WebSphere eXtreme Scale バージョン 7.0
プログラミング・ガイド

WebSphere eXtreme Scale プログラミング・ガイド

WebSphere eXtreme Scale バージョン 7.0
プログラミング・ガイド

WebSphere eXtreme Scale プログラミング・ガイド

本書は、WebSphere eXtreme Scale のバージョン 7、リリース 0、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： WebSphere® eXtreme Scale Version 7.0
Administration Guide
WebSphere eXtreme Scale Administration Guide

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2009.11

目次

図

表

プログラミング・ガイド 情報

WebSphere® eXtreme Scale の資料セットには、WebSphere eXtreme Scale 製品の使用、プログラミング、および管理に必要な情報を提供する 3 つのボリュームがあります。

WebSphere eXtreme Scale ライブラリー

WebSphere eXtreme Scale ライブラリーには、以下の資料が含まれます。

- *管理ガイド* には、アプリケーション・デプロイメント計画の作成方法、容量計画の作成方法、製品のインストールと構成方法、サーバーの始動と停止方法、環境のモニター方法、環境の保護方法など、システム管理者に必要な情報が含まれます。
- *プログラミング・ガイド* には、掲載されている API 情報を使用して WebSphere eXtreme Scale 用のアプリケーションを開発する方法に関する、アプリケーション開発者のための情報が含まれます。
- *製品概要* には、ユース・ケース・シナリオ、およびチュートリアルなど、WebSphere eXtreme Scale 概念の高水準の観点が含まれます。

これらの資料をダウンロードするには、WebSphere eXtreme Scale ライブラリー・ページにアクセスしてください。

このライブラリーと同じ情報は、WebSphere eXtreme Scale インフォメーション・センターからも入手することができます。

本書の対象者

本書は、主にアプリケーション開発者の方々を対象としています。

本書の構成

本書には、以下の主要なトピックに関する情報が入っています。

- **第 1 章**には、WebSphere eXtreme Scale 入門に関する情報が含まれています。
- **第 2 章**には、WebSphere eXtreme Scale のプログラム化の方法に関する情報が含まれています。
- **第 3 章**には、データへのアクセスに関する情報が含まれています。
- **第 4 章**には、システム API とプラグインに関する情報が含まれています。
- **第 5 章**には、Spring フレームワークとの統合に関する情報が含まれています。
- **第 6 章**には、セキュリティー API に関する情報が含まれています。
- **第 7 章**には、管理 API に関する情報が含まれています。
- **第 8 章**には、パフォーマンスの考慮に関する情報が含まれています。
- **第 9 章**には、トラブルシューティングに関する情報が含まれています。
- **第 10 章**には、製品の用語集が含まれています。

本書の更新の取得

本書の更新は、WebSphere eXtreme Scale ライブラリー・ページから最新のバージョンをダウンロードすることで取得できます。

ご意見の送付方法

文書チームにご連絡ください。必要な情報が見つかりましたか? それは正確で完全な情報でしたか? 本書に関するご意見は、電子メールで wasdoc@us.ibm.com までお寄せください。

第 1 章 WebSphere eXtreme Scale 入門

WebSphere eXtreme Scale をスタンドアロン環境にインストールすると、以下の手順でメモリー内のデータ・グリッドとしてのその機能を簡単に導入できます。

スタンドアロンでインストールした WebSphere eXtreme Scale に組み込まれているサンプルを使用すると、インストールした製品を検証し、単純な eXtreme Scale グリッドおよびクライアントの使用方法を実際に確かめることができます。この入門サンプルは *installRoot*/ObjectGrid/gettingstarted ディレクトリーにあります。

この入門サンプルは、eXtreme Scale の機能および基本的な操作を紹介するものです。このサンプルは、シェル・スクリプトおよびバッチ・スクリプトからなります。これらは、ほんの少しカスタマイズするだけで単純なグリッドを開始できるよう設計されています。 さらに、この基本的なグリッドに対して単純な作成、読み取り、更新、および削除 (CRUD) 機能を実行するためのクライアント・プログラムが、ソースを含めて提供されています。

スクリプトとその機能

このサンプルには、以下の 4 つのスクリプトがあります。

env.sh|bat: このスクリプトは、他のスクリプトから呼び出されて、必要な環境変数を設定します。通常は、このスクリプトを変更する必要はありません。

- UNIX Linux `./env.sh`
- Windows `env.bat`

runcat.sh|bat: このスクリプトは、ローカル・システム上で eXtreme Scale カタログ・サービス・プロセスを開始します。

- UNIX Linux `./runcat.sh`
- Windows `runcat.bat`

runcontainer.sh|bat: このスクリプトは、コンテナー・サーバー・プロセスを開始します。指定した固有のサーバー名を使用してこのスクリプトを複数回実行すれば、いくつでもコンテナーを開始できます。それらのインスタンスが一緒に、グリッド内の区画化された冗長な情報をホストすることができます。

- UNIX Linux `./runcontainer.sh` *`unique_server_name`*
- Windows `runcontainer.bat` *`unique_server_name`*

runclient.sh|bat: このスクリプトは、単純な CRUD クライアントを実行し、指定された操作を開始します。

- UNIX Linux `./runclient.sh` *`command value1 value2`*
- Windows `runclient.sh` *`command value1 value2`*

command には、以下のいずれかのオプションを使用します。

- *value2* を、キー *value1* を持つグリッドに挿入するには、`i` と指定します。
- *value1* のキーのオブジェクトを *value2* に更新するには、`u` と指定します。
- *value1* のキーのオブジェクトを削除するには、`d` と指定します。
- *value1* のキーのオブジェクトを検索して表示するには、`g` を指定します。

注: *installRoot*`/ObjectGrid/ gettingstarted/src/Client.java` ファイルは、カタログ・サーバーへの接続、ObjectGrid インスタンスの取得、および ObjectMap API の使用をどのように行うのかを示すクライアント・プログラムです。

基本的な手順

次の手順で最初のグリッドを開始し、グリッドと対話するクライアントを実行します。

1. 端末セッションまたはコマンド行ウィンドウを開きます。
2. 次のコマンドを使用して、`gettingstarted` ディレクトリーに移動します。

   ```
   cd installRoot/ObjectGrid/gettingstarted
   ```

   installRoot の部分は、eXtreme Scale インストール・ルート・ディレクトリーへのパス、または、抽出した eXtreme Scale trial *installRoot* のルート・ファイル・パスに置き換えてください。
3. JAVA_HOME 環境変数を設定またはエクスポートして、有効な JDK または JRE バージョン 1.5 以降のインストール・ディレクトリーを指すようにします。

   UNIX Linux export JAVA_HOME=*Java_home_directory*

   Windows set JAVA_HOME=*Java_home_directory*
4. 次のスクリプトを実行して、ローカル・ホストでカタログ・サービス・プロセスを開始します。

   - UNIX Linux `./runcat.sh`
   - Windows `runcat.bat`

   カタログ・サービス・プロセスは、現行の端末ウィンドウで実行されます。
5. 別の端末セッションまたはコマンド行ウィンドウを開き、次のコマンドを実行して、コンテナー・サーバー・インスタンスを開始します。

   - UNIX Linux `./runcontainer.sh server0`
   - Windows `runcontainer.bat server0`

   コンテナー・サーバーは、現行の端末ウィンドウで実行されます。複製をサポートするためにさらに多くのコンテナー・サーバー・インスタンスを開始したい場合は、ステップ 5 と 6 を繰り返すことができます。
6. クライアント・コマンドを実行するため、別の端末セッションまたはコマンド行ウィンドウを開きます。

- グリッドへのデータの追加:
  - UNIX Linux `./runclient.sh i key1 helloWorld`
  - Windows `runclient.bat i key1 helloWorld`
- 値を検索して表示:
  - UNIX Linux `./runclient.sh g key1`
  - Windows `runclient.bat g key1`
- 値の更新:
  - UNIX Linux `./runclient.sh u key1 goodbyeWorld`
  - Windows `runclient.bat u key1 goodbyeWorld`
- 値の削除:
  - UNIX Linux `./runclient.sh d key1`
  - Windows `runclient.bat d key1`

7. <ctrl+c> を使用して、カタログ・サービス・プロセスおよびコンテナー・サーバーをそれぞれのウィンドウで停止します。

ObjectGrid の定義

サンプルでは、コンテナー・サーバーを開始するため、 *installRoot*/ObjectGrid/gettingstarted/xml ディレクトリーにある objectgrid.xml ファイルと deployment.xml ファイルが使用されます。objectgrid.xml ファイルは ObjectGrid 記述子 XML ファイルであり、deployment.xml ファイルは ObjectGrid デプロイメント・ポリシー記述子 XML ファイルです。両方のファイルが一緒になって、分散 ObjectGrid トポロジーが定義されます。

ObjectGrid 記述子 XML ファイル

ObjectGrid 記述子 XML ファイルは、アプリケーションによって使用される ObjectGrid の構造を定義するのに使用されます。このファイルには、BackingMap 構成のリストが含まれます。 これらの BackingMap はキャッシュ・データ用の実際のデータ・ストレージです。以下の例は、objectgrid.xml ファイルのサンプルです。ファイルの最初の数行には、各 ObjectGrid XML ファイルの必須ヘッダーが含まれています。このサンプル・ファイルは、Map1 と Map2 という BackingMap がある、Grid ObjectGrid を定義しています。

```
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">

    <objectGrids>
        <objectGrid name="Grid">
            <backingMap name="Map1" />
            <backingMap name="Map2" />
        </objectGrid>
    </objectGrids>

</objectGridConfig>
```

デプロイメント・ポリシー記述子 XML ファイル

デプロイメント・ポリシー記述子 XML ファイルは、始動時に ObjectGrid コンテナー・サーバーに渡されます。デプロイメント・ポリシーは ObjectGrid XML ファイルと一緒に使用する必要があり、一緒に使用される ObjectGrid XML と互換でなければなりません。デプロイメント・ポリシー内の各 objectgridDeployment 要素ごとに、対応する 1 つの ObjectGrid 要素が ObjectGrid XML 内に必要です。objectgridDeployment 要素内に定義された backingMap 要素は、ObjectGrid XML 内にある backingMap と整合していなければなりません。すべての backingMap は、1 つの mapSet 内のみで参照する必要があります。

デプロイメント・ポリシー記述子 XML ファイルは、対応する ObjectGrid XML である `objectgrid.xml` ファイルと対で使用されることを想定しています。以下の例では、`deployment.xml` ファイルの最初の数行には、各デプロイメント・ポリシー XML ファイルの必須ヘッダーが含まれています。このファイルは、`objectgrid.xml` ファイル内に定義された Grid ObjectGrid の objectgridDeployment 要素を定義しています。 Grid ObjectGrid 内に定義された Map1 と Map2 の両 BackingMap は、mapSet mapSet に含まれていて、そこでは numberOfPartitions、 minSyncReplicas、および maxSyncReplicas 属性が構成されています。

```
<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy ../deploymentPolicy.xsd"
 xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">

    <objectgridDeployment objectgridName="Grid">
        <mapSet name="mapSet" numberOfPartitions="13" minSyncReplicas="0" maxSyncReplicas="1" >
            <map ref="Map1"/>
            <map ref="Map2"/>
        </mapSet>
    </objectgridDeployment>

</deploymentPolicy>
```

mapSet エレメントの numberOfPartitions 属性は、mapSet の区画の数を指定します。これはオプションの属性であり、デフォルトは 1 です。この数は、グリッドに予想される容量に適した値である必要があります。

mapSet の minSyncReplicas 属性は、mapSet 内の各区画の同期複製の最小数を指定します。これはオプションの属性であり、デフォルトは 0 です。この同期複製の最小数をドメインがサポートできるまでは、プライマリーおよび複製は配置されません。minSyncReplicas 値をサポートするには、minSyncReplicas の値よりも 1 つだけ多いコンテナーが必要です。同期複製の数が minSyncReplicas の値よりも小さくなると、その区画に対しては書き込みトランザクションを行えなくなります。

mapSet の maxSyncReplicas 属性は、mapSet 内の各区画の同期複製の最大数を指定します。これはオプションの属性であり、デフォルトは 0 です。ある特定の区画でこの同期複製数にドメインが達すると、それ以降は、他の同期複製がその区画に対して配置されることはありません。まだ maxSyncReplicas 値を満たしていない場合には、この ObjectGrid をサポートできるコンテナーを追加すると、同期複製の数を増やすことができます。上のサンプルでは maxSyncReplicas は 1 に設定されていますが、これは、ドメインが最大 1 つの同期複製を置くことを意味しています。複数のコンテナー・サーバー・インスタンスを開始する場合、それらのコンテナー・サーバー・インスタンスの 1 つに、同期複製が 1 つだけ置かれます。

ObjectGrid の使用

installRoot/ObjectGrid/gettingstarted/src/ ディレクトリーにある Client.java ファイルは、カタログ・サーバーへの接続、ObjectGrid インスタンスの取得、および ObjectMap API の使用をどのように行うのかを示すクライアント・プログラムです。

クライアント・アプリケーションの観点から、WebSphere eXtreme Scale の使用は以下のステップに分解できます。

1. ClientClusterContext インスタンスを取得することによって、カタログ・サービスに接続する。
2. クライアント ObjectGrid インスタンスを取得する。
3. Session インスタンスを取得する。
4. ObjectMap インスタンスを取得する。
5. ObjectMap メソッドを使用する。

1. ClientClusterContext インスタンスを取得することによって、カタログ・サービスに接続する

カタログ・サーバーに接続するには、ObjectGridManager API の connect メソッドを使用します。このサンプルで使用されている connect メソッドが必要とするのは、hostname:port という形式のカタログ・サーバー・エンドポイントのみです (例えば localhost:2809)。カタログ・サーバーへの接続が成功すれば、connect メソッドは ClientClusterContext インスタンスを戻します。この ClientClusterContext インスタンスは、ObjectGridManager API から ObjectGrid を取得するのに必要です。以下のコード・スニペットは、カタログ・サーバーへの接続方法と ClientClusterContext インスタンスの取得方法を示します。

```
ClientClusterContext ccc = ObjectGridManagerFactory.getObjectGridManager().connect("localhost:2809", null, null);
```

2. ObjectGrid インスタンスを取得する

ObjectGrid インスタンスを取得するには、ObjectGridManager API の getObjectGrid メソッドを使用します。 getObjectGrid メソッドは、ClientClusterContext インスタンスと、 ObjectGrid インスタンスの名前との両方を必要とします。
ClientClusterContext インスタンスは、カタログ・サーバーへの接続中に取得されます。ObjectGrid の名前は、objectgrid.xml ファイルに指定されている「Grid」です。以下のコード・スニペットは、ObjectGridManager API の getObjectGrid メソッドを呼び出すことによって ObjectGrid を取得する方法を示します。

```
ObjectGrid grid = ObjectGridManagerFactory.getObjectGridManager().getObjectGrid(ccc, "Grid");
```

3. Session インスタンスを取得する

取得した ObjectGrid インスタンスから、Session を取得することができます。
Session インスタンスは、ObjectMap の取得とトランザクション区分の実行のために必要です。以下のコード・スニペットは、ObjectGrid API の getSession メソッドを呼び出すことによって Session を取得する方法を示します。

```
Session sess = grid.getSession();
```

4. ObjectMap インスタンスを取得する

Session を取得した後、Session API の getMap メソッドを呼び出すことによって、Session から ObjectMap を取得することができます。ObjectMap を取得するため、マップ名を getMap メソッドにパラメーターとして渡す必要があります。以下のコード・スニペットは、Session API の getMap メソッドを呼び出すことによって ObjectMap を取得する方法を示します。

```
ObjectMap map1 = sess.getMap("Map1");
```

5. ObjectMap メソッドを使用する

ObjectMap を取得した後、ObjectMap API を使用できます。ObjectMap はトランザクション・マップであり、Session API の begin メソッドおよび commit メソッドを使用することによるトランザクション区分を必要とすることに注意してください。明示的なトランザクション区分がない場合、ObjectMap 操作は自動コミット・トランザクションで実行します。

以下のコード・スニペットは、自動コミット・トランザクションでの ObjectMap API の使用方法を示しています。

```
map1.insert(key1, value1);
```

以下のコード・スニペットは、明示的なトランザクション区分がある場合の ObjectMap API の使用方法を示しています。

```
sess.begin();
map1.insert(key1, value1);
sess.commit();
```

追加情報

このサンプルは、カタログ・サーバーおよびコンテナー・サーバーを開始する方法と、スタンドアロン環境での ObjectMap API の使用を示しています。EntityManager API を使用することもできます。

WebSphere eXtreme Scale がインストールされているか使用可能にされている WebSphere Application Server 環境では、最も一般的なシナリオは、ネットワーク接続されたトポロジーです。 ネットワーク接続トポロジーでは、カタログ・サーバーは WebSphere Application Server デプロイメント・マネージャー・プロセス内でホストされ、各 WebSphere Application Server インスタンスが 1 つの eXtreme Scale サーバーを自動的にホストします。Java™ Platform, Enterprise Edition アプリケーションは、ObjectGrid 記述子 XML ファイルと ObjectGrid デプロイメント・ポリシー記述子 XML ファイルの両方を、任意のモジュールの `META-INF` ディレクトリーに含めることのみを必要とし、ObjectGrid は自動的に使用可能になります。 その後、アプリケーションはローカルで使用可能なカタログ・サーバーに接続し、使用する ObjectGrid インスタンスを取得できます。

第 2 章 WebSphere eXtreme Scale のプログラミング

WebSphere eXtreme Scale が提供するいくつかの機能には、Java プログラミング言語を使用し、いくつかのアプリケーション・プログラミング・インターフェース (API) およびシステム・プログラミング・インターフェースを通して、プログラマチックにアクセスできます。

プログラミング・モデル

次の図は eXtreme Scale プログラミング・モデルの概要を示しています。

WebSphere eXtreme Scale API

eXtreme Scale API を使用する場合は、トランザクション操作と非トランザクション操作とを区別する必要があります。トランザクション操作は、トランザクション内で実行される操作です。図では、ObjectMap、EntityManager、Query、および DataGrid API は、1 つのトランザクション・コンテナーである Session 内に包含されているトランザクション API です。非トランザクション操作は、構成操作などのトランザクションとは無関係です。

ObjectGrid、BackingMap、およびプラグイン API は、非トランザクションです。ObjectGrid、BackingMap、およびその他の構成 API は、ObjectGrid コア API としてカテゴリー化されます。プラグインは、キャッシュをカスタマイズして必要な機能を実現するためのものであり、システム・プログラミング API に分類されます。eXtreme Scale のプラグインは、ObjectGrid および BackingMap を含むプラグ可能な eXtreme Scale コンポーネントに特定の機能を提供するコンポーネントです。フィーチャーは、ObjectGrid、Session、BackingMap、ObjectMap など、eXtreme Scale コンポーネントの特定の機能または特性を表します。通常、フィーチャーは構成 API を使用して構成可能です。プラグインは、組み込むことができますが、状況によっては、独自のプラグインの開発が必要となる場合があります。

通常は、ObjectGrid および BackingMap を構成して、ユーザー・アプリケーションの要件を満たすことができます。アプリケーションに特殊な要件が存在する場合は、専用プラグインの使用を検討してください。WebSphere eXtreme Scale が要件を満たす組み込みプラグインを備えている場合があります。例えば、2 つのローカル ObjectGrid インスタンス間または 2 つの分散 eXtreme Scale グリッド間にピアツーピア複製モデルが必要な場合は、組み込み JMSObjectGridEventListener を使用することができます。 組み込みプラグインがどれもビジネス上の問題を解決できない場合は、『システム・プログラミング API』を参照して独自のプラグインを用意してください。

ObjectMap は、単純なマップ・ベースの API です。キャッシュされたオブジェクトが単純で相互関係がない場合、アプリケーションには ObjectMap API が理想的です。オブジェクト関係がある場合は、グラフのような関係をサポートする EntityManager API を使用してください。

Query は、ObjectGrid 内のデータを検索する強力なメカニズムです。Session と EntityManager は両方とも、従来の照会機能を提供します。

DataGrid API は、多くのマシン、複製、および区画を含む分散 eXtreme Scale 環境における強力なコンピューティング機能です。アプリケーションは、分散 eXtreme Scale 環境内のすべてのノードでビジネス・ロジックを並行して実行できます。アプリケーションは、ObjectMap API を介して DataGrid API を取得できます。

第 3 章 WebSphere eXtreme Scale 内のデータへのアクセス

アプリケーションが ObjectGrid インスタンスへの参照を取得すると、WebSphere eXtreme Scale でデータと対話することができます。ObjectGridManager API と共に、ローカル・インスタンスを作成するために createObjectGrid メソッドの 1 つを使用するか、分散グリッド内のクライアント・インスタンスに対して getObjectGrid メソッドを使用します。

アプリケーション内のスレッドには、独自のセッションが必要です。アプリケーションがスレッド上の ObjectGrid を使用するときには、単一の getSession メソッドのみを呼び出して、取得するようにします。この操作は低コストです。ほとんどの場合これらの操作をプールする必要はありません。アプリケーションが、Spring のような依存性注入フレームワークを使用する場合、必要なときにセッションをアプリケーション Bean に注入することができます。

セッションを取得した後、アプリケーションは ObjectGrid 内のマップに保管されたデータにアクセスできます。ObjectGrid がエンティティーを使用する場合、Session.getEntityManager メソッドで取得できる EntityManager API を使用できます。 EntityManager インターフェースは、Java 仕様に近いため、マップ・ベースの API よりもシンプルです。しかし、 EntityManager API はオブジェクト内の変更を追跡するため、パフォーマンスのオーバーヘッドが生じます。マップ・ベースの API は Session.getMap メソッドを使用して取得されます。

WebSphere eXtreme Scale はトランザクションを使用します。アプリケーションがセッションとの対話を行う場合、そのセッションはトランザクションのコンテキストの中にある必要があります。トランザクションはセッション・オブジェクトの Session.begin、Session.commit、および Session.rollback メソッドを使用して、開始されたり、コミットまたはロールバックされます。アプリケーションは、自動コミット・モードで作業を行うこともできます。このモードの場合、アプリケーションがマップとの対話を行うたび、セッションが自動的に開始し、トランザクションをコミットします。ただし、自動コミット・モードは低速です。

トランザクション使用のロジック

トランザクションは遅く見えるかもしれませんが、eXtreme Scale は次の 3 つの理由でトランザクションを使用します。

1. 例外が発生した場合や、状態変更を元に戻すことをビジネス・ロジックが必要とする場合に、変更のロールバックが可能であること。
2. 1 つのトランザクションの存続時間中にデータに対するロックの保持と解除を行うことで、一連の変更がアトミックに行われる、つまり、データに対してすべての変更を行うか、何も変更しないかにできること。
3. 複製のアトミックな単位を生成できること。

WebSphere eXtreme Scale は、セッションを使用して、本当に必要なトランザクションの量をカスタマイズします。アプリケーションでロールバック・サポートおよび

ロックをオフにすることもできますが、アプリケーション側の負担もあります。そのアプリケーションが、これらの失われた機能の処理を行う必要があります。

例えば、アプリケーションで BackingMap ロック・ストラテジーを NONE に構成することで、ロックをオフにすることができます。このストラテジーは高速ですが、並行トランザクションが互いに保護されずに、同じデータを変更できるようになります。NONE を使用する場合は、そのアプリケーションが、すべてのロックおよびデータの整合性に対する責任を持つことになります。

アプリケーションは、トランザクションによってアクセスされたときのオブジェクトのコピー方法を変更することもできます。アプリケーションは、ObjectMap.setCopyMode メソッドを使用して、オブジェクトがどのようにコピーされるのかを指定できます。このメソッドを使用して、CopyMode をオフにすることができます。通常、CopyMode をオフにする操作は、1 つのトランザクション内で同じオブジェクトに対して複数の異なる値が戻されることもある場合に、読み取り専用トランザクションに対して使用されます。1 つのトランザクション内で同じオブジェクトに対して複数の異なる値が戻されることがあり得ます。

例えば、トランザクションが T1 でオブジェクトに対して ObjectMap.get メソッドを呼び出した場合、その時点での値を取得します。その後の T2 で、そのトランザクションの中で get メソッドが再度呼び出された場合、値は別のスレッドによって変更されている可能性があります。値は別のスレッドによって変更されたため、アプリケーションは異なる値を取得することになります。 NONE CopyMode 値を使用して取得されたオブジェクトがアプリケーションによって変更されると、そのオブジェクトのコミット済みのコピーが直接変更されます。このモードでは、トランザクションのロールバックは意味がありません。ObjectGrids での唯一のコピーが変更されます。NONE CopyMode を使用すると処理は速くなりますが、その影響に注意する必要があります。NONE CopyMode を使用するアプリケーションは、トランザクションを決してロールバックしてはなりません。もしアプリケーションがトランザクションをロールバックした場合、索引に変更を反映する更新は行われず、かつ、複製がオンにされていても変更は複製されません。デフォルト値を使用するほうが簡単で、誤りの可能性も低くなります。データ信頼性を犠牲にしてもパフォーマンスを上げたい場合は、意図しない問題を回避するために、アプリケーションは実行内容をよく認識する必要があります。

注意:
ロック値または CopyMode 値のどちらかを変更するときは、慎重に行ってください。これらの値を変更すると、予測不能なアプリケーション動作が発生します。

保管データとの対話

セッションが取得された後、以下のコード断片を使用して、データを挿入するための Map API を使用できます。

```
Session session = ...;
ObjectMap personMap = session.getMap("PERSON");
session.begin();
Person p = new Person();
p.name = "John Doe";
personMap.insert(p.name, p);
session.commit();
```

以下は、EntityManager API を使用した場合の同じ例です。このコード例は、Person オブジェクトがエンティティーにマップされていると想定しています。

```
Session session = ...;
EntityManager em = session.getEntityManager();
session.begin();
Person p = new Person();
p.name = "John Doe";
em.persist(p);
session.commit();
```

このパターンは、スレッドが使用するマップの ObjectMap への参照を取得し、トランザクションを開始し、データを操作し、トランザクションをコミットするように設計されています。

ObjectMap インターフェースには、put、get、および remove などの一般的なマップ操作が含まれています。しかし、 get、getForUpdate、insert、update、および remove といった、より具体的な操作名を使用してください。これらのメソッドは、従来のマップ API より意図を正確に伝えます。

また、フレキシブルな索引付けサポートを使用することもできます。

以下に、Object の更新の例を示します。

```
session.begin();
Person p = (Person)personMap.getForUpdate("John Doe");
p.name = "John Doe";
p.age = 30;
personMap.update(p.name, p);
session.commit();
```

アプリケーションでは、通常は、単純な get ではなく、getForUpdate メソッドを使用してレコードをロックします。 update メソッドは、更新済みの値を実際にマップに提供するために呼び出す必要があります。 update を呼び出さないと、そのマップは変更されません。以下は、EntityManager API を使用した場合の同じコード断片です。

```
session.begin();
Person p = (Person)em.findForUpdate(Person.class, "John Doe");
p.age = 30;
session.commit();
```

EntityManager API はマップを使用した方法よりも単純です。このケースでは、eXtreme Scale がエンティティーを検索し、管理対象オブジェクトをアプリケーションに返します。アプリケーションがオブジェクトを変更し、トランザクションをコミットすると、eXtreme Scale は、管理対象オブジェクトに加えられた変更をコミット時に自動的に追跡し、必要な更新を行います。

トランザクションと区画

WebSphere eXtreme Scale トランザクションは、単一の区画のみ、更新することができます。クライアントからのトランザクションは複数の区画から読み取ることができますが、更新できるのは 1 つの区画のみです。アプリケーションが 2 つの区画の更新を試行すると、トランザクションは失敗し、ロールバックが行われます。組み込まれている ObjectGrid (グリッド・ロジック) を使用するトランザクションには、ルーティング機能はなく、ローカル区画内のデータしか認識できません。このビジネス・ロジックは、常に 2 番目のセッションを取得することができます。この

2 番目のセッションは、他の区画にアクセスするための、本当のクライアント・セッションです。ただし、このトランザクションは独立したトランザクションです。

照会と区画

トランザクションが既にエンティティーを検索済みの場合、そのトランザクションは、そのエンティティーの区画に関連付けられます。エンティティーと関連付けられたトランザクションで実行する照会は、関連付けられた区画にルーティングされます。

以前に関連付けられた区画で照会が実行される場合は、照会に使用される区画 ID を設定する必要があります。区画 ID は整数値です。これで、その照会はその区画にルーティングされます。

照会は単一の区画内のみを検索します。ただし、それと同時に同じ照会を、DataGrid API を使用してすべての区画または区画のサブセットに対して実行します。どの区画にあるかわからないエントリーを検索するには、DataGrid API を使用します。

ObjectGridManager を使用した ObjectGrid との対話

ObjectGridManagerFactory クラスと ObjectGridManager インターフェースは、ObjectGrid インスタンスの作成、アクセス、およびキャッシュを行うメカニズムを提供します。 ObjectGridManagerFactory クラスは、ObjectGridManager インターフェースにアクセスする静的ヘルパー・クラスであり、singleton クラスです。ObjectGridManager インターフェースには、 ObjectGrid オブジェクトのインスタンスを作成するいくつかの便利なメソッドがあります。また、ObjectGridManager は、複数のユーザーがアクセス可能な ObjectGrid インスタンスの作成とキャッシングも容易にします。

プログラミング・モデル

eXtreme Scale の機能をメモリー内データ・グリッドとして使用する前に、次のようなメソッドを使用して ObjectGrid インスタンスを作成し、これと対話する必要があります。

- createObjectGrid メソッド
- getObjectGrid メソッド
- removeObjectGrid メソッド
- ObjectGrid のライフサイクルの制御

createObjectGrid メソッド

このトピックでは、ObjectGridManager インターフェースの 7 つの createObjectGrid メソッドについて説明します。これらのメソッドはそれぞれ、ObjectGrid のローカル・インスタンスを 1 つ作成します。

ローカルのメモリー内インスタンス

以下のコード・スニペットは、eXtreme Scale でローカル ObjectGrid インスタンスを取得および構成する方法を示しています。

```
// Obtain a local ObjectGrid reference
        // you can create a new ObjectGrid, or get configured ObjectGrid
        // defined in ObjectGrid xml file
        ObjectGridManager objectGridManager =
     ObjectGridManagerFactory.getObjectGridManager();
        ObjectGrid ivObjectGrid =
     objectGridManager.createObjectGrid("objectgridName");

        // Add a TransactionCallback into ObjectGrid
        HeapTransactionCallback tcb = new HeapTransactionCallback();
        ivObjectGrid.setTransactionCallback(tcb);

        // Define a BackingMap
        // if the BackingMap is configured in ObjectGrid xml
        // file, you can just get it.
        BackingMap ivBackingMap = ivObjectGrid.defineMap("myMap");

        // Add a Loader into BackingMap
        Loader ivLoader = new HeapCacheLoader();
        ivBackingMap.setLoader(ivLoader);

        // initialize ObjectGrid
        ivObjectGrid.initialize();

        // Obtain a session to be used by the current thread.
        // Session can not be shared by multiple threads
        Session ivSession = ivObjectGrid.getSession();

        // Obtaining ObjectMap from ObjectGrid Session
        ObjectMap objectMap = ivSession.getMap("myMap");
```

デフォルトの共用構成

以下のコードは、ObjectGrid を作成して多くのユーザー間で共用する単純なケースです。

```
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
final ObjectGridManager oGridManager=
  ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid employees =
 oGridManager.createObjectGrid("Employees",true);
employees.initialize();
employees.
/*sample continues..*/
```

前の Java コード・スニペットは、Employees ObjectGrid を作成し、キャッシュに入れます。Employees ObjectGrid は、デフォルト構成によって初期化され、すぐに使用できる状態になっています。createObjectGrid メソッド内の 2 番目のパラメーターは、true に設定されます。これにより、ObjectGridManager は、作成した ObjectGrid インスタンスをキャッシュに入れるよう指示されます。このパラメーターが false に設定されている場合、インスタンスはキャッシュに入れられません。各 ObjectGrid インスタンスには name があり、その名前に基づき、多くのクライアントまたはユーザー間でそのインスタンスを共用できます。

objectGrid インスタンスがピアツーピア共用で使用されている場合は、キャッシングを true に設定する必要があります。ピアツーピア共用について詳しくは、『ピア Java 仮想マシン間の変更の配布』を参照してください。

XML 構成

WebSphere eXtreme Scale は高度な構成が可能です。前の例では、構成を伴わない単純な ObjectGrid を作成する方法を示しました。この例では、XML 構成ファイルに基づいて事前構成された ObjectGrid インスタンスを作成する方法が示されています。ObjectGrid インスタンスは、プログラマチックに構成するか、または XML ベースの構成ファイルを使用して構成することができます。これら 2 つの方法を組み合わせて ObjectGrid を構成することもできます。 ObjectGridManager インターフェースを使用すると、 XML 構成に基づいて ObjectGrid インスタンスを作成できます。 ObjectGridManager インターフェースには、URL を引数として取るいくつかのメソッドがあります。 ObjectGridManager 内に渡される各 XML ファイルについて、スキーマに対する妥当性検査を行う必要があります。 XML の妥当性検査は、以前にファイルの妥当性検査が行われ、最後の妥当検査以降、そのファイルに対しては変更が行われていない場合に限り、使用不可にすることができます。妥当性検査を使用不可にすると、少量のオーバーヘッドが節約されますが、無効な XML ファイルが使用される可能性が生じます。 IBM® Java Developer Kit (JDK) 1.4.2 は、XML 妥当性検査をサポートします。これをサポートしない JDK を使用すると、Apache Xerces で XML を妥当性検査しなければならない場合があります。

以下の Java コード・スニペットは、ObjectGrid を作成するために XML 構成ファイルを渡す方法を示しています。

```
import java.net.MalformedURLException;
import java.net.URL;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
boolean validateXML = true; // turn XML validation on
boolean cacheInstance = true; // Cache the instance
String objectGridName="Employees"; // Name of Object Grid URL
allObjectGrids = new URL("file:test/myObjectGrid.xml");
final ObjectGridManager oGridManager=
 ObjectGridManagerFactory.getObjectGridManager();
 ObjectGrid employees =
 oGridManager.createObjectGrid(objectGridName, allObjectGrids,
  bvalidateXML, cacheInstance);
```

この XML ファイルには、いくつかの ObjectGrids の構成情報が含まれています。前のコード・スニペットは、具体的に ObjectGrid Employees を返します。この場合、Employees の構成は、この XML ファイルに定義されていることを想定しています。 XML 構文については、ObjectGrid 構成を参照してください。
createObjectGrid のメソッドは 7 つ存在しますが、それらは以下のコード・ブロック内に文書化されています。

```
/**
 * A simple factory method to return an instance of an
 * Object Grid. A unique name is assigned.
 * The instance of ObjectGrid is not cached.
 * Users can then use {@link ObjectGrid#setName(String)} to change the
 * ObjectGrid name.
 *
 * @return ObjectGrid an instance of ObjectGrid with a unique name assigned
 * @throws ObjectGridException any error encountered during the
 * ObjectGrid creation
 */
public ObjectGrid createObjectGrid() throws ObjectGridException;
```

```
/**
 * A simple factory method to return an instance of an ObjectGrid with the
 * specified name. The instances of ObjectGrid can be cached. If an ObjectGrid
 * with the this name has already been cached, an ObjectGridException
 * will be thrown.
 *
 * @param objectGridName the name of the ObjectGrid to be created.
 * @param cacheInstance true, if the ObjectGrid instance should be cached
 * @return an ObjectGrid instance
 * @this name has already been cached or
 * any error during the ObjectGrid creation.
 */
public ObjectGrid createObjectGrid(String objectGridName, boolean cacheInstance)
  throws ObjectGridException;

/**
 * Create an ObjectGrid instance with the specified ObjectGrid name. The
 * ObjectGrid instance created will be cached.
 * @param objectGridName the Name of the ObjectGrid instance to be created.
 * @return an ObjectGrid instance
 * @throws ObjectGridException if an ObjectGrid with this name has already
 * been cached, or any error encountered during the ObjectGrid creation
 */
public ObjectGrid createObjectGrid(String objectGridName)
  throws ObjectGridException;

/**
 * Create an ObjectGrid instance based on the specified ObjectGrid name and the
 * XML file. The ObjectGrid instance defined in the XML file with the specified
 * ObjectGrid name will be created and returned. If such an ObjectGrid
 * cannot be found in the xml file, an exception will be thrown.
 *
 * This ObjecGrid instance can be cached.
 *
 * If the URL is null, it will be simply ignored. In this case, this method behaves
 * the same as {@link #createObjectGrid(String, boolean)}.
 *
 * @param objectGridName the Name of the ObjectGrid instance to be returned. It
 * must not be null.
 * @param xmlFile a URL to a wellformed xml file based on the ObjectGrid schema.
 * @param enableXmlValidation if true the XML is validated
 * @param cacheInstance a boolean value indicating whether the ObjectGrid
 * instance(s)
 * defined in the XML will be cached or not. If true, the instance(s) will
 * be cached.
 *
 * @throws ObjectGridException if an ObjectGrid with the same name
 * has been previously cached, no ObjectGrid name can be found in the xml file,
 * or any other error during the ObjectGrid creation.
 * @return an ObjectGrid instance
 * @see ObjectGrid
 */
public ObjectGrid createObjectGrid(String objectGridName, final URL xmlFile,
final boolean enableXmlValidation, boolean cacheInstance)
 throws ObjectGridException;

/**
 * Process an XML file and create a List of ObjectGrid objects based
 * upon the file.
 * These ObjecGrid instances can be cached.
 * An ObjectGridException will be thrown when attempting to cache a
 * newly created ObjectGrid
 * that has the same name as an ObjectGrid that has already been cached.
 *
 * @param xmlFile the file that defines an ObjectGrid or multiple
 * ObjectGrids
 * @param enableXmlValidation setting to true will validate the XML
```

```
 * file against the schema
 * @param cacheInstances set to true to cache all ObjectGrid instances
 * created based on the file
 * @return an ObjectGrid instance
 * @throws ObjectGridException if attempting to create and cache an
 * ObjectGrid with the same name as
 * an ObjectGrid that has already been cached, or any other error
 * occurred during the
 * ObjectGrid creation
 */
public List createObjectGrids(final URL xmlFile, final boolean enableXmlValidation,
boolean cacheInstances) throws ObjectGridException;

/** Create all ObjectGrids that are found in the XML file. The XML file will be
 * validated against the schema. Each ObjectGrid instance that is created will
 * be cached. An ObjectGridException will be thrown when attempting to cache a
 * newly created ObjectGrid that has the same name as an ObjectGrid that has
 * already been cached.
 * @param xmlFile The XML file to process. ObjectGrids will be created based
 * on what is in the file.
 * @return A List of ObjectGrid instances that have been created.
 * @throws ObjectGridException if an ObjectGrid with the same name as any of
 * those found in the XML has already been cached, or
 * any other error encounterred during ObjectGrid creation.
 */
public List createObjectGrids(final URL xmlFile) throws ObjectGridException;

/**
 * Process the XML file and create a single ObjectGrid instance with the
 * objectGridName specified only if an ObjectGrid with that name is found in
 * the file. If there is no ObjectGrid with this name defined in the XML file,
 * an ObjectGridException
 * will be thrown. The ObjectGrid instance created will be cached.
 * @param objectGridName name of the ObjectGrid to create. This ObjectGrid
 * should be defined in the XML file.
 * @param xmlFile the XML file to process
 * @return A newly created ObjectGrid
 * @throws ObjectGridException if an ObjectGrid with the same name has been
 * previously cached, no ObjectGrid name can be found in the xml file,
 * or any other error during the ObjectGrid creation.
 */
public ObjectGrid createObjectGrid(String objectGridName, URL xmlFile)
  throws ObjectGridException;
```

getObjectGrid メソッドの呼び出し中にクライアントがハングする

ObjectGridManager の getObjectGrid メソッドの呼び出し中にクライアントがハングしているように見えたり、例外 com.ibm.websphere.projector.MetadataException がスローされたりすることがあります。EntityMetadata リポジトリーは使用できず、タイムアウトしきい値に達します。その理由は、クライアントが、ObjectGrid サーバーでエンティティー・メタデータが使用可能になるのを待っているためです。このエラーは、コンテナーは開始したけれども、まだコンテナーの初期の数または同期複製の最小数に達していない場合に発生することがあります。ObjectGrid のデプロイメント・ポリシーを参照し、アクティブ・コンテナーの数が、デプロイメント・ポリシー記述子ファイルの numInitialContainers 属性および minSyncReplicas 属性の両方の値以上であることを確認してください。

getObjectGrid メソッド

キャッシュに入れられたインスタンスを取り出すには、ObjectGridManager.getObjectGrid メソッドを使用します。

キャッシュに入れられたインスタンスの取得

Employees ObjectGrid インスタンスは、ObjectGridManager インターフェースによってキャッシュに入れられているため、別のユーザーが以下のコード・スニペットを使用してアクセスすることができます。

```
ObjectGrid myEmployees = oGridManager.getObjectGrid("Employees");
```

キャッシュに入れられた ObjectGrid インスタンスを戻す 2 つの getObjectGrid メソッドを以下に示します。

- **キャッシュに入れられたすべてのインスタンスを取得する**

  以前にキャッシュに入れられたすべての ObjectGrid インスタンスを取得するには、getObjectGrids メソッドを使用します。これは各インスタンスのリストを戻します。キャッシュに入れられたインスタンスが存在しない場合、このメソッドは null を戻します。

- **キャッシュに入れられたインスタンスを名前で取得する**

  キャッシュに入れられた ObjectGrid の 1 つのインスタンスを取得するには、getObjectGrid(String objectGridName) を使用し、キャッシュに入れられたインスタンスの名前をメソッドに渡します。このメソッドは、指定された名前の ObjectGrid インスタンスを戻すか、または、その名前の ObjectGrid インスタンスがない場合は null を戻します。

注: getObjectGrid メソッドを使用して分散グリッドに接続することもできます。詳しくは、 18 ページの『分散 ObjectGrid との接続』を参照してください。

removeObjectGrid メソッド

ObjectGrid インスタンスをキャッシュから除去するには、2 つの異なる removeObjectGrid メソッドを使用できます。

ObjectGrid インスタンスの除去

キャッシュから ObjectGrid インスタンスを除去するには、 removeObjectGrid メソッドの 1 つを使用します。ObjectGridManager は、除去されたインスタンスの参照は保持しません。 2 つの 除去メソッドが存在します。 1 つのメソッドはブール値パラメーターを取ります。ブール値パラメーターが true に設定されている場合、destroy メソッドが ObjectGrid に対して呼び出されます。 ObjectGrid に対して呼び出された destroy メソッドは、ObjectGrid をシャットダウンし、ObjectGrid が使用しているリソースをすべて解放します。2 つの removeObjectGrid メソッドの使用方法の説明は以下のとおりです。

```
/**
 * Remove an ObjectGrid from the cache of ObjectGrid instances
 *
 * @param objectGridName the name of the ObjectGrid instance to remove
 * from the cache
 *
 * @throws ObjectGridException if an ObjectGrid with the objectGridName
 * was not found in the cache
 */
public void removeObjectGrid(String objectGridName) throws ObjectGridException;

/**
```

```
 * Remove an ObjectGrid from the cache of ObjectGrid instances and
 * destroy its associated resources
 *
 * @param objectGridName the name of the ObjectGrid instance to remove
 * from the cache
 *
 * @param destroy destroy the objectgrid instance and its associated
 * resources
 *
 * @throws ObjectGridException if an ObjectGrid with the objectGridName
 * was not found in the cache
 */
public void removeObjectGrid(String objectGridName, boolean destroy)
 throws ObjectGridException;
```

分散 ObjectGrid との接続

カタログ・サービスの接続エンドポイントを使用して分散 ObjectGrid に接続することができます。接続するカタログ・サーバーのホスト名とエンドポイント・ポートが必要です。

分散グリッドに接続するためには、カタログ・サービスとコンテナー・サーバーを使用してサーバー・サイド環境を構成しておく必要があります。

getObjectGrid(ClientClusterContext ccc, String objectGridName) メソッドは、指定のカタログ・サービスに接続し、サーバー・サイドの ObjectGrid インスタンスに対応するクライアント ObjectGrid インスタンスを返します。

以下のコード・スニペットは、分散グリッドへの接続方法の例です。

```
// Create an ObjectGridManager instance.

ObjectGridManager ogm = ObjectGridManagerFactory.getObjectGridManager();

// Obtain a ClientClusterContext by connecting to a catalog
// server based distributed ObjectGrid.  You have to provide
// a connection end point for your catalog server in the format
// of hostName:endPointPort.  The hostName is the machine
// where the catalog server resides, and the endPointPort is
// the catalog server's listening port, whose default is 2809.
ClientClusterContext ccc = ogm.connect("localhost:2809", null, null);

     // Obtain a distributed ObjectGrid using ObjectGridManager and providing
// the ClientClusterContext.

    ObjectGrid og = ogm.getObjectGrid(ccc, "objectgridName");
```

eXtreme Scale クライアントの構成

eXtreme Scale クライアントを要件に応じて構成でき、設定値をオーバーライドすることもできます。

次の方法で、eXtreme Scale クライアントを構成することができます。

- XML 構成
- プログラマチック構成
- Spring Framework 構成
- ニア・キャッシュの使用不可化

以下のプラグインをクライアントにオーバーライドすることができます。

- **ObjectGrid プラグイン**
  - TransactionCallback プラグイン
  - ObjectGridEventListener プラグイン
- **BackingMap プラグイン**
  - Evictor プラグイン
  - MapEventListener プラグイン
  - numberOfBuckets 属性
  - ttlEvictorType 属性
  - timeToLive 属性

XML を使用したクライアントの構成

ObjectGrid XML ファイルを使用して、クライアント・サイドで設定を変更できます。eXtreme Scale クライアントの設定を変更するには、eXtreme Scale サーバーに使用されたファイルに構造が類似している ObjectGrid XML ファイルを作成する必要があります。

以下の XML ファイルがデプロイメント・ポリシー XML ファイルと対になっており、これらのファイルを使用して eXtreme Scale サーバーが始動されたものと想定します。

companyGridServerSide.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">

    <objectGrids>
        <objectGrid name="CompanyGrid">
            <bean id="TransactionCallback"
                classname="com.company.MyTxCallback" />
            <bean id="ObjectGridEventListener"
                classname="com.company.MyOgEventListener" />
            <backingMap name="Customer"
                pluginCollectionRef="customerPlugins" />
            <backingMap name="Item" />
            <backingMap name="OrderLine" numberOfBuckets="1049"
                timeToLive="1600" ttlEvictorType="LAST_ACCESS_TIME" />
            <backingMap name="Order" lockStrategy="PESSIMISTIC"
                pluginCollectionRef="orderPlugins" />
        </objectGrid>
    </objectGrids>

    <backingMapPluginCollections>
        <backingMapPluginCollection id="customerPlugins">
            <bean id="Evictor"
                classname="com.ibm.websphere.objectGrid.plugins.builtins.LRUEvictor" />
            <bean id="MapEventListener"
                classname="com.company.MyMapEventListener" />
        </backingMapPluginCollection>
        <backingMapPluginCollection id="orderPlugins">
            <bean id="MapIndexPlugin"
                classname="com.company.MyMapIndexPlugin" />
        </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

eXtreme Scale サーバーでは、CompanyGrid は `companyGridServerSide.xml` ファイルによる定義に従って動作しますデフォルトでは CompanyGrid クライアントの設定は、サーバーで実行している CompanyGrid の設定と同じです。ただし、設定のいくつかはクライアント・サイドでオーバーライドできます。

クライアント固有の ObjectGrid を作成して、これらの設定のいくつかをオーバーライドします。サーバーの開始に使用された ObjectGrid XML ファイルをコピーし、

そのファイルを編集してクライアント・サイドでカスタマイズします。クライアントからプラグインを除去するには、className 属性の値として空ストリングを使用します。既存のプラグインを変更するには、className 属性に新しい値を指定します。プラグインがサポートされているオーバーライド・プラグインの 1 つである場合、そのプラグインをこのファイルに追加することもできます。

クライアントの属性の 1 つを設定するには、新しい値を指定します。

以下の ObjectGrid XML ファイルを使用すると、CompanyGrid クライアントの属性およびプラグインのいくつかを指定できます。

companyGridClientSide.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">

    <objectGrids>
        <objectGrid name="CompanyGrid">
            <bean id="TransactionCallback"
                className="com.company.MyClientTxCallback" />
            <bean id="ObjectGridEventListener" className="" />
            <backingMap name="Customer" numberOfBuckets="1429"
                pluginCollectionRef="customerPlugins" />
            <backingMap name="Item" />
            <backingMap name="OrderLine" numberOfBuckets="701"
                timeToLive="800" ttlEvictorType="LAST_ACCESS_TIME" />
            <backingMap name="Order" lockStrategy="PESSIMISTIC"
                pluginCollectionRef="orderPlugins" />
        </objectGrid>
    </objectGrids>

    <backingMapPluginCollections>
        <backingMapPluginCollection id="customerPlugins">
            <bean id="Evictor"
                className="com.ibm.websphere.objectGrid.plugins.builtins.LRUEvictor" />
            <bean id="MapEventListener" className="" />
        </backingMapPluginCollection>
        <backingMapPluginCollection id="orderPlugins">
            <bean id="MapIndexPlugin"
                className="com.company.MyMapIndexPlugin" />
        </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

companyGridClientSide.xml ファイルは、CompanyGrid クライアントのいくつかの属性およびプラグインをオーバーライドします。クライアント側では、TransactionCallback は、サーバー上で実行している com.company.MyTxCallback ではなく、com.company.MyClientTxCallback になります。className 値が空ストリングであるため、クライアントの ObjectGridEventListener は除去されます。

Customer backingMap の numberOfBuckets は、クライアントでは 1429 に設定されます。Customer backingMap はその Evictor をサーバー構成のまま保持しますが、MapEventListener はクライアントから除去されます。

numberOfBuckets および timeToLive は、OrderLine backingMap のクライアント・サイドで調整されています。

このファイル内の lockStrategy 属性は、指定されている値に関係なく無視されます。この属性は、クライアント・サイドでのオーバーライド用としてはサポートされていません。

companyGridClientSide.xml ファイルを使用して CompanyGrid クライアントを作成するには、ObjectGrid XML ファイルを URL として、ObjectGridManager の接続メソッドの 1 つに渡します。

Creating the client for XML

```
ObjectGridManager ogManager =
 ObjectGridManagerFactory.ObjectGridManager();
```

```
ClientClusterContext clientClusterContext =
 ogManager.connect("MyServer1.company.com:2809", null, new URL(
                "file:xml/companyGridClientSide.xml"));
```

クライアントのプログラマチック構成

クライアント・サイドの ObjectGrid 設定をプログラマチックにオーバーライドすることもできます。サーバー・サイド ObjectGrid と同様の構造を持つ ObjectGridConfiguration オブジェクトを作成します。以下のコードで、前にセクションで示された companyGridClientSide.xml を使用して作成されるものと機能的に同等なクライアント・サイド ObjectGrid が構成されます。

```
client-side override programmatically
ObjectGridConfiguration companyGridConfig = ObjectGridConfigFactory
    .createObjectGridConfiguration("CompanyGrid");
Plugin txCallbackPlugin = ObjectGridConfigFactory.createPlugin(
    PluginType.TRANSACTION_CALLBACK, "com.company.MyClientTxCallback");
companyGridConfig.addPlugin(txCallbackPlugin);

Plugin ogEventListenerPlugin = ObjectGridConfigFactory.createPlugin(
    PluginType.OBJECTGRID_EVENT_LISTENER, "");
companyGridConfig.addPlugin(ogEventListenerPlugin);

BackingMapConfiguration customerMapConfig = ObjectGridConfigFactory
    .createBackingMapConfiguration("Customer");
customerMapConfig.setNumberOfBuckets(1429);
Plugin evictorPlugin = ObjectGridConfigFactory.createPlugin(PluginType.EVICTOR,
    "com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor");
customerMapConfig.addPlugin(evictorPlugin);

companyGridConfig.addBackingMapConfiguration(customerMapConfig);

BackingMapConfiguration orderLineMapConfig = ObjectGridConfigFactory
    .createBackingMapConfiguration("OrderLine");
orderLineMapConfig.setNumberOfBuckets(701);
orderLineMapConfig.setTimeToLive(800);
orderLineMapConfig.setTtlEvictorType(TTLType.LAST_ACCESS_TIME);

companyGridConfig.addBackingMapConfiguration(orderLineMapConfig);

List ogConfigs = new ArrayList();
ogConfigs.add(companyGridConfig);

Map overrideMap = new HashMap();
overrideMap.put(CatalogServerProperties.DEFAULT_DOMAIN, ogConfigs);

ogManager.setOverrideObjectGridConfigurations(overrideMap);
ClientClusterContext client = ogManager.connect(catalogServerAddresses, null, null);
ObjectGrid companyGrid = ogManager.getObjectGrid(client, objectGridName);
```

overrideMap に組み込まれている ObjectGridConfiguration オブジェクトおよび BackingMapConfiguration オブジェクトのみが、オーバーライドされたプラグインおよび属性をチェックされます。例えば、上記のコードは、OrderLine マップ上のバケットの数をオーバーライドします。ただし、クライアント・サイドの Order マップは、その構成が組み込まれていないので未変更のままです。

Spring Framework でのクライアントの構成

クライアント・サイドの ObjectGrid 設定は、Spring Framework を使用してオーバーライドすることもできます。以下の例の XML ファイルは、ObjectGridConfiguration をビルドし、それをクライアント・サイド設定をオーバーライドするために使用する方法を示しています。この例では、プログラマチック構成で示したのと同じ API が呼び出されます。またこの例は、ObjectGrid XML 構成の例と機能的に同等です。

```
client configuration with Spring<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="companyGrid" factory-bean="manager" factory-method="getObjectGrid"
    singleton="true">
    <constructor-arg type="com.ibm.websphere.objectgrid.ClientClusterContext">
      <ref bean="client" />
    </constructor-arg>
```

```
    <constructor-arg type="java.lang.String" value="CompanyGrid" />
</bean>

  <bean id="manager" class="com.ibm.websphere.objectgrid.ObjectGridManagerFactory"
    factory-method="getObjectGridManager" singleton="true">
    <property name="overrideObjectGridConfigurations">
      <map>
        <entry key="DefaultDomain">
          <list>
            <ref bean="ogConfig" />
          </list>
        </entry>
      </map>
    </property>
  </bean>

  <bean id="ogConfig"
    class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
    factory-method="createObjectGridConfiguration">
    <constructor-arg type="java.lang.String">
      <value>CompanyGrid</value>
    </constructor-arg>
    <property name="plugins">
      <list>
   <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
          factory-method="createPlugin">
          <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
            value="TRANSACTION_CALLBACK" />
          <constructor-arg type="java.lang.String"
            value="com.company.MyClientTxCallback" />
        </bean>
        <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
          factory-method="createPlugin">
          <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
            value="OBJECTGRID_EVENT_LISTENER" />
          <constructor-arg type="java.lang.String" value="" />
        </bean>
      </list>
 </property>
      <property name="backingMapConfigurations">
        <list>
    <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
           factory-method="createBackingMapConfiguration">
           <constructor-arg type="java.lang.String" value="Customer" />
             <property name="plugins">
     <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
                factory-method="createPlugin">
                <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
     value="EVICTOR" />
   <constructor-arg type="java.lang.String"
                  value="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor" />
               </bean>
             </property>
             <property name="numberOfBuckets" value="1429" />
           </bean>
           <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
             factory-method="createBackingMapConfiguration">
               <constructor-arg type="java.lang.String" value="OrderLine" />
                 <property name="numberOfBuckets" value="701" />
   <property name="timeToLive" value="800" />
   <property name="ttlEvictorType">
                  <value type="com.ibm.websphere.objectgrid.
          TTLType">LAST_ACCESS_TIME</value>
   </property>
           </bean>
         </list>
       </property>
     </bean>

     <bean id="client" factory-bean="manager" factory-method="connect"
       singleton="true">
       <constructor-arg type="java.lang.String">
    <value>localhost:2809</value>
       </constructor-arg>
   <constructor-arg
          type="com.ibm.websphere.objectgrid.security.
     config.ClientSecurityConfiguration">
          <null />
       </constructor-arg>
   <constructor-arg type="java.net.URL">
    <null />
       </constructor-arg>
     </bean>
</beans>
```

XML ファイルの作成後、そのファイルをロードし、以下のコード・スニペットで ObjectGrid をビルドします。

```
BeanFactory beanFactory = new XmlBeanFactory(new
  UrlResource("file:test/companyGridSpring.xml"));

ObjectGrid companyGrid = (ObjectGrid) beanFactory.getBean("companyGrid");
```

詳しくは、 209 ページの『第 5 章 Spring フレームワークとの統合』を参照してください。

ニア・キャッシュの使用不可化

ニア・キャッシュは、ロックがオプティミスティックまたはロックなしに構成されている場合、デフォルトで使用可能にされており、ロックがペシミスティックに構成されている場合は使用することができません。

ニア・キャッシュを使用不可にするには、クライアント・オーバーライド ObjectGrid 記述子ファイルで numberOfBuckets 属性を 0 に設定します。

詳しくは、「*管理ガイド*」でマップ・エントリーのロックに関する説明を参照してください。

ObjectGrid のライフサイクルの制御

ObjectGridManager インターフェースで、スタートアップ Bean またはサーブレットのいずれかを使用すると ObjectGrid インスタンスのライフサイクルを制御できます。

スタートアップ Bean でのライフサイクルの管理

スタートアップ Bean は、ObjectGrid インスタンスのライフサイクルの制御に使用できます。スタートアップ Bean はアプリケーションの開始時にロードします。スタートアップ Bean では、アプリケーションが予想通りに開始または停止するときにはいつでもコードを実行できます。スタートアップ Bean を作成するために、ホームの com.ibm.websphere.startupservice.AppStartUpHome インターフェースを使用し、また、リモート側の com.ibm.websphere.startupservice.AppStartUp インターフェースを使用します。 Bean で start メソッドおよび stop メソッドを実行します。 start メソッドは、アプリケーションの始動時に必ず起動されます。 stop メソッドは、アプリケーションのシャットダウン時に必ず起動されます。 start メソッドは、ObjectGrid インスタンスの作成に使用されます。 stop メソッドは、ObjectGrid インスタンスの除去に使用されます。以下は、スタートアップ Bean でのこの ObjectGrid のライフサイクル管理を示すコード・スニペットです。

```
public class MyStartupBean implements javax.ejb.SessionBean {
    private ObjectGridManager objectGridManager;

    /* The methods on the SessionBean interface have been
     * left out of this example for the sake of brevity */

    public boolean start(){
        // Starting the startup bean
        // This method is called when the application starts
        objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
        try {
            // create 2 ObjectGrids and cache these instances
            ObjectGrid bookstoreGrid =
      objectGridManager.createObjectGrid("bookstore", true);
            bookstoreGrid.defineMap("book");
            ObjectGrid videostoreGrid =
```

```
      objectGridManager.createObjectGrid("videostore", true);
            // within the JVM,
            // these ObjectGrids can now be retrieved from the
            //ObjectGridManager using the getObjectGrid(String) method
        } catch (ObjectGridException e) {
            e.printStackTrace();
            return false;
        }

        return true;
    }

    public void stop(){
        // Stopping the startup bean
        // This method is called when the application is stopped
        try {
            // remove the cached ObjectGrids and destroy them
            objectGridManager.removeObjectGrid("bookstore", true);
            objectGridManager.removeObjectGrid("videostore", true);
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }
}
```

start メソッドが呼び出された後、新規に作成された ObjectGrid インスタンスが ObjectGridManager インターフェースから取得されます。例えば、サーブレットがアプリケーションに含まれる場合、サーブレットは以下のコード・スニペットを使用して eXtreme Scale にアクセスします。

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.getObjectGrid("bookstore");
ObjectGrid videostoreGrid = objectGridManager.getObjectGrid("videostore");
```

サーブレットでのライフサイクルの管理

サーブレットで ObjectGrid のライフサイクルを管理するためには、 init メソッドを使用して ObjectGrid インスタンスを作成したり、destroy メソッドを使用して ObjectGrid インスタンスを除去することができます。 ObjectGrid インスタンスがキャッシュされた場合、サーブレット・コードで検索および操作を行います。以下は、サーブレット内での ObjectGrid の作成、操作、および破棄を示すサンプル・コードです。

```
public class MyObjectGridServlet extends HttpServlet implements Servlet {
    private ObjectGridManager objectGridManager;

    public MyObjectGridServlet() {
        super();
    }

    public void init(ServletConfig arg0) throws ServletException {
        super.init();
        objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
        try {
            // create and cache an ObjectGrid named bookstore
            ObjectGrid bookstoreGrid =
      objectGridManager.createObjectGrid("bookstore", true);
            bookstoreGrid.defineMap("book");
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }
```

```
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        ObjectGrid bookstoreGrid = objectGridManager.getObjectGrid("bookstore");
        Session session = bookstoreGrid.getSession();
        ObjectMap bookMap = session.getMap("book");
        // perform operations on the cached ObjectGrid
        // ...
    }

    public void destroy() {
        super.destroy();
        try {
            // remove and destroy the cached bookstore ObjectGrid
            objectGridManager.removeObjectGrid("bookstore", true);
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }
}
```

ObjectGrid 断片へのアクセス

eXtreme Scale などのアプリケーションは、データが存在する場所にロジックを移動し、結果のみをクライアントに戻すことによって、高い処理速度を実現します。

クライアント Java 仮想マシン (JVM)のアプリケーション論理では、データを保持しているサーバー JVM からデータをプルして、トランザクションがコミットされた時点でそのデータをプッシュ・バックすることが必要になります。このプロセスにより、データが処理されるレートが低下します。アプリケーション・ロジックが、データを保持している断片と同じ JVM 上にあれば、ネットワーク待ち時間およびマーシャル・コストはなくなり、パフォーマンスは大幅に向上します。

断片データへのローカル参照

ObjectGrid API には、サーバー・サイド・メソッドに対するセッションが用意されています。このセッションは、その断片のデータに対する直接のリファレンスになります。そのパスでは、ルーティング論理は存在しません。アプリケーション論理は、直接その断片のデータとともに作業できます。ルーティング論理が存在しないため、別の区画のデータにアクセスするのにセッションは使用できません。

ローダー・プラグインには、断片が 1 次区画になる場合にイベントを受信する方法が用意されています。アプリケーションは、ローダーおよび ReplicaPreloadController インターフェースを実装できます。検査プリロード状態メソッドは、断片が 1 次区画になる場合にのみ呼び出されます。そのメソッドに提供されているセッションは、断片データに対するローカル・リファレンスです。これは、区画が主に一部のスレッドを開始したり、区画に関連するトラフィックのメッセージ・ファブリックに加入したりすることを必要としている場合に、通常使用される手法です。getNextKey API を使用して、ローカル・マップ内でメッセージを listen するスレッドを開始します。

連結されたクライアント/サーバーの最適化

アプリケーションがクライアント API を使用し、そのクライアントが含まれる JVM と連結されることになる区画にアクセスする場合は、ネットワークは回避されますが、現行の実装問題のためマーシャルが発生する場合があります。区画に分割

されたグリッドが使用されている場合は、(N-1)/N 個の呼び出しが異なる JVM に送付されるため、アプリケーションのパフォーマンスに影響は与えません。常に断片を伴うローカル・アクセスが必要な場合は、ローダーまたは ObjectGrid API を使用してそのロジックを呼び出します。

セッションを使用したグリッド内データへのアクセス

アプリケーションは、Session インターフェースを介してトランザクションを開始および終了できます。Session インターフェースは、アプリケーションを基にした ObjectMap および JavaMap インターフェースへのアクセスも提供します。

各 ObjectMap または JavaMap インスタンスは、特定のセッション・オブジェクトに直接結合しています。eXtreme Scale にアクセスしたい各スレッドは、まず最初に ObjectGrid オブジェクトからセッションを取得しなければなりません。セッション・インスタンスは、スレッド間で同時に共有することはできません。 WebSphere eXtreme Scale は、スレッドのローカル・ストレージをまったく使用しませんが、プラットフォームの制約事項により、あるスレッドから別のスレッドへのセッションの受け渡しの機会が制限されることがあります。

メソッド

以下のメソッドが Session インターフェースで使用できます。以下のメソッドについての詳細は、API 資料を参照してください。

```
public interface Session {
    ObjectMap getMap(String cacheName) throws UndefinedMapException;

    void begin() throws TransactionAlreadyActiveException, TransactionException;

    void beginNoWriteThrough() throws TransactionAlreadyActiveException,
   TransactionException;

    public void commit() throws NoActiveTransactionException, TransactionException;

    public void rollback() throws NoActiveTransactionException, TransactionException;

    public void flush() throws TransactionException;

    TxID getTxID() throws NoActiveTransactionException;

    boolean isWriteThroughEnabled();

    void setTransactionType(String tranType);

    public void processLogSequence(LogSequence logSequence) throws
   NoActiveTransactionException, UndefinedMapException, ObjectGridException;

    ObjectGrid getObjectGrid();

    public void setTransactionTimeout(int timeout);
    public int getTransactionTimeout();
    public boolean transactionTimedOut();

    public boolean isCommitting();
    public boolean isFlushing();

    public void markRollbackOnly(Throwable t) throws NoActiveTransactionException;
    public boolean isMarkedRollbackOnly();
}
```

get メソッド

アプリケーションは ObjectGrid.getSession メソッドを使用して、セッション・インスタンスを ObjectGrid オブジェクトから取得します。次の例は、Session インターフェースを取得する方法を示しています。

```
ObjectGrid objectGrid = ...; Session sess = objectGrid.getSession();
```

セッションを取得した後、スレッドはそのセッションへの参照を専用に保持します。 getSession メソッドを複数回呼び出すと、その度に新規セッション・オブジェクトが戻されます。

トランザクション・メソッドとセッション・メソッド

セッションは、トランザクションの開始、コミット、またはロールバックに使用できます。 ObjectMap と JavaMap を使用した BackingMap に対する操作は、セッション・トランザクション内では非常に効率よく実行されます。トランザクションが開始された後は、そのトランザクションの有効範囲にある 1 つ以上の BackingMap に対するすべての変更は、そのトランザクションがコミットされるまで、特別のトランザクション・キャッシュに保管されます。トランザクションがコミットされると、保留になっている変更内容は BackingMap とローダーに適用され、その ObjectGrid のその他のクライアントから見えるようになります。

WebSphere eXtreme Scale は、トランザクションを自動的にコミットする機能 (自動コミットともいう) もサポートします。すべての ObjectMap オペレーションがアクティブ・トランザクションのコンテキストの外部で実行される場合は、暗黙のトランザクションはそのオペレーションの前に開始され、そのトランザクションはアプリケーションに制御が戻される前に自動的にコミットされます。

```
Session session = objectGrid.getSession();
ObjectMap objectMap = session.getMap("someMap");
session.begin();
objectMap.insert("key1", "value1");
objectMap.insert("key2", "value2");
session.commit();
objectMap.insert("key3", "value3"); // auto-commit
```

Session.flush メソッド

Session.flush メソッドは、ローダーが BackingMap に関連付けられているときにのみ意味があります。 flush メソッドは、トランザクション・キャッシュ内の変更内容の現行セットを使用してローダーを呼び出します。ローダーは、変更内容をバックエンドに適用します。これらの変更内容は、flush が呼び出されるときはコミットされません。 flush 呼び出しの後、セッション・トランザクションがコミットされると、 flush 呼び出しの後で発生する更新のみがローダーに適用されます。 flush 呼び出しの後、セッション・トランザクションがロールバックされると、フラッシュされた変更内容はトランザクション内のその他すべての保留している変更内容と一緒に廃棄されます。 flush メソッドは、ローダーに対するバッチ操作の機会を制限するので、慎重に使用してください。以下は、Session.flush メソッドの使用例です。

```
Session session = objectGrid.getSession();
session.begin();
// make some changes
...
session.flush(); // push these changes to the Loader, but don't commit yet
// make some more changes
...
session.commit();
```

NoWriteThrough メソッド

いくつかの eXtreme Scale マップはローダーによってバックアップされます。ローダーはマップ内のデータ用に永続ストレージを提供します。eXtreme Scale マップのみにデータをコミットし、ローダーにデータをプッシュアウトしないことが有益な場合があります。 Session インターフェースは、この目的のために beginNoWriteThrough メソッドを提供します。 beginNoWriteThrough メソッドは、begin メソッドのようなトランザクションを開始します。beginNoWriteThrough メソッドでは、トランザクションがコミットされると、データは eXtreme Scale のメモリー内のマップにのみコミットされ、ローダーが提供する永続ストレージにはコミットされません。このメソッドが非常に役立つのは、データがマップにプリロードされるときです。

分散 ObjectGrid インスタンスを使用する場合、サーバーで遠くのキャッシュは変更せずに近くのキャッシュのみを変更するには、beginNoWriteThrough メソッドが役立ちます。近くのキャッシュでデータが不整合であると認識されている場合は、beginNoWriteThrough メソッドを使用すると、エントリーをサーバーでは無効にせずに、近くのキャッシュで無効にすることができます。

Session インターフェースは、現在活動中のトランザクション・タイプを判別する isWriteThroughEnabled メソッドも提供します。

```
Session session = objectGrid.getSession();
session.beginNoWriteThrough();
// make some changes ...
session.commit(); // these changes will not get pushed to the Loader
```

TxID オブジェクト・メソッドの取得

TxID オブジェクトは、内部が見えないオブジェクトで、活動中のトランザクションを識別します。以下の目的には、TxID オブジェクトを使用します。

- ある特定のトランザクションを検索している場合の比較用
- TransactionCallback とローダーのオブジェクト間で共用データを保管するため

オブジェクト・スロット・フィーチャーについての追加情報は、『TransactionCallback プラグイン』と『ローダー』を参照してください。

パフォーマンス・モニター・メソッド

eXtreme Scale を WebSphere Application Server 内で使用する場合、パフォーマンス・モニタリング用にトランザクション・タイプをリセットすることが必要になることがあります。トランザクション・タイプの設定には、setTransactionType メソッドを使用できます。 setTransactionType メソッドについて詳しくは、『WebSphere Application Server Performance Monitoring Infrastructure (PMI) を使用した ObjectGrid パフォーマンスのモニター』を参照してください。

完全な LogSequence メソッドの処理

WebSphere eXtreme Scale は、ある Java 仮想マシンから別のマシンへマップを配布する手段として、マップ変更セットを ObjectGrid リスナーに伝搬できます。リスナーが受信済み LogSequences を処理するのを容易にするために、 Session インターフェースは processLogSequence メソッドを提供します。このメソッドは LogSequence 内で 各 LogElement を検査し、 LogSequence MapName によって識別される BackingMap に対して適切なオペレーション (例えば、挿入、更新、無効化

など) を実行します。 ObjectGrid セッションは、processLogSequence メソッドが呼び出される前に使用可能になっていなければなりません。アプリケーションは、セッションを完了するために適切な commit または rollback 呼び出しを実行する役割があります。自動コミット処理は、このメソッド呼び出しには使用できません。リモート JVM での受信側 ObjectGridEventListener による通常の処理では、この processLogSequence メソッドの呼び出しが続く beginNoWriteThrough メソッド (変更内容のエンドレスな伝搬を防止するもの) を使用し、次にトランザクションをコミットまたはロールバックすることで、セッションを開始することになります。

```
// Use the Session object that was passed in during
//ObjectGridEventListener.initialization...
session.beginNoWriteThrough();
// process the received LogSequence
try {
 session.processLogSequence(receivedLogSequence);
} catch (Exception e) {
 session.rollback(); throw e;
}
// commit the changes
session.commit();
```

markRollbackOnly メソッド

このメソッドを使用して、現行トランザクションに「rollback only」とマークを付けます。トランザクションに「rollback only」とマークを付けると、アプリケーションで commit メソッドが呼び出された場合でも、トランザクションはロールバックされます。このメソッドは、通常、トランザクションのコミットが許可されている場合にデータ破壊が発生する可能性があると認識されているとき、ObjectGrid 自体またはアプリケーションで使用されます。このメソッドが呼び出されると、このメソッドに渡される Throwable オブジェクトが com.ibm.websphere.objectgrid.TransactionException 例外にチェーニングされます。この例外は、以前に「rollback only」とマーク付けされたセッションで commit メソッドが呼び出された場合の結果です。既に「rollback only」とマーク付けされているトランザクションのこのメソッドに対する以降の呼び出しは、無視されます。つまり、ヌル以外の Throwable 参照を渡す最初の呼び出しのみが使用されます。マークされたトランザクションが完了すると、「rollback only」マークは除去されるため、セッションで開始される次のトランザクションはコミットされます。

isMarkedRollbackOnly メソッド

セッションが現在「rollback only」とマークされている場合に戻されます。markRollbackOnly メソッドが以前このセッションで呼び出されており、セッションで開始されたトランザクションがアクティブな場合、かつこの場合に限り、このメソッドによってブール値 true が戻されます。

setTransactionTimeout メソッド

このセッションで開始される次のトランザクションのトランザクション・タイムアウトを特定の秒数に設定します。このメソッドは、このセッションで以前に開始されたトランザクションのトランザクション・タイムアウトには影響を与えません。このメソッドが呼び出された後に開始されたトランザクションにのみ影響を与えます。このメソッドが呼び出されない場合は、com.ibm.websphere.objectgrid.ObjectGrid メソッドの setTxTimeout メソッドに渡されたタイムアウト値が使用されます。

getTransactionTimeout メソッド

このメソッドは、トランザクション・タイムアウト値 (秒単位) を戻します。タイムアウト値として setTransactionTimeout メソッドに渡された最後の値は、このメソッドによって戻されます。 setTransactionTimeout メソッドが呼び出されない場合は、com.ibm.websphere.objectgrid.ObjectGrid メソッドの setTxTimeout メソッドに渡されたタイムアウト値が使用されます。

transactionTimedOut

このメソッドは、このセッションで開始された現行トランザクションがタイムアウトになると、ブール値 true を戻します。

isFlushing メソッド

このメソッドは、呼び出されたセッション・インターフェースの flush メソッドの結果として、すべてのトランザクション変更がローダー・プラグインにフラッシュされる場合、かつこの場合に限り、ブール値 true を戻します。ローダー・プラグインでは、batchUpdate メソッドが呼び出された理由を確認する必要がある場合にこのメソッドが役立ちます。

isCommitting メソッド

このメソッドは、呼び出されたセッション・インターフェースの commit メソッドの結果として、すべてのトランザクション変更がコミットされる場合、かつこの場合に限り、ブール値 true を戻します。ローダー・プラグインでは、batchUpdate メソッドが呼び出された理由を確認する必要がある場合にこのメソッドが役立ちます。

setRequestRetryTimeout メソッド

このメソッドは、セッションの要求再試行タイムアウト値 (ミリ秒) を設定します。クライアントが要求再試行タイムアウトを設定してある場合、セッション設定値がクライアント値をオーバーライドします。

getRequestRetryTimeout メソッド

このメソッドは、セッションの現行の要求再試行タイムアウト設定を取得します。値 `-1` は、タイムアウトが設定されていないことを表します。値 `0` は、フェイル・ファースト・モードであることを表します。`0` より大きい値は、ミリ秒単位のタイムアウト設定値です。

ロックの処理

ロックにはライフサイクルがあり、さまざまな種類のロックはさまざまな方法で互いに互換性を持ちます。ロックはデッドロック・シナリオにならないように、正しい順序で処理する必要があります。

ロックのライフサイクル

ロックのタイムアウト 各 BackingMap には、デフォルトのロック待ちタイムアウト値があります。タイムアウト値は、アプリケーション・エラーによりデッドロック

条件が発生したために、アプリケーションがロック・モードを認可されるのをいつまでも待つことがないように使用します。アプリケーションは、BackingMap インターフェースを使用して、デフォルトのロック待ちタイムアウト値をオーバーライドすることができます。以下の例は、map1 バックアップ・マップのロック待ちタイムアウト値を 60 秒に設定する方法を示しています。

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
...
ObjectGrid og =
 ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("map1");
bm.setLockStrategy( LockStrategy.PESSIMISTIC );
bm.setLockTimeout( 60 );
```

java.lang.IllegalStateException 例外を回避するには、ObjectGrid インスタンスの initialize または getSession メソッドのいずれかを呼び出す前に、setLockStrategy メソッドと setLockTimeout メソッドの両方を呼び出します。setLockTimeout メソッドのパラメーターは、Java プリミティブの整数で、eXtreme Scale がロック・モードを認可されるのを待たなければならない秒数を指定します。 BackingMap に構成されているロック待ちタイムアウト値よりも長くトランザクションが待つ場合は、com.ibm.websphere.objectgrid.LockTimeoutException 例外が発生します。

LockTimeoutException が発生したら、アプリケーションは、アプリケーションの実行が予想よりも遅くなっているためにタイムアウトが発生しているのか、それともデッドロック条件のためにタイムアウトが発生したのかを判別しなければなりません。実際にデッドロック条件が発生した場合は、ロック待ちタイムアウト値を増やしても例外は除去されません。タイムアウト値を増やすと、例外の発生期間が長くなります。しかし、ロック待ちタイムアウト値を増やして例外を除去している場合は、アプリケーションが予想よりも低速で実行されるために問題が発生しました。このケースのアプリケーションでは、パフォーマンスの低下原因を判別しなければなりません。

ObjectMaps のロック待ちタイムアウト

ロック待ちタイムアウトは、ObjectMap.setLockTimeout メソッドを使用すれば、単一の ObjectMap インスタンスに対してオーバーライドできます。ロック・タイムアウト値は、新規のタイムアウト値の設定後に開始されたすべてのトランザクションに影響します。このメソッドは、ロック競合が選択トランザクションで起こりうる、あるいは予想される場合に便利です。

共用ロック、アップグレード可能ロック、および排他的ロック

アプリケーションが ObjectMap インターフェースのいずれかのメソッドを呼び出すか、索引に対して検索メソッドを使用するか、照会を行うと、eXtreme Scale は、アクセスするマップ・エントリーに対して自動的にロックを獲得しようとします。WebSphere eXtreme Scale は、アプリケーションが ObjectMap インターフェース内で呼び出すメソッドを基にした以下のロック・モードを使用します。

- ObjectMap インターフェース上の get と getAll メソッド、索引メソッド、および照会は、マップ・エントリーのキーに対する *S ロック*、つまり共用ロック・モードを獲得します。S ロックが保持されている期間は、使用されるトランザクション分離レベルによります。S ロック・モードでは、同一キーに対して S ロック・モードまたはアップグレード可能ロック (U ロック) モードを獲得しようと

するトランザクション間での並行処理が許されますが、同一キーに対して排他的ロック (X ロック) モードを取得しようとする他のトランザクションはブロックされます。

- getForUpdate および getAllForUpdate メソッドは、マップ・エントリーのキーに対する *U ロック*、つまりアップグレード可能ロック・モードを獲得します。 U ロックは、トランザクションが完了するまで保留になります。U ロック・モードでは、同一キーに対して S ロック・モードを獲得するトランザクション間での並行処理が許されますが、同一キーに対して U ロック・モードまたは X ロック・モードを獲得しようとする他のトランザクションはブロックされます。
- put、putAll、remove、removeAll、insert、update、および touch は、マップ・エントリーのキーに対する *X ロック*、つまり排他的ロック・モードを獲得します。 X ロックは、トランザクションが完了するまで保留になります。 X ロック・モードでは、1 つのトランザクションのみが所定のキー値のマップ・エントリーを挿入、更新、または除去することになります。 X ロックは、同一キーに対する S、U、または X ロック・モードを獲得しようとする他のすべてのトランザクションをブロックします。
- global invalidate および global invalidateAll メソッドは、無効化されている各マップ・エントリーに対する X ロックを獲得します。 X ロックは、トランザクションが完了するまで保留になります。local invalidate および local invalidateAll メソッドはロックを獲得しません。local invalidate メソッドの呼び出しによって無効化される BackingMap エントリーがないためです。

前の定義から、S ロック・モードが U ロック・モードよりも弱体であることは明白です。それは、同一マップ・エントリーにアクセスするときに、より多くのトランザクションが並行して実行されることを許すためです。 U ロック・モードは、S ロック・モードよりも少し強力です。それは、U ロック・モードまたは X ロック・モードのどちらかを要求している他のトランザクションをブロックするためです。S ロック・モードは、X ロック・モードを要求しているその他のトランザクションのみをブロックします。この小さな差が、一部のデッドロックの発生を防止するには重要です。 X ロック・モードは、最強のロック・モードです。これは、同一のマップ・エントリーに対して S、U、または X ロックのモードを取得しようとしているその他すべてのトランザクションをブロックするためにです。 X ロック・モードの最終的な効果は、1 つのトランザクションのみがマップ・エントリーを挿入、更新、または除去できるようにすることと、複数のトランザクションが同一のマップ・エントリーを更新しようとしているときに、更新が失われないようにすることです。

次表は、ロック・モードの互換性マトリックスです。前述のロック・モードをまとめたもので、互いに互換性のあるロック・モードはいずれかを調べる場合に使用してください。このマトリックスを読み取る場合、マトリックスの行は既に認可されているロック・モードを表します。列は、別のトランザクションによって要求されたロック・モードを表します。列に「あり」と表示されている場合は、別のトランザクションによって要求されたロック・モードは認可されています。これは、既に認可されているロック・モードと互換性があるためです。「なし」は、ロック・モードの互換性がないことを表します。その他のトランザクションは、最初のトランザクションが保持しているロックを解放するのを待たなければなりません。

表 1. ロック・モードの互換性マトリックス

ロック	ロック・タイプ S (共用)	ロック・タイプ U (アップグレード可能)	ロック・タイプ X (排他的)	強さ
S (共用)	あり	あり	なし	最弱
U (アップグレード可能)	あり	なし	なし	通常
X (排他的)	なし	なし	なし	最強

ロックのデッドロック

ロック・モード要求の以下のシーケンスについて検討します。

1. X ロックは、トランザクション 1 の key1 に対して認可されています。
2. X ロックは、トランザクション 2 の key2 に対して認可されています。
3. トランザクション 1 によって要求された、key2 に対する X ロック (トランザクション 1 は、トランザクション 2 によって所有されたロックを待機するのをブロックします。)
4. トランザクション 2 によって要求された、key1 に対する X ロック (トランザクション 2 は、トランザクション 1 によって所有されたロックを待機するのをブロックします。)

上記のシーケンスは、2 つのトランザクションからなる古典的なデッドロックの例です。 2 つのトランザクションが複数のロックを獲得しようとし、各トランザクションは異なる順序でロック獲得します。このデッドロックを防止するには、各トランザクションが複数ロックを同じ順序で獲得しなければなりません。オプティミスティック・ロック・ストラテジーが使用され、 ObjectMap インターフェースの flush メソッドがアプリケーションによって絶対に使用されない場合は、ロック・モードがトランザクションによって要求されるのはコミット・サイクル中のみです。コミット・サイクル中、eXtreme Scale は、ロックする必要があるマップ・エントリーのキーを決定し、キー・シーケンスのロック・モードを要求します (決定論的振る舞い)。この方法で、eXtreme Scale は古典的デッドロックの大多数を防止します。しかし、eXtreme Scale がすべてのデッドロック・シナリオを防止するわけでも、防止できるわけでもありません。アプリケーションが考慮する必要があるシナリオがいくつか存在します。以下は、アプリケーションが注意し、予防アクションを取らなければならないシナリオです。

1 つのシナリオは、ロック待ちタイムアウトが発生するのを待たなくとも eXtreme Scale がデッドロックを検出できる場合です。このシナリオが起こる場合、com.ibm.websphere.objectgrid.LockDeadlockException 例外が発生します。以下のコード・スニペットについて検討します。

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
sess.begin();
Person p = (IPerson)person.get("Lynn");
// Lynn had a birthday, so we make her 1 year older.
p.setAge( p.getAge() + 1 );
person.put( "Lynn", p );
sess.commit();
```

この状況では、Lynn の知人は Lynn の年齢を加算しようとするので、Lynn とその知人が同時にこのトランザクションを実行します。この状態では、person.get(″Lynn″)

メソッド呼び出しの結果として両方のトランザクションが PERSON マップの Lynn エントリーに対して S ロック・モードを保持します。person.put (″Lynn″, p) メソッド呼び出しの結果として、両方のトランザクションは S ロック・モードを X ロック・モードに格上げしようとします。両方のトランザクションは、他方のトランザクションが所有している S ロック・モードを解放するのを待つことをブロックします。結果として、デッドロックが発生します。 2 つのトランザクション間に循環待ち条件が存在するためです。循環待ち条件は、複数のトランザクションが同一のマップ・エントリーに対して弱いモードから強いモードへロックを格上げするときに発生します。このシナリオでは、LockTimeoutException 例外ではなく、LockDeadlockException 例外になります。

アプリケーションは、ペシミスティック・ロック・ストラテジーではなく、オプティミスティック・ロック・ストラテジーを使用すれば、前例の LockDeadlockException 例外を防止できます。オプティミスティック・ロック・ストラテジーの使用は、マップが主として読み取りで、マップの更新がまれにしか行われない場合、推奨される解決策です。ペシミスティック・ロック・ストラテジーを使用する必要がある場合は、上記の例の get メソッドの代わりに、 getForUpdate メソッドを使用するか、TRANSACTION_READ_COMMITTED のトランザクション分離レベルを使用する方法があります。

詳しくは、*製品概説* のロック・ストラテジーに関するトピックを参照してください。

TRANSACTION_READ_COMMITTED トランザクション分離レベルを使用すると、通常、get メソッドによって獲得される S ロックは、トランザクション完了まで保持されることがなくなります。キーがトランザクション・キャッシュで無効化されない場合、反復可能読み取りは引き続き保証されます。

詳しくは、*管理ガイド*のマップ・エントリーのロックに関するトピックを参照してください。

トランザクション分離レベルを変更する方法の代替方法が、getForUpdate メソッドの使用です。getForUpdate メソッドを呼び出す最初のトランザクションは、S ロックではなく U ロック・モードを獲得します。このロック・モードにより、2 番目のトランザクションは、getForUpdate メソッドを呼び出したときにブロックされます。U ロック・モードで認可されるトランザクションは 1 つのみだからです。2 番目のトランザクションはブロックされるので、Lynn マップ・エントリーに対するロック・モードを何も所有しません。最初のトランザクションは、最初のトランザクションからの put メソッド呼び出しの結果として、 U ロック・モードから X ロック・モードへの格上げをしようとしたときに、ブロックしません。この働きは、U ロック・モードが*アップグレード可能* ロック・モードと呼ばれる理由を説明しています。最初のトランザクションが完了すると、2 番目のトランザクションがブロックを解除し、U ロック・モードを認可されます。アプリケーションは、ペシミスティック・ロック・ストラテジーが使用されている場合、 get メソッドの代わりに getForUpdate メソッドを使用することにより、ロック格上げによるデッドロック・シナリオを回避できます。

重要: この解決策は、読み取り専用トランザクションがマップ・エントリーを読み取るのを妨げません。読み取り専用トランザクションは、get メソッドを呼び出しますが、put、insert、update、または remove メソッドを呼び出すことはありません。

並行性は、通常の get メソッドが使用されているときと同様に高く維持されます。唯一、並行性が低減するのは、複数のトランザクションによって同一のマップ・エントリーに対して getForUpdate メソッドが呼び出されるときです。

あるトランザクションが複数のマップ・エントリーに対して getForUpdate メソッドを呼び出す場合、各トランザクションによって確実に U ロックが同じ順序で獲得されるように注意しなければなりません。例えば、最初のトランザクションがキー 1 に対する getForUpdate メソッドと、キー 2 に対する getForUpdate メソッドを呼び出すとします。別の並行トランザクションが 2 つの同一キーに対する getForUpdate メソッドを呼び出しますが、逆順で呼び出します。このシーケンスにより、古典的なデッドロックが発生します。複数ロックが異なるトランザクションによって異なる順序で獲得されるためです。アプリケーションでは引き続き、複数のマップ・エントリーにアクセスするどのトランザクションもキー・シーケンスに従い、デッドロックを発生しないようにする必要があります。 U ロックはコミット時ではなく、getForUpdate メソッドが呼び出される時に獲得されるので、eXtreme Scale は、コミット・サイクル中に行われるようにロック要求を順序付けることはできません。アプリケーションは、このケースではロックの順序付けを制御する必要があります。

コミットの前に ObjectMap インターフェースの flush メソッドを使用すれば、ロックの順序付けの考慮を加えることができます。 flush メソッドは、通常、ローダー・プラグインにより、マップに行われた変更をバックエンドに強制する目的に使用されます。この状態では、バックエンドは独自のロック・マネージャーを使用して並行処理を制御するので、ロック待ち条件とデッドロックは、 eXtreme Scale ロック・マネージャー内よりもむしろバックエンド内で発生します。次のトランザクションについて検討します。

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
boolean activeTran = false;
try
{
    sess.begin();
    activeTran = true;
    Person p = (IPerson)person.get("Lynn");
    p.setAge( p.getAge() + 1 );
    person.put( "Lynn", p );
    person.flush();
    ...
    p = (IPerson)person.get("Tom");
    p.setAge( p.getAge() + 1 );
    sess.commit();
    activeTran = false;
}
finally
{
    if ( activeTran ) sess.rollback();
}
```

何かほかのトランザクションが Tom も更新し、flush メソッドを呼び出し、次に Lynn を更新したとします。この状態が発生した場合、2 つのトランザクションの以下のインターリービングの結果、データベースはデッドロック状態になります。

```
flush の実行時に "Lynn" のトランザクション 1 に対して X ロックが認可されます。
flush の実行時に "Tom" のトランザクション 2 に対して X ロックが認可されます。
コミット処理中に "Tom" のトランザクション 1 によって、X ロックが要求されます。
(トランザクション 1 は、
トランザクション 2 によって所有されたロックを待機するのをブロックします。)
コミット処理中に "Lynn" のトランザクション 2 によって、X ロックが要求されます。
(トランザクション 2 は、
トランザクション 1 によって所有されたロックを待機するのをブロックします。)
```

この例は、flush メソッドの使用により、eXtreme Scale 内ではなくデータベース内でデッドロックが発生することを示しています。このデッドロック例は、どのロック・ストラテジーを使用しても発生する可能性があります。アプリケーションは、

flush メソッドを使用しているときと、 Loader が BackingMap にプラグインされているときは、この種のデッドロックの発生を防止することに留意する必要があります。上記の例は、eXtreme Scale がロック待ちタイムアウト機構を備えているもう 1 つの理由を示しています。データベース・ロックを待機するトランザクションは、eXtreme Scale マップ・エントリーのロックを所有している間、待機し続ける可能性があります。その結果、データベース・レベルの問題により、eXtreme Scale ロック・モードの待機時間が過大になり、LockTimeoutException 例外が発生する可能性があります。

一般的なデッドロック・シナリオ

以下のセクションでは、いくつかの最も一般的なデッドロック・シナリオを説明し、その回避方法を提示します。

シナリオ: 単一キーのデッドロック

以下のシナリオでは、S ロックを使用して単一キーにアクセスし、その後、更新する場合にデッドロックがどのように発生するかを示しています。これが 2 つのトランザクションから同時に発生すると、デッドロックになります。

表 2. 単一キーのデッドロックのシナリオ

	スレッド 1	スレッド 2	
1	session.begin()	session.begin()	各スレッドが独立したトランザクションを確立します。
2	map.get(key1)	map.get(key1)	key1 に対して S ロックが両方のトランザクションに認可されます。
3	map.update(Key1,v)		U ロックはありません。更新はトランザクション・キャッシュで実行されます。
4		map.update(key1,v)	U ロックはありません。更新はトランザクション・キャッシュで実行されます。
5	session.commit()		ブロックされます。スレッド 2 が S ロックを保有しているため、key1 に対する S ロックは X ロックにアップグレードできません。
6		session.commit()	デッドロック: T1 が S ロックを保有しているため、key1 に対する S ロックは X ロックにアップグレードできません。

表 3. 単一キーのデッドロック (続き)

	スレッド 1	スレッド 2	
1	session.begin()	session.begin()	各スレッドが独立したトランザクションを確立します。
2	map.get(key1)		key1 に対して S ロックが認可されます。
3	map.getForUpdate(key1,v)		key1 に対して S ロックが U ロックにアップグレードされます。

表 3. 単一キーのデッドロック (続き) (続き)

	スレッド 1	スレッド 2	
4		map.get(key1)	key1 に対して S ロックが認可されます。
5		map.getForUpdate(key1,v)	ブロックされます。T1 が既に U ロックを保有しています。
6	session.commit()		デッドロック: key1 に対する U ロックはアップグレードできません。
7		session.commit()	デッドロック: key1 に対する S ロックはアップグレードできません。

表 4. 単一キーのデッドロック (続き)

	スレッド 1	スレッド 2	
1	session.begin()	session.begin()	各スレッドが独立したトランザクションを確立します。
2	map.get(key1)		key1 に対して S ロックが認可されます。
3	map.getForUpdate(key1,v)		key1 に対して S ロックが U ロックにアップグレードされます。
4		map.get(key1)	key1 に対して S ロックが認可されます。
5		map.getForUpdate(key1,v)	ブロックされます。スレッド 1 が既に U ロックを保有しています。
6	session.commit()		デッドロック: スレッド 2 が S ロックを保有しているため、key1 に対する U ロックは X ロックにアップグレードできません。

ObjectMap.getForUpdate を使用して S ロックを回避すれば、デッドロックは回避されます。

表 5. 単一キーのデッドロック (続き)

	スレッド 1	スレッド 2	
1	session.begin()	session.begin()	各スレッドが独立したトランザクションを確立します。
2	map.getForUpdate(key1)		key1 のスレッド 1 に対して U ロックが認可されます。
3		map.getForUpdate(key1)	U ロック要求がブロックされます。
4	map.update(key1,v)	<blocked>	
5	session.commit()	<blocked>	key1 に対する U ロックは正常に X ロックにアップグレードできます。
6		<released>	スレッド 2 に対して U ロックが最終的に key1 に認可されます。

表 5. 単一キーのデッドロック (続き) (続き)

	スレッド 1	スレッド 2	
7		map.update(key2,v)	key2 に対して U ロックがスレッド 2 に認可されます。
8		session.commit()	key1 に対する U ロックは正常に X ロックにアップグレードできます。

解決策

1. get ではなく getForUpdate メソッドを使用し、S ロックではなく U ロックを獲得します。
2. 読み取りコミット済みのトランザクション分離レベルを使用し、S ロックの保有を回避します。トランザクション分離レベルを下げると、非反復可能読み取りの可能性が増します。しかし、非反復可能読み取りが起こりうるのは、トランザクション・キャッシュが明示的に無効化された場合に限られます。
3. オプティミスティック・ロック・ストラテジーを使用します。オプティミスティック・ロック・ストラテジーを使用するには、オプティミスティック競合例外を処理する必要があります。

シナリオ: 順序付けされた複数のキーのデッドロック

このシナリオでは、2 つのトランザクションが同一のエントリーを直接更新しようとし、他のエントリーに対して S ロックを保有するとどうなるかを説明します。

表 6. 順序付けされた複数のキーのデッドロックのシナリオ

	スレッド 1	スレッド 2	
1	session.begin()	session.begin()	各スレッドが独立したトランザクションを確立します。
2	map.get(key1)	map.get(key1)	key1 に対して S ロックが両方のトランザクションに認可されます。
3	map.get(key2)	map.get(key2)	key2 に対して S ロックが両方のトランザクションに認可されます。
4	map.update(key1,v)		U ロックはありません。更新はトランザクション・キャッシュで実行されます。
5		map.update(key2,v)	U ロックはありません。更新はトランザクション・キャッシュで実行されます。
6.	session.commit()		ブロックされます。スレッド 2 が S ロックを保有しているため、key1 に対する S ロックは X ロックにアップグレードできません。
7		session.commit()	デッドロック: スレッド 1 が S ロックを保有しているため、key2 に対する S ロックはアップグレードできません。

ObjectMap.getForUpdate メソッドを使用して、S ロックを回避すれば、デッドロックを回避できます。

表 7. 順序付けされた複数のキーのデッドロックのシナリオ (続き)

	スレッド 1	スレッド 2	
1	session.begin()	session.begin()	各スレッドが独立したトランザクションを確立します。
2	map.getForUpdate(key1)		key1 に対して U ロックがトランザクション T1 に認可されます。
3		map.getForUpdate(key1)	U ロック要求がブロックされます。
4	map.get(key2)	<blocked>	key2 に対して S ロックが T1 に認可されます。
5	map.update(key1,v)	<blocked>	
6	session.commit()	<blocked>	key1 に対する U ロックは正常に X ロックにアップグレードできます。
7		<released>	T2 に対して U ロックが最終的に key1 に認可されます。
8		map.get(key2)	key2 に対して S ロックが T2 に認可されます。
9		map.update(key2,v)	key2 に対して U ロックが T2 に認可されます。
10		session.commit()	key1 に対する U ロックは正常に X ロックにアップグレードできます。

解決策

1. get メソッドではなく getForUpdate を使用して、最初のキーに対して直接 U ロックを獲得します。このストラテジーが機能するのは、メソッド順序が決定論的な場合に限られます。
2. 読み取りコミット済みのトランザクション分離レベルを使用し、S ロックの保有を回避します。この解決策は、メソッド順序が決定論的でない場合に、最も簡単に実装できます。トランザクション分離レベルを下げると、非反復可能読み取りの可能性が増します。しかし、非反復可能読み取りが起こりうるのは、トランザクション・キャッシュが明示的に無効化された場合に限られます。
3. オプティミスティック・ロック・ストラテジーを使用します。オプティミスティック・ロック・ストラテジーを使用するには、オプティミスティック競合例外を処理する必要があります。

シナリオ: U ロックで順序付けがない

キーが要求される順序が保証できない場合でも、デッドロックは起きる可能性があります。

表 8. U ロックで順序付けがないシナリオ

	スレッド 1	スレッド 2	
1	session.begin()	session.begin()	各スレッドが独立したトランザクションを確立します。
2	map.getforUpdate(key1)	map.getForUpdate(key2)	key1 と key2 に対して U ロックが正常に認可されます。

表 8. U ロックで順序付けがないシナリオ (続き)

	スレッド 1	スレッド 2	
3	map.get(key2)	map.get(key1)	key1 と key2 に対して S ロックが認可されます。
4	map.update(key1,v)	map.update(key2,v)	
5	session.commit()		T2 が S ロックを保有しているため、U ロックは X ロックにアップグレードできません。
6		session.commit()	T1 が S ロックを保有しているため、U ロックは X ロックにアップグレードできません。

解決策

1. すべての作業を単一のグローバル U ロックでラップします (mutex)。この方法は、並行性を低下させますが、アクセスおよび順序が決定論的でない場合に、すべてのシナリオを処理できます。
2. 読み取りコミット済みのトランザクション分離レベルを使用し、S ロックの保有を回避します。この解決策は、メソッド順序が決定論的でない場合に、最も簡単に実装でき、最大の並行性を提供します。トランザクション分離レベルを下げると、非反復可能読み取りの可能性が増します。しかし、非反復可能読み取りが起こりうるのは、トランザクション・キャッシュが明示的に無効化された場合に限られます。
3. オプティミスティック・ロック・ストラテジーを使用します。オプティミスティック・ロック・ストラテジーを使用するには、オプティミスティック競合例外を処理する必要があります。

ロック・シナリオにおける例外処理

前記の例には、例外処理が含まれていません。LockTimeoutException 例外または LockDeadlockException 例外が発生したときに、ロックが過度に長い時間保留されないようにするために、アプリケーションは、予期しない例外をキャッチし、予期しないことが発生したときに rollback メソッドを呼び出す必要があります。以下の例に示すように、前述のコード・スニペットを変更してください。

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
boolean activeTran = false;
try
{
    sess.begin();
    activeTran = true;
    Person p = (IPerson)person.get("Lynn");
    // Lynn had a birthday, so we make her 1 year older.
    p.setAge( p.getAge() + 1 );
    person.put( "Lynn", p );
    sess.commit();
    activeTran = false;
}
finally
{
    if ( activeTran ) sess.rollback();
}
```

コード・スニペットの finally ブロックは、予期しない例外が発生したときにトランザクションがロールバックされるようにしています。LockDeadlockException 例外のみでなく、発生する可能性のあるその他の予期しない例外もすべて処理します。finally ブロックは、commit メソッドの呼び出し時に例外が発生するケースも処理します。この例は、予期しない例外を処理する唯一の方法ではありません。アプリケ

ーションが、発生する予期しない例外のいくつかをキャッチし、そのアプリケーション例外の 1 つを表示するケースも存在するかもしれません。適宜 catch ブロックを追加できますが、アプリケーションは、コード・スニペットがトランザクションを完了せずに終了しないようにする必要があります。

トランザクション分離

トランザクションに関して、各バックアップ・マップ構成を、pessimistic、optimistic、または none の 3 種類のロック・ストラテジーのうちの 1 つで構成できます。 pessimistic ロックおよび optimistic ロックを使用する場合、eXtreme Scale は共用 (S) ロック、アップグレード可能 (U) ロック、および排他的 (X) ロックを使用して、整合性を維持します。 optimistic ロックは保持されないため、このロック動作が最も目立つのは pessimistic ロックを使用しているときです。3 つのトランザクション分離レベル (反復可能読み取り、読み取りコミット済み、および読み取りアンコミット) のうちの 1 つを使用して、各キャッシュ・マップ内で eXtreme Scale が整合性を保持するために使用するロック・セマンティクスを調整することができます。

トランザクション分離の概説

トランザクション分離は、1 つの操作で行われた変更がどのように他の並行操作に可視になるのかを定義します。

WebSphere eXtreme Scale でサポートされている 3 つのトランザクション分離レベル (反復可能読み取り、読み取りコミット済み、および読み取りアンコミット) を利用して、eXtreme Scale が各キャッシュ・マップ内での整合性を保持するために使用するロック・セマンティクスをさらに調整できます。トランザクション分離レベルは、setTransactionIsolation メソッドを使用して Session インターフェースに設定されます。トランザクション分離は、現在進行中のトランザクションがなければ、セッションの存続期間中いつでも変更できます。

この製品では、共用 (S) ロックが要求および保持される方法を調整することによって、さまざまなトランザクション分離セマンティクスが施行されます。トランザクション分離は、オプティミスティック・ロックまたはロックなしストラテジーを使用するように構成されたマップに対して、あるいはアップグレード可能 (U) ロックが獲得される場合は何の影響もありません。

ペシミスティック・ロックでの反復可能読み取り

反復可能読み取りは、デフォルトのトランザクション分離レベルです。この分離レベルは、ダーティー読み取りおよび反復不能読み取りを防止しますが、ファントム読み取りは防止しません。ダーティー読み取りとは、あるトランザクションによって変更されたが、コミットされていないという状態のデータに対して発生する読み取り操作のことです。反復不能読み取りは、読み取り操作実行時に読み取りロックが獲得されていない場合に発生する可能性があります。ファントム読み取りは、2 つの同一の読み取り操作が実行されたが、操作と操作との間にデータに対する更新があったために 2 つの異なる結果セットが戻される場合に、発生する可能性があります。この製品は、すべての S ロックを、ロックを所有するトランザクションが完了するまで保持し続けることによって、反復可能読み取りを実現します。 X ロック

は、すべての S ロックが解放されるまで認可されないため、S ロックを保持するすべてのトランザクションは、再読み取り時に同じ値を参照することが保証されます。

```
map = session.getMap("Order");
session.setTransactionIsolation(Session.TRANSACTION_REPEATABLE_READ);
session.begin();

// An S lock is requested and held and the value is copied into
// the transactional cache.
Order order = (Order) map.get("100");
// The entry is evicted from the transactional cache.
map.invalidate("100", false);

// The same value is requested again.  It already holds the
// lock, so the same value is retrieved and copied into the
// transactional cache.
Order order2 (Order) = map.get("100");

// All locks are released after the transaction is synchronized
// with cache map.
session.commit();
```

ファントム読み取りが可能なのは、照会または索引を使用しているときです。なぜなら、ロックはデータ範囲に対して獲得されるのではなく、索引または照会基準に一致するキャッシュ・エントリーに対してのみ獲得されるからです。以下に例を示します。

```
session1.setTransactionIsolation(Session.TRANSACTION_REPEATABLE_READ);
session1.begin();

// A query is run which selects a range of values.
ObjectQuery query = session1.createObjectQuery
    ("SELECT o FROM Order o WHERE o.itemName='Widget'");

// In this case, only one order matches the query filter.
// The order has a key of "100".
// The query engine automatically acquires an S lock for Order "100".
Iterator result = query.getResultIterator();

// A second transaction inserts an order that also matches the query.
Map orderMap = session2.getMap("Order");
orderMap.insert("101", new Order("101", "Widget"));

// When the query runs again in the current transaction, the
// new order is visible and will return both Orders "100" and "101".
result = query.getResultIterator();

// All locks are released after the transaction is synchronized
// with cache map.
session.commit();
```

ペシミスティック・ロックでの読み取りコミット済み

読み取りコミット済みのトランザクション分離レベルを eXtreme Scale で使用できます。この分離レベルは、ダーティー読み取りを防止しますが、反復不能読み取りまたはファントム読み取りを防止しないため、eXtreme Scale は S ロックを引き続き使用してキャッシュ・マップからデータを読み取りますが、すぐにロックを解放します。

```
map1 = session1.getMap("Order");
session1.setTransactionIsolation(Session.TRANSACTION_READ_COMMITTED);
session1.begin();
```

```
// An S lock is requested but immediately released and
//the value is copied into the transactional cache.

