ORACLE

ZFS STORAGE APPLIANCE

Oracleホワイト・ペーパー
2014年2月

スクリプトを使用した Oracle ZFS Storage Applianceの効果的な管理

ORACLE

目次

概要

Oracle ZFS Storage Applianceの管理には通常、高度なグラフィカル・ユーザー・インタフェースと使いやすい独自機能を提供するブラウザ・ユーザー・インタフェース（BUI）が使用されます。または、ネットワーク経由でOracle ZFS Storage Applianceにアクセスできない場合や、一定のコマンド・セットを繰り返して実行する必要がある場合は、コマンドライン・インタフェース（CLI）を使用すると便利です。CLIには、シリアル・コンソールかセキュア・シェル（SSH）を使用してアクセスできます。

コマンドラインでシステムとやり取りすると、一種のバッチ・ジョブのように、いくつかのタスクを決まった順序で繰り返し実行できます。条件付きで実行されるコードがバッチ機能に必要な場合は、スクリプトを使用します。CLIコマンドは対話形式でCLIに入力するか、またはファイルに保存してCLIに渡すことができます。

Oracle ZFS Storage Applianceのスクリプト言語は、JavaScript（ECMAScriptバージョン3）をベースに、いくつかの拡張機能が追加されています。JavaScriptは型指定の弱い(loosely typed)インタープリタ型のプログラミング言語であり、オブジェクト指向機能を提供します。スクリプト・インタープリタはOracle ZFS Storage Applianceシェルに含まれているため、シェルがスクリプトを解釈して実行します。

ワークフローは、Oracle ZFS Storage Applianceにスクリプトを保存するために使用され、またスクリプトに対するユーザー・アクセスの管理と引数の検証機能を提供します。ワークフローにはスクリプトとバージョニングの情報が含まれ、BUIまたはCLIから起動されます。また、アラート・イベントやタイマー・イベントを使用して起動することもできます。タイマー・イベントはワークフローのスケジュールから作成されます。

このホワイト・ペーパーではおもに、Oracle ZFS Storage Appliance内でJavaScriptのプログラミング機能を使用する方法について詳しく説明します。また、Oracle ZFS Storage Applianceのスクリプト・アーキテクチャ、Oracle ZFS Storage Applianceに対するJavaScriptインタフェース、スクリプトの実行メソッドについても詳述します。Oracle ZFS Storage Appliance向けのJavaScript言語またはCLIコマンド言語のチュートリアルを提供することは、本書の対象外です。C、C++、またはPerlやPythonなどのプログラミング言語を使い慣れていると、本書で提供するJavaScriptの例を理解しやすいでしょう。

付録Aには、JavaScriptについての詳しい情報を含む書籍、オンライン・チュートリアル、ブログ、ドキュメントへのリンクを記載しています。Oracle ZFS Storage Appliance内のオンライン・ヘルプ機能にはBUIからアクセスでき、CLIとBUIの使用法についての詳しい情報を確認できます。また、オラクルの『*Sun ZFS Storage 7000システム管理ガイド*』（本書内では管理ガイドと表記）はこちらからダウンロードできます。

注：Sun ZFS Storage Appliance、Sun ZFS Storage 7000、およびZFS Storage Applianceはすべて、同じOracle ZFS Storage Appliance製品ファミリのことを指しています。引用しているドキュメントや画面のコードの中には、従来の命名規則を使用しているものがあります。

Oracle ZFS Storage Applianceでのスクリプト

CLIを使用してOracle ZFS Storage Applianceとやり取りする方法には、CLIコマンドをまとめてファイルに保存する、CLIコマンドによるスクリプトを作成する、ワークフローを使用する、という3つの種類があります。

CLIコマンドを一定の順序でファイル内にグループ化し、Oracle ZFS Storage Applianceに送信して、SSHを使って実行できます。これはバッチ形式のコマンド処理です。コマンドは、Oracle ZFS Storage Applianceにログインしたユーザーの権限で実行されます。Oracle ZFS Storage Applianceシェルで実行できるコマンドの詳しい説明は、管理ガイドに記載されています。

変数やユーザー定義関数を使用した条件付きコードを実行するなど、もっと柔軟な処理が必要な場合は、CLIコマンドによるスクリプトを作成します。シェルが提供するスクリプト・インタープリタで、JavaScriptプログラミング言語で記述されたユーザー定義スクリプトを実行できます。スクリプトは、Oracle ZFS Storage ApplianceへのSSH接続を介して、またはCLIプロンプトの対話形式で、バッチ・ジョブの場合と同じようにOracle ZFS Storage Applianceに渡されます。

後から実行できるようにスクリプトをカプセル化するワークフローは、Oracle ZFS Storage Appliance内に用意されています。ワークフローはBUIまたはCLIを使用して起動できます。タイマー・イベントやアラート・イベントを使ってワークフローを開始することもできます。Oracle ZFS Storage Applianceにアクセスするには、コンソールまたはSSH接続を使用します。

Oracle ZFS Storage Applianceのスクリプト・アーキテクチャ

Oracle ZFS Storage Applianceシェルは、JavaScriptプログラミング・コードを含むスクリプトの実行機能を提供します。JavaScriptインタープリタはシェルに統合されているため、*アプリケーション組込み環境*として知られています。これは、典型的なWebブラウザによる*クライアント/サーバー型*のJavaScript使用法とは異なります。Document Management（DOM）などのJavaScriptライブラリ関数は、アプリケーション組込み環境では使用できません。JavaScriptドキュメントを参照するときに、Core JavaScript Referenceを使用して、使用できるJavaScriptライブラリ関数についての詳しい情報を確認してください。

組込みJavaScriptインタープリタは実行中にコードを解釈するため、スクリプトはOracle ZFS Storage Applianceシェルのコンテキスト内で実行され、Oracle ZFS Storage Applianceから取得されたすべての情報（およびその操作）は、必要なコマンドをCLIコンテキストに渡す関数によって処理されます。つまり、同じCLIナビゲーション・コマンドを使用して、CLIコンテキスト構造を確認できます。たとえば、以下のCLIコマンドを実行すると、子コンテキストである`net`にナビゲートします。

```
7000ppc1:> configuration net
7000ppc1:configuration net>
```

同じ処理を実行するスクリプト・コマンドは以下のとおりです。

```
run ('configuration net');
```

JavaScriptのコア関数はJavaScriptオブジェクト・ライブラリから使用できます。このライブラリには、配列、文字列、数値オブジェクトに対する計算、正規表現などの、複雑なオブジェクト型を操作する関数（メソッド）が含まれています。

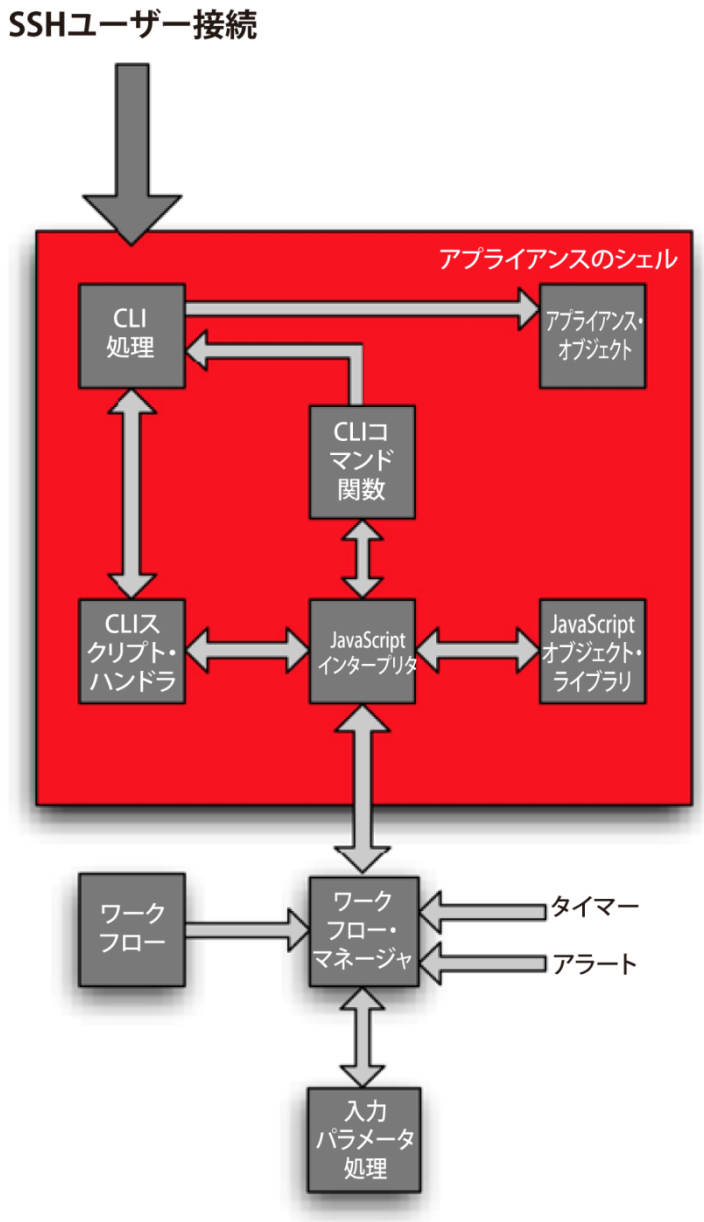

図1：Oracle ZFS Storage Applianceのスクリプト・アーキテクチャ

Oracle ZFS Storage Applianceに保存されたスクリプトは、ワークフロー・マネージャによって制御されます。ワークフロー・マネージャは、CLI/BUIを使用しているユーザーまたはタイマー/アラート・イベントによるスクリプトの開始を制御します。また、ユーザー入力ダイアログ、引数検証処理、ワークフローでのスクリプト実行の開始を管理します。

