Oracle® Solaris 11.3 での Image Packaging System を使用したソフトウェアのパッケージ化と配布

Part No: E62864
2016 年 11 月

Part No: E62864

ドキュメントのアクセシビリティについて

オラクルのアクセシビリティについての詳細情報は、Oracle Accessibility ProgramのWeb サイト(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc)を参照してください。

Oracle Supportへのアクセス

サポートをご契約のお客様には、My Oracle Supportを通して電子支援サービスを提供しています。詳細情報は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info) か、聴覚に障害のあるお客様は (http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs)を参照してください。

目次

このドキュメントの使用法 11

1 IPS の設計目標、概念、および用語 13
IPS の設計目標 13
ソフトウェアの自己アセンブリ 15
ソフトウェアの自己アセンブリ用のツール 16
Oracle Solaris でのソフトウェア自己アセンブリの例 17
IPS パッケージのライフサイクル 19
IPS 用語およびコンポーネント 21
インストール可能なイメージ 21
パッケージ識別子: FMRI 21
パッケージの内容: アクション 25
パッケージリポジトリ 41

2 IPS を使用したソフトウェアのパッケージ化 43
パッケージの設計 43
パッケージの作成および発行 44
パッケージマニフェストを生成する 45
生成されたマニフェストに必要なメタデータを追加する 46
依存関係を評価する 49
必要とされるファセットまたはアクチュエータがあれば追加する 51
パッケージを検証する 52
パッケージを発行する 53
パッケージの署名 55
パッケージをテストする 55
パッケージを配布する 57
SVR4 パッケージから IPS パッケージへの変換 59
SVR4 パッケージから IPS パッケージマニフェストを生成する 60
変換されたパッケージを検証する 62

パッケージ変換に関するその他の考慮事項 63

3 ソフトウェアパッケージのインストール、削除、および更新 65
パッケージの変更が行われる方法 65
入力にエラーがないか確認する 66
システムの終了状態を決定する 66
基本的なチェックを実行する 66
ソルバーを実行する 67
ソルバーの結果を最適化する 67
アクションを評価する 68
内容をダウンロードする 68
アクションを実行する 69
アクチュエータを処理する 69
ブートアーカイブを更新する 69

4 パッケージ依存関係の指定 71
依存関係の種類 71
`require` 依存関係 71
`require-any` 依存関係 72
`optional` 依存関係 72
`conditional` 依存関係 73
`group` 依存関係 73
`group-any` 依存関係 74
`origin` 依存関係 75
`incorporate` 依存関係 76
`parent` 依存関係 77
`exclude` 依存関係 77
制約と凍結 77
インストール可能なパッケージバージョンの制約 78
インストール可能なパッケージバージョンの凍結 79
管理者がインストール可能なパッケージバージョンへの制約を緩和できるようにする 79

5 バリエーションの許可 81
相互に排他的なソフトウェアコンポーネント 81
オプションのソフトウェアコンポーネント 83

6 プログラムによるパッケージマニフェストの変更 85
変換規則 85
取り込み規則 86
変換順序 87
パッケージ化された変換 88

7 パッケージインストールの一環としてのシステム変更の自動化 89
パッケージアクションでのシステム変更の指定 89
SMF サービスの提供 91
1 度だけ実行されるサービスの提供 91
フラグメントファイルからのカスタムファイルの作成 95

8 パッケージ更新の高度なトピック 99
パッケージ内容の競合の回避 99
パッケージ内容の変更 100
パッケージの名前変更、マージ、および分割 101
単一パッケージの名前変更 101
2 つのパッケージのマージ 102
パッケージの分割 102
パッケージの廃止 103
移行する編集可能なパッケージ化ファイルの保持 103
パッケージ解除されたファイルの保持 104
ディレクトリの削除に伴うパッケージ解除されたファイルの移動 104
ディレクトリの個別パッケージ化 105
ブート環境間での内容の共有 107
Oracle Solaris での既存の共有内容 108
共有領域への内容の配布 108
別のパッケージでも配布されるファイルの配布 115
複数のアプリケーション実装の配布 116
調停されたリンクの属性 118
調停されたリンクの指定 119
調停されたリンクのベストプラクティス 121

9 IPS パッケージの署名 123
署名アクション 123
署名付きパッケージ 124
カスタム証明書認証局証明書の使用 125
▼ カスタム証明書認証局証明書の使用方法 125

署名付きパッケージのトラブルシューティング 126
イメージとパブリッシャーのプロパティーの構成 127
チェーン証明書が見つからない 129
承認された証明書が見つからない 129
自己署名付き証明書が信頼できない 130
署名の値が期待された値と一致しない 130
重要な拡張機能が不明である 131
拡張機能の値が不明である 131
証明書の承認されていない使用 131
予期しないハッシュ値 132
証明書が失効している 132

10 非大域ゾーンの処理 133
非大域ゾーンについてのパッケージ化の考慮事項 133
パッケージが大域ゾーンと非大域ゾーンの間の境界を超えるか 133
パッケージのどこまでを非大域ゾーンにインストールするか 134
非大域ゾーンへのパッケージのインストールに関するトラブルシューティング 135
自分自身への parent 依存関係を持つパッケージ 135
自分自身への parent 依存関係を持たないパッケージ 135

11 発行されたパッケージの変更 137
パッケージの再発行 137
パッケージのメタデータの変更 138
パッケージパブリッシャーの変更 138

A パッケージの分類 141
分類の割り当て 141
分類の値 141

B IPS を使用して Oracle Solaris OS をパッケージ化する方法 145
Oracle Solaris パッケージのバージョン管理 145
Oracle Solaris 結合パッケージ 147
依存関係の制約の緩和 147
Oracle Solaris グループパッケージ 148
属性とタグ 148
情報属性 148

Oracle Solaris 属性 149
組織固有の属性 149
Oracle Solaris タグ 150

このドキュメントの使用法

- **概要** – Oracle Solaris Image Packaging System (IPS) 機能を使用して、Oracle Solaris 11 オペレーティングシステム (OS) のソフトウェアパッケージを作成する方法を説明します。
- **対象読者** – Oracle Solaris 11 OS にインストールし、IPS を使用して保守できるパッケージを作成するソフトウェア開発者と、IPS の理解を深め、IPS を使用して Oracle Solaris OS がどのようにパッケージ化されるかを理解したい開発者およびシステム管理者。
- **前提知識** – IPS および Oracle Solaris Service Management Facility (SMF) 機能の使用経験

製品ドキュメントライブラリ

この製品および関連製品のドキュメントとリソースは http://www.oracle.com/pls/topic/lookup?ctx=E62101-01 で入手可能です。

フィードバック

このドキュメントに関するフィードバックを http://www.oracle.com/goto/docfeedback からお聞かせください。

♦ ♦ ♦ 第 1 章

1 IPS の設計目標、概念、および用語

このガイドでは、Oracle Solaris 11 OS にインストールでき、IPS を使用して保守できるパッケージを作成する方法と、IPS を使用して Oracle Solaris OS をパッケージ化する方法を説明します。

この章では、次の内容について説明します。

- ユーザーがより高度な IPS 機能を理解して使用できるようにするための IPS の設計概念
- ソフトウェアの自己アセンブリ: インストール済みソフトウェアが自分自身を作業用構成に構築するための能力
- IPS パッケージのライフサイクルのフェーズ
- インストール可能なイメージ、パッケージパブリッシャー、パッケージアクションなどの概念

IPS の設計目標

IPS は、Oracle Solaris の顧客、開発者、保守管理者、および ISV にとって重大な問題を引き起こしている以前のソフトウェア配布、インストール、および保守メカニズムに関する長期にわたる問題を取り除くために設計されたものです。

IPS の主な設計目標には次が含まれます。

ダウンタイムを最小限に抑える。

計画されたダウンタイムを最小限に抑えるには、システムの稼働中にソフトウェア更新を可能にします。

計画外のダウンタイムを最小限に抑えるには、既知の作業用ソフトウェア構成への高速リブートをサポートします。

インストールおよび更新を自動化する。

新しいソフトウェアと、既存ソフトウェアへの更新のインストールをできるだけ自動化します。

メディア要件を少なくする。

増え続けるソフトウェアサイズと限られた配布メディア容量の問題を解消します。

正しいソフトウェアインストールを検証する。

パッケージの作成者 (パブリッシャー) によって定義されたとおりにパッケージが正しくインストールされるかどうかを確認できるようにしてください。このようなチェックはスプーフィング可能でないようにしてください。

簡単な仮想化を可能にする。

さまざまなレベル (特にゾーンを使用するレベル) で Oracle Solaris の簡単な仮想化を可能にするメカニズムを取り入れます。

アップグレードを簡素化する。

既存のシステム用のパッチまたはアップグレードを生成するために必要な労力を減らします。

簡単なパッケージ作成を可能にする。

ほかのソフトウェアパブリッシャー (ISV やエンドユーザー自身) が Oracle Solaris のパッケージを簡単に作成および発行できるようにします。

これらの目標は、次の考えにつながりました。

ブート環境を必要に応じて作成する。

ZFS スナップショットおよびクローン機能を活用して、ブート環境を必要に応じて動的に作成します。

- Oracle Solaris 11 ではルートファイルシステムとして ZFS が必要であるため、ゾーンファイルシステムも ZFS 上にある必要があります。
- ユーザーはブート環境を必要な数だけ作成できます。
- IPS は、実行中のシステムを変更する前のバックアップの目的で、または新しいバージョンの OS のインストールのために、ブート環境を必要に応じて自動的に作成できます。

インストール、パッチ、および更新を統合する。

インストール、パッチ、および更新に使用する、重複したメカニズムとコードを削除します。

この考えによって、Oracle Solaris の保守方法に、次の重要な例を含む、いくつかの重大な変更が加えられました。

- OS ソフトウェアの更新とパッチの適用はすべて IPS で直接行われます。
- 新しいパッケージがインストールされるときはいつでも、それはすでに正確に正しいバージョンになっています。

インストールが不適切に行われる機会を最小限に抑える。

パッケージインストールでスプーフィング不可能な検証が要求される結果、次の結論に達します。

- パッケージを複数の方法でインストールできるようにする必要がある場合は、検証プロセスでこれが考慮に入れられるように開発者がそれらの方法を指定する必要があります。
- パッケージシステムではスクリプト作成者の意図を判断できないため、スクリプト作成は本質的に検証不可能です。これは、後述する他の問題とともに、パッケージ化の操作中のスクリプト作成の除去につながりました。
- そのあとの検証が不可能なため、パッケージにそれ自身のマニフェストを編集するメカニズムを含めることはできません。
- 管理者が元のパブリッシャーの定義と互換性のない方法でパッケージをインストールする場合は、管理者の変更の範囲がアップグレードにより失われることなく明確で、元のパッケージと同じ方法で検証できるように、パッケージシステムでは管理者が変更するパッケージを簡単に再発行できるようにするべきです。

ソフトウェアリポジトリを提供する。

サイズ制限を回避する必要性は、いくつかの異なる方法を使用してアクセスされるソフトウェアリポジトリモデルにつながりました。さまざまなリポジトリソースを複合して完全なパッケージセットを提供できるので、複数のリポジトリを 1 つのファイルとして配布できます。この方法では、使用可能なすべてのソフトウェアを含めるために単一のメディアは必要ありません。切断状態での操作またはファイアウォールで保護された操作をサポートするために、リポジトリをコピーしてマージするためのツールが用意されています。

メタデータをソフトウェアパッケージの一部として含める。

複数の (場合によっては競合する) ソフトウェアパブリッシャーを有効にしたいという願いは、すべてのパッケージメタデータをパッケージそのものに格納する決定につながりました: すべてのパッケージや依存関係などの情報のためのマスターデータベースは存在しません。ソフトウェアパブリッシャーからの使用可能パッケージのカタログは、パフォーマンスの理由でリポジトリの一部になっていますが、パッケージに含まれるデータからそのカタログを再生成することもできます。

ソフトウェアの自己アセンブリ

上記の目標や考えを踏まえて、IPS はソフトウェアの自己アセンブリという一般概念を取り入れています: システムにインストールされたソフトウェアのどのコレクションも、パッケージ化の操作が完了するときまで、またはソフトウェアの実行時に、そのシステムがブートすると、コレクション自身を作業用構成に組み込むことができます。

ソフトウェアの自己アセンブリでは、IPS でのインストール時のスクリプト作成が必要ありません。ソフトウェアは自身の構成に責任を持ち、パッケージシステムがソフトウェアの代わりにその構成を実行するのに頼ることはありません。ソフトウェアの自己アセンブリを使用すると、パッケージシステムは、現在ブートしていないブート環境や、オフラインのゾーンルートなどの代替イメージでも安全に動作できます。また、自己アセンブリは実行時イメージに対してのみ実行されるため、パッケージ開発者はバージョン間またはアーキテクチャー間の実行時コンテキストに対処する必要はありません。

ブート前にオペレーティングシステムイメージの準備が多少必要です。そうすれば、IPS はこれを透過的に管理できます。イメージの準備には、ブートブロックの更新、ブートアーカイブ (`ramdisk`) の準備、および一部のアーキテクチャーではブート環境選択のメニューの管理などがあります。

ソフトウェアの自己アセンブリ用のツール

次の IPS 機能および特性により、ソフトウェアの自己アセンブリが容易になります。

不可分なソフトウェアオブジェクト

アクションとは IPS でのソフトウェア配布の不可分単位です。各アクションは 1 つのソフトウェアオブジェクトを配布します。そのソフトウェアオブジェクトは、ファイル、ディレクトリ、リンクなどのファイルシステムオブジェクトにも、ユーザー、グループ、デバイスドライバなどのより複雑なソフトウェア構成にもなり得ます。SVR4 パッケージシステムでは、これらのより複雑なアクションタイプは、クラスアクションスクリプトを使用して処理されます。IPS では、スクリプト作成は必要ありません。

アクションは、いくつかのパッケージにまとめられ、ライブイメージとオフラインイメージの両方からインストール、更新、および削除できます。ライブイメージは、現在のゾーンのアクティブな実行中のブート環境の `/` にマウントされたイメージです。

アクションについては、25 ページの「パッケージの内容: アクション」に詳しく説明されています。

構成の合成

IPS では、パッケージ化の操作中にスクリプトを使用して構成ファイルを更新するのではなく、構成ファイルのフラグメントを配布することを推奨しています。このフラグメントは、次の方法で使用できます。

- ファイルフラグメントを認識するようにパッケージ化されたアプリケーションを作成できます。アプリケーションは、その構成の読み取り時に構成ファイルフラグメ

ントに直接アクセスするか、あるいは、フラグメントを完全な構成ファイルにまとめてから読み取ることができます。

- 構成ファイルのフラグメントのインストール、削除、または更新が行われるたびに、SMF サービスは構成ファイルをまとめなおすことができます。

構成ファイルをまとめるサービスを配布するパッケージの作成例については、95 ページの「フラグメントファイルからのカスタムファイルの作成」を参照してください。

アクチュエータと SMF サービス

アクチュエータとは、パッケージシステムによって配布されたアクションに適用されるタグのことで、そのアクションがインストール、削除、または更新されたときにシステム変更をもたらします。これらの変更は通常、SMF サービスとして実装されます。アクチュエータの詳細については、第7章「パッケージインストールの一環としてのシステム変更の自動化」を参照してください。

SMF サービスはソフトウェアを直接構成するか、SMF マニフェストに配布されたデータまたはシステムにインストールされたファイルのデータを使用すると、構成ファイルを作成できます。

SMF には、依存関係を表現するための多機能な構文があります。各サービスが実行されるのは、その必要な依存関係がすべて満たされている場合だけです。SMF サービスの詳細については、『Oracle Solaris 11.3 でのシステムサービスの管理』を参照してください。

どのサービスも、それ自身を `svc:/milestone/self-assembly-complete:default` SMF マイルストーンへの依存関係として追加できます。ブートオペレーティングシステムがこのマイルストーンに到達すると、自己アセンブリのすべての操作が行われます。

不変ゾーンと呼ばれる特別な種類のゾーンは、そのファイルシステムの一部に対する制限付き書き込みアクセス権を持つように構成できるゾーンです。`zonecfg(1M)` のマニュアルページの `file-mac-profile` の説明を参照してください。このタイプのゾーンで自己アセンブリを完了するには、『Oracle Solaris ゾーンの作成と使用』 の 「書き込み可能なルートファイルシステムを持つ読み取り専用ゾーンをブートするオプション」の説明に従い、ゾーン読み取り/書き込みをブートします。`self-assembly-complete` SMF マイルストーンがオンラインになると、必要な `file-mac-profile` 設定に基づいてゾーンが自動的にブートされます。

Oracle Solaris でのソフトウェア自己アセンブリの例

次の例では、Oracle Solaris の一部として配布されるパッケージについて説明します。

Apache Web Server 構成

Apache Web Server 向け Oracle Solaris パッケージ `pkg:/web/server/apache-22` が配布する `httpd.conf` ファイルには、`/etc/apache2/2.2/conf.d` ディレクトリ内の構成ファイルを参照する次の `Include` 指令が含まれています。

```
Include /etc/apache2/2.2/conf.d/*.conf
```

カスタム構成を適用するには、カスタム `.conf` ファイルをその `conf.d` ディレクトリに配布し、新しい `.conf` ファイルを配布するパッケージがインストール、更新、または削除されるたびに `refresh_fmri` アクチュエータを使用して Apache インスタンスを自動的にリフレッシュするパッケージを 1 つまたは複数作成できます。

```
file etc/apache2/2.2/conf.d/custom.conf path=etc/apache2/2.2/conf.d/custom.conf \
    owner=root group=bin mode=0644 refresh_fmri=svc:/network/http:apache22
```

`refresh_fmri` アクチュエータの使用方法については、51 ページの「必要とされるファセットまたはアクチュエータがあれば追加する」および 第7章「パッケージインストールの一環としてのシステム変更の自動化」を参照してください。

Apache サービスインスタンスをリフレッシュすると、Web サーバーによりその構成が再構築されます。これを検証するには、次のコマンドを使用して、Apache サービスインスタンスのリフレッシュ時に実行されるメソッドの名前を表示します。

```
$ svcprop -p refresh/exec http:apache22
/lib/svc/method/http-apache22\ refresh
```

メソッドを確認すると、`http:apache22` インスタンスのリフレッシュで、`graceful` コマンドにより `apachectl` が呼び出され、Apache `httpd` デーモンが再起動されていることが示されています。

ユーザー属性の構成

ユーザー属性は `/etc/user_attr` と、`/etc/user_attr.d` 内の追加の構成ファイルで構成されます。

`/etc/user_attr` 構成ファイルは、システム上の役割とユーザーに対して拡張属性を構成するために使用します。Oracle Solaris 11 では、`/etc/user_attr` ファイルはローカルの変更にだけ使用されます。完全な構成は、`/etc/user_attr.d` ディレクトリに配布された別個のファイルから読み取られます。完全な構成のフラグメントが、複数のパッケージによって配布されます。たとえば `/etc/user_attr.d/core-os` は `system/core-os` パッケージによって配布され、`/etc/user_attr.d/ikev2-daemon` は `system/network/ike` パッケージによって配布されます。

これらの構成ファイルをインストールしても、その結果、サービスが再起動またはリフレッシュされることはありません。これらのファイルをインストール、アンインストール、または更新するときにスクリプト処理は不要です。`/etc/user_attr.d` 内のファイルは、ネームサービスキャッシュデーモン `nscd` によって作成されます。`nscd`

デーモンの動作は svc:/system/name-service/cache サービスにより管理されます。

```
$ svcs -p cache
STATE          STIME    FMRI
online         15:54:24 svc:/system/name-service/cache:default
               15:54:24    100698 nscd
```

ネームサービスキャッシュデーモンは、user_attr の説明と同様の方法でほとんどのネームサービス要求に対し構成の合成を提供します。nscd(1M) のマニュアルページを参照してください。

セキュリティー構成

/etc/security/exec_attr.d/ ディレクトリにはセキュリティー構成ファイルが格納されます。

以前の Oracle Solaris リリースでは、SMF サービスは exec_attr.d に配布されたファイルをマージして 1 つのデータベース /etc/security/exec_attr にしていました。Oracle Solaris 11 では、セキュリティー属性データベースライブラリ libsecdb の関数が、exec_attr.d のフラグメントを直接読み取るため、サービスによりマージを実行する必要がありません。

/etc/security 内で構成フラグメントが含まれているその他のディレクトリ (auth_attr.d や prof_attr.d など) も同様に処理されます。

IPS パッケージのライフサイクル

このセクションでは、IPS パッケージのライフサイクル内の各状態についての概要を説明します。最良の結果を得るために、パッケージ開発者とシステム管理者の両者がパッケージライフサイクルのさまざまな局面について理解しておく必要があります。

作成

だれでもパッケージを作成できます。IPS では、パッケージ作成者に、特定のソフトウェア構築システムもディレクトリ階層も課しません。パッケージ作成の詳細は、第2章「IPS を使用したソフトウェアのパッケージ化」を参照してください。パッケージ作成の各側面については、このガイドの残りの章を通して説明します。

発行

パッケージが IPS リポジトリ (HTTP の場所とファイルシステムのどちらか) に発行されます。発行されたパッケージは .p5p パッケージアーカイブファイルに変換することもできます。IPS リポジトリからソフトウェアにアクセスするには、pkg set-publisher コマンドを使用してリポジトリをシステムに追加することも、pkg コマンドで -g オプションを使用して一時ソースとし

てリポジトリにアクセスすることもできます。パッケージの発行の例は、第2章「IPS を使用したソフトウェアのパッケージ化」に示されています。

インストール

`http://`、`https://`、または `file://` URL 経由でアクセスされる IPS リポジトリから、あるいは `.p5p` パッケージアーカイブからパッケージをシステムにインストールできます。パッケージのインストールについては、第3章「ソフトウェアパッケージのインストール、削除、および更新」で詳しく説明しています。

更新

更新されたバージョンのパッケージが使用可能になり、IPS リポジトリに発行されるか、新しい `.p5p` パッケージアーカイブとして配布される可能性があります。その後、インストールされたパッケージは個別に、またはシステム全体の更新の一部として最新の状態にすることができます。

IPS は SVR4 パッケージシステムが行なっていた「パッチ」という概念を使用しないことに注意してください。IPS パッケージ化されたソフトウェアへの変更はすべて、更新されたパッケージによって配布されます。

パッケージの更新はパッケージのインストールとほぼ同じように行われますが、パッケージシステムは更新されたパッケージによって配布された変更部分のみをインストールするように最適化されています。パッケージの更新については、第3章「ソフトウェアパッケージのインストール、削除、および更新」で詳しく説明しています。

名前の変更

パッケージの有効期間中はパッケージの名前を変更できます。パッケージの名前は、組織上の理由のため、またはパッケージをリファクタリングするために変更されることがあります。パッケージのリファクタリングの例として、いくつかのパッケージを 1 つのパッケージに統合したり、1 つのパッケージを複数の小さなパッケージに分けたりすることがあげられます。

IPS はパッケージ間を移動する内容を正常に処理します。IPS では、古いパッケージ名がシステム上に存続することを許可しており、ユーザーが名前変更されたパッケージをインストールするよう求めると、自動的に新しいパッケージをインストールします。パッケージの名前変更については、101 ページの「パッケージの名前変更、マージ、および分割」で詳しく説明しています。

廃止

最終的に、パッケージはその有効期間の終わりに達する可能性があります。パッケージのパブリッシャーでは、パッケージがサポートされなくなり、入手可能な更新がなくなると判断することがあります。IPS では、パブリッシャーがそのようなパッケージに廃止のマークを付けることを許可しています。

廃止されたパッケージはほとんどの依存関係でターゲットとして他のパッケージから使用できなくなり、廃止バージョンにアップグレードされたパッケージはすべてシステムから自動的に削除されます。パッケージの廃止については、101 ページの「パッケージの名前変更、マージ、および分割」に詳しく説明されています。

削除

最後に、削除するパッケージへの依存関係を持つパッケージがほかにない場合は、そのパッケージをシステムから削除できます。パッケージの削除については、第3章「ソフトウェアパッケージのインストール、削除、および更新」で詳しく説明しています。

IPS 用語およびコンポーネント

このセクションでは、IPS 用語を定義し、IPS コンポーネントについて説明します。

インストール可能なイメージ

IPS はパッケージをイメージにインストールするように設計されています。イメージとはディレクトリツリーであり、必要に応じてさまざまな場所にマウントできます。イメージは、次の 3 つのタイプのいずれかです。

フル

フルイメージでは、イメージ自体の中ですべての依存関係が解決され、IPS が一貫した方法で依存関係を維持します。

ゾーン

非大域ゾーンイメージは、フルイメージ (親の大域ゾーンイメージ) にリンクされていますが、それらだけでは完全なシステムを提供しません。ゾーンイメージでは、IPS は、パッケージ内の依存関係によって定義されているとおりに、非大域ゾーンと大域ゾーンの一貫性を維持します。

ユーザー

ユーザーイメージは、再配置可能なパッケージのみを含みます。

イメージは、インストーラ、`beadm` および `zonecfg` コマンド、および `pkg` コマンドに `--be-name` などのオプションを指定することによって作成またはクローンされます。

パッケージ識別子: FMRI

それぞれのパッケージは障害管理リソース識別子 (FMRI) によって表されます。パッケージの完全な FMRI は、次の形式のスキーム、パブリッシャー、パッケージ名、およびバージョン文字列で構成されます。

scheme://*publisher*/*name*@*version*

すべての IPS パッケージ FMRI のスキームは pkg です。suri ストレージライブラリの次のサンプルパッケージ FMRI では、solaris がパブリッシャー、system/library/storage/suri がパッケージ名、0.5.11,5.11-0.175.3.0.0.19.0:20150329T164922Z がバージョンです。

```
pkg://solaris/system/library/storage/suri@0.5.11,5.11-0.175.3.0.0.19.0:20150329T164922Z
```

FMRI は、結果となる FMRI が引き続き一意である場合は、省略形で指定することができます。スキーム、パブリッシャー、およびバージョンは省略できます。パッケージ名の先頭のコンポーネントは省略できます。

- FMRI が pkg:// または // で始まる場合、// のあとの最初の語はパブリッシャー名である必要があり、パッケージ名からコンポーネントを省略することはできません。パッケージ名からコンポーネントを省略しない場合、そのパッケージ名は完全、またはルート指定とみなされます。
- FMRI が pkg:/ または / で始まる場合、そのスラッシュのあとの最初の語はパッケージ名であり、パッケージ名からコンポーネントを省略することはできません。パブリッシャー名はなくても構いません。
- バージョンが省略されている場合、パッケージは通常、インストール可能な最新バージョンのパッケージに解決されます。

パッケージパブリッシャー

パブリッシャーとは、パッケージを開発および構築するエンティティーの 1 つです。パブリッシャー名 (接頭辞) は、独自の方法でこのソースを識別します。パブリッシャー名には、大文字、小文字、数字、ハイフン、およびピリオド、つまり有効なホスト名と同じ文字を含めることができます。インターネットのドメイン名または登録商標は、自然の名前空間のパーティション分割を提供するので、パブリッシャー名に適しています。

pkg クライアントは、パッケージングソリューションの算出時に指定のパブリッシャーの指定されたすべてのパッケージソースを結合します。

パッケージ名

パッケージ名は階層構造になっていて、任意の数のコンポーネントがスラッシュ (/) 文字で区切られています。パッケージ名のコンポーネントは、文字または数字で始める必要があり、アンダースコア (_)、ハイフン (-)、ピリオド (.)、およびプラス記号 (+) を含めることができます。パッケージ名のコンポーネントは大文字と小文字が区別されます。

パッケージ名の先頭のコンポーネントは、使用するパッケージ名が一意であれば、省略できます。たとえば、/driver/network/ethernet/e1000g は network/ethernet/e1000g、ethernet/e1000g 、または単なる e1000g に縮小できます。FMRI は、

アスタリスク (*) を使ってパッケージ名の一部を一致させることで指定することもできます。したがって、`/driver/*/e1000g` および `/dri*00g` はどちらも `/driver/network/ethernet/e1000g` に展開されます。

パッケージ名を選択するときは、できるだけあいまいさを減らしてください。パッケージ名は、パブリッシャーの間で単一の名前空間を形成します。名前とバージョンが同じでパブリッシャーが異なるパッケージは、外部依存関係や外部インタフェースの観点から取り替え可能とみなされます。パッケージ名と依存関係については、99 ページの「パッケージ内容の競合の回避」を参照してください。

パッケージ名が `pkg:/`、`/`、`pkg://`*publisher*`/`、または `//`*publisher*`/` で始まる場合、パッケージ名は完全とみなされるか、またはルート指定とみなされます。`pkg` クライアントがあいまいなパッケージ名に関してエラーを出す場合は、指定するパッケージ名のコンポーネントを増やすか、完全なルート指定の名前を指定します。

FMRI にパブリッシャー名が含まれている場合は、完全なルート指定のパッケージ名を指定する必要があります。

スクリプトでは、パッケージを参照するときにパブリッシャーを省略できますが、その完全なルート指定の名前を参照するべきです。

パッケージバージョン

パッケージバージョンには、次の 4 つの部分があります。

component_version,*release*-*branch_version*:*time_stamp*

コンポーネントバージョン、リリース、およびブランチバージョンは任意の長さにできますが、整数とピリオド文字 (.) だけで構成されている必要があります。複数の整数の並びを 0 で始めることはできません。製品バージョンを IPS パッケージバージョンに変換するためのヘルプについては、48 ページの「適切なパッケージバージョン文字列の作成」を参照してください。

タイムスタンプには次の部分があります。日付と時間は、整数だけで構成されている必要があります。

*date*T*time*Z

コンポーネントバージョンとリリースは、コンマ (,) で区切ります。リリースとブランチバージョンは、ハイフン (-) で区切ります。ブランチバージョンとタイムスタンプは、コロン (:) で区切ります。

次のサンプルパッケージバージョンについて次に説明します。

```
0.5.11,5.11-0.175.3.0.0.19.0:20150329T164922Z
```

コンポーネントバージョン: `0.5.11`

オペレーティングシステムに緊密に結合されたコンポーネントの場合、コンポーネントバージョンは通常、そのバージョンのオペレーティングシステムでの

`uname -r` の値を含みます。独自の開発ライフサイクルを持つコンポーネントの場合、コンポーネントバージョンはドットで区切られたリリース番号 (2.4.10 など) です。

リリース: `5.11`

リリース (存在する場合) は、コンマ (,) のあとに続ける必要があります。Oracle Solaris では、この並びを使用して、そのパッケージがコンパイルされた OS のリリースを指定します。

ブランチバージョン: `0.175.3.0.0.19.0`

ブランチバージョン (存在する場合) は、ハイフン (-) のあとに続ける必要があります。ブランチバージョンはベンダー固有の情報を提供します。この並びにはビルド番号を含めることも、他のなんらかの情報を指定することもできます。この値は、コンポーネントとは無関係に、パッケージメタデータが変更されたときに増分できます。Oracle Solaris でのブランチバージョンフィールドの使用方法については、145 ページの「Oracle Solaris パッケージのバージョン管理」を参照してください。

タイムスタンプ: `20150329T164922Z`

タイムスタンプ (存在する場合) は、コロン (:) のあとに続ける必要があります。タイムスタンプは、ISO-8601 基本形式 (*YYYYMMDD*T*HHMMSS*Z) のパッケージが発行された日時です。タイムスタンプは、パッケージが発行されるときに自動的に更新されます。

パッケージバージョンは、左から右の順で並べられます: `@` の直後の数字がバージョン領域で最上位の部分です。タイムスタンプは、バージョン領域で最下位の部分です。

`pkg.human-version` 属性を使用すると、人間が読める文字列を提供できます。前述のパッケージ FMRI のパッケージバージョンに加えて `pkg.human-version` 属性の値を指定できますが、この値によってパッケージ FMRI バージョンを置き換えることはできません。人間が読める形式のバージョン文字列は、表示のためにのみ使用されます。詳細は、33 ページの「設定アクション」を参照してください。

任意の長さのバージョンを許可することで、IPS はソフトウェアをサポートするためのさまざまなモデルに対応できます。たとえば、パッケージの作成者はビルドまたはブランチバージョンを使用して、バージョン管理スキームの一部分をセキュリティー更新に割り当て、別の部分を有償または無償のサポート更新に割り当て、さらに別の部分をマイナーバグ修正に割り当てるなど、必要な情報をすべて割り当てることができます。

バージョンをトークン `latest` にして、既知の最新バージョンを指定することもできます。

Oracle Solaris によるバージョン管理の実装方法については、付録B IPS を使用して Oracle Solaris OS をパッケージ化する方法で説明しています。

パッケージの内容: アクション

アクションはパッケージを構成するソフトウェアを定義します。それらはこのソフトウェアコンポーネントの作成に必要なデータを定義します。パッケージの内容は、パッケージマニフェストファイル内に 1 組のアクションとして表されます。

パッケージマニフェストは、大部分がプログラムを使って作成されます。パッケージ開発者はインストールされるオブジェクトについての必要な情報を指定し、第2章「IPS を使用したソフトウェアのパッケージ化」に説明されているように、パッケージ開発ツールを使ってマニフェストを完成させます。

アクションは、パッケージマニフェストファイル内に次の形式で表されます。

action_name attribute1=value1 attribute2=value2 ...