Order order = (Order) map1.get("100");

// The entry is evicted from the transactional cache.
map1.invalidate("100", false);

// A second transaction updates the same order.
// It acquires a U lock, updates the value, and commits.
// The ObjectGrid successfully acquires the X lock during
// commit since the first transaction is using read
// committed isolation.

Map orderMap2 = session2.getMap("Order");
session2.begin();
order2 = (Order) orderMap2.getForUpdate("100");
order2.quantity=2;
orderMap2.update("100", order2);
session2.commit();

// The same value is requested again.  This time, they
// want to update the value, but it now reflects
// the new value
Order order1Copy (Order) = map1.getForUpdate("100");
```

ペシミスティック・ロックでの読み取りアンコミット

読み取りアンコミットのトランザクション分離レベルを eXtreme Scale で使用できます。この分離レベルは、ダーティー読み取り、反復不能読み取り、およびファントム読み取りを許容します。

ルーティング用の SessionHandle

コンテナーごとの区画配置のポリシーを使用している場合、SessionHandle を使用することができます。SessionHandle インスタンスは現行セッションの区画情報を含んでいて、新規セッションに再使用することができます。

SessionHandle は、現行セッションが結合されている区画の情報を保有しています。SessionHandle は、コンテナーごとの区画配置のポリシーを使用している場合に特に有用であり、標準 Java シリアライゼーションでシリアライズできます。

SessionHandle インスタンスがあれば、setSessionHandle(SessionHandle target) メソッドを使用し、そのハンドルをターゲットとして渡すことで、そのハンドルをセッションに適用できます。 SessionHandle は、Session.getSessionHandle メソッドを使用して取得できます。

これはコンテナーごとの配置のシナリオにのみ有効なので、指定された ObjectGrid がコンテナー当たり複数のマップ・セットを保有していたり、まったく保有していない場合は、SessionHandle を取得しようとすると IllegalStateException がスローされます。 setSessionHandle メソッドを前もって呼び出さずに、getSessionHandle メソッドを呼び出した場合、 ClientProperties 構成に基づいて、適切な SessionHandle が選択されます。

helper クラス SessionHandleTransformer を使用して、ハンドルをさまざまなフォーマットに変換することもできます。このクラスのメソッドは、ハンドルの表現を、

バイト配列からインスタンスに、ストリングからインスタンスに変換でき、これらの逆方向にも変換できます。さらに、ハンドルの内容を出力ストリームに書き込むこともできます。

SessionHandle の使用例については、「*製品概要*」に記載されている優先ゾーン・ルーティングに関するトピックを参照してください。

オプティミスティック衝突例外

OptimisticCollisionException は、直接受け取るか、ObjectGridException 例外と一緒に受け取ることができます。

以下のコードは、例外を catch し、そのメッセージを表示する方法の例です。

```
try {
...
} catch (ObjectGridException oe) {
    System.out.println(oe);
}
```

例外の原因

OptimisticCollisionException は、ほとんど同じ時間に 2 つの異なるクライアントが同じマップ・エントリーを更新しようとしたとき作成されます。例えば、あるクライアントがセッションをコミットしてマップ・エントリーを更新しようとした場合に、そのコミットの直前に別のクライアントがデータを読み取っていたとすると、そのデータは正しくありません。このクライアントが正しくないデータをコミットしようとすると、例外が作成されます。

例外をトリガーしたキーの検索

そのような例外のトラブルシューティングのとき、例外をトリガーしたエントリーに対応するキーを検索すると便利です。OptimisticCollisionException の利点は、キーを表すオブジェクトを戻す getKey メソッドが含まれていることです。次の例は、OptimisticCollisionException をキャッチするときの、キーを検索し印刷する方法を示しています。

```
try {
...
} catch (OptimisticCollisionException oce) {
    System.out.println(oce.getKey());
}
```

OptimisticCollisionException の原因となる ObjectGridException

OptimisticCollisionException は、ObjectGridException が表示される原因となる場合があります。この場合、以下のコードを使用して例外タイプを判別し、キーを印刷できます。以下のコードは、以下のセクションで説明するように、findRootCause ユーティリティー・メソッドを使用しています。

```
try {
...
}
catch (ObjectGridException oe) {
    Throwable Root = findRootCause( oe );
    if (Root instanceof OptimisticCollisionException) {
```

```
        OptimisticCollisionException oce = (OptimisticCollisionException)Root;
        System.out.println(oce.getKey());
    }
}
```

一般的な例外処理技法

Throwable オブジェクトの根本原因がわかると、エラーの発生源を分離する場合に役立ちます。次の例では、例外ハンドラーでユーティリティー・メソッドを使用して Throwable オブジェクトの根本原因を検出する方法について説明します。

例:

```
static public Throwable findRootCause( Throwable t )
{
     // Start with Throwable that occurred as the root.
     Throwable root = t;

     // Follow cause chain until last Throwable in chain is found.
     Throwable cause = root.getCause();
     while ( cause != null )
     {
        root = cause;
        cause = root.getCause();
     }

     // Return last Throwable in the chain as the root cause.
     return root;
}
```

ObjectMap API

ObjectMap は Java Map に似ていて、キーと値のペアでデータを保管できるようにします。ObjectMap は、アプリケーションがデータを保管するための簡素で直観的なアプローチを提供します。ObjectMap は、相互関係のないオブジェクトをキャッシュするのに理想的です。オブジェクト関係がある場合は、EntityManager API を使用するようにしてください。

EntityManager API について詳しくは、 56 ページの『EntityManager API』を参照してください。

アプリケーションは通常、WebSphere eXtreme Scale 参照を取得し、その参照からスレッドごとにセッション・オブジェクトを取得します。セッションはスレッド間で共有することはできません。セッションの getMap メソッドは、このスレッドに対して使用する ObjectMap への参照を返します。

ObjectMap の概要

ObjectMap インターフェースは、アプリケーションと BackingMap との間のトランザクション対話のために使用されます。

目的

ObjectMap インスタンスが、現行スレッドと対応するセッション・オブジェクトから獲得されます。 ObjectMap インターフェースは、BackingMap 内のエントリーを変更するためにアプリケーションが使用するメイン媒体です。

ObjectMap インスタンスの取得

アプリケーションは、Session.getMap(String) メソッドを使用して、セッション・オブジェクトから ObjectMap インスタンスを取得します。以下のコード・スニペットは、ObjectMap インスタンスの獲得方法を示すものです。

```
ObjectGrid objectGrid = ...;
BackingMap backingMap = objectGrid.defineMap("mapA");
Session sess = objectGrid.getSession();
ObjectMap objectMap = sess.getMap("mapA");
```

各 ObjectMap インスタンスは、特定のセッション・オブジェクトと対応しています。特定のセッション・オブジェクトで同じ BackingMap 名を使用して getMap メソッドを複数回呼び出すと、常に同じ ObjectMap インスタンスが戻されます。

トランザクションの自動コミット

ObjectMap と JavaMap を使用する BackingMap に対する操作は、セッション・トランザクション内では非常に効率よく実行されます。ObjectMap インターフェースおよび JavaMap インターフェース上のメソッドがセッション・トランザクションの外部で呼び出される場合、WebSphere eXtreme Scale は、自動コミット・サポートを提供します。メソッドは、暗黙のトランザクションを開始し、要求された操作を実行し、その暗黙のトランザクションをコミットします。

メソッドのセマンティクス

以下で、ObjectMap インターフェースおよび JavaMap インターフェース上の各メソッドの背後にあるセマンティクスについて説明します。 setDefaultKeyword メソッド、invalidateUsingKeyword メソッド、およびシリアライズ可能な引数を持つメソッドについては、キーワードのトピックで解説しています。 setTimeToLive メソッドについては、Evictor のトピックで解説しています。これらのメソッドの詳細については、API 資料を参照してください。

containsKey メソッド

containsKey メソッドは、キーが BackingMap または Loader に値を持っているかどうかを判別します。アプリケーションが NULL 値をサポートしている場合は、このメソッドは get 操作から戻された NULL 参照が NULL 値を参照しているのか、 BackingMap および Loader がキーを含んでいないことを示すのかを判別するために使用できます。

flush メソッド

flush メソッドのセマンティクスは、Session インターフェース上の flush メソッドと似ています。注意すべき相違点は、セッション・フラッシュが、現行セッション内で変更されたすべてのマップの現行の保留変更点を適用するということです。このメソッドを使用すると、この ObjectMap インスタンスでの変更のみがローダーにフラッシュされます。

get メソッド

get メソッドは、BackingMap インスタンスからエントリーをフェッチします。BackingMap インスタンス内でエントリーが検出されず、Loader が BackingMap インスタンスと関連付けられている場合、BackingMap インスタンスは、Loader からエントリーをフェッチしようとします。getAll メソッドは、バッチ・フェッチ処理を可能にするために提供されています。

getForUpdate メソッド

getForUpdate メソッドは get メソッドと同じですが、getForUpdate メソッドを使用すると、BackingMap および Loader に対してエントリーを更新することが目的であることが指示されます。Loader はこのヒントを使用して、データベース・バックエンドに「SELECT for UPDATE」照会を発行できます。 BackingMap にペシミスティック・ロック・ストラテジーが定義されている場合、ロック・マネージャーがエントリーをロックします。getAllForUpdate メソッドは、バッチ・フェッチ処理を可能にするために提供されています。

insert メソッド

insert メソッドは、BackingMap および Loader にエントリーを挿入します。このメソッドを使用すると、これまで存在していないエントリーを挿入するということが BackingMap および Loader に通知されます。既存のエントリー上でこのメソッドを起動すると、メソッドが起動されるとき、あるいは現行のトランザクションがコミットされるときに例外が発生します。

invalidate メソッド

invalidate メソッドのセマンティクスは、このメソッドに渡される **isGlobal** パラメーターの値によって決まります。invalidateAll メソッドは、バッチ無効化処理を可能にするために提供されています。

invalidate メソッドの **isGlobal** パラメーターとして値 false が渡される場合は、ローカル無効化を指定します。ローカル無効化は、トランザクション・キャッシュ内のエントリーへのいかなる変更も破棄します。アプリケーションが get メソッドを発行した場合、エントリーは BackingMap 内でコミットされた最後の値からフェッチします。 BackingMap 内にエントリーがない場合は、ローダー内で最後にフラッシュされたかまたはコミットされた値から、エントリーが取り出されます。トランザクションがコミットされるとき、ローカルに無効化されているとマークされたエントリーはいずれも BackingMap に影響を与えません。ローダーにフラッシュされたすべての変更は、エントリーが無効化された場合であってもコミットされます。

invalidate メソッドの **isGlobal** パラメーターとして true が渡される場合、グローバル無効化が指定されます。グローバル無効化は、トランザクション・キャッシュ内のエントリーに対するすべての保留中の変更を破棄し、エントリー上で実行された以降の操作で BackingMap 値をバイパスします。トランザクションがコミットされているとき、グローバルに無効化されているとマークされたエントリーはいずれも BackingMap から除去されます。 以下の無効化のユース・ケースを例として考えます。 BackingMap が自動増分列を持つデータベース表から戻されます。増分列はレコードに固有の番号を割り当てるために有効です。アプリケーションはエントリーを挿入します。挿入の後で、アプリケーションは挿入された行のシーケンス番号を認識しておく必要があります。オブジェクトのコピーが古いことが分かると、グローバル無効化を使用して Loader から値を入手します。以下のコードはこのユース・ケースを説明しています。

```
Session sess = objectGrid.getSession();
ObjectMap map = sess.getMap("mymap");
sess.begin();
map.insert("Billy", new Person("Joe", "Bloggs", "Manhattan"));
sess.flush();
```

```
map.invalidate("Billy", true);
Person p = map.get("Billy");
System.out.println("Version column is: " + p.getVersion());
map.commit();
```

このサンプル・コードは、Billy にエントリーを追加します。 Person のバージョン属性が、データベースの自動増分列を使用して設定されます。アプリケーションは、最初に挿入コマンドを実行します。次にフラッシュを発行して、挿入を Loader およびデータベースに送信します。データベースはこのバージョン列をシーケンスの次の番号に設定します。これによりトランザクション内の Person オブジェクトが期限切れになります。このオブジェクトを更新するために、アプリケーションがグローバルに無効化されます。発行される次の get メソッドは、Loader からエントリーを取得し、トランザクションの値を無視します。エントリーは、更新されたバージョン値を持つデータベースから取り出されます。

put メソッド

put メソッドのセマンティクスは、前の get メソッドがキーに対するトランザクション内で呼び出されたかどうかに依存します。アプリケーションが BackingMap または Loader 内に存在するエントリーを戻す get 操作を発行する場合、put メソッドの呼び出しは更新として解釈され、トランザクション内の前の値を戻します。前に get メソッドが呼び出されることなく put メソッド呼び出しが実行された場合、または前の get メソッド呼び出しでエントリーが見つからなかった場合、操作は挿入と解釈されます。put 操作がコミットされると、insert メソッドおよび update メソッドのセマンティクスが適用されます。putAll メソッドは、バッチの挿入および更新処理を可能にするために提供されています。

remove メソッド

remove メソッドは、BackingMap および Loader (Loader が接続されている場合) からエントリーを除去します。除去されたオブジェクトの値は、このメソッドによって戻されます。そのオブジェクトが存在していない場合、このメソッドはヌル値を戻します。removeAll メソッドは、戻り値なしでバッチ削除処理を可能にするために提供されています。

setCopyMode メソッド

setCopyMode メソッドは、この ObjectMap の CopyMode 値を指定します。このメソッドを使用すると、アプリケーションは、BackingMap 上で指定された CopyMode 値をオーバーライドできます。指定された CopyMode 値は、clearCopyMode メソッドが呼び出されるまで有効になっています。いずれのメソッドも、トランザクションの境界の外側で起動されます。
CopyMode 値は、トランザクションの途中で変更することはできません。

touch メソッド

touch メソッドは、エントリーの最終アクセス時間を更新します。このメソッドは、BackingMap からの値を検索しません。このメソッドは、自身のトランザクション内で使用します。無効化または除去のために、提供されたキーが BackingMap 内に存在しない場合は、コミット処理中に例外が発生します。

update メソッド

update メソッドは、BackingMap および Loader 内のエントリーを明示的に

更新します。このメソッドを使用して、BackingMap および Loader に、既存のエントリーを更新することを示します。存在していないエントリー上でこのメソッドを起動すると、メソッドが起動されるとき、あるいはコミット処理中に例外が発生します。

getIndex メソッド

getIndex メソッドは、BackingMap に作成されている名前付き索引を取得しようとします。この索引は、スレッド間で共用することができず、セッションと同じ規則に基づいて機能します。戻された索引オブジェクトは、MapIndex インターフェース、MapRangeIndex インターフェース、カスタム索引インターフェースなど、正しいアプリケーション索引インターフェースにキャストする必要があります。

clear メソッド

clear メソッドは、すべての区画のマップからすべてのキャッシュ・エントリーを除去します。この操作は、自動コミット機能であるので、clear の呼び出し時には、アクティブ・トランザクションが存在しないようにします。

注: clear メソッドは、その呼び出しが行われたマップのみをクリアし、関連したエンティティー・マップはそのままにしておきます。このメソッドは、ローダー・プラグインを呼び出しません。

動的マップ

グリッドが既に初期化された後に、マップを作成できます。

前のバージョンの eXtreme Scale では、ObjectGrid を初期化する前にマップを定義する必要がありました。その結果として、使用されるすべてのマップを、いずれかのマップに対してトランザクションを実行する前に、作成しておく必要がありました。

動的マップの利点

動的マップの導入によって、初期化の前にすべてのマップを定義しなければならないという制約が軽減されました。テンプレート・マップの使用を通して、ObjectGrid が初期化された後にマップを作成できるようになりました。

テンプレート・マップは、ObjectGrid XML ファイル内に定義されます。前もって定義されていないマップをセッションが要求すると、テンプレート比較が実行されます。新規マップ名と、いずれかのテンプレート・マップの正規表現が一致する場合、動的にマップが作成され、要求されたマップの名前が割り当てられます。この新しく作成されたマップは、ObjectGrid XML ファイルで定義されたテンプレート・マップの設定のすべてを継承します。

動的マップの作成

動的マップ作成は、Session.getMap(String) メソッドと結びついています。このメソッドを呼び出すと、ObjectGrid XML ファイルによって構成された BackingMap に基づいて ObjectMap が戻されます。

いずれかのテンプレート・マップの正規表現に一致するストリングを渡すと、ObjectMap および関連する BackingMap が作成されるという結果になります。

Session.getMap(String cacheName) メソッドについて詳しくは、API 資料を参照してください。

XML 内でのテンプレート・マップの定義は、backingMap 要素に template ブール値属性を設定するだけの単純さです。template が true に設定されている backingMap の名前は、正規表現であると解釈されます。

WebSphere eXtreme Scale は Java 正規表現パターン・マッチングを使用します。Java での正規表現エンジンについて詳しくは、java.util.regex パッケージおよびクラス関する API 資料を参照してください。

テンプレート・マップが 1 つ定義されたサンプル ObjectGrid XML ファイルを以下に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="accounting">
   <backingMap name="payroll" readOnly="false" />
   <backingMap name="templateMap.*" template="true"
    pluginCollectionRef="templatePlugins" lockStrategy="PESSIMISTIC" />
  </objectGrid>
 </objectGrids>

 <backingMapPluginCollections>
  <backingMapPluginCollection id="templatePlugins">
   <bean id="Evictor"
    className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor" />
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

上記の XML ファイルは、1 つのテンプレート・マップと 1 つの非テンプレート・マップを定義しています。テンプレート・マップの名前は正規表現 `templateMap.*` です。この正規表現と一致するマップ名を指定して Session.getMap(String) メソッドが呼び出された場合、アプリケーションは新規マップを作成します。

注: 複数のテンプレート・マップを定義した場合は、Session.getMap(String) メソッドのどの引数の名前も、複数のテンプレート・マップに一致しないようにしてください。

例

動的マップを作成するために、テンプレート・マップの構成が必要です。ObjectGrid XML ファイル内で backingMap に template ブール値を追加します。

```
<backingMap name="templateMap.*" template="true" />
```

このテンプレート・マップの名前は、正規表現として扱われます。

この正規表現に一致する cacheName を指定して Session.getMap(String cacheName) メソッドを呼び出すと、動的マップが作成されるという結果になります。このメソッド呼び出しで 1 つの ObjectMap オブジェクトが戻され、関連する BackingMap オブジェクトが作成されます。

```
Session session = og.getSession();
    ObjectMap map = session.getMap("templateMap1");
```

新しく作成されたマップは、テンプレート・マップ定義に定義されたすべての属性およびプラグインを使用して構成されます。以下のマップが ObjectGrid を定義するのに使用されたと想定します。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="accounting">
   <backingMap name="payroll" readOnly="false" />
   <backingMap name="templateMap.*" template="true"
    pluginCollectionRef="templatePlugins" lockStrategy="PESSIMISTIC" />
  </objectGrid>
 </objectGrids>

 <backingMapPluginCollections>
  <backingMapPluginCollection id="templatePlugins">
   <bean id="Evictor"
   className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor" />
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

この XML ファイル中のテンプレート・マップをベースにして作成される動的マップにはエビクターが構成され、ロック・ストラテジーはペシミスティックになります。

テンプレートは実際の BackingMap ではないことに注意してください。つまり、「accounting」ObjectGrid には、実際の「templateMap.*」マップが含まれているのではありません。テンプレートは動的マップ作成の基礎として使用されるだけです。

テンプレート・マップを使用する際には、Session.getMap(String cacheName) メソッドの動作における変更点を考慮してください。WebSphere eXtreme Scale バージョン 7.0 より以前のバージョンでは、Session.getMap(String cacheName) メソッドの呼び出しはすべて、要求されたマップが存在していなければ UndefinedMapException 例外という結果になりました。動的マップでは、テンプレート・マップの正規表現に一致する名前であれば、マップが作成される結果になります。特に正規表現が総称である場合は、アプリケーションが作成するマップの数に注意するようにしてください。

また、eXtreme Scale セキュリティーが有効にされている場合は、動的マップ作成には ObjectGridPermission.DYNAMIC_MAP が必要です。この許可は Session.getMap(String) メソッドが呼び出されたときにチェックされます。詳しくは、「*製品概要*」でアプリケーション・クライアント許可に関する説明を参照してください。

制約および考慮事項:

制約:

- 動的マップを照会と共に使用することはできません。
- QuerySchema は mapName 用のテンプレートをサポートしません。
- 動的マップと共にエンティティーを使用することはできません。
- エンティティー BackingMap は暗黙的に定義され、クラス名を通してエンティティーにマップされます。

考慮事項:

- 多くのプラグインには、各プラグインが関連付けられたマップを判定する方法がありません。
- 他のプラグインは、差別化するためにマップ名または BackingMap 名を引数として使用します。

ObjectMap および JavaMap

JavaMap インスタンスは、ObjectMap オブジェクトから獲得されます。JavaMap インターフェースは、ObjectMap と同じメソッド・シグニチャーを持ちますが、例外処理の方法は異なります。JavaMap は、java.util.Map インターフェースを拡張します。このため、すべての例外は java.lang.RuntimeException クラスのインスタンスになります。JavaMap は java.util.Map インターフェースを拡張するので、オブジェクト・キャッシュ用に java.util.Map インターフェースを使用する既存のアプリケーションで簡単に WebSphere eXtreme Scale を使用できます。

JavaMap インスタンスの獲得

アプリケーションは、ObjectMap.getJavaMap メソッドを使用して ObjectMap オブジェクトから JavaMap インスタンスを取得します。以下のコード・スニペットは、JavaMap インスタンスの獲得方法を示すものです。

```
ObjectGrid objectGrid = ...;
BackingMap backingMap = objectGrid.defineMap("mapA");
Session sess = objectGrid.getSession();
ObjectMap objectMap = sess.getMap("mapA");
java.util.Map map = objectMap.getJavaMap();
JavaMap javaMap = (JavaMap) javaMap;
```

JavaMap は、JavaMap の獲得元である ObjectMap によって戻されます。特定の ObjectMap を使用して getJavaMap メソッドを複数回呼び出すと、常に同じ JavaMap インスタンスが戻されます。

メソッド

JavaMap インターフェースは java.util.Map インターフェース上のメソッドのサブセットのみをサポートします。 java.util.Map インターフェースは、以下のメソッドをサポートします。

containsKey(java.lang.Object) メソッド

get(java.lang.Object) メソッド

put(java.lang.Object, java.lang.Object) メソッド

putAll(java.util.Map) メソッド

remove(java.lang.Object) メソッド

clear()

java.util.Map インターフェースから継承されたその他のすべてのメソッドは、java.lang.UnsupportedOperationException 例外を生じます。

FIFO キューとしてのマップ

WebSphere eXtreme Scale を使用すると、すべてのマップに first-in first-out (FIFO) キューと類似する機能を持たせることができます。WebSphere eXtreme Scale は、すべてのマップの挿入順序を追跡します。クライアントはマップに対して、マップ内への挿入順序で次のアンロック済みエントリーを要求し、そのエントリーをロックすることができます。このプロセスにより、複数のクライアントが、そのマップのエントリーを効率的に消費できるようになります。

FIFO の例

以下のコード・スニペットは、マップが使い切られるまで、マップからエントリーを処理するループに入るクライアントを示しています。このループはトランザクションを開始してから、ObjectMap.getNextKey(5000) メソッドを呼び出します。このメソッドは、次に使用可能なアンロック済みエントリーのキーを戻して、これをロックします。トランザクションが 5000 ミリ秒を超えてブロックされていると、メソッドはヌルを戻します。

```
Session session = ...;
ObjectMap map = session.getMap("xxx");
// this needs to be set somewhere to stop this loop
boolean timeToStop = false;

while(!timeToStop)
{
  session.begin();
  Object msgKey = map.getNextKey(5000);
  if(msgKey == null)
  {
    // current partition is exhausted, call it again in
    // a new transaction to move to next partition
    session.rollback();
    continue;
  }
  Message m = (Message)ma.get(msgKey);
  // now consume the message
  ...
  // need to remove it
  map.remove(msgKey);
  session.commit();
}
```

ローカル・モードとクライアント・モードの比較

アプリケーションがクライアントではなくローカル・コアを使用している場合は、上述したメカニズムで処理が行われます。

クライアント・モードでは、Java 仮想マシン (JVM) がクライアントである場合、そのクライアントは、まずランダムのプライマリー区画に接続します。その区画に作業が存在しなければ、クライアントはその作業を求めて次の区画に移動します。クライアントは、エントリーの存在する区画を検出するか、最初のランダム区画の周辺でループするかのいずれかとなります。最初の区画の周辺でループすることになった場合のクライアントは、アプリケーションにヌル値を戻します。エントリーのあるマップを持つ区画を検出した場合のクライアントは、そのタイムアウト期間に使用可能なエントリーがなくなるまで、そのマップのエントリーを消費します。タイムアウトになると、ヌルが戻されます。このアクションでは、区画に分割されたマップが使用されている場合にヌルが戻されると、新規トランザクションを開始

して listen を再開することになります。前述のコード例の断片は、このように振る舞います。

例

クライアントとしての実行中に、キーが戻されると、その時点では、該当のトランザクションは、そのキーのエントリーを持つ区画にバインドされています。そのトランザクション中に他のマップの更新を行わなければ、問題はありません。更新を行う場合は、キーを取得したマップと同じ区画にあるマップのみ更新可能です。getNextKey メソッドから戻されたエントリーは、その区画内にある関連データを検出する方法をアプリケーションに示す必要があります。例えば、イベントとそのイベントの影響を受けるジョブの 2 つのマップがあるとします。以下のエンティティーでこの 2 つのマップを定義します。

Job.java
```
package tutorial.fifo;

import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;

@Entity
public class Job
{
 @Id String jobId;

 int jobState;
}
```

JobEvent.java
```
package tutorial.fifo;

import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;
import com.ibm.websphere.projector.annotations.OneToOne;

@Entity
public class JobEvent
{
 @Id String eventId;
 @OneToOne Job job;
}
```

ジョブには ID と状態 (整数) があります。イベントが着信したら状態を増分するとします。イベントは JobEvent マップに保管されています。エントリーには、そのイベントが関与するジョブへの参照があります。リスナーがこれを実行するためのコードは、以下の例のようになります。

JobEventListener.java
```
package tutorial.fifo;

import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class JobEventListener
{
 boolean stopListening;

 public synchronized void stopListening()
 {
  stopListening = true;
 }
```

```
 synchronized boolean isStopped()
 {
  return stopListening;
 }

 public void processJobEvents(Session session)
  throws ObjectGridException
 {
  EntityManager em = session.getEntityManager();
  ObjectMap jobEvents = session.getMap("JobEvent");
  while(!isStopped())
  {
   em.getTransaction().begin();

   Object jobEventKey = jobEvents.getNextKey(5000);
   if(jobEventKey == null)
   {
    em.getTransaction().rollback();
    continue;
   }
   JobEvent event = (JobEvent)em.find(JobEvent.class, jobEventKey);
   // process the event, here we just increment the
   // job state
   event.job.jobState++;
   em.getTransaction().commit();
  }
 }
}
```

リスナーは、スレッド上でアプリケーションによって開始されています。リスナーは、stopListening メソッドが呼び出されるまで実行されます。つまり、stopListening メソッドが呼び出されるまで、processJobEvents メソッドがスレッド上で実行されるということです。ループ・ブロックは JobEvent マップからの eventKey を待機してから、EntityManager を使用してイベント・オブジェクトにアクセスし、ジョブを逆参照し、状態を増分します。

EntityManager API には getNextKey メソッドがありませんが、ObjectMap にはあります。そのためこのコードでは、JobEvent にキーを取得させるために ObjectMap を使用します。エンティティーを持つマップを使用すると、そのマップはそれ以上オブジェクトを保管しません。その代わりに、Tuple を保管します。この Tuple とは、キーの Tuple オブジェクト、および値の Tuple オブジェクトです。EntityManager.find メソッドは、キーのタプルを受け入れます。

イベントを作成するためのコードは、以下の例のようになります。

```
em.getTransaction().begin();
Job job = em.find(Job.class, "Job Key");
JobEvent event = new JobEvent();
event.id = Random.toString();
event.job = job;
em.persist(event); // insert it
em.getTransaction().commit();
```

イベントのジョブを検索し、イベントを構成し、そのイベントにジョブを指示し、JobEvent マップに挿入し、トランザクションをコミットします。

ローダーおよび FIFO マップ

ローダーで FIFO キューとして使用されたマップを戻す場合は、追加作業がいくつか必要になることがあります。マップ内のエントリーの順序が問題ではない場合は、追加作業はありません。順序が問題となる場合は、挿入されたすべてのレコードをバックエンドに存続させる際に、それらのレコードにシーケンス番号を追加する必要があります。プリロードのメカニズムも、始動時にこの順序でレコードを挿入するように記述する必要があります。

EntityManager API

EntityManager API は、関連したオブジェクトからなる複雑なグラフを宣言したり、そのようなグラフと対話するための簡単な方法を提供することにより、eXtreme Scale キャッシュとの対話を単純化します。

エンティティー・マネージャーの概要

アプリケーションは通常、最初に ObjectGrid 参照を取得し、次にその参照からそれぞれのスレッドのセッションを取得します。 セッションはスレッド間で共有することはできません。 getEntityManager メソッドという、セッションの追加メソッドが使用可能です。このメソッドは、このスレッド用に使用するエンティティー・マネージャーへの参照を戻します。EntityManager インターフェースは、すべてのアプリケーションの Session インターフェースと ObjectMap インターフェースを置換することができます。

セッションからの EntityManager インスタンスの取得

getEntityManager メソッドは Session オブジェクトで使用可能です。以下のコードの例は、ローカル ObjectGrid インスタンスの作成および EntityManger へのアクセスの方法を示しています。サポートされているすべてのメソッドの詳細については、API 資料で EntityManager インターフェースを参照してください。

```
ObjectGrid og =
ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("intro-grid");
Session s = og.getSession();
EntityManager em = s.getEntityManager();
```

Session オブジェクトと EntityManager オブジェクトの間には、1 対 1 のリレーションシップが存在します。EntityManager オブジェクトは複数回使用することができます。コード例および詳しい説明については、 「*製品概要*」でエンティティー・マネージャーに関するチュートリアルを参照してください。

エンティティーの永続化

エンティティーの永続化とは、新規エンティティーの状態を ObjectGrid キャッシュに保存することを意味します。persist メソッドが呼び出されると、エンティティーは管理対象状態になります。永続化はトランザクションの操作であり、新規エンティティーはトランザクションのコミット後に ObjectGrid キャッシュに保管されます。

すべてのエンティティーに対応する BackingMap があり、タプルが保管されています。BackingMap はエンティティーと同じ名前で、クラスの登録時に作成されます。以下のコード例は、persist 操作を使用して Order オブジェクトを作成する方法を示します。

```
Order order = new Order(123);
em.persist(order);
order.setX();
...
```

Order オブジェクトはキー 123 を使用して作成され、 persist メソッドに渡されます。それに続けて、トランザクションをコミットする前にオブジェクトの状態を変更することができます。

注: 前記の例には、begin や commit などの必要なトランザクション境界が含まれていません。詳しくは、 「*製品概要*」でエンティティー・マネージャーに関するチュートリアルを参照してください。

エンティティーの検索

ObjectGrid キャッシュ内のエンティティーは、キャッシュに保管された後に、キーを指定することにより find メソッドで見つけることができます。このメソッドは、トランザクション境界を必要としないため、読み取り専用セマンティクスに有用です。以下の例では、1 行のコードのみでエンティティーを見つけることができることを示しています。

```
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
```

エンティティーの除去

remove メソッドは、persist メソッドと同様、トランザクション操作です。以下の例は、begin メソッドと commit メソッドを呼び出すことによってトランザクション境界を示しています。

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
em.remove(foundOrder );
em.getTransaction().commit();
```

エンティティーは、トランザクション境界の内側で find メソッドを呼び出すことによって管理された後でないと、除去できません。その後で、EntityManager インターフェースで remove メソッドを呼び出します。エンティティーのさまざまな状態 (新規、管理対象、切り離し済み、除去済みなど) について詳しくは、 82 ページの『エンティティー・インスタンスのライフサイクル』を参照してください。

エンティティーの無効化

invalidate メソッドの動作は、remove メソッドとよく似ていますが、ローダー・プラグインを呼び出すことはありません。ObjectGrid からエンティティーを除去するが、バックエンド・データ・ストアではそのまま保持するには、このメソッドを使用します。

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
em.invalidate(foundOrder );
em.getTransaction().commit();
```

エンティティーは、トランザクション境界の内側で find メソッドを呼び出すことによって管理された後でないと、無効化できません。 find メソッドを呼び出した後、EntityManager インターフェースで invalidate メソッドを呼び出すことができます。エンティティーのさまざまな状態について詳しくは、 82 ページの『エンティティー・インスタンスのライフサイクル』を参照してください。

エンティティーの更新

update メソッドもトランザクション操作です。更新を適用する前に、エンティティーを管理する必要があります。

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
foundOrder.date = new Date(); // update the date of the order
em.getTransaction().commit();
```

上の例では、エンティティーが更新された後で persist メソッドは呼び出されていません。エンティティーは、トランザクションのコミット時に ObjectGrid キャッシュで更新されます。

照会の使用

柔軟な照会エンジンにより、EntityManager API を使用してエンティティーを取得することができます。ObjectGrid 照会言語を使用することにより、エンティティーまたはオブジェクト・ベースのスキーマで SELECT タイプ照会を作成します。Query インターフェースでは、EntityManager API を使用して照会を実行する方法を詳細に説明しています。照会の使用について詳しくは、 89 ページの『照会 API』を参照してください。

照会キューの使用

エンティティー QueryQueue は、キューに似たデータ構造体であり、エンティティー照会に関連付けられます。これは、照会フィルターの WHERE 条件に一致するすべてのエンティティーを選択し、結果のエンティティーをキューに入れます。その後、クライアントは、このキューからエンティティーを繰り返し取り出すことができます。エンティティーで照会キューを使用する方法について詳しくは、 77 ページの『エンティティー照会キュー』を参照してください。

関連資料

API 資料: EntityManager インターフェース

81 ページの『EntityTransaction インターフェース』
EntityTransaction インターフェースを使用すると、トランザクションを区別できます。

ObjectMap API と EntityManager API

ほとんどのキャッシュ製品では、マップ・ベースの API を使用して、データをキーと値のペアとして保管していました。特に ObjectMap API および WebSphere Application Server の動的キャッシュでは、この方法を使用しています。マップ・ベースの API は良好に機能するものの、いくつかの制限事項があります。

マップ・ベースの API および ObjectMap API の制限

WebSphere Application Server の動的キャッシュや ObjectMap API などのマップ・ベースの API を使用している場合、以下のような制限があります。

- キャッシュは、キャッシュ内のオブジェクトからデータを抽出するためにリフレクションを使用する必要があります。これはパフォーマンスに影響します。
- 2 つのアプリケーションが同一データの異なるオブジェクトを使用する場合、キャッシュを共用することはできません。
- データの展開は不可能です。キャッシュされた Java オブジェクトに簡単に属性を追加することはできません。
- オブジェクトのグラフ処理は煩雑になります。アプリケーションは、オブジェクト間の人工的な参照を保管し、手動で結合させる必要があります。

EntityManager API

EntityManager API は、既存の Map ベースのインフラストラクチャーを使用しますが、Map への保管または Map からの読み取りの前に、エンティティー・オブジェクトとタプル間の変換を行います。エンティティー・オブジェクトはキー・タプルおよび値タプルに変換され、キーと値のペアとして保管されます。タプルとは、画素属性の配列です。

この API の集合は、ほとんどのフレームワークで採用されている Plain Old Java Object (POJO) スタイルのプログラミングに従うことにより、eXtreme Scale の使用を大幅に簡素化します。

エンティティー・スキーマの定義

各 eXtreme Scale が持つことができる論理エンティティー・スキーマの数には、制限がありません。エンティティーは、アノテーション付き Java クラス、XML、または XML と Java クラスの組み合わせを使用して定義されます。定義されたエンティティーは、eXtreme Scale に登録され、バックアップ・マップ、索引、およびその他のプラグインにバインドされます。

エンティティー・スキーマを設計する場合は、以下のタスクを完了する必要があります。

1. エンティティーおよびそのリレーションシップを定義します。
2. eXtreme Scaleを構成します。
3. エンティティーを eXtreme Scale に登録します。
4. eXtreme Scale、エンティティー・マネージャー、およびエンティティーと対話するアプリケーションを作成します。

エンティティー・スキーマ構成

エンティティー・スキーマとは、1 組のエンティティーとそれらエンティティーの間のリレーションシップのことです。複数の区画を持つ eXtreme Scale では、エンティティー・スキーマには以下の制約事項およびオプションが適用されます。

- 各エンティティー・スキーマには、単一のルートが定義されている必要があります。これは、スキーマ・ルートと呼ばれます。

- 一定スキーマのすべてのエンティティーは、同じマップ・セットに入っている必要があります。つまり、キーまたは非キーのリレーションシップによってスキーマ・ルートから到達できるすべてのエンティティーは、スキーマ・ルートと同じマップ・セットに定義する必要があります。
- 各エンティティーは、1 つのエンティティー・スキーマのみに属することができます。
- 各 eXtreme Scale は、複数のスキーマを持つことができます。

エンティティーは、その初期化の前に eXtreme Scale に登録されます。定義された各エンティティーは、固有の名前を持つ必要があり、同じ名前の eXtreme Scale バックアップ・マップに自動的にバインドされます。初期化メソッドは、使用中の構成によって変わります。

スタンドアロン eXtreme Scale

スタンドアロン eXtreme Scale 構成を使用している場合は、エンティティー・スキーマをプログラマチックに構成できます。このモードでは、ObjectGrid.registerEntities メソッドを使用して、アノテーション付きエンティティー・クラスまたはエンティティー・メタデータ記述子ファイルを登録することができます。

分散 eXtreme Scale 構成

分散 eXtreme Scale 構成を使用している場合は、エンティティー・スキーマを含むエンティティー・メタデータ記述子ファイルを指定する必要があります。

エンティティー・クラスの要件

エンティティーは、さまざまなメタデータを Java クラスに関連付けることによって識別されます。メタデータは、Java Platform, Standard Edition 5 のアノテーション、エンティティー・メタデータ記述子ファイル、またはアノテーションと記述子ファイルの組み合わせを使用して指定できます。エンティティー・クラスは、以下の基準を満たしている必要があります。

- `@Entity` アノテーションがエンティティー XML 記述子ファイルで定義または指定されている必要があります。
- 引数を取らない、Public または Protected のコンストラクターを持っている必要があります。
- 最上位クラスである必要があります。インターフェースおよび列挙型 (enum) は、有効なエンティティー・クラスではありません。
- final キーワードを使用することはできません。
- 継承を使用することはできません。
- eXtreme Scale ごとに名前と型を固有とする必要があります。

すべてのエンティティーは固有の名前と型を持っています。アノテーションを使用している場合、名前はデフォルトではクラスの単純名 (短い名前) ですが、`@Entity` アノテーションの name 属性を使用してオーバーライドできます。

パーシスタント属性

エンティティーのパーシスタント状態は、フィールド (インスタンス変数) を使用するか、Enterprise JavaBeans スタイルのプロパティー・アクセサーを使用して、クライアントおよびエンティティー・マネージャーによってアクセスされます。各エンティティーはフィールドまたはプロパティー・ベースのいずれかのアクセスを定義する必要があります。アノテーション付きエンティティーは、クラス・フィールドがアノテーション付きの場合はフィールド・アクセスとなり、プロパティーの getter メソッドがアノテーション付きである場合はプロパティー・アクセスとなります。フィールド・アクセスとプロパティー・アクセスを混在させることはできません。タイプを自動的に判別できない場合は、`@Entity` アノテーションまたは同等の XML で **accessType** 属性を使用してアクセス・タイプを識別できます。

パーシスタント・フィールド

フィールド・アクセス・エンティティー・インスタンス変数は、エンティティー・マネージャーおよびクライアントから直接アクセスされます。transient 修飾子または transient アノテーションでマークされているフィールドは無視されます。パーシスタント・フィールドの修飾子を final または static にすることはできません。

パーシスタント・プロパティー

プロパティー・アクセス・エンティティーは、読み取りおよび書き込みプロパティーに関しては、JavaBeans™ シグニチャー規則に従う必要があります。 JavaBeans 規則に従わないメソッド、または getter メソッドに Transient アノテーションを持つメソッドは無視されます。型 `T` のプロパティーの場合、T getProperty() という getter メソッドと void setProperty(T) という setter メソッドが必要です。ブール型の場合、getter メソッドは boolean isProperty() と表すことができます。パーシスタント・プロパティーは、static 修飾子を持つことができません。

サポートされる属性タイプ

以下のパーシスタント・フィールドおよびプロパティー・タイプがサポートされます。

- ラッパーを含む Java プリミティブ型
- java.lang.String
- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.util.Calendar
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp
- byte[]
- java.lang.Byte[]
- char[]
- java.lang.Character[]
- enum

ユーザー・シリアライズ可能属性の型はサポートされてはいますが、パフォーマンス、照会、および変更検出に制約があります。配列やユーザー・シリアライズ可能オブジェクトなど、プロキシー処理できないパーシスタント・データは、変更された場合にはエンティティーに再割り当てする必要があります。

エンティティー・アソシエーション

双方向および単一方向のエンティティー・アソシエーションまたはエンティティー間リレーションシップは、1 対 1、多対 1、1 対多、および多対多として定義できます。エンティティー・マネージャーは、eXtreme Scale にエンティティーを保管するときに、自動的にエンティティー・リレーションシップを適切なキー参照に解決します。

eXtreme Scale はデータ・キャッシュであり、データベースとは異なり、参照整合性を実施しません。リレーションシップでは子エンティティーに対して永続化操作および除去操作をカスケードすることができますが、オブジェクトとのリンク切れを検出または引き起こすことはありません。子オブジェクトを除去する場合は、そのオブジェクトへの参照を親から除去する必要があります。

2 つのエンティティー間の双方向アソシエーションを定義する場合、リレーションシップの所有者を識別する必要があります。多くのアソシエーションでは、リレーションシップの多側は常に所有側になります。所有権を自動的に判別できない場合、アノテーションの **mappedBy** 属性、または XML におけるそれと同等な属性を指定する必要があります。**mappedBy** 属性は、リレーションシップの所有者であるターゲット・エンティティー内のフィールドを識別します。この属性は、型と基数が同じである複数の属性が存在する場合に、関連するフィールドを識別するのにも役立ちます。

一価アソシエーション

1 対 1 および多対 1 のアソシエーションは、@OneToOne および @ManyToOne アノテーションまたはそれと等価な XML 属性を使用して示されます。ターゲット・エンティティーの型は、属性の型によって決定されます。以下の例では、Person と Address の間に単一方向アソシエーションがあります。Customer エンティティーは、1 つの Address エンティティーへの参照を持っています。この場合、逆のリレーションシップがないため、アソシエーションを多対 1 にすることもできます。

```
@Entity
public class Customer {
  @Id id;
  @OneToOne Address homeAddress;
}

@Entity
public class Address{
  @Id id
  @Basic String city;
}
```

Customer クラスと Address クラスの間の双方向リレーションシップを指定するには、Address クラスから Customer クラスへの参照を追加し、適切なアノテーションを追加して、反対側にリレーションシップを指定します。このアソシエーションは

1 対 1 であるため、@OneToOne アノテーションで mappedBy 属性を使用してリレーションシップの所有者を指定する必要があります。

```
@Entity
public class Address{
  @Id id
  @Basic String city;
  @OneToOne(mappedBy="homeAddress") Customer customer;
}
```

集合価アソシエーション

1 対多および多対多のアソシエーションは、@OneToMany および @ManyToMany アノテーションまたはそれと等価な XML 属性を使用して示されます。多くのリレーションシップはすべて、java.util.Collection、java.util.List、または java.util.Set という型を使用して表されます。ターゲット・エンティティーの型は、Collection、List、または Set という汎用型によって決定されるか、@OneToMany または @ManyToMany アノテーションの **targetEntity** 属性 (またはそれと等価な XML での属性) を使用して明示的に決定されます。

前出の例では、顧客ごとに 1 つの住所オブジェクトを持たせることは現実的ではありません。それは、多くの顧客が 1 つの住所を共用したり、複数の住所を持っていることがあるからです。これを解決する 1 つの良い方法は、「多」のアソシエーションを使用することです。

```
@Entity
public class Customer {
  @Id id;
  @ManyToOne Address homeAddress;
  @ManyToOne Address workAddress;
}

@Entity
public class Address{
  @Id id
  @Basic String city;
  @OneToMany(mappedBy="homeAddress") Collection<Customer> homeCustomers;


  @OneToMany(mappedBy="workAddress", targetEntity=Customer.class)
  Collection workCustomers;
}
```

この例では、同じエンティティー間に「自宅アドレス」リレーションシップと「勤務先アドレス」リレーションシップという 2 つの異なるリレーションシップが存在します。**workCustomers** 属性に非汎用型の Collection が使用されているのは、汎用型を使用できない場合に **targetEntity** 属性を使用する方法を示すためです。

1 次キー

すべてのエンティティーは 1 次キーを持つ必要があり、単純キー (単一属性) または複合キー (複数属性) として指定できます。キー属性は Id アノテーションを使用して示すか、またはエンティティー XML 記述子ファイルで定義します。キー属性には以下の要件があります。

- 1 次キーの値は変更できません。

- 1 次キー属性の型は、Java プリミティブ型およびラッパー、java.lang.String、java.util.Date、または java.sql.Date のいずれかにする必要があります。
- 1 次キーには、一価アソシエーションを任意の数だけ含めることができます。1 次キー・アソシエーションのターゲット・エンティティーは、ソース・エンティティーとの直接的または間接的な逆アソシエーションを持つことができません。

複合 1 次キーは、必要に応じて 1 次キー・クラスを定義できます。エンティティーは、@IdClass アノテーションを使用するか、またはエンティティー XML 記述子ファイルで定義して 1 次キー・クラスに関連付けられます。@IdClass アノテーションは、EntityManager.find メソッドと一緒に使用する場合に役立ちます。

1 次キー・クラスには以下の要件があります。

- 1 次キー・クラスは public で、引数を取らない public コンストラクターを持っている必要があります。
- 1 次キー・クラスのアクセス・タイプは、1 次キー・クラスを宣言しているエンティティーによって決定されます。
- プロパティー・アクセスの場合、1 次キー・クラスのプロパティーは、public または protected にする必要があります。
- 1 次キーのフィールドまたはプロパティーは、参照側のエンティティーで定義されているキー属性の名前と型に一致している必要があります。
- 1 次キー・クラスは、equals メソッドおよび hashCode メソッド を実装している必要があります。

```
@Entity
@IdClass(CustomerKey.class)
public class Customer {
    @Id @ManyToOne Zone zone;
    @Id int custId;
    String name;
    ...
}

@Entity
public class Zone{
    @Id String zoneCode;
    String name;
}

public class CustomerKey {
    Zone zone;
    int custId;

    public int hashCode() {...}
    public boolean equals(Object o) {...}
}
```

エンティティー・プロキシーおよびフィールド・インターセプト

エンティティー・クラスおよび可変のサポートされている属性のタイプは、プロパティー・アクセス・エンティティーのプロキシー・クラスによって拡張され、Java Development Kit (JDK) 5 のフィールド・アクセス・エンティティー向けにバイト・コード拡張されています。内部ビジネス・メソッドおよび equals メソッドを使用す

る場合であっても、エンティティーにアクセスする場合は常に、適切なフィールド・アクセス・メソッドまたはプロパティー・アクセス・メソッドを使用する必要があります。

プロキシーおよびフィールド・インターセプターを使用すると、エンティティー・マネージャーがエンティティーの状態を追跡して、エンティティーが変更されたかどうかを判別し、パフォーマンスを改善できるようになります。フィールド・インターセプターは、エンティティー・インスツルメンテーション・エージェントの構成時に、Java SE 5 プラットフォームでのみ使用できます。

重要: プロパティー・アクセス・エンティティーを使用している場合、equals メソッドによる現行のインスタンスと入力オブジェクトの比較には instanceof 演算子を使用する必要があります。ターゲット・オブジェクトのすべてのイントロスペクションは、ターゲット・オブジェクトのプロパティーを介して行い、フィールドそれ自体を介さないようにする必要があります。これは、ターゲット・オブジェクト・インスタンスはプロキシーになるからです。

emd.xsd ファイル

エンティティー・メタデータ XML スキーマ定義を使用して記述子 XML ファイルを作成し、 WebSphere eXtreme Scale のエンティティー・スキーマを定義します。

emd.xsd ファイルの各エレメントおよび属性の説明は、「*管理ガイド*」でエンティティー・メタデータ記述子ファイルに関する情報を参照してください。

emd.xsd ファイル

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:emd="http://ibm.com/ws/projector/config/emd"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://ibm.com/ws/projector/config/emd"
 elementFormDefault="qualified" attributeFormDefault="unqualified"
 version="1.0">

 <xsd:element name="entity-mappings"
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    <xsd:element name="entity" type="emd:entity" minOccurs="1" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:unique name="uniqueEntityClassName">
   <xsd:selector xpath="emd.entity"/>
   <xsd:field xpath="@class-name"/>
  </xsd:unique>
 </xsd:element>

 <xsd:complexType name="entity">
  <xsd:sequence>
   <xsd:element name="description" type="xsd:string" minOccurs="0"/>
   <xsd:element name="id-class" type="emd:id-class" minOccurs="0"/>
   <xsd:element name="attributes" type="emd:attributes" minOccurs="0"/>
   <xsd:element name="entity-listeners" type="emd:entity-listeners" minOccurs="0"/>
   <xsd:element name="pre-persist" type="emd:pre-persist" minOccurs="0"/>
   <xsd:element name="post-persist" type="emd:post-persist" minOccurs="0"/>
   <xsd:element name="pre-remove" type="emd:pre-remove" minOccurs="0"/>
   <xsd:element name="post-remove" type="emd:post-remove" minOccurs="0"/>
   <xsd:element name="pre-invalidate" type="emd:pre-invalidate" minOccurs="0"/>
   <xsd:element name="post-invalidate" type="emd:post-invalidate" minOccurs="0"/>
   <xsd:element name="pre-update" type="emd:pre-update" minOccurs="0"/>
   <xsd:element name="post-update" type="emd:post-update" minOccurs="0"/>
   <xsd:element name="post-load" type="emd:post-load" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="class-name" type="xsd:string" use="required"/>
  <xsd:attribute name="access" type="emd:access-type"/>
  <xsd:attribute name="schemaRoot" type="xsd:boolean"/>
 </xsd:complexType>

 <xsd:complexType name="attributes">
  <xsd:sequence>
   <xsd:choice>
```

```
   <xsd:element name="id" type="emd:id" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:choice>
  <xsd:element name="basic" type="emd:basic" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="version" type="emd:version" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="many-to-one" type="emd:many-to-one" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="one-to-many" type="emd:one-to-many" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="one-to-one" type="emd:one-to-one" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="many-to-many" type="emd:many-to-many" minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="transient" type="emd:transient" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="access-type">
 <xsd:restriction base="xsd:token">
  <xsd:enumeration value="PROPERTY"/>
  <xsd:enumeration value="FIELD"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="id-class">
 <xsd:attribute name="class-name"> type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="id">
 <xsd:attribute name="name" type="xsd:string" use="required"/>
 <xsd:attribute name="alias" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:complexType name="transient">
 <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="basic">
 <xsd:attribute name="name" type="xsd:string" use="required"/>
 <xsd:attribute name="alias" type="xsd:string"/>
 <xsd:attribute name="fetch" type="emd:fetch-type"/>
</xsd:complexType>

<xsd:simpleType name="fetch-type">
 <xsd:restriction base="xsd:token">
  <xsd:enumeration value="LAZY"/>
  <xsd:enumeration value="EAGER"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="many-to-one">
 <xsd:sequence>
  <xsd:element name="cascade" type="emd:cascade-type" minOccurs="0"/>
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:string" use="required"/>
 <xsd:attribute name="alias" type="xsd:string"/>
 <xsd:attribute name="target-entity" type="xsd:string"/>
 <xsd:attribute name="fetch" type="emd:fetch-type"/>
 <xsd:attribute name="id" type="xsd:boolean"/>
</xsd:complexType>

<xsd:complexType name="one-to-one">
 <xsd:sequence>
  <xsd:element name="cascade" type="emd:cascade-type" minOccurs="0"/>
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:string" use="required"/>
 <xsd:attribute name="alias" type="xsd:string"/>
 <xsd:attribute name="target-entity" type="xsd:string"/>
 <xsd:attribute name="fetch" type="emd:fetch-type"/>
 <xsd:attribute name="mapped-by" type="xsd:string"/>
 <xsd:attribute name="id" type="xsd:boolean"/>
</xsd:complexType>

<xsd:complexType name="one-to-many">
 <xsd:sequence>
  <xsd:element name="order-by" type="emd:order-by" minOccurs="0"/>
  <xsd:element name="cascade" type="emd:cascade-type" minOccurs="0"/>
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:string" use="required"/>
 <xsd:attribute name="alias" type="xsd:string"/>
 <xsd:attribute name="target-entity" type="xsd:string"/>
 <xsd:attribute name="fetch" type="emd:fetch-type"/>
 <xsd:attribute name="mapped-by" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="many-to-many">
 <xsd:sequence>
  <xsd:element name="order-by" type="emd:order-by" minOccurs="0"/>
  <xsd:element name="cascade" type="emd:cascade-type" minOccurs="0"/>
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:string" use="required"/>
 <xsd:attribute name="alias" type="xsd:string"/>
 <xsd:attribute name="target-entity" type="xsd:string"/>
 <xsd:attribute name="fetch" type="emd:fetch-type"/>
 <xsd:attribute name="mapped-by" type="xsd:string"/>
</xsd:complexType>
```

```
<xsd:simpleType name="order-by">
 <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<xsd:complexType name="cascade-type">
 <xsd:sequence>
  <xsd:element name="cascade-all" type="emd:emptyType" minOccurs="0"/>
  <xsd:element name="cascade-persist" type="emd:emptyType" minOccurs="0"/>
  <xsd:element name="cascade-remove" type="emd:emptyType" minOccurs="0"/>
  <xsd:element name="cascade-invalidate" type="emd:emptyType" minOccurs="0"/>
  <xsd:element name="cascade-merge" type="emd:emptyType" minOccurs="0"/>
  <xsd:element name="cascade-refresh" type="emd:emptyType" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="emptyType/>

<xsd:complexType name="version">
 <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="entity-listeners">
 <xsd:sequence>
  <xsd:element name="entity-listener" type="emd:entity-listener" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="entity-listener">
 <xsd:sequence>
  <xsd:element name="pre-persist" type="emd:pre-persist" minOccurs="0"/>
  <xsd:element name="post-persist" type="emd:post-persist" minOccurs="0"/>
  <xsd:element name="pre-remove" type="emd:pre-remove" minOccurs="0"/>
  <xsd:element name="post-remove" type="emd:post-remove" minOccurs="0"/>
  <xsd:element name="pre-invalidate" type="emd:pre-invalidate" minOccurs="0"/>
  <xsd:element name="post-invalidate" type="emd:post-invalidate" minOccurs="0"/>
  <xsd:element name="pre-update" type="emd:pre-update" minOccurs="0"/>
  <xsd:element name="post-update" type="emd:post-update" minOccurs="0"/>
  <xsd:element name="post-load" type="emd:post-load" minOccurs="0"/>
 </xsd:sequence>
 <xsd:attribute name="class-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="pre-persist">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="post-persist">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="pre-remove">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="post-remove">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="pre-invalidate">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="post-invalidate">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="pre-update">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="post-update">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="post-load">
 <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

</xsd:schema>
```

組み込みサーバー API

WebSphere eXtreme Scale では、いくつかのアプリケーション・プログラミング・インターフェース (API) およびシステム・プログラミング・インターフェースにより、Java プログラミング言語を使用してプログラマチックにアクセスされるいくつかのフィーチャーを提供しています。

ServerFactor.getInstance

JVM が eXtreme Scale サーバー・ランタイムをホストできるようにするためには、ServerFactory.getInstance メソッドを発行します。このメソッドは、サーバー singleton を返し、サーバー・ランタイムを開始します。この getInstance メソッドを発行する前に、サーバー・プロパティーを設定する必要があります。そうでないと、メソッド発行後に行った変更が、ランタイムに反映されなくなります。

eXtreme Scale サーバーのインスタンス化

```
Server server = ServerFactory.getInstance();
```

eXtreme Scale サーバー・インスタンスの構成には、いくつかのプロパティーを使用できますが、これは、ServerFactory.getServerProperties メソッドから取得できます。ServerProperties オブジェクトは singleton で、したがって、getServerProperties メソッドの各呼び出しでは同じインスタンスが取得されます。getInstance の最初の呼び出しで設定されたすべてのプロパティーは、サーバーの初期化に使用されます。

サーバー・プロパティーの設定

サーバー・プロパティーは、ServerFactory.getInstance が最初に呼び出されるまで設定できます。 getInstance メソッドの最初の呼び出しで、eXtreme Scale サーバーがインスタンス化され、構成されたすべてのプロパティーが読み取られます。作成後にプロパティーを設定しても効果はありません。

組み込み eXtreme Scale サーバーに対するプロパティーの設定

```
// Get the server properties associated with this process.
ServerProperties serverProperties = ServerFactory.getServerProperties();

// Set the server name for this process.
serverProperties.setServerName("EmbeddedServerA");

// Set the name of the zone this process is contained in.
serverProperties.setZoneName("EmbeddedZone1");

// Set the end point information required to bootstrap to the catalog service.
serverProperties.setCatalogServiceBootstrap("localhost:2809");

// Set the ORB listener host name to use to bind to.
serverProperties.setListenerHost("host.local.domain");

// Set the ORB listener port to use to bind to.
serverProperties.setListenerPort(9010);

// Turn off all MBeans for this process.
serverProperties.setMBeansEnabled(false);

Server server = ServerFactory.getInstance();
```

カタログ・サービスの組み込み

CatalogServerProperties.setCatalogServer メソッドによりフラグが立てれた JVM 設定は、eXtreme Scale のカタログ・サービスをホストできます。このメソッドは、eXtreme Scale サーバー・ランタイムに対して、サーバーの始動時にカタログ・サービスをインスタンス化することを指示します。以下のコードは、eXtreme Scale カタログ・サーバーをインスタンス化する方法を示しています。

```
CatalogServerProperties catalogServerProperties =
 ServerFactory.getCatalogProperties();
catalogServerProperties.setCatalogServer(true);

Server server = ServerFactory.getInstance();
```

eXtreme Scale コンテナーの組み込み

JVM が複数の eXtreme Scale コンテナーをホストするようにするには、Server.createContainer メソッドを発行します。以下のコードは、eXtreme Scale コンテナーをインスタンス化する方法を示しています。

```
Server server = ServerFactory.getInstance();
DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy(
    new File("META-INF/embeddedDeploymentPolicy.xml").toURL(),
    new File("META-INF/embeddedObjectGrid.xml").toURL());
Container container = server.createContainer(policy);
```

自己完結型のサーバー・プロセス

すべてのサービスは、まとめて開始することができ、これは開発に便利で、実動中にも実用的です。サービスをまとめて開始することにより、1 つのプロセスで、カタログ・サービスの開始、コンテナー・セットの開始、クライアント接続ロジックの実行をすべて行うことができます。このような方法でサービスを開始すると、分散環境にデプロイする前にプログラム上の問題を整理することができます。以下のコードは、自己完結型の eXtreme Scale サーバーをインスタンス化する方法を示しています。

```
CatalogServerProperties catalogServerProperties =
 ServerFactory.getCatalogProperties();
catalogServerProperties.setCatalogServer(true);

Server server = ServerFactory.getInstance();
DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy(
    new File("META-INF/embeddedDeploymentPolicy.xml").toURL(),
    new File("META-INF/embeddedObjectGrid.xml").toURL());
Container container = server.createContainer(policy);
```

WebSphere Application Server における eXtreme Scale の組み込み

eXtreme Scale の構成は、WebSphere Extended Deployment DataGrid を WebSphere Application Server 環境にインストールすると、自動的にセットアップされます。サーバーにアクセスしてコンテナーを作成する前にプロパティーを設定する必要はありません。以下のコードは、eXtreme Scale サーバーを WebSphere Application Server でインスタンス化する方法を示しています。

```
Server server = ServerFactory.getInstance();
DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy(
    new File("META-INF/embeddedDeploymentPolicy.xml").toURL(),
    new File("META-INF/embeddedObjectGrid.xml").toURL());
Container container = server.createContainer(policy);
```

EntityManager インターフェースのパフォーマンスへの影響

EntityManager インターフェースは、アプリケーションをグリッドで保持されている状態から切り離します。これは、グリッドとアプリケーションを結合しない場合には主要な検討事項です。例えば、すべてのアプリケーションで、データベースに同じデータ・アクセス・オブジェクトの使用を強制している環境があるとします。この実装は実用的ではありません。ただしこの使用の簡便化にはパフォーマンス・コストが伴います。

EntityManager インターフェースを使用するためのコストは高いものではなく、実行する作業の種類により異なります。アプリケーションが完成した後、必ず EntityManager インターフェースを使用して重要なビジネス・ロジックを最適化してください。EntityManager インターフェースを使用するコードを、マップとタプルを使用するように修正できます。パフォーマンスに重大な影響を与えるコードの割合に関する標準的な 90/10 ルールを適用した場合、コードの 10 % に対してこのコード修正を加える必要があると考えられます。

オブジェクト間のリレーションシップを利用すると、パフォーマンスへの影響が小さくなります。これは、マップを使用しているアプリケーションが、このようなリレーションシップを EntityManager インターフェースと同様に管理する必要があるからです。

ObjectTransformer は自動的に最適化されるため、EntityManager インターフェースを使用するアプリケーションは、ObjectTransformer を提供する必要がありません。

例: マップを使用するための EntityManager コードの修正

以下にサンプル・エンティティーを示します。

```
@Entity
public class Person
{
 @Id
 String ssn;
 String firstName;
 @Index
 String middleName;
 String surname;
}
```

エンティティーを検索し、エンティティーを更新するコードを以下に示します。

```
Person p = null;
s.begin();
p = (Person)em.find(Person.class, "1234567890");
p.middleName = String.valueOf(inner);
s.commit();
```

マップおよびタプルを使用する場合のコードは以下のとおりです。

```
Tuple key = null;
key = map.getEntityMetadata().getKeyMetadata().createTuple();
key.setAttribute(0, "1234567890");
```

```
// The Copy Mode is always NO_COPY for entity maps if not using COPY_TO_BYTES.
// Either we need to copy the tuple or we can ask the ObjectGrid to do it for us:
map.setCopyMode(CopyMode.COPY_ON_READ);
s.begin();
Tuple value = (Tuple)map.get(key);
value.setAttribute(1, String.valueOf(inner));
map.update(key, value);
value = null;
s.commit();
```

これらのコード・スニペットは両方とも同じ結果になります。どちらのコード・スニペットもアプリケーションで同時に使用できます。2 番目のコード・スニペットは、マップを直接使用する方法およびタプルを操作する方法を示しています。キーと値の組がタプルです。値タプルには 0、1 および 2 に索引が設定された名、ミドルネーム、および姓という 3 つの属性があります。キー・タプルには 0 に索引が設定された、ID 番号という単一の属性があります。EntityMetadata#getKeyMetaData メソッドまたは EntityMetadata#getValueMetaData メソッドを使用して、タプルを作成する方法を確認できます。エンティティーのタプルを作成するには、これらのメソッドを使用する必要があります。タプル・インターフェースを実装して、そのタプル実装のインスタンスを渡すような操作は、実行できません。

インスツルメンテーション・エージェント

Java Development Kit (JDK) バージョン 1.5 以降を使用している場合、WebSphere eXtreme Scale エージェントを使用可能にすることで、フィールド・アクセス・エンティティーのパフォーマンスを向上させることができます。このエージェントは、分散 eXtreme Scale 環境でのローカル eXtreme Scale またはクライアント eXtreme Scale でのみ使用されます。

JDK バージョン 1.5 以降での eXtreme Scale エージェントの使用可能化

以下の構文で Java コマンド行オプションを使用して ObjectGrid エージェントを使用可能化することができます。

```
-javaagent:jarpath[=options]
```

jarpath 値は、eXtreme Scale エージェント・クラスおよびサポート・クラスが入っている eXtreme Scale ランタイムの Java アーカイブ (JAR) ファイル (`objectgrid.jar`、`wsobjectgrid.jar`、`ogclient.jar`、`wsogclient.jar`、および `ogagent.jar` ファイルなど) へのパスです。通常、スタンドアロン Java プログラム、または WebSphere Application Server を稼働していない Java Platform, Enterprise Edition 環境では、`objectgrid.jar` ファイルまたは `ogclient.jar` ファイルを使用します。 WebSphere Application Server または複数クラス・ローダー環境では、Java コマンド行エージェント・オプションで `ogagent.jar` ファイルを使用する必要があります。

ブートストラップ・クラスパスに組み込むには、エージェント JAR ファイルに加えて `cglib.jar` ファイルが必要になります。WebSphere Application Server では、`cglib.jar` ファイルが既に lib ディレクトリー内にあり、アプリケーション・サーバー・クラスパスに組み込まれている可能性がありますが、そのアプリケーション・サーバーと関連付けられている Java 仮想マシン (JVM) のクラスパス・プロパティーに `cglib.jar` ファイル・パスを指定する必要があります。

追加情報を指定するには、クラスパスに ogagent.config ファイルを指定するか、エージェント・オプションを使用します。

eXtreme Scale エージェント・オプション

config 構成ファイル名をオーバーライドします。

include
構成ファイルの最初の部分である変換ドメイン定義を指定またはオーバーライドします。

exclude
@Exclude 定義を指定またはオーバーライドします。

fieldAccessEntity
@FieldAccessEntity 定義を指定またはオーバーライドします。

trace トレース・レベルを指定します。レベルには ALL、CONFIG、FINE、FINER、FINEST、SEVERE、WARNING、INFO、および OFF があります。

trace.file
トレース・ファイルのロケーションを指定します。

各オプションを区切るために、区切り文字としてセミコロン (;) を使用します。コンマ (,) は、オプション内の各エレメントの区切り文字として使用します。以下の例は、Java プログラムの eXtreme Scale エージェント・オプションを示します。

```
-javaagent:objectgridRoot/lib/objectgrid.jar=config=myConfigFile;
include=includedPackage;exclude=excludedPackage;
fieldAccessEntity=package1,package2
```

ogagent.config ファイル

ogagent.config ファイルは、指定された eXtreme Scale エージェント構成ファイル名です。ファイル名がクラスパス内にある場合、eXtreme Scale エージェントはそのファイルを検索し、解析します。eXtreme Scale エージェントの構成オプションを使用して、指定されたファイル名をオーバーライドすることができます。以下の例は、構成ファイルの指定方法を示しています。

```
-javaagent:objectgridRoot/lib/objectgrid.jar=config=myOverrideConfigFile
```

eXtreme Scale エージェント構成ファイルには、以下の部分があります。

- **変換ドメイン:** 変換ドメイン部分は、構成ファイルの最初にあります。変換ドメインは、クラス変換プロセスに組み込まれているパッケージおよびクラスのリストです。この変換ドメインには、フィールド・アクセス・エンティティー・クラスであるすべてのクラス、およびそれらのフィールド・アクセス・エンティティー・クラスを参照するその他のクラスが組み込まれる必要があります。フィールド・アクセス・エンティティー・クラス、およびそれらのフィールド・アクセス・エンティティー・クラスを参照するその他のクラスによって、変換ドメインは構成されます。フィールド・アクセス・エンティティー・クラスを @FieldAccessEntity 部分に指定する場合は、この部分にフィールド・アクセス・エンティティー・クラスを組み込む必要はありません。変換ドメインは、完全なものである必要があります。そうでないと、FieldAccessEntityNotInstrumentedException 例外が発生する場合があります。

- **@Exclude:** @Exclude トークンは、このトークンの後にリストされるパッケージおよびクラスが、変換ドメインから除外されることを示します。
- **@FieldAccessEntity:** @FieldAccessEntity トークンは、このトークンの後にリストされるパッケージおよびクラスが、フィールド・アクセス・エンティティー・パッケージおよびクラスであることを示します。@FieldAccessEntity トークンの後に行がない場合は、「@FieldAccessEntity が指定されていない」ことと同じになります。eXtreme Scale エージェントは、定義済みのフィールド・アクセス・エンティティー・パッケージおよびクラスはないものと判断します。
  @FieldAccessEntity トークンの後に行が存在する場合、それらの行は、ユーザー指定のフィールド・アクセス・エンティティー・パッケージおよびクラスを表します。例えば、「フィールド・アクセス・エンティティー・ドメイン」などです。フィールド・アクセス・エンティティー・ドメインは、変換ドメインのサブドメインです。フィールド・アクセス・エンティティー・ドメインにリストされているパッケージおよびクラスは、それらが変換ドメインにリストされていない場合でも変換ドメインの一部です。変換から除外されているパッケージおよびクラスをリストする @Exclude トークンは、フィールド・アクセス・エンティティー・ドメインにはまったく影響しません。 @FieldAccessEntity トークンが指定されている場合、すべてのフィールド・アクセス・エンティティーが、このフィールド・アクセス・エンティティー・ドメインに入っている必要があります。そうでないと、FieldAccessEntityNotInstrumentedException 例外が発生する場合があります。

エージェント構成ファイル (ogagent.config) の例

```
##############################
# The # indicates comment line
##############################
# This is an ObjectGrid agent config file (the designated file name is ogagent.config) that can be found and parsed by the ObjectGrid agent
# if it is in classpath.
# If the file name is "ogagent.config" and in classpath, Java program runs with -javaagent:objectgridRoot/ogagent.jar will have
# ObjectGrid agent enabled.
# If the file name is not "ogagent.config" but in classpath, you can specify the file name in config option of ObjectGrid agent
#     -javaagent:objectgridRoot/lib/objectgrid.jar=config=myOverrideConfigFile
# See comments below for more info regarding instrumentation setting override.

# The first part of the configuration is the list of packages and classes that should be included in transformation domain.
# The includes (packages/classes, construct the instrumentation doamin) should be in the beginning of the file.
com.testpackage
com.testClass

# Transformation domain: The above lines are packages/classes that construct the transformation domain.
# The system will process classes with name starting with above packages/classes for transformation.
#
# @Exclude token : Exclude from transformation domain.
# The @Exclude token indicates packages/classes after that line should be excluded from transformation domain.
# It is used when user want to exclude some packages/classes from above specified included packages
#
# @FieldAccessEntity token: Field-access Entity domain.
# The @FieldAccessEntity token indicates packages/classes after that line are field-access Entity packages/classes.
# If there is no lilne after the @FieldAccessEntity token, it is equivalent to "No @FieldAccessEntity specified".
# The runtime will consider the user does not specify any field-access Entity packages/classes.
# The "field-acces Entity domain" is a sub-domain of transformation domain.
#
# Packages/classes listed in the "field-access Entity domain" will always be part of transformation domain,
# even they are not listed in transformation domain.
# The @Exclude, which lists packages/classes excluded from transformation, has no impact on the "field-acces Entity domain".
# Note: When @FieldAccessEntity is specified, all field-access entities must be in this field-acces Entity domain,
#       otherwise, FieldAccessEntityNotInstrumentedException may occur.
#
# The default ObjectGrid agent config file name is ogagent.config
# The runtime will look for this file as a resource in classpath and process it.
# Users can override this designated ObjectGrid agent config file name via config option of agent.
#
# e.g.
# javaagent:objectgridRoot/lib/objectgrid.jar=config=myOverrideConfigFile
#
# The instrumentation definition, including transformation domain, @Exclude, and  @FieldAccessEntity can be overriden individually
# by corresponding designated agent options.
# Designated agent options include:
#    include              -> used to override instrumentation domain definition that is the first part of the config file
#    exclude              -> used to override @Exclude definition
#    fieldAccessEntity    -> used to override @FieldAccessEntity definition
#
# Each agent option should be separated by ";"
# Within the agent option, the package or class should be seperated by ","
#
# The following is an example that does not override the config file name:
#    -javaagent:objectgridRoot/lib/objectgrid.jar=include=includedPackage;exclude=excludedPackage;fieldAccessEntity=package1,package2
#
##############################

@Exclude
com.excludedPackage
com.excludedClass

@FieldAccessEntity
```

パフォーマンスの考慮

パフォーマンスを向上させるために、変換ドメインおよびフィールド・アクセス・エンティティー・ドメインを指定します。

分散環境におけるエンティティー・マネージャー

ローカルのスタンドアロン型 eXtreme Scale での EntityManager の使用に加え、分散および区画化された WebSphere eXtreme Scale でも EntityManager API を使用できます。主な違いは、リモートのeXtreme Scale へのアクセスまたは接続が行われる方法です。リモートの eXtreme Scale との接続が確立された後は、Session オブジェクトからエンティティー・マネージャーにアクセスすること、および EntityManager API を使用することに変わりはありません。

必須構成ファイル

以下に示した XML 構成ファイルが必要です。

- ObjectGrid 記述子 XML ファイル
- エンティティー記述子 XML ファイル
- デプロイメントまたはグリッド記述子 XML ファイル

始動時にどのエンティティーと BackingMap をホストするのかをサーバーに指示する必要があります。ObjectGrid XML およびエンティティー XML ファイルを作成します。

エンティティー・メタデータ記述子ファイルには、使用されるエンティティーの記述が含まれています。少なくとも、エンティティー・クラスおよび名前を指定する必要があります。Java Platform, Standard Edition 5 環境で稼働している場合、eXtreme Scale は、エンティティー・クラスとそのアノテーションを自動的に読み取ります。エンティティー・クラスにアノテーションがない場合、または何らかのオーバーライドが必要な場合には、追加の XML 属性を定義できます。以下の XML 構成スニペットを使用して、eXtreme Scale をエンティティーと共に定義できます。このスニペットでは、`bookstore` という名前の eXtreme Scale と、関連付ける `order` という名前のバックアップ・マップがサーバーによって作成されます。このスニペットは `entity.xml` ファイルを参照します。この例では、`entity.xml` ファイルに含まれているエンティティーは order エンティティーの 1 つのみです。

```
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">

 <objectGrids>
    <objectGrid name="bookstore" entityMetadataXMLFile="entity.xml">
     <backingMap name="Order"/>
    </objectGrid>
</objectGrids>

</objectGridConfig>
```

図 1. objectgrid.xml

この `objectgrid.xml` ファイルは、entityMetadataXMLFile 属性を使用して `entity.xml` を参照しています。このファイルのロケーションは、`objectgrid.xml` ファイルのロケーションに対して相対的です。`entity.xml` ファイルの例を以下に示します。

```
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">
 <entity class-name="com.ibm.websphere.tutorials.objectgrid.em.distributed.step1.Order" name="Order"/>
</entity-mappings>
```

図 2. entity.xml

eXtreme Scale サーバーの開始方法については、*管理ガイドの WebSphere eXtreme Scale サーバー処理の開始* を参照してください。そこでは、deployment.xml と objectgrid.xml の両方のファイルを使用して、カタログ・サーバーを開始しています。

分散 eXtreme Scale サーバーへの接続

以下のコードは、同じコンピューター上にあるクライアントとサーバー用の接続メカニズムを有効にします。

```
String catalogEndpoints="localhost:2809";
URL clientOverrideURL= new URL("file:etc/emtutorial/distributed/step1/objectgrid.xml");
ClientClusterContext clusterCtx = ogMgr.connect(catalogEndpoints, null, clientOverrideURL);
ObjectGrid objectGrid=ogMgr.getObjectGrid(clusterCtx, "bookstore");
```

図 3. 分散サーバーへの接続

上のコード・スニペットで、リモート eXtreme Scale サーバーへの参照に注意してください。接続が確立した後、EntityManager API 操作 (persist、update、remove、find などのメソッド) を起動できます。

重要: エンティティーを使用している場合、クライアントのオーバーライド ObjectGrid 記述子 XML ファイルを connect メソッドに渡してください。ヌル値が clientOverrideURL プロパティーに渡され、クライアントのディレクトリー構造がサーバーと異なると、クライアントは、ObjectGrid またはエンティティー記述子 XML ファイルを見つけることができない場合があります。最低限できることは、サーバーの ObjectGrid およびエンティティー XML ファイルをクライアントにコピーすることです。

クライアントおよびサーバー・サイドのスキーマ

サーバー・サイド・スキーマは、サーバー上のマップに保管されるデータのタイプを定義します。クライアント・サイド・スキーマは、サーバー上のスキーマからアプリケーション・オブジェクトへのマッピングです。例えば、以下のようなサーバー・サイド・スキーマもあります。

```
@Entity
class ServerPerson
{
  @Id String ssn;
  String firstName;
  String surname;
  int age;
  int salary;
}
```

図 4. ServerPerson.java

クライアントには、以下の例に示しているようなアノテーション付きのオブジェクトもあります。

```
@Entity(name="ServerPerson")
class ClientPerson
{
  @Id @Basic(alias="ssn") String socialSecurityNumber;
  String surname;
}
```

図 5. ClientPerson.java

このクライアントは、サーバー・サイド・エンティティーを受け取り、そのエンティティーのサブセットをクライアント・オブジェクトに射影します。この射影により、クライアント側の処理能力とメモリーを節約できます。その理由は、クライアントは、サーバー・サイド・エンティティーに入っているすべての情報ではなく、クライアントが必要とする情報だけを保有するからです。異なるアプリケーションは、すべてのアプリケーションにデータ・アクセスのためのクラス・セットを強制的に共用させる代わりに、それぞれ独自のオブジェクトを使用することができます。

スキーマの発生元

アプリケーションが Java SE 5 を使用している場合は、アノテーションを使用して、アプリケーションをオブジェクトに追加できます。エンティティー・マネージャーは、それらのオブジェクトのアノテーションからスキーマを読み取ることができます。エンティティー・マネージャーは、調べるオブジェクトがいずれかを知っておく必要があります。アプリケーションは、`entity.xml` ファイルを使用して、これらのオブジェクトについて eXtreme Scale ランタイムに知らせます。このファイルは、`objectgrid.xml` ファイルから参照されます。`entity.xml` ファイルには、すべてのエンティティーがリストされています。このファイルは、エンティティーごとにクラス名またはスキーマのいずれかを指定します。クラス名が指定されている場合には、それらのクラスから Java SE 5 のアノテーションを読み取って、スキーマを判別しようとします。アノテーション付きでないクラス・ファイルの場合、スキーマは XML ファイルから取得されます。この XML ファイルは、すべての属性、キー、およびリレーションシップをエンティティーごとに指定する場合に使用されます。

スタンドアロンの eXtreme Scale の場合、XML ファイルは不要です。プログラムは eXtreme Scale 参照を取得し、ObjectGrid.registerEntities メソッドを呼び出して、Java SE 5 のアノテーションを付けられたクラスのリストまたは XML ファイルを指定します。

ランタイムは、この XML ファイルまたはアノテーション付きクラスのリストを使用して、エンティティー名、属性名とタイプ、キー・フィールドとタイプ、およびエンティティー間のリレーションシップを見つけます。eXtreme Scale がサーバーで実行している場合、またはスタンドアロン・モードで実行している場合は、各エンティティーから付けられた名前を持つマップが自動的に作成されます。アプリケーション、または Spring などの注入フレームワークのいずれかによって設定された、`objectgrid.xml` ファイルまたは API を使用して、これらのマップをさらにカスタマイズすることができます。

エンティティー・メタデータ記述子ファイル

メタデータ記述子ファイルについて詳しくは、 65 ページの『`emd.xsd` ファイル』を参照してください。

関連資料

API 資料: EntityManager インターフェース

81 ページの『EntityTransaction インターフェース』
EntityTransaction インターフェースを使用すると、トランザクションを区別できます。

エンティティー照会キュー

照会キューを使用して、アプリケーションはエンティティーに対し、照会によって限定されるキューをサーバー・サイドまたはローカルの eXtreme Scale に作成できます。照会結果のエンティティーは、このキューに保管されます。現在、照会キューは、ペシミスティック・ロック・ストラテジーを使用しているマップでのみサポートされます。

照会キューは複数のトランザクション/クライアントによって共用されます。照会キューが空になると、このキューに関連付けられたエンティティー照会が再実行され、新しい結果がキューに追加されます。照会キューは、エンティティー照会ストリングとパラメーターによって一意的に識別されます。 1 つの ObjectGrid インスタンス内に存在する各固有の照会キューのインスタンスは 1 つのみです。追加情報については、EntityManager API 資料を参照してください。

照会キューの例

次の例は、照会キューの使用法を示します。

```
/**
 * Get a unassigned question type task
 */
private void getUnassignedQuestionTask() throws Exception {
    EntityManager em = og.getSession().getEntityManager();
    EntityTransaction tran = em.getTransaction();

    QueryQueue queue = em.createQueryQueue("SELECT t FROM Task t
    WHERE t.type=?1 AND t.status=?2", Task.class);
    queue.setParameter(1, new Integer(Task.TYPE_QUESTION));
    queue.setParameter(2, new Integer(Task.STATUS_UNASSIGNED));

    tran.begin();
    Task nextTask = (Task) queue.getNextEntity(10000);
    System.out.println("next task is " + nextTask);
    if (nextTask != null) {
        assignTask(em, nextTask);
    }
    tran.commit();
}
```

上記の例は、最初にエンティティー照会ストリング ″SELECT t FROM Task t WHERE t.type=?1 AND t.status=?2″ を使用して QueryQueue を作成しています。その次に、QueryQueue オブジェクトのパラメーターを設定しています。この照会キューは、タイプが ″question″ のすべての ″unassigned″ (未割り当て) タスクを示します。 QueryQueue オブジェクトは、エンティティー Query オブジェクトに非常によく似ています。

QueryQueue が作成されると、エンティティー・トランザクションが開始され、getNextEntity メソッドが呼び出されます。このメソッドは、タイムアウト値が 10 秒に設定され、次に使用可能なエンティティーを取得します。エンティティーが取

得されると、それは assignTask メソッドで処理されます。assignTask は Task エンティティー・インスタンスを変更し、状況を ″assigned″ (割り当て済み) に変更します。これにより、このエンティティーはもはや QueryQueue のフィルターに一致しなくなるため、事実上キューから削除されます。割り当てが終わると、トランザクションがコミットされます。

この簡単な例からわかるように、照会キューはエンティティー照会に似ています。しかし、両者には次のような違いがあります。

1. 照会キュー内のエンティティーは、反復方式で取得できます。取得するエンティティーの数は、ユーザー・アプリケーションが決定します。例えば、QueryQueue.getNextEntity(timeout) が使用された場合、取得されるエンティティーは 1 つのみです。QueryQueue.getNextEntities(5, timeout) が使用された場合は、5 つのエンティティーが取得されます。分散環境では、エンティティーの数によって、サーバーからクライアントへ転送されるバイト数が直接決まります。
2. エンティティーが照会キューから取得される際、そのエンティティーには U ロックがかけられるため、他のトランザクションはアクセスできません。

ループでのエンティティーの取得

エンティティーをループで取得できます。以下に、未割り当て (UNASSIGNED) の質問 (QUESTION) タイプのすべてのタスクを完了させる方法の例を示します。

```
/**
 * Get all unassigned question type tasks
 */
private void getAllUnassignedQuestionTask() throws Exception {
    EntityManager em = og.getSession().getEntityManager();
    EntityTransaction tran = em.getTransaction();

    QueryQueue queue = em.createQueryQueue("SELECT t FROM Task t WHERE
   t.type=?1 AND t.status=?2", Task.class);
    queue.setParameter(1, new Integer(Task.TYPE_QUESTION));
    queue.setParameter(2, new Integer(Task.STATUS_UNASSIGNED));

    Task nextTask = null;

    do {
        tran.begin();
        nextTask = (Task) queue.getNextEntity(10000);
        if (nextTask != null) {
            System.out.println("next task is " + nextTask);
        }
        tran.commit();
    } while (nextTask != null);
}
```

エンティティー・マップ内に未割り当ての質問タイプのタスクが 10 個あった場合、ユーザーは、10 個のエンティティーがコンソールにプリントされると予想したでしょう。しかし、このサンプルを実行すると、予想に反して、プログラムは永久に終了しません。

照会キューが作成され、getNextEntity が呼び出されると、キューに関連付けられたエンティティー照会が実行され、キューには 10 件の結果が追加されます。getNextEntity が呼び出されると、1 つのエンティティーがキューから取り出されます。getNextEntity 呼び出しが 10 回実行されると、キューは空になります。エンテ

ィティー照会が自動的に再実行されます。これら 10 個のエンティティーはまだ存在し、照会キューのフィルター条件に一致するため、それらは再度キューに追加されます。

次の行を println() ステートメントの後に追加すれば、10 個のエンティティーのみがプリントされるようになります。

```
em.remove(nextTask);
```

すべての区画にデプロイされる照会キュー

分散 eXtreme Scale では、照会キューを 1 つの区画またはすべての区画に作成できます。照会キューをすべての区画に作成する場合、各区画に 1 つの照会キュー・インスタンスが存在します。

クライアントは、QueryQueue.getNextEntity または QueryQueue.getNextEntities メソッドを使用して次のエンティティーを取得しようとするとき、要求を区画の 1 つに送信します。クライアントは、照合要求とピン要求をサーバーに送信します。

- 照合要求では、クライアントが要求をある区画に送信すると、すぐにサーバーから応答が返されます。エンティティーがキュー内にある場合、サーバーはエンティティーを付けて応答を返します。エンティティーがない場合、サーバーはエンティティーなしで応答を返します。いずれの場合も、サーバーは即時に応答を返します。
- ピン要求では、クライアントが要求をある区画に送信すると、サーバーは、エンティティーが使用可能になるまで待機します。エンティティーがキュー内にある場合、サーバーはエンティティーを付けて即時に応答を返します。エンティティーがない場合、サーバーは、エンティティーが使用可能になるか、または要求がタイムアウトになるまでキューで待機します。

すべての区画 (n 個) にデプロイされる照会キューのエンティティーを取得する方法の例を以下に示します。

1. QueryQueue.getNextEntity または QueryQueue.getNextEntities メソッドが呼び出されると、クライアントは 0 から n-1 の中からランダムに区画番号を選出します。
2. クライアントは照合要求を、そのランダムに選出した区画に送信します。
   - エンティティーが使用可能な場合は、エンティティーを返すことで、QueryQueue.getNextEntity または QueryQueue.getNextEntities メソッドは終了します。
   - エンティティーが使用不可で、かつそれがアクセスされていない最後の区画ではない場合、クライアントは照合要求を次の区画に送信します。
   - エンティティーが使用不可で、かつそれがアクセスされていない最後の区画だった場合、クライアントは代わりにピン要求を送信します。
   - 最後の区画に送信されたピン要求がタイムアウトになり、まだ使用可能なデータが存在しない場合、クライアントは、最後の試みとして、照合要求をもう 1 回すべての区画に順番に送信します。結果、以前の区画に使用可能なエンティティーがあれば、クライアントはそれを取得できます。

サブセット・エンティティーおよび非エンティティーのサポート

エンティティー・マネージャーに QueryQueue オブジェクトを作成するメソッドは、次のとおりです。

```
public QueryQueue createQueryQueue(String qlString, Class entityClass);
```

照会キュー内の結果は、メソッドの 2 番目のパラメーターで定義されたオブジェクトである Class entityClass に射影されます。

このパラメーターが指定された場合、クラスには、照会ストリングで指定されたものと同じエンティティー名が必要です。これは、エンティティーをサブセット・エンティティーに射影する場合に便利です。エンティティー・クラスにヌル値が使用された場合は、結果には何も射影されません。マップに保管される値は、エンティティー・タプル・フォーマットになります。

クライアント・サイドのキー競合

分散 eXtreme Scale 環境の場合、ペシミスティック・ロック・モードを使用する eXtreme Scale マップでのみ照会キューがサポートされます。したがって、クライアント・サイドにニア・キャッシュは存在しません。しかし、クライアントはトランザクション・マップ内にデータ (キーと値) を保持している可能性があります。このため、サーバーから取得されたエンティティーが、既にトランザクション・マップ内にあるエントリーと同じキーを共有していた場合、キー競合につながる可能性があります。

キー競合が発生すると、eXtreme Scale クライアント・ランタイムは、次の規則に従って、例外をスローするか、またはサイレントにデータをオーバーライドします。

1. 競合したキーが、照会キューに関連付けられたエンティティー照会で指定されたエンティティーのキーだった場合は、例外がスローされます。この場合、トランザクションはロールバックされ、このエンティティー・キーに対する U ロックはサーバー・サイドで解除されます。
2. そうでない場合、競合したキーがエンティティー・アソシエーションのキーであれば、トランザクション・マップ内のデータは警告なしでオーバーライドされます。

キー競合は、トランザクション・マップ内にデータが存在する場合のみ発生します。すなわち、それが発生するのは、既にダーティーな (新規データが挿入されたか、データが更新された) トランザクション内で getNextEntity または getNextEntities 呼び出しが呼び出されたときに限られます。アプリケーションでキー競合を発生させないようにするには、常にダーティーでないトランザクション内で getNextEntity または getNextEntities を呼び出す必要があります。

クライアント障害

クライアントは、getNextEntity または getNextEntities 要求をサーバーに送信した後、以下のような理由で失敗することがあります。

1. クライアントが要求をサーバーに送信してからダウンする。
2. クライアントが 1 つ以上のエンティティーをサーバーから取得した後でダウンする。

最初のケースでは、サーバーは応答をクライアントに送信しようとするときに、クライアントのダウンをディスカバーします。 2 番目のケースでは、クライアントが 1 つ以上のエンティティーをサーバーから取得すると、それらのエンティティーに X ロックがかけられます。クライアントがダウンすると、トランザクションは最終的にタイムアウトになり、X ロックは解放されます。

ORDER BY 文節を使用する照会

通常、照会キューでは ORDER BY 文節が守られません。照会キューから getNextEntity または getNextEntities を呼び出すと、エンティティーが順序どおりに返される保証はありません。その理由は、区画間でエンティティーを正しい順序にすることができないためです。照会キューがすべての区画にデプロイされるケースでは、getNextEntity または getNextEntities 呼び出しが実行されると、要求を処理する区画がランダムに選出されます。このため、順序は保証されません。

照会キューが単一区画にデプロイされる場合は、ORDER BY が守られます。

関連資料

API 資料: EntityManager インターフェース

『EntityTransaction インターフェース』
EntityTransaction インターフェースを使用すると、トランザクションを区別できます。

EntityTransaction インターフェース

EntityTransaction インターフェースを使用すると、トランザクションを区別できます。

目的

トランザクションを区別するには、エンティティー・マネージャー・インスタンスに関連付けられた EntityTransaction インターフェースを使用できます。エンティティー・マネージャーの EntityTransaction インスタンスを取得するには、EntityManager.getTransaction メソッドを使用します。各 EntityManager インスタンスおよび EntityTransaction インスタンスは、Session に関連付けられます。トランザクションは、EntityTransaction か Session のいずれかを使用して区別できます。EntityTransaction インターフェースのメソッドには、チェック例外はありません。タイプ PersistenceException またはそのサブクラスの実行時例外のみが発生します。

EntityTransaction インターフェースに関して詳しくは、API 資料の EntityTransaction インターフェースを参照してください。

関連概念

56 ページの『EntityManager API』
EntityManager API は、関連したオブジェクトからなる複雑なグラフを宣言したり、そのようなグラフと対話するための簡単な方法を提供することにより、eXtreme Scale キャッシュとの対話を単純化します。

77 ページの『エンティティー照会キュー』
照会キューを使用して、アプリケーションはエンティティーに対し、照会によって限定されるキューをサーバー・サイドまたはローカルの eXtreme Scale に作成できます。照会結果のエンティティーは、このキューに保管されます。現在、照会キューは、ペシミスティック・ロック・ストラテジーを使用しているマップでのみサポートされます。

74 ページの『分散環境におけるエンティティー・マネージャー』
ローカルのスタンドアロン型 eXtreme Scale での EntityManager の使用に加え、分散および区画化された WebSphere eXtreme Scale でも EntityManager API を使用できます。主な違いは、リモートのeXtreme Scale へのアクセスまたは接続が行われる方法です。リモートの eXtreme Scale との接続が確立された後は、Session オブジェクトからエンティティー・マネージャーにアクセスすること、および EntityManager API を使用することに変わりはありません。

エンティティー・インスタンスのライフサイクル

各エンティティー・インスタンスは、ライフサイクルの中でさまざまな状態に変化します。

エンティティー・インスタンスのライフサイクル

エンティティー・インスタンスには、以下の状態があります。

- **新規**: eXtreme Scale キャッシュに存在せず、新規に作成されたエンティティー・インスタンス。
- **管理対象**: eXtreme Scale キャッシュに存在し、エンティティー・マネージャーを使用して取得または永続化されるエンティティー・インスタンス。エンティティーを管理対象状態にするには、アクティブなトランザクションに関連付ける必要があります。
- **切り離し済み**: eXtreme Scale キャッシュに存在しているが、アクティブなトランザクションには関連付けられていないエンティティー・インスタンス。
- **除去済み**: eXtreme Scale キャッシュから除去されたか、トランザクションがフラッシュまたはコミットされるときにキャッシュから除去されたか、除去される予定のエンティティー・インスタンス。
- **無効化**: eXtreme Scale キャッシュで無効にされたか、トランザクションがフラッシュまたはコミットされるときにキャッシュで無効にされたか、無効にされる予定のエンティティー・インスタンス。

エンティティーの状態が変化するときは、ライフサイクル・コールバック・メソッドを起動できます。

以下のセクションでは、新規、管理対象、切り離し済み、除去済み、および無効化の状態がエンティティーに適用されるときの各状態の意味について詳細に説明します。

エンティティーの検索

EntityManager インターフェースは、トランザクション区分を使用する検索操作と使用しない検索操作をサポートします。

```
em.getTransaction().begin();
Order o=(Order)em.find(Order.class,"ASD100");
// Entity instance "o" is now managed
em.getTransaction().commit()
// Entity instance "o" is now detached.
```

図 6. 管理対象エンティティーの例

```
Order o=(Order)em.find(Order,class,"ASD100");
// Detached Entity instance "o". Not within the transaction boundary
```

図 7. 切り離し済みエンティティーの例

エンティティーの永続化

EntityManager インターフェースの persist メソッドを使用して、エンティティーを永続化できます。永続化できるのは、新規エンティティー・インスタンスのみで、それをサンプル・コードで記述しなければなりません。

```
//A customer with Id "10" exists in the eXtreme Scale cache, and the customer
//is placing a new Order.
Order newOrder = createNewOrder();
// Assuming this method sets relationships between Item, OrderLine and Order
em.getTransaction().begin();
Customer customer10=(Customer)em.find(Customer.class, "10");
// entity instance customer10 is now managed
newOrder.customer=customer10;
//persist newOrder. It is in "new" state
em.persist(newOrder);
// newOrder is now in "Managed" state
// Transaction commit operation persists Order, OrderLines and Item.
// Since the Customer already exists and is in Managed state, the persist
// operation ignores the Customer entity in this case.
// If customer object values are modified, the object is updated in
// the eXtreme Scale Cache.
em.getTransaction().commit();
//At this point, all instances are detached.

//The customer places another order, and this time the code uses the detached
//entity instance. This is an Error
// and an Exception is thrown.


