アプリケーションパッケージ開発者ガイド

Part No: E38876
2013 年 1 月

ORACLE

130221@25097

目次

はじめに

『アプリケーションパッケージ開発者ガイド』では、パッケージの設計、作成、および確認を行う手順と、これらの作業に関連する情報を提供しています。また、このガイドでは、パッケージを作成する際に役立つ高度な手法も紹介しています。

注 – この Oracle Solaris のリリースでは、SPARC および x86 系列のプロセッサアーキテクチャーを使用するシステムをサポートしています。サポートされるシステムは、Oracle Solaris OS: Hardware Compatibility Lists に記載されています。このドキュメントでは、プラットフォームにより実装が異なる場合は、それを特記します。

このドキュメントの x86 に関連する用語については、次を参照してください。

- x86 は、64 ビットおよび 32 ビットの x86 互換製品系列を指します。
- x64 は特に 64 ビット x86 互換 CPU を指します。
- 「32 ビット x86」は、x86 をベースとするシステムに関する 32 ビット特有の情報を指します。

サポートされるシステムについては、Oracle Solaris OS: Hardware Compatibility Listsを参照してください。

対象読者

本書は、パッケージの設計および作成を担当するアプリケーション開発者を対象としています。

本書の大部分は、パッケージの開発を担当して間もない開発者に向けられていますが、経験を積んだパッケージ開発者に役立つ情報も含まれています。

内容の紹介

次の表で、本書の各章について説明します。

章のタイトル	章の概要
第 1 章「パッケージの設計」	パッケージのコンポーネントと、パッケージの設計基準について説明します。また、関連するコマンド、ファイル、およびスクリプトについても説明します。
第 2 章「パッケージの構築」	パッケージを作成するプロセスと、必要なタスクについて説明します。また、各タスクの手順についても説明します。
第 3 章「パッケージの機能の拡張 (タスク)」	パッケージにオプションの機能を追加する手順について説明します。
第 4 章「パッケージの確認と転送」	パッケージの整合性を確認する方法と、パッケージを配布媒体に転送する方法について説明します。
第 5 章「パッケージ作成のケーススタディー」	パッケージの作成方法を、さまざまな事例を挙げて説明します。
第 6 章「パッケージの作成のための高度な手法」	パッケージを作成するための高度な手法について説明します。
用語集	このマニュアルで使用される用語について定義します。

関連ドキュメント

System V 用パッケージの作成に関する追加情報については、次の書籍を参照してください。これらの書籍は、書店で購入することができます。

- 『System V Application Binary Interface』
- 『System V Application Binary Interface - SPARC Processor Supplement』
- 『System V Application Binary Interface - Intel386 Processor Supplement』

Oracle サポートへのアクセス

Oracle のお客様は、My Oracle Support を通じて電子的なサポートを利用することができます。詳細は、`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` を参照してください。聴覚に障害をお持ちの場合は、`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` を参照してください。

表記上の規則

次の表では、このマニュアルで使用される表記上の規則について説明します。

表 P–1 表記上の規則

字体	説明	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 machine_name% you have mail.
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	machine_name% **su** Password:
aabbcc123	Placeholder: 実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
AaBbCc123	書名、新しい単語、および強調する単語を示します。	『ユーザーズガイド』の第 6 章を参照してください。 キャッシュは、ローカルに格納されるコピーです。 ファイルを保存しないでください。 注: いくつかの強調された項目は、オンラインでは太字で表示されます。

コマンド例のシェルプロンプト

Oracle Solaris OS に含まれるシェルで使用する、UNIX のデフォルトのシステムプロンプトとスーパーユーザープロンプトを次に示します。コマンド例に示されるデフォルトのシステムプロンプトは、Oracle Solaris のリリースによって異なります。

表 P–2 シェルプロンプト

シェル	プロンプト
Bash シェル、Korn シェル、および Bourne シェル	$
Bash シェル、Korn シェル、および Bourne シェルのスーパーユーザー	#
C シェル	machine_name%

表 P–2 シェルプロンプト (続き)

シェル	プロンプト
C シェルのスーパーユーザー	machine_name#

◆ ◆ ◆ 第 1 章

パッケージの設計

パッケージを構築する前に、作成する必要があるファイルと、実行する必要があるコマンドについて理解しておく必要があります。また、アプリケーションソフトウェアおよび要件および顧客の需要についても考慮する必要があります。顧客が管理者となり、パッケージをインストールします。この章では、パッケージを構築する前に理解および考慮しておくべきファイル、コマンド、および条件について説明します。

この章で説明する情報は次のとおりです。

パッケージタスクの場所

パッケージの構築および確認のための作業を特定するには、次のタスクマップを使用します。

パッケージとは

アプリケーションソフトウェアは、パッケージと呼ばれる単位で配信されます。パッケージは、ソフトウェア製品に必要なファイルおよびディレクトリの集合です。通常、パッケージは、アプリケーションコードの開発が完了したあとでアプリケーション開発者が設計して構築します。配布媒体に転送しやすいように、ソフトウェア製品を 1 つ以上のパッケージに分けて構築する必要があります。これにより、管理者は、ソフトウェア製品を大量に生成し、インストールできるようになります。

パッケージは、定義された形式でのファイルおよびディレクトリの集合です。このフォーマットは、アプリケーションバイナリインタフェース (ABI) に準拠します。ABI は、System V インタフェース定義を補足するものです。

パッケージコンポーネント

パッケージのコンポーネントは 2 つのカテゴリに分類されます。

- パッケージオブジェクトは、インストールされるアプリケーションファイルです。
- 制御ファイル は、パッケージをインストールする方法、場所、および条件を制御します。

また、制御ファイルは、情報ファイルとインストールスクリプトという 2 つのカテゴリに分類されます。制御ファイルには、必須のものとオプションのものがあります。

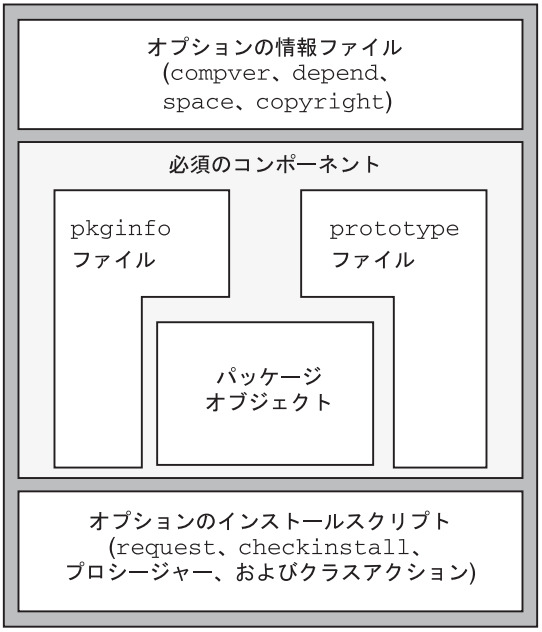
アプリケーションをパッケージ化するには、パッケージを構成する必須コンポーネントをまず作成し、次にオプションのコンポーネントを作成します。そのあと、`pkgmk` コマンドを使用して、パッケージを構築できます。

パッケージを構築するには、次のものを用意します。

- パッケージオブジェクト (アプリケーションソフトウェアのファイルとディレクトリ)
- 必須となる 2 つの情報ファイル (`pkginfo` ファイルと `prototype` ファイル)
- オプションの情報ファイル
- オプションのインストールスクリプト

次の図に、パッケージのコンテンツを示します。

図 1–1 パッケージのコンテンツ



必須のパッケージコンポーネント

パッケージを構築する前に、次のコンポーネントを作成する必要があります。

- パッケージオブジェクト

  このコンポーネントはアプリケーションを構成します。パッケージオブジェクトは、次の項目で構成されます。

  - ファイル (実行可能ファイルまたはデータファイル)
  - ディレクトリ
  - 名前付きパイプ
  - リンク
  - デバイス

- `pkginfo` ファイル

  `pkginfo` ファイルは、パラメータ値を定義する必須のパッケージ情報ファイルです。パラメータ値には、パッケージの省略名、正式名称、およびアーキテクチャーが含まれます。詳細については、26 ページの「`pkginfo` ファイルの作成」と `pkginfo(4)` のマニュアルページを参照してください。

注 – pkginfo(1) のマニュアルページは、2 ページで構成されています。1 ページ目では、インストールされるパッケージに関する情報を表示する、セクション 1 のコマンドについて説明します。2 ページ目では、パッケージの特性を記載した、セクション 4 のファイルについて説明します。マニュアルページにアクセスする場合、マニュアルページの該当するセクションを指定する必要があります。次に入力例を示します。 `man -s 4 pkginfo`

- prototype ファイル

  prototype ファイルは、必須のパッケージ情報ファイルであり、パッケージのコンポーネントの一覧が記述されます。パッケージオブジェクト、情報ファイル、およびインストールスクリプトには、それぞれ 1 つのエントリが存在します。エントリは、場所、属性、ファイルタイプなど、各コンポーネントを説明するいくつかの情報フィールドで構成されます。詳細については、31 ページの「prototype ファイルの作成」 と prototype(4) のマニュアルページを参照してください。

オプションのパッケージコンポーネント

パッケージ情報ファイル

パッケージには、次に示す 4 つのオプションのパッケージ情報ファイルを含めることができます。

- compver ファイル

  作成するパッケージのバージョンと互換性がある、以前のバージョンのパッケージを定義します。

- depend ファイル

  パッケージと特別な関連があるほかのパッケージを示します。

- space ファイル

  ターゲット環境で必要なディスク容量を定義します。prototype ファイルで定義されたオブジェクトが必要とする容量よりも大きなサイズを指定します。たとえば、インストール時にファイルが動的に作成される場合、追加の容量が必要になることがあります。

- copyright ファイル

  パッケージのインストール時に表示される、著作権に関するメッセージのテキストを定義します。

各パッケージ情報ファイルには、prototype ファイル内にエントリが必要です。パッケージ情報ファイルの作成の詳細については、53 ページの「情報ファイルの作成」を参照してください。

パッケージインストールスクリプト

インストールスクリプトは必須ではありません。ただし、インストールスクリプトを用意すると、パッケージのインストール時にカスタマイズしたアクションを実行できます。インストールスクリプトには、次の特性があります。

- スクリプトは、Bourne シェルコマンドで構成されます。
- スクリプトのファイル権限は 0644 に設定するようにしてください。
- スクリプトには、シェル識別子 (`#! /bin/sh`) を含める必要はありません。

スクリプトには、次の 4 つのタイプがあります。

- `request` スクリプト

  `request` スクリプトは、パッケージをインストールしている管理者に入力を要求します。

- `checkinstall` スクリプト

  `checkinstall` スクリプトは、特別なファイル システム検査を実行します。

  注 – `checkinstall` スクリプトは、Solaris 2.5 リリースおよび互換性のあるリリースでのみ使用できます。

- 手続きスクリプト

  手続きスクリプトは、パッケージのインストールおよび削除時の特定の時点で発生するアクションを定義します。次に示す定義済みの名前を指定して、4 種類の手続きスクリプトを作成できます: `preinstall`、`postinstall`、`preremove`、および `postremove` 。

- クラスアクションスクリプト

  クラスアクションスクリプトは、オブジェクトのグループで実行される一連のアクションを定義します。

インストールスクリプトの詳細については、59 ページの「インストールスクリプトの作成」を参照してください。

パッケージを構築する前の考慮事項

パッケージを構築する前に、製品を 1 つ以上のパッケージで構成するかどうかを決定する必要があります。小さいパッケージを数多くインストールする場合、大きなパッケージを 1 つインストールするよりも時間がかかります。単一のパッケージを作成することが良い選択ではありますが、いつでも可能というわけではありません。複数のパッケージを構築することに決めた場合、アプリケーション コードを分割する方法を決定する必要があります。このセクションでは、パッケージの構築を計画するときに使用する条件の一覧を示します。

パッケージ化の条件は、各条件の間でトレードオフが生じることがよくあります。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。これらの条件を重要度の高いものから順に紹介します。ただし、この順序は、大まかな目安であり、状況に応じて変更すべきものです。条件はどれも重要ですが、適切なパッケージのセットを作成するためにこれらの条件を最適化するかどうかはユーザーで判断してください。

パッケージ設計の詳細な手法については、第 6 章「パッケージの作成のための高度な手法」を参照してください。

パッケージをリモートでインストールできるようにする

すべてのパッケージを、リモートでインストールできるようにしてください。パッケージをリモートでインストールできれば、管理者がクライアントシステムでインストールできるので、pkgadd コマンドを実行するルート (/) ファイルシステムにパッケージをインストールする必要はなくなります。

クライアントサーバー構成の最適化

パッケージの配置時に、スタンドアロンシステムやサーバーなど、各種のシステムソフトウェア構成を検討します。構成タイプごとにインストールが最適化されるよう、パッケージを適切に設計し、影響を受けるファイルを分割します。たとえば、ルート (/) ファイルシステムと /usr ファイルシステムの内容を分割すると、サーバー構成のサポートが容易に行えるようになります。

機能ごとのパッケージ

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、UFS を含むパッケージは、すべての UFS 機能を含み、UFS のバイナリにのみ制限されるべきです。

パッケージは、顧客の観点から見て、機能単位ごとに分類するようにしてください。

ロイヤリティの適用規定に沿ったパッケージ

契約上の合意のためにロイヤリティーの支払いが必要となるコードは、専用のパッケージまたはパッケージグループに配置します。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。

依存するシステムごとのパッケージ

システムに依存するバイナリは専用のパッケージに格納します。たとえば、カーネルコードは、各実装アーキテクチャーを個々のパッケージインスタンスで構成して、専用のパッケージに格納するようにしてください。また、この規則は、別のアーキテクチャーのバイナリにも適用されます。たとえば、SPARC システムのバイナリと x86 システムのバイナリは、それぞれ別のパッケージに格納するようにしてください。

パッケージ内での重複の排除

パッケージを作成するとき、重複するファイルはできるだけ削除します。ファイルが無駄に重複すると、サポートとバージョン管理が困難になります。複数のパッケージが製品に含まれる場合、それぞれの内容を繰り返し比較し、重複するファイルがないか調べてください。

ローカライズ版ごとのパッケージ

ローカライズ版に固有の項目は、専用のパッケージに格納するようにしてください。理想的なパッケージモデルは、製品のローカライズ版をロケールごとに 1 つのパッケージとして配信するものです。残念ながら、組織的な境界が、機能的な境界や製品的な境界と両立しない場合があります。

また、各国で共有される標準設定もパッケージで配信できます。このようにパッケージを設計すると、ローカライズ版の変更に必要なファイルを識別し、ローカライズ版パッケージの配信形式を標準化できます。

コマンド、ファイル、およびスクリプトのパッケージ化

このセクションでは、パッケージの操作時に使用する可能性があるコマンド、ファイル、およびスクリプトについて説明します。これらは、各々が実行する固有のタスクとともに、マニュアルページで説明されており、本書の中でも詳細に説明されています。

次の表に、パッケージについての情報を作成、確認、インストール、および取得するのに役立つコマンドを示します。

表 1–1 パッケージ化のためのコマンド

タスク	コマンド/マニュアルページ	説明	参照先
パッケージの作成	pkgproto(1)	pkgmk コマンドに入力するための prototype ファイルを生成します	38 ページの「例—pkgproto コマンドを使用した prototype ファイルの作成」
pkgmk(1)	インストール可能なパッケージの作成	46 ページの「パッケージの構築」	
パッケージのインストール、削除、および転送	pkgadd(1M)	ソフトウェアパッケージをシステムにインストールします	90 ページの「ソフトウェアパッケージのインストール」
pkgask(1M)	応答を request スクリプトに格納します	64 ページの「request スクリプトの設計規則」	
pkgtrans(1)	パッケージを配布媒体にコピーします	100 ページの「配布媒体へのパッケージの転送」	
pkgrm(1M)	パッケージをシステムから削除します	99 ページの「パッケージの削除」	
パッケージに関する情報の取得	pkgchk(1M)	ソフトウェアパッケージの整合性を確認します	92 ページの「パッケージの整合性の確認」
pkginfo(1)	ソフトウェアパッケージの情報を表示します	96 ページの「pkginfo コマンド」	
pkgparam(1)	パッケージのパラメータ値を表示します	94 ページの「pkgparam コマンド」	
インストールされたパッケージの変更	installf(1M)	インストール済みのパッケージに新規のパッケージオブジェクトを組み込みます	70 ページの「手続きスクリプトの設計規則」と第 5 章「パッケージ作成のケーススタディー」
removef(1M)	インストール済みのパッケージからパッケージオブジェクトを削除します	70 ページの「手続きスクリプトの設計規則」	

次の表に、パッケージの構築に役立つ情報ファイルを示します。

表 1–2 パッケージ情報ファイル

ファイル	説明	参照先
admin(4)	パッケージをインストールするためのデフォルトのファイル	132 ページの「管理デフォルトファイル」

表 1–2 パッケージ情報ファイル (続き)

ファイル	説明	参照先
compver(4)	パッケージ互換性ファイル	53 ページの「パッケージの依存関係の定義」
copyright(4)	パッケージの著作権に関する情報ファイル	56 ページの「著作権に関するメッセージの書き込み」
depend(4)	パッケージ依存関係ファイル	53 ページの「パッケージの依存関係の定義」
pkginfo(4)	パッケージ特性ファイル	26 ページの「pkginfo ファイルの作成」
pkgmap(4)	パッケージコンテンツについての説明ファイル	46 ページの「pkgmap ファイル」
prototype(4)	パッケージ情報ファイル	31 ページの「prototype ファイルの作成 」
space(4)	パッケージのディスク容量要件に関するファイル	57 ページの「ターゲットシステムでの追加領域の予約」

次の表では、オプションのインストールスクリプトについて示します。これらのインストールスクリプトはユーザーが書き込むことができ、パッケージのインストール条件および方法に影響を与えます。

表 1–3 パッケージインストールスクリプト

スクリプト	説明	参照先
request	インストーラから情報を要求します	64 ページの「request スクリプトの書き込み」
checkinstall	ファイルシステムデータを収集します	66 ページの「checkinstall スクリプトでのファイルシステムデータの収集」
preinstall	クラスをインストールする前に、いずれかのカスタムインストール要件を実行します	69 ページの「手続きスクリプトの書き込み」
postinstall	ボリュームをすべてインストールしたあとに、カスタムインストール要件をすべて実行します	69 ページの「手続きスクリプトの書き込み」
preremove	クラスを削除する前に、いずれかのカスタム削除要件を実行します	69 ページの「手続きスクリプトの書き込み」
postremove	クラスをすべて削除したあとに、カスタム削除要件をすべて実行します	69 ページの「手続きスクリプトの書き込み」

表 1–3 パッケージインストールスクリプト (続き)

スクリプト	説明	参照先
クラスアクション	オブジェクトの特定のグループで一連のアクションを実行します	71 ページの「クラスアクションスクリプトの書き込み」

◆ ◆ ◆ 第 2 章

2 パッケージの構築

この章では、パッケージの構築に含まれるプロセスとタスクについて説明します。これらのタスクには、必須のものとオプションのものがあります。この章では、必須のタスクについて詳細に説明します。パッケージに機能を追加するためのオプションのタスクについては、第 3 章「パッケージの機能の拡張 (タスク)」と第 6 章「パッケージの作成のための高度な手法」を参照してください。

この章で説明する情報は次のとおりです。

パッケージの構築プロセス (タスクマップ)

パッケージを構築する際のプロセスについて、表 2–1 で説明します。特にパッケージの構築に不慣れな方は、この表を参考にしてください。最初の 4 つのタスクは、表示されている順序のとおりに行う必要はありませんが、この順序にしたがうとパッケージの構築が容易になります。パッケージの設計に熟練したユーザーの場合、目的に合わせてこれらのタスクの順序を入れ替えてもかまいません。

パッケージの設計に熟練したユーザーは、make コマンドおよびメイクファイルを使用してパッケージ構築プロセスを自動化できます。詳細については、make(1S) のマニュアルページを参照してください。

表 2–1 パッケージの構築プロセス (タスクマップ)

タスク	説明	参照先
1. pkginfo ファイルの作成	パッケージの特性を記述するための pkginfo ファイルを作成します。	29 ページの「pkginfo ファイルの作成方法」
2. パッケージコンテンツの編成	パッケージコンポーネントを階層ディレクトリ構造に配置します。	30 ページの「パッケージのコンテンツの編成」
3. (オプション) 情報ファイルの作成	パッケージの依存関係の定義、著作権に関するメッセージの記載、およびターゲットシステムでの追加領域の確保を行います。	第 3 章「パッケージの機能の拡張 (タスク)」
4. (オプション) インストールスクリプトの作成	パッケージのインストールと削除のプロセスをカスタマイズします。	第 3 章「パッケージの機能の拡張 (タスク)」
5. prototype ファイルの作成	prototype ファイルにパッケージ内のオブジェクトを記述します。	31 ページの「prototype ファイルの作成」
6. パッケージの構築	pkgmk コマンドを使用してパッケージを構築します。	46 ページの「パッケージの構築」
7. パッケージの確認および転送	パッケージの整合性を確認してから、パッケージを配布媒体にコピーします。	第 4 章「パッケージの確認と転送」

パッケージ環境変数

必須の情報ファイルである pkginfo および prototype では、変数を使用できます。また、パッケージの構築に使用する pkgmk コマンドには、オプションを使用することもできます。これらのファイルおよびコマンドについてこの章で説明されているように、変数に関しては、よりコンテキストに依存した情報が提供されます。ただし、パッケージの構築を始める前に、ほかにどのような変数があるか、またその変数がパッケージの構築にどのような影響を与えるかについて理解しておくことをおすすめします。

変数には次の 2 種類があります。

- 構築変数

  構築変数は小文字で始まり、*pkgmk* コマンドを使用してパッケージを構築する際、構築時に評価されます。

- インストール変数

  インストール変数は大文字で始まり、pkgadd コマンドを使用してパッケージをインストールする際、インストール時に評価されます。

環境変数の使用に関する一般的な規則

pkginfo ファイルでは、変数の定義は *PARAM=value* という形式になります。*PARAM* の最初の文字は大文字です。これらの変数は、インストール時にのみ評価されます。これらの変数を評価できない場合、pkgadd コマンドはエラーを返して異常終了します。

prototype ファイルでは、変数の定義は !*PARAM=value* または $*variable* の形式にできます。*PARAM* と *variable* はともに、大文字または小文字のどちらからでも始めることができます。構築時に値が既知の変数だけが評価されます。*PARAM* または *variable* が、構築変数またはインストール変数であり、その値が構築時に不明な場合、pkgmk コマンドはエラーを返して異常終了します。

また、*PARAM=value* は、オプションとして pkgmk コマンドに組み込むことができます。このオプションは、有効範囲がパッケージ全体に対してグローバルであるという点を除いて、prototype ファイルの中に指定されたものと同じように機能します。prototype ファイルでの !*PARAM=value* の定義は、そのファイルと、そのファイルで定義されているパッケージの部分に対してローカルです。

PARAM がインストール変数であり、*variable* が既知の値を持つインストール変数または構築変数である場合、その定義が pkgmk コマンドによって pkginfo ファイルに挿入され、その定義をインストール時に使用できるようになります。ただし、pkgmk コマンドは、prototype ファイルに指定されたパス名にある *PARAM* を評価しません。

パッケージ環境変数のサマリー

次の表に、変数の指定形式、場所、および有効範囲をまとめています。

表 2–2 パッケージ環境変数のサマリー

変数が定義されている場所	変数の定義形式	定義される変数の種類	変数が評価されるタイミング	変数が評価される場所	変数が置換される可能性のある項目
pkginfo ファイル	*PARAM=value*	構築	構築時に無視される	該当なし	なし
インストール	インストール時	pkgmap ファイル内	*owner*、*group*、*path*、またはリンクターゲット		
prototype ファイル	!*PARAM=value*	構築	構築時	prototype ファイルおよび任意の組み込みファイル内	*mode*、*owner*、*group*、または*path*

表 2–2 パッケージ環境変数のサマリー (続き)

変数が定義されている場所	変数の定義形式	定義される変数の種類	変数が評価されるタイミング	変数が評価される場所	変数が置換される可能性のある項目
インストール	構築時	prototype ファイルおよび任意の組み込みファイル内	!search コマンドと !command コマンドの使用時のみ		
pkgmk コマンド行	*PARAM=value*	構築	構築時	prototype ファイル内	*mode*、*owner*、*group*、または*path*
インストール	構築時	prototype ファイル内	!search コマンド使用時のみ		
インストール時	pkgmap ファイル内	*owner*、*group*、*path*、またはリンクターゲット			

pkginfo ファイルの作成

pkginfo ファイルは、パッケージの特性を説明するASCII ファイルです。インストールの流れを制御するのに役立つ情報を提供します。

pkginfo ファイルの各エントリは、*PARAM=value* の形式を使用してパラメータの値を設定するための行です。*PARAM* には、pkginfo(4) のマニュアルページで説明されている標準のパラメータのいずれかを指定できます。パラメータの指定には、必要な順序はありません。

注 – それぞれの *value* は、単一引用符または二重引用符で囲むことができます (例: '*value*' または "*value*")。シェル環境に固有であるとみなされる文字が *value* に含まれる場合、引用符を使用するようにしてください。本書の例およびケーススタディーでは、引用符を使用しません。二重引用符を使用する例については、pkginfo(4) のマニュアルページを参照してください。

また、独自のパッケージパラメータを作成するには、pkginfo ファイル内でパッケージパラメータに値を割り当てます。パラメータは、大文字で始めて、その後ろに大文字または小文字を続けます。大文字の場合、パラメータ (変数) は、構築時ではなく、インストール時に評価されます。逆に、小文字の場合は構築時に評価されます。インストール変数と構築変数の違いについては、24 ページの「パッケージ環境変数」を参照してください。

注 – パラメータ値のあとにある空白は無視されます。

---

pkginfo ファイルには、 PKG、 NAME、 ARCH、 VERSION、および CATEGORY という 5 つのパラメータを定義します。PATH、 PKGINST、および INSTDATE の各パラメータは、パッケージの構築時にソフトウェアによって自動的に挿入されます。これら 8 つのパラメータは変更しないでください。残りの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。

パッケージインスタンスの定義

同一のパッケージに、異なるバージョンを含めたり、異なるアーキテクチャーとの互換性を持たせたり、あるいはその両方を組み合わせることができます。パッケージの各バリエーションは、パッケージインスタンスと呼ばれています。パッケージインスタンスは、pkginfo ファイル内で PKG、 ARCH、および VERSION の各パラメータ定義を組み合わせて決定します。

pkgadd コマンドは、インストール時にパッケージ識別子を各パッケージインスタンスに割り当てます。パッケージ識別子は、たとえば SUNWadm.2 のように、数値の接尾辞を含むパッケージの省略名です。この識別子は、同じパッケージのインスタンスを含め、その他のパッケージとパッケージインスタンスを区別します。

パッケージの省略名の定義 (PKG)

パッケージの省略名は、パッケージの名前を短くしたものであり、pkginfo ファイル内で PKG パラメータによって定義されています。パッケージの省略名には、次の特性を持たせてください。

- 省略名は英数字で構成します。最初の文字を数字にすることはできません。
- 省略名は 32 文字の長さを超えることはできません。
- 省略名には、 予約された省略名である install、 new、および all は使用できません。

注 – 最初の 4 文字は、会社に特有のものでなければなりません。たとえば、Sun Microsystems によって構築されたパッケージはすべて、パッケージの省略名の最初の 4 文字に「SUNW」の文字を含みます。

---

pkginfo ファイル内のパッケージ省略名エントリの一例は PKG=SUNWcadap です。

パッケージアーキテクチャーの指定 (ARCH)

pkginfo ファイル内の ARCH パラメータは、パッケージに関連付けるアーキテクチャーを特定します。アーキテクチャー名の最大長は、英数字で 16 文字です。パッケージが複数のアーキテクチャーに関連付けられている場合、アーキテクチャーをコンマ区切りのリストで指定します。

pkginfo ファイル内でパッケージアーキテクチャーを指定する例を次に示します。

```
ARCH=sparc
```

パッケージの命令セットアーキテクチャーの指定 (SUNW_ISA)

pkginfo ファイル内の SUNW_ISA パラメータは、Sun Microsystems パッケージに関連付けられている命令セットアーキテクチャーを特定します。パラメータの値は次のとおりです。

- 64 ビットオブジェクトを含むパッケージの場合: sparcv9
- 32 ビットオブジェクトを含むパッケージの場合: sparc

たとえば、64 ビットオブジェクトを含むパッケージの場合、pkginfo ファイル内の SUNW_ISA 値は次のようになります。

```
SUNW_ISA=sparcv9
```

SUNW_ISA が設定されていない場合、パッケージの命令セットアーキテクチャーはデフォルトで ARCH パラメータの値に設定されます。

パッケージのバージョンの指定 (VERSION)

pkginfo ファイル内の VERSION パラメータは、パッケージのバージョンを特定します。バージョンの最大長は、ASCII 文字で 256 文字です。また、バージョンは、左括弧で始めることはできません。

pkginfo ファイル内でバージョンを指定する例を次に示します。

```
VERSION=release 1.0
```

パッケージ名の定義 (NAME)

パッケージ名は、パッケージの完全な名前であり、pkginfo ファイル内の NAME パラメータによって定義されます。

システム管理者は多くの場合、パッケージのインストールが必要であるかどうかを決定するためにパッケージ名を使用するので、明瞭かつ簡潔で完全なパッケージ名を記述することが重要です。パッケージ名は、次の条件を満たすようにしてください。

- パッケージが必要なタイミングを記述します。たとえば、特定のコマンドまたは機能を提供する場合や、特定のハードウェアに対してパッケージが必要かどうかを記述します。
- デバイスドライバの開発など、パッケージを使用する目的を記述します。

- パッケージの省略名のニーモニックについての説明を記載します。これには、省略名が説明の短縮形であることを示すキーワードを使用します。たとえば、SUNWbnuu というパッケージの省略名に対するパッケージ名は、"BasicNetworking UUCP Utilities, (Usr)" となります。
- パッケージをインストールするパーティションを記述します。
- 業界での意味に沿うように用語を使用します。
- 256 文字の制限を最大限に活用します。

次に示すのは、pkginfo ファイル内で定義されるパッケージ名の例です。

```
NAME=Chip designers need CAD application software to design
abc chips.  Runs only on xyz hardware and is installed in the
usr partition.
```

パッケージカテゴリの定義 (CATEGORY)

pkginfo ファイル内の CATEGORY パラメータは、パッケージが属するカテゴリを指定します。少なくとも、パッケージは system または application のカテゴリに属します。カテゴリ名は、英数字で構成されます。カテゴリ名は最大長が 16 文字であり、大文字と小文字が区別されません。

パッケージが複数のカテゴリに属する場合、カテゴリをコンマ区切りのリストで指定します。

次に示すのは、pkginfo ファイル内で CATEGORY を指定する例です。

```
CATEGORY=system
```

▼ pkginfo ファイルの作成方法

1. 任意のテキストエディタを使用して、**pkginfo** という名前のファイルを作成します。
   このファイルは、システムの任意の場所で作成できます。

2. ファイルを編集し、**5** つの必須パラメータを定義します。
   5 つの必須パラメータは、 PKG、NAME、ARCH、VERSION、および CATEGORY です。これらのパラメータの詳細については、26 ページの「pkginfo ファイルの作成」を参照してください。

3. オプションのパラメータをファイルに追加します。
   独自のパラメータを作成します。また、標準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。

4. ファイルを保存してエディタを終了します。

例 2–1 pkginfo ファイルの作成

次の例では、有効な pkginfo ファイルのコンテンツを示しています。ここでは、5 つの必須パラメータを定義し、BASEDIR パラメータを指定しています。BASEDIR パラメータは、33 ページの「*path* フィールド」で詳細に説明されています。

```
PKG=SUNWcadap
NAME=Chip designers need CAD application software to design abc chips.
Runs only on xyz hardware and is installed in the usr partition.
ARCH=sparc
VERSION=release 1.0
CATEGORY=system
BASEDIR=/opt
```

参照 30 ページの「パッケージのコンテンツの編成方法」を参照してください。

パッケージのコンテンツの編成

インストールあとのターゲットシステム上でのパッケージオブジェクトの構造を模倣した階層ディレクトリ構造にパッケージオブジェクトを編成します。prototype ファイルを作成する前にこの手順を行うと、prototype ファイルの作成時に時間と労力を節約できます。

▼ パッケージのコンテンツの編成方法

1. 作成が必要なパッケージの数と、各パッケージに配置するパッケージオブジェクトを決定します。

   この手順を完了できるようにするには、17 ページの「パッケージを構築する前の考慮事項」を参照してください。

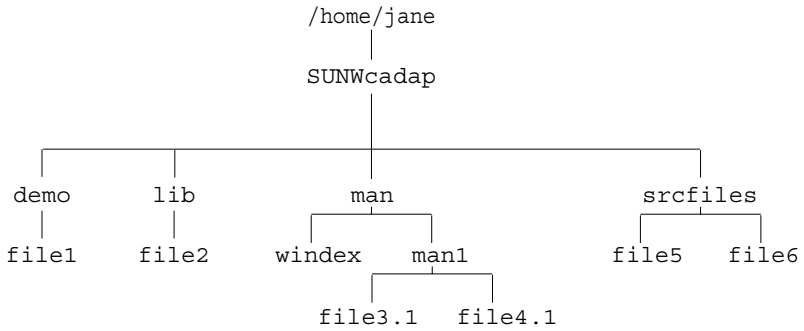
2. 構築する必要がある各パッケージにディレクトリを作成します。

   このディレクトリは、システムの任意の場所で構築し、任意の名前を付けることができます。この章の例では、パッケージディレクトリはパッケージの省略名と同じ名前を持つと仮定しています。

   ```
   $ cd /home/jane
   $ mkdir SUNWcadap
   ```

3 対応するパッケージディレクトリの下にあるディレクトリ構造に各パッケージ内のパッケージオブジェクトを編成します。このディレクトリ構造は、ターゲットシステム上でのパッケージオブジェクトの構造を模倣するようにしてください。

たとえば、CAD アプリケーションパッケージの SUNWcadap には、次のディレクトリ構造が必要です。



4 情報ファイルを保管する場所を決定します。適切な場合、ディレクトリを作成してファイルを 1 か所に保管します。

次の例では、29 ページの「pkginfo ファイルの作成方法」にある pkginfo ファイルの例が Jane のホームディレクトリに作成されていると仮定しています。

```
$ cd /home/jane
$ mkdir InfoFiles
$ mv pkginfo InfoFiles
```

参照 44 ページの「pkgproto コマンドを使用して prototype ファイルを作成する方法」を参照してください。

prototype ファイルの作成

prototype ファイルは、パッケージ内のオブジェクトに関する情報を指定するのに使用する ASCII ファイルです。prototype ファイルの各エントリには、データファイル、ディレクトリ、ソースファイル、実行可能オブジェクトなど、単一のオブジェクトが記述されます。prototype ファイルのエントリは、空白で区切られた、複数のフィールドによる情報で構成されます。これらのフィールドは、特定の順序で表示されるようにしてください。注釈行はポンド記号 (#) で始まり、無視されます。

prototype ファイルは、テキストエディタまたは pkgproto コマンドを使用して作成できます。このファイルをはじめて作成する場合、以前作成したディレクトリ階層に基づいてファイルを作成する pkgproto コマンドで作成したほうが簡単でしょう。30 ページの「パッケージのコンテンツの編成」で説明されているとおりにファイルを編成していない場合、任意のテキストエディタで prototype ファイルを最初から作成する手間がかかります。

prototype ファイルの形式

prototype ファイルでの各行の形式を次に示します。

partftypeclasspathmajorminormodeownergroup

part	パッケージオブジェクトを細かくグループ化できる、オプションの数値フィールドです。デフォルト値は part 1 です。
ftype	オブジェクトの種類を指定する、1 文字のフィールドです。32 ページの「ftype フィールド」を参照してください。
クラス	オブジェクトが属するインストールクラスです。33 ページの「*class* フィールド」を参照してください。
path	パッケージオブジェクトがターゲットシステムに存在する場所を示す、絶対パス名または相対パス名です。33 ページの「*path* フィールド」を参照してください。
major	ブロック特殊デバイスまたは文字特殊デバイスのためのメジャーデバイス番号です。
minor	ブロック特殊デバイスまたは文字特殊デバイスのためのマイナーデバイス番号です。
mode	オブジェクトの 8 進数モードです (例: 0644)。36 ページの「*mode* フィールド」を参照してください。
owner	オブジェクトの所有者です (例: `bin` または `root`)。37 ページの「*owner* フィールド」を参照してください。
group	オブジェクトが属するグループです (例: `bin` または `sys`)。37 ページの「*group* フィールド」を参照してください。

通常では、*ftype* フィールド、*class* フィールド、*path* フィールド、*mode* フィールド、*owner* フィールド、および *group* フィールドだけを定義します。これらのフィールドについては、次のセクションで説明します。これらのフィールドに関する追加情報については、`prototype(4)` のマニュアルページを参照してください。

ftype フィールド

ftype (ファイルタイプ) フィールドは 1 文字のフィールドであり、パッケージオブジェクトのファイルタイプを指定します。次の表は、有効なファイルタイプについて説明しています。

表 2–3 prototype ファイルの有効なファイルタイプ

ファイルタイプのフィールド値	ファイルタイプの説明
f	標準の実行可能ファイルまたはデータファイル
e	インストールまたは削除時に編集するファイル (複数のパッケージで共有する可能性があります)
v	揮発性ファイル (このコンテンツは変更することが想定されています。例: ログファイル)
d	ディレクトリ
x	このパッケージでのみアクセスできる排他的なディレクトリ (登録されていないログやデータベース情報を含む可能性があります)
l	リンクファイル
p	名前付きパイプ
c	文字特殊デバイス
b	ブロック特殊デバイス
i	情報ファイルまたはインストールスクリプト
s	シンボリックリンク

class フィールド

class フィールドは、オブジェクトが属するクラスを指定します。クラスの使用は、オプションのパッケージ設計機能です。この機能は、71 ページの「クラスアクションスクリプトの書き込み」で詳細に説明されています。

クラスを使用しない場合、オブジェクトは none クラスに属します。pkgmk コマンドを実行してパッケージを構築すると、コマンドによって CLASSES=none パラメータが pkginfo ファイルに挿入されます。ファイルタイプ i が指定されたファイルには、空白の *class* フィールドが 1 つ含まれます。

path フィールド

path フィールドは、パッケージオブジェクトがターゲットシステムに存在する場所を定義するために使用されます。絶対パス名 (例: /usr/bin/mail) または相対パス名(例: bin/mail) を使用して場所を示すことができます。絶対パス名を使用すると、ターゲットシステム上のオブジェクトの場所はパッケージによって定義され、変更できません。パッケージオブジェクトに相対パス名が含まれる場合、オブジェクトが再配置可能になります。

再配置可能オブジェクトには、ターゲットシステム上の絶対パスの場所は必要ありません。代わりに、オブジェクトの場所はインストールプロセス中に決定されます。

パッケージのオブジェクトのすべてまたは一部を再配置可能として定義できます。インストールスクリプトを記述したり、prototype ファイルを作成したりする前に、パッケージオブジェクトの場所を固定 (/etc 内の起動スクリプトなど) するか、または再配置可能にするかを決定します。

再配置可能オブジェクトには、集合的再配置可能と個別再配置可能の 2 種類があります。

集合的再配置可能オブジェクト

集合的再配置可能オブジェクトは、ベースディレクトリと呼ばれる共通のインストールベースに対して配置されます。ベースディレクトリは、BASEDIR パラメータを使用して pkginfo ファイルに定義されます。たとえば、tests/generic という名前の prototype ファイル内にある再配置可能オブジェクトは、pkginfo ファイルにデフォルトの BASEDIR パラメータが定義されていることを必要とします。例:

```
BASEDIR=/opt
```

