小限に抑えること

ができます。

アプリケーションのプロファイリング

みなさんがどれだけメモリ管理が得意で、作成したのがごく小さなアプリケーションであっても、リリースする前には必ずアプリケーションのプロファイリングを行うようにします。Flash Builder プロファイラの説明は本記事の範囲を超えますが(プロファイラの使用は科学であると同時に芸術でもあります)、正常に動作する AIR アプリケーションに真剣に取り組むなら、プロファイリングも真剣に行う必要があります。

2. CPU の使用量を減らす

AIR アプリケーションで CPU の使用量について一般的なティップスを述べるのは、アプリケーションが使用する CPU の量がアプリケーションの機能性に深く関与するので、非常に難しいことですが、すべての AIR アプリケーションで CPU の使用量を減らす汎用的な方法が1つあります。それはアプリケーションがアクティブでないとき、そのフレームレートを低くすることです。

Flex フレームワークにはフレームレートスロットル (fps を加減できる仕組み) が組み込まれています。WindowedApplication クラスの backgroundFrameRate プロパティは、アプリケーションがアクティブでないときに使用するフレームレートを示すので、Flex を使用している場合には、単にこのプロパティを 1 のような低い値に設定します。

MailBrew を書いたとき学んだことですが、フレームレートスロットルには関連する要因が少し複雑に絡んでくる場合があります。MailBrew には、新しいメッセージが到着すると、グラデーショナルのアルファトウインで Growl のような通知を表示する通知システムがあります(図2参照)。もちろんこの通知はアプリケーションがアクティブでないときでも表示されるので、滑らかにフェードイン、アウトするには妥当なフレームレートが必要になります。したがってわたしは Flex のフレームレートスロットルの仕組みをオフにして、自分で記述する必要がありました。

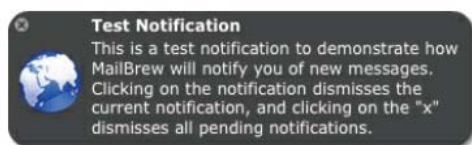


図2: MailBrew の通知はフェードインしフェードアウトするので、アプリケーションがアクティブでないときでも、フレームレートは最低 24 が必要になる

わたしが使ったテクニックは、アプリケーションのデフォルトのフレームレートを、わたしの ModelLocator クラスで指定する方法です。[Cairngorm Framework](#) を使用されている方は見おぼえがあるでしょう。使用し

たことのない方に申し上げると、ModelLocatorはMVCフレームワークのモデルを表す単なるクラスです。ModelLocatorでは定数が次のように定義されています。

```
public static const DEFAULT_FRAME_RATE:uint = 24;
```

その後、アプリケーションの activate と deactivate イベントを次のように監視します。

```
this.nativeApplication.addEventListener(Event.ACTIVATE, onApplicationActivate);  
this.nativeApplication.addEventListener(  
    Event.DEACTIVATE, onApplicationDeactivate);
```

また Bindable (ソースプロパティの変更時、自動的にソースプロパティの値が宛先プロパティにコピーされる Flex の機能)の変数を次のように定義します。

```
[Bindable] public var frameRate:uint;
```

アプリケーションのフレームレートを管理する部分でフレームレートを変更し、ChangeWatcher を使って変数 frameRate の変更を監視します。

```
ChangeWatcher.watch(  
    ModelLocator.getInstance(), "frameRate", onFrameRateChange);
```

するとコードのどこかで ModelLocator の frameRate 変数が変更されると、onFrameRateChange() 関数が呼び出されます。

```
private function onFrameRateChange(e:PropertyChangeEvent):void  
{  
    this.stage.frameRate = ml.frameRate;  
}
```

最後、アプリケーションがアクティブ化または非アクティブ化されると、それにしたがってフレームレートを更新します。

```
private function onApplicationActivate(e:Event):void  
{  
    this.ml.frameRate = ModelLocator.DEFAULT_FRAME_RATE;
```

```
}  
  
private function onApplicationDeactivate(e:Event):void  
{  
    this.ml.frameRate = 1;  
}
```

この基礎的な構造によって、次のことが可能になります。

- ・ModelLocator の frameRate 変数を変更するだけで、アプリケーションのフレームレートがどこでも変更できる
- ・アプリケーションがアクティブでないとき、フレームレートを落とすことができる（バックグラウンドで、またはメインアプリケーションのウィンドウが閉じられているとき）
- ・フレームレートを、通知を表示する前に DEFAULT_FRAME_RATE で指定した値に上げ、通知のフェードアウト後、また落とすことができる

フレームレートのスロットルフレームワークの自作は Flex のフレームワークを使うよりも複雑になりますが、柔軟性が必要で、かつアクティブでないときアプリケーションの CPU 使用量を抑えたいときには、この追加的な作業にも価値が生まれます。

3. 慎重に扱う必要のあるデータの保持

前に述べたように、AIR アプリケーションでデータを残存させる方法はいくつかありますが、どれにも一長一短があります。しかしデータを安全に保持したい場合には、つぎの3つの選択肢が最適です。