次のアクションの例では、`dir` はこのアクションがディレクトリを指定していることを示しています。*name=value* という形式の属性は、そのディレクトリのプロパティーを記述しています。

```
dir path=a/b/c group=sys mode=0755 owner=root
```

アクションメタデータは自由に拡張できます。必要に応じて、さらなる属性をアクションに追加できます。属性名には、空白、引用符、または等号 (=) を含めることはできません。属性値にはそれらをすべて指定できますが、空白を含む値は一重または二重引用符で囲む必要があります。二重引用符で囲まれた文字列の内側では一重引用符をエスケープする必要はなく、一重引用符で囲まれた文字列の内側では二重引用符をエスケープする必要はありません。引用符で囲まれた文字列が終わるのを防ぐために、引用符には接頭辞としてバックスラッシュ文字 (\) を付けることができます。バックスラッシュは、バックスラッシュでエスケープできます。カスタムの属性名には、意図しない名前空間のオーバーラップを防ぐために一意の接頭辞を使用するようにしてください。22 ページの「パッケージパブリッシャー」のパブリッシャー名の説明を参照してください。

アクションは複数の属性を含むことができます。一部の属性は、単一のアクションに対して異なる値を使用して、複数回名前を付けることができます。同じ名前を持つ複数の属性は、順序付けされていないリストとして扱われます。

多くの属性を持つアクションによって、マニフェストファイル内に長い行が作成されることがあります。このような行は、不完全な各行をバックスラッシュ文字で終了することにより折り返すことができます。この継続文字は、属性と値のペアの間に置く必要があることに注意してください。属性も、その値も、さらにその組み合わせも分割することはできません。

一部の属性によって、パッケージングコンテキストの外部で追加の操作が実行されます。詳細は、第7章「パッケージインストールの一環としてのシステム変更の自動化」を参照してください。

ほとんどのアクションにはキー属性があります。キー属性とは、このアクションをイメージ内のほかのすべてアクションとは異なる一意のものにする属性です。ファイルシステムオブジェクトの場合、キー属性はそのオブジェクトのパスになります。

パスにインストールされるアクションは、次のすべてのパスに内容を提供しないでください。

- `/system/volatile`
- `/tmp`
- `/var/pkg`
- `/var/share`
- `/var/tmp`

以降のセクションでは、各 IPS アクションタイプと、それらのアクションを定義する属性について説明します。アクションタイプについては、pkg(5) のマニュアルページに詳しく説明されていますが、参考のためにここで繰り返し説明します。各セクションには、パッケージの作成中にパッケージマニフェスト内に見られるようなアクション例が含まれています。ほかの属性は、発行中に自動的にアクションに追加される可能性があります。

ファイルアクション

`file` アクションは間違いなくもっとも一般的なアクションです。`file` アクションは通常のファイルを表します。`file` アクションはペイロードを参照し、次の 4 つの標準属性があります。

`path` ファイルがインストールされているファイルシステムのパス。これは `file` アクションのキー属性です。`path` 属性の値は、イメージのルートを基準としています。先頭に `/` を含めないでください。

`mode` ファイルのアクセス権。`mode` 属性の値は、ACL ではなく、数値形式による単純なアクセス権です。

`owner` ファイルを所有するユーザーの名前。

`group` ファイルを所有するグループの名前。

ペイロード属性は位置属性です。ペイロード属性はアクション名のあとの最初の語で、通常は属性名はありません。まだ公開されていないマニフェスト内で、ペイロード属性の値はペイロードファイルへのフルパスから先頭のスラッシュ文字 (/) を除いたものです。ペイロード値に等号 (=) が含まれている場合、ペイロード属性値の前に `hash=` を使用します。位置属性および `hash` ペイロード属性の両方が同じアクションで使用される場合、値は同一である必要があります。

次の例は、45 ページの「パッケージマニフェストを生成する」で示すように `pkgsend generate` コマンドによって出力が表示されることがある `file` アクションです。

```
file opt/mysoftware path=opt/mysoftware group=bin mode=0644 owner=root
```

次の例は等号を含むパスを指定する方法を示しています。

```
file hash=opt/my=software path=opt/my=software group=root mode=0644 owner=root
```

公開されたマニフェストでは、ペイロード属性の値はファイルの内容のハッシュで、これはパッケージシステムによって使用されます。詳細は、pkgsend(1) のマニュアルページを参照してください。

`preserve` 属性および `overlay` 属性は、`file` アクションがインストールされるかどうか、およびその方法に影響を及ぼします。

`preserve`

パッケージ操作中にファイルを保持する時期と方法を指定します。

パッケージが最初にインストールされるとき、パッケージによって提供されるファイルが持つ `preserve` 属性が、`abandon` または `install-only` 以外の何らかの値で定義されており、ファイルがイメージ内にすでに存在する場合、既存のファイルが `/var/pkg/lost+found` に格納され、パッケージされたファイルはインストールされます。

パッケージが最初にインストールされるとき、パッケージによって提供されるファイルの `preserve` 属性が定義済みで、ファイルがイメージ内に存在しない場合、ファイルがインストールされるかどうかは `preserve` 属性の値によって決まります。

- `preserve` の値が `abandon` または `legacy` である場合、パッケージされたファイルはインストールされません。
- `preserve` の値が `abandon` または `legacy` でない場合、パッケージされたファイルはインストールされます。

パッケージがダウングレードされるとき、ダウングレードされるバージョンのパッケージによって提供されるファイルの `preserve` 属性が、`abandon` または `install-only` 以外の何らかの値で定義されており、次のすべての条件が真の場合、イメージ内に現在存在するファイルは拡張子 `.update` で名前変更され、ダウングレードされたパッケージからのファイルがインストールされます。

- ファイルがイメージ内に存在する。
- ダウングレードされるバージョンのパッケージによって提供されるファイルの内容が、現在インストールされているバージョンのパッケージによって提供されるファイルの内容と異なる。
- ダウングレードされるバージョンのパッケージによって提供されるファイルの内容が、イメージ内に存在するファイルの内容と異なる。

上の条件のいずれかが真でない場合、このファイルは、パッケージがダウングレードでなくアップグレードされる場合と同様に処理されます。

パッケージがアップグレードされるとき、アップグレードされるバージョンのパッケージによって提供される `file` アクションの `preserve` 属性が何らかの値で定義されており、`file` アクションが、現在インストールされているバージョンのパッケージによって提供される `file` アクションと同じである場合、ファイルはインストールされず、イメージ内に存在するファイルは変更されません。以前のバージョンをインストールしたあとに加えられたすべての変更が保持されます。

パッケージがアップグレードされるとき、アップグレードされるバージョンのパッケージによって提供される `file` アクションの `preserve` 属性が定義済みで、`file` アクションが新しいか、現在インストールされているバージョンのパッケージによって提供される `file` アクションとは異なる場合、アップグレードは次の方法で実行されます。

- アップグレードされるバージョンのパッケージによって提供されるファイルの `preserve` 値が、アップグレードされるパッケージ内で `abandon` または `install-only` の場合、新しいファイルはインストールされず、既存のファイルは変更されません。
- イメージ内にファイルが存在しない場合は、新しいファイルがインストールされます。
- アップグレードされるバージョンのパッケージによって提供されるファイルがイメージ内に存在し、現在インストール済みのバージョンのパッケージ内に存在せず、`original_name` 属性を使用した名前変更または移動を行わなかった場合、既存のファイルは `/var/pkg/lost+found` に格納され、アップグレードされたバージョンのパッケージによって提供されるファイルがインストールされます。次の `original_name` 属性の説明を参照してください。
- アップグレードされるバージョンのパッケージによって提供されるファイルがイメージ内に存在し、現在インストール済みのバージョンのパッケージによって提供されるファイルと内容が異なる場合、アップグレードは `preserve` 属性の値に従って実行されます。
  - アップグレードされるバージョンのパッケージによって提供されるファイルの `preserve` 値が `renameold` の場合、既存のファイルは拡張子 `.old` を使用して名前変更され、新しいファイルは更新されたアクセス権およびタイムスタンプ (存在する場合) を使用してインストールされます。下の `timestamp` 属性の説明を参照してください。
  - アップグレードされるバージョンのパッケージによって提供されるファイルの `preserve` 値が `renamenew` の場合、新しいファイルは拡張子 `.new` を使用してインストールされ、既存のファイルは変更されません。
  - アップグレードされるバージョンのパッケージによって提供されるファイルの `preserve` 値が `true` の場合、新しいファイルはインストールされませんが、既存のファイルのアクセス権およびタイムスタンプ (存在する場合) がリセットされます。
- アップグレードされるバージョンのパッケージによって提供されるファイルがイメージ内に存在し、現在インストール済みのバージョンのパッケージによって提供されるファイルと同じ内容を持ち、`preserve` 値が `renameold` または `renamenew` の場合、既存のファイルはアップグレードされたバージョンのパッ

ケージによって提供されるファイルによって置き換えられ、アクセス権およびタイムスタンプ (存在する場合) も置き換えられます。

- アップグレードされるバージョンのパッケージによって提供されるファイルがイメージ内に存在し、アップグレードされるパッケージ内の `preserve` 値が `legacy` で、現在インストール済みのバージョンのパッケージとは `preserve` 値が異なる場合、既存のファイルは拡張子 `.legacy` を使用して名前変更され、新しいファイルは、更新されたアクセス権およびタイムスタンプ (存在する場合) を使用してインストールされます。
- パッケージのアップグレードバージョンによって配布されるファイルがイメージ内に存在し、アップグレードパッケージと、現在インストール済みのバージョンのパッケージの両方で `preserve` 値が `legacy` の場合、既存のファイルのアクセス権およびタイムスタンプ (存在する場合) がリセットされます。

パッケージがアンインストールされるとき、現在インストール済みのバージョンのパッケージによって提供される `file` アクションの持つ `preserve` 値が `abandon` または `install-only` で、ファイルがイメージ内に存在する場合、ファイルは削除されません。

`overlay`

このアクションによってほかのパッケージが同じ場所にファイルを提供できるか、またはそれによって別のファイルをオーバーレイすることを目的にしたファイルが提供されるかを指定します。この機能は、どの自己アセンブリにも参加しておらず、かつ安全に上書きできる構成ファイルで使用されることを目的にしています。

`overlay` が指定されていない場合は、複数のパッケージが同じ場所にファイルを提供することはできません。

`overlay` 属性は、次のいずれかの値を持ちます。

`allow`	もう 1 つのパッケージが同じ場所にファイルを配布できるようになります。`preserve` 属性も同時に設定されていないかぎり、この値は意味を持ちません。
`true`	このアクションによって配布されたファイルは、`allow` を指定したほかのすべてのアクションを上書きします。

インストールされたファイルへの変更は、オーバーレイしているファイルの `preserve` 属性の値に基づいて保持されます。削除時、ファイルの内容は、`preserve` 属性が指定されたかどうかには関係なく、オーバーレイされているアクションがまだインストールされている場合は保持されます。別のアクションをオーバーレイできるのは 1 つのアクションだけであり、`mode`、`owner`、および `group` 属性が一致している必要があります。

ELF ファイルの場合、次の属性が認識されます。

`elfarch`	ELF ファイルのアーキテクチャー。この値は、そのファイルが構築されたアーキテクチャー上での `uname -p` の出力です。

`elfbits` この値は 32 または 64 です。

`elfhash` この値は、バイナリがロードされるときにメモリーにマップされるファイル内の ELF セクションのハッシュです。これらは、2 つのバイナリの実行可能ファイルの動作が異なるかどうかを判定するときに考慮する必要のある唯一のセクションです。

`file` アクションの場合、次の追加属性が認識されます。

`original_name`

この属性は、パッケージからパッケージに、場所から場所に、あるいはその両方で移動している編集可能なファイルを処理するために使用されます。この属性の値は、元のパッケージの名前のあとに、コロンが続き、そのあとにそのファイルの元のパスが続きます。削除されているファイルはすべて、そのパッケージとパス、または `original_name` 属性の値 (指定されている場合) のどちらかを使用して記録されます。`original_name` 属性が設定された、インストールされている編集可能なファイルはすべて、それが同じパッケージ化の操作の一部として削除されている場合は、その名前のファイルを使用します。

この属性を一度設定したら、パッケージまたはファイルが繰り返し名前変更されても、その値を変更しないでください。同じ値を保持することで、以前のすべてのバージョンからアップグレードを行うことができます。

`release-note`

この属性は、このファイルにリリースノートテキストが含まれていることを示すために使用されます。この属性の値は、パッケージ FMRI です。元のイメージに存在し、元のイメージ内のパッケージよりも新しいバージョンのパッケージ名が FMRI で指定されている場合、このファイルはリリースノートに含まれます。特別な FMRI (`feature/pkg/self`) は、含まれるパッケージを指します。`feature/pkg/self` のバージョンが 0 の場合、このファイルは初期インストールのリリースノートにのみ含まれます。

`revert-tag`

この属性は、セットとして元に戻すべき編集可能なファイルをタグ付けするために使用されます。`revert-tag` 属性の値は *tagname* です。単一の `file` アクションに対して複数の `revert-tag` 属性を指定できます。これらのいずれかのタグを指定して `pkg revert` が呼び出されると、ファイルはマニフェストで定義された状態に戻ります。`pkg revert` コマンドについては、『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「タグ付けされたファイルおよびディレクトリを元に戻す」および pkg(1) のマニュアルページを参照してください。

`revert-tag` 属性は、ディレクトリレベルでも指定できます。後述の「ディレクトリアクション」を参照してください。

`sysattr`

この属性は、このファイルに設定する必要があるシステム属性を指定するために使用されます。`sysattr` 属性の値は、次の例に示すように、詳細システム属性の

カンマ区切りリストまたはコンパクトシステム属性オプションの文字列シーケンスの場合があります。サポートされるシステム属性は、chmod(1) マニュアルページで説明されています。マニフェストに指定されるシステム属性は、オペレーティングシステムのほかのサブシステムによって設定される場合があるシステム属性に追加で設定されます。

```
file path=opt/secret_file sysattr=hidden,sensitive
file path=opt/secret_file sysattr=HT
```

`timestamp`

この属性は、ファイルに対するアクセスおよび変更時間を設定するために使用します。`timestamp` 属性値は、コロンおよびハイフンを省略した、ISO-8601 形式の UTC で表現する必要があります。

`timestamp` 属性は、Python 用の `.pyc` または `.pyo` ファイルをパッケージングする場合に不可欠です。`.pyc` または `.pyo` ファイルに関連する `.py` ファイルは、次の例で示すように、これらのファイル内に埋め込まれたタイムスタンプを使用してマーク付けする必要があります。

```
file path=usr/lib/python2.6/vendor-packages/pkg/__init__.pyc ...
file path=usr/lib/python2.6/vendor-packages/pkg/__init__.py \
     timestamp=20150331T111615Z ...
```

`file` アクションについての次の属性はシステムで自動的に生成されるため、パッケージ開発者によって指定しないでください。`hash`、`chash`、`pkg.size`、`pkg.csize`、および `pkg.content-hash`。

`file` アクションの例は次のとおりです。

```
file path=usr/bin/pkg owner=root group=bin mode=0755
```

ディレクトリアクション

`dir` アクションは、ファイルシステムオブジェクトを表すという点で `file` アクションに似ています。`dir` アクションは、通常ファイルの代わりにディレクトリを表します。`dir` アクションには、`file` アクションが持つのと同じ `path`、`mode`、`owner`、および `group` 属性があり、`path` がキー属性です。`dir` アクションでも `revert-tag` 属性を受け入れますが、属性の値は `file` アクションと `dir` アクションとで異なります。

ディレクトリは、IPS でカウントされる参照です。あるディレクトリを明示的または暗黙的に参照している最後のパッケージが参照を行わなくなると、そのディレクトリは削除されます。そのディレクトリにパッケージ解除されたファイルシステムオブジェクトが含まれている場合、それらの項目は `$IMAGE_META/lost+found` に移動されます。`$IMAGE_META` の値は通常 `/var/pkg` です。

`revert-tag`

この属性は、セットとして削除する必要があるパッケージ解除されたファイルを識別するために使用します。`file` アクションに対してこの属性を指定する方法に

ついての説明は、上記の「ファイルアクション」を参照してください。ディレクトリの場合、`revert-tag` 属性の値は *tagname=pattern* です。単一の `dir` アクションに対して複数の `revert-tag` 属性を指定できます。マッチングする *tagname* を指定して `pkg revert` が呼び出されると、*pattern* に一致する (シェルグロビング文字を使用)、この `dir` ディレクトリの下にあるパッケージ解除されたファイルまたはディレクトリが削除されます。`pkg revert` コマンドについては、『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「タグ付けされたファイルおよびディレクトリを元に戻す」および `pkg(1)` のマニュアルページを参照してください。

`salvage-from`

この属性は、パッケージ解除された内容を新しいディレクトリに移動する場合に使用できます。この属性の値は、回収された項目のディレクトリの名前です。`salvage-from` 属性を持つディレクトリは、作成時に、`salvage-from` 属性の値に指定されたディレクトリのすべての内容を継承します。

インストール中に、`pkg` は、システム上の指定されたディレクトリアクションのすべてのインスタンスに同じ `owner`、`group`、および `mode` 属性値が含まれていることを確認します。競合する値がシステム上に、または同じ操作でインストールされるほかのパッケージ内に見つかった場合、`dir` アクションはインストールされません。

`dir` アクションの例は次のとおりです。

```
dir path=usr/share/lib owner=root group=sys mode=0755
```

リンクアクション

`link` アクションはシンボリックリンクを表します。`link` アクションには、次の標準属性があります。

`path` シンボリックリンクがインストールされるファイルシステムのパス。これは `link` アクションのキー属性です。

`target` シンボリックリンクのターゲット。リンクの解決先のファイルシステムオブジェクト。

`link` アクションは、特定のソフトウェアの複数のバージョンまたは実装をシステムに同時にインストールできるようにする属性 `mediator`、`mediator-version`、`mediator-implementation`、および `mediator-priority` も取ります。そのようなリンクは仲介リンクであるため、管理者はどのリンクがどのバージョンまたは実装を指すかを必要に応じて簡単に切り替えられます。これらの調停されたリンクの属性については、116 ページの「複数のアプリケーション実装の配布」に詳しく説明されています。調整については、『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「デフォルトのアプリケーション実装の指定」でも説明しています。

`link` アクションの例は次のとおりです。

```
link path=usr/lib/libpython2.6.so target=libpython2.6.so.1.0
```

ハードリンクアクション

`hardlink` アクションは、ハードリンクを表します。`link` アクションと同じ `path` および `target` 属性を持ち、`path` がキー属性です。

`hardlink` アクションの例は次のとおりです。

```
hardlink path=opt/myapplication/hardlink target=foo
```

設定アクション

`set` アクションは、パッケージの説明などの、パッケージレベルの属性 (またはメタデータ) を表します。

次の属性が認識されます。

`name` 属性の名前。

`value` 属性に与えられた値。

`set` アクションは、パッケージ作成者が選択した任意のメタデータを提供できます。次の属性名はパッケージシステムにとって特別な意味を持ちます。

`info.classification`
: `pkg` クライアントがパッケージを分類するために使用できる 1 つ以上のトークン。この値には、スキーム (`org.opensolaris.category.2008` や `org.acm.class.1998` など) と実際の分類 (アプリケーション/ゲームなど) がコロン (`:`) で区切られて含まれています。`info.classification` の一連の値が 付録A パッケージの分類に記載されています。

`pkg.description`
: パッケージの内容と機能の詳細な説明。通常は、1 つの段落程度の長さです。この値は、このパッケージをインストールするとよい理由を説明します。

`pkg.fmri`
: 含まれるパッケージの名前とバージョン。23 ページの「パッケージバージョン」を参照してください。

`pkg.human-version`
: IPS により使用されるバージョンスキームは厳密です。23 ページの「パッケージバージョン」を参照してください。`pkg.human-version` 属性の値として、より柔軟なバージョンを指定できます。`pkg info`、`pkg contents`、および `pkg`

search などの IPS ツールにより値が表示されます。pkg.human-version 値はバージョン比較のベースとして使用されず、pkg.fmri バージョンの代わりに使用することはできません。

pkg.obsolete

true の場合、パッケージは廃止としてマークされています。廃止されたパッケージには、set アクション以外のアクションは存在せず、また名前変更としてマークしてはいけません。パッケージの廃止については、103 ページの「パッケージの廃止」に説明されています。

pkg.renamed

true の場合は、パッケージの名前が変更されました。このパッケージには、このパッケージの名前が変更されたあとのパッケージバージョンを指す 1 つ以上の depend アクションも含まれている必要があります。パッケージを名前変更と廃止の両方としてマークすることはできませんが、それ以外は任意の数の set アクションを含めることができます。パッケージの名前変更については、101 ページの「パッケージの名前変更、マージ、および分割」に説明されています。

pkg.summary

この説明の簡単な概要。この値は、pkg list -s の出力の各行の終わりと pkg info の出力の 1 つの行に表示されます。この値は 60 文字以内にします。この値はパッケージとは何かについて説明し、パッケージの名前またはバージョンを繰り返すことはしません。

その他の情報属性と Oracle Solaris によって使用される属性については、付録B IPS を使用して Oracle Solaris OS をパッケージ化する方法で説明しています。

set アクションの例は次のとおりです。

```
set name=pkg.summary value="Image Packaging System"
```

ドライバアクション

driver アクションはデバイスドライバを表します。driver アクションはペイロードを参照しません。ドライバファイル自体を file アクションとしてインストールする必要があります。次の属性が認識されています。これらの属性の値の詳細については、add_drv(1M) のマニュアルページを参照してください。

name　　ドライバの名前。多くの場合はドライババイナリのファイル名ですが、必ずしもそうとは限りません。これは driver アクションのキー属性です。

alias　　ドライバの別名。特定のドライバが複数の alias 属性を持つことができます。特殊な引用符の規則は必要ありません。

`class` ドライバクラス。特定のドライバが複数の `class` 属性を持つことができます。

`perms` このドライバのデバイスノードのファイルシステムアクセス権。

`clone_perms` このドライバに対するクローンドライバのマイナーノードのファイルシステムアクセス権。

`policy` このデバイスの追加のセキュリティーポリシー。特定のドライバが複数の `policy` 属性を持つことができますが、マイナーデバイスの指定が複数の属性に存在することはできません。

`privs` このドライバで使用される特権。特定のドライバが複数の `privs` 属性を持つことができます。

`devlink` `/etc/devlink.tab` のエントリ。この値はファイルに書き込まれる行そのものであり、タブは `\t` で示されます。詳細は、`devlinks(1M)` のマニュアルページを参照してください。特定のドライバが複数の `devlink` 属性を持つことができます。

driver アクションの例は次のとおりです。

```
driver name=vgatext \
    alias=pciclass,000100 \
    alias=pciclass,030000 \
    alias=pciclass,030001 \
    alias=pnpPNP,900 variant.arch=i386 variant.opensolaris.zone=global
```

依存アクション

`depend` アクションは、パッケージ間の依存関係を表します。あるパッケージが別のパッケージに依存することがあります。これは、最初のパッケージの機能を有効にしたり、インストールしたりするために、2 番目のパッケージの機能が必要であるためです。依存関係はオプションです。インストール時に依存関係が満たされない場合、パッケージシステムはほかの制約に応じて、依存パッケージをインストールするか、または十分に新しいバージョンに更新しようとします。依存関係については、第4章「パッケージ依存関係の指定」で詳しく説明されています。

次の属性が認識されます。

`fmri`

依存関係のターゲットを表す FMRI。これは `depend` アクションのキー属性です。FMRI 値にパブリッシャーを含めてはいけません。パッケージ名は、スラッシュ (/) で始まっていない場合でも、ルート指定とみなされます。タイプ `require-any` の依存関係は、複数の `fmri` 属性を持つことができます。`fmri` 値ではバージョン

はオプションですが、依存関係のタイプによっては、バージョンのない FMRI は意味を持ちません。

FMRI 値にはアスタリスク (*) を使用できず、バージョンに対して `latest` トークンを使用することもできません。

`type`

依存関係のタイプ。

`require`

ターゲットパッケージが必要であり、かつ `fmri` 属性で指定されたバージョン以上のバージョンを持っている必要があります。バージョンが指定されていない場合は、任意のバージョンが依存関係を満たします。`require` 依存関係のいずれかを満たすことができない場合、そのパッケージはインストールできません。

`optional`

依存関係のターゲットが存在する場合は、指定されたバージョンレベル以上である必要があります。

`exclude`

指定されたバージョンレベル以上で依存関係のターゲットが存在する場合は、含んでいるパッケージをインストールできません。バージョンが指定されていない場合、ターゲットパッケージは、依存関係を指定しているパッケージと同時にインストールできません。

`incorporate`

依存関係はオプションですが、ターゲットパッケージのバージョンは制約されます。制約と凍結については、第4章「パッケージ依存関係の指定」を参照してください。

`require-any`

依存関係タイプ `require` と同じ規則に従い、複数の `fmri` 属性で指定された複数のターゲットパッケージのいずれか 1 つが依存関係を満たすことができます。

`conditional`

依存関係のターゲットは、`predicate` 属性で定義されたパッケージがシステム上に存在する場合にのみ必要です。

`origin`

このパッケージのインストールの前に、依存関係のターゲット (存在する場合) は変更されるイメージ上の指定された値以上である必要があります。`root-image` 属性の値が `true` である場合、このパッケージをインストールするには、`/` をルートとするイメージ上にターゲットが存在する必要がありま

す。root-image 属性の値が true で、fmri 属性の値が pkg:/feature/firmware/ で始まる場合、fmri 値の残りの部分は、ファームウェアの依存関係を評価する /usr/lib/fwenum 内のコマンドとして扱われます。

group
: 依存関係のターゲットは、パッケージがイメージ回避リスト上にないかぎり必要です。廃止されたパッケージは、暗黙のうちに group 依存関係を満たすことに注意してください。イメージ回避リストについては、pkg(1) のマニュアルページの avoid サブコマンドを参照してください。

group-any
: 複数の fmri 属性によって指定される複数のターゲットパッケージは依存関係を満たすことができます。廃止されていないパッケージシステムが廃止されたパッケージシステムよりも優先されるという点を除き、group 依存関係に適用されるものと同じ規則が group-any 依存関係に対して適用されます。

parent
: 依存関係は、イメージが子イメージ (ゾーンなど) でない場合は無視されます。このイメージが子イメージである場合は、親イメージ内に依存関係のターゲットが存在する必要があります。parent 依存関係でのバージョンの照合は、incorporate 依存関係で使用されるものと同じです。

predicate
: conditional 依存関係の述語を表す FMRI。

root-image
: 先に説明した origin 依存関係に対してのみ有効です。

depend アクションの例は次のとおりです。

```
depend fmri=crypto/ca-certificates type=require
```

ライセンスアクション

license アクションは、パッケージの内容に関連したライセンスやその他の情報ファイルを表します。パッケージは license アクションを通して、ライセンス、免責条項、またはその他のガイダンスをパッケージインストーラに配布できます。

license アクションのペイロードは、パッケージに関連したイメージメタデータディレクトリに提供され、人間が読める形式のテキストデータのみが含まれているべきです。license アクションのペイロードには、HTML やその他の形式のマークアップが含まれていてはいけません。license アクションは、属性を通して、関連するペイロードが表示または同意を必要としていることを pkg クライアントに示すことができます。表示または同意の方法は、pkg クライアントに任されています。

次の属性が認識されます。

`license`

ユーザーがライセンスのテキスト自体を読まなくてもその内容を判断できるように支援するための、ライセンスのわかりやすい説明を提供します。これは `license` アクションのキー属性です。

この値のいくつかの例を次に示します。

- ABC Co. Copyright Notice
- ABC Co. Custom License
- Common Development and Distribution License 1.0 (CDDL)
- GNU General Public License 2.0 (GPL)
- GNU General Public License 2.0 (GPL) Only
- MIT License
- Mozilla Public License 1.1 (MPL)
- Simplified BSD License

上に示すように、可能な場合は常にライセンスのバージョンを説明に含めることをお勧めします。`license` 値は、パッケージ内で一意である必要があります。

`must-accept`

`true` の場合は、関連するパッケージをインストールまたは更新するには、ユーザーがこのライセンスに同意する必要があります。この属性を省略すると、`false` と同等になります。同意の方法 (たとえば、対話型または構成ベース) は、`pkg` クライアントに任されています。パッケージの更新については、ライセンスアクションまたはペイロードが変更されていない場合、この属性は無視されます。

`must-display`

`true` の場合は、パッケージング操作中に、`pkg` クライアントが `license` アクションのペイロードを表示する必要があります。この属性を省略すると、`false` と同等になります。この属性は、コピーライト表示に使用しないでください。この属性は、操作中に表示する必要のある実際のライセンスまたはその他の素材にのみ使用してください。表示の方法は、`pkg` クライアントに任されています。パッケージの更新については、ライセンスアクションまたはペイロードが変更されていない場合、この属性は無視されます。

`license` アクションの例は次のとおりです。

```
license license="Apache v2.0"
```

レガシーアクション

`legacy` アクションは、レガシー SVR4 パッケージシステムで使用されるパッケージデータを表します。`legacy` アクションに関連付けられた属性は、レガシー SVR4 パッ

ケージシステムのデータベースに追加されるため、これらのデータベースに問い合わせを行なっているツールは、レガシーパッケージが実際にインストールされているかのように動作できます。特に、legacy アクションを指定することにより、pkg 属性で指定されたパッケージが SVR4 の依存関係を満たすことができます。

次の属性が認識されています。関連するパラメータについては、pkginfo(4) のマニュアルページを参照してください。

category	CATEGORY パラメータの値。デフォルト値は system です。
desc	DESC パラメータの値。
hotline	HOTLINE パラメータの値。
name	NAME パラメータの値。デフォルト値は none provided です。
pkg	インストールされるパッケージの略語。デフォルト値は、パッケージの FMRI の名前です。これは legacy アクションのキー属性です。
vendor	VENDOR パラメータの値。
version	VERSION パラメータの値。デフォルト値は、パッケージの FMRI のバージョンです。

legacy アクションの例は次のとおりです。

```
legacy pkg=SUNWcsu arch=i386 category=system \
    desc="core software for a specific instruction-set architecture" \
    hotline="Please contact your local service provider" \
    name="Core Solaris, (Usr)" vendor="Oracle Corporation" \
    version=11.11,REV=2009.11.11 variant.arch=i386
```

署名アクション

署名アクションは、IPS でのパッケージ署名のサポートの一部として使用されます。署名アクションについては、第9章「IPS パッケージの署名」で詳しく説明しています。

ユーザーアクション

user アクションは、/etc/passwd、/etc/shadow、/etc/group、および /etc/ftpd/ftpusers ファイルで指定されているように UNIX ユーザーを定義します。user アクションからの情報は適切なファイルに追加されます。

user アクションは、デーモンまたは使用するほかのソフトウェアのためにユーザーを定義することを目的としています。管理または対話アカウントを定義する目的で user アクションを使用しないでください。

次の属性が認識されます。

username	ユーザーの一意の名前。
password	ユーザーの暗号化パスワード。デフォルト値は *LK* です。
uid	ユーザーの一意の数値 ID。デフォルト値は、100 未満の最初に空いている値です。
group	ユーザーのプライマリグループの名前。この名前は /etc/group に存在する必要があります。
gcos-field	/etc/passwd 内の GECOS フィールドに表される、ユーザーの実名。デフォルト値は username 属性の値です。
home-dir	ユーザーのホームディレクトリ。このディレクトリはシステムイメージディレクトリ内にある必要があり、/home などの別のマウントポイントの下であってはいけません。デフォルト値は / です。
login-shell	ユーザーのデフォルトのシェル。デフォルト値は空です。
group-list	ユーザーが属しているセカンダリグループ。group(4) のマニュアルページを参照してください。
ftpuser	true または false に設定できます。true のデフォルト値は、ユーザーが FTP 経由のログインを許可されていることを示します。ftpusers(4) のマニュアルページを参照してください。
lastchg	1970 年 1 月 1 日と、パスワードが最後に変更された日付の間の日数。デフォルト値は空です。
min	パスワード変更の間の必要な最小日数。パスワードの有効期限を有効にするには、このフィールドを 0 以上に設定する必要があります。デフォルト値は空です。
max	パスワードが有効な最大日数。デフォルト値は空です。shadow(4) のマニュアルページを参照してください。
warn	password の期限が切れる前にユーザーに警告が表示される日数。

`inactive` ユーザーに許可されている非活動の日数。これはシステムごとにカウントされます。最終ログインに関する情報は、システムの `lastlog` ファイルから取得されます。

`expire` UNIX エポック (1970 年 1 月 1 日) 以降の日数として表される絶対的な日付。この日数に達すると、ログインを使用できなくなります。たとえば、13514 の期限切れの値は、ログインの有効期限が 2007 年 1 月 1 日であることを指定します。

`flag` 空に設定されています。

`user` アクションの例は次のとおりです。

```
user ftpuser=false gcos-field="AI User" group=aiuser uid=61 username=aiuser
```

グループアクション

`group` アクションは、`group(4)` ファイルで指定されているように UNIX グループを定義します。グループパスワードのサポートは提供されません。`group` アクションで定義されたグループには最初、ユーザーリストがありません。ユーザーは、`user` アクションを使用して追加できます。

次の属性が認識されます。

`groupname` グループの名前の値。

`gid` グループの一意の数値 ID。デフォルト値は、100 未満の最初に空いているグループです。

`group` アクションの例は次のとおりです。

```
group gid=61 groupname=aiuser
```

パッケージリポジトリ

ソフトウェアリポジトリには、1 つ以上のパブリッシャーのパッケージが含まれています。リポジトリをさまざまな方法でアクセスできるように構成できます: HTTP、HTTPS、ファイル (ローカルストレージ上、あるいは NFS または SMB 経由)、および自己完結型パッケージアーカイブファイル (通常は `.p5p` 拡張子を持つ) として。

