WebSphere eXtreme Scale バージョン 7.0
製品概要

WebSphere eXtreme Scale 製品概要

WebSphere eXtreme Scale バージョン 7.0
製品概要

WebSphere eXtreme Scale 製品概要

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

製品概要 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、WebSphere eXtreme Scale の学習に関心をお持ちの方々を対象にしています。

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 の概要が含まれています。
- **第 2 章**には、製品のキャッシングの概念に関する情報が含まれています。
- **第 3 章**には、キャッシュ統合に関する情報が含まれています。
- **第 4 章**には、スケーラビリティーに関する情報が含まれています。
- **第 5 章**には、可用性に関する情報が含まれています。
- **第 6 章**には、セキュリティーに関する情報が含まれています。
- **第 7 章**には、トランザクション処理に関する情報が含まれています。
- **第 8 章**には、基本的な製品概念のチュートリアルが含まれています。
- **第 9 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概要

WebSphere eXtreme Scale は、弾力性があってスケーラブルな、メモリー内データ・グリッドです。これは、複数のサーバーにまたがって、アプリケーション・データおよびビジネス・ロジックを動的にキャッシュに入れ、区画化し、複製し、管理します。WebSphere eXtreme Scale は、高い効率性と直線的に伸びるスケーラビリティーを備えた、大量のトランザクション処理を実現し、トランザクションの整合性、高可用性、予測可能な応答時間などの高いサービス品質を提供します。

WebSphere eXtreme Scale の弾力性のあるスケーラビリティーは、分散オブジェクト・キャッシングによって実現されています。弾力性があるというのは、グリッドがそれ自体をモニターし、管理すること、スケールアウトとスケールインが可能であること、自動的に障害を復旧して自己修復することを意味しています。スケールアウトによって、グリッドの実行中に再始動を必要とせずにメモリー容量を追加することができます。逆に、スケールインは、メモリー容量の即時除去を可能にします。

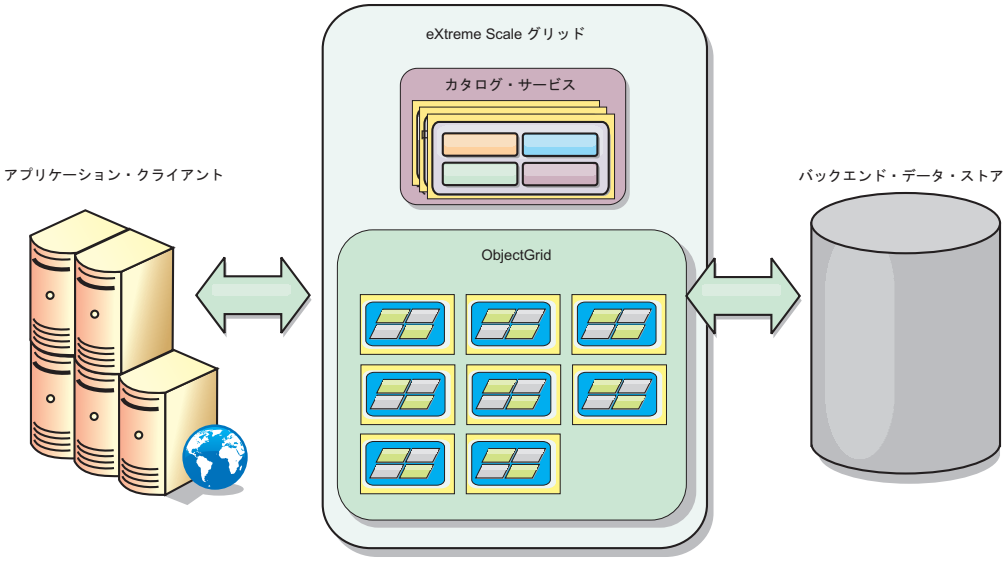
WebSphere eXtreme Scale の使用方法はいくつかあります。つまり、非常に強力なキャッシュとしても、アプリケーション状態を管理する一種のメモリー内データベース処理スペースとしても、また強力な Extreme Transaction Processing (XTP) アプリケーションを構築するためのプラットフォームとしても使用することができます。

ただし、eXtreme Scale を実際のメモリー内データベースと見なすことはできず、その理由の大半は、後者については単純すぎて eXtreme Scale が管理できる複雑さを取り扱えない場合がある点に注意してください。両方のシナリオはいずれもメモリーに入っているため、どちらのシナリオでも同じ利点がいくつかありますが、メモリー内データベースが障害のあるマシンを持っている場合は、この問題をすぐには修復できません。このような結果は、使用環境全体がその 1 つのマシン上にある場合、特に致命的となります。

この種の障害に関する問題に対処するため、eXtreme Scale は所与のデータ・セットを区画に分割し (これはコンストレインド・ツリー・スキーマに相当します)、それぞれの区画がプライマリー・コピー (断片) として存在し、かつデータのバックアップのための複製断片としても存在するようにします。メモリー内データベースはこの種の機能を提供できません。それというのも、メモリー内データベースはこのように構造化されておらず、かつ動的でもなく、eXtreme Scale が自動的に行うことを手動で行わなければならないからです。さらに、メモリー内データベースは当然データベースであるため、SQL 操作を可能にし、メモリー内でないデータベースと比較して処理速度の点でかなり向上しています。WebSphere eXtreme Scale は、SQL サポートよりも独自の照会言語を備えていますが、極めて弾力性に富み、データの区画化を可能にし、信頼性の高い障害リカバリーを提供します。

また、後書きキャッシュ機能によって、WebSphere eXtreme Scale はデータベースのフロントエンド・キャッシュとして働くことができ、データベースの負荷と競合を減らし、かつ、スループットを増やします。WebSphere eXtreme Scale は、予測可能な処理コストで予測可能なスケールインおよびスケールアウトを提供します。

以下の図は、コヒーレントな分散キャッシュ環境で、eXtreme Scale グリッドのクライアントとグリッド間とでデータがやり取りされ、それがバックエンド・データ・ストアで自動的に同期化されることを示しています。すべてのクライアントがキャッシュ内の同じデータを見るので、キャッシュはコヒーレントです。データの各部分はキャッシュ内の 1 つだけの書き込み可能サーバーに保管され、さまざまなバージョンのデータを保管することになりかねない、レコードの無駄なコピーを防止します。コヒーレントなキャッシュは、より多くのサーバーがグリッドに追加されるにつれて、より多くのデータを保持し、グリッドのサイズが増えるのにつれて直線的に増加します。耐障害性を強化するため、オプションでデータを複製することもできます。



WebSphere eXtreme Scale には、メモリー内データ・グリッドを提供するサーバーがあります。これらのサーバーは、WebSphere Application Server 内部でも、単純な Java™ Standard Edition (J2SE) Java 仮想マシン上でも実行でき、1 つの物理マシンで複数のサーバーを実行することが可能です。したがって、メモリー内データ・グリッドは、非常に大きくなる場合があります。これは、アプリケーションまたはアプリケーション・サーバーのメモリーまたはアドレス・スペースに影響することも、それらによって制限を受けることもありません。多数のマシン上で稼働する何千、何万の Java 仮想マシンのメモリーを合計したメモリーを使用できます。

それでも、WebSphere eXtreme Scale を、メモリー内データベース処理スペースとして、ディスク、データベース、またはその両方でバックアップすることができます。

eXtreme Scale にはいくつかの Java API が用意されていますが、多くの場合はユーザー・プログラミングは必要なく、ユーザーの WebSphere インフラストラクチャーでの構成とデプロイを行えばいいだけです。

基本的なパラダイム

グリッドの基礎となるパラダイムは、キーと値のペアです。グリッドは値 (Java オブジェクト) を、関連付けられたキー (別の Java オブジェクト) と一緒に保管し、

それ以降の値の取り出しにはキーが使用されます。eXtreme Scale では、このようなキーと値のペアを格納するコンテナーをマップと呼びます。

WebSphere eXtreme Scale は、単一の 1 つだけのローカル・キャッシュから、複数の JVM あるいはサーバー (またはその両方) を使用する大容量の分散キャッシュにいたるまで、多くのグリッド構成を提供します。

単純な Java オブジェクトを保管することに加えて、リレーションシップを持つオブジェクトを保管することもでき、SQL に似た照会言語 (SELECT ... FROM ... WHERE) を使用してそれらのオブジェクトを取り出すことができます。例えば、1 つの注文オブジェクトが 1 つの顧客オブジェクトを持つことができ、複数の品目オブジェクトをその注文オブジェクトに関連付けることができるといった例が考えられます。WebSphere eXtreme Scale は、1 対 1、1 対多、多対 1、および多対多のリレーションシップをサポートします。

WebSphere eXtreme Scale は、EntityManager プログラミング・インターフェースもサポートします。これはキャッシュにエンティティーを保管するためのもので、Java Enterprise Edition エンティティーによく似ています。エンティティー・リレーションシップは、エンティティー記述子 XML ファイルから、または Java クラス内のアノテーションから、自動的に検出できます。したがって、 EntityManager find メソッドを使用して、1 次キーによってキャッシュからエンティティーを取り出すことができます。エンティティーをグリッドにパーシストすることも、グリッドから除去することも、すべて 1 トランザクション境界内で実行できます。

WebSphere eXtreme Scale は、ビジネス上重要で要求が厳しいアプリケーションをサポートするためのさらにスマートなアプリケーション・インフラストラクチャーを実現する、Extreme Transaction Processing (XTP) 機能を提供します。よりスマートな成果を得るため、および持続可能で競争力のあるビジネス上の優位性を獲得するために必要な、世界的レベルのスケール、処理効率、ビジネス・インテリジェンスを、従来的な IT パフォーマンス制約を克服して作り出すことができます。

WebSphere eXtreme Scale は、業界をリードするリアルタイム Java オファリングである WebSphere Real Time のサポートにより、XTP アプリケーションの応答時間をより一貫性のある予測可能なものにできます。*管理ガイド*にある Real Time サポートに関する情報を参照してください。

実稼働環境に eXtreme Scale をデプロイする前に、使用するサーバーの数、各サーバーのストレージ容量、同期または非同期複製など、いくつかのオプションについて検討する必要があります。

単純な英字名がキーになっている分散例を考えてみましょう。キー別にキャッシュを 4 つの区画に分割できます。区画 1 は A から E で始まるキー、区画 2 は F から L で始まるキーに対応し、以下同様にします。可用性のため、1 つの区画が 1 つのプライマリー断片と 1 つの複製断片を持つことにします (区画はこれらの断片に保管されます)。キャッシュ・データに対する変更は、プライマリー断片に対して行われ、2 次断片 (複製断片) に複製されます。分散キャッシュ (eXtreme Scale 用語ではグリッドまたは ObjectGrid とも呼ばれる) に対して、グリッド・データを収容する eXtreme Scale サーバーの数を構成します。そうすると、eXtreme Scale はこれらのサーバー・インスタンス全部を対象にして断片にデータを分配します。可用性のため、複製断片はプライマリー断片とは別のマシンに置かれます。

WebSphere eXtreme Scale は、カタログ・サービスを使用して、各キーのプライマリー断片を配置します。eXtreme Scale は、eXtreme Scale サーバーに障害が起こるか、またはサーバーが含まれている物理マシンに障害が起こり、その後で復旧されるようなことがあると、サーバー間での断片の移動を処理します。例えば、複製断片を含んでいるサーバーに障害が起こった場合、eXtreme Scale は新しい複製断片を割り振ります。プライマリー断片を含んでいるサーバーに障害が起こった場合は、複製断片がプロモートされてプライマリー断片になり、上記と同じように、新しい複製断片が作成されます。

最も単純な eXtreme Scale プログラミング・インターフェースが ObjectMap です。これは単純なマップ・インターフェースであり、map.put(key,value) でキャッシュに値を入れ、次に map.get(key) で値を取り出します。

単純な Java オブジェクトをキャッシュに入れることに加えて、リレーションシップを持つオブジェクトもキャッシュに入れることができ、SQL に似た照会言語 (SELECT ... FROM ... WHERE) を使用してそれらのオブジェクトを取り出すことができます。例えば、1 つの注文オブジェクトが 1 つの顧客オブジェクトを持つことができ、複数の品目オブジェクトをその注文オブジェクトに関連付けることができるといった例が考えられます。WebSphere eXtreme Scale は、1 対 1、1 対多、多対 1、および多対多のリレーションシップをサポートします。 eXtreme Scale では EntityManager プログラミング・インターフェースもサポートされます。これはキャッシュにエンティティーを保管するためのもので、Java Enterprise Edition エンティティーによく似ています。エンティティー・リレーションシップは、エンティティー記述子 XML ファイルから、または Java クラス内のアノテーションから、自動的に検出できます。したがって、 EntityManager find メソッドを使用して、1 次キーによってキャッシュからエンティティーを取り出すことができます。エンティティーをグリッドにパーシストすることも、グリッドから除去することも、すべて 1 トランザクション境界内で実行できます。

このリリースの新機能と非推奨機能

WebSphere eXtreme Scale のバージョン 7.0 には、動的キャッシュとの統合、バイト配列マップなど、多くの新機能があります。

WebSphere eXtreme Scale バージョン 7.0 の新機能

表 1. WebSphere eXtreme Scale バージョン 7.0 の新機能

機能	説明
動的キャッシュの統合	動的キャッシュ・プロバイダーにより、WebSphere Application Server 動的キャッシュを使用するアプリケーションは、 WebSphere eXtreme Scale の拡張機能とパフォーマンス改善を活用できます。この機能を使用すると、サービスの品質向上、リニアなスケーラビリティー、高可用性が実現し、幅広いビジネス・アプリケーションへの変更の影響を最小限に抑えることができます。 *製品概要*にある動的キャッシュの情報を参照してください。
バイト配列マップ	バイト配列マップを使用すると、アプリケーションはキー値ペアの値を、オブジェクト形式の代わりにバイト配列に保管でき、その結果、オブジェクトの大きなグラフに消費されるメモリー占有スペースを節約できます。*プログラミング・ガイド*にあるバイト配列マップに関する情報を参照してください。
WebSphere Real Time	WebSphere Real Time (業界最先端のリアルタイム Java 製品) のサポートにより、WebSphere eXtreme Scale は、XTP アプリケーションが、より安定した予測可能な応答時間を得られるようにします。*管理ガイド*にある Real Time サポートに関する情報を参照してください。
メトリックの使用可能化	WebSphere eXtreme Scale には、IBM® Tivoli® Monitoring (ITM) と Hyperic HQ との統合を拡充するためのメトリック・アクセス・アダプターの実装が組み込まれており、これにより、ビジネス・ソリューションの動作状況を包括的に把握できるようになります。 *管理ガイド*にあるメトリック使用可能化ベンダー・モニター・ツールに関する情報を参照してください。

表 1. WebSphere eXtreme Scale バージョン 7.0 の新機能 (続き)

機能	説明
動的マップ	マルチテナント・アプリケーションの構築が大幅に単純化されています。マップ・テンプレートによってアプリケーションは要求時に新しいマップを作成できるため、キー内のアプリケーション弁別子を使用する必要性や、使用されない可能性がある余分なマップの作成を回避できます。*プログラミング・ガイド*にある動的マップに関する情報を参照してください。
要求タイムアウト	WebSphere eXtreme Scale は、グリッド・ミドルウェア内の多くの一般的な再試行および例外ロジック・タスクを処理するように拡張されています。クライアントの要求タイムアウトにより、開発者が大半のマップ相互作用操作のために定型的な再試行ロジックを使用する負担はなくなります。大半の再試行可能条件は自動的に処理されるようになったため、開発者は、アプリケーション開発のビジネス・ロジック局面に集中できます。*管理ガイド*にある要求タイムアウトに関する情報を参照してください。
複合索引	この機能は、複数の属性の照会時における索引の使用を単純化して、複数の索引を定義するオーバーヘッドを削減することができます。また、照会も最適化されて、複合索引が活用されるようになっています。 *プログラミング・ガイド*にある複合索引に関する情報を参照してください。

非推奨機能

表 2. 非推奨機能

非推奨機能	推奨されるマイグレーション・アクション
区画化機能 (WPF): 区画化機能は、プログラミング API のセットで、これにより Java EE アプリケーションは非対称クラスタリングをサポートできるようになります。	WPF の機能は、また、WebSphere eXtreme Scale でも実現することができます。
StreamQuery: ObjectGrid マップに保管されている伝送中のデータに対して連続的に行う照会。	なし
静的グリッド構成: クラスター・デプロイメント XML ファイルを使用する静的なクラスター・ベースのトポロジー。	大容量データ・グリッドを管理するために改善された動的デプロイメント・トポロジーに置き換えられています。
非推奨システム・プロパティー: サーバーおよびクライアントのプロパティー・ファイルを指定するシステム・プロパティーは推奨されていません。	これらの引数は、まだ使用できますが、ご使用のシステム・プロパティーを新しい値に変更してください。詳しくは、 *管理ガイド*にあるプロパティー・ファイルに関する情報を参照してください。

無料試用

WebSphere eXtreme Scale を使用し始める前に、無料試用版をダウンロードしてください。拡張機能を使用してデータ・キャッシング概念を拡張することにより、革新的で高性能なアプリケーションを開発することができます。

試用版のダウンロード

eXtreme Scale 試用版のダウンロードから、eXtreme Scale の無料試用版をダウンロードすることができます。

試用版の eXtreme Scale のダウンロードと unzip が完了したら、gettingstarted ディレクトリーに移動して GETTINGSTARTED_README.txt をお読みください。 このチュートリアルには eXtreme Scale の使用を開始するための概要、複数のサーバーにグリッドを作成する方法、グリッド内のデータを保管または取得する簡単なアプリケーションの実行方法などが説明されています。実稼働環境に eXtreme Scale をデプロイする前に、使用するサーバーの数、各サーバーのストレージ容量、同期または非同期複製など、いくつかのオプションについて検討する必要があります。

WebSphere eXtreme Scale の使用

WebSphere eXtreme Scale は、弾力性があってスケーラブルな、メモリー内データ・グリッドです。これは、複数のサーバーにまたがって、アプリケーション・データおよびビジネス・ロジックを動的にキャッシュに入れ、区画化し、複製し、管理します。

eXtreme Scale はメモリー内データベースではないため、これに固有の構成要件を考慮する必要があります。eXtreme Scale データ・グリッドをデプロイするための最初のステップは、コア・グループおよびカタログ・サービスを開始することです。これらは、グリッドに参加している他のすべての Java 仮想マシンの調整役を果たし、構成情報を管理します。WebSphere eXtreme Scale プロセスは、コマンド行からの単純なコマンド・スクリプト呼び出しで開始されます。

次のステップは、グリッドがデータを保管および取得するための WebSphere eXtreme Scale サーバー・プロセスを開始することです。開始されたサーバーは、自動的にコア・グループおよびカタログ・サービスに登録され、連携してグリッド・サービスを提供できるようになります。サーバーが多いほど、グリッドの容量も信頼性も高まります。

ローカル・グリッドは、単一の、単一インスタンスのグリッドであり、1 つのグリッド内にすべてのデータがあります。eXtreme Scale をメモリー内データベース処理スペースとして効果的に使用するように、分散グリッドを構成し、デプロイすることができます。分散グリッドのデータは、これを含む各種 eXtreme Scale サーバーに分散されます。つまり、データは、各サーバーが区画と呼ばれるデータの一部のみを含むように分散されます。

分散グリッド構成パラメーターのうち最も重要なパラメーターが、グリッド内の区画の数です。グリッド・データは、この数のサブセットに分割されます (それぞれのサブセットを区画と呼びます)。 カタログ・サービスは、データの区画を、区画のキーに基づいて配置します。区画数は、グリッドの容量とスケーラビリティーに直接影響します。1 つのサーバーが 1 つ以上のグリッド区画を含むことができます。したがって、区画のサイズはサーバーのメモリー・スペースによって制限されます。逆に、区画の数を増やすと、グリッドの容量は増加します。グリッドの最大容量は、区画数に、1 つのサーバー (JVM であることも可能です) で使用可能なメモリー・サイズを掛けたものです。

1 つの区画のデータは 1 つの断片に保管されます。可用性のため、複製 (同期または非同期のどちらでも可) を持つようにグリッドを構成できます。 グリッド・データに対する変更は、プライマリー断片に対して行われ、複製断片に複製されます。したがって、グリッドで消費される/必要とされるメモリー合計は、グリッドのサイズに (1 (プライマリー) + 複製の数) を掛けたものになります。

WebSphere eXtreme Scale は、グリッドの断片を、そのグリッドを含むサーバーの数だけサーバーに分散させます。それらのサーバーは、同じ物理マシンにある場合も、別の物理マシンにある場合もあります。可用性のため、複製断片はプライマリー断片とは別のマシンに置かれます。

WebSphere eXtreme Scale は、サーバーの状態をモニターし、サーバーまたはサーバーが含まれている物理マシン (あるいはその両方) に障害が起こり、その後で復旧さ

れるようなことがあると、サーバー間で断片を移動します。 例えば、複製断片を含んでいるサーバーに障害が起こった場合、eXtreme Scale は新しい複製断片を割り振り、その新規複製にプライマリーからデータを複製します。 プライマリー断片を含んでいるサーバーに障害が起こった場合は、複製断片がプロモートされてプライマリー断片になり、上記と同じように、新しい複製断片が作成されます。グリッド用に追加サーバーを開始した場合、各サーバーの負荷ができるだけ均衡するように、すべてのサーバーに断片が分配されます。これをスケールアウトと呼びます。同じように、サーバーの 1 つを停止して、グリッドが消費するリソースを削減することができ、それをスケールインと呼びます。これを行うと、障害が起こった場合と同じように、残りのサーバー間で均衡するように断片が分配されます。

製品名の変更について

WebSphere eXtreme Scale は以前はこの名前ではなかったことに注意してください。

製品名の変更について

他の文書、マーケティング資料、またはプレゼンテーションを参照するときには、eXtreme Scale は以前は次のような名称だったことに気をつけてください。

- ObjectGrid
- WebSphere Extended Deployment Data Grid

この製品自体は現在は WebSphere eXtreme Scale という名称ですが、ObjectGrid という語は、グリッド・テクノロジーを可能にする成果物の名前として資料などに出現します。

アプリケーション・デプロイメントの計画

WebSphere eXtreme Scale を実稼働環境にデプロイする前に、以下のオプションを検討してください。

アプリケーション・デプロイメントの計画

次のリストに、検討項目を示します。

- システムおよびプロセッサーの数: 環境内には物理マシンとプロセッサーがいくつ必要ですか?
- サーバーの数: いくつの eXtreme Scale サーバーが eXtreme Scale マップをホストしますか?
- 区画の数: マップ内に保管されるデータの量は、必要な区画の数を決定する 1 つの要因です。
- レプリカの数: ドメイン内の各プライマリーに対してレプリカがいくつ必要ですか?
- 同期または非同期複製: データがきわめて重要であるため、同期複製が必要ですか? それとも、パフォーマンスに高い優先度を置くため、非同期複製が適切な選択ですか?
- ヒープ・サイズ: 各サーバーには、どれほどのデータが保管されますか?

