Guide du développeur pour l'empaquetage d'applications

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

Référence : 820–5496–12
Avril 2009

090508@22510

Table des matières

Préface

Le *Guide du développeur pour l'empaquetage d'applications* contient les instructions étape par étape et les informations de base correspondantes pour la conception, la construction et la vérification de packages. Ce manuel inclut également des techniques avancées qui peuvent s'avérer utiles au cours de la procédure de création des packages.

Remarque – Cette version de Solaris™ prend en charge les systèmes utilisant les architectures de processeur SPARC® et x86 : UltraSPARC®, SPARC64, AMD64, Pentium et Xeon EM64T. Les systèmes pris en charge sont répertoriés dans les *listes de compatibilité matérielle de Solaris* disponibles à l'adresse `http://www.sun.com/bigadmin/hcl`. Ce document présente les différences d'implémentation en fonction des divers types de plates-formes.

Dans ce document, les termes relatifs à x86 suivants ont la signification suivante :

- “x86” désigne la famille des produits compatibles x86 64 bits et 32 bits.
- “x64” désigne des informations 64 bits spécifiques relatives aux systèmes AMD64 ou EM64T.
- “x86 32 bits” désigne des informations 32 bits spécifiques relatives aux systèmes x86.

Pour connaître les systèmes pris en charge, reportez-vous aux *listes de compatibilité matérielle de Solaris*.

Utilisateurs de ce manuel

Ce manuel s'adresse aux développeurs d'applications chargés de la conception et de la création des packages.

Bien que la majeure partie du manuel s'adresse aux développeurs de packages novices, il contient également des informations que les développeurs plus expérimentés trouveront utiles.

Organisation de ce document

Le tableau suivant décrit les chapitres de ce document.

Nom du chapitre	Description du chapitre
Chapitre 1, “Conception d'un package”	Décrit les composants d'un package et les critères de conception d'un package. Décrit également les commandes, fichiers et scripts associés.
Chapitre 2, “Création d'un package”	Décrit la procédure et les opérations obligatoires de la création d'un package. Fournit également des instructions détaillées pour chaque opération.
Chapitre 3, “Amélioration de la fonctionnalité d'un package (opérations)”	Fournit des instructions détaillées pour l'ajout de fonctions facultatives à un package.
Chapitre 4, “Vérification et transfert d'un package”	Décrit les procédures de vérification de l'intégrité d'un package et de transfert d'un package sur un support de distribution.
Chapitre 5, “Création d'un package : Études de cas”	Fournit des études de cas relatives à la création de packages.
Chapitre 6, “Techniques avancées de création de packages”	Fournit des techniques avancées de création de packages.
Glossaire	Définit les termes employés dans ce manuel.

Documentation connexe

La documentation suivante, disponible auprès des libraires, peut être une source d'information de base supplémentaire sur la création des packages du système V.

- *System V Application Binary Interface*
- *System V Application Binary Interface - SPARC Processor Supplement*
- *System V Application Binary Interface - Intel386 Processor Supplement*

Documentation, support et formation

Le site Web Sun fournit des informations sur les ressources supplémentaires suivantes :

- documentation (`http://www.sun.com/documentation/`);
- support (`http://www.sun.com/support/`);
- formation (`http://www.sun.com/training/`).

Sun attend vos commentaires

Afin d'améliorer sa documentation, Sun vous encourage à faire des commentaires et à apporter des suggestions. Pour nous faire part de vos commentaires, accédez au site `http://docs.sun.com`, puis cliquez sur votre avis.

Conventions typographiques

Le tableau ci-dessous décrit les conventions typographiques utilisées dans ce manuel.

TABLEAU P–1 Conventions typographiques

Type de caractères	Signification	Exemple
`AaBbCc123`	Noms des commandes, fichiers et répertoires, ainsi que messages système.	Modifiez votre fichier `.login`. Utilisez `ls -a` pour afficher la liste de tous les fichiers. `nom_machine% Vous avez reçu du courrier.`
`AaBbCc123`	Ce que vous entrez, par opposition à ce qui s'affiche à l'écran.	`nom_machine%` **`su`** `Mot de passe :`
aabbcc123	Paramètre fictif : à remplacer par un nom ou une valeur réel(le).	La commande permettant de supprimer un fichier est `rm` *nom_fichier*.
AaBbCc123	Titres de manuel, nouveaux termes et termes importants.	Reportez-vous au chapitre 6 du *Guide de l'utilisateur*. Un *cache* est une copie des éléments stockés localement. *N'enregistrez pas* le fichier. **Remarque :** En ligne, certains éléments mis en valeur s'affichent en gras.

Invites de shell dans les exemples de commandes

Le tableau suivant présente les invites système et les invites de superutilisateur UNIX® par défaut des C shell, Bourne shell et Korn shell.

TABLEAU P–2 Invites de shell

Shell	Invite
C shell	`nom_machine%`
C shell pour superutilisateur	`nom_machine#`
Bourne shell et Korn shell	`$`
Bourne shell et Korn shell pour superutilisateur	`#`

◆ ◆ ◆ CHAPITRE 1

Conception d'un package

Avant de créer un package, vous devez déterminer les fichiers à créer et les commandes à exécuter. Vous devez également réfléchir à la configuration logicielle requise de l'application et aux besoins de votre clientèle. Vos clients sont les administrateurs chargés d'installer le package. Ce chapitre traite des fichiers, commandes et critères à connaître et à prendre en considération avant de commencer à créer un package.

La liste suivante répertorie les informations disponibles dans le présent chapitre :

Instructions relatives à la conception d'un package

Servez-vous de ces listes de tâches pour trouver des instructions détaillées sur la création et la vérification d'un package.

Package - De quoi s'agit-il ?

Une application logicielle est livrée sous forme d'unités appelées *packages*. Un package est un ensemble de fichiers et de répertoires requis par un produit logiciel. Un package est habituellement conçu et créé par le développeur d'applications après le développement du code de l'application. Chaque produit logiciel doit être créé sous forme d'un ou plusieurs packages afin de faciliter son transfert sur un support de distribution. Le produit logiciel peut ensuite être produit en série et installé par les administrateurs.

Un package est un ensemble de fichiers et de répertoires d'un format donné. Ce format se conforme à l'ABI (Application Binary Interface) qui complète la définition d'interface du système V.

Composants d'un package

Les composants d'un package se répartissent en deux catégories.

- Les *objets du package* correspondent aux fichiers d'application à installer.
- Les *fichiers de contrôle* contrôlent la méthode, l'emplacement et la décision de l'installation.

Les fichiers de contrôle se répartissent également en deux catégories : les *fichiers d'information* et les *scripts d'installation*. Certains fichiers de contrôle sont obligatoires. D'autres sont facultatifs.

Pour créer le package d'une application, vous devez au préalable créer les composants obligatoires ainsi que tout élément facultatif entrant dans la composition du package. Vous pouvez ensuite créer le package à l'aide de la commande `pkgmk`.

Pour créer un package, vous devez fournir les éléments suivants :

- Objets du package (fichiers et répertoires de l'application logicielle)
- Deux fichiers d'information obligatoires (les fichiers `pkginfo` et `prototype`)
- Fichiers d'information facultatifs
- Scripts d'installation facultatifs

La figure suivante illustre le contenu d'un package.

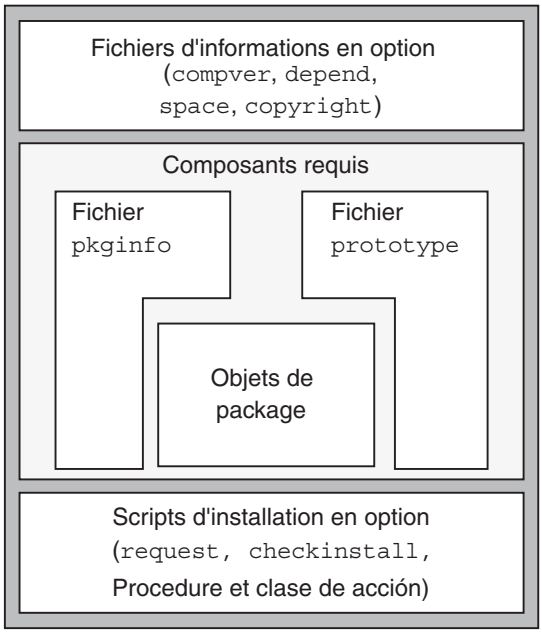


FIGURE 1–1 Contenu d'un package

Composants obligatoires d'un package

Vous devez créer les composants suivants avant de créer un package :

- Objets du package

  Ces composants entrent dans la composition de l'application. Il peut s'agir des éléments suivants :

  - Fichiers (fichiers exécutables ou fichiers de données)
  - Répertoires
  - Tubes nommés
  - Liens
  - Périphériques

- Fichier `pkginfo`

  Le fichier `pkginfo` est un fichier d'information de package obligatoire qui définit les valeurs des paramètres. Les valeurs des paramètres incluent le nom abrégé du package, le nom complet du package et l'architecture du package. Pour plus d'informations, reportez-vous à "Création d'un fichier `pkginfo`" à la page 26 et à la page de manuel `pkginfo(4)`.

Remarque – Deux pages de manuel `pkginfo(1)` sont disponibles. La première décrit une commande de section 1 qui affiche des informations sur les packages installés. La deuxième décrit un fichier de section 4 qui répertorie les caractéristiques d'un package. Lorsque vous accédez aux pages de manuel, n'oubliez pas d'indiquer la section applicable à la page de manuel. Exemple : `man -s 4 pkginfo`.

- Fichier `prototype`

  Le fichier `prototype` est un fichier d'information obligatoire de package qui répertorie les composants d'un package. Chaque objet, fichier d'information et script d'installation d'un package dispose d'une entrée. Chaque entrée se compose de plusieurs champs d'information qui décrivent le composant en question, y compris son emplacement, ses attributs et le type de fichier. Pour plus d'informations, reportez-vous à "Création d'un fichier `prototype`" à la page 31 et à la page de manuel `prototype(4)`.

Composants facultatifs d'un package

Fichiers d'information d'un package

Vous pouvez inclure quatre fichiers d'information facultatifs dans votre package :

- Fichier `compver`

  Ce fichier indique les versions précédentes du package qui sont compatibles avec la version actuelle de votre package.

- Fichier `depend`

  Ce fichier répertorie les packages qui disposent d'un lien particulier avec votre package.

- Fichier `space`

  Ce fichier indique l'espace disque requis sur l'environnement cible, outre l'espace requis par les objets définis dans le fichier `prototype`. De l'espace supplémentaire peut par exemple être nécessaire pour des fichiers créés dynamiquement lors de l'installation.

- Fichier `copyright`

  Ce fichier contient le texte du message de copyright qui s'affiche lors de l'installation du package.

Chaque fichier d'information du package doit avoir une entrée dans le fichier `prototype`. Reportez-vous à "Création de fichiers d'information" à la page 52 pour plus d'informations sur la création de ce type de fichier.

Scripts d'installation d'un package

Les scripts d'installation ne sont pas obligatoires. Vous pouvez cependant inclure des scripts qui effectuent des opérations personnalisées lors de l'installation de votre package. Un script d'installation dispose des caractéristiques suivantes :

- Le script se compose de commandes Bourne shell.
- Les droits d'accès au fichier du script doivent être définis sur 0644.
- Le script ne doit pas nécessairement contenir l'identificateur de shell (`#! /bin/sh`).

Les quatre types de script sont les suivants :

- Script `request`

  Le script `request` demande à l'administrateur qui installe le package de saisir des informations.
- Script `checkinstall`

  Le script `checkinstall` procède à certains contrôles du système de fichiers.

  Remarque – Le script `checkinstall` n'est disponible que pour la version 2.5 de Solaris™ et les versions compatibles.

- Scripts de procédure

  Les scripts de procédure définissent les actions qui se déroulent à des moments donnés de l'installation ou de la désinstallation d'un package. Vous pouvez créer quatre scripts de procédure avec les noms prédéfinis suivants : `preinstall`, `postinstall`, `preremove` et `postremove`.
- Scripts d'action de classe

  Les *scripts d'action de classe* définissent une série d'opérations à effectuer sur un groupe d'objets.

Reportez-vous à "Création de scripts d'installation" à la page 59 pour plus d'informations sur les scripts d'installation.

Critères à prendre en considération avant de créer un package

Avant de créer un package, vous devez décider si votre produit doit être composé d'un ou plusieurs packages. Notez que l'installation de plusieurs packages de petite taille est en général plus longue que l'installation d'un seul package de grande taille. Bien que créer un seul package soit une bonne idée, cela n'est pas toujours possible. Si vous décidez de créer plusieurs packages, vous devez déterminer la segmentation du code de l'application. La présente section répertorie une liste de critères à utiliser pour planifier la création d'un package.

Bon nombre des critères de création d'un package ont des éléments en commun. Satisfaire à tous les critères de manière égale est souvent difficile. Ces critères sont présentés par ordre d'importance. Toutefois, cet ordre n'est qu'indicatif et peut être adapté à chaque circonstance. Bien que chaque critère soit important, leur optimisation afin de produire un ensemble de packages de qualité vous revient.

Pour voir d'autres idées de conception, reportez-vous au Chapitre 6, “Techniques avancées de création de packages”.

Packages pouvant être installés à distance

Tous les packages doivent pouvoir être *installés à distance*. Pouvoir être installé à distance signifie que l'administrateur chargé d'installer votre package a la possibilité de tenter son installation sur un système client et non pas nécessairement sur le système de fichiers root (`/`) sur lequel la commande `pkgadd` est exécutée.

Optimisation pour les configurations client-serveur

Prenez en compte les divers types de configurations logicielles système (par exemple, un système autonome et un serveur) lors de la création des packages. Une bonne conception de packages sépare les fichiers concernés afin d'optimiser l'installation de chaque type de configuration. Par exemple, le contenu des systèmes de fichiers root (`/`) et `/usr` doit être segmenté afin de pouvoir aisément prendre en charge les configurations serveur.

Package conçu d'après des limites fonctionnelles

Les packages doivent être autonomes et facilement identifiés d'après un groupe de fonctions. Par exemple, un package contenant UFS doit contenir toutes les fonctions UFS et être limité aux programmes binaires UFS.

Les packages doivent être organisés du point de vue du client en unités fonctionnelles.

Package en fonction des redevances

Placez le code requérant par contrat le versement d'une redevance dans un package ou groupe packages distinct. Ne répartissez pas le code dans des packages superflus.

Package en fonction des dépendances système

Placez les programmes binaires qui dépendent du système dans des packages distincts. Par exemple, le code du noyau doit être placé dans un package distinct, chaque architecture d'implémentation correspondant à un package différent. Cette règle s'applique également aux programmes binaires de diverses architectures. Par exemple, les programmes binaires d'un système SPARC peuvent être contenus dans un package et ceux d'un système x86, dans un autre.

Fonctions spécifiques à chaque package

Évitez autant que possible de créer des fichiers dupliqués lors de la création de vos packages. Toute duplication inutile se traduit par des problèmes de prise en charge et de version. Si votre produit se compose de plusieurs packages, comparez le contenu des packages régulièrement pour éviter de conserver des fichiers dupliqués.

Package en fonction de la localisation

Les éléments spécifiques à la localisation doivent se trouver dans un package à part. Le modèle idéal de création de packages d'un produit à localiser se compose d'un package par version localisée. Il arrive malheureusement que les limites organisationnelles entrent en conflit avec les critères des limites fonctionnelles et des limites du produit.

Les paramètres internationaux par défaut peuvent également être fournis dans un package distinct. Cette conception isole les fichiers concernés par des modifications à apporter au niveau de la localisation et standardise le format de livraison des packages de localisation.

Packages Image Packaging System (IPS)

Ce document traite des packages SVR4. Pour une livraison dans le système d'exploitation OpenSolaris, optez pour des packages IPS. Le système d'exploitation OpenSolaris prend en charge les packages SVR4 et IPS. Le logiciel IPS dialogue avec les référentiels réseau et utilise le système de fichiers ZFS. Dans le système d'exploitation OpenSolaris, il est possible de publier des packages SVR4 existants dans un référentiel IPS à l'aide de la commande `pkgsend(1)`.

Le tableau suivant permet de comparer les commandes des systèmes de d'empaquetage SVR4 et IPS. Pour de plus amples informations sur IPS, voir Getting Started With the Image Packaging System (`http://dlc.sun.com/osol/docs/content/IPS/ggcph.html`).

TABLEAU 1–1 Conception d'un package : IPS et SVR4

Tâche	Commande IPS	Commande SVR4
Installation d'un nouveau package	`pkg install`	`pkgadd -a`
Affichage des informations sur l'état d'un package	`pkg list`	`pkginfo`
Vérification de l'installation d'un package	`pkg verify`	`pkgchk -v`
Afficher des informations sur un package	`pkg info`	`pkginfo -l`
Listage du contenu d'un package	`pkg contents`	`pkgchk -l`
Désinstallation d'un package	`pkg uninstall`	`pkgrm`

Commandes, fichiers et scripts de conception d'un package

La section suivante décrit les commandes, fichiers et scripts que vous pouvez être amené à utiliser lors de la manipulation de packages. Ces éléments sont décrits dans les pages de manuel et détaillés tout au long de ce guide dans le cadre de l'opération qu'ils effectuent.

Le tableau suivant répertorie les commandes qui permettent de créer, vérifier et installer un package, ainsi que d'obtenir des informations sur celui-ci.

TABLEAU 1–2 Commandes de conception d'un package

Tâche	Commande / Page de manuel	Description	Pour plus d'informations
Création de packages	`pkgproto`(1)	Génère un fichier `prototype` à saisir dans la commande `pkgmk`.	"Exemple : Création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 38
`pkgmk`(1)	Crée un package à installer.	"Création d'un package" à la page 45	
Installation, suppression et transfert de packages	`pkgadd`(1M)	Installe un package logiciel sur un système.	"Installation de packages logiciels" à la page 90
`pkgask`(1M)	Stocke les réponses à un script `request`.	"Règles de conception pour les scripts `request`" à la page 65	
`pkgtrans`(1)	Copie des packages sur un support de distribution.	"Transfert d'un package sur un support distribution" à la page 101	

TABLEAU 1–2 Commandes de conception d'un package *(Suite)*

Tâche	Commande / Page de manuel	Description	Pour plus d'informations
	`pkgrm`(1M)	Supprime un package d'un système.	"Suppression d'un package" à la page 100
Obtention d'informations sur des packages	`pkgchk`(1M)	Vérifie l'intégrité d'un package logiciel.	"Vérification de l'intégrité d'un package" à la page 92
	`pkginfo`(1)	Affiche des informations sur un package logiciel.	"Commande `pkginfo`" à la page 96
	`pkgparam`(1)	Affiche les valeurs des paramètres d'un package.	"Commande `pkgparam`" à la page 94
Modification des packages installés	`installf`(1M)	Incorpore un nouvel objet de package dans un package déjà installé.	"Règles de conception des scripts de procédure" à la page 70 et Chapitre 5, "Création d'un package : Études de cas"
	`removef`(1M)	Supprime un objet de package d'un package déjà installé.	"Règles de conception des scripts de procédure" à la page 70

Le tableau suivant répertorie les fichiers d'information permettant de créer un package.

TABLEAU 1–3 Fichiers d'information d'un package

Fichier	Description	Pour plus d'informations
`admin`(4)	Fichier des paramètres par défaut de l'installation d'un package	"Fichier de valeurs d'administration par défaut" à la page 132
`compver`(4)	Fichier de compatibilité d'un package	"Définition des dépendances d'un package" à la page 53
`copyright`(4)	Fichier d'information sur le copyright d'un package	"Rédaction d'un message de copyright" à la page 55
`depend`(4)	Fichier des dépendances d'un package	"Définition des dépendances d'un package" à la page 53
`pkginfo`(4)	Fichier des caractéristiques d'un package	"Création d'un fichier `pkginfo`" à la page 26
`pkgmap`(4)	Fichier de description du contenu d'un package	"Fichier `pkgmap`" à la page 46
`prototype`(4)	Fichier d'information d'un package	"Création d'un fichier `prototype`" à la page 31

TABLEAU 1–3 Fichiers d'information d'un package *(Suite)*

Fichier	Description	Pour plus d'informations
space(4)	Fichier de l'espace disque requis d'un package	"Réservation d'espace supplémentaire sur un système cible" à la page 57

Le tableau suivant répertorie les scripts d'installation facultatifs disponibles pour influencer la décision d'installer ou non un package et le choix de la méthode d'installation.

TABLEAU 1–4 Scripts d'installation d'un package

Script	Description	Pour plus d'informations
`request`	Demande des informations au programme d'installation.	"Rédaction d'un script `request`" à la page 64
`checkinstall`	Recueille des données du système de fichiers.	"Recueil de données d'un système de fichiers à l'aide du script `checkinstall`" à la page 67
`preinstall`	Effectue toute opération d'installation personnalisée avant l'installation des classes.	"Rédaction de scripts de procédure" à la page 69
`postinstall`	Effectue toute opération d'installation personnalisée après l'installation de tous les volumes.	"Rédaction de scripts de procédure" à la page 69
`preremove`	Effectue toute opération de suppression personnalisée avant la suppression des classes.	"Rédaction de scripts de procédure" à la page 69
`postremove`	Effectue toute opération de suppression personnalisée après la suppression de toutes les classes.	"Rédaction de scripts de procédure" à la page 69
Action de classe	Effectue une série d'opérations sur un groupe d'objets spécifique.	"Rédaction de scripts d'action de classe" à la page 71

◆ ◆ ◆ CHAPITRE 2

Création d'un package

Le présent chapitre décrit la procédure et les opérations liées à la création d'un package. Certaines de ces opérations sont obligatoires. D'autres sont facultatives. Les opérations obligatoires sont traitées en détail dans ce chapitre. Pour plus d'informations sur les opérations facultatives qui permettent d'ajouter d'autres fonctions à votre package, reportez-vous au Chapitre 3, “Amélioration de la fonctionnalité d'un package (opérations)” et au Chapitre 6, “Techniques avancées de création de packages”.

La liste suivante répertorie les informations disponibles dans le présent chapitre :

Procédure de création d'un package (liste de tâches)

Le Tableau 2–1 décrit une procédure que vous pouvez suivre pour créer des packages, tout particulièrement si vous n'avez aucune expérience en la matière. Bien qu'il ne soit pas obligatoire d'effectuer les quatre premières opérations dans l'ordre indiqué, votre apprentissage sera plus aisé si vous vous y tenez. Avec de l'expérience, vous pourrez changer l'ordre de ces opérations à votre gré.

En tant que concepteur de packages expérimenté, vous pouvez automatiser la procédure de création des packages à l'aide de la commande `make` et créer des fichiers. Pour plus d'informations, reportez-vous à la page de manuel `make(1S)`.

TABLEAU 2–1 Procédure de création d'un package (liste de tâches)

Tâche	Description	Voir
1. Création d'un fichier `pkginfo`	Créez le fichier `pkginfo` pour décrire les caractéristiques du package.	"Procédure de création d'un fichier `pkginfo`" à la page 29
2. Organisation du contenu du package	Organisez les composants du package en une structure de répertoires hiérarchique.	"Organisation du contenu d'un package" à la page 30
3. (facultatif) Création de fichiers d'information	Définissez les dépendances du package, y compris le message de copyright, et réservez de l'espace supplémentaire sur le système cible.	Chapitre 3, "Amélioration de la fonctionnalité d'un package (opérations)"
4. (facultatif) Création de scripts d'installation	Personnalisez les procédures d'installation et de désinstallation du package.	Chapitre 3, "Amélioration de la fonctionnalité d'un package (opérations)"
5. Création d'un fichier `prototype`	Décrivez l'objet contenu dans le package dans un fichier `prototype`.	"Création d'un fichier `prototype`" à la page 31
6. Création du package	Créez le package à l'aide de la commande `pkgmk`.	"Création d'un package" à la page 45
7. Vérification et transfert du package	Vérifiez l'intégrité du package avant de le copier sur un support de distribution.	Chapitre 4, "Vérification et transfert d'un package"

Variables d'environnement d'un package

Vous pouvez utiliser des variables dans les fichiers d'information obligatoires, `pkginfo` et `prototype`. Vous pouvez également utiliser une option dans la commande `pkgmk` destinée à créer un package. Davantage d'informations contextuelles sur les variables sont fournies tout au long de ce chapitre dans les sections traitant de ces fichiers et commandes. Toutefois, avant de vous lancer dans la création de votre package, il est important de connaître les différents types de variables et de comprendre la manière dont ils influencent le bon déroulement de la création d'un package.

Il existe deux types de variables :

- Variables de création

  Les *variables de création* commencent par une minuscule et sont évaluées lors de la *phase de création*, pendant que le package est créé à l'aide de la commande `pkgmk`.

- Variables d'installation

  Les *variables d'installation* commencent par une majuscule et sont évaluées lors de la *phase d'installation*, pendant que le package est installé à l'aide de la commande `pkgadd`.

Règles globales d'utilisation des variables d'environnement

Dans le fichier `pkginfo`, la définition d'une variable utilise le format suivant : *PARAM=valeur*, où la première lettre de *PARAM* est une majuscule. Ces variables sont évaluées lors de la phase d'installation. Si une ou plusieurs de ces variables ne peuvent être évaluées, l'exécution de la commande `pkgadd` est suspendue et un message d'erreur est renvoyé.

Dans le fichier `prototype`, la définition d'une variable peut utiliser le format `!`*PARAM=valeur* ou le format `$`*variable*. *PARAM* et *variable* peuvent tous deux commencer par une majuscule ou une minuscule. Seules les variables dont les valeurs sont connues lors de la phase de création sont évaluées. Si *PARAM* ou *variable* est une variable de création ou d'installation dont la valeur est inconnue lors de la phase de création, l'exécution de la commande `pkgmk` est suspendue et un message d'erreur est renvoyé.

L'option *PARAM=valeur* peut aussi être incluse dans la commande `pkgmk`. Cette option fonctionne de la même manière que dans le fichier `prototype`, excepté qu'elle s'applique à l'ensemble du package. La définition de `!`*PARAM=valeur* d'un fichier `prototype` est locale par rapport à ce fichier et à la section du package à laquelle elle se rapporte.

Si *PARAM* est une variable d'installation et que *variable* est une variable d'installation ou de création dont la valeur est connue, la commande `pkgmk` insère la définition dans le fichier `pkginfo` pour que celle-ci soit disponible lors de la phase d'installation. Toutefois, la commande `pkgmk` n'évalue pas les variables *PARAM* qui sont utilisées dans des chemins spécifiés dans le fichier `prototype`.

Récapitulatif des variables d'environnement d'un package

Le tableau suivant récapitule les formats de spécification, l'emplacement et le champ d'application des variables.

TABLEAU 2–2 Récapitulatif des variables d'environnement d'un package

Emplacement de définition de la variable	Format de définition de la variable	Type de variable défini	Moment d'évaluation de la variable	Emplacement d'évaluation de la variable	Éléments que la variable peut remplacer
Fichier `pkginfo`	*PARAM=valeur*	Création	Ignorée lors de la phase de création	SO	Aucune
Installation	Phase d'installation	Dans le fichier `pkgmap`	*propriétaire*, *groupe*, *chemin* ou lien cible		

TABLEAU 2–2 Récapitulatif des variables d'environnement d'un package *(Suite)*

Emplacement de définition de la variable	Format de définition de la variable	Type de variable défini	Moment d'évaluation de la variable	Emplacement d'évaluation de la variable	Éléments que la variable peut remplacer
Fichier `prototype`	`!`*PARAM=valeur*	Création	Phase de création	Dans le fichier `prototype` et tout fichier inclus	*mode*, *propriétaire*, *groupe* ou *chemin*
Installation	Phase de création	Dans le fichier `prototype` et tout fichier inclus	Commandes `!search` et `!command` uniquement		
Ligne de commande `pkgmk`	*PARAM=valeur*	Création	Phase de création	Dans le fichier `prototype`	*mode*, *propriétaire*, *groupe* ou *chemin*
Installation	Phase de création	Dans le fichier `prototype`	Commande `!search` uniquement		
Phase d'installation	Dans le fichier `pkgmap`	*propriétaire*, *groupe*, *chemin* ou lien cible			

Création d'un fichier `pkginfo`

Le fichier `pkginfo` est un fichier ASCII qui décrit les caractéristiques d'un package, ainsi que les informations permettant de contrôler le déroulement de l'installation.

Chaque entrée du fichier `pkginfo` est une ligne décrivant la valeur d'un paramètre d'après le format *PARAM=valeur*. *PARAM* peut être tout paramètre standard décrit à la page de manuel `pkginfo(4)` Les paramètres sont spécifiés dans aucun ordre particulier.

Remarque – Chaque *valeur* peut être indiquée entre guillemets simples ou doubles (par exemple, '*valeur*' ou "*valeur*"). Si *valeur* contient des caractères considérés comme spéciaux dans le cadre d'un environnement shell, employez des guillemets. Les exemples et études de cas figurant dans ce guide n'utilisent pas de guillemets. Reportez-vous à la page de manuel `pkginfo(4)` pour voir un exemple contenant des guillemets doubles.

Vous pouvez également créer vos propres paramètres de package en leur attribuant une valeur dans le fichier `pkginfo`. Vos paramètres doivent commencer par une majuscule suivie d'autres majuscules ou de minuscules. Une majuscule indique que le paramètre (variable) est évalué lors de la phase d'installation, par opposition à la phase de création. Pour plus d'informations sur les différences entre les variables d'installation et les variables de création, reportez-vous à "Variables d'environnement d'un package" à la page 24.

Remarque – Tout espace situé à la fin d'une valeur de paramètre est ignoré.

Vous devez définir les cinq paramètres suivants dans un fichier pkginfo : PKG, NAME, ARCH, VERSION et CATEGORY. Les paramètres PATH, PKGINST et INSTDATE sont automatiquement insérés par le logiciel lors de la création du package. Ne modifiez pas ces huit paramètres. Pour plus d'informations sur les autres paramètres, reportez-vous à la page de manuel pkginfo(4).

Définition d'une instance de package

Un même package peut avoir diverses versions ou être compatible avec diverses architectures, voire les deux. Chaque variante d'un package est appelée une *instance de package*. Une instance de package se définit en alliant les définitions des paramètres PKG, ARCH et VERSION dans le fichier pkginfo.

La commande pkgadd attribue un *identificateur de package* à chaque instance de package lors de la phase d'installation. L'identificateur de package est composé de l'abréviation du package et d'un suffixe numérique, par exemple SUNWadm.2. Cet identificateur permet de distinguer les instances de package d'un package à l'autre mais aussi celles d'un même package.

Définition de l'abréviation d'un package (PKG)

L'*abréviation d'un package* est un nom abrégé de package défini par le paramètre PKG dans le fichier pkginfo. L'abréviation d'un package doit présenter les caractéristiques suivantes :

- L'abréviation doit contenir des caractères alphanumériques. Le premier caractère ne peut être un chiffre.
- L'abréviation ne peut pas contenir plus de 32 caractères.
- L'abréviation ne peut pas être une des abréviations réservées suivantes : install, new ou all.

Remarque – Les quatre premiers caractères doivent être uniques à votre entreprise. Par exemple, les packages créés par Sun Microsystems™ utilisent tous SUNW comme quatre premiers caractères de l'abréviation de leur package.

Exemple d'entrée d'abréviation de package dans le fichier pkginfo : PKG=SUNWcadap.

Spécification d'une architecture de package (ARCH)

Le paramètre ARCH figurant dans le fichier pkginfo identifie les architectures associées au package. Le nom de l'architecture doit être composé de 16 caractères alphanumériques maximum. Si un package est associé à plus d'une architecture, spécifiez les architectures dans une liste en les séparant par des virgules.

Exemple de spécification d'architecture de package dans un fichier pkginfo :

```
ARCH=sparc
```

Spécification de l'architecture du jeu d'instructions d'un package (SUNW_ISA)

Le paramètre SUNW_ISA figurant dans le fichier pkginfo identifie l'architecture du jeu d'instructions associé à un package Sun Microsystems. Les valeurs sont les suivantes :

- sparcv9, pour un package contenant des objets 64 bits
- sparc, pour un package contenant des objets 32 bits

Par exemple, la valeur du paramètre SUNW_ISA figurant dans un fichier pkginfo pour un package contenant 64 bits est :

```
SUNW_ISA=sparcv9
```

Si SUNW_ISA n'est pas défini, l'architecture du jeu d'instructions par défaut du package est définie sur la valeur du paramètre ARCH.

Spécification de la version d'un package (VERSION)

Le paramètre VERSION figurant dans le fichier pkginfo identifie la version du package. La version se compose de 256 caractères ASCII maximum et ne peut commencer par une parenthèse entrante.

