N1 Grid Service Provisioning System 5.0 XML スキーマリファレンスガイド

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819–1542–10
2004 年 12 月

原典: *N1 Grid Service Provisioning System 5.0 XML Schema Reference Guide*

Part No: 819-1090-10

Revision A

050127@10536

目次

例目次

はじめに

『*N1 Grid Service Provisioning System 5.0 XML* スキーマリファレンスガイド』では、コンポーネント、コンポーネント型、プラン、プラグイン、およびプラグインユーザーインタフェースの定義に使用される XML スキーマに関する詳細を説明します。

対象読者

このマニュアルは、N1™ Grid Service Provisioning System 環境のコンポーネント、プラン、またはプラグインの開発者を対象としています。

お読みになる前に

このマニュアルの対象読者は、以下のマニュアルで説明されている、N1 Grid Service Provisioning System 環境における一般的な概念と作業に関する知識が必要です。

- 『*N1 Grid Service Provisioning System 5.0* システム管理者ガイド』
- 『*N1 Grid Service Provisioning System 5.0* オペレーションとプロビジョニングガイド』
- 『*N1 Grid Service Provisioning System 5.0* プランとコンポーネントの開発者ガイド』
- 『*N1 Grid Service Provisioning System 5.0 Plug-in Development Guide*』

このマニュアルの構成

第 1 章では、N1 Grid Service Provisioning System 製品における XML スキーマの概要を説明します。

第 2 章では、一般的な要素に関する詳細を説明します。

第 3 章では、コンポーネントとコンポーネント型の定義に使用される要素と属性に関する詳細を説明します。

第 4 章では、実行プランの定義に使用される要素と属性に関する詳細を説明します。

第 5 章では、プラグインの記述に使用される要素と属性に関する詳細を説明します。

第 6 章では、プラグインに対するインタフェースの定義に使用される属性と要素に関する詳細を説明します。

付録 A では、コンポーネント間での変更の互換性に関する詳細を説明します。

Sun のオンラインマニュアル

`docs.sun.com` では、Sun が提供しているオンラインマニュアルを参照することができます。マニュアルのタイトルや特定の主題などをキーワードとして、検索を行うこともできます。URL は、http://docs.sun.com です。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 **P–1** 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

コード例は次のように表示されます。

- C シェル

  machine_name% **command y|n** [*filename*]

- C シェルのスーパーユーザー

  machine_name# **command y|n** [*filename*]

- Bourne シェルおよび Korn シェル

  $ **command y|n** [*filename*]

- Bourne シェルおよび Korn シェルのスーパーユーザー

  # **command y|n** [*filename*]

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: `Shift` キーを押します)。ただし、キーボードによっては `Enter` キーが `Return` キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、`Ctrl-D` は `Control` キーを押したまま `D` キーを押すことを意味します。

一般規則

- このマニュアルでは、「x86」という用語は、Intel 32 ビット系列のマイクロプロセッサチップ、および AMD が提供する互換マイクロプロセッサチップを意味します。

第 1 章

XML スキーマの概要

この章では、XML スキーマの概要を説明し、XML スキーマを使用して作業を行う前に必要な一般的な情報を詳細に解説します。

次にこの章の内容を示します。

注 – このマニュアルでは、「派生コンポーネント」、「子コンポーネント」、および「親コンポーネント」という用語は、コンポーネントの複合関係ではなく、コンポーネントの継承関係を指します。

サービスプロビジョニング言語およびスキーマ

N1 Grid Service Provisioning Systemソフトウェアは、XML を介して、プラン、コンポーネント、およびプラグインの定義を実装します。プロビジョニングソフトウェア環境の各部分には、固有の XML スキーマがあります。各 XML スキーマは、プロビジョニングソフトウェア 環境の適切な部分を定義するために使用される、一連の固有の要素を定義します。

N1 Grid Service Provisioning Systemソフトウェアには 5 つの XML スキーマが含まれています。これらのスキーマは、次の 2 つの言語グループに分かれます。

- プランおよびコンポーネント言語は、次の 3 つのスキーマで構成されています。
  - `component.xsd` – コンポーネントとコンポーネント型の定義に使用されるコンポーネントのスキーマ。詳細は、第 3 章を参照してください。
  - `plan.xsd` – 実行プランの定義に使用されるプランのスキーマ。詳細は、第 4 章を参照してください。
  - `planCompShared.xsd` – プランとコンポーネントに共通する要素が含まれるスキーマ。詳細は、第 2 章を参照してください。
- プラグイン言語は、次の 2 つのスキーマで構成されます。
  - `plugin.xsd` – プラグイン記述子ファイルを介してプラグインのパーツを記述するために使用されるプラグインのスキーマ。詳細は、第 5 章を参照してください。
  - `pluginUI.xsd` – N1 Grid Service Provisioning Systemブラウザインタフェース内でプラグインに対するインタフェースを定義するために使用されるプラグインユーザーインタフェースのスキーマ。詳細については、第 6 章を参照してください。

各スキーマについては、このマニュアルの参照先の章で詳細に説明されています。すべてのスキーマに関連する一般的な情報は、この章の後ろで説明します。

ロケールと文字セットに関する要件

プランとコンポーネントは複数バイトデータを含むことができます。プランまたはコンポーネントが XML で作成されている場合、入力ファイルは UTF-8 形式であるか、ファイルの先頭でバイトオーダーマーク (BOM) を使用してその Unicode 符号を指定する必要があります。プランとコンポーネントがダウンロードされる場合、それらは常に UTF-8 形式で書き込まれます。単純コンポーネントが構成可能なリソースを参照する場合、そのリソースは Master Server のネイティブ符号化で符号化されている必要があり、またバイトオーダーマークを使用してその符号化を指定する必要があります。

パターン一致

要素の属性の多くは、正規表現パターンを含むことができます。特に指定がないかぎり、これらのパターンは完全な総称正規表現ではなく glob スタイルのパターンです。つまり、0 個以上の文字と一致させる場合はアスタリスク (*) が使用され、厳密に 1

文字だけと一致させる場合は疑問符 (?) が使用されます。角括弧 ([]) 内の文字は、厳密に 1 文字だけと一致させる文字の範囲を指定します。ハイフン (-) で区切られた文字は、アルファベット順でその範囲内にある文字と、指定された文字を含む任意の文字に一致します。

たとえば、[ab] は a または b に一致します。[a-z] は、任意の小文字に一致します。この場合、厳密な ASCII 文字のみが一致し、アクセント記号が付いた文字など、文字の拡張バリアントには一致しません。

すべての Unicode 文字に一致させるには、パターンには、Perl 5 の regex [[:lower]] などの POSIX 文字クラスが含まれる必要があります。また、非 ASCII 文字を直接含めることもできます。たとえば、[eé] は e または é に一致します。

変数とパラメータの受け渡し

プランとコンポーネントはどちらも、ステップで使用する変数を宣言できます。

コンポーネント変数は、コンポーネントのインストール時に評価、設定されます。したがって、コンポーネント制御ブロック内の手順が、コンポーネントスコープ変数を参照している場合、使用される値はコンポーネントのインストール時の変数の値と同じです。

ただしプラン変数は、プランの実行ごとに評価、設定されます。したがって、プラン内のステップがプラン変数を参照している場合、使用される値はプラン実行時に定義された値となり、実行ごとに異なる可能性があります。

プランは、パラメータと変数の両方を宣言できます。 変数の値は、ほかの変数と定数の値をもとに宣言時に定義されます。パラメータは、呼び出し元が値を定義する特殊な変数です。上位プランの場合、呼び出し元はプランの実行を開始したユーザーです。プランによって宣言される各パラメータに対して、プランを実行する前に、ユーザーはそのパラメータの値を指定します。プランが <execSubplan> 呼び出しの結果として呼び出された場合、その呼び出しを含むプランは、呼び出されたプランによって宣言される各パラメータの値を明示的に渡す必要があります。

<install>、<uninstall>、<snapshot> 、および <control> ブロックは、パラメータとローカル変数を宣言できます。プランの場合と同様に、ローカル変数の値はローカルで定義され、パラメータの値はブロックの呼び出し元が渡した値をもとに定義されます。どちらもプランの実行ごとに異なることがあります。

変数またはパラメータの値を再割り当てすることはできません。

コンポーネントの互換性

すでに配備されているコンポーネントの新しいバージョンを作成する際には、互換性が問題になります。コンポーネントの新しいバージョンを作成する際には、そのコンポーネントを使用または参照するそのほかのオブジェクトが、変更の結果として破損しないようにする必要があります。依存性、コンポーネント包含、継承、およびコンポーネントターゲッターを使用することで、コンポーネントの使用および参照を行うことができます。

プロビジョニングソフトウェア は、次の種類のコンポーネントの互換性をサポートしています。

- 呼び出しの互換性は、コンポーネントに対してユーザーが行える一連の変更を規定します。また呼び出しの互換性により、依存性とコンポーネントターゲッターを通じて存在する関係が保証されます。
- インストールの互換性は、コンポーネントに対してユーザーが行える一連の変更を規定します。またコンポーネントの互換性により、継承とコンポーネント包含関係を通じて存在する関係が保証されます。

さまざまな時点で、データセンターのさまざまな部分に対して異なるバージョンのコンポーネントが配備できるため、開発者は互換性の要件に注意し、あるコンポーネントに対する変更がほかの既存コンポーネントに対してどのように影響するかを理解する必要があります。特定のケースでは、システムにより互換性の要件が強制される場合もあり、また開発者が新しいコンポーネントの互換性を確保しなければならない場合もあります。

あるコンポーネントに対して実施可能な変更の種類のリストは、付録 A を参照してください。

呼び出しの互換性

あるコンポーネントが別のコンポーネントと呼び出し互換であるのは、以下の場合のように、最初のコンポーネントの使用を、別のコンポーネントの使用に安全に置き換えることができる場合です。

- 最初のコンポーネントの `<control>`、`<uninstall>`、または `<snapshot>` ブロックへの呼び出しを行う場合
- `<checkDependency>` または `<createDependency>` を使用して、最初のコンポーネントの依存性を確認する場合
- 構成生成 `:[component:AB:var]` を使用することで、最初のコンポーネントの変数を参照する場合

呼び出しの互換性は、API の互換性やインタフェースの互換性と同じです。

注 – 通常、2 つの呼び出し互換コンポーネントは、同じバージョンツリー内のバージョンが異なるコンポーネントです。ただし、2 つ目のコンポーネントは、最初のコンポーネントのインスタンスである場合、別のバージョンツリー内に存在することも可能です。

プロビジョニングソフトウェアでは、システムサービスを提供するコンポーネントに対して、呼び出しの互換性が強制されます。システムサービスが新しいコンポーネントを参照するよう更新される場合、新しいコンポーネントが元のコンポーネントに対して呼び出し互換性を持つようにします。これにより、システムサービスのアップグレード時にも、システムサービスのクライアントが正常に機能し続けることを保証できます。

プロビジョニングソフトウェアは、一部のインストール済みコンポーネントターゲッターに参照されているコンポーネントを解釈処理するときにも、オプションで呼び出し互換性を検証します。詳細は、46 ページの「インストール済みコンポーネントターゲッター」を参照してください。

コンポーネントがその旧バージョンに対して呼び出し互換である必要はありませんが、呼び出し互換性を持たせることをお勧めします。

インストールの互換性

コンポーネントは、別のコンポーネントとインストール互換になることが可能です。最初のコンポーネントは、もう一方のコンポーネントと呼び出し互換である必要があります。以下の場合、もう一方のコンポーネントの使用は、最初のコンポーネントの使用と安全に置き換えられる必要があります。

- もう一方のコンポーネントのインストールブロックへの呼び出しを行う場合
- コンポーネントがもう一方のコンポーネントから拡張される場合
- コンポーネントがもう一方のコンポーネントへの参照を含む場合

インストールの互換性とは、構造上の互換性と同じです。

インストールの互換性の制約により、既存のインストール済みコンポーネントは、別のインストール互換コンポーネントに安全に置き換えることができる必要があります。オリジナルがどのようにインストールされたかを記述するデータ構造を変更する必要はありません。呼び出し互換のコンポーネントは、データ構造を適切に更新するために再インストールしなければならない場合があるので、呼び出し互換性はかなり弱いステートメントになります。

注 – インストールの互換性を保つためには、両方のコンポーネントは同じバージョンツリーに属している必要があります。2 つの別々のバージョンツリーからのコンポーネントであってはなりません。

プロビジョニングソフトウェア は、コンポーネント型と呼ばれる、型として機能するコンポーネントだけにインストールの互換性を要求します。コンポーネント型が新しいバージョンのコンポーネントを参照するために更新されるときには、新しいバージョンは元のバージョンに対してインストール互換性を持ちます。 したがって、その型から派生するすべての既存コンポーネントを再構築や再インストールせずに、コンポーネント型に対してインストール互換の更新を実行できます。 インストール互換ではないコンポーネント型への変更は、独自の型名を持つ新しい別のコンポーネントバージョンツリーとして符号化する必要があります。このような場合、新しいコンポーネント型は、元のコンポーネント型から拡張することによって、呼び出し互換性を維持できます。 型間の関係を簡単に特定するには、コンポーネント型名を符号化するバージョン管理システムを使用します。 たとえば、コンポーネント名 `EJB-1.0` と `EJB-1.1` がある場合、`EJB-1.1` は `EJB-1.0` コンポーネント型の新しいバージョンであることを簡単に特定できます。

インストール互換である場合は呼び出し互換でもありますが、呼び出し互換であってもインストール互換であるとは限りません。また、コンポーネントが呼び出し互換でない場合は、インストール互換ではありません。

ターゲット可能コンポーネント

`<targetRef>` 要素が存在していれば、インストールされたコンポーネントは、ほかのコンポーネントの配備ターゲットとして使用できることを示します。このようにターゲット可能にするには、コンポーネントの各インストール済みインスタンスを、一意の仮想ホストまたは物理ホストに関連付けます。

ターゲット可能コンポーネントを使用して、関連付けられているホストをターゲットにすることで、プランはインストール済みコンポーネントを論理上ターゲットにすることができます。通常、ターゲット可能コンポーネントは仮想ホストを作成します。ただし、ターゲット可能コンポーネントに物理ホストを作成させることもできます。物理ホストは、関連付けられたホストが独自の Remote Agent を持つモデルで便利です。Solaris™ Zone をモデル化するコンポーネントがこれに該当します。

`<targetRef>` 要素を定義するコンポーネントは、ターゲット可能コンポーネントです。ターゲット可能コンポーネントは、インストール時に、そのほかのコンポーネントの配備ターゲットとして機能するホストを作成します。ターゲット可能コンポーネントをアンインストールすると、そのコンポーネントによって作成されたホストが自動的に削除されます。このようなホストは「Hosts」ページに一覧表示されますが、一覧表示から削除することはできません。またそれらに対して可能な編集の種類も制限されています。

共通の属性型

属性型は、プランまたはコンポーネントの属性の値に対する制約として機能します。ある属性に特定の型が記載されていない場合、その値には制約がありません。

以降の節では、スキーマによって使用される属性型の形式について説明します。\p{N} はすべての Unicode 数字を表し、\p{L} はすべての Unicode 文字を表します。

entityName 属性型

entityName 型の属性は、最大文字長は 512 文字で、次のパターンに一致します。

```
[\p{N}\p{L}-_. ]+
```

特別なケースとして、ドット (.) および 2 つのドット (..) はエンティティ名になることはできません。

systemName 属性型

systemName 型の属性は、次のように、最大文字長が 64 文字である *simpleSystemName*、およびオプションで同じく最大文字長が 64 文字である *pluginName* から構成されています。

simpleSystemName
pluginName#
simpleSystemName

simpleSystemName は次のパターンに一致します。

```
[\p{L}_][\p{N}\p{L}-_. +]*
```

identifier 属性型

identifier 型の属性は、最大文字長は 32 文字で、次のパターンに一致します。

```
[\p{L}_][\p{N}\p{L}_]*
```

pathName 属性型

pathName 型の属性は、最大文字長は 512 文字で、次のいずれかのパターンに一致します。

```
"/"
( "/" pathPart )+
```

ここで *pathPart* は `[\p{N}\p{L}-_.]+` です。

特別なケースとして、ドット (.) および 2 つのドット (..) を、*pathPart* に含めることはできません。

pathReference 属性型

pathReference 型の属性の構文は次のようになります。

```
pathReference:
    absolutePath
    relativePath

absolutePath:
    "/"
    "/" relativePath

relativePath:
    relativePathStart
    relativePathStart "/" relativePath

relativePathStart:
    "."
    ".."
    identifier
```

modifierEnum 属性型

modifierEnum 型の属性は、値として ABSTRACT または FINAL を取ります。一般的に、値 ABSTRACT は、関連付けられるエンティティが派生コンポーネントによってオーバーライドされることを示します。値 FINAL は、関連付けられるエンティティがオーバーライドされないことを示します。

accessEnum 属性型

accessEnum 型の属性は、次のいずれかの値を取ります。

- PUBLIC – 関連付けられたエンティティのアクセシビリティが無制限であることを示します。
- PROTECTED – アクセシビリティが、同じパス内にある派生コンポーネントまたはエンティティに制限されることを示します。
- PATH – アクセシビリティが同じパス内にあるエンティティに制限されることを示します。

- PRIVATE – アクセシビリティが宣言元コンポーネントに制限されることを示します。

version 属性型

version 型の属性は、次のパターンに一致します。

```
[0-9]+\.[0-9]+
```

schemaVersion 属性型

schemaVersion 型の属性は、唯一の値 5.0 のみを取ることができます。

第 2 章

コンポーネントと単純プランにより使用される共有スキーマ

この章では、単純プランとコンポーネントの両方により使用可能な 3 つの共有エンティティについて説明します。

特に断りのない限り、この章で説明する属性は、コンポーネントスコープの置換変数を参照できません。

共有ステップ

この節では、コンポーネントまたは単純プランのいずれかで使用可能なステップを説明します。

<call> ステップ

ターゲットホストにインストールされているコンポーネントに関連付けられた制御ブロックを実行するには、<call> ステップを使用します。

<call> ステップには次の子要素があります。

- <argList> – オプション要素で、制御ブロックに渡す引数の一覧。この要素を指定する場合、この要素は 1 回しか出現できません。
- インストール済みコンポーネントターゲッター – 実行する制御ブロックが含まれるコンポーネントを特定するオプション要素。この要素は <call> ステップがコンポーネントに出現する場合はオプションになりますが、ステップがプランに出現する場合はオプションではありません。この要素を省略すると、<thisComponent> が使用されます。この要素を指定する場合、この要素は 1 回しか出現できません。詳細は、46 ページの「インストール済みコンポーネントターゲッター」を参照してください。

<call> ステップの属性

<call> 要素には、*blockName* 型の 1 つの必須属性 entityName があり、これはインストール済みコンポーネントに対して実行する制御ブロックの名前です。

<argList> 要素

<argList> 要素は、<call>、<install>、<uninstall>、<execSubplan>、および <addSnapshot> ステップの子です。この要素は、呼び出されるサービスに引数として渡される一連の変数を指定します。

呼び出されるサービスの方は、待ち受ける変数を <paramList> 要素を使用して宣言します。<argList> 内の変数と呼び出される <paramList> 内の変数は同じである必要はありません。しかし、呼び出される <paramList> 内の変数のうちデフォルト値を持たないものについては、<argList> 内に同じ名前の変数が存在しなければなりません。この条件が満たされない場合、プラン実行時にプリフライトエラーが発生します。呼び出される <paramList> 内の変数のうち <argList> 内に対応する変数が存在するものは、呼び出されるサービスの実行時に <argList> 内の対応する変数の値をその値として取得します。

<argList> 内の変数のうち 呼び出される <paramList> 内の変数に対応しないものは、単純に無視されます。よって、下位互換性を維持しながら、追加されたパラメータによりサービスを再定義できます。そのため、1 つのプランで新旧両方のサービスバージョンを呼び出すことができます。

<argList> 要素の引数は属性として表現されます。引数の順番はあまり重要ではありません。たとえば、次の <argList> は 2 つの引数、「*password*」と「*path*」を宣言しています。

```
<argList password=":[password]" path="/tmp"/>
```

<argList> 要素属性

<argList> 要素には、属性を少なくとも 1 つは指定する必要があります。各属性は、呼び出されるサービスに渡される名前付き変数として扱われます。各属性の名前は置換変数参照のない識別子でなければならず、呼び出されるサービス内のパラメータの名前に対応する必要があります。各属性の値は任意の文字列であり、包含するスコープ内の変数の参照を含むことができます (<argList> 内のほかの引数は含むことができない)。

<checkDependency> ステップ

<checkDependency> ステップは、特定のコンポーネントがターゲットホスト上にインストール済みであることを検証するために使用できます。適切なコンポーネントがインストールされていない場合、ステップは失敗し、実行が停止します。

この処理では、包含されているコンポーネントターゲッターによって依存性がチェックされます。このターゲッターが正常にコンポーネントを解決処理すると、依存性が満たされます。正常に解決処理されない場合、依存性は満たされません。

<checkDependency> ステップには、1 つの必須子要素があり、これはインストール済みのコンポーネントターゲッターで、依存性をチェックするコンポーネントを特定します。詳細は、46 ページの「インストール済みコンポーネントターゲッター」を参照してください。

<execJava> ステップ

このステップは、ターゲットホスト上で Java ™ Executor インスタンスを実行します。Executor インスタンスが例外を送出する場合、ステップは失敗し、実行が停止します。

<execJava> ステップには、1 つのオプション子要素である <argList> があり、これは Executor インスタンスに渡す引数の一覧です。この要素を指定する場合、この要素は 1 回しか出現できません。

<execJava> ステップの属性

<execJava> 要素には次の属性があります。

- *className* – 必須属性で、(Plan Executor で指定されているとおりに) ExecutorFactory インタフェースを実装する、public no-arg コンストラクタを持つ public クラスのフルネーム。この属性は、単純置換変数を参照できます。
- *classPath* – オプション属性で、*className* 属性で指定されたクラスを含むクラスパス。この属性を指定しないと、Remote Agent のシステムクラスパスが使用されます。クラスパスの形式は、エージェント上の Java Archive (JAR) ファイルの絶対パスをセミコロンで区切ったリストです。この属性は、単純置換変数を参照できま

す。

- *timeout* – positiveInteger 型のオプション属性で、コマンドの完了を待機する時間を秒数で指定し、この時間を経過するとタイムアウトになります。この属性を指定しないと、プランの <execNative> タイムアウト期間が適用されます。この値は 0 より大きくする必要があります。

<execJava> ステップがコンポーネント内に含まれる場合、通常このステップはそのコンポーネントによって配備された 1 つ以上のリソースに含まれるクラスを呼び出します。ステップがプラン内に含まれる場合、呼び出されるクラスはエージェント自体のシステムクラスとして、あるいは既存コンポーネントによって配備されたリソースとして、エージェント上にすでに存在します。

<execNative> ステップ

<execNative> ステップは、ターゲットホストでオペレーティングシステムに対してネイティブコマンドを実行します。 コマンドが予期しなかった結果を生成する場合、ステップは失敗し、実行が停止します。

<execNative< ステップには次の子要素があります。

- <env> – オプション要素で、子プロセスの環境変数を指定します。各環境変数に対しては、1 つの <env> 要素を指定します。
- <background> – オプション要素で、コマンドをバックグラウンドプロセスとして実行することを指定します。この要素を指定する場合、この要素は 1 回しか出現できません。<background> 要素を指定する場合は、<outputFile> 要素と <errorFile> 要素も指定する必要があります。
- <outputFile> – オプション要素で、コマンドの標準出力を格納するファイルの名前。この要素を指定する場合、この要素は 1 回しか出現できません。この要素は、<background> 要素が指定してある場合は必須要素になります。
- <errorFile> – オプション要素で、コマンドの標準エラー出力を格納するファイルの名前。この要素を指定する場合、この要素は 1 回しか出現できません。この要素は、<background> 要素が指定してある場合は必須要素になります。
- <inputText> – オプション要素で、コマンドの標準入力として使用するテキスト。この要素を指定する場合、この要素は 1 回しか出現できません。<inputFile> 要素が指定されていると、この要素は指定できません。
- <inputFile> – オプション要素で、コマンドの標準入力として使用するファイル名。この要素を指定する場合、この要素は 1 回しか出現できません。<inputText> 要素が指定されていると、この要素は指定できません。
- <exec> – 必須要素で、実行する実行可能ファイル名を指定します。<shell> 要素が指定されていると、この要素は指定できません。
- <shell> – 必須要素で、実行するシェルコマンドを指定します。この要素は 1 回しか出現できません。<exec> 要素が指定されていると、この要素は指定できません。
- <successCriteria> – オプション要素で、このステップが成功したか失敗したかを判断するために使用される基準。この要素を指定する場合、この要素は 1 回しか出現できません。

<execNative> ステップの属性

<execNative> ステップには次の属性があります。

- *userToRunAs* – オプション属性で、このコマンドを実行するためのユーザーの名前。この属性を指定しないと、コマンドは構成ファイル内の defaultUserToRunAs の値として実行されます。値として、文字列によるユーザー名または数値によるユーザー ID を指定できます。この値は、空でない文字列にする必要があります。この属性は、単純置換変数を参照できます。
- *dir* – オプション属性で、コマンドの作業ディレクトリの絶対パス。この属性を指定しないと、値のデフォルトはエージェント固有の構成可能ディレクトリになります。この値は、空でない文字列にする必要があります。この属性は、単純置換変数を参照できます。
- *timeout* – positiveInteger 型のオプション属性で、コマンドの完了を待機する時間を秒数で指定します。この時間を経過するとタイムアウトとなります。この属性を指定しないと、プランの <execNative> タイムアウト期間が適用されます。この値は 0 より大きくする必要があります。

<env> 要素

<execNative> 要素は、コマンドの環境変数を指定する目的で、必要に応じて <env> 要素を含むことができます。<env> を使用すると、コマンド環境の新しい変数を提供することも、既存の変数を無効にすることもできます。

コマンドの環境変数セットは、Remote Agent の環境変数セットと、<env> 要素を使用して提供される変数を合わせたものです。

<env> 要素属性

<env> 要素には次の属性があります。これらの属性は、単純置換変数を参照できます。

- *name* – 必須属性で、環境変数の名前。この値は、空でない文字列にする必要があります。
- *value* – 必須属性で、環境変数の値。この値には、Remote Agent の環境変数の値を参照するように、 ${*var-name* } 文字列を含めることができます。<env> 要素により無効にされた変数を *var-name* が参照する場合、その置換された値は Remote Agent の環境内で定義されたものであり、無効にされた値ではありません。値に ${ という文字列を表示させる必要がある場合は、 ${{ を使用してこの文字列をエスケープします。${{ のインスタンスはすべて ${ に置換されます。

<background> 要素

<background> 要素は <execNative> 要素の子です。この要素が存在する場合、コマンドをバックグラウンドプロセスとして実行する必要があることを示します。この要素には属性も子要素もありません。

<background> 要素が指定された <execNative> の場合、以下の制限が適用されます。

- <successCriteria> は指定の必要はありません。ステップは、バックグラウンドプロセスを開始する上で何も問題が存在しなかった場合には成功します。指定すると、コマンドをバックグラウンドプロセスとして実行するために使用されるスクリプトに照らして <successCriteria> のテストが行われます。
- 実行されるコマンドの標準出力または標準エラー出力はキャプチャされません。

  ブラウザインタフェースの「details」を参照すると、終了ステータス 0 と、空の標準出力、および標準エラー出力が表示されます。しかし、バックグラウンドでネイティブコマンドを開始する際に問題があった場合は、別の終了ステータスが表示されます。その値はエラーの特性と診断出力に応じて決まります。
- <outputFile> および <errorFile> 要素を指定する必要があります。指定したファイルがすでに存在する場合、それらは上書きされます。