プログラミング・ガイドおよび管理ガイド

「*製品概要*」は、WebSphere eXtreme Scale を理解するために必要な基本概念を説明しています。本書に記述された概念をさらに詳細に説明する 2 つの追加ガイドがあります。

構成および一般的な管理タスクについては「*管理ガイド*」を使用し、eXtreme Scale グリッドにアクセスしたりグリッドを構成するための Java API の説明については「*プログラミング・ガイド*」を使用してください。

他の WebSphere Application Server 製品との統合

WebSphere eXtreme Scale を他のサーバー製品 (WebSphere Application Server や WebSphere Application Server Community Edition など) と統合することができます。

WebSphere Application Server Community Edition と連動する HTTP セッション・マネージャーの構成

WebSphere Application Server Community Edition はセッション状態を共有できますが、効率的でスケーラブルな方法ではありません。 WebSphere eXtreme Scale は、状態の複製に使用できるハイパフォーマンスな分散パーシスタンス層を提供しますが、WebSphere Application Server の外部にあるアプリケーション・サーバーと容易には統合しません。この 2 つの製品を統合することで、スケーラブルなセッション管理ソリューションを提供することができます。詳しくは、「*管理ガイド*」を参照してください。

WebSphere Application Server と連動する WebSphere eXtreme Scale セッション・マネージャーの構成

HTTP セッション・マネージャーが初めて同梱されたのは WebSphere Extended Deployment DataGrid バージョン 6.1.0.0 です。それ以降、バージョン 6.1.0.5 までのバージョンでは、統合およびセッション取得に関する Jave 2 Enterprise Edition (J2EE) 仕様に合致していたため利用方法は変更しませんでしたが、各リリースでパフォーマンスおよび QoS の強化は行われてきました。 最高のサービス品質を得られるようにするため、 WebSphere eXtreme Scale バージョン 6.1.0.5 フィックスパックを適用することをお勧めします。

詳しくは、「*管理ガイド*」を参照してください。

第 2 章 キャッシングの概念

WebSphere eXtreme Scale は、データベース・バックエンドにインライン・キャッシングを提供するために使用するか、サイド・キャッシュとして使用できるメモリー内のデータベース処理スペースとして機能できます。インライン・キャッシングは、データと対話するための基本手段として eXtreme Scale を使用します。eXtreme Scale をサイド・キャッシュとして使用する場合は、eXtreme Scale と連動してバックエンドが使用されます。このセクションでは、さまざまなキャッシングの概念やシナリオを説明し、eXtreme Scale グリッドをデプロイするために使用可能なトポロジーについても説明します。

アーキテクチャーおよびトポロジー

ローカルのメモリー内でのデータ・キャッシング、および分散クライアント/サーバーでのデータ・キャッシングを、WebSphere eXtreme Scale を使用して構成できます。

WebSphere eXtreme Scale を作動させるには、最低限の追加インフラストラクチャーが必要です。インフラストラクチャーは、サーバー上で Java Platform, Enterprise Edition アプリケーションをインストール、開始、および停止するためのスクリプトで構成されます。キャッシュ・データは eXtreme Scale サーバー内に保管され、クライアントはリモート側でサーバーに接続します。

分散キャッシュは、より高いパフォーマンス、可用性、およびスケーラビリティーをもたらすもので、動的トポロジーを使用して構成できます。こうした構成では、サーバーのバランスが自動的に取られます。また、既存の eXtreme Scale サーバーを再始動せずに、別のサーバーを追加することもできます。単純なデプロイメントを作成することも、数千ものサーバーが必要になる大規模なテラバイト・サイズのデプロイメントを作成することもできます。

マップ

マップはキーと値のペアを格納するコンテナーであり、アプリケーションはマップを使用して、キーで索引付けされた値を保管できます。マップでは、キーまたは値の索引属性に追加できる索引がサポートされます。こうした索引が照会ランタイムによって自動的に使用され、照会を実行するのに最も効率的な方法が決定されます。

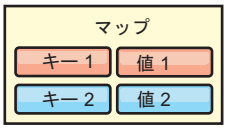


図 1. マップ

マップ・セットは、共通の区画化アルゴリズムを持つマップの集合です。マップ内のデータは、マップ・セットに定義されたポリシーに基づいて複製されます。マッ

プ・セットは分散トポロジーでのみ使用され、ローカル・トポロジーでは必要ありません。

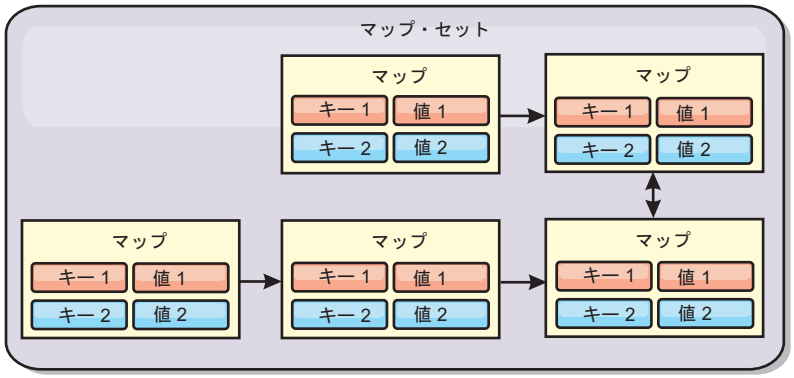


図 2. マップ・セット

マップ・セットには、関連のあるスキーマを含めることができます。スキーマとは、同種のオブジェクト・タイプまたはエンティティーを使用している場合に各マップ間のリレーションシップを記述したメタデータです。

eXtreme Scale は、ObjectMap API を使用して、シリアライズ可能な Java オブジェクトを各マップに保管できます。スキーマはマップ全体に定義することができ、それぞれのマップが単一の型のオブジェクトを保持している場合に、それらのマップ内のオブジェクト間のリレーションシップを表します。マップ・オブジェクトの内容を照会するには、マップにスキーマを定義しておく必要があります。eXtreme Scale では、複数のマップ・スキーマを定義できます。詳しくは、*プログラミング・ガイド*の ObjectMap API の説明を参照してください。

eXtreme Scale は、EntityManager API を使用して、エンティティーを保管することもできます。各エンティティーは、マップに関連付けられています。エンティティー・マップ・セットのスキーマは、エンティティー記述子 XML ファイルまたはアノテーション付き Java クラスのどちらかを使用して自動的に検出されます。各エンティティーは、キー属性のセット、および非キー属性のセットを持ちます。また、他のエンティティーへのリレーションシップも持つことができます。eXtreme Scale では、1 対 1、1 対多、多対 1、および多対多のリレーションシップがサポートされています。各エンティティーは、マップ・セット内の単一のマップに物理的にマップされます。エンティティーにより、複数のマップにまたがる複雑なオブジェクト・グラフを簡単にアプリケーションに装備できます。分散トポロジーは、複数のエンティティー・スキーマを持つことができます。詳しくは、*プログラミング・ガイド*の EntityManager API の説明を参照してください。

コンテナー、区画、および断片

コンテナーは、グリッドのアプリケーション・データを保管するサービスです。通常、このデータは区画と呼ばれるパーツに分割され、複数のコンテナーでホストされます。これを受けて各コンテナーは、完全なデータのサブセットをホストします。JVM は 1 つ以上のコンテナーをホストすることができ、各コンテナーは複数の断片をホストできます。

要確認: すべてのデータをホストするコンテナーのヒープ・サイズを計画してください。それにあわせて、ヒープ設定を適宜構成してください。

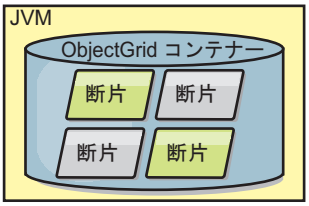


図 3. コンテナー

区画は、グリッド内のデータのサブセットをホストします。eXtreme Scale は、自動的に複数の区画を単一コンテナーに配置し、追加のコンテナーが使用可能になるとそれらに区画を分散させます。

重要: 区画の数は動的に変更できないため、最終的なデプロイメントの前に、区画の数を慎重に選択してください。ネットワーク内での区画の配置にはハッシュ・メカニズムが使用され、いったんデプロイされた後にデータ・セット全体を ObjectGrid が再ハッシュする方法はありません。区画の数は多めに見積もってください。

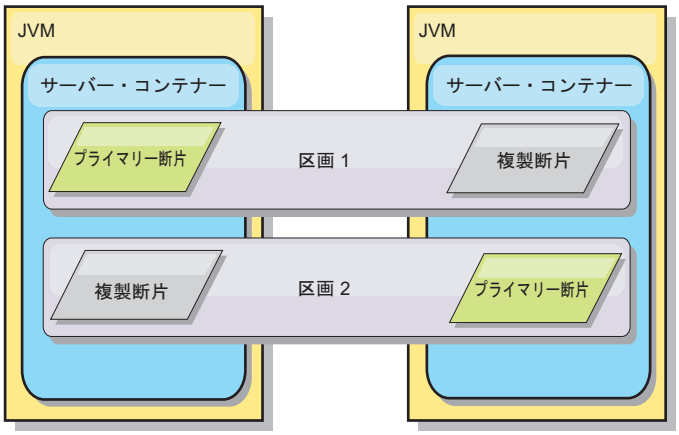


図 4. 区画

断片とは区画のインスタンスであり、プライマリーまたはレプリカの 2 つのロールのいずれか 1 つを持ちます。プライマリー断片とそのレプリカによって、区画の物理的な実体が構成されます。各区画はいくつかの断片を持ち、それぞれの断片が、その区画に含まれるデータ全体をホストします。 1 つの断片がプライマリー断片であり、他は複製断片です。複製断片は、プライマリー断片に含まれているデータの冗長コピーです。プライマリー断片は、トランザクションからキャッシュへの書き込みが可能な唯一の区画インスタンスです。複製断片は、「ミラーリングされた」区画インスタンスです。複製断片は、同期または非同期にプライマリー断片から更新内容を受信します。複製断片の場合、トランザクションはキャッシュからの読み取りのみが可能です。レプリカは、プライマリーと同じコンテナーではホストされません。また、通常はプライマリーと同じマシン上ではホストされません。

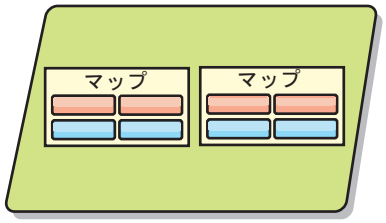


図 5. 断片

データの可用性を向上させる、または永続性の保証を高めるには、データを複製する必要があります。ただし、複製はトランザクションのコストを増加させるため、可用性と引き換えにパフォーマンスが犠牲になります。eXtreme Scale では、同期複製と非同期複製のサポートに加え、同期と非同期の両方の複製モードを使用するハイブリッド複製モデルがサポートされるため、このコストをコントロールできます。同期複製断片は、データ整合性を保証するため、プライマリー断片のトランザクションの一部として更新内容を受信します。トランザクション完了の前に、プライマリーと同期複製の両方でトランザクションをコミットする必要があるため、同期複製では応答時間が倍の長さになることがあります。非同期複製断片は、パフォーマンスへの影響を制限するために、トランザクションがコミットされた後に更新内容を受信します。しかし、非同期複製では、プライマリーよりトランザクションがいくつか遅れることがあるため、非同期複製断片でデータ損失の可能性が生じます。

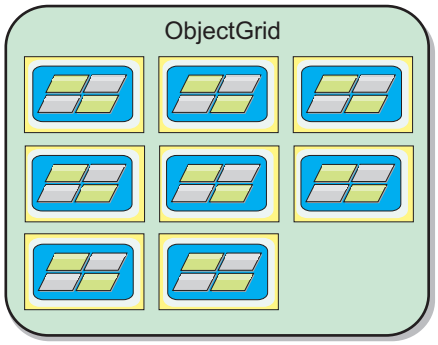


図 6. ObjectGrid

クライアント

クライアントは、カタログ・サービスに接続し、サーバー・トポロジーの記述を取得し、必要に応じて各サーバーと直接通信します。新規サーバーの追加または既存サーバーの障害などのためにサーバー・トポロジーが変更されると、クライアントは、データをホスティングする適切なサーバーに自動的にルーティングされます。クライアントは、アプリケーション・データのキーを調べて、要求をどの区画に送付するのかを決定しなければなりません。クライアントは、単一のトランザクションで複数の区画からデータを読み取ることができます。ただし、クライアントが更新できるのは、1 つのトランザクションで単一区画のみです。クライアントが複数のエントリーを更新した場合、クライアント・トランザクションはその区画を更新に使用する必要があります。

可能なデプロイメントの組み合わせが、次のリストに示されています。

- カタログ・サービスは、Java 仮想マシン内で自身のグリッド内に存在します。単一のカタログ・サービスを使用して、eXtreme Scale の複数インスタンスを管理できます。
- コンテナーは、JVM 内で単独で開始することも、別の ObjectGrid インスタンスの他のコンテナーと一緒に任意の JVM にロードすることもできます。
- クライアントは任意の JVM 内に存在でき、1 つ以上の ObjectGrid インスタンスと通信できます。また、クライアントはコンテナーと同じ JVM 内に存在することも可能です。

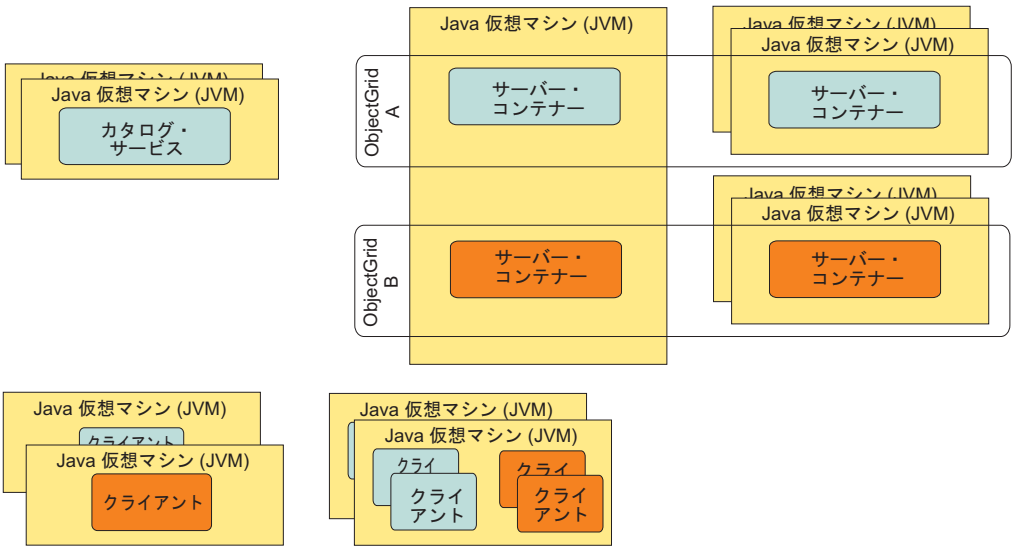


図 7. 可能なトポロジー

カタログ・サービス

カタログ・サービスは、定常状態ではアイドルになるロジックをホストし、スケーラビリティーにはほとんど影響しません。カタログ・サービスが構築されている目的は、同時に使用可能になる数百ものコンテナーにサービスを提供し、それらのコンテナーを管理するサービスを実行することです。

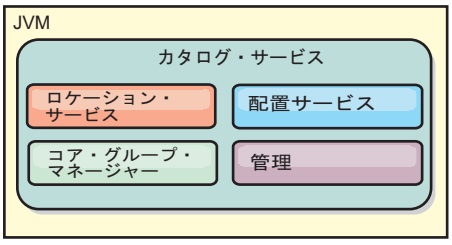


図 8. カタログ・サービス

カタログの役割分担には以下のサービスが含まれます。

ロケーション・サービス

ロケーション・サービスは、アプリケーションをホスティングするコンテナーを探しているクライアントと、ホスティングされるアプリケーションを配

置サービスに登録しようとしているコンテナーの局所性を提供します。ロケーション・サービスは、この機能をスケールアウトするために、すべてのグリッド・メンバーで実行されます。

配置サービス

配置サービスは、グリッドの中枢神経的な存在であり、個々の断片をホスト・コンテナーに割り振る責任を担います。配置サービスは、クラスター内で N 個の中から 1 つ選ばれたサービスとして実行されるため、配置サービスの実行中のインスタンスは常に 1 つのみです。そのインスタンスが停止する必要がある場合は、別のプロセスが引き継ぎます。カタログ・サービスのあらゆる状態は、予備のために、カタログ・サービスをホスティングするすべてのサーバーに複製されます。

コア・グループ・マネージャー

コア・グループ・マネージャーは、ヘルス・モニタリングのためにピアのグループ化を管理し、コンテナーを少数のサーバーからなるグループに編成し、サーバーのグループを自動的に統合します。初めてカタログ・サービスにコンタクトしたコンテナーは、いくつかの Java 仮想マシン (JVM) からなる新規または既存のグループのいずれかに割り当てられるのを待機します。Java 仮想マシンの各グループは、ハートビートを通してそれらの各メンバーの可用性をモニターします。再割り振りと経路転送によって障害に対処できるように、グループ・メンバーの 1 つが可用性情報をカタログ・サービスに中継します。

管理 WebSphere eXtreme Scale 環境の管理には、計画、デプロイ、管理、およびモニターの 4 つのステージがあります。各ステージの詳細については、「*管理ガイド*」を参照してください。

可用性のために、カタログ・サービス・グリッドを構成します。カタログ・サービス・グリッドは、複数の Java 仮想マシン (マスター JVM が 1 つと、多数のバックアップ Java 仮想マシン) から構成されます。

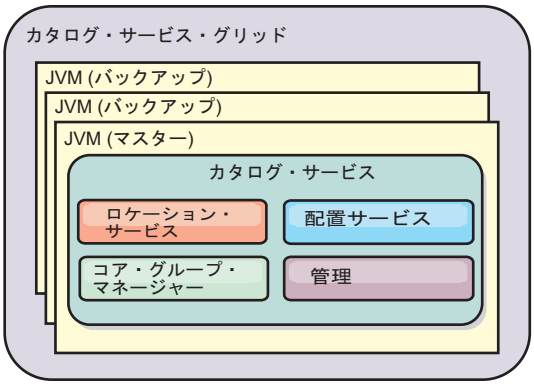


図 9. カタログ・サービス・グリッド

ローカルのメモリー内のキャッシュ

最も単純なケースでは、eXtreme Scale は、ローカルのメモリー内のデータ・グリッド・キャッシュとして使用できます。これは、特に複数のスレッドにより一時データにアクセスして変更する必要がある、高い並行性を持つアプリケーションで有効になります。ローカル eXtreme Scale グリッドに保持されるデータは、索引を付け、WebSphere eXtreme Scale の照会サポートを使用して検索することができます。データ照会を可能にする機能は、Java 仮想マシン (JVM) が提供するそのままの状態で作動可能な制限付きデータ構造サポートに比べ、開発者がメモリー内の大量のデータ・セットを処理する場合に非常に役に立ちます。

eXtreme Scale でのローカルのメモリー内キャッシュ・トポロジーは、単一 Java 仮想マシン内で、一時データへの整合したトランザクション・アクセスを可能にするために使用されます。

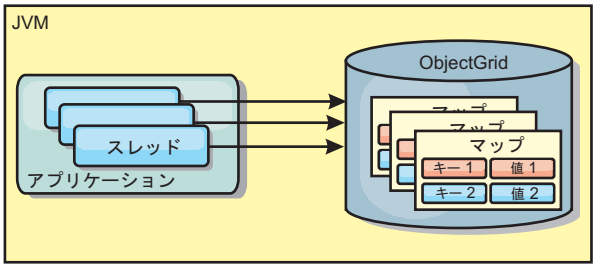


図 10. ローカルのメモリー内のキャッシュ・シナリオ

利点

- セットアップが簡単: ObjectGrid は、プログラマチックに作成することも、ObjectGrid デプロイメント記述子 XML ファイルまたは Spring などのその他のフレームワークを使用して宣言的に作成することもできます。
- 高速: 各 BackingMap は、最適のメモリー使用効率および並行性が得られるように独立して調整できます。
- 扱うデータ・セットが小さい単一 Java 仮想マシン・トポロジー、また頻繁にアクセスされるデータのキャッシングに最適。
- トランザクション型。BackingMap 更新は、単一の作業単位にまとめることができ、Java Transaction Architecture (JTA) トランザクションなどの 2 フェーズ・トランザクションの最終参加者として統合することができます。

欠点

- フォールト・トレラントでない。
- データは複製されない。メモリー内キャッシュは読み取り専用参照データに最適。
- スケーラブルでない。データベースが必要とするメモリーの量が Java 仮想マシンを圧倒するおそれがある。
- Java 仮想マシンを追加するときに、次のような問題が発生する。
  - データを簡単には区画化できない

- Java 仮想マシン間で状態を手動で複製しなければならない。そうしないと、各キャッシュ・インスタンスが同一データの別バージョンを保持するようになります
- 無効化にかかるコストが高い。
- 各キャッシュは個別にウォームアップが必要になる。ウォームアップは、有効なデータがキャッシュに設定されるようにデータをロードする期間です。