パッケージアーカイブにより、IPS パッケージを容易に配布できるようになります。詳細は、58 ページの「パッケージアーカイブファイルとしての配布」を参照してください。

HTTP または HTTPS 経由でアクセスされるリポジトリは、`pkg/server` SMF サービスと `pkg.depotd` プロセスにより管理され、場合によっては `pkg/depot` SMF サービスによっても管理されます。`pkg/depot` サービスは、`package/pkg/depot` パッケージによって提供されます。ファイルリポジトリの場合、リポジトリソフトウェアはアクセスする `pkg` クライアントの一部として実行されます。

例については、53 ページの「パッケージを発行する」および57 ページの「パッケージを配布する」を参照してください。IPS パッケージリポジトリの作成、アクセス、更新、および構成については、『Oracle Solaris 11.3 パッケージリポジトリのコピーと作成』を参照してください。

♦ ♦ ♦ 第 2 章

2 IPS を使用したソフトウェアのパッケージ化

この章では、次を含む、独自のパッケージを構築するところから始めます。

- 新しいパッケージの設計、作成、および発行
- SVR4 パッケージから IPS パッケージへの変換

パッケージの設計

このセクションで説明している優れたパッケージ開発の基準の多くでは、妥協点をさぐることが必要になります。すべての要件を同じように満たすのは難しいことが多々あります。次の基準は、重要度の高い順に示されています。ただし、この順番は状況に応じてゆるやかな目安となるように意図されています。これらの各基準は重要ですが、適切なパッケージセットを作成するために、これらの要件を最大限に生かせるかどうかはユーザー次第です。

パッケージ名を選択します。

Oracle Solaris では、IPS パッケージに階層的な命名方法を使用します。可能なかぎり、同じスキームに適合するようにパッケージ名を決定します。ユーザーが `pkg install` などのコマンドに短いパッケージ名を指定できるように、パッケージ名の最後の部分は一意になるようにします。

クライアントサーバー構成用に最適化します。

パッケージを配置するときに各種パターンのソフトウェア使用 (クライアントとサーバー) を検討します。適切なパッケージ設計では、影響を受けるファイルを分割して各構成タイプのインストールを最適化します。たとえば、ネットワークプロトコル実装では、パッケージユーザーがサーバーを必ずしもインストールしなくてもクライアントをインストールできるようにします。クライアントとサーバーが実装コンポーネントを共有する場合は、共有ビットを含む基本パッケージを作成します。

機能境界に従ってパッケージ化します。

パッケージは自己完結型にし、一連の機能で明確に識別されるようにします。たとえば、ZFS を含むパッケージはすべての ZFS ユーティリティーを含み、ZFS バイナリのみに限定されるようにします。

パッケージは、ユーザーの視点から機能ユニットにまとめるようにします。

ライセンスまたは使用料の境界に従ってパッケージ化します。

契約上の合意により使用料の支払いを必要とするコードや、明確なソフトウェアライセンス条項を含むコードは、専用パッケージまたはパッケージのグループに入れます。コードを必要以上に多くのパッケージに分散しないでください。

パッケージ間のオーバーラップを回避または管理します。

オーバーラップするパッケージは同時にインストールできません。オーバーラップするパッケージの例は、ファイルシステムの同じ場所に異なる内容を配布するパッケージです。このエラーはユーザーがそのパッケージをインストールしようとするまで見つからない可能性があるため、オーバーラップしているパッケージはユーザーエクスペリエンスを低下させることがあります。`pkglint(1)` ツールは、パッケージのオーサリングプロセス中にこのエラーを検出するのに役立つ場合があります。

パッケージの内容が異なる必要がある場合は、`exclude` 依存関係を宣言して、IPS がこれらパッケージをまとめてインストールすることを許可しないようにします。

パッケージのサイズを適切な大きさにします。

パッケージはソフトウェアの単一ユニットを表しており、インストールされるかされないかのどちらかです。(パッケージがオプションのソフトウェアコンポーネントを配布する方法については、83 ページの「オプションのソフトウェアコンポーネント」のファセットの説明を参照してください。)常にまとめてインストールするパッケージは、結合しておくようにします。IPS では更新時に変更されたファイルのみをダウンロードするため、変更が少ない場合は大きなパッケージでもすぐに更新されます。

次のすべてのパスに内容を提供しないでください。

- `/system/volatile`
- `/tmp`
- `/var/pkg`
- `/var/share`
- `/var/tmp`

パッケージの作成および発行

自動化の量が多いために、通常、IPS を使用したソフトウェアのパッケージ化は簡単です。自動化によって、パッケージ化のほとんどのバグの主な原因になっていると思われる単調な反復作業が回避されます。

IPS での発行は、次の手順で構成されます。

1. パッケージマニフェストを生成します。
2. 生成されたマニフェストに必要なメタデータを追加します。
3. 依存関係を評価します。
4. 必要とされるファセットまたはアクチュエータがあれば追加します。
5. パッケージを検証します。
6. パッケージを発行します。
7. パッケージをテストします。

パッケージマニフェストを生成する

インストールしたシステムに必要なコンポーネントファイルを同じディレクトリ構造にまとめることから始めるのがもっとも簡単な方法です。

これを行う 2 つの方法は次のとおりです。

- パッケージ化するソフトウェアがすでに tarball に入っている場合は、その tarball をサブディレクトリに解凍します。`autoconf` ユーティリティーを使用するオープンソースのソフトウェアパッケージの多くでは、目的のプロトタイプ領域を指すように `DESTDIR` 環境変数を設定すると、これが行われます。`autoconf` ユーティリティーは、`pkg:/developer/build/autoconf` パッケージで入手できます。
- メイクファイル内の `install` ターゲットを使用します。

ソフトウェアがバイナリ、ライブラリ、およびマニュアルページで構成され、このソフトウェアを、`mysoftware` という名前の `/opt` 下のディレクトリにインストールするとします。このレイアウトにソフトウェアが表示されるディレクトリをビルド領域に作成します。次の例では、このディレクトリに `proto` という名前が付けられています。

```
proto/opt/mysoftware/lib/mylib.so.1
proto/opt/mysoftware/bin/mycmd
proto/opt/mysoftware/man/man1/mycmd.1
```

`pkgsend generate` コマンドを使用して、この `proto` 領域のマニフェストを生成します。`pkgfmt` によって出力用パッケージマニフェストをパイプで連結して、そのマニフェストをより読み取りやすくします。詳細は、`pkgsend(1)` および `pkgfmt(1)` のマニュアルページを参照してください。

次の例では、`proto` ディレクトリは現在の作業用ディレクトリにあります。

```
$ pkgsend generate proto | pkgfmt > mypkg.p5m.1
```

出力用の `mypkg.p5m.1` ファイルには次の行が含まれています。

```
dir  path=opt owner=root group=bin mode=0755
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
```

```
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
```

パッケージ化されるファイルのパスは、`file` アクションに 2 回出現します。

- `file` という語のあとの最初の語は、`proto` 領域でのそのファイルの場所を表しています。
- `path=` 属性のパスは、そのファイルがインストールされる場所を示しています。

この二重のエントリによって、`proto` 領域を変更しなくてもインストール場所を変更できます。この機能によって、別のオペレーティングシステムにインストールするように作られたソフトウェアを再パッケージ化する場合などに、かなりの時間を節約できる可能性があります。

`pkgsend generate` によってディレクトリの所有者とグループにデフォルト値が適用されていることに注意してください。`/opt` の場合、デフォルトは正しくありません。そのディレクトリはシステム上にすでに存在しているほかのパッケージによって配布されるので削除してください。そうすれば、`/opt` の属性がすでにシステム上に存在しているものと競合する場合、`pkg(1)` はそのパッケージをインストールしません。下記の 46 ページの「生成されたマニフェストに必要なメタデータを追加する」では、不要なディレクトリをプログラムで削除する方法を示しています。

ファイル名に等号 (=)、二重引用符 (")、または空白文字が含まれている場合、次の例に示すように、`pkgsend` は `hash` 属性をマニフェスト内に生成します。

```
$ mkdir -p proto/opt
$ touch proto/opt/my\ file1
$ touch proto/opt/"my file2"
$ touch proto/opt/my=file3
$ touch proto/opt/'my"file4'
$ pkgsend generate proto
dir group=bin mode=0755 owner=root path=opt
file group=bin hash=opt/my=file3 mode=0644 owner=root path=opt/my=file3
file group=bin hash="opt/my file2" mode=0644 owner=root path="opt/my file2"
file group=bin hash='opt/my"file4' mode=0644 owner=root path='opt/my"file4'
file group=bin hash="opt/my file1" mode=0644 owner=root path="opt/my file1"
```

53 ページの「パッケージを発行する」で説明しているように、そのパッケージが発行されると (`Publish the Package`を参照)、26 ページの「ファイルアクション」属性の値はファイル内容の SHA-1 ハッシュになります。

生成されたマニフェストに必要なメタデータを追加する

パッケージでは次のメタデータを定義するようにします。これらの値とその設定方法の詳細は、33 ページの「設定アクション」を参照してください。

`pkg.fmri`

パッケージの名前とバージョン (22 ページの「パッケージ名」および 23 ページの「パッケージバージョン」 を参照)。パッケージ名と依存関係については、99 ページの「パッケージ内容の競合の回避」を参照してください。53 ページの「パッケージを発行する」に示すように、パッケージが発行されるとパブリッシャー名が自動的に追加されます。パッケージバージョンを設定するための追加のヘルプについては、48 ページの「適切なパッケージバージョン文字列の作成」を参照してください。Oracle Solaris でのバージョン管理については、145 ページの「Oracle Solaris パッケージのバージョン管理」を参照してください。

`pkg.description`

パッケージの内容の説明

`pkg.summary`

その説明の 1 行の概要。

`variant.arch`

このパッケージに適した各アーキテクチャー。パッケージ全体をどのアーキテクチャーでもインストールできる場合は、`variant.arch` を省略できます。さまざまなアーキテクチャーのさまざまなコンポーネントを含むパッケージの作成については、第5章「バリエーションの許可」で説明しています。

`info.classification`

`packagemanager(1)` GUI で使用されるグループ化スキーム。サポートされる値は、付録A パッケージの分類に記載しています。このセクション内の例では、任意の分類が指定されています。

この例ではまた、`mysoftware` 下の `man` ディレクトリを指す `link` アクションを `/usr/share/man/index.d` に追加します。このリンクについては、51 ページの「必要とされるファセットまたはアクチュエータがあれば追加する」で詳しく説明されています。

生成されたマニフェストを直接変更するのではなく、`pkgmogrify(1)` を使用して生成されたマニフェストを編集します。`pkgmogrify` を使用してパッケージマニフェストを変更する方法の詳細は、第6章「プログラムによるパッケージマニフェストの変更」を参照してください。

次の `pkgmogrify` 入力ファイルを作成して、マニフェストに加えられる変更を指定します。このファイルを `mypkg.mog` という名前にします。この例では、アーキテクチャーの定義にマクロが使用され、マニフェストから `/opt` ディレクトリを削除するのに正規表現の照合が使用されます。

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description value="This is a full description of \
```

```
all the interesting attributes of this example package."
set name=variant.arch value=$(ARCH)
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
link path=usr/share/man/index.d/mysoftware target=/opt/mysoftware/man
<transform dir path=opt$->drop>
```

mypkg.mog の変更とともに mypkg.p5m.1 マニフェストで pkgmogrify を実行します。

```
$ pkgmogrify -DARCH=`uname -p` mypkg.p5m.1 mypkg.mog | pkgfmt > mypkg.p5m.2
```

出力用の mypkg.p5m.2 ファイルには次の内容が含まれています。path=opt の dir アクションが削除され、mypkg.mog からのメタデータとリンクの内容が元の内容 mypkg.p5m.1 に追加されました。

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this
example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=/opt/mysoftware/man
```

適切なパッケージバージョン文字列の作成

IPS パッケージバージョン文字列のコンポーネントバージョン、リリース、およびブランチバージョン (23 ページの「パッケージバージョン」を参照してください) には、次の制約があります。

- すべての内容がピリオドまたは整数だけである必要があります。製品バージョンにほかの文字 (英字など) が含まれている場合は、整数とピリオドだけを使用して同じ意味を伝える IPS パッケージのバージョンを選択します。たとえば、製品バージョンが P17-u4-r3 である場合は、パッケージバージョンとして 17.4.3 を使用できます。
- 複数の整数の並びを 0 で始めることはできません。この形式により、パッケージバージョンでのソートが可能になります。たとえば、バージョン 1.20 はソートでバージョン 1.0.2 より新しくなりますが、1.02 は無効です。製品バージョン 17.03 がその製品のバージョン 17 の 3 番目のテストリリースを示し、最終的にリリースされる製品がバージョン 17.0 になる場合は、テストリリースとして 16.99.3、16.99.4 などのパッケージバージョンを使用できます。バージョン 17.0 が 16.99

バージョンより新しいと見なされるのに対して、17.0 は 17.0.3 より新しくありません。

次の例に示すように、pkg.human-version 属性を使用して実際の製品バージョン文字列を指定できます。

```
set name=pkg.human-version value="P17-u4-r3"
```

パッケージ FMRI のパッケージバージョンに加えて pkg.human-version 属性の値を指定できますが、この値によってパッケージ FMRI バージョンを置き換えることはできません。pkg.human-version のバージョン文字列は、表示のためにのみ使用されます。詳細は、33 ページの「設定アクション」を参照してください。

依存関係を評価する

pkgdepend(1) コマンドを使用して、パッケージの依存関係を自動的に生成します。生成される depend アクションは 35 ページの「依存アクション」に定義しており、第4章「パッケージ依存関係の指定」で詳しく説明しています。

依存関係の生成は 2 つの別々の手順から成ります。

1. 依存関係の生成。ソフトウェアが依存するファイルを特定します。pkgdepend generate コマンドを使用します。
2. 依存関係の解決。ソフトウェアが依存するそれらのファイルを含むパッケージを特定します。pkgdepend resolve コマンドを使用します。

パッケージの依存関係を生成する

ヒント - depend アクションを手動で宣言するのではなく、pkgdepend を使用して依存関係を生成するようにします。手動による依存関係は、パッケージの内容が時間の経過に伴って変わると、不正確または不必要になることがあります。たとえば、アプリケーションが依存しているファイルが別のパッケージに移動されると、手動で宣言した前のパッケージへの依存関係はどれもその依存関係にとって正しくないものになります。

手動で宣言した依存関係の一部は、pkgdepend が依存関係を完全に特定できない場合に必要になることがあります。そのような場合は、説明のコメントをマニフェストに追加するようにしてください。

次のコマンドでは、-m オプションによって pkgdepend はマニフェスト全体をその出力に含めます。-d オプションは proto ディレクトリをコマンドに渡します。

```
$ pkgdepend generate -md proto mypkg.p5m.2 | pkgfmt > mypkg.p5m.3
```

出力用の mypkg.p5m.3 ファイルには次の内容が含まれています。pkgdepend ユーティリティーは、mylib.so.1 と mycmd の両方による libc.so.1 への依存関係についての表記を追加しました。mycmd と mylib.so.1 の間の内部依存関係は暗黙のうちに省略されます。

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this
example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=/opt/mysoftware/man
depend fmri=__TBD pkg.debug.depend.file=libc.so.1 \
    pkg.debug.depend.reason=opt/mysoftware/bin/mycmd \
    pkg.debug.depend.type=elf type=require pkg.debug.depend.path=lib \
    pkg.debug.depend.path=opt/mysoftware/lib pkg.debug.depend.path=usr/lib
depend fmri=__TBD pkg.debug.depend.file=libc.so.1 \
    pkg.debug.depend.reason=opt/mysoftware/lib/mylib.so.1 \
    pkg.debug.depend.type=elf type=require pkg.debug.depend.path=lib \
    pkg.debug.depend.path=usr/lib
```

パッケージの依存関係を解決する

依存関係を解決するには、pkgdepend で、イメージに現在インストールされているパッケージのうち、ソフトウェアの構築に使用されるものを調べます。デフォルトでは、pkgdepend はその出力を mypkg.p5m.3.res に格納します。この手順ではそれが実行されているシステムに関する大量の情報をロードするため、実行するのにしばらく時間がかかります。この時間をすべてパッケージに分散させると、pkgdepend ユーティリティーは同時に多くのパッケージを解決できます。1 度に 1 つのパッケージに対して pkgdepend を実行すると、時間効率がよくありません。

```
$ pkgdepend resolve -m mypkg.p5m.3
```

これが完了すると、出力用の mypkg.p5m.3.res ファイルには次の内容が含まれています。pkgdepend ユーティリティーは、libc.so.1 へのファイル依存関係についての表記を、そのファイルを配布する pkg:/system/library へのパッケージ依存関係に変換しました。

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this
```

```
example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=/opt/mysoftware/man
depend fmri=pkg:/system/library@0.5.11-0.175.2.0.0.18.0 type=require
```

必要とされるファセットまたはアクチュエータがあれば追加する

ファセットは、必須ではないがオプションでインストールできるアクションを示します。アクチュエータは、関連するアクションがインストール、更新、または削除されたときに発生する必要があるシステム変更を指定します。ファセットについては、第5章「バリエーションの許可」で詳しく説明しており、アクチュエータについては、第7章「パッケージインストールの一環としてのシステム変更の自動化」で詳しく説明しています。

このサンプルパッケージは、マニュアルページを `opt/mysoftware/man/man1` に配布します。このセクションでは、マニュアルページがオプションであることを示すファセットタグを追加する方法について説明します。ユーザーは、そのパッケージのマニュアルページ以外のすべてをインストールすることを選択できます。(ユーザーが『Oracle Solaris 11.3 ソフトウェアの追加と更新』の「オプションのコンポーネントのインストールの制御」で説明されているように、ファセットプロパティー `doc.man=false` を設定した場合、`facet.doc.man=true` でタグ付けされたアクションはどのパッケージからもインストールされません)。

マニュアルページをインデックスに含めるには、そのパッケージがインストールされるときに `svc:/application/man-index:default` SMF サービスを再起動する必要があります。このセクションでは、`restart_fmri` アクチュエータを追加してそのタスクを実行する方法について説明します。`man-index` サービスは、マニュアルページを含むディレクトリへのシンボリックリンクを `/usr/share/man/index.d` で探して、各リンクのターゲットを、それがスキャンするディレクトリのリストに追加します。マニュアルページをインデックスに含めるために、このサンプルパッケージには `/usr/share/man/index.d/mysoftware` から `/opt/mysoftware/man` へのリンクが含まれています。このリンクとこのアクチュエータを含めることは、15 ページの「ソフトウェアの自己アセンブリ」で説明されている、Oracle Solaris OS のパッケージ化全体にわたって使用される自己アセンブリのよい例です。

使用できる一連の pkgmogrify 変換は /usr/share/pkg/transforms で入手できます。これらの変換は Oracle Solaris OS のパッケージ化に使用され、第6章「プログラムによるパッケージマニフェストの変更」で詳しく説明しています。

ファイル /usr/share/pkg/transforms/documentation には、この例でマニュアルページのファセットを設定したり、man-index サービスを再起動したりするために必要な変換と同じような変換が含まれています。この例はマニュアルページを /opt に配布するため、documentation 変換は下記のように変更される必要があります。これらの変更された変換には正規表現 opt/.+/man(/.+)? が含まれており、これは man サブディレクトリを含む、opt のすぐ下のすべてパスに一致します。次の変更された変換を /tmp/doc-transform に保存します。

```
<transform dir file link hardlink path=opt/.+/man(/.+)? -> \
    default facet.doc.man true>
<transform file path=opt/.+/man(/.+)? -> \
    add restart_fmri svc:/application/man-index:default>
```

次のコマンドを使用して、これらの変換をマニフェストに適用します。

```
$ pkgmogrify mypkg.p5m.3.res /tmp/doc-transform | pkgfmt > mypkg.p5m.4.res
```

入力用の mypkg.p5m.3.res マニフェストには、次の 3 つのマニュアルページ関連アクションが含まれています。

```
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
```

変換の適用後、出力用の mypkg.p5m.4.res マニフェストには次の変更されたアクションが含まれています。

```
dir  path=opt/mysoftware/man owner=root group=bin mode=0755 facet.doc.man=true
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755 \
    facet.doc.man=true
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644 \
    restart_fmri=svc:/application/man-index:default facet.doc.man=true
```

ヒント - 効率を良くするため、これらの変換は、依存関係の評価前の、メタデータが最初に追加されたときに追加することもできました。

パッケージを検証する

発行前の最後の手順は、マニフェストに対して pkglint(1) を実行して、発行およびテスト前に特定できるエラーを見つけることです。pkglint で見つけられるエラーの中には、発行時またはユーザーがパッケージのインストールを試みたときに見つかるものもありますが、当然ながらパッケージオーサリングプロセスでできるだけ早くエラーを特定するのが望ましいことです。

pkglint で報告されるエラーの例には次があります。

- 配布するファイルがすでに別のパッケージで所有されている。
- ディレクトリのような、共有された参照カウントアクションのメタデータに違いがある。このエラーの例については、45 ページの「パッケージマニフェストを生成する」の終わりで説明されています。

`pkglint` が実行するチェックの完全なリストを表示するには、`pkglint -L` コマンドを使用します。特定のチェックを有効化、無効化、および省略する方法の詳細な情報は、`pkglint(1)` のマニュアルページに記載されています。このマニュアルページには、追加のチェックを実行するために `pkglint` を拡張する方法も詳述されています。

`pkglint` は次のいずれかのモードで実行できます。

- パッケージマニフェストに対して直接。通常、マニフェストの有効性をすばやくチェックするためには、このモードで十分です。
- パッケージマニフェストに対して (パッケージリポジトリも参照)。このモードは、リポジトリへの発行前に少なくとも 1 回は使用してください。

  リポジトリを参照することで、`pkglint` は追加のチェックを行なって、パッケージがそのリポジトリ内のほかのパッケージとうまく相互作用することを確認できます。

次の出力には、サンプルマニフェストの問題が示されています。

```
$ pkglint mypkg.p5m.4.res
Lint engine setup...
Starting lint run...
WARNING pkglint.action005.1       obsolete dependency check skipped: unable
to find dependency pkg:/system/library@0.5.11-0.175.2.0.0.18.0 for
pkg:/mypkg@1.0,5.11-0
```

この警告はこの例で許容されます。`pkglint.action005.1` 警告は、`pkglint` が、このサンプルパッケージが依存している `pkg:/system/library@0.5.11-0.175.2.0.0.18.0` と呼ばれるパッケージを見つけられなかったことを伝えています。この依存関係パッケージはパッケージリポジトリ内にあり、`pkglint` が引数にマニフェストファイルのみを指定して呼び出されたため、見つかりませんでした。

次のコマンドでは、`-r` オプションによって依存関係パッケージを含むリポジトリが参照されます。`-c` オプションは、lint リポジトリおよび参照リポジトリからパッケージのメタデータをキャッシュするために使用されるローカルディレクトリを指定します。

```
$ pkglint -c ./solaris-reference -r http://pkg.oracle.com/solaris/release mypkg.p5m.4.res
```

パッケージを発行する

ローカルファイルベースのリポジトリにパッケージを発行します。このリポジトリは、この新しいパッケージの開発およびテスト用のリポジトリです。一般的な使用の目的でリポジトリを作成する場合は、そのリポジトリ用の個別ファイルシステムの作成などの追加手順を含めてください。一般的な用途のパッケージリポジトリを作成す

ることについては、『Oracle Solaris 11.3 パッケージリポジトリのコピーと作成』を参照してください。

非大域ゾーンを含むパッケージをテストするには、システムリポジトリを介してリポジトリの場所にアクセス可能である必要があります。非大域ゾーン内で `pkg publisher` または `pkg list`コマンドを使用して、パッケージがアクセス可能であることを確認します。

`pkgrepo(1)` コマンドを使用して、システムにリポジトリを作成します。

```
$ pkgrepo create my-repository
$ ls my-repository
pkg5.repository
```

このリポジトリのデフォルトパブリッシャーを設定します。デフォルトパブリッシャーは、リポジトリの `publisher/prefix` プロパティーの値です。

```
$ pkgrepo -s my-repository set publisher/prefix=mypublisher
```

`pkgsend publish` コマンドを使用して、新しいパッケージを発行します。複数の `pkgsend publish` プロセスが同じ `-s` リポジトリに対して同時に発行する場合、パブリッシャーカタログに対する更新は連続して実行する必要があるため、`--no-catalog` オプションを指定することを推奨します。複数のプロセスが同時にパッケージを発行するときは、`--no-catalog` オプションを使用しないと、発行のパフォーマンスが大幅に低下する可能性があります。公開の完了後、`pkgrepo refresh` コマンドを使用してそれぞれのパブリッシャーカタログに新しいパッケージを追加できます。

```
$ pkgsend -s my-repository publish -d proto mypkg.p5m.4.res
pkg://mypublisher/mypkg@1.0,5.11-0:20130720T005452Z
PUBLISHED
```

リポジトリのデフォルトパブリッシャーがパッケージ FMRI に適用されている点に注意してください。

新しいリポジトリのアクセス権、内容、および署名が正しいことを確認します。

```
$ pkgrepo verify -s my-repository
```

リポジトリを調べるには、`pkgrepo` および `pkg list` コマンドを使用できます。

```
$ pkgrepo info -s my-repository
PUBLISHER   PACKAGES STATUS           UPDATED
mypublisher 1        online           2013-07-20T00:54:52.758591Z
$ pkgrepo list -s my-repository
PUBLISHER   NAME                    O VERSION
mypublisher mypkg                     1.0,5.11-0:20130720T005452Z
$ pkg list -afv -g my-repository
FMRI                                                IFO
pkg://mypublisher/mypkg@1.0,5.11-0:20130720T005452Z      ---
```

`pkgrepo list` の `O` 列は、パッケージが廃止された (`o`) または名前変更された (`r`) ことを示します。

HTTP 経由で発行するときには着信パッケージに対する承認または認証のチェックがないため、通常、HTTP リポジトリに新しいパッケージを直接発行する

ことはお勧めしません。パッケージを HTTP リポジトリに発行する代わりに、57 ページの「パッケージリポジトリへの配布」で説明するように、すでに発行されているパッケージを HTTP リポジトリに配布します。HTTP リポジトリへの発行は、ファイルリポジトリへの NFS または SMB アクセスが不可能な場合にいくつかのシステムにわたって同じパッケージをテストするときや、セキュリティー保護されたネットワーク上で便利なことがあります。HTTP リポジトリに直接発行する場合、そのリポジトリは `svc:/application/pkg/server` サービスの読み取り/書き込みインスタンスがあるシステム (`pkg/readonly` プロパティーの値が `false`) でホストされている必要があります。

パッケージの署名

パッケージに署名する場合は、この時点で行なってからパッケージをテストし、一般的な用途のリポジトリまたはパッケージアーカイブにパッケージを配布します。

次のコマンドは、パッケージマニフェストのハッシュ値を使用して、パッケージに署名します。独自の署名鍵と証明書を指定することもできます。詳細については、第9章「IPS パッケージの署名」を参照してください。パッケージのタイムスタンプは変更されていません。

```
$ pkgsign -s my-repository -a sha256 '*'
Signed pkg://mypublisher/mypkg@1.0,5.11-0:20130720T005452Z
```

パッケージをテストする

パッケージ開発の最後の手順は、パッケージをインストールして、発行されたパッケージが正しくパッケージ化されているかどうかをテストすることです。

root 権限を必要としないでインストールをテストするには、テストユーザーにソフトウェアインストールプロファイルを割り当てます。テストユーザーにソフトウェアインストールに関連するプロファイルを割り当てるには、`usermod` コマンドの `-P` オプションを使用します。

注記 - このイメージに子イメージ (非大域ゾーン) がインストールされている場合は、`pkg install` コマンドで `-g` オプションを使用して、このパッケージのインストールをテストすることはできません。イメージ内で `mypublisher` パブリッシャーを構成する必要があります。

次の `pkg set-publisher` コマンドは、`my-repository` リポジトリ内のすべてのパブリッシャーを、このイメージで構成されているパブリッシャーの一覧に追加します。

```
$ pkg publisher
PUBLISHER    TYPE    STATUS P LOCATION
solaris      origin  online F http://pkg.oracle.com/solaris/release/
$ pkg set-publisher -p my-repository
```

```
pkg set-publisher:
  Added publisher(s): mypublisher
$ pkg publisher
PUBLISHER    TYPE    STATUS P LOCATION
solaris      origin  online F http://pkg.oracle.com/solaris/release/
mypublisher  origin  online F file:///home/username/my-repository/
```

イメージを変更せずに、インストールコマンドが実行する内容を確認するには、`pkg install` コマンドに `-nv` オプションを使用します。次のコマンドでは、実際にパッケージをインストールします。

```
$ pkg install mypkg
           Packages to install:  1
       Create boot environment: No
Create backup boot environment: No
            Services to change:  1

DOWNLOAD                                  PKGS       FILES   XFER (MB)   SPEED
Completed                                  1/1         3/3     0.0/0.0  787k/s

PHASE                                        ITEMS
Installing new actions                       16/16
Updating package state database               Done
Updating image state                          Done
Creating fast lookup database                 Done
Reading search index                          Done
Updating search index                          1/1
```

システム上に配布されたソフトウェアを調べます。

```
$ find /opt/mysoftware
/opt/mysoftware
/opt/mysoftware/bin
/opt/mysoftware/bin/mycmd
/opt/mysoftware/lib
/opt/mysoftware/lib/mylib.so.1
/opt/mysoftware/man
/opt/mysoftware/man/man1
/opt/mysoftware/man/man1/mycmd.1
/opt/mysoftware/man/man-index
/opt/mysoftware/man/man-index/term.dic
/opt/mysoftware/man/man-index/term.req
/opt/mysoftware/man/man-index/term.pos
/opt/mysoftware/man/man-index/term.exp
/opt/mysoftware/man/man-index/term.doc
/opt/mysoftware/man/man-index/.index-cache
/opt/mysoftware/man/man-index/term.idx
```

バイナリとマニュアルページに加えて、アクチュエータが `man-index` サービスを再起動した結果としてマニュアルページのインデックスもシステムにより生成されました。

`pkg info` コマンドは、パッケージに追加されたメタデータを表示します。

```
$ pkg info mypkg
          Name: mypkg
       Summary: This is an example package
   Description: This is a full description of all the interesting attributes of
                this example package.
      Category: Applications/Accessories
         State: Installed
```

```
     Publisher: mypublisher
       Version: 1.0
 Build Release: 5.11
        Branch: 0
Packaging Date: July 20, 2013 00:54:52 AM
          Size: 12.95 kB
          FMRI: pkg://mypublisher/mypkg@1.0,5.11-0:20130720T005452Z
```

`pkg search` コマンドは、`mypkg` によって配布されるファイルのクエリー検索時にヒットしたものを返します。

```
$ pkg search -l mycmd
INDEX      ACTION VALUE                    PACKAGE
basename   file   opt/mysoftware/bin/mycmd pkg:/mypkg@1.0-0
```

パッケージを配布する

IPS には、ユーザーがパッケージをインストールできるようにする 3 通りのパッケージ配布方法があります。

ローカルファイルベースのリポジトリ

ユーザーはローカルネットワーク経由でこのリポジトリにアクセスします。パブリッシャーの起点はリポジトリのパス (`/net/host1/export/ipsrepo` など) です。

リモート HTTP ベースのリポジトリ

ユーザーは HTTP または HTTPS 経由でこのリポジトリにアクセスします。パブリッシャーの起点は `http://pkg.example.com/` などのアドレスです。

パッケージアーカイブ

パッケージアーカイブはスタンドアロンファイルです。パブリッシャーの起点はアーカイブファイルのパス (`/net/host1/export/ipsarchive.p5p` など) です。

どの場合でも、パッケージは 53 ページの「パッケージを発行する」で説明するように `pkgsend publish` コマンドを使用してすでに発行されています。一般的な用途のために既存のリポジトリまたはパッケージアーカイブにパッケージを取得するには、`pkgrecv` コマンドを使用します。詳細は、`pkgrecv(1)` のマニュアルページを参照してください。一般的な使用のためにリポジトリを作成および保持する方法については、『Oracle Solaris 11.3 パッケージリポジトリのコピーと作成』を参照してください。

パッケージリポジトリへの配布

次の例は、一般的な使用の目的でセットアップされたローカルファイルリポジトリに、テストリポジトリから新しいパッケージを配布する方法を示します。この例のパッケージは小さいため、get と send のサイズはゼロです。

```
$ pkgrecv -s my-repository -d /net/host1/export/ipsrepo mypkg
Processing packages for publisher mypublisher ...
Retrieving and evaluating 1 package(s)...
PROCESS                                     ITEMS   GET (MB)   SEND (MB)
Completed                                     1/1    0.0/0.0     0.0/0.0
```

新しいリポジトリにパッケージが存在することを確認します。

```
$ pkgrepo info -s /net/host1/export/ipsrepo
PUBLISHER   PACKAGES STATUS           UPDATED
solaris     4455     online           2013-07-09T23:41:24.312974Z
mypublisher 1        online           2013-07-22T20:57:36.951042Z
$ pkgrepo list -p mypublisher -s /net/host1/export/ipsrepo
PUBLISHER   NAME                    O VERSION
mypublisher mypkg                     1.0,5.11-0:20130720T005452Z
```