Oracle ZFS Storage ApplianceのCLIの使用

Oracle ZFS Storage ApplianceのCLIにアクセスするには、SSH接続を使用します。showコマンドを実行すると、CLIのルート・コンテキストや使用できるプロパティ、子コンテンツに関する情報が表示されます。[Tab]キーを押すと、現在のコンテキストで使用できるコマンドの一覧が表示されます。

```
pBrouwers MacBook-Pro:~ pBrouwer$ ssh root@192.168.56.101
Password:
Last login: Mon Feb 14 16:09:05 2011 from 192.168.56.1
7000ppc1:> show
Properties:
                    showcode = false
                   showstack = false
                exitcoverage = false
                 showmessage = true
                  asserterrs = false

Children:
                  Configuration => Perform configuration actions
                    maintenance => Perform maintenance actions
                            raw => Make raw XML-RPC calls
                      analytics => Manage appliance analytics
                         status => View appliance status
                         shares => Manage shares

7000ppc1:>
analytics        coverage    help            script        status
assert           date        ifconfig        set           time
assertlabels     deny        maintenance     shares        traceroute
configuration    exit        nslookup        shell         tree
confirm          get         ping            show
cover            getent      raw             sleep
7000ppc1:>
```

scriptコマンドを実行するとJavaScriptインタープリタが開き、スクリプト文を入力できます。実行するには、ピリオド（.）を入力して[Enter]キーを押します。

```
pBrouwers MacBook-Pro:~ pBrouwer$ ssh root@192.168.0.140
Password:
Last login: Mon Feb 14 16:09:05 2011 from 192.168.56.1
7000ppc1:> script
("." to run)> var i=10 ;
("." to run)> printf("i = %d¥n",i);
("." to run)> .
```

```
i = 10
7000ppc1:>
```

スクリプト文はファイルに保存できます。その後でファイルのコンテンツを使用し、SSH接続を介してOracle ZFS Storage Applianceにスクリプト文を送信できます（付録Bを参照）。
Oracle ZFS Storage Applianceコンテキスト構造の階層間をナビゲートするには、現在のコンテキストから始めて、その上に存在するすべての子コンテキストを頭に付けたナビゲート先のコンテキスト名を指定します。doneコマンドを実行すると、その前のコンテキスト環境に戻ります。UNIXのcdコマンドを実行すると上位階層に移動し、cd /を実行するとルート・コンテキストに戻ります。

同じ処理を実行するスクリプト・プログラム文は以下のとおりです。

```
    run('cd/');
```

コマンドの後ろに子コンテキストへのパスを指定すると、特定の子コンテキストで実行できるコマンドを現在のコンテキストから直接実行できます。

```
7000ppc1:> configuration net interfaces list
INTERFACE   STATE   CLASS   LINKS       ADDRS                   LABEL
e1000g0     up      ip      e1000g0     192.168.56.101/24       i_e1000g0
e1000g1     up      ip      e1000g1     192.168.0.147/24        i_e1000g1
7000ppc1:>
```

Oracle ZFS Storage ApplianceのCLIレイヤーへのアクセス

Oracle ZFS Storage Applianceからステータス情報を取得して、その中の構成情報を操作する手段がないと、スクリプトは使いものになりません。表1に、Oracle ZFS Storage Applianceとのやり取りを可能にするために使用できるJavaScript関数の拡張を示します。

表1：JavaScript関数の拡張

関数	説明
run	指定されたコマンドをシェル内で実行し、出力を文字列として返します。出力に複数の行が含まれる場合、返される文字列は改行されます
props	現在のコンテキストに対するプロパティ名の配列を返します
get	指定されたプロパティの値を返します。この関数は値をネイティブ形式で返します。たとえば、日付は Date オブジェクトとして返されます
set	2 つの文字列引数を受け取り、指定されたプロパティを指定された値に設定します
list	現在のコンテキストの動的な子コンテキストに相当するトークンの配列を返します
choice	choice 関数は、既知で列挙可能である値をもつプロパティに対して、有効なプロパティ値の配列を返します

上記コマンドは、Oracle ZFS Storage ApplianceのCLIコンテキスト構造内で実行されます。詳しくは、管理ガイドの第1章にある"CLIスクリプト"を参照してください。

スクリプト・プログラミング言語

Oracle ZFS Storage Applianceのスクリプト機能は、CLIレイヤー内に用意されたJavaScript言語インタープリタによって実装されています。サポートされるJavaScript構文はECMA-3標準に基づいており、いくつかの拡張が追加されています。

JavaScriptはその名前に反してJavaとは無関係です。JavaScriptはオブジェクト指向のプログラム言語であり、強い型指定チェックを持たないという点においてC++やJavaとは異なります。変数の型はコンパイル時ではなく実行時に定義されます。つまり、変数型は動的型付けです。

JavaScriptという名前の*スクリプト*という部分から、JavaScriptは単純な手続き型プログラミング言語だと思われるかもしれませんが、実際にはオブジェクト指向の豊富な機能セットを組み込んでいます。C++などで使用されているオブジェクト指向のプログラミング概念に慣れている場合は、JavaScriptに隠された真の能力を理解しやすいでしょう。

単純なタスクであれば、従来からある手続き型のプログラミング機能で十分です。タスクがより複雑になると、プロパティとメソッドを含むオブジェクトを使用してオブジェクトのプロパティを操作するなどの、JavaScriptのオブジェクト指向機能が必要になります。より複雑なJavaScriptコードを扱うには、変数のスコープ・メカニズムやオブジェクトと配列の取扱いなど、JavaScriptのオブジェクト指向の側面を理解することが不可欠です。

本書では、より複雑なタスクにJavaScriptを使用するときに知っておくべき、いくつかのJavaScriptの概念に関する基本情報を提供します。JavaScriptに関する詳しい情報は、付録Aに記載する参考資料を参照してください。

Oracle ZFS Storage Applianceで使用するJavaScript環境は、各種の参考資料でアプリケーション組込み環境と呼ばれています。ただし、クライアントサイドJavaScriptの使用について説明した資料は、Oracle ZFS Storage Applianceでの使用に適用できません。コアJavaScriptに関する項のみを参照してください。

構文

JavaScript言語の構文はC言語の構文に似ています。大文字と小文字を区別する必要があるため、変数名と関数名には一貫性が必要になります。文の最後にはセミコロンを使用し、一連の文をグループ化するには中括弧（{}）を使用します。JavaScriptではセミコロンを省略できるケースがありますが、コードの読みやすさを維持するために、常に文の最後にセミコロンを付けることをベスト・プラクティスとして推奨します。また、CおよびC++のコメント構文が認識されます。

データ型と変数

JavaScriptでは、数値、ブール、文字列などのプリミティブなデータ型に加えて、複雑なデータ型がオブジェクトという形式でサポートされます。プリミティブ・データ型の値は、処理に必要な型に自動変換されます。

```
var index=1;
var textarray = new Array('Line 1','Line 2','Line 3');
var array_elements = textarray.length
for ( ; index < array_elements   ; index++) {
printf('Array element' + index + ': ' + textarray[index] + '¥n');
}
```

関数は特殊なオブジェクト型の1つです。つまり、変数や配列、オブジェクトの内部に関数を保存で

きます。また、関数を引数として別の関数に渡すことができます。

その他の言語と同様に、JavaScriptでは、変数に有効な値が入っていないことを示すためにnull値が使用されます。JavaScript自身がnull値を設定することはないため、プログラマーが明示的にこれを設定する必要があります。宣言されただけで値が割り当てられていない変数を識別するためには、undefinedという値が使用されます。この場合、変数の型はまだ決まっていません。また、undefinedという値は、まだ存在しないオブジェクト・プロパティへの参照を識別する目的でも使用されます。

JavaScriptの変数には、値または参照でアクセスできます。文字列は特殊なケースです。JavaScriptでは文字列の内容は不変であり、変更することはできません。文字列を変更するには、新しい文字列を作成して、元の文字列から変更しない部分をコピーする必要があります。