- ・EncryptedLocalStore クラス
- ・暗号化された SQL データベース
- ・暗号化の自作

ユーザー名とパスワードを保持したいだけなら、EncryptedLocalStore (ELS) クラスをおすすめします。しかしもっと大きなデータを保持したいときには、暗号化されたデータベース (AIR で完全にサポートされます) を使うか、暗号化を自作し、暗号化したデータをディスクに保存する方法を取る方がよいでしょう (自作した暗号化の管理は本記事の範囲を超えるので、以下では暗号化されたデータベースを使うものとします)。

ELS の最もすぐれた点は、データの暗号化と復号化 (暗号を元に戻すこと) にパスワードやパスフレーズが必要ないことです。これによってアプリケーションの使い勝手がかなり向上します。たとえば、サービスに使用

するユーザー名とパスワードを保持しても、それを復号化するためにまた別のパスワードやパスフレーズをユーザーに求めていたのでは、ユーザーにとって何のメリットもありません。では ELS が保持できないほど大きなデータを暗号化するとき、ELS と同等の良好な使い勝手を提供するにはどうすればよいのでしょうか？

訳者注：

EncryptedLocalStorage はデータが 10MB を超えると低速化する場合があります。またモバイルデバイスでは EncryptedLocalStorage はサポートされません。

それには次の手順にしたがいます。

1. 適切にランダム化されたパスワードを生成します。
2. EncryptedLocalStorage を使ってパスワードを保持します。
3. そのパスワードを使って、暗号化されたセキュアなデータベースキーを生成します。
4. そのキーを使って、データベースを暗号化し、復号化します。

これは複雑そうに見えますが、ありがたいことに必要なコードはほとんど用意されています。ではこの角手順を詳しく見ていきましょう。

ランダムなパスワードの生成

次のコードは、わたしが記述した、ランダムで推測しづらいパスワードを生成する関数のセットです。

```
private static const POSSIBLE_CHARS:Array =
    ["abcdefghijklmnopqrstuvwxyz", "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
     "0123456789", "~!@#%&*_()-+=[] | ; : ¥ ¤ ¢ £ ¤ ¤ < . > / ? "];

private function generateStrongPassword(length:uint = 32):String
{
    if (length < 8) length = 8;
    var pw:String = new String();
    var charPos:uint = 0;
    while (pw.length < length)
    {
        var chars:String = POSSIBLE_CHARS[charPos];
        var char:String = chars.charAt(
            this.getRandomWholeNumber(0, chars.length - 1));
        var splitPos:uint = this.getRandomWholeNumber(0, pw.length);
```

```

        pw = (pw.substring(0, splitPos) + char +
              pw.substring(splitPos, pw.length));
        charPos = (charPos == 3) ? 0 : charPos + 1;
    }
    return pw;
}

private function getRandomWholeNumber(min:Number, max:Number):Number
{
    return Math.round(((Math.random() * (max - min)) + min));
}

```

これでランダムなパスワードが作成できたので、次はそれを保持します。

ランダムなパスワードの保持

パスワードをセキュアに保持する最良の方法は、EncryptedLocalStorageクラスを使ってデータベースの暗号化キーを生成する方法です。ELSのAPIは使い方が簡単ですが、わたしは自分で作った[as3preferenceslib](#)プロジェクトを通常使います。as3preferenceslibを使用するメリットは、アプリケーションのすべての設定（プリファレンス）を同じAPIで保持できることです。as3preferenceslibは舞台裏でELSを使ってセキュアに指定したデータを保持します。以下はそのコードです。

```

var ml:ModelLocator = ModelLocator.getInstance();
var prefs:Preference = ml.prefs;
var databasePassword:String =
    prefs.getValue(PreferenceKeys.DATABASE_PASSWORD);

if (databasePassword == null)
{
    databasePassword = this.generateStrongPassword();
    // 第3引数はセキュアなストレージを示す
    ml.prefs.setValue(
        PreferenceKeys.DATABASE_PASSWORD, databasePassword, true);
    ml.prefs.save();
}

```

暗号化されたセキュアなデータベースキーの生成

暗号化されたセキュアなデータベースキーの生成は複雑ですが、ラッキーなことにそのコードは用意されています。わたしは[as3corelibプロジェクト](#)のEncryptionKeyGeneratorを使っています。

EncryptionKeyGenerator を使うと、特定のユーザーアカウントと特定のマシンをランダムなパスワードに関連づけることで、暗号化されたデータの最上位にセキュリティレイヤーが追加されます。別の言い方をすると、たとえ何者かがランダムなパスワードを見破ったとしても、ユーザーのマシンを所有し、そのユーザーとしてログインしない限り、何の役にも立たないのです。

EncryptionKeyGenerator を使うときには、EncryptionKeyGenerator が返す暗号化キーを保持しないことが重要です。そうでなく、その種に使用するパスワードを保持し、オンデマンドで暗号化キーを生成します。以下はこのテクニックの例です。

```
// 前述のコードを使って Preference オブジェクトから databasePassword を取得した後、
var keyGenerator:EncryptionKeyGenerator = new EncryptionKeyGenerator();
var encryptionKey:ByteArray = keyGenerator.getEncryptionKey(databasePassword);
// すると encryptionKey を使った、データベースの暗号化と復号化が行える
```