Order newOrder2 = createNewOrder();
// Assuming this method sets relationships between Item, OrderLine and Order
em.getTransaction().begin();
newOrder.customer=customer10; // using a detached customer10 instance.
em.persist(newOrder2);
// The Commit call persists the Order in ObjectGrid. The customer10
// instance already exists in the eXtreme Scale, but since the instance
// is in detached state, the commit operation throws an
// EntityExistsException for the Customer Object.
em.getTransaction().commit();
```

エンティティーを永続化するときは、次の点に注意してください。

- 新規エンティティーのみ永続化できます。エンティティーは、em.persist(object) の呼び出し後は管理対象状態になります。エンティティーは、トランザクションがコミットまたはロールバックされるまで、管理対象状態を維持します。これは、本質的には挿入操作です。
- 既に管理対象状態にあるエンティティーに対するパーシスト操作は無視され、すべての値の変更が更新されます。トランザクションが最初にロールバックされた場合、persist メソッドの呼び出しは、更新操作と同等になります。
- CascadeType の値が PERSIST または ALL の場合、操作はアソシエーションにカスケードされます。
- 新規作成されたエンティティーまたは切り離し済みのエンティティーが永続化される場合、eXtreme Scale キャッシュにキーが存在すると EntityExistsException 例外になります。永続化は挿入操作と似ていて、重複キーが原因で例外が生じます。

エンティティーの除去または無効化

エンティティーの除去または無効化には、persist メソッドと似た意味体系があります。エンティティーの除去では、エンティティーが eXtreme Scale とオペレーティング・システムの両方から除去されます。エンティティーの無効化は、エンティティーを eXtreme Scale から除去するだけで、ローダー・プラグインは起動されません。

- エンティティーを除去するには、エンティティーが管理対象状態でなければなりません。
- CascadeType の値が REMOVE または ALL の場合、除去操作はアソシエーションにカスケードされます。
- CascadeType の値が INVALIDATE または ALL の場合、無効化操作はアソシエーションにカスケードされます。
- 切り離し済みエンティティーを除去または無効化しようとすると、IllegalArgumentException 例外が発生します。
- エンティティーが新規の場合、エンティティーは除去操作または無効化操作で無視されます。カスケードは、CascadeType の値に基づいて守られます。

```
//A correct way of removing.
//The an Order with OrderNumber ASD100 exists and needs to be removed:
em.getTransaction().begin();
Order o=(Order)em.find(Order.class,"ASD100"); // Managed
em.remove(o);
em.getTransaction().commit();

//An incorrect way of removing:
Order o=(Order)em.find(Order.class,"ASD100");
// Detached. Not within transaction demarcation
em.getTransaction().begin();
em.remove(o); // Can throw an IllegalArgumentException exception
em.getTransaction().commit();
```

エンティティー・リスナーおよびコールバック・メソッド

アプリケーションは、エンティティーの状態が遷移した場合に通知を受けることができます。状態変更イベントに対しては、2 つのコールバック・メカニズムが存在します。1 つはエンティティー・クラスに定義されているライフサイクル・コール

バック・メソッドで、エンティティーの状態が変更されると必ず呼び出されます。もう 1 つはエンティティー・リスナーで、いくつかのエンティティーに登録できるのでより一般的になっています。

エンティティー・ライフサイクル・コールバック・メソッド

エンティティー・ライフサイクル・コールバック・メソッドは、エンティティー・クラスに定義でき、エンティティーの状態が変わると呼び出されます。こうしたメソッドは、エンティティー・フィールドの妥当性検査や、通常ではエンティティーで持続することのない過渡状態の更新で役立ちます。エンティティー・ライフサイクル・コールバック・メソッドは、エンティティーを使用していないクラスでも定義することができます。こうしたクラスは、複数のエンティティー・タイプに関連付けることができるエンティティー・リスナー・クラスです。ライフサイクル・コールバック・メソッドは、以下のようにメタデータ・アノテーションを使用しても、エンティティー・メタデータ XML 記述子ファイルを使用しても定義できます。

- **アノテーション**: ライフサイクル・コールバック・メソッドは、エンティティー・クラス内で PrePersist、PostPersist、PreRemove、PostRemove、PreUpdate、PostUpdate、および PostLoad アノテーションを使用して示すことができます。
- **エンティティー XML 記述子**: ライフサイクル・コールバック・メソッドは、アノテーションが使用可能でない場合は XML を使用して記述できます。

エンティティー・リスナー

エンティティー・リスナー・クラスは、エンティティーを使用しないクラスであり、1 つ以上のエンティティー・ライフサイクル・コールバック・メソッドを定義します。エンティティー・リスナーは、汎用の監査アプリケーションまたはロギング・アプリケーションで有用です。エンティティー・リスナーは、以下のようにメタデータ・アノテーションを使用しても、エンティティー・メタデータ XML 記述子ファイルを使用しても定義できます。

- **アノテーション**: EntityListeners アノテーションは、エンティティー・クラス上の 1 つ以上のエンティティー・リスナー・クラスを示す場合に使用できます。複数のエンティティー・リスナーが定義されている場合、それらが呼び出される順序は、EntityListeners アノテーションに指定されている順序によって決定されます。
- **エンティティー XML 記述子**: XML 記述子は、エンティティー・リスナーの呼び出し順序を指定するか、メタデータ・アノテーションに指定されている順序をオーバーライドするための代替方法として使用できます。

コールバック・メソッドの要件

アノテーションのどのようなサブセットまたは組み合わせでも、エンティティー・クラスまたはリスナー・クラスに指定できます。1 つのクラスは、同じライフサイクル・イベントに対する複数のライフサイクル・コールバック・メソッドを持つことができません。ただし、同じメソッドを複数のコールバック・イベントに使用することができます。エンティティー・リスナー・クラスには、引数を取らない public コンストラクターが必要です。エンティティー・リスナーはステートレスです。エンティティー・リスナーのライフサイクルは、指定されません。 eXtreme Scale はエンティティー継承をサポートしないため、コールバック・メソッドは、エンティティー・クラスでしか定義できず、スーパークラスでは定義できません。

コールバック・メソッド・シグニチャー

エンティティー・ライフサイクル・コールバック・メソッドは、エンティティー・リスナー・クラスで定義するか、エンティティー・クラスで直接定義するか、あるいはその両方で定義できます。エンティティー・ライフサイクル・コールバック・メソッドは、メタデータ・アノテーションを使用しても、エンティティー XML 記述子を使用しても定義できます。エンティティー・クラスとエンティティー・リスナー・クラスでコールバック・メソッドに使用されるアノテーションは、同じです。コールバック・メソッドのシグニチャーは、エンティティー・クラスで定義する場合と、エンティティー・リスナー・クラスで定義する場合とでは異なります。エンティティー・クラスまたはマップされたスーパークラスで定義されるコールバック・メソッドは、以下のシグニチャーを持ちます。

```
void <METHOD>()
```

エンティティー・リスナー・クラスで定義されるコールバック・メソッドは、以下のシグニチャーを持ちます。

```
void <METHOD>(Object)
```

Object 引数は、コールバック・メソッドの呼び出し対象のエンティティー・インスタンスです。 Object 引数は、java.lang.Object オブジェクトまたは実際のエンティティー・タイプとして宣言できます。

コールバック・メソッドには public、private、protected、または package レベルのアクセスが可能ですが、static または final は使用できません。

対応するタイプのライフサイクル・イベント・コールバック・メソッドを指定するために、以下のアノテーションが定義されます。

- com.ibm.websphere.projector.annotations.PrePersist
- com.ibm.websphere.projector.annotations.PostPersist
- com.ibm.websphere.projector.annotations.PreRemove
- com.ibm.websphere.projector.annotations.PostRemove
- com.ibm.websphere.projector.annotations.PreUpdate
- com.ibm.websphere.projector.annotations.PostUpdate
- com.ibm.websphere.projector.annotations.PostLoad

詳しくは、API 資料を参照してください。各アノテーションには、エンティティー・メタデータ XML 記述子ファイルで定義された同等の XML 属性があります。

ライフサイクル・コールバック・メソッドのセマンティクス

以下のように、異なるライフサイクル・コールバック・メソッドは、それぞれ異なる目的を持ち、エンティティー・ライフサイクルの異なるフェーズで呼び出されます。

PrePersist

エンティティーに対して、そのエンティティーがストアに対して永続化される前に呼び出されます。こうしたエンティティーには、カスケード操作のために永続化されているエンティティーが含まれます。このメソッドは、EntityManager.persist 操作のスレッドで呼び出されます。

PostPersist

エンティティーに対して、そのエンティティーがストアに対して永続化された後に呼び出されます。こうしたエンティティーには、カスケード操作のために永続化されているエンティティーが含まれます。このメソッドは、EntityManager.persist 操作のスレッドで呼び出されます。これは、EntityManager.flush または EntityManager.commit が呼び出された後で呼び出されます。

PreRemove

エンティティーに対して、そのエンティティーが除去される前に呼び出されます。こうしたエンティティーには、カスケード操作のために除去されたエンティティーが含まれます。このメソッドは、EntityManager.remove 操作のスレッドで呼び出されます。

PostRemove

エンティティーに対して、そのエンティティーが除去された後に呼び出されます。こうしたエンティティーには、カスケード操作のために除去されたエンティティーが含まれます。このメソッドは、EntityManager.remove 操作のスレッドで呼び出されます。これは、EntityManager.flush または EntityManager.commit が呼び出された後で呼び出されます。

PreUpdate

エンティティーに対して、そのエンティティーがストアに対して更新される前に呼び出されます。このメソッドは、トランザクション・フラッシュ操作またはコミット操作のスレッドで呼び出されます。

PostUpdate

エンティティーに対して、そのエンティティーがストアに対して更新された後に呼び出されます。このメソッドは、トランザクション・フラッシュ操作またはコミット操作のスレッドで呼び出されます。

PostLoad

エンティティーに対して、そのエンティティーがストアからロードされた後に呼び出されます。こうしたエンティティーには、アソシエーションによってロードされたエンティティーが含まれます。このメソッドは、EntityManager.find や照会などのロード操作のスレッドで呼び出されます。

ライフサイクル・コールバック・メソッドの重複

エンティティー・ライフサイクル・イベントに対して複数のコールバック・メソッドが定義されている場合、これらのメソッドの呼び出し順序は以下のとおりです。

1. **エンティティー・リスナーで定義されたライフサイクル・コールバック・メソッド**: エンティティー・クラスのエンティティー・リスナー・クラスで定義されたライフサイクル・コールバック・メソッドは、EntityListeners アノテーションまたは XML 記述子でエンティティー・リスナー・クラスが指定されているのと同じ順序で呼び出されます。
2. **リスナー・スーパー・クラス**: エンティティー・リスナーのスーパー・クラスで定義されたコールバック・メソッドは、子の前に呼び出されます。
3. **エンティティー・ライフサイクル・メソッド**: WebSphere eXtreme Scale はエンティティー継承をサポートしないため、エンティティー・ライフサイクル・メソッドはエンティティー・クラス内でしか定義できません。

例外

ライフサイクル・コールバック・メソッドで実行時例外が発生する場合があります。ライフサイクル・コールバック・メソッドの結果としてトランザクション内で実行時例外が発生した場合、そのトランザクションがロールバックされます。実行時例外となった後は、それ以上ライフサイクル・コールバック・メソッドが呼び出されません。

エンティティー・リスナーの例

このトピックには、エンティティー・リスナーの例が含まれています。

アノテーションを使用するエンティティー・リスナーの例

以下の例では、ライフサイクル・コールバック・メソッド呼び出しとその呼び出し順序を示しています。エンティティー・クラス Employee および EmployeeListener と EmployeeListener2 という 2 つのエンティティー・リスナーが存在しているものとします。

```
@Entity
@EntityListeners(EmployeeListener.class, EmployeeListener2.class)
public class Employee {
    @PrePersist
    public void checkEmployeeID() {
        ....
    }
}

public class EmployeeListener {
    @PrePersist
    public void onEmployeePrePersist(Employee e) {
        ....
    }
}

public class PersonListener {
    @PrePersist
    public void onPersonPrePersist(Object person) {
        ....
    }
}

public class EmployeeListener2 {
    @PrePersist
    public void onEmployeePrePersist2(Object employee) {
        ....
    }
}
```

Employee インスタンスで PrePersist イベントが発生した場合、以下のメソッドがこの順序で呼び出されます。

1. onEmployeePrePersist メソッド
2. onPersonPrePersist メソッド
3. onEmployeePrePersist2 メソッド
4. checkEmployeeID メソッド

XML を使用するエンティティー・リスナーの例

以下の例は、エンティティー記述子 XML ファイルを使用して、エンティティーでエンティティー・リスナーを設定する方法を示したものです。

```
<entity
    class-name="com.ibm.websphere.objectgrid.sample.Employee"
    name="Employee" access="FIELD">
    <attributes>
        <id name="id" />
        <basic name="value" />
    </attributes>
    <entity-listeners>
        <entity-listener
            class-name="com.ibm.websphere.objectgrid.sample.EmployeeListener">
            <pre-persist method-name="onListenerPrePersist" />
            <post-persist method-name="onListenerPostPersist" />
        </entity-listener>
    </entity-listeners>
    <pre-persist method-name="checkEmployeeID" />
</entity>
```

エンティティー Employee は、com.ibm.websphere.objectgrid.sample.EmployeeListener エンティティー・リスナー・クラスによって構成されています。このクラスには、2 つのライフサイクル・コールバック・メソッドが定義されています。onListenerPrePersist メソッドは PrePersist イベントに対応するもので、onListenerPostPersist メソッドは PostPersist イベントに対応するものです。また PrePersist イベントを listen するために、checkEmployeeID メソッドが Employee クラスで構成されています。

照会 API

WebSphere eXtreme Scale は、EntityManager API を使用したエンティティーの検索、および ObjectQuery API を使用した Java オブジェクトの検索用の柔軟な照会エンジンを提供します。

WebSphere eXtreme Scale の照会機能

eXtreme Scale 照会エンジンを使用すると、eXtreme Scale 照会言語を使用して、エンティティーまたはオブジェクト・ベースのスキーマで SELECT タイプの照会ができます。

この照会言語では、以下の機能が提供されます。

- 単一および多値結果
- 集約関数
- ソートおよびグループ化
- 結合
- 副照会を使用した条件式
- 名前付きおよび定位置パラメーター
- eXtreme Scale 索引の使用
- オブジェクト・ナビゲーションのパス式構文
- ページ編集

Query インターフェース

エンティティー照会の実行を制御する場合に、照会インターフェースを使用します。

EntityManager.createQuery(String) メソッドを使用して、Query を作成します。各照会インスタンスを、それが取り出された EntityManager インスタンスと共に複数回使用できます。

各照会の結果、1 つのエンティティーが生成されます。この場合、エンティティー・キーは、行 ID (型 long の) であり、エンティティー値には、SELECT 文節のフィールド結果が含まれています。各照会結果を、それ以降の照会で使用できます。

以下のメソッドは、com.ibm.websphere.objectgrid.em.Query インターフェースで使用できます。

public ObjectMap getResultMap()

getResultMap メソッドは SELECT 照会を実行し、結果を照会で指定した順序で ObjectMap オブジェクトに戻します。結果の ObjectMap は、現行のトランザクションに対してのみ有効です。

マップ・キーは、結果の数値であり、型 long で 1 から始まります。マップ値は、タイプ com.ibm.websphere.projector.Tuple であり、この場合、各属性および関連は、照会の select 文節内の順序位置に基づいて指定されます。このメソッドを使用して、マップ内に保管されている Tuple オブジェクトに対する EntityMetadata を取り出してください。

getResultMap メソッドは、複数の結果が存在する可能性がある場合に、照会結果のデータを取り出す、最も高速なメソッドです。結果のエンティティーの名前は、ObjectMap.getEntityMetadata() および EntityMetadata.getName() メソッドを使用して取り出すことができます。

例: 以下の照会では、2 つの行を返します。

```
String ql = SELECT e.name, e.id, d from Employee e join e.dept d WHERE d.number=5
 Query q = em.createQuery(ql);
 ObjectMap resultMap = q.getResultMap();
 long rowID = 1; // starts with index 1
 Tuple tResult = (Tuple) resultMap.get(new Long(rowID));
 while(tResult != null) {
     // The first attribute is name and has an attribute name of 1
     // But has an ordinal position of 0.
     String name = (String)tResult.getAttribute(0);
     Integer id = (String)tResult.getAttribute(1);

     // Dept is an association with a name of 3, but
     // an ordinal position of 0 since it's the first association.
     // The association is always a OneToOne relationship,
     // so there is only one key.
     Tuple deptKey = tResult.getAssociation(0,0);
     ...
     ++rowID;
     tResult = (Tuple) resultMap.get(new Long(rowID));

 }
```

public Iterator getResultIterator

getResultIterator メソッドは SELECT 照会を実行し、照会の結果を Iterator を使用して戻します。この場合、各結果は、Object (単一値照会の場合) または Object[]

(複数値照会の場合) のいずれかです。Object[] 結果内の値は、照会順序で保管されます。結果の Iterator は、現行のトランザクションに対してのみ有効です。

このメソッドは、EntityManager コンテキスト内の照会結果を取り出す場合に推奨されます。オプションの setResultEntityName(String) メソッドを使用して、結果のエンティティーを指定し、以降の照会で使用できるようにすることができます。

例: 以下の照会では、2 つの行を返します。

```
String ql = SELECT e.name, e.id, e.dept from Employee e WHERE e.dept.number=5
 Query q = em.createQuery(ql);
 Iterator results = q.getResultIterator();
 while(results.hasNext()) {
     Object[] curEmp = (Object[]) results.next();
     String name = (String) curEmp[0];
     Integer id = (Integer) curEmp[1];
     Dept d = (Dept) curEmp[2];
     ...
 }
```

public Iterator getResultIterator(Class resultType)

getResultIterator(Class resultType) メソッドは、SELECT 照会を実行し、エンティティー Iterator を使用して照会結果を戻します。エンティティーの型は、resultType パラメーターによって決定されます。結果の Iterator は、現行のトランザクションに対してのみ有効です。

EntityManager API を使用して結果のエンティティーにアクセスする場合は、このメソッドを使用してください。

例: 以下の照会では、1 つの事業部について、全従業員と、従業員が所属する部門を給与順に返します。給与の高い順に 5 人の従業員を印刷してから、同じ作業セット内の 1 つの部門のみから、従業員の作業を選択する場合は、以下のコードを使用します。

```
String string_ql = "SELECT e.name, e.id, e.dept from Employee e WHERE
  e.dept.division='Manufacturing' ORDER BY e.salary DESC";
Query query1 = em.createQuery(string_ql);
query1.setResultEntityName("AllEmployees");
Iterator results1 = query1.getResultIterator(EmployeeResult.class);
int curEmployee = 0;
System.out.println("Highest paid employees");
while (results1.hasNext() && curEmployee++ < 5) {
 EmployeeResult curEmp = (EmployeeResult) results1.next();
 System.out.println(curEmp);
 // Remove the employee from the resultset.
 em.remove(curEmp);
}

// Flush the changes to the result map.
em.flush();

// Run a query against the local working set without the employees we
// removed
String string_q2 = "SELECT e.name, e.id, e.dept from AllEmployees e
  WHERE e.dept.name='Hardware'";
Query query2 = em.createQuery(string_q2);
Iterator results2 = query2.getResultIterator(EmployeeResult.class);
System.out.println("Subset list of Employees");
while (results2.hasNext()) {
 EmployeeResult curEmp = (EmployeeResult) results2.next();
 System.out.println(curEmp);
}
```

public Object getSingleResult

getSingleResult メソッドは単一の結果を戻す SELECT 照会を実行します。

SELECT 文節に複数のフィールドが定義されている場合には、結果はオブジェクト配列となります。この場合、配列内の各エレメントは、照会の SELECT 文節内の順序位置に基づきます。

```
String ql = SELECT e from Employee e WHERE e.id=100"
 Employee e = em.createQuery(ql).getSingleResult();

 String ql = SELECT e.name, e.dept from Employee e WHERE e.id=100"
 Object[] empData = em.createQuery(ql).getSingleResult();
 String empName= (String) empData[0];
 Department empDept = (Department) empData[1];
```

public Query setResultEntityName(String entityName)

setResultEntityName(String entityName) メソッドは照会結果エンティティーの名前を指定します。

getResultIterator または getResultMap メソッドが呼び出されるたびに、ObjectMap を備えたエンティティーが動的に作成されて照会の結果を保持します。エンティティーが指定されていないか、またはヌルである場合、エンティティーおよび ObjectMap 名は自動的に生成されます。

すべての照会結果が、トランザクションの存続期間中に使用可能であるため、照会名は、単一トランザクション内で再使用することはできません。

public Query setPartition(int partitionId)

照会の経路指定先に区画を設定します。

このメソッドは、照会内のマップが区画化されており、エンティティー・マネージャーに、単一スキーマのルート・エンティティー区画に対するアフィニティーがない場合に、必要になります。

PartitionManager インターフェースを使用して、指定されたエンティティーのバックアップ・マップに対する区画の数を決定してください。

以下の表に、照会インターフェースを通して使用可能なその他のメソッドの概要を示します。

表 9. その他のメソッド

メソッド	結果
public Query setMaxResults(int maxResult)	取り出す結果の最大数を設定します。
public Query setFirstResult(int startPosition)	取り出す最初の結果の位置を設定します。
public Query setParameter(String name, Object value)	引数を、名前付きパラメーターにバインドします。
public Query setParameter(int position, Object value)	引数を、定位置パラメーターにバインドします。
public Query setFlushMode(FlushModeType flushMode)	照会が実行されるときに使用されるフラッシュ・モード・タイプを設定し、EntityManager に対して設定されたフラッシュ・モード・タイプをオーバーライドします。

eXtreme Scale 照会の要素

eXtreme Scale 照会エンジンを使用すると、eXtreme Scale キャッシュの検索について単一の照会言語を使用することができます。この照会言語は、ObjectMap オブジェクトや Entity オブジェクトに保管されている Java オブジェクトの照会が可能です。以下の構文を使用して照会ストリングを作成します。

eXtreme Scale 照会は、以下の要素を含むストリングです。

- 返すオブジェクトまたは値を指定する SELECT 文節。
- オブジェクト集合に名前を付ける FROM 文節。
- 集合に対する検索述部を含むオプションの WHERE 文節。
- オプションの GROUP BY および HAVING 文節 (eXtreme Scale 照会の集約関数を参照)。
- 結果の集合の順序付けを指定するオプションの ORDER BY 文節。

Java オブジェクト集合は、照会の FROM 文節で名前が使用されることで識別されます。

照会言語の各要素については、以下の関連トピックでより詳しく説明します。

- 111 ページの『ObjectGrid 照会の Backus-Naur Form』 構文
- 103 ページの『eXtreme Scale 照会のための参照』

以下のトピックでは、Query API の使用方法について説明しています。

- 98 ページの『EntityManager 照会 API』
- 『ObjectQuery API の使用』

ObjectQuery API の使用

ObjectQuery API は、ObjectMap API を使用して保管された ObjectGrid 内のデータを照会するためのメソッドを提供します。スキーマが ObjectGrid インスタンスで定義される場合、ObjectQuery API を使用して、オブジェクト・マップに保管されている異種のオブジェクトに対して照会を作成し、実行することができます。

照会とオブジェクト・マップ

ObjectMap API を使用して保管されたオブジェクトに対して、拡張された照会機能を使用できます。 これらの照会によって、非キー属性を使用してオブジェクトを取り出すことや、照会条件と一致するすべてのデータに、sum、avg、min、max などの単純な集計を実行することができます。アプリケーションは、Session.createObjectQuery メソッドを使用して照会を構成できます。このメソッドは、ObjectQuery オブジェクトを戻します。このオブジェクトはその後、照会結果を取得するための問い合わせを受けることができます。また、照会オブジェクトを使用すれば、照会を実行する前にカスタマイズすることも可能です。照会結果を戻す任意のメソッドが呼び出されると、照会は自動的に実行されます。

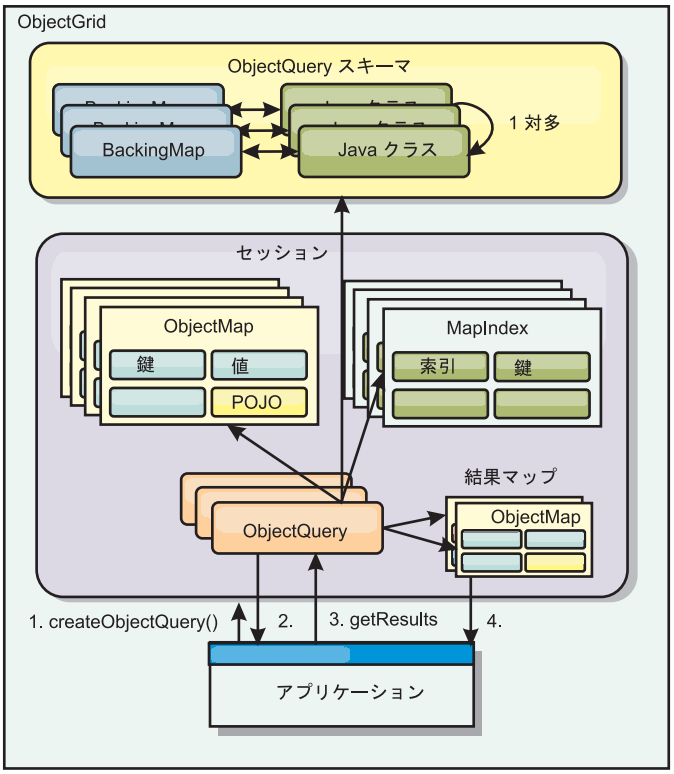

図 8. ObjectGrid オブジェクト・マップと照会との対話、および、スキーマがどのようにクラスに対して定義され、ObjectGrid マップと関連付けられるか

ObjectMap スキーマの定義

オブジェクト・マップは、さまざまな形式でオブジェクトを保管するために使用されるため、多くの場合、形式を認識しません。スキーマは、データのフォーマットを定義する ObjectGrid で定義される必要があります。スキーマは、以下のもので構成されます。

- ObjectMap に保管されているオブジェクトのタイプ
- ObjectMap 間のリレーションシップ
- それぞれの照会がオブジェクト (フィールドまたはプロパティー・メソッド) 内のデータ属性へのアクセスに使用するメソッド
- オブジェクト内の 1 次キー属性名。

詳細については、『ObjectQuery スキーマの構成』を参照してください。

スキーマをプログラマチックに作成する方法や、ObjectGrid 記述子 XML ファイルを使用して作成する方法の例については、「*製品概要*」で ObjectQuery に関するチュートリアルを参照してください。

ObjectQuery API を使用したオブジェクトの照会

ObjectQuery インターフェースを使用して、非エンティティー・オブジェクト (ObjectGrid ObjectMap に直接保管された異種のオブジェクト) の照会を行うことが

できます。ObjectQuery API には、キーワード・メカニズムおよび索引メカニズムを直接使用することなく、ObjectMap オブジェクトを簡単に検索する方法があります。

ObjectQuery から結果を取得するには、getResultIterator と getResultMap の 2 つのメソッドがあります。

getResultIterator を使用した照会結果の取得

照会結果とは、基本的に属性のリストのことです。照会で、y=z の場合に X から a,b,c を選択するとします。この照会では、a、b、および c を含む行のリストが戻されます。このリストは実際に、トランザクション有効範囲マップに保管されます。つまり、人工キーを各行と関連付け、各行で増加される整数を使用する必要があります。このマップは、ObjectQuery.getResultMap() メソッドを使用して取得します。以下のようなコードを使用して、各行のエレメントにアクセスすることができます。

```
ObjectQuery q = session.createQuery(
    "select c.id, c.firstName, c.surname from Customer c where c.surname=?1");

  q.setParameter(1, "Claus");

  Iterator iter = q.getResultIterator();
  while(iter.hasNext())
  {
    Object[] row = (Object[])iter.next();
    System.out.println("Found a Claus with id "
      + row[objectgrid: 0 ] + ", firstName: "
      + row[objectgrid: 1 ] + ", surname: "
      + row[objectgrid: 2 ]);
  }
```

getResultMap を使用した照会結果の取得

また、照会結果は、結果マップを直接使用して取得することもできます。以下の例は、一致するカスタマーの特定部分を取得する照会、および、結果行へのアクセス方法を示しています。 ObjectQuery オブジェクトを使用してデータにアクセスする場合は、生成される long 行 ID は非表示になりますので注意してください。その long 行は、ObjectMap を使用して結果にアクセスした場合にのみ表示されます。

トランザクションが完了すると、このマップは消去されます。また、マップは使用されたセッション、つまり通常はそのマップを作成したスレッドに対してのみ可視となります。マップは、行 ID を表す Long タイプのキーを使用します。マップに保管される値は、Object タイプか Object[] タイプのいずれかです。 Object[] タイプの場合、各エレメントは、選択された照会の文節にあるエレメントのタイプと同じになります。

```
ObjectQuery q = em.createQuery(
      "select c.id, c.firstName, c.surname from Customer c where c.surname=?1");
  q.setParameter(1, "Claus");
  ObjectMap qmap = q.getResultMap();
  for(long rowId = 0; true; ++rowId)
  {
    Object[] row = (Object[]) qmap.get(new Long(rowId));
    if(row == null) break;
```

```
    System.out.println(" I Found a Claus with id " + row[0]
      + ", firstName: " + row[1]
      + ", surname: " + row[2]);
  }
```

ObjectQuery の使用例については、「*製品概要*」内の ObjectQuery API に関するチュートリアルを参照してください。

ObjectQuery スキーマの構成

ObjectQuery は、スキーマまたは形状情報によってセマンティック検査を実行し、パス式を評価します。このセクションでは、スキーマを XML で、またはプログラマチックに定義する方法について説明します。

スキーマの定義

ObjectMap スキーマの定義は、ObjectGrid デプロイメント記述子 XML で、または標準の eXtreme Scale 構成手法を用いてプログラマチックに行います。スキーマの作成方法の例については、『ObjectQuery スキーマの構成』を参照してください。

スキーマ情報は Plain Old Java Object (POJO) を記述します。つまり、POJO を構成している属性、存在する属性のタイプ、属性が 1 次キー・フィールドなのか、単一値のリレーションシップまたは多値のリレーションシップなのか、それとも双方向リレーションシップなのかを記述します。ObjectQuery は、スキーマ情報に基づいてフィールド・アクセスまたはプロパティー・アクセスを使用します。

照会可能属性

スキーマが ObjectGrid で定義されていると、そのスキーマ内のオブジェクトはリフレクションを使用してイントロスペクトされ、照会に使用できる属性が決定されます。以下の属性タイプを照会できます。

- ラッパーを含む Java プリミティブ型
- java.lang.String
- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp
- java.util.Calendar
- byte[]
- java.lang.Byte[]
- char[]
- java.lang.Character[]
- J2SE 列挙型

上記以外の組み込みのシリアライズ可能な型も、照会結果に組み込むことができますが、照会の WHERE 文節または FROM 文節に組み込むことはできません。シリアライズ可能属性はナビゲート可能ではありません。

型がシリアライズ可能ではない場合、フィールドまたはプロパティーが静的な場合、またはフィールドが一時的なものである場合は、属性型をスキーマから除外できます。すべてのマップ・オブジェクトはシリアライズ可能でなければならないため、ObjectGrid は、オブジェクトからの永続可能な属性のみを含みます。それ以外のオブジェクトは無視されます。

フィールド属性

フィールドを使用してオブジェクトにアクセスするようスキーマが構成されている場合、すべてのシリアライズ可能な非一時的フィールドは自動的にスキーマに組み込まれます。照会内のフィールド属性を選択するには、クラス定義に記述されているとおりのフィールド ID 名を使用します。

スキーマには、すべの public、private、protected、および package protected フィールドが組み込まれます。

プロパティー属性

プロパティーを使用してオブジェクトにアクセスするようスキーマが構成されている場合、JavaBeans プロパティー命名規則に従うすべてのシリアライズ可能メソッドが自動的にスキーマに組み込まれます。照会用にプロパティー属性を選択するには、JavaBeans スタイルのプロパティー命名規則を使用します。

スキーマには、すべの public、private、protected および package protected プロパティーが組み込まれます。

以下のクラスでは、名前、誕生日、および有効性を示す属性がスキーマに追加されます。

```
public class Person {
  public String getName(){}
  private java.util.Date getBirthday(){}
  boolean isValid(){}
  public NonSerializableObject getData(){}
}
```

COPY_ON_WRITE の CopyMode を使用する場合、照会スキーマは、常にプロパティー・ベースのアクセスを使用しなければなりません。COPY_ON_WRITE では、マップからオブジェクトが取得される場合は常にプロキシー・オブジェクトを作成し、それらのオブジェクトにアクセスできるのはプロパティー・メソッドを使用する場合に限られます。そうしない場合、各照会結果がヌルに設定されます。

リレーションシップ

各リレーションシップは、スキーマ構成に明示的に定義する必要があります。リレーションシップの基数は、属性の型によって自動的に決定されます。属性が java.util.Collection インターフェースを実装している場合、リレーションシップは 1 対多または多対多のいずれかのリレーションシップです。

エンティティー照会とは異なり、キャッシュされている他のオブジェクトを参照している属性は、そのオブジェクトへの直接参照を保管することはできません。他の

オブジェクトへの参照は、そのオブジェクトを包含するオブジェクトのデータの一部としてシリアライズされます。代わりに、関連するオブジェクトへのキーを保管してください。

例えば、Customer と Order の間に、以下のような多対 1 のリレーションシップがあるとします。

誤。オブジェクト参照を保管しています。

```
public class Customer {
  String customerId;
  Collection<Order> orders;
}

public class Order {
  String orderId;
  Customer customer;
}
```

正。関連オブジェクトへのキー。

```
public class Customer {
  String customerId;
  Collection<String> orders;
}

public class Order {
  String orderId;
  String customer;
}
```

2 つのマップ・オブジェクトを 1 つに結合する照会を実行すると、キーは自動的に大きくなります。例えば、以下の照会は Customer オブジェクトを返します。

```
SELECT c FROM Order o JOIN Customer c WHERE orderId=5
```

索引の使用

ObjectGrid は、索引プラグインを使用して、マップに索引を追加します。照会エンジンは、com.ibm.websphere.objectgrid.plugins.index.HashIndex 型のスキーマ・マップ・エレメントで定義されている索引を自動的に組み込み、rangeIndex プロパティーは true に設定されます。索引の型が HashIndex ではなく、rangeIndex プロパティーが true に設定されていない場合、照会はその索引を無視します。スキーマに索引を追加する方法を示す例については、「*製品概要*」内の Object Query チュートリアルを参照してください。

EntityManager 照会 API

EntityManager API は、EntityManager API を使用して保管された ObjectGrid 内のデータを照会するためのメソッドを提供します。EntityManager 照会 API は、eXtreme Scale に定義された 1 つ以上のエンティティーに関する照会の作成と実行に使用されます。

エンティティーの照会と ObjectMap

eXtreme Scale に保管されたエンティティーの拡張照会機能が WebSphere Extended Deployment v6.1 で導入されました。これらの照会によって、非キー属性を使用してオブジェクトを取り出すことや、照会条件と一致するすべてのデータに、合計、平

均、最小、最大などの単純な集計を実行することができます。アプリケーションは、EntityManager.createQuery API を使用して照会を構成します。これにより、Query オブジェクトを戻した後、照会結果を取得するための問い合わせを受けることができます。また、照会オブジェクトを使用すれば、照会を実行する前にカスタマイズすることも可能です。照会結果を戻す任意のメソッドが呼び出されると、照会は自動的に実行されます。

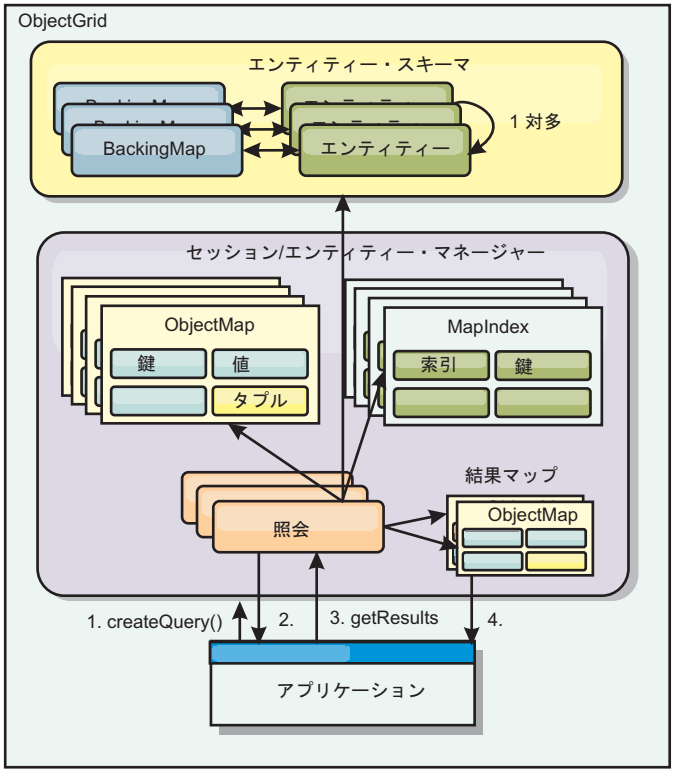

図 9. ObjectGrid オブジェクト・マップと照会との対話、および、エンティティー・スキーマがどのように定義され、ObjectGrid マップと関連付けられるか。

getResultIterator メソッドを使用した照会結果の取得

照会結果は、属性のリストです。照会で、y=z の場合に X から a,b,c を選択すると、a、b、および c を含む行のリストが戻されます。このリストは、トランザクション有効範囲マップに保管されます。これはつまり、人工キーが各行と関連付けられており、各行で増加される整数を使用する必要があることを意味します。このマップは、Query.getResultMap メソッドを使用して取得されます。 マップには、関連付けられているマップ内の各行について説明する、EntityMetaData があります。以下のようなコードを使用して、各行のエレメントにアクセスすることができます。

```
Query q = em.createQuery("select c.id, c.firstName, c.surname from Customer c where c.surname=?1");

  q.setParameter(1, "Claus");

  Iterator iter = q.getResultIterator();
  while(iter.hasNext())
  {
    Object[] row = (Object[])iter.next();
    System.out.println("Found a Claus with id "  + row[objectgrid: 0 ]
      + ", firstName: " + row[objectgrid: 1 ]
      + ", surname: " + row[objectgrid: 2 ]);
  }
```

getResultMap を使用した照会結果の取得

以下のコードは、一致するカスタマーの特定部分の取得、および、結果行へのアクセス方法を示しています。 Query オブジェクトを使用してデータにアクセスする場合は、生成される long 行 ID は非表示になります。その long は、ObjectMap を使用して結果にアクセスした場合にのみ表示されます。トランザクションが完了すると、このマップは消えます。 マップは、使用されたセッション、つまり通常はそのマップを作成したスレッドに対してのみ可視となります。マップは単一の属性、long 行 ID を持つキーのタプルを使用します。その値は、結果セット内の各列の属性を持つ別のタプルです。

以下は、これを示したサンプル・コードです。

```
Query q = em.createQuery("select c.id, c.firstName, c.surname from
Customer c where c.surname=?1");
q.setParameter(1, "Claus");
ObjectMap qmap = q.getResultMap();
Tuple keyTuple = qmap.getEntityMetadata().getKeyMetadata().createTuple();
for(long i = 0; true; ++i)
{
  keyTuple.setAttribute(0, new Long(i));
  Tuple row = (Tuple)qmap.get(keyTuple);
  if(row == null) break;
  System.out.println(" I Found a Claus with id "  + row.getAttribute(0)
    + ", firstName: " + row.getAttribute(1)
    + ", surname: " + row.getAttribute(2));
}
```

エンティティー結果イテレーターを使用した照会結果の取得

以下のコードは、通常のマップ API を使用して各結果行を取得する、照会とループを示しています。マップのキーは Tuple です。そのため、createTuple メソッドを使用して適切なタイプの 1 つを構成した結果は、keyTuple になります。rowIds を持つすべての行を、0 以上の値から取得しようとします。キーが見つからなかったことを示すヌルが戻された場合、ループは終了します。keyTuple の最初の属性が、検索する long になるように設定します。 get によって戻される値も、照会結果内の各列の属性を持つタプルです。その後、getAttribute を使用して、値タプルから各属性をプルします。

以下は、次のコードの断片です。

```
Query q2 = em.createQuery("select c.id, c.firstName, c.surname from Customer c where c.surname=?1");
q2.setResultEntityName("CustomerQueryResult");
q2.setParameter(1, "Claus");

Iterator iter2 = q2.getResultIterator(CustomerQueryResult.class);
while(iter2.hasNext())
{
  CustomerQueryResult row = (CustomerQueryResult)iter2.next();
  // firstName is the id not the firstName.
  System.out.println("Found a Claus with id " + row.id
    + ", firstName: " + row.firstName
    + ", surname: " + row.surname);
}

em.getTransaction().commit();
```

照会に ResultEntityName 値が指定されています。この値は、各行を特定の 1 つのオブジェクト、この例では CustomerQueryResult に射影することを、照会エンジンに指示します。クラスは次のとおりです。

```
@Entity
public class CustomerQueryResult {
 @Id long rowId;
 String id;
 String firstName;
 String surname;
};
```

最初のスニペットで、各照会行が Object[] ではなく CustomerQueryResult オブジェクトとして戻される点に注意してください。照会の結果列は、CustomerQueryResult オブジェクトに射影されます。結果を射影することは、実行時には少し遅くなりますが、読みやすさは優れています。照会結果エンティティーは、開始時に eXtreme Scale に登録されていてはなりません。エンティティーが登録されている場合、同じ名前のグローバル・マップが作成され、マップ名が重複していることを示すエラーによって照会は失敗します。

EntityManager を使用した単純照会

このトピックでは、EntityManager 照会 API を使用して単純照会を実行する方法を示します。

EntityManager 照会 API は、オブジェクトを照会する、SQL の他の照会エンジンにとても似ています。照会が定義されてから、各種の getResult メソッドを使用して、照会から結果が取り出されます。

以下の例は、「製品概要」にある EntityManager チュートリアルで使用されているエンティティーを参照しています。

単純照会の実行

次の例では、Claus という名字の顧客が照会されます。

```
em.getTransaction().begin();

  Query q = em.createQuery("select c from Customer c where c.surname='Claus'");

  Iterator iter = q.getResultIterator();
  while(iter.hasNext())
  {
    Customer c = (Customer)iter.next();
    System.out.println("Found a claus with id " + c.id);
  }

  em.getTransaction().commit();
```

パラメーターの使用

Claus という名字のすべての顧客の検索で、この照会を複数回使用する場合もあるので、名字を指定するパラメーターが使用されます。

定位置パラメーターの例

```
Query q = em.createQuery("select c from Customer c where c.surname=?1");
  q.setParameter(1, "Claus");
```

照会が複数回使用される場合、パラメーターの使用は非常に重要です。
EntityManager は、照会ストリングを構文解析して、照会の計画をビルドする必要が

あり、これにはコストがかかります。パラメーターを使用することで、EntityManager は照会の計画をキャッシュに入れるので、照会の実行にかかる時間が削減されます。

定位置パラメーターと、名前が指定されたパラメーターの両方が使用されます。

名前が指定されたパラメーターの例

```
Query q = em.createQuery("select c from Customer c where c.surname=:name");
  q.setParameter("name", "Claus");
```

パフォーマンスを改善するための索引の使用

顧客が何百万人もいる場合には、前述の照会では、顧客マップ内のすべての行をスキャンする必要があります。これは、あまり効率的ではありません。しかし、eXtreme Scale は、エンティティーの個々の属性に対する索引を定義するためのメカニズムを提供しています。照会では適宜、この索引が自動的に使用されるため、照会の速度が大幅に上がります。

エンティティー属性で @Index 注釈を使用すれば、索引付けする属性を非常に簡単に指定できます。

```
@Entity
public class Customer
{
  @Id String id;
  String firstName;
  @Index String surname;
  String address;
  String phoneNumber;
}
```

EntityManager は、名字属性に対する適切な ObjectGrid 索引を Customer エンティティー内に作成し、照会エンジンはこの索引を自動的に使用します。これにより、照会時間は大幅に短縮されます。

パフォーマンスを改善するためのページ編集の使用

Claus という名前の顧客が 100 万人いる場合には、100 万人の顧客を表示したページを表示するのは現実的ではありません。一度に 10 または 25 人の顧客を表示することになると考えられます。

Query setFirstResult メソッドおよび setMaxResults メソッドは、結果のサブセットのみを戻すため、役に立ちます。

ページ編集の例

```
Query q = em.createQuery("select c from Customer c where c.surname=:name");
  q.setParameter("name", "Claus");
  // Display the first page
  q.setFirstResult=1;
  q.setMaxResults=25;
  displayPage(q.getResultIterator());

  // Display the second page
  q.setFirstResult=26;
  displayPage(q.getResultIterator());
```

eXtreme Scale 照会のための参照

WebSphere eXtreme Scale は固有の言語を持ち、それによってユーザーはデータを照会することができます。

ObjectGrid 照会の FROM 文節

FROM 文節は、照会を適用するオブジェクトの集合を指定します。各集合は、抽象スキーマ名と識別変数 (範囲変数)、または一価または多値リレーションシップと識別変数を識別する集合メンバー宣言のいずれかによって識別されます。

概念的には、照会のセマンティクスは、まずタプルの一時的な集合を形成することであり、R と呼ばれます。タプルは、FROM 文節で識別される集合からの要素で構成されています。各タプルには、FROM 文節内の各集合からの要素が 1 つ含まれています。集合のメンバー宣言で指定された制約に従って、すべての可能な組み合わせが形成されます。パーシスタント・ストア内にレコードがないコレクションを識別するスキーマ名がある場合は、一時集合 R は空です。

FROM の使用例

DeptBean オブジェクトには、レコード 10、20、30 があります。EmpBean オブジェクトには、部門 10 に関連付けられたレコード 1、2、および 3 と、部門 20 に関連付けられたレコード 4 および 5 があります。部門 30 に関連付けられた従業員はいません。

```
FROM DeptBean d, EmpBean e
```

この文節によって、15 のタプルを持つ一時集合 R が形成されます。

```
FROM  DeptBean d,  DeptBean d1
```

この文節によって、9 のタプルを持つ一時集合 R が形成されます。

```
FROM  DeptBean d,  IN (d.emps) AS e
```

この文節によって、5 のタプルを持つ一時集合 R が形成されます。部門 30 には従業員がいないため、R 一時集合内にはありません。部門 10 は 3 回、部門 20 は 2 回、R 一時集合に含まれます。

IN(d.emps) as e を使用する代わりに、JOIN 述部を使用すると次のようになります。

```
FROM DeptBean d JOIN d.emps as e
```

一時集合が形成されると、WHERE 文節の検索条件が R 一時集合に適用され、新しい一時集合 R1 が形成されます。ORDER BY 文節と SELECT 文節が R1 に適用されて、最終的な結果セットが形成されます。

識別変数は、FROM 文節で IN 演算子またはオプションの AS 演算子を使用して宣言される変数です。

```
FROM DeptBean AS d,  IN (d.emps) AS e
```

これは、下記と同じです。

```
FROM DeptBean d, IN (d.emps) e
```

抽象スキーマ名として宣言される識別変数は、範囲変数と呼びます。前の照会では、「d」が範囲変数です。多値パス式として宣言される識別変数は、コレクション・メンバー宣言と呼びます。前の例では、「d」および「e」の値がコレクション・メンバー宣言です。

FROM 文節内での一価パス式の使用例を示します。

```
FROM EmpBean e, IN(e.dept.mgr) as m
```

ObjectGrid 照会の SELECT 文節

SELECT 文節の構文を、以下の例に示します。

```
SELECT { ALL | DISTINCT }  [ selection , ]*  selection

selection  ::= {single_valued_path_expression |
                identification_variable |
                OBJECT ( identification_variable) |
        aggregate_functions } [[ AS ] id ]
```

SELECT 文節は、以下の要素の 1 つ以上で構成されます。FROM 文節で定義される単一の識別変数、オブジェクト参照やオブジェクト値を評価する一価パス式、および集約関数。DISTINCT キーワードを使用し、重複参照を取り除くことができます。

スカラー副選択は、単一値を返す副選択です。

SELECT の使用例

従業員 John の収入を超える従業員をすべて検索します。

```
SELECT OBJECT(e) FROM EmpBean ej, EmpBean eWHERE  ej.name = 'John'  and
e.salary > ej.salary
```

収入が 20000 に満たない従業員が 1 人以上いる部門をすべて検索します。

```
SELECT DISTINCT e.dept  FROM EmpBean e where e.salary < 20000
```

照会には、任意の値を評価するパス式を含めることができます。

```
SELECT e.dept.name  FROM EmpBean e where e.salary < 20000
```

前の照会では、収入が 20000 に満たない従業員がいる部門の名前値の集合が返されます。

照会は、集約値を返すこともできます。

```
SELECT  avg(e.salary)  FROM EmpBean e
```

収入の低い従業員について、名前とオブジェクト参照を取得する照会は、次のようになります。

```
SELECT e.name as name, object(e) as emp from EmpBean e where e.salary <
50000
```

ObjectGrid 照会の WHERE 文節

WHERE 文節には、以下の要素で構成される検索条件が含まれています。検索条件が TRUE と評価されると、結果セットにタプルが追加されます。

ObjectGrid 照会のリテラル

ストリング・リテラルは、単一引用符で囲みます。ストリング・リテラル内にある単一引用符は、2 つの単一引用符で表します。例: 'Tom''s。

数値リテラルは、以下の任意の値が使用可能です。

- 57、-957、+66 などの厳密値
- Java long 型でサポートされる任意の値
- 57.5、-47.02 などの小数リテラル
- 7E3、-57.4E-2 などの概算数値

ブール・リテラルは TRUE および FALSE です。

一時リテラルは、属性のタイプに基づいて JDBC エスケープ構文の後に続きます。

- java.util.Date: yyyy-mm-ss
- java.sql.Date: yyyy-mm-ss
- java.sql.Time: hh-mm-ss
- java.sql.Timestamp: yyyy-mm-dd hh:mm:ss.f...
- java.util.Calendar: yyyy-mm-dd hh:mm:ss.f...

列挙型リテラルは、完全修飾列挙型クラス名を使用する Java 列挙型リテラル構文によって表されます。

ObjectGrid 照会の入力パラメーター

順序位置または変数名を使用して、入力パラメーターを指定することができます。入力パラメーターを使用して照会を記述することを強く推奨します。入力パラメーターを使用すると、ObjectGrid が実行アクションの間に照会計画をキャッチできるようになり、パフォーマンスが向上するためです。

入力パラメーターは、以下の型のいずれかが可能です。
Byte、Short、Integer、Long、Float、Double、BigDecimal、BigInteger、String、Boolean、Char、java.util.Date、java.sql.Date、java.sql.Time、java.sql.Timestamp、java.util.Calendar、Java SE 5 enum、Entity または POJO Object、または Java byte[] 形式のバイナリー・データ・ストリング。

入力パラメーターにヌル値を含めないでください。ヌル値の存在を検索するには、NULL 述部を使用してください。

定位置パラメーター

定位置入力パラメーターは、次のように疑問符 (?) の後ろに正数を付けたものを使用して定義します。

?[正整数]

定位置入力パラメーターは 1 から始まる番号が付けられており、照会の引数に対応しています。したがって、入力引数の数を超える入力パラメーターを照会に含めることはできません。

例: SELECT e FROM Employee e WHERE e.city = ?1 and e.salary >= ?2

名前付きパラメーター

名前付き入力パラメーターは、次の形式で変数名を使用して定義します。:[パラメーター名]

例: SELECT e FROM Employee e WHERE e.city = :city and e.salary >= :salary

ObjectGrid 照会の BETWEEN 述部

BETWEEN 述部は、ある値が他の 2 つの値の間にあるかどうかを調べます。

式 [NOT] BETWEEN 式 2 AND 式 3

例 1

```
e.salary BETWEEN 50000 AND 60000
```

これは、下記と同じです。

```
e.salary >= 50000  AND e.salary <= 60000
```

例 2

```
e.name NOT BETWEEN 'A' AND 'B'
```

これは、下記と同じです。

```
e.name < 'A'  OR e.name > 'B'
```

ObjectGrid 照会の IN 述部

IN 述部は、1 つの値を、値のセットと比較します。以下の 2 つの形式のいずれかを使用して IN 述部を使用できます。

```
expression [NOT] IN ( subselect )expression [NOT] IN  ( value1, value2,
.... )
```

ValueN の値は、リテラル値でも入力パラメーターでも構いません。式は、参照型に対する評価は行うことができません。

例 1

```
e.salary IN ( 10000, 15000 )
```

は、次の式と等価です。

```
( e.salary = 10000  OR  e.salary = 15000 )
```

例 2

```
e.salary IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

は、次の式と等価です。

```
e.salary = ANY   ( select  e1.salary from EmpBean e1 where e1.dept.deptno =
10)
```

例 3

```
e.salary NOT IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno =
10)
```

は、次の式と等価です。

```
e.salary <> ALL    ( select  e1.salary from EmpBean e1 where e1.dept.deptno
= 10)
```

ObjectGrid 照会の LIKE 述部

LIKE 述部は、ある特定のパターンのストリング値を検索します。

```
string-expression   [NOT]  LIKE  pattern    [ ESCAPE escape-character ]
```

パターン値は、ストリング型のストリング・リテラルまたはパラメーター・マーカーで、アンダースコアー (_) は任意の 1 文字を表し、パーセント (%) は空シーケンスを含む任意の文字シーケンスを表します。その他の文字はその文字自身を表します。エスケープ文字は、文字 _ および % の検索に使用できます。エスケープ文字は、ストリング・リテラルとしても、入力パラメーターとしても指定できます。

ストリング式がヌルの場合、結果は不明となります。

ストリング式とパターンの両方が空の場合は、結果は true となります。

例

```
'' LIKE '' is true
'' LIKE '%' is true
e.name LIKE '12%3' is true for '123' '12993' and false for '1234'
e.name LIKE 's_me' is true for 'some' and 'same', false for 'soome'
e.name LIKE '/_foo' escape '/' is true for '_foo', false for 'afoo'
e.name LIKE '//_foo' escape '/' is true for '/afoo' and for '/bfoo'
e.name LIKE '///_foo' escape '/' is true for '/_foo' but false for '/afoo'
```

ObjectGrid 照会の NULL 述部

NULL 述部は、ヌル値であるかの検査を行います。

```
{single-valued-path-expression | input_parameter} IS [NOT] NULL
```

例

```
e.name IS NULL
e.dept.name IS NOT NULL
e.dept IS NOT NULL
```

ObjectGrid 照会の EMPTY コレクション述部

EMPTY コレクション述部を使用して、コレクションが空であるかどうかを検査します。

多値リレーションシップが空であるかどうかを検査するには、次の構文を使用します。

```
collection-valued-path-expression  IS [NOT]  EMPTY
```

例

Empty コレクション述部。従業員のいない部門を検索する照会は次のようになります。

```
SELECT OBJECT(d) FROM DeptBean d  WHERE  d.emps IS EMPTY
```

ObjectGrid 照会の MEMBER OF 述部

以下の式は、一価パス式または入力パラメーターで指定されたオブジェクト参照が、指定した集合のメンバーであるかどうかを検査します。集合価パス式が空の集合を指定している場合、MEMBER OF 式の値は FALSE になります。

```
{ single-valued-path-expression | input_parameter } [ NOT ] MEMBER [ OF ]
collection-valued-path-expression
```

例

指定する部門番号のメンバーではない従業員を検索する照会は、次のようになります。

```
SELECT OBJECT(e) FROM EmpBean e , DeptBean d
WHERE e NOT MEMBER OF d.emps AND d.deptno = ?1
```

指定する部門番号のメンバーである管理者を持つ従業員を検索する照会は、次のようになります。

```
SELECT OBJECT(e) FROM EmpBean e, DeptBean d
WHERE e.dept.mgr MEMBER  OF d.emps  and d.deptno=?1
```

ObjectGrid 照会の EXISTS 述部

EXISTS 述部は、副選択によって指定された条件の有無を検査します。

```
EXISTS (副選択)
```

副選択から最低 1 つの値が返されると EXISTS の結果は true になり、値が返されない場合は結果は false になります。

EXISTS 述部を否定するには、述部の前に NOT 論理演算子を指定します。

例

1000000 を超える収入がある従業員が最低 1 人いる部門を返す照会は、次のようになります。

```
SELECT  OBJECT(d) FROM  DeptBean d
WHERE EXISTS ( SELECT  e  FROM IN (d.emps) e WHERE  e.salary > 1000000 )
```

従業員がいない部門を返す照会は次のようになります。

```
SELECT OBJECT(d) FROM DeptBean d
WHERE NOT EXISTS  ( SELECT e FROM IN (d.emps) e)
```

次の例に示すように、前の照会を書き換えることもできます。

```
SELECT OBJECT(d) FROM DeptBean d WHERE SIZE(d.emps)=0
```

ObjectGrid 照会の ORDER BY 文節

ORDER BY 文節は、結果集合内のオブジェクトの順序を指定します。以下に例を示します。

ORDER BY [order_element ,]* order_element order_element ::={ path-expression }[ASC | DESC]

パス式では、byte、short、int、long、float、double、char などのプリミティブ型、または Byte、Short、Integer、Long、Float、Double、BigDecimal、String、Character、java.util.Date、java.sql.Date、java.sql.Time、 java.sql.Timestamp、java.util.Calendar などのラッパー型の一価フィールドを指定する必要があります。ASC 順序要素は、結果を昇順に表示するよう指定します (デフォルト)。DESC 順序要素は、結果を降順に表示するよう指定します。

例

部門オブジェクトを返します。部門番号を降順で表示します。

```
SELECT OBJECT(d) FROM DeptBean d ORDER BY d.deptno DESC
```

従業員オブジェクトを返し、部門番号と部門名でソートします。

```
SELECT OBJECT(e) FROM EmpBean e ORDER BY e.dept.deptno ASC, e.name DESC
```

ObjectGrid 照会集約関数

集約関数は、1 セットの値を操作して単一のスカラー値を返します。これらの関数は、select メソッドおよび subselect メソッドで使用できます。以下に、集約の例を示します。

```
SELECT SUM (e.salary) FROM EmpBean e WHERE e.dept.deptno =20
```

この集約では、部門 20 の給料の合計を計算します。

集約関数は、AVG、COUNT、MAX、MIN、および SUM です。集約関数の構文を、以下の例で示します。

```
aggregation-function ( [ ALL | DISTINCT ] expression )
```

または、

```
COUNT( [ ALL | DISTINCT ] identification-variable )
```

DISTINCT オプションを使用すると、関数を適用する前に重複値が除去されます。ALL オプションは、デフォルトのオプションで、重複値は除去されません。NULL 値は集約関数の計算においては無視されますが、COUNT(identification-variable) 関数を使用する場合は無視されず、セット内のすべてのエレメントの数が返されます。

戻りの型の定義

MAX および MIN 関数は、すべての数値、ストリング、または日時のデータ型に適用でき、対応するデータ型を返します。SUM および AVG 関数は、入力として数値型を必要とします。AVG 関数は double 型を返します。SUM 関数は、入力型が integer 型の場合は long 型を返しますが、入力が Java BigInteger 型の場合は、Java BigInteger 型を返します。SUM 関数は、入力型が integer 型でない場合は double 型を返しますが、入力が Java BigDecimal 型の場合は、Java BigDecimal 型を返します。COUNT 関数は、コレクション以外のすべてのデータ型を入力でき、long 型を返します。

空集合に適用される場合は、SUM、AVG、MAX、および MIN 関数は NULL 値を返すことができます。COUNT 関数は、空集合に適用されるとゼロ (0) を戻します。

GROUP BY および HAVING 文節の使用

集約関数で使用される値のセットは、照会の FROM および WHERE 文節に起因するコレクションによって決定されます。セットをグループに分割して、各グループに集約関数を適用することができます。このアクションを実行するには、照会で GROUP BY 文節を使用します。GROUP BY 文節によりグループ化メンバーが定義され、パス式のリストが構成されます。各パス式には、プリミティブ型の byte、short、int、long、float、double、boolean、char か、またはラッパー型の Byte、Short、Integer、Long、Float、Double、BigDecimal、String、Boolean、Character、java.util.Date、java.sql.Date、java.sql.Time、java.sql.Timestamp、java.util.Calendar、Java SE 5 enum の各フィールドを指定します。

以下の例では、照会で GROUP BY 文節を使用して各部門の平均給与を計算する場合を示します。

```
SELECT e.dept.deptno, AVG ( e.salary) FROM EmpBean e GROUP BY e.dept.deptno
```

セットをグループに分割する場合は、NULL 値は別の NULL 値と等しいとみなされます。

グループ化は HAVING 文節を使用してフィルター操作でき、集約関数またはグループ化メンバーを組み込む前にグループ・プロパティーをテストします。このフィルター操作は、WHERE 文節が FROM 文節からタプル (すなわち、戻りコレクション値のレコード) をフィルター操作する方法に類似しています。HAVING 文節の例を以下に示します。

```
SELECT e.dept.deptno, AVG ( e.salary) FROM EmpBean e
GROUP BY e.dept.deptno
HAVING COUNT(e) > 3 AND e.dept.deptno > 5
```

この照会は、従業員が 3 人より多く、部門番号が 5 より大きい部門の平均給与を返します。

GROUP BY 文節がなくても、HAVING 文節を使用することができます。この場合は、完全なセットは単一グループとして扱われ、HAVING 文節が適用されます。

ObjectGrid 照会の Backus-Naur Form

ObjectGrid 照会の BNF (Backus-Naur Form) 記法のまとめを以下に示します。

表 10. BNF 要約への鍵

表記	説明
{...}	グループ化
[...]	オプションの構文
太字	キーワード
*	ゼロ以上
|	代替

```
ObjectGrid QL ::=select_clause from_clause [where_clause] [group_by_clause]
  [having_clause] [order_by_clause]

from_clause ::=FROM identification_variable_declaration
  [,identification_variable_declaration]*

identification_variable_declaration ::=collection_member_declaration |
  range_variable_declaration

collection_member_declaration ::=IN ( collection_valued_path_expression |
  single_valued_navigation) [AS] identifier | [LEFT [OUTER]
  | INNER] JOIN collection_valued_path_expression |
  single_valued_navigation [AS] identifier

range_variable_declaration ::=abstract_schema_name [AS] identifier

single_valued_path_expression ::={single_valued_navigation | identification_variable}.
  { state_field | state_field.value_object_attribute } | single_valued_navigation

single_valued_navigation ::=identification_variable.[ single_valued_association_field. ]*
  single_valued_association_field

collection_valued_path_expression ::=identification_variable.[
  single_valued_association_field. ]* collection_valued_association_field

select_clause ::= SELECT [DISTINCT] [ selection , ]* selection

selection ::= {single_valued_path_expression |identification_variable | OBJECT
  ( identification_variable) |aggregate_functions } [[ AS ] id ]

order_by_clause ::= ORDER BY [ {identification_variable.[ single_valued_association_field.
  ]*state_field} [ASC|DESC],]* {identification_variable.[
  single_valued_association_field. ]*state_field}[ASC|DESC]

where_clause ::= WHERE conditional_expression

conditional_expression ::= conditional_term | conditional_expression OR conditional_term

conditional_term ::= conditional_factor | conditional_term AND conditional_factor

conditional_factor ::= [NOT] conditional_primary

conditional_primary ::= simple_cond_expression | (conditional_expression)

simple_cond_expression ::= comparison_expression | between_expression | like_expression |
  in_expression | null_comparison_expression | empty_collection_comparison_expression |
  exists_expression | collection_member_expression
```

```
between_expression ::= numeric_expression [NOT] BETWEEN numeric_expression
  AND numeric_expression | string_expression [NOT] BETWEEN
  string_expression AND string_expression | datetime_expression [NOT]
  BETWEEN datetime_expression AND datetime_expression

in_expression ::= identification_variable.[ single_valued_association_field. ]state_field
  [*NOT] IN { (subselect) | ( atom ,]* atom) }

atom ::= { string_literal | numeric_literal | input_parameter }

like_expression ::=string_expression [NOT] LIKE {string_literal | input_parameter}
  [ESCAPE {string_literal | input_parameter}]

null_comparison_expression ::= {single_valued_path_expression | input_parameter} IS
  [ NOT ] NULL

empty_collection_comparison_expression ::= collection_valued_path_expression IS
  [NOT] EMPTY

collection_member_expression ::={ ssingle_valued_path_expression | input_parameter }[
  NOT ] MEMBER [ OF ]collection_valued_path_expression

exists_expression ::= EXISTS {(subselect)}

subselect ::= SELECT [{ ALL | DISTINCT }] subselection from_clause
  [where_clause] [group_by_clause] [having_clause]

subselection ::= {single_valued_path_expression |identification_variable |
  aggregate_functions }

group_by_clause ::= GROUP BY[single_valued_path_expression,]*
  single_valued_path_expression

having_clause ::= HAVING conditional_expression

comparison_expression ::= numeric_ expression comparison_operator { numeric_expression
  | {SOME | ANY | ALL} (subselect) } | string_expression
  comparison_operator {

string_expression | {SOME | ANY | ALL}(subselect) } |

datetime_expression comparison_operator {

datetime_expression {SOME | ANY | ALL}(subselect) } |

boolean_expression {=|<>} {

boolean_expression {SOME | ANY | ALL}(subselect) } |

entity_expression {=|<>} {

entity_expression {SOME| ANY | ALL}(subselect) }

comparison_operator ::= = | > | >= | < | <= | <>

string_expression ::= string_primary | (subselect)

string_primary ::=state_field_path_expression |string_literal | input_parameter |
  functions_returning_strings

datetime_expression ::= datetime_primary |(subselect)

datetime_primary ::=state_field_path_expression | string_literal | long_literal
  | input_parameter | functions_returning_datetime

boolean_expression ::= boolean_primary |(subselect)

boolean_primary ::=state_field_path_expression | boolean_literal | input_parameter

entity_expression ::=single_valued_association_path_expression |
  identification_variable | input_parameter

numeric_expression ::= simple_numeric_expression |(subselect)

simple_numeric_expression ::= numeric_term | numeric_expression {+|-} numeric_term

numeric_term ::= numeric_factor | numeric_term {*|/} numeric_factor

numeric_factor ::= {+|-} numeric_primary

numeric_primary ::= single_valued_path_expression | numeric_literal |
  ( numeric_expression ) | input_parameter | functions

aggregate_functions :=

AVG([ALL|DISTINCT] identification_variable.
  [ single_valued_association_field. ]*state_field) |
```

```
COUNT([ALL|DISTINCT] {single_valued_path_expression |
  identification_variable}) |

MAX([ALL|DISTINCT] identification_variable.[
  single_valued_association_field. ]*state_field) |

MIN([ALL|DISTINCT] identification_variable.[
  single_valued_association_field. ]*state_field) |

SUM([ALL|DISTINCT] identification_variable.[
  single_valued_association_field. ]*state_field)

functions ::=

ABS (simple_numeric_expression) |

CONCAT (string_primary , string_primary) |

LOWER (string_primary) |

LENGTH(string_primary) |

LOCATE(string_primary, string_primary [, simple_numeric_expression]) |

MOD (simple_numeric_expression, simple_numeric_expression) |

SIZE (collection_valued_path_expression) |

SQRT (simple_numeric_expression) |

SUBSTRING (string_primary, simple_numeric_expression[, simple_numeric_expression]) |

UPPER (string_primary) |

TRIM ([[LEADING | TRAILING | BOTH] [trim_character]
  FROM] string_primary)
```

照会パフォーマンス調整

照会のパフォーマンスを調整する場合は、以下の手法とヒントを使用してください。

パラメーターの使用

照会を実行する場合、照会ストリングを構文解析し、照会を実行する計画を開発する必要がありますが、両方ともコストがかかる可能性があります。WebSphere eXtreme Scale は、照会ストリングによって照会計画をキャッシュに入れます。キャッシュは有限サイズであるため、照会ストリングを可能な限り再利用することが重要です。名前付きパラメーターまたは定位置パラメーターを使用しても、照会計画の再利用が促進され、パフォーマンスが向上します。

```
Positional Parameter Example  Query q = em.createQuery("select c from
Customer c where c.surname=?1");   q.setParameter(1, "Claus");
```

索引の使用

マップに対する適切な索引付けは、マップ・パフォーマンス全体にいくらかのオーバーヘッドをもたらしますが、照会パフォーマンスに著しい効果をもたらす場合があります。照会に関係するオブジェクト属性に索引付けを行わない場合、照会エンジンは、属性ごとにテーブル・スキャンを実行します。テーブル・スキャンは、照会実行時に最もコストのかかる操作です。照会に関係するオブジェクト属性に対する索引付けにより、照会エンジンは、不必要なテーブル・スキャンを回避でき、照会パフォーマンス全体を改善することができます。アプリケーションが最も読み取られるマップに対して照会を集中的に使用するように設計されている場合は、照会に関係するオブジェクト属性に対して索引を構成してください。マップがほとんど

更新される場合は、照会パフォーマンスの改善と、マップに対する索引付けオーバーヘッドとのバランスを取る必要があります。詳しくは、 125 ページの『索引付け』を参照してください。

Plain Old Java Object (POJO) がマップ内に保管されている場合、適切に索引付けすることによって、Java リフレクションを回避できます。次の例では、予算フィールドに索引が作成済みである場合、照会は WHERE 文節を範囲見出し検索と置換します。それ以外の場合、照会では、マップ全体をスキャンし、Java リフレクションを使用して最初に予算を取得してから、予算を値 50000 と比較することによって、WHERE 文節を評価します。

```
SELECT d FROM DeptBean d WHERE d.budget=50000
```

個別照会を最適に調整する方法、および各種の構文、オブジェクト・モデル、および索引が照会のパフォーマンスにどのように影響するかについて詳しくは、『照会計画』を参照してください。

ページ編集の使用

クライアント/サーバー環境では、照会エンジンは、結果マップ全体をクライアントにトランスポートします。戻されるデータは、妥当なチャンクに分割される必要があります。EntityManager Query および ObjectMap ObjectQuery の両インターフェースは、結果のサブセットを戻すことを照会に許可する setFirstResult および setMaxResults メソッドをサポートします。

エンティティーの代わりにプリミティブ値を戻す

EntityManager Query API を使用すると、エンティティーは照会パラメーターとして戻されます。照会エンジンは、現在のところ、これらのエンティティーに対するキーをクライアントに戻します。クライアントが getResultIterator メソッドからの Iterator を使用して、これらのエンティティーを繰り返すとき、各エンティティーは、EntityManager インターフェース上の find メソッドで作成されたかのように、自動的に拡張され、管理されます。エンティティー・グラフ全体は、クライアント上のエンティティー ObjectMap からビルドされます。エンティティー値属性およびその他の関連エンティティーは、可能な限り解決されます。

コストのかかるグラフのビルドを回避するには、パス・ナビゲーションを使用して個々の属性を戻すように照会を変更してください。

例:

```
// Returns an entity
SELECT p FROM Person p
// Returns attributes SELECT p.name, p.address.street, p.address.city, p.gender FROM Person p
```

照会計画

すべての eXtreme Scale 照会には照会計画があります。この計画は、照会エンジンが ObjectMap および索引とどのように対話するかを説明するものです。照会計画を表示すると、照会ストリングまたは索引が適切に使用されているかどうかを判断できます。また照会計画を使用すると、照会ストリング中のわずかな変更が eXtreme Scale による照会の実行方法に及ぼす変化を検討することもできます。

照会計画は、以下のいずれかの手段で表示できます。

- EntityManager Query または ObjectQuery の getPlan API メソッド
- ObjectGrid 診断トレース

getPlan メソッド

ObjectQuery および Query インターフェースの getPlan メソッドは、照会計画を説明するストリングを戻します。このストリングは、標準出力で表示することも、照会計画を表示するためのログで表示することもできます。注: 分散環境では、getPlan メソッドは、サーバーに対して実行されず、定義された索引を示しません。計画を表示するには、エージェントを使用して、サーバー上でその計画を表示します。

照会計画トレース

照会計画は、ObjectGrid トレースを使用して表示できます。照会計画トレースを有効とするには、以下のトレース仕様を使用します。

QueryEnginePlan=debug=enabled

トレース・ログ・ファイルを有効にする方法およびその検出方法について詳しくは、301 ページの『ログおよびトレース』を参照してください。

照会計画の例

この照会計画では、for という単語を使用して、この照会が ObjectMap コレクションで繰り返されるか、または派生するコレクション (q2.getEmps()、q2.dept、または内部ループによって返される一時的コレクションなど) で繰り返されることを示します。コレクションが ObjectMap のコレクションである場合、照会計画は、順次スキャン (INDEX SCAN で指示) や固有または非固有の索引が使用されているかどうかを示します。また、照会計画ではフィルター・ストリングを使用して、コレクションに適用される条件式をリストします。

通常、デカルト積は対象照会では使用されません。以下の照会では、外部ループ内の EmpBean マップ全体をスキャンし、内部ループ内の DeptBean マップ全体をスキャンします。

```
SELECT e, d FROM EmpBean e, DeptBean d

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in DeptBean ObjectMap using INDEX SCAN
   returning new Tuple( q2,  q3  )
```

以下の照会では、EmpBean マップを順次スキャンして特定部門の全従業員名を検索し、従業員オブジェクトを取得します。この照会では、従業員オブジェクトからその部門オブジェクトにナビゲートして、d.no=1 フィルターを適用します。この例の場合、各従業員はただ 1 つの部門オブジェクト参照を持つため、内部ループが 1 回実行されます。

```
SELECT e.name FROM EmpBean e JOIN e.dept d WHERE d.no=1

Plan trace:
```

```
for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in  q2.dept
      filter ( q3.getNo() = 1 )
   returning new Tuple( q2.name  )
```

以下の照会は、前記の照会と同等です。ただし、以下の照会では、まず DeptBean 1 次キー・フィールド番号に対して定義された固有索引を使用することで、結果が 1 つの部門オブジェクトに絞り込まれるため、実行効率が高まります。照会により、この部門オブジェクトから従業員オブジェクトにナビゲートされ、以下のように従業員名が取得されます。

```
SELECT e.name FROM DeptBean d JOIN d.emps e WHERE d.no=1

Plan trace:

for q2 in DeptBean ObjectMap using UNIQUE INDEX key=(1)
     for q3 in  q2.getEmps()
   returning new Tuple( q3.name  )
```

以下の照会を使用して、開発または販売に従事するすべての従業員を検索します。この照会では、EmpBean マップ全体をスキャンするとともに、式 d.name = 'Sales' or d.name='Dev' を評価することで追加のフィルタリングを実行します。

```
SELECT e FROM EmpBean e, in (e.dept) d WHERE d.name = 'Sales'
 or d.name='Dev'

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in  q2.dept
      filter (( q3.getName() = Sales ) OR ( q3.getName() = Dev ) )
   returning new Tuple( q2  )
```

以下の照会は前記の照会と同等ですが、この照会では異なる照会計画を実行し、フィールド名について作成された範囲索引を使用します。一般的に、部門オブジェクトの範囲の絞り込みに名前フィールドの索引が使用されることにより、開発または販売部門がごく少数である場合は照会が高速実行されるため、この照会の方が性能が高くなります。

```
SELECT e FROM DeptBean d, in(d.emps) e WHERE d.name='Dev' or d.name='Sales'

Plan trace:

IteratorUnionIndex of

     for q2 in DeptBean ObjectMap using INDEX on name = (Dev)
       for q3 in  q2.getEmps()


     for q2 in DeptBean ObjectMap using INDEX on name = (Sales)
       for q3 in  q2.getEmps()
```

以下の照会を使用して、従業員のいない部門を検索します。

```
SELECT d FROM DeptBean d WHERE NOT EXISTS(select e from d.emps e)

Plan trace:

for q2 in DeptBean ObjectMap using INDEX SCAN
    filter ( NOT  EXISTS (    correlated collection defined as
```

```
       for q3 in  q2.getEmps()
       returning new Tuple( q3      )

   returning new Tuple( q2  )
```

以下の照会は前述の照会と同等ですが、この照会では SIZE スカラー関数が使用されます。この照会でパフォーマンスは同じですが、作成が容易になっています。

```
SELECT d FROM DeptBean d WHERE SIZE(d.emps)=0
for q2 in DeptBean ObjectMap using INDEX SCAN
    filter (SIZE( q2.getEmps()) = 0 )
   returning new Tuple( q2  )
```

以下の例は、同様の性能を持つ前述の照会と同じ照会を書き込む別の方法を示していますが、この照会も容易に書き込むことができます。

```
SELECT d FROM DeptBean d WHERE d.emps is EMPTY

Plan trace:

for q2 in DeptBean ObjectMap using INDEX SCAN
    filter ( q2.getEmps() IS EMPTY  )
   returning new Tuple( q2  )
```

以下の照会では、パラメーターの値と等しい名前を持つ従業員の住所のうち少なくとも 1 つと一致する住所を持つすべての従業員を検索します。内部ループは外部ループに依存関係を持ちません。この照会では、内部ループは 1 回のみ実行されます。

```
SELECT e FROM EmpBean e WHERE e.home =  any (SELECT e1.home FROM EmpBean e1
 WHERE e1.name=?1)
for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( q2.home =ANY     temp collection defined as

       for q3 in EmpBean ObjectMap using INDEX on name = ( ?1)
       returning new Tuple( q3.home      )
 )
   returning new Tuple( q2  )
```

以下の照会は前述の照会と同等ですが、この照会には相関副照会があり、さらに内部ループが繰り返し実行されます。

```
SELECT e FROM EmpBean e WHERE EXISTS(SELECT e1 FROM EmpBean e1 WHERE
 e.home=e1.home and e1.name=?1)

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( EXISTS (    correlated collection defined as

       for q3 in EmpBean ObjectMap using INDEX on name = (?1)
        filter ( q2.home =  q3.home )
       returning new Tuple( q3      )

   returning new Tuple( q2  )
```

索引を使用した照会の最適化

索引を適切に定義および使用すると、照会のパフォーマンスをかなり改善できます。

WebSphere eXtreme Scale 照会では、組み込み HashIndex プラグインを使用すると、照会のパフォーマンスを改善できます。索引は、エンティティーまたはオブジ

ェクト属性に対して定義できます。照会エンジンは、その WHERE 文節で以下のいずれかのストリングが使用されると、定義された索引を自動的に使用します。

- 以下の演算子を使用する比較式: =、<、>、<=、または >= (等しくない <> を除くすべての比較式)
- BETWEEN 式
- 式のオペランドが定数またはシンプル・ターム

要件

照会で使用される場合、索引には以下の要件があります。

- すべての索引は組み込み HashIndex プラグインを使用する必要があります。
- すべての索引は静的に定義されていなければなりません。動的索引はサポートされません。
- 自動的に静的 HashIndex プラグインを作成するために @Index アノテーションを使用できます。
- すべての単一属性索引の RangeIndex プロパティーは true に設定されていなければなりません。
- すべての複合索引の RangeIndex プロパティーは false に設定されていなければなりません。
- すべてのアソシエーション (リレーションシップ) 索引の RangeIndex プロパティーは false に設定されていなければなりません。

キャッシュされたオブジェクトを検索するためのより効果的な方法については、131 ページの『複合 HashIndex』を参照してください。

索引選択に関するヒントの使用

索引は、HINT_USEINDEX 定数付きの setHint メソッドを Query および ObjectQuery インターフェースで使用すると、手動で選択することができます。これは、最も効率的な索引を使用するよう照会を最適化する際に役立ちます。

属性索引を使用する照会例

以下の例では、シンプル・ターム e.empid、e.name、e.salary、d.name、d.budget、および e.isManager が使用されています。これらの例では、索引がエンティティーまたは値オブジェクトの名前、給与、および予算フィールドに対して定義済みであることを前提としています。empid フィールドは 1 次キーであり、isManager には索引が定義されていません。

以下の照会では、名前と給与の両フィールドに対して索引を使用します。この場合、名前が最初のパラメーターの値に一致するか、給与が 2 番目のパラメーターの値に一致するすべての従業員が戻されます。

```
SELECT e FROM EmpBean e where e.name=?1 or e.salary=?2
```

以下の照会では、名前と予算の両フィールドに対して索引を使用します。この照会は、2000 より大きい予算を持つ 'DEV' という名前の付いたすべての部門を戻します。

```
SELECT d FROM DeptBean dwhere d.name='DEV' and d.budget>2000
```

以下の照会では、給与が 3000 より高く、かつパラメーターの値と等しい isManager フラグ値を持つ従業員をすべて戻します。この照会では、給与フィールドに対して定義された索引を使用するとともに、比較式 e.isManager=?1. を評価することで追加のフィルタリングを実行します。

```
SELECT e FROM EmpBean e where e.salary>3000 and e.isManager=?1
```

次の照会では、1 番目のパラメーターより大きい給与を得ているか、または管理者である従業員をすべて検索します。給与フィールドには索引が定義済みですが、照会では、EmpBean フィールドの 1 次キーに対して作成された組み込み索引をスキャンし、式 e.salary>?1 または e.isManager=TRUE を評価します。

```
SELECT e FROM EmpBean e WHERE e.salary>?1 or e.isManager=TRUE
```

以下の照会では、文字 a が含まれている名前の従業員を戻します。名前フィールドには索引が定義済みですが、名前フィールドが LIKE 式で使用されているため、照会ではこの索引を使用しません。

```
SELECT e FROM EmpBean e WHERE e.name LIKE '%a%'
```

以下の照会では、名前が「Smith」ではない従業員をすべて検索します。名前フィールドには索引が定義済みですが、照会では等しくない (<>) 比較演算子を使用するため、この索引を使用しません。

```
SELECT e FROM EmpBean e where e.name<>'Smith'
```

以下の照会では、予算がパラメーターの値より小さく、かつ従業員給与が 3000 より大きい部門をすべて検索します。この照会では、給与の索引を使用しますが、dept.budget がシンプル・タームではないため、予算の索引を使用しません。dept オブジェクトは、コレクション e から導き出されます。dept オブジェクトを検索するのに、予算の索引を使用する必要はありません。

```
SELECT dept from EmpBean e, in (e.dept) dept where e.salary>3000 and
dept.budget<?
```

以下の照会では、1、2、および 3 の empid を持つ従業員の給与より大きい給与の従業員をすべて検索します。比較には副照会が含まれているため、索引 salary は使用されません。empid は、1 次キーですが、すべての 1 次キーには組み込み索引が定義済みであるため、固有索引の検索に使用されます。

```
SELECT e FROM EmpBean e WHERE e.salary > ALL (SELECT e1.salary    FROM
EmpBean e1 WHERE e1.empid=1 or e1.empid =2 or e1.empid=99)
```

索引が照会で使用されているかどうかを確認する場合は、114 ページの『照会計画』を表示できます。以下に、前述の照会の照会計画例を示します。

```
for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( q2.salary >ALL     temp collection defined as
       IteratorUnionIndex of

         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(1)
        )
```

```
         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(2)
        )

         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(99)
        )
       returning new Tuple( q3.salary )
   returning new Tuple( q2  )

for q2 in EmpBean ObjectMap using RANGE INDEX on salary with range(3000,)
     for q3 in  q2.dept
      filter ( q3.budget <  ?1 )
   returning new Tuple( q3  )
```

属性の索引付け

前に定義された制約付きで、任意の単一属性タイプに対して索引を定義できます。

@Index を使用したエンティティー索引の定義

エンティティーに索引を定義するには、単にアノテーションを定義します。

Entities using annotations

```
  @Entity
  public class Employee {
  @Id int empid;
  @Index String name
  @Index double salary
  @ManyToOne Department dept;
}
@Entity
  public class Department {
  @Id int deptid;
  @Index String name;
  @Index double budget;
  boolean isManager;
  @OneToMany Collection<Employee> employees;
  }
```

XML の使用

XML を使用して索引を定義することもできます。

Entities without annotations

```
  public class Employee {
  int empid;
  String name
  double salary
  Department dept;
  }

  public class Department {
  int deptid;
  String name;
  double budget;
  boolean isManager;
  Collection employees;
  }
```

ObjectGrid XML with attribute indexes

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
```

```
  <objectGrid name="DepartmentGrid" entityMetadataXMLFile="entity.xml>
  <backingMap name="Employee" pluginCollectionRef="Emp"/>
  <backingMap name="Department" pluginCollectionRef="Dept"/>
  </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
  <backingMapPluginCollection id="Emp">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.salary"/>
  <property name="AttributeName" type="java.lang.String" value="salary"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  <backingMapPluginCollection id="Dept">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.budget"/>
  <property name="AttributeName" type="java.lang.String" value="budget"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  </backingMapPluginCollections>
  </objectGridConfig>
```

Entity XML

```
<?xml version="1.0" encoding="UTF-8"?>
  <entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

  <description>Department entities</description>
  <entity class-name="acme.Employee" name="Employee" access="FIELD">
  <attributes>
  <id name="empid" />
  <basic name="name" />
  <basic name="salary" />
  <many-to-one name="department"
  target-entity="acme.Department"
  fetch="EAGER">
  <cascade><cascade-persist/></cascade>
  </many-to-one>
  </attributes>
  </entity>
  <entity class-name="acme.Department" name="Department" access="FIELD">
  <attributes>
  <id name="deptid" />
  <basic name="name" />
  <basic name="budget" />
  <basic name="isManager" />
  <one-to-many name="employees"
  target-entity="acme.Employee"
  fetch="LAZY" mapped-by="parentNode">
  <cascade><cascade-persist/></cascade>
  </one-to-many>
  </attributes>
  </entity>
  </entity-mappings>
```

XML を使用した非エンティティーの索引の定義

非エンティティー・タイプに対する索引は XML 内で定義されます。MapIndexPlugin を作成するときに、エンティティー・マップに対しての場合と非エンティティー・マップに対しての場合で相違はありません。

Java bean

```
public class Employee {
  int empid;
  String name
  double salary
  Department dept;
```

```
public class Department {
 int deptid;
 String name;
 double budget;
 boolean isManager;
 Collection employees;
 }
```

ObjectGrid XML with attribute indexes

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
  <objectGrid name="DepartmentGrid">
  <backingMap name="Employee" pluginCollectionRef="Emp"/>
  <backingMap name="Department" pluginCollectionRef="Dept"/>
  <querySchema>
  <mapSchemas>
  <mapSchema mapName="Employee" valueClass="acme.Employee"
  primaryKeyField="empid" />
  <mapSchema mapName="Department" valueClass="acme.Department"
  primaryKeyField="deptid" />
  </mapSchemas>
  <relationships>
  <relationship source="acme.Employee"
  target="acme.Department"
  relationField="dept" invRelationField="employees" />
  </relationships>
  </querySchema>
  </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
  <backingMapPluginCollection id="Emp">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.salary"/>
  <property name="AttributeName" type="java.lang.String" value="salary"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  <backingMapPluginCollection id="Dept">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.budget"/>
  <property name="AttributeName" type="java.lang.String" value="budget"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  </backingMapPluginCollections>
  </objectGridConfig>
```

索引付けのリレーションシップ

WebSphere eXtreme Scale は、関連エンティティーの外部キーを親オブジェクト内に保管します。エンティティーの場合、キーは基本となるタプルに保管されます。非エンティティー・オブジェクトの場合、キーは親オブジェクトに明示的に保管されます。

リレーションシップ属性に索引を追加すると、循環参照を使用するか、IS NULL、IS EMPTY、SIZE、および MEMBER OF 照会フィルターを使用する照会をスピードアップすることができます。単一値関連と多値関連がともに、ObjectGrid 記述子 XML ファイル内に @Index アノテーションまたは HashIndex プラグイン構成を持つ場合があります。

@Index を使用したエンティティー・リレーションシップ索引の定義

以下の例では、@Index アノテーションのあるエンティティーを定義します。

Entity with annotation

```
@Entity
public class Node {
    @ManyToOne @Index
    Node parentNode;

    @OneToMany @Index
    List<Node> childrenNodes = new ArrayList();

    @OneToMany @Index
    List<BusinessUnitType> businessUnitTypes = new ArrayList();
}
```

XML を使用したエンティティー・リレーションシップ索引の定義

以下の例は、XML と HashIndex プラグインを使用して、同じエンティティーおよび索引を定義しています。

Entity without annotations

```
public class Node {
int nodeId;
Node parentNode;
List<Node> childrenNodes = new ArrayList();
List<BusinessUnitType> businessUnitTypes = new ArrayList();
}
```

ObjectGrid XML

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
  <objectGrid name="ObjectGrid_Entity" entityMetadataXMLFile="entity.xml>
  <backingMap name="Node" pluginCollectionRef="Node"/>
  <backingMap name="BusinessUnitType" pluginCollectionRef="BusinessUnitType"/>
  </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
  <backingMapPluginCollection id="Node">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="parentNode"/>
  <property name="AttributeName" type="java.lang.String" value="parentNode"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="businessUnitType"/>
  <property name="AttributeName" type="java.lang.String" value="businessUnitTypes"/>

<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
 </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="childrenNodes"/>
  <property name="AttributeName" type="java.lang.String" value="childrenNodes"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
  </bean>
  </backingMapPluginCollection>
  </backingMapPluginCollections>
  </objectGridConfig>
```

Entity XML

```
  <?xml version="1.0" encoding="UTF-8"?>
  <entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

  <description>My entities</description>
  <entity class-name="acme.Node" name="Account" access="FIELD">
  <attributes>
  <id name="nodeId" />
```

```
  <one-to-many name="childrenNodes"
  target-entity="acme.Node"
  fetch="EAGER" mapped-by="parentNode">
  <cascade><cascade-all/></cascade>
  </one-to-many>
  <many-to-one name="parentNodes"
  target-entity="acme.Node"
  fetch="LAZY" mapped-by="childrenNodes">
  <cascade><cascade-none/></cascade>
  </one-to-many>
  <many-to-one name="businessUnitTypes"
  target-entity="acme.BusinessUnitType"
  fetch="EAGER">
  <cascade><cascade-persist/></cascade>
  </many-to-one>
</attributes>
  </entity>
  <entity class-name="acme.BusinessUnitType" name="BusinessUnitType" access="FIELD">
  <attributes>
  <id name="buId" />
  <basic name="TypeDescription" />
  </attributes>
  </entity>
  </entity-mappings>
```

前に定義された索引を使用すると、以下の例のエンティティー照会が最適化されます。

```
SELECT n FROM Node n WHERE n.parentNode is null
SELECT n FROM Node n WHERE n.businessUnitTypes is EMPTY
  SELECT n FROM Node n WHERE size(n.businessUnitTypes)>=10
  SELECT n FROM BusinessUnitType b, Node n WHERE b member of n.businessUnitTypes and b.name='TELECOM'
```

非エンティティー・リレーションシップ索引の定義

以下の例では、ObjectGrid 記述子 XML ファイル内の非エンティティー・マップの HashIndex プラグインを定義します。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="ObjectGrid_POJO">
      <backingMap name="Node" pluginCollectionRef="Node"/>
      <backingMap name="BusinessUnitType" pluginCollectionRef="BusinessUnitType"/>
      <querySchema>
        <mapSchemas>
          <mapSchema mapName="Node"
     valueClass="com.ibm.websphere.objectgrid.samples.entity.Node"
            primaryKeyField="id" />
          <mapSchema mapName="BusinessUnitType"
            valueClass="com.ibm.websphere.objectgrid.samples.entity.BusinessUnitType"
            primaryKeyField="id" />
        </mapSchemas>
        <relationships>
          <relationship source="com.ibm.websphere.objectgrid.samples.entity.Node"
            target="com.ibm.websphere.objectgrid.samples.entity.Node"
            relationField="parentNodeId" invRelationField="childrenNodeIds" />
          <relationship source="com.ibm.websphere.objectgrid.samples.entity.Node"
            target="com.ibm.websphere.objectgrid.samples.entity.BusinessUnitType"
            relationField="businessUnitTypeKeys" invRelationField="" />
        </relationships>
      </querySchema>
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="Node">
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="parentNode"/>
<property name="Name" type="java.lang.String" value="parentNodeId"/>
<property name="AttributeName" type="java.lang.String" value="parentNodeId"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
      </bean>
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="businessUnitType"/>
        <property name="AttributeName" type="java.lang.String" value="businessUnitTypeKeys"/>

<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
   </bean>
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="childrenNodeIds"/>
  <property name="AttributeName" type="java.lang.String" value="childrenNodeIds"/>
  <property name="RangeIndex" type="boolean" value="false"
```

```
      description="Ranges are not supported for association indexes." />
</bean>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

上記の索引構成が指定されると、以下の例のオブジェクト照会が最適化されます。

```
SELECT n FROM Node n WHERE n.parentNodeId is null
SELECT n FROM Node n WHERE n.businessUnitTypeKeys is EMPTY
SELECT n FROM Node n WHERE size(n.businessUnitTypeKeys)>=10
SELECT n FROM BusinessUnitType b, Node n WHERE
 b member of n.businessUnitTypeKeys and b.name='TELECOM'
```

索引付け

索引付けフィーチャーは、MapIndexPlugin プラグインで表され、BackingMap 上にいくつかの索引を作成して、非キー・データ・アクセスをサポートするために使用します。

索引のタイプおよび構成

索引付けフィーチャーは、MapIndexPlugin と表されるか、または略して Index で表されます。Index は BackingMap プラグインです。BackingMap では、各索引プラグインが索引構成規則に従っている限り、複数の索引プラグインを構成できます。

索引付けフィーチャーは、1 つ以上の索引を BackingMap に作成する場合に使用できます。1 つの索引は、BackingMap 内の 1 つのオブジェクトの 1 つの属性または属性のリストから作成されます。このフィーチャーにより、アプリケーションはより迅速に特定のオブジェクトを見つけることができます。索引付けフィーチャーを使用すると、アプリケーションは特定の値を持つオブジェクトや、ある範囲の索引属性値内にあるオブジェクトを見つけることができます。

可能な索引付けには、静的および動的という 2 つのタイプがあります。静的索引付けの場合、ObjectGrid インスタンスを初期化する前に、BackingMap に索引プラグインを構成する必要があります。この構成を行うには、BackingMap を XML で構成するか、またはプログラマチックに構成します。静的索引付けでは、まず最初に、ObjectGrid の初期化中に索引を作成します。索引は常に BackingMap に同期しており、いつでも使用できる準備ができています。静的索引付けプロセスが既に開始している場合、索引は、eXtreme Scale トランザクション管理プロセスの一環として保守されます。トランザクションが変更をコミットすると、それらの変更は静的索引も更新し、トランザクションがロールバックされれば索引の変更もロールバックされます。

動的索引付けの場合は、索引を含む ObjectGrid インスタンスの初期化の前または後に、BackingMap に索引を作成することができます。動的索引付けプロセスのライフサイクルはアプリケーションによって制御されるので、不要になったら動的索引を削除することができます。アプリケーションが動的索引を作成する場合は、索引作成プロセスを完了するまでに時間がかかるために、その索引をすぐに使用できないことがあります。この時間は索引付けされるデータの量に依存するので、特定の索引付けイベントが発生したときにそのことを通知してもらいたいアプリケーションのために、DynamicIndexCallback インターフェースが提供されています。これらの

イベントには、準備完了、エラー、および破棄があります。アプリケーションは、このコールバック・インターフェースを実装し、動的索引付けプロセスに登録できます。

BackingMap に索引プラグインが構成されている場合、対応する ObjectMap からアプリケーション索引プロキシー・オブジェクトを取得することができます。ObjectMap の getIndex メソッドを呼び出し、索引プラグインの名前を渡すと、索引プロキシー・オブジェクトが戻されます。索引プロキシー・オブジェクトを適切なアプリケーション索引インターフェース (MapIndex、MapRangeIndex、またはカスタマイズされた索引インターフェースなど) にキャストする必要があります。索引プロキシー・オブジェクトを取得したら、アプリケーション索引インターフェースで定義されたメソッドを使用して、キャッシュされたオブジェクトを検出することができます。

次のリストに、索引付けの使用手順をまとめます。

- 静的または動的索引プラグインを BackingMap に追加します。
- ObjectMap の getIndex メソッドを発行して、アプリケーション索引プロキシー・オブジェクトを取得します。
- MapIndex、MapRangeIndex またはカスタマイズされた索引インターフェースなどの適切なアプリケーション索引インターフェースに、索引プロキシー・オブジェクトをキャストします。
- アプリケーション索引インターフェースで定義されたメソッドを使用して、キャッシュされたオブジェクトを検出します。

ユーザー独自の索引プラグインの作成について詳しくは、「*プログラミング・ガイド*」内の索引プラグイン作成に関する説明を参照してください。

索引付けの使用方法について詳しくは、「*プログラミング・ガイド*」内の非キー・データ・アクセスでの索引付けの使用に関する説明および 131 ページの『複合 HashIndex』を参照してください。

データ品質に関する考慮事項

索引照会メソッドの結果が表わすのは、特定の時刻におけるデータのスナップショットのみです。結果がアプリケーションに戻された後には、データ項目に対するロックは取得されません。アプリケーションは、戻されたデータ・セットに対してデータ更新が発生する可能性があることに注意する必要があります。例えば、アプリケーションは MapIndex の findAll メソッドを実行して、キャッシュされたオブジェクトのキーを取得します。戻されたこのキー・オブジェクトは、キャッシュ内のデータ項目に関連付けられています。アプリケーションは、キー・オブジェクトを提供することにより、ObjectMap に対して get メソッドを実行して、オブジェクトを検出できるようになっている必要があります。get メソッドが呼び出される直前に、別のトランザクションがキャッシュからそのデータ・オブジェクトを削除した場合、戻される結果はヌルです。

索引付けのパフォーマンスに関する考慮事項

索引付けフィーチャーの主な目的の 1 つは、BackingMap の全体的なパフォーマンスを改善することです。索引付けの使い方が不適切な場合は、アプリケーションのパフォーマンスが低下する可能性があります。このフィーチャーを使用する前に、次の要因について検討します。

- **並行書き込みトランザクションの数:** 索引処理は、トランザクションが BackingMap にデータを書き込むたびに起こりえます。アプリケーションが索引照会操作を試行しているときに、多くのトランザクションがデータをマップに書き込んでいると、パフォーマンスが低下します。
- **照会操作で戻される結果セットのサイズ:** 結果セットのサイズが大きくなるにつれて、照会のパフォーマンスは低下します。結果セットのサイズが BackingMap の 15% 以上になるとパフォーマンスは低下する傾向にあります。
- **同じ BackingMap に作成される索引の数:** 各索引がシステム・リソースを消費します。BackingMap に作成される索引の数が増えると、パフォーマンスは低下します。

索引付け機能は、BackingMap パフォーマンスを大幅に改善できることがあります。理想的なケースは、BackingMap の大部分の操作が読み取りであり、照会の結果セットが BackingMap エントリーのわずかな割合に過ぎず、ごく少数の索引が BackingMap に対して作成される場合です。

非キー・データ・アクセスの索引付けの使用

データのキー・アクセスの代わりに索引付けを使用したほうがずっと効率的です。

必要なステップ

1. BackingMap に、静的または動的索引プラグインを追加します。
2. ObjectMap の getIndex メソッドを発行して、アプリケーション索引プロキシー・オブジェクトを取得します。
3. MapIndex、MapRangeIndex またはカスタマイズされた索引インターフェースなどの適切なアプリケーション索引インターフェースに、索引プロキシー・オブジェクトをキャストします。
4. アプリケーション索引インターフェースに定義されたメソッドを使用して、キャッシュされたオブジェクトを検出します。

HashIndex クラスは、組み込みアプリケーション索引インターフェースである MapIndex と MapRangeIndex の両方をサポートすることのできる索引プラグイン実装です。ユーザー独自の索引を作成することもできます。

注: 分散 ObjectGrid 環境では、索引オブジェクトは、クライアント ObjectGrid から取得された場合は、タイプがクライアント索引オブジェクトになり、すべての索引操作はリモート・サーバー ObjectGrid において実行されます。マップが区画化されている場合、索引操作は各区画でリモートに実行され、各区画からの結果はマージされてから、アプリケーションに戻されます。パフォーマンスは、区画数と、各区画が戻す結果のサイズによって決まります。これらの要因が両方とも大きいと、パフォーマンスが低下することがあります。

ユーザー独自の索引プラグインを作成したい場合、 148 ページの『索引プラグインの書き込み』を参照してください。

索引付けについては、 125 ページの『索引付け』および 131 ページの『複合 HashIndex』を参照してください。

静的索引プラグインの追加

静的索引プラグインを BackingMap 構成に追加するには、XML 構成およびプログラマチック構成という 2 つのアプローチを使用することができます。以下の例は、XML 構成アプローチを示します。

静的索引プラグインの追加: XML 構成アプローチ

```
<backingMapPluginCollection id="person">
     <bean id="MapIndexplugin"
   className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
          <property name="Name" type="java.lang.String" value="CODE"
      description="index name" />
          <property name="RangeIndex" type="boolean" value="true"
      description="true for MapRangeIndex" />
          <property name="AttributeName" type="java.lang.String" value="employeeCode"
      description="attribute name" />
     </bean>
</backingMapPluginCollection>
```

この XML 構成例では、組み込み HashIndex クラスが索引プラグインとして使用されています。HashIndex は、ユーザーが構成できるプロパティーをサポートしています。上の例にある Name、RangeIndex、 AttributeName などです。

- Name プロパティーは、この索引プラグインを識別するストリングである「CODE」と構成されています。Name プロパティー値は、BackingMap の有効範囲内で固有でなければならず、BackingMap の ObjectMap インスタンスから名前で索引オブジェクトを取り出すのに使用できます。
- RangeIndex プロパティーは「true」と構成されています。これが意味するのは、取り出された索引オブジェクトをアプリケーションが MapRangeIndex インターフェースにキャストできるということです。RangeIndex プロパティーが「false」と構成されている場合は、アプリケーションは取り出された索引オブジェクトを MapIndex インターフェースにしかキャストできません。MapRangeIndex は、範囲関数 greater than や less than、あるいは両方を使用するデータ検出をサポートしますが、MapIndex は equals 関数のみをサポートします。索引が照会によって使用される場合、RangeIndex プロパティーは、単一属性索引に対して「true」と構成されていなければなりません。リレーションシップ索引および複合索引に対しては、RangeIndex プロパティーは「false」と構成される必要があります。
- AttributeName プロパティーは「employeeCode」と構成されています。これは、キャッシュに入れられたオブジェクトの employeeCode 属性を使用して、単一属性索引が構築されることを意味しています。複数の属性を持つ、キャッシュに入れられたオブジェクトをアプリケーションが検索する必要がある場合、AttributeName プロパティーには、属性をコンマで区切ったリストを設定でき、そうすると複合索引が生成されます。

詳しくは、*管理ガイド*の HashIndex の構成に関する説明を参照してください。

BackingMap インターフェースには、静的索引プラグインを追加するために使用できるメソッドとして、addMapIndexplugin と setMapIndexplugins の 2 つがあります。詳しくは、API の資料を参照してください。

以下に、プログラマチック構成アプローチのコード例を示します。

静的索引プラグインの追加：プログラマチック構成アプローチ

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;

    ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
    ObjectGrid ivObjectGrid = ogManager.createObjectGrid( "grid" );
    BackingMap personBackingMap = ivObjectGrid.getMap("person");

    // use the builtin HashIndex class as the index plugin class.
    HashIndex mapIndexplugin = new HashIndex();
    mapIndexplugin.setName("CODE");
    mapIndexplugin.setAttributeName("EmployeeCode");
    mapIndexplugin.setRangeIndex(true);
    personBackingMap.addMapIndexplugin(mapIndexplugin);
```

静的索引の使用

静的索引プラグインが BackingMap 構成に追加され、含んでいる ObjectGrid インスタンスが初期化された後であれば、アプリケーションは BackingMap の ObjectMap インスタンスから名前によって索引オブジェクトを取得できます。 索引オブジェクトは、アプリケーション索引インターフェースにキャストします。これで、アプリケーション索引インターフェースがサポートしている操作を実行できるようになります。

以下のコード例は、静的索引をどのように取得し、使用するのかを示しています。

静的索引の使用例

```
Session session = ivObjectGrid.getSession();
    ObjectMap map = session.getMap("person ");
    MapRangeIndex codeIndex = (MapRangeIndex) m.getIndex("CODE");
    Iterator iter = codeIndex.findLessEqual(new Integer(15));
    while (iter.hasNext()) {
        Object key = iter.next();
        Object value = map.get(key);
    }
```

動的索引の追加、除去、および使用

BackingMap インスタンスから動的索引を、いつでもプログラマチックに作成および除去することができます。動的索引と静的索引の違いは、動的索引は、索引を含む ObjectGrid インスタンスが初期化されたあとでも作成できる、という点です。動的索引付けは、静的索引付けとは違って非同期プロセスであり、使用される前に作動可能状態になっている必要があります。このメソッドは、動的索引の取得および使用に、静的索引と同じアプローチを使用します。動的索引は、不要になると除去できます。BackingMap インターフェースには、動的索引を作成および除去するためのメソッドがあります。

createDynamicIndex メソッドおよび removeDynamicIndex メソッドについて詳しくは、BackingMap API を参照してください。

動的索引の使用例

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;

       ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
       ObjectGrid og = ogManager.createObjectGrid("grid");
       BackingMap bm = og.getMap("person");
```

```
og.initialize();

// create index after ObjectGrid initialization without DynamicIndexCallback.
bm.createDynamicIndex("CODE", true, "employeeCode", null);

try {
    // If not using DynamicIndexCallback, need to wait for the Index to be ready.
    // The waiting time depends on the current size of the map
    Thread.sleep(3000);
} catch (Throwable t) {
    // ...
}

// When the index is ready, applications can try to get application index
// interface instance.
// Applications have to find a way to ensure that the index is ready to use,
// if not using DynamicIndexCallback interface.
// The following example demonstrates the way to wait for the index to be ready
// Consider the size of the map in the total waiting time.