変数の宣言には、常にvar文を使用します。コード内で宣言された時点で、変数のスコープが決まるためです。var文を使用しない場合、変数は自動的にグローバル変数になり、予期せぬ悪影響を招く可能性があります。たとえば、JavaScriptのガベージ・コレクション・メカニズムはグローバル変数を破棄しないため、メモリ・リークにつながる場合があります。

JavaScriptのプロパティと変数

JavaScript言語は変数とプロパティの両方をサポートしています。一見すると、変数とプロパティは同じように見え、どちらも値を保持します。違いは、JavaScriptのインタープリタが実行時にこれらを作成する方法にあります。基本的に、プロパティはオブジェクトに属し、変数はコンテキストに属します。標準的なユーザーにとって実質的な違いはありません。

JavaScript言語では、プロパティはオブジェクトに関する変数の情報を追加するために使用されます（例：traffic-light.current-color=red）。関数もオブジェクトとして扱われるため、関数内で定義された変数はこの関数のプロパティと見なされます。

重要なのは、プロパティに値を割り当てる各種のメソッドと構文について理解することです。メソッドには値渡しと参照渡しの2種類があり、それぞれのメソッドに対して複数の異なる構文が使用できます。読みやすく、メンテナンスしやすいコードにするために、関数、オブジェクト、テキスト文字列に対する参照を含む変数に対しては"参照渡し"を使用します。

テキスト文字列の定義はプログラムの最初に記述します。こうすることで、コードのメンテナンス中に簡単に見つけることができます。

以下の例では、ワークフロー構造のプロパティの値がコードの最初で定義されています。ワークフロー構造は、後でプログラムの最後で宣言する必要があります。この例では、"参照渡し"と"値渡し"の両方のメソッドに対する構文が示されています。

例1：異なるメソッドを使用した、workflowオブジェクトのプロパティの初期化

```
/// File: Example 1
// Object initialized with two properties, using the object literal syntax,
// in which each property name/value pair is followed by a comma.
// The property name is followed by colon.
var MyWorkflow = {
       MyVersion:          '1.0',
       MyName:             'Example 1',
       MyDescription:      'Example showing basic Object Workflow structure'
```

```
}

// New property added using variable assignment syntax.
MyWorkflow.MyDescription =       'Use of properties example';

// Workflow object initialized using literal syntax and references.
// Values do not need to be literals; they can be references to other
objects,
// as can be seen here.
var workflow = {
       name:          MyWorkflow.MyName,          // Reference syntax
       description:   MyWorkflow.MyDescription,   // Reference syntax
       version:       MyWorkflow.MyVersion,             // Reference syntax
       origin:        'Oracle',                         // Literal syntax
       execute:       function () { return('Hello World'); }// Literal Syntax
```

JavaScriptの演算子

JavaScriptは、その他の言語で使用されているおなじみの演算子を使用します。その他に、「===」および「!==」という新しいタイプの演算子が"同一性"をチェックするために使用されます。これらの演算子は「==」および「!=」演算子とは違い、自動的なキャスト変換を実行しません。違いがわかりにくいことがあるため、これらの演算子の細かい部分に注意してください。

```
var num = 10;
var num_string= '10';

if ( num==num_string )
    print('True because values are the same¥n');
if ( num===num_string)
    print('Variables are of identical type and have the same value¥n');
else
    print('Variables do not have same value OR are not of identical type¥n');
```

例1のコードは以下の出力を生成します。

```
True because values are the same
Variables do not have same value OR are not of identical type
```

すべての算術演算子は代入演算子と組み合わせて使用できます（例：「*=」、「+=」、「%=」）。これらの文では、「*a*演算子 *= b*」が「*a = a*演算子*b*」という意味になります。

JavaScriptの文

UNIXシェルやCまたはC++のプログラミングに精通している場合、JavaScriptでのプログラム作成は簡単です。これらJavascript以外の言語で使い慣れた、`if then`、`while`、`for`、`switch case`などの文はすべて、JavaScript言語で使用できます。

プログラムのベスト・プラクティスとして、プログラムの実行中に発生する可能性のある実行時例外を捕捉することを推奨します。例外イベントは、関数が失敗した場合や、コマンド`run ('select myshare')`で存在しない共有を選択した場合などにトリガーされます。`throw`文は明示的に例外信号(exception signal)を発生させます。

実行時例外の捕捉と例外からのリカバリは、JavaScript式の`try/catch/finally`文で処理されます。

以下に示す単純なクラスタ構成のテスト・スクリプトは、現在のOracle ZFS Storage Applianceノードがクラスタノードとなっているかどうかをチェックします。

例2：単純なクラスタ構成のテスト・スクリプト

```
7000ppc1:> script
("." to run)> function ClusterTest(){
("." to run)>     try {
("." to run)>             run ('cd /');
("." to run)>             run ('configuration cluster');
("." to run)>             return(true);
("." to run)>             }
("." to run)>     catch (err) {
("." to run)>             if (err=EAKSH_BADCMD) {
("." to run)>                     return(false);
("." to run)>             }
("." to run)>             else {    // catch unknown condition
("." to run)>                     throw("Unexpected cluster test error");
("." to run)>             }
("." to run)>     }
("." to run)> }
("." to run)> // Main start
("." to run)> printf("Let's see if this node is part of a cluster; ");
("." to run)> if ( ClusterTest() )
("." to run)>     printf("Yes it is¥n");
("." to run)> else
("." to run)>     printf("No it is not¥n");
("." to run)> .
Let's see if this node is part of a cluster; No it is not
```

関数ClusterTestは、try/catch構成を使用してrunコマンドを実行し、子コンテキストclusterにナビゲートしています。runコマンドが失敗するとcatch文によって捕捉されるため、ここで、runコマンドが失敗したかどうかをチェックします。これ以外の予想外のエラー状態が発生した場合、throw文を使用してプログラムを終了します。

Oracle ZFS Storage Applianceでのスクリプトの実行

以下のコードは、Oracle ZFS Storage Applianceにスクリプトをロードして実行するためのコマンドを示しています。スクリプトは、Oracle ZFS Storage Applianceシェル内のスクリプト・インタープリタに渡されることに注意してください。

```
pBrouwers MacBook-Pro:~ pBrouwer$ ssh root@192.168.0.140
Password:
Last login: Mon Feb 14 16:09:05 2011 from 192.168.56.1
7000ppc1:> script
("." to run)> var i=10 ;
("." to run)> printf("i = %d¥n",i);
("." to run)> .
i = 10
7000ppc1:>
```

上記の例では、コマンドscriptを使用してスクリプト・インタープリタを有効にした後に、スクリプト文が入力されています。ピリオド（.）を入力して［Enter］キーを押した後で、スクリプト文が実行されています。この方法はシンプルな対話形式の用途には十分です。

より複雑なスクリプトを扱う場合は、テキスト・ファイルにコマンドをグループ化し、SSH接続を使用してOracle ZFS Storage Applianceに送信する方が簡単です。この環境は完全なJavaScript機能を提供しており、関数や条件付きの文などを使用できます。

以下のスクリプト例は簡易バージョンのスクリプトを使用して、既存のプールとプロジェクト内に共有(share)を作成して削除しています。

例3：単純なスクリプトによる共有の作成と削除

```
// File: Example3.txt script
// For ease of use, group all our arguments in one object.
// One could even use a shell script to create this JavaScript from a
template
// using the arguments passed on to the user in a shell script.
// Version:  1.1.3
var MyArguments = {
       pool:         'poola',
       project:      'projecta',
       share:        'volumeTest',
       what:         'delete'
}
function CreateDeleteShare (Arg) {
       run('cd /');          // Make sure we are at root child context level
       run('shares');
       try {
```

```
            run('set pool=' + Arg.pool);
      } catch (err) {
            printf("Specified pool, %s not found¥n ",Arg.pool);
            return;
      }
      try    {
             run('select ' + Arg.project);
      } catch (err) {
            printf("Specified project, %s not found¥n ",Arg.project);
            return;
      }
      if (Arg.what=='create' ) {
            try {
                  run('filesystem ' + Arg.share);
                  run('commit');
            } catch(err) {
                  printf("Unable to create share, %s¥n",Arg.share);
                  return;
            }
            printf('Successfully created share, '+Arg.share);
            return;
      } else {
            try {
                  run('select' + Arg.share);          // Check if share is
there
                  run('done');                        // Release for delete
                  run('confirm destroy ' + Arg.share);
            } catch (err) {
                  if (err.code == 10004 )
                        printf("Specified share, %s, does not

exist¥n",Arg.share);
                  else   printf("Unable to delete share, %s¥n",Arg.share);
                  return;
            }
            printf("Successfully deleted share, %s¥n", Arg.share);
      }
}

// Kick off the create delete function using our object MyArguments to pass
on the
// parameters needed for the job.
printf("About to %s share, %s from project, %s in pool %s¥n",
             MyArguments.what,
             MyArguments.share,
```

```
                MyArguments.project,
                MyArguments.pool);
CreateDeleteShare(MyArguments);
```