データベースの暗号化と復号化

暗号化されたセキュアなデータベースキーが得られたので、後はデータベースに接続するときに、それを渡すだけです。次のサンプルコードでは(イベントリスナーが欠けている簡略化版です)、暗号化されたデータベースファイルをロードし、接続を作成する方法を示しています。

```
var keyGenerator:EncryptionKeyGenerator = new EncryptionKeyGenerator();
var encryptionKey:ByteArray = keyGenerator.getEncryptionKey(databasePassword);
var dbFile:File =
    File.applicationStorageDirectory.resolvePath("myEncryptedDatabase.db");
var aConn:SQLConnection = new SQLConnection();
aConn.openAsync(dbFile, SQLMode.CREATE, null, false, 1024, this.encryptionKey);
```

4. “ヘッドレス”アプリケーションの記述

メインウィンドウが閉じられた後も実行をつづけるアプリケーションは“ヘッドレス”アプリケーションと呼ばれることがあります。Mac や Windows システムの多くのアプリケーションはこのパラダイム(枠組み)を使っています。IM やメールクライアントなどのアプリケーションがシステムトレイに最小化されることはあまりありません(図3参照)。



図3: Windows のシステムトレイに最小化された MailBrew

AIR アプリケーションは、メインアプリケーションのウィンドウが閉じられたとき終了するよう設計することも、ヘッドレスアプリケーションとして実行することもできます。メインアプリケーションのウィンドウが閉じられた後もアプリケーションに実行を継続させる一番簡単な方法は、NativeApplication の autoExit プロパティを次のように false に設定することです。

```
private function onApplicationComplete():void
{
    NativeApplication.nativeApplication.autoExit = false;
}
```

Flex フレームワークを使っている場合には、次のように WindowedApplication タグの autoExit 属性を使ってこのプロパティを設定することもできます。

```
<s:WindowedApplication
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:c="com.mailbrew.components.*"
    width="500" height="400" minWidth="500" minHeight="400"
    showStatusBar="false" backgroundFrameRate="-1"
    autoExit="true"
    applicationComplete="onApplicationComplete();">
```

メインアプリケーションのウィンドウが閉じられたときアプリケーションを終了させなくしたので、ユーザーがアプリケーションを再度必要としたとき(おそらく、ドックやシステムトレイのアイコンをクリックしたとき)のメインアプリケーションの再オープンを考える必要があります。アプリケーションがこの種の相互操作性をサポートするための、最も簡単な設計は、メインアプリケーションのインターフェイスを、NativeWindow の子としてそれ自体のコンポーネントに置くことです。次のコードでは、隠されるかシステムトレイに最小化された後、メインアプリケーションを再オープンするクロスプラットフォームのテクニックを示しています。

```
private function onApplicationComplete():void
```

```

{
    NativeApplication.nativeApplication.autoExit = false;
    if (NativeApplication.supportsDockIcon)
    {
        NativeApplication.nativeApplication.addEventListener(
            InvokeEvent.INVOKE, onShowWindow);
    }
    else if (NativeApplication.supportsSystemTrayIcon)
    {
        SystemTrayIcon(
            NativeApplication.nativeApplication.icon).addEventListener(
                ScreenMouseEvent.CLICK, onShowWindow);
    }
}

private function onShowWindow(e:Event):void
{
    var mainApplicationUI:MainApplicationUI = new MainApplicationUI();
    mainApplicationUI.open(true);
}

```

ヘッドレスアプリケーションを記述するもう1つのテクニックは、メインアプリケーションのウィンドウを閉じるのを止め、その代わりにただ隠す方法です。このテクニックでは、メインアプリケーションのウィンドウを実際には閉じないので、NativeWindow の autoExit プロパティを false に設定する必要があります。記述するコードでは次のように、ウィンドウを閉じるのを止め、visible プロパティを false に設定します。

```

private function onWindowClosing(e:Event):void
{
    e.preventDefault();
    this.visible = false;
    ml.frameRate = 1;
}

```

メインアプリケーションのウィンドウを元に戻すテクニックは前のものと似ていますが、メインアプリケーションの UI のインスタンスを新たに作成するのではなく、ただアプリケーションの NativeWindow.visible プロパティを再び true に戻します。

```
private function onShowWindow(e:Event):void
{
    this.visible = true;
    ml.frameRate = ModelLocator.DEFAULT_FRAME_RATE;
    this.nativeWindow.activate();
}
```

メインアプリケーションのvisibleプロパティをトグルする(オン/オフを切り替える)このテクニックの方が簡単で、ほとんどの場合でうまく動作します。これが意味を持たないのは、ユーザーがメインアプリケーションのインスタンスを複数開けるようにしたい[PixelWindow](#)のようなアプリケーションの場合だけです。

5. ドックアイコンとシステムトレイアイコンの更新

ヘッドレスアプリケーションで視覚的な存在を保つには、Mac のドックや Windows のシステムトレイが利用できます。これらのアイコンはユーザーにメインアプリケーションのウィンドウを元に戻す方法を提供し、エンドユーザーに情報を提供する方法をアプリケーションに与えます。AIR にはアプリケーションアイコンの上にテキストを表示できる API はありませんが、ビットマップをダイナミックに生成する API があるので、これをドックやシステムトレイのアプリケーションアイコンの更新に利用できます(図4参照)。