使用する場合

ローカルのメモリー内キャッシュのデプロイメント・トポロジーは、キャッシュに入れるデータ量が小さく (1 つの Java 仮想マシンに収まる場合)、比較的安定している場合に限って使用するようにしてください。このアプローチの場合、不整合データの存在を許容する必要があります。Evictor を使用して、最も使用頻度が高いデータまたは最近使用されたデータをキャッシュに保持するようにすると、キャッシュ・サイズを小さく維持し、データの関連性を高くすることができます。

ピア複製されるローカルのメモリー内キャッシュ

ローカル WebSphere eXtreme Scale キャッシュの制約の 1 つは、独立したキャッシュ・インスタンスに複数のプロセスがある場合、キャッシュが同期を保つことが困難である点です。

eXtreme Scale には、ピア eXtreme Scale インスタンス間にトランザクション変更を自動的に伝搬する 2 つのプラグインがあります。JMSObjectGridEventListener プラグインは、Java Messaging Service (JMS) を使用して、eXtreme Scale 変更を自動的に伝搬します。

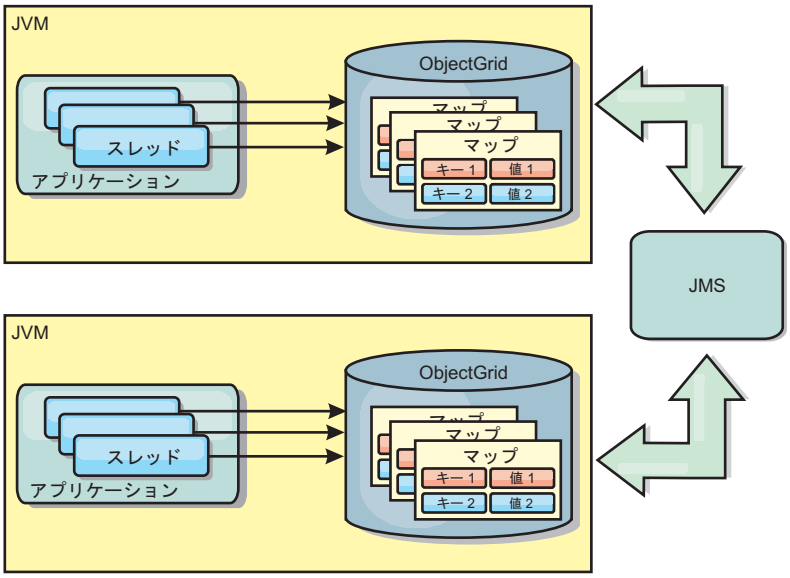


図 11. JMS によって変更が伝搬されるピア複製キャッシュ

WebSphere Application Server 環境を実行している場合は、TranPropListener プラグインも使用可能です。 TranPropListener プラグインは、高可用性 (HA) マネージャーを使用して、各ピア eXtreme Scale キャッシュ・インスタンスに変更を伝搬しま

す。

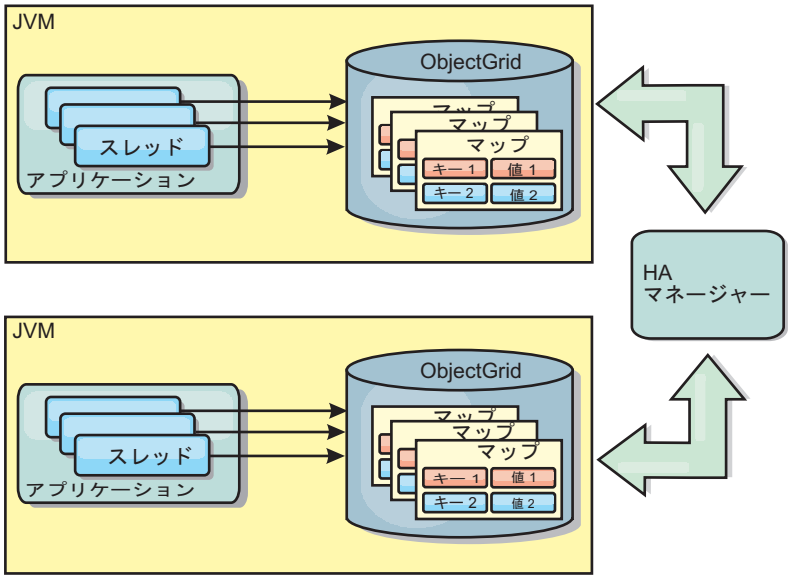


図 12. HA マネージャーによって変更が伝搬されるピア複製キャッシュ

利点

- より頻繁にデータが更新されるため、データが有効な場合が増えます。
- TranPropListener プラグインを使用すると、ローカル環境と同様、 eXtreme Scale デプロイメント記述子 XML ファイルや他のフレームワーク (Spring など) を使用して、eXtreme Scale をプログラマチックまたは宣言的に作成できます。HA マネージャーとの統合は自動的に行われます。
- 最適のメモリー使用効率および並行性が得られるように、各 BackingMap を独立して調整できます。
- BackingMap 更新は、単一の作業単位にまとめることができ、Java Transaction Architecture (JTA) トランザクションなどの 2 フェーズ・トランザクションの最終参加者として統合することができます。
- 十分小さなデータ・セットの少数 JVM トポロジー、または頻繁にアクセスされるデータのキャッシングに最適です。
- eXtreme Scale に対する変更は、すべてのピア eXtreme Scale インスタンスに複製されます。 変更は、永続サブスクリプションが使用されている限り、整合性が保たれます。

欠点

- JMSObjectGridEventListener の構成および保守は、複雑になる場合があります。eXtreme Scale は、eXtreme Scale デプロイメント記述子 XML ファイルまたは Spring などのその他のフレームワークを使用して、プログラマチックまたは宣言的に作成できます。
- スケーラブルではありません。データベースが必要とするメモリー量が、JVM の負担になる場合があります。
- Java 仮想マシンを追加する場合に不適切な機能:
  - データを簡単には区画化できない

– 無効化にコストがかかります。
– 各キャッシュは個別にウォームアップが必要になります。

使用する場合

このデプロイメント・トポロジーは、キャッシュに入れるデータ量が小さく (1 つの JVM に収まる)、かつ比較的安定している場合に限って使用するようにしてください。

分散キャッシュ

WebSphere eXtreme Scale は、共用キャッシュとして使用されることが最も多く、これまで使用されていたような従来のデータベースに代わり、データへのトランザクション・アクセスを複数のコンポーネントに提供します。これにより、データベースを構成する必要がなくなるため、アプリケーションの開発およびデプロイメントがより容易になります。

すべてのクライアントがキャッシュ内の同じデータを見るので、キャッシュはコヒーレントです。各データはキャッシュ内の 1 つのサーバーのみに保管されるため、さまざまなバージョンのデータを保管することになりかねない、レコードの無駄なコピーが防止されます。コヒーレントなキャッシュは、より多くのサーバーがグリッドに追加されるにつれて、より多くのデータを保持することができ、グリッドのサイズが増えるのにつれて直線的に増加します。クライアントはこのグリッドからのデータに、リモート・プロシージャー・コールを使用してアクセスするので、このキャッシュはリモート・キャッシュ (または、ファー・キャッシュ) とも呼ばれます。データの区画化により、各プロセスは、全データ・セットの中から固有のサブセットを保持します。グリッドが大きいほどより多くのデータを保持でき、そのデータに対するより多くの要求にサービスを提供できます。コヒーレントであることによって、失効データが存在しないため、グリッドの周囲で無効化データをプッシュする必要がなくなります。コヒーレント・キャッシュは、各データの最新コピーのみを保持します。

WebSphere Application Server 環境を実行している場合は、TranPropListener プラグインも使用可能です。TranPropListener プラグインは、WebSphere Application Server 高可用性コンポーネント (HA マネージャー) を使用して、変更を各ピア ObjectGrid キャッシュ・インスタンスに伝搬します。

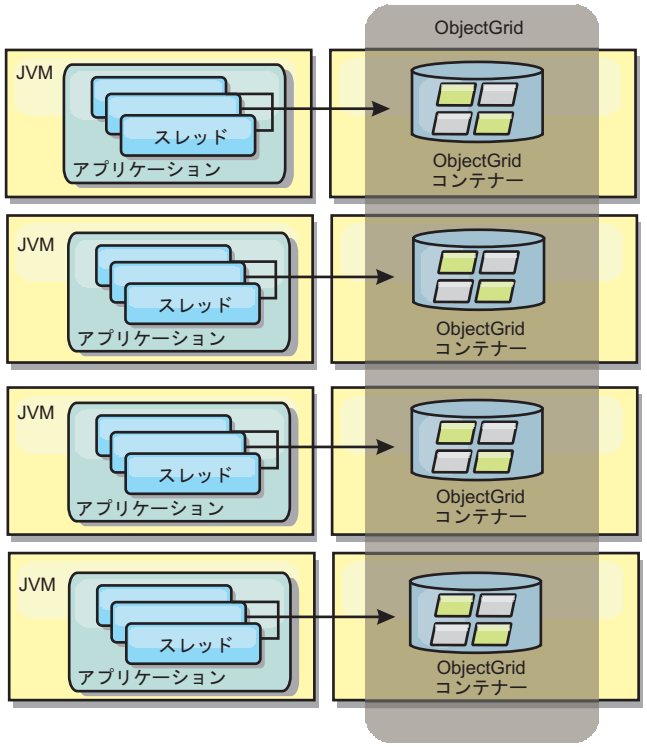


図 13. 分散キャッシュ

ニア・キャッシュ

クライアントは、eXtreme Scale が分散トポロジーで使用されている場合、オプションでローカルのインライン・キャッシュを持つことができます。オプションのこのキャッシュはニア・キャッシュと呼ばれます。これは、各クライアントにある独立した ObjectGrid であり、リモート用のキャッシュ (サーバー・サイド・キャッシュ) として機能します。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。

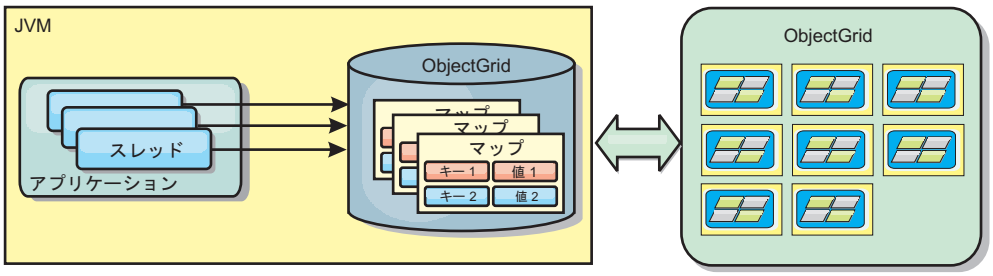


図 14. ニア・キャッシュ

ニア・キャッシュは、リモート側で eXtreme Scale サーバーに保管されているキャッシュ・データ・セット全体のサブセットへのメモリー内アクセスを可能にするため、非常に高速です。ニア・キャッシュは区画化されず、任意のリモート eXtreme Scale 区画からのデータを含みます。WebSphere eXtreme Scale は、以下のように、3 つまでのキャッシュ層を持つことができます。

1. トランザクション層キャッシュには、単一トランザクションのすべての変更が含まれます。トランザクション・キャッシュは、トランザクションがコミットされるまで、データの作業用コピーを保持します。クライアント・トランザクションが ObjectMap のデータを要求すると、最初にトランザクションがチェックされます。
2. クライアント層のニア・キャッシュは、サーバー層のデータのサブセットを保持します。トランザクション層にデータがない場合、データはニア・キャッシュにあればニア・キャッシュから取り出され、トランザクション・キャッシュに挿入されます。
3. サーバー層のグリッドには大半のデータが含まれ、すべてのクライアント間で共用されます。サーバー層は区画に分割できるので、大量のデータをキャッシュに入れることができます。クライアントのニア・キャッシュにデータが存在しないと、サーバー層からデータがフェッチされ、クライアント・キャッシュに挿入されます。サーバー層は、ローダー・プラグインを保持することもできます。グリッドに要求されたデータがない場合、Loader が呼び出され、結果のデータがバックエンドのデータ・ストアからグリッドに挿入されます。

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 eXtreme Scale 記述子構成で numberOfBuckets 属性を 0 に設定します。eXtreme Scale のロック・ストラテジーについて詳しくは、マップ・エントリーのロック (Map Entry Locking) を参照してください。ニア・キャッシュは、クライアント・オーバーライド eXtreme Scale 記述子構成を使用して、別の除去ポリシーや異なるプラグインを使用するように構成することもできます。

利点

- データへのアクセスがすべてローカルで行われるため、応答時間が速くなります。

欠点

- 失効したデータの期間が増大します。
- メモリー不足を回避するため、Evictor を使用してデータを無効化する必要があります。

使用する場合

応答時間が重要で、失効したデータは許容できる場合に使用します。

組み込みキャッシュ

eXtreme Scale グリッドは、組み込み eXtreme Scale サーバーとして既存のプロセス内で実行することも、外部プロセスとして管理することもできます。組み込みグリッドは、WebSphere Application Server などのアプリケーション・サーバー内で実行する場合に便利です。組み込まれていない eXtreme Scale サーバーは、コマンド行スクリプトを使用し、Java プロセスで実行することによって開始できます。

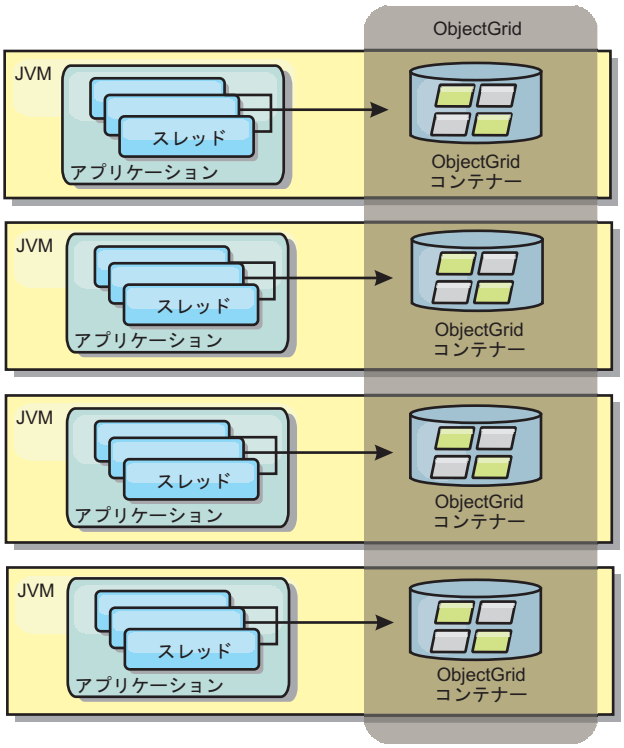


図 15. 組み込みキャッシュ

利点

- 管理するプロセスが減るため、管理が簡単になります。
- グリッドがクライアント・アプリケーションのクラス・ローダーを使用するため、アプリケーション・デプロイメントが容易になります。
- 区画化と高可用性をサポートします。

欠点

- すべてのデータがプロセス内に連結されるため、クライアント・プロセスのメモリー占有スペースが増えます。
- クライアント要求にサービスを提供するための CPU 使用率が高くなります。
- クライアントがサーバーと同じアプリケーション Java アーカイブ・ファイルを使用しているため、アプリケーション・アップグレードの処理がさらに難しくなります。
- 柔軟性が低くなります。クライアントとグリッド・サーバーは、同じレートで拡張することができません。サーバーを外部で定義すると、プロセス数の管理の柔軟性が増します。

使用する場合

組み込みグリッドは、クライアント・プロセスにグリッド・データおよび潜在的なフェイルオーバー・データ用の空きメモリーが豊富にある場合に使用します。

詳しくは、*管理ガイド*のクライアント無効化メカニズムの使用可能化に関するトピックを参照してください。

データベース統合

WebSphere eXtreme Scale が使用される目的は、従来のデータベースをその背後に置くことで、通常はデータベースにプッシュされる読み取りアクティビティーをなくすことです。コヒーレント・キャッシュは、オブジェクト関連マッパーを直接または間接に使用することにより、アプリケーションで使用できます。コヒーレント・キャッシュは、データベースまたは読み取りからの下流工程の負荷を軽減します。シナリオがもう少し複雑で、一部のデータのみが従来のパーシスタンス保証を必要とするデータ・セットへのトランザクション・アクセスなどの場合は、フィルター操作を使用して書き込みトランザクションの負荷を軽減します。

eXtreme Scale は、高度にフレキシブルなメモリー内のデータベース処理スペースとして機能するように構成できます。ただし、eXtreme Scale は、オブジェクト・リレーショナル・マッパー (ORM) ではありません。 eXtreme Scale は、それに含まれているデータがどこから取得されたのかを認識しません。アプリケーションまたは ORM は、データを eXtreme Scale サーバーに配置できます。データの発生元であるデータベースとの一貫性を保つのは、データのソースの責任です。これは、データベースから取り出されたデータを eXtreme Scale は自動的に無効化できないことを意味します。アプリケーションまたはマッパーは、この機能を提供して、eXtreme Scale に保管されているデータを管理する必要があります。

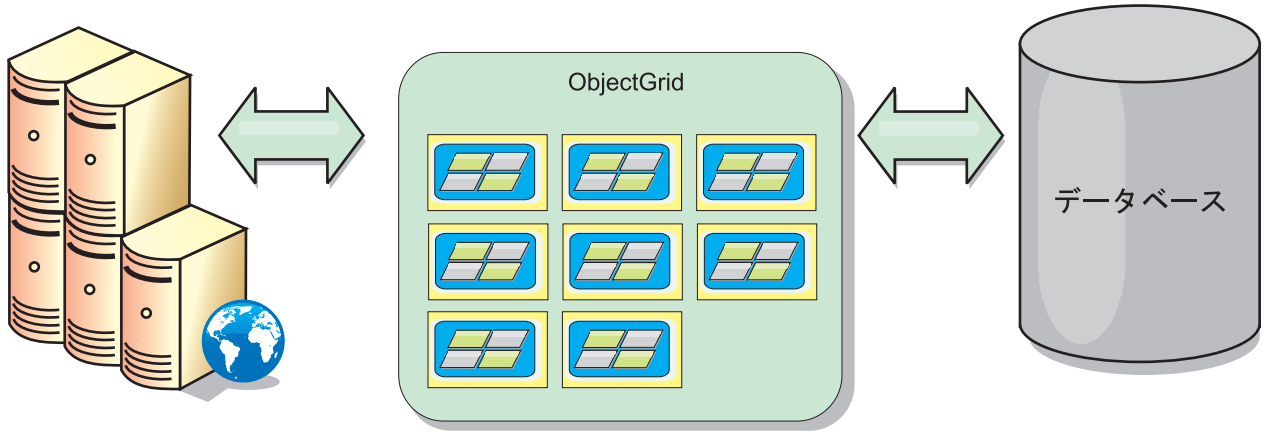


図 16. データベース・バッファーとしての ObjectGrid

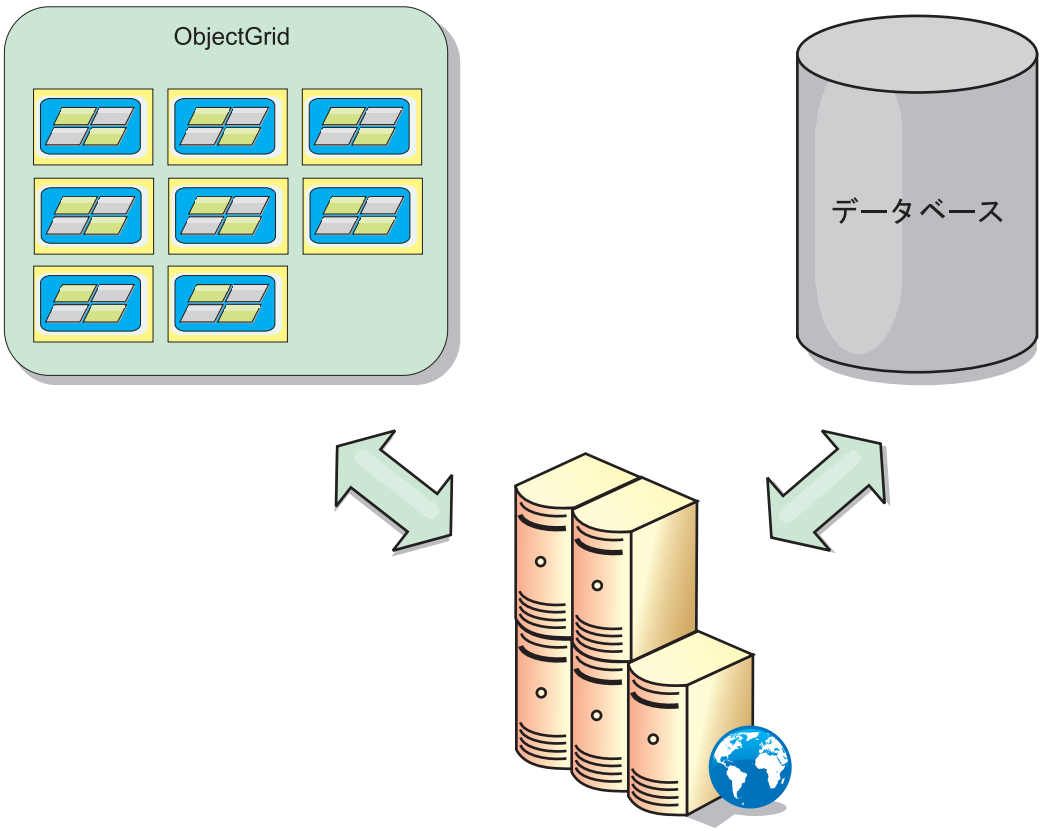


図 17. サイド・キャッシュとしての ObjectGrid

スパース・キャッシュおよび完全キャッシュ

WebSphere eXtreme Scale は、スパース・キャッシュまたは完全キャッシュとして使用できます。完全キャッシュがデータすべてを保持するのと違って、スパース・キャッシュはデータ全体のサブセットを保持し、要求時にデータをゆっくり取り込むことができます。通常、スパース・キャッシュは、データが部分的にしか使用可能でないため、キーを使用して (索引や照会を使用せず) アクセスされます。