このスクリプトでは、使用するプール、プロジェクト、共有(share)の名前を格納するために、オブジェクトMyArgumentsを使用しています。変数をスクリプト全体でハードコードする代わりに、スクリプトの最初で、この構造体にすべての変数要素が格納されます。実行時エラーはtry/catchを使用して処理されます。本スクリプトでは単純化のため、発生したエラーの種類は区別されません。変数err.codeを使用すると、runコマンドを失敗させたエラーの種類を確認できます。

この方法で使用されたスクリプトは、事前定義されたタスクを実行する必要のあるバッチ型の環境に最適です。スクリプトの使用をより厳密に制御する必要がある場合には、ワークフローが適しています。ワークフローで使用するスクリプトはOracle ZFS Storage Appliance上に格納されますが、コードがいったんロードされると、コードの中身はエンドユーザーから見えなくなるため、スクリプトを変更することはできません。

また、ワークフローにはアクセス制御を適用できるため、管理タスク向けに作成されたワークフローなどの使用を制限することができます。ワークフローではプロンプトを通じてユーザーに入力を要求できます（次項を参照）。

ワークフローを使う場合と比較すると、SSHを使ったスクリプトは、ユーザー入力のプロンプトと、Oracle ZFS Storage Applianceのタイマー・イベントやアラート・イベントによるトリガーの起動以外の制約はありません。

Oracle ZFS Storage Applianceワークフローの使用

前の例では、SSH接続経由またはコンソールCLIから対話形式で、Oracle ZFS Storage Applianceにスクリプトをロードしました。Oracle ZFS Storage Applianceに永続的にスクリプトを格納するには、ワークフロー・マネージャの制御下にスクリプトを配置する必要があります。ワークフローは、システムのログインに使用されたユーザーの資格証明を使って、Oracle ZFS Storage Applianceシェル内で非同期に実行されます。

ワークフロー・マネージャにスクリプトを格納して実行するには、スクリプトには追加情報が必要になります。ワークフロー・マネージャはこの追加情報を使用してユーザーに情報を提示し、ワークフローの開始関数にアクセスします。この情報は`workflow`というオブジェクトに格納されます。このオブジェクトのプロパティを表2に示します。

表2：オブジェクトworkflowのプロパティ・メンバー

必須プロパティ	JavaScript タイプ	説明
`name`	文字列	ワークフローの名前
`description`	文字列	ワークフローの簡単な説明
`execute`	関数	実行するスクリプト・コード
オプション・プロパティ		
`version`	文字列	このワークフローのバージョン、ドット付きの 10 進形式（メジャー.マイナー.マイクロ）
`required`	文字列	このワークフローを実行するために必要な Oracle ZFS Storage Appliance ソフトウェアの最小バージョン
`origin`	文字列	ワークフロー・プロバイダの名前
`parameters`	オブジェクト	スクリプト入力パラメータを定義する構造体
`validate`	関数	入力パラメータを検証する JavaScript 関数

オブジェクト`workflow`の作成は、JavaScriptオブジェクトのリテラル構文（コロンで句切られたプロパティ/値のペアをカンマで区切ってリストにし、最後に中括弧で囲む）に従います。

以下のコードで、`<property>`には表2に記載したいずれかのプロパティの名前が入ります。

```
var workflow = {
       <property>:   <object literal>|<object reference>,
       <property>:   <object literal>|<object reference>,
};
```

<object literal>に入る値のタイプは、表2に示した文字列、関数、オブジェクトのいずれかです。

必要最小限のワークフロー・スクリプトは以下のようになります。

```
// Example of basic workflow definition using object literals
// in the workflow object constructor.
var workflow = {
       name:          'Minimum workflow code',
       description:   'Example of basic workflow structure',
       execute:       function() { return('Hello World'); }
};
```

オブジェクトworkflow内でプロパティにデータ値を直接指定する代わりに、先に定義したオブジェクトの参照または変数を使用できます。以下の例に最小限のワークフロー定義を示します。

例4：最小限のワークフローを定義するコード

```
// File: Example4.txt
// Example of basic workflow definition using variables
// in the workflow object's constructor
//
//
var WorkflowName = 'Minimum workflow code';
var WorkflowDescription = 'Example of basic workflow structure';

function Main () {
       return('Hello World');
}
var workflow = {
       name:        WorkflowName,
       description: WorkflowDescription,
       execute:     Main
};
```

上記のワークフロー例をOracle ZFS Storage Applianceにロードするには、Oracle ZFS Storage Applianceの子コンテキスト"maintenance workflow"で、BUIまたはCLIのuploadコマンドを使用します。

以下のスクリーンショットに、BUIワークフローのロード手順を示します。

図2：Oracle ZFS Storage Appliance BUIのワークフロー

「+」ボタンをクリックすると、ワークフロー・ファイルをロードするためのダイアログ・ボックスが表示されます。

図3：BUIのAdd Workflowウィンドウ

ロード・プロセス中に、ワークフロー・ファイル内の構文エラーがチェックされます。

ワークフローがロードされると、`workflow`オブジェクトのnameプロパティとdescriptionプロパティに設定された名前および説明情報が表示されます。

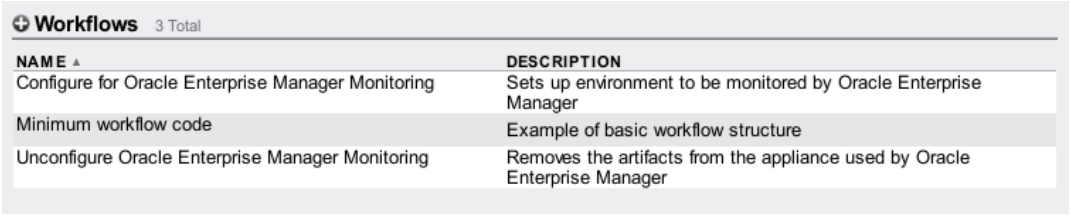

図4：新しく追加されたワークフローを表示したBUI

ワークフローを実行するには、ワークフローをダブルクリックします。

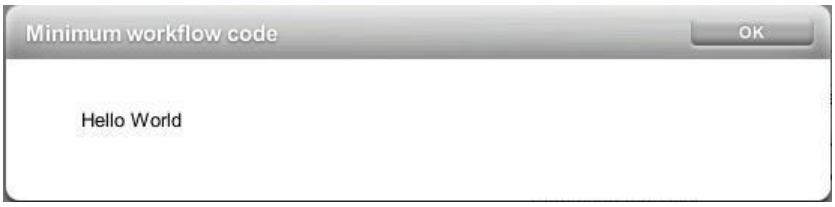

図5：新しいワークフローの出力を表示したBUI

ワークフロー・プロパティparametersを使用して、ワークフローに入力パラメータを追加すると、ワークフローの利便性が良くなります。

```
parameters = {
       <parameter1name>: {
              label:         <String>
              type:          <String>
       }
<parameter2name>: {
              label:         <String>
              type:          <String>
              options:       <Array>
              optionlabels:  <Array>
       }
<parameterNname>: {
              label:         <String>
              type:          <String>
              optional:      <Boolean>
       }
};
```

上記は、それ自体が以下の構造を持つオブジェクトになっています。

- プロパティparameterNnameは、これを使ってスクリプト・コード内で参照できる入力パラメータの名前です。
- プロパティparameterNnameはこれ自体がオブジェクトであり、常にlabelプロパティとtypeプロパティを含む必要があります。
- プロパティlabelの値は、例5に示すコード内で使用されます。
- プロパティtypeの値は、paramenterNnameオブジェクトに格納された値のタイプ（ブール、文字列、ファイルなど）を指定します。

注：type定義の完全なリストは、Oracle ZFS Storage Appliance管理者ガイドを参照してください。

以下のプロパティは必須ではありませんが、パラメータに対する要件を指定するために使用できます。

- プロパティoptionalにtrueを設定すると、このパラメータに対するUIでの入力が必須ではなくなります。
- プロパティ・ペアoptions-optionlabelsは、不変の入力値のリストです。この機能を使うには、プロパティtypeの値をChooseOneに設定する必要があります。

次に、ワークフローの開始時に、ワークフロー・マネージャがどのようにオブジェクト内のプロパティを使用するかについて確認しましょう。この例ではBUIインタフェースを使用しています。対話形式のCLIについては、管理ガイドを参照してください。ワークフロー・マネージャが使用する処理メカニズムは、BUIとCLIのどちらでも同じです。

