



Sun Cluster 3.1 データサービス開発ガイド

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

Part No: 817-1019-05
2003 年 2 月, Revision A

Copyright 2003 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

本製品に採用されているテクノロジーに関する知的財産権は Sun Microsystems, Inc. (以下、米国 Sun Microsystems 社とします) が保有しています。特に、これらの知的財産権には、ウェブサイト <http://www.sun.com/patents> にリスト表示されている米国特許、または米国および他の国へ出願中の特許が含まれている可能性があります。

本製品は、本製品やドキュメントの使用、コピー、配布、および逆コンパイルを規制するライセンス規定に従って配布されます。本製品のいかなる部分も、その形態および方法を問わず、Sun およびそのライセンサーの事前の書面による許可なく複製することを禁じます。フォントテクノロジーを含むサードパーティ製のソフトウェアの著作権およびライセンスは、Sun のサプライヤが保有しています。

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company, Ltd. が独占的にライセンスしている米国ならびに他の国における登録商標です。

Sun, Sun Microsystems, Sun のロゴ、docs.sun.com、Answerbook、Answerbook2 Java、Sun StorEdge、Sun One Web Server、Sun Cluster、SunPlex は、米国およびその他の国における米国 Sun Microsystems 社の商標もしくは登録商標です。

OPENLOOK、OpenBoot、JLE は、サン・マイクロシステムズ株式会社の登録商標です。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

ORACLE® は、Oracle Corporation の登録商標です。Netscape™ は、米国およびその他の国における Netscape Communications Corporation の商標もしくは登録商標です。Adobe® のロゴは、Adobe Systems, Incorporated の登録商標です。

連邦政府による取得: 市販ソフトウェア -- 米国政府機関による使用は、標準のライセンス条項に従うものとします。

この製品には、Apache Software Foundation (<http://www.apache.org/>) で開発されたソフトウェアが含まれています。

本書で参照されている製品やサービスに関しては、該当する会社または組織に直接お問い合わせください。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含みそれに限定されず、明示的であるか黙示的であるかを問わない、なんらの保証も行われぬものとします。

本製品が、外国為替および外国貿易管理法 (外為法) に定められる戦略物資等 (貨物または役務) に該当する場合、本製品を輸出または日本国外へ持ち出す際には、サン・マイクロシステムズ株式会社の事前の書面による承諾を得ることのほか、外為法および関連法規に基づく輸出手続き、また場合によっては、米国商務省または米国所轄官庁の許可を得ることが必要です。

原典: *Sun Cluster 3.1 Data Services Developer's Guide*

Part No: 816-3385

Revision A



030224@5533



目次

はじめに	11
1	リソース管理の概要 17
	Sun Cluster アプリケーション環境 17
	RGM モデル 19
	リソースタイプ 19
	リソース 20
	リソースグループ 20
	Resource Group Manager 21
	コールバックメソッド 21
	プログラミングインタフェース 22
	Resource Management API 23
	データサービス開発ライブラリ (DSDL) 23
	SunPlex Agent Builder 23
	Resource Group Manager の管理インタフェース 24
	SunPlex Manager 24
	管理コマンド 24
2	データサービスの開発 27
	アプリケーションの適合性の分析 27
	使用するインタフェースの決定 29
	データサービス作成用開発環境の設定 30
	▼ 開発環境の設定方法 31
	▼ データサービスをクラスタに転送する方法 31
	リソースとリソースタイププロパティの設定 32

リソースタイププロパティの宣言	33
リソースプロパティの宣言	35
拡張プロパティの宣言	38
コールバックメソッドの実装	40
リソースとリソースグループのプロパティ情報へのアクセス	40
メソッドの呼び出し回数への非依存性	41
汎用データサービス	41
アプリケーションの制御	42
リソースの起動と停止	42
リソースの初期化と終了	44
リソースの監視	45
メッセージログのリソースへの追加	46
プロセス管理の提供	46
リソースへの管理サポートの提供	47
フェイルオーバーリソースの実装	48
スケーラブルリソースの実装	48
スケーラブルサービスの妥当性検査	51
データサービスの作成と検証	52
キープアライブの使用法	52
HA データサービスの検証	53
リソース間の依存関係の調節	53
3 リソースタイプの更新	55
概要	55
リソースタイプ登録ファイル	56
リソースタイプ名	56
ディレクティブ	57
RTR ファイル内の RT_Version の変更	57
以前のバージョンの Sun Cluster のリソースタイプ名	58
Type_version リソースプロパティ	58
リソースを別のバージョンへ移行	59
リソースタイプのアップグレード	60
▼ リソースタイプのアップグレード	60
デフォルトのプロパティ値	61
リソースタイプ開発者の文書	62
リソースタイプ名とリソースタイプモニターの実装	62
アプリケーションのアップグレード	63

リソースタイプのアップグレード例	63
パッケージの問題	67
RTR ファイルの変更	67
モニターコードの変更	68
メソッドコードの変更	68
4 Resource Management API リファレンス	71
RMAPI アクセスメソッド	72
RMAPI シェルコマンド	72
C 関数	73
RMAPI コールバックメソッド	77
メソッドの引数	77
終了コード	78
制御および初期化コールバックメソッド	78
管理サポートメソッド	80
ネットワーク関連コールバックメソッド	81
モニター制御コールバックメソッド	81
5 サンプルデータサービス	83
サンプルデータサービスの概要	83
リソースタイプ登録ファイルの定義	84
RTR ファイルの概要	84
サンプル RTR ファイルのリソースタイププロパティ	85
サンプル RTR ファイルのリソースプロパティ	86
すべてのメソッドに共通な機能の提供	90
コマンドインタプリタの指定およびパスのエクスポート	90
PMF_TAG と SYSLOG_TAG 変数の宣言	91
関数の引数の構文解析	92
エラーメッセージの生成	93
プロパティ情報の取得	94
データサービスの制御	94
Start メソッド	95
Stop メソッド	98
障害モニターの定義	100
検証プログラム	101
Monitor_start メソッド	106
Monitor_stop メソッド	107

	Monitor_check メソッド	108
	プロパティ更新の処理	109
	Validate メソッド	110
	Update メソッド	114
6	データサービス開発ライブラリ (DSDL)	117
	DSDL の概要	117
	構成プロパティの管理	118
	データサービスの起動と停止	119
	障害モニターの実装	119
	ネットワークアドレス情報へのアクセス	120
	実装したリソースタイプのデバッグ	120
	高可用性ローカルファイルシステムの有効化	121
7	リソースタイプの設計	123
	RTR ファイル	123
	Validate メソッド	124
	Start メソッド	126
	Stop メソッド	127
	Monitor_start メソッド	128
	Monitor_stop メソッド	129
	Monitor_check メソッド	129
	Update メソッド	129
	Init、Fini、Boot の各メソッド	130
	障害モニターデーモンの設計	131
8	サンプル DSDL リソースタイプの実装	135
	X Font Server について	135
	構成ファイル	136
	TCP ポート番号	136
	命名規約	136
	SUNW.xfnts の RTR ファイル	137
	scds_initialize の呼び出し	138
	xfnts_start メソッド	138
	起動前のサービスの検証	138
	サービスの起動	139
	svc_start からの復帰	140

xfnts_stop メソッド	143
xfnts_monitor_start メソッド	144
xfnts_monitor_stop メソッド	145
xfnts_monitor_check メソッド	147
SUNW.xfnts 障害モニター	147
xfnts_probe のメインループ	148
svc_probe 関数	150
障害モニターのアクションの決定	153
xfnts_validate メソッド	153
xfnts_update メソッド	156

9 SunPlex Agent Builder 157

Agent Builder の使用	158
アプリケーションの分析	158
Agent Builder のインストールと構成	158
Agent Builder の起動	159
作成画面の使用	161
構成画面の使用	163
完成した作業内容の再利用	167
既存のリソースタイプのクローンの作成	167
生成されたソースコードの編集	168
コマンド行バージョンの Agent Builder の使用	168
ディレクトリ構造	169
出力	170
ソースファイルとバイナリファイル	170
ユーティリティスクリプトとマニュアルページ	171
サポートファイル	172
パッケージディレクトリ	173
rtconfig ファイル	173
ナビゲーション	174
「ブラウズ」ボタン	174
メニュー	175

10 汎用データサービス 177

汎用データサービスの概要	177
コンパイル済みリソースタイプ	178
GDS を使用する利点	178

	GDS を使用するサービスの作成方法	178
	GDS の必須入力	179
	GDS の任意入力	180
	SunPlex Agent Builder を使って GDS ベースのサービスを作成	183
	Agent Builder を使って GDS ベースのサービスを作成	183
	SunPlex Agent Builder の出力	187
	標準的な Sun Cluster 管理コマンドを使って GDS ベースのサービスを作成	187
	▼ 標準的な Sun Cluster 管理コマンドを使って GDS ベースの高可用性サービスを作成する方法	188
	▼ 標準的な Sun Cluster 管理コマンドを使って GDS ベースのスケーラブルサービスを作成する方法	188
	SunPlex Agent Builder のコマンド行インタフェース	189
	▼ SunPlex Agent Builder のコマンド行インタフェースを使って GDS ベースのサービスを作成する方法	189
11	DSDL のリファレンス	193
	DSDL Functions	193
	汎用関数	193
	プロパティ関数	195
	ネットワークリソースアクセス関数	195
	TCP 接続を使用する障害監視	196
	PMF 関数	196
	障害監視関数	197
	ユーティリティ関数	197
A	標準プロパティ	199
	リソースタイププロパティ	199
	リソースプロパティ	203
	リソースグループプロパティ	212
	リソースプロパティの属性	217
B	データサービスのコード例	219
	リソースタイプ登録ファイルのリスト	219
	Start メソッドのコードリスト	222
	Stop メソッドのコードリスト	225
	gettime ユーティリティのコードリスト	227
	PROBE プログラムのコードリスト	228

	Monitor_start メソッドのコードリスト	234
	Monitor_stop メソッドのコードリスト	236
	Monitor_check メソッドのコードリスト	238
	Validate メソッドのコードリスト	240
	Update メソッドのコードリスト	244
C	サンプル DSDL リソースタイプのコード例	247
	xfnts.c	247
	xfnts_monitor_check メソッド	259
	xfnts_monitor_start メソッド	260
	xfnts_monitor_stop メソッド	261
	xfnts_probe メソッド	262
	xfnts_start メソッド	265
	xfnts_stop メソッドのコードリスト	267
	xfnts_update メソッドのコードリスト	268
	xfnts_validate メソッドのコードリスト	269
D	RGM の有効な名前と値	271
	RGM の有効な名前	271
	RGM の値	272
E	非クラスタ対応のアプリケーションの要件	273
	多重ホストデータ	273
	多重ホストデータを配置するためのシンボリックリンクの使用	274
	ホスト名	275
	多重ホームホスト	275
	INADDR_ANY へのバインドと特定の IP アドレスへのバインド	276
	クライアントの再試行	277
	索引	279

はじめに

このマニュアルでは、RMAPI (Resource Management (リソース管理) API) を使用して Sun Cluster データサービスを開発する方法について説明します。

対象読者

このマニュアルは、Sun のソフトウェアとハードウェアについて豊富な知識を持っている経験のある開発者を対象にしています。このマニュアルの情報は、Solaris™ オペレーティング環境の知識があることを前提としています。

内容の紹介

このマニュアルには、次の章と付録が含まれています。

- 第 1 章では、データサービスを開発するのに必要な概念について説明します。
- 第 2 章では、データサービスの開発に関する詳細な情報を説明します。
- 第 4 章では、Resource Management API (RMAPI) を構成するアクセス関数とコールバックメソッドに関する情報を説明します。
- 第 5 章では、`in.named()` アプリケーション用の Sun Cluster データサービスの例を示します。
- 第 6 章では、Data Services Development Library (DSDL) を形成するアプリケーションプログラミングインタフェースの概要を説明します。
- 第 7 章では、リソースタイプ的设计と実装における DSDL の代表的な使用例について説明します。

- 第 8 章では、DSDL により実装されるリソースタイプの例を説明します。
- 第 9 章では、SunPlex Agent Builder について説明します。
- 第 10 章では、一般的なデータサービスの作成方法について説明します。
- 第 11 章では、DSDL API 関数について説明します。
- 付録 A では、標準リソースタイプ、リソースグループ、およびリソースプロパティについて説明します。
- 付録 B では、データサービスの例について、それぞれのメソッドの完全なコードを示します。
- 付録 C では、SUNW.xfnts() リソースタイプにおける各メソッドの完全なコードを示します。
- 付録 D では、Resource Group Manager (RGM) の名前と値についての文字の要件を説明します。
- 付録 E では、クラスタに対応していない、通常のアプリケーションを高可用性に適用させる要件を説明します。

表記上の規則

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

表 P-1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% su password:
<i>AaBbCc123</i>	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm <i>filename</i> と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。

表 P-1 表記上の規則 (続き)

字体または記号	意味	例
「」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	<pre>sun% grep `^#define \ XV_VERSION_STRING`</pre>

コード例は次のように表示されます。

■ C シェル

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

■ C シェルのスーパーユーザー

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

■ Bourne シェルおよび Korn シェル

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

■ Bourne シェルおよび Korn シェルのスーパーユーザー

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

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

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

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

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

一般規則

- このマニュアルでは、英語環境での画面イメージを使っています。このため、実際に日本語環境で表示される画面イメージとこのマニュアルで使っている画面イメージが異なる場合があります。本文中で画面イメージを説明する場合には、日本語のメニュー、ボタン名などの項目名と英語の項目名が、適宜併記されています。

- このマニュアルでは、「IA」という用語は、Intel 32 ビットのプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium III Xeon、Celeron、Pentium III、Pentium III Xeon、Pentium 4 の各プロセッサ、および AMD が提供する互換マイクロプロセッサチップが含まれます。

関連マニュアル

説明内容	タイトル	パート番号
概念	『Sun Cluster 3.1 の概念』	817-1018
ハードウェア	『Sun Cluster 3.1 Hardware Administration Manual』	817-0168
	『Sun Cluster 3.1 With Sun StorEdge 3310 Array Manual』	817-0180
	『Sun Cluster 3.1 With Sun StorEdge 3900 or 6900 Series System Manual』	817-0179
	『Sun Cluster 3.1 With Sun StorEdge 9900 Series Installation and Service Manual』	817-0177
	『Sun Cluster 3.1 With Sun StorEdge A1000 or Netra st A1000 Array Manual』	817-0171
	『Sun Cluster 3.1 With Sun StorEdge A3500/A3500FC System Manual』	817-0174
	『Sun Cluster 3.1 With Sun StorEdge A5x00 Array Installation and Service Manual』	817-0173
	『Sun Cluster 3.1 With Sun StorEdge D1000 or Netra st D1000 Disk Array Manual』	817-0170
	『Sun Cluster 3.1 With Sun StorEdge Multipack Enclosure Manual』	817-0169
	『Sun Cluster 3.1 With Sun StorEdge Netra D130 or StorEdge S1 Enclosure Manual』	817-0178
	『Sun Cluster 3.1 With Sun StorEdge T3 or T3+ Array Partner-Group Configuration Manual』	817-0176
	『Sun Cluster 3.1 With Sun StorEdge T3 or T3+ Array Single Controller Configuration Manual』	817-0175
	『Sun Cluster With Sun StorEdge D2 Array Manual』	817-0172

説明内容	タイトル	パート番号
データサービス	『Sun Cluster 3.1 データサービスのインストールと構成』	817-1017
API 開発	『Sun Cluster 3.1 データサービス開発ガイド』	817-1019
管理	『Sun Cluster 3.1 のシステム管理』	817-1021
エラーメッセージ	『Sun Cluster 3.1 Error Messages Guide』	816-3382
マニュアルページ	『Sun Cluster 3.1 Man Page Reference Manual』	816-5251
最新情報	『Sun Cluster 3.1 ご使用にあたって』	817-1022
	『Sun Cluster 3.1 Release Notes Supplement』	816-3381

Sun のオンラインマニュアル

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

問い合わせについて

Sun Cluster をインストールまたは使用しているときに問題が発生した場合は、ご購入先に連絡し、次の情報をお伝えください。

- 名前と電子メールアドレス (利用している場合)
- 会社名、住所、および電話番号
- システムのモデルとシリアル番号
- オペレーティング環境のリリース番号 (たとえば、Solaris 9)
- Sun Cluster のリリース番号 (たとえば、Sun Cluster 3.1)

ご購入先に知らせる、システム上の各ノードについての情報を収集するには、次のコマンドを使用します。

コマンド	機能
<code>prtconf -v</code>	システムメモリのサイズと周辺デバイス情報を表示する

コマンド	機能
<code>psrinfo -v</code>	プロセッサの情報を表示する
<code>showrev -p</code>	インストールされているパッチを報告する
<code>prtdiag -v</code>	システム診断情報を表示する
<code>scinstall -pv</code>	Sun Cluster のリリースとパッケージバージョン情報を表示する
<code>scrgadm -pvv</code>	既存のリソースタイプ、リソースグループ、およびリソースについての静的な属性について、詳細なリストを表示する
<code>scstat -g</code>	すべてのリソースおよびリソースグループについて、動的な状態情報を表示する

上記の情報にあわせて、`/var/adm/messages` ファイルの内容もご購入先にお知らせください。

第 1 章

リソース管理の概要

このマニュアルでは、Oracle、Sun™ One Web Server、DNS などのソフトウェアアプリケーション用のリソースタイプを作成するためのガイドラインを説明します。したがって、このマニュアルはリソースタイプの開発者を対象としています。

この章では、データサービスを開発するために理解しておく必要がある概念について説明します。この章の内容は、次のとおりです。

- 17 ページの「Sun Cluster アプリケーション環境」
- 19 ページの「RGM モデル」
- 21 ページの「Resource Group Manager」
- 21 ページの「コールバックメソッド」
- 22 ページの「プログラミングインタフェース」
- 24 ページの「Resource Group Manager の管理インタフェース」

注 - このマニュアルでは、「リソースタイプ」と「データサービス」という用語を同じ意味で使用しています。また、このマニュアルではほとんど使用されることはありませんが、「エージェント」という用語も「リソースタイプ」や「データサービス」と同じ意味で使用されます。

Sun Cluster アプリケーション環境

Sun Cluster システムを使用すると、アプリケーションを高度な可用性とスケーラビリティを備えたリソースとして実行および管理できます。RGM (Resource Group Manager) というクラスタ機能は、高可用性とスケーラビリティを実現するための機構を提供します。この機能を利用するためのプログラミングインタフェースを形成する要素は、次のとおりです。

- ユーザーが作成するコールバックメソッド。RGM は、このコールバックメソッドを利用してクラスタ上のアプリケーションを制御します。

- Resource Management API (RMAPI)。コールバックメソッドの作成に使用する低レベルの API コマンドおよび API 関数です。これらの API は、libscha.so ライブラリに実装されます。
- クラスタ上のプロセスを監視および再起動するプロセス管理機能。
- データサービス開発ライブラリ (Data Service Development Library: DSDL)。低レベル API およびプロセス管理機能をより高レベルでカプセル化し、コールバックメソッドの作成を支援するいくつかの機能を追加するライブラリ関数です。これらの関数は、libdsdev.so ライブラリに実装されます。

次の図は、これらの要素の相互関係を示しています。

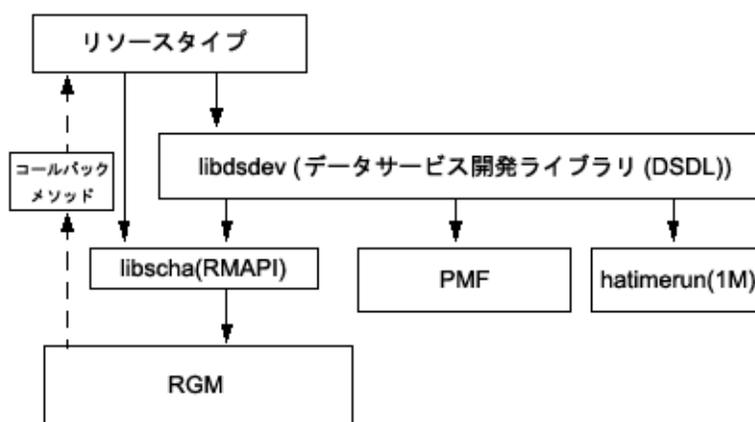


図 1-1 プログラミングアーキテクチャ

Sun Cluster パッケージには、データサービスの作成プロセスを自動化する SunPlex Agent Builder™ というツールが含まれます (第 9 章を参照)。Agent Builder はデータサービスのコードを C 言語または Korn シェル (ksh) のどちらでも生成できます。前者の場合は DSDL 関数、後者の場合は低レベルの API コマンドを使ってコールバックメソッドを作成します。

RGM は各クラスタ上でデーモンとして動作して、事前構成したポリシーに従って、選択したノード上のリソースを自動的に起動および停止します。リソースの高可用性を実現するために、RGM は、ノードが異常終了または再起動すると、影響を受けるノード上でリソースを停止し、別のノード上でリソースを起動します。また、リソースに固有のモニター (監視機能) を起動および停止することによって、障害のあるリソースを検出し、別のノードに再配置したり、さまざまな視点からリソース性能を監視します。

RGM はフェイルオーバーリソースとスケラブルリソースの両方をサポートします。フェイルオーバーリソースとは、同時に 1 つのノード上だけでオンラインになることができるリソースのことです。スケラブルリソースとは、同時に複数のノード上でオンラインになることができるリソースのことです。

RGM モデル

ここでは、基本的な用語をいくつか紹介し、RGM とそれに関連するインタフェースについて詳細に説明します。

RGM は、「リソースタイプ」、「リソース」、「リソースグループ」という 3 種類の相互に関連するオブジェクトを処理します。これらのオブジェクトを紹介するために、次のような例を使用します。

開発者は、既存の Oracle DBMS アプリケーションを高可用性にするためのリソースタイプ `ha-oracle` を実装します。エンドユーザーは、マーケティング、エンジニアリング、および財務ごとに異なるデータベースを定義し、それぞれのリソースタイプを `ha-oracle` にします。クラスタ管理者は、上記リソースを異なるリソースグループに分類することによって、異なるノード上で実行したり、個別にフェイルオーバーできるようにします。開発者は、もう 1 つのリソースタイプ `ha-calendar` を作成し、Oracle データベースを必要とする高可用性のカレンダーサーバーを実装します。クラスタ管理者は、財務カレンダーリソースと財務データベースリソースを同じリソースグループに分類することによって、両方のリソースを同じノード上で実行したり、一緒にフェイルオーバーできるようにします。

リソースタイプ

リソースタイプは、クラスタ上で実行されるソフトウェアアプリケーション、アプリケーションをクラスタリソースとして管理するために RGM がコールバックメソッドとして使用する制御プログラムおよびクラスタの静的な構成の一部を形成するプロパティセットからなります。RGM は、リソースタイププロパティを使って特定のタイプのリソースを管理します。

注 - リソースタイプは、ソフトウェアアプリケーションだけでなく、その他のシステムリソース (ネットワークアドレスなど) も表します。

リソースタイプの開発者は、リソースタイププロパティを指定し、その値をリソースタイプ登録 (RTR) ファイルに設定します。RTR ファイルの形式は、32 ページの「リソースとリソースタイププロパティの設定」と `rt_reg` のマニュアルページの記述に従い明確に定義されています。リソースタイプ登録ファイルの例については、84 ページの「リソースタイプ登録ファイルの定義」を参照してください。

表 A-1 に、リソースタイププロパティのリストを示します。

クラスタ管理者は、リソースタイプの実装と実際のアプリケーションをクラスタにインストールし、登録します。さらに登録手順で、リソースタイプ登録ファイルの情報をクラスタ構成に入力します。データサービスの登録手順については、『*Sun Cluster 3.1 データサービスのインストールと構成*』を参照してください。

リソース

リソースは、そのリソースタイプからプロパティと値を継承します。さらに、開発者は、リソースタイプ登録ファイルでリソースプロパティを宣言できます。リソースプロパティのリストについては、表 A-2 を参照してください。

クラスタ管理者は、リソースタイプ登録 (RTR) ファイルにプロパティを指定することによって、特定のプロパティの値を変更できます。たとえば、プロパティ定義に値の許容範囲を指定しておきます。これにより、プロパティが調節可能なときに、作成時、常時、不可などを指定できます。このような許容範囲内であれば、クラスタ管理者は管理コマンドでプロパティを変更できます。

クラスタ管理者は、同じタイプのリソースを多数作成して、各リソースに独自の名前とプロパティ値のセットを持たせることができます。これによって、実際のアプリケーションの複数のインスタンスをクラスタ上で実行できます。このとき、各インスタンスにはクラスタ内で一意の名前が必要です。

リソースグループ

各リソースはリソースグループに構成する必要があります。RGM は、同じグループのすべてのリソースを同じノード上でオンラインかオフラインにします。このとき、グループ内の個々のリソースに対してコールバックメソッドを呼び出します。

リソースグループがオンラインになっているノードを主ノードと呼びます。リソースグループは、その主ノードによってマスター (制御) されます。各リソースグループは、クラスタ管理者が設定した独自の *Nodelist* プロパティを持っており、これによってリソースグループの潜在的な主ノードを識別します。

リソースグループはプロパティセットも持っています。このようなプロパティには、クラスタ管理者が設定できる構成プロパティや、RGM が設定してリソースグループのアクティブな状態を反映する動的プロパティがあります。

RGM は 2 種類のリソースグループ、フェイルオーバー (failover) とスケーラブル (scalable) を定義します。フェイルオーバーリソースグループは、同時に 1 つのノード上だけでオンラインになることができます。一方、スケーラブルリソースグループは、同時に複数のノード上でオンラインになることができます。RGM は、各種類のリソースグループを作成するためのプロパティセットを提供します。このようなプロパティの詳細については、48 ページの「フェイルオーバーリソースの実装」と 48 ページの「スケーラブルリソースの実装」を参照してください。

リソースグループのプロパティのリストについては、表 A-3 を参照してください。

Resource Group Manager

Resource Group Manager (RGM) は rgmd デーモンとして実装され、クラスタの各メンバー (ノード) 上で動作します。rgmd プロセスはすべて互いに通信し、単一のクラスタ規模の機能として動作します。

RGM は、次の機能をサポートします。

- ノードが起動またはクラッシュしたとき、管理対象のすべてのリソースグループを適切なマスター上で自動的にオンラインにし、その可用性を維持します。
- 特定のリソースが異常終了した場合、そのモニタープログラムはリソースグループを同じマスター上で再起動するか、新しいマスターに切り替えるかを要求できません。
- クラスタ管理者は管理コマンドを発行して、次のいずれかのアクションを要求できます。
 - リソースグループをマスターする権利の変更
 - リソースグループ内の特定のリソースの有効化または無効化
 - リソース、リソースグループ、リソースタイプの作成、削除、変更

RGM は、構成を変更するとき、そのアクションをクラスタのすべてのメンバー (ノード) 間で調整します。このような動作を「再構成」と呼びます。RGM は、個々のリソースの状態を変更をするため、各リソース上でリソースタイプに固有のコールバックメソッドを呼び出します。コールバックメソッドについては、21 ページの「コールバックメソッド」を参照してください。

コールバックメソッド

Sun Cluster フレームワークは、コールバックメソッドを使用して、データサービスと RGM 間の通信を実現します。また、コールバックメソッド (引数と戻り値を含む) のセットと、RGM が個々のメソッドを呼び出す環境を定義します。

データサービスを作成するには、個々のコールバックメソッドのセットをコーディングし、個々のメソッドを RGM から呼び出し可能な制御プログラムとして実装します。つまり、データサービスは単一の実行可能コードではなく、多数の実行可能なスクリプト (ksh) またはバイナリ (C 言語) から構成されており、それぞれを RGM から直接呼び出すことができます。

コールバックメソッドを RGM に登録するには、リソースタイプ登録 (RTR) ファイルを使用します。RTR ファイルには、データサービスとして実装した各メソッドのプログラムを指定します。システム管理者がデータサービスをクラスタに登録すると、RGM は RTR ファイルにあるさまざまな情報の中からコールバックプログラムの識別情報を読み取ります。

リソースタイプの必須コールバックメソッドは、起動メソッド (Start または `Prenet_start`) と停止メソッド (Stop または `Postnet_stop`) だけです。

コールバックメソッドは、次のようなカテゴリに分類できます。

- 制御および初期化メソッド
 - Start と Stop は、オンラインまたはオフラインにするグループ内のリソースを起動または停止します。
 - Init、Fini、Boot は、リソース上で初期化と終了コードを実行します。
- 管理サポートメソッド
 - Validate は、管理アクションによって設定されるプロパティを確認します。
 - Update は、オンラインリソースのプロパティ設定を更新します。
- ネットワーク関連メソッド
 - `Prenet_start` と `Postnet_stop` は、同じリソースグループ内のネットワークアドレスが「起動」に構成される前、または「停止」に構成された後に、特別な起動アクションまたは停止アクションを実行します。
- モニター制御メソッド
 - `Monitor_start` と `Monitor_stop` は、リソースのモニターを起動または停止します。
 - `Monitor_check` は、リソースグループがノードに移動される前に、ノードの信頼性を査定します。

コールバックメソッドの詳細については、第 4 章と `rt_callbacks (1HA)` のマニュアルページを参照してください。コールバックメソッドの使用例については、第 5 章および第 8 章を参照してください。

プログラミングインタフェース

リソース管理アーキテクチャは、データサービス用のコードを作成するため、低レベルのベース API、ベース API 上のものより高いレベルのライブラリを提供します。さらに、いくつかの基本的な入力情報をもとにデータサービスを自動的に生成するツール、SunPlex Agent Builder を提供します。

Resource Management API

データサービスは、Resource Management API (RMAPI) の低レベルルーチンを使って、システム内のリソース、リソースタイプ、リソースグループの情報にアクセスします。ローカルの再起動やフェイルオーバーの要求、リソースの状態の設定もを行います。これらの関数にアクセスするには、`libscha.so` ライブラリを使用します。RMAPI は、これらのコールバックメソッドを、シェルコマンドまたは C 関数のかたちで提供できます。RMAPI ルーチンの詳細については、`scha_calls(3HA)` のマニュアルページと第 4 章を参照してください。データサービスコールバックメソッドにおける RMAPI の使用例については、第 5 章を参照してください。

データサービス開発ライブラリ (DSDL)

データサービス開発ライブラリ (Data Service Development Library: DSDL) は、RMAPI 上に構築されており、RGM のメソッドコールバックモデルを基盤にして上位レベルの統合フレームワークを提供します。DSDL は、次のようなさまざまなデータサービス開発向けの機能を提供します。

- `libscha.so`—低レベルのリソース管理 API
- PMF—プロセスとその子孫の監視、停止したプロセスの再起動などを実行するプロセス管理機能 (`pmfadm(1M)` および `rpc.pmf(1M)` のマニュアルページを参照)
- `hatimerun`—タイムアウトを適用してプログラムを実行するための機能 (`hatimerun(1M)` のマニュアルページを参照)

DSDL は、大多数のアプリケーションに対して、データサービスの構築に必要なほとんどまたはすべての機能を提供します。DSDL は、低レベルの API の代わりになるのではなく、低レベルの API をカプセル化および拡張するためのものであることに注意してください。実際、多くの DSDL 関数は `libscha.so` 関数を呼び出します。`libscha.so` 関数を直接呼び出すこともできますが、DSDL を使用することにより、データサービスの大部分を作成できます。DSDL 関数は `libdsdev.so` ライブラリとして実装されています。

DSDL の詳細については、第 6 章と `scha_calls(3HA)` のマニュアルページを参照してください。

SunPlex Agent Builder

Agent Builder は、データサービスの作成を自動化するツールです。ユーザーがターゲットアプリケーションの基本情報を入力すると、ソースコードと実行コード (C または Korn シェル)、カスタマイズされた RTR ファイル、Solaris パッケージを利用して、ユーザーの代わりにデータサービスを作成します。

大多数のアプリケーションでは、Agent Builder を使用することにより、わずかなコードを手作業で変更するだけで完全なデータサービスを生成できます。追加プロパティの妥当性検査を必要とするような、より要件の厳しいアプリケーションには、

Agent Builder では対応できないこともあります。しかし、このような場合でもコードの大部分を生成できるので、手作業によるコーディングは残りの部分だけで済みます。Agent Builder を使用すれば、少なくとも独自の Solaris パッケージを生成することができます。

Resource Group Manager の管理インタフェース

Sun Cluster はクラスタを管理するために、グラフィカルユーザーインタフェースとコマンドセットの両方を提供します。

SunPlex Manager

SunPlex Manager は、次の作業を実行できる Web ベースのツールです。

- クラスタのインストール
- クラスタの管理
- リソースおよびリソースグループの作成と構成
- Sun Cluster ソフトウェアを使ったデータサービスの構成

SunPlex Manager のインストール方法、SunPlex Manager によるクラスタソフトウェアのインストール方法については、『*Sun Cluster 3.1* ソフトウェアのインストール』を参照してください。管理作業については、SunPlex Manager のオンラインヘルプを参照してください。

管理コマンド

RGM オブジェクトの管理用 Sun Cluster コマンドは、`scrgadm(1M)`、`scswitch(1M)`、および `scstat(1M) - g` の 3 つです。

`scrgadm` コマンドでは、RGM が使用するリソースタイプ、リソースグループ、リソースオブジェクトの表示、作成、構成が可能です。このコマンドはクラスタの管理インタフェースの一部であり、この章の残りで説明しているアプリケーションインタフェースとは異なったプログラミングコンテキストで使用されます。しかし、このコマンドを使って、API が動作するクラスタ構成を構築することもできます。管理インタフェースを理解すると、アプリケーションインタフェースも理解しやすくなります。`scrgadm` コマンドで実行できる管理作業の詳細については、`scrgadm(1M)` のマニュアルページを参照してください。

scswitch コマンドでは、指定のノード上のリソースグループのオンラインとオフラインの切り替えや、リソースまたはそのモニターの有効と無効の切り替えが可能です。scswitch コマンドで実行できる管理作業の詳細については、scswitch(1M) のマニュアルページを参照してください。

scstat -g コマンドでは、すべてのリソースグループおよびリソースの現在の動的な状態を表示できます。

第 2 章

データサービスの開発

この章では、データサービスを開発するための詳細な方法について説明します。

この章の内容は、次のとおりです。

- 27 ページの「アプリケーションの適合性の分析」
- 29 ページの「使用するインタフェースの決定」
- 30 ページの「データサービス作成用開発環境の設定」
- 32 ページの「リソースとリソースタイププロパティの設定」
- 40 ページの「コールバックメソッドの実装」
- 42 ページの「アプリケーションの制御」
- 45 ページの「リソースの監視」
- 46 ページの「メッセージログのリソースへの追加」
- 46 ページの「プロセス管理の提供」
- 47 ページの「リソースへの管理サポートの提供」
- 48 ページの「フェイルオーバーリソースの実装」
- 48 ページの「スケーラブルリソースの実装」
- 52 ページの「データサービスの作成と検証」

アプリケーションの適合性の分析

データサービスを作成するための最初の手順では、ターゲットアプリケーションが高可用性またはスケーラビリティを備えるための要件を満たしているかどうかを判定します。すべての要件を満たしていない場合は、要件を満たすようにアプリケーションのソースコードを変更します。

次に、アプリケーションが高可用性またはスケーラビリティを備えるための要件を要約します。要件に関する詳細情報を確認したい場合や、アプリケーションのソースコードを変更する必要がある場合は、付録 B を参照してください。

注 - スケーラブルサービスを実現するためには、次に示す高可用性の要件をすべて満たした上で、いくつかの追加要件も満たしている必要があります。

- Sun Cluster 環境では、ネットワーク対応 (クライアントサーバーモデル) とネットワーク非対応 (クライアントレス) のアプリケーションはどちらも、高可用性またはスケーラビリティを備えることが可能です。ただし、タイムシェアリング環境では、アプリケーションはサーバー上で動作し、telnet または rlogin 経由でアクセスされるため、Sun Cluster の可用性を強化することはできません。
- アプリケーションはクラッシュに対する耐障害性 (クラッシュトレラント) を備えていなければなりません。つまり、ノードが予期せぬ停止状態になった後、アプリケーションは再起動時に必要なディスクデータを復元できなければなりません。さらに、クラッシュ後の復元時間にも制限が課せられます。ディスクを復元し、アプリケーションを再起動できる能力は、データの整合性に関わる問題であるため、クラッシュトレラントであることは、アプリケーションが高可用性を備えるための前提条件となります。データサービスは接続を復元できる必要はありません。
- アプリケーションは、自身が動作するノードの物理的なホスト名に依存してはなりません。詳細については、275 ページの「ホスト名」を参照してください。
- アプリケーションは、複数の IP アドレスが構成されている環境で正しく動作する必要があります。たとえば、ノードが複数のパブリックネットワーク上に存在する多重ホームホスト環境や、単一のハードウェアインタフェース上に複数の論理インタフェースが構成されているノードが存在する環境で正しく動作しなければなりません。
- 高可用性を備えるには、アプリケーションデータはクラスタファイルシステム内に格納されている必要があります。273 ページの「多重ホストデータ」を参照してください。

アプリケーションがデータの格納先を示すのに固定されたパス名を使用している場合、アプリケーションのソースコードを変更しなくても、そのパスをクラスタファイルシステム内の場所を指すシンボリックリンクに変更できる場合があります。詳細については、274 ページの「多重ホストデータを配置するためのシンボリックリンクの使用」を参照してください。
- アプリケーションのバイナリとライブラリは、ローカルの各ノードまたはクラスタファイルシステムのどちらにも格納できます。クラスタファイルシステム上に格納する利点は、1 箇所にインストールするだけで済む点です。欠点としては、アプリケーションが RGM の制御下で動作している間はバイナリファイルが使用中になるので、ローリングアップグレードの問題が生じることが挙げられます。
- 初回の照会がタイムアウトした場合、クライアントは自動的に照会を再試行できる必要があります。アプリケーションとプロトコルがすでに単一サーバーのクラッシュと再起動に対応できている場合、関連するリソースグループのフェイルオーバーまたはスイッチオーバーにも対応する必要があります。詳細については、277 ページの「クライアントの再試行」を参照してください。
- アプリケーションは、クラスタファイルシステム内で UNIX ドメインソケットまたは名前付きパイプを使用してはなりません。

さらに、スケーラブルサービスは、次の要件も満たしている必要があります。

- アプリケーションは、複数のインスタンスを実行でき、すべてのインスタンスがクラスタファイルシステム内の同じアプリケーションデータを処理できる必要があります。
- アプリケーションは、複数のノードからの同時アクセスに対してデータの整合性を保証する必要があります。
- アプリケーションは、クラスタファイルシステムのように、広域的に使用可能な機構を備えたロック機能を実装している必要があります。

スケーラブルサービスの場合、アプリケーションの特性により負荷均衡ポリシーが決定されます。たとえば、負荷均衡ポリシー `LB_WEIGHTED` は、任意のインスタンスがクライアントの要求に回答できるポリシーですが、クライアント接続にサーバー上のメモリー内キャッシュを使用するアプリケーションには適用されません。この場合、特定のクライアントのトラフィックをアプリケーションの1つのインスタンスに制限する負荷均衡ポリシーを指定する必要があります。負荷均衡ポリシー `LB_STICKY` と `LB_STICKY_WILD` は、クライアントからのすべての要求を同じアプリケーションインスタンスに繰り返して送信します。この場合、アプリケーションはメモリー内キャッシュを使用できます。異なるクライアントから複数の要求が送信された場合、RGM はサービスの複数のインスタンスに要求を分配します。スケーラブルデータサービスに対応した負荷均衡ポリシーを設定する方法については、48 ページの「スケーラブルリソースの実装」を参照してください。

使用するインタフェースの決定

Sun Cluster 開発者サポートパッケージ (SUNWscdev) は、データサービスメソッドのコーディング用に 2 種類のインタフェースセットを提供します。

- Resource Management API (RMAPI) - 低レベルのルーチンセット (`libscha.so` ライブラリとして実装されている)
- データサービス開発ライブラリ (Data Service Development Library: DSDL) - RMAPI の機能をカプセル化し、いくつかの追加機能を提供する、より高いレベルの関数セット (`libdsdev.so` ライブラリとして実装されている)

Sun Cluster 開発者サポートパッケージには、データサービスの作成を自動化するツールである SunPlex Agent Builder も含まれています。

次に、データサービスを開発する際の推奨手順を示します。

1. C 言語または Korn シェル (ksh) のどちらでコーディングするかを決定します。DSDL は C 言語用のインタフェースしか提供しないため、Korn シェルでコーディングする場合は使用できません。
2. Agent Builder を実行します。必要な情報を入力するだけで、ソースコード、実行可能コード、RTR ファイル、パッケージを含むデータサービスを生成できます。

3. 生成されたデータサービスをカスタマイズする必要がある場合は、生成されたソースファイルに DSDL コードを追加できます。Agent Builder は、ソースファイル内において独自のコードを追加できる場所にコメント文を埋め込みます。
4. ターゲットアプリケーションをサポートするために、さらにコードをカスタマイズする必要がある場合は、既存のソースコードに RMAPI 関数を追加できます。

実際には、データサービスを作成する方法はいくつもあります。たとえば、Agent Builder によって生成されたコード内の特定の場所に独自のコードを追加する代わりに、生成されたメソッドの 1 つや生成された監視プログラムを DSDL や RMAPI 関数を使って最初から作成したプログラムで完全に置き換えることができます。しかし、使用方法に関わらず、ほとんどの場合は Agent Builder を使って開発作業を開始することをお勧めします。次に、その理由を示します。

- Agent Builder が生成するコードは本質的に汎用であり、多数のデータサービスでテストされています。
- Agent Builder は、RTR ファイル、make ファイル、リソースのパッケージなど、データサービス用のサポートファイルを作成します。データサービスのコードをまったく使用しない場合でも、このようなサポートファイルを使用することによってかなりの作業を省略できます。
- 生成されたコードは変更できます。

注 - RMAPI は C 言語用の関数セットとスクリプト用のコマンドセットを提供しますが、DSDL は C 言語用の関数インタフェースしか提供しません。DSDL は ksh コマンドを提供しないので、Agent Builder で ksh 出力を指定した場合、生成されるソースコードは RMAPI を呼び出します。

データサービス作成用開発環境の設定

データサービスの開発を始める前に、Sun Cluster 開発パッケージ (SUNWscdev) をインストールして、Sun Cluster のヘッダーファイルやライブラリファイルにアクセスできるようにする必要があります。このパッケージがすでにすべてのクラスタノード上にインストールされている場合でも、通常は、クラスタノード上にはない独立した開発マシン、すなわちクラスタノード以外の開発マシンで開発を行います。このような場合、pkgadd(1M) を使って、開発マシンに SUNWscdev パッケージをインストールする必要があります。

コードをコンパイルおよびリンクするとき、ヘッダーファイルとライブラリファイルを識別するオプションを設定する必要があります。(クラスタノード以外の) 開発マシンで開発が終了すると、完成したデータサービスをクラスタに転送して、実行および検証できます。

注 – 必ず開発バージョンの Solaris 8 以上を使用してください。

この節では、次の手順を使用します。

- Sun Cluster 開発パッケージ (SUNWscdev) をインストールして、適切なコンパイラオプションとリンカーオプションを設定します。
- データサービスをクラスタに転送します。

▼ 開発環境の設定方法

SUNWscdev パッケージをインストールして、コンパイラオプションとリンカーオプションをデータサービス開発用に設定する方法について説明します。

1. **CD-ROM** のあるディレクトリに移動します。

```
cd appropriate_CD-ROM_directory
```

2. **SUNWscdev** パッケージを現在のディレクトリにインストールします。

```
pkgadd -d . SUNWscdev
```

3. **makefile** に、データサービスのコードが使用する **include** ファイルとライブラリファイルを示すコンパイラオプションとリンカーオプションを指定します。
-I オプションは、Sun Cluster のヘッダーファイルを指定します。-L オプションは、開発システム上にあるコンパイル時ライブラリの検索パスを指定します。-R オプションはクラスタの実行時リンカーの検索パスを指定します。

```
# Makefile for sample data service
...

-I /usr/cluster/include

-L /usr/cluster/lib

-R /usr/cluster/lib
...
```

▼ データサービスをクラスタに転送する方法

開発マシン上でデータサービスの開発が完了したなら、クラスタに転送して検証する必要があります。この転送を行うときは、エラーが発生する可能性を減らすために、データサービスのコードと RTR ファイルと一緒にパッケージに保管して、その後、クラスタのすべてのノード上でパッケージをインストールすることを推奨します。

注 - データサービスをインストールするときは、pkgadd を使用するかどうかに関わらず、すべてのクラスタノード上にデータサービスをインストールする必要があります。Agent Builder は、RTR ファイルとデータサービスのコードを自動的にパッケージ化します。

リソースとリソースタイププロパティの設定

Sun Cluster は、データサービスの静的な構成を定義するためのリソースタイププロパティおよびリソースプロパティのセットを提供します。リソースタイププロパティでは、リソースのタイプ、そのバージョン、API のバージョンと同時に、各コールバックメソッドへのパスも指定できます。表 A-1 に、すべてのリソースタイププロパティのリストを示します。

リソースプロパティ (Failover_mode、Thorough_probe_interval など) やメソッドタイムアウトも、リソースの静的な構成を定義します。動的なリソースプロパティ (Resource_state や Status など) は、管理対象のリソースの動作状況を反映します。リソースプロパティについては、表 A-2 を参照してください。

リソースタイプおよびリソースプロパティは、データサービスの重要な要素であるリソースタイプ登録 (RTR) ファイルで宣言します。RTR ファイルは、クラスタ管理者が Sun Cluster でデータサービスを登録するとき、データサービスの初期構成を定義します。

Agent Builder が宣言するプロパティセットはどのようなデータサービスにとっても有用かつ必須です。したがって、独自のデータサービス用の RTR ファイルを生成するときは、Agent Builder を使用することを推奨します。たとえば、ある種のプロパティ (Resource_type など) が RTR ファイルで宣言されていない場合、データサービスの登録は失敗します。必須ではなくても、その他のプロパティも RTR ファイルで宣言されていなければ、システム管理者はそれらのプロパティを利用することはできません。いくつかのプロパティは RTR ファイルで宣言されていなくても使用することができますが、これは RGM がそのプロパティを定義し、そのデフォルト値を提供しているためです。このような複雑さを回避するためにも、Agent Builder を使用して、適切な RTR ファイルを生成するようにしてください。必要であれば、Agent Builder で生成した RTR ファイルを編集して、特定の値を変更することもできます。

以降では、Agent Builder で作成した RTR ファイルの例を示します。

リソースタイププロパティの宣言

クラスタ管理者は、RTR ファイルで宣言されているリソースタイププロパティを構成することはできません。このようなリソースタイププロパティは、リソースタイプの恒久的な構成の一部を形成します。

注 - `Installed_nodes` というリソースタイププロパティは、システム管理者が構成できません。事実、`Installed_nodes` はシステム管理者が構成できる唯一のリソースタイププロパティであり、RTR ファイルでは宣言できません。

次に、リソースタイプ宣言の構文を示します。

```
property_name = value;
```

注 - RGM はプロパティ名の大文字と小文字を区別します。Sun が提供する RTR ファイルのプロパティに対する命名規則では、名前の最初の文字が大文字で、残りが小文字です (メソッド名は例外)。メソッド名は、プロパティ属性と同様にすべて大文字です。

次に、サンプルのデータサービス (smpl) 用の RTR ファイルにおけるリソースタイプ宣言を示します。

```
# Sun Cluster Data Services Builder template version 1.0
# Registration information and resources for smpl
#
#NOTE: Keywords are case insensitive, i.e., you can use
#any capitalization style you prefer.
#
Resource_type = "smpl";
Vendor_id = SUNW;
RT_description = "Sample Service on Sun Cluster";

RT_version = "1.0";
API_version = 2;
Failover = TRUE;

Init_nodes = RG_PRIMARYES;

RT_basedir=/opt/SUNWsmpl/bin;

Start          =    smpl_svc_start;
Stop           =    smpl_svc_stop;

Validate       =    smpl_validate;
Update        =    smpl_update;

Monitor_start  =    smpl_monitor_start;
Monitor_stop   =    smpl_monitor_stop;
```

```
Monitor_check =      smpl_monitor_check;
```

ヒント – RTR ファイルの最初のエントリには、Resource_type プロパティを宣言する必要があります。宣言しないと、リソースタイプの登録は失敗します。

リソースタイプ宣言の最初のセットは、次のようなリソースタイプについての基本的な情報を提供します。

Resource_type および Vendor_id

リソースタイプの名前を提供します。リソースタイプ名は Resource_type プロパティ (この例では「smpl」) 単独で指定できます。Vendor_id を接頭辞として使用し、リソースタイプ (この例では「SUNW.smpl」) との区切りにドット (.) を入力することもできます。Vendor_id を使用する場合、リソースタイプを定義する企業の略号にします。リソースタイプ名はクラスタ内で一意である必要があります。

注 – 便宜上、リソースタイプ名 (Resource_type と Vendor_id) はパッケージ名として使用されます。パッケージ名は 9 文字に制限されているので、これら 2 つのプロパティの文字数の合計も 9 文字以内に制限することをお勧めします。RGM は 9 文字の制限を適用しません。一方、Agent Builder はリソースタイプ名からパッケージ名を系統だてて生成します。つまり、Agent Builder は 9 文字の制限を適用します。

Rt_version

サンプルデータサービスのバージョンです。

API_version

API のバージョンです。たとえば、「API_version =2」は、データサービスが Sun Cluster バージョン 3.0 の管理下で動作していることを示します。

Failover = TRUE

このデータサービスが、複数のノード上で同時にオンラインにできるリソースグループ 上では実行できないサービス、すなわちフェイルオーバーデータサービスであることを示します。詳細については、48 ページの「フェイルオーバーリソースの実装」を参照してください。

Start、Stop、Validate など

RGM によって呼び出されるコールバックメソッドプログラムのパスを提供します。これらのパスは、RT_basedir で指定されたディレクトリからの相対パスです。

リソースタイプ宣言の残りのセットは、次のような構成情報を提供します。

Init_nodes = RG_PRIMARYES

RGM が、データサービスをマスターできるノード上でのみ Init、Boot、Fini、Validate の各メソッドを呼び出すように指定します。

RG_PRIMARYES で指定されたノードは、データサービスがインストールされているすべての

	ノードのサブセットです。この値に RT_INSTALLED_NODES を設定した場合、RGM は、データサービスがインストールされているすべてのノード上で上記メソッドを呼び出します。
RT_basedir	コールバックメソッドパスのような完全な相対パスとして、/opt/SUNWsample/bin をポイントします。
Start、Stop、Validate など	RGM によって呼び出されるコールバックメソッドプログラムのパスを提供します。これらのパスは、RT_basedir で指定されたディレクトリからの相対パスです。

リソースプロパティの宣言

リソースタイププロパティと同様に、リソースプロパティも RTR ファイルで宣言します。便宜上、リソースプロパティ宣言は RTR ファイルのリソースタイププロパティ宣言の後に行います。リソース宣言の構文では、一連の属性と値のペアを記述して、全体を中括弧で囲みます。

```
{
    Attribute = Value;
    Attribute = Value;
    .
    .
    Attribute = Value;
}
```

Sun Cluster が提供するリソースプロパティ (システム定義プロパティ) の場合、特定の属性は RTR ファイルで変更できます。たとえば、Sun Cluster はコールバックメソッドごとにメソッドタイムアウトプロパティを定義して、そのデフォルト値を提供します。RTR ファイルを使用すると、異なるデフォルト値を指定できます。

Sun Cluster が提供するプロパティ属性を使用することにより、RTR ファイル内に新しいリソースプロパティ (拡張プロパティ) を定義することもできます。表 A-4 に、リソースプロパティを変更および定義するための属性を示します。拡張プロパティ宣言は RTR ファイルのシステム定義プロパティ宣言の後に行います。

システム定義リソースプロパティの最初のセットでは、コールバックメソッドのタイムアウト値を指定します。

...

```
# Resource property declarations appear as a list of bracketed
# entries after the resource-type declarations. The property
# name declaration must be the first attribute after the open
```

```

# curly bracket of a resource property entry.
#
# Set minimum and default for method timeouts.
{
    PROPERTY = Start_timeout;
    MIN=60;
    DEFAULT=300;
}

{
    PROPERTY = Stop_timeout;
    MIN=60;
    DEFAULT=300;
}

{
    PROPERTY = Validate_timeout;
    MIN=60;
    DEFAULT=300;
}

{
    PROPERTY = Update_timeout;
    MIN=60;
    DEFAULT=300;
}

{
    PROPERTY = Monitor_Start_timeout;
    MIN=60;
    DEFAULT=300;
}

{
    PROPERTY = Monitor_Stop_timeout;
    MIN=60;
    DEFAULT=300;
}

{
    PROPERTY = Monitor_Check_timeout;
    MIN=60;
    DEFAULT=300;
}

```

プロパティ名 (PROPERTY = *value*) は、各リソースプロパティ宣言における最初の属性でなければなりません。リソースプロパティは、RTR ファイルのプロパティ属性で定義された範囲内で構成することができます。たとえば、各メソッドタイムアウト用のデフォルト値は 300 秒です。システム管理者はこの値を変更できますが、指定できる最小値は、MIN 属性で指定されているように 60 秒です。リソースプロパティ属性の完全なリストについては、表 A-4 を参照してください。

リソースプロパティの次のセットは、データサービスにおいて特定の目的に使用されるプロパティを定義します。

```

{
    PROPERTY = Failover_mode;
    DEFAULT=SOFT;
    TUNABLE = ANYTIME;
}

```

```

}
{
    PROPERTY = Thorough_Probe_Interval;
    MIN=1;
    MAX=3600;
    DEFAULT=60;
    TUNABLE = ANYTIME;
}

# The number of retries to be done within a certain period before concluding
# that the application cannot be successfully started on this node.
{
    PROPERTY = Retry_Count;
    MAX=10;
    DEFAULT=2;
    TUNABLE = ANYTIME;
}

# Set Retry_Interval as a multiple of 60 since it is converted from seconds
# to minutes, rounding up. For example, a value of 50 (seconds)
# is converted to 1 minute. Use this property to time the number of
# retries (Retry_Count).
{
    PROPERTY = Retry_Interval;
    MAX=3600;
    DEFAULT=300;
    TUNABLE = ANYTIME;
}

{
    PROPERTY = Network_resources_used;
    TUNABLE = WHEN_DISABLED;
    DEFAULT = "";
}

{
    PROPERTY = Scalable;
    DEFAULT = FALSE;
    TUNABLE = AT_CREATION;
}

{
    PROPERTY = Load_balancing_policy;
    DEFAULT = LB_WEIGHTED;
    TUNABLE = AT_CREATION;
}

{
    PROPERTY = Load_balancing_weights;
    DEFAULT = "";
    TUNABLE = ANYTIME;
}

{
    PROPERTY = Port_list;
    TUNABLE = AT_CREATION;
    DEFAULT = ;
}
}

```

上記のリソースプロパティ宣言では、システム管理者が値を設定し、制限を設けることができる TUNABLE 属性が追加されています。AT_CREATION は、システム管理者が値を指定できるのはリソースの作成時だけであり、後で変更できないことを示します。

上記のプロパティのほとんどは、特に理由がないかぎり、Agent Builder が生成するデフォルト値を使用しても問題ありません。こうしたプロパティのあとには、次のような情報が続きます。(詳細については、203 ページの「リソースプロパティ」または r_properties(5) のマニュアルページを参照してください。)

Failover_mode

Start または Stop メソッドの失敗時、RGM がリソースグループを再配置するか、ノードを停止するかを指定します。

Thorough_probe_interval, Retry_count, Retry_interval

障害モニターで使用します。障害モニターが適切に機能していない場合、システム管理者はいつでも調整できます。

Network_resources_used

データサービスで使用される論理ホスト名または共有アドレスリソースのリスト。このプロパティは、Agent Builder によって宣言されるので、システム管理者はデータサービスを構成するとき必要に応じてリソースのリストを指定できます。

Scalable

FALSE に設定した場合、このリソースはクラスタネットワークング (共有アドレス) 機能を使用しません。この設定は、リソースタイプ Failover プロパティに TRUE を設定して、フェイルオーバーサービスを指定するのと同じです。このプロパティの詳しい使用方法については、48 ページの「フェイルオーバーリソースの実装」および 48 ページの「スケーラブルリソースの実装」を参照してください。

Load_balancing_policy, Load_balancing_weights

これらのプロパティは自動的に宣言されますが、フェイルオーバーリソースタイプでは使用されません。

Port_list

サーバーが待機するポートのリストです。このプロパティは Agent Builder によって宣言されるので、システム管理者はデータサービスを構成するときポートのリストを指定できます。

拡張プロパティの宣言

次に、RTR ファイルの最後の例として、拡張プロパティを示します。

```
# Extension Properties
#
# The cluster administrator must set the value of this property to point to the
# directory that contains the configuration files used by the application.
# For this application, smpl, specify the path of the configuration file on
# PXFS (typically named.conf).
```

```

{
    PROPERTY = Confdir_list;
    EXTENSION;
    STRINGARRAY;
    TUNABLE = AT_CREATION;
    DESCRIPTION = "The Configuration Directory Path(s)";
}

# The following two properties control restart of the fault monitor.
{
    PROPERTY = Monitor_retry_count;
    EXTENSION;
    INT;
    DEFAULT = 4;
    TUNABLE = ANYTIME;
    DESCRIPTION = "Number of PMF restarts allowed for fault monitor.";
}
{
    PROPERTY = Monitor_retry_interval;
    EXTENSION;
    INT;
    DEFAULT = 2;
    TUNABLE = ANYTIME;
    DESCRIPTION = "Time window (minutes) for fault monitor restarts.";
}
# Time out value in seconds for the probe.
{
    PROPERTY = Probe_timeout;
    EXTENSION;
    INT;
    DEFAULT = 120;
    TUNABLE = ANYTIME;
    DESCRIPTION = "Time out value for the probe (seconds)";
}

# Child process monitoring level for PMF (-C option of pmfadm).
# Default of -1 means to not use the -C option of pmfadm.
# A value of 0 or greater indicates the desired level of child-process.
# monitoring.
{
    PROPERTY = Child_mon_level;
    EXTENSION;
    INT;
    DEFAULT = -1;
    TUNABLE = ANYTIME;
    DESCRIPTION = "Child monitoring level for PMF";
}
# User added code -- BEGIN VVVVVVVVVVVV
# User added code -- END   ^^^^^^^^^^^^^^^

```

次に示すように、Agent Builder は、ほとんどのデータサービスにとって有用な拡張プロパティを作成します。

Confdir_list

アプリケーション構成ディレクトリへのパスを指定します。このプロパティは多くのアプリケーションにとって有用な情報です。データサービスを構成するときに、システム管理者はこのディレクトリの場所を指定できます。