Session session = og.getSession();
ObjectMap m = session.getMap("person");
MapRangeIndex codeIndex = null;

int counter = 0;
int maxCounter = 10;
boolean ready = false;
while (!ready && counter < maxCounter) {
    try {
        counter++;
        codeIndex = (MapRangeIndex) m.getIndex("CODE");
        ready = true;
    } catch (IndexNotReadyException e) {
        // implies index is not ready, ...
        System.out.println("Index is not ready. continue to wait.");
        try {
            Thread.sleep(3000);
        } catch (Throwable tt) {
            // ...
        }
    } catch (Throwable t) {
        // unexpected exception
        t.printStackTrace();
    }
}

if (!ready) {
    System.out.println("Index is not ready.  Need to handle this situation.");
}

// Use the index to peform queries
// Refer to the MapIndex or MapRangeIndex interface for supported operations.
// The object attribute on which the index is created is the EmployeeCode.
// Assume that the EmployeeCode attribute is Integer type: the
// parameter that is passed into index operations has this data type.

Iterator iter = codeIndex.findLessEqual(new Integer(15));

// remove the dynamic index when no longer needed

bm.removeDynamicIndex("CODE");
```

DynamicIndexCallback インターフェース

DynamicIndexCallback インターフェースは、作動可能、エラー、または破棄という索引付けイベントの発生時に、そのことを通知してもらう必要のあるアプリケーションのために設計されています。 DynamicIndexCallback は、BackingMap の createDynamicIndex メソッドのオプション・パラメーターです。アプリケーションは、索引付けイベントの通知を受け取ると、登録済みの DynamicIndexCallback インスタンスを使用して、ビジネス・ロジックを実行することができます。例えば、作動可能イベントは、索引を使用する準備が整ったことを意味します。アプリケーションは、このイベントの通知を受け取ると、アプリケーション索引インターフェースのインスタンスの取得および使用を試行することができます。詳しくは、API 資料で DynamicIndexCallback API を参照してください。

以下のコード例は、DynamicIndexCallback インターフェースの使い方を示したものです。

DynamicIndexCallback インターフェースの使用

```
BackingMap personBackingMap = ivObjectGrid.getMap("person");
    DynamicIndexCallback callback = new DynamicIndexCallbackImpl();
    personBackingMap.createDynamicIndex("CODE", true, "employeeCode", callback);


    class DynamicIndexCallbackImpl implements DynamicIndexCallback {
        public DynamicIndexCallbackImpl() {
```

```
    }

    public void ready(String indexName) {
        System.out.println("DynamicIndexCallbackImpl.ready() -> indexName = " + indexName);

        // Simulate what an application would do when notified that the index is ready.
        // Normally, the application would wait until the ready state is reached and then proceed
        // with any index usage logic.
        if("CODE".equals(indexName)) {
            ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
            ObjectGrid og = ogManager.createObjectGrid( "grid" );
            Session session = og.getSession();
            ObjectMap map = session.getMap("person");
            MapIndex codeIndex = (MapIndex) map.getIndex("CODE");
            Iterator iter = codeIndex.findAll(codeValue);
        }
    }

    public void error(String indexName, Throwable t) {
        System.out.println("DynamicIndexCallbackImpl.error() -> indexName = " + indexName);
        t.printStackTrace();
    }

    public void destroy(String indexName) {
        System.out.println("DynamicIndexCallbackImpl.destroy() -> indexName = " + indexName);
    }
}
```

複合 HashIndex

複合 HashIndex により、照会のパフォーマンスが向上し、高いコストがかかるマップのスキャンを避けることができます。また、この機能は、検索条件に多くの属性が関係する際、キャッシュ・オブジェクトを検索するための便利な方法を HashIndex API に提供します。

パフォーマンスの改善

複合 HashIndex を使用すると、一致検索条件に入れた複数の属性によって、キャッシュされたオブジェクトを高速かつ簡単に見つけることができます。複合索引は、完全属性一致検索をサポートしますが、範囲検索はサポートしません。

注: 複合索引は ObjectGrid 照会言語での BETWEEN 演算子の使用をサポートしません。BETWEEN は範囲サポートを必要とすることがあるためです。より大 (>)、より小 (<) 条件も、範囲索引を必要とするため機能しません。

適切な複合索引が WHERE 条件で使用可能な場合、複合索引によって照会のパフォーマンスを改善できます。適切な複合索引とは、全属性一致の WHERE 条件に含まれているのとまったく同じ属性をその複合索引が持っているという意味です。

照会では、次の例のように 1 つの条件に多数の属性が関係することがあります。

```
SELECT a FROM Address a WHERE a.city='Rochester' AND a.state='MN' AND
a.zipcode='55901'
```

複合索引では、マップのスキャンを回避したり、複数の単一属性索引の結果を結合したりすることで、照会のパフォーマンスを改善できます。例の場合、属性 (city,state,zipcode) を持つ複合索引が定義されている場合は、照会エンジンは、複合索引を使用して、`city='Rochester'`、`state='MN'`、および `zipcode='55901'` のエントリーを検索できます。複合索引も、city、state、および zipcode 属性に対する属性索引もなければ、照会エンジンは、マップをスキャンするか、複数の単一属性検索を結合する必要があり、それには通常コストの高いオーバーヘッドが生じます。また、複合索引の照会がサポートするのは、完全一致パターンのみです。

複合索引の構成

複合索引を構成する方法は 3 とおりあります。XML を使用するか、プログラマチックに行うか、または (エンティティー・マップの場合のみ) エンティティー・アノテーションを付ける方法です。

XML の使用

XML で複合索引を構成するには、下のようなコードを構成ファイルの backingMapPluginCollections エレメント内に組み込みます。

```
Composite index - XML configuration approach
<bean id="MapIndexPlugin"  className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Address.CityStateZip"/>
<property name="AttributeName" type="java.lang.String" value="city,state,zipcode"/>
</bean>
```

プログラマチック構成

プログラマチックに行う場合、以下のサンプル・コードによって、上記 XML と同じ複合索引が作成されます。

```
HashIndex mapIndex = new HashIndex();
 mapIndex.setName("Address.CityStateZip");
 mapIndex.setAttributeName(("city,state,zipcode"));
 mapIndex.setRangeIndex(true);

 BackingMap bm = objectGrid.defineMap("mymap");
 bm.addMapIndexPlugin(mapIndex);
```

複合索引の構成は、XML を使用した通常の索引の構成と同じですが、attributeName プロパティー値は例外なので注意してください。複合索引の場合、attributeName の値は、コンマ区切りの属性のリストです。例えば、値クラス Address は、city、state、および zipcode の 3 つの属性を持つとします。この場合、"city,state,zipcode" という attributeName プロパティー値を使用して複合索引を定義し、複合索引に city、state、および zipcode が含まれていることを示すことができます。

また、複合 HashIndexes は、範囲検索をサポートしないため、RangeIndex プロパティーを true に設定しないよう注意してください。

エンティティー・アノテーション

エンティティー・マップの場合、アノテーションによる方法を使用して複合索引を定義できます。エンティティー・クラスのレベルで、CompositeIndexes アノテーション内に CompositeIndex のリストを定義できます。CompositeIndex には name と attributeNames プロパティーがあります。各 CompositeIndex は、エンティティーの関連の BackingMap に適用される HashIndex インスタンスに関連付けられます。HashIndex は、非範囲索引として構成されます。

```
@Entity
@CompositeIndexes({
    @CompositeIndex(name="CityStateZip", attributeNames="city,state,zipcode"),
    @CompositeIndex(name="lastnameBirthday", attributeNames="lastname,birthday")
 })
public class Address {
    @Id int id;
    String street;
    String city;
    String state;
    String zipcode;
    String lastname;
    Date birthday;
}
```

各複合索引の name プロパティーは、エンティティーおよび BackingMap 内で固有でなければなりません。名前が指定されない場合は、生成された名前が使用されます。attributeNames プロパティーを使用して、HashIndex attributeName のデータ (コンマ区切りの属性のリスト) が設定されます。属性名は、エンティティーがフィールド・アクセスを使用するように構成されているときは、パーシスタント・フィールド名と一致します。そのように構成されていない場合は、プロパティー・アクセス・エンティティーに対する JavaBeans 命名規則の定義に従いプロパティー名と一致します。例えば、属性名が「street」だった場合、プロパティー getter メソッドの名前は getStreet です。

複合索引の検索の実行

複合索引が構成されたら、アプリケーションは、MapIndex インターフェースの findAll(Object) メソッドを使用して、意下のように検索を実行できます。

```
Object[] compositeValue = new Object[]{
            MapIndex.EMPTY_VALUE , "MN", "55901"};
     Iterator iter = mapIndex.findAll(compositeValue);
```

MapIndex.EMPTY_VALUE は compositeValue[0] に割り当てられ、評価から city 属性が除外されることを示します。結果には、state 属性が"MN" に等しく、zipcode 属性が"55901" に等しいオブジェクトのみが含まれます。

次の照会では、上記の複合索引の構成が有効です。

```
SELECT a FROM Address a WHERE a.city='Rochester' AND a.state='MN' AND
a.zipcode='55901'

SELECT a FROM Address a WHERE a.state='MN' AND a.zipcode='55901'
```

照会エンジンは適切な複合索引を見つけ、それを使用して全属性一致のケースで照会のパフォーマンスを高めます。

シナリオによっては、全属性一致のすべての照会に対応するために、一部の属性がオーバーラップする複数の複合索引をアプリケーションで定義する必要がある場合があります。索引の数が増えることの欠点は、マップ操作でパフォーマンス・オーバーヘッドが生じる可能性があることです。

マイグレーションおよびインターオペラビリティー

複合索引の使用に関する唯一の制約は、異種のコンテナーがある分散環境では、アプリケーションが複合索引を構成できないことです。古いコンテナーは、複合索引構成を認識しないため、古いコンテナーと新しいコンテナーは混用できません。複合索引は、既存の通常の属性索引とよく似ていますが、複合索引では、複数の属性にまたがる索引付けが許可される点が異なります。通常の属性索引のみを使用する場合、コンテナー混在環境はそのまま存続できます。

Data Grid API

DataGrid API は、ObjectGrid のすべてまたはサブセットに対して、データが置かれている場所と並行してビジネス・ロジックを実行するための、単純なプログラミング・インターフェースを提供します。

DataGrid API と区画化

DataGrid API を使用して、クライアントは、グリッド内の 1 つの区画、区画のサブセット、またはすべての区画に、要求を送信できます。クライアントはキーのリストを指定でき、WebSphere eXtreme Scale はそれらのキーをホスティングしている区画のセットを判定します。次に要求はセット内のすべての区画に並行して送信され、クライアントはその結果を待ちます。クライアントは、キーを指定せずに要求を送信することもでき、したがって、要求はすべての区画に送信されます。

グリッドにデプロイされているエージェントは、クライアント・モードでは動作しません。これらのエージェントは、プライマリーの断片を直接操作します。プライマリーの断片を直接操作すると、最高のパフォーマンスが得られ、秒当たり何万あるいはそれ以上のトランザクションを処理できます。これは、エージェントがメモリーの最高速度でデータを操作するからです。プライマリー断片を直接操作するということは、エージェントはその断片内のデータしか見ることができないということでもあります。これにより、クライアントでは実行できない興味あるいくつかの機会が与えられます。

標準的な eXtreme Scale クライアントは、要求をルーティングする必要があるため、トランザクションから区画を決定できなければなりません。エージェントが断片に直接接続されている場合は、ルーティングは不要です。すべての要求はその断片に送られます。エージェントは断片に直接接続されているためにルーティングが発生しないので、共通の区画化キーなどを考慮しなくても、その断片内の他のマップに含まれるデータにアクセスすることができます。

DataGrid エージェントとエンティティー・ベースのマップ

マップは、キー・オブジェクトと値オブジェクトを保持します。キー・オブジェクトは生成済みタプルであり、値オブジェクトもそうです。通常エージェントにはアプリケーションによって指定されたキー・オブジェクトが与えられます。

キー・オブジェクトは生成済みタプルであり、値オブジェクトもそうです。通常エージェントにはアプリケーションによって指定されたキー・オブジェクトが与えられます。このキー・オブジェクトが、アプリケーション、またはアプリケーションがエンティティー・マップの場合はタプルによって使用されるキー・オブジェクトになります。エンティティーを使用するアプリケーションがタプルを直接操作することはあまりなく、エンティティーにマップされた Java オブジェクトを使用して作業するほうが一般的です。

したがって、Agent クラスは EntityAgentMixin インターフェースを実装できます。これにより、Agent クラスは強制的にもう 1 つのメソッドである getClassForEntity() を実装します。このメソッドは、サーバー・サイドのエージェントとともに使用されるエンティティー・クラスを返します。キーは、process メソッドおよび reduce メソッドを呼び出す前に、このエンティティーに変換されます。

これは、これらのメソッドにキーのみが与えられている非 EntityAgentMixin エージェントとは異なるセマンティックです。EntityAgentMixin を実装しているエージェントは、1 つのオブジェクトにキーと値を組み込んでいるエンティティー・オブジェクトを受け取ります。

注: エンティティーがサーバー上に存在しない場合、キーは、管理対象エンティティーではなく、キーの未加工のタプル・フォーマットになります。

DataGrid API の例

DataGrid API では、グリッド・プログラミングの 2 つの一般的なパターンがサポートされます。最初のパターンは、並列マップで、2 番目のパターンは、並列削減です。

並列マップ

並列マップでは、一連のキーのエントリーを処理することができ、処理されたそれぞれのエントリーに対する結果が返されます。アプリケーションでは、キーのリストが作成され、Map オペレーションの呼び出し後、キー/結果ペアの Map を受け取ります。結果は、各キーのエントリーに対して関数が適用されたものです。関数はアプリケーションによって提供されます。

MapGridAgent 呼び出しのフロー

キーのコレクションを使用して AgentManager.callMapAgent メソッドが呼び出されると、MapGridAgent インスタンスがシリアライズされ、各キーで解決されたそれぞれのプライマリー区画に送信されます。すなわち、エージェントに保管されているインスタンス・データは、すべてサーバーに送信できます。したがって、各プライマリー区画は、エージェントのインスタンスを 1 つ保持します。各キーのインスタンスごとに 1 回 process メソッドが呼び出され、その結果、区画が解決されます。各 process メソッドの結果はその後、シリアライズされてクライアントへ返され、マップ・インスタンス内で呼び出し元に返されます。そこでは、結果はマップの中の値として提示されます。

キーのコレクションが指定されずに AgentManager.callMapAgent メソッドが呼び出されると、MapGridAgent インスタンスがシリアライズされ、すべてのプライマリー区画に送信されます。すなわち、エージェントに保管されているインスタンス・データは、すべてサーバーに送信できます。したがって、各プライマリー区画は、エージェントのインスタンス (区画) を 1 つ保持します。processAllEntries メソッドは、区画ごとに呼び出されます。各 processAllEntries メソッドの結果はその後、シリアライズされてクライアントへ返され、マップ・インスタンス内で呼び出し元に返されます。以下の例は、次のような形状の Person エンティティーが存在することを前提とします。

```
import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;
@Entity
public class Person
{
  @Id String ssn;
  String firstName;
  String surname;
  int age;
}
```

アプリケーション提供の関数は、MapAgentGrid インターフェースを実装するクラスとして作成されています。Person の年齢を 2 倍にした値を返す関数のエージェントの例を以下に示します。

```
public class DoublePersonAgeAgent implements MapGridAgent, EntityAgentMixin
{
  private static final long serialVersionUID = -2006093916067992974L;

  int lowAge;
  int highAge;

  public Object process(Session s, ObjectMap map, Object key)
  {
    Person p = (Person)key;
    return new Integer(p.age * 2);
  }

  public Map processAllEntries(Session s, ObjectMap map)
  {
    EntityManager em = s.getEntityManager();
    Query q = em.createQuery("select p from Person p where p.age > ?1 and p.age < ?2");
    q.setParameter(1, lowAge);
    q.setParameter(2, highAge);
    Iterator iter = q.getResultIterator();
    Map<Person, Interger> rc = new HashMap<Person, Integer>();
   while(iter.hasNext())
    {
   Person p = (Person)iter.next();
   rc.put(p, (Integer)process(s, map, p));
  }
  return rc;
 }
 public Class getClassForEntity()
 {
  return Person.class;
 }
}
```

この例は、Person を 2 倍にする Map エージェントを示しています。最初に、process メソッドについて説明します。最初の process メソッドでは、処理する Person が提供されます。単純に、エントリーの年齢を 2 倍にした値が返されます。2 番目の process メソッドは、各区画で呼び出され、年齢が lowAge と highAge 間にあるすべての Person オブジェクトを検出し、その年齢を 2 倍にした値を返します。

```
Session s = grid.getSession();
ObjectMap map = s.getMap("Person");
AgentManager amgr = map.getAgentManager();

DoublePersonAgeAgent agent = new DoublePersonAgeAgent();

// make a list of keys
ArrayList<Person> keyList = new ArrayList<Person>();
Person p = new Person();
p.ssn = "1";
keyList.add(p);
p = new Person ();
p.ssn = "2";
keyList.add(p);

// get the results for those entries
Map<Tuple, Object> = amgr.callMapAgent(agent, keyList);
```

この例は、Person Map への Session および参照を取得するクライアントを示しています。エージェント・オペレーションは、特定の Map に対して実行されます。AgentManager インターフェースはその Map から取得されます。呼び出されるエージェントのインスタンスが作成され、属性を設定することにより、必要な状態がオブジェクトに追加されます。ただし、この例では追加はありません。次に、キーのリストが構成されます。person 1 については 2 倍にした値と、person 2 については同じ値を保持する Map が戻されます。

エージェントがキー・セットに対して呼び出されます。指定したキーを使用して、グリッド内の各区画で、並行してエージェントの process メソッドが呼び出されま

す。Map は、指定のキーに対する結果をマージして戻されます。この例では、person 1 の年齢を 2 倍にした値および person 2 の同様の値を保持する Map が返されます。

キーが存在しない場合でも、エージェントは呼び出されます。この場合、エージェントでマップ・エントリーを作成する機会が与えられます。 EntityAgentMixin を使用する場合、処理するキーはエンティティーではなく、エンティティーに対する実際の Tuple キー値になります。キーが不明の場合、特定の形状の Person オブジェクトを検出するためにすべての区画に問い合わせて、年齢の 2 倍の戻り値を得ることができます。以下に例を示します。

```
Session s = grid.getSession();
  ObjectMap map = s.getMap("Person");
  AgentManager amgr = map.getAgentManager();

  DoublePersonAgeAgent agent = new DoublePersonAgeAgent();
  agent.lowAge = 20;
  agent.highAge = 9999;

  Map m = amgr.callMapAgent(agent);
```

上の例では、 AgentManager が Person Map のために取得され、エージェントは、該当の Person の最小年齢と最大年齢で構成され、初期化されています。次に、callMapAgent メソッドを使用してエージェントが呼び出されます。キーが提供されていないことに注意してください。したがって、ObjectGrid により、グリッド内のすべての区画で並行してエージェントが呼び出され、マージされた結果がクライアントに返されます。最低と最高の間にある年齢のすべての Person オブジェクトがグリッド内で検出され、それらの Person オブジェクトの年齢の 2 倍が計算されます。つまり、特定の照会に適合するエンティティーを検出するためのグリッド API の使用方法を示しています。エージェントは、ObjectGrid により、単にシリアライズされて、必要なエントリーとともに区画へトランスポートされます。結果も同様に、クライアントへのトランスポートのためにシリアライズされます。Map API には注意が必要です。ObjectGrid でテラバイトのオブジェクトをホスティングする場合や、ObjectGrid が多数のサーバーで実行される場合、クライアントを実行する大容量のマシン以外では処理できない可能性があります。小規模のサブセットの処理にのみ使用する必要があります。大規模なサブセットを処理する必要がある場合、削減エージェントを使用して、1 つのクライアントではなく、グリッド内で処理することをお勧めします。

並列削減または集約エージェント

このスタイルのプログラミングでは、エントリーのサブセットが処理され、エントリーのグループに対して単一の結果が計算されます。このような結果の例は、次のとおりです。

- 最小値
- 最大値
- その他のビジネス固有関数

削減エージェントのコーディングおよび呼び出しは、Map エージェントと非常によく似ています。

ReduceGridAgent 呼び出しのフロー

キーのコレクションを使用して AgentManager.callReduceAgent メソッドが呼び出されると、ReduceGridAgent インスタンスがシリアライズされ、各キーで解決されたそれぞれのプライマリー区画に送信されます。すなわち、エージェントに保管されているインスタンス・データは、すべてサーバーに送信できます。したがって、各プライマリー区画は、エージェントのインスタンスを 1 つ保持します。reduce(Session s, ObjectMap map, Collection keys) メソッドは、インスタンス (区画) ごとに 1 回、区画に解決されるキーのサブセットを指定して呼び出されます。各 reduce メソッドの結果はその後、シリアライズされてクライアントへ返されます。reduceResults メソッドは、各リモートでの reduce 呼び出しから返されたそれぞれの結果のコレクションを使用して、クライアント ReduceGridAgent インスタンスに対して呼び出されます。reduceResults メソッドの結果は、callReduceAgent メソッドの呼び出し元に返されます。

キーのコレクションが指定されずに AgentManager.callReduceAgent メソッドが呼び出されると、ReduceGridAgent インスタンスがシリアライズされ、各プライマリー区画に送信されます。すなわち、エージェントに保管されているインスタンス・データは、すべてサーバーに送信できます。したがって、各プライマリー区画は、エージェントのインスタンスを 1 つ保持します。reduce(Session s, ObjectMap map) メソッドは、インスタンス (区画) ごとに 1 回呼び出されます。各 reduce メソッドの結果はその後、シリアライズされてクライアントへ返されます。reduceResults メソッドは、各リモートでの reduce 呼び出しから返されたそれぞれの結果のコレクションを使用して、クライアント ReduceGridAgent インスタンスに対して呼び出されます。reduceResults メソッドの結果は、callReduceAgent メソッドの呼び出し元に返されます。適合するエントリーの年齢を単純に加算する削減エージェントの例を以下に示します。

```
public class SumAgeReduceAgent implements ReduceGridAgent, EntityAgentMixin
{
  private static final long serialVersionUID = 2521080771723284899L;

  int lowAge;
  int highAge;

  public Object reduce(Session s, ObjectMap map, Collection keyList)
  {
    Iterator<Person> iter = keyList.iterator();
  int sum = 0;
  while (iter.hasNext())
  {
   Person p = iter.next();
   sume += p.age;
  }
  return new Integer(sum);
 }

 public Object reduce(Session s, ObjectMap map)
 {
  EntityManager em = s.getEntityManager ();
  Query q = em.createQuery("select p from Person p where p.age > ?1 and p.age < ?2");
  q.setParameter(1, lowAge);
  q.setParameter(2, highAge);
  Iterator<Person> iter = q.getResultIterator();
  int sum = 0;
  while(iter.hasNext())
  {
   sum += iter.next().age;
  }
  return new Integer(sum);
 }

 public Class getClassForEntity()
 {
  return Person.class;
 }
}
```

上の例はエージェントを示しています。このエージェントには、3 つの重要部分があります。1 番目の部分では、特定のエントリー・セットが照会なしで処理されま

す。単に、エントリーの年齢が繰り返し加算されます。メソッドから合計が返されます。 2 番目の部分では、照会が使用され、集約されるエントリーが選択されます。該当するすべての Person の年齢が合計されます。 3 番目のメソッドは、各区画からの結果を単一の結果に集約するために使用されます。ObjectGrid では、グリッド中のエントリー集約が並行して実行されます。各区画で中間結果が作成されるので、それを他の区画の中間結果と合わせて集約する必要があります。 3 番目のメソッドでこのタスクが実行されます。次の例の場合、エージェントが呼び出され、年齢が 10 歳から 20 歳までの Person のみの年齢が集約されます。

```
Session s = grid.getSession();
  ObjectMap map = s.getMap("Person");
  AgentManager amgr = map.getAgentManager();

  SumAgeReduceAgent agent = new SumAgeReduceAgent();

  Person p = new Person();
  p.ssn = "1";
  ArrayList<Person> list = new ArrayList<Person>();
 list.add(p);
 p = new Person ();
 p.ssn = "2";
 list.add(p);
 Integer v = (Integer)amgr.callReduceAgent(agent, list);
```

エージェントの機能

エージェントは、それが稼働しているローカル断片の内部で、自由に ObjectMap または EntityManager 操作を実行できます。エージェントは Session を受け取り、その Session が表す区画のデータの追加、更新、照会、読み取り、または削除を行うことができます。グリッドからデータを照会するだけのアプリケーションもあるでしょうが、特定の照会に一致するすべての Person の年齢を 1 だけ増やすようなエージェントを作成することもできます。エージェントが呼び出されるときには Session にトランザクションがあり、例外がスローされない限り、エージェントが戻るときにそのトランザクションはコミットされます。

エラー処理

マップ・エージェントが不明なキーで呼び出された場合、返される値は、EntryErrorValue インターフェースを実装するエラー・オブジェクトです。

トランザクション

マップ・エージェントはクライアントから分離したトランザクションで実行されます。エージェントの呼び出しは単一トランザクションにグループ化される場合があります。エージェントが失敗した (例外がスローされた) 場合、トランザクションはロールバックされます。トランザクション内で正常に実行したエージェントがある場合、失敗したエージェントと一緒にそれらのエージェントもロールバックされます。AgentManager は、正常に実行した、ロールバックされたエージェントを、新しいトランザクションで再実行します。

API 資料

WebSphere eXtreme Scale API には、パッケージまたはクラス名を検索して、システムまたはアプリケーション・プログラミング・インターフェースに関する詳細を見つけるために使用できる情報が含まれています。

API 資料については、WebSphere eXtreme Scale インフォメーション・センターを参照してください。

第 4 章 システム API およびプラグイン

プラグインとは、プラグ可能なコンポーネントに特定の機能を提供するコンポーネントです。ObjectGrid や BackingMap があります。eXtreme Scale をメモリー内データ・グリッドまたはデータベース処理スペースとして最も効果的に使用するために、使用可能なプラグインのパフォーマンスを最大限に活用できる最善の方法を慎重に決定してください。

プラグインの概要

WebSphere eXtreme Scale のプラグインは、プラグ可能なコンポーネント (ObjectGrid および BackingMap も含む) に、ある特定のタイプの機能を提供するコンポーネントです。WebSphere eXtreme Scale には、いくつかのプラグ・ポイントが用意されていて、アプリケーションおよびキャッシュのプロバイダーは、それらを使用して、さまざまなデータ・ストアや代替クライアント API と統合することができ、キャッシュの全体的なパフォーマンスを向上させることができます。この製品には事前にビルド済みのデフォルトのプラグインがいくつか付属していますが、ユーザーはアプリケーションを使用してカスタム・プラグインをビルドすることもできます。

すべてのプラグインは、1 つ以上の eXtreme Scale プラグイン・インターフェースを実装する具象クラスです。これらのクラスは、適切なタイミングで ObjectGrid によってインスタンス化され、呼び出されます。ObjectGrid および BackingMap では、それぞれカスタム・プラグインの登録が可能です。

ObjectGrid プラグイン

ObjectGrid インスタンスでは以下のプラグインが使用可能です。

- **TransactionCallback**: TransactionCallback プラグインは、トランザクション・ライフサイクル・イベントを提供します。
- **ObjectGridEventListener**: ObjectGridEventListener プラグインは、ObjectGrid、断片、およびトランザクションに対して ObjectGrid ライフサイクル・イベントを提供します。
- **SubjectSource、MapAuthorization、SubjectValidation**: eXtreme Scale は、カスタム認証メカニズムを eXtreme Scale と統合することを可能にするいくつかのセキュリティー・エンドポイントを提供します。

共通 ObjectGrid プラグイン要件

ObjectGrid は、JavaBeans 規則を使用し、プラグイン・インスタンスをインスタンス化して初期化します。前述のすべてのプラグインの実装には以下の要件があります。

- プラグイン・クラスは最上位レベルのパブリック・クラスでなければなりません。
- プラグイン・クラスは、引数を取らない public コンストラクターを提供する必要があります。

- プラグイン・クラスは、サーバーおよびクライアント (必要に応じて) の両方のクラスパスで使用可能でなければなりません。
- 属性は、JavaBeans スタイル・プロパティー・メソッドを使用して設定する必要があります。
- プラグインは、特に記述のない限り ObjectGrid の初期化より前に登録され、ObjectGrid が初期化された後は変更できません。

BackingMap プラグイン

BackingMap では、以下のプラグインが使用可能です。

- **Evictor**: デフォルトのキャッシュ・エントリー除去メカニズムと、カスタム・エビクターを作成するためのプラグインが提供されています。
- **Loader**: ObjectGrid マップ上のローダー・プラグインは、通常は同じシステムまたは他のシステム上のパーシスタント・ストアに保管されるデータ用のメモリー・キャッシュのような働きをします。
- **ObjectTransformer**: ObjectTransformer プラグインを使用すると、キャッシュ内のオブジェクトをシリアライズ、デシリアライズ、およびコピーすることができます。
- **OptimisticCallback**: OptimisticCallback プラグインは、オプティミスティック・ロック・ストラテジーを使用している場合に、キャッシュ・オブジェクトのバージョン管理および比較操作のカスタマイズを可能にします。
- **MapEventListener**: MapEventListener プラグインは、BackingMap について発生するコールバック通知および重要なキャッシュ状態変更を提供します。
- **Indexing**: MapIndexplug-in プラグインで表される索引付け機能を使用して、1 つ以上の索引を BackingMap マップにビルドし、非キー・データ・アクセスをサポートできます。

プラグイン・ライフサイクル

ほとんどのプラグインには、本来機能するように設計されたメソッドに加えて、initialize メソッドと destroy メソッド (または同等のメソッド) があります。各プラグインのこうした特殊化されたメソッドは、指定された機能ポイントで呼び出すことができます。initialize メソッドと destroy メソッドの両方でプラグインのライフサイクルが定義されます。これらのメソッドは、その「所有者」オブジェクトによって制御されます。所有者オブジェクトは、実際に指定のプラグインを使用するオブジェクトです。所有者はグリッド・クライアント、サーバー、またはバックアップ・マップである場合があります。

所有者オブジェクトは、初期化中、その所有するプラグインの initialize メソッドを呼び出します。所有者オブジェクトの破棄サイクル中は、最終的にプラグインの destroy メソッドも呼び出されます。各プラグインで使用できる他のメソッドと同様に、initialize メソッドと destroy メソッドの特性ついて詳しくは、各プラグインの関連トピックを参照してください。

例えば、分散環境を考えてみます。クライアント・サイド ObjectGrid およびサーバー・サイド ObjectGrid は両方とも、独自のプラグインを持っています。クライアン

ト・サイド ObjectGrid のライフサイクルおよび当然そのプラグイン・インスタンスは、すべてのサーバー・サイド ObjectGrid とプラグイン・インスタンスから独立しています。

こうした分散トポロジーで、objectGrid.xml ファイル内に定義された「myGrid」という名前の ObjectGrid があり、「myObjectGridEventListener」という名前のカスタマイズされた ObjectGridEventListener によって構成されているとします。objectGridDeployment.xml ファイルは、myGrid ObjectGrid のデプロイメント・ポリシーを定義します。コンテナー・サーバーを始動するために、objectGrid.xml と objectGridDeployment.xml の両方が使用されます。コンテナー・サーバーの始動過程で、サーバー・サイド myGrid ObjectGrid インスタンスが初期化され、myObjectGrid インスタンスによって所有される myObjectGridEventListener インスタンスの initialize メソッドが呼び出されます。コンテナー・サーバーの始動後、アプリケーションはサーバー・サイド myGrid ObjectGrid インスタンスに接続して、クライアント・サイド・インスタンスを取得できます。

クライアント・サイド myGrid ObjectGrid インスタンスが取得されると、クライアント・サイド myGrid インスタンスが自身の初期化サイクルを経て、自身のクライアント・サイド myObjectGridEventListener インスタンスの initialize メソッドを呼び出します。このクライアント・サイド myObjectGridEventListener インスタンスは、サーバー・サイド myObjectGridEventListener インスタンスとは独立しています。そのライフサイクルは、その所有者、つまりクライアント・サイド myGrid ObjectGrid インスタンスによって制御されます。

アプリケーションがクライアント・サイド myGrid ObjectGrid インスタンスを切断または破棄する場合、所有されるクライアント・サイド myObjectGridEventListener インスタンスの destroy メソッドが自動的に呼び出されます。ただし、これはサーバー・サイド myObjectGridEventListener インスタンスには何の影響もありません。サーバー・サイド myObjectGridEventListener インスタンスの destroy メソッドが呼び出されるのは、コンテナー・サーバーを停止する際のサーバー・サイド myGrid ObjectGrid インスタンスの破棄サイクル時のみです。つまり、コンテナー・サーバーを停止する際には、含まれる ObjectGrid インスタンスが破棄され、その所有されるすべてのプラグインの destroy メソッドが呼び出されます。

前の例はクライアントとサーバーの ObjectGrid インスタンスの場合に特に適用されますが、プラグインの所有者は BackingMap でもあるので、こうしたライフサイクル考慮事項に基づいて作成するプラグイン構成を決定する際には注意が必要です。

イベント・リスナー

ObjectGridEventListener および MapEventListener プラグインを使用すると、eXtreme Scale キャッシュ内のさまざまなイベントの通知を構成できます。リスナー・プラグインは、他の eXtreme Scale プラグインと同様に、ObjectGrid または BackingMap インスタンスに登録されて、アプリケーションおよびキャッシュ・プロバイダーの統合およびカスタマイズの場所になります。

ObjectGridEventListener プラグイン

ObjectGridEventListener プラグインは、ObjectGrid インスタンス、断片、およびトランザクション用の eXtreme Scale ライフサイクル・イベントを提供します。

ObjectGridEventListener プラグインを使用して、ObjectGrid で重大なイベントが発生したときに通知を受け取ります。これらのイベントには、ObjectGrid の初期化、トランザクションの開始、トランザクションの終了、および ObjectGrid の破棄などがあります。これらのイベントを listen するには、ObjectGridEventListener インターフェースを実装するクラスを作成して、eXtreme Scale に追加します。

ObjectGridEventListener プラグインの作成について詳しくは、 145 ページの『ObjectGridEventListener プラグイン』を参照してください。また、 API 資料でも詳細を参照できます。

ObjectGridEventListener インスタンスの追加および除去

ObjectGrid は、複数の ObjectGridEventListener リスナーを持つことが可能です。リスナーの追加および除去は、ObjectGrid インターフェースで addEventListener、setEventListeners、および removeEventListener メソッドを使用して行います。また、 ObjectGridEventListener プラグインを ObjectGrid 記述子ファイルに明示的に登録することもできます。例については、 145 ページの『ObjectGridEventListener プラグイン』を参照してください。

MapEventListener プラグイン

MapEventListener プラグインは、BackingMap インスタンスに対して発生するコールバック通知および重要なキャッシュ状態変更を提供します。MapEventListener プラグインの作成について詳しくは、 『MapEventListener プラグイン』を参照してください。また、 API 資料でも詳細を参照できます。

MapEventListener インスタンスの追加および除去

eXtreme Scale は、複数の MapEventListener リスナーを持つことが可能です。リスナーの追加および除去は、BackingMap インターフェースで addMapEventListener、setMapEventListeners、および removeMapEventListener メソッドを使用して行います。また、 MapEventListener リスナーを ObjectGrid 記述子ファイルに明示的に登録することもできます。例については、『MapEventListener プラグイン』を参照してください。

MapEventListener プラグイン

MapEventListener プラグインは、マップがプリロードを終了したり、エントリーがマップから除去されたりしたときに、BackingMap オブジェクトに対して発生するコールバック通知および重要なキャッシュ状態変更を提供します。 MapEventListener プラグインは、MapEventListener インターフェースを実装するカスタム・クラスです。

MapEventListener プラグイン規則

MapEventListener プラグインを開発する際には、共通のプラグイン規則に従う必要があります。プラグイン規則について詳しくは、 141 ページの『プラグインの概要』を参照してください。その他のタイプのリスナー・プラグインについては、143 ページの『イベント・リスナー』を参照してください。

MapEventListener 実装を作成すると、プログラムで、あるいは、XML 構成を使用してそれを BackingMap 構成にプラグインできます。

MapEventListener 実装の作成

MapEventListener プラグインの実装は、アプリケーションに組み込むことができます。このプラグインで、MapEventListener インターフェースを実装し、マップに関する重要なイベントを受信する必要があります。エントリーがマップから除去されたとき、およびマップのプリロードが完了したときに、イベントが MapEventListener プラグインに送られます。

XML を使用した MapEventListener 実装のプラグイン

MapEventListner 実装は、XML を使用して構成できます。以下の XML は、myGrid.xml ファイルに存在しなければなりません。

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="myGrid">
            <backingMap name="myMap" pluginCollectionRef="myPlugins" />
        </objectGrid>
    </objectGrids>
    <backingMapPluginCollections>
        <backingMapPluginCollection id="myPlugins">
            <bean id="MapEventListener" className=
           "com.company.org.MyMapEventListener" />
        </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

このファイルを ObjectGridManager インスタンスに提供すると、この構成の作成が容易になります。以下のコード・スニペットは、この XML ファイルを使用して ObjectGrid インスタンスを作成する方法を示しています。新規に作成された ObjectGrid インスタンスにおいて、myMap BackingMap で MapEventListener が設定されます。

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid =
 objectGridManager.createObjectGrid("myGrid", new URL("file:etc/test/myGrid.xml"),
  true, false);
```

MapEventListener 実装のプログラムによるプラグイン

カスタム MapEventListener のクラス名は、com.company.org.MyMapEventListener クラスです。このクラスは MapEventListener インターフェースを実装します。以下のコード・スニペットは、カスタム MapEventListener オブジェクトを作成し、それを BackingMap オブジェクトに追加します。

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
BackingMap myMap = myGrid.defineMap("myMap");
MyMapEventListener myListener = new MyMapEventListener();
myMap.addMapEventListener(myListener);
```

ObjectGridEventListener プラグイン

ObjectGridEventListener プラグインは、ObjectGrid、断片、およびトランザクション用の WebSphere eXtreme Scale ライフサイクル・イベントを提供します。ObjectGridEventListener プラグインは、ObjectGrid が初期化または破棄されたとき、

およびトランザクションが開始または終了したときに通知を行います。
ObjectGridEventListener プラグインは、ObjectGridEventListener インターフェースを実装するカスタム・クラスです。必要な場合、この実装は、ObjectGridEventGroup サブインターフェースを含み、共通の eXtreme Scale プラグイン規則に従います。

概説

ObjectGridEventListener プラグインはローダー・プラグインが使用可能である場合に便利で、トランザクションの開始時と終了時に Java Database Connectivity (JDBC) 接続またはバックエンドへの接続を初期化する必要があります。通常、ObjectGridEventListener プラグインとローダー・プラグインは一緒に作成します。

ObjectGridEventListener プラグインの作成

ObjectGridEventListener プラグインは、重要な eXtreme Scale イベントに関する通知を受け取るために ObjectGridEventListener インターフェースを実装する必要があります。以下のインターフェースを実装して、追加のイベント通知を受け取ることができます。以下のサブインターフェースが ObjectGridEventGroup インターフェースに組み込まれています。

- ShardEvents インターフェース
- ShardLifecycle インターフェース
- TransactionEvents インターフェース

これらのインターフェースについて詳しくは、 API 資料を参照してください。

断片イベント

カタログ・サービスがプライマリー区画やレプリカの断片を Java 仮想マシン (JVM) に配置すると、その JVM 内に新しい ObjectGrid インスタンスが作成されて、その断片をホスティングします。JVM ホスト上でスレッドを開始する必要があるアプリケーションでは、プライマリーがこれらのイベントの通知を必要とします。ObjectGridEventGroup.ShardEvents インターフェースは、shardActivate メソッドおよび shardDeactivate メソッドを宣言します。これらのメソッドは、断片がプライマリーとして活動化される場合と、断片がプライマリーから非活動化される場合にのみ呼び出されます。アプリケーションでは、これら 2 つのイベントを使用することで、断片がプライマリーのときに追加スレッドを開始したり、断片がレプリカに戻ったときやサービスから除外されたときにスレッドを停止したりできます。

アプリケーションは、 shardActivate メソッドに提供されている ObjectGrid 参照で ObjectGrid#getMap メソッドを使用して特定の BackingMap を検索することで、どの区画が活動状態になっているかを特定できます。それからアプリケーションは、BackingMap#getPartitionId() メソッドを使用して区画番号を確認できます。各区画の番号は 0 から始まるため、最後の区画番号はデプロイメント記述子内の区画数 - 1 になります。

断片のライフサイクル・イベント

ObjectGridEventListener.initialize メソッドおよび ObjectGridEventListener.destroy メソッドのイベントは、ObjectGridEventGroup.ShardLifecycle インターフェースを使用して配信されます。

トランザクション・イベント

ObjectGridEventListener.transactionBegin メソッドおよび
ObjectGridEventListener.transactionEnd メソッドは、
ObjectGridEventGroup.TransactionEvents インターフェースを通じて導き出されます。

この方法の利点

ObjectGridEventListener プラグインが ObjectGridEventListener インターフェースおよび ShardLifecycle インターフェースを実装すると、リスナーに配信されるイベントは断片ライフサイクル・イベントだけになります。どの新規 ObjectGridEventGroup 内部インターフェースを実装しても、eXtreme Scale は、新規インターフェースを使用してそうした特定のイベントのみを配信するようになります。この実装では、コードの下位互換性が維持されます。新しい内部インターフェースを使用する場合は、必要な特定のイベントのみを受け取るようにすることができます。

ObjectGridEventListener プラグインの使用

カスタム ObjectGridEventListener プラグインを使用するには、
ObjectGridEventListener インターフェースおよびオプションの ObjectGridEventGroup サブインターフェースを実装するクラスをまず作成します。重大なイベントの通知を受け取れるように、カスタム・リスナーを eXtreme Scale に追加します。
ObjectGridEventListener プラグインを eXtreme Scale 構成に追加するには、プログラマチック構成と XML 構成の 2 つの方法があります。

ObjectGridEventListener プラグインのプログラマチックな構成

eXtreme Scale イベント・リスナーのクラス名が
com.company.org.MyObjectGridEventListener クラスであると想定します。このクラスは、ObjectGridEventListener インターフェースを実装します。以下のコード・スニペットは、カスタム ObjectGridEventListener を作成し、eXtreme Scale に追加します。

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
MyObjectGridEventListener myListener = new MyObjectGridEventListener();
myGrid.addEventListener(myListener);
```

XML を使用した ObjectGridEventListener プラグインの構成

ObjectGridEventListener プラグインは、XML を使用して構成することもできます。以下の XML は、前述のプログラムで作成した ObjectGrid イベント・リスナーと同等の構成を作成します。以下のテキストは、myGrid.xml ファイルに存在しなければなりません。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="myGrid">
            <bean id="ObjectGridEventListener"
       className="com.company.org.MyObjectGridEventListener" />
            <backingMap name="Book"/>
        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

Bean 宣言が backingMap 宣言の前にあることに注意してください。このファイルを ObjectGridManager プラグインに提供することで、この構成の作成が容易になります。以下のコード・スニペットは、この XML ファイルを使用して ObjectGrid インスタンスを作成する方法を示しています。作成した ObjectGrid インスタンスの myGrid ObjectGrid には、ObjectGridEventListener リスナーが設定されています。

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid",
 new URL("file:etc/test/myGrid.xml"), true, false);
```

索引プラグインの書き込み

MapIndexPlugin プラグイン (つまり索引) を使用すると、eXtreme Scale が提供する組み込み索引以上の、カスタムの索引付けストラテジーを作成できます。

索引付けに関する一般情報は、 125 ページの『索引付け』を参照してください。

索引付けの使用方法について詳しくは、 127 ページの『非キー・データ・アクセスの索引付けの使用』を参照してください。

MapIndexPlugin 実装は、MapIndexPlugin インターフェースを使用し、 eXtreme Scale プラグインの共通規則に従う必要があります。

以下のセクションに、この索引インターフェースの重要なメソッドをいくつか示します。

setProperties メソッド

setProperties メソッドを使用して、索引プラグインをプログラマチックに初期化することができます。このメソッドに渡される Properties オブジェクト・パラメーターには、索引プラグインの適切な初期化に必要な構成情報を含める必要があります。分散環境では、索引プラグインの構成がクライアントとサーバーのプロセス間で移動するため、getProperties メソッドの実装と一緒に setProperties メソッドの実装が必要です。以下に、このメソッドの実装例を示します。

```
setProperties(Properties properties)

// setProperties method sample code
    public void setProperties(Properties properties) {
        ivIndexProperties = properties;

        String ivRangeIndexString = properties.getProperty("rangeIndex");
        if (ivRangeIndexString != null && ivRangeIndexString.equals("true")) {
            setRangeIndex(true);
        }
        setName(properties.getProperty("indexName"));
        setAttributeName(properties.getProperty("attributeName"));

        String ivFieldAccessAttributeString = properties.getProperty("fieldAccessAttribute");
        if (ivFieldAccessAttributeString != null && ivFieldAccessAttributeString.equals("true")) {
            setFieldAccessAttribute(true);
        }

        String ivPOJOKeyIndexString = properties.getProperty("POJOKeyIndex");
        if (ivPOJOKeyIndexString != null && ivPOJOKeyIndexString.equals("true")) {
            setPOJOKeyIndex(true);
        }
    }
```

getProperties メソッド

getProperties メソッドは、MapIndexPlugin インスタンスから索引プラグインの構成を抽出します。抽出したプロパティーを使用して、別の MapIndexPlugin インスタンスを初期化し内部状態が同一になるようにすることができます。分散環境では、

getProperties メソッドと setProperties メソッドの実装が必要です。以下に、getProperties メソッドの実装例を示します。

```
getProperties()

// getProperties method sample code
    public Properties getProperties() {
        Properties p = new Properties();
        p.put("indexName", indexName);
        p.put("attributeName", attributeName);
        p.put("rangeIndex", ivRangeIndex ? "true" : "false");
        p.put("fieldAccessAttribute", ivFieldAccessAttribute ? "true" : "false");
        p.put("POJOKeyIndex", ivPOJOKeyIndex ? "true" : "false");
        return p;
    }
```

setEntityMetadata メソッド

setEntityMetadata メソッドは、初期化時に WebSphere eXtreme Scale ランタイムにより呼び出され、関連する BackingMap の EntityMetadata を MapIndexPlugin インスタンスに設定します。 EntityMetadata は、タプル・オブジェクトの索引のサポートに必要です。タプルとは、エンティティー・オブジェクトまたはそのキーを表すデータ・セットです。 BackingMap がエンティティー用である場合は、このメソッドを実装する必要があります。

以下のコード例は、setEntityMetadata メソッドを実装します。

```
setEntityMetadata(EntityMetadata entityMetadata)

// setEntityMetadata method sample code
    public void setEntityMetadata(EntityMetadata entityMetadata) {
        ivEntityMetadata = entityMetadata;
        if (ivEntityMetadata != null) {
            // this is a tuple map
            TupleMetadata valueMetadata = ivEntityMetadata.getValueMetadata();
            int numAttributes = valueMetadata.getNumAttributes();
            for (int i = 0; i < numAttributes; i++) {
                String tupleAttributeName = valueMetadata.getAttribute(i).getName();
                if (attributeName.equals(tupleAttributeName)) {
                    ivTupleValueIndex = i;
                    break;
                }
            }

            if (ivTupleValueIndex == -1) {
                // did not find the attribute in value tuple, try to find it on key tuple.
                // if found on key tuple, implies key indexing on one of tuple key attributes.
                TupleMetadata keyMetadata = ivEntityMetadata.getKeyMetadata();
                numAttributes = keyMetadata.getNumAttributes();
                for (int i = 0; i < numAttributes; i++) {
                    String tupleAttributeName = keyMetadata.getAttribute(i).getName();
                    if (attributeName.equals(tupleAttributeName)) {
                        ivTupleValueIndex = i;
                        ivKeyTupleAttributeIndex = true;
                        break;
                    }
                }
            }

            if (ivTupleValueIndex == -1) {
                // if entityMetadata is not null and we could not find the
      // attributeName in entityMetadata, this is an
                // error
                throw new ObjectGridRuntimeException("Invalid attributeName.  Entity: " +
       ivEntityMetadata.getName());
            }
        }
    }
```

属性名メソッド

setAttributeName メソッドは、索引付けされる属性の名前を設定します。キャッシュ・オブジェクト・クラスは、索引付き属性に対し get メソッドを提供する必要が

あります。例えば、オブジェクトに属性 employeeName または EmployeeName がある場合、索引ではそのオブジェクトで getEmployeeName メソッドを呼び出し、属性値を抽出します。属性名はその get メソッド内の名前と同一にし、その属性では Comparable インターフェースを実装している必要があります。属性がブール・タイプである場合は、isAttributeName メソッドのパターンを使用することもできます。

getAttributeName メソッドは、索引付き属性の名前を戻します。

getAttribute メソッド

getAttribute メソッドは、指定したオブジェクトからの索引付き属性値を戻します。例えば、Employee オブジェクトに索引が付けられた employeeName という属性がある場合は、getAttribute メソッドを使用して、指定された Employee オブジェクトから employeeName の属性値を抽出できます。このメソッドは、分散 WebSphere eXtreme Scale 環境の場合には必須です。

```
getAttribute(Object value)

// getAttribute method sample code
    public Object getAttribute(Object value) throws ObjectGridRuntimeException {
        if (ivPOJOKeyIndex) {
            // In the POJO key indexing case, no need to get attribute from value object.
            // The key itself is the attribute value used to build the index.
            return null;
        }

        try {
            Object attribute = null;
            if (value != null) {
                // handle Tuple value if ivTupleValueIndex != -1
                if (ivTupleValueIndex == -1) {
                    // regular value
                    if (ivFieldAccessAttribute) {
                        attribute = this.getAttributeField(value).get(value);
                    } else {
                        attribute = getAttributeMethod(value).invoke(value, emptyArray);
                    }
                } else {
                    // Tuple value
                    attribute = extractValueFromTuple(value);
                }
            }
            return attribute;
        } catch (InvocationTargetException e) {
             throw new ObjectGridRuntimeException(
                    "Caught unexpected Throwable during index update processing,
           index name = " + indexName + ": " + t,
                    t);
        } catch (Throwable t) {
            throw new ObjectGridRuntimeException(
                    "Caught unexpected Throwable during index update processing,
          index name = " + indexName + ": " + t,
                    t);
        }
    }
```

TransactionCallback プラグイン

オプティミスティック・ロック・ストラテジーを使用しているときは、TransactionCallback プラグインによってキャッシュ・オブジェクトのバージョン管理および比較操作をカスタマイズすることができます。

com.ibm.websphere.objectgrid.plugins.OptimisticCallback インターフェースを実装するプラグ可能オプティミスティック・コールバック・オブジェクトを用意できます。エンティティー・マップの場合、ハイパフォーマンス OptimisticCallback プラグインが自動的に構成されます。

目的

OptimisticCallback インターフェースを使用して、マップの値としてオプティミスティック比較演算を提供します。オプティミスティック・ロック・ストラテジーを使用するときは、OptimisticCallback の実装が必要です。 WebSphere eXtreme Scale はデフォルトの OptimisticCallback 実装を提供します。ただし、通常、アプリケーションは独自の OptimisticCallback インターフェースの実装をプラグインする必要があります。詳しくは、「*製品概要*」でロック・ストラテジーに関する説明を参照してください。

デフォルト実装

eXtreme Scale フレームワークは、 OptimisticCallback インターフェースのデフォルト実装を提供します。この実装は、前のセクションで説明したように、アプリケーション提供の OptimisticCallback オブジェクトをアプリケーションがプラグインしない場合に使用します。デフォルト実装は、値のバージョン・オブジェクトとして、常に特殊値 NULL_OPTIMISTIC_VERSION を戻し、バージョン・オブジェクトの更新は行いません。このアクションにより、オプティミスティック比較はノーオペレーション関数になります。オプティミスティック・ロック・ストラテジーを使用しているとき、たいていの場合、ノーオペレーション関数が発生することは望まないと思われます。ご使用のアプリケーションが OptimisticCallback インターフェースを実装し、独自の OptimisticCallback 実装をプラグインする必要がある場合、デフォルト実装は使用しません。ただし、デフォルト提供の OptimisticCallback 実装が有用なシナリオが少なくとも 1 つ存在します。次のような状態について考えてみます。

- ローダーがバックアップ・マップ用にプラグインされている。
- ローダーが、OptimisticCallback プラグインからの支援なしに、オプティミスティック比較を実行する方法を認識している。

ローダーが、OptimisticCallback オブジェクトからの支援なしで、オプティミスティック・バージョン管理を認識できる方法について考えてみます。 ローダーは、値クラス・オブジェクトを認知し、オプティミスティック・バージョン管理の値としてどの値オブジェクトのフィールドを使用するかを認識しています。例えば、従業員マップの値オブジェクトに対して次のインターフェースを使用するとします。

```
public interface Employee
{
    // Sequential sequence number used for optimistic versioning.
    public long getSequenceNumber();
    public void setSequenceNumber(long newSequenceNumber);
    // Other get/set methods for other fields of Employee object.
}
```

この場合、ローダーは、getSequenceNumber メソッドを使用して、Employee 値オブジェクトの現行バージョン情報を取得できることを認識しています。ローダーは、戻り値を増分して、新規 Employee 値で永続ストレージを更新する前に、新規バージョン番号を生成します。 Java Database Connectivity (JDBC) ローダーの場合、過剰 SQL 更新ステートメントの WHERE 文節内の現行シーケンス番号が使用され、新規生成シーケンス番号を使用して、シーケンス番号列が新規シーケンス番号の値に設定されます。

このほかにも、オプティミスティック・バージョン管理に使用できる非表示の列を自動的に更新するなんらかのバックエンド提供の関数をローダーが利用する可能性

があります。場合によっては、ストアード・プロシージャーまたはトリガーを使用して、バージョン情報が入っている列を保守できるようにすることもあります。ローダーが、オプティミスティック・バージョン情報を保守するためにこれらの技法のいずれかを使用している場合は、アプリケーションが OptimisticCallback 実装を提供する必要はありません。ローダーは OptimisticCallback オブジェクトからの支援なしにオプティミスティック・バージョン管理を処理できるため、デフォルトの OptimisticCallback 実装を使用することができます。

エンティティーのデフォルト実装

エンティティーは、タプル・オブジェクトを使用して、ObjectGrid に保管されます。デフォルトの OptimisticCallback 実装は、非エンティティー・マップに対する振る舞いと同様の振る舞いをします。ただし、エンティティー内のバージョン・フィールドは、エンティティー記述子 XML ファイルの @Version アノテーションまたはバージョン属性を使用して識別されます。

バージョン属性の型は、int、Integer、short、Short、long、Long、java.sql.Timestamp のいずれかになります。エンティティーには、1 つだけのバージョン属性が定義される必要があります。バージョン属性は、構成時にのみ設定される必要があります。エンティティーが永続化されると、バージョン属性の値は変更してはなりません。

バージョン属性が構成されず、オプティミスティック・ロック・ストラテジーが使用される場合、タプルの全体の状態を使用して、タプル全体が暗黙的にバージョン設定されます。

以下の例では、Employee エンティティーに SequenceNumber という long バージョン属性が設定されています。

```
@Entity
public class Employee
{
private long sequence;
    // Sequential sequence number used for optimistic versioning.
    @Version
    public long getSequenceNumber() {
        return sequence;
    }
    public void setSequenceNumber(long newSequenceNumber) {
        this.sequence = newSequenceNumber;
    }
    // Other get/set methods for other fields of Employee object.
}
```

OptimisticCallback 実装の記述

OptimisticCallback 実装は、OptimisticCallback インターフェースを実装し、共通 ObjectGrid プラグイン規則に準拠する必要があります。

次のリストには、OptimisticCallback インターフェース内の各メソッドについての説明または考慮事項があります。

NULL_OPTIMISTIC_VERSION

この特殊値は、アプリケーション提供の OptimisticCallback 実装の代わりにデフォルトの OptimisticCallback 実装が使用される場合に、getVersionedObjectForValue メソッドによって戻されます。

getVersionedObjectForValue メソッド

getVersionedObjectForValue メソッドは、値のコピーを戻します。あるいはバージョン管理のために使用できる値の属性を戻すことがあります。このメソッドは、オブジェクトがトランザクションに関連付けられるたびに呼び出されます。ローダーがバックアップ・マップ内に設定されていない場合、バックアップ・マップは、コミット時刻にこの値を使用してオプティミスティック・バージョン管理比較を行います。オプティミスティック・バージョン管理比較は、このトランザクションが、このトランザクションによって変更されたマップ・エントリーに最初にアクセスしてから、バージョンが変更されていないことを確認するために、バックアップ・マップによって使用されます。別のトランザクションが既にこのマップ・エントリーのバージョンを変更している場合、バージョン比較は失敗し、バックアップ・マップは OptimisticCollisionException 例外を表示して、トランザクションを強制的にロールバックします。ローダーがプラグインされている場合、バックアップ・マップはオプティミスティック・バージョン管理情報を使用しません。代わりに、ローダーは、オプティミスティック・バージョン管理比較を行い、必要に応じてバージョン管理情報を更新する責任があります。ローダーは通常、ローダーの batchUpdate メソッドに渡される LogElement から、初期バージョン管理オブジェクトを取得します。このオブジェクトは、フラッシュ操作が発生するか、トランザクションがコミットされたときに呼び出されます。

次のコードは、EmployeeOptimisticCallbackImpl オブジェクトによって使用される実装を示しています。

```
public Object getVersionedObjectForValue(Object value)
{
    if (value == null)
    {
        return null;
    }
    else
    {
        Employee emp = (Employee) value;
        return new Long( emp.getSequenceNumber() );
    }
}
```

前の例に示すように、sequenceNumber 属性は、ローダーが予期するように、java.lang.Long オブジェクト内に戻されます。これは、ローダーの作成者と同一人物が EmployeeOptimisticCallbackImpl を作成したか、 EmployeeOptimisticCallbackImpl を実装した人物と協力して作業を行ったかのいずれかであることを示しています。例えば、これらの人物は getVersionedObjectForValue メソッドによって戻された値に合意しました。前に示したように、デフォルトの OptimisticCallback 実装は、バージョン・オブジェクトとして特殊値 NULL_OPTIMISTIC_VERSION を戻します。

updateVersionedObjectForValue メソッド

updateVersionedObjectForValue メソッドは、トランザクションが値を更新し、新バージョンのオブジェクトが必要になったときに呼び出されます。
getVersionedObjectForValue メソッドがこの値の属性を戻した場合、このメソッドは通常、属性値を新バージョンのオブジェクトに更新します。
getVersionedObjectForValue メソッドがこの値のコピーを戻した場合、このメソッドは通常、更新しません。デフォルトの OptimisticCallback は、getVersionedObjectForValue メソッドのデフォルト実装がバージョン・オブジェクトとして常に特殊値 NULL_OPTIMISTIC_VERSION を戻すため、更新は行いません。
次の例は、OptimisticCallback セクションで使用される EmployeeOptimisticCallbackImpl オブジェクトによって使用される実装を示しています。

```
public void updateVersionedObjectForValue(Object value)
{
    if ( value != null )
    {
        Employee emp = (Employee) value;
        long next = emp.getSequenceNumber() + 1;
        emp.updateSequenceNumber( next );
    }
}
```

前の例で示すように、sequenceNumber 属性は、次に getVersionedObjectForValue メソッドが呼び出されたときに、戻される java.lang.Long 値が元のシーケンス番号である長整数値を持つように、1 ずつ増分されます。例えば、1 を加えたものは、この従業員インスタンスの次のバージョン値です。この場合も、この例は、ローダーを作成者が EmployeeOptimisticCallbackImpl 実装の作成者と同一人物であるか、EmployeeOptimisticCallbackImpl 実装を実装した人物と協力して作業を行ったかのいずれかであることを示しています。

serializeVersionedValue メソッド

このメソッドは、指定されたストリームにバージョン値を書き込みます。実装によっては、バージョン値を使用して、オプティミスティック更新の衝突を識別することができます。一部の実装では、バージョン値は元の値のコピーです。それ以外の実装では、値のバージョンを示すシーケンス番号またはその他のいくつかのオブジェクトがあります。実際の実装が不明であるため、このメソッドは適切なシリアライゼーションを実行するために提供されます。デフォルト実装は writeObject メソッドを呼び出します。

inflateVersionedValue メソッド

このメソッドは、バージョン値のシリアライズ・バージョンを取り、実際のバージョン値オブジェクトを戻します。実装によっては、バージョン値を使用して、オプティミスティック更新の衝突を識別することができます。一部の実装では、バージョン値は元の値のコピーです。それ以外の実装では、値のバージョンを示すシーケンス番号またはその他のいくつかのオブジェクトがあります。実際の実装が不明であるため、このメソッドは適切なデシリアライゼーションを行うために提供されます。デフォルト実装は readObject メソッドを呼び出します。

アプリケーション提供の OptimisticCallback 実装の使用

アプリケーション提供の OptimisticCallback を BackingMap 構成に追加する場合、プログラマチック構成と XML 構成の 2 つの方法があります。

OptimisticCallback のプログラマチックなプラグイン

次の例は、grid1 ObjectGrid インターフェース内の従業員のバックアップ・マップ用に、アプリケーションで OptimisticCallback オブジェクトをプログラマチックにプラグインする方法を示しています。

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid1" );
BackingMap bm = dg.defineMap("employees");
EmployeeOptimisticCallbackImpl cb = new EmployeeOptimisticCallbackImpl();
bm.setOptimisticCallback( cb );
```

OptimisticCallback 実装をプラグインするための XML 構成方法

前の例の EmployeeOptimisticCallbackImpl オブジェクトは、OptimisticCallback インターフェースを実装する必要があります。次の例に示すように、アプリケーションは、XML ファイルを使用して、その OptimisticCallback オブジェクトをプラグインすることもできます。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
    <objectGrid name="grid1">
        <backingMap name="employees" pluginCollectionRef="employees" lockStrategy="OPTIMISTIC" />
    </objectGrid>
</objectGrids>

<backingMapPluginCollections>
    <backingMapPluginCollection id="employees">
        <bean id="OptimisticCallback" className="com.xyz.EmployeeOptimisticCallbackImpl" />
    </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

プラグイン・スロットの概要

プラグイン・スロットは、トランザクション・コンテキストを共用するプラグイン用に予約された、トランザクション・ストレージ・スペースです。これらのスロットは、eXtreme Scale プラグインが互いに通信し、トランザクション・コンテキストを共有し、トランザクション内でトランザクション・リソースが整合性を保って正しく使用されるようにする手段を提供します。

プラグインは、トランザクション・コンテキスト (データベース接続、Java Message Service (JMS) 接続など) をプラグイン・スロットに保管できます。保管されたトランザクション・コンテキストは、プラグイン・スロット番号 (トランザクション・コンテキストを検索するキーとして機能する) を認識しているいずれのプラグインからも検索することができます。

プラグイン・スロットの使用

プラグイン・スロットは TxID インターフェースの一部です。このインターフェースについて詳しくは、API 資料を参照してください。スロットは、ArrayList 配列のエントリーです。プラグインは、ObjectGrid.reserveSlot メソッドを呼び出し、すべて

の TxID オブジェクトでスロットが必要であることを示すことによって、ArrayList 配列のエントリーを予約できます。スロットが予約されると、プラグインはそれぞれの TxID オブジェクトのスロットにトランザクション・コンテキストを保管し、後でそれを取得することができます。 put および get 操作は、ObjectGrid.reserveSlot メソッドから返されるスロット番号によって調整されます。

プラグインには通常、ライフサイクルがあります。プラグイン・スロットの使用はプラグインのライフサイクルに適合する必要があります。通常、プラグインは初期化ステージの間にプラグイン・スロットを予約し、それぞれのスロットのスロット番号を取得する必要があります。標準的なランタイムでは、プラグインは適切なポイントで TxID オブジェクトの予約済みスロットにトランザクション・コンテキストを保管します。このポイントは、通常はトランザクションの開始時点です。当該のプラグインまたは他のプラグインが、スロット番号によってトランザクション内の TxID から保管されたトランザクション・コンテキストを取得することができます。

通常、プラグインは、トランザクション・コンテキストおよびスロットを削除することによってクリーンアップを実行します。以下のコード・スニペットは、TransactionCallback プラグインでプラグイン・スロットを使用する方法を示しています。

```
public class DatabaseTransactionCallback implements TransactionCallback {
    int connectionSlot;
    int autoCommitConnectionSlot;
    int psCacheSlot;
    Properties ivProperties = new Properties();

    public void initialize(ObjectGrid objectGrid) throws TransactionCallbackException {
        // In initialization stage, reserve desired plug-in slots by calling the
        //reserveSlot method of ObjectGrid and
        // passing in the designated slot name, TxID.SLOT_NAME.
        // Note: you have to pass in this TxID.SLOT_NAME that is designated
        // for application.
        try {
            // cache the returned slot numbers
            connectionSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
            psCacheSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
            autoCommitConnectionSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
        } catch (Exception e) {
            }
    }

    public void begin(TxID tx) throws TransactionCallbackException {
        // put transactional contexts into the reserved slots at the
        // beginning of the transaction.
        try {
            Connection conn = null;
            conn = DriverManager.getConnection(ivDriverUrl, ivProperties);
            tx.putSlot(connectionSlot, conn);
            conn = DriverManager.getConnection(ivDriverUrl, ivProperties);
            conn.setAutoCommit(true);
            tx.putSlot(autoCommitConnectionSlot, conn);
            tx.putSlot(psCacheSlot, new HashMap());
        } catch (SQLException e) {
            SQLException ex = getLastSQLException(e);
            throw new TransactionCallbackException("unable to get connection", ex);
        }
    }

    public void commit(TxID id) throws TransactionCallbackException {
        // get the stored transactional contexts and use them
        // then, clean up all transactional resources.
        try {
            Connection conn = (Connection) id.getSlot(connectionSlot);
            conn.commit();
            cleanUpSlots(id);
        } catch (SQLException e) {
            SQLException ex = getLastSQLException(e);
            throw new TransactionCallbackException("commit failure", ex);
        }
    }

    void cleanUpSlots(TxID tx) throws TransactionCallbackException {
        closePreparedStatements((Map) tx.getSlot(psCacheSlot));
        closeConnection((Connection) tx.getSlot(connectionSlot));
```

```
        closeConnection((Connection) tx.getSlot(autoCommitConnectionSlot));
    }

    /**
     * @param map
     */
    private void closePreparedStatements(Map psCache) {
        try {
            Collection statements = psCache.values();
            Iterator iter = statements.iterator();
            while (iter.hasNext()) {
                PreparedStatement stmt = (PreparedStatement) iter.next();
                stmt.close();
            }
        } catch (Throwable e) {
        }

    }

    /**
     * Close connection and swallow any Throwable that occurs.
     *
     * @param connection
     */
    private void closeConnection(Connection connection) {
        try {
            connection.close();
        } catch (Throwable e1) {
        }
    }

    public void rollback(TxID id) throws TransactionCallbackException
        // get the stored transactional contexts and use them
        // then, clean up all transactional resources.
        try {
            Connection conn = (Connection) id.getSlot(connectionSlot);
            conn.rollback();
            cleanUpSlots(id);
        } catch (SQLException e) {
        }
    }

    public boolean isExternalTransactionActive(Session session) {
        return false;
    }

    // Getter methods for the slot numbers, other plug-in can obtain the slot numbers
    // from these getter methods.

    public int getConnectionSlot() {
        return connectionSlot;
    }
    public int getAutoCommitConnectionSlot() {
        return autoCommitConnectionSlot;
    }
    public int getPreparedStatementSlot() {
        return psCacheSlot;
    }
```

以下のコード・スニペットは、直前の TransactionCallback プラグインの例で保管されたトランザクション・コンテキストを、Loader が取得する方法の例を示しています。

```
public class DatabaseLoader implements Loader
{
    DatabaseTransactionCallback tcb;
    public void preloadMap(Session session, BackingMap backingMap) throws LoaderException
    {
        // The preload method is the initialization method of the Loader.
        // Obtain interested plug-in from Session or ObjectGrid instance.
        tcb =
   (DatabaseTransactionCallback)session.getObjectGrid().getTransactionCallback();
    }
    public List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException
    {
        // get the stored transactional contexts that is put by tcb's begin method.
        Connection conn = (Connection)txid.getSlot(tcb.getConnectionSlot());
        // implement get here
        return null;
    }
    public void batchUpdate(TxID txid, LogSequence sequence) throws LoaderException,
   OptimisticCollisionException
    {
        // get the stored transactional contexts that is put by tcb's begin method.
        Connection conn = (Connection)txid.getSlot(tcb.getConnectionSlot());
        // implement batch update here ...
    }
}
```

外部トランザクション・マネージャー

通常、eXtreme Scale トランザクションは、session.begin メソッドで開始し、session.commit メソッドで終了します。しかし、eXtreme Scale が組み込まれている場合、外部トランザクション・コーディネーターがトランザクションの開始と終了を実行することができます。この場合、session.begin メソッドを呼び出す必要も、session.commit メソッドで終了する必要もありません。

外部トランザクションの調整

TransactionCallback プラグインは、eXtreme Scale セッションと外部トランザクションを関連づける isExternalTransactionActive(Session session) メソッドにより拡張されます。このメソッドのヘッダーは次のとおりです。

```
public synchronized boolean isExternalTransactionActive(Session session)
```

例えば、eXtreme Scale をセットアップして WebSphere Application Server および WebSphere Extended Deployment と統合することができます。

また、eXtreme Scale には、WebSphere という名前の組み込みプラグインもあります (150 ページの『TransactionCallback プラグイン』)。これは、WebSphere Application Server 環境向けにプラグインをビルドする方法を記述しますが、他のフレームワーク用にプラグインを適応させることもできます。

このシームレスな統合の鍵となるのが、WebSphere Application Server バージョン 5.x およびバージョン 6.x の ExtendedJTATransaction API の利用です。ただし、WebSphere Application Server バージョン 6.0.2 をご使用の場合は、このメソッドをサポートするために APAR PK07848 を適用する必要があります。次のサンプル・コードを使用して、ObjectGrid セッションを WebSphere Application Server トランザクション ID と関連付けます。

```
/**
* This method is required to associate an objectGrid session with a WebSphere
* Application Server transaction ID.
*/
Map/**/ localIdToSession;
public synchronized boolean isExternalTransactionActive(Session session)
{
    // remember that this localid means this session is saved for later.
    localIdToSession.put(new Integer(jta.getLocalId()), session);
    return true;
}
```

外部トランザクションの検索

TransactionCallback プラグインを使用するために、外部トランザクション・サービス・オブジェクトを検索しなければならない場合があります。WebSphere Application Server サーバーでは、次の例に示すように、名前空間から ExtendedJTATransaction オブジェクトを検索します。

```
public J2EETransactionCallback() {
    super();
    localIdToSession = new HashMap();
    String lookupName="java:comp/websphere/ExtendedJTATransaction";
    try
    {
        InitialContext ic = new InitialContext();
        jta = (ExtendedJTATransaction)ic.lookup(lookupName);
```

```
            jta.registerSynchronizationCallback(this);
        }
        catch(NotSupportedException e)
        {
            throw new RuntimeException("Cannot register jta callback", e);
        }
        catch(NamingException e){
            throw new RuntimeException("Cannot get transaction object");
        }
    }
```

他の製品の場合は、トランザクション・サービス・オブジェクトを検索するために同じような方法を使用することができます。

外部コールバックによりコミットを制御する

TransactionCallback プラグインは、eXtreme Scale セッションをコミットまたはロールバックするために、外部信号を受信する必要があります。この外部信号を受信するには、外部トランザクション・サービスからのコールバックを使用します。外部コールバック・インターフェースを実装し、それを外部トランザクション・サービスで登録する必要があります。例えば、WebSphere Application Server の場合、次の例に示すように、SynchronizationCallback インターフェースを実装します。

```
public class J2EETransactionCallback implements
 com.ibm.websphere.objectgrid.plugins.TransactionCallback, SynchronizationCallback {
   public J2EETransactionCallback() {
      super();
      String lookupName="java:comp/websphere/ExtendedJTATransaction";
      localIdToSession = new HashMap();
      try {
         InitialContext ic = new InitialContext();
         jta = (ExtendedJTATransaction)ic.lookup(lookupName);
         jta.registerSynchronizationCallback(this);
      } catch(NotSupportedException e) {
         throw new RuntimeException("Cannot register jta callback", e);
      }
      catch(NamingException e) {
         throw new RuntimeException("Cannot get transaction object");
      }
   }

   public synchronized void afterCompletion(int localId, byte[] arg1,boolean didCommit) {
      Integer lid = new Integer(localId);
      // find the Session for the localId
      Session session = (Session)localIdToSession.get(lid);
      if(session != null) {
          try {
            // if WebSphere Application Server is committed when
            // hardening the transaction to backingMap.
            // We already did a flush in beforeCompletion
            if(didCommit) {
               session.commit();
            } else {
               // otherwise rollback
               session.rollback();
            }
         } catch(NoActiveTransactionException e) {
            // impossible in theory
         } catch(TransactionException e) {
            // given that we already did a flush, this should not fail
         } finally {
            // always clear the session from the mapping map.
            localIdToSession.remove(lid);
         }
      }
   }

   public synchronized void beforeCompletion(int localId, byte[] arg1) {
      Session session = (Session)localIdToSession.get(new Integer(localId));
      if(session != null) {
         try {
            session.flush();
         } catch(TransactionException e) {
            // WebSphere Application Server  does not formally define
            // a way to signal the
            // transaction has failed so do this
            throw new RuntimeException("Cache flush failed", e);
```

```
        }
      }
    }
}
```

TransactionCallback プラグインでの eXtreme Scale API の使用

TransactionCallback プラグインは、eXtreme Scale 内での自動コミットを使用不可にします。eXtreme Scale の通常の使用パターンは以下のとおりです。

```
Session ogSession = ...;
ObjectMap myMap = ogSession.getMap("MyMap");
ogSession.begin();
MyObject v = myMap.get("key");
v.setAttribute("newValue");
myMap.update("key", v);
ogSession.commit();
```

この TransactionCallback プラグインが使用されている場合、 eXtreme Scale は、コンテナー管理対象トランザクションが存在するときにアプリケーションが eXtreme Scale を使用すると想定します。前出のコードの断片は、この環境で次のコードに変わります。

```
public void myMethod() {
   UserTransaction tx = ...;
   tx.begin();
   Session ogSession = ...;
   ObjectMap myMap = ogSession.getMap("MyMap");
   yObject v = myMap.get("key");
   v.setAttribute("newValue");
   myMap.update("key", v);
   tx.commit();
}
```

myMethod メソッドは、Web アプリケーションのシナリオに類似しています。アプリケーションは通常の UserTransaction インターフェースを使用してトランザクションを開始、コミット、およびロールバックします。 eXtreme Scale はコンテナー・トランザクションなどを自動的に開始およびコミットします。メソッドが TX_REQUIRES 属性を使用する Enterprise JavaBeans (EJB) メソッドの場合は、UserTransaction 参照および UserTransaction 呼び出しを除去してトランザクションを開始、コミットすると、メソッドが同じように動作します。この場合、コンテナーがトランザクションの開始と終了を行います。

ローダーの使用

eXtreme Scale ローダー・プラグインを使用すると、通常は、同一システムあるいは別システムの永続ストアに保持されるデータのメモリー・キャッシュとして ObjectGrid マップを動作させることができます。通常、データベースまたはファイル・システムは永続ストアとして使用されます。リモート Java 仮想マシン (JVM) は、データ・ソースとして使用することもでき、ObjectGrid を使用したハブ・ベースのキャッシュを作成できます。ローダーには、永続ストアとの間のデータの読み取りおよび書き込みを行うロジックがあります。

ローダーは、変更がバックアップ・マップに対して行われた場合、または、バックアップ・マップがデータ要求を満足できない (キャッシュ・ミス) 場合に呼び出されるバックアップ・マップ・プラグインです。

詳しくは、*製品概要* のキャッシングの概念に関する情報を参照してください。

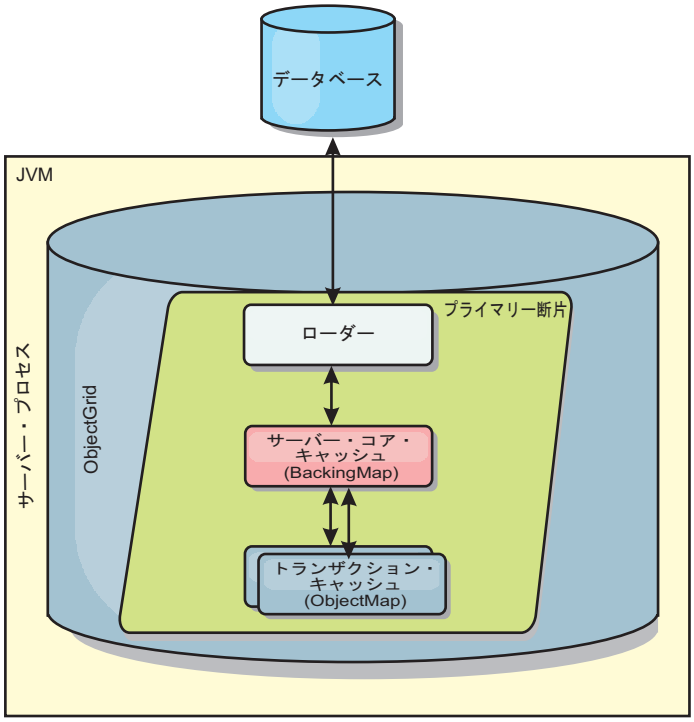

図 10. ローダー

WebSphere eXtreme Scale には、リレーショナル・データベース・バックエンドと統合する 2 つの組み込みローダーがあります。 Java Persistence API (JPA) ローダーは、JPA 仕様の OpenJPA 実装と Hibernate 実装の両方のオブジェクト・リレーショナル・マッピング (ORM) 機能を使用します。

ローダーの使用

ローダーを BackingMap 構成に追加するには、プログラマチック構成または XML 構成を使用します。ローダーは、バックアップ・マップとの間で以下の関係を持ちます。

- バックアップ・マップは 1 つしかローダーを持つことができません。
- クライアント・バックアップ・マップ (ニア・キャッシュ) はローダーを持つことができません。
- ローダー定義は複数のバックアップ・マップに適用できますが、各バックアップ・マップには独自のローダー・インスタンスがあります。

ローダーのプログラマチックなプラグイン

以下のコード・スニペットは、ObjectGrid API を使用してアプリケーションが提供するローダーを map1 のバックアップ・マップに接続する方法を示しています。

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
```

```
ObjectGrid og = ogManager.createObjectGrid( "grid" );
BackingMap bm = og.defineMap( "map1" );
MyLoader loader = new MyLoader();
loader.setDataBaseName("testdb");
loader.setIsolationLevel("read committed");
bm.setLoader( loader );
```

このスニペットでは、 MyLoader クラスは、com.ibm.websphere.objectgrid.plugins.Loader インターフェースを実装するアプリケーション提供のクラスであることが前提になります。 ObjectGrid の初期化後は、ローダーとバックアップ・マップとの関連付けを変更できないので、呼び出されている ObjectGrid インターフェースの initialize メソッドを起動する前にコードを実行する必要があります。初期化が起こった後に setLoader メソッドが呼び出された場合、IllegalStateException 例外が発生します。

アプリケーションが提供する Loader には、set プロパティーがあります。例では、MyLoader ローダーを使用して、リレーショナル・データベースの表からデータを読み書きします。ローダーにより、データベースの名前と SQL 分離レベルが指定されることが必要です。 MyLoader ローダーには、setDataBaseName メソッドと setIsolationLevel メソッドがあり、アプリケーションはこれらのメソッドを使用してこれら 2 つの Loader プロパティーを設定できます。

ローダーのプラグインの XML 構成アプローチ

アプリケーションが提供するローダーは、XML ファイルを使用して接続することも可能です。以下の例は、MyLoader ローダーが、同じデータベース名および分離レベル・ローダー・プロパティーで map1 バックアップ・マップに接続される方法を示しています。

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
    <objectGrid name="grid">
        <backingMap name="map1" pluginCollectionRef="map1" lockStrategy="OPTIMISTIC" />
    </objectGrid>
</objectGrids>
<backingMapPluginCollections>
    <backingMapPluginCollection id="map1">
        <bean id="Loader" className="com.myapplication.MyLoader">
            <property name="dataBaseName"
                      type="java.lang.String"
                      value="testdb"
                      description="database name" />
            <property name="isolationLevel"
                      type="java.lang.String"
                      value="read committed"
                      description="iso level" />
        </bean>
    </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

ローダーの作成

アプリケーションにローダー・プラグイン実装を組み込むができますLoader 実装クラスでは Loader インターフェースを実装し、WebSphere eXtreme Scale プラグインの共通規則に従う必要があります。

ローダー・プラグインの組み込み

この Loader インターフェースには、以下の定義があります。

```
public interface Loader
{
    static final SpecialValue KEY_NOT_FOUND;
    List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException;
    void batchUpdate(TxID txid, LogSequence sequence) throws
    LoaderException, OptimisticCollisionException;
    void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
}
```

詳しくは、「*管理ガイド*」でローダーに関する説明を参照してください。

get メソッド

バッキング・マップは Loader の get メソッドを呼び出し、keyList 引数として渡されるキー・リストに関連付けられた値を取得します。 get メソッドは、キー・リストにある各キーのうちの 1 つの値の java.lang.util.List リストを戻す必要があります。値リストに戻される最初の値はキー・リストの最初のキーに対応し、値リストに戻される 2 番目の値はキー・リストの 2 番目のキーに対応し、以降同様になります。キー・リスト内でキーの値を検出できなかったローダーは、 Loader インターフェースで定義された特別な KEY_NOT_FOUND 値オブジェクトを戻す必要があります。バッキング・マップは、null を有効な値として許可するよう構成できるので、キーを検出できない Loader が特別な KEY_NOT_FOUND オブジェクトを戻すことが極めて重要になります。この特殊値により、バッキング・マップは null 値とキーが検出できなかったため存在しない値とを区別できます。バックアップ・マップが null 値をサポートしない場合、存在しないキーについて KEY_NOT_FOUND オブジェクトではなく null 値を戻す Loader は、例外を発生します。

forUpdate 引数は、アプリケーションがマップ上で get メソッドまたは getForUpdate メソッドのいずれを呼び出したかを Loader に通知します。詳しくは、API 資料の ObjectMap インターフェースを参照してください。ローダーは、永続ストアへの並行アクセスを制御する、並行性制御ポリシーの実装を担当します。例えば、多くのリレーショナル・データベース管理システムは、リレーショナル・テーブルからデータを読み取るために使用される SQL SELECT ステートメントの FOR UPDATE 構文をサポートします。ローダーは、ブール値 true が、このメソッドの forUpdate パラメーターに引数値として渡されるかどうかに基づいて、SQL SELECT ステートメントの FOR UPDATE 構文を使用することを選択できます。通常、ローダーはペシミスティック並行性の制御ポリシーが使用される場合にのみ FOR UPDATE 構文を使用します。オプティミスティック並行性制御の場合、ローダーは SQL SELECT ステートメントで FOR UPDATE 構文を使用することはありません。ローダーは、そのローダーが使用している並行性制御ポリシーに基づいて forUpdate 引数の使用を判別します。

txid パラメーターの説明については、 150 ページの『TransactionCallback プラグイン』を参照してください。

batchUpdate メソッド

batchUpdate メソッドは、Loader インターフェースにおいて重要です。eXtreme Scale によって現在のすべての変更が Loader に適用される必要がある場合、必ずこのメソッドが呼び出されます。ローダーには、選択されたマップの変更のリストが与えられます。変更は繰り返され、バックエンドに適用されます。このメソッドは

現行の TxID 値および適用する変更を受け取ります。以下のサンプルは、一連の変更に対して繰り返し適応され、3 つの Java Database Connectivity (JDBC) ステートメント (INSERT、UPDATE、および DELETE) をバッチ処理します。

```
import java.util.Collection;
import java.util.Map;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
import com.ibm.websphere.objectgrid.plugins.Loader;
import com.ibm.websphere.objectgrid.plugins.LoaderException;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;

    public void batchUpdate(TxID tx, LogSequence sequence) throws LoaderException {
        // Get a SQL connection to use.
        Connection conn = getConnection(tx);
        try {
            // Process the list of changes and build a set of prepared
            // statements for executing a batch update, insert, or delete
            // SQL operation.
            Iterator iter = sequence.getPendingChanges();
            while (iter.hasNext()) {
                LogElement logElement = (LogElement) iter.next();
                Object key = logElement.getKey();
                Object value = logElement.getCurrentValue();
                switch (logElement.getType().getCode()) {
                case LogElement.CODE_INSERT:
                    buildBatchSQLInsert(tx, key, value, conn);
                    break;
                case LogElement.CODE_UPDATE:
                    buildBatchSQLUpdate(tx, key, value, conn);
                    break;
                case LogElement.CODE_DELETE:
                    buildBatchSQLDelete(tx, key, conn);
                    break;
                }
            }
            // Execute the batch statements that were built by above loop.
            Collection statements = getPreparedStatementCollection(tx, conn);
            iter = statements.iterator();
            while (iter.hasNext()) {
                PreparedStatement pstmt = (PreparedStatement) iter.next();
                pstmt.executeBatch();
            }
        } catch (SQLException e) {
            LoaderException ex = new LoaderException(e);
            throw ex;
        }
    }
```

前のサンプルは、LogSequence 引数の処理の高水準ロジックを示していますが、SQL の INSERT、UPDATE、または DELETE ステートメントがビルドされる方法の詳細については示されていません。示されているキーポイントには、以下のようなものがあります。

- getPendingChanges メソッドは、LogSequence 引数で呼び出され、ローダーが処理を必要とする LogElements のリストのイテレーターを取得します。
- LogElement.getType().getCode() メソッドを使用して、LogElement が SQL の INSERT、UPDATE、または DELETE 操作用であるかどうかを判断します。
- SQLException 例外はキャッチされ、バッチ更新中に発生した例外を報告するために発行される LoaderException 例外にチェーンされます。
- JDBC バッチ更新サポートは、作成する必要のあるバックエンドへの照会の数を最小化するために使用されます。

preloadMap メソッド

eXtreme Scale の初期化中に、定義された各バックアップ・マップが初期化されます。Loader がバッキング・マップにプラグインされると、バッキング・マップは Loader インターフェースで preloadMap メソッドを呼び出し、ローダーがバックエンドからデータをプリフェッチし、マップにデータをロードできるようにします。

以下のサンプルでは、Employee テーブルの最初の 100 行がデータベースから読み取られて、マップにロードされると仮定します。 EmployeeRecord クラスはアプリケーションが提供するクラスであり、従業員テーブルから読み取った従業員データを保持します。

```
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.TxID;
import com.ibm.websphere.objectgrid.plugins.Loader;
import com.ibm.websphere.objectgrid.plugins.LoaderException

    public void preloadMap(Session session, BackingMap backingMap) throws LoaderException {
        boolean tranActive = false;
        ResultSet results = null;
        Statement stmt = null;
        Connection conn = null;
        try {
            session.beginNoWriteThrough();
            tranActive = true;
            ObjectMap map = session.getMap(backingMap.getName());
            TxID tx = session.getTxID();
            // Get a auto-commit connection to use that is set to
            // a read committed isolation level.
            conn = getAutoCommitConnection(tx);
            // Preload the Employee Map with EmployeeRecord
            // objects. Read all Employees from table, but
            // limit preload to first 100 rows.
            stmt = conn.createStatement();
            results = stmt.executeQuery(SELECT_ALL);
            int rows = 0;
            while (results.next() && rows < 100) {
                int key = results.getInt(EMPNO_INDEX);
                EmployeeRecord emp = new EmployeeRecord(key);
                emp.setLastName(results.getString(LASTNAME_INDEX));
                emp.setFirstName(results.getString(FIRSTNAME_INDEX));
                emp.setDepartmentName(results.getString(DEPTNAME_INDEX));
                emp.updateSequenceNumber(results.getLong(SEQNO_INDEX));
                emp.setManagerNumber(results.getInt(MGRNO_INDEX));
                map.put(new Integer(key), emp);
                ++rows;
            }
            // Commit the transaction.
            session.commit();
            tranActive = false;
        } catch (Throwable t) {
            throw new LoaderException("preload failure: " + t, t);
        } finally {
            if (tranActive) {
                try {
                    session.rollback();
                } catch (Throwable t2) {
                    // Tolerate any rollback failures and
                    // allow original Throwable to be thrown.
                }
            }
            // Be sure to clean up other databases resources here
            // as well such a closing statements, result sets, etc.
        }
    }
```

このサンプルは以下のキーポイントを示します。

- preloadMap のバックアップ・マップはセッション引数として渡されるセッション・オブジェクトを使用します。
- Session.beginNoWriteThrough メソッドを使用して、begin メソッドの代わりにトランザクションを開始します。
- マップのロードに関してこのメソッドで発生する各 put 操作にローダーを呼び出すことはできません。
- ローダーは、従業員テーブルの列を EmployeeRecord Java オブジェクトのフィールドにマップすることができます。 ローダーは、発生したすべてのスロー可能な例外をキャッチし、 LoaderException 例外を、その例外にチェーンされているキャッチしたスロー可能な例外と一緒にスローします。
- finally ブロックにより、beginNoWriteThrough メソッドが呼び出される時点から commit メソッドが呼び出される時点までの間に発生するすべてのスロー可能な例

外は、確実に finally ブロックにアクティブなトランザクションをロールバックします。このアクションは、preloadMap メソッドによって開始されたすべてのトランザクションが、呼び出し側に戻される前に確実に完了させるために、重要です。 finally ブロックは、Java Database Connectivity (JDBC) 接続やその他の JDBC オブジェクトのクローズのような、必要とされる可能性のあるそれ以外のクリーンアップ・アクションを行う場所としても適切です。

preloadMap サンプルは、テーブルの行をすべて選択する SQL SELECT ステートメントを使用しています。アプリケーションが提供する Loader では、マップにプリロードするテーブルの数を制御するために、 1 つ以上の Loader プロパティーを設定します。

preloadMap メソッドは BackingMap の初期化中に 1 回しか呼び出されないので、1 回だけのローダー初期化コードの実行場所としても適切です。ローダーがバックエンドからデータをプリフェッチせず、データをマップにロードしないことを選択した場合であっても、それ以外に何らかの 1 回だけの初期化を実行し、さらに効率的なローダーの別のメソッドを作成する必要があると考えられます。以下は、TransactionCallback オブジェクトおよび OptimisticCallback オブジェクトを Loader のインスタンス変数としてキャッシングして、Loader の別のメソッドがこれらのオブジェクトにアクセスするためにメソッド呼び出しを行わなくても済むようにする例です。 BackingMap の初期化後に、TransactionCallback オブジェクトおよび OptimisticCallback オブジェクトを変更または置換できなくなるため、この ObjectGrid プラグインの値のキャッシングを行うことが可能です。これらのオブジェクト参照をローダーのインスタンス変数としてキャッシュに入れることは許容されます。

```
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.OptimisticCallback;
import com.ibm.websphere.objectgrid.plugins.TransactionCallback;

    // Loader instance variables.
    MyTransactionCallback ivTcb; // MyTransactionCallback

    // extends TransactionCallback
    MyOptimisticCallback ivOcb; // MyOptimisticCallback

    // implements OptimisticCallback
    // ...
    public void preloadMap(Session session, BackingMap backingMap) throws LoaderException
   [Replication programming]
        // Cache TransactionCallback and OptimisticCallback objects
        // in instance variables of this Loader.
        ivTcb = (MyTransactionCallback) session.getObjectGrid().getTransactionCallback();
        ivOcb = (MyOptimisticCallback) backingMap.getOptimisticCallback();
        // The remainder of preloadMap code (such as shown in prior example).
    }
```

複製フェイルオーバーに関係するプリロードおよび回復可能なプリロードについ詳しくは、*製品概要*で複製に関する説明を参照してください。

エンティティー・マップが設定されたローダー

ローダーがエンティティー・マップにプラグインされている場合は、ローダーでタプル・オブジェクトを処理する必要があります。タプル・オブジェクトは特別なエンティティー・データ・フォーマットです。ローダーでは、タプルとその他のデータ・フォーマット間でデータ変換を実行する必要があります。例えば、get メソッドにより、このメソッドに渡されるキーのセットに対応する値のリストが返されます。渡されたキーは Tuple のタイプに置かれ、キー・タプルと呼ばれます。ローダーが JDBC を使用しているデータベースでマップをパーシストすると想定した場

合、get メソッドは、各キー・タプルをエンティティー・マップにマップされているテーブルの 1 次キーの列に対応する属性値リストに変換し、データベースからデータをフェッチする基準として変換された属性値を使用する WHERE 文節が含まれている SELECT ステートメントを実行した後、返されたデータを値タプルに変換する必要があります。get メソッドは、データベースからデータを取得し、渡されたキー・タプルに対する値タプルにそのデータを変換した後、呼び出し元に渡されたタプル・キーのセットに対応する値タプルのリストを返します。get メソッドは 1 つの SELECT ステートメントを実行して一度にすべてのデータをフェッチするか、または各キー・タプルに対して SELECT ステートメントを実行します。データがエンティティー・マネージャーを使用して保管されるときにローダーをどのように使用するのかを示すプログラミングの詳細は、171 ページの『エンティティー・マップおよびタプルとのローダーの使用』を参照してください。

JPA ローダーのプログラミング考慮事項

Java Persistence API (JPA) ローダーは、JPA を使用してデータベースと対話するローダー・プラグイン実装です。JPA ローダーを使用するアプリケーションの開発時には、以下の考慮事項に注意してください。

eXtreme Scale エンティティーと JPA エンティティー

eXtreme Scale エンティティー・アノテーション、XML 構成、あるいはその両方を使用して、POJO クラスを eXtreme Scale エンティティーに指定することができます。また、JPA エンティティー・アノテーション、XML 構成、あるいはその両方を使用して、同じ POJO クラスを JPA エンティティーに指定することもできます。

eXtreme Scale エンティティー: eXtreme Scale エンティティーは、ObjectGrid マップに保管された永続データを表します。エンティティー・オブジェクトはキー・タプルおよび値タプルに変換され、キーと値のペアとしてマップに保管されます。タプルとは、画素属性の配列です。

JPA エンティティー: JPA エンティティーは、コンテナー管理パーシスタンスを使用して自動的にリレーショナル・データベースに保管された永続データを表します。データは、例えば、データベース内のデータベース・タプルのように、何らかのデータ・ストレージ・システム形式内の適切な形式で永続化されます。

eXtreme Scale エンティティーが永続化される場合、その関係は別のエンティティー・マップに保管されます。例えば、ShippingAddress エンティティーと 1 対多の関係にある Consumer エンティティーを永続化する場合、cascade-persist が有効になっている場合、ShippingAddress エンティティーは、タプル形式で shippingAddress マップに保管されます。JPA エンティティーを永続化する場合、JPA エンティティーも、cascade-persist が有効になっている場合、データベース表に対して永続化されます。 POJO クラスが、eXtreme Scale エンティティーと JPA エンティティーの両方として指定される場合、データは ObjectGrid エンティティー・マップとデータベースの両方に対して永続化できます。一般的な使用は以下のようになります。

- **プリロード・シナリオ**: JPA プロバイダーを使用してエンティティーがデータベースからロードされ、これを ObjectGrid エンティティー・マップに永続化します。

- **ローダー・シナリオ**: ローダー実装が、ObjectGrid エンティティー・マップに対してプラグインされ、ObjectGrid エンティティー・マップに保管されたエンティティーが JPA プロバイダーを使用してデータベースに対して永続化され、またはこれをデータベースからロードできるようにします。

また、POJO クラスが JPA エンティティーのみとして指定されることも一般的です。その場合、ObjectGrid マップに保管されるのは POJO インスタンスで、これに対してエンティティー・タプルは ObjectGrid エンティティー・ケースに保管されます。

エンティティー・マップに関するアプリケーション設計の考慮事項

JPALoader インターフェースをプラグインする場合、オブジェクト・インスタンスは直接 ObjectGrid マップに保管されます。

しかし、JPAEntityLoader をプラグインする場合、エンティティー・クラスは、eXtreme Scale エンティティーと JPA エンティティーの両方として指定されます。その場合、このエンティティーには、ObjectGrid エンティティー・マップと JPA パーシスタンス・ストアの 2 つの永続ストアがあるものとしてこれを取り扱います。アーキテクチャーは、JPALoader の場合よりも複雑になります。

JPAEntityLoader プラグインおよびアプリケーション設計の考慮事項に関して詳しくは、「*管理ガイド*」で JPAEntityLoader プラグインに関する情報を参照してください。エンティティー・マップに独自のローダーを実装する予定の場合にも、この情報が参考になります。

パフォーマンスの考慮事項

関係に対して適切な EAGER または LAZY のフェッチ・タイプを必ず設定してください。例えば、パフォーマンスの違いを説明するため、OpenJPA による 1 対多の双方向関係 Consumer と ShippingAddress を参考にします。この例では、JPA 照会では `select o from Consumer o where . . .` を実行して、バルク・ロードを行い、さらに関連するすべての ShippingAddress オブジェクトをロードしようとします。Consumer クラスに定義される 1 対多の関係は以下のようになります。

```
@Entity
public class Consumer implements Serializable {

    @OneToMany(mappedBy="consumer",cascade=CascadeType.ALL, fetch =FetchType.EAGER)
    ArrayList <ShippingAddress> addresses;
```

ShippingAddress クラスに定義された多対 1 の関係 consumer を以下に示します。

```
@Entity
public class ShippingAddress implements Serializable{

    @ManyToOne(fetch=FetchType.EAGER)
    Consumer consumer;
}
```

どちらの関係のフェッチ・タイプも `EAGER` で構成されている場合、OpenJPA では、N+1+1 の照会を使用してすべての Consumer オブジェクトおよび ShippingAddress オブジェクトを取得します。ここで、N は ShippingAddress オブジ

ェクトの数です。しかし、次のように ShippingAddress が LAZY のフェッチ・タイプを使用するように変更されると、2 つだけの照会を使用してすべてのデータを取得します。

```
@Entity
public class ShippingAddress implements Serializable{

    @ManyToOne(fetch=FetchType.LAZY)
    Consumer consumer;
}
```

照会は同じ結果を返しますが、照会の数が少なくなると、データベースとの相互作用が著しく減り、その結果、アプリケーション・パフォーマンスが向上する可能性があります。

JPAEntityLoader プラグイン

JPAEntityLoader プラグインは、EntityManager API を使用する場合に Java Persistence API (JPA) を使用してデータベースと通信する組み込みローダー実装です。 ObjectMap API を使用する場合は、JPALoader ローダーを使用します。

ローダーの詳細

ObjectMap API を使用してデータを保管する場合、JPALoader プラグインを使用します。 EntityManager API を使用してデータを保管する場合、JPAEntityLoader プラグインを使用します。

ローダーでは、2 つの主要な関数を提供しています。

1. **get**: get メソッドでは、JPAEntityLoader プラグインは、まず、javax.persistence.EntityManager.find(Class entityClass, Object key) メソッドを呼び出し、JPA エンティティーを検索します。次にこの JPA エンティティーをエンティティー・タプルに射影します。射影時には、タプル属性とアソシエーション・キーの両方が値タプルに保管されます。各キーの処理後、get メソッドは、エンティティー値タプルのリストを返します。
2. **batchUpdate**: batchUpdate メソッドでは、LogElement オブジェクトのリストを含む LogSequence オブジェクトを使用します。各 LogElement オブジェクトには、キー・タプルと値タプルが含まれています。JPA プロバイダーと対話するため、まず、キー・タプルに基づいて eXtreme Scale エンティティーを検出する必要があります。LogElement タイプに基づいて、以下の JPA 呼び出しを実行します。
   - **insert**: javax.persistence.EntityManager.persist(Object o)
   - **update**: javax.persistence.EntityManager.merge(Object o)
   - **remove**: javax.persistence.EntityManager.remove(Object o)

タイプが **update** の LogElement は、JPAEntityLoader に javax.persistence.EntityManager.merge(Object o) メソッドを呼び出させ、エンティティーをマージします。しかし、**update** タイプの LogElement は、com.ibm.websphere.objectgrid.em.EntityManager.merge(object o) 呼び出しか、eXtreme Scale EntityManager 管理インスタンスの属性変更のいずれかの結果である可能性があります。以下の例を参照してください。