Exemple de la spécification d'une version dans le fichier pkginfo :

```
VERSION=release 1.0
```

Définition du nom d'un package (NAME)

Le*nom d'un package* est le nom complet du package, défini par le paramètre NAME dans le fichier pkginfo.

Étant donné que les administrateurs système se réfèrent souvent au nom d'un package pour déterminer si celui doit ou non être installé, il est important de donner à chaque package un nom clair, précis et non abrégé. Les noms de package doivent répondre aux critères suivants :

- Indiquer dans quelles circonstances le package est requis (par exemple, pour fournir certaines commandes ou fonctionnalités) ou indiquer si le package est requis pour du matériel spécifique.
- Indiquer le cadre d'utilisation du package (par exemple, le développement de pilotes de périphérique).

- Inclure une description du procédé mnémonique de l'abréviation du package, à l'aide de mots clés qui indiquent que l'abréviation représente la description abrégée. Par exemple, le nom de package correspondant à l'abréviation de package SUNWbnuu est Basic Networking UUCP Utilities, (Usr).
- Nommer la partition sur laquelle le package est installé.
- Utiliser des termes tels qu'ils sont utilisés dans l'industrie.
- Exploiter la limite de 256 caractères.

Exemple de nom de package défini dans un fichier pkginfo :

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

Définition d'une catégorie de package (CATEGORY)

Le paramètre CATEGORY contenu dans le fichier pkginfo spécifie les catégories auxquelles un package appartient. Un package doit appartenir au moins à la catégorie system ou à la catégorie application. Les noms de catégorie sont composés de caractères alphanumériques. Les noms de catégorie peuvent être composés de 16 caractères maximum et ne respectent pas la casse.

Si un package appartient à plus d'une catégorie, spécifiez-les dans une liste en les séparant par des virgules.

Exemple de spécification du paramètre CATEGORY dans le fichier pkginfo :

```
CATEGORY=system
```

▼ Procédure de création d'un fichier pkginfo

1 **À l'aide d'un éditeur de texte, créez un fichier enregistré sous le nom pkginfo.**

Créez ce fichier sur votre système à l'emplacement de votre choix.

2 **Modifiez le fichier et définissez les cinq paramètres obligatoires.**

Les cinq paramètres obligatoires sont les suivants : PKG, NAME, ARCH, VERSION et CATEGORY. Pour plus d'informations sur ces paramètres, reportez-vous à "Création d'un fichier pkginfo" à la page 26.

3 **Ajoutez au fichier les paramètres facultatifs de votre choix.**

Créez vos propres paramètres ou reportez-vous à la page de manuel pkginfo(4) pour plus d'informations sur les paramètres standard.

4 **Enregistrez les modifications et quittez l'éditeur.**

Exemple 2–1 Création d'un fichier `pkginfo`

L'exemple suivant illustre le contenu d'un fichier `pkginfo` correct, dans lequel les cinq paramètres obligatoires et le paramètre `BASEDIR` sont définis. Le paramètre `BASEDIR` est abordé plus en détail à la rubrique "Champ *path*" à la page 33.

```
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
```

Voir aussi Reportez-vous à "Procédure d'organisation du contenu d'un package" à la page 30.

Organisation du contenu d'un package

Organisez les objets de votre package en une structure de répertoires hiérarchique qui reproduit la structure des objets du package sur le système cible à l'issue de l'installation. En effectuant cette opération avant de créer un fichier `prototype`, vous y gagnez en temps et en effort lors de la création du fichier.

▼ Procédure d'organisation du contenu d'un package

1 **Déterminez le nombre de packages à créer et les objets de package que chacun d'eux doit contenir.**

Pour plus d'informations sur cette étape, reportez-vous à "Critères à prendre en considération avant de créer un package" à la page 17.

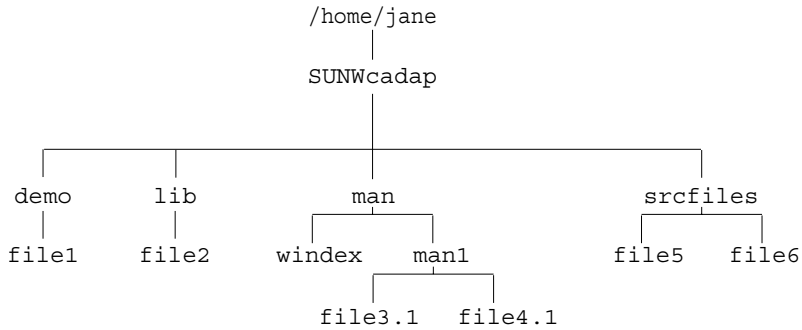
2 **Créez un répertoire pour chacun des packages à créer.**

Le répertoire peut être créé sur votre système à l'emplacement et avec le nom de votre choix. Les exemples fournis dans le présent chapitre présument que le nom du répertoire d'un package correspond à l'abréviation du package correspondant.

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

3 **Organisez les objets de chaque package dans une structure de répertoires placée dans le répertoire du package correspondant. La structure de répertoires doit reproduire celle prévue pour les objets du package sur le système cible.**

Par exemple, le package d'une application de CAO, SUNWcadap, requiert la structure de répertoires suivante :



4 **Décidez si les fichiers d'information doivent être conservés. Si nécessaire, créez un répertoire destiné à centraliser tous les fichiers.**

L'exemple suivant présume que l'exemple de fichier pkginfocréé au cours de la rubrique "Procédure de création d'un fichier pkginfo" à la page 29 est stocké dans le répertoire personnel de l'utilisateur Jane.

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

Voir aussi Reportez-vous à "Procédure de création d'un fichier prototype à l'aide de la commande pkgproto" à la page 43.

Création d'un fichier prototype

Le fichier prototype est un fichier ASCII qui sert à spécifier des informations sur les objets d'un package. Chaque entrée du fichier prototype décrit un seul objet, tel un fichier de données, un répertoire, un fichier source ou un objet exécutable. Les entrées du fichier prototype se compose de plusieurs champs d'information séparés par des espaces. Notez que ces champs *doivent* apparaître dans un ordre spécifique. Les lignes de commentaires commencent par une dièse (#) et sont ignorés.

Vous pouvez créer un fichier prototype dans un éditeur de texte ou à l'aide de la commande pkgproto. Lorsque vous créez ce fichier pour la première fois, l'utilisation de la commande pkgproto est recommandée car elle permet de créer le fichier d'après la hiérarchie des

répertoires créée précédemment. Si vous n'avez pas organisé vos fichiers comme il l'est suggéré à la rubrique "Organisation du contenu d'un package" à la page 30, il vous faut créer le fichier prototype de zéro dans un éditeur de texte, tâche qui s'avère bien plus longue.

Format du fichier prototype

Format de chaque ligne du fichier prototype :

partftypeclasspathmajorminormodeownergroup

part	Champ numérique facultatif qui vous permet de regrouper les objets de package en catégories. La valeur par défaut est part 1.
ftype	Champ à un caractère spécifiant le type de l'objet. Reportez-vous à "Champ ftype" à la page 32.
classe	Champ indiquant la classe d'installation à laquelle l'objet appartient. Reportez-vous à "Champ *class*" à la page 33.
path	Champ spécifiant le nom de chemin absolu ou relatif du répertoire dans lequel l'objet du package réside sur le système cible. Reportez-vous à "Champ *path*" à la page 33.
major	Champ contenant le numéro de périphérique principal des périphériques spéciaux en mode bloc ou mode caractère.
minor	Champ contenant le numéro de périphérique secondaire des périphériques spéciaux en mode bloc ou mode caractère.
mode	Champ indiquant le mode octal de l'objet (par exemple, 0644). Reportez-vous à "Champ *mode*" à la page 36.
owner	Indique le propriétaire de l'objet (par exemple, bin ou root). Reportez-vous à "Champ *owner*" à la page 37.
group	Indique le groupe auquel l'objet appartient (par exemple, bin ou sys). Reportez-vous à "Champ *group*" à la page 37.

Seuls les champs *ftype*, *class*, *path*, *mode*, *owner* et *group* sont habituellement définis. Ces champs sont décrits dans les sections suivantes. Reportez-vous à la page de manuel prototype(4) pour plus d'informations sur ces champs.

Champ ftype

Le champ *ftype* (type de fichier) est un champ à un caractère qui indique le type de fichier de l'objet du package. Les types de fichier valides sont décrits dans le tableau suivant :

TABLEAU 2–3 Types de fichier valides dans le fichier prototype

Valeur du champ ftype	Description du type de fichier
f	Fichier exécutable ou de données standard
e	Fichier à modifier lors de l'installation ou de la désinstallation (peut être partagé par plusieurs packages)
v	Fichier volatile (dont le contenu est sujet à modifications, tel un fichier journal)
d	Répertoire
x	Répertoire exclusif accessible uniquement par ce package (contient parfois des journaux ou des informations de base de données non enregistrés)
l	Fichier lié
p	Tube nommé
c	Périphérique spécial en mode caractère
b	Périphérique spécial en mode bloc
i	Fichier d'information ou script d'installation
s	Lien symbolique

Champ *class*

Le champ *class* nomme la classe à laquelle un objet appartient. L'utilisation des classes est une fonction de conception de package facultative. Cette fonction est traitée en détail à la la rubrique "Rédaction de scripts d'action de classe" à la page 71.

Lorsque vous n'utilisez pas de classes, les objets appartiennent à la classe none. Lorsque vous exécutez la commande pkgmk pour créer votre package, la commande insère le paramètre CLASSES=none dans le fichier pkginfo. Les fichiers dont le type est i doivent contenir un champ *class* vide.

Champ *path*

Le champ *path* sert à définir le répertoire dans lequel l'objet du package réside sur le système cible. Vous pouvez indiquer l'emplacement à l'aide d'un nom de chemin absolu (par exemple, /usr/bin/mail) ou d'un nom de chemin relatif (par exemple, bin/mail). L'utilisation d'un nom de chemin absolu signifie que l'emplacement de l'objet sur le système cible est défini par le package et ne peut être modifié. Un objet de package dont le nom de chemin est relatif est *réadressable*.

Un *objet réadressable* n'a pas besoin d'un chemin absolu sur le système cible. Son emplacement est en fait déterminé au cours de la procédure d'installation.

Vous pouvez définir certains, voire l'ensemble des objets d'un package comme étant des objets réadressables. Avant de rédiger des scripts d'installation ou de créer le fichier prototype, décidez si les objets du package doivent avoir un emplacement fixe (tels les scripts de démarrage dans /etc) ou s'ils doivent être réadressables.

Il existe deux types d'objet réadressable, ceux qui sont *réadressables collectivement* et ceux qui sont *réadressables individuellement*.

Objets réadressables collectivement

Les objets réadressables collectivement sont situés relativement à une base d'installation commune appelée le *répertoire de base*. Un répertoire de base est défini dans le fichier pkginfo à l'aide du paramètre BASEDIR. Par exemple, un objet réadressable figurant dans le fichier prototype appelé tests/generic nécessite que le fichier pkginfo définisse le paramètre BASEDIR par défaut. Exemple :

```
BASEDIR=/opt
```

Cet exemple signifie que lors de son installation, l'objet est placé dans le répertoire /opt/tests/generic.

Remarque – Le répertoire /opt est le seul répertoire dans lequel tout logiciel n'appartenant pas aux logiciels Solaris de base peut être installé.

Servez-vous autant que possible d'objets réadressables collectivement. De manière générale, la majeure partie d'un package peut être réadressable à l'aide de quelques fichiers (notamment les fichiers placés dans /etc ou /var) définis comme absolus. Toutefois, si un package contient divers emplacements de réadressage, envisagez la séparation du package en plusieurs packages employant des valeurs BASEDIR distinctes dans leurs fichiers pkginfo.

Objets réadressables individuellement

Les objets réadressables individuellement ne sont pas limités au même répertoire que les objets réadressables collectivement. Pour définir un objet réadressable individuellement, vous devez spécifier une variable d'installation dans le champ *path* du fichier prototype. Une fois la variable d'installation spécifiée, créez un script request pour demander au programme d'installation le répertoire de base réadressable, ou un script checkinstall pour déterminer le nom du chemin à partir des données du système de fichiers. Pour plus d'informations sur les scripts request, reportez-vous à "Rédaction d'un script request" à la page 64, et sur les scripts checkinstall, à "Procédure de recueil de données d'un système de fichiers" à la page 68.

Attention – Les objets réadressables individuellement sont difficiles à gérer. L'utilisation d'objets réadressables individuellement conduit parfois à la dispersion des composants de package qu'il est alors difficile d'isoler lors de l'installation de plusieurs versions ou architectures du même package. Servez-vous autant que possible d'objets réadressables collectivement.

Noms de chemin paramétriques

Un *nom de chemin paramétrique* est un nom de chemin qui inclut une spécification de variable. Par exemple, `/opt/$PKGINST/`*nomdefichier* est un nom de chemin paramétrique en raison de la spécification de la variable `$PKGINST`. La valeur par défaut d'une spécification de variable *doit* être définie dans le fichier `pkginfo`. La valeur peut ensuite être modifiée par un script `request` ou un script `checkinstall`.

La spécification de variable d'un chemin doit se trouver au début ou à la fin du chemin, ou être liée par des barres obliques (/). Un nom de chemin paramétrique valide utilise le format suivant :

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

La spécification de variable, une fois définie, peut conduire le chemin à être évalué en tant que chemin absolu ou réadressable. Dans l'exemple ci-après, le fichier `prototype` contient l'entrée suivante :

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

Le fichier `pkginfo` contient l'entrée suivante :

```
DIRLOC=/myopt
```

Le nom de chemin `$DIRLOC/tests/generic` est évalué comme étant le nom de chemin absolu `/myopt/tests/generic`, indépendamment de la définition (ou de l'absence de définition) du paramètre `BASEDIR` dans le fichier `pkginfo`.

Dans cet exemple, le fichier `prototype` est identique à celui de l'exemple précédent et le fichier `pkginfo` contient les entrées suivantes :

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

Le nom de chemin `$DIRLOC/tests/generic` est évalué comme étant le nom de chemin réadressable `/opt/firstcut/tests/generic`.

Pour plus d'informations sur les noms de chemin paramétriques, reportez-vous à “Utilisation des répertoires de base paramétriques” à la page 134.

Remarques sur les emplacements source et de destination des objets

Le champ *path* du fichier prototype définit l'emplacement de l'objet sur le système cible. Spécifiez l'emplacement actuel des objets du package dans le fichier prototype si la structure de leurs répertoires ne reproduit pas celle souhaitée sur le système cible. Reportez-vous à "Organisation du contenu d'un package" à la page 30 pour plus d'informations sur l'organisation des objets d'un package.

Si votre zone de développement n'est pas organisée de la façon souhaitée pour votre package, vous pouvez utiliser le format *chemin1=chemin2* dans le champ *path*. Dans ce format, *chemin1* correspond à l'emplacement souhaité pour l'objet sur le système cible et *chemin2*, à l'emplacement de l'objet sur votre système.

Vous pouvez aussi utiliser le format de nom de chemin *chemin1=chemin2*, avec *chemin1* comme nom d'objet réadressable et *chemin2* comme nom de chemin complet d'accès à l'objet en question sur votre système.

Remarque – *chemin1* ne peut pas contenir des variables de création non définies mais peut contenir des variables d'installation non définies. *chemin2* ne peut pas contenir des variables non définies bien qu'il soit possible d'utiliser des variables de création ou d'installation. Pour plus d'informations sur les différences entre les variables d'installation et les variables de création, reportez-vous à "Variables d'environnement d'un package" à la page 24.

Les liens doivent respecter le format *chemin1= chemin2* car ils sont créés à l'aide de la commande pkgadd. En règle générale, la variable *chemin2* d'un lien ne doit jamais être absolue ; elle doit par contre être relative à la section du répertoire de *chemin1*.

Une alternative à l'utilisation du format *chemin1=chemin2* est l'utilisation de la commande !search. Pour plus d'informations, reportez-vous à "Offre d'un chemin de recherche pour la commande pkgmk." à la page 43.

Champ *mode*

Le champ *mode* peut contenir un nombre octal, un point d'interrogation (?) ou une spécification de variable. Le nombre octal indique le mode de l'objet lors de son installation sur le système cible. Le point d'interrogation (?) indique que le mode reste inchangé lors de l'installation de l'objet, impliquant que l'objet du même nom existe déjà sur le système cible.

Une spécification de variable du type $*mode*, dans laquelle la première lettre de la variable doit être une minuscule, signifie que ce champ est défini lors de la création du package. Notez que cette variable doit être définie lors de la phase de création dans le fichier prototype ou comme option dans la commande pkgmk. Pour plus d'informations sur les différences entre les variables d'installation et les variables de création, reportez-vous à "Variables d'environnement d'un package" à la page 24.

Les fichiers de type i (fichier d'information), l (lien physique) et s (lien symbolique) ne doivent rien indiquer dans ce champ.

Champ *owner*

Le champ *owner* peut contenir un nom d'utilisateur, un point d'interrogation (?) ou une spécification de variable. Un nom d'utilisateur peut contenir 14 caractères maximum et doit correspondre à un nom existant sur le système cible (tel bin ou root). Le point d'interrogation (?) indique que le propriétaire reste inchangé lors de l'installation de l'objet, impliquant que l'objet du même nom existe déjà sur le système cible.

Une spécification de variable peut être du type $*Owner* ou $*owner*, où la première lettre de la variable est une majuscule ou une minuscule. Si la variable commence par une minuscule, elle doit être définie lors de la création du package, dans le fichier prototype ou comme option de la commande pkgmk. Si la variable commence par une majuscule, la spécification de variable est insérée dans le fichier pkginfo comme valeur par défaut et peut être redéfinie lors de l'installation par un script request. Pour plus d'informations sur les différences entre les variables d'installation et les variables de création, reportez-vous à "Variables d'environnement d'un package" à la page 24.

Les fichiers dont le type est i (fichier d'information) et lb (lien physique) ne doivent rien indiquer dans ce champ.

Champ *group*

Le champ *group* peut contenir un nom d'utilisateur, un point d'interrogation (?) ou une spécification de variable. Un nom de groupe peut contenir 14 caractères maximum et doit correspondre à un nom existant sur le système cible (tel bin ou sys). Le point d'interrogation (?) indique que le groupe reste inchangé lors de l'installation de l'objet, impliquant que l'objet du même nom existe déjà sur le système cible.

Une spécification de variable peut être du type $*Group* ou $*group*, où la première lettre de la variable est une majuscule ou une minuscule. Si la variable commence par une minuscule, elle doit être définie lors de la création du package, dans le fichier prototype ou comme option de la commande pkgmk. Si la variable commence par une majuscule, la spécification de variable est insérée dans le fichier pkginfo comme valeur par défaut et peut être redéfinie lors de l'installation par un script request. Pour plus d'informations sur les différences entre les variables d'installation et les variables de création, reportez-vous à "Variables d'environnement d'un package" à la page 24.

Les fichiers dont le type est i (fichier d'information) et l (lien physique) ne doivent rien indiquer dans ce champ.

Création d'un fichier prototype de zéro

Vous pouvez vous servir d'un éditeur de texte pour créer un fichier prototype de zéro, en ajoutant une entrée par objet de package. Reportez-vous à "Format du fichier prototype" à la page 32 et à la page de manuel prototype(4) pour plus d'informations sur le format de ce fichier. Cependant, une fois que tous les objets du package sont définis, vous avez la possibilité d'inclure des fonctions décrites à la rubrique "Ajout de fonctions au fichier prototype" à la page 40.

Exemple : Création d'un fichier prototype à l'aide de la commande pkgproto

Vous pouvez utiliser la commande pkgproto pour créer un fichier prototype de base, à condition d'avoir organisé la structure de répertoires du package de la manière décrite à la rubrique "Organisation du contenu d'un package" à la page 30. Par exemple, d'après l'exemple de structure de répertoires et de fichier pkginfo décrit au cours des sections précédentes, les commandes pour créer le fichier prototype sont les suivantes :

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

Le fichier prototype ressemble à ce qui suit :

```
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
```

Remarque – Le propriétaire effectif et le groupe de la personne créant le package sont enregistrés par la commande pkgproto. Une bonne technique consiste à utiliser les commandes chown -R et chgrp -R pour définir le propriétaire et le groupe souhaités *avant* d'exécuter la commande pkgproto.

Cet exemple de fichier prototype n'est pas terminé. Reportez-vous à la section suivante pour plus d'informations sur l'achèvement de ce fichier.

Derniers ajustements à apporter à un fichier prototype créé à l'aide de la commande pkgproto

Bien que la commande pkgproto soit utile pour créer un fichier prototype initial, elle ne crée pas d'entrées pour tous les objets de package qui doivent être définis. Cette commande ne crée pas des entrées complètes. La commande pkgproto n'effectue pas les opérations suivantes :

- Création d'entrées complètes pour les objets dont le type de fichier est v (fichiers volatiles), e (fichiers modifiables), x (répertoires exclusifs) ou i (fichiers d'information ou scripts d'installation) ;
- Prise en charge de plusieurs classes à l'aide d'un seul appel.

Création d'entrées d'objet dont le type de fichier est v, e, x et i

Vous devez systématiquement modifier le fichier prototype pour ajouter des objets dont le type de fichier est i. Si vous avez stocké vos fichiers d'information et vos scripts d'installation au premier niveau du répertoire de votre package (par exemple, /home/jane/SUNWcadap/pkginfo), une entrée du fichier prototype est comme suit :

```
i pkginfo
```

Si vous n'avez pas stocké vos fichiers d'information ni vos scripts d'installation au premier niveau du répertoire de votre package, vous devez spécifier leur emplacement source. Exemple :

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

Vous pouvez également utiliser la commande !search pour spécifier l'emplacement de la commande pkgmk afin de déterminer à quel moment créer le package. Pour plus d'informations, reportez-vous à “Offre d'un chemin de recherche pour la commande pkgmk.” à la page 43.

Pour ajouter des entrées d'objets dont le type de fichier est v, e et x, respectez le format décrit à la rubrique “Format du fichier prototype” à la page 32 ou reportez-vous à la page de manuel prototype(4).

Remarque – N'oubliez pas d'attribuer une classe aux fichiers dont le type est e (modifiable) et d'associer un script d'action de classe à celle-ci. Sans cette classe, les fichiers sont supprimés lors de la désinstallation du package même si le nom de chemin est partagé avec d'autres packages.

Utilisation de plusieurs définitions de classe

Si vous utilisez la commande pkgproto pour créer votre fichier prototype de base, vous pouvez attribuer tous les objets de package à la classe none ou à une classe spécifique. Comme illustré dans l'“Exemple : Création d'un fichier prototype à l'aide de la commande pkgproto” à la page 38, la commande pkgproto de base attribue tous les objets à la classe none. Pour attribuer tous les objets à une classe spécifique, vous pouvez utiliser l'option -c. Exemple :

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

Si vous utilisez plusieurs classes, il peut s'avérer nécessaire de modifier manuellement le fichier prototype pour modifier le champ *class* de chaque objet. S vous utilisez des classes, vous devez également définir le paramètre CLASSES du fichier pkginfo et rédiger des scripts d'action de classe. L'utilisation de classes est facultative. Cette fonction est traitée en détail à la la rubrique "Rédaction de scripts d'action de classe" à la page 71.

Exemple : Derniers ajustements à apporter à un fichier prototype créé à l'aide de la commande pkgproto

Le fichier prototype créé par la commande pkgproto dans l'"Exemple : Création d'un fichier prototype à l'aide de la commande pkgproto" à la page 38, requiert plusieurs modifications.

- Le fichier pkginfo doit avoir une entrée.
- Les champs *path* doivent être remplacés par le format *chemin1=chemin2* car la source du package se trouve dans /home/jane . Puisque la source du package est un répertoire hiérarchique et que la commande !search n'effectue pas de recherches de manière récurrente, l'utilisation du format *chemin1=chemin2* peut s'avérer plus simple.
- Les champs *owner* et *group* doivent contenir le nom des utilisateurs et des groupes existant sur le système cible. Autrement dit, le propriétaire jane génère une erreur car ce propriétaire n'est pas présent sur le système d'exploitation SunOS™.

  Le fichier prototype une fois modifié est comme suit :

```
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
```

Ajout de fonctions au fichier prototype

Outre la définition de chaque objet de package dans le fichier prototype, vous pouvez également effectuer les opérations suivantes :

- Définition d'objets supplémentaires à créer lors de la phase d'installation.

- Création de lien lors de la phase d'installation.
- Distribution de packages sur divers volumes.
- Imbrication de fichiers `prototype`.
- Définition d'une valeur par défaut pour les champs *mode*, *owner* et *group*.
- Offre d'un chemin de recherche pour la commande `pkgmk`.
- Définition de variables d'environnement.

Reportez-vous aux sections suivantes pour plus d'informations sur ces modifications.

Définition d'objets supplémentaires à créer lors de la phase d'installation

Vous pouvez utiliser le fichier `prototype` pour définir des objets qui ne sont pas fournis sur le support d'installation. Au cours de l'installation, la commande `pkgadd` permet de créer ces objets avec les types de fichier appropriés s'ils n'existent pas déjà au moment de l'installation.

Pour spécifier la création d'un objet sur le système cible, ajoutez l'entrée correspondante dans le fichier `prototype` pour indiquer le type de fichier approprié.

Pour par exemple créer un répertoire sur le système cible sans le fournir sur le support d'installation, ajoutez l'entrée suivante pour ce répertoire dans le fichier `prototype` :

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

Si vous souhaitez créer un fichier vide sur le système cible, l'entrée correspondant à ce fichier dans le fichier `prototype` doit être comme suit :

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

Les seuls objets qui *doivent* être fournis sur le support d'installation sont les fichiers standard et les scripts de modification (types de fichier `e`, `v`, `f`) et les répertoires requis pour les stocker. Tout autre objet est créé sans référence aux objets, répertoires, tubes nommés, périphériques, liens physiques ni aux liens symboliques fournis.

Création de lien lors de la phase d'installation

Pour créer des liens lors de la phase d'installation, indiquez les informations suivantes dans l'entrée de l'objet lié figurant dans le fichier `prototype` :

- Son type de fichier `l` (lien) ou `s` (lien symbolique) ;
- Le nom de chemin de l'objet lié au format *chemin1=chemin2*, où *chemin1* correspond au fichier de destination et *chemin2*, au fichier source ; En règle générale, la variable *chemin2* d'un lien ne doit jamais être absolue ; elle doit par contre être relative à la section du répertoire de *chemin1*. Par exemple, l'entrée d'un fichier `prototype` définissant un lien symbolique peut être comme suit :

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

Les liens relatifs sont spécifiés de cette manière que le package soit installé comme étant absolu ou réadressable.

Distribution de packages sur divers volumes

Lorsque vous créez votre package à l'aide de la commande pkgmk, celle-ci effectue les calculs et opérations nécessaires pour organiser le package en plusieurs volumes. Un package à plusieurs volumes est appelé un *package segmenté*.

Vous pouvez toutefois utiliser le champ facultatif *part* du fichier prototype pour définir la section dans laquelle l'objet doit être placé. Tout chiffre entré dans ce champ ignore la commande pkgmk et oblige l'élément à être placé dans la section donnée dans le champ. Notez qu'il existe une correspondance exacte entre les sections et volumes des supports amovibles formatés en tant que systèmes de fichiers. Si les volumes sont préattribués par le développeur, la commande pkgmk renvoie une erreur lorsque l'espace est insuffisant sur un des volumes.

Imbrication de fichiers prototype

Vous pouvez créer plusieurs fichiers prototype et les insérer dans le fichier prototype à l'aide de la commande !include. L'imbrication des fichiers facilite leur maintenance.

L'exemple suivant contient trois fichiers prototype. Le fichier principal (prototype) est en cours de modification. Les deux autres fichiers (proto2 et proto3) y sont insérés.

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

Définition de valeurs par défaut pour les champs *mode*, *owner* et *group*

Pour définir des valeurs par défaut pour les champs *mode*, *owner* et *group* d'objets de package spécifiques, vous pouvez insérer la commande !default dans le fichier prototype. Exemple :

```
!default 0644 root other
```

Remarque – La plage d'application de la commande !default débute à son point d'insertion et s'étend jusqu'à la fin du fichier. La plage d'application de la commande ne s'étend pas aux fichiers imbriqués.

Toutefois, pour les répertoires (type de fichier d) et les fichiers modifiables (type de fichier e) que vous êtes sûr de trouver sur les systèmes cibles (tels que /usr ou /etc/vfstab), vérifiez que les champs *mode*, *owner* et *group* du fichier prototype sont définis sur la valeur point d'interrogation (?). De cette manière, vous n'écrasez aucun paramètre existant susceptible d'avoir été modifié par l'administrateur du site.

Offre d'un chemin de recherche pour la commande pkgmk.

Si l'emplacement source des objets de package diffère de leur emplacement de destination et que vous ne souhaitez pas vous servir du format *chemin1=chemin2* comme décrit à la rubrique "Remarques sur les emplacements source et de destination des objets" à la page 36, vous pouvez utiliser la commande `!search` dans le fichier `prototype`.

Si vous avez par exemple créé un répertoire `pkgfiles` dans votre répertoire personnel dans lequel vous avez stocké tous vos fichiers d'information et scripts d'installation, vous pouvez spécifier que la recherche s'effectue dans ce répertoire lorsque le package est créé à l'aide de la commande `pkgmk`.

La commande dans le fichier prototype ressemble à la suivante :

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

Remarque – Les demandes de recherche ne s'étendent pas aux fichiers imbriqués. D'autre part, une recherche se limite aux répertoires répertoriés et ne s'effectue pas de manière récurrente.

Définition de variables d'environnement

Vous pouvez également ajouter des commandes au fichier `prototype` du type `!`*PARAM=valeur*. Les commandes de ce type définissent des variables dans l'environnement actuel. Si vous avez créé plusieurs fichiers `prototype`, notez que cette commande ne s'applique qu'au fichier `prototype` dans lequel elle est définie.

La variable *PARAM* peut commencer par une minuscule ou une majuscule. Si la valeur de la variable *PARAM* est inconnue lors de la phase de création, l'exécution de la commande `pkgmk` est suspendue et une erreur est renvoyée. Pour plus d'informations sur les différences entre les variables d'installation et les variables de création, reportez-vous à "Variables d'environnement d'un package" à la page 24.

▼ Procédure de création d'un fichier prototype à l'aide de la commande pkgproto

---

Remarque – Il est plus simple de créer les fichiers d'information et les scripts d'installation avant de créer un fichier `prototype`. Cet ordre n'est cependant pas obligatoire. Vous pouvez toujours modifier le fichier `prototype` après avoir modifié le contenu de votre package. Pour plus d'informations sur les fichiers d'information et les scripts d'installation, reportez-vous au Chapitre 3, "Amélioration de la fonctionnalité d'un package (opérations)".

---

1 **Si vous ne l'avez pas déjà fait, déterminez les objets de package devant être absolus et ceux devant être réadressables.**

Pour plus d'informations à ce sujet, reportez-vous à "Champ *path*" à la page 33.

2 **Organisez les objets de votre package de façon à reproduire leur emplacement sur le système cible.**

Si vous avez déjà organisé vos packages comme décrit à la rubrique "Organisation du contenu d'un package" à la page 30, notez qu'il vous faudra peut-être apporter certaines modifications en fonction des décisions que vous avez prises à l'Étape 1. Si vous n'avez pas encore organisé votre package, faîtes-le maintenant. Lorsqu'un package n'est pas organisé, il est impossible d'utiliser la commande `pkgproto` pour créer un fichier `prototype` de base.

3 **Si votre package contient des objets réadressables collectivement, modifiez le fichier `pkginfo` pour donner au paramètre `BASEDIR` la valeur appropriée.**

Exemple :

```
BASEDIR=/opt
```

Pour plus d'informations sur les objets réadressables collectivement, reportez-vous à la rubrique "Objets réadressables collectivement" à la page 34.

4 **Si votre package contient des objets réadressables individuellement, créez un script `request` invitant le programme d'installation à entrer le nom de chemin approprié. Vous pouvez également créer un script `checkinstall` pour déterminer le chemin approprié à partir des données du système de fichiers.**

La liste suivante indique le numéro de page de référence des opérations courantes :

- Pour créer un script `request`, reportez-vous à "Procédure de rédaction d'un script `request`" à la page 66.
- Pour créer un script `checkinstall`, reportez-vous à "Procédure de recueil de données d'un système de fichiers" à la page 68.
- Pour plus d'informations sur les objets réadressables individuellement, reportez-vous à la rubrique "Objets réadressables individuellement" à la page 34.