図4: 未読のメッセージ数を表示するドックの MailBrew アイコン

次のコードは MailBrew から取ったサンプルで、ドックアイコンの上にテキストとグラフィックを追加する方法を示しています。これによりユーザーはひと目で未読のメッセージ数が分かります(図4参照)。システムトレイアイコンでも同様のテクニックが使用できますが、システムトレイアイコンの画像は 16 ピクセルの正方形なので、アイコン画像の上に文字を多くせるのはいささか困難です。Windows 7 ではもっと表現力のあるタスクバーアイコンが使用できるので、AIR の API でも将来的なサポートが計画されています。

```
// Windows の場合には処理せずリターン。アイコンはシステムトレイではきれいに表示されない。
if (NativeApplication.supportsSystemTrayIcon) return;
```

```
// 未読メッセージの数を数える関数
var unseenCount:uint = getUnreadMessageCount();
var unreadCountSprite:Sprite = new Sprite();
unreadCountSprite.width = 128;
unreadCountSprite.height = 128;
unreadCountSprite.x = 0;
unreadCountSprite.y = 0;

var padding:uint = 10;

// FTE の API を使って、最良の外見のテキストにする
var fontDesc:FontDescription = new FontDescription("Arial", "bold");
var elementFormat:ElementFormat = new ElementFormat(fontDesc, 30, 0xFFFFFFFF);
var textElement:TextElement =
    new TextElement(String(unseenCount), elementFormat);
var textBlock:TextBlock = new TextBlock(textElement);
var textLine:TextLine = textBlock.createTextLine();
textLine.x = (((128 - textLine.textWidth) - padding) + 2);
textLine.y = 32;
unreadCountSprite.graphics.beginFill(0xE92200);
unreadCountSprite.graphics.drawEllipse((((128 - textLine.textWidth) - padding) - 3),
    2, textLine.textWidth + padding, textLine.textHeight + padding);
unreadCountSprite.graphics.endFill();
unreadCountSprite.addChild(textLine);
var shadow:DropShadowFilter = new DropShadowFilter(3, 45, 0, .75);
var bevel:BevelFilter = new BevelFilter(1);
unreadCountSprite.filters = [shadow, bevel];
var unreadCountData:BitmapData = new BitmapData(128, 128, true, 0x00000000);
unreadCountData.draw(unreadCountSprite);

// Dynamic128IconClass は埋め込まれている
var appData:BitmapData = new Dynamic128IconClass().bitmapData;
appData.copyPixels(unreadCountData, new Rectangle(0, 0, unreadCountData.width,
    unreadCountData.height), new Point(0, 0), null, null, true);
var appIcon:Bitmap = new Bitmap(appData);
```

```
// Windows のシステムトレイアイコンを変更したい場合には、この配列に 16x16 のアイコンを追加する  
InteractiveIcon(NativeApplication.nativeApplication.icon).bitmaps = [appIcon];
```

6. ネットワーク接続の変化の処理

クラウドに移動するデータが増えつづけると、それにアクセスしローカルにキャッシュするデスクトップアプリケーションがますます重要になってきます。Adobe AIR がこういったアプリケーションの記述に完璧なプラットフォームだと言えるのは、次の理由からです。

- ・多くのプロトコルのサポート: ランタイム自体とサードパーティ製の ActionScript ライブラリ間では、希望するようなプロトコルでも、デスクトップクライアントと Web サービス間のデータ交換に使用できます。また HTTP や TCP ソケットの上位に自作のプロトコルを記述することもできます。
- ・マルチプラットフォームのサポート: Web は本質的にクロスプラットフォームなので、Web サービスの上位にデスクトップクライアントを記述すると、それもクロスプラットフォームになります。
- ・Web テクノロジーのサポート: AIR は Web テクノロジーをサポートするので、Web アプリケーションに使用するツールとスキルがデスクトップクライアントの構築にも使用できます。

Web サービスの上位にデスクトップクライアントを記述するときの課題の1つは、ネットワーク接続の判定です。WiFi や 3G(もうすぐ 4G)などの高速ワイヤレスデータプロトコルが急増しても、つねに接続しているわけではありません。したがってネットワーク接続に依存するアプリケーションでは、ネットワークに接続しているかどうか明確に知る情報が必要になります。

この課題に取り組む最初の試みは、NativeApplication クラスの NETWORK_CHANGE イベントです。NETWORK_CHANGE イベントは、ネットワーク接続が有効になるか無効になったとき発射されます。われわれは当初、これで十分だと思っていました。しかし特定のサービスに到達できるかどうかを開発者に知らせるにはこの情報では不十分だということがすぐに判明しました。

ネットワーク接続はたとえば、来ては去っていくものです。これは VPN 接続にオープンとクローズが、仮想マシンにスタートと終了が、ワイヤレスネットワークに範囲の内と外が、ケーブルに抜き差しがあるのと同じです。また言うまでもなく、アプリケーションのサービスがどこにあるのか、公開されたインターネット上なのか、ファイアーウォールの内部なのか、はたまたローカルマシン上なのか、予見することは不可能な場合があります。最後に、サービスに到達できると判定できたとしても、アクセスする瞬間、そのサービスが稼働し、応答できる状態にあるという保証はありません。こういったすべての要因はわれわれに、もっと包括的な API が必要であることを示唆し、その使用方法に関する何らかのベストプラクティスの必要性を示すものでした。