この例の場合、オブジェクトをインストールすると、そのオブジェクトは /opt/tests/generic ディレクトリに配置されます。

注 – /opt ディレクトリは、基本 Oracle Solaris ソフトウェアの一部ではないソフトウェアの配信先となる可能性のある唯一のディレクトリです。

できる限り、集合的再配置可能オブジェクトを使用してください。通常、ほとんどのパッケージは、絶対パスで指定された一部のファイル (/etc or /var 内のファイルなど) で再配置できます。ただし、パッケージに多くの異なる再配置が含まれる場合には、pkginfo ファイル内で個別の BASEDIR 値を指定してパッケージを複数のパッケージに分割することを検討してください。

個別再配置可能オブジェクト

個別再配置可能オブジェクトは、集合的再配置可能オブジェクトと同じディレクトリの場所に制限されません。個別再配置可能オブジェクトを定義するには、prototype ファイル内の *path* フィールドにインストール変数を指定する必要があります。インストール変数を指定したら、インストーラに対して再配置可能なベースディレクトリを要求する request スクリプトを作成するか、ファイルシステムデータからパス名を決定する checkinstall スクリプトを作成します。request スクリプトについては、64 ページの「request スクリプトの書き込み」を参照してください。また、checkinstall スクリプトについては、68 ページの「ファイルシステムデータを収集する方法」を参照してください。

注意 – 個別再配置可能オブジェクトは管理が困難です。個別再配置可能オブジェクトを使用すると、パッケージコンポーネントが広範囲に分散する場合があり、パッケージの複数のバージョンまたはアーキテクチャーをインストールする場合にコンポーネントを特定するのが困難になります。できる限り、集合的再配置可能オブジェクトを使用してください。

パラメータ型パス名

パラメータ型パス名は、変数指定が含まれるパス名です。たとえば、/opt/$PKGINST/*filename*は、$PKGINST という変数指定があるのでパラメータ型のパス名になります。変数指定のデフォルト値を pkginfo ファイルに定義します。この値は、request スクリプトまたは checkinstall スクリプトで変更される可能性があります。

パス内の変数指定は、パス名で開始または終了するか、斜線 (/) で囲みます。有効なパラメータ型パス名は、次のような形式をとります。

```
$PARAM/tests
tests/$PARAM/generic
/tests/$PARAM
```

変数指定を定義すると、パスは絶対パスまたは再配置可能なパスとして評価される可能性があります。次の例では、prototype ファイルに次のようなエントリが含まれます。

```
f none $DIRLOC/tests/generic
```

pkginfo ファイルには、次のエントリが含まれます。

```
DIRLOC=/myopt
```

パス名 $DIRLOC/tests/generic は、BASEDIR パラメータが pkginfo ファイルに設定されているかどうかに関係なく、絶対パス名 /myopt/tests/generic と評価されます。

この例では、prototype ファイルは前の例のものと同一であり、pkginfo ファイルには次のエントリが含まれます。

```
DIRLOC=firstcut
BASEDIR=/opt
```

パス名 $DIRLOC/tests/generic は、再配置可能なパス名 /opt/firstcut/tests/generic と評価されます。

パラメータ型のパス名については、134 ページの「パラメータ型ベースディレクトリの使用」を参照してください。

オブジェクトのソースおよびターゲットの場所の簡略な記述

prototype ファイル内の path フィールドは、オブジェクトがターゲットシステムに配置される場所を定義します。パッケージオブジェクトのディレクトリ構造がターゲットシステム上で意図された構造を模倣していない場合、prototype ファイル内でのパッケージオブジェクトの現在の場所を指定します。パッケージ内でのオブジェクトの構造化については、30 ページの「パッケージのコンテンツの編成」を参照してください。

パッケージの目的の構造と同じように開発領域が構造化されていない場合、*path* フィールドで *path1=path2* という形式を使用できます。この形式では、*path1* はターゲットシステム上でのオブジェクトの場所であり、*path2* は使用中のシステム上でのオブジェクトの場所です。

また、*path1=path2* というパス名の形式を使用する場合、*path1* を再配置可能なオブジェクト名に指定し、使用中のシステム上での対象のオブジェクトへのフルパス名に *path2* を指定することもできます。

注 – *path1* には、未定義の構築変数を含むことはできませんが、未定義のインストール変数は含むことができます。*path2* では、構築変数とインストール変数の両方を使用することができますが、未定義の変数を含むことはできません。インストール変数と構築変数の違いについては、24 ページの「パッケージ環境変数」を参照してください。

リンクは *pkgadd* コマンドで作成するので、リンクには path1= path2 形式を使用します。一般的な規則として、リンクの *path2* は絶対リンクにはせず、*path1* のディレクトリの一部を基準とした相対リンクにするようにしてください。

path1=path2 形式の代わりに、!search コマンドを使用することもできます。詳細については、43 ページの「pkgmk コマンドの検索パスの指定」を参照してください。

mode フィールド

mode フィールドには、8 進数、疑問符 (?)、または変数指定を含める可能性があります。8 進数は、ターゲットシステムにオブジェクトをインストールするときにオブジェクトのモードを指定します。? は、オブジェクトのインストール時にモードを変更しないことを示します。つまり、同じ名前のオブジェクトがターゲットシステム上にすでに存在することを意味します。

変数指定の形式 $*mode* では、変数の最初の文字を小文字で記述し、対象のフィールドはパッケージの構築時に設定されます。この変数は、prototype ファイルの作成時、または pkgmk コマンドのオプションとしての作成時に定義します。インストール変数と構築変数の違いについては、24 ページの「パッケージ環境変数」を参照してください。

ファイルタイプ i (情報ファイル)、l (ハードリンク)、および s (シンボリックリンク) が指定されたファイルでは、このフィールドを空白のままにしてください。

owner フィールド

owner フィールドには、ユーザー名、疑問符 (?)、または変数指定を含める可能性があります。ユーザー名は、最大 14 文字であり、ターゲットシステム上にすでに存在する名前にしてください (bin または root など)。? は、オブジェクトのインストール時に所有者を変更しないことを示します。つまり、同じ名前のオブジェクトがターゲットシステム上にすでに存在することを意味します。

変数指定は、$*Owner* または $*owner* の形式 (変数の最初の文字が大文字または小文字) を使用できます。変数が小文字で始まる場合、prototype ファイル内または pkgmk コマンドのオプションとして、パッケージの構築時に変数が定義されます。変数が大文字で始まる場合、変数指定は pkginfo ファイルにデフォルト値として挿入され、インストール時に request スクリプトによって再定義される可能性があります。インストール変数と構築変数の違いについては、24 ページの「パッケージ環境変数」を参照してください。

ファイルタイプ i (情報ファイル) と l (ハードリンク) が指定されたファイルでは、このフィールドを空白のままにしてください。

group フィールド

group フィールドには、グループ名、疑問符 (?)、または変数指定を含める可能性があります。グループ名は、最大 14 文字であり、ターゲットシステム上にすでに存在する名前にしてください (bin または sys など)。? は、オブジェクトのインストール時にグループを変更しないことを示します。つまり、同じ名前のオブジェクトがターゲットシステム上にすでに存在することを意味します。

変数指定は、$*Group* または $*group* の形式 (変数の最初の文字が大文字または小文字) を使用できます。変数が小文字で始まる場合、prototype ファイル内または pkgmk コマンドのオプションとして、パッケージの構築時に変数が定義されます。変数が大文字で始まる場合、変数指定は pkginfo ファイルにデフォルト値として挿入され、インストール時に request スクリプトによって再定義される可能性があります。インストール変数と構築変数の違いについては、24 ページの「パッケージ環境変数」を参照してください。

ファイルタイプ i (情報ファイル) と l (ハードリンク) が指定されたファイルでは、このフィールドを空白のままにしてください。

prototype ファイルを最初から作成する

prototype ファイルを最初から作成する場合、任意のテキストエディタでファイルを作成し、パッケージオブジェクトごとにエントリを 1 つ追加できます。prototype ファイルの形式については、32 ページの「prototype ファイルの形式」 と

prototype(4) のマニュアルページを参照してください。ただし、各パッケージオブジェクトを定義したあとには、41 ページの「prototype ファイルへの機能の追加」で説明されている機能をいくつか組み込むことが必要になる場合もあります。

例—pkgproto コマンドを使用した prototype ファイルの作成

Organizing a Package's Contentsで説明されているとおりにパッケージディレクトリ構造を編成している限り、pkgproto コマンドを使用して基本的な30 ページの「パッケージのコンテンツの編成」ファイルを作成できます。たとえば、前のセクションで説明したディレクトリ構造のサンプルと pkginfo ファイルを使用すると、prototype ファイルを作成するコマンドは次のようになります。

```
$ cd /home/jane
$ pkgproto ./SUNWcadap > InfoFiles/prototype
```

この prototype ファイルの内容は次のとおりです。

```
d none SUNWcadap 0755 jane staff
d none SUNWcadap/demo 0755 jane staff
f none SUNWcadap/demo/file1 0555 jane staff
d none SUNWcadap/srcfiles 0755 jane staff
f none SUNWcadap/srcfiles/file5 0555 jane staff
f none SUNWcadap/srcfiles/file6 0555 jane staff
d none SUNWcadap/lib 0755 jane staff
f none SUNWcadap/lib/file2 0644 jane staff
d none SUNWcadap/man 0755 jane staff
f none SUNWcadap/man/windex 0644 jane staff
d none SUNWcadap/man/man1 0755 jane staff
f none SUNWcadap/man/man1/file4.1 0444 jane staff
f none SUNWcadap/man/man1/file3.1 0444 jane staff
```

注 – パッケージ作成者の実際の所有者とグループは、pkgproto コマンドによって記録されます。pkgproto コマンドを実行する前に、chown -R コマンドと chgrp -R コマンドを使用して、希望の所有者とグループを設定することをお勧めします。

なお、この例の prototype ファイルはまだ完成されていません。このファイルを完成するには、次のセクションを参照してください。

pkgproto コマンドで作成した prototype ファイルの微調整

pkgproto コマンドは、初期の prototype ファイルを作成するのに役立ちますが、定義が必要な各パッケージオブジェクトのエントリは作成しません。このコマンドでは、エントリは完了されません。pkgproto コマンドは、次のどちらの処理も実行しません。

- ファイルタイプ v (揮発性ファイル)、e (編集可能なファイル)、x (排他的なディレクトリ)、または i (情報ファイルまたはインストールスクリプト) を指定したオブジェクトの完全なエントリの作成
- 1 回の呼び出しでの複数のクラスのサポート

ファイルタイプ v、e、x、および i を指定したオブジェクトのエントリの作成

少なくとも、prototype ファイルを変更して、ファイルタイプ i を指定したオブジェクトを追加する必要があります。情報ファイルとインストールスクリプトをパッケージディレクトリの最初のレベルに格納した場合 (例: /home/jane/SUNWcadap/pkginfo)、prototype ファイル内のエントリは次のようになります。

```
i pkginfo
```

情報ファイルとインストールスクリプトをパッケージディレクトリの最初のレベルに格納しなかった場合、それらのソースの場所を指定する必要があります。例:

```
i pkginfo=/home/jane/InfoFiles/pkginfo
```

あるいは、!search コマンドを使用して、パッケージの構築時に pkgmk コマンドの場所を検索するように指定できます。詳細については、43 ページの「pkgmk コマンドの検索パスの指定」を参照してください。

ファイルタイプ v、e、および x を指定したオブジェクトのエントリを追加するには、32 ページの「prototype ファイルの形式」で説明されている形式に従うか、prototype(4) のマニュアルページを参照してください。

注 – 必ず、e (編集可能) のファイルタイプを指定したファイルにクラスを割り当て、そのクラスの関連するクラスアクションスクリプトを含めるようにしてください。そのようにしない場合、パス名がその他のパッケージと共有されていても、パッケージの削除時にこれらのファイルも削除されます。

複数のクラス定義の使用

pkgproto コマンドを使用して基本的なprototype ファイルを作成する場合、すべてのパッケージオブジェクトを、 none クラスまたはある特定のクラスに割り当てることができます。38 ページの「例—pkgproto コマンドを使用した prototype ファイルの作成」で示されるように、基本的な pkgproto コマンドは、すべてのオブジェクトを none クラスに割り当てます。すべてのオブジェクトをある特定のクラスに割り当てるには、-c オプションを使用します。例:

```
$ pkgproto -c classname /home/jane/SUNWcadap > /home/jane/InfoFiles/prototype
```

複数のクラスを使用する場合、prototype ファイルを手動で編集し、各オブジェクトの *class* フィールドを変更することが必要になります。また、クラスを使用する場合には、pkginfo ファイルに CLASSES パラメータを定義し、クラスアクションスクリプトを書く必要もあります。クラスの使用は、オプションの機能であり、71 ページの「クラスアクションスクリプトの書き込み」で詳細に説明されています。

例—pkgproto コマンドを使用して作成した prototype ファイルの微調整

Example–Creating a prototype File With the pkgprotoCommandで説明されている pkgproto コマンドで38 ページの「例—pkgproto コマンドを使用した prototype ファイルの作成」 ファイルを作成した場合、いくつかの変更を行う必要があります。

- pkginfo ファイルにはエントリが必要です。
- パッケージソースが */home/jane*にあるので、*path* フィールドを path1=path2 形式に変更する必要があります。パッケージソースが階層ディレクトリになっており、また、!search コマンドは再帰的に検索しないので、 *path1=path2* 形式を使用したほうが容易な場合があります。
- *owner* フィールドと *group* フィールドには、ターゲットシステム上にある既存のユーザーおよびグループの名前を含めるようにしてください。すなわち、所有者 jane は、SunOS オペレーティングシステムに含まれないのでエラーになります。

  変更した prototype ファイルは次のようになります。

```
i pkginfo=/home/jane/InfoFiles/pkginfo
d none SUNWcadap=/home/jane/SUNWcadap 0755 root sys
d none SUNWcadap/demo=/home/jane/SUNWcadap/demo 0755 root bin
f none SUNWcadap/demo/file1=/home/jane/SUNWcadap/demo/file1 0555 root bin
d none SUNWcadap/srcfiles=/home/jane/SUNWcadap/srcfiles 0755 root bin
f none SUNWcadap/srcfiles/file5=/home/jane/SUNWcadap/srcfiles/file5 0555 root bin
f none SUNWcadap/srcfiles/file6=/home/jane/SUNWcadap/srcfiles/file6 0555 root bin
d none SUNWcadap/lib=/home/jane/SUNWcadap/lib 0755 root bin
f none SUNWcadap/lib/file2=/home/jane/SUNWcadap/lib/file2 0644 root bin
d none SUNWcadap/man=/home/jane/SUNWcadap/man 0755 bin bin
f none SUNWcadap/man/windex=/home/jane/SUNWcadap/man/windex 0644 root other
d none SUNWcadap/man/man1=/home/jane/SUNWcadap/man/man1 0755 bin bin
f none SUNWcadap/man/man1/file4.1=/home/jane/SUNWcadap/man/man1/file4.1 0444 bin bin
f none SUNWcadap/man/man1/file3.1=/home/jane/SUNWcadap/man/man1/file3.1 0444 bin bin
```

prototype ファイルへの機能の追加

prototype ファイルでは、各パッケージオブジェクトの定義に加えて、次の事項を行うことができます。

- インストール時に作成される追加のオブジェクトを定義します。
- インストール時にリンクを作成します。
- 複数のボリュームにわたるパッケージを配布します。
- prototype ファイルを入れ子にします。
- *mode* フィールド、*owner* フィールド、および *group* フィールドのデフォルト値を設定します。
- pkgmk コマンドの検索パスを指定します。
- 環境変数を設定します。

これらの変更を行う方法については、次のセクションを参照してください。

インストール時に作成される追加のオブジェクトの定義

prototype ファイルを使用して、実際にはインストールメディアに提供されないオブジェクトを定義できます。これらのオブジェクトは、インストール時に pkgadd コマンドを使用すると、インストール時にまだ存在しない場合、必要なファイルタイプで作成されます。

オブジェクトがターゲットシステムに作成されるように指定するには、適切なファイルタイプを指定した prototype ファイルにそのオブジェクトのエントリを追加します。

たとえば、ターゲットシステムにディレクトリを作成し、インストールメディアにはそのディレクトリを提供しない場合、prototype ファイル内でそのディレクトリに次のエントリを追加します。

```
d none /directory 0644 root other
```

ターゲットシステム上で空のファイルを作成する場合、prototype ファイル内での空のファイルに対するエントリは次のようになります。

```
f none filename=/dev/null 0644 bin bin
```

インストールメディアに提供されるオブジェクトは、通常ファイルおよび編集スクリプト (ファイルタイプ e、 v、f) と、そのファイルおよびスクリプトを格納する必要があるディレクトリだけです。追加のオブジェクトは、提供されるオブジェクト、ディレクトリ、名前付きパイプ、デバイス、ハードリンク、およびシンボリックリンクを参照せずに作成されます。

インストール時のリンクの作成

パッケージのインストール時にリンクを作成するには、リンクオブジェクトのprototype ファイルエントリ内で次の内容を定義します。

- ファイルタイプを l (リンク) または s (シンボリックリンク) にします。
- リンクオブジェクトのパス名を *path1=path2* 形式で指定します。この場合、*path1* はターゲット、*path2* はソースファイルです。一般的な規則として、リンクの *path2* は絶対リンクにはせず、*path1* のディレクトリの一部を基準とした相対リンクにするようにしてください。たとえば、シンボリックリンクを定義する prototype ファイルのエントリは、次のようになります。

  ```
  s none etc/mount=../usr/etc/mount
  ```

パッケージが絶対パスまたは再配置可能としてインストールされるかどうかにかかわらず、相対リンクはこの方法で指定できます。

複数のボリュームにわたるパッケージの配布

pkgmk コマンドを使用してパッケージを構築すると、このコマンドにより、複数ボリュームのパッケージを編成するのに必要な計算と処理が実行されます。複数ボリュームのパッケージは、分割パッケージと呼ばれます。

ただし、prototype ファイル内でオプションの *part* フィールドを使用して、配置するオブジェクトの部分を定義することもできます。このフィールドの数値が pkgmk コマンドに優先され、このフィールドに指定された部分にコンポーネントが強制的に配置されます。ファイルシステムとしてフォーマットされたリムーバブルメディアでは、各部分と各ボリューム間に一対一の対応が存在します。ボリュームが開発者によって事前に割り当てられている場合、容量が不足しているボリュームが存在すると pkgmk コマンドでエラーが発生します。

prototype ファイルの入れ子化

複数の prototype ファイルを作成し、!include コマンドを使用してそれらのファイルを prototype ファイルに組み込むことができます。保守を容易にするために、ファイルを入れ子にすることもできます。

次の例では、prototype ファイルが 3 つあります。メインファイル (prototype) は編集します。その他の 2 つのファイル (proto2と proto3) は組み込みます。

```
!include /source-dir/proto2
!include /source-dir/proto3
```

mode フィールド、*owner* フィールド、および *group* フィールドのデフォルト値の設定

mode フィールド、 *owner* フィールド、および *group* フィールドにデフォルト値を設定するには、 `!default` コマンドを `prototype` ファイルに挿入します。例:

```
!default 0644 root other
```

注 – `!default` コマンドの範囲は、コマンドの挿入位置から始まり、ファイルの末尾までに及びます。コマンドの範囲は、組み込まれたファイルには適用されません。

ただし、ターゲットシステム (`/usr` または `/etc/vfstab` など) に存在することがわかっているディレクトリ (ファイルタイプ `d`) および編集可能なファイル (ファイルタイプ `e`) の場合、*prototype* ファイル内で *mode* フィールド、*owner* フィールド、および `group` フィールドを疑問符 (`?`) に設定する必要があります。これにより、サイト管理者が変更した可能性がある既存の設定が破棄されません。

`pkgmk` コマンドの検索パスの指定

パッケージオブジェクトのソースの場所がターゲットの場所と異なり、36 ページの「オブジェクトのソースおよびターゲットの場所の簡略な記述」で説明されている *path1=path2* 形式を使用しない場合、`prototype` ファイル内で `!search` コマンドを使用できます。

たとえば、ホームディレクトリに `pkgfiles` というディレクトリを作成し、そのディレクトリにすべての情報ファイルとインストールスクリプトが含まれる場合、`pkgmk` コマンドでパッケージを構築するときにこのディレクトリが検索されるように指定できます。

prototype ファイル内のこのコマンドは次のようになります。

```
!search /home-dir/pkgfiles
```

注 – 検索要求は、組み込まれたファイルには適用されません。また、検索は、リストされた特定のディレクトリに制限され、再帰的には検索しません。

環境変数の設定

`!PARAM=value` 形式の *prototype* ファイルには、コマンドを追加することもできます。この形式のコマンドは、現在の環境の変数を定義します。複数の `prototype` ファイルを使用する場合、このコマンドのスコープは、コマンドが定義されている `prototype` ファイルに対してローカルです。

変数 *PARAM* は、小文字または大文字で始めることができます。*PARAM* 変数の値が構築時に不明な場合、pkgmk コマンドはエラーで中断されます。構築変数とインストール変数の違いについては、24 ページの「パッケージ環境変数」を参照してください。

▼ pkgproto コマンドを使用して prototype ファイルを作成する方法

注 – 情報ファイルとインストールスクリプトは、prototype ファイルを作成する前に作成したほうが簡単です。ただし、この順序は必須ではありません。パッケージコンテンツを変更したあと、prototype ファイルをいつでも編集できます。情報ファイルとインストールスクリプトについては、第 3 章「パッケージの機能の拡張 (タスク)」を参照してください。

1. まだ決定していない場合、絶対的なパッケージオブジェクトと再配置可能なパッケージオブジェクトを決定します。

   この手順を完了させるための情報については、33 ページの「*path* フィールド」を参照してください。

2. ターゲットシステム上での配置を模倣するようにパッケージオブジェクトを編成します。

   30 ページの「パッケージのコンテンツの編成」で説明されているようにパッケージを編成済みの場合、手順 1 での決定に基づき、いくつかの変更を加える必要が生じることがあります。パッケージをまだ編成していない場合、ここで編成するようにしてください。パッケージを編成しない場合、pkgproto コマンドを使用して基本的な prototype ファイルを作成できません。

3. 集合的再配置可能オブジェクトがパッケージに含まれる場合、**pkginfo** ファイルを編集して、**BASEDIR** パラメータを適切な値に設定します。

   例:

   ```
   BASEDIR=/opt
   ```

   集合的再配置可能オブジェクトについては、34 ページの「集合的再配置可能オブジェクト」を参照してください。

4. 個別再配置可能オブジェクトがパッケージに含まれる場合、インストーラに対して適切なパス名を要求する **request** スクリプトを作成します。あるいは、ファイルシステムデータから適切なパスを決定する **checkinstall** スクリプトを作成します。

   次の一覧に、共通のタスクを参照するためのページ番号を示します。

   - request スクリプトを作成するには、65 ページの「request スクリプトを書く方法」を参照してください。

- checkinstall スクリプトを作成するには、68 ページの「ファイルシステムデータを収集する方法」を参照してください。
- 個別再配置可能オブジェクトについては、34 ページの「個別再配置可能オブジェクト」を参照してください。

5 すべてのパッケージコンポーネントの所有者とグループを、ターゲットシステム上で希望の所有者とグループに変更します。

パッケージと情報ファイルのディレクトリ上で chown -R と chgrp -R コマンドを使用します。

6 **pkgproto** コマンドを実行して、基本的な **prototype** ファイルを作成します。

pkgproto コマンドは、ディレクトリをスキャンして基本的なファイルを作成します。例:

```
$ cd package-directory
$ pkgproto ./package-directory > prototype
```

prototype ファイルは、システム上の任意の場所に配置できます。情報ファイルとインストールスクリプトを 1 つの場所に保管すると、アクセスと保守が容易になります。pkgproto コマンドについては、pkgproto(1) のマニュアルページを参照してください。

7 任意のテキストエディタを使用して **prototype** ファイルを編集し、**v**、 **e**、**x**、および **i** のタイプのファイルにエントリを追加します。

必要となる特定の変更内容については、39 ページの「pkgproto コマンドで作成した prototype ファイルの微調整」を参照してください。

8 (オプション) 複数のクラスを使用する場合、**prototype** ファイルと **pkginfo** ファイルを編集します。任意のテキストエディタを使用して、必要な変更を加え、対応するクラスアクションスクリプトを作成します。

必要となる特定の変更内容については、39 ページの「pkgproto コマンドで作成した prototype ファイルの微調整」および71 ページの「クラスアクションスクリプトの書き込み」を参照してください。

9 任意のテキストエディタで **prototype** ファイルを編集して、パス名を再定義し、その他のフィールド設定を変更します。

詳細については、39 ページの「pkgproto コマンドで作成した prototype ファイルの微調整」を参照してください。

10 (オプション) 任意のテキストエディタで **prototype** ファイルを編集して、**prototype** ファイルに機能を追加します。

詳細については、41 ページの「prototype ファイルへの機能の追加」を参照してください。

11 ファイルを保存してエディタを終了します。

参照　次のタスクの準備が整っている場合、47 ページの「パッケージの構築方法」を参照してください。

パッケージの構築

pkgmk コマンドを使用してパッケージを構築します。 pkgmk コマンドは、次のタスクを実行します。

- prototype ファイル内で定義されたすべてのオブジェクトをディレクトリ形式に入力します。
- prototype ファイルと置き換える pkgmap ファイルを作成します。
- pkgadd コマンドへの入力値として使用される、インストール可能なパッケージを生成します。

最も簡単な pkgmk コマンドの使用

pkgmk コマンドの最も簡単な形式は、オプションを指定しない場合です。オプションを指定せずに pkgmk コマンドを使用する前に、現在の作業用ディレクトリにパッケージの prototype ファイルを含める必要があります。コマンド、ファイル、およびディレクトリの出力は、/var/spool/pkg ディレクトリに書き込まれます。

pkgmap ファイル

pkgmk コマンドでパッケージを構築する場合、prototype ファイルに置き換える pkgmap ファイルがコマンドによって作成されます。前の例で使用した pkgmap ファイルには、次のコンテンツが含まれています。

```
$ more pkgmap
: 1 3170
1 d none SUNWcadap 0755 root sys
1 d none SUNWcadap/demo 0755 root bin
1 f none SUNWcadap/demo/file1 0555 root bin 14868 45617 837527496
1 d none SUNWcadap/lib 0755 root bin
1 f none SUNWcadap/lib/file2 0644 root bin 1551792 62372 837527499
1 d none SUNWcadap/man 0755 bin bin
1 d none SUNWcadap/man/man1 0755 bin bin
1 f none SUNWcadap/man/man1/file3.1 0444 bin bin 3700 42989 837527500
1 f none SUNWcadap/man/man1/file4.1 0444 bin bin 1338 44010 837527499
1 f none SUNWcadap/man/windex 0644 root other 157 13275 837527499
1 d none SUNWcadap/srcfiles 0755 root bin
1 f none SUNWcadap/srcfiles/file5 0555 root bin 12208 20280 837527497
1 f none SUNWcadap/srcfiles/file6 0555 root bin 12256 63236 837527497
1 i pkginfo 140 10941 837531104
$
```

このファイルの形式は、prototype ファイルの形式とよく似ています。ただし、pkgmap ファイルには、次の情報が含まれます。

- 最初の行には、パッケージがまたがるボリューム数と、インストール時のパッケージの近似サイズが示されます。

  たとえば、: 1 3170 は、パッケージが 1 つのボリュームにわたり、インストール時に約 3170 ブロック (512 バイトブロック単位) が使用されることを意味します。
- 各パッケージオブジェクトのサイズ、チェックサム、および変更時間を定義する、3 つの追加のフィールドがあります。
- パッケージのインストールにかかる時間を削減するために、クラスおよびパス名のアルファベット順でパッケージオブジェクトが一覧表示されます。

▼ パッケージの構築方法

1. まだ作成していない場合、**pkginfo** ファイルを作成します。

   作成手順については、29 ページの「pkginfo ファイルの作成方法」を参照してください。

2. まだ作成していない場合、**prototype** ファイルを作成します。

   作成手順については、44 ページの「pkgproto コマンドを使用して prototype ファイルを作成する方法」を参照してください。

3. 現在の作業用ディレクトリを、パッケージの **prototype** ファイルを含むディレクトリと同じものにします。

4. パッケージを構築します。

   ```
   $ pkgmk [-o] [-a arch] [-b base-src-dir] [-d device]
     [-f filename] [-l limit] [-p pstamp] [-r rootpath]
     [-v version] [PARAM=value] [pkginst]
   ```

-o	既存のバージョンのパッケージを上書きします。
-a *arch*	pkginfo ファイル内のアーキテクチャー情報をオーバーライドします。
-b *base-src-dir*	pkgmk コマンドが開発システム上のオブジェクトを検索するときに *base-src-dir* が再配置可能なパス名の先頭に追加されるように要求します。
-d *device*	*device* (絶対ディレクトリパス名、フロッピーディスク、またはリムーバブルディスクの場合がある) にパッケージがコピーされるように指定します。
-f *filename*	prototype ファイルとして使用されるファイル *filename* を指定します。デフォルト名は、prototype または Prototype です。
-l *limit*	512 バイトブロック単位で、出力デバイスの最大サイズを指定します。

`-p` *pstamp*	`pkginfo` ファイル内の製品スタンプ定義をオーバーライドします。
`-r` *rootpath*	開発システム上でオブジェクトを検出するためにルートディレクトリ *rootpath* が使用されるように要求します。
`-v` *version*	`pkginfo` ファイル内のバージョン情報をオーバーライドします。
PARAM=value	グローバルの環境変数を設定します。小文字で始まる変数は、構築時に解釈処理されます。大文字で始まる変数は `pkginfo` ファイルに配置され、インストール時に使用されます。
pkginst	パッケージの省略名または特定のインスタンス (例: `SUNWcadap.4`) を指定します。

詳細については、`pkgmk(1)` のマニュアルページを参照してください。

5 パッケージのコンテンツを確認します。

```
$ pkgchk -d device-name pkg-abbrev
Checking uninstalled directory format package pkg-abbrev
from device-name
## Checking control scripts.
## Checking package objects.
## Checking is complete.
$
```

`-d` *device-name*	パッケージの位置を指定します。*device-name* には、完全なディレクトリパス、またはテープやリムーバルディスクの識別子を指定できます。
pkg-abbrev	チェックされる、1 つ以上の (空白で区切られた) パッケージの名前です。やこの引数を省略すると、`pkgchk` コマンドにより、使用可能なすべてのパッケージがチェックされます。

`pkgchk` コマンドは、必要に応じて、パッケージのチェックされる内容を印刷し、警告やエラーを表示します。`pkgchk` コマンドについては、92 ページの「パッケージの整合性の確認」を参照してください。

注意 – エラーについては、真剣に検討するようにしてください。エラーがある場合、スクリプトの修正が必要な可能性があります。`pkgchk` コマンドの出力が好ましくない場合、すべてのエラーをチェックしてから先に進んでください。

例 2–2 パッケージの構築

次の例では、`Fine-Tuning a prototype File Created With the pkgprotoCommand`で作成した 39 ページの「`pkgproto` コマンドで作成した `prototype` ファイルの微調整」ファイルを使用します。

```
$ cd /home/jane/InfoFiles
$ pkgmk
## Building pkgmap from package prototype file.
## Processing pkginfo file.
WARNING: parameter  set to "system990708093144"
WARNING: parameter  set to "none"
## Attempting to volumize 13 entries in pkgmap.
part  1 -- 3170 blocks, 17 entries
## Packaging one part.
/var/spool/pkg/SUNWcadap/pkgmap
/var/spool/pkg/SUNWcadap/pkginfo
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/demo/file1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/lib/file2
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/man1/file3.1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/man1/file4.1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/windex
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/srcfiles/file5
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/srcfiles/file6
## Validating control scripts.
## Packaging complete.
$
```

例 2–3 再配置可能ファイルのソースディレクトリの指定

パッケージに再配置可能ファイルが含まれる場合、`pkgmk` コマンドの `-b` *base-src-dir* オプションを使用して、パッケージの作成時に再配置可能なパス名の先頭に追加するパス名を指定します。このオプションは、再配置可能ファイルに対して *path1=path2* 形式を使用していない場合や、`prototype` ファイル内で `!search` コマンドを使用して検索パスを指定していない場合に便利です。

次のコマンドでは、次に示す特性によってパッケージを構築します。

- パッケージは、`pkgproto` コマンドで作成される `prototype` ファイルのサンプルによって構築されます。詳細については、38 ページの「例—pkgproto コマンドを使用した prototype ファイルの作成」を参照してください。
- パッケージは、*path* フィールドを変更せずに構築されます。
- パッケージは、`pkginfo` ファイルにエントリを追加します。

```
$ cd /home/jane/InfoFiles
$ pkgmk -o -b /home/jane
## Building pkgmap from package prototype file.
## Processing pkginfo file.
WARNING: parameter  set to "system960716102636"
WARNING: parameter  set to "none"
## Attempting to volumize 13 entries in pkgmap.
part  1 -- 3170 blocks, 17 entries
## Packaging one part.
/var/spool/pkg/SUNWcadap/pkgmap
/var/spool/pkg/SUNWcadap/pkginfo
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/demo/file1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/lib/file2
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/man1/file3.1
```

```
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/man1/file4.1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/windex
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/srcfiles/file5
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/srcfiles/file6
## Validating control scripts.
## Packaging complete.
```

この例では、-o オプションを指定して、デフォルトのディレクトリ /var/spool/pkg にパッケージが構築されます。このオプションにより、例 2–2 で作成されたパッケージが上書きされます。

例 2–4 情報ファイルとパッケージオブジェクトへの異なるソースディレクトリの指定

パッケージ情報ファイル (pkginfo および prototype) とパッケージオブジェクトを 2 つの異なるディレクトリに配置する場合、pkgmk コマンドの -b *base-src-dir* オプションと - r *rootpath* オプションを使用してパッケージを作成できます。パッケージオブジェクトが /product/pkgbin というディレクトリに含まれ、その他のパッケージ情報ファイルが /product/pkgsrc というディレクトリに含まれる場合、次のコマンドを使用してパッケージを /var/spool/pkg ディレクトリに配置できます。

```
$ pkgmk -b /product/pkgbin -r /product/pkgsrc -f /product/pkgsrc/prototype
```

また、次のコマンドを使用した場合でも、同じ結果が得られます。

```
$ cd /product/pkgsrc
$ pkgmk -o -b /product/pkgbin
```

この例では、pkgmk コマンドは現在の作業用ディレクトリを使用してパッケージの残りの部分 (prototype 情報ファイルや pkginfo 情報ファイルなど) を検索します。

参照 オプションの情報ファイルおよびインストールスクリプトをパッケージに追加するには、第 3 章「パッケージの機能の拡張 (タスク)」を参照してください。それ以外の場合は、パッケージを構築したあとに、整合性を確認するようにしてください。第 4 章「パッケージの確認と転送」では、パッケージの確認方法について説明し、確認済みのパッケージを配布媒体に転送する方法について順を追って説明します。

◆ ◆ ◆ 第 3 章

3 パッケージの機能の拡張 (タスク)

この章では、パッケージのオプション情報ファイルとインストールスクリプトの作成方法を説明します。第 2 章「パッケージの構築」ではパッケージ作成の最低限の要件を説明しましたが、この章ではパッケージに組み込むことのできる追加機能について説明します。この追加機能は、パッケージの設計方法を計画する際に考慮した条件に基づいています。詳細については、17 ページの「パッケージを構築する前の考慮事項」を参照してください。

この章の内容は以下のとおりです。

情報ファイルとインストールスクリプトの作成 (タスクマップ)

次のタスクマップでは、パッケージに組み込むことのできるオプション機能について説明します。

表 3–1 情報ファイルとインストールスクリプトの作成(タスクマップ)

タスク	説明	参照先
1. 情報ファイルを作成する	パッケージの依存関係を定義する パッケージの依存関係を定義すると、パッケージが以前のバージョンと互換性があるかどうか、ほかのパッケージに依存するかどうか、またはほかのパッケージがそのパッケージに依存するかどうかを指定できます。	54 ページの「パッケージの依存関係を定義する方法」
	著作権に関するメッセージを書く `copyright` ファイルを用意すると、ソフトウェアアプリケーションを法的に保護できます。	56 ページの「著作権に関するメッセージを書く方法」
	ターゲットシステムに追加領域を予約する `space` ファイルにより、ターゲットシステム上でブロックが取り置かれます。これにより、`pkgmap` ファイルで定義されていないファイルを、インストール時に作成できます。	58 ページの「ターゲットシステムに追加領域を予約する方法」
2. インストールスクリプトを作成する	インストーラから情報を取得する `request` スクリプトを使用すると、パッケージをインストールする人から情報を取得できます。	65 ページの「`request` スクリプトを書く方法」
	インストールに必要なファイルシステムデータを収集する `checkinstall` スクリプトを使用すると、ターゲットシステムの分析を行なって、インストール用の環境を適切に設定したり、インストールをクリーンに停止したりできます。	68 ページの「ファイルシステムデータを収集する方法」
	手続きスクリプトを書く 手続きスクリプトを使用すると、インストールプロセスまたは削除プロセスの特定のフェーズの間に、カスタマイズしたインストール命令を挿入できます。	70 ページの「手続きスクリプトを書く方法」
	クラスアクションスクリプトを書く クラスアクションスクリプトを使用すると、パッケージのインストールおよび削除の間にパッケージオブジェクトの特定のグループに対して、一連の命令を指定して実行できます。	79 ページの「クラスアクションスクリプトを書く方法」

情報ファイルの作成

このセクションでは、オプションのパッケージ情報ファイルについて説明します。パッケージ情報ファイルを使用すると、パッケージの依存関係の定義、著作権に関するメッセージの提供、およびターゲットシステム上の追加領域の予約が可能になります。

パッケージの依存関係の定義

作成するパッケージがほかのパッケージに依存するかどうか、およびほかのパッケージが作成するパッケージに依存するかどうかを、判定する必要があります。パッケージの依存関係と非互換性は、2 つのオプションパッケージ情報ファイル、`compver` と `depend` で定義できます。

`compver` ファイルを提供することで、インストールされているパッケージと互換性のある、以前のバージョンのパッケージを指定できます。

`depend` ファイルを提供することで、パッケージに関連付けられた 3 種類の依存関係を定義できます。これらの依存タイプは次のとおりです。

- 必須パッケージ – 作成するパッケージは、別のパッケージの存在に依存します。
- 逆依存 – 別のパッケージが、作成するパッケージの存在に依存します。

注 – 逆依存タイプは、`depend` ファイルを提供できないパッケージが、作成するパッケージに依存する場合にのみ使用します。

- 非互換パッケージ – 作成するパッケージは、指定されているパッケージと互換性がありません。

`depend` ファイルは、とても基本的な依存関係のみを判定します。作成するパッケージが特定のファイル、そのファイルの内容、動作に依存する場合、`depend` ファイルによる判定だけでは正確性は十分ではありません。このような場合は、`request` スクリプトまたは `checkinstall` スクリプトを使用して、詳細な依存関係チェックを行うようにしてください。`checkinstall` スクリプトは、パッケージのインストールプロセスをクリーンに停止できる唯一のスクリプトでもあります。

注 – `depend` ファイルと `compver` ファイルのエントリが `prototype` ファイルにあることを確認してください。ファイルタイプは、`i` (パッケージ情報ファイル) としてください。

詳細は、`depend(4)` および `compver(4)` のマニュアルページを参照してください。

▼ パッケージの依存関係を定義する方法

1. 情報ファイルが格納されているディレクトリを、現在の作業用ディレクトリにします。

2. 以前のバージョンのパッケージが存在し、新しいパッケージと互換性があることを指定する必要がある場合は、任意のテキストエディタを使用して **compver** という名前のファイルを作成します。

   作成するパッケージと互換性があるバージョンを一覧表示します。次の形式を使用します。

   string string . . .

   string の値は、各互換パッケージの `pkginfo` ファイルの `VERSION` パラメータに割り当てられている値と同じです。

3. ファイルを保存してエディタを終了します。

4. 作成するパッケージがほかのパッケージの存在に依存する場合、ほかのパッケージが作成するパッケージの存在に依存する場合、または作成するパッケージが別のパッケージと互換性がない場合は、任意のテキストエディタを使用して **depend** という名前のファイルを作成します。

