Sun Cluster 3.0 12/01 データサービス開発ガイド

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto,, CA 94303-4900
U.S.A. 650-960-1300

Part Number 816-3341
2001 年 12 月, Revision A

原典: *Sun Cluster 3.0 12/01 Data Services Developer's Guide*

Part No: 816–2025

Revision A

目次

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。

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。

UNIX コマンドの使い方

このマニュアルは、システムの停止、システムの起動、デバイスの構成など、UNIX® の基本的なコマンドや手順については説明しません。

このような情報については、次のマニュアルを参照してください。

- Solaris ソフトウェア環境の AnswerBook™ オンラインマニュアル
- このマニュアル以外にシステムに付属しているソフトウェアマニュアル
- Solaris オペレーティング環境のマニュアルページ

表記上の規則

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

表 P–1 表記上の規則

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

ただし AnswerBook2 では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェルプロンプト

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

- Bourne シェルおよび Korn シェルのプロンプト

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

■ スーパーユーザーのプロンプト

```
system# 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。

関連マニュアル

説明内容	タイトル	パート番号
ハードウェア	『*Sun Cluster 3.0 12/01 Hardware Guide*』	816-2023
データサービス	『*Sun Cluster 3.0 12/01* データサービスのインストールと構成』	816-3347
API 開発	『*Sun Cluster 3.0 12/01* データサービス開発ガイド』	816-3341
管理	『*Sun Cluster 3.0 12/01* のシステム管理』	816-3349
概念	『*Sun Cluster 3.0 12/01* の概念』	816-3337
エラーメッセージ	『*Sun Cluster 3.0 12/01 Error Messages Manual*』	816-2028
最新情報	『*Sun Cluster 3.0 12/01* ご使用にあたって』	816-3354

問い合わせについて

Sun Cluster 3.0 12/01 Error Messages Manual 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、Sun Cluster 3.0 12/01 Release Notes)
- Sun Cluster のリリース番号(たとえば、Solaris 8 3.0)

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
`prtconf -v`	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
`psrinfo -v`	プロセッサの情報を表示する
`showrev --p`	インストールされているパッチを報告する
`prtdiag -v`	システム診断情報を表示する
`scinstall -pv`	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
`scrgadm -pvv`	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
`scstat -g`	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、iPlanet™ 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おり、手順、操作を行うのはリソースタイプの開発者を想定しています。

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。

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。

- コールバックメソッド - RGM がクラスタ上のアプリケーションを制御するために作成します。
- RMAPI (Resource Management (リソース管理) API) - コールバックメソッドを作成するときに使用できる低レベルの API コマンドと関数のセットです。RMAPI は `libscha.so` ライブラリとして実装されています。
- プロセス管理機能 - クラスタ上のプロセスを監視および再起動します。
- DSDL (Data Service Development Library (データサービス開発ライブラリ)) - 低レベルの API とプロセス管理機能をより高いレベルでカプセル化して、コールバックメソッドの作成を容易にする機能を追加するライブラリ関数のセットです。DSDL 関数は `libdsdev.so` ライブラリとして実装されています。

図 1–1 に、これら要素の相互関係を示します。

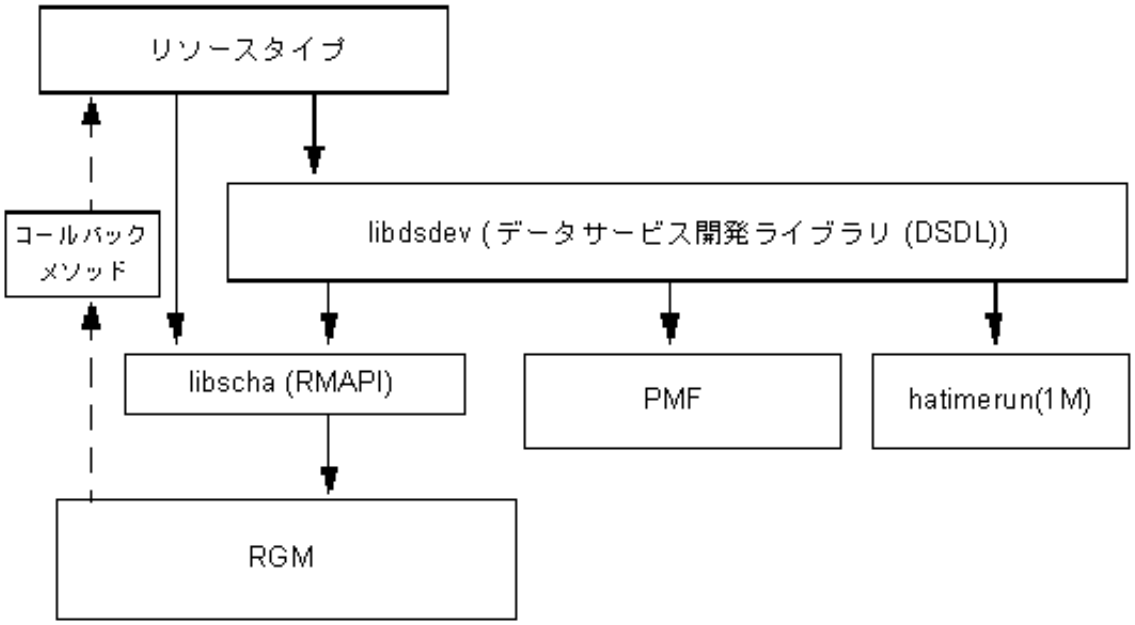


図 1–1　プログラミングアーキテクチャ

Sun Cluster パッケージには、データサービスを作成する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 8 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 関数を使用して、後者の場合は低レベルの API コマンドを使用して、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。RGM はまた、リソースに固有なモニター (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たりできます。

RGM はフェイルオーバーリソースとスケー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。スケー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 のモデル

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。次に、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。次に、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サーバーを実装します。クラスタ管理者は、財務カレンダ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用して、特定のタイプのリソースを管理します。

注 - ソフトウェアアプリケーションに加えて、リソースタイプは、他のシステムリソース (ネットワークアドレスなど) も表すことができます。

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 ファイルの形式は明確に定義されています。詳細は、33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(4)` のマニュアルページを参照してください。また、リソースタイプ登録

ファイルの例については、75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て、登録します。そして、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 3.0 12/01* 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たくさん作成して、各リソースに独自の名前とプロパティ値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 (どちらか一方だけ) にします。RGM は、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に、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現在オンラインであるノードのことを主ノードと呼びます。リソースグループは、自分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、この Nodelist プロパティがリソースグループのすべての潜在的な主ノード (つまり、マスター) を識別します。

リソースグループはまた、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含まれます。

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についての詳細は、50ページの「フェイルオーバーリソースの実装」と51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

RGM (Resource Group Manager) 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` プロセスはすべてお互いに通信し、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、RGM は管理されているすべてのリソースグループの可用性を維持するために、適切なマスター上で自動的にオンラインにします。
- 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す。
- クラスタ管理者は管理コマンドを発行して、次のアクションの 1 つを要求できます。
  - リソースグループをマスターする権利を変更する。
  - リソースグループ内の特定のリソースを有効または無効にする。
  - リソース、リソースグループ、またはリソースタイプを作成、削除、変更する。

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。このような活動のことを「再構成」と呼びます。状態の変更を個々のリソースにもたらすために、RGM はリソースタイプに固有なコールバックメソッドをそのリソース上で呼び出します。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。Sun Cluster フレームワークは、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 が各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て、各メソッドを 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行います。

- 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についての詳細は、 第 3 章 と rt_callbacks(1HA) のマニュアルページを参照してください。また、コールバックメソッドの使用例については、第 4 章 および 第 7 章を参照してください。

プログラミングインタフェース

データサービス用のコードを作成するために、リソース管理アーキテクチャは、低レベルの ベース API、ベース API 上に構築されているより高いレベルのライブラリ、および、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。

RMAPI (Resource Management (リソース管理) API)

データサービスは、RMAPI (Resource Management (リソース管理) API) の 低レベルルーチンを使って、システム内のリソースやリソースタイプ、リソースグループの情報にアクセスしたり、ローカルの再起動やフェイルオーバーを要求したり、リソースの状態を設定したりします。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 関数のどちらの形でも提供できます。RMAPI ルーチンの詳細については、scha_calls(3HA) と第 3 章を参照してください。サ

ンプルのデータサービス用の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 4 章を参照してください。

DSDL (Data Service Development Library(データサービス開発ライブラリ))

DSDL (Data Service Development Library (データサービス開発ライブラリ)) は、RMAPI の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保持しながら、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を監視したり、プロセスが停止したときに再起動したりできるプロセス管理機能 (`pmfadm`(1M) と `rpc.pmfd`(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)

ほとんどのアプリケーションにおいて、DSDL は、データサービスを構築するのに必要なほとんどまたはすべての機能を提供します。ただし、DSDL は低レベルの API に置き換るものではなく、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。事実、多くの DSDL 関数は `libscha.so` 関数を呼び出します。これと同じように、開発者は、DSDL を使用しながら `libscha.so` 関数を直接呼び出すことによって、データサービスの大半を作成することができます。`libdsdev.so` リブラリには、DSDL 関数が含まれています。

DSDL の詳細については、第 5 章と `scds_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。このツールでは、ターゲットアプリケーションと作成するデータサービスについての基本的な情報を入力します。Agent Builder は、ソースコードと実行可能コード (C 言語または Korn シェル)、カスタマイズされた RTR ファイル、および Solaris パッケージを一体化して、データサービスを生成します。

ほとんどのアプリケーションでは、Agent Builder を使用すると、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の場合、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、Agent Builder により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。少なくとも Agent Builder を使用することにより、独自の Solaris パッケージを生成することができます。

リソースグループマネージャ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 Web ベースのツールであり、次のような作業を行えます。

- 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 ソフトウェアによるデータサービスの構成

SunPlex Manager をインストールする方法や SunPlex Manager を使用してクラスタソフトウェアをインストールする方法については、『*Sun Cluster 3.0 12/01* 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 オブジェクトを管理するには、`scrgadm(1M)`、`scswitch(1M)`、および `scstat(1M) -g` の 3 種類の Sun Cluster 3.0 コマンドを使用できます。

scrgadm(1M) コマンドを使用すると、RGM が使用するリソースタイプ、リソースグループ、およびリソースオブジェクトを表示、作成、構成、および削除できます。scrgadm(1M) 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るプログラミングコンテキストで使用されます。scrgadm(1M) は、API が動作するクラスタ構成を構築するためのツールです。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(1M) 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(1M) コマンドは、指定したノード上にあるリソースグループのオンラインとオフラインを切り替えたり、リソースまたはその監視を有効または無効にしたりします。scswitch(1M) 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(1M) -g コマンドは、すべてのリソースグループとリソースについて、現在の動的な状態を表示します。

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。

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。要件についてのより詳細な情報が必要な場合、あるいは、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ている必要があり、さらに追加の要件も満さなければなりません。

- 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 は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、265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。263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もあります。詳細については、264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できる必要があります。詳細については、267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応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、51 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。

- RMAPI (Resource Management API (リソース管理 API)) - 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)
- DSDL (Data Service Development Library (データサービス開発ライブラリ)) - RMAPI の機能をカプセル化および拡張する、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)

また、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以外は提供しないため、Ksh でコーディングする場合は DSDL を使用できません。
2. Agent Builder を使用すると、必要な情報を入力するだけで、データサービスを生成できます。これには、ソースコードと実行可能コード、RTR ファイル、およびパッケージが含まれます。
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。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もできます。しかし、使用する方法に関わらず、ほとんどの場合、Agent Builder を使用して開発作業を開始することが重要です。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、数多く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、かなりの作業を節約できます。
- 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だけしか提供しません。つまり、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 (つまり、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) を使用して、`SUNWscdev` パッケージを開発マシンに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 (またはそれ以降) 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。

▼ 開発環境を設定する方法

この手順では、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。