オブジェクトparametersを使用して構築されるダイアログ・ボックスにはフィールドが含まれており、ここに、オブジェクトparametersが要求する入力値を入力します。フィールドに入力して「PROCEED」をクリックすると、ワークフロー・マネージャは、プロパティvalidateに指定されたとおりに検証関数を実行し、オブジェクトparametersへの参照を引数として使用します。validate関数によってエラーが発生した場合、ワークフロー・マネージャはダイアログ・ボックスの元の状態に戻り、エラーが検出されたフィールドを表示します。

validate関数が問題なく終了すると、ワークフロー・マネージャは、ワークフロー・プロパティexecuteに指定された関数を呼び出し、この時もオブジェクトparametersへの参照を引数として使用します。

例5は、共有(share)の作成/削除スクリプトを例に取り、ワークフローでの使用向けに変更したものです。ワークフロー・マネージャは、共有の作成/削除操作を実行する対象となるプール名とプロジェクト名の入力を要求します。作成または削除の選択には、プルダウン・リスト構成が使用されます。

例5：基本的なワークフロー・スクリプトによる共有の作成と削除

```
// Simple example of how to create/delete a share.
// File example5.txt
// Information to be used in workflow object:
var MyWorkflow = {
       MyVersion:           '1.0.0',
       MyName:              'Create/Delete a share',
       MyDescription:       'Example of how to create/delete a share',
       Origin:              'Oracle Corporation',
       err: {               // Definition of error codes.
                            // Define a range for your project that
                            // can be recognized by the sysadmins in
                            // your org.
       WP_SCRIPT_WORKFLOWS_CREATE_SHARE: 8001,
       WP_SCRIPT_WORKFLOWS_DELETE_SHARE: 8002,
       }
}

// This example workflow uses four input parameters.
// The last parameter is an example of the use of a fixed list of values.
var MyParams = {
       pool: {              // Pool to create/delete the share.
              label:        'Pool Name',
              type:         'String',
       },
              project: {    // Project to create/delete the share.
              label:        'Project name',
              type:         'String',
       },
       share: {             // Share to create/delete.
              label:        'Share name',
```

```
		type:		'String',
	},

	what: {			// Create or delete the share.
		label:		'Operation',
		type:		'ChooseOne',
		options:	['create','delete'],
		optionlabels:	['Create','Delete'],
	}
}

// Verify function from workflow.
// Check whether pool and project exist.
	function VerifyPoolandProject(p) {
		var err_msg = ' does not exist';
	run ('cd /');		// Make sure we are at root child context level.
	run ('shares');
	try {				// Check whether pool name exists.
		run('set pool='+p.pool);
	} catch(err) {
		return( {pool: 'Specified pool, ' + p.pool + err_msg } );
	}
	try {
		run('select ' + p.project);

	} catch(err) {
		return( {project: 'Specified project, ' + p.project + err_msg }
);
	}

	if (p.what=='delete' ) {   // Check whether the share to be deleted
exists.
		try {
			run('select'+ p.share);
		}
		catch(err) {
			return({share: 'Specified share, ' + p.share + err_msg
});
		}
	}
	return;
}
function CreateDeleteShare (p) {
	run('cd /');		// Make sure we are at root child context level.
	run('shares');
```

```
	run('set pool=' + p.pool);
	run('select ' + p.project);
	if (p.what=='create' ) {
		try {
			run('filesystem ' + p.share);
			run('commit');
		} catch(err) {
			throw {
				code:
MyWorkflow.err.WP_SCRIPT_WORKFLOWS_CREATE_SHARE,
				message: 'Unable to create ' +
					p.share + ',' + err.message
			}
		}
		return('Successfully created share, '+p.share);
	} else {
		try {
			run('confirm destroy ' + p.share);
		} catch(err) {
			throw {
				code:
MyWorkflow.err.WP_SCRIPT_WORKFLOWS_DELETE_SHARE,
				message: 'Unable to delete ' +
					p.share + ',' + err.message
			}
		}


		return('Successfully deleted share, '+p.share);
	}
}

var workflow = {
	name:		MyWorkflow.MyName,
	description:	MyWorkflow.MyDescription,
	version:	MyWorkflow.MyVersion,
	origin:		MyWorkflow.Origin,
	parameters:	MyParams,
	validate:	VerifyPoolandProject,
	execute:	CreateDeleteShare
};
```

コードの最後にオブジェクトworkflowが指定されているのは、このオブジェクト内で参照されているすべてのオブジェクトと関数を先に定義する必要があるためです。読みやすく、管理しやすいコードにするため、すべてのスクリプト情報はオブジェクト内に格納し、コードの最初に指定しています。このオブジェクトの要素に対する参照は、後からオブジェクトworkflow内で使用されています。ワークフロー内のプロパティvalidateおよびexecuteに対しても、同じことが当てはまります。これにより、ワークフロー・スクリプトが読みやすくなります。

オブジェクトMyParamsには、ユーザーがこのワークフローに対して入力する必要のある情報が指定されています。要求された4つのパラメータのうち、3つのタイプはテキストであり、1つは2つの項目を含むリストです（図6を参照）。オブジェクトMyParamsへの参照が、オブジェクトworkflow内のプロパティparametersに格納されます。

図6：ユーザー・ダイアログ・ボックスの例

BUIでワークフローを実行すると、図7に示すダイアログ・ボックスが表示されます。

この例で、volumeqqqは存在しない共有です。「APPLY」をクリックすると、オブジェクトworkflow内の検証関数ValidatePoolandProjectで共有選択エラーが発生します（図7）。

図7：ユーザー・ダイアログ・ボックスの入力エラー例

このエラーは、次の文から出力された結果です。

```
return({share: 'Specified share, ' + p.share + err_msg });
```

return文のshare:には、エラー・フラグの付いた入力パラメータのプロパティ名に対する参照が含まれています。その結果として、ワークフロー・マネージャは、エラーを引き起こした入力フィールドの値をハイライト表示し、オブジェクトMyParams内のフィールド定義のlabelプロパティの値とエラー・メッセージを表示します。

エラーは、JavaScriptのtry/catch機能を使用して捕捉されます。

アラート・ワークフロー

ワークフローを使うと、Oracle ZFS Storage Applianceが生成したアラートに対応するカスタム関数またはアクションを実装できます。Oracle ZFS Storage Applianceはさまざまな状況に対してアラートを生成できます。アラート・メッセージは、「Maintenance」→「Logs」→「Alerts」内に保存されます。アラート・アクションを定義すると、アラートにワークフローをバインドすることができます。アラート・アクションによって起動されたワークフローはバックグラウンドで実行され、ユーザー入力は要求しません。管理ガイドの「構成」の項に、アラートとそのカスタマイズ方法、アラート・ログのある場所に関する詳しい情報が記載されています。

ワークフローとイベントに関連付けるには、オブジェクトworkflowに新しいプロパティを追加する必要があります。ワークフロー・ファイルの所有者権限（ownersロールに設定されたもの）でワークフローを実行可能にするには、alertプロパティとsetidプロパティの両方をtrueに設定する必要があります。ユーザー入力は必要ないため、入力パラメータを表すオブジェクトは必要ありません。

例6に示したコードは、非常に単純なアラート・アクション・ワークフローを提供するために前の例を変更したものです。

例6：アラート・ワークフローを実行するための最小限のコード

```
// File: Example 6
// Example 5 adapted for alert usage
var MyWorkflow = {
       MyVersion:      '1.0',
       MyName:         'Example 6',
       MyDescription:  'Example of use of Alert',
       Origin:         'Oracle'
};

var workflow = {
       name:           MyWorkflow.MyName,
       description:    MyWorkflow.MyDescription,
       version:        MyWorkflow.MyVersion,
       alert:          true,                      // Workflow triggered by
       alert setid:    true,
       origin:         MyWorkflow.Origin,
       execute:        function (MyAlert) {
                           audit('workflow started for alert'+MyAlert.uuid);
                             }
};
```

ワークフローを使用して外部に情報を送信するには、関数auditを使用する必要があります。この

関数は1つの文字列を引数として受け取り、Oracle ZFS Storage Applianceの監査ログにこのテキストを書き込みます。ワークフロー・マネージャは、プロパティexecuteに対して定義された関数にオブジェクトを渡します。このオブジェクトに含まれる要素を表3に示します。

表3：executeプロパティに含まれる要素

プロパティ	タイプ	説明
class	文字列	アラートのクラス
code	文字列	アラートのコード
uuid	文字列	アラートに対する一意の識別子
timestamp	日付	イベントの時刻
items	オブジェクト	イベントに関する詳しい情報を含むオブジェクト

プロパティitemsは、ワークフローを起動したイベントに関する以下の詳細情報を含むオブジェクトです。

表4：イベント情報

プロパティ	タイプ	説明
url	文字列	イベントの説明を含む Web ページの URL
action	文字列	イベントに応じてユーザーが取るべきアクション
impact	文字列	アラートを引き起こしたイベントの影響
description	文字列	判読可能な文字列を使ったアラートの説明
severity	文字列	アラートを引き起こしたイベントの重大度
response	文字列	自動化された応答アクションの情報
type	文字列	アラートの種類（重要度が高い、低いなど）