   依存関係ごとにエントリを追加します。次の形式を使用します。

   type pkg-abbrev pkg-name
   (*arch*) *version*
   (*arch*) *version . . .*

   type 依存タイプを定義します。次のいずれかの文字を指定する必要があります。`P` (必須パッケージ)、`I` (非互換パッケージ)、または `R` (逆依存)。

   pkg-abbrev パッケージの省略名を指定します (`SUNWcadap` など)。

   pkg-name 完全なパッケージ名を指定します。たとえば、「`Chip designers need CAD application software to design abc chips. Runs only on xyz hardware and is installed in the usr partition.`」などです。

   (*arch*) オプション。パッケージが稼働するハードウェアの種類を指定します。たとえば、`sparc` や `x86` などです。アーキテクチャーを指定する場合は、区切り文字として丸括弧を使用する必要があります。

   version オプション。`pkginfo` ファイルで `VERSION` パラメータに割り当てられている値を指定します。

   詳細については、depend(4) を参照してください。

5. ファイルを保存してエディタを終了します。

6. 次のいずれかのタスクを完了します。

- 追加情報ファイルおよびインストールスクリプトを作成する場合は、次のタスク、56 ページの「著作権に関するメッセージを書く方法」に進んでください。
- prototype ファイルを作成していない場合は、44 ページの「pkgproto コマンドを使用して prototype ファイルを作成する方法」の手順を完了して、手順 7 に進んでください。
- prototype ファイルをすでに作成している場合は、それを編集し、前の手順で作成した各ファイルのエントリを追加します。

7 パッケージを構築します。

必要な場合は、47 ページの「パッケージの構築方法」を参照してください。

例 3–1 compver ファイル

この例では、4 つのバージョンのパッケージ (1.0、1.1、2.0 と新しいパッケージ 3.0) があります。新しいパッケージには、以前の 3 つのすべてのバージョンとの互換性があります。最新バージョンの compver ファイルは次のような内容です。

```
release 3.0
release 2.0
version 1.1
1.0
```

エントリは、順番になっていなくてもかまいません。ただし、各パッケージの pkginfo ファイルでの VERSION パラメータの定義と正確に一致するようにしてください。この例では、パッケージ設計者は最初の 3 つのバージョンで異なる形式を使用しています。

例 3–2 depend ファイル

この例では、サンプルパッケージ SUNWcadap は、SUNWcsr パッケージと SUNWcsu パッケージがターゲットシステムにすでにインストールされている必要があるものとします。 SUNWcadap の depend ファイルは、次のような内容です。

```
P SUNWcsr Core Solaris, (Root)
P SUNWcsu Core Solaris, (Usr)
```

参照 パッケージを構築したあと、実際にインストールして、正しくインストールされることを確認し、整合性を検証します。第 4 章「パッケージの確認と転送」では、これらのタスクについて説明し、検証済みのパッケージを配布媒体に転送する方法の手順を示します。

著作権に関するメッセージの書き込み

パッケージのインストール中に著作権に関するメッセージを表示するべきかどうかを決める必要があります。表示する必要がある場合は、`copyright` ファイルを作成します。

注 – ソフトウェアアプリケーションを法的に保護するには、`copyright` ファイルを含めるようにしてください。メッセージの正確な文言については、会社の法務部門に確認してください。

著作権に関するメッセージを提供するには、`copyright` という名前のファイルを作成する必要があります。インストール中に、ファイルに記述されているとおりに (書式設定されずに) メッセージが表示されます。詳細については、`copyright(4)` のマニュアルページを参照してください。

注 – `copyright` ファイルのエントリが `prototype` ファイルにあることを確認してください。ファイルタイプは、`i` (パッケージ情報ファイル) としてください。

▼ 著作権に関するメッセージを書く方法

1. 情報ファイルが格納されているディレクトリを、現在の作業用ディレクトリにします。
2. 任意のテキストエディタを使用して、**`copyright`** という名前のファイルを作成します。

   パッケージのインストール時に表示する、著作権に関するメッセージの文章を、正確に入力します。
3. ファイルを保存してエディタを終了します。
4. 次のいずれかのタスクを完了します。
   - 追加情報ファイルおよびインストールスクリプトを作成する場合は、次のタスク、58 ページの「ターゲットシステムに追加領域を予約する方法」に進んでください。
   - *prototype* ファイルを作成していない場合は、44 ページの「`pkgproto` コマンドを使用して `prototype` ファイルを作成する方法」の手順を完了して、手順 5 に進んでください。

- prototype ファイルをすでに作成している場合は、それを編集し、前の手順で作成した情報ファイルのエントリを追加します。

5 パッケージを構築します。

必要な場合は、47 ページの「パッケージの構築方法」を参照してください。

例 3–3 copyright ファイル

たとえば、著作権に関するメッセージの一部分は次のようになります。

```
Copyright (c) 2003 Company Name
All Rights Reserved

This product is protected by copyright and distributed under
licenses restricting copying, distribution, and decompilation.
```

参照 パッケージを構築したあと、実際にインストールして、正しくインストールされることを確認し、整合性を検証します。第 4 章「パッケージの確認と転送」では、これらのタスクについて説明し、検証済みのパッケージを配布媒体に転送する方法の手順を示します。

ターゲットシステムでの追加領域の予約

ターゲットシステムでは、パッケージは追加のディスク領域が必要かどうかを判定する必要があります。この領域は、パッケージオブジェクトで必要な領域とは別に追加するものです。必要な場合は、space 情報ファイルを作成します。このタスクは、41 ページの「インストール時に作成される追加のオブジェクトの定義」で説明した、インストール時の空のファイルとディレクトリの作成とは異なるものです。

pkgadd コマンドを使用すると、pkgmap ファイルでのオブジェクト定義に基づいて、パッケージのインストールに十分なディスク領域が確保されます。ただし、パッケージでは、pkgmap ファイルで定義されているオブジェクトが必要とするディスク領域以上の追加の領域が必要になる場合があります。たとえば、パッケージのインストール後に、ディスク領域を消費するデータベース、ログファイルなど、ファイルサイズが大きくなる可能性があるファイルが作成されるような場合です。このような場合のために領域を予約するには、必要なディスク領域を指定する space ファイルを含めるようにしてください。pkgadd コマンドは、space ファイルで指定されている追加領域をチェックします。詳細については、space(4) のマニュアルページを参照してください。

注 – space ファイルのエントリが prototype ファイルにあることを確認してください。ファイルタイプは、i (パッケージ情報ファイル) としてください。

▼ ターゲットシステムに追加領域を予約する方法

1 情報ファイルが格納されているディレクトリを、現在の作業用ディレクトリにします。

2 任意のテキストエディタを使用して、**spacet** という名前のファイルを作成します。

パッケージに必要な追加ディスク領域の要件を指定します。次の形式を使用します。

pathname blocks inodes

pathname ディレクトリ名を指定します。ファイルシステムのマウントポイントであってもなくてもかまいません。

blocks 予約する 512 バイトブロックの数を指定します。

i ノード 必要な i ノードの数を指定します。

詳細については、space(4) のマニュアルページを参照してください。

3 ファイルを保存してエディタを終了します。

4 次のいずれかのタスクを完了します。

- インストールスクリプトを作成する場合は、次のタスク、65 ページの「request スクリプトを書く方法」に進んでください。
- prototype ファイルを作成していない場合は、44 ページの「pkgproto コマンドを使用して prototype ファイルを作成する方法」の手順を完了して、手順 5 に進んでください。
- prototype ファイルをすでに作成している場合は、それを編集し、前の手順で作成した情報ファイルのエントリを追加します。

5 パッケージを構築します。

必要な場合は、47 ページの「パッケージの構築方法」を参照してください。

例 3–4 space ファイル

この space ファイルの例では、1000 個の 512 バイトブロックと 1 個の i ノードをターゲットシステムの /opt ディレクトリに予約するように指定しています。

```
/opt   1000   1
```

参照　パッケージを構築したあと、実際にインストールして、正しくインストールされることを確認し、整合性を検証します。第4章「パッケージの確認と転送」では、これらのタスクについて説明し、検証済みのパッケージを配布媒体に転送する方法の手順を示します。

インストールスクリプトの作成

このセクションでは、オプションのパッケージインストールスクリプトについて説明します。`pkgadd` コマンドでは、パッケージ情報ファイルが入力として使用され、パッケージのインストールに必要なすべてのアクションが自動的に実行されます。パッケージインストールスクリプトを提供する必要はありません。ただし、カスタマイズしたインストール手順を作成する場合は、インストールスクリプトを使用して実現できます。インストールスクリプトには次の条件があります。

- Bourne シェル (`sh`) で実行可能である必要があります。
- Bourne シェルのコマンドとテキストで構成します。
- `#!/bin/sh` シェル識別子を含める必要はありません。
- 実行可能ファイルである必要はありません。

カスタマイズしたアクションを実行できるインストールスクリプトには 4 種類あります。

- `request` スクリプト

  `request` スクリプトは、パッケージをインストールしている管理者に、環境変数の割り当てまたは再定義のためのデータを要求します。

- `checkinstall` スクリプト

  `checkinstall` スクリプトは、ターゲットシステムで必要なデータがあるかを検査し、パッケージの環境変数を設定または変更して、インストールを続行するかどうかを判定します。

  注 – `checkinstall` スクリプトは、Solaris 2.5 およびその互換リリース以降で使用できます。

- 手続きスクリプト

  手続きスクリプトは、パッケージのインストールまたは削除の前後に呼び出される手続きを特定します。手続きスクリプトには、`preinstall`、`postinstall`、`preremove` および `postremove` の 4 種類があります。

- クラスアクションスクリプト

クラスアクションスクリプトでは、インストールまたは削除の間にファイルのクラスに適用するべき 1 つまたは複数のアクションを定義します。独自のクラスを定義できます。または、4 つの標準クラス (sed、awk、build、および preserve) のいずれかを使用することもできます。

パッケージインストール時のスクリプトの処理

使用するスクリプトの種類は、インストールプロセスのどの時点でスクリプトのアクションが必要かによって決まります。パッケージのインストールでは、pkgadd コマンドは次の手順を実行します。

1. request スクリプトを実行します。

   パッケージをインストールしている管理者に入力を要求できるのは、この時点のみです。

2. checkinstall スクリプトを実行します。

   checkinstall スクリプトは、ファイルシステムのデータを収集し、環境変数の定義を作成または変更して、以降のインストールを制御できます。パッケージ環境変数の詳細については、24 ページの「パッケージ環境変数」を参照してください。

3. preinstall スクリプトを実行します。
4. インストールするクラスごとに、パッケージオブジェクトをインストールします。

   これらのファイルのインストールはクラスごとに行われ、それに従ってクラスアクションスクリプトが実行されます。対象となるクラスのリストおよびインストールの順序は、最初は pkginfo ファイルの CLASSES パラメータで定義されています。ただし、request スクリプトまたは checkinstall スクリプトで、CLASSES パラメータの値を変更できます。インストール時のクラスの処理方法の詳細については、72 ページの「パッケージインストール時のクラスの処理方法」を参照してください。

   a. シンボリックリンク、デバイス、名前付きパイプ、および必要なディレクトリを作成します。
   b. クラスに基づいて、通常ファイル (ファイルタイプ e、 v、f) をインストールします。

      クラスアクションスクリプトは、インストールする通常ファイルのみに渡されます。ほかのパッケージオブジェクトはすべて、pkgmap ファイルの情報から自動的に作成されます。

   c. すべてのハードリンクを作成します。
5. postinstall スクリプトを実行します。

パッケージ削除時のスクリプトの処理

パッケージを削除するときは、pkgrm コマンドで次の手順が実行されます。

1. preremove スクリプトを実行します。
2. 各クラスのパッケージオブジェクトを削除します。

   削除もクラスごとに行われます。削除スクリプトは、CLASSES パラメータで定義されている順序に基づいて、インストールの逆の順序で処理されます。インストール時のクラスの処理方法の詳細については、72 ページの「パッケージインストール時のクラスの処理方法」を参照してください。

   a. ハードリンクを削除します。
   b. 通常ファイルを削除します。
   c. シンボリックリンク、デバイス、および名前付きパイプを削除します。
3. postremove スクリプトを実行します。

request スクリプトは、パッケージの削除時には処理されません。ただし、このスクリプトの出力はインストールされたパッケージに保持されており、削除スクリプトで利用できます。request スクリプトの出力は、環境変数のリストです。

スクリプトで使用できるパッケージ環境変数

次の環境変数のグループは、すべてのインストールスクリプトで使用できます。一部の環境変数は、request スクリプトまたは checkinstall スクリプトで変更できます。

- request スクリプトまたは checkinstall スクリプトでは、pkginfo ファイルに含まれる標準パラメータのうち、必須パラメータ以外のいずれかのパラメータを設定または変更できます。標準インストールパラメータの詳細については、pkginfo(4) のマニュアルページを参照してください。

  注 – BASEDIR パラメータは、Solaris 2.5 リリースおよびその互換リリース以降でのみ変更できます。

- pkginfo ファイルで値を割り当てることにより、独自のインストール環境変数を定義できます。このような環境変数の名前は、先頭が大文字の英数字でなければなりません。これらの環境変数はすべて、request スクリプトまたは checkinstall スクリプトで変更できます。
- request スクリプトと checkinstall スクリプトのどちらでも、環境変数に値を割り当ててインストール環境に設定することで、新しい環境変数を定義できます。

- 次の表は、環境を通じてすべてのインストールスクリプトで使用できる環境変数の一覧です。これらの環境変数はいずれも、スクリプトでは変更できません。

環境変数	説明
CLIENT_BASEDIR	ターゲットシステムに関するベースディレクトリ。BASEDIR はインストールシステム (通常はサーバー) から特定のパッケージオブジェクトを参照する場合に使用する変数ですが、CLIENT_BASEDIR はクライアントシステムに配置されるファイルに格納するパスです。CLIENT_BASEDIR は、BASEDIR が存在する場合に存在し、PKG_INSTALL_ROOT がない場合は BASEDIR と同じです。
INST_DATADIR	現在読み取られているパッケージが配置されるディレクトリ。パッケージがテープから読み取られている場合、この変数は、パッケージがディレクトリ形式に転送された一時ディレクトリの場所を表します。つまり、パッケージ名に拡張子がないとすると (SUNWstuff.d など)、現在のパッケージの request スクリプトは $INST_DATADIR/$PKG/install にあります。
PATH	スクリプトの呼び出しでコマンドを見つけるために sh が使用する検索リスト。通常、PATH は /sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin に設定されています。
PKGINST	インストールされているパッケージのインスタンス識別子。パッケージの別のインスタンスがまだインストールされていない場合、値はパッケージの省略名 (SUNWcadap など) です。すでにインストールされている場合は、パッケージの省略名のあとに接尾辞が付加された値 (SUNWcadap.4 など) になります。
PKGSAV	削除スクリプトで使用するためにファイルを保存できるディレクトリ、または以前に保存されたファイルを見つけることのできるディレクトリ。Solaris 2.5 リリースおよびその互換リリースでのみ使用できます。
PKG_CLIENT_OS	パッケージがインストールされているクライアントのオペレーティング システム。この変数の値は Solaris です。
PKG_CLIENT_VERSION	x.y 形式の Solaris のバージョン。
PKG_CLIENT_REVISION	Solaris 構築リビジョン。
PKG_INSTALL_ROOT	パッケージがインストールされているターゲットシステムのルートファイルシステム。この変数は、pkgadd コマンドと pkgrm コマンドが -R オプションを指定して呼び出された場合にのみ存在します。このように条件付きで存在するため、手続きスクリプトでは、${PKG_INSTALL_ROOT}/*somepath* の形式で容易に使用できます。
PKG_NO_UNIFIED	pkgadd コマンドと pkgrm コマンドが -M および -R オプションを指定して呼び出された場合に設定される環境変数。この環境変数は、パッケージ環境の一部であるいずれかのパッケージインストールスクリプトまたはパッケージコマンドに渡されます。

環境変数	説明
UPDATE	この環境変数は、ほとんどのインストール環境には存在しません。この変数が存在する場合は値が yes に設定され、次の 2 つのどちらかを意味します。同じ名前、バージョン、およびアーキテクチャーのパッケージが、システムにすでに存在します。または、このパッケージは、管理者の指示により、同じ名前のインストール済みパッケージを上書きしています。これらの場合は、元のベースディレクトリが常に使用されます。

スクリプト用パッケージ情報の取得

スクリプトから 2 つのコマンドを使用して、パッケージについての情報を要求できます。

- pkginfo コマンドは、インスタンス識別子やパッケージ名など、ソフトウェアパッケージについての情報を返します。
- pkgparam コマンドは、要求された環境変数の値を返します。

  詳細は、pkginfo(1) マニュアルページ、pkgparam(1) マニュアルページ、および第 4 章「パッケージの確認と転送」を参照してください。

スクリプトの終了コード

各スクリプトは、次の表に示す終了コードのいずれかで終了する必要があります。

表 3–2 インストールスクリプトの終了コード

コード	意味
0	スクリプトが正常終了したことを表します。
1	致命的エラー。インストールプロセスは、この段階で終了します。
2	警告、またはエラーの可能性がある状態です。インストールは継続されます。警告メッセージは、インストール完了時に表示されます。
3	pkgadd コマンドがクリーンに停止されたことを表します。checkinstall スクリプトのみがこのコードを返します。
10	選択されているすべてのパッケージのインストールが完了した時点で、システムがリブートされるべきであることを表します。(この値は、1 桁の終了コードに追加されるはずです。)
20	現在のパッケージのインストールが終了したら、ただちにシステムをリブートするべきであることを表します。(この値は、1 桁の終了コードに追加されるはずです。)

インストールスクリプトによって返される終了コードの例については、第 5 章「パッケージ作成のケーススタディー」を参照してください。

注 – パッケージとともに提供されるすべてのインストールスクリプトは、prototype ファイルにエントリするようにしてください。ファイルタイプは、i (パッケージインストールスクリプトの場合) としてください。

---

request スクリプトの書き込み

request スクリプトは、パッケージをインストールしている管理者とパッケージが直接対話できる唯一の手段です。たとえば、このスクリプトを使用すると、パッケージのオプション部分をインストールするべきかどうかを、管理者に尋ねることができます。

request スクリプトの出力は、環境変数とその値のリストでなければなりません。このリストは、pkginfo ファイルで作成したいずれかのパラメータおよび CLASSES と BASEDIR パラメータを含むことができます。また、リストでは、どこでも定義されていない環境変数を使用することもできます。ただし、実際に使用するには、pkginfo ファイルでデフォルト値を提供するようにしてください。パッケージ環境変数の詳細については、24 ページの「パッケージ環境変数」を参照してください。

request スクリプトで環境変数に値を割り当てるときは、pkgadd コマンドやほかのパッケージスクリプトでこの値を使用できるようにする必要があります。

request スクリプトの動作

- request スクリプトでは、どのファイルも変更できません。このスクリプトは、パッケージをインストールしている管理者と対話し、その対話に基づいて環境変数割り当てのリストを作成するだけです。request スクリプトは、特権を持たないユーザー noaccess が存在する場合は、そのユーザーとして実行されます。それ以外の場合は、スクリプトは root として実行されます。
- pkgadd コマンドは、スクリプトの応答ファイルを示す 1 つの引数を指定して、request スクリプトを呼び出します。応答ファイルには、管理者の応答が格納されます。
- request スクリプトは、パッケージ削除時には実行されません。ただし、このスクリプトによって割り当てられた環境変数は保存され、パッケージ削除時に使用できます。

request スクリプトの設計規則

- パッケージごとに存在できる request スクリプトは 1 つだけです。スクリプトの名前は、request でなければなりません。

- 環境変数の割り当ては、pkgadd コマンドやほかのパッケージスクリプトが使用できるように、応答ファイル (スクリプトには $1 として認識される) に書き込むことで、インストール環境に追加するようにしてください。
- CLASSES および BASEDIR パラメータを除くシステム環境変数と標準インストール環境変数は、request スクリプトでは変更できません。独自に作成されたほかの環境変数はすべて変更できます。

注 – request スクリプトで BASEDIR パラメータを変更できるリリースは、Solaris 2.5 およびその互換リリース以降のみです。

- request スクリプトで操作する可能性のあるすべての環境変数に、pkginfo ファイルのデフォルト値を割り当てるようにしてください。
- 出力リストは、*PARAM=value* の形式としてください。例:

```
CLASSES=none class1
```

- request スクリプトに対しては、管理者の端末が標準入力として定義されます。
- request スクリプトでは、ターゲットシステムについての特別な分析は一切実行しないでください。特定のバイナリまたは動作がシステムに存在するかどうかをテストすること、およびその分析に基づいて環境変数を設定することは危険です。インストール時に request スクリプトが実際に実行される保証はありません。パッケージをインストールする管理者が、request スクリプトを呼び出さないで、環境変数を挿入する応答ファイルを提供する可能性があります。request スクリプトでもターゲットファイルシステムを評価している場合、その評価は行われないことがあります。特別な処理についてのターゲットシステムの分析は、checkinstall スクリプトに任せるのが最善です。

注 – パッケージをインストールする管理者が JumpStart 製品を使用する可能性がある場合は、パッケージのインストールを対話形式にしてはいけません。パッケージで request スクリプトを提供しないようにするか、インストールの前に pkgask コマンドを使用するべきであることを管理者に伝える必要があります。pkgask コマンドは、応答を request スクリプトに格納します。pkgask コマンドの詳細については、pkgask(1M) のマニュアルページを参照してください。

▼ request スクリプトを書く方法

1. 情報ファイルが格納されているディレクトリを、現在の作業用ディレクトリにします。
2. 任意のテキストエディタを使用して、**request** という名前のファイルを作成します。

3. ファイルを保存してエディタを終了します。

4. 次のいずれかのタスクを完了します。
   - 追加のインストールスクリプトを作成する場合は、次のタスク、68 ページの「ファイルシステムデータを収集する方法」に進んでください。
   - prototype ファイルを作成していない場合は、44 ページの「pkgproto コマンドを使用して prototype ファイルを作成する方法」の手順を完了して、手順 5 に進んでください。
   - prototype ファイルをすでに作成している場合は、それを編集し、前の手順で作成したインストールスクリプトのエントリを追加します。

5. パッケージを構築します。
   必要な場合は、47 ページの「パッケージの構築方法」を参照してください。

例 3–5 request スクリプトの書き込み

request スクリプトで環境変数に値を割り当てるときは、その値を pkgadd コマンドで使用できるようにする必要があります。この例では、4 つの環境変数についてこのタスクを実行する request スクリプトのセグメントを示します。 環境変数は、CLASSES、NCMPBIN、EMACS、および NCMPMAN です。これらの環境変数は、スクリプトですでに、管理者との対話セッションで定義されているものとします。

```
# make environment variables available to installation
# service and any other packaging script we might have

cat >$1 <<!
CLASSES=$CLASSES
NCMPBIN=$NCMPBIN
EMACS=$EMACS
NCMPMAN=$NCMPMAN
!
```

参照 パッケージを構築したあと、実際にインストールして、正しくインストールされることを確認し、整合性を検証します。第 4 章「パッケージの確認と転送」では、これらのタスクについて説明し、検証済みのパッケージを配布媒体に転送する方法の手順を示します。

checkinstall スクリプトでのファイルシステムデータの収集

checkinstall スクリプトは、オプションの request スクリプトの直後に実行されます。checkinstall スクリプトは、noaccess ユーザーが存在する場合はそのユーザーとして実行されます。checkinstall スクリプトには、ファイルシステムの

データを変更する権限はありません。ただし、スクリプトが収集する情報に基づいて、環境変数を作成または変更し、以降のインストールを制御できます。また、このスクリプトは、インストールプロセスをクリーンに停止することもできます。

checkinstall スクリプトは、pkgadd コマンドにとって普通ではない、ファイルシステム上の基本的なチェックを行うためのものです。たとえば、このスクリプトを使用して、現在のパッケージのいずれかのファイルが既存のファイルを上書きするかどうかを事前にチェックしたり、一般的なソフトウェアの依存関係を管理したりできます。depend ファイルは、パッケージレベルの依存関係のみを管理します。

request スクリプトとは異なり、checkinstall スクリプトは応答ファイルが提供されるかどうかに関係なく実行されます。スクリプトが存在しても、パッケージが対話型としてブランドを設定されることはありません。checkinstall スクリプトは、request スクリプトが使用できない場合、または管理的な対話が実際的ではない場合に使用できます。

注 – checkinstall スクリプトは、Solaris 2.5 およびその互換リリース以降で使用できます。

---

checkinstall スクリプトの動作

- checkinstall スクリプトは、どのファイルも変更できません。このスクリプトは、システムの状態を分析し、その対話に基づいて環境変数割り当てのリストを作成するだけです。この制限を強制するため、checkinstall スクリプトは、特権を持たないユーザー noaccess が存在する場合は、このユーザーとして実行されます。このユーザーが存在しない場合は、特権を持たないユーザー nobody として実行されます。checkinstall スクリプトには、スーパーユーザー権限はありません。
- pkgadd コマンドは、スクリプトの応答ファイルを示す 1 つの引数を指定して、checkinstall スクリプトを呼び出します。スクリプトの応答ファイルは、管理者の応答を格納するファイルです。
- checkinstall スクリプトは、パッケージ削除時には実行されません。ただし、このスクリプトによって割り当てられた環境変数は保存され、パッケージ削除時に使用できます。

checkinstall スクリプトの設計規則

- パッケージごとに存在できる checkinstall スクリプトは 1 つだけです。スクリプトの名前は、checkinstall でなければなりません。
- 環境変数の割り当ては、pkgadd コマンドやほかのパッケージスクリプトが使用できるように、応答ファイル (スクリプトには $1 として認識される) に書き込むことで、インストール環境に追加するようにしてください。

- CLASSES および BASEDIR パラメータを除くシステム環境変数と標準インストール環境変数は、checkinstall スクリプトでは変更できません。独自に作成されたほかの環境変数はすべて変更できます。
- checkinstall スクリプトで操作する可能性のあるすべての環境変数に、pkginfo ファイルでデフォルト値を割り当てるようにしてください。
- 出力リストは、*PARAM=value* の形式としてください。例:

```
CLASSES=none class1
```

- checkinstall スクリプトの実行中は、管理者と対話することはできません。管理者との対話はすべて、request スクリプトに制限されます。

▼ ファイルシステムデータを収集する方法

1. 情報ファイルが格納されているディレクトリを、現在の作業用ディレクトリにします。
2. 任意のテキストエディタを使用して、**checkinstall** という名前のファイルを作成します。
3. ファイルを保存してエディタを終了します。
4. 次のいずれかのタスクを完了します。
   - 追加のインストールスクリプトを作成する場合は、次のタスク、70 ページの「手続きスクリプトを書く方法」に進んでください。
   - prototype ファイルを作成していない場合は、44 ページの「pkgproto コマンドを使用して prototype ファイルを作成する方法」の手順を完了して、手順 5 に進んでください。
   - prototype ファイルをすでに作成している場合は、それを編集し、前の手順で作成したインストールスクリプトのエントリを追加します。
5. パッケージを構築します。
   必要な場合は、47 ページの「パッケージの構築方法」を参照してください。

例 3–6 checkinstall スクリプトの書き込み

checkinstall スクリプトのこの例では、SUNWcadap パッケージで必要なデータベースソフトウェアがインストールされているかどうかをチェックします。

```
# checkinstall script for SUNWcadap
#
# This confirms the existence of the required specU database
```

```
# First find which database package has been installed.
pkginfo -q SUNWspcdA    # try the older one

if [ $? -ne 0 ]; then
   pkginfo -q SUNWspcdB    # now the latest

      if [ $? -ne 0 ]; then    # oops
            echo "No database package can be found. Please install the"
            echo "SpecU database package and try this installation again."
            exit 3        # Suspend
      else
            DBBASE="`pkgparam SUNWsbcdB BASEDIR`/db"    # new DB software
      fi
else
      DBBASE="`pkgparam SUNWspcdA BASEDIR`/db"    # old DB software
fi

# Now look for the database file we will need for this installation
if [ $DBBASE/specUlatte ]; then
      exit 0        # all OK
else
      echo "No database file can be found. Please create the database"
      echo "using your installed specU software and try this"
      echo "installation again."
      exit 3        # Suspend
fi
```

参照　パッケージを構築したあと、実際にインストールして、正しくインストールされることを確認し、整合性を検証します。第4章「パッケージの確認と転送」では、これらのタスクについて説明し、検証済みのパッケージを配布媒体に転送する方法の手順を示します。

手続きスクリプトの書き込み

手続きスクリプトは、パッケージのインストールまたは削除に対する特定の段階で実行する一連の命令を提供します。4つの手続きスクリプトには、命令をいつ実行するかに応じて、定義済みの名前のいずれかを設定する必要があります。スクリプトは、引数なしで実行されます。

- `preinstall` スクリプト

  クラスのインストールが開始する前に実行します。このスクリプトではファイルをインストールするべきではありません。

- `postinstall` スクリプト

  すべてのボリュームがインストールされたあとで実行します。

- `preremove` スクリプト

  クラスの削除が開始する前に実行します。このスクリプトではファイルを削除するべきではありません。

- postremove スクリプト
  すべてのクラスが削除されたあとで実行します。

手続きスクリプトの動作

手続きスクリプトは、uid=root および gid=other として実行されます。

手続きスクリプトの設計規則

- 各スクリプトは、パッケージ内のボリュームごとに 1 回実行されるので、1 回を超えて実行できるようにしてください。つまり、あるスクリプトの実行結果は、同じ入力であれば何度実行しても常に同じであることを意味します。
- pkgmap ファイル内にないパッケージオブジェクトをインストールする各手続きスクリプトは、installf コマンドを使用して、パス名を追加または変更することをパッケージデータベースに通知する必要があります。すべての追加または変更が完了したあとでは、-f オプションを指定してこのコマンドを呼び出すようにしてください。この方法でパッケージオブジェクトをインストールできるスクリプトは、postinstall および postremove スクリプトのみです。詳細は、installf(1M) のマニュアルページおよび第 5 章「パッケージ作成のケーススタディー」を参照してください。
- 手続きスクリプトの実行中は、管理者と対話することはできません。管理者との対話はすべて、request スクリプトに制限されます。
- pkgmap ファイルからインストールされたものではないファイルを削除する各手続きスクリプトは、removef コマンドを使用して、パス名を削除することをパッケージデータベースに通知する必要があります。削除が完了したあとでは、-f オプションを指定してこのコマンドを呼び出すようにしてください。詳細および例については、removef(1M) のマニュアルページおよび第 5 章「パッケージ作成のケーススタディー」を参照してください。

注 – 手続きスクリプトは pkgmap ファイルにリストされているパス名と自動的には関連付けられないので、installf および removef コマンドを使用する必要があります。

▼ 手続きスクリプトを書く方法

1. 情報ファイルが格納されているディレクトリを、現在の作業用ディレクトリにします。
2. 任意のテキストエディタを使用して、1 つ以上の手続きスクリプトを作成します。
   手続きスクリプトの名前は、定義済みの名前のいずれかである必要があります。つまり、preinstall、postinstall、preremove、または postremove です。

3 ファイルを保存してエディタを終了します。

4 次のいずれかのタスクを完了します。

- クラスアクションスクリプトを作成する場合は、次のタスク、79 ページの「クラスアクションスクリプトを書く方法」に進んでください。
- prototype ファイルを作成していない場合は、44 ページの「pkgproto コマンドを使用して prototype ファイルを作成する方法」の手順を完了して、手順 5に進んでください。
- prototype ファイルをすでに作成している場合は、それを編集し、前の手順で作成した各インストールスクリプトのエントリを追加します。

5 パッケージを構築します。

必要な場合は、47 ページの「パッケージの構築方法」を参照してください。

参照 パッケージを構築したあと、実際にインストールして、正しくインストールされることを確認し、整合性を検証します。第 4 章「パッケージの確認と転送」では、これらのタスクについて説明し、検証済みのパッケージを配布媒体に転送する方法の手順を示します。

クラスアクションスクリプトの書き込み

オブジェクトクラスの定義

オブジェクトクラスを使用すると、インストール時または削除時に、パッケージオブジェクトのグループに対して一連のアクションを実行できます。クラスへのオブジェクトの割り当ては、prototype ファイルで行います。すべてのパッケージオブジェクトにはクラスを指定する必要がありますが、特別なアクションを必要としないオブジェクトには、デフォルトで none クラスが使用されます。

pkginfo ファイルで定義されているインストールパラメータ CLASSES は、インストールするクラスのリストです (none クラスを含みます)。

注 – pkgmap ファイルで定義されていても、pkginfo ファイルのこのパラメータにリストされていないクラスに属するオブジェクトは、インストールされません。

CLASSES のリストで、インストールの順序が判定されます。none クラスがある場合は、常に最初にインストールされて、最後に削除されます。ディレクトリはほかのすべてのシステムオブジェクトに対する基礎となるサポート構造なので、すべてのディレクトリは none クラスに割り当てられるようにしてください。例外がある場合もありますが、一般的に none クラスが最も安全です。このようにすること

で、ディレクトリに格納されるオブジェクトより前に、確実にディレクトリが作成されます。また、空になっていないディレクトリの削除が試みられることがありません。

パッケージインストール時のクラスの処理方法

次に、クラスのインストール時にシステムが実行するアクションについて説明します。アクションは、パッケージのボリュームごとに、そのボリュームのインストール時に 1 度行われます。

1. pkgadd コマンドが、パス名のリストを作成します。

   pkgadd コマンドは、アクションスクリプトの対象となるパス名のリストを作成します。このリストの各行には、ソースパス名とターゲットパス名がスペースで区切られて記述されています。ソースパス名は、インストールされるオブジェクトがインストールボリューム上で常駐する場所を示します。ターゲットパス名は、オブジェクトがインストールされるべきターゲットシステム上の場所を示します。リストの内容は、次の条件によって制限されます。

   - リストには、関連付けられたクラスに属するパス名のみが含まれます。
   - パッケージオブジェクトの作成が失敗すると、リストに含まれるディレクトリ、名前付きパイプ、文字デバイス、ブロックデバイス、およびシンボリックリンクには、/dev/null というソースパス名が設定されます。通常、これらのアイテムは pkgadd コマンドによって自動的に作成され (まだ存在しない場合)、pkgmap ファイルでの定義に従って適切な属性 (モード、所有者、グループ) が設定されます。
   - ファイルタイプが l のリンクファイルは、どのような場合にもリストには追加されません。特定のクラスのハードリンクは、以降の 4 番目のアクションで作成されます。

2. 特定のクラスのインストールに対してクラスアクションスクリプトが提供されない場合は、生成されるリストのパス名が、ボリュームから適切なターゲットの場所にコピーされます。
3. クラスアクションスクリプトが存在する場合は実行されます。

   クラスアクションスクリプトは、1 番目のアクションで生成されたリストを含む標準入力で呼び出されます。パッケージで最後のボリュームの場合、またはクラスで最後のオブジェクトの場合は、スクリプトは単一の引数 ENDOFCLASS を指定して実行されます。

   注 – このクラスの通常ファイルがパッケージ内に存在しない場合でも、クラスアクションスクリプトは、空のリストと ENDOFCLASS 引数で少なくとも 1 回は呼び出されます。

   ---

4. pkgadd コマンドがコンテンツと属性の監査を実行し、ハードリンクを作成します。

2 番目または 3 番目のアクションが正常に実行されたあと、pkgadd コマンドはパス名のリストについて内容と属性の情報を監査します。pkgadd コマンドは、クラスと関連付けられたリンクを自動的に作成します。生成されたリストのすべてのパス名について、検出された属性の不整合が修正されます。

パッケージ削除時のクラスの処理方法

オブジェクトはクラスごとに削除されます。パッケージに存在していても CLASSES パラメータに含まれないクラスが、最初に削除されます (たとえば、installf コマンドでインストールされたオブジェクト)。CLASSES パラメータにリストされているクラスが、逆の順序で削除されます。none クラスは、常に最後に削除されます。次に、クラスの削除時に行われるシステムのアクションについて説明します。

1. pkgrm コマンドが、パス名のリストを作成します。

   pkgrm コマンドは、指定されたクラスに属するインストール済みのパス名のリストを作成します。別のパッケージによって参照されているパス名は、ファイルタイプが e であるものを除き、リストから除外されます。e というファイルタイプは、インストール時または削除時にファイルを編集するべきであることを意味します。

   削除されるパッケージがインストール時にタイプ e のいずれかのファイルを変更していた場合は、そのときに追加した行だけを削除するようにしてください。空ではない編集可能なファイルは削除しないでください。パッケージが追加した行を削除します。

2. クラスアクションスクリプトが存在しない場合は、パス名が削除されます。

   パッケージにクラスに対する削除クラスアクションスクリプトが存在しない場合は、pkgrm コマンドによって生成されたリストのすべてのパス名が削除されます。

   注 – ファイルタイプが e (編集可能) のファイルは、クラスおよび関連するクラスアクションスクリプトに割り当てられません。これらのファイルは、パス名がほかのパッケージと共有されている場合であっても、この時点で削除されます。

   ---

3. クラスアクションスクリプトが存在する場合は、実行されます。

   pkgrm コマンドが、1 番目のアクションで生成されたリストを含む、スクリプトに対する標準入力でクラスアクションスクリプトを呼び出します。

4. pkgrm コマンドが監査を実行します。

   クラスアクションスクリプトの実行に成功したあと、pkgrm コマンドは、パス名が別のパッケージによって参照されていない場合は、パッケージデータベースからパス名への参照を削除します。

クラスアクションスクリプト

クラスアクションスクリプトは、パッケージのインストールまたは削除時に実行される一連のアクションを定義しています。アクションは、クラス定義に基づいてパス名のグループに対して実行されます。クラスアクションスクリプトの例については、第 5 章「パッケージ作成のケーススタディー」を参照してください。

クラスアクションスクリプトの名前は、対象となるクラス、およびこれらの操作が、パッケージのインストール時や削除時に実行されるべきかどうかに基づきます。次の表では、2 種類の名前形式を示します。

名前の形式	説明
`i.`*class*	パッケージインストール時に、示されているクラスのパス名に対して実行されます。
`r.`*class*	パッケージ削除時に、示されているクラスのパス名に対して実行されます。

たとえば、`manpage` という名前のクラスのインストールスクリプトの名前は、`i.manpage` となります。削除スクリプトは、`r.manpage` という名前になります。

注 – このファイル名形式は、`sed`、`awk`、`build` の各システムクラスに属するファイルには使用されません。これらの特殊なクラスの詳細については、75 ページの「特殊なシステムクラス」を参照してください。

クラスアクションスクリプトの動作

- クラスアクションスクリプトは、`uid=root` および `gid=other` として実行されます。
- スクリプトは、現在のボリューム上の指定されたクラスに含まれるすべてのファイルに対して実行されます。
- `pkgadd` と `pkgrm` コマンドは、クラスに属する `pkgmap` ファイルにリストされているすべてのオブジェクトのリストを作成します。結果として、クラスアクションスクリプトは、特定のクラスに属する `pkgmap` で定義されているパス名に対してのみ実行できます。
- クラスアクションスクリプトは、最後に実行されるときには (つまり、そのクラスに属しているクラスがそれ以上ないとき)、キーワード引数 `ENDOFCLASS` を指定して実行されます。
- クラスアクションスクリプトの実行中は、管理者と対話することはできません。

クラスアクションスクリプトの設計規則

- パッケージが複数のボリュームに分かれている場合、クラスアクションスクリプトはクラスに属するファイルが 1 つでも含まれるボリュームごとに 1 回ずつ実行されます。したがって、各スクリプトは 2 回以上実行できるようになっている必要があります。つまり、あるスクリプトの実行結果は、同じ入力であれば何度実行しても同じになるということを意味します。
- あるファイルがクラスアクションスクリプトを持つクラスの一部である場合、スクリプトはそのファイルをインストールする必要があります。`pkgadd` コマンドは、クラスアクションスクリプトが存在するファイルをインストールしませんが、インストールを検証します。
- クラスアクションスクリプトでは、`pkgadd` コマンドで生成されるリストに出現しないパス名またはシステム属性を追加、削除、または変更してはいけません。このリストの詳細については、72 ページの「パッケージインストール時のクラスの処理方法」の手順 1 を参照してください。
- スクリプトで `ENDOFCLASS` 引数を検出した場合は、クリーンアップなどの事後処理アクションをスクリプトに組み込みます。
- 管理者との対話はすべて、`request` スクリプトに制限されます。クラスアクションスクリプトを使用して管理者から情報の取得を試みないでください。