```
# 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 ファイルを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 は自動的に RTR ファイルとデータサービスのコードを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は、リソースのタイプ、そのバージョン、API のバージョンなどを指定できると同時に、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活動状況を反映します。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の、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なものであるため、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て、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 で生成した後に、RTR ファイルを編集すれば特定の値を変更できます。

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す。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 (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 (`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 テンプレート バージョン 1.0
# smpl 用の登録情報とリソース
#
# ▼注：キーワードには大文字と小文字の区別がないため、
# 大文字小文字の使い方は自由である。
#
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_PRIMARIES;

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 と Vendor_id - リソースタイプ名を提供します。リソースタイプ名を指定するには、Resource_type プロパティを単独で使用するか (この例では、「smpl」)、Vendor_id を接頭辞として使用し、ドット (.) でリソースタイプと区切ります (この例では、「SUNW.smpl」)。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のがいいでしょう (ただし、RGM にはこの制限はありません)。一方、Agent Builder はリソースタイプ名からパッケージ名を明示的に生成しますので、それ自体には 9 文字の制限はありません。

- Rt_version—サンプルのデータサービスのバージョンを指定します。
- API_version—API のバージョンを指定します。「API_version = 2」は、データサービスが Sun Cluster バージョン 3.0 の下で動作することを示します。
- Failover = TRUE—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。つまり、フェイルオーバーデータサービスを指定します。詳細については、50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 が呼び出す各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。

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_PRIMARIES—データサービスをマスターできるノード上だけで、RGM が INIT、BOOT、FINI、および VALIDATE のメソッドを呼び出すことを指定します。RG_PRIMARIES 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、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、RGM が上記メソッドを呼び出すことを指定します。
- RT_basedir—コールバックメソッドパスのように、ディレクトリパスに /opt/SUNWsample/bin を付加して、相対パスを補います。
- START、STOP、VALIDATE など—RGM が呼び出す各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。

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 (つまり、「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 (つまり、「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。

```
...

# リソースプロパティの宣言は、リソースタイプ宣言の後に
# エントリを中括弧で囲んで指定する。
# プロパティ名宣言は、リソースプロパティエントリの左中括弧の
# 直後にある最初の属性でなければならない。
#
# メソッドタイムアウト用の最小値とデフォルト値を設定する。
{
        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;
{
```

(続く)

```
        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;
}

# ある期間内に再試行する回数。この回数を超えると、
# 当該ノード上ではアプリケーションを起動できないと判断される。
{
        PROPERTY = Retry_Count;
        MAX=10;
        DEFAULT=2;
        TUNABLE = ANYTIME;
}

# Retry_Interval に 60 の倍数を指定する。
# この値は秒から分に変換され、切り上げられる。
# たとえば、50 秒は 1 分に変換される。このプロパティは、
# 再試行回数 (Retry_Count) の間隔を指定する。
{
        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、特に理由がない限り、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。このようなプロパティに関する情報を以下に示します (詳細については、190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 メソッドが失敗した場合、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示します。
- Thorough_probe_interval、Retry_count、Retry_interval— 障害モニターによって使用されます。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に、必要に応じてリソースのリストを指定できます。

- Scalable— この値を FALSE に設定した場合、このリソースがクラスタネットワーキング (共有アドレス) 機能を使用しないことを示します。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。このプロパティの使用方法については、50ページの「フェイルオーバーリソースの実装」と51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 Agent Builder によって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— サーバーがリッスンするポートのリストを指定します。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に、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。

```
# 拡張プロパティ
#

# クラスタ管理者は、このプロパティの値によって、アプリケーション
# が使用する構成ファイルが格納されているディレクトリを指定する
# 必要がある。このアプリケーション (smpl) の場合は、
# PXFS 上にあるファイル (通常は named.conf) のパスを指定する。
{
 PROPERTY = Confdir_list;
 EXTENSION;
 STRINGARRAY;
 TUNABLE = AT_CREATION;
 DESCRIPTION = "The Configuration Directory Path(s)";
}

# 次の 2 つのプロパティは、障害モニターの再起動を制御する。
{
        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.";
}
# 検証用のタイムアウト値 (秒)
{
        PROPERTY = Probe_timeout;
        EXTENSION;
        INT;
        DEFAULT = 30;
        TUNABLE = ANYTIME;
        DESCRIPTION = "Time out value for the probe (seconds)";
}

# PMF 用の子プロセス監視レベル (pmfadm の -C オプション)。
# デフォルトの -1 は、pmfadm の -C プションを使用しないことを示す。
# 0 以上の値は、必要な子プロセス監視レベルを示す。
{
        PROPERTY = Child_mon_level;
        EXTENSION;
        INT;
        DEFAULT = -1;
        TUNABLE = ANYTIME;
        DESCRIPTION = ``Child monitoring level for PMF";
}
# ユーザー追加コード -- BEGIN VVVVVVVVVVV
# ユーザー追加コード -- 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いくつか作成します。

- 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、障害モニター自身の再起動を制御します。
- Child_mon_level— PMF が行う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`(1M) 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用して、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に (つまり、RGM がメソッドを呼び出した時点で) エラー終了するため、どのようなコールバックメソッドからも `scrgadm` を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。つまり、同じリソース上で STOP メソッドまたは MONITOR_STOP メソッドを (同じパラメータで) 何回も連続で呼び出しても、一回だけ呼び出したときと同じ結果になることを意味します。

また、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ており、動作していなくても、STOP メソッドと MONITOR_STOP メソッドは 0 (成功) を戻す必要があるということも意味します。

注 - INIT、FINI、BOOT、UPDATE 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。

アプリケーションの制御

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、RGM はコールバックメソッドを使用して、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、RGM は、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 メソッドは、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方法を RGM に提供するように実装する必要があります。

ローカルノード上でリソースが起動され、利用可能になるまで、START メソッドは戻ってはなりません。初期化に時間がかかるリソースタイプでは、十分な長さのタイムアウト値をその START メソッドに設定する必要があります。リソースタイプ登録ファイルで Start_timeout プロパティのデフォルト値と最小値を設定します。

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 STOP メソッドをそのリソースグループ内のリソース上で呼び出して、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後、RGM は、ノード 2 上でそのリソースグループをもう一度オンラインにします。

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、STOP メソッドは戻ってはなりません。最も安全な STOP の実装方法は、ローカルノード上で資源に関連するすべてのプロセスを終了することです。シャットダウンに時間がかかるリソースタイプでは、十分な長さのタイムアウト値をその STOP メソッドに設定する必要があります。リソースタイプ登録ファイルで Stop_timeout プロパティを設定します。

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、すべてのエラー状態から回復するように、STOP と MONITOR_STOP メソッドを実装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 と 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行う必要があります。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後に、

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 (もしあれば)。
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 (もしあれば)。