<outputFile> 要素

<outputFile> 要素は <execNative> 要素の子です。<outputFile> 要素は、実行されるコマンドの標準出力が格納される、エージェント上のローカルファイルのパスを指定します。ローカルファイルのパスは、<outputFile> 要素の *name* 属性を使用して指定します。相対パスは、コマンドの作業ディレクトリに相対的であると解釈されます。

<background> 要素を指定する場合は、<outputFile> 要素を指定する必要があります。

<outputFile> を指定せず、<successCriteria> 要素で *outputMatches* も指定しないと、コマンドの出力は格納されずに消失します。*outputMatches* を指定すると、標準出力は一時ファイルに格納され、コマンドの実行後そのファイルが削除されます。

<outputFile> 要素属性

<outputFile> 要素には 1 つの必須属性 *name* があり、これはコマンドの標準出力の書き込み先となるファイル名です。この値は、空でない文字列にする必要があります。この属性は、単純置換変数を参照できます。

<errorFile> 要素

<errorFile> 要素は、実行されるコマンドのエラー出力を格納する、エージェント上のローカルファイルのパスを指定します。ローカルファイルのパスは、<errorFile> 要素の *name* 属性を使用して指定します。相対パスは、コマンドの作業ディレクトリに相対的であると解釈されます。

<background> 要素を指定する場合は、<errorFile> 要素を指定する必要があります。

<errorFile> を指定せず、<successCriteria> 要素で *errorMatches* も指定しないと、コマンドの出力は格納されずに消失します。*errorMatches* を指定すると、エラー出力は一時ファイルに格納され、コマンドの実行後そのファイルが削除されます。

<errorFile> 要素属性

<errorFile> 要素には 1 つの必須属性 *name* があり、これはコマンドの標準エラー出力の書き込み先となるファイル名です。この値は、空でない文字列にする必要があります。この属性は、単純置換変数を参照できます。

<inputText> 要素

<inputText> 要素は <execNative> 要素の子です。この子要素は、コマンドの標準入力として使用する必要がある任意のテキストを指定します。このテキストは、この要素のコンテンツとして指定されます。

```
<execNative>
    <inputText>
        ls -l | fgrep '*test*' | sort -u > file.out
    </inputText>
    <command exec="sh"/>
</execNative>
```

<inputText> 要素の本体は、CDATA セクションで囲むことができます。これにより、入力フォーマットを保持するとともに、文字 & や < を含む入力で発生しうる解析エラーを防ぐことができます。

<inputText> を指定すると、<execNative> ステップの XML を生成する際に常に CDATA セクション内に包含されます。<inputText> 内の文字はすべて、実行されるコマンドに標準入力として渡されます。<inputText> に空白文字だけを含めると、それらの空白文字は指定されているとおりに、実行されるコマンドに渡されます。

<inputText> のコンテンツは config 生成されます。

<inputText> 要素に属性はありません。

<inputFile> 要素

<inputFile> 要素は <execNative> の子要素です。この子要素は、実行されるコマンドの標準入力として使用されるコンテンツが入った、Remote Agent 上のローカルファイルのパスを指定します。ローカルファイルのパスは、<inputFile> 要素の *name* 属性を使用して指定します。

注 – <inputFile> 要素は、<execNative> コマンド内で <inputText> 要素と併用することはできません。

<inputFile> 要素属性

<inputFile> 要素には 1 つの必須属性 *name* があり、これはコマンドに対する標準入力として動作するファイルです。この値は、空でない文字列にする必要があります。相対パスを指定した場合、コマンドの作業ディレクトリに対して相対的であると解釈されます。この属性は、単純置換変数を参照できます。

<exec> 要素

<execNative> ステップには <exec> 要素を 1 つしか含めることができません。<exec> 要素は、実行されるネイティブコマンドの詳細を指定します。

<exec> 要素には次の属性が含まれます。

- 実行するコマンドの名前を指定する *cmd* 属性
- *cmd* コマンドの各引数用の、入れ子になった一連の <arg> 要素

<execNative> は、*cmd* により指定されたコマンドを、指定された引数を順に使用して実行します。

次に、<execNative> ステップが ps -fu sps コマンド実行する例を示します。

```
<execNative>
    <exec cmd="ps">
        <arg value="-fu"/>
        <arg value="sps"/>
    </exec>
</execNative>
```

<exec> 要素にはオプションの子要素 <arg> があります。実行するコマンドに対する各引数に 1 つの <arg> 要素を指定します。

<arg> 要素には 1 つの必須属性 *value* があり、この属性は引数値です。この引数は、<arg> がコマンド要素の *n* 番目の子であるコマンドに *n* 番目の引数として渡されます。この属性は、単純置換変数を参照できます。

<exec> 要素属性

<exec> 属性には 1 つの必須属性 *cmd* があり、この属性は実行するコマンドのパスです。指定したパスが絶対パスでない場合、Remote Agent に対して設定された、プラットフォーム指定の PATH 環境変数を使用してコマンドの検索が行われます。この値は、空でない文字列にする必要があります。この属性は、単純置換変数を参照できます。

<shell> 要素

<shell> 要素は <execNative> の子要素です。<shell> 要素のコンテンツは、実行されるコマンドを指定します。実行されるコマンドは、*cmd* 属性で指定されるインタプリタによって解釈されます。したがって、そのコマンドはプラットフォームの sh -c "command" 構文を使用して実行されます。この書式では、コマンドの実行に使用するシェルコマンドを示すために *cmd* 属性を指定する必要があります。

次に例を示します。

```
<execNative>
    <shell cmd="/usr/bin/bash -c">
        ls -l | fgrep '*test*' | sort -u > file.out
    </shell>
</execNative>
```

前に示した <execNative> の例では、次のコマンドを実行していました。

```
/usr/bin/bash -c 'ls -l | fgrep '*test*' | sort -u > file.out'
```

書式を保持するとともに XML の解析問題を回避するため、<execNative> ステップから XML 表現を生成する際には常に CDATA 要素内にコマンドのテキストコンテンツが包含されます。

コマンド文字列を空の状態にしたり、空白文字だけを入れたりすることは認められません。コマンド文字列は、(周囲の空白も含め) 指定されているとおりにシェルに送られます。

<shell> 要素のコンテンツは config 生成されます。

<shell> 要素属性

<shell> には 1 つの必須属性 *cmd* があり、この属性は sh -c 構文内のシェルコマンドです。この文字列に、埋め込みの引用符文字を含めないでください。この文字列は、空白を区切りとして使用して、シェル名と引数を取得するために解析されます。たとえば、/usr/bin/bash -c は空でない文字列にします。この属性は、単純置換変数を参照できます。

<successCriteria> 要素

<successCriteria> 要素は <execNative> の子要素です。この子要素は、<execNative> ステップが正常に実行されたかを評価するために使用される基準を指定します。 この要素が指定されない場合のデフォルト値は <successCriteria status="0"/> です。

空の <successCriteria> を指定すると、<successCriteria> は無視されます。

<successCriteria/> を指定した場合、コマンドがどのような出力または終了コードを生成した場合でも、ステップは常に成功します。

<successCriteria> 要素属性

<successCriteria> 要素には次のオプション属性があります。*status*、*outputMatches*、および *errorMatches* 属性の中から複数の属性が指定されると、それらの AND (論理積) がとられます。

- *status* – 整数型のオプション属性で、コマンドの適切な終了ステータス。この値は正の整数でなければなりません。
- *outputMatches* – オプション属性で、コマンドによって生成された標準出力を照合する正規表現。この値は、空でない文字列にする必要があります。この属性は、単純置換変数を参照できます。
- *errorMatches* – オプション属性で、コマンドによって生成された標準エラー出力を照合する正規表現。この値は、空でない文字列にする必要があります。この属性は、単純置換変数を参照できます。
- *inverse* – ブール型のオプション属性で、true に設定すると、<successCriteria> で指定された各条件が無視されます。デフォルト値は、false です。つまり、<successCriteria> の各属性で指定された条件が満たされない場合だけステップは成功します。

  たとえば、次の <successCriteria> を持つ <execNative> ステップが成功するのは、*status* が 1 ではなく、標準出力が bin に一致せず、標準エラーが none に一致しない場合です。

```
<successCriteria status="1" outputMatches="bin"
errorMatches="none" inverse="true"/>
```

  inverse 属性だけを含む <successCriteria> 要素は、*inverse* 属性なしで保存されます。つまり、次のように指定すると、要素は <successCriteria/> として格納され、<successCriteria> 要素は無視されます。

```
<successCriteria inverse="true"/>
```

<if> ステップ

このステップは、ステップブロックを条件付きで実行するために使用されます。このステップには属性がありません。子要素は <condition> 、<then>、および <else> です。<condition> 要素と <then> 要素は同時に出現する必要があります。<else> 要素を指定する場合、この要素は 1 回しか出現できません。

<condition> 要素のコンテンツが true に評価される場合、<then> ブロックのステップは実行されます。true に評価されない場合、 <else> ブロックのステップが実行されます (存在する場合)。

例 **2–1** <if> ステップの使用方法

以下に、条件付きで再起動するために使用される <if> ステップの例を示します。

```
<if>
    <condition><istrue value=":[restart]"/></condition>
    <then>
```

例 **2–1** <if> ステップの使用方法　(続き)

```
        <call blockName="restart"/>
    </then>
</if>
```

<condition> 要素

<condition> 要素は <if> ステップの子要素で、ブール式を指定します。この要素に属性はありません。この要素は、ブール演算子の子要素を 1 つだけ含む必要があります。詳細は、57 ページの「ブール型演算子」を参照してください。

<then> 要素

<then> 要素は <if> ステップの子要素です。この要素は、関連付けられた条件が true の場合に実行するステップを指定します。<then> 要素には、<if> ステップを含むブロックのスコープ内で許可されるステップをいくつでも含めることができます。

<else> 要素

<else> 要素は <if> ステップの子要素です。この要素は、関連付けられた条件が true でない場合に実行するステップを指定します。<else> 要素には、<if> ステップを含むブロックのスコープ内で許可されるステップをいくつでも含めることができます。

<pause> ステップ

<pause> ステップは、指定された期間、プランの実行を一時的に中止します。<pause> は、必要なサービスが起動されたあと、このサービスがオンラインになるのをプランに待機させるために使用できます。

<pause> ステップ属性

<pause> 要素は positiveInteger 型の 1 つの必須属性 *delaySecs* を持ちます。この属性は待機する時間 (秒数) を指定します。

<processTest> ステップ

<processTest> ステップは、ターゲットホストで特定のプロセスが実行されているかを調べるために使用されます。目的のプロセスが存在しない場合、ステップは失敗し、実行が停止します。

注 – このステップは、UNIX® システムでの使用を意図したものです。

<processTest> ステップの属性

<processTest> ステップには次の属性があります。

- *delaySecs* – positiveInteger 型の必須属性で、プロセスが存在するかどうかをテストする前に待機する時間 (秒数)。
- *timeoutSecs* – positiveInteger 型の必須属性で、プロセスがオンラインになるのを待機する時間 (秒数)。この時間が経過すると失敗になります。この時間は、遅延時間が経過したあとからカウントされます。
- *processNamePattern* – 必須属性で、指定のプロセス名の照合に使用する glob スタイルパターン。この属性は、単純置換変数を参照できます。
- *user* – オプション属性で、プロセス所有者の名前を照合するために使用される glob スタイルパターン。この属性を指定しないと、プロセス所有者はテストの一環と見なされません。この属性は、単純置換変数を参照できます。

<raise> ステップ

<raise> ステップは、常に失敗するステップです (<try> ステップにおける捕捉と処理は可能)。 このステップには「*message*」という属性だけが存在し、子要素はありません。

<raise> ステップは、人為的なステップを構築することなくエラー状況を示す手段として使用されます。このステップの出現がもっとも多いのは <catch> ブロック内であり、クリーンアップ後のエラー状況を伝えるために使用されます。

<raise> ステップの属性

<raise> 要素には 1 つのオプション属性 *message* があり、この属性はエラー状況を説明するメッセージです。デフォルトは、メッセージはシステムで指定された一般的なメッセージです。この属性は、単純置換変数を参照できます。

例 **2–2** <raise> ステップの使用方法

以下の例は、ログ内にエラーを見つけたあと <catch> ブロック内からエラー状況を再伝達するために <raise> ステップをどのように使用するかを示しています。

```
<control blockName="default">
    <try>
        <block>
            <!-- ここで任意の処理が行われます -->
```

例 **2–2** <raise> ステップの使用方法　(続き)

```
        </block>
        <catch>
            <!-- ログのエラーに注意してください -->
            <execNative>
                <exec cmd="appendLog">
                    <arg value="an error occurred"/>
                </exec>
            </execNative>
            <!-- rethrow エラーです -->
            <raise/>
        </catch>
    </try>
</control>
```

MS Windows:<reboot> ステップ

このステップは、プランの残り部分を続行する前にエージェントにリブートを行わせます。このステップは、Microsoft Windows (MS Windows) ベースのシステム上でしか使用できません。MS Windows 以外のプラットフォームで検出された場合は、エラーが発生します。Master Server と同じホスト上に存在するエージェントをリブートした場合もエラーとなります。ここで説明しているエラーはプリフライトエラーです。

MS Windows:<reboot> ステップの属性

<reboot> ステップには positiveInteger 型の 1 つのオプション属性である *timeout* があります。この属性はサーバーのリブートを待機する最長時間 (秒数) です。この属性を指定しないと、プランの <execNative> タイムアウト期間が適用されます。

<retarget> ステップ

このステップは、一連のステップの実行対象を変更します。retarget ステップは入れ子にすることができます。

<retarget> ステップには次の子要素があります。

- <varList> – オプション要素で、包含されたステップで使用できるローカル変数の一覧。これらの変数は、新しいホストのスコープ内で評価されます。この要素を指定する場合、この要素は 1 回しか出現できません。
- **Steps –** オプション要素で、新しいホスト上で実行するステップが含まれます。<retarget> ステップを含むブロックのスコープ内で許可される任意のステップを指定できます。複数のステップを指定できます。例については、例 2–3を参照してください。

<retarget> ステップの属性

<retarget> ステップには必須属性 *host* があり、この属性は包含されたステップの実行対象となるホストです。この属性は、単純置換変数を参照できます。

この属性は、各種のコンポーネントターゲッター要素のほか、<retarget> ステップでも使用できます。この属性の値はホストの名前であり、置換変数参照を含むことができます。また、シンボリック名「/」を含めて現在の実行対象のルート物理ホストを参照することも、あるいは「..(/..)*」を含めて現在の実行対象の親ホストを参照することもできます。

注 – *host* 属性をコンポーネントターゲッター内に指定した場合、意味的には、包含するステップを <retarget> ステップ内に含めたのと同じになります。

<retarget> ステップの実行セマンティクス

<retarget> ステップが検出される場合、まず呼び出し側の現在のホストのコンテキストで「*host*」属性を評価します。

「*host*」属性の値が現在のホストの名前と異なる場合、以下のステップが実行されます。

1. ホスト名を実際のホストに解決します。そのようなホストが存在しない場合、エラーが発生します。
2. プランのフォルダ、またはこのステップを含むコンポーネントに関して、現在のユーザーが、そのホストにおいて「実行」アクセス権を所有しているかを検証します。所有していない場合、エラーが発生します。
3. ホストのルート物理ホストに関して、以下の条件が満たされていることを検証します。

   - Remote Agent が含まれている。
   - ルート物理ホストに接続できる。
   - ルート物理ホストが最新の状態で準備が整っている。

   これらの条件が満たされない場合、エラーが発生します。
4. 以前にアクセスしたすべてのホストでロックを保持したまま、このホストでロックを取得します。現在の実行スレッドがすでにホストをロックしている場合、この処理は事実上ノーオペレーション (空命令) となります。ホストがすでにほかの実行スレッドによってロックされている場合、この処理はホストのロックが解除されるまでブロックします。ロック要求のためにデッドロックが起きた場合は、エラーが発生します。
5. そのホストが再設定され、新しい「現」ホストになります。「物理」ホストは、新しい「現」ホストに基づいて再設定されます。「初期」ホストは変化しません。

上記のステップが完了したところで、<varList> 要素の変数が新しい現ホストのコンテキストで評価されます。ローカル変数は、包含するスコープの変数を隠蔽することができます。続いて、各ステップが新しい現ホストのコンテキストで実行されます。

最後に、対象変更の処理によって現ホストが変わった場合には、そのロックが適宜解除され、その現ホストが呼び出し側の現ホストとして再設定されます。現在の実行スレッドでホストが以前にロックされている場合は、ロックを初めに取得したブロックが完了するまでロックしたままとなります。

空の <retarget> ブロックを使用することにより、現在のユーザーが適切なアクセス許可を持っていることと、ホストの準備が適切に行われていることをあらかじめ検証できます。

例 **2–3** <retarget> ステップの使用方法

以下に、WebLogic 管理対象サーバー上で使用できる「再起動」制御サービスの例を示します。この制御サービスは、管理サーバー上で制御を呼び出して管理対象サーバーを「停止」し、続いてローカルサーバーで呼び出しを行なってサーバーを起動することによって実施されます。

「adminHostName 」変数は、呼び出し側の現ホスト (これは管理対象サーバーを含んでいる仮想ホストと見なされる) で評価されます。「domainName」変数は、新しいターゲットホスト (これは管理サーバーを含んでいる仮想ホストと見なされる) で評価されます。ADMIN_SERVER コンポーネントも、新しいターゲットホストで解決処理されます。

```
<control name="restart">
    <varList>
        <var name="adminHostName" default=":[target:adminHostName]"/>
    </varList>
    <retarget host=":[adminHostName]">
        <varList>
            <var name="domainName" default=":[target:domainName]"/>
        </varList>
        <call blockName="stopServer">
            <argList serverName=":[serverName]"
          domainName=":[domainName]"/>
            <installedComponent name="ADMIN_SERVER"/>
        </call>
    </retarget>
    <call blockName="start"/>
</control>
```

<sendCustomEvent> ステップ

<sendCustomEvent> ステップは、特定のメッセージを使用してカスタムイベントを生成するために使用されます。このステップを通知規則モジュールと併用することで、特定のホストでこのステップが検出されるたびに電子メールを送信できます。

<sendCustomEvent> ステップの属性

<sendCustomEvent> ステップには 1 つの必須属性 *message* があり、この属性はイベントテキストとして含めるメッセージ です。この属性は、単純置換変数を参照できます。

<transform> ステップ

<transform> ステップは、ターゲットホスト上にあるファイルのテキストベース変換を行うために使用されます。現時点では、Perl タイプおよび XSLT ベースの変換がサポートされています。

<transform> の子要素は、入力ファイルに適用される変換を指定します。これらの子要素は、以下のどれでもかまいません。

- 入力ソースに適用する XSLT 変換を定義する単一の <stylesheet> 要素
- 入力ソースに順次適用する Perl に似た置換パターンを定義する 1 つ以上の <subst> 要素
- 変換を含んでいる外部ファイルを命名する単一の <source> 要素
- 空。これは、入力ファイルのコンテンツが出力ファイルに直接コピーされることを意味します。

  これは、zip アーカイブから抽出する場合または zip アーカイブに書き込む場合に便利です。

<transform> ステップの属性

<transform> ステップには次の属性があります。

- *input* – オプション属性で、変換の適用先である、ターゲットホスト上のファイルの一般パス。この属性を指定しないと、入力は出力ファイルから読み取られます。この属性は、単純置換変数を参照できます。
- *output* – 必須属性で、変換結果の書き込み先である、ターゲットホスト上のファイルの一般パス。この属性は、単純置換変数を参照できます。

input 属性と *output* 属性は、同じファイルまたは個別のファイルを参照できます。これらの属性の値は、zip アーカイブ (または zip の派生フォーマットである JAR など) をディレクトリ要素として含むことができる一般パスです(例: webapp/myapp.jar/config.xml)。

<stylesheet> 要素

<stylesheet> 要素は <transform> ステップの子であり、入力ソースに適用する XSLT 変換を指定します。特定の <transform> 要素の子として指定できるのは <stylesheet> 要素 1 つであり、<stylesheet> 要素をほかの子要素と併用することはできません。

<stylesheet> 要素は、名前空間 http://www.w3.org/1999/XSL/Transform で定義されている XSLT バージョン 1.0 要素です。詳細は、http://www.w3.org/TR/xslt.html の『*XSL Transformations (XSLT)*』仕様のバージョン 1.0 を参照してください。<transform> 要素の子として受け入れられるのは、XSLT <stylesheet> 要素だけです。XSLT 仕様の節 2.3 で説明された XSLT シノニム <transform>、単純化された XSLT 変換構文とも、<transform> 要素の子としてはサポートされません。

<stylesheet> 要素本体には、本体が有効な XSLT であるかぎり、初めに変数置換を行うことなく置換変数参照を含めることができます。

<stylesheet> 要素が変換として使用される場合、入力ファイルは XML 形式でなければなりません。詳細は、『*XSL Transformations (XSLT)*』仕様のバージョン 1.0 を参照してください。

例 **2–4** <stylesheet> 要素の使用方法

```
<transform output="/etc/hosts">
    <xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="/">
            <xsl:for-each select="a">
                <xsl:value-of select="b"/>
            </xsl:for-each>
        </xsl:template>
    </xsl:stylesheet>
</transform>
```

<subst> 要素

<subst> 要素は <transform> ステップの子であり、変換として適用する Perl に似た置換パターンを指定します。複数の <subst> 要素を <transform> 要素の子として指定できますが、ほかの子要素とは併用できません。複数の <subst> 要素が出現する場合、それらは順次適用されます。

入力ファイル内に出現するパターンは、1 行内に複数出現している場合も含めすべて置換されます。

サポートされる構文の詳細は、Java ドキュメント (class java.util.regex.Pattern) を参照してください。

<subst> 要素属性

<subst> 要素には次の属性があります。

- *match* – 必須属性で、入力後に検索される Perl に似た正規表現 (大文字小文字の区別がなされる)。この属性は、単純置換変数を参照できます。
- *replace* – 必須属性で、「*match*」で指定されたパターンが出現するごとに置換される Perl に似た置換値。この値は一語一語解釈されるのではなく、構成体 $*n* が検索表現内で *n* 番目に出現する括弧付きの表現として解釈されます。この属性は、単

純置換変数を参照できます。

例 **2–5** <subst> 要素の使用方法

以下の変換は、ファイル /etc/hosts 内で、文字列 127.0.0. *xxx* をすべて 10.10.0.*xxx* に変換します。

```
<transform output=”/etc/hosts”>
    <subst match="127\.0\.0\.(\d+)"
    replace="10.10.0.$1"/>
</transform>
```

<source> 要素

<source> 要素は <transform> ステップの子であり、入力ファイルに適用する変換が入っている、ターゲットホスト上の外部ファイルを指定します。特定の <transform> 要素の子として指定できるのは <source> 要素 1 つであり、<source> 要素をほかの子要素と併用することはできません。

指定されたソースファイルに対して <transform> ステップの一部として config 生成が行われることはありません。しかし、指定されたソースファイルが、コンポーネントインストールの一部として配備された config 型のリソースファイルである場合があります。このようなケースでは、ソースファイルに含まれる置換変数は、ソースファイルが配備された際に置換されています。

<source> 要素属性

<source> 要素には次の属性があります。

- *type* – 必須属性で、指定されたファイル内に含まれる変換のタイプ。次の値が使用できます。
  - PERL – <subst> 要素による変換に類似した Perl に似た変換。この場合、指定されるファイルは以下のような形式をとる必要があります。

    ```
    <?xml version='1.0'?>
    <transform>
        <subst match="127\.0\.0\.(\d+)"
        replace="10.10.0.$1"/>
    </transform>
    ```

    Perl 型の外部変換ファイルは、任意の数の <subst> 要素を含むことができます。
  - XSLT – XSLT 変換。この場合、指定されたファイルには、名前空間 http://www.w3.org/1999/XSL/Transform で定義されているように、標準の XSLT バージョン 1.0 変換が含まれます。XSLT <stylesheet> 要素しか許可しないインライン変換と異なり、外部ソースファイルに含まれる XSLT 変換は任意の有効な最上位 XSLT 変換要素を含むことができます。このような要素としては、 <stylesheet>、<transform>、および単純化された XSLT 構

文などが該当します。単純化された XSLT 構文は、XSLT 仕様の節 2.3 で説明されています。

- *name* – 必須属性で、変換を含む、ターゲットホスト上のファイルの名前。ファイルのコンテンツは、「*type*」属性で定義された型に一致する必要があります。この名前は、ディレクトリ要素として `zip` アーカイブを含むことはできません。この属性は、単純置換変数を参照できます。

<try> ステップ

<try> ステップは、ステップブロックの典型的なエラー処理とクリーンアップロジックを指定するために使用されます。このステップには属性がありません。このステップには次の 3 つの子要素、<block>、<catch>、および <finally> があります。

<try> ステップは次のように実行されます。

- <block> 要素内のステップは、すべてが完了するか、ステップのどれかが失敗するまで順に実行されます。各ステップの失敗は、Plan Executor により決定されます。
- <catch> 要素が定義されており、かつプランの中断 (abort) またはプランのタイムアウト以外のステップエラーでブロック要素の実行が終了した場合にかぎって、すべてが完了するかあるいはステップのどれかが失敗するまで <catch> 要素内のステップが順に実行されます。
- <finally> 要素が定義されている場合、そのステップはそれらがすべてが完了するかステップのどれかが失敗するまで順に実行されます。これは、プランの中断またはプランのタイムアウトのために先の 2 つの要素のどちらか一方が停止しないかぎり、それらの要素の成否に関係なく発生します。
- プランの中断またはプラン (ステップではなく) のタイムアウトのために、包含されるステップのどれかが失敗する場合、包含されるほかのステップは実行されず、<try> ステップの実行はエラーを示して終了します。

<catch> 要素は、<block> 要素内で発生するエラーの抑制と、これらのエラーからの回復のために使用されます。<finally> 要素は、典型的なエラーが検出されるかどうかにかかわらず、無条件でクリーンアップを行うために使用されます。

<try> ステップは、以下のように成否が決まります。

- <try> ステップに <finally> 要素だけが含まれる場合、<block> または <finally> 要素の実行が失敗すると、このステップは失敗します。
- <try> ステップに <catch> 要素だけが含まれる場合、このステップが失敗するのは、<catch> 要素の実行が行われて失敗した、あるいはプランの中断またはプランのタイムアウトのために <block> 要素が失敗した場合のみです。
- <try> ステップに <catch> および <finally> 要素の両方が含まれる場合、このステップは以下の状況のいずれかが発生するときだけ失敗します。
  - <catch> または <finally> 要素が実行されて失敗した場合。

- <finally> 要素が実行されて失敗した場合。
- プランの中断またはプランのタイムアウトのために <block> 要素が失敗した場合。

以上のことから、<block> 要素の失敗は <catch> 要素の存在で抑制できると言えます。

<try> ステップには次の子要素があります。

- <block> – 必須要素で、初めに実行されるステップから構成されています。
- <catch> – オプション要素で、典型的なエラー時に実行されるステップが含まれています。<finally> 要素が指定されない場合には、指定する必要があります。この要素を指定する場合、この要素は 1 回しか出現できません。
- <finally> – オプション要素で、典型的なエラーであるかどうかにかかわらず実行されるステップが含まれます。<catch> 要素が指定されない場合には、指定する必要があります。この要素を指定する場合、この要素は 1 回しか出現できません。

<block> 要素

<block> 要素は、<try> ステップの子要素です。この要素は、<try> ステップによって実行される主要なステップを指定します。<block> 要素には、<try> ステップを含んでいるブロックのスコープ内で許可される 1 つ以上のステップが含まれます。