BUIでアラートに関する詳細情報を要求した場合もこの情報が表示されます（図8）。

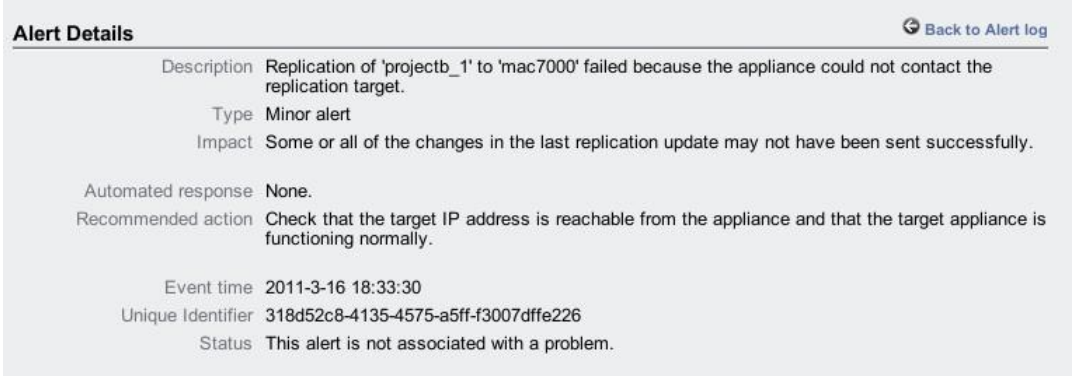

図8：Oracle ZFS Storage ApplianceのBUIに表示された詳細アラート情報

サンプル・コードを、たとえばOracle ZFS Storage Applianceのレプリケーション・イベントにバ

インドするには、はじめにOracle ZFS Storage Applianceにワークフローをロードする必要があります。図9のスクリーンショットに示したダイアログ・ボックスで、構成コンテキストにアラート・アクションを定義する必要があります。アラート・アクションは1つのイベント・カテゴリにバインドします。各カテゴリに対して、サブセットのイベント・タイプを選択できます。

アラート・アクションは複数設定できます。「Execute workflow」を選ぶと、以前にロードされたサンプル・スクリプトを実行させることができます。

この例では、レプリケーション・アクションが失敗した場合に、電子メールの送信とワークフローの開始という2つのアクションを実行するように設定されています。TESTオプションを使用すると、設定したアクションを手動で起動できます。

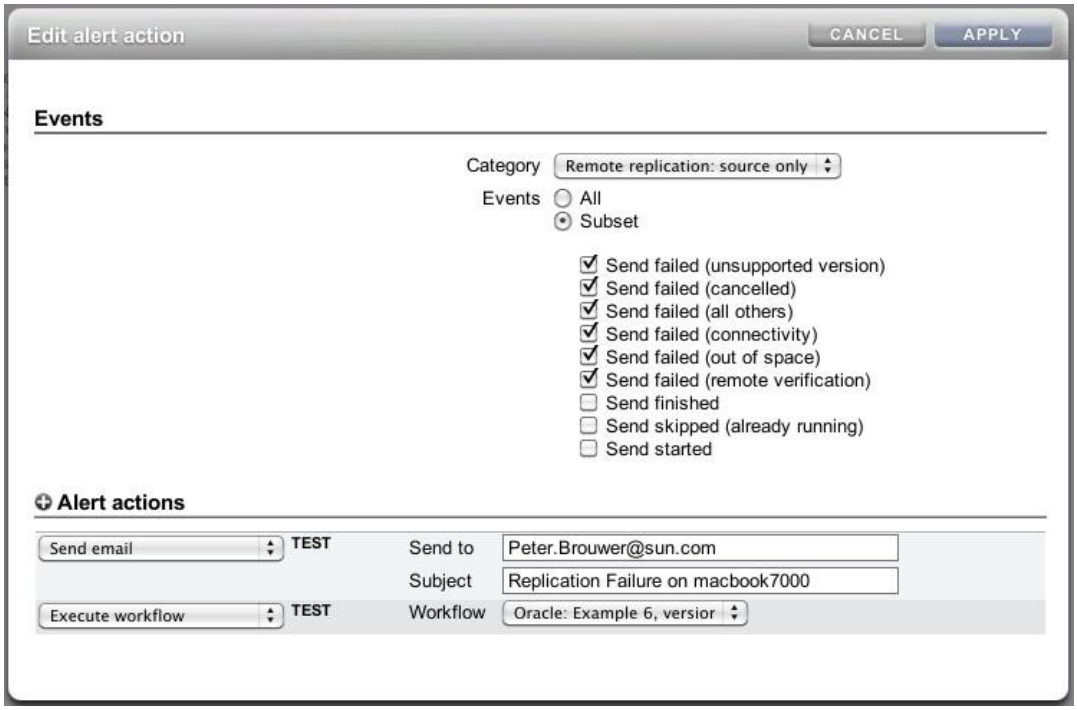

図9：Oracle ZFS Storage ApplianceのBUIでのアラート・アクションの追加

ワークフローのスケジュール設定

ワークフローのスケジュールを設定すると、ワークフローの開始を管理できます。特定の時間にワークフローを起動するよう、ワークフロー・マネージャがスケジュールからタイマー・イベントを作成します。

CLIを使って、既存のワークフローに対するスケジュールを作成できます。または、表5に示す構造を使用して、1つ以上のエントリを含む配列タイプのscheduleオブジェクトを定義することで、ワークフローにスケジュールを組み込むことができます。

表5：ワークフローのscheduleオブジェクトのプロパティ

プロパティ	タイプ	説明
offset	数値	定義された期間内で開始点を決定します。出発点ゼロは木曜日です
period	数値	スケジュールの頻度を定義します
unit	文字列	offset と period で使う単位を seconds または month で指定します

例7：ワークフロー・スケジュールの定義例

```
// File: Example 7
var MyWorkflow = {
        MyVersion:          '1.0',
        MyName:     'Example 7',
        MyDescription: 'Example of use of Schedules',
        Origin:     'Oracle'
};
var MySchedules = [
        // use the inline arithmetic to create readable code
        // half hr interval
        // Starting day seems to be Thursday for the offset.
        { offset: 0, period: 1800, units: "seconds" },
        // every Monday on 10:15
        { offset: 4*24*60*60+10*60*60+15*60, period: 7*24*60*60, units:
"seconds"}
];
var workflow = {
        name:          MyWorkflow.MyName,
        description:   MyWorkflow.MyDescription,
        version:       MyWorkflow.MyVersion,
        origin:        MyWorkflow.Origin,
        alert:         false,
        setid:         true,
        schedules:     MySchedules,
        scheduled:     true,
        execute:       function () {
                          audit ('Example 7: started via scheduled event');
                          }
};
```

Oracle ZFS Storage Applianceにワークフロー・コードをアップロードしたら、CLIを使ってワークフロー・スケジュールを検証できます。

```
Node1:> maintenance workflows
Node1:maintenance workflows> select workflow-007
```

```
Node1:maintenance workflow-007> schedules
Node1:maintenance workflow-007 schedules> show
Schedules:

NAME                    FREQUENCY         DAY                 HH:MM
schedule-000            halfhour          -                   --:00
schedule-001            week              Monday              10:15
```

スクリプトへのベスト・プラクティスの適用

ここからは、スクリプトの開発と実装に対する推奨事項について説明します。

コーディング・スタイルのベスト・プラクティス

JavaScriptのプログラミング構造は、CやC++といったプログラミング言語の構造と似ているため、CやC++に対するコード記述の標準手法がJavaScriptのコードにも適用できます。たとえば、本書でも中括弧やインデントが使用されています。

変数には、意味が分かり、理解しやすい名前を付けます。`i`や`n`、`x`といった変数名は使わないでください。変数名は短く簡潔で分かりやすいものにします。`FC_Lun`や`iSCSI_Lun`といった変数名は、単なる`LUN`という変数名よりも適切です。

プログラム言語を使う場合は英語を使います。こうすることで、ワークフロー・スクリプトの共有や入力の要求が簡単になり、現地語以外を話す同僚からのサポートが得やすくなります。

適切に構造化されたシンプルなコード文を使用し、コメントを加えます。すでにある情報を繰り返すのではなく、新しい情報を追加するコメントにします。適切なコメントがあれば、1年後に更新を行う際も目的の個所を探しやすいでしょう。ワークフローを共有することになっている場合、その用途と目的が組み込まれたコメントから明確に分かる必要があります。

