Cocoa はやっぱり! 出張版

2002.1

7. UNIX コマンドの実行

■ 今回のテーマ

今回のテーマは、UNIX コマンドの実行です。Apple のサイトにあるサンプルプログラムの Moriarity の説明をしながら、Cocoa アプリケーションから UNIX コマンドを実行する方法について解説を行っていきます。このサンプルは、Apple サイト内の以下の URL にありますので、まずはダウンロードしておいてください。

http://developer.apple.com/samplecode/Sample_Code/Cocoa/Moriarity.htm

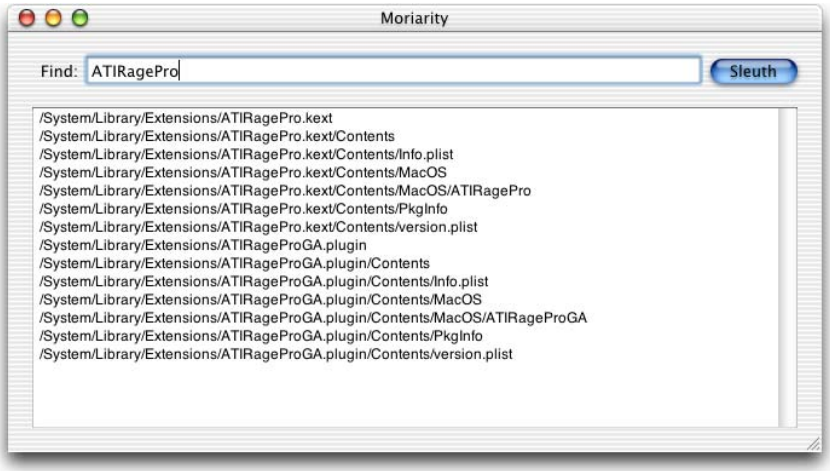
◎ 推奨環境

この解説は、以下の環境を前提にしていますので、ご確認ください。

・Mac OS X 10.1 以降
・Project Builder 1.1.1 (December 2001 Developer Tools) 以降
・Interface Builder 2.2 (December 2001 Developer Tools) 以降

■ アプリケーション概要

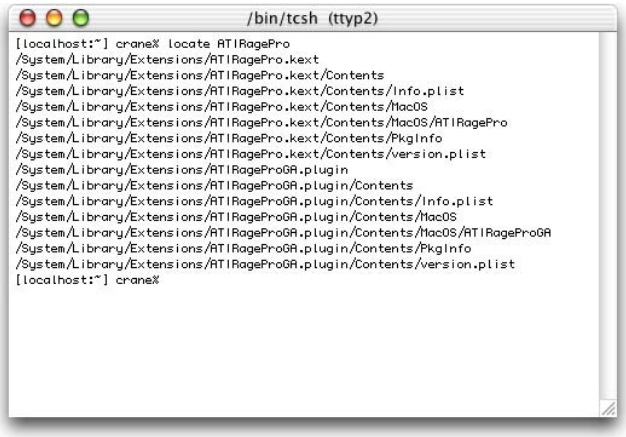
この Moriarity というアプリケーションは、以下のような画面になっています。



【図】 アプリケーションの実行画面

Find : の右側に文字列を入力して Sleuth ボタンをクリックすると、入力した文字列を含むパスが中央の `NSTextView` に表示されます。

見た目は、普通のアプリケーションですが、内部では UNIX の `locate ( /usr/bin/locate )` というコマンドを実行しています。コマンドのパラメータをウィンドウ上部の `NSTextField` から取り出して渡して、コマンドの実行結果をウィンドウ中央の `NSTextView` に表示しています。実際に、Terminal アプリケーションで同じことをやってみた結果が以下の図です。コマンドラインから " `locate ATIRagePro` " とタイプしています。



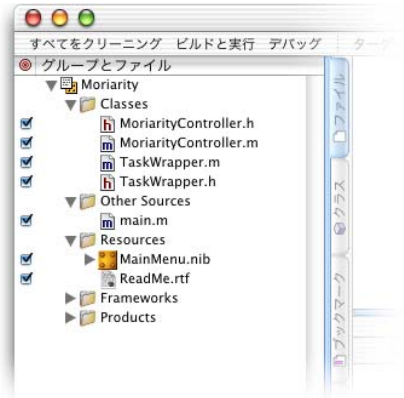
【図】Terminal アプリケーションでの実行画面

このように Cocoa アプリケーションからは UNIX のコマンドを実行することができるのです。

> **ちょっと補足**：Moriarity というのは、シャーロック・ホームズに出てくる「モリアリティ教授」の名前から取ったもののようです。また、Sleuth というのは「探偵として働く」という意味です。親友である「ワトソン博士」の名前も後から出てきます。