キーが存在しない場合 (キャッシュ・ミスの場合)、次の層が呼び出され、データがフェッチされ、それぞれのキャッシュ層に挿入されます。照会または索引を使用する場合、現在ロードされている値のみがアクセスされ、要求は他の層に転送されません。 完全キャッシュには必要なすべてのデータが含まれ、索引または照会により非キー属性を使用してアクセスできます。

完全キャッシュには、アプリケーションが使用する前にデータがプリロードされ、データベースの代用として効率的に機能します。完全キャッシュは、データがロードされた後は、データベースと同様に扱うことができます。すべてのデータがあるので、照会および索引を使用して、データの検出と集約を行うことができます。

サイド・キャッシュとインライン・キャッシュ

WebSphere eXtreme Scale は、データベース・バックエンドにインライン・キャッシングを提供するために使用されるか、データベースのサイド・キャッシュとして使

用されます。インライン・キャッシングは、データと対話するための基本手段として eXtreme Scale を使用します。eXtreme Scale をサイド・キャッシュとして使用する場合は、eXtreme Scale と連動してバックエンドが使用されます。

サイド・キャッシュ

eXtreme Scale は、アプリケーションのデータ・アクセス層のサイド・キャッシュとして使用できます。このシナリオの場合、eXtreme Scale は、通常であればバックエンド・データベースから取得されるオブジェクトを一時的に保管するために使用されます。アプリケーションは、必要なデータが eXtreme Scale に含まれているかどうかチェックします。そこに必要なデータがあった場合、そのデータが呼び出し元に返されます。そこに必要なデータがない場合、データがバックエンドから取得され、次の要求がキャッシュ・コピーを使用できるように、データは eXtreme Scale に挿入されます。次の図は、OpenJPA や Hibernate といった任意のデータ・アクセス層を使用しながら eXtreme Scale をサイド・キャッシュとして使用する方法を示しています。

Hibernate および OpenJPA 向けサイド・キャッシュ・プラグイン

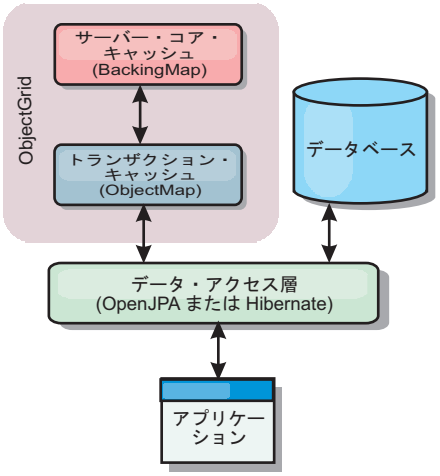


図 18. サイド・キャッシュ

eXtreme Scale には、eXtreme Scale を自動サイド・キャッシュとして使用できるようにする、 OpenJPA および Hibernate の両方に使用できるキャッシュ・プラグインが組み込まれています。eXtreme Scale をキャッシュ・プロバイダーとして使用すると、データの読み取りおよび照会時のパフォーマンスが高まり、データベースへの負荷が軽減されます。eXtreme Scale ではキャッシュが自動的にすべてのプロセス間で複製されるので、組み込みキャッシュ実装をしのぐ利点があります。あるクライアントが、値をキャッシュに入れると、他のすべてのクライアントが、そのキャッシュされた値を使用できるようになります。

インライン・キャッシュ

インライン・キャッシュとして使用される場合、eXtreme Scale はローダー・プラグインを使用してバックエンドと対話します。このシナリオでは、アプリケーションが直接 eXtreme Scale API にアクセスできるようになるため、データ・アクセスが単純化されます。キャッシュ内のデータとバックエンドのデータが確実に同期され

るようにするための数種類のキャッシング・シナリオが、eXtreme Scale においてサポートされています。次の図は、インライン・キャッシュがアプリケーションおよびバックエンドと対話する方法を示しています。

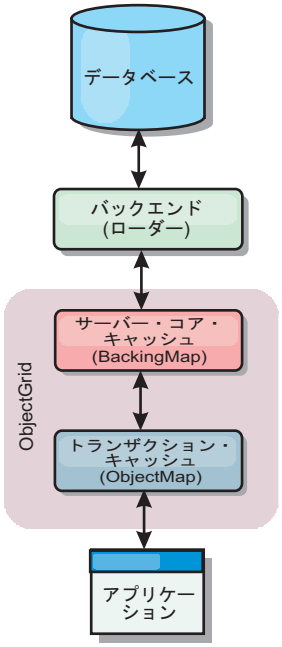


図 19. インライン・キャッシュ

データベースの同期手法

WebSphere eXtreme Scale をキャッシュとして使用する際、データベースを eXtreme Scale トランザクションとは独立して更新できる場合、失効データを許容するようにアプリケーションを作成する必要があります。同期されたメモリー内データベース処理スペースとして機能するため、eXtreme Scale はキャッシュを常に最新の状態に保つ方法をいくつか備えています。

データベースの同期手法

定期的リフレッシュ

時間ベースの Java Persistence API (JPA) データベース・アップデーターを使用して、定期的なキャッシュの無効化または更新を自動的に実行できます。このアップデーターは、JPA プロバイダーを使用してデータベースを定期的に照会することによって、前回の更新以降に発生した更新または挿入があるかどうかを調べます。示された変更は、スパース・キャッシュで使用された場合、自動的に無効にされるか、更新されます。完全キャッシュで使用された場合、エントリーをディスカバーして、キャッシュに挿入することができます。エントリーがキャッシュから除去されることはありません。

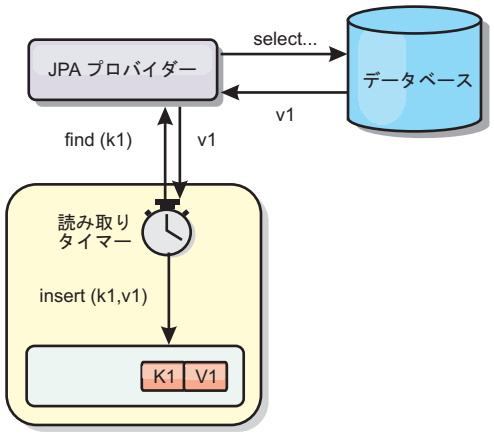


図 20. 定期的リフレッシュ

除去

スパース・キャッシュでは、除去ポリシーを使用して、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ます。eXtreme Scale には、Time-To-Live (存続時間)、Least-Recently-Used (最長未使用時間)、および Least-Frequently-Used (最も使用頻度の少ない) という 3 つの組み込みポリシーがあります。メモリー・ベースの除去オプションを使用可能にすると、メモリーが制約状態になるので、3 つのポリシーではすべて、必要であればデータをより積極的に除去することができます。詳しくは、27 ページの『除去』のトピックを参照してください。

イベント・ベースの無効化

スパース・キャッシュおよび完全キャッシュは、Java Message Service (JMS) などのイベント・ジェネレーターを使用して無効化または更新することができます。JMS を使用した無効化は、データベース・トリガーを使用してバックエンドを更新するどのプロセスにも手動で関連付けることができます。サーバー・キャッシュで変更があった場合にクライアントに通知できる JMS ObjectGridEventListener プラグインが eXtreme Scale で提供されています。これにより、クライアントが失効データを表示する時間を短縮できます。

プログラマチックな無効化

eXtreme Scale API により、 Session.beginNoWriteThrough()、ObjectMap.invalidate()、および EntityManager.invalidate() API メソッドを使用したニア・キャッシュおよびサーバー・キャッシュの手動対話が可能になります。クライアントまたはサーバーのプロセスでデータの一部がもう必要ない場合、無効化メソッドを使用して、ニア・キャッシュまたはサーバー・キャッシュからデータを除去できます。beginNoWriteThrough メソッドは、ローダーを呼び出すことなく、ObjectMap または EntityManager 操作をローカル・キャッシュに適用します。クライアントから呼び出された場合のこの操作は、ニア・キャッシュのみに適用されます (リモート・ローダーは呼び出されません)。サーバーで呼び出された場合のこの操作は、ローダーを呼び出すことなく、サーバー・コア・キャッシュのみに適用されます。

定期的リフレッシュ

WebSphere eXtreme Scale のメモリー内データベース処理スペース能力がキャッシュとして使用されたとき、eXtreme Scale トランザクションとは無関係にデータベースを更新できる場合は、失効データを許容するようにアプリケーションを作成する必要があります。定期的リフレッシュを使用することは、eXtreme Scale がキャッシュを常に最新の状態に保てるようにする方法の 1 つです。

時間ベースの Java Persistence API (JPA) データベース・アップデーターを使用して、定期的なキャッシュの無効化または更新を自動的に実行できます。このアップデーターは、JPA プロバイダーを使用してデータベースを定期的に照会することによって、前回の更新以降に発生した更新または挿入があるかどうかを調べます。示された変更は、スパース・キャッシュで使用された場合、自動的に無効にされるか、更新されます。完全キャッシュで使用された場合、エントリーをディスカバーして、キャッシュに挿入することができます。エントリーがキャッシュから除去されることはありません。

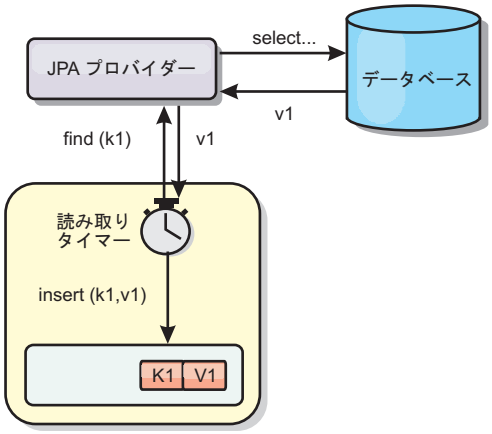


図 21. 定期的リフレッシュ

スパース・キャッシュでは、除去ポリシーを使用して、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ます。eXtreme Scale には、Time-To-Live (存続時間)、Least-Recently-Used (最長未使用時間)、および Least-Frequently-Used (最少使用頻度) という 3 つのポリシーが組み込まれています。メモリー・ベースの除去オプションを使用可能にすると、メモリーが制約状態になるので、3 つのポリシーではすべて、必要であればデータをより積極的に除去することができます。エビクターについて詳しくは、『除去』を参照してください。

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。

インライン・キャッシングのシナリオ

インライン・キャッシングは、データと対話するための基本手段として eXtreme Scale を使用します。eXtreme Scale がインライン・キャッシュとして使用される場合、アプリケーションは、ローダー・プラグインを使用してバックエンドと対話します。

インライン・キャッシング・オプションにより、アプリケーションが eXtreme Scale API に直接アクセスできるようになるため、データ・アクセスが単純化されます。WebSphere eXtreme Scale は、以下のような複数のインライン・キャッシング・シナリオをサポートします。

- リードスルー
- ライトスルー
- 後書き

リードスルー・キャッシングのシナリオ

リードスルー・キャッシュは、データ・エントリーの要求時にキーによるそのロードが暫時的に行われるスパース・キャッシュです。これが行われる場合、呼び出し元は、エントリーがどのように取り込まれるかを知る必要はありません。データが eXtreme Scale キャッシュに見つからない場合、eXtreme Scale は、その欠落データをローダー・プラグインから取得します。このプラグインは、バックエンド・データベースからデータをロードして、そのデータをキャッシュに挿入します。同じデータ・キーに対する後続の要求は、削除、無効化、または除去されるまでキャッシュに存在します。

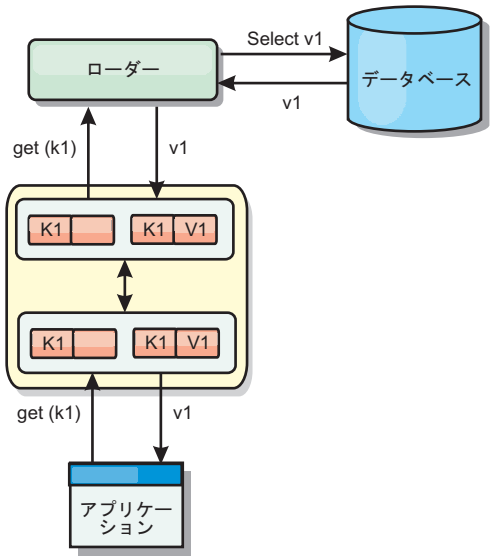


図 22. リードスルー・キャッシング

ライトスルー・キャッシングのシナリオ

ライトスルー・キャッシュでは、キャッシュへの書き込みが行われるたびに、ローダーを使用してデータベースへの書き込みが同期的に行われます。このメソッドでは、バックエンドとの整合性はありますが、データベース操作が同期されるため、書き込みパフォーマンスは低下します。キャッシュとデータベースがともに更新されるため、同じデータに対する後続の読み取りはキャッシュに残り、データベース呼び出しが回避されます。ライトスルー・キャッシュは、多くの場合、リードスルー・キャッシュと一緒に使用されます。

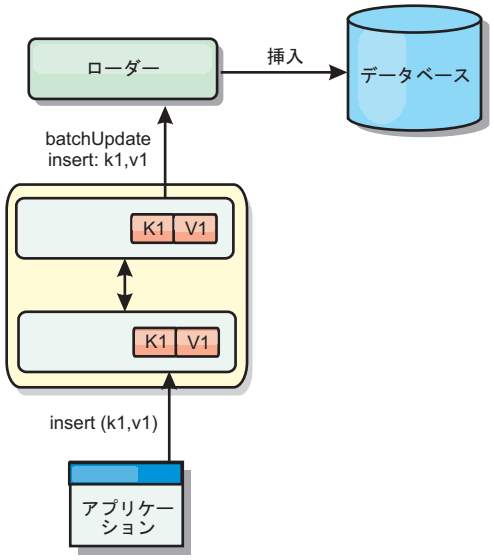


図 23. ライトスルー・キャッシング

後書きキャッシングのシナリオ

変更を非同期的に書き込むことにより、データベースの同期性が改善されます。後書きキャッシュまたはライト・バック・キャッシュとも呼ばれます。通常はローダーに対して同期的に書き込まれる変更は、eXtreme Scale 内でバッファー化されてから、バックグラウンド・スレッドを使用してデータベースに書き込まれます。データベース操作をクライアント・トランザクションから除去し、データベース書き込みを圧縮できるため、書き込みパフォーマンスが著しく向上します。詳しくは、 34 ページの『後書きキャッシング』を参照してください。

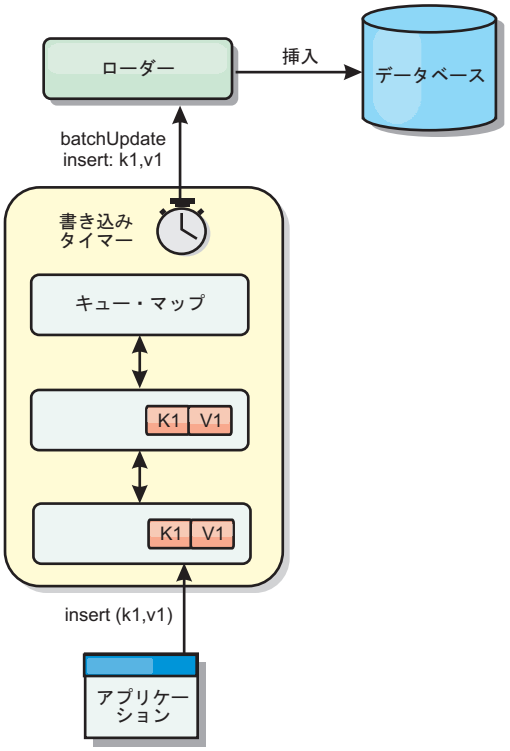


図 24. 後書きキャッシング

詳しくは、 34 ページの『後書きキャッシング』を参照してください。

ローダー

ローダーは、BackingMap とバックエンド (データベースなど) との間のリンクとして機能するプラグインです。

ローダーは、キーに関する要求をキャッシュが満足できなくなったときに起動され、リードスルー機能や、キャッシュにデータをゆっくり設定する機能を提供します。また、ローダーによって、キャッシュ値が変わったときのデータベース更新が可能になります。1 つのトランザクション内のすべての変更は、データベースとの対話の数を最小化できるよう、まとめてグループ化されます。ローダーと共に TransactionCallback プラグインが、バックエンド・トランザクションの境界をトリガーするために使用されます。このプラグインの使用は、複数のマップが 1 つのトランザクションに含まれている場合、または、トランザクション・データがコミットなしでキャッシュに書き込まれる場合に重要です。

ローダーは、データベース・ロックの保持を回避するために、資格過剰の更新を使用することもできます。バージョン属性をキャッシュ値の中に入れることによって、値がキャッシュ内で更新されるときにローダーは値の前と後のイメージを見ることができます。その後、データベースまたはバックエンドを更新する際にこの値を使用して、データが更新されていないことを検証できます。 ローダーは、開始時にグリッドをプリロードするよう構成することもできます。区画に分割されている場合、各区画ごとに 1 つのローダー・インスタンスが関連付けられます。例えば、「Company」マップに 10 個の区画がある場合、プライマリー区画ごとに 1 つずつ、10 個のローダー・インスタンスがあります。このマップのプライマリー断片がアクティブにされると、ローダーに対して preloadMap メソッドが同期または非同期で呼び出され、マップ区画にバックエンドからのデータが自動的にロードされます。非同期で呼び出される場合、すべてのクライアント・トランザクションはブロックされ、グリッドへの矛盾するアクセスを防止します。代わりに、クライアント・プリローダーを使用してグリッド全体にデータをロードできます。

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。

後書きキャッシング

後書きキャッシングを使用して、バックエンド・データベースを更新する際に発生するオーバーヘッドを減らすことができます。

概要

後書きキャッシングは、ローダー・プラグインへの更新情報を非同期でキューに入れます。eXtreme Scale トランザクションをデータベース・トランザクションから分離することにより、マップの更新、挿入、および除去の、パフォーマンスを改善できます。非同期更新は、時間ベースの遅延 (例えば、5 分間)、またはエントリー・ベースの遅延 (例えば、1000 エントリー) で実行されます。

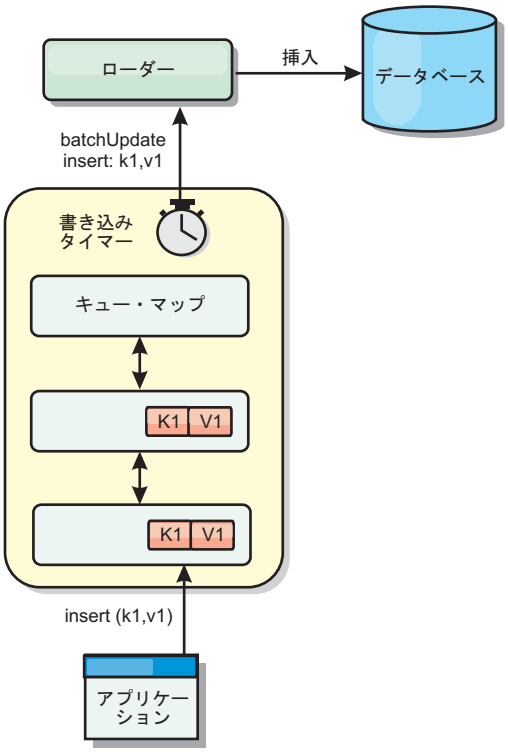


図 25. 後書きキャッシング

BackingMap の後書き構成により、ローダーとマップとの間にスレッドが作成されます。次に、ローダーは、BackingMap.setWriteBehind メソッド内の構成設定に従って、そのスレッドを通してデータ要求を委任します。eXtreme Scale トランザクションが、マップのエントリーを挿入、更新、または削除すると、これらの各レコードごとに 1 つずつ LogElement オブジェクトが作成されます。これらのエレメントは後書きローダーに送信され、キュー・マップと呼ばれる特別な ObjectMap 内でキューに入れられます。後書き設定が有効になっているバックアップ・マップは、それぞれ独自のキュー・マップを持っています。後書きスレッドは、キューに入れられたデータを定期的にキュー・マップから除去し、それらのデータを実際のバックエンド・ローダーにプッシュします。

後書きローダーは、挿入、更新、および削除タイプの LogElement オブジェクトのみを実際のローダーに送信します。それ以外のタイプの LogElement オブジェクト (例えば、EVICT タイプ) はすべて無視されます。

利点

後書きサポートを使用可能にすることによる利点は以下のとおりです。

- **バックエンド障害の分離:** 後書きキャッシングは、バックエンド障害からの分離層を提供します。バックエンドのデータベースで障害が発生すると、更新はキュー・マップ内でキューに入れられます。アプリケーションは、トランザクションを eXtreme Scale に送り続けることができます。バックエンドが復旧すると、キュー・マップ内のデータはバックエンドにプッシュされます。

- **バックエンドの負荷の削減:** 後書きローダーは更新をキー単位でマージします。その結果、キュー・マップ内には、キーごとにマージされた更新が 1 つのみ存在します。このマージにより、バックエンド・データベースに対する更新の数が減ります。
- **トランザクション・パフォーマンスの向上:** データがバックエンドと同期するのをトランザクションが待つ必要がないため、個々の eXtreme Scale トランザクションの時間が減ります。

アプリケーション設計に関する考慮事項

後書きサポートを使用可能にするのは簡単ですが、後書きサポートを処理するアプリケーションの設計は十分に考慮する必要があります。後書きサポートがない場合、ObjectGrid トランザクションにバックエンド・トランザクションが包含されます。 ObjectGrid トランザクションはバックエンド・トランザクションの開始前に開始し、バックエンド・トランザクションの終了後に終了します。

後書きサポートが有効な場合、ObjectGrid トランザクションは、バックエンド・トランザクションが開始する前に終了します。ObjectGrid トランザクションとバックエンド・トランザクションは切り離されます。