同じ pkgrecv コマンドを使用して、HTTP または HTTPS リポジトリにパッケージを配布します。この場合、-d 引数として、適切な pkg/server サービスインスタンスの pkg/inst_root プロパティーの値を指定します。このリポジトリは、pkg.depotd を実行する svc:/application/pkg/server サービスによりユーザーに提供されます。詳細は、pkg.depotd(1M) のマニュアルページを参照してください。

このイメージに子イメージ (非大域ゾーン) がない場合、次のコマンドに示すように、ユーザーは -g オプションを使用して新しいパッケージをインストールできます。-g オプションは、このイメージで構成されているパブリッシャーの一覧に mypublisher パブリッシャーを追加します。

```
$ pkg install -g /net/host1/export/ipsrepo mypkg
```

このイメージに子イメージが含まれている場合、次のコマンドに示すように、ユーザーはイメージ内で mypublisher パブリッシャーを構成する必要があります。

```
$ pkg set-publisher -p /net/host1/export/ipsrepo
```

パッケージアーカイブファイルとしての配布

パッケージアーカイブは、パブリッシャー情報と、そのパブリッシャーから提供される 1 つまたは複数のパッケージが含まれているスタンドアロンファイルです。パッケージをパッケージアーカイブとして配布する方法は、パッケージリポジトリにアクセスできないユーザーにとって便利です。パッケージアーカイブは、Web サイトからのダウンロード、USB キーへのコピー、DVD への書き込みが容易にできます。

pkgrecv コマンドでは、パッケージリポジトリからパッケージアーカイブにパッケージを追加したり、パッケージアーカイブからパッケージリポジトリにパッケージを追加したりすることができます。パッケージアーカイブからパッケージリポジトリにパッケージを追加する場合は、パッケージアーカイブにはデフォルトのパブリッシャー接頭辞などのリポジトリ構成が含まれていない点に注意してください。ほとんどの pkgrepo サブコマンドは、パッケージアーカイブでは動作しません。pkgrepo list コマンドはパッケージアーカイブで動作します。

次のコマンドは、mypkg パッケージのパッケージアーカイブを作成します。このアーカイブはまだ存在しないため、-a オプションを指定する必要があります。慣例的に、パッケージアーカイブにはファイル拡張子 .p5p が付きます。

```
$ pkgrecv -s my-repository -a -d myarchive.p5p mypkg
Retrieving packages for publisher mypublisher ...
Retrieving and evaluating 1 package(s)...
DOWNLOAD                                  PKGS       FILES   XFER (MB)   SPEED
Completed                                  1/1         3/3     0.0/0.0  782k/s

ARCHIVE                                               FILES   STORE (MB)
myarchive.p5p                                         14/14      0.0/0.0
```

このイメージに子イメージ (非大域ゾーン) がない場合、次のコマンドに示すように、ユーザーは -g オプションを使用して新しいパッケージをインストールできます。-g オプションは、このイメージで構成されているパブリッシャーの一覧に mypublisher パブリッシャーを追加します。

```
$ pkg install -g myarchive.p5p mypkg
```

このイメージに子イメージが含まれている場合、次のコマンドに示すように、ユーザーはイメージ内で mypublisher パブリッシャーを構成する必要があります。

```
$ pkg set-publisher -p myarchive.p5p
```

パッケージアーカイブを、非大域ゾーンのローカルパブリッシャーのソースとして設定できます。

パッケージリポジトリとパッケージアーカイブの使用

pkgrepo コマンドを使用して、リポジトリまたはアーカイブから使用可能な最新のパッケージを一覧表示します。

```
$ pkgrepo list -s my-repository '*@latest'
PUBLISHER   NAME                                                  O VERSION
mypublisher mypkg                                                   1.0,5.11-0:20130720T005452Z
$ pkgrepo list -s myarchive.p5p '*@latest'
PUBLISHER   NAME                                                  O VERSION
mypublisher mypkg                                                   1.0,5.11-0:20130720T005452Z
```

この出力は、ある特定のリポジトリから最新バージョンのすべてのパッケージでアーカイブを作成するためのスクリプトの構築に役立つことがあります。

SVR4 パッケージから IPS パッケージへの変換

このセクションでは、SVR4 パッケージから IPS パッケージへの変換例について説明し、特別な注意が必要な部分を強調して説明します。

SVR4 パッケージから IPS パッケージに変換するには、この章で前述した、IPS でソフトウェアをパッケージ化するための手順と同じ手順に従います。これらの手順のほ

とんどは、SVR4 から IPS パッケージへの変換でも同じであるため、このセクションでは再度説明しません。このセクションでは、新しいパッケージの作成時ではなく、パッケージの変換時に異なる手順について説明します。

SVR4 パッケージから IPS パッケージマニフェストを生成する

`pkgsend generate` コマンドの *source* 引数は SVR4 パッケージにできます。サポートされるソースの完全なリストについては、pkgsend(1) のマニュアルページを参照してください。*source* が SVR4 パッケージである場合、`pkgsend generate` は配布するファイルを含むパッケージ内部のディレクトリではなく、その SVR4 パッケージ内の pkgmap(4) ファイルを使用します。

`prototype` ファイルのスキャン中に、`pkgsend` ユーティリティーはそのパッケージを IPS に変換する際に問題を引き起こす可能性のあるエントリの検索も行います。`pkgsend` ユーティリティーは、それらの問題を報告し、生成されたマニフェストを出力します。

このセクションで使用されるサンプルの SVR4 パッケージには、次の pkginfo(4) ファイルが含まれています。

```
VENDOR=My Software Inc.
HOTLINE=Please contact your local service provider
PKG=MSFTmypkg
ARCH=i386
DESC=A sample SVR4 package of My Sample Package
CATEGORY=system
NAME=My Sample Package
BASEDIR=/
VERSION=11.11,REV=2011.10.17.14.08
CLASSES=none manpage
PSTAMP=linn20111017132525
MSFT_DATA=Some extra package metadata
```

このセクションで使用されるサンプルの SVR4 パッケージには、次の対応する prototype(4) ファイルが含まれています。

```
i pkginfo
i copyright
i postinstall
d none opt 0755 root bin
d none opt/mysoftware 0755 root bin
d none opt/mysoftware/lib 0755 root bin
f none opt/mysoftware/lib/mylib.so.1 0644 root bin
d none opt/mysoftware/bin 0755 root bin
f none opt/mysoftware/bin/mycmd 0755 root bin
d none opt/mysoftware/man 0755 root bin
d none opt/mysoftware/man/man1 0755 root bin
f none opt/mysoftware/man/man1/mycmd.1 0644 root bin
```

これらのファイルを使用して構築された SVR4 パッケージに対して `pkgsend generate` コマンドを実行すると、次の IPS マニフェストが生成されます。

```
$ pkgsend generate ./MSFTmypkg | pkgfmt
pkgsend generate: ERROR: script present in MSFTmypkg: postinstall

set name=pkg.summary value="My Sample Package"
set name=pkg.description value="A sample SVR4 package of My Sample Package"
set name=pkg.send.convert.msft-data value="Some extra package metadata"
dir path=opt owner=root group=bin mode=0755
dir path=opt/mysoftware owner=root group=bin mode=0755
dir path=opt/mysoftware/bin owner=root group=bin mode=0755
file reloc/opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0755
dir path=opt/mysoftware/lib owner=root group=bin mode=0755
file reloc/opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir path=opt/mysoftware/man owner=root group=bin mode=0755
dir path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file reloc/opt/mysoftware/man/man1/mycmd.1 \
    path=opt/mysoftware/man/man1/mycmd.1 owner=root group=bin mode=0644
legacy pkg=MSFTmypkg arch=i386 category=system \
    desc="A sample SVR4 package of My Sample Package" \
    hotline="Please contact your local service provider" \
    name="My Sample Package" vendor="My Software Inc." \
    version=11.11,REV=2011.10.17.14.08
license install/copyright license=MSFTmypkg.copyright
```

`pkgsend generate` の出力に関する次の点に注意してください。

- `pkg.summary` および `pkg.description` 属性は `pkginfo` ファイル内のデータから自動的に作成されました。
- `set` アクションは `pkginfo` ファイル内の追加パラメータから生成されました。この `set` アクションは `pkg.send.convert.*` 名前空間の下に設定されています。`pkgmogrify(1)` 変換を使用して、そのような属性をより適切な属性名に変換してください。
- `legacy` アクションが `pkginfo` ファイル内のデータから生成されました。
- SVR4 パッケージ内で使用されたコピーライトファイルを指す `license` アクションが生成されました。
- 変換できないスクリプト作成操作に関するエラーメッセージが発行されました。

次のチェックでは、`pkgsend generate` からのエラーメッセージとゼロ以外の戻りコードが示されています。

```
$ pkgsend generate MSFTmypkg > /dev/null
pkgsend generate: ERROR: script present in MSFTmypkg: postinstall
$ echo $?
1
```

SVR4 パッケージは、IPS の同等のものに直接変換できない `postinstall` スクリプトを使用しています。このスクリプトは手動で検査する必要があります。

パッケージ内の `postinstall` スクリプトには次の内容が含まれています。

```
#!/usr/bin/sh
catman -M /opt/mysoftware/man
```

51 ページの「必要とされるファセットまたはアクチュエータがあれば追加する」で説明しているように、既存の SMF サービス `svc:/application/man-index:`

`default` を指す `restart_fmri` アクチュエータを使用すれば、このスクリプトと同じ結果を得ることができます。アクチュエータの詳細は、第7章「パッケージインストールの一環としてのシステム変更の自動化」を参照してください。

`pkgsend generate` コマンドでは、クラスアクションスクリプトの存在も確認し、調査すべきスクリプトを示すエラーメッセージを生成します。

SVR4 パッケージから IPS パッケージへのどの変換でも、必要な機能はおそらく既存のアクションタイプまたは SMF サービスを使用して実装できます。使用できるアクションタイプの詳細は、25 ページの「パッケージの内容: アクション」を参照してください。SMF およびパッケージのアクションについては、第7章「パッケージインストールの一環としてのシステム変更の自動化」を参照してください。

パッケージメタデータの追加と依存関係の解決は、44 ページの「パッケージの作成および発行」で説明されているのと同じ方法で行われるため、このセクションでは説明しません。変換されたパッケージに関する固有の問題を示す可能性のある次のパッケージ作成手順は、検証手順です。

変換されたパッケージを検証する

SVR4 パッケージの変換時に発生するエラーの一般的な原因は、SVR4 パッケージで配布されたディレクトリと IPS パッケージによって配布された同じディレクトリの間の属性の不一致です。

この例の SVR4 パッケージでは、サンプルマニフェスト内の `/opt` に対するディレクトリアクションには、システムパッケージによってこのディレクトリ用に定義されている属性とは異なる属性が含まれています。

31 ページの「ディレクトリアクション」セクションでは、すべての参照カウントアクションには同じ属性が含まれている必要があると明確に記述されていました。これまで生成してきた `mypkg` のバージョンをインストールしようとすると、次のエラーが発生します。

```
$ pkg install mypkg
Creating Plan /
pkg install: The requested change to the system attempts to install multiple actions
for dir 'opt' with conflicting attributes:

    1 package delivers 'dir group=bin mode=0755 owner=root path=opt':
        pkg://mypublisher/mypkg@1.0,5.11-0:20111017T020042Z
    1 package delivers 'dir group=sys mode=0755 owner=root path=opt':
        pkg://solaris/system/core-os@0.5.11,5.11-0.175.3.13.0.3.0:20160926T221733Z

These packages may not be installed together. Any non-conflicting set may
be, or the packages must be corrected before they can be installed.
```

インストール時ではなく、パッケージの発行前にエラーを見つけるには、次の例に示すように、`pkglint(1)` コマンドを参照リポジトリとともに使用します。

```
$ pkglint -c ./cache -r file:///scratch/solaris-repo ./mypkg.mf.res
Lint engine setup...

PHASE                                          ITEMS
4                                          4292/4292
Starting lint run...

ERROR pkglint.dupaction007        path opt is reference-counted but has different
 attributes across 5
duplicates: group: bin -> mypkg group: sys -> developer/build/onbld system/core-os system/
ldoms/ldomsmanager
```

異なるパッケージに異なる属性が含まれている `path opt` に関するエラーメッセージに注意してください。

`pkglint` によって報告される追加の `ldomsmanager` パッケージは参照パッケージリポジトリ内にありますが、テストシステムにはインストールされません。`ldomsmanager` パッケージはインストールされないため、前に `pkg install` によって報告されたエラーには記載されていません。

パッケージ変換に関するその他の考慮事項

Oracle Solaris 11 システムでは SVR4 パッケージを直接インストールできますが、それより IPS パッケージを作成するようにしてください。SVR4 パッケージのインストールは暫定的な解決です。

`Legacy Actions`で説明した 38 ページの「レガシーアクション」 アクションは別として、2 つのパッケージシステム間には関連性がなく、SVR4 パッケージと IPS パッケージはそれぞれから互いにパッケージメタデータを参照することはありません。

IPS には、パッケージ化された内容が正しくインストールされているかどうかを確認できる、`pkg verify` などのコマンドがあります。ただし、別のパッケージシステムが正当にパッケージをインストールしたり、IPS パッケージによってインストールされたディレクトリやファイルを変更するインストールスクリプトを実行したりした場合は、エラーが発生することがあります。

IPS の `pkg fix` および `pkg revert` コマンドは、IPS パッケージだけでなく SVR4 パッケージによって配布されたファイルも上書きでき、それによってパッケージ化されたアプリケーションが正常に機能しなくなる可能性があります。

`pkg install` などのコマンドは、通常は重複するアクションや参照カウントアクションの一般的な属性をチェックしますが、異なるパッケージシステムからのファイルが競合するときは、潜在的なエラーの検出に失敗する可能性があります。

これらの潜在的なエラーを考慮に入れ、IPS には包括的なパッケージ開発ツールチェーンがあることを考えると、Oracle Solaris 11 では SVR4 パッケージではなく IPS パッケージを開発することが推奨されます。

♦ ♦ ♦ 第 3 章

3 ソフトウェアパッケージのインストール、削除、および更新

この章では、イメージにインストールされたソフトウェアのインストール、更新、および削除時に IPS pkg クライアントが内部的にどのように機能するかについて説明します。

pkg がこれらの操作をどのように実行するかを理解することは、発生する可能性のあるさまざまなエラーを理解したり、パッケージの依存関係の問題を迅速に解決したりするために重要です。

パッケージの変更が行われる方法

次の手順は、pkg が呼び出されて、イメージにインストールされているソフトウェアを変更するときに実行されます。

- 入力にエラーがないか確認する
- システムの終了状態を決定する
- 基本的なチェックを実行する
- ソルバーを実行する
- ソルバーの結果を最適化する
- アクションを評価する
- 内容をダウンロードする
- アクションを実行する
- アクチュエータを処理する

これらの手順を大域ゾーンで実行するとき、pkg はシステム上のどの非大域ゾーンでも動作できます。たとえば、pkg は大域ゾーンと非大域ゾーンの間の依存関係が正しいことを確認し、内容をダウンロードして、非大域ゾーンに必要なアクションを実行します。ゾーンについては、第10章「非大域ゾーンの処理」で詳しく説明しています。

入力にエラーがないか確認する

コマンド行に指定されたオプションについて基本的なエラーチェックが行われます。

システムの終了状態を決定する

システムの望ましい終了状態の記述が作成されます。イメージ内のすべてのパッケージを更新する場合、望ましい終了状態は「現在インストールされているすべてのパッケージ、またはそれらの新しいバージョン」のようなものになる可能性があります。パッケージを削除する場合、望ましい終了状態は「これを除く現在インストールされているすべてのパッケージ」となります。

IPS では、ユーザーがこの終了状態をどのようなものであると意図しているかを判断しようとします。場合によっては、IPS では終了状態がユーザーのリクエストしたものと一致していたとしても、その終了状態をユーザーの意図したものではないと判断することがあります。

トラブルシューティングの際は、できるだけ具体的にすることが最善です。次のコマンドは具体的ではありません。

```
$ pkg update
```

このコマンドが「このイメージに使用できる更新はありません」といったメッセージで失敗する場合は、次のコマンドのようなより具体的なコマンドを試すことをお勧めします。

```
$ pkg update "*@latest"
```

このコマンドでは、終了状態をより正確に定義しているため、より的を射たエラーメッセージが生成されます。

基本的なチェックを実行する

ソリューションが可能であることを確認するために、システムの望ましい終了状態がレビューされます。この基本的なレビュー中に、pkg はすべての依存関係の妥当なバージョンが存在すること、および目的のパッケージがお互いを除外していないことを確認します。

明らかなエラーが存在する場合、pkg は適切なエラーメッセージを出力して終了します。

ソルバーを実行する

ソルバーは、イメージ内の制約およびインストール用の新しいパッケージによって導入された制約を考慮に入れてインストール、更新、または削除できるパッケージを判断するために pkg(5) によって使用される演算処理エンジンの中核を成しています。

この問題は、ブール型の充足可能性問題の一例であり、SAT ソルバーによって解決できます。

すべてのパッケージの可能性のあるさまざまな選択肢がブール型変数に割り当てられ、それらのパッケージ間のすべての依存関係や必須パッケージなどが論理積正規形のブール式としてキャストされます。

生成された一連の式は MiniSAT に渡されます。MiniSAT がソリューションを 1 つも見つけられない場合は、エラー処理コードが、インストールされた一連のパッケージと試された操作を調べて、可能性のあるそれぞれの選択肢が除外された理由を出力します。

現在インストールされているパッケージのセットは要件を満たしており、ほかに満たしているセットがない場合、pkg は何もすることがないことを報告します。

前述のように、エラーメッセージの生成と具体性は pkg への入力によって決まります。pkg に発行されるコマンド内をできるだけ具体的にすると、もっとも役立つエラーメッセージが生成されます。

MiniSAT が可能性のあるソリューションを見つけた場合は、最適化フェーズが開始されます。

ソルバーの結果を最適化する

SAT ソルバーに対し、あるソリューションがほかのソリューションよりも望ましいと記述する方法はないため、最適化フェーズが必要になります。その代わりに、ソリューションが見つかると、IPS はその問題に制約を加えてあまり望ましくない選択肢を分離し、さらに現在のソリューションも分離します。その後、IPS は繰り返し MiniSAT を呼び出し、ソリューションがそれ以上見つからなくなるまで上記の操作を繰り返します。最後に成功したソリューションが最良のものとみなされます。

ソリューションを見つける難しさは、可能性のあるソリューションの数に比例します。望ましい結果についてより具体的にすると、ソリューションがより迅速に見つかります。

発生した問題をもっともよく満たす一連のパッケージ FMRI が見つかると、評価フェーズが開始されます。

アクションを評価する

評価フェーズでは、IPS はシステムに現在インストールされている、終了状態を含むパッケージを比較し、さらに古いパッケージと新しいパッケージのパッケージマニフェストを比較して、3 つのリストを調べます。

- 削除されているアクション。
- 追加されているアクション。
- 更新されているアクション。

その後、このアクションリストは次の方法で更新されます。

- ディレクトリアクションとリンクアクションは参照カウントの対象であるため、調停されたリンクの処理が行われます。
- ハードリンクは、それらのターゲットファイルが更新されると、修復のためにマーク付けされます。これが行われるのは、現在実行中のプロセスにとって安全な方法でハードリンクのターゲットを更新すると、そのハードリンクが壊れてしまうからです。
- パッケージ間を移動する編集可能なファイルは、ユーザーによる編集が失われないように適切に処理されます。
- アクションリストは、削除、追加、および更新が正しい順序で表示されるようにソートされます。

その後、現在インストールされているパッケージが、パッケージの競合がないことを確認するために相互にチェックされます。競合の例には、ファイルを同じ場所に提供する 2 つのパッケージや、異なるディレクトリ属性を持つ同じディレクトリを配布する 2 つのパッケージなどがあります。

競合が存在する場合、その競合は報告され、pkg はエラーメッセージを出して終了します。

最後に、アクションリストがスキャンされて、この操作が行われる場合に SMF サービスを再起動する必要があるかどうか、この変更を実行中のシステムに適用できるかどうか、ブートアーカイブを再構築する必要があるかどうか、そして必要な容量が使用可能であるかどうかが判断されます。

内容をダウンロードする

pkg コマンドを -n フラグなしで実行している場合、処理はダウンロードフェーズへ進みます。

内容を必要とするアクションごとに、IPS は必要なすべてのファイルをハッシュによってダウンロードし、それらをキャッシュに入れます。取り込む内容の量が大きい場合は、この手順にしばらく時間がかかることがあります。

ダウンロードが完了したあと、ライブシステム (/ をルートに持つイメージ) に変更が適用され、さらにリブートが必要とされる場合は、実行中のシステムのクローンが作成され、ターゲットイメージはそのクローンに切り替えられます。

アクションを実行する

アクションの実行では、実際に各アクションタイプに固有のインストールメソッドまたは削除メソッドをイメージに対して実行する必要があります。

実行は、すべての削除アクションの実行から始まります。システムから削除しているディレクトリで予期しない内容が見つかった場合、その内容は `/var/pkg/lost+found` に入れられます。

次に、実行はインストールおよび更新アクションに進みます。すべてのアクションがすべてのパッケージにわたって混在していることに注意してください。したがって、1 つのパッケージ操作での変更はパッケージごとではなく、システムに同時に適用されます。これにより、パッケージが互いに依存し合ったり、内容を安全に交換したりすることが可能になります。ファイルがどのように更新されるかについての詳細は、26 ページの「ファイルアクション」を参照してください。

アクチュエータを処理する

変更がライブシステムに適用されている場合、この時点で保留中のアクチュエータがすべて実行されます。これらは一般に SMF サービスの再起動とリフレッシュです。これらが起動されると、IPS はローカルの検索インデックスを更新します。アクチュエータについては、第7章「パッケージインストールの一環としてのシステム変更の自動化」で詳しく説明しています。

ブートアーカイブを更新する

必要に応じて、ブートアーカイブが更新されます。

♦ ♦ ♦ 第 4 章

パッケージ依存関係の指定

依存関係は、パッケージがどのように関連しているかを定義します。この章では、次について説明します。

- パッケージの依存関係のタイプ。依存関係のタイプは35 ページの「依存アクション」で紹介されました。この章では、さらに詳細な情報を提供します。
- ソフトウェアインストールを制御するために、各依存関係のタイプを使用する方法。作業用ソフトウェアシステムを構築するための依存関係と凍結の使用方法。

依存関係の種類

IPS では、パッケージのすべて依存関係が満たされないかぎり、パッケージをインストールすることはできません。IPS では、パッケージは相互に依存し合う (循環型依存関係を持つ) ことができます。IPS では、パッケージは同時に同じパッケージへのさまざまな種類の依存関係を持つこともできます。

この章の各セクションには、`depend` アクションがパッケージの作成中にマニフェスト内に指定されているときの例が含まれています。依存関係の指定は、パブリッシャーの名前を含みません。可能性のある競合については、99 ページの「パッケージ内容の競合の回避」を参照してください。

`require` 依存関係

依存関係のもっとも基本的なタイプは `require` 依存関係です。これらの依存関係は通常、ライブラリやインタプリタ (Python や Perl など) などの機能依存性を表現するために使用します。

パッケージ `pkg-a@1.0` にパッケージ `pkg-b@2` への `require` 依存関係が含まれている場合、`pkg-a@1.0` をインストールしたら、バージョン 2 以上の `pkg-b` パッケージもインストールする必要があります。この上位バージョンのパッケージの受け入れは、既存のパッケージのより新しいバージョンにバイナリ互換性があることへの暗黙の期待を反映しています。

depend アクションに指定されたパッケージのどのバージョンも受け入れ可能である場合は、指定された FMRI のバージョン部分を省略できます。

require 依存関係の例は次のとおりです。

```
depend fmri=pkg:/system/library type=require
```

require-any 依存関係

複数の fmri 属性によって指定される複数のターゲットパッケージのいずれか 1 つが依存関係を満たす場合、require-any 依存関係が使用されます。依存関係がまだ満たされていない場合、IPS はそれらのパッケージのいずれかを選んでインストールします。

たとえば、require-any 依存関係を使用して、Perl の少なくとも 1 つのバージョンがシステムに確実にインストールされるようできます。バージョン管理は、require 依存関係の場合と同じように処理されます。

require-any 依存関係にパッケージが一覧表示される順序は、依存関係を満たすために選択されるパッケージに影響しません。たとえば、先頭に一覧表示されているパッケージがほかのどれよりも優先されるわけではありません。例外は、すでにインストールされており、pkg 操作のほかの部分によって削除されないパッケージは、まだインストールされていないパッケージより優先されることです。

require-any 依存関係の例は次のとおりです。

```
depend type=require-any fmri=pkg:/editor/gnu-emacs/gnu-emacs-gtk \
    fmri=pkg:/editor/gnu-emacs/gnu-emacs-no-x11 \
    fmri=pkg:/editor/gnu-emacs/gnu-emacs-x11
```

optional 依存関係

optional 依存関係は、指定されたパッケージをインストールする場合に、それが指定のバージョン以上である必要があることを指定します。

このタイプの依存関係は通常、パッケージが内容を転送する場合に対処するために使用します。この場合、転送後のパッケージの各バージョンには、他方のパッケージの転送後のバージョンへの optional 依存関係が含まれるため、2 つのパッケージの互換のないバージョンをインストールすることはできません。optional 依存関係でバージョンを省略すると、その依存関係は意味がなくなりますが、許可されます。

optional 依存関係の例は次のとおりです。

```
depend fmri=pkg:/x11/server/xorg@1.9.99 type=optional
```

conditional 依存関係

conditional 依存関係には、predicate 属性と fmri 属性があります。predicate 属性の値に指定されたパッケージが、指定されたバージョン以上でシステムに存在する場合、conditional 依存関係は fmri 属性のパッケージへの require 依存関係として扱われます。predicate 属性で指定したパッケージがシステムに存在しないか、下位バージョンで存在する場合、conditional 依存関係は無視されます。

conditional 依存関係は、必須の基本パッケージがシステムに存在する場合にオプションの拡張機能をパッケージにインストールするためにもっともよく使用します。

たとえば、X11 バージョンと端末バージョンの両方を持つエディタパッケージは、X11 バージョンを別個のパッケージに入れ、必須の X クライアントライブラリパッケージが predicate として存在するテキストバージョンから X11 バージョンへの conditional 依存関係を含めることができます。

次の例の conditional 依存関係では、指定されたパッケージがすでに十分にバージョン制約されているため、パッケージバージョン番号は必要ありません。

```
depend fmri=library/python/pycurl-27 predicate=runtime/python-27 type=conditional
```

group 依存関係

group 依存関係は、パッケージのグループを構築するために使用します。

group 依存関係は指定されたバージョンを無視します。指定されたパッケージのどのバージョンもこの依存関係を満たします。

指定されたパッケージは、そのパッケージが次のいずれかの操作の対象になっていないかぎり必要です。

- パッケージが回避リスト上に置かれました。回避リストについては、pkg(1) のマニュアルページを参照してください。
- fmri 値に一致するパッケージが不明です。
- パッケージが pkg install --reject で拒否されました。
- パッケージが pkg uninstall でアンインストールされました。

これらの 3 つのオプションを使用すると、管理者は group 依存関係の対象になっているパッケージを選択解除できます。これらの 3 つのオプションのいずれかが使用された場合、IPS ではパッケージが別の依存関係で引き続き必要とされた場合を除き、更新中にそのパッケージを再インストールしません。新しい依存関係がそれに続く別の操作で削除された場合、そのパッケージは再度アンインストールされます。

廃止されたパッケージは、暗黙のうちにグループの依存関係を満たし、実質的に依存関係を無視します。

これらの依存関係の使用方法のよい例は、システムの通常の使用に必要とされるパッケージへの group 依存関係を含むパッケージを構築することです。いくつかの例として、solaris-large-server 、solaris-desktop、または developer-gnu があげられます。148 ページの「Oracle Solaris グループパッケージ」には、group 依存関係を配布する一連の Oracle Solaris パッケージが示されています。

グループパッケージをインストールすると、この OS の新しいバージョンへの以降の更新のすべてにわたって、適切なパッケージがシステムに確実に追加されるようになります。

group 依存関係の例は次のとおりです。

```
depend fmri=package/pkg type=group
```

group-any 依存関係

複数の fmri 属性によって指定される複数のターゲットパッケージのいずれか 1 つが依存関係を満たす場合、group-any 依存関係が使用されます。依存関係がまだ満たされていない場合、IPS はそれらのパッケージのいずれかを選んでインストールします。廃止されていないパッケージシステムが廃止されたパッケージシステムよりも優先されるという点を除き、group 依存関係に適用されるものと同じ規則が group-any 依存関係に対して適用されます。

group-any 依存関係にパッケージが一覧表示される順序は、依存関係を満たすために選択されるパッケージに影響しません。たとえば、先頭に一覧表示されているパッケージがほかのどれよりも優先されるわけではありません。次の選択された設定は存在しません。

- すでにインストールされており、pkg 操作のほかの部分によって削除されないパッケージは、まだインストールされていないパッケージより優先されます。
- 一部のターゲットパッケージが回避された場合、依存関係を満たすために別のターゲットを使用します。すべてのターゲットパッケージが回避された場合、依存関係は無視されます。
- 一部のターゲットパッケージが廃止された場合、依存関係を満たすために別のターゲットを使用します。すべてのターゲットパッケージが廃止された場合、依存関係は無視されます。

group 依存関係の例は次のとおりです。

```
depend type=group-any \
    fmri=runtime/python-26 \
    fmri=runtime/python-27
```

origin 依存関係

origin 依存関係は、中間移行が必要となるアップグレード問題を解決するために存在します。デフォルトの動作では、更新中のイメージに存在している必要のある最小バージョンのパッケージ (インストール済みの場合) を指定します。root-image 属性の値が true である場合、このパッケージをインストールするには、/をルートとするイメージ上にパッケージが存在する必要があります。

たとえば、典型的な使用法として、データベースパッケージのバージョン 5 で、バージョン 3 以上からのアップグレードをサポートし、それより前のバージョンからのアップグレードはサポートしない場合があげられます。この場合、バージョン 5 にはバージョン 3 でのそれ自身への origin 依存関係があります。このため、バージョン 5 が新たにインストールされた場合、インストールは続行します。ただし、バージョン 1 のパッケージがインストールされていた場合、そのパッケージをバージョン 5 に直接アップグレードすることはできません。この場合、pkg update database-package はバージョン 5 を選択せず、アップグレードする可能な最新バージョンとして代わりにバージョン 3 を選択します。

root-image 属性の値が true の場合、依存関係のターゲットは、更新中のイメージではなく実行中のシステムに存在する場合には、指定されたバージョン以上でなければなりません。この形式の origin 依存関係は通常、ブートブロックインストーラへの依存関係など、オペレーティングシステムの問題に使用されます。

origin 依存関係の例は次のとおりです。

```
depend fmri=pkg:/database/mydb@3.0 type=origin
```

ファームウェアが手動で保守されるデバイスドライバ

デバイスドライバはそのファームウェアを管理する必要があります。ファームウェアはドライバパッケージで配布され、管理者が pkg update コマンドを使用してドライバを更新するときに更新される必要があります。ドライバの設計については、『Oracle Solaris 11.2 デバイスドライバの記述』 の 「ファームウェア互換性」を参照してください。また、一部の新機能がサポートされていない場合でも、ドライバは引き続き downrev ファームウェアを使用して機能する必要があります。

いくつかのドライバでは、pkg update の実行によるドライバの更新とは別にデバイスファームウェアを更新するために手動での介入が必要です。ファームウェアが手動で保守されるドライバのいくつかは、古いバージョンのファームウェアと互換性がなく、ファームウェアに関する最小バージョン要件があります。現在インストールされているファームウェアとの互換性がないドライバのインストールを防ぐには、origin 依存関係を使用できます。このようなドライバのインストールによりシステムアップグレードが行われず、その結果システムが部分的に機能しなくなることがあります。

origin 依存関係を使用すると、配布されるドライバのバージョンと互換性があるデバイスファームウェアの最小バージョンを指定できます。root-image 属性の値が true

で、fmri 属性の値が pkg:/feature/firmware/ で始まる場合、fmri 値の残りの部分は、ファームウェアの依存関係を評価する /usr/lib/fwenum 内のコマンドとして扱われます。このタイプの依存関係を指定するパッケージを管理者が更新し、ファームウェア列挙子によってファームウェア依存関係が満たされていないと判断される場合、エラーメッセージが表示され、更新は実行されず、システムは変更されません。エラーメッセージには、このドライバにより管理されるデバイスに必要なファームウェアバージョンが表示されます。ファームウェアが更新されたら、管理者は pkg update を再試行できます。

次に、最小ファームウェアバージョン要件のある origin 依存関係の例を示します。

```
depend fmri=pkg:/feature/firmware/mpt_sas minimum-version=1.0.0.0 \
root-image=true type=origin variant.opensolaris.zone=global
```

pkg クライアントは、次の例に示すようにファームウェア列挙子を呼び出します。

```
/usr/lib/fwenum/mpt_sas minimum-version=1.0.0.0
```

次に示す pkg クライアントからのメッセージの例は、管理者に対し、mpt_sas ドライバが管理する 2 つのデバイスに、そのバージョンが最小要件を満たしていないファームウェアがあることを示しています。このメッセージには、ファームウェアの必要最小バージョン要件も示されます。

```
There are 2 instances of downrev firmware for the mpt_sas devices present on this system;
upgrade each to version 1.0.0.0 or greater to permit installation of this version of
 Solaris.
```

