VisiTelcoLog ガイド

Borland VisiBroker® 7.0

Borland®
Excellence Endures™

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

2006 年 5 月 11 日初版発行

著者：Borland Software Corporation

発行：ボーランド株式会社

PDF

目次

第 1 章

Borland VisiBroker の概要

Borland は，CORBA 開発者に向けて，業界最先端の VisiBroker オブジェクトリクエストブローカー（ORB）を活用するために *VisiBroker for Java*，*VisiBroker for C++*，および *VisiBroker for .NET* を提供しています。この 3 つの VisiBroker は CORBA 2.6 仕様の実装です。

VisiBroker の概要

VisiBroker は，CORBA が Java オブジェクトと Java 以外のオブジェクトの間でやり取りする必要がある分散配布で使用されます。幅広いプラットフォーム（ハードウェア，オペレーティングシステム，コンパイラ，および JDK）で使用できます。VisiBroker は，異種環境の分散システムに関連して一般に発生するすべての問題を解決します。

VisiBroker は次のコンポーネントからなります。

- VisiBroker for Java，VisiBroker for C++，および VisiBroker for .NET（業界最先端のオブジェクトリクエストブローカーの 3 つの実装）。
- VisiNaming Service - Interoperable Naming Specification バージョン 1.3 の完全な実装。
- GateKeeper - ファイアウォールの背後の CORBA サーバーとの接続を管理するプロキシサーバー。
- VisiBroker Console - CORBA 環境を簡単に管理できる GUI ツール。
- コモンオブジェクトサービス - VisiNotify（通知サービス仕様の実装），VisiTransact（トランザクションサービス仕様の実装），VisiTelcoLog（Telecom ログサービス仕様の実装），VisiTime（タイムサービス仕様の実装），VisiSecure など。

VisiBroker の機能

VisiBroker には次の機能があります。

- セキュリティと Web 接続性を容易に装備できます。
- J2EE プラットフォームにシームレスに統合できます（CORBA クライアントが EJB に直接アクセスできる）。
- 堅牢なネーミングサービス（VisiNaming）とキャッシュ，永続的ストレージ，および複製によって高可用性を実現します。
- プライマリサーバーにアクセスできない場合に，クライアントをバックアップサーバーに自動的にフェイルオーバーします。
- CORBA サーバークラスタ内で負荷分散を行います。
- OMG CORBA 2.6 仕様に完全に準拠します。
- Borland JBuilder 統合開発環境と統合されます。
- Borland AppServer などの他の Borland 製品と最適に統合されます。

VisiBroker のマニュアル

VisiBroker のマニュアルセットは次のマニュアルで構成されています。

- *Borland VisiBroker インストールガイド* — VisiBroker をネットワークにインストールする方法について説明します。このマニュアルは，Windows または UNIX オペレーティングシステムに精通しているシステム管理者を対象としています。
- *Borland VisiBroker セキュリティガイド* — VisiSecure for VisiBroker for Java および VisiBroker for C++ など，VisiBroker のセキュリティを確保するための Borland のフレームワークについて説明しています。
- *Borland VisiBroker for Java 開発者ガイド* — Java による VisiBroker アプリケーションの開発方法について記載されています。Visibroker ORB の設定と管理，およびプログラミングツールの使用方法について説明します。また, IDL コンパイラ, スマートエージェント，ロケーションサービス，ネーミングサービス，イベントサービス，オブジェクトアクティベーションデーモン (OAD), Quality of Service (QoS), インターフェースリポジトリ，および Web サービスサポートについても説明します。
- *Borland VisiBroker for C++ 開発者ガイド* — C++ による VisiBroker アプリケーションの開発方法について記載されています。Visibroker ORB の設定と管理，およびプログラミングツールの使用方法について説明します。また, IDL コンパイラ, スマートエージェント，ロケーションサービス，ネーミングサービス，イベントサービス，OAD, QoS，プラグイン可能トランスポートインターフェース，RT CORBA 拡張機能，Web サービスサポート，およびインターフェースリポジトリについても説明します。
- *Borland VisiBroker for .NET 開発者ガイド* — .NET 環境による VisiBroker アプリケーションの開発方法について記載されています。
- *Borland VisiBroker for C++ API リファレンス* — VisiBroker for C++ に付属するクラスとインターフェースについて説明します。
- *Borland VisiBroker VisiTime ガイド* — Borland による OMG Time Service 仕様の実装について説明します。
- *Borland VisiBroker VisiNotify ガイド* — Borland による OMG 通知サービス仕様の実装について説明します。通知メッセージフレームワークの主な機能として, 特に Quality of Service (QoS) のプロパティ, フィルタリング, および Publish/Subscribe Adapter (PSA) の使用方法が記載されています。

- *Borland VisiBroker VisiTransact ガイド* — Borland による OMG Object Transaction Service 仕様の実装および Borland Integrated Transaction Service コンポーネントについて説明します。
- *Borland VisiBroker VisiTelcoLog ガイド* — Borland による OMG Telecom Log Service 仕様の実装について説明します。
- *Borland VisiBroker GateKeeper ガイド* — Web ブラウザやファイアウォールによるセキュリティ制約の下で，VisiBroker GateKeeper を使用して，VisiBroker のクライアントがネットワークを介してサーバーとの通信を確立する方法について説明します。

通常，マニュアルにアクセスするには，VisiBroker とともにインストールされるヘルプビューアを使用します。ヘルプは，スタンドアロンのヘルプビューアからアクセスすることも，VisiBroker コンソールからアクセスすることもできます。どちらの場合も，ヘルプビューアを起動すると独立したウィンドウが表示されるため，このウィンドウからヘルプビューアのメインツールバーにアクセスしてナビゲーションや印刷を行ったり，ナビゲーションペインにアクセスすることができます。ヘルプビューアのナビゲーションペインには，すべての VisiBroker ブックとリファレンス文書の目次，完全なインデックス，および包括的な検索を実行できるページがあります。

重要 Web サイト http://www.borland.com/techpubs には，PDF 版のマニュアルと最新の製品マニュアルがあります。

スタンドアロンヘルプビューアからの VisiBroker オンラインヘルプトピックへのアクセス

製品がインストールされているコンピュータでスタンドアロンのヘルプビューアからオンラインヘルプにアクセスするには，次のいずれかの手順を実行します。

Windows

- ［スタート｜プログラム｜ Borland VisiBroker ｜ Help Topics］の順に選択します。
- または，コマンドプロンプトを開き，製品のインストールディレクトリの ¥bin ディレクトリに移動し，次のコマンドを入力します。

```
help
```

UNIX コマンドシェルを開き，製品のインストールディレクトリの /bin ディレクトリに移動し，次のコマンドを入力します。

```
help
```

ヒント UNIX システムにインストールするときの指定で，PATH エントリのデフォルトに bin を含まないようにします。カスタムインストールオプションを選択して PATH エントリのデフォルトを変更せず，PATH に現在のディレクトリのエントリがない場合は，./help を使用してヘルプビューアを起動できます。

VisiBroker コンソールからの VisiBroker オンラインヘルプトピックへのアクセス

VisiBroker コンソールから VisiBroker オンラインヘルプトピックにアクセスするには，［Help ｜ Help Topics］を選択します。

［Help］メニューには，オンラインヘルプ内のいくつかの文書へのショートカットもあります。ショートカットの 1 つを選択すると，ヘルプトピックビューアが起動し，［Help］メニューで選択した項目が表示されます。

マニュアルの表記規則

VisiBroker のマニュアルでは，文中の特定の部分を表すために，次の表に示す書体と記号を使用します。

表 1.1　マニュアルの表記規則

表記規則	用途
italic	新規の用語およびマニュアル名に使用されます。
`computer`	ユーザーやアプリケーションが提供する情報，サンプルコマンドライン，およびコードです。
bold computer	本文では，ユーザーが入力する情報を示します。サンプルコードでは，重要なステートメントを強調表示します。
[ ]	省略可能な項目。
...	繰り返しが可能な直前の引数。
|	二者択一の選択。

プラットフォームの表記

VisiBroker マニュアルでは，次の記号を使用してプラットフォーム固有の情報を示します。

表 1.2　プラットフォームの表記

記号	意味
Windows	サポートされているすべての Windows プラットフォーム
Win2003	Windows 2003 のみ
WinXP	Windows XP のみ
Win2000	Windows 2000 のみ
UNIX	すべての UNIX プラットフォーム
Solaris	Solaris のみ
Linux	Linux のみ

Borland サポートへの連絡

ボーランド社は各種のサポートオプションを用意しています。それらにはインターネット上の無償サービスが含まれており，大規模な情報ベースを検索したり，他の Borland 製品ユーザーからの情報を得ることができます。さらに Borland 製品のインストールに関するサポートから有償のコンサルタントレベルのサポートおよび高レベルなアシスタンスに至るまでの複数のカテゴリから，電話サポートの種類を選択できます。

Borland のサポートサービスの詳細や Borland テクニカルサポートへの問い合わせについては，Web サイト `http://support.borland.com` で地域を選択してください。

ボーランド社のサポートへの連絡にあたっては，次の情報を用意してください。

- 名前
- 会社名およびサイト ID
- 電話番号
- ユーザー ID 番号（米国のみ）
- オペレーティングシステムおよびバージョン
- Borland 製品名およびバージョン
- 適用済みのパッチまたはサービスパック
- クライアントの言語とそのバージョン（使用している場合）
- データベースとそのバージョン（使用している場合）

- 発生した問題の詳細な内容と経緯
- 問題を示すログファイル
- 発生したエラーメッセージまたは例外の詳細な内容

オンラインリソース

ネットワーク上の次のサイトから情報を得ることができます。

Web サイト http://www.borland.com/jp/

オンラインサポート http://support.borland.com（ユーザー ID が必要）

リストサーバー 電子ニュースレター（英文）を購読する場合は，次のサイトに用意されているオンライン書式を使用してください。

http://www.borland.com/products/newsletters

Web サイト

定期的に http://www.borland.com/jp/products/visibroker/index.html をチェックしてください。VisiBroker 製品チームによるホワイトペーパー，競合製品の分析，FAQ の回答，サンプルアプリケーション，最新ソフトウェア，最新のマニュアル，および新旧製品に関する情報が掲載されます。