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 (もしあれば)。
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 (もしあれば)。
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後に、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に、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に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後、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」に構成された後だけです。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す。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、一度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、つまり、起動または再起動されたノード上で、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 (3HA) または DSDL 関数 scds_fm_action(3HA) 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 は、 リソースモニターを直接呼び出すことはありませんが、リソース用のモニターを自動的に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 の scha_control(3HA) 関数と (scds_fm_action を呼び出す) DSDL の scha_control(3HA) 関数を使用すると、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 1 つとして MONITOR_CHECK 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な使用方法ですが、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 を使って障害モニターを実装する例については、75 ページの「障害モニターの定義」を参照してください。DSDL を使って障害モニターを実装する例については、111 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 3.0 12/01* 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。または、scha_cluster_get(1HA)(3HA) 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

Resource Management API と DSDL には、リソースモニターやリソース制御コールバックを実装するためのプロセス管理機能が備わっています。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(1M) と rpc.pmfd(1M) — プロセス監視機能 (PMF) は、プロセスとその子孫プロセスを監視し、停止した場合は再起動する方法を提

供します。この機能は、pmfadm(1M) コマンド (監視するプロセスを起動および制御する) と rpc.pmfd(1M) デーモンからなります。

- halockrun(1M) — ファイルロックを保持したまま、子プログラムを実行するプログラム。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(1M) — タイムアウト制御下で、子プログラムを実行するプログラム。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の scds_hatimerun(3HA) 関数を提供します。

DSDL は、PMF 機能を実装するための scds_pmf_*(3HA) 関数セットを提供します。DSDL の PMF 機能の概要と、個々の関数のリストについては、183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。

リソースが作成されたとき、および、リソースまたはリソースグループ (リソースを含む) 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 任意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、185ページの「リソースタイププロパティ」 または 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 を呼び出すのは、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。

RGM は、任意の 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後に、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ー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。

スケーラブルリソースの実装

スケーラブルリソースは、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。スケーラブルリソースには、Sun Cluster HA for iPlanet 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。

- `Network_resources_used` – このリソースが使用する共有アドレスリソースを識別します。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。`scsetup`(1M) コマンドと SunPlex Manager 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。
- `Load_balancing_policy` – 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 RTR ファイルに明示的に設定しても、デフォルトの LB_WEIGHTED を使用してもかまいません。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。

  - LB_WEIGHTED – 負荷は、`Load_balancing_weights` プロパティに設定されたウェイトに従って、さまざまなノード間に分散されます。
  - LB_STICKY – スケーラブルサービスのクライアント (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。
  - LB_STICKY_WILD – 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ているクライアント (クライアントの IP アドレスで識別される) は、着信しているポート番号に関わらず、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 `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` – 各ノードに送信される負荷を指定します。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、このノードのウェイトをアクティブなインスタンスのすべてのウェ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 – サーバーが通信するポートを識別します。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 (つまり、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 3.0 12/01 の概念*』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。

```
Port_list=80/tcp,40/udp
```

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と、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。

他にも、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と、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(1M)` 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も検証します。

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。Sun Cluster でアポイントメントカレンダ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リソースを構築して、アポイントメントカレンダリソースを 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`(1M) 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

RMAPI のリファレンス

この章では、RMAPI (Resource Management (リソース管理) API) 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。詳細は、Resource Management API 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。

- 60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_open_get_close(3HA)`)
  - `scha_resource_setstatus(1HA)(3HA)`
  - `scha_resourcetype_get(1HA)` `scha_resourcetype__open_get_close(3HA)`
  - `scha_resource_resourcegroup_get(1HA)(3HA)` `scha_resource_resourcegroup_open_get_close(3HA)`
  - `scha_control(1HA)(3HA)`
  - `scha_cluster_get(1HA)` `scha_resource_cluster_open_get_close(3HA)`
  - `scha_cluster_getlogfacility(3HA)`
  - `scha_cluster_getnodename(3HA)`
  - `scha_strerror(3HA)`
- 66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
- PRENET_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。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。特に注記しない限り、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(1HA) – RGM の制御下にあるリソースまたはリソースタイプについての情報にアクセスします。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(1HA) – RGM の制御下にあるリソースの Status プロパティと 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(1HA) – このコマンドは、C 関数 scha_resourcetype_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(1HA) – RGM の制御下にあるリソースグループについての情報にアクセスします。 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。

- scha_control(1HA) – RGM の制御下にあるリソースグループの再起動、または、異なる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(1HA) – 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。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。特に注記しない限り、各関数と同じ名前のマニュアルページがあります。

C 関数の出力引数や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) – これらの関数は一緒に、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。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。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。このメッセージは致命的なエラーメッセージではないため、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 の制御下にあるリソースの Status プロパティと 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

これらの関数は一緒に、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、リソースタイプ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。このメッセージは致命的なエラーメッセージではないため、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) でアクセスできます。

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)–これらの関数は一緒に、RGM に管理されているリソースグループについて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。このメッセージは致命的なエラーメッセージではないため、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) でアクセスできます。

- scha_control(3HA)– RGM の制御下にあるリソースグループの再起動、または、異なる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) – これらの関数は一緒に、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。

  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) でアクセスできます。

  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。このメッセージは致命的なエラーメッセージではないため、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ているシステムログ機能番号を戻します。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) – 関数が呼び出される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_ 関数の 1 つから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) と共に使用すると、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予想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。これらすべて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をクラスタノード上でオンラインにするとき、その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。

  ローカルノード上でリソースが起動され、利用可能になるまで、START メソッドは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。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をクラスタノード上でオフラインにするとき、その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。

  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閉じるまで、STOP メソッドは終了してはなりません。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閉じるまで、STOP メソッドは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 – この任意メソッドは、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) に呼び出され、一度だけ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 – この任意メソッドは、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 – この任意メソッドは、INIT と同様にリソースの初期化を実行します。ただし、リソースを含むリソースグループがすでに RGM の管理下に置かれている状態で、新たにクラスタに参加したノード上で呼び出されます。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 – この任意メソッドは、リソースが作成されるとき、および、リソースまたはリソースグループ (リソースを含む) 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 – この任意メソッドは、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。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 – この任意メソッドは、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」に構成される前に呼び出され、特別な起動アクションを行います。
- POSTNET_STOP – この任意メソッドは、同じリソースグループ内のネットワークアドレスが「停止」に構成された後に呼び出され、特別なシャットダウンアクションを行います。

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 – この任意メソッドは、リソースが起動した後に呼び出され、リソースを監視するモニターを起動します。
- MONITOR_STOP – この任意メソッドは、リソースが停止する前に呼び出され、リソースのモニターを停止します。
- MONITOR_CHECK – この任意メソッドは、リソースグループがノードに再配置される前に呼び出され、ノードの信頼性を査定します。

第 4 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。

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 33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、210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、その後に、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.
#
# Registration information for Domain Name Service (DNS)
#


#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せん。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、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)。

  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。

- 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。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 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 (*decrease* 状態)。
{
 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を下すま
# でに、指定された期間内 (Retry_Interval) に行なう再試行回数
{
        PROPERTY = Retry_Count;
        MIN=0;
        MAX=10;
        DEFAULT=2;
        TUNABLE = ANYTIME;
}

# Set Retry_Interval には 60 の倍数を指定する。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。
# このプロパティ値は再試行回数 (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 = ````;
}
```

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、206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。詳細については、110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。プロパティを分類する方法については、r_properties(5) のマニュアルページまたは 190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を設定し、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 (DNS) の場合は、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)'';
}
```

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とき、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。

- 81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」.
- 82ページの「PMF_TAG と SYSLOG_TAG 変数の宣言」.
- 83ページの「関数の引数の構文解析」.
- 85ページの「エラーメッセージの生成」.
- 85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

 # SIGTERM シグナルをデータサービスに送信し、
 # 合計タイムアウト値の 80% だけ待機する。
 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
...
# モニターが動作しているかどうかを調べ、動作
# している場合は強制終了する。
if pmfadm -q $PMF_TAG.monitor; then
```

(続く)

```
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についての詳細は、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。詳細は、104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。

```
#############################################################################
# プログラム引数を解析する。
#
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
         ;;
         RESOURCE_NAME=$OPTARG
         ;;
      G)
         # リソースが構成されるリソースグループ
         # の名前
         RESOURCEGROUP_NAME=$OPTARG
         ;;
         RESOURCEGROUP_NAME=$OPTARG
         ;;
      T)
         # リソースタイプの名前
         RESOURCETYPE_NAME=$OPTARG
         ;;
         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

の値だけを戻します。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戻します。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、44ページの「START と 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、START 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、81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、213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。

```
# 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}'`
```

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。89ページの「START の終了状態」 を参照してください。

```
# Check if $CONFIG_DIR is accessible.
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
```

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最大回数の値を取得する。
RETRY_CNT=scha_resource_get -O Retry_Count -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME
# RTR ファイルから再試行最大期間の値を取得する。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、端数は切り捨て
# 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、
# $RETRY_COUNT の回数だけ、クラッシュおよび再起動できる。どちらかの
# 値以上クラッシュした場合、PMF は再起動をやめる。
# <$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。
#
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する。
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、81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、215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、停止処理を規則正しく行うことです。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 シグナルを送信します。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 メソッドは呼び出し回数に依存し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二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% は予約されています)。次の例を参照してください。