わたしは、デスクトップクライアントアプリケーションは一般的に2つのカテゴリに収まることを見出しました。1

つはあらかじめ分かっているサービスにアクセスするアプリケーション(たとえば Twitter や Facebook クライアント)で、もう1つは予見できないさまざまなサービスにアクセスするアプリケーション(RSS アグリゲーターやメール、IM クライアントなど)です。わたしは自分の経験から、これら2つのタイプのアプリケーションの接続の変化をそれぞれで処理することが理にかなっていると思っています。

あらかじめ分かっているサービスにアクセスするアプリケーション

アプリケーションがアクセスする Web サービスが分かっている場合には、`air.net.URLMonitor` クラスを使った可用性(使えるのかどうか)の監視が最も簡単です(HTTP ではなく TCP を使う場合には、`air.net.SocketMonitor`)。URLMonitor クラスは本質的に、指定された URL を指定された間隔でポーリングし、状態の変化を知らせます。以下はその例です。

```
private var urlMonitor:URLMonitor;
private function onCreateComplete():void
{
    var req:URLRequest =
        new URLRequest("http://www.myserver.com/myservice");
    // 正常な結果を表す状態コード
    this.urlMonitor = new URLMonitor(req, [200, 304]);
    // 毎分
    this.urlMonitor.pollInterval = 60 * 1000; // Every minute
    this.urlMonitor.addEventListener(StatusEvent.STATUS, onStatusChange);
    this.urlMonitor.start();
}

private function onStatusChange(e:Event):void
{
    if (this.urlMonitor.available)
    {
        // すべて OK
    }
    else
    {
        // Service is not available.サービスは利用できないので、
        // ユーザーに警告を出すことを考える
    }
}
```

任意のサービスにアクセスするアプリケーション

アプリケーションがユーザーによって構成され(メールクライアントなど)、どのサービスにアクセスするか不明な場合や、アクセスするサービスの数が分からない場合(RSS アグリゲーターなど)には、URLMonitor は実用的ではありません。アプリケーションがどのサービスにアクセスするのか、予見度が低くなるほど、そのサービスに実際にアクセスできるのかどうか、知ることは難しくなります。たとえば、アプリケーションが公開されたインターネットには信頼性の高い接続が持てないと分かっている場合でも、ファイアーウォール内の RSS フィードを収集する必要があるかも知れません。また利用できるネットワーク接続がまったくない場合でも、ローカルマシンで稼働するサービスにアクセスさせたいかも知れません。ネットワーク接続はその予見と判定がどんどん難しくなっているため、多くの環境下におけるベストプラクティスとしては、ただ接続を作成し、それが失敗したらエラーを報告させるという方法が考えられます。

MailBrew はこの好例です。MailBrew は複数のメールアカウントにアクセスするよう構成できるアプリケーションなので、サービスに実際に到達できるのかどうかは操作してみるまで分かりません。したがって MailBrew では以下の事柄を行うことで、ネットワークのエラーを処理しています。

- ・問題の発生を示すイベント(`IOErrorEvent` や `HTTP_RESPONSE_STATUS_EVENT` など)をすべて登録する
- ・接続の問題が見つかったら、サービスを到達不可能と指定する、データベースのフラグを更新する
- ・サービスが利用できないことを、UI を更新してユーザーに知らせる。図5では、わたしの2つの Gmail アカウントにはアクセスでき(外部にはネットワーク接続できるので)、Adobe アカウントにはアクセスできない(VPN 接続がないので)ことを示しています。

Note: サービスの可用性を URLMonitor を使って調べる場合、URLMonitor は当てにできません。なぜなら URLMonitor は、直近に監視したときに使用可能と見なしたサービスが、アプリケーションが実際にアクセスしたタイミングで使用できない場合でも、常に使用可能と見なすからです。したがって、接続の問題の発生を示す適切なイベントを常に監視し(イベントリスナーで)、その問題を処理する必要があります。

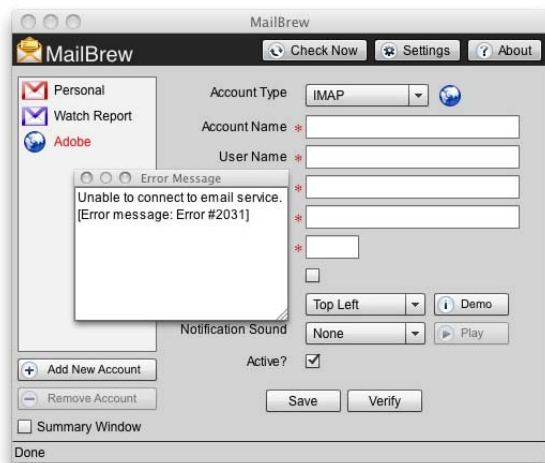


図5: 接続エラーが発生するとアカウント名が赤に変わる。ユーザーはオプションでエラーメッセージウィンドウを開くこともできる

ネットワーク接続は複雑になる可能性があります、エンドユーザーが直観的に理解できる情報を与える方法をアプリケーションに付与することが大切です。ここで述べた2つのテクニックは、接続の信頼度や予見度にかかわらず、すべての環境でみなさんのアプリケーションにすぐに適用できる方法です。