特に，次の URL をチェックすることをお勧めします。

- http://www.borland.com/products/downloads/download_visibroker.html（最新の VisiBroker ソフトウェアおよび他のファイル）
- http://www.borland.com/techpubs（マニュアルの更新および PDF）
- http://info.borland.com/devsupport/bdp/faq/（VisiBroker の FAQ）
- http://community.borland.com（英語，開発者向けの弊社 Web ベースニュースマガジン）

Borland ニュースグループ

Borland VisiBroker を対象とした数多くのニュースグループに参加できます。VisiBroker などの Borland 製品のユーザーによるニュースグループへの参加については，http://www.borland.com/newsgroups を参照してください。

メモ これらのニュースグループはユーザーによって管理されているものであり，ボーランド社の公式サイトではありません。

第 2 章

VisiTelcoLog サービスの概要

VisiTelcoLog サービスは，Borland による OMG Telecom Log Service 仕様バージョン 1.1.2 の OMG 準拠インプリメンテーションです。このサービスは，ログインターフェースのすべてのオペレーション，ログインターフェースのファクトリ，詳細なセマンティクスなど，OMG 仕様で定義されているすべての機能をサポートしています。このマニュアルは，VisiTelcoLog サービスのユーザーガイドとして，OMG Telecom Log Service 仕様に精通したユーザーを対象としています。

VisiTelcoLog サービスの主な目的は，イベントチャネルや通知サービスを介して渡されるイベントを透過的にログ処理することです。通常，このサービスは，通信管理ネットワーク（TMN）などのミッションクリティカルな分散監視制御アプリケーションで使用されます。このようなアプリケーションには，わずかなオーバーヘッドでイベントを転送できる高パフォーマンスのイベント／通知サービスが必要なだけでなく，これらのイベントの一部または全部を効率よく透過的にログ処理する機能が必要です。仕様は「OMG Telecom Log Service」と呼ばれ，Borland のインプリメンテーションは「VisiTelcoLog サービス」と呼ばれていますが，アーキテクチャ自体はたいへん汎用的で，あらゆるアプリケーションで使用できます。

VisiTelcoLog サービスは，イベントログ処理の詳細からアプリケーションを分離する高レベルのイベントログモデルを提供します。これにより，サードパーティがより高いパフォーマンスを備えたアプリケーション汎用のログサービスを実装できます。アプリケーションでは，VisiTelcoLog サービスを使用しなくても，イベントコンシューマを実装および接続して，受信したすべてのイベントを通常のデータベースなどの外部永続的リポジトリにログとして記録できます。ただし，このようにアプリケーションレベルでイベントログ処理を独自に構築する方法には，アプリケーション開発者がイベントのアンマーシャリングを完全に実装したり，アプリケーション固有のレコードスキーマやイベント／レコード間の変換コードを開発する必要があるという欠点があります。その結果，パフォーマンス（イベントスループット）は低下し，開発とメンテナンスのコストは上昇してしまいます。

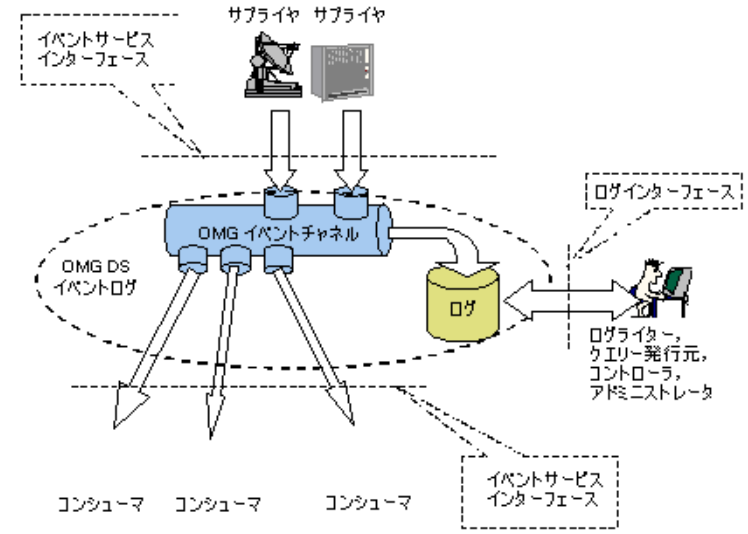
VisiTelcoLog サービスでは，イベントチャネルまたは通知チャネルで受信されたイベントがアプリケーションレベルで透過的にログ処理されます。また，イベントログオブジェクト（このマニュアルでは，*DsEventLog* オブジェクトまたはイベントベースログオブジェクト）は，通常の OMG イベントチャネルであり，OMG イベントチャネルから拡張されます。これにより，イベントをログ処理する方法に関係なく，アプリケーションを設計および開発できます。既存のイベントベースアプリケーションでも，アプリケーションコードを変更したり再配布することなく，VisiTelcoLog サービスのイベントログ処理を利用できます。

イベント／通知ベースアプリケーションでの透過性に加えて，DsEventLog は，ログオブジェクトからも拡張されます。このログオブジェクトでは，ログレコードのクエリー，更新，削除，ログオブジェクトの制御／管理操作のほか，明示的な非イベントレコードのログ処理も実行できます。DsEventLog オブジェクトは，通常のイベントチャネルとログオブジェクトから拡張されます。

イベントチャネル，型付きイベントチャネル，通知チャネル，型付き通知チャネルなど，OMG 定義の各イベントチャネルに対応するログオブジェクトがあります。イベント対応ではないアプリケーションには，BasicLog オブジェクトも提供されています。

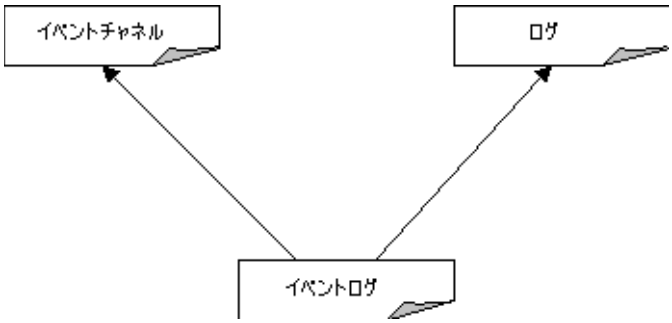
次の図は，VisiTelcoLog サービスの EventLog のアーキテクチャとインターフェース継承を示しています。最初の図は，イベントサプライヤがイベントをログ処理すると同時に，イベントをすべてのコンシューマに転送する方法を示しています。ログインターフェースを使用して，ログ処理されたイベントを照会することもできます。

図 2.1　EventLog アーキテクチャ



2 つめの図は，イベントベースのログオブジェクトの階層を示しています。

図 2.2　EventLog インターフェース継承



第 3 章

イベント対応アプリケーションのログ

この章では，イベント／通知サービスベースのアプリケーション（一般には*イベント対応*アプリケーション）で，VisiTelcoLog サービスを使ってイベントをログ処理する方法について説明します。VisiTelcoLog サービスは，基本的にイベントロガーです。この場合の*ログ*とは，イベントを永続的ストレージに記録することだけでなく，イベントを伝達するイベントチャネルを指します。

イベント対応アプリケーションが使用できるイベントベースのログオブジェクトには，次の 4 つがあります。

- EventLog
- NotifyLog
- TypedEventLog
- TypedNotifyLog

次の表で，イベント／通知サービスベースのアプリケーションで使用できる VisiTelcoLog サービスのモジュール，インターフェース，および機能について説明します。

機能	OMG イベントサービスアプリケーション	OMG 通知サービスアプリケーション
モジュール名	DsEventLogAdmin	DsNotifyLogAdmin
ファクトリインターフェース名	EventLogFactory	NotifyLogFactory
ログインターフェース名	EventLog	NotifyLog
ファクトリサービス名	EventLogService	NotifyLogService
型付きイベントモジュール名	DsTypedEventLogAdmin	DsTypedNotifyLogAdmin
型付きイベントファクトリインターフェース名	TypedEventLogFactory	TypedNotifyLogFactory
型付きイベントログインターフェース名	TypedEventLog	TypedNotifyLog
型付きイベントファクトリサービス名	TypedEventLogService	TypedNotifyLogService
ログの転送	はい	はい
ログ転送時のフィルタリング	いいえ	はい
保存時のフィルタリング	いいえ	はい

この章では，次の項目について説明します。

- イベントベースのログオブジェクトを取得するために「ログファクトリの使用」（10 ページ）
- イベントベースのログオブジェクトで「イベントのログ」（10 ページ）
- コンシューマに「ログ処理したイベントの転送」（13 ページ）
- ログ処理する「イベントのフィルタリング」（13 ページ）

ログファクトリの使用

イベント対応アプリケーションでイベントをログ処理する場合は，ログファクトリを使用して，イベントベースのログが最初にブートストラップされます。たとえば，通知サービスベースのアプリケーションは，最初にオブジェクト名 NotifyLogService を使って NotifyLogFactory を取得し，次に NotifyLog タイプのログを取得します。ほかのタイプのイベントベースアプリケーションについては，上記の表を参照してください。ここでは，イベントベースのログオブジェクトへのリファレンスを取得する方法について説明します。

次のコードは，最初にオブジェクト名 NotifyLogService を使用して，NotifyLogFactory にブートストラップします。次に，このファクトリから，ID が 100 の NotifyLog ログを検索します。NotifyLog が見つからない場合は作成します。最大サイズは 0 を指定します。これは，制限の定義を使用しないという意味です。ただし，制限を定義しておくことをお勧めします。