Monitor_retry_count、Monitor_retry_interval、Probe_timeout

サーバーデーモンではなく、障害モニター自体の再起動を制御します。

Child_mon_level

PMFによる監視レベルを設定します。詳細については、pmfadm(1M)のマニュアルページを参照してください。

「ユーザー追加コード」というコメント文で囲まれた部分に、追加の拡張プロパティを作成できます。

コールバックメソッドの実装

この節では、コールバックメソッドの実装に関する一般的な情報について説明します。

リソースとリソースグループのプロパティ情報へのアクセス

一般に、コールバックメソッドはリソースのプロパティにアクセスする必要があります。RMAPIは、リソースのシステム定義プロパティと拡張プロパティにアクセスするために、コールバックメソッドで使用できるシェルコマンドとC関数の両方を提供します。詳細については、scha_resource_get(1HA)とscha_resource_get(3HA)のマニュアルページを参照してください。

DSDLは、システム定義プロパティにアクセスするためのC関数セット(プロパティごとに1つ)と、拡張プロパティにアクセスするための関数を提供します。詳細については、scds_property_functions(3HA)とscds_get_ext_property(3HA)のマニュアルページを参照してください。

StatusとStatus_msgの設定を除き、リソースプロパティを設定するAPI関数が存在しないため、プロパティ機構を使用して、データサービスの動的な状態情報を格納することはできません。したがって、動的な状態情報は、広域ファイルに格納するようにします。

注 - クラスタ管理者は、`scrgadm` コマンド、グラフィカル管理コマンド、またはグラフィカル管理インタフェースを使って、ある種のリソースプロパティを設定することができます。ただし、`scrgadm` はクラスタの再構築時、すなわち RGM がメソッドを呼び出した時点でエラー終了するため、コールバックメソッドから呼び出さないようにします。

メソッドの呼び出し回数への非依存性

一般に、RGM は、同じリソース上で同じメソッドを (同じ引数で) 何回も連続して呼び出すことはありません。ただし、`Start` メソッドが失敗した場合には、リソースが起動していなくても、RGM はそのリソース上で `Stop` メソッドを呼び出すことができます。同様に、リソースデーモンが自発的に停止している場合でも、RGM はそのリソース上で `Stop` メソッドを呼び出すことができます。`Monitor_start` メソッドと `Monitor_stop` メソッドにも、同じことが当てはまります。

このような理由のため、`Stop` メソッドと `Monitor_stop` メソッドは呼び出し回数に依存しないように組み込む必要があります。同じリソース上で、同じパラメータを指定して `Stop` メソッドまたは `Monitor_stop` メソッドを何回連続して呼び出しても、1 回だけ呼び出したときと同じ結果になります。

呼び出し回数に依存しないということは、リソースまたはモニターがすでに停止し、動作していなくても、`Stop` メソッドと `Monitor_stop` メソッドが 0 (成功) を返す必要があるということも意味します。

注 - `Init`、`Finis`、`Boot`、`Update` の各メソッドも呼び出し回数に依存しない必要があります。`Start` メソッドは呼び出し回数に依存してもかまいません。

汎用データサービス

汎用データサービス (GDS) は、単純なアプリケーションを Sun Cluster の Resource Group Manager フレームワークに組み込むことにより、スケーラビリティと高可用性を実現する機構です。この機構では、アプリケーションを可用性の高いものにしたリ、スケーラブルなものにするために通常必要になるエージェントのコーディングは必要ありません。

GDS モデルは、コンパイル済みのリソースタイプ `SUNQ.gds` により、RGM フレームワークとやりとりします。

詳細については、第 10 章を参照してください。

アプリケーションの制御

RGM は、ノードがクラスタに結合されるとき、またはクラスタから切り離されるとき、コールバックメソッドを使って実際のリソース (アプリケーション) を制御できます。

リソースの起動と停止

リソースタイプを実装するには、少なくとも、Start メソッドと Stop メソッドが必要です。RGM は、リソースタイプのメソッドプログラムを適切なノード上で適切な回数だけ呼び出して、リソースグループをオフラインまたはオンラインにします。たとえば、クラスタノードのクラッシュ後、そのノードがマスターしているリソースグループを新しいノードに移動します。Start メソッドを実装して、RGM が正常に動作しているホストノード上で各リソースを再起動できるようにする必要があります。

Start メソッドは、ローカルノード上でリソースが起動し、使用可能な状態になるまで終了してはいけません。初期化に時間がかかるリソースタイプでは、その Start メソッドに、十分な長さのタイムアウト値を設定する必要があります。

Start_timeout プロパティのデフォルト値と最小値は、リソースタイプ登録ファイルで設定します。

Stop メソッドは、RGM がリソースをオフラインにする状況に合わせて実装する必要があります。たとえば、リソースグループがノード 1 上でオフラインになり、ノード 2 上でもう一度オンラインになると仮定します。リソースグループをオフラインにしている間、RGM は、そのリソースグループ内のリソース上で Stop メソッドを呼び出して、ノード 1 上のすべての活動を停止しようとします。ノード 1 上ですべてのリソースの Stop メソッドが完了したら、RGM は、ノード 2 上でそのリソースグループを再度オンラインにします。

Stop メソッドは、ローカルノード上でリソースがすべての活動を完全に停止し完全にシャットダウンするまで終了してはいけません。Stop メソッドは、ローカルノード上でリソース関連のすべてのプロセスを終了することで最も安全に実装できます。シャットダウンに時間がかかるリソースタイプでは、その Stop メソッドに十分な長さのタイムアウト値を設定する必要があります。Stop_timeout プロパティはリソースタイプ登録ファイルで設定します。

Stop メソッドが失敗またはタイムアウトすると、リソースグループはエラー状態になり、システム管理者の介入が必要となります。この状態を回避するには、Stop および Monitor_stop メソッドがすべてのエラー状態から回復するようにする必要があります。理想的には、これらのメソッドは 0 (成功) のエラー状態で終了し、ローカルノード上でリソースとそのモニターのすべての活動を正常に停止するべきです。

Start および Stop メソッドを使用するかどうかの決定

この節では、Start メソッドと Stop メソッドを使用するか、または、Prenet_start メソッドと Postnet_stop メソッドを使用するかを決定するときのいくつかの注意事項について説明します。どちらのメソッドが適切かを決定するには、クライアントおよびデータサービスのクライアントサーバー型ネットワークプロトコルについて十分に理解している必要があります。

ネットワークアドレスリソースを使用するサービスでは、論理ホスト名のアドレス構成から始まる順番で、起動手順または停止手順を実行する必要があります。コールバックメソッドの Prenet_start と Postnet_stop を使用してリソースタイプを実装すると、同じリソースグループ内のネットワークアドレスが「起動」に構成される前、または「停止」に構成された後に、特別な起動アクションまたは停止アクションを行います。

RGM は、データサービスの Prenet_start メソッドを呼び出す前に、ネットワークアドレスを取り付ける (plumb、ただし起動には構成しない) メソッドを呼び出します。RGM は、データサービスの Postnet_stop メソッドを呼び出したあとに、ネットワークアドレスを取り外す (unplumb) メソッドを呼び出します。RGM がリソースグループをオンラインにするときは、次のような順番になります。

1. ネットワークアドレスを取り付けます。
2. データサービスの Prenet_start メソッドを呼び出します (存在する場合)。
3. ネットワークアドレスを「起動」に構成します。
4. データサービスの Start メソッドを呼び出します (存在する場合)。

RGM がリソースグループをオフラインにするときは、逆の順番になります。

1. データサービスの Stop メソッドを呼び出します (存在する場合)。
2. ネットワークアドレスを「停止」に構成します。
3. データサービスの Postnet_stop メソッドを呼び出します (存在する場合)。
4. ネットワークアドレスを取り外します。

Start、Stop、Prenet_start、Postnet_stop のうち、どのメソッドを使用するかを決定するには、まずサーバー側を考えます。データサービスアプリケーションリソースとネットワークアドレスリソースの両方を持つリソースグループをオンラインにするとき、RGM は、データサービスリソースの Start メソッドを呼び出す前に、ネットワークアドレスを「起動」に構成するメソッドを呼び出します。したがって、データサービスを起動するときにネットワークアドレスが「起動」に構成されている必要がある場合は、Start メソッドを使用してデータサービスを起動します。

同様に、データサービスアプリケーションリソースとネットワークアドレスリソースの両方を持つリソースグループをオフラインにするとき、RGM は、データサービスリソースの Stop メソッドを呼び出したあとに、ネットワークアドレスを「停止」に構成するメソッドを呼び出します。したがって、データサービスを停止するときにネットワークアドレスが「起動」に構成されている必要がある場合は、Stop メソッドを使用してデータサービスを停止します。

たとえば、データサービスを起動または停止するとき、データサービスの管理ユーティリティまたはライブラリを呼び出す必要がある場合もあります。また、クライアントサーバー型ネットワークインタフェースを使用して管理を実行するような管理ユーティリティまたはライブラリを持っているデータサービスもあります。つまり、管理ユーティリティがサーバーデーモンを呼び出すので、管理ユーティリティまたはライブラリを使用するためには、ネットワークアドレスが「起動」に構成されている必要があります。このような場合は、Start メソッドと Stop メソッドを使用します。

データサービスが起動および停止するときネットワークアドレスが「停止」に構成されている必要がある場合は、Prenet_start メソッドと Postnet_stop メソッドを使用してデータサービスを起動および停止します。クラスタ再構成、scha_control ギブオーバー、または scswitch スイッチオーバーのあと、ネットワークアドレスとデータサービスのどちらが最初にオンラインになるかどうかによって、クライアントソフトウェアの応答が異なるかどうかを考えます。たとえば、クライアントの実装が最小限の再試行を行うだけで、データサービスのポートが利用できないと判断すると、すぐにあきらめる場合もあります。

データサービスを起動するときネットワークアドレスが「起動」に構成されている必要がない場合、ネットワークインタフェースが「起動」に構成される前に、データサービスを起動します。すると、ネットワークアドレスが「起動」に構成されるとすぐに、データサービスはクライアントの要求に応答できます。したがって、クライアントが再試行を停止する可能性も減ります。このような場合は、Start ではなく、Prenet_start メソッドを使用してデータサービスを起動します。

Postnet_stop メソッドを使用した場合、ネットワークアドレスが「停止」に構成されている時点では、データサービスリソースは「起動」のままです。Postnet_stop メソッドを呼び出すのは、ネットワークアドレスが「停止」に構成されたあとだけです。結果として、データサービスの TCP または UDP のサービスポート (つまり、その RPC プログラム番号) は、常に、ネットワーク上のクライアントから利用できません。ただし、ネットワークアドレスが応答しない場合を除きます。

Start メソッドと Stop メソッドを使用するか、Prenet_start メソッドと Postnet_stop メソッドを使用するか、または両方を使用するかを決定するには、サーバーとクライアントの要件と動作を考慮に入れる必要があります。

リソースの初期化と終了

RGM は、3つの任意のメソッド Init、Fini、Boot を使用し、リソース上で初期化と終了コードを実行できます。リソースを管理下に置くとき (リソースが属しているリソースグループを管理していない状態から管理している状態に切り替えるとき、またはすでに管理されているリソースグループでリソースを作成するとき)、RGM は Init メソッドを呼び出して、1回だけリソースの初期化を実行します。

リソースを管理下から外すとき (リソースが属しているリソースグループを管理していない状態に切り替えるとき、またはすでに管理されているリソースグループからリソースを削除するとき)、RGM は `Fini` を呼び出して、リソースをクリーンアップします。クリーンアップは呼び出し回数に依存しない必要があります。つまり、すでにクリーンアップが行われている場合、`Fini` は 0 (成功) で終了する必要があります。

RGM は、新たにクラスタに結合したノード、すなわち起動または再起動したノード上で、`Boot` メソッドを呼び出します。

`Boot` メソッドは、通常、`Init` と同じ初期化を実行します。この初期化は呼び出し回数に依存しない必要があります。つまり、ローカルノード上ですでにリソースが初期化されている場合、`Boot` と `Init` は 0 (成功) で終了する必要があります。

リソースの監視

通常、モニターは、リソース上で定期的に障害検証を実行し、検証したリソースが正しく動作しているかどうかを検出するように実装します。障害検証が失敗した場合、モニターは、ローカルで再起動するか、RMAPI 関数 `scha_control` または DSDL 関数 `scds_fm_action` を呼び出して、影響を受けるリソースグループのフェイルオーバーを要求できます。

また、リソースの性能を監視して、性能を調節または報告できます。可能であれば、リソースタイプに固有な障害モニターを作成することを推奨します。このような障害モニターを作成しなくても、リソースタイプは `Sun Cluster` により基本的なクラスタの監視が行われます。`Sun Cluster` は、ホストハードウェアの障害、ホストのオペレーティングシステムの全体的な障害、およびパブリックネットワーク上で通信できるホストの障害を検出します。

RGM がリソースモニターを直接呼び出すことはありませんが、RGM は自動的にリソース用のモニターを起動する準備を整えます。リソースをオフラインにするとき、RGM は、リソース自体を停止する前に、`Monitor_stop` メソッドを呼び出して、ローカルノード上でリソースのモニターを停止します。リソースをオンラインにするとき、RGM は、リソース自体を起動した後に、`Monitor_start` メソッドを呼び出します。

RMAPI 関数 `scha_control` と、`scds_fm_action` を呼び出す DSDL 関数 `scha_control` を使用すると、リソースモニターは異なるノードへのリソースグループのフェイルオーバーを要求できます。`Monitor_check` が定義されている場合、`scha_control` は妥当性検査の一環としてこの関数を呼び出して、リソースが属するリソースグループをマスターするのに要求されたノードが十分信頼できるかどうかを判断します。`Monitor_check` が「このノードは信頼できない」と報告した場合、あるいはメソッドがタイムアウトした場合、RGM はフェイルオーバー要求に適する別のノードを探します。すべてのノードで `Monitor_check` が失敗した場合、フェイルオーバーは取り消されます。

リソースモニターは、モニターから見たリソースの状態を反映するように `Status` と `Status_msg` プロパティを設定します。これらのプロパティを設定するには、RMAPI の `scha_resource_setstatus(1HA)` コマンドまたは同 (3HA) 関数、あるいは DSDL の `scds_fm_action(3HA)` 関数を使用します。

注 - `Status` と `Status_msg` はリソースモニターに固有の使用方法ですが、これらのプロパティは任意のプログラムで設定できます。

RMAPI による障害モニターの実装例については、100 ページの「障害モニターの定義」を参照してください。DSDL による障害モニターの実装例については、147 ページの「SUNW.xfnts 障害モニター」を参照してください。Sun が提供するデータサービスに組み込まれている障害モニターについては、『*Sun Cluster 3.1 データサービスのインストールと構成*』を参照してください。

メッセージログのリソースへの追加

状態メッセージを他のクラスタメッセージと同じログファイルに記録する場合は、`scha_cluster_getlogfacility` 関数を使用して、クラスタメッセージを記録するために使用されている機能番号を取得します。

この機能番号を通常の Solaris `syslog` 関数で使用して、状態メッセージをクラスタログに書き込みます。`scha_cluster_get` 汎用インタフェースからでも、クラスタログ機能情報にアクセスできます。

プロセス管理の提供

リソースモニターとリソース制御コールバックを実装するために、プロセス管理機能が Resource Management API および DSDL に提供されています。RMAPI は次の機能を定義します。これらのコマンドとプログラムの詳細については、各マニュアルページを参照してください。

プロセス監視機能 `pmfadm` および `rpc.pmf`

プロセス監視機能 (Process Monitor Facility: PMF) は、プロセスとその子孫を監視し、これらが終了したときに再起動する手段を提供します。この機能は、監視するプロセスを起動および制御する `pmfadm` コマンドと、`rpc.pmf` デーモンからなります。

`halockrun`

ファイルロックを保持しながら子プログラムを実行するためのプログラムです。このコマンドはシェルスクリプトで使用すると便利です。

hatimerun

タイムアウト制御下で子プログラムを実行するためのプログラムです。このコマンドはシェルスクリプトで使用すると便利です。

DSDL は、hatimerun 機能を実装するために `scds_hatimerun` 関数を提供します。

DSDL は、PMF 機能を実装するための関数セット (`scds_pmf_*`) を提供します。DSDL の PMF 機能の概要と、個々の関数のリストについては、196 ページの「PMF 関数」を参照してください。

リソースへの管理サポートの提供

リソース上での管理アクションには、リソースプロパティの設定と変更があります。このような管理アクションを行うために、API は `Validate` と `Update` というコールバックメソッドを定義しています。

リソースの作成時や、管理アクションによるリソースまたはリソースグループのプロパティの更新時、RGM は、オプションの `Validate` メソッドを呼び出します。RGM は、リソースとそのリソースグループのプロパティ値を `Validate` メソッドに渡します。RGM は、リソースタイプの `Init_nodes` プロパティが示す複数のクラスタノード上で `Validate` を呼び出します。 `Init_nodes` の詳細については、199 ページの「リソースタイププロパティ」か、`rt_properties(5)` のマニュアルページを参照してください。RGM は、作成または更新が行われる前に `Validate` を呼び出します。任意のノード上でメソッドから失敗の終了コードが戻ってくると、作成または更新は取り消されます。

RGM が `Validate` メソッドを呼び出すのは、管理アクションがリソースまたはグループのプロパティを変更したときだけです。RGM がプロパティを設定したときや、モニターがリソースプロパティ `Status` と `Status_msg` を設定したときではありません。

RGM は、オプションの `Update` メソッドを呼び出して、プロパティが変更されたことを実行中のリソースに通知します。RGM は、管理アクションがリソースまたはそのリソースグループのプロパティの設定に成功したあとに、`Update` を呼び出します。RGM は、リソースがオンラインであるノード上で、このメソッドを呼び出します。このメソッドは、API アクセス関数を使用して、アクティブなリソースに影響する可能性があるプロパティ値を読み取り、その値に従って、実行中のリソースを調節できます。

フェイルオーバーリソースの実装

フェイルオーバーリソースグループには、ネットワークアドレス (組み込みリソースタイプである論理ホスト名や共有アドレスなど) やフェイルオーバーリソース (フェイルオーバーデータサービス用のデータサービスアプリケーションリソースなど) があります。データサービスがフェイルオーバーするかスイッチオーバーされると、ネットワークアドレスリソースは関連するデータサービスリソースと共にクラスタノード間を移動します。RGM は、フェイルオーバーリソースの実装をサポートするプロパティをいくつか提供します。

ブール型リソースタイププロパティ `Failover` を `TRUE` に設定し、同時に複数のノード上でオンラインになることができるリソースグループだけで構成されるようにリソースを制限します。このプロパティのデフォルト値は `FALSE` です。したがって、フェイルオーバーリソースを実現するためには、RTR ファイルで `TRUE` として宣言する必要があります。

`Scalable` リソースプロパティは、リソースがクラスタ共有アドレス機能を使用するかどうかを決定します。フェイルオーバーリソースの場合、フェイルオーバーリソースは共有アドレスを使用しないので、`Scalable` には `FALSE` を設定します。

`RG_mode` リソースグループプロパティを使用すると、クラスタ管理者はリソースグループがフェイルオーバーまたはスケラブルのどちらであるかを識別できます。`RG_mode` が `FAILOVER` の場合、RGM はリソースグループの `Maximum primaries` プロパティを 1 に設定して、リソースグループが単一のノードでマスターされるように制限します。RGM は、`Failover` プロパティが `TRUE` であるリソースを、`RG_mode` が `SCALABLE` であるリソースグループで作成することを禁止します。

`Implicit_network_dependencies` リソースグループプロパティは、リソースグループ内におけるネットワークアドレスリソース (論理ホスト名や共有アドレス) への非ネットワークアドレスリソースの暗黙で強力な依存関係を、RGM が強制することを指定します。これは、リソースグループ内のネットワークアドレスが「起動」に構成されるまで、リソースグループ内の非ネットワークアドレス (データサービス) リソースが、自分の `Start` メソッドを呼び出さないことを意味します。この `Implicit_network_dependencies` プロパティのデフォルト値は `TRUE` です。

スケラブルリソースの実装

スケラブルリソースは、同時に複数のノード上でオンラインになることができます。スケラブルリソースには、Sun Cluster HA for Sun One Web Server や HA-Apache などのデータサービスがあります。

RGM は、スケーラブルリソースの実装をサポートするプロパティをいくつか提供します。

ブール型リソースタイププロパティの `Failover` を `FALSE` に設定し、一度に複数のノードでオンラインにできるリソースグループ内でリソースが構成されるようにします。

`Scalable` リソースプロパティは、リソースがクラスタ共有アドレス機能を使用するかどうかを決定します。スケーラブルサービスは共有アドレスリソースを使用するので (スケーラブルサービスの複数のインスタンスが単一のサービスであるかのようにクライアントに見せるため)、`Scalable` には `TRUE` を設定します。

`RG_mode` プロパティを使用すると、クラスタ管理者はリソースグループがフェイルオーバーまたはスケーラブルのどちらであるかを識別できます。`RG_mode` が `SCALABLE` の場合、RGM は `Maximum primaries` が 1 より大きな値を持つこと、つまり同時に複数のノードがグループをマスターすることを許可します。RGM は、`Failover` プロパティが `FALSE` であるリソースが、`RG_mode` が `SCALABLE` であるリソースグループ内でインスタンス化されることを許可します。

クラスタ管理者は、スケーラブルサービスリソースが属するためのスケーラブルリソースグループを作成します。また、スケーラブルリソースが依存する共有アドレスリソースが属するためのフェイルオーバーリソースグループも別に作成します。

クラスタ管理者は、`RG_dependencies` リソースグループプロパティを使用して、あるノード上でリソースグループをオンラインまたはオフラインにする順番を指定します。スケーラブルリソースとそれらが依存する共有アドレスリソースは異なるリソースグループに属するので、この順番はスケーラブルサービスにとって重要です。スケーラブルデータサービスが起動する前に、そのネットワークアドレス (共有アドレス) リソースが構成されていることが必要です。したがって、クラスタ管理者は、スケーラブルサービスが属するリソースグループの `RG_dependencies` プロパティを設定して、共有アドレスリソースが属するリソースグループを組み込む必要があります。

リソースの `RTR` ファイルでスケーラブルプロパティを宣言した場合、RGM はそのリソースに対して、次のようなスケーラブルプロパティのセットを自動的に作成します。

<code>Network_resources_used</code>	このリソースによって使用される共有アドレスリソースです。このプロパティのデフォルト値は空の文字列です。したがって、クラスタ管理者は、リソースを作成するときに、スケーラブルサービスが使用する実際の共有アドレスのリストを指定する必要があります。 <code>scsetup</code> コマンドと <code>SunPlex Manager</code> は、スケーラブルサービスに必要なリソースとグループを自動的に設定する機能を提供します。
<code>Load_balancing_policy</code>	リソースの負荷均衡ポリシーを指定します。このポリシーは <code>RTR</code> ファイルに明示的に設定しても、デフォルトの <code>LB_WEIGHTED</code> を使用してもかまいません。

せん。どちらの場合でも、クラスタ管理者はリソースを作成するときに値を変更できます (RTR ファイルで `Load_balancing_policy` を `NONE` または `FALSE` に設定していない場合)。有効な値は次のとおりです。

<code>LB_WEIGHTED</code>	<code>Load_balancing_weights</code> プロパティで設定されているウエイトに従って、さまざまなノードに負荷が分散されます。
<code>LB_STICKY</code>	スケーラブルサービスの指定のクライアント (クライアントの IP アドレスで識別される) は、常に同じクラスタノードに送信されます。
<code>LB_STICKY_WILD</code>	指定のクライアント (クライアントの IP アドレスで識別される) はワイルドカードスティッキーサービスの IP アドレスに接続され、送信時に使用されるポート番号とは無関係に、常に同じクラスタノードに送信されます。

`Load_balancing_policy`、`LB_STICKY`、`LB_STICKY_WILD` を持つスケーラブルなサービスの場合、サービスがオンラインの状態では `Load_balancing_weights` を変更すると、既存のクライアントとの関連がリセットされることがあります。リセットされると、(同じクラスタ内にある) 今までサービスを行っていたノードとは別のノードが、後続のクライアント要求を処理します。

同様に、サービスの新しいインスタンスをクラスタ上で開始すると、既存のクライアントとの関連がリセットされることがあります。

`Load_balancing_weights` 個々のノードへ負荷を送信することを指定します。形式は `weight@node,weight@node` です。 `weight` は、 `node` に分散される負荷の相対的な割り当てを示す整数です。ノードに分散される負荷の割合は、このノードのウエイトをアクティブなインスタンスのすべてのウエイトの合計で割った値になります。たとえば、 `1@1,3@2` は、ノード 1 に負荷の 1/4 が割り当てられ、ノード 2 に負荷の 3/4 が割り当てられることを意味します。

Port_list

サーバーが待機するポートです。このプロパティのデフォルト値は空の文字列です。ポートのリストは RTR ファイルに指定できます。このファイルで指定しない場合、クラスタ管理者は、リソースを作成するときに、実際のポートのリストを提供する必要があります。

データサービスは、管理者がスケーラブルまたはフェイルオーバーのどちらにでも構成できるように作成できます。このためには、データサービスの RTR ファイルにおいて、Failover リソースタイププロパティと Scalable リソースプロパティの両方を FALSE に宣言します。Scalable プロパティは作成時に調整できるように指定します。

Failover プロパティが FALSE の場合、リソースはスケーラブルリソースグループに構成できます。管理者はリソースを作成するときに Scalable を TRUE に変更する、すなわちスケーラブルサービスを作成することによって、共有アドレスを有効にできます。

一方、Failover が FALSE の場合でも、管理者はリソースをフェイルオーバーリソースグループに構成して、フェイルオーバーサービスを実装できます。この場合、Scalable の値 (FALSE) は変更しません。このような偶然性に対処するために、Scalable プロパティの Validate メソッドで妥当性を検査する必要があります。Scalable が FALSE の場合、リソースがフェイルオーバーリソースグループに構成されていることを確認します。

スケーラブルリソースの詳細については、『Sun Cluster 3.1 Concepts』を参照してください。

スケーラブルサービスの妥当性検査

Scalable プロパティが TRUE であるリソースが作成または更新されるたびに、RGM は、さまざまなリソースプロパティの妥当性を検査します。プロパティが正しく構成されていない場合、RGM は作成または更新を拒否します。RGM は次の検査を行います。

- Network_resources_used プロパティは、空の文字列であってはならず、既存の共有アドレスリソースの名前を含む必要があります。スケーラブルリソースを含むリソースグループの Nodelist にあるすべてのノードは、指定した共有アドレスリソースの 1 つである NetIfList プロパティまたは AuxNodeList プロパティに存在する必要があります。
- スケーラブルリソースを含むリソースグループの RG_dependencies プロパティは、スケーラブルリソースの Network_resources_used プロパティに存在する、すべての共有アドレスリソースのリソースグループを含む必要があります。
- Port_list プロパティは、空の文字列であってはならず、ポートとプロトコル (tcp または udp) のペアのリストを含む必要があります。次に例を示します。

データサービスの作成と検証

この節では、データサービスを作成および検証する方法について説明します。

キープアライブの使用方法

サーバー側で TCP キープアライブを有効にしておく、サーバーはダウン時の (または、ネットワークで分割された) クライアントのリソースを浪費しません。長時間稼働するようなサーバーでこのようなリソースがクリーンアップされない場合、浪費されたリソースが無制限に大きくなり、最終的にはクライアントに障害が発生して再起動します。

クライアントサーバー通信が TCP ストリームを使用する場合、クライアントとサーバーは両方とも TCP キープアライブ機構を有効にしなければなりません。これは、非高可用性の単一サーバーの場合でも適用されます。

ほかにも、キープアライブ機構を持っている接続指向のプロトコルは存在します。

クライアント側で TCP キープアライブを有効にしておく、ある物理ホストから別の物理ホストに論理ホストがフェイルオーバーまたはスイッチオーバーしたとき、接続の切断がクライアントに通知されます。このようなネットワークアドレスリソースの転送 (フェイルオーバーやスイッチオーバー) が発生すると、TCP 接続が切断されます。しかし、クライアント側で TCP キープアライブを有効にしておかなければ、接続が休止したとき、必ずしも接続の切断はクライアントに通知されません。

たとえば、クライアントが、実行に時間がかかる要求に対するサーバーからの応答を待っており、また、クライアントの要求メッセージがすでにサーバーに到着しており、TCP 層で認識されているものと想定します。この状況では、クライアントの TCP モジュールは要求を再転送し続ける必要はないので、クライアントアプリケーションはブロックされて、要求に対する応答を待ちます。

TCP キープアライブ機構は必ずしもあらゆる限界状況に対応できるわけではないので、クライアントアプリケーションは、可能であれば、TCP キープアライブ機構に加えて、独自の定期的なキープアライブをアプリケーションレベルで実行する必要があります。アプリケーションレベルのキープアライブ機構を使用するには、通常、クライアントサーバー型プロトコルが NULL 操作、または、少なくとも効率的な読み取り専用操作 (状態操作など) をサポートする必要があります。

HA データサービスの検証

この節では、高可用性環境における実装を検証する方法について説明します。この検証は一例であり、完全ではないことに注意してください。実際に稼働させるマシンに影響を与えないように、検証時は、検証用の Sun Cluster 構成にアクセスする必要があります。

リソースグループが物理ホスト間で移動するような場合を想定して、HA データサービスが適切に動作するかどうかを検証します。たとえば、システムがクラッシュした場合や、scswitch コマンドを使用した場合です。また、このような場合にクライアントマシンがサービスを受け続けられるかどうかを検証します。

メソッドの呼び出し回数への非依存性を検証します。たとえば、各メソッドを一時的に、元のメソッドを 2 回以上呼び出す短いシェルスクリプトに変更します。

リソース間の依存関係の調節

あるクライアントサーバーのデータサービスが、クライアントからの要求を満たすために、別のクライアントサーバーのデータサービスに要求を行うことがあります。このように、データサービス A が自分のサービスを提供するために、データサービス B にそのサービスを提供してもらう場合、データサービス A はデータサービス B に依存していると言います。この要件を満たすために、Sun Cluster では、リソースグループ内でリソースの依存関係を構築できます。依存関係は、Sun Cluster がデータサービスを起動および停止する順番に影響します。詳細については、scrgadm(1M) のマニュアルページを参照してください。

あるリソースタイプのリソースが別のリソースタイプのリソースに依存する場合、データサービス開発者は、リソースとリソースグループを適切に構成するようにユーザーに指示するか、これらを正しく構成するスクリプトまたはツールを提供する必要があります。依存するリソースを依存されるリソースと同じノード上で実行する必要がある場合、両方のリソースを同じリソースグループ内で構成する必要があります。

明示的なリソースの依存関係を使用するか、このような依存関係を省略して、HA データサービス独自のコードで別のデータサービスの可用性をポーリングするかを決定します。依存するリソースと依存されるリソースが異なるノード上で動作できる場合は、これらのリソースを異なるリソースグループ内で構成します。この場合、グループ間にはリソースの依存関係を構築できないため、ポーリングが必要です。

データサービスによっては、データを自分自身で直接格納せず、別のバックエンドデータサービスに依頼して、すべてのデータを格納してもらうものもあります。このようなデータサービスは、すべての読み取り要求と更新要求をバックエンドデータサービスへの呼び出しに変換します。たとえば、すべてのデータを SQL データベース (Oracle など) に格納するようなクライアントサーバー型のアポイントメントカレンダーサービスの場合、このサービスは独自のクライアントサーバー型ネットワークプロトコルを持っています。たとえば、RPC 仕様言語 (ONC™ RPC など) を使用するプロトコルを定義している場合があります。

Sun Cluster 環境では、HA-ORACLE を使用してバックエンド Oracle データベースを高可用性にできます。つまり、アポイントメントカレンダーデーモンを起動および停止する簡単なメソッドを作成できます。エンドユーザーは Sun Cluster でアポイントメントカレンダーのリソースタイプを登録できます。

アポイントメントカレンダーアプリケーションが Oracle データベースと同じノード上で動作する必要がある場合、エンドユーザーは、HA-ORACLE リソースと同じリソースグループ内でアポイントメントカレンダーリソースを構築して、アポイントメントカレンダーリソースを HA-ORACLE リソースに依存するようにします。この依存関係を指定するには、`scrgadm` の `Resource_dependencies` プロパティを使用します。

アポイントメントカレンダーリソースが HA-ORACLE リソースとは別のノード上で動作できる場合、エンドユーザーはこれらのリソースを2つの異なるリソースグループ内で構成します。カレンダーリソースグループのリソースグループ依存関係を、Oracle リソースグループ上で構築することもできます。しかし、リソースグループ依存関係が有効になるのは、両方のリソースグループが同時に同じノード上で起動または停止されたときだけです。したがって、カレンダーデータサービスデーモンは、起動後、Oracle データベースが利用可能になるまで、ポーリングして待機します。この場合、通常、カレンダーリソースタイプの `start` メソッドは単に成功を戻すだけです。これは、`start` メソッドが無限にブロックされると、そのリソースグループがビジー状態になり、それ以降、リソースグループで状態の変化 (編集、フェイルオーバー、スイッチオーバーなど) が行われなくなるためです。しかし、カレンダーリソースの `start` メソッドがタイムアウトまたは非ゼロで終了すると、Oracle データベースが利用できない間、リソースグループが複数のノード間でやりとりを無限に繰り返す可能性があります。

第 3 章

リソースタイプの更新

この章では、リソースタイプの開発者がリソースタイプの更新や移行を行うために必要な情報を提供します。

- 55 ページの「概要」
- 56 ページの「リソースタイプ登録ファイル」
- 58 ページの「Type_version リソースプロパティ」
- 59 ページの「リソースを別のバージョンへ移行」
- 60 ページの「リソースタイプのアップグレード」
- 61 ページの「デフォルトのプロパティ値」
- 62 ページの「リソースタイプ開発者の文書」
- 62 ページの「リソースタイプ名とリソースタイプモニターの実装」
- 63 ページの「アプリケーションのアップグレード」
- 63 ページの「リソースタイプのアップグレード例」
- 67 ページの「パッケージの問題」

概要

システム管理者は、既存のリソースタイプの新しいバージョンをインストールおよび登録できなければなりません。これは、リソースを削除したり作成し直したりすることなく、複数のバージョンのリソースタイプを登録したり、既存のリソースを新しいバージョンのリソースタイプに移行したりする必要があるからです。リソース開発者は、リソースタイプのアップグレードや移行の要件を把握しておく必要があります。

アップグレードを念頭に置いたリソースタイプの開発を「アップグレード対応」と呼びます。

新しいバージョンのリソースタイプは、次の点で前のバージョンとは異なっている可能性があります。

- リソースタイププロパティの属性

- 標準プロパティ、拡張プロパティを含む宣言済みリソースプロパティ
- リソースプロパティの属性 (default、min、max、arraymin、arraymax、または Tunable 属性)
- 宣言済みメソッド
- メソッドやモニターの実装

リソースタイプ開発者は、既存のリソースを新しいバージョンへ移行するタイミングを次の Tunable 属性のオプションによって特定します。制約の小さいものから順に、次のようなオプションがあります。

- 任意の時点 (Anytime)
- リソースが監視されていないとき (When_unmonitored)
- リソースがオフラインのとき (When_offline)
- リソースが無効のとき (When_disabled)
- リソースグループが管理されていないとき (When_unmanaged)
- 作成時 (At_creation)

各オプションについては、58 ページの「Type_version リソースプロパティ」を参照してください。

注 - この章では、アップグレード手順の記述で常に `scrgadm` コマンドを使用します。これは、管理者が `scrgadm` コマンドしか使用できないということではありません。GUI、`scsetup` コマンドなども使用できます。

リソースタイプ登録ファイル

リソースタイプ名

リソースタイプ名は、RTR ファイルに指定されたプロパティ、`Vendor_id`、`Resource_type`、`RT_version` の3つで構成されます。`scrgadm` コマンドは、ピリオドとコロンを区切り文字として使用し、リソースタイプ名を作成します。

```
vendor_id.resource_type:rt_version
```

`Vendor_id` 接頭辞では、同一名でベンダーの異なる2つの登録ファイルを識別できません。`RT_version` では、同じベースリソースタイプの複数のバージョン (アップグレード) を識別できます。重複を防ぐため、`Vendor_id` には、リソース型の作成元の会社のストックシンボルを使用することをお勧めします。

`RT_version` 文字列には、空白文字、タブ、スラッシュ、バックスラッシュ、アスタリスク、疑問符、コンマ、セミコロン、角括弧は使用できません。

RT_Version プロパティは、Sun Cluster 3.0 まではオプションでしたが、Sun Cluster 3.1 以降では必須です。

完全修飾名は、次のコマンドで取得できます。

```
scha_resource_get -O Type -R <resourcename> -G <resourcegroupname>
```

Sun Cluster 3.1 以前に登録されたリソースタイプ名は次の形式をとります。

```
vendor_id.resource_type
```

ディレクティブ

アップグレード対応リソースタイプの RTR ファイルには、次の形式の `#$upgrade` ディレクティブが必要です。ほかにディレクティブがある場合は、このディレクティブの後ろに続きます。アップグレード用により多くの容量を必要とするファイルシステムの制約は変更できませんが (`changeable` とマークされなければならない)、このようなファイルシステムに `layout_constraint` を使用すれば、その `minimum_size` 値を変更できます。

```
#$upgrade_from <version> <tunability>
```

`upgrade_from` ディレクティブは、文字列 `#$upgrade_from`、`RT_Version`、リソースの `Tunable` 属性の制約で構成されます。アップグレードを行う前のリソースタイプにバージョンがない場合、`RT_Version` は、以下の例の最後の行のように空文字列として指定されます。

```
#$upgrade_from "1.1" when_offline
#$upgrade_from "1.2" when_offline
#$upgrade_from "1.3" when_offline
#$upgrade_from "2.0" when_unmonitored
#$upgrade_from "2.1" anytime
#$upgrade_from "" when_unmanaged
```

システム管理者がリソース `Type_version` を変更しようとする、RGM によってこれらの制約が課されます。現在のリソースタイプのバージョンがリストに表示されない場合、`Tunable` 属性は `When_unmanaged` になります。

これらのディレクティブは、RTR ファイル内のリソースタイププロパティ宣言とリソース宣言セクションの間に指定されます。 `rt_reg 4` を参照してください。

RTR ファイル内の RT_Version の変更

RTR ファイルの内容が変更されたときは、必ず RTR ファイル内の `RT_Version` 文字列を変更します。このプロパティの値は、どちらのリソースタイプが新しく、どちらが古いかをはっきりと示す必要があります。RTR ファイルに変更が加えられていなければ、`RT_Version` 文字列を変更する必要はありません。

以前のバージョンの Sun Cluster のリソースタイプ名

Sun Cluster 3.0 のリソースタイプ名には、バージョン接尾辞がありません。

```
vendor_id.resource_name
```

元々 Sun Cluster 3.0 で登録されたリソースタイプは、クラスタリングソフトウェアを Sun Cluster 3.1 にアップグレードしたあとも、この形式の名前を持ちます。RTR ファイルを Sun Cluster 3.1 ソフトウェアで登録した場合でも、RTR ファイル内に `#$upgrade` ディレクティブの指定がないリソースタイプは、バージョン接尾辞のない Sun Cluster 3.0 の形式の名前を付与されます。

Sun Cluster 3.0 では、`#$upgrade` ディレクティブや `#$upgrade_from` ディレクティブを使った RTR ファイルの登録は可能ですが、既存のリソースの新しいリソースタイプへの移行はサポートされません。

Type_version リソースプロパティ

標準リソースプロパティ `Type_version` は、リソースタイプの `RT_Version` プロパティを格納します。このプロパティは、RTR ファイル内には指定されません。システム管理者は、次のコマンドを使って、`Type_version` プロパティを編集します。:アップグレード用により多くの容量を必要とするファイルシステムの制約は変更できませんが (`changeable` とマークされなければならない)、このようなファイルシステムに `layout_constraint` を使用すれば、その `minimum_size` 値を変更できます。

```
scrgadm -c -j <resource> -y Type_version=<new version>
```

Tunable 属性は、次の項目によって決まります。

- 現在のリソースタイプのバージョン
- RTR ファイル内の `#$upgrade_from` ディレクティブ

`#$upgrade_from` ディレクティブの値は次のとおりです。

`Anytime`

リソースをいつアップグレードしてもよい場合、リソースを完全にオンラインにできます。

`When_unmonitored`

新しいリソースタイプのバージョンの `Update`、`Stop`、`Monitor_check` および `Postnet_stop` メソッドが古いリソースタイプのバージョンの起動メソッド (`Prenet_stop` および `Start`) と互換することがわかっている場合、新しいリソースタイプのバージョンの `Finis` メソッドが古いバージョンの `Init` メソッドと互換することがわかっている場合は、アップグレード前にリソース監視プログラムを停止するだけで済みます。

When_offline

新しいリソースタイプのバージョンの Update、Stop、Monitor_check または Postnet_stop メソッドと古いリソースタイプのバージョンの起動メソッド (Prenet_stop および Start) には互換性がないが、これらのメソッドと古いリソースタイプのバージョンの Init メソッドには互換性があることがわかっている場合、タイプのアップグレード時にリソースはオフラインになります。

When_disabled

When_offline と同様です。ただし、より厳しい条件で、リソースが無効化されている場合になります。

When_unmanaged

新しいリソースタイプのバージョンの Fini メソッドが古いバージョンの Init メソッドと互換しないことがわかっている場合、リソースのアップグレード前に既存のリソースグループを管理されていない状態にする必要があります。

At_creation

新しいリソースタイプのバージョンにアップグレードできないリソースの場合、新しいバージョンの新しいリソースしか作成できません。

Tunable 属性が At_creation の場合、リソースタイプ開発者は既存のリソースを新しいタイプに移行することを禁止できます。この場合、システム管理者は、リソースを削除して作成し直す必要があります。これは、リソースのバージョンが作成時にだけ設定されるという宣言と同じことです。

リソースを別のバージョンへ移行

システム管理者がリソースの Type_version プロパティを編集すると、既存のリソースは新しいリソースタイプを取得します。これは、その他のリソースプロパティの編集時と同じ規則に従います。ただし、一部の情報が現在のバージョンではなく新しいリソースタイプから取得されるという点を除きます。

- min、max、arraymin、arraymax、デフォルト、Tunable 属性など、すべてのプロパティのリソースプロパティ属性は、新しいリソースタイプのバージョンから取得されます。
- Type_version プロパティに適用される Tunable 属性は、RTR ファイル内の #Supgrade_from ディレクティブと、既存のリソースのリソースタイプの RT_version プロパティから取得されます。Tunable 属性は、property_attributes 5 のマニュアルページの説明とは異なります。
- 新しいリソースタイプのバージョンの Validate メソッドが適用されます。このため、プロパティ属性は、新しいリソースタイプで有効です。既存のリソースプロパティ属性が新しいリソースタイプのバージョンの妥当性検査の条件を満たしていない場合、システム管理者は、scrgadm コマンドに、こうしたプロパティの有効な値を指定する必要があります。この手続きは、新しいリソースタイプのバージョンが、以前のバージョンでは宣言されていなかったプロパティを使用し、デ

フォルト値がない場合に行われます。既存のリソースが、新しいリソースタイプのバージョンでは無効なプロパティ値を持っている場合にも、この手続きが行われま
す。

- 古いバージョンのリソースタイプで宣言されたリソースプロパティが新しいバージョンでは宣言されないことがあります。こうしたリソースを新しいバージョンに移行すると、リソースからプロパティが削除されます。

注 - Validate メソッドは、リソースの新しい Type_version (Validate コマンド行に渡される) だけでなく現在の Type_version も照会できます (scha_resource_get を使用)。このため、Validate、サポートされないバージョンのアップグレードを禁止することができます。

リソースタイプのアップグレード

リソースタイプのアップグレードおよび移行の詳細については、『Sun Cluster 3.1 データサービスのインストールと構成』の「リソースタイプのアップグレード」を参照してください。

▼ リソースタイプのアップグレード

1. 新しいリソースタイプのアップグレード手順の説明を読み、リソースタイプの変更とリソースの **Tunable** 属性の制約を確認します。
2. すべてのクラスタノードに、リソースタイプアップグレードパッケージをインストールします。
新しいリソースタイプパッケージのインストールは、段階的に行うことをお勧めします。ノードが非クラスタモードで起動している間に pkgadd を実行します。
クラスタモードのノードに新しいリソースタイプパッケージをインストールする場合は、条件に合わせて異なった方法を使用します。
 - リソースタイプパッケージのインストールによってメソッドコードが変更されず、モニターだけが更新される場合は、インストール中にそのタイプのすべてのリソース上で監視を停止する必要があります。
 - リソースタイプパッケージのインストールによってメソッドとモニターの両方のコードが変更されない場合は、ディスク上に新しい RTR ファイルが追加されるだけなので、インストール中にリソース上での監視を停止する必要はありません。
3. アップグレードの **RTR** ファイルを参照し、**scrgadm** または同等のコマンドを使って新しいリソースタイプのバージョンを登録します。
RGM により、次の形式の新しいリソースタイプが作成されます。

```
vendor_id.resource_type:version
```

4. リソースタイプのアップグレードがノードのサブセットにだけインストールされる場合、追加の手順が必要になります。新しいリソースタイプの **Installed_nodes** プロパティに実際のインストール先ノードを設定する必要があります。リソースが新しいタイプ (新しく作成された、または更新された) を取得するとき、RGM は、リソースグループ **nodelist** がリソースタイプの **Installed_nodes** リストのサブセットであることを要求します。

```
scrgadm -c -t <resource_type> -h <installed_node_list>
```

5. 以前にタイプがアップグレードされ、これからそのタイプに移行する各リソースに対して、**scswitch** を呼び出し、そのリソース自体またはそのリソースグループを、アップグレード手順に示されている適切な状態に変更します。
6. 以前にタイプがアップグレードされ、これからそのタイプに移行する各リソースを編集し、**Type_version** プロパティに新しいバージョンを設定します。

```
scrgadm -c -j <resource> -y Type_version=<new version>
```

必要に応じて、同じコマンドを使って、同じリソースのその他のプロパティに適切な値を設定します。

7. 手順 5 で呼び出したコマンドを逆方向に実行すると、リソースまたはリソースグループの以前の状態を回復できます。

デフォルトのプロパティ値

RGM は、システム管理者によって明示的に設定されなかったためリソースの CCR エントリに格納されていないデフォルトのプロパティなど、すべてのリソースを格納します。RGM は、CCR からリソースが読み込まれるとき、欠落したリソースプロパティのデフォルト値をそのリソースタイプから取得します。リソースタイプにデフォルト値が定義されていない場合は、システム定義のデフォルト値を使用します。アップグレードされたリソースタイプは、この方法により、新しいプロパティや既存のプロパティの新しいデフォルト値を定義することができます。

リソースプロパティが変更された場合、RGM は編集コマンドで指定されたプロパティを CCR に格納します。

リソースタイプのアップグレードされたバージョンが、デフォルトのプロパティの新しいデフォルト値を宣言する場合、新しいデフォルト値は既存のリソースから継承されます。この手続きは、プロパティが作成時または無効化されている場合だけ「Tunable」と宣言されている場合でも変わりません。新しいデフォルトのアプリケーションで、Stop、Postnet_stop、Finiなどのメソッドが失敗する場合、リソースタイプの実装者は、アップグレード時のリソースの状態を適切に制限する必要があります。具体的には、Type_version プロパティの Tunable 属性を制限します。

新しいリソースタイプのバージョンの `validate` メソッドでは、既存のプロパティ属性が適切であるかどうかを検査できます。既存のプロパティ属性が適切でない場合、システム管理者は、`Type_version` プロパティを編集するコマンドを使って、新しいリソースタイプのバージョンにリソースをアップグレードできるように、プロパティの属性値を変更できます。

注 – Sun Cluster 3.0 で作成されたリソースは、新しいバージョンに移行しても、リソースタイプからデフォルトのプロパティ属性を継承しません。これは、これらのリソースのデフォルトのプロパティ値が CCR に格納されているからです。

リソースタイプ開発者の文書

リソースタイプ開発者は、新しいリソースに関する文書を提供する必要があります。必須記載事項は次のとおりです。

- プロパティの追加、変更、削除の説明
- プロパティを新しい要件に準拠させた方法の説明
- リソースの `Tunable` 属性の制約
- 新しいデフォルトプロパティ属性
- `Type_version` プロパティの編集に使用するコマンドを使って、既存のリソースプロパティを適切な値に変更し、リソースを新しいリソースタイプのバージョンにアップグレードできることの通知 (システム管理者向けの情報)

リソースタイプ名とリソースタイプモニターの実装

Sun Cluster 3.0 でもアップグレード対応のリソースタイプを登録することはできませんが、これらの名前はバージョン接尾辞なしで CCR に記録されています。Sun Cluster 3.0 と 3.1 を両方とも正常に実行するには、このリソースタイプのモニターが両方の命名規則を処理できなければなりません。

```
vendor_id.resource_name:version  
vendor_id.resource_name
```

モニターコードは、次のコマンドと同等のコマンドを実行することにより、適切な名前を判断できます。

```
scha_resourcetype_get -O RT_VERSION -T VEND.myrt
scha_resourcetype_get -O RT_VERSION -T VEND.myrt:vers
```

次に、`vers` と出力値を比較します。同じリソースタイプと同じバージョンを別の名前で2回以上登録することはできません。したがって、`vers` の特定の値に対して成功するのは、上記のコマンドのいずれか一方のみとなります。

アプリケーションのアップグレード

アプリケーションコードのアップグレードは、いくつかの共通点はあるものの、エージェントコードのアップグレードとは大きく異なっています。アプリケーションのアップグレードでは、リソースタイプのアップグレードが必要な場合とそうでない場合があります。

リソースタイプのアップグレード例

以下では、リソースタイプのインストールとアップグレードの例をいくつか紹介します。Tunable 属性とパッケージ情報は、リソースタイプ実装の変更に基づいて選択されています。Tunable 属性は、リソースを新しいリソースタイプに移行するときに適用されます。

すべての例は、次の条件に従うものとします。

- リソースタイプは Solaris 形式のパッケージで配布されています。 `pkgadd 1M` および `pkgrm 1M` のマニュアルページを参照してください。
- リソースタイプの以前のバージョンは1つしかないので、新しい RTR ファイル内には `#$upgrade_from` ディレクティブが1つしかありません。
- RGM がディスクから削除されたメソッドを呼び出すことができる場合、インストールによってメソッドが削除または上書きされることはありません。
- 特に注記がない場合、新しいメソッドと古いメソッドには互換性があります。
- リソースおよびリソースグループは、インストールまたは移行前に適切な `scswitch 1M` コマンド (または同等のコマンド) により適切な状態に設定されます。次に示すのは、リソースグループを管理対象外に設定する例です。

```
scswitch -M -n -j <resource>
scswitch -n -j <resource>
scswitch -F -g <resource group>
scswitch -u -g <resource group>
```

- 登録は次のコマンド (または同等のコマンド) で行います。

```
scrgadm -a -t <resource type> -f <path to RTR file>
```

- 移行は次のコマンド (または同等のコマンド) で行います。

```
scrgadm -c -j <resource> -y Type_version=<version> \
-y <property=value> \
-x <property=value> ...
```

- リソースおよびリソースグループは、移行後に適切な `scswitch 1M` コマンド (または同等のコマンド) により以前の状態に戻されます。

```
scswitch -M -e -j <resource>
scswitch -e -j <resource>
scswitch -o -g <resource group>
scswitch -Z -g <resource group>
```

場合によっては、リソースタイプ開発者は、例で使用されているものより制限の厳しい Tunable 属性の値を指定する必要があります。Tunable 属性の値は、リソースタイプ実装の変更内容によって決定します。リソースタイプ開発者は、例で使用されている Solaris 形式のパッケージ以外のパッケージスキーマを選択することもできます。

表 3-1 リソースタイプのアップグレード例

変更の種類	Tunable 属性	パッケージ	手続き
プロパティの変更は、RTR ファイルのみに加えられました。	Anytime	新しい RTR ファイルを配布するだけです。	新しい RTR ファイルの <code>pkgadd</code> をすべてのノード上で実行します。 新しいリソースタイプを登録します。 リソースを移行します。
メソッドが更新されました。	Anytime	更新されたメソッドを古いメソッドとは異なったパスに配置します。	更新されたメソッドの <code>pkgadd</code> をすべてのノード上で実行します。 新しいリソースタイプを登録します。 リソースを移行します。
新しいモニタープログラムです。	When_unmonitored	以前のバージョンのモニターを上書きするだけです。	監視を無効化します。 新しい監視プログラムの <code>pkgadd</code> をすべてのノード上で実行します。 新しいリソースタイプを登録します。 リソースを移行します。 監視を有効化します。

表 3-1 リソースタイプのアップグレード例 (続き)

変更の種類	Tunable 属性	パッケージ	手続き
メソッドが更新されました。新しい Update/ Stop メソッドと古い Start メソッドには互換性がありません。	When_offline	更新されたメソッドを古いメソッドとは異なったパスに配置します。	更新されたメソッドの pkgadd をすべてのノード上で実行します。 新しいリソースタイプを登録します。 リソースをオフラインにします。 リソースを移行します。 リソースをオンラインにします。
メソッドが更新され、RTR ファイルに新しいプロパティが追加されました。新しいメソッドには新しいプロパティが必要です。リソースの所属リソースグループをオンラインのまま保持しながらリソースがオンラインになることを防ぐ必要があります。このためには、リソースグループをノード上でオフラインの状態からオンラインの状態に変更します。	When_disabled	以前のバージョンのメソッドを上書きします。	リソースを無効にします。 各ノード上で次の処理を行います。 ■ クラスタからノードを削除 ■ 更新されたメソッドの pkgrm/pkgadd の実行 ■ クラスタにノードを戻す 新しいリソースタイプを登録します。 リソースを移行します。 リソースを有効にします。

表 3-1 リソースタイプのアップグレード例 (続き)

変更の種類	Tunable 属性	パッケージ	手続き
メソッドが更新され、RTR ファイルに新しいプロパティが追加されました。新しいメソッドは新しいプロパティを必要としません。	Anytime	以前のバージョンのメソッドを上書きします。	<p>各ノード上で次の処理を行います。</p> <ul style="list-style-type: none"> ■ クラスタからノードを削除 ■ 更新されたメソッドの pkgrm/pkgadd の実行 ■ クラスタにノードを戻す <p>この手続きの間に、RGM は新しいメソッドを呼び出します。これは、まだ移行が完了しておらず、新しいプロパティが構成されていない場合でも変わりません。新しいメソッドは、新しいプロパティなしで正常に機能しなければなりません。</p> <p>新しいリソースタイプを登録します。</p> <p>リソースを移行します。</p>
メソッドが更新されました。新しい Fini メソッドと古い Init メソッドには互換性がありません。	When_unmanaged	更新されたメソッドを古いメソッドとは異なったパスに配置します。	<p>リソースの所属リソースグループを管理対象外にします。</p> <p>更新されたメソッドの pkgadd をすべてのノード上で実行します。</p> <p>リソースタイプを登録します。</p> <p>リソースを移行します。</p> <p>リソースの所属リソースグループを管理対象にします。</p>
メソッドが更新されました。RTR ファイルは変更されていません。	なし (RTR ファイルへの変更なし)	以前のバージョンのメソッドを上書きします。	<p>各ノード上で次の処理を行います。</p> <ul style="list-style-type: none"> ■ クラスタからノードを削除 ■ 更新されたメソッドの pkgadd を実行 ■ クラスタにノードを戻す <p>RTR ファイルが変更されなかったため、リソースを登録または移行する必要はありません。</p>

パッケージの問題

新しいリソースタイプをインストールするときは、次の2つの要件が満たされていなければなりません。

- リソースタイプが登録されている場合、ディスク上の RTR ファイルにアクセスできなければなりません。
- 新しいタイプのリソースを作成した場合、新しいタイプの宣言済みメソッドのパス名および監視プログラムがディスク上に存在し、実行できなければなりません。リソースが使用されている間は、以前のメソッドおよび監視プログラムを定位置に確保しておく必要があります。

最適のパッケージを決定するため、リソースタイプの実装者は、次のことを考慮する必要があります。

- RTR ファイルが変更されたか
- プロパティのデフォルト値または `tunable` 属性が変更されたか
- プロパティの `min` または `max` 値が変更されたか
- アップグレードによってプロパティが追加されたか、または削除されたか
- メソッドコードが変更されたか
- モニターコードが変更されたか
- 新しいメソッドまたはモニターコードが以前のバージョンのものと互換するか

RTR ファイルの変更

リソースタイプをアップグレードしても、新しいメソッドまたはモニターコードが呼び出されない場合があります。たとえば、リソースプロパティのデフォルト値または `tunable` 属性の値だけが変更される場合があります。メソッドコードは変更されないため、読み取り可能な RTR ファイルの有効なパス名を指定するだけでアップグレードをインストールできます。

古いリソースタイプを登録し直す必要がない場合、新しい RTR ファイルによって以前のバージョンのものが上書きされます。それ以外の場合、新しい RTR ファイルは新しいパス名で配置できます。

アップグレードによってプロパティのデフォルト値または `tunable` 属性が変更される場合、移行時に新しいバージョンの `validate` メソッドで既存のプロパティ属性が新しいリソースタイプでも有効かどうかを検査できます。アップグレードによってプロパティの `min`、`max` または `type` 属性が変更される場合、移行時に `scrgadm` コマンドでこれらの制約が自動的に検査されます。

アップグレード文書には、新しいデフォルトのプロパティ値をすべて記載する必要があります。この文書では、システム管理者に、`Type_version` コマンドを編集するコマンドを使って、新しいリソースタイプのバージョンにプロパティをアップグレードできるように、既存のリソースプロパティの値を変更できることを通知します。

アップグレードによってプロパティが追加または削除される場合、コールバックメソッドまたはモニターコードの一部を変更しなければならないことがあります。

モニターコードの変更

更新後のリソースタイプでモニターコードだけが変更される場合、パッケージのインストールによってモニターのバイナリが上書きされます。文書には、システム管理者向けの情報として、新しいパッケージをインストールする前に監視を一時停止する指示を記述します。

メソッドコードの変更

更新後のリソースタイプでメソッドコードだけが変更される場合、新しいメソッドコードが以前のバージョンのものと互換するかどうかを確認する必要があります。これにより、新しいメソッドコードを新しいパス名で格納するか、古いメソッドを上書きするかが決定します。

新しい Stop、Postnet_stop および Fini メソッドが宣言されていて、古いバージョンの Start、Prenet_stop または Init メソッドによって初期化または起動されたリソースに適用できる場合、新しいメソッドで古いメソッドを上書きできます。

新しいメソッドコードが以前のバージョンのものと互換しない場合、アップグレードされたリソースタイプに移行する前に、古いバージョンのメソッドを使ってリソースを停止または構成を解除する必要があります。新しいメソッドが古いメソッドを上書きする場合、リソースタイプをアップグレードする前に、そのタイプのすべてのリソースを停止 (場合によっては、さらに管理対象外に設定) しなければならないことがあります。新しいメソッドが古いものと別の場所に格納されていて、両方に同時にアクセスできる場合は、後方互換性がなくても、新しいリソースタイプのバージョンをインストールし、リソースを1つずつアップグレードすることができます。