<catch> 要素

<catch> 要素は、<try> ステップの子要素です。この要素は、<block> 要素のステップを実行している際に典型的なエラーが発生する場合に実行するステップを指定します。<catch> 要素には、 <try> ステップを含んでいるブロックのスコープ内で許可される任意の数のステップを含めることができます。

<catch> 要素には、<block> 要素の典型的なエラーを抑制するという機能と、典型的なエラー回復作業を定義するという機能があります。この要素は空にすることができます。空にすると、<catch> は典型的なエラーの抑制だけを行います。

<finally> 要素

<try> ステップの子要素です。<finally> は、<try> または <catch> ステップ内で典型的なエラーが以前に発生したかどうかにかかわらず実行するステップを指定します。典型的なエラーには、プランの中断とプランのタイムアウトがあり、これらのエラーが発生した場合、<finally> 要素のステップはスキップされます。
<finally> には、<try> ステップを含んでいるブロックのスコープ内で許可される任意の数のステップを含めることができます。

<finally> 要素は、エラーを考慮することなく常に実行する必要があるクリーンアップステップを指定するために使用されます。エラーに対応してクリーンアップステップを実行するには、代わりに <catch> 要素内にクリーンアップステップを配置します。

例 **2–6** <try> ステップの使用方法

以下の例では、コンポーネントのインストールは、再起動があとに続くリソース配備で構成されています。このケースでは、コンポーネントのインストールが完了したと見なされるのは、そのリソースが配備され、再起動時における失敗がそのインストール状態に影響を与えない場合だけです。典型的なエラーは、以下のように再起動時に抑制できます。

```
<installSteps blockName="default">
    <deployResource/>
    <try>
        <block>
            <call blockName="restart"><thisComponent/></call>
        </block>
        <catch/><!-- すべての典型的なエラーを抑制します -->
    </try>
</installSteps>
```

<try> ブロックは、インテリジェントな自動アップグレードのモデル化に使用できます。バージョン 1.1 のコンポーネントが 2 つの異なるインストールルーチンを持っているとします。あるユーザーはコンポーネントのフレッシュインストールを行い、そのコンポーネントのバージョン 1.0 が以前にインストールされていた場合、もう 1 人のユーザーがアップグレードインストールを行います。この状況は、以下のように単一のインストールブロックとしてモデル化できます。

```
<installSteps blockName="default">
    <try>
        <block>
          <checkDependency>
              <installedComponent name="foo" version="1.0"/>
          </checkDependency>
          <!-- 1.0 のインストールが存在するため、アップグレードを行います -->
        </block>
        <catch>
          <!-- 1.0 のインストールが存在しないため、フレッシュインストールを行います -->
        </catch>
    </try>
</installSteps>
```

<finally> ブロックは、通常、一時リソースのクリーンアップに使用されます。次の例では、一時ファイルの作成、処理、および削除が行われます。

```
<control blockName="default">
    <varList>
        <var name="file" default="/tmp/file.txt"/>
    </varList>
    <execNative outputFile=":[file]">
        <exec cmd="ls"><arg value="-l"/></exec>
    </execNative>
```

例 **2–6** <try> ステップの使用方法　(続き)

```
    <try>
        <block>
            <!-- 何らかの方法でファイルを処理します -->
        </block>
        <finally>
            <execNative>
                <exec cmd="rm"><arg value=":[file]"/></exec>
            </execNative>
        </finally>
    </try>
</control>
```

<urlTest> ステップ

このステップは、特定の URL のコンテンツが予期されたパターンに一致するかどうかを検証するために使用されます。妥当なパターンに一致しない場合、ステップは失敗し、実行が停止します。

<urlTest> ステップの属性

<urlTest> ステップには次の属性があります。

- *delaySecs* – A required attribute of type positiveInteger 型の必須属性で、URL コンテンツのテストを実施する前に待機する時間 (秒数)。
- *timeoutSecs* – positiveInteger 型の必須属性で、URL コンテンツの受け取りを待機する時間 (秒数)。この時間を過ぎると失敗します。この時間は、遅延時間が経過したあとからカウントされます。
- *URL* – 必須属性で、コンテンツのテストが行われる URL。現在サポートされているのは HTTP プロトコルだけです。この属性は、単純置換変数を参照できます。
- *pattern* – 必須属性で、テストされる URL のコンテンツに一致すると予測される glob スタイルのパターン。この値は複数バイト符号化をサポートしています。この属性は、単純置換変数を参照できます。

インストール済みコンポーネントターゲッター

この節では、特定のインストール済みコンポーネントを包含ステップ (制御サービスコールなど) の対象として特定する要素を示します。対象となるすべてのステップですべてのターゲッターを使用できるわけではありません。各ターゲッターは、それ自体が使用できるステップを指定します。

<installedComponent> インストール済みコンポーネントターゲッター

<installedComponent> 要素は、ターゲットホスト上にインストールされていると想定される特定のインストール済みコンポーネントを特定します。

この要素は、<checkDependency>、<createDependency>、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。

このターゲッターは指定されたコンポーネントを直接照合するもので、指定されたコンポーネントの派生インスタンスの照合には使用できません。特定の型から派生したコンポーネントを対象とする場合は、<systemType> ターゲッターを使用してください。

<installedComponent> ターゲッターの属性

このターゲッターには次の属性があります。

- *name* – entityName 型の必須属性で、インストール済みコンポーネントの名前。
- *path* – pathReference 型のオプション属性で、コンポーネントのパス。この属性を指定しないと、包含するエンティティのパスが使用されます。
- *version* – version 型のオプション属性で、インストール済みコンポーネントのバージョン。この属性を指定しないと、バージョンを考慮せず、最後にインストールされたコンポーネントが使用されます。

- *versionOp* – オプション属性で、*version* 属性と、ターゲットホストにインストールされているコンポーネントのバージョンを比較する際に使用する演算子を指定します。複数のインストール済みコンポーネントが適用される場合、最後にインストールされたコンポーネントが使用されます。使用できる値は、=、>=、および > です。

  デフォルトでは >= 演算子が使用されます。*version* を指定しないと、*versionOp* は無視されます。

- *onlyCompat* – オプション属性で、照合を行うコンポーネントを指定します。

  値が `true` である場合、バージョン *version* のコンポーネントと呼び出し互換性のあるコンポーネントだけを照合する必要があります。バージョン *version* のコンポーネントが存在する必要があります。デフォルトでは値は `false` です。*version* を指定しないと、この要素は無視されます。

- *installPath* – オプション属性で、インストール済みコンポーネントのインストールパス。この属性を指定しないと、任意のパスの一番新しいインストール済みコンポーネントが使用されます。この値は、コンポーネントの解決処理の前に共通書式に変換されます。この属性は、単純置換変数を参照できます。

- *host* – オプション属性で、コンポーネントがインストールされているホスト。デフォルトでは、*host* は現在のホストです。*host* 属性の詳細については、38 ページの「<retarget> ステップの属性」を参照してください。この属性は、単純置換変数を参照できます。

<systemService> インストール済みコンポーネントターゲッター

<systemService> 要素は、現在の物理ホストにインストールされていると想定される特定のシステムサービスコンポーネントを特定します。システムサービスがプラグインにより定義されている場合は、*pluginName*# *serviceName* のように、サービス名にはプラグイン名を接頭辞として付ける必要があります。

この要素は、<checkDependency>、<createDependency>、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。

<systemService> ターゲッターを使用すると、暗黙に現在のホストのルート物理ホストに対象が変更されます。別のホスト上のシステムサービスを対象とする必要がある場合は、<retarget> ステップを使用する必要があります。このステップを使用しないかぎり、<systemService> ターゲッター内で新しいホストを指定することはできません。

<systemService> ターゲッターの属性

<systemService> ターゲッターには systemName 型の 1 つの必須属性 *name* があり、これはシステムサービスコンポーネントの名前です。システムサービスがプラグインにより定義されている場合は、*pluginName*# *serviceName* のように、システムサービス名にはプラグイン名を接頭辞として付ける必要があります。

<systemType> インストール済みコンポーネントターゲッター

<systemType> 要素は、ターゲットホスト上にインストールされていると想定される特定の型のインスタンスであるコンポーネントを特定します。指定した基準に複数のインストール済みコンポーネントが一致する場合、一番最近インストールされたコンポーネントが使用されます。

<systemType> 要素は、<checkDependency>、<createDependency>、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。

<systemType> ターゲッターの属性

<systemType> ターゲッターには次の属性があります。

- *name* – systemName 型の必須属性で、システム型コンポーネントの名前。システム型がプラグインにより定義されている場合は、*pluginName*# *typeName* のように、システム型名にはプラグイン名を接頭辞として付ける必要があります。
- *installPath* – オプション属性で、必要なコンポーネントのインストールパス。この値は、コンポーネントの解決処理の前に共通書式に変換されます。この属性は、単純置換変数を参照できます。
- *host* – オプション属性で、コンポーネントがインストールされるホストです。デフォルトでは、*host* は現在のホストです。*host* 属性の詳細については、38 ページの「<retarget> ステップの属性」を参照してください。この属性は、単純置換変数を参照できます。

<thisComponent> インストール済みコンポーネントターゲッター

<thisComponent> 要素は、ステップを含むコンポーネントを、ステップの対象として使用する必要があることを指定します。 このターゲッターを使用できるのは、コンポーネント内に含まれるステップだけです。この要素には属性がありません。

<thisComponent> 要素は、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。

リストされたステップにコンポーネントターゲッター要素が含まれない場合、デフォルトで <thisComponent> が使用されます。

<superComponent> インストール済みコンポーネントターゲッター

<superComponent> 要素は、ステップを含むコンポーネントのベースコンポーネントを、ステップの対象として使用する必要があることを示します。このターゲッターを使用できるのは、派生コンポーネント内に含まれるステップだけです。

<superComponent> 要素は、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。この要素には属性がありません。

このターゲッターは、派生コンポーネントが無効にする場合でも、常にベースコンポーネントによる該当ステップの定義にバインドします。

<nestedRef> インストール済みコンポーネントターゲッター

<nestedRef> 要素は、現在の複合コンポーネントによって宣言または継承が行われた、入れ子になったコンポーネント参照を特定します。このターゲッターを使用できるのは、複合コンポーネント内のステップだけです。

<nestedRef> 要素は、<checkDependency>、<call>、<uninstall> 、および <addSnapshot> ステップのターゲッターとして使用できます。

この際、指定されたコンポーネント参照が呼び出し側コンポーネントによってすでにインストールされていると想定され、インストールされていない場合はエラーが生成されます。入れ子になったコンポーネント参照が現在のターゲットホスト以外のホストにインストールされた場合は、<nestedRef> ターゲッターを使用することで、関連付けられているステップの対象がそのホストに暗黙に変更されます。

<nestedRef> ターゲッターの属性

<nestedRef> ターゲッターには identifier 型の 1 つの必須属性 *name* があり、これは当該コンポーネント内の入れ子になったコンポーネント参照の名前です。

<allNestedRefs> インストール済みコンポーネントターゲッター

<allNestedRefs> 要素は、現在の複合コンポーネントによって宣言または継承が行われた、入れ子になったコンポーネント参照をすべて特定します。 このターゲッターを使用できるのは、複合コンポーネント内のステップだけです。

この要素は、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。

このターゲッターは、任意の数のコンポーネントを特定できます。コンポーネントがまったく特定されない場合、ステップはノーオペレーション (空命令) となります。このターゲッターが複数のコンポーネントを特定する場合、特定されるコンポーネントごとに <nestedRef> ターゲッターを使用する個別のステップが存在するかのように、ステップが意味的に展開されます。ステップは同時にではなく連続的に実行され、ステップの順序はステップ型に応じて変化します。コンポーネントの 1 つに対してステップを実行しエラーが生じる場合は、そのステップは一致するほかのコンポーネントに対しては実行されません。

<call> または <addSnapshot> ステップのターゲッターとして使用される場合、ターゲッターは当該コンポーネントによって現在インストールされている、入れ子になったすべてのコンポーネント参照を照合します。コンポーネントの照合はインストール順に行われます。

<uninstall> ステップのターゲッターとして使用される場合、ターゲッターは当該コンポーネントによって現在インストールされている、入れ子になったすべてのコンポーネント参照を照合します。コンポーネントの照合はインストールの逆順に行われます。

<toplevelRef> インストール済みコンポーネントターゲッター

<toplevelRef> 要素は、現在の複合コンポーネントによって宣言または継承が行われた最上位のコンポーネント参照を特定します。このターゲッターを使用できるのは、複合コンポーネント内のステップだけです。

<toplevelRef> 要素は、<checkDependency>、<createDependency>、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。

このターゲッターは、*name*、*path*、および *version* 属性が参照先コンポーネントに基づいて事前定義されることを除き、意味的に <installedComponent> ターゲッターと同じです。詳細は、47 ページの「 <installedComponent> インストール済みコンポーネントターゲッター」を参照してください。

<toplevelRef> ターゲッターの属性

<toplevelRef> ターゲッターには次の属性があります。

- *name* – identifier 型の必須属性で、当該コンポーネント内の最上位コンポーネント参照の名前。
- *versionOp* – オプション属性で、参照先コンポーネントのバージョンをターゲットホスト上にインストールされたコンポーネントのバージョンと比較する際に使用する演算子を指定します。複数のインストール済みコンポーネントが適用される場合、最後にインストールされたコンポーネントが使用されます。

使用できる値は、=、>=、および > です。この属性を指定しないと、>= が使用されます。

- *onlyCompat* – オプション属性で、照合されるコンポーネントを指定します。true の場合、参照先コンポーネントと呼び出し互換性のあるコンポーネントだけを照合する必要があることを示します。デフォルト値は false です。
- *installPath* – オプション属性で、参照先コンポーネントのインストールパス。この属性を指定しないと、任意のパスの最新の参照先コンポーネントインストールが使用されます。この値は、コンポーネントの解決処理の前に共通書式に変換されます。この属性は、単純置換変数を参照できます。
- *host* – オプション属性で、参照先コンポーネントがインストールされているホスト。デフォルトでは、*host* は現在のホストです。*host* 属性の詳細は、38 ページの「<retarget> ステップの属性」を参照してください。この属性は、単純置換変数を参照できます。

<dependee> インストール済みコンポーネントターゲッター

<dependee> 要素は、呼び出し側コンポーネントによってその依存性 (<createDependency> で作成される) が宣言されたインストール済みコンポーネントを特定します。このターゲッターを使用できるのは、コンポーネント内のステップだけです。

<dependee> 要素は、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。

<dependee> ターゲッターの属性

<dependee> ターゲッターには identifier 型の 1 つの必須属性 *name* があり、これは当該コンポーネントによって作成される依存性の名前です。

<allDependants> インストール済みコンポーネントターゲッター

<allDependants> 要素は、呼び出し側コンポーネントに対して依存性 (<createDependency> で作成) を宣言してあるインストール済みコンポーネントのセットを特定します。このターゲッターを使用できるのは、コンポーネント内のステップだけです。

<allDependants> 要素は、<call>、<uninstall>、および <addSnapshot> ステップのターゲッターとして使用できます。

このターゲッターは、包含するステップを、一致するすべてのコンポーネントにマップさせるという点で <addNestedRefs> ターゲッターと似た機能を持ちます。依存コンポーネントに対するマップの順序は指定されません。

<allDependants> ターゲッターの属性

<allDependants> ターゲッターは、identifier 型の 1 つの必須属性 *name* を持っています。これはほかのコンポーネントによって当該コンポーネントに対して作成される依存性の名前です。

<targetableComponent> インストール済みコンポーネントターゲッター

<targetableComponent> 要素は、ホストを対象とする特定のコンポーネントと関連付けられた、ターゲット可能コンポーネントを特定します。

<targetableComponent> 要素は、<call>、<uninstall>、<checkDependency>、<createDependency>、および <addSnapshot> ステップのターゲッターとして使用できます。

<targetableComponent> ターゲッターの属性

<targetableComponent> ターゲッターには 1 つのオプション属性 *name* があり、これはホストを対象とするコンポーネントの名前です。この属性を指定しないと、値は現在のターゲットホストになります。この属性は、単純置換変数を参照できます。

共通イントールパスの書式

インストールパスはインストール済みコンポーネント参照内に指定できます。その場合、*installPath* 属性の値は、インストール済みコンポーネント参照を解釈処理する前に、共通書式に変換されます。このような変換が行われるのは、インストール済みコンポーネントのインストールパスが、共通書式でも保存されるためです。

共通書式では、インストールパス内にあるパス区切り文字はすべてスラッシュ「/」に置き換えられます。パス区切り文字は、Master Server で動作中のオペレーティングシステムに固有です。末尾の「/」は削除されます。ルートインストールパス (/) は例外で、空のパスには変換されません。

Master Server アプリケーションが UNIX ベースのシステムで実行中である場合、末尾のスラッシュは無視されます。そのため、/opt/apache ディレクトリにインストールされているコンポーネントを指すためには、/opt/apache/ と /opt/apache の両方を使用できます。

リポジトリコンポーネントターゲッター

この節では、Master Server リポジトリに包含ステップ (install など) の対象として存在する特定のコンポーネントを指定する要素を説明します。対象となるすべてのステップですべてのターゲッターを使用できるわけではありません。各ターゲッターは、それ自体が使用できるステップを指定します。

<component> リポジトリコンポーネントターゲッター

<component> 要素は、コンポーネントリポジトリに存在すると想定される特定のコンポーネントを特定します。このターゲッターを使用できるのは、単純プラン内に含まれるステップだけです。

この要素は、<install> ステップのターゲッターとして使用できます。

<component> ターゲッターの属性

<component> ターゲッターには次の属性があります。

- *name* – entityName 型の必須属性で、コンポーネントの名前。
- *path* – pathReference 型のオプション属性で、コンポーネントのパス。この属性を指定しないと、包含するエンティティのパスが使用されます。
- *version* – version 型のオプション属性で、コンポーネントのバージョン。この属性を指定しないと、最新バージョンのコンポーネントが使用されます。
- *host* – オプション属性で、コンポーネントがインストールされるホスト。デフォルトでは、*host* は現在のホストです。*host* 属性の詳細は、38 ページの「<retarget> ステップの属性」を参照してください。この属性は、単純置換変数を参照できます。

<thisComponent> リポジトリコンポーネントターゲッター

<thisComponent> 要素は、ステップを含むコンポーネントを、ステップの対象として使用する必要があることを指定します。このターゲッターを使用できるのは、コンポーネント内に含まれるステップだけです。この要素には属性がありません。

この要素は、<install> ステップのターゲッターとして使用できます。

リストされたステップにコンポーネントターゲッター要素が含まれない場合、デフォルトで <thisComponent> が使用されます。

<superComponent> リポジトリコンポーネントターゲッター

<superComponent> 要素は、ステップを含むコンポーネントのベースコンポーネントを、ステップの対象として使用する必要があることを示します。このターゲッターを使用できるのは、派生コンポーネント内に含まれるステップだけです。この要素には属性がありません。

この要素は、<install> ステップのターゲッターとして使用できます。

このターゲッターは、派生コンポーネントが無効にする場合でも、常にベースコンポーネントによる該当ステップの定義にバインドします。

<nestedRef> リポジトリコンポーネントターゲッター

<nestedRef> 要素は、現在の複合コンポーネントによって宣言または継承が行われた、入れ子になったコンポーネント参照を特定します。このターゲッターを使用できるのは、複合コンポーネント内のステップだけです。

この要素は、<install> ステップのターゲッターとして使用できます。

この際、指定されたコンポーネント参照が呼び出し側コンポーネントによってまだインストールされていないと想定され、インストールされている場合はエラーが生成されます。入れ子になった参照先コンポーネントが別のホストにインストールされる場合は、<retarget> ステップを使用する必要があります。このステップを使用しないかぎり、<nestedRef> ターゲッター内で新しいホストを指定することはできません。 特定の包含コンポーネントについて、入れ子になったコンポーネント参照を複数のホストにインストールすることはできません。

<nestedRef> ターゲッターの属性

<nestedRef> ターゲッターには、identifier 型の 1 つの必須属性 *name* があり、これは当該コンポーネント内の入れ子になったコンポーネント参照の名前です。

<allNestedRefs> リポジトリコンポーネントターゲッター

<allNestedRefs> 要素は、現在の複合コンポーネントによって宣言または継承が行われた、入れ子になったコンポーネント参照をすべて特定します。 このターゲッターを使用できるのは、複合コンポーネント内のステップだけです。

この要素は、<install> ステップのターゲッターとして使用できます。

このターゲッターは、任意の数のコンポーネントを特定できます。コンポーネントがまったく特定されない場合、ステップはノーオペレーション (空命令) となります。このターゲッターが複数のコンポーネントを特定する場合、特定されるコンポーネントごとに <nestedRef> ターゲッターを使用する個別のステップが存在するかのように、ステップが意味的に展開されます。ステップは同時にではなく連続的に実行され、ステップの順序はステップ型に応じて変化します。コンポーネントの 1 つに対してステップを実行しエラーが生じる場合は、そのステップは一致するほかのコンポーネントに対しては実行されません。

<install> ステップのターゲッターとして使用される場合、ターゲッターは当該コンポーネントによって宣言されている、入れ子になったすべてのコンポーネント参照を照合します。コンポーネントの照合は宣言順に行われます。

<toplevelRef> リポジトリコンポーネントターゲッター

<toplevelRef> 要素は、現在の複合コンポーネントによって宣言または継承が行われた最上位のコンポーネント参照を特定します。このターゲッターを使用できるのは、複合コンポーネント内のステップだけです。

この要素は、<install> ステップのターゲッターとして使用できます。

このターゲッターは、*name*、*path*、および *version* 属性が参照先コンポーネントに基づいて事前定義されることを除き、意味的に <component> ターゲッターと同じです。最上位コンポーネント参照は、任意の数のホストに任意の回数だけインストールできます。

<toplevelRef> ターゲッターの属性

<toplevelRef> ターゲッターには次の属性があります。

- *name* – identifier 型の必須属性で、当該コンポーネント内の最上位コンポーネント参照の名前。
- *host* – オプション属性で、参照先コンポーネントがインストールされるホスト。デフォルトでは、*host* は現在のホストです。*host* 属性の詳細は、38 ページの「<retarget> ステップの属性」を参照してください。この属性は、単純置換変数を参照できます。

ブール型演算子

この節では、ブール型演算子として機能し、<if> ステップの条件内で使用される要素を紹介します。ブール型演算子は、true または false にのみ評価可能です。

<istrue> ブール型演算子

特定の値が true かどうかを判断するために使用するブール型演算子です。<istrue> は *value* という属性を 1 つ含み、子要素は持ちません。<istrue> は、*value* が true と等しいときだけ、true となります。比較では大文字と小文字が区別されます。

<istrue> ブール型演算子の属性

<istrue> 演算子には 1 つの必須属性 *value* があり、これは true と比較する値です。この属性は、単純置換変数を参照できます。

例 **2–7** <istrue> ブール型演算子の使用方法

次の例に、<istrue> の使用方法と結果を示します。

- 次の文は true になります。

```
<istrue value="True"/>
```

- 次の文は false になります。

```
<istrue value="yes"/>
```

- 次の文は、*var* が true である場合、true になります。

```
<istrue value=":[var]"/>
```

<equals> ブール型演算子

特定の値が別の値と等しいかどうかを判断するために使用されるブール型演算子です。この演算子には *value1*、*value2*、および *exact* 属性があります。この演算子には子要素はなく、*value1* 属性と *value2* 属性が等しい ときだけ、true となります。*exact* が true の場合、値は大文字小文字を含めて等しくなければなりません。*exact* が false であれば、比較では大文字と小文字は区別されません。

<istrue value="..."/> は次の文の構文上の短縮形です。

```
<equals value1="..." value2="true"/>
```

<equals> ブール型演算子の属性

<equals> 演算子には次の属性があります。

- *value1* – 必須属性で、比較する値。この属性は、単純置換変数を参照できます。
- *value2* – 必須属性で、比較するもう 1 つの値。この属性は、単純置換変数を参照できます。
- *exact* – boolean 型のオプション属性で、true の場合は大文字小文字も区別され、false の場合は区別されません。デフォルトは false です。

例 **2–8** <equals> ブール型演算子の使用方法

次の例に、<equals> の使用方法と結果を示します。

- 次の文は true になります。

  ```
  <equals value1="True" value2="true"/>
  ```

- 次の文は false になります。

  ```
  <equals value1="True" value2="true" exact="true"/>
  ```

- 次の文は true になります。

  ```
  <equals value1="apple" value2="apple" exact="true"/>
  ```

- 次の文は false になります。

  ```
  <equals value1="apple" value2="orange"/>
  ```

- *var1* が *var2* に等しい場合、次の文は true になります。

  ```
  <equals value1=":[var1]" value2=":[var2]"/>
  ```

<matches> ブール型演算子

特定の値がパターンに一致するかどうかを判断するために使用されるブール型演算子です。この演算子には *value*、*pattern*、および *exact* 属性があります。この演算子には子要素がなく、*value* の値が *pattern* に含まれる glob スタイルのパターンに一致する場合だけ、true となります。*exact* が true である場合、値は大文字小文字を含めて一致しなければなりません。そのほかの場合は大文字と小文字は区別されません。

<matches> ブール型演算子の属性

<matches> 演算子には次の属性があります。

- *value* – 必須属性で、パターンと照合される値。この属性は、単純置換変数を参照できます。

- *pattern* – 必須属性で、一致しなければならないパターン。この属性は、単純置換変数を参照できます。
- *exact* – オプション属性で、true の場合は大文字小文字も区別され、false の場合は区別されません。デフォルトは false です。

例 **2–9** <matches> ブール型演算子の使用方法

次の例に、<matches> の使用方法と結果を示します。

- 次の文は true になります。

```
<matches value="True" pattern="true"/>
```

- 次の文は true になります。

```
<matches value="True" pattern="t*"/>
```

- 次の文は false になります。

```
<matches value="blue" pattern="*u"/>
```

- 次の文は true になります。

```
<matches value="True" pattern="t?ue"/>
```

- 次の文は false になります。

```
<matches value="Tue" pattern="t?ue"/>
```

- 次の文は false になります。

```
<matches value="True" pattern="t*" exact="true"/>
```

- *var1* が *var2* のパターンに一致する場合、次の文は true になります。

```
<matches value=":[var1]" pattern=":[var2]"/>
```

<not> ブール型演算子

別のブール型演算子の結果を否定するブール型演算子です。この演算子は属性を含まず、ほかのブール型演算子の子要素である子要素を 1 つだけ含みます。この演算子は、含んでいる演算子が true でない場合だけ、true になります。

例 **2–10** <not> ブール型演算子の使用方法

次の例に、<not> の使用方法と結果を示します。

- 次の文は false になります。

```
<not><istrue value="True"/></not>
```

- 次の文は true になります。

```
<not><equals value1="apple" value2="orange"/></not>
```

<and> ブール型演算子

ほかのブール型演算子の結果の AND 論理演算を行うブール型演算子です。この演算子は属性を含まず、またほかのブール型演算子である子要素をいくつでも含むことができます。<and> 演算子は、すべての子要素が true の場合だけ、true になります。

例 **2–11** <and> ブール型演算子の使用方法

次の例に、<and> の使用方法と結果を示します。

- 次の文は true になります。

```
<and/>
```

- 次の文は true になります。

```
<and><istrue value="True"/></and>
```

- 次の文は false になります。

```
<and><equals value1="apple" value2="orange"/></and>
```

- 次の文は true になります。

```
<and>
    <matches value="apple" value2="ap*e"/>
    <istrue value="TRUE"/>
    <not><equals value1="apple" value2="orange"/></not>
</and>
```

- 次の文は false になります。

```
<and>
    <matches value="apple" value2="ap*e"/>
    <istrue value="TRUE"/>
    <equals value1="apple" value2="orange"/>
</and>
```