参照保全性の制約

後書きサポートで構成されているそれぞれのバックアップ・マップは、データをバックエンドにプッシュするための独自の後書きスレッドを持ちます。したがって、1 つの ObjectGrid トランザクションにさまざまなマップを更新するデータが含まれていても、バックエンドでは、それぞれ異なるバックエンド・トランザクションでデータの更新が行われます。例えば、トランザクション T1 はマップ Map1 のキー key1 とマップ Map2 のキー key2 を更新するとします。マップ Map1 に対する key1 の更新は、あるバックエンド・トランザクションの中でバックエンドに対して行われ、マップ Map2 に対する key2 の更新は、異なる後書きスレッドを使用して、別のバックエンド・トランザクションの中でバックエンドに対して実行されます。Map1 に保管されたデータと Map2 に保管されたデータがバックエンドでの外部キー制約などの関係を持つ場合、更新が失敗する可能性があります。

バックエンド・データベースの参照保全性制約を設計するときは、順不同の更新に必ず対応できるようにしてください。

キュー・マップのロックの振る舞い

トランザクションの動作で他に大きく異なる点は、ロックの振る舞いです。ObjectGrid は、PESSIMISTIC、OPTIMISITIC、および NONE の 3 つの異なるロック・ストラテジーをサポートします。後書きキュー・マップは、バックアップ・マップに構成されているロック・ストラテジーに関係なく、ペシミスティック・ロック・ストラテジーを使用します。キュー・マップのロックを獲得する操作には 2 つの異なるタイプがあります。

- ObjectGrid トランザクションのコミット時、またはフラッシュ (マップ・フラッシュまたはセッション・フラッシュ) の発生時、トランザクションはキュー・マップ内のキーを読み取り、キーに S ロックをかけます。
- ObjectGrid トランザクションのコミット時、トランザクションは、キーの S ロックを X ロックにアップグレードしようとします。

キュー・マップのこの余分な動作のため、ロックの動作に少々違いがあります。

- ユーザー・マップが ペシミスティック・ロック・ストラテジーで構成されている場合、ロックの動作にほとんど違いはありません。フラッシュまたはコミットが呼び出されるたび、キュー・マップ内の同じキーに S ロックがかけられます。コミット時間中、ユーザー・マップ内のキーに X ロックが獲得されるだけでなく、キュー・マップ内のキーに対しても X ロックが獲得されます。
- ユーザー・マップが OPTIMISTIC または NONE ロック・ストラテジーで構成されている場合、ユーザー・トランザクションは PESSIMISTIC ロック・ストラテジーのパターンに従います。フラッシュまたはコミットが呼び出されるたび、キュー・マップ内の同じキーの S ロックが獲得されます。コミット時は、同じトランザクションを使用して、キュー・マップ内のキーの X ロックが獲得されます。

ローダー・トランザクションの再試行

ObjectGrid は、2 フェーズ・トランザクションまたは XA トランザクションをサポートしません。後書きスレッドは、キュー・マップからレコードを除去して、バックエンドに対してそのレコードを更新します。トランザクションの最中にサーバーに障害が起こると、一部のバックエンドの更新が失われる可能性があります。

後書きローダーは、失敗したトランザクションの書き込みを自動的に再試行し、データ損失を防ぐために未確定 LogSequence をバックエンドに送信します。このアクションを行うには、ローダーがべき等である必要があります。この意味は、Loader.batchUpdate(TxId, LogSequence) が同じ値で 2 回呼び出されたとき、それは適用された回数があたかも 1 回だったかのように、同じ結果を返すということです。ローダー実装は、この機能を使用可能にするため、RetryableLoader インターフェースを実装しなければなりません。詳しくは、API 資料を参照してください。

ローダーの障害

ローダー・プラグインは、バックエンド・データベースと通信できない場合、失敗することがあります。データベース・サーバーまたはネットワーク接続がダウンすると、このような状態が発生します。後書きローダーは、更新をキューに入れ、データ変更を定期的にローダーにプッシュしようと試みます。ローダーは、LoaderNotAvailableException 例外をスローして、データベース接続の問題があることを ObjectGrid ランタイムに通知しなければなりません。

したがって、ローダー実装で、データ障害または物理的ローダー障害を識別できるようになっている必要があります。データ障害は LoaderException または OptimisticCollisionException としてスローまたは再スローされる必要がありますが、物理的なローダーの障害は LoaderNotAvailableException としてスローまたは再スローされる必要があります。 ObjectGrid は、これら 2 つの例外を異なる方法で処理します。

- 後書きローダーが LoaderException を catch した場合、後書きローダーは、その障害の原因を重複キー障害など、何らかのデータ障害と見なします。後書きローダーは、更新のバッチ処理を解除し、データ障害を分離するため、1 度に 1 レコードずつ更新しようとします。1 レコードの更新時に再度 {{LoaderException}} がキャッチされると、失敗した更新レコードが作成され、失敗した更新マップのログに記録されます。

- 後書きローダーが LoaderNotAvailableException を catch した場合、後書きローダーは、データベースに接続できないことが、その障害の原因だと判断します (例えば、バックエンド・データベースがダウンしている、データベース接続が使用不可である、ネットワークがダウンしているなど)。後書きローダーは 15 秒待機した後、データベースへのバッチ更新を再試行します。

一般的な間違いは、LoaderNotAvailableException がスローされるべきなのに、LoaderException がスローされることです。そうすると、後書きローダーのキューに入れられたすべてのレコードが失敗した更新レコードになり、バックエンドの障害分離という目的にそぐわなくなります。

パフォーマンスの考慮事項

後書きキャッシング・サポートの場合、ローダー更新をトランザクションから除去することで、応答時間が増加します。また、データベース更新が結合されるため、データベース・スループットも増えます。データをキュー・マップからプルし、ローダーにプッシュされる後書きスレッドの導入によって生じるオーバーヘッドを理解しておく必要があります。

最大更新カウントまたは最大更新時間は、予期される使用パターンおよび環境に基づいて調整する必要があります。最大更新カウントまたは最大更新時間の値が小さすぎると、後書きスレッドのオーバーヘッドが、その利点を帳消しにするおそれがあります。これら 2 つのパラメーターに大きな値を設定する場合も、データのキューイングに必要なメモリー使用が増え、データベース・レコードが不整合になる時間が増加するおそれがあります。

最善のパフォーマンスを得るために、後書き関係のパラメーターは、以下の要因を考慮に入れて調整してください。

- 読み取りトランザクションと書き込みトランザクションの比率
- 同一レコード更新の頻度
- データベース更新の待ち時間

データのプリロードおよびウォームアップ

多くのシナリオで、グリッドにデータをプリロードしておくと便利です。

グリッドは、完全キャッシュとして使用される場合、データのすべてを保持しなければならず、いずれかのクライアントが接続する前にデータがロードされている必要があります。スパース・キャッシュとして使用する場合は、クライアントが接続時にデータにすぐにアクセスできるよう、キャッシュをデータでウォームアップしておくべきです。

以下のセクションで説明するように、データをグリッドにプリロードする方法は 2 つあります。1 つはローダー・プラグインを使用する方法で、もう 1 つはクライアント・ローダーを使用する方法です。

ローダー・プラグイン

ローダー・プラグインは、各マップに関連付けられ、1 つのプライマリー区画断片をデータベースと同期化させる役割を担います。断片がアクティブになると、ローダー・プラグインの preloadMap メソッドが自動的に呼び出されます。 したがっ

て、100 の区画がある場合、ローダーのインスタンスは 100 存在し、それぞれが、各自の区画のためにデータをロードします。同期的に実行された場合、プリロードが完了するまですべてのクライアントがブロックされます。

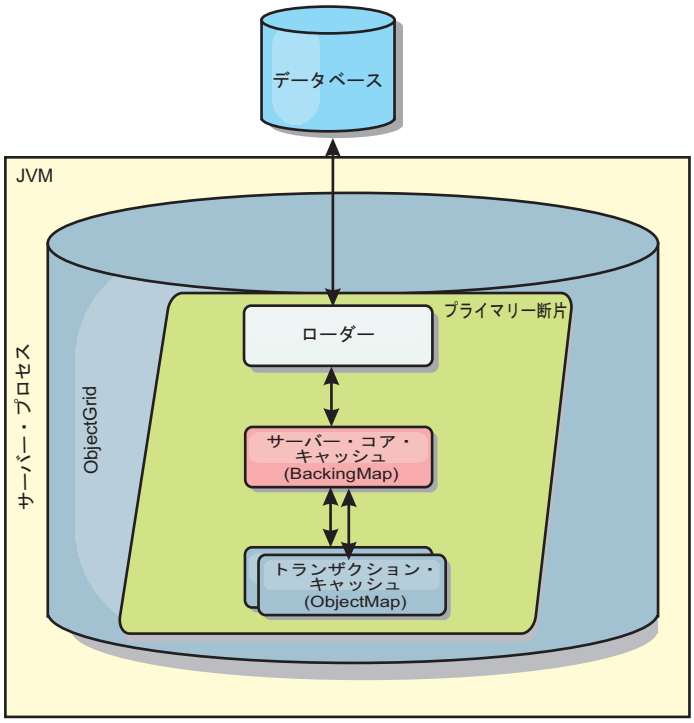


図 26. ローダー・プラグイン

クライアント・ローダー

クライアント・ローダーは、1 つ以上のクライアントを使用してグリッドにデータをロードするパターンです。複数のクライアントを使用してグリッドにデータをロードすることは、区画スキーマがデータベースに保管されない場合は効率的です。クライアント・ローダーは手動で呼び出すか、グリッドの開始時に自動的に呼び出すことができます。グリッドにデータをプリロードしている間はクライアントがグリッドにアクセスできないように、クライアント・ローダーは、オプションで、StateManager を使用してグリッドの状態をプリロード・モードに設定できます。WebSphere eXtreme Scale には Java Persistence API (JPA) ベースのローダーが組み込まれていて、OpenJPA または Hibernate JPA プロバイダーのどちらかでグリッドに自動的にロードするために使用できます。

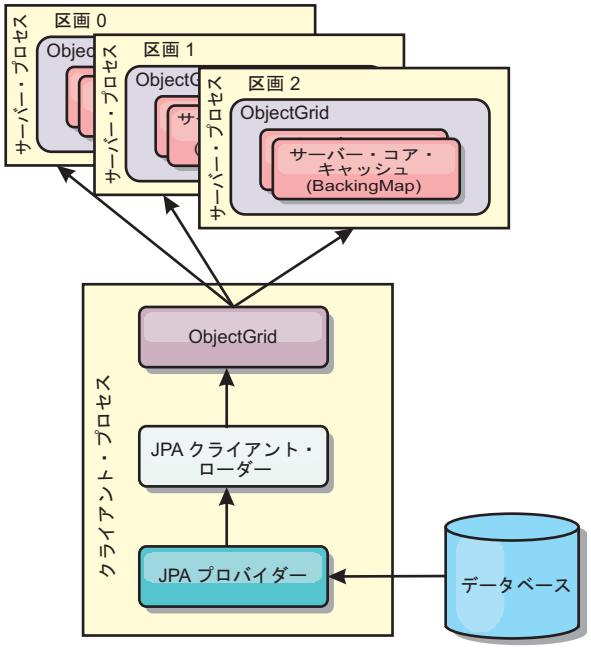


図 27. クライアント・ローダー

Java オブジェクト・キャッシュ概念

WebSphere eXtreme Scale は、主として Java オブジェクト用のデータ・グリッドおよびキャッシュとして使用されます。 これらのオブジェクトにアクセスし、これらを保管するために eXtreme Scale グリッドと対話するのに使用できる、いくつかの API があります。

このトピックでは、一般的な API のうちのいくつかについて説明し、API およびデプロイメントのトポロジーを選択する際に認識しておく必要があるいくつかの概念についても説明します。 eXtreme Scale が提供するさまざまなサービスおよびトポロジーに関する説明については、9 ページの『アーキテクチャーおよびトポロジー』トピックを参照してください。

WebSphere eXtreme Scale の中心的なコンポーネントは ObjectGrid です。ObjectGrid は、関連するデータを保管する名前空間であり、ハッシュ・マップの集合を含んでいます。各マップにはキーと値のペアが保持されます。これらのマップは、グループ化して、区画に分割することができ、可用性の高い、スケーラブルなものにできます。

グリッドはその本質上 Java オブジェクトを保持するので、アプリケーションを設計するときには、グリッドがデータを効率的に保管しアクセスできるようにするための重要な考慮事項がいくつかあります。 スケーラビリティー、パフォーマンス、およびメモリー使用効率に影響を与える可能性のある要因を以下に記述します。

クラス・ローダーおよびクラスパスの考慮事項

eXtreme Scale は Java オブジェクトをデフォルトではキャッシュに保管するので、データがアクセスされる場所では、必ずクラスパスにクラス定義が存在していなければなりません。

具体的には、eXtreme Scale クライアントおよびコンテナー・プロセスは、プロセス開始時に、クラスパスにクラスまたは jar を組み込む必要があります。 eXtreme Scale と共に使用するアプリケーションを設計する際、ビジネス・ロジックと永続データ・オブジェクトは分けて考えてください。

詳しくは、WebSphere Application Server Network Deployment インフォメーション・センターでクラス・ロードを参照してください。

Spring Framework 設定内での考慮事項については、*プログラミング・ガイド*の Spring Framework との統合に関するトピックのパッケージ化セクションを参照してください。

WebSphere eXtreme Scale インスツルメンテーション・エージェントの使用に関連した設定については、*プログラミング・ガイド*のインスツルメンテーション・エージェントのトピックを参照してください。

リレーションシップ管理

Java などのオブジェクト指向言語、およびリレーショナル・データベースは、リレーションシップまたは関連をサポートします。リレーションシップは、オブジェクト参照または外部キーの使用を通してストレージ量を削減します。

グリッド内でリレーションシップを使用するときには、データはコンストレインド・ツリーに編成されている必要があります。そのツリーには 1 つのルート・タイプがなければならず、すべての子は 1 つのルートのみに関連付けられていなければなりません。例: 部門には多数の従業員が属し、1 人の従業員が多くのプロジェクトを持つことができます。しかし、1 つのプロジェクトが異なる部門に属している多くの従業員を持つことはできません。ルートが定義されると、ルート・オブジェクトとその子孫へのすべてのアクセスはルートを通して管理されるようになります。WebSphere eXtreme Scale は、ルート・オブジェクトのキーのハッシュ・コードを使用して区画を選択します。

例: partition = (hashCode MOD numPartitions)

1 つのリレーションシップに関係するすべてのデータが単一のオブジェクト・インスタンスに結びついている場合、ツリー全体を 1 つの区画内に配列し、1 つのトランザクションを使用して非常に効率的にアクセスすることができます。データが複数のリレーションシップにまたがっている場合は、複数の区画についての処理が必要になるため、追加のリモート呼び出しが必要になり、結果的にパフォーマンス上のボトルネックとなることがあります。

参照データ

一部のリレーションシップは、ルックアップまたは参照データを含んでいます。例: CountryName。これは、データがどの区画にも存在しているという特殊なケースです。このような場合は、データはどのルート・キーでもアクセスでき、同じ結果が

戻ります。 このような参照データは、すべての区画での更新が必要でコストがかかるため、データが相当に静的なケースに限って使用するべきです。参照データを最新に保つ手法として、DataGrid API がよく使われます。

正規化のコストと利点

リレーションシップを使用してデータを正規化することは、データの重複が減るため、グリッドによるメモリー使用量を削減するのに寄与します。ただし、一般的には、追加される関係データが多いほど、スケールアウトは少なくなります。データがグループ化されている場合、リレーションシップを維持し、管理できる程度のサイズに保つために、より多くのコストがかかるようになります。グリッドは、ツリーのルートのキーに基づいてデータを区画に分けるので、ツリーのサイズは考慮には入れられません。したがって、1 つのツリー・インスタンスに対して多数のリレーションシップがある場合、グリッドは不平衡になり、結果的に 1 つの区画に他の区画よりも多くのデータが入ってしまうことが起こりえます。

データが非正規化またはフラット化されている場合、通常であれば 2 つのオブジェクト間で共有されるデータが、そうされずに複写され、各表は別々に区画化されるので、より平衡したグリッドになります。 これは使用されるメモリー量を増やしますが、必要なデータがすべて入っている単一のデータ行にアクセスできるため、アプリケーションはスケーラブルになります。データ保守にかかるコストは最近ますます高くなっているため、読み取り主体のグリッドにはこれは理想的です。

詳しくは、 XTP システムの分類およびスケーリングを参照してください。

データ・アクセス API を使用したリレーションシップ管理

ObjectMap API は、最も高速かつ柔軟で粒度の細かいデータ・アクセス API であり、マップのグリッド内のデータにアクセスする手段として、トランザクションを使用する、セッション・ベースの方法を提供します。ObjectMap API によって、クライアントは、標準 CRUD (作成、読み取り、更新、および削除) 操作を使用して分散グリッド内のオブジェクトのキーと値のペアを管理することができます。

ObjectMap API を使用するときには、オブジェクトのリレーションシップが、すべてのリレーションシップの外部キーを親オブジェクトに埋め込むことによって表される必要があります。

以下に例を示します。

```
public class Department {
 Collection<String> employeeIds;
}
```

EntityManager API は、外部キーを含んでいるオブジェクトから永続データを抽出することにより、リレーションシップ管理を単純にします。以下の例に示すように、オブジェクトが後でグリッドから取り出されるとき、リレーションシップ・グラフは再構築されます。

```
@Entity
public class Department {
 Collection<String> employees;
}
```

EntityManager API は、JPA や Hibernate といった他の Java オブジェクト・パーシスタンス・テクノロジー (管理対象 Java オブジェクト・インスタンスのグラフはパ

ーシスタント・ストアと同期化されます) にとてもよく似ています。このケースでは、パーシスタント・ストアは eXtreme Scale グリッドであり、そこでは、各エンティティーがマップとして表され、マップはオブジェクト・インスタンスではなくエンティティー・データを含みます。

キャッシュ・キーに関する考慮事項

WebSphere eXtreme Scale は、ハッシュ・マップを使用してデータをグリッドに保管します。グリッドではキーに Java オブジェクトが使用されます。

指針

キーを選択するときには、以下の要件を考慮してください。

- キーは、決して変更できません。キーの一部分を変更する必要がある場合、キャッシュ・エントリーをいったん削除してから再挿入する必要があります。
- キーは小さくしてください。キーはすべてのデータ・アクセス操作で使用されるので、キーを小さくして、シリアライズが効率的に行われるようにし、使用されるメモリーを少なくするのが望ましい方法です。
- 優れたハッシュおよび同値アルゴリズムを実装してください。hashCode() メソッドと equals(Object o) メソッドは、各キー・オブジェクトごとに常にオーバーライドされる必要があります。
- キーのハッシュ・コードをキャッシュに入れてください。可能であれば、hashCode() 計算を速くするため、キー・オブジェクト・インスタンス内のハッシュ・コードをキャッシュに入れてください。キーは不変なので、ハッシュ・コードはキャッシュに入れることができるはずです。
- キーを値に複写することを避けてください。ObjectMap API を使用している場合、値オブジェクトの中にキーを保管すると便利です。そうした場合、キー・データがメモリー内で重複します。

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。

第 3 章 キャッシュ統合

WebSphere eXtreme Scale を他のキャッシュ関連製品と統合することができます。WebSphere eXtreme Scale とデータベースとの間で、変更を統合するためのローダーとして JPA を使用することができます。また、WebSphere eXtreme Scale 動的キャッシュ・プロバイダーを使用して、WebSphere eXtreme Scale を WebSphere Application Server 内の動的キャッシュ・コンポーネントに接続することもできます。 WebSphere Application Server に対する拡張としてもう 1 つ考えられるのは、HTTP セッションをキャッシュに入れる操作を支援する WebSphere eXtreme Scale HTTP セッション・マネージャーです。

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。

JPA ローダーの概要

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。

JPALoader の com.ibm.websphere.objectgrid.jpa.JPALoader および JPAEntityLoader com.ibm.websphere.objectgrid.jpa.JPAEntityLoader プラグインは、ObjectGrid マップとデータベースを同期するために使用される 2 つの組み込み JPA ローダー・プラグインです。この機能を使用するには、Hibernate または OpenJPA などの JPA 実装がなくてはなりません。データベースは、選択された JPA プロバイダーがサポートする任意のバックエンドを使用できます。

ObjectMap API を使用してデータを保管する場合、JPALoader プラグインを使用することができます。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。

JPA ローダー・アーキテクチャー

JPA ローダー は、Plain Old Java Object (POJO) を保管する eXtreme Scale マップに使用されます。

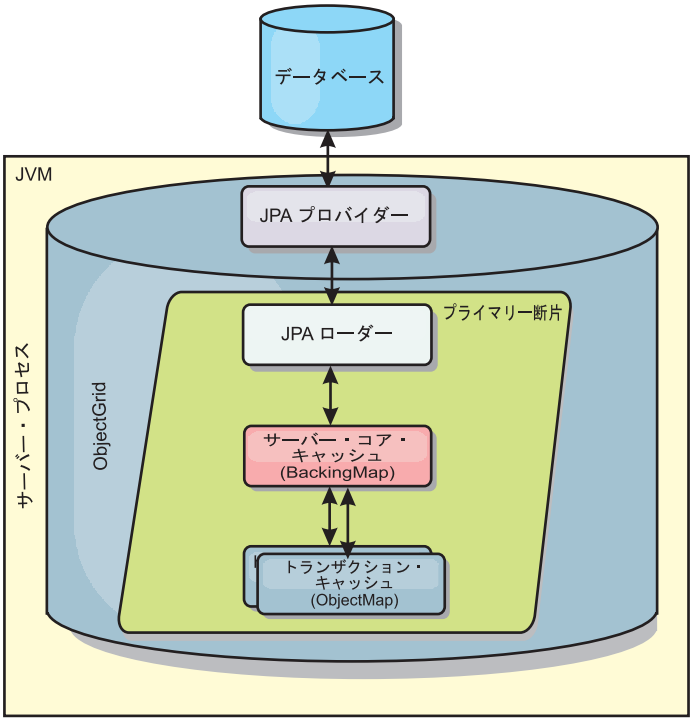


図 28. JPA Loader アーキテクチャー

ObjectMap.get(Object key) メソッドが呼び出されると、eXtreme Scale ランタイムが、まず ObjectMap 層にエントリーがあるかどうかをチェックします。ない場合、ランタイムは、要求を JPA Loader に委任します。キーのロード要求時に、JPALoader は JPA EntityManager.find(Object key) メソッドを呼び出して、JPA 層からのデータを検索します。データが JPA エンティティー・マネージャーに含まれている場合、そのデータが返されます。含まれていない場合は、JPA プロバイダーがデータベースと対話して値を取得します。