```
com.ibm.websphere.objectgrid.em.EntityManager em = og.getSession().getEntityManager();
em.getTransaction().begin();
Consumer c1 = (Consumer) em.find(Consumer.class, c.getConsumerId());
c1.setName("New Name");
em.getTransaction().commit();
```

この例では、update タイプの LogElement が、マップ・コンシューマーの JPAEntityLoader に送られます。 JPA 管理エンティティーに対しては属性更新が呼び出されますが、JPA エンティティー・マネージャーに対しては javax.persistence.EntityManager.merge(Object o) メソッドが呼び出されます。この変更された振る舞いのため、このプログラミング・モデルの使用にはいくつかの制限があります。

アプリケーション設計の規則

エンティティーには、他のエンティティーとのリレーションシップがあります。リレーションシップが含まれ、JPAEntityLoader がプラグインされているアプリケーションを設計する場合、さらなる考慮が必要です。アプリケーションは、以下のセクションに記載しているように、次の 4 つの規則に従う必要があります。

リレーションシップの深さのサポートの制限

JPAEntityLoader がサポートされるのは、リレーションシップのないエンティティー、または 1 レベルのリレーションシップのエンティティーを使用する場合に限られます。`Company > Department > Employee` など、複数レベルのリレーションシップはサポートされません。

マップごとに 1 つのローダー

Consumer-ShippingAddress エンティティーのリレーションシップを例に使用して、EAGER フェッチを使用可能にして、1 件の consumer をロードする場合、すべての関連 ShippingAddress オブジェクトをロードできます。Consumer オブジェクトを永続化またはマージする場合、cascade-persist または cascade-merge が有効化されている場合は、関連する ShippingAddress オブジェクトを永続化またはマージできます。

Consumer エンティティー・タプルを保管するルート・エンティティーのローダーをプラグインすることはできません。各エンティティー・マップごとに 1 つのローダーを構成する必要があります。

JPA と eXtreme Scale に同じカスケード・タイプを設定

改めてエンティティー Consumer が ShippingAddress と 1 対多のリレーションシップがあるシナリオを考えます。このリレーションシップに cascade-persist が有効化されたシナリオを見てみます。Consumer オブジェクトが eXtreme Scale にパーシストされる場合、関連する N 個の ShippingAddress オブジェクトも eXtreme Scale にパーシストされます。

ShippingAddress に対して cascade-persist リレーションシップがある Consumer オブジェクトの persist 呼び出しは、JPAEntityLoader 層により 1 つの javax.persistence.EntityManager.persist(consumer) メソッド呼び出しと *N* 個の javax.persistence.EntityManager.persist(shippingAddress) メソッド呼び出しに変換されます。しかし、ShippingAddress オブジェクトに対するこれら N 個の余分の persist

呼び出しは、JPA プロバイダーの観点からは、cascade-persist 設定のため不必要です。この問題を解決するため、eXtreme Scale では、新たなメソッド isCascaded を LogElement インターフェースに提供しています。isCascaded メソッドは、LogElement が eXtreme Scale EntityManager のカスケード操作の結果であるかどうかを示します。この例では、ShippingAddress マップの JPAEntityLoader は、cascade persist 呼び出しにより *N* 個の LogElement オブジェクトを受け取ります。JPAEntityLoader は、isCascaded メソッドが true を返すことを検出し、JPA 呼び出しを行わずにこれらを無視します。したがって、JPA の観点からは、1 つの javax.persistence.EntityManager.persist(consumer) メソッド呼び出しのみを受け取ります。

カスケードを有効にしてエンティティーをマージしたり、エンティティーを除去する場合、同じ振る舞いが示されます。カスケードされた操作は、JPAEntityLoader プラグインによって無視されます。

カスケード・サポートの設計では、JPA プロバイダーに対して eXtreme Scale EntityManager 操作をやり直すことになります。これらの操作には、パーシスト、マージ、および 除去操作があります。カスケード・サポートを使用可能にするには、JPA のカスケード設定と eXtreme Scale EntityManager が同じであることを確認してください。

エンティティー更新の使用は注意すること

前述のようにカスケード・サポートの設計では、JPA プロバイダーに対して eXtreme Scale EntityManager 操作をやり直すことになります。アプリケーションが eXtreme Scale EntityManager に対して ogEM.persist(consumer) メソッドを呼び出す場合、cascade-persist 設定のために関連の ShippingAddress オブジェクトがパーシストされていても、JPAEntityLoader は JPA プロバイダーに対して jpAEM.persist(consumer) メソッドのみを呼び出します。

ただし、アプリケーションが管理エンティティーを更新する場合、この更新は JPAEntityLoader プラグインによる JPA merge 呼び出しに変換されます。このシナリオでは、複数レベルのリレーションシップおよびキー・アソシエーションのサポートは保証されません。この場合、ベスト・プラクティスは、管理エンティティーを更新する代わりに javax.persistence.EntityManager.merge(o) メソッドを使用することです。

エンティティー・マップおよびタプルとのローダーの使用

エンティティー・マネージャーは、すべてのエンティティー・オブジェクトをタプル・オブジェクトに変換してから、WebSphere eXtreme Scale マップに保管します。どのエンティティーにも、キー・タプルと値タプルがあります。このキーと値のペアは、エンティティーの関連 eXtreme Scale マップに保管されます。eXtreme Scale マップをローダーと共に使用する場合、ローダーは、タプル・オブジェクトと対話する必要があります。

概説

eXtreme Scale には、リレーショナル・データベースとの統合を簡素化するローダー・プラグインが含まれています。Java Persistence API (JPA) ローダーは、Java

Persistence API を使用して、データベースと対話し、エンティティー・オブジェクトを作成します。この JPA ローダーは、eXtreme Scale エンティティーと互換性があります。

タプル

タプルには、エンティティーの属性およびアソシエーションに関する情報が入っています。プリミティブ値は、プリミティブ・ラッパーを使用して保管されます。他のサポートされるオブジェクト・タイプは、そのネイティブ・フォーマットで保管されます。他のエンティティーに対するアソシエーションは、ターゲット・エンティティーのキーを表すキー・タプル・オブジェクトのコレクションとして保管されます。

各属性またはアソシエーションは、ゼロ・ベース索引を使用して保管されます。各属性の索引を getAttributePosition メソッド、または getAssociationPosition メソッドを使用して取得できます。位置が取得されると、その位置は eXtreme Scale ライフサイクルの実行期間中は変更されません。位置が変更される可能性があるのは、eXtreme Scale が再始動されるときです。タプルのエレメントの更新には、setAttribute メソッド、setAssociation メソッド、および setAssociations メソッドが使用されます。

重要: タプル・オブジェクトを作成または更新する場合、各プリミティブ・フィールドを非ヌル値で更新します。`int` などのプリミティブ値は、ヌルであってはなりません。値をデフォルトに変更しないと、パフォーマンスが低下するという問題が起こる可能性があり、エンティティー記述子 XML ファイル内の @Version アノテーションでマークされたフィールドやバージョン属性にも影響します。

以下の例では、タプルの処理方法について詳しく説明します。 この例の場合のエンティティーの定義について詳しくは、*製品概要*のエンティティー・マネージャーのチュートリアルにある Order エンティティー・スキーマに関する説明を参照してください。WebSphere eXtreme Scale は各エンティティーでローダーを使用するよう構成されています。また、取得されるのは Order エンティティーのみで、この特定のエンティティーは Customer エンティティーと多対 1 のリレーションシップを保有しています。属性名は `customer` で、これは OrderLine エンティティーと 1 対多のリレーションシップを保有しています。

プロジェクターを使用して、エンティティーから自動的にタプル・オブジェクトを作成します。プロジェクターを使用すると、Hibernate や JPA などのオブジェクト関係マッピング・ユーティリティーを使用する場合にローダーを簡素化することができます。

order.java

```
@Entity
public class Order
{
    @Id  String orderNumber;
    java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order") @OrderBy("lineNumber") List<OrderLine> lines;
}
```

customer.java

```
@Entity
public class Customer {
    @Id String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
```

orderLine.java

```
@Entity
public class OrderLine
{
    @Id @ManyToOne(cascade=CascadeType.PERSIST) Order order;
    @Id int lineNumber;
    @OneToOne(cascade=CascadeType.PERSIST) Item item;
    int quantity;
    double price;
}
```

Loader インターフェースを実装する OrderLoader クラスを以下のコードに示します。以下の例では、関連の TransactionCallback プラグインが定義されているものとします。

orderLoader.java

```
public class OrderLoader implements com.ibm.websphere.objectgrid.plugins.Loader {

 private EntityMetadata entityMetaData;
 public void batchUpdate(TxID txid, LogSequence sequence)
            throws LoaderException, OptimisticCollisionException {
        ...
        }
        public List get(TxID txid, List keyList, boolean forUpdate)
            throws LoaderException {
        ...
        }
        public void preloadMap(Session session, BackingMap backingMap)
            throws LoaderException {
  this.entityMetaData=backingMap.getEntityMetadata();
 }

}
```

eXtreme Scale からの preLoadMap メソッド呼び出し中に、インスタンス変数 entityMetaData が初期化されます。エンティティーを使用するようにマップが構成されている場合、*entityMetaData* 変数はヌルにはなりません。それ以外の場合、値は NULL です。

batchUpdate メソッド

batchUpdate メソッドを使用することで、アプリケーションがどのアクションを実行しようとしているかを知ることができます。挿入、更新、または削除操作に基づいて、データベースへの接続がオープンされ、作業が実行されます。キーと値のタイプは Tuple のため、これらを SQL ステートメントで意味を成す値に変換する必要があります。

以下のコードに示されているように、ORDER テーブルは、以下のデータ定義言語 (DDL) 定義を使用して作成されました。

```
CREATE TABLE ORDER (ORDERNUMBER VARCHAR(250) NOT NULL, DATE TIMESTAMP, CUSTOMER_ID VARCHAR(250))
ALTER TABLE ORDER ADD CONSTRAINT PK_ORDER PRIMARY KEY (ORDERNUMBER)
```

以下のコードは、Tuple を Object に変換する方法を示しています。

```
public void batchUpdate(TxID txid, LogSequence sequence)
            throws LoaderException, OptimisticCollisionException {
        Iterator iter = sequence.getPendingChanges();
        while (iter.hasNext()) {
            LogElement logElement = (LogElement) iter.next();
            Object key = logElement.getKey();
            Object value = logElement.getCurrentValue();

            switch (logElement.getType().getCode()) {
            case LogElement.CODE_INSERT:

1)              if (entityMetaData!=null) {

// The order has just one key orderNumber
2)                  String ORDERNUMBER=(String) getKeyAttribute("orderNumber", (Tuple) key);
// Get the value of date
3)                  java.util.Date unFormattedDate = (java.util.Date) getValueAttribute("date",(Tuple)value);
// The values are 2 associations. Lets process customer because
// the our table contains customer.id as primary key
4)                  Object[] keys= getForeignKeyForValueAssociation("customer","id",(Tuple) value);
                    //Order to Customer is M to 1. There can only be 1 key
5)                  String CUSTOMER_ID=(String)keys[0];
// parse variable unFormattedDate and format it for the database as formattedDate
6)                   String formattedDate = "2007-05-08-14.01.59.780272"; // formatted for DB2
// Finally, the following SQL statement to insert the record
7) //INSERT INTO ORDER (ORDERNUMBER, DATE, CUSTOMER_ID) VALUES(ORDERNUMBER,formattedDate, CUSTOMER_ID)
                }
                break;
            case LogElement.CODE_UPDATE:
                break;
            case LogElement.CODE_DELETE:
                break;
            }
        }

    }
// returns the value to attribute as stored in the key Tuple
 private Object getKeyAttribute(String attr, Tuple key) {
        //get key metadata
        TupleMetadata keyMD = entityMetaData.getKeyMetadata();
        //get position of the attribute
        int keyAt = keyMD.getAttributePosition(attr);
        if (keyAt > -1) {
            return key.getAttribute(keyAt);
        } else { // attribute undefined
            throw new IllegalArgumentException("Invalid position index for  "+attr);
        }

    }
// returns the value to attribute as stored in the value Tuple
    private Object getValueAttribute(String attr, Tuple value) {
        //similar to above, except we work with value metadata instead
        TupleMetadata valueMD = entityMetaData.getValueMetadata();

        int keyAt = valueMD.getAttributePosition(attr);
        if (keyAt > -1) {
            return value.getAttribute(keyAt);
        } else {
            throw new IllegalArgumentException("Invalid position index for  "+attr);
        }
    }
// returns an array of keys that refer to association.
    private Object[] getForeignKeyForValueAssociation(String attr, String fk_attr, Tuple value) {
        TupleMetadata valueMD = entityMetaData.getValueMetadata();
        Object[] ro;

        int customerAssociation = valueMD.getAssociationPosition(attr);
        TupleAssociation tupleAssociation = valueMD.getAssociation(customerAssociation);

        EntityMetadata targetEntityMetaData = tupleAssociation.getTargetEntityMetadata();

        Tuple[] customerKeyTuple = ((Tuple) value).getAssociations(customerAssociation);

        int numberOfKeys = customerKeyTuple.length;
        ro = new Object[numberOfKeys];

        TupleMetadata keyMD = targetEntityMetaData.getKeyMetadata();
        int keyAt = keyMD.getAttributePosition(fk_attr);
        if (keyAt < 0) {
            throw new IllegalArgumentException("Invalid position index for  " + attr);
        }
        for (int i = 0; i < numberOfKeys; ++i) {
            ro[i] = customerKeyTuple[i].getAttribute(keyAt);
        }

        return ro;

    }
```

1. entityMetaData が非ヌルであることを確認します。これは、そのキーと値のキャッシュ・エントリーのタイプが Tuple であることを意味します。 entityMetaData から Key TupleMetaData が取り出されます。これは、Order メタデータのキー部分のみを実際に反映したものです。
2. KeyTuple を処理して Key Attribute orderNumber の値を取得します。
3. ValueTuple を処理して属性の日付の値を取得します。
4. ValueTuple を処理して、関連するカスタマーから Keys の値を取得します。
5. CUSTOMER_ID を抽出します。リレーションシップをベースにして、Order には単一のカスタマーのみが存在し、ユーザーは単一のキーのみを保有することができます。そのため、キーのサイズは 1 です。簡単にするために、フォーマットを訂正する日付の構文解析はスキップします。
6. これは挿入操作のため、SQL ステートメントがデータ・ソース接続に渡されて、挿入操作が完了されます。

トランザクション区分およびデータベースへのアクセスは、162 ページの『ローダーの作成』で取り上げています。

get メソッド

キャッシュ内でキーが検出されなかった場合は、ローダー・プラグインの get メソッドを呼び出し、キーを検出します。

キーは Tuple です。最初のステップは Tuple から、SELECT SQL ステートメントに渡すことができるプリミティブ値への変換を行います。データベースからすべての属性を取得したら、Tuple に変換する必要があります。以下のコードは Order クラスを示しています。

```
public List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException {
  System.out.println("OrderLoader: Get called");
  ArrayList returnList = new ArrayList();

1)  if (entityMetaData != null) {
   int index=0;
   for (Iterator iter = keyList.iterator(); iter.hasNext();) {
2)    Tuple orderKeyTuple=(Tuple) iter.next();

    // The order has just one key orderNumber
3)    String ORDERNUMBERKEY = (String) getKeyAttribute("orderNumber",orderKeyTuple);
    //We need to run a query to get values of
4)    // SELECT CUSTOMER_ID, date FROM ORDER WHERE ORDERNUMBER='ORDERNUMBERKEY'

5)    //1) Foreign key: CUSTOMER_ID
6)    //2) date
    // Assuming those two are returned as
7)                              String  CUSTOMER_ID = "C001"; // Assuming Retrieved and initialized
8)    java.util.Date retrievedDate = new java.util.Date();
                                // Assuming this date reflects the one in database

    // We now need to convert this data into a tuple before returning

    //create a value tuple
9)    TupleMetadata valueMD = entityMetaData.getValueMetadata();
    Tuple valueTuple=valueMD.createTuple();


    //add retrievedDate object to Tuple
    int datePosition = valueMD.getAttributePosition("date");
10)    valueTuple.setAttribute(datePosition, retrievedDate);

    //Next need to add the Association
11)    int customerPosition=valueMD.getAssociationPosition("customer");
    TupleAssociation customerTupleAssociation =
                                  valueMD.getAssociation(customerPosition);
    EntityMetadata customerEMD = customerTupleAssociation.getTargetEntityMetadata();
    TupleMetadata customerTupleMDForKEY=customerEMD.getKeyMetadata();
12)    int customerKeyAt=customerTupleMDForKEY.getAttributePosition("id");
```

```
    Tuple customerKeyTuple=customerTupleMDForKEY.createTuple();
    customerKeyTuple.setAttribute(customerKeyAt, CUSTOMER_ID);
13)    valueTuple.addAssociationKeys(customerPosition, new Tuple[] {customerKeyTuple});


14)    int linesPosition = valueMD.getAssociationPosition("lines");
    TupleAssociation linesTupleAssociation = valueMD.getAssociation(linesPosition);
    EntityMetadata orderLineEMD = linesTupleAssociation.getTargetEntityMetadata();
    TupleMetadata orderLineTupleMDForKEY = orderLineEMD.getKeyMetadata();
    int lineNumberAt = orderLineTupleMDForKEY.getAttributePosition("lineNumber");
    int orderAt = orderLineTupleMDForKEY.getAssociationPosition("order");

    if (lineNumberAt < 0 || orderAt < 0) {
     throw new IllegalArgumentException(
                                        "Invalid position index for lineNumber or order "+
                                        lineNumberAt + " " + orderAt);
    }
15) // SELECT LINENUMBER FROM ORDERLINE WHERE ORDERNUMBER='ORDERNUMBERKEY'
    // Assuming two rows of line number are returned with values 1 and 2

    Tuple orderLineKeyTuple1 = orderLineTupleMDForKEY.createTuple();
    orderLineKeyTuple1.setAttribute(lineNumberAt, new Integer(1));// set Key
    orderLineKeyTuple1.addAssociationKey(orderAt, orderKeyTuple);

    Tuple orderLineKeyTuple2 = orderLineTupleMDForKEY.createTuple();
    orderLineKeyTuple2.setAttribute(lineNumberAt, new Integer(2));// Init Key
    orderLineKeyTuple2.addAssociationKey(orderAt, orderKeyTuple);

16)    valueTuple.addAssociationKeys(linesPosition, new Tuple[]
                                    {orderLineKeyTuple1, orderLineKeyTuple2 });

    returnList.add(index, valueTuple);

    index++;

   }
  }else {
   // does not support tuples
  }
  return returnList;
 }
```

1. get メソッドは、ローダーが取り出すキーと要求を ObjectGrid キャッシュによって検出できなかった場合に呼び出されます。entityMetaData 値を検査し、非ヌルであれば処理を続行します。
2. keyList に Tuple が含まれます。
3. 属性 orderNumber の値を取得します。
4. 日付 (値) およびカスタマー ID (外部キー) を取得するには、照会を実行します。
5. CUSTOMER_ID は、アソシエーション・タプルで設定する必要がある外部キーです。
6. 日付は値で、事前に設定されている必要があります。
7. この例は JDBC 呼び出しを実行しないため、CUSTOMER_ID が想定されます。
8. この例は JDBC 呼び出しを実行しないため、日付が想定されます。
9. 値 Tuple を作成します。
10. その位置をベースにして、Tuple に日付の値を設定します。
11. Order には 2 つのアソシエーションがあります。まず、Customer エンティティーを参照する属性 customer から開始します。Tuple に設定する ID の値が必要です。
12. カスタマー・エンティティー上で ID の位置を検索します。
13. アソシエーション・キーの値のみを設定します。
14. また、行はカスタマー・アソシエーションの場合と同様にアソシエーション・キーのグループとしてセットアップする必要があるアソシエーションです。

15. このオーダーと関連する lineNumber のキーをセットアップする必要があるため、SQL を実行して lineNumber の値を取得します。
16. valueTuple でアソシエーション・キーをセットアップします。これで BackingMap に戻される Tuple の作成が完了します。

このトピックには、タプルの作成手順、および Order エンティティーの説明のみが含まれています。他のエンティティーや TransactionCallback プラグインと結び付けられているプロセス全体に対しても同様の手順を実行してください。詳しくは、150 ページの『TransactionCallback プラグイン』を参照してください。

レプリカ・プリロード・コントローラーを使用したローダーの作成

レプリカ・プリロード・コントローラーを使用した Loader は、com.ibm.websphere.objectgrid.plugins.ReplicaPreloadController インターフェースを実装することもできます。

概説

ReplicaPreloadController インターフェースは、プライマリー断片になるレプリカが、以前のプライマリー断片がプリロード・プロセスを完了したかどうかを認識する方法を提供するように設計されています。プリロードが部分的に完了している場合は、以前のプライマリーが完了していない部分をピックアップする情報が提供されます。ReplicaPreloadController インターフェースを実装すると、プライマリーとなるレプリカは、以前のプライマリーが完了していないプリロード・プロセスを続行し、プリロード全体が完了するまで続行します。

分散 WebSphere eXtreme Scale 環境では、マップは、レプリカを含み、初期化中に大容量のデータをプリロードすることも可能です。プリロードは Loader アクティビティーであり、初期化中のプライマリー・マップでのみ行われます。大容量のデータがプリロードされる場合は、プリロードの完了に長時間かかる場合があります。プライマリー・マップでプリロード・データのほとんどが処理されたものの、初期化中に不明な理由でプリロードが停止した場合、レプリカはプライマリーとなります。この場合には、通常、新しいプライマリーが無条件にプリロードを実行するため、以前のプライマリーによってプリロードされたデータは失われます。無条件プリロードにより、新しいプライマリーはプリロード・プロセスを初期状態から開始し、以前にプリロードされたデータは無視されます。以前のプライマリーがプリロード・プロセス中に完了しなかった部分を、新しいプライマリーにピックアップさせるには、ReplicaPreloadController インターフェースを実装した Loader を指定します。ReplicaPreloadController インターフェースについて詳しくは、API 資料を参照してください。

ローダーについて詳しくは、*管理ガイド*のローダーに関する説明を参照してください。通常のローダー・プラグインの作成については、 162 ページの『ローダーの作成』を参照してください。

ReplicaPreloadController インターフェースの定義は、以下のとおりです。

```
public interface ReplicaPreloadController
{
    public static final class Status
    {
        static public final Status PRELOADED_ALREADY = new Status(K_PRELOADED_ALREADY);
        static public final Status FULL_PRELOAD_NEEDED = new Status(K_FULL_PRELOAD_NEEDED);
```

```
        static public final Status PARTIAL_PRELOAD_NEEDED = new Status(K_PARTIAL_PRELOAD_NEEDED);
    }

    Status checkPreloadStatus(Session session, BackingMap bmap);
}
```

以下のセクションでは、Loader および ReplicaPreloadController インターフェースのいくつかのメソッドについて説明します。

checkPreloadStatus メソッド

Loader で ReplicaPreloadController インターフェースが実装されると、マップ初期化中に preloadMap メソッドが呼び出される前に、checkPreloadStatus メソッドが呼び出されます。このメソッドの戻り状況により、preloadMap メソッドが呼び出されるかどうかが決まります。このメソッドによって Status#PRELOADED_ALREADY が返された場合、preload メソッドは呼び出されません。返されない場合は、preload メソッドが実行されます。この動作によって、このメソッドは Loader 初期化メソッドとして機能します。このメソッドで Loader プロパティーを初期化する必要があります。このメソッドにより正しい状況が返される必要があり、返されない場合は、プリロードは予定通りに動作しません。

```
public Status checkPreloadStatus(Session session, BackingMap backingMap) {
        // When a loader implements ReplicaPreloadController interface, this method
    // will be called before preloadMap method during map initialization.
        // Whether the preloadMap method will be called depends on teh returned status of this method.
        // So, this method also serve as Loader's initialization method.
        // This method has to return the right staus, otherwise the preload may not work as expected.

        // Note: must initialize this loader instance here.
        ivOptimisticCallback = backingMap.getOptimisticCallback();
        ivBackingMapName = backingMap.getName();
        ivPartitionId = backingMap.getPartitionId();
        ivPartitionManager = backingMap.getPartitionManager();
        ivTransformer = backingMap.getObjectTransformer();
        preloadStatusKey = ivBackingMapName + "_" + ivPartitionId;

        try {
            // get the preloadStatusMap to retrieve preload status that could be set by other JVMs..
            ObjectMap preloadStatusMap = session.getMap(ivPreloadStatusMapName);

            // retrieve last recorded preload data chunk index.
            Integer lastPreloadedDataChunk = (Integer) preloadStatusMap.get(preloadStatusKey);

            if (lastPreloadedDataChunk == null) {
                preloadStatus = Status.FULL_PRELOAD_NEEDED;
            } else {
                preloadedLastDataChunkIndex = lastPreloadedDataChunk.intValue();
                if (preloadedLastDataChunkIndex == preloadCompleteMark) {
                    preloadStatus = Status.PRELOADED_ALREADY;
                } else {
                    preloadStatus = Status.PARTIAL_PRELOAD_NEEDED;
                }
            }

            System.out.println("TupleHeapCacheWithReplicaPreloadControllerLoader.checkPreloadStatus()
     -> map = " + ivBackingMapName + ", preloadStatusKey = " + preloadStatusKey
                    + ", retrieved lastPreloadedDataChunk =" + lastPreloadedDataChunk + ",
         determined preloadStatus = "
                    + getStatusString(preloadStatus));

        } catch (Throwable t) {
            t.printStackTrace();
        }

        return preloadStatus;
    }
```

preloadMap メソッド

preloadMap メソッドの実行は、checkPreloadStatus メソッドから返された結果によって異なります。preloadMap メソッドが呼び出されると、このメソッドは、通常、指定されたプリロード状況マップからプリロード状況の情報を取得し、どのようにプリロードを進行するかを決定する必要があります。理想的には、プリロードが部分的に完了されており、どこから開始すればよいかが preloadMap メソッドで正確に認識されている必要があります。データ・プリロード中に、preloadMap メソッドは、指定されたプリロード状況マップでプリロード状況を更新する必要があります。プリロード状況マップに保管されているプリロード状況は、プリロード状況を確認する必要がある場合に、checkPreloadStatus メソッドによって取得されます。

```
public void preloadMap(Session session, BackingMap backingMap) throws LoaderException {
        EntityMetadata emd = backingMap.getEntityMetadata();
        if (emd != null && tupleHeapPreloadData != null) {
            // The getPreLoadData method is similar to fetching data from database.
            // These data will be push into cache as preload process.
            ivPreloadData = tupleHeapPreloadData.getPreLoadData(emd);

            ivOptimisticCallback = backingMap.getOptimisticCallback();
            ivBackingMapName = backingMap.getName();
            ivPartitionId = backingMap.getPartitionId();
            ivPartitionManager = backingMap.getPartitionManager();
            ivTransformer = backingMap.getObjectTransformer();
            Map preloadMap;

            if (ivPreloadData != null) {
                try {
                    ObjectMap map = session.getMap(ivBackingMapName);

                    // get the preloadStatusMap to record preload status.
                    ObjectMap preloadStatusMap = session.getMap(ivPreloadStatusMapName);

                    // Note: when this preloadMap method is invoked, the checkPreloadStatus
                    // has been called,
                    // Both preloadStatus and preloadedLastDataChunkIndex have been set.
                    // And the preloadStatus must be either PARTIAL_PRELOAD_NEEDED
         // or FULL_PRELOAD_NEEDED that
                    // will require a preload again.

                    // If large amount of data will be preloaded, the data usually is divided into
                    // few chunks and the preload process will process each chunk within its own tran.
                    // This sample only preload few entries and assuming each entry represent a chunk.
                    // so that the preload process an entry in a tran to simulate chunk preloading.

                    Set entrySet = ivPreloadData.entrySet();
                    preloadMap = new HashMap();
                    ivMap = preloadMap;

                    // The dataChunkIndex represent the data chunk that is in processing
                    int dataChunkIndex = -1;
                    boolean shouldRecordPreloadStatus = false;
                    int numberOfDataChunk = entrySet.size();
                    System.out.println("    numberOfDataChunk to be preloaded = "
              + numberOfDataChunk);

                    Iterator it = entrySet.iterator();
                    int whileCounter = 0;
                    while (it.hasNext()) {
                        whileCounter++;
                        System.out.println("preloadStatusKey = " + preloadStatusKey + " ,
                 whileCounter = " + whileCounter);

                        dataChunkIndex++;

                        // if the current dataChunkIndex <= preloadedLastDataChunkIndex
                        // no need to process, becasue it has been preloaded by other JVM before.
                        // only need to process dataChunkIndex > preloadedLastDataChunkIndex
                        if (dataChunkIndex <= preloadedLastDataChunkIndex) {
                            System.out.println("ignore current dataChunkIndex = " + dataChunkIndex
                                    + " that has been previously preloaded.");
                            continue;
                        }
```

```
// Note: This sample simulate data chunk as an entry.
// each key represent a data chunk for simplicity.
// If the primary server or shard stopped for unknown reason, the preload status that
// indicates the progress of preload should be available in preloadStatusMap.
// A replica that become a primary can get the preload status and determine how to preload
// again.
// Note: recording preload status should be in the same tran as putting data into cache; so that
// if tran rollback or error, the recorded preload status is the actual status.

Map.Entry entry = (Entry) it.next();
Object key = entry.getKey();
Object value = entry.getValue();
boolean tranActive = false;

System.out.println("processing data chunk. map = " + this.ivBackingMapName
        + ", current dataChunkIndex = " + dataChunkIndex + ", key = " + key);

try {
    shouldRecordPreloadStatus = false; // re-set to false
    session.beginNoWriteThrough();
    tranActive = true;

    if (ivPartitionManager.getNumOfPartitions() == 1) {
        // if just only 1 partition, no need to deal with partition.
        // just push data into cache
        map.put(key, value);
        preloadMap.put(key, value);
        shouldRecordPreloadStatus = true;
    } else if (ivPartitionManager.getPartition(key) == ivPartitionId) {
        // if map is partitioned, need to consider the partition key
        // only preload data that belongs to this partition.
        map.put(key, value);
        preloadMap.put(key, value);
        shouldRecordPreloadStatus = true;
    } else {
        // ignore this entry, because it does not belong to this partition.
    }

    if (shouldRecordPreloadStatus) {
        System.out.println("record preload status. map = " + this.ivBackingMapName
                + ", preloadStatusKey = " + preloadStatusKey + ", current dataChunkIndex ="
                + dataChunkIndex);
        if (dataChunkIndex == numberOfDataChunk) {
            System.out.println("record preload status. map = " + this.ivBackingMapName
                    + ", preloadStatusKey = " + preloadStatusKey + ", mark complete ="
                    + preloadCompleteMark);
            // means we are at the lastest data chunk, if commit successfully, record preload
            // complete.
            // at this point, the preload is considered to be done
            // use -99 as special mark for preload complete status.
            preloadStatusMap.get(preloadStatusKey);
            // a put follow a get will become update if the get return an object,
            // otherwise, it will be insert.
            preloadStatusMap.put(preloadStatusKey, new Integer(preloadCompleteMark));

        } else {
            // record preloaded current dataChunkIndex into preloadStatusMap
            // a put follow a get will become update if teh get return an object,
            // otherwise, it will be insert.
            preloadStatusMap.get(preloadStatusKey);
            preloadStatusMap.put(preloadStatusKey, new Integer(dataChunkIndex));
        }
    }

    session.commit();
    tranActive = false;

    // to simulate preloading large amount of data
    // put this thread into sleep for 30 secs.
    // The real app should NOT put this thread to sleep
    Thread.sleep(10000);

} catch (Throwable e) {
    e.printStackTrace();
    throw new LoaderException("preload failed with exception: " + e, e);
} finally {
    if (tranActive && session != null) {
        try {
```

```
                        session.rollback();
                    } catch (Throwable e1) {
                        // preload ignoring exception from rollback
                    }
                }
            }
        }

        // at this point, the preload is considered to be done for sure
        // use -99 as special mark for preload complete status.
        // this is a insurance to make sure the complete mark is set.
        // besides, when partitioning, each partition does not know when is its last data chunk.
        // so the following block serves as the overall preload status complete reporting.
        System.out.println("Overall preload status complete -> record preload status. map = "
                + this.ivBackingMapName + ", preloadStatusKey = " + preloadStatusKey + ", mark complete ="
                + preloadCompleteMark);
        session.begin();
        preloadStatusMap.get(preloadStatusKey);
        // a put follow a get will become update if teh get return an object,
        // otherwise, it will be insert.
        preloadStatusMap.put(preloadStatusKey, new Integer(preloadCompleteMark));
        session.commit();

        ivMap = preloadMap;
    } catch (Throwable e) {
        e.printStackTrace();
        throw new LoaderException("preload failed with exception: " + e, e);
    }
  }
 }
}
```

プリロード状況マップ

ReplicaPreloadController インターフェースの実装をサポートするには、プリロード状況マップを使用する必要があります。preloadMap メソッドは、常にプリロード状況マップに保管されているプリロード状況を最初に確認し、データがキャッシュにプッシュされた場合には、プリロード状況マップのプリロード状況を更新する必要があります。checkPreloadStatus メソッドはプリロード状況マップからプリロード状況を取得することができ、プリロード状況を判別して、その状況を呼び出し元に返します。プリロード状況マップは、レプリカ・プリロード・コントローラー Loader を持つ他のマップと同じ mapSet に置かれている必要があります。

LogElement および LogSequence

アプリケーションがトランザクション中にマップに変更を加えた場合、LogSequence オブジェクトはこれらの変更を追跡します。アプリケーションがマップ内のエントリーを変更する場合には、その変更の詳細を提供する、対応する LogElement オブジェクトが存在します。

アプリケーションがフラッシュを必要とするか、トランザクションにコミットすると必ず、特定のマップのための LogSequence オブジェクトにローダーが提供されます。ローダーは LogSequence オブジェクト内の LogElement オブジェクトで繰り返されて、各 LogElement オブジェクトをバックエンドに適用します。

ObjectGrid に登録されている ObjectGridEventListener リスナーも LogSequence オブジェクトを使用します。これらのリスナーには、コミット済みトランザクションの各マップに LogSequence オブジェクトが提供されます。アプリケーションはこれらのリスナーを使用して、従来のデータベースでのトリガーのような、変更に対する特定のエントリーを待機できます。

以下のログ関連インターフェースまたはクラスは、eXtreme Scale フレームワークによって提供されます。

- com.ibm.websphere.objectgrid.plugins.LogElement
- com.ibm.websphere.objectgrid.plugins.LogSequence
- com.ibm.websphere.objectgrid.plugins.LogSequenceFilter
- com.ibm.websphere.objectgrid.plugins.LogSequenceTransformer

LogElement インターフェース

LogElement は、トランザクション中のエントリーに関する操作を示します。LogElement オブジェクトには、その各種の属性を取得するためのいくつかのメソッドがあります。最も一般的に使用される属性は、getType() でフェッチされる type 属性と getCurrentValue() でフェッチされる current value 属性です。

type は、LogElement インターフェース内で定義される定数 INSERT、UPDATE、DELETE、EVICT、FETCH、または TOUCH のうちの 1 つで表わされます。

current value は、INSERT、UPDATE、または FETCH 操作の場合にその新規の値を表します。操作が TOUCH、DELETE、または EVICT の場合は、current value は NULL になります。ValueInterface が使用中である場合、この値を ValueProxyInfo へキャストできます。

LogElement インターフェースについて詳しくは、API 資料を参照してください。

LogSequence インターフェース

ほとんどのトランザクションで、マップ内の複数エントリーに対する操作が行われるため、複数の LogElement オブジェクトが作成されます。複数の LogElement オブジェクトのコンポジットとして動作するオブジェクトを作成する必要があります。LogSequence インターフェースは、LogElement オブジェクトのリストを含むことによってこの目的に対応します。

LogSequence インターフェースについて詳しくは、API 資料を参照してください。

LogElement および LogSequence の使用

LogElement と LogSequence は、eXtreme Scale や、操作が 1 つのコンポーネントまたはサーバーから別のコンポーネントまたはサーバーに伝搬されるときにユーザーによって作成された ObjectGrid プラグインによって、幅広く使用されています。例えば、LogSequence オブジェクトは、分散 ObjectGrid トランザクション伝搬機能によって変更を他のサーバーに伝えるために使用できます。あるいは、ローダーによってパーシスタンス・ストアに適用することもできます。 LogSequence は主に以下のインターフェースによって使用されます。

- com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener
- com.ibm.websphere.objectgrid.plugins.Loader
- com.ibm.websphere.objectgrid.plugins.Evictor
- com.ibm.websphere.objectgrid.Session

ローダーの例

このセクションでは、LogSequence および LogElement オブジェクトがローダーで使用される方法について説明します。ローダーは、永続ストアからデータをロードし、永続ストアにデータを保管するために使用されます。ローダー・インターフェースの batchUpdate メソッドは、以下のように LogSequence オブジェクトを使用します。

```
void batchUpdate(TxID txid, LogSequence sequence) throws
   LoaderException, OptimisticCollisionException;
```

ObjectGrid が現在のすべての変更をローダーに適用する必要がある場合に、batchUpdate メソッドが呼び出されます。ローダーには、マップのための LogElement オブジェクトのリストが、カプセル化されて LogSequence オブジェクトに与えられています。batchUpdate メソッドの実装は変更を繰り返し、それらの変更をバックエンドに適用する必要があります。以下のコード・スニペットは、ローダーが LogSequence オブジェクトを使用する方法を示しています。このスニペットは、一連の変更を繰り返し、INSERT、UPDATE、および DELETE という 3 つのバッチ Java Database Connectivity (JDBC) ステートメントをビルドします。

```
public void batchUpdate(TxID tx, LogSequence sequence) throws LoaderException
{
    // Get a SQL connection to use.
    Connection conn = getConnection(tx);
    try
    {
    // Process the list of changes and build a set of prepared
    // statements for executing a batch update, insert, or delete
    // SQL operations. The statements are cached in stmtCache.
    Iterator iter = sequence.getPendingChanges();
    while ( iter.hasNext() )
    {
        LogElement logElement = (LogElement)iter.next();
        Object key = logElement.getCacheEntry().getKey();
        Object value = logElement.getCurrentValue();
        switch ( logElement.getType().getCode() )
        {
            case LogElement.CODE_INSERT:
                buildBatchSQLInsert( key, value, conn );
                break;
            case LogElement.CODE_UPDATE:
                buildBatchSQLUpdate( key, value, conn );
                break;
            case LogElement.CODE_DELETE:
                buildBatchSQLDelete( key, conn );
                break;
        }
    }
    // Run the batch statements that were built by above loop.
    Collection statements = getPreparedStatementCollection( tx, conn );
    iter = statements.iterator();
    while ( iter.hasNext() )
    {
        PreparedStatement pstmt = (PreparedStatement) iter.next();
        pstmt.executeBatch();
    }
} catch (SQLException e)
{
    LoaderException ex = new LoaderException(e);
    throw ex;
}
}
```

前のサンプルは、LogSequence 引数処理の高水準ロジックを示していますが、SQL の INSERT、UPDATE、または DELETE ステートメントのビルド方法の詳細は示していません。getPendingChanges メソッドが LogSequence 引数で呼び出され、ローダーが処理する必要のある LogElement オブジェクトのイテレーターを取得します。また、LogElement.getType().getCode() メソッドを使用して、LogElement が SQL の挿入、更新、または削除操作に使用されるかどうかを判別します。

Evictor の例

Evictor で LogSequence および LogElement オブジェクトを使用することもできます。Evictor は、特定の基準に基づいてバッキング・マップからマップ・エントリーを除去するために使用します。 Evictor インターフェースの apply メソッドは、LogSequence を使用します。

```
/**
* This is called during cache commit to allow the evictor to track object usage
* in a backing map. This will also report any entries that have been successfully
* evicted.
*
* @param sequence LogSequence of changes to the map
*/
void apply(LogSequence sequence);
```

LogSequenceFilter および LogSequenceTransformer インターフェース

場合によっては、特定の基準の LogElement オブジェクトのみを受け入れ、その他のオブジェクトを拒否するように、 LogElement オブジェクトをフィルターに掛ける必要があります。例えば、何らかの基準に基づいて、特定の LogElement を直列化する場合があります。

LogSequenceFilter は、以下のメソッドでこの問題を解決します。

```
public boolean accept (LogElement logElement);
```

このメソッドは、操作で特定の LogElement を使用する必要がある場合は true を、その必要がない場合は false を返します。

LogSequenceTransformer は、LogSequenceFilter 関数を使用するクラスです。LogSequenceFilter を使用して一部の LogElement オブジェクトにフィルターを掛け、次に、その受け入れた LogElement オブジェクトを直列化します。このクラスには、2 つのメソッドがあります。最初のメソッドは以下のとおりです。

```
public static void serialize(Collection logSequences, ObjectOutputStream stream,
   LogSequenceFilter filter, DistributionMode mode) throws IOException
```

このメソッドにより、呼び出し元は、直列化プロセスに組み込む LogElements を判定するためのフィルターを提供できます。呼び出し元は、DistributionMode パラメーターを使用して直列化プロセスを制御します。例えば、分散モードが無効化のみである場合、値を直列化する必要はありません。このクラスの 2 番目のメソッドは、以下のような inflate メソッドです。

```
public static Collection inflate(ObjectInputStream stream, ObjectGrid
   objectGrid) throws IOException, ClassNotFoundException
```

inflate メソッドは、serialize メソッドによって作成されたログ・シーケンスの直列化済みフォームを、提供されたオブジェクト入力ストリームから読み取ります。

JPA を利用した eXtreme Scale の使用

Java Persistence API (JPA) は、 Java オブジェクトをリレーショナル・データベースにマップするための仕様です。JPA には、Java 言語メタデータ・アノテーション、XML 記述子、またはその両方を使用して、Java オブジェクトとリレーショナル・データベースとの間のマッピングを定義するための、完全なオブジェクト・リレーショナル・マッピング (ORM) 仕様が含まれています。オープン・ソースおよび商用の実装がいくつかあります。

JPA を使用するには、サポートされる JPA プロバイダー (OpenJPA や Hibernate など)、JAR ファイル、および `META-INF/persistence.xml` ファイルがクラスパスになければなりません。

クライアント・ベース JPA プリロード・ユーティリティーの概要

クライアント・ベース Java Persistence API (JPA) プリロード・ユーティリティーは、ObjectGrid に対するクライアント接続を使用して、データを eXtreme Scale バックアップ・マップにロードします。

この機能は、データベース照会の区画化が行えない場合に eXtreme Scale マップにデータをロードする作業を簡素化します。JPA ローダーなどのローダーを使用することもでき、これはデータを並行してロードできる場合は理想的です。

クライアント・ベース JPA プリロード・ユーティリティーは、OpenJPA または Hibernate JPA 実装のいずれかを使用して、データベースから ObjectGrid にデータをロードすることができます。WebSphere eXtreme Scale はデータベースまたは Java Database Connectivity (JDBC) と直接対話するわけではないため、OpenJPA または Hibernate がサポートする任意のデータベースを使用して ObjectGrid にデータをロードできます。

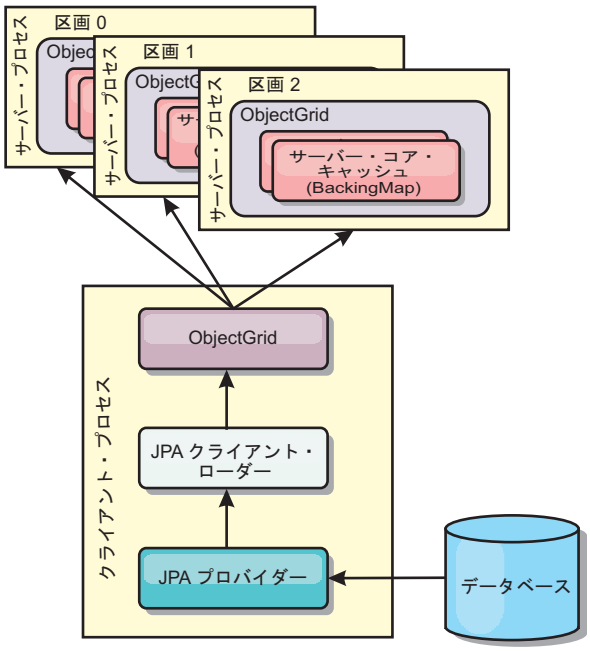

図 11. ObjectGrid へのロードに JPA 実装を使用するクライアント・ローダー

通常、ユーザー・アプリケーションが、パーシスタンス・ユニット名、エンティティー・クラス名、およびクライアント・ローダーに対する JPA 照会を提供します。クライアント・ローダーは、パーシスタンス・ユニット名に基づいて JPA エンティティー・マネージャーを取得し、このエンティティー・マネージャーを使用して、提供されたエンティティー・クラスと JPA 照会によりデータベースからデータを照会し、最終的にデータを分散 ObjectGrid マップにロードします。この照会にマルチレベルの関係が関与する場合、パフォーマンスを最適化するためにカスタム照会ストリングを使用できます。オプションで、パーシスタンス・プロパティー・マップを提供し、構成されたパーシスタンス・プロパティーをオーバーライドすることができます。

クライアント・ローダーは、以下の表に示すように 2 つの異なるモードでデータをロードできます。

表 11. クライアント・ローダーのモード

モード	説明
プリロード	すべてのエントリーをクリアし、バックアップ・マップにロードします。マップがエンティティー・マップの場合、ObjectGrid CascadeType.REMOVE オプションが有効になっている場合、関連するエンティティー・マップもすべてクリアされます。
再ロード	JPA 照会が ObjectGrid に対して実行され、照会に一致するマップ内のエンティティーをすべて無効化します。マップがエンティティー・マップの場合、ObjectGrid CascadeType.INVALIDATE オプションが有効になっている場合、関連するエンティティー・マップもすべてクリアされます。

いずれの場合も、JPA 照会を使用して、必要なエンティティーをデータベースから選択およびロードして、それらを ObjectGrid マップに保管します。ObjectGrid マップがエンティティー・マップでない場合は、JPA エンティティーは切り離され、直接保管されます。ObjectGrid マップがエンティティー・マップである場合は、JPA エンティティーは、ObjectGrid エンティティー・タプルとして保管されます。JPA 照会を指定するか、デフォルトの照会 `select o from EntityName o` を使用することができます。

クライアント・ベース JPA プリロード・ユーティリティーの構成について詳しくは、*プログラミング・ガイド*の説明を参照してください。

クライアント・ベースの JPA プリロード・ユーティリティー・プログラミング

クライアント・ベース Java Persistence API (JPA) プリロード・ユーティリティーは、ObjectGrid に対するクライアント接続を使用して、データを eXtreme Scale バックアップ・マップにロードします。データのプリロードおよび再ロードをアプリケーションに実装できます。

StateManager インターフェースの使用

StateManager インターフェースの setObjectGridState メソッドを使用して、ObjectGrid 状態の値を OFFLINE、ONLINE、QUIESCE または PRELOAD のいずれかに設定します。StateManager インターフェースは、ObjectGrid がまだオンラインでない場合に、他のクライアントがこれにアクセスしないようにします。

例えば、データのあるマップをロードする前に ObjectGrid 状態を PRELOAD に設定します。データ・ロードが完了したら、ObjectGrid 状態を ONLINE に戻します。詳しくは、「*管理ガイド*」で ObjectGrid の可用性の設定に関する情報を参照してください。

異なるマップを 1 つの ObjectGrid にプリロードする場合、ObjectGrid 状態を 1 度 `PRELOAD` に設定し、すべてのマップがデータ・ロードを終了した後に値を `ONLINE` に戻します。この調整は、ClientLoadCallback インターフェースで行うことができます。 ObjectGrid インスタンスからの最初の preStart 通知後に ObjectGrid 状態を `PRELOAD` に設定し、最後の postFinish 通知後にこれを `ONLINE` に戻します。複数の Java 仮想マシン間で調整する必要がある場合、ユーザーのアプリケーションで調整を行う必要があります。

Client ベースのプリロードの例

データ・プリロードのフローは、次のとおりです。

1. プリロードされるマップをクリアします。エンティティー・マップの場合、cascade-remove に構成されている関係がある場合、関連マップもクリアされます。
2. JPA に対する照会をバッチ内のエンティティーについて実行します。バッチ・サイズは、1000 です。
3. 各バッチについて、各区画のキー・リストおよび値リストを作成します。
4. 各区画に対して、データ・グリッド・エージェントを呼び出し、それが eXtreme Scale クライアントの場合、サーバー・サイドのデータを直接挿入または更新します。グリッドがローカル・インスタンスの場合は、ObjectGrid マップのデータを直接挿入または更新します。

次のサンプル・コード・スニペットは、単純なクライアントのロードを示しています。

```
// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);

ClientLoader c = ClientLoaderFactory.getClientLoader();

// Load the data
c.load(objectGrid, "CUSTOMER", "custPU", null,
    null, null, null, true, null);

// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);
```

この例では、CUSTOMER マップがエンティティー・マップとして構成されています。ObjectGrid エンティティー・メタデータ記述子 XML ファイルに構成されてい

る Customer エンティティー・クラスには、Order エンティティーと 1 対多の関係があります。Customer エンティティーは、Order エンティティーとの関係で、CascadeType.ALL オプションが有効になっています。

ClientLoader.load が呼び出される前に、ObjectGrid 状態が PRELOAD に設定されます。

ClientLoader.load メソッドで使用されているパラメーターは、次のとおりです。

1. **objectGrid** : ObjectGrid インスタンス。これは、クライアント・サイドの ObjectGrid インスタンスです。
2. ″**CUSTOMER**″ : ロードされるマップ。Customer は、Order エンティティーと cascade-all の関係になっていますので、Order エンティティーもロードされます。
3. ″**custPU**″ : Customer および Order エンティティーの JPA パーシスタンス・ユニット名。
4. **null** : persistenceProps マップがヌルです。これは、persistence.xml に構成されたデフォルトのパーシスタンス・プロパティーが使用されることを示しています。
5. **null** : entityClass がヌルに構成されています。これは、マップ ″CUSTOMER″ に対する ObjectGrid エンティティー・メタデータ記述子 XML に構成されたエンティティー・クラス、この場合は Customer.class に設定されます。
6. **null** : loadSql がヌルです。これは、JPA エンティティーの照会にデフォルトの ″select o from CUSTOMER o″ が使用されることを示しています。
7. **null** : 照会パラメーター・マップがヌルです。
8. **true** : これはデータ・ロード・モードがプリロードであることを示しています。したがって、cascade-remove 関係が CUSTOMER および ORDER マップ間にあるため、クリア操作が両者に対して呼び出され、ロード前にすべてのデータがクリアされます。
9. **null** : ClientLoaderCallback がヌルです。

必要なパラメーターについて詳しくは、API 資料の ClientLoader API を参照してください。

再ロードの例

マップの再ロードは、isPreload 引数が ClientLoader.load メソッドで false に設定されることを除き、マップのプリロードと同じです。

再ロード・モードの場合、データ・ロードのフローは、次のようになります。

1. 提供された照会を ObjectGrid マップ上で実行し、すべての結果を無効化します。エンティティー・マップの場合、CascadeType.INVALIDATE オプションで構成された関係がある場合、関連エンティティーもマップから無効化されます。
2. 提供された照会を JPA に対して実行し、バッチ内の JPA エンティティーを照会します。バッチ・サイズは、1000 です。
3. 各バッチについて、各区画のキー・リストおよび値リストを作成します。

4. 各区画に対して、データ・グリッド・エージェントを呼び出し、それが eXtreme Scale クライアントの場合、サーバー・サイドのデータを直接挿入または更新します。グリッドがローカル eXtreme Scale 構成の場合は、ObjectGrid マップのデータを直接挿入または更新します。

再ロードの例は次のようになります。

```
// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);

ClientLoader c = ClientLoaderFactory.getClientLoader();

// Load the data
String loadSql = "select c from CUSTOMER c
    where c.custId >= :startCustId and c.custId < :endCustId ";
Map<String, Long> params = new HashMap<String, Long>();
params.put("startCustId", 1000L);
params.put("endCustId", 2000L);

c.load(objectGrid, "CUSTOMER", "customerPU", null, null,
    loadSql, params, false, null);

// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);
```

プリロード・サンプルと比較した場合の主な違いは、loadSql とパラメーターを指定している点です。このサンプルでは、ID が 1000 と 2000 の間の Customer データのみを再ロードします。

この照会ストリングは、JPA 照会構文と eXtreme Scale エンティティー照会構文の両方に準拠している点に注意してください。照会ストリングは、2 度の実行、すなわち、1 度は ObjectGrid に対して実行して一致する ObjectGrid エンティティーを無効化し、2 度目は JPA に対して実行して一致する JPA エンティティーをロードするために使用されるので、これは重要なことです。

ローダー実装におけるクライアント・ローダーの呼び出し

ローダー・インターフェースには、preload メソッドがあります。

```
void preloadMap(Session session, BackingMap backingMap) throws
LoaderException;
```

このメソッドは、ローダーにデータをマップにプリロードするように通知します。ローダー実装では、クライアント・ローダーを使用して、データをそのすべての区画にプリロードすることができます。例えば、JPA ローダーでは、クライアント・ローダーを使用して、データをマップにプリロードします。

詳しくは、「*製品概要*」で JPA ローダーの概要のトピックを参照してください。

preloadMappreloadMap メソッドでクライアント・ローダーを使用してマップをプリロードする方法の例は以下のとおりです。この例では、まず、現在の区画番号がプリロード区画と同じかどうかをチェックします。区画番号がプリロード区画と同じでない場合は、何もアクションはありません。区画番号が一致する場合、クライア

ント・ローダーが呼び出されてデータがマップにロードされます。クライアント・ローダーの呼び出しが、1 つのみの区画で行われることが重要です。

```
ObjectGrid og = session.getObjectGrid();
int partitionId = backingMap.getPartitionId();
int numPartitions = backingMap.getPartitionManager().getNumOfPartitions();

// Only call client loader data in one partition
if (partitionId == preloadPartition) {

    ClientLoader loader = ClientLoaderFactory.getClientLoader();

    // Call the client loader to load the data
    try {

        loader.load(og, backingMap.getName(), txCallback.getPersistenceUnitName(),
     null, entityClass, null, null, true, null);
    } catch (ObjectGridException e) {
        LoaderException le = new LoaderException("Exception caught in ObjectMap " +
     ogName + "." + mapName);
        le.initCause(e);
        throw le;
    }
}
```

手動のクライアント・ロード

ClientLoader.load メソッドは、ほとんどのシナリオを満足するクライアント・ロード機能を提供しています。ただし、ClientLoader.load メソッドを使用しないでデータをロードする必要がある場合は、独自のプリロードを実装できます。

手動クライアント・ロードのテンプレートは次のようになります。

```
// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);


// Load the data
...

// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);
```

データをクライアント・サイドからロードする場合、DataGrid エージェントを使用してデータをロードすると、パフォーマンスが向上する可能性があります。DataGrid エージェントを使用すれば、すべてのデータ読み取りおよび書き込みが、サーバー・プロセスで行われます。また、必ず複数の区画に対して DataGrid エージェントが並列して実行されるようにアプリケーションを設計し、さらにパフォーマンスを改善するようにすることもできます。詳しくは、

DataGrid エージェントでデータ・プリロードを実装する場合、次の例を参照してください。

データ・プリロード実装を作成したら、以下のタスクを実行する一般のローダーを作成できます。

1. データベースからのデータをバッチで照会する。
2. 各区画のキー・リストおよび値リストを作成する。
3. 各区画について、agentMgr.callReduceAgent(agent, aKey) を呼び出して、スレッド内でそのサーバーのエージェントを実行します。スレッド内で実行すると、複数の区画で同時にエージェントを実行できます。

例: DataGrid エージェントを使用したデータ・プリロード

データをクライアント・サイドからロードする場合、DataGrid エージェントを使用してデータをロードすると、パフォーマンスが向上する可能性があります。DataGrid エージェントを使用すれば、すべてのデータ読み取りおよび書き込みが、サーバー・プロセスで行われます。また、必ず複数の区画に対して DataGrid エージェントが並列して実行されるようにアプリケーションを設計し、さらにパフォーマンスを改善するようにすることもできます。

DataGrid エージェントを使用してデータをロードする方法の例を次に示します。

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

import com.ibm.websphere.objectgrid.NoActiveTransactionException;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridRuntimeException;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.TransactionException;
import com.ibm.websphere.objectgrid.datagrid.ReduceGridAgent;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class InsertAgent implements ReduceGridAgent, Externalizable {

    private static final long serialVersionUID = 6568906743945108310L;

    private List keys = null;

    private List vals = null;

    protected boolean isEntityMap;

    public InsertAgent() {
    }

    public InsertAgent(boolean entityMap) {
        isEntityMap = entityMap;
    }

    public Object reduce(Session sess, ObjectMap map) {
        throw new UnsupportedOperationException(
            "ReduceGridAgent.reduce(Session, ObjectMap)");
    }

    public Object reduce(Session sess, ObjectMap map, Collection arg2) {
        Session s = null;
        try {
            s = sess.getObjectGrid().getSession();
            ObjectMap m = s.getMap(map.getName());
            s.beginNoWriteThrough();
            Object ret = process(s, m);
            s.commit();
            return ret;
        } catch (ObjectGridRuntimeException e) {
            if (s.isTransactionActive()) {
                try {
                    s.rollback();
                } catch (TransactionException e1) {
```

```
            } catch (NoActiveTransactionException e1) {
            }
        }
        throw e;
    } catch (Throwable t) {
        if (s.isTransactionActive()) {
            try {
                s.rollback();
            } catch (TransactionException e1) {
            } catch (NoActiveTransactionException e1) {
            }
        }
        throw new ObjectGridRuntimeException(t);
    }

}

public Object process(Session s, ObjectMap m) {
    try {

        if (!isEntityMap) {
            // In the POJO case, it is very straightforward,
            // we can just put everything in the
            // map using insert
            insert(m);
        } else {
            // 2. Entity case.
            // In the Entity case, we can persist the entities
            EntityManager em = s.getEntityManager();
            persistEntities(em);

        }
        return Boolean.TRUE;
    } catch (ObjectGridRuntimeException e) {
        throw e;
    } catch (ObjectGridException e) {
        throw new ObjectGridRuntimeException(e);
    } catch (Throwable t) {
        throw new ObjectGridRuntimeException(t);
    }

}

/**
 * Basically this is fresh load.
 * @param s
 * @param m
 * @throws ObjectGridException
 */
protected void insert(ObjectMap m) throws ObjectGridException {

    int size = keys.size();

    for (int i = 0; i < size; i++) {
        m.insert(keys.get(i), vals.get(i));
    }

}


protected void persistEntities(EntityManager em) {
    Iterator<Object> iter = vals.iterator();

    while (iter.hasNext()) {
        Object value = iter.next();
        em.persist(value);
    }
```

```
}

public Object reduceResults(Collection arg0) {
    return arg0;
}

public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    int v = in.readByte();
    isEntityMap = in.readBoolean();
    vals = readList(in);
    if (!isEntityMap) {
        keys = readList(in);
    }
}

public void writeExternal(ObjectOutput out) throws IOException {
    out.write(1);
    out.writeBoolean(isEntityMap);

    writeList(out, vals);
    if (!isEntityMap) {
        writeList(out, keys);
    }

}

public void setData(List ks, List vs) {
    vals = vs;
    if (!isEntityMap) {
        keys = ks;
    }
}

/**
 * @return Returns the isEntityMap.
 */
public boolean isEntityMap() {
    return isEntityMap;
}

static public void writeList(ObjectOutput oo, Collection l)
  throws IOException {
    int size = l == null ? -1 : l.size();
    oo.writeInt(size);
    if (size > 0) {
        Iterator iter = l.iterator();
        while (iter.hasNext()) {
            Object o = iter.next();
            oo.writeObject(o);
        }
    }
}

public static List readList(ObjectInput oi)
  throws IOException, ClassNotFoundException {
    int size = oi.readInt();
    if (size == -1) {
        return null;
    }


    ArrayList list = new ArrayList(size);
    for (int i = 0; i < size; ++i) {
        Object o = oi.readObject();
        list.add(o);
```

```
        }
        return list;
    }
}
```

JPA 時間ベース・データ・アップデーター

Java Persistence API (JPA) 時間ベース・データベース・アップデーターは、データベース内の最新の変更で ObjectGrid マップを更新します。

WebSphere eXtreme Scale グリッドの背後にあるデータベースに変更が直接行われた場合、それらの変更は同時には eXtreme Scale グリッドに反映されません。eXtreme Scale をメモリー内のデータベース処理スペースとして正しく実装するには、グリッドがデータベースと同期しなくなる可能性があることを考慮する必要があります。時間ベース・データベース・アップデーターは、Oracle 10g のシステム変更番号 (SCN) 機能および DB2® 9.5 の行変更タイム・スタンプを使用して、無効化または更新のためにデータベース内の変更をモニターします。また、アップデーターを使用すると、複数のアプリケーションが同じ目的でユーザー定義フィールドを設定することもできます。

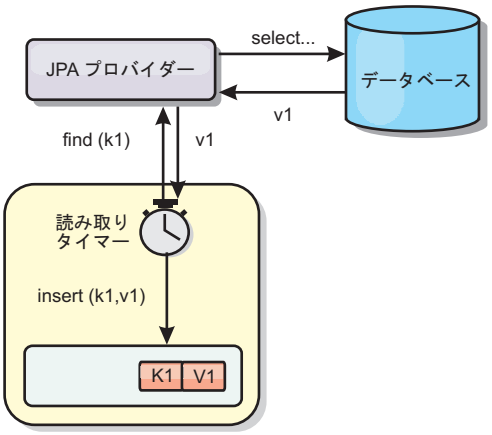

図 12. 定期的リフレッシュ

時間ベース・データベース・アップデーターでは、JPA インターフェースを使用して、定期的にデータベースを照会し、データベース内で新たに挿入され、更新されたレコードを示す JPA エンティティーを取得します。レコードを定期的に更新するために、データベース内のすべてのレコードには、レコードが最後に更新または挿入された時点の時刻または順序を識別するためのタイム・スタンプが必要です。タイム・スタンプはタイム・スタンプ形式になっている必要はありません。タイム・スタンプ値は、固有の漸増値を生成する場合は、整数または長形式とすることができます。

この機能は、いくつかの市販のデータベースで提供されています。

例えば、DB2 9.5 では、 ROW CHANGE TIMESTAMP 形式を使用する列を以下のように定義できます。

```
ROWCHGTS TIMESTAMP NOT NULL
      GENERATED ALWAYS
      FOR EACH ROW ON UPDATE AS
      ROW CHANGE TIMESTAMP
```

Oracle では、レコードのシステム変更番号を示す **ora_rowscn** という疑似列を使用することができます。

時間ベース・データベース・アップデーターは、ObjectGrid マップを次の 3 つの異なる方法で更新します。

1. INVALIDATE_ONLY: データベース内の対応するレコードが変更された場合に、ObjectGrid マップのエントリーを無効化します。
2. UPDATE_ONLY: データベース内の対応するレコードが変更された場合に、ObjectGrid マップのエントリーを更新します。ただし、データベースに新たに挿入されたレコードは、すべて無視されます。
3. INSERT_UPDATE: ObjectGrid マップの既存のエントリーをデータベースの最新の値で更新します。また、データベースに新たに挿入されたレコードが、すべて ObjectGrid マップに挿入されます。

JPA 時間ベース・データ・アップデーターについて詳しくは、「*管理ガイド*」に記載されている説明を参照してください。

JPA 時間ベース・アップデーターの開始

Java Persistence API (JPA) 時間ベース・アップデーターの開始時に、ObjectGrid マップがデータベース内の最新の変更で更新されます。

始める前に

時間ベース・アップデーターを構成します。「*管理ガイド*」で JPA 時間ベース・データ・アップデーターの構成に関する情報を参照してください。

このタスクについて

Java Persistence API (JPA) 時間ベース・データ・アップデーターがどのように機能するかについて詳しくは、 194 ページの『JPA 時間ベース・データ・アップデーター』を参照してください。

- 時間ベース・データベース・アップデーターを開始します。
  - **分散 eXtreme Scale に対する自動開始:**

    バックアップ・マップに対して timeBasedDBUpdate 構成を作成する場合、時間ベース・データベース・アップデーターは、分散 ObjectGrid プライマリー断片がアクティブ化された時点で自動的に開始されます。複数区画がある ObjectGrid の場合、時間ベース・データベース・アップデーターは、区画 0 でのみ開始されます。
  - **ローカル eXtreme Scale に対する自動開始:**

    バックアップ・マップに対して timeBasedDBUpdate 構成を作成する場合、時間ベース・データベース・アップデーターは、ローカル・マップがアクティブ化された時点で自動的に開始されます。
  - **手動開始:**

    また、時間ベース・データベース・アップデーターは、TimeBasedDBUpdater API を使用して、手動で開始または停止することもできます。

```
public synchronized void startDBUpdate(ObjectGrid objectGrid, String mapName,
 String punitName, Class entityClass, String timestampField, DBUpdateMode mode) {
```

1. **ObjectGrid**: ObjectGrid インスタンス (ローカルまたはクライアント)。
2. **mapName**: 時間ベース・データベース・アップデーターが開始されるバックアップ・マップの名前。
3. **punitName**: JPA エンティティー・マネージャー・ファクトリーを作成するための JPA パーシスタンス・ユニット名。デフォルト値は、persistence.xml ファイル内で定義された最初のパーシスタンス・ユニット名です。
4. **entityClass**: Java Persistence API (JPA) プロバイダーと対話するために使用されるエンティティー・クラス名。このエンティティー・クラス名は、エンティティー照会を使用した JPA エンティティーの取得に使用されます。
5. **timestampField**: データベース・バックエンド・レコードが最終更新または挿入された時間ないし順序を識別するための、エンティティー・クラスのタイム・スタンプ・フィールド。
6. **mode**: 時間ベース・データベース更新モード。INVALIDATE_ONLY タイプでは、データベース内の対応するレコードが変更された場合、ObjectGrid マップのエントリーが無効化されます。UPDATE_ONLY タイプは、ObjectGrid マップの既存のエントリーがデータベースの最新の値で更新されることを示しますが、データベースに新たに挿入されたレコードはすべて無視されます。INSERT_UPDATE タイプは、ObjectGrid マップの既存のエントリーがデータベースの最新の値で更新され、新たにデータベースに挿入されたレコードもすべて ObjectGrid マップに挿入されます。

時間ベース・データベース・アップデーターを停止したい場合は、以下のメソッドを呼び出せば、アップデーターを停止することができます。

```
public synchronized void stopDBUpdate(ObjectGrid objectGrid, String mapName)
```

ObjectGrid および mapName パラメーターは、startDBUpdate メソッドに渡されたものと同じにする必要があります。

- ご使用のデータベースにタイム・スタンプ・フィールドを作成します。
  - **DB2**

    オプティミスティック・ロック機能の一部として、DB2 9.5 では、行変更タイム・スタンプ機能を提供しています。ROW CHANGE TIMESTAMP 形式を使用して列 ROWCHGTS を次のように定義できます。

    ```
    ROWCHGTS TIMESTAMP NOT NULL
         GENERATED ALWAYS
         FOR EACH ROW ON UPDATE AS
         ROW CHANGE TIMESTAMP
    ```

    次に、アノテーションまたは構成によって、この列に対応するエンティティー・フィールドをタイム・スタンプ・フィールドとして指示することができます。以下に例を示します。

    ```
    @Entity(name = "USER_DB2")
    @Table(name = "USER1")
    public class User_DB2 implements Serializable {

        private static final long serialVersionUID = 1L;
    ```

```
    public User_DB2() {
    }

    public User_DB2(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;

    @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
    @Column(name = "ROWCHGTS", updatable = false, insertable = false)
    public Timestamp rowChgTs;
}
```

– **Oracle**

Oracle の場合、レコードのシステム変更番号用に疑似列 ora_rowscn があります。この列を同じ目的に使用することができます。時間ベース・データベース更新のタイム・スタンプ・フィールドとしてこの ora_rowscn フィールドを使用するエンティティーの例を以下に示します。

```
@Entity(name = "USER_ORA")
@Table(name = "USER1")
public class User_ORA implements Serializable  {

    private static final long serialVersionUID = 1L;

    public User_ORA() {
    }

    public User_ORA(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;

    @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
    @Column(name = "ora_rowscn", updatable = false, insertable = false)
    public long rowChgTs;
}
```

– **その他のデータベース**

その他のタイプのデータベースの場合、変更を追跡する表列を作成できます。列の値は、表を更新するアプリケーションによって手動で管理する必要があります。

Apache Derby データベースを例として取り上げます。変更番号をトラッキングするために列 ″ROWCHGTS″ を作成します。また、この表に対する最新変更番号もトラッキングされます。レコードが挿入または更新されるたびに、表の最新変更番号が増分され、レコードの ROWCHGTS 列の値がその増分された番号で更新されます。

```
@Entity(name = "USER_DER")
@Table(name = "USER1")
public class User_DER implements Serializable {

    private static final long serialVersionUID = 1L;

    public User_DER() {
    }

    public User_DER(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;

    @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
    @Column(name = "ROWCHGTS", updatable = true, insertable = true)
    public long rowChgTs;
}
```

OptimisticCallback プラグイン

オプティミスティック・ロック・ストラテジーを使用しているときは、OptimisticCallback プラグインによってキャッシュ・オブジェクトのバージョン管理および比較操作をカスタマイズすることができます。

com.ibm.websphere.objectgrid.plugins.OptimisticCallback インターフェースを実装するプラグ可能オプティミスティック・コールバック・オブジェクトを用意できます。エンティティー・マップの場合、ハイパフォーマンス OptimisticCallback プラグインが自動的に構成されます。

目的

OptimisticCallback インターフェースを使用して、マップの値としてオプティミスティック比較演算を提供します。オプティミスティック・ロック・ストラテジーを使用するときは、OptimisticCallback プラグインが必要です。この製品はデフォルトの OptimisticCallback 実装を提供しています。ただし、通常、アプリケーションは独自の OptimisticCallback インターフェースの実装をプラグインする必要があります。

デフォルト実装

eXtreme Scale フレームワークは、 OptimisticCallback インターフェースのデフォルト実装を提供します。この実装は、アプリケーション提供の OptimisticCallback オブジェクトをアプリケーションがプラグインしない場合に使用します。デフォルト実装は、値のバージョン・オブジェクトとして、常に特殊値 NULL_OPTIMISTIC_VERSION を戻し、バージョン・オブジェクトの更新は行いません。このアクションにより、オプティミスティック比較は「ノーオペレーション」関数になります。オプティミスティック・ロック・ストラテジーを使用しているとき、たいていの場合、「ノーオペレーション」関数が発生することは望まないと思われます。ご使用のアプリケーションが OptimisticCallback インターフェースを実装し、独自の OptimisticCallback 実装をプラグインする必要がある場合、デフォルト実装は使用しません。ただし、デフォルト提供の OptimisticCallback 実装が有用なシナリオが、少なくとも 1 つ存在します。次のような状態について考えてみます。

- ローダーがバックアップ・マップ用にプラグインされている。
- ローダーが、OptimisticCallback プラグインからの支援なしに、オプティミスティック比較を実行する方法を認識している。

ローダーが、OptimisticCallback オブジェクトからの支援なしで、オプティミスティック・バージョン管理を実行できる方法について考えてみます。 ローダーは、値クラス・オブジェクトを認知し、オプティミスティック・バージョン管理の値としてどの値オブジェクトのフィールドを使用するかを認識しています。例えば、従業員マップの値オブジェクトに対して次のインターフェースを使用するとします。

```
public interface Employee
{
    // Sequential sequence number used for optimistic versioning.
    public long getSequenceNumber();
    public void setSequenceNumber(long newSequenceNumber);
    // Other get/set methods for other fields of Employee object.
}
```

この例では、ローダーは、getSequenceNumber メソッドを使用して、Employee 値オブジェクトの現行バージョン情報を取得できることを認識しています。ローダーは、戻り値を増分して、新規 Employee 値で永続ストレージを更新する前に、新規バージョン番号を生成します。 Java Database Connectivity (JDBC) ローダーの場合、過剰 SQL UPDATE ステートメントの WHERE 文節内の現行シーケンス番号が使用され、新規生成シーケンス番号を使用して、シーケンス番号列が新規シーケンス番号の値に設定されます。このほかにも、オプティミスティック・バージョン管理に使用できる非表示の列を自動的に更新するなんらかのバックエンド提供の関数をローダーが利用する可能性があります。

状況によっては、ストアード・プロシージャーまたはトリガーを使用して、バージョン情報が入っている列を保守できるようにすることもあります。ローダーが、オプティミスティック・バージョン情報を保守するためにこれらの技法のいずれかを使用している場合は、アプリケーションが OptimisticCallback 実装を提供する必要はありません。デフォルトの OptimisticCallback 実装は、ローダーが OptimisticCallback オブジェクトからの支援なしにオプティミスティック・バージョン管理を処理できるため、このシナリオでは便利です。

エンティティーのデフォルト実装

エンティティーは、タプル・オブジェクトを使用して、ObjectGrid に保管されます。デフォルトの OptimisticCallback 実装の振る舞いは、非エンティティー・マップに対する振る舞いと似ています。ただし、エンティティー内のバージョン・フィールドは、エンティティー記述子 XML ファイルの @Version アノテーションまたはバージョン属性を使用して識別されます。

バージョン属性の型は、int、Integer、short、Short、long、Long、java.sql.Timestamp のいずれかになります。エンティティーにはバージョン属性を 1 つだけ定義することができます。バージョン属性は構成時にのみ設定するようにしてください。エンティティーが永続化されると、バージョン属性の値は変更してはなりません。

バージョン属性が構成されず、オプティミスティック・ロック・ストラテジーが使用される場合、タプルの全体の状態を使用して、タプル全体が暗黙的にバージョン設定されますが、これははるかに高コストになります。

以下の例では、Employee エンティティーに SequenceNumber という long バージョン属性が設定されています。

```
@Entity
public class Employee
{
private long sequence;
    // Sequential sequence number used for optimistic versioning.
    @Version
    public long getSequenceNumber() {
        return sequence;
    }
    public void setSequenceNumber(long newSequenceNumber) {
        this.sequence = newSequenceNumber;
    }
    // Other get/set methods for other fields of Employee object.
}
```

OptimisticCallback プラグインの記述

OptimisticCallback プラグインは、OptimisticCallback インターフェースを実装し、共通 ObjectGrid プラグイン規則に準拠する必要があります。詳しくは、API 資料中の OptimisticCallback インターフェースを参照してください。

次のリストには、OptimisticCallback インターフェース内の各メソッドについての説明または考慮事項があります。

NULL_OPTIMISTIC_VERSION

この特殊値は、OptimisticCallback 実装がバージョン検査を必要としない場合に、getVersionedObjectForValue メソッドによって戻されます。com.ibm.websphere.objectgrid.plugins.builtins.NoVersioningOptimisticCallback クラスの組み込みプラグイン実装では、このプラグイン実装を指定するとバージョン管理が使用不可になるため、この値が使用されます。

getVersionedObjectForValue メソッド

getVersionedObjectForValue メソッドは、バージョン管理のために使用できる値のコピーまたは値の属性を戻すことがあります。このメソッドは、オブジェクトがトラ

ンザクションに関連付けられるたびに呼び出されます。ローダーがバックアップ・マップ内にプラグインしていない場合、バックアップ・マップは、コミット時刻にこの値を使用してオプティミスティック・バージョン管理比較を行います。オプティミスティック・バージョン管理比較は、このトランザクションがこのトランザクションによって変更されたマップ・エントリーに最初にアクセスした後でバージョンが変更されていないことを確認するために、バックアップ・マップによって使用されます。別のトランザクションが既にこのマップ・エントリーのバージョンを変更している場合、バージョン比較は失敗し、バックアップ・マップは OptimisticCollisionException 例外を表示して、トランザクションを強制的にロールバックします。ローダーがプラグインされている場合、バックアップ・マップはオプティミスティック・バージョン管理情報を使用しません。代わりに、ローダーは、オプティミスティック・バージョン管理比較を行い、必要に応じてバージョン管理情報を更新する責任があります。ローダーは通常、ローダーの batchUpdate メソッドに渡される LogElement から、初期バージョン管理オブジェクトを取得します。このオブジェクトは、フラッシュ操作が発生するか、トランザクションがコミットされたときに呼び出されます。

次のコードは、EmployeeOptimisticCallbackImpl オブジェクトによって使用されるインプリメンテーションを示しています。

```
public Object getVersionedObjectForValue(Object value)
{
    if (value == null)
    {
        return null;
    }
    else
    {
        Employee emp = (Employee) value;
        return new Long( emp.getSequenceNumber() );
    }
}
```

前の例に示すように、sequenceNumber 属性は、ローダーが予期するように、java.lang.Long オブジェクト内に戻されます。これは、ローダーの作成者と同一人物が EmployeeOptimisticCallbackImpl を作成したか、 EmployeeOptimisticCallbackImpl を実装した人物と協力して作業を行ったか (例えば、getVersionedObjectForValue メソッドによって戻された値に合意した) のいずれかであることを示しています。デフォルトの OptimisticCallback プラグインは、特殊値 NULL_OPTIMISTIC_VERSION をバージョン・オブジェクトとして戻します。

updateVersionedObjectForValue メソッド

このメソッドは、トランザクションが値を更新し、新バージョンのオブジェクトが必要になるたびに呼び出されます。 getVersionedObjectForValue メソッドがこの値の属性を戻した場合、このメソッドは通常、属性値を新バージョンのオブジェクトに更新します。 getVersionedObjectForValue メソッドがこの値のコピーを戻した場合、このメソッドは通常、いかなるアクションも完了しません。デフォルトの OptimisticCallback プラグインは、getVersionedObjectForValue のデフォルト実装がバージョン・オブジェクトとして常に特殊値 NULL_OPTIMISTIC_VERSION を戻すため、このメソッドではいかなるアクションも完了しません。次の例は、OptimisticCallback セクションで使用される EmployeeOptimisticCallbackImpl オブジェクトによって使用される実装を示しています。

```
public void updateVersionedObjectForValue(Object value)
{
    if ( value != null )
    {
        Employee emp = (Employee) value;
        long next = emp.getSequenceNumber() + 1;
        emp.updateSequenceNumber( next );
    }
}
```

前の例で示すように、sequenceNumber 属性は、次に getVersionedObjectForValue メソッドが呼び出されたときに、戻される java.lang.Long 値が長整数値を持つように、1 ずつ増分されます。この長整数値は、元のシーケンス番号の値に 1 を加えたもの (例えば、この従業員インスタンスの次のバージョン値) です。この例は、ローダーを作成者が EmployeeOptimisticCallbackImpl の作成者と同一人物であるか、EmployeeOptimisticCallbackImpl を実装した人物と協力して作業を行ったかのいずれかであることを示しています。

serializeVersionedValue メソッド

このメソッドは、指定されたストリームにバージョン値を書き込みます。インプリメンテーションによっては、バージョン値を使用して、オプティミスティック更新の衝突を識別することができます。一部のインプリメンテーションでは、バージョン値は元の値のコピーです。それ以外のインプリメンテーションでは、値のバージョンを示すシーケンス番号またはその他のいくつかのオブジェクトがあります。実際の実装が不明であるため、このメソッドは適切なシリアライゼーションを実行するために提供されます。デフォルト実装は writeObject メソッドを呼び出します。

inflateVersionedValue メソッド

このメソッドは、バージョン値の直列化バージョンを取り、実際のバージョン値オブジェクトを戻します。インプリメンテーションによっては、バージョン値を使用して、オプティミスティック更新の衝突を識別することができます。一部のインプリメンテーションでは、バージョン値は元の値のコピーです。それ以外のインプリメンテーションでは、値のバージョンを示すシーケンス番号またはその他のいくつかのオブジェクトがあります。実際の実装が不明であるため、このメソッドは適切なデシリアライゼーションを行うために提供されます。デフォルト実装は readObject メソッドを呼び出します。

アプリケーション提供の OptimisticCallback オブジェクトの使用

アプリケーション提供の OptimisticCallback オブジェクトを BackingMap 構成に追加する場合、XML 構成とプログラマチック構成の 2 つの方法があります。

OptimisticCallback オブジェクトをプラグインするための XML 構成方法

次の例に示すように、アプリケーションは、XML ファイルを使用して、その OptimisticCallback オブジェクトをプラグインすることができます。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
    <objectGrid name="grid1">
        <backingMap name="employees" pluginCollectionRef="employees" lockStrategy="OPTIMISTIC" />
```

```
    </objectGrid>
</objectGrids>

<backingMapPluginCollections>
    <backingMapPluginCollection id="employees">
        <bean id="OptimisticCallback" className="com.xyz.EmployeeOptimisticCallbackImpl" />
    </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

OptimisticCallback オブジェクトのプログラマチックなプラグイン

次の例は、ローカル grid1 ObjectGrid インスタンス内の従業員のバックアップ・マップ用に、アプリケーションで OptimisticCallback オブジェクトをプログラマチックにプラグインする方法を示しています。

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid1" );
BackingMap bm = dg.defineMap("employees");
EmployeeOptimisticCallbackImpl cb = new EmployeeOptimisticCallbackImpl();
bm.setOptimisticCallback( cb );
```

ObjectTransformer プラグイン

ObjectTransformer プラグインを使用すると、キャッシュ内のオブジェクトをシリアライズ、デシリアライズ、およびコピーすることができます。ハイパフォーマンスを必要とするときは、 ObjectTransformer プラグインを使用します。

プロセッサーの使用に関するパフォーマンス上の問題がある場合は、各マップに ObjectTransformer プラグインを追加します。 ObjectTransformer プラグインを使用しない場合、合計プロセッサー時間の 60 から 70 パーセントまではエントリーのシリアライズとコピーに費やされます。

目的

ObjectTransformer プラグインがあれば、アプリケーションで以下の操作に対するカスタム・メソッドを提供できます。

- エントリーに対するキーのシリアライズまたはデシリアライズ
- エントリーに対する値のシリアライズまたはデシリアライズ
- エントリーに対するキーまたは値のコピー

ObjectTransformer プラグインが提供されない場合、 ObjectGrid はシリアライズおよびデシリアライズのシーケンスを使用してオブジェクトをコピーするので、ユーザーがキーと値のシリアライズを行う必要があります。この方法には費用がかかるので、パフォーマンスが重大である場合には ObjectTransformer プラグインを使用してください。アプリケーションが、トランザクションのオブジェクトを最初に検索する際に、コピーが行われます。このコピーは、マップのコピー・モードを NO_COPY に設定すると行われません。あるいは、コピー・モードを COPY_ON_READ に設定すると、コピー数を軽減できます。アプリケーションの必要に応じて、このプラグインにカスタム・コピー・メソッドを提供することによっ

て、コピー操作を最適化します。このようなプラグインにより、コピー・オーバーヘッドを合計プロセッサー時間の 65-70 パラメーターから 2/3 パーセントに軽減できます。

デフォルトの copyKey および copyValue メソッド実装では、最初に clone メソッド (このメソッドが提供されている場合) を使用しようとします。 clone メソッド実装が提供されていない場合は、実装のデフォルトはシリアライゼーションになります。

eXtreme Scale が分散モードで実行されているときは、オブジェクト・シリアライゼーションも直接使用されます。 LogSequence は、変更内容を ObjectGrid のピアに送信する前に、ObjectTransformer プラグインを使用して、キーおよび値のシリアライズを支援します。組み込み Java Developer Kit シリアライゼーションを使用するのではなく、シリアライゼーションのカスタム・メソッドを提供するときは、注意が必要です。オブジェクトのバージョン管理は複雑な問題であり、カスタム・メソッドがバージョン管理用に設計されていることが確認できない場合、バージョンの互換性に問題が発生することがあります。

以下のリストでは、eXtreme Scale がキーと値の両方のシリアライズを試みる方法を説明しています。

- カスタム ObjectTransformer プラグインが作成され、プラグインされている場合、eXtreme Scale は ObjectTransformer インターフェース内のメソッドを呼び出して、キーと値をシリアライズし、オブジェクトのキーおよび値のコピーを取得します。
- カスタム ObjectTransformer プラグインが使用されていない場合、eXtreme Scale はデフォルトに従って値のシリアライズとデシリアライズを行います。デフォルト・プラグインが使用されている場合、各オブジェクトは、外部化可能またはシリアライズ可能として実装されます。
  - オブジェクトが Externalizable インターフェースをサポートする場合、writeExternal メソッドが呼び出されます。外部化可能として実装されたオブジェクトは、パフォーマンスを向上させます。
  - Externalizablee インターフェースをサポートせず、Serializable インターフェースを実装しないオブジェクトは、ObjectOutputStream メソッドを使用して保管されます。

ObjectTransformer オブジェクトの記述

ObjectTransformer は、ObjectTransformer インターフェースを実装し、共通 ObjectGrid プラグイン規則に準拠している必要があります。

ObjectTransformer インターフェースの使用

ObjectTransformer オブジェクトを BackingMap 構成に追加する場合、プログラマチック構成と XML 構成の 2 つの方法が使用されます。両方とも以下のセクションで説明します。

ObjectTransformer をプラグインするための XML 構成方法

ObjectTransformer 実装のクラス名が、com.company.org.MyObjectTransformer クラスであると仮定します。このクラスは、ObjectTransformer インターフェースを実装します。ObjectTransformer 実装は、以下の XML を使用して構成することができます。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="myGrid">
     <backingMap name="myMap" pluginCollectionRef="myMap" />
        </objectGrid>
    </objectGrids>

    <backingMapPluginCollections>
 <backingMapPluginCollection id="myMap">
      <bean id="ObjectTransformer" className="com.company.org.MyObjectTransformer" />
 </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

ObjectTransformer オブジェクトのプログラマチックなプラグイン

以下のコード・スニペットは、カスタム ObjectTransformer オブジェクトを作成し、それを BackingMap に追加します。

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
BackingMap backingMap = myGrid.getMap("myMap");
MyObjectTransformer myObjectTransformer = new MyObjectTransformer();
backingMap.setObjectTransformer(myObjectTransformer);
```

ObjectTransformer の使用に関するシナリオ

ObjectTransformer プラグインは、以下の状態で使用できます。

- シリアライズ不能オブジェクト
- シリアライズ可能オブジェクトであるが、シリアライゼーション・パフォーマンスを改善する
- キーまたは値のコピー

以下の例で、ObjectGrid は Stock クラスのストアに使用されます。

```
/**
* Stock object for ObjectGrid demo
*
*
*/
public class Stock implements Cloneable {
    String ticket;
    double price;
    String company;
    String description;
    int serialNumber;
    long lastTransactionTime;
    /**
    * @return Returns the description.
    */
    public String getDescription() {
        return description;
    }
    /**
    * @param description The description to set.
    */
    public void setDescription(String description) {
        this.description = description;
    }
```

```
    /**
    * @return Returns the lastTransactionTime.
    */
    public long getLastTransactionTime() {
        return lastTransactionTime;
    }
    /**
    * @param lastTransactionTime The lastTransactionTime to set.
    */
    public void setLastTransactionTime(long lastTransactionTime) {
        this.lastTransactionTime = lastTransactionTime;
    }
    /**
    * @return Returns the price.
    */
    public double getPrice() {
        return price;
    }
    /**
    * @param price The price to set.
    */
    public void setPrice(double price) {
        this.price = price;
    }
    /**
    * @return Returns the serialNumber.
    */
    public int getSerialNumber() {
        return serialNumber;
    }
    /**
    * @param serialNumber The serialNumber to set.
    */
    public void setSerialNumber(int serialNumber) {
        this.serialNumber = serialNumber;
    }
    /**
    * @return Returns the ticket.
    */
    public String getTicket() {
        return ticket;
    }
    /**
    * @param ticket The ticket to set.
    */
    public void setTicket(String ticket) {
        this.ticket = ticket;
    }
    /**
    * @return Returns the company.
    */
    public String getCompany() {
        return company;
    }
    /**
    * @param company The company to set.
    */
    public void setCompany(String company) {
        this.company = company;
    }
    //clone
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

Stock クラス用に、カスタム・オブジェクト変換プログラム・クラスを作成できます。

```
/**
* Custom implementation of ObjectGrid ObjectTransformer for stock object
*
*/
public class MyStockObjectTransformer implements ObjectTransformer {
/* (non-Javadoc)
* @see
* com.ibm.websphere.objectgrid.plugins.ObjectTransformer#serializeKey
* (java.lang.Object,
* java.io.ObjectOutputStream)
```

```
*/
public void serializeKey(Object key, ObjectOutputStream stream) throws IOException {
    String ticket= (String) key;
    stream.writeUTF(ticket);
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#serializeValue(java.lang.Object,
java.io.ObjectOutputStream)
*/
public void serializeValue(Object value, ObjectOutputStream stream) throws IOException {
    Stock stock= (Stock) value;
    stream.writeUTF(stock.getTicket());
    stream.writeUTF(stock.getCompany());
    stream.writeUTF(stock.getDescription());
    stream.writeDouble(stock.getPrice());
    stream.writeLong(stock.getLastTransactionTime());
    stream.writeInt(stock.getSerialNumber());
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#inflateKey(java.io.ObjectInputStream)
*/
public Object inflateKey(ObjectInputStream stream) throws IOException, ClassNotFoundException {
    String ticket=stream.readUTF();
    return ticket;
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#inflateValue(java.io.ObjectInputStream)
*/

public Object inflateValue(ObjectInputStream stream) throws IOException, ClassNotFoundException {
    Stock stock=new Stock();
    stock.setTicket(stream.readUTF());
    stock.setCompany(stream.readUTF());
    stock.setDescription(stream.readUTF());
    stock.setPrice(stream.readDouble());
    stock.setLastTransactionTime(stream.readLong());
    stock.setSerialNumber(stream.readInt());
    return stock;
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#copyValue(java.lang.Object)
*/
public Object copyValue(Object value) {
    Stock stock = (Stock) value;
    try {
        return stock.clone();
    }
    catch (CloneNotSupportedException e)
    {
        // display exception message    }
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#copyKey(java.lang.Object)
*/
public Object copyKey(Object key) {
    String ticket=(String) key;
    String ticketCopy= new String (ticket);
    return ticketCopy;
}
}
```

次に、このカスタム MyStockObjectTransformer クラスを BackingMap にプラグインします。

```
ObjectGridManager ogf=ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogf.getObjectGrid("NYSE");
BackingMap bm = og.defineMap("NYSEStocks");
MyStockObjectTransformer ot = new MyStockObjectTransformer();
bm.setObjectTransformer(ot);
```

WebSphereTransactionCallback プラグイン

WebSphereTransactionCallback プラグインを使用すると、WebSphere Application Server 環境で実行しているエンタープライズ・アプリケーションは ObjectGrid トランザクションを管理できます。

コンテナー管理トランザクションを使用するよう構成されているメソッド内で ObjectGrid セッションを使用している場合、エンタープライズ・コンテナーが ObjectGrid トランザクションを自動的に、開始、コミットまたはロールバックします。 Java Transaction API (JTA) UserTransaction オブジェクトを使用していると、ObjectGrid トランザクションは UserTransaction オブジェクトによって自動的に管理されます。

このプラグインの実装について詳しくは、158 ページの『外部トランザクション・マネージャー』を参照してください。

注: ObjectGrid では、2 フェーズの XA トランザクションはサポートしていません。このプラグインは、ObjectGrid トランザクションをトランザクション・マネージャーに登録しません。したがって、ObjectGrid がコミットに失敗した場合、XA トランザクションによって管理される他のリソースはロールバックしません。

WebSphereTransactionCallback プラグインの使用可能化

プログラマチック構成または XML 構成によって ObjectGrid 構成への WebSphereTransactionCallback を使用可能にすることができます。

WebSphereTransactionCallback オブジェクトをプラグインするための XML 構成方法

以下の XML 構成は、WebSphereTransactionCallback オブジェクトを作成して、ObjectGrid に追加するものです。以下のテキストは、`myGrid.xml` ファイルに存在しなければなりません。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="myGrid">
            <bean id="TransactionCallback" className=
 "com.ibm.websphere.objectgrid.plugins.builtins.WebSphereTransactionCallback" />

        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

WebSphereTransactionCallback オブジェクトのプログラマチックなプラグイン

以下のコード・スニペットは、WebSphereTransactionCallback オブジェクトを作成し、それを ObjectGrid に追加します。

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
WebSphereTransactionCallback wsTxCallback= new WebSphereTransactionCallback ();
myGrid.setTransactionCallback(wsTxCallback);
```

第 5 章 Spring フレームワークとの統合

Spring は、Java アプリケーションの開発によく使用されるフレームワークです。WebSphere eXtreme Scale では、 Spring を使用して eXtreme Scale トランザクションを管理し、デプロイされたメモリー内データ・グリッドに含まれるクライアントおよびサーバーの構成を行うことがサポートされています。

Spring 管理ネイティブ・トランザクション

Spring は、 Java Platform, Enterprise Edition アプリケーション・サーバーに似たコンテナー管理トランザクションを提供します。しかし、Spring メカニズムはさまざまな実装環境でプラグ可能です。WebSphere eXtreme Scale が提供するトランザクション・マネージャー統合は、Spring が ObjectGrid トランザクションのライフサイクルを管理することを可能にします。詳しくは、「*プログラミング・ガイド*」内のネイティブ・トランザクションに関する説明を参照してください。

Spring 管理拡張 Bean および名前空間のサポート

また、eXtreme Scale が Spring と統合されることによって、拡張ポイントまたはプラグイン用に Spring スタイルの Bean を定義することが可能になります。この機能によって、拡張ポイントの構成の柔軟性が高まり、洗練された構成ができるようになります。

Spring 管理の拡張 Bean に加えて、 eXtreme Scale は、「objectgrid」という名前の Spring 名前空間を提供します。Bean および組み込みの実装がこの名前空間に事前定義されていて、ユーザーが eXtreme Scale をより簡単に構成できるようになっています。これらのトピックに関する詳しい説明と、Spring 構成を使用して eXtreme Scale コンテナー・サーバーを開始する方法の例については、 214 ページの『Spring 拡張 Bean および名前空間のサポート』を参照してください。

断片有効範囲サポート

従来のスタイルの Spring 構成では、 ObjectGrid Bean は singleton タイプかプロトタイプ・タイプのどちらかです。ObjectGrid は、「断片」有効範囲と呼ばれる新しい有効範囲もサポートします。Bean が断片有効範囲と定義されている場合、断片当たり 1 つの Bean のみが作成されます。同じ断片内でその Bean 定義に一致する ID を持つ Bean に対する要求はすべて、その 1 つの特定の Bean インスタンスが Spring コンテナーによって戻される結果になります。

以下の例に示す com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl Bean の定義では、有効範囲が断片であると設定されています。したがって、断片当たり、JPAPropFactoryImpl クラスの 1 つのインスタンスのみが作成されます。

```
<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl" scope="shard" />
```

Spring Web Flow

Spring Web Flow は、デフォルトではセッション状態を HTTP セッションに保管します。Web アプリケーションがセッション管理に eXtreme Scale を使用するよう構

成されている場合、この状態を保管するために Spring によってそれが自動的に使用され、セッションと同じ方法でフォールト・トレラントにされます。

パッケージ化

eXtreme Scale Spring 拡張は ogspring.jar ファイルに入っています。Spring サポートが正しく機能するためには、この Java アーカイブ (JAR) ファイルがクラスパスになければなりません。 WebSphere Extended Deployment で実行している JEE アプリケーションが WebSphere Application Server Network Deployment を拡張した場合、そのアプリケーションは spring.jar ファイルおよびその関連ファイルをエンタープライズ・アーカイブ (EAR) モジュールに入れる必要があります。同じ場所に ogspring.jar ファイルも入れる必要があります。

ネイティブ・トランザクション

Spring は、Java アプリケーションの開発によく使用されるフレームワークです。WebSphere eXtreme Scale では、Spring を使用して eXtreme Scale トランザクションを管理したり、eXtreme Scale クライアントおよびサーバーの構成を行うことがサポートされています。

WebSphere eXtreme Scale を使用したネイティブ・トランザクション

Spring は、 Java Platform, Enterprise Edition アプリケーション・サーバーのスタイルに沿ったコンテナー管理トランザクションを提供しますが、Spring のメカニズムによりさまざまな実装を組み込むことができるという利点があります。このトピックでは、Spring とともに使用できる eXtreme Scale プラットフォーム・トランザクション・マネージャーについて説明します。これを使用すると、プログラマーは、POJO (Plain Old Java Object) にアノテーションを付けてから、 Spring に eXtreme Scale からの Session を自動獲得させて、 eXtreme Scale トランザクションを開始、コミット、ロールバック、中断、および再開させることができます。このトピックでは、Spring トランザクション・メカニズムについては説明しません。 Spring トランザクションについては、Spring User Guide の第 9 章に説明されています。このトピックでは、 eXtreme Scale トランザクション・マネージャーを作成して、それをアノテーション付きの POJO で使用する方法を説明します。また、この方法をクライアントまたはローカル eXtreme Scale および連結された Data Grid スタイル・アプリケーションとともに使用する方法についても説明します。

トランザクション・マネージャーの作成

eXtreme Scale は Spring PlatformTransactionManager の実装を提供します。このマネージャーは、管理対象の eXtreme Scale セッションを Spring が管理する POJO に提供することができます。 Spring は、アノテーションの使用により、トランザクション・ライフサイクルの単位で POJO のセッションを管理します。次の XML スニペットは、トランザクション・マネージャーの作成方法を示しています。

```
<aop:aspectj-autoproxy/>
 <tx:annotation-driven transaction-manager="transactionManager"/>

 <bean id="ObjectGridManager"
    class="com.ibm.websphere.objectgrid.ObjectGridManagerFactory"
    factory-method="getObjectGridManager"/>

 <bean id="ObjectGrid"
    factory-bean="ObjectGridManager"
```

```
    factory-method="createObjectGrid"/>

 <bean id="transactionManager"
    class="com.ibm.websphere.objectgrid.spring.ObjectGridSpringFactory"
    factory-method="getLocalPlatformTransactionManager"/>
 </bean>

 <bean id="Service" class="com.ibm.websphere.objectgrid.spring.test.TestService">
  <property name="txManager" ref+"transactionManager"/>
 </bean>
```

これは、transactionManager Bean が宣言され、Spring トランザクションを使用する Service Bean に接続されるところを示しています。これはアノテーションを使用して示されますが、これが先頭に tx:annotation 文節のある理由です。

現行 Spring トランザクションのための ObjectGrid セッションの取得

Spring が管理するメソッドを持つ POJO は現在、次のメソッドを使用して現行トランザクションのための ObjectGrid セッションを取得することができます。

```
Session s = txManager.getSession();
```

これは、POJO が使用するセッションを返します。同じトランザクションに関係する Bean は、このメソッドを呼び出したとき、同じセッションを受け取ります。Spring はセッションに対して begin を自動的に処理し、また必要なときに commit または rollback を自動的に呼び出します。また、セッション・オブジェクトから getEntityManager を呼び出すだけでも ObjectGrid EntityManager を取得することができます。

アノテーションを使用するサンプル POJO

これは、アノテーションを使用して、Spring に対してトランザクションの意図を宣言する POJO です。すべてのメソッドはデフォルトで REQUIRED トランザクション・セマンティクスを使用することを指示するクラス・レベル・アノテーションがクラスにあることが分かります。クラスは、クラス上のすべてのメソッドに対して、メソッドとのインターフェースを実装します。これは、バイトコードを作成できない場合の Spring AOP が機能するために必要です。クラスには、Spring xml ファイルを使用して ObjectGrid トランザクション・マネージャーに接続するインスタンス変数 txManager があります。各メソッドは、txManager.getSession メソッドを呼び出すだけで、そのメソッドのために使用するセッションを取得します。queryNewTx メソッドには REQUIRES_NEW セマンティックを示すアノテーションが付けられています。このため、既存のトランザクションは中断され、そのメソッドに対して新しい独立トランザクションが作成されます。

```
@Transactional(propagation=Propagation.REQUIRED)
public class TestService implements ITestService
{
    SpringLocalTxManager txManager;

    public TestService()
    {

    }

    public void initialize()
        throws ObjectGridException
    {
        Session s = txManager.getSession();
        ObjectMap m = s.getMap("TEST");
        m.insert("Hello", "Billy");
    }

    public void update(String updatedValue)
        throws ObjectGridException
    {
```

```
        Session s = txManager.getSession();
        System.out.println("Update using " + s);
        ObjectMap m = s.getMap("TEST");
        String v = (String)m.get("Hello");
        m.update("Hello", updatedValue);
    }

    public String query()
        throws ObjectGridException
    {
        Session s = txManager.getSession();
        System.out.println("Query using " + s);
        ObjectMap m = s.getMap("TEST");
        return (String)m.get("Hello");
    }

    @Transactional(propagation=Propagation.REQUIRES_NEW)
    public String queryNewTx()
        throws ObjectGridException
    {
        Session s = txManager.getSession();
        System.out.println("QueryTX using " + s);
        ObjectMap m = s.getMap("TEST");
        return (String)m.get("Hello");
    }

    public void testRequiresNew(ITestService bean)
        throws ObjectGridException
    {
        update("1");
        String txValue = bean.query();
        if(!txValue.equals("1"))
        {
            System.out.println("Requires didnt work");
            throw new IllegalStateException("requires didn't work");
        }
        String committedValue = bean.queryNewTx();
        if(committedValue.equals("1"))
        {
            System.out.println("Requires new didnt work");
            throw new IllegalStateException("requires new didn't work");
        }
    }

    public SpringLocalTxManager getTxManager() {
        return txManager;
    }

    public void setTxManager(SpringLocalTxManager txManager) {
        this.txManager = txManager;
    }
}
```