7. “デバッグ”と“テスト”モードの作成

アプリケーションの開発者は、コードの記述とはまったく反復的な作業であると知っています。そのワークフローは通常、コードを記述し、アプリケーションを実行してテストし、これをアプリケーションの規模に応じて、何十回、何百回、何千回と繰り返します。

さらにアプリケーションが外部のサービスにアクセスする場合には、アクセスする頻度が制限されたり (Twitter や IM クライアントの場合)、リモートサービスへのアクセスで単にその処理に時間がかかる (何百回も繰り返すようなとき) といった事実から、処理がもっと面倒になることが考えられます。このような場合わたしは、テストモードとデバッグモードという2つのテクニックを使います。

テストモード

TweetCards という最初のモバイル AIR アプリケーションを記述したとき、Twitter からのデータの取得はその毎回の繰り返して増減できないと早々に知りました。これによって開発のスピードが遅くなるだけでなく、アクセスする頻度が制限されることになり (Twitter は一定時間置かないと接続要求を拒否します)、またネットワーク接続のない場所ではアプリケーションの開発ができなくなりました (飛行機の中でも行っていたので)。



図6: ダミーデータを使った“テストモード”で動作する TweetCards

その答えは、認証フィールドのユーザー名とパスワードに“test”を入力したときには、テストモードを有効にするという方法です。次のコードはこの概念を示しています。

```
private function onSaveAccountInfo(e:MouseEvent = null):void
{
    var username:String = this.usernameInput.value;
    var password:String = this.passwordInput.value;
    var ml:ModelLocator = ModelLocator.getInstance();
    ml.testMode = (this.usernameInput.value ==
                  "test" && this.passwordInput.value == "test");
    ml.credentials =
    {"username":this.usernameInput.value, "password":this.passwordInput.value};
    ml.currentScreen = Screen.READ_SCREEN;
}
```

アプリケーションはテストモードのときには、Twitter にデータの要求を出さず、ダミーのテストデータを使います。

```
private function getTweets():void
{
    var ml:ModelLocator = ModelLocator.getInstance();
    if (ml.testMode)
    {
        this.createTestData();
    }
    else
```

```
    {  
        this.queryTwitter();  
    }  
}
```

これによりデータが即座にロードされるので、Twitter に要求を出さなくてもアプリケーションのテストが行えます。

デバッグモード

もう1つのテクニックは MailBrew を書いていたとき発見したもので、アプリケーションに“デバッグモード”を構築します。デバッグモードは、ユーザーのメールアカウントを調べるコードを書いたら、アプリケーションのスタート時すぐに必要になります。そのような場合には、アプリケーションで何かをテストしたいときに毎回、ボタンの位置のような小さなことであっても、いくつかのメールサービスに要求を出す必要があるため、結果的に開発スピードが鈍ることになります。これを回避するのが次のコードです。

```
private function onApplicationComplete():void  
{  
    // ADL から実行している場合、アプリケーションをデバッグモードにする  
    ModelLocator.debugMode = Capabilities.isDebugger;  
}
```

訳者注:

Capabilities.isDebugger は、そのシステムがデバッグ用の特別なバージョンか(true)、または正式にリリースされたバージョンか(false)を示すプロパティで、Flash Player のデバッグ版と ADL で実行するときには true に設定されます。したがってデバッグ時には ModelLocator.debugMode が true になります。

InitCommand() (アプリケーションの初期化時に実行する命令) では、次のコードを追加します。

```
if (!ModelLocator.debugMode) new CheckMailEvent().dispatch();
```

これで問題は解決できます。このようにすると、MailBrew を ADL から実行するとき、CheckMailEvent は送出されなくなります。この方法にはいくつかのメリットもあります。たとえば、わたしは ADL から実行するときには AIR 2 の新しいグローバルなエラー処理機能を追加したくなかったので、このコード

```
this.loaderInfo.uncaughtErrorEvents.addEventListener(  
    UncaughtErrorEvent.UNCAUGHT_ERROR, onUncaughtError);
```

を次のように変えました(実際には1行)。

```
if (!ModelLocator.debugMode)
    this.loaderInfo.uncaughtErrorEvents.addListener(
        UncaughtErrorEvent.UNCAUGHT_ERROR, onUncaughtError);
```

最後、Updater.update()や NativeApplication.startAtLogin という API があります。これらは ADL から呼び出されるとランタイム例外を投げるので、開発やテスト環境時には意味をなしません。アプリケーションでこれらの API を使うときには、デバッグモードであるかどうかを調べる条件でラップするようにします。

訳者注:

Updater.update()は AIR アプリケーションの更新を行うメソッドで、ADL でアプリケーションをテストしているときに呼び出すと、IllegalOperationError 例外が発生します。NativeApplication.startAtLogin はユーザーがログインしたとき常にそのアプリケーションを自動的に起動するかどうかを指定するプロパティで、ADL から起動されている場合には(アプリケーションが実際にインストールされていないので)IllegalOperationError 例外が発生します。