5 **Faites du propriétaire et du groupe de tous les composants du package le propriétaire et le groupe requis sur les systèmes cibles.**

Utilisez les commandes `chown -R` et `chgrp -R` sur le répertoire de votre package et sur les fichiers d'information.

6 **Exécutez la commande `pkgproto` pour créer un fichier `prototype` de base.**

La commande `pkgproto` balaye vos répertoires afin de créer un fichier de base. Exemple :

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

Le fichier `prototype` peut se trouver à tout endroit de votre système. La centralisation de vos fichiers d'information et scripts d'installation simplifie l'accès et la maintenance. Pour plus d'informations sur la commande `pkgproto`, reportez-vous à la page de manuel `pkgproto(1)`.

7 **Modifiez le fichier `prototype` à l'aide d'un éditeur de texte afin d'y ajouter des entrées pour les fichiers de type `v`, `e`, `x` et `i`.**

Pour plus d'informations sur les modifications spécifiques que vous pouvez avoir à apporter, reportez-vous à "Derniers ajustements à apporter à un fichier `prototype` créé à l'aide de la commande `pkgproto`" à la page 39.

8 **(Facultatif) Si vous utilisez plusieurs classes, modifiez les fichiers `prototype` et `pkginfo`. Utilisez un éditeur de texte pour apporter les modifications nécessaires, puis créez les scripts d'action de classe correspondantes.**

Pour plus d'informations sur les modifications spécifiques que vous pouvez avoir à apporter, reportez-vous à "Derniers ajustements à apporter à un fichier `prototype` créé à l'aide de la commande `pkgproto`" à la page 39 et à "Rédaction de scripts d'action de classe" à la page 71.

9 **Modifiez le fichier `prototype` à l'aide d'un éditeur de texte afin d'y redéfinir les noms de chemin et autres valeurs de champ.**

Pour plus d'informations, reportez-vous à "Derniers ajustements à apporter à un fichier `prototype` créé à l'aide de la commande `pkgproto`" à la page 39.

10 **(Facultatif) Modifiez le fichier `prototype` à l'aide d'un éditeur de texte pour lui ajouter des fonctions.**

Pour plus d'informations, reportez-vous à "Ajout de fonctions au fichier `prototype`" à la page 40.

11 **Enregistrez les modifications et quittez l'éditeur.**

Voir aussi Si vous êtes prêt à passer à l'étape suivante, reportez-vous à "Procédure de création d'un package" à la page 47.

Création d'un package

Utilisez la commande `pkgmk` pour créer votre package. La commande `pkgmk` effectue les opérations suivantes :

- Enregistrement de tous les objets définis dans le fichier `prototype` au format répertoire.
- Création du fichier `pkgmap` pour remplacer le fichier `prototype`.
- Création d'un package pouvant être installé, utilisé comme entrée de la commande `pkgadd`.

Utilisation de la commande pkgmk la plus simple

La forme la plus simple de cette commande est la commande `pkgmk` sans aucune option. Avant d'utiliser la commande `pkgmk` sans aucune option, vérifiez que votre répertoire de travail actuel contient le fichier `prototype` du package. Les résultats de la commande, des fichiers et des répertoires sont consignés dans le répertoire `/var/spool/pkg`.

Fichier pkgmap

Lorsque vous créez un package à l'aide de la commande `pkgmk`, un fichier `pkgmap` est créé en remplacement du fichier `prototype`. Le fichier `pkgmap` de l'exemple précédent contient les données suivantes :

```
$ 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
$
```

Le format de ce fichier est très similaire au format du fichier `prototype`. Toutefois, le fichier `pkgmap` inclut les informations suivantes :

- La première ligne indique le nombre de volumes balayés par le package et la taille approximative du package une fois installé.

  Par exemple, `: 1 3170` indique que le package balaye un seul volume et qu'il utilise environ 3170 blocs de 512 octets une fois installé.
- Trois autres champs définissent la taille, la somme de contrôle et le temps de modification de chaque objet de package.
- Les objets de package sont répertoriés par ordre alphabétique (par classe puis par nom de chemin) pour réduire le temps d'installation du package.

▼ Procédure de création d'un package

1 **Si nécessaire, créez un fichier `pkginfo`.**

Pour une procédure détaillée, reportez-vous à "Procédure de création d'un fichier `pkginfo`" à la page 29.

2 **Si nécessaire, créez un fichier `prototype`.**

Pour une procédure détaillée, reportez-vous à "Procédure de création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 43.

3 **Faites du répertoire contenant le fichier `prototype` du package votre répertoire de travail.**

4 **Créez le package.**

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

`-o`	Remplace la version existante du package.
`-a` *arch*	Remplace les informations sur l'architecture du fichier `pkginfo`.
`-b` *rép-src-base*	Demande que *rép-src-base* soit ajouté au début des noms de chemin réadressables lorsque la commande `pkgmk` recherche des objets sur le système de création.
`-d` *périphérique*	Indique que la package doit être copié sur le *périphérique* qui peut être un nom de chemin de répertoire absolu, une disquette ou un disque amovible.
`-f` *nomdefichier*	Nomme le fichier *nomdefichier* utilisé comme fichier `prototype`. Les noms par défaut sont `prototype` ou `Prototype`.
`-l` *limite*	Spécifie la taille maximale, en blocs de 512 octets, du périphérique de sortie.
`-p` *horodp*	Remplace la définition de l'horodatage de production figurant dans le fichier `pkginfo`.
`-r` *cheminroot*	Demande que le répertoire root *cheminroot* soit utilisé pour localiser les objets sur le système de développement.
`-v` *version*	Remplace les informations de version figurant dans le fichier `pkginfo`.
PARAM=valeur	Définit des variables d'environnement globales. Les variables commençant par une minuscule sont résolues lors de la phase de création. Celles qui commencent par une majuscule sont placées dans le fichier `pkginfo` afin d'être utilisée lors de la phase d'installation.
instpkg	Indique un package par son abréviation ou une instance spécifique (par exemple, `SUNWcadap.4`).

Pour plus d'informations, reportez-vous à la page de manuel pkgmk(1).

5 Vérifiez le contenu du package.

```
$ 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 *nom-périphérique*	Indique l'emplacement du package. Notez que *nom-périphérique* peut être un nom de chemin d'accès à un répertoire complet ou être les identificateurs d'une bande ou d'un disque amovible.
pkg-abrév	Correspond au nom d'un ou plusieurs packages à vérifier (séparés par des espaces). En cas d'omission, la commande pkgchk vérifie tous les packages disponibles.

La commande pkgchk indique quels aspects du package sont vérifiés et, le cas échéant, affiche des avertissements ou des messages d'erreur. Pour plus d'informations sur la commande pkgchk, reportez-vous à "Vérification de l'intégrité d'un package" à la page 92.

Attention – Les erreurs doivent être prises très au sérieux. Une erreur peut indiquer la nécessité de corriger un script. Consultez toutes les erreurs et ignorez-les si vous êtes en désaccord avec le résultat de la commande pkgchk.

Exemple 2–2 Création d'un package

L'exemple suivant utilise le fichier prototype créé à la rubrique "Derniers ajustements à apporter à un fichier prototype créé à l'aide de la commande pkgproto" à la page 39.

```
$ 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.
$
```

Exemple 2–3 Spécification d'un répertoire source pour les fichiers réadressables

Si votre package contient des fichiers réadressables, vous pouvez utiliser l'option `-b` *rép-src-base* de la commande `pkgmk` pour spécifier un nom de chemin à ajouter au début des noms de chemin réadressables lors de la création du package. Cette option est utile lorsque vous n'avez pas utilisé le format *chemin1=chemin2* pour les fichiers réadressables, ni spécifié de chemin de recherche à l'aide de la commande `!search` dans le fichier `prototype`.

La commande suivante crée un package doté des caractéristiques suivantes :

- Le package est créé à partir de l'exemple de fichier `prototype` créé par la commande `pkgproto`. Reportez-vous à "Exemple : Création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 38 pour plus d'informations.
- Le package est créé sans modifier les champs *path*.
- Le package ajoute une entrée pour le fichier `pkginfo`.

```
$ 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.
```

Dans cet exemple, le package est créé dans le répertoire par défaut, `/var/spool/pkg`, en spécifiant l'option `-o`. Cette option remplace le package créé à l'Exemple 2–2.

Exemple 2–4 Spécification de répertoires source distincts pour les fichiers d'information et les objets de package

Si vous placez les fichiers d'information de package (notamment `pkginfo` et `prototype`) et les objets de package dans deux répertoires distincts, vous pouvez créer votre package à l'aide des options `-b` *rép-src-base* et `-r` *cheminroot* dans la commande `pkgmk`. Si vous placez les objets de package dans un répertoire appelé `/product/pkgbin` et les autres fichiers d'information de package dans un répertoire appelé `/product/pkgsrc`, vous pouvez utiliser la commande suivante pour placer le package dans le répertoire `/var/spool/pkg` :

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

(Facultatif) Vous pouvez utiliser les commandes suivantes pour obtenir le même résultat :

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

Dans cet exemple, la commande `pkgmk` utilise le répertoire de travail actuel pour localiser les autres éléments du package (notamment les fichiers d'information `prototype` et `pkginfo`).

Voir aussi Pour ajouter tout fichier d'information ou script d'installation facultatif à votre package, reportez-vous au Chapitre 3, “Amélioration de la fonctionnalité d'un package (opérations)”. Sinon, une fois le package créé, vous devez vérifier son intégrité. Le Chapitre 4, “Vérification et transfert d'un package” vous explique comment vérifier l'intégrité du package et décrit sa procédure de transfert sur un support de distribution.

◆ ◆ ◆ CHAPITRE 3

Amélioration de la fonctionnalité d'un package (opérations)

Ce chapitre décrit la création facultative de fichiers d'information et de scripts d'installation d'un package. Alors que le Chapitre 2, “Création d'un package” aborde la configuration minimale requise pour la création d'un package, le présent chapitre traite des fonctions supplémentaires qui peuvent être intégrées à un package. Ces fonctions supplémentaires dépendent des critères que vous avez retenus pour concevoir votre package. Pour plus d'informations, reportez-vous à “Critères à prendre en considération avant de créer un package” à la page 17.

La liste suivante répertorie les informations fournies dans ce chapitre :

Création de fichiers d'information et de scripts d'installation (liste de tâches)

La liste de tâches suivante décrit les fonctions facultatives pouvant être intégrées à un package.

TABLEAU 3–1 Création de fichiers d'information et de scripts d'installation (liste de tâches)

Tâche	Description	Voir
1. Création de fichiers d'information	*Définissez les dépendances du package.* La définition des dépendances d'un package vous permet de spécifier si le package est compatible avec des versions précédentes, s'il dépend d'autres packages ou si d'autres packages dépendent de lui.	“Procédure de définition des dépendances d'un package” à la page 53

TABLEAU 3–1 Création de fichiers d'information et de scripts d'installation (liste de tâches) *(Suite)*

Tâche	Description	Voir
	Rédigez un message de copyright. Le fichier de `copyright` décrit la protection juridique de l'application logicielle.	"Procédure de rédaction d'un message de copyright" à la page 56
	Réservez de l'espace supplémentaire sur le système cible. Un fichier `space` réserve des blocs sur le système cible afin de vous permettre, lors de l'installation, de créer des fichiers non définis dans le fichier `pkgmap`.	"Procédure de réservation d'espace supplémentaire sur un système cible" à la page 58
2. Création de scripts d'installation	*Obtenez des informations du programme d'installation.* Un script `request` vous permet d'obtenir des informations de la personne qui installe votre package.	"Procédure de rédaction d'un script `request`" à la page 66
	Recueillez les données du système de fichiers requises pour l'installation. Un script `checkinstall` vous permet d'effectuer une analyse du système cible et de configurer l'environnement approprié ou l'arrêt net de l'installation.	"Procédure de recueil de données d'un système de fichiers" à la page 68
	Rédigez des scripts de procédure. Les scripts de procédure vous permettent de fournir des instructions personnalisées lors de phases spécifiques de la procédure d'installation ou de suppression.	"Procédure de rédaction de scripts de procédure" à la page 71
	Rédigez des scripts d'action de classe. Les scripts d'action de classe vous permettent de spécifier une série d'instructions à exécuter lors de l'installation et de la suppression du package sur des groupes d'objets de package particuliers.	"Procédure de rédaction de scripts d'action de classe" à la page 79

Création de fichiers d'information

Cette section se consacre aux fichiers facultatifs d'information de package. Ces fichiers permettent de définir les dépendances des packages, de fournir un message de copyright et de réserver de l'espace supplémentaire sur le système cible.

Définition des dépendances d'un package

Vous devez déterminer si votre package a des dépendances vis à vis d'autres packages et si d'autres packages dépendent du vôtre. Les dépendances et les incompatibilités d'un package peuvent être définies à l'aide de deux des fichiers facultatifs d'information de package, `compver` et `depend`.

Le fichier `compver` vous permet d'indiquer les versions antérieures de votre package qui sont compatibles avec le package à installer.

Le fichier `depend` vous permet de définir trois types de dépendances associées à votre package. Ces types de dépendances sont les suivants :

- *Package prérequis* : votre package dépend de l'existence d'un autre package ;
- *Dépendance inverse* : un autre package dépend de l'existence du vôtre ;

  Remarque – Utilisez le type dépendance inverse uniquement lorsqu'un package incapable de fournir un fichier `depend` dépend de votre package.

- *Package incompatible* : votre package est incompatible avec le package nommé.

Le fichier `depend` ne résout que des dépendances très simples. Si votre package dépend d'un fichier spécifique, de son contenu ou de son comportement, le fichier `depend` ne fournit pas les informations appropriées. Dans ce cas, vous devez vous servir d'un script `request` ou d'un script `checkinstall` pour effectuer un contrôle détaillé des dépendances. Le script `checkinstall` est également le seul script capable d'effectuer un arrêt net de la procédure d'installation du package.

Remarque – Vérifiez que les fichiers `depend` et `compver` ont des entrées dans le fichier `prototype`. Le type de fichier doit être `i` (fichier d'information de package).

Reportez-vous aux pages de manuel `depend(4)` et `compver(4)` pour plus d'informations.

▼ Procédure de définition des dépendances d'un package

1 Faites du répertoire contenant vos fichiers d'information votre répertoire de travail actuel.

2 **S'il existe des versions précédentes de votre package et que vous devez spécifier que la nouvelle version est compatible avec les anciennes, créez un fichier nommé `compver` à l'aide d'un éditeur de texte.**

Répertoriez les versions compatibles avec votre package. Utilisez le format suivant :

string string . . .

La valeur de *chaîne* est identique à la valeur attribuée au paramètre `VERSION` dans le fichier `pkginfo` pour chaque package compatible.

3 **Enregistrez les modifications et quittez l'éditeur.**

4 **Si votre package dépend de l'existence d'autres packages, si d'autres packages dépendent de l'existence du vôtre ou, si votre package est incompatible avec un autre package, créez un fichier nommé `depend` avec votre éditeur de texte.**

Ajoutez une entrée pour chaque dépendance. Utilisez le format suivant :

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

type — Définit le type de dépendance. Doit être l'un des caractères suivants : `P` (package prérequis), `I` (package incompatible) ou `R` (dépendance inverse).

pkg-abrév — Indique l'abréviation du package telle que `SUNWcadap`.

pkg-nom — Indique le nom complet du package tel que `Chip designers need CAD application software to design abc chips. Runs only on xyz hardware and is installed in the usr partition.`

(*arch*) — Facultatif. Indique le type de matériel sur lequel le package s'exécute. Par exemple, `sparc` ou `x86`. Si vous spécifiez une architecture, vous devez utiliser des parenthèses comme séparateurs.

version — Facultatif. Indique la valeur attribuée au paramètre `VERSION` dans le fichier `pkginfo`.

Pour plus d'informations, reportez-vous à `depend(4)`.

5 **Enregistrez les modifications et quittez l'éditeur.**

6 **Effectuez l'*une* des opérations suivantes :**

- Si vous souhaitez créer des fichiers d'information et des scripts d'installation supplémentaires, passez à l'étape suivante, "Procédure de rédaction d'un message de copyright" à la page 56.
- Si vous n'avez pas créé de fichier `prototype`, suivez la procédure "Procédure de création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 43. Passez à l'Étape 7.

- Si vous avez déjà créé un fichier `prototype`, modifiez-le en ajoutant une entrée pour chaque fichier qui vient d'être créé.

7 Créez votre package.

Si nécessaire, reportez-vous à la rubrique "Procédure de création d'un package" à la page 47.

Exemple 3–1 Fichier `compver`

Cet exemple contient quatre versions d'un package : 1.0, 1.1, 2.0 et le nouveau package, 3.0. Le nouveau package est compatible avec les trois versions précédentes. Le fichier `compver` de la nouvelle version est comme suit :

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

Les entrées ne doivent pas nécessairement apparaître en séquence. Elles doivent cependant parfaitement correspondre à la définition du paramètre `VERSION` figurant dans le fichier `pkginfo` de chaque package. Dans cet exemple, les concepteurs de package ont utilisé des formats différents dans les trois premières versions.

Exemple 3–2 Fichier `depend`

Cet exemple suppose que le package `SUNWcadap` nécessite l'installation préalable des packages `SUNWcsr` et `SUNWcsu` sur le système cible. Le fichier `depend` de `SUNWcadap` est comme suit :

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

Voir aussi Une fois le package créé, installez-le pour confirmer qu'il s'installe correctement et vérifier son intégrité. Le Chapitre 4, "Vérification et transfert d'un package" vous explique comment vérifier l'intégrité du package et décrit sa procédure de transfert sur un support de distribution.

Rédaction d'un message de copyright

Vous devez décider si votre package doit afficher un message de copyright lors de son installation. Dans l'affirmative, créez le fichier `copyright`.

Remarque – Vous devez inclure un fichier `copyright` afin d'offrir une protection juridique à votre application logicielle. Renseignez-vous auprès du service juridique de votre société pour connaître la formulation exacte du message.

Pour fournir un message de copyright, vous devez créer un fichier nommé `copyright`. Le message s'affiche au cours de l'installation tel qu'il apparaît dans le fichier (sans aucun formatage). Reportez-vous à la page de manuel `copyright(4)` pour plus d'informations.

Remarque – Vérifiez que votre fichier `copyright` dispose d'une entrée dans le fichier `prototype`. Le type de fichier doit être `i` (fichier d'information de package).

▼ Procédure de rédaction d'un message de copyright

1. **Faites du répertoire contenant vos fichiers d'information votre répertoire de travail actuel.**
2. **Créez un fichier nommé `copyright` à l'aide de votre éditeur de texte.**

   Saisissez le texte du message de copyright tel qu'il doit apparaître lors de l'installation de votre package.
3. **Enregistrez les modifications et quittez l'éditeur.**
4. **Effectuez l'*une* des opérations suivantes :**
   - Si vous souhaitez créer des fichiers d'information et des scripts d'installation supplémentaires, passez à l'étape suivante, "Procédure de réservation d'espace supplémentaire sur un système cible" à la page 58.
   - Si vous n'avez *pas* créé de fichier `prototype`, suivez la procédure "Procédure de création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 43. Passez à l'Étape 5.
   - Si vous avez déjà créé un fichier `prototype`, modifiez-le en ajoutant une entrée pour le fichier d'information qui vient d'être créé.
5. **Créez votre package.**

   Si nécessaire, reportez-vous à la rubrique "Procédure de création d'un package" à la page 47.

Exemple 3–3 Fichier `copyright`

Un message de copyright partiel peut être comme suit :

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

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

Voir aussi Une fois le package créé, installez-le pour confirmer qu'il s'installe correctement et vérifier son intégrité. Le Chapitre 4, "Vérification et transfert d'un package" vous explique comment vérifier l'intégrité du package et décrit sa procédure de transfert sur un support de distribution.

Réservation d'espace supplémentaire sur un système cible

Vous devez déterminer si votre package nécessite de l'espace disque supplémentaire sur le système cible. Cet espace vient s'ajouter à l'espace requis par les objets du package. Dans l'affirmative, créez le fichier d'information `space`. Cette opération diffère de la création de fichiers et répertoires vides lors de la phase d'installation, comme indiqué à la rubrique "Définition d'objets supplémentaires à créer lors de la phase d'installation" à la page 41.

La commande `pkgadd` vérifie que suffisamment d'espace disque est disponible pour l'installation de votre package, en fonction de la définition des objets figurant dans le fichier `pkgmap`. Toutefois, un package requiert parfois de l'espace disque supplémentaire pour accommoder d'autres éléments outre les objets définis dans le fichier `pkgmap`. Par exemple, votre package peut créer un fichier à l'issue de l'installation contenant une base de données, des fichiers journaux ou tout autre fichier à taille croissante et utilisant donc de plus en plus d'espace disque. Pour vous assurer que suffisamment d'espace disque soit réservé à cet effet, vous devez inclure un fichier `space` spécifiant l'espace disque requis. La commande `pkgadd` recherche l'espace supplémentaire spécifié dans un fichier `space`. Reportez-vous à la page de manuel`space(4)` pour plus d'informations.

Remarque – Vérifiez que votre fichier `space` dispose d'une entrée dans le fichier `prototype`. Le type de fichier doit être `i` (fichier d'information de package).

▼ Procédure de réservation d'espace supplémentaire sur un système cible

1 **Faites du répertoire contenant vos fichiers d'information votre répertoire de travail actuel.**

2 **Créez un fichier nommé `space` à l'aide de votre éditeur de texte.**

Spécifiez l'espace disque supplémentaire requis par votre package. Utilisez le format suivant :

pathname blocks inodes

nomdechemin	Indique un nom de répertoire qui peut correspondre au point de montage de systèmes de fichiers.
blocs	Indique le nombre de blocs de 512 octets à réserver.
i-nodes	Indique le nombre d'i-nodes requis.

Pour plus d'informations, reportez-vous à la page de manuel `space(4)`.

3 **Enregistrez les modifications et quittez l'éditeur.**

4 **Effectuez l'une des opérations suivantes :**

- Pour créer des scripts d'installation, passez à l'opération suivante, "Procédure de rédaction d'un script `request`" à la page 66.
- Si vous n'avez pas créé de fichier `prototype`, suivez la procédure "Procédure de création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 43. Passez à l'Étape 5.
- Si vous avez déjà créé un fichier `prototype`, modifiez-le en ajoutant une entrée pour le fichier d'information qui vient d'être créé.

5 **Créez votre package.**

Si nécessaire, reportez-vous à la rubrique "Procédure de création d'un package" à la page 47.

Exemple 3–4 Fichier `space`

Cet exemple de fichier `space` indique que 1000 blocs de 512 octets et 1 i-node doivent être réservés dans le répertoire `/opt` sur le système cible.

```
/opt   1000   1
```

Voir aussi Une fois le package créé, installez-le pour confirmer qu'il s'installe correctement et vérifier son intégrité. Le Chapitre 4, "Vérification et transfert d'un package" vous explique comment vérifier l'intégrité du package et décrit sa procédure de transfert sur un support de distribution.

Création de scripts d'installation

Cette section décrit les scripts d'installation de package facultatifs. La commande `pkgadd` effectue automatiquement toutes les opérations nécessaires pour installer un package à l'aide de fichiers d'information de package comme entrée. Il n'est *pas* nécessaire de fournir des scripts d'installation de package. Toutefois, pour créer des procédures d'installation personnalisées pour votre package, vous pouvez faire appel à des scripts d'installation. Scripts d'installation :

- Ils doivent pouvoir être exécutés par Bourne shell (`sh`).
- Ils doivent contenir des commandes Bourne shell et du texte.
- Ils ne doivent pas nécessairement contenir l'identifiant shell `#!/bin/sh`.
- Ils ne doivent pas nécessairement s'agir de fichiers exécutables.

Il existe quatre types de scripts d'installation vous permettant d'effectuer des opérations personnalisées :

- Script `request`

  Le script `request` demande des données à l'administrateur qui installe un package afin d'attribuer ou de redéfinir des variables d'environnement.

- Script `checkinstall`

  Le script `checkinstall` examine le système cible à la recherche des données nécessaires, peut définir ou modifier les variables d'environnement du package et détermine si l'installation peut avoir lieu.

  Remarque – Le script `checkinstall` est disponible depuis la version 2.5 de Solaris et versions compatibles.

- Scripts de procédure

  Les scripts de procédure identifient la procédure à appeler avant ou après l'installation ou la suppression d'un package. Les quatre scripts de procédure sont `preinstall`, `postinstall`, `preremove` et `postremove`.

- Scripts d'action de classe

  Les scripts d'action de classe définissent une action ou un groupe d'actions à appliquer à une classe de fichiers lors de l'installation ou de la suppression. Vous pouvez définir vos propres classes. Vous pouvez aussi utiliser l'une des quatre classes standard (`sed`, `awk`, `build` et `preserve`).

Traitement des scripts pendant l'installation d'un package

Le type de scripts à utiliser dépend du stade de la procédure d'installation auquel l'action du script est requise. Pendant l'installation d'un package, la commande `pkgadd` effectue les opérations suivantes :

1. Elle exécute le script `request`.

   Cette opération est la seule au cours de laquelle votre package peut demander la participation de l'administrateur chargé d'installer le package .

2. Elle exécute le script `checkinstall`.

   Le script `checkinstall` recueille les données du système de fichiers et peut créer ou modifier la définition des variables d'environnement afin de contrôler l'installation ultérieure. Pour plus d'informations sur les variables d'environnement d'un package, reportez-vous à "Variables d'environnement d'un package" à la page 24.

3. Elle exécute le script `preinstall`.
4. Elle installe les objets de package de chaque classe à installer.

   L'installation de ces fichiers s'effectue classe par classe et les scripts d'action de classe sont exécutés en conséquence. La liste des classes employées et l'ordre dans lequel elles doivent être installées sont initialement définis avec le paramètre `CLASSES` dans votre fichier `pkginfo`. Toutefois, votre script `request` ou `checkinstall` peut modifier la valeur du paramètre `CLASSES`. Pour plus d'informations sur le traitement des classes pendant l'installation, reportez-vous à "Traitement des classes pendant l'installation d'un package" à la page 72.

   a. Crée des liens symboliques, des périphériques, des tubes nommés et les répertoires requis.

   b. Installe les fichiers standard (fichiers de type `e`, `v`, `f`), en fonction de leur classe.

      Seuls les fichiers standard à installer sont transmis au script d'action de classe. Tous les autres objets de package sont créés automatiquement à partir des informations figurant dans le fichier `pkgmap`.

   c. Crée tous les liens physiques.

5. Elle exécute le script `postinstall`.

Traitement des scripts pendant la suppression d'un package

Lors de la suppression d'un package, la commande `pkgrm` effectue les opérations suivantes :

1. Elle exécute le script `preremove`.
2. Elle supprime les objets de package de chaque classe.

   La suppression s'effectue également classe par classe. Les scripts de suppression sont traités dans l'ordre inverse de l'installation, en fonction de la séquence définie dans le paramètre `CLASSES`. Pour plus d'informations sur le traitement des classes pendant l'installation, reportez-vous à "Traitement des classes pendant l'installation d'un package" à la page 72.

   a. Supprime les liens physiques.
   b. Supprime les fichiers standard.
   c. Supprime les liens symboliques, les périphériques et les tubes nommés.
3. Elle exécute le script `postremove`.

Le script `request` n'est pas traité lors de la suppression d'un package. Toutefois, le résultat du script est conservé dans le package installé et mis à la disposition des scripts de suppression. Le résultat du script `request` est une liste de variables d'environnement.

Variables d'environnement de package mises à la disposition des scripts

Les groupes suivants de variables d'environnement sont mis à la disposition de tous les scripts d'installation. Certaines variables d'environnement peuvent être modifiées par un script `request` ou un script `checkinstall`.

- Le script `request` ou le script `checkinstall` peut définir ou modifier tout paramètre standard contenu dans le fichier `pkginfo`, à l'exception des paramètres obligatoires. Les paramètres d'installation standard sont décrits en détail à la page de manuel `pkginfo(4)`.

  Remarque – Le paramètre `BASEDIR` ne peut être modifié qu'à partir de la version 2.5 de Solaris et des versions compatibles.

- Vous pouvez définir vos propres variables d'environnement d'installation en leur attribuant des valeurs dans le fichier `pkginfo`. Ces variables d'environnement doivent être alphanumériques et commencer par une majuscule. Ces variables d'environnement peuvent être modifiées par un script `request` ou un script `checkinstall`.

- Le script `request` et le script `checkinstall` peuvent tous deux définir des variables d'environnement en leur attribuant des valeurs et en les plaçant dans l'environnement d'installation.
- Le tableau suivant répertorie les variables d'environnement mises à la disposition de tous les scripts d'installation via l'environnement. Aucune de ces variables d'environnement ne peut être modifiée par un script.

Variable d'environnement	Description
`CLIENT_BASEDIR`	Répertoire de base pour le système cible. Alors que `BASEDIR` est la variable à utiliser pour faire référence à un package spécifique du système d'installation (un serveur dans la majorité des cas), `CLIENT_BASEDIR` est le chemin à inclure dans les fichiers placés sur le système client. `CLIENT_BASEDIR` existe si `BASEDIR` existe et est identique à `BASEDIR` en l'absence de `PKG_INSTALL_ROOT`.
`INST_DATADIR`	Répertoire dans lequel le package lu se trouve. Si le package est lu à partir d'une bande, cette variable correspond à l'emplacement d'un répertoire temporaire dans lequel le package a été transféré au format répertoire. En d'autres termes, en supposant que le nom du package ne porte pas d'extension (par exemple, `SUNWstuff.d`), le script `request` du package actuel se trouve dans `$INST_DATADIR/$PKG/install`.
`PATH`	Liste de recherche utilisée par `sh` pour trouver des commandes à l'appel d'un script. La variable `PATH` est habituellement définie sur `/sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin`.
`PKGINST`	Identificateur d'instance du package installé. Si aucune autre instance du package n'est installée, la valeur correspond à l'abréviation du package (par exemple, `SUNWcadap`). Sinon, la valeur correspond à l'abréviation du package auquel est ajouté un suffixe, comme par exemple `SUNWcadap.4`.
`PKGSAV`	Répertoire dans lequel les fichiers peuvent être enregistrés afin d'être utilisés par les scripts de suppression ou dans lequel des fichiers précédemment enregistrés se trouvent. Uniquement disponible dans la version 2.5 de Solaris et les versions compatibles.
`PKG_CLIENT_OS`	Système d'exploitation du client sur lequel le package doit être installé. La valeur de cette variable est `Solaris`.
`PKG_CLIENT_VERSION`	Version de Solaris au format `x.y`.
`PKG_CLIENT_REVISION`	Révision de la version Solaris.
`PKG_INSTALL_ROOT`	Système de fichiers racine du système cible sur lequel le package doit être installé. Cette variable n'existe que si les commandes `pkgadd` et `pkgrm` ont été appelées avec l'option `-R`. Cette existence conditionnelle simplifie son utilisation dans les scripts de procédure sous la forme `${PKG_INSTALL_ROOT}`*/cheminquelconque*.
`PKG_NO_UNIFIED`	Variable d'environnement définie si les commandes `pkgadd` et `pkgrm` ont été appelées avec les options `-M` et `-R`. Cette variable d'environnement est transmise à tout script d'installation de package ou commande de package faisant partie de l'environnement du package.

Variable d'environnement	Description
UPDATE	Cette variable d'environnement est absente de la plupart des environnements d'installation. Si cette variable existe (valeur yes), elle signifie l'une des deux choses suivantes : Un package de même nom, de même version et de même architecture est déjà installé sur le système. Ou, ce package remplace un package déjà installé de même nom, sous la direction de l'administrateur. Dans ces situations, le répertoire de base d'origine est toujours utilisé.

Obtention d'informations sur un package pour un script

Deux commandes peuvent être utilisées à partir de scripts pour demander des informations sur un package :

- La commande `pkginfo` renvoie des informations sur les packages, notamment l'identificateur de l'instance et le nom du package.
- La commande `pkgparam` renvoie des valeurs pour les variables d'environnement demandées.

  Reportez-vous aux pages de manuel `pkginfo(1)` et `pkgparam(1)`, ainsi qu'au Chapitre 4, “Vérification et transfert d'un package” pour plus d'informations.

Codes de sortie des scripts

Chaque script doit quitter son exécution avec l'un des codes de sortie figurant dans le tableau suivant.

TABLEAU 3–2 Codes de sortie des scripts d'installation

Code	Signification
0	Script exécuté avec succès.
1	Erreur fatale. La procédure d'installation est annulée à ce moment-là.
2	Avertissement ou cas d'erreur. L'installation se poursuit. Un message d'avertissement s'affiche en fin de procédure.
3	Arrêt net de la commande `pkgadd`. Seul le script `checkinstall` renvoie ce code.
10	Le système doit être réinitialisé à l'issue de l'installation de tous les packages sélectionnés. (Cette valeur doit être ajoutée à l'un des codes de sortie à un seul chiffre.)

TABLEAU 3–2 Codes de sortie des scripts d'installation *(Suite)*

Code	Signification
20	Le système doit être immédiatement réinitialisé à l'issue de l'installation du package actuel. (Cette valeur doit être ajoutée à l'un des codes de sortie à un seul chiffre.)

Reportez-vous au Chapitre 5, "Création d'un package : Études de cas" pour consulter des exemples de codes de sortie renvoyés par les scripts d'installation.

Remarque – Tous les scripts d'installation fournis avec votre package doivent disposer d'une entrée dans le fichier `prototype`. Le type de fichier doit être `i` (script d'installation de package).

Rédaction d'un script `request`

Le script `request` est le seul moyen d'interaction directe avec l'administrateur chargé d'installer le package. Ce script peut être utilisé par exemple pour demander à l'administrateur s'il souhaite installer certains des éléments facultatifs du package.

Le résultat du script `request` doit être une liste de variables d'environnement et de leurs valeurs. Cette liste peut inclure tout paramètre créé dans le fichier `pkginfo`, ainsi que les paramètres `CLASSES` et `BASEDIR`. La liste peut également introduire des variables d'environnement non définies ailleurs. Toutefois et dans la mesure du possible, les valeurs par défaut.le doivent toujours être fournies par le fichier `pkginfo` . Pour plus d'informations sur les variables d'environnement d'un package, reportez-vous à "Variables d'environnement d'un package" à la page 24.

Lorsque votre script `request` attribue des valeurs à une variable d'environnement, il doit par la suite mettre ces valeurs à la disposition de la commande `pkgadd` et des autres scripts du package.

Comportements du script `request`

- Le script `request` ne peut modifier aucun fichier. Ce script ne dialogue qu'avec l'administrateur qui installe le package et crée une liste d'attribution de variables d'environnement basée sur cette interaction. Le script `request` s'exécute en tant qu'utilisateur non privilégié `install` si ce dernier existe. Si tel n'est pas le cas, le script est exécuté en tant qu'utilisateur `root`.
- La commande `pkgadd` appelle le script `request` avec un argument nommant le fichier réponse du script. Le fichier réponse stocke les réponses de l'administrateur.
- Le script `request` n'est pas exécuté pendant la suppression du package. Toutefois, les variables d'environnement attribuées par le script sont enregistrées et disponibles au cours de la suppression du package.

Règles de conception pour les scripts `request`

- Un seul script `request` est autorisé par package. Le script doit être nommé `request`.
- Les attributions de variables d'environnement doivent être ajoutées à l'environnement d'installation pour que la commande `pkgadd` et d'autres scripts d'empaquetage puissent les utiliser en les enregistrant dans le fichier réponse (connu du script sous le nom `$1`).
- Les variables d'environnement système et les variables d'environnement d'installation standard, à l'exception des paramètres `CLASSES` et `BASEDIR`, ne peuvent pas être modifiées par un script `request`. Toute variable d'environnement créée par vous peut être modifiée.

  Remarque – Un script `request` peut modifier le paramètre `BASEDIR` depuis la version 2.5 de Solaris et versions compatibles uniquement.

- Une valeur par défaut doit être attribuée dans le fichier `pkginfo` à toute variable d'environnement susceptible d'être manipulée par le script `request`.
- Le format de la liste de résultats est *PARAM=valeur*. Exemple :