■ プロジェクトとモジュールの構成

プロジェクトファイルを Project Builder で開いて、プロジェクトを構成するファイルを確認してみましょう。



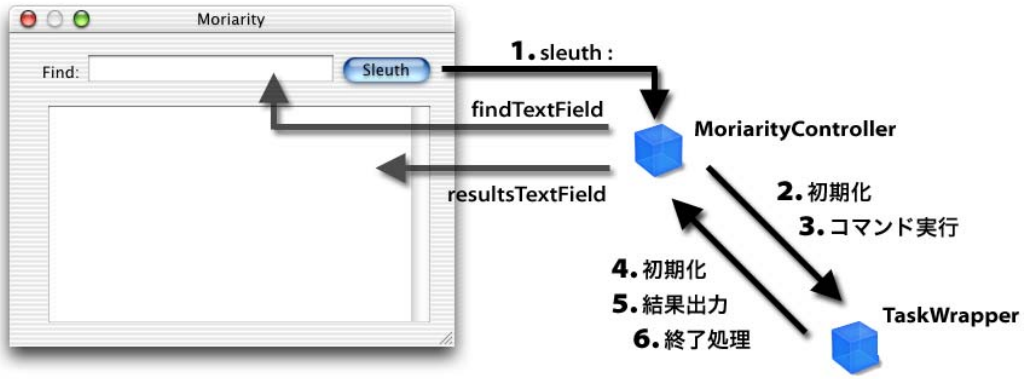
【図】 プロジェクトの構成ファイル

`TaskWrapper.m` / `TaskWrapper.h` というのが、UNIX のコマンドとのやり取りを行う **`TaskWrapper`** クラ

スを定義しているファイルです。Cocoa の Foundation フレームワークには **NSTask という UNIX コマンドを実行するためのクラス**がありますが、これをさらに簡単に使えるようにするためのクラスが TaskWrapper です。

MoriarityController.m / MoriarityController.h というのが、ウィンドウと TaskWrapper の仲介役になる **MoriarityController** クラスを定義しているファイルです。ユーザーが入力した内容をアウトレットの findTextField から取り出して TaskWrapper に渡して起動し、また、TaskWrapper からの実行結果をアウトレットの resultsTextField に書き込むという仕事を行います。

これらのモジュールの関係を図にしてみました。

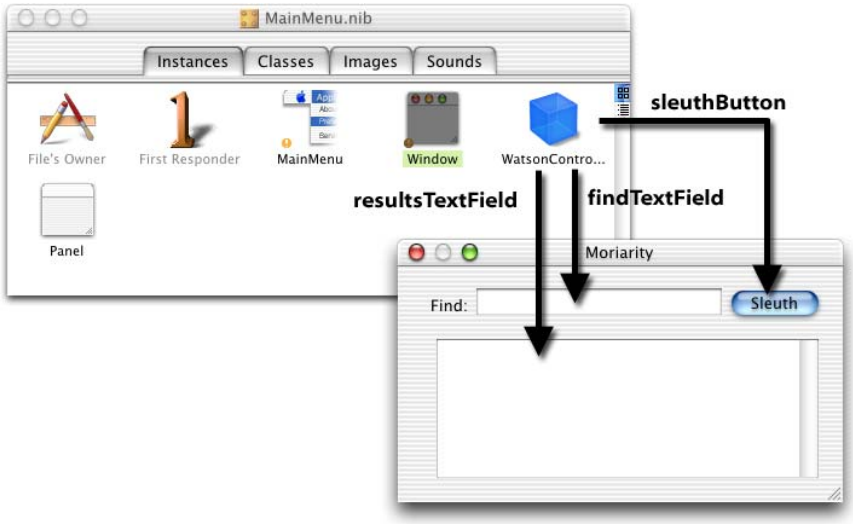


【図】 モジュールの関係図

簡単に処理の流れを追ってみます。まず、ユーザーが Sleuth ボタンをクリックします。すると、TaskWrapperController の sleuth : メソッドが実行されます…**1**。そして、TaskWrapper を初期化して…**2**、コマンドの実行を指示します…**3**。そうすると、TaskWrapper は MoriarityController に対して出力先の初期化 (このサンプルではテキストフィールドのクリア) を行って…**4**、UNIX コマンドを実行開始します。得られた結果を随時 MoriarityController に渡して画面に表示させます…**5**。コマンド実行が終了したら終了処理を行います…**6**。