<or> ブール型演算子

ほかのブール型演算子の結果の OR 論理演算を行うブール型演算子です。この演算子は属性を含まず、またほかのブール型演算子である子要素をいくつでも含むことができます。<or> 演算子は、true である子要素を 1 つ以上含む場合だけ、true になります。

例 **2–12** <or> ブール型演算子の使用方法

次の例に、<or> の使用方法と結果を示します。

- 次の文は false になります。

```
<or/>
```

- 次の文は true になります。

```
<or><istrue value="True"/></or>
```

例 **2–12** <or> ブール型演算子の使用方法　(続き)

- 次の文は false になります。

```
<or><equals value1="apple" value2="orange"/></or>
```

- 次の文は false になります。

```
<or>
    <matches value="apple" value2="p*e"/>
    <istrue value="FALSE"/>
    <equals value1="apple" value2="orange"/>
</or>
```

- 次の文は true になります。

```
<or>
    <matches value="apple" value2="p*e"/>
    <not><istrue value="FALSE"/></not>
    <equals value1="apple" value2="orange"/>
</or>
```

第 3 章

コンポーネントのスキーマ

この章では、N1 Grid Service Provisioning System software コンポーネントの XML スキーマについて説明します。以下の内容を説明します。

特に断りのない限り、この章で説明する属性は、コンポーネントスコープの置換変数を参照できません。

プロビジョニングソフトウェア の XML スキーマアーキテクチャーについての概要は、第 1 章を参照してください。

<component> 要素

コンポーネントは、<component> 要素で囲まれます。1 つのコンポーネントのバージョンはすべて、同じ名前とパスを持つ必要があります。この要素の属性は、コンポーネントスコープ置換変数を参照できます。

<component> 要素には次の子要素があり、記載順で出現する必要があります。これらの子要素は、独自の子要素や属性 (両方も可能) を持つことができます。

- 90 ページの「<controlList> 要素」
- 91 ページの「<diff> 要素」

<component> 要素の属性

<component> 要素には次の属性があります。

- *xmlns* – 値 http://www.sun.com/schema/SPS を持つ必須文字列。
- *xmlns:xsi* – 値 http://www.w3.org/2001/XMLSchema-instance を持つ必須文字列。
- *xsi:schemaLocation* – オプション文字列。推奨値は http://www.sun.com/schema/SPS component.xsd です。
- *access* – オプション属性で、コンポーネントのアクセシビリティを指定します。次の 3 つの値が有効です。
  - PATH – コンポーネントを参照できるのはそのコンポーネントと同じパス内に存在するほかのコンポーネントだけです。当該コンポーネントを直接インストールすることはできず、入れ子になった参照を使用してほかのコンポーネントに当該コンポーネントを含めることしかできません。
  - PUBLIC – コンポーネントは任意のコンポーネントから参照でき、PATH では課せられる制限にも制約されません。これがデフォルト値です。
- *modifier* – modifierEnum 型のオプション値で、コンポーネントに関する次のオーバーライド要件を指定します。
  - ABSTRACT – コンポーネントを抽象コンポーネントに指定します。抽象コンポーネントは、ほかのコンポーネントを拡張するためのベースコンポーネントとしてのみ機能し、インストールできません。抽象的な子要素を宣言できるのは抽象コンポーネントだけです。
  - FINAL – コンポーネントを最終コンポーネントとして指定します。これは、そのコンポーネントは別のコンポーネントによる拡張が不可能であることを意味します。

  この属性を指定しないと、コンポーネントの拡張とインストールが行えます (これがデフォルト)。
- *name* – entityName 型の必須値で、コンポーネントの名前。
- *path* – pathName 型のオプション値で、コンポーネントの絶対パス。この属性を指定しないと、ルートパス (/) がデフォルト値になります。値の名前は、コンポーネントが保存された時点で存在するフォルダである必要があります。
- *description* – オプション文字列で、コンポーネントの説明。
- *label* – オプション文字列で、コンポーネントの簡潔な説明。
- *softwareVendor* – オプション文字列で、当該コンポーネントでモデル化されたソフトウェアアプリケーションのベンダー名。
- *author* – オプション文字列で、当該コンポーネントの作成者の名前。
- *version* – schemaVersion 型の必須値で、コンポーネントスキーマのバージョン。現在許可されている値は 5.0 のみです。

- *platform* – オプション文字列で、当該コンポーネントのインストール先として有効な物理ターゲットであるホストを含むホストセットの名前を指定します。

  この属性を指定しないと、Remote Agent を含み、サポートされているプラットフォームである、すべてのホストが有効な物理ターゲットになります。この属性を指定する場合、当該コンポーネントをインストールするプランの物理ターゲットは、指定されたホストセットに含まれるホストのサブセットでなければなりません。指定されたホストセットの一部ではないホストが物理ターゲットに含まれる場合、プランは実行時エラーを出力します。このようなプラン実行時エラーは、プリフライトエラーとして報告されます。サポートされているプラットフォームホストセットに対応しない名前を指定した場合、コンポーネント保存時エラーになります。プラットフォームホストセットにはすべて接頭辞 *system*# プラグイン名が付けられます。コンポーネントプラットフォームセットがサポートされていない場合、プラットフォームが変更されるまで、新しいバージョンのコンポーネントをチェックすることはできません。サポートされていないプラットフォームホストセットを指す、既存のコンポーネントバージョンに対する操作は失敗します。

- *limitToHostSet* – オプション文字列で、当該プランの有効な対象であるホストを含むホストセットの名前を指定します。

  この属性を指定しないと、すべてのホストが有効な対象になります。この属性を指定する場合、指定された対象は、指定のホストセットに含まれるホストのサブセットでなければなりません。指定されたホストセットの一部ではないホストが対象に含まれる場合、プランは実行時エラーを出力します。このようなプラン実行時エラーは、プリフライトエラーとして報告されます。サポートされている既存のホストセットに対応しない名前を指定した場合、コンポーネント保存時エラーが発生します。指定されたホストセットがプラグインにより定義されたホストである場合、*pluginName* は、ホストセット名に対する接頭辞 (*pluginName*# *hostSetName* など) である必要があります。

  platform 属性と *limitToHostSet* 属性では、2 つの大きな違いがあります。

  - 1 つ目の違いは、*platform* は事前定義されたプラットフォームホストセットの 1 つを指定するのに対し、*limitToHostSet* はユーザー定義のホストセットを指定することです。 したがって、カスタムホストセットに基づいてインストールを制限する必要がある場合は *limitToHostSet* を使用する必要があります。
  - 2 つ目の違いは、コンポーネントの対象を仮想ホストにした場合、*limitToHostSet* のテストはその仮想ホストに照らして行われるのに対し、*platform* のテストはその仮想ホストのルート物理ホストに照らして行われることです。

  したがって、 *limitToHostSet* を設定し *platform* は設定しないという方法をとることで、異なる物理プラットフォーム上に存在する特定の仮想ホスト (複数) にコンポーネントをインストールできます (WebLogic アプリケーションはこの方法を採用)。 ただし、*platform* を設定し *limitToHostSet* は設定しないようにすると、指定のプラットフォームを持つ物理ホストをルート (親) とする任意のホスト上にコンポーネントをインストールできます。platform、limitToHostSet の両方を設定すると、両方の範囲を制約できます。

- *installPath* – 必須文字列で、非派生コンポーネントにしか認められません。このパスは、当該コンポーネントのインストール時に使用されます。単純コンポーネントの場合、この値はリソースをインストールするルートディレクトリにも相当しま

す。当該コンポーネントのインスタンスがインストールされる際に、パスは共通書式で保存されます。53 ページの「共通イントールパスの書式」を参照してください。

installPath 属性と *limitToHostSet* 属性を除き、component 属性は継承されません。

installPath 属性は継承されます。派生コンポーネントによりオーバーライドすることはできません。しかし、ベースコンポーネントはコンポーネント変数を使用してその値を指定でき、これらの変数の値をオーバーライドできます。

ベースコンポーネントが *limitToHostSet* を指定しなかった場合だけ、*limitToHostSet* 属性は継承され、派生コンポーネントでオーバーライドできます。

limitToHostSet 値は、ユーザーによって管理される可変のエンティティを指定します。このため、ホストセットの関係をプラットフォームと同じように考えることはできません。

platform 属性は継承されません。しかし、派生コンポーネントの platform 値はベースコンポーネントの platform 値と同様に、一般的ではありません。派生コンポーネントで *platform* を指定しない場合は、*platform* はベースコンポーネントでも指定できません (あるいは `any` として指定する必要があります)。

<extends> 要素

<extends> 要素は <component> 要素の子であり、当該コンポーネントの派生元であるベースコンポーネントの宣言に使用されます。ベースコンポーネントは最終になることはできません。この要素を使用する場合、この要素は 1 回しか出現できません。

当該コンポーネントは、ベースコンポーネントの各種の属性と要素を自動的に継承します。コンポーネントは、継承されたデータの特定の部分を選択的にオーバーライドできます。継承とオーバーライドの許可は、当の属性または要素の説明内で指定します。

コンポーネントは、その拡張コンポーネントのインスタンスと言えます。また、ベースコンポーネントをインスタンスとするコンポーネントのインスタンスでもあります。

<extends> 要素には 1 つの子要素 <type> があります。この子要素は必須で、ベースコンポーネントを指定します。<type> 要素は、<component> 要素ごとに必ず 1 回使用する必要があります。

<type> 要素

<type> 要素はベースコンポーネントの型を指定します。この要素は、<extends>、<componentRefList>、および <componentRef> 要素の子です。

<*type*> 要素の属性

<type> 要素には、systemName 型の 1 つの必須属性 *name* があり、これはベース型として機能するシステム型コンポーネントの名前です。指定された型がプラグインにより定義された型である場合、*pluginName* は *pluginName*# *typeName* のように、型名に対する接頭辞である必要があります。

<varList> 要素

<varList> 要素は <component> 要素のオプションの子です。この要素は、当該コンポーネントと当該コンポーネントに含まれる構成リソースが使用するコンポーネントスコープの置換変数のリストを宣言します。この要素を使用する場合、この要素は 1 回しか出現できません。

<varList> 要素には 1 つの必須子要素 <var> があり、この子要素はコンポーネント置換変数を宣言します。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの、アクセスが可能な <varList> 要素コンテンツを継承します。派生コンポーネントが <varList> を宣言する場合、そのコンテンツはベースコンポーネントのコンテンツと完全に結合されます。派生コンポーネントは新しい <var> 要素を宣言することで継承された要素をオーバーライドできますが、ベースコンポーネントによって宣言された要素を削除することはできません。

<var> 要素

<var> 要素は、<component> 要素の子である <varlist> の子です。<var> 要素はコンポーネント置換変数を宣言します。変数の名前と、宣言する必要がある各置換変数の値デフォルト値を指定する必要があります。

デフォルトでは、派生コンポーネントはそのベースコンポーネントのすべてのアクセス可能変数 (アクセスモード、修飾子、デフォルト値、プロンプトなど) を継承します。<var> 要素は <varList> 要素内で 1 回または複数回出現できます。

派生コンポーネントは、ベースコンポーネントから継承されている変数に含まれない名前を使用することで、追加の変数を定義できます。派生コンポーネントは、同じ名前を使用して変数を宣言し直すことにより、継承された非最終変数のプロンプト、デフォルト値、修飾子、アクセスモードなどをオーバーライドできます。変数をオーバーライドする場合は、変数の全コンテンツ (プロンプト、デフォルト値、アクセスモード、修飾子など) を宣言し直す必要があります。デフォルト値は、オーバーライドする変数が非抽象である場合にのみ指定できます。アクセスモードは、ベースコンポーネントのモードよりも厳しくすることはできません。

変数をオーバーライドすると、ベースコンポーネント内の参照も含め、その変数に対する参照はすべてオーバーライドされた値に評価されます。

派生コンポーネントを非抽象と宣言する場合、ベースコンポーネントによって宣言された抽象変数は派生コンポーネントによってオーバーライドする必要があります。

<var> 要素の属性

<var> 要素には次の属性があります。

- *access* – accessEnum 型のオプション値で、変数のアクセシビリティを指定します。この属性は次の値を取ることができます。
  - PUBLIC – アクセスはまったく制限されません。これがデフォルトです。
  - PROTECTED – アクセスは同じパス内の派生コンポーネントとエンティティに制限されます。
  - PATH – アクセスは同じパス内のエンティティに制限されます。
  - PRIVATE – アクセスは当該コンポーネントに制限されます。
- *modifier* – modifierEnum 型のオプション値で、変数のオーバーライド要件を指定します。この属性は次の値を取ることができます。
  - ABSTRACT – 変数の *default* 属性は省略されます。このため、この属性は 非抽象派生コンポーネントによって指定する必要があります。変数を抽象と宣言できるのは、コンポーネントも抽象と宣言されている場合だけです。抽象変数は専用には設定できません。非抽象変数は、デフォルト値を宣言する必要があります。
  - FINAL – 変数を派生コンポーネントによってオーバーライドすることはできません。

  この属性を指定しない場合、派生コンポーネントは変数をオーバーライドするかどうかを選択できます。
- *name* – identifier 型の必須値で、置換変数の名前。<varList> 要素内の <var> 要素により宣言される各変数名は一意でなければなりません。
- *default* – 非抽象変数用の必須文字列で、置換変数のデフォルト値。この値には、ほかの置換変数、セッション変数、ターゲットホスト属性、インストールされているコンポーネント変数に対する参照を含めることができます。ただし、抽象変数はデフォルト値を定義できないため、この属性は抽象変数には使用できません。
- *prompt* – オプション文字列で、ユーザーが読める形式での変数の説明。

<targetRef> 要素

<targetRef> 要素は、<component> 要素のオプションの子です。この要素は、コンポーネントが対象設定可能であることを宣言します。対象設定可能コンポーネントは、インストール時にコンポーネントと関連付けられる物理ホストまたは仮想ホストを自動的に作成するコンポーネントです。この要素は、使用する際には、<varList> 要素の直後に 1 回だけ出現できます。この要素は、単純コンポーネントと複合コンポーネントの両方で使用できます。ただし、<varList> 要素は、最上位コンポーネントとしてインストールされているコンポーネントでのみ使用できます。

<targetRef> 要素には、オプションの子要素 <agent> があり、これは関連付けられたホストが物理ホストと仮想ホストのどちらであるかを示します。<agent> 要素が存在する場合、ホストは物理ホストです。要素の本体は、Remote Agent の構成を定義します。この要素が存在しない場合、ホストは仮想ホストです (これがデフォルト)。

<targetRef> 要素の属性

<targetRef> 要素には次の属性があります。

- *hostName* – 必須値で、当該コンポーネントのインストール時に作成するホスト名。この名前はインストール時に一意であり、また有効なホスト名でなければなりません。この値には、コンポーネントスコープの置換変数参照を含めることができます。
- *typeName* – systemName 型のオプション値で、関連付けられたホストに使用するホスト型の名前。コンポーネントの保存時には、指定された型が存在する必要があります。このホスト型がプラグインによって定義されている場合、その名前には「*pluginName* # *typeName*」のように、接頭辞として *pluginName* が含まれている必要があります。

  この属性を指定しないと、値は system#crhost になります。

<agent> 要素

<agent> 要素は、<targetRef> 要素の子です。この要素は、関連付けられたホストが物理ホストであることを示します。この要素はオプションで、1 回しか使用できません。

この要素を指定しないと、関連付けられるホストは仮想ホストとして作成されます。この要素を使用する場合、関連付けられるホストは物理ホストとして作成され、<agent> 要素が、関連付けられる Remote Agent の構成を指定します。

この要素は派生コンポーネントによって継承されます。ベースコンポーネントが宣言を行っていない場合にのみ、派生コンポーネントはローカルな <targetRef> 要素を宣言できます。これは、派生コンポーネントが、継承された <targetRef> 要素をオーバーライドできないことを意味します。

<agent> 要素の属性

<agent> 要素には次の属性があります。これらの属性は、コンポーネントスコープの置換変数を参照できます。

- *connection* – 必須値で、Remote Agent への接続に使用される接続型を指定します。この属性は次のいずれかの値を取ることができます。
  - RAW
  - SSL

- SSH

- *ipAddr* – 必須値で、物理ホストの IP アドレス。この値には、サーバー名と IP アドレスのいずれかを使用できます。サーバー名は、Master Server によって IP アドレスに解決できる必要があります。
- *port* – オプション値で、Remote Agent の監視対象であるポート。*connection* が RAW または SSL である場合、*port* のデフォルト値は 1131 です。*connection* が SSH である場合、この属性は無視されます。
- *params* – オプション値で、Remote Agent への接続に使用される最低 1 つのパラメータ。

<resourceRef> 要素

<resourceRef> 要素は <component> 要素のオプションの子であり、当該コンポーネントによって管理されるリソースを指定します。この要素を使用できるのは単純コンポーネントのみです。この要素は、複合コンポーネントのみが使用できる<componentRefList> 要素とは併用できません。この要素およびその子の構成可能属性は、コンポーネント置換変数を参照できます。リソースは、暗黙の PUBLIC アクセスモードを取ります。この要素を使用する場合、この要素は 1 回しか出現できません。

コンポーネントが単純コンポーネントから派生する場合、あるいは <resourceRef> 要素を含む非派生コンポーネントである場合、そのコンポーネントは単純コンポーネントになります。派生コンポーネントが単純コンポーネントから派生している場合、その派生コンポーネントに含めることができるのは <resourceRef> 要素だけです。

<resourceRef> 要素には子要素があり、それらは次の順序で出現する必要があります。

- <installSpec> – 非派生コンポーネント用の必須要素で、リソースのインストール方法を指定します。この要素を派生コンポーネントに含めることはできません。
- <resource> – 非抽象コンポーネント用の必須要素で、関連付けられたリソースを特定します。この要素を抽象的なコンポーネントに含めることはできません。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの <resourceRef> 要素を継承します。

派生コンポーネントは、<resourceRef> 要素を宣言し直すことによって、継承された非最終 <resourceRef> 要素の修飾子と <resource> 要素をオーバーライドできます。<resourceRef> 要素がオーバーライドされる際に、<installSpec> 要素は除外されます。これはそのコンテンツをオーバーライドすることができないためです。<resource> 要素が指定されるのは、優先する <resourceRef> が抽象でない場合だけです。

<resourceRef> がオーバーライドされる際に、ベースコンポーネント内の使用も含め、リソースの使用 (<deployResource>、 <addResource> など) はすべてオーバーライドされた値に解決されます。

派生コンポーネントが非抽象と宣言された場合で、ベースコンポーネントの <resourceRef> 要素が抽象のとき、その派生コンポーネントは <resourceRef> 要素をオーバーライドする必要があります。

<resourceRef> 要素の属性

<resourceRef> 要素には 1 つのオプション属性 *modifier*があり、この属性は modifierEnum 型の値を持っています。*modifier* 属性は次のように、リソースのオーバーライド要件を指定します。

- ABSTRACT – <resourceRef> の <resource> 要素は省略されます。このため、この要素は非抽象派生コンポーネントによって指定する必要があります。<resourceRef> を抽象と宣言できるのは、コンポーネントも抽象と宣言されている場合だけです。非抽象 <resourceRef> は、<resource> 要素を宣言する必要があります。
- FINAL – 派生コンポーネントで <resourceRef> をオーバーライドすることはできません。

この属性を指定しない場合、派生コンポーネントは <resourceRef> をオーバーライドするかどうかを選択できます。

<installSpec> 要素

<installSpec> 要素は <resourceRef> 要素の子であり、関連付けられたリソースのインストール方法を指定します。この要素は派生コンポーネントによって継承され、オーバーライドできません。しかし、ベースコンポーネントは <installSpec> 属性の値指定にコンポーネント変数を使用でき、これらの変数の値はオーバーライドできます。

<installSpec> 要素の属性

<installSpec> 要素には次の属性があります。これらの属性は、コンポーネントスコープの置換変数を参照できます。

- *name* – 必須文字列で、リソースのインストール時にリソースに使用する名前。
- *path* – オプション文字列で、リソースのインストール先となるパス。相対ディレクトリは、包含コンポーネントの *installPath* 属性に相対的であると見なされます。この引数を指定しないと、デフォルトでコンポーネントの *installPath* 属性が使用されます。
- *permissions* – オプション文字列で、インストール時に当該リソースに割り当てるアクセス許可を示します。

  文字列の書式は、UNIX chmod コマンドでコマンドで定義されているように 3 桁の 8 進数を使用します。chmod(1M) のマニュアルページを参照してください。この属性を指定しないと、リソースはデフォルトのアクセス許可でインストールされます。

- *user* – オプション文字列で、インストール時の当該リソースの所有者。この属性を指定しないと、ユーザーはプラン実行機能 (plan executor) によって決定されます。
- *group* – オプション文字列で、インストール時に当該リソースに割り当てるグループ。この属性を指定しないと、グループはプラン実行機能 (plan executor) によって決定されます。
- *deployMode* – オプション属性で、関連付けられたディレクトリリソースの配備方法を指定します。リソースがディレクトリでない場合、この属性は無視されます。
  - `ADD_TO` – 対象ディレクトリ内の任意の既存ファイルにディレクトリコンテンツが追加されます。
  - `REPLACE` – 対象ディレクトリ内のすべての既存ファイルがディレクトリコンテンツによって置き換えられます。

  この引数を指定しない場合、デフォルト値 `REPLACE` が使用されます。
- *diffDeploy* – `boolean` 型のオプション値で、リソースを差分配備モードで配備すべきかどうかを指定します。この属性を指定しないと、差分配備モードが無効になります。差分配備モードを有効にした場合、それまでに配備されたことがないリソースだけが配備されます。

<resource> 要素

<resource> 要素は <resourceRef> 要素の子であり、当該コンポーネントによって配備されるリソースを特定します。

参照されたリソースが構成可能なリソースである場合、包含コンポーネントにアクセス可能な任意のコンポーネントスコープ変数に対する置換変数参照を含むことができます。

<resource> 要素の属性

<resource> 要素には次の属性があります。

- *name* – 必須文字列で、チェックインコンポーネントに含まれるリソースの完全なパス名。
- *version* – `version` 型の必須値で、以前のコンポーネントチェックインにより作成されたリソースのバージョン。

<componentRefList> 要素

<componentRefList> 要素は <component> 要素のオプションの子であり、当該コンポーネントによって参照されるコンポーネントの一覧を指定します。この要素は、<resourceRef> 要素とは併用できません。この要素およびその子の構成可能属性は、コンポーネント置換変数を参照できます。この要素を使用する場合、この要素は 1 回しか出現できません。

コンポーネントが複合コンポーネントから派生する場合、あるいは <resourceRef> 要素を含まない非派生コンポーネントである場合、そのコンポーネントは複合コンポーネントになります。派生コンポーネントが複合コンポーネントから派生している場合、その派生コンポーネントに含めることができるのは <componentRefList> 要素だけです。

<componentRefList> 要素には、次のオプションの子要素があります。

- <type> – すべての参照先コンポーネントがインスタンスとなるべき型を指定します。この要素を指定しないと、参照先コンポーネントは任意の型になります。

  この要素はオプションです。この要素を使用する場合、この要素は <componentRefList> 要素 1 つにつき 1 回しか出現できません。
- <componentRef> – コンポーネントに対する参照。この要素はオプションです。この要素を使用する場合、この要素は複数回出現することができます。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの <componentRefList> 要素のコンテンツを継承します。派生コンポーネントが <componentRefList> を宣言する場合、そのコンテンツはベースコンポーネントのコンテンツと完全に結合されます。派生コンポーネントは、新しい <componentRef> 要素を宣言することで継承された要素をオーバーライドできます。ベースコンポーネントによって宣言された要素を削除することはできません。

派生コンポーネントは、親コンポーネントの <componentRefList> 要素によって宣言されている <type> 要素をオーバーライドできます。このためには、<componentRefList> 内の <type> 属性を宣言し直します。この場合、オーバーライドされる型は、オリジナルの型のインスタンスであるか、あるいはオリジナルの型が指定されている必要があります。さらに、すべての参照先コンポーネント (ベースコンポーネントから継承されるものを含む) はオーバーライドされる型のインスタンスでなければなりません。

<componentRefList> 要素の属性

<componentRefList> 要素には 1 つのオプション属性 *modifier* があり、これはリソースのオーバーライド要件を指定します。この属性を指定する場合、値は FINAL でなければなりませんが、これは派生コンポーネントが新しい <componentRef> 要素を宣言できないことを意味します。

この属性を指定しない場合、派生コンポーネントは新しい <componentRef> 要素を追加できます。どちらの場合も、派生コンポーネントは <type> 要素と非最終継承 <componentRef> 要素をオーバーライドできます。ベースコンポーネントの <componentRefList> *modifier* 属性が FINAL である場合、派生コンポーネントの *modifier* 属性も FINAL である必要があります。

<componentRefList> 要素の *modifier* が FINAL である場合、含まれる各 <componentRef> の *modifier* 属性は必ずしも FINAL ではありません。

<componentRef> 要素

<componentRef> 要素は <componentRefList> 要素の子で、当該コンポーネントによって参照されるコンポーネントを指定します。この要素は、暗黙的に PUBLIC アクセスとなります。

<componentRef> 要素には子要素があり、それらは次の順序で出現する必要があります。

- <type> – 参照先コンポーネントがインスタンスとなるべきコンポーネント型を指定します。これは、包含する <componentRefList> によって指定されるコンポーネント型のインスタンスでなければなりません。

  この要素はオプションです。この要素を使用する場合、この要素は <componentRef> 要素 1 つにつき 1 回しか出現できません。

  この要素を指定しないと、包含する <componentRefList> によって指定されたコンポーネント型が使用されます。

- <argList> – オプション要素で、参照先コンポーネントのインストール時にそれらのコンポーネント変数設定として使用される値の一覧。

  この要素を使用する場合、この要素は <componentRef> 要素の子としてのみ 1 回しか出現できません。

- <component> – 必須要素で、参照先コンポーネントを指定します。この要素は、抽象 <componentRef> 要素では使用できません。

デフォルトでは、派生コンポーネントはそのベースコンポーネントのすべてのコンポーネント参照を継承します。

ベースコンポーネントの <componentRefList> 要素が最終でない場合、派生コンポーネントはベースコンポーネントから継承されたコンポーネント参照に使用されていない名前を使用して、別のコンポーネント参照を定義できます。

派生コンポーネントは、同じ名前を持つコンポーネント参照を宣言し直すことにより、非最終継承コンポーネント参照のコンポーネント参照をオーバーライドできます。コンポーネント参照がオーバーライドされた場合には、コンポーネント参照の全コンテンツを宣言し直す必要があります。オーバーライドする *installMode* は、オリジナルの参照のものと同じでなければなりません。オーバーライドする <type> 要素は、オリジナルの型のインスタンスでなければなりません。オーバーライドする <argList> 要素は、オリジナルのものと結合されます。詳細は、76 ページの「<argList> 要素」を参照してください。<component> 要素は、オーバーライドする参照が非抽象の場合にのみ指定されます。

コンポーネント参照をオーバーライドすると、ベースコンポーネント内のものも含め、そのコンポーネント参照の使用はすべてオーバーライド値に評価されます。

派生コンポーネントを非抽象と宣言する場合、ベースコンポーネントによって宣言された抽象的なコンポーネント参照は、派生コンポーネントによってオーバーライドする必要があります。

<componentRef> 要素の属性

<componentRef> 要素には次の属性があります。

- *modifier* – modifierEnum 型のオプション属性で、コンポーネント参照のオーバーライド要件を指定します。この属性は次の値を取ります。
  - ABSTRACT – <componentRef> 要素の <component> 子要素は省略されるため、非抽象派生コンポーネントによって指定する必要があります。<componentRef> を抽象と宣言できるのは、そのコンポーネントも抽象と宣言されている場合だけです。非抽象 <componentRef> は、<component> 要素を宣言する必要があります。
  - FINAL – <componentRef> は派生コンポーネントによってはオーバーライドできません。