新しいメソッドに後方互換性がある場合でも、その他のリソースが古いメソッドを使用し続けている間は、リソースを1つずつアップグレードして、新しいメソッドを使用できるようにする必要があります。また、新しいメソッドは、古いメソッドを上書きすることがないように別のディレクトリに格納する必要があります。

個々のリソースタイプのバージョンのメソッドを別々のディレクトリに格納すると、新しいバージョンで問題が発生したとき、元のリソースタイプのバージョンに切り替える手続きが簡単であるという点で有利です。

引き続きサポートされている以前のバージョンをすべてパッケージに含めるという方法もあります。この方法では、古いメソッドのパスを上書きまたは削除することなく、新しいパッケージのバージョンで古いバージョンを置き換えることができます。サポートされる以前のバージョンの数は、リソースタイプ開発者が決定します。

注 - メソッド、または現在クラスタ内にあるノード上の `pkgrm/ pkgadd` メソッドの上書きはお勧めしません。RGM がディスク上のアクセス不能なメソッドを呼び出そうとすると、予測できない結果を招くことがあります。実行中のメソッドのバイナリの削除または置き換えでも、予測できない結果を招く可能性があります。

第 4 章

Resource Management API リファレンス

この章では、Resource Management API (RMAPI) を構成するアクセス関数やコールバックメソッドに関する情報を提供します。ここでは、各関数やメソッドについて簡単に説明します。詳細については、Resource Management API の各々のマニュアルページを参照してください。

この章の内容は、次のとおりです。

- 72 ページの「RMAPI アクセスメソッド」 - シェルスクリプトコマンド (1HA) と C 関数 (3HA)
 - `scha_resource_get(1HA)`、`scha_resource_close(3HA)`、`scha_resource_get(3HA)`、`scha_resource_open(3HA)`
 - `scha_resource_setstatus(1HA)`、`scha_resource_setstatus(3HA)`
 - `scha_resourcetype_get(1HA)`、`scha_resourcetype_close(3HA)`、`scha_resourcetype_get(3HA)`、`scha_resourcetype_open(3HA)`
 - `scha_resourcegroup_get(1HA)`、`scha_resourcegroup_get(3HA)`、`scha_resourcegroup_close(3HA)`、`scha_resourcegroup_open(3HA)`
 - `scha_control(1HA)`、`scha_control(3HA)`
 - `scha_cluster_get(1HA)`、`scha_cluster_close(3HA)`、`scha_cluster_get(3HA)`、`scha_cluster_open(3HA)`
 - `scha_cluster_getlogfacility(3HA)`
 - `scha_cluster_getnodename(3HA)`
 - `scha_strerror(3HA)`
- 77 ページの「RMAPI コールバックメソッド」 - `rt_callbacks(1HA)` のマニュアルページで説明されている内容
 - Start
 - Stop
 - Init
 - Fini
 - Boot

- Prenet_start
- Postnet_stop
- Monitor_start
- Monitor_stop
- Monitor_check
- Update
- ValidateS

RMAPI アクセスメソッド

APIは、リソース、リソースタイプ、リソースグループのプロパティ、および他のクラスタ情報にアクセスするための関数を提供します。これらの関数はシェルコマンドとC関数の両方の形で提供されるため、リソースタイプの開発者はシェルスクリプトまたはCプログラムのどちらでも制御プログラムを実装できます。

RMAPI シェルコマンド

シェルコマンドは、クラスタのRGMによって制御されるサービスを表すリソースタイプのコールバックメソッドを、シェルスクリプトで実装するときに使用します。このコマンドを使用すると、次のことを行えます。

- リソース、リソースタイプ、リソースグループ、クラスタについての情報にアクセスする。
- モニターと併用し、リソースの `Status` プロパティと `Status_msg` プロパティを設定する。
- リソースグループの再起動と再配置を要求する。

注 - この節では、シェルコマンドについて簡単に説明します。詳細については、各コマンドの (1HA) マニュアルページを参照してください。特に注記しないかぎり、各コマンドと同じ名前のマニュアルページがあります。

RMAPI リソースコマンド

以下のコマンドを使用すると、リソースについての情報にアクセスしたり、リソースの `Status` プロパティや `Status_msg` プロパティを設定できます。

`scha_resource_get`

RGMの制御下のリソースまたはリソースタイプに関する情報にアクセスできます。このコマンドは、C関数 `scha_resource_get(3HA)` と同じ情報を提供します。

`scha_resource_setstatus`

RGM の制御下のリソースの `status` および `Status_msg` プロパティを設定します。このコマンドはリソースのモニターによって使用され、モニターから見たリソースの状態を反映します。このコマンドは、C 関数 `scha_resource_setstatus(3HA)` と同じ機能を提供します。

注 - `scha_resource_setstatus` はリソースモニター専用の関数ですが、任意のプログラムから呼び出すことができます。

リソースタイプコマンド

このコマンドは、RGM に登録されているリソースタイプについての情報にアクセスします。

`scha_resourcetype_get`

このコマンドは、C 関数 `scha_resourcegroup_get(3HA)` と同じ機能を提供します。

リソースグループコマンド

以下のコマンドを使用すると、リソースグループについての情報にアクセスしたり、リソースグループを再起動できます。

`scha_resourcegroup_get`

RGM の制御下のリソースグループに関する情報にアクセスできます。このコマンドは、C 関数 `scha_resourcegroup_get` と同じ機能を提供します。

`scha_control`

RGM の制御下のリソースグループの再起動、または別のノードへの再配置を要求します。このコマンドは、C 関数 `scha_control(3HA)` と同じ機能を提供します。

クラスタコマンド

このコマンドは、クラスタについての情報 (ノード名、ノード ID、ノードの状態、クラスタ名、リソースグループなど) にアクセスします。

`scha_cluster_get`

このコマンドは、C 関数 `scha_cluster_get(3HA)` と同じ情報を提供します。

C 関数

C 関数は、クラスタの RGM によって制御されるサービスを表すリソースタイプのコールバックメソッドを、C プログラムで実装するときに使用します。この関数を使用すると、次のことを行えます。

- リソース、リソースタイプ、リソースグループ、クラスタについての情報にアクセスする。
- モニターと併用し、リソースの `Status` プロパティと `Status_msg` プロパティを設定する。
- リソースグループの再起動と再配置を要求する。
- エラーコードを適切なエラーメッセージに変換する。

注 - この節では、C 関数について簡単に説明します。詳細については、各関数の (3HA) マニュアルページを参照してください。特に注記しないかぎり、各関数と同じ名前のマニュアルページがあります。C 関数の出力関数および戻りコードについては、`scha_calls` (3HA) のマニュアルページを参照してください。

リソース関数

以下の関数は、RGM に管理されているリソースについての情報にアクセスします。モニターから見たリソースの状態を表します。

`scha_resource_open(3HA)`、`scha_resource_get(3HA)`、
`scha_resource_close(3HA)`

これらの関数は、RGM によって管理されるリソースの情報にアクセスします。

`scha_resource_open` 関数は、リソースへのアクセスを初期化し、`scha_resource_get` のハンドルを戻します。`scha_resource_get` 関数は、リソースの情報にアクセスします。`scha_resource_close` 関数は、ハンドルを無効にし、`scha_resource_get` の戻り値に割り当てられているメモリーを解放します。

`scha_resource_open` 関数がリソースのハンドルを戻した後に、クラスタの再構成や管理アクションによって、リソースが変更されることがあります。この場合、`scha_resource_get` 関数がハンドルを通じて獲得した情報は正しくない可能性があります。リソース上でクラスタの再構成や管理アクションが行われた場合、RGM は `scha_err_seqid` エラーコードを `scha_resource_get` 関数に戻し、リソースが変更されたことを示します。これは、重大ではないエラーメッセージです。関数は正常に終了します。したがって、このメッセージは無視してもかまいません。また、現在のハンドルを閉じて新しいハンドルを開き、リソースの情報にアクセスし直してもかまいません。

これら 3 つの関数は 1 つのマニュアルページ内で説明しています。このマニュアルページには、個々の関数名 `scha_resource_open(3HA)`、`scha_resource_get(3HA)`、`scha_resource_close(3HA)` でアクセスできます。

`scha_resource_setstatus(3HA)`

RGM の制御下のリソースの `Status` および `Status_msg` プロパティを設定します。この関数はリソースのモニターによって使用され、モニターから見たリソースの状態を反映します。

注 - `scha_resource_setstatus` はリソースモニター専用の関数ですが、任意のプログラムから呼び出すことができます。

リソースタイプ関数

これらの関数は、RGM に登録されているリソースタイプについての情報にアクセスします。

`scha_resourcetype_open(3HA)`、`scha_resourcetype_get(3HA)`、`scha_resourcetype_close(3HA)`

`scha_resourcetype_open` 関数は、リソースへのアクセスを初期化し、`scha_resourcetype_get` のハンドルを戻します。`scha_resourcetype_get` 関数はリソースタイプの情報にアクセスします。`scha_resourcetype_close` 関数は、ハンドルを無効にし、`scha_resourcetype_get` の戻り値に割り当てられているメモリーを解放します。

`scha_resourcetype_open` 関数がリソースタイプのハンドルを戻した後に、クラスタの再構成や管理アクションによって、リソースタイプが変更されることがあります。この場合、`scha_resourcetype_get` 関数がハンドルを通じて獲得した情報は正しくない可能性があります。リソースタイプ上でクラスタの再構成や管理アクションが行われた場合、RGM は `scha_err_seqid` エラーコードを `scha_resourcetype_get` 関数に戻し、リソースタイプが変更されたことを示します。これは、重大ではないエラーメッセージです。関数は正常に終了します。したがって、このメッセージは無視してもかまいません。また、現在のハンドルを閉じて新しいハンドルを開き、リソースタイプの情報にアクセスし直してもかまいません。

これら 3 つの関数は 1 つのマニュアルページ内で説明しています。このマニュアルページには、個々の関数名 `scha_resourcetype_open(3HA)`、`scha_resourcetype_get(3HA)`、`scha_resourcetype_close(3HA)` からアクセスできます。

リソースグループ関数

以下の関数を使用すると、リソースグループについての情報にアクセスしたり、リソースグループを再起動できます。

`scha_resourcegroup_open(3HA)`、`scha_resourcegroup_get(3HA)`、`scha_resourcegroup_close(3HA)`

これらの関数は、RGM によって管理されるリソースグループの情報にアクセスします。`scha_resourcegroup_open` 関数は、リソースグループへのアクセスを初期化し、`scha_resourcegroup_get` のハンドルを戻します。

`scha_resourcegroup_get` 関数は、リソースグループの情報にアクセスします。`scha_resourcegroup_close` 関数は、ハンドルを無効にし、`scha_resourcegroup_get` の戻り値に割り当てられているメモリーを解放します。

`scha_resourcegroup_open` 関数がリソースグループのハンドルを戻した後に、クラスタの再構成や管理アクションによって、リソースグループが変更されることがあります。この場合、`scha_resourcegroup_get` 関数がハンドルを通じて獲得した情報は正しくない可能性があります。リソースグループ上でクラスタの再構成や管理アクションが行われた場合、RGM は `scha_err_seqid` エラーコードを `scha_resourcegroup_get` 関数に戻し、リソースグループが変更されたことを示します。これは、重大ではないエラーメッセージです。関数は正常に終了します。したがって、このメッセージは無視してもかまいません。また、現在のハンドルを閉じて新しいハンドルを開き、リソースグループの情報にアクセスし直してもかまいません。

これら 3 つの関数は 1 つのマニュアルページ内で説明しています。このマニュアルページには、個々の関数名 `scha_resourcegroup_open(3HA)`、`scha_resourcegroup_get(3HA)`、`scha_resourcegroup_close(3HA)` からアクセスできます。

`scha_control(3HA)`

RGM の制御下のリソースグループの再起動、または別のノードへの再配置を要求します。

クラスタ関数

以下の関数は、クラスタについての情報にアクセスし、その情報を戻します。

`scha_cluster_open(3HA)`、`scha_cluster_get(3HA)`、`scha_cluster_close(3HA)`

これらの関数は、ノード名、ID、状態、クラスタ名、リソースグループなど、クラスタに関する情報にアクセスします。

これら 3 つの関数は 1 つのマニュアルページ内で説明しています。このマニュアルページには、個々の関数名 `scha_cluster_open(3HA)`、`scha_cluster_get(3HA)`、`scha_cluster_close(3HA)` からアクセスできます。

`scha_cluster_open` 関数がクラスタのハンドルを戻した後に、再構成や管理アクションによって、クラスタが変更されることがあります。この場合、`scha_cluster_get` 関数がハンドルを通じて獲得した情報は正しくない可能性があります。クラスタ上でクラスタの再構成や管理アクションが行われた場合、RGM は `scha_err_seqid` エラーコードを `scha_cluster_get` 関数に戻し、クラスタが変更されたことを示します。これは、重大ではないエラーメッセージです。関数は正常に終了します。したがって、このメッセージは無視してもかまいません。また、現在のハンドルを閉じて新しいハンドルを開き、クラスタの情報にアクセスし直してもかまいません。

`scha_cluster_getlogfacility(3HA)`

クラスタログとして使用されるシステムログ機能の数を戻します。戻された番号を Solaris の `syslog(3)` 関数で使用すると、イベントと状態メッセージをクラスタログに記録できます。

`scha_cluster_getnodename(3HA)`

関数が呼び出されたクラスタノードの名前を戻します。

ユーティリティ関数

この関数は、エラーコードをエラーメッセージに変換します。

`scha_strerror(3HA)`

`scha_` 関数によって戻されるエラーコードを適切なエラーメッセージに変換します。この関数を `logger(1)` と併用すると、メッセージをシステムログ (`syslog(3)`) に記録できます。

RMAPI コールバックメソッド

コールバックメソッドは、リソースタイプを実装するための API が提供する重要な要素です。コールバックメソッドを使用すると、RGM は、クラスタのメンバーシップが変更されたとき (ノードが起動またはクラッシュしたとき) にクラスタ内のリソースを制御できます。

注 - クライアントプログラムがクラスタシステム上の HA サービスを制御するため、コールバックメソッドはルートのアクセス権を持つ RGM によって実行されます。したがって、このようなコールバックメソッドをインストールおよび管理するときは、ファイルの所有権とアクセス権を制限します。特に、このようなコールバックメソッドには、特権付き所有者 (`bin` や `root` など) を割り当てます。

さらに、このようなコールバックメソッドは、書き込み可能にしてはなりません。この節では、コールバックメソッドの引数と終了コードについて説明し、次のカテゴリのコールバックメソッドについて説明します。

- 制御および初期化メソッド
- 管理サポートメソッド
- ネットワーク関連メソッド
- モニター制御メソッド

注 - この節では、メソッドが呼び出されるタイミングやよそうされるリソースへの影響など、コールバックメソッドについて簡単に説明します。詳細については `rt_callbacks(1HA)` のマニュアルページを参照してください。

メソッドの引数

RGM はコールバックメソッドを呼び出すとき、次のような引数を使用します。

`method -R resource-name -T type-name -G group-name`

method は、`start` や `stop` などのコールバックメソッドとして登録されているプログラムのパス名です。リソースタイプのコールバックメソッドは、それらの登録ファイルで宣言します。

コールバックメソッドの引数はすべて、フラグ付きの値として渡されます。`-R` はリソースインスタンスの名前を示し、`-T` はリソースのタイプを示し、`-G` はリソースが構成されているグループを示します。このような引数をアクセス関数で使用すると、リソースについての情報を取得できます。

`validate` メソッドを呼び出すときは、追加の引数 (リソースのプロパティ値と呼び出されるリソースグループ) を使用します。

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

終了コード

終了コードは、すべてのコールバックメソッドで共通で、メソッドの呼び出しによるリソースの状態への影響を示すように定義されています。これらの終了コードについては、`scha_calls(3HA)` のマニュアルページを参照してください。終了コードには、以下のものがあります。

- 0 (ゼロ) – メソッドは成功しました。
- ゼロ以外の任意の値 – メソッドは失敗しました。

RGM は、コールバックメソッドの実行の異常終了 (タイムアウトやコアダンプ) も処理します。

メソッドは、各ノード上で `syslog(3)` を使用して障害情報を出力するように実装する必要があります。`stdout` や `stderr` に書き込まれる出力は、ローカルノードのコンソール上には表示されますが、それをユーザーが確認するかどうかは保証できないためです。

制御および初期化コールバックメソッド

制御および初期化コールバックメソッドは、主に、リソースを起動および停止します。その他にも、リソース上で初期化と終了コードを実行します。

Start

この必須メソッドは、リソースを含むリソースグループがクラスタノード上でオンラインになったとき、そのクラスタノード上で呼び出されます。このメソッドは、そのノード上でリソースを起動します。

`start` メソッドは、ローカルノード上でリソースが起動し、使用可能な状態になるまで終了してはなりません。したがって、`start` メソッドは終了する前にリソースをポーリングし、リソースが起動しているかどうかを判断する必要があります。

す。さらに、このメソッドには、十分な長さのタイムアウト値を設定する必要があります。たとえば、あるリソース (データベースデーモンなど) が起動するのに時間がかかる場合、そのメソッドには十分な長さのタイムアウト値を設定する必要があります。

RGM が Start メソッドの失敗に応答する方法は、Failover_mode プロパティの設定によって異なります。

リソースの Start メソッドのタイムアウト値を設定するには、リソースタイプ登録ファイルの START_TIMEOUT プロパティを使用します。

Stop

この必須メソッドは、リソースを含むリソースグループがクラスタノード上でオフラインになったとき、そのクラスタノード上で呼び出されます。このメソッドは、リソースを (アクティブであれば) 停止します。

Stop メソッドは、ローカルノード上でリソースがすべての活動を完全に停止し、すべてのファイル記述子を終了するまで終了してはなりません。そうしないと、RGM が (実際にはアクティブであるのに) リソースが停止したと判断するため、データが破壊されることがあります。データの破壊を防ぐために最も安全な方法は、ローカルノード上でリソースに関連するすべてのプロセスを停止することです。

Stop メソッドは終了する前にリソースをポーリングし、リソースが停止しているかどうかを判断する必要があります。さらに、このメソッドには、十分な長さのタイムアウト値を設定する必要があります。たとえば、あるリソース (データベースデーモンなど) が停止するのに時間がかかる場合、そのメソッドには十分長めのタイムアウト値を設定する必要があります。

RGM が Stop メソッドの失敗に応答する方法は、Failover_mode プロパティの設定によって異なります (表 A-2 を参照)。

リソースの Stop メソッドのタイムアウト値を設定するには、リソースタイプ登録ファイルの STOP_TIMEOUT プロパティを使用します。

Init

このオプションメソッドは、リソースを管理下に置くとき (リソースが属しているリソースグループを管理していない状態から管理している状態に切り替えるとき、またはすでに管理されているリソースグループでリソースを作成するとき)、1 回だけ呼び出され、リソースの初期化を実行します。このメソッドは、Init_nodes リソースプロパティが示すノード上で呼び出されます。

Fini

このオプションメソッドは、リソースを管理下から外すとき (リソースが属しているリソースグループを管理していない状態に切り替えるとき、またはすでに管理されているリソースグループからリソースを削除するとき) に呼び出され、リソースをクリーンアップします。このメソッドは、Init_nodes リソースプロパティが示すノード上で呼び出されます。

Boot

このオプションメソッドは、Initと同様に、リソースの所属リソースグループがRGMの管理下に置かれたあと、クラスタを結合するノード上のリソースを初期化します。このメソッドは、Init_nodes リソースプロパティが示すノード上で呼び出されます。Boot メソッドは、起動または再起動の結果とし、ノードがクラスタに結合または再結合したときに呼び出されます。

注 - Init、Fini、Boot メソッドが失敗すると、syslog(3) 関数がエラーメッセージを生成しますが、それ以外はRGMのリソース管理に影響しません。

管理サポートメソッド

リソース上での管理アクションには、リソースプロパティの設定と変更があります。Validate と Update コールバックメソッドを使用してリソースタイプを実装すると、このような管理アクションを行うことができます。

Validate

このオプションメソッドは、リソースの作成時と、管理アクションによってリソースまたはそのリソースグループのプロパティが更新されたときに呼び出されます。このメソッドは、リソースタイプの Init_nodes プロパティが示す複数のクラスタノード上で呼び出されます。Validate は、作成または更新が行われる前に呼び出されます。任意のノード上でメソッドから失敗の終了コードが戻ると、作成または更新は取り消されます。

Validate が呼び出されるのは、リソースまたはリソースグループのプロパティが管理アクションを通じて変更されたときだけです。RGM がプロパティを設定したときや、モニターがリソースプロパティ Status や Status_msg を設定したときではありません。

Update

このオプションメソッドを呼び出して、プロパティが変更されたことを実行中のリソースに通知することができます。Update は、管理アクションがリソースまたはリソースグループのプロパティの設定に成功した後に呼び出されます。このメソッドは、リソースがオンラインであるノード上で呼び出されます。このメソッドは、API アクセス関数を使用し、アクティブなリソースに影響する可能性があるプロパティ値を読み取り、その値に従って実行中のリソースを調節します。

Update メソッドが失敗すると、syslog(3) 関数がエラーメッセージを生成しますが、それ以外はRGMのリソース管理に影響しません。

ネットワーク関連コールバックメソッド

ネットワークアドレスリソースを使用するサービスでは、ネットワークアドレス構成に相対的な順番で、起動手順または停止手順を行う必要があります。任意コールバックメソッドの `Prenet_start` と `Postnet_stop` を使用してリソースタイプを実装すると、関連するネットワークアドレスが「起動」に構成される前、または、「停止」に構成された後に、特別な起動アクションまたはシャットダウンアクションを行うことができます。

`Prenet_start`

このオプションメソッドを呼び出して、同じリソースグループ内のネットワークアドレスが構成される前に特殊な起動アクションを実行することができます。

`Postnet_stop`

このオプションメソッドを呼び出して、同じリソースグループ内のネットワークアドレスの構成後に特殊な終了アクションを実行することができます。

モニター制御コールバックメソッド

リソースタイプは、オプションとして、リソースの性能を監視したり、その状態を報告したり、リソースの障害に対処するようなプログラムを含むようにも実装できます。`Monitor_start`、`Monitor_stop`、`Monitor_check` メソッドは、リソースタイプへのリソースモニターの実装をサポートします。

`Monitor_start`

このオプションメソッドを呼び出して、リソースの起動後にリソースの監視を開始することができます。

`Monitor_stop`

このオプションメソッドを呼び出して、リソースの停止前にリソースの監視を停止することができます。

`Monitor_check`

このオプションメソッドを呼び出して、リソースグループを新しいノードに配置する前に、そのノードの信頼性を査定することができます。`Monitor_check` メソッドは、並行して実行中のその他のメソッドと競合しない方法で実装する必要があります。

第 5 章

サンプルデータサービス

この章では、`in.named` アプリケーションを Sun Cluster データサービスとして稼働する HA-DNS について説明します。`in.named` デーモンは Solaris におけるドメインネームサービス (DNS) の実装です。サンプルのデータサービスでは、Resource Management API を使用して、アプリケーションの高可用性を実現する方法を示します。

RMAPI は、シェルスクリプトと C プログラムの両方のインタフェースをサポートします。この章のサンプルアプリケーションはシェルスクリプトインタフェースで作成されています。

この章の内容は、次のとおりです。

- 83 ページの「サンプルデータサービスの概要」
- 84 ページの「リソースタイプ登録ファイルの定義」
- 90 ページの「すべてのメソッドに共通な機能の提供」
- 94 ページの「データサービスの制御」
- 100 ページの「障害モニターの定義」
- 109 ページの「プロパティ更新の処理」

サンプルデータサービスの概要

サンプルのデータサービスはクラスタのイベント (管理アクション、アプリケーションの異常終了、ノードの異常終了など) に応じて、DNS アプリケーションを起動、停止、再起動、およびラスタノード間の切り替えを行います。

アプリケーションの再起動は、プロセス監視機能 (PMF) によって管理されます。アプリケーションの障害が再試行最大期間または再試行最大回数を超えると、障害モニターはアプリケーションリソースを含むリソースグループを別のノードにフェイルオーバーします。

サンプルのデータサービスは、PROBE メソッドという形で障害監視機能を提供します。PROBE メソッドは、nslookup コマンドを使用し、アプリケーションが正常な状態であることを保証します。DNS サービスのハングを検出すると、PROBE メソッドは、DNS アプリケーションをローカルで再起動することによって、この状況を修正しようとします。この方法で状況が改善されず、サービスの問題が繰り返し検出される場合、PROBE メソッドは、サービスをクラスタ内の別のノードにフェイルオーバーしようとします。

サンプルのアプリケーションには、具体的に、次のような機能が含まれています。

- リソースタイプ登録ファイル - データサービスの静的なプロパティを定義します。
- Start コールバックメソッド - HA-DNS データサービスを含むリソースグループがオンラインになるときに RGM によって呼び出され、in.named デーモンを起動します。
- Stop コールバックメソッド - HA-DNS データサービスを含むリソースグループがオフラインになるときに RGM によって呼び出され、in.named デーモンを停止します。
- 障害モニター - DNS サーバーが動作しているかどうかを確認することによって、サービスの信頼性を検査します。障害モニターはユーザー定義の PROBE メソッドによって実装され、Monitor_start と Monitor_stop コールバックメソッドによって起動および停止されます。
- Validate コールバックメソッド - RGM によって呼び出され、サービスの構成ディレクトリがアクセス可能であるかどうかを検査します。
- Update コールバックメソッド - システム管理者がリソースプロパティの値を変更したときに RGM によって呼び出され、障害モニターを再起動します。

リソースタイプ登録ファイルの定義

この例で使用するサンプルのリソースタイプ登録 (RTR) ファイルは、DNS リソースタイプの静的な構成を定義します。このタイプのリソースは、RTR ファイルで定義されているプロパティを継承します。

RTR ファイル内の情報は、クラスタ管理者が HA-DNS データサービスを登録したときに RGM によって読み取られます。

RTR ファイルの概要

RTR ファイルの形式は明確に定義されています。リソースタイププロパティ、システム定義リソースプロパティ、拡張プロパティという順番で並んでいます。詳細は、rt_reg(4) のマニュアルページと 32 ページの「リソースとリソースタイププロパティの設定」を参照してください。

この節では、サンプルの RTR ファイルの特定のプロパティについて説明します。この節で扱うリストは、サンプルの RTR ファイルの一部だけです。サンプルの RTR ファイルの完全なリストについては、219 ページの「リソースタイプ登録ファイルのリスト」を参照してください。

サンプル RTR ファイルのリソースタイププロパティ

次のリストに示すように、サンプルの RTR ファイルはコメントから始まり、その後、HA-DNS 構成を定義するリソースタイププロパティが続きます。

```
#
# Copyright (c) 1998-2001 by Sun Microsystems, Inc.
# All rights reserved.
#
# DNS の登録情報
#

#pragma ident    "@(#)SUNW.sample  1.1  00/05/24 SMI"

RESOURCE_TYPE = "sample";
VENDOR_ID = SUNW;
RT_DESCRIPTION = "Domain Name Service on Sun Cluster";

RT_VERSION = "1.0";
API_VERSION = 2;
FAILOVER = TRUE;

RT_BASEDIR=/opt/SUNWsample/bin;
PKGLIST = SUNWsample;

START          =  dns_svc_start;
STOP           =  dns_svc_stop;

VALIDATE       =  dns_validate;
UPDATE        =  dns_update;

MONITOR_START =  dns_monitor_start;
MONITOR_STOP  =  dns_monitor_stop;
MONITOR_CHECK =  dns_monitor_check;
```

ヒント – RTR ファイルの最初のエントリには、Resource_type プロパティを宣言する必要があります。宣言しないと、リソースタイプの登録は失敗します。

注 – RGM はプロパティ名の大文字と小文字を区別します。Sun が提供する RTR ファイルのプロパティに対する命名規則では、名前の最初の文字が大文字で、残りが小文字です (ただし、メソッド名は例外です)。メソッド名は (プロパティ属性と同様に) すべて大文字です。次に、サンプルのデータサービス (smp1) 用の RTR ファイルにおけるリソースタイプ宣言を示します。

次に、これらのプロパティについての情報を説明します。

- リソースタイプ名は、Resource_type プロパティだけで指定できます (例 :sample)。あるいは、Vendor_id+"."+Resource_type という形式でも指定できます (例 :SUNW.sample)。
Vendor_id を使用する場合、リソースタイプを定義する企業の略号にします。リソースタイプ名はクラスター内で一意である必要があります。
- Rt_version プロパティは、ベンダーによって指定されたサンプルのデータサービスのバージョンを識別します。
- API_version プロパティは Sun Cluster のバージョンを識別します。たとえば、「API_version =2」は、データサービスが Sun Cluster バージョン 3.0 の管理下で動作していることを示します。
- Failover = TRUE の場合、同時に複数のノード上でオンラインになることができるリソースグループでは、データサービスが動作できないことを示します。
- RT_basedir は相対パス (コールバックメソッドのパスなど) を補完するためのディレクトリパスで、/opt/SUNWsample/bin を指します。
- Start、Stop、Validate などは、RGM によって呼び出される個々のコールバックメソッドプログラムへのパスを提供します。これらのパスは、RT_basedir で指定されたディレクトリからの相対パスです。
- Pkglist は、SUNWsample をサンプルのデータサービスのインストールを含むパッケージとして識別します。

この RTR ファイルに指定されていないリソースタイププロパティ (Single_instance、Init_nodes、Installed_nodes など) は、デフォルト値を取得します。リソースタイププロパティの完全なリストとそのデフォルト値については、表 A-1 を参照してください。

クラスター管理者は、RTR ファイルのリソースタイププロパティに指定されている値を変更できません。

サンプル RTR ファイルのリソースプロパティ

慣習上、RTR ファイルでは、リソースプロパティをリソースタイププロパティの後に宣言します。リソースプロパティには、Sun Cluster が提供するシステム定義プロパティと、データサービス開発者が定義する拡張プロパティが含まれます。どちらのタイプの場合でも、Sun Cluster が提供するプロパティ属性の数 (最小、最大、デフォルト値など) を指定できます。

RTR ファイルのシステム定義プロパティ

次のリストは、サンプル RTR ファイルのシステム定義プロパティを示しています。

```
# リソースタイプ宣言の後に、中括弧に囲まれたリソースプロパティ宣言の  
# リストが続く。プロパティ名宣言は、各エントリの左中括弧の直後にある最初  
# の属性である必要がある。
```

```
# <method>_timeout プロパティの値は、RGM がメソッドの呼び出しが  
# 失敗したと結論するまでの時間（秒）を設定する。
```

```
# すべてのメソッドタイムアウトの MIN 値は 60 秒に設定されている。  
# これは、管理者が短すぎる時間を設定するのを防ぐためである。短すぎる  
# 時間を設定すると、スイッチオーバーやフェイルオーバーの性能が上がらず、  
# さらには予期せぬ RGM アクションが発生する可能性がある（誤った  
# フェイルオーバー、ノードの再起動、リソースグループの  
# ERROR_STOP_FAILED 状態への移行など、管理者の介入を必要とする  
# RGM アクション）。  
# メソッドタイムアウトに短すぎる時間を設定すると、データサービス全体の  
# 可用性が低下する。
```

```
{  
    PROPERTY = Start_timeout;  
    MIN=60;  
    DEFAULT=300;  
}  
  
{  
    PROPERTY = Stop_timeout;  
    MIN=60;  
    DEFAULT=300;  
}  
  
{  
    PROPERTY = Validate_timeout;  
    MIN=60;  
    DEFAULT=300;  
}  
  
{  
    PROPERTY = Update_timeout;  
    MIN=60;  
    DEFAULT=300;  
}  
  
{  
    PROPERTY = Monitor_Start_timeout;  
    MIN=60;  
    DEFAULT=300;  
}  
  
{  
    PROPERTY = Monitor_Stop_timeout;  
    MIN=60;  
    DEFAULT=300;  
}  
  
{  
    PROPERTY = Thorough_Probe_Interval;  
    MIN=1;  
    MAX=3600;  
}
```

```

        DEFAULT=60;
        TUNABLE = ANYTIME;
    }

# 当該ノード上でアプリケーションを正常に起動できないと結論するまでに
# 指定された時間内 (Retry_Interval) に行う再試行回数。
{
    PROPERTY = Retry_Count;
    MIN=0;
    MAX=10;
    DEFAULT=2;
    TUNABLE = ANYTIME;
}

# Retry_Interval には 60 の倍数を指定する。これは、この値は秒から分に変換され、
# 端数が切り上げられるためである。
# たとえば、50 (秒) は 1 分に変換される。このプロパティ値は再試行回数
# (Retry_Count ) のタイミングを指定する。
{
    PROPERTY = Retry_Interval;
    MIN=60;
    MAX=3600;
    DEFAULT=300;
    TUNABLE = ANYTIME;
}

{
    PROPERTY = Network_resources_used;
    TUNABLE = AT_CREATION;
    DEFAULT = "";
}

```

Sun Cluster はシステム定義プロパティを提供しますが、リソースプロパティ属性を使用すると、異なるデフォルト値を設定できます。リソースプロパティに適用するために利用できる属性の完全なリストについては、217 ページの「リソースプロパティの属性」を参照してください。

サンプルの RTR ファイル内のシステム定義リソースプロパティについては、次の点に注意してください。

- Sun Cluster は、すべてのタイムアウトに最小値 (1 秒) とデフォルト値 (3600 秒) を提供します。サンプルの RTR ファイルは、最小値をそのまま (60 秒) にし、デフォルト値を 300 秒に変更しています。クラスタ管理者は、このデフォルト値を使用することも、タイムアウト値を変更することもできます (たとえば、60 秒以上)。Sun Cluster は正当な最大値を持っていません。
- Thorough_Probe_Interval、Retry_count、Retry_interval プロパティの TUNABLE 属性は ANYTIME に設定されています。この設定は、データサービスが動作中でも、クラスタ管理者がこれらのプロパティの値を変更できることを意味します。上記のプロパティは、サンプルのデータサービスによって実装される障害モニターによって使用されます。サンプルのデータサービスは、管理アクションによってさまざまなリソースが変更されたときに障害モニターを停止および再起動するように、Update を実装します。114 ページの「Update メソッド」を

参照してください。

- リソースプロパティは次のように分類されます。
 - 必須—クラスタ管理者はリソースを作成するときに必ず値を指定する必要があります。
 - 任意—クラスタ管理者が値を指定しない場合、システムがデフォルト値を提供します。
 - 条件付き—RTR ファイルで宣言されている場合だけ、RGM はプロパティを作成します。

サンプルのデータサービスの障害モニターは、`Thorough_probe_interval`、`Retry_count`、`Retry_interval`、`Network_resources_used` という条件付きプロパティを使用しているため、開発者はこれらのプロパティを RTR ファイルで宣言する必要があります。プロパティの分類方法については、`r_properties` (5) のマニュアルページまたは 203 ページの「リソースプロパティ」を参照してください。

RTR ファイルの拡張プロパティ

次に、RTR ファイルの最後の例として、拡張プロパティを示します。

```
# 拡張プロパティ
# クラスタ管理者は、このプロパティの値として、アプリケーションによって
# 使用される構成ファイルが格納されているディレクトリのパスを指定する。
# このアプリケーション (DNS) は、PXFS 上の DNS 構成ファイルのパス (通常
# named.conf) のパスを指定する。
{
    PROPERTY = Confdir;
    EXTENSION;
    STRING;
    TUNABLE = AT_CREATION;
    DESCRIPTION = "The Configuration Directory Path";
}

# 検証の失敗が宣言されるまでのタイムアウト値 (秒)。
{
    PROPERTY = Probe_timeout;
    EXTENSION;
    INT;
    DEFAULT = 120;
    TUNABLE = ANYTIME;
    DESCRIPTION = "Time out value for the probe (seconds)";
}
```

サンプルの RTR ファイルは 2 つの拡張プロパティ、`Confdir` と `Probe_timeout` を定義します。`Confdir` は、DNS 構成ディレクトリへのパスを指定します。このディレクトリには、DNS が正常に動作するために必要な `in.named` ファイルが格納されています。サンプルのデータサービスの `Start` と `Validate` メソッドはこのプロパティを使用し、DNS を起動する前に、構成ディレクトリと `in.named` ファイルがアクセス可能であるかどうかを確認します。

データサービスが構成される時、Validate メソッドは、新しいディレクトリがアクセス可能であるかどうかを確認します。

サンプルのデータサービスの PROBE メソッドは、Sun Cluster コールバックメソッドではなく、ユーザー定義メソッドです。したがって、Sun Cluster はこの Probe_timeout プロパティを提供しません。開発者はこの拡張プロパティを RTR ファイルに定義し、クラスタ管理者が Probe_timeout の値を構成できるようにする必要があります。

すべてのメソッドに共通な機能の提供

この節では、サンプルのデータサービスのすべてのメソッドで使用される次のような機能について説明します。

- 90 ページの「コマンドインタプリタの指定およびパスのエクスポート」
- 91 ページの「PMF_TAG と SYSLOG_TAG 変数の宣言」
- 92 ページの「関数の引数の構文解析」
- 93 ページの「エラーメッセージの生成」
- 94 ページの「プロパティ情報の取得」

コマンドインタプリタの指定およびパスのエクスポート

シェルスクリプトの最初の行は、コマンドインタプリタを指定します。サンプルのデータサービスの各メソッドスクリプトは、次に示すように、コマンドインタプリタを指定します。

```
#!/bin/ksh
```

サンプルアプリケーション内のすべてのメソッドスクリプトは、Sun Cluster のバイナリとライブラリへのパスをエクスポートします。ユーザーの PATH 設定には依存しません。

```
#####  
# MAIN  
#####  
  
export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH
```

PMF_TAG と SYSLOG_TAG 変数の宣言

すべてのメソッドスクリプト (Validate を除く) は、リソース名を渡し、pmfadm (1M) を使用してデータサービスまたはモニターのいずれかを起動 (または停止) します。各スクリプトは変数 PMF_TAG を定義し、pmfadm に渡すことによって、データサービスまたはモニターを識別できます。

同様に、各メソッドスクリプトは、logger (1) コマンドを使用してメッセージをシステムログに記録します。各スクリプトは変数 SYSLOG_TAG を定義し、-t オプションで logger に渡すことによって、メッセージが記録されるリソースのリソースタイプ、リソースグループ、リソース名を識別できます。

すべてのメソッドは、次に示す例と同じ方法で SYSLOG_TAG を定義します。

dns_probe、dns_svc_start、dns_svc_stop、dns_monitor_check の各メソッドは、次のように PMF_TAG を定義します。なお、pmfadm と logger は dns_svc_stop メソッドのものを使用しています。

```
#####  
# MAIN  
#####
```

```
PMF_TAG=$RESOURCE_NAME.named
```

```
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME
```

```
# データサービスに SIGTERM シグナルを送り、タイムアウト値の 80%  
# が経過するまで待機する。  
pmfadm -s $PMF_TAG.named -w $SMOOTH_TIMEOUT TERM  
if [ $? -ne 0 ]; then  
    logger -p ${SYSLOG_FACILITY}.info \  
        -t [${SYSLOG_TAG}] \  
        "${ARGV0} Failed to stop HA-DNS with SIGTERM; Retry with \  
        SIGKILL"
```

dns_monitor_stop、dns_monitor_stop、dns_update の各メソッドは、次のように PMF_TAG を定義します (なお、pmfadm は dns_monitor_stop メソッドのものを使用しています)。

```
#####  
# MAIN  
#####
```

```
PMF_TAG=$RESOURCE_NAME.monitor
```

```
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME
```

```
...
```

```
# in.named が実行中であるかどうかを確認し、実行中であれば強制終了する。
```

```
if pmfadm -q $PMF_TAG.monitor; then  
    pmfadm -s $PMF_TAG.monitor KILL
```

関数の引数の構文解析

RGM は、次に示すように、すべてのコールバックメソッド (Validate を除く) を呼び出します。

```
method_name -R resource_name -T resource_type_name -G resource_group_name
```

`method_name` は、コールバックメソッドを実装するプログラムのパス名です。データサービスは、各メソッドのパス名を RTR ファイルに指定します。このようなパス名は、RTR ファイルの `Rt_basedir` プロパティに指定されたディレクトリからのパスになります。たとえば、サンプルのデータサービスの RTR ファイルでは、ベースディレクトリとメソッド名は次のように指定されます。

```
RT_BASEDIR=/opt/SUNWsample/bin;
START = dns_svc_start;
STOP = dns_svc_stop;
...
```

コールバックメソッドの引数はすべて、フラグ付きの値として渡されます。-R はリソースインスタンスの名前を示し、-T はリソースのタイプを示し、-G はリソースが構成されているグループを示します。コールバックメソッドの詳細については、`rt_callbacks (1HA)` のマニュアルページを参照してください。

注 - Validate メソッドを呼び出すときは、追加の引数 (リソースのプロパティ値と呼び出されるリソースグループ) を使用します。詳細については、109 ページの「プロパティ更新の処理」を参照してください。

各コールバック、メソッドには、渡された引数を構文解析する関数が必要です。すべてのコールバックメソッドには同じ引数が渡されるので、データサービスは、アプリケーション内のすべてのコールバックメソッドで使用される単一の構文解析関数を提供します。

次に、サンプルのアプリケーションのコールバックメソッドで使用される `parse_args` 関数を示します。

```
#####
# プログラム引数の解析。
#
function parse_args # [args ...]
{
    typeset opt

    while getopts 'R:G:T:' opt
    do
        case "$opt" in
            R)
                # DNS リソース名。
                RESOURCE_NAME=$OPTARG
                ;;
            G)

```

```

        # リソースが構成されたリソースグループ名。
        RESOURCEGROUP_NAME=${OPTARG}
        ;;
T)
        # リソースタイプ名。
        RESOURCETYPE_NAME=${OPTARG}
        ;;
*)
        logger -p ${SYSLOG_FACILITY}.err \
        -t [${RESOURCETYPE_NAME},${RESOURCEGROUP_NAME},${RESOURCE_NAME}] \
        "ERROR: Option ${OPTARG} unknown"
        exit 1
        ;;
esac
done
}

```

注 – サンプルのアプリケーションの PROBE メソッドはユーザー定義メソッドですが、Sun Cluster コールバックメソッドと同じ引数で呼び出されます。したがって、このメソッドには、他のコールバックメソッドと同じ構文解析関数が含まれています。

構文解析関数は、次に示すように、MAIN の中で呼び出されます。

```
parse_args "$@"
```

エラーメッセージの生成

エラーメッセージをエンドユーザーに出力するには、syslog 機能をメソッドに使用することを推奨します。サンプルのデータサービスのすべてのコールバックメソッドは、次に示すように、scha_cluster_get コマンドを使用し、クラスタログ用に使用されている syslog 機能番号を取得します。

```
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`
```

この値はシェル変数 SYSLOG_FACILITY に格納されます。logger (1) コマンドの機能として使用すると、エラーメッセージをクラスタログに記録できます。たとえば、サンプルのデータサービスの Start メソッドは、次に示すように、SYSLOG_FACILITY を取得し、データサービスが起動したことを示すメッセージを記録します。:

```

SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`
...
if [ $? -eq 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err \
        -t [${SYSLOG_TAG}] \
        "${ARGV0} HA-DNS successfully started"
fi

```

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

プロパティ情報の取得

ほとんどのコールバックメソッドは、データサービスのリソースとリソースタイプのプロパティについての情報を取得する必要があります。このために、API は `scha_resource_get` コマンドを提供しています。

リソースプロパティには 2 種類 (システム定義プロパティと拡張プロパティ) あります。システム定義プロパティは事前に定義されており、拡張プロパティはデータサービス開発者が RTR ファイルに定義します。

`scha_resource_get` を使用してシステム定義プロパティの値を取得するときは、`-O` パラメータでプロパティの名前を指定します。このコマンドは、プロパティの値だけを返します。たとえば、サンプルのデータサービスの `Monitor_start` メソッドは検証プログラムを特定し、起動できるようにしておく必要があります。検証プログラムはデータベースのベースディレクトリ (`RT_BASEDIR` プロパティが指す位置) 内に存在します。したがって、`Monitor_start` メソッドは、次に示すように、`RT_BASEDIR` の値を取得し、その値を `RT_BASEDIR` 変数に格納します。

```
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME -G \  
$RESOURCEGROUP_NAME`
```

拡張プロパティの場合、データサービス開発者は、これが拡張プロパティであることを示す `-O` パラメータを指定し、最後のパラメータとしてプロパティの名前を指定する必要があります。拡張プロパティの場合、このコマンドは、プロパティのタイプと値の両方を返します。たとえば、サンプルのデータサービスの検証プログラムは、次に示すように、`probe_timeout` 拡張プロパティのタイプと値を取得し、次に `awk` (1) コマンドを使用して値だけを `PROBE_TIMEOUT` シェル変数に格納します。

```
probe_timeout_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \  
-G $RESOURCEGROUP_NAME Probe_timeout`  
PROBE_TIMEOUT=`echo $probe_timeout_info | awk '{print $2}'`
```

データサービスの制御

データサービスは、クラスタ上でアプリケーションデーモンを起動するために `Start` メソッドまたは `Prenet_start` メソッドを提供し、クラスタ上でアプリケーションデーモンを停止するために `Stop` メソッドまたは `Postnet_stop` メソッドを提供する必要があります。サンプルのデータサービスは、`Start` メソッドと `Stop` メソッドを実装します。代わりに `Prenet_start` メソッドと `Postnet_stop` メソッドを使用する場合は、43 ページの「`Start` および `Stop` メソッドを使用するかどうかの決定」を参照してください。

Start メソッド

データサービスリソースを含むリソースグループがクラスタノード上でオンラインになるとき、あるいは、リソースグループがすでにオンラインになっていて、そのリソースが有効なとき、RGM はそのノード上で start メソッドを呼び出します。サンプルのアプリケーションでは、メソッドはそのノード上で in.named (DNS) デーモンを起動します。

この節では、サンプルのアプリケーションの start メソッドの重要な部分だけを説明します。parse_args 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、90 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

start メソッドの完全なリストについては、222 ページの「start メソッドのコードリスト」を参照してください。

Start の概要

DNS を起動する前に、サンプルのデータサービスの start メソッドは、構成ディレクトリと構成ファイル (named.conf) がアクセス可能で利用可能であるかどうかを確認します。DNS が正常に動作するためには、named.conf の情報が重要です。

このコールバックメソッドは、プロセス監視機能 (pmfadm) を使用し、DNS デーモン (in.named) を起動します。DNS がクラッシュしたり、起動に失敗したりすると、このメソッドは、一定の期間に一定の回数だけ DNS の起動を再試行します。再試行の回数と期間は、データサービスの RTR ファイル内のプロパティで指定されます。

構成の確認

DNS が動作するためには、構成ディレクトリ内の named.conf ファイルからの情報が必要です。したがって、start メソッドは、DNS を起動しようとする前にいくつかの妥当性検査を実行し、ディレクトリやファイルがアクセス可能であるかどうかを確認します。

Confdir 拡張プロパティは、構成ディレクトリへのパスを提供します。プロパティ自身は RTR ファイルに定義されています。しかし、実際の位置は、クラスタ管理者がデータサービスを構成するときに指定します。

サンプルのデータサービスでは、start メソッドは scha_resource_get (1HA) コマンドを使用して構成ディレクトリの位置を取得します。

注 - Confdir は拡張プロパティであるため、scha_resource_get はタイプと値の両方を戻します。したがって、awk (1) コマンドで値だけを取得し、シェル変数 CONFIG_DIR に格納します。

```
# クラスタ管理者がリソースの追加時に設定した Confdir の値を検索。
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
```

```
-G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get は、拡張プロパティの値とともにタイプを返す。
# 拡張プロパティの値だけを取得。
CONFIG_DIR=`echo $config_info | awk '{print $2}'`
```

次に、Start メソッドは CONFIG_DIR の値を使用し、ディレクトリがアクセス可能であるかどうかを確認します。アクセス可能ではない場合、Start メソッドはエラーメッセージを記録し、エラー状態で終了します。97 ページの「Start の終了状態」を参照してください。

```
# $CONFIG_DIR がアクセス可能かどうか確認。
if [ ! -d $CONFIG_DIR ]; then
  logger -p ${SYSLOG_FACILITY}.err \
    -t [${SYSLOG_TAG} \
      "${ARGV0} Directory $CONFIG_DIR is missing or not mounted"
  exit 1
fi
```

アプリケーションデーモンを起動する前に、このメソッドは最終検査を実行し、named.conf ファイルが存在するかどうかを確認します。存在しない場合、Start メソッドはエラーメッセージを記録し、エラー状態で終了します。

```
# データファイルに相対パス名が含まれている場合は $CONFIG_DIR
# ディレクトリに移動。
cd $CONFIG_DIR

# named.conf ファイルが存在することを確認。
if [ ! -s named.conf ]; then
  logger -p ${SYSLOG_FACILITY}.err \
    -t [${SYSLOG_TAG} \
      "${ARGV0} File $CONFIG_DIR/named.conf is missing or empty"
  exit 1
fi
```

アプリケーションの起動

このメソッドは、プロセス監視機能 (pmfadm) を使用してアプリケーションを起動します。pmfadm コマンドを使用すると、アプリケーションを再起動するときの期間と回数を指定できます。このため、RTR ファイルには2つのプロパティ Retry_count と Retry_interval があります。Retry_count は、アプリケーションを再起動する回数を指定し、Retry_interval は、アプリケーションを再起動する期間を指定します。

Start メソッドは、scha_resource_get コマンドを使用して Retry_count と Retry_interval の値を取得し、これらの値をシェル変数に格納します。次に、-n オプションと -t オプションを使用し、これらの値を pmfadm に渡します。

```
# RTR ファイルから再試行回数の値を取得する。
RETRY_CNT=`scha_resource_get -O Retry_Count -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`
```

```

# RTR ファイルから次の再試行までの時間 (秒数) を取得。この値は pmfadm
# に渡されるため分数に変換される。変換時に端数が切り上げられる点に注意。
# たとえば 50 秒は 1 分に変換される。
((RETRY_INTRVAL=`scha_resource_get -O Retry_Interval -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME / 60))

# PMF 制御下で in.named デーモンを起動する。RETRY_INTERVAL の期間、
# $RETRY_COUNT の回数だけクラッシュおよび再起動できる。
# それ以上の回数クラッシュした場合、PMF はそれ以上再試行しない。
# <$PMF_TAG> タグで登録済みのプロセスがある場合、PMF はプロセスが
# すでに実行中であるという警告メッセージを送信する。
pmfadm -c $PMF_TAAG -n $RETRY_CNT -t $RETRY_INTRVAL \
/usr/sbin/in.named -c named.conf

# HA-DNS が起動していることを示すメッセージを記録。
if [ $? -eq 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err \
        -t [$SYSLOG_TAG] \
        "${ARGV0} HA-DNS successfully started"
fi
exit 0

```

Start の終了状態

Start メソッドは、実際のアプリケーションが本当に動作して実行可能になるまで、成功で終了してはなりません。特に、他のデータサービスが依存している場合は注意する必要があります。これを実現するための1つの方法は、Start メソッドが終了する前に、アプリケーションが動作しているかどうかを確認することです。複雑なアプリケーション(データベースなど)の場合、RTR ファイルの Start_timeout プロパティに十分高い値を設定することによって、アプリケーションが初期化され、クラッシュ回復を実行できる時間を提供します。

注 - サンプルのデータサービスのアプリケーションリソース DNS は直ちに起動するため、サンプルのデータサービスは、成功で終了する前に、ポーリングでアプリケーションが動作していることを確認していません。

このメソッドが DNS の起動に失敗し、失敗状態で終了すると、RGM は Failover_mode プロパティを検査し、どのように対処するかを決定します。サンプルのデータサービスは明示的に Failover_mode プロパティを設定していないため、このプロパティはデフォルト値 NONE が設定されています(ただし、クラスタ管理者がデフォルトを変更して異なる値を指定していないと仮定します)。したがって、RGM は、データサービスの状態を設定するだけで、他のアクションは行いません。同じノード上で再起動したり、別のノードにフェイルオーバーしたりするには、ユーザーの介入が必要です。

Stop メソッド

HA-DNS リソースを含むリソースグループがクラスターノード上でオフラインになるとき、あるいは、リソースグループがオンラインでリソースが無効なとき、RGM は Stop メソッドを呼び出します。このメソッドは、そのノード上で `in.named` (DNS) デーモンを停止します。

この節では、サンプルのアプリケーションの Stop メソッドの重要な部分だけを説明します。 `parse_args` 関数や `syslog` 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、90 90 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

Stop メソッドの完全なリストについては、225 ページの「Stop メソッドのコードリスト」を参照してください。

Stop の概要

データサービスを停止するときは、考慮すべきことが 2 点あります。1 点は、停止処理を正しい順序で行うことです。これを実現する最良の方法は、`pmfadm` 経由で `SIGTERM` シグナルを送信することです。

もう 1 点は、データサービスが本当に停止していることを保証することによって、データベースが `Stop_failed` 状態にならないようにすることです。これを実現する最良の方法は、`pmfadm` 経由で `SIGKILL` シグナルを送信することです。

サンプルのデータサービスの Stop メソッドは、このような点を考慮しています。まず、`SIGTERM` シグナルを送信し、このシグナルがデータサービスの停止に失敗した場合は、`SIGKILL` シグナルを送信します。

DNS を停止しようとする前に、この Stop メソッドは、プロセスが実際に動作しているかどうかを確認します。プロセスが動作している場合、Stop メソッドはプロセス監視機能 (`pmfadm`) を使用してプロセスを停止します。

この Stop メソッドは何回か呼びだしてもその動作が変わらないことが保証されます。RGM は、`Start` の呼び出しでデータサービスを起動せずに、Stop メソッドを 2 回呼び出すことはありません。しかし、RGM は、リソースが起動されていなくても、あるいは、リソースが自発的に停止している場合でも、Stop メソッドをそのリソース上で呼び出すことができます。つまり、DNS がすでに動作していない場合でも、この Stop メソッドは成功で終了します。

アプリケーションの停止

Stop メソッドは、データサービスを停止するために 2 段階の方法を提供します。`pmfadm` 経由で `SIGTERM` シグナルを使用する規則正しい方法と、`SIGKILL` シグナルを使用する強制的な方法です。Stop メソッドは、Stop メソッドが戻るまでの時間を示す `Stop_timeout` 値を取得します。次に、Stop メソッドはこの時間の 80% を規則正しい方法に割り当て、15% を強制的な方法に割り当てます (5% は予約済み)。次の例を参照してください。


```

# 状態にするのを避けるため成功で終了する。

exit 0

fi

# DNS の停止に成功。メッセージを記録して成功で終了する。
logger -p ${SYSLOG_FACILITY}.err \
-t [${RESOURCE_TYPE_NAME},${RESOURCE_GROUP_NAME},${RESOURCE_NAME}] \
"HA-DNS successfully stopped"
exit 0

```

Stop の終了状態

Stop メソッドは、実際のアプリケーションが本当に停止するまで、成功で終了してはなりません。特に、他のデータサービスが依存している場合は注意する必要があります。そうしなければ、データが破壊される可能性があります。

複雑なアプリケーション (データベースなど) の場合、RTR ファイルの Stop_timeout プロパティに十分高い値を設定することによって、アプリケーションが停止中にクリーンアップできる時間を提供します。

このメソッドが DNS の停止に失敗し、失敗状態で終了すると、RGM は Failover_mode プロパティを検査し、どのように対処するかを決定します。サンプルのデータサービスは明示的に Failover_mode プロパティを設定していないため、このプロパティはデフォルト値 NONE が設定されています (ただし、クラスタ管理者がデフォルトを変更して異なる値を指定していないと仮定します)。したがって、RGM は、データサービスの状態を Stop_failed に設定するだけで、他のアクションは行いません。アプリケーションを強制的に停止し、Stop_failed 状態をクリアするには、ユーザーの介入が必要です。

障害モニターの定義

サンプルのアプリケーションは、DNS リソース (in.named) の信頼性を監視する基本的な障害モニターを実装します。障害モニターは、次の要素から構成されます。

- dns_probe - nslookup(1M) を使用し、サンプルのデータサービスの制御下にある DNS リソースが動作しているかどうかを確認するユーザー定義プログラム。DNS が動作していない場合、このメソッドは DNS をローカルで再起動しようとします。あるいは、再起動の再試行回数によっては、RGM がデータサービスを別のノードに再配置することを要求します。
- dns_monitor_start - dns_probe を起動するコールバックメソッド。監視が有効である場合、RGM は、サンプルのデータサービスがオンラインになった後、自動的に dns_monitor_start を呼び出します。

- `dns_monitor_stop` - `dns_probe` を停止するコールバックメソッド。RGM は、サンプルのデータサービスがオフラインになる前に、自動的に `dns_monitor_stop` を呼び出します。
- `dns_monitor_check` - PROBE プログラムがデータサービスを新しいノードにフェイルオーバーするとき、`Validate` メソッドを呼び出し、構成ディレクトリが利用可能であるかどうかを確認するコールバックメソッド。

検証プログラム

`dns_probe` プログラムは、サンプルのデータサービスの管理下にある DNS リソースが動作しているかどうかを確認する、連続して動作するプロセスを実行します。

`dns_probe` は、サンプルのデータサービスがオンラインになった後、RGM によって自動的に呼び出される `dns_monitor_start` メソッドによって起動されます。データサービスは、サンプルのデータサービスがオフラインになる前、RGM によって呼び出される `dns_monitor_stop` メソッドによって停止されます。

この節では、サンプルのアプリケーションの PROBE メソッドの重要な部分だけを説明します。`parse_args` 関数や `syslog` 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、90 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

PROBE メソッドの完全なリストについては、228 ページの「PROBE プログラムのコードリスト」を参照してください。

検証プログラムの概要

検証プログラムは無限ループで動作します。検証プログラムは、`nslookup(1M)` を使用し、適切な DNS リソースが動作しているかどうかを確認します。DNS が動作している場合、検証プログラムは一定の期間 (`Thorough_probe_interval` システム定義プロパティに設定されている期間) だけ休眠し、その後、再び検証を行います。DNS が動作していない場合、検証プログラムは DNS をローカルで再起動しようとするか、再起動の再試行回数によっては、RGM がデータサービスを別のノードに再配置することを要求します。

プロパティ値の取得

このプログラムには、次のプロパティ値が必要です。

- `Thorough_probe_interval` - 検証プログラムが休眠する期間を設定します。
- `Probe_timeout` - `nslookup` コマンドが検証を行う期間 (タイムアウト値) を設定します。
- `Network_resources_used` - DNS が動作するサーバーを設定します。
- `Retry_count` と `Retry_interval` - 再起動を行う回数と期間を設定します。