  ```
  CLASSES=none class1
  ```

- Le terminal de l'administrateur est défini comme entrée standard pour le script `request`.
- N'effectuez pas d'analyses spéciales du système cible dans un fichier `request`. Il est risqué de tester le système pour déterminer l'éventuelle présence de certains fichiers binaires ou de certains comportements, et de définir des variables d'environnement en fonction de cette analyse. Il n'existe aucune garantie quant à l'exécution du script `request` lors de la phase d'installation. L'administrateur chargé d'installer le package peut fournir un fichier réponse dont le rôle est d'insérer les variables d'environnement sans jamais appeler le script `request`. Si le script `request` évalue aussi le système de fichiers cible, cette évaluation risque de ne pas avoir lieu. Il est recommandé de confier l'analyse du système cible à des fins particulières au script `checkinstall`.

Remarque – Si les administrateurs chargés d'installer votre package utilisent le produit JumpStart™, l'installation du package ne doit pas être interactive. Vous pouvez au choix, ne pas fournir de script `request` avec votre package ou, indiquer aux administrateurs qu'ils doivent utiliser la commande `pkgask` avant l'installation. La commande `pkgask` stocke les réponses fournies au script `request`. Pour plus d'informations sur la commande `pkgask`, reportez-vous à la page de manuel `pkgask(1M)`.

▼ Procédure de rédaction d'un script `request`

1. **Faites du répertoire contenant vos fichiers d'information votre répertoire de travail actuel.**
2. **Créez un fichier nommé `request` à l'aide de votre éditeur de texte.**
3. **Enregistrez vos modifications puis quittez l'éditeur.**
4. **Effectuez l'une des opérations suivantes :**
   - Pour créer des scripts d'installation supplémentaires, passez à l'opération suivante, "Procédure de recueil de données d'un système de fichiers" à la page 68.
   - Si vous n'avez pas créé de fichier `prototype`, suivez la procédure "Procédure de création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 43. Passez à l'Étape 5.
   - Si vous avez déjà créé un fichier `prototype`, modifiez-le en ajoutant une entrée pour le script d'installation qui vient d'être créé.
5. **Créez votre package.**

   Si nécessaire, reportez-vous à la rubrique "Procédure de création d'un package" à la page 47.

Exemple 3–5 Rédaction d'un script `request`

Lorsqu'un script `request` attribue des valeurs à des variables d'environnement, il doit mettre ces valeurs à la disposition de la commande `pkgadd`. Cet exemple illustre le segment d'un script `request` qui effectue cette opération pour les quatre variables d'environnement : `CLASSES`, `NCMPBIN`, `EMACS` et `NCMPMAN`. Supposons que ces variables ont été définies lors d'une session interactive avec l'administrateur au début du script.

```
# 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
!
```

Voir aussi Une fois le package créé, installez-le pour confirmer qu'il s'installe correctement et vérifier son intégrité. Le Chapitre 4, "Vérification et transfert d'un package" vous explique comment vérifier l'intégrité du package et décrit sa procédure de transfert sur un support de distribution.

Recueil de données d'un système de fichiers à l'aide du script `checkinstall`

Le script `checkinstall` est exécuté peu de temps après le script `request` facultatif. Le script `checkinstall` s'exécute en tant qu'utilisateur `install`, si celui-ci existe, ou en tant qu'utilisateur `nobody`. Le script `checkinstall` ne dispose pas des droits nécessaires pour modifier les données du système de fichiers. Il peut cependant, en fonction des informations qu'il recueille, créer ou modifier des variables d'environnement afin de contrôler la procédure d'installation résultante. Le script est également capable de procéder à un arrêt net de la procédure d'installation.

Le script `checkinstall` a pour rôle d'effectuer des contrôles de base sur un système de fichiers, contrôles inhabituels pour la commande `pkgadd`. Ce script peut par exemple être utilisé pour déterminer à l'avance si certains fichiers du package actuel vont écraser des fichiers existants, ou pour gérer les dépendances globales des logiciels. Le fichier `depend` ne gère que les dépendances au niveau du package.

À l'inverse du script `request`, le script `checkinstall` est exécuté qu'un fichier réponse soit fourni ou non. La présence du script ne qualifie pas le package d'interactif. Le script `checkinstall` peut être des situations où un script `request` est interdit ou que l'interaction avec l'administrateur n'est pas pratique.

Remarque – Le script `checkinstall` est disponible depuis la version 2.5 de Solaris et versions compatibles.

Comportements du script `checkinstall`

- Le script `checkinstall` ne peut modifier aucun fichier. Ce script analyse simplement l'état du système et crée une liste d'attribution de variables d'environnement basée sur cette interaction. Afin d'appliquer cette restriction, le script `request` est exécuté en tant qu'utilisateur non privilégié `checkinstall`, si celui-ci existe. Sinon, le script est exécuté en tant qu'utilisateur non privilégié `nobody`. Le script `checkinstall` ne dispose pas des droits de superutilisateur.
- La commande `pkgadd` appelle le script `checkinstall` avec un argument nommant le fichier réponse du script. Le fichier réponse du script est le fichier dans lequel les réponses de l'administrateur sont stockées.
- Le script `checkinstall` n'est pas exécuté pendant la suppression du package. Toutefois, les variables d'environnement attribuées par le script sont enregistrées et disponibles au cours de la suppression du package.

Règles de conception pour les scripts `checkinstall`

- Un seul script `checkinstall` est autorisé par package. Le script doit être nommé `checkinstall`.
- Les attributions de variables d'environnement doivent être ajoutées à l'environnement d'installation pour que la commande `pkgadd` et d'autres scripts d'empaquetage puissent les utiliser en les enregistrant dans le fichier réponse (connu du script sous le nom `$1`).
- Les variables d'environnement système et les variables d'environnement d'installation standard, à l'exception des paramètres `CLASSES` et `BASEDIR`, ne peuvent pas être modifiées par un script `checkinstall`. Toute variable d'environnement créée par vous peut être modifiée.
- Une valeur par défaut doit être attribuée dans le fichier `pkginfo` à toute variable d'environnement susceptible d'être manipulée par le script `checkinstall`.
- Le format de la liste de résultats est *PARAM=valeur*. Exemple :

  ```
  CLASSES=none class1
  ```

- L'interaction avec l'administrateur n'est pas autorisée pendant l'exécution d'un script `checkinstall`. L'interaction avec l'administrateur se limite au script `request`.

▼ Procédure de recueil de données d'un système de fichiers

1 Faites du répertoire contenant vos fichiers d'information votre répertoire de travail actuel.

2 Créez un fichier nommé `checkinstall` à l'aide de votre éditeur de texte.

3 Enregistrez vos modifications puis quittez l'éditeur.

4 Effectuez l'une des opérations suivantes :

- Pour créer des scripts d'installation supplémentaires, passez à l'étape suivante, "Procédure de rédaction de scripts de procédure" à la page 71.
- Si vous n'avez pas créé de fichier `prototype`, suivez la procédure "Procédure de création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 43. Passez à l'Étape 5.
- Si vous avez déjà créé un fichier `prototype`, modifiez-le en ajoutant une entrée pour le script d'installation qui vient d'être créé.

5 Créez votre package.

Si nécessaire, reportez-vous à la rubrique "Procédure de création d'un package" à la page 47.

Exemple 3–6 Rédaction d'un script `checkinstall`

Cet exemple de script `checkinstall` vérifie que le logiciel de base de données requis par le package `SUNWcadap` est installé.

```
# 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
```

Voir aussi Une fois le package créé, installez-le pour confirmer qu'il s'installe correctement et vérifier son intégrité. Le Chapitre 4, “Vérification et transfert d'un package” vous explique comment vérifier l'intégrité du package et décrit sa procédure de transfert sur un support de distribution.

Rédaction de scripts de procédure

Les scripts de procédure fournissent la liste d'instructions à suivre à certains stades de l'installation ou de la suppression d'un package. Les quatre scripts de procédure doivent porter un des noms prédéfinis liés au stade de l'exécution des instructions. Les scripts sont exécutés sans arguments.

- Script `preinstall`

  Ce script est exécuté avant le début de l'installation des classes. Ce script ne doit installer aucun fichier.
- Script `postinstall`

  Ce script est exécuté une fois tous les volumes installés.
- Script `preremove`

  Ce script est exécuté avant le début de la suppression des classes. Ce script ne doit supprimer aucun fichier.
- Script `postremove`

  Ce script est exécuté après la suppression de toutes les classes.

Comportements des scripts de procédure

Les scripts de procédure sont exécutés en tant que `uid=root` et `gid=other`.

Règles de conception des scripts de procédure

- Chaque script doit pouvoir être exécuté plusieurs fois puisqu'il est exécuté pour chaque volume d'un package. L'exécution répétée d'un script à partir de la même entrée produit ainsi les mêmes résultats que si le script était exécuté une seule fois.
- Chaque script de procédure installant un objet de package dans un fichier autre que `pkgmap` doit faire appel à la commande `installf` pour avertir la base de données du package qu'il ajoute ou modifie un nom de chemin. Une fois tous les ajouts et modifications terminés, cette commande doit être appelée avec l'option `-f`. Seuls les scripts `postinstall` et `postremove` peuvent installer les objets de package de cette façon. Reportez-vous à la page de manuel `installf`(1M) et au Chapitre 5, “Création d'un package : Études de cas” pour plus d'informations.
- L'interaction avec l'administrateur n'est pas autorisée lors de l'exécution d'un script de procédure. L'interaction avec l'administrateur se limite au script `request`.
- Chaque script de procédure supprimant des fichiers non installés du fichier `pkgmap` doit se servir de la commande `removef` pour avertir la base de données du package qu'il supprime un nom de chemin. Une fois la suppression terminée, cette commande doit être appelée avec l'option `-f`. Reportez-vous à la page de manuel `removef`(1M) et au Chapitre 5, “Création d'un package : Études de cas” pour plus d'informations et quelques exemples.

Remarque – Les commandes `installf` et `removef` doivent être utilisées car les scripts de procédure ne sont pas automatiquement associés aux noms de chemin répertoriés dans le fichier `pkgmap`.

▼ Procédure de rédaction de scripts de procédure

1 **Faites du répertoire contenant vos fichiers d'information votre répertoire de travail actuel.**

2 **Créez un ou plusieurs scripts de procédure à l'aide de votre éditeur de texte.**

Un script de procédure doit porter l'un des noms prédéfinis suivants : `preinstall`, `postinstall`, `preremove` ou `postremove`.

3 **Enregistrez les modifications et quittez l'éditeur.**

4 **Effectuez l'une des opérations suivantes :**

- Pour créer des scripts d'action de classe, passez à l'opération suivante, "Procédure de rédaction de scripts d'action de classe" à la page 79.
- Si vous n'avez pas créé de fichier `prototype`, suivez la procédure "Procédure de création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 43. Passez à l'Étape 5.
- Si vous avez déjà créé un fichier `prototype`, modifiez-le en ajoutant une entrée pour chaque script d'installation qui vient d'être créé.

5 **Créez votre package.**

Si nécessaire, reportez-vous à la rubrique "Procédure de création d'un package" à la page 47.

Voir aussi Une fois le package créé, installez-le pour confirmer qu'il s'installe correctement et vérifier son intégrité. Le Chapitre 4, "Vérification et transfert d'un package" vous explique comment vérifier l'intégrité du package et décrit sa procédure de transfert sur un support de distribution.

Rédaction de scripts d'action de classe

Définition de classes d'objets

Les classes d'objets permettent d'effectuer une série d'opérations sur un groupe d'objets de package lors de l'installation ou de la suppression. Vous attribuez des objets à une classe dans le fichier `prototype`. Une classe doit être attribuée à tous les objets de package, bien que la classe `none` soit utilisée par défaut pour les objets ne requérant aucune opération particulière.

Le paramètre d'installation `CLASSES` défini dans le fichier `pkginfo` correspond à la liste de classes à installer (y compris la classe `none`).

Remarque – Les objets définis dans le fichier `pkgmap` appartenant à une classe non répertoriée dans ce paramètre du fichier `pkginfo` ne sont *pas* installés.

La liste `CLASSES` détermine l'ordre d'installation. La classe `none`, lorsqu'elle est présente, est toujours installée en premier et supprimée en dernier. Étant donné que les répertoires représentent la structure de prise en charge élémentaire de tous les autres objets d'un système de fichiers, ils doivent tous être attribués à la classe `none`. Des exceptions peuvent être faites mais en règle générale, la classe `none` est la plus sûre. Cette stratégie garantit que les répertoires sont créés avant les objets qu'ils contiennent. D'autre part, aucune tentative de suppression d'un répertoire n'est faite avant qu'il n'ait été vidé.

Traitement des classes pendant l'installation d'un package

La section suivante décrit le déroulement des opérations système lors de l'installation d'une classe. Les opérations sont répétées une fois pour chaque volume d'un package à l'installation du volume en question.

1. La commande `pkgadd` crée une liste de noms de chemin.

   La commande `pkgadd` crée une liste de noms de chemin sur lesquels le script d'action opère. Chaque ligne de la liste contient des noms de chemin source et de destination séparés par un espace. Le nom de chemin source indique où l'objet à installer réside sur le volume d'installation. Le nom de chemin de destination indique l'emplacement d'installation de l'objet sur le système cible. Le contenu de la liste est restreint par les critères suivants :

   - La liste ne contient que des noms de chemin appartenant à la classe associée.
   - Si la tentative de création des objets du package échoue, les répertoires, tubes nommés, périphériques en mode caractère, périphériques en mode bloc et liens symboliques sont inclus dans la liste avec comme nom de chemin source, `/dev/null`. En règle générale, ces éléments sont automatiquement créés par la commande `pkgadd` (s'ils n'existent pas encore) et il leur est attribué des attributs propres (mode, owner, group) tels qu'ils sont définis dans le fichier `pkgmap`.
   - Les fichiers liés dont le type est `l` ne sont en aucun cas inclus dans la liste. Les liens physiques de la classe donnée sont créés à l'étape 4.

2. Si aucun script d'action de classe n'est fourni pour l'installation d'une classe spécifique, les noms de chemin figurant dans la liste générée sont copiés du volume à l'emplacement cible approprié.
3. Si un script d'action de classe est présent, il est exécuté.

   Le script d'action de classe est appelé avec l'entrée standard contenant la liste générée à l'étape 1. Si ce volume est le dernier du package ou si cette classe ne contient plus d'objets, le script est exécuté avec le seul argument `ENDOFCLASS`.

Remarque – Même si aucun fichier standard de cette classe ne figure dans le package, le script d'action de classe est appelé au moins une fois avec une liste vide et l'argument `ENDOFCLASS`.

4. La commande `pkgadd` réalise un audit du contenu et des attributs, et crée des liens physiques.

   Une fois l'étape 2 ou l'étape 3 exécutée avec succès, la commande `pkgadd` vérifie les informations du contenu et des attributs de la liste des noms de chemin. La commande `pkgadd` crée automatiquement les liens associés à la classe. Toute incohérence d'attributs des noms de chemin est corrigée dans la liste générée.

Traitement des classes pendant la suppression d'un package

Les objets sont supprimés d'une classe après l'autre. Les classes existant pour un package mais non répertoriées dans le paramètre `CLASSES` sont supprimées en premier (par exemple, un objet installé à l'aide de la commande `installf)`. Les classes répertoriées dans le paramètre `CLASSES` sont supprimées dans l'ordre inverse. La classe `none` est toujours supprimée en dernier. La section suivante décrit le déroulement des opérations système lors de la suppression d'une classe :

1. La commande `pkgrm` crée une liste de noms de chemin.

   La commande `pkgrm` crée une liste des noms de chemin installés appartenant à la classe indiquée. Les noms de chemin auxquels un autre package fait référence sont exclus de la liste à moins que leur type de fichier ne soit `e`. Le type de fichier `e` signifie que le fichier doit être modifié lors de l'installation ou de la suppression.

   Si le package à supprimer a modifié des fichiers de type `e` lors de l'installation, il ne doit supprimer que les lignes qu'il a ajoutées. Ne supprimez pas un fichier modifiable non vide. Supprimez les lignes ajoutées par le package.

2. Si aucun script d'action de classe n'est présent, les noms de chemin sont supprimés.

   Si votre package ne contient aucun script d'action de classe de suppression pour la classe, tous les noms de chemin figurant dans la liste générée par la commande `pkgrm` sont supprimés.

   Remarque – Les fichiers de type `e` (modifiables) ne sont pas attribués à une classe ni à un script d'action de classe associé. Ces fichiers sont à ce stade supprimés, même si le nom de chemin est partagé avec d'autres packages.

3. Si un script d'action de classe est présent, il est exécuté.

   La commande `pkgrm` appelle le script d'action de classe avec une entrée standard pour le script contenant la liste générée à l'étape 1.

4. La commande `pkgrm` réalise un audit.

Après avoir exécuté le script d'action de classe, la commande `pkgrm` supprime les références aux noms de chemin figurant dans la base de données du package, à l'exception de celles auxquelles un autre package fait référence.

Script d'action de classe

Ce script d'action de classe définit un groupe d'opérations à exécuter pendant l'installation ou la suppression d'un package. Les opérations sont effectuées sur un un groupe de noms de chemin d'après leur définition de classe. Reportez-vous au Chapitre 5, “Création d'un package : Études de cas” pour consulter des exemples de scripts d'action de classe.

Le nom d'un script d'action de classe est basé sur la classe à laquelle il s'applique et la phase durant laquelle les opérations ont lieu, à savoir pendant l'installation ou la suppression du package. Les deux formats de nom sont indiqués dans le tableau suivant :

Format du nom	Description
`i.`*classe*	Opère sur des noms de chemin de la classe indiquée pendant l'installation du package.
`r.`*classe*	Opère sur des noms de chemin de la classe indiquée pendant la suppression du package.

Par exemple, le nom du script d'installation d'une classe nommée `manpage` est `i.pmanpage`. Le script de suppression est nommé `r.manpage`.

Remarque – Ce format de nom de fichier n'est pas utilisé pour les fichiers appartenant aux classes système `sed`, `awk` ou `build`. Pour plus d'informations sur ces classes spéciales, reportez-vous à “Classes système spéciales” à la page 75.

Comportements des scripts d'action de classe

- Les scripts d'action de classe sont exécutés en tant que `uid=root` et `gid=other`.
- Un script est exécuté pour tous les fichiers appartenant à la classe donnée sur le volume actuel.
- Les commandes `pkgadd` et `pkgrm` créent la liste de tous les objets répertoriés dans le fichier `pkgmap` qui appartiennent à la classe. Pour cette raison, un script d'action de classe ne peut agir que sur les noms de chemin définis dans le fichier `pkgmap` qui appartiennent à une classe particulière.
- Lorsqu'un script d'action de classe est exécuté pour la dernière fois (autrement dit, qu'aucun autre fichier n'appartient à cette classe), il est exécuté une fois avec l'argument de mots clés `ENDOFCLASS`.

- L'interaction avec l'administrateur n'est pas autorisée lors de l'exécution d'un script d'action de classe.

Règles de conception des scripts d'action de classe

- Si l'installation d'un package a nécessité plusieurs volumes, le script d'action de classe est exécuté une fois sur chaque volume contenant au moins un fichier appartenant à une classe. Chaque script doit pour cette raison pouvoir être exécuté plusieurs fois. L'exécution répétée d'un script à partir de la même entrée doit produire les mêmes résultats que si le script était exécuté une seule fois.
- Lorsqu'un fichier appartient à une classe associée à un script d'action de classe, le script doit installer le fichier. La commande `pkgadd` n'installe pas les fichiers pour lesquels un script d'action de classe existe, bien qu'elle vérifie leur installation.
- Un script d'action de classe ne doit jamais ajouter, supprimer, ni modifier un nom de chemin ou un attribut système qui ne figure pas dans la liste générée par la commande `pkgadd`. Pour plus d'informations sur cette liste, reportez-vous à l'étape 1 de "Traitement des classes pendant l'installation d'un package" à la page 72.
- Lorsque votre script détecte l'argument `ENDOFCLASS`, placez des opérations posttraitement dans votre script, telle une opération de nettoyage.
- L'interaction avec l'administrateur se limite au script `request`. Ne tentez pas d'obtenir des informations auprès de l'administrateur à l'aide d'un script d'action de classe.

Classes système spéciales

Le système fournit quatre classes spéciales :

- Classe `sed`

  Offre un moyen d'utiliser les instructions `sed` pour modifier des fichiers lors de l'installation et de la suppression d'un package.

- Classe `awk`

  Offre un moyen d'utiliser les instructions `awk` pour modifier des fichiers lors de l'installation et de la suppression d'un package.

- Classe `build`

  Offre un moyen de créer ou de modifier un fichier dynamiquement à l'aide de commandes Bourne shell.

- La classe `preserve`

  Offre un moyen de conserver des fichiers qui ne doivent pas être remplacés lors de futures installations de packages.

- La classe `manifest`

  Permet l'installation et la désinstallation automatisées des services SMF associés à un manifeste. La classe `manifest` doit être utilisée pour tous les manifestes SMF d'un package.

Si plusieurs fichiers d'un package nécessitent un traitement spécial pouvant être intégralement défini par les commandes `sed`, `awk` ou `sh`, l'installation est plus rapide avec les classes système qu'avec plusieurs classes et les scripts d'action de classe qui leur sont associés.

Script de classe `sed`

La classe `sed` offre un moyen de modifier un objet existant sur le système cible. Le script d'action de classe `sed` s'exécute automatiquement à l'installation en présence d'un fichier appartenant à la classe `sed`. Le nom du script d'action de classe `sed` doit être identique au nom du fichier sur lequel les instructions sont exécutées.

Un script d'action de classe `sed` fournit des instructions `sed` au format suivant :

Deux commandes indiquent à quel moment les instructions doivent être exécutées. Les instructions `sed` qui suivent la commande `!install` sont exécutées pendant l'installation du package. Les instructions `sed` qui suivent la commande `!remove` sont exécutées pendant la suppression du package. L'ordre dans lequel les commandes sont utilisées dans le fichier n'est pas important.

Pour plus d'informations sur les instructions `sed`, reportez-vous à la page de manuel `sed(1)` Pour consulter des exemples de scripts d'action de classe `sed`, reportez-vous au Chapitre 5, “Création d'un package : Études de cas”.

Script de classe `awk`

La classe `awk` offre un moyen de modifier un objet existant sur le système cible. Les modifications sont fournies sous forme d'instructions `awk` dans un script d'action de classe `awk`.

Le script d'action de classe `awk` s'exécute automatiquement à l'installation en présence d'un fichier appartenant à la classe `awk`. Un tel fichier contient des instructions pour un script de classe `awk` au format suivant :

Deux commandes indiquent à quel moment les instructions doivent être exécutées. Les instructions `awk` qui suivent la commande `!install` sont exécutées pendant l'installation du package. Les instructions qui suivent la commande `!remove` sont exécutées pendant la suppression du package. Ces commandes peuvent être utilisées dans un ordre quelconque.

Le nom du script d'action de classe `awk` doit être identique au nom du fichier sur lequel les instructions sont exécutées.

Le fichier à modifier est utilisé comme entrée de la commande `awk` et le résultat du script remplace en fin de compte l'objet d'origine. Aucune variable d'environnement ne peut être transmise à la commande `awk` avec cette syntaxe.

Pour plus d'informations sur les instructions `awk`, reportez-vous à la page de manuel `awk(1)`.

Script de classe `build`

La classe `build` crée ou modifie un fichier d'objets de package en exécutant des instructions Bourne shell. Ces instructions sont fournies en tant qu'objet de package. Les instructions s'exécutent automatiquement lors de l'installation si l'objet de package appartient à la classe `build`.

Le nom du script d'action de classe `build` doit être identique au nom du fichier sur lequel les instructions sont exécutées. Le nom doit également pouvoir être exécuté par la commande `sh`. Le résultat du script devient la nouvelle version du fichier à mesure qu'il est créé ou modifié. Si le script ne produit aucun résultat, le fichier n'est ni créé ni modifié. De ce fait, le script peut modifier ou créer le fichier.

Par exemple, si un package fournit un fichier par défaut, `/etc/randomtable`, qui n'existe pas encore sur le système cible, l'entrée du fichier `prototype` peut être comme suit :

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

L'objet de package, `/etc/randomtable`, peut être comme suit :

```
!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
```

Reportez-vous au Chapitre 5, "Création d'un package : Études de cas" pour consulter un autre exemple utilisant la classe `build`.

Script de classe `preserve`

La classe `preserve` conserve un fichier d'objets de package en déterminant si un fichier existant doit ou non être remplacé à l'installation du package. Deux des scénarios possibles avec l'utilisation du script de classe `preserve` sont :

- Si le fichier à installer n'existe pas encore dans le répertoire cible, il est installé de la manière habituelle.
- Si le fichier à installer se trouve déjà dans le répertoire cible, un message s'affiche pour vous en avertir et le fichier n'est pas installé.

Le résultat des deux scénarios est considéré comme positif par le script `preserve`. Un message d'erreur s'affiche uniquement dans le deuxième scénario si le fichier ne peut pas être copié dans le répertoire cible.

À compter de la version 7 de Solaris, le script `i.preserve` et une copie de ce script, `i.CONFIG.prsv`, sont fournis dans le répertoire `/usr/sadm/install/scripts` avec les autres scripts d'action de classe.

Modifiez le script pour indiquer le nom du ou des fichiers à conserver.

Script de classe `manifest`

La classe `manifest` installe et désinstalle automatiquement les services SMF (Service Management Facility) associés à un manifeste SMF. Si vous n'êtes pas familier avec SMF, reportez-vous au Chapitre 17, “Managing Services (Overview)” du *System Administration Guide: Basic Administration* pour plus d'informations sur la gestion des services à l'aide de SMF.

Tous les manifestes de service contenus dans les packages doivent être identifiés avec la classe `manifest`. Les scripts d'action de classe qui installent et suppriment les manifestes de service sont inclus dans le sous-système du package. Lorsque la commande `pkgadd`(1M) est invoquée, le manifeste du service est importé. Lorsque la commande `pkgrm`(1M) est invoquée, les instances contenues dans le manifeste du service qui sont désactivées sont supprimées. Tous les services contenus dans le manifeste et pour lesquels il n'existe plus aucune instance sont également supprimés. Si l'option `-R` est ajoutée à la commande `pkgadd`(1M) ou `pkgrm`(1M), ces actions de manifeste de service sont effectuées à la prochaine réinitialisation du système avec cet autre chemin racine.

La portion de code suivante provient d'un fichier d'information de package et illustre l'utilisation de la classe `manifest`.