メモ　このサンプルコードは　<install_dir>/examples/vbroker/telcolog/primitive_cpp ディレクトリにあります。

C++

```
// サービスリファレンスを取得します。
CORBA::Object_var service =
  orb->resolve_initial_references("NotifyLogService");

DsNotifyLogAdmin::NotifyLogFactory_var factory =
  DsNotifyLogAdmin::NotifyLogFactory::_narrow(service);

// ID が 100 のログを検索します。
DsLogAdmin::LogId id = 100;
DsLogAdmin::Log_var log = factory->find_log(id);

// 作成されていない場合はログを作成します。
if( log.in() ==  NULL )
{
  CORBA::ULongLong max_size = 4 * 1024 * 1024;
  DsLogAdmin::CapacityAlarmThresholdList thresholds;
  CosNotification::QoSProperties initial_qos;
  CosNotification::AdminProperties initial_admin;

  log = factory->create_with_id(id, DsLogAdmin::wrap,
    max_size, thresholds, initial_qos, initial_admin);
}

DsNotifyLogAdmin::NotifyLog_var notify_log=
  DsNotifyLogAdmin::NotifyLog::_narrow(log.in());
```

メモ　このサンプルコードは　<install_dir>/examples/vbroker/telcolog/primitive_java ディレクトリにあります。

Java

```
// サービスリファレンスを取得します。
org.omg.CORBA.Object service =
  orb.resolve_initial_references("NotifyLogService");
```

```
org.omg.DsNotifyLogAdmin.NotifyLogFactory factory =
  org.omg.DsNotifyLogAdmin.NotifyLogFactoryHelper.narrow(
  service);

// ID が 100 のログを検索します。
int id = 100;
org.omg.DsLogAdmin.Log log = factory.find_log(id);

// 作成されていない場合はログを作成します。
if( log == null )
{
  long max_size = 4 * 1024 * 1024;
  log = factory.create_with_id(id,
    org.omg.DsLogAdmin.wrap.value, max_size, new short[0],
    new org.omg.CosNotification.Property[0],
    new org.omg.CosNotification.Property[0]);
}

org.omg.DsNotifyLogAdmin.NotifyLog notify_log =
  org.omg.DsNotifyLogAdmin.NotifyLogHelper.narrow(log);
```

イベントのログ

イベントベースのログオブジェクトへのリファレンスが解決されると，プッシュやプルなどのイベント伝達（または転送）操作を使用して，イベントが伝達されます。このチャネルオブジェクトもログの性質を持つため，チャネルオブジェクトを介して伝達されるすべてのイベントがログ処理されます。ログにフィルタを設定することもできます。イベントを選択してログ処理する方法については，13 ページの「イベントのフィルタリング」を参照してください。

さらに，通知ベースのアプリケーションでは，QoS フレームワーク，イベントフィルタなどのすべての通知サービス機能を使用できます。

通知サービスサプライヤアプリケーションの開発については,『VisiBroker *Visinotify ガイド*』の「サプライヤ／コンシューマアプリケーションの開発」を参照してください。

VisiTelcoLog サービスは，GIOP レベルでのイベントログ処理を最適化します。

*ログフル*状態では，ログフルアクションが *wrap* に設定されている場合は，最も古いイベントが上書きされます。ログフルアクションが *halt* に設定され，ログレコードの存続期間が指定されている場合は，期限切れのイベントが上書きされます。そうでない場合は，次の例外が生成されます。

- **容量不足:**イベントをログ処理するだけのログ容量が不足している場合は,マイナーコード LOGFULL（1001）の NO_RESOURCE システム例外が生成されます。
- **ログが動作していない：**ログが動作していない場合は，マイナーコード LOGOFFDUTY（1000）の NO_RESOURCE システム例が生成されます。
- **ログがロックされている：**ログがロックされている場合は，マイナーコード LOGLOCKED（1003）の NO_PERMISSION システム例が生成されます。
- **ログが無効:**ログが無効な場合は，マイナーコードLOGDISABLED（1002）のTRANSIENT システム例が生成されます。

サプライヤがイベントバッチを使用している場合，サプライヤに例外は送信されません。イベントバッチの詳細については,『VisiBroker *Visinotify ガイド*』の「VisiBroker イベントバッファリング／バッチ」を参照してください。

また，プルサプライヤでは，チャネルはイベントをプルしてからログ処理します。ログフル状態では，ログ容量に空きができるまで，チャネルは継続してログ処理を試みます。サプライヤアプリケーションでこの状態を認識する方法はありません。

vbroker.dslog.waitForLogAvailable プロパティを使用すると，このループの待機時間を指定できます。デフォルトでは，20 秒です。

次のサンプルコードは，構造化サプライヤが TMN QoS アラームイベントをログ処理しています。このサプライヤアプリケーションは，（ログもチャネルなので）最初にログからデフォルトのサプライヤ管理を取得し，構造化プロキシプッシュコンシューマを取得してから，それに接続します。次に TMN QoS アラームイベントを作成し，ログを介してイベントをプッシュします。ログにイベントがプッシュされると，ログはそのイベントを保存し，ログの転送状態に基づいてイベントを転送します。

C++

メモ　このサンプルコードは　<install_dir>/examples/vbroker/telcolog/primitive_cpp ディレクトリにあります。

```
// ログからデフォルトのサプライヤ管理オブジェクトを取得します。
CosNotifyChannelAdmin::SupplierAdmin_var admin =
  notify_log->default_supplier_admin();

CosNotifyChannelAdmin::ProxyID  proxy_id;

// ログでプロキシコンシューマを作成します。
CosNotifyChannelAdmin::ProxyConsumer_var proxy =
  admin->obtain_notification_push_consumer(
  CosNotifyChannelAdmin::STRUCTURED_EVENT, proxy_id);

CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
  Consumer = CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(proxy);

// プロキシコンシューマに接続します。
consumer->connect_structured_push_supplier(NULL);

// 構造化イベントに TMN QoS アラームイベントを設定します。
TMN::Event event;
CosNotification::StructuredEvent structured;
TMN::QoSAlarmInfo qosalrm_info;
misc::forge_qosAlrmInfo(qosalrm_info);
event.name = (const char*)
  " TMN::Events::qosAlarm";
event.info <<= qosalrm_info;
misc::gathering(event, structured);

// 構造化イベントをログにプッシュします。
consumer->push_structured_event(structured);
```

Java

メモ　このサンプルコードは　<install_dir>/examples/vbroker/telcolog/primitive_java ディレクトリにあります。

```
// ログからデフォルトのサプライヤ管理オブジェクトを取得します。
org.omg.CosNotifyChannelAdmin.SupplierAdmin admin
  = notify_log.default_supplier_admin();

org.omg.CORBA.IntHolder proxy_id =
  new org.omg.CORBA.IntHolder();

// ログでプロキシコンシューマを作成します。
org.omg.CosNotifyChannelAdmin.ProxyConsumer proxy =
  admin.obtain_notification_push_consumer(
    org.omg.CosNotifyChannelAdmin.ClientType.STRUCTURED_EVENT,
    proxy_id);

org.omg.CosNotifyChannelAdmin.StructuredProxyPushConsumer
  consumer =
```

```
    org.omg.CosNotifyChannelAdmin.StructuredProxyPushConsumerHelper.narrow(
      proxy);

// プロキシコンシューマに接続します。
consumer.connect_structured_push_supplier(null);

// 構造化イベントに TMN QoS アラームイベントを設定します。
TMN.Event event = new TMN.Event();
org.omg.CosNotification.StructuredEvent structured =
  new org.omg.CosNotification.StructuredEvent();
TMN.QoSAlarmInfo qosalrm_info = new TMN.QoSAlarmInfo();
event.header = new TMN.EventHeader();
event.info = orb.create_any();
Util.forge_event_header(event.header);
Util.forge_qosAlrmInfo(qosalrm_info);
event.name = "TMN::Events::qosAlarm";
TMN.QoSAlarmInfoHelper.insert(event.info,qosalrm_info);
Util.gathering(event, structured);

// 構造化イベントをログにプッシュします。
consumer.push_structured_event(structured);
```

ログ処理したイベントの転送

ログにプッシュまたはログからプルされたイベントは，ログ処理された後でそれぞれの下流のコンシューマに転送されます。コンシューマアプリケーションは，伝達されたイベントの消費を開始します。コンシューマアプリケーションの記述方法については，『VisiBroker *Visinotify ガイド*』の「サプライヤ／コンシューマアプリケーションの開発」を参照してください。

転送状態を *off* に設定すると，ログオブジェクトがログ処理されたイベントを転送しないように設定できます。次のコードは，アプリケーションで NotifyLog オブジェクトの転送を無効にし，ログの現在の転送状態をチェックする方法を示しています。

フィルタの適用や QoS など，イベントサービスと通知サービスのすべての機能をイベントの伝達で使用できます。

C++

```
notify_log->set_forwarding_state(DsLogAdmin::off);

DsLogAdmin::ForwardingState current_state =
  notify_log->get_forwarding_state();
```

Java

```
notify_log.set_forwarding_state(
  org.omg.DsLogAdmin.ForwardingState.off);

org.omg.DsLogAdmin::ForwardingState current_state =
  notify_log.get_forwarding_state();
```

イベントのフィルタリング

NotifyLog や TypedNotifyLog に設定されたフィルタは，ログにログ処理されるイベントもフィルタリングできます。ログは，通知サービスで定義されたフィルタオブジェクト CosNotifyFilter::Filter を使用します。フィルタの作成と制約の記述の詳細については，『VisiBroker *Visinotify ガイド*』の「Quality of Service とフィルタの設定」を参照してください。

ログに関連付けることができるフィルタオブジェクトは 1 つだけです。デフォルトでは，ログに関連付けられたフィルタオブジェクトはなく，すべてのイベントがログ処理されます。また，set_filter() メソッドが呼び出されると，ログは AttributeValueChange イベントを生成します。

次のサンプルは，フィルタの作成，ログへのフィルタの設定，およびログからのフィルタの取得を行う方法を示しています。

C++

```
// フィルタを使用します。
// ステップ 1) デフォルトのフィルタファクトリを取得します。
CosNotifyFilter::FilterFactory_var ffact =
  log->default_filter_factory();

// ステップ 2) フィルタを作成します。
CosNotifyFilter::Filter_var filter1;
filter1 = ffact->create_filter("EXTENDED_TCL");

// ステップ 3) 制約を作成します。
CosNotifyFilter::ConstraintExpSeq constr_seq1;
constr_seq1.length(1);
constr_seq1[0].constraint_expr = CORBA::string_dup(
  "$type_name == 'TMN::Events::qosAlarm'"
);

// ステップ 4) フィルタに制約を設定します。
filter1->add_constraints( constr_seq1 );

// ステップ 5) ログにフィルタを設定します。
log->set_filter( filter1 );

// ステップ 6) ログに関連付けられているフィルタを取得します。
CosNotifyFilter::Filter_var filter2;
Filter2 = log->get_filter();
```