JavaScriptのベスト・プラクティス

以下のベスト・プラクティスを適用します。

- セミコロンを使用します。通常、JavaScriptの各文はセミコロンで終了します。しかし、特定の状況ではセミコロンがなくても許され、JavaScriptがセミコロンを想定して処理します。また、ワークフロー・マネージャはOracle ZFS Storage Applianceへのワークフローのロード中にチェックを行います。曖昧さをなくすため、文の最後には必ずセミコロンを付けます。
- グローバル名前空間が乱雑にならないように注意します。グローバル変数を使用すると、名前が競合したり、スコープ・チェーン内の誤ったレベルで参照が解決されたりするリスクがつきまといます。例5の`MyWorkflow`オブジェクトのように、オブジェクトを使用して変数をカプセル化します。また、オブジェクトを使うと、移植可能でメンテナンスしやすいコードを作成できます。
- 変数は、使用する前に必ず`var`文を使用して定義します。変数が定義されていない場合、JavaScriptは変数がグローバル・スコープで定義されたものと見なします。未定義の（グローバルと見なされている）変数はJavaScriptのガベージ・コレクション・メカニズムに含まれないため、スクリプトのメモリ使用量が増え続ける結果となります。この現象はメモリ・リークと呼ばれています。
- `with`文は使用しないようにします。`with`文はしばしば、ネストの深いオブジェクト階層を扱うときに入力を省略するために使用されます。しかし、JavaScriptが階層チェーンを越えるときの変数解決方法は制御されていません。オブジェクトを`with`文で使用する代わりに、このオブジェクトへの参照を含む変数を使用します。

トレーニングのベスト・プラクティス

推奨されるトレーニングのベスト・プラクティスは以下のとおりです。

- 多くの情報に目を通して学習すること。インターネット上にブログや記事として提供されている、JavaScriptに関する豊富な情報を利用します。ただし、紙の出版物の価値を過小評価しないでください。快適な椅子に座って、一杯のコーヒーとともに読書する時間はかけがえのないものです。
- シミュレーション。Oracle ZFS Storage Appliance Simulatorは、スクリプトおよびワークフロー環境に慣れるのに最適なツールです。このシミュレータはすべてのスクリプト機能とワークフロー機能に対応しています。

クライアント側のアプライアンス制御に関するヒントと例

ここでは、問題を解決する優れた方法の発想に結びつく可能性のある、いくつかの簡単な例を紹介します。記載する例は、間違いのない完全な解決策では決してなく、簡易化のために、エラー・チェックや例外の処理はコード例に含まれていません。

SSHを使ったCLIスクリプトの自動実行

スクリプトを実行するホストでssh- keygen -t rsa -b 1024コマンドを使って生成した公開鍵をOracle ZFS Storage Applianceにインストールすると、SSHを使ってCLIスクリプトを実行するたびにパスワードを要求されなくなります。ssh用のセットアップ・スクリプトは以下の構文を使用して実行できます。<key_rsa>には、ssh-keygenコマンドで生成した鍵を含むファイルの名前を指定します。

```
ssh -i .ssh/<key_rsa> root@MyAppliance
```

UNIXシェルでのCLIスクリプトの使用

SSH接続を使ってアクセスしている場合、連続したコマンドをOracle ZFS Storage Applianceに送信することは紛らわしい場合があります。使用できるオプションを理解するために、基本に戻りましょう。UNIX型のコマンド・インタフェースの場合、SSHにはstdinとstdoutという2つのI/Oメカニズムがあります。基本的に、stdinはSSHへの通信インタフェースであり、stdoutはSSHからの通信インタフェースになります。入力と出力はリダイレクトできます。

```
ssh root@myAppliance < inputfile > outputfile
```

上記コマンドでは、sshに対する入力がリダイレクトされて、inputfileファイルから文字が読み取られます。また、sshからの出力はファイルにリダイレクトされます。このため、以下のようなinputfileがある場合、inputfile内の文字がOracle ZFS Storage ApplianceのCLIに送信されます。

```
configuration net interfaces list
```

コマンド・シーケンスの出力はsshによって取り込まれ、

outputfileにリダイレクトされます。

Oracle ZFS Storage ApplianceのCLIプロンプトでは、対話形式でJavaScriptコマンドを入力して実行できます。つまり、inputfileファイル内にJavaScriptコマンドを記述し、sshを使ってOracle ZFS Storage Applianceに送信し、JavaScriptコマンドからの出力をoutputfileに取り込むことができます。これにより、"インテリジェント"なバッチ・ジョブを作成できるため大きな効果があります。この機能は、batchコマンド環境にインテリジェントで動的なコーディング環境を統合します。

sshのメカニズムを理解したら、sshを使ってOracle ZFS Storage Applianceに送信するスクリプトを記述し、クライアント側でスクリプトの終了ステータスをチェックできます。

たとえば、共有(share)を作成するスクリプトがあり、このスクリプトが失敗したかどうかをチェックする必要があるとします。この場合、エラー・コードをSSH（クライアントのシェル）に返して、クライアント・シェル・コード内でエラー状況に対応できます。

例8は、例3に示した古い共有(share)の作成/削除スクリプトを使用しています。はじめに、テキスト・メッセージを返すすべてのコードを、数値コードを返すコードで置換します。テキスト文字列の処理はシェル内では困難です。次に、シェルの引数を処理し、Oracle ZFS Storage Applianceの

スクリプト部分に変数を渡すために使用されるシェル変数を開始するシェル・コードを追加します。

ファイルに含まれる2つの"EOF"文字列の間にあるテキストは、ssh stdinを使ってOracle ZFS Storage Applianceに送信されるJavaScriptコードの一部です。JavaScript部分からのエラー・コードは、JavaScriptコードの最後にあるprintコマンドを使って返され、シェル・スクリプトのScriptError変数で捕捉されます。

例8：Oracle ZFS Storage Applianceスクリプトに引数を渡すシェル・スクリプト

```
#!/bin/sh
# File example8.txt
# Shell script using input arguments to be passed to appliance script job.
# Script error codes are passed back to the shell.
Usage() {
    echo "$1 -u <Appliance user> -h <appliance> -s <share> -c|-d -j <project>
-p <pool>"
    exit 1
}
# Error code definitions
PoolNotFound=100
ProjectNotFound=101
CreateShareFailed=102
DeleteShareFailed=103
UnKnownError=999
#
# Shell script main
PROG=$0
# Check used command line options
while getopts u:h:s:j:p:cd flag
do
    case "$flag" in
    c)     create="true"; action="create";;
    d)     delete="true"; action="delete";;
    p)     pool="$OPTARG";;
    j)     project="$OPTARG";;
    s)     share="$OPTARG";;
    u)     user="$OPTARG";;Courier New
    h)     appliance="$OPTARG";;
    ¥?)    Usage $PROG ;;
    esac
done
# Create and Delete action are multi-exclusive
[ "$create" = "true" -a "$delete" = "true" ] && Usage $PROG
# None of the arguments can be empty
[ -z "$pool" -o -z "$project" -o -z "$share" -o -z "$appliance" -o -z "$user"
] && Usage $PROG
```

```
#
# Now get to the job at hand
# Start ssh and feed the script code in using stdin
ScriptError=`ssh $user@$appliance << EOF
script
// Above command activates script mode in the appliance.
// For ease of use, group all arguments in one object.
// Note we use the shell variables obtained from the shell command line.
// Version:  1.0.0
var MyArguments = {
        pool:               '$pool',
        project:            '$project',
        share:              '$share',
        what:               '$action'
}
// We could use the script variables throughout the scripting code but it
might create
// confusion, so keep all the shell-script variable interaction concentrated
in one
// place in the script.
var MyErrors = {
        PoolNotFound:       '$PoolNotFound',
        ProjectNotFound:    '$ProjectNotFound',
        CreateShareFailed:  '$CreateShareFailed',
        DeleteShareFailed:  '$DeleteShareFailed',
        UnKnownError:       '$UnknownError',
}
function CreateDeleteShare (Arg) {
        run('cd /');        // Make sure we are at root child context level
        run('shares');
        try {
                run('set pool=' + Arg.pool);
        } catch (err) {
                return(MyErrors.PoolNotFound);
        }
        try  {
                run('select ' + Arg.project);
        } catch (err) {
                return(MyErrors.ProjectNotFound);
        }
        if (Arg.what=='create' )
                { try {
                        run('filesystem ' + Arg.share);
                        run('commit');
                } catch(err) {
```

```
                        return(MyErrors.CreateShareFailed);
                }
                return(0);
        } else {
                try {
                        run('select' + Arg.share);      // Check if share is
there
                        run('done');                    // Release for delete
                        run('confirm destroy ' + Arg.share);
                } catch (err) {
                        if (err.code == 10004 )
                                return(MyErrors.DeleteShareFailed);
                        else return(MyErrors.UnKnownError);
                }
                return(0);
        }
}
// Kick off the create delete function using our MyArguments object to pass
on the
// parameters needed for the job.
err=CreateDeleteShare(MyArguments);
// The devil is in the tail; return the error code to stdout of ssh so the
shell can
// pick it up in the ScriptError variable.
print(err);
.
EOF`
echo $ScriptError

[ "$ScriptError" != "0" ] && {
   case $ScriptError in
        $PoolNotFound      Message="Specified pool : $pool, not found";;
        $ProjectNotFound)   Message="Specified project : $project, not found";;
        $CreateShareFailed) Message="Share $share could not be created";;
        $DeleteShareFailed) Message="Share $share could not be deleted";;
        $UnknownError)     Message="Unexpected script error";;
   esac
        echo $Message
        exit 1
}
echo "$action of share: share in project: $project, pool: $pool, was
successful"
```