- `Rt_basedir` - PROBE プログラムと `gettime` ユーティリティーが格納されているディレクトリを設定します。

`scha_resource_get` コマンドは、次に示すように、上記プロパティの値を取得し、シェル変数に格納します。

```
PROBE_INTERVAL=`scha_resource_get -O THOROUGH_PROBE_INTERVAL \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`

probe_timeout_info=`scha_resource_get -O Extension -R $RESOURCE_NAME
\
-G $RESOURCEGROUP_NAME Probe_timeout`
PROBE_TIMEOUT=`echo $probe_timeout_info | awk '{print $2}'`

DNS_HOST=`scha_resource_get -O NETWORK_RESOURCES_USED -R $RESOURCE_NAME
\
-G $RESOURCEGROUP_NAME`

RETRY_COUNT=`scha_resource_get -O RETRY_COUNT -R $RESOURCE_NAME
-G\
$RESOURCEGROUP_NAME`

RETRY_INTERVAL=`scha_resource_get -O RETRY_INTERVAL -R $RESOURCE_NAME
-G\
$RESOURCEGROUP_NAME`

RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME -G\
$RESOURCEGROUP_NAME`
```

注 - システム定義プロパティ (`Thorough_probe_interval` など) の場合、`scha_resource_get` は値だけを戻します。拡張プロパティ (`Probe_timeout` など) の場合、`scha_resource_get` はタイプと値を戻します。値だけを取得するには `awk(1)` コマンドを使用します。

サービスの信頼性の検査

検証プログラム自身は、`nslookup(1M)` コマンドの `while` による無限ループです。`while` ループの前に、`nslookup` の応答を保管する一時ファイルを設定します。`probfail` 変数と `retries` 変数は 0 に初期化されます。

```
# nslookup の応答用一時ファイルを設定。
DNSPROBEFILE=/tmp/.$RESOURCE_NAME.probe
probfail=0
retries=0
```

`while` ループ自身は、次の作業を行います。

- 検証プログラム用の休眠期間を設定します。
- `hatimerun(1M)` を使用し、`nslookup` に `Probe_timeout` の値とターゲットホストを渡して起動します。

- nslookup の戻りコード (成功または失敗) に基づいて、*probefail* 変数を設定しません。
- *probefail* が 1 (失敗) に設定された場合、nslookup への応答がサンプルのデータサービスから来ており、他の DNS サーバーから来ているのではないことを確認します。

次に、while ループコードを示します。

```
while :
do
# 検証が実行される時間は THOROUGH_PROBE_INTERVAL プロパティ
# に指定されている。したがって、THOROUGH_PROBE_INTERVAL の間
# 検証が休眠するように設定する。
sleep $PROBE_INTERVAL

# DNS がサービスを提供している IP アドレス上で nslookup コマンドを実行する。
hatimerun -t $PROBE_TIMEOUT /usr/sbin/nslookup $DNS_HOST $DNS_HOST \
> $DNSPROBEFILE 2>&1

    retcode=$?
    if [ $retcode -ne 0 ]; then
        probefail=1
    fi

# nslookup への応答が /etc/resolv.conf ファイルに指定されているその他の
# ネームサーバーではなく HA-DNS サーバーから返されていることを確認する。
if [ $probefail -eq 0 ]; then
# Get the name of the server that replied to the nslookup query.
SERVER=`awk ' $1=="Server:" { print $2 }' \
$DNSPROBEFILE | awk -F. ' { print $1 } ' `
if [ -z "$SERVER" ]; then
    probefail=1
else
    if [ $SERVER != $DNS_HOST ]; then
        probefail=1
    fi
fi
fi
fi
```

再起動とフェイルオーバーの評価

probefail 変数が 0 (成功) 以外である場合、nslookup コマンドがタイムアウトしたか、あるいは、サンプルのサービスの DNS 以外のサーバーから応答が来ていることを示します。どちらの場合でも、DNS サーバーは期待どおりに機能していないので、障害モニターは `decide_restart_or_failover` 関数を呼び出し、データサービスをローカルで起動するか、RGM がデータサービスを別のノードに再配置することを要求するかを決定します。*probefail* 変数が 0 の場合、検証が成功したことを示すメッセージが生成されます。

```
if [ $probefail -ne 0 ]; then
    decide_restart_or_failover
else
```

```
logger -p ${SYSLOG_FACILITY}.err\  
-t [${SYSLOG_TAG}]\  
"${ARGV0} Probe for resource HA-DNS successful"
```

fi

decide_restart_or_failover 関数は、再試行最大期間 (Retry_interval) と再試行最大回数 (Retry_count) を使用し、DNS をローカルで再起動するか、RGM がデータサービスを別のノードに再配置することを要求するかを決定します。この関数は、次のような条件付きコードを実装します。コードリストについては、228 ページの「PROBE プログラムのコードリスト」にある decide_restart_or_failover を参照してください。

- 最初の障害である場合、データサービスをローカルで再起動します。エラーメッセージを記録し、retries 変数の再試行カウンタをインクリメントします。
- 最初の障害ではなく、再試行時間が再試行最大期間を過ぎている場合、データサービスをローカルで再起動します。エラーメッセージを記録し、再試行カウンタをリセットし、再試行時間をリセットします。
- 再試行時間が再試行最大期間を過ぎておらず、再試行カウンタが再試行最大回数を超えている場合、別のノードにフェイルオーバーします。フェイルオーバーが失敗すると、エラーメッセージを記録し、検証プログラムを状態 1 (失敗) で終了します。
- 再試行時間が再試行最大期間を過ぎておらず、再試行カウンタが再試行最大回数を超えていない場合、データサービスをローカルで再起動します。エラーメッセージを記録し、retries 変数の再試行カウンタをインクリメントします。

期限 (再試行最大期間) 内に再起動の回数 (再試行カウンタ) が制限 (再試行最大回数) に到達した場合、この関数は、RGM がデータサービスを別のノードに再配置することを要求します。再起動の回数が制限に到達していない場合、あるいは、再試行最大期間を過ぎていて、再試行カウンタをリセットする場合、この関数は DNS を同じノード上で再起動しようとします。この関数については、次の点に注意してください。

- gettimeofday ユーティリティを使用すると、再起動間の時間を追跡できます。このユーティリティは C プログラムで、(Rt_basedir) ディレクトリ内にあります。
- Retry_count と Retry_interval のシステム定義リソースプロパティは、再起動を行う回数と期間を決定します。RTR ファイルのデフォルト値は、Retry_count が 2 回、Retry_interval が 5 分 (300 秒) です。クラスタ管理者はこのデフォルトを変更できます。
- restart_service 関数は、同じノード上でデータサービスの再起動を試行する場合に呼び出されます。この関数の詳細については、105 ページの「データサービスの再起動」を参照してください。
- API コマンド scha_control は、GIVEOVER オプションを指定すると、サンプルデータサービスを含むリソースグループをオフラインにし、別のノード上でオンラインにし直します。

データサービスの再起動

restart_service 関数は、decide_restart_or_failover によって呼び出され、同じノード上でデータサービスの再起動を試行します。この関数は次の作業を行います。

- データサービスが PMF 下にまだ登録されているかどうかを調べます。サービスが登録されている場合、この関数は次の作業を行います。
 - データサービスの Stop メソッド名と Stop_timeout 値を取得します。
 - hatimerun を使用してデータサービスの Stop メソッドを起動し、Stop_timeout 値を渡します。
 - (データサービスが正常に停止した場合) データサービスの Start メソッド名と Start_timeout 値を取得します。
 - hatimerun を使用してデータサービスの Start メソッドを起動し、Start_timeout 値を渡します。
- データサービスが PMF 下に登録されていない場合は、データサービスが PMF 下で許可されている再試行最大回数を超過していることを示しています。したがって、GIVEOVER オプションを指定して scha_control 関数を呼び出し、データサービスを別のノードにフェイルオーバーします。

```
function restart_service
{
    # データサービスを再起動するには、まずデータサービス自体が PMF に
    # 登録されているかどうかを確認する。
    pmfadm -q $PMF_TAG
    if [[ $? -eq 0 ]]; then
        # データサービスの TAG が PMF に登録されている場合、データ
        # サービスを停止し、再起動する。当該リソースの Stop メソッド名と
        # STOP_TIMEOUT 値を取得する。
        STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT \
            -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
        STOP_METHOD=`scha_resource_get -O STOP \
            -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
        hatimerun -t $STOP_TIMEOUT $RT_BASEDIR/$STOP_METHOD \
            -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
            -T $RESOURCE_TYPE_NAME

        if [[ $? -ne 0 ]]; then
            logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                "${ARGV0} Stop method failed."
            return 1
        fi

        # 当該リソースの Start メソッド名と START_TIMEOUT 値を取得する。
        START_TIMEOUT=`scha_resource_get -O START_TIMEOUT \
            -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
        START_METHOD=`scha_resource_get -O START \
            -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
        hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD \
            -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
```

```

        -T $RESOURCE_TYPE_NAME

    if [[ $? -ne 0 ]]; then
        logger-p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}] \
            "${ARGV0} Start method failed."
        return 1
    fi

else
    # データサービスの TAG が PMF に登録されていない場合、
    # データサービスが PMF で許可されている最大再試行回数
    # を超過している。したがって、データサービスを再起動しては
    # ならない。代わりに、同じクラスタ内の別のノードへの
    # フェイルオーバーを試みる。
    scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME \
        -R $RESOURCE_NAME
fi

return 0
}

```

検証プログラムの終了状態

ローカルでの再起動が失敗したり、別のノードへのフェイルオーバーが失敗したりすると、サンプルのデータサービスの PROBE プログラムは失敗で終了し、“Failover attempt failed (フェイルオーバーは失敗しました)” というエラーメッセージを記録します。

Monitor_start メソッド

サンプルのデータサービスがオンラインになった後、RGM は Monitor_start メソッドを呼び出し、dns_probe メソッドを起動します。

この節では、サンプルアプリケーションの Monitor_start メソッドの重要な部分だけを説明します。parse_args 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、90 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

Monitor_start メソッドの完全なリストについては、234 ページの「Monitor_start メソッドのコードリスト」を参照してください。

Monitor_start の概要

このメソッドは、プロセス監視機能 (pmfadm) を使用して検証プログラムを起動します。

検証プログラムの起動

Monitor_start メソッドは、Rt_basedir プロパティの値を取得し、PROBE プログラムへの完全パス名を構築します。このメソッドは、pmfadm の無限再試行オプション (-n -1, -t -1) を使用して検証プログラムを起動します。つまり、検証プログラムの起動に失敗しても、PMF メソッドは検証プログラムを無限に再起動します。

```
# リソースの RT_BASEDIR プロパティを取得し、検証プログラムが存在する
# 場所を確認する。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`

# PMF の制御下でデータサービスの検証を開始する。無限再試行オプションを使って
# 検証プログラムを起動する。リソースの名前、タイプ、グループを検証
# プログラムに渡す。
pmfadm -c $RESOURCE_NAME.monitor -n -1 -t -1 \
    $RT_BASEDIR/dns_probe -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
    -T $RESOURCETYPE_NAME
```

Monitor_stop メソッド

サンプルのデータサービスがオフラインになるとき、RGM は Monitor_stop メソッドを呼び出し、dns_probe の実行を停止します。

この節では、サンプルアプリケーションの Monitor_stop メソッドの重要な部分だけを説明します。parse_args 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、90 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

Monitor_stop メソッドの完全なリストについては、236 ページの「Monitor_stop メソッドのコードリスト」を参照してください。

Monitor_stop の概要

このメソッドは、プロセス監視機能 (pmfadm) を使用して検証プログラムが動作しているかどうかを判断し、動作している場合は検証プログラムを停止します。

検証プログラムの停止

Monitor_stop メソッドは、pmfadm -q を使用して検証プログラムが動作しているかどうかを判断し、動作している場合は pmfadm -s を使用して検証プログラムを停止します。検証プログラムがすでに停止している場合でも、このメソッドは成功で終了します。これによって、メソッドが呼び出し回数に依存しないことが保証されます。

```
# 検証プログラムが動作しているかどうかを判断し、動作している場合は停止する。
if pmfadm -q $PMF_TAG; then
    pmfadm -s $PMF_TAG KILL
```

```

if [ $? -ne 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err \
        -t [${SYSLOG_TAG} \
            "${ARGV0} Could not stop monitor for resource " \
            $RESOURCE_NAME
    exit 1
else
    # 検証プログラムの停止に成功。メッセージを記録する。
    logger -p ${SYSLOG_FACILITY}.err \
        -t [${SYSLOG_TAG} \
            "${ARGV0} Monitor for resource " $RESOURCE_NAME \
            " successfully stopped"
fi
fi
exit 0

```



注意 – 検証プログラムを停止するときは、必ず、`pmfadm` で `KILL` シグナルを使用するようにしてください。絶対に、マスク可能なシグナル (`TERM` など) は使用しないでください。そうしないと、`Monitor_stop` メソッドが無限にハングし、結果としてタイムアウトする可能性があります。この問題の原因は、`PROBE` メソッドがデータサービスを再起動またはフェイルオーバーする必要があるときに、`scha_control` を呼び出すところにあります。`scha_control` がデータサービスをオフラインにするプロセスの一部として `Monitor_stop` メソッドを呼び出したときに、`Monitor_stop` メソッドがマスク可能なシグナルを使用していると、`Monitor_stop` メソッドは `scha_control` が終了するのを待ち、`scha_control` は `Monitor_stop` メソッドが終了するのを待つため、結果として両方がハングします。

Monitor_stop の終了状態

`PROBE` メソッドを停止できない場合、`Monitor_stop` メソッドはエラーメッセージを記録します。RGM は、主ノード上でサンプルのデータサービスを `MONITOR_FAILED` 状態にするため、そのノードに障害が発生することがあります。

`Monitor_stop` メソッドは、検証プログラムが停止するまで終了してはなりません。

Monitor_check メソッド

`PROBE` メソッドが、データサービスを含むリソースグループを新しいノードにフェイルオーバーしようとするとき、RGM は `Monitor_check` メソッドを呼び出します。

この節では、サンプルアプリケーションの `Monitor_check` メソッドの重要な部分だけを説明します。`parse_args` 関数や `syslog` 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、90 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

`Monitor_check` メソッドの完全なリストについては、238 ページの「`Monitor_check` メソッドのコードリスト」を参照してください。

Monitor_check メソッドは、並行して実行中のその他のメソッドと競合しない方法で実装する必要があります。

Monitor_check メソッドは Validate メソッドを呼び出し、新しいノード上で DNS 構成ディレクトリが利用可能かどうかを確認します。Confdir 拡張プロパティが DNS 構成ディレクトリを指します。したがって、Monitor_check は Validate メソッドのパスと名前、および Confdir の値を取得します。Monitor_check は、次のように、この値を Validate に渡します。

```
# リソースタイプの RT_BASEDIR プロパティから Validate メソッドの
# 完全パスを取得する。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# 当該リソースの Validate メソッド名を取得する。
VALIDATE_METHOD=`scha_resource_get -O VALIDATE \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`

# データサービスを起動するための Confdir プロパティの値を取得する。入力された
# リソース名とリソースグループを使用し、リソースを追加するときに設定した
# Confdir の値を取得する。
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get は、Confdir 拡張プロパティの値とともにタイプも戻す。
# awk を使用し、Confdir 拡張プロパティの値だけを取得する。
CONFIG_DIR=`echo $config_info | awk '{print $2}'`

# Validate メソッドを呼び出し、データサービスを新しいノードにフェイルオーバー
# できるかどうかを確認する。
$RT_BASEDIR/$VALIDATE_METHOD -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCETYPE_NAME -x Confdir=$CONFIG_DIR
```

ノードがデータサービスのホストとして最適であるかどうかをサンプルアプリケーションが確認する方法については、110 ページの「Validate メソッド」を参照してください。

プロパティ更新の処理

サンプルのデータサービスは、クラスタ管理者によるプロパティの更新を処理するために、Validate メソッドと Update メソッドを実装します。

Validate メソッド

リソースが作成されたとき、および、リソースまたは (リソースを含む) リソースグループのプロパティが管理アクションによって更新される時、RGM は Validate メソッドを呼び出します。RGM は、作成または更新が行われる前に、Validate メソッドを呼び出します。任意のノード上でメソッドから失敗の終了コードが戻ると、作成または更新は取り消されます。

RGM が Validate メソッドを呼び出すのは、管理アクションがリソースまたはグループのプロパティを変更したときだけです。RGM がプロパティを設定したときや、モニターがリソースプロパティ `Status` と `Status_msg` を設定したときではありません。

注 - PROBE メソッドがデータサービスを新しいノードにフェイルオーバーする場合、`Monitor_check` メソッドも明示的に Validate メソッドを呼び出します。

Validate の概要

RGM は、他のメソッドに渡す追加の引数 (更新されるプロパティと値を含む) を指定して、Validate メソッドを呼び出します。したがって、サンプルのデータサービスの Validate メソッドは、追加の引数を処理する別の `parse_args` 関数を実装する必要があります。

サンプルのデータサービスの Validate メソッドは、単一のプロパティである `Confdir` 拡張プロパティを確認します。このプロパティは、DNS が正常に動作するために重要な DNS 構成ディレクトリを指します。

注 - DNS が動作している間、構成ディレクトリは変更できないため、`Confdir` プロパティは `RTR` ファイルで `TUNABLE = AT CREATION` と宣言します。したがって、Validate メソッドが呼び出されるのは、更新の結果として `Confdir` プロパティを確認するためではなく、データサービスリソースが作成されているときだけです。

RGM が Validate メソッドに渡すプロパティの中に `Confdir` が存在する場合、`parse_args` 関数はその値を取得および保存します。次に、Validate メソッドは、`Confdir` の新しい値が指すディレクトリがアクセス可能かどうか、および、`named.conf` ファイルがそのディレクトリ内に存在し、データを持っているかどうかを確認します。

`parse_args` 関数が、RGM から渡されたコマンド行引数から `Confdir` の値を取得できない場合でも、Validate メソッドは `Confdir` プロパティの妥当性を検査しようとします。まず、Validate メソッドは `scha_resource_get` 関数を使用し、静的な構成から `Confdir` の値を取得します。次に、同じ検査を実行し、構成ディレクトリがアクセス可能かどうか、および、空でない `named.conf` ファイルがそのディレクトリ内に存在するかどうかを確認します。

Validate メソッドが失敗で終了した場合、Confdir だけでなく、すべてのプロパティの更新または作成が失敗します。

Validate メソッドの構文解析関数

RGM は、他のコールバックメソッドとは異なるパラメータを Validate メソッドに渡します。したがって、Validate メソッドには、他のメソッドとは異なる引数を構文解析する関数が必要です。Validate メソッドやその他のコールバックメソッドに渡される引数の詳細については、`rt_callbacks(1HA)` のマニュアルページを参照してください。次に、Validate メソッドの `parse_args` 関数を示します。

```
#####  
# Validate 引数の構文解析。  
#  
function parse_args # [args...]  
{  
  
    typeset opt  
    while getopts 'cur:x:g:R:T:G:' opt  
    do  
        case "$opt" in  
            R)  
                # DNS リソース名。  
                RESOURCE_NAME=$OPTARG  
                ;;  
            G)  
                # リソースが構成されたリソースグループ名。  
                RESOURCEGROUP_NAME=$OPTARG  
                ;;  
            T)  
                # リソースタイプ名。  
                RESOURCETYPE_NAME=$OPTARG  
                ;;  
            r)  
                # メソッドはシステム定義プロパティ  
                # にアクセスしていない。したがって、  
                # このフラグは動作なし。  
                ;;  
            g)  
                # メソッドはリソースグループプロパティに  
                # アクセスしていない。したがって、  
                # このフラグは動作なし。  
                ;;  
            c)  
                # Validate メソッドがリソースの作成中に  
                # 呼び出されていることを示す。したがって、  
                # このフラグは動作なし。  
                ;;  
            u)  
                # リソースがすでに存在しているときは、  
                # プロパティの更新を示す。Confdir  
                # プロパティを更新する場合、Confdir
```

```

# がコマンド行引数に現れる。現れない場合、メソッドは
# scha_resource_get を使用して
# Confdir を探す必要がある。
UPDATE_PROPERTY=1
;;
x)
# 拡張プロパティのリスト。プロパティ
# と値のペア。区切り文字は「=」
PROPERTY=`echo $OPTARG | awk -F= '{print $1}'`
VAL=`echo $OPTARG | awk -F= '{print $2}'`
# Confdir 拡張プロパティがコマンド行
# 上に存在する場合、その値を記録する。
if [ $PROPERTY == "Confdir" ]; then
    CONFDIR=$VAL
    CONFDIR_FOUND=1
fi
;;
*)
logger -p ${SYSLOG_FACILITY}.err \
-t [${SYSLOG_TAG}] \
"ERROR: Option $OPTARG unknown"
exit 1
;;
esac
done
}

```

他のメソッドの `parse_args` 関数と同様に、この関数は、リソース名を取得するためのフラグ (R)、リソースグループ名を取得するためのフラグ (G)、RGM から渡されるリソースタイプを取得するためのフラグ (T) を提供します。

このメソッドはリソースが更新されるときに拡張プロパティの妥当性を検査するために呼び出されるため、`r` フラグ (システム定義プロパティを示す)、`g` フラグ (リソースグループプロパティを示す)、`c` フラグ (リソースの作成中に妥当性の検査が行われていることを示す) は無視されます。

`u` フラグは、`UPDATE_PROPERTY` シェル変数の値を 1 (TRUE) に設定します。`x` フラグは、更新されているプロパティの名前と値を取得します。更新されているプロパティの中に `Confdir` が存在する場合、その値が `CONFDIR` シェル変数に格納され、`CONFDIR_FOUND` 変数が 1 (TRUE) に設定されます。

Confdir の妥当性検査

`Validate` メソッドはまず、その `MAIN` 関数において、`CONFDIR` 変数を空の文字列に設定し、`UPDATE_PROPERTY` と `CONFDIR_FOUND` を 0 に設定します。

```

CONFDIR=""
UPDATE_PROPERTY=0
CONFDIR_FOUND=0

```

次に、`Validate` メソッドは `parse_args` 関数を呼び出し、RGM から渡された引数を構文解析します。

```
parse_args "$@"
```

次に、Validate は、Validate メソッドがプロパティの更新の結果として呼び出されているかどうか、および、Confdir 拡張プロパティがコマンド行上に存在するかどうかを検査します。次に、Validate メソッドは、Confdir プロパティが値を持っているかどうかを確認します。値を持っていない場合、Validate メソッドはエラーメッセージを記録し、失敗状態で終了します。

```
if ( (( $UPDATE_PROPERTY == 1 )) && (( CONFDIR_FOUND == 0 )) ); then
    config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
        -G $RESOURCEGROUP_NAME Confdir`
    CONFDIR=`echo $config_info | awk '{print $2}'`
fi

# Confdir プロパティが値を持っているかどうかを確認する。持っていない場合、状態1 (失敗) で終了する。
if [[ -z $CONFDIR ]]; then
    logger -p ${SYSLOG_FACILITY}.err \
        "${ARGV0} Validate method for resource "$RESOURCE_NAME " failed"
    exit 1
fi
```

注 - 上記コードにおいて、Validate メソッドが更新の結果として呼び出されているのか ($\$UPDATE_PROPERTY == 1$)、および、プロパティがコマンド行上に存在しないのか ($CONFDIR_FOUND == 0$) を検査し、両者が TRUE である場合に、`scha_resource_get` 関数を使用して Confdir の既存の値を取得するところに注目してください。Confdir がコマンド行上に存在する ($CONFDIR_FOUND == 1$) 場合、CONFDIR の値は、`scha_resource_get` 関数からではなく、`parse_args` 関数から取得されます。

次に、Validate メソッドは CONFDIR の値を使用し、ディレクトリがアクセス可能であるかどうかを確認します。アクセス可能ではない場合、Validate メソッドはエラーメッセージを記録し、エラー状態で終了します。

```
# $CONFDIR がアクセス可能であるかどうかを検査する。
if [ ! -d $CONFDIR ]; then
    logger -p ${SYSLOG_FACILITY}.err \
        -t [${SYSLOG_TAG}] \
        "${ARGV0} Directory $CONFDIR missing or not mounted"
    exit 1
fi
```

Confdir プロパティの更新の妥当性を検査する前に、Validate メソッドは最終検査を実行し、`named.conf` ファイルが存在するかどうかを確認します。存在しない場合、Validate メソッドはエラーメッセージを記録し、エラー状態で終了します。

```
# named.conf ファイルがConfdir ディレクトリ内に存在するかどうかを
# 検査する。
if [ ! -s $CONFDIR/named.conf ]; then
    logger -p ${SYSLOG_FACILITY}.err \
        -t [${SYSLOG_TAG}] \
```

```
    "${ARGV0} File $CONFDIR/named.conf is missing or empty"
    exit 1
fi
```

最終検査を通過した場合、Validate メソッドは、成功を示すメッセージを記録し、成功状態で終了します。

```
# Validate メソッドが成功したことを示すメッセージを記録する。
logger -p ${SYSLOG_FACILITY}.err \
    -t [SYSLOG_TAG] \
    "${ARGV0} Validate method for resource "$RESOURCE_NAME \
    " completed successfully"

exit 0
```

Validate の終了状態

Validate メソッドが成功 (0) で終了すると、新しい値を持つ Confdir が作成されず。Validate メソッドが失敗 (1) で終了すると、Confdir を含むすべてのプロパティが作成されず、理由を示すメッセージがクラスタ管理者に送信されます。

Update メソッド

リソースのプロパティが変更されたとき、RGM は Update メソッドを呼び出し、動作中のリソースにその旨を通知します。RGM は、管理アクションがリソースまたはそのリソースグループのプロパティの設定に成功したあとに、Update を呼び出します。このメソッドは、リソースがオンラインであるノード上で呼び出されます。

Update の概要

Update メソッドはプロパティを更新しません。プロパティの更新は RGM が行います。その代わりに、動作中のプロセスに更新が発生したことを通知します。サンプルのデータサービスでは、プロパティの更新によって影響を受けるプロセスは障害モニターだけです。したがって、Update メソッドは、障害モニターを停止および再起動します。

Update メソッドは、障害モニターが動作していることを確認してから、pmfadm で障害モニターを強制終了する必要があります。Update メソッドは、障害モニターを実装する検証プログラムの位置を取得し、その後、もう一度 pmfadm で障害モニターを再起動します。

Update による障害モニターの停止

Update メソッドは、pmfadm -q を使用し、障害モニターが動作していることを確認します。動作している場合、pmfadm -s TERM で障害モニターを強制終了します。障害モニターが正常に終了した場合、その影響を示すメッセージが管理ユーザーに送信されます。障害モニターが停止できない場合、Update メソッドは、エラーメッセージを管理ユーザーに送信し、失敗状態で終了します。

```

if pmfadm -q $RESOURCE_NAME.monitor; then

# すでに動作している障害モニターを強制終了する。
pmfadm -s $PMF_TAG TERM
  if [ $? -ne 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err \
      -t [${SYSLOG_TAG}] \
        "${ARGV0} Could not stop the monitor"
    exit 1
  else
    # DNS の停止に成功。メッセージを記録する。
    logger -p ${SYSLOG_FACILITY}.err \
      -t [${RESOURCETYPE_NAME}, $RESOURCEGROUP_NAME, $RESOURCE_NAME] \
        "Monitor for HA-DNS successfully stopped"
  fi

```

障害モニターの再起動

障害モニターを再起動するために、Update メソッドは検証プログラムを実装するスクリプトの位置を見つける必要があります。検証プログラムはデータサービスのベースディレクトリ (Rt_basedir プロパティが指すディレクトリ) 内にあります。Update は、次に示すように、Rt_basedir の値を取得し、RT_BASEDIR 変数に格納します。

```

RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`

```

次に、Update は、RT_BASEDIR の値を pmfadm で使用し、dns_probe プログラムを再起動します。検証プログラムを再起動できた場合、Update メソッドはその影響を示すメッセージを管理ユーザーに送信し、成功で終了します。pmfadm が検証プログラムを再起動できない場合、Update メソッドはエラーメッセージを記録し、失敗状態で終了します。

Update の終了状態

Update メソッドが失敗すると、リソースが“update failed” (更新失敗) の状態になります。この状態は RGM のリソース管理に影響しません。しかし、syslog 機能を通じて、管理ツールへの更新アクションが失敗したことを示します。

第 6 章

データサービス開発ライブラリ (DSDL)

この章では、データサービス開発ライブラリ (Data Service Development Library: DSDL) を構成するアプリケーションプログラミングインタフェースの概要について説明します。DSDL は `libdsdev.so` ライブラリとして実装されており、Sun Cluster パッケージに含まれています。

この章の内容は、次のとおりです。

- 117 ページの「DSDL の概要」
- 118 ページの「構成プロパティの管理」
- 119 ページの「データサービスの起動と停止」
- 119 ページの「障害モニターの実装」
- 120 ページの「ネットワークアドレス情報へのアクセス」
- 120 ページの「実装したリソースタイプのデバッグ」

DSDL の概要

DSDL API は、RMAPI の最上位の階層を形成します。DSDLAPI は RMAPI の代わりになるものではなく、RMAPI をカプセル化および拡張するためのものであることに注意してください。DSDL は、特定の Sun Cluster 統合問題に対する事前定義されたソリューションを提供することによって、データサービスの開発を簡素化します。その結果、アプリケーションに本来求められている高可用性とスケーラビリティの実現に、より多くの開発時間を割くことが可能になります。また、アプリケーションの起動、シャットダウン、および監視機能を Sun Cluster に統合する際に、多くの時間を費やすこともありません。

構成プロパティの管理

すべてのコールバックメソッドは構成プロパティにアクセスする必要があります。DSDL は、以下により、プロパティへのアクセスを容易にします。

- 環境の初期化
- プロパティ値を簡単に取得できる関数セットの提供

`scds_initialize` 関数 (各コールバックメソッドの開始時に呼び出す必要がある) は、次の処理を行います。

- RGM がコールバックメソッドに渡すコマンド行引数 (`argc` と `argv[]`) を検査および処理します。そのため、コマンド行解析関数を作成する必要はありません。
- 他の DSDL 関数が使用できるように内部データ構造を設定します。たとえば、DSDL で提供されている関数によって RGM から取得されたプロパティ値はこのデータ構造に格納されます。同様に、コマンド行から入力された値 (RGM から取得された値よりも優先される) もこのデータ構造に格納されます。

注 - `Validate` メソッドの場合、`scds_initialize` はコマンド行で渡されたプロパティ値を解析します。そのため、`Validate` 用の解析関数を作成する必要はありません。

また、`scds_initialize` 関数はロギング環境を初期化して、障害モニターの検証設定の妥当性を検査します。

DSDL は、リソース、リソースタイプ、リソースグループのプロパティ、および、よく使用される拡張プロパティを取得するための関数セットを提供します。これらの関数は、次のような規則に従って、プロパティへのアクセスを標準化しています。

- 各関数は、`scds_initialize` から戻されるハンドル引数だけを取ります。
- 各関数は特定のプロパティに対応します。つまり、関数の戻り値のタイプは取得するプロパティ値のタイプに一致します。
- 値は `scds_initialize` によってあらかじめ算出されているため、関数はエラーを戻しません。新しい値がコマンド行で渡された場合を除き、関数は RGM から値を取得します。

データサービスの起動と停止

Start メソッドは、クラスタノード上でデータサービスを起動するために必要なアクションを実行します。通常、このようなアクションには、リソースプロパティの取得、アプリケーション固有の実行可能ファイルおよび構成ファイルの格納先の特定、および適切なコマンド行引数を用いたアプリケーションの起動が含まれます。

`scds_initialize` 関数はリソース構成を取得します。Start メソッドはプロパティ用の `DSDL` 関数を使用して、アプリケーションを起動するのに必要な構成ディレクトリや構成ファイルを識別するための特定のプロパティ (`Confdir_list` など) の値を取得します。

Start メソッドは、`scds_pmf_start` を呼び出して、プロセス監視機能 (PMF) の制御下でアプリケーションを起動します。PMF を使用すると、プロセスに適用する監視レベルを指定したり、異常終了したプロセスを再起動したりできます。`DSDL` で実装する Start メソッドの例については、138 ページの「`xfnts_start` メソッド」を参照してください。

Stop メソッドは呼び出し回数に依存しないように実装されていなければなりません。つまり、アプリケーションが動作していないときにノード上で呼び出された場合でも、正常終了する必要があります。Stop メソッドが失敗した場合、停止するリソースが `STOP_FAILED` 状態に設定され、クラスタの再起動を招いてしまう可能性があります。

リソースが `STOP_FAILED` 状態になるのを防止するために、Stop メソッドはあらゆる手段を構じてリソースを停止する必要があります。`scds_pmf_stop` 関数は、段階的にリソースを停止しようとします。まず、`SIGTERM` シグナルを使用してリソースを停止しようとします。これに失敗した場合は、`SIGKILL` シグナルを使用します。詳細については、`scds_pmf_stop` のマニュアルページを参照してください。

障害モニターの実装

`DSDL` は、事前に定義されたモデルを提供することによって、障害モニターを実装する際の煩雑さをほとんど取り除きます。リソースがノード上で起動すると、`Monitor_start` メソッドは `PMF` の制御下で障害モニターを起動します。リソースがノード上で動作している間、障害モニターは無限ループを実行します。次に、`DSDL` 障害モニターのロジックの概要を示します。

- `scds_fm_sleep` 関数は `Thorough_probe_interval` プロパティを使用して、検証を行う期間を決定します。この期間中に `PMF` がアプリケーションプロセスの失敗を決定した場合、リソースは再起動されます。

- 検証機能自身は、障害の重要度を示す値を戻します。この値の範囲は、0 (障害なし) から 100 (致命的な障害) までです。
- 検証機能が戻した値は、`scds_action` 関数に送信されます。`scds_action` 関数は、`Retry_interval` プロパティの期間中に、障害の履歴を累積します。
- `scds_action` 関数は、次に示すような、障害が発生した場合の処置を決定します。
 - 累積した障害が 100 より少ない場合は、何もしません。
 - 累積した障害が 100 に到達した場合 (完全な障害)、データサービスを再起動します。`Retry_interval` を超えた場合、障害の履歴をリセットします。
 - `Retry_interval` で指定された期間中に、再起動の回数が `Retry_count` プロパティを上回った場合、データサービスをフェイルオーバーします。

ネットワークアドレス情報へのアクセス

DSDL は、リソースおよびリソースグループのネットワークアドレス情報を戻す関数を提供します。たとえば、`scds_get_netaddr_list` は、リソースが使用するネットワークアドレスリソースを取得して、障害モニターがアプリケーションを検証できるようにします。

また、DSDL は TCP ベースの監視を行う関数セットも提供します。通常、このような関数はサービスとの間に単純なソケット接続を確立し、サービスのデータを読み書きした後で、サービスとの接続を切断します。検証の結果を DSDL の `scds_fm_action` 関数に送信し、次に実行すべき処理を決定できます。

TCP ベースの障害監視の例については、150 ページの「`svc_probe` 関数」を参照してください。

実装したリソースタイプのデバッグ

DSDL は、データサービスをデバッグするときに役立つ組み込み機能を提供します。

DSDL の `scds_syslog_debug()` ユーティリティは、実装したリソースタイプにデバッグ文を追加するための基本的なフレームワークを提供します。デバッグレベル (1 から 9 までの数字) は、各クラスタノード上のリソースタイプごとに動的に設定できます。ファイル `/var/cluster/rgm/rt/<rtname>/loglevel` は、1 から 9 までの整数だけが含まれているファイルであり、すべてのリソースタイプコールバックメソッドはこの

ファイルを読み取ります。DSDL の `scds_initialize()` ルーチンはこのファイルを読み取って、内部デバッグレベルを指定されたレベルに設定します。デフォルトのデバッグレベルは 0 であり、この場合、データサービスはデバッグメッセージを記録しません。

`scds_syslog_debug()` 関数は、`LOG_DEBUG` の優先順位において、`scha_cluster_getlogfacility(3HA)` 関数から戻された機能を使用します。このようなデバッグメッセージは `/etc/syslog.conf` で構成できます。

`scds_syslog` ユーティリティーを使用すると、いくつかのデバッグメッセージをリソースタイプの通常の動作 (おそらくは `LOG_INFO` 優先順位) における情報メッセージとして使用することができます。第 8 章のサンプル DSDL アプリケーションでは、`scds_syslog_debug` と `scds_syslog` 関数が多用されています。

高可用性ローカルファイルシステムの有効化

HAStoragePlus リソースタイプを使用すると、ローカルファイルシステムを Sun Cluster 環境内で高可用性にすることができます。このためには、ローカルファイルシステムのパーティションが広域ディスクグループに存在し、アフィニティスイッチオーバーが有効であり、Sun Cluster 環境がフェイルオーバー用に構成されている必要があります。これによって、多重ホストディスクに直接接続された任意のホストから、多重ホストディスク上の任意のファイルシステムにアクセスできるようになります。入出力が多いデータサービスでは、高可用性のローカルファイルシステムを使用することを強く推奨します。また、『Sun Cluster 3.1 データサービスのインストールと構成』の登録と構成の手順には、HAStoragePlus リソースタイプの構成が追加されます。

第 7 章

リソースタイプの設計

この章では、リソースタイプの設計や実装で DSDL を通常どのように使用するかについて説明します。この章では、特に、リソース構成を検証したり、リソースの開始、停止、および監視を行なったりするためのリソースタイプの設計について説明します。そして、最後に、リソースタイプのコールバックメソッドを DSDL を使って導入する方法を説明します。詳細は、`rt_callbacks(1HA)` のマニュアルページを参照してください。

リソースタイプの開発者がこのような作業を行うためには、リソースのプロパティ設定値にアクセスできなければなりません。プログラムは DSDL のユーティリティ関数 `scds_initialize()` を使うことで、統一された方法でこのようリソースプロパティにアクセスできます。この機能は、各コールバックメソッドの始めの部分で呼び出す必要があります。このユーティリティ関数は、クラスタフレームワークからリソースのすべてのプロパティを取り出します。これによって、これらのプロパティは、`scds_get()` 関数群からアクセスできるようになります。

RTR ファイル

RTR (Resource Type Registration、リソースタイプ登録) ファイルは、リソースタイプの中でも重要なコンポーネントです。Sun Cluster は、リソースタイプの詳細な情報をこのファイルから取得します。この情報には、この実装に必要なプロパティや、それらのデータタイプやデフォルト値、リソースタイプの実装に必要なコールバックメソッドのファイルシステムパス、システム定義プロパティのさまざまな設定値などがあります。

ほとんどのリソースタイプ実装では、DSDL に添付されるサンプル RTR ファイルだけで十分なはずですが、ただし、リソースタイプの名前やリソースタイプのコールバックメソッドのパス名など、いくつかの基本的な要素は編集する必要があります。リソー

スタイプを実装する際に新しいプロパティが必要な場合は、そのプロパティをリソースタイプ実装のリソースタイプ登録 (RTR) ファイルに拡張プロパティとして宣言します。新しいプロパティには、DSDL の `scds_get_ext_property()` ユーティリティを使ってアクセスできます。

Validate メソッド

リソースタイプ実装の `Validate` メソッドは、1) リソースタイプの新しいリソースが作成されているときや、2) リソースまたはリソースグループのプロパティが更新されているときにそれぞれ RGM から呼び出されます。この2つの操作は、リソースの `Validate` メソッドに渡されるコマンド行オプション `-c` (作成) と `-u` (更新) によって区別されます。

`Validate` メソッドは、リソースタイププロパティ `INIT_NODES` の値で定義されるノード群の各ノードに対して呼び出されます。たとえば、`INIT_NODES` に `RG_PRIMARYES` が設定されている場合、`Validate` は、そのリソースのリソースグループを収容できる (その主ノードになりうる) 各ノードに対して呼び出されます。`INIT_NODES` が `RT_INSTALLED_NODES` に設定されている場合、`Validate` は、リソースタイプソフトウェアがインストールされている各ノード (通常は、クラスタのすべてのノード) に対して呼び出されます。`INIT_NODES` のデフォルト値は `RG_PRIMARYES` です (`rt_reg(4)` のマニュアルページを参照)。`Validate` メソッドが呼び出される時点では、RGM はまだリソースを作成していません (作成コールバックの場合)。あるいは、更新するプロパティの更新値をまだ適用していません (更新コールバックの場合)。リソースタイプ実装の `Validate` コールバックメソッドの目的は、リソースの新しい設定値 (リソースに対して指定された新しいプロパティ設定値) がそのリソースタイプにとって有効であるかどうかを検査することにあります。

注 - `HAStoragePlus` によって管理されるローカルファイルシステムを使用している場合は、`scds_hasp_check` を使って `HAStoragePlus` リソースの状態を検査します。この情報は、そのリソースが依存するすべての `SUNW.HAStoragePlus(5)` リソースの状態 (オンラインかどうか) から、そのリソースに定義されたシステムプロパティ `Resource_dependencies` または `Resource_dependencies_weak` を使って取得されます。`scds_hasp_check` 呼び出しから返される状態コードの完全なリストについては、`scds_hasp_check(3HA)` のマニュアルページを参照してください。

DSDL 関数 `scds_initialize()` は、リソースの作成や更新をそれぞれ次のように処理します。

- リソースの作成では、コマンド行から渡された新しいリソースプロパティを解析します。これによって、リソースタイプの開発者は、リソースプロパティの新しい値を、そのリソースがすでにシステムに作成されているかのように使用できます。

- リソースやリソースグループの更新では、管理者によって更新されようとしているプロパティの新しい値をコマンド行から読み込み、残りのプロパティ (値が更新されないもの) をリソース管理 API を使って Sun Cluster から読み込みます。ただし、リソースタイプの開発者は、DSDL を使用する限り、このような初期作業を行う必要はありません。さらに、開発者は、リソースのすべてのプロパティが使用可能であるものとして、リソースの検証を行うことができます。

次の図に示す `svc_validate()` は、リソースプロパティの検証を実装する関数です。この関数は、`scds_get_*()` 関数群を使って、検証しようとするプロパティを検査します。リソースの設定が有効ならこの関数から戻りコード 0 が返されるとすると、リソースタイプの `Validate` メソッドは、次のコード部分のようになります。

```

in
tmain(int argc, char *argv[])
{
    scds_handle_t handle;
    int rc;

    if (scds_initialize(&handle, argc, argv) != SCHA_ERR_NOERR) {
        return (1); /* 初期化エラー */
    }
    rc = svc_validate(handle);
    scds_close(&handle);
    return (rc);
}

```

さらに、検証関数は、リソースの設定が有効でない場合は、その理由を記録する必要があります。`svc_validate()` ルーチンの例 (詳細は省略) は、次のようになります (実際的な検証ルーチンについては、次の章を参照)。

```

int
svc_validate(scds_handle_t handle)
{
    scha_str_array_t *confdirs;
    struct stat      statbuf;
    confdirs = scds_get_confdir_list(handle);
    if (stat(confdirs->str_array[0], &statbuf) == -1) {
        return (1); /* 無効なリソースプロパティ設定 */
    }
    return (0); /* 有効な設定 */
}

```

このように、リソースタイプの開発者は、`svc_validate()` ルーチンの実装だけに集中できます。リソースタイプ実装の典型的な例としては、`app.conf` というアプリケーション構成ファイルを `Confdir_list` プロパティの下に置く処理があります。この処理は、`Confdir_list` プロパティから取り出した適切なパス名に対して `stat()` システム呼び出しを実行することによって実装できます。

Start メソッド

リソースタイプ実装の `start` コールバックメソッドは、特定のクラスタノードのリソースを開始するときに RGM によって呼び出されます。リソースグループ名とリソース名、およびリソースタイプ名はコマンド行から渡されます。`start` メソッドは、クラスタノードでデータサービスリソースを開始するために必要なアクションを行います。通常、このようなアクションには、リソースプロパティの取得や、アプリケーション固有の実行可能ファイルと構成ファイル (または、どちらか) の場所の特定、適切なコマンド行引数を使用したアプリケーションの起動などがあります。

DSDL では、リソース構成ファイルが `scds_initialize()` ユーティリティによってすでに取得されています。アプリケーションの起動アクションは、`svc_start()` ルーチンに指定できます。さらに、アプリケーションが実際に起動されたかどうかを確認するために、`svc_wait()` ルーチンを呼び出すことができます。`start` メソッドのコード (詳細は省略) は、次のようになります。

```
int
main(int argc, char *argv[])
{
    scds_handle_t handle;

    if (scds_initialize(&handle, argc, argv) != SCHA_ERR_NOERR) {
        return (1); /* 初期化エラー */
    }
    if (svc_validate(handle) != 0) {
        return (1); /* 無効な設定 */
    }
    if (svc_start(handle) != 0) {
        return (1); /* 起動に失敗 */
    }
    return (svc_wait(handle));
}
```

この起動メソッドの実装では、`svc_validate()` を呼び出してリソース構成を検証します。検証結果が正しくない場合は、リソース構成とアプリケーション構成が一致していないか、このクラスタノードのシステムに関して何らかの問題があることを示しています。たとえば、リソースに必要な広域ファイルシステムが現在このクラスタノードで使用できない可能性などが考えられます。その場合には、このクラスタノードでこのリソースを起動しても意味がないので、RGM を使って別のノードのリソースを起動すべきです。ただし、この場合、`svc_validate()` は十分に限定的であるものとします。その場合、このルーチンは、アプリケーションが必要とするリソースがあるかどうかをそのクラスタノードだけで検査します。そうでないと、このリソースはすべてのクラスタノードで起動に失敗し、`START_FAILED` の状態になる可能性があります。リソースのこの状態については、`scswitch (1M)` のマニュアルページおよび『*Sun Cluster 3.1 データサービスのインストールと構成*』を参照してください。

`svc_start()` ルーチンは、このノードでリソースの起動に成功した場合は戻りコード 0 を、問題を検出した場合は 0 以外の戻りコードをそれぞれ返す必要があります。このルーチンから 0 以外の値が返されると、RGM は、このリソースを別のクラスタノードで起動しようと試みます。

DSDL を最大限に活用するには、`svc_start()` ルーチンで `scds_pmf_start()` ユーティリティを使って、アプリケーションを PMF (プロセス管理機能) のもとで起動できます。このユーティリティは、PMF の障害コールバックアクション機能 (`pmfadm(1M)` の `-a` アクションフラグを参照) を使って、プロセス障害の検出を実装します。

Stop メソッド

リソースタイプ実装の Stop コールバックメソッドは、特定のクラスタノードでアプリケーションを停止するときに RGM によって呼び出されます。Stop メソッドのコールバックが有効であるためには、次の条件が必要です。

- Stop メソッドは結果に依存しない命令 (*idempotent*) でなければなりません。つまり、Stop メソッドは、そのノードで Start メソッドが正常に終了していなくても、RGM から呼び出されることがあります。したがって、Stop メソッドは、そのクラスタノードでアプリケーションが動作していない場合でも (したがって、特別な処理が必要ない場合でも)、正常に (終了コード 0 で) 終了しなければなりません。
- リソースタイプの Stop メソッドが特定のクラスタノードで失敗に終わると (戻りコードが 0 以外だと)、そのリソースタイプは `STOP_FAILED` の状態になります。この場合、リソースの `Failover_mode` 設定によっては、このクラスタノードが RGM によって強制的に再起動されることがあります。したがって、Stop メソッドの設計時には、アプリケーションを停止する手段をメソッドに組み込んでおくことが重要です。たとえば、アプリケーションが停止しない場合は、`SIGKILL` などを使って、アプリケーションを強制的かつ即時に停止する必要があります。さらに、この処理は一定の時間内に行われなければなりません。Stop_timeout で設定した時間が経過すると、停止が失敗したものとみなされ、リソースは `STOP_FAILED` の状態になるからです。

ほとんどのアプリケーションには、DSDL ユーティリティ `scds_pmf_stop()` で十分なはずですが。このユーティリティは、まず、アプリケーションが PMF の `scds_pmf_start()` で起動されたものとみなして、アプリケーションを `SIGTERM` で「静かに」停止しようとします。これで停止しない場合は、プロセスに対して `SIGKILL` を適用します。このユーティリティの詳細については、196 ページの「PMF 関数」を参照してください。

アプリケーションを停止するそのアプリケーション固有のルーチンを `svc_stop()` とし、これまで使用してきたコードモデルに従うとするなら、`Stop` メソッドは、次のように実装できます。`svc_stop()` の実装で `scds_pmf_stop()` が使用されているかどうかは、ここでは関係ありません。それが使用されているかどうかは、アプリケーションが PMF のもとで `Start` メソッドによって起動されているかどうかに依存します。

```
if (scds_initialize(&handle, argc, argv) != SCHA_ERR_NOERR)
{
    return (1);    /* 初期化エラー */
}
return (svc_stop(handle));
```

`Stop` メソッドの実装では、`svc_validate()` メソッドは使用されません。システムに問題があったとしても、`Stop` メソッドは、このノードでこのアプリケーションを `Stop` すべきだからです。

Monitor_start メソッド

RGM は、`Monitor_start` メソッドを呼び出して、リソースに対する障害モニターを起動します。障害モニターは、このリソースによって管理されているアプリケーションの状態を監視します。リソースタイプの実装では、通常、障害モニターはバックグラウンドで動作する独立したデーモンとして実装されます。このデーモンの起動には、適切な引数をもつ `Monitor_start` コールバックメソッドが使用されます。

モニターデーモン自体は障害が発生しやすいため (たとえば、モニターは、アプリケーションを、監視されない状態にしたまま停止することがある)、モニターデーモンは、PMF を使って起動すべきです。DSDL ユーティリティー `scds_pmf_start()` には、障害モニターを起動する機能が組み込まれています。このユーティリティーは、モニターデーモンプログラムの相対パス名 (リソースタイプコールバックメソッド実装の場所を表す `RT_basedir` との相対パス) を使用します。さらに、ユーティリティーは、DSDL によって管理される `Monitor_retry_interval` と `Monitor_retry_count` 拡張プロパティを使って、デーモンが際限なく再起動されるのを防止します。モニターデーモンのコマンド行構文には、コールバックメソッドに対して定義されたコマンド行構文と同じものが使用されます (`-R resource -G resource_group -T resource_type`)。ただし、モニターデーモンが RGM から直接呼び出されることはありません。このユーティリティーでは、モニターデーモン実装自体が `scds_initialize()` ユーティリティーを使って独自の環境を設定できます。したがって、主な作業は、モニターデーモン自体を設計することです。

Monitor_stop メソッド

RGM は、Monitor_stop メソッドを使って、Monitor_start メソッドで起動された障害モニターデーモンを停止します。このコールバックメソッドの失敗は、Stop メソッドの失敗とまったく同じように処理されます。したがって、Monitor_stop メソッドは、Stop メソッドと同じように強固なものでなければなりません。

障害モニターデーモンを `scds_pmf_start()` ユーティリティを使って起動したら、`scds_pmf_stop()` ユーティリティで停止する必要があります。

Monitor_check メソッド

クラスタノードが特定のリソースを支配できるかどうかを確認するために (つまり、そのリソースによって管理されるアプリケーションがそのノードで正常に動作するかどうかを確認するために)、そのリソースの Monitor_check コールバックメソッドがそのリソースのノードで呼び出されます。通常、この呼び出しでは、アプリケーションに必要なすべてのシステムリソースが本当にクラスタノードで使用可能かどうかを確認されます。124 ページの「Validate メソッド」で述べたように、開発者が実装する `svc_validate()` ルーチンでは、少なくともこの確認が行われなければなりません。

リソースタイプ実装によって管理されているアプリケーションによっては、Monitor_check メソッドでその他の作業を行うことがあります。Monitor_check メソッドは、並行して実行中のその他のメソッドと競合しない方法で実装する必要があります。DSDL を使用する場合には、リソースプロパティに対するアプリケーション固有の検証を実装するために作成された `svc_validate()` ルーチンを Monitor_check メソッドで活用することをお勧めします。

Update メソッド

RGM は、リソースタイプ実装の Update メソッドを呼び出して、システム管理者が行なったすべての変更をアクティブリソースの構成に適用します。Update メソッドは、そのリソースがオンラインになっているすべてのノードに対して呼び出されます。

リソースの構成に対して行われた変更は、リソースタイプ実装にとって必ず有効なものです。RGM は、リソースタイプの Update メソッドを呼び出す前に Validate メソッドを呼び出すからです。Validate メソッドは、リソースやリソースグループの

プロパティが変更される前に呼び出されます。したがって、Validate メソッドは新しい変更を拒否できます。変更が適用されると、Update メソッドが呼び出され、新しい設定値がアクティブ (オンライン) リソースに通知されます。

リソースタイプの開発者は、どのプロパティを動的に変更できるようにするかを慎重に決定し、RTR ファイルでこれらのプロパティに TUNABLE=ANYTIME を設定する必要があります。通常、リソースタイプ実装の障害モニターデーモンによって使用されるすべてのプロパティは、動的に変更できるように設定できます。ただし、Update メソッドの実装が少なくともモニターデーモンを再起動できなければなりません。

このようなプロパティの候補には次のものがあります。

- Thorough_Probe_Interval
- Retry_Count
- Retry_Interval
- Monitor_retry_count
- Monitor_retry_interval
- Probe_timeout

これらのプロパティは、障害モニターデーモンがサービスの状態検査をどのような頻度でどのように行うかや、どのような履歴間隔でエラーを追跡するか、PMF によってどのような再起動しきい値が設定されるかなどに影響します。DSDL には、これらのプロパティの更新を実装するための `scds_pmf_restart()` ユーティリティーが備わっています。

リソースプロパティを動的に更新可能に設定する必要があるが、プロパティの変更によって動作中のアプリケーションに影響が及ぶ可能性がある、とリソースタイプの開発者が判断した場合は、適切なアクションを実装することによって、プロパティに対する変更がアプリケーションの動作中のインスタンスに正しく適用されるようにしなければなりません。DSDL には、現在のところ、この問題をサポートする機能はありません。変更されたプロパティをコマンド行から Update に渡すことはできません (Validate に渡すことは可能)。

Init、Fini、Boot の各メソッド

これらのメソッドは、「1 度だけのアクション」を行うためのものです (リソース管理 API 仕様の定義を参照)。DSDL のサンプル実装には、これらのメソッドの使い方は示されていません。しかし、これらのメソッドを使用する必要がある場合には、DSDL のすべての機能をこれらのメソッドでも使用できます。通常、「1 度だけのアクション」を実装するリソースタイプ実装では、Init メソッドと Boot メソッドはまったく同じように機能します。Fini メソッドは、一般に、Init メソッドや Boot メソッドのアクションを「取り消す」ためのアクションに使用されます。

障害モニターデーモンの設計

DSDL を使用したリソースタイプ実装の障害モニターデーモンには、通常、次の役割があります。

- 管理されているアプリケーションの状態を定期的に監視します。モニターデーモンのこの役割はアプリケーションに大きく依存します。したがって、リソースタイプによって大幅に異なることがあります。DSDL には、TCP に基づく簡単なサービスの状態を検査するいくつかのユーティリティー関数が組み込まれています。HTTP、NNTP、IMAP、POP3 など、ASCII ベースのプロトコルを実装するアプリケーションは、これらのユーティリティーを使って実装できます。
- アプリケーションによって検出された問題をリソースプロパティ `Retry_interval` や `Retry_count` を使って追跡します。さらに、アプリケーションが異常停止した場合には、PMF アクションスクリプトを使ってサービスを再起動すべきかどうかや、アプリケーションの障害が頻繁に発生するためにフェイルオーバーを考慮すべきかどうかを判断します。DSDL では、この機構の実装を助けるユーティリティーとして `scds_fm_action()` と `scds_fm_sleep()` が提供されます。
- アプリケーションを再起動するか、リソースを含むリソースグループのフェイルオーバーを試みるなど、適切なアクションを実行します。DSDL ユーティリティー `scds_fm_action()` には、このようなアルゴリズムが実装されています。そのために、このアルゴリズムは、過去の `Retry_interval` で指定した秒数の間に起った検証障害の累積を計算します。
- リソースの状態を更新します。これによって、`scstat` コマンドやクラスタ管理 GUI からアプリケーションの状態を知ることができます。

DSDL ユーティリティーの設計では、障害モニターデーモンの主要ループは次のようになっています。

DSDL を使って実装された障害モニターでは、

- アプリケーションプロセスの異常停止は、`scds_fm_sleep()` によって比較的迅速に検出されます。これは、PMF によるプロセス停止の通知が非同期に行われるためです。これは、障害モニターが時々ループから復帰してサービスの状態を検査し、アプリケーションの停止を検出する方法と対比的です。DSDL を使用した障害モニターでは障害検出時間が大幅に短縮されるため、サービスの可用性が向上します。
- RGM が `scha_control(3HA)` API によるサービスのフェイルオーバーを拒否すると、`scds_fm_action()` は、現在の障害履歴を「リセット」(消去) します。このようにするのは、障害履歴が `Retry_count` の値をすでに超えているからです。さらに、モニターデーモンは、次のサイクルでスリープから復帰した後に、デーモンの状態検査を正常に完了できないと、`scha_control()` を再び呼び出そうとするはずですが、しかし、前回のサイクルで呼び出しが拒否され状況が依然として残っていれば、この呼び出しは今回も拒否されるはずですが。履歴がリセットされていれば、障害モニターは、少なくとも、次のサイクルでアプリケーションの再起

動などによってその状況を内部的に訂正しようとします。

- 再起動が失敗に終わった場合、`scds_fm_action()` は、アプリケーション障害履歴をリセットしません。これは、状況が訂正されなければ、`scha_control()` が間もなく呼び出される可能性が高いからです。
- ユーティリティー `scds_fm_action()` は、障害履歴に従って、`SCHA_RSSTATUS_OK`、`SCHA_RSSTATUS_DEGRADED`、`SCHA_RSSTATUS_FAULTED` のどれかをリソースステータスに設定します。これによって、ステータスをクラスタシステム管理から使用できるようになります。

ほとんどの場合、アプリケーション固有の状態検査アクションは、スタンドアロンの別個のユーティリティー (たとえば、`svc_probe()`) として実装してから、この汎用的なメインループに統合できます。

```
for (;;) {  
  
    /* 正常な検証と検証の間の thorough_probe_interval  
    * だけスリープする。*/  
    (void) scds_fm_sleep(scds_handle,  
        scds_get_rs_thorough_probe_interval(scds_handle));  
  
    /* 使用するすべてのipaddress を検証する。  
    * 次の各要素を繰り返し検証する。  
    * 1. 使用するすべてのネットリソース  
    * 2. 特定のリソースのすべてのipaddresses  
    * 検証するipaddress ごとに  
    * 障害履歴を計算する。*/  
    probe_result = 0;  
    /* すべてのリソースを繰り返し調べて、  
    * svc_probe() の呼び出しに使用する各IP アドレスを取得する。*/  
    for (ip = 0; ip < netaddr->num_netaddrs; ip++) {  
        /* 状態を検証する必要があるホスト名とポート  
        * を取得する。  
        */  
        hostname = netaddr->netaddrs[ip].hostname;  
        port = netaddr->netaddrs[ip].port_proto.port;  
        /*  
        * HA-XFS は1 つのポートしかサポートしないため、  
        * ポート配列の最初のエン트리から  
        * ポート値を取得する。  
        */  
        ht1 = gethrtime(); /* Latch probe start time */  
        probe_result = svc_probe(scds_handle,  
  
            hostname, port, timeout);  
        /*  
        * サービス検証履歴を更新し、  
        * 必要に応じてアクションを実行する。  
        * 検証終了時刻を保存する。  
        */  
        ht2 = gethrtime();  
        /* ミリ秒に変換する。*/  
        dt = (ulong_t)((ht2 - ht1) / 1e6);
```

```
/*
 * 障害履歴を計算し、
 * 必要に応じてアクションを実行する。
 */
(void) scds_fm_action(scds_handle,
probe_result, (long)dt);
} /* 各ネットワークリソース */
} /* 検証を続ける。*/
```