スレッドの ObjectGrid インスタンスの設定

単一の Java 仮想マシン (JVM) で多数の ObjectGrid インスタンスをホストすることができます。JVM に置かれた各プライマリー断片には独自の ObjectGrid インスタンスがあります。リモート ObjectGrid に対してクライアントとして機能する JVM は、connect メソッドの ClientClusterContext から戻される ObjectGrid インスタンスを使用して、その Grid と対話します。 ObjectGrid の Spring トランザクションを使用して POJO でメソッドを呼び出す前に、使用する ObjectGrid インスタンスでスレッドを事前準備する必要があります。 TransactionManager インスタンスには、特定の ObjectGrid インスタンスの指定を可能にするメソッドがあります。これが指定されると、後続の txManager.getSession 呼び出しはその ObjectGrid インスタンスのセッションを返します。

テストのための簡単なブートストラップ

次の例は、この機能を実行するためのサンプル・メインを示しています。

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(new String[]
            {"applicationContext.xml"});
    SpringLocalTxManager txManager = (SpringLocalTxManager)ctx.getBean("transactionManager");
    txManager.setObjectGridForThread(og);

    ITestService s = (ITestService)ctx.getBean("Service");
    s.initialize();
    assertEquals(s.query(), "Billy");
```

```
s.update("Bobby");
assertEquals(s.query(), "Bobby");
System.out.println("Requires new test");
s.testRequiresNew(s);
assertEquals(s.query(), "1");
```

ここでは Spring ApplicationContext を使用します。 ApplicationContext は、txManager への参照を取得して、このスレッドで使用する ObjectGrid を指定するために使用されます。次にコードは、サービスへの参照を取得して、そのサービス上でメソッドを呼び出します。このレベルの各メソッドにより、Spring はセッションを作成し、メソッド呼び出しの周辺で begin/commit 呼び出しを行います。例外が発生するとロールバックが行われます。

新規 eXtreme Scale インターフェース

ここでは、SpringLocalTxManager という新しいインターフェースを 1 つだけ紹介します。このインターフェースは ObjectGrid プラットフォーム・トランザクション・マネージャーによって実装されるもので、パブリック・インターフェースをすべて持っています。このインターフェース上のメソッドは、スレッドで使用する ObjectGrid インスタンスを選択し、そのスレッドのセッションを取得するためのものです。 ObjectGrid ローカル・トランザクションを使用する POJO には、このマネージャー・インスタンスへの参照を入れる必要があります。また、単一インスタンスのみを作成する必要があります (つまり、そのスコープは singleton でなければなりません)。このインスタンスは、ObjectGridSpringFactory 上の静的メソッドを使用して作成されます。 getLocalPlatformTransactionManager()。

JTA およびグローバル・トランザクションのための eXtreme Scale

eXtreme Scale は、主としてスケーラビリティーと関係があるさまざまな理由から、JTA および 2 フェーズ・コミットをサポートしません。したがって、最後の単一フェーズ参加者の場合を除き、ObjectGrid は XA または JTA タイプのグローバル・トランザクションでは対話しません。このプラットフォーム・マネージャーは、ローカル ObjectGrid トランザクションの使用を Spring 開発者のためにできるだけ容易にするように意図されています。

Spring が管理する拡張 Bean

ObjectGrid には、objectgrid.xml ファイル内で拡張ポイントとして使用する POJO を宣言する方法が含まれています。 ObjectGrid はこの Bean に名前を付け、クラス名を指定する方法を提供します。 ObjectGrid は通常、指定されたクラスのインスタンスを作成し、そのインスタンスをプラグインとして使用します。 ObjectGrid は現在、このプラグイン・オブジェクトのインスタンスを取得するための Bean ファクトリーとして機能するように Spring に委任することができます。アプリケーションが Spring を使用する場合は、通常、このような POJO をアプリケーションの残り部分に接続する必要があります。

ObjectGrid は、特定の名前付き ObjectGrid のために使用する Spring Bean ファクトリー・インスタンスをアプリケーションが登録できるように変更されました。アプリケーションは、BeanFactory のインスタンスまたは Spring アプリケーション・コンテキストを作成してから、次の静的メソッドを使用してそれを ObjectGrid に登録する必要があります。

```
void registerSpringBeanFactoryAdapter(String objectGridName, Object springBeanFactory)
```

このメソッドは、className が接頭部 {spring} で始まる拡張 Bean (ObjectTransformer、Loader、TransactionCallback など) を ObjectGrid が検出した場合に、名前の残り部分を Spring Bean 名として使用し、Spring Bean ファクトリーを使用して Bean インスタンスを取得することを指定します。 ObjectGrid は、デフォルトの Spring xml 構成ファイルから Spring Bean ファクトリーを作成することもできます。与えられた ObjectGrid の Bean ファクトリーが登録されていなかった場合は、ObjectGrid が 'ObjectGridName'_spring.xml という xml ファイルの検出を試みます。例えば、グリッドの名前が GRID の場合は、xml ファイルの名前は '/GRID_spring.xml' で、このファイルはルート・パッケージのクラスパスにあるはずです。このファイルが検出されると、ObjectGrid はそのファイルを使用して ApplicationContext を作成し、その Bean ファクトリーから Bean を作成します。例えば、クラス名は次のようになります。

```
"{spring}MyLoaderBean"
```

これにより、ObjectGrid は Spring に ″MyLoaderBean″ という名前の Bean を要求します。この方法を使用すれば、Bean ファクトリーが事前に登録されている限り、ObjectGrid 内の拡張ポイントに Spring 管理 POJO を指定することができます。ObjectGrid Spring 拡張は ogspring.jar ファイルに入っています。 Spring サポートが正しく機能するためには、この JAR ファイルがクラスパスになければなりません。XD で実行中の JavaEE アプリケーションが ND を拡張した場合、そのアプリケーションは spring.jar ファイルとその関連ファイルを EAR モジュールに入れる必要があります。 ogspring.jar も同じロケーションに置く必要があります。

Spring Webflow

Spring Webflow は、デフォルトで、そのセッション状態を HTTP セッションに保管します。 Web アプリケーションがセッション管理に ObjectGrid を使用するように構成されていると、Spring がこの状態を保管するために自動的に ObjectGrid を使用し、ObjectGrid はセッションと同じ方法でフォールト・トレラントになります。

Spring 拡張 Bean および名前空間のサポート

WebSphere eXtreme Scale には、 objectgrid.xml ファイル内で拡張ポイントとして使用するために Plain Old Java Object (POJO) を宣言する機能があり、Bean を指定してからクラス名を指定する方法が提供されています。通常、指定されたクラスのインスタンスが作成され、それらのオブジェクトはプラグインとして使用されます。eXtreme Scale は、これらのプラグイン・オブジェクトのインスタンスの取得を Spring に委任できます。アプリケーションが Spring を使用する場合は、通常、このような POJO をアプリケーションの残り部分に接続する必要があります。

場合によっては、特定のプラグイン・オブジェクトを構成するのに Spring を使用する必要があります。例として以下の構成を参考にしてください。

```
<objectGrid name="Grid">
    <bean id="TransactionCallback" className="com.ibm.websphere.objectgrid.jpa.JPATxCallback">
        <property name="persistenceUnitName" type="java.lang.String"  value="employeePU" />
    </bean>
    ...
</objectGrid>
```

組み込み TransactionCallback 実装である com.ibm.websphere.objectgrid.jpa.JPATxCallback クラスは、 TransactionCallback クラスとして構成されます。このクラスは上の例のように、1 つのプロパティー

persistenceUnitName を使用して構成されます。 JPATxCallback クラスには JPAPropertyFactory 属性もあり、これのタイプは java.lang.Object です。ObjectGrid XML 構成は、このタイプの構成をサポートできません。

eXtreme Scale Spring 統合は Bean 作成を Spring フレームワークに委任することでこの問題を解決します。修正後の構成は、次のようになります。

```
<objectGrid name="Grid">
    <bean id="TransactionCallback" className="{spring}jpaTxCallback"/>
    ...
</objectGrid>
```

″Grid″ オブジェクト用の Spring ファイルには以下の情報が入っています。

```
<bean id="jpaTxCallback" class="com.ibm.websphere.objectgrid.jpa.JPATxCallback" scope="shard">
    <property name="persistenceUnitName" value="employeeEMPU"/>
    <property name="JPAPropertyFactory" ref ="jpaPropFactory"/>
</bean>

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.
JPAPropFactoryImpl" scope="shard">
</bean>
```

ここでは、上の例に示されているように、{spring}jpaTxCallback として TransactionCallback が指定され、 Spring ファイル内に jpaTxCallback および jpaPropFactory Bean が構成されています。このような Spring 構成によって、JPAPropertyFactory Bean を JPATxCallback オブジェクトのパラメーターとして構成することが可能になります。

デフォルトの Spring Bean ファクトリー

eXtreme Scale が、接頭部 {spring} で始まる classname 値を持つプラグインまたは拡張 Bean (ObjectTransformer、Loader、TransactionCallback など) を検出した場合、eXtreme Scale は名前の残りの部分を Spring Bean 名として使用し、Spring Bean ファクトリーを使用して Bean インスタンスを取得します。

デフォルトでは、与えられた ObjectGrid 用に登録された Bean ファクトリーがない場合、ObjectGridName_spring.xml ファイルを見つけようとします。例えば、グリッドの名前が ″Grid″ の場合は、XML ファイルの名前は /Grid_spring.xml です。このファイルはクラスパスにあるか、クラスパス内の META-INF ディレクトリーにあるはずです。このファイルが見つかったら、eXtreme Scale は、そのファイルを使用して ApplicationContext を作成し、その Bean ファクトリーから Bean を作成します。

カスタム Spring Bean ファクトリー

WebSphere eXtreme Scale には ObjectGridSpringFactory API もあり、これを使用して、特定の指定された ObjectGrid のために使用するよう Spring Bean ファクトリー・インスタンスを登録できます。この API は、以下の静的メソッドを使用して、BeanFactory のインスタンスを eXtreme Scale に登録します。

```
void registerSpringBeanFactoryAdapter(String objectGridName, Object
springBeanFactory)
```

名前空間サポート

バージョン 2.0 以降の Spring には、Bean の定義と構成のため、基本的な Spring XML フォーマットをスキーマ・ベースで拡張するメカニズムが備わっています。ObjectGrid はこの新しい機能を使用して、ObjectGrid Bean の定義と構成を行います。Spring XML スキーマ拡張では、eXtreme Scale プラグインのいくつかの組み込み実装、およびいくつかの ObjectGrid Bean が ″objectgrid″ 名前空間に事前定義されます。 Spring 構成ファイルを作成するとき、これらの組み込みの完全クラス名を指定する必要はありません。代わりに、事前定義された Bean を参照するようにできます。

また、XML スキーマ内に Bean の属性が定義されていることによって、間違った属性名を指定する可能性が減少します。XML スキーマに基づいた XML 妥当性検査は、この種のエラーを開発サイクルの初期にキャッチできます。

XML スキーマ拡張に定義されている Bean は、以下のとおりです。

- transactionManager
- register
- server
- カタログ (catalog)
- コンテナー
- JPALoader
- JPATxCallback
- JPAEntityLoader
- LRUEvictor
- LFUEvictor
- HashIndex

これらの Bean は objectgrid.xsd XML スキーマ内に定義されています。この XSD ファイルは、`ogspring.jar` ファイル中の `com/ibm/ws/objectgrid/spring/namespace/objectgrid.xsd` ファイルとして出荷されます。XSD ファイルおよび XSD ファイルで定義された Bean について詳しくは、「*管理ガイド*」に記載されている Spring 記述子ファイルに関する説明を参照してください。

前のセクションにある JPATxCallback 例をまた使用します。前のセクションでは、JPATxCallback Bean は次のように構成されていました。

```
<bean id="jpaTxCallback" class="com.ibm.websphere.objectgrid.jpa.JPATxCallback" scope="shard">
    <property name="persistenceUnitName" value="employeeEMPU"/>
    <property name="JPAPropertyFactory" ref ="jpaPropFactory"/>
</bean>

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl" scope="shard">
</bean>
```

この名前空間フィーチャーを使用して、Spring XML 構成を次のようにコーディングできます。

```
<objectgrid:JPATxCallback id="jpaTxCallback" persistenceUnitName="employeeEMPU"
 jpaPropertyFactory="jpaPropFactory" />

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl"
 scope="shard">
</bean>
```

ここでは、前の例でのように ″com.ibm.websphere.objectgrid.jpa.JPATxCallback″ クラスを指定する代わりに、事前定義された ″objectgrid:JPATxCallback″ Bean を直接使用することに注意してください。見て分かるように、この構成のほうが冗長でなく、誤りがないかチェックするのも簡単です。

Spring 拡張 Bean を使用したコンテナー・サーバーの開始

この例では、ObjectGrid Spring 管理拡張 Bean および名前空間のサポートを使用して、ObjectGrid サーバーを開始する方法を示します。

ObjectGrid XML ファイル

まず最初に、1 つの ObjectGrid ″Grid″ と 1 つのマップ ″Test″ が含まれているだけの、単純な ObjectGrid XML ファイルを定義します。この ObjectGrid には ″partitionListener″ という名前の ObjectGridEventListener プラグインがあり、マップ ″Test″ には ″testLRUEvictor″ という名前の Evictor プラグインがあります。ObjectGridEventListener プラグインと Evictor プラグインの両方とも、名前に ″{spring}″ が含まれるため、Spring を使用して構成されることに注意してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="Grid">
     <bean id="ObjectGridEventListener" className="{spring}partitionListener" />
            <backingMap name="Test" pluginCollectionRef="test" />
        </objectGrid>
    </objectGrids>

    <backingMapPluginCollections>
        <backingMapPluginCollection id="test">
            <bean id="Evictor" className="{spring}testLRUEvictor"/>
        </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

ObjectGrid デプロイメント XML ファイル

次に、以下に示すように単純な ObjectGrid デプロイメント XML ファイルを作成します。これは ObjectGrid を 5 個の区画に分けます。複製は必要ありません。

```
<?xml version="1.0" encoding="UTF-8"?>
<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy ../deploymentPolicy.xsd"
 xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">
    <objectgridDeployment objectgridName="Grid">
        <mapSet name="mapSet" numInitialContainers="1" numberOfPartitions="5" minSyncReplicas="0"
            maxSyncReplicas="1" maxAsyncReplicas="0">
            <map ref="Test"/>
        </mapSet>
    </objectgridDeployment>
</deploymentPolicy>
```

ObjectGrid Spring XML ファイル

次に、 ObjectGrid Spring 管理拡張 Bean および名前空間のサポート機能を両方とも使用して、ObjectGrid Bean を構成します。spring xml ファイルの名前は ″Grid_spring.xml″ です。この XML ファイルには 2 つのスキーマが含まれていることに注意してください。spring-beans-2.0.xsd は Spring 管理 Bean を使用するためのもので、objectgrid.xsd は objectgrid 名前空間内に事前定義された Bean を使用するためのものです。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:objectgrid="http://www.ibm.com/schema/objectgrid"
       xsi:schemaLocation="
       http://www.ibm.com/schema/objectgrid
   http://www.ibm.com/schema/objectgrid/objectgrid.xsd
       http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <objectgrid:register id="ogregister" gridname="Grid"/>

    <objectgrid:server id="server" isCatalog="true" name="server">
        <objectgrid:catalog host="localhost" port="2809"/>
    </objectgrid:server>

    <objectgrid:container id="container"
   objectgridxml="com/ibm/ws/objectgrid/test/springshard/objectgrid.xml"
      deploymentxml="com/ibm/ws/objectgrid/test/springshard/deployment.xml"
   server="server"/>

    <objectgrid:LRUEvictor id="testLRUEvictor" numberOfLRUQueues="31"/>

    <bean id="partitionListener"
   class="com.ibm.websphere.objectgrid.springshard.ShardListener" scope="shard"/>
</beans>
```

この spring XML ファイルには、次の 6 個の Bean が定義されました。

1. *objectgrid:register*: これは、ObjectGrid ″Grid″ に対してデフォルトの Bean ファクトリーを登録します。
2. *objectgrid:server*: これは、″server″ という名前で ObjectGrid サーバーを定義します。objectgrid:catalog Bean がネストされているので、このサーバーはカタログ・サービスも提供します。
3. *objectgrid:catalog*: これは、″localhost:2809″ に設定された ObjectGrid カタログ・サービス・エンドポイントを定義します。
4. *objectgrid:container*: これは、前述したように、指定された objectgrid XML ファイルおよびデプロイメント XML ファイルと共に ObjectGrid コンテナーを定義します。server プロパティーは、このコンテナーがどのサーバーでホストされているのかを指定します。
5. *objectgrid:LRUEvictor*: これは、使用する LRU キューの数を 31 に設定して LRUEvictor を定義します。
6. *bean partitionListener*: これは ShardListener プラグインを定義します。このクラスは、ユーザーによってプラグインされるクラスであるため、事前定義された Bean を使用することはできません。また、この Bean の有効範囲は ″shard″ (断片) に設定されています。これは、この ShardListener のインスタンスが ObejctGrid 断片当たり 1 つのみであることを意味します。

サーバーの始動

以下のスニペットは、コンテナー・サービスとカタログ・サービスの両方をホストする ObjectGrid サーバーを開始します。これを見て分かるように、サーバーを開始するために呼び出す必要のあるメソッドは、Bean ファクトリーからの Bean ″container″ の get だけです。これは、ロジックの大部分を Spring 構成に移すことになり、プログラミングの複雑さが軽減されます。

```
public class ShardServer extends TestCase
{
    Container container;
    org.springframework.beans.factory.BeanFactory bf;

    public void startServer(String cep)
    {
        try
        {
```

```
            bf = new org.springframework.context.support.ClassPathXmlApplicationContext(
                "/com/ibm/ws/objectgrid/test/springshard/Grid_spring.xml", ShardServer.class);
            container = (Container)bf.getBean("container");
        }
        catch(Exception e)
        {
            throw new ObjectGridRuntimeException("Cannot start OG container", e);
        }
    }

    public void stopServer()
    {
        if(container != null)
            container.teardown();
    }
}
```

第 6 章 セキュリティー API

WebSphere eXtreme Scale は、オープン・セキュリティー・アーキテクチャーを採用しています。認証、許可、およびトランスポート・セキュリティーの基本的なセキュリティー・フレームワークを提供し、さらにセキュリティー・インフラストラクチャーを完全なものにするためにユーザーにプラグインの実装を求めています。

次の図は、eXtreme Scale サーバーにおけるクライアントの認証および許可の基本的フローを示しています。

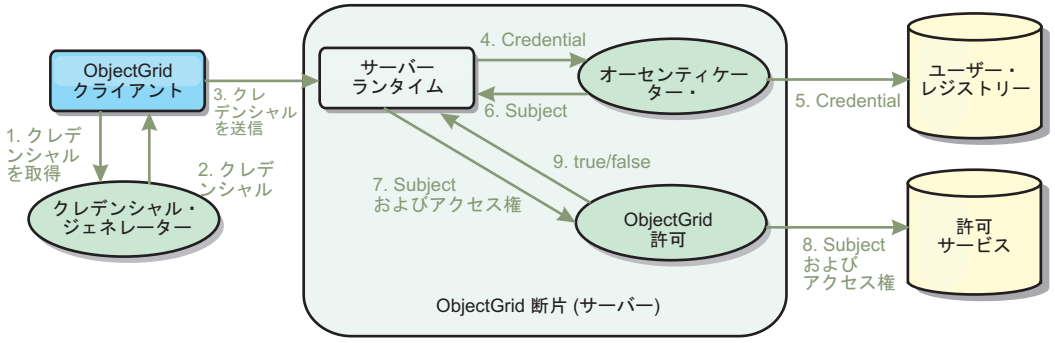

図 13. クライアントの認証および許可のフロー

認証フローと許可フローは、以下のようになります。

認証フロー

1. 認証フローは、eXtreme Scale クライアントのクレデンシャル取得で始まります。これは、com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator プラグインにより実行されます。
2. CredentialGenerator オブジェクトは、有効なクライアント・クレデンシャル (例えば、ユーザー ID とパスワードのペア、Kerberos チケットなど) の生成方法を認識しています。生成されたこのクレデンシャルは、クライアントに送り戻されます。
3. クライアントが CredentialGenerator オブジェクトを使用して Credential オブジェクトを取得すると、この Credential オブジェクトは、eXtreme Scale サーバーに eXtreme Scale 要求と共に送信されます。
4. eXtreme Scale サーバーは、eXtreme Scale 要求を処理する前に、 Credential オブジェクトの認証を行います。その後、サーバーは Authenticator プラグインを使用して Credential オブジェクトを認証します。
5. Authenticator プラグインは、ユーザー・レジストリーへのインターフェース (例えば、Lightweight Directory Access Protocol (LDAP) サーバーまたはオペレーティング・システムのユーザー・レジストリーなど) になります。Authenticator は、ユーザー・レジストリーを参考にして、認証の決定をします。
6. 正常に認証されると、このクライアントを表す Subject オブジェクトが戻されます。

許可フロー

WebSphere eXtreme Scale は、アクセス権ベースの許可メカニズムを採用し、各種の許可クラスによって表されるさまざまな許可カテゴリーがあります。例えば、com.ibm.websphere.objectgrid.security.MapPermission オブジェクトは、ObjectMap のデータ・エントリーの読み取り、書き込み、挿入、無効化、および除去の許可を表します。 WebSphere eXtreme Scale は、Java 認証および承認サービス (JAAS) 許可をそのままサポートするため、許可ポリシーを指定すれば JAAS を使用して許可を処理できます。

また、eXtreme Scale は、カスタム許可もサポートします。カスタム許可は、プラグイン com.ibm.websphere.objectgrid.security.plugins.ObjectGridAuthorization によって組み込まれます。カスタム許可のフローは以下のとおりです。

7. サーバー・ランタイムが Subject オブジェクトと必要なアクセス権を許可プラグインに送信します。
8. 許可プラグインは、許可サービスを参照して、許可決定を下します。この Subject オブジェクトに対してアクセス権が許可される場合、値 `true` が戻されて、そうでない場合は `false` が戻されます。
9. この true または false の許可決定がサーバー・ランタイムに戻されます。

セキュリティーの実装

このセクションのトピックでは、セキュアな WebSphere eXtreme Scale デプロイメントのプログラム化とプラグイン実装のプログラム化方法について説明します。このセクションは、さまざまなセキュリティー機能を基にして編成されています。各サブトピックで、関係するプラグインとそのプラグインの実装方法を説明します。認証のセクションでは、WebSphere eXtreme Scale のセキュアなデプロイメント環境への接続方法を示します。

クライアント認証: クライアント認証のトピックでは、WebSphere eXtreme Scale クライアントがどのようにクレデンシャルを取得し、サーバーがどのようにクライアントを認証するかについて説明します。また、WebSphere eXtreme Scale クライアントが WebSphere eXtreme Scale のセキュアなサーバーに接続する方法についても説明します。

許可: 許可のトピックでは、JAAS 許可の他にカスタム許可を行うためにどのように ObjectGridAuthorization を使用するかを説明します。

グリッド認証: グリッド認証のトピックでは、サーバー秘密のセキュア・トランスポートのためにどのように SecureTokenManager を使用できるかについて解説します。

Java Management Extensions (JMX) プログラミング: WebSphere eXtreme Scale サーバーを保護する際、JMX クライアントが、サーバーに JMX クレデンシャルを送信する必要がある場合があります。

クライアント認証プログラミング

認証のために WebSphere eXtreme Scale は、クライアントからサーバー・サイドにクレデンシャルを送信するランタイムを提供し、次にオーセンティケーター・プラグインを呼び出してユーザーを認証します。

WebSphere eXtreme Scale のユーザーは、認証を実行するために以下のプラグインを実装する必要があります。

- Credential: Credential は、クライアント・クレデンシャル (ユーザー ID とパスワードのペアなど) を表します。
- CredentialGenerator: CredentialGenerator は、クレデンシャルを生成するためのクレデンシャル・ファクトリーを表します。
- Authenticator: Authenticator は、クライアント・クレデンシャルを認証し、クライアント情報を取得します。

Credential および CredentialGenerator プラグイン

eXtreme Scale クライアントは、認証を必要とするサーバーに接続するときにはクライアント・クレデンシャルを提示する必要があります。クライアントのクレデンシャルは、com.ibm.websphere.objectgrid.security.plugins.Credential インターフェースによって表されます。クライアント・クレデンシャルには、ユーザー名とパスワードのペア、Kerberos チケット、クライアント証明書、またはクライアントとサーバーが同意する任意の形式でのデータがあります。詳しくは、API 資料中のクレデンシャル API に関する情報を参照してください。このインターフェースでは、equals(Object) メソッドおよび hashCode メソッドを定義します。 Credential オブジェクトをサーバー・サイドの鍵として使用することによって認証済み Subject オブジェクトがキャッシュされるため、この 2 つのメソッドは重要です。さらに、WebSphere eXtreme Scale はクレデンシャルを生成するプラグインを提供します。このプラグインは、com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator インターフェースによって示され、クレデンシャルに期限がある場合に役立ちます。この場合は、getCredential メソッドが呼び出されてクレデンシャルが更新されます。

Credential インターフェースでは、equals(Object) メソッドおよび hashCode メソッドを明示的に定義します。 Credential オブジェクトをサーバー・サイドの鍵として使用することによって認証済み Subject オブジェクトがキャッシュされるため、この 2 つのメソッドは重要です。

また、クレデンシャルを生成するために提供されたプラグインも使用することができます。このプラグインは、com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator インターフェースによって示され、クレデンシャルに期限がある場合に役立ちます。この場合は、getCredential メソッドが呼び出されてクレデンシャルが更新されます。詳しくは、API 資料を参照してください。

クレデンシャル・インターフェース用として次の 3 つのデフォルトの実装が提供されています。

- com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential 実装は、ユーザー ID とパスワードのペアを含みます。

- com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredential 実装は、WebSphere Application Server 固有の認証および許可トークンを含みます。これらのトークンを使用すると、同じセキュリティー・ドメイン内のアプリケーション・サーバーにセキュリティー属性を伝搬することができます。

さらに、WebSphere eXtreme Scale はクレデンシャルを生成するプラグインを提供します。このプラグインは、com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator インターフェースによって表されます。WebSphere eXtreme Scale は、次に示す 2 つのデフォルト組み込み実装を提供します。

- com.ibm.websphere.objectgrid.security.plugins.builtins. UserPasswordCredentialGenerator コンストラクターは、ユーザー ID およびパスワードを取ります。getCredential メソッドは、呼び出されると、ユーザー ID およびパスワードが含まれている UserPasswordCredential オブジェクトを返します。
- com.ibm.websphere.objectgrid.security.plugins.builtins. WSTokenCredentialGenerator は、WebSphere Application Server で実行中のクレデンシャル (セキュリティー・トークン) 生成プログラムを表します。getCredential メソッドを呼び出すと、現在のスレッドに関連した Subject が取得されます。その後、この Subject オブジェクトのセキュリティー情報が WSTokenCredential オブジェクトに変換されます。定数 WSTokenCredentialGenerator.RUN_AS_SUBJECT または WSTokenCredentialGenerator.CALLER_SUBJECT を使用して、スレッドから runAs サブジェクトか呼び出し元サブジェクトのいずれを検索するかを指定できます。

UserPasswordCredential および UserPasswordCredentialGenerator

テストの目的で、WebSphere eXtreme Scale は以下のプラグイン実装を提供します。

1. `com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential`
2. `com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator`

ユーザー・パスワードのクレデンシャルでは、ユーザー ID とパスワードを保管します。次にユーザー・パスワードのクレデンシャル・ジェネレーターは、このユーザー ID とパスワードを収容します。

これら 2 つのプラグインを実装する方法を、以下のコード例で示します。

UserPasswordCredential.java

```
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import com.ibm.websphere.objectgrid.security.plugins.Credential;

/**
 * This class represents a credential containing a user ID and password.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see Credential
 * @see UserPasswordCredentialGenerator#getCredential()
 */
public class UserPasswordCredential implements Credential {

    private static final long serialVersionUID = 1409044825541007228L;
```

```
private String ivUserName;

private String ivPassword;

/**
 * Creates a UserPasswordCredential with the specified user name and
 * password.
 *
 * @param userName the user name for this credential
 * @param password the password for this credential
 *
 * @throws IllegalArgumentException if userName or password is <code>null</code>
 */
public UserPasswordCredential(String userName, String password) {
    super();
    if (userName == null || password == null) {
        throw new IllegalArgumentException("User name and password cannot be null.");
    }
    this.ivUserName = userName;
    this.ivPassword = password;
}

/**
 * Gets the user name for this credential.
 *
 * @return   the user name argument that was passed to the constructor
 *           or the <code>setUserName(String)</code>
 *           method of this class
 *
 * @see #setUserName(String)
 */
public String getUserName() {
    return ivUserName;
}

/**
 * Sets the user name for this credential.
 *
 * @param userName the user name to set.
 *
 * @throws IllegalArgumentException if userName is <code>null</code>
 */
public void setUserName(String userName) {
    if (userName == null) {
        throw new IllegalArgumentException("User name cannot be null.");
    }
    this.ivUserName = userName;
}

/**
 * Gets the password for this credential.
 *
 * @return   the password argument that was passed to the constructor
 *           or the <code>setPassword(String)</code>
 *           method of this class
 *
 * @see #setPassword(String)
 */
public String getPassword() {
    return ivPassword;
}

/**
 * Sets the password for this credential.
 *
 * @param password the password to set.
 *
 * @throws IllegalArgumentException if password is <code>null</code>
 */
public void setPassword(String password) {
    if (password == null) {
        throw new IllegalArgumentException("Password cannot be null.");
    }
    this.ivPassword = password;
}

/**
 * Checks two UserPasswordCredential objects for equality.
 * <p>
 * Two UserPasswordCredential objects are equal if and only if their user names
 * and passwords are equal.
 *
 * @param o the object we are testing for equality with this object.
 *
 * @return <code>true</code> if both UserPasswordCredential objects are equivalent.
 *
 * @see Credential#equals(Object)
 */
public boolean equals(Object o) {
    if (this == o) {
```

```
            return true;
        }
        if (o instanceof UserPasswordCredential) {
            UserPasswordCredential other = (UserPasswordCredential) o;
            return other.ivPassword.equals(ivPassword) && other.ivUserName.equals(ivUserName);
        }

        return false;
    }

    /**
     * Returns the hashcode of the UserPasswordCredential object.
     *
     * @return the hash code of this object
     *
     * @see Credential#hashCode()
     */
    public int hashCode() {
        return ivUserName.hashCode() + ivPassword.hashCode();
    }
}
```

UserPasswordCredentialGenerator.java

```
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import java.util.StringTokenizer;

import com.ibm.websphere.objectgrid.security.plugins.Credential;
import com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator;

/**
 * This credential generator creates <code>UserPasswordCredential</code> objects.
 * <p>
 * UserPasswordCredentialGenerator has a one to one relationship with
 * UserPasswordCredential because it can only create a UserPasswordCredential
 * representing one identity.
 *
 * @since WAS XD 6.0.1
 * @ibm-api
 *
 * @see CredentialGenerator
 * @see UserPasswordCredential
 */
public class UserPasswordCredentialGenerator implements CredentialGenerator {

    private String ivUser;

    private String ivPwd;

    /**
     * Creates a UserPasswordCredentialGenerator with no user name or password.
     *
     * @see #setProperties(String)
     */
    public UserPasswordCredentialGenerator() {
        super();
    }

    /**
     * Creates a UserPasswordCredentialGenerator with a specified user name and
     * password
     *
     * @param user the user name
     * @param pwd the password
     */
    public UserPasswordCredentialGenerator(String user, String pwd) {
        ivUser = user;
        ivPwd = pwd;
    }

    /**
     * Creates a new <code>UserPasswordCredential</code> object using this
     * object's user name and password.
     *
     * @return a new <code>UserPasswordCredential</code> instance
     *
     * @see CredentialGenerator#getCredential()
     * @see UserPasswordCredential
     */
    public Credential getCredential() {
        return new UserPasswordCredential(ivUser, ivPwd);
    }
```

```
    /**
     * Gets the password for this credential generator.
     *
     * @return   the password argument that was passed to the constructor
     */
    public String getPassword() {
        return ivPwd;
    }

    /**
     * Gets the user name for this credential.
     *
     * @return   the user argument that was passed to the constructor
     *           of this class
     */
    public String getUserName() {
        return ivUser;
    }
    /**
     * Sets additional properties namely a user name and password.
     *
     * @param properties a properties string  with a user name and
     *                   a password separated by a blank.
     *
     * @throws IllegalArgumentException if the format is not valid
     */
    public void setProperties(String properties) {
        StringTokenizer token = new StringTokenizer(properties, " ");
        if (token.countTokens() != 2) {
            throw new IllegalArgumentException(
                "The properties should have a user name and password and separated by a blank.");
        }

        ivUser = token.nextToken();
        ivPwd = token.nextToken();
    }
    /**
     * Checks two UserPasswordCredentialGenerator objects for equality.
     * <p>
     * Two UserPasswordCredentialGenerator objects are equal if and only if
     * their user names and passwords are equal.
     *
     * @param obj the object we are testing for equality with this object.
     *
     * @return <code>true</code> if both UserPasswordCredentialGenerator objects
     *         are equivalent.
     */
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }

        if (obj != null && obj instanceof UserPasswordCredentialGenerator) {
            UserPasswordCredentialGenerator other = (UserPasswordCredentialGenerator) obj;

            boolean bothUserNull = false;
            boolean bothPwdNull = false;

            if (ivUser == null) {
                if (other.ivUser == null) {
                    bothUserNull = true;
                } else {
                    return false;
                }
            }

            if (ivPwd == null) {
                if (other.ivPwd == null) {
                    bothPwdNull = true;
                } else {
                    return false;
                }
            }

            return (bothUserNull || ivUser.equals(other.ivUser)) && (bothPwdNull || ivPwd.equals(other.ivPwd));
        }

        return false;
    }

    /**
     * Returns the hashcode of the UserPasswordCredentialGenerator object.
     *
     * @return the hash code of this object
     */
    public int hashCode() {

        return ivUser.hashCode() + ivPwd.hashCode();
    }

}
```

UserPasswordCredential クラスには、2 つの属性、ユーザー名およびパスワードが含まれています。 UserPasswordCredentialGenerator は、UserPasswordCredential オブジェクトが含まれるファクトリーとしてサービス提供します。

WSTokenCredential および WSTokenCredentialGenerator

WebSphere eXtreme Scale クライアントおよびサーバーがすべて WebSphere Application Server にデプロイされている場合、クライアント・アプリケーションは、以下の条件が満たされている場合は、これら 2 つの組み込み実装を使用することができます。

1. WebSphere Application Server グローバル・セキュリティーがオンになっている。
2. すべての WebSphere eXtreme Scale クライアントおよびサーバーが WebSphere Application Server Java 仮想マシンで実行されている。
3. アプリケーション・サーバーが、同じセキュリティー・ドメインにある。
4. クライアントが WebSphere Application Server で既に認証されている。

この場合、クライアントはcom.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredentialGenerator クラスを使用して、クレデンシャルを生成できます。サーバーでは、WSAuthenticator 実装・クラスを使用して、クレデンシャルを認証します。

このシナリオは、eXtreme Scale クライアントが既に認証済みであるという事実を利用します。サーバーがあるアプリケーション・サーバーが、クライアントを格納するアプリケーション・サーバーと同じセキュリティー・ドメインにあるため、クライアントから サーバーにセキュリティー・トークンを伝搬することができます。これにより、同じユーザー・レジストリーを再認証する必要がなくなります。

注: CredentialGenerator が常に同じクレデンシャルを生成するわけではありません。有効期限があるリフレッシュ可能なクレデンシャルの場合、CredentialGenerator は、認証が確実に成功するようにするため、最新の有効なクレデンシャルを生成できなければなりません。Credential オブジェクトとして Kerberos チケットを使用することが、1 つの例です。Kerberos チケットがリフレッシュされると、CredentialGenerator は、 CredentialGenerator.getCredential が呼び出されたときに、リフレッシュ後のチケットを取得しなければなりません。

Authenticator プラグイン

eXtreme Scale クライアントが CredentialGenerator オブジェクトを使用して Credential オブジェクトを取得すると、このクライアント Credential オブジェクトがクライアント要求とともに eXtreme Scale サーバーに送信されます。サーバーは、要求の処理前に Credential オブジェクトの認証を行います。 Credential オブジェクトが正常に認証されると、このクライアントを表す Subject オブジェクトが戻されます。

そうすると、この Subject オブジェクトはキャッシュされますが、存続時間がセッション・タイムアウト値に達すると有効期限が切れます。ログイン・セッション・タイムアウト値は、クラスター XML ファイル内にある loginSessionExpirationTime

プロパティーを使用して設定できます。例えば、
`loginSessionExpirationTime="300"` と設定すると、Subject オブジェクトの有効期限は 300 秒で切れます。

この Subject オブジェクトは、後で示すように、要求の認可に使用されます。
eXtreme Scale サーバーは、Authenticator プラグインを使用して、Credential オブジェクトの認証を行います。詳しくは、API 資料中のオーセンティケーターに関する情報を参照してください。

Authenticator プラグインは、eXtreme Scale ランタイムがクライアント・ユーザー・レジストリー (例えば、Lightweight Directory Access Protocol (LDAP) サーバー) からの Credential オブジェクトを認証する所です。

WebSphere eXtreme Scale は即時に使用可能なユーザー・レジストリー構成を提供するわけではありません。ユーザー・レジストリーの構成と管理は、単純化と柔軟性のため、WebSphere eXtreme Scale の外部に残されています。このプラグインはユーザー・レジストリーへの接続と認証時に実装されます。例えば、Authenticator の実装では、クレデンシャルからユーザー ID とパスワードを抽出し、その情報を使用して LDAP サーバーに接続し、検証します。この認証の結果として、Subject オブジェクトが作成されます。この実装で、JAAS ログイン・モジュールを使用する可能性があります。認証の結果として、Subject オブジェクトが戻されます。

このメソッドでは、2 つの例外 InvalidCredentialException および
ExpiredCredentialException が作成されることに注意してください。
InvalidCredentialException 例外は、クレデンシャルが無効であることを示します。
ExpiredCredentialException 例外は、クレデンシャルの期限が切れていることを示します。認証メソッドの結果としてこの 2 つの例外のいずれかが発生した場合、例外はクライアントに送り返されます。ただし、クライアント・ランタイムによって、この 2 つの例外は別々に処理されます。

- エラーが InvalidCredentialException 例外である場合は、クライアント・ランタイムにこの例外が表示されます。ご使用のアプリケーションで例外を処理する必要があります。CredentialGenerator を修正するなどして、操作を再試行します。
- エラーが ExpiredCredentialException 例外であり、再試行数 0 以外の場合は、クライアント・ランタイムによって、CredentialGenerator.getCredential メソッドが再度呼び出され、新しい Credential オブジェクトがサーバーに送信されます。新しいクレデンシャル認証が成功すると、サーバーは要求を処理します。新しいクレデンシャル認証が失敗すると、クライアントに例外が送り返されます。認証の再試行回数が許可値に達しても、クライアントがまだ ExpiredCredentialException 例外を受け取る場合は、ExpiredCredentialException 例外となります。ご使用のアプリケーションでエラーを処理する必要があります。

Authenticator インターフェースは、柔軟性に優れています。Authenticator インターフェースは、独自の方法で実装することができます。例えば、2 種類のユーザー・レジストリーをサポートするように、このインターフェースを実装することもできます。

WebSphere eXtreme Scale には、サンプルのオーセンティケーター・プラグイン実装があります。WebSphere Application Server オーセンティケーター・プラグインの場合を除いて、他の実装はテスト目的のサンプルに過ぎません。

KeyStoreLoginAuthenticator

この例では、テストとサンプルを目的とする eXtreme Scale 組み込み実装である KeyStoreLoginAuthenticator を使用しています (鍵ストアは単純なユーザー・レジストリーであり、実動環境には使用しないようにしてください)。このクラスは、オーセンティケーターの実装方法の説明が目的で表示されていることに注意してください。

KeyStoreLoginAuthenticator.java

```
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007

package com.ibm.websphere.objectgrid.security.plugins.builtins;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import com.ibm.websphere.objectgrid.security.plugins.Authenticator;
import com.ibm.websphere.objectgrid.security.plugins.Credential;
import com.ibm.websphere.objectgrid.security.plugins.ExpiredCredentialException;
import com.ibm.websphere.objectgrid.security.plugins.InvalidCredentialException;
import com.ibm.ws.objectgrid.Constants;
import com.ibm.ws.objectgrid.ObjectGridManagerImpl;
import com.ibm.ws.objectgrid.security.auth.callback.UserPasswordCallbackHandlerImpl;

/**
 * This class is an implementation of the <code>Authenticator</code> interface
 * when a user name and password are used as a credential.
 * <p>
 * When user ID and password authentication is used, the credential passed to the
 * <code>authenticate(Credential)</code> method is a UserPasswordCredential object.
 * <p>
 * This implementation will use a <code>KeyStoreLoginModule</code> to authenticate
 * the user into the key store using the JAAS login module "KeyStoreLogin". The key
 * store can be configured as an option to the <code>KeyStoreLoginModule</code>
 * class. Please see the <code>KeyStoreLoginModule</code> class for more details
 * about how to set up the JAAS login configuration file.
 * <p>
 * This class is only for sample and quick testing purpose. Users should
 * write your own Authenticator implementation which can fit better into
 * the environment.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see Authenticator
 * @see KeyStoreLoginModule
 * @see UserPasswordCredential
 */
public class KeyStoreLoginAuthenticator implements Authenticator {

    /**
     * Creates a new KeyStoreLoginAuthenticator.
     */
    public KeyStoreLoginAuthenticator() {
        super();
    }

    /**
     * Authenticates a <code>UserPasswordCredential</code>.
     * <p>
     * Uses the user name and password from the specified UserPasswordCredential
     * to login to the KeyStoreLoginModule named "KeyStoreLogin".
     *
     * @throws InvalidCredentialException if credential isn't a
     *         UserPasswordCredential or some error occurs during processing
     *         of the supplied UserPasswordCredential
     *
     * @throws ExpiredCredentialException if credential is expired.  This exception
     *         is not used by this implementation
     *
     * @see Authenticator#authenticate(Credential)
     * @see KeyStoreLoginModule
     */
    public Subject authenticate(Credential credential) throws InvalidCredentialException,
   ExpiredCredentialException {

        if (credential == null) {
            throw new InvalidCredentialException("Supplied credential is null");
```

```
        }

        if (! (credential instanceof UserPasswordCredential) ) {
            throw new InvalidCredentialException("Supplied credential is not a UserPasswordCredential");
        }

        UserPasswordCredential cred = (UserPasswordCredential) credential;
        LoginContext lc = null;
        try {
            lc = new LoginContext("KeyStoreLogin",
                    new UserPasswordCallbackHandlerImpl(cred.getUserName(), cred.getPassword().toCharArray()));

            lc.login();

            Subject subject = lc.getSubject();

            return subject;
        }
        catch (LoginException le) {
            throw new InvalidCredentialException(le);
        }
        catch (IllegalArgumentException ile) {
            throw new InvalidCredentialException(ile);
        }
    }
}
```

KeyStoreLoginModule.java

```
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import java.io.File;
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
import javax.security.auth.x500.X500Principal;
import javax.security.auth.x500.X500PrivateCredential;

import com.ibm.websphere.objectgrid.ObjectGridRuntimeException;
import com.ibm.ws.objectgrid.Constants;
import com.ibm.ws.objectgrid.ObjectGridManagerImpl;
import com.ibm.ws.objectgrid.util.ObjectGridUtil;

/**
 * A KeyStoreLoginModule is keystore authentication login module based on
 * JAAS authentication.
 * <p>
 * A login configuration should provide an option "<code>keyStoreFile</code>" to
 * indicate where the keystore file is located. If the <code>keyStoreFile</code>
 * value contains a system property in the form, <code>${system.property}</code>,
 * it will be expanded to the value of the system property.
 * <p>
 * If an option "<code>keyStoreFile</code>" is not provided, the default keystore
 * file name is <code>"${java.home}${/}.keystore"</code>.
 * <p>
 * Here is a Login module configuration example:
 * <pre><code>
 *    KeyStoreLogin {
 *        com.ibm.websphere.objectgrid.security.plugins.builtins.KeystoreLoginModule required
 *            keyStoreFile="${user.dir}${/}security${/}.keystore";
 *    };
 * </code></pre>
 *
 * @ibm-api
 * @since WAS XD 6.0.1
```

```
 *
 * @see LoginModule
 */
public class KeyStoreLoginModule implements LoginModule {

    private static final String CLASS_NAME = KeyStoreLoginModule.class.getName();

    /**
     * Key store file property name
     */
    public static final String KEY_STORE_FILE_PROPERTY_NAME = "keyStoreFile";

    /**
     * Key store type. Only JKS is supported
     */
    public static final String KEYSTORE_TYPE = "JKS";

    /**
     * The default key store file name
     */
    public static final String DEFAULT_KEY_STORE_FILE = "${java.home}${/}.keystore";

    private CallbackHandler handler;

    private Subject subject;

    private boolean debug = false;

    private Set principals = new HashSet();

    private Set publicCreds = new HashSet();

    private Set privateCreds = new HashSet();

    protected KeyStore keyStore;

    /**
     * Creates a new KeyStoreLoginModule.
     */
    public KeyStoreLoginModule() {
    }

    /**
     * Initializes the login module.
     *
     * @see LoginModule#initialize(Subject, CallbackHandler, Map, Map)
     */
    public void initialize(Subject sub, CallbackHandler callbackHandler,
            Map mapSharedState, Map mapOptions) {

        // initialize any configured options
        debug = "true".equalsIgnoreCase((String) mapOptions.get("debug"));

        if (sub == null)
            throw new IllegalArgumentException("Subject is not specified");

        if (callbackHandler == null)
            throw new IllegalArgumentException(
            "CallbackHander is not specified");

        // Get the key store path
        String sKeyStorePath = (String) mapOptions
            .get(KEY_STORE_FILE_PROPERTY_NAME);

        // If there is no key store path, the default one is the .keystore
        // file in the java home directory
        if (sKeyStorePath == null) {
            sKeyStorePath = DEFAULT_KEY_STORE_FILE;
        }

        // Replace the system enviroment variable
        sKeyStorePath = ObjectGridUtil.replaceVar(sKeyStorePath);

        File fileKeyStore = new File(sKeyStorePath);

        try {
            KeyStore store = KeyStore.getInstance("JKS");
            store.load(new FileInputStream(fileKeyStore), null);

            // Save the key store
            keyStore = store;

            if (debug) {
                System.out.println("[KeyStoreLoginModule] initialize: Successfully loaded key store");
            }
        }
        catch (Exception e) {
            ObjectGridRuntimeException re = new ObjectGridRuntimeException(
                    "Failed to load keystore: " + fileKeyStore.getAbsolutePath());
            re.initCause(e);
            if (debug) {
```

```
            System.out.println("[KeyStoreLoginModule] initialize: Key store loading failed with exception "
                    + e.getMessage());
        }
    }

    this.subject = sub;
    this.handler = callbackHandler;
}

/**
 * Authenticates a user based on the keystore file.
 *
 * @see LoginModule#login()
 */
public boolean login() throws LoginException {

    if (debug) {
        System.out.println("[KeyStoreLoginModule] login: entry");
    }

    String name = null;
    char pwd[] = null;

    if (keyStore == null || subject == null || handler == null) {
        throw new LoginException("Module initialization failed");
    }

    NameCallback nameCallback = new NameCallback("Username:");
    PasswordCallback pwdCallback = new PasswordCallback("Password:", false);

    try {
        handler.handle(new Callback[] { nameCallback, pwdCallback });
    }
    catch (Exception e) {
        throw new LoginException("Callback failed: " + e);
    }

    name = nameCallback.getName();
    char[] tempPwd = pwdCallback.getPassword();

    if (tempPwd == null) {
        // treat a NULL password as an empty password
        tempPwd = new char[0];
    }
    pwd = new char[tempPwd.length];
    System.arraycopy(tempPwd, 0, pwd, 0, tempPwd.length);

    pwdCallback.clearPassword();

    if (debug) {
        System.out.println("[KeyStoreLoginModule] login: "
                + "user entered user name: " + name);
    }

    // Validate the user name and password
    try {
        validate(name, pwd);
    }
    catch (SecurityException se) {
        principals.clear();
        publicCreds.clear();
        privateCreds.clear();
        LoginException le = new LoginException(
        "Exception encountered during login");
        le.initCause(se);

        throw le;
    }

    if (debug) {
        System.out.println("[KeyStoreLoginModule] login: exit");
    }
    return true;
}

/**
 * Indicates the user is accepted.
 * <p>
 * This method is called only if the user is authenticated by all modules in
 * the login configuration file. The principal objects will be added to the
 * stored subject.
 *
 * @return false if for some reason the principals cannot be added; true
 *         otherwise
 *
 * @exception LoginException
 *                LoginException is thrown if the subject is readonly or if
 *                any unrecoverable exceptions is encountered.
 *
 * @see LoginModule#commit()
 */
```

```
public boolean commit() throws LoginException {
    if (debug) {
        System.out.println("[KeyStoreLoginModule] commit: entry");
    }

    if (principals.isEmpty()) {
        throw new IllegalStateException("Commit is called out of sequence");
    }

    if (subject.isReadOnly()) {
        throw new LoginException("Subject is Readonly");
    }

    subject.getPrincipals().addAll(principals);
    subject.getPublicCredentials().addAll(publicCreds);
    subject.getPrivateCredentials().addAll(privateCreds);

    principals.clear();
    publicCreds.clear();
    privateCreds.clear();

    if (debug) {
        System.out.println("[KeyStoreLoginModule] commit: exit");
    }
    return true;
}

/**
 * Indicates the user is not accepted
 *
 * @see LoginModule#abort()
 */
public boolean abort() throws LoginException {
    boolean b = logout();
    return b;
}

/**
 * Logs the user out. Clear all the maps.
 *
 * @see LoginModule#logout()
 */
public boolean logout() throws LoginException {


    // Clear the instance variables
    principals.clear();
    publicCreds.clear();
    privateCreds.clear();

    // clear maps in the subject
    if (!subject.isReadOnly()) {
        if (subject.getPrincipals() != null) {
            subject.getPrincipals().clear();
        }

        if (subject.getPublicCredentials() != null) {
            subject.getPublicCredentials().clear();
        }

        if (subject.getPrivateCredentials() != null) {
            subject.getPrivateCredentials().clear();
        }
    }
    return true;
}

/**
 * Validates the user name and password based on the keystore.
 *
 * @param userName user name
 * @param password password
 * @throws SecurityException if any exceptions encountered
 */
private void validate(String userName, char password[])
    throws SecurityException {

    PrivateKey privateKey = null;

    // Get the private key from the keystore
    try {
        privateKey = (PrivateKey) keyStore.getKey(userName, password);
    }
    catch (NoSuchAlgorithmException nsae) {
        SecurityException se = new SecurityException();
        se.initCause(nsae);
        throw se;
    }
    catch (KeyStoreException kse) {
        SecurityException se = new SecurityException();
        se.initCause(kse);
```

```
            throw se;
        }
        catch (UnrecoverableKeyException uke) {
            SecurityException se = new SecurityException();
            se.initCause(uke);
            throw se;
        }

        if (privateKey == null) {
            throw new SecurityException("Invalid name: " + userName);
        }

        // Check the certificats
        Certificate certs[] = null;
        try {
            certs = keyStore.getCertificateChain(userName);
        }
        catch (KeyStoreException kse) {
            SecurityException se = new SecurityException();
            se.initCause(kse);
            throw se;
        }

        if (debug) {
            System.out.println("  Print out the certificates:");
            for (int i = 0; i < certs.length; i++) {
                System.out.println("  certificate " + i);
                System.out.println("    " + certs[i]);
            }
        }

        if (certs != null && certs.length > 0) {

            // If the first certificate is an X509Certificate
            if (certs[0] instanceof X509Certificate) {
                try {
                    // Get the first certificate which represents the user
                    X509Certificate certX509 = (X509Certificate) certs[0];

                    // Create a principal
                    X500Principal principal = new X500Principal(certX509
                            .getIssuerDN()
                            .getName());
                    principals.add(principal);

                    if (debug) {
                        System.out.println("  Principal added: " + principal);
                    }
                    // Create the certification path object and add it to the
                    // public credential set
                    CertificateFactory factory = CertificateFactory
                        .getInstance("X.509");
                    java.security.cert.CertPath certPath = factory
                        .generateCertPath(Arrays.asList(certs));
                    publicCreds.add(certPath);

                    // Add the private credential to the private credential set
                    privateCreds.add(new X500PrivateCredential(certX509,
                            privateKey, userName));

                }
                catch (CertificateException ce) {
                    SecurityException se = new SecurityException();
                    se.initCause(ce);
                    throw se;
                }
            }
            else {
                // The first certificate is not an X509Certificate
                // We just add the certificate to the public credential set
                // and the private key to the private credential set.
                publicCreds.add(certs[0]);
                privateCreds.add(privateKey);
            }
        }
    }
}
```

LDAP オーセンティケーター・プラグインの使用

LDAP サーバーに対するユーザー名およびパスワード認証を処理するために、com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPAuthenticator のデフォルトの実装が用意されています。この実装では、LDAPLogin ログイン・モジュールを使

用して、ユーザーを Lightweight Directory Access Protocol (LDAP) サーバーにログインさせます。以下のスニペットでは、認証メソッドの実装方法を説明しています。

```
/**
* @see com.ibm.ws.objectgrid.security.plugins.Authenticator#
* authenticate(LDAPLogin)
*/
public Subject authenticate(Credential credential) throws
InvalidCredentialException, ExpiredCredentialException {

    UserPasswordCredential cred = (UserPasswordCredential) credential;
    LoginContext lc = null;
    try {
        lc = new LoginContext("LDAPLogin",
            new UserPasswordCallbackHandlerImpl(cred.getUserName(),
            cred.getPassword().toCharArray()));

        lc.login();

        Subject subject = lc.getSubject();

        return subject;
    }
    catch (LoginException le) {
        throw new InvalidCredentialException(le);
    }
    catch (IllegalArgumentException ile) {
        throw new InvalidCredentialException(ile);
    }
}
```

また、この目的のため、eXtreme Scale には、ログイン・モジュール com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPLoginModule が同梱されています。JAAS ログイン構成ファイルに以下の 2 つのオプションを指定する必要があります。

- providerURL: LDAP サーバー・プロバイダー URL
- factoryClass: LDAP コンテキスト・ファクトリー実装クラス

LDAPLoginModule モジュールは、com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPAuthenticationHelper.authenticate メソッドを呼び出します。以下のコード・スニペットは、LDAPAuthenticationHelper の認証メソッドを実装する方法を示しています。

```
/**
* Authenticate the user to the LDAP directory.
* @param user the user ID, e.g., uid=xxxxxx,c=us,ou=bluepages,o=ibm.com
* @param pwd the password
*
* @throws NamingException
*/
public String[] authenticate(String user, String pwd)
throws NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, factoryClass);
    env.put(Context.PROVIDER_URL, providerURL);
    env.put(Context.SECURITY_PRINCIPAL, user);
    env.put(Context.SECURITY_CREDENTIALS, pwd);
    env.put(Context.SECURITY_AUTHENTICATION, "simple");

    InitialContext initialContext = new InitialContext(env);

    // Look up for the user
    DirContext dirCtx = (DirContext) initialContext.lookup(user);

    String uid = null;
    int iComma = user.indexOf(",");
    int iEqual = user.indexOf("=");
```

```
    if (iComma > 0 && iComma > 0) {
        uid = user.substring(iEqual + 1, iComma);
    }
    else {
        uid = user;
    }

    Attributes attributes = dirCtx.getAttributes("");

    // Check the UID
    String thisUID = (String) (attributes.get(UID).get());

    String thisDept = (String) (attributes.get(HR_DEPT).get());

    if (thisUID.equals(uid)) {
        return new String[] { thisUID, thisDept };
    }
    else {
        return null;
    }
}
```

認証が成功した場合、ID とパスワードは有効であるとみなされます。次に、ログイン・モジュールは、この認証メソッドから ID 情報および部門情報を取得します。ログイン・モジュールでは、2 つのプリンシパル SimpleUserPrincipal および SimpleDeptPrincipal が作成されます。認証済みサブジェクトを使用して、グループ許可 (ここでは、部門がグループです) および個々の許可を行うことができます。

以下に、LDAP サーバーへのログインに使用されるログイン・モジュールの構成例を示します。

```
LDAPLogin { com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPLoginModule required
    providerURL="ldap://directory.acme.com:389/"
    factoryClass="com.sun.jndi.ldap.LdapCtxFactory";
};
```

前の構成では、LDAP サーバーは `ldap://directory.acme.com:389/server` を指しています。この設定をご使用の LDAP サーバーに変更します。このログイン・モジュールでは、指定された ID およびパスワードを使用して、LDAP サーバーに接続します。この実装は、テスト目的のみです。

WebSphere Application Server オーセンティケーター・プラグインの使用

さらに、eXtreme Scale には、WebSphere Application Server セキュリティー・インフラストラクチャーを使用するための com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenAuthenticator 組み込み実装も用意されています。この組み込み実装は、以下の条件が満たされている場合に使用できます。

1. WebSphere Application Server グローバル・セキュリティーがオンになっている。
2. すべての eXtreme Scale クライアントおよびサーバーが WebSphere Application Server JVM で起動している。
3. これらのアプリケーション・サーバーが、同じセキュリティー・ドメインにある。
4. eXtreme Scale クライアントが、WebSphere Application Server で認証済みである。

クライアントはcom.ibm.websphere.objectgrid.security.plugins.builtins.
WSTokenCredentialGenerator クラスを使用して、クレデンシャルを生成できます。サーバーでは、Authenticator 実装クラスを使用して、クレデンシャルを認証します。トークンが正常に認証されると、Subject オブジェクトが戻されます。

このシナリオの利点は、クライアントが既に認証済みであることです。サーバーがあるアプリケーション・サーバーが、クライアントを格納するアプリケーション・サーバーと同じセキュリティー・ドメインにあるため、クライアントから サーバーにセキュリティー・トークンを伝搬することができます。これにより、同じユーザー・レジストリーを再認証する必要がなくなります。

Tivoli® Access Manager オーセンティケーター・プラグインの使用

Tivoli Access Manager は、セキュリティー・サーバーとして幅広く使用されています。Tivoli Access Manager が提供するログイン・モジュールを使用して、オーセンティケーターを実装することもできます。

Tivoli Access Manager でユーザーを認証するには、com.tivoli.mts.PDLoginModule ログイン・モジュールを適用します。このモジュールの場合、呼び出し側アプリケーションが以下の情報を提供する必要があります。

1. 短縮名または X.500 名 (DN) として指定されたプリンシパル名
2. パスワード

ログイン・モジュールはプリンシパルを認証し、Tivoli Access Manager クレデンシャルを返します。ログイン・モジュールは、呼び出し側アプリケーションによって以下の情報が提供されると想定しています。

1. javax.security.auth.callback.NameCallback オブジェクトを通してのユーザー名
2. javax.security.auth.callback.PasswordCallback オブジェクトを通してのパスワード

Tivoli Access Manager クレデンシャルが正常に取得されると、JAAS LoginModule によって Subject および PDPrincipal が作成されます。Tivoli Access Manager 認証用の組み込みは、PDLoginModule モジュールで使用されるだけなので、用意されていません。詳しくは、「IBM Tivoli Access Manager Authorization Java Classes デベロッパーズ・リファレンス」を参照してください。

WebSphere eXtreme Scale へのセキュアな接続

eXtreme Scale クライアントをサーバーにセキュアに接続するには、ObjectGridManager インターフェースで、ClientSecurityConfiguration オブジェクトを使用する connect メソッドを使用します。以下に簡単な例を示します。

```
public ClientClusterContext connect(String catalogServerAddresses,
   ClientSecurityConfiguration securityProps,
     URL overRideObjectGridXml) throws ConnectException;
```

このメソッドは、クライアント・セキュリティー構成を表すインターフェースである ClientSecurityConfiguration タイプのパラメーターを使用します。
com.ibm.websphere.objectgrid.security.config.ClientSecurityConfigurationFactory public API を使用して、このインターフェースのインスタンスをデフォルト値で作成するか、または WebSphere eXtreme Scale クライアント・プロパティー・ファイルを渡

してインスタンスを作成します。このファイルには、認証に関連した以下のプロパティーが含まれています。正符号 (+) でマークされた値はデフォルトです。

- securityEnabled (true, false+): このプロパティーは、セキュリティーが使用可能かどうかを示します。クライアントがサーバーに接続されている場合、クライアント・サイドとサーバー・サイドの securityEnabled 値は、両方とも true または false である必要があります。例えば、接続されるサーバーのセキュリティーが使用可能な場合、クライアントはこのプロパティーを true に設定してサーバーに接続する必要があります。
- authenticationRetryCount (an integer value, 0+): このプロパティーでは、クレデンシャルの期限が切れている場合のログインの再試行回数を決定します。値が 0 の場合、再試行は行われません。認証再試行は、クレデンシャルの期限が切れている場合にのみ適用されます。クレデンシャルが無効な場合、再試行は行われません。操作の再試行は、ご使用のアプリケーションで対応する必要があります。

com.ibm.websphere.objectgrid.security.config.ClientSecurityConfiguration オブジェクトを作成した後、以下のメソッドを使用して、クライアントに credentialGenerator オブジェクトを設定します。

```
/**
* Set the {@link CredentialGenerator} object for this client.
* @param generator the CredentialGenerator object associated with this client
*/
void setCredentialGenerator(CredentialGenerator generator);
```

以下のように、WebSphere eXtreme Scale クライアント・プロパティー・ファイルにも CredentialGenerator オブジェクトを設定できます。

- credentialGeneratorClass: CredentialGenerator オブジェクトのクラス実装名。デフォルトのコンストラクターを指定する必要があります。
- credentialGeneratorProps: CredentialGenerator クラスのプロパティー。この値がヌル以外の場合、このプロパティーは、setProperties(String) メソッドを使用して、構成済みの CredentialGenerator オブジェクトに設定されます。

以下に、ClientSecurityConfiguration のインスタンスを生成し、このインスタンスを使用してサーバーに接続する例を示します。

```
/**
* Get a secure ClientClusterContext
* @return a secure ClientClusterContext object
*/
protected ClientClusterContext connect() throws ConnectException {
ClientSecurityConfiguration csConfig = ClientSecurityConfigurationFactory
.getClientSecurityConfiguration("/properties/security.ogclient.props");

UserPasswordCredentialGenerator gen= new
UserPasswordCredentialGenerator("manager", "manager1");

csConfig.setCredentialGenerator(gen);

return objectGridManager.connect(csConfig, null);
}
```

接続が呼び出されると、WebSphere eXtreme Scale クライアントは、CredentialGenerator.getCredential メソッドを呼び出してクライアント・クレデンシャルを取得します。このクレデンシャルは、接続要求とともにサーバーに送信されて、認証されます。

セッションごとに異なる CredentialGenerator インスタンスの使用

WebSphere eXtreme Scale クライアントは 1 つのクライアント ID を表す場合もあれば、複数の ID を表す場合もあります。以下に、複数の ID を表す場合の例を示します。この例では、WebSphere eXtreme Scale クライアントは、Web サーバーで作成され、共用されます。この Web サーバーのすべてのサーブレットで、この 1 つの WebSphere eXtreme Scale クライアントが使用されます。各サーブレットが異なる Web クライアントを表すため、WebSphere eXtreme Scale サーバーへ要求を送信するときは、異なるクレデンシャルを使用します。

WebSphere eXtreme Scale は、セッション・レベルでのクレデンシャルの変更に対応しています。各セッションでは、個別の CredentialGenerator オブジェクトを使用できます。したがって、前のシナリオは、サーブレットで個別の CredentialGenerator オブジェクトを使用してセッションを取得することにより実装されます。以下の例は、ObjectGridManager インターフェースの ObjectGrid.getSession (CredentialGenerator) メソッドを示しています。

```
/**
     * Get a session using a <code>CredentialGenerator</code>.
     * <p>
     * This method can only be called by the ObjectGrid client in an ObjectGrid
     * client server environment. If ObjectGrid is used in a local model, that is,
     * within the same JVM with no client or server existing, <code>getSession(Subject)</code>
     * or the <code>SubjectSource</code> plugin should be used to secure the ObjectGrid.
     *
     * <p>If the <code>initialize()</code> method has not been invoked prior to
     * the first <code>getSession</code> invocation, an implicit initialization
     * will occur.  This ensures that all of the configuration is complete
     * before any runtime usage is required.</p>
     *
     * @param credGen A <code>CredentialGenerator</code> for generating a credential
     *                for the session returned.
     *
     * @return An instance of <code>Session</code>
     *
     * @throws ObjectGridException if an error occurs during processing
     * @throws TransactionCallbackException if the <code>TransactionCallback</code>
     *         throws an exception
     * @throws IllegalStateException if this method is called after the
     *         <code>destroy()</code> method is called.
     *
     * @see #destroy()
     * @see #initialize()
     * @see CredentialGenerator
     * @see Session
     * @since WAS XD 6.0.1
 */
Session getSession(CredentialGenerator credGen) throws
ObjectGridException, TransactionCallbackException;
```

以下に例を示します。

```
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();

CredentialGenerator credGenManager = new UserPasswordCredentialGenerator("manager", "xxxxxx");
CredentialGenerator credGenEmployee = new UserPasswordCredentialGenerator("employee", "xxxxxx");

ObjectGrid og = ogManager.getObjectGrid(ctx, "accounting");

// Get a session with CredentialGenerator;
Session session = og.getSession(credGenManager );

// Get the employee map
ObjectMap om = session.getMap("employee");

// start a transaction.
session.begin();

Object rec1 = map.get("xxxxxx");

session.commit();

// Get another session with a different CredentialGenerator;
session = og.getSession(credGenEmployee );

// Get the employee map
om = session.getMap("employee");
```

```
// start a transaction.
session.begin();

Object rec2 = map.get("xxxxx");

session.commit();
```

ObjectGird.getSession メソッドを使用して Session オブジェクトを取得する場合、このセッションでは、ClientConfigurationSecurity オブジェクトに設定されている CredentialGenerator オブジェクトを使用します。ObjectGrid.getSession (CredentialGenerator) メソッドは、ClientSecurityConfiguration オブジェクトに設定されている CredentialGenerator をオーバーライドします。

Session オブジェクトを再使用できる場合は、パフォーマンスが向上します。ただし、ObjectGrid.getSession(CredentialGenerator) メソッドの呼び出しにかかるコストは、さほど高くありません。主なオーバーヘッドは、増加したオブジェクト・ガーベッジ・コレクション時間となります。Session オブジェクトの完了後には、必ず参照を解放してください。一般的に、Session オブジェクトで ID を共用できる場合は、Session オブジェクトを再使用してください。そうでない場合は、ObjectGrid.getSession(CredentialGenerator) メソッドを使用してください。

クライアント許可プログラミング

WebSphere eXtreme Scale は、Java 認証・承認サービス (JAAS) 許可をすぐに使用できるようサポートし、ObjectGridAuthorization インターフェースを使用するカスタム許可もサポートします。

ObjectGridAuthorization プラグインは、Subject オブジェクトで表されるプリンシパルに対して ObjectGrid、ObjectMap、および JavaMap の各アクセスを独自の方法で許可する場合に使用します。このプラグインの通常の実装の目的は、Subject オブジェクトからプリンシパルを取得し、このプリンシパルに指定の許可が付与されているかどうかを確認することです。

checkPermission(Subject, Permission) メソッドに渡される許可は、以下の許可のいずれかです。

1. MapPermission
2. ObjectGridPermission
3. ServerMapPermission
4. AgentPermission

詳しくは、ObjectGridAuthorization API 資料を参照してください。

MapPermission

com.ibm.websphere.objectgrid.security.MapPermission パブリック・クラスは ObjectGrid リソース、特に ObjectMap または JavaMap インターフェースのメソッドへの許可を表します。 WebSphere eXtreme Scale は、ObjectMap および JavaMap のメソッドにアクセスするための以下の許可ストリングを定義します。

1. **read**: マップからデータを読み取る許可。整数定数は `MapPermission.READ` として定義されます。

2. **write**: マップのデータを更新する許可。整数定数は `MapPermission.WRITE` として定義されます。
3. **insert**: マップにデータを挿入する許可。整数定数は `MapPermission.INSERT` として定義されます。
4. **remove**: マップからデータを読み取る許可。整数定数は `MapPermission.REMOVE` として定義されます。
5. **invalidate**: マップのデータを無効にする許可。整数定数は `MapPermission.INVALIDATE` として定義されます。
6. **all**: 上記すべての許可(read、write、insert、remote、invalidate)。整数定数は `MapPermission.ALL` として定義されます。

詳しくは、MapPermission API 資料を参照してください。

([ObjectGrid_name].[ObjectMap_name]) というフォーマットの完全修飾 ObjectGrid マップ名、および許可ストリングまたは整数値を渡すことによって、MapPermission オブジェクトを構成できます。許可ストリングは、上記の許可ストリングで構成されるコンマ区切りストリングにしたり (read, insert など)、または all にしたりできます。許可整数値は、上記のすべての許可整数定数いずれにも、または MapPermission.READ|MapPermission.WRITE など、いくつかの整数許可定数の数値にすることができます。

ObjectMap または JavaMap メソッドが呼び出されると、許可が実行されます。eXtreme Scale ランタイムが、さまざまなメソッドの異なる許可を確認します。必要な許可がクライアントに与えられていない場合は、AccessControlException が発生します。

表 12. メソッドと必要な MapPermission のリスト

許可	ObjectMap/JavaMap
read	boolean containsKey(Object)
	boolean equals(Object)
	Object get(Object)
	Object get(Object, Serializable)
	List getAll(List)
	List getAll(List keyList, Serializable)
	List getAllForUpdate(List)
	List getAllForUpdate(List, Serializable)
	Object getForUpdate(Object)
	Object getForUpdate(Object, Serializable)
	public Object getNextKey(long)
write	Object put(Object key, Object value)
	void put(Object, Object, Serializable)
	void putAll(Map)
	void putAll(Map, Serializable)
	void update(Object, Object)
	void update(Object, Object, Serializable)

表 12. メソッドと必要な MapPermission のリスト (続き)

許可	ObjectMap/JavaMap
insert	public void insert (Object, Object)
	void insert(Object, Object, Serializable)
remove	Object remove (Object)
	void removeAll(Collection)
	void clear()
invalidate	public void invalidate (Object, boolean)
	void invalidateAll(Collection, boolean)
	void invalidateUsingKeyword(Serializable)
	int setTimeToLive(int)

許可の基になるのは、使用するメソッドのみであり、メソッドが実際に行う機能ではありません。例えば、put メソッドでは、レコードの有無に基づいて、レコードを挿入または更新できます。ただし、挿入と更新の事例の区別はされません。

ある種類の操作を組み合わせて、別の種類の操作を実現することもできます。例えば、更新は、除去と挿入によって達成することができます。許可ポリシーを設計する場合は、これらの組み合わせを考慮に入れてください。

ObjectGridPermission

com.ibm.websphere.objectgrid.security.ObjectGridPermission は、ObjectGrid への許可を表します。

- Query: オブジェクト照会またはエンティティー照会を作成する許可。整数定数は ObjectGridPermission.QUERY として定義されます。
- Dynamic map: マップ・テンプレートに基づいて動的マップを作成する許可。整数定数は ObjectGridPermission.DYNAMIC_MAP として定義されます。

詳しくは、ObjectGridPermission API 資料を参照してください。

以下の表は、メソッドと必要な ObjectGridPermission のリストです。

表 13. メソッドと必要な ObjectGridPermission のリスト

許可アクション	メソッド
query	com.ibm.websphere.objectgrid.Session.createObjectQuery(String)
query	com.ibm.websphere.objectgrid.em.EntityManager.createQuery(String)
dynamicmap	com.ibm.websphere.objectgrid.Session.getMap(String)

ServerMapPermission

ServerMapPermission は、サーバーでホストされる ObjectMap への許可を表します。許可の名前は ObjectGrid マップ名のフルネームです。以下の 2 つのアクションを備えています。

1. replicate: ニア・キャッシュにサーバー・マップを複製するための許可。

2. dynamicIndex: クライアントがサーバーの動的索引を作成または削除するための許可。

詳しくは、ServerMapPermission API 資料を参照してください。以下の表に、さまざまな ServerMapPermission を必要とするメソッドを示します。

表 14. サーバーでホストされる ObjectMap への許可

許可アクション	メソッド
replicate	com.ibm.websphere.objectgrid.ClientReplicableMap.enableClientReplication(Mode, int[], ReplicationMapListener)
dynamicIndex	com.ibm.websphere.objectgrid.BackingMap.createDynamicIndex(String, boolean, String, DynamicIndexCallback)
dynamicIndex	com.ibm.websphere.objectgrid.BackingMap.removeDynamicIndex(String)

AgentPermission

AgentPermission は、datagrid エージェントに対する許可を表します。許可の名前は、ObjectGrid マップのフルネームで、アクションの名前は、エージェント実装クラス名またはパッケージ名をコンマで区切ったストリングです。

詳しくは、AgentPermission API 資料を参照してください。

クラス com.ibm.websphere.objectgrid.datagrid.AgentManager の以下のメソッドには、AgentPermission が必要です。

```
com.ibm.websphere.objectgrid.datagrid.AgentManager#callMapAgent(MapGridAgent, Collection)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callMapAgent(MapGridAgent)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callReduceAgent(ReduceGridAgent, Collection)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callReduceAgent(ReduceGridAgent, Collection)
```

許可メカニズム

WebSphere eXtreme Scale は、2 種類の許可メカニズム、Java 認証・承認サービス (JAAS) 許可とカスタム許可をサポートしています。これらのメカニズムは、すべての許可に適用されます。JAAS 許可は、ユーザー中心のアクセス制御により Java セキュリティー・ポリシーを拡張します。許可の付与は、実行されているコードだけではなく、コードの実行者に基づいて行うこともできます。JAAS 許可は、SDK バージョン 1.4 以降に付属しています。

さらに、WebSphere eXtreme Scale では、以下のプラグインによってカスタム許可もサポートしています。

- ObjectGridAuthorization: すべての成果物へのアクセスの許可方法をカスタマイズします。

JAAS 許可を使用したくない場合は、独自の許可メカニズムを実装できます。カスタム許可メカニズムでは、ポリシー・データベース、ポリシー・サーバー、または Tivoli Access Manager を使用して、許可を管理できます。

許可メカニズムを構成するには、以下の 2 とおりの方法があります。

1. *XML 構成:* ObjectGrid XML ファイルを使用して ObjectGrid を定義し、許可メカニズムを AUTHORIZATION_MECHANISM_JAAS または AUTHORIZATION_MECHANISM_CUSTOM のいずれかに設定します。以下は、エンタープライズ・アプリケーション ObjectGridSample で使用している secure-objectgrid-definition.xml ファイルです。

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS">
  <bean id="TransactionCallback"
classname="com.ibm.websphere.samples.objectgrid.HeapTransactionCallback" />
...
</objectGrids>
```

2. *プログラマチック構成:* メソッド ObjectGrid.setAuthorizationMechanism(int) を使用して ObjectGrid を作成する場合、以下のメソッドを呼び出して許可メカニズムを設定できます。このメソッドの呼び出しは、直接 ObjectGrid インスタンスを生成する場合のローカル WebSphere eXtreme Scale プログラミング・モデルにのみ適用されます。

```
/**
 * Set the authorization Mechanism. The default is
 * com.ibm.websphere.objectgrid.security.SecurityConstants.
 * AUTHORIZATION_MECHANISM_JAAS.
 * @param authMechanism the map authorization mechanism
*/
void setAuthorizationMechanism(int authMechanism);
```

JAAS 許可

javax.security.auth.Subject オブジェクトは、認証済みユーザーを表します。 Subject は、プリンシパルのセットから構成され、各プリンシパルはそのユーザーの ID を表します。例えば、Subject には、名前のプリンシパル (Joe Smith など) とグループのプリンシパル (manager など) を持たせることができます。

JAAS 許可ポリシーを使用すると、許可を特定のプリンシパルに付与することができます。WebSphere eXtreme Scale は、Subject と現行のアクセス制御コンテキストを関連付けます。 ObjectMap または JavaMap メソッドに対する各呼び出しごとに、Java ランタイムによって自動的に、ポリシーが特定のプリンシパルのみに必要な許可を付与しているかどうかが判断されます。付与している場合、アクセス制御コンテキストに関連付けられた Subject に、指定されたプリンシパルが含まれているときにのみ操作が許可されます。

ポリシー・ファイルのポリシー構文について理解している必要があります。JAAS 許可について詳しくは、「JAAS Reference Guide」を参照してください。

WebSphere eXtreme Scale には、ObjectMap および JavaMap メソッドの呼び出しに対する JAAS 許可の検査に使用される特別なコードベースがあります。この特別なコードベースは、`http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction` です。プリンシパルに ObjectMap または JavaMap 許可を与える場合は、このコード・ベースを使用します。この特別なコードは、eXtreme Scale の Java アーカイブ (JAR) ファイルにすべての許可が与えられるため、作成されました。

MapPermission 許可を付与するためのポリシーのテンプレートは、以下のとおりです。

```
grant codeBase "http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction"
   <Principal field(s)>{
    permission com.ibm.websphere.objectgrid.security.MapPermission
               "[ObjectGrid_name].[ObjectMap_name]", "action";
```

```
    ....
    permission com.ibm.websphere.objectgrid.security.MapPermission
               "[ObjectGrid_name].[ObjectMap_name]", "action";
  };
```

Principal フィールドの例は、以下のとおりです。

```
principal Principal_class "principal_name"
```

このポリシーでは、insert および read 許可のみが特定のプリンシパルに対する 4 つのマップに与えられます。他のポリシー・ファイル fullAccessAuth.policy では、プリンシパルに対するこれらのマップに all 許可が与えられます。アプリケーションを実行する前に、principal_name とプリンシパル・クラスを適切な値に変更してください。 principal_name の値は、ユーザー・レジストリーに応じて異なります。例えば、ローカル OS をユーザー・レジストリーとして使用する場合は、マシン名は MACH1、ユーザー ID は user1、principal_name は MACH1/user1 になります。

JAAS 許可ポリシーは、Java ポリシー・ファイルに直接入れることができます。または、別の JAAS 許可ファイルに入れてから、

```
-Djava.security.auth.policy=file:[JAAS_AUTH_POLICY_FILE]
```

JVM 引数を使用するか、

```
-Dauth.policy.url.x=file:[JAAS_AUTH_POLICY_FILE]
```

プロパティーを java.security ファイル内で使用することによって設定できます。

カスタム ObjectGrid 許可

ObjectGridAuthorization プラグインは、Subject オブジェクトで表されるプリンシパルに対して ObjectGrid、ObjectMap、および JavaMap の各アクセスを独自の方法で許可する場合に使用します。このプラグインの通常の実装の目的は、Subject オブジェクトからプリンシパルを取得し、このプリンシパルに指定の許可が付与されているかどうかを確認することです。

checkPermission(Subject, Permission) メソッドに渡される許可は、以下のいずれかです。

1. MapPermission
2. ObjectGridPermission
3. AgentPermission
4. ServerMapPermission

詳しくは、ObjectGridAuthorization API 資料を参照してください。

ObjectGridAuthorization プラグインは、以下の方法で構成することができます。

1. *XML 構成:* ObjectGrid XML ファイルを使用して、ObjectAuthorization プラグインを定義できます。以下に例を示します。

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_CUSTOM">
...
  <bean id="ObjectGridAuthorization"
className="com.acme.ObjectGridAuthorizationImpl" />
</objectGrids>
```

2. *プログラマチック構成:* API メソッド ObjectGrid.setObjectGridAuthorization (ObjectGridAuthorization) を使用して ObjectGrid を作成する場合、以下のメソッドを呼び出して許可プラグインを設定できます。このメソッドは、直接 ObjectGrid インスタンスを生成するときに、ローカル eXtreme Scale プログラミング・モデルにのみ適用されます。

```
/**
     * Sets the <code>ObjectGridAuthorization</code> for this ObjectGrid instance.
     * <p>
     * Passing <code>null</code> to this method removes a previously set
     * <code>ObjectGridAuthorization</code> object from an earlier invocation of this method
     * and indicates that this <code>ObjectGrid</code> is not associated with a
     * <code>ObjectGridAuthorization</code> object.
     * <p>
     * This method should only be used when ObjectGrid security is enabled. If
     * the ObjectGrid security is disabled, the provided <code>ObjectGridAuthorization</code> object
     * will not be used.
     * <p>
     * A <code>ObjectGridAuthorization</code> plugin can be used to authorize
     * access to the ObjectGrid and maps. Please refer to <code>ObjectGridAuthorization</code> for more details.
     *
     * <p>
     * As of XD 6.1, the <code>setMapAuthorization</code> is deprecated and
     * <code>setObjectGridAuthorization</code> is recommended for use. However,
     * if both <code>MapAuthorization</code> plugin and <code>ObjectGridAuthorization</code> plugin
     * are used, ObjectGrid will use the provided <code>MapAuthorization</code> to authorize map accesses,
     * even though it is deprecated.
     * <p>
     * Note, to avoid an <code>IllegalStateException</code>, this method must be
     * called prior to the <code>initialize()</code> method.  Also, keep in mind
     * that the <code>getSession</code> methods implicitly call the
     * <code>initialize()</code> method if it has yet to be called by the
     * application.
     *
     * @param ogAuthorization the <code>ObjectGridAuthorization</code> plugin
     *
     * @throws IllegalStateException if this method is called after the
     *         <code>initialize()</code> method is called.
     *
     * @see #initialize()
     * @see ObjectGridAuthorization
     * @since WAS XD 6.1
     */
    void setObjectGridAuthorization(ObjectGridAuthorization ogAuthorization);
```

ObjectGridAuthorization の実装

ObjectGridAuthorization インターフェースの boolean checkPermission(Subject subject, Permission permission) メソッドが WebSphere eXtreme Scale ランタイムによって呼び出されて、渡された Subject オブジェクトに、渡された許可があるかどうかが検査されます。オブジェクトに許可がある場合は、ObjectGridAuthorization インターフェースの実装によって true が返され、許可がない場合は false が返されます。

このプラグインの通常の実装は、Subject オブジェクトからプリンシパルを検索し、指定された許可が特定のポリシーを参照してプリンシパルに与えられているかどうかを確認することです。これらのポリシーは、ユーザーが定義します。例えば、ポリシーはデータベース、プレーン・ファイル、または Tivoli Access Manager で定義できます。

例えば、Tivoli Access Manager ポリシー・サーバーを使用して許可ポリシーを管理し、その API を使用してアクセスを許可できます。 Tivoli Access Manager

Authorization API の使用方法について詳しくは、「IBM Tivoli Access Manager Authorization Java Classes デベロッパーズ・リファレンス」を参照してください。

このサンプル実装では、以下を想定します。

1. MapPermission の許可だけをチェックします。他の許可については常に true を戻します。
2. Subject オブジェクトには、com.tivoli.mts.PDPrincipal プリンシパルが含まれます。
3. Tivoli Access Manager ポリシー・サーバーには、ObjectMap または JavaMap 名オブジェクトの以下の許可を定義しました。このポリシー・サーバーに定義されるオブジェクトの名前は、[ObjectGrid_name].[ObjectMap_name] 形式で、ObjectMap または JavaMap 名と同じにする必要があります。許可は、MapPermission 許可で定義される許可ストリングの先頭文字です。例えば、ポリシー・サーバーで定義される許可「r」は、ObjectMap マップに対する read 許可を表します。

以下は、checkPermission メソッドを実装する方法を示したコード断片です。

```
/**
* @see com.ibm.websphere.objectgrid.security.plugins.
*   MapAuthorization#checkPermission
* (javax.security.auth.Subject, com.ibm.websphere.objectgrid.security.
*   MapPermission)
*/
public boolean checkPermission(final Subject subject,
 Permission p) {

  // For non-MapPermission, we always authorize.
  if (!(p instanceof MapPermission)){
    return true;
  }

  MapPermission permission = (MapPermission) p;

  String[] str = permission.getParsedNames();

  StringBuffer pdPermissionStr = new StringBuffer(5);
  for (int i=0; i<str.length; i++) {
    pdPermissionStr.append(str[i].substring(0,1));
  }

  PDPermission pdPerm = new PDPermission(permission.getName(),
  pdPermissionStr.toString());

  Set principals = subject.getPrincipals();

  Iterator iter= principals.iterator();
  while(iter.hasNext()) {
    try {
      PDPrincipal principal = (PDPrincipal) iter.next();
      if (principal.implies(pdPerm)) {
        return true;
      }
    }
    catch (ClassCastException cce) {
      // Handle exception
    }
  }
  return false;
}
```

グリッド認証

セキュア・トークン・マネージャー・プラグインは、サーバー間認証に使用される新たなプラグインです。セキュア・トークン・マネージャー・プラグインは、com.ibm.websphere.objectgrid.security.plugins.SecureTokenManager インターフェースによって表されます。

generateToken(Object) メソッドは保護されるオブジェクトを取得し、外部に識別されないトークンを生成します。verifyTokens(byte[]) メソッドは逆に、トークンを元のオブジェクトに変換して戻します。

単純な SecureTokenManager 実装は XOR アルゴリズムなど単純なエンコード・アルゴリズムを使用して、オブジェクトをシリアライズ済みフォームでエンコードし、対応するデコード・アルゴリズムを使用してトークンをデコードします。この実装は保護されていないため、簡単に中断されます。

WebSphere eXtreme Scale デフォルト実装

WebSphere eXtreme Scale には、このインターフェース用のすぐに使用可能な実装が用意されています。このデフォルト実装は、鍵ペアを使用して署名し、署名を検査します。また、秘密鍵を使用してコンテンツを暗号化します。すべてのサーバーには JCKES タイプの鍵ストアが備えられており、鍵ペア、秘密鍵と公開鍵、および秘密鍵が保管されています。鍵ストアは、秘密鍵を保管する JCKES タイプである必要があります。これらの鍵は、送信側で秘密ストリングを暗号化し、署名または検証する場合に使用されます。また、トークンは有効期限の時間に関連付けられています。受信側で、データの検証、暗号化解除、および受信側の秘密ストリングとの比較が行われます。サーバーのペアの間での認証には、Secure Sockets Layer (SSL) 通信プロトコルは必要ありません。これは、秘密鍵と公開鍵の目的が同じであるためです。ただし、サーバー通信が暗号化されていない場合は、通信時に侵入者にデータを盗まれる可能性があります。トークンの有効期限が近いため、リプレイ・アタックの危険性は少なくなっています。この可能性は、すべてのサーバーをファイアウォールの後ろにデプロイすると、非常に小さくなります。

この方法の欠点は、WebSphere eXtreme Scale 管理者が鍵を生成し、生成した鍵をすべてのサーバーにトランスポートする必要があるため、トランスポート中にセキュリティー・ブリーチ (抜け穴) が発生する可能性があることです。

ローカル・セキュリティー

WebSphere eXtreme Scale によりいくつかのセキュリティー・エンドポイントが提供され、カスタム・メカニズムを統合できるようになります。ローカル・プログラミング・モデルにおける主なセキュリティー機能は許可で、認証サポートはありません。WebSphere Application Server の外側で認証を行う必要があります。ただし、Subject オブジェクトを取得および検証するプラグインは備えられています。

セキュリティーの使用可能化

以下に示すのは、ローカル・セキュリティーを使用可能にする 2 つの方法です。

- **XML 構成** ObjectGrid XML ファイルを使用して ObjectGrid を定義し、その ObjectGrid に対するセキュリティーを使用可能にできます。以下のファイルは

secure-objectgrid-definition.xml ファイルで、ObjectGridSample エンタープライズ・アプリケーション・サンプルで使用されます。この XML ファイルでは、セキュリティーは securityEnabled 属性を true に設定することによって使用可能になります。

```
<objectGrids>
    <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
        authorizationMechanism="AUTHORIZATION_MECHANISM_JASS">
    ...
</objectGrids>
```

- **プログラミング** API メソッド ObjectGrid.setSecurityEnabled() を使用して ObjectGrid を作成する場合は、ObjectGrid インターフェース上で以下のメソッドを呼び出して、セキュリティーを使用可能にします。

```
/**
 * Enable the ObjectGrid security
 */
void setSecurityEnabled();
```

認証

ローカル・プログラミング・モデルでは、eXtreme Scale は認証メカニズムを提供しておらず、認証に関して、アプリケーション・サーバーまたはアプリケーションのいずれかの環境に依存しています。 eXtreme Scale が WebSphere Application Server または WebSphere Extended Deployment で使用される場合、アプリケーションは WebSphere Application Server セキュリティー認証メカニズムを使用できます。eXtreme Scale が Java 2 Platform, Standard Edition (J2SE) 環境で稼働している場合、アプリケーションが Java 認証および承認サービス (JAAS) 認証またはその他の認証メカニズムを使用して認証を管理する必要があります。 JAAS 認証の使用方法については、「JAAS リファレンス・ガイド」を参照してください。アプリケーションと ObjectGrid インスタンスの契約には、javax.security.auth.Subject オブジェクトを使用します。クライアントがアプリケーション・サーバーまたはアプリケーションによって認証されると、アプリケーションは認証された javax.security.auth.Subject オブジェクトを検索し、この Subject オブジェクトを使用して ObjectGrid.getSession(Subject) メソッドを呼び出すことによって、ObjectGrid インスタンスからセッションを取得できます。この Subject オブジェクトを使用して、マップ・データへのアクセスを許可します。この契約は、サブジェクト引き渡し機構と呼ばれます。以下の例に、ObjectGrid.getSession(Subject) API を示します。

```
/**
 * This API allows the cache to use a specific subject rather than the one
 * configured on the ObjectGrid to get a session.
 * @param subject
 * @return An instance of Session
 * @throws ObjectGridException
 * @throws TransactionCallbackException
 * @throws InvalidSubjectException the subject passed in is not valid based
 * on the SubjectValidation mechanism.
 */
public Session getSession(Subject subject)
throws ObjectGridException, TransactionCallbackException, InvalidSubjectException;
```

ObjectGrid インターフェースの ObjectGrid.getSession() メソッドは、Session オブジェクトを取得するのに使用することもできます。

```
/**
 * This method returns a Session object that can be used by a single thread at a time.
 * You cannot share this Session object between threads without placing a
```

```
 * critical section around it. While the core framework allows the object to move
 * between threads, the TransactionCallback and Loader might prevent this usage,
 * especially in J2EE environments. When security is enabled, this method uses the
 * SubjectSource to get a Subject object.
 *
 * If the initialize method has not been invoked prior to the first
 * getSession invocation, then an implicit initialization occurs.  This
 * initialization ensures that all of the configuration is complete before
 * any runtime usage is required.
 *
 * @see #initialize()
 * @return An instance of Session
 * @throws ObjectGridException
 * @throws TransactionCallbackException
 * @throws IllegalStateException if this method is called after the
 *         destroy() method is called.
 */
public Session getSession()
throws ObjectGridException, TransactionCallbackException;
```

API 資料で指定されているように、セキュリティーが使用可能になると、このメソッドは SubjectSource プラグインを使用して Subject オブジェクトを取得します。SubjectSource プラグインは、Subject オブジェクトの伝搬をサポートするために eXtreme Scale で定義されるセキュリティー・プラグインの 1 つです。詳細情報については、『セキュリティー関連プラグイン』を参照してください。
getSession(Subject) メソッドは、ローカル ObjectGrid インスタンスでのみ呼び出すことができます。分散 eXtreme Scale 構成のクライアント・サイドで getSession(Subject) メソッドを呼び出すと、IllegalStateException が発生します。

セキュリティー・プラグイン

WebSphere eXtreme Scale は、サブジェクト引き渡し機構に関連する 2 つのセキュリティー・プラグイン、SubjectSource プラグインと SubjectValidation プラグインを提供します。

SubjectSource プラグイン

SubjectSource プラグインは、com.ibm.websphere.objectgrid.security.plugins.SubjectSource インターフェースによって表され、 eXtreme Scale を実行している環境から Subject オブジェクトを取得するために使用されます。この環境は、ObjectGrid を使用するアプリケーションや、アプリケーションをホストするアプリケーション・サーバーなどです。サブジェクト引き渡し機構の代わりとなる SubjectSource プラグインについて考えます。サブジェクト引き渡し機構を使用すると、アプリケーションは Subject オブジェクトを検索し、それを使用して ObjectGrid セッション・オブジェクトを取得します。
SubjectSource プラグインを使用すると、eXtreme Scale ランタイムは Subject オブジェクトを検索し、それを使用してセッション・オブジェクトを取得します。サブジェクト引き渡し機構は、アプリケーションに Subject オブジェクトの制御を与え、 SubjectSource プラグイン機構は Subject オブジェクトの検索からアプリケーションを解放します。 SubjectSource プラグインを使用すると、許可に使用できる eXtreme Scale クライアントを表す Subject オブジェクトを取得できます。
ObjectGrid.getSession メソッドが呼び出されると、Subject getSubject は、セキュリティーが使用可能な場合に、ObjectGridSecurityException をスローします。 WebSphere eXtreme Scale により、このプラグインのデフォルトの実装である com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectSourceImpl が提供されます。 この実装を使用すると、アプリケーションが WebSphere Application Server で稼働している場合に、スレッドから呼び出し元サブジェクトまたは RunAs サブジ

ェクトを検索することができます。 WebSphere Application Server で eXtreme Scale を使用している場合は、このクラスを ObjectGrid 記述子 XML ファイル内で SubjectSource 実装クラスとして構成することができます。以下に、WSSubjectSourceImpl.getSubject メソッドの主なフローを示すコード・スニペットを示します。

```
Subject s = null;
try {
  if (finalType == RUN_AS_SUBJECT) {
    // get the RunAs subject
    s = com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();
  }
  else if (finalType == CALLER_SUBJECT) {
    // get the callersubject
    s = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();
  }
}
catch (WSSecurityException wse) {
  throw new ObjectGridSecurityException(wse);
}

return s;
```

その他の詳細については、SubjectSource プラグインおよび WSSubjectSourceImpl 実装に関する API 資料を参照してください。

SubjectValidation プラグイン

com.ibm.websphere.objectgrid.security.plugins.SubjectValidation インターフェースで表される SubjectValidation プラグインは、別のセキュリティー・プラグインです。SubjectValidation プラグインを使用すると、ObjectGrid に渡されるか、または SubjectSource プラグインによって検索される javax.security.auth.Subject が、改ざんされていない有効な Subject であることを検証できます。

SubjectValidation インターフェースの SubjectValidation.validateSubject(Subject) メソッドにより、Subject オブジェクトが取得されて返されます。Subject オブジェクトが有効とみなされるかどうか、および戻される Subject オブジェクトは、すべて実装によって決定されます。 Subject オブジェクトが無効の場合は、InvalidSubjectException になります。

このプラグインは、このメソッドに渡される Subject オブジェクトを信頼できない場合に使用できます。 Subject オブジェクトを検索するコードを作成するアプリケーション開発者は信頼できると考えられるためこれは稀なケースです。

Subject オブジェクトが改ざんされたかどうかは作成者のみが知っているため、このプラグインの実装は、Subject オブジェクト作成者からのサポートが必要です。ただし、サブジェクトの作成者が、Subject が改ざんされたかどうかを関知していない場合もあります。その場合、このプラグインの使用はお勧めできません。

WebSphere eXtreme Scale は、SubjectValidation のデフォルトの実装、com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectValidationImpl を提供します。この実装を使用して、WebSphere Application Server で認証済みのサブジェクトの妥当性検査を行うことができます。WebSphere Application Server で eXtreme Scale を使用している場合は、このクラスを SubjectValidation 実装クラスとして構成することができます。 WSSubjectValidationImpl 実装は、この Subject オブジェク

トに関連付けられているクレデンシャル・トークンが改ざんされていない場合にのみ、この Subject オブジェクトを有効であるとみなします。 Subject オブジェクトの他のパーツを変更できます。 WSSubjectValidationImpl 実装は、WebSphere Application Server にクレデンシャル・トークンに一致するオリジナルの Subject を依頼し、そのオリジナルの Subject オブジェクトを検証済みの Subject オブジェクトとして戻します。このため、クレデンシャル・トークン以外の Subject コンテンツに加えられた変更は、無効になります。以下のコード・スニペットは、WSSubjectValidationImpl.validateSubject(Subject) の基本フローを示します。

```
// Create a LoginContext with scheme WSLogin and
// pass a Callback handler.
LoginContext lc = new LoginContext("WSLogin",
new WSCredTokenCallbackHandlerImpl(subject));

// When this method is called, the callback handler methods
// will be called to log the user in.
lc.login();

// Get the subject from the LoginContext
return lc.getSubject();
```

このコード・スニペットでは、クレデンシャル・トークンのコールバック・ハンドラー・オブジェクトである WSCredTokenCallbackHandlerImpl は、検証対象である Subject オブジェクトとともに作成されます。次に、LoginContext オブジェクトがログイン・スキーム WSLogin とともに作成されます。 lc.login メソッドが呼び出されると、WebSphere Application Server セキュリティーは Subject オブジェクトからクレデンシャル・トークンを検索し、その後、検証済みの Subject オブジェクトとして対応する Subject を戻します。

その他の詳細については、SubjectValidation および WSSubjectValidationImpl 実装の Java API を参照してください。

プラグイン構成

SubjectValidation プラグインおよび SubjectSource プラグインは、以下の 2 つの方法で構成できます。

- **XML 構成** ObjectGrid XML ファイルを使用して ObjectGrid を定義し、これら 2 つのプラグインを設定します。以下に例を示します。この例では、WSSubjectSourceImpl クラスが SubjectSource プラグインとして構成され、WSSubjectValidation クラスが SubjectValidation プラグインとして構成されます。

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS">
        <bean id="SubjectSource"
classNameName="com.ibm.websphere.objectgrid.security.plugins.builtins.
 WSSubjectSourceImpl" />
    <bean id="SubjectValidation"
 classNameName="com.ibm.websphere.objectgrid.security.plugins.builtins.
 WSSubjectValidationImpl" />
    <bean id="TransactionCallback"
classNameName="com.ibm.websphere.samples.objectgrid.
 HeapTransactionCallback" />
...
</objectGrids>
```

- **プログラミング** API を通して ObjectGrid を作成する場合は、以下のメソッドを呼び出して、SubjectSource または SubjectValidation プラグインを設定できます。

```
**
* Set the SubjectValidation plug-in for this ObjectGrid instance. A
* SubjectValidation plug-in can be used to validate the Subject object
* passed in as a valid Subject. Refer to {@link SubjectValidation}
* for more details.
* @param subjectValidation the SubjectValidation plug-in
*/
void setSubjectValidation(SubjectValidation subjectValidation);


/**
* Set the SubjectSource plug-in. A SubjectSource plug-in can be used
* to get a Subject object from the environment to represent the
* ObjectGrid client.
*
* @param source the SubjectSource plug-in
*/
void setSubjectSource(SubjectSource source);
```

独自の JAAS 認証コードを作成

独自の Java 認証および承認サービス (JAAS) 認証コードを作成して、認証を処理できます。独自のログイン・モジュールを作成し、認証モジュール用のログイン・モジュールを構成する必要があります。

ログイン・モジュールは、ユーザーに関する情報を受け取り、ユーザーを認証します。この情報は、ユーザーの識別に使用可能な何らかのものです。例えば、情報はユーザー ID およびパスワード、クライアント証明書などです。情報を受け取ると、ログイン・モジュールにより情報が有効なサブジェクトを表示していることが検証され、次に Subject オブジェクトが作成されます。現在、ログイン・モジュールのいくつかの使用可能な実装が公開されています。

ログイン・モジュールの作成後、このログイン・モジュールをランタイムが使用できるように構成します。JAAS ログイン・モジュールを構成する必要があります。このログイン・モジュールには、ログイン・モジュールおよびその認証スキームが含まれています。以下に例を示します。

```
FileLogin
{
    com.acme.auth.FileLoginModule required
};
```

認証スキームは FileLogin であり、ログイン・モジュールは com.acme.auth.FileLoginModule です。必須のトークンは、FileLoginModule モジュールがこのログインを検証する必要があることを示すか、またはスキーム全体が失敗したことを示します。

JAAS ログイン・モジュール構成ファイルの設定は、以下のいずれか 1 つの方法で実行できます。

- JAAS ログイン・モジュール構成ファイルを、以下の例のように、java.security ファイルの login.config.url プロパティーに設定します。