Java

```
// フィルタを使用します。
//[1] フィルタファクトリを取得します。
org.omg.CosNotifyFilter.FilterFactory ffact =
  channel.default_filter_factory();

//[2] フィルタを作成します。
org.omg.CosNotifyFilter.Filter filter = null;
filter = ffact.create_filter("EXTENDED_TCL");

//[3] 制約を作成します。
org.omg.CosNotifyFilter.ConstraintExp [] constraints =
  new org.omg.CosNotifyFilter.ConstraintExp[1];
constraints [0] =
  new org.omg.CosNotifyFilter.ConstraintExp();
constraints [0].constraint_expr =
  new String ("$type_name == 'TMN::Events::qosAlarm'");

//[4] フィルタに制約を設定します。
org.omg.CosNotifyFilter.ConstraintInfo[] info = null;
info = filter.add_constraints(constraints);

//[5] ログにフィルタを設定します。
log.set_filter (filter);

//[6] ログに関連付けられているフィルタを取得します。
org.omg.CosNotifyFilter.Filter filter2 = null;
filter2 = log.get_filter();
```

第 4 章

イベント非対応アプリケーションのログ

レガシーアプリケーションや*イベント非対応*クライアントでも，VisiTelcoLog サービスを使用できます。BasicLog インターフェースと，CORBA Any による明示的な書き込み操作を使用することで，イベント非対応アプリケーションで VisiTelcoLog サービスを使用できます。ただし，このようなアプリケーションでは，ログのフィルタリング，転送，イベント生成などの機能は使用できません。

次の表で，イベント非対応アプリケーションで使用できる VisiTelcoLog サービスのモジュール，インターフェース，およびログ機能について説明します。

機能	イベント非対応アプリケーション
モジュール名	DsLogAdmin
ファクトリインターフェース名	BasicLogFactory
ログインターフェース名	BasicLog
ファクトリサービス名	BasicLogService
ログの転送	いいえ
ログ転送時のフィルタリング	いいえ
保存時のフィルタリング	いいえ

この章では，次の項目について説明します。

- イベント非対応アプリケーションで，ログオブジェクトを取得するために「ログファクトリの使用」（15 ページ）
- イベント非対応アプリケーションで，「ログレコードの書き込み」（17 ページ）

ログファクトリの使用

イベント非対応アプリケーションでログを実行するには，ファクトリ BasicLogFactory から BasicLog へのリファレンスを取得する必要があります。基本的なログオブジェクトの作成に加えて，このファクトリインターフェースは，検索やリスト作成などの基本管理操作もいくつかサポートしています。

BasicLogService 名を解決すると，BasicLogFactory オブジェクトリファレンスを取得できます。次のコードでは，ID が 100 の BasicLog を検索し，見つからなかった場合は，サイズが 0 の BasicLog を作成しています。サイズ 0 は，サイズの制限が定義されていないという意味です。ログサイズを 0 に設定すると，ディスクスペースをすべて使用してしまうまでログが拡張されます。そのため，適切な値を設定することをお勧めします。

C++

```
// サービスリファレンスを取得します。
CORBA::Object_var service =
  orb->resolve_initial_references("BasicLogService");

DsLogAdmin::BasicLogFactory_var factory =
  DsLogAdmin::BasicLogFactory::_narrow(service);

// ID が 100 のログを検索します。
DsLogAdmin::LogId id = 100;
DsLogAdmin::Log_var log = factory->find_log(id);

// 作成されていない場合はログを作成します。
if( log.in() ==  NULL )
{
  CORBA::ULongLong max_size = 4 * 1024 * 1024;
  // max_size=0 では，最大ログサイズが無制限になります。

  log = factory->create_with_id(id, DsLogAdmin::wrap,
    max_size);
}

DsLogAdmin::BasicLog_var basic_log=
  DsLogAdmin::BasicLog::_narrow(log.in());
```

Java

```
// サービスリファレンスを取得します。
org.omg.CORBA.Object service =
  orb.resolve_initial_references("BasicLogService");

org.omg.DsLogAdmin.BasicLogFactory factory =
  org.omg.DsLogAdmin.BasicLogFactoryHelper.narrow(
  service);

// ID が 100 のログを検索します。
int id = 100;
org.omg.DsLogAdmin.Log log = factory.find_log(id);

// 作成されていない場合はログを作成します。
if( log == null )
{
  long max_size = 4 * 1024 * 1024;
  // max_size=0 では，最大ログサイズが無制限になります。

  log = factory.create_with_id(id,
    org.omg.DsLogAdmin.wrap.value, max_size);
}

org.omg.DsLogAdmin.BasicLog basic_log =
  org.omg.DsLogAdmin.BasicLogHelper.narrow(log);
```

ログレコードの書き込み

write_records オペレーションは，ログにレコードを書き込むために使用されます。このオペレーションの入力パラメータは，CORBA Any のシーケンスです。このシーケンス内の各 Any は，各ログレコードを表します。

書き込み中にログがいっぱいになると，LogFull ユーザー例外が生成されます。この例外には，元の Anys シーケンスから書き込まれたレコード数も含まれます。

ログの状態が off_duty の場合は，LogOffDuty ユーザー例外が生成されます。ログの状態が locked の場合は，LogLocked ユーザー例外が生成されます。ログが disabled の場合は，LogDisabled 例外が生成されます。

次のサンプルコードは，write_records オペレーションを使用して，いくつかの TMN イベントを書き込む手順を示しています。

C++

```
// TMN イベント
TMN::Event event;
TMN::AttrValChgSeq attrvalchg_info;
TMN::AttrValSeq objcrt_info;
TMN::AttrValSeq objdel_info;
TMN::QoSAlarmInfo qosalrm_info;

// TMN イベントにデータを設定します。
misc::forge_event_header(event.header);
misc::forge_attrValChgInfo(attrvalchg_info);
misc::forge_objCrtInfo(objcrt_info);
misc::forge_objDelInfo(objdel_info);
misc::forge_qosAlrmInfo(qosalrm_info);

// 書き込まれる Any のシーケンス
DsLogAdmin::Anys  anys;
anys.length(4);

// TMN イベントを Any シーケンスに挿入します。
event.name = (const char*)
  "TMN::Events::attributeValueChange";
event.info <<= attrvalchg_info;
anys[0] <<= event;

event.name = (const char*)
  "TMN::Events::objectCreation";
event.info <<= objcrt_info;
anys[1] <<= event;

event.name = (const char*)
  "TMN::Events::objectDeletion";
event.info <<= objdel_info;
anys[2] <<= event;

event.name = (const char*)
  "TMN::Events::qosAlarm";
event.info <<= qosalrm_info;
anys[3] <<= event;

// Any のシーケンスをログに書き込みます。
basic_log->write_records(anys);
```

Java

```
// TMN イベント
TMN.Event event = new TMN.Event();
```

```
TMN.AttrValChgSeqHolder  attrvalchg_info =
  new TMN.AttrValChgSeqHolder();
TMN.AttrValSeqHolder objcrt_info =
  new TMN.AttrValSeqHolder();
TMN.AttrValSeqHolder objdel_info =
  new TMN.AttrValSeqHolder();
TMN.QoSAlarmInfo qosalrm_info =
  new TMN.QoSAlarmInfo();

// TMN イベントにデータを設定します。
event.header = new TMN.EventHeader();
event.info = orb.create_any();
Util.forge_event_header(event.header);
Util.forge_attrValChgInfo(attrvalchg_info);
Util.forge_objCrtInfo(objcrt_info);
Util.forge_objDelInfo(objdel_info);
Util.forge_qosAlrmInfo(qosalrm_info);

// 書き込まれる Any のシーケンス
org.omg.CORBA.Any[] anys =
  new org.omg.CORBA.Any[4];
for (int i = 0; i < 4; i++)
{
  anys[i] = orb.create_any();
}

// TMN イベントを Any シーケンスに挿入します。
event.name = "TMN::Events::attributeValueChange";
TMN.AttrValChgSeqHelper.insert(event.info,
  attrvalchg_info.value);
TMN.EventHelper.insert(anys[0],event);

event.name = "TMN::Events::objectCreation";
TMN.AttrValSeqHelper.insert(event.info,objcrt_info.value);
TMN.EventHelper.insert(anys[1],event);

event.name = "TMN::Events::objectDeletion";
           TMN.AttrValSeqHelper.insert(event.info,objdel_info.value);
TMN.EventHelper.insert(anys[2],event);

event.name = "TMN::Events::qosAlarm";
           TMN.QoSAlarmInfoHelper.insert(event.info,qosalrm_info);
TMN.EventHelper.insert(anys[3],event);

// Any のシーケンスをログに書き込みます。
basic_log.write_records(anys);
```

第 5 章

ログインターフェースの概要

ログの特性は，イベントベースのログオブジェクトや基本ログオブジェクトと同じです。これらの特性は，DsLogAdmin::Log インターフェースに格納されています。すべてのログオブジェクトはこのインターフェースを継承するため，共通の特性を保有します。

この章では，次の項目について説明します。

ログレコードと型付きログレコード

イベント対応または非対応アプリケーションが VisiTelcoLog サービスを使用し，push，pull，または write_record オペレーションを使ってログにレコードを書き込む場合は，受信したイベントまたは Any シーケンス内の CORBA Any ごとに LogRecord が作成されます。同様に，受信した型付きイベントごとに作成されるログレコードは TypedLogRecord です。