ドライバで複数ベンダーからの同じデバイスがサポートされている場合、依存関係には minimum-version 属性に加えて vendor 属性も指定できます。

incorporate 依存関係

incorporate 依存関係は、指定されたパッケージがインストールされる場合に、それが指定のバージョン (指定のバージョン精度) である必要があることを指定します。たとえば、依存する FMRI にバージョン 1.4.3 が含まれている場合、1.4.3 未満のバージョンも 1.4.4 以上のバージョンもこの依存関係を満たしません。バージョン 1.4.3.7 はこのサンプル依存関係を満たします。

incorporate 依存関係の一般的な使用方法は、それらの多くを同じパッケージに入れて、互換性のあるサーフェスをパッケージバージョン領域に定義することです。そのような incorporate 依存関係セットを含むパッケージは、しばしば結合と呼ばれます。結合は通常、同時に作成されるソフトウェアパッケージのセットを定義するために使用され、個別にバージョン管理されません。incorporate 依存関係は、ソフトウェアの互換性のあるバージョンがまとめてインストールされるようにするために Oracle Solaris でよく使用されます。

incorporate 依存関係の例は次のとおりです。

```
depend type=incorporate \
    fmri=pkg:/driver/network/ethernet/e1000g@0.5.11,5.11-0.175.0.0.0.2.1
```

parent 依存関係

parent 依存関係は、ゾーンまたはほかの子イメージに使用します。この場合、その依存関係は子イメージでのみチェックされ、親イメージまたは大域ゾーンに存在している必要のあるパッケージとバージョンを指定します。指定されたバージョンは、指定されたレベルの精度に一致する必要があります。

たとえば、親の依存関係が A@2.1 にある場合、2.1 から始まる A のすべてのバージョンが一致します。この依存関係は、パッケージが非大域ゾーンと大域ゾーンの間で同期した状態に保たれる必要があるときによく使用します。ショートカットとして、この依存関係を含むパッケージの正確なバージョンの同義語として特別なパッケージ名 feature/package/dependency/self が使用されます。

parent 依存関係は、非大域ゾーンにインストールされた主要なオペレーティングシステムコンポーネント (libc.so.1 など) を、大域ゾーンにインストールされたカーネルと同期した状態に維持するために使用されます。parent 依存関係については、第10章「非大域ゾーンの処理」でも説明しています。

parent 依存関係の例は次のとおりです。

```
depend type=parent fmri=feature/package/dependency/self \
    variant.opensolaris.zone=nonglobal
```

exclude 依存関係

exclude 依存関係を含むパッケージは、依存するパッケージが指定されたバージョンレベル以上でイメージにインストールされる場合はインストールできません。

exclude 依存関係の FMRI からバージョンを省いた場合、依存関係を指定するパッケージと同時にインストールできる、除外されたパッケージのバージョンはありません。

exclude 依存関係はほとんど使用されません。これらの制約は、管理者の負担になることがあるため、可能であれば回避するようにしてください。

exclude 依存関係の例は次のとおりです。

```
depend fmri=pkg:/x11/server/xorg@1.10.99 type=exclude
```

制約と凍結

前述の依存関係タイプを慎重に使用することによって、パッケージのアップグレードを許可する方法を制約できます。

- incorporate 依存関係により、まとめて更新するサポートされているソフトウェアサーフェスを定義できます。
- 凍結により、管理者はサーフェスやその他のソフトウェアを特定のバージョンに維持できます。
- バージョンロックファセットにより、管理者はサーフェスの一部のコンポーネントに対するバージョン制約を無効にできます。

インストール可能なパッケージバージョンの制約

一般に、システムにインストールされた一連のパッケージのサポートやアップグレードはまとめて行われるのが望ましいことです (セット内のパッケージがすべて更新されるか、セット内のパッケージがどれも更新されないかのどちらか)。パッケージをセットとしてこのように扱うには、incorporate 依存関係を使用します。

『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「カスタム Incorporation のインストール」に、インストール可能な pkg:/entire incorporation のバージョンを制約するためのカスタム incorporation の作成例を示します。このセクションの以降の部分では、結合についての一般的な説明を示します。

次の 3 つの部分的なパッケージマニフェストは、pkg-a および pkg-b パッケージと myincorp 結合パッケージの関係を示しています。

次に、pkg-a パッケージマニフェストからの抜粋を示します。

```
set name=pkg.fmri value=pkg-a@1.0
dir path=opt/tool-a owner=root group=bin mode=0755
depend fmri=myincorp type=require
```

次に、pkg-b パッケージマニフェストからの抜粋を示します。

```
set name=pkg.fmri value=pkg-b@1.0
dir path=opt/tool-b owner=root group=bin mode=0755
depend fmri=myincorp type=require
```

次に、myincorp パッケージマニフェストからの抜粋を示します。

```
set name=pkg.fmri value=myincorp@1.0
depend fmri=pkg-a@1.0 type=incorporate
depend fmri=pkg-b@1.0 type=incorporate
```

pkg-a および pkg-b パッケージの両方に、myincorp 結合への require 依存関係が含まれています。myincorp パッケージには、次の方法で pkg-a および pkg-b パッケージを制約する incorporate 依存関係が含まれています。

- pkg-a および pkg-b パッケージは、最大でもバージョン 1.0 (依存関係に指定されたバージョン番号で定義された精度レベル) にしかアップグレードできません。
- pkg-a および pkg-b パッケージをインストールする場合、それらはバージョン 1.0 以上である必要があります。

バージョン 1.0 への `incorporate` 依存関係では、バージョン 1.0.1 や 1.0.2.1 などは許可されますが、バージョン 1.1、2.0、0.9 などは許可されません。`incorporate` 依存関係を上位バージョンで指定する、更新済みの結合パッケージがインストールされると、`pkg-a` および `pkg-b` パッケージはその上位バージョンに更新することを許可されます。

`pkg-a` および `pkg-b` パッケージの両方に、`myincorp` パッケージへの `require` 依存関係が含まれているため、`pkg-a` または `pkg-b` のいずれかがインストールされている場合に結合パッケージがインストールされます。

インストール可能なパッケージバージョンの凍結

前のセクションで、パッケージマニフェストを変更することでパッケージのオーサリングプロセス中に適用される制約について説明しました。管理者は、実行時に制約をシステムに適用することもできます。

`pkg freeze` コマンドを使用すると、管理者はある特定のパッケージが、現在インストールされているバージョン (タイムスタンプを含む)、またはコマンド行で指定したバージョンから変更されるのを防ぐことができます。この機能は事実上 `incorporate` 依存関係と同じです。

`freeze` コマンドの詳細については 『Oracle Solaris 11.3 ソフトウェアの追加と更新』および `pkg`(1) のマニュアルページを参照してください。

さらに複雑な依存関係をイメージに適用するには、それらの依存関係を含むパッケージを作成してインストールします。

管理者がインストール可能なパッケージバージョンへの制約を緩和できるようにする

管理者は、依存関係のバージョン制約を無効にしなければならない場合があります。`version-lock` ファセットタグを提供して、管理者がそれらのタグ付き `incorporate` 依存関係を無効にできるようにします。管理者は、`pkg change-facet` コマンドを使用して、対応するファセットイメージプロパティーの値を `false` に設定できます。ファセットタグに関する一般的な情報については、第5章「バリエーションの許可」を参照してください。

前の例を続けると、おそらく `pkg-b` は `pkg-a` と関係なく機能できますが、`pkg-a` を、結合の `incorporate` 依存関係によって定義された一連のバージョン内にとどめておく必要があります。`myincorp` パッケージマニフェストには、`pkg-b` 依存関係の `version-lock` ファセットタグなど、次の行を含めることができます。慣例的に、`version-lock` ファセットタグには `facet.version-lock.`*package-name* という名

前が付けられます。ここで *package-name* は、その depend アクションの fmri に指定された、バージョンのない名前です。

```
set name=pkg.fmri value=myincorp@1.0
depend fmri=pkg-a@1.0 type=incorporate
depend fmri=pkg-b@1.0 type=incorporate facet.version-lock.pkg-b=true
```

デフォルトで、この結合には pkg-b パッケージへの depend アクションが含まれており、pkg-b をバージョン 1.0 に制約しています。この制約は、次のコマンドによって緩和されます。

```
$ pkg change-facet version-lock.pkg-b=false
```

このコマンドの実行が成功したあとで、pkg-b パッケージは結合の制約から解放され、必要に応じて上位のバージョンにアップグレードできます。

次の例では、この結合で、java-8-incorporation パッケージをインストールする必要があり、それがバージョン 1.8.0.92.14-0 である必要があることを指定しています。ただし、指定された facet.version-lock ファセットにより、管理者は別のバージョンのインストールを試みることもできます。

```
depend fmri=consolidation/java-8/java-8-incorporation type=require
depend facet.version-lock.consolidation/java-8/java-8-incorporation=true \
    fmri=consolidation/java-8/java-8-incorporation@1.8.0.92.14-0 type=incorporate
```

おそらく上位バージョンの java-8-incorporation パッケージもこの結合で動作します。管理者は、pkg change-facet コマンドを使用して、version-lock.consolidation/java-8/java-8-incorporation ファセットプロパティーを false に設定し、結合の更新と別に、java-8-incorporation パッケージの更新を試みることができます。

♦ ♦ ♦ 第 5 章

5 バリエーションの許可

この章では、各種インストールオプションをエンドユーザーに提供する方法について説明します。相互に排他的なコンポーネントのインストールはバリアントによって制御し、オプションのコンポーネントのインストールはファセットによって制御します。

バリアントとファセットのどちらも次の 2 つのコンポーネントで構成されます。

- パッケージマニフェスト内のアクションに設定されたタグ
- イメージに設定されたプロパティー

『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「オプションのコンポーネントのインストールの制御」では、バリアントとファセットがインストールに与える影響と管理者がイメージ内のバリアントとファセットのプロパティーの値を変更する方法について説明しています。

相互に排他的なソフトウェアコンポーネント

バリアントは、パッケージ内の次の 2 つの場所に出現します。

- `set` アクションはバリアントの名前を指定し、このパッケージに適用される値を定義します。
- `set` アクションに指定されたバリアント値のサブセットに対してのみインストールできるアクションにはどれも、そのバリアントの名前と、このアクションがインストールされるときの値を指定するタグが含まれています。

相互に排他的なコンポーネントの 1 つの例はシステムアーキテクチャーです。アクションには、さまざまなバリアント名の複数のタグを含めることができます。たとえば、1 つのパッケージに SPARC と x86 の両方のデバッグバイナリと非デバッグバイナリの両方を含めることができます。

バリアントには、その名前と使用可能な値のリストの 2 つの部分があります。`pkg variant -v` コマンドは、インストール済みパッケージに設定できるすべてのバリアント値を表示します。

```
$ pkg variant -v
VARIANT                 VALUE
arch                    i386
arch                    sparc
debug.osnet             false
debug.osnet             true
opensolaris.zone        global
opensolaris.zone        nonglobal
```

IPS は、異なるアーキテクチャー用のアクションに異なるバリアントタグ値を指定することによって、単一パッケージ内で複数のアーキテクチャーをサポートします。バリアントタグは、アーキテクチャー間で異なるすべてのアクションに適用されます。SPARC および x86 の両方に提供されるコンポーネントはバリアントタグを受け取りません。たとえば、シンボリックリンク `/var/ld/64` を配布するパッケージには次の定義を含めることができます。

```
set name=variant.arch value=sparc value=i386
dir group=bin mode=0755 owner=root path=var/ld
dir group=bin mode=0755 owner=root path=var/ld/amd64 variant.arch=i386
dir group=bin mode=0755 owner=root path=var/ld/sparcv9 variant.arch=sparc
link path=var/ld/32 target=.
link path=var/ld/64 target=sparcv9 variant.arch=sparc
link path=var/ld/64 target=amd64 variant.arch=i386
```

相互に排他的なもう 1 つのコンポーネントは大域ゾーンまたは非大域ゾーンです。カーネルコンポーネントは通常は非大域ゾーンに含まれません。カーネルコンポーネントが非大域ゾーンにインストールされるのを防ぐために、`opensolaris.zone` バリアントタグを値をそれらの各アクションに適用し、値を `global` に設定します。

`pkgmogrify` 規則を使用して、発行中にマニフェストにバリアントタグを適用します。`pkgmogrify` コマンドの使用方法については、第6章「プログラムによるパッケージマニフェストの変更」で詳しく説明しています。次に、`pkgmerge` を使用して SPARC および x86 ビルドからのパッケージをマージします。`pkgmerge` コマンドは、必要に応じて、同時に複数の異なるバリアントにわたってマージします。詳細は、`pkgmogrify(1)` および `pkgmerge(1)` のマニュアルページを参照してください。

不明なバリアントプロパティー値は、デフォルトでイメージ内で `false` です。したがって、新しいバリアントタグ名を導入する場合、そのバリアントの値は `true` または `false` のみに設定できます。アクションのバリアントタグを `true` に設定し、そのアクションをインストールするために、管理者に `pkg change-variant` コマンドを使用してイメージ内のバリアントプロパティーの値を `true` に変更するよう通知します。

名前が `variant.debug.` で始まるバリアントは、デフォルトでイメージ内で `false` です。ユーザーは、コンポーネントのデバッグバージョンを用意し、それらのコンポーネントにカスタムの `variant.debug.` バリアントをタグ付けできます。

注記 - バリアントはイメージごとに設定されます。そのソフトウェアの適切な解決に固有のバリアント名を選択してください。

オプションのソフトウェアコンポーネント

本体に含まれているソフトウェアの一部はオプションであり、一部のユーザーはそれらをインストールする必要がない場合があります。例として、異なるロケール用のローカライズファイル、マニュアルページとその他のドキュメント、開発者または DTrace ユーザーにのみ必要なヘッダーファイルなどがあげられます。

従来、オプションの内容は別個のパッケージで配布されていました。管理者は、これらのオプションのパッケージをインストールすることでオプションの内容をインストールしました。この解決方法の問題の 1 つは、管理者が使用可能なパッケージのリストを調べて、オプションのパッケージを見つけ出し、インストールする必要があることです。

IPS はファセットを使用して、オプションのパッケージ内容を配布します。ファセットはバリアントに似ています: 各ファセットには名前と値があり、アクションにはさまざまなファセット名の複数のタグを含めることができます。

- イメージ内で `facet.debug.` または `facet.optional.` で始まるすべてのファセットプロパティーのデフォルト値は、`false` です。その他のすべてのファセットプロパティーのデフォルト値は `true` です。
- パッケージ化されたアクションで、ファセットタグの値は、`true` または `all` のいずれかとして指定できます。

  `all` の値を持つファセットタグのあるアクションは、`all` の値を持つそのアクションのいずれかのファセットが、イメージ内に `true` の値を持つ場合のみインストールされます。

  `true` の値を持つファセットタグのあるアクションは、`true` の値を持つそのアクションのいずれかのファセットが、イメージ内に `true` の値を持つ場合のみインストールされます。

正規表現を使用してさまざまなタイプのファイルを一致させることで、ファセットタグをパッケージマニフェストに追加するには、`pkgmogrify` コマンドを使用します。`pkgmogrify` コマンドの使用方法については、第6章「プログラムによるパッケージマニフェストの変更」で詳しく説明しています。

パッケージ化されたアクションに設定するために使用可能なファセットのリストを生成するには、`pkg facet -a` コマンドを使用して、イメージに明示的に設定されたすべてのファセットとインストール済みパッケージに設定されたすべてのファセットの値を表示します。

ソフトウェアに新しいファセットを導入する場合、そのアクションがインストールされるかどうかは、ファセットに選択した名前と、そのアクションに設定されているほかのファセットによって異なります。新しいファセットがアクションに設定された唯一のファセットである場合、アクションは次のようにインストールされます。

- ファセットに `facet.debug.`*mysoftware* または `facet.optional.`*mycomponent* という名前を付けた場合、ユーザーがファセットイメージプロパティーの値を `true` に設定している場合にのみアクションがインストールされます。
- 新しいファセットにほかの任意の名前を選択した場合は、ユーザーがファセットイメージプロパティーの値を `false` に設定していないかぎり、アクションがインストールされます。

オプションのコンポーネントの指定に加えて、77 ページの「制約と凍結」に説明するように、`facet.version-lock.` ファセットタグを使用して、依存関係バージョン制限を指定できます。

♦ ♦ ♦ 第 6 章

6 プログラムによるパッケージマニフェストの変更

この章では、自動的に注釈が付けられチェックされるように、パッケージマニフェストをプログラムによって編集する方法について説明します。

第2章「IPS を使用したソフトウェアのパッケージ化」では、パッケージの発行に必要なすべての手法が取り上げられています。この章では、次のタスクの実行に役立つ追加情報を提供します。

- 大きいパッケージの発行
- 大量のパッケージの発行
- 一定期間にわたるパッケージの再発行

パッケージには、第5章「バリエーションの許可」で説明しているようにバリアントまたはファセットでタグ付けする必要がある多数のアクション、または第7章「パッケージインストールの一環としてのシステム変更の自動化」で説明しているようにサービスの再起動によってタグ付けする必要がある多数のアクションが含まれている可能性があります。手動でパッケージマニフェストを編集したり、この作業を行うスクリプトやプログラムを作成したりするのではなく、IPS の `pkgmogrify` ユーティリティーを使用して、パッケージマニフェストの変換を迅速かつ正確に、そして繰り返し行います。

`pkgmogrify` ユーティリティーは次の 2 種類の規則を適用します。

- 変換規則によってアクションが変更されます。
- 取り込み規則によって他のファイルが処理されます。

`pkgmogrify` ユーティリティーは、これらの規則をファイルから読み取り、それらを指定されたパッケージマニフェストに適用します。

変換規則

このセクションでは、変換規則の例を示し、すべての変換規則の各部について説明します。

Oracle Solaris では、kernel という名前のサブディレクトリに配布するファイルはカーネルモジュールとして扱われ、リブートが必要としてタグ付けされます。次のタグは、path 属性値に kernel が含まれるアクションに適用されます。

```
reboot-needed=true
```

このタグを適用するには、次の規則を pkgmogrify 規則ファイルに指定します。

```
<transform file path=.*kernel/.+ -> default reboot-needed true>
```

区切り記号	この規則は < と > で囲む必要があります。その規則の -> の左側の部分は、選択セクションまたは照合セクションです。-> の右側の部分は、その操作の実行セクションです。
transform	この規則のタイプ。
file	この規則を file アクションにのみ適用します。これは、その規則の選択セクションと呼ばれます。
path=.*kernel/.+	正規表現 path=.*kernel/.+ に一致する path 属性を持つ file アクションのみを変換します。これは、その規則の照合セクションと呼ばれます。
default	default に続く属性と値を、まだその属性に値が設定されていない一致するアクションに追加します。
reboot-needed	設定される属性。
true	設定される属性の値。

変換規則の選択セクションまたは照合セクションは、アクションタイプによって、またアクション属性値によって制限されることがあります。これらの一致規則のしくみについての詳細は、pkgmogrify のマニュアルページを参照してください。通常は、ファイルシステムの指定された領域に配布するアクションを選択するために使用します。たとえば、次の規則では、*operation* は usr/bin と usr/bin 内で配布されたすべてのものがデフォルトで適切なユーザーまたはグループになるようにするために使用できます。

```
<transform file dir link hardlink path=usr/bin.* -> operation>
```

pkgmogrify(1) のマニュアルページでは、pkgmogrify がアクション属性の追加、削除、設定、編集、およびアクション全体の追加と削除を実行できる多数の操作について説明されています。

取り込み規則

取り込み規則では、変換をさまざまなマニフェストによって再利用される複数のファイルおよびサブセットに広げることができます。2 つのパッケージ A と B を配布する

必要があるとします。両方のパッケージの source-url は同じ URL に設定されていますが、/etc 内のパッケージ B のファイルのみが group=sys になるよう設定されています。

パッケージ A のマニフェストでは、source-url 変換を含むファイルを引き入れる取り込み規則を指定します。パッケージ B のマニフェストでは、ファイルグループ設定変換を含むファイルを引き入れる取り込み規則を指定します。最後に、source-url 変換を含むファイルを引き入れる取り込み規則を、パッケージ B、またはグループを設定する変換を含むファイルのどちらかに追加します。

変換順序

変換は、ファイル内でそれらが検出される順序で適用されます。この順序付けを使用すると、変換の一致する部分を簡略化できます。

/foo に配布されるすべてのファイルにはデフォルトの sys グループが含まれているが、/foo/bar に配布されるファイルにはデフォルトの bin グループが含まれているとします。

/foo/bar で始まるパスを除き、/foo で始まるすべてのパスに一致する複雑な正規表現を作成できます。変換の順序付けを使用すると、この照合が非常に簡単になります。

デフォルトの変換を順序付けるときは、必ずもっとも具体的なものからもっとも一般的なものへの順にします。それ以外の場合、後者の規則は決して使用しません。

この例では、次の 2 つの規則を使用します。

```
<transform file path=foo/bar/.* -> default group bin>
<transform file path=foo/.* -> default group sys>
```

一度しか配布されない各パッケージに適合したパターンを見つける必要があるため、変換を使用して上述の照合を使用するアクションを追加することは困難です。pkgmogrify ツールはこの問題に役立つ合成アクションを作成します。pkgmogrify は pkg.fmri 属性を設定するマニフェストごとにマニフェストを処理するので、pkg 合成アクションが pkgmogrify によって作成されます。pkg アクションは、それが実際にマニフェスト内にあるかように照合できます。

たとえば、配布されたソフトウェアのソースコードが見つかる Web サイト example.com を含むアクションをすべてのパッケージに追加する必要があったとします。次の変換でそれを行うことができます。

```
<transform pkg -> emit set info.source-url=http://example.com>
```

パッケージ化された変換

開発者にとって便利なように、Oracle Solaris OS をパッケージ化するときに使用された一連の変換は、`/usr/share/pkg/transforms` 内の次のファイルで入手できます。

`developer`	`facet.devel` を、`/usr/.*/include` に配布された `*.h` ヘッダーファイル、アーカイブライブラリと lint ライブラリ、`pkg-config`(1) データファイル、および `autoconf`(1) マクロに置きます。
`documentation`	さまざまな `facet.doc.*` ファセットをドキュメントファイルに置きます。
`locale`	さまざまな `facet.locale.*` ファセットをロケール固有のファイルに置きます。
`smf-manifests`	パッケージ化されたすべての SMF マニフェスト上の `svc:/system/manifest-import:default` を指す `restart_fmri` アクチュエータを追加して、パッケージがインストールされたあとにシステムがそのマニフェストをインポートするようにします。

♦ ♦ ♦ 第 7 章

7 パッケージインストールの一環としてのシステム変更の自動化

この章では、サービス管理機能 (SMF) を使用して、パッケージインストールの結果として生じる必要なすべてのシステム変更を自動的に処理する方法について説明します。SMF サービスの詳細については、『Oracle Solaris 11.3 でのシステムサービスの管理』を参照してください。

この章では、次について説明します。

- パッケージアクションでサービスアクチュエータを使用する方法
- IPS パッケージ内で SMF サービスを提供する方法
- IPS パッケージ内で最初のブートサービスを提供する方法
- パッケージインストールで複数のファイルを構成ファイルなどの 1 つのファイルにまとめる SMF サービスを提供する方法

パッケージアクションでのシステム変更の指定

最初に、インストール、更新、または削除されるときにシステムへの変更をもたらすアクションを特定します。たとえば、一部のシステム変更は 15 ページの「ソフトウェアの自己アセンブリ」で説明しているソフトウェアの自己アセンブリの概念を実装するために必要です。

それらのパッケージアクションごとに、必要なシステム変更を提供する既存の SMF サービスを特定します。あるいは、必要な機能を提供する新しいサービスを作成し、91 ページの「SMF サービスの提供」の説明に従ってそのサービスがシステムに配布されるようにします。

インストールされるときにシステムへの変更をもたらす一連のアクションを特定したら、システム変更を生じさせるために、パッケージマニフェスト内のそれらのアクションにタグを付けます。システム変更を生じさせるタグの値は、アクチュエータと呼ばれます。

次のアクチュエータタグは、マニフェスト内のどのアクションにも追加できます。

`reboot-needed`

このアクチュエータは `true` または `false` の値を取ります。このアクチュエータは、パッケージシステムがライブイメージ上で動作している場合、タグ付きアクションの更新または削除を新しいブート環境で実行する必要があることを宣言します。新しいブート環境の作成は、`be-policy` イメージプロパティーによって制御されます。`be-policy` プロパティーについての詳細は、`pkg`(1) のマニュアルページの「イメージプロパティー」セクションを参照してください。

SMF アクチュエータ

これらのアクチュエータは SMF サービスに関連しています。

SMF アクチュエータは値として単一のサービス FMRI を取りますが、場合により複数の FMRI に一致するグロッビング文字を含むこともあります。同じサービス FMRI が複数のアクションによって (場合により操作されている複数のパッケージにわたって) タグ付けされる場合、IPS はそのアクチュエータを一度だけトリガーします。

次の SMF アクチュエータのリストには、指定された各アクチュエータの値である、サービス FMRI への影響が説明されています。これらの説明では、「パッケージのアンインストール」は、サービスを別のパッケージに提供する `file` アクションの移動も含みます。

`disable_fmri`

パッケージをアンインストールする前に、指定されたサービスを無効にします (`svcadm disable`)。

`refresh_fmri`

パッケージのインストール、更新、またはアンインストール後に、指定されたサービスをリフレッシュします (`svcadm refresh`)。

`restart_fmri`

パッケージのインストール、更新、またはアンインストール後に、指定されたサービスを再起動します (`svcadm restart`)。

`suspend_fmri`

パッケージをインストールする前に指定されたサービスを一時的に無効化し (`svcadm disable -t`)、パッケージをインストールしたあとでサービスを有効化します (`svcadm enable`)。

これらの SMF アクチュエータは次のような場合は実行されません。

- 代替ルートで実行されているとき (`pkg -R` */path/to/BE*)。
- 大域ゾーンから再帰呼び出しするとき (`pkg` *subcommand* `-r`)。

SMF サービスの提供

新しい SMF サービスを配布するには、SMF マニフェストファイルおよびメソッドスクリプトを配布するパッケージを作成します。マニフェストファイルアクションで、マニフェストインポートサービスを再起動してシステム上のすべてのサービスマニフェストを再読み取りするため、次のアクチュエータを含めます。

```
restart_fmri=svc:/system/manifest-import:default
```

このアクチュエータは、マニフェストが追加、更新、または削除されると、`manifest-import` サービスが再起動され、それによってその SMF マニフェストによって配布されたサービスが追加、更新、または削除されるようにします。

次の例は、`restart_fmri` 属性が指定された完全な `file` アクションを示します。

```
file lib/svc/manifest/network/network-ipmgmt.xml \
    path=lib/svc/manifest/network/network-ipmgmt.xml \
    group=sys mode=0444 owner=root \
    restart_fmri=svc:/system/manifest-import:default
```

パッケージがライブシステムに追加される場合、このアクションはそのパッケージ化操作中にすべてのパッケージがシステムに追加されると実行されます。パッケージが代替ブート環境に追加される場合、このアクションはそのブート環境の最初のブート中に実行されます。

パッケージがインストールされている環境に不変非大域ゾーンが含まれている場合、不変ゾーンに新しいディレクトリをインストールするには、再ブートが必要です。読み取り専用でリブートする前に、不変ゾーンは `svc:/milestone/self-assembly-complete:default` マイルストーンまで読み取り/書き込みモードでブートします。91 ページの「1 度だけ実行されるサービスの提供」に、サービスに `self-assembly-complete` マイルストーンサービスへの依存関係を作成する方法を示します。

1 度だけ実行されるサービスの提供

このセクションでは、1 回の構成を実行する SMF サービスを配布するパッケージの例を示します。

次のパッケージマニフェストは、run-once サービスを配布します。

```
set name=pkg.fmri value=myapp-run-once@1.0
set name=pkg.summary value="Deliver a service that runs once"
set name=pkg.description \
    value="This example package delivers a service that runs once. The service
is marked with a flag so that it will not run again."
set name=org.opensolaris.smf.fmri value=svc:/site/myapplication-run-once \
    value=svc:/site/myapplication-run-once:default
```

```
set name=variant.arch value=i386
file lib/svc/manifest/site/myapplication-run-once.xml \
    path=lib/svc/manifest/site/myapplication-run-once.xml owner=root group=sys \
mode=0444 restart_fmri=svc:/system/manifest-import:default
file lib/svc/method/myapplication-run-once.sh \
    path=lib/svc/method/myapplication-run-once.sh owner=root group=bin \
    mode=0755
depend fmri=pkg:/shell/ksh93@93.21.0.20110208,5.11-0.175.3.0.0.19.0 type=require
depend fmri=pkg:/system/core-os@0.5.11,5.11-0.175.3.0.0.19.0 type=require
```

次のスクリプトは、このサービスの構成処理を実行します。このメソッドスクリプトは、スクリプトが 1 回だけ実行されるようにサービスで設定されているプロパティー `config/ran` を使用します。このプロパティーには、サービスマニフェストとメソッドスクリプトではそれぞれ異なる値が設定されます。exit 呼び出しのコメントが `svcs` コマンドによって表示されます。

```
#!/bin/sh

# Load SMF shell support definitions
. /lib/svc/share/smf_include.sh

# If nothing to do, exit with temporary disable.
ran=$(/usr/bin/svcprop -p config/ran $SMF_FMRI)
if [ "$ran" == "true" ] ; then
    smf_method_exit $SMF_EXIT_TEMP_DISABLE done "service ran"
fi

# Do the configuration work.

# Record that this run-once service has done its work.
svccfg -s $SMF_FMRI setprop config/ran = true
svccfg -s $SMF_FMRI refresh

smf_method_exit $SMF_EXIT_TEMP_DISABLE done "service ran"
```

次のリストに、この例の SMF サービスマニフェストを示します。このマニフェストの一部の機能について、リストのあとに説明します。

```
<?xml version="1.0" ?>
<!DOCTYPE service_bundle SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<service_bundle type="manifest" name="myapplication-run-once">
<service
    name='site/myapplication-run-once'
    type='service'
    version='1'>
    <dependency
        name='fs-local'
        grouping='require_all'
        restart_on='none'
        type='service'>
            <service_fmri value='svc:/system/filesystem/local:default' />
    </dependency>
    <dependent
        name='myapplication-run-once-complete'
        grouping='optional_all'
        restart_on='none'>
        <service_fmri value='svc:/milestone/self-assembly-complete' />
    </dependent>
    <instance enabled='true' name='default'>
        <exec_method
            type='method'
            name='start'
```

```
            exec='/lib/svc/method/myapplication-run-once.sh'
            timeout_seconds='60'/>
        <exec_method
            type='method'
            name='stop'
            exec=':true'
            timeout_seconds='0'/>
        <property_group name='startd' type='framework'>
            <propval name='duration' type='astring' value='transient' />
        </property_group>
        <property_group name='config' type='application'>
            <propval name='ran' type='boolean' value='false' />
        </property_group>
    </instance>
    <template>
         <common_name>
             <loctext xml:lang='C'>
                 Run-once service
             </loctext>
         </common_name>
         <description>
             <loctext xml:lang='C'>
                 This service checks and sets a property so that it runs
                 only once. This service is a dependency of the
                 self-assembly-complete milestone.
             </loctext>
         </description>
    </template>
</service>
</service_bundle>
```

- dependent 要素で、このサービスは self-assembly-complete システムマイルストーンにサービス自体を依存関係として追加します。
- svc.startd(1M) がこのサービスの処理を追跡しないように、このサービスの startd/duration プロパティーが transient に設定されています。
- このサービスの config/ran プロパティーは false に設定されています。サービスが 1 回だけ実行されるようにするため、サービスメソッドによってこのプロパティーが true に設定されます。
- このサービスでは、start メソッドの timeout_seconds は 60 に設定されています。timeout_seconds が 0 に設定されている場合、メソッドスクリプトが終了するまで SMF が無制限に待機します。

このサービスを 1 回だけ実行する理由をユーザーが理解できるようにするため、メソッドスクリプト出口にコメントを組み込み、サービステンプレートデータにサービス名と説明を組み込んでください。

サービスマニフェストが有効であることを確認します。

```
$ svccfg validate proto/lib/svc/manifest/site/myapplication-run-once.xml
```

53 ページの「パッケージを発行する」の説明に従ってパッケージを発行します。

パッケージのインストール前とインストール後に pkg verify を実行します。各実行の出力を比較して、そのスクリプトが編集可能としてマークされていないファイルの変更を試みないようにします。