特殊なシステムクラス

システムには 4 つの特殊なクラスが用意されています。

- `sed` クラス

  `sed` 命令を使用して、パッケージのインストールおよび削除時にファイルを編集するための方法を提供します。

- `awk` クラス

  `awk` 命令を使用して、パッケージのインストールおよび削除時にファイルを編集するための方法を提供します。

- `build` クラス

  Bourne シェルコマンドを使用して、ファイルを動的に構成または変更するための方法を提供します。

- `preserve` クラス

  今後のパッケージインストールで上書きされるべきでないファイルを保持する方法を提供します。

- `manifest` クラス

  マニフェストに関連する SMF (サービス管理機能) サービスの自動的なインストールおよびアンインストールを提供します。パッケージ内のすべての SMF マニフェストに、`manifest` クラスが使用されなければなりません。

パッケージ内の複数のファイルで必要な特殊な処理が、sed、awk、または sh コマンドを使用して完全に定義できる場合は、システムクラスを使用すると、複数のクラスとそれに対応するクラスアクションスクリプトを使用するより、インストールの時間を短縮できます。

sed クラススクリプト

sed クラスは、ターゲットシステム上の既存オブジェクトを変更する方法を提供します。sed クラスアクションスクリプトは、sed クラスに属するファイルが存在する場合は、インストール時に自動的に実行されます。sed クラスアクションスクリプトの名前は、命令が実行される対象のファイルの名前と同じであるようにしてください。

sed クラスアクションスクリプトは、次の形式で sed 命令を提供します。

2 つのコマンドが、命令を実行する必要があるときを示します。!install コマンドに続く sed 命令は、パッケージのインストール時に実行されます。!remove コマンドに続く sed 命令は、パッケージの削除時に実行されます。ファイル内でこれらのコマンドが使用される順序は関係ありません。

sed 命令の詳細については、sed(1) のマニュアルページを参照してください。sed クラスアクションスクリプトの例については、第 5 章「パッケージ作成のケーススタディー」を参照してください。

awk クラススクリプト

awk クラスは、ターゲットシステム上の既存オブジェクトを変更する方法を提供します。変更は、awk クラスアクションスクリプト内の awk 命令として提供されます。

awk クラスアクションスクリプトは、awk クラスに属するファイルが存在する場合、インストール時に自動的に実行されます。このようなファイルには、awk クラススクリプトに対する命令が次の形式で含まれます。

2 つのコマンドが、命令を実行する必要があるときを示します。!install コマンドに続く awk 命令は、パッケージのインストール時に実行されます。!remove コマンドに続く命令は、パッケージの削除時に実行されます。これらのコマンドは、任意の順序で使用できます。

awk クラスアクションスクリプトの名前は、命令が実行される対象のファイルの名前と同じであるようにしてください。

変更対象のファイルは、awk コマンドに対する入力として使用され、スクリプトの出力は最終的に元のオブジェクトを置き換えます。この構文では、環境変数を awk コマンドに渡すことはできません。

awk 命令の詳細については、awk(1) のマニュアルページを参照してください。

build クラススクリプト

build クラスは、Bourne シェルの命令を実行して、パッケージオブジェクトファイルを作成または変更します。これらの命令は、パッケージオブジェクトとして提供されます。パッケージオブジェクトが build クラスに属している場合は、インストール時に命令が自動的に実行されます。

build クラスアクションスクリプトの名前は、命令が実行される対象のファイルの名前と同じであるようにしてください。また、名前は sh コマンドで実行できる必要もあります。スクリプトの出力は、構築または変更されると新しいバージョンのファイルになります。スクリプトが出力を生成しない場合、ファイルは作成または変更されません。したがって、スクリプトはファイル自体を変更または作成できます。

たとえば、パッケージがデフォルトファイル /etc/randomtable を提供し、ファイルがターゲットシステム上にまだ存在しない場合は、prototype ファイルのエントリは次のような内容です。

```
e build /etc/randomtable ? ? ?
```

また、パッケージオブジェクト /etc/randomtable は、次のような内容です。

```
!install
# randomtable builder
if [ -f $PKG_INSTALL_ROOT/etc/randomtable ]; then
        echo "/etc/randomtable is already in place.";
        else
        echo "# /etc/randomtable" > $PKG_INSTALL_ROOT/etc/randomtable
        echo "1121554    # first random number" >> $PKG_INSTALL_ROOT/etc/randomtable
fi

!remove
# randomtable deconstructor
if [ -f $PKG_INSTALL_ROOT/etc/randomtable ]; then
        # the file can be removed if it's unchanged
        if [ egrep "first random number" $PKG_INSTALL_ROOT/etc/randomtable ]; then
            rm $PKG_INSTALL_ROOT/etc/randomtable;
        fi
fi
```

build クラスを使用する別の例については、第 5 章「パッケージ作成のケーススタディー」を参照してください。

preserve クラススクリプト

preserve クラスは、パッケージのインストール時に既存のファイルを上書きするべきかどうかを決定することによって、パッケージオブジェクトファイルを保持します。preserve クラススクリプトを使用する場合、2 つの可能なシナリオは次のとおりです。

- インストールするファイルがターゲットディレクトリにまだ存在しない場合は、ファイルは正常にインストールされます。
- インストールするファイルがターゲットディレクトリに存在する場合は、ファイルが存在することを示すメッセージが表示されて、ファイルはインストールされません。

どちらのシナリオの結果も、preserve スクリプトとしては成功と見なされます。失敗は、2 番目のシナリオでのみ発生し、ファイルをターゲットディレクトリにコピーできない場合です。

Solaris 7 リリース以降、i.preserve スクリプトおよびこのスクリプトのコピー i.CONFIG.prsv は、ほかのクラスアクションスクリプトとともに、/usr/sadm/install/scripts ディレクトリに置かれています。

保持するファイル名を含むには、スクリプトを変更します。

manifest クラススクリプト

manifest クラスは、SMF マニフェストに関連する SMF (サービス管理機能) サービスを自動的にインストールおよびアンインストールします。SMF に精通していない場合は、『Oracle Solaris の管理: 基本管理』の第 18 章「サービスの管理 (概要)」で、SMF を使用してサービスを管理する方法に関する情報を参照してください。

パッケージ内のすべてのサービスマニフェストは、クラス manifest によって識別されます。サービスマニフェストのインストールと削除を行うクラスアクションスクリプトは、パッケージ化サブシステムに含まれています。pkgadd(1M) が呼び出されると、サービスマニフェストがインポートされます。pkgrm(1M) が呼び出されると、無効になっているサービスマニフェスト内のインスタンスが削除されます。また、インスタンスが残っていないマニフェスト内のサービスもすべて削除されます。pkgadd(1M) または pkgrm(1M) に -R オプションを指定した場合、これらのサービスマニフェストアクションは、次にシステムが代替ルートパスでリブートしたときに実行されます。

次のパッケージ情報ファイルのコードの一部では、manifest クラスの使用が示されています。

```
# packaging files
i pkginfo
i copyright
i depend
i preinstall
i postinstall
#
# source locations relative to the prototype file
#
d none var 0755 root sys
d none var/svc 0755 root sys
```

```
d none var/svc/manifest 0755 root sys
d none var/svc/manifest/network 0755 root sys
d none var/svc/manifest/network/rpc 0755 root sys
f manifest var/svc/manifest/network/rpc/smserver.xml 0444 root sys
```

注 – これらのクラスアクションスクリプトは Oracle Solaris OS の一部であり、リリース間で異なるため、パッケージには `i.manifest` および `r.manifest` ファイルを含めないでください。これらのファイルを含めると、異なる Oracle Solaris リリース間のパッケージの移植性が低くなります。

▼ クラスアクションスクリプトを書く方法

1. 情報ファイルが格納されているディレクトリを、現在の作業用ディレクトリにします。

2. **prototype** ファイル内のパッケージオブジェクトに、目的のクラス名を割り当てます。

   たとえば、`application` および `manpage` クラスへのオブジェクトの割り当ては、次のような内容です。

   ```
   f manpage /usr/share/man/manl/myappl.1l
   f application /usr/bin/myappl
   ```

3. **pkginfo** ファイル内の **CLASSES** パラメータを変更し、パッケージで使用するクラス名を追加します。

   たとえば、`application` および `manpage` クラスのエントリは次のような内容です。

   ```
   CLASSES=manpage application none
   ```

   注 – `none` クラスは、`CLASSES` パラメータの定義での出現位置に関係なく、常に最初にインストールされて、最後に削除されます。

   ---

4. **sed**、**awk**、 **build** のいずれかのクラスに属するファイルのクラスアクションスクリプトを作成している場合は、パッケージオブジェクトを含むディレクトリを、現在の作業用ディレクトリにします。

5. クラスアクションスクリプトまたはパッケージオブジェクト (**sed**、**awk**、または **build** クラスに属すファイルの場合) を作成します。

   たとえば、`application` という名前のクラスに対するインストールスクリプトは `i.application` という名前にし、削除スクリプトは `r.application` という名前にします。

   あるファイルがクラスアクションスクリプトを持つクラスの一部である場合、スクリプトはそのファイルをインストールする必要があります。`pkgadd` コマンドは、ク

ラスアクションスクリプトが存在するファイルをインストールしませんが、インストールを検証します。また、クラスを定義しても、クラスアクションスクリプトを提供しないと、そのクラスに対して行われるアクションは、インストールメディアからターゲットシステムへのコンポーネントのコピーだけです (pkgadd のデフォルトの動作)。

6 次のいずれかのタスクを完了します。

- prototype ファイルを作成していない場合は、44 ページの「pkgproto コマンドを使用して prototype ファイルを作成する方法」の手順を完了し、手順 7 に進んでください。
- prototype ファイルをすでに作成している場合は、それを編集し、前の手順で作成した各インストールスクリプトのエントリを追加します。

7 パッケージを構築します。

必要な場合は、47 ページの「パッケージの構築方法」を参照してください。

参考 次のステップ

パッケージを構築したあと、実際にインストールして、正しくインストールされることを確認し、整合性を検証します。第 4 章「パッケージの確認と転送」では、この作業について説明し、検証済みのパッケージを配布媒体に転送する方法の手順を示します。

署名付きパッケージの作成

署名付きパッケージを作成するプロセスには、複数の手順が含まれ、新しい概念と用語を理解する必要があります。このセクションでは、署名付きパッケージ、その用語、および証明書管理に関して説明します。このセクションでは、署名付きパッケージの作成手順についても説明します。

署名付きパッケージ

署名付きパッケージは、次のことを証明するデジタル署名 (次に定義する、PEM でエンコードされた PKCS7 デジタル署名) の付いた通常のストリーム形式のパッケージです。

- そのパッケージに署名したエンティティーがそのパッケージの作成者である。
- そのエンティティーが実際にそのパッケージに署名した。
- そのパッケージがエンティティーによる署名後に変更されていない。
- そのパッケージに署名したエンティティーが信頼されたエンティティーである。

署名付きパッケージは、署名が付いている点以外は、署名なしパッケージと同一です。署名付きパッケージと署名なしパッケージは、バイナリレベルで互換性があります。したがって、署名付きパッケージは古いバージョンのパッケージツールで使用できます。ただし、その場合、署名は無視されます。

署名付きパッケージ技術には新しい用語と省略名がいくつかあり、それについて次の表で説明します。

用語	定義
ASN.1	Abstract Syntax Notation 1 - 抽象オブジェクトを表現する方法。たとえば、ASN.1 では、公開鍵証明書、証明書を構成するすべてのオブジェクト、オブジェクトの収集順序などが定義されています。ただし、ASN.1 では、オブジェクトを保存用または転送用に直列化する方法は定義されていません。
X.509	ITU-T Recommendation X.509 - 広く採用されている X.509 公開鍵証明書の構文を指定します。
DER	Distinguished Encoding Rules - ASN.1 オブジェクトのバイナリ表現であり、コンピューティング環境で保存用または転送用に ASN.1 オブジェクトを直列化する方法を定義しています。
PEM	Privacy Enhanced Message - Base 64 エンコーディングおよびオプションのヘッダーを使用して、(DER または別のバイナリ形式の) ファイルをエンコードする方法。PEM はもともと、MIME タイプの電子メールメッセージをエンコードするために使用されました。また、PEM は、証明書と非公開鍵をファイルシステム上または電子メールメッセージ内のファイルに符号化する際にも広く使用されています。
PKCS7	Public Key Cryptography Standard #7 - デジタル署名やデジタル封筒などの暗号化データに対する汎用的な構文を定めた規格です。署名付きパッケージには、埋め込まれた PKCS7 署名が含まれます。この署名には少なくとも、パッケージの暗号化されたダイジェストと署名者の X.509 公開鍵証明書が含まれています。また、署名付きパッケージはチェーン証明書を含むこともできます。チェーン証明書は、署名者の証明書からローカルに保存された信頼できる証明書まで、信頼の連鎖を形成するときに使用できます。
PKCS12	Public Key Cryptography Standard #12 - この規格では、暗号化されたオブジェクトをディスクに保存するための構文が規定されています。パッケージのキーストアは、この形式で保持されます。
パッケージキーストア	パッケージツールを使用して照会できる証明書と鍵のリポジトリ。

証明書管理

署名付きパッケージを作成するには、先にパッケージキーストアが存在している必要があります。このパッケージキーストアには、証明書がオブジェクトの形式で含まれます。パッケージキーストアには、2 種類のオブジェクトが存在します。

信頼できる証明書
: 信頼できる証明書には、別のエンティティーに属する単一の公開鍵証明書が含まれます。信頼できる証明書という呼び名は、証明書に含まれる公開鍵が、その証明書の「サブジェクト」(所有者) によって示された本人のものであることを、キーストアの所有者が信頼することに由来しています。この信頼を表明するために、証明書の発行者はその証明書に署名します。

  信頼できる証明書は、署名を検証するとき、およびセキュリティー保護されたサーバーへの接続 (SSL) を開始するときに使用されます。

ユーザー鍵
: ユーザー鍵は、暗号鍵に関する機密情報を保持します。この情報は、不正なアクセスを防ぐために、セキュリティーが施された形式で格納されます。ユーザー鍵は、ユーザーの非公開鍵と対応する公開鍵証明書から構成されます。

  ユーザー鍵は、署名付きパッケージを作成するときに使用されます。

デフォルトでは、パッケージキーストアは `/var/sadm/security` ディレクトリに格納されます。個別のユーザーも、独自のキーストアをデフォルトで `$HOME/.pkg/security` ディレクトリに格納できます。

ディスク上でのパッケージキーストアには、2 種類の形式があります。つまり、複数ファイル形式と単一ファイル形式です。複数ファイル形式は、オブジェクトを複数のファイルに格納します。オブジェクトの種類ごとに、異なるファイルに保存されます。これらのファイルはすべて、同じパスフレーズを使用して暗号化される必要があります。単一ファイルキーストアは、すべてのオブジェクトをファイルシステムの単一のファイルに格納します。

証明書とパッケージキーストアの管理に使用する主なユーティリティは、`pkgadm` コマンドです。次に、パッケージキーストアの管理に使用される一般的なタスクについて説明します。

パッケージキーストアへの信頼できる証明書の追加

信頼できる証明書をパッケージキーストアに追加するには、`pkgadm` コマンドを使用します。PEM または DER の形式の証明書を使用できます。例:

```
$ pkgadm addcert -t /tmp/mytrustedcert.pem
```

この例では、mytrustedcert.pem という名前の PEM 形式の証明書を、パッケージキーストアに追加します。

パッケージキーストアへのユーザー証明書と非公開鍵の追加

pkgadm コマンドは、ユーザー証明書または非公開鍵は生成しません。ユーザー証明書と非公開鍵は、通常、Verisign などの認証局から入手します。または、自己署名付き証明書としてローカルで生成します。入手した鍵と証明書は、pkgadm コマンドを使用してパッケージキーストアにインポートできます。例:

```
pkgadm addcert -n myname -e /tmp/myprivkey.pem /tmp/mypubcert.pem
```

この例では、次のオプションを使用しています。

-n *myname*	パッケージキーストアに含まれる対象のエンティティー (*myname*) を特定します。*myname* エンティティーは、オブジェクトが格納される別名になります。
-e /tmp/*myprivkey.pem*	非公開鍵を含むファイルを指定します。この場合、ファイルは *myprivkey.pem* であり、/tmp ディレクトリにあります。
/tmp/*mypubcert.pem*	*mypubcert.pem* という名前の PEM 形式の証明書ファイルを指定します。

パッケージキーストアの内容の確認

pkgadm コマンドは、パッケージキーストアの内容の表示にも使用します。例:

```
$ pkgadm listcert
```

このコマンドは、パッケージキーストアに含まれる信頼できる証明書と非公開鍵を表示します。

パッケージキーストアからの信頼できる証明書と非公開鍵の削除

pkgadm コマンドを使用すると、信頼できる証明書と非公開鍵をパッケージキーストアから削除できます。

ユーザー証明書を削除するときは、証明書と鍵のペアの別名を指定する必要があります。例:

```
$ pkgadm removecert -n myname
```

証明書の別名は証明書の共通名であり、pkgadm listcert コマンドを使用して識別できます。たとえば、次のコマンドは、Trusted CA Cert 1 という名前の信頼できる証明書を削除します。

```
$ pkgadm removecert -n "Trusted CA Cert 1"
```

注 – 信頼できる証明書とユーザー証明書を同じ別名で保存した場合は、-n オプションを指定するとどちらも削除されます。

署名付きパッケージの作成

署名付きパッケージ作成のプロセスは、3 つの基本手順から成ります。

1. 署名なしディレクトリ形式パッケージの作成。
2. 署名証明書、CA 証明書、および非公開鍵のパッケージキーストアへのインポート。
3. 手順 2 の証明書による手順 1 のパッケージへの署名。

注 – パッケージツールでは証明書は作成されません。これらの証明書は、Verisign や Thawte などの認証局から入手する必要があります。

次に、署名付きパッケージ作成の各手順について説明します。

▼ 署名なしディレクトリ形式パッケージを作成する方法

署名なしディレクトリ形式パッケージを作成する手順は、すでに説明した通常のパッケージの作成手順と同じです。次の手順では、この署名なしディレクトリ形式パッケージを作成するプロセスを説明します。詳細については、パッケージの構築に関する前のセクションを参照してください。

1 **pkginfo ファイルを作成します。**

pkginfo ファイルは、次の基本的な内容となるようにしてください。

```
PKG=SUNWfoo
BASEDIR=/
NAME=My Test Package
ARCH=sparc
VERSION=1.0.0
CATEGORY=application
```

2 **prototype** ファイルを作成します。

prototye ファイルは、次の基本的な内容となるようにしてください。

```
$cat prototype
i pkginfo
d none usr 0755 root sys
d none usr/bin 0755 root bin
f none usr/bin/myapp=/tmp/myroot/usr/bin/myapp 0644 root bin
```

3 オブジェクトソースディレクトリの内容を一覧表示します。

例:

```
$ ls -lR /tmp/myroot
```

出力は次のような内容です。

```
/tmp/myroot:
total 16
drwxr-xr-x   3 abc      other        177 Jun  2 16:19 usr

/tmp/myroot/usr:
total 16
drwxr-xr-x   2 abc      other        179 Jun  2 16:19 bin

/tmp/myroot/usr/bin:
total 16
-rw-------   1 abc      other       1024 Jun  2 16:19 myapp
```

4 署名なしパッケージを作成します。

```
pkgmk -d ‘pwd‘
```

出力は次のような内容です。

```
## Building pkgmap from package prototype file.
## Processing pkginfo file.
WARNING: parameter <PSTAMP> set to "syrinx20030605115507"
WARNING: parameter <CLASSES> set to "none"
## Attempting to volumize 3 entries in pkgmap.
part  1 -- 84 blocks, 7 entries
## Packaging one part.
/tmp/SUNWfoo/pkgmap
/tmp/SUNWfoo/pkginfo
/tmp/SUNWfoo/reloc/usr/bin/myapp
## Validating control scripts.
## Packaging complete.
```

現在のディレクトリにパッケージが存在するようになります。

▼ 証明書をパッケージキーストアにインポートする方法

インポートする証明書と非公開鍵は、PEM またはDER でエンコードされた X.509 である必要があります。さらに、署名する証明書を認証局証明書にリンクするいずれ

かの中間、つまり「チェーン」証明書も、パッケージに署名する前にパッケージキーストアにインポートする必要があります。

注 – 各認証局は、さまざまな形式で証明書を発行できます。PKCS12 ファイルから PEM エンコードされた X.509 ファイル (パッケージキーストアへのインポートに適したもの) に証明書と非公開鍵を抽出するには、OpenSSL などのフリーウェアの変換ユーティリティを使用します。

非公開鍵が暗号化されている場合 (通常の場合) は、パスフレーズの入力を求められます。また、生成されるパッケージキーストアを保護するためのパスワードの指定を求められます。パスワードを指定しないこともできますが、その場合は、生成されるパッケージキーストアは暗号化されません。

次の手順では、証明書が適切な形式になったあと、`pkgadm` コマンドを使用して証明書をインポートする方法について説明します。

1 **PEM または DER でエンコードされた X.509 証明書ファイルに含まれるすべての認証局証明書をインポートします。**

たとえば、`ca.pem` に含まれるすべての認証局証明書をインポートするには、次のように入力します。

```
$ pkgadm addcert -k ~/mykeystore -ty ca.pem
```

出力は次のような内容です。

```
Trusting certificate <VeriSign Class 1 CA Individual \
Subscriber-Persona Not Validated>
Trusting certificate </C=US/O=VeriSign, Inc./OU=Class 1 Public \
Primary Certification Authority
Type a Keystore protection Password.
Press ENTER for no protection password (not recommended):
For Verification: Type a Keystore protection Password.
Press ENTER for no protection password (not recommended):
Certificate(s) from <ca.pem> are now trusted
```

署名に使用する鍵をパッケージキーストアにインポートするには、あとでパッケージに署名するときに使用する別名を指定する必要があります。この別名は、パッケージキーストアから鍵を削除する場合にも使用できます。

たとえば、署名鍵を `sign.pem` ファイルからインポートするには、次のように入力します。

```
$ pkgadm addcert -k ~/mykeystore -n mycert sign.pem
```

出力は次のような内容です。

```
Enter PEM passphrase:
Enter Keystore Password:
Successfully added Certificate <sign.pem> with alias <mycert>
```

2 パッケージキーストアに証明書があることを確認します。

たとえば、前の手順で作成したキーストア内の証明書を表示するには、次のように入力します。

```
$ pkgadm listcert -k ~/mykeystore
```

▼ パッケージに署名する方法

証明書をパッケージキーストアにインポートしたら、パッケージに署名できます。パッケージに実際に署名するには、pkgtrans コマンドを使用します。

- **pkgtrans** コマンドを使用してパッケージに署名します。署名なしパッケージの場所と、パッケージに署名するための鍵の別名を指定します。

  たとえば、前の手順の例を使用すると、SUNWfoo.signed という名前の署名付きパッケージを作成するには、次のように入力します。

  ```
  $ pkgtrans -g -k ~/mykeystore -n mycert . ./SUNWfoo.signed SUNWfoo
  ```

  このコマンドの出力は、次のような内容です。

  ```
  Retrieving signing certificates from keystore </home/user/mykeystore>
  Enter keystore password:
  Generating digital signature for signer <Test User>
  Transferring <SUNWfoot> package instance
  ```

  署名付きパッケージは、SUNWfoo.signed ファイルにパッケージストリーム形式で作成されます。この署名付きパッケージは、Web サイトにコピーし、pkgadd コマンドと URL を使用してインストールするのに適しています。

◆ ◆ ◆ 第 4 章

パッケージの確認と転送

この章では、パッケージの整合性を確認する方法と、パッケージをフロッピーディスクや CD-ROM などの配布媒体に転送する方法について説明します。

この章の内容は以下のとおりです。

パッケージの確認と転送 (タスクマップ)

次の表では、パッケージの整合性を確認して、配布媒体に転送するために従うべき手順について説明します。

表 4–1 パッケージの確認と転送のタスクマップ

タスク	説明	手順
1. パッケージを構築する	ディスク上でパッケージを構築します。	第 2 章「パッケージの構築」
2. パッケージをインストールする	パッケージをインストールし、エラーなしでインストールされることを確認して、パッケージをテストします。	91 ページの「スタンドアロンシステムまたはサーバーにパッケージをインストールする方法」
3. パッケージの整合性を確認する	`pkgchk` コマンドを使用して、パッケージの整合性を確認します。	93 ページの「パッケージの整合性を確認する方法」
4. ほかのパッケージ情報を取得する	オプション。`pkginfo` コマンドおよび `pkgparam` コマンドを使用して、パッケージ固有の確認を行います。	94 ページの「インストール済みパッケージについての追加情報の表示」

表 4–1 パッケージの確認と転送のタスクマップ (続き)

タスク	説明	手順
5. インストール済みパッケージを削除する	pkgrm コマンドを使用して、インストールされているパッケージをシステムから削除します。	99 ページの「パッケージを削除する方法」
6. パッケージを配布媒体に転送する	pkgtrans コマンドを使用して、パッケージを (パッケージ形式で) 配布媒体に転送します。	100 ページの「配布媒体にパッケージを転送する方法」

ソフトウェアパッケージのインストール

ソフトウェアパッケージは、pkgadd コマンドを使用してインストールします。このコマンドを使用すると、配布媒体またはディレクトリからソフトウェアパッケージの内容が転送され、システムにインストールされます。

このセクションでは、パッケージが正しくインストールされることを確認するために、パッケージのインストール上の基本的な説明について紹介します。

インストールソフトウェアデータベース

システムにインストールされたすべてのパッケージの情報が、インストールソフトウェアデータベースに保持されます。パッケージ内のすべてのオブジェクトのエントリと、コンポーネント名、常駐場所、種類などの情報が保存されます。エントリには、コンポーネントが属するパッケージのレコード、コンポーネントを参照する可能性があるほかのパッケージ、およびパス名、コンポーネントの常駐場所、コンポーネントの種類などの情報が含まれます。エントリを自動的に追加したり削除したりするには、pkgadd および pkgrm コマンドを使用します。データベース内の情報を表示するには、pkgchk および pkginfo コマンドを使用します。

各パッケージコンポーネントには、2 種類の情報が関連付けられています。属性情報には、コンポーネント自体の内容が記述されます。たとえば、コンポーネントのアクセス権、所有者 ID、グループ ID は属性情報です。内容情報には、ファイルサイズや最終変更時間など、コンポーネントの内容が記述されます。

インストールソフトウェアデータベースは、パッケージのステータスを追跡します。パッケージは、完全にインストールされる場合 (インストールプロセスが正常に完了した場合) と、部分的にインストールされる場合 (インストールプロセスが正常に完了しなかった場合) があります。

パッケージが部分的にインストールされる場合、インストールが中止される前にパッケージの一部分がインストールされる可能性があります。したがって、パッケージの一部分はインストールされてデータベースに記録されますが、残りの部分はインストールも記録もされません。pkgadd コマンドは、データベースにアクセスしてすでにインストールされている部分を検出できるの

で、パッケージを再インストールするときに、前回のインストールが停止した場所から開始するかどうかの指示を求められます。また、pkgrm コマンドを使用して、インストールソフトウェアデータベースの情報に基づいてインストール済みの部分を削除することもできます。

pkgadd コマンドでの対話

pkgadd コマンドは、問題を検出すると、まずインストール管理ファイルで命令を確認します (詳細は、admin(4) を参照。)命令がない場合、または管理ファイルの関連するパラメータが ask に設定されている場合は、pkgadd は問題を説明するメッセージを表示して、応答を求めます。通常、プロンプトには「Do you want to continue with this installation?」と表示されます。「 yes」、「no」、または「quit」で応答してください。

複数のパッケージを指定している場合、「no」で応答すると、インストール中のパッケージのインストールは停止しますが、pkgadd ではほかのパッケージのインストールは継続されます。「quit」で応答すると、pkgadd ではすべてのパッケージのインストールが停止されます。

同機種環境内のスタンドアロンシステムまたはサーバーへのパッケージのインストール

このセクションでは、同機種環境内のスタンドアロンシステムまたはサーバーシステムにパッケージをインストールする方法について説明します。

▼ スタンドアロンシステムまたはサーバーにパッケージをインストールする方法

1. パッケージを構築します。
   必要に応じて、46 ページの「パッケージの構築」を参照してください。
2. システムにスーパーユーザーとしてログインします。
3. ソフトウェアパッケージをシステムに追加します。

   # pkgadd -d *device-name* [*pkg-abbrev*...]

   -d *device-name* — パッケージの位置を指定します。*device-name* は、完全なディレクトリパス名でも、テープ、フロッピーディスク、またはリムーバブルディスクの識別子でもかまいません。

pkg-abbrev	追加する 1 つ以上のパッケージの名前を空白で区切って指定します。省略すると、pkgadd は使用可能なすべてのパッケージをインストールします。

例 4–1 スタンドアロンまたはサーバーでのパッケージのインストール

pkgA という名前のソフトウェアパッケージを、/dev/rmt/0 という名前のテープデバイスからインストールするには、次のコマンドを入力します。

```
# pkgadd -d /dev/rmt/0 pkgA
```

次のようにパッケージ名を空白で区切ることで、同時に複数のパッケージをインストールすることもできます。

```
# pkgadd -d /dev/rmt/0 pkgA pkgB pkgC
```

パッケージが常駐しているデバイスを指定しないと、コマンドによってデフォルトのスプールディレクトリ (/var/spool/pkg) がチェックされます。パッケージが見つからないと、インストールは失敗します。

参照 この手順が終了したら、次のタスク、93 ページの「パッケージの整合性を確認する方法」に進みます。

パッケージの整合性の確認

pkgchk コマンドを使用すると、パッケージの整合性、パッケージがシステムにインストールされているか、パッケージ形式でインストールされているか (pkgadd コマンドでインストールできる状態) を確認できます。パッケージの構造またはインストールされているファイルとディレクトリを確認したり、パッケージオブジェクトについての情報を表示したりします。pkgchk コマンドは、次の内容について一覧表示したり、チェックしたりできます。

- パッケージインストールスクリプト。
- システムに現在インストールされているオブジェクトの内容または属性、またはその両方。
- スプールされている未インストールのパッケージの内容。
- 指定した pkgmap ファイルで記述されているオブジェクトの内容または属性、またはその両方。

このコマンドの詳細は、pkgchk(1M) を参照してください。

pkgchk コマンドでは、2 種類のチェックが実行されます。ファイルの属性 (ファイルのアクセス権と所有権、およびブロックまたは文字型特殊デバイスのメジャー/マイ

ナー番号) のチェックとファイルの内容 (サイズ、チェックサム、および変更日時) のチェックです。デフォルトでは、ファイルの属性と内容の両方がチェックされます。

また、pkgchk コマンドでは、インストールされているパッケージとインストールソフトウェアデータベースの間で、ファイルの属性と内容の比較も行われます。パッケージに関するエントリは、インストール時から変わる場合があります。たとえば、別のパッケージによってパッケージコンポーネントが変更されているような場合です。データベースは、その変更を反映します。

▼ パッケージの整合性を確認する方法

1 パッケージをインストールします。

必要に応じて、91 ページの「スタンドアロンシステムまたはサーバーにパッケージをインストールする方法」を参照してください。

2 パッケージの整合性を確認します。

```
# pkgchk [-v] [-R root-path] [pkg-abbrev...]
```

-v	処理されたファイルを一覧表示します。
-R *root-path*	クライアントシステムのルートファイルシステムの場所を指定します。
pkg-abbrev	チェックする 1 つ以上のパッケージの名前を空白で区切って指定します。省略すると、pkgchk では使用可能なすべてのパッケージがチェックされます。

例 4–2 パッケージの整合性の確認

この例では、インストールされているパッケージの整合性の確認に使用する必要のあるコマンドを示します。

```
$ pkgchk pkg-abbrev
$
```

エラーがある場合、pkgchk コマンドではそのエラーが出力されます。エラーがない場合は、何も出力されず、終了コード 0 が返されます。パッケージの省略名を指定しないと、システム上のすべてのパッケージがチェックされます。

または、-v オプションを使用すると、エラーがない場合にはパッケージに含まれるファイルが一覧表示されます。例:

```
$ pkgchk -v SUNWcadap
/opt/SUNWcadap
/opt/SUNWcadap/demo
```

```
/opt/SUNWcadap/demo/file1
/opt/SUNWcadap/lib
/opt/SUNWcadap/lib/file2
/opt/SUNWcadap/man
/opt/SUNWcadap/man/man1
/opt/SUNWcadap/man/man1/file3.1
/opt/SUNWcadap/man/man1/file4.1
/opt/SUNWcadap/man/windex
/opt/SUNWcadap/srcfiles
/opt/SUNWcadap/srcfiles/file5
/opt/SUNWcadap/srcfiles/file6
$
```

クライアントシステムのルートファイルシステムにインストールされているパッケージを確認する必要がある場合は、次のコマンドを使用します。

```
$ pkgchk -v -R root-path pkg-abbrev
```

参照 以上の手順が終了したら、次のタスク、99 ページの「pkginfo コマンドで情報を取得する方法」に進みます。

インストール済みパッケージについての追加情報の表示

インストール済みパッケージについての情報を表示するには、他に 2 つのコマンドがあります。

- pkgparam コマンドでは、パラメータの値が表示されます。
- pkginfo コマンドでは、インストールソフトウェアデータベースからの情報が表示されます。

pkgparam コマンド

pkgparam コマンドを使用すると、コマンド行で指定したパラメータに関連付けられた値を表示できます。この値は、特定のパッケージの pkginfo ファイル、または指定したファイルから取得されます。1 行に 1 つのパラメータ値が表示されます。値のみ、またはパラメータと値を表示できます。

▼ pkgparam コマンドで情報を取得する方法

1. パッケージをインストールします。
   必要に応じて、91 ページの「スタンドアロンシステムまたはサーバーにパッケージをインストールする方法」を参照してください。

2. パッケージについての追加情報を表示します。

   ```
   # pkgparam [-v] pkg-abbrev [param...]
   ```

-v	パラメータの名前とその値を表示します。
pkg-abbrev	特定のパッケージの名前です。
param	値を表示するパラメータを 1 つ以上指定します。

例 4–3 pkgparam コマンドでの情報の取得

たとえば、値のみを表示するには、次のコマンドを使用します。

```
$ pkgparam SUNWcadap
none
/opt
US/Mountain
/sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin
/usr/sadm/sysadm
SUNWcadap
Chip designers need CAD application software to design abc
chips.  Runs only on xyz hardware and is installed in the usr
partition.
system
release 1.0
SPARC
venus990706083849
SUNWcadap
/var/sadm/pkg/SUNWcadap/save
Jul 7 1999 09:58
$
```

パラメータとその値を表示するには、次のコマンドを使用します。

```
$ pkgparam -v SUNWcadap
pkgparam -v SUNWcadap
CLASSES='none'
BASEDIR='/opt'
TZ='US/Mountain'
PATH='/sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin'
OAMBASE='/usr/sadm/sysadm'
PKG='SUNWcadap'
NAME='Chip designers need CAD application software to design abc chips.
Runs only on xyz hardware and is installed in the usr partition.'
CATEGORY='system'
VERSION='release 1.0'
ARCH='SPARC'
PSTAMP='venus990706083849'
PKGINST='SUNWcadap'
PKGSAV='/var/sadm/pkg/SUNWcadap/save'
INSTDATE='Jul 7 1999 09:58'
$
```

特定のパラメータの値を表示する場合は、次の形式を使用します。

```
$ pkgparam SUNWcadap BASEDIR
/opt
$
```

詳細は、pkgparam(1) を参照してください。

参照　以上の手順が終了したら、次のタスク、99 ページの「パッケージを削除する方法」に進みます。

pkginfo コマンド

インストール済みパッケージについての情報を表示するには、pkginfo コマンドを使用します。このコマンドには複数のオプションがあり、表示の形式と内容をカスタマイズできます。

任意の数のパッケージインスタンスについての情報を要求できます。

pkginfo のデフォルトの表示

オプションを指定しないで pkginfo コマンドを実行すると、システムに完全にインストールされているすべてのパッケージのカテゴリ、パッケージインスタンス、およびパッケージ名が表示されます。次の例に示すように、表示はカテゴリ別に整理されます。

```
$ pkginfo
.
.
.
system      SUNWinst       Install Software
system      SUNWipc        Interprocess Communications
system      SUNWisolc      XSH4 conversion for ISO Latin character sets
application SUNWkcspf      KCMS Optional Profiles
application SUNWkcspg      KCMS Programmers Environment
application SUNWkcsrt      KCMS Runtime Environment
.
.
.
$
```

pkginfo の表示の形式のカスタマイズ

pkginfo の表示は、簡易、抽出、詳細の 3 種類の形式で取得できます。

短形式がデフォルトです。96 ページの「pkginfo のデフォルトの表示」に示すように、簡易形式では、カテゴリ、パッケージの省略名、およびパッケージの完全な名前のみが表示されます。

抽出形式では、パッケージの省略名、名前、アーキテクチャー(ある場合)、およびバージョン (ある場合) が表示されます。次の例のように、抽出形式をリクエストするには -x オプションを使用します。

```
$ pkginfo -x
.
.
```

```
.
SUNWipc         Interprocess Communications
                (sparc) 11.8.0,REV=1999.08.20.12.37
SUNWisolc       XSH4 conversion for ISO Latin character sets
                (sparc) 1.0,REV=1999.07.10.10.10
SUNWkcspf       KCMS Optional Profiles
                (sparc) 1.1.2,REV=1.5
SUNWkcspg       KCMS Programmers Environment
                (sparc) 1.1.2,REV=1.5
.
.
.
$
```

-l オプションを使用すると長形式で表示され、次の例のように、パッケージについて入手できるすべての情報が表示されます。

```
$ pkginfo -l SUNWcadap
   PKGINST:  SUNWcadap
      NAME:  Chip designers need CAD application software to
design abc chips.  Runs only on xyz hardware and is installed
in the usr partition.
  CATEGORY:  system
      ARCH:  SPARC
   VERSION:  release 1.0
   BASEDIR:  /opt
    PSTAMP:  system980706083849
  INSTDATE:  Jul 7 1999 09:58
    STATUS:  completely installed
     FILES:     13 installed pathnames
                 6 directories
                 3 executables
              3121 blocks used (approx)
$
```

pkginfo の長形式のパラメータの説明

次の表で、各パッケージで表示できるパッケージパラメータについて説明します。パラメータとその値は、パラメータに値が割り当てられている場合にのみ表示されます。

表 4–2 パッケージパラメータ

パラメータ	説明
ARCH	パッケージがサポートするアーキテクチャー。
BASEDIR	ソフトウェアパッケージが常駐するベースディレクトリ (パッケージが再配置可能な場合に表示されます)。
CATEGORY	パッケージがメンバーであるソフトウェアのカテゴリ (たとえば、system や application)。

表 4–2 パッケージパラメータ (続き)

パラメータ	説明
CLASSES	パッケージに定義されているクラスのリスト。リストの順序で、クラスのインストール順序が決まります。最初にリストされているクラスが、最初にインストールされます(メディア単位)。このパラメータは、request スクリプトによって変更される場合があります。
DESC	パッケージを説明するテキスト。
EMAIL	ユーザー照会のための電子メールアドレス。
HOTLINE	パッケージについてのホットラインヘルプを受け取る方法についての情報。
INTONLY	NULL 以外の値に設定されている場合は、パッケージを対話形式でのみインストールする必要があることを示します。
ISTATES	パッケージのインストールに対する許容可能な実行状態のリスト (S s 1 など)。
MAXINST	マシン上で同時に許可する必要のあるパッケージインスタンスの最大数。デフォルトでは、1 つのパッケージインスタンスのみが許可されます。
NAME	パッケージ名。通常は、パッケージの省略名を説明するテキスト。
ORDER	媒体に格納する順序を定義するクラスのリスト。パッケージを作成する際に pkgmk コマンドで使用されます。このパラメータで定義されていないクラスは、標準の順序付け手順を使用して媒体に格納されます。
PKGINST	インストールされているパッケージの省略名。
PSTAMP	パッケージの製造スタンプ。
RSTATES	パッケージの削除に対する許容可能な実行状態のリスト (Ss 1 など)。
ULIMIT	このパラメータが設定されると、ulimit コマンドに引数として渡されます。これにより、インストールの間にファイルの最大サイズが設定されます。これは、手続きスクリプトによって作成されるファイルに対してのみ適用されます。
VENDOR	ソフトウェアパッケージの提供ベンダーの名前。
VERSION	パッケージのバージョン。
VSTOCK	ベンダー提供のストック番号。

pkginfo コマンドの詳細は、pkginfo(1) のマニュアルページを参照してください。

▼ pkginfo コマンドで情報を取得する方法

1. パッケージをインストールします。
   必要に応じて、91 ページの「スタンドアロンシステムまたはサーバーにパッケージをインストールする方法」を参照してください。