  この属性を指定しない場合、派生コンポーネントはコンポーネント参照をオーバーライドするかどうかを選択できます。
- *name* – identifier 型の必須属性で、参照先コンポーネントのローカル名を指定します。この名前は、すべての兄弟 <componentRef> 要素の中で一意となるようにする必要があります。
- *installMode* – オプション属性で、参照先コンポーネントをインストールし、その後対象設定を行う方法を指定します。この属性を指定しないと、値は NESTED になります。

  この属性は次の値を取ります。
  - TOPLEVEL – 参照先コンポーネントが TOPLEVEL としてインストールされている場合、1 つのプランで直接インストールされたかのように、ほかの任意のコンポーネントがそのコンポーネントを使用できます。
  - NESTED – 参照先コンポーネントが NESTED としてインストールされている場合、そのインストールの範囲は参照元コンポーネントのインストール範囲に暗黙に限定されます。そのサービスも参照元コンポーネントでしか利用できません。

  論理的に、入れ子になった参照先コンポーネントは参照元コンポーネントが要求する細かな機能単位を定義しますが、この機能はほかのコンポーネントに役立つことはありません。一方、最上位の参照先コンポーネントは参照元コンポーネントが使用するサービスを定義しますが、ほかのコンポーネントもこのサービスを使用できます。

  入れ子になった参照先コンポーネントの有効期間は、参照元コンポーネントの有効期間と暗黙に同じになります。入れ子になった参照先コンポーネントのインストールは参照元コンポーネントのインストールの最中にしか行えず、参照元コンポーネントがアンインストールされる際に暗黙にアンインストールされます。これに対し、最上位の参照先コンポーネントの有効期間は、参照元コンポーネントの有効期間には拘束されていません。最上位の参照先コンポーネントは、参照元コンポーネントのインストール時に参照元コンポーネントによってインストールすることができます。また、最上位の参照先コンポーネントがすでにインストールされている場

合は、ほかの方法でインストールすることもできます。参照元コンポーネントがアンインストールされる場合、明示的に参照元コンポーネントによってアンインストールされない限り、最上位の参照先コンポーネントはインストールされたままとなります。 また、ほかのコンポーネントも参照元コンポーネントをアンインストールできます。

<targetRef> 要素を定義するコンポーネントを参照するには、TOPLEVEL <componentRef> を使用する必要があります。NESTED <componentRef> は使用できません。

<argList> 要素

<argList> 要素は <componentRef> 要素の子で、参照先コンポーネントのインストール時に、そのコンポーネント変数設定として使用される値の一覧を指定します。この <argList> の書式は、 <call> ステップの <argList> 子要素の書式と同じです。詳細は、26 ページの「<call> ステップ」を参照してください。

参照が ABSTRACT の場合、<argList> 要素の各属性は、参照先コンポーネントにおけるコンポーネント変数または宣言された型を指定します。<argList> 要素の属性の値は、参照先コンポーネントのインストール時に、指定されたコンポーネント変数に使用されるオーバーライド値です。

コンポーネント参照が派生コンポーネントによってオーバーライドされる場合、ベースコンポーネントおよび派生コンポーネントの <argList> 要素は、効率よく結合されます。この結合は、ベースコンポーネントの <argList> のコンテンツを参照先コンポーネントに適用し、続いて派生コンポーネントの <argList> を適用することによって行われます。ベースコンポーネント参照の <argList> を処理する場合、考慮されるのはベースコンポーネント参照の宣言された型で定義された変数だけです。

<argList> で指定されていないコンポーネント変数は、インストール時にそれらのデフォルト値を使用します。<argList> が指定されていない場合、参照先コンポーネントがインストールされ、またそのすべての変数のデフォルト値を使用します。<argList> で指定される参照先コンポーネントの変数は、参照元コンポーネントからアクセス可能でなければなりません。また、これらの変数はアクセスモード FINAL ではなく、PUBLIC または PROTECTED で宣言されていなければなりません。

最上位の参照先コンポーネントの場合、<argList> 要素が使用されるのは参照先コンポーネントが参照元コンポーネントによってインストールされている場合だけです。参照先コンポーネントは、ほかの方法でインストールされていることもあります。この場合、<argList> は意味を持ちません。

<component> 要素

<component> 要素は <componentRef> 要素の子で、参照先コンポーネントを特定します。この要素の構造は、*host* 属性が許可されない点を除き、<component> リポジトリコンポーネントターゲッターと同じです。参照先コンポーネントバージョンは、包含コンポーネントの保存時にリポジトリ内に存在する必要があります。

version 属性を指定しないと、version は包含コンポーネントの保存時に存在する参照先コンポーネントの最新バージョンに解決されます。参照先コンポーネントのバージョンが存在しない場合、保存時のエラーが発生します。包含コンポーネントが保存されると、当該コンポーネントが参照するすべてのコンポーネントのバージョンがロックされます。参照先コンポーネントは、コンテナコンポーネントの新しいバージョンを作成しない限り変更できません。

<installList> 要素

<installList> 要素は <component> 要素の子であり、1 つ以上の名前付きインストールステップブロックを含みます。各ブロックは、当該コンポーネントをインストールする個々の方法を提供します。多くのコンポーネントには、<installSteps> 要素の子としてインストールブロックが1 つだけ存在します (この節で後述)。

この要素は、非派生コンポーネントの場合は必須で、派生コンポーネントの場合はオプションです。この要素を使用する場合、この要素は 1 回しか出現できません。

インストール環境ごとに異なるステップが必要な場合は、複数のインストールブロックを使用できます。たとえば、EBJ アプリケーションをサーバークラスタに配備するインストールブロック、単一の管理対象サーバーに配備する別のインストールブロック、最初のインストール用のインストールブロック、およびアプリケーションをアップグレードするインストールブロックを作成できます。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの、アクセスが可能な <installList> 要素コンテンツを継承します。派生コンポーネントが <installList> を宣言する場合、そのコンテンツはベースコンポーネントのコンテンツと完全に結合されます。派生コンポーネントは新しい <installSteps> 要素を宣言することで継承された要素をオーバーライドできますが、ベースコンポーネントによって宣言された要素を削除することはできません。

<installList> 要素には 1 つの子要素 <installSteps> があり、この子要素は、当該コンポーネントをインストールするために実行されるステップのシーケンスを一覧表示します。<installSteps> 要素は、1 回または複数回出現することができます。

<installSteps> 要素

<installList> 要素には 1 つの子要素 <installSteps> があり、この子要素は、当該コンポーネントをインストールするために実行されるステップのシーケンスを一覧表示します。<install> ステップによって当該コンポーネントがインストールされる場合、ここに挙げるステップが順に実行されます。一般に、単純コンポーネントのインストールステップには <deployResource> ステップが含まれます。複合コンポーネントのインストールステップには、参照先コンポーネントをインストールするため、1 つ以上の <install> ステップが含まれます。

<installSteps> 要素の子は、オプションの <paramList> 要素から構成され、このあとには本体が続き、これはオプションのローカル <varList> 要素から構成されています。ローカル <varList> 要素には、ゼロ個以上の「共通」ステップまたは「コンポーネントのインストール専用」ステップが続きます。インストールブロックが抽象と宣言されている場合、本体は含められません。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの、アクセス可能なすべてのインストールブロックを継承します。

派生コンポーネントは、ベースコンポーネントから継承されたインストールブロックに含まれない名前を使用して別のインストールブロックを定義できます。派生コンポーネントは、同じ名前を使用してブロックを宣言し直すことにより、継承した非最終インストールブロックをオーバーライドできます。ブロックの無効化は名前を使用してしか行えず、パラメータに基づいてオーバーロード(多重定義) することはできません。ブロックをオーバーライドする場合は、ブロックの全コンテンツ (アクセスモード、修飾子、およびパラメータ) を宣言し直す必要があります。本体を指定できるのは、優先するブロックが非抽象の場合だけです。アクセスモードは、ベースコンポーネントのモードよりも厳しくすることはできません。

派生コンポーネント内の優先ブロックのシグニチャーは、ベースコンポーネントのシグニチャーと互換性がなければなりません。互換性があるということは、ベースブロックが使用できる引数はすべて派生ブロックでも使用できることを意味します。

派生ブロックがベースブロックと互換性があるのは、必須パラメータを新たに宣言せず、親ブロックでオプションパラメータを必須パラメータとして再定義していない場合です。

以下のシグニチャー変更は互換性があると見なされます。

- 必須パラメータまたはオプションパラメータの削除
- 必須パラメータのオプションパラメータへの変更
- オプションパラメータの追加

ブロックをオーバーライドすると、ベースコンポーネント内の参照も含め、そのブロックに対する参照はすべてオーバーライドされた値に評価されます。

派生コンポーネントを非抽象と宣言する場合は、ベースコンポーネントによって宣言されているすべての抽象ブロックを、派生コンポーネントによってオーバーライドする必要があります。

<superComponent> ターゲッターを使用して派生コンポーネントがブロックをオーバーライドする場合でも、派生コンポーネント内のブロックは、ベースコンポーネントによって定義されているブロックを明示的に呼び出すことができます。

<installSteps> 要素の属性

<installSteps> 要素には次の属性があります。

- *access* – accessEnum 型のオプション属性で、インストールブロックのアクセシビリティを指定します。次の値が使用できます。

- PUBLIC – アクセスはまったく制限されません。デフォルトのアクセスモードです。
- PROTECTED – アクセスは同じパス内の派生コンポーネントとエンティティに制限されます。
- PATH – アクセスは同じパス内のエンティティに制限されます。
- PRIVATE – アクセスは当該コンポーネントに制限されます。

コンポーネントから直接実行できるのは PUBLIC ブロックだけです。

- *modifier* – ModifierEnum 型のオプション属性で、インストールブロックのオーバーライド要件を指定します。次の値が使用できます。
  - ABSTRACT – ブロックに本体を含めることはできません。本体は、非抽象派生コンポーネントによって指定する必要があります。インストールブロックを抽象と宣言できるのは、コンポーネントも抽象と宣言されている場合だけです。抽象ブロックは専用にはできません。非抽象ブロックの場合、本体を宣言する必要があります。
  - FINAL – インストールブロックを派生コンポーネントによってオーバーライドすることはできません。

  modifier 属性を指定しない場合、派生コンポーネントはブロックをオーバーライドするかどうかを選択できます。
- *name* – entityName 型の必須属性で、インストールブロックの名前。この名前は、包含している <installList> 内のすべてのインストールブロックの中で一意である必要があります。
- *description* – オプション属性で、インストールブロックを説明する文字列。この属性は文書化に便利です。

<paramList> 要素

<paramList> 要素は、<installSteps>、<uninstallSteps>、<snapshot>、および <control> 要素の子です。この要素は、包含要素のステップで使用できるパラメータの一覧を宣言します。パラメータの値は、呼び出し側の <argList> 要素のコンテンツに基づいて呼び出し側によって定義されます。たとえば、<installSteps> ブロック内の <paramList> の場合、パラメータ値は <installSteps> ブロックを呼び出した <install> ステップの <argList> に基づいて定義されます。

包含要素のステップは、次の変数とパラメータを使用できます。

- ローカル <varList> 要素で宣言されたローカルスコープ変数
- <paramList> 要素で宣言されたパラメータ
- 包含コンポーネントのコンポーネント <varList> 要素で宣言されたコンポーネントスコープ変数

<paramList> パラメータの名前がコンポーネント <varList> 変数と同じである場合、パラメータの値が使用されます。このような場合、パラメータはコンポーネント変数を「隠蔽」していると言われます。隠蔽は、ローカル変数とパラメータの間では許可されません。これは、それらの名前が個別のものでなければならないためです。

<paramList> 要素には 1 つの必須子要素 <param> があります。この子要素はパラメータ宣言で、名前とデフォルト値が含まれます。定義する必要があるパラメータごとに、1 つの <param> 要素を指定します。

<param> 要素

<param> 要素は <paramList> 要素の子で、(名前、デフォルト値が含まれる) パラメータを宣言します。デフォルト値が使用されるのは、呼び出し側がこのパラメータの値を明示的に渡さない場合だけです。デフォルト値が指定されず、呼び出し側がこのパラメータの値を明示的に渡さない場合、プランの実行時にプリフライトエラーが発生します。

<param> 要素の属性

<param> 要素には次の属性が含まれます。包含しているインストールブロック、アンインストールブロック、または制御ブロックが、プランまたはほかのコンポーネントからではなくユーザーによって直接呼び出される場合に、*prompt* 属性および *displayMode* 属性を使用します。

- *name* – identifier 型の必須属性で、パラメータの名前。この名前は、包含要素によって宣言されているほかのすべてのローカル変数とパラメータにおいて一意でなければなりません。
- *prompt* – オプション属性で、パラメータの値を求める際にユーザインタフェースで表示されるテキストを指定する文字列。この属性を指定しないと、*name* の値が使用されます。
- *default* – オプション属性で、パラメータのデフォルト値を指定する文字列。パラメータには、コンポーネント変数、ターゲットホスト属性、セッション変数、およびインストール済みコンポーネント変数への参照を含めることができますが、ほかのパラメータへの参照を含めることはできません。
- *displayMode* – オプション属性で、パラメータの表示モードを指定します。次の値が有効です。
  - PASSWORD – ユーザー指定の値が隠されます。つまりパスワードは表示されないか、アスタリスクに置き換えられます。
  - BOOLEAN – チェックボックスを使用してパラメータが指定されます。
  - CLEAR – 入力時に値が表示されます。

値が CLEAR または BOOLEAN の場合は、入力時に安全に表示されます。この属性を指定しないと、値は CLEAR になります。

ローカル *<varList>* 要素

ローカル <varList> 要素は、<installSteps>、<uninstallSteps>、<snapshot>、および <control> 要素の子です。この要素は、包含要素のステップで使用できる変数の一覧を宣言します。これらの変数の値は宣言時に定義され、再定義は行えません。

包含要素のステップは、次の変数とパラメータを使用できます。

- ローカル <varList> 要素で宣言されたローカルスコープ変数
- <paramList> 要素で宣言されたパラメータ
- 包含コンポーネントのコンポーネント <varList> 要素で宣言されたコンポーネントスコープ変数

ローカル <varList> 変数の名前がコンポーネント <varList> 変数と同じである場合、ローカル変数の値が使用されます。このような場合、ローカル変数はコンポーネント変数を「隠蔽」していると言われます。隠蔽は、ローカル変数とパラメータの間では許可されません。これは、それらの名前が個別のものでなければならないためです。

ローカル <varList> 要素には 1 つの必須子要素 <var> があります。この要素は、ローカル変数 (名前、デフォルト値など) の宣言です。複数の <var> 要素を指定できます。

ローカル <var> 要素
ローカル <var> 要素は、ローカル <varList> 要素の必須子要素で、ローカル変数名とその値を宣言するために使用されます。

ローカル <var> 要素の属性
ローカル <var> 要素には次の属性があります。

- *name* – identifier 型の必須属性で、ローカル変数の名前を指定します。この名前は、包含要素によって宣言されているほかのすべてのローカル変数とパラメータにおいて一意でなければなりません。
- *default* – String 型の必須属性で、ローカル変数のデフォルト値。このローカル変数は、次の参照を含むことができます。
  - 先に宣言されているほかのローカル変数
  - パラメータ
  - コンポーネント変数
  - ターゲットホスト属性
  - セッション変数
  - インストール済みのコンポーネント変数

<uninstallList> 要素

<uninstallList> 要素は <component> 要素の子です。この要素は 1 つ以上の名前付き <uninstall> ステップブロックを含み、 各ブロックは、当該コンポーネントをアンインストールする個々の方法を提供します。多くのコンポーネントには、この要素の子としてアンインストールブロックが 1 つだけ存在します。インストール環境ごとに異なるステップが必要な場合は、複数のアンインストールブロックを使用できます。

たとえば、EBJ アプリケーションの配備をサーバークラスタから解除するアンインストールブロックや、単一の管理対象サーバーから解除するアンインストールブロックなどが作成できます。アンインストールブロックは、インストールブロックと一対一で対応していることが少なくありません。このような場合には、慣習上同じ名前を使用して対応を示してください。

この要素は、非派生コンポーネントの場合は必須で、派生コンポーネントの場合はオプションです。この要素を使用する場合、この要素は 1 回しか出現できません。

<uninstallList> 要素には 1 つの必須子要素 <uninstallSteps> があります。この子要素は、当該コンポーネントをアンインストールするために実行できるステップを含む、名前付きのアンインストールブロックです。コンポーネントをアンインストールする各方法に対して、それぞれ 1 つの <uninstallSteps> 要素を指定します。

デフォルトでは、派生コンポーネントはベースコンポーネントのアクセス可能な <uninstallList> 要素コンテンツを継承します。派生コンポーネントが <uninstallList> を宣言する場合、そのコンテンツはベースコンポーネントのコンテンツと完全に結合されます。派生コンポーネントは新しい <uninstallSteps> 要素を宣言することで継承された要素をオーバーライドできますが、ベースコンポーネントによって宣言された要素を削除することはできません。

<uninstallSteps> 要素

<uninstallSteps> 要素は <uninstallList> 要素の子であり、当該コンポーネントをアンインストールするために実行される一連のステップを示します。<uninstall> ステップによって当該コンポーネントがアンインストールされる場合、この要素に挙げられるステップが順に実行されます。単純コンポーネントの <uninstallSteps> 要素は <undeployResource> ステップを含むことが許可されていますが、これは必須ではありません。複合コンポーネントの <uninstallSteps> 要素は、参照先コンポーネントをアンインストールするため、1 つ以上の <uninstall> ステップが含むことができますが、これらのステップは必須ではありません。

<uninstallSteps> 要素の子は、オプションの <paramList> 要素とそれに続く本体から構成されます。この本体は、オプションのローカル <varList> 要素と、それに続くオプションの <dependantCleanup> ブロックおよびゼロ個以上の「共通」ステップまたは「コンポーネントのアンインストール専用」ステップから構成されます。アンインストールブロックが抽象と宣言されている場合、本体は含められません。

次の例に、サンプル <uninstallSteps> 要素のコンテンツを示します。</paramList> 以降が本体を定義しています。

```
<uninstallSteps name="default">
  <paramList>
     <param name="param1"/>
   </paramList>
  <varList>
```

```
    <var name="var1" default="my var 1"/>
  </varList>
  <dependantCleanup>
    <uninstall blockName="default">
        <allDependants name="child2parent"/>
    </unisntall>
  </dependantCleanup>
  <undeployResource/>
</uninstallSteps>
```

デフォルトでは、派生コンポーネントはそのベースコンポーネントの、アクセス可能なすべてのアンインストールブロックを継承します。アンインストールブロックをオーバーライドするセマンティクスは、インストールブロックをオーバーライドするセマンティクスと同じです。

<uninstallSteps> 要素の属性

<uninstallSteps> 要素には次の属性があります。

- *access* – accessEnum 型のオプション属性で、アンインストールブロックのアクセシビリティを指定します。次の値が使用できます。
  - PUBLIC – アクセスは制限されません。これがデフォルトです。
  - PROTECTED – アクセスは同じパス内の派生コンポーネントとエンティティに制限されます。
  - PATH – アクセスは同じパス内のエンティティに制限されます。
  - PRIVATE – アクセスは当該コンポーネントに制限されます。

  コンポーネントから直接実行できるのは PUBLIC ブロックだけです。
- *modifier* – modifierEnum 型のオプション属性で、アンインストールブロックのオーバーライド要件を指定します。次の値が使用できます。
  - ABSTRACT – ブロックに本体を含めることはできません。 本体は、非抽象派生コンポーネントによって指定する必要があるためです。アンインストールブロックを抽象と宣言できるのは、コンポーネントも抽象と宣言されている場合だけです。抽象ブロックは専用にはできません。非抽象ブロックの場合、本体を宣言する必要があります。
  - FINAL – アンインストールブロックを派生コンポーネントによってオーバーライドすることはできません。

  この属性を指定しない場合、派生コンポーネントはブロックをオーバーライドするかどうかを選択できます。
- *name* – entityName 型の必須属性で、アンインストールブロックの名前。この名前は、包含している <uninstallList> 内のすべてのアンインストールブロックの中で一意である必要があります。
- *description* – オプション属性で、アンインストールブロックを説明する文字列。この属性は文書化に便利です。

<dependantCleanup> 要素

<dependantCleanup> 要素は <uninstallSteps> 要素の子です。<dependantCleanup> 要素は、呼び出し側コンポーネントに現在依存しているコンポーネントを削除するために実行される一連のステップを指定します。この要素に属性はなく、包含するアンインストールブロックのスコープ内で許可される任意の数のステップを含むことができます。

この要素を含めると、依存コンポーネントのチェックはブロックのコンテンツの実行が完了するまで延期されます。ブロックの実行が完了したあとも依存コンポーネントが存在する場合は、アンインストールが失敗し、コンポーネントはインストールされたままとなります。依存コンポーネントが残っていない場合、アンインストールが継続されて残りのステップが行われます。

包含コンポーネントが対象設定可能である場合、ブロックを使用して、ホストを対象とする関連付けられたコンポーネントにインストールされたコンポーネントを削除できます。このブロックが完了したあとも、関連付けられているホスト上にインストール済みコンポーネントが残っている場合、アンインストールは失敗します。

アンインストールブロックに <dependantCleanup> ブロックを含めないと、依存コンポーネントが存在する場合、アンインストールブロックはただちに停止します。

<dependantCleanup> ブロックは、依存コンポーネントをまとめてアンインストールする目的で、し<allDependants> ターゲッターと併用されることもあります。

<snapshotList> 要素

<snapshotList> 要素は <component> 要素のオプションの子です。この要素には、1 つ以上の名前付きスナップショットブロックが含まれます。各ブロックは、ターゲットホスト上における当該コンポーネントのインストール状態をキャプチャする個別の方法を提供します。複数のスナップショットブロックを使用してインストール状態のさまざまな情報をキャプチャできるため、キャプチャされたインストール状態とコンポーネントの現在の状態を詳しく比較できます。この要素を使用する場合、この要素は 1 回しか出現できません。

この要素には 1 つの必須子要素 <snapshot> があります。これは当該コンポーネントのインストール状態をキャプチャするために実行できる名前付きスナップショットブロックです。1 つ以上の <snapshot> 要素を使用できます。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの、アクセスが可能な <snapshotList> 要素コンテンツを継承します。派生コンポーネントが <snapshotList> を宣言する場合、そのコンテンツはベースコンポーネントのコンテンツと完全に結合されます。派生コンポーネントは新しい <snapshot> 要素を宣言することで継承された要素をオーバーライドできますが、ベースコンポーネントによって宣言された要素を削除することはできません。

<snapshot> 要素

<snapshot> 要素は <snapshotList> 要素の子で、当該コンポーネントのインストール状態をキャプチャするために実行される一連のステップを定義します。<createSnapshot> または <addSnapshot> ステップがこのスナップショットブロックを指定する場合、prepare ブロック内のステップが順に実行されます。続いて capture ブロックで指定されたファイルターゲットサーバーのキャプチャ領域でキャプチャされ、最後に cleanup ブロック内のステップが順に実行されます。

スナップショットブロックは、コンポーネントの現在の状態をそのインストール時の状態と比較するためにも使用されます。具体的には、ターゲットサーバーで prepare ステップが再実行され、続いてインストール時にキャプチャされたファイルが現在のファイル状態と比較され、cleanup ステップが再実行されます。

<snapshot> 要素には次の子要素があります。

- <paramList> – オプション要素で、当該スナップショットの prepare、capture、および cleanup の各ブロック内で使用するためのパラメータの一覧。この要素は 1 回しか出現できません。
- <varList> – オプション要素で、当該スナップショットの prepare、capture、および cleanup の各ブロック内で使用するためのローカル変数の一覧。この要素は 1 回しか出現できません。
- <prepare> – オプション要素で、ファイルのキャプチャまたは比較に備えて実行されるステップを含みます。この要素は 1 回しか出現できません。
- <capture> – オプション要素で、当該スナップショットの一部としてキャプチャされるファイルおよびディレクトリの一覧を含みます。この要素は 1 回しか出現できません。
- <cleanup> – オプション要素で、キャプチャまたは比較の完了後に実行されるステップを含みます。この要素は 1 回しか出現できません。

当該スナップショットが <createSnapshot> ステップから呼び出される場合は、必要なパラメータをその <paramList> 要素で宣言することはできません。

<varList>、 <prepare>、<capture>、および <cleanup> 要素は、集合的にスナップショットの本体を定義します。スナップショットブロックが抽象と宣言される場合、本体は含められません。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの、アクセス可能なすべてのスナップショットブロックを継承します。スナップショットブロックをオーバーライドするセマンティクスは、インストールブロックをオーバーライドするセマンティクスと同じです。

派生コンポーネントスナップショットブロックの prepare ブロックからベースコンポーネントスナップショットブロックの prepare ブロックを呼び出す方法はありません。同じことが cleanup についても言えます。このような呼び出しを行うには、ベースコンポーネントはその <prepare> および <cleanup> ステップを、派生コンポーネントによって呼び出すことができる制御ブロックに含める必要があります。

<snapshot> 要素の属性

この要素には次の属性があります。

- *access* – accessEnum 型のオプション属性で、スナップショットブロックのアクセシビリティを指定します。次の値が使用できます。
  - PUBLIC – アクセスは制限されません。これがデフォルトです。
  - PROTECTED – アクセスは同じパス内の派生コンポーネントとエンティティに制限されます。
  - PATH – アクセスは同じパス内のエンティティに制限されます。
  - PRIVATE – アクセスは当該コンポーネントに制限されます。
- *modifier* – modifierEnum 型のオプション属性で、スナップショットブロックのオーバーライド要件を指定します。次の値が使用できます。
  - ABSTRACT – ブロックに本体を含めることはできません。 本体は、非抽象派生コンポーネントによって指定する必要があるためです。スナップショットブロックを抽象と宣言できるのは、コンポーネントも抽象と宣言されている場合だけです。抽象ブロックは専用にはできません。非抽象ブロックの場合、本体を宣言する必要があります。
  - FINAL – スナップショットブロックを派生コンポーネントによってオーバーライドすることはできません。

  この属性を指定しない場合、派生コンポーネントはブロックをオーバーライドするかどうかを選択できます。
- *name* – entityName 型の必須属性で、スナップショットブロックの名前です。この名前は、包含している <snapshotList> 内のすべてのスナップショットブロックの中で一意である必要があります。
- *description* – オプション属性で、スナップショットブロックを説明する文字列。この属性は文書化に便利です。

<prepare> 要素

<prepare> 要素は <snapshot> 要素の子で、ファイルのキャプチャまたは比較に備えた一連のステップを定義します。これらのステップは、当該スナップショットを対象とする <createSnapshot> または <addSnapshot> ステップの結果として、かつ当該スナップショットを対象とする比較作業として実行されます。どのような場合も、これらのステップはファイルのキャプチャまたは比較作業に先立って実行されます。

<prepare> 要素の子は、1 つ以上の <call>、<execNative>、および <transform> ステップから構成されます。ほかのステップは許可されません。包含されるステップは、スナップショットブロックで宣言されたローカルパラメータおよびローカル変数と、隠蔽されていない component 置換変数を参照できます。

<capture> 要素

<capture> 要素は、当該スナップショットの一部としてキャプチャされるファイルとリソースを定義します。キャプチャは、<prepare> ブロック内のステップが実行されたあとでしか行われません。キャプチャが完了すると、<cleanup> ブロック内のステップが実行されます。