```
STOP_TIMEOUT=scha_resource_get -O STOP_TIMEOUT -R $RESOURCE_NAME
\

-G $RESOURCEGROUP_NAME
((SMOOTH_TIMEOUT=$STOP_TIMEOUT * 80/100))

((HARD_TIMEOUT=$STOP_TIMEOUT * 15/100))
```

STOP メソッドは pmfadm -q を使用し、DNS デーモンが動作しているかどうかを確認します。動作している場合、STOP メソッドはまず pmfadm -s を使用して TERM シグナルを送信し、DNS プロセスを終了します。このシグナルを送信してからタイムアウト値の 80% が経過してもプロセスが終了しない場合、STOP メソッドは SIGKILL シグナルを送信します。このシグナルを送信してからタイムアウト値

の 15% が経過してもプロセスが終了しない場合、STOP 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。

pmfadm がプロセスを終了した場合、STOP メソッドはプロセスが停止したことを示すメッセージを記録し、成功で終了します。

DNS プロセスが動作していない場合、STOP メソッドは DNS プロセスが動作していないことを示すメッセージを記録しますが、成功で終了します。次のコード例に、STOP メソッドがどのように pmfadm を使用して DNS プロセスを停止するかを示します。

```
# 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; then
 # SIGTERM シグナルをデータサービスに送信し、合計タイムアウト値
 # の 80% だけ待つ。
 pmfadm -s $RESOURCE_NAME.named -w $SMOOTH_TIMEOUT TERM
 if [ $? -ne 0 ]; then
  logger -p ${SYSLOG_FACILITY}.err \
      -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
      ''${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、合計タイムアウト値の 15% だけ待つ。
  pmfadm -s $PMF_TAG -w $HARD_TIMEOUT KILL
  if [ $? -ne 0 ]; then
      logger -p ${SYSLOG_FACILITY}.err \
      -t [$SYSLOG_TAG]
      ''${ARGV0} Failed to stop HA-DNS; Exiting UNSUCCESFUL''

      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、
 # 成功で終了する。
 logger -p ${SYSLOG_FACILITY}.err \
         -t [$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、成功で終了する。
logger -p ${SYSLOG_FACILITY}.err \
    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
```

(続く)

```
    ''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、81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、219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。*probefail* 変数と *retries* 変数は 0 に初期化されます。

```
# nslookup 応答用の一時ファイルを設定する。
DNSPROBEFILE=/tmp/.$RESOURCE_NAME.probe
probefail=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する。
- 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する。
- nslookup の戻りコード (成功または失敗) に基づいて、*probefail* 変数を設定する。
- *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する。

次に、while ループコードを示します。

```
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
    probefail=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。
```

(続く)

```
if [ $probefail -eq 0 ]; then
     # nslookup 照会に応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
        probefail=1
     else
        if [ $SERVER != $DNS_HOST ]; then
          probefail=1
        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、219

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。

- gettime ユーティリティーを使用すると、再起動間の時間を追跡できます。このユーティリティーは 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。99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え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 下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。
                # 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_NAMÈ
                STOP_METHOD≅scha_resource_get -O STOP \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                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。
                # this resource.
```

(続く)

```
                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 下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。代わりに、同じクラスタ内にある別のノード
                # 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。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、81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、224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 プロパティを取得することによって、検証プログラ
# ム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、81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、226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、
# 検証プログラムを停止する。
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、81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、228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 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、104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、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。

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

VALIDATE メソッドを呼び出すとき、RGM は追加の引数 (更新されるプロパティとその値など) を他のメソッドに渡します。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であるかどうか、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であるかどうか、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 メソッドや他のコールバックメソッドに渡される引数についての詳細は、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
         ;;
         RESOURCE_NAME=$OPTARG
         ;;
      G)
         # リソースが構成されるリソースグループ
         # の名前
         RESOURCEGROUP_NAME=$OPTARG
         ;;
         RESOURCEGROUP_NAME=$OPTARG
         ;;
      T)
         # リソースタイプの名前
         RESOURCETYPE_NAME=$OPTARG
         ;;
         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。
      c)
         # Validate メソッドがリソースの作成中に
```

(続く)

```
          # 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}'`


          PROPERTY=`echo $OPTARG | awk -F= '{print $1}'`
          VAL=`echo $OPTARG | awk -F= '{print $2}'`
```

```
          # If # 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のかどうか、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 が作成されます。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リソースグループのプロパティの設定に成功した後に、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 5 章

DSDL (データサービス開発ライブラリ)

この章では、DSDL (Data Service Development Library (データサービス開発ライブラリ)) を構成する API (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。

DSDL の概要

DSDL API は、RMAPI の最上位の階層を形成します。したがって、これは は RMAPI に置き換るものではなく、RMAPI 機能をカプセル化および拡張するためのものです。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、135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、146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() ユーティリティーは、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*l 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 7 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。

第 6 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。プログラマは DSDL のユーティリティー `scds_initialize()` を使うことで、統一された方法でこのような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ー関数は、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、リソースタイプのとて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。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()` ユーティリティー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_PRIMARIES` 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_PRIMARIES` です (`rt_reg(4)` 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。

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。

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

  if (scds_initialize(&handle, argc, argv)!= SCHA_ERR_NOERR) {
  return (1);/* 初期設定のエラー */
  }
  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、次の章を参照してください)。

```
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); /* リソースプロパティの設定が無効 */
  }
  return (0); /* 設定が有効 */
}
```

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() ユーティリティー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。

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

  if (scds_initialize(&handle, argc, argv)!= SCHA_ERR_NOERR) {
  return (1); /* 初期設定のエラー */
  }
  if (svc_validate(handle) != 0) {
  return (1); /* 設定値が無効 */
  }
  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のかもしれません。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) と『*Sun Cluster 3.0 12/01 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ら戻りコード 0 を、問題を検出したら 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() ユーティリティーを使って、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ーは、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 ユーティリティー scds_pmf_stop() で十分なはずです。このユーティリティーは、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ーの詳細については、183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); /* 初期設定のエラー */
}
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() ユーティリティーを使って起動したら、scds_pmf_stop() ユーティリティー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が確認されます。120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。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` に渡すことはできます)。

INIT、FINI、および BOOT メソッド

これらのメソッドは、「1 度だけのアクション」を行なうためのものです (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(1m) 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リ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象的です。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。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。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トリから
  * ポート値を取得する。
  */
  ht1 = gethrtime();   /* 検証開始時刻を保存する。*/
  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);
  }            /* 各ネットリソース */
  }       /* 検証を続ける。 */
```

第 7 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。

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/xfs` 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。`xfs` 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/xfs -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 のポート番号

xfs サーバーデーモンがリッスンする 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、xfs 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、xfs 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 xfs 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。

命名規則

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、33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 7–1

```
{
 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リッスンするポートを指定します。

例 7–2

```
{
 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 xfs 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。xfnts_start は、scds_initialize を最初に呼び出します。

注 - 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、134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。

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、xfs 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、150ページの「xfnts_validate メソッド」を参照してください。

例 7–3

```
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、xfs 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、xfs デーモンを起動するためのコマンドを示します。

```
xfs -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 7–4

```
 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、xfs 用のコマンド行を形成します。

例 7–5

```
  /* 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 7–6

```
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リ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 7–7

```
/* 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 7–8

```
/* 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 7–9

```
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 7–10

```
#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戻る。*/
   scds_free_netaddr_list(netaddr);
   return (0);
  }
   /* サービスがなん度も失敗する場合は、
  iscds_svc_wait()を呼び出す。
   != 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);
```

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、134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。xfnts_stop は、scds_initialize を最初に呼び出します。

注 - 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、134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_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 7–11

```
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、134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、134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 7–12

```
 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.");
  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リ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 が割り当てたリソースを再利用します。詳細については、134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。xfnts_monitor_stop は、scds_initialize を最初に呼び出します。

注 - 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、134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 7–13

```
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 が割り当てたリソースを再利用します。詳細については、134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、xfs 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、150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 7–14

```
 /* 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。

xfonts_probe のメインループ

xfonts_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、xfonts_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 7–15

```
/* 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、休止状態になります。

例 7–16

```
timeout = scds_get_ext_probe_timeout(scds_handle);
for (;;) {

 /*
  * 連続する検証の間、Throrough_probe_interval で指定された期間、休止状態になる。
  *  xfnts_probe メソッドは次のようなループを実装します。
  */
 (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 7–17

```
 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
     * アクションを行う。
     */
    (void) scds_fm_action(scds_handle,
        probe_result, (long)dt);
  } /* ネットリソースごと */
}  /* 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 7–18

```
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との接続を行う。
  * 実際には、接続は probe_timeout 値の 95% に達するとタイムアウトする。
  */
 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));
  /* これは致命的な障害である。 */
  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。
  * そうでない場合、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,
     "Failed to disconnect to port %d of resource %s.",
     port, scds_get_resource_name(scds_handle));
 /* これは部分的な障害である。 */
 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。さもないと、
 * 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 xfs 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。

次回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るとき、また、管理アクションがリソースまたは (リソースが属する) グループのプロパティを更新するとき、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。つまり、任意のノード上で VALIDATE メソッドから失敗を示す終了コードが戻されると、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 7–19

```
 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); /* 検証は失敗。*/
 }
```

- 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 7–20

```
 (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 7–21

```
 if (stat("/usr/openwin/bin/xfs", &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 7–22