  ```
  login.config.url.1=file:${java.home}/lib/security/file.login
  ```

- **-Djava.security.auth.login.config** Java 仮想マシン (JVM) 引数を使用して、以下のように、コマンド行から JAAS ログイン・モジュール構成ファイルを設定します。 `-Djava.security.auth.login.config ==$JAVA_HOME/lib/security/file.login`

コードが WebSphere Application Server 上で実行されている場合、管理コンソールで JAAS ログインを構成し、このログイン構成をアプリケーション・サーバー構成に格納する必要があります。詳細については、『Java 認証および承認サービスのログイン構成』を参照してください。

第 7 章 管理 API を使用した組み込み eXtreme Scale コンテナー・サーバーの始動

WebSphere eXtreme Scale では、組み込みサーバーおよびコンテナーのライフサイクルの管理にプログラマチック API を使用できます。コマンド行オプションやファイル・ベースのサーバー・プロパティーでも構成可能な任意のオプションを使用して、プログラムでサーバーを構成できます。コンテナー・サーバー、カタログ・サービス、またはその両方として、組み込みサーバーの構成が可能です。

始める前に

既存の Java 仮想マシン内からコードを実行するためのメソッドが必要です。eXtreme Scale クラスが、クラス・ローダー・ツリーから利用可能でなければなりません。

このタスクについて

管理 API を使用して多くの管理タスクを実行できます。API の一般的な使用法の 1 つとして、Web アプリケーションの状態を保管する内部サーバーとしての使用があります。 Web サーバーは、組み込み WebSphere eXtreme Scale サーバーを始動し、コンテナー・サーバーをカタログ・サービスに報告することができ、サーバーは、より幅広い分散グリッドのメンバーとして追加されます。この使用法では、本来は揮発性のデータ・ストアにスケーラビリティーと高可用性が提供されます。

組み込み eXtreme Scale サーバーの全ライフサイクルをプログラムで制御できます。例は、できる限り汎用的にして、概要を説明したステップの直接的なコードの例のみを示しています。

1. ServerFactory クラスから ServerProperties オブジェクトを取得し、必要なオプションを構成します。

   すべての eXtreme Scale サーバーに、一連の構成可能なプロパティーがあります。コマンド行からサーバーが始動されると、それらのプロパティーはデフォルトに設定されますが、外部ソースまたはファイルを指定することによって、複数のプロパティーをオーバーライドすることができます。組み込み有効範囲では、ServerProperties オブジェクトでプロパティーを直接設定できます。これらのプロパティーは、ServerFactory クラスからサーバー・インスタンスを取得する前に設定する必要があります。以下の例のスニペットでは、ServerProperties オブジェクトを取得して CatalogServiceBootStrap フィールドを設定し、複数のオプション・サーバー設定を初期化します。構成可能な設定のリストについては、API 資料を参照してください。

```
ServerProperties props = ServerFactory.getServerProperties();
props.setCatalogServiceBootstrap("host:port"); // required to connect to specific catalog service
props.setServerName("ServerOne"); // name server
props.setTraceSpecification("com.ibm.ws.objectgrid=all=enabled"); // Sets trace spec
```

2. サーバーをカタログ・サービスにする場合には、CatalogServerProperties オブジェクトを取得します。

すべての組み込みサーバーが、カタログ・サービスまたはコンテナー・サーバー、あるいはその両方になることができます。以下の例では、CatalogServerProperties オブジェクトを取得し、カタログ・サービス・オプションを使用可能にし、さまざまなカタログ・サービス設定を構成します。

```
CatalogServerProperties catalogProps = ServerFactory.getCatalogProperties();
catalogProps.setCatalogServer(true); // false by default, it is required to set as a catalog service
catalogProps.setQuorum(true); // enables / disables quorum
```

3. ServerFactory クラスから Server インスタンスを取得します。 Server インスタンスは、グリッド内のメンバーシップの管理に関与するプロセス・スコープの singleton です。このインスタンスが初期化された後、このプロセスが接続され、グリッド内の他のサーバーで高度に利用可能になります。 以下の例は、Server インスタンスの作成方法を示しています。

   ```
   Server server = ServerFactory.getInstance();
   ```

   上記の例において、ServerFactory クラスは、 Server インスタンスを返す静的メソッドを提供します。 ServerFactory クラスは、Server インスタンスを取得するための唯一のインターフェースとして意図されています。そのため、このクラスは必ず、インスタンスが singleton であるか、または各 JVM または独立したクラス・ローダーの 1 つインスタンスであるようにします。 getInstance メソッドで Server インスタンスを初期化します。インスタンスを初期化する前にすべてのサーバー・プロパティーの構成が必要です。 Server クラスは、新規の Container インスタンスの作成に関与します。 ServerFactory クラスと Server クラスの両方を使用して、組み込み Server インスタンスのライフサイクルを管理できます。

4. Server インスタンスを使用して Container インスタンスを開始します。

   断片を組み込みサーバーに配置するには、その前に、サーバーにコンテナーを作成する必要があります。 Server インターフェースの createContainer メソッドは、DeploymentPolicy 引数を使用します。以下の例では、取得したサーバー・インスタンスを使用して、作成した DeploymentPolicy ファイルでコンテナーを作成します。 Container は、シリアライゼーションのためにアプリケーション・バイナリーが使用可能になっているクラス・ローダーを必要とします。これらのバイナリーは、使用するクラス・ローダーを Thread コンテキスト・クラス・ローダーに設定して createContainer メソッドを呼び出すことによって、使用可能にできます。

```
DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy(new URL("file://urltodeployment.xml"),
 new URL("file://urltoobjectgrid.xml"));
Container container = server.createContainer(policy);
```

5. コンテナーを除去してクリーンアップします。

   コンテナー・サーバーを除去してクリーンアップするには、取得した Container インスタンスで teardown メソッドを実行します。コンテナーで teardown メソッドを実行すると、コンテナーを適切にクリーンアップし、組み込みサーバーからコンテナーを除去します。

   コンテナーのクリーンアップ処理には、そのコンテナー内のすべての断片の移動と終了処理が含まれます。各サーバーには、多くのコンテナーと断片が含まれます。コンテナーをクリーンアップしても、親の Server インスタンスのライフサイクルには影響しません。以下の例は、サーバーで teardown メソッドを実行する方法を示しています。teardown メソッドは、ContainerMBean インターフェー

スを通して使用可能になります。ContainerMBean インターフェースを使用することによって、このコンテナーに対するプログラムによるアクセスがもうなくても、その MBean でコンテナーを除去してクリーンアップすることができます。また、terminate メソッドが Container インターフェースに存在しますが、どうしても必要でない限り、このメソッドは使用しないでください。このメソッドは強制力が強く、断片の適切な移動とクリーンアップの調整は行いません。

```
container.teardown();
```

6. 組み込みサーバーを停止します。

   組み込みサーバーを停止するときには、そのサーバーで実行されているコンテナーと断片も停止します。組み込みサーバーの停止時には、開いているすべての接続をクリーンアップして、すべての断片を移動または終了処理する必要があります。以下の例では、サーバーの停止方法と、Server インターフェースで waitFor メソッドを使用して Server インスタンスが確実に完全にシャットダウンするようにする方法を示しています。コンテナーの例と同様に、stopServer メソッドは、ServerMBean インターフェースを通して使用可能になります。このインターフェースでは、該当の Managed Bean (MBean) によりサーバーを停止できます。

```
ServerFactory.stopServer();  // Uses the factory to kill the Server singleton
 // or
server.stopServer(); // Uses the Server instance directly
server.waitFor(); // Returns when the server has properly completed its shutdown procedures
```

全コードの例:

```
 import java.net.MalformedURLException;
import java.net.URL;

import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.deployment.DeploymentPolicy;
import com.ibm.websphere.objectgrid.deployment.DeploymentPolicyFactory;
import com.ibm.websphere.objectgrid.server.Container;
import com.ibm.websphere.objectgrid.server.Server;
import com.ibm.websphere.objectgrid.server.ServerFactory;
import com.ibm.websphere.objectgrid.server.ServerProperties;

public class ServerFactoryTest {

    public static void main(String[] args) {

        try {

            ServerProperties props = ServerFactory.getServerProperties();
            props.setCatalogServiceBootstrap("catalogservice-hostname:catalogservice-port");
            props.setServerName("ServerOne"); // name server
            props.setTraceSpecification("com.ibm.ws.objectgrid=all=enabled"); // TraceSpec

            /*
             * In most cases, the server will serve as a container server only
             * and will connect to an external catalog service. This is a more
             * highly available way of doing things. The commented code excerpt
             * below will enable this Server to be a catalog service.
             *
             *
             * CatalogServerProperties catalogProps =
             * ServerFactory.getCatalogProperties();
             * catalogProps.setCatalogServer(true); // enable catalog service
             * catalogProps.setQuorum(true); // enable quorum
             */

            Server server = ServerFactory.getInstance();

            DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy(new URL("url to deployment xml"),
                            new URL("url to objectgrid xml file"));
            Container container = server.createContainer(policy);

            /*
             * Shard will now be placed on this container if the deployment requirements are met.
             * This encompasses embedded server and container creation.
             *
             * The lines below will simply deomnstrate calling the cleanup methods
             */
```

```
            container.teardown();
            server.stopServer();
            int success = server.waitFor();

        } catch (ObjectGridException e) {
            // Container failed to initialize
        } catch (MalformedURLException e2) {
            // invalid url to xml file(s)
        }

    }

}
```

第 8 章 パフォーマンスの考慮事項

メモリー内データ・グリッドまたはデータベース処理スペースのパフォーマンスを向上させるため、いくつかの考慮事項を調べることができます。考慮事項には、Java 仮想マシン設定を調整することや、ロック、シリアライゼーション、照会の実行などの製品フィーチャーに関するベスト・プラクティスを使用することなどがあります。

JVM の調整

このページでは、WebSphere eXtreme Scale 用の Java 仮想マシン (JVM) 調整に関するいくつかの具体的な問題を示します。

4 コア当たり 1 JVM で 1 から 2Gb のヒープをお勧めします。ヒープ・サイズは、この資料の後で説明する、サーバーに保管されるオブジェクトの特性に依存します。

ヒープ・サイズおよびガーベッジ・コレクションの推奨事項

最適なヒープ・サイズ値は、次の 3 つの要因に基づきます。

1. ヒープ内のライブ・オブジェクトの数。
2. ヒープ内のライブ・オブジェクトの複雑さ。
3. JVM 用に使用可能なコアの数。

例えば、10K バイトの配列を保管するアプリケーションは、POJO の複雑なグラフを使用するアプリケーションよりもずっと大きなヒープを実行できます。

最近の JVM はすべてパラレル・ガーベッジ・コレクションのアルゴリズムを使用していますが、これは、より多くのコアを使用するとガーベッジ・コレクション中の停止を削減できることを意味します。したがって、8 コアのコンピューターのほうが、4 コアのものよりも速く収集されます。

実メモリー使用量とヒープ仕様

1Gb ヒープ JVM は、約 1.3Gb の実メモリーを使用します。テストでは、16Gb の RAM のコンピューターでは、10 個の 1Gb JVM を実行できませんでした。 JVM ヒープが 800 MB を超えると、ページングが始まります。

ガーベッジ・コレクション

IBM JVM の場合、更新率が高いシナリオ (トランザクションの 100% がエントリーを変更する) には avgoptpause コレクターを使用してください。データがほとんど更新されない (10% 以下の頻度) ようなシナリオでは、avgoptpause コレクターより gencon コレクターの方がより適切に機能します。両方のコレクターを使用して実験を行い、シナリオで最も適切に機能するコレクターを確認します。パフォーマンス上の問題を確認できる場合は、詳細ガーベッジ・コレクションをオンにして実行し、ガーベッジの収集に費やされている時間の割合を確認します。調整で問題が

修正されるまでガーベッジ・コレクションで時間の 80% が費やされたシナリオが発生しました。

JVM パフォーマンス

WebSphere eXtreme Scale は、異なるバージョンの Java 2 Platform, Standard Edition (J2SE) 上で実行できます。ObjectGrid バージョン 6.1 は J2SE バージョン 1.4.2 以降をサポートしています。開発者の生産性およびパフォーマンスを向上させるためには、J2SE 5 またはそれ以降を使用して、アノテーションおよび改良されたガーベッジ・コレクションを活用してください。ObjectGrid は、32 ビットまたは 64 ビット JVM 上で動作します。

ObjectGrid バージョン 6.0.2 クライアントは、ObjectGrid バージョン 6.1 グリッドに接続できます。J2SE バージョン 1.4.2 または良好なクライアントに対しては、ObjectGrid バージョン 6.1 クライアントを使用します。ObjectGrid バージョン 6.0.2 クライアントを使用する唯一の理由は、J2SE バージョン 1.3 サポート用であるためです。

WebSphere eXtreme Scale がテストされたのは、使用可能な仮想マシンの一部ですが、サポートのリストは排他的なものではありません。バージョン 1.4.2 以上の任意のバージョンで WebSphere eXtreme Scale を実行できますが、JVM に関する問題が特定されている場合は、JVM ベンダーにサポートを依頼する必要があります。可能であれば、 WebSphere Application Server がサポートするどのプラットフォーム上でも、WebSphere ランタイムの JVM を使用してください。

最良の JVM は Java Platform, Standard Edition 6 です。Java 2 Platform, Standard Edition v 1.4 は、gencon コレクターが大きく影響するようなシナリオの場合は特に、パフォーマンスが悪くなります。Java Platform Standard Edition 5 は良好に動作しますが、Java Platform, Standard Edition 6 のほうが優れています。

orb.properties の調整

実動には以下の orb.properties ファイルを使用することをお勧めします。このファイルは、1500 JVM までのグリッドでの使用についてテスト済みです。orb.properties ファイルは、使用される JRE の lib フォルダー内にあります。

```
# IBM JDK properties for ORB
org.omg.CORBA.ORBClass=com.ibm.CORBA.iiop.ORB
org.omg.CORBA.ORBSingletonClass=com.ibm.rmi.corba.ORBSingleton

# WS Interceptors
org.omg.PortableInterceptor.ORBInitializerClass.com.ibm.ws.objectgrid.corba.ObjectGridInitializer

# WS ORB & Plugins properties
com.ibm.CORBA.ForceTunnel=never
com.ibm.CORBA.RequestTimeout=10
com.ibm.CORBA.ConnectTimeout=10

# Needed when lots of JVMs connect to the catalog at the same time
com.ibm.CORBA.ServerSocketQueueDepth=2048

# Clients and the catalog server can have sockets open to all JVMs
com.ibm.CORBA.MaxOpenConnections=1016

# Thread Pool for handling incoming requests, 200 threads here
com.ibm.CORBA.ThreadPool.IsGrowable=false
com.ibm.CORBA.ThreadPool.MaximumSize=200
com.ibm.CORBA.ThreadPool.MinimumSize=200
com.ibm.CORBA.ThreadPool.InactivityTimeout=180000

# No splitting up large requests/responses in to smaller chunks
com.ibm.CORBA.FragmentSize=0
```

スレッド数

スレッド数はいくつかの要因に依存します。単一の断片が管理できるスレッド数には制限があります。JVM ごとの断片数が多いほど、より多くのスレッドおよび並行性が存在できます。追加の各断片により、データへの並行性パスが提供されます。各断片はできるだけ並行ですが、それでも制限はあります。

CopyMode のベスト・プラクティス

WebSphere eXtreme Scale は、CopyMode 設定に基づいて値をコピーします。

BackingMap API setCopyMode(CopyMode, valueInterfaceClass) メソッドを使用して、com.ibm.websphere.objectgrid.CopyMode クラスで定義される、次の最終の静的フィールドの 1 つに、コピー・モードを設定することができます。

アプリケーションが WebSphere eXtreme Scale インターフェースを使用してマップ・エントリーに対する参照を取得する場合、その参照は、それを取得した ObjectGrid トランザクション内でのみ使用してください。別のトランザクションでその参照を使用するとエラーになることがあります。例えば BackingMap に対してペシミスティック・ロック・ストラテジーを使用する場合は、get メソッド呼び出しまたは getForUpdate メソッド呼び出しにより、トランザクションに応じて S (shared) ロックまたは U (update) ロックを取得します。トランザクションの終了時に get メソッドは値に参照を戻し、取得されているロックは解放されます。トランザクションは get メソッドまたは getForUpdate メソッドを呼び出して、別のトランザクションでマップ・エントリーをロックする必要があります。各トランザクションは、複数のトランザクションで同じ値参照を再利用する代わりに get メソッドまたは getForUpdate メソッドを呼び出すことにより、値への独自の参照を取得する必要があります。

エンティティー・マップに対する CopyMode

EntityManager API エンティティーと関連付けられたマップを使用する場合、そのマップは常にエンティティー Tuple オブジェクトを直接戻し、 COPY_TO_BYTES コピー・モードが使用されていない限り、コピーは作成しません。変更を行う場合、CopyMode が更新される、または、Tuple が適切にコピーされることが重要です。

COPY_ON_READ_AND_COMMIT

COPY_ON_READ_AND_COMMIT モードはデフォルトのモードです。このモードが使用される場合、valueInterfaceClass 引数は無視されます。このモードは、BackingMap に含まれている値オブジェクトへの参照がアプリケーションに含まれていないことを保証します。その代わりに、アプリケーションは常に BackingMap 内の値のコピーを操作します。COPY_ON_READ_AND_COMMIT モードでは、BackingMap にキャッシュされているデータをアプリケーションが誤って壊してしまうことはありません。アプリケーションのトランザクションが指定されたキーの ObjectMap.get メソッドを呼び出し、それがそのキーにとって、ObjectMap エントリーへの初めてのアクセスの場合は、値のコピーが戻されます。トランザクションがコミットされると、アプリケーションによってコミットされたすべての変更は BackingMap にコピーされ、BackingMap にコミットされた値への参照をアプリケーションが持つことはありません。

COPY_ON_READ

COPY_ON_READ モードは、トランザクションがコミットされたときに発生するコピーを除去することによって、COPY_ON_READ_AND_COMMIT モード全体にわたるパフォーマンスを改善します。このモードが使用される場合、valueInterfaceClass 引数は無視されます。 BackingMap データの整合性を保持するために、アプリケーションは、エントリーに対する各参照がトランザクションのコミット後に破棄されることを保証します。このモードでは、ObjectMap.get メソッドは、値への参照の代わりに値のコピーを返し、アプリケーションがその値に対して行った変更が、トランザクションがコミットされるまで BackingMap 値に影響しないことを保証します。ただし、トランザクションがコミットすると変更のコピーは行われません。代わりに、ObjectMap.get メソッドによって戻された、コピーへの参照が BackingMap に保管されます。トランザクションがコミットされた後、アプリケーションはすべてのマップ・エントリー参照を破棄します。アプリケーションがマップ・エントリー参照を破棄しなかった場合、そのアプリケーションは、BackingMap 内にキャッシュされているデータを破壊してしまうことがあります。アプリケーションがこのモードを使用し、問題がある場合は、COPY_ON_READ_AND_COMMIT モードに切り替えてその問題がまだ続いているかどうかを調べます。問題が解消されている場合は、トランザクションがコミットされた後でアプリケーションはその参照のすべてを破棄するのに失敗したことになります。

COPY_ON_WRITE

COPY_ON_WRITE モードは、指定したキーのトランザクションによって ObjectMap.get メソッドが初めて呼び出されるときに起こるコピーを排除することにより、COPY_ON_READ_AND_COMMIT モードを超えるパフォーマンスを実現します。 ObjectMap.get メソッドは、値オブジェクトへの直接参照の代わりに値のプロキシーを戻します。プロキシーは、アプリケーションが valueInterfaceClass 引数によって指定した値インターフェース上で set メソッドを呼び出さない限り、値のコピーが行われないことを保証します。プロキシーは、copy on write インプリメンテーションを提供します。トランザクションがコミットすると、BackingMap はプロキシーを検査して、呼び出される set メソッドの結果としてコピーが行われたかどうかを判別します。コピーが行われた場合は、そのコピーへの参照が BackingMap に保管されます。このモードの大きな利点は、トランザクションが値を変更するために set メソッドを呼び出さない場合には、読み取りまたはコミットの時点で値がコピーされないことです。

COPY_ON_READ_AND_COMMIT および COPY_ON_READ モードはどちらも、値が ObjectMap から検索される場合にディープ・コピーを行います。アプリケーションがトランザクションで検索されたいくつかの値を更新するだけの場合は、このモードは最適ではありません。 COPY_ON_WRITE モードはこの振る舞いを効率的な方法でサポートしますが、アプリケーションがシンプルなパターンを使用する必要があります。インターフェースをサポートするには、値オブジェクトが必要です。アプリケーションは、eXtreme Scale セッション内で値と対話するときに、このインターフェースのメソッドを使用する必要があります。その場合、eXtreme Scale はアプリケーションに戻される値のプロキシーを作成します。プロキシーは実際の値になる参照を持ちます。アプリケーションが読み取り操作のみを実行した場合、その読み取り操作は常に実際のコピーに対して実行されます。アプリケーションがオブジェクト上の属性を変更する場合、プロキシーは実際のオブジェクトをコピーし

て、それからそのコピーに対して変更を行います。プロキシーは次に、そのポイントからコピーを使用します。このコピーを使用することにより、アプリケーションによって読み取られるだけのオブジェクトに対するコピー操作は完全に避けることができます。すべての変更操作は設定されたプレフィックスで開始する必要があります。 Enterprise JavaBeans は通常、オブジェクト属性を変更するメソッドに対してこのスタイルのメソッドの名前付けを使用するためにコード化されます。この規則に従わなければいけません。変更されたすべてのオブジェクトは、アプリケーションによって変更されるときにコピーされます。この読み取りと書き込みのシナリオは、eXtreme Scale がサポートしている、最も効率的なシナリオです。
COPY_ON_WRITE モードを使用するようマップを構成するには、以下の例を使用してください。この例では、アプリケーションは、Map 内の名前を使用してキーが付けられている Person オブジェクトを保管します。Person オブジェクトは以下のコード・スニペットで表されます。

```
class Person {
    String name;
    int age;
    public Person() {
    }
    public void setName(String n)  {
        name = n;
    }
    public String getName() {
        return name;
    }
    public void setAge(int a) {
        age = a;
    }
    public int getAge() {
        return age;
    }
}
```

アプリケーションは、ObjectMap から取り出された値と対話する場合にのみ IPerson インターフェースを使用します。次の例のようにオブジェクトを変更してインターフェースを使用します。

```
interface IPerson
{
    void setName(String n);
    String getName();
    void setAge(int a);
    int getAge();
}
// Modify Person to implement IPerson interface
class Person implements IPerson {
    ...
}
```

それからアプリケーションは、次の例のように、COPY_ON_WRITE モードを使用するために BackingMap を構成する必要があります。

```
ObjectGrid dg = ...;
BackingMap bm = dg.defineMap("PERSON");
// use COPY_ON_WRITE for this Map with
// IPerson as the valueProxyInfo Class
bm.setCopyMode(CopyMode.COPY_ON_WRITE,IPerson.class);
// The application should then use the following
// pattern when using the PERSON Map.
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
```

```
...
sess.begin();
// the application casts the returned value to IPerson and not Person
IPerson p = (IPerson)person.get("Billy");
p.setAge(p.getAge()+1);
...
// make a new Person and add to Map
Person p1 = new Person();
p1.setName("Bobby");
p1.setAge(12);
person.insert(p1.getName(), p1);
sess.commit();
// the following snippet WON'T WORK. Will result in ClassCastException
sess.begin();
// the mistake here is that Person is used rather than
// IPerson
Person a = (Person)person.get("Bobby");
sess.commit();
```

最初のセクションはマップ内で Billy と名前を付けられた値を検索するアプリケーションを示しています。このアプリケーションは、戻り値を Person オブジェクトではなく、IPerson オブジェクトにキャストします。その理由は、返されたプロキシーは以下の 2 つのインターフェースを実装しているからです。

- BackingMap.setCopyMode メソッド呼び出しで指定されたインターフェース
- com.ibm.websphere.objectgrid.ValueProxyInfo インターフェース

プロキシーを 2 つのタイプにキャストすることができます。先ほどのコード・スニペットの最後の部分は、COPY_ON_WRITE モードでは許可されないことを示しています。このアプリケーションは Bobby レコードを取り出して、そのレコードを Person オブジェクトにキャストしようとします。このアクションはクラス・キャスト例外により失敗します。戻されるプロキシーが Person オブジェクトではないからです。戻されたプロキシーは IPerson オブジェクトと ValueProxyInfo を実装します。

ValueProxyInfo インターフェースおよび部分更新サポート: このインターフェースはアプリケーションに対して、プロキシーによって参照される、コミットされた読み取り専用の値か、またはこのトランザクション中に変更された属性セットのどちらかの検索を許可します。

```
public interface ValueProxyInfo {
    List /**/ ibmGetDirtyAttributes();
    Object ibmGetRealValue();
}
```

ibmGetRealValue メソッドは、オブジェクトの読み取り専用のコピーを戻します。アプリケーションはこの値を変更してはいけません。ibmGetDirtyAttributes メソッドは、このトランザクション中にアプリケーションによって変更された属性を示すストリングのリストを戻します。 ibmGetDirtyAttributes は主に、Java database connectivity (JDBC) または CMP ベースのローダーで使用されます。リストに指定された属性だけを、SQL ステートメントまたはテーブルにマップされたオブジェクト上で更新する必要があります。これにより、Loader により生成される、さらに効率的な SQL が可能です。 copy on write トランザクションがコミットされ、ローダーが接続されると、ローダーは変更されたオブジェクトの値を ValueProxyInfo インターフェースにキャストしてこの情報を取得することができます。

COPY_ON_WRITE またはプロキシーを使用する場合の equals メソッドの処理: 例えば、次のコードは Person オブジェクトを構成してから、それを ObjectMap に挿入します。次に、ObjectMap.get メソッドを使用して同じオブジェクトを取り出します。値はインターフェースにキャストされます。値が Person インターフェースにキャストされる場合は、ClassCastException 例外が起こります。戻り値が、Person オブジェクトではなく、IPerson インターフェースをインプリメントするプロキシーだからです。== 操作を使用する場合は、等価チェックが失敗します。これらは同じオブジェクトではないからです。

```
session.begin();
// new the Person object
Person p = new Person(...);
personMap.insert(p.getName, p);
// retrieve it again, remember to use the interface for the cast
IPerson p2 = personMap.get(p.getName());
if(p2 == p) {
    // they are the same
} else {
    // they are not
}
```

equals メソッドをオーバーライドする必要がある場合は、ほかにも考慮しなければならないことがあります。次のコード・スニペットに示すように、equals メソッドは、引数が IPerson インターフェースをインプリメントし、その引数をキャストして IPerson にするオブジェクトであることを検証する必要があります。引数が、IPerson インターフェースをインプリメントするプロキシーかもしれないので、インスタンス変数が等しいかどうかを比較するときに getAge メソッドと getName メソッドを使用する必要があります。

```
{
    if ( obj == null ) return false;
    if ( obj instanceof IPerson ) {
        IPerson x = (IPerson) obj;
        return ( age.equals( x.getAge() ) && name.equals( x.getName() ) )
    }
    return false;
}
```

ObjectQuery および HashIndex 構成の要件: COPY_ON_WRITE を ObjectQuery または HashIndex プラグインと共に使用する場合、プロパティー・メソッドを使用してオブジェクトにアクセスするように ObjectQuery スキーマおよび HashIndex プラグインを構成する (これがデフォルトです) ことが重要です。フィールド・アクセスを使用するように構成されると、照会エンジンおよび索引は、プロキシー・オブジェクト内のフィールドにアクセスしようとし、その場合、オブジェクト・インスタンスがプロキシーになるため、常にヌルまたは 0 が返されます。

NO_COPY

NO_COPY によって、アプリケーションは、ObjectMap.get メソッドを使用して取得した値オブジェクトを、パフォーマンス向上と交換に変更しないことを保証できます。このモードが使用される場合、valueInterfaceClass 引数は無視されます。このモードを使用する場合は、値がコピーされることはありません。アプリケーションが値を変更すると、BackingMap 内のデータが壊れます。 NO_COPY モードは基本的に、アプリケーションによってデータが変更されることのない、読み取り専用マップで有用です。アプリケーションがこのモードを使用し、問題がある場合は、COPY_ON_READ_AND_COMMIT モードに切り替えてその問題がまだ存在するかど

うかを調べます。問題が解消されている場合は、トランザクション中またはトランザクションがコミットされた後でアプリケーションは ObjectMap.get メソッドによって戻された値を変更しています。 EntityManager API エンティティーに関連付けられたすべてのマップは、eXtreme Scale 構成の指定にかかわらず、自動的にこのモードを使用します。

EntityManager API エンティティーに関連付けられたすべてのマップは、eXtreme Scale 構成の指定にかかわらず、自動的にこのモードを使用します。

COPY_TO_BYTES

POJO 形式の代わりに、シリアライズ形式でオブジェクトを保管できます。COPY_TO_BYTES 設定を使用すると、大きなオブジェクト・グラフが消費するメモリー占有スペースを削減できます。追加情報については、『バイト配列マップ』を参照してください。

CopyMode の不正な使用

上記で説明したように、アプリケーションが COPY_ON_READ、COPY_ON_WRITE、または NO_COPY コピー・モードを使用してパフォーマンスを改善しようとすると、エラーが発生します。コピー・モードを COPY_ON_READ_AND_COMMIT モードに変更する際には偶発的なエラーは発生しません。

問題

この問題は、使用したコピー・モードのプログラミング契約にアプリケーションが違反し、その結果発生した ObjectGrid マップ内のデータ破壊に起因する場合があります。データ破壊は、予測不能なエラーが、偶発的または解明不能または予期しない形で発生する原因になることがあります。

ソリューション

アプリケーションは、使用中のコピー・モード用プログラミング契約に従う必要があります。 COPY_ON_READ および COPY_ON_WRITE コピー・モードの場合、アプリケーションは、値参照を取得したトランザクションの有効範囲外の値オブジェクトへの参照を使用します。これらのモードを使用するためには、アプリケーションはトランザクションの完了後に値オブジェクトへの参照を削除し、値オブジェクトにアクセスするそれぞれのトランザクションの値オブジェクトへの新規参照を取得する必要があります。NO_COPY コピー・モードの場合、アプリケーションが値オブジェクトを一切変更しないようにする必要があります。この場合、値オブジェクトを変更しないようにアプリケーションを作成するか、別のコピー・モードを使用するようにアプリケーションを設定します。

バイト配列マップ

キーと値のペアの値を、POJO 形式で保管する代わりに、バイト配列に保管することができます。そうすると、大きなオブジェクト・グラフが消費する可能性のあるメモリー占有スペースが減ります。

利点

オブジェクト・グラフ中のオブジェクト数が増えるのにしたがって、メモリー消費量は増加します。複雑なオブジェクト・グラフを縮小して 1 つのバイト配列にすることによって、いくつかのオブジェクトの代わりに、1 つだけのオブジェクトがヒープ内に保持されるようになります。このようにヒープ内のオブジェクト数が減ることで、Java ランタイムがガーベッジ・コレクション中に検索するオブジェクトが少なくなります。

WebSphere eXtreme Scale が使用するデフォルトのコピー・メカニズムは、シリアライゼーションであり、これは高コストの処理です。例えば、デフォルトのコピー・モード COPY_ON_READ_AND_COMMIT を使用している場合、読み取り時と取得時の両方でコピーが作成されます。バイト配列を使用すると、読み取り時にコピーを作成する代わりに、値はバイトから送り込まれ、コミット時にコピーを作成する代わりに、値はシリアライズされてバイトに入れられます。バイト配列を使用した結果、データ整合性に関してはデフォルト設定と同等であり、使用メモリーは削減されます。

バイト配列を使用する際は、メモリー消費量の削減を実現するには、最適化されたシリアライゼーション・メカニズムが重要であることに注意してください。詳しくは、287 ページの『シリアライゼーション・パフォーマンス』を参照してください。

バイト配列マップの構成

バイト配列マップを使用可能にするには、以下の例に示すように、ObjectGrid XML ファイルで、マップが使用する CopyMode 属性の設定を COPY_TO_BYTES に変更します。

```
<backingMap name="byteMap" copyMode="COPY_TO_BYTES" />
```

詳しくは、「*管理ガイド*」で ObjectGrid 記述子 XML ファイルに関するトピックを参照してください。

考慮事項

特定のシナリオでバイト配列マップを使用するかどうかは、よく検討する必要があります。バイト配列を使用すると、メモリー使用量は減らせますが、プロセッサー使用量は増える場合があります。

以下に、バイト配列マップ機能の使用を選択する前に検討する必要があるいくつかの要因の概略を示します。

オブジェクト・タイプ

オブジェクト・タイプによっては、バイト配列マップを使用してもメモリー削減を期待できないものがあります。 つまり、バイト配列マップを使用するべきでない、いくつかのタイプのオブジェクトがあるということです。Java プリミティブ・ラッパーのいずれかを値として使用している場合、または、他のオブジェクトへの参照を含んでいない (プリミティブ・フィールドのみを保管する) POJO を 1 つ使用している場合、Java オブジェクトの数は既に最小限になっていて、1 つしかありません。オブジェクトが使用するメモリー量は既に最適化されているので、バイト配列

マップをこれらのタイプのオブジェクトに使用することはお勧めしません。バイト配列マップが適しているのは、POJO オブジェクト総数が 1 より大きい、他のオブジェクトまたはオブジェクトのコレクションを含んでいるオブジェクト・タイプです。

例えば、顧客オブジェクトが職場住所と自宅住所を 1 つずつ含んでいて、さらに、注文のコレクションも含んでいる場合、バイト配列マップの使用によって、ヒープ内のオブジェクト数と、これらのオブジェクトが使用するバイト数を減らすことができます。

ローカル・アクセス

その他のコピー・モードを使用する際、コピーが作成されているとき (オブジェクトがデフォルトの ObjectTransformer により Cloneable である場合)、または最適化された copyValue メソッドがカスタム ObjectTransformer に提供されているときに、アプリケーションを最適化できます。他のコピー・モードと比べて、オブジェクトにローカルでアクセスする場合、読み取り、書き込み、またはコミット操作時のコピー作成で追加コストがかかります。例えば、分散トポロジーでニア・キャッシュがある場合、またはローカルまたはサーバーの ObjectGrid インスタンスに直接アクセスしている場合は、アクセスおよびコミットの時間は、バイト配列マップを使用すると、直列化のコストがかかるため増加します。同様のコストは、ObjectGridEventGroup.ShardEvents プラグイン使用時に、データ・グリッド・エージェントを使用したり、サーバー・プライマリーにアクセスすると、分散トポロジーでも発生します。

プラグイン対話

バイト配列マップを使用すると、クライアントからサーバーに通信しているときには、サーバーが POJO フォームを必要としない限りオブジェクトはインフレートされません。マップ値と対話するプラグインでは、値をインフレートする要求が原因のパフォーマンス低下が起こります。

この追加コストは、LogElement.getCacheEntry または LogElement.getCurrentValue を使用するすべてのプラグインで発生します。キーを取得したい場合は、LogElement.getKey を使用すると、LogElement.getCacheEntry().getKey メソッドに関連した追加オーバーヘッドを回避できます。以下のセクションでは、プラグインについて、バイト配列の使用を考慮に入れて説明します。

索引および照会

オブジェクトが POJO 形式で保管されている場合、オブジェクトをインフレートする必要がないので、索引付けおよび照会を実行するコストは最小限ですみます。バイト配列マップを使用している場合、オブジェクトをインフレートするための追加コストがかかります。一般的に、アプリケーションが索引または照会を使用する場合は、キー属性に対してのみ照会を実行するのでない場合は、バイト配列マップの使用は推奨されません。

オプティミスティック・ロック

オプティミスティック・ロック・ストラテジーを使用している場合、更新操作および無効化操作中に追加コストがかかります。これは、サーバー上の値をインフレー

トして、オプティミスティック衝突のチェックを行うためのバージョン値を取得する必要があるためです。フェッチ操作を保証するためだけにオプティミスティック・ロックを使用していて、オプティミスティック衝突のチェックは必要ない場合、com.ibm.websphere.objectgrid.plugins.builtins.NoVersioningOptimisticCallback を使用して、バージョン検査を使用不可にできます。

ローダー

ローダーを使用している場合、値をインフレートしてから再シリアライズする操作をローダーが値を使用するときに行うため、eXtreme Scale ランタイムでもコストがかかります。それでも、ローダーと共にバイト配列マップを使用することができますが、そのようなシナリオでは値に変更を加えるためのコストを考慮に入れる必要があります。例えば、ほとんどが読み取りのキャッシュという状況でバイト配列機能を使用できます。この場合、ヒープ内のオブジェクト数が少なく、使用されるメモリーも少ないという利点のほうが、挿入および更新操作時にバイト配列の使用でコストが生じるというマイナス点を上回ります。

ObjectGridEventListener

ObjectGridEventListener プラグイン内で transactionEnd メソッドを使用している場合、LogElement の CacheEntry または現行値にアクセスするときのリモート要求に対する追加コストがサーバー・サイドで生じます。このメソッドの実装がこれらのフィールドにアクセスしないようになっている場合は、このような追加コストはありません。

除去

WebSphere eXtreme Scale には、キャッシュ・エントリーを除去するためのデフォルトのメカニズムと、カスタム Evictor を作成するためのプラグインが用意されています。Evictor は、各 BackingMap のエントリーのメンバーシップを制御します。デフォルトの Evictor は、各 BackingMap に対して存続時間 (TTL) 除去ポリシーを使用します。プラグ可能 Evictor 機構を提供すると、この機構では通常、時間ではなく、エントリーの数に基づいた除去ポリシーが使用されます。

デフォルトの存続時間 Evictor

WebSphere eXtreme Scale は、各 BackingMap に対して存続時間 (TTL) Evictor を提供します。 TTL Evictor は、作成される各エントリーの有効期限の時間を保守します。あるエントリーの有効期限の時間になると、Evictor はそのエントリーを BackingMap から除去します。エントリー除去のパフォーマンスへの影響を最小化するために、TTL Evictor は、有効期限の時間になるまで待機してからエントリーを除去します。TTL Evictor は、エントリーが有効期限切れになる前にエントリーを除去することはありません。

BackingMap には、存続時間 Evictor が各エントリーの有効期限の時間を計算する方法を制御する際に使用する属性があります。アプリケーションは、ttlType 属性を設定して、 TTL Evictor が有効期限の時間を計算する方法を指定します。 ttlType 属性は、以下の値のいずれかに設定できます。

1. None: BackingMap 内のエントリーに有効期限が切れないことを示します。 TTL Evictor は、これらのエントリーを除去しません。

2. Creation time : 有効期限の時間計算にエントリーの作成時刻が使用されることを示します。
3. Last access time : 有効期限の時間計算に、エントリーが最後にアクセスされた時刻が使用されることを示します。

BackingMap に ttlType 属性が設定されていない場合は、TTL Evictor がエントリーを除去しないように、デフォルトのタイプである「None」が使用されます。 ttlType 属性が「creation time」または「last access time」のいずれかに設定されている場合は、有効期限を計算する際に、BackingMap の存続時間属性の値が作成時刻または最終アクセス時刻のいずれかに加算されます。存続時間マップ属性の時刻の精度は、秒単位です。存続時間属性の値 0 は、マップ・エントリーが永続的に存続できることを示す場合に使用する特殊値です。つまり、アプリケーションによってマップ・エントリーが明示的に除去または無効化されるまで、そのエントリーがマップ内に存在し続けることを示します。

オプション Evictor

デフォルトの TTL Evictor は、時刻ベースの除去ポリシーを使用し、 BackingMap 内のエントリーの数は、エントリーの有効期限の時間には影響を及ぼしません。オプションのプラグ可能 Evictor を使用して、時刻ではなく、存在するエントリー数に基づいてエントリーを除去することができます。

以下のオプションのプラグ可能 Evictor は、 BackingMap が一定のサイズの限界を超えたときに除去するエントリーを決定するために、一般に使用されるアルゴリズムをいくつか提供します。 *

- LRUEvictor Evictor は、BackingMap が最大エントリー数を超えたときに除去するエントリーを決定する際、最長未使用時間 (LRU) アルゴリズムを使用します。
- LFUEvictor Evictor は、BackingMap が最大エントリー数を超えたときに除去するエントリーを決定する際、最少使用頻度 (LFU) アルゴリズムを使用します。

BackingMap は、トランザクション内でエントリーが作成、変更、または除去されると Evictor に通知します。 BackingMap は、これらのエントリーを継続的に追跡し、 BackingMap から 1 つ以上のエントリーをいつ除去するかを選択します。

BackingMap には、最大サイズについての構成情報はありません。代わりに、Evictor の振る舞いを制御する Evictor プロパティーが設定されます。 LRUEvictor と LFUEvictor の両方の最大サイズ・プロパティーを使用して、最大サイズを超えた後、Evictor がエントリーを除去開始するようにします。 TTL Evictor と同様に、LRU Evictor と LFU Evictor では、最大エントリー数に達した場合、パフォーマンスへの影響を最小化するためにエントリーを直ちに除去することはありません。

特定のアプリケーションに LRU または LFU 除去アルゴリズムが適していない場合、独自の Evictor を作成して、除去ストラテジーを作成できます。

メモリー・ベースの除去

重要: メモリー・ベースの除去は、Java Platform, Enterprise Edition バージョン 5 以降でのみサポートされます。

組み込み Evictor はすべて、メモリー・ベースの除去をサポートし、これは、BackingMap の evictionTriggers 属性を「MEMORY_USAGE_THRESHOLD」に設定することにより使用可能にできます。BackingMap での evictionTriggers 属性の設定方法について詳しくは、 *管理ガイド*にある BackingMap インターフェースおよび ObjectGrid 記述子 XML ファイルに関する情報を参照してください。

メモリー・ベースの除去は、ヒープ使用量のしきい値に基づいています。BackingMap でメモリー・ベースの除去が使用可能になっていて、BackingMap に組み込み Evictor がある場合、使用量のしきい値は、まだ設定されていなければ、合計メモリーのデフォルトのパーセンテージに設定されます。

メモリー・ベースの除去を使用している場合、ガーベッジ・コレクションしきい値を、ターゲット・ヒープ使用率と同じ値に構成する必要があります。例えば、メモリー・ベースの除去のしきい値が 50 パーセントに設定されていて、ガーベッジ・コレクションのしきい値がデフォルトの 70 パーセント・レベルであると、ヒープ使用率は 70 パーセントまで上がる可能性があります。このヒープ使用率の増加が起きるのは、メモリー・ベースの除去が 1 ガーベッジ・コレクション・サイクルの後にのみトリガーされるためです。

WebSphere eXtreme Scale が使用するメモリー・ベースの除去アルゴリズムは、使用中のガーベッジ・コレクションのアルゴリズムの動作に影響を受けやすいのです。メモリー・ベースの除去の最善のアルゴリズムは、IBM デフォルト・スループット・コレクターです。世代ガーベッジ・コレクション・アルゴリズムは、好ましくない動作を引き起こす可能性があるため、メモリー・ベースの除去と一緒に、このアルゴリズムを使用すべきではありません。

使用量しきい値のパーセンテージを変更するには、eXtreme Scale サーバー・プロセスのコンテナーおよびサーバーのプロパティー・ファイルで memoryThresholdPercentage プロパティーを設定します。

実行時に、メモリー使用量がターゲットの使用量しきい値を超えると、メモリー・ベースの Evictor はエントリーの除去を開始して、メモリー使用量がターゲットの使用量しきい値を下回るようにします。ただし、継続してシステム・ランタイムによるメモリー消費が迅速に進むと、除去速度が十分速くても、メモリー不足エラーが起こる可能性がなくなるという保証はありません。

デフォルト Evictor のベスト・プラクティス

属性とプロパティーを設定することにより、デフォルトの存続時間 (TTL) Evictor の振る舞いを変更できます。

プラグイン Evictor パフォーマンスのベスト・プラクティスのトピックで説明しているプラグイン Evictor の他に、すべてのバックアップ・マップでデフォルトの TTL Evictor が作成されます。デフォルトの Evictor は、存続時間 の概念に基づいてエントリーを除去します。この振る舞いは ttlType 属性で定義されます。以下の 3 つの ttlTypes 属性があります。

- None: エントリーの期限切れがないように、それによってマップからエントリーが除去されることがないように指定します。
- Creation time: 作成された時に応じてエントリーが除去されるように指定します。

- Last accessed time: 最後にアクセスされた時に応じてエントリーが除去されるように指定します。

TimeToLive プロパティー

このプロパティーは ttlType プロパティーと並んで、パフォーマンスの観点から見ると最も重要です。 CREATION_TIME ttlType を使用している場合、 Evictor は、作成からの時間がその TimeToLive 属性値と等しいときにエントリーを除去します。 TimeToLive 属性値を 10 秒に設定すると、10 秒間経過した後で全マップ内のすべてが除去されます。この値を CREATION_TIME ttlType に設定する場合は注意が必要です。この Evictor は、一定時間にのみ使用される、キャッシュへの妥当な追加量がある場合に、最も有効に使用されます。このストラテジーによって、作成されたものはすべて、一定時間後に除去されます。

以下は、CREATION_TIME の TTL タイプが有効である場合の例です。ユーザーは株価情報を入手する Web アプリケーションを使用しており、最新情報を入手することが重要でないとします。この場合、株価情報は 20 分間 ObjectGrid にキャッシュされます。 20 分後、ObjectGrid マップの有効期限が切れ、除去されます。ほぼ 20 分ごとに、ObjectGrid マップはローダー・プラグインを使用してマップ・データをデータベースの新しいデータで更新します。データベースは 20 分ごとに最新の株価情報によって更新されます。つまり、このアプリケーションの場合、TimeToLive 値を 20 分にして使用するのが理想的です。

LAST_ACCESSED_TIME ttlType 属性を使用している場合は、CREATION_TIME ttlType を使用している場合よりも TimeToLive をより低い数に設定します。エントリーの TimeToLive 属性は、アクセスされるたびにリセットされるからです。言い換えれば、TimeToLive 属性が 15 で、エントリーが 14 秒間存在し、それからアクセスされた場合、このエントリーはあと 15 秒間有効期限が切れることはありません。TimeToLive を比較的高い数値に設定した場合は、多くのエントリーがまったく除去されなくなる可能性があります。ただし、この値を 15 秒程度に設定すると、エントリーは頻繁にアクセスされない場合に除去されることになります。

以下は、LAST_ACCESSED_TIME の TTL タイプが有効である場合の例です。ObjectGrid マップはクライアントからのセッション・データを保持するために使用されます。セッション・データは、クライアントがそのセッション・データを一定時間使用しない場合は破棄する必要があります。例えば、セッション・データは、クライアントによるアクティビティーが 30 分間なかった後にタイムアウトになるとします。この場合、LAST_ACCESSED_TIME の TTL タイプを使用し、その TimeToLive 属性を 30 分に設定すると、このアプリケーションにまさに必要な条件になります。

以下の例ではバックアップ・マップを設定し、そのデフォルト Evictor の ttlType 属性を設定し、TimeToLive プロパティーを設定しています。

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("SomeMap");
bMap.setTtlEvictorType(TTLType.LAST_ACCESSED_TIME);
bMap.setTimeToLive(1800);
```

Evictor のほとんどの設定は、ObjectGrid を初期化する前に設定しておく必要があります。

独自のエビクターを作成することもできます。詳しくは、 *プログラミング・ガイド* のカスタム・エビクターの作成に関する説明を参照してください。

カスタム Evictor の作成

WebSphere eXtreme Scale を拡張して、任意の除去アルゴリズムを使用できます。

Evictor インターフェースを実装するカスタム Evictor を作成し、一般的な eXtreme Scale プラグイン規則に従う必要があります。インターフェースは、次のとおりです。

```
public interface Evictor
{
    void initialize(BackingMap map, EvictionEventCallback callback);
    void activate();
    void apply(LogSequence sequence);
    void deactivate();
    void destroy();
}
```

- initialize メソッドは、BackingMap オブジェクトの初期化中に呼び出されます。このメソッドは、BackingMap への参照、および com.ibm.websphere.objectgrid.plugins.EvictionEventCallback インターフェースを実装するオブジェクトへの参照を用いて、Evictor プラグインを初期化します。
- activate メソッドは、Evictor を活動化する場合に呼び出されます。このメソッドが呼び出されると、Evictor は EvictionEventCallback インターフェースを使用して、マップ・エントリーを除去します。activate メソッドが呼び出される前に、Evictor が EvictionEventCallback インターフェースを使用してマップ・エントリーを除去しようとすると、IllegalStateException 例外が発生します。
- apply メソッドは、BackingMap の 1 つ以上のエントリーにアクセスするトランザクションがコミットされたときに呼び出されます。 apply メソッドは、com.ibm.websphere.objectgrid.plugins.LogSequence インターフェースを実装するオブジェクトへの参照に渡されます。 LogSequence インターフェースを使用すると、Evictor プラグインは、トランザクションによって作成、変更、または除去された BackingMap エントリーを判別することができます。 Evictor は、この情報を使用して、いつ、どのエントリーを除去するかを決定します。
- deactivate メソッドは、Evictor を非活動化する場合に呼び出されます。このメソッドが呼び出されると、Evictor は、EvictionEventCallback インターフェースを使用してマップ・エントリーの除去を停止する必要があります。このメソッドが呼び出された後、Evictor が EvictionEventcallback インターフェースを使用すると、IllegalStateException 例外が発生します。
- destroy メソッドは、BackingMap を破棄するときに呼び出されます。このメソッドを使用すると、Evictor は作成した任意のスレッドを終了できます。

EvictionEventCallback インターフェースには、以下のメソッドがあります。

```
public interface EvictionEventCallback
{
    void evictMapEntries(List evictorDataList) throws ObjectGridException;
    void evictEntries(List keysToEvictList) throws ObjectGridException;
    void setEvictorData(Object key, Object data);
    Object getEvictorData(Object key);
}
```

EvictionEventCallback メソッドは、次のように Evictor プラグインによって、eXtreme Scale フレームワークをコールバックするために使用されます。

- setEvictorData メソッドは Evictor によって使用され、使用されている ObjectGrid フレームワークに、その Evictor が作成した Evictor オブジェクトを保管し、引数 key で示されるエントリーにその Evictor オブジェクトを関連付けるよう要求します。このデータは、Evictor 固有であり、使用するアルゴリズムを実装するために Evictor が必要とする情報によって判別されます。例えば、最少使用頻度アルゴリズムでは、Evictor は、指定されたキーのエントリーを参照する LogElement で apply メソッドが呼び出された回数を追跡するために、そのカウント数を Evictor データ・オブジェクトに保持します。
- getEvictorData メソッドは Evictor によって使用され、前の apply メソッドの呼び出し中に setEvictorData メソッドに渡されたデータを取り出します。指定された引数 key に対応する Evictor データが見つからない場合は、 EvictorCallback インターフェースで定義されている特別な KEY_NOT_FOUND オブジェクトが戻されます。
- evictMapEntries メソッドは Evictor によって使用され、1 つ以上のマップ・エントリーの除去を要求します。evictorDataList パラメーター内の各オブジェクトは、com.ibm.websphere.objectgrid.plugins.EvictorData インターフェースを実装する必要があります。また、setEvictorData メソッドに渡されるのと同じ EvictorData インスタンスが、このメソッドの Evictor データ・リスト・パラメーターに存在している必要があります。除去するマップ・エントリーを決定する場合は、EvictorData インターフェースの getKey メソッドが使用されます。キャッシュ・エントリーの Evictor データ・リストにあるのとまったく同一の EvictorData インスタンスが現在このキャッシュ・エントリーに含まれている場合、このマップ・エントリーは除去されます。
- evictEntries メソッドは Evictor によって使用され、1 つ以上のマップ・エントリーの除去を要求します。このメソッドが使用されるのは、setEvictorData メソッドに渡されるオブジェクトが、 com.ibm.websphere.objectgrid.plugins.EvictorData インターフェースを実装していない場合のみです。

eXtreme Scale は、トランザクションの完了後に、Evictor インターフェースの apply メソッドを呼び出します。完了しているトランザクションによって獲得されたすべてのトランザクション・ロックは、保持されなくなります。そのため、複数のスレッドが同時に apply メソッドを呼び出し、各スレッドが別々のトランザクションを完了することも起こり得ます。トランザクション・ロックは、完了しているトランザクションによって既に解放されているので、apply メソッドはそれ自体で同期化を行って、それがスレッド・セーフであることを保証する必要があります。

EvictorData インターフェースを実装して、evictEntries メソッドの代わりに evictMapEntries メソッドを使用する理由は、そのような時間帯をなくすことにあります。次のイベント・シーケンスを考えてみましょう。

1. トランザクション 1 が完了し、LogSequence で apply メソッドを呼び出して、キー 1 のマップ・エントリーを削除する。
2. トランザクション 2 が完了し、LogSequence で apply メソッドを呼び出して、キー 1 の新規マップ・エントリーを挿入する。つまり、トランザクション 2 は、トランザクション 1 が削除したマップ・エントリーを再作成します。

Evictor は、トランザクションを実行するスレッドとは非同期で実行するので、その Evictor でキー 1 を除去する場合、Evictor は、トランザクション 1 が完了する前に存在していたマップ・エントリーを除去するか、トランザクション 2 が再作成したマップ・エントリーを除去する可能性があります。時間帯をなくすため、どのバージョンのキー 1 のマップ・エントリーを除去するのかを明確にするために、setEvictorData メソッドに渡されるオブジェクトによって EvictorData インターフェースを実装します。マップ・エントリーの存続期間中は、同じ EvictorData インスタンスを使用します。そのマップ・エントリーが削除され、次に別のトランザクションによって再作成されるときは、Evictor は、EvictorData 実装の新規インスタンスを使用する必要があります。Evictor は、EvictorData 実装および evictMapEntries メソッドを使用することにより、マップ・エントリーに関連付けられているキャッシュ・エントリーに正しい EvictorData インスタンスが含まれている場合に限り、そのマップ・エントリーが除去されることを保証できます。

Evictor インターフェースと EvictonEventCallback インターフェースにより、アプリケーションは、ユーザー定義の除去アルゴリズムを実装する Evictor を接続することができます。次のコードの断片は、Evictor インターフェースの initialize メソッドを実装する方法を示しています。

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.EvictionEventCallback;
import com.ibm.websphere.objectgrid.plugins.Evictor;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;
import java.util.LinkedList;
// Instance variables
private BackingMap bm;
private EvictionEventCallback evictorCallback;
private LinkedList queue;
private Thread evictorThread;
public void initialize(BackingMap map, EvictionEventCallback callback)
{
    bm = map;
    evictorCallback = callback;
    queue = new LinkedList();
    // spawn evictor thread
    evictorThread = new Thread( this );
    String threadName = "MyEvictorForMap-" + bm.getName();
    evictorThread.setName( threadName );
    evictorThread.start();
}
```

前掲のコードでは、マップ・オブジェクトとコールバック・オブジェクトへの参照をインスタンス変数内に保存します。これにより、これらのオブジェクトを apply メソッドと destroy メソッドで使用することができます。この例では、最長未使用時間 (LRU) アルゴリズムを実装するための先入れ、先出しキューとして使用されるリンク・リストが作成されます。スレッドが作成され、そのスレッドへの参照は、インスタンス変数として保持されます。この参照を保持することにより、destroy メソッドは作成されたスレッドに割り込んで終了させることができます。

次のコード・スニペットは、コードをスレッド・セーフにするための同期要件を無視して、Evictor インターフェースの apply メソッドを実装する方法を示しています。

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.EvictionEventCallback;
import com.ibm.websphere.objectgrid.plugins.Evictor;
import com.ibm.websphere.objectgrid.plugins.EvictorData;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;
public void apply(LogSequence sequence)
{
    Iterator iter = sequence.getAllChanges();
    while ( iter.hasNext() )
    {
        LogElement elem = (LogElement)iter.next();
```

```
            Object key = elem.getKey();
            LogElement.Type type = elem.getType();
            if ( type == LogElement.INSERT )
            {
                // do insert processing here by adding to front of LRU queue.
                EvictorData data = new EvictorData(key);
                evictorCallback.setEvictorData(key, data);
                queue.addFirst( data );
            }
            else if ( type == LogElement.UPDATE || type == LogElement.FETCH || type == LogElement.TOUCH )
            {
                // do update processing here by moving EvictorData object to
                // front of queue.
                EvictorData data = evictorCallback.getEvictorData(key);
                queue.remove(data);
                queue.addFirst(data);
            }
            else if ( type == LogElement.DELETE || type == LogElement.EVICT )
            {
                // do remove processing here by removing EvictorData object
                // from queue.
                EvictorData data = evictorCallback.getEvictorData(key);
                if ( data == EvictionEventCallback.KEY_NOT_FOUND )
                {
                    // Assumption here is your asynchronous evictor thread
                    // evicted the map entry before this thread had a chance
                    // to process the LogElement request. So you probably
                    // need to do nothing when this occurs.
                }
                else
                {
                    // Key was found. So process the evictor data.
                    if ( data != null )
                    {
                        // Ignore null returned by remove method since spawned
                        // evictor thread may have already removed it from queue.
                        // But we need this code in case it was not the evictor
                        // thread that caused this LogElement to occur.
                        queue.remove( data );
                    }
                    else
                    {
                        // Depending on how you write you Evictor, this possibility
                        // may not exist or it may indicate a defect in your evictor
                        // due to improper thread synchronization logic.
                    }
                }
            }
        }
    }
```

apply メソッド内への挿入処理では、通常、EvictionEventCallback インターフェースの setEvictorData メソッドに渡される Evictor データ・オブジェクトが作成されます。この Evictor は、LRU 実装を示すためのものなので、initialize メソッドで作成されたキューの前に EvictorData も追加されます。 apply メソッド内の更新処理は通常、apply メソッドの前の呼び出し (例えば、apply メソッドの挿入処理による) によって作成された Evictor データ・オブジェクトを更新します。この Evictor は LRU 実装であるため、 EvictorData オブジェクトをその現在のキュー位置からキューの前方に移動させる必要があります。作成された Evictor スレッドは、最後のキュー・エレメントが LRU エントリーを表しているため、キュー内の最後の EvictorData オブジェクトを除去します。その前提として、EvictorData オブジェクトは getKey メソッドを持っています。これによって、Evictor スレッドは、除去する必要があるエントリーのキーを認識します。この例では、コードをスレッド・セーフにするための同期要件を無視していることに留意してください。実際のカスタム Evictor は、同期と同期点の結果として発生するパフォーマンスの障害を取り扱っているため、より複雑です。

以下のコード・スニペットは、initialize メソッドが作成した実行可能スレッドの destroy メソッドと run メソッドを示しています。

```
// Destroy method simply interrupts the thread spawned by the initialize method.
public void destroy()
{
    evictorThread.interrupt();
}
```

```
// Here is the run method of the thread that was spawned by the initialize method.
public void run()
{
    // Loop until destroy method interrupts this thread.
    boolean continueToRun = true;
    while ( continueToRun )
    {
        try
        {
            // Sleep for a while before sweeping over queue.
            // The sleepTime is a good candidate for a evictor
            // property to be set.
            Thread.sleep( sleepTime );
            int queueSize = queue.size();
            // Evict entries if queue size has grown beyond the
            // maximum size. Obviously, maximum size would
            // be another evictor property.
            int numToEvict = queueSize - maxSize;
            if ( numToEvict > 0 )
            {
                // Remove from tail of queue since the tail is the
                // least recently used entry.
                List evictList = new ArrayList( numToEvict );
                while( queueSize > ivMaxSize )
                {
                    EvictorData data = null;
                    try
                    {
                        EvictorData data = (EvictorData) queue.removeLast();
                        evictList.add( data );
                        queueSize = queue.size();
                    }
                    catch ( NoSuchElementException nse )
                    {
                        // The queue is empty.
                        queueSize = 0;
                    }
                }
                // Request eviction if key list is not empty.
                if ( ! evictList.isEmpty() )
                {
                    evictorCallback.evictMapEntries( evictList );
                }
            }
        }
        catch ( InterruptedException e )
        {
            continueToRun = false;
        }
    } // end while loop
} // end run method.
```

オプションの RollBackEvictor インターフェース

com.ibm.websphere.objectgrid.plugins.RollbackEvictor インターフェースは、オプションで、Evictor プラグインによって実装することができます。このインターフェースを実装することにより、トランザクションがコミットされたときだけでなく、トランザクションがロールバックされたときにも、Evictor を呼び出すことができます。

```
public interface RollbackEvictor
{
    void rollingBack( LogSequence ls );
}
```

apply メソッドは、トランザクションがコミットされたときにのみ呼び出されます。トランザクションがロールバックされたとき、Evictor が RollbackEvictor インターフェースを実装している場合は、rollingBack メソッドが呼び出されます。RollbackEvictor インターフェースが実装されていない場合は、トランザクションがロールバックされても、apply メソッドおよび rollingBack メソッドは呼び出されません。

プラグイン Evictor パフォーマンスのベスト・プラクティス

プラグイン Evictor を使用する場合は、Evictor を作成し、これらを使用することをバックアップ・マップに伝えるまで Evictor はアクティブになりません。LFU Evictor と LRU Evictor について、ベスト・プラクティスとパフォーマンスのヒントを使用してください。

LFU Evictor

LFU Evictor の概念は、頻繁に使用されないマップからエントリーを除去することです。マップのエントリーは、一定量のバイナリー・ヒープを超えて広がります。特定のキャッシュ・エントリーの使用量が増えると、それはヒープの高位に配列されます。Evictor が一連の除去を試行する場合、バイナリー・ヒープの特定のポイントよりも低い位置にあるキャッシュ・エントリーだけを除去します。この結果として、頻繁に使用されないエントリーが除去されます。

LRU Evictor

LRU Evictor は LFU Evictor と同じ概念に従いますが、2、3 の点が異なります。主な違いは、LRU ではバイナリー・ヒープのセットの代わりに先入れ先出し (FIFO) キューを使用することです。キャッシュ・エントリーにアクセスされるたびに、そのエントリーはキューの先頭に移動します。この結果、キューの先頭には最後に使用されたマップ・エントリーが含まれ、キューの最後は最長未使用時間のマップ・エントリーになります。例えば、A キャッシュ・エントリーが 50 回使用され、B キャッシュ・エントリーが A キャッシュ・エントリーの直後に 1 回だけ使用されるとします。この場合、最後に使用された B キャッシュ・エントリーがキューの先頭になり、A キャッシュ・エントリーはキューの最後になります。 LRU Evictor は、キューの末尾にあるキャッシュ・エントリー、すなわち最も古いマップ・エントリーを除去します。

LFU および LRU プロパティーおよびパフォーマンスを向上させるためのベスト・プラクティス

ヒープ数

LFU Evictor を使用する場合は、特定のマップのすべてのキャッシュ・エントリーが指定するヒープ数を超えて配列されます。これによってパフォーマンスが劇的に上がり、また、そのマップのすべての配列を含む、1 つのバイナリー・ヒープ上ですべての除去が同期するのを防ぎます。ヒープが多い場合も、各ヒープのエントリーが少ないので再配列に必要な時間を短縮できます。ご使用の BaseMap でエントリー数の 10% のヒープ数を設定してください。

キューの数

LRU Evictor を使用する場合は、特定のマップのすべてのキャッシュ・エントリーは指定する LRU キューの数を超えて配列されます。これによってパフォーマンスが劇的に上がり、また、そのマップのすべての配列を含む、1 つのキュー上ですべての除去が同期するのを防ぎます。ご使用の BaseMap でエントリー数の 10% のキューの数を設定してください。

MaxSize プロパティー

LFU または LRU Evictor がエントリーの除去を開始すると、MaxSize Evictor プロパティーを使用して、いくつのバイナリー・ヒープまたは LRU キュー・エレメントを除去するかを判別します。例えば、各マップ・キューにおよそ 10 のマップ・エントリーを持つようにヒープまたはキューの数を設定するとします。 MaxSize プロパティーが 7 に設定されている場合は、Evictor は各ヒープまたはキュー・オブジェクトの 3 つのエントリーを除去して、各ヒープまたはキューのサイズを 7 にします。Evictor は、ヒープまたはキューに、エレメントの MaxSize プロパティーの値を超えるエレメントがある場合にのみ、マップ・エントリーをヒープまたはキューから除去します。MaxSize をヒープまたはキュー・サイズの 70% に設定してください。この例の場合、値は 7 に設定されます。ユーザーは、BaseMap エントリーの数を、使用するヒープまたはキューの数で割ることによって、各ヒープまたはキューのおおよそのサイズを得ることができます。

SleepTime プロパティー

Evictor はマップから常にエントリーを除去するわけではありません。その代わり、一定時間アイドル状態となり、マップの検査のみが n 秒間に 1 回行われます。ここで、n は SleepTime プロパティーを示します。このプロパティーも確実にパフォーマンスに影響します。あまり頻繁に除去スイープを実行すると、それを処理するためにリソースが必要となり、パフォーマンスが低下します。ただし、エビクターを頻繁に使用しないと、不要なエントリーがマップ内に存在するという結果となります。不要なエントリーでいっぱいのマップは、メモリー所要量にもマップに必要な処理用リソースにも悪影響を与えます。除去スイープ間隔を 15 秒に設定すると、ほとんどのマップで良好な事例が得られます。マップが頻繁に書き込まれ、高速のトランザクションで使用される場合は、この値をより低く設定することを検討してください。頻繁にマップにアクセスしない場合は、この時間をより高い値に設定することができます。

例

以下の例ではマップを定義し、新しい LFU Evictor を作成し、Evictor のプロパティーを設定し、Evictor を使用するようにマップを設定します。

```
//Use ObjectGridManager to create/get the ObjectGrid. Refer to
// the ObjectGridManger section
ObjectGrid objGrid = ObjectGridManager.create.....
BackingMap bMap = objGrid.defineMap("SomeMap");

//Set properties assuming 50,000 map entries
LFUEvictor someEvictor = new LFUEvictor();
someEvictor.setNumberOfHeaps(5000);
someEvictor.setMaxSize(7);
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

LRU Evictor を使用するのは LFU Evictor を使用するのとよく似ています。以下に例を示します。

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("SomeMap");

//Set properties assuming 50,000 map entries
LRUEvictor someEvictor = new LRUEvictor();
```

```
someEvictor.setNumberOfLRUQueues(5000);
someEvictor.setMaxSize(7);
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

LFU Evictor の例とは 2 行だけ異なっていることに注意してください。

ロック・パフォーマンスのベスト・プラクティス

ロック・ストラテジーおよびトランザクション分離設定は、アプリケーションのパフォーマンスに影響します。

キャッシュ付きインスタンスの検索

詳しくは、*管理ガイド*のマップ・エントリーのロックに関する説明を参照してください。

ペシミスティック・ロック・ストラテジー

キーがしばしば衝突する場合のマップの読み取りおよび書き込み操作には、ペシミスティック・ロック・ストラテジーを使用します。ペシミスティック・ロック・ストラテジーは、パフォーマンスに最大の影響があります。

読み取りコミット済みおよび読み取りアンコミットのトランザクション分離

ペシミスティック・ロック・ストラテジーを使用する場合、Session.setTransactionIsolation メソッドを使用してトランザクション分離レベルを設定します。読み取りコミット済み分離または読み取りアンコミット分離の場合、分離に応じて Session.TRANSACTION_READ_COMMITTED 引数または Session.TRANSACTION_READ_UNCOMMITTED 引数を使用します。トランザクション分離レベルをデフォルトのペシミスティック・ロックの振る舞いにリセットするには、Session.REPEATABLE_READ 引数を持つ Session.setTransactionIsolation メソッドを使用します。

読み取りコミット済み分離では、共用ロックの期間が短縮され、並行性が向上して、デッドロックの可能性が低くなります。この分離レベルは、トランザクションが、トランザクションの期間中、読み取り値が変更されないままである保証が不要な場合に使用してください。

アンコミット読み取りは、トランザクションがコミット済みデータを参照する必要がない場合に使用します。

オプティミスティック・ロック・ストラテジー

オプティミスティック・ロックはデフォルト構成です。このストラテジーはペシミスティック・ストラテジーと比較して、パフォーマンスおよびスケーラビリティーの両方において優れています。アプリケーションが若干のオプティミスティック更新の失敗を許容でき、ペシミスティック・ストラテジーよりもパフォーマンスに優れている場合は、このストラテジーを使用します。このストラテジーは、読み取り操作や、更新頻度の低いアプリケーションに最適です。

OptimisticCallback プラグイン

オプティミスティック・ロック・ストラテジーでは、キャッシュ・エントリーのコピーを作成し、必要に応じてそれらを比較します。エントリーのコピーには、クローン作成やシリアライゼーションが関係する可能性があるため、この操作はコストが高くつきます。パフォーマンスをできる限り高速にするには、非エンティティー・マップ用にカスタム・プラグインを実装してください。

詳しくは、*製品概要*の OptimisticCallback プラグインに関する説明を参照してください。

エンティティーに対するバージョン・フィールドの使用

エンティティーに対してオプティミスティック・ロックを使用している場合、@Version アノテーション、または、エンティティー・メタデータ記述子ファイルの同等の属性を使用します。バージョン・アノテーションを使用すれば、ObjectGrid で非常に効率的にオブジェクトのバージョンを追跡することができます。エンティティーにバージョン・フィールドがなく、エンティティーに対してオプティミスティック・ロックが使用されている場合、エンティティー全体がコピーされ、比較されます。

ロックなしストラテジー

読み取り専用アプリケーションでは、ロックなしストラテジーを使用します。ロックなしストラテジーではいかなるロックも取得せず、ロック・マネージャーも使用しません。このため、このストラテジーは最も並行性、パフォーマンス、スケーラビリティーに優れています。

マップ・エントリー・ロックと照会および索引

このトピックでは、eXtreme Scale Query API および MapRangeIndex 索引付けプラグインがロックとどのように相互作用するのかを説明し、ペシミスティック・ロック・ストラテジー使用時に並行性を増し、デッドロックを減らす、ベスト・プラクティスをいくつか示します。

概説

ObjectGrid Query API では、ObjectMap キャッシュ・オブジェクトおよびエンティティーに対して SELECT 照会を行うことができます。照会エンジンが実行されると、可能であれば MapRangeIndex を使用して、照会の WHERE 文節にある値に一致する一致キーを検索し、または、リレーションシップをブリッジします。索引が使用可能でない場合、照会エンジンは、1 つ以上のマップの各エントリーをスキャンして、適切なエントリーを検索します。照会エンジンおよび索引プラグインは、どちらもロックを獲得して、ロック・ストラテジー、トランザクション分離レベル、およびトランザクション状態に応じて、整合データを検査します。

HashIndex プラグインによるロック

eXtreme Scale HashIndex プラグインを使用すると、キャッシュ・エントリー値に保管された単一の属性に基づいてキーを検出できます。索引は、キャッシュ・マップとは別のデータ構造に索引付けされた値を保管します。索引は、ユーザーに返す前にマップ・エントリーに対してキーを検証し、正確な結果セットになるようにします。ペシミスティック・ロック・ストラテジーが使用され、ローカル ObjectMap イ

ンスタンス (クライアント/サーバー ObjectMap に対するものとして) に対して索引が使用される場合、索引は各一致エントリーに対してロックを獲得します。オプティミスティック・ロックまたはリモート ObjectMap を使用する場合、ロックは直ちに解放されます。

獲得されるロックのタイプは、ObjectMap.getIndex メソッドに渡される forUpdate 引数によって異なります。forUpdate 引数は、索引が獲得すべきロックのタイプを指定します。false の場合、共用可能 (S) ロックが獲得され、true の場合は、アップグレード可能 (U) ロックが獲得されます。

ロック・タイプが共用可能の場合、セッションのトランザクション分離設定が適用され、ロックの期間に影響します。トランザクション分離を使用してアプリケーションに並行性を追加する方法についての詳細は、トランザクション分離のトピックを参照してください。

共用ロックと照会

eXtreme Scale 照会エンジンは、キャッシュ・エントリーが照会のフィルター基準を満たしているかどうかを検査するためにキャッシュ・エントリーをイントロスペクトするのに必要な場合は、S ロックを獲得します。ペシミスティック・ロックで反復可能読み取りトランザクション分離を使用する場合、照会結果に含まれるエレメントに対してのみ S ロックが保持され、結果に含まれていないエントリーについては解放されます。低いトランザクション分離レベルまたはオプティミスティック・ロックを使用している場合、S ロックは保持されません。

共用ロックと、クライアントからサーバーに対する照会

eXtreme Scale 照会をクライアントから使用する場合、照会内で参照されているすべてのマップまたはエンティティーがクライアントに対してローカル (例: クライアント複製マップまたは照会結果エンティティー) でない限り、通常、照会はサーバーで実行されます。読み取り/書き込みトランザクションで実行されるすべての照会は、前のセクションで説明したように S ロックを保持します。トランザクションが読み取り/書き込みトランザクションでない場合は、セッションはサーバーで保持されず、S ロックは解放されます。

読み取り/書き込みトランザクションは、プライマリー区画に対してのみルーティングされ、セッションは、クライアント・セッションについてはサーバーで維持されます。トランザクションは、以下の条件で読み取り/書き込みにプロモートできます。

1. ペシミスティック・ロックを使用するように構成されたマップが、ObjectMap get および getAll API メソッド、または、EntityManager.find メソッドを使用してアクセスされる場合。
2. トランザクションがフラッシュされ、それによって更新がサーバーに送られる場合。
3. オプティミスティック・ロックを使用するように構成されたマップが、ObjectMap.getForUpdate メソッド、または、EntityManager.findForUpdate メソッドを使用してアクセスされる場合。

アップグレード可能ロックと照会

共用可能ロックは、並行性および整合性が重要な場合に有効です。共用可能ロックでは、トランザクションの存続期間中、エントリーの値が変わらないことが保証されます。他の S ロックが保持されている間、他のトランザクションが値を変更することはできず、エントリーを更新するインテントを設定できるのは他の 1 つのトランザクションのみです。S、U、および X ロック・モードに関する詳細は、ペシミスティック・ロック・モードのトピックを参照してください。

アップグレード可能ロックは、ペシミスティック・ロック・ストラテジーを使用する場合にキャッシュ・エントリーの更新インテントを特定するために使用されます。アップグレード可能ロックでは、キャッシュ・エントリーを変更しようとするトランザクション間の同期を行うことができます。トランザクションは、S ロックを使用して引き続きエントリーを参照することができますが、他のトランザクションは U ロックまたは X ロックを獲得できなくなります。多くのシナリオでは、デッドロックを回避するため、先に S ロックを獲得せずに U ロックを獲得することが必要になります。一般的なデッドロックの例については、ペシミスティック・ロック・モードのトピックを参照してください。

ObjectQuery および EntityManager Query インターフェースでは、照会結果の用途の特定に setForUpdate メソッドを提供しています。特に、照会エンジンは、照会結果に含まれる各マップ・エントリーに対して S ロックではなく U ロックを獲得します。

```
ObjectMap orderMap = session.getMap("Order");
ObjectQuery q = session.createQuery("SELECT o FROM Order o WHERE o.orderDate=?1");
q.setParameter(1, "20080101");
q.setForUpdate(true);
session.begin();
// Run the query.  Each order has  U lock
Iterator result = q.getResultIterator();
// For each order, update the status.
while(result.hasNext()) {
    Order o = (Order) result.next();
    o.status = "shipped";
    orderMap.update(o.getId(), o);
}
// When committed, the
session.commit();
```

```
Query q = em.createQuery("SELECT o FROM Order o WHERE o.orderDate=?1");
q.setParameter(1, "20080101");
q.setForUpdate(true);
emTran.begin();
// Run the query.  Each order has  U lock
Iterator result = q.getResultIterator();
// For each order, update the status.
while(result.hasNext()) {
    Order o = (Order) result.next();
    o.status = "shipped";
}
tmTran.commit();
```

setForUpdate 属性が使用可能になっている場合、トランザクションは、自動的に読み取り/書き込みトランザクションに変換され、予期されたようにサーバーに対してロックが保持されます。照会が索引を使用できない場合、マップをスキャンして、照会結果を満足しないマップ・エントリーに対して一時的に U ロックをかけ、結果に含まれるエントリーに対しては U ロックを保持するようにする必要があります。

ObjectTransformer インターフェースのベスト・プラクティス

ObjectTransformer インターフェースは、アプリケーションへのコールバックを使用して、通常の操作と、オブジェクト・シリアライゼーションやオブジェクトに対するディープ・コピーなどのコストのかかる操作のカスタム実装を提供します。

概説

ObjectTransformer インターフェースについて詳しくは、 203 ページの『ObjectTransformer プラグイン』を参照してください。 パフォーマンスの観点、および CopyMode メソッドのベスト・プラクティスのトピックに含まれる情報から見ると、NO_COPY モードが使用されている場合を除き、すべての場合に eXtreme Scale が値をコピーするのは明白です。 eXtreme Scale 内で採用されているデフォルトのコピー・メカニズムはシリアライゼーションであり、これはコストのかかる操作として知られています。 ObjectTransformer インターフェースはこのような状況で使用します。ObjectTransformer インターフェースは、アプリケーションへのコールバックを使用して、通常の操作と、オブジェクト・シリアライズやオブジェクトに対するディープ・コピーなどのコストのかかる操作のカスタム実装を提供します。

アプリケーションで、マップに対する ObjectTransformer インターフェースの実装が提供できると、eXtreme Scale は、このオブジェクトに対するメソッドに権限を委任し、インターフェースにおける各メソッドの最適化バージョンの提供はアプリケーションに頼ります。 ObjectTransformer インターフェースは以下のようになります。

```
public interface ObjectTransformer {
    void serializeKey(Object key, ObjectOutputStream stream) throws IOException;
    void serializeValue(Object value, ObjectOutputStream stream) throws IOException;
    Object inflateKey(ObjectInputStream stream) throws IOException, ClassNotFoundException;
    Object inflateValue(ObjectInputStream stream) throws IOException, ClassNotFoundException;
    Object copyValue(Object value);
    Object copyKey(Object key);
}
```

次のコード例を使用して、ObjectTransformer インターフェースを BackingMap に関連付けることができます。

```
ObjectGrid g = ...;
BackingMap bm = g.defineMap("PERSON");
MyObjectTransformer ot = new MyObjectTransformer();
bm.setObjectTransformer(ot);
```

オブジェクト・シリアライゼーションおよびオブジェクト・インフレーションの調整

オブジェクト・シリアライゼーションは、eXtreme Scale を使用した場合に通常、最も重要なパフォーマンスの考慮事項です。この eXtreme Scale は、アプリケーションで ObjectTransformer プラグインが提供されない場合に、デフォルトのシリアライズ化メカニズムを使用します。アプリケーションは Serializable readObject と writeObject の実装を供給するか、または、Externalizable インターフェースを実装するオブジェクトを持つことができますが、後者の方が 10 倍高速です。マップ内のオブジェクトを変更できない場合、アプリケーションは ObjectTransformer インターフェースを ObjectMap に関連付けることができます。serialize メソッドおよび inflate メソッドが提供されることにより、アプリケーションは、システムのパフォーマンスに大きく影響するこれらの操作を最適化するためのカスタム・コードを提供できます。serialize メソッドは、与えられたストリームにオブジェクトをシリアライズします。 inflate メソッドは入力ストリームを提供します。そしてアプリケー

ションがオブジェクトを作成し、ストリーム内のデータを使用してオブジェクトをインフレートし、最後にオブジェクトを戻すものと想定します。 serialize メソッドと inflate メソッドの実装は、相互にミラーリングする必要があります。

ディープ・コピー操作を調整する

アプリケーションが ObjectMap からオブジェクトを受け取った後で、eXtreme Scale は、オブジェクト値に対してディープ・コピーを実行し、BaseMap マップ内のコピーがデータ保全性を維持するようにします。その後アプリケーションはこのオブジェクト値を安全に変更できます。トランザクションがコミットすると、BaseMap マップ内のオブジェクト値のコピーは新しく変更される値に更新され、アプリケーションはその時点からその値の使用を停止します。コミット・フェーズで再度オブジェクトをコピーして、プライベート・コピーを作成した可能性があります。ただし、この場合は、このアクションのパフォーマンス・コストは、トランザクションのコミットの後で値を使用しないようアプリケーション・プログラマーに要求することに対してトレードオフされました。デフォルトの ObjectTransformer は、clone または serialize と inflate のペアを使用して、コピーを生成しようとします。直列化とインフレーションのペアは、最悪なパフォーマンス・シナリオです。プロファイル作成によって、serialize と inflate がご使用のアプリケーションにとって問題であることが判明したら、ディープ・コピーを作成する適切な clone メソッドを書きます。クラスを変更できない場合は、カスタム ObjectTransformer プラグインを作成し、より効率的な copyValue および copyKey メソッドを実装します。

シリアライゼーション・パフォーマンス

WebSphere eXtreme Scale は、複数の Java プロセスを使用してデータを保持します。Java オブジェクト・インスタンス形式のデータはバイトに変換され、必要に応じて再びオブジェクトに戻されます。この変換は、クライアント・プロセスとサーバー・プロセスの間でのデータ移動のために行われます。データのマーシャルは最もコストのかかる操作であり、アプリケーション開発者は、スキーマを設計し、グリッドを構成し、データ・アクセス API と対話する際に、それに対処する必要があります。

デフォルトの Java シリアライゼーション・ルーチンおよびコピー・ルーチンは、比較的遅く、標準的なセットアップではプロセッサーの 60 から 70 パーセントを消費する場合があります。以降のセクションに、シリアライゼーションのパフォーマンスを改善するための選択肢を示します。

各 BackingMap 用 ObjectTransformer の作成

ObjectTransformer は、BackingMap に関連付けることができます。ObjectTransformer インターフェースを実装し、かつ以下の操作のための実装を提供するクラスを、アプリケーションに含めることができます。

- 値のコピー
- ストリーム間での、キーのシリアライズとインフレーション
- ストリーム間での、値のシリアライズとインフレーション

キーは不変であると見なされるため、アプリケーションはキーをコピーする必要はありません。

注: ObjectTransformer は、変換中のデータを ObjectGrid が理解している場合にのみ起動されます。例えば、DataGrid API エージェントが使用される場合は、エージェントそのものに加えて、エージェント・インスタンス・データまたはエージェントから返されるデータも、カスタムのシリアライゼーション技法を使用して最適化されなければなりません。ObjectTransformer は、DataGrid API エージェントに対しては起動されません。

エンティティーの使用

エンティティーで EntityManager API が使用されている場合、ObjectGrid は、エンティティー・オブジェクトを BackingMap に直接的には保管しません。EntityManager API はエンティティー・オブジェクトを Tuple オブジェクトに変換します。詳しくは、詳しくは、*プログラミング・ガイド*のエンティティー・マップおよびタプルでのローダーの使用に関するトピックを参照してください。 エンティティー・マップは、高度に最適化された ObjectTransformer と自動的に関連付けられます。ObjectMap API または EntityManager API を使用してエンティティー・マップと対話する際、必ずエンティティー ObjectTransformer が起動されます。

カスタムのシリアライゼーション

一部のケースでは、オブジェクトを変更して、カスタム・シリアライゼーションを使用するようにする必要があります (例えば、java.io.Externalizable インターフェースを実装する、または {[java.io.Serializable]} インターフェースを実装しているクラスの writeObject および readObject メソッドを実装するなど)。ObjectGrid API または EntityManager API のメソッド以外のメカニズムを使用してオブジェクトをシリアライゼーションするときは、カスタムのシリアライズした技法を採用する必要があります。

例えば、オブジェクトまたはエンティティーがインスタンス・データとして DataGrid API エージェント内に保管されるとき、またはエージェントがオブジェクトやエンティティーを返すとき、それらのオブジェクトは ObjectTransformer を使用して変換されません。ただし、`EntityMixin` インターフェースが使用されている場合、エージェントは、自動的に ObjectTransformer を使用します。詳しくは、『DataGrid エージェントとエンティティー・ベースのマップ』を参照してください。

バイト配列

ObjectMap または DataGrid API を使用している場合、クライアントがグリッドと対話するとき、および、オブジェクトが複製されるときには、キーと値のオブジェクトがシリアライズされます。シリアライゼーションのオーバーヘッドを避けるには、Java オブジェクトの代わりにバイト配列を使用します。バイト配列を使用すればメモリーへの保管にかかるコストはずっと少なくてすみます。これは、JDK がガーベッジ・コレクション中に検索するオブジェクトが少なく、必要なときだけインフレートできるためです。バイト配列は、照会または索引を使用してオブジェクトにアクセスする必要がない場合にのみ使用するべきです。データはバイトとして保管されるので、データにはキーを介してのみアクセスできます。

WebSphere eXtreme Scale は、 CopyMode.COPY_TO_BYTES マップ構成オプションを使用して、自動的にデータをバイト配列として保管できますが、クライアントに

よる手動での処理も可能です。このオプションは、データをメモリーに効率的に保管し、照会および索引によるオンデマンドでの使用のために、バイト配列内のオブジェクトを自動的にインフレートすることもできます。

照会パフォーマンス調整

照会のパフォーマンスを調整する場合は、以下の手法とヒントを使用してください。

パラメーターの使用

照会を実行する場合、照会ストリングを構文解析し、照会を実行する計画を開発する必要がありますが、両方ともコストがかかる可能性があります。WebSphere eXtreme Scale は、照会ストリングによって照会計画をキャッシュに入れます。キャッシュは有限サイズであるため、照会ストリングを可能な限り再利用することが重要です。名前付きパラメーターまたは定位置パラメーターを使用しても、照会計画の再利用が促進され、パフォーマンスが向上します。

```
Positional Parameter Example  Query q = em.createQuery("select c from
Customer c where c.surname=?1");   q.setParameter(1, "Claus");
```

索引の使用

マップに対する適切な索引付けは、マップ・パフォーマンス全体にいくらかのオーバーヘッドをもたらしますが、照会パフォーマンスに著しい効果をもたらす場合があります。照会に関係するオブジェクト属性に索引付けを行わない場合、照会エンジンは、属性ごとにテーブル・スキャンを実行します。テーブル・スキャンは、照会実行時に最もコストのかかる操作です。照会に関係するオブジェクト属性に対する索引付けにより、照会エンジンは、不必要なテーブル・スキャンを回避でき、照会パフォーマンス全体を改善することができます。アプリケーションが最も読み取られるマップに対して照会を集中的に使用するように設計されている場合は、照会に関係するオブジェクト属性に対して索引を構成してください。マップがほとんど更新される場合は、照会パフォーマンスの改善と、マップに対する索引付けオーバーヘッドとのバランスを取る必要があります。詳しくは、125 ページの『索引付け』を参照してください。

Plain Old Java Object (POJO) がマップ内に保管されている場合、適切に索引付けすることによって、Java リフレクションを回避できます。次の例では、予算フィールドに索引が作成済みである場合、照会は WHERE 文節を範囲見出し検索と置換します。それ以外の場合、照会では、マップ全体をスキャンし、Java リフレクションを使用して最初に予算を取得してから、予算を値 50000 と比較することによって、WHERE 文節を評価します。

```
SELECT d FROM DeptBean d WHERE d.budget=50000
```

個別照会を最適に調整する方法、および各種の構文、オブジェクト・モデル、および索引が照会のパフォーマンスにどのように影響するかについて詳しくは、114 ページの『照会計画』を参照してください。

ページ編集の使用

クライアント/サーバー環境では、照会エンジンは、結果マップ全体をクライアントにトランスポートします。戻されるデータは、妥当なチャンクに分割される必要があります。EntityManager Query および ObjectMap ObjectQuery の両インターフェースは、結果のサブセットを戻すことを照会に許可する setFirstResult および setMaxResults メソッドをサポートします。

エンティティーの代わりにプリミティブ値を戻す

EntityManager Query API を使用すると、エンティティーは照会パラメーターとして戻されます。照会エンジンは、現在のところ、これらのエンティティーに対するキーをクライアントに戻します。クライアントが getResultIterator メソッドからの Iterator を使用して、これらのエンティティーを繰り返すとき、各エンティティーは、EntityManager インターフェース上の find メソッドで作成されたかのように、自動的に拡張され、管理されます。エンティティー・グラフ全体は、クライアント上のエンティティー ObjectMap からビルドされます。エンティティー値属性およびその他の関連エンティティーは、可能な限り解決されます。

コストのかかるグラフのビルドを回避するには、パス・ナビゲーションを使用して個々の属性を戻すように照会を変更してください。

例:

```
// Returns an entity
SELECT p FROM Person p
// Returns attributes SELECT p.name, p.address.street, p.address.city, p.gender FROM Person p
```

索引を使用した照会の最適化

索引を適切に定義および使用すると、照会のパフォーマンスをかなり改善できます。

WebSphere eXtreme Scale 照会では、組み込み HashIndex プラグインを使用すると、照会のパフォーマンスを改善できます。索引は、エンティティーまたはオブジェクト属性に対して定義できます。照会エンジンは、その WHERE 文節で以下のいずれかのストリングが使用されると、定義された索引を自動的に使用します。

- 以下の演算子を使用する比較式: =、<、>、<=、または >= (等しくない <> を除くすべての比較式)
- BETWEEN 式
- 式のオペランドが定数またはシンプル・ターム

要件

照会で使用される場合、索引には以下の要件があります。

- すべての索引は組み込み HashIndex プラグインを使用する必要があります。
- すべての索引は静的に定義されていなければなりません。動的索引はサポートされません。
- 自動的に静的 HashIndex プラグインを作成するために @Index アノテーションを使用できます。

- すべての単一属性索引の RangeIndex プロパティーは true に設定されていなければなりません。
- すべての複合索引の RangeIndex プロパティーは false に設定されていなければなりません。
- すべてのアソシエーション (リレーションシップ) 索引の RangeIndex プロパティーは false に設定されていなければなりません。

キャッシュされたオブジェクトを検索するためのより効果的な方法については、131 ページの『複合 HashIndex』を参照してください。

索引選択に関するヒントの使用

索引は、HINT_USEINDEX 定数付きの setHint メソッドを Query および ObjectQuery インターフェースで使用すると、手動で選択することができます。これは、最も効率的な索引を使用するよう照会を最適化する際に役立ちます。

属性索引を使用する照会例

以下の例では、シンプル・ターム e.empid、e.name、e.salary、d.name、d.budget、および e.isManager が使用されています。これらの例では、索引がエンティティーまたは値オブジェクトの名前、給与、および予算フィールドに対して定義済みであることを前提としています。empid フィールドは 1 次キーであり、isManager には索引が定義されていません。

以下の照会では、名前と給与の両フィールドに対して索引を使用します。この場合、名前が最初のパラメーターの値に一致するか、給与が 2 番目のパラメーターの値に一致するすべての従業員が戻されます。

```
SELECT e FROM EmpBean e where e.name=?1 or e.salary=?2
```

以下の照会では、名前と予算の両フィールドに対して索引を使用します。この照会は、2000 より大きい予算を持つ 'DEV' という名前の付いたすべての部門を戻します。

```
SELECT d FROM DeptBean dwhere d.name='DEV' and d.budget>2000
```

以下の照会では、給与が 3000 より高く、かつパラメーターの値と等しい isManager フラグ値を持つ従業員をすべて戻します。この照会では、給与フィールドに対して定義された索引を使用するとともに、比較式 `e.isManager=?1.` を評価することで追加のフィルタリングを実行します。

```
SELECT e FROM EmpBean e where e.salary>3000 and e.isManager=?1
```

次の照会では、1 番目のパラメーターより大きい給与を得ているか、または管理者である従業員をすべて検索します。給与フィールドには索引が定義済みですが、照会では、EmpBean フィールドの 1 次キーに対して作成された組み込み索引をスキャンし、式 `e.salary>?1` または `e.isManager=TRUE` を評価します。

```
SELECT e FROM EmpBean e WHERE e.salary>?1 or e.isManager=TRUE
```

以下の照会では、文字 a が含まれている名前の従業員を戻します。名前フィールドには索引が定義済みですが、名前フィールドが LIKE 式で使用されているため、照会ではこの索引を使用しません。

```
SELECT e FROM EmpBean e WHERE e.name LIKE '%a%'
```

以下の照会では、名前が「Smith」ではない従業員をすべて検索します。名前フィールドには索引が定義済みですが、照会では等しくない (<>) 比較演算子を使用するため、この索引を使用しません。

```
SELECT e FROM EmpBean e where e.name<>'Smith'
```

以下の照会では、予算がパラメーターの値より小さく、かつ従業員給与が 3000 より大きい部門をすべて検索します。この照会では、給与の索引を使用しますが、dept.budget がシンプル・タームではないため、予算の索引を使用しません。dept オブジェクトは、コレクション e から導き出されます。dept オブジェクトを検索するのに、予算の索引を使用する必要はありません。

```
SELECT dept from EmpBean e, in (e.dept) dept where e.salary>3000 and
dept.budget<?
```

以下の照会では、1、2、および 3 の empid を持つ従業員の給与より大きい給与の従業員をすべて検索します。比較には副照会が含まれているため、索引 salary は使用されません。empid は、1 次キーですが、すべての 1 次キーには組み込み索引が定義済みであるため、固有索引の検索に使用されます。

```
SELECT e FROM EmpBean e WHERE e.salary > ALL (SELECT e1.salary    FROM
EmpBean e1 WHERE e1.empid=1 or e1.empid =2 or e1.empid=99)
```

索引が照会で使用されているかどうかを確認する場合は、114 ページの『照会計画』を表示できます。以下に、前述の照会の照会計画例を示します。

```
for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( q2.salary >ALL     temp collection defined as
       IteratorUnionIndex of

         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(1)
        )

         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(2)
        )

         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(99)
        )
       returning new Tuple( q3.salary )
   returning new Tuple( q2  )

for q2 in EmpBean ObjectMap using RANGE INDEX on salary with range(3000,)
     for q3 in  q2.dept
      filter ( q3.budget <  ?1 )
   returning new Tuple( q3  )
```

属性の索引付け

前に定義された制約付きで、任意の単一属性タイプに対して索引を定義できます。

@Index を使用したエンティティー索引の定義

エンティティーに索引を定義するには、単にアノテーションを定義します。

Entities using annotations

```
  @Entity
  public class Employee {
  @Id int empid;
  @Index String name
  @Index double salary
  @ManyToOne Department dept;
}
@Entity
  public class Department {
  @Id int deptid;
  @Index String name;
  @Index double budget;
  boolean isManager;
  @OneToMany Collection<Employee> employees;
  }
```

XML の使用

XML を使用して索引を定義することもできます。

Entities without annotations

```
  public class Employee {
  int empid;
  String name
  double salary
  Department dept;
  }

  public class Department {
  int deptid;
  String name;
  double budget;
  boolean isManager;
  Collection employees;
  }
```

ObjectGrid XML with attribute indexes

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
  <objectGrid name="DepartmentGrid" entityMetadataXMLFile="entity.xml>
  <backingMap name="Employee" pluginCollectionRef="Emp"/>
  <backingMap name="Department" pluginCollectionRef="Dept"/>
  </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
  <backingMapPluginCollection id="Emp">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.salary"/>
  <property name="AttributeName" type="java.lang.String" value="salary"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  <backingMapPluginCollection id="Dept">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.budget"/>
```

```
  <property name="AttributeName" type="java.lang.String" value="budget"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  </backingMapPluginCollections>
  </objectGridConfig>
```

Entity XML

```
<?xml version="1.0" encoding="UTF-8"?>
  <entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

  <description>Department entities</description>
  <entity class-name="acme.Employee" name="Employee" access="FIELD">
  <attributes>
  <id name="empid" />
  <basic name="name" />
  <basic name="salary" />
  <many-to-one name="department"
  target-entity="acme.Department"
  fetch="EAGER">
  <cascade><cascade-persist/></cascade>
  </many-to-one>
  </attributes>
  </entity>
  <entity class-name="acme.Department" name="Department" access="FIELD">
  <attributes>
  <id name="deptid" />
  <basic name="name" />
  <basic name="budget" />
  <basic name="isManager" />
  <one-to-many name="employees"
  target-entity="acme.Employee"
  fetch="LAZY" mapped-by="parentNode">
  <cascade><cascade-persist/></cascade>
  </one-to-many>
  </attributes>
  </entity>
  </entity-mappings>
```

XML を使用した非エンティティーの索引の定義

非エンティティー・タイプに対する索引は XML 内で定義されます。MapIndexPlugin を作成するときに、エンティティー・マップに対しての場合と非エンティティー・マップに対しての場合で相違はありません。

Java bean

```
public class Employee {
  int empid;
  String name
  double salary
  Department dept;

 public class Department {
  int deptid;
  String name;
  double budget;
  boolean isManager;
  Collection employees;
  }
```

ObjectGrid XML with attribute indexes

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
  <objectGrid name="DepartmentGrid">
  <backingMap name="Employee" pluginCollectionRef="Emp"/>
  <backingMap name="Department" pluginCollectionRef="Dept"/>
  <querySchema>
  <mapSchemas>
  <mapSchema mapName="Employee" valueClass="acme.Employee"
  primaryKeyField="empid" />
  <mapSchema mapName="Department" valueClass="acme.Department"
  primaryKeyField="deptid" />
  </mapSchemas>
  <relationships>
```

```
<relationship source="acme.Employee"
target="acme.Department"
relationField="dept" invRelationField="employees" />
</relationships>
</querySchema>
</objectGrid>
</objectGrids>
<backingMapPluginCollections>
<backingMapPluginCollection id="Emp">
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Employee.name"/>
<property name="AttributeName" type="java.lang.String" value="name"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Employee.salary"/>
<property name="AttributeName" type="java.lang.String" value="salary"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
</backingMapPluginCollection>
<backingMapPluginCollection id="Dept">
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Department.name"/>
<property name="AttributeName" type="java.lang.String" value="name"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Department.budget"/>
<property name="AttributeName" type="java.lang.String" value="budget"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
</backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

索引付けのリレーションシップ

WebSphere eXtreme Scale は、関連エンティティーの外部キーを親オブジェクト内に保管します。エンティティーの場合、キーは基本となるタプルに保管されます。非エンティティー・オブジェクトの場合、キーは親オブジェクトに明示的に保管されます。

リレーションシップ属性に索引を追加すると、循環参照を使用するか、IS NULL、IS EMPTY、SIZE、および MEMBER OF 照会フィルターを使用する照会をスピードアップすることができます。単一値関連と多値関連がともに、ObjectGrid 記述子 XML ファイル内に @Index アノテーションまたは HashIndex プラグイン構成を持つ場合があります。

@Index を使用したエンティティー・リレーションシップ索引の定義

以下の例では、@Index アノテーションのあるエンティティーを定義します。

Entity with annotation

```
@Entity
public class Node {
    @ManyToOne @Index
    Node parentNode;

    @OneToMany @Index
    List<Node> childrenNodes = new ArrayList();

    @OneToMany @Index
    List<BusinessUnitType> businessUnitTypes = new ArrayList();
}
```

XML を使用したエンティティー・リレーションシップ索引の定義

以下の例は、XML と HashIndex プラグインを使用して、同じエンティティーおよび索引を定義しています。

Entity without annotations

```
public class Node {
int nodeId;
Node parentNode;
List<Node> childrenNodes = new ArrayList();
List<BusinessUnitType> businessUnitTypes = new ArrayList();
}
```

ObjectGrid XML

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
  <objectGrid name="ObjectGrid_Entity" entityMetadataXMLFile="entity.xml">
  <backingMap name="Node" pluginCollectionRef="Node"/>
  <backingMap name="BusinessUnitType" pluginCollectionRef="BusinessUnitType"/>
  </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
  <backingMapPluginCollection id="Node">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="parentNode"/>
  <property name="AttributeName" type="java.lang.String" value="parentNode"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="businessUnitType"/>
  <property name="AttributeName" type="java.lang.String" value="businessUnitTypes"/>

<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
 </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="childrenNodes"/>
  <property name="AttributeName" type="java.lang.String" value="childrenNodes"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
  </bean>
  </backingMapPluginCollection>
  </backingMapPluginCollections>
  </objectGridConfig>
```

Entity XML

```
  <?xml version="1.0" encoding="UTF-8"?>
  <entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

  <description>My entities</description>
  <entity class-name="acme.Node" name="Account" access="FIELD">
  <attributes>
  <id name="nodeId" />
  <one-to-many name="childrenNodes"
  target-entity="acme.Node"
  fetch="EAGER" mapped-by="parentNode">
  <cascade><cascade-all/></cascade>
  </one-to-many>
  <many-to-one name="parentNodes"
  target-entity="acme.Node"
  fetch="LAZY" mapped-by="childrenNodes">
  <cascade><cascade-none/></cascade>
  </one-to-many>
  <many-to-one name="businessUnitTypes"
  target-entity="acme.BusinessUnitType"
  fetch="EAGER">
  <cascade><cascade-persist/></cascade>
  </many-to-one>
</attributes>
  </entity>
  <entity class-name="acme.BusinessUnitType" name="BusinessUnitType" access="FIELD">
  <attributes>
  <id name="buId" />
  <basic name="TypeDescription" />
  </attributes>
  </entity>
  </entity-mappings>
```

前に定義された索引を使用すると、以下の例のエンティティー照会が最適化されます。

```
SELECT n FROM Node n WHERE n.parentNode is null
SELECT n FROM Node n WHERE n.businessUnitTypes is EMPTY
  SELECT n FROM Node n WHERE size(n.businessUnitTypes)>=10
  SELECT n FROM BusinessUnitType b, Node n WHERE b member of n.businessUnitTypes and b.name='TELECOM'
```

非エンティティー・リレーションシップ索引の定義

以下の例では、ObjectGrid 記述子 XML ファイル内の非エンティティー・マップの HashIndex プラグインを定義します。

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="ObjectGrid_POJO">
      <backingMap name="Node" pluginCollectionRef="Node"/>
      <backingMap name="BusinessUnitType" pluginCollectionRef="BusinessUnitType"/>
      <querySchema>
        <mapSchemas>
          <mapSchema mapName="Node"
     valueClass="com.ibm.websphere.objectgrid.samples.entity.Node"
            primaryKeyField="id" />
          <mapSchema mapName="BusinessUnitType"
            valueClass="com.ibm.websphere.objectgrid.samples.entity.BusinessUnitType"
            primaryKeyField="id" />
        </mapSchemas>
        <relationships>
          <relationship source="com.ibm.websphere.objectgrid.samples.entity.Node"
            target="com.ibm.websphere.objectgrid.samples.entity.Node"
            relationField="parentNodeId" invRelationField="childrenNodeIds" />
          <relationship source="com.ibm.websphere.objectgrid.samples.entity.Node"
            target="com.ibm.websphere.objectgrid.samples.entity.BusinessUnitType"
            relationField="businessUnitTypeKeys" invRelationField="" />
        </relationships>
      </querySchema>
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="Node">
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="parentNode"/>
<property name="Name" type="java.lang.String" value="parentNodeId"/>
<property name="AttributeName" type="java.lang.String" value="parentNodeId"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
      </bean>
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="businessUnitType"/>
        <property name="AttributeName" type="java.lang.String" value="businessUnitTypeKeys"/>

<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
   </bean>
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="childrenNodeIds"/>
  <property name="AttributeName" type="java.lang.String" value="childrenNodeIds"/>
  <property name="RangeIndex" type="boolean" value="false"
      description="Ranges are not supported for association indexes." />
</bean>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

上記の索引構成が指定されると、以下の例のオブジェクト照会が最適化されます。