第 8 章

サンプル DSDL リソースタイプの実装

この章では、DSDL で実装したサンプルのリソースタイプ `SUNW.xfnts` について説明します。データサービスは C 言語で作成されています。使用するアプリケーションは TCP/IP ベースのサービスである X Font Server です。

この章の内容は、次のとおりです。

- 135 ページの「X Font Server について」
- 137 ページの「`SUNW.xfnts` の RTR ファイル」
- 138 ページの「`scds_initialize` の呼び出し」
- 138 ページの「`xfnts_start` メソッド」
- 143 ページの「`xfnts_stop` メソッド」
- 144 ページの「`xfnts_monitor_start` メソッド」
- 145 ページの「`xfnts_monitor_stop` メソッド」
- 147 ページの「`xfnts_monitor_check` メソッド」
- 147 ページの「`SUNW.xfnts` 障害モニター」
- 153 ページの「`xfnts_validate` メソッド」

X Font Server について

X Font Server は、フォントファイルをクライアントに提供する、簡単な TCP/IP ベースのサービスです。クライアントはサーバーに接続してフォントセットを要求します。サーバーはフォントファイルをディスクから読み取って、クライアントにサービスを提供します。X Font Server デーモンはサーバーバイナリである `/usr/openwin/bin/xfns` から構成されます。このデーモンは通常、`inetd` から起動されますが、このサンプルでは、`/etc/inetd.conf` ファイル内の適切なエントリが (たとえば、`fsadmin -d` コマンドによって) 無効にされているものと想定します。したがって、デーモンは Sun Cluster だけの制御下にあります。

次に、X Font Server の構成ファイルについて説明します。

構成ファイル

デフォルトでは、X Font Server はその構成情報をファイル `/usr/openwin/lib/X11/fontserver.cfg` から読み取ります。このファイルのカタログエントリには、デーモンがサービスを提供できるフォントディレクトリのリストが入っています。クラスタ管理者は広域ファイルシステム上のフォントディレクトリの格納先を指定できます。こうすることによって、システム上でフォントデータベースのコピーを1つだけ保持すれば済むので、Sun Cluster 上の X Font Server の使用を最適化できます。広域ファイルシステム上のフォントディレクトリの格納先を指定するには、`fontserver.cfg` を編集して、フォントディレクトリの新しいパスを反映させる必要があります。

構成を簡単にするために、管理者は構成ファイル自身も広域ファイルシステム上に配置できます。`xfps` デーモンはデフォルトの格納先 (このファイルの組み込み場所) を変更するためのコマンド行引数を提供します。`SUNW.xfnts` リソースタイプは、次のコマンドを使用して、Sun Cluster の制御下でデーモンを起動します。

```
/usr/openwin/bin/xfps -config <location_of_cfg_file>/fontserver.cfg \  
-port <portnumber>
```

`SUNW.xfnts` v リソースタイプの実装では、`Confdir_list` プロパティを使用して、`fontserver.cfg` 構成ファイルの格納場所を管理できます。

TCP ポート番号

`xfps` サーバーデーモンが通信する TCP ポートの番号は、一般に「fs」ポート (通常、`/etc/services` ファイルで 7100 と定義されている) です。ただし、`xfps` コマンド行で `-port` オプションを指定することにより、システム管理者はデフォルトの設定を変更できます。`SUNW.xfnts` リソースタイプの `Port_list` プロパティを使用すると、デフォルト値を設定したり、`xfps` コマンド行で `-port` オプションを指定できるようになります。RTR ファイルにおいて、このプロパティのデフォルト値を `7100/tcp` と定義します。`SUNW.xfnts` の `Start` メソッドで、`Port_list` を `xfps` コマンド行の `-port` オプションに渡します。このようにすると、このリソースタイプのユーザーはポート番号を指定する必要がなくなります。つまり、デフォルトのポートが `7100/tcp` になります。ただし、リソースタイプを構成するときに、`Port_list` プロパティに異なる値を指定することにより、別のポートを指定することも可能です。

命名規約

次の命名規則を覚えておけば、サンプルコード内で関数とメソッドを簡単に識別できます。

- RMAPI 関数の名前は、`scha_` で始まります。

- DSDL 関数の名前は、`scds_` で始まります。
- コールバックメソッドの名前は、`xfnts_` で始まります。
- ユーザー定義関数の名前は、`svc_` で始まります。

SUNW.xfnts の RTR ファイル

この節では、SUNW.xfnts の RTR ファイルで宣言されている、いくつかの重要なプロパティについて説明します。各プロパティの目的については説明しません。プロパティの詳細については、32 ページの「リソースとリソースタイププロパティの設定」を参照してください。

次に示すように、`Confdir_list` 拡張プロパティは構成ディレクトリ (または、ディレクトリのリスト) を指定します。

例 8-1

```
{
    PROPERTY = Confdir_list;
    EXTENSION;
    STRINGARRAY;
    TUNABLE = AT_CREATION;
    DESCRIPTION = "The Configuration Directory Path(s)";
}
```

`Confdir_list` プロパティには、デフォルト値は設定されていません。クラスタ管理者はリソースを作成するときに、構成ディレクトリを指定する必要があります。

「`TUNABLE = AT_CREATION`」が指定されているので、作成時以降、この値を変更することはできません。

次に示すように、`Port_list` プロパティは、サーバーデーモンが通信するポートを指定します。

例 8-2

```
{
    PROPERTY = Port_list;
    DEFAULT = 7100/tcp;
    TUNABLE = AT_CREATION;
}
```

このプロパティにはデフォルト値が設定されているため、クラスタ管理者はリソースを作成するときに、新しい値を指定するか、デフォルト値を使用するかを選択します。「`TUNABLE = AT_CREATION`」が指定されているので、作成時以降、この値を変更することはできません。

scds_initialize の呼び出し

DSDL では、各コールバックメソッドがメソッドの開始時に `scds_initialize` (3HA) 関数を呼び出す必要があります。この関数は次の作業を行います。

- フレームワークがデータサービスメソッドに渡すコマンド行引数 (`argc` と `argv[]`) を検査および処理します。そのため、データサービスメソッドは、コマンド行引数について追加の処理を実行する必要はありません。
- 他の DSDL 関数が使用できるように内部データ構造を設定します。
- ロギング環境を初期化します。
- 障害モニターの検証設定の妥当性を検査します。

`scds_close` 関数を使用すると、`scds_initialize` が割り当てたリソースを再利用できます。

xfnts_start メソッド

データサービスリソースを含むリソースグループがオンラインになったとき、あるいは、リソースが有効になったとき、RGM はそのクラスタノード上で `start` メソッドを呼び出します。サンプルの `SUNW.xfnts` リソースタイプでは、`xfnts_start` メソッドが当該ノード上で `xfns` デーモンを起動します。

`xfnts_start` メソッドは `scds_pmf_start` を呼び出して、PMF の制御下でデーモンを起動します。PMF は、自動障害通知、再起動機能、および障害モニターとの統合を提供します。

注 - `xfnts_start` は、`scds_initialize` を最初に呼び出します。これによって、いくつかのハウスキーピング関数が実行されます。詳細については、138 ページの「`scds_initialize` の呼び出し」と `scds_initialize3HA` のマニュアルページを参照してください。

起動前のサービスの検証

次に示すように、`xfnts_start` メソッドは X Font Server を起動する前に `svc_validate` を呼び出して、`xfns` デーモンをサポートするための適切な構成が存在していることを確認します。詳細については、153 ページの「`xfnts_validate` メソッド」を参照してください。

例 8-3

```
rc = svc_validate(scds_handle);
if (rc != 0) {
    scds_syslog(LOG_ERR,
        "Failed to validate configuration.");
    return (rc);
}
```

サービスの起動

`xfnts_start` メソッドは、`xfnts.c` で定義されている `svc_start` メソッドを呼び出して、`xfns` デーモンを起動します。ここでは、`svc_start` について説明します。

以下に、`xfns` デーモンを起動するためのコマンドを示します。

```
xfns -config config_directory/fontserver.cfg -port port_number
```

`Confdir_list` 拡張プロパティには `config_directory` を指定します。一方、`Port_list` システムプロパティには `port_number` を指定します。クラスタ管理者はデータサービスを構成するときに、これらのプロパティの値を指定します。

次に示すように、`xfnts_start` メソッドはこれらのプロパティを文字列配列として宣言し、`scds_get_ext_confdir_list(3HA)` と `scds_get_port_list(3HA)` 関数を使用して、管理者が設定した値を取得します。

例 8-4

```
scha_str_array_t *confdirs;
scds_port_list_t  *portlist;
scha_err_t  err;

/* Confdir_list プロパティから構成ディレクトリを取得する。*/
confdirs = scds_get_ext_confdir_list(scds_handle);

(void) sprintf(xfnts_conf, "%s/fontserver.cfg", confdirs->str_array[0]);

/* Port_list プロパティから XFS が使用するポートを取得する。*/
err = scds_get_port_list(scds_handle, &portlist);
if (err != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Could not access property Port_list.");
    return (1);
}
```

`confdirs` 変数は配列の最初の要素 (0) を指していることに注意してください。

次に示すように、`xfnts_start` メソッドは `sprintf` を使用して、`xfns` 用のコマンド行を形成します。

例 8-5

```
/* xfs デーモンを起動するコマンドを構築する。*/
(void) sprintf(cmd,
    "/usr/openwin/bin/xfs -config %s -port %d 2>/dev/null",
    xfnts_conf, portlist->ports[0].port);
```

出力が `dev/null` にリダイレクトされていることに注意してください。こうすることによって、デーモンが生成するメッセージが抑制されます。

次に示すように、`xfnts_start` メソッドは `xfs` コマンド行を `scds_pmf_start` に渡して、PMF の制御下でデータサービスを起動します。

例 8-6

```
scds_syslog(LOG_INFO, "Issuing a start request.");
err = scds_pmf_start(scds_handle, SCDS_PMF_TYPE_SVC,
    SCDS_PMF_SINGLE_INSTANCE, cmd, -1);

if (err == SCHA_ERR_NOERR) {
    scds_syslog(LOG_INFO,
        "Start command completed successfully.");
} else {
    scds_syslog(LOG_ERR,
        "Failed to start HA-XFS ");
}
```

`scds_pmf_start` を呼び出すときは、次のことに注意してください。

- `SCDS_PMF_TYPE_SVC` パラメータには、データサービスアプリケーションとして起動するプログラムを指定します。このメソッドは他のタイプのアプリケーション (障害モニターなど) も起動できます。
- `SCDS_PMF_SINGLE_INSTANCE` パラメータには、これが単一インスタンスのソースであることを指定します。
- `cmd` パラメータは、すでに生成されているコマンド行を示します。
- 最後のパラメータである `-1` には、子プロセスの監視レベルを指定します。`-1` は、PMF がすべての子プロセスを親プロセスと同様に監視することを示します。

次に示すように、`svc_pmf_start` は `portlist` 構造体に割り当てられているメモリーを解放してから戻ります。

```
scds_free_port_list(portlist);
return (err);
```

svc_start からの復帰

`svc_start` が正常終了したときでも、使用するアプリケーションが起動に失敗した可能性があります。そのため、`svc_start` はアプリケーションを検証して、アプリケーションが動作していることを確認してから、正常終了のメッセージを戻す必要が

あります。このとき、アプリケーションがただちに利用できない理由として、アプリケーションの起動にはある程度時間がかかるということを考慮しておく必要があります。次に示すように、`svc_start` メソッドは `xfnts.c` で定義されている `svc_wait` を呼び出して、アプリケーションが動作していることを確認します。

例 8-7

```
/* サービスが完全に起動するまで待つ。*/
scds_syslog_debug(DBG_LEVEL_HIGH,
    "Calling svc_wait to verify that service has started.");

rc = svc_wait(scds_handle);

scds_syslog_debug(DBG_LEVEL_HIGH,
    "Returned from svc_wait");

if (rc == 0) {
    scds_syslog(LOG_INFO, "Successfully started the service.");
} else {
    scds_syslog(LOG_ERR, "Failed to start the service.");
}
```

次に示すように、`svc_wait` メソッドは `scds_get_netaddr_list(3HA)` を呼び出して、アプリケーションを検証するのに必要なネットワークアドレスリソースを取得します。

例 8-8

```
/* 検証に使用するネットワークリソースを取得する。*/
if (scds_get_netaddr_list(scds_handle, &netaddr)) {
    scds_syslog(LOG_ERR,
        "No network address resources found in resource group.");
    return (1);
}

/* ネットワークリソースが存在しない場合は、エラーを戻す。*/
if (netaddr == NULL || netaddr->num_netaddrs == 0) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    return (1);
}
```

次に示すように、`svc_wait` は `start_timeout` と `stop_timeout` 値を取得します。

例 8-9

```
svc_start_timeout = scds_get_rs_start_timeout(scds_handle)
probe_timeout = scds_get_ext_probe_timeout(scds_handle)
```

サーバーの起動に時間がかかることを考慮して、`svc_wait` は `scds_svc_wait` を呼び出して、`start_timeout` 値の 3% であるタイムアウト値を渡します。次に、`svc_wait` は `svc_probe` を呼び出して、アプリケーションが起動していることを確

認します。svc_probe メソッドは指定されたポート上でサーバーとの単純ソケット接続を確立します。ポートへの接続が失敗した場合、svc_probe は値 100 を戻して、致命的な障害であることを示します。ポートとの接続は確立したが、切断に失敗した場合、svc_probe は値 50 を戻します。

svc_probe が完全にまたは部分的に失敗した場合、svc_wait は scds_svc_wait をタイムアウト値 5 で呼び出します。scds_svc_wait メソッドは、検証の周期を 5 秒ごとに制限します。また、このメソッドはサービスを起動しようとした回数も数えます。この回数がリソースの Retry_interval プロパティで指定された期限内にリソースの Retry_count プロパティの値を超えた場合、scds_svc_wait メソッドは失敗します。この場合、svc_start メソッドも失敗します。

例 8-10

```
#define SVC_CONNECT_TIMEOUT_PCT 95
#define SVC_WAIT_PCT 3
if (scds_svc_wait(scds_handle, (svc_start_timeout * SVC_WAIT_PCT)/100)
    != SCHA_ERR_NOERR) {

    scds_syslog(LOG_ERR, "Service failed to start.");
    return (1);
}

do {
    /*
     * ネットワークリソースのIP アドレスとportname 上で
     * データサービスを検証する。
     */
    rc = svc_probe(scds_handle,
        netaddr->netaddrs[0].hostname,
        netaddr->netaddrs[0].port_proto.port, probe_timeout);
    if (rc == SCHA_ERR_NOERR) {
        /* 成功。リソースを解放して終了。*/
        scds_free_netaddr_list(netaddr);
        return (0);
    }

    /* サービスが何度も失敗する場合は、
     * iscds_svc_wait()を呼び出す。
     */
    if (scds_svc_wait(scds_handle, SVC_WAIT_TIME)
        != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR, "Service failed to start.");
        return (1);
    }

    /* RGM のタイムアウトを待ってプログラムを終了する。*/
} while (1);
```

注 -xfnts_start メソッドは終了する前に scds_close を呼び出して、scds_initialize が割り当てたリソースを再利用します。詳細については、138 138 ページの「scds_initialize の呼び出し」と scds_close(3HA) のマニュアルページを参照してください。

xfnts_stop メソッド

xfnts_start メソッドは scds_pmf_start を使用して PMF の制御下でサービスを起動するので、xfnts_stop は scds_pmf_stop を使用してサービスを停止します。

注 -xfnts_stop は、scds_initialize を最初に呼び出します。これによって、いくつかのハウスキーピング関数が実行されます。詳細については、scds_initialize(3HA) のマニュアルページを参照してください。

次に示すように、xfnts_stop メソッドは、xfnts.c で定義されている svc_stop メソッドを呼び出します。

例 8-11

```
scds_syslog(LOG_ERR, "Issuing a stop request.");
err = scds_pmf_stop(scds_handle,
    SCDS_PMF_TYPE_SVC, SCDS_PMF_SINGLE_INSTANCE, SIGTERM,
    scds_get_rs_stop_timeout(scds_handle));

if (err != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Failed to stop HA-XFS.");
    return (1);
}

scds_syslog(LOG_INFO,
    "Successfully stopped HA-XFS.");
return (SCHA_ERR_NOERR); /* 正常に停止。*/
```

svc_stop から scds_pmf_stop 関数を呼び出すときは、次のことに注意してください。

- SCDS_PMF_TYPE_SVC パラメータには、データサービスアプリケーションとして停止するプログラムを指定します。このメソッドは他のタイプのアプリケーション (障害モニターなど) も停止できます。

- SCDS_PMF_SINGLE_INSTANCE パラメータには、シグナルを指定します。
- SIGTERM パラメータには、リソースインスタンスを停止するのに使用するシグナルを指定します。このシグナルでインスタンスを停止できなかった場合、scds_pmf_stop は SIGKILL を送信してインスタンスを停止しようとします。このシグナルでもインスタンスを停止できなかった場合、タイムアウトエラーで戻ります。詳細については、scds_pmf_stop(3HA) のマニュアルページを参照してください。
- タイムアウト値は、リソースの Stop_timeout プロパティの値を示します。

注 - xfnts_stop メソッドは終了する前に scds_close を呼び出して、scds_initialize が割り当てたリソースを再利用します。詳細については、138 138 ページの「scds_initialize の呼び出し」と scds_close(3HA) のマニュアルページを参照してください。

xfnts_monitor_start メソッド

リソースがノード上で起動した後、RGM はそのノード上で Monitor_start メソッドを呼び出して障害モニターを起動します。xfnts_monitor_start メソッドは scds_pmf_start を使用して PMF の制御下でモニターデーモンを起動します。

注 - xfnts_monitor_start は、scds_initialize を最初に呼び出します。これによって、いくつかのハウスキーピング関数が実行されます。詳細については、138 138 ページの「scds_initialize の呼び出し」と scds_initialize(3HA) のマニュアルページを参照してください。

次に示すように、xfnts_monitor_start メソッドは xfnts.c に定義されている mon_start メソッドを呼び出します

例 8-12

```
scds_syslog_debug(DBG_LEVEL_HIGH,
    "Calling Monitor_start method for resource <%s>.",
    scds_get_resource_name(scds_handle));

/* scds_pmf_start を呼び出し、検証の名前を渡す。*/
err = scds_pmf_start(scds_handle, SCDS_PMF_TYPE_MON,
    SCDS_PMF_SINGLE_INSTANCE, "xfnts_probe", 0);

if (err != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Failed to start fault monitor.");
}
```

例 8-12 (続き)

```
    return (1);
}

scds_syslog(LOG_INFO,
    "Started the fault monitor.");

return (SCHA_ERR_NOERR); /* モニターを正常に起動。*/
}
```

svc_mon_start から scds_pmf_start 関数を呼び出すときは、次のことに注意してください。

- SCDS_PMF_TYPE_MON パラメータには、障害モニターとして起動するプログラムを指定します。このメソッドは他のタイプのアプリケーション(データサービスなど)も起動できます。
- SCDS_PMF_SINGLE_INSTANCE パラメータには、これが単一インスタンスのソースであることを指定します。
- xfnts_probe パラメータには、起動するモニターデーモンを指定します。このモニターデーモンは、他のコールバックプログラムと同じディレクトリに存在するものと想定されます。
- 最後のパラメータである 0 は、子プロセスの監視レベルを指定します。この場合、モニターデーモンだけを監視することを示します。

注 - xfnts_monitor_start メソッドは終了する前に scds_close を呼び出して、scds_initialize が割り当てたりソースを再利用します。詳細については、138 138 ページの「scds_initialize の呼び出し」と scds_close(3HA) のマニュアルページを参照してください。

xfnts_monitor_stop メソッド

xfnts_monitor_start メソッドは scds_pmf_start を使用して PMF の制御下でモニターデーモンを起動するので、xfnts_monitor_stop は scds_pmf_stop を使用してモニターデーモンを停止します。

注 - `xfnts_monitor_stop` は、`scds_initialize` を最初に呼び出します。これによって、いくつかのハウスキーピング関数が実行されます。詳細については、138 138 ページの「`scds_initialize` の呼び出し」と `scds_initialize(3HA)` のマニュアルページを参照してください。

次に示すように、`xfnts_monitor_stop` メソッドは `xfnts.c` で定義されている `mon_stop` メソッドを呼び出します。

例 8-13

```
scds_syslog_debug(DBG_LEVEL_HIGH,
    "Calling scds_pmf_stop method");

err = scds_pmf_stop(scds_handle, SCDS_PMF_TYPE_MON,
    SCDS_PMF_SINGLE_INSTANCE, SIGKILL,
    scds_get_rs_monitor_stop_timeout(scds_handle));

if (err != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Failed to stop fault monitor.");
    return (1);
}

scds_syslog(LOG_INFO,
    "Stopped the fault monitor.");

return (SCHA_ERR_NOERR); /* モニターを正常に停止。 */
}
```

`svc_mon_stop` から `scds_pmf_stop` 関数を呼び出すときは、次のことに注意してください。

- `SCDS_PMF_TYPE_MON` パラメータには、障害モニターとして停止するプログラムを指定します。このメソッドは他のタイプのアプリケーション (データサービスなど) も停止できます。
- `SCDS_PMF_SINGLE_INSTANCE` パラメータには、これが単一インスタンスのリソースであることを指定します。
- `SIGKILL` パラメータには、リソースインスタンスを停止するのに使用するシグナルを指定します。このシグナルでインスタンスを停止できなかった場合、`scds_pmf_stop` はタイムアウトエラーで戻ります。詳細については、`scds_pmf_stop(3HA)` のマニュアルページを参照してください。
- タイムアウト値は、リソースの `Monitor_stop_timeout` プロパティの値を示します。

注 - `xfnts_monitor_stop` メソッドは終了する前に `scds_close` を呼び出して、`scds_initialize` が割り当てたりソースを再利用します。詳細については、138 138 ページの「`scds_initialize` の呼び出し」と `scds_close` 3HA のマニュアルページを参照してください。

xfnts_monitor_check メソッド

障害モニターがリソースが属するリソースグループを別のノードにフェイルオーバーしようとするたびに、RGM は `Monitor_check` メソッドを呼び出します。`xfnts_monitor_check` メソッドは `svc_validate` メソッドを呼び出して、`xfns` デーモンをサポートするための適切な構成が存在していることを確認します。詳細については、153 ページの「`xfnts_validate` メソッド」を参照してください。次に、`xfnts_monitor_check` のコードを示します。

例 8-14

```
/* RGM から渡された引数を処理し、syslog を初期化する。*/
if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
    scds_syslog(LOG_ERR, "Failed to initialize the handle.");
    return (1);
}

rc = svc_validate(scds_handle);
scds_syslog_debug(DBG_LEVEL_HIGH,
    "monitor_check method "
    "was called and returned <%d>.", rc);

/* scds_initialize が割り当てたすべてのメモリーを解放する。*/
scds_close(&scds_handle);

/* モニター検査の一環として実行した検証メソッドの結果を戻す。*/
return (rc);
}
```

SUNW.xfnts 障害モニター

リソースがノード上で起動した後、RGM は、`PROBE` メソッドを直接呼び出すのではなく、`Monitor_start` メソッドを呼び出してモニターを起動します。`xfnts_monitor_start` メソッドは PMF の制御下で障害モニターを起動します。`xfnts_monitor_stop` メソッドは障害モニターを停止します。

SUNW.xfnts 障害モニターは、次の処理を実行します。

- 単純な TCP ベースのサービス (xfs など) を検査するために特別に設計されたユーティリティーを使用して、定期的に xfs サーバーデーモンの状態を監視します。
- (Retry_count と Retry_interval プロパティを使用して) ある期間内にアプリケーションが遭遇した問題を追跡し、アプリケーションが完全に失敗した場合に、データサービスを再起動するか、フェイルオーバーするかどうかを決定します。scds_fm_action と scds_fm_sleep 関数は、この追跡および決定機構の組み込みサポートを提供します。
- scds_fm_action を使用して、フェイルオーバーまたは再起動の決定を実装します。
- リソースの状態を更新して、管理ツールや GUI で利用できるようにします。

xfnts_probe のメインループ

xfnts_probe メソッドは無限ループを実行します。ループを実行する前に、xfnts_probe は次の処理を行います。

- 次に示すように、xfnts リソース用のネットワークアドレスリソースを取得します。

例 8-15

```
/* 当該リソース用に利用できる IP アドレスを取得する。*/
if (scds_get_netaddr_list(scds_handle, &netaddr)) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    scds_close(&scds_handle);
    return (1);
}

/* ネットワークリソースが存在しない場合、エラーを戻す。*/
if (netaddr == NULL || netaddr->num_netaddrs == 0) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    return (1);
}
```

- scds_fm_sleep を呼び出し、タイムアウト値として Thorough_probe_interval の値を渡します。検証を実行する間、検証機能は Thorough_probe_interval で指定された期間、休眠状態になります。

例 8-16

```
timeout = scds_get_ext_probe_timeout(scds_handle);

for (;;) {
    /*
     * 連続する検証の間、Thorough_probe_interval で指定された期間、
```

例 8-16 (続き)

```
* 休眠状態になる。
*/
(void) scds_fm_sleep(scds_handle,
    scds_get_rs_thorough_probe_interval(scds_handle));
```

xfnts_probe メソッドは次のようなループを実装します。

例 8-17

```
for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
    /*
     * 状態を監視するホスト名とポートを取得する。
     */
    hostname = netaddr->netaddrs[ip].hostname;
    port = netaddr->netaddrs[ip].port_proto.port;
    /*
     * HA-XFS がサポートするポートは 1 つだけなので、
     * ポート値はポートの配列の最初の
     * エントリから取得する。
     */
    ht1 = gethrtime(); /* Latch probe start time */
    scds_syslog(LOG_INFO, "Probing the service on port: %d.", port);

    probe_result =
    svc_probe(scds_handle, hostname, port, timeout);

    /*
     * サービス検証履歴を更新し、
     * 必要に応じて、アクションを行う。
     * 検証終了時間を取得する。
     */
    ht2 = gethrtime();

    /* ミリ秒に変換する。*/
    dt = (ulong_t)(ht2 - ht1) / 1e6);

    /*
     * 障害の履歴を計算し、必要に応じて
     * アクションを実行する。
     */
    (void) scds_fm_action(scds_handle,
        probe_result, (long)dt);
} /* ネットワークリソースごと */
} /* 検証を永続的に繰り返す。*/
```

svc_probe 関数は検証ロジックを実装します。svc_probe からの戻り値は scds_fm_action に渡されます。そして scds_fm_action は、アプリケーションを再起動するか、リソースグループをフェイルオーバーするか、あるいは何もしないかを決定します。

svc_probe 関数

svc_probe 関数は、scds_fm_tcp_connect を呼び出すことによって、指定されたポートとの単純ソケット接続を確立します。接続に失敗した場合、svc_probe は 100 の値を戻して、致命的な障害であることを示します。接続には成功したが、切断に失敗した場合、svc_probe は 50 の値を戻して、部分的な障害であることを示します。接続と切断の両方に成功した場合、svc_probe は 0 の値を戻して、成功したことを示します。

次に、svc_probe のコードを示します。

例 8-18

```
int svc_probe(scds_handle_t scds_handle,
char *hostname, int port, int timeout)
{
    int rc;
    hrtime_t t1, t2;
    int sock;
    char testcmd[2048];
    int time_used, time_remaining;
    time_t connect_timeout;

    /*
     * データサービスを検証するには、port_list プロパティに指定された、
     * XFS データサービスを提供するホスト上にあるポートとのソケット接続
     * を確立する。
     * 指定されたポート上でリスンするように構成されたXFS サービスが接続に
     * 応答した場合、検証が成功したと判断する。probe_timeout プロパティに
     * 設定された期間待機しても応答がない場合、検証が失敗したと判断する。
     */

    /*
     * SVC_CONNECT_TIMEOUT_PCT をタイムアウトの
     * 百分率として使用し、ポートと接続する。
     */
    connect_timeout = (SVC_CONNECT_TIMEOUT_PCT * timeout)/100;
    t1 = (hrtime_t)(gethrtime()/1E9);

    /*
     * 検証機能は、指定されたホスト名とポートを接続する。
     * 接続は、probe_timeout 値の 95% に達するとタイムアウトになる。
     */
    rc = scds_fm_tcp_connect(scds_handle, &sock, hostname, port,
        connect_timeout);
    if (rc) {
        scds_syslog(LOG_ERR,
            "Failed to connect to port <%d> of resource <%s>.",
            port, scds_get_resource_name(scds_handle));
        /* 致命的な障害 */
        return (SCDS_PROBE_COMPLETE_FAILURE);
    }
}
```

例 8-18 (続き)

```
t2 = (hrtime_t)(gethrtime()/1E9);

/*
 * 接続にかかる実際の時間を計算する。この値は、
 * 接続に割り当てられた時間を示す connect_timeout 以下
 * である必要がある。接続に割り当てられた時間をすべて
 * 使い切った場合、probe_timeout に残った値が当該関数に
 * 渡され、切断タイムアウトとして使用される。
 * それ以外の場合、接続呼び出しで残った時間が
 * 切断タイムアウトに追加される。
 */

time_used = (int)(t2 - t1);

/*
 * 残った時間(タイムアウトから接続にかかった時間を引いた値)を
 * 切断に使用する。
 */

time_remaining = timeout - (int)time_used;

/*
 * すべての時間を使い切った場合、ハードコーディングされた小さな
 * タイムアウト値を使用して、切断しようとする。
 * これによって、fd リークを防ぐ。
 */
if (time_remaining <= 0) {
    scds_syslog_debug(DBG_LEVEL_LOW,
        "svc_probe used entire timeout of "
        "%d seconds during connect operation and exceeded the "
        "timeout by %d seconds. Attempting disconnect with timeout"
        " %d ",
        connect_timeout,
        abs(time_used),
        SVC_DISCONNECT_TIMEOUT_SECONDS);

    time_remaining = SVC_DISCONNECT_TIMEOUT_SECONDS;
}

/*
 * 切断に失敗した場合、部分的な障害を戻す。
 * 理由: 接続呼び出しは成功した。これは、
 * アプリケーションが正常に動作していることを意味する。
 * 切断が失敗した原因は、アプリケーションがハングしたか、
 * 負荷が高いためである。
 * 後者の場合、アプリケーションが停止したとは宣言しない
 * (つまり、致命的な障害を戻さない)。その代わりに、部分的な
 * 障害であると宣言する。この状態が続く場合、切断呼び出しは
 * 再び失敗し、アプリケーションは再起動される。
 */
rc = scds_fm_tcp_disconnect(scds_handle, sock, time_remaining);
if (rc != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
```

例 8-18 (続き)

```
        "Failed to disconnect to port %d of resource %s.",
        port, scds_get_resource_name(scds_handle));
/* 部分的な障害 */
return (SCDS_PROBE_COMPLETE_FAILURE/2);
}

t2 = (hrtime_t)(gethrtime()/1E9);
time_used = (int)(t2 - t1);
time_remaining = timeout - time_used;

/*
 * 時間が残っていない場合、fsinfo による完全な
 * テストを行わない。その代わりに、
 * SCDS_PROBE_COMPLETE_FAILURE/2 を戻す。これによって、
 * このタイムアウトが続く場合、サーバーは再起動される。
 */
if (time_remaining <= 0) {
    scds_syslog(LOG_ERR, "Probe timed out.");
    return (SCDS_PROBE_COMPLETE_FAILURE/2);
}

/*
 * ポートとの接続と切断に成功した。
 * fsinfo コマンドを実行して、
 * サーバーの状態を完全に検査する。
 * stdout をリダイレクトする。そうでない場合、
 * fsinfo からの出力はコンソールに送られる。
 */
(void) sprintf(testcmd,
    "/usr/openwin/bin/fsinfo -server %s:%d> /dev/null",
    hostname, port);
scds_syslog_debug(DBG_LEVEL_HIGH,
    "Checking the server status with %s.", testcmd);
if (scds_timerun(scds_handle, testcmd, time_remaining,
    SIGKILL, &rc) != SCHA_ERR_NOERR || rc != 0) {

    scds_syslog(LOG_ERR,
        "Failed to check server status with command <%s>",
        testcmd);
    return (SCDS_PROBE_COMPLETE_FAILURE/2);
}
return (0);
}
```

svc_probe は終了時に、成功 (0)、部分的な障害 (50)、または致命的な障害 (100) を戻します。xfnts_probe メソッドはこの値を scds_fm_action に渡します。

障害モニターのアクションの決定

`xfnts_probe` メソッドは `scds_fm_action` を呼び出して、行うべきアクションを決定します。`scds_fm_action` のロジックは次のとおりです。

- `Retry_interval` プロパティで指定された期間中に、障害の履歴を累積します。
- 累積した障害が 100 に到達した場合 (致命的な障害)、データサービスを再起動します。`Retry_interval` を超えた場合、障害の履歴をリセットします。
- `Retry_interval` で指定された期間中に、再起動の回数が `Retry_count` プロパティを上回った場合、データサービスをフェイルオーバーします。

たとえば、検証機能が `xfns` サーバーに正常に接続したが、切断に失敗したものと想定します。これは、サーバーは動作しているが、ハングしていたり、一時的に過負荷状態になっている可能性を示しています。切断に失敗すると、`scds_fm_action` に部分的な障害 (50) が送信されます。この値は、データサービスを再起動するしきい値を下回っていますが、値は障害の履歴に記録されます。

次の検証でもサーバーが切断に失敗した場合、`scds_fm_action` が保持している障害の履歴に値 50 が再度追加されます。累積した障害の履歴が 100 になるので、`scds_fm_action` はデータサービスを再起動します。

`xfnts_validate` メソッド

リソースが作成されたとき、および、リソースまたは (リソースを含む) リソースグループのプロパティが管理アクションによって更新されるとき、RGM は `Validate` メソッドを呼び出します。RGM は、作成または更新が行われる前に、`Validate` メソッドを呼び出します。任意のノード上でメソッドから失敗の終了コードが戻ると、作成または更新は取り消されます。

RGM が `Validate` メソッドを呼び出すのは、管理アクションがリソースまたはグループのプロパティを変更したときだけです。RGM がプロパティを設定したときや、モニターがリソースプロパティ `Status` と `Status_msg` を設定したときではありません。

注 - `PROBE` メソッドがデータサービスを新しいノードにフェイルオーバーする場合、`Monitor_check` メソッドも明示的に `Validate` メソッドを呼び出します。

RGM は、他のメソッドに渡す追加の引数 (更新されるプロパティと値を含む) を指定して、`Validate` メソッドを呼び出します。`xfnts_validate` の開始時に実行される `scds_initialize` の呼び出しにより、RGM が `xfnts_validate` に渡したすべての引数が解析され、その情報が `scds_handle` パラメータに格納されます。この情報は、`xfnts_validate` が呼び出すサブルーチンによって使用されます。

xfnts_validate メソッドは svc_validate を呼び出して、次のことを検証します。

- Confdir_list プロパティがリソース用に設定されており、単一のディレクトリが定義されているかどうか。

例 8-19

```
scha_str_array_t *confdirs;
confdirs = scds_get_ext_confdir_list(scds_handle);

/* Confdir_list 拡張プロパティが存在しない場合、エラーを戻す。*/
if (confdirs == NULL || confdirs->array_cnt != 1) {
    scds_syslog(LOG_ERR,
        "Property Confdir_list is not set properly.");
    return (1); /* 検証に失敗 */
}
```

- Confdir_list で指定されたディレクトリに fontserver.cfg ファイルが存在しているかどうか。

例 8-20

```
(void) sprintf(xfnts_conf, "%s/fontserver.cfg", confdirs->str_array[0]);

if (stat(xfnts_conf, &statbuf) != 0) {
    /*
     * errno.h プロトタイプには void 引数がないので、
     * lint エラーが抑制される。
     */
    scds_syslog(LOG_ERR,
        "Failed to access file <s> : <s>",
        xfnts_conf, strerror(errno)); /*lint !e746 */
    return (1);
}
```

- サーバーデーモンバイナリがクラスタノード上でアクセスできるかどうか。

例 8-21

```
if (stat("/usr/openwin/bin/xfns", &statbuf) != 0) {
    scds_syslog(LOG_ERR,
        "Cannot access XFS binary : <s> ", strerror(errno));
    return (1);
}
```

- Port_list プロパティが単一のポートを指定しているかどうか。

例 8-22

```
scds_port_list_t *portlist;
err = scds_get_port_list(scds_handle, &portlist);
if (err != SCHA_ERR_NOERR) {
```

例 8-22 (続き)

```
        scds_syslog(LOG_ERR,
            "Could not access property Port_list: %s.",
            scds_error_string(err));
        return (1); /* 検証に失敗 */
    }

#ifdef TEST
    if (portlist->num_ports != 1) {
        scds_syslog(LOG_ERR,
            "Property Port_list must have only one value.");
        scds_free_port_list(portlist);
        return (1); /* 検証に失敗 */
    }
#endif
```

- データサービスが属するリソースグループに、少なくとも1つのネットワークアドレスリソースが属しているかどうか。

例 8-23

```
scds_net_resource_list_t *snrlp;
if ((err = scds_get_rs_hostnames(scds_handle, &snrlp))
    != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group: %s.",
        scds_error_string(err));
    return (1); /* 検証に失敗 */
}

/* ネットワークアドレスリソースが存在しない場合エラーを戻す。*/
if (snrlp == NULL || snrlp->num_netresources == 0) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    rc = 1;
    goto finished;
}
```

次に示すように、`svc_validate` は戻る前に、割り当てられているすべてのリソースを解放します。

例 8-24

```
finished:
    scds_free_net_list(snrlp);
    scds_free_port_list(portlist);

    return (rc); /* 検証結果を戻す。*/
```

注 - `xfnts_validate` メソッドは終了する前に `scds_close` を呼び出して、`scds_initialize` が割り当てたリソースを再利用します。詳細については、138 138 ページの「`scds_initialize` の呼び出し」と `scds_close(3HA)` のマニュアル ページを参照してください。

xfnts_update メソッド

プロパティが変更された場合、RGM は Update メソッドを呼び出して、そのことを動作中のリソースに通知します。xfnts データサービスにおいて変更可能なプロパティは、障害モニターに関連したものだけです。したがって、プロパティが更新されたとき、`xfnts_update` メソッドは `scds_pmf_restart_fm` を呼び出して、障害モニターを再起動します。

例 8-25

```
/*
 * 障害モニターがすでに動作していることを検査し、動作している場合、
 * 障害モニターを停止および再起動する。scds_pmf_restart_fm() への
 * 2 番目のパラメータは、再起動する必要がある障害モニターの
 * インスタンスを一意に識別する。
 */

scds_syslog(LOG_INFO, "Restarting the fault monitor.");
result = scds_pmf_restart_fm(scds_handle, 0);
if (result != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Failed to restart fault monitor.");
    /* scds_initialize が割り当てたすべてのメモリーを解放する。*
    scds_close(&scds_handle);
    return (1);
}

scds_syslog(LOG_INFO,
    "Completed successfully.");
```

注 - `scds_pmf_restart_fm` への 2 番目のパラメータは、複数のインスタンスが存在する場合に、再起動する障害モニターのインスタンスを一意に識別します。この例の場合、値 0 は障害モニターのインスタンスが 1 つしか存在しないことを示します。

第 9 章

SunPlex Agent Builder

この章では、Resource Group Manager (RGM) の管理下で動作するリソースタイプ (データサービス) の作成を自動化するツール、SunPlex™ Agent Builder について説明します。リソースタイプとは、アプリケーションが RGM の制御下にあるクラスタ環境で動作できるようにするアプリケーションのラッパーのことです。

Agent Builder は、アプリケーションや作成したいリソースタイプの種類に関する簡単な情報を入力するための画面ベースのインタフェースを提供します。Agent Builder は、入力された情報に基づいて、次のソフトウェアを生成します。

- フェイルオーバーリソースタイプまたはスケーラブルリソースタイプのメソッドコールバックに対応する複数のソースファイル。C、Korn シェル (ksh)、汎用データサービス (Generic Data Service: GDS) など。
- C シェルまたは Korn シェルのソースを生成する場合は、カスタマイズされたリソースタイプ登録 (Resource Type Registration: RTR) ファイル。
- リソースタイプのインスタンス (リソース) を起動、停止、および削除するためのカスタマイズされたユーティリティスクリプト。また、これらのファイルの使用方法を説明するカスタマイズされたマニュアルページ。
- C のソースを生成する場合はバイナリを含む Solaris パッケージとユーティリティスクリプト。C または Korn シェルのソースを生成する場合は RTR ファイルを含む Solaris パッケージとユーティリティスクリプト。

Agent Builder は、クライアントとの通信にネットワークを使用するネットワーク対応アプリケーションと、非ネットワーク対応のスタンドアロンアプリケーションをサポートします。Agent Builder を使って、プロセス監視機能 (PMF) によって個別に監視および再起動される複数の独立したプロセスツリーを持つアプリケーション用のリソースタイプを生成できます。166 ページの「複数の独立したプロセスツリーを持つリソースタイプの作成」を参照してください。

Agent Builder の使用

この節では、Agent Builder の使用方法と、Agent Builder を使用する前に実行する作業について説明します。リソースタイプコードを生成した後で Agent Builder を活用する方法についても説明します。

アプリケーションの分析

Agent Builder を使用する前に、アプリケーションが高可用性またはスケーラビリティを備えるための要件を満たしているかどうかを判定します。この分析はアプリケーションの実行時特性だけに基づくものなので、Agent Builder はこの分析を行うことができません。詳細については、27 ページの「アプリケーションの適合性の分析」を参照してください。

Agent Builder は必ずしもアプリケーション用の完全なリソースタイプを作成できるわけではありませんが、ほとんどの場合、少なくとも部分的なソリューションを提供します。たとえば、より複雑なアプリケーションでは、Agent Builder がデフォルトで生成しないコード (プロパティの妥当性検査を追加するコードや Agent Builder がエクスポートしないパラメータを調節するコード) を追加しなければならない場合があります。このような場合、生成されたコードまたは RTR ファイルを修正する必要があります。Agent Builder は、このような柔軟性を提供するように設計されています。

Agent Builder は、ソースファイル内において独自のリソースタイプコードを追加できる場所にコメント文を埋め込みます。ソースコードを修正した後、Agent Builder が生成した Makefile を使用すれば、ソースコードを再コンパイルし、リソースタイプパッケージを生成し直すことができます。

Agent Builder が生成したリソースタイプコードを使用せずに、リソースタイプコードを完全に作成し直す場合でも、Agent Builder が生成した Makefile やディレクトリ構造を使用すれば、独自のリソースタイプ用の Solaris パッケージを作成できます。

Agent Builder のインストールと構成

Agent Builder を個別にインストールする必要はありません。Agent Builder は、Sun Cluster ソフトウェアの標準インストールでデフォルトでインストールされる SUNWscdev パッケージに含まれています。『*Sun Cluster 3.1 Software Installation Guide*』を参照してください。Agent Builder を使用する前に、次のことを確認してください。

- `$PATH` 変数に Java が指定されていること。Agent Builder は Java (Java Development Kit バージョン 1.3.1 以上) に依存しています。`$PATH` 変数に Java が指定されていない場合、`scdsbuilder` はエラーメッセージを戻します。
- Solaris 8 以降の「Developer System Support」ソフトウェアグループがインストールされていること。

- \$PATH 変数に cc コンパイラが指定されていること。Agent Builder は、\$PATH 変数内の最初の cc を、リソースタイプの C バイナリコードを生成するコンパイラとします。cc が \$PATH に存在しない場合、Agent Builder は C コードを生成するオプションを無効にします。詳細については、161 ページの「作成画面の使用」を参照してください。

注 – Agent Builder では、標準の cc コンパイラ以外のコンパイラも使用できます。このためには、\$PATH において、cc から別のコンパイラ (gcc など) にシンボリックリンクを作成します。もう 1 つの方法は、Makefile におけるコンパイラ指定を変更して (現在は、CC=cc)、別のコンパイラへの完全パスを指定します。たとえば、Agent Builder が生成する Makefile において、CC=cc を CC=*pathname*/gcc に変更します。この場合、Agent Builder を直接実行することはできません。代わりに、make や make pkg コマンドを使用して、データサービスコードとパッケージを生成する必要があります。

Agent Builder の起動

Agent Builder を起動するには、scdsbuilder(1HA) コマンド を使用します。

```
% /usr/cluster/bin/scdsbuilder
```

次の図は、Agent Builder の初期画面です。



図 9-1 初期画面

注 - GUI バージョンにアクセスできない場合は、コマンド行インタフェースバージョンの Agent Builder を使用できます。詳細については、168 ページの「コマンド行バージョンの Agent Builder の使用」を参照してください。

Agent Builder では、次の 2 つの画面を使用して、新しいリソースタイプを作成します。

1. 作成 — この画面では、リソースタイプ名、リソースタイプテンプレートを作成および構成する作業ディレクトリなどの基本情報を指定して、生成されたファイル用のリソースタイプを作成します。作成するリソースの種類 (スケラブルまたはフェイルオーバー)、ベースアプリケーションがネットワーク対応かどうか (つまり、ネットワークを使用してクライアントと通信するかどうか)、生成するコードのタイプ (C、ksh または GDS) も指定できます。GDS の詳細については、第 10 章を参照してください。この画面に必要な情報をすべて入力し、「作成」を選択すると、対応する出力が生成されます。その後、構成画面に進みます。
2. 構成 — この画面では、アプリケーションを起動するコマンドを提供する必要があります。オプションとして、アプリケーションを停止するコマンドや検証するコマンドも提供できます。これらのコマンドを指定しない場合、生成される出力コードは、シグナルを使用してアプリケーションを停止し、デフォルトの検証メカニズム

を使用してアプリケーションを検証します。検証コマンドについては、163 ページの「構成画面の使用」を参照してください。また、この画面では、上記の各コマンドのタイムアウト値も変更できます。

注 - 既存のリソースタイプの作業ディレクトリから起動する場合、Agent Builder は、作成および構成画面を既存のリソースタイプの値に初期化します。

Agent Builder の画面上のボタンやメニューの使用方法については、174 ページの「ナビゲーション」を参照してください。

作成画面の使用

リソースタイプを作成する最初の段階では、Agent Builder を起動したときに表示される 作成画面に必要な情報を入力します。すると、次の画面が表示されます。

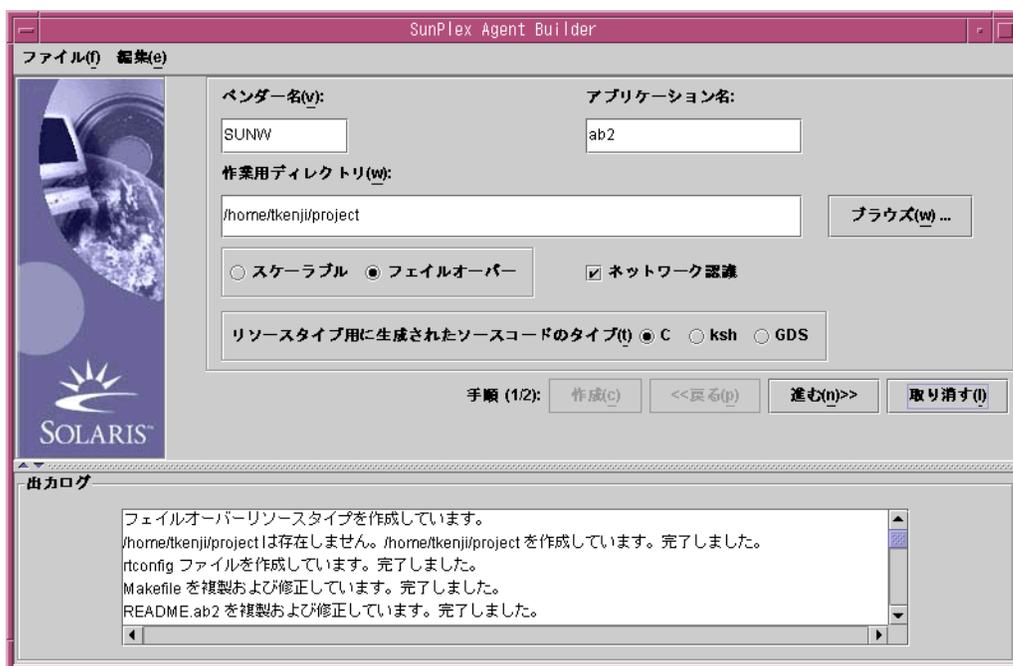


図 9-2 作成画面

作成画面には、次のフィールド、ラジオボタン、およびチェックボックスがあります。

- ベンダー名 — リソースタイプのベンダーを識別する名前。通常、ベンダーの略号を指定します。ベンダーを一意に識別する名前であれば、どのような名前でも有効です。英数字文字だけを使用します。
- アプリケーション名 — リソースタイプ名。英数字文字だけを使用します。

注 — ベンダー名とアプリケーション名の両方で、リソースタイプの完全な名前が形成されます。完全な名前は9文字を超えてはなりません。

- 作業用ディレクトリ — Agent Builder は、このディレクトリの下に、ターゲットリソースタイプ用のすべてのファイルを格納するディレクトリ構造を作成します。1つの作業ディレクトリには1つのリソースタイプしか作成できません。Agent Builder は、このフィールドを Agent Builder が起動されたディレクトリのパスで初期化します。ただし、別のディレクトリ名を入力したり、「ブラウズ」ボタンを使用して異なるディレクトリを指定することもできます。

Agent Builder は、作業ディレクトリの下にリソースタイプ名を持つサブディレクトリを作成します。たとえば、ベンダー名が SUNW で、アプリケーション名が ftp である場合、Agent Builder はこのサブディレクトリに SUNWftp という名前を付けます。

Agent Builder は、ターゲットリソースタイプのすべてのディレクトリとファイルをこのサブディレクトリの下に置きます(169 ページの「ディレクトリ構造」を参照)。

- スケーラブルまたはフェイルオーバー — ターゲットリソースタイプがスケーラブルまたはフェイルオーバーのどちらであるかを指定します。
- ネットワーク認識 — ベースアプリケーションがネットワーク対応かどうかを指定します。つまり、アプリケーションがネットワークを使用してクライアントと通信するかどうかを指定します。ネットワーク対応であれば、チェックボックスにチェックマークを入れます。非ネットワーク対応であれば、チェックボックスをそのままにします。Korn シェルコードの場合、アプリケーションはネットワーク対応でなければなりません。したがって、「ksh」または「GDS」チェックボックスにチェックマークを入れた場合、このチェックボックスは自動的に無効になります。
- C, ksh — 生成されるソースコードの言語を指定します。このオプションは、どちらか一方しか指定できません。ただし、Agent Builder を使用すれば、ksh 用に生成されたコードでリソースタイプを作成した後、同じ情報を再利用して C 言語のコードを作成できます。詳細については、167 ページの「既存のリソースタイプのクローンの作成」を参照してください。
- GDS — このサービスが汎用データサービスであることを示します。汎用データサービスの詳しい作成および構成方法については、第 10 章を参照してください。

注 - cc コンパイラが \$PATH に存在しない場合、「C」オプションボタンは無効になり、「ksh」ボタンにチェックマークが入ります。異なるコンパイラを指定する方法については、158 ページの「Agent Builder のインストールと構成」の最後にある注を参照してください。

必要な情報を入力した後、「作成」ボタンをクリックします。画面の一番下にある「出力ログ」には、Agent Builder が行なったアクションが表示されます。「編集」メニューの「出力ログを保存」コマンドを使用すれば、出力ログ内の情報を保存できます。

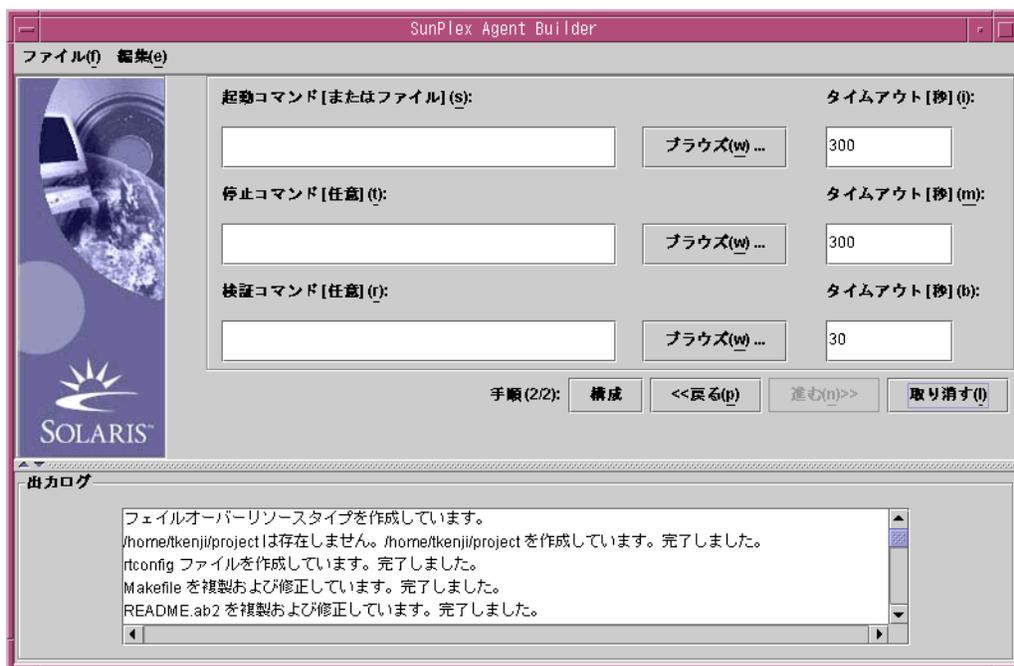
終了したなら、Agent Builder は成功メッセージまたは警告メッセージを表示します。警告メッセージは作成段階が完了しなかったことを示します。その場合は、出力ログの情報から原因を調べます。

Agent Builder が成功メッセージを表示した場合は、「進む」ボタンをクリックして「構成」画面に進むことができます。「構成」画面では、リソースタイプの生成を完結することができます。

注 - 完全なリソースタイプを生成するには、2 段階の作業が必要ですが、最初の段階（つまり、作成）が完了した後に Agent Builder を終了しても、入力した情報や Agent Builder で作成した内容が失われることはありません。詳細については、167 ページの「完成した作業内容の再利用」を参照してください。

構成画面の使用

次の構成画面は、Agent Builder がリソースの作成処理を完了し、ユーザーが作成画面の「進む」ボタンをクリックすると表示されます。リソースタイプの作成が完了していなければ、構成画面にはアクセスできません。



構成画面には、次のフィールドがあります。

- 起動コマンド — ベースアプリケーションを起動するために任意の UNIX シェルに渡すことができる完全なコマンド行。このコマンドは必ず指定する必要があります。このフィールドにコマンドを入力するか、「ブラウズ」ボタンを使用して、アプリケーションを起動するコマンドが記述されているファイルを指定します。

完全なコマンド行には、アプリケーションを起動するのに必要なすべての要素が含まれていなければなりません。たとえば、ホスト名、ポート番号、構成ファイルへのパスなどです。コマンド行にホスト名を指定する必要があるアプリケーションの場合、Agent Builder が定義する \$hostnames 変数を使用できます。詳細については、166 ページの「Agent Builder の \$hostnames 変数の使用」を参照してください。

コマンドは二重引用符 (") で囲むではありません。

注 — ベースアプリケーションが複数の独立したプロセスツリーを持ち、各プロセスツリーが PMF の制御下で独自のタグによって起動される場合、単一のコマンドは指定できません。代わりに、各プロセスツリーを起動するための個々のコマンドを記述したテキストファイルを作成し、そのファイルへのパスを「起動コマンド」テキストフィールドに指定する必要があります。このファイルが適切に機能するために必要な特性については、166 ページの「複数の独立したプロセスツリーを持つリソースタイプの作成」を参照してください。

- 停止コマンド — ベースアプリケーションを停止するために任意の UNIX シェルに渡すことができる完全なコマンド行。このフィールドにコマンドを入力するか、「ブラウズ」ボタンを使用して、アプリケーションを停止するコマンドが記述されているファイルを指定します。コマンド行にホスト名を指定する必要があるアプリケーションの場合、Agent Builder が定義する \$hostnames 変数を使用できます。詳細については、166 ページの「Agent Builder の \$hostnames 変数の使用」を参照してください。

このコマンドは省略可能です。停止コマンドを指定しない場合、生成されるコードは、次に示すように、Stop メソッドでシグナルを使用して、アプリケーションを停止します。

- Stop メソッドは SIGTERM を送信してアプリケーションを停止しようとします。そして、アプリケーション用のタイムアウト値の 80% だけ待機して、停止しない場合は終了します。
- SIGTERM シグナルが失敗した場合、Stop メソッドは SIGKILL を送信して、アプリケーションを停止しようとします。そして、アプリケーション用のタイムアウト値の 15% だけ待機して、停止しない場合は終了します。
- SIGKILL シグナルが失敗した場合、Stop メソッドは失敗します。タイムアウト値の残りの 5% はオーバーヘッドとなります。



注意 – 停止コマンドは、アプリケーションが完全に停止するまで戻らないことに注意してください。

- 検証コマンド — 定期的に実行され、アプリケーションの状態を検査して、0 (正常) から 100 (致命的な障害) の範囲の終了状態に戻すコマンド。このコマンドは省略可能です。このフィールドにコマンドの完全パスを入力するか、「ブラウズ」ボタンを使用して、アプリケーションを検証するコマンドが記述されているファイルを指定します。

通常は、単にベースアプリケーションのクライアントを指定します。検証コマンドを指定しない場合、生成されるコードは、リソースが使用するポートへの接続と切断を試みます。接続と切断に成功すれば、アプリケーションの状態が正常であると判断します。検証コマンドを使用できるのはネットワーク対応アプリケーションだけです。Agent Builder は常に検証コマンドを生成しますが、非ネットワーク対応アプリケーションでは検証コマンドを無効にします。

検証コマンド行にホスト名を指定する必要があるアプリケーションでは、Agent Builder が定義する \$hostnames 変数を使用できます。詳細については、166 ページの「Agent Builder の \$hostnames 変数の使用」を参照してください。

- タイムアウト (コマンドごと) — 各コマンドのタイムアウト値 (秒単位)。新しい値を指定するか、Agent Builder が提供するデフォルト値を受け入れます。起動コマンドと停止コマンドのデフォルト値は 300 秒です。検証コマンドのデフォルト値は 30 秒です。

Agent Builder の \$hostnames 変数の使用

多くのアプリケーション (特に、ネットワーク対応アプリケーション) では、アプリケーションが通信し、顧客の要求に対してサービスを提供するホスト名をコマンド行に指定して、アプリケーションに渡す必要があります。そのため、多くの場合、ホスト名は、構成画面において、ターゲットリソースタイプの起動、停止、および検証コマンドに指定する必要があるパラメータです。ただし、アプリケーションが通信するホスト名はクラスタに固有です。つまり、ホスト名は、リソースがクラスタ上で動作するときに決定され、Agent Builder でリソースタイプコードを生成するときに決定することはできません。