このサンプルは、MVC (Model – View – Controller) の設計になっていて、ウィンドウが View、MoriarityController が Controller、TaskWrapper が Model になっています。うまく作れば、Model である TaskWrapper に変更を加えることなく、ファイルからパラメータを読み込んで、結果をファイルに出力するというような別のアプリケーションに作り替えることが出来ます。

次に、プロジェクトウィンドウの Resources の中にある MainMenu.nib をダブルクリックして開いてみましょう。Interface Builder で見ると以下のようになっています (ウィンドウは小さくしてあります)。**WatsonController** というのが MoriarityController のインスタンスになっています。



【図】 MainMenu.nib の中身

TaskWrapper のインスタンスはありませんが、WatsonController (MoriarityController) の中に含まれています。MoriarityController.h を見てみると以下のようになっています。**searchTask** というのがそれです。

MoriarityController.h > @interface MoriarityController

```
@interface MoriarityController : NSObject <TaskWrapperController>
{
    IBOutlet id findTextField;
    IBOutlet id resultsTextField;
      : 省略
    BOOL findRunning;        // 現在実行中
    TaskWrapper *searchTask; // タスクラッパー
}
  - (IBAction) sleuth : (id) sender; // アクションメソッド
      : 省略
@end
```

このヘッダーから MoriarityController は、**TaskWrapperController** プロトコルを持っていることが分かります。TaskWrapper は、コントローラを制御して出力等を行いますので、コントローラに必要なインターフェイス (Objective-C でいうところのプロトコル) を TaskWrapperController として定義していて、それを MoriarityController が持っています。

TaskWrapper.h > @protocol TaskWrapperController

```
@protocol TaskWrapperController
  - (void) appendOutput : (NSString *) output; // 結果の出力
  - (void) processStarted;  // 初期化
  - (void) processFinished; // 終了処理
@end
```

続いて TaskWrapper のヘッダーも見てみましょう。

TaskWrapper.h > @protocol TaskWrapperController

```
@interface TaskWrapper : NSObject
{
    NSTask  *task;                              // 実行するコマンド
    id      <TaskWrapperController> controller; // コントローラ
    NSArray *arguments;                         // コマンドのパラメータ
}
  - (id) initWithController : (id <TaskWrapperController>) controller
                  arguments : (NSArray *) args;  // 初期化
  - (void) startProcess; // コマンド実行開始
  - (void) stopProcess;  // コマンド実行中止
@end
```

task には NSTask のインスタンス記憶し、arguments にはコマンド実行のためのパラメータを記憶しています。出力をコントローラに任せるために、コントローラのインスタンスも controller に記憶しています。

■ MoriarityController の詳細

では、詳細の説明に入っていきます。ユーザーが Sleuth ボタンをクリックすると実行される sleuth : メソッドからです。

MoriarityController.m > sleuth :

```
- (IBAction) sleuth : (id) sender {

  if ( findRunning ) { // コマンド実行中
    [ searchTask stopProcess ]; // 実行中止
      : 省略
  }
  else { // コマンド未実行時
      : 省略
    searchTask = [ [ TaskWrapper alloc ]
        initWithController : self
                 arguments : [ NSArray arrayWithObjects :
                                     @"/usr/bin/locate",
                                     [ findTextField stringValue ],
                                     nil ] ]; // 初期化
    [ searchTask startProcess ]; // コマンド実行
  }
}
```

コマンドが既に実行されている場合は、TaskWrapper のコマンド実行を止める **stopProcess** メソッドを呼んだり、その他の終了処理を行います。実行されていない場合は、TaskWrapper クラスのインスタンスであ