LogRecord 構造体と TypedLogRecord 構造体を次の IDL に示します。

```
struct LogRecord
{
  RecordId id;
  TimeT time;
  NVList attr_list;
  any info;
};

struct TypedLogRecord
{
  RecordId id;
```

```
    TimeT time;
    NVList attr_list;
    RepositoryId interface_id;
    Identifier operation_name;
    ArgumentList arg_list;
  };
```

構造体定義の詳細については，OMG Telecom Log Service 仕様を参照してください。

上記の IDL で示された構造体で，RecordId id は，ログによってレコードに割り当てられた一意の番号です。これは，そのログでのみ一意です。

TimeT time は，レコードが基底のバックエンドに書き込まれたときのタイムスタンプです。

NVList attr_list には，各ログレコードのユーザー定義属性のリストを格納できます。この属性は，書き込みの時点ではなく，別の set_attribute() API を使ってログレコードに関連付けられます。属性の設定の詳細については，21 ページの「ログ属性の設定」を参照してください。

ログデータ自体は，CORBA Any に格納されます。型付きイベントの場合，ログデータは，型付きイベントオペレーションの引数リストにカプセル化されます。

RepositoryId interface_id はインターフェースのリポジトリ ID，Identifier operation_name は型付きイベントを発行したオペレーションの名前です。

QoS のログ

OMG Telecom Log Service 仕様に基づいて，VisiTelcoLog サービスは，set_log_qos() API と get_log_qos() API を備えた軽量の QoS フレームワークを提供します。これは，通知サービス仕様の拡張 QoS フレームワークに追加されます。

VisiTelcoLog サービスは，次の QoS プロパティをサポートしています。

QoS プロパティ	説明
QoSNone	これが指定された場合，QoS は約束されません。flush() オペレーションを呼び出しても，ログレコードはフラッシュされません。
QoSFlush	これが指定された場合，flush() を呼び出すと，すべてのログレコードがバックエンドにフラッシュ／コミットされます。
QoSReliability	これが指定された場合，ログレコードは直接バックエンドに書き込まれます。

VisiTelcoLog サービスは，set_log_qos() オペレーションで指定されている最高値の QoS だけを使用します。たとえば，3 つの QoS プロパティがすべて指定されている場合は，QoSReliability だけが使用されます。これは，get_log_qos() オペレーションに反映されます。次のコードは，この動作を示しています。

C++

```
DsLogAdmin::QosList qos;
qos.length(3);
qos[0] = DsLogAdmin::QoSNone;
qos[1] = DsLogAdmin::QoSFlush;
qos[2] = DsLogAdmin::QoSReliability;

// 3 つの QoS をすべて設定します。
basic_log->set_log_qos(qos);

// QoSReliability のみ
qos = basic_log->get_log_qos();
```

ログサイズと操作

ここでは，ログサイズの制御，ログフルアクションの決定，およびログレコード存続期間の制御の方法について説明します。

ログサイズの制御

ログの最大サイズ（バイト）は，ログの作成時に指定できます。ログファクトリのログ作成オペレーションは，どれもログサイズパラメータを受け取ります（10 ページの「ログファクトリの使用」のサンプルコードを参照）。ログサイズは，ログが最大限拡張できるサイズです。サイズ 0 は，制限が定義されておらず，ログが無制限に拡張できるという意味です。サイズを設定した後でも，`set_max_size()` オペレーションと `get_max_size()` オペレーションを使ってサイズを変更できます。ログの最大サイズは，現在のサイズとは異なります。現在のサイズは，ログレコードが使用しているバイト数です。

ログの現在のサイズより小さな値を使って `set_max_size()` を呼び出すと，`InvalidParam` ユーザー例外が生成されます。また，1 MB 未満の値を使って `set_max_size()` を呼び出すと，`InvalidParam` が生成されます。最大サイズの値として設定できる最小値は 1 MB です。これはインプリメンテーションの制限です。1 MB より小さな初期最大サイズでログを作成しようとすると，自動的に 1 MB に設定されます。

ログフルアクション

ログの現在のサイズが最大サイズに達した場合，ログは*ログフル*状態であると呼ばれます。VisiTelcoLog サービスでは，ログフル状態にあるログに対して実行する必要があるログフルアクションを指定します。デフォルトのログフルアクションは，ログの作成時に指定されます。

`set_log_full_action()` を呼び出すと，ログフル状態で実行するアクションとして，ログの `wrap` または `halt` を指定できます。ログフルアクションが `wrap` の場合は，新しいログレコードを書き込むために十分な容量ができるまで，古いログレコードが削除されます。

ログフルアクションが `halt` の場合は，ログの最大レコード存続期間が指定されていれば，期限切れのログレコードがすべて削除されます。期限切れのレコードが削除されると，書き込み操作がやり直されます。書き込みに再度失敗すると，対応する例外が生成されます。生成される例外と書き込み操作の詳細については，9 ページの「イベント対応アプリケーションのログ」と 15 ページの「イベント非対応アプリケーションのログ」を参照してください。

ログレコード存続期間

ログレコード存続期間は，`set_max_record_life()` API を使って秒単位で指定できます。最大レコード存続期間を値 0 に指定すると，ログレコードの存続期間は無制限になります。

ログレコード存続期間を指定した場合は，ガベージコレクタスレッドが定期的に期限切れのログレコードを削除します。デフォルトでは，ガベージコレクタスレッドが 60 分ごとに起動されます。このスレッドの時間間隔は，`vbroker.dslog.backend.garbageCollectorInterval` プロパティを使って設定できます。

ログ属性の設定

OMG Telecom Log Service 仕様に基づいて，VisiTelcoLog サービスでは，クライアントアプリケーションがログレコードの属性リストを定義できます。属性リストは，アプリケーションにとって意味のある一連の名前／値ペアで構成されます。これらのログレコード属性は（ログレコード構造体で示される），読み書きすることができます。

ログレコード ID または文法／制約を使用して，ログレコードの属性を設定したり取得することができます。set_record_attribute() API を使用すると，ログレコード ID に基づいてログレコードに属性を設定できます。同様に，set_records_attribute() API を使用すると，文法／制約パラメータで指定された制約式に一致する複数のログレコードに属性を設定できます。

VisiTelcoLog サービスは，ログの*書き込み*に対して最適化されています。そのため，これらは比較的負荷が大きい操作です。属性を設定する際は，ログ全体がコピーされて置き換えられます。

ログのコピー

OMG Telecom Log Service 仕様に基づいて，VisiTelcoLog サービスには，既存のログオブジェクトをコピーするためのコピーオペレーションが 2 つ用意されています。copy() オペレーションは，元のログと同じ特性の空のログを作成します。新しいコピーログオブジェクトのログ ID が出力パラメータに戻されます。

copy_with_id() オペレーションは，ログ ID を受け取り，この入力ログ ID を使って元のログと同じ特性の空のログを作成します。この入力ログ ID を持つログがすでに存在する場合は，LogIdAlreadyExists ユーザー例外が生成されます。どちらのオペレーションも，リソースの制約のためにログファクトリが新しいログを作成できない場合は，NO_RESOURCES システム例外を生成します。

ログレコードのクエリー，取得，反復子

OMG Telecom Log Service 仕様に基づいて，VisiTelcoLog サービスでは，ログレコードを照会するメソッドが 2 つ用意されています。

- retrieve メソッドは，時刻に基づいてレコードを取得します。
- query メソッドは，制約に基づいてレコードを取得します。

型付きログレコードの場合，対応するメソッドは次のとおりです。

- typed_retrieve メソッドは，時刻に基づいてレコードを取得します。
- typed_query メソッドは，制約に基づいてレコードを取得します。

retrieve メソッドと query メソッドは，大量の取得レコードを処理するための反復子を出力パラメータとして戻します。query オペレーションと retrieve オペレーションは，その特性上逐次実行されるため，ログレコード数が非常に多い場合は時間がかかります。

時刻に基づくレコードの取得

ログインターフェースには，時刻に基づいてクエリーを実行する retrieve() メソッドと typed_retrieve() メソッドが用意されています。また，指定された時刻から前方または後方に順番に取得するレコードの数も指定できます。大量の取得レコードを処理するための反復子が提供されます。次のコードは，時刻に基づいてレコードを取得する方法を示しています。

C++

```
DsLogAdmin::TimeT from_time;
DsLogAdmin::RecordList_var time_recs;
DsLogAdmin::Iterator_var time_itr;
...
// from_time から後方に 10 レコード（-10）を取得します。
// 取得するレコードの数が 1000 を超える場合は，
// 残りのレコードを反復子 time_itr に保存します。
time_recs =
```

```
  log->retrieve( from_time, -10, time_itr.out() );
...
```

Java

```
org.omg.DsLogAdmin.TimeT from_time;
org.omg.DsLogAdmin.RecordList time_recs = null;
org.omg.DsLogAdmin.Iterator time_itr = null;
...
// from_time から後方に 10 レコード（-10）を取得します。
// 取得するレコードの数が 1000 を超える場合は，
// 残りのレコードを反復子 time_itr に保存します。
time_recs =
  log.retrieve( from_time, -10, time_itr );
...
```

制約に基づくレコードのクエリー

ログインターフェースには，特定の制約に基づいてクエリーを実行する query() メソッドと typed_query() メソッドが用意されています。制約は，VisiBroker の VisiNotify フィルタ制約に基づいています。Extended Trader Constraint Language（TCL）を使って制約を記述する方法については，『VisiBroker VisiNotify ガイド』の「フィルタ制約式の記述」を参照してください。query 呼び出しは，使用する文法と制約式を受け取り，大量のレコードを処理するために反復子が提供されます。

LogRecord 構造体または TypedLogRecord 構造体を照会する制約を記述する場合は，それぞれの定義について，19 ページの「ログレコードと型付きログレコード」を参照してください。

次の例に，制約を使用する照会の方法を示します。VisiTelcoLog サービスは，制約の文法としてデフォルトの EXTENDED_TCL だけを認識します。

C++

```
DsLogAdmin::RecordList_var recs_found;
DsLogAdmin::Iterator_var itr;
...
// 「EXTENDED_TCL」文法を使って照会し，
// ID が 100 未満「$.id
```