<capture> 要素の子は、1 つ以上の <addFile>、<addSnapshot>、および <addResource> 要素から構成されます。このブロックに挙げられるファイルとディレクトリは、指定された順にキャプチャされます。包含される子は、スナップショットブロックで宣言されたローカルパラメータおよびローカル変数と、隠蔽されていない component 置換変数を参照できます。

<addFile> 要素

<addFile> 要素は <capture> 要素の子であり、包含するスナップショットの一部としてキャプチャされるファイルを指定します。

<addFile> 要素の属性

<addFile> 要素には次の属性が含まれます。

- *path* – 必須属性で、ターゲットホストのファイルシステム上のファイルまたはディレクトリのパス名を指定します。この属性は、単純置換変数を参照できます。
- *ownership* – オプション属性で、キャプチャされたファイルの所有権オプションを指定します。

  スナップショットによってキャプチャされるインストール状態の情報の 1 つに、ファイルおよびディレクトリの所有権があります。この所有権は UNIX のアクセス許可と同じではありません。所有権は、参照カウントの概念により近いと言えます。具体的には、ファイルまたはディレクトリは、1 つ以上のスナップショットによって所有されているものとしてキャプチャできます。

  ほかのコンポーネントのインストールにより、ファイルの所有者が変更された場合、そのファイルがその初期状態と比較されたときに、この変更が認識され報告されます。この機能により、あるコンポーネントが、別のコンポーネントに関連付けられているファイルを誤って上書きしたために生じた差異を追跡することができます。スナップショットの所有権情報は、所有者テーブルと呼ばれる、ターゲットホスト上のリポジトリ内にキャプチャされます。

  ownership 属性の値は以下のセマンティクスを持ちます。

  - SET_SELF – ownership 値がこの値に設定された場合、所有者テーブルは関連付けられたファイルまたはディレクトリ用の単一のエントリを含むように更新されます。そのエントリは、実行するインストール済みコンポーネントとスナップショットを所有者としてリストします。また、キャプチャされたファイルコンテンツまたはディレクトリコンテンツのキャプチャ領域 ID もリストします。

- ADD_SELF – ownership がこの値に設定された場合、インストール済みコンポーネントとスナップショットを付加的な所有者として追加し、以前の所有者として既存のキャプチャ領域 ID を共有させる必要があります。
- ADD_TEMP – この値は ADD_SELF と似ていますが、新しいキャプチャが常に作成されてその ID が常に新しいエントリに使用されます (ほかの所有者の ID を共有することはありません)。

この属性を指定しない場合、デフォルト値は SET_SELF です。

- *filter* – boolean 型のオプション属性で、ファイルまたはディレクトリ、あるいはそれらの両方をキャプチャすべきかを示します。

  path の値がディレクトリである場合、この属性は、ディレクトリ自体、そのディレクトリに含まれるファイル、またはそれらの両方のどれをキャプチャすべきかを示すために使用されます。*path* の値がディレクトリでない場合、この属性は無視され、ファイルが直接キャプチャされます。この属性を指定しない場合、デフォルトの BOTH が使用されます。

  この属性の値には次のものがあります。

  - DIRECTORIES
  - FILES
  - BOTH
- *recursive* – オプション属性で、現在のフィルタ設定を使用してサブディレクトリを再帰的にキャプチャすべきかどうかを示します。

  path の値がディレクトリである場合、この属性は、現在のフィルタ設定を使用してサブディレクトリを再帰的にキャプチャすべきかどうかを示します。*path* の値がディレクトリでない場合、この属性は無視され、ファイルが直接キャプチャされます。デフォルト値は true です。
- *displayName* – オプション属性で、当該スナップショットエントリがほかのエントリと比較される際に表示に含めるテキスト。

  この属性は、単純置換変数を参照できます。

<addSnapshot> 要素

<addSnapshot> 要素は、外部スナップショットブロックを実行する必要があること、およびそのコンテンツを当該スナップショットに追加すべきであることを示します。

<addSnapshot> ステップを使用することは、次のシナリオのすべてと意味的に同じです。

- 呼び出されたスナップショットの <prepare> ステップを、呼び出し側スナップショットの prepare ブロックの最後に追加する
- 呼び出されたスナップショットの <cleanup> ステップを、呼び出し側スナップショットの cleanup ブロックの先頭に追加する
- 呼び出されたスナップショットの <capture> ステップを、(<addSnapshot> ステップの代わりに) 呼び出し側スナップショットの capture ブロックに追加する

呼び出される側および呼び出し側のスナップショットブロックによって参照されるファイルが重複しないように注意する必要があります。

<capture> 要素には任意の数の <addSnapshot> ステップを指定できます。呼び出される側のスナップショットは、その <capture> 要素に任意の数の <addSnapshot> ステップを含むことができます。

<addSnapshot> コールアウトを使用してファイルをスナップショットに間接的に追加する場合、スナップショットのキャプチャを開始した最上位のコンポーネントは (<addFile> 命令が入ったコンポーネントとは対照的に) ファイルの所有者と見なされます。同様に、スナップショットに対して比較が実行される場合は、スナップショットを開始した最上位コンポーネントの <diff> 要素の <ignore> 命令だけが考慮されます。<addSnapshot> コールアウトの結果として出現したコンポーネントに含まれる <diff> 要素の <ignore> 命令は考慮されません。

<addSnapshot> 要素には、*entityName* 型の 1 つの必須要素 *blockName* があります。これは、実行する外部スナップショットブロックの名前です。

<addSnapshot> ステップには次の子要素があります。

- <argList> – オプション要素で、スナップショットブロックに渡す引数の一覧。この要素は 1 回しか出現できません。
- **Installed component targeter –** オプション要素で、スナップショットブロックを含むコンポーネントを特定します。この要素を指定しない場合は、<thisComponent> が使用されます。

<addResource> 要素

<addResource> 要素は、包含するスナップショットの一部としてキャプチャされるコンポーネントに関連付けられたリソースを指定します。この要素は、単純コンポーネントにしか含めることができません。

この要素は、次に示すように、同等の <addFile> 要素の簡略構文として機能します。

- 関連付けられたリソースが *deployMode* =ADD_TO のディレクトリリソースである場合、<addResource> は次の文と同義です。

  ```
  <addFile path="path-of-deployed-directory" filter="FILES"
  displayName="resourceSourcePath"/>
  ```

- 関連付けられたリソースが *deployMode* =REPLACE のディレクトリリソースである場合、<addResource> は次の文と同義です。

  ```
  <addFile path="path-of-deployed-directory"
  displayName="resourceSourcePath"/>
  ```

- これらのどちらでもない場合、関連付けられたリソースはファイルベースのリソースであり、<addResource> は次の文と同義です。

  ```
  <addFile path="path-of-deployed-file"
  displayName="resourceSourcePath"/>
  ```

<cleanup> 要素

<cleanup> 要素は <snapshot> 要素の子であり、ファイルのキャプチャまたは比較が完了したあとで実行される一連のステップを定義します。これらのステップは、当該スナップショットを対象とする <createSnapshot> または <addSnapshot> ステップの結果として、かつ当該スナップショットを対象とする比較作業として実行されます。どのような場合も、これらのステップはすべてのファイルのキャプチャまたは比較作業のあとに実行されます。cleanup ブロックは、prepare ブロックで作成された一時ファイルを削除するために使用します。

<cleanup> 要素の子は、1 つ以上の <call>、<execNative>、および <transform> ステップから構成されます。ほかのステップは許可されません。包含されるステップは、スナップショットブロックで宣言されたローカルパラメータおよびローカル変数と、隠蔽されていないコンポーネント置換変数を参照できます。

<controlList> 要素

<controlList> 要素は <component> 要素のオプションの子であり、当該コンポーネントで利用できる制御ブロックをリストします。この要素を使用する場合、この要素は 1 回しか出現できません。

この要素には制御ブロックである必須子要素 <control>があります。複数の <control> 要素を指定できます。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの、アクセスが可能な <controlList> 要素コンテンツを継承します。派生コンポーネントが <controlList> を宣言する場合、そのコンテンツはベースコンポーネントのコンテンツと完全に結合されます。派生コンポーネントは新しい <control> 要素を宣言することで継承された要素をオーバーライドできますが、ベースコンポーネントによって宣言された要素を削除することはできません。

<control> 要素

<control> 要素は、当該コンポーネントで利用できる制御ブロックを定義します。制御ブロックは、当該コンポーネントのインストールが完了したあとで実行できる一連のステップです。たとえば、データベースアプリケーションのコンポーネントに制御ブロックを含め、データベースの「起動」または「停止」を行えます。 <call> ステップは、名前で制御ブロックを呼び出すことができます。この呼び出しにより、制御ブロックステップが順に実行されます。

<control> 要素の子は、オプションの <paramList> 要素とそれに続く本体から構成されます。この本体は、オプションのローカル <varList> 要素と、それに続くゼロ個以上の「共通」ステップから構成されます。詳細は、25 ページの「共有ステップ」を参照してください。制御ブロックが抽象と宣言されている場合、本体は含められません。

デフォルトでは、派生コンポーネントはそのベースコンポーネントの、アクセス可能なすべての制御ブロックを継承します。制御ブロックをオーバーライドするセマンティクスは、インストールブロックをオーバーライドするセマンティクスと同じです。

<control> 要素の属性

<control> 要素には次の属性があります。

- *access* – accessEnum 型のオプション属性で、制御ブロックのアクセシビリティを指定します。次の値が有効です。
  - PUBLIC – アクセスは制限されません。これがデフォルト値です。
  - PROTECTED – アクセスは同じパス内の派生コンポーネントとエンティティに制限されます。
  - PATH – アクセスは同じパス内のエンティティに制限されます。
  - PRIVATE – アクセスは当該コンポーネントに制限されます。

  コンポーネントから直接実行できるのは PUBLIC ブロックだけです。
- *modifier* – modifierEnum 型のオプション属性で、制御ブロックのオーバーライド要件を指定します。次の値が有効です。
  - ABSTRACT – ブロックに本体を含めることはできません。その代わりに本体は、非抽象派生コンポーネントによって指定する必要があります。制御ブロックを抽象と宣言できるのは、コンポーネントも抽象と宣言されている場合だけです。抽象ブロックは専用にはできません。非抽象ブロックの場合、本体を宣言する必要があります。
  - FINAL – 制御ブロックを派生コンポーネントによってオーバーライドすることはできません。

  この属性を指定しない場合、派生コンポーネントはブロックをオーバーライドするかどうかを選択できます。
- *name* – entityName 型の必須属性で、制御ブロックの名前。これは、制御を実行するために <call> ステップから参照されます。
- *description* – オプション属性で、制御ブロックを説明する文字列。

<diff> 要素

<diff> 要素は <component> 要素のオプションの子です。この要素は、当該コンポーネントで比較を実行する際に比較エンジンによって使用される命令のリストを含みます。この要素を使用する場合、この要素は 1 回しか出現できません。

<diff> 要素には 1 つの必須子要素である <ignore> があります。この要素は、比較の際に無視するディレクトリパスを指定します。<ignore> 要素は、1 回または複数回使用することができます。

派生コンポーネントは、ベースコンポーネントによって宣言されたすべての無視命令を自動的に継承します。また、それ自体の <diff> 要素内で無視命令をさらに宣言することもできます。継承された命令を削除することはできません。

<ignore> 要素

<ignore> 要素は <diff> 要素の子であり、比較で当該コンポーネントを使用する場合に無視するファイル名パスのパターンを指定します。この要素は、一般に、インストールされたアプリケーションによって作成されるファイルとディレクトリ (ログファイルなど) に使用されます。この要素の構成可能属性は、コンポーネント置換変数を参照できます。

<ignore> 要素には 1 つの必須属性 *path* があります。これは、無視するファイル名パスに一致する glob スタイルのパターンです(例: /logs/*.log) 。この属性は、コンポーネントスコープの置換変数を参照できます。

コンポーネントのインストール専用のステップ

この節では、コンポーネントのインストールブロック内でしか使用できないステップを示します。

- 92 ページの「<createDependency> ステップ」
- 94 ページの「<createSnapshot> ステップ」
- 95 ページの「<install> ステップ」
- 96 ページの「<deployResource> ステップ」

<deployResource> ステップを除くすべてのステップは、単純コンポーネントと複合コンポーネントの両方で使用できます。<deployResource> ステップは、単純コンポーネントのみが使用できます。

<createDependency> ステップ

このステップは、ほかのコンポーネントに対する現在のコンポーネントの永続的な依存性を作成します。このステップは、その実行時にまず指定された基準に一致するインストール済みコンポーネントが存在しないかチェックします。そのようなコンポーネントが存在しない場合、このステップは停止します (この動作は <checkDependency> ステップに同じ)。一致するコンポーネントが見つかると、一致したコンポーネント (依存先コンポーネント) と呼び出しコンポーネント (依存元コンポーネント) の間に永続的な依存性が作成されます。

複数のインストール済みコンポーネントが基準に一致する場合 (インストールパスを指定しないとこのような状況となり得る) は、最新のものが依存先として使用されます。永続的なコンポーネントは、ステップで指定される名前で作成されます。 この名前は、インストールブロックで作成されるすべての依存性間で一意のものでなければなりません。

作成された永続的な依存性は、依存コンポーネントがアンインストールされるまで持続します。永続的な依存性を作成したあとで後続のステップで依存コンポーネントのインストールが停止する場合、依存性はこの停止時にただちに削除されます。

個々のコンポーネントは、<createDependency> ステップをほかのコンポーネントごとに実行することにより任意の数のコンポーネントに依存できます。依存性が満たされない場合、実際の処理が行われる前にインストールが停止するように、<createDependency> ステップはインストールブロックの最初のステップに指定する必要があります。

<createDependency> ステップには 1 つの必須子要素があり、この要素が依存先コンポーネントを特定します。その子要素はインストール済みコンポーネントターゲッターです。詳細は、46 ページの「インストール済みコンポーネントターゲッター」を参照してください。

<createDependency> ステップの属性

<createDependency> ステップには identifier 型の 1 つの必須属性 *name* があり、これは作成する依存性の名前です。この名前は、現在のコンポーネントによって作成されるすべての依存性の間で一意のものでなければなりません。

<createDependency> ステップのアンインストールについて

コンポーネントのアンインストールは、そのコンポーネントに対して依存するインストール済みコンポーネントが存在する場合は行えません。コンポーネントのアンインストールブロックが検出された場合で、そのコンポーネントを依存先とする 1 つ以上の永続的な依存性が存在するときは、アンインストールはただちに停止します。

しかし、コンポーネント A によってコンポーネント B のアンインストールが進行している場合、A によって作成された B に対する依存性が B のアンインストールを防止することはなく、この依存性は B が正常にアンインストールされた時点で暗黙に削除されます。

依存先コンポーネントは、<dependantCleanup> ブロックを使用してその依存元 (dependant) をアンインストールするアクションを指定できます。

<createDependency> ステップの再インストールについて

同じホストとインストールパス上にインストールされた同じバージョンツリー内に既存コンポーネントが存在する場合、コンポーネントのインストールは、再インストールと見なされます。コンポーネントの再インストールが行えるのは、その新しいコンポーネントがオリジナルコンポーネントのすべての依存元に適合する場合だけです。コンポーネントは常に同じバージョンのコンポーネントで再インストールできます。ただし、永続的な依存性を作成した <createDependency> ステップ内で指定した制約に、新しいバージョンが適合する場合には、異なるバージョンで再インストールすることもできます。

コンポーネントの単純インストールブロックが検出された場合は、インストールによって既存のインストールが上書きされるかどうかを、プロビジョニングソフトウェアが判別します。上書きされる場合、プロビジョニングソフトウェアは、既存のコンポーネントを依存先としている永続的な依存性をすべて検出し、インストールされる新しいコンポーネントについても依存性の制約が適合するかを検証し直します。適合しないものがある場合、新しいコンポーネントのインストールは失敗し、オリジナルコンポーネントがインストールされたままとなります。

すべて適合する場合、新しいコンポーネントがそのインストールを正常に完了した時点で、当該コンポーネントがすべての永続的な依存性の新しい依存先となります。オリジナルのコンポーネントはアンインストールされたと見なされ、そのコンポーネントの依存元であったすべての永続的な依存性が削除されます。つまり、新しいコンポーネントは必要に応じて依存性を作成し直す必要があります。

<createDependency> ステップの命名規則

依存性の名前は、「*xxx*2 *yyy*」という形式に従います。*xxx* は依存元コンポーネントの名前、 *yyy* は依存先コンポーネントの名前を示します。

たとえば、管理対象サーバー WebLogic はその管理サーバーに対して server2domain という依存性を持ち、その包含クラスタに対して server2cluster という依存性を持つ例が考えられます。この慣例は依存関係の状態を自己記録として表現するのに便利であり、特定のコンポーネントの依存先関係と依存元関係の両方を示すことができます。

<createSnapshot> ステップ

このステップは、インストール中のコンポーネントの現在のインストール状態のスナップショットを作成します。インストールブロックには、任意の数の <createSnapshot> ステップを指定できます。

<createSnapshot> ステップは引数の引き渡しをサポートしないため、必要なパラメータを指定されたスナップショットブロックで宣言することはできません。引数の引き渡しがサポートされないのは、この機能をサポートすると生成されたスナップショットに対してあとで比較が行われる際に引数の収集と引き渡しのサポートも必要となるためです。

複合コンポーネントに対して行われる比較には、そのコンポーネントにより直接作成されるすべてのスナップショットが含まれます。またこのような比較には、入れ子になったすべてのコンポーネント参照の再帰的なインストールによって作成されたスナップショットのツリー一式も含まれます。しかし、最上位のコンポーネント参照に関連付けられたスナップショットは考慮されません。そのため、<addSnapshot> キャプチャ命令を使用して、複合コンポーネントのスナップショット内にそのようなスナップショットを明示的に含める必要があります。

また、入れ子になったコンポーネントは相互に依存していることがあり、このような場合にはそれらのコンポーネントがすべてインストールされるまでスナップショットのキャプチャを延期する必要があります。相互依存の関係にある例として、ディレクトリを配備する入れ子コンポーネントと、そのディレクトリにファイルを配備する入れ子コンポーネントが挙げられます。このような相互依存の場合、包含コンポーネントは、インストール中にスナップショットを作成しないように入れ子コンポーネントに指示するパラメータの引渡し、または特殊なインストールブロックを使用して、入れ子コンポーネントをインストールする必要があります。その後、<addSnapshot> 命令を使用して、包含コンポーネントのスナップショットブロックに、入れ子コンポーネントの適切なスナップショットを含めるようにします。

<createSnapshot> ステップの属性

<createSnapshot> ステップには entityName 型の 1 つの必須属性 *blockName* があり、これは当該コンポーネント内で実行するスナップショットブロックの名前です。

<install> ステップ

このステップは、ターゲットホストにコンポーネントをインストールします。このステップを使用すると、対象コンポーネントの指定されたインストールブロックのステップが実行されます。

このステップの構文は、コンポーネントターゲッターを省略できる (ターゲッターを省略すると <thisComponent> が指定されたと想定される) 点を除き、単純プラン <install> ステップ (105 ページの「<install> ステップ」を参照) で指定する場合と同じです。

コンポーネント内で使用される場合、このステップは同一コンポーネント内のほかのインストールブロックの呼び出しに使用されます。この場合、一番外側のインストールブロックがその実行を完了するまで、コンポーネントのインストールが完了したとは見なされません。また、ほかのローカルインストールブロックの呼び出し対象がほかのホストに変更された場合でも、コンポーネントは最初のインストールステップの対象となったホストにインストールされたとしか見なされません。

このステップは、参照先コンポーネントをインストールする目的で複合コンポーネント内でも使用できます。参照先コンポーネントは、包含コンポーネントのインストール先以外のホストにもインストールできます。包含コンポーネントのインストールが失敗する場合、この失敗以前にインストールが正常に行われている入れ子になった参

照先コンポーネントはすべて、アンインストールブロックを実行することなく暗黙にアンインストールされます。しかし、失敗以前にインストールが正常に行われた最上位の参照先コンポーネントはインストールされたままとなります。

<deployResource> ステップ

このステップは、包含コンポーネントのリソースを配備します。このステップは、単純コンポーネントのインストールブロックでのみ使用可能です。属性または子要素はありません。

deployMode=ADD_TO であるディレクトリ型リソースは、それが包含している各ファイルをコピーし、ディレクトリ構造を維持します。また、必要に応じて新しいディレクトリが作成されます。既存のディレクトリの構造とコンテンツは (リソースコンテンツのコピーを除き) 変化しません。 個々のファイルアクセス許可と所有権は、適宜更新されます。

deployMode=REPLACE であるディレクトリ型リソースは、配備前に既存ディレクトリが再帰的に削除されることを除き、deployMode=ADD_TO と同様に扱われます。

ほかのリソースはすべてコピーされ、続いてアクセス許可と所有権が適宜更新されます。構成可能としてチェックインされたリソースには、コピーされる前に変数置換が行われます。 構成可能リソースは、リソースが宣言されているコンポーネントにアクセス可能な任意の変数を参照できます。

コンポーネントのアンインストール専用のステップ

この節では、コンポーネントのアンインストールブロック内でだけ使用できる、<uninstall> ステップと <undeployResource> ステップを説明します。

<uninstall> ステップ

このステップは、ターゲットホストからコンポーネントをアンインストールするために使用され、任意のコンポーネントのアンインストールブロックで使用できます。このステップを使用することで、対象コンポーネントの指定されたアンインストールブロックのステップが実行されます。

このステップの構文は、コンポーネントターゲッターを省略できる (ターゲッターを省略すると <thisComponent> が指定されたと想定される) 点を除き、単純プラン <uninstall> ステップで指定する場合と同じです。

コンポーネント内で使用される場合、このステップは同一コンポーネント内のほかの既存アンインストールブロックの呼び出しに使用されます。この場合、一番外側のアンインストールブロックがその実行を完了するまで、コンポーネントはアンインストールされません。また、ほかのローカルアンインストールブロックの呼び出し対象がほかのホストに変更された場合でも、コンポーネントは当初アンインストールステップの対象となったホストでのみアンインストールされます。

このステップは、参照先コンポーネントをアンインストールする目的で、複合コンポーネント内でも使用できます。 複合コンポーネントがアンインストールされる場合、入れ子になったその参照先コンポーネントのうち明示的にアンインストールされなかったものはすべて、アンインストールブロックを実行することなくシステムによって暗黙にアンインストールされます。しかし、明示的にアンインストールされなかった最上位の参照先コンポーネントはインストールされたままとなります。

<undeployResource> ステップ

このステップは、包含コンポーネントからリソースを削除するために使用されます。このステップは単純コンポーネントのアンインストールブロックでのみ使用可能で、また属性も子要素もありません。

deployMode=ADD_TO であるディレクトリ型リソースの場合、そのファイルは削除されますがサブディレクトリは残ります。

deployMode=REPLACE であるディレクトリ型リソースの場合、ディレクトリ全体とそのコンテンツが削除されます。

ほかのリソースはどれも単純ファイルとして扱われ、削除されます。

リソースは、当該コンポーネントのインストール時に初めから配備されたかどうかにかかわらず削除されます。つまり、<deployResource> ステップを含まないインストールブロックを使用して当該コンポーネントがインストールされた場合でも、<undeployResource> ステップによって、リソースはコンポーネントによって初めからインストールされていたかのように削除されます。

第 4 章

プランのスキーマ

この章では、N1 Grid Service Provisioning System プランの XML スキーマについて説明します。次のトピックを説明します。

特に断りのない限り、この章で説明する属性は、コンポーネントスコープの置換変数を参照できません。

XML スキーマアーキテクチャーの概要は、第 1 章を参照してください。

<executionPlan> 要素

プラン全体は、<executionPlan> 要素によって包含されます。

プランは、単純プランまたは複合プランのどちらかです。単純プランは、ターゲットサーバーの特定セットに対して実行されるステップの連続リストです。単純プランはほかのプランを含まず、ほかのプランを呼び出すこともありません。複合プランは、ほかのサブプランだけから構成されています。各単純サブプランは、異なる対象セットで実行可能であるため、複合プランは直接の対象とはなりません。

<executionPlan> 要素には次の子要素があり、以下に示す順序で出現する必要があります。これらの子要素は、独自の子要素や属性 (両方も可能) を持つことができます。

<executionPlan> 要素属性

<executionPlan> 要素には次の属性があります。

- *xmlns* – 必須文字列で、次の値を持っています。

  http://www.sun.com/schema/SPS

- *xmlns:xsi* – 必須文字列で、次の値を持っています。

  http://www.w3.org/ 2001/XMLSchema-instance

- *xsi:schemaLocation* – オプション文字列で、推奨値は次の値です。

  http://www.sun.com/schema/SPS plan.xsd

- *name* – entityName 型の必須属性で、実行プランの名前。
- *path* – pathName 型のオプション属性で、実行プランの絶対パス。この属性を指定しない場合、ルートパス (/) が使用されます。値は、プランが保存された時点で存在するフォルダを指定する必要があります。
- *description* – オプション属性で、実行プランを説明する文字列。
- *version* – schemaVersion 型の必須属性で、使用されているプランスキーマのバージョン。現在許可されている値は 5.0 のみです。

<paramList> 要素

<paramList> 要素は、<executionPlan> 要素のオプションの子です。この要素は、プラン内に含まれるステップおよび参照されるコンポーネントが使用するパラメータのリストを宣言するために使用されます。この要素を指定する場合、この要素は 1 回しか出現できません。

このプランが最上位のプランとして実行される場合、呼び出し側はこのリストで宣言された全パラメータの値を入力するように求められます。このプランがほかのプラン内で <execSubplan> ステップの結果として呼び出される場合、呼び出し側のプランは、デフォルト値を持たない <paramList> で宣言されたすべてのパラメータの値を明示的に渡す必要があります。

<paramList> 要素には 1 つの必須子要素 <param> があり、これはプランパラメータの宣言です。この宣言には、名前、プロンプト、デフォルト値が含まれます。宣言する必要があるパラメータごとに、1 つの <param> 要素を指定できます。

<param> 要素

<param> 要素はプラン <paramList> 要素の子であり、プラン内で使用するパラメータの宣言に使用されます。

<param> 要素の属性

<param> 要素には次の属性があります。

- *name* – identifier 型の必須属性で、プランパラメータの名前です。この名前は、最上位のすべてのプランパラメータおよびプラン変数にわたって一意でなければなりません。
- *prompt* – 文字列のオプション属性で、パラメータの値を入力するよう求める際のユーザーインタフェースで表示されます。この属性を指定しないと、*name* の値が使用されます。
- *default* – 文字列のオプション属性で、パラメータのデフォルト値。このデフォルト値は、セッション変数への参照を含むことはできません。
- *displayMode* – オプション属性で、パラメータの表示モードを指定します。次の有効な値が使用できます。
  - PASSWORD – ユーザー指定の値が隠されます。つまりパスワードは表示されないか、アスタリスクに置き換えられます。
  - BOOLEAN – チェックボックスを使用してパラメータが指定されます。
  - CLEAR – 入力時に値が表示されます。

  値が CLEAR または BOOLEAN である場合、入力時に安全に表示されます。この属性を指定しないと、値は CLEAR になります。

<varList> 要素

<varList> 要素は <executionPlan> 要素と <inlineSubplan> ステップのオプションの子です。<inlineSubplan> ステップの詳細については、104 ページの「<inlineSubplan> ステップ」を参照してください。 <varList> 要素は、プラン内に含まれるステップおよび参照されるコンポーネントが使用する変数のリストを宣言するために使用されます。これらの変数の値は宣言時に定義され、再定義は行えません。この要素を指定する場合、この要素は 1 回しか出現できません。