Note : Capabilities.isDebugger は、モードを調べたいすべての場所で使うのではなく、グローバルな debugMode フラグを設定する1箇所ですべての場所で使っていることに注目してください。このメリットは、アプリケーションを debugMode にするかどうかを1箇所ですべて簡単に変更できることにあります。たとえば、コマンドラインの引数を使ってアプリケーションをデバッグモードに設定する機能を付加し、アプリケーションのインストール版をテスト目的のデバッグモードに設定することができます。

8. ユーザーのアイドル状態の判定

AIR アプリケーションで通知システムが記述できるようにしようとしていたとき、われわれはまた、ユーザーが実際にコンピュータの前にいるのかいないのかを判定できる方法が必要なことに気づきました。これにより、通知を実際に表示することに意味があるのかないのかが分かります。たとえば、MailBrew が通知の表示と注意喚起の再生をつづけても、ユーザーがそこにいて見たり聞いたりしないことには、意味がありません。

われわれはこの問題を NativeApplication クラスの USER_IDLE と USER_PRESENT イベントで解決しました。これらのイベントを登録すると、ユーザーがアイドル状態(何もしない状態)に入ったとき、コンピュータに戻ったときに、アプリケーションに通知が行われます。以下はその例です。

```
private function onCreationComplete():void
```

```

{
    // 秒単位--デフォルトは 5 分
    NativeApplication.nativeApplication.idleThreshold = 2 * 60;
    NativeApplication.nativeApplication.addEventListener(
        Event.USER_IDLE, onUserIdle);
    NativeApplication.nativeApplication.addEventListener(
        Event.USER_PRESENT, onUserPresent);
}

private function onUserIdle(e:Event):void
{
    // この 2 分間、キーボード入力もマウス入力もない
}

private function onUserPresent(e:Event):void
{
    // ユーザーが戻った!
}

```

わたしは、わたしのアプリケーションでこれらのイベントを直接登録しても全然便利に使えないことに気づきました。通常は通知用のフレームワークのライブラリにこれを行わせるのが妥当です。たとえば MailBrew では、MailBrew が自動的に使用する通知フレームワークにこれらのイベントを登録し、ユーザーがアイドル状態に入ったら通知をキューに入れ、ユーザーが復帰したら自動的に通知表示を始めます。またこれらの API を使うときには、みなさんのアプリケーションのワークフローを考慮することが大切です。ビデオを表示するアプリケーションで言うと、ユーザーはメディアの再生中、おそらくアイドル状態ではないので、アイドルングのタイマーは無効にしておく方がよいでしょう。

単純な AIR アプリケーションですが、これらの API を実際に使用したサンプルとして、[Screen Protection Factor](#) (SPF) という、わたしが記述したスクリーンセーバーがあるので、参考にしてください。

9. 2つめのウィンドウの管理

AIR アプリケーションでは、プリファレンスやアバウトボックスといった2つめのウィンドウを開くことはあまり一般的ではありません。AIR で2つめのウィンドウを開くのは簡単ですが、同じインスタンスを複数開くということには相当な用心が必要です。AIR にはモーダルウィンドウ(子ウィンドウとして作成され、ユーザーがそれに適切に応答しない限り、制御が親ウィンドウに戻らない仕組みになっているウィンドウ)の概念がないので、ユ

ユーザーに一度に複数のウィンドウを開かせたくない場合には、自分でプログラミングする必要があります。



図7: MailBrew の設定ウィンドウ、一度に1つだけ開くことができる

ラッキーなことに、これは実に簡単です。わたしは2つめのウィンドウの使用に便利な関数を持った WindowManager というクラスを作成しました。

```
/**
 * まだ開いていないなら、NativeWindow のインスタンスを返す
 */
public static function getWindowByTitle(title:String):NativeWindow {
    var allWindows:Array = NativeApplication.nativeApplication.openedWindows;
    for each (var win:NativeWindow in allWindows)
    {
        if (win.title == title)
        {
            return win;
        }
    }
    return null;
}
```

この getWindowByTitle() では次のようにウィンドウを開きます。

```
private function onOpenSettings(e:MouseEvent):void
```

```

{
    var win:NativeWindow =
        WindowManager.getWindowByTitle(WindowManager.PREFERENCES);
    if (win != null)
    {
        win.activate();
    }
    else
    {
        var prefsWin:PreferencesWindow = new PreferencesWindow();
        prefsWin.open(true);
    }
}

```

このテクニックでは、たとえユーザーがうっかり[Preferences]ボタンをダブルクリックしても、プリファレンスウィンドウ(PreferencesWindow)は1度に1つしか開けません。プリファレンスウィンドウがすでに開いていたときには、それを複製するのではなく、ただアクティブ化する(前面に移動させフォーカスを与える)だけです。

以下はおまけで、WindowsManager クラスのもう1つの関数です。

```

/**
 * ウィンドウをメインモニタのセンターに置く
 */
public static function centerWindowOnMainScreen(win:NativeWindow):void
{
    var initialBounds:Rectangle = new Rectangle(
        (Screen.mainScreen.bounds.width / 2 - (win.width/2)),
        (Screen.mainScreen.bounds.height / 2 - (win.height/2)),
        win.width,
        win.height
    );

    win.bounds = initialBounds;
    win.visible = true;
}