2. パッケージについての追加情報を表示します。

   ```
   # pkginfo [-x | -l] [pkg-abbrev]
   ```

`-x`	パッケージの情報が抽出形式で表示されます。
`-l`	パッケージの情報が長形式で表示されます。
pkg-abbrev	特定のパッケージの名前です。省略すると、`pkginfo` コマンドでは、インストールされているすべてのパッケージに関する情報がデフォルトの形式で表示されます。

参考 次のステップ

以上の手順が終了したら、次のタスク、99 ページの「パッケージを削除する方法」に進みます。

パッケージの削除

パッケージを削除するときは、`rm` コマンドを使用したくなるかもしれませんが、`pkgrm` コマンドだとソフトウェア製品データベースの情報が更新されるため、`pkgrm` コマンドを使用して削除することが重要です。たとえば、`rm` コマンドを使用してバイナリ実行可能ファイルを削除することは、`pkgrm` を使用してそのバイナリ実行可能ファイルを含むソフトウェアパッケージを削除することと同じではありません。`rm` コマンドを使用してパッケージのファイルを削除すると、ソフトウェア製品用データベースが破壊されます。1 つのファイルだけを削除する場合は、`removef` コマンドを使用してください。これはソフトウェア製品データベースを正しく更新します。

▼ パッケージを削除する方法

1. スーパーユーザーとしてシステムにログインします。

2. インストール済みのパッケージを削除します。

   ```
   # pkgrm pkg-abbrev ...
   ```

pkg-abbrev	1 つ以上のパッケージの名前を空白で区切って指定します。省略すると、pkgrm では使用可能なすべてのパッケージが削除されます。

3 **pkginfo** コマンドを使用して、パッケージが正常に削除されたことを確認します。

```
$ pkginfo | egrep pkg-abbrev
```

pkg-abbrev がインストールされている場合に、pkginfo コマンドを使用すると、そのインストール情報が返されます。インストールされていない場合は、pkginfo からシステムプロンプトが返されます。

配布媒体へのパッケージの転送

pkgtrans コマンドでは、パッケージの移動とパッケージ形式の変換が実行されます。pkgtrans コマンドを使用すると、インストール可能なパッケージに対して次の変換を実行できます。

- ファイルシステム形式からデータストリーム形式
- データストリーム形式からファイルシステム形式
- あるファイルシステム形式から別のファイルシステム形式

▼ 配布媒体にパッケージを転送する方法

1 まだ行っていない場合は、パッケージを構築し、ディレクトリ形式のパッケージを作成します。

詳細については、47 ページの「パッケージの構築方法」を参照してください。

2 パッケージをインストールして、正常にインストールされることを確認します。

必要に応じて、91 ページの「スタンドアロンシステムまたはサーバーにパッケージをインストールする方法」を参照してください。

3 パッケージの整合性を確認します。

必要に応じて、93 ページの「パッケージの整合性を確認する方法」、99 ページの「pkginfo コマンドで情報を取得する方法」、および94 ページの「pkgparam コマンドで情報を取得する方法」を参照してください。

4 インストール済みパッケージをシステムから削除します。

必要な場合は、99 ページの「パッケージを削除する方法」を参照してください。

5 パッケージを (パッケージ形式で) 配布媒体に転送します。

基本的な変換を実行するには、次のコマンドを実行します。

```
$ pkgtrans device1 device2 [pkg-abbrev...]
```

device1	パッケージが現在常駐しているデバイスの名前です。
device2	変換後のパッケージを書き込むデバイスの名前です。
[*pkg-abbrev*]	1 つ以上のパッケージの省略名です。

パッケージ名を指定しないと、*device1* に常駐しているすべてのパッケージが変換されて、*device2* に書き込まれます。

注 – パッケージの複数のインスタンスが *device1* に常駐している場合は、パッケージのインスタンス識別子を使用する必要があります。パッケージ識別子については、27 ページの「パッケージインスタンスの定義」を参照してください。変換中のパッケージのインスタンスが *device2* にすでに存在している場合、`pkgtrans` コマンドでは変換は実行されません。`-o` オプションを使用して、`pkgtrans` コマンドが出力先デバイスの既存のインスタンスを上書きするように指定し、`-n` オプションを使用して、インスタンスがすでに存在する場合にこのコマンドが新しいインスタンスを作成するように指定できます。*device2* がデータストリーム形式をサポートする場合はこのチェックが適用されないことに注意してください。

参考

次のステップ

この段階で、パッケージの設計、構築、確認、および転送に必要な手順が完了しています。ケーススタディーを見る場合は、第 5 章「パッケージ作成のケーススタディー」を参照してください。高度なパッケージ設計のアイデアを見る場合は、第 6 章「パッケージの作成のための高度な手法」を参照してください。

◆ ◆ ◆ 第 5 章

パッケージ作成のケーススタディー

この章では、パッケージ作成のさまざまなシナリオを、ケーススタディーとして示します。この章のケーススタディーでは、オブジェクトを条件に応じてインストールしたり、作成するファイル数を実行時に決定したり、パッケージのインストール時および削除時に既存のデータファイルを変更する例を取り上げます。

各ケーススタディーでは、最初に説明を示し、次にパッケージ作成に使用するテクニックのリストと、それらのテクニックを使用する際のアプローチにについて説明したあと、ケーススタディーに関連するサンプルのファイルとスクリプトを示します。

この章で説明するケーススタディーは次のとおりです。

管理者による入力の要求

このケーススタディーで使用するパッケージには、3 種類のオブジェクトがあります。インストールするオブジェクトの種類と、インストール先マシン内でオブジェクトを配置する場所は、管理者が選択できます。

手法

このケーススタディーでは、次の手法の実例を示します。

- パラメータ型パス名 (オブジェクトのパス名に変数を含めたもの) を使用し、複数のベースディレクトリを確立する

  パラメータ型パス名については、35 ページの「パラメータ型パス名」を参照してください。
- request スクリプトを使用して、管理者による入力を要求する

  request スクリプトについては、64 ページの「request スクリプトの書き込み」を参照してください。
- インストールパラメータの条件値を設定する。

アプローチ

このケーススタディーで選択的インストールを設定するには、次のタスクを完了する必要があります。

- インストール可能な各タイプのオブジェクトについてクラスを定義します。

  このケーススタディーの場合、オブジェクトのタイプには、パッケージ実行可能ファイル、マニュアルページ、および emacs 実行可能ファイルの 3 種類があります。各タイプには、それぞれ bin、man、および emacs というクラスが定義されています。prototype ファイルでは、すべてのオブジェクトファイルは、これら 3 つのクラスのいずれかに属しています。
- pkginfo ファイル内の CLASSES パラメータを null に初期化します。

  通常、クラスを定義する場合は、そのクラスを pkginfo ファイルの CLASSES パラメータにリストします。リストしない場合、そのクラスのオブジェクトはインストールされません。このケーススタディーでは、このパラメータは初期値として null に設定されているため、オブジェクトはインストールされません。CLASSES パラメータは、管理者の選択に基づいて、request スクリプトによって変更されます。そうすることで、CLASSES パラメータには、管理者がインストールするように選択したオブジェクトタイプだけが設定されます。

  注 – 通常は、各パラメータはデフォルト値に設定することをお勧めします。3 つすべてのオブジェクトタイプに共通のコンポーネントがこのパッケージに含まれている場合は、それらのコンポーネントを none クラスに割り当て、CLASSES パラメータを none に設定します。

- prototype ファイルにパラメータ型パス名を挿入します。

これらの環境変数は、request スクリプトによって、管理者が指定した値に設定されます。次に、pkgadd コマンドによって、インストール時にこれらの環境変数が解釈処理され、パッケージのインストール先が決定されます。

この例で使用される 3 つの環境変数は、pkginfo ファイルでそれぞれのデフォルト値に設定されます。これらの環境変数の用途は次のとおりです。

- $NCMPBIN は、オブジェクト実行可能ファイルの場所を定義します。
- $NCMPMAN は、マニュアルページの場所を定義します。
- $EMACS は、emacs 実行可能ファイルの場所を定義します。

サンプルの prototype ファイルに、変数を使用してオブジェクトパス名を定義する方法を示します。

- 管理者に、パッケージのどの部分をインストールし、どこに配置するかを質問する request スクリプトを作成します。

  このパッケージ用の request スクリプトは、管理者に対して次の 2 つの質問をします。

  - パッケージのこの部分をインストールするかどうか

    管理者が「y (はい)」 と答えた場合には、適切なクラス名が CLASSES パラメータに追加されます。たとえば、管理者がこのパッケージに関連するマニュアルページをインストールするように選択した場合は、クラス man が CLASSES パラメータに追加されます。

  - インストールする場合、パッケージのこの部分をどこに配置するか

    この質問に対する応答として指定した場所が、該当する環境変数に設定されます。マニュアルページの例では、変数 $NCMPMAN が、応答として指定した値に設定されています。

  これら 2 つの質問は、3 つのオブジェクトタイプのそれぞれについて繰り返されます。

  request スクリプトの実行が終了すると、これらのパラメータはインストール環境で pkgadd コマンドやその他のパッケージ作成スクリプトを実行する際に使用できるようになります。request スクリプトは、これらの定義を呼び出しユーティリティから指定されたファイルに書き込むことで、パラメータを使用可能にします。このケーススタディーでは、これ以外のスクリプトは使用しません。

  このケーススタディーで使用する request スクリプトでは、質問はデータ検証ツール ckyorn および ckpath によって生成されています。これらのツールの詳細は、ckyorn(1)および ckpath(1) を参照してください。

ケーススタディーのファイル

pkginfo ファイル

```
PKG=ncmp
NAME=NCMP Utilities
CATEGORY=application, tools
BASEDIR=/
ARCH=SPARC
VERSION=RELEASE 1.0, Issue 1.0
CLASSES=""
NCMPBIN=/bin
NCMPMAN=/usr/man
EMACS=/usr/emacs
```

prototype ファイル

```
i pkginfo
i request
x bin $NCMPBIN 0755 root other
f bin $NCMPBIN/dired=/usr/ncmp/bin/dired 0755 root other
f bin $NCMPBIN/less=/usr/ncmp/bin/less 0755 root other
f bin $NCMPBIN/ttype=/usr/ncmp/bin/ttype 0755 root other
f emacs $NCMPBIN/emacs=/usr/ncmp/bin/emacs 0755 root other
x emacs $EMACS 0755 root other
f emacs $EMACS/ansii=/usr/ncmp/lib/emacs/macros/ansii 0644 root other
f emacs $EMACS/box=/usr/ncmp/lib/emacs/macros/box 0644 root other
f emacs $EMACS/crypt=/usr/ncmp/lib/emacs/macros/crypt 0644 root other
f emacs $EMACS/draw=/usr/ncmp/lib/emacs/macros/draw 0644 root other
f emacs $EMACS/mail=/usr/ncmp/lib/emacs/macros/mail 0644 root other
f emacs $NCMPMAN/man1/emacs.1=/usr/ncmp/man/man1/emacs.1 0644 root other
d man $NCMPMAN 0755 root other
d man $NCMPMAN/man1 0755 root other
f man $NCMPMAN/man1/dired.1=/usr/ncmp/man/man1/dired.1 0644 root other
f man $NCMPMAN/man1/ttype.1=/usr/ncmp/man/man1/ttype.1 0644 root other
f man $NCMPMAN/man1/less.1=/usr/ncmp/man/man1/less.1 0644 inixmr other
```

request スクリプト

```
trap 'exit 3' 15
# determine if and where general executables should be placed
ans=`ckyorn -d y \
-p "Should executables included in this package be installed"
` || exit $?
if [ "$ans" = y ]
then
   CLASSES="$CLASSES bin"
   NCMPBIN=`ckpath -d /usr/ncmp/bin -aoy \
   -p "Where should executables be installed"
   ` || exit $?
fi
# determine if emacs editor should be installed, and if it should
# where should the associated macros be placed
ans=`ckyorn -d y \
-p "Should emacs editor included in this package be installed"
```

```
‘ || exit $?
if [ "$ans" = y ]
then
   CLASSES="$CLASSES emacs"
   EMACS=‘ckpath -d /usr/ncmp/lib/emacs -aoy \
   -p "Where should emacs macros be installed"
   ‘ || exit $?
fi
```

request スクリプトは、ファイルシステム上にファイルを残さずに終了できます。Oracle Solaris の 2.5 よりも前のバージョンおよび互換バージョンでのインストールでは、checkinstall スクリプトが使用されていない可能性があるので、インストールが確実に成功するように、request スクリプトを使用してファイルシステムの必要なテストを行うことをお勧めします。request スクリプトがコード 1 で終了すると、インストールはクリーンに終了します。

これらのファイル例では、パラメータ型パスを使用して複数のベースディレクトリを確立する方法が示されています。しかし、pkgadd コマンドによって管理および検証される BASEDIR パラメータを使用する必要が生じる場合もあります。複数のベースディレクトリを使用する際には、同じプラットフォームに複数のバージョンやアーキテクチャーをインストールする場合に備えて、特別な注意を払う必要があります。

インストール時のファイル作成と削除時のファイル保存

このケーススタディーでは、インストール時にデータベースファイルを作成し、パッケージの削除時にデータベースのコピーを保存します。

手法

このケーススタディーでは、次の手法の実例を示します。

- クラスおよびクラスアクションスクリプトを使用して、オブジェクトのさまざまなセットに対して特別なアクションを実行する

  詳細については、71 ページの「クラスアクションスクリプトの書き込み」を参照してください。

- space ファイルを使用して、このパッケージを正しくインストールするには追加のスペースが必要であることを pkgadd コマンドに通知する

  space ファイルについては、57 ページの「ターゲットシステムでの追加領域の予約」を参照してください。

- installf コマンドを使用して、prototype および pkgmap ファイルで定義されていないファイルをインストールする。

アプローチ

このケーススタディーに従って、インストール時にデータベースファイルを作成し、削除時にコピーを保存するには、次のタスクを完了する必要があります。

- 3 つのクラスを定義します。

  このケーススタディーで使用するパッケージには、CLASSES パラメータで次の 3 つのクラスを定義しておく必要があります。

  - 標準クラスである none。これには、サブディレクトリ bin にある一連のプロセスが含まれています。
  - admin クラス。これには、実行可能ファイル config と、データファイルを格納したディレクトリが含まれています。
  - cfgdata クラス。これには、ディレクトリが含まれています。
- パッケージを、全体として再配置可能にします。

  prototype ファイルに、スラッシュまたは環境変数で始まるパス名が存在しないことに注目してください。このことは、これらが全体として再配置可能であることを示しています。
- データベースファイルに必要なスペースのサイズを計算し、パッケージとともに配信する space ファイルを作成します。このファイルは pkgadd コマンドに対して、パッケージが追加スペースを必要とすることを通知し、そのサイズを指定します。
- admin クラス用のクラスアクションスクリプト (i.admin) を作成します。

  サンプルのスクリプトでは、admin クラスに属するデータファイルを使用して、データベースを初期化しています。このタスクを実行するために、スクリプトは次の処理を実行します。

  - ソースデータファイルを、その適切なターゲットにコピーします。
  - config.data という名前の空ファイルを作成し、cfgdata のクラスに割り当てます。
  - bin/config コマンド (パッケージとともに提供され、すでにインストールされている) を実行し、admin クラスに属するデータファイルを使用して、データベースファイル config.data を生成します。
  - installf -f コマンドを実行し、config.data のインストールをファイナライズします。

  削除時に、admin クラスに対して特別なアクションは必要ないため、削除クラスアクションスクリプトは作成されません。つまり、admin クラスのすべてのファイルとディレクトリは、システムから削除されます。
- cfgdata クラス用の削除クラスアクションスクリプト (r.cfgdata) を作成します。

この削除スクリプトは、データベースファイルが削除される前に、そのコピーを作成します。インストール時に、このクラスに対して特別なアクションは必要ないため、インストールクラスアクションスクリプトは必要ありません。

削除スクリプトへの入力は、削除するパス名のリストです。パス名は、常にアルファベットの逆順に表示されます。この削除スクリプトは、ファイルを `$PKGSAV` というディレクトリにコピーします。すべてのパス名が処理されると、スクリプトは先頭のパス名に戻り、`cfgdata` クラスに関連付けられたすべてのディレクトリとファイルを削除します。

この削除スクリプトが実行されると、その結果として、`config.data`が `$PKGSAV` にコピーされ、次に `config.data`ファイルとデータディレクトリが削除されます。

ケーススタディーのファイル

pkginfo ファイル

```
PKG=krazy
NAME=KrAzY Applications
CATEGORY=applications
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1
CLASSES=none cfgdata admin
```

prototype ファイル

```
i pkginfo
i request
i i.admin
i r.cfgdata
d none bin 555 root sys
f none bin/process1 555 root other
f none bin/process2 555 root other
f none bin/process3 555 root other
f admin bin/config 500 root sys
d admin cfg 555 root sys
f admin cfg/datafile1 444 root sys
f admin cfg/datafile2 444 root sys
f admin cfg/datafile3 444 root sys
f admin cfg/datafile4 444 root sys
d cfgdata data 555 root sys
```

space ファイル

```
# extra space required by config data which is
# dynamically loaded onto the system
data 500 1
```

i.admin クラスアクションスクリプト

```
# PKGINST parameter provided by installation service
# BASEDIR parameter provided by installation service
while read src dest
do
   cp $src $dest || exit 2
done
# if this is the last time this script will be executed
# during the installation, do additional processing here.
if [ "$1" = ENDOFCLASS ]
then
# our config process will create a data file based on any changes
# made by installing files in this class; make sure the data file
# is in class ‘cfgdata’ so special rules can apply to it during
# package removal.
   installf -c cfgdata $PKGINST $BASEDIR/data/config.data f 444 root
   sys || exit 2
   $BASEDIR/bin/config > $BASEDIR/data/config.data || exit 2
   installf -f -c cfgdata $PKGINST || exit 2
fi
exit 0
```

ここでは、クラスアクションスクリプトで `installf` が使用される珍しい例が示されています。`space` ファイルを使用して、対象のファイルシステム上でスペースを予約しているため、この新しいファイルは、`pkgmap` ファイルに含まれていない場合でも、安全に追加されます。

r.cfgdata 削除スクリプト

```
# the product manager for this package has suggested that
# the configuration data is so valuable that it should be
# backed up to $PKGSAV before it is removed!
while read path
do
# path names appear in reverse lexical order.
   mv $path $PKGSAV || exit 2
   rm -f $path || exit 2
done
exit 0
```

パッケージの互換性と依存関係の定義

このケーススタディーのパッケージでは、パッケージの互換性と依存関係を定義したり、インストール中に著作権に関するメッセージを表示したりするために、オプションの情報ファイルを使用しています。

手法

このケーススタディーでは、次の手法の実例を示します。

- `copyright` ファイルを使用する

- compver ファイルを使用する
- depend ファイルを使用する

これらのファイルについては、53 ページの「情報ファイルの作成」を参照してください。

アプローチ

この説明での必要条件を満たすには、次の作業を行う必要があります。

- copyright ファイルを作成します。

  copyright ファイルには、著作権に関するメッセージを記述した ASCII テキストが含まれています。サンプルファイルに示したメッセージは、パッケージのインストール中に画面に表示されます。

- compver ファイルを作成します。

  次の図で示す pkginfo ファイルでは、このパッケージバージョンをバージョン 3.0 として定義しています。compver ファイルでは、バージョン 3.0 がバージョン 2.3、2.2、2.1、2.1.1、2.1.3、および 1.7 と互換性があると定義しています。

- depend ファイルを作成します。

  depend ファイル内に記載されているファイルは、パッケージをインストールする際に、あらかじめシステムにインストールされている必要があります。サンプルのファイルには、インストール時にあらかじめシステムにインストールされている必要のある 11 のパッケージが記載されています。

ケーススタディーのファイル

pkginfo ファイル

```
PKG=case3
NAME=Case Study #3
CATEGORY=application
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 3.0
CLASSES=none
```

copyright ファイル

```
Copyright (c) 1999 company_name
All Rights Reserved.
THIS PACKAGE CONTAINS UNPUBLISHED PROPRIETARY SOURCE CODE OF
company_name.
The copyright notice above does not evidence any
actual or intended publication of such source code
```

compver ファイル

```
Version 3.0
Version 2.3
Version 2.2
Version 2.1
Version 2.1.1
Version 2.1.3
Version 1.7
```

depend ファイル

```
P acu Advanced C Utilities
Issue 4 Version 1
P cc C Programming Language
Issue 4 Version 1
P dfm Directory and File Management Utilities
P ed Editing Utilities
P esg Extended Software Generation Utilities
Issue 4 Version 1
P graph Graphics Utilities
P rfs Remote File Sharing Utilities
Issue 1 Version 1
P rx Remote Execution Utilities
P sgs Software Generation Utilities
Issue 4 Version 1
P shell Shell Programming Utilities
P sys System Header Files
Release 3.1
```

標準クラスとクラスアクションスクリプトを使用したファイルの変更

このケーススタディーでは、標準クラスとクラスアクションスクリプトを使用して、パッケージのインストール中に既存のファイルを変更します。ここでは、3 種類の変更方法のうちの 1 つを使用します。ほかの 2 つの方法については、115 ページの「sed クラスと postinstall スクリプトを使用したファイルの変更」および117 ページの「build クラスを使用したファイルの変更」を参照してください。変更する対象のファイルは、/etc/inittab です。

手法

このケーススタディーでは、インストールおよび削除クラスアクションスクリプトの使用方法を説明します。詳細については、71 ページの「クラスアクションスクリプトの書き込み」を参照してください。

アプローチ

クラスおよびクラスアクションスクリプトを使用して、インストール中に /etc/inittab を変更するには、次のタスクを完了する必要があります。

- クラスを作成します。

  inittab という名前のクラスを作成します。このクラスに対して、インストールおよび削除クラスアクションスクリプトを作成する必要があります。pkginfo ファイルの CLASSES パラメータで inittab クラスを定義します。

- inittab ファイルを作成します。

  このファイルには、/etc/inittab に追加するエントリの情報を格納します。prototype ファイルの図で、inittab は inittab クラスのメンバーで、ファイルタイプは編集可能を表す e となっていることに注目してください。

- インストールクラスアクションスクリプト (i.inittab) を作成します。

  クラスアクションスクリプトは、実行されるたびに同じ結果を生じる必要があります。このクラスアクションスクリプトは、次の手順を実行します。

  - このエントリが以前に追加されているかどうかをチェックします。
  - 追加されている場合は、このエントリの以前のバージョンをすべて削除します。
  - inittab ファイルを編集し、このエントリの出自がわかるようにコメント行を追加します。
  - 一時ファイルを /etc/inittab に戻します。
  - ENDOFCLASS インジケータを受け取ったときに init q コマンドを実行します。

  init q コマンドは、このインストールスクリプトで実行できます。このアプローチでは、1 行だけの postinstall スクリプトは必要ありません。

- 削除クラスアクションスクリプト (r.inittab) を作成します。

  削除スクリプトは、インストールスクリプトと非常によく似ています。インストールスクリプトが追加した情報を削除し、init q コマンドを実行します。

このケーススタディーは、次のケーススタディーよりも複雑です。115 ページの「sed クラスと postinstall スクリプトを使用したファイルの変更」を参照してください。2 つではなく 3 つのファイルを作成する必要があり、配信された /etc/inittab ファイルは実際には、挿入するエントリの断片を含んだ可変部分に過ぎません。pkgadd コマンドが i.inittab ファイルに渡すファイルを必要としなければ、これは i.inittab ファイルに組み込まれていたかも知れません。また、削除の手順も別個のファイル (r.inittab) に置く必要があります。この方法はうまく行きますが、複数のファイルをインストールする必要のある非常に複雑なケースに最も向いています。119 ページの「インストール時の crontab ファイルの変更」を参照してください。

115 ページの「sed クラスと postinstall スクリプトを使用したファイルの変更」で使用されている sed プログラムは、inittab エントリの最後にあるコメントがパッケージのインスタンスに基づいているため、複数のパッケージのインスタンスをサポートします。117 ページの「build クラスを使用したファイルの変更」のケーススタディーでは、インストール中に /etc/inittab を編集するための、より効率的なアプローチが紹介されています。

ケーススタディーのファイル

pkginfo ファイル

```
PKG=case5
NAME=Case Study #5
CATEGORY=applications
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1d05
CLASSES=inittab
```

prototype ファイル

```
i pkginfo
i i.inittab
i r.inittab
e inittab /etc/inittab ? ? ?
```

i.inittab インストールクラスアクションスクリプト

```
# PKGINST parameter provided by installation service
while read src dest
do
# remove all entries from the table that
# associated with this PKGINST
sed -e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" $dest >
/tmp/$$itab ||
exit 2
sed -e "s/$/#$PKGINST" $src >> /tmp/$$itab ||
exit 2
mv /tmp/$$itab $dest ||
exit 2
done
if [ "$1" = ENDOFCLASS ]
then
/sbin/init q ||
exit 2
fi
exit 0
```

r.inittab 削除クラスアクションスクリプト

```
# PKGINST parameter provided by installation service
while read src dest
do
```

```
# remove all entries from the table that
# are associated with this PKGINST
sed -e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" $dest >
/tmp/$$itab ||
exit 2
mv /tmp/$$itab $dest ||
exit 2
done
/sbin/init q ||
exit 2
exit 0
```

inittab ファイル

```
rb:023456:wait:/usr/robot/bin/setup
```

sed クラスと postinstall スクリプトを使用したファイルの変更

このケーススタディーでは、パッケージのインストール中に、インストール先マシンに存在しているファイルを変更します。ここでは、3 種類の変更方法のうちの 1 つを使用します。ほかの 2 つの方法については、112 ページの「標準クラスとクラスアクションスクリプトを使用したファイルの変更」および117 ページの「build クラスを使用したファイルの変更」を参照してください。変更する対象のファイルは、/etc/inittab です。

手法

このケーススタディーでは、次の手法の実例を示します。

- sed クラスを使用する

  sed クラスの詳細については、76 ページの「sed クラススクリプト」を参照してください。

- postinstall スクリプトを使用する

  このスクリプトの詳細については、69 ページの「手続きスクリプトの書き込み」を参照してください。

アプローチ

インストール時に /etc/inittab を変更するには、sed クラスを使用して、次のタスクを完了する必要があります。

- sed クラススクリプトを prototype ファイルに追加します。

スクリプトの名前は、編集対象ファイルの名前である必要があります。このケースでは、編集対象のファイルが /etc/inittab であるため、sed スクリプトの名前も /etc/inittab となります。sed スクリプトのモード、所有者、およびグループには必要条件はありません (サンプルの prototype ではクエスチョンマークで表されています)。sed スクリプトのファイルタイプは e (編集可能であることを表す) である必要があります。

- CLASSES パラメータに sed クラスを含めます。

  例として示したファイルでは、インストールされるクラスは sed だけです。ただし、必要に応じて、任意の数のクラスを指定することができます。

- sed クラスアクションスクリプトを作成します。

  /etc/inittab は動的ファイルで、パッケージをインストールする時点での内容を知る方法がないため、必要な動作を指定した /etc/inittab のコピーをパッケージで提供することはできません。しかし、sed スクリプトを使用することで、パッケージのインストール時に /etc/inittab ファイルを変更できます。

- postinstall スクリプトを作成します。

  init q コマンドを実行して、/etc/inittab が変更されたことをシステムに通知する必要があります。この例で、そのアクションを実行できるのは、postinstall スクリプトだけです。サンプルの postinstall スクリプトは、init q コマンドの実行だけを行なっています。

この方法でインストール中に /etc/inittab を編集する場合、欠点が 1 つあります。それは、init q コマンドを実行するだけのために、完全なスクリプト (postinstall スクリプト) を提供する必要があるという点です。

ケーススタディーのファイル

pkginfo ファイル

```
PKG=case4
NAME=Case Study #4
CATEGORY=applications
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1d05
CLASSES=sed
```

prototype ファイル

```
i pkginfo
i postinstall
e sed /etc/inittab ? ? ?
```

sed クラスアクションスクリプト (/etc/inittab)

```
!remove
# remove all entries from the table that are associated
# with this package, though not necessarily just
# with this package instance
/^[^:]*:[^:]*:[^:]*:[^#]*#ROBOT$/d
!install
# remove any previous entry added to the table
# for this particular change
/^[^:]*:[^:]*:[^:]*:[^#]*#ROBOT$/d
# add the needed entry at the end of the table;
# sed(1) does not properly interpret the '$a'
# construct if you previously deleted the last
# line, so the command
# $a\
# rb:023456:wait:/usr/robot/bin/setup #ROBOT
# will not work here if the file already contained
# the modification. Instead, you will settle for
# inserting the entry before the last line!
$i\
rb:023456:wait:/usr/robot/bin/setup #ROBOT
```

postinstall スクリプト

```
# make init re-read inittab
/sbin/init q ||
exit 2
exit 0
```

build クラスを使用したファイルの変更

このケーススタディーでは、パッケージのインストール中に、インストール先マシンに存在しているファイルを変更します。ここでは、3 種類の変更方法のうちの 1 つを使用します。ほかの 2 つの方法については、112 ページの「標準クラスとクラスアクションスクリプトを使用したファイルの変更」および115 ページの「sed クラスと postinstall スクリプトを使用したファイルの変更」を参照してください。変更する対象のファイルは、/etc/inittab です。

手法

このケーススタディーでは、build クラスの使用方法を説明します。build クラスの詳細については、77 ページの「build クラススクリプト」を参照してください。

アプローチ

このアプローチでは、/etc/inittab を変更するために build クラスを使用します。build クラススクリプトはシェルスクリプトとして実行され、実行されている

ファイルの新しいバージョンを出力します。つまり、このパッケージに付属するデータファイル /etc/inittab が実行され、その実行の出力が /etc/inittab になります。

build クラススクリプトは、パッケージのインストールおよび削除の際に実行されます。ファイルがインストール時に実行される場合には、引数 installがファイルに渡されます。サンプルの build クラススクリプトでは、この引数をテストすることで、インストールのアクションが定義されています。

buildクラスを使用して /etc/inittab を編集するには、次のタスクを完了する必要があります。

- prototypeファイルで構築ファイルを定義します。

  prototype ファイル内の構築ファイルのエントリでは、構築ファイルを build クラスとして指定し、そのファイルタイプを e として定義する必要があります。pkginfo ファイルの CLASSES パラメータが build として定義されていることを確認してください。
- build クラススクリプトを作成します。

  サンプルの build クラススクリプトは、次の手順を実行します。

  - /etc/inittab ファイルを編集し、このパッケージに関する既存の変更をすべて削除する。ファイル名 /etc/inittab は、sed コマンドにハードコードされています。
  - パッケージがインストールされる場合は、/etc/inittab の最後に新しい行を追加する。この新しいエントリには、そのエントリの出自を示すコメントタグが含まれています。
  - init q コマンドを実行する。

この解決方法は、112 ページの「標準クラスとクラスアクションスクリプトを使用したファイルの変更」および115 ページの「sed クラスと postinstall スクリプトを使用したファイルの変更」のケーススタディーで説明した欠点に対応しています。短いファイルがひとつ (pkginfo および prototype ファイル以外に) 必要になるだけです。PKGINST パラメータを使用することにより、このファイルはパッケージの複数のインスタンスを処理することができ、init q コマンドを build クラススクリプトから実行することにより、postinstall スクリプトが不要になります。

ケーススタディーのファイル

pkginfo ファイル

```
PKG=case6
NAME=Case Study #6
CATEGORY=applications
BASEDIR=/opt
```

```
ARCH=SPARC
VERSION=Version 1d05
CLASSES=build
```

prototype ファイル

```
i pkginfo
e build /etc/inittab ? ? ?
```

構築ファイル

```
# PKGINST parameter provided by installation service
# remove all entries from the existing table that
# are associated with this PKGINST
sed -e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" /etc/inittab ||
exit 2
if [ "$1" = install ]
then
# add the following entry to the table
echo "rb:023456:wait:/usr/robot/bin/setup #$PKGINST" ||
exit 2
fi
/sbin/init q ||
exit 2
exit 0
```

インストール時の crontab ファイルの変更

このケーススタディーでは、パッケージのインストール中に crontab ファイルを変更します。

手法

このケーススタディーでは、次の手法の実例を示します。

- クラスおよびクラスアクションスクリプトを使用する

  詳細については、71 ページの「クラスアクションスクリプトの書き込み」を参照してください。
- クラスアクションスクリプト内で crontab コマンドを使用する。

アプローチ

インストール中に複数のファイルを編集する最も効率的な方法は、クラスを定義し、クラスアクションスクリプトを作成することです。build クラスによるアプローチを使用した場合は、編集する crontab ファイルごとに 1 つの build クラススクリプトを作成する必要があります。cron クラスを定義する方法は、より汎用性のあるアプローチとなります。このアプローチで crontab ファイルを編集するには、次の作業を行う必要があります。

- prototype ファイルで、編集対象の crontab ファイルを定義します。

  prototype ファイルで、編集対象の crontab ファイルごとにエントリを作成します。クラスを cron として定義し、ファイルタイプをファイルごとに e として定義します。編集対象とするファイルの実際の名前を使用してください。
- パッケージ用の crontab ファイルを作成します。

  これらのファイルには、同じ名前の既存の crontab ファイルに追加する情報を格納します。
- cron クラス用のインストールクラスアクションスクリプトを作成します。

  サンプルの i.cron スクリプトは、次の手順を実行します。
  - ユーザー ID (UID) を決定する。

    i.cron スクリプトは、変数 *user* を、処理対象の cron クラススクリプトのベース名に設定します。この名前が UID です。たとえば、/var/spool/cron/crontabs/root のベース名は root で、これが UID でもあります。
  - UID と -l オプションを使用して、crontab を実行します。

    -l オプションは、定義されたユーザーの crontab ファイルの内容を標準出力に送信するように、crontab に指示します。
  - crontab コマンドの出力を、このインストール手法で以前に追加されたすべてのエントリを削除する sed スクリプトにパイプで連結します。
  - 編集された出力を一時ファイルに置きます。
  - ルート UID のデータファイル (パッケージとともに提供されたもの) を一時ファイルに追加し、これらのエントリの出自を示すタグを追加します。
  - 同じ UID の crontab を実行し、一時ファイルを入力として指定します。
- cron クラス用の削除クラスアクションスクリプトを作成します。

  r.cron スクリプトは、crontab ファイルに情報を追加する手順がない点を除き、インストールスクリプトと同じです。

  これらの手順は、cron クラスのすべてのファイルに対して実行されます。

ケーススタディーのファイル

次に示す i.cron および r.cron スクリプトは、スーパーユーザーによって実行されます。ほかのユーザーの crontab ファイルをスーパーユーザーとして編集すると、予期しない結果が生じる可能性があります。必要な場合は、各スクリプトで次に示すエントリを変更してください。

```
crontab $user < /tmp/$$crontab ||
```

から

```
su $user -c "crontab /tmp/$$crontab" ||
```

pkginfo コマンド

```
PKG=case7
NAME=Case Study #7
CATEGORY=application
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1.0
CLASSES=cron
```

prototype ファイル

```
i pkginfo
i i.cron
i r.cron
e cron /var/spool/cron/crontabs/root ? ? ?
e cron /var/spool/cron/crontabs/sys ? ? ?
```

i.cron インストールクラスアクションスクリプト

```
# PKGINST parameter provided by installation service
while read src dest
do
user=‘basename $dest‘ ||
exit 2
(crontab -l $user |
sed -e "/#$PKGINST$/d" > /tmp/$$crontab) ||
exit 2
sed -e "s/$/#$PKGINST/" $src >> /tmp/$$crontab ||
exit 2
crontab $user < /tmp/$$crontab ||
exit 2
rm -f /tmp/$$crontab
done
exit 0
```

r.cron 削除クラスアクションスクリプト

```
# PKGINST parameter provided by installation service
while read path
do
user=‘basename $path‘ ||
exit 2
(crontab -l $user |
sed -e "/#$PKGINST$/d" > /tmp/$$crontab) ||
exit 2
crontab $user < /tmp/$$crontab ||
exit 2
rm -f /tmp/$$crontab
done
exit
```

crontab ファイル 1

```
41,1,21 * * * * /usr/lib/uucp/uudemon.hour > /dev/null
45 23 * * * ulimit 5000; /usr/bin/su uucp -c
"/usr/lib/uucp/uudemon.cleanup" >
/dev/null 2>&1
11,31,51 * * * * /usr/lib/uucp/uudemon.poll > /dev/null
```

crontab ファイル 2

```
0 * * * 0-6 /usr/lib/sa/sa1
20,40 8-17 * * 1-5 /usr/lib/sa/sa1
5 18 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -A
```

注 – 複数のファイルを編集することにより、ファイルの合計サイズが 10K を超えて増加する場合は、pkgadd コマンドがこの増加に対応できるように、space ファイルを作成してください。space ファイルについては、57 ページの「ターゲットシステムでの追加領域の予約」を参照してください。

手続きスクリプトによるドライバのインストールと削除

このパッケージでは、ドライバをインストールします。

手法

このケーススタディーでは、次の手法の実例を示します。

- postinstall スクリプトを使用して、ドライバをインストールおよびロードする
- preremove スクリプトを使用して、ドライバをアンロードする

これらのスクリプトの詳細については、69 ページの「手続きスクリプトの書き込み」を参照してください。

アプローチ

- request スクリプトを作成します。

  request スクリプトは、管理者に対してドライバオブジェクトのインストール先を質問し、その答えを $KERNDIR パラメータに割り当てることで、ドライバオブジェクトのインストール先を決定します。

  このスクリプトは、CLASSES および KERNDIR の 2 つのパラメータを、インストール環境と postinstall スクリプトで使用できるようにするルーチンを実行して終了します。
- postinstall スクリプトを作成します。

postinstall スクリプトは、実際にはドライバのインストールを実行します。ドライバのインストールは、buffer と buffer.conf という 2 つのファイルがインストールされたあとに実行されます。この例で示した postinstall ファイルは、次のアクションを実行します。

- add_drv コマンドを使用して、ドライバをシステムにロードします。
- installf コマンドを使用して、デバイスのリンクを作成します。
- installf -f コマンドを使用して、インストールをファイナライズします。
- preremove スクリプトを作成します。

  preremove スクリプトは rem_drv コマンドを使用して、システムからドライバをアンロードしたあと、リンク /dev/buffer0 を削除します。

ケーススタディーのファイル

pkginfo ファイル

```
PKG=bufdev
NAME=Buffer Device
CATEGORY=system
BASEDIR=/
ARCH=INTEL
VERSION=Software Issue #19
CLASSES=none
```

prototype ファイル

インストール時にドライバをインストールするには、ドライバのオブジェクトおよび構成ファイルを prototypeファイルに含める必要があります。

この例では、ドライバの実行モジュールの名前は buffer です。add_drv コマンドは、このファイルに対して実行されます。カーネルは、構成ファイル buffer.conf をドライバの構成に利用します。

```
i pkginfo
i request
i postinstall
i preremove
f none $KERNDIR/buffer 444 root root
f none $KERNDIR/buffer.conf 444 root root
```

この例の prototype ファイルを見て、次の点に注意してください。

- パッケージオブジェクトについて特別な処理は必要ないため、それらのオブジェクトは標準の none クラスに配置できます。pkginfo ファイルの CLASSES パラメータは none に設定されています。