<varList> 要素には 1 つの必須子要素 <var> があり、これはプラン変数の宣言です。宣言には名前と値が含まれます。宣言する必要がある変数ごとに、1 つの <var> 要素を指定します。

<var> 要素

<var> 要素は plan <varList> 要素の子であり、プラン変数 (名前や値など) を宣言するために使用されます。

<var> 要素の属性

この要素には次の属性があります。

- *name* – identifier 型の必須属性で、ローカル変数の名前。この名前は、包含する <varList> 内のすべての変数にわたって一意でなければなりません。最上位の <executionPlan> に関連付けられた変数は、プランパラメータにおいても一意でなければなりません。

- *default* – 文字列である必須属性で、プラン変数のデフォルト値。この値には、以前に宣言されているプラン変数、セッション変数、およびプランパラメータへの参照を含めることができます。このプランが単純プランの場合には、ターゲットホスト属性およびインストール済みコンポーネント変数への参照を含めることができます。

<simpleSteps> 要素

<simpleSteps> 要素は <executionPlan> 要素および <inlineSubplan> ステップのオプションの子です。<inlineSubplan> ステップの詳細については、104 ページの「<inlineSubplan> ステップ」を参照してください。<simpleSteps> 要素には、1 つ以上の「共通」ステップまたは「単純プラン専用」ステップが含まれます。<simpleSteps> 要素の存在は、そのプランが (複合プランではなく) 単純プランであることを示します。この要素を指定する場合、この要素は 1 回しか出現できません。

実行時にこの要素内のステップは、呼び出し側によって選択された一連の論理ターゲットホストで順次実行されます。このようなホストは初期ターゲットホストと呼ばれます。このプランの実行時に、そのステップは、初期ホスト以外のホストで実行されるようリダイレクトすることができます。プランが実際に実行されるホストは現在のターゲットホストと呼ばれます。ステップがリダイレクトされない場合、現在のホストと初期ホストは同じになります。初期ホストは、現在のホストと同じように、仮想ホストと物理ホストのどちらにもなれます。物理ホストは現在のホストのルート親ホストで、現在のホストが物理ホストである場合、現在のホストと同じです。

<simpleSteps> 要素の子は、1 つ以上の「共通」ステップまたは「単純プラン専用」ステップから構成されています。これらのステップには、プランのパラメータおよび変数への参照を含めることができます。詳細は、105 ページの「単純プラン専用のステップ」を参照してください。

<simpleSteps> 要素の属性

<simpleSteps> 要素には次の属性があります。

- *executionMode* – オプション属性で、包含した複数のステップをターゲットホストに対して順次実行するか同時に実行するかを示します。次の値が有効です。
  - PARALLEL
  - SERIES

  この属性を指定しないと、値は PARALLEL になります。
- *limitToHostSet* – オプション属性で、当該プランの有効な対象と見なされるホストを含むホストセットの名前を指定します。

  この属性を指定しないと、すべてのホストが有効な対象と見なされます。すべてのホストを有効な対象としない場合、ユーザーが指定する対象は、指定されたホストセットに含まれるホストのサブセットでなければなりません。指定されたホストセットに含まれないホストが対象内に存在する場合、プランで実行時エラーが発生

します。サポートされている既存のホストセットに対応しない名前を指定した場合も、プランで実行時エラーが発生します。指定されたホストセットがプラグインにより定義されたホストである場合、*pluginName* は、ホストセット名に対する接頭辞 (*pluginName*# *hostSetName* など) である必要があります。このようなプラン実行時エラーは、プリフライトが始まる前に検証の段階で報告されます。

<compositeSteps> 要素

<compositeSteps> 要素は <executionPlan> 要素および <inlineSubplan> ステップのオプションの子です。<inlineSubplan> ステップの詳細については、104 ページの「<inlineSubplan> ステップ」を参照してください。
<compositeSteps> 要素には、1 つ以上の「複合プラン専用」ステップが含まれます。<compositeSteps> 要素の存在は、そのプランが複合プランであることを示します。<compositeSteps> 要素に属性は含まれません。この要素を指定する場合、この要素は 1 回しか出現できません。

<compositeSteps> 要素の子は、1 つ以上の「複合プラン専用」ステップから構成されます。これらのステップには、プランのパラメータおよび変数への参照を含めることができます。詳細は、103 ページの「複合プラン専用のステップ」を参照してください。

複合プラン専用のステップ

この節では、複合プラン内でだけ使用できるステップを説明します。複合プラン内に含まれる一部のステップの属性は、プラン変数およびプランパラメータの参照を含むことができます。

<execSubplan> ステップ

<execSubplan> ステップは、ほかのプランを実行します。<execSubplan> ステップは、<compositeSteps> 要素の子としてしか指定できません。

<execSubplan> ステップには、1 つのオプション子要素 <argList> があり、これは呼び出されたプランに渡す引数の一覧です。呼び出されたプランの <paramList> セクション内の、デフォルト値が宣言されていないパラメータごとに、この <argList> によって宣言された対応する引数が存在しなければなりません。詳細は、26 ページの「<argList> 要素」を参照してください。この要素を指定する場合、この要素は 1 回しか出現できません。

<execSubplan> ステップの属性

<execSubplan> ステップには次の属性があります。

- *planName* – entityName 型の必須属性で、実行するプランの名前。このステップを実行する際には、この名前を持つ、対応する最上位の <executionPlan> を指定する必要があります。この名前でインラインサブプランを参照することはできません。
- *planPath* – pathReference 型のオプション属性で、実行するプランのパス。この属性を指定しないと、包含するプランのパスが使用されます。
- *planVersion* – Version 型のオプション属性で、実行するプランのバージョンです。この属性を指定しないと、指定されたプランの最新バージョンが使用されます。

<inlineSubplan> ステップ

<inlineSubplan> ステップは、連続した複数のステップを実行します。このステップは、<compositeSteps> 要素の子としてしか指定できません。

<inlineSubplan> ステップは <execSubplan> ステップに類似していますが、<execSubplan> ステップは実行する外部プランの名前を指定するのに対し、<inlineSubplan> ステップは子要素として実行するプランを直接包含します。

インラインサブプランと最上位プランの大きな違いは、インラインサブプランは別個の名前付きエンティティとしては保存されない点です。そのためインラインサブフランは、<execSubplan> ステップでは外部から参照することができません。最上位プランは別個の名前付きエンティティであり、<execSubplan> ステップからの参照が可能です。

インラインサブプランは、コンテンツが簡潔で、呼び出し側プランのコンテキストとロジックに直接結合されている場合に便利です。このような場合以外は、独立したプランとして使用しても無意味です。呼び出し側プランのコンテキストとロジックに直接結合されている場合は、自己包含した 1 つのユニットに全ステップを含めるとプランのメンテナンスが容易になるほか、プランが読みやすくなります。

最上位プランと異なり、インラインサブプランはパラメータを宣言できません。インラインサブプランは、包含するすべてのプランのパラメータと変数を暗黙に継承します。インラインサブプランがそれ自身にローカルな変数を別途宣言することは可能であり、これにより包含するプランの変数とパラメータを隠蔽できます。サブプラン変数と包含するプランの変数が同じ名前を持つと、サブプラン変数が包含するプランの変数を隠蔽します。このようなケースでは、そのステップで使用できるのは一番内側のサブプランによって宣言された変数の値だけです。

<inlineSubplan> ステップはオプションの <varList> から構成され、インラインサブプランが単純プランであるか複合プランであるかに基づき、1 つの追加子要素 <simpleSteps> または <compositeSteps> がそのあとに続きます。

<inlineSubplan> ステップには次の子要素があります。

- <varList> – オプション要素で、インラインサブプラン内で使用するプラン変数の一覧。この要素を指定する場合、この要素は 1 回しか出現できません。

- <simpleSteps> – オプション要素で、単純ステップの一覧を含みます。この <simpleSteps> 要素または <compositeSteps> 要素の一方のみが存在できます。この要素を指定する場合、この要素は 1 回しか出現できません。
- <compositeSteps> – オプション要素で、複合ステップの一覧を含みます。この <compositeSteps> または <simpleSteps> 要素の一方のみが存在できます。この要素を指定する場合、この要素は 1 回しか出現できません。

<inlineSubplan> ステップの属性

<inlineSubplan> ステップには次の属性があります。

- *planName* – entityName 型の必須属性で、インラインサブプランの特定に使用される名前。この名前は主に表示目的で使用され、ほかのプラン (インラインまたは最上位) の名前と区別する必要はありません。
- *description* – オプション属性で、インラインサブプランを説明する文字列。この属性は文書化に便利です。

単純プラン専用のステップ

この節では、単純プラン内でだけ使用できるステップを説明します。プラン内に含まれるステップは、そのプランによって宣言された変数を参照できます。またこれらのステップは、すべての包含プランの隠蔽されていないあらゆる変数とパラメータも参照できます。

<install> ステップ

<install> ステップは、コンポーネントをターゲットホストにインストールします。これにより、関連付けられたコンポーネントの指定された <installSteps> 要素のステップが実行されます。 このステップは、<simpleSteps> 要素の子としてしか指定できません。

<install> ステップには次の子要素があります。

- <argList> – オプション要素で、<installSteps> ブロックに渡す引数の一覧。この要素を指定する場合、この要素は 1 回しか出現できません。詳細は、26 ページの「<argList> 要素」を参照してください。
- リポジトリコンポーネントターゲッター – 必須要素で、インストールするコンポーネントを特定します。詳細は、54 ページの「リポジトリコンポーネントターゲッター」を参照してください。

<install> ステップの属性

<install> ステップには entityName 型の 1 つの必須属性*blockName* があり、これは対象コンポーネント内で実行されるインストールブロックの名前です。

<uninstall> ステップ

<uninstall> ステップは、ターゲットホスト上に現在インストールされているコンポーネントのリソースをアンインストールします。これにより、関連付けられたコンポーネントの指定された <uninstallSteps> 要素のステップが実行されます。このステップは、<simpleSteps> 要素の子としてしか指定できません。

<uninstall> ステップには次の子要素があります。

- <argList> – オプション要素で、<uninstallSteps> ブロックに渡す引数の一覧。この要素を指定する場合、この要素は 1 回しか出現できません。詳細は、26 ページの「<argList> 要素」を参照してください。
- **installed component targeter –** 必須要素で、アンインストールするコンポーネントを特定します。詳細は、46 ページの「インストール済みコンポーネントターゲッター」を参照してください。

<uninstall> ステップの属性

<uninstall> ステップには entityName 型の 1 つの必須属性 *blockName* があり、これは対象コンポーネント内で実行するアンインストールブロックの名前です。

第 5 章

プラグイン記述スキーマ

この章では、プラグインの定義に使用する XML スキーマを説明します。この章の内容は、次のとおりです。

<plugin> 要素の概要

<plugin> 要素は、プラグインスキーマの最上位の要素です。<plugin> 要素は、プラグインのパーツを特定します。

<plugin> 要素の属性

<plugin> 要素には次の属性があります。

- *name* – プラグインの名前。*name* 属性の最大長は 64 です。プラグイン名は、com.sun.solaris の形式に似た構造に従います。
- *description* (オプション) – プラグインの説明。
- *vendor* – プラグインの提供元。
- *version* – プラグインのバージョン。バージョン番号は標準的な x.y (メジャー.マイナー) の形式に従う必要があります。

- *previousversion* (オプション) – システム上に存在すると想定されている当該プラグインのバージョン。指定されていない場合は、初期インストールが想定されます。指定されている場合、値はプラグインをアップグレードする元のバージョンを表します。
- *xmlns* – 必須値は、`http://www.sun.com/schema/SPS`。
- *xmlns:xsi* – 必須値は、 `http://www.w3.org/2001/XMLSchema-instance`。
- *xsi:schemaLocation* (オプション) – 推奨値は、`http://www.sun.com/schema/SPS plugin.xsd`。
- *schemaVersion* – 使用されているプラグイン XML スキーマのバージョン。現在許可されている値は `5.0` のみです。

<plugin> 子要素

<plugin> 要素は、次の子要素を含むことができます。

- <readme> (オプション) – プラグインの作成者によって書かれた readme ファイルへのパス
- <serverPluginJAR> (オプション) – Master Server で実行するサーバー側のプラグインコードが含まれる JAR ファイルへのパス
- <gui> (オプション) – プラグイン用のオプション GUI 拡張機能
- <dependencyList> (オプション) – 当該プラグインが依存する外部プラグインのリスト
- <memberList> (オプション) – プラグインの一部として作成するメンバーオブジェクトのリスト。これらのメンバーオブジェクトは、任意の数のフォルダ、ホスト型、ホストセット、ホスト検索、コンポーネント、およびプランのオブジェクトを含むことができます。これらのメンバーオブジェクトは、任意の順序で指定できます。

<readme> 要素

<readme> 要素は <plugin> 要素の子で、プラグイン JAR の `readme.txt` ファイルの位置を宣言するために使用されます。`readme.txt` ファイルは、Unicode で符号化されたテキストファイルであると想定されています。Unicode 符号化を指定するには、バイトオーダーマーク (BOM) を使用します。BOM が存在しない場合は、デフォルトで UTF-8 符号化が使用されます。

<readme> 要素には 1 つの属性 *jarPath* があり、これには readme ファイルへのパス名が含まれます。readme ファイルのパス名は、プラグイン JAR ファイルのルートに相対的です。先頭のスラッシュ (/) またはピリオド (.) 文字は、*jarPath* では使用できません。

<serverPluginJAR> 要素

<serverPluginJAR> 要素は <plugin> 要素の子で、Master Server で実行するコードを含むサーバー側プラグイン JAR ファイルの位置の宣言に使用されます。JAR ファイルの位置は、プラグイン JAR ファイルのルートに相対的です。一般的にこの JAR ファイルには、プラグインにより定義されたコンポーネント型の、コンポーネントエクスポータ実装が含まれます。Master Server で実行されるよう外部で書かれたコード (エクスポータクラスなど) は、プラグイン用のエージェントで実行されるコードと同じ JAR ファイル内に存在する必要はありません (同じ位置に存在すること自体は可能)。

<serverPluginJAR> 要素には 1 つの属性 *jarPath* があり、これにはサーバー側プラグイン JAR ファイルへのパス名が含まれます。パス名は、プラグイン JAR ファイルのルートに相対的です。先頭のスラッシュ (/) またはピリオド (.) 文字は、*jarPath* では使用できません。

<gui> 要素

<gui> 要素は <plugin> 要素のオプションの子で、当該プラグインをサポートする GUI 拡張機能のセット用に、独立したプラグイン UI 記述子ファイルの位置を宣言するために使用されます。このプラグイン UI 記述子ファイルの構文は、第 6 章で説明されています。

<gui> 要素には 1 つの属性 *jarPath* があり、これにはプラグイン UI 記述子ファイルへのパス名が含まれます。パス名は、プラグイン JAR ファイルのルートに相対的です。先頭のスラッシュ (/) またはピリオド (.) 文字は、*jarPath* では使用できません。

<dependencyList> 要素

<dependencyList> 要素は <plugin> 要素の子で、当該プラグインが依存するそのほかのプラグインのリストを宣言するために使用されます。この要素には属性はありませんが、当該プラグインの依存性を特定する <pluginRef> 要素が 1 つ以上含まれています。当該プラグインが破棄されると、N1 Grid Service Provisioning System により、これらの依存性と照らし合わせたチェックが行われます。

<pluginRef> 要素

<pluginRef> 要素は <dependencyList> 要素の子で、別のプラグインへの参照の宣言に使用されます。<pluginRef> 要素には次の 2 つの必須属性があります。

- *name* – 参照先プラグインの名前。*name* 属性の最大長は 64 です。プラグイン名は、com.sun.solaris の形式に似た構造に従います。
- *version* – 参照先プラグインの最低限のバージョン。参照先プラグインは、このバージョン以上のシステムにインストールする必要があります。

<pluginRef> 要素には子要素はありません。

<memberList> 要素

<memberList> 要素は <plugin> 要素の子で、当該プラグインの一部であるシステムオブジェクトのリストの宣言に使用されます。これらのオブジェクトは、任意の順序で指定できます。

<memberList> 要素には属性がありませんが、次の子要素の中の最低 1 つの要素を含みます。

- <folder> – フォルダの宣言。
- <hostType> – ホスト型の宣言。
- <hostSet> – ホストセットの宣言。
- <hostSearch> – ホスト検索の宣言。
- <component> – コンポーネントの宣言。
- <plan> – プランの宣言。

<folder> 要素

<folder> 要素は <memberList> 要素の子で、プラグインにより参照されるフォルダの宣言に使用されます。

プラグインは、フルパス名の形式で、作成されるフォルダを指定できます。たとえば /a/b/c という例の場合、a および b は内部フォルダで c はリーフフォルダです。当該プラグインはこのリーフフォルダを所有します。*admin* グループはフォルダ所有者グループとしてリストされ、当該フォルダはプラグインによって所有されていると特定されます。プラグインは、自らが所有するフォルダにのみコンポーネントとプランを作成できます。プラグインが所有するフォルダには、ユーザーはコンポーネント、プランまたはサブフォルダを作成できません。

プラグインの読み込み時に内部フォルダが存在しない場合、内部フォルダが暗黙に作成されます。プラグインは内部フォルダを所有できません。プラグインが作成した内部フォルダの所有者グループは *admin* グループですが、そのフォルダはプラグインに

属しているとは特定されません。内部フォルダが暗黙にプラグインによって所有されるよう、プラグインの作成者が意図した場合、そのようなフォルダを個別に作成する必要があります。上記の例では、まずフォルダ /a が作成され、次にフォルダ /a/b が作成され、さらにフォルダ /a/b/c が作成されます。

所有されている内部フォルダの下位には、所有されていない内部フォルダを作成することはできません。この要件により、プラグインの作成者は、フォルダ階層において所有されているフォルダの間にあるフォルダにコンポーネントやプランを作成できなくなるため、削除のセマンティクスが複雑になります。

ある内部フォルダが存在し、プラグインの読み込み時点では所有されていない場合、その内部フォルダは直接使用されます。内部フォルダが存在し、あるプラグインによって所有されている場合、その内部フォルダは現在のプラグインによって所有されているか、 現在のプラグインが直接依存しているプラグインによって所有されている必要があります。この要件により、複数の協調的なプラグインを、プラグインベンダーによって個別に分配することができます。Java パッケージスタイルの命名規則に従うことで、フォルダを作成する際に、ベンダーはフォルダネームスペースの衝突を回避できます。

<folder> 要素の属性

<folder> 要素には次の 2 つの属性があります。

- *name* – フォルダのパス名。
- *description* (オプション) – フォルダの説明。

<hostType> 要素

<hostType> 要素は <memberList> 要素の子で、プラグインにより参照されるホスト型の宣言に使用されます。システムでホスト型が作成される際に、ホスト型名には暗黙にプラグイン名の接頭辞が付けられます。

<hostType> 要素の属性

<hostType> 要素には次の 2 つの属性があります。

- *name* – ホスト型の名前。*name* 属性の最大長は 32 文字です。名前の先頭は Unicode 文字またはアンダースコア文字 (_) で、そのあとには Unicode 文字、数字、アンダースコア文字 (_)、ドット (.) またはダッシュ (-) が続く必要があります。
- *description* (オプション) – ホスト型の説明。

<varlist> 要素

<hostType> 要素にはオプションの <varlist> 子要素が含まれます。<varlist> 要素は、<hostType> 要素に追加され、後にホストが構成で使用する変数のリストを指定します。

<varlist> 子要素には、1 つ以上の <var> 子要素が含まれます。<var> 要素は、次の 2 つの必須属性を通じて、<hostType> 要素の変数宣言を行います。

- *name* – 変数の名前
- *default* – 変数のデフォルト値

<hostSet> 要素

<hostSet> 要素は <memberList> 要素の子で、プラグインにより参照されるホストセットの宣言に使用されます。プラグインはホストを定義できないため、<hostSet> 要素はホストを含むことができません。プラットフォームホストセットは、システムプラグイン以外のプラグインでは定義できません。システムでホストセットが作成される際に、ホストセット名には暗黙にプラグイン名の接頭辞が付けられます。

<hostSet> 要素の属性

<hostSet> 要素には次の 3 つの属性があります。

- *name* – ホストセットの名前。*name* 属性の最大長は 32 文字です。名前の先頭は Unicode 文字またはアンダースコア文字 (_) で、そのあとには Unicode 文字、数字、アンダースコア文字 (_)、ドット (.) またはダッシュ (-) が続く必要があります。
- *description* (オプション) – ホストセットの説明。
- *unsupported* (オプション) – true である場合、ホストセットはサポートされません。デフォルトは false です。

<hostSet> 要素の子要素

<hostSet> 要素には次の 2 つのオプションの子要素が含まれます。

- <hostSetRef>
- <hostSearchRef>

<hostSetRef> 要素

<hostSetRef> 要素は <hostSet> 要素の子で、サブホストセットを指定します。このホストセットは、当該フラグイン、または当該プラグインが直接依存するプラグインで、事前に定義されている必要があります。別のプラグインで定義されたホストセットへの参照は、com.foo.other#hostSetName のように、そのプラグイン名を含む必要があります。修飾されていない参照は、当該プラグインにより作成されたオブジェクトと見なされます。

<hostSetRef> 要素には 1 つの属性 *name* があります。この属性はホストセット参照の名前を指定します。*name* 属性にはオプションの *pluginName* があり、最大長は 64 文字です。*pluginName* のあとには、 # 区切り文字と、最大長が 32 文字の *hostEntityName* が続きます。

<hostSearchRef> 要素

<hostSearchRef> 要素は <hostSet> 要素の子で、サブホストの検索を指定します。このホスト検索は、当該プラグイン、または当該プラグインが直接依存するプラグインで、事前に定義されている必要があります。別のプラグインで定義されたホスト検索への参照は、com.foo.other#hostSearchName のように、そのプラグイン名を含む必要があります。修飾されていない参照は、当該プラグインにより作成されたオブジェクトと見なされます。

<hostSearchRef> 要素には 1 つの属性 *name* があります。この属性はホスト検索参照の名前を指定します。*name* 属性にはオプションの *pluginName* があり、その最大長は 64 文字で、そのあとには # 区切り文字が続き、さらに (最大長が 32 文字である) *hostEntityName* が続きます。

<hostSearch> 要素

プラグイン <hostSearch> 要素は <memberList> 要素の子で、プラグインにより参照されるホスト検索の宣言に使用されます。システムでホスト検索が作成される際に、ホスト検索名には暗黙にプラグイン名の接頭辞が付けられます。

<hostSearch> 要素の属性

<hostSearch> 要素には次の 2 つの属性があります。

- *name* – ホスト検索の名前。*name* 属性の最大長は 32 文字です。名前の先頭は Unicode 文字またはアンダースコア文字 (_) で、そのあとには Unicode 文字、数字、アンダースコア文字 (_)、ドット (.) またはスラッシュ (-) が続く必要があります。
- *description* (オプション) – ホスト検索の説明。

<hostSearch> 要素の子要素

<hostSearch> 要素には、次の子要素の少なくとも 1 つが含まれます。

- <criteriaList>
- <appTypeCriteria>
- <physicalCriteria>

注 – <criteriaList>、<appTypeCriteria>、および <physicalCriteria> 要素はそれぞれオプションですが、この 3 つの要素のいずれかを指定する必要があります。

<criteriaList> 要素

<criteriaList> 要素は <hostSearch> 要素の子で、<hostSearch> 要素に追加する基準のリストを指定します。<appTypeCriteria> と <physicalCriteria> が指定されていない場合は、<criteriaList> 要素を指定する必要があります。

<criteriaList> 要素には 1 つ以上の <criteria> 要素が含まれます。<criteria> 要素は、名前、一致型、パターンを含む、検索基準を指定します。この要素には次の 3 つの属性があります。

- *name* – 照合するホスト変数の名前。
- *pattern* – 照合するパターン。
- *match* – 基準の一致型。有効な値は、EQUALS または CONTAINS です。デフォルトは EQUALS です。

<appTypeCriteria> 要素

<appTypeCriteria> 要素は <hostSearch> 要素の子で、<hostSearch> 要素に追加するアプリケーション型の基準のリストを指定します。<appTypeCriteria> 要素の引数は属性として表現され、その順序は重要ではありません。値が false であるか、要素が空または未指定である場合、検索実行時にこの基準は無視されます。<criteriaList> および <physicalCriteria> が指定されていない場合、<appTypeCriteria> 要素を指定する必要があります。

<appTypeCriteria> 要素には次の 3 つのオプション属性があります。

- *ms* – true である場合、ホスト検索で MasterServer アプリケーション型を照合します。デフォルトは false です。
- *ld* – true である場合、ホスト検索で LocalDistributor アプリケーション型を照合します。デフォルトは false です。
- *ra* – true である場合、ホスト検索で RemoteAgent アプリケーション型を照合します。デフォルトは false です。

<physicalCriteria> 要素

<physicalCriteria> 要素は <hostSearch> 要素の子で、<hostSearch> 要素に追加する物理型の基準のリストを指定します。<physicalCriteria> 要素の引数は属性として表現され、その順序は重要ではありません。値が false であるか、要素が空または未指定である場合、検索実行時にこの基準は無視されます。<criteriaList> および <appTypeCriteria> が指定されていない場合、<physicalCriteria> 要素を指定する必要があります。

<physicalCriteria> 要素には次の 2 つのオプション属性があります。

- *physical* – true である場合、ホスト検索で物理ホスト型を照合します。デフォルトは false です。
- *virtual* – true である場合、ホスト検索で仮想ホスト型を照合します。デフォルトは false です。

<component> 要素

<component> 要素は <memberList> 要素の子で、プラグイン JAR ファイルでコンポーネントを宣言するために使用されます。このコンポーネントにより参照されるすべてのオブジェクトは、当該プラグイン、または当該プラグインが直接依存するプラグインで、事前に定義されている必要があります。

<component> 要素の属性

<component> 要素には次の 2 つの属性があります。

- *jarPath* – コンポーネント XML ファイルの位置で、プラグイン JAR のルートに相対的です (先頭の / または . 文字は使用できません)。コンポーネント XML の書式は、プランおよびコンポーネント言語の仕様により指定されます。詳細は、第 3 章を参照してください。
- *majorVersion* (オプション) – コンポーネントを新しいメジャーバージョンとしてチェックインするかどうかを決定します。デフォルトは false です。

<component> 要素の子要素

<component> 要素には次の 3 つのオプションの子要素が含まれます。

- <systemService>
- <componentType>
- <resource>

<systemService> 要素

<systemService> 要素は <component> 要素の子で、包含コンポーネントにより返されるシステムサービスの宣言に使用されます。この要素は <componentType> 要素とは併用できません。<systemService> 要素が <component> 要素で使用されている場合、コンポーネントが読み込まれ、そのコンポーネントを参照する <systemServiceRef> が作成されます。システムでシステムサービスが作成される際に、システムサービス名にはプラグイン名の接頭辞が付けられます。

<systemService> 要素には次の 2 つの属性があります。

- *name* – システムサービスの名前
- *description* (オプション) – システムサービスの説明

<componentType> 要素

<componentType> 要素は <component> 要素の子で、包含コンポーネントにより返されるコンポーネント型の宣言に使用されます。<componentType> 要素は <systemService> 要素とは併用できません。<componentType> 要素が

<component> 要素で使用されている場合、コンポーネントが読み込まれ、そのコンポーネントにより返されるコンポーネント型が作成されます。システムでコンポーネント型が作成される際に、コンポーネント型名にはプラグイン名の接頭辞が付けられます。

コンポーネント型は、プラグインによりグループ化され、これらのグルーピング内でのコンポーネント型の順序により順序付けられます。グルーピングは、プラグインの順序に従って順序付けられます。特定のプラグイン内では、コンポーネント型により定義された個々のグループ名の下で、コンポーネント型がインデントされます。