る searchTask を生成します。alloc した後、**initWithController : arguments :** メソッドを呼んで初期化します。このメソッドでコントローラと UNIX コマンドとパラメータを指定できます。コントローラは自分自身ですので self を渡します。パラメータは、配列で渡しますが、最初の要素は実行するコマンドのパスで、それ以降がコマンドのパラメータになっています。そして、**startProcess** メソッドを呼ぶことで UNIX コマンドが実行されます。stopProcess と startPorcess については、後程説明します。

startProcess メソッドを呼ぶと、searchTask からコントローラ側の初期化としての processStarted メソッドが呼び返されます。結果を表示するテキストフィールドをクリアしたり、Sleuth ボタンのタイトルを変更したりします。

MoriarityController.m > processStarted

```
- (void) processStarted {
    findRunning = YES;
    [ resultsTextField setString : @"" ];     // テキストフィールドのクリア
    [ sleuthButton      setTitle : @"Stop" ]; // Sleuthボタンのタイトル変更
}
```

その後、UNIX コマンドが実行開始されます。UNIX コマンドから結果が出力されると、TaskWrapper から **appendOutput :** メソッドが呼ばれます。このメソッドは、繰り返し呼ばれます。NSTextView にテキストを追加してスクロールする処理を行っています。

MoriarityController.m > appendOutput :

```
- (void) appendOutput : (NSString *) output {

  [ [ resultsTextField textStorage ]
      appendAttributedString : [ [ [ NSAttributedString alloc ]
                                     initWithString : output ] autorelease ] ];
  [ self performSelector : @selector(scrollToVisible:)
              withObject : nil
              afterDelay : 0.0 ];
}

- (void) scrollToVisible : (id) ignore {
    [ resultsTextField scrollRangeToVisible :
       NSMakeRange( [ [ resultsTextField string ] length ], 0 ) ];
}
```

コマンド実行が終了したら、TaskWrapper から processFinished メソッドが呼ばれます。ここで Sleuth ボタンのタイトルを元に戻したりしています。

MoriarityController.m > processFinished

```
- (void) processFinished {
  findRunning = NO;
  [ sleuthButton setTitle : @"Sleuth" ]; // Sleuthボタンのタイトル変更
}
```

■ TaskWrapper の詳細

ここからは、TaskWrapper クラスの詳細について説明を行います。TaskWrapper は、Foundation フレームワークの NSTask という UNIX のコマンドを呼び出すクラスと直接やり取りを行うクラスですので、ここからが本題となります。

◎ コマンドの実行まで

まずは、TaskWrapper の初期化メソッドである **initWithController : arguments :** からです。このメソッドは alloc された直後に呼ばれます。やりとりを行うコントローラのインスタンスと、コマンドのパスとパラメータを配列で受け取って記憶します。

MoriarityController.m > initWithController : arguments

```
- (id) initWithController : (id <TaskWrapperController>) cont
                 arguments : (NSArray *) args {
    self       = [ super init ];  // スーパークラスによる初期化
    controller = cont;            // コントローラを記憶
    arguments  = [ args retain ]; // コマンドとコマンドに渡すパラメータを記憶
    return self;
}
```

次は、実際にコマンドを実行する startProcess メソッドです。ちょっと長くなりますが、まずは全体を見てください。最初のところでは、先程説明しました processStarted というコントローラを初期化するメソッドを実行しています。

MoriarityController.m > startProcess

```
- (void) startProcess {

    [ controller processStarted ];  // コントローラの初期化

    // 実行準備
    task = [ [ NSTask alloc ] init ];                          // 生成と初期化
    [ task setStandardOutput : [ NSPipe pipe ] ];              // 標準出力変更
    [ task setStandardError  : [ task standardOutput ] ];      // 標準エラー変更
    [ task setLaunchPath : [ arguments objectAtIndex : 0 ] ]; // コマンド設定
    [ task  setArguments : [ arguments subarrayWithRange :
                             NSMakeRange( 1, ([arguments count] - 1) ) ] ];
                                                               // パラメータ設定
    // 標準出力からのデータを受け取るための通知設定
    [ [ NSNotificationCenter defaultCenter ]
        addObserver : self
           selector : @selector(getData:)
               name : NSFileHandleReadCompletionNotification
             object : [ [ task standardOutput ] fileHandleForReading ] ];

    [ [ [ task standardOutput ] fileHandleForReading ]
                                  readInBackgroundAndNotify ];

    [ task launch ]; // コマンド実行

}
```