```
 scds_port_list_t *portlist;
 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); /* 検証は失敗。*/
 }
#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); /* Validation Failure */
 }
#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 7–23

```
 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。 */
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 7–24

```
finished:
 scds_free_net_list(snrlp);
 scds_free_port_list(portlist);
 return (rc); /* 検証の結果を戻す。*/
```

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、134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 7–25

```
 * 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 8 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。入力された情報に基づいて、Agent Builder は次のものを生成します。

- ソースファイルセット - フェイルオーバーまたはスケーラブルのリソースタイプ用の C 言語コードまたは korn シェル (ksh) スクリプト。リソースタイプのメソッドコールバックに対応しています。
- カスタマイズされた 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ースクリプト。また、これらのファイルをどのように使用するかを説明する、カスタマイズされたマニュアルページ
- Solaris パッケージ。バイナリ (C ソースの場合)、RTR ファイル、およびユーティリティースクリプトが含まれます。

Agent Builder はネットワーク対応アプリケーション (ネットワークを使用してクライアントと通信するアプリケーション) とネットワーク非対応 (スタンドアロン) アプリケーションをサポートします。また、Agent Builder を使用すると、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、つまり、Process Monitor Facility (PMF) で個々のプロセスを監視および再起動する必要があるアプリケーション用の

リソースタイプも生成できます。詳細については、165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 を使用する前に行なっておく作業について説明します。また、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、28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、Agent Builder は、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、つまり、プロパティの妥当性検査を追加したり、Agent Builder がエクスポーズしないパラメータを調節したりするためのコードを追加しなければならない場合もあります。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 を別途インストールする必要はありません。これは、Sun Cluster ソフトウェアの標準インストールの一環として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。詳細については、『*Sun Cluster 3.0 12/01* ソフトウェアのインストール』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。

- Java が $PATH 変数に含まれているかどうか。Agent Builder は Java (Java Development Kit バージョン 1.2.2_05a 以降) に依存するため、Java が $PATH に存在しない場合、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かどうか。
- Tcc コンパイラが $PATH 変数に含まれているかどうか。Agent Builder は $PATH 変数内で最初に現れる cc を使用して、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を識別します。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、159ページの「Create 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
```

図 8–1 のような、Agent Builder の初期画面が表示されます。

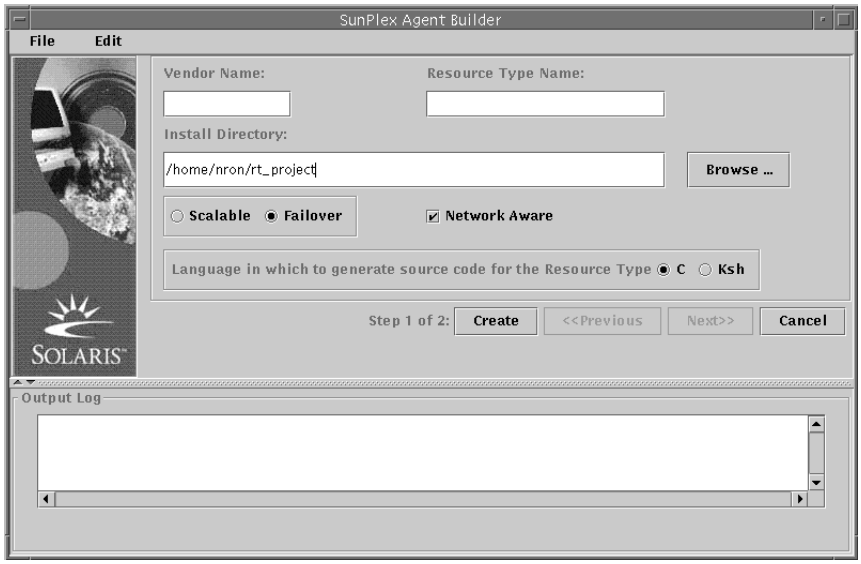


図 8–1　初期画面

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、159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. **Create** 画面—この画面では、作成するリソースタイプについての基本的な情報を提供します。たとえば、リソースタイプの名前や、生成されるファイル用のインストールディレクトリを入力します。また、作成するリソースの種類 (スケー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 または ksh) も指定できます。この画面に必要な情報をすべて入力し、「Create」を選択すると、対応する出力が生成されます。この後、Configure 画面に進むことができます。
2. **Configure** 画面—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、162ページの「Configure 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インストールディレクトリから Agent Builder を起動する場合、Agent Builder は Create と Configure 画面を既存のリソースタイプの値に初期化します。

Agent Builder の画面上にあるボタンやメニューの使用方法については、173ページの「ナビゲーション」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フェースと同様に、2 段階の入力手順が必要です。ただし、GUI に情報を入力するのではなく、2 つのコマンド、`scdscreate`(1HA) と `scdsconfigure`(1H) にパラメータを渡します。詳細については、各コマンドのマニュアルページを参照してください。

Create 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 Create 画面に必要な情報を入力します。図 8–2 に、各フィールドに情報を入力した後の Create 画面を示します。

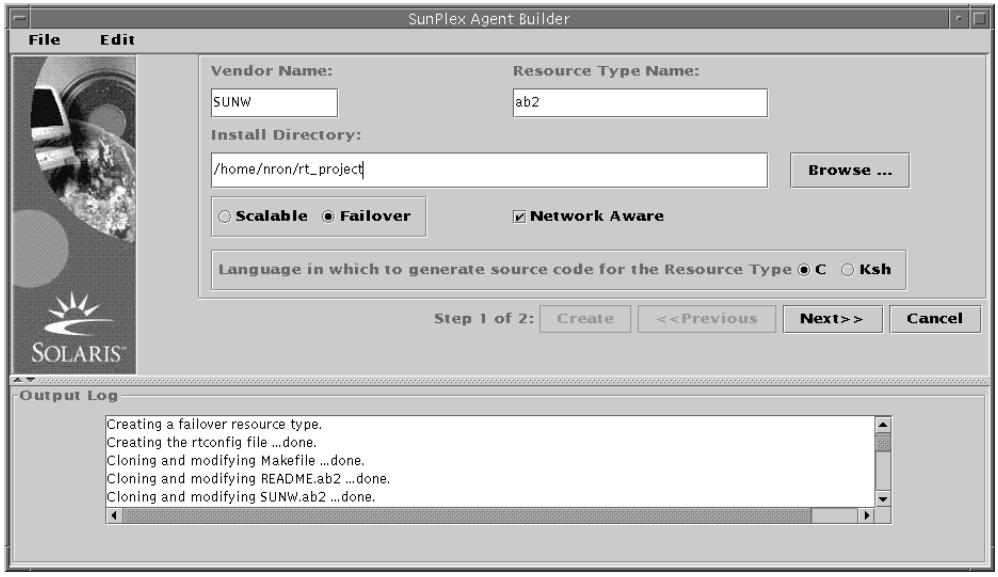


図 8–2　Create 画面

Create 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。

- **Vendor Name —** 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。
- **Resource Type Name —** リソースタイプ名。英数字文字だけを使用します。

注 - ベンダー名とリソースタイプ名の両方で、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。

- **Install Directory —** 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 つのインストールディレクトリには 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、「Browse」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、インストール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、リソースタイプ名が `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(167ページの「ディレクトリ構造」を参照)。

- **Failover or Scalable —** ターゲットリソースタイ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指定します。
- **Network Aware —** 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。ksh 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。たとえば、「Ksh」ボタンにチェックマークを入れると、Agent Builder は Network Aware チェックボックスにチェックマークを入れ、チェックボックスをグレー表示にします。
- **C or 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ておき、その後、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、166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。

注 - `cc` コンパイラが `$PATH` に存在しない場合、Agent Builder は「C」オプションボタンをグレー表示し、「Ksh」ボタンにチェックマークを入れます。異なるコンパイラを指定する方法については、157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、「Create」ボタンをクリックします。画面の一番下にある「Output Log」には、Agent Builder が行ったアクションが表示されます。「Edit」メニューの「Save Output Log」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 Create 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、「Next」ボタンをクリックすると、Configure 画面に進むことができます。Configure 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、Create) 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、165 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。

Configure 画面の使用

Configure 画面 (図 8–3 を参照) は、Agent Builder の Create 画面でリソースタイプの作成が完了し、「Next」ボタンを押した後に表示されます。リソースタイプの作成が完了していなければ、Configure 画面にはアクセスできません。

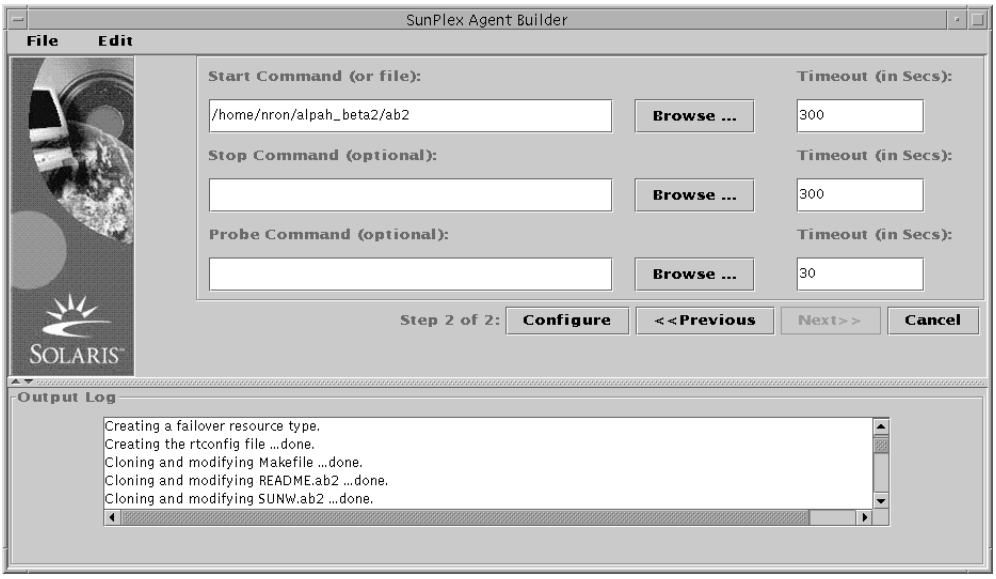


図 8–3　Configure 画面

Configure 画面には、次のフィールドがあります。

- **Start Command —** 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、「Browse」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 `$hostnames` 変数を使用できます。詳細については、164ページの「Agent Builder の `$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 ("") で囲んではなりません。

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 Start Command 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、165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。

- **Stop Command —**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、「Browse」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 `$hostnames` 変数を使用できます。詳細については、164ページの「Agent Builder の `$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。

- **Probe Command —**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) の範囲の終了状態を戻すコマンド。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、「Browse」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。

  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 `$hostnames` 変数を使用できます。詳細については、164ページの「Agent Builder の `$hostnames` 変数の使用」を参照してください。

- **Timeout (**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リッスン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、Configure 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リッスン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ンマで区切って $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 Configure 画面の Start Command 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インストールディレクトリ **(**`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 **Create** 画面と **Configure** 画面に、そのリソースタイプ用の値がロードされます。
   b. 「**File**」メニューの「**Load Resource Type**」コマンドを使用します。
2. **Create** 画面でインストール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、「Browse」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「Create」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。
   「Create」を選択して、リソースタイプを作成します。「Next」を選択して、Configure 画面に進みます。「Configure」を選択して、リソースタイプを構成します。最後に、「Cancel」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 がエクスポーズ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 vvvvvvvvvvvvvvv */
/* User added code -- END   ^^^^^^^^^^^^^^^ */
```