例えば、ObjectMap.update(Object key, Object value) メソッドを使用して ObjectMap に対する更新が行われると、eXtreme Scale ランタイムは、この更新に対する LogElement を作成し、これを JPALoader に送ります。JPALoader は、JPA EntityManager.merge(Object value) メソッドを呼び出して、データベースに対する値を更新します。

JPAEntityLoader の場合も、同じ 4 つの層が含まれます。ただし、JPAEntityLoader プラグインは、eXtreme Scale エンティティーを保管するマップに使用されるため、エンティティー間の関係が使用シナリオを複雑にする可能性があります。eXtreme Scale エンティティーは、JPA エンティティーとは区別されます。詳しくは、JPAEntityLoader プラグインを参照してください。

メソッド

ローダーでは、3 つの主要なメソッドを提供しています。

1. get: JPA を使用してデータを取得することにより、渡されたキーのリストに対応する値のリストを返します。このメソッドは、JPA を使用して、データベース内のエンティティーを検出します。JPALoader プラグインの場合、返されるリストには、find 操作から直接得られた JPA エンティティーのリストが含まれます。

JPAEntityLoader プラグインの場合、返されるリストには、JPA エンティティーから変換された eXtreme Scale エンティティー値タプルが含まれます。

2. batchUpdate: ObjectGrid マップのデータをデータベースに書き込みます。異なる操作タイプ (挿入、更新、削除) に応じて、ローダーは、JPA パーシスト、マージ、および除去操作を使用してデータベースに対するデータを更新します。JPALoader の場合、マップ内のオブジェクトが JPA エンティティーとして直接使用されます。JPAEntityLoader の場合、マップ内のエンティティー・タプルが、JPA エンティティーとして使用されるオブジェクトに変換されます。
3. preloadMap: ClientLoader.load クライアント・ローダー・メソッドを使用してマップをプリロードします。区画化マップの場合、preloadMap メソッドは 1 つの区画でのみ呼び出されます。区画は、JPALoader または JPAEntityLoader クラスの preloadPartition プロパティーに指定します。 preloadPartition 値がゼロより小さく設定されているか、*total_number_of_partitions* - 1) より大きく設定されている場合、プリロードは使用不可になります。

JPALoader と JPAEntityLoader のいずれのプラグインも、JPATxCallback クラスで動作し、eXtreme Scale トランザクションと JPA トランザクションを調整します。これら 2 つのローダーを使用するには、JPATxCallback を ObjectGrid インスタンス内に構成する必要があります。

JPA キャッシュ・プラグイン

WebSphere eXtreme Scale には、OpenJPA と Hibernate Java Persistence API (JPA) プロバイダーの両方に対するレベル 2 (L2) のキャッシュ・プラグインが組み込まれています。

eXtreme Scale を L2 キャッシュ・プロバイダーとして使用することにより、データ読み取りおよび照会時のパフォーマンスが向上し、データベースに対する負荷が軽減します。WebSphere eXtreme Scale ではキャッシュがすべてのプロセスで自動的に複製されるので、組み込みキャッシュ実装をしのぐ利点があります。あるクライアントが値をキャッシュすると、他のすべてのクライアントが、そのキャッシュされた値をローカルのメモリー内で使用できるようになります。

OpenJPA および Hibernate ObjectGrid キャッシュ・プラグインを使用すると、組み込み、組み込みの区画化、およびリモートの 3 つのトポロジー・タイプが作成できます。

組み込みトポロジー

組み込みトポロジーでは、各アプリケーションのプロセス・スペース内に eXtreme Scale サーバーを作成します。OpenJPA および Hibernate が、キャッシュのメモリー内コピーで直接読み取りを行い、他のすべてのコピーに書き込みを行います。非同期複製を使用することによって、書き込みのパフォーマンスを向上させることができます。このデフォルト・トポロジーは、キャッシュ・データの量が少なく、1 つのプロセスに十分収まる場合に最も良く機能します。

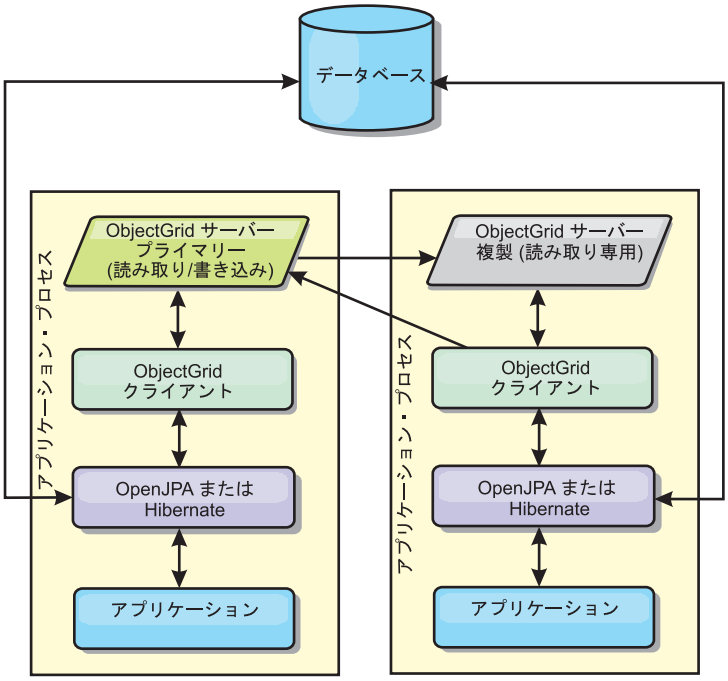


図 29. JPA 組み込みトポロジー

長所:

- すべてのキャッシュ読み取りが非常に速く、ローカル・アクセスである。
- 構成が簡単である。

制約:

- データ量が、プロセスのサイズに限られる。
- すべてのキャッシュ更新が 1 つのプロセスに送られる。

組み込みの区画化トポロジー

キャッシュ・データの量が多く、1 つのプロセスに収まらない場合、組み込みの区画化トポロジーでは、ObjectGrid 区画を使用して、データを複数のプロセスに分割します。多くのキャッシュ読み取りがリモートになるため、パフォーマンスは組み込みトポロジーほど高くありません。データベース待ち時間が大きい場合でも、このオプションを使用できます。

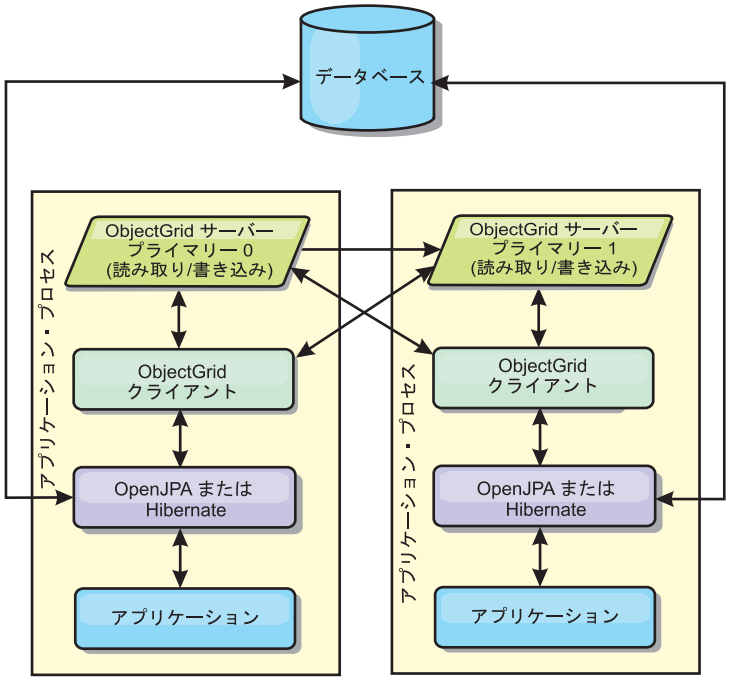


図 30. JPA 組み込みの区画化トポロジー

長所:

- 大容量のデータを保管できる。
- 構成が簡単である。
- キャッシュ更新が複数のプロセスに分散される。

制約:

- ほとんどのキャッシュ読み取りおよび更新がリモートで行われる。

例えば、JVM あたり最大 1 GB で 10 GB のデータをキャッシュに入れる場合、Java 仮想マシン が 10 必要になります。したがって、区画数は 10 以上に設定されます。理想的には、区画数は素数に設定され、各断片が適当な量のメモリーを保管することが望ましいと言えます。通常、numberOfPartitions は、Java 仮想マシン の数に等しくなるようにします。このように設定すれば、各 JVM に 1 つの区画が格納されます。複製を使用可能にする場合、システム内の Java 仮想マシン 数を増やす必要があります。そうでないと、各 JVM ごとに 1 つのレプリカ区画も格納することになり、この区画はプライマリー区画と同量のメモリーを消費します。

例えば、4 つの Java 仮想マシン があるシステムで numberOfPartitions 設定値が 4 の場合、各 JVM は 1 つのプライマリー区画をホストします。読み取り操作では、25 パーセントの確率でローカルで使用可能な区画からデータを取り出すことができ、これは、リモート JVM からデータを取得する場合と比較してはるかに速くなります。照会の実行などの読み取り操作で 4 つの区画が均等に関わるデータ・コレクションを取り出す必要がある場合、呼び出しの 75 パーセントがリモートで、呼び出しの 25 パーセントがローカルです。ReplicaMode が `SYNC` または `ASYNC` に設定され、ReplicaReadEnabled が `true` に設定されている場合、4 つのレプリカ区画が作成され、これが 4 つの Java 仮想マシン に分散されます。各 JVM は、1 つのプライマリー区画と 1 つのレプリカ区画をホストします。読み取り操作をローカル

で実行する確率は、50 パーセントに増えます。4 つの区画が均等に関わるデータ・コレクションを取り出す読み取り操作では、50 パーセントのリモート呼び出しと 50 パーセントのローカル呼び出しがあります。ローカル呼び出しは、リモート呼び出しよりはるかに高速です。リモート呼び出しが行われると、パフォーマンスは落ちます。

リモート・トポロジー

リモート・トポロジーでは、すべてのキャッシュ・データを 1 つ以上の個別のプロセスに保管し、アプリケーション・プロセスのメモリー使用を減らします。区画化され、複製されるように、eXtreme Scaleを構成できます。リモート構成はアプリケーションおよび JPA プロバイダーから独立して管理します。

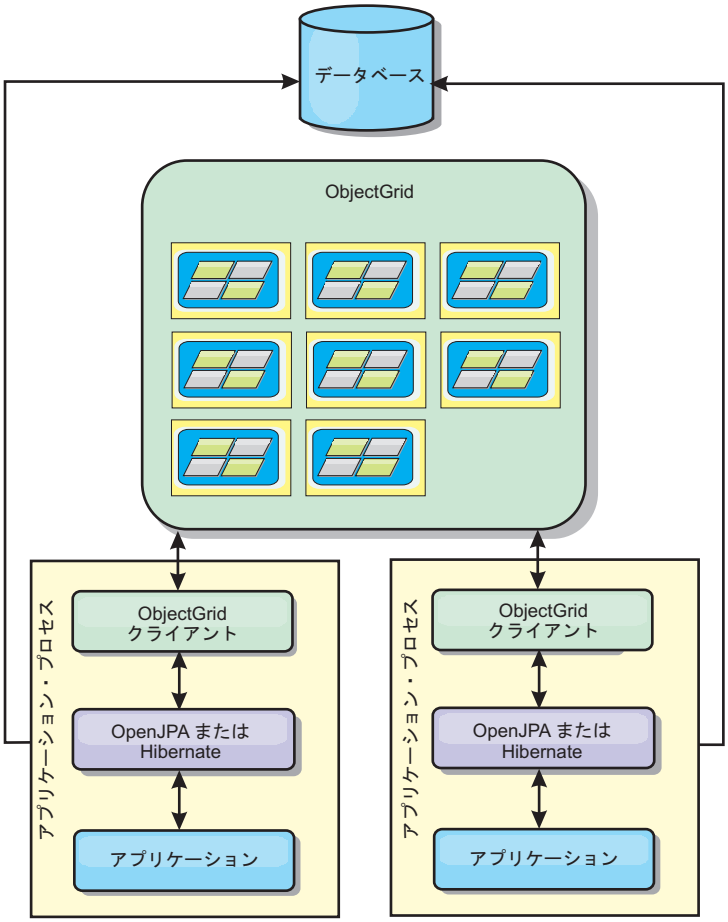


図 31. JPA リモート・トポロジー

長所:

- 大容量のデータを保管できる。
- アプリケーション・プロセスがキャッシュ・データから解放される。
- キャッシュ更新が複数のプロセスに分散される。
- 非常に柔軟な構成オプションがある。

制約:

- すべてのキャッシュ読み取りおよび更新がリモートで行われる。

HTTP セッション管理

WebSphere Application Server バージョン 5 以降の環境では、アプリケーション・サーバーにあるデフォルトのセッション・マネージャーを、WebSphere eXtreme Scale に同梱されている HTTP セッション・マネージャーでオーバーライドすることができます。

WebSphere eXtreme Scale HTTP セッション・マネージャーも WebSphere Application Server バージョン 6.0.2 以降で、または、WebSphere Application Server Community Edition や Apache Tomcat など、WebSphere Application Server を実行しない環境でも実行できます。

フィーチャー

セッション・マネージャーは、いずれの Java Platform, Enterprise Edition バージョン 1.4 コンテナーでも実行できるように設計されています。セッション・マネージャーは、WebSphere API にはまったく依存していないため、ベンダーのアプリケーション・サーバー環境をサポートするのと同様に、さまざまなバージョンの WebSphere Application Server をサポートすることができます。

HTTP セッション・マネージャーは、関連するアプリケーションのセッション管理機能を提供します。セッション・マネージャーは、そのアプリケーションに関連付けられた HTTP セッションを作成し、HTTP セッションのライフサイクルを管理します。このライフサイクル管理には、タイムアウト、明示的サーブレット、または JavaServer Pages (JSP) 呼び出しを基にしたセッションの無効化、およびそのセッションまたは Web アプリケーションと関連付けられているセッション・リスナーの起動などが含まれます。セッション・マネージャーは、そのセッションを ObjectGrid インスタンス内にパーシストします。このインスタンスは、ローカルのメモリー内インスタンスか、あるいは完全に複製されたインスタンス、クラスター化されたインスタンス、および区画に分割されたインスタンスです。後者のトポロジーを使用すると、セッション・マネージャーが、アプリケーション・サーバーがシャットダウンまたは予期せずに終了した場合の HTTP セッション・フェイルオーバー・サポートを提供できます。セッション・マネージャーは、要求をアプリケーション・サーバー層に分散するロード・バランサー層によってアフィニティーが強制されない、アフィニティーをサポートしていない環境でも機能します。

使用に関するシナリオ

セッション・マネージャーは、以下のシナリオで使用できます。

- 典型的マイグレーション・シナリオなど、さまざまなバージョンの WebSphere Application Server をアプリケーション・サーバーとして使用する環境。
- さまざまなベンダーのアプリケーション・サーバーを使用するデプロイメント。例えば、オープン・ソース・アプリケーション・サーバーで開発され、WebSphere Application Server でホストされているアプリケーションが挙げられます。別の例としては、ステージングから実動にプロモートされるアプリケーションがあります。すべての HTTP セッションがライブで、サービスされている間は、これらのアプリケーション・サーバー・バージョンのシームレスなマイグレーションが可能です。

- サーバーのフェイルオーバー中、WebSphere Application Server のデフォルトのサービスの品質 (QoS) レベルよりも高い QoS レベルで、かつセッションの可用性もより強く保証された状態でセッションを保持するようにユーザーに要求する環境。
- セッションのアフィニティーが保証されない環境、または、アフィニティーがベンダーのロード・バランサーによって保守されるため、アフィニティー・メカニズムをそのロード・バランサー用にカスタマイズする必要がある環境。
- セッション管理のオーバーヘッド、および外部 Java プロセスに対するストレージの負荷を軽減する環境。
- セル間のセッション・フェイルオーバーを使用可能にする複数のセル。

セッション・マネージャーの動作

セッション・マネージャーは、それ自体をサーブレット・フィルターの形で要求パスに導入します。WebSphere eXtreme Scale に付属しているツールを使用して、このサーブレット・フィルターをアプリケーション内の各 Web モジュールに追加できます。また、これらのフィルターを、アプリケーションの Web デプロイメント記述子に手動で追加することも可能です。このフィルターは、ターゲット・アプリケーション内のサーブレットまたは JSP ファイルの前に要求を受け取ります。このときフィルターは、そのフィルターの実装環境で、HTTPRequest オブジェクトおよび HTTPResponse オブジェクトを代行受信し、ラッパー・オブジェクトを作成します。

このフィルターの実装環境は、HTTPRequest オブジェクトおよび HTTPResponse オブジェクトで作成された、HTTP セッションに関連するすべての呼び出しを代行受信します。これらの呼び出しは、セッション・マネージャーによって処理され、基礎となる Java Platform, Enterprise Edition サーバー実装環境内の基本セッション・マネージャーには渡されません。セッション・マネージャーは、セッションおよびそのセッションのライフサイクルを作成および管理します。これには、タイムアウト・ベースの無効化、セッション無効化のリスナーの始動、およびその他のライフサイクル・イベントが含まれます。

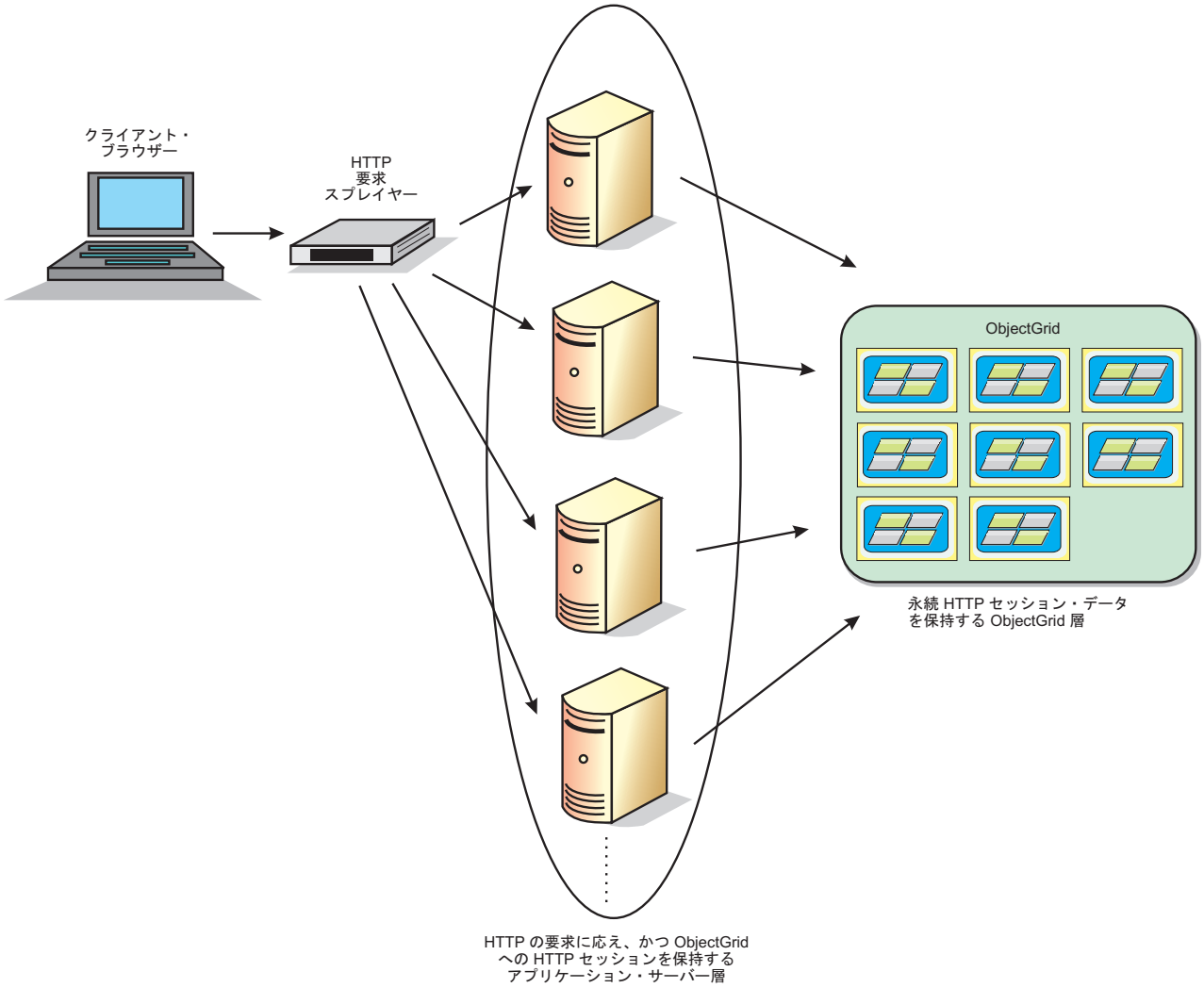


図 32. リモート・コンテナー構成を含む HTTP セッション管理トポロジー

デプロイメント・トポロジー

セッション・マネージャーは、以下の 2 つの異なる動的デプロイメント・シナリオを使用して構成することができます。

- **組み込みの、ネットワーク接続 eXtreme Scale コンテナー**

  このシナリオでは、eXtreme Scale サーバーは、サーブレットと同じプロセス内に連結されます。セッション・マネージャーはローカル ObjectGrid インスタンスに直接通信できるため、コストのかかるネットワーク遅延を回避することができます。

- **リモートの、ネットワーク接続 eXtreme Scale コンテナー**

  このシナリオでは、eXtreme Scale サーバーは、サーブレットが実行される外部プロセスで実行されます。セッション・マネージャーは、リモート eXtreme Scale と通信します。