では、細かく見ていきます。

```
    // 実行準備
    task = [ [ NSTask alloc ] init ];                          // 生成と初期化
    [ task setStandardOutput : [ NSPipe pipe ] ];              // 標準出力変更
    [ task setStandardError  : [ task standardOutput ] ];      // 標準エラー変更
```

まず、最初に NSTask の生成と初期化を alloc と init で行います。

UNIX コマンドの多くは、実行結果を標準出力に出力します。そのため、何もしないと結果はコンソールに出力されてしまいます。このサンプルでは、テキストフィールドに出力していますので、コンソールに出力させずに、アプリケーション側で受け取る必要があります。NSTask には、**標準出力の出力先を変更する setStandardOutput : というメソッド**がありますので、これを使います。

出力先には、ファイルハンドル（ NSFileHandle ）とパイプ（ NSPipe ）のどちらかが使用できます。ファイルハンドルを使用すると、指定のファイルへ書き出すことが出来ますし、パイプを使用するとパイプに出力内容がたまりますので、そこからファイルハンドルを使って読み出すことが出来ます。このサンプルでは、パイプに出力させて、そこから読み出してテキストフィールドに書き込んでいます。

標準エラーも同じく出力先をパイプにしておきます。メソッドは **setStandardError :** を使います。

NSTask : コマンドの標準出力先を変更

書式
```
- (void) setStandardOutput : (id) file
```
入力
file : 出力先のファイル(NSFileHandle)かパイプ(NSPipe)を指定する。

NSTask : コマンドの標準出力先を取得

書式
```
- (id) standardOutput
```
出力
返り値 : 設定されているファイル(NSFileHandle)かパイプ(NSPipe)のインスタンス。

NSTask : コマンドの標準エラーの出力先を変更

書式
```
- (void) setStandardError : (id) file
```
入力
file : ファイル(NSFileHandle)かパイプ(NSPipe)を指定する。

NSTask : コマンドの標準エラーの出力先を取得

書式
```
- (id) standardError
```
出力
返り値 : 設定されているファイル(NSFileHandle)かパイプ(NSPipe)のインスタンス。

パイプを作るには、NSPipe のクラスメソッドである **pipe** を実行するだけです。

NSPipe : パイプのインスタンスを生成

書式
```
+ (id) pipe
```
出力
返り値 : パイプのインスタンス。

続いて、実行するコマンドを **setLaunchPath :** で設定し、コマンドに渡すパラメータを **setArguments :** で設定します。

```
[ task setLaunchPath : [ arguments objectAtIndex : 0 ] ]; // コマンド設定
[ task  setArguments : [ arguments subarrayWithRange :
                         NSMakeRange( 1, ([arguments count] - 1) ) ] ];
                                                    // パラメータ設定
```

NSTask ： 起動するコマンドのパスを変更

書式
```
- (void) setLaunchPath : (NSString *) path
```
入力
path ： 起動するコマンドのフルパスを指定する。

NSTask ： 起動するコマンドのパスを取得

書式
```
- (NSString *) launchPath
```
出力
返り値 ： 起動するコマンドのフルパス。

NSTask ： コマンドに渡すパラメータを変更

書式
```
- (void) setArguments : (NSArray *) arguments
```
入力
arguments ： コマンドに渡すパラメータ。

NSTask ： コマンドに渡すパラメータを取得

書式
```
- (NSArray *) arguments
```
出力
返り値 ： コマンドに渡すパラメータ。

このサンプルでは使用していませんが、カレントディレクトリを指定したり、取得したりするメソッドも用意されています。

NSTask ： 実行時のカレントディレクトリを変更

書式
```
- (void) setCurrentDirectoryPath : (NSString *) path
```
入力
path ： 実行時のカレントでディレクトリのフルパス。

NSTask ： 実行時のカレントディレクトリを取得

書式
```
- (NSString *) currentDirectoryPath
```
出力
返り値 ： 実行時のカレントでディレクトリのフルパス。

通知設定のところは後回しにして、最後の **launch** メソッドを先に説明します。この launch メソッドを実行することで、UNIX コマンドが実行されます。UNIX コマンドは非同期で動きますので、この launch メソッドを呼んでもすぐに制御は戻ってきます。

```
    [ task launch ]; // コマンド実行

}
```