注 - これらのコメント文は Ksh 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。

また、*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、33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。

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。インストールディレクトリは Create 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、インストールディレクトリの下に、Create 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インストール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 8–1

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 はベンダー名とリソースタイプ名をピリオド (.) 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。たとえば、次のように指定します。 man -M *install_directory*/SUNWftp/man removeftp.
pkg	最終的なパッケージが格納されます。
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 ユーティリティースクリプトが格納されます。詳細については、170ページの「ユーティリティー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リソースタイプ名を追加します。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 が生成する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、`START` 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、18ページの「RGM のモデル」、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に、Agent Builder はコールバックメソッドとして機能する 8 つの実行可能なプログラム (C) またはスクリプト (ksh) を *install_directory*/*rt_name*/`bin` ディレクトリに生成します。

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、次のファイルを生成します。

- 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ープログラム (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。167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。

ユーティリティー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ースクリプトに加え、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、スクリプト名にリソースタイプ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、リソースタイプ名が 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、リソースタイプ名が 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、33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、ベンダー名とリソースタイプ名を連結したものです (たとえば、SUNWftp)。*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ー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と、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 remove*rt_name* スクリプトを実行してから、任意のノードで pkgrm を実行する。
- 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ナップ処理を行った後、(必要であれば同時に) 残りのノードで pkgrm を実行する。

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

Agent Builder は、Create 画面と Configure 画面で入力された情報が格納されている構成ファイル rtconfig をインストールディレクトリに生成します。Agent Builder を既存のリソースタイプのインストールディレクトリから実行した場合、あるいは、「File」メニューの「Load Resource Type」コマンドを使用して、既存のリソースタイプをロードした場合、Agent Builder は rtconfig ファイルを読み取って、Create 画面と Configure 画面を既存のリソースタイプの値で初期化します。これは、既存のリソースタイプのクローンを作成するときに有用です。詳細については、166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 (Create と Configure) 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する。
- 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する。
- 相互に排他的なラジオボタンセットの 1 つを選択する (たとえば、「Scalable」または「Failover」)。
- オン/オフボックスにチェックマークを入れる。たとえば、「Network Aware」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。

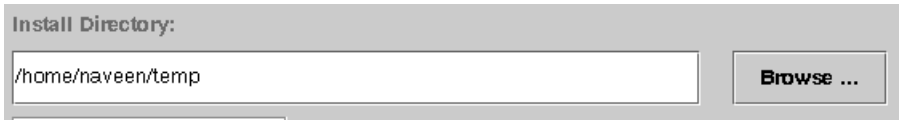
たとえば、**Create** 画面において、必要なフィールドに入力し、希望のオプションにチェックマークを付けてから、画面の下にある「Create」ボタンをクリックします。この時点で、以前の画面は存在しないので、「Previous」ボタンはグレー表示されます。また、この作業が完成するまで次の手順には進めないので、「Next」ボタンもグレー表示されます。

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、「Next」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、「Cancel」ボタンだけが強調表示されます。

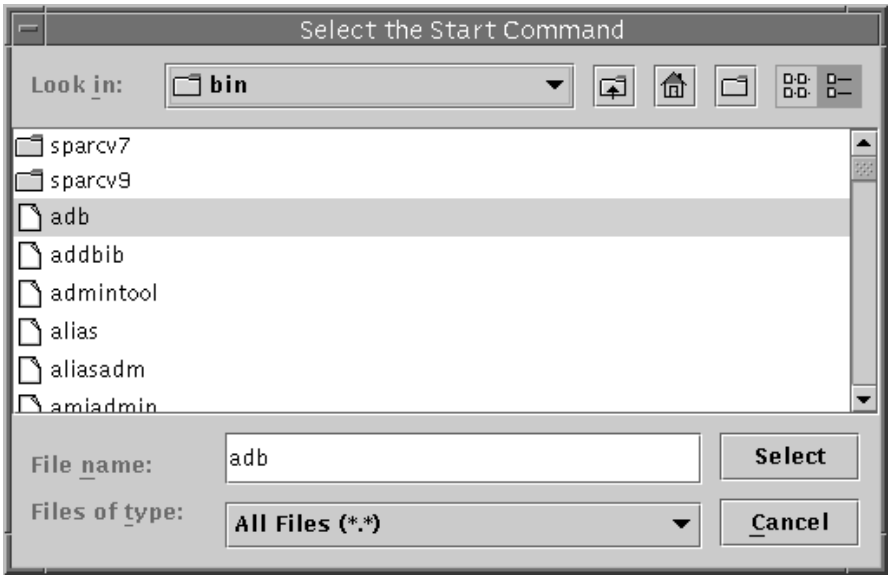
「Cancel」 ボタンを押すと、いつでも Agent Builder を終了できます。

「Browse」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、「Browse」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。



「Browse」をクリックすると、次のような画面が表示されます。



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「File name」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、「Select」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、「Open」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、「Close」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、「File」と「Edit」メニューがあります。

「File」メニュー

「File」メニューでは、次の 2 つのコマンドを使用できます。

- **Load Resource Type —**既存のリソースタイプをロードします。Agent Builder が提供するブラウズ画面を使用して、既存のリソースタイプ用のインストール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。「Load Resource Type」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、166 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。
- **Exit —**Agent Builder を終了します。Create 画面または Configure 画面で「Cancel」をクリックしても、Agent Builder を終了できます。

「Edit」メニュー

「Edit」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。

- **Clear Output Log —** 出力ログから情報を消去します。「Create」または「Configure」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。
- **Save Log File —**ログの出力をファイルに保存します。Agent Builder が提供するブラウズ画面を使用すると、ディレクトリを選択して、ファイル名を指定できます。

第 9 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。

DSDL 関数

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`(3HA) – リソースを割り当て、DSDL 環境を初期化します。
- `scds_close`(3HA) – `scds_initialize`(3HA) 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`(3HA) – 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`(3HA) – 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`(3HA) – 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`(3HA) – 指定した拡張プロパティの値を取得します。
- `scds_free_ext_property`(3HA) – `scds_get_ext_property`(3HA) 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`(3HA) – リソースグループをフェイルオーバーします。
- `scds_restart_rg`(3HA) – リソースグループを再起動します。

- scds_restart_resource(3HA) – 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(3HA) – タイムアウトを適用してコマンドを実行します。
- scds_error_string(3HA) – 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(3HA) を使用して、コマンド行引数を解析します。scds_initialize(3HA) 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(3HA) – 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(3HA) – 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(3HA) – scds_get_rg_hostnames(3HA) または scds_get_rs_hostnames(3HA) 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(3HA) – scds_get_rg_hostnames(3HA) または scds_get_rs_hostnames(3HA) 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(3HA) – 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(3HA) – scds_get_port_list(3HA) 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(3HA) – scds_get_port_list(3HA) 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(3HA) – 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(3HA) – scds_get_netaddr_list(3HA) 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(3HA) – scds_get_netaddr_list(3HA) 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(3HA) – プロセスとの TCP 接続を確立します。
- scds_tcp_read(3HA) – 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(3HA) – 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(3HA) – 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(3HA) – 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。PMF 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(3HA) – 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(3HA) – PMF を使用して障害モニターを再起動します。
- scds_pmf_signal(3HA) – 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(3HA) – 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(3HA) – 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(3HA) — 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(3HA) – 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(3HA) – 検証終了後にアクションを実行します。
- scds_fm_print_probes(3HA) – 検証状態情報をシステムログに書き込みます。