- buffer と buffer.conf のパス名が、変数 $KERNDIR で始まっています。この変数は request スクリプトで設定され、管理者はこの変数によってドライバファイルのインストール先を決定できます。デフォルトのディレクトリは /kernel/drv です。
- postinstallスクリプト (ドライバのインストールを実行するスクリプト) のエントリが存在します。

request スクリプト

```
trap ’exit 3’ 15
# determine where driver object should be placed; location
# must be an absolute path name that is an existing directory
KERNDIR=‘ckpath -aoy -d /kernel/drv -p \
“Where do you want the driver object installed”‘ || exit $?

# make parameters available to installation service, and
# so to any other packaging scripts
cat >$1 <<!

CLASSES=’$CLASSES’
KERNDIR=’$KERNDIR’
!
exit 0
```

postinstall スクリプト

```
# KERNDIR parameter provided by ‘request’ script
err_code=1                    # an error is considered fatal
# Load the module into the system
cd $KERNDIR
add_drv -m ’* 0666 root sys’ buffer || exit $err_code
# Create a /dev entry for the character node
installf $PKGINST /dev/buffer0=/devices/eisa/buffer*:0 s
installf -f $PKGINST
```

preremove スクリプト

```
err_code=1                    # an error is considered fatal
# Unload the driver
rem_drv buffer || exit $err_code
# remove /dev file
removef $PKGINST /dev/buffer0 ; rm /dev/buffer0
removef -f $PKGINST
```

sed クラスと手続きスクリプトを使用したドライバのインストール

このケーススタディーでは、sed クラスと手続きスクリプトを使用してドライバをインストールする方法を説明します。このパッケージは絶対オブジェクトと再配置可能オブジェクトの両方から構成されるため、この方法は前のケーススタディー (122 ページの「手続きスクリプトによるドライバのインストールと削除」を参照) とは異なります。

手法

このケーススタディーでは、次の手法の実例を示します。

- 絶対オブジェクトと再配置可能オブジェクトの両方を使用して prototype ファイルを作成する

  prototype ファイルの作成の詳細については、31 ページの「prototype ファイルの作成」を参照してください。

- postinstall スクリプトを使用する

  このスクリプトの詳細については、69 ページの「手続きスクリプトの書き込み」を参照してください。

- preremove スクリプトを使用する

  このスクリプトの詳細については、69 ページの「手続きスクリプトの書き込み」を参照してください。

- copyright ファイルを使用する

  このファイルについては、56 ページの「著作権に関するメッセージの書き込み」を参照してください。

アプローチ

- 絶対パッケージオブジェクトと再配置可能パッケージオブジェクトの両方を含んだ prototype ファイルを作成します。

  詳細は、126 ページの「prototype ファイル」を参照してください。

- sed クラススクリプトを prototype ファイルに追加します。

  スクリプトの名前は、編集対象ファイルの名前である必要があります。このケースでは、編集対象のファイルが /etc/devlink.tab であるため、sed スクリプトの名前も /etc/devlink.tab となります。sed スクリプトのモード、所有者、およびグループには必要条件はありません (サンプルの prototypeではクエスチョンマークで表されています)。sed スクリプトのファイルタイプは e (編集可能であることを表す) である必要があります。

- CLASSES パラメータに sed クラスを含めます。
- sed クラスアクションスクリプト (/etc/devlink.tab) を作成します。
- postinstall スクリプトを作成します。

  postinstall スクリプトでは、add_drv コマンドを実行してデバイスドライバをシステムに追加する必要があります。
- preremove スクリプトを作成します。

  preremove スクリプトでは、パッケージを削除する前に rem_drv コマンドを実行してデバイスドライバをシステムから削除する必要があります。
- copyright ファイルを作成します。

  copyright ファイルには、著作権に関するメッセージを記述した ASCII テキストが含まれています。サンプルファイルに示したメッセージは、パッケージのインストール中に画面に表示されます。

ケーススタディーのファイル

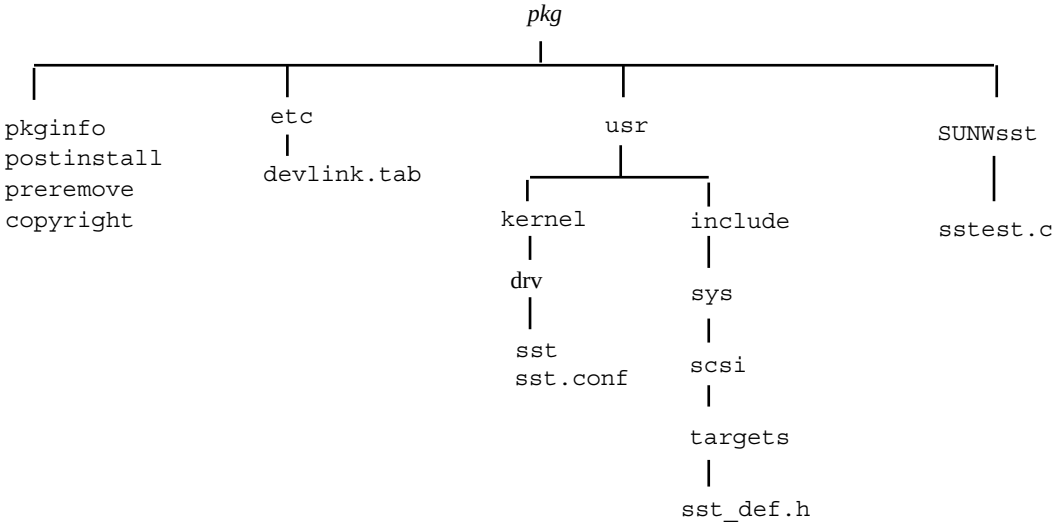
pkginfo ファイル

```
PKG=SUNWsst
NAME=Simple SCSI Target Driver
VERSION=1
CATEGORY=system
ARCH=sparc
VENDOR=Sun Microsystems
BASEDIR=/opt
CLASSES=sed
```

prototype ファイル

たとえばこのケーススタディーでは、パッケージオブジェクトについて、次の図に示す階層型レイアウトを使用しています。

図5–1 パッケージの階層型ディレクトリ構造



パッケージオブジェクトは、上に示したpkgディレクトリ内での場所と同じ場所にインストールされます。ドライバモジュール(sstおよびsst.conf)は/usr/kernel/drvにインストールされ、インクルードファイルは/usr/include/sys/scsi/targetsにインストールされます。sst、sst.conf、およびsst_def.hファイルは絶対オブジェクトです。テストプログラムsstest.cとそのディレクトリSUNWsstは再配置可能オブジェクトです。それらのインストール場所はBASEDIRパラメータによって設定されます。

パッケージの残りのコンポーネント(すべての制御ファイル)は、開発マシン上でのパッケージの最上位ディレクトリにインストールされますが、sedクラススクリプトだけは例外です。このクラススクリプトは、それが変更するファイルにならってdevlink.tabと呼ばれ、etcにインストールされます。このディレクトリには、実際のdevlink.tabファイルが格納されています。

pkgディレクトリからpkgprotoコマンドを次のように実行します。

```
find usr SUNWsst -print | pkgproto > prototype
```

このコマンドによる出力は、たとえば次のようになります。

```
d none usr 0775 pms mts
d none usr/include 0775 pms mts
d none usr/include/sys 0775 pms mts
d none usr/include/sys/scsi 0775 pms mts
d none usr/include/sys/scsi/targets 0775 pms mts
f none usr/include/sys/scsi/targets/sst_def.h 0444 pms mts
d none usr/kernel 0775 pms mts
d none usr/kernel/drv 0775 pms mts
f none usr/kernel/drv/sst 0664 pms mts
```

```
f none usr/kernel/drv/sst.conf 0444 pms mts
d none SUNWsst 0775 pms mts
f none SUNWsst/sstest.c 0664 pms mts
```

この prototype ファイルはまだ完成していません。このファイルを完成するには、次の修正を加える必要があります。

- 制御ファイルのエントリを挿入します (ファイルタイプ i)。これは、制御ファイルの書式がほかのパッケージオブジェクトとは異なるためです。
- ターゲットシステムにすでに存在しているディレクトリのエントリを削除します。
- 各エントリのアクセス権と所有者を変更します。
- 絶対パッケージオブジェクトの前にスラッシュを付加します。

最終的な prototype ファイルを次に示します。

```
i pkginfo
i postinstall
i preremove
i copyright
e sed /etc/devlink.tab ? ? ?
f none /usr/include/sys/scsi/targets/sst_def.h 0644 bin bin
f none /usr/kernel/drv/sst 0755 root sys
f none /usr/kernel/drv/sst.conf 0644 root sys
d none SUNWsst 0775 root sys
f none SUNWsst/sstest.c 0664 root sys
```

sed スクリプトのエントリにあるクエスチョンマークは、インストール先マシンの既存ファイルのアクセス権と所有者を変更してはならないことを示します。

sed クラスアクションスクリプト (/etc/devlink.tab)

ドライバの例では、sed クラススクリプトはドライバのエントリをファイル /etc/devlink.tab に追加するために使用されています。このファイルは、/dev から /devices へのシンボリックリンクを作成するために、devlinks コマンドによって使用されます。sed スクリプトを次に示します。

```
# sed class script to modify /etc/devlink.tab
!install
/name=sst;/d
$i\
type=ddi_pseudo;name=sst;minor=character    rsst\\A1

!remove
/name=sst;/d
```

pkgrm コマンドは、このスクリプトの削除に関する部分を実行しません。/etc/devlink.tab ファイルからエントリを削除するには、preremove スクリプトに sed を直接実行するための行を追加する必要があります。

postinstallインストールスクリプト

この例では、このスクリプトはadd_drvコマンドの実行だけを行います。

```
# Postinstallation script for SUNWsst
# This does not apply to a client.
if [$PKG_INSTALL_ROOT = "/" -o -z $PKG_INSTALL_ROOT]; then
   SAVEBASE=$BASEDIR
   BASEDIR=””; export BASEDIR
   /usr/sbin/add_drv sst
   STATUS=$?
   BASEDIR=$SAVEBASE; export BASEDIR
   if [ $STATUS -eq 0 ]
   then
         exit 20
   else
         exit 2
   fi
else
   echo "This cannot be installed onto a client."
   exit 2
fi
```

add_drvコマンドはBASEDIRパラメータを使用するため、スクリプトはこのコマンドを実行する前にBASEDIRの設定を解除し、実行後に復元する必要があります。

add_drvコマンドの動作の1つは、devlinksを実行することです。これは、sedクラススクリプトが/etc/devlink.tabに挿入したエントリを使用して、ドライバの/devエントリを作成します。

postinstallスクリプトの終了コードは重要です。終了コード20はpkgaddコマンドに対して、ユーザーにシステムのリブート(ドライバのインストール後に必要)を要求するように指示します。終了コード2はpkgaddコマンドに対して、インストールが部分的に失敗したことをユーザーに知らせるように指示します。

preremove削除スクリプト

このドライバの例では、このスクリプトは/dev内のリンクを削除し、ドライバに対してrem_drvコマンドを実行します。

```
# Pre removal script for the sst driver
echo “Removing /dev entries”
/usr/bin/rm -f /dev/rsst*

echo “Deinstalling driver from the kernel”
SAVEBASE=$BASEDIR
BASEDIR=””; export BASEDIR
/usr/sbin/rem_drv sst
BASEDIR=$SAVEBASE; export BASEDIR

exit
```

このスクリプトは/devのエントリを削除します。/devicesのエントリは、rem_drvコマンドによって削除されます。

copyright ファイル

これは、著作権表示のテキストを格納した単純な ASCII ファイルです。著作権表示は、パッケージのインストール開始時に、ファイルに記載されているとおりに表示されます。

```
    Copyright (c) 1999 Drivers-R-Us, Inc.
    10 Device Drive, Thebus, IO 80586

All rights reserved. This product and related documentation is
protected by copyright and distributed under licenses
restricting its use, copying, distribution and decompilation.
No part of this product or related documentation may be
reproduced in any form by any means without prior written
authorization of Drivers-R-Us and its licensors, if any.
```

◆ ◆ ◆ 第 6 章

パッケージの作成のための高度な手法

Oracle Solaris OS に実装する際に System V の全機能をパッケージ化すると、ソフトウェア製品をインストールするための強力なツールとなります。パッケージの設計者はこれらの機能を利用できます。Oracle Solaris OS の一部ではないパッケージ (アンバンドル版パッケージ) では、クラスメカニズムを使用すると、サーバーおよびクライアントのインストールをカスタマイズできます。再配置可能パッケージは、管理者の要件に応じて設計できます。複雑な製品は、パッケージの依存関係が自動的に解決される複合パッケージのセットとして配信できます。パッケージの設計者は、アップグレードとパッチをカスタマイズできます。パッチを適用したパッケージは、パッチを適用していないパッケージと同じ方法で配信できます。また、製品にバックアウトアーカイブを含めることもできます。

この章の内容は以下のとおりです。

ベースディレクトリの指定

パッケージがインストールされる場所はさまざまな方法を使用して指定することができ、インストール時に動的にインストールベースを変更できることは重要です。インストール時に動的にインストールベースを変更できる場合、管理者は複数のバージョンと複数のアーキテクチャーを容易にインストールできます。

このセクションでは、最初に一般的な方法について説明してから、異機種システムへのインストールを行うアプローチを説明します。

管理デフォルトファイル

パッケージのインストールを行う管理者は、管理ファイルを使用してパッケージのインストールを制御できます。ただし、パッケージの設計者は、管理ファイルと、設計者が意図するパッケージのインストールを管理者がどのように変更できるかを理解する必要があります。

管理ファイルによって、通常行われるチェックやプロンプトを実行するかどうかが pkgadd コマンドに通知されます。そのため、管理者はパッケージのインストールプロセスと、関連するスクリプトについて理解してから管理ファイルを使用するようにしてください。

基本的な管理デフォルトファイルは、SunOS オペレーティングシステムの /var/sadm/install/admin/default に付属しています。これは、ソフトウェア製品のインストールに関して最も基本的な管理ポリシーを確立するファイルです。このファイルは、出荷時には次のようになっています。

```
#ident "@(#)default
1.4 92/12/23 SMI"    /* SVr4.0 1.5.2.1    */
mail=
instance=unique
partial=ask
runlevel=ask
idepend=ask
rdepend=ask
space=ask
setuid=ask
conflict=ask
action=ask
basedir=default
```

管理者は、このファイルを編集して新しいデフォルト動作を確立したり、異なる管理ファイルを作成し、pkgadd コマンドに -a オプションを使用してファイルを指定したりすることができます。

管理ファイルでは 11 個のパラメータを定義できますが、すべてのパラメータを定義する必要はありません。詳細については、admin(4) を参照してください。

basedir パラメータは、パッケージをインストールする場合にベースディレクトリを取得する方法を指定します。ほとんどの管理者はこのパラメータを default のままにしますが、basedir は次のいずれかに設定できます。

- ask。常に管理者にベースディレクトリを確認します。
- 絶対パス名。
- $PKGINST 構成を含む絶対パス名。常にパッケージインスタンスから派生したベースディレクトリにインストールします。

注 – pkgadd コマンドに引数 -a none を付けて呼び出した場合、常に管理者にベースディレクトリを確認します。ただし、この場合にはファイル内のすべてのパラメータがデフォルト値の quit に設定されます。これにより別の問題が発生する可能性があります。

疑問の解決

管理者は、管理ファイルを使用してシステム上のすべてのパッケージのインストールを制御できます。しかし、パッケージの設計者は、しばしば管理者の要望を無視して、代替の管理デフォルトファイルを組み込んでいます。

パッケージの設計者は、管理者ではなく設計者がパッケージのインストールを制御できるように代替管理ファイルを組み込む場合もあります。管理デフォルトファイルの basedir エントリはほかのすべてのベースディレクトリに優先するため、この方法でインストール時に簡単に適切なベースディレクトリを選択することができます。Solaris 2.5 リリースよりも前のすべての Oracle Solaris OS バージョンでは、これがもっとも簡単にベースディレクトリを制御する方法であると考えられていました。

しかし、製品のインストールに関しては管理者の要望を受け入れる必要があります。インストールを制御するために一時的な管理デフォルトファイルを組み込む方法は、管理者の信用を失うことにつながります。requestスクリプトと checkinstall スクリプトを使用して、管理者の監督の下でこれらのインストールを制御するようにしてください。requestスクリプトを誠実に使用してプロセスに管理者を関与させることで、System V のパッケージ化は管理者とパッケージの設計者の双方にとって効果的に機能します。

BASEDIR パラメータの使用

すべての再配置可能パッケージの pkginfo ファイルに、次の形式のエントリでデフォルトベースディレクトリを含める必要があります。

BASEDIR=*absolute_path*

これは唯一のデフォルトベースディレクトリです。管理者はインストール時に変更できます。

複数のベースディレクトリが必要なパッケージもありますが、このパラメータを使用してパッケージを配置する利点は、ベースディレクトリが適切に配置され、インストール開始時に有効なディレクトリとして書き込み可能であることが保証されることです。サーバーとクライアントのベースディレクトリの適切なパスが、予約された環境変数の形式ですべての手続きスクリプトで使用可能になります。また、pkginfo -r SUNWstuf コマンドを使用すると、パッケージの現在のインストールベースを表示できます。

checkinstall スクリプトでは、BASEDIR は pkginfo ファイルで定義されたままのパラメータです。まだ条件は付いていません。ターゲットベースディレクトリを検査するには、${PKG_INSTALL_ROOT}$BASEDIR 構成が必要です。つまり、request スクリプトまたは checkinstall スクリプトを使用して、インストール環境の BASEDIR の値を予測可能な結果に変更できます。システムがクライアントの場合でも、preinstall スクリプトを呼び出すときまでに、BASEDIR パラメータは、ターゲットシステム上の実際のベースディレクトリを指す条件付きポインタになります。

注 – request スクリプトでは、SunOS オペレーティングシステムのリリースごとに異なる BASEDIR パラメータが使用されます。request スクリプトの BASEDIR パラメータをテストするには、次のコードを使用して、実際に使用されるベースディレクトリを判断するようにしてください。

```
# request script
constructs base directory
if [ ${CLIENT_BASEDIR} ]; then
      LOCAL_BASE=$BASEDIR
else
      LOCAL_BASE=${PKG_INSTALL_ROOT}$BASEDIR
fi
```

パラメータ型ベースディレクトリの使用

パッケージに複数のベースディレクトリが必要な場合、パラメータ型パス名を使用して複数のベースディレクトリを確立できます。この方法は一般的になりましたが、次の欠点があります。

- パラメータ型パス名を使用するパッケージは通常は絶対パッケージのように機能しますが、pkgadd コマンドでは再配置可能パッケージのように扱われます。BASEDIR パラメータは、使用しない場合でも定義する必要があります。
- 管理者は、System V ユーティリティーを使用してパッケージのインストールベースを確認することができません。pkginfo -r コマンドは機能しません。
- 管理者は、確立された方法を使用してパッケージを再配置することができません。これは再配置可能パッケージと呼ばれますが、絶対パッケージとして動作します。
- 複数のアーキテクチャーまたは複数のバージョンのインストールでは、ターゲットベースディレクトリごとに不測事態対応計画を作成する必要があります。これは、しばしば複数の複雑なクラスアクションスクリプトを指します。

ベースディレクトリを決定するパラメータは pkginfo ファイルで定義されますが、request スクリプトで変更できます。このことが、このアプローチが一般的である主な理由の 1 つです。ただし、欠点は長期にわたるものであるため、この構成は最後の手段として検討するようにしてください。

例 — パラメータ型ベースディレクトリの使用

pkginfo ファイル

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/
EZDIR=/usr/stuf/EZstuf
HRDDIR=/opt/SUNWstuf/HRDstuf
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none
PSTAMP=hubert980707141632
```

`pkgmap` ファイル

```
: 1 1758
1 d none $EZDIR 0775 root bin
1 f none $EZDIR/dirdel 0555 bin bin 40 773 751310229
1 f none $EZDIR/usrdel 0555 bin bin 40 773 751310229
1 f none $EZDIR/filedel 0555 bin bin 40 773 751310229
1 d none $HRDDIR 0775 root bin
1 f none $HRDDIR/mksmart 0555 bin bin 40 773 751310229
1 f none $HRDDIR/mktall 0555 bin bin 40 773 751310229
1 f none $HRDDIR/mkcute 0555 bin bin 40 773 751310229
1 f none $HRDDIR/mkeasy 0555 bin bin 40 773 751310229
1 d none /etc    ? ? ?
1 d none /etc/rc2.d ? ? ?
1 f none /etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
```

ベースディレクトリの管理

複数のバージョンまたは複数のアーキテクチャーで使用可能なパッケージは、必要に応じてベースディレクトリを調査して設計するようにしてください。ベースディレクトリの調査とは、以前のバージョンまたは異なるアーキテクチャーのパッケージがすでにベースディレクトリにインストールされている場合に、インストールされるパッケージがこの問題を解決することです。通常は、わずかに異なる名前で新しいベースディレクトリが作成されます。Solaris 2.5 および互換リリースの `request` スクリプトと `checkinstall` スクリプトには、`BASEDIR` 環境変数を変更する機能があります。これより前のバージョンの Oracle Solaris OS ではこれがあてはまりません。

古いバージョンの Oracle Solaris OS でも、request スクリプトにはインストールベース内のディレクトリを再定義する権限があります。request スクリプトは、ほとんどの管理設定をサポートするようにディレクトリを再定義できます。

再配置の対応

さまざまなパッケージに対して、アーキテクチャーおよびバージョンごとに一意であることが保証されるベースディレクトリを選択できますが、これによって不要なディレクトリ階層が作成されます。たとえば、SPARC ベースおよび x86 ベースのプロセッサ用の製品では、次のようにプロセッサとバージョンごとにベースディレクトリを編成できます。

ベースディレクトリ	バージョンおよびプロセッサ
/opt/SUNWstuf/sparc/1.0	バージョン 1.0、SPARC
/opt/SUNWstuf/sparc/1.2	バージョン 1.2、SPARC
/opt/SUNWstuf/x86/1.0	バージョン 1.0、x86

これでも問題なく動作しますが、名前と数値が管理者にとって意味を持つように扱っています。より適切なアプローチは、管理者に説明して許可された後、これを自動的に実行することです。

つまり、設計者がパッケージのすべてのジョブを実行できます。管理者が手動で実行する必要はありません。設計者は任意のベースディレクトリを割り当て、postinstall スクリプトで透過的に適切なクライアントリンクを確立できます。また、pkgadd コマンドを使用して、パッケージのすべてまたは一部を postinstall スクリプトでクライアントにインストールすることもできます。このパッケージについて通知する必要があるユーザーまたはクライアントを管理者に確認して、PATH 環境変数と /etc ファイルを自動的に更新することもできます。これは、パッケージのインストール時に行った操作が削除時にすべて元に戻される限り許容されます。

ベースディレクトリの調査

インストール時にベースディレクトリを制御する方法は 2 通りあります。1 つ目の方法は、Solaris 2.5 および互換リリースのみに新しいパッケージをインストールする場合に最適です。この方法では、管理者に非常に有用なデータを提供し、複数のバージョンおよびアーキテクチャーのインストールをサポートし、最小限の作業で実行できます。2 つ目の方法は任意のパッケージで使用でき、構築パラメータに対する request スクリプト固有の制御を使用してインストールを実行します。

BASEDIR パラメータの使用

checkinstall スクリプトは、インストール時に適切なベースディレクトリを選択できます。つまり、ベースディレクトリをディレクトリツリー内の低い位置に配置できます。この例では、/opt/SUNWstuf、/opt/SUNWstuf.1、/opt/SUNWstuf.2 という形式でベースディレクトリを順に増加させます。管理者は、pkginfo コマンドを使用して、各ベースディレクトリにインストールするアーキテクチャーおよびバージョンを決定できます。

SUNWstuf パッケージ (要素となるユーティリティーのセットが含まれる) でこの方法を使用する場合、pkginfo ファイルおよび pkgmap ファイルは次のようになります。

pkginfo ファイル

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/opt/SUNWstuf
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none daemon
PSTAMP=hubert990707141632
```

pkgmap ファイル

```
: 1 1758
1 d none EZstuf 0775 root bin
1 f none EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none HRDstuf 0775 root bin
1 f none HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 d none /etc    ? ? ?
1 d none /etc/rc2.d ? ? ?
1 f daemon /etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
1 i i.daemon 509 39560 752978103
1 i r.daemon 320 24573 742152591
```

例 — BASEDIR を調査する分析スクリプト

x86 版の SUNWstuf がすでにサーバーの /opt/SUNWstuf にインストールされていると仮定します。管理者が pkgadd コマンドを使用して SPARC 版をインストールする場合、request スクリプトは x86 版を検出して、インストールに関して管理者と対話する必要があります。

注 – checkinstall スクリプトでは管理者と対話しなくてもベースディレクトリを調査できますが、このような勝手な処理が頻繁に実行されると、プロセスに対する管理者の信頼を失います。

この状況を処理するパッケージの request スクリプトおよび checkinstall スクリプトは、次のようになっています。

request スクリプト

```
# request script
for SUNWstuf to walk the BASEDIR parameter.

PATH=/usr/sadm/bin:${PATH}    # use admin utilities

GENMSG="The base directory $LOCAL_BASE already contains a \
different architecture or version of $PKG."

OLDMSG="If the option \"-a none\" was used, press the  \
key and enter an unused base directory when it is requested."

OLDPROMPT="Do you want to overwrite this version? "

OLDHELP="\"y\" will replace the installed package, \"n\" will \
stop the installation."

SUSPEND="Suspending installation at user request using error \
code 1."

MSG="This package could be installed at the unused base directory $WRKNG_BASE."

PROMPT="Do you want to use to the proposed base directory? "

HELP="A response of \"y\" will install to the proposed directory and continue,
\"n\" will request a different directory. If the option \"-a none\" was used,
press the  key and enter an unused base directory when it is requested."

DIRPROMPT="Select a preferred base directory ($WRKNG_BASE) "

DIRHELP="The package $PKG will be installed at the location entered."

NUBD_MSG="The base directory has changed. Be sure to update \
any applicable search paths with the actual location of the \
binaries which are at $WRKNG_BASE/EZstuf and $WRKNG_BASE/HRDstuf."

OldSolaris=""
```

```
Changed=""
Suffix="0"

#
# Determine if this product is actually installed in the working
# base directory.
#
Product_is_present () {
      if [ -d $WRKNG_BASE/EZstuf -o -d $WRKNG_BASE/HRDstuf ]; then
            return 1
      else
            return 0
      fi
}

if [ ${BASEDIR} ]; then
      # This may be an old version of Solaris. In the latest Solaris
      # CLIENT_BASEDIR won't be defined yet. In older version it is.
      if [ ${CLIENT_BASEDIR} ]; then
            LOCAL_BASE=$BASEDIR
            OldSolaris="true"
      else    # The base directory hasn't been processed yet
            LOCAL_BASE=${PKG_INSTALL_ROOT}$BASEDIR
fi

WRKNG_BASE=$LOCAL_BASE

    # See if the base directory is already in place and walk it if
    # possible
while [ -d ${WRKNG_BASE} -a Product_is_present ]; do
         # There is a conflict
         # Is this an update of the same arch & version?
         if [ ${UPDATE} ]; then
               exit 0    # It's out of our hands.
         else
               # So this is a different architecture or
               # version than what is already there.
               # Walk the base directory
               Suffix=`expr $Suffix + 1`
               WRKNG_BASE=$LOCAL_BASE.$Suffix
               Changed="true"
         fi
done

    # So now we can propose a base directory that isn't claimed by
    # any of our other versions.
if [ $Changed ]; then
         puttext "$GENMSG"
         if [ $OldSolaris ]; then
               puttext "$OLDMSG"
               result=`ckyorn -Q -d "a" -h "$OLDHELP" -p "$OLDPROMPT"`
               if [ $result="n" ]; then
                     puttext "$SUSPEND"
                     exit 1    # suspend installation
               else
                     exit 0
               fi
         else    # The latest functionality is available
               puttext "$MSG"
```

```
                result=‘ckyorn -Q -d "a" -h "$HELP" -p "$PROMPT"‘
                if [ $? -eq 3]; then
                      echo quitinstall >> $1
                      exit 0
                fi

                if [ $result="n" ]; then
                      WRKNG_BASE=‘ckpath -ayw -d "$WRKNG_BASE" \
                      -h "$DIRHELP" -p "$DIRPROMPT"‘
                else if [ $result="a" ]
                      exit 0
                fi
           fi
           echo "BASEDIR=$WRKNG_BASE" >> $1
           puttext "$NUBD_MSG"
      fi
fi
exit 0
```

checkinstall スクリプト

```
# checkinstall
script for SUNWstuf to politely suspend

grep quitinstall $1
if [ $? -eq 0 ]; then
    exit 3        # politely suspend installation
fi

exit 0
```

このアプローチは、ベースディレクトリが単に /opt だった場合には正常に機能しません。/opt を調査するのは困難であるため、このパッケージは BASEDIR をより正確に呼び出す必要があります。実際、マウントスキームによっては不可能な場合もあります。この例では、/opt の下に新しいディレクトリを作成してベースディレクトリを調査しています。これによって問題が発生することはありません。

この例では request スクリプトと checkinstall スクリプトを使用していますが、2.5 リリースより前のバージョンの Oracle Solaris では checkinstall スクリプトを実行することはできません。この例の checkinstall スクリプトは、quitinstall という文字列形式の非公開メッセージに応答して、インストールをていねいに停止するために使用されます。Solaris 2.3 リリースでこのスクリプトを実行する場合、checkinstall スクリプトは無視され、request スクリプトはエラーメッセージを表示してインストールを停止します。

Solaris 2.5 および互換リリースより前では、BASEDIR パラメータは読み取り専用パラメータであり、request スクリプトで変更することはできません。このため、(条件付きの CLIENT_BASEDIR 環境変数をテストして) 旧バージョンの SunOS オペレーティングシステムが検出された場合、request スクリプトには継続するか終了するかの 2 つのオプションしかありません。

相対パラメータ型パスの使用

ソフトウェア製品が旧バージョンの SunOS オペレーティングシステムにインストールされる可能性がある場合、request スクリプトで必要な作業をすべて実行する必要があります。このアプローチを使用すると、複数のディレクトリを操作することもできます。追加のディレクトリが必要な場合、簡単に管理できる製品を提供するには、単一のベースディレクトリの下にディレクトリを含める必要があります。BASEDIR パラメータでは最新の Oracle Solaris リリースで使用できる粒度のレベルは提供されませんが、request スクリプトを使用してパラメータ型パスを操作することで、パッケージからベースディレクトリを調査できます。pkginfo ファイルおよび pkgmap ファイルは次のようになります。

pkginfo ファイル

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/opt
SUBBASE=SUNWstuf
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none daemon
PSTAMP=hubert990707141632
```

pkgmap ファイル

```
: 1 1758
1 d none $SUBBASE/EZstuf 0775 root bin
1 f none $SUBBASE/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none $SUBBASE/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none $SUBBASE/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none $SUBBASE/HRDstuf 0775 root bin
1 f none $SUBBASE/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none $SUBBASE/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none $SUBBASE/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none $SUBBASE/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 d none /etc    ? ? ?
1 d none /etc/rc2.d ? ? ?
1 f daemon /etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
1 i i.daemon 509 39560 752978103
1 i r.daemon 320 24573 742152591
```

この例は完璧ではありません。pkginfo -r コマンドではインストールベースに /opt が返されますが、これは非常にあいまいです。多くのパッケージが /opt にありますが、少なくともこれは意味のあるディレクトリです。前の例と同様、次の例でも複数のアーキテクチャーおよびバージョンをサポートしています。request スクリプトは、特定のパッケージの要件に合わせて、適切な依存関係を解決することができます。

例 — 相対パラメータ型パスを調査する request スクリプト

```
# request script
for SUNWstuf to walk a parametric path

PATH=/usr/sadm/bin:${PATH}    # use admin utilities

MSG="The target directory $LOCAL_BASE already contains \
different architecture or version of $PKG. This package \
could be installed at the unused target directory $WRKNG_BASE."

PROMPT="Do you want to use to the proposed directory? "

HELP="A response of \"y\" will install to the proposed directory \
and continue, \"n\" will request a different directory. If \
the option \"-a none\" was used, press the <RETURN> key and \
enter an unused base directory when it is requested."

DIRPROMPT="Select a relative target directory under $BASEDIR/"

DIRHELP="The package $PKG will be installed at the location entered."

SUSPEND="Suspending installation at user request using error \
code 1."

NUBD_MSG="The location of this package is not the default. Be \
sure to update any applicable search paths with the actual \
location of the binaries which are at $WRKNG_BASE/EZstuf \
and $WRKNG_BASE/HRDstuf."

Changed=""
Suffix="0"

#
# Determine if this product is actually installed in the working
# base directory.
#
Product_is_present () {
      if [ -d $WRKNG_BASE/EZstuf -o -d $WRKNG_BASE/HRDstuf ]; then
            return 1
      else
            return 0

      fi
}

if [ ${BASEDIR} ]; then
      # This may be an old version of Solaris. In the latest Solaris
      # CLIENT_BASEDIR won't be defined yet. In older versions it is.
```

```
      if [ ${CLIENT_BASEDIR} ]; then
            LOCAL_BASE=$BASEDIR/$SUBBASE
      else    # The base directory hasn't been processed yet
            LOCAL_BASE=${PKG_INSTALL_ROOT}$BASEDIR/$SUBBASE
      fi

WRKNG_BASE=$LOCAL_BASE

# See if the base directory is already in place and walk it if
# possible
while [ -d ${WRKNG_BASE} -a Product_is_present ]; do
         # There is a conflict
         # Is this an update of the same arch & version?
         if [ ${UPDATE} ]; then
               exit 0    # It's out of our hands.
         else
               # So this is a different architecture or
               # version than what is already there.
               # Walk the base directory
               Suffix=`expr $Suffix + 1`
               WRKNG_BASE=$LOCAL_BASE.$Suffix
               Changed="true"
         fi
done

# So now we can propose a base directory that isn't claimed by
# any of our other versions.
if [ $Changed ]; then
         puttext "$MSG"
         result=`ckyorn -Q -d "a" -h "$HELP" -p "$PROMPT"`
         if [ $? -eq 3 ]; then
               puttext "$SUSPEND"
               exit 1
         fi

         if [ $result="n" ]; then
               WRKNG_BASE=`ckpath -lyw -d "$WRKNG_BASE" -h "$DIRHELP" \
               -p "$DIRPROMPT"`

            elif [ $result="a" ]; then
                   exit 0
            else
                   exit 1
            fi
            echo SUBBASE=$SUBBASE.$Suffix >> $1
            puttext "$NUBD_MSG"
      fi
fi
exit 0
```

異機種システム混在環境での再配置のサポート

System V のパッケージ化の背後にあるもともとの概念では、システムごとに 1 つのアーキテクチャーを想定していました。サーバーの概念は設計に関与しませんでした。今では当然、単一のサーバーが複数のアーキテクチャーをサポートしています。つまり、1 つのサーバー上に同じソフトウェアが異なるアーキテクチャーごとに複数存在する場合があります。Oracle Solaris パッケージは推奨されるファイルシステムの境界内 (/ や /usr など) に隔離されますが、サーバーおよび各クライアント上の製品データベースでは、この分割がすべてのインストールでサポートされているとは限りません。特定の実装では、まったく異なる構造をサポートして、共通製品データベースを含んでいます。クライアントを異なるバージョンに向けることは簡単ですが、実際に System V のパッケージを異なるベースディレクトリにインストールした場合、管理者にとって複雑な状況をもたらす可能性があります。

パッケージを設計する場合、管理者が新しいバージョンのソフトウェアのインストールに使用する一般的な方法についても検討するようにしてください。管理者は、しばしば現在インストールされているバージョンと共存させて最新バージョンをインストールし、テストしようとします。その手順として、現在のバージョンとは異なるベースディレクトリに新しいバージョンをインストールして、少数の重要ではないクライアントをテストとして新しいバージョンにします。確信が得られたら、管理者はより多くのクライアントを新しいバージョンにします。最終的に、管理者は非常時用にのみ旧バージョンを保持して、最後には削除します。

つまり、現代の異機種システムで使用するパッケージは、管理者がファイルシステム上の任意の適切な場所に配置でき、正常に動作するという意味で本当の再配置をサポートする必要があります。Solaris 2.5 および互換リリースでは、多数の有用なツールが提供され、同じシステムに複数のアーキテクチャーおよびバージョンをクリーンにインストールできます。Solaris 2.4 および互換バージョンでも本当の再配置をサポートしていますが、そのためのタスクは明白ではありません。

従来のアプローチ

再配置可能パッケージ

System V ABI は、再配置可能パッケージの背後にある当初の目的は、パッケージのインストールを管理者にとってより便利にすることだったということを含意しています。今では、再配置可能パッケージの必要性はそれ以上のものになっています。利便性だけの問題ではなく、インストール時にアクティブなソフトウェア製品がすでにデフォルトディレクトリにインストールされている可能性が非常に高くなっています。この状況に対応できないパッケージは、既存の製品を上書きするか、インストールに失敗します。しかし、複数のアーキテクチャーおよび複数のバージョンに対応したパッケージは、スムーズにインストールでき、従来の管理方法と十分に互換性がある幅広いオプションを管理者に提供します。

ある意味で、複数のアーキテクチャーの問題と複数のバージョンの問題は同じです。既存のパッケージのバリアントは、ほかのバリアントと共存させてインストールできる必要があります。また、エクスポートされたファイルシステムのクライアントまたはスタンドアロンのコンシューマは、機能を低下させることなく、いずれかのバリアントに変更できる必要もあります。Sun はサーバー上で複数のアーキテクチャーに対応する方法を確立していますが、管理者はこれらの提案に従わない場合もあります。すべてのパッケージは、管理者のインストールに関する合理的な要望に応じることができるようにする必要があります。

例 - 従来の再配置可能パッケージ

この例では、従来の再配置可能パッケージを示します。パッケージは `/opt/SUNWstuf` に配置され、`pkginfo` ファイルおよび `pkgmap` ファイルは次のようになります。

pkginfo ファイル

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/opt
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none
PSTAMP=hubert990707141632
```

pkgmap ファイル

```
: 1 1758
1 d none SUNWstuf 0775 root bin
1 d none SUNWstuf/EZstuf 0775 root bin
1 f none SUNWstuf/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none SUNWstuf/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none SUNWstuf/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none SUNWstuf/HRDstuf 0775 root bin
1 f none SUNWstuf/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
```

これが従来の方法と呼ばれる理由は、すべてのパッケージオブジェクトが pkginfo ファイルの BASEDIR パラメータで定義されるベースディレクトリにインストールされるためです。たとえば、pkgmap ファイルの最初のオブジェクトは /opt/SUNWstuf ディレクトリにインストールされます。

絶対パッケージ

絶対パッケージは、ファイルシステムの特定のルート (/) にインストールされるパッケージです。絶対パッケージは、複数のバージョンおよびアーキテクチャーの立場から対処することが困難です。原則的に、すべてのパッケージを再配置可能にしてください。ただし、再配置可能パッケージに絶対的な要素を含める合理的な理由もあります。

例 - 従来の絶対パッケージ

SUNWstuf パッケージが絶対パッケージの場合、BASEDIR パラメータは pkginfo ファイルでは定義されず、pkgmap ファイルは次のようになります。

pkgmap ファイル

```
: 1 1758
1 d none /opt ? ? ?
1 d none /opt/SUNWstuf 0775 root bin
1 d none /opt/SUNWstuf/EZstuf 0775 root bin
1 f none /opt/SUNWstuf/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none /opt/SUNWstuf/HRDstuf 0775 root bin
1 f none /opt/SUNWstuf/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
```

この例では、管理者がインストール時に代替ベースディレクトリを指定した場合、pkgadd コマンドでは無視されます。このパッケージは、常にターゲットシステムの /opt/SUNWstuf にインストールされます。

pkgadd コマンドの -R 引数は予期されるとおりに機能します。たとえば、

```
pkgadd -d . -R /export/opt/client3 SUNWstuf
```