NSTask : コマンドを実行

書式
```
  - (void) launch
```
備考
コマンドは非同期に実行されるため、すぐに制御が戻ってくる。

◎ 通知を用いた実行結果の読み出し

先程、標準出力をパイプに変更しました。そして、パイプからの出力を読み出すのにはファイルハンドルを用いると説明しました。具体的には以下のようにしてファイルハンドルを得ることが出来ます。

```
NSFileHandle *fout = [ [ task standardOutput ] fileHandleForReading ] ];
```

standardOutput メソッドからパイプ (NSPipe) のインスタンスが返ってきますので、NSPipe 読み込み用のファイルハンドルを取得する **fileHandleForReading** を呼ぶことでファイルハンドルが得られます。

NSPipe : パイプから情報を読み出すためのファイルハンドルを取得

書式
```
  - (NSFileHandle *) fileHandleForReading
```
出力
返り値 : パイプから情報を読み出すためのファイルハンドルのインスタンス。
availableData、readDataToEndOfFile、readDataOfLengthで読み出す。

NSPipe : パイプに情報を書き込むためのファイルハンドルを取得

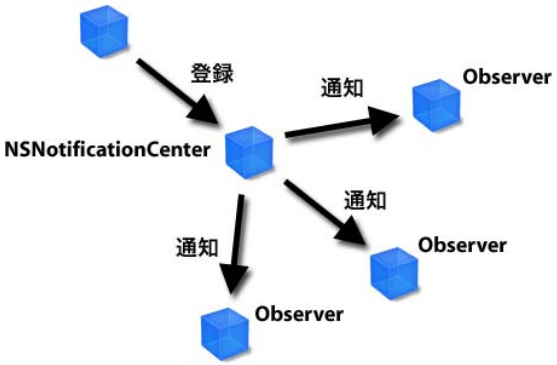
書式
```
  - (NSFileHandle *) fileHandleForWriting
```
出力
返り値 : パイプに情報を書き込むためのファイルハンドルのインスタンス。
writeDataで書き込む。

後は読み出し用のメソッド availableData などを用いれば、読み出しが行えます。ただし、どういったタイミングで、パイプからデータが出力されてくるのかは、実行する UNIX コマンドに依存します。今回の locate コマンドでは、瞬時にデータが出てきますが、インターネットから情報を取ってくるようなコマンドもあります。そのため、このサンプルでは、読み出しメソッドを呼ぶのではなくデータが得られた時にファイルハンドルに通知してもらうようなスタイルを採っています。

Cocoa のフレームワークには、通知 (Notification) の仕組みがあります。「 こういうことが起きたらここに通知してほしい 」ということを通知センター (**NSNotificationCenter**) に知らせておくと、通知が発生

した時点で、指定のインスタンスの指定のメソッドを呼んでくれます。そのメソッドには通知情報（**NSNotification**）が送られますので、その中身を見ると、通知の種類や通知に附随するデータなどを取得することができます。

この通知というのは、単なるコールバックの仕組みではありません。あるオブジェクトが通知センターに通知を登録したら、その**通知を待っている全てのオブジェクト対して通知が送られる**のです。この**通知を待っているオブジェクトのことをオブザーバ（Observer）**と呼びます。



【図】通知の概念図

ファイルハンドルは、読み出せるデータがあるときに通知をする機能がありますので、TaskWrapper は、その通知を受け取れるように通知センターにオブザーバ登録を行います。それを行っているのが以下です。

```
    // 標準出力からのデータを受け取るための通知設定
    [ [ NSNotificationCenter defaultCenter ]
        addObserver : self
           selector : @selector(getData:)
               name : NSFileHandleReadCompletionNotification
             object : [ [ task standardOutput ] fileHandleForReading ] ];
```

通知センターのインスタンスは **defaultCenter** メソッドで得られます。このインスタンスに対して、**addObserver : selector : name : object :** メソッドを使って、オブザーバを登録します。

NSNotificationCenter : オブザーバ登録

書式

```
- (void) addObserver : (id        ) anObserver
             selector : (SEL       ) aSelector
                 name : (NSString *) notificationName
               object : (id        ) anObject
```