```
# packaging files
i pkginfo
i copyright
i depend
i preinstall
i postinstall
i i.manifest
i r.manifest
#
# 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
```

▼ Procédure de rédaction de scripts d'action de classe

1 **Faites du répertoire contenant vos fichiers d'information votre répertoire de travail actuel.**

2 **Donnez aux objets du package le nom de classe souhaité dans le fichier** `prototype`**.**

Par exemple, l'attribution des noms de classe `application` et `manpage` à des objets est comme suit :

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

3 **Modifiez le paramètre** `CLASSES` **figurant dans le fichier** `pkginfo` **afin qu'il contienne les noms de classe à utiliser dans votre package.**

Par exemple, des entrées pour les classes `application` et `manpage` sont comme suit :

```
CLASSES=manpage application none
```

Remarque – La classe `none` est toujours installée en premier et supprimée en dernier, indépendamment de son emplacement dans la définition du paramètre `CLASSES`.

---

4 **Si vous créez un script d'action de classe pour un fichier appartenant à la classe** `sed`**,** `awk` **ou** `build`**, faites du répertoire contenant l'objet de package votre répertoire de travail.**

5 **Créez les scripts d'action de classe ou les objets de package (pour les fichiers appartenant à la classe** `sed`**,** `awk` **ou** `build`**).**

Par exemple, un script d'installation pour une classe nommée `application` doit être nommé `i.application` et un script de suppression, `r.application`.

N'oubliez pas que lorsqu'un fichier appartient à une classe associée à un script d'action de classe, le script doit installer le fichier. La commande `pkgadd` n'installe pas les fichiers pour lesquels un script d'action de classe existe, bien qu'elle vérifie leur installation. De plus, si vous définissez une classe sans fournir de script d'action de classe, la seule opération effectuée sur cette classe est la copie de composants, du support d'installation au système cible (comportement `pkgadd` par défaut).

6 **Effectuez l'*une* des opérations suivantes :**

- Si vous n'avez *pas* créé de fichier `prototype`, suivez la procédure "Procédure de création d'un fichier `prototype` à l'aide de la commande `pkgproto`" à la page 43, puis passez à l'Étape 7.
- Si vous avez déjà créé un fichier `prototype`, modifiez-le en ajoutant une entrée pour chaque script d'installation qui vient d'être créé.

7 Créez votre package.

Si nécessaire, reportez-vous à la rubrique "Procédure de création d'un package" à la page 47.

Informations supplémentaires

Autres ressources

Une fois le package créé, installez-le pour confirmer qu'il s'installe correctement et vérifier son intégrité. Le Chapitre 4, "Vérification et transfert d'un package" vous explique comment vérifier l'intégrité du package et décrit sa procédure de transfert sur un support de distribution.

Création de packages signés

La procédure de création de packages signés implique un certain nombre d'étapes et nécessite la compréhension de nouveaux concepts et d'une nouvelle terminologie. Cette section fournit des informations sur les packages signés, la terminologie correspondante et des informations sur la gestion des certificats. Elle fournit également des procédures détaillées relatives à la création d'un package signé.

Packages signés

Un package signé est un package au format flux de données standard contenant une signature numérique (signature numérique PKCS7 au chiffrement PEM définie ci-après) qui garantit les points suivants :

- Le package provient de l'entité qui l'a signé ;
- L'entité l'a en effet signé ;
- Le package n'a pas été modifié depuis que l'entité l'a signé ;
- L'entité qui l'a signé est une entité de confiance.

La seule différence entre un package signé et un package non signé est sa signature ; Un package signé est compatible binaire avec un package non signé. De ce fait, un package signé peut être utilisé avec d'anciennes versions d'outils d'empaquetage. Toutefois, la signature est dans ce cas ignorée.

La technologie d'empaquetage signé emploie une nouvelle terminologie et de nouvelles abréviations décrites dans le tableau suivant :

Terme	Définition
ASN.1	Langage de description de syntaxe abstraite (numéro 1) : Méthode d'expression d'objets abstraits. Par exemple, le langage ASN.1 définit un certificat à clé publique, l'ensemble des objets composant le certificat, et l'ordre dans lequel les objets sont recueillis. Toutefois, le langage ASN.1 n'indique pas la manière dont les objets sont sérialisés à des fins de stockage ou de transmission.
X.509	Norme X.509 de l'UIT-T : Spécifie la syntaxe X.509 très répandue des certificats à clés publiques.
DER	Distinguished Encoding Rules : Représentation binaire d'un objet ASN.1 qui définit la manière dont un objet ASN.1 est sérialisé à des fins de stockage ou de transmission dans les environnements informatiques.
PEM	Privacy Enhanced Message : Technique de chiffrement de fichier (au format DER ou autre format binaire) utilisant le codage 64 de base et quelques en-têtes facultatifs. La technique PEM était à l'origine utilisée pour le chiffrement d'e-mails de type MIME. La technique PEM est également largement utilisée pour le chiffrement des certificats et des clés privées d'un fichier stocké sur un système de fichiers ou joint à un message électronique.
PKCS7	Norme de chiffrement par clé publique numéro 7 : Norme décrivant la syntaxe générale des données qu'il est possible de chiffrer, notamment les signatures numériques et les enveloppes numériques. Un package signé contient une signature PKCS7 incorporée. Cette signature contient au minimum la synthèse du package, ainsi que le certificat à clés publiques X.509 du signatatire. Le package signé peut également contenir des certificats à chaînes. Les certificats à chaînes peuvent être utilisés lors de la formation d'une chaîne de confiance, du certificat du signataire à un certificat de confiance stocké localement.
PKCS12	Norme de chiffrement par clé publique numéro 12 : Norme décrivant la syntaxe du stockage d'objets de chiffrement sur disque. Le keystore de package utilise ce format.
Keystore de package	Référentiel de certificats et de clés pouvant être interrogé par les outils d'un package.

Gestion de certificats

Avant de créer un package signé, vous devez disposer d'un keystore de package. Ce keystore de package contient des certificats sous la forme d'objets. Un keystore de package contient deux types d'objets :

Certificat de confiance — Certificat de confiance contenant un seul certificat à clé publique appartenant à une autre entité. Le certificat de confiance est nommé

	ainsi parce que le propriétaire du keystore espère que la clé publique contenue dans le certificat appartient en effet à l'entité indiquée par le sujet (propriétaire) du certificat. L'émetteur du certificat garantit cette confiance en signant le certificat. Les certificats de confiance sont utilisés lors de la vérification des signatures et lors de l'initialisation d'une connexion à un serveur sécurisé (SSL).
Clé utilisateur	Une clé utilisateur renferme toutes les informations sensibles de la clé de chiffrement. Ces informations sont stockées dans un format protégé pour empêcher tout accès non autorisé. Une clé utilisateur se compose de la clé privée d'un utilisateur et du certificat à clé publique correspondant à la clé privée. Les clés utilisateur sont utilisées dans le cadre de la création d'un package signé.

Par défaut, le keystore du package est stocké dans le répertoire `/var/sadm/security`. Les utilisateurs peuvent disposer de leur propre keystore stocké par défaut dans le répertoire `$HOME/.pkg/security`.

Sur disque, un keystore de package peut utiliser deux formats : un format multifichier et un format monofichier. Un format multifichier stocke ses objets dans plusieurs fichiers. Chaque type d'objet est stocké dans un fichier distinct. Tous ces fichiers doivent être chiffrés à l'aide de la même phrase de passe. Un keystore monofichier stocke tous ses objets dans un seul fichier dans le système de fichiers.

Le principal utilitaire servant à gérer les certificats et le keystore du package est la commande `pkgadm`. Les sous-sections suivantes décrivent les opérations de gestion du keystore du package les plus courantes.

Ajout de certificats de confiance au keystore du package

Vous pouvez ajouter un certificat de confiance au keystore du package à l'aide de la commande `pkgadm`. Le certificat peut être au format PEM ou DER. Exemple :

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

Dans cet exemple, le certificat au format PEM appelé `moncertdeconf.pem` est ajouté au keystore du package.

Ajout d'un certificat utilisateur et d'une clé privée au keystore du package

La commande `pkgadm` ne génère pas de certificats utilisateur ni de clés privées. Les certificats utilisateur et les clés privées sont en général obtenus auprès d'une autorité de certification telle que Verisign. Ils sont aussi générés localement en tant que certificat autosigné. Une fois la clé et le certificat obtenus, vous pouvez les importer dans le keystore du package à l'aide de la commande `pkgadm`. Exemple :

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

Dans cet exemple, les options suivantes sont utilisées :

`-n` *monnom*	Identifie l'entité (*monnom*) du keystore du package sur laquelle opérer. L'entité *monnom* devient l'alias sous lequel les objets sont stockés.
`-e /tmp/`*maclépriv.pem*	Spécifie le fichier contenant la clé privée. Dans ce cas, le fichier est *maclépriv.pem*, stocké dans le répertoire `/tmp`.
`/tmp/`*moncertpub.pem*	Indique le fichier de certificat au format PEM appelé *moncertpub.pem*.

Vérification du keystore du package

La commande `pkgadm` sert également à afficher le contenu du keystore du package. Exemple :

```
$ pkgadm listcert
```

Cette commande affiche les certificats de confiance et clés privées contenus dans le keystore du package.

Suppression de certificats de confiance et de clés privées du keystore d'un package

La commande `pkgadm` peut être utilisée pour supprimer des certificats de confiance et des clés privées du keystore du package.

Lorsque vous supprimez des certificats utilisateur, l'alias de la paire certificat/clé doit être spécifié. Exemple :

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

L'alias du certificat correspond au nom usuel du certificat, qui peut être identifié à l'aide de la commande `pkgadm listcert`. Par exemple, la commande suivante supprime un certificat de confiance appelé Trusted CA Cert 1 :

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

Remarque – Si un certificat de confiance et un certificat utilisateur sont stockés sous le même nom d'alias, ils sont tous deux supprimés lorsque vous spécifiez l'option `-n`.

Création de packages signés

La procédure de création de packages signés se déroule en trois étapes :

1. Création d'un package non signé au format répertoire.
2. Importation du certificat de signature, des certificats AC et de la clé privée dans le keystore du package.
3. Signature du package de l'étape 1 avec les certificats de l'étape 2.

Remarque – Les outils d'empaquetage ne créent pas de certificats. Ces certificats doivent être obtenus auprès d'une autorité de certification telle que Verisign ou Thawte.

Chaque étape de la création de packages signés est décrite dans les procédures suivantes.

▼ Procédure de création d'un package non signé au format répertoire

La procédure de création d'un package non signé au format répertoire est identique à la procédure de création d'un package standard décrite précédemment dans ce manuel. La procédure suivante décrit les étapes de la création d'un package non signé au format répertoire. Pour plus d'informations, reportez-vous aux sections précédentes relatives à la création des packages.

1 Créez le fichier `pkginfo`.

Le fichier `pkginfo` doit avoir le contenu de base suivant :

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

2 Créez le fichier `prototype`.

Le fichier `prototype` doit avoir le contenu de base suivant :

```
$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 Répertoriez le contenu du répertoire source des objets.

Exemple :

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

Le résultat pourrait être comme suit :

```
/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 Créez le package non signé.

```
pkgmk -d ‘pwd‘
```

Le résultat peut être comme suit :

```
## 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.
```

Le package se trouve dorénavant dans le répertoire actuel.

▼ Procédure d'importation des certificats dans le keystore du package

Le certificat et la clé privée à importer doivent être un certificat X.509 et une clé privé chiffrés au format PEM ou DER. En outre, tout certificat intermédiaire ou à chaînes liant votre certificat de signature à l'autorité de certification doit être importé dans le keystore du package avant la signature d'un package.

Remarque – Chaque autorité de certification peut établir des certificats dans divers formats. Pour extraire les certificats et la clé privée du fichier PKCS12 et les importer dans le fichier X.509 chiffré au format PEM (approprié à l'importation dans le keystore du package), servez-vous d'un utilitaire de conversion en freeware tel que OpenSSL.

Si votre clé privée est chiffrée (ce qui est habituellement le cas), un message vous invite à saisir la phrase de passe. Un autre message vous invite par ailleurs à saisir un mot de passe afin de protéger le keystore du package résultant. Ce mot de passe n'est pas obligatoire mais sans mot de passe, le keystore du package n'est pas chiffré.

La procédure suivante décrit l'importation des certificats à l'aide de la commande `pkgadm` une fois convertis au format approprié.

1 Importez tous les certificats fournis par l'autorité de certification situés dans votre fichier de certificats X.509 chiffrés au format PEM ou DER.

Par exemple, pour importer tous les certificats de l'autorité de certification situés dans le fichier `ca.pem`, vous devez saisir la commande suivante :

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

Le résultat peut être comme suit :

```
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
```

Pour importer votre clé de signature dans le keystore du package, vous devez fournir un alias qui sera utilisé ultérieurement lors de la signature du package. Cet alias peut également être utilisé pour supprimer la clé du keystore du package.

Par exemple, pour importer votre clé de signature à partir du fichier `sign.pem`, vous devez saisir la commande suivante :

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

Le résultat peut être comme suit :

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

2 **Vérifiez que les certificats se trouvent dans le keystore du package.**

Par exemple, pour afficher les certificats contenus dans le keystore créé à l'étape précédente, vous devez saisir la commande suivante :

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

▼ Procédure de signature du package

Une fois les certificats importés dans le keystore du package, le package peut être signé. La signature du package en tant que telle s'effectue à l'aide de la commande `pkgtrans`.

- **Signez le package à l'aide de la commande `pkgtrans`. Indiquez l'emplacement du package non signé et l'alias de la clé pour signer le package.**

  Par exemple, en partant des exemples des procédures précédentes, vous devez saisir la commande suivante pour créer un package signé appelé `SUNWfoo.signed` :

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

  Le résultat de cette commande peut être comme suit :

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

  Le package signé est créé dans le fichier `SUNWfoo.signed`, au format flux de données de package. Ce package signé peut être copié sur un site Web et, installé à l'aide de la commande `pkgadd` et d'une URL.

◆ ◆ ◆ CHAPITRE 4

Vérification et transfert d'un package

Le présent chapitre vous explique comment vérifier l'intégrité de votre package et le transférer sur un support de distribution, notamment une disquette ou un CD-ROM.

La liste suivante répertorie les informations fournies dans ce chapitre :

Vérification et transfert d'un package (liste de tâches)

Le tableau suivant décrit les étapes à suivre pour vérifier l'intégrité et le transfert de votre package sur un support de distribution.

TABLEAU 4–1 Vérification et transfert d'un package (liste de tâches)

Tâche	Description	Instructions
1. Créez votre package.	Créez votre package sur disque.	Chapitre 2, "Création d'un package"
2. Installez votre package.	Testez votre package en l'installant et en vérifiant que l'installation s'effectue sans erreurs.	"Procédure d'installation d'un package sur un système autonome ou un serveur" à la page 91
3. Vérifiez l'intégrité de votre package.	Utilisez la commande pkgchk pour vérifier l'intégrité de votre package.	"Procédure de vérification de l'intégrité d'un package" à la page 93

TABLEAU 4–1 Vérification et transfert d'un package (liste de tâches) *(Suite)*

Tâche	Description	Instructions
4. Obtenez d'autres informations sur le package.	*Facultatif.* Utilisez les commandes `pkginfo` et `pkgparam` pour effectuer la vérification d'un package.	"Affichage d'informations supplémentaires sur les packages installés" à la page 94
5. Supprimez le package installé.	Utilisez la commande `pkgrm` pour supprimer votre package du système.	"Procédure de suppression d'un package" à la page 100
6. Transférez votre package sur un support distribution.	Utilisez la commande `pkgtrans` pour transférer votre package (au format package) sur un support de distribution.	"Procédure de transfert d'un package sur un support de distribution" à la page 101

Installation de packages logiciels

Les packages logiciels sont installés à l'aide de la commande `pkgadd`. Cette commande transfère le contenu d'un package logiciel, du support de distribution ou du répertoire, sur le système afin de l'installer.

Cette section fournit des instructions de base pour installer votre package afin de vérifier qu'il s' installe correctement.

Base de données des logiciels d'installation

Les informations se rapportant à l'ensemble des packages installés sur un système sont conservées dans la base de données des logiciels d'installation. Chaque objet de package dispose d'une entrée qui fournit des informations le concernant, notamment son nom, son emplacement et son type. Une entrée indique le package auquel l'élément appartient, les autres packages susceptibles de faire référence à l'élément et d'autres informations, notamment le nom de chemin, l'emplacement et le type de l'élément. Les entrées sont automatiquement ajoutées et supprimées par les commandes `pkgadd` et `pkgrm`. Vous pouvez afficher les informations figurant dans la base de données à l'aide des commandes `pkgchk` et `pkginfo`.

Deux types d'information sont associés à chaque élément de package. L'attribut décrit l'élément en soi. Par exemple, les droits d'accès à l'élément, l'ID du propriétaire de l'élément et l'ID du groupe de l'élément sont des attributs. Les informations relatives au contenu décrivent le contenu de l'élément, notamment la taille du fichier et la date/heure de dernière modification.

La base de données des logiciels d'installation indique l'état du package. Un package peut être entièrement installé (la procédure d'installation s'est déroulée correctement) ou partiellement installé (la procédure d'installation ne s'est pas déroulée correctement).

Lorsqu'un package est partiellement installé, cela signifie que certaines parties du package ont été installées avant la suspension de la procédure d'installation, et qu'une partie du package est donc installée et consignée en tant que tel dans la base de données alors que l'autre partie ne l'est

pas. Lorsque vous réinstallez le package, un message vous invite à redémarrer l'installation de l'endroit où elle a été suspendue car la commande `pkgadd` est en mesure d'accéder à la base de données et de détecter les parties déjà installées. Vous pouvez également supprimer les parties déjà installées, en vous basant sur les informations figurant dans la base de données des logiciels d'installation à l'aide de la commande `pkgrm`.

Interaction avec la commande pkgadd

Si la commande `pkgadd` rencontre un problème, elle recherche en premier lieu des instructions dans le fichier d'administration de l'installation. (Voir `admin(4)` pour plus d'informations.) S'il ne contient aucune instruction ou si le paramètre correspondant dans le fichier d'administration a la valeur `ask`, la commande `pkgadd` affiche un message décrivant le problème et contenant une invite. L'invite est habituellement `Do you want to continue with this installation?`. Vous devez répondre à l'invite par `yes`, `no` ou `quit`.

Si vous avez spécifié plusieurs packages, `no` arrête l'installation du package en cours d'installation mais `pkgadd` poursuit l'installation des autres packages. `quit` indique que `pkgadd` doit arrêter l'installation de tous les packages.

Installation de packages sur des systèmes autonomes ou des serveurs dans un environnement homogène

Cette section décrit l'installation des packages sur un système autonome ou un serveur dans un environnement homogène.

▼ Procédure d'installation d'un package sur un système autonome ou un serveur

1. **Créez votre package.**

   Si nécessaire, reportez-vous à “Création d'un package” à la page 45.

2. **Connectez-vous en tant que superutilisateur.**

3. **Ajoutez le package logiciel sur le système.**

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

`-d` *nom-périphérique*	Indique l'emplacement du package. Notez que *nom-périphérique* peut être un nom de chemin d'accès à un répertoire complet ou être les identificateurs d'une bande, d'une disquette ou d'un disque amovible.
pkg-abrév	Correspond au nom d'un ou plusieurs packages à ajouter (séparés par des espaces). En cas d'omission, `pkgadd` installe tous les packages disponibles.

Exemple 4–1 Installation de packages sur des systèmes autonomes et des serveurs

Pour installer un package logiciel nommé `pkgA` à partir d'un périphérique à bande nommé `/dev/rmt/0`, saisissez la commande suivante :

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

Vous pouvez également installer plusieurs packages simultanément, à condition de séparer les noms de package par un espace, comme suit :

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

Si vous ne nommez pas le périphérique sur lequel le package réside, la commande regarde dans le répertoire spool par défaut (`/var/spool/pkg`). Si le package ne s'y trouve pas, l'installation échoue.

Voir aussi Si vous êtes prêt à passer à l'étape suivante, reportez-vous à "Procédure de vérification de l'intégrité d'un package" à la page 93.

Vérification de l'intégrité d'un package

La commande `pkgchk` permet de contrôler l'intégrité des packages, qu'ils soient installés sur un système ou en format package (prêts à être installés avec la commande `pkgadd`). Elle confirme la structure d'un package ou, les fichiers et répertoires installés, ou encore affiche des informations sur les objets du package. La commande `pkgchk` peut répertorier ou vérifier les éléments suivants :

- Les scripts d'installation du package ;
- Le contenu ou les attributs (voire les deux) des objets installés sur le système ;
- Le contenu d'un package non installé et mis en spool ;
- Le contenu ou les attributs (voire les deux) des objets décrits dans le fichier `pkgmap` spécifié.

Pour plus d'informations sur cette commande, reportez-vous à `pkgchk(1M)`.

La commande `pkgchk` effectue deux types de contrôle. Elle contrôle les attributs de fichier (droits d'accès et de propriété d'un fichier et, nombres majeurs/mineurs de périphériques

spéciaux en mode bloc ou en mode caractère) et le contenu des fichiers (taille, somme de contrôle et date de modification). Par défaut, la commande contrôle à la fois les attributs des fichiers et le contenu des fichiers.

La commande `pkgchk` compare également les attributs et le contenu des fichiers du package installé avec la base de données des logiciels d'installation. Les entrées concernant un package peuvent avoir été modifiées depuis l'installation ; par exemple, un autre package peut avoir modifié un élément du package. La base de données reflète ce changement.

▼ Procédure de vérification de l'intégrité d'un package

1 Installez votre package.

Si nécessaire, reportez-vous à "Procédure d'installation d'un package sur un système autonome ou un serveur" à la page 91.

2 Vérifiez l'intégrité de votre package.

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

`-v`	Répertorie les fichiers à mesure qu'ils sont traités.
`-R` *chemin-root*	Spécifie l'emplacement du système de fichiers root du système client.
pkg-abrév	Correspond au nom d'un ou plusieurs packages à vérifier (séparés par des espaces). En cas d'omission, `pkgchk` vérifie tous les packages disponibles.

Exemple 4–2 Vérification de l'intégrité d'un package

L'exemple suivant indique la commande à utiliser pour vérifier l'intégrité d'un package installé.

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

S'il contient des erreurs, la commande `pkgchk` les imprime. Sinon, elle n'imprime rien et renvoie le code de sortie 0. Si vous n'indiquez aucune abréviation de package, elle vérifie tous les packages présents dans le système.

Vous pouvez aussi vous servir de l'option `-v` qui imprime la liste des fichiers du package en l'absence d'erreurs. Exemple :

```
$ 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
$
```

Si vous devez vérifier un package installé sur le système de fichiers root d'un système client, utilisez la commande suivante :

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

Voir aussi Si vous êtes prêt à passer à l'étape suivante, reportez-vous à "Procédure d'obtention d'informations à l'aide de la commande `pkginfo`" à la page 99.

Affichage d'informations supplémentaires sur les packages installés

Vous pouvez vous servir de deux autres commandes pour afficher des informations sur les packages installés :

- La commande `pkgparam` affiche la valeur des paramètres.
- La commande `pkginfo` affiche des informations extraites de la base de données de logiciels d'installation.

Commande `pkgparam`

La commande `pkgparam` vous permet d'afficher les valeurs associées aux paramètres que vous spécifiez sur la ligne de commande. Les valeurs sont extraites du fichier `pkginfo` d'un package spécifique ou du fichier que vous précisez. Une valeur de paramètre s'affiche sur chaque ligne. Vous pouvez au choix afficher les valeurs seules ou les paramètres et leurs valeurs.

▼ Procédure d'obtention d'informations à l'aide de la commande `pkgparam`

1 **Installez votre package.**

Si nécessaire, reportez-vous à "Procédure d'installation d'un package sur un système autonome ou un serveur" à la page 91.

2 **Affichez des informations supplémentaires sur votre package.**

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

`-v`	Affiche le nom du paramètre et sa valeur.
pkg-abrév	Correspond au nom d'un package spécifique.
param	Spécifie un ou plusieurs paramètres dont la valeur est affichée.

Exemple 4–3 Obtention d'informations à l'aide de la commande `pkgparam`

Par exemple, pour n'afficher que les valeurs, utilisez la commande suivante :

```
$ 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
$
```

Pour afficher les paramètres et leurs valeurs, utilisez la commande suivante :

```
$ 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'
$
```

Ou, pour afficher la valeur d'un paramètre spécifique, utilisez le format suivant :

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

Pour plus d'informations, reportez-vous à `pkgparam(1)`.

Voir aussi Si vous êtes prêt à passer à l'étape suivante, reportez-vous à "Procédure de suppression d'un package" à la page 100.

Commande `pkginfo`

Vous pouvez afficher des informations sur les packages installés à l'aide de la commande `pkginfo`. Cette commande dispose de plusieurs options permettant de personnaliser le format et le contenu de l'affichage.

Vous pouvez demander des informations sur un nombre d'instances de package de votre choix.

Affichage par défaut de la commande `pkginfo`

Lorsque vous exécutez la commande `pkginfo` sans indiquer d'options, celle-ci affiche la catégorie, l'instance de package et le nom de package de tous les packages entièrement installés sur votre système. L'affichage est organisé en catégories comme illustré dans l'exemple suivant :

```
$ 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
.
.
.
$
```

Personnalisation du format de l'affichage de la commande `pkginfo`

L'affichage de la commande `pkginfo` peut utiliser trois formats différents : court, extrait et long.

Le format court est le format par défaut. Il ne comprend que la catégorie, l'abréviation du package et le nom complet du package, comme illustré à la rubrique "Affichage par défaut de la commande `pkginfo`" à la page 96.

Le format extrait comprend l'abréviation du package, le nom du package, l'architecture du package (si elle est disponible) et la version du package (si elle est disponible). Utilisez l'option `-x` pour demander le format extrait, comme illustré dans l'exemple suivant :

```
$ 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'utilisation de l'option `-l` permet d'obtenir un affichage au format long qui inclut toutes les informations sur un package, comme illustré dans l'exemple suivant :

```
$ 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)
$
```

Description des paramètres du format long de la commande `pkginfo`

Le tableau suivant décrit les paramètres pouvant être affichés pour chaque package. Un paramètre et sa valeur ne s'affichent que si une valeur a été attribuée au paramètre en question.

TABLEAU 4–2 Paramètres de package

Paramètre	Description
`ARCH`	Architecture prise en charge par le package.
`BASEDIR`	Répertoire de base dans lequel le package logiciel réside (indiqué si le package est réadressable).
`CATEGORY`	Catégories de logiciels auxquelles le package appartient (par exemple, `system` ou `application`).
`CLASSES`	Liste des classes définies pour un package. L'ordre de la liste détermine l'ordre dans lequel les classes sont installées. Les classes apparaissant en début de liste sont installées en premier (support par support). Ce paramètre peut être modifié par le script request.
`DESC`	Description du package.
`EMAIL`	Adresse e-mail utilisée par les utilisateurs pour adresser leurs questions.
`HOTLINE`	Informations sur l'obtention de l'aide sur ce package via le service d'assistance téléphonique.
`INTONLY`	Indique que le package ne doit être installé de manière interactive que lorsqu'une valeur non nulle est définie.
`ISTATES`	Liste des états d'exécution autorisés pour l'installation d'un package (par exemple, `S s 1`).
`MAXINST`	Nombre maximum d'instances de package autorisées sur une machine simultanément. Par défaut, une seule instance de package est autorisée.
`NAME`	Nom du package décrivant habituellement l'abréviation du package.

TABLEAU 4–2 Paramètres de package *(Suite)*

Paramètre	Description
ORDER	Liste des classes définissant l'ordre dans lequel elles doivent être placées sur le support. Utilisée par la commande `pkgmk` pour créer le package. Les classes non définies dans ce paramètre sont placées sur le support par les procédures de tri standard.
PKGINST	Abréviation du package à installer.
PSTAMP	Horodatage de production du package.
RSTATES	Liste des états d'exécution autorisés pour la suppression d'un package (par exemple, `S s 1`).
ULIMIT	S'il est défini, ce paramètre est transféré en tant qu'argument à la commande `ulimit` qui établit la taille maximale d'un fichier lors de l'installation. Ceci ne concerne que les fichiers créés par des scripts de procédure.
VENDOR	Nom du fournisseur ayant fourni le package logiciel.
VERSION	Version du package.
VSTOCK	Numéro de stock fourni par le fournisseur.

Pour plus d'informations sur la commande `pkginfo`, reportez-vous à la page de manuel `pkginfo(1)`.

▼ Procédure d'obtention d'informations à l'aide de la commande `pkginfo`

1 Installez votre package.

Si nécessaire, reportez-vous à "Procédure d'installation d'un package sur un système autonome ou un serveur" à la page 91.

2 Affichez des informations supplémentaires sur votre package.

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

`-x`	Affiche les informations sur le package au format extrait.
`-l`	Affiche les informations sur le package au format long.
pkg-abrév	Correspond au nom d'un package spécifique. En cas d'omission, la commande `pkginfo` affiche les informations sur l'ensemble des packages installés au format par défaut.

Informations supplémentaires

Autres ressources

Si vous êtes prêt à passer à l'étape suivante, reportez-vous à "Procédure de suppression d'un package" à la page 100.

Suppression d'un package

Étant donné que la commande `pkgrm` met à jour les informations de la base de données des produits logiciels, il est important lors de la suppression d'un package d'utiliser la commande `pkgrm`, même si vous êtes tenté d'utiliser à la place la commande `rm`. Vous pouvez par exemple utiliser la commande `rm` pour supprimer un fichier binaire exécutable ; toutefois, cela ne revient pas au même qu'utiliser la commande `pkgrm` pour supprimer le package logiciel contenant le fichier binaire exécutable. Utiliser la commande `rm` pour supprimer les fichiers d'un package endommage la base de données des produits logiciels. Si vous souhaitez véritablement supprimer un seul fichier, vous pouvez utiliser la commande `removef` qui met à jour la base de données des produits logiciels correctement.

▼ Procédure de suppression d'un package

1 Connectez-vous au système en tant que superutilisateur.

2 Supprimez un package installé.

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

pkg-abrév	Correspond au nom d'un ou plusieurs packages à supprimer (séparés par des espaces). En cas d'omission, `pkgrm` supprime tous les packages disponibles.

3 Utilisez la commande `pkginfo` pour vérifier que le package a été correctement supprimé.

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

Si *pkg-abrév* est installé, la commande `pkginfo` renvoie une ligne d'informations le concernant. Sinon, `pkginfo` renvoie l'invite système.

Transfert d'un package sur un support distribution

La commande `pkgtrans` déplace les packages et effectue leur conversion au format package. Vous pouvez utiliser la commande `pkgtrans` pour effectuer les conversions suivantes d'un package pouvant être installé :

- Du format système de fichiers au format flux de données.
- Du format flux de données au format système de fichiers.
- D'un format de système de fichiers à un autre format de système de fichiers.

▼ Procédure de transfert d'un package sur un support de distribution

1 **Créez votre package comme un package au format répertoire si cela n'est pas déjà fait.**

Pour plus d'informations, reportez-vous à "Procédure de création d'un package" à la page 47.

2 **Installez votre package pour vérifier qu'il s'installe correctement.**

Si nécessaire, reportez-vous à "Procédure d'installation d'un package sur un système autonome ou un serveur" à la page 91.

3 **Vérifiez l'intégrité de votre package.**

Si nécessaire, reportez-vous aux rubriques "Procédure de vérification de l'intégrité d'un package" à la page 93, "Procédure d'obtention d'informations à l'aide de la commande `pkginfo`" à la page 99 et "Procédure d'obtention d'informations à l'aide de la commande `pkgparam`" à la page 95.

4 **Supprimez le package installé du système.**

Si nécessaire, reportez-vous à la rubrique "Procédure de suppression d'un package" à la page 100.

5 **Transférez le package (au format package) sur un support de distribution.**

Pour effectuer une conversion de base, exécutez la commande suivante :

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

périphérique1	Correspond au nom du périphérique sur lequel le package réside actuellement.
périphérique2	Correspond au nom du périphérique sur lequel le package converti doit être transféré.
[*pkg-abrév*]	Correspond à une ou plusieurs abréviations de package.

Si aucun nom de package n'est spécifié, tous les packages résidant sur *périphérique1* sont convertis et transférés sur *périphérique2*.

Remarque – Si plusieurs instances d'un même package résident sur *périphérique1*, vous devez utiliser l'identificateur d'instance du package. Pour plus d'informations sur l'identificateur de package, reportez-vous à "Définition d'une instance de package" à la page 27. Lorsqu'une instance de package à convertir se trouve déjà sur *périphérique2*, la commande `pkgtrans` n'effectue pas la conversion. Vous pouvez utiliser l'option `-o` pour indiquer à la commande `pkgtrans` d'écraser toute instance sur le périphérique de destination et l'option `-n` pour lui indiquer de créer une nouvelle instance s'il en existe déjà une. Notez que cette vérification n'est pas utilisée lorsque *périphérique2* prend en charge un format flux de données.

Informations supplémentaires

Autres ressources

Vous venez d'achever les étapes nécessaires à la conception, à la création, à la vérification et au transfert de votre package. Si vous souhaitez consulter quelques études cas, reportez-vous au Chapitre 5, "Création d'un package : Études de cas". Si vous êtes intéressé par des idées de conception de package avancées, reportez-vous au Chapitre 6, "Techniques avancées de création de packages".

◆ ◆ ◆ CHAPITRE 5

Création d'un package : Études de cas

Ce chapitre présente des études de cas permettant d'illustrer divers scénarios d'empaquetage, notamment l'installation d'objets de manière conditionnelle, la prise de décision lors de l'exécution du nombre de fichiers à créer et la modification d'un fichier de données existant au cours de l'installation et de la suppression d'un package.

Chaque étude de cas commence par une description, suivie d'une liste de techniques d'empaquetage utilisées, d'une description de la démarche à suivre pour appliquer ces techniques et d'exemples de fichiers et de scripts associés à l'étude de cas.

Les études de cas décrites dans ce chapitre sont les suivantes :

Demande de participation de l'administrateur

Le package décrit dans cette étude de cas contient trois types d'objets. L'administrateur peut sélectionner le type d'objets qu'il souhaite installer et l'emplacement des objets sur la machine d'installation.

Techniques

Cette étude de cas illustre les techniques suivantes :

- Utilisation de noms de chemin paramétriques (variables contenues dans les noms de chemin d'objet utilisées pour définir plusieurs répertoires de base)

  Pour plus d'informations sur les noms de chemin paramétriques, reportez-vous à "Noms de chemin paramétriques" à la page 35.
- Utilisation d'un script `request` pour demander la participation de l'administrateur

  Pour plus d'informations sur les scripts `request`, reportez-vous à "Rédaction d'un script `request`" à la page 64.
- Définition de valeurs conditionnelles pour un paramètre d'installation.

Démarche

Pour définir l'installation sélective de cette étude de cas, vous devez effectuer les opérations suivantes :

- Définir une classe pour chaque type d'objet pouvant être installé.

  Dans cette étude de cas, les trois types d'objet sont les fichiers exécutables du package, les pages de manuel et les exécutables`emacs`. Chaque type dispose de sa propre classe : respectivement `bin`, `man` et `emacs`. Notez que dans le fichier `prototype`, tous les fichiers d'objets appartiennent à une de ces trois classes.
- Donner au paramètre `CLASSES` du fichier `pkginfo` la valeur nulle.