ユーティリティー関数

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(3HA) – メッセージをシステムログに書き込みます。
- scds_syslog_debug(3HA) – 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。

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

表 A–1 に、Sun Cluster によって定義されてい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、以下のように分類されます (分類の列)。

- 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 — 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、管理ユーティリィ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RGM 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/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、管理ユーティリィ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、管理ユーティリィ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	分類
`API_version` (整数)	このリソースタイプの実装によって使用されるリソース管理 API のバージョン。 SC 3.0 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/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

表 A–1 リソースタイププロパティ　続く

プロパティ名	説明	更新の可否	分類
FINI (文字列)	任意のコールバックメソッド。RGM 管理からこのタイプのリソースを削除するときに RGM が呼び出すプログラムへのパス。	不可	条件付/明示
INIT (文字列)	任意のコールバックメソッド。このタイプのリソースが RGM によって管理されるようになったときに、RGM が呼び出すプログラムへのパス。	不可	条件付/明示
Init_nodes (列挙)	値には、RG_primaries (リソースをマスターできるノードだけ)、または RT_installed_nodes (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 INIT、FINI、BOOT、VALIDATE 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、RG_primaries です。	不可	任意
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。このタイプのリソースの障害モニターが要求するフェイルオーバーを行う前に、RGM が呼び出すプログラム。	不可	条件付/明示
Monitor_start (文字列)	任意のコールバックメソッド。このタイプのリソースの障害モニターを起動するために、RGM が呼び出すプログラムへのパス。	不可	条件付/明示

表 A–1 リソースタイププロパティ　続く

プロパティ名	説明	更新の可否	分類
Monitor_stop (文字列)	Monitor_start が設定されている場合の、必須のコールバックメソッド。このタイプのリソースの障害モニターを停止するために、RGM が呼び出すプログラムへのパス。	不可	条件付/明示
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/明示
Postnet_stop (文字列)	任意のコールバックメソッド。このタイプのリソースが依存する任意のネットワークアドレスリソース (Network_resources_used) の STOP メソッドを呼び出した後で、RGM が呼び出すプログラムへのパス。ネットワークインタフェースが停止に構成された後に必要な STOP アクションを行う必要があります。	不可	条件付/明示
Prenet_start (文字列)	任意のコールバックメソッド。このタイプのリソースが依存する、任意のネットワークアドレスリソース (Network_resources_used) の START メソッドを呼び出す前に、RGM が呼び出すプログラムへのパス。ネットワークインタフェースが起動に構成された後に必要な START アクションを行う必要があります。	不可	条件付/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)

表 A–1 リソースタイププロパティ　続く

プロパティ名	説明	更新の可否	分類
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。	不可	条件付
Resource_type (文字列)	リソースタイプの名前。クラスタのインストールにおいて一意でなければなりません。このプロパティは、RTR ファイルの最初のエントリで宣言される必要があります。最初のエントリで宣言されていない場合は、リソースタイプの登録に失敗します。 さらに、リソースタイプを識別するために、Vendor_id を指定できます。Vendor_id とリソースタイプ名は、ピリオドで区切られます (例:SUNW.http)。リソースタイプは、Resource_type と Vendor_id で完全に指定することも、 Vendor_id を省略することもできます。たとえば、SUNW.http と http は、両方とも有効です。Vendor_id を指定する場合は、リソースタイプを定義する会社の株式銘柄を使用してください。クラスタ内で Vendor_id のみが異なるリソースタイプがある場合は、名前を省略できません。 デフォルトは空の文字列。	不可	必須
RT_version (文字列)	このリソースタイプを実装する任意のバージョン文字列。	不可	条件付/明示
Single_instance (ブール値)	True の場合は、このタイプのリソースがクラスタ内に 1 つだけ存在できることを指定します。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

表 A–1 リソースタイププロパティ　続く

プロパティ名	説明	更新の可否	分類
START (文字列)	コールバックメソッド。このタイプのリソースを開始するためにRGMが呼び出すプログラムへのパス。	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。このタイプのリソースを停止するためにRGM が呼び出すプログラムへのパス。	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。実行中のこのタイプのリソースのプロパティが変更された場合に、RGM が呼び出すプログラムへのパス。	不可	条件付/明示
VALIDATE (文字列)	任意のコールバックメソッド。このタイプのリソースのプロパティ値を検査するために呼び出すプログラムへのパス。	不可	条件付/明示
Vendor_ID (文字列)	Resource_type を参照してください。	不可	条件付

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、以下のように分類されます (分類の列)。

- 必須 — 管理者は、管理ユーティリィ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 — 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よって、必須または任意になります。詳細は、各条件付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 は、リソースプロパティが更新可能かどうか、また、いつ更新できるかも示しています。

None または False	更新不可
True または Anytime	任意の時点
At_creation	リソースをクラスタに追加するとき
When_disabled	リソースを無効にするとき

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	分類
Cheap_probe_interval (整数)	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、RGM 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で Tunable 属性が指定されていない場合は、プロパティの Tunable 値は、When_disabled (無効化にするとき) になります。 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。	無効にするとき	条件付
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

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
Failover_mode (列挙)	リソースでの START または STOP メソッドの呼び出しの失敗に対して、RGM がリソースグループを再配置するか、またはノードを異常終了させるかを制御します。None は、RGM が単にリソース状態をメソッド失敗に設定し、システム管理者の介入を待つことを示します。Soft は、START メソッドが失敗したときに、RGM がリソースのグループを別のノードに再配置し、また、STOP メソッドが失敗したときに、RGM がリソース状態を設定し、システム管理者の介入を待つことを示します。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 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。 デフォルトは、None です。	任意	任意

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
Load_balancing_policy (文字列)	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。 Load_balancing_policy は、次の値をとることができます。 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。 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。 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。 デフォルト値は、Lb_weighted です。	作成時	条件付/任意

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
`Load_balancing_weights` (文字配列)	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。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。 `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。 デフォルト値は、空の文字列 ("") です。	任意	条件付/任意
リソースタイプの各コールバックメソッドの *method*`_timeout` (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	条件付 任意

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
Monitored_ switch (列挙)	クラスタ管理者が管理ユーティリィ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	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。スケーラブルサービスの場合、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限り使用できません。 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	条件付/必須

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
On_off_switch (列挙)	クラスタ管理者が管理ユーティリィ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	照会のみ
Port_list (文字配列)	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限り使用できません。 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。	作成時	条件付 必須
R_description (string)	リソースの簡単な説明。 デフォルトは空の文字列。	任意	任意
Resource_dependencies (文字配列)	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。 デフォルトは、空のリストです。	任意	任意

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
Resource_ dependencies_ weak (文字配列)	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。 デフォルトは、空のリストです。	任意	任意
Resource_name (文字列)	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。	不可	必須
各クラスタノードの Resource_state (列挙)	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、Detached。 このプロパティは、ユーザーは構成できません。	不可	照会のみ

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
Retry_count (整数)	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。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。 リソースタイプファイル内で Tunable 属性が指定されていない場合は、プロパティの Tunable 値は、When_disabled (無効化にするとき) になります。 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。	無効化された時	条件付
Retry_interval (整数)	失敗したリソースを再起動する回数をカウントする間隔 (秒)。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。 リソースタイプファイル内で Tunable 属性が指定されていない場合は、プロパティの Tunable 値は、When_disabled (無効化にするとき) になります。 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。	無効化された時	条件付

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
Scalable (ブール値)	リソースがスケー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、次のスケーラブルサービスを自動的に作成します。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限り、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。 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。 このプロパティが RTR ファイルで宣言されていない場合、リソースはスケー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 はスケー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、スケーラブルサービスだけでなく、非スケーラブルサービスでも有用です。	作成時	任意
各クラスタノードの Status (列挙)(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、degraded、faulted、unknown、offline です。RGM は、リソースがオンラインになると、値を unknown に設定し、オフラインになると offline に設定します。	不可	照会のみ

表 A–2 リソースプロパティ 続く

プロパティ名	説明	更新の可否	分類
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	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。 リソースタイプファイル内で Tunable 属性が指定されていない場合は、プロパティの Tunable 値は、When_disabled (無効化にするとき) になります。 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。	無効化された時	条件付
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	必須

リソースグループプロパティ

表 A–3 に、Sun Cluster によって定義された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、以下のように分類されます (分類の列)。

- 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。

更新の可否の列は、初期設定後に、そのプロパティが更新可能 (Y) なのか、更新できない (N) のかを示しています。

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	分類
`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で任意のリソースに使用されるかどうかを示します。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　続く

プロパティ名	説明	更新の可否	分類
Implicit_ network_ dependencies (ブール値)	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。 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。 デフォルトは、True です。	可	任意
Maximum_ primaries (整数)	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。 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
Nodelist (文字配列)	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主ノードまたはマスターです。 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。	可	任意
Pathprefix (文字列)	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めます。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。 デフォルトは空の文字列。	可	任意