<componentType> 要素には次の 5 つの属性があります。

- *name* – コンポーネント型の名前。

  名前の最大長は 64 文字です。名前は文字またはアンダースコアで始まる必要があり、そのあとには任意の文字、数字、または (アンダースコア (_)、ピリオド (.)、プラス記号 (+)、マイナス記号 (-)、および空白 ( ) などの) 特殊文字が続きます。Unicode 文字および数字が使用できます。
- *description* (オプション) – コンポーネント型の説明。
- *group* – 当該コンポーネントがコンポーネント型の階層の一部である場合の、コンポーネント型のグループ名。

  グループ名は、コンポーネント型名と同じ要件に従います。また、グループは hidden として宣言できるため、このように宣言すると、「component list 」ページのコンポーネント型のドロップダウンリストでは型が表示されません。
- *order* – ブラウザインタフェースのコンポーネント型のドロップダウンリストで、当該コンポーネント型を配置する場所を特定する番号です。

  order は、最大 18 文字です。Unicode 文字および数字のほかに、ASCII キーボードで入力できるすべての文字が使用できます。order によって、特定のプラグイン内で定義されているすべての型を順序付けられる必要があります。
- *indentlevel* – 0 から 10 までの番号で、ブラウザインタフェースのコンポーネント型の階層で当該コンポーネント型をインデントするレベルを特定します。

<resource> 要素

<resource> 要素は <component> 要素の子であり、JAR ファイル内でのリソースファイルの名前と位置を指定します。リソースは、常に単純ファイル型のリソースとしてチェックインされます。<resource> 要素を含むコンポーネントは単純コンポーネントであり、その単純コンポーネントの <resourceRef> 要素は <resource> 要素により作成されたリソースを参照する必要があります。

<resource> 要素には次の 3 つの属性があります。

- *jarPath* – プラグイン JAR ファイルのルートに相対的な、リソースファイルの位置。先頭の / または . 文字は使用できません。
- *majorVersion* (オプション) – 当該リソースを新しいメジャーバージョンとしてチェックインするかどうかを決定します。デフォルトは false です。
- *name* (オプション) – リソースの名前。指定されていない場合、名前のデフォルトは (相対として指定されている場合に絶対に変換される) 絶対 *jarPath* になります。

<plan> 要素

<plan> 要素は <memberList> 要素の子で、プラグイン JAR でプランを宣言するために使用されます。このプランにより参照されるすべてのオブジェクトは、当該プラグイン、または当該プラグインが直接依存するプラグインで、事前に定義されている必要があります。

<plan> 要素の属性

<plan> 要素には次の 2 つの属性があります。

- *jarPath* – プラグイン JAR ファイルのルートに相対的な、プラン XML の位置。先頭の / または . 文字は使用できません。プラン XML の書式は、プランおよびコンポーネント言語の仕様により指定されます。詳細は、第 4 章を参照してください。
- *majorVersion* (オプション) – 当該プランを新しいメジャーバージョンとしてチェックインするかどうかを決定します。デフォルトは false です。

<plugin> 要素のサンプル XML

例 **5–1** サンプルのプラグイン記述子ファイル

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin name="com.bigCo.logic.pluginName"
        description="imitation WL plugin"
        vendor="bigCo"
        version="1.3"
        previousVersion="1.2"
        xmlns="http://www.sun.com/schema/SPS"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.sun.com/schema/SPS
                            plugin.xsd"
        schemaVersion="5.0">
     <readme jarPath="docs/readme.txt"/>
     <serverPluginJAR jarPath="lib/appserver/serverCode.jar"/>
     <gui jarPath="custom/weblogic/gui/pluginUI.xml"/>
     <dependencyList>
        <pluginRef name="webLogicUtils" version="1.0"/>
        <pluginRef name="otherPlugin" version="1.3"/>
     </dependencyList>
     <memberList>
        <folder name="/com/bea/weblogic/6.0" description="Weblogic 6.0 plugin folder"/>
        <folder name="/folder2" description="second place sees dust"/>
        <hostType name="WL Admin Server" description="Host Type for Weblogic Admin Servers">
           <varList>
               <var name="adminPort" default="7001"/>
               <var name="adminUser" default="weblogic"/>
```

例 **5–1** サンプルのプラグイン記述子ファイル　(続き)

```
            <var name="secureConnect" default="false"/>
         </varList>
      </hostType>
      <hostSet name="Weblogic Admin Servers" description="The Weblogic Admin Servers">
         <hostSetRef name="WL boxes"/>
         <hostSearchRef name="WL Admin Search"/>
      </hostSet>
      <hostSearch name="WL box search" description="matches Weblogic boxes">
         <criteriaList>
            <criteria name="sys.OS" match="CONTAINS" pattern="SunOS"/>
            <criteria name="sys.OSVersion" pattern="5.9"/>
         </criteriaList>
         <appTypeCriteria ms="false" ld="false" ra="true"/>
         <physicalCriteria physical="true" virtual="true"/>
      </hostSearch>
      <hostSet name="Weblogic Servers" description="All Weblogic Servers">
         <hostSetRef name="Weblogic Admin Servers"/>
         <hostSetRef name="com.bigCo.logic.cluster#Weblogic Clusters"/>
      </hostSet>
      <component jarPath="comps/system/weblogic/foo.xml" majorVersion="true">
         <componentType name="contained EJB CT"
                   description="contained ejb comp type ref"
                   group="hidden"
                   order="001-003-002"
                   indentLevel="2"/>
      </component>
      <component jarPath="weblogic/system/comps/bar.xml">
         <systemService name="WebLogic SS" description="WL service ref"/>
      </component>
      <component jarPath="weblogic/system/comps/baz.xml"/>
      <plan jarPath="weblogic/system/plans/bar.xml" majorVersion="false"/>
      <component jarPath="weblogic/system/comps/dee.xml">
         <resource jarPath="weblogic/system/plugin-core.jar" majorVersion="true"/>
      </component>
  </memberList>
</plugin>
```

第 6 章

プラグインユーザーインタフェーススキーマ

この章では、プラグインのユーザーインタフェースの定義に使用する、XML スキーマを説明します。この章の内容は、次のとおりです。

<pluginUI> 要素の概要

<pluginUI> 要素を使用すると、プラグインの作成者は、カスタムのショートカットページで表示される機能の制限付きセットを記述できます。機能セットはショートカットを介して表示されます。また、次のカテゴリに分類されています。

- コンポーネント型ショートカット:
  - 当該コンポーネント型を拡張するすべてのコンポーネントを一覧表示する
  - 当該コンポーネント型を拡張するコンポーネントを作成する
- コンポーネントショートカット:
  - 指定のコンポーネントを管理する (「component details」ページへのリンク)
  - 当該コンポーネントがインストールされている仮想ホストおよび物理ホストを一覧する
- プランショートカット:
  - 指定のプランを管理する (「plan details」ページへのリンク)

<pluginUI> 要素属性

<pluginUI> 要素には次の属性があります。

- *menuItem* – ブラウザインタフェースのメニューで表示するテキスト。名前の長さは 20 文字以下にします。実際の文字制限は、属性型により定義されます。
- *tooltip* (オプション) – ブラウザインタフェースのメニュー項目で表示するツールヒント。アイコンを用意することを選択した場合、*menuItem* にはアイコンが含まれます。
- *xmlns* – 必須値は、http://www.sun.com/schema/SPS 。
- *xmlns:xsi* – 必須値は、 http://www.w3.org/2001/XMLSchema-instance 。
- *xsi:schemaLocation* (オプション) – 推奨値は、
  http://www.sun.com/schema/SPS pluginUI.xsd。
- *schemaVersion* – 使用中の pluginUI.xsd スキーマのバージョン。現在許可されている値は 5.0 のみです。

<pluginUI> 子要素

<pluginUI> 要素には次の要素が含まれます。

- <icon> – 当該プラグインのインタフェース内で表示させる、グラフィック (アイコン) へのパスを提供する
- <customPage> – ブラウザインタフェースのメニュー項目のリンク先カスタムページのコンテンツを定義する

<icon> 要素

<icon> 要素は <pluginUI> 要素の子です。<icon> 要素は、プラグインアイコンの位置を宣言します。アイコンは GIF または JPEG ファイル形式であることが想定されています。アイコンの次元は、幅 32 ピクセル、高さ 26 ピクセルである必要があります。

<icon> 要素属性

<icon> 要素には 1 つの必須属性 *jarPath* があります。*jarPath* 属性は、プラグイン JAR ファイルのルートに相対的な、プラグインアイコンの位置を指定します。先頭の / または . 文字は使用できません。

<customPage> 要素

<customPage> 要素は <pluginUI> 要素の子で、ブラウザインタフェースのメニュー項目のリンク先カスタムページのコンテンツを定義します。 <customPage> 要素は、1 つ以上の <section> 要素を含み、*name* 属性を持っています。

<customPage> 要素属性

<customPage> 要素には、1 つの必須属性 *name* があります。*name* 属性は、カスタムページのブレッドクラムおよびタイトルセクションで使用されます。

<section> 要素

<section> 要素は <customPage> 要素の子で、カスタムページのセクションを定義します。<section> 要素には 1 つ以上の <entry> 要素があり、2 つの属性を持っています。

- *title* – セクションのタイトル
- *description* (オプション) – セクションの説明 (副次的なテキスト)

<entry> 要素

<entry> 要素は <section> 要素の子で、ユーザー操作のエントリポイントを定義します。<entry> 要素は 0 個以上の <action> 要素があり、次の属性を持っています。

- *title* – エントリのタイトル
- *description* (オプション) – エントリの説明

<action> 要素

<action> 要素は <entry> 要素の子で、ユーザー操作を定義します。各 <action> 要素は、必ず 1 つの子要素を含む必要があります。

<action> 要素には次の 2 つの属性があります。

- *text* – ユーザー操作のリンク用にレンダリングされるテキスト
- *tooltip* (オプション) – ユーザー操作のリンクとして使用されるツールヒント

各 <action> 要素には、次の子要素のいずれかが含まれている必要があります。

<compCreate>	<compCreate> 要素は <action> 要素の子で、名前が指定されたコンポーネント型の「component create」ページへのリンクを定義します。

<compCreate> 要素には次の 1 つの必須属性があります。

- *typeName* – コンポーネント型の名前。コンポーネント型は、当該プラグイン、または当該プラグインが直接依存するプラグインに含まれている必要があります。コンポーネント型を hidden として定義することはできません。pluginName は、fullPluginName#componentTypeName のように、コンポーネント型名の接頭辞である必要があります。

<compDetails>

<compDetails> 要素は <action> 要素の子で、名前付きコンポーネントの最新バージョンに関する「component details」ページへのリンクを定義します。

<compDetails> 要素には次の 2 つの必須属性があります。

- *path* – コンポーネントの絶対パス。
- *name* – コンポーネントの名前。コンポーネントは、当該プラグイン、または当該プラグインが直接依存するプラグインに含まれている必要があります。

<compList>

<compList> 要素は <action> 要素の子で、名前が指定されたコンポーネント型によりフィルタリングされる「component list」ページへのリンクを定義します。

<compList> 要素には次の 1 つの必須属性があります。

- *typeName* – コンポーネント型の名前。コンポーネント型は、当該プラグイン、または当該プラグインが直接依存するプラグインに含まれている必要があります。コンポーネント型を hidden として定義することはできません。pluginName は、fullPluginName#componentTypeName のように、コンポーネント型名の接頭辞である必要があります。

<compWhereInstalled>

<compWhereInstalled> 要素は <action> 要素の子で、名前付きコンポーネントの最新バージョンに関する「Component Where Installed」ページへのリンクを定義します。

<compWhereInstalled> 要素には次の 2 つの必須属性があります。

- *path* – コンポーネントの絶対パス。

- *name* – コンポーネントの名前。コンポーネントは、当該プラグイン、または当該プラグインが直接依存するプラグインに含まれている必要があります。

`<hostList>`

`<hostList>` 要素は `<action>` 要素の子で、名前が指定されたホスト型によりフィルタリングされる「host list」ページへのリンクを定義します。

`<hostList>` 要素には次の 1 つの必須属性があります。

- *typeName* – ホスト型の名前。ホスト型は、当該プラグイン、または当該プラグインが直接依存するプラグインに含まれている必要があります。ホスト型を `hidden` として定義することはできません。`pluginName` は、`fullPluginName#hostTypeName` のように、ホスト型名の接頭辞である必要があります。

`<planDetails>`

`<planDetails>` 要素は `<action>` 要素の子で、名前付きプランの最新バージョンに関する「plan details」ページへのリンクを定義します。

`<planDetails>` 要素には次の 2 つの必須属性があります。

- *path* – プランの絶対パス。
- *name* – プランの名前。プランは、当該プラグイン、または当該プラグインが直接依存するプラグインに含まれている必要があります。

`<pluginUI>` 要素のサンプル XML

例 **6–1** サンプル `<pluginUI>` 記述子ファイル

```
<?xml version="1.0" encoding="UTF-8"?>
<pluginUI menuItem="pluginName"
          tooltip="view wl server pages"
          xmlns="http://www.sun.com/schema/SPS"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.sun.com/schema/SPS
                              pluginUI.xsd"
          schemaVersion="5.0">
  <icon jarPath="custom/gui/img/WLicon-small.gif"/>
  <customPage name="WebLogic">
    <section title="WebLogic application tasks"
```

例 **6–1** サンプル <pluginUI> 記述子ファイル　*(続き)*

```
        description="capture and edit your WebLogic applications...">
  <entry title="enterprise applications (EARs)"
        description="capture, edit and deploy your enterprise applications">
    <action text="view all" toolTip="view all EARs">
      <compList typeName="com.bigCo.logic.pluginName.WebLogic enterprise application"/>
    </action>
    <action text="create new" toolTip="create new enterprise application">
      <compCreate typeName="com.bigCo.logic.pluginName.WebLogic enterprise application"/>
    </action>
  </entry>
  <entry title="web applications (WARs)"
        description="capture, edit and deploy web applications">
    <action text="view all" toolTip="view all WARs">
      <compList typeName="com.bigCo.logic.pluginName.WebLogic web application"/>
    </action>
    <action text="create new" toolTip="create new webapp">
      <compCreate typeName="com.bigCo.logic.pluginName.WebLogic web application"/>
    </action>
  </entry>
  <entry title="java archives containing EJBs (JARs)"
        description="capture, edit and deploy your JARS containing EJBs">
    <action text="view all" toolTip="view all JARs">
      <compList typeName="com.bigCo.logic.pluginName.WebLogic EJB"/>
    </action>
    <action text="create new" toolTip="create new java archive containing EJBs">
      <compCreate typeName="com.bigCo.logic.pluginName.WebLogic EJB"/>
    </action>
  </entry>
</section>
<section title="WebLogic infrastructure"
        description="create and edit your WebLogic infrastructure...">
  <entry title="admin servers"
        description="WebLogic domains / administration servers">
    <action text="manage admin servers" toolTip="manage WebLogic admin servers">
      <compDetails path="/com/bea/weblogic" name="WL Admin Server 7.0"/>
    </action>
    <action text="view admin servers" toolTip="list of WebLogic admin servers">
      <compWhereInstalled path="/com/bea/weblogic" name="WL Admin Server 7.0"/>
    </action>
  </entry>
  <entry title="clusters"
        description="WebLogic clusters">
    <action text="manage clusters" toolTip="manage WebLogic clusters">
      <compDetails path="/com/bea/weblogic" name="WL Cluster"/>
    </action>
    <action text="view clusters" toolTip="list of WebLogic clusters">
      <compWhereInstalled path="/com/bea/weblogic" name="WL Cluster"/>
    </action>
  </entry>
  <entry title="managed servers"
        description="WebLogic server instances">
    <action text="manage server instances" toolTip="WebLogic managed servers">
```

例 **6–1** サンプル <pluginUI> 記述子ファイル　(続き)

```
          <compDetails path="/com/bea/weblogic" name="WL Managed Server"/>
        </action>
        <action text="view managed servers" toolTip="list of WebLogic managed servers">
          <compWhereInstalled path="/com/bea/weblogic" name="WL Managed Server"/>
        </action>
         <action text="update managed servers" toolTip="run a plan on managed servers">
          <planDetails path="/com/bea/weblogic/updates" name="updatePlan"/>
        </action>
     </entry>
    </section>
  </customPage>
</pluginUI>
```

付録 A

コンポーネント変更の互換性

この付録では、コンポーネントに加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

コンポーネントには次の変更を加えることができます。

コンポーネントに加えることができる変更

<component> 要素の変更

次の表に、<component> 要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
非最終から最終	なし	あり
最終から非最終	あり	あり
非抽象から抽象	なし	あり
抽象から非抽象	あり	あり
アクセス制限の強化	なし	なし
アクセス制限の緩和	あり	あり
description、*label*、*softwareVendor*、または *author* 属性の値の変更	なし[1]	あり
name または *path* 属性の値の変更[2]	なし	あり
単純コンポーネントから複合コンポーネントへの変更	なし	なし
複合コンポーネントから単純コンポーネントへの変更	なし	なし

[1] 属性の値は、インストールされた変数設定レコードに格納されます。

[2] この変更は、完全にバージョンツリーの変更から構成され、システムサービスが更新される状況でのみ実施可能です。この場合、新しいコンポーネントは元のコンポーネントのインスタンスである必要があります。

platform 属性の変更

次の表に、<platform> 要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
より一般的なプラットフォームへの変更	あり	あり

変更の種類	インストール互換	呼び出し互換
より固有のプラットフォームへの変更	なし	あり
関係性のないプラットフォーム	なし	あり

プラットフォーム X が Y の子孫の場合、プラットフォーム X は Y より固有となります。X が Y の祖先の場合、プラットフォーム X は Y より一般的になります。

limitToHostSet 属性の変更

次の表に、<limitToHostSet> 要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
limitToHostSet 属性へのすべての変更	なし	あり

platform 属性とは異なり、*limitToHostSet* は明示的な制御が存在しない一般的なユーザー指定のホストセットを指定します。ホストセットのメンバーシップはいつでも変更される可能性があるため、固有性の高いホストセットまたは固有性の低いホストセットは指定できません。

<extends> 要素の変更

次の表に、<extends> 要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
オリジナルコンポーネントの新しいベースコンポーネントインスタンス	なし	あり
新しいコンポーネントのオリジナルベースコンポーネントインスタンス	なし	なし
オリジナルコンポーネントと無関係な新しいベースコンポーネント	なし	なし
オリジナルコンポーネントとインストール互換である新しいベースコンポーネント	あり	あり
オリジナルコンポーネントと呼び出し互換である新しいベースコンポーネント	なし	あり

変数への変更

次の表に、変数に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
新しい変数の追加	あり[1]	あり
非専用変数の削除または名前変更	なし	なし
専用変数の削除または名前変更	あり	あり
最終変数のデフォルト値の変更	なし	あり
非最終変数のデフォルト値の変更	あり[2]	あり
prompt 属性の変更	あり	あり
非最終から最終	なし	あり
最終から非最終	あり	あり
非抽象から抽象	なし	あり
抽象から非抽象	あり	あり
アクセス制限の強化	なし	なし
アクセス制限の緩和	あり[1]	あり

[1] より制限の厳しいアクセスモードを使用して、すでに変数を定義している派生コンポーネントが存在している場合があります。このような場合は、変更により発生コンポーネントが無効になります。コンポーネント型に新しい非抽象変数を追加できるのは、同じ名前を持つ変数をすでに定義している派生コンポーネントが存在せず、かつコンポーネントのすべてのインストール済みインスタンスに対して変数のデフォルト値を再計算できる場合のみです。

[2] インストール済みの値は、新しいデフォルト値をオーバーライドすると見なすことができるため、再インストールは必要ありません。

<targetRef> 要素の変更

次の表に、<targetRef> 要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
<targetRef> 要素の削除	なし	なし
<targetRef> 要素の追加	なし	あり
hostName 属性の変更	あり[1]	あり

[1] 既存のインストール済みコンポーネントに関連付けられたホストの属性は、このような変更の結果としては更新されません。

変更の種類	インストール互換	呼び出し互換
typeName 属性の変更	なし	なし
<agent> 子要素の追加または削除	なし	なし
<agent> 子要素の *connection* 属性の変更	あり[1]	あり
<agent> 子要素の *ipAddr* 属性の変更	あり[1]	あり
<agent> 子要素の *port* 属性の変更	あり[1]	あり
<agent> 子要素の *params* 属性の変更	あり[1]	あり

[1] 既存のインストール済みコンポーネントに関連付けられたホストの属性は、このような変更の結果としては更新されません。

<componentRefList> 要素の変更

次の表に、<componentRefList> 要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
非最終から最終	なし	あり
最終から非最終	あり	あり
オリジナルコンポーネントの新しい型インスタンス	なし	あり
新しいコンポーネントのオリジナルの型インスタンス	なし	なし
オリジナルと関連性のない新しい型	なし	なし
オリジナルとインストール互換である新しい型	あり	あり
オリジナルと呼び出し互換である新しい型	なし	あり

<componentRef> 要素の変更

次の表に、<componentRef> 要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
非最終から最終	あり[1]	あり

[1] より制限の厳しいアクセスモードを使用するか、互換性のない差異を使用して、すでにコンポーネント参照を定義している派生コンポーネントが存在している場合があります。このような場合は、変更により派生コンポーネントが無効になります。新しい非抽象コンポーネント参照をコンポーネント型に追加できるのは、同じ名前を持つコンポーネント参照をすでに定義している派生コンポーネントが存在しない場合のみです。

変更の種類	インストール互換	呼び出し互換
最終から非最終	あり	あり
非抽象から抽象	なし	あり
抽象から非抽象	あり	あり
installMode 属性の変更	なし	なし
新しいコンポーネント参照の追加	あり[1]、[2]	あり
入れ子にされた <componentRef> の削除または名前変更	なし	なし
最上位 <componentRef> の削除または名前変更	なし	なし
入れ子コンポーネントの <argList> からの、引数の追加、変更、または削除	なし	あり
最上位コンポーネントの <argList> からの、引数の追加、変更、または削除	あり[3]	あり
オリジナルコンポーネントの新しい型インスタンス	なし	あり
新しいコンポーネントのオリジナルの型インスタンス	なし	なし
オリジナルと関連性のない新しい型	なし	なし
オリジナルとインストール互換である新しい型	あり	あり
オリジナルと呼び出し互換である新しい型	なし	あり
オリジナルコンポーネントの新しい入れ子コンポーネントインスタンス	なし	あり
新しいコンポーネントのオリジナルの入れ子コンポーネントインスタンス	なし	なし
オリジナルと関係のない新しい入れ子コンポーネント	なし	なし
オリジナルとインストール互換の新しい入れ子コンポーネント	あり	あり
オリジナルと呼び出し互換の新しい入れ子コンポーネント	なし	あり

[1] より制限の厳しいアクセスモードを使用するか、互換性のない差異を使用して、すでにコンポーネント参照を定義している派生コンポーネントが存在している場合があります。このような場合は、変更により派生コンポーネントが無効になります。新しい非抽象コンポーネント参照をコンポーネント型に追加できるのは、同じ名前を持つコンポーネント参照をすでに定義している派生コンポーネントが存在しない場合のみです。

[2] 入れ子コンポーネントの追加は、技術上は可能です。既存のインストールは、入れ子コンポーネントを使用せずにインストールされたかのように扱われます。インストールに依存する機能が破損する可能性があるため、入れ子コンポーネントを使用せずにコンポーネントが安全に動作する場合にのみ、この変更を行うことをお勧めします。上記の条件が満たされない場合、この変更は、インストール互換ではない変更として扱う必要があります。

[3] 当該コンポーネントが、最上位コンポーネントを実際にインストールしたコンポーネントであるかどうかは判別できません。そのため、当該コンポーネントは、<argList> 値に対応する変数を持っている最上位コンポーネントに依存することはできません。

リソースの変更

次の表に、リソースに加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
非最終から最終	なし	あり
最終から非最終	あり	あり
非抽象から抽象	なし	あり
抽象から非抽象	あり	あり
installPath、*name*、*group*、または *user* 属性の変更	なし	あり
rsrcName または *rsrcVersion* 属性の変更	なし	あり

<install>、<control>、および <uninstall> ブロックの変更

次の表に、<install>、<control>、または <uninstall> ブロックに加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
非最終から最終	あり[1]	あり
最終から非最終	あり	あり
非抽象から抽象	なし	あり
抽象から非抽象	あり	あり
アクセス制限の強化	なし	なし
アクセス制限の緩和	あり[1]	あり
新しい非専用ブロックの追加	あり[1]	あり
新しい専用ブロックの追加	あり	あり
非専用ブロックの削除または名前変更	なし	なし
専用ブロックの削除または名前変更	あり	あり

[1] より制限の厳しいアクセスモードを使用するか、互換性のない差異を使用して、すでにブロックを定義している派生コンポーネントが存在している場合があります。このような場合は、変更により派生コンポーネントが無効になります。新しい非抽象ブロックをコンポーネント型に追加できるのは、同じ名前を持つブロックをすでに定義している派生コンポーネントが存在しない場合です。

変更の種類	インストール互換	呼び出し互換
ブロック本体の変更	あり[2]	あり
ブロックのローカル変数の追加、変更、または削除	あり[2]	あり
専用ブロックのパラメータの追加、変更、または削除	あり	あり
必須パラメータの非専用ブロックへの追加	なし	なし
オプションパラメータの追加	あり	あり
オプション/必須パラメータの削除	あり[3]	あり[3]
オプションのパラメータの名前変更	あり[4]	あり[4]
非専用ブロックでの必須パラメータの名前変更	なし	なし
非専用ブロックでのパラメータのオプションから必須への変更	なし	なし
パラメータの必須からオプションへの変更	あり	あり
パラメータの *displayMode* 属性の変更	あり[2]	あり
パラメータの *prompt* 属性の変更	あり	あり

[2] 当該ブロックの以前の実行のプラン実行履歴は更新されません。そのため、以前の実行は新しいブロックコンテンツと直接一致しない場合があります。

[3] 呼び出し元から渡される余分な引数は無視されるため、この変更が可能になります。

[4] この変更は、オプションのパラメータを削除して追加することと同じです。ただし、当初渡されたパラメータ値が無視され、デフォルト値に置き換えられるため、呼び出し元は想定外の結果を受け取る可能性があります。

<snapshot> ブロックへの変更

次の表に、<snapshot> 要素の、<prepare>、<cleanup>、または <capture> 子要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。<prepare>、<capture>、および <cleanup> ブロックを処理する場合以外、<snapshot> ブロックの互換性マトリックスは、基本的にほかのブロックのマトリックスと同じです。

変更の種類	インストール互換	呼び出し互換
<prepare>/<cleanup> ブロックの追加、変更、または削除	あり	あり
<capture> 手順の追加、変更、または削除	あり[1]	あり

[1] snapshot <capture> ブロックのコンテンツは 1 回しか評価されません。初期インストール中にスナップショットが取られる際に、評価が行われます。比較時には、比較を行うために、格納されたキャプチャコンテンツが使用され、<capture> ブロックに発生した可能性のある変更は無視されます。そのため、既存のスナップショットはこのような変更の影響を受けません。変更の目的が既存のスナップショットのコンテンツに影響を与えることである場合、この変更は、インストール互換ではない変更としてモデル化される必要があります。

<diff> 要素の <ignore> 子への変更

次の表に、<diff> 要素の <ignore> 子要素に加えることができる変更と、各変更がインストール互換/呼び出し互換であるかどうかを示します。

変更の種類	インストール互換	呼び出し互換
<ignore> 要素の追加、変更、または削除	あり	あり

比較を実行しても既存のスナップショットの状態に影響しない場合にのみ、<ignore> 要素が考慮されます。

索引

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

X

あ

い

き

へ

も

よ

り

ろ