  En règle générale lorsque vous définissez une classe, vous devez indiquer cette classe dans le paramètre `CLASSES` du fichier `pkginfo`. Aucun objet de cette classe ne peut sinon être installé. Dans cette étude de cas, la valeur de ce paramètre est au départ nulle, signifiant qu'aucun objet ne peut être installé. Le paramètre `CLASSES` est ensuite remplacé via le script `request`, en fonction des choix de l'administrateur. De cette façon, le paramètre `CLASSES` ne contient que la valeur des types d'objet que l'administrateur souhaite installer.

  Remarque – Il est recommandé de donner aux paramètres une valeur par défaut. Si le package contient par exemple des composants communs aux trois types d'objet, vous pouvez les attribuer à la classe `none` et donner au paramètre `CLASSES` la valeur `none`.

- Insérer les noms de chemin paramétriques dans le fichier `prototype`.

  Le script `request` donne à ces variables d'environnement la valeur fournie par l'administrateur. La commande `pkgadd` résout ensuite ces variables d'environnement lors de la phase d'installation pour connaître l'emplacement de l'installation du package.

Les trois variables d'environnement utilisées dans cet exemple sont définies sur leur valeur par défaut dans le fichier `pkginfo` et ont les rôles suivants :

- `$NCMPBIN` définit l'emplacement des exécutables des objets ;
- `$NCMPMAN` définit l'emplacement des pages de manuel ;
- `$EMACS` définit l'emplacement des exécutables `emacs`.

L'exemple de fichier `prototype` illustre la manière de définir les noms de chemin des objets avec des variables.

- Créer un script `request` pour demander à l'administrateur quelles parties du package doivent être installées et à quel endroit les placer.

  Le script `request` de ce package pose deux questions à l'administrateur :

  - Cette partie du package doit-elle être installée ?

    Lorsque la réponse est oui, le nom de classe approprié est ajouté au paramètre `CLASSES`. Par exemple, lorsque l'administrateur choisit d'installer les pages de manuel associées au package, la classe `man` est ajoutée au paramètre `CLASSES`.
  - Dans ce cas, à quel endroit cette partie du package doit-elle être placée ?

    La variable d'environnement appropriée est définie sur la réponse à cette question. Dans l'exemple des pages de manuel, la variable `$NCMPMAN` prend la valeur donnée en réponse.

  Ce deux questions sont réitérées pour chacun de trois types d'objet.

  À la fin du script `request`, les paramètres sont mis à la disposition de l'environnement d'installation pour la commande `pkgadd` et tout autre script d'empaquetage. Le script `request` effectue cette opération en enregistrant ces définitions dans le fichier fourni par l'utilitaire d'appel. Aucun autre script n'est fourni pour cette étude de cas.

  Remarquez dans ce script `request` que les questions sont générées par les outils de validation des données `ckyorn` et `ckpath`. Pour plus d'informations sur ces outils, reportez-vous à `ckyorn(1)` et `ckpath(1)`.

Fichiers de l'étude de cas

Fichier `pkginfo`

```
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
```

Fichier `prototype`

```
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
```

Script `request`

```
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
```

Remarquez qu'un script `request` peut s'arrêter sans ne laisser aucun fichier dans le système de fichiers. Dans le cadre des installations sur les versions antérieures à 2.5 de Solaris et versions compatibles (où aucun script `checkinstall` ne peut être utilisé), le script `request` est l'endroit où effectuer tous les tests nécessaires vis à vis du système de fichiers afin de garantir le succès de l'installation. Lorsque le script `request` contient le code 1, l'installation s'arrête nette.

Ces exemples de fichiers illustrent l'emploi des chemins paramétriques pour définir plusieurs répertoires de base. Toutefois, la méthode recommandée implique l'utilisation du paramètre `BASEDIR` qui est géré et validé par la commande `pkgadd`. Chaque fois que plusieurs répertoires de base sont utilisés, prenez les mesures nécessaires en prévision de l'installation de plusieurs versions et architectures sur la même plate-forme.

Création d'un fichier lors de l'installation et enregistrement du fichier lors de la suppression

Cette étude de cas crée un fichier de base de données lors de la phase d'installation et enregistre une copie de la base de données à la suppression du package.

Techniques

Cette étude de cas illustre les techniques suivantes :

- Utilisation de classes et de scripts d'action de classe pour effectuer des opérations particulières sur différents groupes d'objets

  Pour plus d'informations, reportez-vous à "Rédaction de scripts d'action de classe" à la page 71.
- Utilisation du fichier `space` pour informer la commande `pkgadd` que de l'espace supplémentaire est nécessaire au bon déroulement de l'installation de ce package

  Pour plus d'informations sur le fichier `space`, reportez-vous à "Réservation d'espace supplémentaire sur un système cible" à la page 57.
- Utilisation de la commande `installf` pour installer un fichier non défini dans les fichiers `prototype` et `pkgmap`.

Démarche

Pour créer un fichier de base de données lors de l'installation et enregistrer une copie lors de la suppression, vous devez pour cette étude de cas effectuer les opérations suivantes :

- Définir trois classes.

  Les trois classes suivantes doivent être définies dans le paramètre `CLASSES` pour le package de cette étude de cas :

- La classe standard `none` qui contient un groupe de processus appartenant au sous-répertoire `bin` ;
- La classe `admin` qui contient un fichier exécutable `config` et un répertoire contenant des fichiers de données ;
- La classe `cfgdata` qui contient un répertoire.

- Faire du package un package réadressable collectivement.

  Remarquez dans le fichier `prototype` qu'aucun des noms de chemin ne commence par une barre oblique ni par une variable d'environnement. Ceci indique qu'ils sont réadressables collectivement.

- Calculer la quantité d'espace requise par le fichier de la base de données et créer un fichier `space` à fournir avec le package. Ce fichier informe la commande `pkgadd` que le package nécessite de l'espace supplémentaire et indique la quantité requise.

- Créer un script d'action de classe pour la classe `admin` (`i.admin`).

  L'exemple de script initialise une base de données à l'aide des fichiers de données appartenant à la classe `admin`. Pour effectuer cette opération, il procède comme suit :

  - Il copie le fichier de données source à l'emplacement de destination approprié.
  - Il crée un fichier vide nommé `config.data` et l'attribue à une classe `cfgdata`.
  - Il exécute la commande `bin/config`, qui est fournie avec le package et est déjà installée, pour renseigner le fichier de base de données `config.data` à l'aide des fichiers de données appartenant à la classe `admin`.
  - Il exécute la commande `installf -f` pour terminer l'installation de `config.data`.

  Aucune opération particulière n'est nécessaire vis à vis de la classe `admin` lors de la phase de suppression et pour cette raison, aucun script d'action de classe de suppression n'est créé. Tous les fichiers et répertoires de la classe `admin` sont donc supprimés du système.

- Créer un script d'action de classe pour la classe `cfgdata` (`r.cfgdata`).

  Le script de suppression crée une copie du fichier de la base de données avant sa suppression. Aucune opération particulière n'est nécessaire vis à vis de cette classe lors de la phase d'installation et pour cette raison, aucun script d'action de classe d'installation n'est nécessaire.

  N'oubliez pas que les valeurs à indiquer dans un script de suppression correspondent à la liste des noms de chemin à supprimer. Les noms de chemin s'affichent toujours par ordre alphabétique inverse. Ce script de suppression copie les fichiers dans le répertoire nommé `$PKGSAV`. Une fois tous les noms de chemin traités, le script supprime tous les répertoires et fichiers associés à la classe `cfgdata`.

  Le résultat de cette suppression est la copie de `config.data` dans `$PKGSAV`, suivie de la suppression du fichier `config.data` et du répertoire de données.

Fichiers de l'étude de cas

Fichier `pkginfo`

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

Fichier `prototype`

```
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
```

Fichier `space`

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

Script d'action de classe `i.admin`

```
# 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
```

Cet exemple illustre une situation rare dans laquelle `installf` est approprié dans un script d'action de classe. Le fichier `space` ayant été utilisé pour réserver de l'espace sur un système de fichiers spécifique, ce nouveau fichier peut être ajouté sans problèmes bien qu'il ne soit pas inclus dans le fichier `pkgmap`.

Script de suppression `r.cfgdata`

```
# 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
```

Définition des compatibilités et des dépendances d'un package

Le package de cette étude de cas utilise des fichiers d'information facultatifs pour définir les compatibilités et les dépendances d'un package, et afficher un message de copyright au cours de l'installation.

Techniques

Cette étude de cas illustre les techniques suivantes :

- Utilisation du fichier `copyright`
- Utilisation du fichier `compver`
- Utilisation du fichier `depend`

Pour plus d'informations sur ces fichiers, reportez-vous à "Création de fichiers d'information" à la page 52.

Démarche

Pour respecter les critères fournis dans la description, vous devez :

- Créez un fichier `copyright`.

  Un fichier `copyright` contient le texte ASCII d'un message de copyright. Le message illustré dans l'exemple de fichier s'affiche à l'écran lors de l'installation du package.

- Créer un fichier `compver`.

  Le fichier `pkginfo` illustré sur la figure suivante définit la version du package comme étant la version 3.0. Le fichier `compver` définit la version 3.0 comme étant compatible avec les versions 2.3, 2.2, 2.1, 2.1.1, 2.1.3 et 1.7.

- Créer un fichier `depend`.

  Les fichiers répertoriés dans un fichier `depend` doivent déjà être installés sur le système au moment de l'installation d'un package. L'exemple de fichier indique que 11 packages doivent déjà être installés sur le système au moment de l'installation.

Fichiers de l'étude de cas

Fichier `pkginfo`

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

Fichier `copyright`

```
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
```

Fichier `compver`

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

Fichier `depend`

```
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
```

Modification d'un fichier à l'aide de classes standard et de scripts d'action de classe

Cette étude de cas modifie un fichier existant lors de l'installation d'un package à l'aide de classes standard et de scripts d'action de classe. Elle utilise une des trois méthodes de modification disponibles. Les deux autres méthodes sont décrites dans les rubriques "Modification d'un fichier à l'aide de la classe `sed` et d'un script `postinstall`" à la page 115 et "Modification d'un fichier à l'aide de la classe `build`" à la page 117. Le fichier modifié est `/etc/inittab` .

Techniques

Cette étude de cas illustre la façon d'utiliser les scripts d'action de classe d'installation et de suppression. Pour plus d'informations, reportez-vous à "Rédaction de scripts d'action de classe" à la page 71.

Démarche

Pour modifier `/etc/inittab` au cours de l'installation à l'aide de classes et de scripts d'action de classe, vous devez effectuer les opérations suivantes :

- Créer une classe.

  Créez une classe appelée `inittab` . Vous devez fournir un script d'action de classe d'installation et de suppression pour cette classe. Définissez la classe `inittab` dans le paramètre `CLASSES` du fichier `pkginfo`.

- Créer un fichier `inittab`.

  Ce fichier contient les informations de l'entrée à ajouter dans `/etc/inittab` . Remarquez dans le fichier `prototype` que `inittab` appartient à la classe `inittab` et que son type de fichier est `e` (modifiable).

- Créer un script d'action de classe d'installation (`i.inittab`).

  N'oubliez pas que les scripts d'action de classe doivent produire les mêmes résultats chaque fois qu'ils sont exécutés. Le script d'action de classe effectue les procédures suivantes :

  - Il regarde si cette entrée a déjà été ajoutée.
  - Si elle a déjà été ajoutée, il supprime toutes les versions de l'entrée qu'il trouve.
  - Il modifie le fichier `inittab` et ajoute des lignes de commentaires indiquant d'où provient l'entrée.
  - Il replace le fichier temporaire dans `/etc/inittab`.
  - Il exécute la commande `init q` lorsqu'il reçoit l'indicateur `ENDOFCLASS`.

  Notez que la commande `init q` peut être effectuée par ce script d'installation. Cette démarche ne requiert pas de script `postinstall` d'une ligne.

- Créer un script d'action de classe de suppression (`r.inittab`).

  Le script de suppression est très similaire au script d'installation. Les informations ajoutées par les scripts d'installation sont supprimées et la commande `init q` est exécutée.

Cette étude de cas est plus complexe que la suivante ; reportez-vous à "Modification d'un fichier à l'aide de la classe `sed` et d'un script `postinstall`" à la page 115. Au lieu de fournir deux fichiers, trois sont nécessaires et le fichier `/etc/inittab` fourni n'est qu'un substituant contenant un segment de l'entrée à insérer. Ce segment aurait pu être placé dans le fichier `i.inittab` mais la commande `pkgadd` doit avoir un fichier à transmettre au fichier `i.inittab`. En outre, la procédure de suppression doit être placée dans un fichier distinct (`r.inittab`). Bien que cette méthode fonctionne bien, il est recommandé de la réserver aux situations impliquant l'installation complexe de plusieurs fichiers. Reportez-vous à "Modification de fichiers `crontab` au cours de l'installation" à la page 119.

Le programme `sed` utilisé dans "Modification d'un fichier à l'aide de la classe `sed` et d'un script `postinstall`" à la page 115 prend en charge plusieurs instances d'un package puisque le commentaire situé à la fin de l'entrée `inittab` se base sur l'instance d'un package. L'étude de cas

de la rubrique "Modification d'un fichier à l'aide de la classe build" à la page 117 illustre une démarche plus rationalisée de modification du fichier /etc/inittab pendant l'installation.

Fichiers de l'étude de cas

Fichier pkginfo

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

Fichier prototype

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

Script d'action de classe d'installation i.inittab

```
# 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
```

Script d'action de classe de suppression r.inittab

```
# 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
```

Fichier inittab

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

Modification d'un fichier à l'aide de la classe sed et d'un script postinstall

Cette étude de cas modifie un fichier présent sur la machine d'installation lors de l'installation du package. Elle utilise une des trois méthodes de modification disponibles. Les deux autres méthodes sont décrites dans les rubriques "Modification d'un fichier à l'aide de classes standard et de scripts d'action de classe" à la page 112 et "Modification d'un fichier à l'aide de la classe build" à la page 117. Le fichier modifié est /etc/inittab.

Techniques

Cette étude de cas illustre les techniques suivantes :

- Utilisation de la classe sed

  Pour plus d'informations sur la classe sed, reportez-vous à "Script de classe sed" à la page 76.

- Utilisation d'un script postinstall

  Pour plus d'informations sur ce script, reportez-vous à "Rédaction de scripts de procédure" à la page 69.

Démarche

Pour modifier /etc/inittab lors de l'installation à l'aide de la classe sed, vous devez effectuer les opérations suivantes :

- Ajoutez le script de la classe sed dans le fichier prototype.

  Le nom du script doit être le nom du fichier à modifier. Dans l'exemple, le fichier à modifier est /etc/inittab et le script sed est de ce fait nommé /etc/inittab. Les champs mode, owner et groupe d'un script sed n'ont aucune spécification (ce qui est indiqué dans l'exemple prototype par des points d'interrogation). Le type de fichier du script sed doit être e (indiquant qu'il s'agit d'un fichier modifiable).

- Incluez dans le paramètre CLASSES la classe sed.

  Comme l'illustre l'exemple de fichier, sed est la seule classe à installer. Il pourrait cependant s'agir de bien d'autres classes.

- Créer un script d'action de classe sed.

  Votre package ne peut pas fournir de copie de /etc/inittab à l'apparence souhaitée, car /etc/inittab est un fichier dynamique ; il est donc impossible de prévoir son apparence à l'installation du package. Toutefois, l'utilisation d'un script sed vous permet de modifier le fichier /etc/inittab lors de l'installation du package.

- Créez un script postinstall.

  Vous devez exécuter la commande init q pour informer le système que le fichier /etc/inittab a été modifié. Dans l'exemple, cette opération ne peut être effectuée que dans un script postinstall. En examinant l'exemple du script postinstall, vous remarquerez que son seul objectif est d'exécuter la commande initq.

Cette démarche de modification du fichier /etc/inittab lors de l'installation présente un inconvénient ; elle nécessite un script complet (le script postinstall) pour la simple exécution de la commande init q.

Fichiers de l'étude de cas

Fichier pkginfo

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

Fichier prototype

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

Script d'action de classe sed (/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
```

Script postinstall

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

Modification d'un fichier à l'aide de la classe build

Cette étude de cas modifie un fichier présent sur la machine d'installation lors de l'installation du package. Elle utilise une des trois méthodes de modification disponibles. Les deux autres méthodes sont décrites dans les rubriques "Modification d'un fichier à l'aide de classes standard et de scripts d'action de classe" à la page 112 et "Modification d'un fichier à l'aide de la classe sed et d'un script postinstall" à la page 115. Le fichier modifié est /etc/inittab.

Techniques

Cette étude de cas illustre l'utilisation de la classe build. Pour plus d'informations sur la classe build, reportez-vous à "Script de classe build" à la page 77.

Démarche

Cette démarche de modification du fichier /etc/inittab utilise la classe build. Un script de classe build est exécuté comme un script shell et son résultat devient la nouvelle version du fichier exécuté. Autrement dit, le fichier de données /etc/inittab fourni avec ce package est exécuté et le résultat de cette exécution devient /etc/inittab.

Le script de classe build est exécuté lors de l'installation et de la suppression du package. L'argument install est transmis au fichier s'il est exécuté lors de la phase d'installation. Remarquez dans l'exemple de script de classe build que la procédure d'installation est définie en testant cet argument.

Pour modifier /etc/inittab à l'aide de la classe build, vous devez effectuer les opérations suivantes :

- Définir le fichier de version dans le fichier prototype.

  L'entrée correspondant au fichier de version dans le fichier prototype doit le placer dans la classe build et définir son type de fichier comme étant e. Vérifiez que la valeur du paramètre CLASSES figurant dans le fichier pkginfo est build.

- Créer le script de classe build.

  L'exemple de script de classe build effectue les procédures suivantes :

  - Il modifie le fichier /etc/inittab pour modifier toute modification apportée à ce package. Remarquez que le nom de fichier /etc/inittab est codé en dur dans la commande sed.
  - Si l'installation du package est en cours, il ajoute le nouvelle ligne à la fin du fichier /etc/inittab. La nouvelle entrée inclut une balise de commentaire qui indique sa provenance.
  - Il exécute la commande init q.

Cette solution résout les inconvénients décrits dans les études de cas des rubriques "Modification d'un fichier à l'aide de classes standard et de scripts d'action de classe" à la page 112 et "Modification d'un fichier à l'aide de la classe sed et d'un script postinstall" à la page 115. Seul un bref fichier est nécessaire (outre les fichiers pkginfo et prototype). Le fichier fonctionne avec plusieurs instances d'un package puisque le paramètre PKGINST est utilisé ; par ailleurs, aucun script postinstall n'est requis puisque la commande init q peut être exécutée à partir du script de classe build.

Fichiers de l'étude de cas

Fichier pkginfo

```
PKG=case6
NAME=Case Study #6
CATEGORY=applications
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1d05
CLASSES=build
```

Fichier prototype

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

Fichier de version

```
# 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
```

Modification de fichiers crontab au cours de l'installation

Cette étude de cas modifie les fichiers crontab lors de l'installation du package.

Techniques

Cette étude de cas illustre les techniques suivantes :

- Utilisation de classes et de scripts d'action de classe

  Pour plus d'informations, reportez-vous à "Rédaction de scripts d'action de classe" à la page 71.

- Utilisation de la commande crontab dans un script d'action de classe.

Démarche

La manière la plus efficace de modifier plusieurs fichiers lors de l'installation consiste à définir une classe et à fournir un script d'action de classe. Si vous avez utilisé la démarche de la classe build, vous devez fournir un script de classe build pour chaque fichier crontab modifié. La définition d'une classe cron représente une démarche plus globale. Pour modifier les fichiers crontaben suivant cette démarche, vous devez effectuer les opérations suivantes :

- Définir les fichiers crontab à modifier dans le fichier prototype.

  Créez une entrée dans le fichier prototype pour chaque fichier crontab à modifier. Définissez cron comme classe et e comme type de fichier de chaque fichier. Utilisez le nom réel du fichier à modifier.
- Créer les fichiers crontab du package.

  Ces fichiers contiennent les informations à ajouter aux fichiers crontab actuels du même nom.
- Créer un script d'action de classe d'installation pour la classe cron.

  L'exemple de script de classe i.cron effectue les procédures suivantes :

  - Il détermine l'ID utilisateur (UID).

    Le script i.cron définit la variable *user* sur le nom de base du script de classe cron traité. Ce nom est l'UID. Par exemple, le nom de base de /var/spool/cron/crontabs/root est root, qui est également l'UID.
  - Il exécute la commande crontab, en se servant de l'UID et de l'option -l.

    L'utilisation de l'option -l indique à crontab de transmettre dans le résultat standard, le contenu du fichier crontab de l'utilisateur défini.
  - Il envoie le résultat de la commande crontab dans un script sed qui supprime toute entrée ajoutée précédemment par cette technique d'installation.
  - Il place le résultat modifié dans un fichier temporaire.
  - Il ajoute le fichier de données de l'UID root (fourni avec le package) dans le fichier temporaire et insère une balise indiquant la provenance des entrées.
  - Il exécute la commande crontab avec le même UID et lui transmet les fichiers temporaires comme entrée.
- Créer un script d'action de classe de suppression pour la classe cron.

  Le script r.cron est identique au script d'installation à ceci près qu'il n'existe pas de procédure pour ajouter des informations dans le fichier crontab.

  Ces procédures sont effectuées pour chaque fichier de la classe cron.

Fichiers de l'étude de cas

Les scripts i.cron et r.cron décrits ci-après sont exécutés par le superutilisateur. Modifier le fichier crontab d'un autre superutilisateur peut avoir des conséquences imprévisibles. Si nécessaire, remplacez l'entrée suivante de chaque script :

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

par celle-ci :

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

Commande pkginfo

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

Fichier prototype

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

Script d'action de classe d'installation i.cron

```
# 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
```

Script d'action de classe de suppression `r.cron`

```
# 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
```

Premier fichier `crontab`

```
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
```

Deuxième fichier `crontab`

```
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
```

Remarque – Si la modification d'un groupe de fichiers augmente la taille globale des fichiers de plus de 10 Ko, fournissez un fichier `space` pour que la commande `pkgadd` puisse accommoder cette augmentation. Pour plus d'informations sur le fichier `space`, reportez-vous à “Réservation d'espace supplémentaire sur un système cible” à la page 57.

Installation et suppression d'un pilote à l'aide de scripts de procédure

Ce package installe un pilote.

Techniques

Cette étude de cas illustre les techniques suivantes :

- Installation et chargement d'un pilote à l'aide d'un script `postinstall`
- Déchargement d'un pilote à l'aide d'un script `preremove`

Pour plus d'informations sur ces scripts, reportez-vous à "Rédaction de scripts de procédure" à la page 69.

Démarche

- Créer un script `request`.

  Le script `request` détermine l'emplacement d'installation des objets du pilote en interrogeant l'administrateur et en appliquant la réponse obtenue au paramètre `$KERNDIR`.

  Le script se termine par une routine permettant de mettre les deux paramètres `CLASSES` et `KERNDIR` à disposition de l'environnement d'installation et du script `postinstall`.

- Créez un script `postinstall`.

  Le script `postinstall` effectue l'installation du pilote. Il est exécuté une fois que les deux fichiers `buffer` et `buffer.conf` sont installés. Le fichier `postinstall` illustré dans cet exemple effectue les opérations suivantes :

  - Il utilise la commande `add_drv` pour charger le pilote sur le système.
  - Il crée un lien vers le périphérique à l'aide de la commande `installf`.
  - Il termine l'installation à l'aide de la commande `installf -f`.
  - Il crée un script `preremove`.

    Le script `preremove` utilise la commande `rem_drv` pour décharger le pilote du système, puis supprime le lien `/dev/buffer0` .

Fichiers de l'étude de cas

Fichier `pkginfo`

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

Fichier `prototype`

Pour installer un pilote lors de l'installation, vous devez inclure les fichiers d'objets et de configuration du pilote dans le fichier `prototype`.

Dans cet exemple, le module exécutable du pilote est nommé `buffer` ; la commande `add_drv` opère sur ce fichier. Le noyau utilise le fichier de configuration, `buffer.conf` pour aider à configurer le pilote.

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

Remarquez les points suivants dans le fichier `prototype` de cet exemple :

- Les objets du package ne nécessitant aucun traitement particulier, vous pouvez les placer dans la classe standard `none`. Le paramètre `CLASSES` est défini sur `none` dans le fichier `pkginfo`.
- Les noms de chemin de `buffer` et `buffer.conf` commencent par la variable `$KERNDIR`. Cette variable est définie dans le script `request` et permet à l'administrateur de choisir l'emplacement d'installation des pilotes. Le répertoire par défaut est `/kernel/drv`.
- Le script `postinstall` (script effectuant l'installation du pilote) dispose d'une entrée.

Script `request`

```
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
```

Script `postinstall`

```
# 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
```

Script preremove

```
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
```

Installation d'un pilote à l'aide de la classe sed et de scripts de procédure

Cette étude de cas décrit l'installation d'un pilote à l'aide de la classe sed et de scripts de procédure. Elle diffère par ailleurs de l'étude de cas précédente (reportez-vous à "Installation et suppression d'un pilote à l'aide de scripts de procédure" à la page 122) sur le fait que ce package est composé à la fois d'objets absolus et d'objets réadressables.

Techniques

Cette étude de cas illustre les techniques suivantes :

- Création d'un fichier prototype contenant des objets absolus et des objets réadressables.

  Pour plus d'informations sur la création d'un fichier prototype, reportez-vous à "Création d'un fichier prototype" à la page 31.
- Utilisation d'un script postinstall

  Pour plus d'informations sur ce script, reportez-vous à "Rédaction de scripts de procédure" à la page 69.
- Utilisation d'un script preremove

  Pour plus d'informations sur ce script, reportez-vous à "Rédaction de scripts de procédure" à la page 69.
- Utilisation d'un fichier copyright

  Pour plus d'informations sur ce fichier, reportez-vous à "Rédaction d'un message de copyright" à la page 55.

Démarche

- Créez un fichier `prototype` contenant des objets de package absolus et des objets de package réadressables.

  La procédure est abordée en détail à la rubrique "Fichier `prototype`" à la page 126.
- Ajoutez le script de la classe `sed` dans le fichier `prototype`.

  Le nom du script doit être le nom du fichier à modifier. Dans l'exemple, le fichier à modifier est `/etc/devlink.tab` et le script `sed` est de ce fait nommé `/etc/devlink.tab`. Les champs mode, owner et groupe d'un script `sed` n'ont aucune spécification (ce qui est indiqué dans l'exemple `prototype` par des points d'interrogation). Le type de fichier du script `sed` doit être `e` (indiquant qu'il s'agit d'un fichier modifiable).
- Incluez dans le paramètre `CLASSES` la classe `sed`.
- Créez un script d'action de classe `sed` (`/etc/devlink.tab`).
- Créez un script `postinstall`.

  Le script `postinstall` doit exécuter la commande `add_drv` pour ajouter le pilote du périphérique sur le système.
- Créez un script `preremove`.

  Le script `preremove` doit exécuter la commande `rem_drv` pour supprimer le pilote du périphérique du système avant la suppression du package.
- Créez un fichier `copyright`.

  Un fichier `copyright` contient le texte ASCII d'un message de copyright. Le message illustré dans l'exemple de fichier s'affiche à l'écran lors de l'installation du package.

Fichiers de l'étude de cas

Fichier `pkginfo`

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

Fichier `prototype`

Par exemple, cette étude de cas utilise la structure hiérarchique des objets de package illustrée ci-après.

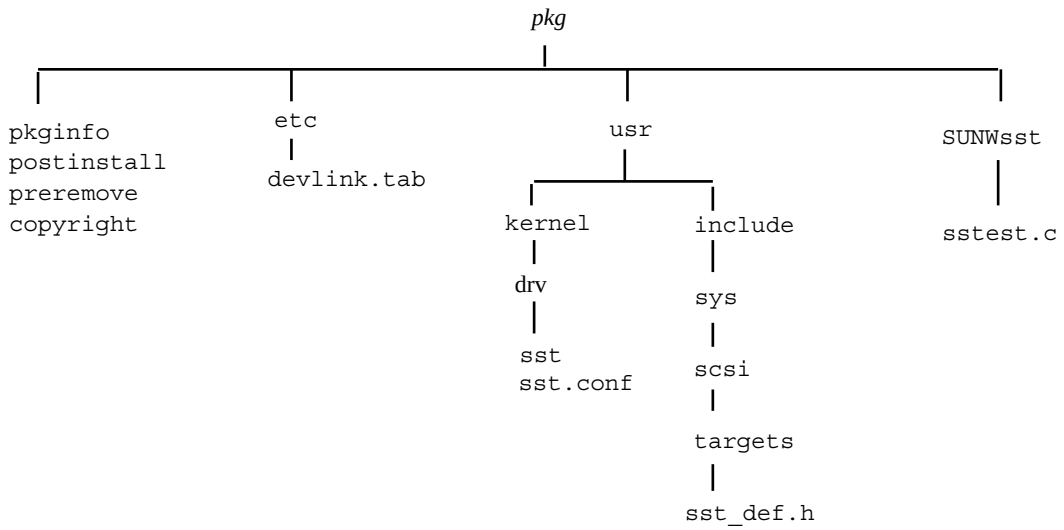


FIGURE 5–1 Structure du répertoire d'un package hiérarchique

Les objets du package sont installés dans les mêmes fichiers que sur l'illustration du répertoire `pkg` ci-dessus. Les modules du pilote (`sst` et `sst.conf`) sont installés dans `/usr/kernel/drv` et le fichier include est installé dans `/usr/include/sys/scsi/targets`. Les fichiers `sst`, `sst.conf` et `sst_def.h` sont des objets absolus. Le programme test `sstest.c` et son répertoire `SUNWsst` sont tous deux réadressables ; leur emplacement d'installation est défini par le paramètre `BASEDIR`.

Les autres composants du package (tous les fichiers de contrôle) doivent être placés dans le répertoire supérieur du package sur la machine de développement, à l'exception du script de la classe `sed` . Celui-ci est appelé `devlink.tab` d'après le nom du fichier qu'il modifie et doit être placé dans `etc`, répertoire contenant le vrai fichier `devlink.tab`.

À partir du répertoire `pkg`, exécutez la commande `pkgproto` comme suit :

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

Le résultat de la commande ci-dessus est comme suit :

```
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
```

Le fichier `prototype` n'est pas encore terminé. Pour terminer le fichier, vous devez lui apporter les modifications suivantes :

- Insérez les entrées pour les fichiers de contrôle (type de fichier `i`) car ils utilisent un format différent des autres objets de package.
- Supprimez les entrées correspondant aux répertoires qui se trouvent déjà sur le système cible.
- Modifiez les droits d'accès et la propriété de chaque entrée.
- Préfixez le nom des objets de package absolus d'une barre oblique.

Fichier `prototype` final :

```
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
```

Les points d'interrogation de l'entrée correspondant au script `sed` indiquent que les droits d'accès et la propriété du fichier présent sur la machine d'installation ne doivent pas être modifiés.

Script d'action de classe `sed` (`/etc/devlink.tab`)

Dans l'exemple de pilote, un script de classe `sed` est utilisé pour ajouter une entrée correspondant au pilote dans le fichier `/etc/devlink.tab`. Ce fichier est utilisé par la commande `devlinks` pour créer des liens symboliques de `/dev` à `/devices`. Script `sed` :

```
# 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
```

La commande pkgrm n'exécute pas la partie du script correspondant à la suppression. Il s'avère parfois nécessaire d'ajouter une ligne au script preremove afin d'exécuter sed directement pour supprimer l'entrée du fichier /etc/devlink.tab.

Script d'installation postinstall

Dans cet exemple, il suffit au script d'exécuter la commande add_drv.

```
# 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
```

La commande add_drv utilise le paramètre BASEDIR ; le script doit pour cette raison annuler la valeur définie pour BASEDIR avant d'exécuter la commande et la rétablir après l'exécution.

Une des opérations effectuées par la commande add_drv est l'exécution de devlinks qui utilise l'entrée placée dans /etc/devlink.tab par le script de classe sed pour créer les entrées /dev correspondant au pilote.

Le code de sortie du script postinstall est significatif. Le code de sortie 20 indique à la commande pkgadd de demander à l'utilisateur de réinitialiser le système (opération nécessaire après l'installation d'un pilote) et le code de sortie 2 indique à la commande pkgadd d'informer l'utilisateur que l'installation a partiellement échoué.

Script de suppression preremove

Dans cet exemple de pilote, il supprime les liens figurant dans /dev et exécute la commande rem_drv sur le pilote.