JavaScript全体でシェル変数を使うこともできますが、JavaScriptのメンテナンスが難しくなります。使いやすくするため、使用するすべてのシェル変数はJavaScriptの最初の部分に集約されてい

ます。また、Oracle ZFS Storage Applianceにスクリプトが送信される前に、CLIスクリプト内のシェル変数がシェルによって置換されています。

付録A：参考資料

注：Sun ZFS Storage Appliance、Sun ZFS Storage 7000、およびZFS Storage Applianceはすべて、同じOracle ZFS Storage Appliance製品ファミリのことを指しています。引用しているドキュメントや画面のコードの中には、従来の命名規則を使用しているものがあります。

- *Oracle ZFS Storage Applianceシステム管理ガイド*：
  http://www.oracle.com/technetwork/jp/documentation/oracle-unified-ss-193371.html
- Oracle ZFS Storage Applianceの管理ガイドは、アプリケーションのBUIからアクセスできるオンライン・ヘルプで参照できます。
- *JavaScript：The Definitive Guide, 6th Edition*、David Flanagan著（O'Reilly Media、2006年）
- ECMAスクリプトのWikipediaページ：
  https://ja.wikipedia.org/wiki/ECMAScript
- JavaScriptのWikipediaページ：
  https://ja.wikipedia.org/wiki/JavaScript
- ECMAスクリプト言語の仕様[y1]：
  http://www.ecmascript.org/docs.php
- JavaScriptのオンライン・チュートリアル[y2]：
  http://www.howtocreate.co.uk/tutorials/javascript/introduction

付録B：Oracle ZFS Storage Appliance Simulatorの使用

Oracle ZFS Storage Appliance Simulatorは、CLIインターフェイスとスクリプト言語に慣れるのに最適なプラットフォームです。CLIにアクセスするには、Oracle ZFS Storage Applianceコンソールまたは SSH接続を使用します。

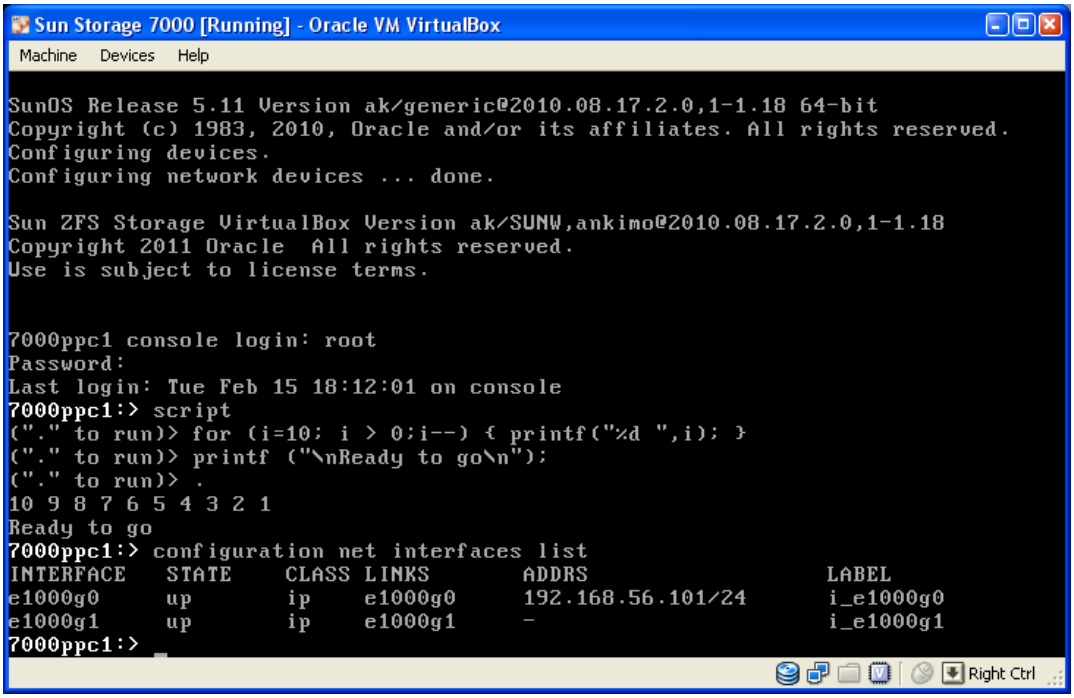

図10：シミュレータを使用したOracle ZFS Storage Applianceコンソールのアクセス

スクリプト・コマンドを対話形式で入力すると、Oracle ZFS Storage ApplianceでのJavaScript言語の構文と使用法を調査できます。

コンソールから入手できるネットワーク・インタフェース構成情報を使用して、Oracle ZFS Storage ApplianceへのSSH接続を開始できます。SSHを使うと、CLIコマンドやシンプルなスクリプトのバッチをOracle ZFS Storage Applianceで簡単に実行できます。

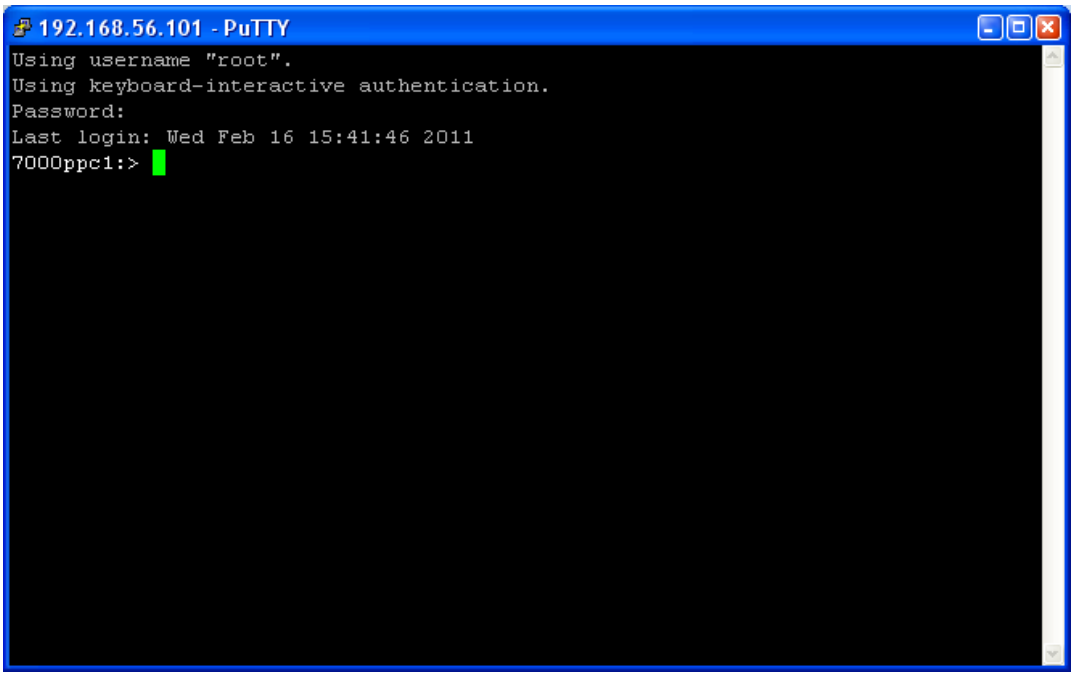

図11：Oracle ZFS Storage ApplianceへのSSH接続

ファイル内でコマンドをグループ化することができます。sshコマンドのstdinオプションに対してファイルを提供することで、1組のコマンドをOracle ZFS Storage Applianceに提供できます。

例として、以下のコマンドをファイルmybatchjobに保存しました。

```
Configuration net interfaces list
script printf(“hello¥n”) ; printf(“End of My Batch Job¥n”);
```

次に、このファイルをsshコマンドに渡します。

```
pBrouwers MacBook-Pro:~ pBrouwer$ ssh root@192.168.56.101 < mybatchjob
Password:
Last login: Mon Feb 14 16:09:05 2011 from 192.168.56.1
INTERFACE     STATE  CLASS  LINKS         ADDRESS            LABEL
E1000g0       up     ip     e1000g0       192.168.56.101     i_e1000g0

Hello
End of My Batch Job
pBrouwers MacBook-Pro:~ pBrouwer$
```

ORACLE

スクリプトを使用したOracle ZFS Storage Applianceの効果的な管理
2014年2月、バージョン2.0、
著者：Peter Brouwer

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together