この問題を解決するために、Agent Builder は \$hostnames 変数を提供します。この変数を使用すると、起動、停止、および検証コマンドのコマンド行にホスト名を指定できます。\$hostnames 変数を指定する方法は、実際のホスト名を指定する方法と同じです。たとえば、次のようになります。

```
/opt/network_aware/echo_server -p port_no -l $hostnames
```

ターゲットリソースタイプのリソースがあるクラスタ上で動作するとき、(リソースの Network_resources_used リソースプロパティで) そのリソースに構成されている LogicalHostname または SharedAddress ホスト名が \$hostnames 変数の値に置き換えられます。

Network_resources_used プロパティに複数のホスト名を構成している場合、すべてのホスト名をコマンドで区切って \$hostnames 変数に指定します。

複数の独立したプロセスツリーを持つリソースタイプの作成

Agent Builder は、複数の独立したプロセスツリーを持つアプリケーション用のリソースタイプを作成できます。プロセスツリーが独立しているということは、PMF が各プロセスツリーを個別に監視および起動することを意味しています。PMF は独自のタグを使用して各プロセスツリーを起動します。

複数の独立したプロセスツリーを持つベースアプリケーションの場合、1つのコマンド行だけでアプリケーションを起動することはできません。代わりに、アプリケーションの各プロセスツリーを起動するコマンドへの完全パスを行ごとに記述したテキストファイルを作成します。このファイルには空白行を含めることはできません。そして、このファイルへのパスを構成画面の「起動コマンド」テキストフィールドに指定します。

また、このテキストファイルに実行権が設定されていないことを確認する必要があります。これにより、Agent Builder は、このファイルが複数のプロセスツリーを起動するためのものであり、単に複数のコマンドが記述されている実行可能スクリプトではないことを認識できます。このテキストファイルに実行権を与えた場合、リソースはクラスタ上で動作するように見えますが、すべてのコマンドが1つのPMFタグ下で起動されるため、PMF は各プロセスツリーを個別に監視および再起動することはできません。

完成した作業内容の再利用

Agent Builder を使用すると、次に示すように、完成した作業内容を再利用できます。

- Agent Builder で作成した既存のリソースタイプのクローンを作成できます。
- Agent Builder が生成したソースコードを編集して、そのコードを再コンパイルすれば、新しいパッケージを作成できます。

既存のリソースタイプのクローンの作成

Agent Builder で作成した既存のリソースタイプのクローンを作成するには、次の手順に従います。

1. 既存のリソースタイプを **Agent Builder** にロードします。これは、次の 2 つの方法で行えます。
 - a. (**Agent Builder** で作成した) 既存のリソースタイプ用の作業ディレクトリ (**rtconfig** ファイルが格納されているディレクトリ) から **Agent Builder** を起動します。すると、**Agent Builder** の作成画面と構成画面に、そのリソースタイプ用の値がロードされます。
 - b. 「ファイル」メニューの「リソースタイプをロード」コマンドを使用します。
2. 作成画面で作業ディレクトリを変更します。

ディレクトリを選択するときは、「ブラウズ」ボタンを使用する必要があります。つまり、新しいディレクトリ名を入力するだけでは十分ではありません。ディレクトリを選択した後、Agent Builder は「作成」ボタンを有効に戻します。
3. 必要な変更を行います。

この手順は、リソースタイプ用に生成されたコードのタイプを変更するときに使用できます。たとえば、最初は ksh バージョンのリソースタイプを作成していたが、後で C バージョンのリソースタイプが必要になった場合などです。この場合、既存の ksh バージョンのリソースタイプをロードし、出力用の言語を C に変更すると、Agent Builder は C バージョンのリソースタイプを構築します。
4. リソースタイプのクローンを作成します。

「作成」を選択して、リソースタイプを作成します。「進む」を選択して、構成画面に進みます。「構成」を選択して、リソースタイプを構成します。最後に、「取り消す」を押して、終了します。

生成されたソースコードの編集

リソースタイプを作成するプロセスを簡単にするために、Agent Builder は入力数を制限しています。必然的に、生成されるリソースタイプの範囲も制限されます。したがって、より複雑な機能、たとえば、追加のプロパティの妥当性を検査したり、Agent Builder がエクスポートしないパラメータを調整したりする機能を追加するには、生成されたソースコードまたは RTR ファイルを修正する必要があります。

ソースファイルは *install_directory/rt_name/src* ディレクトリに置かれます。Agent Builder は、ソースコード内においてコードを追加できる場所にコメント文を埋め込みます。このようなコメントの形式は次のとおりです (C コードの場合)。

```
/* User added code -- BEGIN vvvvvvvvvvvvvvvvvv */  
/* User added code -- END   ^^^^^^^^^^^^^^^^^^^ */
```

注 – これらのコメント文は Korn シェルコードの場合も同じです。ただし、コメント行の先頭にはシャープ記号 (#) が使用されます。

たとえば、*rt_name.h* は、異なるプログラムが使用するすべてのユーティリティールーチンを宣言します。宣言リストの最後はコメント文になっており、ここでは自分のコードに追加したいルーチンを宣言できます。

install_directory/rt_name/src ディレクトリには、適切なターゲットとともに、Makefile も生成されます。make コマンドを使用すると、ソースコードを再コンパイルできます。また、make pkg コマンドを使用すると、リソースタイプパッケージを生成し直すことができます。

RTR ファイルは *install_directory/rt_name/etc* ディレクトリに置かれます。RTR ファイルは標準のテキストエディタで編集できます。RTR ファイルの詳細については、32 ページの「リソースとリソースタイププロパティの設定」を参照してください。プロパティの詳細については、付録 A を参照してください。

コマンド行バージョンの Agent Builder の使用

コマンド行バージョンの Agent Builder では、グラフィカルユーザーインターフェースと同様に、2 段階の入力手順が必要です。ただし、グラフィカルユーザーインターフェースに情報を入力する代わりに、*scdscreate 1HA* および *scdsconfig 1HA* コマンドにパラメータを渡します。

コマンド行バージョンの Agent Builder の使用方法は次のとおりです。

1. アプリケーションに高可用性またはスケーラビリティを持たせるため、**scdscreate** を使って **Sun Cluster** リソースタイプテンプレートを作成します。
2. **scdsconfigure** を使って、**scdscreate** で作成したリソースタイプテンプレートを構成します。

3. 作業ディレクトリの **pkg** サブディレクトリに移動します。
4. **pkgadd(1M)** コマンドを使って、**scdscreate** で作成したパッケージをインストールします。
5. 必要に応じて、生成されたソースコードを編集します。
6. 起動スクリプトを実行します。

ディレクトリ構造

Agent Builder は、ターゲットリソースタイプ用に生成するすべてのファイルを格納するためのディレクトリ構造を作成します。作業ディレクトリは作成画面で指定します。開発するリソースタイプごとに異なるインストールディレクトリを指定する必要があります。Agent Builder は、作業ディレクトリの下に、作成画面で入力されたベンダー名とリソースタイプ名を連結した名前を持つサブディレクトリを作成します。たとえば、SUNW というベンダー名を指定し、ftp というリソースタイプを作成した場合、Agent Builder は SUNWftp というディレクトリを作業ディレクトリの下に作成します。

Agent Builder は、このサブディレクトリの下に、次のようなディレクトリを作成し、各ディレクトリにファイルを配置します。

表 9-1

ディレクトリ名	内容
bin	C 出力の場合、ソースファイルからコンパイルしたバイナリファイルが格納されます。ksh の場合、src ディレクトリと同じファイルが格納されます。
etc	RTR ファイルが格納されます。Agent Builder はベンダー名とアプリケーション名をピリオド (.) で区切って連結し、RTR ファイル名を形成します。たとえば、ベンダー名が SUNW で、リソースタイプ名が ftp である場合、RTR ファイル名は SUNW.ftp となります。
man	start、stop、および remove ユーティリティースクリプト用にカスタマイズされたマニュアルページ (man1m) が格納されます。たとえば、startftp(1M)、stopftp(1M)、および removeftp(1M) が格納されます。 これらのマニュアルページを表示するには、man-M オプションでパスを指定します。次に例を示します。 <code>man -M install_directory/SUNWftp/man removeftp .</code>
pkg	最終的なパッケージが格納されます。

表 9-1 (続き)

ディレクトリ名	内容
src	Agent Builder によって生成されたソースファイルが格納されます。
util	Agent Builder によって生成された start、stop、および remove ユーティリティスクリプトが格納されます。171 ページの「ユーティリティスクリプトとマニュアルページ」を参照してください。Agent Builder は、これらのスクリプト名にアプリケーション名を追加します。たとえば、startftp、stopftp、および removeftp のようになります。

出力

この節では、Agent Builder の出力について説明します。

ソースファイルとバイナリファイル

リソースグループと、最終的にはクラスタ上のリソースを管理する Resource Group Manager (RGM) は、コールバックモデル上で動作します。つまり、特定のイベント（ノードの障害など）が発生したとき、RGM は、当該ノード上で動作しているリソースごとにリソースタイプのメソッドを呼び出します。たとえば、RGM は stop メソッドを呼び出して、当該ノード上で動作しているリソースを停止します。次に、stop メソッドを呼び出して、異なるノード上でリソースを起動します。このモデルの詳細については、19 ページの「RGM モデル」、21 ページの「コールバックメソッド」、および `rt_callbacks(1HA)` のマニュアルページを参照してください。

このモデルをサポートするため、Agent Builder は、`install_directory/rt_name/bin` ディレクトリに、コールバックメソッドとして機能する 8 つの実行可能プログラム (C) またはスクリプト (ksh) を生成します。

注 - 厳密には、障害モニターを実装する `rt_name_probe` プログラムはコールバックプログラムではありません。RGM は `rt_name_probe` を直接呼び出すのではなく、`rt_name_monitor_start` および `rt_name_monitor_stop` を呼び出します。そして、これらのメソッドが `rt_name_probe` を呼び出すことによって、障害モニターが起動および停止されます。

Agent Builder が生成する 8 つのメソッドは次のとおりです。

- `rt_name_monitor_check`
- `rt_name_monitor_start`

- `rt_name_monitor_stop`
- `rt_name_probe`
- `rt_name_svc_start`
- `rt_name_svc_stop`
- `rt_name_update`
- `rt_name_validate`

各メソッドに固有な情報については、`rt_callbacks(1HA)` のマニュアルページを参照してください。

Agent Builder は、`install_directory/rt_name/src` ディレクトリ (C 出力の場合) に、次のファイルを作成します。

- ヘッダーファイル (`rt_name.h`)
- すべてのメソッドに共通なコードが記述されているソースファイル (`rt_name.c`)
- 共通なコード用のオブジェクトファイル (`rt_name.o`)
- 各メソッド用のソースファイル (`*.c`)
- 各メソッド用のオブジェクトファイル (`*.o`)

Agent Builder は `rt_name.o` ファイルを各メソッドの `.o` ファイルにリンクして、実行可能ファイルを `install_directory/rt_name/bin` ディレクトリに作成します。

ksh 出力の場合、`install_directory/rt_name/bin` と `install_directory/rt_name/src` ディレクトリは同じです。各ディレクトリには、7つのコールバックメソッドと PROBE メソッドに対応する 8つの実行可能スクリプトが格納されます。

注 - ksh 出力には、2つのコンパイル済みユーティリティプログラム (`gettime` と `gethostnames`) が含まれます。これらのプログラムは、時間を取得して検証を行うのに必要なコールバックメソッドです。

ソースコードを編集して、`make` コマンドを実行すると、コードを再コンパイルできます。さらに、再コンパイル後、`make pkg` コマンドを実行すると、新しいパッケージを生成できます。ソースコードの修正をサポートするために、Agent Builder はソースコード中の適切な場所に、コードを追加するためのコメント文を埋め込みます。168 ページの「生成されたソースコードの編集」を参照してください。

ユーティリティスクリプトとマニュアルページ

リソースタイプを生成し、そのパッケージをクラスタにインストールした後は、リソースタイプのインスタンス (リソース) をクラスタ上で実行する必要があります。一般に、リソースを実行するには、管理コマンドまたは SunPlex Manager を使用します。Agent Builder は、ターゲットリソースタイプのリソースを起動するためのカスタマイズされたユーティリティスクリプトに加え、リソースを停止および削除するスクリプトも生成します。これら 3つのスクリプトは `install_directory/rt_name/util` ディレクトリに格納されており、次のような処理を行います。

- 起動スクリプト - リソースタイプを登録し、必要なリソースグループとリソースを作成します。また、アプリケーションがネットワーク上のクライアントと通信するためのネットワークアドレスリソース (LocalHostname または SharedAddress) も作成します。
- 停止スクリプト — リソースを停止し、無効にします。
- 削除スクリプト — 起動スクリプトの処理を無効にします。つまり、リソース、リソースグループ、およびターゲットリソースタイプを停止して、システムから削除します。

注 - これらのスクリプトは内部的な規則を使用して、リソースとリソースグループの名前付けを行います。そのため、削除スクリプトを使用できるリソースは、対応する起動スクリプトで起動されたリソースだけです。

Agent Builder は、スクリプト名にアプリケーション名を追加することにより、スクリプトの名前付けを行います。たとえば、アプリケーション名が ftp の場合、各スクリプトは startftp、stopftp、および removeftp になります。

Agent Builder は、各ユーティリティースクリプト用のマニュアルページを *install_directory/rt_name/man/man1m* ディレクトリに格納します。これらのマニュアルページにはスクリプトに渡す必要があるパラメータについての説明が記載されているので、各スクリプトを起動する前に、これらのマニュアルページをお読みください。

これらのマニュアルページを表示するには、man コマンドに -M オプションを付けて、上記のマニュアルページが格納されているディレクトリへのパスを指定する必要があります。たとえば、ベンダーが SUNW で、アプリケーション名が ftp である場合、startftp(1M) のマニュアルページを表示するには、次のコマンドを使用します。

```
man -M install_directory/SUNWftp/man startftp
```

クラスタ管理者は、マニュアルページユーティリティースクリプトも利用できます。Agent Builder が生成したパッケージをクラスタ上にインストールすると、ユーティリティースクリプト用のマニュアルページは、*/opt/rt_name/man* ディレクトリに格納されます。たとえば、startftp(1M) のマニュアルページを表示するには、次のコマンドを使用します。

```
man -M /opt/SUNWftp/man startftp
```

サポートファイル

Agent Builder はサポートファイル (pkginfo、postinstall、postremove、preremove など) を *install_directory/rt_name/etc* ディレクトリに格納します。このディレクトリには、Resource Type Registration (RTR) ファイルも格納されます。RTR ファイルは、ターゲットリソースタイプが利用できるリソースとリソースタイププロ

パティを宣言して、リソースをクラスタに登録するときにプロパティ値を初期化します。詳細については、32 ページの「リソースとリソースタイププロパティの設定」を参照してください。RTR ファイルの名前は、ベンダー名とリソースタイプ名をピリオドで区切って連結したものです。(たとえば、SUNW.ftp)。

RTR ファイルは、ソースコードを再コンパイルしなくても、標準のテキストエディタで編集および修正できます。ただし、make pkg コマンドを使用して、パッケージを再構築する必要があります。

パッケージディレクトリ

install_directory/rt_name/pkg ディレクトリには、Solaris パッケージが格納されます。パッケージの名前は、ベンダー名とアプリケーション名を連結したものです(たとえば、SUNW.ftp)。*install_directory/rt_name/src* ディレクトリ内の Makefile は、新しいパッケージを作成するのに役立ちます。たとえば、ソースファイルを修正し、コードを再コンパイルした場合、あるいは、パッケージユーティリティスクリプトを修正した場合、make pkg コマンドを使用して新しいパッケージを作成します。

パッケージをクラスタから削除する場合、同時に複数のノードから pkgrm コマンドを実行しようとする、pkgrm コマンドが失敗する可能性があります。この問題を解決するには、次の 2 つの方法があります。

- クラスタのいずれかのノードで *removert_name* スクリプトを実行してから、任意のノードで pkgrm を実行します。
- クラスタの 1 つのノードで pkgrm を実行して、必要なすべてのクリーンアップ処理を行なった後、(必要であれば同時に) 残りのノードで pkgrm を実行します。

同時に複数のノードから pkgrm を実行しようとして失敗した場合は、再度いずれかのノードで pkgrm を実行した後、残りのノードで pkgrm を実行します。

rtconfig ファイル

C または ksh ソースコードを生成する場合、Agent Builder は、作業ディレクトリ内に構成ファイル *rtconfig* を生成します。このファイルには、作成および構成画面でユーザーが入力した情報が含まれています。既存のリソースタイプの作業ディレクトリから Agent Builder を起動した場合、または、「ファイル」メニューの「リソースタイプをロード」コマンドを使って既存のリソースタイプをロードした場合、Agent Builder は *rtconfig* ファイルを読み取り、作成および構成画面に、既存のリソースタイプ用としてユーザーが指定した情報を自動入力します。この機能は、既存のリソースタイプのクローンを作成したい場合に便利です。詳細については、167 ページの「既存のリソースタイプのクローンの作成」を参照してください。

ナビゲーション

Agent Builder のナビゲーションは操作が簡単でわかりやすいものです。Agent Builder は 2 段階形式のウィザードであり、各段階 (作成と構成) に対応した画面を提供します。各画面では、次のように情報を入力します。

- フィールドに情報を入力。
- ディレクトリ構造をブラウズして、ファイルまたはディレクトリを選択。
- 相互に排他的なラジオボタンセットの 1 つを選択 (たとえば、「スケーラブル」または「フェイルオーバー」)。
- チェックボックスのオン/オフ。たとえば、「ネットワーク認識」ボックスにチェックマークを入れると、ベースアプリケーションがネットワーク対応であることを指定します。チェックマークを入れなければ、アプリケーションが非ネットワーク対応であることを指定します。

各画面の下にあるボタンを使用すると、作業を完了したり、次の画面に進んだり、以前の画面に戻ったり、Agent Builder を終了したりできます。Agent Builder は、必要に応じて、これらのボタンを強調表示またはグレー表示します。

たとえば、作成画面において、必要なフィールドに入力し、希望のオプションにチェックマークを付けてから、画面の下にある「作成」ボタンをクリックします。この時点で、以前の画面は存在しないので、「戻る」ボタンはグレー表示されます。また、この作業が完成するまで次の手順には進めないで、「進む」ボタンもグレー表示されます。



Agent Builder は、画面の下にある出力ログ領域に進捗メッセージを表示します。作業が終了したとき、Agent Builder は成功メッセージまたは警告メッセージを出力ログに表示します。このとき、「進む」ボタンが強調表示されます。あるいは、この画面が最後の画面である場合、「キャンセル」ボタンだけが強調表示されます。

「キャンセル」ボタンを押すと、いつでも Agent Builder を終了できます。

「ブラウズ」ボタン

Agent Builder のフィールドの中には、情報を直接入力することも、「ブラウズ」ボタンをクリックしてディレクトリ構造をブラウズし、ファイルまたはディレクトリを選択することも可能なフィールドがあります。



「ブラウズ」をクリックします。

表示された画面で、フォルダをダブルクリックすると、フォルダが開きます。ファイルを強調表示すると、そのファイル名が「ファイル名」ボックスに表示されます。適切なファイルを選択し、強調表示してから、「選択」をクリックします。

注 - ディレクトリをブラウズしている場合は、ディレクトリを強調表示して、「開く」ボタンを選択します。サブディレクトリがない場合、Agent Builder はブラウズウィンドウを閉じて、強調表示されたディレクトリ名を適切なフィールドに表示します。サブディレクトリがある場合、「閉じる」ボタンをクリックすると、ブラウズウィンドウが閉じて、以前の画面に戻ります。Agent Builder は、強調表示したディレクトリ名を適切なフィールドに表示します。

画面の右上隅にあるアイコンは、次のような処理を行います。



ディレクトリツリーの1つ上のレベルに移動します。



ホームフォルダに戻ります。



現在選択しているフォルダの下に新しいフォルダを作成します。



ビューを切り替えます。将来のために予約されています。

メニュー

Agent Builder には、「ファイル」と「編集」メニューがあります。

「ファイル」メニュー

「ファイル」メニューでは、次の2つのコマンドを使用できます。

- リソースタイプをロード — 既存のリソースタイプをロードします。Agent Builder が提供するブラウザ画面を使用して、既存のリソースタイプ用の作業ディレクトリを選択します。Agent Builder を起動したディレクトリにリソースタイプが存在する場合、Agent Builder は自動的にそのリソースタイプをロードします。「リソースタイプをロード」コマンドを使用すると、任意のディレクトリから Agent Builder を起動した後、既存のリソースタイプを選択して、新しいリソースタイプを作成するためのテンプレートとして使用できます。詳細については、167 ページの「既存のリソースタイプのクローンの作成」を参照してください。
- 終了 — Agent Builder を終了します。作成画面または構成画面で「取り消す」をクリックしても、Agent Builder を終了できます。

「編集」メニュー

「編集」メニューでは、出力ログを消去または保存するコマンドを使用できます。

- 出力ログをクリア — 出力ログから情報を消去します。「作成」または「構成」を選択するたびに、Agent Builder は状態メッセージを出力ログに追加します。対話形式で繰り返してソースコードを修正し、Agent Builder で出力を生成し直しているときに、出力の生成ごとに状態メッセージを記録したい場合は、出力ログを使用するたびにログファイルの内容を保存および消去できます。
- 出力ログを保存 — ログの出力をファイルに保存します。Agent Builder が提供するブラウザ画面を使用すると、ディレクトリを選択して、ファイル名を指定できます。

第 10 章

汎用データサービス

この章では、最初に汎用データサービス (GDS) の概要について説明した後に、SunPlex Agent Builder または標準的な Sun Cluster 管理コマンドを使って GDS ベースのサービスを作成する方法について説明します。

- 177 ページの「汎用データサービスの概要」
- 183 ページの「SunPlex Agent Builder を使って GDS ベースのサービスを作成」
- 187 ページの「標準的な Sun Cluster 管理コマンドを使って GDS ベースのサービスを作成」
- 189 ページの「SunPlex Agent Builder のコマンド行インタフェース」

汎用データサービスの概要

汎用データサービス (GDS) とは、ネットワーク対応のシンプルなアプリケーションを Sun Cluster Resource Group Management フレームワークにプラグインすることによって、これらのアプリケーションを可用性の高いものにしたたり、スケーラブルなものにするための機構です。この機構では、アプリケーションを可用性の高いものにしたたり、スケーラブルなものにするために通常必要になるエージェントのコーディングは必要ありません。

GDS は、あらかじめコンパイルされた単一のデータサービスです。このアプローチでは、コールバックメソッド (rt_callbacks(1HA)) の実装やリソースタイプ登録ファイル (rt_reg(4)) など、コンパイル済みのデータサービスやそのコンポーネントを変更することはできません。

コンパイル済みリソースタイプ

汎用データサービスのリソースタイプ `SUNW.gds` は、`SUNWscgds` パッケージに含まれています。このパッケージは、クラスタのインストール時に `scinstall(1M)` ユーティリティでインストールされます。`SUNWscgds` パッケージには次のファイルが格納されています。

```
# pkgchk -v SUNWscgds

/opt/SUNWscgds
/opt/SUNWscgds/bin
/opt/SUNWscgds/bin/gds_monitor_check
/opt/SUNWscgds/bin/gds_monitor_start
/opt/SUNWscgds/bin/gds_monitor_stop
/opt/SUNWscgds/bin/gds_probe
/opt/SUNWscgds/bin/gds_svc_start
/opt/SUNWscgds/bin/gds_svc_stop
/opt/SUNWscgds/bin/gds_update
/opt/SUNWscgds/bin/gds_validate
/opt/SUNWscgds/etc
/opt/SUNWscgds/etc/SUNW.gds
```

GDS を使用する利点

GDS には、`SunPlex Agent Builder` が生成するソースコードモデル (`scdscreate` (1HA) のマニュアルページを参照) や標準的な `Sun Cluster` 管理コマンドを使用するのに比べ、次の利点があります。

- GDS は使い易いデータサービスです。
- GDS とそのメソッドはコンパイル済みであるため、変更できません。
- `SunPlex Agent Builder` を使って、アプリケーションを起動するスクリプトを生成できます。これらのスクリプトは、複数のクラスタ間で再利用できる `Solaris` でインストール可能なパッケージとしてパッケージ化されます。

GDS を使用するサービスの作成方法

GDS を使用するサービスの作成方法は 2 通りあります。

- `SunPlex Agent Builder` を使用
- 標準的な `Sun Cluster` 管理コマンドを使用

GDS と SunPlex Agent Builder

`SunPlex Agent Builder` を使用し、生成されるソースコードのタイプとして GDS を選択します。特定のアプリケーションのリソースを設定する起動スクリプト群を生成するためにユーザーの入力が必要です。

GDS と標準的な Sun Cluster 管理コマンド

この方法では SUNWscgds にあるコンパイル済みデータサービスコードを使用しますが、システム管理者は、標準的な Sun Cluster 管理コマンド (scrgadm(1M) と scswitch(1M)) を使って、リソースの作成と構成を行う必要があります。

GDS ベースのサービスを作成する方法の選択

188 ページの「標準的な Sun Cluster 管理コマンドを使って GDS ベースの高可用性サービスを作成する方法」や 188 ページの「標準的な Sun Cluster 管理コマンドを使って GDS ベースのスケラブルサービスを作成する方法」の手順からわかるように、適切な scrgadm や scswitch コマンドを実行するためには、かなりの分量の入力を行う必要があります。

/GDS と SunPlex Agent Builder を使用する方法では、この処理が簡単になります。この方法では、生成される起動スクリプトがユーザーに代わって scrgadm と scswitch コマンドを出力するからです。

GDS の使用が適さない場合

GDS には多くの利点がありますが、GDS の使用が適さない場合もあります。GDS の使用が適切でないのは次のような場合です。

- コンパイル済みリソースタイプを使用する場合よりも高度な制御が必要な場合。たとえば拡張プロパティを追加する場合や、デフォルト値を変更する場合など
- 特別な機能を追加するためにソースコードを変更する必要がある場合
- 複数のプロセスツリーを使用する場合
- ネットワーク対応でないアプリケーションを使用する場合

GDS の必須入力

次のプロパティを指定する必要があります。

- Start_command (拡張プロパティ)
- Port_list

Start_command

アプリケーションは、start_command プロパティに指定された起動コマンドで起動されます。このコマンドは、引数を備えた完全な UNIX コマンドでなければなりません。コマンドは、アプリケーションを起動するシェルに直接渡されます。

Port_list

Port_list プロパティは、アプリケーションが待機するポート群を指定したものです。Port_list プロパティは、SunPlex Agent Builder によって生成される start スクリプトか、scrgadm コマンド (標準的な Sun Cluster 管理コマンドを使用する場合) に指定されていなければなりません。

GDS の任意入力

GDS の任意入力には次のものがあります。

- Network_resources_used
- Stop_command (拡張プロパティ)
- Probe_command (拡張プロパティ)
- Start_timeout
- Stop_timeout
- Probe_timeout (拡張プロパティ)
- Child_mon_level (標準的な管理コマンドだけで使用される拡張プロパティ)
- Failover_enabled (拡張プロパティ)
- Stop_signal (拡張プロパティ)

Network_resources_used

このプロパティのデフォルト値は Null です。アプリケーションが 1 つ以上の特定のアドレスにバインドする必要がある場合は、このプロパティを指定する必要があります。このプロパティを省略するか、このプロパティが Null の場合、アプリケーションはすべてのアドレスに対して待機するものとみなされます。

GDS リソースを作成する場合は、LogicalHostname か SharedAddress リソースがあらかじめ構成されていなければなりません。LogicalHostname または SharedAddress リソースの構成方法については、『Sun Cluster 3.1 データサービスのインストールと構成』を参照してください。

値を指定する場合は、1 つまたは複数のリソース名を指定します。個々のリソース名には、1 つ以上の LogicalHostname か 1 つ以上の SharedAddress を指定できます。詳細は、r_properties(5) のマニュアルページを参照してください。

Stop_command

stop コマンドは、アプリケーションを停止し、アプリケーションが完全に停止した後で終了します。このコマンドは、アプリケーションを停止するシェルに直接渡される完全な UNIX コマンドでなければなりません。

Stop_command が指定されていると、GDS 停止メソッドは、停止タイムアウトの 80% を指定して停止コマンドを起動します。さらに、GDS 停止メソッドは、停止コマンドの起動結果がどうであれ、停止タイムアウトの 15% を指定して SIGKILL を送信します。タイムアウトの残り 5% は、処理のオーバーヘッドのために使用されます。

stop コマンドが省略されていると、GDS は、Stop_signal に指定されたシグナルを使ってアプリケーションを停止します。

Probe_command

probe コマンドは、特定のアプリケーションの状態を周期的にチェックします。このコマンドは、引数を備えた完全な UNIX コマンドでなければなりません。コマンドは、アプリケーションの状態をチェックするシェルに直接渡されます。アプリケーションの状態が正常であれば、probe コマンドは終了ステータスとして 0 を返します。

probe コマンドの終了ステータスは、アプリケーションの障害の重大度を判断するために使用されます。終了ステータス (probe ステータス) は、0 (正常) から 100 (全面的な障害) までの整数でなければなりません。probe ステータスは 201 という特別な値をもつことがあります。この場合には、Failover_enabled が false に設定されている場合を除き、アプリケーションのフェイルオーバーが直ちに行われます。GDS プローブアルゴリズム (scds_fm_action(3HA) のマニュアルページを参照) は、この probe ステータスを使って、アプリケーションをローカルに再起動するか別のノードにフェイルオーバーするかを決定します。終了ステータス 201 なら、アプリケーションは直ちにフェイルオーバーされます。

probe コマンドが省略されていると、GDS は、Newtork_resources_used プロパティか、scds_get_netaddr_list(3HA) の出力から得られる IP アドレス群を使ってアプリケーションに接続する独自の簡単な検証を行います。この検証では、接続に成功すると、接続を直ちに切り離します。接続と切り離しが両方とも正常なら、アプリケーションは正常に動作しているものとみなされます。

注 - GDS 提供の検証は、全機能を備えたアプリケーション固有の検証を代替するものではありません。

Start_timeout

このプロパティでは、start コマンドの起動タイムアウトを指定します (詳細は、179 179 ページの「Start_command」を参照)。Start_timeout のデフォルトは 300 秒です。

Stop_timeout

このプロパティでは、stop コマンドの停止タイムアウトを指定します (詳細は、180 180 ページの「Stop_command」を参照)。Stop_timeout のデフォルトは 300 秒です。

Probe_timeout

このプロパティでは、probe コマンドのプローブタイムアウトを指定します (詳細は、181 181 ページの「Probe_command」を参照)。Probe_timeout のデフォルトは 30 秒です。

Child_mon_level

このプロパティでは、PMF を通してどのプロセスを監視するかを制御します。このプロパティは、フォークされた子プロセスをどのようなレベルで監視するかを表します。これは、pmfadm(1M) コマンドの -c 引数と同等です。

このプロパティを省略するか、このプロパティにデフォルト値の -1 を指定することは、pmfadm コマンドで -c オプションを省略するのと同じ効果があります。つまり、すべての子プロセスとその子孫プロセスが監視されます。詳細については、pmfadm(1M) のマニュアルページを参照してください。

注 - このオプションは、標準的な Sun Cluster 管理コマンドを使用するときだけ指定できます。SunPlex Agent Builder を使用するときには指定できません。

Failover_enabled

ブール値のこの拡張プロパティでは、リソースのフェイルオーバー動作を制御します。この拡張プロパティに true を設定すると、アプリケーションは、再起動回数が retry_interval 秒間に retry_count を超えるとフェイルオーバーされます。

この拡張プロパティに false を設定すると、再起動回数が retry_interval 秒間に retry_count を超えてもアプリケーションの再起動やフェイルオーバーは行われません。

この拡張プロパティを使用すれば、アプリケーションリソースがリソースグループのフェイルオーバーを引き起こすことを防止できます。デフォルトは true です。

Stop_signal

GDS は、整数値のこの拡張プロパティを使って、PMF によるアプリケーションの停止に使用するシグナルを判別します。指定できる整数値については、signal(3head) のマニュアルページを参照してください。デフォルトは 15 (SIGTERM) です。

SunPlex Agent Builder を使って GDS ベースのサービスを作成

SunPlex Agent Builder を使って GDS ベースのサービスを作成できます。SunPlex Agent Builder の詳細については、第 9 章を参照してください。

Agent Builder を使って GDS ベースのサービスを作成

▼ Agent Builder を使って GDS ベースのサービスを作成する方法

1. **SunPlex Agent Builder** を起動します。

```
# /usr/cluster/bin/scdsbuilder
```

2. 「**SunPlex Agent Builder**」パネルが表示されます。



3. 「ベンダー名」にベンダー名を入力します。
4. 「アプリケーション名」にアプリケーション名を入力します。

注 - 「ベンダー名」と「アプリケーション名」の合計は9文字以内でなければなりません。この組み合わせは、起動スクリプトのパッケージ名として使用されます。

5. 作業ディレクトリに移動します。
「ブラウズ」のプルダウンを使用すれば、パスを入力する代わりにディレクトリを選択することができます。
6. データサービスがスケラブルなのかフェイルオーバーなのかを選択します。
GDSを作成するときには「ネットワーク認識」がデフォルトですので、これを選択する必要はありません。
7. 「GDS」を選択します。
8. 「作成」ボタンをクリックして起動スクリプトを作成します。
9. サービスの作成結果が「SunPlex Agent Builder」パネルに表示されます。「作成」ボタンが無効になります。「進む」ボタンを押します。



▼ 起動スクリプトを構成する方法

起動スクリプトを作成したなら、SunPlex Agent Builder を使って新しいサービスを構成する必要があります。

1. 「進む」 ボタンをクリックすると、構成パネルが表示されます。
2. 起動コマンドの場所を入力するか、「ブラウズ」 ボタンを使って起動コマンドの場所を選択します。
3. (省略可能) 停止コマンドを入力するか、「ブラウズ」 ボタンを使って停止コマンドの場所を選択します。
4. (省略可能) 検証コマンドを入力するか、「ブラウズ」 ボタンを使って検証コマンドの場所を選択します。
5. (省略可能) 起動、停止、検証コマンドのタイムアウト値を指定します。
6. 「構成」 をクリックして起動スクリプトの構成を開始します。

注 - このパッケージ名は、ベンダー名とアプリケーション名が結合したものです。

起動スクリプトのパッケージが作成され、次の場所に格納されます。

```
<working-dir>/<vendor_name><application>/pkg  
たとえば、 /export/wdir/NETapp/pkg のようになります。
```

7. 完成したパッケージをクラスタのすべてのノードにインストールします。

```
# cd /export/wdir/NETapp/pkg  
# pkgadd -d . NETapp
```

8. **pkgadd** の実行で次のファイルがインストールされます。

```
/opt/NETapp  
/opt/NETapp/README.app  
/opt/NETapp/man  
/opt/NETapp/man/man1m  
/opt/NETapp/man/man1m/removeapp.1m  
/opt/NETapp/man/man1m/startapp.1m  
/opt/NETapp/man/man1m/stopapp.1m  
/opt/NETapp/man/man1m/app_config.1m  
/opt/NETapp/util  
/opt/NETapp/util/removeapp  
/opt/NETapp/util/startapp  
/opt/NETapp/util/stopapp  
/opt/NETapp/util/app_config
```

注 – マニュアルページとスクリプト名は、上で入力したアプリケーション名の前にスクリプト名を付けたものです。たとえば、startapp のようになります。

マニュアルページを表示するには、マニュアルページへのパスを指定する必要があります。たとえば、startapp のマニュアルページを表示する場合は、次のように入力します。

```
# man -M /opt/NETapp/man startapp
```

9. クラスタのいずれかのノードでリソースを構成し、アプリケーションを起動します。

```
# /opt/NETapp/util/startapp -h <logichostname> -p <port and protocol list>
```

起動スクリプトの引数は、リソースのタイプがフェイルオーバーかスケラブルかで異なります。カスタマイズしたマニュアルページを検査するか、起動スクリプトを引数なしで実行して引数リストを入手してください。

```
# /opt/NETapp/util/startapp  
The resource name of LogicalHostname or SharedAddress must be specified.  
For failover services:  
Usage: startapp -h <logical host name>  
        -p <port and protocol list>  
        [-n <ipmpgroup/adaptor list>]  
For scalable services:  
Usage: startapp  
        -h <shared address name>  
        -p <port and protocol list>
```

```
[-l <load balancing policy>]
[-n <ipmpgroup/adapter list>]
[-w <load balancing weights>]
```

SunPlex Agent Builder の出力

SunPlex Agent Builder は 3 つの起動スクリプトと、パッケージ作成時の入力内容に基づいた構成ファイルを生成します。構成ファイルには、リソースグループとリソースタイプの名前が指定されます。

生成される起動スクリプトは次のとおりです。

- 起動スクリプト: リソースを構成し、RGM 制御のもとでアプリケーションを起動します。
- 停止スクリプト: アプリケーションを停止し、リソースやリソースグループを停止します。
- 削除スクリプト: 起動スクリプトによって作成されたリソースやリソースグループを削除します。

これらの起動スクリプトのインタフェースや動作は、SunPlex Agent Builder によって非 GDS ベースのエージェント用に作成されるユーティリティスクリプトのものと同じです。これらのスクリプトは、複数のクラスタで再利用できる Solaris にインストール可能なパッケージとしてパッケージ化されます。

構成ファイルをカスタマイズすれば、リソースグループの名前など、一般には `scrgadm` コマンドへの入力として指定されるパラメータを独自に設定できます。スクリプトをカスタマイズしないと、SunPlex Agent Builder が `scrgadm` のパラメータに対し妥当なデフォルトを設定します。

標準的な Sun Cluster 管理コマンドを使って GDS ベースのサービスを作成

この節では、これらのパラメータが実際に GDS に入力されるまでの手順を示します。GDS の使用や管理は、`scrgadm` や `scswitch` など、すでにある Sun Cluster 管理コマンドを通して行われます。

起動スクリプトに適切な機能が含まれている場合は、この節で述べる低位レベルの管理コマンドを入力する必要はありません。ただし、GDS ベースのリソースをより細かく制御したい場合は、このような低位レベルのコマンドを入力することもできます。起動スクリプトでは、これらのコマンドが実際に実行されます。

▼ 標準的な Sun Cluster 管理コマンドを使って GDS ベースの高可用性サービスを作成する方法

1. リソースタイプ **SUNW.gds** を登録します。

```
# scrgadm -a -t SUNW.gds
```

2. **LogicalHostname** リソースとフェイルオーバーサービス自体を含むリソースグループを作成します。

```
# scrgadm -a -g haapp_rg
```

3. **LogicalHostname** リソースのリソースを作成します。

```
# scrgadm -a -L -g haapp_rs -l hhead
```

4. フェイルオーバーサービス自体のリソースを作成します。

```
# scrgadm -a -j haapp_rs -g haapp_rg -t SUNW.gds \  
-y Scalable=false -y Start_timeout=120 \  
-y Stop_timeout=120 -x Probe_timeout=120 \  
-y Port_list="2222/tcp" \  
-x Start_command="/export/ha/appctl/start" \  
-x Stop_command="/export/ha/appctl/stop" \  
-x Probe_command="/export/app/bin/probe" \  
-x Child_mon_level=0 -y Network_resources_used=hhead \  
-x Failover_enabled=true -x Stop_signal=9
```

5. リソースグループ **haapp_rg** をオンラインにします。

```
# scswitch -Z -g haapp_rg
```

▼ 標準的な Sun Cluster 管理コマンドを使って GDS ベースのスケーラブルサービスを作成する方法

1. リソースタイプ **SUNW.gds** を登録します。

```
# scrgadm -a -t SUNW.gds
```

2. **SharedAddress** リソースのリソースグループを作成します。

```
# scrgadm -a -g sa_rg
```

3. **SharedAddress** リソースを **sa_rg** に作成します。

```
# scrgadm -a -S -g sa_rg -l hhead
```

4. スケーラブルサービスのリソースグループを作成します。

```
# scrgadm -a -g app_rg -y Maximum primaries=2 \  
-y Desired primaries=2 -y RG_dependencies=sa_rg
```

5. スケーラブルサービス自体のリソースグループを作成します。

```
# scrgadm -a -j app_rs -g app_rg -t SUNW.gds \  
-y Scalable=true -y Start_timeout=120 \  
-y Stop_timeout=120 -x Probe_timeout=120 \  
-y Port_list="2222/tcp" \  
-x Start_command="/export/app/bin/start" \  
-x Stop_command="/export/app/bin/stop" \  
-x Probe_command="/export/app/bin/probe" \  
-x Child_mon_level=0 -y Network_resource_used=hhead \  
-x Failover_enabled=true -x Stop_signal=9
```

6. ネットワークリソースを含むリソースグループをオンラインにします。

```
# scswitch -Z -g sa_rg
```

7. リソースグループ `app_rg` をオンラインにします。

```
# scswitch -Z -g app_rg
```

SunPlex Agent Builder のコマンド行インタフェース

SunPlex Agent Builder には、GUI インタフェースと同等の機能をもつコマンド行インタフェースがあります。このインタフェースは、`scsdcreate(1HA)` と `scdsconfig(1HA)` からなります。この節では、GUI ベースの手順によるものと同じ機能をコマンド行インタフェースを使って行います。

▼ SunPlex Agent Builder のコマンド行インタフェースを使って GDS ベースのサービスを作成する方法

1. サービスを作成します。

フェイルオーバーサービスの場合:

```
# scsdcreate -g -V NET -T app -d /export/wdir
```

スケーラブルサービスの場合:

```
# scsdcreate -g -s -V NET -T app -d /export/wdir
```

注 `-d` パラメータは任意です。このパラメータを指定しない場合は、作業ディレクトリとして現在のディレクトリが使用されます。

2. サービスを構成します。

```
# scdsconfig -s "/export/app/bin/start" -t "/export/app/bin/stop" \
-m "/export/app/bin/probe" -d /export/wdir
```

注 – start コマンドだけが必須で、他のパラメータはすべて任意です。

3. 完成したパッケージをクラスタのすべてのノードにインストールします。

```
# cd /export/wdir/NETapp/pkg
# pkgadd -d . NETapp
```

4. **pkgadd** の実行で次のファイルがインストールされます。

```
/opt/NETapp
/opt/NETapp/README.app
/opt/NETapp/man
/opt/NETapp/man/man1m
/opt/NETapp/man/man1m/removeapp.1m
/opt/NETapp/man/man1m/startapp.1m
/opt/NETapp/man/man1m/stopapp.1m
/opt/NETapp/man/man1m/app_config.1m
/opt/NETapp/util
/opt/NETapp/util/removeapp
/opt/NETapp/util/startapp
/opt/NETapp/util/stopapp
/opt/NETapp/util/app_config
```

注 – マニュアルページとスクリプト名は、上で入力した Application Name の前にスクリプト名を付けたものですたとえば、startapp のようになります。

マニュアルページを表示するには、マニュアルページへのパスを指定する必要があります。たとえば、startapp のマニュアルページを表示する場合は、次のように入力します。

```
# man -M /opt/NETapp/man startapp
```

5. クラスタのいずれかのノードでリソースを構成し、アプリケーションを起動します。

```
# /opt/NETapp/util/startapp -h <logichostname> -p <port and protocol list>
```

起動スクリプトの引数は、リソースのタイプがフェイルオーバーかスケラブルかで異なります。リソースのタイプがフェイルオーバーかスケラブルかで異なります。カスタマイズしたマニュアルページを検査するか、起動スクリプトを引数なしで実行して引数リストを入手してください。

```
# /opt/NETapp/util/startapp
The resource name of LogicalHostname or SharedAddress must be specified.
For failover services:
Usage: startapp -h <logical host name>
```

```
-p <port and protocol list>
[-n <ipmpgroup/adapter list>]
For scalable services:
Usage: startapp
-h <shared address name>
-p <port and protocol list>
[-l <load balancing policy>]
[-n <ipmpgroup/adapter list>]
[-w <load balancing weights>]
```


第 11 章

DSDL のリファレンス

この章では、DSDL の API 関数について簡単に説明します。個々の DSDL 関数の詳細については、そのマニュアルページ (3HA) を参照してください。DSDL は C 言語用のインタフェースだけを定義します。スクリプト用の DSDL インタフェースはありません。

DSDL が提供する関数は、次のカテゴリに分類されます。

- 193 ページの「汎用関数」
- 195 ページの「プロパティ関数」
- 195 ページの「ネットワークリソースアクセス関数」
- 196 ページの「PMF 関数」
- 197 ページの「障害監視関数」
- 197 ページの「ユーティリティ関数」

DSDL Functions

この節では、DSDL 関数の各カテゴリを簡単に説明します。DSDL 関数を定義するリファレンスについては、個々のマニュアルページ (3HA) を参照してください。

汎用関数

このカテゴリの関数は、さまざまな機能を提供します。これらの関数は、次のような処理を行います。

- DSDL 環境を初期化します。
- リソース、リソースタイプ、およびリソースグループの名前、ならびに、拡張プロパティの値を取得します。
- リソースグループをフェイルオーバーおよび再起動し、リソースを再起動します。

- エラー文字列をエラーメッセージに変換します。
- タイムアウトを適用してコマンドを実行します。

次の関数は、呼び出しメソッドを初期化します。

- `scds_initialize` – リソースを割り当て、DSDL 環境を初期化します。
- `scds_close` – `scds_initialize` が割り当てたリソースを解放します。

次の関数は、リソース、リソースタイプ、リソースグループ、および拡張プロパティについての情報を取得します。

- `scds_get_resource_name` – 呼び出しプログラム用のリソース名を取得します。
- `scds_get_resource_type_name` – 呼び出しプログラム用のリソースタイプ名を取得します。
- `scds_get_resource_group_name` – 呼び出しプログラム用のリソースグループ名を取得します。
- `scds_get_ext_property` – 指定した拡張プロパティの値を取得します。
- `scds_free_ext_property` – `scds_get_ext_property` が割り当てたメモリーを解放します。

次の関数は、リソースが使用している SUNW.HAStoragePlus リソースについての状態情報を取得します。

- `scds_hasp_check` – リソースが使用している SUNW.HAStoragePlus リソースについての状態情報を取得します。当該リソース用に定義されている `Resource_dependencies` または `Resource_dependencies_weak` のシステム属性を使用することによって、当該リソースが依存しているすべての SUNW.HAStoragePlus リソース状態 (オンラインであるか、オンラインでないか) についての情報が得られます。

SUNW.HAStoragePlus については、『*Sun Cluster 3.1 データサービスのインストールと構成*』の「高可用性ローカルファイルシステムの有効化」を参照してください。

次の関数は、リソースまたはリソースグループをフェイルオーバーまたは再起動します。

- `scds_failover_rg` – リソースグループをフェイルオーバーします。
- `scds_restart_rg` – リソースグループを再起動します。
- `scds_restart_resource` – リソースを再起動します。

次の2つの関数は、タイムアウトを適用してコマンドを実行し、エラーコードをエラーメッセージに変換します。

- `scds_timerun` – タイムアウトを適用してコマンドを実行します。
- `scds_error_string` – エラーコードをエラーメッセージに変換します。

プロパティ関数

このカテゴリの関数は、関連するリソース、リソースグループ、およびリソースタイプ (よく使用される一部の拡張プロパティも含む) に固有なプロパティにアクセスするのに有用な API を提供します。DSDL は、`scds_initialize` を使用して、コマンド行引数を解析します。`scds_initialize` 関数は、関連するリソース、リソースグループ、およびリソースタイプの様々なプロパティをキャッシュに入れます。

これらすべての関数については、`scds_property_functions(3HA)` のマニュアルページを参照してください。このカテゴリには、次の関数が含まれます。

- `scds_get_rs_property_name`
- `scds_get_rg_property_name`
- `scds_get_rt_property_name`
- `scds_get_ext_property_name`

ネットワークリソースアクセス関数

このカテゴリの関数は、リソースおよびリソースグループが使用するネットワークリソースを、取得、出力、および解放します。ここで説明する `scds_get_*` 関数は、RMAPI 関数を使用して `Network_resources_used` や `Port_list` などのプロパティを照会しなくても、ネットワークリソースを取得できる便利な方法を提供します。`scds_print_*` 関数は、`scds_get_*` 関数から戻されたデータ構造から値を出力します。`scds_free_*` 関数は `scds_get_*` 関数が割り当てたメモリーを解放します。

次の関数は、ホスト名に関連した処理を行います。

- `scds_get_rg_hostnames` – ネットワークグループ内のネットワークリソースが使用するホスト名のリストを取得します。
- `scds_get_rs_hostnames` – リソースが使用するホスト名のリストを取得します。
- `scds_print_net_list` – `scds_get_rg_hostnames` または `scds_get_rs_hostnames` が戻したホスト名のリストの内容を出力します。
- `scds_free_net_list` – `scds_get_rg_hostnames` または `scds_get_rs_hostnames` が割り当てたメモリーを解放します。

次の関数は、ポートリストに関連した処理を行います。

- `scds_get_port_list` – リソースが使用するポートとプロトコルのペアのリストを取得します。
- `scds_print_port_list` – `scds_get_port_list` が戻したポートとプロトコルのペアのリストの内容を出力します。
- `scds_free_port_list` – `scds_get_port_list` が割り当てたメモリーを解放します。

次の関数は、ネットワークアドレスに関連した処理を行います。

- `scds_get_netaddr_list` – リソースが使用するネットワークアドレスのリストを取得します。
- `scds_print_netaddr_list` – `scds_get_netaddr_list` が戻したネットワークアドレスのリストの内容を出力します。
- `scds_free_netaddr_list` – `scds_get_netaddr_list` が割り当てたメモリーを解放します。

TCP 接続を使用する障害監視

このカテゴリの関数は、TCP ベースの監視を行います。通常、障害モニターはこれらの関数を使用して、サービスとの単純ソケット接続を確立し、サービスのデータを読み書きしてサービスの状態を確認した後、サービスとの接続を切断します。

このカテゴリには、次の関数が含まれます。

- `scds_tcp_connect` – プロセスとの TCP 接続を確立します。
- `scds_tcp_read` – TCP 接続を使用して、監視対象のプロセスからデータを読み取ります。
- `scds_tcp_write` – TCP 接続を使用して、監視対象のプロセスにデータを書き込みます。
- `scds_simple_probe` – プロセスとの TCP 接続を確立および終了することによって、プロセスを検証します。
- `scds_tcp_disconnect` – 監視対象のプロセスとの接続を終了します。

PMF 関数

このカテゴリの関数は、PMF 機能をカプセル化します。OMF 経由の監視における DSDL モデルは、`pmfadm(1M)` に対して、暗黙のタグ値を作成および使用します。また、PMF 機能は、`Restart_interval`、`Retry_count`、および `action_script` 用の暗黙値も使用します (`pmfadm(1M)` の `-t`、`-n`、および `-a` オプション)。最も重要な点は、DSDL が、PMF によって検出されたプロセス停止履歴を、障害モニターによって検出されたアプリケーション障害履歴に結びつけ、再起動またはフェイルオーバーのどちらを行うかを決定することです。

このカテゴリには、次の関数が含まれます。

- `scds_pmf_get_status` – 指定されたインスタンスが PMF の制御下で監視されているかどうかを判定します。
- `scds_pmf_restart_fm` – PMF を使用して障害モニターを再起動します。
- `scds_pmf_signal` – 指定されたシグナルを PMF の制御下で動作しているプロセスツリーに送信します。
- `scds_pmf_start` – 指定されたプログラム (障害モニターを含む) を PMF の制御下で実行します。

- `scds_pmf_stop` – PMF の制御下で動作しているプロセスを終了します。
- `scds_stop_monitoring` – PMF の制御下で動作しているプロセスの監視を停止します。

障害監視関数

このカテゴリの関数は、障害履歴を保持し、その履歴を `Retry_count` および `Retry_interval` プロパティと関連付けて評価することにより、障害監視の事前定義モデルを提供します。

このカテゴリには、次の関数が含まれます。

- `scds_fm_sleep` – 障害モニター制御ソケット上でメッセージを待ちます。
- `scds_fm_action` – 検証終了後にアクションを実行します。
- `scds_fm_print_probes` – 検証状態情報をシステムログに書き込みます。

ユーティリティ関数

このカテゴリの関数は、メッセージやデバッグ用メッセージをシステムログに書き込みます。このカテゴリには、次の 2 つの関数が含まれます。

- `scds_syslog` – メッセージをシステムログに書き込みます。
- `scds_syslog_debug` – デバッグ用メッセージをシステムログに書き込みます。

付録 A

標準プロパティ

この付録では、標準リソースタイプ、リソースグループ、リソースプロパティについて説明します。また、システム定義プロパティの変更および拡張プロパティの作成に使用するリソースプロパティ属性についても説明します。

この章の内容は次のとおりです。

- 199 ページの「リソースタイププロパティ」
- 203 ページの「リソースプロパティ」
- 212 ページの「リソースグループプロパティ」
- 217 ページの「リソースプロパティの属性」

注 - True や False などのプロパティ値は、大文字と小文字は区別されません。

リソースタイププロパティ

以下の表に、Sun Cluster によって定義されるリソースタイププロパティを示します。プロパティ値は、次のように分類されます(「カテゴリ」列)。

- 必須 — Resource Type Registration (RTR) ファイル内に利用値を必要とするプロパティです。値がない場合は、プロパティが属するオブジェクトを作成できません。ブランクまたは空の文字列を値として指定することはできません。
- 条件付き — このプロパティが存在するためには、RTR ファイル内で宣言する必要があります。宣言されていない場合は、RGM はこのプロパティを作成しないため、管理ユーティリティで利用できません。ブランクまたは空の文字列を値として指定できます。プロパティが RTR ファイル内で宣言されており、値が指定されていない場合には、RGM はデフォルト値を使用します。

- 条件付き / 明示 — このプロパティが存在するためには、明示的に値を指定して、RTR ファイル内で宣言する必要があります。宣言されていない場合は、RGM はこのプロパティを作成しないため、管理ユーティリティで利用できません。空白または空の文字列を値として指定することはできません。
- 任意 — プロパティを RTR ファイル内で宣言できます。宣言しない場合は、RGM はこのプロパティを作成し、デフォルト値を使用します。プロパティが RTR ファイル内で宣言されており、値が指定されていない場合は、RGM は、プロパティが RTR ファイル内で宣言されないときのデフォルト値と同じ値を使用します。

リソースタイププロパティは、`Installed_nodes` を除き、管理ユーティリティによって更新することができません。`Installed_nodes` は、RTR ファイル内で宣言できないため、管理者が設定する必要があります。

表 A-1 リソースタイププロパティ

プロパティ名	説明	更新の可否	カテゴリ
API_version (整数)	このリソースタイプの実装が使用するリソース管理 API のバージョン。 SC 3.1 のデフォルトは 2 です。	不可	任意
Boot (文字列)	任意のコールバックメソッド。ノード上で RGM が呼び出すプログラムへのパス。このプログラムは、このタイプのリソースがすでに管理状態にあるときに、クラスタの結合または再結合を行います。このメソッドは、Init メソッドと同様に、このタイプのリソースに対し、初期化アクションを行う必要があります。	不可	条件付き / 明示
Failover (ブール値)	True は、複数のノード上で同時にオンラインになることのできる任意のグループで、このタイプのリソースを構成できないことを示します。デフォルトは、False です。	不可	任意
Fini (文字列)	任意のコールバックメソッド。このリソースタイプを RGM 管理の対象外にすると RGM によって呼び出されるプログラムのパスを指定します。	不可	条件付き / 明示
Init (文字列)	任意のコールバックメソッド。このリソースタイプを RGM 管理対象にすると RGM によって呼び出されるプログラムのパスを指定します。	不可	条件付き / 明示
Init_nodes (列挙)	値には、 <code>RG primaries</code> (リソースをマスターできるノードだけ)、または <code>RT installed_nodes</code> (リソースタイプがインストールされるすべてのノード) を指定できます。RGM が <code>Init</code> 、 <code>Fini</code> 、 <code>Boot</code> 、 <code>Validate</code> メソッドをコールするノードを示します。 デフォルト値は、 <code>RG primaries</code> です。	不可	任意

表 A-1 リソースタイププロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
Installed_nodes (文字配列)	リソースタイプの実行が許可されるクラスタノード名のリスト。RGM は、自動的にこのプロパティを作成します。クラスタ管理者は値を設定できます。このプロパティは、RTR ファイル内で宣言できません。 デフォルトは、すべてのクラスタノードです。	可	クラスタ管理者は構成可能
Monitor_check (文字列)	任意のコールバックメソッド。障害モニターの要求によってこのリソースタイプのフェイルオーバーを実行する前に、RGM によって呼び出されるプログラムのパスを指定します。	不可	条件付き / 明示
Monitor_start (文字列)	任意のコールバックメソッド。このリソースタイプの障害モニターを起動するために RGM によって呼び出されるプログラムのパスを指定します。	不可	条件付き / 明示
Monitor_stop (文字列)	Monitor_start が設定されている場合、必須のコールバックメソッドになります。このリソースタイプの障害モニターを停止するために RGM によって呼び出されるプログラムのパスを指定します。	不可	条件付き / 明示
各クラスタノード上の Num_resource_restarts (整数)。	このプロパティは、RGM によって、このノード上のこのリソースに対して過去 n 秒間 (n はリソースの Retry_interval プロパティの値) に実行された scha_control RESTART 呼び出しの回数に設定されません。リソースタイプが Retry_interval プロパティを宣言していない場合、この型のリソースは Num_resource_restarts プロパティを使用できません。	不可	照会のみ
Pkglist (文字配列)	リソースタイプのインストールに含まれている任意のパッケージリスト。	不可	条件付き / 明示
Postnet_stop (文字列)	任意のコールバックメソッド。このリソースタイプがネットワークアドレスリソース (Network_resources_used) に依存している場合、このネットワークアドレスリソースの stop メソッドの呼び出し後に RGM によって呼び出されるプログラムのパスを指定します。このメソッドは、ネットワークインタフェースの停止設定に続いて、必要な Stop アクションを行います。	不可	条件付き / 明示