```
# 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
```

Le script supprime lui-même les entrées `/dev` alors que les entrées `/devices` sont supprimées par la commande `rem_drv`.

Fichier `copyright`

Ce fichier est un simple fichier ASCII contenant le texte d'un avis de copyright. L'avis s'affiche au début de l'installation du package tel qu'il apparaît dans le fichier.

```
    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.
```

◆ ◆ ◆ CHAPITRE 6

Techniques avancées de création de packages

Toutes les possibilités offertes par le package System V tel qu'il est implémenté dans le système d'exploitation Solaris représentent un outil puissant d'installation de produits logiciels. En tant que concepteur de package, vous pouvez bénéficier de ces capacités. Les packages ne faisant pas partie du système d'exploitation Solaris (packages non fournis en standard) peuvent utiliser le mécanisme de classes pour personnaliser les installations serveur/client. Les packages réadressables peuvent être conçus de sorte à répondre aux besoins de l'administrateur. Un produit complexe peut être livré sous forme de packages composites capables de résoudre automatiquement les dépendances des packages. La mise à niveau et l'application de patchs peuvent être personnalisées par le concepteur de package. Les packages auxquels des patchs ont été appliqués peuvent être livrés de la même manière que les packages sans patchs et les archives de désinstallation peuvent aussi être incluses dans le produit.

La liste suivante répertorie les informations fournies dans ce chapitre :

Spécification du répertoire de base

Plusieurs méthodes sont disponibles pour spécifier l'emplacement de l'installation d'un package et il est important de pouvoir modifier le répertoire de base de l'installation dynamiquement lors de la phase d'installation. Lorsque cette spécification est effectuée correctement, un administrateur peut installer plusieurs versions et plusieurs architectures sans difficulté.

Cette section aborde les méthodes courantes avant de passer à des démarches permettant d'améliorer les installations sur des systèmes hétérogènes.

Fichier de valeurs d'administration par défaut

Les administrateurs chargés de l'installation des packages peuvent se servir des fichiers d'administration pour contrôler l'installation des packages. Toutefois, en tant que concepteur de package, vous devez savoir ce qu'est un fichier d'administration et de quelle manière un administrateur peut modifier l'installation prévue d'un package.

Un fichier d'administration indique à la commande `pkgadd` si elle doit ou non effectuer les contrôles et afficher les invites qu'elle effectue/affiche habituellement. Les administrateurs doivent pour cette raison maîtriser la procédure d'installation d'un package et les scripts qui lui sont associés avant de se servir des fichiers d'administration.

Le système d'exploitation SunOS est livré avec un fichier de valeurs d'administration par défaut que vous trouverez dans `/var/sadm/install/admin/default`. Il s'agit du fichier établissant le niveau de politique d'administration le plus basique en matière d'installation de produits logiciels. Le fichier tel qu'il est livré est comme suit :

```
#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
```

L'administrateur peut modifier ce fichier afin d'établir de nouveaux comportements par défaut ou, créer un autre fichier d'administration et spécifier son existence à l'aide de l'option `-a` dans la commande `pkgadd`.

Onze paramètres peuvent être définis dans un fichier d'administration ; toutefois, certains sont facultatifs. Pour plus d'informations, reportez-vous à `admin(4)`.

Le paramètre `basedir` spécifie la manière dont le répertoire de base est généré à l'installation d'un package. La plupart des administrateurs conservent la valeur `default` mais le paramètre `basedir` peut prendre une des valeurs suivantes :

- `ask`, qui signifie que le répertoire de base doit toujours être demandé à l'administrateur ;
- un nom de chemin absolu ;
- un nom de chemin absolu contenant `$PKGINST`, qui signifie que l'installation doit toujours être effectuée dans un répertoire de base généré à partir de l'instance du package.

Remarque – Si la commande `pkgadd` est appelée avec l'argument `-a none`, elle demande systématiquement le répertoire de base à l'administrateur. Malheureusement, cette commande définit également *tous* les paramètres du fichier sur la valeur par défaut `quit` qui est susceptible d'engendrer des problèmes supplémentaires.

S'habituer à l'incertitude

Un administrateur contrôle tous les packages installés sur un système via l'utilisation d'un fichier d'administration. Un autre fichier de valeurs d'administration par défaut est malheureusement souvent fourni par le *concepteur du package* ; ce fichier ignore en effet les souhaits de l'administrateur.

Les concepteurs de package incluent parfois un autre fichier d'administration qui leur permet (et non pas à l'administrateur) de contrôler l'installation d'un package. Étant donné que l'entrée `basedir` du fichier de valeurs d'administration par défaut ignore tous les autres répertoires de base, elle représente une méthode simple de sélection du répertoire de base lors de la phase d'installation. Dans toutes les versions du système d'exploitation Solaris antérieures à Solaris 2.5, ceci était considéré comme la méthode la plus simple de contrôle du répertoire de base.

Toutefois, vous devez tenir compte des souhaits de l'administrateur relatifs à l'installation du produit. La présence d'un fichier temporaire de valeurs d'administration par défaut destiné à contrôler l'installation induit la méfiance chez les administrateurs. Utilisez un script `request` et un script `checkinstall` pour contrôler ces installations sous la direction de l'administrateur. Si le script `request` implique véritablement l'administrateur dans la procédure, l'empaquetage du System V bénéficiera à la fois aux administrateurs et aux concepteurs de package.

Utilisation du paramètre `BASEDIR`

Le fichier `pkginfo` de tout package réadressable doit contenir une entrée spécifiant le répertoire de base par défaut dans le format suivant :

`BASEDIR=`*absolute_path*

Il ne s'agit que du répertoire de base par défaut et peut de ce fait être remplacé par l'administrateur au cours de l'installation.

Bien que certains packages requièrent plusieurs répertoires de base, l'avantage que présente ce paramètre pour le placement du package est que le répertoire de base est déjà créé et prêt à être utilisé dès le lancement de l'installation. Le chemin d'accès approprié au répertoire de base du serveur et du client est mis à la disposition de tous les scripts de procédure sous la forme de variables d'environnement réservées. La commande `pkginfo -r SUNWstuf` affiche la base d'installation actuelle du package.

Dans le script `checkinstall`, `BASEDIR` est le paramètre tel qu'il a été défini dans le fichier `pkginfo` (non encore déterminé). Afin de vérifier le répertoire de base cible, la construction `${PKG_INSTALL_ROOT}$BASEDIR` est nécessaire. Ceci signifie que le script `request` ou `checkinstall` peut remplacer la valeur de `BASEDIR` dans l'environnement d'installation avec des résultats prévisibles. Le temps que le script `preinstall` soit appelé, le paramètre `BASEDIR` a été remplacé par le pointeur entièrement conditionné du répertoire de base actuel sur le système cible, même si le système est un client.

Remarque – Le script `request` se sert du paramètre `BASEDIR` différemment selon la version du système d'exploitation SunOS. Afin de tester un paramètre `BASEDIR` dans un script `request`, le code suivant doit être utilisé pour déterminer le répertoire de base alors utilisé.

```
# request script
constructs base directory
if [ ${CLIENT_BASEDIR} ]; then
      LOCAL_BASE=$BASEDIR
else
      LOCAL_BASE=${PKG_INSTALL_ROOT}$BASEDIR
fi
```

Utilisation des répertoires de base paramétriques

Si un package nécessite plusieurs répertoires de base, vous pouvez les créer à l'aide de noms de chemin paramétriques. Cette méthode est devenue assez populaire bien qu'elle présente les inconvénients suivants :

- Un package contenant des noms de chemin paramétriques se comporte généralement comme un package absolu mais est traité par la commande `pkgadd` comme un package réadressable. Le paramètre `BASEDIR` doit être défini même s'il n'est pas utilisé.
- L'administrateur ne peut pas déterminer la base d'installation du package avec les utilitaires du System V (la commande `pkginfo -r` ne fonctionne pas).
- L'administrateur ne peut pas se servir de la méthode établie pour réadresser le package (il est qualifié d'adressable mais se comporte comme un package absolu).
- L'installation de plusieurs architectures ou de plusieurs versions nécessite de prévoir des plans d'urgence pour chacun des répertoires de base cible, ce qui se traduit souvent par l'utilisation de scripts d'action de classe complexes.

Bien que les paramètres qui déterminent les répertoires de base soient définis dans le fichier `pkginfo`, ils peuvent être modifiés par le script `request`. Cette caractéristique est une des raisons principales de la popularité de cette approche. Les inconvénients sont cependant chroniques et il est recommandé de n'utiliser cette configuration qu'en dernier ressort.

Exemples : Utilisation des répertoires de base paramétriques

Fichier `pkginfo`

```
# 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
```

Fichier `pkgmap`

```
: 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
```

Gestion du répertoire de base

Tout package disponible en plusieurs versions ou destiné à plusieurs architectures doit être conçu de sorte à pouvoir, le cas échéant, *parcourir* le répertoire de base. Le parcours d'un répertoire de base signifie que si une version précédente ou une architecture différente du

package sur le point d'être installée se trouve déjà dans le répertoire de base, ce package résout le problème, en créant par exemple un nouveau répertoire de base sous un nom légèrement différent. Les scripts `request` et `checkinstall` de Solaris 2.5 et des versions compatibles sont en mesure de modifier la variable d'environnement `BASEDIR`. Ceci n'est pas le cas des versions précédentes du système d'exploitation Solaris.

Même dans les versions plus anciennes du système d'exploitation Solaris, le script `request` pouvait redéfinir les répertoires au sein de la base d'installation. Le script `request` est en mesure d'effectuer cette opération d'une manière prenant en charge la plupart des préférences d'administration.

Prise en compte du réadressage

La possibilité de sélectionner un répertorie de base unique à une architecture et à une version pour chaque package installé se traduit malheureusement par des niveaux hiérarchiques de répertoires superflus. Par exemple, pour un produit conçu pour les processeurs SPARC et x86, vous pouvez organiser les répertoires de base par processeur et par version comme illustré ci-dessous.

Répertoire de base	Version et processeur
`/opt/SUNWstuf/sparc/1.0`	Version 1.0, SPARC
`/opt/SUNWstuf/sparc/1.2`	Version 1.2, SPARC
`/opt/SUNWstuf/x86/1.0`	Version 1.0, x86

Cette organisation est faisable. Elle suppose toutefois que ces noms et numéros aient une signification pour l'administrateur. Une meilleure approche consiste à effectuer automatiquement cette opération *après* avoir fourni des explications à l'administrateur et avoir obtenu son autorisation.

Ceci vous permet d'effectuer toute l'opération dans le package sans que l'administrateur ait besoin de le faire manuellement. Vous pouvez attribuer le répertoire de base de manière arbitraire, puis établir clairement les liens client appropriés dans un script `postinstall`. Vous pouvez également utiliser la commande `pkgadd` pour installer tout au partie du package sur les clients dans le script `postinstall`. Vous pouvez même demander à l'administrateur la liste des utilisateurs ou clients devant connaître l'existence de ce package et mettre automatiquement à jour les variables d'environnement `PATH` et les fichiers `/etc`. Ceci est tout à fait acceptable tant que toutes les opérations effectuées au cours de l'installation du package sont annulées lors de sa désinstallation.

Parcours des répertoires de base

Vous pouvez vous servir de deux méthodes pour contrôler le répertoire de base lors de la phase d'installation. La première est la mieux adaptée aux nouveaux packages qui ne peuvent être installés que sur Solaris 2.5 et des versions compatibles ; elle offre à l'administrateur des données très utiles ; elle prend par ailleurs en charge l'installation de plusieurs versions et architectures et ne nécessite qu'un minimum de travail particulier. La deuxième méthode peut être utilisée par tous les packages et fait appel au contrôle des paramètres de développement inhérent aux scripts `request` pour assurer le bon déroulement des installations.

Utilisation du paramètre `BASEDIR`

Le script `checkinstall` peut sélectionner le répertoire de base approprié lors de la phase d'installation, ce qui permet de placer le répertoire de base très près de la racine de l'arborescence de répertoires. L'exemple suivant incrémente le répertoire de base de manière séquentielle, ce qui se traduit par des répertoires aux formats `/opt/SUNWstuf`, `/opt/SUNWstuf.1` et `/opt/SUNWstuf.2`. L'administrateur peut se servir de la commande `pkginfo` pour déterminer quelles architecture et version sont installées dans chaque répertoire de base.

Si le package `SUNWstuf` utilise cette méthode, ses fichiers `pkginfo` et `pkgmap` ressemblent aux suivants :

Fichier `pkginfo`

```
# 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
```

Fichier `pkgmap`

```
: 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
```

Exemple : Analyse de scripts parcourant un `BASEDIR`

Supposons que la version x86 de `SUNWstuf` est déjà installée sur le serveur dans `/opt/SUNWstuf`. Lorsque l'administrateur utilise la commande `pkgadd` pour installer la version SPARC, le script `request` doit détecter l'existence de la version x86 et interagir avec l'administrateur pour l'installation.

Remarque – Le répertoire de base peut être parcouru sans l'interaction de l'administrateur dans un script `checkinstall` ; toutefois, lorsque des opérations arbitraires de ce type se produisent trop souvent, les administrateurs n'ont plus confiance dans la procédure.

Les scripts `request` et `checkinstall` d'un package gérant cette situation peuvent ressembler aux suivants :

Script `request`

```
# 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
```

Script `checkinstall`

```
# checkinstall
script for SUNWstuf to politely suspend

grep quitinstall $1
if [ $? -eq 0 ]; then
    exit 3        # politely suspend installation
fi

exit 0
```

Cette approche ne serait pas très efficace si le répertoire de base était simplement `/opt`. Ce package doit faire référence à `BASEDIR` de manière plus précise car le parcours de `/opt` pourrait s'avérer difficile. En fait, selon le schéma de montage, l'opération peut même s'avérer impossible. L'exemple parcourt le répertoire de base en créant un nouveau répertoire sous `/opt`, ce qui ne crée aucun problème.

Cet exemple utilise un script `request` et un script `checkinstall`, bien que les versions antérieures à Solaris 2.5 ne puissent pas exécuter de script `checkinstall`. Le script `checkinstall` de cet exemple est utilisé afin de suspendre poliment l'installation en réponse à un message privé sous la forme d'une chaîne `quitinstall` ». Si ce script s'exécute sous Solaris 2.3, le script `checkinstall` est ignoré et le script `request` suspend l'installation en affichant un message d'erreur.

N'oubliez pas que pour les versions antérieures à Solaris 2.5 et versions compatibles, le paramètre `BASEDIR` est un paramètre en lecture seule qui ne peut pas être modifié par le script `request.` Pour cette raison, lorsqu'une ancienne version du système d'exploitation SunOS est détectée (en recherchant une variable d'environnement `CLIENT_BASEDIR` conditionnée), le script `request` n'a que deux possibilités : poursuivre ou abandonner l'opération.

Utilisation de chemins paramétriques relatifs

Si votre produit logiciel a des chances d'être installé sur d'anciennes versions du système d'exploitation SunOS, le script `request` doit effectuer toutes les opérations requises. Cette approche peut également être utilisée pour manipuler plusieurs répertoires. Si des répertoires supplémentaires sont nécessaires, ils doivent toutefois être inclus dans un seul répertoire de base afin que le produit reste simple à gérer. Bien que le paramètre `BASEDIR` n'offre pas le niveau de granularité offert par la toute dernière version de Solaris, votre package peut malgré tout parcourir le répertoire de base à l'aide du script `request` pour manipuler les chemins paramétriques. L'exemple suivant illustre un fichier `pkginfo` et un fichier `pkgmap` :

Fichier pkginfo

```
# 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
```

Fichier pkgmap

```
: 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
```

Cet exemple n'est pas parfait. Une commande `pkginfo -r` renvoie `/opt` comme base d'installation, ce qui est assez ambigu. Bon nombre de packages se trouvent en effet dans `/opt` mais il s'agit au moins d'un répertoire précis. Tout comme l'exemple précédent, l'exemple suivant prend parfaitement en charge plusieurs architectures et plusieurs versions. Le script `request` peut être personnalisé afin de répondre aux besoins d'un package spécifique et de résoudre toute dépendance applicable.

Exemple : Script `request` parcourant un chemin paramétrique relatif

```
# 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
```

Prise en charge du réadressage dans un environnement hétérogène

Le concept d'origine sur lequel repose l'empaquetage du System V présumait une architecture par système. Le concept d'un serveur ne jouait aucun rôle dans la conception. De nos jours, bien entendu, un seul serveur peut prendre en charge plusieurs architectures, ce qui peut se traduire par la présence de plusieurs copies d'un logiciel sur un même serveur, chacune étant destinée à une architecture différente. Bien que les packages Solaris soient isolés dans la limite des systèmes de fichiers recommandés (par exemple, `/` et `/usr`), les bases de données des produits étant stockées à la fois sur le serveur et sur chaque client, les installations ne prennent pas toutes nécessairement en charge cette division. Certaines implémentations prennent en charge une structure entièrement différente et impliquent une base de données de produits commune. Bien que diriger les clients vers des versions différentes soit très simple, installer des packages System V dans des répertoires de base différents peut s'avérer complexe pour l'administrateur.

Lors de la conception de votre package, vous devez également prendre en considération les méthodes courantes employées par les administrateurs pour introduire la nouvelle version d'un logiciel. Les administrateurs souhaitent souvent installer et tester la toute dernière version parallèlement à la version déjà installée. La procédure consiste à installer la nouvelle version dans un répertoire de base autre que celui dans lequel la version actuelle est installée et à rediriger une poignée de clients non importants vers la nouvelle version afin de la tester. À mesure que sa confiance augmente, l'administrateur redirige de plus en plus de clients vers la nouvelle version. Après un certain temps, l'administrateur ne conserve l'ancienne version que pour des cas d'urgence et finit par la supprimer.

En d'autres termes, les packages destinés aux systèmes hétérogènes modernes doivent prendre en charge le vrai réadressage au sens où l'administrateur peut être amené à les placer à tout endroit raisonnable du système de fichiers et s'attendre à pouvoir utiliser toutes leurs fonctionnalités. L'environnement Solaris 2.5 et les versions compatibles offrent divers outils très utiles qui permettent d'installer correctement plusieurs architectures et versions sur un même système. Solaris 2.4 et les versions compatibles prennent également en charge le vrai réadressage mais leur procédure n'est pas aussi évidente.

Approche traditionnelle

Packages réadressables

L'ABI du System V suggère que le premier objectif d'un package réadressable était de simplifier l'installation du package pour l'administrateur. De nos jours, la nécessité des packages réadressables va plus loin. Il n'est plus seulement question de commodité puisqu'il arrive souvent lors de l'installation qu'un produit logiciel actif se trouve déjà dans le répertoire par défaut. Un package non conçu dans l'optique de gérer cette situation peut soit remplacer le produit existant, soit suspendre l'installation. Par contre, un package conçu pour gérer plusieurs

architectures et plusieurs versions peut procéder à l'installation correctement et offrir à l'administrateur diverses options entièrement compatibles avec les habitudes d'administration déjà en place.

D'une certaine manière, le problème lié à l'existence de plusieurs architectures et celui lié à plusieurs versions est le même. Il faut pouvoir installer une variante du package actuel parallèlement à d'autres variantes et pouvoir diriger les clients ou les utilisateurs de systèmes autonomes de fichiers exportés vers n'importe laquelle de ces variantes sans y perdre en fonctionnalité. Bien que Sun ait établi des méthodes permettant de gérer plusieurs architectures sur un serveur, les administrateurs ne se conforment pas toujours à ces recommandations. Tous les packages doivent pouvoir se conformer aux souhaits raisonnables des administrateurs en matière d'installation.

Exemple : Package réadressable traditionnel

L'exemple suivant illustre l'apparence d'un package réadressable traditionnel. Le package doit être placé dans `/opt/SUNWstuf` et ses fichiers `pkginfo` et `pkgmap` peuvent s'afficher comme suit :

Fichier `pkginfo`

```
# 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
```

Fichier `pkgmap`

```
: 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
```

On parle ici de méthode traditionnelle car chaque objet de package est installé dans le répertoire de base défini par le paramètre `BASEDIR` du fichier `pkginfo`. Par exemple, le premier objet du fichier `pkgmap` est installé comme répertoire `/opt/SUNWstuf`.

Packages absolus

Un package absolu est un package qui s'installe sur un système de fichiers root (`/`) particulier. Ces packages sont difficiles à gérer du point de vue de plusieurs versions et architectures. En règle générale, tous les packages doivent être réadressables. Il existe cependant de très bonnes raisons d'inclure des éléments absolus dans un package réadressable.

Exemple : Package absolu traditionnel

Si le package `SUNWstuf` était un package absolu, le paramètre `BASEDIR` ne devrait pas être défini dans le fichier `pkginfo` et le fichier `pkgmap` serait comme suit :

Fichier `pkgmap`

```
: 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
```

Dans cet exemple, si l'administrateur a spécifié un autre répertoire de base lors de l'installation, il est ignoré par la commande `pkgadd`. Ce package s'installe toujours dans `/opt/SUNWstuf` du système cible.

L'argument `-R` de la commande `pkgadd` fonctionne comme prévu. Par exemple,

```
pkgadd -d . -R /export/opt/client3 SUNWstuf
```

installe les objets dans `/export/opt/client3/opt/SUNWstuf` ; toutefois, il s'agit du seul semblant de réadressage de ce package.

Notez l'emploi du point d'interrogation (`?`) pour le répertoire `/opt` du fichier `pkgmap`. Il indique que les attributs actuels ne doivent pas être modifiés. Cela ne signifie pas « créer le répertoire à partir des attributs par défaut », bien que cela puisse se produire dans certaines situations. Tout répertoire spécifique au nouveau package doit spécifier tous les attributs de manière explicite.

Packages composites

Tout package contenant des objets réadressables est appelé un package réadressable. Ceci peut porter à confusion puisque le fichier `pkgmap` du package réadressable peut contenir des chemins absolus. L'utilisation d'une entrée racine (`/`) dans un fichier `pkgmap` peut améliorer les aspects réadressables du package. Les packages contenant à la fois des entrées réadressables et des entrées racines sont appelés des packages *composites*.

Exemple : Solution traditionnelle

Supposons qu'un objet dans le package `SUNWstuf` soit un script de démarrage exécuté au niveau d'exécution 2. Vous devez installer le fichier `/etc/rc2.d/S70dostuf` en tant que partie du package mais vous ne pouvez pas le placer dans le répertoire de base. Supposons qu'un package réadressable soit l'unique solution ; les fichiers `pkginfo` et `pkgmap` seraient comme suit :

Fichier `pkginfo`

```
# 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
```

Fichier `pkgmap`

```
: 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
```

Il n'existe pas de grandes différences entre cette approche et celle utilisée pour le package absolu. Un package absolu est en fait plus approprié car si l'administrateur a spécifié un autre répertoire de base, cette solution ne fonctionne pas.

En fait, seul un fichier de ce package doit être relatif à la racine ; les autres peuvent être placés n'importe où. L'utilisation d'un package composite comme solution à ce problème est abordée dans la suite de cette section.

Au-delà de la tradition

L'approche décrite dans cette section n'est pas applicable à tous les packages mais elle permet d'améliorer les performances lors de l'installation sur un environnement hétérogène. Les packages fournis dans le cadre du système d'exploitation Solaris (packages fournis en standard) sont peu concernés par cette approche ; par contre, les packages non fournis en standard peuvent se servir de l'empaquetage non traditionnel.

La raison pour laquelle les packages réadressables sont recommandés est la prise en charge de la spécification suivante :

Lorsqu'un package est ajouté ou supprimé, les comportements actuels souhaités des produits logiciels restent inchangés.

Les packages non fournis en standard doivent résider dans `/opt` pour s'assurer que le nouveau package n'interfère pas avec les produits existants.

Informations supplémentaires sur les packages composites

Deux règles doivent être appliquées lors de la création d'un package composite fonctionnel :

- Établir le répertoire de base en fonction de la destination de la majorité des objets du package.

- Lorsqu'un objet de package doit être placé dans un répertoire commun autre que le répertoire de base (par exemple, `/etc`), le spécifier en tant que nom de chemin absolu dans le fichier `prototype`.

En d'autres termes, puisque qu'un objet réadressable peut par définition être installé à tout endroit et fonctionner correctement ainsi, aucun script de démarrage exécuté par `init` lors de l'initialisation ne peut être considéré comme réadressable. Bien que spécifier `/etc/passwd` en tant que chemin relatif dans le package livré n'est en soi pas incorrect, il ne peut y avoir qu'une destination.

Noms de chemins absolus à l'apparence réadressable

Lorsque vous créez un package composite, les chemins absolus doivent fonctionner sans interférer avec les logiciels déjà installés. Un package pouvant être intégralement contenu dans `/opt` évite ce problème puisque aucun fichier ne représente un obstacle. Lorsqu'un fichier placé dans `/etc` est inclus dans le package, vous devez vous assurer que les noms de chemins absolus se comportent comme des noms de chemins relatifs. Réfléchissez aux deux exemples suivants :

Exemple : Modification d'un fichier

Description

Une entrée est ajoutée à un tableau ou l'objet est un nouveau tableau susceptible d'être modifié par d'autres programmes ou packages.

Implémentation

Définissez l'objet comme ayant le type de fichier `e` et appartenant à la classe `build`, `awk` ou `sed`. Le script effectuant cette opération doit se supprimer aussi efficacement qu'il s'ajoute.

Exemple

Une entrée doit être ajoutée à `/etc/vfstab` pour prendre en charge le nouveau disque dur électronique.

L'entrée dans le fichier `pkgmap` pourrait être :

```
1 e sed /etc/vfstab ? ? ?
```

Le script `request` demande à l'opérateur si `/etc/vfstab` doit être modifié par le package. Si l'opérateur répond par la négative, le script request imprime les instructions de la procédure manuelle et exécute :

```
echo "CLASSES=none" >> $1
```

Si l'opérateur répond par l'affirmative, il exécute :

```
echo "CLASSES=none sed" >> $1
```

Ceci active le script d'action de classe qui doit apporter les modifications requises. La classe `sed` indique que le fichier de package `/etc/vfstab` est un programme `sed` qui contient les procédures d'installation et de suppression du fichier du même nom sur le système cible.

Exemple : Création d'un fichier

Description

L'objet est un fichier entièrement nouveau peu susceptible d'être modifié ultérieurement ou un fichier qui en remplace un autre appartenant à un package différent.

Implémentation

Définissez l'objet de package comme ayant le type de fichier `f` et installez-le à l'aide d'un script d'action de classe capable d'annuler la modification.

Exemple

Un tout nouveau fichier est nécessaire dans `/etc` afin de fournir les informations requises pour la prise en charge du disque dur électronique, appelé `/etc/shdisk.conf` . L'entrée dans le fichier `pkgmap` peut être :

```
.
.
.
1 f newetc /etc/shdisk.conf
.
.
.
```

Le script d'action de classe `i.newetc` est chargé d'installer ce fichier ainsi que tout autre fichier devant être placé dans `/etc`. Il vérifie qu'aucun autre fichier ne s'y trouve. Si le répertoire est vide, il y copie tout simplement le fichier. S'il contient déjà un fichier, il le sauvegarde avant d'installer le nouveau fichier. Le script `r.newetc` supprime ces fichiers et le cas échéant, rétablit les originaux. Vous trouverez ci-après le fragment clé du script d'installation.

```
# 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
```

Notez que ce script utilise la variable d'environnement PKGSAV pour stocker une sauvegarde du fichier à remplacer. Lorsque l'argument ENDOFCLASS est transmis au script, autrement dit lorsque la commande pkgadd informe le script qu'il s'agit des dernières entrées de cette classe, le script archive et compresse les fichiers enregistrés à l'aide d'un programme de compression privé stocké dans le répertoire d'installation du package.

Bien que l'utilisation de la variable d'environnement PKGSAV ne soit pas fiable pendant la mise à jour d'un package, si le package n'est pas mis à jour (à l'aide d'un patch par exemple), le fichier de sauvegarde est sécurisé. Le script de suppression suivant inclut un code permettant de résoudre le deuxième problème, à savoir, le fait que dans les anciennes versions, la commande pkgrm ne transmet pas aux scripts le bon chemin d'accès à la variable d'environnement PKGSAV .

Le script de suppression peut être comme suit :

```
# 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
```

Ce script utilise un algorithme de désinstallation privé (`unsquish`) situé dans le répertoire d'installation de la base de données des packages. Ceci est automatiquement effectué par la commande `pkgadd` lors de la phase d'installation. Tous les scripts non spécifiquement reconnus comme ayant l'attribut d'installation seule par la commande `pkgadd` sont stockés dans ce répertoire afin d'être utilisés par la commande `pkgrm`. Vous ne pouvez pas compter sur l'emplacement de ce répertoire mais vous pouvez être sûr qu'il est plat et qu'il contient toutes les informations et scripts d'installation appropriés du package. Ce script trouve le répertoire en raison du fait que le script d'action de classe s'exécute systématiquement à partir du répertoire contenant le programme `unsquish`.

Notez également que ce script ne suppose pas simplement que le répertoire cible est `/etc`. Il peut en fait s'agir de `/export/root/client2/etc` . Le répertoire approprié peut être créé de deux façons.

- utilisez la construction `${PKG_INSTALL_ROOT}/etc`, ou
- utilisez le nom de répertoire d'un fichier transmis par la commande `pkgadd` (ce que le script fait).

L'utilisation de cette approche pour chaque objet absolu du package vous assure que le comportement actuel souhaité reste inchangé ou qu'il est au moins récupérable.

Exemple : Package composite

L'exemple suivant illustre les fichiers `pkginfo` et `pkgmap` d'un package composite.

Fichier `pkginfo`

```
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
```

Fichier `pkgmap`

```
: 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
```

Alors que `S70dostuf` appartient à la classe `daemon`, les répertoires qui y accèdent (déjà en place lors de la phase d'installation) appartiennent eux à la classe `none`. Même si les répertoires étaient uniques à ce package, vous devriez les laisser dans la classe `none`. Ceci, parce que les répertoires doivent être créés en premier et supprimés en dernier, ce qui est toujours vrai de la classe `none`. La commande `pkgadd` crée des répertoires qui ne sont ni copiés à partir du package, ni transmis à un script d'action de classe afin d'être créés. Ils sont par contre créés par la commande `pkgadd` avant qu'elle n'appelle le script d'action de classe ; la commande `pkgrm` par ailleurs supprime les répertoires une fois le script d'action de classe de suppression terminé.

De ce fait, lorsqu'un répertoire d'une classe spéciale contient des objets de la classe `none`, la tentative de suppression du répertoire par la commande `pkgrm` échoue car le répertoire n'est pas vidé à temps. Si un objet de la classe `none` doit être inséré dans un répertoire d'une classe spéciale, le répertoire en question n'est pas créé à temps pour recevoir l'objet. La commande `pkgadd` crée le répertoire à la volée au cours de l'installation de l'objet et ne parvient parfois pas à synchroniser les attributs de ce répertoire lorsqu'elle trouve la définition `pkgmap`.

Remarque – Lorsque vous attribuez un répertoire à une classe, pensez toujours à l'ordre de création et de suppression.

Création de packages pouvant être installés à distance

Tous les packages *doivent* pouvoir être installés à distance. Pouvoir être installé à distance signifie que vous ne pouvez pas présumer que l'administrateur va installer votre package sur le système de fichiers racine (`/`) du système exécutant la commande `pkgadd`. Pour accéder au fichier `/etc/vfstab` du système cible dans l'un de vos scripts de procédure, la variable d'environnement `PKG_INSTALL_ROOT` doit être utilisée. En d'autres termes, le nom de chemin `/etc/vfstab` vous permet d'accéder au fichier `/etc/vfstab` du système exécutant la

commande pkgadd, mais l'administrateur peut effectuer l'installation sur un client dans /export/root/client3. Le chemin ${PKG_INSTALL_ROOT}/etc/vfstab vous garantit l'accès au système de fichiers cible.

Exemple : Installation sur un système client

Dans cet exemple, le package SUNWstuf est installé dans client3, qui est configuré avec /opt dans son système de fichiers racine (/). Une autre version de ce package est déjà installée dans client3 et le répertoire de base est défini sur basedir=/opt/$PKGINST à partir d'un fichier d'administration (thisadmin). Pour plus d'informations sur les fichiers d'administration, reportez-vous à "Fichier de valeurs d'administration par défaut" à la page 132. La commande pkgadd exécutée sur le serveur est :

```
# pkgadd -a thisadmin -R /export/root/client3 SUNWstuf
```

Le tableau suivant répertorie les variables d'environnement et les valeurs transmises aux scripts de procédure :

TABLEAU 6–1 Valeurs transmises aux scripts de procédure

Variable d'environnement	Valeur
PKGINST	SUNWstuf.2
PKG_INSTALL_ROOT	/export/root/client3
CLIENT_BASEDIR	/opt/SUNWstuf.2
BASEDIR	/export/root/client3/opt/SUNWstuf.2

Exemple : Installation sur un serveur ou un système autonome

Pour effectuer l'installation sur le serveur ou un système autonome dans les mêmes circonstances que l'exemple précédent, la commande est la suivante :

```
# pkgadd -a thisadmin SUNWstuf
```

Le tableau suivant répertorie les variables d'environnement et les valeurs transmises aux scripts de procédure :

TABLEAU 6–2 Valeurs transmises aux scripts de procédure

Variable d'environnement	Valeur
PKGINST	SUNWstuf.2
PKG_INSTALL_ROOT	Non définie.
CLIENT_BASEDIR	/opt/SUNWstuf.2
BASEDIR	/opt/SUNWstuf.2

Exemple : Montage de systèmes de fichiers partagés

Supposons que le package SUNWstuf crée et partage un système de fichiers sur le serveur dans /export/SUNWstuf/share. Lorsque le package est installé sur les systèmes client, leurs fichiers /etc/vfstab doivent être mis à jour pour monter ce système de fichiers partagé. Cette situation est un exemple d'application de la variable CLIENT_BASEDIR.

L'entrée sur le client doit présenter le point de montage en référence au système de fichiers du client. Cette ligne doit être créée correctement, que l'installation s'effectue à partir du serveur ou du client. Supposons que le nom du système du serveur soit $SERVER. Vous pouvez accéder à $PKG_INSTALL_ROOT/etc/vfstab puis, à l'aide de la commande sed ou de la commande awk, créer la ligne suivante pour le fichier /etc/vfstab du client.