オブジェクトは /export/opt/client3/opt/SUNWstuf にインストールされますが、これはこのパッケージが再配置可能になることとほぼ同じです。

pkgmap ファイルの /opt ディレクトリに疑問符 (?) を使用しています。これは、既存の属性を変更できないことを示します。これは「デフォルト属性でディレクトリを作成する」ということではありませんが、特定の状況ではそうなる場合もあります。新しいパッケージに固有のディレクトリは、すべての属性を明示的に指定する必要があります。

複合パッケージ

再配置可能オブジェクトを含むパッケージは、再配置可能パッケージと呼ばれます。再配置可能パッケージは pkgmap ファイルに絶対パスが含まれる場合があるため、これは誤解を招く可能性があります。pkgmap ファイルでルート (/) エントリを使用することで、パッケージの再配置可能な側面を強化できます。再配置可能なエントリとルートエントリの両方が含まれるパッケージは、複合パッケージと呼ばれます。

例 - 従来の解決方法

SUNWstuf パッケージのオブジェクトの 1 つが、実行レベル 2 で実行される起動スクリプトだと仮定します。/etc/rc2.d/S70dostuf ファイルはパッケージの一部としてインストールする必要がありますが、ベースディレクトリに配置することはできません。再配置可能パッケージが唯一の解決方法だとすると、pkginfo および pkgmap は次のようになります。

pkginfo ファイル

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none
PSTAMP=hubert990707141632
```

pkgmap ファイル

```
: 1 1758
1 d none opt/SUNWstuf/EZstuf 0775 root bin
1 f none opt/SUNWstuf/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none opt/SUNWstuf/HRDstuf 0775 root bin
1 f none opt/SUNWstuf/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/HRDstuf/mktall 0555 bin bin 40 773 751310229
```

```
1 f none opt/SUNWstuf/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 d none etc    ? ? ?
1 d none etc/rc2.d ? ? ?
1 f none etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
```

このアプローチと絶対パッケージのアプローチには大きな違いはありません。実際、これは絶対パッケージよりも良い状態です。管理者がこのパッケージの代替ベースディレクトリを指定した場合、このパッケージは機能しません。

このパッケージのファイルは 1 つだけルートと相対的にする必要がありますが、残りは任意の場所に移動できます。複合パッケージを使用してこの問題を解決する方法は、このセクションの後半で説明します。

従来の方法を超えて

このセクションで説明するアプローチはすべてのパッケージには適用されませんが、異機種システム混在環境にインストールする場合にパフォーマンスを向上できます。このアプローチは、Oracle Solaris OS の一部として提供されるパッケージ (バンドル版のパッケージ) にはほとんど適用されませんが、アンバンドルのパッケージでは従来とは異なるパッケージ化を実行できます。

再配置可能パッケージを奨励する理由は、次の要件をサポートするためです。

パッケージを追加または削除しても、インストールされたソフトウェア製品の既存の動作は変わりません。

アンバンドルのパッケージは、新しいパッケージが既存の製品と干渉しないことを保証できるように、`/opt` の下に配置するようにしてください。

複合パッケージの別の側面

有効な複合パッケージを構築する場合、次の 2 つの規則に従います。

- パッケージオブジェクトの大部分が配置される場所に基づいてベースディレクトリを確立します。
- パッケージオブジェクトがベースディレクトリ以外の共通ディレクトリ (例: `/etc`) に配置される場合、`prototype` ファイルで絶対パス名として指定します。

つまり、「再配置可能」とはオブジェクトを任意の場所にインストールして実行できることを意味するため、ブート時に `init` で実行される起動スクリプトは再配置可能と見なすことができません。提供されるパッケージで相対パスとして `/etc/passwd` を指定することに問題はありませんが、配置できる場所は 1 つしかありません。

再配置可能に見える絶対パス名の作成

複合パッケージを構築する場合、インストールされた既存のソフトウェアに干渉しないように絶対パスを処理する必要があります。すべて /opt に含まれるパッケージでは、既存のファイルが存在しないためこの問題を回避できます。/etc のファイルがパッケージに含まれる場合、絶対パス名が相対パス名から予測されるのと同様に機能することを保証する必要があります。次の 2 つの例を考えてみましょう。

例 — ファイルの変更

説明

エントリがテーブルに追加されるか、オブジェクトがほかのプログラムまたはパッケージによって変更される可能性が高い新しいテーブルです。

実装

オブジェクトを、build、awk、または sed クラスに属するファイルタイプ e として定義します。このタスクを実行するスクリプトは、自身を追加するのと同程度効率的に自身を削除する必要があります。

例

新しいソリッドステートのハードディスクをサポートするには、エントリを /etc/vfstab に追加する必要があります。

pkgmap ファイルのエントリは次のようになります。

```
1 e sed /etc/vfstab ? ? ?
```

request スクリプトは、パッケージで /etc/vfstab を変更するかどうかをオペレータに確認します。オペレータが "no" と答えると、このジョブを手動で実行する方法が表示され、次の処理が実行されます。

```
echo "CLASSES=none" >> $1
```

オペレータが "yes" と答えると、次の処理が実行されます。

```
echo "CLASSES=none sed" >> $1
```

これによって、必要な変更を行うクラスアクションスクリプトが起動されます。sed クラスは、パッケージファイル /etc/vfstab が、ターゲットシステム上の同じ名前のファイルに対するインストールおよび削除の両方の処理を含む sed プログラムであることを意味します。

例 — 新しいファイルの作成

説明

オブジェクトはまったく新しいファイルで、後から編集される可能性は高くありません。または、別のパッケージが所有するファイルを置き換えます。

実装

パッケージオブジェクトをファイルタイプ f として定義して、変更を取り消すことができるクラスアクションスクリプトを使用してインストールします。

例

ソリッドステートのハードディスクをサポートするために必要な情報を提供するには、/etc に /etc/shdisk.conf という名前の新しいファイルが必要です。pkgmap ファイルのエントリは、次のようになります。

```
.
.
.
1 f newetc /etc/shdisk.conf
.
.
.
```

クラスアクションスクリプト i.newetc は、/etc に配置する必要があるファイルのインストールに使用されます。このスクリプトは、所定の場所に別のファイルが存在するかどうかを確認します。存在しない場合は新しいファイルをコピーするだけです。所定の場所にファイルが存在する場合、そのファイルをバックアップしてから新しいファイルをインストールします。スクリプト r.newetc は、必要に応じてこれらのファイルを削除して、元のファイルを復元します。インストールスクリプトの重要な部分を次に示します。

```
# i.newetc
while read src dst; do
      if [ -f $dst ]; then
        dstfile=‘basename $dst‘
        cp $dst $PKGSAV/$dstfile
      fi
      cp $src $dst
done

if [ "${1}" = "ENDOFCLASS" ]; then
      cd $PKGSAV
      tar cf SAVE.newetc .
      $INST_DATADIR/$PKG/install/squish SAVE.newetc
fi
```

このスクリプトでは、PKGSAV 環境変数を使用して置き換えるファイルのバックアップを格納しています。引数 ENDOFCLASS がスクリプトに渡された場合、これらがこのクラスの最後のエントリであることをスクリプトに通知するのは pkgadd コマンドです。この時点で、パッケージのインストールディレクトリに格納された非公開の圧縮プログラムを使用して、保存されたファイルがアーカイブおよび圧縮されます。

パッケージの更新時には PKGSAV 環境変数の使用は信頼性がありませんが、(たとえばパッチによって) パッケージが更新されない場合、バックアップファイルはセキュリティー保護されています。次の削除スクリプトには、旧バージョンの pkgrm コマンドがスクリプトに PKGSAV 環境変数への正しいパスを渡さないという別の問題に対応するコードが含まれています。

削除スクリプトは次のようになります。

```
# r.newetc

# make sure we have the correct PKGSAV
if [ -d $PKG_INSTALL_ROOT$PKGSAV ]; then
      PKGSAV="$PKG_INSTALL_ROOT$PKGSAV"
fi

# find the unsquish program
UNSQUISH_CMD=‘dirname $0‘/unsquish

while read file; do
      rm $file
done

if [ "${1}" = ENDOFCLASS ]; then
      if [ -f $PKGSAV/SAVE.newetc.sq ]; then
         $UNSQUISH_CMD $PKGSAV/SAVE.newetc
      fi

      if [ -f $PKGSAV/SAVE.newetc ]; then
         targetdir=dirname $file    # get the right directory
         cd $targetdir
            tar xf $PKGSAV/SAVE.newetc
            rm $PKGSAV/SAVE.newetc
      fi
fi
```

このスクリプトでは、パッケージデータベースのインストールディレクトリの非公開アンインストールアルゴリズム (unsquish) が使用されます。これはインストール時に pkgadd コマンドによって自動的に実行されます。pkgadd コマンドによって明確にインストール専用として認識されないスクリプトはすべて、pkgrm コマンドで使用するためにこのディレクトリに残されます。このディレクトリの場所を知ることはできませんが、このディレクトリが平坦で、パッケージの適切な情報ファイルおよびインストールスクリプトがすべて含まれていることは信頼できます。このスクリプトでは、 unsquish プログラムが含まれるディレクトリからクラスアクションスクリプトが実行されることが保証されていることに基づいてディレクトリを検索します。

また、このスクリプトでは、ターゲットディレクトリが /etc だけであるとは仮定していません。実際には /export/root/client2/etc である場合もあります。正しいディレクトリは、2 通りの方法のいずれかで構成できます。

- ${PKG_INSTALL_ROOT}/etc 構成を使用する方法
- pkgadd コマンドで渡されるファイルのディレクトリ名を取得する方法 (このスクリプトで実行)

パッケージ内の絶対オブジェクトごとにこのアプローチを使用することで、現在の望ましい動作が変わらないか、少なくとも回復可能であることを保証できます。

例 — 複合パッケージ

次に示すのは、複合パッケージに対して pkginfo ファイルと pkgmap ファイルを使用する例です。

pkginfo ファイル

```
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/opt
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none daemon
PSTAMP=hubert990707141632
```

pkgmap ファイル

```
: 1 1758
1 d none SUNWstuf/EZstuf 0775 root bin
1 f none SUNWstuf/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none SUNWstuf/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none SUNWstuf/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none SUNWstuf/HRDstuf 0775 root bin
1 f none SUNWstuf/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 d none /etc    ? ? ?
1 d none /etc/rc2.d ? ? ?
1 e daemon /etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i i.daemon 509 39560 752978103
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
1 i r.daemon 320 24573 742152591
```

S70dostuf は daemon クラスに属しますが、これにつながるディレクトリ (インストール時に配置済み) は none クラスに属します。ディレクトリがこのパッケージに固有の場合でも、none クラスのままにしてください。この理由は、最初にディレクトリを作成して、最後に削除する必要がありますが、これは none クラスの場合も常に該当するためです。pkgadd コマンドによってディレクトリが作成されます。このディレクトリは、パッケージからコピーされたり、作成されるクラスアクションスクリプトに渡されたりすることはありません。その代わり、pkgadd コマンドで作成してからインストールクラスアクションスクリプトが呼び出され、削除クラスアクションスクリプトの完了後に、pkgrm コマンドによってディレクトリが削除されます。

つまり、特殊クラスのディレクトリに none クラスのオブジェクトが含まれる場合、pkgrm コマンドでディレクトリを削除しようとすると、ディレクトリが時間内に空にならないので失敗します。none クラスのオブジェクトが特殊クラスのディレクトリに挿入される場合、時間内にオブジェクトを受け入れるディレクトリは存在しません。pkgadd コマンドにより、オブジェクトのインストール時にオンザフライでディレクトリが作成され、最終的に pkgmap 定義を確認した場合にそのディレクトリの属性を同期できない場合があります。

注 – クラスにディレクトリを割り当てる場合は、常に作成と削除の順序を覚えておいてください。

リモートでインストール可能なパッケージの作成

すべてのパッケージをリモートでインストール可能にする必要があります。リモートでインストール可能とは、パッケージをインストールする管理者が、pkgadd コマンドを実行するシステムのルート (/) ファイルシステムにインストールすることを仮定できないということです。手続きスクリプトの 1 つで、ターゲットシステムの /etc/vfstab ファイルを取得する必要がある場合、PKG_INSTALL_ROOT 環境変数を使用する必要があります。つまり、パス名 /etc/vfstab を使用して pkgadd コマンドを実行するシステムの /etc/vfstab ファイルを取得できますが、管理者はクライアントの /export/root/client3 にインストールする場合もあります。パス ${PKG_INSTALL_ROOT}/etc/vfstab は、ターゲットファイルシステムを取得できることが保証されています。

例 – クライアントシステムへのインストール

この例では、SUNWstuf パッケージを、ルート (/) ファイルシステムの /opt で構成された client3 にインストールします。このパッケージのほかのバージョンの 1 つが、すでに client3 にインストールされています。また、ベースディレクトリは管理ファイル thisadmin から basedir=/opt/$PKGINST に設定されています。管理ファイル

の詳細については、132 ページの「管理デフォルトファイル」を参照してください。サーバーで実行する pkgadd コマンドは、次のとおりです。

```
# pkgadd -a thisadmin -R /export/root/client3 SUNWstuf
```

次の表は、環境変数と、手続きスクリプトに渡される値の一覧です。

表 6–1 手続きスクリプトに渡される値

環境変数	値
PKGINST	SUNWstuf.2
PKG_INSTALL_ROOT	/export/root/client3
CLIENT_BASEDIR	/opt/SUNWstuf.2
BASEDIR	/export/root/client3/opt/SUNWstuf.2

例 – サーバーまたはスタンドアロンシステムへのインストール

前の例と同じ状況でサーバーまたはスタンドアロンシステムにインストールするには、次のコマンドを使用します。

```
# pkgadd -a thisadmin SUNWstuf
```

次の表は、環境変数と、手続きスクリプトに渡される値の一覧です。

表 6–2 手続きスクリプトに渡される値

環境変数	値
PKGINST	SUNWstuf.2
PKG_INSTALL_ROOT	未定義。
CLIENT_BASEDIR	/opt/SUNWstuf.2
BASEDIR	/opt/SUNWstuf.2

例 – 共有ファイルシステムのマウント

SUNWstufパッケージにより、サーバーに /export/SUNWstuf/share を作成し、ファイルシステムを共有すると仮定します。パッケージをクライアントシステムにインストールする場合、/etc/vfstab ファイルを更新して、この共有ファイルシステムにマウントする必要があります。これは、CLIENT_BASEDIR 変数を使用できる場合です。

クライアントのエントリは、クライアントのファイルシステムを参照してマウントポイントを指定する必要があります。インストールをサーバーから行う場合でもクライアントから行う場合でも、この行を正確に構成するようにしてください。サーバーのシステム名は `$SERVER` であるとします。`$PKG_INSTALL_ROOT/etc/vfstab` に移動して、`sed` コマンドまたは `awk` コマンドを使用してクライアントの `/etc/vfstab` ファイルの次の行を構成します。

```
$SERVER:/export/SUNWstuf/share - $CLIENT_BASEDIR/usr nfs - yes ro
```

たとえば、サーバー `universe` とクライアントシステム `client9` の場合、クライアントシステムの `/etc/vfstab` ファイルは、次の行になります。

```
universe:/export/SUNWstuf/share - /opt/SUNWstuf.2/usr nfs - yes ro
```

これらのパラメータを正確に使用して、ローカルで構成する場合でもサーバーから構成する場合でも常にクライアントのファイルシステムをマウントします。

パッケージのパッチ

パッケージのパッチは、元のパッケージの特定のファイルを上書きするためのスパースパッケージです。供給媒体の容量を節約することを除いて、スパースパッケージを出荷することに実質的な理由はありません。少数のファイルを変更して元のパッケージ全体を出荷したり、ネットワーク上で変更されたパッケージにアクセスできるようにしたりすることもできます。実際に異なっているのはこれらの新しいファイルだけである (その他のファイルは再コンパイルされていない) 場合に限り、`pkgadd` コマンドで差分をインストールできます。パッケージのパッチについては、次のガイドラインを確認してください。

- 複雑なシステムでは、パッチ識別システムを確立して、別々の異常な動作を訂正する場合に 2 つのパッチが同じファイルを置き換えることがないようにします。たとえば、Sun のパッチベース番号は、ターゲットファイルの相互排他的なセットに割り当てられます。
- パッチをバックアウトできるようにする必要があります。

パッチパッケージのバージョン番号を、元のパッケージのバージョン番号と同じにすることが重要です。次の形式の `pkginfo` ファイルエントリを個別に使用して、パッケージのパッチステータスを管理するようにしてください。

`PATCH=`*patch_number*

パッチのためにパッケージのバージョンが変わる場合、パッケージの別のインスタンスを作成します。パッチを適用された製品を管理することは極めて困難になります。この連続的なインスタンスパッチの方法は、Oracle Solaris OS の早期リリースで一定の利点をもたらしましたが、複雑なシステムの管理では面倒になります。

パッチ内のすべてのゾーンパラメータがパッケージ内のゾーンパラメータと一致している必要があります。

Oracle Solaris OS を構成するパッケージに関するかぎり、パッケージデータベースにはパッケージのコピーは 1 つしかないはずです。ただし、パッチを適用されたインスタンスは複数ある場合があります。インストールされたパッケージから (removef コマンドを使用して) オブジェクトを削除するには、そのファイルを所有するインスタンスを特定する必要があります。

ただし、(Oracle Solaris OS の一部ではない) パッケージが、Oracle Solaris OS の一部である特定のパッケージのパッチレベルを決定する必要がある場合、これは解決すべき問題になります。インストールスクリプトは、ターゲットファイルシステムに格納されないため、重大な影響を与えることなく大きくすることができます。クラスアクションスクリプトおよびその他のさまざまな手続きスクリプトにより、PKGSAV 環境変数 (またはその他の永続的なディレクトリ) を使用して、変更されたファイルを保存し、インストールされたパッチのバックアウトを可能にすることができます。また、request スクリプトで適切な環境変数を設定して、パッチの履歴を監視することもできます。次のセクションのスクリプトでは、複数のパッチが存在する可能性があることを仮定しています。パッチの番号付けスキームは、単一のパッケージに適用する場合に意味を持ちます。この場合、個別のパッチ番号は、パッケージ内の機能的に関連したファイルのサブセットを表します。2 つのパッチ番号が異なる場合、同じファイルを変更することはできません。

通常のスパースパッケージをパッチパッケージにするには、次のセクションで説明するスクリプトをパッケージに組み込みます。最後の 2 つが patch_checkinstall および patch_postinstall という名前であることを除いて、すべてを標準パッケージコンポーネントとして認識できます。パッチのバックアウト機能が必要な場合には、これら 2 つのスクリプトをバックアウトパッケージに組み込むことができます。スクリプトは非常に単純で、さまざまなタスクは明確です。

注 – このパッチの方法は、クライアントシステムのパッチに使用できますが、サーバー上のクライアントルートディレクトリには、ユーザーの install または nobody で読み取りできる適切な権限が必要です。

---

checkinstall スクリプト

checkinstall スクリプトを使用して、特定のパッケージについてパッチが適切であることを確認します。確認が終わると、パッチ一覧とパッチ情報一覧が作成され、レスポンスファイルに挿入してパッケージデータベースに組み込まれます。

パッチ一覧は現在のパッケージに影響するパッチの一覧です。このパッチ一覧は、インストールされたパッケージの pkginfo ファイルに次のような行で記録されます。

PATCHLIST=*patch_id patch_id ...*

パッチ情報一覧は、現在のパッチが依存しているパッチの一覧です。パッチ情報一覧も、pkginfo ファイルに次のような行で記録されます。

```
PATCH_INFO_103203-01=Installed... Obsoletes:103201-01 Requires: \ Incompatibles: 120134-01
```

注 – これらの行 (およびその形式) は、公開インタフェースとして宣言されています。Oracle Solaris パッケージのパッチを出荷する企業は、この一覧を適切に更新するようにしてください。パッチが提供される場合、パッチ内の各パッケージにはこのタスクを実行する checkinstall スクリプトが含まれます。同じ checkinstall スクリプトにより、その他のパッチ固有のパラメータも更新されます。これは、直接インスタンスパッチと呼ばれる新しいパッチアーキテクチャーです。

この例では、元のパッケージとそのパッチの両方が同じディレクトリに存在します。2 つの元のパッケージは、SUNWstuf.v1 および SUNWstuf.v2 という名前です。パッチは、 SUNWstuf.p1 および SUNWstuf.p2 という名前です。これによって、手続きスクリプトでこれらのファイルの元のディレクトリを特定するのは非常に困難になる場合があります。PKG パラメータではパッケージ名の点 (.) の後ろがすべて削除され、PKGINST 環境変数はソースインスタンスではなくインストールされたインスタンスを参照するためです。手続きスクリプトでソースディレクトリを特定するには、checkinstall スクリプト (常にソースディレクトリから実行される) で照会を行い、SCRIPTS_DIR 変数として場所を渡します。SUNWstuf というソースディレクトリにパッケージが 1 つしかなかった場合には、手続きスクリプトで $INSTDIR/$PKG を使用して検出できました。

```
# checkinstall script to control a patch installation.
# directory format options.
#
#       @(#)checkinstall 1.6 96/09/27 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

PATH=/usr/sadm/bin:$PATH

INFO_DIR=‘dirname $0‘
INFO_DIR=‘dirname $INFO_DIR‘    # one level up

NOVERS_MSG="PaTcH_MsG 8 Version $VERSION of $PKG is not installed on this system."
ALRDY_MSG="PaTcH_MsG 2 Patch number $Patch_label is already applied."
TEMP_MSG="PaTcH_MsG 23 Patch number $Patch_label cannot be applied until all \
restricted patches are backed out."

# Read the provided environment from what may have been a request script
. $1

# Old systems can’t deal with checkinstall scripts anyway
if [ "$PATCH_PROGRESSIVE" = "true" ]; then
        exit 0
```

```
fi

#
# Confirm that the intended version is installed on the system.
#
if [ "${UPDATE}" != "yes" ]; then
        echo "$NOVERS_MSG"
        exit 3
fi

#
# Confirm that this patch hasn't already been applied and
# that no other mix-ups have occurred involving patch versions and
# the like.
#
Skip=0
active_base=`echo $Patch_label | nawk '
        { print substr($0, 1, match($0, "Patchvers_pfx")-1) } '`
active_inst=`echo $Patch_label | nawk '
        { print substr($0, match($0, "Patchvers_pfx")+Patchvers_pfx_lnth) } '`

# Is this a restricted patch?
if echo $active_base | egrep -s "Patchstrict_str"; then
        is_restricted="true"
        # All restricted patches are backoutable
        echo "PATCH_NO_UNDO=" >> $1
else
        is_restricted="false"
fi

for patchappl in ${PATCHLIST}; do
        # Is this an ordinary patch applying over a restricted patch?
        if [ $is_restricted = "false" ]; then
                if echo $patchappl | egrep -s "Patchstrict_str"; then
                        echo "$TEMP_MSG"
                        exit 3;
                fi
        fi

        # Is there a newer version of this patch?
        appl_base=`echo $patchappl | nawk '
                { print substr($0, 1, match($0, "Patchvers_pfx")-1) } '`
        if [ $appl_base = $active_base ]; then
                appl_inst=`echo $patchappl | nawk '
                        { print substr($0, match($0, "Patchvers_pfx")\
+Patchvers_pfx_lnth) } '`
                result=`expr $appl_inst \> $active_inst`
                if [ $result -eq 1 ]; then
                        echo "PaTcH_MsG 1 Patch number $Patch_label is \
superceded by the already applied $patchappl."
                        exit 3
                elif [ $appl_inst = $active_inst ]; then
                        # Not newer, it's the same
                        if [ "$PATCH_UNCONDITIONAL" = "true" ]; then
                                if [ -d $PKGSAV/$Patch_label ]; then
                                        echo "PATCH_NO_UNDO=true" >> $1
                                fi
                        else
                                echo "$ALRDY_MSG"
```

```
                                exit 3;
                        fi
                fi
        fi
done

# Construct a list of applied patches in order
echo "PATCHLIST=${PATCHLIST} $Patch_label" >> $1

#
# Construct the complete list of patches this one obsoletes
#
ACTIVE_OBSOLETES=$Obsoletes_label

if [ -n "$Obsoletes_label" ]; then
        # Merge the two lists
        echo $Obsoletes_label | sed 'y/\ /\n/' | \
        nawk -v PatchObsList="$PATCH_OBSOLETES" '
        BEGIN {
                printf("PATCH_OBSOLETES=");
                PatchCount=split(PatchObsList, PatchObsComp, " ");

                for(PatchIndex in PatchObsComp) {
                        Atisat=match(PatchObsComp[PatchIndex], "@");
                        PatchObs[PatchIndex]=substr(PatchObsComp[PatchIndex], \
0, Atisat-1);
                        PatchObsCnt[PatchIndex]=substr(PatchObsComp\
[PatchIndex], Atisat+1);
                }
        }
        {
                Inserted=0;
                for(PatchIndex in PatchObs) {
                        if (PatchObs[PatchIndex] == $0) {
                                if (Inserted == 0) {
                                        PatchObsCnt[PatchIndex]=PatchObsCnt\
[PatchIndex]+1;
                                        Inserted=1;
                                } else {
                                        PatchObsCnt[PatchIndex]=0;
                                }
                        }
                }
                if (Inserted == 0) {
                        printf ("%s@1 ", $0);
                }
                next;
        }
        END {
                for(PatchIndex in PatchObs) {
                        if ( PatchObsCnt[PatchIndex] != 0) {
                                printf("%s@%d ", PatchObs[PatchIndex], \
PatchObsCnt[PatchIndex]);
                        }
                }
                printf("\n");
        } ' >> $1
        # Clear the parameter since it has already been used.
        echo "Obsoletes_label=" >> $1
```

```
        # Pass it's value on to the preinstall under another name
        echo "ACTIVE_OBSOLETES=$ACTIVE_OBSOLETES" >> $1
fi

#
# Construct PATCH_INFO line for this package.
#

tmpRequire=`nawk -F= ' $1 ~ /REQUIR/ { print $2 } ' $INFO_DIR/pkginfo `
tmpIncompat=`nawk -F= ' $1 ~ /INCOMPAT/ { print $2 } ' $INFO_DIR/pkginfo `

if [ -n "$tmpRequire" ] && [ -n "$tmpIncompat" ]
then
        echo "PATCH_INFO_$Patch_label=Installed: `date` From: `uname -n` \
          Obsoletes: $ACTIVE_OBSOLETES Requires: $tmpRequire \
          Incompatibles: $tmpIncompat" >> $1
elif [ -n "$tmpRequire" ]
then
        echo "PATCH_INFO_$Patch_label=Installed: `date` From: `uname -n` \
          Obsoletes: $ACTIVE_OBSOLETES Requires: $tmpRequire \
Incompatibles: " >> $1
elif [ -n "$tmpIncompat" ]
then
        echo "PATCH_INFO_$Patch_label=Installed: `date` From: `uname -n` \
          Obsoletes: $ACTIVE_OBSOLETES Requires: Incompatibles: \
$tmpIncompat" >> $1
else
        echo "PATCH_INFO_$Patch_label=Installed: `date` From: `uname -n` \
          Obsoletes: $ACTIVE_OBSOLETES Requires: Incompatibles: " >> $1
fi

#
# Since this script is called from the delivery medium and we may be using
# dot extensions to distinguish the different patch packages, this is the
# only place we can, with certainty, trace that source for our backout
# scripts. (Usually $INST_DATADIR would get us there).
#
echo "SCRIPTS_DIR=`dirname $0`" >> $1

# If additional operations are required for this package, place
# those package-specific commands here.

#XXXSpecial_CommandsXXX#

exit 0
```

preinstall スクリプト

preinstall スクリプトは、構成されるバックアウトパッケージの prototype ファイル、情報ファイル、およびインストールスクリプトの初期化を行います。このスクリプトは非常に単純で、この例の残りのスクリプトではバックアウトパッケージに通常のファイルを記述することしかできません。

バックアウトパッケージのシンボリックリンク、ハードリンク、デバイス、および名前付きパイプを復元する場合、pkgproto コマンドを使用するように preinstall ス

クリプトを変更して、提供される pkgmap ファイルをインストールされたファイルと比較できます。その後、バックアウトパッケージの変更されないファイルごとに prototype ファイルエントリが作成されます。使用する方法は、クラスアクションスクリプトの方法と同様です。

スクリプト patch_checkinstall および patch_postinstall は、preinstall スクリプトからパッケージソースツリーに挿入されます。これら 2 つのスクリプトにより、パッチが取り消されます。

```
# This script initializes the backout data for a patch package
# directory format options.
#
#       @(#)preinstall 1.5 96/05/10 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

PATH=/usr/sadm/bin:$PATH
recovery="no"

if [ "$PKG_INSTALL_ROOT" = "/" ]; then
        PKG_INSTALL_ROOT=""
fi

# Check to see if this is a patch installation retry.
if [ "$INTERRUPTION" = "yes" ]; then
    if [ -d "$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST" ] || [ -d \
"$PATCH_BUILD_DIR/$Patch_label.$PKGINST" ]; then
        recovery="yes"
    fi
fi

if [ -n "$PATCH_BUILD_DIR" -a -d "$PATCH_BUILD_DIR" ]; then
        BUILD_DIR="$PATCH_BUILD_DIR/$Patch_label.$PKGINST"
else
        BUILD_DIR="$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST"
fi

FILE_DIR=$BUILD_DIR/files
RELOC_DIR=$BUILD_DIR/files/reloc
ROOT_DIR=$BUILD_DIR/files/root
PROTO_FILE=$BUILD_DIR/prototype
PKGINFO_FILE=$BUILD_DIR/pkginfo
THIS_DIR=‘dirname $0‘

if [ "$PATCH_PROGRESSIVE" = "true" ]; then
        # If this is being used in an old-style patch, insert
        # the old-style script commands here.

        #XXXOld_CommandsXXX#

        exit 0
fi

#
```

```
# Unless specifically denied, initialize the backout patch data by
# creating the build directory and copying over the original pkginfo
# which pkgadd saved in case it had to be restored.
#
if [ "$PATCH_NO_UNDO" != "true" ] && [ "$recovery" = "no" ]; then
        if [ -d $BUILD_DIR ]; then
                rm -r $BUILD_DIR
        fi

        # If this is a retry of the same patch then recovery is set to
        # yes. Which means there is a build directory already in
        # place with the correct backout data.

        if [ "$recovery" = "no" ]; then
                mkdir $BUILD_DIR
                mkdir -p $RELOC_DIR
                mkdir $ROOT_DIR
        fi

        #
        # Here we initialize the backout pkginfo file by first
        # copying over the old pkginfo file and themn adding the
        # ACTIVE_PATCH parameter so the backout will know what patch
        # it's backing out.
        #
        # NOTE : Within the installation, pkgparam returns the
        # original data.
        #
        pkgparam -v $PKGINST | nawk '
                $1 ~ /PATCHLIST/        { next; }
                $1 ~ /PATCH_OBSOLETES/  { next; }
                $1 ~ /ACTIVE_OBSOLETES/ { next; }
                $1 ~ /Obsoletes_label/  { next; }
                $1 ~ /ACTIVE_PATCH/     { next; }
                $1 ~ /Patch_label/      { next; }
                $1 ~ /UPDATE/   { next; }
                $1 ~ /SCRIPTS_DIR/      { next; }
                $1 ~ /PATCH_NO_UNDO/    { next; }
                $1 ~ /INSTDATE/ { next; }
                $1 ~ /PKGINST/  { next; }
                $1 ~ /OAMBASE/  { next; }
                $1 ~ /PATH/     { next; }
                { print; } ' > $PKGINFO_FILE
        echo "ACTIVE_PATCH=$Patch_label" >> $PKGINFO_FILE
        echo "ACTIVE_OBSOLETES=$ACTIVE_OBSOLETES" >> $PKGINFO_FILE

        # And now initialize the backout prototype file with the
        # pkginfo file just formulated.
        echo "i pkginfo" > $PROTO_FILE

        # Copy over the backout scripts including the undo class
        # action scripts
        for script in $SCRIPTS_DIR/*; do
                srcscript=`basename $script`
                targscript=`echo $srcscript | nawk '
                        { script=$0; }
                        /u\./ {
                                sub("u.", "i.", script);
                                print script;
```

```
                                next;
                        }
                        /patch_/ {
                                sub("patch_", "", script);
                                print script;
                                next;
                        }
                        { print "dont_use" } ’‘
                if [ "$targscript" = "dont_use" ]; then
                        continue
                fi

                echo "i $targscript=$FILE_DIR/$targscript" >> $PROTO_FILE
                cp $SCRIPTS_DIR/$srcscript $FILE_DIR/$targscript
        done
         #
        # Now add entries to the prototype file that won’t be passed to
        # class action scripts. If the entry is brand new, add it to the
        # deletes file for the backout package.
        #
        Our_Pkgmap=‘dirname $SCRIPTS_DIR‘/pkgmap
        BO_Deletes=$FILE_DIR/deletes

        nawk -v basedir=${BASEDIR:-/} ’
                BEGIN { count=0; }
                {
                        token = $2;
                        ftype = $1;
                }
                $1 ~ /[#\!:]/ { next; }
                $1 ~ /[0123456789]/ {
                        if ( NF >= 3) {
                                token = $3;
                                ftype = $2;
                        } else {
                                next;
                        }
                }
                { if (ftype == "i" || ftype == "e" || ftype == "f" || ftype == \
"v" || ftype == "d") { next; } }
                {
                        equals=match($4, "=")-1;
                        if ( equals == -1 ) { print $3, $4; }
                        else { print $3, substr($4, 0, equals); }
                }
                ’ < $Our_Pkgmap | while read class path; do
                        #
                        # NOTE: If pkgproto is passed a file that is
                        # actually a hard link to another file, it
                        # will return ftype "f" because the first link
                        # in the list (consisting of only one file) is
                        # viewed by pkgproto as the source and always
                        # gets ftype "f".
                        #
                        # If this isn’t replacing something, then it
                        # just goes to the deletes list.
                        #
                        if valpath -l $path; then
                                Chk_Path="$BASEDIR/$path"
```

```
                                Build_Path="$RELOC_DIR/$path"
                                Proto_From="$BASEDIR"
                        else    # It's an absolute path
                                Chk_Path="$PKG_INSTALL_ROOT$path"
                                Build_Path="$ROOT_DIR$path"
                                Proto_From="$PKG_INSTALL_ROOT"
                        fi
                         #
                        # Hard links have to be restored as regular files.
                        # Unlike the others in this group, an actual
                        # object will be required for the pkgmk.
                        #
                        if [ -f "$Chk_Path" ]; then
                                mkdir -p `dirname $Build_Path`
                                cp $Chk_Path $Build_Path
                                cd $Proto_From
                                pkgproto -c $class "$Build_Path=$path" 1>> \
$PROTO_FILE 2> /dev/null
                                cd $THIS_DIR
                        elif [ -h "$Chk_Path" -o \
                             -c "$Chk_Path" -o \
                             -b "$Chk_Path" -o \
                             -p "$Chk_Path" ]; then
                                pkgproto -c $class "$Chk_Path=$path" 1>> \
$PROTO_FILE 2> /dev/null
                        else
                                echo $path >> $BO_Deletes
                        fi
                done
fi

# If additional operations are required for this package, place
# those package-specific commands here.

#XXXSpecial_CommandsXXX#

exit 0
```

クラスアクションスクリプト

クラスアクションスクリプトにより、既存のファイルを置き換えるファイルごとにコピーを作成して、バックアウトパッケージの prototype ファイルに対応する行を追加します。この処理はすべて、非常に単純な nawk スクリプトを使用して実行します。クラスアクションスクリプトは、対応するインストール済みファイルに一致しない通常のファイルからなるソースとターゲットのペアの一覧を受け取ります。シンボリックリンクおよびその他のファイル以外のものは、preinstall スクリプトで対応する必要があります。

```
# This class action script copies the files being replaced
# into a package being constructed in $BUILD_DIR. This class
# action script is only appropriate for regular files that
# are installed by simply copying them into place.
#
# For special package objects such as editable files, the patch
```

```
# producer must supply appropriate class action scripts.
#
# directory format options.
#
#       @(#)i.script 1.6 96/05/10 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

PATH=/usr/sadm/bin:$PATH

ECHO="/usr/bin/echo"
SED="/usr/bin/sed"
PKGPROTO="/usr/bin/pkgproto"
EXPR="/usr/bin/expr"    # used by dirname
MKDIR="/usr/bin/mkdir"
CP="/usr/bin/cp"
RM="/usr/bin/rm"
MV="/usr/bin/mv"

recovery="no"
Pn=$$
procIdCtr=0

CMDS_USED="$ECHO $SED $PKGPROTO $EXPR $MKDIR $CP $RM $MV"
LIBS_USED=""

if [ "$PKG_INSTALL_ROOT" = "/" ]; then
        PKG_INSTALL_ROOT=""
fi

# Check to see if this is a patch installation retry.
if [ "$INTERRUPTION" = "yes" ]; then
        if [ -d "$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST" ] ||
\
[ -d "$PATCH_BUILD_DIR/$Patch_label.$PKGINST" ]; then
                recovery="yes"
        fi
fi

if [ -n "$PATCH_BUILD_DIR" -a -d "$PATCH_BUILD_DIR" ]; then
        BUILD_DIR="$PATCH_BUILD_DIR/$Patch_label.$PKGINST"
else
        BUILD_DIR="$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST"
fi

FILE_DIR=$BUILD_DIR/files
RELOC_DIR=$FILE_DIR/reloc
ROOT_DIR=$FILE_DIR/root
BO_Deletes=$FILE_DIR/deletes
PROGNAME=‘basename $0‘

if [ "$PATCH_PROGRESSIVE" = "true" ]; then
        PATCH_NO_UNDO="true"
fi

# Since this is generic, figure out the class.
Class=‘echo $PROGNAME | nawk ’ { print substr($0, 3)  }’‘
```

```
# Since this is an update, $BASEDIR is guaranteed to be correct
BD=${BASEDIR:-/}

cd $BD

#
# First, figure out the dynamic libraries that can trip us up.
#
if [ -z "$PKG_INSTALL_ROOT" ]; then
        if [ -x /usr/bin/ldd ]; then
                LIB_LIST=`/usr/bin/ldd $CMDS_USED | sort -u | nawk '
                        $1 ~ /\// { continue; }
                        { printf "%s ", $3 } ''
        else
                LIB_LIST="/usr/lib/libc.so.1 /usr/lib/libdl.so.1
\
/usr/lib/libw.so.1 /usr/lib/libintl.so.1 /usr/lib/libadm.so.1 \
/usr/lib/libelf.so.1"
        fi
fi

#
# Now read the list of files in this class to be replaced. If the file
# is already in place, then this is a change and we need to copy it
# over to the build directory if undo is allowed. If it's a new entry
# (No $dst), then it goes in the deletes file for the backout package.
#
procIdCtr=0
while read src dst; do
        if [ -z "$PKG_INSTALL_ROOT" ]; then
                Chk_Path=$dst
                for library in $LIB_LIST; do
                        if [ $Chk_Path = $library ]; then
                                $CP $dst $dst.$Pn
                                LIBS_USED="$LIBS_USED $dst.$Pn"
                                LD_PRELOAD="$LIBS_USED"
                                export LD_PRELOAD
                        fi
                done
        fi

        if [ "$PATCH_PROGRESSIVE" = "true" ]; then
                # If this is being used in an old-style patch, insert
                # the old-style script commands here.