表 A-1 リソースタイププロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
Prenet_start (文字列)	任意のコールバックメソッド。このリソースタイプがネットワークアドレスリソース (Network_resources_used) に依存している場合、このネットワークアドレスリソースの Start メソッドの呼び出し前に RGM によって呼び出されるプログラムのパスを指定します。ネットワークインタフェースが起動に構成される前に必要な Start アクションを行う必要があります。	不可	条件付き / 明示
RT_basedir (文字列)	コールバックメソッドの相対パスを補うために使用するディレクトリパス。このパスは、リソースタイプパッケージのインストール場所に設定します。スラッシュ (/) で開始する完全なパスを指定する必要があります。すべてのメソッドパス名が絶対パスの場合には、指定する必要はありません。	不可	必須 (絶対パスでないメソッドパスがある場合)
RT_description (文字列)	リソースタイプの簡単な説明。 デフォルトは空の文字列です。	不可	条件付き
Resource_type (文字列)	リソースタイプの名前。 現在登録されているリソースタイプ名を表示するには、次のコマンドを使用します。 <code>scrgadm -p</code> Sun Cluster 3.1 以降、リソースタイプ名は次の形式をとります。 <code>vendor_id.resource_type:version</code> リソースタイプ名は、RTR ファイル内に指定された 3 つのプロパティ <code>Vendor_id</code> 、 <code>Resource_type</code> 、 <code>RT_version</code> で構成されます。scrgadm コマンドでは、区切り文字としてピリオドとコロンを使用します。リソースタイプ名の最後の部分、 <code>RT_version</code> には、 <code>RT_version</code> プロパティと同じ値が入ります。 重複を防ぐため、 <code>Vendor_id</code> には、リソースタイプの作成元の会社のストックシンボルを使用することをお勧めします。 Sun Cluster 3.1 以前に作成されたリソースタイプ名は次の形式をとります。 <code>vendor_id.resource_type</code> デフォルトは空の文字列です。	不可	必須

表 A-1 リソースタイププロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
RT_version (文字列)	Sun Cluster 3.1 以降、このリソースタイプの実装に必要なバージョンを指定します。RT_version は、完全なリソースタイプ名の末尾の部分です。	不可	条件付き / 明示
Single_instance (ブール値)	True の場合、このリソースタイプはクラスタ内に 1 つだけ存在できます。したがって、RGM は、同時に 1 つのこのリソースタイプだけに、クラスタ全体に渡っての実行を許可します。 デフォルト値は、False です。	不可	任意
Start (文字列)	コールバックメソッド。このリソースタイプを起動するために RGM によって呼び出されるプログラムのパスを指定します。	不可	必須 (RTR ファイルで Prenet_start メソッドが宣言されていない場合)
Stop (文字列)	コールバックメソッド。このリソースタイプを停止するために RGM によって呼び出されるプログラムのパスを指定します。	不可	必須 (RTR ファイルで Postnet_stop メソッドが宣言されていない場合)
Update (文字列)	任意のコールバックメソッド。この型の実行中のリソースのプロパティが変更されたとき RGM によって呼び出されるプログラムのパスを指定します。	不可	条件付き / 明示
Validate (文字列)	任意のコールバックメソッド。この型のリソースのプロパティ値を検査するために呼び出されるプログラムのパスを指定します。	不可	条件付き / 明示
Vendor_ID (文字列)	Resource_type を参照してください。	不可	条件付き

リソースプロパティ

表 A-2 に、Sun Cluster によって定義されているリソースプロパティを示します。プロパティ値は、次のように分類されます (「カテゴリ」列)。

- 必須 — 管理者は、管理ユーティリティでリソースを作成するときに、必ず値を指定する必要があります。
- 任意 — 管理者がリソースグループの作成時に値を指定しない場合、システムがデフォルト値を提供します。
- 条件付き — プロパティが RTR ファイルで宣言されている場合にのみ、RGM がプロパティを作成します。宣言されていない場合は、プロパティは存在せず、システム管理者はこれを利用できません。RTR ファイルで宣言されている条件付きのプロパティは、デフォルト値が RTR ファイル内で指定されているかどうかにか

よって、必須または任意になります。詳細については、各条件付きプロパティの説明を参照してください。

- 照会のみ— 管理ツールから直接設定できません。

表 A-2 の「更新の可否」列では、リソースプロパティが更新可能かどうか、更新可能な場合はいつ更新できるかを示しています。

None または False	不可
True または Anytime	任意の時点 (Anytime)
At_creation	リソースをクラスタに追加するとき
When_disabled	リソースが無効なとき

表 A-2 リソースプロパティ

プロパティ名	説明	更新の可否	カテゴリ
Affinity_timeout (整数)	<p>リソース内のサービスのクライアント IP アドレスからの接続は、この時間 (秒数) 内に同じサーバーノードに送信されます。</p> <p>このプロパティは、Load_balancing_policy が Lb_sticky または Lb_sticky_wild の場合にかぎり有効です。さらに、Weak_affinity を False (デフォルト値) に設定する必要があります。</p> <p>このプロパティは、スケーラブルサービス専用です。</p>	任意の時点 (Anytime)	任意
Cheap_probe_interval (整数)	<p>リソースの即時障害検証の呼び出しの間隔 (秒数)。このプロパティは、RGM のみが作成でき、RTR ファイル内で宣言されている場合には、管理者はこのプロパティを利用できます。</p> <p>デフォルト値が RTR ファイル内で指定されている場合は、このプロパティは任意です。リソースタイプファイル内に Tunable 属性が指定されていない場合、このプロパティの Tunable 値は When_disabled になります。</p> <p>Default 属性が RTR ファイルのプロパティ宣言に指定されていない場合は、このプロパティは必須です。</p>	無効時 (When_disabled)	条件付き

表 A-2 リソースプロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
拡張プロパティ	そのリソースのタイプの RTR ファイルで宣言される拡張プロパティ。リソースタイプの実装によって、これらのプロパティを定義します。拡張プロパティに設定可能な各属性については、表 A-4 を参照してください。	特定のプロパティに依存	条件付き
Failover_mode (列挙)	<p>設定可能な値は None、Soft、Hard です。リソース上の start、stop、または monitor_stop メソッド呼び出しに失敗した場合、RGM がリソースグループを再配置するか、あるいはノードを終了するかを制御します。None は、RGM が単にリソース状態をメソッド失敗に設定し、オペレータの介入を待つことを示します。Soft は、RGM が、start メソッドの失敗時にはリソースのグループを別のノードに再配置するが、stop または monitor_stop メソッドの失敗時にはリソースを STOP_FAILED 状態、リソースグループを ERROR_STOP_FAILED 状態にしてオペレータの介入を待つことを示します。stop または monitor_stop の失敗時には、None と Soft のどちらに設定していても同じ結果になります。Hard は、start メソッドが失敗したときに、グループの再配置を行い、stop または monitor_stop メソッドが失敗したときに、クラスタノードを異常終了させることで、リソースの強制的な停止を行うことを示します。</p> <p>デフォルトは、None です。</p>	任意の時点 (Anytime)	オプション

表 A-2 リソースプロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
Load_balancing_policy (文字列)	<p>使用する負荷均衡ポリシーを定義する文字列。このプロパティは、スケーラブルサービスに対してのみ使用します。Scalable プロパティが RTR ファイルで宣言されている場合、RGM は自動的にこのプロパティを作成します。</p> <p>Load_balancing_policy は、次の値をとることができます。</p> <p>Lb_weighted (デフォルト)。 Load_balancing_weights プロパティで設定されているウエイトに従って、さまざまなノードに負荷が分散されます。</p> <p>Lb_sticky スケーラブルサービスの指定のクライアント (クライアントの IP アドレスで識別される) は、常に同じクラスタノードに送信されます。</p> <p>Lb_sticky_wild。指定のクライアント (クライアントの IP アドレスで識別される) はワイルドカードスティッキーサービスの IP アドレスに接続され、送信時に使用されるポート番号とは無関係に、常に同じクラスタノードに送信されます。</p> <p>デフォルト値は、Lb_weighted です。</p>	作成時 (At-creation)	条件付き / 任意
Load_balancing_weights (文字配列)	<p>このプロパティは、スケーラブルサービスに対してのみ使用します。Scalable プロパティが RTR ファイルで宣言されている場合、RGM は自動的にこのプロパティを作成します。形式は、「weight@node,weight@node」になります。ここで、weight は、指定したノード (node) に対する負荷分散の相対的な割り当てを示す整数になります。ノードに分散される負荷の割合は、すべてのウエイトの合計でこのノードのウエイトを割った値になります。たとえば、1@1,3@2 は、ノード 1 に負荷の 1/4 が割り当てられ、ノード 2 に負荷の 3/4 が割り当てられることを意味します。デフォルトの空の文字列 ("") は、一定の分散を指定します。明示的にウエイトを割り当てられていないノードのウエイトは、デフォルトで 1 になります。</p> <p>Tunable 属性がリソースタイプファイルに指定されていない場合は、プロパティの Tunable 値は Anytime (任意の時点) になります。このプロパティを変更すると、新しい接続時にのみ分散が変更されます。</p> <p>デフォルト値は、空の文字列 ("") です。</p>	任意の時点 (Anytime)	条件付き / 任意

表 A-2 リソースプロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
リソースタイプの各コールバックメソッドの <i>method_timeout</i> (整数)	RGM がメソッドの呼び出しに失敗したと判断するまでの時間 (秒)。 メソッド自身が RTR ファイルで宣言されている場合、デフォルトは、3,600 秒 (1 時間) です。	任意の時点 (Anytime)	条件付き 任意
Monitored_switch (列挙)	クラスタ管理者が管理ユーティリティを使用してモニターを有効または無効にすると、RGM によって Enabled または Disabled に設定されます。Disabled に設定されると、再び有効に設定されるまで、モニターは Start メソッドを呼び出しません。リソースが、モニターのコールバックメソッドを持っていない場合は、このプロパティは存在しません。 デフォルトは Enabled です。	不可 (None)	照会のみ
Network_resources_used (文字配列)	リソースが使用する論理ホスト名または共有アドレスネットワークリソースのリスト。スケールアップサービスの場合、このプロパティは別のリソースグループに存在する共有アドレスリソースを参照する必要があります。フェイルオーバーサービスの場合、このプロパティは同じリソースグループに存在する論理ホスト名または共有アドレスを参照します。Scalable プロパティが RTR ファイルで宣言されている場合、RGM は自動的にこのプロパティを作成します。Scalable が RTR ファイルで宣言されていない場合、Network_resources_used は RTR ファイルで明示的に宣言されていないかぎり使用できません。 Tunable 属性がリソースタイプファイルに指定されていない場合は、プロパティの Tunable 値は、At_creation (作成時) になります。	作成時 (At_creation)	条件付き / 必須
On_off_switch (列挙)	クラスタ管理者が管理ユーティリティを使用してリソースを有効または無効にすると、RGM によって Enabled または Disabled に設定されます。無効に設定されると、再び有効に設定されるまで、リソースはコールバックを呼び出しません。 デフォルトは、Disabled です。	不可 (None)	照会のみ

表 A-2 リソースプロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
Port_list (文字配列)	<p>サーバーが待機するポート番号をコンマで区切ったリスト。各ポート番号の末尾に、そのポートが使用しているプロトコルが追加されます (例:Port_list=80/tcp)。Scalable プロパティが RTR ファイルで宣言されている場合、RGM は自動的に Port_list を作成します。それ以外の場合、このプロパティは RTR ファイルで明示的に宣言されていないかぎり使用できません。</p> <p>Apache 用にこのプロパティを設定する場合は、このマニュアルの Apache に関する章を参照してください。</p>	作成時 (At_creation)	条件付き 必須
R_description (文字列)	<p>リソースの簡単な説明。 デフォルトは空の文字列です。</p>	任意の時点 (Anytime)	任意
Resource_dependencies (文字配列)	<p>このリソースをオンラインにするために、順にオンラインにする必要のある同じグループ内のリソースのリスト。リスト内の任意のリソースの起動に失敗した場合、このリソースは起動されません。グループをオフラインにすると、このリソースを停止してから、リスト内のリソースが停止されます。このリソースが先に無効にならなければ、リスト内のリソースは無効にできません。</p> <p>デフォルトは、空のリストです。</p>	任意の時点 (Anytime)	任意
Resource_dependencies_weak (文字配列)	<p>グループ内のメソッド呼び出しの順序を決定する同じグループ内のリソースのリスト。RGM は、このリスト内のリソースの Start メソッドを先に呼び出してから、このリソースの Start メソッドを呼び出します。また、停止する場合は、このリソースの Stop メソッドを先に呼び出してから、リスト内のリソースの Stop メソッドを呼び出します。リスト内のリソースが開始に失敗した場合、または無効になっても、リソースはオンラインを維持できます。</p> <p>デフォルトは、空のリストです。</p>	任意の時点 (Anytime)	任意
Resource_name (文字列)	<p>リソースインスタンスの名前。クラスタ構成内で一意にする必要があります。リソースが作成された後で変更はできません。</p>	不可 (None)	必須

表 A-2 リソースプロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
Resource_project_name (文字列)	<p>リソースに関連付けられた Solaris プロジェクト名。このプロパティは、CPU の共有、クラスタデータサービスのリソースプールといった Solaris のリソース管理機能に適用できます。RGM は、リソースをオンラインにすると、このプロジェクト名を持つ関連プロセスを起動します。このプロパティが指定されていない場合、プロジェクト名は、リソースが属しているリソースグループの RG_project_name プロパティから取得されます。rg_properties(5) を参照してください。どちらのプロパティも指定されなかった場合、RGM は事前定義済みのプロジェクト名 default を使用します。プロジェクトデータベース内に存在するプロジェクト名を指定する必要があります。また、root ユーザーは、このプロジェクトのメンバーとして構成されている必要があります。このプロパティは Solaris 9 以降でサポートされます。</p> <p>注 - このプロパティへの変更を有効にするには、リソースを再起動する必要があります。</p> <p>デフォルトは null です。</p>	任意の時点 (Anytime)	オプション
各クラスタノードの Resource_state (列挙)	<p>RGM が判断した各クラスタノード上のリソースの状態。可能な状態は次のとおりです。Online、Offline、Stop_failed、Start_failed、Monitor_failed、Online_not_monitored。</p> <p>このプロパティは、ユーザーは構成できません。</p>	不可 (None)	照会のみ
Retry_count (整数)	<p>リソースの起動に失敗した場合にモニターが再起動を試みる試行回数です。このプロパティは、RGM のみが作成でき、RTR ファイル内で宣言されている場合は、管理者は利用できます。デフォルト値が RTR ファイルで指定されている場合は、このプロパティは任意です。</p> <p>リソースタイプファイル内に Tunable 属性が指定されていない場合、このプロパティの Tunable 値は When_disabled になります。</p> <p>Default 属性が RTR ファイルのプロパティ宣言に指定されていない場合は、このプロパティは必須です。</p>	無効時 (When_disabled)	条件付き

表 A-2 リソースプロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
Retry_interval (整数)	<p>失敗したリソースを再起動するまでの秒数。リソースモニターは、Retry_count と共にこのプロパティを使用します。このプロパティは、RGM のみが作成でき、RTR ファイルで宣言されている場合は、管理者は利用できます。デフォルト値が RTR ファイルで指定されている場合は、このプロパティは任意です。</p> <p>リソースタイプファイル内に Tunable 属性が指定されていない場合、このプロパティの Tunable 値は When_disabled になります。</p> <p>RTR ファイルのプロパティ宣言内に Default 属性が指定されていない場合、このプロパティは必須です。</p>	無効時 (When_disabled)	条件付き
Scalable (ブール値)	<p>リソースがスケラブルかどうかを示します。このプロパティが RTR ファイルで宣言されている場合は、そのタイプのリソースに対して、RGM は、次のスケラブルサービスプロパティを自動的に作成します。</p> <p>Network_resources_used, Port_list, Load_balancing_policy, Load_balancing_weights。これらのプロパティは、RTR ファイルで明示的に宣言されないかぎり、デフォルト値を持ちます。RTR ファイルで宣言されている場合、Scalable のデフォルトは True です。</p> <p>このプロパティが RTR ファイルで宣言されている場合、Tunable 属性は、At_creation (作成時) に設定する必要があります。設定しなければ、リソースの生成に失敗します。</p> <p>このプロパティが RTR ファイルで宣言されていない場合、リソースはスケラブルにはなりません。したがって、クラスタ管理者はこのプロパティを調整することができず、RGM はスケラブルサービスプロパティを設定しません。ただし、必要に応じて、明示的に Network_resources_used および Port_list プロパティを RTR ファイルで宣言できます。これらのプロパティは、スケラブルサービスだけでなく、非スケラブルサービスでも有用です。</p>	作成時 (At_creation)	任意

表 A-2 リソースプロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
各クラスタノードの Status (列挙)	リソースモニターによって設定されます。指定可能な値は、 <code>degraded</code> 、 <code>faulted</code> 、 <code>unknown</code> 、 <code>offline</code> です。RGM は、リソースがオンラインになると、値を <code>unknown</code> に設定し、オフラインになると <code>offline</code> に設定します。	不可 (None)	照会のみ
各クラスタノードの Status_msg (文字列)	リソースモニターによって、Status プロパティと同時に設定されます。このプロパティは、各ノードのリソースごとに設定可能です。RGM は、リソースがオフラインになると、このプロパティに空の文字列を設定します。	不可 (None)	照会のみ
Thorough_probe_interval (整数)	高オーバーヘッドのリソース障害検証の呼び出し間隔 (秒)。このプロパティは、RGM のみが作成でき、RTR ファイル内で宣言されている場合は、管理者は利用できません。デフォルト値が RTR ファイルで指定されている場合は、このプロパティは任意です。 リソースタイプファイル内に Tunable 属性が指定されていない場合、このプロパティの Tunable 値は <code>When_disabled</code> になります。 RTR ファイルのプロパティ宣言内に Default 属性が指定されていない場合、このプロパティは必須です。	無効時 (When_disabled)	条件付き
Type (文字列)	このリソースがインスタントであるリソースタイプ。	不可 (None)	必須
Type_version (文字列)	現在このリソースに関連付けられているリソースタイプのバージョンを指定します。このプロパティは RTR ファイル内に宣言できません。したがって、RGM によって自動的に作成されます。このプロパティの値は、リソースの型の <code>RT_version</code> プロパティと等しくなります。リソースの作成時、 <code>Type_version</code> プロパティはリソースタイプ名の接尾辞として表示されるだけで、明示的には指定されません。リソースを編集すると、 <code>Type_version</code> の値が変更されます。 次の項目から派生: <ul style="list-style-type: none"> ■ 現在のリソースタイプのバージョン ■ RTR ファイル内の <code>#\$upgrade_from</code> ディレクティブ 	説明を参照	説明を参照

表 A-2 リソースプロパティ (続き)

プロパティ名	説明	更新の可否	カテゴリ
UDP_affinity (ブール値)	<p>true の場合、指定のクライアントからの UDP トラフィックはすべて現在クライアントの TCP トラフィックを処理しているサーバーノードに送信されます。</p> <p>このプロパティは、Load_balancing_policy が Lb_sticky または Lb_sticky_wild の場合にかぎり有効です。さらに、Weak_affinity を False (デフォルト値) に設定する必要があります。</p> <p>このプロパティは、スケーラブルサービス専用です。</p>	無効時 (When_disabled)	任意
Weak_affinity (ブール値)	<p>true の場合、弱い形式のクライアントアフィニティが有効になります。これにより、指定のクライアントからの接続を同じサーバーノードに送信できます。ただし、次の場合は例外です。</p> <ul style="list-style-type: none"> ■ 障害モニターの再起動、リソースのフェイルオーバーまたはスイッチオーバー、障害発生後のノードのクラスタへの再接続などによるサーバーリスナーの起動時 ■ 管理アクションによるスケーラブルリソースの load_balancing_weights の変更時 <p>メモリーの消費とプロセッササイクルの点で、弱いアフィニティによるオーバーヘッドはデフォルトの形式よりも低くなります。</p> <p>このプロパティは、Load_balancing_policy が Lb_sticky または Lb_sticky_wild の場合にかぎり有効です。</p> <p>このプロパティは、スケーラブルサービス専用です。</p>	無効時 (When_disabled)	オプション

リソースグループプロパティ

以下の表に、Sun Cluster によって定義されるリソースグループプロパティを示します。プロパティ値は、次のように分類されます(「カテゴリ」列)。

- 必須 — 管理者は、管理ユーティリティでリソースグループを作成するときに、必ず値を指定する必要があります。
- 任意 — 管理者がリソースグループの作成時に値を指定しない場合、システムがデフォルト値を提供します。

- 照会のみ— 管理ツールから直接設定できません。

「更新の可否」列は、初期設定後に、そのプロパティが更新可能かどうかを示しています。

表 A-3 リソースグループプロパティ

プロパティ名	説明	更新の可否	概要
Auto_start_on_new_cluster (ブール値)	このプロパティを使用すると、新しいクラスタを形成するとき、Resource Group の自動起動を無効にすることができます。 デフォルトは TRUE です。TRUE の場合、クラスタが再起動するとき、Resource Group Manager はリソースグループを自動的に起動して、Desired primaries を有効にしようと試みます。FALSE に設定されている場合、クラスタのすべてのノードが同時に再起動したとき、Resource Group Manager はリソースグループを自動的に起動しません。	可	任意
Desired primaries (整数)	グループが同時にオンラインになることができるノードの数。 デフォルトは 1 です。RG_mode プロパティが Failover の場合、このプロパティの値を 1 より大きく設定することはできません。RG_mode プロパティが Scalable の場合は、1 より大きな値を設定できます。	可	任意
Failback (ブール値)	クラスタメンバーシップが変更されたとき、グループがオンラインになるノードセットを再計算するかどうかを指定するブール値。再計算によって、RGM はグループを優先度の低いノードでオフラインにし、優先度の高いノードでオンラインにします。 デフォルトは、False です。	可	任意
Global_resources_used (文字配列)	クラスタファイルシステムがこのリソースグループ内のリソースによって使用されるかどうかを指定します。管理者は、すべての広域リソース (アスタリスク記号 *) または広域リソースなし (空の文字列 "") に指定できます。 デフォルトでは、すべての広域リソースです。	可	任意

表 A-3 リソースグループプロパティ (続き)

プロパティ名	説明	更新の可否	概要
Implicit_network_dependencies (ブール値)	<p>True の場合に、グループ内のネットワークアドレスリソースに対し、非ネットワークアドレスリソースの暗黙の強い依存性を RGM が強制することを指定するブール値。ネットワークアドレスリソースには、論理ホスト名と共有アドレスリソースタイプが含まれます。</p> <p>スケーラブルリソースグループの場合、ネットワークアドレスリソースを含んでいないため、このプロパティは効果がありません。</p> <p>デフォルトは、True です。</p>	可	任意
Maximum primaries (整数)	<p>グループが同時にオンラインになることのできるノードの最大数。</p> <p>デフォルトは 1 です。RG_mode プロパティが Failover の場合、このプロパティの値を 1 より大きく設定することはできません。</p> <p>RG_mode プロパティが Scalable の場合は、1 より大きな値を設定できます。</p>	可	任意
Nodelist (文字配列)	<p>優先順位に従ってグループをオンラインにできるクラスタノードのリスト。これらのノードは、リソースグループの潜在的な主ノードまたはマスターです。</p> <p>デフォルトは、すべてのクラスタノードのリストになります。</p>	可	任意
Pathprefix (文字列)	<p>グループ内のリソースが書き込めるクラスタファイルシステムにあるディレクトリは、重要な管理ファイルを書き込みます。一部のリソースでは、このプロパティは必須です。各リソースグループの Pathprefix は、一意にする必要があります。</p> <p>デフォルトは空の文字列です。</p>	可	任意

表 A-3 リソースグループプロパティ (続き)

プロパティ名	説明	更新の可否	概要
Pingpong_interval (整数)	<p>再構成が生じた場合、あるいは <code>scha_control giveover</code> コマンドまたは関数の実行結果として、どのノードでリソースグループをオンラインにするかを判断するときに RGM が使用する負以外の整数値 (秒)。</p> <p>再構成において、リソースの <code>Start</code> または <code>Prenet_start</code> メソッドがゼロ以外の値で終了、またはタイムアウトによって終了したことが原因で、<code>Pingpong_interval</code> で指定した秒数内に、リソースグループをオンラインにするのを 2 回以上失敗した場合、RGM はそのノードはリソースグループのホストとして不適切だと判断し、別のマスターを探します。</p> <p>リソースの <code>scha_control(1ha)(3ha)</code> コマンドまたは関数の呼び出しによって、<code>Pingpong_interval</code> で指定した秒数内に特定のノード上でリソースグループがオフラインになった場合、別のノードから生じる後続の <code>scha_control</code> 呼び出しの結果、そのノードはリソースグループのホストとして不適切だと判断されます。</p> <p>デフォルト値は、3,600 秒 (1 時間) です。</p>	可	任意
Resource_list (文字配列)	<p>グループに含まれるリソースのリスト。管理者はこのプロパティを直接設定しません。このプロパティは、管理者がリソースグループにリソースを追加したり、リソースを削除したときに、RGM によって更新されます。</p> <p>デフォルトは、空のリストです。</p>	不可	照会のみ
RG_dependencies (文字配列)	<p>同じノード上の別のグループをオンライン/オフラインにするときの優先順位を示すリソースグループのリスト (任意)。別のノードでグループをオンラインにする場合は、このリストは無効です。</p> <p>デフォルトは、空のリストです。</p>	可	任意
RG_description (文字列)	<p>リソースグループの簡単な説明。</p> <p>デフォルトは空の文字列です。</p>	可	任意

表 A-3 リソースグループプロパティ (続き)

プロパティ名	説明	更新の可否	概要
RG_mode (列挙)	<p>リソースグループがフェイルオーバーグループなのか、スケーラブルグループなのかを指定します。このプロパティの値が Failover の場合、RGM はグループの Maximum primaries プロパティを 1 に設定し、そのリソースグループをマスターするのを単一のノードに制限します。</p> <p>このプロパティの値が Scalable の場合、RGM は Maximum primaries プロパティが 1 より大きい値を持つことを許可し、複数のノードで同時にそのグループをマスターできるようにします。RGM は、RG-mode が Scalable に設定されているリソースグループに、Failover プロパティが True に設定されているリソースを追加することを許可しません。</p> <p>Maximum primaries に 1 が設定されている場合のデフォルトは、Failover です。 Maximum primaries に 2 以上が設定されている場合のデフォルトは、Scalable です。</p>	不可	任意
RG_name (文字列)	リソースグループの名前。クラスタ内で一意にする必要があります。	不可	必須
RG_project_name (文字列)	<p>リソースグループに関連付けられた Solaris プロジェクト名。このプロパティは、CPU の共有、クラスタデータサービスのリソースプールといった Solaris のリソース管理機能に適用できます。RGM は、リソースグループをオンラインにするとき、Resource project_name プロパティセットを持たないリソースに対して、このプロジェクトで関連付けられたプロセスを起動します。プロジェクトデータベース内に存在するプロジェクト名を指定する必要があります。また、root ユーザーは、このプロジェクトのメンバーとして構成されている必要があります。</p> <p>このプロパティは Solaris 9 以降でサポートされます。</p> <p>注 - このプロパティへの変更を有効にするには、リソースを再起動する必要があります。</p>	任意の時点	必須

表 A-3 リソースグループプロパティ (続き)

プロパティ名	説明	更新の可否	概要
各クラスタノードの RG_state (列挙)	<p>RGM によって Online、Offline、Pending_online、Pending_offline、Error_stop_failed に設定され、各クラスタノード上のグループの状態を示します。グループが RGM の制御下でない場合は、非管理状態で存在できます。</p> <p>このプロパティは、ユーザーは構成できません。</p> <p>デフォルトは、Offline です。</p>	不可	照会のみ

リソースプロパティの属性

以下の表に、システム定義のプロパティの変更または拡張プロパティの作成に使用できるリソースプロパティ属性を一覧します。



注意 - boolean、enum、int タイプのデフォルト値に、NULL または空の文字列 ("") は指定できません。

表 A-4 リソースプロパティの属性

プロパティ	説明
Property	リソースプロパティの名前。
Extension	このプロパティを使用すると、RTR ファイルのエントリで、リソースタイプの実装によって定義された拡張プロパティが宣言されていることを示します。使用されない場合は、そのエントリはシステム定義プロパティです。
Description	プロパティを簡潔に記述した注記 (文字列)。RTR ファイル内でシステム定義プロパティに対する Description 属性を設定することはできません。
プロパティのタイプ	指定可能なタイプは、string、boolean、int、enum、stringarray です。RTR ファイル内で、システム定義プロパティに対するタイプ属性を設定することはできません。タイプは、RTR ファイルのエントリに登録できる、指定可能なプロパティ値とタイプ固有の属性を決定します。enum タイプは、文字列値のセットです。
Default	プロパティのデフォルト値を示します。

表 A-4 リソースプロパティの属性 (続き)

プロパティ	説明
Tunable	<p>クラスタ管理者が、リソースのプロパティ値をいつ設定できるかを示します。管理者がプロパティを設定できないようにするには、None または False に設定します。管理者にプロパティの調整を許可する属性値は、次のとおりです。True または Anytime (任意の時点)、At_creation (リソースの作成時のみ)、When_disabled (リソースがオフラインのとき)。</p> <p>デフォルトは、True (Anytime) です。</p>
Enumlist	enum タイプの場合、プロパティに設定できる文字列値のセット。
Min	int タイプの場合、プロパティに設定できる最小値。
Max	int タイプの場合、プロパティに設定できる最大値。
Minlength	string および stringarray タイプの場合、設定できる文字列の長さの最小値。
Maxlength	string および stringarray タイプの場合、設定できる文字列の長さの最大値。
Array_minsize	stringarray タイプの場合、設定できる配列要素の最小数。
Array_maxsize	stringarray タイプの場合、設定できる配列要素の最大数。

付録 B

データサービスのコード例

この付録では、データサービスの各メソッドの完全なコード例を示します。また、リソースタイプ登録 (RTR) ファイルの内容も示します。

この付録に含まれるコードリストは、次のとおりです。

- 219 ページの「リソースタイプ登録ファイルのリスト」
- 222 ページの「Start メソッドのコードリスト」
- 225 ページの「Stop メソッドのコードリスト」
- 227 ページの「gettime ユーティリティのコードリスト」
- 228 ページの「PROBE プログラムのコードリスト」
- 234 ページの「Monitor_start メソッドのコードリスト」
- 236 ページの「Monitor_stop メソッドのコードリスト」
- 238 ページの「Monitor_check メソッドのコードリスト」
- 240 ページの「Validate メソッドのコードリスト」
- 244 ページの「Update メソッドのコードリスト」

リソースタイプ登録ファイルのリスト

リソースタイプ登録 (RTR) ファイルには、クラスタ管理者がデータサービスを登録するとき、データサービスの初期構成を定義するリソースとリソースタイプのプロパティ宣言が含まれています。

例 B-1 SUNW.Sample RTR ファイル

```
#
# Copyright (c) 1998-2000 by Sun Microsystems, Inc.
# All rights reserved.
#
# ドメインネームサービス (DNS) の登録情報
#
```

例 B-1 SUNW.Sample RTR ファイル (続き)

```
#pragma ident    "@(#)SUNW.sample  1.1  00/05/24 SMI"

RESOURCE_TYPE = "sample";
VENDOR_ID = SUNW;
RT_DESCRIPTION = "Domain Name Service on Sun Cluster";

RT_VERSION = "1.0";
API_VERSION = 2;
FAILOVER = TRUE;

RT_BASEDIR=/opt/SUNWsample/bin;
PKGLIST = SUNWsample;

START          = dns_svc_start;
STOP           = dns_svc_stop;

VALIDATE       = dns_validate;
UPDATE        = dns_update;

MONITOR_START  = dns_monitor_start;
MONITOR_STOP   = dns_monitor_stop;
MONITOR_CHECK  = dns_monitor_check;

# リソースタイプ宣言の後に、中括弧に囲まれたリソースプロパティ宣言のリスト
# が続く。プロパティ名宣言は、各エントリの左中括弧の直後にある最初
# の属性である必要がある。
#
# <method>_timeout プロパティは、RGM がメソッドの呼び出しが失敗
# したという結論を下すまでの時間 (秒) を設定する。
# すべてのメソッドタイムアウトの MIN 値は 60 秒に設定されている。こ
# れは、管理者が短すぎる時間を設定することを防ぐためである。短すぎ
# る時間を設定すると、スイッチオーバーやフェイルオーバーの性能が上
# がらず、さらには、予期せぬ RGM アクションが発生する可能性がある
# (間違ったフェイルオーバー、ノードの再起動、リソースグループの
# ERROR_STOP_FAILED 状態への移行、オペレータの介入の必要性など)。
# メソッドタイムアウトに短すぎる時間を設定すると、データサービス全
# 体の可用性を下げることになる。
{
    PROPERTY = Start_timeout;
    MIN=60;
    DEFAULT=300;
}

{
    PROPERTY = Stop_timeout;
    MIN=60;
    DEFAULT=300;
}

{
    PROPERTY = Validate_timeout;
    MIN=60;
    DEFAULT=300;
}
```

例 B-1 SUNW.Sample RTR ファイル (続き)

```
}
{
    PROPERTY = Update_timeout;
    MIN=60;
    DEFAULT=300;
}
{
    PROPERTY = Monitor_Start_timeout;
    MIN=60;
    DEFAULT=300;
}
{
    PROPERTY = Monitor_Stop_timeout;
    MIN=60;
    DEFAULT=300;
}
{
    PROPERTY = Thorough_Probe_Interval;
    MIN=1;
    MAX=3600;
    DEFAULT=60;
    TUNABLE = ANYTIME;
}

# 当該ノード上でアプリケーションを正常に起動できないと結論を下すま
# でに、ある期間 (Retry_Interval) に行う再試行の回数。
{
    PROPERTY = Retry_Count;
    MIN=0;
    MAX=10;
    DEFAULT=2;
    TUNABLE = ANYTIME;
}

# Retry_Interval には 60 の倍数を設定する。これは、秒から分に変換さ
# れ、端数が切り上げられるためである。たとえば、50 (秒) という値を
# 指定すると、1 分に変換される。
# このプロパティは再試行数 (Retry_Count) のタイミングを決定する。
{
    PROPERTY = Retry_Interval;
    MIN=60;
    MAX=3600;
    DEFAULT=300;
    TUNABLE = ANYTIME;
}

{
    PROPERTY = Network_resources_used;
    TUNABLE = AT_CREATION;
    DEFAULT = "";
}

#
# 拡張プロパティ
```

例 B-1 SUNW.Sample RTR ファイル (続き)

```
# クラスタ管理者はこのプロパティの値を設定して、アプリケーションが使用
# する構成ファイルが入っているディレクトリを示す必要がある。このアプリ
# ケーションの場合、DNS は PXFS (通常は named.conf) 上の DNS 構成ファイ
# ルのパスを指定する。
{
    PROPERTY = Confdir;
    EXTENSION;
    STRING;
    TUNABLE = AT_CREATION;
    DESCRIPTION = "The Configuration Directory Path";
}

# 検証が失敗したと宣言するまでのタイムアウト値 (秒)。
{
    PROPERTY = Probe_timeout;
    EXTENSION;
    INT;
    DEFAULT = 30;
    TUNABLE = ANYTIME;
    DESCRIPTION = "Time out value for the probe (seconds)";
}
```

Start メソッドのコードリスト

データサービスリソースを含むリソースグループがオンラインになったとき、あるいは、リソースが有効になったとき、RGM はそのクラスタノード上で start メソッドを呼び出します。サンプルのアプリケーションでは、メソッドはそのノード上で in.named (DNS) デーモンを起動します。

例 B-2 dns_svc_start メソッド

```
#!/bin/ksh
#
# HA-DNS の START メソッド
#
# このメソッドは PMF の制御下でデータサービスを起動する。DNS の
# in.named プロセスを起動する前に、いくつかの妥当性検査を実行する。
#
# データサービスの PMF タグは $RESOURCE_NAME.named である。PMF は、
# 指定された回数 (Retry_count) だけ、サービスを起動しようとする。そ
# して、指定された期間 (Retry_interval) 内で試行回数がこの値を超えた
# 場合、PMF はサービスの起動に失敗したことを報告する。
# Retry_count と Retry_interval は両方とも RTR ファイルに設定されて
# いるリソースプロパティである。
#pragma ident    "@(#)dns_svc_start    1.1    00/05/24 SMI"

#####
```

例 B-2 dns_svc_start メソッド (続き)

```
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
    typeset opt

    while getopts `R:G:T:` opt
    do
        case "$opt" in
            R)
                # DNS リソース名。
                RESOURCE_NAME=$OPTARG
                ;;
            G)
                # リソースが構成されているリソース
                # グループの名前。
                RESOURCEGROUP_NAME=$OPTARG
                ;;
            T)
                # リソースタイプ名。
                RESOURCETYPE_NAME=$OPTARG
                ;;
            *)
                logger -p ${SYSLOG_FACILITY}.err \
                -t [${RESOURCETYPE_NAME},${RESOURCEGROUP_NAME},${RESOURCE_NAME}]
                \
                "ERROR: Option $OPTARG unknown"
                exit 1
                ;;
        esac
    done
}

#####
# MAIN
#
#####

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能番号を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=${RESOURCE_NAME}.named
```

例 B-2 dns_svc_start メソッド (続き)

```
SYSLOG_TAG=$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# DNS を起動するため、リソースの Confdir プロパティの値を取得する。
# 入力されたリソース名とリソースグループを使用して、リソースを
# 追加するときにクラスタ管理者が設定した Confdir の値を見つける。
#
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME
-G $RESOURCEGROUP_NAME Confdir`
# scha_resource_get は拡張プロパティの「タイプ」と「値」を戻す。
# 拡張プロパティの値だけを取得する。
CONFIG_DIR=`echo $config_info | awk '{print $2}'`

# $CONFIG_DIR がアクセス可能であるかどうかを検査する。
if [ ! -d $CONFIG_DIR ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [SYSLOG_TAG] \
        "${ARGV0} Directory $CONFIG_DIR missing or not mounted"
    exit 1
fi

# データファイルへの相対パス名が存在する場合、$CONFIG_DIR
# ディレクトリに移動する。
cd $CONFIG_DIR

# named.conf ファイルが $CONFIG_DIR ディレクトリ内に存在するかどうか
# を検査する。
if [ ! -s named.conf ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [SYSLOG_TAG] \
        "${ARGV0} File $CONFIG_DIR/named.conf is missing or
empty"
    exit 1
fi

# RTR ファイルから Retry_count の値を取得する。
RETRY_CNT=`scha_resource_get -O Retry_Count -R $RESOURCE_NAME
-G \ $RESOURCEGROUP_NAME`

# RTR ファイルから Retry_interval の値を取得する。この値の単位は秒
# であり、pmfadm に渡すときは分に変換する必要がある。変換時、端数は
# 切り上げられるので注意すること。たとえば、50 秒は 1 分に切り上げられる。
((RETRY_INTRVAL = `scha_resource_get -O Retry_Interval
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME` / 60))

# PMF の制御下で in.named デーモンを起動する。$RETRY_INTERVAL の期
# 間、$RETRY_COUNT の回数だけ、クラッシュおよび再起動できる。どちら
# かの値以上クラッシュした場合、PMF は再起動を中止する。
# <$PMF_TAG> というタグですでにプロセスが登録されて
# いる場合、PMF はすでにプロセスが動作していることを示す警告メッセ
# ージを送信する。
echo "Retry interval is "$RETRY_INTRVAL
pmfadm -c $PMF_TAG.named -n $RETRY_CNT -t $RETRY_INTRVAL \
    /usr/sbin/in.named -c named.conf

# HA-DNS が起動していることを示すメッセージを記録する。
```

例 B-2 dns_svc_start メソッド (続き)

```
if [ $? -eq 0 ]; then
    logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG]\
        "${ARGV0} HA-DNS successfully started"
fi
exit 0
```

Stop メソッドのコードリスト

HA-DNS リソースを含むリソースグループがクラスタノード上でオフラインになるとき、あるいは、HA-DNS リソースが無効になるとき、RGM は Stop メソッドを呼び出します。このメソッドは、そのノード上で in.named (DNS) デーモンを停止します。

例 B-3 dns_svc_stop メソッド

```
#!/bin/ksh
#
# HA-DNS の Stop メソッド
#
# このメソッドは、PMF を使用するデータサービスを停止する。サービス
# が動作していない場合、このメソッドは状態 0 で終了する。その他の値
# は戻さない。リソースは STOP_FAILED 状態になる。

#pragma ident    "@(#)dns_svc_stop  1.1  00/05/24 SMI"

#####
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
    typeset opt

    while getopts `R:G:T:` opt
    do
        case "$opt" in
            R)
                # DNS リソースの名前。
                RESOURCE_NAME=$OPTARG
                ;;
            G)
                # リソースが構成されているリソース
                # グループの名前。
                RESOURCEGROUP_NAME=$OPTARG
                ;;
            T)

```

例 B-3 dns_svc_stop メソッド (続き)

```
        # リソースタイプの名前。
        RESOURCETYPE_NAME=$OPTARG
        ;;

*)
    logger -p ${SYSLOG_FACILITY}.err \
    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
    "ERROR: Option $OPTARG unknown"
    exit 1
    ;;

esac

done

}

#####
# MAIN
#
#####

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能番号を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.named
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# RTR ファイルから Stop_timeout 値を取得する。
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT -R $RESOURCE_NAME
-G \ $RESOURCEGROUP_NAME`

# PMF 経由で SIGTERM シグナルを使用する規則正しい方法でデータサービ
# スを停止しようとする。SIGTERM がデータサービスを停止できるまで、
# Stop_timeout 値の 80% だけ待つ。停止できない場合、SIGKILL を送信
# して、データサービスを停止しようとする。SIGKILL がデータサービス
# を停止できるまで、Stop_timeout 値の 15% だけ待つ。停止できない場合、
# メソッドは何か異常があったと判断し、0 以外の状態で終了する。
# Stop_timeout の残りの 5% は他の目的のために予約されている。
((SMOOTH_TIMEOUT=$STOP_TIMEOUT * 80/100))

((HARD_TIMEOUT=$STOP_TIMEOUT * 15/100))

# in.named が動作しているかどうかを調べて、動作していれば停止する。
if pmfadm -q $PMF_TAG.named; then
    # SIGTERM シグナルをデータサービスに送信して、合計タイムアウト値
    # の 80% だけ待つ。

```

例 B-3 dns_svc_stop メソッド (続き)

```
pmfadm -s $PMF_TAG.named -w $SMOOTH_TIMEOUT TERM
if [ $? -ne 0 ]; then
    logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
        "${ARGV0} Failed to stop HA-DNS with SIGTERM; Retry
with \
    SIGKILL"

    # SIGTERM シグナルでデータサービスが停止しないので、今度は
    # SIGKILL を使用して、合計タイムアウト値の 15% だけ待つ。
    pmfadm -s $PMF_TAG.named -w $HARD_TIMEOUT KILL
    if [ $? -ne 0 ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [SYSLOG_TAG]
        \
            "${ARGV0} Failed to stop HA-DNS; Exiting UNSUCCESSFUL"

        exit 1
    fi
fi
else
    # この時点でデータサービスは動作していない。メッセージを記録して、
    # 成功で終了する。
    logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
        "HA-DNS is not started"

    # HA-DNS が動作していない場合でも、成功で終了し、データサービス
    # リソースが STOP_FAILED 状態にならないようにする。

    exit 0
fi

# DNS の停止に成功。メッセージを記録して、成功で終了する。
logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
    "HA-DNS successfully stopped"
exit 0
```

gettime ユーティリティのコードリスト

gettime ユーティリティは、検証の再起動間の経過時間を PROBE プログラムが追跡するための C プログラムです。このプログラムは、コンパイル後、コールバックメソッドと同じディレクトリ (RT_basedir プロパティが指すディレクトリ) に格納する必要があります。

例 B-4 gettimeofday.c ユーティリティープログラム

```
#
# このユーティリティープログラムは、データサービスの検証メソッドによ
# って使用され、既知の参照ポイント(基準点)からの経過時間(秒)を
# 追跡する。このプログラムは、コンパイル後、データサービスのコール
# バックメソッドと同じディレクトリ(RT_basedir)に格納しておくこと。
```

```
#pragma ident    "@(#)gettime.c    1.1    00/05/24 SMI"
```

```
#include <stdio.h>
#include <sys/types.h>
#include <time.h>
```

```
main()
{
    printf("%d\n", time(0));
    exit(0);
}
```

PROBE プログラムのコードリスト

PROBE プログラムは、nslookup(1M) コマンドを使用して、データサービスの可用性を検査します。このプログラムは、Monitor_start コールバックメソッドによって起動され、Monitor_stop コールバックメソッドによって停止されます。

例 B-5 dns_probe プログラム

```
#!/bin/ksh
#pragma ident    "@(#)dns_probe    1.1    00/04/19 SMI"
#
# HA-DNS の Probe メソッド
#
# このプログラムは、nslookup を使用して、データサービスの可用性を検査
# する。nslookup は DNS サーバーに照会することによって、DNS
# サーバー自身を探す。サーバーが応答しない場合、あるいは、別のサー
# バーが照会に応答した場合、probe メソッドはデータサービスまたはク
# ラスタ内の別のノードになんらかの問題が発生したという結論を下す。
# 検証は、RTR ファイルの THOROUGH_PROBE_INTERVAL で設定さ
# れた間隔で行われる。
```

```
#pragma ident    "@(#)dns_probe    1.1    00/05/24 SMI"
```

```
#####
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
```

例 B-5 dns_probe プログラム (続き)

```
{
    typeset opt

    while getopts `R:G:T:` opt
    do
        case "$opt" in
            R)
                # DNS リソースの名前。
                RESOURCE_NAME=$OPTARG
                ;;
            G)
                # リソースが構成されているリソース
                # グループの名前。
                RESOURCEGROUP_NAME=$OPTARG
                ;;
            T)
                # リソースタイプの名前。
                RESOURCETYPE_NAME=$OPTARG
                ;;
            *)
                logger -p ${SYSLOG_FACILITY}.err \
                -t [$RESOURCECETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
                \
                "ERROR: Option $OPTARG unknown"
                exit 1
                ;;
        esac
    done
}

#####
# restart_service ()
#
# この関数は、まずデータサービスの stop メソッドを呼び出し、
# 次に start メソッドを呼び出すことによって、データサービスを再起動
# しようとする。データサービスがすでに起動しておらず、
# データサービスのタグが PMF に登録されていない場合、
# この関数はデータサービスをクラスタ内の
# 別のノードにフェイルオーバーする。
#
function restart_service
{
    # データサービスを再起動するには、まず、データサービス自身が
    # PMF 下に登録されているかどうかを確認する。
    pmfadm -q $PMF_TAG
    if [[ $? -eq 0 ]]; then
        # データサービスの TAG が PMF に登録されている場合、
        # データサービスを停止し、起動し直す。
    fi
}
```

例 B-5 dns_probe プログラム (続き)

```
# 当該リソースの stop メソッド名と STOP_TIMEOUT 値を取得する。
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT
\
        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
STOP_METHOD=`scha_resource_get -O STOP
\
        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
hatimerun -t $STOP_TIMEOUT $RT_BASEDIR/$STOP_METHOD
\
        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\
        -T $RESOURCETYPE_NAME

if [[ $? -ne 0 ]]; then
    logger-p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}
\
        "${ARGV0} Stop method failed."
    return 1
fi

# 当該リソースの start メソッド名と START_TIMEOUT 値を取得する。
START_TIMEOUT=`scha_resource_get -O START_TIMEOUT
\
        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
START_METHOD=`scha_resource_get -O START
\
        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD
\
        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\
        -T $RESOURCETYPE_NAME

if [[ $? -ne 0 ]]; then
    logger-p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}
\
        "${ARGV0} Start method
failed."
    return 1
fi

else
    # データサービスの TAG が PMF に登録されていない場合、
    # データサービスが PMF 下で許可されている再試行最大回数を
    # 超えていることを示す。したがって、データサービスを再起動
    # してはならない。その代わりに、同じクラス内にある別のノード
    # にフェイルオーバーを試みる。
    scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME
\
        -R $RESOURCE_NAME
fi
```

例 B-5 dns_probe プログラム (続き)

```
        return 0
    }

#####
# decide_restart_or_failover ()
#
# この関数は、検証が失敗したときに行うべきアクション、つまり、デー
# タサービスをローカルで再起動するか、クラスタ内の別のノードに
# フェイルオーバーするかを決定する。
#
# 最初の再起動の試行であるかどうかを検査する。
if [ $retries -eq 0 ]; then
    # 最初の失敗である。最初の試行の時刻を記録する。
    start_time=`$RT_BASEDIR/gettimè
    retries=`expr $retries + 1`
    # 最初の失敗であるので、データサービスを再起動しようと試行する。
    restart_service
    if [ $? -ne 0 ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}] \
            "${ARGV0} Failed to restart data service."
        exit 1
    fi
else
    # 最初のではない。
    current_time=`$RT_BASEDIR/gettimè
    time_diff=`expr $current_time - $start_time
    if [ $time_diff -ge $RETRY_INTERVAL ]; then
        # この失敗は再試行最大期間後に発生した。
        # したがって、再試行カウンタをリセットし、
        # 再試行時間をリセットし、さらに再試行する。
        retries=1
        start_time=$current_time
        # 前回の失敗が Retry_interval よりも以前に発生しているので、
        # データサービスを再起動しようと試行する。
        restart_service
        if [ $? -ne 0 ]; then
            logger -p ${SYSLOG_FACILITY}.err \
                -t [${SYSLOG_TAG}
                "${ARGV0} Failed to restart HA-DNS."
            exit 1
        fi
    elif [ $retries -ge $RETRY_COUNT ]; then
        # 再試行最大期間内であり、再試行カウンタは満了
        # している。したがって、フェイルオーバーする。
        retries=0
        scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME \
            -R $RESOURCE_NAME
        if [ $? -ne 0 ]; then
```

例 B-5 dns_probe プログラム (続き)

```
        logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}] \  
            "${ARGV0} Failover attempt failed."\  
        exit 1\  
    fi\  
else\  
    # 再試行最大期間内であり、再試行カウンタは満了\  
    # していない。したがって、さらに再試行する。\  
    retries=`expr $retries + 1`\  
    restart_service\  
    if [ $? -ne 0 ]; then\  
        logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}] \  
            "${ARGV0} Failed to restart HA-DNS."\  
        exit 1\  
    fi\  
fi\  
fi\  
}  
  
#####  
# MAIN  
#####  
  
export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH  
  
# メッセージの記録に使用する syslog 機能番号を取得する。\  
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`  
  
# このメソッドに渡された引数を構文解析する。\  
parse_args "$@"  
  
PMF_TAG=$RESOURCE_NAME.named\  
SYSLOG_TAG=$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME  
  
# 検証が行われる間隔はシステム定義プロパティ THOROUGH_PROBE_INTERVAL  
# に設定されている。scha_resource_get でこのプロパティの値を取得する。\  
PROBE_INTERVAL=`scha_resource_get -O THOROUGH_PROBE_INTERVAL  
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`  
  
# 検証用のタイムアウト値を取得する。この値は RTR ファイルの  
# PROBE_TIMEOUT 拡張プロパティに設定されている。nslookup のデフォル  
# トのタイムアウトは 1.5 分。\  
probe_timeout_info=`scha_resource_get -O Extension -R $RESOURCE_NAME  
-G \${RESOURCEGROUP_NAME} Probe_timeout`\  
PROBE_TIMEOUT=`echo $probe_timeout_info | awk '{print $2}'`  
  
# リソースの NETWORK_RESOURCES_USED プロパティの値を取得して、  
# DNS がサービスを提供するサーバーを見つける。\  
DNS_HOST=`scha_resource_get -O NETWORK_RESOURCES_USED -R  
$RESOURCE_NAME -G \${RESOURCEGROUP_NAME}`  
  
# システム定義プロパティ Retry_count から再試行最大回数を取得する。\  
RETRY_COUNT=`scha_resource_get -O RETRY_COUNT -R $RESOURCE_NAME`
```

例 B-5 dns_probe プログラム (続き)

```
-G \${RESOURCEGROUP_NAME}

# システム定義プロパティ Retry_interval から再試行最大期間を取得する。
RETRY_INTERVAL=`scha_resource_get -O RETRY_INTERVAL -R
${RESOURCE_NAME} -G \${RESOURCEGROUP_NAME}`

# リソースタイプの RT_basedir プロパティから gettime コーティリティーの
# 完全パスを取得する。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R ${RESOURCE_NAME}
-G \${RESOURCEGROUP_NAME}`

# 検証は無限ループで動作し、nslookup コマンドを実行し続ける。
# nslookup 応答用の一時ファイルを設定する。
DNSPROBEFILE=/tmp/.${RESOURCE_NAME}.probe
probfail=0
retries=0

while :
do
# 検証が動作すべき期間は THOROUGH_PROBE_INTERVAL プロパティに指
# 定されている。したがって、THOROUGH_PROBE_INTERVAL の間、検証
# プログラムが休眠するように設定する。
sleep ${PROBE_INTERVAL}

# DNS がサービスを提供している IP アドレス上で nslookup コマンド
# を実行する。
hatimerun -t ${PROBE_TIMEOUT} /usr/sbin/nslookup ${DNS_HOST} ${DNS_HOST}
\
> ${DNSPROBEFILE} 2>&1

retcode=$?
if [ retcode -ne 0 ]; then
    probfail=1
fi

# nslookup への応答が HA-DNS サーバーから来ており、
# /etc/resolv.conf ファイル内に指定されている他のネームサーバー
# から来ていないことを確認する。
if [ $probfail -eq 0 ]; then
    # nslookup 照会に回答したサーバーの名前を取得する。
    SERVER=`awk ` $1=="Server:" {
print $2 }' \
        ${DNSPROBEFILE} | awk -F. ` { print $1 } ` `
    if [ -z "$SERVER" ];
then
        probfail=1
    else
        if [ $SERVER != $DNS_HOST ]; then
            probfail=1
        fi
    fi
fi
```

例 B-5 dns_probe プログラム (続き)

```
# probefail 変数が 0 以外である場合、nslookup コマンドがタイム
# アウトしたか、あるいは、別のサーバー (/etc/resolv.conf ファイ
# ルに指定されている) から照会への応答が来ていることを示す。
# どちらの場合でも、DNS サーバーは応答していないので、
# このメソッドは decide_restart_or_failover を呼び出して、
# データサービスをローカルで起動するか、あるいは、別のノードに
# フェイルオーバーするかを評価する。

if [ $probefail -ne 0 ]; then
    decide_restart_or_failover
else
    logger -p ${SYSLOG_FACILITY}.info -t [${SYSLOG_TAG}]\
        "${ARGV0} Probe for resource HA-DNS successful"
fi
done
```

Monitor_start メソッドのコードリス ト

このメソッドは、データサービスの PROBE プログラムを起動します。

例 B-6 dns_monitor_start メソッド

```
#!/bin/ksh
#
# HA-DNS の Monitor_start メソッド
#
# このメソッドは、PMF の制御下でデータサービスのモニター(検証) を
# 起動する。モニターは一定の間隔でデータサービスを検証するプロセス
# で、問題が発生すると、データサービスを同じノード上で再起動するか、
# クラスタ内の別のノードにフェイルオーバーする。モニター用の PMF
# タグは $RESOURCE_NAME.monitor。

#pragma ident    "@(#)dns_monitor_start    1.1    00/05/24 SMI"

#####
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
    typeset opt

    while getopts `R:G:T:` opt
    do
        case "$opt" in
```

例 B-6 dns_monitor_start メソッド (続き)

```

R)
    # DNS リソースの名前。
    RESOURCE_NAME=$OPTARG
    ;;
G)
    # リソースが構成されているリソース
    # グループの名前。
    RESOURCEGROUP_NAME=$OPTARG
    ;;
T)
    # リソースタイプの名前。
    RESOURCETYPE_NAME=$OPTARG
    ;;

*)
    logger -p ${SYSLOG_FACILITY}.err \
        -t [${RESOURCETYPE_NAME},${RESOURCEGROUP_NAME},${RESOURCE_NAME}]
    \
        "ERROR: Option $OPTARG unknown"
        exit 1
        ;;
esac
done
}

#####
# MAIN
#
#####

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCETYPE_NAME, $RESOURCEGROUP_NAME, $RESOURCE_NAME

# データサービスの RT_BASEDIR プロパティを取得することによって、検
# 証メソッドが存在する場所を見つける。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME
-G \ $RESOURCEGROUP_NAME`

# PMF の制御下でデータサービスの検証を開始する。無限再試行オプション
# を使用して検証メソッドを起動する。リソースの名前、タイプ、および
# グループを検証メソッドに渡す。
pmfadm -c $PMF_TAG.monitor -n -l -t -l \

```

例 B-6 dns_monitor_start メソッド (続き)

```
$RT_BASEDIR/dns_probe -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\  
-T $RESOURCETYPE_NAME

# HA-DNS のモニターが起動されたことを示すメッセージを記録する。
if [ $? -eq 0 ]; then
    logger -p ${SYSLOG_FACILITY}.info -t [${SYSLOG_TAG}] \  
        "${ARGV0} Monitor for HA-DNS successfully started"
fi
exit 0
```

Monitor_stop メソッドのコードリスト

このメソッドは、データサービスの PROBE プログラムを停止します。

例 B-7 dns_monitor_stop メソッド

```
#!/bin/ksh
#
# HA-DNS の Monitor_stop メソッド
#
# PMF を使用して動作しているモニターを停止する。

#pragma ident    "@(#)dns_monitor_stop    1.1    00/05/24 SMI"

#####
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
    typeset opt

    while getopts `R:G:T:` opt
    do
        case "$opt" in
            R)
                # DNS リソースの名前。
                RESOURCE_NAME=$OPTARG
                ;;
            G)
                # リソースが構成されているリソース
                # グループの名前。

```

例 B-7 dns_monitor_stop メソッド (続き)

```

        RESOURCEGROUP_NAME=$OPTARG
        ;;
T)
    # リソースタイプの名前。
    RESOURCETYPE_NAME=$OPTARG
    ;;
*)
    logger -p ${SYSLOG_FACILITY}.err \
    -t [$RESOURCECETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
    "ERROR: Option $OPTARG unknown"
    exit 1
    ;;
esac
done
}

#####
# MAIN
#
#####

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCECETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# モニターが動作しているかどうかを調べて、動作していれば停止する。
if pmfadm -q $PMF_TAG.monitor; then
    pmfadm -s $PMF_TAG.monitor KILL
    if [ $? -ne 0 ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}] \
        "${ARGV0} Could not stop monitor for resource " \
        $RESOURCE_NAME
        exit 1
    else
        # モニターは正常に停止している。メッセージを記録する。
        logger -p ${SYSLOG_FACILITY}.info -t [${SYSLOG_TAG}] \
        "${ARGV0} Monitor for resource " $RESOURCE_NAME
\
        " successfully stopped"
    fi
fi

```

例 B-7 dns_monitor_stop メソッド (続き)

```
exit 0
```

Monitor_check メソッドのコードリスト

このメソッドは、Confdir プロパティが示すディレクトリの存在を確認します。PROBE メソッドがデータサービスを新しいノードにフェイルオーバーするとき、RGM は Monitor_check を呼び出し、また、潜在マスターであるノードを検査します。

例 B-8 dns_monitor_check メソッド

```
#!/bin/ksh
#
# DNS の Monitor_check メソッド
#
# 障害モニターがデータサービスを新しいノードにフェイルオーバー
# するとき、RGM はこのメソッドを呼び出す。Monitor_check は
# Validate メソッドを呼び出して、新しいノード上で構成ディレク
# トリおよびファイルが利用できるかどうかを確認する。

#pragma ident  "@(#)dns_monitor_check 1.1 00/05/24 SMI"

#####
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
    typeset opt

    while getopts `R:G:T:` opt
    do
        case "$opt" in

            R)
                # DNS リソースの名前。
                RESOURCE_NAME=$OPTARG
                ;;

            G)
                # リソースが構成されているリソースグループの名前。
                RESOURCEGROUP_NAME=$OPTARG
                ;;
        esac
    done
}
```