Java

```
omg.org.DsLogAdmin.RecordList recs_found = null;
omg.org.DsLogAdmin.Iterator itr = null;
...
// 「EXTENDED_TCL」文法を使って照会し，
// ID が 100 未満「$.id
```

反復子

大量のログレコードが戻される場合は，retrieve() メソッドまたは query() メソッドから反復子が戻されます。retrieve() メソッドまたは query() メソッドが何個のレコードを戻す場合に反復子を使用するかは，vbroker.dslog.getRecMaxList プロパティで制御されます。query() オペレーションまたは retrieve() オペレーションに一致したレコード数が，vbroker.dslog.getRecMaxList で指定された値より大きい場合は，過剰分のログレコードが反復子に追加されます。typed_retrieve() または typed_query() が呼び出された場合は，TypedRecordIterator が戻されます。

ログ反復子は，get() と destroy() の 2 つのメソッドを提供します。get() メソッドを使用すると，反復子によって保存されたレコードを取得できます。get() メソッドを呼び出す場合は，開始位置と，その位置から取得するレコードの数を指定する必要があります。反復

子内の位置は前方にしか移動しないため，最後の要求の位置より前の値を要求することはできません。無効な値を要求すると，InvalidParam 例外が生成されます。

次のコードは，反復子の get() メソッドを使用する方法を示しています。

C++

```
DsLogAdmin::RecordList_var recs_found;
DsLogAdmin::Iterator_var itr;
...
// 「EXTENDED_TCL」文法を使って照会し，
// ID が 100 未満「$.id
```

Java

```
omg.org.DsLogAdmin.RecordList recs_found = null;
omg.org.DsLogAdmin.Iterator itr = null;
...
// 「EXTENDED_TCL」文法を使って照会し，
// ID が 100 未満「$.id
```

反復子の末尾に達し，反復子内の最後のレコードの位置を使って get() を呼び出すと，get() メソッドは空のログレコードリストを戻します。これは，反復子の末尾に達したことを示します。アプリケーションでは，VisiTelcoLog サービスから取得したオブジェクトを破棄するために，destroy() メソッドを呼び出す必要があります。

ログレコードの削除

ログインターフェースでは，文法／制約式または ID を使用して，ログレコードや型付きログレコードを削除できます。そのために，delete_records() と delete_records_by_id() の 2 つの API が用意されています。この API について次の表で説明します。

メソッド	説明
delete_records()	文法と制約式に基づいてログレコードを削除します。
delete_records_by_id()	ログレコードの ID 番号に基づいてログレコードを削除します。

VisiTelcoLog サービスは，イベントログレコードや型付きイベントログレコードの削除を最適化するために，これらをただちに削除するのではなく，削除のマークを付けます。時間の経過とともに，この最適化によってログがフラグメント化する可能性があります。そのため，フラグメンテーションが一定の限度を超えると，フラグメンテーション解消スレッドで自動的に削除操作が開始されます。フラグメンテーションの限度は，デフォルトでは 75% ですが，プロパティを使って設定できます。詳細については，プロパティのセクションを参照してください。フラグメンテーション解消ロジックは基本的にコピー操作です。つまり，すべてのログレコードが再度書き込まれます。フラグメンテーション解消は負荷が大きい操作であることに注意してください。

次のコードは，文法と制約式を使用して，ID が 200 のログレコードを削除する方法を示しています。delete_records_by_id() を使用して，同じ処理を実行することもできます。

C++

```
// ID = 200 のログレコードの制約
const char* grammar = "EXTENDED_TCL";
const char* constraint = "$.id == 200";

// 制約に一致するログレコードを削除します。
basic_log->delete_records(grammar, constraint);
```

Java

```
// ID = 200 のログレコードを削除します。
basic_log.delete_records("EXTENDED_TCL", "$.id == 200");
```

</html

第 6 章

高度な機能

この節では，次の高度な機能について説明します。

ログ継続時間

ログ継続時間を設定すると，ロック解除された有効なログオブジェクトが機能する期間として，粗粒度の時間間隔（ウィンドウ）を作成できます。ログ継続時間が設定された場合，ログオブジェクトは，ログレコードまたはログイベントを指定の時間内にだけログに書き込むことができます。

ログ継続時間は，次のメソッドを使用して，設定および取得されます。

```
set_interval(in DsLogAdmin::TimeInterval interval);
```

および

```
DsLogAdmin::TimeInterval get_interval();
```

入力パラメータと戻り値は，次のように定義された IDL 構造体です。

```
module DsLogAdmin {
 typedef TimeBase::TimeT TimeT;
 struct TimeInterval {
  TimeT start;
  TimeT stop;
 };
};
```

時間間隔の `start` フィールドと `stop` フィールドは，`CORBA::ULongLong` 型です。これらの値は，グリニッジ標準時（GMT）1582 年 10 月 15 日 00:00:00 からの経過時間を 10^{-7} 秒（または 100 ナノ秒）単位で表します。

`start` 時間と `stop` 時間の単位は，OMG によって 10^{-7} 秒と指定されていますが，VisiTelcoLog によってサポートされる実際の時間精度は秒単位です。`set_interval()` で指定された `start` 値と `stop` 値は，VisiTelcoLog サービスによって四捨五入で秒単位の値に丸められます。

start 値と stop 値がどちらも 0 に設定されるか，0 に丸められた場合，ログは常に動作状態になります。

現在のログ継続時間設定を取得するには，ログの get_interval() オペレーションを呼び出します。

ログのスケジュール

ログのスケジュールを使用すると，特定のログオブジェクトに細粒度の週間時間間隔（週マスク）を設定できます。スケジュールが設定された場合，ログオブジェクトは，ログレコードまたはログイベントをそれらの時間間隔内にだけログに書き込むことができます。ただし，その時間がログ継続時間内にあり（上記の 27 ページの「ログ継続時間」を参照），ログがロック解除された有効な状態である必要があります。

ログスケジュールの時間間隔は，次のメソッドを使用して，設定および取得されます。

```
set_week_mask(in DsLogAdmin::WeekMask weekmask);
```

および

```
DsLogAdmin::WeekMask get_week_mask();
```

これらのメソッドの入力パラメータと戻り値は，IDL 構造体 WeekMaskItem の IDL シーケンスです。次のように定義されます。

```
module DsLogAdmin {
 struct Time24 {
  unsigned short hour; // 0 - 23
  unsigned short minute; // 0 - 59
 };

 struct Time24Interval {
  Time24 start;
  Time24 stop;
 };

 typedef sequence<Time24Interval> IntervalsOfDay;
 typedef unsigned short DaysOfWeek;

 struct WeekMaskItem {
  DaysOfWeek days;
  IntervalsOfDay intervals;
 };

 typedef sequence<WeekMaskItem> WeekMask;
};
```

デフォルトではグリニッジ標準時（GMT）が使用されます。次のプロパティ設定で VisiTelcoLog サービスを開始することで，ログサーバーのローカルタイムゾーンを使用できます。

```
vbroker.dslog.scheduleByServerLocalTime=true
```

次のプロパティ設定で VisiTelcoLog サービスを開始すると，診断目的で，ログスケジュール設定の変更とログの動作をコンソールの stdout で監視できます。

```
vbroker.dslog.timerDebug=true
```

VisiTelcoLog サービスは，ログスケジュールの例とともに次のディレクトリに配置されています。

```
<install_dir>/examples/vbroker/telcolog/primitive_cpp/scheduler.C
```

次の C++ コードは，set_week_mask() を使用する方法を示しています。

```
// 7:30 am から 12:00 am
DsLogAdmin::Time24Interval morning = {{7,30},{12,0}};

// 13:30 (1:30 pm) から 17:30 (5:30 pm)
DsLogAdmin::Time24Interval afternoon = {{13,30},{17,30}};

// 21:00 (9:00 pm) から 23:30 (11:30 pm)
DsLogAdmin::Time24Interval night = {{21,0},{23,30}};

// 19:30 (7:30 pm) から 22:30 (11:30 pm)
DsLogAdmin::Time24Interval evening = {{19,30},{22,30}};

// 9:00 am から 16:30 (4:30 pm)
DsLogAdmin::Time24Interval wkend_day = {{9,0},{16,30}};

DsLogAdmin::WeekMask new_weekmask;
new_weekmask.length(2);

// 0 番めの週マスク項目内の平日スケジュール
new_weekmask[0].days = (DsLogAdmin::Monday
                        | DsLogAdmin::Tuesday
                        | DsLogAdmin::Wednesday
                        | DsLogAdmin::Thursday
                        | DsLogAdmin::Friday );

new_weekmask[0].intervals.length(3); // 3 個の時間間隔
new_weekmask[0].intervals[0] = morning;
new_weekmask[0].intervals[1] = afternoon;
new_weekmask[0].intervals[2] = night;

// 最初の週マスク項目内の週末スケジュール
new_weekmask[1].days = (DsLogAdmin::Sunday
                        | DsLogAdmin::Saturday );

new_weekmask[1].intervals.length(2); // 2 個の時間間隔
new_weekmask[1].intervals[0] = wkend_day;
new_weekmask[1].intervals[1] = evening;

// ログに新しい週マスクを設定します。
log->set_week_mask(new_weekmask);
```

次の C++ コードは，get_week_mask() を使用し，結果を処理する方法を示しています。

```
// ログから現在の週マスクを取得します。
DsLogAdmin::WeekMask_var holder;
holder = log->get_week_mask();

const char* day_names[7] = {
 "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
};

const DsLogAdmin::WeekMask& mask = holder.in();
CORBA::Short day, daybit;
CORBA::ULong i, j;

// 取得したスケジュールを曜日ごとに出力します。
for(day=0,daybit=1;day<7;daybit = daybit*2, day++) {
    cout << "    " << day_names[day] << ": ";
    for(i=0;i<mask.length();i++) {
        const DsLogAdmin::WeekMaskItem& item = mask[i];

        if( (daybit & item.days) == 0 ) {
            continue;
        }
```

```
            for(j=0;j<item.intervals.length();j++) {
                const DsLogAdmin::Time24Interval& interval =
                                                item.intervals[j];
                char buf[32];

                sprintf(buf, "[%02u:%02u-%02u:%02u] ",
                    interval.start.hour,
                    interval.start.minute,
                    interval.stop.hour,
                    interval.stop.minute);

        cout << buf;
            }
        }

        cout << endl;
        }
    }
```

set_week_mask() 要求の処理で，ログオブジェクトサーバーは，入力された週マスクパラメータを検証します。set_week_mask() で発生する例外と，それに対応する週マスク設定のエラーについて，次の表で説明します。