パッケージのインストール後に、次の出力を確認します。

- svcs コマンドを使用してサービスの状態を表示します。svcs コマンドの各種オプションにより、追加情報が表示されます。ログファイル (-L) に、サービスメソッドが実行されたことが示されます。コメントとサービスの説明に、サービスが無効化されている理由が示されます。

```
$ svcs myapplication-run-once
STATE          STIME    FMRI
disabled       16:10:26 svc:/site/myapplication-run-once:default
$ svcs -x myapplication-run-once
svc:/site/myapplication-run-once:default (Run-once service)
 State: disabled since March 30, 2015 04:10:26 PM PDT
Reason: Temporarily disabled by an administrator.
   See: http://support.oracle.com/msg/SMF-8000-1S
   See: /var/svc/log/site-myapplication-run-once:default.log
Impact: This service is not running.
$ svcs -l myapplication-run-once
fmri         svc:/site/myapplication-run-once:default
name         Run-once service
enabled      false (temporary)
state        disabled
next_state   none
state_time   March 30, 2015 04:10:26 PM PDT
logfile      /var/svc/log/site-myapplication-run-once:default.log
restarter    svc:/system/svc/restarter:default
manifest     /lib/svc/manifest/site/myapplication-run-once.xml
dependency   require_all/none svc:/system/filesystem/local:default (online)
$ svcs -xL myapplication-run-once
svc:/site/myapplication-run-once:default (Run-once service)
 State: disabled since March 30, 2015 04:10:26 PM PDT
Reason: Temporarily disabled by an administrator.
   See: http://support.oracle.com/msg/SMF-8000-1S
   See: /var/svc/log/site-myapplication-run-once:default.log
Impact: This service is not running.
   Log:
[ 2015 Mar 30 16:10:22 Enabled. ]
[ 2015 Mar 30 16:10:22 Rereading configuration. ]
[ 2015 Mar 30 16:10:25 Executing start method ("/lib/svc/method/myapplication-run-
once.sh"). ]
[ 2015 Mar 30 16:10:26 Method "start" exited with status 101. ]
[ 2015 Mar 30 16:10:26 "start" method requested temporary disable: "service ran" ]
   Use: 'svcs -Lv svc:/site/myapplication-run-once:default' to view the complete log.
```

- myapplication-run-once サービスが self-assembly-complete サービスの依存関係であることを表示するため、svcs コマンドの -d オプションを使用します。

```
$ svcs -d svc:/milestone/self-assembly-complete:default | grep once
disabled       16:37:20 svc:/site/myapplication-run-once:default
```

- サービスが再度実行されることを防ぐフラグとして使用されるプロパティーの値を確認します。

```
$ svcprop -p config/ran myapplication-run-once
true
```

次の `svccfg` コマンドは、サービスマニフェストでこのプロパティーの値が `false` に設定され、その後 `true` にリセットされたことを示します。

```
$ svccfg -s myapplication-run-once:default listprop -l all config/ran
config/ran boolean     admin                true
config/ran boolean     manifest             false
```

このサービスを有効にすると、ログファイルに "Rereading configuration" 行が出力されず、サービスは構成処理を再実行せずに終了しています。

フラグメントファイルからのカスタムファイルの作成

このセクションでは、IPS パッケージを使用して複数のファイルを配布し、これらの複数ファイルを 1 つのファイルにまとめる SMF サービスを配布する方法を説明します。

次のパッケージマニフェストは、自己アセンブリサービスを配布します。`isvapp-self-assembly` サービスは、`/opt/isvapp/config.d` ディレクトリ内のファイル `inc1`、`inc2`、および `inc3` を 1 つのファイル `/opt/isvapp/isvconf` にまとめます。

```
set name=pkg.fmri value=isvappcfg@1.0
set name=pkg.summary value="Deliver isvapp config files and assembly service"
set name=pkg.description \
    value="This example package delivers a directory with fragment configuration
 files and a service to assemble them."
set name=org.opensolaris.smf.fmri value=svc:/site/isvapp-self-assembly \
    value=svc:/site/isvapp-self-assembly:default
set name=variant.arch value=i386
file lib/svc/manifest/site/isvapp-self-assembly.xml \
    path=lib/svc/manifest/site/isvapp-self-assembly.xml owner=root group=sys \
    mode=0444 restart_fmri=svc:/system/manifest-import:default
file lib/svc/method/isvapp-self-assembly.sh \
    path=lib/svc/method/isvapp-self-assembly.sh owner=root group=bin \
    mode=0755
dir  path=opt/isvapp owner=root group=bin mode=0755
dir  path=opt/isvapp/config.d owner=root group=bin mode=0755
file opt/isvapp/config.d/inc1 path=opt/isvapp/config.d/inc1 owner=root \
    group=bin mode=0644
file opt/isvapp/config.d/inc2 path=opt/isvapp/config.d/inc2 owner=root \
    group=bin mode=0644
file opt/isvapp/config.d/inc3 path=opt/isvapp/config.d/inc3 owner=root \
    group=bin mode=0644
file opt/isvapp/isvconf path=opt/isvapp/isvconf owner=root group=bin mode=0644
```

```
depend fmri=pkg:/shell/ksh93@93.21.0.20110208,5.11-0.175.3.0.0.19.0 type=require
depend fmri=pkg:/system/core-os@0.5.11,5.11-0.175.3.0.0.19.0 type=require
```

ほかのパッケージが同じ名前の構成ファイルを配布できるようにするには、ファイルに `overlay` および `preserve` 属性を追加します。例については、115 ページの「別のパッケージでも配布されるファイルの配布」を参照してください。

構成の新しいフラグメントをインストール、削除、または更新するときに構成ファイルをふたたびまとめるには、`restart_fmri` または `refresh_fmri` アクチュエータを構成ファイルに追加します。例については、18 ページの「Apache Web Server 構成」を参照してください。

次のスクリプトは、サービスの構成アセンブリを行います。exit 呼び出しのコメントが `svcs` コマンドによって表示されます。

```
#!/bin/sh

# Load SMF shell support definitions
. /lib/svc/share/smf_include.sh

# If files exist in /opt/isvapp/config.d,
# and if /opt/isvapp/isvconf exists,
# merge all into /opt/isvapp/isvconf

# After this script runs, the service does not need to remain online.
smf_method_exit $SMF_EXIT_TEMP_DISABLE done "/opt/isvapp/isvconf assembled"
```

次のリストに、この例の SMF サービスマニフェストを示します。このマニフェストは `svcbundle` コマンドを使用して作成されており、`multi-user` マイルストーンサービスへのデフォルトの依存関係を指定します。91 ページの「1 度だけ実行されるサービスの提供」に示すこの依存関係セクションを変更することがあります。

```
<?xml version="1.0" ?>
<!DOCTYPE service_bundle
  SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>
<!--
    Manifest created by svcbundle (2015-Mar-30 13:22:37-0700)
-->
<service_bundle type="manifest" name="site/isvapp-self-assembly">
    <service version="1" type="service" name="site/isvapp-self-assembly">
        <!--
            The following dependency keeps us from starting until the
            multi-user milestone is reached.
        -->
        <dependency restart_on="none" type="service"
            name="multi_user_dependency" grouping="require_all">
            <service_fmri value="svc:/milestone/multi-user"/>
        </dependency>
        <exec_method timeout_seconds="60" type="method" name="start"
            exec="/lib/svc/method/isvapp-self-assembly.sh"/>
        <!--
            The exec attribute below can be changed to a command that SMF
            should execute to stop the service.  See smf_method(5) for more
            details.
        -->
        <exec_method timeout_seconds="60" type="method" name="stop"
            exec=":true"/>
        <!--
            The exec attribute below can be changed to a command that SMF
```

```
            should execute when the service is refreshed.  Services are
            typically refreshed when their properties are changed in the
            SMF repository.  See smf_method(5) for more details.  It is
            common to retain the value of :true which means that SMF will
            take no action when the service is refreshed.  Alternatively,
            you may wish to provide a method to reread the SMF repository
            and act on any configuration changes.
        -->
        <exec_method timeout_seconds="60" type="method" name="refresh"
            exec=":true"/>
        <property_group type="framework" name="startd">
            <propval type="astring" name="duration" value="transient"/>
        </property_group>
        <instance enabled="true" name="default"/>
        <template>
            <common_name>
                <loctext xml:lang="C">
                    ISV app self-assembly
                </loctext>
            </common_name>
            <description>
                <loctext xml:lang="C">
                    Assembly of configuration fragment files for ISV app.
                </loctext>
            </description>
        </template>
    </service>
</service_bundle>
```

`svccfg validate` コマンドを使用して、サービスマニフェストが有効であることを確認します。

パッケージを発行してインストールしたあとで、`svcs` コマンドにより `isvapp-self-assembly` サービスが一時的に無効化されていることが表示され、ファイルアセンブリが完了しているというメソッド出口コメントがログファイルに出力されます。

91 ページの「1 度だけ実行されるサービスの提供」で示す例とは対照的に、`isvapp-self-assembly` サービスを有効にすると、サービスがふたたび無効になる前にサービス起動スクリプトが再度実行されます。

♦ ♦ ♦ 第 8 章

パッケージ更新の高度なトピック

この章では、次の内容について説明します。

- パッケージ内容の競合の回避
- パッケージ内容の変更
- パッケージの名前変更、マージ、および分割
- パッケージの廃止
- 移動するファイルまたはパッケージ化されていないファイルの保持
- ブート環境間での情報の共有
- 複数のアプリケーション実装の配布

パッケージ内容の競合の回避

一般に、パッケージでは同じパスで複数のアクションを提供するべきではありません。次のアクションだけが例外です。

- 同一の属性値を持つディレクトリ、リンク、およびハードリンク
- 調停されたリンクおよびハードリンク

最高のユーザーエクスペリエンスのため、`pkglint` ユーティリティーを使用して競合をチェックすることで、インストールおよび更新エラーを回避します。`pkglint` の詳細は、52 ページの「パッケージを検証する」および`pkglint`(1) のマニュアルページを参照してください。

単一のパッケージで、同じ内容のさまざまなバージョンを同じパスに配布する必要がある場合、バリアントを使用して、管理者がさまざまなバージョンの内容から選択してインストールできるようにします。詳細については、81 ページの「相互に排他的なソフトウェアコンポーネント」および `pkg`(7) のマニュアルページを参照してください。

複数のパッケージ (同じパッケージの 2 つのバージョンではなく) で、同じ内容のさまざまなバージョンを同じパスに配布する必要がある場合は、調停されたリンクを

使用して、管理者がその内容のさまざまなバージョンから選択できるようにします。競合している内容は異なる親ディレクトリにインストールし、ターゲットの場所に各バージョンの内容を指すリンクを作成する必要があります。管理者は `pkg set-mediator` コマンドを使用して、リンクのターゲットを変更することによって、異なる内容を切り替えられます。競合するパッケージの内容を調停する方法については、116 ページの「複数のアプリケーション実装の配布」を参照してください。

注記 - 調停は、両方のパッケージで同じタイプのアクションを提供する場合で、`link` および `hardlink` アクションに対してのみ使用できます。さらに、両方のパッケージで内容の調停を実装する必要があります。一方のパッケージで `/opt/tool` をリンクとして提供し、もう一方のパッケージで調停なしで `/opt/tool` をリンクとして提供した場合、競合が引き続き存在するため、ユーザーはエラーメッセージを受け取り、どちらのパッケージもインストールできないことがあります。

---

2 つの異なるパッケージで一部同じ内容を提供しているが、いずれかのパッケージだけをインストールする場合、パッケージの名前が異なることを確認します。パブリッシャー `example.com` がパブリッシャー `solaris` と一部同じ内容を提供する場合、エンドユーザーは、インストールコマンドにパブリッシャーを含む完全なパッケージ名を指定することによって競合を回避できる可能性があります。ただし、依存関係で引き続き問題が発生する可能性があります。`depend` アクションの `fmri` 属性は、パブリッシャーを除く完全なパッケージ名を指定します。次の依存関係は、`pkg://solaris/cat/subcat/tool` と `pkg://example.com/cat/subcat/tool` の両方に一致します。

```
depend fmri=pkg:/cat/subcat/tool type=require
```

これらの依存関係を区別して正しいパッケージをインストールするには、パッケージ名を変更します。次の提案ではパッケージ名を変更していますが、名前 `tool` を引き続き維持しています。

```
pkg://example.com/cat/subcat/example.com/tool
pkg://example.com/cat/subcat/example.com,tool
pkg://example.com/cat/subcat/vendor/example.com/tool
```

パッケージ内容の変更

パッケージマニフェストにはパッケージの完全な内容が記載されています。

- ファイルまたはその他のアクションをパッケージマニフェストから削除した場合、`pkg` コマンドを使用してパッケージが更新されたときにそのアクションはイメージから削除されます。
- パッケージマニフェスト内のファイルまたはその他のアクションの名前を変更した場合、`pkg` コマンドを使用してパッケージが更新されたときに新しいアクションがイメージにインストールされて古いアクションはイメージから削除されます。アク

ションの名前を変更する前に、99 ページの「パッケージ内容の競合の回避」を確認してください。

パッケージの名前変更、マージ、および分割

元のパッケージ内の誤り、時間の経過に伴う製品またはその使用法の変更、あるいは周辺のソフトウェア環境の変更のために、ソフトウェアコンポーネントの望ましい構成が変わることがあります。パッケージ名の変更のみが必要な場合もあります。そのような変更を計画する場合、アップグレードを行うユーザーは、意図しない副作用が絶対に発生しないように検討してください。

`pkg update` への考慮事項がますます多様化しているため、このセクションでは、3 種類のパッケージ再編成について説明します。

1. 単一パッケージの名前変更
2. 2 つのパッケージのマージ
3. パッケージの分割

単一パッケージの名前変更

単一パッケージの名前変更は簡単です。IPS には、パッケージの名前が変更されたことを知らせるメカニズムが備わっています。

パッケージの名前を変更するには、次の 2 つのアクションが含まれており、内容を持たない既存のパッケージの新しいバージョンを発行します。

- 次の書式による `set` アクション:

  ```
  set name=pkg.renamed value=true
  ```

- 新しいパッケージへの `require` 依存関係。

  ```
  depend fmri=pkg:/newpkgname@version type=require
  ```

名前変更されたパッケージは、`depend` または `set` アクション以外の内容を配布できません。

新しいパッケージでは、それを名前変更前の元のパッケージと同時にインストールできないようにする必要があります。どちらのパッケージも同じ結合依存関係の対象となっている場合は、この制約が自動的に適用されます。そうでない場合は、新しいパッケージには、名前変更されたバージョンに古いパッケージへの `optional` 依存関係が含まれている必要があります。これにより、ソルバーが両方のパッケージを選択して競合チェックが失敗することがなくなります。

この名前変更されたパッケージをインストールするユーザーはその名前変更された新しいパッケージを自動的に受け取ります。これは、それが古いバージョンの依存関係であるからです。名前変更されたパッケージが他のどのパッケージからも依存されていない場合、それは自動的にシステムから削除されます。古いソフトウェアが存在していると、名前変更されたいくつかのパッケージがインストール済みとして示されることがあります。その古いソフトウェアが削除されると、名前変更されたパッケージも自動的に削除されます。

パッケージの名前変更は何度行なっても問題はありませんが、ユーザーに紛らわしいことがあるため推奨されていません。

2 つのパッケージのマージ

パッケージのマージも簡単です。次の 2 つの事例は、パッケージをマージする例です。

- あるパッケージが名前変更バージョンで別のパッケージを取り込みます。

  パッケージ `A@2` がパッケージ `B@3` を取り込む必要があるとします。これを行うには、パッケージ `B` の名前をパッケージ `A@2` に変更します。前述のように両方のパッケージをまとめて更新できるように結合される場合を除き、`B@3` への `optional` 依存関係を `A@2` に忘れずに含めてください。`A` が `B` を取り込んだので、`B` を `B@3` にアップグレードするユーザーは `A` をインストールするようになります。
- 2 つのパッケージが同じ新しいパッケージ名に名前変更されます。

  この場合は、両方のパッケージの名前をマージされた新しいパッケージの名前に変更して、それらにほかの制約がなければ、古いパッケージへの 2 つの `optional` 依存関係を新しいパッケージに含めます。

パッケージの分割

パッケージを分割する場合は、101 ページの「単一パッケージの名前変更」に説明しているように、結果となるそれぞれの新しいパッケージの名前を変更します。結果となる新しいパッケージのいずれか 1 つを名前変更しない場合、そのパッケージの分割前と分割後のバージョンは互換性がなくなり、エンドユーザーがそのパッケージの更新を試みると、依存関係ロジックに違反する可能性があります。

元のパッケージの名前を変更し、分割によって作成されたすべての新しいパッケージへの `require` 依存関係を含めます。これにより、元のパッケージへの依存関係を持ったパッケージは、新しい内容をすべて入手します。

分割されたパッケージの一部のコンポーネントは、マージとして既存のパッケージに取り込むことができます。109 ページの「アプリケーションが共有領域を使用できるようにする方法」を参照してください。

パッケージの廃止

パッケージの廃止とは、パッケージの内容を取り出して空にしたり、パッケージをシステムから削除するときに使われるメカニズムです。

パッケージを廃止するには、次の `set` アクションが含まれており内容を持たない新しいバージョンを発行します。廃止されたパッケージは、`set` アクション以外の内容を配布できません。

```
set name=pkg.obsolete value=true
```

廃止されるパッケージの名前が以前に変更されている場合は、名前が変更されたこれらのパッケージも廃止し、その名前変更依存関係を削除する必要があります。パッケージを名前変更済みと廃止の両方としてマークすることはできません。名前変更されたパッケージで、`pkg.renamed` を `pkg.obsolete` に変更し、このパッケージの名前変更後のパッケージを指定する `depend` アクションを削除します。パッケージの名前変更のために実行された処理を確認するには、101 ページの「単一パッケージの名前変更」を参照してください。

廃止されたパッケージは `require` 依存関係を満たしません。インストールされたパッケージに、更新で廃止されたパッケージに対する `require` 依存関係がある場合、廃止されたパッケージに対する `require` 依存関係が含まれていない依存関係パッケージの新しいバージョンも提供されないかぎり、更新は失敗します。

廃止されたパッケージを廃止されていない状態にするには、廃止としてマークされていない新しいバージョンを発行します。廃止されたパッケージのインストール時にユーザーが更新を実行すると、廃止されたパッケージはシステムから削除されます。パッケージが廃止される前にユーザーが更新を実行したが、新しい廃止されていないバージョンのパッケージが発行されるまでは更新を再度実行しない場合、更新によってその新しいバージョンがインストールされます。

移行する編集可能なパッケージ化ファイルの保持

編集可能なパッケージ化ファイルは、パッケージ間で移動するか、インストールされたファイルシステム内で位置を変更することが必要な場合があります。

- パッケージ間で編集可能ファイルを移行する。

  IPS では、ファイル名とファイルパスが変わらない場合、パッケージ間を移動する編集可能ファイルの移行を試みます。パッケージの名前変更は、パッケージ間でファイルを移動する一例です。
- ファイルシステム内で編集可能ファイルを移行する。

  ファイルパスが変わる場合は、ユーザーによるそのファイルへのカスタマイズを保持するために `original_name` 属性が割り当てられているようにします。

最初にこのファイルを配布したパッケージ内の file アクションに属性 original_name が含まれていない場合は、その属性を更新済みパッケージに追加します。その属性の値を、元のパッケージの名前のあとにコロンとそのファイルへの元のパスを続けた値 (先頭の / なし) に設定します。

original_name 属性が編集可能ファイルに存在しているときは、その属性値を変更しないでください。更新時にスキップされたバージョンの数に関係なくユーザーの内容が適切に保持されるように、この値は今後のすべて移動において一意の識別子として機能します。

パッケージ解除されたファイルの保持

デフォルトでは、含まれているディレクトリがインストールされているその他のパッケージから参照されない場合、パッケージ解除された内容は自動的に /var/pkg/lost+found に回収されます。このセクションでは、次の代替手法を実装する方法について説明します。

- パッケージ解除された内容を、新しいパッケージ化ロケーションに移動する
- パッケージ化された内容がすべてその存在する場所からアンインストールされる場合でも、パッケージ解除された内容をその場所に維持します

ディレクトリの削除に伴うパッケージ解除されたファイルの移動

この例では、IPS を使用して、パッケージ解除された内容を別のパッケージ化されたディレクトリに回収する方法を示します。

この例では、パッケージ myapp@1.0 がディレクトリ /opt/myapp/logfiles をインストールします。myapp アプリケーションはそのディレクトリにログファイルを書き込みます。

myapp@2.0 パッケージは /opt/myapp/history ディレクトリを配布しますが、/opt/myapp/logfiles ディレクトリは配布しません。インストールされている myapp@1.0 パッケージを myapp@2.0 に更新するユーザーの場合、/opt/myapp/logfiles ディレクトリが存在しなくなります。このようなユーザーに対しては、/opt/myapp/logfiles の内容が /var/pkg/lost+found/opt/myapp/logfiles に保存されたことを通知するメッセージが、pkg update 出力の終わりに表示されます。

myapp パッケージの更新時に、IPS を使用して /opt/myapp/logfiles から /opt/myapp/history にファイルの内容を移動するには、/opt/myapp/history ディレクトリに salvage-from 属性を使用します。pkgmogrify 入力ファイルには次のエントリが必要です。

```
<transform dir path=opt/myapp/history -> \
    add salvage-from /opt/myapp/logfiles>
```

`pkgmogrify` の実行後には、このディレクトリに対するパッケージマニフェストアクションは次のようになります。

```
dir  path=opt/myapp/history owner=root group=bin mode=0755 \
    salvage-from=/opt/myapp/logfiles
```

ユーザーが `pkg update myapp` を実行したあとでは、`/opt/myapp/logfiles` ディレクトリが削除され、新しい `/opt/myapp/history` ディレクトリがインストールされ、`/opt/myapp/logfiles` のファイル内容が `/opt/myapp/history` に格納されます。

113 ページの「非共有の内容を共有領域に移行する方法」に、`salvage-from` 属性を使用する別の例を示します。

ディレクトリの個別パッケージ化

パッケージ化された内容がすべてそのディレクトリから削除される場合でも、パッケージ解除された内容をその存在する場所に維持するには、ディレクトリを個別にパッケージ化してインストールします。ディレクトリからほかのパッケージ化された内容がすべてアンインストールされた場合でも、そのディレクトリをインストールしたパッケージがインストールされている状態である限り、そのディレクトリはインストールされたままになります。

たとえば次の手順を実行し、*dir* に内容を配布するほかのインストール済み IPS パッケージがない場合は、*dir* ディレクトリの内容 (IPS によって配布されていないアプリケーションを含む) が `/var/pkg/lost+found` に回収されます。

1. *dir* に IPS パッケージとして配布されなかったアプリケーションをインストールします。
2. *dir* に内容をインストールする IPS パッケージをインストールします。
3. *dir* に内容をインストールする IPS パッケージをアンインストールします。

dir にインストールされているパッケージ解除されたソフトウェアを維持するには、*dir* ディレクトリを専用の IPS パッケージにパッケージ化します。

必要なディレクトリまたはディレクトリ構造を配布する IPS パッケージを作成します。そのパッケージをインストールします。そのパッケージをアンインストールするまでは、そのディレクトリ構造はその場所に維持されます。そのディレクトリに内容を配布するほかのパッケージをアンインストールしても、そのディレクトリは削除されません。

IPS によりすでに配布されているディレクトリを配布するパッケージは作成しないでください。異なる所有権、アクセス権、またはほかの属性が設定されているディレクトリが更新によってインストールされる場合、その更新は正常に完了しない可能性があります。次の手順の `pkgmogrify` ステップを参照してください。

▼ 内容のアンインストール後にディレクトリを保持する方法

1. **配布するディレクトリ構造を作成します。**

   この例は /usr/local を示します。/usr/local/bin または IPS パッケージで配布されない異なるディレクトリ構造を含めるように、容易に拡張できます。

```
$ mkdir -p usrlocal/usr/local
```

2. **初期パッケージマニフェストを作成します。**

```
$ pkgsend generate usrlocal | pkgfmt > usrlocal.p5m.1
$ cat usrlocal.p5m.1
dir path=usr owner=root group=bin mode=0755
dir path=usr/local owner=root group=bin mode=0755
```

3. **IPS によりすでに配布されているディレクトリを除外します。**

   配布する /usr はすでに Oracle Solaris により配布されているので除外し、またメタデータを追加するため、pkgmogrify 入力ファイルを作成します。ディレクトリの所有権またはアクセス権をデフォルトから変更するために変換を追加することもできます。

```
$ cat usrlocal.mog
set name=pkg.fmri value=pkg://site/usrlocal@1.0
set name=pkg.summary value="Create the /usr/local directory."
set name=pkg.description value="This package installs the /usr/local \
directory so that /usr/local remains available for unpackaged files."
set name=variant.arch value=$(ARCH)
<transform dir path=usr$->drop>
```

4. **変更を初期マニフェストに適用します。**

```
$ pkgmogrify -DARCH=`uname -p` usrlocal.p5m.1 usrlocal.mog | \
pkgfmt > usrlocal.p5m.2
$ cat usrlocal.p5m.2
set name=pkg.fmri value=pkg://site/usrlocal@1.0
set name=pkg.summary value="Create the /usr/local directory."
set name=pkg.description \
    value="This package installs the /usr/local directory so that /usr/local
remains available for unpackaged files."
set name=variant.arch value=i386
dir path=usr/local owner=root group=bin mode=0755
```

5. **作業内容を確認します。**

```
$ pkglint usrlocal.p5m.2
Lint engine setup...
Starting lint run...
$
```

6. **パッケージをリポジトリに発行します。**

   この例では、yourlocalrepo リポジトリのデフォルトパブリッシャーはすでに site に設定されています。

```
$ pkgsend -s yourlocalrepo publish -d usrlocal usrlocal.p5m.2
pkg://site/usrlocal@1.0,5.11:20140303T180555Z
```

```
PUBLISHED
```

7. インストールする新しいパッケージが表示されていることを確認します。

```
$ pkg refresh site
$ pkg list -a usrlocal
NAME (PUBLISHER)      VERSION      IFO
usrlocal (site)       1.0          ---
```

8. パッケージをインストールします。

```
$ pkg install -v usrlocal
           Packages to install:         1
     Estimated space available:  20.66 GB
Estimated space to be consumed: 454.42 MB
       Create boot environment:        No
Create backup boot environment:        No
          Rebuild boot archive:        No

Changed packages:
site
  usrlocal
    None -> 1.0,5.11:20140303T180555Z
PHASE                                        ITEMS
Installing new actions                         5/5
Updating package state database               Done
Updating package cache                         0/0
Updating image state                          Done
Creating fast lookup database                 Done
Reading search index                          Done
Updating search index                          1/1
```

9. パッケージがインストールされたことを確認します。

```
$ pkg list usrlocal
NAME (PUBLISHER)      VERSION      IFO
usrlocal (site)       1.0          i--
$ pkg info usrlocal
          Name: usrlocal
       Summary: Create the /usr/local directory.
   Description: This package installs the /usr/local directory so that
                /usr/local remains available for unpackaged files.
         State: Installed
     Publisher: site
       Version: 1.0
 Build Release: 5.11
        Branch: None
Packaging Date: March  3, 2014 06:05:55 PM
          Size: 0.00 B
          FMRI: pkg://site/usrlocal@1.0,5.11:20140303T180555Z
$ ls -ld /usr/local
drwxr-xr-x   2 root     bin            2 Mar  3 10:17 /usr/local/
```

ブート環境間での内容の共有

IPS パッケージの内容は、BE 内のファイルシステムにのみインストールできます。たとえば、Oracle Solaris 11 のデフォルトのインストールでは、`rpool/ROOT/`*BEname*`/` 内

のデータセットだけがパッケージ操作でサポートされています。IPS を使用して BE 外部の内容を直接配布すると、システムが古い BE をブートまたはクローニングできなくなります。

108 ページの「Oracle Solaris での既存の共有内容」で説明したように、一部の内容は BE 間で共有されます。

パッケージ化された内容を共有領域に配布するには、108 ページの「共有領域への内容の配布」の説明に従ってリンクを使用します。

Oracle Solaris での既存の共有内容

一部のファイルは、複数の BE を含む環境で正常なシステム操作を保持するために BE 間で共有される必要があります。次のディレクトリは、IPS によって BE 間ですでに共有されています。

```
/var/audit
/var/cores
/var/crash
/var/mail
```

各 BE 内では、これらのディレクトリは次の共有ディレクトリへのシンボリックリンクです。

```
/var/share/audit
/var/share/cores
/var/share/crash
/var/share/mail
```

これらの共有ディレクトリは、`/var/share` にマウントされた共有データセットである VARSHARE データセット内にあります。

BE 間で共有する必要があるほかのデータも同様の方法で処理できます。

共有領域への内容の配布

BE 間でデータを共有するには、共有データセットと、BE 内のディレクトリ構造からその共有データセットを指すシンボリックリンクを使用します。IPS パッケージが BE 内部でシンボリックリンクを配布します。共有データセットを作成およびマウントする SMF サービスが、同じパッケージまたは別のパッケージから配布されます。パッケージがインストールされている BE から新しい共有データセットへのリンクは、108 ページの「Oracle Solaris での既存の共有内容」に示す `/var/share` へのリンクに似ています。BE で実行されるアプリケーションは、共有領域への書き込みまたは共有領域からの読み取りをリンクを介して実行します。

ベストプラクティスは、異なる BE で実行されている複数のアプリケーションが内容を共有できる 1 つのデータセットを作成することです。共有するデータのディレクト

リごとに個別のデータセットを作成すると、各非大域ゾーンに複数のデータセットが作成される結果となりますが、1 つのゾーンに複数のデータセットを作成することは望ましくありません。たとえば /opt/share でマウントする OPTSHARE データセットを作成するとします。さまざまなアプリケーションが、/opt/share 内の異なるディレクトリ内のデータを共有できます。

個別のパッケージとサービスを使用して共有データセットを作成できますが、このようなパッケージは別の BE からはインストールできない点に注意してください。共有データセットを作成したパッケージとサービスが現行 BE にインストールされていない場合でも、その共有データセットが使用可能である場合があります。これらの例に示されているアプリケーションパッケージによって配布されるサービスは、データセットがすでに存在しているかどうかを確認し、まだ存在していない場合はデータセットを作成します。

▼ アプリケーションが共有領域を使用できるようにする方法

この手順では、このアプリケーションが複数 BE 内の複数のアプリケーションとデータを共有できるようにするために、共有データセットと、現行 BE からその共有データセットへのリンクを提供する方法を説明します。アプリケーションパッケージは次を配布します。

- 共有データセットを作成する SMF サービス
- 現行 BE 内に存在しており、そのターゲットが共有データセット内にあるリンク

この例のパッケージには、共有データセットを作成してその共有データセットにリンクするために必要なアクションだけを示します。実行可能ファイルや構成ファイルなど、アプリケーションのほかのアクションはこの例では省略されています。

1. パッケージ開発領域を作成します。

BE に必要なディレクトリと、BE 外部の共有領域へのリンクを含む、パッケージ開発のための領域を作成します。

a. 共有データセットを作成するサービス起動メソッドとサービスマニフェストを配布するために必要な構造を作成します。

```
$ mkdir -p proto/lib/svc/manifest/site
$ mkdir -p proto/lib/svc/method
```

b. アプリケーションが共有データセットにアクセスするために使用できるリンクを配布します。

```
$ mkdir -p proto/opt/myapp
$ ln -s ../../opt/share/myapp/logfiles proto/opt/myapp/logfiles
```

2. サービス起動メソッドを作成します。

proto/lib/svc/method で、次のタスクを実行するスクリプトを作成します。

- BE 間で共有される rpool/OPTSHARE データセットを作成します。共有データセットの作成は、プールごとに 1 回だけ行う必要があります。プール内の現行 BE および将来の BE はすべて、そのデータセットにアクセスできます。データセットを作成する前に、そのデータセットがすでに存在しているかどうかを確認します。
- 共有データセット内に、この例でのリンクのターゲット (/opt/share/myapp/logfiles) を含むディレクトリ構造を作成します。

このスクリプトは、サービスの起動メソッドです。この例では、スクリプトの名前は myapp-share-files.sh です。次の手順でサービスマニフェストを作成するときに、このファイル名が必要になります。

スクリプトには、次のプロトタイプに示されている要素が必要です。デフォルトでは sh が ksh93 である点に注意してください。smf_method_exit には smf_include.sh ファイルが必要です。smf_method_exit の 3 番目の引数は、サービスログファイルと svcs コマンドからの出力に示されます。

```
#!/bin/sh

# Load SMF shell support definitions
. /lib/svc/share/smf_include.sh

# Create rpool/OPTSHARE with mount point /opt/share if it does not already exist
# Create /opt/share/myapp/logfiles if it does not already exist

# After this script runs, the service does not need to remain online.
smf_method_exit $SMF_EXIT_TEMP_DISABLE done "shared area created"
```

3. サービスマニフェストを作成します。

proto/lib/svc/manifest/site で、svcbundle コマンドを使用してサービスを作成します。これは内部サービスであるため、service-name にカテゴリ site を含めます。start-method の値は、前の手順で作成したスクリプトの名前です。

```
$ svcbundle -s service-name=site/myapp-share-files \
-s start-method=/lib/svc/method/myapp-share-files.sh -o myapp-share-files.xml
```

作成されたサービスマニフェストを編集し、common_name と description の情報を template データ領域に追加します。また、documentation とその他のテンプレートデータも追加できます。マイルストーン依存関係や起動メソッドのタイムアウトなど、デフォルト設定の一部を変更することもできます。デフォルトでは、作成されたインスタンスには default という名前が付けられ、有効化されます。

サービスマニフェストが有効であることを確認します。

```
$ svccfg validate myapp-share-files.xml
```

4. 初期パッケージマニフェストを生成します。

pkgsend generate コマンドを使用して、パッケージ開発領域から初期パッケージマニフェストを作成します。

```
$ pkgsend generate proto | pkgfmt > share.p5m.1
$ cat share.p5m.1
dir  path=lib owner=root group=bin mode=0755
```

```
dir  path=lib/svc owner=root group=bin mode=0755
dir  path=lib/svc/manifest owner=root group=bin mode=0755
dir  path=lib/svc/manifest/site owner=root group=bin mode=0755
file lib/svc/manifest/site/myapp-share-files.xml \
    path=lib/svc/manifest/site/myapp-share-files.xml owner=root group=bin \
    mode=0644
dir  path=lib/svc/method owner=root group=bin mode=0755
file lib/svc/method/myapp-share-files.sh \
    path=lib/svc/method/myapp-share-files.sh owner=root group=bin mode=0755
dir  path=opt owner=root group=bin mode=0755
dir  path=opt/myapp owner=root group=bin mode=0755
link path=opt/myapp/logfiles target=../../opt/share/myapp/logfiles
```