例 B-8 dns_monitor_check メソッド (続き)

```

T)
# リソースタイプの名前。
RESOURCE_TYPE_NAME=$OPTARG
;;

*)
logger -p ${SYSLOG_FACILITY}.err \
-t [$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
"ERROR: Option $OPTARG unknown"
exit 1
;;

esac
done

}

#####
# MAIN
#####

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.named
SYSLOG_TAG=$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# リソースタイプの RT_BASEDIR プロパティから Validate メソッドの完全パスを
# 取得する。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME
\
-G $RESOURCEGROUP_NAME`

# 当該リソースの Validate メソッド名を取得する。
VALIDATE_METHOD=`scha_resource_get -O VALIDATE \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`

# データサービスを起動するための Confdir プロパティの値を取得する。
# 入力されたリソース名とリソースグループを使用して、リソースを
# 追加するときに設定した Confdir の値を取得する。
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME
-G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get は、拡張プロパティの値とともにタイプも戻す。
# awk を使用して、拡張プロパティの値だけを取得する。
CONFIG_DIR=`echo $config_info | awk '{print $2}'`

```

例 B-8 dns_monitor_check メソッド (続き)

```
# Validate メソッドを呼び出して、データサービスを新しいノードに
# フェイルオーバーできるかどうかを確認する。
$RT_BASEDIR/$VALIDATE_METHOD -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\
  -T $RESOURCE_TYPE_NAME -x Confdir=$CONFIG_DIR

# モニター検査が成功したことを示すメッセージを記録する。
if [ $? -eq 0 ]; then
  logger -p ${SYSLOG_FACILITY}.info -t [${SYSLOG_TAG}] \
    "${ARGV0} Monitor check for DNS successful."
  exit 0
else
  logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}] \
    "${ARGV0} Monitor check for DNS not successful."
  exit 1
fi
```

Validate メソッドのコードリスト

このメソッドは、Confdir プロパティが示すディレクトリの存在を確認します。RGM がこのメソッドを呼び出すのは、クラスタ管理者がデータサービスを作成したときと、データサービスのプロパティを更新したときです。障害モニターがデータサービスを新しいノードにフェイルオーバーしたときは、Monitor_check メソッドは常にこのメソッドを呼び出します。

例 B-9 dns_validate メソッド

```
#!/bin/ksh
#
# HA-DNS の Validate メソッド
#
# このメソッドは、リソースの Confdir プロパティを妥当性検査する。
# Validate メソッドが呼び出されるのは、リソースが作成されたときと、リソース
# プロパティが更新されたときの 2 つである。リソースが作成されたとき、
# Validate メソッドは -c フラグで呼び出され、すべてのシステム定義プロ
# パティと拡張プロパティがコマンド行引数として渡される。リソースプロ
# パティが更新されたとき、Validate メソッドは -u フラグで呼び出され、
# 更新されるプロパティのプロパティ/値のペアだけがコマンド行引数とし
# て渡される。
#
# 例: リソースが作成されたとき、コマンド行引数は次のようになる。
#
# dns_validate -c -R <...> -G <...> -T <...>
-r <sysdef-prop=value>...
#     -x <extension-prop=value>.... -g <resourcegroup-prop=value>....
```

例 B-9 dns_validate メソッド (続き)

```
#
# 例: リソースプロパティが更新されたとき、コマンド行引数は次のようになる。
#
# dns_validate -u -R <...> -G <...> -T <...>
# -r <sys-prop_being_updated=value>
# または
# dns_validate -u -R <...> -G <...> -T <...>
# -x <extn-prop_being_updated=value>
#

#pragma ident    "@(#)dns_validate    1.1    00/05/24 SMI"

#####
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
    typeset opt

    while getopts `cur:x:g:R:T:G:` opt
    do
        case "$opt" in
            R)
                # DNS リソースの名前。
                RESOURCE_NAME=$OPTARG
                ;;
            G)
                # リソースが構成されているリソース
                # グループの名前。
                RESOURCEGROUP_NAME=$OPTARG
                ;;
            T)
                # リソースタイプの名前。
                RESOURCETYPE_NAME=$OPTARG
                ;;
            r)
                # メソッドはシステム定義プロパティにアクセスして
                # いない。したがって、このフラグは動作なし。
                ;;
            g)
                # メソッドはリソースグループプロパティにアクセスして
                # いない。したがって、このフラグは動作なし。
                ;;
            c)
                # Validate メソッドがリソースの作成中に呼び出されてい
                # ることを示す。したがって、このフラグは動作なし。
                ;;
            u)

```

例 B-9 dns_validate メソッド (続き)

```
# リソースがすでに存在しているときは、プロパティの更新
# を示す。Confdir プロパティを更新する場合、Confdir
# がコマンド行引数に現れるはずである。現れない場合、
# メソッドは scha_resource_get を使用して
# Confdir を探す必要がある。
UPDATE_PROPERTY=1
;;

x)
# 拡張プロパティのリスト。プロパティと値のペア。
# 区切り文字は「=」。
PROPERTY=`echo $OPTARG | awk -F= `{print $1}``
VAL=`echo $OPTARG | awk -F= `{print $2}``

# Confdir 拡張プロパティがコマンド行上に存在する場合、
# その値を記録する。
if [ $PROPERTY == "Confdir" ];
then
CONFDIR=$VAL
CONFDIR_FOUND=1
fi
;;

*)
logger -p ${SYSLOG_FACILITY}.err \
-t [${SYSLOG_TAG}] \
"ERROR: Option $OPTARG unknown"
exit 1
;;

esac

done
}

#####
# MAIN
#
#####

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# CONFDIR の値を NULL に設定する。この後、このメソッドは Confdir プロパ
# ティの値を、コマンド行から取得するか、scha_resource_get を使って
# 取得する。
CONFDIR=""
UPDATE_PROPERTY=0
CONFDIR_FOUND=0

# このメソッドに渡された引数を構文解析する。
parse_args "$@"
```

例 B-9 dns_validate メソッド (続き)

```
# プロパティの更新の結果として呼び出されている場合、Validate メソッドはコマンド行から Confdir 拡張プロパティの値を取得する。そうでない場合、scha_resource_get を使用して Confdir の値を取得する。
if ( (( $UPDATE_PROPERTY == 1 )) && (( CONFDIR_FOUND == 0 )) ); then
    config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME
\
    -G $RESOURCEGROUP_NAME Confdir`
    CONFDIR=`echo $config_info | awk `{print $2}``
fi

# Confdir プロパティが値を持っているかどうかを確認する。持っていない場合、状態 1 (失敗) で終了する。
if [[ -z $CONFDIR ]]; then
    logger -p ${SYSLOG_FACILITY}.err \
        "${ARGV0} Validate method for resource "$RESOURCE_NAME " failed"
    exit 1
fi

# 実際の Confdir プロパティ値の妥当性検査はここから始まる。

# $CONFDIR がアクセス可能であるかどうかを検査する。
if [ ! -d $CONFDIR ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}] \
        "${ARGV0} Directory $CONFDIR missing or not
mounted"
    exit 1
fi

# named.conf ファイルが Confdir ディレクトリ内に存在するかどうかを
# 検査する。
if [ ! -s $CONFDIR/named.conf ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}]
\
        "${ARGV0} File $CONFDIR/named.conf is missing
or empty"
    exit 1
fi

# Validate メソッドが成功したことを示すメッセージを記録する。
logger -p ${SYSLOG_FACILITY}.info -t [${SYSLOG_TAG}] \
    "${ARGV0} Validate method for resource "$RESOURCE_NAME
\
    " completed successfully"

exit 0
```

Update メソッドのコードリスト

リソースのプロパティが変更されたとき、RGM は Update メソッドを呼び出し、動作中のリソースにその旨を通知します。

例 B-10 dns_update メソッド

```
#!/bin/ksh
#
# HA-DNS の Update メソッド
#
# 実際のプロパティの更新は RGM が行う。更新の影響を受けるのは障害モ
# ニターだけである。したがって、このメソッドは障害モニターを再起動
# する必要がある。

#pragma ident  "@(#)dns_update  1.1  00/05/24  SMI"

#####
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
    typeset opt

    while getopts `R:G:T:` opt
    do
        case "$opt" in
            R)
                # DNS リソースの名前。
                RESOURCE_NAME=$OPTARG
                ;;
            G)
                # リソースが構成されているリソース
                # グループの名前。
                RESOURCEGROUP_NAME=$OPTARG
                ;;
            T)
                # リソースタイプの名前。
                RESOURCETYPE_NAME=$OPTARG
                ;;
            *)
                logger -p ${SYSLOG_FACILITY}.err \
                -t [${RESOURCETYPE_NAME},${RESOURCEGROUP_NAME},${RESOURCE_NAME}]
                "ERROR: Option $OPTARG unknown"
                exit 1
                ;;
        esac
    done
}
```

例 B-10 dns_update メソッド (続き)

```
}

#####
# MAIN
#
#####

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCE_NAME, $RESOURCEGROUP_NAME, $RESOURCE_NAME

# リソースの RT_BASEDIR プロパティを取得することによって、検証メソッド
# が存在する場所を見つける。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME
-G $RESOURCEGROUP_NAME`

# Update メソッドが呼び出されると、RGM は更新されるプロパティの値を
# 更新する。このメソッドは、障害モニター (検証メソッド) が動作し
# ているかどうかを検査し、動作している場合は強制終了し、再起動
# する必要がある。
if pmfadm -q $PMF_TAG.monitor; then

    # すでに動作している障害モニターを強制終了する。
    pmfadm -s $PMF_TAG.monitor TERM
    if [ $? -ne 0 ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}
\
        "${ARGV0} Could not stop the monitor"
        exit 1
    else
        # DNS の停止に成功。メッセージを記録する。
        logger -p ${SYSLOG_FACILITY}.info -t [${SYSLOG_TAG}
\
        "Monitor for HA-DNS successfully stopped"
    fi

# モニターを再起動する。
pmfadm -c $PMF_TAG.monitor -n -1 -t -1 $RT_BASEDIR/dns_probe \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME -T $RESOURCE_NAME
if [ $? -ne 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG} \
    "${ARGV0} Could not restart monitor for HA-DNS "
```

例 B-10 dns_update メソッド (続き)

```
        exit 1
    else
        logger -p ${SYSLOG_FACILITY}.info -t [${SYSLOG_TAG}]\
            "Monitor for HA-DNS successfully restarted"
    fi
fi
exit 0
```

付録 C

サンプル DSDL リソースタイプのコード例

この付録では、SUNW.xfnts リソースタイプの各メソッドの完全なコード例を示します。また、コールバックメソッドが呼び出すサブルーチンのコードを含む、xfntc.c のコード例を示します。この付録に含まれるコードリストは、次のとおりです。

- 247 ページの「xfnts.c」
- 259 ページの「xfnts_monitor_check メソッド」
- 260 ページの「xfnts_monitor_start メソッド」
- 261 ページの「xfnts_monitor_stop メソッド」
- 262 ページの「xfnts_probe メソッド」
- 265 ページの「xfnts_start メソッド」
- 267 ページの「xfnts_stop メソッドのコードリスト」
- 268 ページの「xfnts_update メソッドのコードリスト」
- 269 ページの「xfnts_validate メソッドのコードリスト」

xfnts.c

このファイルは、SUNW.xfnts メソッドが呼び出すサブルーチンを実装します。

例 C-1 xfnts.c

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts.c - HA-XFS 用の一般的なユーティリティ
 *
 * このユーティリティは、データサービスと障害モニターの
 * 妥当性検査、起動、および停止を実行するメソッドを持つ。
 * また、データサービスの状態を検証するメソッドも持つ。
 * 検証機能は、成功または失敗だけを戻す。xfnts_probe.c ファイル
 * 内のメソッドで戻された値に基づいて、アクションが行われる。
```

例 C-1 xfnts.c (続き)

```
*
*/

#pragma ident "@(#)xfnts.c 1.47 01/01/18 SMI"

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <scha.h>
#include <rgm/libdsdev.h>
#include <errno.h>
#include "xfnts.h"

/*
 * HA-XFS データサービスが完全に起動して動作するまでの
 * 初期タイムアウト。サービスを検証する前に、start_timeout
 * 時間の 3% (SVC_WAIT_PCT) だけ待つ。
 */
#define SVC_WAIT_PCT 3

/*
 * probe_timeout の 95% の時間でポートと接続する必要がある。
 * また、svc_probe 関数は、残りの時間を使用して、ポートとの接続を切断する。
 */
#define SVC_CONNECT_TIMEOUT_PCT 95

/*
 * SVC_WAIT_TIME は、svc_wait() で起動している間だけ使用される。
 * svc_wait() では、サービスが起動していることを確認してから戻る必要がある。
 * そのため、svc_probe() を呼び出し、サービスを監視する必要がある。
 * SVC_WAIT_TIME はこのような検証を繰り返す時間間隔である。
 */
#define SVC_WAIT_TIME 5

/*
 * この値は、probe_timeout に時間が残っていない場合に、
 * 切断タイムアウトとして使用される。
 */
#define SVC_DISCONNECT_TIMEOUT_SECONDS 2

/*
 * svc_validate():
 *
 * リソース構成に対して、HA-XFS 固有の妥当性検査を行う。

```

例 C-1 xfnts.c (続き)

```
*
* svc_validate は、次の妥当性を検査する。
* 1. Confdir_list 拡張プロパティ
* 2. fontserver.cfg ファイル
* 3. xfs バイナリ
* 4. port_list プロパティ
* 5. ネットワークリソース
* 6. その他の拡張プロパティ
*
* 上記の妥当性検査のいずれかが失敗した場合、0 以上の値を戻す。
* それ以外の場合、成功を示す 0 を戻す。
*/

int
svc_validate(scds_handle_t scds_handle)
{
    char    xfnts_conf[SCDS_ARRAY_SIZE];
    scha_str_array_t *confdirs;
    scds_net_resource_list_t *snrlp;
    int rc;
    struct stat statbuf;
    scds_port_list_t *portlist;
    scha_err_t err;

    /*
     * XFS データサービス用の構成ディレクトリを confdir_list
     * 拡張プロパティから取得する。
     */
    confdirs = scds_get_ext_confdir_list(scds_handle);

    /* confdir_list 拡張プロパティが存在しない場合、エラーを戻す。
     */
    if (confdirs == NULL || confdirs->array_cnt != 1) {
        scds_syslog(LOG_ERR,
            "Property Confdir_list is not set properly.");
        return (1); /* 妥当性検査は失敗。*/
    }

    /*
     * 構成ファイルへのパスを confdir_list 拡張プロパティから構築する。
     * HA-XFS が持つ構成は 1 つだけなので、confdir_list
     * プロパティの最初のエンタリを使用する必要がある。
     */
    (void) sprintf(xfnts_conf, "%s/fontserver.cfg",
        confdirs->str_array[0]);

    /*
     * HA-XFS の構成ファイルが適切な場所に存在することを確認する。
     * HA-XFS 構成ファイルにアクセスして、アクセス権が
     * 適切に設定されていることを確認する。
     */
    if (stat(xfnts_conf, &statbuf) != 0) {
        /*
```

例 C-1 xfnts.c (続き)

```
    * errno.h プロトタイプには void 引数がないので、
    * lint エラーが抑制される。
    */
    scds_syslog(LOG_ERR,
        "Failed to access file <%s> : <%s>",
        xfnts_conf, strerror(errno)); /*lint !e746 */
    return (1);
}

/*
 * XFS バイナリが存在し、アクセス権が正しいことを確認する。
 * XFS バイナリは、広域ファイルシステムではなく、
 * ローカルファイルシステム上にあるものと想定する。
 */
if (stat("/usr/openwin/bin/xfns", &statbuf)
!= 0) {
    scds_syslog(LOG_ERR,
        "Cannot access XFS binary : <%s> ",
        strerror(errno));
    return (1);
}

/* HA-XFS はポートを 1 つだけ持つ。*/
err = scds_get_port_list(scds_handle, &portlist);
if (err != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Could not access property Port_list: %s.",
        scds_error_string(err));
    return (1); /* 妥当性検査は失敗。*/
}

#ifdef TEST
    if (portlist->num_ports != 1) {
        scds_syslog(LOG_ERR,
            "Property Port_list must have only one value.");
        scds_free_port_list(portlist);
        return (1); /* 妥当性検査は失敗。*/
    }
#endif

/*
 * 当該リソースが利用できるネットワークアドレスリソースを
 * 取得しようとして失敗した場合、エラーを戻す。
 */
if ((err = scds_get_rs_hostnames(scds_handle, &snrlp))
!= SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group: %s.",
        scds_error_string(err));
    return (1); /* 妥当性検査は失敗。*/
}

/* ネットワークアドレスリソースが存在しない場合、エラーを戻す。*/
```

例 C-1 xfnts.c (続き)

```
if (snrlp == NULL || snrlp->num_netresources == 0) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    rc = 1;
    goto finished;
}

/* 他の重要な拡張プロパティが設定されていることを確認する。*/
if (scds_get_ext_monitor_retry_count(scds_handle) <= 0)
{
    scds_syslog(LOG_ERR,
        "Property Monitor_retry_count is not set.");
    rc = 1; /* 妥当性検査は失敗。*/
    goto finished;
}
if (scds_get_ext_monitor_retry_interval(scds_handle) <=
0) {
    scds_syslog(LOG_ERR,
        "Property Monitor_retry_interval is not set.");
    rc = 1; /* 妥当性検査は失敗。*/
    goto finished;
}

/* すべての妥当性検査は成功した。*/
scds_syslog(LOG_INFO, "Successful validation.");
rc = 0;

finished:
    scds_free_net_list(snrlp);
    scds_free_port_list(portlist);

    return (rc); /* 妥当性検査の結果を戻す。*/
}

/*
 * svc_start():
 *
 * X フォントサーバーを起動する。
 * 成功したときは 0 を戻し、失敗したときは 0 以上の値を戻す。
 *
 * XFS サービスを起動するには、
 * /usr/openwin/bin/xfns -config <fontserver.cfg file> -port <port
 * to listen> コマンドを実行する。
 * XFS は PMF の制御下で起動する。XFS は単一インスタンスのサービスとして
 * 起動する。
 * データサービス用のPMF タグの形式は、<resourcegroupname, resourcename,
 * instance_number.svc>。
 * XFS の場合、インスタンスは 1 つだけなので、
 * タグ内の instance_number は 0 である。
 */

int
```

例 C-1 xfnts.c (続き)

```
svc_start(scds_handle_t scds_handle)
{
    char    xfnts_conf[SCDS_ARRAY_SIZE];
    char    cmd[SCDS_ARRAY_SIZE];
    scha_str_array_t *confdirs;
    scds_port_list_t *portlist;
    scha_err_t err;

    /* 構成ディレクトリを confdir_list プロパティから取得する。*/
    confdirs = scds_get_ext_confdir_list(scds_handle);

    (void) sprintf(xfnts_conf, "%s/fontserver.cfg",
confdirs->str_array[0]);

    /* XFS が使用するポートを Port_list プロパティから取得する。*/
    err = scds_get_port_list(scds_handle, &portlist);
    if (err != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR,
            "Could not access property Port_list.");
        return (1);
    }

    /*
    * HA-XFS を起動するためのコマンドを構築する。
    * 注: XFS を停止している間、XFS デーモンは
    * 次のメッセージを出力する。
    * 「/usr/openwin/bin/xfns notice: terminating」
    * このデーモンメッセージを抑制するには、
    * 出力を /dev/null にリダイレクトする。
    */
    (void) sprintf(cmd,
        "/usr/openwin/bin/xfns -config %s -port %d 2>/dev/null",
        xfnts_conf, portlist->ports[0].port);

    /*
    * HA-XFS を PMF の制御下で起動する。HA-XFS は単一インスタンスの
    * サービスとして起動される。scds_pmf_start 関数の最後の引数は、
    * 監視する子プロセスのレベルを示す。このパラメータが
    * -1 である場合、すべての子プロセスを親プロセスと同様に監視することを示す。
    */
    scds_syslog(LOG_INFO, "Issuing a start request.");
    err = scds_pmf_start(scds_handle, SCDS_PMF_TYPE_SVC,
        SCDS_PMF_SINGLE_INSTANCE, cmd, -1);

    if (err == SCHA_ERR_NOERR) {
        scds_syslog(LOG_INFO,
            "Start command completed successfully.");
    } else {
        scds_syslog(LOG_ERR,
            "Failed to start HA-XFS ");
    }

    scds_free_port_list(portlist);
}
```

例 C-1 xfnts.c (続き)

```
    return (err); /* 成功または失敗の状態を戻す。*/
}

/*
 * svc_stop():
 *
 * XFS サーバーを停止する。
 * 成功したときは 0 を戻し、失敗したときは 0 以上の値を戻す。
 *
 * svc_stop は、scds_pmf_stop ツールキット関数を呼び出すことによって
 * サーバーを停止する。
 */
int
svc_stop(scds_handle_t scds_handle)
{
    scha_err_t    err;

    /*
     * 停止メソッドが成功できるタイムアウト値を Stop_Timeout
     * (システム定義) プロパティに設定する。
     */
    scds_syslog(LOG_ERR, "Issuing a stop request.");
    err = scds_pmf_stop(scds_handle,
        SCDS_PMF_TYPE_SVC, SCDS_PMF_SINGLE_INSTANCE, SIGTERM,
        scds_get_rs_stop_timeout(scds_handle));

    if (err != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR,
            "Failed to stop HA-XFS.");
        return (1);
    }

    scds_syslog(LOG_INFO,
        "Successfully stopped HA-XFS.");
    return (SCHA_ERR_NOERR); /* 停止は成功。*/
}

/*
 * svc_wait():
 *
 * データサービスが完全に起動するまで待ち、動作状態を確認する。
 */
int
svc_wait(scds_handle_t scds_handle)
{
    int rc, svc_start_timeout, probe_timeout;
    scds_netaddr_list_t    *netaddr;

    /* 検証に使用するネットワークリソースを取得する。*/
    if (scds_get_netaddr_list(scds_handle, &netaddr)) {
        scds_syslog(LOG_ERR,
```

例 C-1 xfnts.c (続き)

```
        "No network address resources found in resource group.");
    return (1);
}

/* ネットワークリソースが存在しない場合、エラーを戻す。*/
if (netaddr == NULL || netaddr->num_netaddrs == 0) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    return (1);
}

/*
 * 起動メソッドのタイムアウト、検証を行うポート番号、
 * および検証用のタイムアウト値を取得する。
 */
svc_start_timeout = scds_get_rs_start_timeout(scds_handle);
probe_timeout = scds_get_ext_probe_timeout(scds_handle);

/*
 * データサービスを実際に検証する前に、start_timeout の
 * SVC_WAIT_PCT (%) だけ休止状態になる。これによって、データサービスが
 * 完全に起動し、検証に応答できるようになる。
 * 注: SVC_WAIT_PCT の値はデータサービス
 * ごとに異なる可能性がある。
 * sleep() ではなく scds_svc_wait() を
 * 呼び出すと、サービスが繰り返して失敗する場合には中断して、
 * すぐに戻ることができる。
 */
if (scds_svc_wait(scds_handle, (svc_start_timeout * SVC_WAIT_PCT)/100)
    != SCHA_ERR_NOERR) {

    scds_syslog(LOG_ERR, "Service failed to start.");
    return (1);
}

do {
    /*
     * ネットワークリソースの IP アドレスとポート名で、
     * データサービスを検証する。
     */
    rc = svc_probe(scds_handle,
        netaddr->netaddrs[0].hostname,
        netaddr->netaddrs[0].port_proto.port, probe_timeout);
    if (rc == SCHA_ERR_NOERR) {
        /* 成功。リソースを解放して戻る。*/
        scds_free_netaddr_list(netaddr);
        return (0);
    }
}

/*
 * データサービスはまだ起動しようとする。しばらくの
 * 間休止状態になり、もう一度検証を行う。sleep() ではなく
 * scds_svc_wait() を呼び出すと、サービスが
```

例 C-1 xfnts.c (続き)

```
    * 繰り返し失敗する場合には中断して、すぐに戻ることができる。
    */
    if (scds_svc_wait(scds_handle, SVC_WAIT_TIME)
        != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR, "Service failed to start.");
        return (1);
    }

    /* RGM がタイムアウトするのを待って、プログラムを終了する。*/
} while (1);

}

/*
 * この関数は、HA-XFS リソース用の障害モニターを起動する。
 * そのためには、検証機能を PMF の制御下で起動する。PMF タグの形式は
 * <RG-name,RS-name,instance_number.mon>。
 * PMF の再起動オプションを使用するが、
 * 無限に再起動しない。代わりに、interval/retry_time を RTR ファイルから取得する。
 */

int
mon_start(scds_handle_t scds_handle)
{
    scha_err_t    err;

    scds_syslog_debug(DBG_LEVEL_HIGH,
        "Calling MONITOR_START method for resource <%s>.",
        scds_get_resource_name(scds_handle));

    /*
     * xfnts_probe 検証機能は、他の RT 用のコールバックメソッドが
     * インストールされているのと同じサブディレクトリにあるものと想定する。
     * scds_pmf_start の最後のパラメータは、
     * 監視する子プロセスのレベルを示す。検証機能は PMF の制御下で開始されるので
     * 検証プロセスだけを監視すればよい。
     * したがって、この値は 0 である。
     */
    err = scds_pmf_start(scds_handle, SCDS_PMF_TYPE_MON,
        SCDS_PMF_SINGLE_INSTANCE, "xfnts_probe",
0);

    if (err != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR,
            "Failed to start fault monitor.");
        return (1);
    }

    scds_syslog(LOG_INFO,
        "Started the fault monitor.");

    return (SCHA_ERR_NOERR); /* モニターの起動に成功。*/
}

```

例 C-1 xfnts.c (続き)

```
/*
 * この関数は、HA-XFS リソース用の障害モニターを停止する。
 * これは PMF 経由で行われる。障害モニター用の PMF タグの形式は、
 * <RG-name_RS-name,instance_number.mon>。
 */

int
mon_stop(scds_handle_t scds_handle)
{
    scha_err_t    err;

    scds_syslog_debug(DBG_LEVEL_HIGH,
        "Calling scds_pmf_stop method");

    err = scds_pmf_stop(scds_handle, SCDS_PMF_TYPE_MON,
        SCDS_PMF_SINGLE_INSTANCE, SIGKILL,
        scds_get_rs_monitor_stop_timeout(scds_handle));

    if (err != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR,
            "Failed to stop fault monitor.");
        return (1);
    }

    scds_syslog(LOG_INFO,
        "Stopped the fault monitor.");

    return (SCHA_ERR_NOERR); /* モニターの停止は成功。 */
}

/*
 * svc_probe(): データサービスに固有な検証を行う。
 * 0 (成功) から100 (致命的な障害) の範囲の整数値を返す。
 *
 * 検証機能は、リソースの Port_list 拡張プロパティで指定されたポート上で、
 * XFS サーバーとの単純ソケット接続を行い、データサービスを ping する。
 * ポートとの接続に失敗した場合、100 の値を戻して、
 * 致命的な障害であることを示す。ポートとの接続は成功したが
 * 切断が失敗した場合、50 の値を戻して、部分的な障害であることを示す。
 */
int
svc_probe(scds_handle_t scds_handle, char *hostname, int port, int
timeout)
{
    int rc;
    hrttime_t  t1, t2;
    int sock;
    char testcmd[2048];
    int time_used, time_remaining;

```

例 C-1 xfnts.c (続き)

```
time_t      connect_timeout;

/*
 * データサービスを検証するには、
 * port_list プロパティで指定されている、XFS データサービスを
 * 提供するホスト上のポートとソケット接続を行う。
 * 指定されたポート上で通信するように構成された
 * tXFS サービスが接続に応答した場合、検証が成功したと判断する。
 * probe_timeout プロパティに設定された時間だけ待っても
 * 応答がない場合、検証に失敗したと判断する。
 */

/*
 * SVC_CONNECT_TIMEOUT_PCT をタイムアウトの
 * 百分率として使用し、ポートと接続する。
 */
connect_timeout = (SVC_CONNECT_TIMEOUT_PCT * timeout)/100;
t1 = (hrtime_t)(gethrtime()/1E9);

/*
 * 検証機能は、指定されたホスト名とポートを使用して接続を行う。
 * 実際には、接続は probe_timeout の 95% の時間でタイムアウトする。
 */
rc = scds_fm_tcp_connect(scds_handle, &sock, hostname, port,
                        connect_timeout);
if (rc) {
    scds_syslog(LOG_ERR,
                "Failed to connect to port <%d> of resource <%s>.",
                port, scds_get_resource_name(scds_handle));
    /* 致命的な障害。*/
    return (SCDS_PROBE_COMPLETE_FAILURE);
}

t2 = (hrtime_t)(gethrtime()/1E9);

/*
 * 接続にかかる実際の時間を計算する。この値は、
 * 接続に割り当てられた時間を示す connect_timeout 以下である
 * 必要がある。接続に割り当てられた時間をすべて使い切った場合、
 * probe_timeout に残った値が当該関数に渡され、
 * 切断タイムアウトとして使用される。そうでなければ、
 * 接続呼び出しで残った時間は切断タイムアウトに追加される。
 */

time_used = (int)(t2 - t1);

/*
 * 残った時間 (タイムアウトから接続にかかった時間を引いた値) を切断に
 * 使用する。
 */
```

例 C-1 xfnts.c (続き)

```
time_remaining = timeout - (int)time_used;

/*
 * すべての時間を使い切った場合、ハードコーディングされた小さな
 * タイムアウトを使用して、切断しようとする。
 * これによって、fd リークを防ぐ。
 */
if (time_remaining <= 0) {
    scds_syslog_debug(DBG_LEVEL_LOW,
        "svc_probe used entire timeout of "
        "%d seconds during connect operation and exceeded the "
        "timeout by %d seconds. Attempting disconnect with timeout"
        " %d ",
        connect_timeout,
        abs(time_used),
        SVC_DISCONNECT_TIMEOUT_SECONDS);

    time_remaining = SVC_DISCONNECT_TIMEOUT_SECONDS;
}

/*
 * 切断に失敗した場合、部分的な障害を戻す。
 * 理由: 接続呼び出しは成功した。これは、アプリケーションが
 * 動作していることを意味する。切断が失敗した原因は、
 * 後者の場合、アプリケーションが停止したとは宣言しない。
 * つまり、致命的な障害を戻さない。その代わりに、
 * 部分的な障害であると宣言する。この状態が続く場合、
 * 切断呼び出しは再び失敗し、アプリケーションは再起動される。
 */
rc = scds_fm_tcp_disconnect(scds_handle, sock, time_remaining);
if (rc != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Failed to disconnect to port %d of resource %s.",
        port, scds_get_resource_name(scds_handle));
    /* 部分的な障害。*/
    return (SCDS_PROBE_COMPLETE_FAILURE/2);
}

t2 = (hrtime_t)(gethrtime()/1E9);
time_used = (int)(t2 - t1);
time_remaining = timeout - time_used;

/*
 * 時間が残っていない場合、fsinfo による完全なテストを実行せず、
 * SCDS_PROBE_COMPLETE_FAILURE/2 を戻す。これによって、
 * このタイムアウトが続く場合、
 * サーバーは再起動される。
 */
if (time_remaining <= 0) {
    scds_syslog(LOG_ERR, "Probe timed out.");
    return (SCDS_PROBE_COMPLETE_FAILURE/2);
}
```

例 C-1 xfnts.c (続き)

```
/*
 * ポートへの接続および接続の切断に成功。
 * fsinfo コマンドを実行してサーバーの状態を完全に検査する。
 * stdout をリダイレクトする。そうしないと fsinfo の実行結果は
 * コンソールに出力される。
 */
(void) sprintf(testcmd,
    "/usr/openwin/bin/fsinfo -server %s:%d> /dev/null",
    hostname, port);
scds_syslog_debug(DBG_LEVEL_HIGH,
    "Checking the server status with %s.", testcmd);
if (scds_timerun(scds_handle, testcmd, time_remaining,
    SIGKILL, &rc) != SCHA_ERR_NOERR || rc != 0) {

    scds_syslog(LOG_ERR,
        "Failed to check server status with command <%s>",
        testcmd);
    return (SCDS_PROBE_COMPLETE_FAILURE/2);
}
return (0);
}
```

xfnts_monitor_check メソッド

このメソッドは、基本的なリソースタイプ構成が有効であることを確認します。

例 C-2 xfnts_monitor_check.c

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_check.c - HA-XFS のモニター検査メソッド
 */

#pragma ident "@(#)xfnts_monitor_check.c 1.11 01/01/18
SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * サービスに対して簡単な妥当性検査を行う。
 */

int
main(int argc, char *argv[])
```

例 C-2 xfnts_monitor_check.c (続き)

```
{
    scds_handle_t    scds_handle;
    int    rc;

    /* RGM から渡された引数を処理して、syslog を初期化する。*/
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }

    rc = svc_validate(scds_handle);
    scds_syslog_debug(DBG_LEVEL_HIGH,
        "monitor_check method "
        "was called and returned <%d>.", rc);

    /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
    scds_close(&scds_handle);

    /* モニター検査の一環として実行した検証メソッドの結果を戻す。*/
    return (rc);
}
```

xfnts_monitor_start メソッド

このメソッドは、xfnts_probe メソッドを起動します。

例 C-3 xfnts_monitor_start.c

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_start.c - HA-XFS のモニター起動メソッド
 */

#pragma ident "@(#)xfnts_monitor_start.c 1.10 01/01/18
SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * このメソッドは、HA-XFS リソース用の障害モニターを起動する。
 * そのためには、検証機能を PMF の制御下で起動する。PMF タグの形式は
 * <RG-name, RS-name.mon> である。PMF の再起動オプションを
 * 使用するが、無限に再起動しない。その代わりに、
```

例 C-3 xfnts_monitor_start.c (続き)

```
* interval/retry_time を RTR ファイルから取得する。
*/

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int    rc;

    /* RGM から渡された引数を処理して、syslog を初期化する。*/
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }

    rc = mon_start(scds_handle);

    /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
    scds_close(&scds_handle);

    /* monitor_start メソッドの結果を戻す。*/
    return (rc);
}
```

xfnts_monitor_stop メソッド

このメソッドは、xfnts_probe メソッドを停止します。

例 C-4 xfnts_monitor_stop.c

```
/*
 * Copyright (c) 2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_stop.c -HA-XFS のモニター停止メソッド
 */

#pragma ident "@(#)xfnts_monitor_stop.c 1.9 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * このメソッドは、HA-XFS リソース用の障害モニターを停止する。
 * この処理は PMF 経由で行われる。障害モニター用の
 * PMF タグの形式は <RG-name_RS-name.mon> である。
 */
```

例 C-4 xfnts_monitor_stop.c (続き)

```
*/

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int             rc;

    /* RGM から渡された引数を処理して、syslog を初期化する。*/
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }
    rc = mon_stop(scds_handle);

    /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
    scds_close(&scds_handle);

    /* monitor_stop メソッドの結果を戻す。*/
    return (rc);
}
```

xfnts_probe メソッド

xfnts_probe メソッドは、アプリケーションの可用性を検査して、データサービスをフェイルオーバーするか、再起動するかを決定します。xfnts_probe メソッドは、xfnts_monitor_start コールバックメソッドによって起動され、xfnts_monitor_stop コールバックメソッドによって停止されます。

例 C-5 xfnts_probe.c+

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_probe.c - HA-XFS の検査
 */

#pragma ident "@(#)xfnts_probe.c 1.26 01/01/18 SMI"

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <signal.h>
```

例 C-5 xfnts_probe.c+ (続き)

```
#include <sys/time.h>
#include <sys/socket.h>
#include <strings.h>
#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * main():
 * sleep() を実行して、PMF アクションスクリプトが sleep() に割り込むのを
 * 待機する無限ループ。sleep() への割り込みが発生すると、HA-XFS 用の
 * 起動メソッドを呼び出して、再起動する。
 *
 */

int
main(int argc, char *argv[])
{
    int          timeout;
    int          port, ip, probe_result;
    scds_handle_t scds_handle;

    hrttime_t    ht1, ht2;
    unsigned long dt;

    scds_netaddr_list_t *netaddr;
    char         *hostname;

    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }

    /* 当該リソースに利用できる IP アドレスを取得する。*/
    if (scds_get_netaddr_list(scds_handle, &netaddr)) {
        scds_syslog(LOG_ERR,
            "No network address resource in resource group.");
        scds_close(&scds_handle);
        return (1);
    }

    /* ネットワークリソースが存在しない場合、エラーを戻す。*/
    if (netaddr == NULL || netaddr->num_netaddrs == 0) {
        scds_syslog(LOG_ERR,
            "No network address resource in resource group.");
        return (1);
    }

    /*
     * x プロパティからタイムアウト値を設定する。つまり、

```

例 C-5 xfnts_probe.c+ (続き)

```
* 当該リソース用に構成されたすべてのネットワークリソース間で
* タイムアウト値を分割するのではなく、検証を行うたびに、
* 各ネットワークリソースに設定されているタイムアウト値を
* 取得することを意味する。
*/
timeout = scds_get_ext_probe_timeout(scds_handle);

for (;;) {

    /*
    * 連続する検証の間、Throrough_probe_interval
    * の期間、状態になる。
    */
    (void) scds_fm_sleep(scds_handle,
        scds_get_rs_thorough_probe_interval(scds_handle));

    /*
    * 使用するすべての IP アドレスを検証する。
    * 以下をループで検証する。
    * 1. 使用するすべてのネットワークリソース
    * 2. 指定されたりソースのすべての IP アドレス
    * 検証する IP アドレスごとに、障害履歴を計算する。
    */
    probe_result = 0;
    /*
    * すべてのリソースを繰り返し検証して、svc_probe() の
    * 呼び出しに使用する各 IP アドレスを取得する。
    */
    for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
        /*
        * 状態を監視するホスト名とポートを取得する。
        */
        hostname = netaddr->netaddrs[ip].hostname;
        port = netaddr->netaddrs[ip].port_proto.port;
        /*
        * HA-XFS がサポートするポートは 1 つだけなので、
        * ポート値はポートの配列の最初の
        * エントリから取得する。
        */
        ht1 = gethrtime(); /* 検証開始時間を取得する。*/
        scds_syslog(LOG_INFO, "Probing the service on "
            "port: %d.", port);

        probe_result =
            svc_probe(scds_handle, hostname, port, timeout);

        /*
        * サービス検証履歴を更新し、
        * 必要に応じて、アクションを実行する。
        * 検証終了時間を取得する。
        */
        ht2 = gethrtime();
```

例 C-5 xfnets_probe.c+ (続き)

```
/* ミリ秒に変換する。*/
dt = (ulong_t)(ht2 - ht1) / 1e6);

/*
 * 障害の履歴を計算し、
 * 必要に応じて、アクションを実行する。
 */
(void) scds_fm_action(scds_handle,
    probe_result, (long)dt);
} /* ネットワークリソースごと */
} /* 検証を永続的に繰り返す。*/
}
```

xfnets_start メソッド

データサービスリソースを含むリソースグループがオンラインになったとき、あるいは、リソースが有効になったとき、RGMはそのクラスタノード上で start メソッドを呼び出します。xfnets_start メソッドはそのノード上で xfs デーモンを起動します。

例 C-6 xfnets_start.c

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnets_svc_start.c - HA-XFS の起動メソッド
 */

#pragma ident "@(#)xfnets_svc_start.c 1.13 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnets.h"

/*
 * HA-XFS 用の起動メソッド。リソース設定に対していくつかの
 * サニティ検査を行った後、アクションスクリプトを使用して HA-XFS を
 * PMF の制御下で起動する。
 */

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int rc;

    /*
```

例 C-6 xfnets_start.c (続き)

```
    * RGM から渡された引数を処理して、syslog を初期化する。
    */

if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
    scds_syslog(LOG_ERR, "Failed to initialize the handle.");
    return (1);
}

/* 構成の妥当性を検査する。エラーがあれば戻る。*/
rc = svc_validate(scds_handle);
if (rc != 0) {
    scds_syslog(LOG_ERR,
        "Failed to validate configuration.");
    return (rc);
}

/* データサービスを起動する。失敗した場合、エラーで戻る。*/
rc = svc_start(scds_handle);
if (rc != 0) {
    goto finished;
}

/* サービスが完全に起動するまで待つ。*/
scds_syslog_debug(DBG_LEVEL_HIGH,
    "Calling svc_wait to verify that service has started.");

rc = svc_wait(scds_handle);

scds_syslog_debug(DBG_LEVEL_HIGH,
    "Returned from svc_wait");

if (rc == 0) {
    scds_syslog(LOG_INFO, "Successfully started the service.");
} else {
    scds_syslog(LOG_ERR, "Failed to start the service.");
}

finished:
/* 割り当てられた環境リソースを解放する。*/
scds_close(&scds_handle);

return (rc);
}
```

xfnts_stop メソッドのコードリスト

HA-XFS リソースを含むリソースグループがクラスタのノード上でオフラインになったとき、あるいは、リソースが無効になったとき、RGM はそのクラスタノード上で stop メソッドを呼び出します。xfnts_stop メソッドはそのノード上で xfs デモンを停止します。

例 C-7 xfnts_stop.c

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_svc_stop.c - HA-XFS の停止メソッド
 */

#pragma ident "@(#)xfnts_svc_stop.c 1.10 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * PMF を使用して HA-XFS プロセスを停止する。
 */

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int              rc;

    /* RGM から渡された引数を処理して、syslog を初期化する。*/
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }

    rc = svc_stop(scds_handle);

    /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
    scds_close(&scds_handle);

    /* svc_stop メソッドの結果を戻す。*/
    return (rc);
}
```

xfnts_update メソッドのコードリスト

リソースのプロパティが変更されたとき、RGM は Update メソッドを呼び出し、動作中のリソースにその旨を通知します。RGM は、管理アクションがリソースまたはそのリソースグループのプロパティの設定に成功したあとに、Update を呼び出します。

例 C-8 xfnts_update.c

```
#pragma ident "@(#)xfnts_update.c 1.10      01/01/18 SMI"

/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_update.c - HA-XFS の更新メソッド
 */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <rgm/libdsdev.h>

/*
 * リソースのプロパティが更新された可能性がある。
 * このような更新可能なプロパティはすべて障害モニターに関連するもので
 * あるため、障害モニターを再起動する必要がある。
 */

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    scha_err_t      result;

    /* RGM から渡された引数を処理して、syslog を初期化する。*/
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }

    /*
     * 障害モニターがすでに動作していることを検査し、
     * 動作している場合、障害モニターを停止および再起動する。
     * scds_pmf_restart_fm() への 2 番目のパラメータは、再起動する
     * 必要がある障害モニターのインスタンスを一意に識別する。
     */
}
```

例 C-8 xfnts_update.c (続き)

```
scds_syslog(LOG_INFO, "Restarting the fault monitor.");
result = scds_pmf_restart_fm(scds_handle, 0);
if (result != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Failed to restart fault monitor.");
    /* initialize が割り当てたすべてのメモリーを解放する。*/
    scds_close(&scds_handle);
    return (1);
}

scds_syslog(LOG_INFO,
    "Completed successfully.");

/* initialize が割り当てたすべてのメモリーを解放する。*/
scds_close(&scds_handle);

return (0);
}
```

xfnts_validate メソッドのコードリスト

xfnts_validate メソッドは、Confdir_list プロパティが示すディレクトリの存在を確認します。RGM がこのメソッドを呼び出すのは、クラスタ管理者がデータサービスを作成したときと、データサービスのプロパティを更新したときです。障害モニターがデータサービスを新しいノードにフェイルオーバーしたときは、Monitor_check メソッドは常にこのメソッドを呼び出します。

例 C-9 xfnts_validate.c

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_validate.c - HA-XFS の検証メソッド
 */

#pragma ident "@(#)xfnts_validate.c 1.9 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * プロパティが正しく設定されていることを確認する。
 */
```

例 C-9 xfnts_validate.c (続き)

```
int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int    rc;

    /* RGM から渡された引数进行处理して、syslog を初期化する。*/
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }
    rc = svc_validate(scds_handle);

    /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
    scds_close(&scds_handle);

    /* 検証メソッドの結果を戻す。*/
    return (rc);
}
```

RGM の有効な名前と値

この付録では、RGM の名前と値に対する有効な文字の要件を示します。

RGM の有効な名前

RGM の名前は次の 5 つのカテゴリに分類されます。

- リソースグループ名
- リソースタイプ名
- リソース名
- プロパティ名
- 列挙型リテラル名

リソースタイプ名を除いて、すべての名前は次の規則に従う必要があります。

- ASCII であること。
- 英字で始まること。
- 名前に使用できる文字は、英字の大文字と小文字、数字、ハイフン (-)、下線 (_)。
- 255 文字を超えないこと。

リソースタイプ名は、簡単な名前 (RTR ファイルの `Resource_type` プロパティで指定) または完全な名前 (RTR ファイルの `Vendor_id` と `Resource_type` で指定) のどちらでもかまいません。これら両方のプロパティを指定するとき、RGM は、`Vendor_id` と `Resource_type` の間にピリオドを挿入して、完全な名前を作成します。たとえば、`Vendor_id=SUNW` で、`Resource_type=sample` の場合、完全な名前は `SUNW.sample` です。これは、RGM の名前において、ピリオドが有効な文字である場合だけです。

RGM の値

RGM の値は、プロパティ値と記述値という2つのカテゴリに分類されます。両方とも、次のような同じ規則が適用されます。

- 値は ASCII であること。
- 値の最大長は 4M - 1 バイト (つまり、4,194,303 バイト) であること。
- NULL、改行文字、コンマ、セミコロンは、値に使用できない。

非クラスタ対応のアプリケーションの要件

通常、非クラスタ対応のアプリケーションの高可用性 (HA) を実現するには、特定の要件を満たす必要があります。そのための要件の一覧が、27 ページの「アプリケーションの適合性の分析」に示されています。この付録では、それらの要件のうち、特定のものについて詳細に説明します。

アプリケーションの高可用性を実現するには、そのリソースをリソースグループで構成します。アプリケーションのデータは、高可用性の広域ファイルシステムに格納されます。したがって、1つのサーバーが異常終了しても、正常に動作しているサーバーがデータにアクセスできます。『Sun Cluster 3.1 の概念』のクラスタファイルシステムに関する情報も参照してください。

ネットワーク上のクライアントがネットワークにアクセスする場合、論理ネットワーク IP アドレスは、データサービスリソースと同じリソースグループにある論理ホスト名リソースで構成されます。データサービスリソースとネットワークアドレスリソースは共にフェイルオーバーします。この場合、データサービスのネットワーククライアントは新しいホスト上のデータサービスリソースにアクセスします。

多重ホストデータ

高可用性の広域ファイルシステムのディスクセットは多重ホスト化されているため、ある物理ホストがクラッシュしても、正常に動作している物理ホストの1つがディスクにアクセスできます。アプリケーションの高可用性を実現するには、そのデータが高可用性であること、つまり、そのデータが広域 HA ファイルシステムに格納されていることが必要です。

広域ファイルシステムは、独立したものであるように作成されたディスクグループにマウントされます。ユーザーは、あるディスクグループをマウントされた広域ファイルシステムとして使用し、別のディスクグループをデータサービス (HA Oracle など) で使用する raw デバイスとして使用することもできます。

アプリケーションは、データファイルの位置を示すコマンド行スイッチまたは構成ファイルを持っていることがあります。アプリケーションが固定されたパス名を使用する場合は、アプリケーションのコードを変更せずに、このパス名を広域ファイルシステム内のファイルの位置を指すシンボリックリンクに変更できます。シンボリックリンクの詳しい使用方法については、274 ページの「多重ホストデータを配置するためのシンボリックリンクの使用」を参照してください。

最悪の場合は、実際のデータの位置を示すような何らかの機構を使用するように、アプリケーションのソースコードを変更する必要があります。この作業は、コマンド行スイッチを追加することにより行うことができます。

Sun Cluster は、ボリューム管理ソフトウェアに構成されている UNIX UFS ファイルシステムと HA の raw デバイスの使用をサポートします。インストールおよび構成するとき、システム管理者はどのディスクリソースを UFS ファイルシステムまたは raw デバイス用に使用するかを指定する必要があります。通常、raw デバイスを使用するのは、データベースサーバーとマルチメディアサーバーだけです。

多重ホストデータを配置するためのシンボリックリンクの使用

アプリケーションの中には、そのデータファイルへのパス名が固定されており、しかも、固定されたパス名を変更する機構がないものがあります。このような場合に、シンボリックリンクを使用すればアプリケーションのコードを変更せずに、済ませられる場合もあります。

たとえば、アプリケーションがそのデータファイルに固定されたパス名 `/etc/mydatafile` を指定すると仮定します。このパスは、論理ホストのファイルシステムの 1 つにあるファイルを示す値を持つシンボリックリンクに変更できます。たとえば、`/global/phys-schost-2/mydatafile` へのシンボリックリンクに変更できます。

ただし、データファイルの名前を内容とともに変更するアプリケーション (または、その管理手順) の場合には、シンボリックリンクをこのように使用すると問題が生じる可能性があります。たとえば、アプリケーションが更新を実行するとき、まず、新しい一時ファイル `/etc/mydatafile.new` を作成すると仮定します。次に、このデータベースは `rename(2)` システムコール (または `mv(1)` プログラム) を使用し、この一時ファイルの名前を実際のファイルの名前に変更します。一時ファイルを作成し、その名前を実際のファイルの名前に変更することにより、データサービスは、そのデータファイルの内容が常に適切であるようにします。

この時、`rename(2)` の操作はシンボリックリンクを破壊します。このため、`/etc/mydatafile` という名前は通常ファイルとなり、クラスタの広域ファイルシステムの中ではなく、`/etc` ディレクトリと同じファイルシステムの中に存在することになります。`/etc` ファイルシステムは各ホスト専用であるため、テイクオーバーまたはスイッチオーバー後はデータが利用できなくなります。

このような状況の根本的な問題は、既存のアプリケーションがシンボリックリンクに気付かない、つまり、シンボリックリンクを考慮するように作成されていないことにあります。シンボリックリンクを使用し、データアクセスを論理ホストのファイルシステムにリダイレクトするには、アプリケーション実装がシンボリックリンクを消去しないように動作する必要があります。したがって、シンボリックリンクは、論理ホストのファイルシステムへのデータの配置に関する問題をすべて解決できるわけではありません。

ホスト名

データサービス開発者は、データサービスが動作しているサーバーのホスト名を、データサービスが知る必要があるかどうかを判断する必要があります。知る必要があると判断した場合は、物理ホストではなく、論理ホストのホスト名 (つまり、アプリケーションリソースと同じリソースグループ内にある論理ホスト名リソース内に構成されているホスト名) を使用するようにデータサービスを変更する必要があります。

データサービスのクライアントサーバープロトコルでは、サーバーが自分のホスト名をクライアントへのメッセージの一部としてクライアントに戻すことがあります。このようなプロトコルでは、クライアントは戻されたホスト名をサーバーに接続するときのホスト名として使用できます。戻されたホスト名をテイクオーバーやスイッチオーバーが発生した後も使用できるようにするには、物理ホストではなく、リソースグループの論理ホスト名を使用する必要があります。物理ホスト名を使用している場合は、論理ホスト名をクライアントに戻すようにデータサービスのコードを変更する必要があります。

多重ホームホスト

多重ホームホストとは、複数のパブリックネットワーク上にあるホストのことです。このようなホストは複数 (つまり、ネットワークごとに1つ) のホスト名/IP アドレスのペアを持ちます。Sun Cluster は、1つのホストが複数のネットワーク上に存在できるように設計されています。1つのホストが単一のネットワーク上に存在することも可能ですが、このような場合は「多重ホームホスト」とは呼びません。物理ホスト名が複数のホスト名/IP アドレスのペアを持つように、各リソースグループも複数 (つまり、パブリックネットワークごとに1つ) のホスト名/IP アドレスのペアを持ちます。Sun Cluster がリソースグループをある物理ホストから別の物理ホストに移動するとき、そのリソースグループに対するホスト名/IP アドレスのペアもすべて移動します。

リソースグループに対するホスト名/IP アドレスのペアは、リソースグループに含まれる論理ホスト名リソースとして構成されます。このようなネットワークアドレスリソースは、システム管理者がリソースグループを作成および構成するときに指定します。Sun Cluster データサービス API は、このようなホスト名/IP アドレスのペアを照会する機能を持っています。

Solaris 環境用に書かれているほとんどの市販のデータサービスデーモンは、多重ホームホストを適切に処理できます。ネットワーク通信を行うとき、多くのデータサービスは Solaris のワイルドカードアドレス INADDR_ANY にバインドします。すると、INADDR_ANY は、すべてのネットワークインタフェースのすべての IP アドレスを自動的に処理します。INADDR_ANY は、現在マシンに構成されているすべての IP アドレスに効率的にバインドします。一般的に、INADDR_ANY を使用するデータサービスデーモンは、変更しなくても、Sun Cluster 論理ネットワークアドレスを処理できます。

INADDR_ANY へのバインドと特定の IP アドレスへのバインド

Sun Cluster の論理ネットワークアドレスの概念では、多重ホーム化されていない環境でも、マシンは複数の IP アドレスを持つことができます。つまり、独自の物理ホストの IP アドレスを 1 つだけ持ち、さらに、現在マスターしているネットワークアドレス (論理ホスト名) リソースごとに 1 つの IP アドレスを持ちます。ネットワークアドレスリソースのマスターになると、マシンは動的に追加の IP アドレスを獲得します。ネットワークアドレスリソースのマスターを終了するとき、マシンは動的に IP アドレスを放棄します。

データサービスの中には、INADDR_ANY にバインドしていると、Sun Cluster 環境で適切に動作しないもあります。このようなデータサービスは、リソースグループのマスターになると、またマスターをやめるときに、バインドしている IP アドレスのセットを動的に変更する必要があります。このようなデータサービスが再バインドする方法の 1 つが、起動メソッドと停止メソッドを使用し、データサービスのデーモンを強制終了および再起動するという方法です。

Network_resources_used リソースプロパティを使用すると、エンドユーザーは、アプリケーションリソースをバインドすべきネットワークアドレスリソースを構成できます。この機能が必要なリソースタイプの場合、そのリソースタイプの RTR ファイルで Network_resources_used プロパティを宣言する必要があります。

リソースグループをオンラインまたはオフラインにするとき、Sun Cluster は、データサービスリソースメソッドを呼び出す順番に従って、ネットワークアドレスの取り付け (plumb)、取り外し (unplumb)、「起動」または「停止」への構成を行います。43 ページの「start および stop メソッドを使用するかどうかの決定」を参照してください。

データサービスは、stop メソッドが戻るまでに、リソースグループのネットワークアドレスの使用を終了している必要があります。同様に、データサービスは、start メソッドが戻るまでに、リソースグループのネットワークアドレスの使用を開始している必要があります。

データサービスが、個々の IP アドレスではなく、INADDR_ANY にバインドする場合、データサービスリソースメソッドが呼び出される順番とネットワークアドレスメソッドが呼び出される順番には重要な関係があります。

データサービスの停止メソッドと起動メソッドでデータサービスのデーモンを終了および再起動する場合、データサービスは適切な時間にネットワークアドレスの使用を停止および開始します。

クライアントの再試行

ネットワーククライアントから見ると、テイクオーバーやスイッチオーバーは、論理ホストに障害が発生し、高速再起動しているように見えます。したがって、クライアントアプリケーションとクライアントサーバープロトコルは、このような場合に何回か再試行するように構成されていることが理想的です。すでに、単一サーバーの障害と高速再起動を処理するように構成されているアプリケーションとプロトコルは、上記のような場合も、リソースグループのテイクオーバーやスイッチオーバーとして処理します。無限に再試行するようなアプリケーションもあります。また、何回も再試行していることをユーザーに通知し、さらに継続するかどうかをユーザーにたずねるような、より洗練されたアプリケーションもあります。

索引

数字・記号

`#$upgrade`, 57
`#$upgrade_from`, directive, 58
`#$upgrade_from` ディレクティブ
 Anytime, 58
 At_creation, 59
 Tunable 属性の値, 58
 When_disabled, 59
 When_offline, 59
 When_unmanaged, 59
 When_unmonitored, 58

A

Anytime, `#$upgrade_from` ディレクティブ, 58
arraymax, リソースタイプの移行, 56
At_creation, `#$upgrade_from` ディレクティブ, 59

C

CCR, 61

D

directive, `#$upgrade_from`, 58
DSDL
 概要, 18
 実装場所, 18

L

libdsdev.so, DSDL, 18
libscha.so, RMAPI, 18

M

max, リソースタイプの移行, 56
min, リソースタイプの移行, 56

R

Resource Management API, 18
Resource_type, および移行, 56
RMAPI, 18
 libscha.so, 18
 実装場所, 18
RT_version, および移行, 56
RT_Version
 変更しないとき, 57
 変更するとき, 57
 目的, 57
RTR ファイル, および移行, 56

T

Tunable 属性のオプション, 56
 Anytime, 56
 At_creation, 56
 When_disabled, 56
 When_offline, 56

Tunable 属性のオプション (続き)

When_unmanaged, 56

When_unmonitored, 56

Tunable 属性の制約, 文書の要件, 62

Type_version リソースプロパティ, 58

Tunable 属性, 58

編集方法, 58

V

Validate, アップグレードのためにプロパティ値を検査, 62

Validate, およびアップグレード, 60

Vendor_id

および移行, 56

識別, 56

W

When_disabled, #\${upgrade_from} ディレクティブ, 59

When_offline, #\${upgrade_from} ディレクティブ, 59

When_unmanaged, #\${upgrade_from} ディレクティブ, 59

When_unmonitored, #\${upgrade_from} ディレクティブ, 58

あ

アップグレード, 文書の要件, 62

アップグレード対応, 定義済み, 55

アップグレード用文書, 62

か

完全修飾名, 取得方法, 57

き

技術サポート, 15

こ

コールバックメソッド, 概要, 17

て

ディレクティブ, 57

#\${upgrade_from}, 57

RTR ファイル内の配置, 57

Tunable 属性の制約, 57

デフォルトの Tunable 属性, 57

データサービス開発ライブラリ

DSDLを参照

デフォルトのプロパティ値

アップグレード用の新しい値, 61

および CCR, 61

および Sun Cluster 3.0, 62

およびアップグレード, 61

継承されるとき, 61

と

問い合わせ, 15

ふ

複数の登録バージョンの識別, RT_version, 56

古いリソースタイプのバージョンの起動メ

ソッドとの互換性

Monitor_check, 59

Postnet_stop, 59

Stop, 59

Update, 59

プロセス管理機能, 概要, 18

文書の要件

Tunable 属性の制約, 62

アップグレード用, 62

へ

ベンダーの識別, Vendor_id, 56

り

リソースタイプ

移行の要件, 55

複数のバージョン, 55

リソースタイプの移行, 55

リソースタイプ名

Sun Cluster 3.0, 58

制約, 56

バージョン接尾辞, 56

バージョン接尾辞なし, 58