                #XXXOld_CommandsXXX#
                echo >/dev/null # dummy
        fi

        if [ "${PATCH_NO_UNDO}" != "true" ]; then
                #
                # Here we construct the path to the appropriate source
                # tree for the build. First we try to strip BASEDIR. If
                # there's no BASEDIR in the path, we presume that it is
                # absolute and construct the target as an absolute path
                # by stripping PKG_INSTALL_ROOT. FS_Path is the path to
                # the file on the file system (for deletion purposes).
                # Build_Path is the path to the object in the build
```

```
                # environment.
                #
                if [ "$BD" = "/" ]; then
                        FS_Path=‘$ECHO $dst | $SED s@"$BD"@@‘
                else
                        FS_Path=‘$ECHO $dst | $SED s@"$BD/"@@‘
                fi

                # If it’s an absolute path the attempt to strip the
                # BASEDIR will have failed.
                if [ $dst = $FS_Path ]; then
                        if [ -z "$PKG_INSTALL_ROOT" ]; then
                                FS_Path=$dst
                                Build_Path="$ROOT_DIR$dst"
                        else
                                Build_Path="$ROOT_DIR‘echo $dst | \
                                    sed s@"$PKG_INSTALL_ROOT"@@‘"
                                FS_Path=‘echo $dst | \
                                    sed s@"$PKG_INSTALL_ROOT"@@‘
                        fi
                else
                        Build_Path="$RELOC_DIR/$FS_Path"
                fi

                if [ -f $dst ]; then    # If this is replacing something
                        cd $FILE_DIR
                        #
                        # Construct the prototype file entry. We replace
                        # the pointer to the filesystem object with the
                        # build directory object.
                        #
                        $PKGPROTO -c $Class $dst=$FS_Path | \
                            $SED -e s@=$dst@=$Build_Path@ >> \
                            $BUILD_DIR/prototype

                        # Now copy over the file
                        if [ "$recovery" = "no" ]; then
                                DirName=‘dirname $Build_Path‘
                                $MKDIR -p $DirName
                                $CP -p $dst $Build_Path
                        else
                                # If this file is already in the build area skip it
                                if [ -f "$Build_Path" ]; then
                                        cd $BD
                                        continue
                                else
                                        DirName=‘dirname $Build_Path‘
                                        if [ ! -d "$DirName" ]; then
                                                $MKDIR -p $DirName
                                        fi
                                        $CP -p $dst $Build_Path
                                fi
                        fi

                        cd $BD
                else    # It’s brand new
                        $ECHO $FS_Path >> $BO_Deletes
                fi
        fi
```

```
        # If special processing is required for each src/dst pair,
        # add that here.
        #
        #XXXSpecial_CommandsXXX#
        #

        $CP $src $dst.$$$procIdCtr
        if [ $? -ne 0 ]; then
                $RM $dst.$$$procIdCtr 1>/dev/null 2>&1
        else
                $MV -f $dst.$$$procIdCtr $dst
                for library in $LIB_LIST; do
                        if [ "$library" = "$dst" ]; then
                                LD_PRELOAD="$dst"
                                export LD_PRELOAD
                        fi
                done
        fi
        procIdCtr=‘expr $procIdCtr + 1‘
done

# If additional operations are required for this package, place
# those package-specific commands here.

#XXXSpecial_CommandsXXX#

#
# Release the dynamic libraries
#
for library in $LIBS_USED; do
        $RM -f $library
done

exit 0
```

postinstall スクリプト

postinstallスクリプトにより、その他のスクリプトで提供される情報を使用して、バックアウトパッケージを作成します。pkgmk コマンドおよび pkgtrans コマンドではパッケージデータベースは必要ではないため、パッケージのインストール内で実行できます。

例では、(PKGSAV 環境変数を使用して) 保存ディレクトリにストリーム形式パッケージを作成することで、パッチの取り消しが可能になります。明白ではありませんが、pkgadd 処理の際に保存ディレクトリが移動されるため、このパッケージはストリーム形式にする必要があります。パッケージ自体の保存ディレクトリで pkgadd コマンドをパッケージに適用する場合、任意の時点でのパッケージソースの場所に関する想定の信頼性が非常に低くなります。ストリーム形式のパッケージは一時ディレクトリに展開され、一時ディレクトリからインストールされます。ディレクトリ形式のパッケージでは、保存ディレクトリからインストールが開始され、pkgadd 復旧処理時に突然再配置されます。

パッケージに適用するパッチを決定するには、次のコマンドを使用します。

```
$ pkgparam SUNWstuf PATCHLIST
```

Sun の公開インタフェースである PATCHLIST を除いて、この例のパラメータ名には意味はありません。PATCH の代わりに、従来の SUNW_PATCHID および PATCH_EXCL などのほかの一覧を使用して、それに合うように PATCH_REQD の名前を変更することができます。

特定のパッチパッケージが、同じ媒体から利用できるほかのパッチパッケージに依存している場合、checkinstall スクリプトでこれを決定して、アップグレードの例 (176 ページの「パッケージのアップグレード」を参照) と同じ方法で postinstall スクリプトから実行されるスクリプトを作成できます。

```
# This script creates the backout package for a patch package
#
# directory format options.
#
# @(#) postinstall 1.6 96/01/29 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

# Description:
#       Set the TYPE parameter for the remote file
#
# Parameters:
#       none
#
# Globals set:
#       TYPE

set_TYPE_parameter () {
        if [ ${PATCH_UNDO_ARCHIVE:?????} = "/dev" ]; then
                # handle device specific stuff
                TYPE="removable"
        else
                TYPE="filesystem"
        fi
}

#
# Description:
#       Build the remote file that points to the backout data
#
# Parameters:
#       $1:     the un/compressed undo archive
#
# Globals set:
#       UNDO, STATE

build_remote_file () {
        remote_path=$PKGSAV/$Patch_label/remote
        set_TYPE_parameter
```

```
        STATE="active"

        if [ $1 = "undo" ]; then
                UNDO="undo"
        else
                UNDO="undo.Z"
        fi

        cat > $remote_path << EOF
# Backout data stored remotely
TYPE=$TYPE
FIND_AT=$ARCHIVE_DIR/$UNDO
STATE=$STATE
EOF
}

PATH=/usr/sadm/bin:$PATH

if [ "$PKG_INSTALL_ROOT" = "/" ]; then
        PKG_INSTALL_ROOT=""
fi

if [ -n "$PATCH_BUILD_DIR" -a -d "$PATCH_BUILD_DIR" ]; then
        BUILD_DIR="$PATCH_BUILD_DIR/$Patch_label.$PKGINST"
else
        BUILD_DIR="$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST"
fi

if [ ! -n "$PATCH_UNDO_ARCHIVE" ]; then
        PATCH_UNDO_ARCHIVE="none"
fi

FILE_DIR=$BUILD_DIR/files
RELOC_DIR=$FILE_DIR/reloc
ROOT_DIR=$FILE_DIR/root
BO_Deletes=$FILE_DIR/deletes
THIS_DIR=‘dirname $0‘
PROTO_FILE=$BUILD_DIR/prototype
TEMP_REMOTE=$PKGSAV/$Patch_label/temp

if [ "$PATCH_PROGRESSIVE" = "true" ]; then
        # remove the scripts that are left behind
        install_scripts=‘dirname $0‘
        rm $install_scripts/checkinstall \
$install_scripts/patch_checkinstall $install_scripts/patch_postinstall

        # If this is being used in an old-style patch, insert
        # the old-style script commands here.

        #XXXOld_CommandsXXX#

        exit 0
fi
#
# At this point we either have a deletes file or we don’t. If we do,
# we create a prototype entry.
#
if [ -f $BO_Deletes ]; then
        echo "i deletes=$BO_Deletes" >> $BUILD_DIR/prototype
```

```
fi

#
# Now delete everything in the deletes list after transferring
# the file to the backout package and the entry to the prototype
# file. Remember that the pkgmap will get the CLIENT_BASEDIR path
# but we have to actually get at it using the BASEDIR path. Also
# remember that removef will import our PKG_INSTALL_ROOT
#
Our_Deletes=$THIS_DIR/deletes
if [ -f $Our_Deletes ]; then
        cd $BASEDIR

        cat $Our_Deletes | while read path; do
                Reg_File=0

                if valpath -l $path; then
                        Client_Path="$CLIENT_BASEDIR/$path"
                        Build_Path="$RELOC_DIR/$path"
                        Proto_Path=$BASEDIR/$path
                else    # It's an absolute path
                        Client_Path=$path
                        Build_Path="$ROOT_DIR$path"
                        Proto_Path=$PKG_INSTALL_ROOT$path
                fi

                # Note: If the file isn't really there, pkgproto
                # doesn't write anything.
                LINE=`pkgproto $Proto_Path=$path`
                ftype=`echo $LINE | nawk '{ print $1 }'`
                if [ $ftype = "f" ]; then
                        Reg_File=1
                fi

                if [ $Reg_File = 1 ]; then
                        # Add source file to the prototype entry
                        if [ "$Proto_Path" = "$path" ]; then
                                LINE=`echo $LINE | sed -e s@$Proto_Path@$Build_Path@2`
                        else
                                LINE=`echo $LINE | sed -e s@$Proto_Path@$Build_Path@`
                        fi

                        DirName=`dirname $Build_Path`
                        # make room in the build tree
                        mkdir -p $DirName
                        cp -p $Proto_Path $Build_Path
                fi

                # Insert it into the prototype file
                echo $LINE 1>>$PROTO_FILE 2>/dev/null

                # Remove the file only if it's OK'd by removef
                rm `removef $PKGINST $Client_Path` 1>/dev/null 2>&1
        done
        removef -f $PKGINST

        rm $Our_Deletes
fi
```

```
#
# Unless specifically denied, make the backout package.
#
if [ "$PATCH_NO_UNDO" != "true" ]; then
        cd $BUILD_DIR   # We have to build from here.

        if [ "$PATCH_UNDO_ARCHIVE" != "none" ]; then
                STAGE_DIR="$PATCH_UNDO_ARCHIVE"
                ARCHIVE_DIR="$PATCH_UNDO_ARCHIVE/$Patch_label/$PKGINST"
                mkdir -p $ARCHIVE_DIR
                mkdir -p $PKGSAV/$Patch_label
        else
                if [ -d $PKGSAV/$Patch_label ]; then
                        rm -r $PKGSAV/$Patch_label
                fi
                STAGE_DIR=$PKGSAV
                ARCHIVE_DIR=$PKGSAV/$Patch_label
                mkdir $ARCHIVE_DIR
        fi

        pkgmk -o -d $STAGE_DIR 1>/dev/null 2>&1
        pkgtrans -s $STAGE_DIR $ARCHIVE_DIR/undo $PKG 1>/dev/null 2>&1
        compress $ARCHIVE_DIR/undo
        retcode=$?
        if [ "$PATCH_UNDO_ARCHIVE" != "none" ]; then
                if [ $retcode != 0 ]; then
                        build_remote_file "undo"
                else
                        build_remote_file "undo.Z"
                fi
        fi
        rm -r $STAGE_DIR/$PKG

        cd ..
        rm -r $BUILD_DIR
        # remove the scripts that are left behind
        install_scripts=‘dirname $0‘
        rm $install_scripts/checkinstall $install_scripts/patch_\
checkinstall $install_scripts/patch_postinstall
fi

#
# Since this apparently worked, we’ll mark as obsoleted the prior
# versions of this patch - installpatch deals with explicit obsoletions.
#
cd ${PKG_INSTALL_ROOT:-/}
cd var/sadm/pkg

active_base=‘echo $Patch_label | nawk ’
        { print substr($0, 1, match($0, "Patchvers_pfx")-1) } ’‘

List=‘ls -d $PKGINST/save/${active_base}*‘
if [ $? -ne 0 ]; then
        List=""
fi

for savedir in $List; do
        patch=‘basename $savedir‘
        if [ $patch = $Patch_label ]; then
```

```
                break
        fi

        # If we get here then the previous patch gets deleted
        if [ -f $savedir/undo ]; then
                mv $savedir/undo $savedir/obsolete
                echo $Patch_label >> $savedir/obsoleted_by
        elif [ -f $savedir/undo.Z ]; then
                mv $savedir/undo.Z $savedir/obsolete.Z
                echo $Patch_label >> $savedir/obsoleted_by
        elif  [ -f $savedir/remote ]; then
                ‘grep . $PKGSAV/$patch/remote | sed ’s/STATE=.*/STATE=obsolete/
’ > $TEMP_REMOTE‘
                rm -f $PKGSAV/$patch/remote
                mv $TEMP_REMOTE $PKGSAV/$patch/remote
                rm -f $TEMP_REMOTE
                echo $Patch_label >> $savedir/obsoleted_by
        elif  [ -f $savedir/obsolete -o -f $savedir/obsolete.Z ]; then
                echo $Patch_label >> $savedir/obsoleted_by
        fi
done

# If additional operations are required for this package, place
# those package-specific commands here.

#XXXSpecial_CommandsXXX#

exit 0
```

patch_checkinstall スクリプト

```
# checkinstall script to validate backing out a patch.
# directory format option.
#
#       @(#)patch_checkinstall 1.2 95/10/10 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

PATH=/usr/sadm/bin:$PATH

LATER_MSG="PaTcH_MsG 6 ERROR: A later version of this patch is applied."
NOPATCH_MSG="PaTcH_MsG 2 ERROR: Patch number $ACTIVE_PATCH is not installed"
NEW_LIST=""

# Get OLDLIST
. $1

#
# Confirm that the patch that got us here is the latest one installed on
# the system and remove it from PATCHLIST.
#
Is_Inst=0
Skip=0
active_base=‘echo $ACTIVE_PATCH | nawk ’
```

```
        { print substr($0, 1, match($0, "Patchvers_pfx")-1) } ’‘
active_inst=‘echo $ACTIVE_PATCH | nawk ’
        { print substr($0, match($0, "Patchvers_pfx")+1) } ’‘
for patchappl in ${OLDLIST}; do
        appl_base=‘echo $patchappl | nawk ’
                { print substr($0, 1, match($0, "Patchvers_pfx")-1) } ’‘
        if [ $appl_base = $active_base ]; then
                appl_inst=‘echo $patchappl | nawk ’
                        { print substr($0, match($0, "Patchvers_pfx")+1) } ’‘
                result=‘expr $appl_inst \> $active_inst‘
                if [ $result -eq 1 ]; then
                        puttext "$LATER_MSG"
                        exit 3
                elif [ $appl_inst = $active_inst ]; then
                        Is_Inst=1
                        Skip=1
                fi
        fi
        if [ $Skip = 1 ]; then
                Skip=0
        else
                NEW_LIST="${NEW_LIST} $patchappl"
        fi
done

if [ $Is_Inst = 0 ]; then
        puttext "$NOPATCH_MSG"
        exit 3
fi

#
# OK, all’s well. Now condition the key variables.
#
echo "PATCHLIST=${NEW_LIST}" >> $1
echo "Patch_label=" >> $1
echo "PATCH_INFO_$ACTIVE_PATCH=backed out" >> $1

# Get the current PATCH_OBSOLETES and condition it
Old_Obsoletes=$PATCH_OBSOLETES

echo $ACTIVE_OBSOLETES | sed ’y/\ /\n/’ | \
nawk -v PatchObsList="$Old_Obsoletes" ’
        BEGIN {
                printf("PATCH_OBSOLETES=");
                PatchCount=split(PatchObsList, PatchObsComp, " ");

                for(PatchIndex in PatchObsComp) {
                        Atisat=match(PatchObsComp[PatchIndex], "@");
                        PatchObs[PatchIndex]=substr(PatchObsComp[PatchIndex], \
0, Atisat-1);
                        PatchObsCnt[PatchIndex]=substr(PatchObsComp\
[PatchIndex], Atisat+1);
                }
        }
        {
                for(PatchIndex in PatchObs) {
                        if (PatchObs[PatchIndex] == $0) {
                                PatchObsCnt[PatchIndex]=PatchObsCnt[PatchIndex]-1;
                        }
```

```
                }
                next;
        }
        END {
                for(PatchIndex in PatchObs) {
                        if ( PatchObsCnt[PatchIndex] > 0 ) {
                                printf("%s@%d ", PatchObs[PatchIndex], PatchObsCnt\
[PatchIndex]);
                        }
                }
                printf("\n");
        } ' >> $1

        # remove the used parameters
        echo "ACTIVE_OBSOLETES=" >> $1
        echo "Obsoletes_label=" >> $1

exit 0
```

patch_postinstall スクリプト

```
# This script deletes the used backout data for a patch package
# and removes the deletes file entries.
#
# directory format options.
#
#       @(#)patch_postinstall 1.2 96/01/29 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#
PATH=/usr/sadm/bin:$PATH
THIS_DIR=`dirname $0`

Our_Deletes=$THIS_DIR/deletes

#
# Delete the used backout data
#
if [ -f $Our_Deletes ]; then
        cat $Our_Deletes | while read path; do
                if valpath -l $path; then
                        Client_Path=`echo "$CLIENT_BASEDIR/$path" | sed s@//@/@`
                else    # It's an absolute path
                        Client_Path=$path
                fi
                rm `removef $PKGINST $Client_Path`
        done
        removef -f $PKGINST

        rm $Our_Deletes
fi

#
# Remove the deletes file, checkinstall and the postinstall
#
```

```
rm -r $PKGSAV/$ACTIVE_PATCH
rm -f $THIS_DIR/checkinstall $THIS_DIR/postinstall

exit 0
```

パッケージのアップグレード

パッケージのアップグレードのプロセスは、パッケージの上書きのプロセスとは異なっています。Oracle Solaris OS の一部として提供される標準パッケージのアップグレードをサポートする特殊ツールはありますが、アンバンドルのパッケージは独自のアップグレードをサポートするように設計できます。前のいくつかの例で、先を見越して、管理者の管理下でインストールの正確な方法を制御するパッケージについて説明しました。同様に、パッケージの直接アップグレードをサポートする `request` スクリプトを設計できます。管理者が、古いファイルが残らないように、あるパッケージをすべて置き換えて別のパッケージをインストールすることにした場合、パッケージスクリプトでこの処理を行うことができます。

この例の `request` スクリプトおよび `postinstall` スクリプトでは、単純なアップグレード可能パッケージを提供しています。`request` スクリプトは管理者と対話して、古いパッケージインスタンスを削除するための簡単なファイルを `/tmp` ディレクトリに設定します。`request` スクリプトはファイルを作成します (これは禁止されている) が、`/tmp` には全員がアクセスできるため、問題ありません。

`postinstall` スクリプトは、`/tmp` のシェルスクリプトを実行します。このスクリプトは、古いパッケージに対して必要な `pkgrm` コマンドを実行してから、自分自身を削除します。

この例では基本的なアップグレードについて説明します。いくつかの非常に長いメッセージを含めて、コードは 50 行未満です。設計者の要求により、アップグレードのバックアウトを拡張したり、パッケージにほかの大きな変換を行ったりすることができます。

アップグレードオプションのユーザーインタフェースの設計時には、管理者がプロセスについてはっきり認識して、並列インストールではなくアップグレードを積極的に要求していることを確実に確認する必要があります。ユーザーインタフェースによって操作が明確になっている限りは、アップグレードのような汎用的で複雑な処理を実行しても問題はありません。

`request` スクリプト

```
# request script
control an upgrade installation

PATH=/usr/sadm/bin:$PATH
UPGR_SCRIPT=/tmp/upgr.$PKGINST
```

```
UPGRADE_MSG="Do you want to upgrade the installed version ?"

UPGRADE_HLP="If upgrade is desired, the existing version of the \
    package will be replaced by this version. If it is not \
    desired, this new version will be installed into a different \
    base directory and both versions will be usable."

UPGRADE_NOTICE="Conflict approval questions may be displayed. The \
    listed files are the ones that will be upgraded. Please \
    answer \"y\" to these questions if they are presented."

pkginfo -v 1.0 -q SUNWstuf.\*

if [ $? -eq 0 ]; then
      # See if upgrade is desired here
      response=‘ckyorn -p "$UPGRADE_MSG" -h "$UPGRADE_HLP"‘
      if [ $response = "y" ]; then
            OldPkg=‘pkginfo -v 1.0 -x SUNWstuf.\* | nawk ’ \
            /SUNW/{print $1} ’‘
            # Initiate upgrade
            echo "PATH=/usr/sadm/bin:$PATH" > $UPGR_SCRIPT
            echo "sleep 3" >> $UPGR_SCRIPT
            echo "echo Now removing old instance of $PKG" >> \
            $UPGR_SCRIPT
            if [ ${PKG_INSTALL_ROOT} ]; then
                  echo "pkgrm -n -R $PKG_INSTALL_ROOT $OldPkg" >> \
                  $UPGR_SCRIPT
            else
                  echo "pkgrm -n $OldPkg" >> $UPGR_SCRIPT
            fi
            echo "rm $UPGR_SCRIPT" >> $UPGR_SCRIPT
            echo "exit $?" >> $UPGR_SCRIPT

            # Get the original package’s base directory
            OldBD=‘pkgparam $OldPkg BASEDIR‘
            echo "BASEDIR=$OldBD" > $1
            puttext -l 5 "$UPGRADE_NOTICE"
       else
             if [ -f $UPGR_SCRIPT ]; then
                   rm -r $UPGR_SCRIPT
             fi
       fi
fi

exit 0
```

postinstall スクリプト

```
# postinstall
to execute a simple upgrade

PATH=/usr/sadm/bin:$PATH
UPGR_SCRIPT=/tmp/upgr.$PKGINST

if [ -f $UPGR_SCRIPT ]; then
```

```
      sh $UPGR_SCRIPT &
fi

exit 0
```

クラスアーカイブパッケージの作成

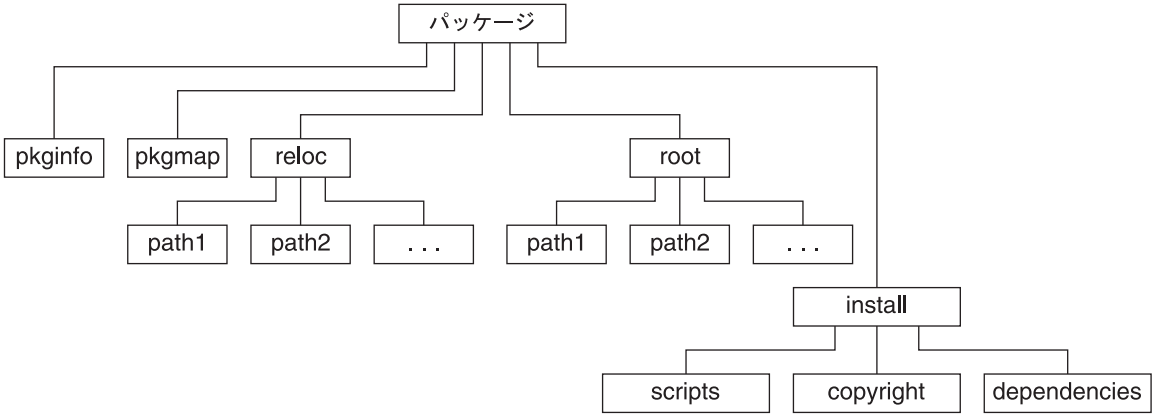
クラスアーカイブパッケージは、アプリケーションバイナリインタフェース (ABI) の拡張機能です。これは、特定のファイルセットが単一のファイル (アーカイブ) に結合され、オプションで圧縮または暗号化されたパッケージです。クラスアーカイブ形式は、アクティブな可能性があるファイルシステムに対して、最初のインストールの速度を最大 30% 向上し、パッケージおよびパッチのインストール時の信頼性が改善されます。

次のセクションでは、アーカイブパッケージディレクトリの構造、キーワード、および `faspac` ユーティリティーについて説明します。

アーカイブパッケージディレクトリの構造

次の図に示すパッケージエントリは、パッケージファイルが含まれるディレクトリを表します。このディレクトリは、パッケージと同じ名前にする必要があります。

図 6–1 パッケージディレクトリ構造



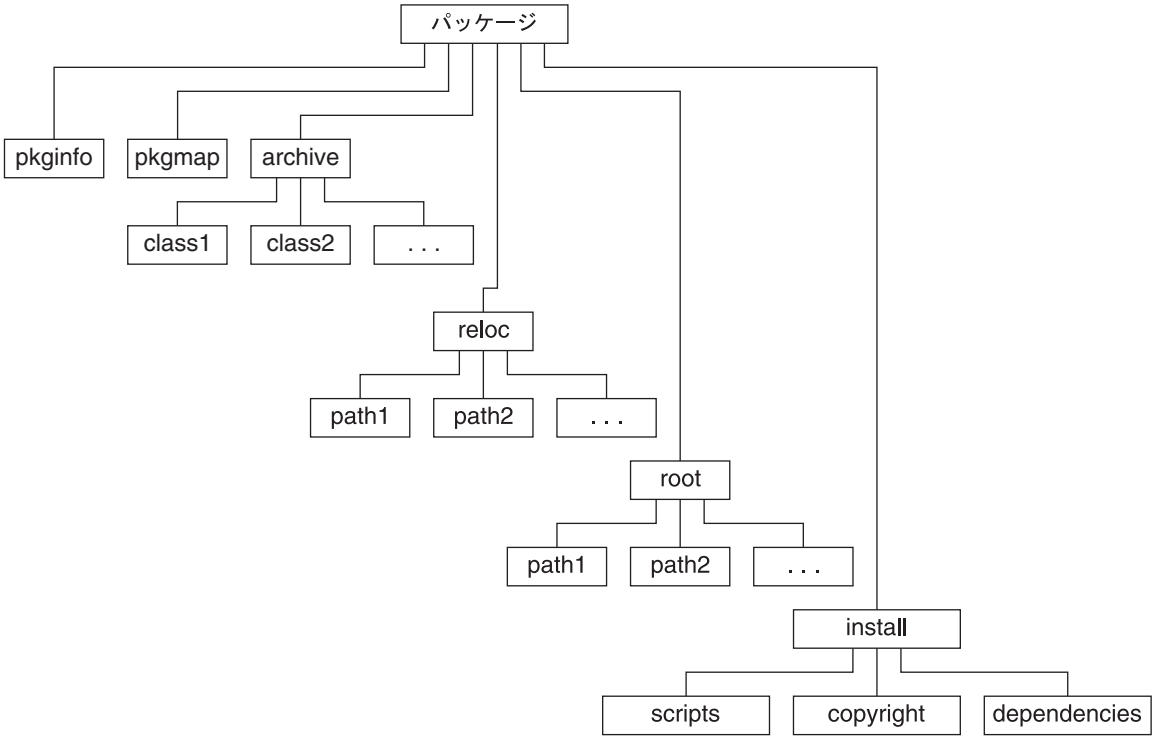
パッケージディレクトリに含まれるファイルおよびディレクトリの機能を次に示します。

項目	説明
pkginfo	特別な環境変数およびインストール命令を含めて、パッケージ全体を記述するファイル
pkgmap	インストールされる各オブジェクト (ファイル、ディレクトリ、パイプなど) を記述するファイル
reloc	ベースディレクトリから相対的にインストールされるファイル (再配置可能オブジェクト) が含まれる、オプションのディレクトリ
root	root ディレクトリから相対的にインストールされるファイル (ルートオブジェクト) が含まれる、オプションのディレクトリ
install	スクリプトおよびその他の補助的なファイルが含まれる、オプションのディレクトリ (pkginfo および pkgmap を除き、すべての ftype i ファイルはここに含まれる)

クラスアーカイブ形式では、パッケージビルダにより reloc および root の各ディレクトリからアーカイブにファイルを結合できます。アーカイブは、インストールの速度の向上、パッケージサイズの縮小、またはパッケージのセキュリティの向上のために、圧縮、暗号化、またはその他の任意の方法で処理できます。

ABI により、パッケージ内の任意のファイルをクラスに割り当てることができます。特定のクラス内のすべてのファイルは、クラスアクションスクリプトで定義されるカスタムメソッドを使用してディスクにインストールできます。このカスタムメソッドでは、ターゲットシステムで使用できるプログラムまたはパッケージとともに提供されるプログラムを使用できます。結果の形式は、標準 ABI 形式によく似ています。次の図に示すように、別のディレクトリが追加されます。アーカイブ用ファイルのすべてのクラスは、そのまま単一のファイルに結合され、archive ディレクトリに配置されます。アーカイブされたすべてのファイルが reloc および root の各ディレクトリから削除され、インストールクラスアクションスクリプトが install ディレクトリに配置されます。

図 6–2 アーカイブパッケージディレクトリの構造



クラスアーカイブパッケージをサポートするキーワード

この新しいクラスアーカイブ形式をサポートするために、pkginfo ファイル内で特殊な意味を持つキーワード形式の新しいインタフェースが 3 つあります。これらのキーワードを使用して、特殊な処理が必要なクラスを指定します。各キーワード文の形式は、 keyword=class1[class2 class3 ...] です。各キーワードの値は、次の表で定義されています。

キーワード	説明
PKG_SRC_NOVERIFY	これは、提供されるパッケージの reloc または root の各ディレクトリ内のファイルが指定されたクラスに属する場合に、pkgadd でファイルの存在とプロパティーを確認しないように指定します。アーカイブされたクラスではファイルが reloc または root のディレクトリ内に存在しないため、このキーワードが必要です。それらのファイルは、archive ディレクトリの非公開形式のファイルです。

キーワード	説明
PKG_DST_QKVERIFY	これらのクラスのファイルは、インストール後にテキスト出力がほとんどまたはまったくない簡易アルゴリズムを使用して確認されます。簡易確認では、最初に各ファイルの属性を正確に設定して、処理が正常に実行されたかどうかを確認します。pkgmap に対して、ファイルサイズと変更時間のテストも行われます。checksum 検証は実行されず、エラー復旧は標準の検証メカニズムのエラー復旧よりも非力です。インストール時に停電またはディスク障害が発生した場合には、コンテンツファイルがインストールされたファイルと一致しないことがあります。この不一致は、どのような場合でも pkgrm で解決できます。
PKG_CAS_PASSRELATIVE	通常、インストールクラスアクションスクリプトは、インストールするファイルを指定するソースとターゲットのペアの一覧を、stdin から受け取ります。PKG_CAS_PASSRELATIVE に割り当てられたクラスでは、ソースとターゲットのペアが取得されません。その代わりに単一の一覧を受け取ります。この一覧の最初のエントリはソースパッケージの場所で、一覧の残りはターゲットのパスです。これはアーカイブからの抽出を単純化するためです。ソースパッケージの場所から archive ディレクトリのアーカイブを検索できます。ターゲットのパスは、アーカイブのコンテンツを抽出する機能に渡されます。各ターゲットパスは、元のパスが root または reloc のどちらであったかに応じて、絶対パスまたはベースディレクトリからの相対パスのいずれかで指定されます。このオプションを選択すると、相対パスと絶対パスの両方を単一のクラスに結合することは困難な場合があります。

アーカイブされたクラスごとにクラスアクションスクリプトが必要です。これは Bourne シェルコマンドが含まれたファイルで、アーカイブから実際にファイルをインストールするために pkgadd から実行されます。パッケージの install ディレクトリにクラスアクションスクリプトが検出された場合、pkgadd からそのスクリプトにインストールのすべての責任が委任されます。クラスアクションスクリプトは root 権限で実行され、ターゲットシステムの任意の場所にファイルを配置できます。

注 – クラスアーカイブパッケージを実装するために絶対に必要なキーワードは、PKG_SRC_NOVERIFY だけです。ほかのキーワードは、インストールの速度を向上したりコードを保護したりするために使用できます。

faspac ユーティリティー

faspac ユーティリティーは、標準 ABI パッケージを、バンドル版のパッケージで使用されるクラスアーカイブ形式に変換します。このユーティリティーでは、cpio を使用してアーカイブを行い、compress を使用して圧縮を行います。結果のパッケージには、最上位のディレクトリに archive というディレクトリが追加されています。このディレクトリには、クラスごとに名前が付けられたすべてのアーカイブが含まれます。install ディレクトリには、各アーカイブの展開に必要なクラスアクションスクリプトが含まれます。絶対パスはアーカイブされません。

faspac ユーティリティーの形式は次のとおりです。

```
faspac [-m Archive Method] -a -s -q [-d Base Directory] /
[-x Exclude List] [List of Packages]
```

次の表で、faspac コマンドの各オプションについて説明します。

オプション	説明
-m アーカイブメソッド	アーカイブまたは圧縮の方法を示します。bzip2 はデフォルトで使用される圧縮ユーティリティーです。zip または unzip に切り替えるには、-m zip を使用します。cpio または compress の場合は、-m cpio を使用します。
-a	属性を固定します (ルートで実行する必要がある)。
-s	標準 ABI 型のパッケージ変換を示します。このオプションは、cpio または圧縮されたパッケージを取り、標準 ABI 準拠のパッケージ形式にします。
-q	非出力モードを示します。
-d ベースディレクトリ	コマンド行で要求される場合にすべてのパッケージの基準になるディレクトリを示します。これは *List of Packages* エントリとは相互排他的です。
-x 除外リスト	処理から除外するパッケージの、コンマ区切りまたは引用符付きスペース区切りの一覧を示します。
パッケージの一覧	処理されるパッケージの一覧を示します。

用語集

ABI
「Application Binary Interface, ABI (アプリケーションバイナリインタフェース)」を参照してください。

Abstract Syntax Notation 1
抽象オブジェクトを表現する手段。たとえば、ASN.1 では、公開鍵証明書、証明書を構成するすべてのオブジェクト、オブジェクトの収集順序などが定義されています。ただし、ASN.1 では、オブジェクトを保存用または転送用に直列化する方法は定義されていません。

ASN.1
「Abstract Syntax Notation 1, ASN.1 (抽象構文記法 1)」を参照してください。

class action script
(クラスアクションスクリプト**)**
パッケージオブジェクトのグループに対して実行される一連のアクションを定義しているファイル。

collectively relocatable object
(集合的再配置可能オブジェクト**)**
共通のインストールベースに対応して配置されるパッケージオブジェクト。「ベースディレクトリ」も参照してください。

common name
(共通名**)**
署名付きパッケージのパッケージキーストアでリストされている別名。

composite package
(複合パッケージ**)**
再配置可能なパス名と絶対パス名の両方を含むパッケージ。

`compver` file
(compver ファイル**)**
パッケージの下位互換性を指定する手段。

control file
(制御ファイル**)**
パッケージをインストールする方法、場所、条件を制御するファイル。「情報ファイル」および「インストールスクリプト」を参照してください。

copyright
(著作権**)**
ソフトウェア、ソースコード、ドキュメントなどの知的財産を所有および販売するための権利。SunSoft またはほかの組織が著作権を所有しているかどうか、所有権を CD-ROM および挿入文で表記する必要があります。著作権の所有は、SunSoft のドキュメントでも承認されます。

depend file
(depend ファイル**)**
パッケージの基本的な依存関係を解決する手段。「`compver` ファイル」も参照してください。

DER
「Distinguished Encoding Rules, DER (識別符号化規則)」を参照してください。

Distinguished Encoding Rules (DER)
ASN.1 オブジェクトのバイナリ表現であり、コンピュータ環境で保存用または転送用に ASN.1 オブジェクトを直列化する方法が定義されています。署名付きパッケージで使用されます。

install time
(インストール時)
パッケージが pkgadd コマンドでインストールされている間の時間。

install variable
(インストール変数)
大文字で始まり、インストール時に評価される変数。

ITU-T Recommendation X.509
広く採用されている X.509 公開鍵証明書構文を指定するプロトコル。

parametric path name
(パラメータ型パス名)
変数の指定を含むパス名。

PEM
「Privacy Enhanced Message, PEM (プライバシー強化メール)」を参照してください。

PKCS12
「Public Key Cyptography Standard #12, PKCS12 (公開鍵暗号化標準 #12)」を参照してください。

PKCS7
「Public Key Cryptography Standard #7, PKCS7 (公開鍵暗号化標準 #7)」を参照してください。

prerequisite package
(必須パッケージ)
別のパッケージの存在に依存するパッケージ。「depend ファイル」も参照してください。

Privacy Enhanced Message, PEM
(プライバシー強化メール)
Base 64 エンコーディングとオプションのヘッダーを使用してファイルをエンコードする方法。証明書と非公開鍵をファイルシステム上または電子メールメッセージ内のファイルに符号化するために広く使用されています。

procedure script
(手続きスクリプト)
パッケージのインストールと削除の間の特定のポイントで発生するアクションを定義するスクリプト。

Public Key Cryptography Standard #12
暗号化オブジェクトをディスクに格納するための構文を定めた規格。パッケージのキーストアは、この形式で保持されます。

Public Key Cryptography Standard #7
デジタル署名やデジタル封筒などの暗号化が適用された可能性があるデータに対する汎用的な構文を定めた規格。署名付きパッケージには、埋め込まれた PKCS7 署名が含まれます。

relocatable object
(再配置可能オブジェクト)
ターゲットシステム上に絶対パスの場所を必要としないパッケージオブジェクト。その代わりに、パッケージオブジェクトの場所は、インストールプロセスの間に決定されます。「集合的再配置可能オブジェクト」および「個別再配置可能オブジェクト」も参照してください。

reverse dependency
(逆依存)
対象のパッケージの存在に別のパッケージが依存する状態。「depend ファイル」も参照してください。

segmented
(分割)
フロッピーディスクなどの単一のボリュームに収まらないパッケージ。

tar	Tape Archive Retrieval。メディアにファイルを追加したり、メディアからファイルを抽出したりするための Oracle Solaris コマンド。
user key (ユーザー鍵)	暗号鍵に関する機密情報を保持する鍵。この情報は、不正な使用を防ぐために、セキュリティーが施された形式で格納されます。署名付きパッケージを作成するときに使用されます。
X.509	「ITU-T Recommendation X.509」を参照してください。
アプリケーションバイナリインタフェース	コンパイル済みアプリケーションとアプリケーションが動作するオペレーティングシステム間のバイナリシステムインタフェースの定義。
インストールスクリプト	パッケージのカスタマイズしたインストール手順を提供できるスクリプト。
クラス	パッケージオブジェクトのグループ化に使用される名前。「クラスアクションスクリプト」も参照してください。
公開鍵	暗号化鍵として生成される値。公開鍵から派生する非公開鍵と組み合わせることで、メッセージとデジタル署名を効果的に暗号化するために使用できます。
構築時	パッケージが `pkgmk` コマンドで構築されている期間。
構築変数	小文字で始まり、構築時に評価される変数。
個別再配置可能オブジェクト	集合的再配置可能オブジェクトと同じディレクトリ位置に制限されないパッケージオブジェクト。`prototype` ファイルの *path* フィールドのインストール変数を使用して定義し、インストール場所は `request` スクリプトまたは `checkinstall` スクリプトで指定します。
再配置可能	`prototype` ファイルで相対パス名により定義されているパッケージオブジェクト。
情報ファイル	パッケージの依存関係を定義し、著作権メッセージを提供し、ターゲットシステム上に領域を確保できるファイル。
署名されていないパッケージ	暗号化またはデジタル署名されていない通常の ABI パッケージ。
署名付きパッケージ	デジタル署名の付いた通常のストリーム形式パッケージ。これは、次の点を確認します。そのパッケージに署名したエンティティーがそのパッケージの作成者であること、そのエンティティーが実際にそのパッケージに署名したこと、そのパッケージがエンティティーによる署名後に変更されていないこと、そのパッケージに署名したエンティティーが信頼されたエンティティーであること。
信頼できる証明書	別のエンティティーに属する単一の公開鍵証明書を含む証明書。信頼できる証明書は、デジタル署名を検証するとき、およびセキュリティー保護されたサーバー (SSL サーバー) への接続を確立するときに使用されます。
デジタル署名	パッケージの整合性とセキュリティを確認するために使用されるエンコードされたメッセージ。

認証局 **(CA)**	パッケージの署名に使用される証明書を発行する Verisign などの機関。
パッケージ	ソフトウェアアプリケーションに必要なファイルとディレクトリの集合。
パッケージインスタンス	パッケージのバリエーション。パッケージの `pkginfo` ファイルの `PKG`、`ARCH`、および `VERSION` パラメータの定義の組み合わせによって決定されます。
パッケージオブジェクト	ターゲットシステムにインストールされるパッケージに含まれるアプリケーションファイルの別の呼び方。
パッケージキーストア	パッケージツールを使用して照会できる証明書と鍵のリポジトリ。
パッケージ識別子	`pkgadd` コマンドによってパッケージの省略名に追加される数値接尾辞。
パッケージの省略名	`pkginfo` ファイルの `PKG` パラメータで定義されているパッケージの短い名前。
パッチ一覧	現在のパッケージに影響するパッチの一覧。このパッチの一覧は、`pkginfo` ファイルでインストールされるパッケージに記録されます。
非公開鍵	秘密のメッセージを交換する当事者のみが知っている暗号化および復号化のための鍵。署名付きパッケージを作成するために公開鍵とともに使用されます。
非互換パッケージ	特定のパッケージと互換性のないパッケージ。「`depend` ファイル」も参照してください。
ベースディレクトリ	再配置可能オブジェクトがインストールされる場所。`pkginfo` ファイルで `BASEDIR` パラメータを使用して定義されます。

索引

O

P

R

S

あ

い

お

か

き