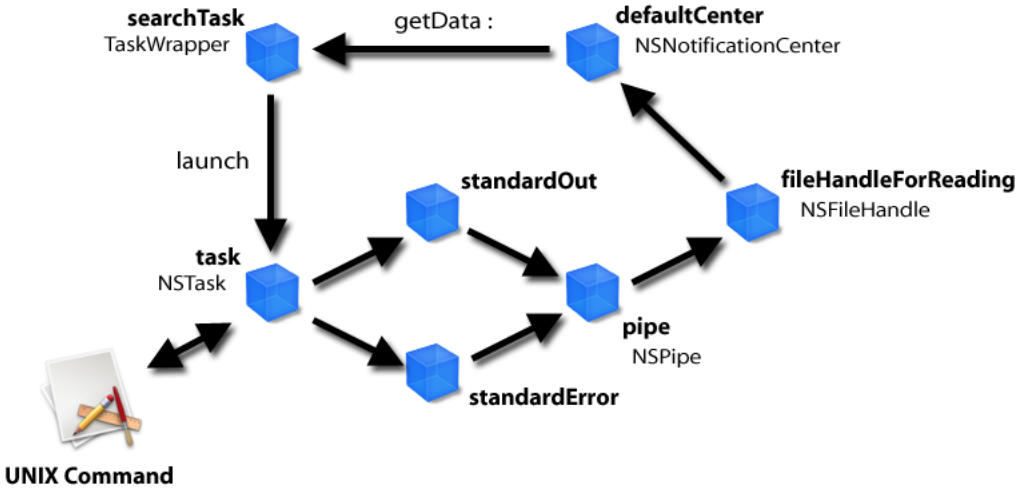
入力

```
anObserver : 通知先のインスタンス。
selector   : 通知先のインスタンスの実行するメソッド。
name       : 通知してほしい通知の種類。nilにすると全ての通知を受ける。
object     : 通知の発信先の指定。nilにすると全てのインスタンスからの通知を受ける。
```

最初の 2 つのパラメータがオブザーバの設定で、通知先を自分（self）に、実行するメソッドを getData : にしています。残りの 2 つのパラメータは、受け取りたい通知の指定を行っています。name に

NSFileHandleReadCompletionNotification を指定することで、ファイルハンドルからデータが読み出せるようになった時の通知が来ます。object には、パイプの読み出し用のファイルハンドルを指定していますので、もし、アプリケーションの中にこれ以外のファイルハンドルがあったとしても、そのインスタンスからの通知は届きません。

さて、ここで UNIX のコマンドを起動してから出力結果がどのような流れで通知されるかを簡単な図にしてみましたので、これでイメージを掴んでください。



【図】UNIX コマンドの起動と出力の流れ

ファイルハンドルは、読み出せるデータがあったとしても、通常は読み出しメソッドが呼ばれるのを待っています。そこで、ファイルハンドルから通知を受けるためには、そのファイルハンドルに通知を出すようにお願いをしておく必要があります。そのためには、**readInBackgroundAndNotify** メソッドを実行しておきます。

```
    [ [ [ task standardOutput ] fileHandleForReading ]
                                    readInBackgroundAndNotify ];
```

NSFileHandle : バックグラウンドでの読み出しと通知の指定

書式

```
  - (void) readInBackgroundAndNotify
```

備考

NSFileHandleから通知を受けるためには、このメソッドを予め実行しておく必要がある。

◎ コマンドの実行結果の受け取り

これで通知が発生するようになりましたので、実行したコマンドからの出力の受け取りの方法について説明します。先程、通知が発生した時に呼び出されるメソッドに指定した getData : を見ていきます。**通知を受けるメソッドは、必ず NSNotification クラスのパラメータを 1 つ持たなければなりません**ので以下のようになっています。

MoriarityController.m > getData :

```
- (void) getData : (NSNotification *) aNotification {

  // 出力されたデータを取得
  NSData *data = [ [ aNotification userInfo ]
                       objectForKey : NSFileHandleNotificationDataItem ];

  if ( [ data length ] ) { // 文字列に変換して出力
    [ controller appendOutput :
        [ [ [ NSString alloc ] initWithData : data
                                   encoding : NSUTF8StringEncoding ]
                                   autorelease ] ];
    } else {
        [ self stopProcess ]; // データがとれなくなったら終了
    }

    if ( task ) { // タスクがある時は、再度、通知を発行してもらう
        [ [ aNotification object ] readInBackgroundAndNotify ];
    }
    else { // タスクがなくなっていたら終了 ( エラー処理 )
        [ self stopProcess ];
    }
}
```