セッション・マネージャー・アプリケーションにおける eXtreme Scale セッションの取得

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

        HttpSession session = req.getSession(true);

        System.out.println("calling getSession");
        // call getAttribute("com.ibm.websphere.objectgrid.session")
    // to get the ObjectGrid session instance
        Session ogSession = (Session)session.getAttribute
      ("com.ibm.websphere.objectgrid.session");

        System.out.println("ogSession = "+ogSession);
    }
```

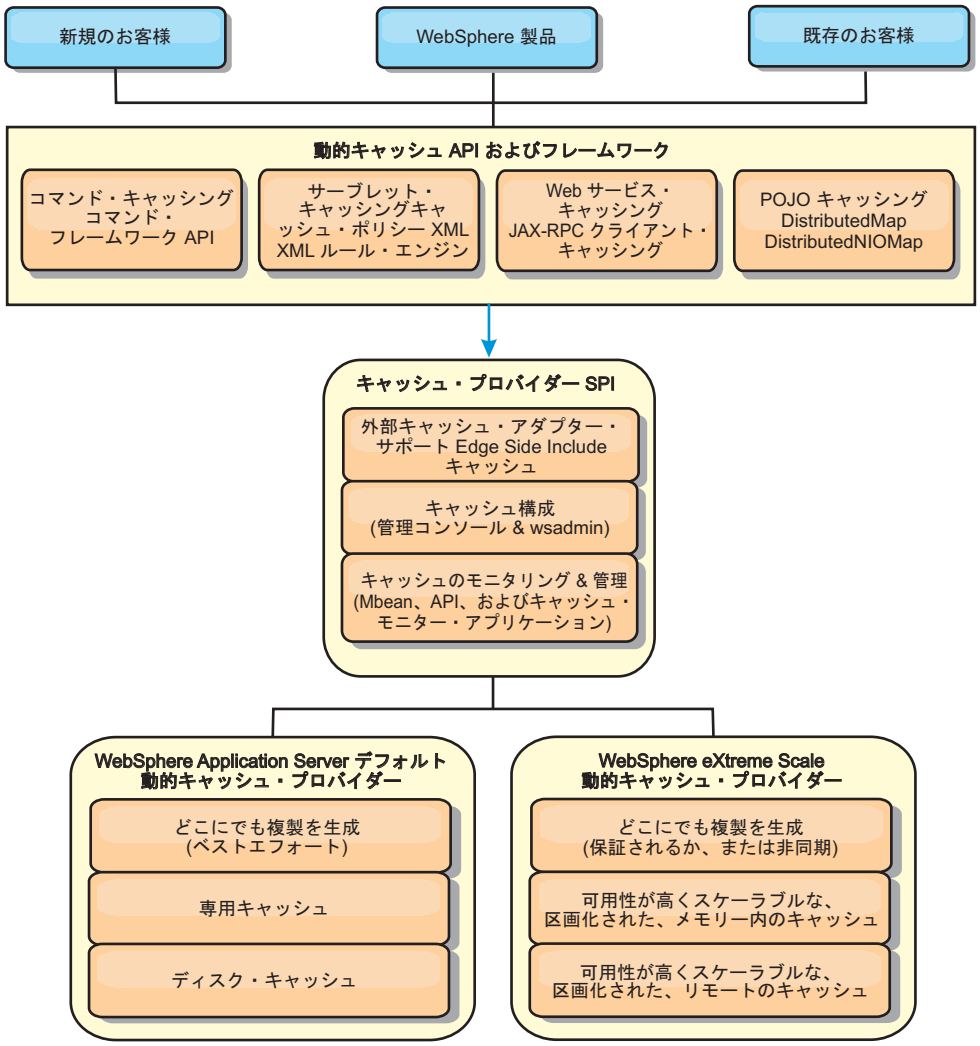
getAttribute メソッド呼び出しから返されるセッションでマップが変更された場合、その変更は、基礎となるセッションがコミットされた時点でコミットされます。

WebSphere eXtreme Scale 動的キャッシュ・プロバイダー

WebSphere Application Server にデプロイされた Java EE アプリケーションでは、動的キャッシュ API を使用できます。動的キャッシュは、ビジネス・データや生成された HTML をキャッシュに入れるために、または、データ複製サービス (DRS) を使用してセル内のキャッシュ・データを同期化するために利用できます。

概説

以前は、動的キャッシュ API の唯一のサービス・プロバイダーは、 WebSphere Application Server に組み込まれた、デフォルトの動的キャッシュ・エンジンでした。ユーザーは、 WebSphere Application Server 内の動的キャッシュ・サービス・プロバイダー・インターフェースを使用して、eXtreme Scale を動的キャッシュに接続できます。この機能をセットアップすると、動的キャッシュ API を使用して書かれたアプリケーションまたはコンテナー・レベル・キャッシュを使用するアプリケーション (サーブレットなど) が WebSphere eXtreme Scaleの機能およびパフォーマンス能力を利用できるようになります。



動的キャッシュ・プロバイダーのインストールと構成の手順については、69 ページの『WebSphere eXtreme Scale の動的キャッシュ・プロバイダーの構成』を参照してください。

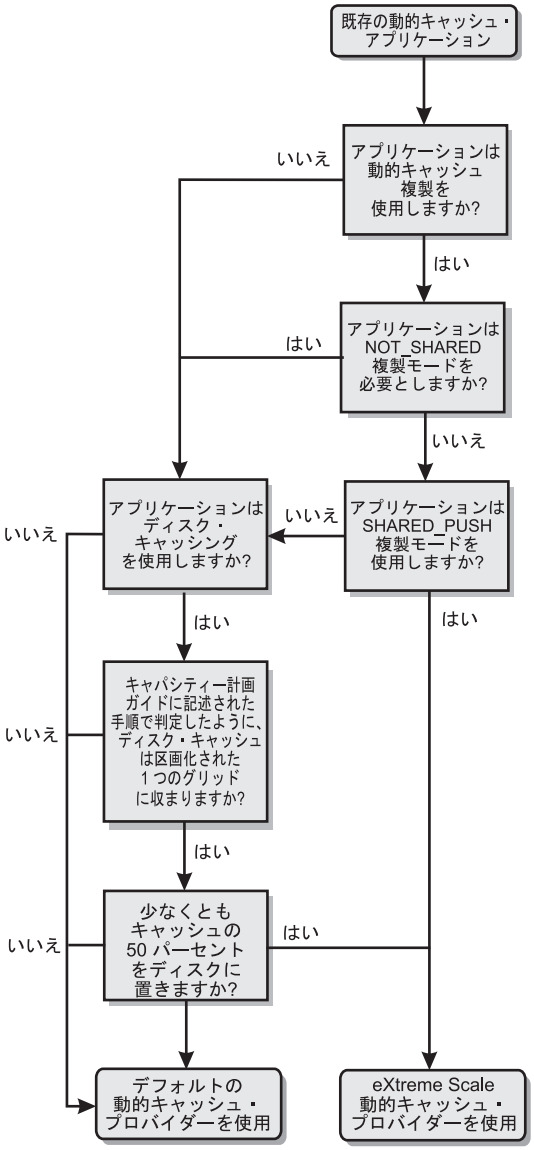
WebSphere eXtreme Scale の利用方法の決定

WebSphere eXtreme Scale で使用可能なフィーチャーによって動的キャッシュ API の分散機能は大幅に強化され、デフォルト動的キャッシュ・エンジンおよびデータ複製サービスで提供される機能を超えるものになっています。eXtreme Scale を使用することにより、複数のサーバー間で単に複製し、同期化するだけでなく、サーバー間で本当に分散したキャッシュを作成できます。さらに、eXtreme Scale キャッシュは、トランザクション・ベースであり、可用性がとても高く、動的キャッシュ・サービスに関して各サーバーが同じ内容を参照することを保証します。WebSphere eXtreme Scale は、キャッシュ複製に関して、DRS よりも高いサービス品質を提供します。

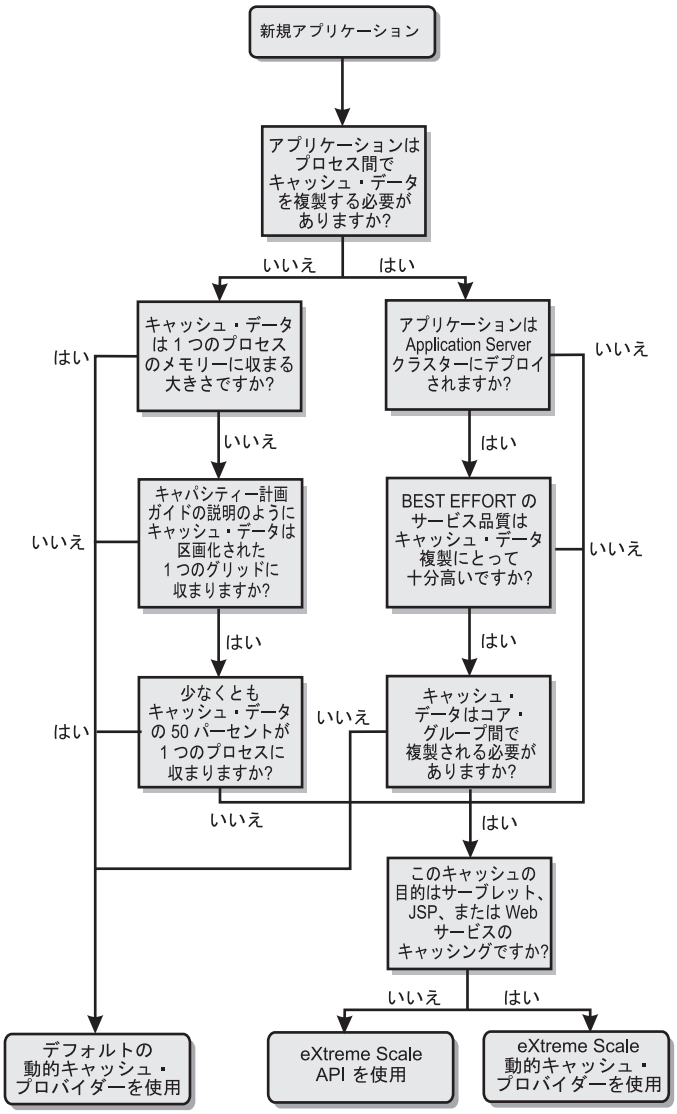
ただし、これらの利点は、どのようなアプリケーションでも eXtreme Scale 動的キャッシュ・プロバイダーが正しい選択であることを意味するわけではありません。以下のデシジョン・ツリーおよび機能比較マトリックスを使用して、ご使用のアプ

リケーションに最適のテクノロジーを決定してください。

既存の動的キャッシュ・アプリケーションをマイグレーションする際のデシジョン・ツリー



新規アプリケーションのキャッシュ・プロバイダーを選択する際のデシジョン・ツリー



機能比較

表 3. 機能比較

キャッシュ機能	デフォルト・プロバイダー	eXtreme Scale プロバイダー	eXtreme ScaleAPI
ローカルのメモリー内のキャッシング	x	x	x
分散キャッシング	組み込み	組み込み、組み込み区画化、およびリモート区画化	Multiple
直線的にスケーラブル		x	x
信頼できる複製 (同期)		ORB	ORB

表 3. 機能比較 (続き)

キャッシュ機能	デフォルト・プロバイダー	eXtreme Scale プロバイダー	eXtreme ScaleAPI
ディスク・オーバーフロー	x		
除去	LRU/TTL/ヒープ・ベース	LRU/TTL (区画ごと)	Multiple
無効化	x	x	x
リレーションシップ	依存関係 ID、テンプレート	依存関係 ID、テンプレート	x
非キー検索			照会および索引
バックエンド統合			ローダー
トランザクションの		暗黙	x
キー・ベースの保管	x	x	x
イベントおよびリスナー	x	x	x
WebSphere Application Server 統合	単一セルのみ	複数セル	セルに無関係
Java Standard Edition サポート		x	x
モニタリングおよび統計	x	x	x
セキュリティー	x	x	x

表 4. シームレスなテクノロジー統合

キャッシュ機能	デフォルト・プロバイダー	eXtreme Scale プロバイダー	eXtreme ScaleAPI
WebSphere Application Server サーブレット/JSP 結果キャッシング	V5.1+	V6.1.0.25+	
WebSphere Application Server Web Services (JAX-RPC) 結果キャッシング	V5.1+	V6.1.0.25+	
HTTP セッション・キャッシング			x
OpenJPA および Hibernate 用のキャッシュ・プロバイダー			x
OpenJPA および Hibernate を使用したデータベース同期			x

表 5. プログラミング・インターフェース

キャッシュ機能	デフォルト・プロバイダー	eXtreme Scale プロバイダー	eXtreme ScaleAPI
コマンド・ベース API	コマンド・フレームワーク API	コマンド・フレームワーク API	DataGrid API
マップ・ベース API	DistributedMap API	DistributedMap API	ObjectMap API
EntityManager API			x

eXtreme Scale 分散キャッシュがどのように機能するのかについて詳しくは、「*プログラミング・ガイド*」の『eXtreme Scale のデプロイメント構成』を参照してください。

注: eXtreme Scale 分散キャッシュは、キーと値が両方とも java.io.Serializable インターフェースを実装するエントリーのみを保管できます。

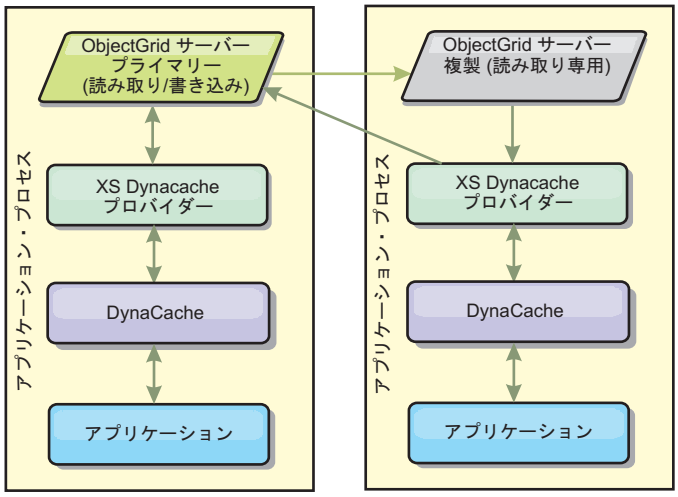
トポロジーのタイプ

eXtreme Scale プロバイダーを使用して作成された動的キャッシュ・サービスは、パフォーマンス、リソース、および管理上の必要に合わせて、3 つのトポロジーのいずれかにデプロイできます。これらのトポロジーは、組み込み、組み込み区画化、およびリモートです。

組み込みトポロジー

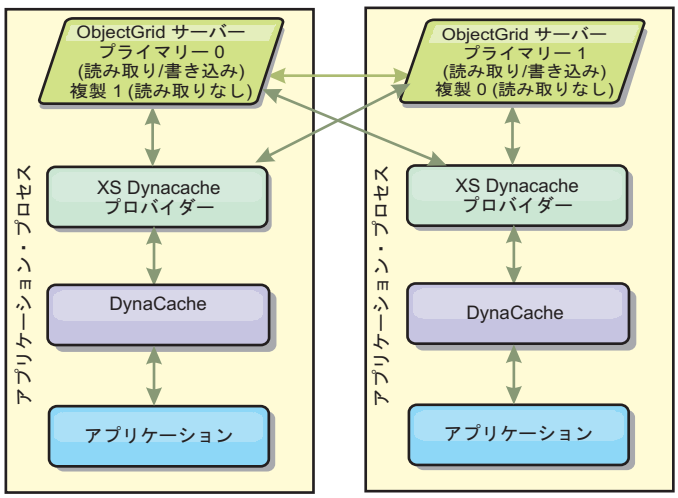
組み込みトポロジーは、デフォルトの動的キャッシュおよび DRS プロバイダーに似ています。組み込みトポロジーで作成された分散キャッシュ・インスタンスは、動的キャッシュ・サービスにアクセスする各 eXtreme Scale プロセスの内部でキャッシュの完全コピーを保持するので、すべての読み取り操作をローカルで実行できます。すべての書き込み操作は、シングル・サーバー・プロセスを完了し、そのプロセスでトランザクションのロックが管理された後で残りのサーバーに複製されます。したがって、このトポロジーは、キャッシュ読み取り操作がキャッシュ書き込み操作よりもずっと多いワークロードに向いています。

組み込みトポロジーでは、新規および更新されたキャッシュ・エントリーが、すべてのサーバー・プロセスで即時に可視になるわけではありません。キャッシュ・エントリーは、WebSphere eXtreme Scale の非同期複製サービスによって伝搬されるまでは、そのエントリーを生成したサーバーに対してさえ可視になりません。これらのサービスは、ハードウェアで可能な限り速く作動しますが、それでも少しは遅延が発生します。次の図に、組み込みトポロジーを示します。



組み込み区画化トポロジー

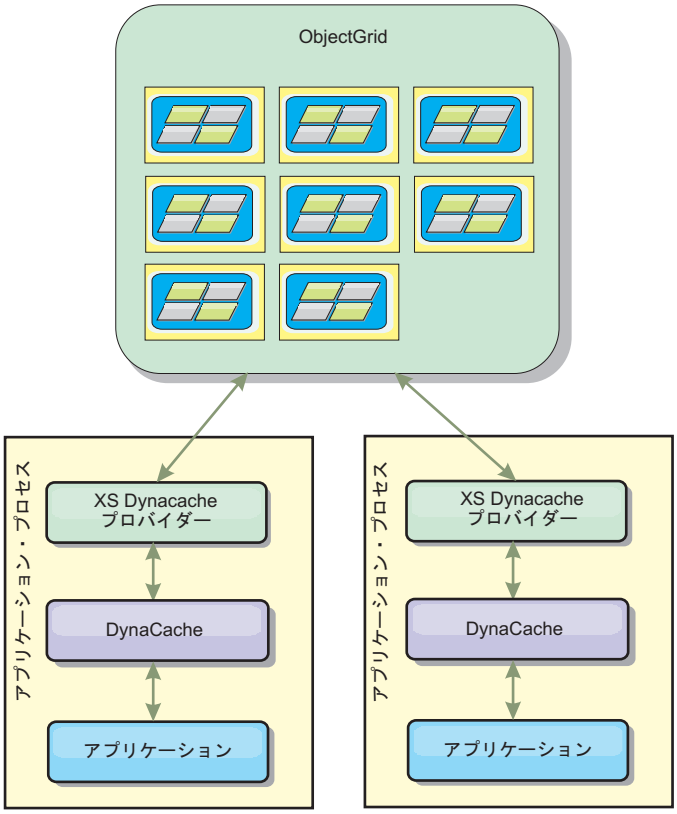
キャッシュ書き込みが読み取りと同じくらいか、より頻繁に発生するようなワークロードの場合、組み込み区画化トポロジーまたはリモート・トポロジーが推奨されます。組み込み区画化トポロジーは、キャッシュにアクセスする WebSphere Application Server プロセスの内部でキャッシュ・データのすべてを保持します。しかし、各プロセスが保管するのは、キャッシュ・データの一部分のみです。この「区画」に置かれたデータに対するすべての読み取りおよび書き込みは、そのプロセスを実行します。したがって、キャッシュに対するほとんどの要求はリモート・プロシージャー・コールで達成されることになります。その結果、読み取り操作の待ち時間は組み込みトポロジーよりも大きくなりますが、読み取り操作および書き込み操作を処理するための分散キャッシュの容量は、キャッシュにアクセスする WebSphere Application Server プロセスの数に応じて直線的に変化します。また、このトポロジーでは、キャッシュの最大サイズが 1 つの WebSphere プロセスのサイズに制約されることはありません。各プロセスはキャッシュの一部分だけを保持するので、キャッシュの最大サイズは、すべてのプロセスの総計サイズからプロセスのオーバーヘッドを引いたものになります。次の図に、組み込み区画化トポロジーを示します。



例えば、あるグリッドのサーバー・プロセスのそれぞれに、動的キャッシュ・サービスをホストするための 256 メガバイトの空きヒープがあるとします。組み込みトポロジーを使用する場合、デフォルトの動的キャッシュ・プロバイダーと eXtreme Scale プロバイダーの両方とも、メモリー内キャッシュのサイズは、256 メガバイトからオーバーヘッドを引いた値に制限されます。この資料のもう少し後にある『キャパシティー・プランニングおよび高可用性』セクションを参照してください。組み込み区画化トポロジーを使用する eXtreme Scale プロバイダーの場合、キャッシュ・サイズは 1 ギガバイトからオーバーヘッドを引いた値に制限されます。このようにして、WebSphere eXtreme Scale プロバイダーは、単一のサーバー・プロセスのサイズよりも大きいメモリー内の動的キャッシュのサービスを可能にします。デフォルトの動的キャッシュ・プロバイダーは、ディスク・キャッシュの使用に頼ることで、キャッシュ・インスタンスが大きくなって単一プロセスのサイズを超えることを可能にしています。多くのシチュエーションで、WebSphere eXtreme Scale プロバイダーを使用すると、実行に必要なディスク・キャッシュやコストの高いディスク・ストレージ・システムの必要性をなくすことができます。

リモート・トポロジー

リモート・トポロジーを使用しても、ディスク・キャッシュの必要性をなくすことができます。 リモート・トポロジーと組み込み区画化トポロジーとの唯一の相違点は、リモート・トポロジーを使用しているときは、キャッシュ・データのすべてが WebSphere Application Server プロセスの外側に保管されることです。WebSphere eXtreme Scale は、キャッシュ・データ用にスタンドアロンのコンテナー・プロセスをサポートします。これらのコンテナー・プロセスのオーバーヘッドは WebSphere Application Server プロセスよりも小さく、特定の Java 仮想マシン (JVM) を使用しなければならないという制限もありません。例えば、 32 ビット WebSphere Application Server プロセスによってアクセスされる動的キャッシュ・サービスのデータを、64 ビット JVM 上で実行している eXtreme Scale コンテナー・プロセス内に置くことが可能です。これによって、ユーザーは、64 ビット・プロセスの大きなメモリー容量をキャッシング用に利用できると同時に、アプリケーション・サーバー・プロセス用には 64 ビットの追加オーバーヘッドを負わなくてもすみます。次の図に、リモート・トポロジーを示します。



データ圧縮

WebSphere eXtreme Scale 動的キャッシュ・プロバイダーによって提供される、キャッシュ・オーバーヘッド管理についてユーザーを支援するもう 1 つのパフォーマンス機能が、圧縮です。デフォルトの動的キャッシュ・プロバイダーは、メモリー内のキャッシュ・データの圧縮を許可していません。eXtreme Scale プロバイダーを使用すると、これが可能になります。3 種類の分散トポロジーのどれでも、デフレート・アルゴリズムを使用するキャッシュ圧縮を使用可能にできます。圧縮を使用可能にすると、読み取りおよび書き込み操作のオーバーヘッドは増えますが、サーブレットおよび JSP キャッシングのようなアプリケーション用のキャッシュ密度は大幅に増加します。

ローカルのメモリー内のキャッシュ

WebSphere eXtreme Scale 動的キャッシュ・プロバイダーは、**複製使用不可**の動的キャッシュ・インスタンスをバックアップするためにも使用できます。デフォルトの動的キャッシュ・プロバイダーと同じように、これらのキャッシュは非シリアライズ可能データを保管できます。また、eXtreme Scale コード・パスはメモリー内キャッシュの並行性を最大化するよう設計されているため、大容量のマルチプロセッサー・エンタープライズ・サーバー上ではデフォルト動的キャッシュ・プロバイダーよりも優れたパフォーマンスを提供します。

動的キャッシュ・エンジンおよび eXtreme Scale の機能の相違点

ローカルのメモリー内のキャッシュで、複製が使用不可にされている場合は、デフォルトの動的キャッシュ・プロバイダーによってバックアップされたキャッシュと

WebSphere eXtreme Scale によってバックアップされたキャッシュとの間には、それほどの機能的な違いはありません。WebSphere eXtreme Scale がバックアップしたキャッシュは、メモリー内キャッシュのサイズに関係する統計および操作、またはディスク・オフロードをサポートしない点は除いて、ユーザーはこれら 2 つのキャッシュの機能的な違いに気付かないはずです。

複製が使用可能にされているキャッシュの場合は、デフォルトの動的キャッシュ・プロバイダーを使用しているのか、それとも eXtreme Scale 動的キャッシュ・プロバイダーを使用しているのかに関わらず、ほとんどの動的キャッシュ API 呼び出しで戻される結果にはそれほどの相違はありません。一部の操作については、eXtreme Scale を使用して動的キャッシュ・エンジンの動作をエミュレートできません。

動的キャッシュ統計

動的キャッシュ統計は、CacheMonitor アプリケーションまたは動的キャッシュ MBean を介して報告されます。eXtreme Scale 動的キャッシュ・プロバイダーを使用している場合でも、統計はこれらのインターフェースを通して報告されますが、統計値のコンテキストは異なります。

A、B、C という名前の 3 つのサーバー間で 1 つの動的キャッシュ・インスタンスが共有されている場合、動的キャッシュ統計オブジェクトは、その呼び出しが実行されたサーバーにあるキャッシュのコピーに関する統計だけを戻します。サーバー A で統計が取得された場合、その統計はサーバー A でのアクティビティーのみを反映します。

eXtreme Scale を使用している場合、すべてのサーバー間で共有されている分散キャッシュは 1 つしかありません。したがって、デフォルトの動的キャッシュ・プロバイダーのようにほとんどの統計値をサーバーごとにトラッキングするというのは不可能です。WebSphere eXtreme Scale 動的キャッシュ・プロバイダーを使用している場合に、キャッシュ統計 API によって報告される統計のリストと、それぞれの統計が何を表すのかを以下に示します。デフォルト・プロバイダーのように、これらの統計は同期化されていないため、並行ワークロードのために最高 10% は変わる可能性があります。

- **キャッシュ・ヒット** : キャッシュ・ヒットはサーバーごとにトラッキングされます。サーバー A 上のトラフィックが 10 キャッシュ・ヒットを生成し、サーバー B 上のトラフィックが 20 キャッシュ・ヒットを生成する場合、キャッシュ統計は、サーバー A での 10 キャッシュ・ヒットとサーバー B での 20 キャッシュ・ヒットを報告します。
- **キャッシュ・ミス**: キャッシュ・ミスは、キャッシュ・ヒットと同様、サーバーごとにトラッキングされます。
- **メモリー・キャッシュ・エントリー数**: この統計値は、分散キャッシュ内のキャッシュ・エントリーの数を報告します。この統計値に関しては、キャッシュにアクセスするすべてのサーバーが同じ値を報告し、その値は、サーバーすべてのメモリー内のキャッシュ・エントリーの総数です。
- **メモリー・キャッシュ・サイズ (MB)**: このメトリックは現在はサポートされていません。常に -1 が戻されます。

- **キャッシュ除去**: この統計値は、任意の方法でキャッシュから除去されたエントリーの総数を報告し、分散キャッシュ全体の集約値です。サーバー A 上のトラフィックが 10 の無効化を生成し、サーバー B 上のトラフィックが 20 の無効化を生成する場合、両サーバーの値は 30 になります。
- **キャッシュ最長未使用時間 (LRU) 除去**: この統計値は、キャッシュ除去と同様、集約値です。キャッシュを最大サイズより小さく保っておくために除去されたエントリーの数がトラッキングされます。
- **タイムアウト無効化**: これも集約の統計値であり、タイムアウトになったために除去されたエントリーの数をトラッキングします。
- **明示的無効化**: これも集約の統計値であり、キー、依存関係 ID、またはテンプレートによる直接的な無効化で除去されたエントリーの数をトラッキングします。
- **拡張統計**: eXtreme Scale 動的キャッシュ・プロバイダーは、以下の拡張統計キー・ストリングをエクスポートします。
  - **com.ibm.websphere.xs.dynacache.remote_hits**: eXtreme Scale コンテナーでトラッキングされたキャッシュ・ヒットの総数。これは、集約統計値であり、拡張統計マップ内ではこの値は `long` です。
  - **com.ibm.websphere.xs.dynacache.remote_misses**: eXtreme Scale コンテナーでトラッキングされたキャッシュ・ミスの総数。集約統計値であり、拡張統計マップ内ではこの値は `long` です。