```

10. 異なるオペレーティングシステム用のプログラミング

AIR はクロスプラットフォームのランタイムですが、これはプラットフォーム間で違うものが作成できないとか、違うものにすべきではない、ということではありません。本記事では前に、Windows のシステムトレイと Mac OS X のドックでアイコンを使うときにはそのサイズを変える必要があると述べましたが、プラットフォーム間の違いはもっと深部に及ぶことがあります。たとえばわたしは Mac OS X の MailBrew の設定ウィンドウに、新しいメールが届いたときにドックアイコンがバウンスする(跳ねる)、Windows にはないオプションを設け、Windows の設定ウィンドウには、タスクバーアイコンが明滅する、Mac にはないオプションを作りました。

こういったプラットフォーム特有の問題を処理するテクニックはいくつかあります。どれが正しくどれがすぐれているかということではなく、わたしが MailBrew の記述で経験したテクニックを2, 3紹介しましょう。以下は PreferencesWindow コンポーネントで使った、プラットフォームの違いの処理に関するコードです。

```
<s:Window creationComplete="onCreationComplete();">
  <fx:Script>
    <![CDATA[
      private function onCreationComplete():void
      {
        if (NativeApplication.supportsDockIcon)
        {
          this.currentState = "supportsDockIcon";
          this.bounceDockIconCheckbox.selected =
prefs.getValue(PreferenceKeys.APPLICATION_ALERT, false);
        }
        else
        {
          this.currentState = "supportsSystemTray";
          this.flashTaskBarCheckbox.selected =
prefs.getValue(PreferenceKeys.APPLICATION_ALERT, false);
        }
      }
    ]]>
  </fx:Script>
  <s:states>
    <s:State name="supportsDockIcon"/>
    <s:State name="supportsSystemTray"/>
  </s:states>
  <s:VGroup width="100%" height="100%">
```

```

        <s:Group width="100%" includeIn="supportsDockIcon">
            <s:Label text="Bounce Dock Icon" fontWeight="bold" y="5"
left="5"/>

            <s:Label y="20" width="210" left="5">新しいメッセージが届いたと
きドックアイコンをジャンプさせますか？ </s:Label>

            <s:CheckBox id="bounceDockIconCheckbox" y="5"
right="20"/>

        </s:Group>
        <s:Group width="100%" includeIn="supportsSystemTray">
            <s:Label text="Flash Task Bar Icon" fontWeight="bold" y="5"
left="5"/>

            <s:Label y="20" width="210" left="5">新しいメッセージが届いたと
きタスクバーアイコンをチカチカさせますか？ </s:Label>

            <s:CheckBox id="flashTaskBarCheckbox" y="5" right="20"/>

        </s:Group>
    </s:VGroup>
</s:Window>

```

ご覧のようにこのコードのポイントは、いたって便利な Flex の states 機能の使用にあり、作業の大部分はこれが行います。とはいえ無論、方法はほかにもあります。たとえば MailBrew では、メインアプリケーションウインドウのツールバーがプラットフォームによって違います。Mac OS X ではタイトルとロゴは左上にあり、ボタンは右にあります(図8参照)。



図8：後ろが Windows の、前が Mac OS X の MailBrew。ツールバーのレイアウトが違う点に注意。

しかし Windows で右上にインタラクティブなものを配置しようとする、うっかりウィンドウコントロールをクリックしてしまい、ウィンドウ自体を閉じてしまう恐れがあります。つまりボタンの位置は Mac OS X と逆の方がよいということになります。

```
<s:Group creationComplete="onCreationComplete();">
  <fx:Script>
    <![CDATA[
      private function onCreationComplete():void
      {
        if (NativeApplication.supportsSystemTrayIcon) //
Windows
          {
            this.logoBitmap.visible = false;
            this.logoLabel.right = 3;
            this.buttonGroup.left = 5;
          }
        else
          {
            this.logoBitmap.visible = true;
            this.logoBitmap.left = 4;
            this.logoLabel.left = 31;
            this.buttonGroup.right = 5;
          }
        }
      ]>
    </fx:Script>
    <s:BitmapImage id="logoBitmap"
source="{ModelLocator.getInstance().TopLeftLogo}" top="3"/>
    <s:Label id="logoLabel" text="MailBrew" top="11" fontSize="20"
color="0xf2f2f2" fontFamily="_sans"/>
    <s:HGroup id="buttonGroup" top="5">
      <mx:Button id="checkNowButton" label="Check Now"
icon="{ModelLocator.getInstance().CheckNowIconClass}"/>
      <mx:Button id="settingsButton" label="Settings"
icon="{ModelLocator.getInstance().ConfigureIconClass}"/>
      <mx:Button id="aboutButton" label="About"
```

```
icon="{ModelLocator.getInstance().AboutIconClass}"/>
  </s:HGroup>
</s:Group>
```

この例では `states` を使わず、`left` や `right`、`visible` プロパティを使って再配置や消去を行っています。

このアプリケーションでは、プラットフォームごとの違いを最小限に抑えようとしていますが、実際にはケースバイケースで変わってくるでしょう。わたしの意見では、プラットフォーム間に違いはないという振りをするコードより、プラットフォーム間の違いを処理するコードを書く方が良いように思えます。