5. メタデータとアクチュエータを追加します。

a. 次の `pkgmogrify` 入力ファイルを作成し、`share.mog` という名前を付けます。

- パッケージに名前、バージョン、サマリー、および説明を指定します。
- `opt`、`lib/svc/manifest/site`、および `lib/svc/method` ディレクトリはほかのパッケージによってすでに配布されているため、これらのディレクトリアクションを削除します。
- `/lib/svc/manifest` 内のほかのマニフェストに対応するように、サービスマニフェストのグループを `sys` に変更します。
- システム上のほかのマニフェストとメソッドに対応するように、サービスマニフェストのモードを `0444` に変更し、サービスメソッドのモードを `0555` に変更します。
- メソッドファイルのインストールまたは更新時には常に `manifest-import` サービスを再起動するようにするため、サービスマニフェストのアクチュエータとメソッドファイルを追加します。

```
set name=pkg.fmri value=myapp@2.0
set name=pkg.summary value="Deliver shared directory"
set name=pkg.description value="This example package delivers a directory \
and link that allows myapp content to be shared across BEs."
set name=variant.arch value=$(ARCH)
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
<transform dir path=opt$->drop>
<transform dir path=lib$->drop>
<transform dir path=lib/svc$->drop>
<transform dir path=lib/svc/manifest$->drop>
<transform dir path=lib/svc/manifest/site$->drop>
<transform dir path=lib/svc/method$->drop>
<transform file path=lib/svc/manifest/site/myapp-share-files.xml -> \
    edit group bin sys>
<transform file path=lib/svc/manifest/site/myapp-share-files.xml -> \
    edit mode 0644 0444>
<transform file path=lib/svc/manifest/site/myapp-share-dir.xml -> \
    add restart_fmri svc:/system/manifest-import:default>
<transform file path=lib/svc/method/myapp-share-files.sh -> \
    edit mode 0755 0555>
<transform file path=lib/svc/method/myapp-share-dir.sh -> \
    add restart_fmri svc:/system/manifest-import:default>
```

b. share.mog の変更とともに share.p5m.1 マニフェストに対し pkgmogrify を実行します。

```
$ pkgmogrify -DARCH=`uname -p` share.p5m.1 share.mog | pkgfmt > share.p5m.2
$ cat share.p5m.2
set name=pkg.fmri value=myapp@2.0
set name=pkg.summary value="Deliver shared directory"
set name=pkg.description \
    value="This example package delivers a directory and link that allows myapp
content to be shared across BEs."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
file lib/svc/manifest/site/myapp-share-files.xml \
    path=lib/svc/manifest/site/myapp-share-files.xml owner=root group=sys \
    mode=0444
file lib/svc/method/myapp-share-files.sh \
    path=lib/svc/method/myapp-share-files.sh owner=root group=bin mode=0755
dir  path=opt/myapp owner=root group=bin mode=0555
link path=opt/myapp/logfiles target=../../opt/share/myapp/logfiles
```

6. パッケージの依存関係を評価して解決します。

pkgdepend コマンドを使用して、パッケージの依存関係を自動的に生成および解決します。依存関係の解決からの出力は、接尾辞 .res が付いたファイルに自動的に保存されます。

この例では、ksh93 の依存関係が生成されます。起動メソッドの処理によっては、追加の依存関係が生成されることがあります。また、サービスとサービスインスタンスが org.opensolaris.smf.fmri で宣言されます。

```
$ pkgdepend generate -md proto share.p5m.2 | pkgfmt > share.p5m.3
$ pkgdepend resolve -m share.p5m.3
```

出力 share.p5m.3.res ファイルに次の行が追加されます。

```
set name=org.opensolaris.smf.fmri value=svc:/site/myapp-share-files \
    value=svc:/site/myapp-share-files:default

depend fmri=pkg:/shell/ksh93@93.21.0.20110208-0.175.2.0.0.37.1 type=require
depend fmri=pkg:/system/core-os@0.5.11-0.175.2.0.0.37.0 type=require
```

7. パッケージを検証します。

```
$ pkglint share.p5m.3.res
```

8. パッケージを発行します。

```
$ pkgsend -s site publish -d protosl share.p5m.3.res
pkg://site/myapp@2.0,5.11:20140417T000014Z
PUBLISHED
```

9. パッケージをテストします。

パッケージをインストールします。

```
$ pkg install -g ./site myapp
```

データセットとリンクが存在することを確認します。

```
$ zfs list rpool/OPTSHARE
NAME            USED  AVAIL  REFER  MOUNTPOINT
rpool/OPTSHARE  38.5K  24.8G  38.5K  /opt/share
$ ls -l /opt/myapp
lrwxrwxrwx   1 root   root   21 Apr 16 17:32 logfiles -> /opt/share/myapp/logfiles
```

パッケージをアンインストールします。`/opt/myapp/logfiles` リンクが削除され、サービスマニフェストとメソッドスクリプトが削除されているはずです。`rpool/OPTSHARE` データセットはパッケージ化された内容ではないため、存在している必要があります。これはサービスによって作成されたものです。

▼ 非共有の内容を共有領域に移行する方法

この手順は、前述の手順を拡張したものです。この例では、共有する必要がある一部のデータがすでに存在しています。アプリケーションパッケージは、ステージング領域を配布し、共有するデータをステージング領域にコピーします。SMF サービスはステージング領域から共有領域にデータを移動します。

- リンクとして再定義されるディレクトリにすでに存在しているパッケージ解除された内容を保存するため、パッケージはリンクに加えステージング領域を BE に配布します。
- SMF サービスは、共有データセットを作成するだけでなく、ステージング領域の既存の内容をすべて共有領域に移動します。

1. パッケージ開発領域を作成します。

`logfiles` のすべてのオカレンスを `logs` に変更します。

```
$ mkdir -p proto/lib/svc/manifest/site
$ mkdir -p proto/lib/svc/method
$ mkdir -p proto/opt/myapp
$ ln -s ../../opt/share/myapp/logs proto/opt/myapp/logs
```

この例では、`myapp@1.0` パッケージが `/opt/myapp/logs` をディレクトリとしてインストールし、`myapp` アプリケーションが内容をそのディレクトリに書き込みます。この新しい `myapp@3.0` パッケージが `/opt/myapp/logs` をリンクとしてインストールすると、`/opt/myapp/logs` ディレクトリの内容はすべて `/var/pkg/lost+found` に保存されます。代わりにその内容を新しい共有領域に保存するには、その内容のコピーを保持する領域を配布します。

```
$ mkdir -p proto/opt/myapp/.migrate/logs
```

このサービスにより BE 内のステージング領域から共有領域に内容が移動されます。

`myapp@3.0` パッケージが `/opt/myapp/logs` に書き込む内容は、前の例で示したようにリンクを介して共有領域に直接移動されます。

2. サービス起動メソッドを作成します。

前の手順で作成した proto/lib/svc/method/myapp-share-files.sh スクリプトに、「ステージング領域から共有領域への内容の移動」タスクを追加します。

サービスを使用して空のステージング領域を削除しないでください。ステージング領域はパッケージ化された内容であり、パッケージのアンインストールによってのみ削除される必要があります。

```
#!/bin/sh

# Load SMF shell support definitions
. /lib/svc/share/smf_include.sh

# Create rpool/OPTSHARE with mount point /opt/share if it does not already exist
# Create /opt/share/myapp/logfiles if it does not already exist
# Move any content from /opt/myapp/.migrate/logs to /opt/share/myapp/logs

# After this script runs, the service does not need to remain online.
smf_method_exit $SMF_EXIT_TEMP_DISABLE done "myapp shared files moved"
```

3. サービスマニフェストを作成します。

```
$ svcbundle -s service-name=site/myapp-share-files \
-s start-method=/lib/svc/method/myapp-share-files.sh -o myapp-share-files.xml
```

svccfg validate コマンドを使用して、サービスマニフェストが有効であることを確認します。

4. 初期パッケージマニフェストを生成します。

このマニフェストは、前述の例のパッケージマニフェストと同じですが、次の点が変更されています。

- logfiles のオカレンスがすべて logs に変更されています。
- 次の 2 つのアクションが追加されています。

```
dir  path=opt/myapp/.migrate owner=root group=bin mode=0755
dir  path=opt/myapp/.migrate/logs owner=root group=bin mode=0755
```

5. メタデータとアクチュエータを追加します。

前述の例の pkgmogrify の share.mog 入力ファイルに、次の行を追加します。salvage-from 属性は、/opt/myapp/logs ディレクトリのパッケージ解除された内容をすべて /opt/myapp/.migrate/logs ディレクトリに移動します。その後サービスが /opt/myapp/.migrate/logs から /opt/share/myapp/logs にその内容を移動します。

```
<transform dir path=opt/myapp/.migrate/logfiles -> \
    add salvage-from /opt/myapp/logfiles>
```

このパッケージに myapp@3.0 という名前を付けます。

前述の例と同様に pkgmogrify を実行します。

6. パッケージの依存関係を評価して解決します。

7. **パッケージを検証します。**

8. **パッケージを発行します。**

9. **パッケージをテストします。**
   標準ディレクトリとして `/opt/myapp/logs` を作成し、いくつかのファイルを配置します。

   `myapp@3.0` パッケージをインストールします。データセットの存在が正しく検出および処理されること、新しいリンクが存在していること、`/opt/myapp/logs` ディレクトリが空であること、`/opt/myapp/.migrate/logs` ディレクトリが存在しており空であること、`/opt/share/myapp/logs` ディレクトリが存在しており、`/opt/myapp/logs` ディレクトリに最初に保管されていた内容が含まれていることを確認します。

別のパッケージでも配布されるファイルの配布

IPS パッケージを使用すると、別のパッケージによりすでに配布されているファイルのカスタマイズバージョンを提供できます。デフォルトでは、特定の場所にファイルを配布できるのは 1 つの IPS パッケージだけです。IPS パッケージを使用して、別の IPS パッケージにより配布されるファイルのカスタムバージョンを配布するには、`file` アクションで次の属性が設定されていることを確認します。

- 置換対象ファイルで `overlay=allow` および `preserve=true` 属性が設定されています。
- 置換ファイルで `overlay=true` 属性と `preserve` 属性 (任意の値) が設定されています。

26 ページの「ファイルアクション」の `overlay` および `preserve` 属性の説明を参照してください。

`overlay=allow` 属性が設定されているバージョンは、`overlay=true` 属性が設定されているファイルのバージョンに置き換えられ、`overlay=allow` 属性が設定されているファイルのバージョンは `/var/pkg/lost+found/` に保存されます。

たとえば、次の `file` アクションを含む `isvapp` というパッケージをインストールするとします。

```
file opt/isvapp/isvconf path=opt/isvapp/isvconf owner=root group=bin mode=0644 \
    overlay=allow preserve=true
```

このパッケージは次のファイルをインストールします。

```
-rw-r--r--   1 root     bin        11358 Apr 17 18:44 /opt/isvapp/isvconf
```

このファイルのサイト固有のバージョンをすべてのシステムに配置するとします。このファイルの新しいバージョンを配布するには、次の `file` アクションを含む `isvconf` という名前のパッケージを作成します。

```
file opt/isvapp/isvconf path=opt/isvapp/isvconf owner=root group=bin mode=0644 \
    overlay=true preserve=renameold
```

isvconf がインストールされると、次のファイルがシステムに配置されます。

```
$ ls -l /opt/isvapp/isvconf
-rw-r--r--   1 root     bin        72157 Apr 17 18:47 /opt/isvapp/isvconf
$ ls -l /var/pkg/lost+found/opt/isvapp
total 24
-rw-r--r--   1 root     bin        11358 Apr 17 18:44 isvconf-20140417T184756Z
```

同じパスのファイルを配布する別のパッケージ (この例では isvconf2) をインストールしようとすると、次の例外でインストールが失敗します。

```
Creating Plan (Checking for conflicting actions): -
pkg install: The following packages all deliver file actions to opt/isvapp/isvconf:

  pkg://site/isvconf2@1.0,5.11:20140417T190405Z
  pkg://site/isvapp@1.0,5.11:20140417T182316Z
  pkg://site/isvconf@1.0,5.11:20140417T185420Z

These packages may not be installed together. Any non-conflicting set may
be, or the packages must be corrected before they can be installed.
```

最初の置換ファイルを配布したパッケージの更新で、ファイルの新しいバージョンを配布できます。isvconf@2.0 がインストールされると、次のファイルがシステムに配置されます。

```
$ ls -l /opt/isvapp/isvconf*
-rw-r--r--   1 root     bin        64064 Apr 17 18:52 /opt/isvapp/isvconf
-rw-r--r--   1 root     bin        54365 Apr 17 18:47 /opt/isvapp/isvconf.old
$ ls -l /var/pkg/lost+found/opt/isvapp
total 24
-rw-r--r--   1 root     bin        11358 Apr 17 18:44 isvconf-20140417T184756Z
```

次の 2 つの条件の両方に該当するため、既存のファイルは isvconf.old に保存されました。

- isvconf パッケージに preserve=renameold が指定されています。
- このファイルは、isvconf@1.0 のインストール後、isvconf@2.0 のインストール前に編集されています。

lost+found 領域は変更されておらず、isvapp により配布された元のファイルが引き続きこの領域に含まれています。

複数のアプリケーション実装の配布

次のような特性を持つ、特定のアプリケーションの複数の実装を配布することが必要な場合があります。

- すべて実装がイメージ内で使用できます。

- 検出しやすいように、/usr/bin などの共通ディレクトリからいずれかの実装が使用可能です。
- 管理者は、パッケージの追加や削除を行わずに、共通ディレクトリから使用可能な実装を容易に変更できます。

Oracle Solaris 11 には、Java や Python など各種アプリケーションの複数の実装が用意されています。/usr/bin などの共通ディレクトリから使用可能な実装を指定し、管理者がその選択を容易に変更できるようにするには、調停されたリンクを使用します。

調停されたリンクは、アプリケーションの複数の実装を 1 つのイメージで管理します。調停されたリンクは、mediator 属性が設定されたシンボリックリンクです (118 ページの「調停されたリンクの属性」を参照)。mediator 属性を持つ link アクションでパッケージ化されたソフトウェアは、調停の参加要素です。/usr/bin などの共通ディレクトリから使用可能な調停参加要素は、優先バージョンと呼ばれます。調停の優先バージョンは、次のいずれかの方法で決定します。

パッケージマニフェストでの指定

調停の参加要素ごとに、バージョン (mediator-version) またはバージョン管理された実装 (mediator-implementation) を指定できます。競合が発生する場合に備えてオーバーライド優先度 (mediator-priority) を指定できます。

システムによる選択

調停の参加要素に優先度が指定されている場合、優先度がもっとも高い参加要素が優先実装として選択されます。

優先度が指定されている調停の参加要素が存在せず、参加要素にバージョンが指定されている場合は、もっとも高い値のバージョンが指定されている参加要素が優先実装として選択されます。

調停の参加要素に優先度もバージョンも指定されていない場合、任意の参加要素が優先実装として選択されます。選択されている参加要素の mediator-implementation にバージョン文字列が含まれている場合、その mediator-implementation にもっとも高い値のバージョン文字列が指定されている参加要素が優先実装となります。

管理者による指定

管理者は pkg set-mediator コマンドを使用して優先実装を設定できます。『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「デフォルトのアプリケーション実装の指定」を参照してください。

特定の調停のインスタンスが 1 つだけイメージにインストールされる場合、そのインスタンスはその調停の優先実装として自動的に選択されます。パッケージのインストール後に優先実装がシステム管理者により設定される場合、この同じ調停に追加の参加要素をインストールしても、管理者によって設定された優先実装は変更されません。

調停されたリンクの属性

次の属性を link アクションで設定すると、調停されたリンクの配布方法を制御できます。

mediator

特定の調停グループに参加しているすべてのパス名で共有されている調停名前空間内のエントリを指定します。例としては、java、python、および ruby などがあります。

mediator 属性を持つすべてのリンクには、mediator-version 属性または mediator-implementation 属性も必要です。特定のパス名に対する仲介リンクはすべて、同じ mediator を指定する必要があります。ただし、すべてのメディエータバージョンおよび実装が、特定のパスでリンクを提供する必要はありません。調停参加要素がリンクを提供しない場合、その参加要素が優先実装として選択されるとそのリンクが削除されます。

mediator-version

mediator 属性によって記述されるインタフェースのバージョンを指定します。この属性は、mediator が指定され、mediator-implementation が指定されていない場合に必要です。mediator-version の値は、整数のドットで区切られた並びです。操作を容易にするため、指定する値は、そのリンクを配布するパッケージのバージョンに一致している必要があります。たとえば runtime/ruby-19 パッケージは mediator-version=1.9 を指定する必要があります。バージョン値を適切に設定することで、管理者は調停に参加しているソフトウェア、そのソフトウェアを配布するパッケージ、および優先バージョンとして現在設定されているソフトウェアのバージョンを判別できます。mediator-priority が設定されている調停の参加要素がない場合、IPS は、mediator-version の値がもっとも高い調停参加要素を優先実装として選択します。

mediator-implementation

mediator 属性によって記述されるインタフェースの実装を指定します。この属性は、mediator が指定され、mediator-version は指定されていない場合に必要です。実装の文字列は順序付けられているとはみなされません。mediator-version または mediator-priority 属性が設定されている調停の参加要素がない場合、IPS によって優先実装として任意の実装が選択されます。

mediator-implementation の値は、英数字と空白で構成された任意の長さの文字列にすることができます。実装自体をバージョン管理できる合は、文字列の最後の記号 (@) のあとにバージョンを指定します。このバージョンは、整数のドットで区切られた並びです。実装のバージョンが複数存在する場合は、最上位バージョンの実装が選択されます。たとえば値が 4DB@12 の mediator-implementation は、値が 4DB@11 の mediator-implementation よりも優先して選択されます。

mediator-priority

mediator 属性によって記述されるインタフェースの優先度を指定します。mediator-version または mediator-implementation も指定する必要があります。たとえば、調停の 1 つの参加要素の mediator-version に値 1.6 が設定されており、別の参加要素の mediator-version に値 1.7 が設定されている場合、mediator-version の値が 1.6 の参加要素を優先実装として指定するには、mediator-priority 属性を割り当てます。

mediator-priority 属性は、次のいずれかの値を持ちます。

vendor　このリンクは、mediator-priority が指定されていないリンクより優先されます。

site　このリンクは、mediator-priority が指定されていないリンクと、mediator-priority の値が vendor であるリンクよりも優先されます。

調停されたリンクの指定

次のコマンドは、現在選択されている Python、Ruby、および Secure Shell の優先実装を表示します。

```
$ pkg mediator python ruby ssh
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
python      vendor    2.6     vendor
ruby        system    1.9     system
ssh         vendor              vendor     sunssh
```

次のコマンドは、各調停の参加要素をすべて表示します。

```
$ pkg mediator -a python ruby ssh
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
python      vendor    2.6     vendor
python      system    2.7     system
ruby        system    1.9     system
ruby        system    1.8     system
ssh         vendor              vendor     sunssh
```

下位バージョンがシステムにより優先 Python 実装として選択されていますが、これは、VER. SRC. および IMPL. SRC. と次のコマンドによって示されるように、このバージョンには mediator-priority が指定されているためです。

```
$ pkg contents -Ho action.raw -t link -a path=usr/bin/python 'runtime/python*'
link mediator=python mediator-version=2.7 path=usr/bin/python pkg.linted.pkglint
.dupaction010.2=true target=python2.7
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
python target=python2.6
```

-a オプションの引数として mediator=python を指定すると、出力では python 調停にさらに多くのリンクが表示されます。調停に必要なすべてのパスを含めることを忘れないでください。

```
$ pkg contents -Ho action.raw -t link -a mediator=python runtime/python-26
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
2to3 target=2to3-2.6
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
python target=python2.6
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
pydoc target=pydoc-2.6
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
idle target=idle-2.6
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
python-config target=python2.6-config
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
amd64/python target=python2.6 variant.arch=i386
link mediator=python mediator-priority=vendor mediator-version=2.6 path=usr/bin/
amd64/python-config target=python2.6-config variant.arch=i386
link facet.doc.man=all mediator=python mediator-priority=vendor mediator-version
=2.6 path=usr/share/man/man1/python.1 target=python2.6.1
```

`runtime/python-27` パッケージ内の調停されたリンク `usr/bin/python` の `pkg.linted.pkglint.dupaction010.2=true` 属性は、`/usr/bin/python` リンクが複数のパッケージによって配布されており、有効な調停されたリンクであることを示します。調停されたリンクは、アクションを配布できるのは 1 つのパッケージだけであるという規則の例外です。`pkglint` ユーティリティーが、重複するアクションの有無を確認します。`pkg.linted.`*check.id* を `true` に設定すると、そのアクションの *check.id* の確認が省略されます。52 ページの「パッケージを検証する」および`pkglint(1)` のマニュアルページを参照してください。`pkglint` が実行するチェックの完全なリストを表示するには、`pkglint -L` コマンドを使用します。`pkglint.dupaction010` チェックの説明は「調停されたリンクは有効である」です。

システムにより、上位バージョンが優先 Ruby 実装として選択されます。

```
$ pkg contents -Ho action.raw -t link -a path=usr/bin/ruby runtime/ruby-19
link mediator=ruby mediator-version=1.9 path=usr/bin/ruby pkg.linted.pkglint.dup
action010.2=true target=./ruby19
```

`ssh` 調停の参加要素は 1 つだけです。追加のアプリケーション実装を配布することを予定している場合は、ほかの実装が配布される場合に元のパッケージが調停の参加要素となるように、元のパッケージで調停を定義します。このようにしないと、元のパッケージに対する更新を配布する必要が生じます。更新を配布しないと、ユーザーが元の実装を優先実装として選択できなくなります。

次のコマンドは、リンクアクションのほかに、このアクションが定義されているパッケージの名前も表示します。

```
$ pkg contents -o pkg.name,action.raw -t link -a path=usr/bin/ssh '*'
PKG.NAME    ACTION.RAW
network/ssh link mediator=ssh mediator-implementation=sunssh mediator-priority=
vendor path=usr/bin/ssh target=../lib/sunssh/bin/ssh
```

`mediator-implementation` を指定する調停されたリンクは、`mediator-version`、`mediator-priority`、またはこの両方も指定できます。調停のすべての参加要素が `mediator-implementation` だけを指定している場合、システムは優先実装を任意に選択します。選択された `mediator-implementation` がバージョン管理されている場合、次のコマンドに示すように最上位バージョンが選択されます。

```
$ pkg mediator -a myapp
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp       system            system     db@12
myapp       system            system     db@11
myapp       system            system     db
$ pkg mediator myapp
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp       system            system     db@12
```

調停に別の実装が追加されると、次のコマンドにより示されるように、システムによってその実装が選択されることがあります。

```
$ pkg mediator -a myapp
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp       system            system     aa
myapp       system            system     db@12
myapp       system            system     db@11
myapp       system            system     db
$ pkg mediator myapp
MEDIATOR    VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp       system            system     aa
```

調停されたリンクのベストプラクティス

同じパスを、あるパッケージではリンクとして配布し、別のパッケージではディレクトリまたはファイルとして配布することはしないでください。一般に、同じパスを 2 回以上配布しないでください。同じリンクパスを 2 回以上配布する場合は、各インスタンスで異なるターゲットが指定され、各インスタンスが同じ調停に参加しているようにします。

調停に必要なすべてのパスを含めることを忘れないでください。ライブラリ、構成ファイル、マニュアルページ、およびその他のファイルシステムオブジェクトは、実装によって異なる可能性があります。

追加のアプリケーション実装を配布することを予定している場合は、ほかの実装が配布される場合に元のパッケージが調停の参加要素となるように、元のパッケージで調停を定義します。このようにしないと、元のパッケージに対する更新を配布する必要が生じます。更新を配布しないと、ユーザーが元の実装を優先実装として選択できなくなります。

調停に参加しているソフトウェアに対し、ほかのソフトウェアが依存関係を持つ場合、およびそのソフトウェアのいずれかのバージョンが依存関係を満たす場合は、`require-any` 依存関係を使用します。`require-any` 依存関係については、35 ページの「依存アクション」を参照してください。

操作しやすいように、`mediator-version` には、リンクを配布するパッケージのバージョンと一致する値を指定してください。バージョン値を適切に設定することで、管理者は調停に参加しているソフトウェア、そのソフトウェアを配布するパッケージ、および優先バージョンとして現在設定されているソフトウェアのバージョンを判別できます。

♦ ♦ ♦ 第 9 章

9 IPS パッケージの署名

この章では、IPS パッケージの署名と、開発者および品質保証組織による新しいパッケージまたは既存の署名付きパッケージの署名方法について説明します。

- システムにインストールされたソフトウェアが、実際にパブリッシャーが最初に指定したとおりであることを検証する機能は、IPS の重要な機能の 1 つです。インストールされたシステムを検証するこの機能は、ユーザーとサポートエンジニアリング要員のどちらにとっても重要です。
- 署名は、検証に加えて、ほかの組織または関係者による承認を示すためにも使用できます。たとえば、パッケージが本稼働での使用に適格であると認められると、内部の品質管理組織がパッケージのマニフェストに署名する場合が挙げられます。そのような承認がインストールに必要となることがあります。
- パッケージは複数回署名して、複数レベルでの承認を示すことができます。パッケージの署名により、`signature` アクションがマニフェストに追加されますが、ほかにパッケージは変更されません。パッケージの署名により、以前の署名が削除または無効にされることはありません。
- 署名ポリシーはイメージまたは特定のパブリッシャーに対して設定できます。ポリシーには、署名の無視、既存の署名の検証、署名の要求、信頼チェーン内の特定の共通名の要求などがあります。

署名アクション

ほかのすべてのマニフェスト内容がアクションとして表されるのとまったく同様に、署名もアクションとして表されます。マニフェストにはすべてのパッケージメタデータ (ファイルアクセス権、所有権、内容のハッシュなど) が含まれているため、マニフェストがその発行以降変わってないことを検証する `signature` アクションはシステム検証の重要な部分です。

`signature` アクションは、インストールされたソフトウェアの完全な検証ができるように、配布されたバイナリが含まれるツリーを形成します。

`signature` アクションの形式は次のとおりです。

```
signature hash_of_certificate algorithm=signature_algorithm \
    value=signature_value \
```

```
chain="hashes_of_certificates_needed_to_validate_primary_certificate" \
version=pkg_version_of_signature
```

ペイロードおよび chain 属性は、発行元のリポジトリから取り出せる x.509 証明書を含む、PEM (Privacy Enhanced Mail) ファイルのパッケージハッシュを表します。ペイロード証明書とは、value の値を検証する証明書です。value は、後述のように用意された、マニフェストのメッセージテキストの署名付きハッシュです。

提示されたほかの証明書は、ペイロード証明書からトラストアンカーに通じる証明書パスを形成する必要があります。

2 種類の署名アルゴリズムがサポートされています。

RSA
: 最初のタイプの署名アルゴリズムは RSA グループのアルゴリズムです。RSA 署名アルゴリズムの一例は rsa-sha256 です。ハイフンのあとの文字列 (この例では sha256) は、メッセージテキストを RSA アルゴリズムが使用できる単一値に変えるために使用するハッシュアルゴリズムを指定します。

ハッシュのみ
: 2 つめのタイプの署名アルゴリズムはハッシュのみの計算です。このタイプのアルゴリズムは、主としてテストやプロセス検証のために存在し、ハッシュを署名値として提示します。このタイプの署名アクションは、ペイロード証明書ハッシュがないことによって示されます。このタイプの署名アクションは、イメージが署名をチェックするように構成されている場合に検証されます。ただし、署名が必要な場合、その存在は署名とみなされません。次の例は、ハッシュのみの signature アクションを示しています。

```
signature algorithm=hash_algorithm value=hash \
    version=pkg_version_of_signature
```

署名付きパッケージ

マニフェストには、ほかに依存しない署名を複数含めることができます。署名を追加または削除しても、存在している他の署名が無効になることはありません。この機能により、パスに従って署名を使用することで途中過程での完了が示され、本稼働でのハンドオフが容易になります。以降の手順では、オプションでいつでも前の署名を削除できます。pkgsign コマンドのオプションの説明と使用例については、pkgsign(1) のマニュアルページを参照してください。

パッケージに署名するには、次の 2 つの手順を実行します。2 番目の手順は必要な回数だけ実行して、複数の署名を追加できます。

1. 53 ページの「パッケージを発行する」に示すように、署名されていないパッケージをリポジトリに公開します。

2. 55 ページの「パッケージの署名」に示すように、pkgsign コマンドを使用して、署名アクションをリポジトリ内のマニフェストに追加します。signature アクションの追加を除き、パッケージはそのタイムスタンプも含め、変更されません。パッケージの署名は、パッケージ開発の、パッケージをテストする前の最後の手順にするようにしてください。

pkgsign コマンドにより、パッケージパブリッシャー以外のだれでも元のパブリッシャーの署名を無効にすることなく、パッケージに signature アクションを追加できます。パッケージの再発行により、新しいタイムスタンプが作成され、元の署名が無効にされます。pkgsign コマンドによって、たとえば品質管理部門では、社内にインストールされたすべてのパッケージに署名して、パッケージを再発行しなくても使用が承認されていることを示せます。

注記 - pkgsign コマンドを使用することは、署名付きパッケージに署名を追加する唯一の方法です。すでに署名が含まれているパッケージを発行した場合、その署名は削除され、警告が発行されます。

バリアントを含む署名アクションは無視されます。そのため、マニフェストのペアに対して pkgmerge を実行すると、以前に適用されたすべての署名が無効になります。

カスタム証明書認証局証明書の使用

カスタム認証局 (CA) 証明書は、ほかの証明書に署名するために使用されます。システムは、証明書で参照されている CA が、/etc/certs/CA に対応する CA 証明書を持つことを確認することによって、鍵と証明書が有効かどうかを判断します。

▼ カスタム証明書認証局証明書の使用方法

1. **カスタム CA 証明書を作成します。**

   独自の CA 証明書の作成およびテストについては、『Oracle Solaris 11.3 パッケージリポジトリのコピーと作成』 の 「自己署名付きサーバー認証局の作成」を参照してください。

2. **CA 証明書を trust-anchor-directory プロパティーで指定されたディレクトリに配置します。**

   trust-anchor-directory イメージプロパティーについては、127 ページの「イメージとパブリッシャーのプロパティーの構成」を参照してください。

   - CA 証明書を trust-anchor-directory で指定されたディレクトリに直接配置します。証明書を別のサブディレクトリに配置しないでください。

- ディレクトリ内にすでに存在する証明書の複製である CA 証明書をディレクトリ内に配置しないでください。
- 有効な証明書ファイルではないファイルをディレクトリ内に配置しないでください。

3. **`ca-certificates` サービスをリフレッシュします。**

```
$ svcadm refresh svc:/system/ca-certificates:default
```

サービスがオンラインであることを確認します。

```
$ svcs ca-certificates
```

サービスが `online` 状態でない場合、または CA が `/etc/certs/ca-certificates.crt` に表示されない場合、次のサービスログファイルを確認します。

```
$ svcs -xL ca-certificates
```

4. **(オプション) カスタム証明書および鍵をパッケージ化します。**

証明書および鍵ファイルをパッケージ化すると、複数のシステムの証明書および鍵の更新が簡単になります。証明書を変更する必要がある場合、パッケージを更新し、各システムでパッケージの `pkg update` を実行します。

   a. **配布する各証明書および鍵ファイルにリフレッシュアクチュエータを追加します。**

```
file group=sys mode=0644 owner=root path=etc/certs/CA/mycert.pem \
refresh_fmri=svc:/system/ca-certificates:default
```

次の `pkgmogrify` 規則はこのリフレッシュアクチュエータの追加を自動化します。

```
<transform file path=etc/certs/CA/.*\.pem ->
  add refresh_fmri svc:/system/ca-certificates:default>
```

   b. **パッケージで `/etc/certs/CA` ディレクトリを配布しないでください。**

`/etc`、`/etc/certs`、および `/etc/certs/CA` ディレクトリはすでにシステムによって配布されています。46 ページの「生成されたマニフェストに必要なメタデータを追加する」および 52 ページの「パッケージを検証する」を参照してください。

署名付きパッケージのトラブルシューティング

`pkgsign` ツールでは、パッケージの署名時にその入力に対して可能なすべてのチェックを行うわけではありません。そのため、署名付きパッケージをチェックして、署名後にそれらが適切にインストールされるようにすることが重要です。

このセクションでは、署名付きパッケージのインストールまたは更新を試みたときに発生する可能性のあるエラーを示し、それらのエラーの説明と問題の解決策を提供します。

署名付きパッケージは、署名付きパッケージに固有の理由でインストールまたは更新に失敗することがあります。たとえば、パッケージの署名の検証に失敗した場合、または信頼チェーンが検証できないか、信頼できる証明書にアンカーを設定できない場合、そのパッケージはインストールに失敗します。

イメージとパブリッシャーのプロパティーの構成

このセクションで説明するイメージとパブリッシャーのプロパティーは、署名付きパッケージに対して行われるチェックに影響します。

イメージプロパティーを構成するには、`pkg` コマンドの `set-property`、`add-property-value`、`remove-property-value`、および `unset-property` サブコマンドを使用します。