```
SELECT n FROM Node n WHERE n.parentNodeId is null
SELECT n FROM Node n WHERE n.businessUnitTypeKeys is EMPTY
SELECT n FROM Node n WHERE size(n.businessUnitTypeKeys)>=10
SELECT n FROM BusinessUnitType b, Node n WHERE
 b member of n.businessUnitTypeKeys and b.name='TELECOM'
```

照会計画

すべての eXtreme Scale 照会には照会計画があります。この計画は、照会エンジンが ObjectMap および索引とどのように対話するかを説明するものです。照会計画を表示すると、照会ストリングまたは索引が適切に使用されているかどうかを判断で

きます。また照会計画を使用すると、照会ストリング中のわずかな変更が eXtreme Scale による照会の実行方法に及ぼす変化を検討することもできます。

照会計画は、以下のいずれかの手段で表示できます。

- EntityManager Query または ObjectQuery の getPlan API メソッド
- ObjectGrid 診断トレース

getPlan メソッド

ObjectQuery および Query インターフェースの getPlan メソッドは、照会計画を説明するストリングを戻します。このストリングは、標準出力で表示することも、照会計画を表示するためのログで表示することもできます。注: 分散環境では、getPlan メソッドは、サーバーに対して実行されず、定義された索引を示しません。計画を表示するには、エージェントを使用して、サーバー上でその計画を表示します。

照会計画トレース

照会計画は、ObjectGrid トレースを使用して表示できます。照会計画トレースを有効とするには、以下のトレース仕様を使用します。

QueryEnginePlan=debug=enabled

トレース・ログ・ファイルを有効にする方法およびその検出方法について詳しくは、301 ページの『ログおよびトレース』を参照してください。

照会計画の例

この照会計画では、for という単語を使用して、この照会が ObjectMap コレクションで繰り返されるか、または派生するコレクション (q2.getEmps()、q2.dept、または内部ループによって返される一時的コレクションなど) で繰り返されることを示します。コレクションが ObjectMap のコレクションである場合、照会計画は、順次スキャン (INDEX SCAN で指示) や固有または非固有の索引が使用されているかどうかを示します。また、照会計画ではフィルター・ストリングを使用して、コレクションに適用される条件式をリストします。

通常、デカルト積は対象照会では使用されません。以下の照会では、外部ループ内の EmpBean マップ全体をスキャンし、内部ループ内の DeptBean マップ全体をスキャンします。

```
SELECT e, d FROM EmpBean e, DeptBean d

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in DeptBean ObjectMap using INDEX SCAN
   returning new Tuple( q2,  q3  )
```

以下の照会では、EmpBean マップを順次スキャンして特定部門の全従業員名を検索し、従業員オブジェクトを取得します。この照会では、従業員オブジェクトからその部門オブジェクトにナビゲートして、d.no=1 フィルターを適用します。この例の場合、各従業員はただ 1 つの部門オブジェクト参照を持つため、内部ループが 1 回実行されます。

```
SELECT e.name FROM EmpBean e JOIN e.dept d WHERE d.no=1

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in  q2.dept
      filter ( q3.getNo() = 1 )
   returning new Tuple( q2.name  )
```

以下の照会は、前記の照会と同等です。ただし、以下の照会では、まず DeptBean 1 次キー・フィールド番号に対して定義された固有索引を使用することで、結果が 1 つの部門オブジェクトに絞り込まれるため、実行効率が高まります。照会により、この部門オブジェクトから従業員オブジェクトにナビゲートされ、以下のように従業員名が取得されます。

```
SELECT e.name FROM DeptBean d JOIN d.emps e WHERE d.no=1

Plan trace:

for q2 in DeptBean ObjectMap using UNIQUE INDEX key=(1)
     for q3 in  q2.getEmps()
   returning new Tuple( q3.name  )
```

以下の照会を使用して、開発または販売に従事するすべての従業員を検索します。この照会では、EmpBean マップ全体をスキャンするとともに、式 d.name = 'Sales' or d.name='Dev' を評価することで追加のフィルタリングを実行します。

```
SELECT e FROM EmpBean e, in (e.dept) d WHERE d.name = 'Sales'
 or d.name='Dev'

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in  q2.dept
      filter (( q3.getName() = Sales ) OR ( q3.getName() = Dev ) )
   returning new Tuple( q2  )
```

以下の照会は前記の照会と同等ですが、この照会では異なる照会計画を実行し、フィールド名について作成された範囲索引を使用します。一般的に、部門オブジェクトの範囲の絞り込みに名前フィールドの索引が使用されることにより、開発または販売部門がごく少数である場合は照会が高速実行されるため、この照会の方が性能が高くなります。

```
SELECT e FROM DeptBean d, in(d.emps) e WHERE d.name='Dev' or d.name='Sales'

Plan trace:

IteratorUnionIndex of

     for q2 in DeptBean ObjectMap using INDEX on name = (Dev)
       for q3 in  q2.getEmps()


     for q2 in DeptBean ObjectMap using INDEX on name = (Sales)
       for q3 in  q2.getEmps()
```

以下の照会を使用して、従業員のいない部門を検索します。

```
SELECT d FROM DeptBean d WHERE NOT EXISTS(select e from d.emps e)

Plan trace:

for q2 in DeptBean ObjectMap using INDEX SCAN
```

```
    filter ( NOT  EXISTS (    correlated collection defined as

       for q3 in  q2.getEmps()
       returning new Tuple( q3      )

   returning new Tuple( q2  )
```

以下の照会は前述の照会と同等ですが、この照会では SIZE スカラー関数が使用されます。この照会でパフォーマンスは同じですが、作成が容易になっています。

```
SELECT d FROM DeptBean d WHERE SIZE(d.emps)=0
for q2 in DeptBean ObjectMap using INDEX SCAN
    filter (SIZE( q2.getEmps()) = 0 )
   returning new Tuple( q2  )
```

以下の例は、同様の性能を持つ前述の照会と同じ照会を書き込む別の方法を示していますが、この照会も容易に書き込むことができます。

```
SELECT d FROM DeptBean d WHERE d.emps is EMPTY

Plan trace:

for q2 in DeptBean ObjectMap using INDEX SCAN
    filter ( q2.getEmps() IS EMPTY  )
   returning new Tuple( q2  )
```

以下の照会では、パラメーターの値と等しい名前を持つ従業員の住所のうち少なくとも 1 つと一致する住所を持つすべての従業員を検索します。内部ループは外部ループに依存関係を持ちません。この照会では、内部ループは 1 回のみ実行されます。

```
SELECT e FROM EmpBean e WHERE e.home =  any (SELECT e1.home FROM EmpBean e1
 WHERE e1.name=?1)
for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( q2.home =ANY     temp collection defined as

       for q3 in EmpBean ObjectMap using INDEX on name = ( ?1)
       returning new Tuple( q3.home      )
 )
   returning new Tuple( q2  )
```

以下の照会は前述の照会と同等ですが、この照会には相関副照会があり、さらに内部ループが繰り返し実行されます。

```
SELECT e FROM EmpBean e WHERE EXISTS(SELECT e1 FROM EmpBean e1 WHERE
 e.home=e1.home and e1.name=?1)

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( EXISTS (    correlated collection defined as

       for q3 in EmpBean ObjectMap using INDEX on name = (?1)
        filter ( q2.home =  q3.home )
       returning new Tuple( q3      )

   returning new Tuple( q2  )
```

第 9 章 トラブルシューティング

eXtreme Scale のメモリー内データ・グリッドの構成をトラブルシューティングするには、ログとトレース、メッセージ、およびリリース情報を使用できます。

関連概念

XML 構成のトラブルシューティング

ログおよびトレース

ログおよびトレースを使用して、環境のモニターおよびトラブルシューティングを実行できます。ログは、構成によってさまざまなロケーションにあります。IBM サポートに協力を依頼する場合、サーバーに関するトレースを提供する必要がある場合があります。

WebSphere Application Server によるロギング

詳しくは、WebSphere Application Server インフォメーション・センターを参照してください。

スタンドアロン環境での WebSphere eXtreme Scale によるロギング

スタンドアロン・カタログおよびコンテナー・サービスを使用して、ログのロケーションおよびトレース仕様を設定します。カタログ・サーバー・ログは、サーバー始動コマンドを実行したロケーションにあります。

コンテナー・サーバーのログ・ロケーションの設定

デフォルトでは、コンテナーのログは、サーバー・コマンドが実行されたディレクトリーにあります。*<eXtremeScale_home>*/bin ディレクトリーでサーバーを始動する場合、ログおよびトレース・ファイルは bin ディレクトリーの logs/*<server_name>* ディレクトリー内にあります。コンテナー・サーバー・ログの代替ロケーションを指定するには、以下のコンテンツを使用して server.properties ファイルなどのプロパティー・ファイルを作成します。

```
workingDirectory=<directory>
traceSpec=
systemStreamToFileEnabled=true
```

workingDirectory プロパティーは、ログおよびオプションのトレース・ファイルのルート・ディレクトリーです。WebSphere eXtreme Scale は、traceSpec オプションでトレースが使用可能になっていると、SystemOut.log ファイル、SystemErr.log ファイル、およびトレース・ファイルを使用して、コンテナー・サーバーの名前を持つディレクトリーを作成します。コンテナー開始中にプロパティー・ファイルを使用するには、**-serverProps** オプションを使用して、サーバー・プロパティー・ファイルのロケーションを指定します。

SystemOut.log ファイル内で参照する共通の情報メッセージは、開始確認メッセージです。特定のメッセージについて詳しくは、 305 ページの『メッセージ』を参照してください。

WebSphere Application Server によるトレース

詳しくは、WebSphere Application Server インフォメーション・センターを参照してください。

カタログ・サービスでのトレース

カタログ・サービスの始動中に、 **-traceSpec** および **-traceFile** パラメーターを使用して、カタログ・サービスでトレースを設定できます。以下に例を示します。

```
startOgServer.sh catalogServer -traceSpec
ObjectGridPlacement=all=enabled -traceFile
/home/user1/logs/trace.log
```

<eXtremeScale_home>/bin ディレクトリーでカタログ・サービスを始動する場合、ログおよびトレース・ファイルは bin ディレクトリーの logs/*<catalog_service_name>* ディレクトリー内にあります。 *管理ガイド*にあるスタンドアロン環境でのカタログ・サービス・プロセスの開始に関する情報を参照してください。

スタンドアロン・コンテナー・サーバーでのトレース

コンテナー・サーバーでトレースを使用可能にするには 2 つの方法があります。ログ・セクションでの説明どおりに、サーバー・プロパティー・ファイルを作成するか、始動時にコマンド行を使用してトレースを使用可能にすることができます。サーバー・プロパティー・ファイルを使用してコンテナー・トレースを使用可能にするには、必要なトレース仕様で **traceSpec** プロパティーを更新します。始動パラメーターを使用して、コンテナー・トレースを使用可能にするには、**-traceSpec** および **-traceFile** パラメーターを使用します。以下に例を示します。

```
startOgServer.sh c0 -objectGridFile ../xml/myObjectGrid.xml
-deploymentPolicyFile ../xml/myDepPolicy.xml -catalogServiceEndpoints
server1.rchland.ibm.com:2809 -traceSpec
ObjectGridPlacement=all=enabled -traceFile /home/user1/logs/trace.log
```

<eXtremeScale_home>/bin ディレクトリーでサーバーを始動する場合、ログおよびトレース・ファイルは bin ディレクトリーの logs/*<server_name>* ディレクトリー内にあります。

詳しくは、

ObjectGridManager インターフェースによるトレース

別の方法として、ObjectGridManager インターフェースで実行時にトレースを設定する方法があります。 ObjectGridManager インターフェースでのトレース設定を使用すると、eXtreme Scale に接続してトランザクションをコミットしている間に eXtreme Scale クライアント上でトレースを取得することができます。ObjectGridManager インターフェースでトレースを設定するには、トレース仕様およびトレース・ログを指定します。

```
ObjectGridManager manager = ObjectGridManagerFactory.getObjectGridManager();
...
manager.setTraceEnabled(true);
manager.setTraceFileName("logs/myClient.log");
manager.setTraceSpecification("ObjectGridReplication=all=enabled");
```

xsadmin ユーティリティーでトレースを使用可能にする

xsadmin ユーティリティーを使用してトレースを使用可能にする場合、**setTraceSpec** オプションを使用します。 xsadmin ユーティリティーを使用して、開始時ではなく実行時にスタンドアロン環境でトレースを使用可能にすることができます。すべてのサーバーおよびカタログ・サービスに対してトレースを使用可能にすることができます。あるいは、ObjectGrid 名などでサーバーをフィルタリングすることもできます。例えば、カタログ・サービス・サーバーへのアクセスを使用して ObjectGridReplication トレースを使用可能にするには、以下を実行します。

```
<eXtremeScale_home>/bin>xsadmin.bat -setTraceSpec "ObjectGridReplication=all=enabled"
```

トレース仕様を `*=all=disabled` に設定することでトレースを使用不可にすることもできます。

詳しくは *管理ガイド*にある xsAdmin ユーティリティーに関する情報を参照してください。

ffdc ディレクトリーおよびファイル

FFDC ファイルは、IBM サポートがデバッグの補助とするファイルです。これらのファイルは、問題が生じた場合に IBM サポートによって要求される場合があります。

これらのファイルは、`ffdc` という名前のディレクトリーに存在し、以下のファイルに類似したファイルが含まれています。

```
server2_exception.log
server2_20802080_07.03.05_10.52.18_0.txt
```

トレース・オプション

トレースを使用可能にすることで、ご使用の環境に関する情報を IBM サポートに提供することができます。

トレースについて

WebSphere eXtreme Scale のトレースは、いくつかの異なるコンポーネントに分けられます。 WebSphere Application Server のトレースと同様、使用するトレース・レベルを指定することができます。一般的なトレースのレベルには、all、debug、entryExit、および event があります。

トレース・ストリングの例は、以下のとおりです。

```
ObjectGridComponent=level=enabled
```

トレース・ストリングは連結することができます。`*` (アスタリスク) 記号を使用してワイルドカード値を指定します (例: `ObjectGrid*=all=enabled`)。トレースを IBM サポートに提供する必要がある場合は、特定のトレース・ストリングが要求されます。 例えば、複製に関する問題が発生した場合には、

ObjectGridReplication=debug=enabled トレース・ストリングが要求される可能性があります。

トレース仕様

ObjectGrid
汎用・コア・キャッシュ・エンジン。

ObjectGridCatalogServer
汎用カタログ・サービス。

ObjectGridChannel
静的デプロイメント・トポロジー通信。

ObjectgridCORBA
動的デプロイメント・トポロジー通信。

ObjectGridDataGrid
AgentManager API。

ObjectGridDynaCache
WebSphere eXtreme Scale 動的キャッシュ・プロバイダー

ObjectGridEntityManager
EntityManager API。Projector オプションとともに使用。

ObjectGridEvictors
ObjectGrid 組み込み Evictor。

ObjectGridJPA
Java Persistence API (JPA) ローダー

ObjectGridJPACache
JPA キャッシュ・プラグイン

ObjectGridLocking
ObjectGrid キャッシュ・エントリー・ロック・マネージャー。

ObjectGridMBean
管理 Bean

ObjectGridPlacement
カタログ・サーバー断片配置サービス。

ObjectGridQuery
ObjectGrid 照会。

ObjectGridReplication
レプリケーション・サービス。

ObjectGridRouting
クライアント/サーバー・ルーティングの詳細。

ObjectGridSecurity
セキュリティー・トレース。

ObjectGridStats
ObjectGrid 統計。

ObjectGridStreamQuery
ストリーム照会 API。

ObjectGridWriteBehind
ObjectGrid 後書き。

Projector
EntityManager API 内のエンジン。

QueryEngine
オブジェクト照会 API および EntityManager 照会 API のための照会エンジン。

QueryEnginePlan
照会計画診断。

メッセージ

製品インターフェースのログまたはその他の部分にメッセージが表示された場合は、そのコンポーネントの接頭部でメッセージを検索して、詳細情報を確認してください。

メッセージの検索

ログにメッセージが表示された場合、そのメッセージ番号を (接頭部の文字と番号と共に) コピーして、インフォメーション・センターで検索します (例えば、CWOBJ1526I)。メッセージを検索すると、そのメッセージの詳細説明や、問題解決のために実行できる可能性のあるアクションを確認できます。

製品メッセージの索引については、インフォメーション・センターを参照してください。

リリース情報

製品のサポート Web サイト、製品資料、および製品の最新の更新、制限、および既知の問題へのリンクが提供されています。

- 『最新の更新、制限、および既知の問題へのアクセス』
- 306 ページの『システムのアクセスおよびソフトウェア要件』
- 306 ページの『製品資料へのアクセス』
- 306 ページの『製品のサポート Web サイトへのアクセス』
- 306 ページの『IBM ソフトウェア・サポートへの連絡』

最新の更新、制限、および既知の問題へのアクセス

リリース情報は、製品のサポート・サイトの技術情報から入手できます。
WebSphere eXtreme Scale のすべての技術情報のリストを参照するには、サポート Web ページにアクセスしてください。

- バージョン 7.0 のリリース情報のリストを参照するには、サポート Web ページにアクセスしてください。
- バージョン 6.1 のリリース情報のリストを参照するには、リリース情報ウィキ・ページにアクセスしてください。

システムのアクセスおよびソフトウェア要件

ハードウェアおよびソフトウェア要件は以下のページに記載されています。

- システム要件の詳細

製品資料へのアクセス

情報セット全体に関しては、ライブラリー・ページにアクセスしてください。

製品のサポート Web サイトへのアクセス

最新の技術情報、ダウンロード、フィックス、およびその他のサポート関連情報を検索するには、 サポート・ページにアクセスしてください。

IBM ソフトウェア・サポートへの連絡

この製品で問題が発生した場合には、最初に以下のアクションを試行してください。

- 製品資料に記載されているステップを実行します。
- 関連資料をオンライン・ヘルプで検索します。
- エラー・メッセージをメッセージ解説書で検索します。

上記の方法で問題を解決できない場合、IBM テクニカル・サポートに連絡してください。

第 10 章 用語集

この用語集には、WebSphere eXtreme Scale の用語と定義が含まれています。

この用語集では以下の相互参照が使用されています。

1. 「からを参照」は、ある用語から使用が推奨される同義語への参照、または頭字語あるいは省略語から完全な表現形式の定義への参照を読者に指示します。
2. 「からも参照」は、関連用語または対比用語を読者に示すものです。

他の IBM 製品の用語集を表示するには、www.ibm.com/software/globalization/terminology を参照してください。

アップグレード可能ロック. ペシミスティック・ロックを使用する場合に、キャッシュ・エントリーの更新意図を識別するロック。

アップストリーム (upstream). プロセスの開始 (アップストリーム) からプロセスの終了 (ダウンストリーム) へと流れる、フローの方向に関する用語。

宛先 (destination). バックエンド・システムまたは取引先に文書を配信するのに使用する出口点。

後書きキャッシュ (write-behind cache). ローダーを使用して、データベースに対する各書き込み操作が非同期に行われるキャッシュ。

アプリケーション (application). 特定のビジネス・プロセス (複数可) を直接サポートする機能を実現する 1 つ以上のコンピューター・プログラムまたはソフトウェア・コンポーネント。

アプリケーション・サーバー (application server). 分散ネットワーク内のサーバー・プログラムであり、アプリケーション・プログラムのための実行環境を提供する。

アプリケーション・プログラミング・インターフェース (API) (application programming interface (API)). 高水準言語で記述されたアプリケーション・プログラムがオペレーティング・システムまたは別のプログラムの特定のデータまたは機能を使用できるようにするインターフェース。

イテレーター (iterator). オブジェクトの集合を一度にステップスルーするために使用するクラスまたは構造。

イベント (event).

1. 操作、ビジネス・プロセス、またはヒューマン・タスクの完了または失敗などの状態の変更で、イベント・データのデータ・リポジトリーへの保管や、別のビジネス・プロセスを呼び出すなどの後続アクションをトリガーすることができる。

2. エンタープライズ情報システム (EIS) のデータに対する変更の 1 つ。アダプターによって処理され、EIS からのビジネス・オブジェクトを、変更を通知する必要があるエンドポイント (アプリケーション) に送信するために使用される。

インスタンス (instance). あるクラスに属するオブジェクトの特定のオカレンス。

インスタンス化 (instantiate). 抽象概念を具象化されたインスタンスで表現すること。

インストール・ターゲット (installation target). 選択されたインストール・パッケージがインストールされるシステム。

インストール・パッケージ (installation package). ソフトウェア製品のインストール可能単位。ソフトウェア製品パッケージは、そのソフトウェア製品の他のパッケージとは無関係に作動できる別々にインストール可能な単位である。

インターネット・プロトコル (Internet Protocol (IP)). 1 つのネットワークまたは相互接続ネットワークを介してデータを送付するプロトコル。このプロトコルは、高位プロトコル層と物理ネットワークの間の仲介として機能する。

インターフェース (interface). クラスのサービスまたはコンポーネントを指定するために使われる一連のオペレーションのこと。

インテリム・フィックス (interim fix). 正規にスケジュールされたフィックスパック、リフレッシュ・パック、またはリリースまでの間に、すべてのお客様で一般出荷可能になる認定された修正のこと。「フィックスパック (fix pack)」も参照。

インフォメーション・センター (information center). 製品に関する情報がまとめられており、複数製品へのサポートをユーザーに提供する。それぞれの製品から起動でき、ナビゲーション、検索エンジン、およびトピックのリストを表示するパネルが提供される。

インポート (import).

1. モジュール外部にあるサービスを取り込む(インポートする)ためのもの。

2. SCA モジュールが外部サービス (SCA モジュールにない外部サービス) に対して、ローカルであるかのようにしてアクセスするためのポイントとして定義される。インポートは、SCA モジュールとサービス・プロバイダー間のインターフェースを定義する。インポートには 1 つのバインディングと 1 つ以上のインターフェースが定義できる。

ウェイター (waiter). 接続を待機しているスレッド。

エージェント. ユーザーによる介入や定期スケジュールなしにユーザーまたは他のプログラムのためにアクションを実行し、結果をユーザーまたはプログラムに報告するプログラム。

永続データ・ストア (persistent data store). セッションの境界を越えて維持され、作成元のプログラムまたはプロセスの実行後にも継続して存在するイベント・データ用の不揮発性ストレージ (データベース・システムなど)。

エクスポート (export). Service Component Architecture (SCA) モジュールからの公開インターフェースで、モジュール外部にビジネス・サービスを提供する。エクスポートは、サービス・リクエスターにサービスへアクセスさせる方法 (例えば Web サービスとして) を定義するバインディングを持つ。

エクスポート・ファイル (export file).

1. インバウンド操作の開発過程で作成された、インバウンド処理の構成設定を含むファイル。

2. エクスポートしたデータを含むファイル。

エディション. 成果物セットの、特定バージョンにおける連続したデプロイメントの世代。

エディター領域 (editor area). Eclipse および Eclipse ベースの製品では、編集作業のためにファイルが開かれる、ワークベンチ・ウィンドウ内のエリア。

エラー (error). 値や状態が、計算したものと実際のものとの間で、監視したものと指定したものとの間で、あるいは測定したものと理論的に正しいものとの間で、一致しない状態。

エラー・ログ・ストリーム (error log stream). 事前定義フォーマットを使用して伝送されるエラー情報の連続フロー。

エンタープライズ Bean (enterprise bean). ビジネス・タスクまたはビジネス・エンティティーを実装し、EJB コンテナー内に常駐するコンポーネント。エンティティー Bean、セッション Bean、およびメッセージ駆動型 Bean はすべてエンタープライズ Bean である。Bean も参照。

エンタープライズ・アーカイブ (enterprise archive (EAR)). Java EE 標準で定義され、Java EE アプリケーションを Java EE アプリケーション・サーバーにデプロイするために使用される、特殊なタイプの JAR ファイル。EAR ファイルには、EJB コンポーネント、デプロイメント記述子、および個々の Web アプリケーション用の Web アーカイブ (WAR) ファイルが含まれる。「Web アーカイブ (Web archive)」も参照。

エンタープライズ・アプリケーション・プロジェクト (enterprise application project (EAR project)). デプロイメント記述子および IBM 拡張文書、デプロイメント記述子に定義されているすべての Java EE モジュールに共通のファイルを含むフォルダーやファイルの構造および階層。

エンタープライズ・サービス・バス (enterprise service bus (ESB)). アプリケーションとサービスを統合するための高い柔軟性を持つ接続インフラストラクチャー。柔軟で扱いやすいサービス指向アーキテクチャーの実装への手引きとなる。

エンティティー.

1. データベース表の行またはマップ内のエントリーを表わす単一の Java クラス。

2. XML などのマークアップ言語において、例えば文書内に頻繁に繰り返されるテキストや特殊文字を組み込むために、1 単位として参照できる文字の集合。

エンティティー Bean (entity bean). EJB プログラミングにおいて、データベース内で保守される永続的データを表すエンタープライズ Bean。各エンティティー Bean は独自の ID を持つ。

エンドポイント (endpoint).

1. JCA アプリケーションまたはエンタープライズ情報システムからのイベントを利用するその他のクライアント利用者。

2. セッションの発信元または宛先であるシステム。

エンドポイント・リスナー (endpoint listener). Web サービスの着信メッセージが、サービス統合バスで受信されるポイントまたはアドレス。

エントリー・ブレークポイント (entry breakpoint). コンポーネント・エレメントに設定されるブレークポイント。コンポーネント・エレメントが呼び出される前にヒットする。

オートディスカバリー (autodiscovery). ファイル・システム、外部レジストリー、またはその他のソース内のサービス成果物を発見すること。

オートノミック・マネージャー (autonomic manager). 他のソフトウェアまたはハードウェア・コンポーネントの動作を、人間が管理するように管理するポリシーによって構成された、一連のソフトウェアまたはハードウェア・コンポーネント。オートノミック・マネージャーには、モニター、分析、計画、実行の各コンポーネントからなる制御ループが組み込まれている。

オープン・ソース (open source). ソース・コードを公に使用または修正することができるソフトウェアに関する用語。通常、オープン・ソース・ソフトウェアは、公開のコラボレーションとして開発され、無償で使用可能にされる。ただし、その使用と再配布はライセンスの制約を受ける。 Linux は、オープン・ソース・ソフトウェアとしてよく知られている。

オブジェクト (object). オブジェクト指向設計およびプログラミングにおいて、データとそのデータに関連付けられた操作で構成される、1 つのクラスを具体的に実現したもの (インスタンス)。オブジェクトには、クラスで定義されたインスタンス・データが含まれているが、クラスはデータに関連付けられた操作を持っている。

オブジェクト・リクエスト・ブローカー (Object Request Broker (ORB)). オブジェクト指向プログラミングでは、オブジェクトが要求や応答を交換することを透過的に可能にすることによって、仲介としてサービスを提供するソフトウェアのこと。

オブジェクト指向プログラミング (object-oriented programming). データの抽象化と継承の概念に基づいたプログラミング・アプローチ。プロシージャー型プログラミング技法と異なり、オブジェクト指向プログラミングは、何らかのものを達成する方法に専念するのではなく、代わりに、問題がどのデータ・オブジェクトから構成されているか、およびそれらを操作する方法に専念する。

ガーベッジ・コレクション (garbage collection). プログラム・セグメントまたは非アクティブ・データのスペースを再利用するため、メモリーを検索するルーチン。

下位ノード (child node). 別のノードの有効範囲内にあるノード。

カスタマイズ・インストール・パッケージ (CIP). カスタマイズされたインストール・イメージ。1 つ以上の保守パッケージ、スタンドアロン・サーバー・プロファイルからの構成アーカイブ・ファイル、1 つ以上のエンタープライズ・アーカイブ・ファイル、スクリプト、および結果としてのインストールのカスタマイズに役立つその他のファイルを組み込むことができる。

仮想化 (virtualization). 他のシステムがリソースと対話する手段からリソースの特性をカプセル化する技法。

仮想ホスト (virtual host). 単一のホスト・マシンを複数のホスト・マシンのように機能させることが可能な構成。ある仮想ホストに関連付けられたリソースは、別の仮想ホストに関連付けられたリソースとデータを共有することはできない。このことは、これらの仮想ホストが同じ物理マシンを共有している場合であっても該当する。

仮想マシン (virtual machine). コンピューティング・デバイスの抽象的な仕様。さまざまな方法でソフトウェアおよびハードウェアに実装できる。

カタログ (catalog). コンテナーのタイプに基づいて、プロセス、データ、リソース、組織、またはレポートをプロジェクト・ツリーで保持するコンテナー。

カタログ・サービス. 断片の配置を制御し、コンテナーのヘルスをディスカバーおよびモニターするサービス。

カテゴリー (category). 共用の属性または品質に基づいてエレメントをグループ化する、構造ダイアグラムで使用されるコンテナー。

可用性.

1. ユーザーにアプリケーションとデータのアクセスおよび使用を許可する条件。

2. リソースがアクセス可能となる時間。例えばある請負業者のアベイラビリティーは、平日は毎日午前 9 時から午後 5 時、土曜日は午前 9 時から午後 3 時までとなる。

環境 (environment). 機能のパフォーマンスをサポートするのに使用される論理リソースおよび物理リソースの名前付きコレクション。

環境変数 (environment variable). オペレーティング・システムまたは他のプログラムの動作方法を指定する変数、あるいはオペレーティング・システムが認識するデバイスを指定する変数。

管理 Bean (MBean) (Managed Bean (MBean)). Java Management Extensions (JMX) 仕様において、リソースとそのインスツルメンテーションを実装する Java オブジェクト。

管理者 (administrator). アクセス許可およびコンテンツ・マネージメントなどの管理タスクの担当者。管理者は権限レベルをユーザーに付与することもできる。

キー.

1. 暗号の数値であり、メッセージをデジタル署名、検証、暗号化、または暗号化解除するために使用される。

2. モニター・コンテキストによってトラッキングされる実際のエンティティーを特徴づけ、一意的に識別するための情報。

キーワード (keyword). プログラミング言語、人工言語、アプリケーション、またはコマンドの事前定義語。

キャッシュ・インスタンス・リソース (cache instance resource). Java Platform, Enterprise Edition (Java EE) アプリケーションがデータを保管、配布、および共有できる場所。

キャッシュ複製 (cache replication). 同じ複製ドメイン内の別のサーバーとの、キャッシュ ID、キャッシュ・エントリー、およびキャッシュ無効化の共用。

共用ロック. 同時に実行するアプリケーション・プロセスを、データベース・データの読み取り専用操作に制限するロック。

許可 (authorization). ユーザー、システム、またはプロセスに、オブジェクト、リソース、または機能への完全なアクセス権限または制限付きのアクセス権限を付与するプロセス。

許可テーブル (authorization table). 特定のリソースに対してクライアントに許可されたアクセス権限を識別する、ユーザーのロールまたはグループ・マッピング情報を含むテーブル。

許可ポリシー (authorization policy). ビジネス・サービスをポリシー・ターゲットとするポリシーで、その契約には、チャネル・アクションを実行するための許可を付与する 1 つ以上のアサーションが含まれる。

区画化機能. エンタープライズ Bean、HTTP トラフィック、およびデータベース・アクセスの区画化の概念をサポートする、プログラミング・フレームワークおよびシステム管理インフラストラクチャー。

組み込みサーバー (embedded server). 既存のプロセス内に存在し、そのプロセス内で始動および停止されるカタログ・サービスまたはコンテナー・サーバー。

クライアント (client). サーバーに対してサービスを要求するソフトウェア・プログラムまたはコンピューター。「ホスト (host)」も参照。

クライアント/サーバー (client/server). 分散データ処理において、あるサイトのプログラムが他のサイトのプログラムへの要求を送信して応答を待つ形の対話モデル。要求を出すプログラムをクライアントと言い、応答するプログラムをサーバーと言う。

クライアント・アプリケーション (client application). ワークステーション上で実行されるクライアントにリンクするアプリケーションで、サーバーのキューイング・サービスを利用して、アプリケーションからのアクセスを提供する。

クラス. オブジェクト指向の設計またはプログラミングにおいて、オブジェクトを生成するために用いる、オブジェクトに共通の定義および共通のプロパティー、操作、振る舞いを持つモデルまたはテンプレート。オブジェクトは、クラスのインスタンスである。

クラス・ファイル (class file). Java ソース・ファイルをコンパイルして生成される中間コード・ファイル。

クラス・ローダー (class loader). クラス・ファイルの検索およびロードを行う Java 仮想マシン (JVM) のパーツ。クラス・ローダーは、アプリケーションのパッケージ化、およびアプリケーション・サーバーにデプロイされたパッケージ済みアプリケーションの実行時動作に影響を与える。

クラスター (cluster). ワークロード・バランシングおよびフェイルオーバーの目的で協調して動作する複数のアプリケーション・サーバーから構成されるグループ。

クラスパス (class path). プログラムが実行時に動的にロードできるリソース・ファイルまたは Java クラスを含むディレクトリーおよび JAR ファイルのリスト。

クラス階層 (class hierarchy). 単一継承を共有するクラス間の関係。

グループ.

1. 保護リソースに対するアクセス権限を共用できるユーザーの集合。

2. 交換内の関連する文書セット。交換には、多数のグループを含めることができる (グループを含めないことも可能)。

3. プレースでは、1 つのプレースでメンバーシップのためにグループになっている複数の人物。

クレデンシャル. Java 認証・承認サービス (JAAS) フレームワークにおいて、セキュリティー関連属性を所有するサブジェクト・クラス。これらの属性には、新規サービスに対してサブジェクトを認証するのに使用される情報を含めることができる。

グローバル (global).

1. ワークスペースの任意のプロセスが使用可能なエレメントに関する用語。グローバル・エレメントは、プロジェクト・ツリーに表示され、複数のプロセスで使用できる。タスク、プロセス、リポジトリー、およびサービスは、グローバル (プロジェクトの任意のプロセスによって参照される) かローカル (単一のプロセスに固有) のいずれかとなる。

2. 複数のプログラムまたはサブルーチンに有効な情報に関すること。

グローバル・インスタンス ID (global instance identifier). アプリケーションまたはエミッターによって生成され、イベント識別の 1 次キーとして使用されるグローバルな固有 ID。

グローバル・エレメント (global element). XML において、複合型定義の一部としてではなく、スキーマ・エレメントの子として宣言されるエレメント。グローバル・エレメントは、ref 属性を使用する 1 つ以上のコンテンツ・モデル内で参照できる。

グローバル・セキュリティー (global security). 環境内で実行されているすべてのアプリケーションに適用され、セキュリティーが使用されるかどうか、認証の際に使用されるレジストリーのタイプ、およびその他の値 (多くはデフォルトで動作する) を決定する。

グローバル・トランザクション (global transaction). 分散トランザクション環境で 1 つ以上のリソース・マネージャーによって実行され、外部トランザクション・マネージャーによって調整されるリカバリー可能な作業単位。

グローバル属性 (global attribute). XML において、複合型定義の一部としてではなく、スキーマ・エレメントの子として宣言される属性。グローバル属性は、ref 属性を使用する 1 つ以上のコンテンツ・モデル内で参照できる。

グローバル変数 (global variable). 変換中に割り当てられた値を保持および操作するために使用する変数。マップ間および文書変換の間で共用される。Data Interchange Services のマッピング・コマンド言語でサポートされる 3 つの変数タイプの一つである。

継承 (inheritance). 既存のクラスを他のクラスを作成するための基礎として使用するオブジェクト指向プログラミング技法。継承によって、より一般化されたエレメントの構造および動作が、より特殊化されたエレメントに組み込まれる。

結合 (join).

1. 決定または fork の後に並列処理のパスを再結合および同期化するプロセス要素。結合は、プロセスの続行を許可する前に、各着信ブランチで入力の到着を待機する。

2. 2 つの表から (通常、結合列を指定する結合条件に基づいて) データを取得できる SQL 関係演算。

3. リンクの動作を決定する着信リンク上の構成。

公開 (public).

1. オブジェクト指向プログラミングにおいて、すべてのクラスにアクセス可能なクラス・メンバーのこと。

2. Java プログラミング言語において、他のクラス内に存在するエレメントがアクセスできるメソッドまたは変数を指す。

高可用性 (high availability (HA)). ノードまたはデーモンに障害が発生した場合に、ワークロードをクラスター内に残っているノードに再配布できるように再構成されるクラスター化されたシステムを指す。

構造化照会言語 (SQL) (Structured Query Language (SQL)). リレーショナル・データベース内のデータを定義および操作するための標準化言語。

構文 (syntax). コマンドまたはステートメントを構成する際の規則。

コヒーレント・キャッシュ. すべてのクライアントが同一のデータを見れるよう、整合性を維持するキャッシュ。

コマンド Bean (command bean). execute() メソッドを使用して、単一操作を呼び出すことができるプロキシー。

コマンド行. コマンド、オプション番号、または選択を入力できるモニター上のブランク行。

コレクション証明書ストア (collection certificate store). 中間証明書または証明書取り消しリスト (CRL) のコレクション。検証のための証明書チェーンを構築するために証明書パスで使用される。

コンテナー・サーバー. 複数の断片をホスティングできるサーバー・インスタンス。1 台の Java 仮想マシン (JVM) は、複数のコンテナー・サーバーをホスティングできる。

コンバーター (converter). Enterprise JavaBeans (EJB) プログラミングでは、データベース表記をオブジェクト・タイプに (またはその逆に) 変換するクラス。

コンパイル時間 (compile time). コンピューター・プログラムを実行可能プログラムにコンパイルするための時間。

コンパイル単位 (compilation unit). プログラムのソース・コード群を分割してコンパイルする際に、分割された個々の部分。

コンポーネント (component).

1. 特定の機能を実行し、他のコンポーネントやアプリケーションと共に動作する、再使用可能なオブジェクトまたはプログラム。

2. Eclipse では、別個の機能セットを供給するために、一緒に動作する 1 つ以上のプラグイン。

コンポーネント・インスタンス (component instance). 同じコンポーネントの他のインスタンスと並列に実行できる実行コンポーネント。

コンポーネント・エレメント (component element). ビジネス・プロセスのアクティビティーまたは Java Snippet、あるいはメディエーション・フローのメディエーション・プリミティブまたはノードのような、ブレークポイントを設定できるコンポーネント内のエンティティー。

コンポーネント・テスト (component test). エンタープライズ・アプリケーションの 1 つ以上のコンポーネント (Java クラス、EJB Bean、または Web サービスが含まれる場合がある) の自動化されたテスト。

コンマ区切りファイル (comma delimited file). レコード内のフィールドがコンマで区切られているファイル。

サーバー (server). 別のソフトウェア・プログラムまたは別のコンピューターにサービスを提供するソフトウェア・プログラムまたはコンピューター。「ホスト (host)」も参照。

サーバー・クラスター (server cluster). 通常は別々の物理マシン上に配置され、内部に同じアプリケーションが構成されているが、単一の論理サーバーとして機能するサーバーのグループ。

サーバント領域 (servant region). 負荷が増大すると動的に開始し、負荷が軽減されると自動的に停止する仮想ストレージの連続区域。

サービス・レベル・アグリーメント (service level agreement (SLA)). 可用性やパフォーマンスなどの測定可能な目標に関して、期待されるサービス・レベルを明記した顧客とサービス・プロバイダー間の契約。

サービスの品質 (QoS) (quality of service (QoS)). アプリケーションが要求する一連の通信特性。サービスの品質 (QoS) は、特定の伝送優先順位、経路信頼性のレベル、およびセキュリティー・レベルを定義する。

サーブレット (servlet). Web サーバー上で稼働し、Web クライアントの要求に応答して動的コンテンツを生成することにより、サーバー機能を拡張する Java プログラム。一般に、サーブレットは、データベースを Web に接続するために使用される。

再帰 (recursion). プログラムまたはルーチンが自分自身を呼び出して、ある操作中で一連のステップを実行するプログラミング手法。この手法では、各ステップが前のステップからの出力内容を使用する。

サイレント・インストール. コンソールに対してメッセージは送信されず、メッセージおよびエラーがログ・ファイルに格納されるインストール。サイレント・インストールでは、データ入力に応答ファイルを使用できる。

サイレント・モード (silent mode). GUI 表示なしでコマンド行から製品コンポーネントをインストールまたはアンインストールする方法。サイレント・モードを使用する場合、インストールまたはアンインストール・プログラムで必要なデータは、コマンド行で直接指定するか、(オプション・ファイルまたは応答ファイルと呼ばれる) ファイル内に指定する。

索引. キーの値によって論理的に順序付けられているポインターのセット。索引を利用すると、データに迅速にアクセスでき、また表にある行のキー値の固有性を高めることができる。

サブクラス (subclass). Java において、特定のクラスから継承を通じて派生したクラス。

シェル・スクリプト (shell script). オペレーティング・システムのシェルで解釈されるプログラムまたはスクリプト。

しきい値 (threshold). シミュレーションの中断に適用される設定。あるイベントが指定の比率で発生した場合、既存の条件に基づいて、いつプロセス・シミュレーションを停止すべきかを定義する。

システム分析者 (systems analyst). ビジネス要件からシステム定義およびソリューションを作成する責任を持つ専門家。

実行トレース (execution trace). 統合テスト・クライアントの「イベント」ページで、階層形式で記録および表示される一連のイベント。

シャーシ (chassis). 各種電子部品が取り付けられる金属製のフレーム。

修飾子 (qualifier). 別の一般的な複合エレメントまたは単一エレメントに固有の意味を提供する単一エレメント。修飾子は、単一または複数のオカレンスをマッピングする際に使用される。修飾子を使用すると、名前の 2 番目の部分 (通常は ID と呼ばれる) の解釈で使用する名前空間を指定することもできる。

種別 (classifier). プロセス要素をグループ化およびカラー・コーディングするのに使用される特殊属性。

照会 (query).

1. 特定の条件に基づいてデータベースからの情報を求める要求。例えば、顧客テーブル内で ¥10,000 を上回る残高のすべてのお客様のリストを求める要求。

2. 1 つ以上のモデル・エレメントについての情報を求める再使用可能な要求。

署名者証明書 (signer certificate). 通常はトラストストア・ファイルにあるトラステッド証明書エントリー。

シリアライザー (serializer). オブジェクト・データを別のフォーム (例えば、バイナリー、または XML) に変換するためのメソッド。

シリアライゼーション (serialization). オブジェクト指向プログラミングにおいて、プログラム・メモリーから通信メディアに順次データを書き込むこと。

シン・アプリケーション・クライアント (thin application client). エンタープライズ Bean との対話が可能な、軽量でダウンロード可能な Java アプリケーション・ランタイム。

シン・クライアント (thin client). ソフトウェアがほとんどまたはまったくインストールされていないが、接続先のネットワーク・サーバーで管理および配信されるソフトウェアへのアクセス権限を持つクライアント。シン・クライアントは、ワークステーションなどの全機能を搭載したクライアントの代替である。

スクリプティング (scripting). アプリケーション構築の基礎として既存のコンポーネントを再利用するプログラミング・スタイル。

スクリプト (script). 一連のコマンドをファイルにまとめたもの。ファイルの実行時に特定の機能を実行する。スクリプトは、その実行時に解釈される。

スケーラビリティー. プロセッサー、メモリー、ストレージなどのリソースを追加する際のシステムの拡張能力。

スケルトン (skeleton). 実装クラスのスケルトン。

スコープ (scope).

1. システム・リソースをその範囲内で使用できる境界の指定。

2. Web サービスにおいて、呼び出し要求のサービスを行うオブジェクトの存続期間を識別するプロパティー。

スタック (stack). 一般に一時的なレジスター情報、パラメーター値、サブルーチンの戻りアドレスなどの情報を保管するメモリー内の領域。後入れ先出し (LIFO) の原則に基づいている。

スタンドアロン (stand-alone). ほかのどのデバイス、プログラム、システムからも独立していること。ネットワーク環境において、スタンドアロン・マシンは、必要なすべてのリソースにローカルにアクセスする。

スタンドアロン・サーバー (stand-alone server). サーバー・プロセスの開始および停止を行う、オペレーティング・システムから管理されるカタログ・サービスまたはコンテナー・サーバー

ストリング (string). プログラム言語における、テキストを保管および操作するために使用するデータの形式。

スループット (throughput). 一定期間に渡ってコンピューターやプリンターなどのデバイスで実行される作業量の指標 (1 日当たりのジョブ数など)。

スレッド (thread). プロセスの制御下にあるコンピューター命令のストリーム。 オペレーティング・システムによっては、スレッドとはプロセスでの最小単位の演算命令のこと。複数のスレッドを並行して実行し、それぞれのスレッドで異なるジョブを実行することができる。

スレッド競合 (thread contention). あるスレッドが、別のスレッドが保持しているロックまたはオブジェクトを待機している状態。

静的 (static). Java プログラミング言語のキーワードの一つであり、変数をクラス変数として定義するために使用される。

セキュリティー・トークン (security token). クライアントによって生成された資格証明のセットを表し、名前、パスワード、ID、キー、証明書、グループ、特権などを含めることができる。

セキュリティー管理者 (security administrator). ビジネス・データおよびプログラム機能へのアクセスを制御する担当者。

セッション.

1. ネットワーク上の 2 つのステーション、ソフトウェア・プログラム、またはデバイス間の論理接続または仮想接続。これにより、2 つのエレメントがデータ通信およびデータ交換を行うことができる。

2. 同じブラウザーで同じユーザーから発信される、サーブレットへの一連の要求。

3. Java EE において、複数の HTTP 要求にわたる Web アプリケーションとユーザーとの対話を追跡するためにサーブレットが使用するオブジェクト。

セッション・アフィニティー (session affinity). クライアントが常に同じサーバーに接続するようなアプリケーションの構成方法。この構成では、最初に接続した後、クライアント要求が常に同じサーバーに送られるので、ワークロード管理を行うことはできない。

セル (cell).

1. 同じデプロイメント・マネージャーにフェデレートされて、高可用性を持つコア・グループを含めることができる、管理対象プロセスのグループ。

2. ランタイム・コンポーネントをホストする 1 つ以上のプロセス。それぞれが名前付きのコア・グループを 1 つ以上持つ。

セル・スコープ・バインディング (cell-scoped binding). バインディングがノードまたはサーバーに固有でなく、関連がない場合のバインディング・スコープ。このタイプの名前バインディングは、セルの永続的なルート・コンテキストに従って作成される。

ゾーン・ベース・サポート (zone-based support). ルール・ベースの断片配置を有効にして、階、建物、地域などが異なるさまざまなデータ・センターにまたがって断片を配置することで、グリッドの可用性を高める機能。

操作 (operation). あるオブジェクトが呼び出されて実行する、機能や照会の実装。

粗視化 (coarse-grained). オブジェクト群を論理的にハイレベル、要約レベルから観察する手法。

組織 (organization). 規定の目標を達成するために人々が協力し合うエンティティーのこと。例えば、企業、会社、工場など。

存続時間 (time to live). キャッシュに存在する項目が破棄されるまでの時間を秒単位で表したもの。

ダーティー読み取り (dirty read). いかなるロック・メカニズムも伴わない読み取り要求。つまり、データを読み取ることはできるが、その後ロールバックされた結果として、読み取られたものとデータベースに入っているものとが一致しなくなることがある。

タイマー (timer). 特定の時点で出力を生成するタスク。

タイミング制約 (timing constraint). 1 つのメソッド呼び出しまたは一連のメソッド呼び出しの期間を測定するために使用される特殊な検証アクション。

ダウンストリーム (downstream). フローの方向に関して、プロセスの最初のノード (アップストリーム) からプロセスの最後のノード (ダウンストリーム) に向かう方向のこと。

ダッシュボード (dashboard). ビジネス・データをグラフィカルに示す 1 つ以上のビューアーを含むことが可能な Web ページ。

断片. 区画のインスタンス。断片は基本またはレプリカとすることができる。

データ・グリッド (data grid). テラバイトまたはペタバイトのデータにアクセスするためのシステム。

デーモン (daemon). ネットワーク制御など、連続的または周期的な機能をバックグラウンドで実行するプログラム。

デジタル証明書 (digital certificate). 個人、システム、サーバー、会社、またはその他のエンティティーを識別するために使用され、公開鍵をそのエンティティーに関連付けるために使用される電子文書。デジタル証明書は、認証局によって発行され、その認証局によってデジタル署名される。

デシリアライゼーション (deserialization). シリアライズされた変数をオブジェクト・データに変換するメソッド。

デッドロック (deadlock). 2 つの独立した制御スレッドがブロックされ、一方が何らかのアクションを実行するため他方を待っている状態。競合状態を避けるため、同期メカニズムの追加からデッドロックが生じることがよくある。

デプロイ. 作動環境にファイルを置いたりソフトウェアをインストールしたりすること。Java Platform, Enterprise Edition (Java EE) では、デプロイされるアプリケーションのタイプに適したデプロイメント記述子の作成を伴う。

デプロイ・フェーズ (deploy phase). 「デプロイメント・フェーズ (deployment phase)」も参照。

デプロイメント・コード (deployment code). アプリケーション開発者によって記述された Bean 実装コードが特定の EJB ランタイム環境で動作できるようにする追加コード。デプロイメント・コードは、アプリケーション・サーバー・ベンダーが提供するツールで生成できる。

デプロイメント・ディレクトリー (deployment directory). アプリケーション・サーバーがインストールされたマシン上で公開サーバー構成と Web アプリケーションが配置されるディレクトリー。

デプロイメント・トポロジー (deployment topology). デプロイメント環境でのサーバーおよびクラスターの構成と、それらの間の物理関係および論理関係。

デプロイメント・フェーズ (deployment phase). アプリケーションのホスティング環境の作成とそれらのアプリケーションのデプロイメントの組み合わせを含むフェーズ。これにはアプリケーションのリソース依存、操作条件、キャパシティー要件、および保全性とアクセス権限の制約の解決を含む。

デプロイメント・ポリシー (deployment policy). システム数、サーバー数、区画数、レプリカ数 (レプリカ・タイプを含む)、各サーバーのヒープ・サイズなど、さまざまな項目に基づいて eXtreme Scale 環境を構成するためのオプションの手段。

デプロイメント・マネージャー. 論理グループまたは他のサーバーのセルの操作を管理するサーバー。

デプロイメント環境 (deployment environment). 構成済みのクラスター、サーバー、およびミドルウェアの組み合わせによって、ソフトウェア・モジュールをホストするための環境を提供する。例えば、デプロイメント環境はメッセージの宛先のホスト、ビジネス・イベントのプロセッサーまたはソーター、および管理プログラムを含むことがある。

デプロイメント記述子 (deployment descriptor). 構成オプションおよびコンテナー・オプションを指定することにより、モジュールまたはアプリケーションをデプロイする方法を記述している XML ファイル。例えば、 EJB デプロイメント記述子は、エンタープライズ Bean を管理、制御する方法に関する情報を EJB コンテナーに渡す。

伝送制御プロトコル/インターネット・プロトコル (Transmission Control Protocol/Internet Protocol (TCP/IP)). 業界標準の独占されていない通信プロトコルのセットのことで、異なる種類の相互接続ネットワークにおいて、アプリケーション間の信頼性のあるエンドツーエンド接続を提供する。

トークン (token).

1. シミュレーションの実行中にプロセス・インスタンスの現在の状態を追跡するために使用するマーカー。
2. ネットワーク上で転送を行うときの許可または一時的な制御を示す特定のメッセージまたはビット・パターン。

同期化 (synchronize). ある機能または成果物を別のものと一致するように加算、減算、または変更すること。

同期複製 (synchronous replica). データの整合性を保証するため、プライマリー断片においてトランザクションの一部として更新を受信する断片。この場合、非同期複製に比べて応答時間が増す可能性がある。

同期プロセス (synchronous process). 要求/応答オペレーションを起動することによって開始されるプロセス。プロセスの結果は、同じオペレーションによって戻される。

統合開発環境 (IDE) (integrated development environment (IDE)). ソース・エディター、コンパイラー、デバッガーなど、一連のソフトウェア開発ツールのこと。単一ユーザー・インターフェースからアクセス可能。

動的キャッシュ (dynamic cache). あるサービスの中のサーブレット、Web サービス、WebSphere コマンドを含むいくつかのキャッシング・アクティビティーの集まりで、構成情報を共有しパフォーマンスが向上するように機能する。

動的クラスター. クラスター・メンバーから収集されたパフォーマンス情報に基づき、重みを使用して、クラスター・メンバーのワークロードを動的にバランスさせるサーバー・クラスター。

トポロジー (topology). ネットワーク内のネットワーキング・コンポーネントまたはノードの場所に関する物理的または論理的なマッピング。一般的なネットワーク・トポロジーとしては、バス、リング、スター、ツリーなどがある。

ドメイン (domain). あるドメイン内のリソースを表現する別のオブジェクトが入っているオブジェクト、アイコン、およびコンテナー。ドメイン・オブジェクトを使用すると、これらのリソースを管理できる。

ドメイン・ネーム・システム (DNS). ドメイン・ネームを IP アドレスにマップする分散データベース・システム。

トラストストア・ファイル (truststore file). トラステッド・エンティティーの公開鍵が入っている鍵データベース・ファイル。

ドロップダウン (drop-down). 「プルダウン (pull-down)」を参照。

名前空間 (namespace). すべての名前が固有である論理コンテナー。成果物の固有 ID は、名前空間と、成果物のローカル名で構成される。

認証 (authentication). コンピューター・システムのユーザーが本人であることを証明するセキュリティー・サービス。このサービスを実装する一般的な手段として、パスワードやデジタル署名などがある。認証は許可とは異なり、システム・リソースへのアクセスの許可または拒否とは関係がない。

認証済みユーザー (authenticated user). 有効なアカウント (ユーザー ID およびパスワード) でポータルにログインしたポータル・ユーザー。認証済みユーザーはすべてのパブリック・プレースへのアクセス権限を持つ。

認証別名 (authentication alias). リソース・アダプターおよびデータ・ソースへのアクセスを許可する別名。認証別名にはユーザー ID およびパスワードなどの認証データが含まれる。

ノード・エージェント (node agent). ノード上のすべてのアプリケーション・サーバーを管理し、管理セル内のノードを表す管理エージェント。

パーシスタンス (persistence).

1. セッション境界を超えて保持されるデータ、または作成元のプログラムまたはプロセスの実行後も引き続き存在するオブジェクトの特性。通常は、データベース・システムなどの不揮発性ストレージに存在する。

2. Java EE において、エンティティー Bean の状態をそのインスタンス変数と基本データベース間で転送するためのプロトコル。

パーシスト (persist). 通常、データベース・システムやディレクトリーなどの不揮発性ストレージ内で、セッション境界を越えて保持されること。

ハートビート (heartbeat). エンティティーがまだアクティブであることを通知するために別のエンティティーに送信するシグナル。

パーミッション (permission). ローカル・ファイルの読み取りと書き込み、ネットワーク接続の作成、ネイティブ・コードのロードなどのアクティビティーを実行する権限。

排他ロック. 同時に実行するアプリケーション・プロセスがデータベースのデータにアクセスできないようにするロック。「共用ロック (shared lock)」も参照。

バイトコード (bytecode). Java コンパイラーによって生成され、Java インタープリターによって実行される、マシンから独立したコード。

バイナリー形式 (binary format). 各フィールドの長さが 2 バイトまたは 4 バイトであるような 10 進値表現。フィールドの左端のビットは符号 (+ または -) であり、フィールドの残りのビットは数値である。正数の符号ビットは 0 である。正数は true 形式で表現される。負数の符号ビットは 1 である。負数は 2 の補数形式で表現される。

派生 (derivation). オブジェクト指向プログラミングで、1 つのクラスから別のクラスへの改良または拡張。

パッケージ (package).

1. Java プログラミングにおけるタイプのグループ。パッケージは、パッケージ・キーワードによって宣言される。

2. 文書の内容を囲むラッパーで、インターネット経由で文書を送信するのに使用するフォーマットを定義する。RNIF、AS1、および AS2 など。

3. コンポーネントを組み立ててモジュールにし、モジュールを組み立ててエンタープライズ・アプリケーションにすること。

発生 (fire). オブジェクト指向プログラミングにおいて、状態遷移を起こすこと。

反復 (iteration). 「ループ (loop)」を参照。

汎用オブジェクト (generic object). 概念、カスタム・エンティティー、またはコレクションを参照するために API 呼び出しおよび XPath 式で使用するオブジェクト。例えば、XPath 式 /WSRR/GenericObject は、WebSphere Service Registry and Repository からすべての概念を取得する。

微細化 (fine-grained). オブジェクトを個別に詳細に見ていくこと。

非推奨 (deprecated). サポートされているが推奨されなくなり、廃止される可能性のあるエンティティー (プログラミング・エレメントまたはフィーチャーなど) について使用される言葉。

非同期 (asynchronous). 時間内に同期しないイベント、あるいは定期的または予測可能な時間間隔で発生しないイベントに関する用語。

非同期複製 (asynchronous replica). トランザクションのコミット後に更新を受信する断片。この方式は、同期複製に比べて高速であるが、プライマリー断片の背後のいくつかのトランザクションが非同期複製となる場合があるため、データ損失が発生する可能性がある。

非同期メッセージング (asynchronous messaging). プログラムがメッセージ・キューにメッセージを入れたら、メッセージへの応答を待たずに次の処理に進める、プログラム間での通信方式。

非武装地帯 (demilitarized zone (DMZ)). インターネットに見られるような、企業のイントラネットと公衆ネットワークとの間に保護層として追加された、複数のファイアウォールを含む構成。

ビルド・パス (build path). Java ソース・コードのコンパイル中に、別のプロジェクトにある参照クラスを検出するために使用されるパス。

ビルド計画 (build plan). 成果物をビルドするために必要な処理を定義し、処理が行われるマシンを指定するための XML ファイル。

ビルド時のデータ (build time data). EDI 標準、レコード指向データ文書タイプ、およびマップなど、変換プログラムで使用されないオブジェクト。

ビルド定義ファイル (build definition file). カスタマイズ・インストール・パッケージ (CIP) のコンポーネントと特性を特定する XML ファイル。

ブートストラッピング (bootstrapping). ネーミング・サービスの初期参照を取得するプロセス。ブートストラップ設定およびホスト名が、Java Naming and Directory Interface (JNDI) 参照の初期コンテキストを形成する。

ブートストラップ (bootstrap). システムを初期化する一連の処理。

ファイアウォール (firewall). セキュア・ネットワークに入ったり出たりする承認されないトラフィックを阻止するために使われるネットワーク構成のこと。通常はハードウェアおよびソフトウェアの両方が使われる。

ファクトリー (factory). オブジェクト指向プログラミングにおいて、別のクラスのインスタンスを作成するために使用するクラスのこと。ファクトリーを使用すると、新たな機能追加をしたい特定クラスのオブジェクト作成をそこだけで行うことができ、あちこちコード変更をしないで済む。

フィックスパック (fix pack). 出荷スケジュールが決められたリフレッシュ・パック、製造リフレッシュ、リリースの間に提供される、累積フィックスがまとめられたもの。お客様が特定の保守レベルに移行できることを意図したもの。「インテリム・フィックス (interim fix)」も参照。

フェイルオーバー (failover). ソフトウェア、ハードウェア、またはネットワークの障害が発生した場合に、冗長システムまたは待機システムに自動的に切り替わること。

フォーク (fork). 同時に並列処理される処理パスに対して、入力のコピーをそれらに渡すためのプロセス要素のこと。

フォルダー (folder). オブジェクトをまとめるために使用するコンテナー。

副照会 (subquery). SQL の述部で使用される副選択。他の SQL ステートメントの WHERE 節または HAVING 節内の select ステートメントなど。

複製 (replication). 複数のロケーションで定義済みのデータ・セットを保守するプロセス。複製には、あるロケーション (ソース) の指定された変更を、別のロケーション (ターゲット) にコピーして、両方のロケーションのデータを同期化することが含まれる。

プラグイン. 既存のプログラム、アプリケーション、またはインターフェースに機能を追加する、個別にインストール可能なソフトウェア・モジュール。

プリミティブ型 (primitive type). Java におけるデータ型のカテゴリー。その型に対する適切なサイズおよび形式 (数値、文字、またはブール値) の単一の値を含む変数を記述する。プリミティブ型の種類の例としては、byte、short、int、long、float、double、char、boolean がある。

ブレークポイント (breakpoint). プロセスまたはプログラマチック・フローでのマークを付けられたポイントで、ポイントに到達するとフローが一時停止し、通常はデバッグまたはモニターが可能になる。

プロキシー (proxy). 特定のネットワーク・アプリケーション用 (Telnet や FTP など) に、あるネットワークから別のネットワークへ転送するアプリケーション・ゲートウェイ。例えば、ファイアウォール・プロキシー Telnet サーバーがユーザーの認証を実行すると、トラフィックは、プロキシーが存在しないかのようにそのプロキシーを流れる。機能はクライアント・ワークステーションではなくファイアウォールで実行されるため、ファイアウォールの負荷が増す。

プロキシー・クラスター (proxy cluster). HTTP 要求をクラスター全体にわたって配布するプロキシー・サーバーのグループ。

プロキシー・サーバー (proxy server).

1. アプリケーションまたは Web サーバーによってホストされる、HTTP Web 要求の仲介として動作するサーバー。プロキシー・サーバーは、エンタープライズ内のコンテンツ・サーバーの代理の役割を果たす。

2. 別のサーバーを対象とした要求を受信し、要求されたサービスを獲得するために、クライアントに代わって (クライアントのプロキシーとして) 働くサーバー。プロキシー・サーバーは、クライアントとサーバーが、直接接続するには非互換であるという場合によく使用される。例えば、クライアントはサーバーのセキュリティー認証要件に合わせることができないが、一部のサービスの許可が必要な場合がこれに該当する。

プロキシー・ピア・アクセス・ポイント (proxy peer access point). 直接にはアクセスできないピア・アクセス・ポイントの通信設定を識別する手段。

プログラム一時修正 (program temporary fix (PTF)). System i、System p、および System z 製品の場合、すべてのお客様が入手できるようにしてある IBM テスト済みの修正。「フィックスパック (fix pack)」も参照。

プログラム診断依頼書 (authorized program analysis report (APAR)). サポート対象リリースの IBM 提供プログラムにおける問題点に対する修正要求。

プロセス (process).

1. 特定の結果または結末に体系的に導かれる一連の制御されたアクティビティーで構成された、連続的に続く手順。

2. ビジネス・トランザクションを実行するためにコミュニティー・マネージャーと参加プログラムの間で交換される文書またはメッセージのシーケンス。

プロトコル・バインディング (protocol binding). エンタープライズ・サービス・バスが通信プロトコルとは無関係にメッセージを処理できるようにするバインディング。

プロパティー (property). オブジェクトを記述するオブジェクトの特性。プロパティーは変更できる。プロパティーは、オブジェクト名前、タイプ、値、振る舞いなどの事項を記述できる。

プロファイル (profile). ユーザー、グループ、リソース、プログラム、デバイス、またはリモート・ロケーションの特性を記述しているデータ。

プロンプト (prompt). フィールドにユーザー入力し、出力画面へ遷移することを確認できるコンポーネント。

分散 eXtreme Scale (distributed eXtreme Scale). サーバーおよびクライアントが複数のプロセスに存在する場合に、eXtreme Scale と対話するための使用パターン。

文書タイプ定義 (DTD) (document type definition (DTD)). SGML または XML 文書の個々のクラスの構造を指定する規則。DTD は、エレメント、属性、および表記法を使用して構造を定義する。また、各エレメント、属性、および表記法を、文書の個々のクラス内で使用する方法に関する制約も規定する。

ペシミスティック・ロック (pessimistic locking). 行が選択されてから、その行に対して検索更新操作または検索削除操作が試みられるまでの間、ロックが保たれるようなロック戦略。

ヘルス (health). データベース環境の全般的条件または状態。

変数 (variable). 可変値を表す。

ポート (port). Web サービス記述言語 (WSDL) の資料に定義されているように、バインディングとネットワーク・アドレスの組み合わせとして定義される単一エンドポイント。

ポート番号 (port number). インターネット通信において、アプリケーション・エンティティーとトランスポート・サービスの間の論理結合子の ID。

保守モード (maintenance mode). 管理者が実稼働環境の着信トラフィックを中断することなく、ノードまたはサーバーの診断、保守、またはチューニングに使用できる、ノードまたはサーバーの状態。

ホスト (host).

1. ネットワークに接続され、そのネットワークへのアクセス・ポイントを提供するコンピューター。ホストはクライアント、サーバー、または同時にその両方である場合がある。

2. パフォーマンス・プロファイル作成では、プロファイル作成されているプロセスを所有しているマシン。サーバー (server) も参照。

ホスト・システム (host system). 3270 アプリケーションをホストするエンタープライズ・メインフレーム・コンピューター・システム。3270 端末サービス開発ツールでは、開発者は 3270 端末サービス・レコーダーを使用してホスト・システムに接続する。

ホスト名 (host name).

1. インターネット通信では、コンピューターに付けられた名前。ホスト名は、完全修飾ドメイン名 (例: mycomputer.city.company.com) の場合も、あるいは、固有のサブネーム (例: mycomputer) の場合もあります。

2. ノードがインストールされている物理マシン上のネットワーク・アダプターのネットワーク名。

ボトムアップ開発 (bottom-up development). Web サービスにおいて、Web サービス記述言語 (WSDL) ファイルからではなく、Java Bean またはエンタープライズ Bean などの既存の成果物からサービスを開発するプロセス。

ボトルネック. リソースの競合がパフォーマンスに影響を与えるシステムの場所。

ポリシー. 管理対象リソースまたはユーザーの振る舞いに影響を与える一連の考慮事項。

マップ (map).

1. キーを値にマップするデータ構造。

2. ソースとターゲットの間の変換を定義するファイル。

3. EJB 開発環境で、エンタープライズ Bean のコンテナー管理の永続フィールドが、リレーショナル・データベースの表または他の永続ストレージにある列に対応する方法の指定。

メソッド (method). オブジェクト指向プログラミングにおいて、オブジェクトが実行できるオペレーション。オブジェクトには多数のメソッドがある。

メトリック (metric). モニター関連の用語で、情報 (通常は業績測定) のホルダー。

メモリー・リーク (memory leak). 既に不要なために新たに再生すべきオブジェクト参照をプログラムが保持し続けることによる悪影響のある現象。

呼び出し (invocation). プログラムまたはプロシージャーを活動化すること。

ライトスルー・キャッシュ (write-through cache). ローダーを使用して、データベースに対する各書き込み操作が同期的に行われるキャッシュ。

ライフサイクル. ソフトウェア開発における方向付け、推敲、作成、および移行の 4 つのフェーズを一巡すること。

ライブラリー (library).

1. ビジネス・アイテム、プロセス、タスク、リソース、組織などのモデル・エレメントの集合。

2. 開発、バージョン管理、および共用リソースの編成のために使用されるプロジェクト。ビジネス・オブジェクトやインターフェースなど、成果物タイプのサブセットのみをライブラリーに作成および保管することができる。

ランタイム (run time). コンピューター・プログラムが実行している間の時間枠。

ランタイム・トポロジー (runtime topology). 環境の現行の状態を表したもの。

リードスルー・キャッシュ (read-through cache). 要求されたデータ・エントリーをキーによってロードするスパース・キャッシュ。データがキャッシュに見つからない場合、その欠落データがローダーによって検索され、このローダーがそのデータをバックエンド・データ・リポジトリーからロードしてキャッシュに挿入する。

リスナー (listener). 接続要求を受け付け、関連チャネルを開始するプログラム。

リスナー・ポート (listener port). 接続ファクトリー、宛先、およびデプロイされたメッセージ駆動型 Bean 間の関連を定義するオブジェクト。リスナー・ポートは、これらのリソース間の関連の管理を単純化する。

リソース (resource).

1. 離散的アセット。例えば、アプリケーション・スイート、アプリケーション、ビジネス・サービス、インターフェース、エンドポイント、ビジネス・イベントなど。

2. ジョブ、タスク、または実行中のプログラムが必要とするコンピューター・システムまたはオペレーティング・システムの機能。リソースには、メイン・ストレージ、入出力装置、処理装置、データ・セット、ファイル、ライブラリー、フォルダー、アプリケーション・サーバー、制御プログラム、処理プログラムなどがある。

3. タスクまたはプロジェクトを実行するための個人、装置、または資料。各リソースは、リソース定義の特定の存在または例である。

リフレッシュ・パック (refresh pack). 修正の累積コレクションで、新規機能を含む。 フィックスパック (fix pack)、暫定修正 (interim fix) も参照。

領域 (region). 共通特性を備え、プロセス間で共用可能な仮想ストレージの連続区域。

ループ (loop). 反復して実行される命令のシーケンス。

例外 (exception). 通常の処理では扱うことのできない条件またはイベント。

例外ハンドラー (exception handler). 異常条件に対応するルーチンのセット。例外ハンドラーは割り込みを行い、通常の処理の実行を再開できる。

レプリカ. ディレクトリーのコピー、または別のサーバーのディレクトリーのコピーが含まれるサーバー。レプリカにより、パフォーマンスや応答時間の改善のため、またはデータ保全性の維持のために、サーバーがバックアップされます。

ローカル.

1. ユーザー・システムから、通信回線を使用せずに直接アクセスする装置、ファイル、またはシステムを示す用語。

2. 特定のプロセス内でのみ使用可能なエレメントに関する用語。

ローカル・データベース (local database). 使用しているワークステーションに配置されているデータベース。

ローダー. 永続ストアでデータの読み書きを行うコンポーネント。

ロード・バランシング (load balancing). アプリケーション・サーバーの監視と、サーバー上のワークロード管理。あるサーバーのワークロードが超過すると、要求は、容量のより大きい別のサーバーへ転送される。

ロール (role).

1. 個人またはバルク・リソースによって実行される機能の記述、および機能を遂行するために必要な資格。シミュレーションおよび分析において、ロールという用語は、資格のあるリソースを指すためにも使用される。

2. ユーザーが実行できるタスク、およびユーザーがアクセスできるリソースを識別するジョブの機能。1 人のユーザーに 1 つ以上のロールを割り当てることができる。

3. 一連の許可を提供するプリンシパルの論理グループ。オペレーションへのアクセスは、ロールへのアクセスを認可することにより制御される。

4. リレーションにおいて、ロールは、エンティティーの機能および関与を決定する。ロールは、関与するエンティティーと関与の方法に対する構造および制約の要件を取り込む。例えば、雇用関係におけるロールは、雇用者と被雇用者である。

ロギング (logging). エラーなど、システム上の特定のイベントについてのデータを記録すること。

ロック (lock). 1 つのアプリケーション・プロセスによって行われたコミットされていない変更が別のアプリケーション・プロセスに感知されないようにし、1 つのアプリケーション・プロセスが別のアプリケーション・プロセスでアクセス中のデータを更新できないようにする手段。ロックにより、各ユーザーが同時に不整合データにアクセスできなくなるので、データの保全性が保たれる。

ロング・ネーム (long name). z/OS プラットフォーム上のサーバーに論理名を指定するプロパティー。

ワークスペース (workspace).

1. すべてのプロジェクト・ファイルとプリファレンスなどの情報を含むディスク上のディレクトリー。

2. 管理クライアントが使用する構成情報の一時的なリポジトリー。

3. Eclipse では、現在ユーザーがワークベンチで開発を行っているプロジェクトおよびその他のリソースの集合。これらのリソースに関するメタデータは、ファイル・システム上のディレクトリーにある。リソースが同じディレクトリーにある場合もある。

ワークロード・マネージャー (WLM) (Workload Manager (WLM)). z/OS のコンポーネントの 1 つ。単一の z/OS イメージまたは複数のイメージで同時に複数のワークロードを実行するための機能を提供する。

ワークロード管理 (workload management). アプリケーション・サーバーやエンタープライズ Bean、サーブレットなど、要求を効率的に処理できるオブジェクトに対して、着信した作業要求を最適な方法で分配すること。

1 次キー (primary key).

1. 特定のタイプのエンティティー Bean を一意的に識別するオブジェクト。

2. リレーショナル・データベースで、データベース・テーブルのある 1 つの行を一意的に識別するキー。

APAR. 「プログラム診断依頼書 (authorized program analysis report)」を参照。

API. 「アプリケーション・プログラミング・インターフェース (application programming interface)」を参照。

Bean. JavaBeans コンポーネントの定義およびインスタンス。「JavaBeans」、「エンタープライズ Bean (Enterprise Bean)」も参照。

Bean Scripting Framework. スクリプト言語機能を Java アプリケーションに取り込むアーキテクチャー。

Bean 管理トランザクション (BMT) (bean-managed transaction (BMT)). トランザクションをコンテナー経由ではなく直接管理するための、セッション Bean、サーブレット、またはアプリケーション・クライアント・コンポーネントの機能。

Bean 管理パーシスタンス (BMP) (bean-managed persistence (BMP)). エンティティー Bean の変数とリソース・マネージャーの間で行われるデータ転送がエンティティー Bean によって管理されるときに使用されるメカニズム。

Bean 管理メッセージング (bean-managed messaging). メッセージング・インフラストラクチャー全体の完全な制御をエンタープライズ Bean に与える非同期メッセージングの機能。

Bean クラス (bean class). Enterprise JavaBeans (EJB) プログラミングにおいて、javax.ejb.EntityBean クラスまたは javax.ejb.SessionBean クラスを実装する Java クラス。

BMP. 「Bean 管理パーシスタンス (bean-managed persistence)」を参照。

BMT. 「Bean 管理トランザクション (bean-managed transaction)」を参照。

CIP. 「カスタマイズ・インストール・パッケージ (customized installation package)」を参照。

cloudscape. 組み込み可能で、すべてが Java で書かれたオブジェクト・リレーショナル・データベース管理システム (ORDBMS)。

create メソッド (create method). エンタープライズ Bean では、エンタープライズ Bean を作成するために、ホーム・インターフェース内に定義され、クライアントによって呼び出されるメソッド。

DB2. リレーショナル・データベース管理用 IBM ライセンス・プログラム・ファミリー。

DNS. 「ドメイン・ネーム・システム (Domain Name System)」を参照。

do while ループ (do-while loop). ある条件が満足される限りアクティビティーの同じシーケンスを反復するループ。do while ループは while ループと異なり、ループの終わりで条件をテストする。つまり、アクティビティーのシーケンスは最低 1 回は必ず実行されることを意味する。

DTD. 「文書タイプ定義 (document type definition)」を参照。

DTD 文書定義 (DTD document definition). XML DTD に基づく XML 文書の記述またはレイアウト。

EAR. 「エンタープライズ・アーカイブ (enterprise archive)」を参照。

EAR プロジェクト (EAR project). 「エンタープライズ・アプリケーション・プロジェクト (enterprise application project)」を参照。

Eclipse. ISV やその他のツール・デベロッパーに対して、互換性のある開発ツールを作成するための標準プラットフォームを提供する、オープン・ソース・イニシアチブ。

EJB. 「Enterprise JavaBeans」を参照。

EJB JAR ファイル (EJB JAR file). EJB モジュールを含む Java アーカイブ。

EJB オブジェクト (EJB object). エンタープライズ Bean において、エンタープライズ Bean リモート・インターフェースを実装するクラスを持つオブジェクト。

EJB 継承 (EJB inheritance). エンタープライズ Bean が、同じグループ内の他のエンタープライズ Bean からプロパティー、メソッド、およびメソッド・レベルの制御記述子属性を継承する際の形式。

EJB コンテキスト (EJB context). エンタープライズ Bean において、コンテナーによって提供されるサービスを呼び出すこと、およびクライアントに呼び出されたメソッドの呼び出し側についての情報を取得することをエンタープライズ Bean に許可するオブジェクト。

EJB コンテナー (EJB container). Java EE アーキテクチャーの EJB コンポーネント規約を実装するコンテナー。この規約は、エンタープライズ Bean に対してランタイム環境を規定する。これには、セキュリティー、並行性、ライフサイクル管理、トランザクション、デプロイメント、およびその他のサービスが含まれる。

EJB サーバー (EJB server). EJB コンテナーにサービスを提供するソフトウェア。 EJB サーバーは、1 つ以上の EJB コンテナーをホスティングできる。

EJB 参照 (EJB reference). ターゲットの動作環境で、エンタープライズ Bean のホーム・インターフェースの位置を指定するためにアプリケーションが使用する論理名。

EJB 照会 (EJB query). EJB 照会言語において、戻される EJB オブジェクトを特定するオプションの SELECT 節、Bean コレクションを指定する FROM 節、コレクションでの検索述部を含むオプションの WHERE 節、結果のコレクションの順序を指定するオプションの ORDER BY 節、および finder メソッドの引数に対応する入力パラメーターを含むストリング。

EJB ファクトリー (EJB factory). エンタープライズ Bean インスタンスの作成と検索を単純化するアクセス Bean。

EJB プロジェクト (EJB project). エンタープライズ Bean、ホーム・インターフェース、ローカル・インターフェース、リモート・インターフェース、JSP ファイル、サーブレット、およびデプロイメント記述子など、EJB アプリケーションに必要なリソースを含むプロジェクト。

EJB ホーム・オブジェクト (EJB home object). Enterprise JavaBeans (EJB) プログラミングにおいて、エンタープライズ Bean に対してライフサイクル操作 (create、remove、find) を実行するオブジェクトのこと。

EJB モジュール. 1 つ以上のエンタープライズ Bean および EJB デプロイメント記述子からなるソフトウェア単位。

Enterprise JavaBeans (EJB). オブジェクト指向の分散型エンタープライズ・レベル・アプリケーション (Java EE) の開発とデプロイメントのため、Sun Microsystems によって定義されたコンポーネント・アーキテクチャー。

ESB. 「エンタープライズ・サービス・バス (enterprise service bus)」を参照。

Evictor. 各 BackingMap インスタンス内にあるエントリーのメンバーシップを制御するコンポーネント。スパース・キャッシュでは、エビクターを使用して、データベースに影響を及ぼすことなくキャッシュからデータを自動的に除去できる。

expression. 1 つの SQL または XQuery オペランド、あるいは SQL または XQuery 演算子とオペランドからなる 1 つのコレクション。単一の値を生成する。

Extensible Markup Language (XML). Standard Generalized Markup Language (SGML) に基づくマークアップ言語を定義する標準メタ言語。

eXtreme Scale グリッド (eXtreme Scale grid). データおよびクライアントがすべて 1 つのプロセスにある場合に、eXtreme Scale と対話するために使用されるパターン。

for ループ (for loop). 同一の複数アクティビティーの順序処理を、指定した回数だけ繰り返すループのこと。

General Inter-ORB Protocol (GIOP). Common Object Request Broker Architecture (CORBA) がメッセージのフォーマットを定義するために使用するプロトコル。

getter メソッド (getter method). インスタンスの値、またはクラス変数を取得することを目的としたメソッド。これにより、別のオブジェクトがその変数のうちの 1 つの値を取得できる。

GIOP. 「General Inter-ORB Protocol」を参照。

HA. 「高可用性 (high availability)」を参照。

HA グループ. プロセスの高可用性を実現するために使用されるメンバーの集まり。

HA ポリシー. HA グループのために定義するルールのセットで、0 またはそれ以上のメンバーをアクティブにするかどうかを決定する。このポリシーは、ポリシー一致基準をグループ名と突き合わせることによって、特定の HA グループと関連付けられる。

HA マネージャー (high availability manager). コア・グループ・メンバーシップが判別され、状況がコア・グループ・メンバー間で伝達されるフレームワーク。

HTTP over SSL (HTTPS). トランザクションを保護するための Web プロトコル。ユーザー・ページ要求および Web サーバーで戻されるページを暗号化、および復号化を行う。

HTTPS.

1. 「SSL を使用する HTTP (HTTP over SSL)」を参照。
2. 「Hypertext Transfer Protocol Secure」を参照。

Hypertext Transfer Protocol Secure (HTTPS). ハイパーメディア文書をインターネット経由で安全に転送および表示するために、Web サーバーと Web ブラウザーで使用されるインターネット・プロトコル。

IDE. 「統合開発環境 (integrated development environment)」を参照。

if-then ルール (if-then rule). 条件 (if 部分) が真のときにのみ、アクション (then 部分) が実行されるルール。

IIOP. 「Internet Inter-ORB Protocol」を参照。

Internet Inter-ORB Protocol (IIOP). Common Object Request Broker Architecture (CORBA) オブジェクト・リクエスト・ブローカー間の通信に使用されるプロトコル。

IP. 「インターネット・プロトコル (Internet Protocol)」を参照。

IP スプレイヤー (IP sprayer). 複数ユーザーからのインバウンド要求と複数アプリケーション・サーバー・ノードの間に位置し、リクエストを複数ノードに転送する装置。

JAAS . 「Java 認証・承認サービス (Java Authentication and Authorization Service)」を参照。

JAF. 「JavaBeans Activation Framework」を参照。

JAR ファイル (JAR file). Java アーカイブ・ファイル。「Web アーカイブ (Web archive)」、「エンタープライズ・アーカイブ (enterprise archive)」も参照。

Java. リモート・オブジェクト内の相互作用をサポートする、移植可能な解釈コード用のオブジェクト指向プログラム言語。Java は、Sun Microsystems によって開発および指定されたものである。

Java API for XML (JAX). Extensible Markup Language (XML) を介して定義されたデータに関連するさまざまな操作を処理するための Java ベースの API のセット。

Java Authentication and Authorization Service (JAAS). Java EE テクノロジーにおいて、セキュリティー・ベースのオペレーションを実行するための標準 API。サービスは、JAAS を介して、ユーザーを認証および承認し、基礎となるテクノロジーからアプリケーションを独立させておくことを可能にする。

Java Command Language. Java 環境用のスクリプト言語で、Web コンテンツの作成および Java アプリケーションの制御に使用される。

Java Database Connectivity (JDBC). Java プラットフォームと広範なデータベースとの間のデータベース独立の接続用の業界標準。JDBC インターフェースは、SQL ベースおよび XQuery ベースのデータベース・アクセス用にコール・レベル・インターフェースを提供する。

Java EE. 「Java Platform, Enterprise Edition」を参照。

Java EE アプリケーション (Java EE application). Java EE 機能のデプロイ可能な任意の単位。この単位には、Java EE アプリケーションのデプロイメント記述子と一緒にエンタープライズ・アーカイブ (EAR) ファイルにパッケージされた、単一モジュールまたはモジュール・グループがある。

Java EE コネクター・アーキテクチャー (JCA). Java EE プラットフォームを異機種混合のエンタープライズ情報システム (EIS) に接続するための標準アーキテクチャー。

Java EE サーバー (Java EE server). EJB コンテナーまたは Web コンテナーを提供するランタイム環境。

Java Management Extensions (JMX). Java テクノロジーを介して Java テクノロジーの管理を行う手段のこと。JMX は、管理用の Java プログラミング言語のユニバーサルかつオープンな拡張機能であり、管理が必要とされるすべての業界でデプロイできる。

Java Message Service (JMS). メッセージ処理用の Java 言語機能を提供する、アプリケーション・プログラミング・インターフェース。

Java Naming and Directory Interface (JNDI). Java プラットフォームの拡張により、異機種の命名サービスとディレクトリー・サービス用の標準インターフェースが提供される。

Java Platform, Enterprise Edition (Java EE). エンタープライズ・アプリケーションを開発およびデプロイするための環境であり、Sun Microsystems によって定義されている。Java EE プラットフォームは、多層化された Web ベース・アプリケーションを開発するための機能を提供する、一連のサービス、アプリケーション・プログラミング・インターフェース (API)、およびプロトコルで構成される。

Java Platform, Standard Edition (Java SE). Java テクノロジー・プラットフォームの中核。

Java SE. 「Java Platform, Standard Edition」を参照。

Java SE Development Kit (JDK). Sun Microsystems が提供する Java プラットフォーム用のソフトウェア開発キットの名前。

Java Secure Socket Extension (JSSE). セキュア・インターネット通信を可能にする Java パッケージ。この Java パッケージは、Java バージョンの Secure Sockets Layer (SSL) および Transport Layer Security (TLS) プロトコルを実装して、データ暗号化、サーバー認証、メッセージ保全性、およびオプションで、クライアント認証をサポートする。

Java Specification Request (JSR). Java プラットフォームに対して、公式に提案された仕様。

Java virtual machine Profiler Interface (JVMPI). ガーベッジ・コレクションに関するデータなどの情報の収集や、アプリケーション・サーバーを実行する Java 仮想マシン (JVM) API をサポートするプロファイル作成ツール。

Java アーカイブ (Java archive). Java プログラムをインストールおよび実行するために必要なすべてのリソースを単一ファイルで保管するための、圧縮されたファイル・フォーマット。「Web アーカイブ (Web archive)」、「エンタープライズ・アーカイブ (enterprise archive)」も参照。

Java 仮想マシン (Java virtual machine (JVM)). コンパイルされた Java コード (アプレットおよびアプリケーション) を実行するプロセッサーのソフトウェア実装。

Java クラス (Java class). Java 言語で記述されるクラス。

Java コネクター・セキュリティー (Java Connector security). Java EE ベースのアプリケーションのエンドツーエンド・セキュリティー・モデルを拡張して、エンタープライズ情報システム (EIS) を組み込むよう設計されたアーキテクチャー。

Java ファイル (Java file). 編集可能なソース・ファイル (拡張子 .java)。バイトコード (.class ファイル) にコンパイルが可能。

Java プラットフォーム (Java platform). プログラム作成のための Java 言語、また、プログラムの開発、コンパイルおよびエラー・チェックに使用する API のセット、クラス・ライブラリー、およびその他プログラム、そしてクラス・ファイルをロードし実行する Java 仮想マシンに対する総称。

Java プロジェクト (Java project). Eclipse では、コンパイル可能な Java ソース・コードを含み、ソース・フォルダーまたはパッケージのコンテナーとなるプロジェクト。

Java ランタイム環境 (Java runtime environment). 標準的 Java プラットフォームを構成する中核の実行可能プログラムおよびファイルを含む Java Developer Kit のサブセット。JRE には Java 仮想マシン (JVM)、コア・クラス、およびそれらをサポートするファイルが含まれる。

JavaBeans. Sun Microsystems によって Java 用に定義された、ポータブルでプラットフォーム非依存の、再使用可能なコンポーネント・モデル。「Bean」も参照。

JavaBeans Activation Framework (JAF). 任意のデータ・タイプおよび使用可能な操作を判別し、関連するサービスを実行するように Bean をインスタンス化できる、Java プラットフォームに対する標準拡張。

Javadoc.

1. 1 組のソース・ファイルの中の宣言およびドキュメンテーション・コメントを解析して、クラス、内部クラス、インターフェース、コンストラクター、メソッド、およびフィールドを記述する 1 組の HTML ページを作成するツール。

2. 1 組のソース・ファイルの中の宣言およびドキュメンテーション・コメントを解析して、クラス、内部クラス、インターフェース、コンストラクター、メソッド、およびフィールドを記述する 1 組の HTML ページを作成するツールに関する用語。

JavaMail API. Java ベースのメール・クライアント・アプリケーションを構築するための、プラットフォームとプロトコルに依存しないフレームワーク。

JavaScript. ブラウザーと Web サーバーの両方で使用される、Web スクリプト言語の 1 つ。

JavaScript Object Notation. JavaScript のオブジェクト・リテラル記法に基づく単純なデータ交換形式。JSON はプログラミング言語に対して中立的だが、C、C++、C#、Java、JavaScript、Perl、Python などの言語の規則を使用する。

JavaServer Pages (JSP). サーバー・サイド・スクリプト・テクノロジーの 1 つで、これにより、Java コードが Web ページ (HTML ファイル) 内に動的に組み込まれ、そのページにサービスが提供されると、実行されて動的コンテンツをクライアントに戻すことが可能になる。

JAX. 「Java API for XML」を参照。

JCA. 「Java EE コネクター・アーキテクチャー (Java EE Connector Architecture)」を参照。

JDBC. 「Java Database Connectivity」を参照。

JDK. 「Java SE Development Kit」を参照。

JMS. 「Java Message Service」を参照。

JMS データ・バインディング (JMS data binding). 外部 JMS メッセージによって使用されるフォーマットと、サービス・コンポーネント・アーキテクチャー (SCA) モジュールによって使用されるサービス・データ・オブジェクト (SDO) の間のマッピングを提供するデータ・バインディング。

JMX. 「Java Management Extensions」を参照。

JNDI. 「Java Naming and Directory Interface」を参照。

JSP. 「JavaServer Pages」を参照。

JSP ファイル (JSP file). .jsp 拡張子を持ち、Web ページへの動的コンテンツの組み込みを可能にする、スクリプト化された HTML ファイル。JSP ファイルは、URL として直接要求するか、サーブレットで呼び出すか、HTML ページ内から呼び出すことができる。

JSP ページ (JSP page). 応答を作成する要求の処理方法を説明する、固定テンプレート・データおよび JSP エレメントを使用したテキスト・ベースの文書。

JSR. 「Java Specification Request」を参照。

JSSE. 「Java Secure Socket Extension」を参照。

JVM. 「Java 仮想マシン (Java virtual machine)」を参照。

JVMPI. 「Java Virtual Machine Profiler Interface」を参照。

Jython. Python プログラミング言語を Java プラットフォームに組み込んで実装したもの。

LDAP. 「Lightweight Directory Access Protocol」を参照。

LDAP ディレクトリー (LDAP directory). 人、組織、およびその他のリソースに関する情報を保管するリポジトリーの一種。LDAP プロトコルを使用してアクセスされる。リポジトリー内の項目は、階層構造に編成される。場合によっては、階層構造は組織の構造または組織の地理的分布を表す。

Lightweight Directory Access Protocol (LDAP). TCP/IP を使用して、X.500 モデルをサポートするディレクトリーを知る方法を提供し、より複雑な X.500 Directory Access Protocol (DAP) のリソース要件を発生させない公開プロトコル。例えば、LDAP を使用して、インターネットまたはイントラネット・ディレクトリー内の個人、組織、およびその他のリソースを見つけることができる。

Lightweight Third Party Authentication (LTPA). 分散環境において、暗号方式を使用してセキュリティーをサポートするプロトコル。

LTPA. 「Lightweight Third Party Authentication」を参照。

MBean. 「Managed Bean」を参照。

MBean プロバイダー (MBean provider). Java Management Extensions (JMX) MBean の実装および MBean の Extensible Markup Language (XML) 記述子ファイルを含むライブラリー。

node.

1. 複数の管理対象サーバーから成る論理グループ。

2. ツリー制御の項目。単一エレメント、複合エレメント、マッピング・コマンド、コメント、またはグループ・ノードが含まれる。

3. XML では、文書における有効で完全な構造の最小単位。

4. 図を構成する基本の形状。

ObjectGrid. Java で書かれたアプリケーション用のグリッドに対応したメモリー・データベース。ObjectGrid は、メモリー内のデータベースとして使用することも、ネットワーク全体にデータを分散するために使用することもできる。

ODBC. 「Open Database Connectivity」を参照。

Open Database Connectivity (ODBC). リレーショナルおよび非リレーショナルの両方のデータベース管理システムのデータにアクセスするための、標準的なアプリケーション・プログラミング・インターフェース (API)。各データベース管理システムが異なるデータ・ストレージ形式およびプログラミング・インターフェースを採用している場合でも、データベース・アプリケーションは、この API を使用することにより、さまざまなコンピューター上のデータベース管理システムに保管されているデータにアクセスできます。

ORB. 「オブジェクト・リクエスト・ブローカー (Object Request Broker)」を参照。

Performance Monitoring Infrastructure (PMI). パフォーマンス・データを収集、配送、処理、および表示するために割り当てられたパッケージおよびライブラリーの集合。

PMI. 「Performance Monitoring Infrastructure」を参照。

point-to-point. メッセージの宛先が送信側のアプリケーションによって認識されている、メッセージング・アプリケーションのスタイル。

PTF. 「プログラム一時修正 (program temporary fix)」を参照。

QoS. 「サービスの品質 (quality of service)」を参照。

root. 最大の権限を持つシステム・ユーザーのユーザー名。

SDK. 「Software Development Kit」を参照。

Secure Sockets Layer (SSL). 通信のプライバシーを提供するセキュリティー・プロトコルの 1 つ。 SSL を使用すれば、盗聴、改ざん、およびメッセージ偽造を防止するよう設計された方法で、クライアント/サーバー・アプリケーションは通信することができる。

setter メソッド (setter method). インスタンスの値、またはクラス変数を設定することを目的としたメソッド。この能力により、他のオブジェクトがその変数の 1 つの値を設定できるようになる。

SLA. 「サービス・レベル・アグリーメント (service level agreement)」を参照。

Software Development Kit (SDK). 特定のコンピューター言語または特定の稼働環境用のソフトウェア開発を支援するツールのセット、API、およびドキュメンテーションのこと。

SQL. 「構造化照会言語 (Structured Query Language)」を参照。

SQL 照会 (SQL query). 結果テーブルを指定する特定の SQL ステートメントのコンポーネント。

SSL. 「Secure Sockets Layer」を参照。

SSL チャネル (SSL channel). トランスポート・チェーン内のチャネルの一種。Secure Sockets Layer (SSL) 構成レパートリーをトランスポート・チェーンに関連付ける。

TCP. 「伝送制御プロトコル (Transmission Control Protocol)」を参照。

TCP チャネル (TCP channel). トランスポート・チェーン内のチャネルの一種。これにより、クライアント・アプリケーションは、ローカル・エリア・ネットワーク (LAN) 内で永続的な接続を行うことができる。

TCP/IP. 「伝送制御プロトコル/インターネット・プロトコル (Transmission Control Protocol/Internet Protocol)」を参照。

TCP/IP モニター・サーバー (TCP/IP monitoring server). TCP/IP アクティビティーだけでなく、Web ブラウザーとアプリケーション・サーバー間のすべての要求と応答をモニターするランタイム環境。

timeout. あるイベントが発生または完了するのを待機する時間間隔。これを過ぎると操作が中断される。

Tivoli Performance Viewer. アプリケーション・サーバーから Performance Monitoring Infrastructure (PMI) データを取得して、それを各種の形式で表示する Java クライアント。WAS Version 6.0 以降は Web アプリケーションとして管理コンソールに統合。

transaction. トランザクション中に行われたデータ変更がすべて一緒に 1 単位としてコミットまたは 1 単位としてロールバックされるプロセス。

Transmission Control Protocol (TCP). インターネット、および Internet Engineering Task Force (IETF) のインターネットワーク・プロトコル標準に準拠するネットワークで使用される通信プロトコル。TCP は、パケット交換通信ネットワークとそのようなネットワークで相互接続されたシステムで、信頼できるホスト間プロトコルを提供する。

type.

1. Java プログラミングにおけるクラス、またはインターフェース。

2. WSDL 文書では、何らかの型システム (XSD など) を使用するデータ型定義を含むエレメントのこと。

UDDI. 「Universal Description, Discovery, and Integration」を参照。

Uniform Resource Identifier (URI).

1. 抽象的または物理的なリソースを識別するための簡潔な文字ストリング。

2. テキストのページ、ビデオ・クリップやサウンド・クリップ、静止画や動画、またはプログラムなどの Web 上のコンテンツを識別するのに使用する固有のアドレス。URI の最も一般的な形式は Web アドレスである。Web アドレスは URI の特別な形式またはサブセットで URL (Uniform Resource Locator) と呼ばれる。通常 URI が表すのは、リソースへのアクセス方法、リソースを含むコンピューター、およびコンピューター上のリソース名 (ファイル名) を表す。

Uniform Resource Locator (URL). インターネットなどのネットワークでアクセス可能な情報リソースの固有アドレス。URL には、情報リソースへのアクセスに使用されるプロトコルの省略名と情報リソースを位置指定するためにプロトコルが使用する情報が含まれている。

Uniform Resource Name (URN). クライアントに対し Web サービスを一意的に識別する名前。

Universal Description, Discovery, and Integration (UDDI). 会社およびアプリケーションがインターネット上で迅速かつ容易に Web サービスを検索および利用できるようにする、標準ベースの仕様のセット。

Universally Unique Identifier (UUID). 2 つのコンポーネントが同じ ID を持たないようにするために使用される 128 ビットの数値 ID。

UNIX システム・サービス (UNIX System Services). XPG4 UNIX 1995 仕様に準拠した UNIX 環境を構築する z/OS のエレメント。 z/OS オペレーティング・システム上で、アプリケーション・プログラミング・インターフェース (API) および対話式シェル・インターフェースという 2 つのオープン・システム・インターフェースを提供する。

URI. 「Uniform Resource Identifier」を参照。

URL. 「Uniform Resource Locator」を参照。

URL スキーム (URL scheme). 別のオブジェクト参照を含む形式。

URN. 「Uniform Resource Name」を参照。

UUID. 「汎用固有 ID (Universally Unique Identifier)」を参照。

version. 通常では重要な新規コードまたは新規機能を備えている、別個にライセンスされるプログラム。

WAR. 「Web アーカイブ (Web archive)」を参照。

WCCM. 「WebSphere Common Configuration Model」を参照。

Web アーカイブ (Web archive (WAR)). 単一ファイルで Web アプリケーションをインストールおよび実行するために必要なすべてのリソースを保管するための、Java EE 標準で定義された圧縮ファイル・フォーマット。「エンタープライズ・アーカイブ (enterprise archive)」も参照。

Web クローラー (Web crawler). Web 文書を検索して、その文書内のリンクをたどることにより Web を探索するタイプのクローラー。

Web コンテナー (Web container). Java EE アーキテクチャーの Web コンポーネント規約を実装するコンテナー。

Web コンテナー・チャネル (Web container channel). トランスポート・チェーン内のチャネルの一種。HTTP インバウンド・チャネルとサーブレットまたは JavaServer Pages (JSP) エンジン間のトランスポート・チェーン内にブリッジを作成する。

Web コンポーネント (Web component). サーブレット、JavaServer Pages (JSP) ファイル、またはハイパーテキスト・マークアップ言語 (HTML) ファイル。 Web モジュールは、1 つ以上の Web コンポーネントによって構成される。

Web サーバー (Web server). Hypertext Transfer Protocol (HTTP) 要求のサービスを提供できるソフトウェア・プログラム。

Web サーバー・プラグイン (Web server plug-in). サーブレットのような動的コンテンツの要求においてアプリケーション・サーバーと通信するために、Web サーバーをサポートするソフトウェア・モジュール。

Web サーバーの分離 (Web server separation). Web サーバーがアプリケーション・サーバーから物理的に分離しているトポロジー。

Web サイト (Web site). 単一エンティティー (1 組織または 1 個人) によって管理され、そのユーザーのためにハイパーテキストの形での情報が含まれている、Web 上で使用可能な関連するファイルの集合。Web サイトには、他の Web サイトへのハイパーテキスト・リンクが含まれていることがある。

Web ブラウザー (Web browser). Web サーバーへの要求を開始し、サーバーが戻す情報を表示するクライアント・プログラム。

WebSphere. e-business アプリケーションおよび Web アプリケーションを実行するミドルウェアの開発用のツールを包含する IBM 製品ブランド名。

WebSphere Common Configuration Model (WCCM). 構成データにプログラムによってアクセスするためのモデル。

what you see is what you get (WYSIWYG). 印刷、またはレンダリング時とまったく同様のページを表示するエディターの機能。

while ループ (while loop). ある条件が満足される限りアクティビティーの同じシーケンスを反復するループ。while ループでは、ループを開始するたびにその条件がテストされる。開始から条件が偽である場合、そのループに含まれる一連のアクティビティーは実行されない。

WLM. 「ワークロード・マネージャー (Workload Manager)」を参照。

WYSIWYG. 「what you see is what you get」を参照。

X/Open XA. X/Open 分散トランザクション処理 XA インターフェース。分散トランザクション通信用に提案された標準である。この標準では、トランザクション内の共用リソースを知る方法を提供するリソース・マネージャー間の双方向インターフェース、およびトランザクションをモニターして解決するトランザクション・サービス間の双方向インターフェースが規定されている。

XA. 共用リソースへのアクセスを可能にする 1 つ以上のリソース・マネージャーとトランザクションをモニターして解決するトランザクション・マネージャーとの間にある双方向インターフェース。

XML. 「Extensible Markup Language」を参照。

z/OS. 64 ビットの実ストレージを使用する IBM のメインフレーム用オペレーティング・システム。

特記事項

本書に記載の製品、プログラム、またはサービスが日本においては提供されていない場合があります。日本で利用可能な製品、プログラム、またはサービスについては、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。IBM 製品、プログラムまたはサービスに代えて、IBM の知的所有権を侵害することのない機能的に同等のプログラムまたは製品を使用することができます。ただし、IBM によって明示的に指定されたものを除き、他社の製品と組み合わせた場合の動作の評価と検証はお客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

商標

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

LINUX は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft、Windows、Windows NT および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

Printed in Japan

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21