通知から出力データを取り出すには **userInfo** メソッドを使います。この userInfo メソッドの返り値は辞書になっています (通知の種類によっては沢山の情報を持っているため) ので、辞書から objectForKey : メソッドを使って出力結果を取り出します。辞書から取り出すためのキーは **NSFileHandleNotificationDataItem** です。これで、UNIX コマンドが出力したデータが NSData クラスとして得られるので、これを文字列に変換して appendOutput : メソッドを使って、コントローラに出力します。

NSNotification : 通知に添付されているデータを取得

書式

```
- (NSDictionary *) userInfo
```

出力

返り値 : 通知に添付されているデータのインスタンス。データがない場合はnilが返る。

NSNotification : 通知の名前を取得

書式

```
- (NSString *) name
```

出力

返り値 : 通知の名前。

NSNotification : 通知に関連するインスタンスを取得

書式
```
- (id) object
```
出力
返り値 : 通知に関連するインスタンス。通知を行ったインスタンスのことが多いが、通知に依存する。
インスタンスが無い場合はnilが返る。

データが取れなくなってデータ長が 0 になったら終了処理を行います。そうでない場合は、再度ファイルハンドルに対して通知発行を依頼して、次の通知を待ちます。

続いては、コマンド実行中に、Stop ボタンを押された場合の処理です。sleuth : メソッド経由で stopProcess メソッドが呼ばれます。ここでは、コントローラの終了処理を行った後、オブザーバの削除とコマンドの中止を行っています。

MoriarityController.m > stopProcess

```
- (void) stopProcess {

    [ controller processFinished ]; // コントローラー終了処理
    controller = nil;

    // オブザーバー削除
    [ [ NSNotificationCenter defaultCenter ]
          removeObserver : self
                    name : NSFileHandleReadCompletionNotification
                  object : [ [ task standardOutput ] fileHandleForReading ] ];

    [ task terminate ]; // コマンド実行中止

}
```

NSNotificationCenter : オブザーバーの削除

書式
```
- (void) removeObserver : (id        ) anObserver
                   name : (NSString *) notificationName
                 object : (id        ) anObject
```
入力
anObserver : 削除するオブザーバのインスタンス。
name : 削除する通知の種類。nilにすると全ての通知を解除する。
object : 削除する通知の発信先の指定。nilにすると全てのインスタンスの通知を解除する。

NSTask : 実行時を中止

書式
```
- (void) terminate
```

コマンドの実行を一時停止する機能もあります。suspend で一時停止、resume で実行再開です。suspend を複数回実行した場合は、同じ回数 resume をすることで初めて実行が再開されます。

NSTask : 実行を一時停止

書式
```
- (BOOL) suspend
```
出力
返り値 : 成功 – YES、失敗 – NO (実行されていない時など)
備考
複数回実行可能で、同回数のresumeで実行が再開される。

NSTask : 実行を再開

書式
```
- (BOOL) resume
```
出力
返り値 : 成功 – YES、失敗 – NO (実行されていない、suspendされていない時など)
備考
suspendと同回数実行すると再開される。

■ まとめ

以上のように Cocoa アプリケーションからは比較的簡単に UNIX のコマンドを呼び出すことが出来ます。UNIX のコマンドの中には便利なものも多いですし、既存の UNIX コマンドに GUI を付けて使いやすくすることもできるわけです。

また、Cocoa アプリケーションはパッケージ形式になっていて、色んなファイルをアプリケーションの中に閉じ込めておくことが出来ますので、今までと違った開発スタイルを採ることも出来ます。パッケージの中に UNIX のコマンドだけでなく、perl のスクリプトファイルを閉じ込めておいて perl を起動してスクリプトを実行させることも簡単です。

Cocoa を使って、UNIX のパワフルなアプリケーションに、Mac OS の使いやすさを付加したアプリケーションを作るというのも、これからの新しいスタイルとなるでしょう。