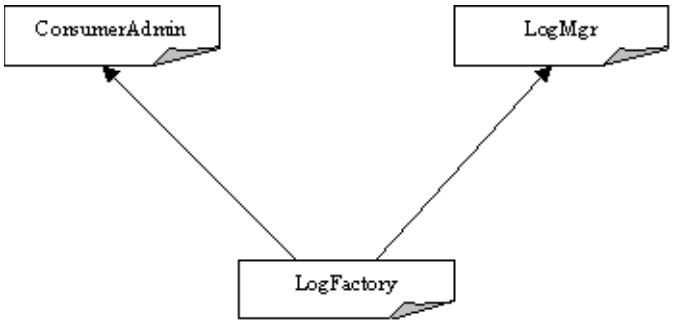
例外	説明
DsLogAdmin::InvalidTime	間隔の start フィールドまたは stop フィールドの hour フィールドまたは minute フィールドが範囲外です。hour の有効な範囲は 0 ～ 23，minute の有効な範囲は 0 ～ 59 です。
DsLogAdmin::InvalidTimeInterval	*ケース 1 :* start 時刻が stop 時刻より後になっている場合。そのため，午前 0 時より前に開始し，午前 0 時より後に終了する間隔はサポートされていません。2 日にまたがる間隔を設定するには，午前 0 時の直前（23:59）に終了する間隔と，次の日の午前 0 時（00:00）に開始する間隔の 2 つを使用する必要があります。 *ケース 2 :* 時間間隔どうしが重なっている場合。スケジュールされた間隔の start 時刻または stop 時刻が，同じ週マスクパラメータ内でスケジュールされた別の間隔の範囲内にあります。

エラーによって set_week_mask() が失敗した場合，ログの既存の週マスクはそのまま残り，DsLogNotification::ProcessingErrorAlarm ログイベント（30 ページの「ログ生成イベント」を参照）が送信されます。set_week_mask() が成功すると，既存の週マスクは新しい週マスクに完全に置き換えられます。したがって，既存の週マスクをすべて削除するには，空の週マスク（長さが 0 の週マスク）を使って set_week_mask() を呼び出します。空の週マスクが設定されたログは，週全体のログの記録を許可します。

ログ生成イベント

OMG Telecom Log Service 仕様に基づいて，イベント対応のログファクトリとログは，ログオブジェクトの作成と削除，状態や属性の変更，しきい値の超過，および処理エラーに関するイベントを生成できます。VisiTelcoLog サービスの付加価値拡張機能を使用すると，BasicLog オブジェクトでこれらのイベントを生成できます。これらのログ生成イベントは，*ログイベント*と呼ばれます。そのため，VisiTelcoLog サービスのログファクトリ（Basic，Event，TypedEvent，Notify，または TypedNotify ファクトリ）は CosNotifyChannelAdmin::ConsumerAdmin です。

図 6.1　ログファクトリの継承インターフェース



LogFactory が ConsumerAdmin である目的は，各ログファクトリ内でイベントチャネルの下流側（コンシューマ側）の機能を公開することです。このイベントチャネルは，*ログイベントチャネル*と呼ばれます。ログファクトリやそのログから生成されたログイベントはすべて，このファクトリのログイベントチャネルに送信されます。アプリケーションでログイベントを受信するには，ConsumerAdmin から継承したオペレーションを介してログファクトリのコンシューマ側プロキシを作成し，そのプロキシに接続します。

次の C++ コード（<install_dir>/examples/vbroker/telcolog/primitive_cpp/logEventReceiver.C ディレクトリ内）は，イベントコンシューマを NotifyLogFactory のログイベントチャネルに接続する方法を示しています。

```
int main(int argc, char** argv)
{
 CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

    // サービスリファレンス（通知ログファクトリ）を取得します。
 CORBA::Object_var service
  = orb->resolve_initial_references(
                                  "NotifyLogService");

 // ファクトリをコンシューマ管理者に直接ナローイングします。
 CosNotifyChannelAdmin::ConsumerAdmin_var admin
        = CosNotifyChannelAdmin::ConsumerAdmin
                                   ::_narrow(service);

 CosNotifyChannelAdmin::ProxyID proxy_id;

 // プロキシを作成します。
 CosNotifyChannelAdmin::ProxySupplier_var proxy
  = admin->obtain_notification_push_supplier(
   CosNotifyChannelAdmin::ANY_EVENT, proxy_id);

 CosNotifyChannelAdmin::ProxyPushSupplier_var supplier;
 supplier =  CosNotifyChannelAdmin::ProxyPushSupplier
                                         ::_narrow(proxy);

 // コンシューマインプリメンテーションを割り当てます。
 PushConsumerImpl* servant = new PushConsumerImpl;

 // これをルート POA でアクティブ化します。
 CORBA::Object_var obj
        = orb->resolve_initial_references("RootPOA");
 PortableServer::POA_var poa
        = PortableServer::POA::_narrow(obj);
 poa->activate_object(servant);

 // ルート POA をアクティブ化します。
 PortableServer::POAManager_var poa_manager
        = poa->the_POAManager();
 poa_manager->activate();
```

```
// コンシューマオブジェクトリファレンスを取得します。
CORBA::Object_var ref
       = poa->servant_to_reference(servant);
CosNotifyComm::PushConsumer_var consumer =
  CosNotifyComm::PushConsumer::_narrow(ref);

// コンシューマをサプライヤプロキシに接続します。
supplier->connect_any_push_consumer(consumer);

cout << "log event receiver is ready" << endl;

// 作業ループ
orb->run();
  }
  catch(CORBA::Exception& e) {
  cout << "caught exception:" << endl << e << endl;
  }

  return 0;
}
```

次の Java コードは，イベントコンシューマを NotifyLogFactory のログイベントチャネルに接続する方法を示しています。

```
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.PortableServer.*;
import org.omg.CosNotifyComm.*;

public class logEventReceiver {

    public static void main(String[] args) {
      try {
        org.omg.CORBA.ORB orb
            = org.omg.CORBA.ORB.init(args, null);

        // サービスリファレンス（通知ログファクトリ）を取得します。
        org.omg.CORBA.Object service
            = orb.resolve_initial_references(
                              "NotifyLogService");

        // ファクトリをコンシューマ管理者に直接ナローイングします。
        ConsumerAdmin admin
         = ConsumerAdminHelper.narrow(service);

        org.omg.CORBA.IntHolder proxy_id
            = new org.omg.CORBA.IntHolder();

        // プロキシを作成します。
        ProxySupplier proxy
            = admin.obtain_notification_push_supplier(
                        ClientType.ANY_EVENT, proxy_id);

        ProxyPushSupplier supplier
            = ProxyPushSupplierHelper.narrow(proxy);

        // コンシューマインプリメンテーションを割り当てます。
        PushConsumerImpl servant = new PushConsumerImpl();

        // これをルート POA でアクティブ化します。
        org.omg.CORBA.Object obj
            = orb.resolve_initial_references("RootPOA");
        POA poa = POAHelper.narrow(obj);
        poa.activate_object(servant);

        // ルート POA をアクティブ化します。
        POAManager poa_manager = poa.the_POAManager();
```

```
            poa_manager.activate();

            // コンシューマオブジェクトリファレンスを取得します。
            org.omg.CORBA.Object ref
                = poa.servant_to_reference(servant);
            PushConsumer consumer
                = PushConsumerHelper.narrow(ref);

            // コンシューマをサプライヤプロキシに接続します。
            supplier.connect_any_push_consumer(consumer);

            System.out.println("untyped push consumer is ready");

            // 作業ループ
            orb.run();
          } catch(Exception e) {
              e.printStackTrace();
          }
        }
    }
```

OMG によって指定されている有効なログイベントとその意味を以下のセクションで説明します。

オブジェクト作成イベント

このイベントは，ログオブジェクトの作成が正しく実行されたときにログファクトリ自体から発行されます。新しいログ ID とログ作成時間が CORBA Any イベント本体にカプセル化されます。これは，次のように定義された IDL 構造体です。

```
module DsNotification {
struct ObjectCreation
   {
        LogId   id;
        TimeT   time;
   };
};
```

オブジェクト削除イベント

このイベントは，ログオブジェクトの削除が正しく実行されたときにログファクトリ自体から発行されます。削除されたログ ID とログ削除時間が CORBA Any イベント本体にカプセル化されます。これは，次のように定義された IDL 構造体です。

```
module DsNotification {
struct ObjectDeletion
   {
        LogId   id;
        TimeT   time;
   };
};
```

属性値変更（AVC）イベント

このイベントは，ログ属性値の変更が正しく実行されたときにログから発行されます。属性値の変更に関する情報が CORBA Any イベント本体にカプセル化されます。これは，次のように定義された IDL 構造体です。

```
module DsNotification {
struct AttributeValueChange
    {
        Log           logref;
        LogId         id;
        TimeT         time;
        AttributeType type;
        Any           old_value;
        Any           new_value;
    };
};
```

この構造体の内容は次のとおりです。

- logref は，ログオブジェクト自体のリファレンスです。
- id は，ログオブジェクトのログ ID です。
- time は，属性値の変更が行われた時間です。
- type は，変更された属性のタイプを示します。次の説明を参照してください。
- old_value は，変更前の属性の元の値をカプセル化します。
- new_value は，変更後の属性の新しい値をカプセル化します。