```
$SERVER:/export/SUNWstuf/share - $CLIENT_BASEDIR/usr nfs - yes ro
```

Par exemple, pour le serveur universe et le système client client9, la ligne figurant dans le fichier /etc/vfstab du système client serait :

```
universe:/export/SUNWstuf/share - /opt/SUNWstuf.2/usr nfs - yes ro
```

Lorsque ces paramètres sont utilisés correctement, l'entrée monte toujours le système de fichiers du client, qu'il soit créé localement ou à partir du serveur.

Application de patchs à des packages

Un patch n'est en fait qu'un package creux conçu pour remplacer certains fichiers dans le package d'origine. La seule raison de livrer un package creux est de faire des économies d'espace sur le support de livraison. Vous pouvez aussi livrer le package d'origine dans son intégralité, avec certains fichiers modifiés, ou fournir l'accès au package modifié sur le réseau. Tant que seuls ces nouveaux fichiers sont différents (les autres fichiers n'ont pas été recompilés), la commande pkgadd installe les différences. Consultez les instructions suivantes concernant l'application de patchs à des packages.

- Si le système est très complexe, il est recommandé d'établir un système d'identification des patchs qui empêche que deux patchs remplacent le même fichier lorsqu'ils ont pour objectif de corriger des comportements anormaux distincts. Par exemple, des groupes de fichiers qui s'excluent mutuellement sont attribués aux nombres de base des patchs Sun. Ces groupes de fichiers deviennent leur responsabilité.
- Un patch doit pouvoir être désinstallé.

Il est essentiel que le numéro de version du package du patch soit identique à celui du package d'origine. Vous devez conserver une trace de l'état du patch du package via une entrée dans le fichier `pkginfo` du type :

`PATCH=`*patch_number*

Lorsque vous modifiez la version du package pour un patch, vous créez en fait une autre instance du package et il devient dès lors extrêmement compliqué de gérer le produit auquel les patchs ont été appliqués. Cette méthode d'application de patchs par instance progressive comporte certains avantages pour les premières versions du système d'exploitation Solaris mais elle rend la gestion de systèmes plus complexes fastidieuse.

Tous les paramètres de zone du patch doivent correspondre aux paramètres de zone du package.

En ce qui concerne les packages formant l'environnement d'exploitation Solaris, une seule copie du package doit figurer dans la base de données des packages, bien qu'il puisse y avoir plusieurs instances auxquelles des patchs ont été appliqués. Afin de supprimer un objet d'un package installé (à l'aide de la commande `removef`), vous devez déterminer à quelles instances appartient le fichier.

Toutefois, si votre package (non fourni dans le cadre du système d'exploitation Solaris) doit déterminer le niveau de patch d'un package donné *fourni* dans le cadre du système d'exploitation Solaris, cela pose un problème qui doit être résolu ici. Les scripts d'installation peuvent être assez volumineux sans pour autant avoir un impact significatif puisqu'ils sont stockés sur le système de fichiers cible. Grâce à l'utilisation de scripts d'action de classe et de divers autres scripts de procédure, vous pouvez enregistrer les fichiers modifiés avec la variable d'environnement `PKGSAV` (ou dans un autre répertoire plus permanent) afin de pouvoir, le cas échéant, désinstaller les patchs. Vous pouvez également surveiller l'historique des patchs en définissant les variables d'environnement appropriées via des scripts `request`. Les scripts des sections suivantes présument l'existence de plusieurs patchs dont la numérotation a un sens lorsqu'ils sont appliqués à un même package. Dans ce cas, les numéros de patch individuels représentent un sous-groupe de fichiers de fonctionnalités associés au sein du package. Deux numéros de patch différents ne peuvent pas modifier un même fichier.

Afin de transformer un package creux standard en package de patch, les scripts décrits dans les sections suivantes peuvent tout simplement être intégrés au package. Tous sont reconnus en tant que composants de package standard à l'exception des deux derniers, `patch_checkinstall` et `patch_postinstall`. Ces deux scripts peuvent être incorporés au package de désinstallation

si vous souhaitez inclure la fonction de désinstallation du patch. Les scripts sont relativement simples et les opérations qu'ils effectuent faciles à comprendre.

Remarque – Cette méthode d'application de patchs peut être utilisée pour appliquer des patchs aux systèmes client ; toutefois, les répertoires racine des clients disponibles sur le serveur doivent disposer des droits appropriés pour autoriser la lecture par l'utilisateur `install` ou `nobody`.

Script `checkinstall`

Le script `checkinstall` vérifie que le patch concerne bien le package en question. Une fois qu'il en a la confirmation, il crée la *liste de patchs* et la liste *d'informations sur les patchs*, puis les insère dans le fichier réponse à incorporer dans la base de données des packages.

La liste de patchs contient tous les patchs qui ont été appliqués au package actuel. Cette liste de patchs est enregistrée sur une ligne du fichier `pkginfo` du package installé, comme suit :

`PATCHLIST=`*`patch_id patch_id ...`*

La liste d'informations sur les patchs contient le nom des patchs desquels dépendent le patch actuel. Cette liste de patchs est également enregistrée sur une ligne du fichier `pkginfo`, comme suit :

```
PATCH_INFO_103203-01=Installed... Obsoletes:103201-01 Requires: \ Incompatibles: 120134-01
```

Remarque – Ces lignes (et leur format) sont déclarées en tant qu'interface publique. Toute société fournissant des patchs destinés aux packages Solaris doit mettre cette liste à jour en conséquence. Lorsqu'un patch est fourni, chaque package présent dans le patch contient un script `checkinstall` qui effectue cette opération. Ce même script `checkinstall` met également à jour d'autres paramètres spécifiques au patch. Il s'agit d'une nouvelle architecture de patchs appelée « application de patch d'instance directe » (Direct Instance Patching).

Dans cet exemple, les packages d'origine et leurs patchs se trouvent dans le même répertoire. Les deux packages d'origine se nomment `SUNWstuf.v1` et `SUNWstuf.v2`, et leurs patchs, `SUNWstuf.p1` et `SUNWstuf.p2`. Pour cette raison, un script de procédure pourrait avoir des difficultés à déterminer de quel répertoire proviennent ces fichiers, puisque toutes les informations concernant le nom du package placées après le point (« . ») sont coupées pour le paramètre `PKG`, et la variable d'environnement `PKGINST` se réfère à l'instance installée et non pas à l' instance source. Les scripts de procédure peuvent donc déterminer le répertoire source, le script `checkinstall` (qui est toujours exécuté à partir du répertoire source) lance la requête et transmet l'emplacement sous forme de variable `SCRIPTS_DIR`. Si le répertoire source ne contenait qu'un seul package appelé `SUNWstuf`, les scripts de procédure auraient pu le trouver à l'aide de `$INSTDIR/$PKG`.

```
# 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
```

Script `preinstall`

Le script `preinstall` initialise le fichier `prototype`, les fichiers d'information et les scripts d'installation du package de désinstallation à créer. Ce script est très simple et les autres scripts de cet exemple ne permettent au package de désinstallation que de décrire les fichiers standard.

Pour rétablir les liens symboliques, les liens physiques, les périphériques et les canaux nommés d'un package de désinstallation, vous pouvez modifier le script `preinstall` afin d'utiliser la commande `pkgproto` pour comparer le fichier `pkgmap` fourni aux fichiers installés, puis créer une entrée de fichier `prototype` pour chaque élément autre qu'un fichier à modifier dans le package de désinstallation. La méthode à utiliser est similaire à la méthode employée dans le script d'action de classe.

Les scripts `patch_checkinstall` et `patch_postinstall` sont insérés dans l'arborescence source du package à partir du script `preinstall`. Ces deux scripts annulent les opérations effectuées par le patch.

```
# 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
```

Script d'action de classe

Le script d'action de classe crée une copie de chaque fichier de remplacement et ajoute une ligne correspondante dans le fichier `prototype` pour le package de désinstallation. Ces opérations sont effectuées à l'aide de scripts `nawk` relativement simples. Le script d'action de classe reçoit une liste de paires source/destination comprenant des fichiers ordinaires qui ne correspondent pas aux fichiers installés. Les liens symboliques et éléments autres que des fichiers doivent être traités dans le script `preinstall`.

```
# 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
```

Script `postinstall`

Le script `postinstall` crée le package de désinstallation à partir des informations fournies par les autres scripts. Étant donné que les commandes `pkgmk` et `pkgtrans` n'ont pas besoin de la base de données des packages, elles peuvent être exécutées au sein de l'installation du package.

Dans l'exemple, la désinstallation du patch peut être effectuée en créant un package au format flux de données dans le répertoire d'enregistrement (à l'aide de la variable d'environnement`PKGSAV`). Ceci n'est pas évident mais le package doit être au format flux de données parce que le répertoire d'enregistrement est déplacé pendant une opération `pkgadd`. Si la commande `pkgadd` est appliquée à un package dans son propre répertoire d'enregistrement, l'emplacement supposé de la source du package à un moment donné s'avère très peu fiable. Un package au format flux de données est placé dans un répertoire temporaire et installé à partir de celui-ci. (Un package au format répertoire commencerait l'installation à partir du répertoire d'enregistrement et serait soudainement déplacé au cours d'une opération `pkgadd` à sécurité intégrée.)

Pour déterminer quels patchs sont appliqués à un a package, utilisez la commande suivante :

```
$ pkgparam SUNWstuf PATCHLIST
```

À l'exception de `PATCHLIST`, qui est une interface publique Sun, les noms de paramètre de cet exemple ne contiennent rien de significatif. `PATCH` peut être remplacé par la liste traditionnelle `SUNW_PATCHID` et les diverses autres listes telles que `PATCH_EXCL` et `PATCH_REQD` peuvent être renommées en conséquence.

Si certains packages de patchs dépendent d'autres packages de patchs se trouvant sur le même support, le script `checkinstall` peut le déterminer et créer un script à exécuter par le script `postinstall` comme dans l'exemple de mise à niveau (reportez-vous à "Mise à niveau des packages" à la page 181).

```
# 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
```

Script patch_checkinstall

```
# 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
```

Script patch_postinstall

```
# 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
```

Mise à niveau des packages

La procédure de mise à niveau d'un package est très différente de la procédure de remplacement d'un package. Bien qu'il existe des outils particuliers prenant en charge la mise à niveau des packages standard fournis dans le cadre du système d'exploitation Solaris, un package non fourni en standard peut être conçu pour prendre en charge sa propre mise à niveau ; plusieurs des exemples précédents décrivent des packages qui prévoient et contrôlent minutieusement la méthode d'installation sous la direction de l'administrateur. Vous pouvez concevoir le script `request` pour qu'il prenne aussi en charge la mise à niveau directe d'un package. Si l'administrateur décide d'installer un package en remplacement intégral d'un autre package, ne laissant aucun fichier obsolète, les scripts du package peuvent se charger de l'opération.

Le script `request` et le script `postinstall` de cet exemple fournissent un package simple pouvant être mis à niveau. Le script `request` communique avec l'administrateur et définit ensuite un fichier simple dans le répertoire `/tmp` pour supprimer l'ancienne instance du package. Bien que le script `request` crée un fichier (ce qui est interdit), l'opération est passée outre car tous les utilisateurs ont accès au fichier `/tmp`.

Le script `postinstall` exécute ensuite le script shell dans `/tmp`, lequel exécute la commande `pkgrm` requise sur l'ancien package avant de s'auto-supprimer.

Cet exemple illustre une mise à niveau de base. Il se compose de moins de cinquante lignes de code et contient d'assez longs messages. Il pourrait être étendu afin de désinstaller la mise à niveau ou d'apporter d'autres transformations majeures au package en fonction des besoins du concepteur.

Le concepteur de l'interface utilisateur d'une option de mise à niveau doit être certain que l'administrateur connaît la procédure et qu'il a expressément demander une mise à niveau

plutôt qu'une installation en parallèle. La réalisation d'une opération complexe bien comprise telle qu'une mise à niveau est tout à fait acceptable tant que l'interface utilisateur clarifie l'opération.

Script `request`

```
# 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
```

Script `postinstall`

```
# 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
```

Création de packages d'archives de classe

Un package d'archive de classe, qui est une amélioration par rapport à l'ABI (Application Binary Interface), est un package dans lequel certains fichiers ont été regroupés en un seul fichier (ou archive) puis compressés ou chiffrés. Les formats d'archive de classe augmentent la vitesse initiale d'installation de 30 % et améliorent la fiabilité de la procédure d'installation des packages et des patchs sur des systèmes de fichiers potentiellement actifs.

Les sections suivantes fournissent des informations sur la structure du répertoire d'un package d'archive, les mots clés et l'utilitaire `faspac`.

Structure du répertoire d'un package d'archive

L'entrée du package illustrée sur la figure suivante représente le répertoire contenant les fichiers du package. Ce répertoire doit porter le même nom que le package.

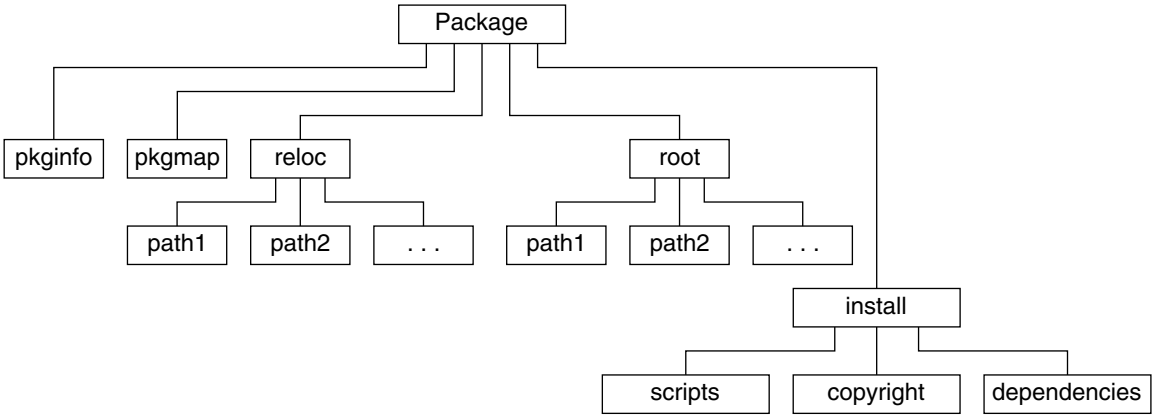


FIGURE 6–1 Structure du répertoire d'un package

Le tableau suivant répertorie les fonctions des fichiers et répertoires contenus dans le répertoire du package.

Élément	Description
`pkginfo`	Fichier décrivant le package dans son ensemble, y compris les variables spéciales d'environnement et la procédure d'installation.
`pkgmap`	Fichier décrivant chaque objet à installer, tel qu'un fichier, un répertoire ou un tube.
`reloc`	Répertoire facultatif contenant les fichiers à installer par rapport au répertoire de base (objets réadressables).
`racine`	Répertoire facultatif contenant les fichiers à installer par rapport au répertoire `root` (objets racine).
`install`	Répertoire facultatif contenant les scripts et autres fichiers auxiliaires (à l'exception de `pkginfo` et de `pkgmap`, tous les fichiers `ftype i` sont inclus).

Le format d'archive de classe permet au développeur du package de regrouper des fichiers provenant des répertoires `reloc` et `root` dans des archives qui peuvent être compressées, chiffrées ou autrement traitées afin d'accélérer l'installation, de réduire la taille du package ou de renforcer la sécurité du package.

L'ABI permet à tout fichier d'un package d'être attribué à une classe. Tous les fichiers d'une classe donnée peuvent être installés sur le disque à l'aide d'une méthode personnalisée définie par un script d'action de classe. Cette méthode personnalisée peut faire appel à des programmes installés sur le système cible ou à des programmes fournis avec le package. Le format résultant ressemble de près au format ABI standard. Comme l'illustre la figure suivante, un autre répertoire est ajouté. Toute classe de fichiers destinée à l'archivage est tout simplement

convertie en un seul fichier et placée dans le répertoire `archive`. Tous les fichiers archivés sont supprimés des répertoires `reloc` et `root`, et un script d'action de classe d'installation est placé dans le répertoire `install`.

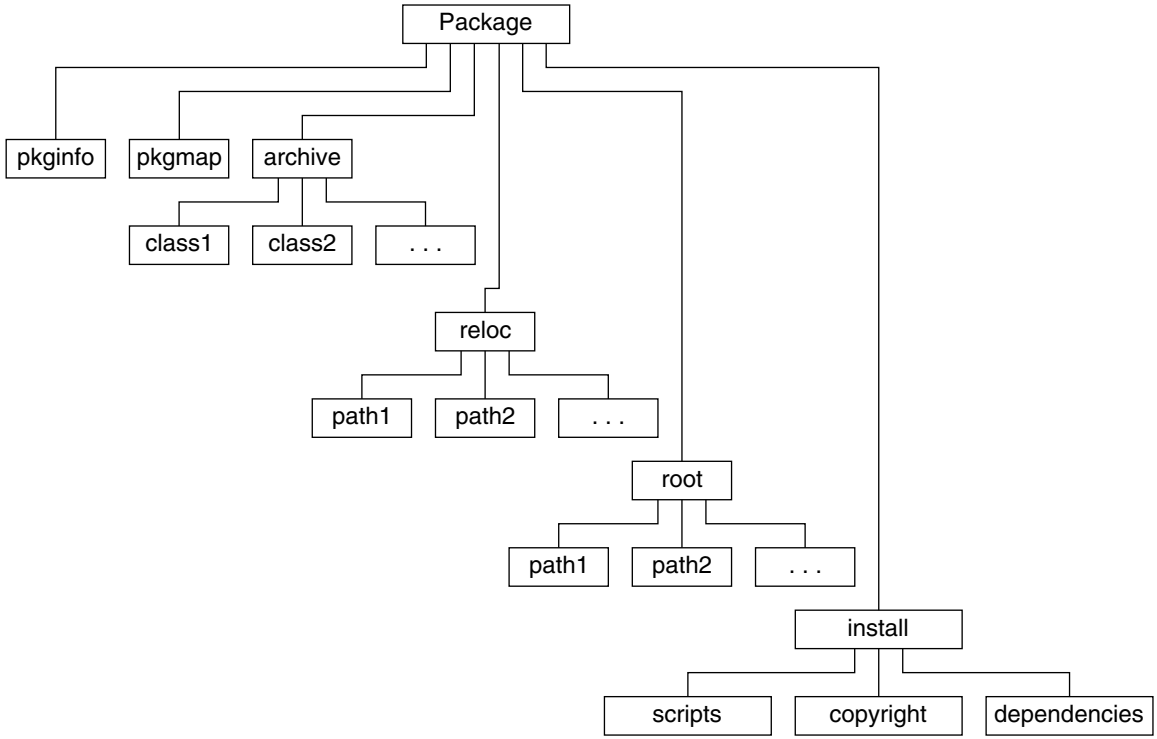


FIGURE 6–2 Structure du répertoire d'un package d'archive

Mots clés de prise en charge des packages d'archive de classe

Pour prendre en charge ce nouveau format d'archive de classe, trois nouvelles interfaces représentées par des mots clés ont une signification particulière dans le fichier `pkginfo`. Ces mots clés sont utilisés pour désigner les classes requérant un traitement particulier. Le format de la déclaration de chaque mot clé est le suivant : `keyword=class1[class2 class3 ...]`. Chaque valeur de mot clé est définie dans le tableau suivant :

Mot-clé	Description
PKG_SRC_NOVERIFY	Ce mot clé indique à pkgadd de ne pas vérifier l'existence ni les propriétés des fichiers figurant dans les répertoires reloc ou root du package fourni s'ils appartiennent à une classe nommée. Il est obligatoire pour toutes les classes archivées car ces fichiers ne se trouvent plus dans un répertoire reloc ni un répertoire root. Il s'agit de fichiers au format privé stockés dans le répertoire archive.
PKG_DST_QKVERIFY	Les fichiers appartenant à ces classes sont vérifiés après l'installation à l'aide d'un algorithme rapide ne nécessitant pas ou peu de données. Cette vérification rapide définit en premier lieu les attributs de chaque fichier puis vérifie que l'opération s'est déroulée correctement. Un test est ensuite effectué pour comparer la taille du fichier et le temps de modification au fichier pkgmap. Aucune somme de contrôle n'est effectuée et la reprise sur erreur est moins efficace que celle offerte par le mécanisme de vérification standard. Dans l'éventualité d'une coupure de courant ou d'une défaillance du disque pendant l'installation, le fichier contenu risque de ne pas concorder avec les fichiers installés. Cette incohérence peut toujours être résolue avec pkgrm.
PKG_CAS_PASSRELATIVE	Le script d'action de classe de l'installation reçoit en général de la part du fichier stdin une liste de paires source/destination lui indiquant quels fichiers installer. Les classes attribuées à PKG_CAS_PASSRELATIVE ne reçoivent pas ces paires source/destination. Elles reçoivent à la place une seule liste dont la première entrée correspond à l'emplacement du package source et les autres correspondent aux chemins de destination. Ceci sert spécifiquement à simplifier l'extraction d'une archive. À partir de l'emplacement du package source, vous pouvez trouver l'archive dans le répertoire archive. Les chemins de destination sont ensuite transmis à la fonction chargée d'extraire le contenu de l'archive. Chaque chemin de destination fourni est absolu ou relatif au répertoire de base selon que son emplacement d'origine se trouve dans root ou dans reloc. Lorsque cette option est sélectionnée, il peut être difficile d'utiliser des chemins relatifs et des chemins absolus dans une même classe.

Un script d'action de classe est nécessaire pour chaque classe archivée. Il s'agit d'un fichier contenant des commandes Bourne shell qui est exécuté par pkgadd pour installer les fichiers à partir de l'archive. Si un script d'action de classe est détecté dans le répertoire install du package, pkgadd délègue toute la responsabilité de l'installation au script. Le script d'action de classe est exécuté avec les droits d'accès à root et peut placer ses fichiers à tout endroit du système cible.

Remarque – Le seul mot clé essentiel à l'implémentation d'un package d'archive de classe est `PKG_SRC_NOVERIFY`. Les autres peuvent être utilisés pour accélérer l'installation ou conserver le code.

Utilitaire `faspac`

L'utilitaire `faspac` convertit un package ABI standard en format d'archive de classe utilisé pour les packages fournis en standard. Cet utilitaire archive des fichiers à l'aide de cpio et compresse à l'aide de compress. Le package obtenu dispose d'un répertoire supplémentaire qui est stocké dans le répertoire supérieur `archive`. Ce répertoire doit contenir toutes les archives nommées par classe. Le répertoire `install` doit contenir les scripts d'action de classe nécessaires à la décompression de chaque archive. Les chemins absolus ne sont pas archivés.

L'utilitaire `faspac` a le format suivant :

`faspac [-m` *Archive Method*`] -a -s -q [-d` *Base Directory*`] /`
`[-x` *Exclude List*`] [`*List of Packages*`]`

Toutes les options de la commande `faspac` sont décrites dans le tableau suivant :

Option	Description
`-m` *Méthode d'archivage*	Spécifie une méthode d'archivage ou de compression. `bzip2` est l'utilitaire de compression utilisé par défaut. Pour utiliser à la place la méthode zip/unzip, utilisez `-m zip` ou pour utiliser la méthode cpio/compress, utilisez `-m cpio`.
`-a`	Corrige les attributs (seul l'utilisateur root peut effectuer cette opération).
`-s`	Indique la conversion des packages de type ABI standard. Cette option convertit un package cpio ou compressé en package conforme ABI standard.
`-q`	Spécifie le mode silencieux.
`-d` *Répertoire de base*	Indique le répertoire contenant tous les packages concernés par l'opération de la ligne de commande. Cette option et l'entrée *Liste de packages* s'excluent mutuellement.
`-x` *Liste d'exclusions*	Spécifie une liste de packages, séparés par des virgules ou entre guillemets, à exclure de l'opération.
Liste de packages	Spécifie la liste des packages à traiter.

Glossaire

ABI Voir Application Binary Interface.

abréviation de package Nom abrégé d'un package défini via le paramètre `PKG` dans le fichier `pkginfo`.

Application Binary Interface Définition de l'interface système binaire entre les applications compilées et le système d'exploitation sur lequel elles sont exécutées.

ASN.1 Voir notation de syntaxe abstraite numéro 1.

certificat de confiance Certificat contenant un seul certificat à clé publique appartenant à une autre entité. Les certificats de confiance sont utilisés lors de la vérification des signatures numériques et de l'initialisation d'une connexion à un serveur sécurisé (SSL).

certificate authority (autorité de certification) Agence telle que Verisign qui délivre les certificats utilisés pour la signature des packages.

classe Nom utilisé pour regrouper des objets de package. Voir aussi script d'action de classe.

clé utilisateur Clé renfermant toutes les informations sensibles de la clé de chiffrement. Ces informations sont stockées dans un format protégé pour empêcher tout accès non autorisé. Les clés utilisateur sont utilisées lorsqu'un package signé est créé.

copyright Droit de propriété et de vente d'une propriété intellectuelle telle qu'un logiciel, un code source ou une documentation. Le droit de propriété doit être déclaré sur le CD-ROM ainsi que sur le texte placé à l'intérieur, que le copyright appartienne à SunSoft ou à un tiers. Le droit de propriété du copyright est également reconnu dans la documentation SunSoft.

dépendance inverse Situation où un autre package dépend de l'existence de votre package. Voir aussi fichier `depend`.

DER Voir Distinguished Encoding Rules.

Distinguished Encoding Rules Représentation binaire d'un objet ASN.1 qui définit la manière dont un objet ASN.1 est sérialisé à des fins de stockage ou de transmission dans les environnements informatiques. Utilisée avec les packages signés.

fichier `compver` Méthode de spécification de la compatibilité ascendante des packages.

fichier d'information Fichier permettant de définir les dépendances des packages, de fournir un message de copyright ou de réserver de l'espace sur le système cible.

fichier de contrôle Fichier contrôlant la méthode, l'emplacement et la décision de l'installation. Voir fichier d'information et script d'installation.

fichier `depend` Méthode de résolution des dépendances de base des packages. Voir aussi fichier `compver`.

identificateur de package Suffixe numérique ajouté par la commande `pkgadd` à l'abréviation d'un package.

instance de package Variante d'un package déterminée en alliant les définitions des paramètres `PKG`, `ARCH` et `VERSION` dans le fichier `pkginfo`.

keystore de package Référentiel de certificats et de clés pouvant être interrogé par les outils d'un package.

liste de patchs Liste des patchs applicables au package en question. Cette liste de patchs est enregistrée dans le package installé dans le fichier `pkginfo`.

nom commun Alias répertorié dans le keystore des packages pour les packages signés.

nom de chemin paramétrique Nom de chemin incluant une spécification de variable.

norme de chiffrement par clé publique numéro 12 Norme décrivant une syntaxe de stockage d'objets de chiffrement sur disque. Le keystore de package utilise ce format.

norme de chiffrement par clé publique numéro 7 Norme décrivant la syntaxe générale des données qu'il est possible de chiffrer, notamment les signatures numériques et les enveloppes numériques. Un package signé contient une signature PKCS7 incorporée.

norme X.509 de l'UIT-T Protocole qui spécifie la syntaxe X.509 très répandue des certificats à clés publiques.

notation de syntaxe abstraite numéro 1 Méthode d'expression des objets abstraits. Par exemple, le langage ASN.1 définit un certificat à clé publique, l'ensemble des objets composant le certificat, et l'ordre dans lequel les objets sont recueillis. Toutefois, le langage ASN.1 n'indique pas la manière dont les objets sont sérialisés à des fins de stockage ou de transmission.

objet de package Autre nom donné à un fichier d'application contenu dans un package à installer sur un système cible.

objet réadressable Objet de package ne requérant pas d'emplacement au chemin absolu sur le système cible. Son emplacement est en fait déterminé au cours de la procédure d'installation. Voir aussi objet réadressable collectivement et objet réadressable individuellement.

objet réadressable collectivement Objet de package placé de manière relative à une base d'installation commune. Voir aussi répertoire de base.

objet réadressable individuellement Objet de package non limité au même répertoire qu'un objet réadressable collectivement. Il est défini à l'aide d'une variable d'installation dans le champ *path* du fichier `prototype`, et l'emplacement de l'installation est déterminé via un script `request` ou un script `checkinstall`.

package Groupe de fichiers et répertoires requis par une application logicielle.

package composite Package contenant des noms de chemin réadressables et absolus.

package incompatible Package non compatible avec le package indiqué. Voir aussi fichier `depend`.

package non signé Package ABI standard ne contenant aucun chiffrement ni aucune signature numérique.

package prérequis Package dépendant de l'existence d'un autre package. Voir aussi fichier `depend`.

package signé Package de flux de données standard comportant une signature numérique vérifiant les points suivants : le package provient de l'entité qui l'a signé, l'entité l'a en effet signé, le package n'a pas été modifié depuis qu'il a été signé par l'entité et l'entité qui l'a signé est une entité de confiance.

PEM Voir Privacy Enhanced Message.

phase d'installation Phase au cours de laquelle un package est installé à l'aide de la commande `pkgadd`.

phase de création Phase au cours de laquelle un package est créé à l'aide de la commande `pkgmk`.

PKCS12 Voir norme de chiffrement par clé publique numéro 12.

PKCS7 Voir norme de chiffrement par clé publique numéro 7.

Privacy Enhanced Message Technique de chiffrement de fichier utilisant le codage 64 de base et quelques en-têtes facultatifs. Technique largement utilisée pour le chiffrement des certificats et des clés privées d'un fichier stocké sur un système de fichiers ou joint à un message électronique.

private key (clé privée) Clé de chiffrement/déchiffrement connue uniquement du ou des parties échangeant des messages secrets. Cette clé privée est utilisée avec les clés publiques pour créer des packages signés.

public key (clé publique) Valeur générée en tant que clé de chiffrement qui, alliée à la clé privée dérivée de la clé publique, peut être utilisée pour chiffrer efficacement des messages et des signatures numériques.

réadressable Objet de package défini dans un fichier `prototype` à l'aide d'un nom de chemin relatif.

répertoire de base Répertoire dans lequel les objets réadressables doivent être installés. Il est défini dans le fichier `pkginfo` à l'aide du paramètre `BASEDIR`.

script d'action de classe Fichier définissant une série d'opérations à effectuer sur un groupe d'objets de package.

script d'installation Script vous permettant de fournir des procédures d'installation personnalisées pour un package.

script de procédure Script définissant les actions qui se déroulent à des moments donnés de l'installation ou de la désinstallation d'un package.

segmenté Package ne pouvant être contenu sur un seul volume, notamment une disquette.

signature numérique Message codé utilisé pour vérifier l'intégrité et la sécurité d'un package.

`tar`	Tape Archive Retrieval (extraction et archivage de/sur bande). Commande Solaris permettant d'ajouter des fichiers à un support ou d'en extraire.
variable d'installation	Variable commençant par une majuscule et qui est évaluée lors de la phase d'installation.
variable de création	Variable commençant par une minuscule et qui est évaluée lors de la phase de création.
X.509	Voir norme X.509 de l'UIT-T.

Index

D

E

F

I

K

L

M

N

O

P

R

S

T

U

V