特定のパブリッシャーの署名ポリシーと必要な名前を指定するには、`set-publisher` サブコマンドの `--set-property`、`--add-property-value`、`--remove-property-value`、および `--unset-property` オプションを使用します。

例については、『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「パッケージの署名プロパティーの構成」を参照してください。

次のイメージプロパティーを、署名付きパッケージを使用するように構成します。

`signature-policy`

このプロパティーの値により、イメージ内のパッケージのインストール、更新、変更、または検証時に、マニフェストに実行されるチェックが決まります。パッケージに適用される最終的なポリシーは、イメージポリシーとパブリッシャーポリシーの組み合わせに依存します。この組み合わせの厳格さは、少なくとも、この 2 つのポリシーが個別に適用された場合の厳格な方と同じです。デフォルトでは、パッケージクライアントは証明書が失効済みかどうかをチェックしません。そのようなチェック (クライアントから外部 Web サイトへのアクセスが必要な場合がある) を有効にするには、`check-certificate-revocation` イメージプロパティーを `true` に設定します。次の値が許可されます。

`ignore`

すべてのマニフェストの署名を無視します。

`verify`

署名が含まれているすべてのマニフェストが有効に署名されていることを確認しますが、インストール済みパッケージがすべて署名されている必要はありません。

これがデフォルト値です。

`require-signatures`

新しくインストールされたすべてのパッケージに、有効な署名が少なくとも 1 つ含まれている必要があります。インストール済みパッケージに有効な署名が含まれていない場合は、`pkg fix` および `pkg verify` コマンドでも警告が表示されます。

`require-names`

`require-signatures` と同じ要件に従いますが、`signature-required-names` イメージプロパティーで一覧表示される文字列が、署名の信頼のチェーンを検証するために使用される証明書の共通名としても表示される必要があります。

`signature-required-names`

このプロパティーの値は、パッケージの署名の検証中に、証明書の共通名として表示される必要のある名前の一覧です。

`trust-anchor-directory`

イメージのトラストアンカーを含むディレクトリの相対パス名 (イメージのルートに対して相対的)。デフォルトは `etc/certs/CA/` です。

独自の SSL 証明書認証局証明書を作成する場合は、125 ページの「カスタム証明書認証局証明書の使用方法」の説明に従って、それらの証明書を `trust-anchor-directory` によって名前が付けられたディレクトリに配置し、`ca-certificates` サービスをリフレッシュします。CA 証明書は `trust-anchor-directory` によって名前が付けられたディレクトリに直接配置し、証明書を別のサブディレクトリに配置しないでください。

次のパブリッシャープロパティーを、特定のパブリッシャーからの署名付きパッケージを使用するように構成します。

`signature-policy`

このプロパティーの機能は、このプロパティーが指定したパブリッシャーからのパッケージにのみ適用される点を除き、`signature-policy` イメージプロパティーの機能と同じです。

`signature-required-names`

このプロパティーの機能は、このプロパティーが指定したパブリッシャーからのパッケージにのみ適用される点を除き、`signature-required-names` イメージプロパティーの機能と同じです。

チェーン証明書が見つからない

次のエラーは、信頼チェーン内の証明書が存在しないか、または間違っている場合に発生します。

```
pkg install: The certificate which issued this certificate:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_ch1_ta3/emailAddress=cs1_ch1_ta3
could not be found. The issuer is:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=ch1_ta3/emailAddress=ch1_ta3
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T184152Z
```

この例では、パッケージが署名されるときに信頼チェーン内に 3 つの証明書がありました。信頼チェーンはトラストアンカーの `ta3` という名前の証明書をルートに持ちます。`ta3` 証明書は、`ch1_ta3` という名前のチェーン証明書に署名し、`ch1_ta3` は `cs1_ch1_ta3` という名前のコード署名証明書に署名しました。

`pkg` コマンドがパッケージのインストールを試みると、コード署名証明書 `cs1_ch1_ta3` は見つかりましたが、チェーン証明書 `ch1_ta3` は見つからなかったため、信頼チェーンを確立できませんでした。

この問題のもっとも一般的な原因は、正しい証明書を `pkgsign` の `-i` オプションに指定しなかったことです。

承認された証明書が見つからない

次のエラーは前の例に示したエラーに似ていますが、原因が異なります。

```
pkg install: The certificate which issued this certificate:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_cs8_ch1_ta3/emailAddress=cs1_cs8_ch1_ta3
could not be found. The issuer is:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs8_ch1_ta3/emailAddress=cs8_ch1_ta3
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T201101Z
```

この場合、パッケージは `cs1_cs8_ch1_ta3` 証明書を使って署名され、その証明書は `cs8_ch1_ta3` 証明書によって署名されたものでした。

問題は、`cs8_ch1_ta3` 証明書にほかの証明書に署名する権限がなかったことです。特に、`cs8_ch1_ta3` 証明書では `basicConstraints` 拡張機能が `CA:false` に設定されており、クリティカルのマークが付けられていました。

`pkg` コマンドが信頼チェーンを検証する際、`cs1_cs8_ch1_ta3` 証明書への署名を許可されている証明書を検出しません。信頼チェーンをリーフからルートまで検証することはできないため、`pkg` コマンドはそのパッケージをインストールできないようにします。

自己署名付き証明書が信頼できない

次のエラーは、信頼チェーンが、システムによって信頼されていない自己署名付き証明書で終わるときに発生します。

```
pkg install: Chain was rooted in an untrusted self-signed certificate.
The package involved is:pkg://test/example_pkg@1.0,5.11-0:20110919T185335Z
```

テストのために OpenSSL を使用して証明書チェーンを作成した場合、テストのためにのみ使用される証明書を社外で検証してもらう理由がほとんどないために、ルート証明書は通常、自己署名されます。

テストの状況では、2 つの解決策があります。

- 最初の解決策は、信頼チェーンのルートとなる自己署名付き証明書を `/etc/certs/CA` 内に追加し、`system/ca-certificates` サービスをリフレッシュすることです。これは、オペレーティングシステムとともにトラストアンカーとして配布される証明書を最終的にルートとする証明書で本稼働用のパッケージが署名されるという、顧客が遭遇する可能性のある状況を反映しています。
- 2 つめの解決策は、`pkg set-publisher` コマンドで `--approve-ca-cert` オプションを使用することで、テスト用のパッケージを提供するパブリッシャーの自己署名付き証明書を承認することです。

署名の値が期待された値と一致しない

次のエラーは、`signature` アクションに対する値を、そのアクションの要求どおりにそのパッケージの署名に使用された鍵とペアになっている証明書を使用して検証できなかったときに発生します。

```
pkg install: A signature in pkg://test/example_pkg@1.0,5.11-0:20110919T195801Z
could not be verified for this reason:
The signature value did not match the expected value. Res: 0
The signature's hash is 0ce15c572961b7a0413b8390c90b7cac18ee9010
```

このようなエラーの考えられる原因は 2 つあります。

- 可能性のある最初の原因は、パッケージがその署名以降に変更されていることです。これは起こりそうにないことですが、署名以降にパッケージマニフェストが手動で編集された場合は起こり得ます。パッケージが新しいタイムスタンプを取得すると古い署名は無効になるため、`pkgsend` は発行中に既存の `signature` アクションを取り除きます。そのため、手動操作がなければ、パッケージはその署名以降に変更されなかったはずです。
- 2 つ目のより可能性の高い原因は、パッケージの署名に使用された鍵と証明書が一致するペアではなかったことです。`pkgsign` の `-c` オプションに指定した証明書が、`pkgsign` の `-k` オプションに指定した鍵を使用して作成されなかった場合、そのパッケージは署名されますが、その署名は検証されません。

重要な拡張機能が不明である

次のエラーは、信頼チェーン内の証明書で pkg が認識できない重要な拡張機能が使用されたときに発生します。

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs2_ch1_ta3/emailAddress=cs2_ch1_ta3
could not be verified because it uses a critical extension that pkg5 cannot
handle yet. Extension name:issuerAltName
Extension value:<EMPTY>
```

pkg がその重要な拡張機能の処理方法を認識するまで、唯一の解決策は、問題となっている重要な拡張機能を使わずにその証明書を再生成することです。

拡張機能の値が不明である

次のエラーは前述のエラーと似ていますが、その問題が未知の重要な拡張機能に関するものではなく、pkg が認識している拡張機能の、pkg が認識していない値に関するものである点が異なります。

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs5_ch1_ta3/emailAddress=cs5_ch1_ta3
could not be verified because it has an extension with a value that pkg(7)
does not understand.
Extension name:keyUsage
Extension value:Encipher Only
```

この場合、pkg は keyUsage 拡張機能を認識しますが、その値 Encipher Only は認識しません。このエラーは、問題となっている拡張機能が重要であろうとなかろうと同じように表示されます。

この解決策は、pkg が問題となっている値を認識するまで、その値をその拡張機能から削除するか、その拡張機能を完全に削除することです。

証明書の承認されていない使用

次のエラーは、証明書が承認されなかった目的で使用されたときに発生します。

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=ch1_ta3/emailAddress=ch1_ta3
could not be verified because it has been used inappropriately.
The way it is used means that the value for extension keyUsage must include
'DIGITAL SIGNATURE' but the value was 'Certificate Sign, CRL Sign'.
```

この場合は、証明書 ch1_ta3 がパッケージの署名に使用されました。証明書の keyUsage 拡張機能は、その証明書がほかの証明書と CRL (証明書失効リスト) の署名にのみ有効であることを意味しています。

予期しないハッシュ値

次のエラーは、証明書がパブリッシャーから最後に取り出された以降に変更されたことを示しています。

```
pkg install: Certificate
/tmp/ips.test.7149/0/image0/var/pkg/publisher/test/
certs/0ce15c572961b7a0413b8390c90b7cac18ee9010
has been modified on disk. Its hash value is not what was expected.
```

指定されたパスにある証明書はインストールされているパッケージの検証に使用されますが、ディスク上にあるそれらの内容のハッシュが、署名アクションが想定していたものと一致しません。

簡単な解決策は、その証明書を削除し、pkg でその証明書を再度ダウンロードできるようにすることです。

証明書が失効している

次のエラーは、インストールされているパッケージの信頼チェーン内に存在していた、問題になっている証明書がその証明書の発行者によって失効されたことを示しています。

```
pkg install: This certificate was revoked:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_ch1_ta4/emailAddress=cs1_ch1_ta4
for this reason: None
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T205539Z
```

♦ ♦ ♦ 第 10 章

10 非大域ゾーンの処理

ゾーンで常に機能するパッケージの開発には通常、追加の作業がほとんど必要ないか、まったく必要ありません。この章では、次について説明します。

- IPS がゾーンを処理する方法
- 大域ゾーンコンポーネントと非大域ゾーンコンポーネントの両方を提供するソフトウェアをパッケージする方法

非大域ゾーンについてのパッケージ化の考慮事項

ゾーンとパッケージ化を検討する場合は、2 つの問題を解決する必要があります。

- パッケージ内のすべてのものに大域ゾーンと非大域ゾーンの間の境界を越えるインタフェースがあるか。
- そのパッケージのどこまでを非大域ゾーンにインストールするか。

パッケージが大域ゾーンと非大域ゾーンの間の境界を超えるか

pkgA がカーネルとユーザーランドの両方の機能を配布し、そのインタフェースの両側がそれに合わせて更新される必要がある場合、pkgA が非大域ゾーンで更新されたときは必ず、pkgA がインストールされているほかのすべてのゾーンでも pkgA が更新される必要があります。

この更新が正しく行われるようにするには、pkgA の parent 依存関係を使用します。単一のパッケージがインタフェースの両側を配布する場合、feature/package/dependency/self への parent 依存関係は、大域ゾーンと非大域ゾーンに同じバージョンのパッケージが含まれるようにし、インタフェース間でバージョンスキューが発生するのを防ぎます。

また、parent 依存関係はそのパッケージが非大域ゾーンにある場合にそれが大域ゾーンにも存在するようにします。

インタフェースが複数のパッケージに及ぶ場合、そのインタフェースの非大域ゾーン側を含むパッケージには、そのインタフェースの大域ゾーン側を配布するパッケージへの parent 依存関係を含める必要があります。parent 依存関係については、71 ページの「依存関係の種類」でも説明されています。

パッケージのどこまでを非大域ゾーンにインストールするか

パッケージを非大域ゾーンにインストールするときにパッケージのすべてをインストールする場合は、そのパッケージが正しく機能できるようにするために何か行う必要はありません。しかし、パッケージの顧客にとっては、パッケージ作成者がゾーンへのインストールをきちんと考慮し、このパッケージがゾーンで機能できると結論づけたことがわかっていると安心できることがあります。このような理由から、パッケージ機能が大域ゾーンと非大域ゾーンの両方にあることを明示的に記述するようにしてください。これを行うには、次のアクションをマニフェストに追加します。

```
set name=variant.opensolaris.zone value=global value=nonglobal
```

非大域ゾーンにインストールできる内容がパッケージにない場合 (カーネルモジュールやドライバのみを配布するパッケージなど)、そのパッケージでそれを非大域ゾーンにインストールできないことを明記するようにします。これを行うには、次のアクションをマニフェストに追加します。

```
set name=variant.opensolaris.zone value=global
```

パッケージの一部ではあるがすべてではない内容を非大域ゾーンにインストールできる場合は、次の手順を実行します。

1. 次の set アクションを使用して、そのパッケージを大域ゾーンと非大域ゾーンの両方にインストールできることを記述します。

   ```
   set name=variant.opensolaris.zone value=global value=nonglobal
   ```

2. 大域ゾーンにのみ、または非大域ゾーンにのみ関係があるアクションを特定します。大域ゾーンにのみ関係があるアクションに次の属性を割り当てます。

   ```
   variant.opensolaris.zone=global
   ```

   非大域ゾーンにのみ関係があるアクションに次の属性を割り当てます。

   ```
   zone:variant.opensolaris.zone=nonglobal
   ```

パッケージに parent 依存関係が含まれていたり、大域ゾーンと非大域ゾーンで異なる要素が含まれている場合は、そのパッケージが非大域ゾーンと大域ゾーンで予想どおりに動作していることを確認するためのテストを実行します。

パッケージにそれ自身への parent 依存関係が含まれている場合、大域ゾーンではそのパッケージをその起点の 1 つとして配布するリポジトリを構成する必要がありま

す。そのパッケージをまず大域ゾーンにインストールし、次にテストのために非大域ゾーンにインストールします。

非大域ゾーンへのパッケージのインストールに関するトラブルシューティング

このセクションでは、非大域ゾーンへのパッケージのインストールを試みたときに発生する可能性のある問題について説明します。

自分自身への parent 依存関係を持つパッケージ

非大域ゾーンへのパッケージのインストール中に問題が発生した場合は、次のサービスが大域ゾーンでオンラインになっていることを確認します。

```
svc:/application/pkg/zones-proxyd:default
svc:/application/pkg/system-repository:default
```

次のサービスが非大域ゾーンでオンラインになっていることを確認します。

```
svc:/application/pkg/zones-proxy-client:default
```

これらの 3 つのサービスは、非大域ゾーンに対するパブリッシャー構成と、大域ゾーンから提供されたシステムパブリッシャーに割り当てられたリポジトリにリクエストを出すために非大域ゾーンが使用できる通信チャネルを提供します。

非大域ゾーン内のパッケージはそれ自身への parent 依存関係を持っているため、更新することはできません。大域ゾーンから更新を開始します。`pkg` は非大域ゾーンを大域ゾーンとともに更新します。

パッケージを非大域ゾーンにインストールしたら、そのパッケージの機能をテストします。

自分自身への parent 依存関係を持たないパッケージ

パッケージがそれ自身への `parent` 依存関係を持たない場合は、大域ゾーンでパブリッシャーを構成する必要はないため、そのパッケージを大域ゾーンにインストールしないようにしてください。大域ゾーンのパッケージを更新しても、非大域ゾーンのパッケージは更新されません。この場合、大域ゾーンのパッケージを更新すると、それより古い非大域ゾーンのパッケージをテストしたときに予期しない結果が生じる可能性があります。

この状況でのもっとも簡単な解決策は、パブリッシャーを非大域ゾーンで使用できるようにし、非大域ゾーンの内部からパッケージをインストールして更新することです。

ゾーンがパブリッシャーのリポジトリにアクセスできない場合は、大域ゾーンでパブリッシャーを構成すると、`zones-proxy-client` および `system-repository` サービスが非大域ゾーンのパブリッシャーにプロキシ経由でアクセスできるようになります。その後、パッケージを非大域ゾーンにインストールして更新します。

♦ ♦ ♦ 第 11 章

11 発行されたパッケージの変更

場合によっては、生成しなかったパッケージの変更が必要になることがあります。たとえば、属性をオーバーライドしたり、パッケージの一部を内部実装に置き換えたり、システムで許可されないバイナリを削除したりすることが必要な場合があります。

この章では、ローカルの状況に合わせて既存のパッケージを変更する方法について説明します。

この章では、次の内容について説明します。

- パッケージの再発行
- パッケージのメタデータの変更
- パッケージパブリッシャーの変更

パッケージの再発行

IPS を使用すると、既存のパッケージを、最初にそのパッケージを発行していなくても、変更を加えて再発行することが簡単にできます。また、`pkg update` がユーザーの期待どおりに機能し続けるように、変更済みパッケージの新しいバージョンを再発行することもできます。

パッケージを変更して再発行するには、次の手順を使用します。

1. `pkgrecv(1)` を使用して、再発行するパッケージを指定のディレクトリに raw 形式でダウンロードします。すべてのファイルにはそのハッシュ値で名前が付けられ、マニフェストには `manifest` という名前が付けられます。必要なプロキシ構成を `http_proxy` 環境変数に忘れずに設定してください。
2. `pkgmogrify(1)` を使用して、マニフェストに対して必要な変更を行います。46 ページの「生成されたマニフェストに必要なメタデータを追加する」および第6章「プログラムによるパッケージマニフェストの変更」を参照してください。
   - 発行中に混乱が起きないように、内部パッケージの FMRI からタイムスタンプを削除します。

- すべての署名アクションを削除します。

3. pkglint(1) を使用して結果となるパッケージを検証します。
4. pkgsend(1) を使用してパッケージを再発行します。再発行によって存在している署名がパッケージから取り除かれ、pkg.fmri によって指定されたタイムスタンプが無視されます。警告メッセージを防ぐには、pkgmogrify の手順で署名アクションを削除します。

   パッケージの元のソースに発行する権限がない場合は、pkgrepo(1) を使用してリポジトリを作成したあとで、次のコマンドを使用して、パブリッシャー検索順で元のパブリッシャーの前に新しいパブリッシャーを設定します。

   ```
   $ pkg set-publisher --search-before=original_publisher  new_publisher
   ```

5. 必要に応じて、pkgsign(1) を使用してパッケージに署名します。クライアントのキャッシュ問題を防ぐには、テストの場合であっても、パッケージをインストールする前にパッケージに署名します。第9章「IPS パッケージの署名」を参照してください。

パッケージのメタデータの変更

次の例では、元の pkg.summary の値が「IPS has lots of features」に変更されています。pkgrecv の --raw オプションを使用してパッケージがダウンロードされます。デフォルトでは、最新バージョンのパッケージのみがダウンロードされます。その後、パッケージが新しいリポジトリに再発行されます。

```
$ mkdir republish; cd republish
$ pkgrecv -d . --raw -s http://pkg.oracle.com/solaris/release package/pkg
$ cd package* # The package name contains a '/' and is url-encoded.
$ cd *
$ cat > fix-pkg
# Change the value of pkg.summary
<transform set name=pkg.summary -> edit value '.*' "IPS has lots of features">
# Delete any signature actions
<transform signature -> drop>
# Remove the time stamp from the fmri so that the new package gets a new time stamp
<transform set name=pkg.fmri -> edit value ":20.+" "">
^D
$ pkgmogrify manifest fix-pkg > new-manifest
$ pkgrepo create ./mypkg
$ pkgsend -s ./mypkg publish -d . new-manifest
```

パッケージパブリッシャーの変更

もう 1 つの一般的な使用事例は、新しいパブリッシャー名でパッケージを再発行することです。これが有効な場合の事例として、複数のリポジトリからのパッケージを 1 つのリポジトリに統合することがあげられます。たとえば、統合テストのために、い

くつかの異なる開発チームのリポジトリからのパッケージを 1 つのリポジトリに統合することが必要な場合があります。

新しいパブリッシャー名で再発行するには、前の例に示した `pkgrecv`、`pkgmogrify`、`pkgrepo`、および `pkgsend` の手順を使用します。

次の変換の例では、パブリッシャーを `mypublisher` に変更します。

```
<transform set name=pkg.fmri -> edit value pkg://[^/]+/ pkg://mypublisher/>
```

単純なシェルスクリプトを使用すれば、リポジトリ内のパッケージ全体にわたって繰り返すことができます。リポジトリからの最新のパッケージのみを処理するには、`pkgrecv --newest` からの出力を使用します。

次のスクリプトでは、上記の変換を `change-pub.mog` というファイルに保存し、そのあとで `development-repo` から新しいリポジトリ `new-repo` に再発行し、その途中でパッケージのパブリッシャーを変更します。

```
#!/usr/bin/ksh93
pkgrepo create new-repo
pkgrepo -s new-repo set publisher/prefix=mypublisher
mkdir incoming
for package in $(pkgrecv -s ./development-repo --newest); do
    pkgrecv -s development-repo -d incoming --raw $package
done
for pdir in incoming/*/* ; do
    pkgmogrify $pdir/manifest change-pub.mog > $pdir/manifest.newpub
    pkgsend -s new-repo publish -d $pdir $pdir/manifest.newpub
done
```

このスクリプトは、特定のパッケージのみを選択する、パッケージのバージョン管理スキームに追加の変更を加える、各パッケージを再発行するときの進捗を示す、などのタスクを実行するように変更できます。

♦ ♦ ♦ 付 録 A

パッケージの分類

この付録の内容:

- パッケージに対する分類を指定する方法
- 分類スキームの定義

分類の割り当て

パッケージマネージャーの GUI は、`info.classification` パッケージ属性をスキーム `org.opensolaris.category.2008` とともに使用してパッケージをカテゴリ別に表示します。ユーザーは `pkg search` コマンドを使用して、ある特定の分類を持つパッケージを表示することもできます。

次の例に示すように、分類をパッケージに割り当てるには `set` アクションを使用します。

```
set name=info.classification \
    value="org.opensolaris.category.2008:System/Administration and Configuration"
```

カテゴリとサブカテゴリは、スラッシュ文字で区切ります。属性値に空白が含まれる場合は引用符で囲む必要があります。

次の例に示すように、1 つのパッケージに複数の分類を割り当てることができます。

```
set name=info.classification \
    value="org.opensolaris.category.2008:Meta Packages/Group Packages" \
    value="org.opensolaris.category.2008:Web Services/Application and Web Servers"
```

分類の値

次のカテゴリ値とサブカテゴリ値が定義されています。

Meta Packages

    Group Packages

Incorporations

Applications

Accessories
Configuration and Preferences
Games
Graphics and Imaging
Internet
Office
Panels and Applets
Plug-ins and Run-times
Sound and Video
System Utilities
Universal Access

Desktop (GNOME)

Documentation
File Managers
Libraries
Localizations
Scripts
Sessions
Theming
Trusted Extensions
Window Managers

Development

C
C++
Databases
Distribution Tools
Editors
Fortran
GNOME and GTK+
GNU
High Performance Computing
Java
Objective C
Other Languages
PHP
Perl
Python
Ruby

- Source Code Management
- Suites
- System
- X11

Drivers

- Display
- Media
- Networking
- Other Peripherals
- Ports
- Storage

System

- Administration and Configuration
- Core
- Databases
- Enterprise Management
- File System
- Fonts
- Hardware
- Internationalization
- Libraries
- Localizations
- Media
- Multimedia Libraries
- Packaging
- Printing
- Security
- Services
- Shells
- Software Management
- Text Tools
- Trusted
- Virtualization
- X11

Web Services

- Application and Web Servers
- Communications

♦ ♦ ♦ 付 録 B

IPS を使用して Oracle Solaris OS をパッケージ化する方法

この付録の内容:

- パッケージ FMRI のバージョン文字列の詳細
- Oracle Solaris OS の作業用パッケージセットを定義するために、`incorporate` 依存関係、`facet.version-lock.*` ファセット、`group` 依存関係、およびアクションのタグを使用する方法

Oracle Solaris パッケージのバージョン管理

21 ページの「パッケージ識別子: FMRI」では、ソフトウェア開発のさまざまなモデルをサポートするためにバージョンフィールドをどのように使用できるかを含め、`pkg.fmri` 属性とバージョンフィールドのさまざまな部分について説明しました。このセクションでは、Oracle Solaris OS によるバージョンフィールドの使用方法について説明し、きめの細かいバージョン管理スキームが役立つことがある理由を明らかにします。パッケージでは、Oracle Solaris OS が使用するのと同じバージョン管理スキームに従う必要はありません。

次のサンプルパッケージ FMRI のバージョン文字列の各部の意味は下記のとおりです。

```
pkg://solaris/system/library/storage/suri@0.5.11,5.11-0.175.3.0.0.19.0:20150329T164922Z
```

`0.5.11`

コンポーネントバージョン。Oracle Solaris OS の一部であるパッケージの場合、これは OS `major.minor` バージョンです。ほかのパッケージでは、これはアップストリームバージョンです。たとえば、次の Apache Web サーバーパッケージのコンポーネントバージョンは 2.2.29 です。

```
pkg:/web/server/apache-22@2.2.29,5.11-0.175.3.0.0.19.0:20150329T181125Z
```

`5.11`

リリース。これは、このパッケージが構築される OS リリースを定義するために使用されます。Oracle Solaris 11 のために作成されるパッケージでは、リリースを常に 5.11 にするようにしてください。

`0.175.3.0.0.19.0`

ブランチバージョン。Oracle Solaris パッケージでは、パッケージ FMRI のバージョン文字列のブランチバージョン部分に次の情報が示されます。

`0.175`
: メジャーリリース番号。メジャーまたはマーケティング開発リリースのビルド番号。この例では、`0.175` は Oracle Solaris 11 を示します。

`3`
: 更新リリース番号。この Oracle Solaris リリースの更新リリース番号。更新値は、Oracle Solaris リリースの FCS (First Customer Shipment) では 0、そのリリースの最初の更新では 1、そのリリースの 2 回目の更新では 2 というようになります。この例では、`3` は Oracle Solaris 11.3 を示します。

`0`
: SRU 番号。この更新リリースのサポート・リポジトリの更新 (SRU) 番号。SRU はほぼ 1 か月に 1 回、更新され、バグを修正したり、セキュリティーの問題を修正したり、新しいハードウェアに対するサポートを提供します。SRU に新機能は含まれません。Oracle サポート・リポジトリはサポート契約下のシステムでのみ使用できます。

`0`
: 予約済み。このフィールドは現在 Oracle Solaris パッケージには使用されていません。

`19`
: リリースまたは SRU ビルド番号。SRU のビルド番号、またはメジャーリリースの更新番号。

`0`
: ナイトリービルド番号。個々のナイトリービルドのビルド番号。

パッケージが IDR (Interim Diagnostic Relief) である場合、パッケージ FMRI のブランチバージョンには次の 2 つの追加フィールドが含まれます。IDR は、正式なパッケージ更新が発行されるまで、顧客の問題の診断を支援したり、問題の一時的な解決策を提供したりするためのパッケージ更新です。次の例は `idr824` (FMRI `pkg://solaris/idr824@4,5.11:20131114T034951Z` を含む) についてのもので、`pkg:/system/library@0.5.11-0.175.1.6.0.4.2.824.4` などのパッケージを含んでいます。

`824`

IDR の名前。

4

IDR のバージョン。

`20150329T164922Z`

タイムスタンプ。タイムスタンプは、パッケージが発行されるときに定義されます。

Oracle Solaris 結合パッケージ

Oracle Solaris は一連のパッケージとして配布され、パッケージのサブセットは結合の制約を受けます。結合パッケージの詳細は、『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「incorporation パッケージ」を参照してください。

利用できる結合の一覧を表示するには、次のコマンドを使用します。

```
$ pkg list -as entire \*incorporation
```

結合パッケージは `incorporate` 依存関係を含みます。詳細は、76 ページの「`incorporate` 依存関係」を参照してください。

`pkg:/entire` パッケージは、各結合パッケージへの `require` および `incorporate` の両方の依存関係を含めることにより、ほかの結合を同じビルドに制限する特別な結合です。このようにして、`pkg:/entire` 結合はコアの Oracle Solaris システムパッケージが 1 つのグループとしてアップグレードされるようにソフトウェアサーフェスを定義します。

依存関係の制約の緩和

結合の中には、`facet.version-lock.*` ファセットを使用して、管理者が `pkg change-facet` コマンドを使って指定のパッケージの結合に対する制約を緩和できるようにするものがあります。詳細については、79 ページの「管理者がインストール可能なパッケージバージョンへの制約を緩和できるようにする」を参照してください。

次の一覧では、`pkg:/entire` 結合での `facet.version-lock.*` 定義の例を示します。

注意 - `pkg:/entire` のファセットバージョンをロック解除すると、サポートされないシステムになります。そのようなパッケージは、Oracle サポートからのアドバイスを受けてロック解除するしかありません。

```
depend fmri=consolidation/desktop/gnome-incorporation type=require
depend facet.version-lock.consolidation/desktop/gnome-incorporation=true \
       fmri=consolidation/desktop/gnome-incorporation@0.5.11-0.175.3.10.0.4.0
 type=incorporate
```

```
depend fmri=consolidation/desktop/gnome-incorporation@0.5.11-0.175.3 type=incorporate
depend fmri=consolidation/ips/ips-incorporation type=require
depend facet.version-lock.consolidation/ips/ips-incorporation=true \
       fmri=consolidation/ips/ips-incorporation@0.5.11-0.175.3.13.0.3.0 type=incorporate
depend fmri=consolidation/ips/ips-incorporation@0.5.11-0.175.3 type=incorporate
depend fmri=consolidation/java-8/java-8-incorporation type=require
depend facet.version-lock.consolidation/java-8/java-8-incorporation=true \
       fmri=consolidation/java-8/java-8-incorporation@1.8.0.102.14-0 type=incorporate
depend fmri=consolidation/java-8/java-8-incorporation@1.8.0 type=incorporate
```

Oracle Solaris グループパッケージ

Oracle Solaris では、`group` 依存関係を含むいくつかのグループパッケージを定義しています。これらのグループパッケージを使用すると、共通のパッケージセットをインストールするときに便利です。詳細は、73 ページの「`group` 依存関係」および『Oracle Solaris 11.3 ソフトウェアの追加と更新』 の 「グループパッケージ」を参照してください。

利用できるグループパッケージの一覧を表示するには、次のコマンドを使用します。

```
$ pkg list -as entire group\*
```

属性とタグ

このセクションでは、一般的な属性、Oracle Solaris アクション属性、および Oracle Solaris 属性タグについて説明します。

情報属性

次の属性は正しいパッケージインストールに不可欠ではありませんが、取り決めを共有すると、パブリッシャーとユーザーの間で起きる混乱が少なくなります。

`info.classification`

`info.classification` 属性については、33 ページの「設定アクション」を参照してください。付録A パッケージの分類の分類のリストを参照してください。

`info.keyword`

検索によってこのパッケージが返される追加の用語のリスト。

`info.maintainer`

パッケージを提供するエンティティーについて説明する、人間が読める文字列。この文字列は、個人の名前、個人の名前と電子メール、または組織の名前にします。

`info.maintainer-url`
: パッケージを提供するエンティティーに関連付けられた URL。

`info.upstream`
: ソフトウェアを作成するエンティティーについて説明する、人間が読める文字列。この文字列は、個人の名前、個人の名前と電子メール、または組織の名前にします。

`info.upstream-url`
: パッケージで配布されるソフトウェアを作成するエンティティーに関連付けられた URL。

`info.source-url`
: パッケージのソースコードバンドルへの URL (該当する場合)。

`info.repository-url`
: パッケージのソースコードリポジトリへの URL (該当する場合)。

`info.repository-changeset`
: `info.repository-url` に含まれるソースコードのバージョンのチェンジセット ID。

Oracle Solaris 属性

`org.opensolaris.arc-caseid`
: ARC ケース (Architecture Review Committee)、またはパッケージによって配布されたコンポーネントに関連付けられたケースに関連付けられた 1 つ以上のケース識別子 (PSARC/2008/190 など)。

`org.opensolaris.smf.fmri`
: このパッケージによって配布された SMF サービスを表す 1 つ以上の FMRI。これらの属性は、SMF サービスマニフェストを含むパッケージの `pkgdepend` によって自動的に生成されます。`pkgdepend(1)` のマニュアルページを参照してください。

組織固有の属性

パッケージに追加のメタデータを提供するには、属性名に組織固有の接頭辞を使用します。組織はこの方法を使用して、その組織で開発されたパッケージに追加のメタデータを提供したり、既存のパッケージのメタデータを変更したりできます。既存のパッケージのメタデータを変更するには、そのパッケージが発行されるリポジト

リを制御する必要があります。たとえば、サービス組織は `service.example.com,support-level` または `com.example.service,support-level` という名前の属性を導入して、パッケージとその内容のサポートのレベルを記述できます。

Oracle Solaris タグ

`variant.opensolaris.zone`

非大域ゾーン、大域ゾーン、または非大域ゾーンと大域ゾーンのどちらかにインストールできるパッケージ内のアクションを指定します。詳細については、第10章「非大域ゾーンの処理」を参照してください。