表 A–3 リソースグループプロパティ　続く

プロパティ名	説明	更新の可否	分類
Pingpong_ interval (整数)	再構成が生じた場合、scha_control giveover コマンドの実行結果、あるいは実行されている機能によって、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)。 再構成において、リソースの START または PRENET_START 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、Pingpong_interval 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、別のマスターを捜します リソースの scha_control(1ha)(3ha) コマンドまたは機能の呼び出しによって、Pingpong_interval 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 scha_control 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。 デフォルト値は、3,600 秒 (1 時間) です。	可	任意
Resource_ list (文字配列)	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。 デフォルトは、空のリストです。	不可	照会のみ
RG_ dependencies (文字配列)	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。 デフォルトは、空のリストです。	可	任意

表 A–3 リソースグループプロパティ　続く

プロパティ名	説明	更新の可否	分類
RG_description (文字列)	リソースグループの簡単な説明。 デフォルトは空の文字列。	可	任意
RG_mode (列挙)	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。 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。 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。	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
各クラスタノードの RG_state (列挙)(列挙)	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の制御下にない場合は、非管理状態で存在できます。 このプロパティは、ユーザーは構成できません。 デフォルトは、Offline です。	不可	照会のみ

リソースプロパティの属性

表 A–4 に、システム定義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の属性を示します。

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。
Tunable	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)。 デフォルトは、 True (Anytime) です。
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。

表 A–4　リソースプロパティの属性　続く

プロパティ	説明
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最小長。
Maxlength	string および stringarray タイプの場合、設定できる文字列の最大。
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。.

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
#


#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
```

(続く)

```
# (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;
}
{
        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
```

(続く)

```
# 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
#
# 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クラスタのノード上で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、START 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 $RESOURCE_NAME.named である。PMF は、
# 指定された回数 (Retry_count) だけ、サービスを起動しようとする。そ
# して、指定された期間 (Retry_interval) 内で試行回数がこの値を超えた
# 場合、PMF はサービスの起動に失敗したことを報告する。
# Retry_count と Retry_interval は両方とも RTR ファイルに設定されて
# いるリソースプロパティである。
#pragma ident ``@(#)dns_svc_start 1.1 00/05/24 SMI''
################################################################################
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
        typeset opt
        while getopts`R:G:T:' opt
        do
                case ``$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''
```

(続く)

```
                    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

# 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
```

(続く)

```
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 は再起動をやめる。
# <$PMF_TAG> というタグですでにプロセスが登録されて
# いる場合、PMF はすでにプロセスが動作していることを示す警告メッセ
# ージを送信する。
#
echo ``Retry interval is ``$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。
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

```
#
# 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=$OPTARG
                        ;;
                T)
                        # リソースタイプの名前
                        RESOURCETYPE_NAME=$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。
```

(続く)

```
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_NAMÈ

# 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。
 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 UNSUCCESFUL''

      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
```

(続く)

```
# 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 ユーティリティーのコードリスト

gettime ユーティリティーは、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 gettime.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
# サーバー自身を捜す。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 ...]
{
        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
```

(続く)

```
                        ;;

                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 下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 下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。

                # 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_NAMÈ
                STOP_METHOD≅scha_resource_get -O STOP
\
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                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_NAMÈ
                START_METHOD≅scha_resource_get -O START
\
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD
```

(続く)

```
\
                        -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 下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、同じクラスタ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

        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。
#
function decide_restart_or_failover
{

 # 最初の再起動の試行であるかどうかを検査する。
 if [ $retries -eq 0 ]; then
   # 最初の障害である。最初の試行の時刻を記録する。
   start_time≧$RT_BASEDIR/gettimè
   retries≧expr $retries + ì
   # 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
```

(続く)

```
  # 最初の障害ではない。
  current_time=$RT_BASEDIR/gettime
  time_diff=expr $current_time - $start_time
  if [ $time_diff -ge $RETRY_INTERVAL ]; then
   # この障害は再試行最大期間後に発生した。
   # したがって、再試行カウンタをリセットし、
   # 再試行時間をリセットし、さらに再試行する。
   retries=1
    start_time=$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
    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。
```

(続く)

```
parse_args ``$@''

PMF_TAG=$RESOURCE_NAME.named
SYSLOG_TAG=$RESOURCETYPE_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することによっ
# 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
-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
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 ユーティリティーの
# 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
probefail=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
```

(続く)

```
 retcode=$?
        if [ retcode -ne 0 ]; then
                probefail=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 [ $probefail -eq 0 ]; then
  # nslookup 照会に応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
                        probefail=1
                else
                        if [ $SERVER != $DNS_HOST ]; then
                                probefail=1
                        fi
                fi
        fi

 # 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 ``$opt'' in
                R)
                        # DNS リソースの名前
                        RESOURCE_NAME=$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
#
###############################################################################
```

(続く)

```
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=$RESOURCE_NAME.monitor
SYSLOG_TAG=$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 -1 -t -1 \
    $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。
started.
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
                        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
```

(続く)

```
# 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=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCETYPE_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

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
  ;;

  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
```

(続く)

```
# 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

# 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}'

# 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 $RESOURCETYPE_NAME -x Confdir=$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: 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
```

(続く)

```
                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)
# 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 \
```

(続く)

```
                    -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 機能番号を取得する。
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
# 用して取得する。
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 ''$@''

# 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]\
```

(続く)

```
            ''${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

RGM は、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。

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''

################################################################################
```

(続く)

```
# 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

}



################################################################################
# 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=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME
```

(続く)

```
# 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_NAMÈ

# 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 $RESOURCETYPE_NAME
 if [ $? -ne 0 ]; then
      logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]\
              ''${ARGV0} Could not restart monitor for HA-DNS ''
  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。

xfnts.c のコードリスト

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ー
 *
 * このユーティリティーは、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。
 *
 */

#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。 */
```

(続く)

```
#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。
 *
 * 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. 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トリを使用する必要がある。
  */
 (void) sprintf(xfnts_conf, ``%s/fontserver.cfg'',
confdirs->str_array[0]);

 /*
```

(続く)

```
  * 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) {
  /*
   * 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/xfs'', &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));
```

(続く)

```
  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。 */
 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/xfs -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> である。
 * 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。
 */
```

(続く)

```
int
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/xfs 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/xfs -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
  * プロセスプロセスと同様に監視することを示す。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 '');
```

(続く)

```
 }

 scds_free_port_list(portlist);
 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,
```

(続く)

```
     ''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
  * 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。    */
```

(続く)

```
  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,
 * 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
  * 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); /* Successfully started Monitor */
}


/*
 * この関数は、HA-XFS リソース用の障害モニターを停止する。
```

(続く)

```
 * 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>である。
 */

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) の範囲の整数値を戻す。
 *
 * 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;
 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、
```

(続く)

```
  * 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リスン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));
  /* これは致命的な障害である。 */
  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。  */

 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。
```

(続く)

```
  */
 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));
  /* これは部分的な障害である。 */
  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
  * 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。
  */
 (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>'',
```

(続く)

```
   testcmd);
  return (SCDS_PROBE_COMPLETE_FAILURE/2);
 }
 return (0);
}
```

xfnts_monitor_check メソッドのコードリスト

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 - Monitor Check method for HA-XFS
 */
#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[])
{
 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);
 }
```

(続く)

```
 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 メソッドのコードリスト

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 - Monitor Start method for HA-XFS
 */
#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、
 * interval/retry_time を RTR ファイルから取得する。
 */

int
main(int argc, char *argv[])
```

(続く)

```
{
 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 メソッドのコードリスト

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 - Monitor Stop method for HA-XFS
 */
#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。
```

(続く)

```
 */

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 メソッドのコードリスト

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 - Probe for HA-XFS
 */
```

(続く)

```
#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>
#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;

 hrtime_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.'');
```

(続く)

```
 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、
 * 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、休止状態になる。
  */
 (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. 指定されたリソースのすべての 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、
```

(続く)

```
     * 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();

    /* 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。 */
}
```

xfnts_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クラスタのノード上で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。xfnts_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** xfnts_start.c

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_svc_start.c - Start method for HA-XFS
 */
#pragma ident ''@(#)xfnts_svc_start.c 1.13 01/01/18 SMI''

#include <rgm/libdsdev.h>
#include ''xfnts.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
```

(続く)

```
 * 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;

 /*
  * 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 sevice 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。*/
```

(続く)

```
 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 デーモンを停止します。

例 C–7 xfnts_stop.c

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_svc_stop.c - Stop method for HA-XFS
 */
#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.'');
```

(続く)

```
  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

RGM は 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動作中のリソースに通知します。RGM が UPDATE メソッドを呼び出すのは、管理アクションによってリソースまたはリソースグループのプロパティの設定が成功したときです。

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 - Update method for HA-XFS
 */

#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
```

(続く)

```
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。
  */

 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.'');

 /* 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 (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 - validate method for HA-XFS
 */
#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。
 */

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_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);

}
```

付録 D

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。

付録 **E**

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、28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 3.0 12/01 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アプリケーション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もあります。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を使用する方法についての詳細は、264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。Sun Cluster を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もあります。アプリケーションのコードを変更せずに、シンボリックリンクを使用できる場合もあります。

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。

シンボリックリンクの使用には、問題があります。つまり、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) もあります。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。

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です。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に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き、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き、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、RGM 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を取り付けて (plumb) 取り外し (unplumb)、「起動」または「停止」に構成します。詳細は、44ページの「START と 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。