OMG では，ログオブジェクトの属性タイプを次のように指定しています。

属性タイプ	説明
capacityAlarmThreshold （タイプ = 0）	このタイプの AVC イベントは，ログオブジェクトの set_capacity_thresholds() 呼び出しが正しく実行されるとトリガーされ，これまでの容量アラームしきい値設定を変更します。
logFullAction （タイプ = 1）	このタイプの AVC イベントは，ログオブジェクトの set_full_action() 呼び出しが正しく実行されるとトリガーされ，これまでのログフルアクション設定を変更します。
maxLogSize （タイプ = 2）	このタイプの AVC イベントは，ログオブジェクトの set_max_size() 呼び出しが正しく実行されるとトリガーされ，これまでの最大ログサイズ設定を変更します。
startTime （タイプ = 3）	このタイプの AVC イベントは，ログオブジェクトの set_interval() 呼び出しが正しく実行されるとトリガーされ，ログ期間開始時刻設定を変更します。
stopTime （タイプ = 4）	このタイプの AVC イベントは，ログオブジェクトの set_interval() 呼び出しが正しく実行されるとトリガーされ，ログ期間終了時刻設定を変更します。
weekMask （タイプ = 5）	このタイプの AVC イベントは，ログオブジェクトの set_week_mask() 呼び出しが正しく実行されるとトリガーされます。
filter （タイプ = 6）	このタイプの AVC イベントは，ログオブジェクトの set_filter() 呼び出しが正しく実行されるとトリガーされ，フィルタを変更します。
maxRecordLife （タイプ = 7）	このタイプの AVC イベントは，ログオブジェクトの set_max_record_life() 呼び出しが正しく実行されるとトリガーされ，最大レコード存続期間設定を変更します。
qualityOfService （タイプ = 8）	このタイプの AVC イベントは，ログオブジェクトの set_log_qos() 呼び出しが正しく実行されるとトリガーされ，ログ QoS 設定を変更します。

状態変化イベント

このイベントは，OMG 指定のログ状態の変化があるとログから発行されます。状態の変化に関する情報が CORBA Any イベント本体にカプセル化されます。これは，次のように定義された IDL 構造体です。

```
module DsNotification {
struct StateChange
   {
        Log        logref;
        LogId      id;
        TimeT      time;
        StateType  type;
        Any        new_value;
   };
};
```

この構造体の内容は次のとおりです。

- logref は，ログオブジェクト自体のリファレンスです。
- id は，ログオブジェクトのログ ID です。
- time は，状態変化の時刻です。
- type は，変化した状態のタイプを示します。次の説明を参照してください。
- new_value は，変化後の新しい状態値をカプセル化します。

OMG では，ログオブジェクトの状態変化イベントタイプを次のように指定しています。

状態変化イベントタイプ	説明
administrativeState（タイプ = 0）	このタイプの状態変化イベントは，ログオブジェクトの set_administrative_state() 呼び出しが正しく実行されるとトリガーされ，管理状態を変更して，ログレコード書き込み操作（挿入，更新，削除など）を許可または禁止します。
operationalState（タイプ = 1）	このタイプは，今回のリリースの VisiTelcoLog サービスインプリメンテーションでは使用されません。
forwardingState（タイプ = 2）	このタイプは，ログオブジェクトの set_forwarding_state() 呼び出しが正しく実行されるとトリガーされ，転送状態を変更して，イベントの転送を有効または無効にします。

しきい値アラームイベント

このイベントは，ログの書き込み操作によってログがサイズしきい値を超えた場合にログオブジェクトから発行されます。属性値の変更に関する情報が CORBA Any イベント本体にカプセル化されます。これは，次のように定義された IDL 構造体です。

```
module DsNotification {
struct ThresholdAlarm
    {
        Log                  logref;
        LogId                id;
        TimeT                time;
        Threshold            crossed_value;
        Threshold            observed_value;
        PerceivedSeverityType perceived_severity;
    };
};
```

この構造体の内容は次のとおりです。

- logref は，ログオブジェクト自体のリファレンスです。
- id は，ログオブジェクトのログ ID です。

- time は，アラームの発生時刻です。
- crossed_value は，このイベントで超えたしきい値です。
- observed_value は，現在のログ容量使用割合です。
- perceived_severity は重要度を示し，0 は高，1 は中，2 は低です。

処理エラーアラームイベント

このイベントは，ログファクトリまたはログオブジェクト内で問題が発生すると，そのログファクトリまたはログオブジェクトから送信されます。属性値の変更に関する情報が CORBA Any イベント本体にカプセル化されます。これは，次のように定義された IDL 構造体です。

```
module DsNotification {
struct ProcessingErrorAlarm
    {
        long     error_num;
        string   error_string;
    };
};
```

この構造体の内容は次のとおりです。

- error_num は，このフィールドの上位 20 ビットがベンダー固有のエラー ID 用に予約されています。
- error_string は，エラーを説明するテキスト文字列です。

第 7 章

VisiTelcoLog サービスの実行

VisiTelcoLog サービスは，C++ サービスとして実装されています。VisiTelcoLog サービスを実行するには，VisiBroker for C++ が必要です。このサービスを実行するには，VisiBroker Smart Agent（osagent プログラム）がネットワーク上で稼動していることを確認してください。バックグラウンドで VisiTelcoLog サービスを起動するには，次のコマンドを使用します。

UNIX：

```
prompt> visitelcolog  &
```

Windows：

```
prompt> start visitelcolog.exe
```

デフォルトでは，サービスはポート 14200 で起動されます。このポートを変更するには，プロパティ vbroker.dslog.listener.port を使用します。起動されたサービスは，コンソールに次のメッセージを出力します。

```
Telco Log service is ready
```

VisiTelcoLog サービスは，すべての永続的データを保存するために visidslog.dir という名前のディレクトリを作成します。デフォルトでは，このディレクトリは現在のディレクトリ内に作成されます。データ保存ディレクトリの位置は，vbroker.dslog.dir プロパティを使って変更できます。このディレクトリには，ログバックエンドも含まれます。

便宜を図るために，OMG Telecom Log Service IDL のコンパイル済みスタブ／スケルトンコードが静的ライブラリとして提供されています。使い方については，例を参照してください。生成されたスケルトンは POA 対応です。

エントリリファレンスの取得

VisiTelcoLog は，デフォルトでポート 14200 で起動されます。このポートは，vbroker.dslog.listener.port プロパティで変更できます。

BasicLogService，EventLogService，NotifyLogService，TypedEventLogService，または TypedNotifyLogService にバインドするアプリケーションは，corbaloc を使ってサービスの初期リファレンスを解決できます。

アプリケーションで使用できる ORB プロパティは次のとおりです。

```
-ORBInitRef corbaloc::<host>:<port>/BasicLogService
-ORBInitRef corbaloc::<host>:<port>/EventLogService
-ORBInitRef corbaloc::<host>:<port>/NotifyLogService
-ORBInitRef corbaloc::<host>:<port>/TypedEventLogService
-ORBInitRef corbaloc::<host>:<port>/TypedNotifyLogService
```

プロパティ

プロパティ	デフォルト値	説明
vbroker.dslog.listener.port	14200	サービスのリスナーポートを指定します。ポート範囲内の任意の正規ポート値を指定できます。
vbroker.dslog.console	true	true の場合は，サービスの起動時にコンソールに出力されます。デーモンプロセスでは，false に設定する必要があります。
vbroker.dslog.dir	./visidslog.dir	サービスは，すべての永続的データを指定されたディレクトリに保存します。ディレクトリが有効でないか，正しいアクセス許可がない場合は，サービスの起動に失敗します。任意の有効なディレクトリ位置を指定できます。
vbroker.dslog.getRecListMax	1000	クエリーで何個の LogRecord が一致したら反復子を戻すかを指定します。
vbroker.dslog.scheduleByServerLocalTime	false	true に設定された場合は，スケジューラの時間に対して tzset() を呼び出します。
vbroker.dslog.waitForLogAvailable	20	プルサプライヤがプルされたイベントのログ処理に空き容量を使用できるようになるまで待機する時間（秒単位）です。0 以外の待機時間（秒単位）を指定できます。
vbroker.dslog.basicLogFactory.name	VisiBasicLogFactory	アクティブ化する BasicLog ファクトリの名前です。任意のオブジェクト名を指定できます。
vbroker.dslog.basicLogFactory.iorFile	null	BasicLog ファクトリオブジェクトの IOR が書き込まれるファイルの名前です。任意の有効なファイル名を指定できます。
vbroker.dslog.eventLogFactory.name	VisiEventLogFactory	アクティブ化するイベントログファクトリの名前です。任意のオブジェクト名を指定できます。
vbroker.dslog.eventLogFactory.iorFile	null	イベントログファクトリオブジェクトの IOR が書き込まれるファイルの名前です。任意の有効なファイル名を指定できます。
vbroker.dslog.notifyLogFactory.name	VisiNotifyLogFactory	アクティブ化する通知ログファクトリの名前です。任意のオブジェクト名を指定できます。
vbroker.dslog.notifyLogFactory.iorFile	null	通知ログファクトリオブジェクトの IOR が書き込まれるファイルの名前です。任意の有効なファイル名を指定できます。
vbroker.dslog.backend.garbageCollectorInterval	60	ログレコードガベージコレクタスレッドを実行する時間間隔（分単位）です。このスレッドが実行されると，期限切れのすべてのログレコードがガベージコレクションの対象になります。このスレッドは，ログのレコード存続期間が指定されている場合にだけ実行されます。そうでない場合は実行されません。1 ～ 180 分の値を指定できます。
vbroker.dslog.backend.file.fragmentationLimit	75%	自動的なフラグメンテーション解消がトリガーされるときのフラグメンテーションの割合です。自動的なフラグメンテーション解消は，削除が行われるときにだけ実行されます。10 ～ 80% の値を指定できます。

プロパティ	デフォルト値	説明
vbroker.dslog.backend.file.dir	null	バックエンドデータベースとサポートファイルのディレクトリ位置です。ディレクトリパスが有効であり，適切なアクセス許可がある必要があります。サービスのパフォーマンスがこのディレクトリに依存することに注意してください。
vbroker.log.enable	false	このサーバーのデバッグログステートメントを表示するには，このプロパティを true に設定します。デバッグログフィルタのさまざまなソース名オプションについては，『VisiBroker for C++ 開発者ガイド』の「デバッグログのプロパティ」を参照してください。

索引