統計リセットの報告

動的キャッシュ・プロバイダーを使用して、キャッシュ統計をリセットすることができます。デフォルト・プロバイダーでは、リセット操作でクリアされるのは、影響を受けるサーバーの統計のみです。 eXtreme Scale 動的キャッシュ・プロバイダーは、リモート・キャッシュ・コンテナーの統計データの大部分をトラッキングします。このデータは、統計がリセットされたときに、クリアされることも、変更されることもありません。代わりに、デフォルト動的キャッシュ動作がクライアント上でシミュレートされ、ある統計の現行値と、そのサーバーで最後にリセットが呼び出されたときのその統計の値との差が報告されます。

例えば、サーバー A 上のトラフィックが 10 のキャッシュ除去を生成する場合、サーバー A とサーバー B の統計は 10 の除去を報告します。サーバー B の統計がリセットされ、サーバー A 上のトラフィックが追加で 10 の除去を生成したとすると、サーバー A の統計は 20 の除去を報告し、サーバー B の統計は 10 の除去を報告します。

動的キャッシュ・イベント

動的キャッシュ API を使用して、ユーザーはイベント・リスナーを登録できます。動的キャッシュ・プロバイダーとして eXtreme Scale を使用している場合、イベント・リスナーはローカルのメモリー内キャッシュに対して、予期されるように機能します。

分散キャッシュに対するイベント動作は、使用されるトポロジーに依存します。組み込みトポロジーを使用しているキャッシュの場合、イベントは書き込み操作を処理するサーバー (つまり、プライマリー断片) で生成されます。これは、1 つのサーバーのみがイベント通知を受け取ることを意味しますが、動的キャッシュ・プロバ

イダーで一般的に予期されるイベント通知はすべてそのサーバーが受け取ります。WebSphere eXtreme Scale は実行時にプライマリー断片を選択するため、ある特定のサーバー・プロセスが常にこれらのイベントを受け取るように保証することはできません。

組み込み区画化キャッシュは、キャッシュのいずれかの区画をホストするどのサーバー上でも、イベントを生成します。 したがって、11 の区画に分割されたキャッシュがあり、WebSphere Network Deployment グリッドの 11 のサーバーそれぞれが 1 つの区画をホストしている場合、各サーバーは、そのサーバーがホストしているキャッシュ・エントリーの動的キャッシュ・イベントを受け取ります。11 すべての区画が 1 つのサーバー・プロセスでホストされているのでない限り、1 つのサーバー・プロセスだけがイベントのすべてを認識するということはありません。組み込みトポロジーの場合と同様、ある特定のサーバー・プロセスが、ある特定のイベント・セットまたはイベントを受け取るように保証することはできません。

リモート・トポロジーを使用するキャッシュは、動的キャッシュ・イベントをサポートしません。

MBean 呼び出し

WebSphere eXtreme Scale 動的キャッシュ・プロバイダーはディスク・キャッシングをサポートしません。ディスク・キャッシングに関係する MBean 呼び出しはすべて機能しません。

動的キャッシュ複製ポリシーのマッピング

WebSphere Application Server 組み込み動的キャッシュ・プロバイダーは、複数のキャッシュ複製ポリシーをサポートします。これらのポリシーは、グローバルに構成するか、各キャッシュ・エントリーに対して構成することができます。動的キャッシュ資料で、これらのポリシーに関する説明を参照してください。

eXtreme Scale 動的キャッシュ・プロバイダーは、直接これらのポリシーに従うわけではありません。キャッシュの複製に関する特性は、動的キャッシュ・サービスによってエントリーに設定される複製ポリシーに関わらず、構成された eXtreme Scale 分散トポロジー・タイプによって決まり、そのキャッシュ内に置かれるすべての値に適用されます。以下に、動的キャッシュ・サービスでサポートされる複製ポリシーのすべてをリストし、どの eXtreme Scale トポロジーが類似の複製特性を提供するのかを示します。

eXtreme Scale 動的キャッシュ・プロバイダーは、キャッシュまたはキャッシュ・エントリーに対する DRS 複製ポリシー設定を無視することに注意してください。ユーザーは、複製のニーズに適したトポロジーを選択する必要があります。

- NOT_SHARED – 現在は、eXtreme Scale 動的キャッシュ・プロバイダーによって提供されるトポロジーのうち、このポリシーに近似するものはありません。これは、キャッシュに保管されるすべてのデータは、java.io.Serializable を実装するキーおよび値を持っていなければならないことを意味します。
- SHARED_PUSH – 組み込みトポロジーが、この複製ポリシーに近似しています。キャッシュ・エントリーが作成されると、そのエントリーはすべてのサーバーに複製されます。サーバーは、キャッシュ・エントリーをローカルで検索するだけ

です。エントリーがローカルで検出されなかった場合は存在しないと見なされ、そのエントリーを探すために他のサーバーが照会されることはありません。

- SHARED_PULL および SHARED_PUSH_PULL – 組み込み区画化トポロジーおよびリモート・トポロジーが、この複製ポリシーに近似しています。キャッシュの分散状態は、すべてのサーバー間で完全に一貫しています。

この情報が主として提供されるので、トポロジーをユーザーの分散整合性ニーズに確実に合わせることができます。例えば、ユーザーのデプロイメントおよびパフォーマンスのニーズには組み込みトポロジーが適しているが、SHARED_PUSH_PULL で提供されるレベルのキャッシュ整合性を必要とする場合、パフォーマンスが少し劣ることになっても、組み込み区画化トポロジーを使用することを検討してください。

セキュリティー

組み込みトポロジーまたは組み込み区画化トポロジーで実行している動的キャッシュ・インスタンスを、WebSphere Application Server に構築されたセキュリティー機能を使用して保護できます。WebSphere Application Server インフォメーション・センターでアプリケーション・サーバーの保護を参照してください。

リモート・トポロジーでキャッシュが実行している場合、スタンドアロン eXtreme Scale クライアントはそのキャッシュに接続して、動的キャッシュ・インスタンスの内容に影響を与えることが可能です。eXtreme Scale 動的キャッシュ・プロバイダーに備わっている低オーバーヘッドの暗号化機能は、非 WebSphere Application Server クライアントによるキャッシュ・データの読み取りまたは変更を防ぐことができます。この機能を使用可能にするには、オプション・パラメーター **com.ibm.websphere.xs.dynacache.encryption_password** を、動的キャッシュ・プロバイダーにアクセスするすべての WebSphere Application Server インスタンスで同じ値に設定します。これによって、128 ビット AES 暗号化を使用して CacheEntry の値およびユーザー・メタデータが暗号化されます。 すべてのサーバーで同じ値に設定されていることが重要です。サーバーは、このパラメーターに異なる値が設定されているサーバーによってキャッシュに入れられたデータを読み取ることはできません。

eXtreme Scale プロバイダーは、同じキャッシュでこの変数に異なる値が設定されていることを検出すると、警告を生成して eXtreme Scale コンテナー・プロセスのログに入れます。

SSL またはクライアント認証が必要な場合は、セキュリティーに関する eXtreme Scale 資料を参照してください。

追加情報

- 動的キャッシュに関するレッドブック
- 動的キャッシュ文書
  - WebSphere Application Server 7.0
  - WebSphere Application Server 6.1
- DRS 資料
  - WebSphere Application Server 7.0

– WebSphere Application Server 6.1

WebSphere eXtreme Scale の動的キャッシュ・プロバイダーの構成

eXtreme Scale の動的キャッシュ・プロバイダーのインストールと構成は、要件と、セットアップした環境によって異なります。

始める前に

動的キャッシュ・プロバイダーをインストールします。

動的キャッシュ・プロバイダーを使用するには、WebSphere Application Server ノードの各デプロイメント (デプロイメント・マネージャー・ノードを含む) 上に WebSphere eXtreme Scale をインストールしておく必要があります。 eXtreme Scale 動的キャッシュ・プロバイダーは、以下のバージョンの WebSphere Application Server でサポートされています。

- WebSphere Application Server 6.1.0.25 + PK85622 以降
- WebSphere Application Server 7.0.0.3 + PK85622 以降

インストールの説明は、バージョン 6.1 の場合のインストール (Installing for 6.1)または、バージョン 7.0 の場合のインストール (Installing for 7.0)を参照してください。

このタスクについて

eXtreme Scale 動的キャッシュ・プロバイダーを構成するには、以下のステップに従ってください。

1. eXtreme Scale 動的キャッシュ・プロバイダーを使用可能にします。

   WebSphere Application Server 7.0 では、動的キャッシュ・サービスを構成すると、管理コンソールまたはカスタム・プロパティーで eXtreme Scale 動的キャッシュ・プロバイダーを使用できます。

   eXtreme Scale をインストールすると、 eXtreme Scale 動的キャッシュ・プロバイダーが、管理コンソールの「キャッシュ・プロバイダー」オプションとして即時に使用可能になります。詳しくは、 動的キャッシュ・サービス・プロバイダーの選択 (Selecting a cache service provider) を参照してください。

   また、キャッシュ・インスタンスの eXtreme Scale 動的キャッシュ・プロバイダーは、以下のカスタム・プロパティーと値の組をそのインスタンス上で設定することでも構成できます。これらのカスタム・プロパティーは、WebSphere Application Server 6.1 上の eXtreme Scale プロバイダーを使用可能にする唯一の方法で、次のようになります。

   ```
   com.ibm.ws.cache.CacheConfig.cacheProviderName =
      "com.ibm.ws.objectgrid.dynacache.CacheProviderImpl"
   ```

   eXtreme Scale 動的キャッシュ・プロバイダーを JSP、Web サービス、あるいは、コマンド・キャッシングのために使用する場合、 eXtreme Scale 動的キャッシュ・プロバイダーを使用するように baseCache インスタンスを設定する必要があります。 baseCache インスタンスの構成には同じ構成プロパティーが使用され

ます。また、これらの構成プロパティーは、JVM カスタム・プロパティーとして設定する必要があることに注意してください。この注意は、サーブレット・キャッシングを除き、このセクションで説明するすべてのキャッシュ構成プロパティーに該当します。サーブレット・キャッシングに動的キャッシュ・プロバイダーを使用して eXtreme Scale を使用する場合には、必ず、カスタム・プロパティーではなくシステム・プロパティーで使用可能化を構成してください。

baseCache が eXtreme Scale 動的キャッシュ・プロバイダーを使用するように構成されると、そのサーバーの他のすべてのキャッシュ・インスタンスが、デフォルトで eXtreme Scale プロバイダーを使用するようになります。キャッシュ・インスタンスがデフォルトの動的キャッシュ・プロバイダーを使用するようにするには、キャッシュ・プロバイダー・プロパティーを次のように設定します: com.ibm.ws.cache.CacheConfig.cacheProviderName = ″default″ baseCache に設定されるすべての eXtreme Scale 動的キャッシュ・プロバイダー構成プロパティーは、eXtreme Scale でサポートされるすべてのキャッシュ・インスタンスのデフォルトの構成プロパティーになります。これらのプロパティーでデフォルト値を利用することについては、お客様は十分気をつける必要があります。

2. キャッシュの複製設定を構成します。

   eXtreme Scale 動的キャッシュ・プロバイダーを使用すると、ローカル・キャッシュ・インスタンスと複製キャッシュ・インスタンスを持つことができます。ローカル・キャッシュ・インスタンスの場合、それ以上の構成は必要がないので、このセクションの残りの部分はスキップしてかまいません。

   複製キャッシュの作成には、2 とおりの方法があります。 1 つ目の方法は、管理コンソールからキャッシュ複製を使用可能にすることです。この使用可能化は、WebSphere Application Server バージョン 7.0 では常に実行できますが、WebSphere Application Server バージョン 6.1 では、DRS 複製ドメインの作成が必要になります。

   2 つ目の方法は、以下のカスタム・プロパティーと値の組を使用して、たとえ DRS 複製ドメインがそのキャッシュに割り当てられていないとしても、そのキャッシュが複製されたキャッシュであると強制的に示すことです。

   ```
   com.ibm.ws.cache.CacheConfig.enableCacheReplication = "true"
   ```

3. 動的キャッシュ・サービスのトポロジーを構成します。

   eXtreme Scale 動的キャッシュ・プロバイダーで唯一必須の構成パラメーターが、キャッシュ・トポロジーです。動的キャッシュ・サービスのカスタム・プロパティーとして、以下の変数を設定する必要があります。

   ```
   com.ibm.websphere.xs.dynacache.topology
   ```

   このプロパティーには次の 3 つの値が設定可能です。

   - embedded
   - embedded_partitioned
   - remote

   許可されている値のいずれかを使用する必要があります。

4. 動的キャッシュ・サービスの初期コンテナー数を構成します。

   初期コンテナーの数を構成することで、組み込み区画化トポロジーを使用するキャッシュのパフォーマンスを最大化することができます。 WebSphere Application Server Java 仮想マシンで、システム・プロパティーとして以下の変数を設定する必要があります。

   ```
   com.ibm.websphere.xs.dynacache.num_initial_containers
   ```

   この構成プロパティーの推奨値は、この分散キャッシュ・インスタンスにアクセスする WebSphere Application Server インスタンスの合計数に等しいか、わずかに少ない整数です。例えば、動的キャッシュ・サービスがグリッド・メンバー間で共有される場合、この変数はグリッド・メンバーの数に設定すべきです。
5. eXtreme Scale カタログ・サービス・グリッドを構成します。

   分散キャッシュ・インスタンスの動的キャッシュ・プロバイダーとして eXtreme Scale を使用している場合、 eXtreme Scale カタログ・サービス・グリッドを構成する必要があります。

   単一のカタログ・サービス・グリッドで、 eXtreme Scale でサポートされる複数の動的キャッシュ・サービス・プロバイダーに対応することができます。

   カタログ・サービスは、 WebSphere Application Server プロセスの内部または外部で実行されます。

   カタログ・サービス・グリッドを実行している場合、カタログ・サービス・エンドポイントの **catalog.services.cluster** カスタム・プロパティーを設定する必要があります。
6. カスタム・キー・オブジェクトを構成します。

   キーとしてカスタム・オブジェクトを使用している場合、オブジェクトは Serializable インターフェースまたは Externalizable インターフェースを実装している必要があります。組み込みトポロジーあるいは組み込み区画化トポロジーを使用している場合、まるでデフォルトの動的キャッシュ・プロバイダーで使用しているかのようにオブジェクトを WebSphere 共有ライブラリー・パスに配置する必要があります。詳細は、WebSphere Application Server Network Deployment インフォメーション・センターの 動的キャッシュ用の DistributedMap および DistributedObjectCache インターフェースの使用を参照してください。

   リモート・トポロジーを使用している場合は、カスタム・キー・オブジェクトをスタンドアロン eXtreme Scale コンテナーの CLASSPATH に配置する必要があります。
7. eXtreme Scale コンテナー・サーバーを構成します。

   キャッシュ・データは、WebSphere eXtreme Scale コンテナーに保管されます。コンテナーは、 WebSphere Application Server プロセスの内部または外部で実行できます。キャッシュ・インスタンスに組み込み・トポロジーまたは組み込み区画化トポロジーを使用している場合、eXtreme Scale プロバイダーは、WebSphere プロセスの内部に自動的にコンテナーを作成します。これらのトポロジーの場合、それ以上の構成は必要ありません。

リモート・トポロジーを使用している場合は、キャッシュ・インスタンスにアクセスする WebSphere Application Server インスタンスが開始する前に、スタンドアロン eXtreme Scale コンテナーを開始しておく必要があります。 特定の動的キャッシュ・サービスのコンテナーがすべて、同じカタログ・サービスのエンドポイントをポイントしていることを確認してください。

スタンドアロン eXtreme Scale 動的キャッシュ・プロバイダー・コンテナーの XML 構成ファイルは、WebSphere Application Server 上のインストールの場合は `<install_root>/customLibraries/ObjectGrid/dynacache/etc` ディレクトリーか、あるいは、スタンドアロン・インストールの場合は `<install_root>/ObjectGrid/dynacache/etc` ディレクトリーに配置されます。このファイルの名前は、`dynacache-remote-objectgrid.xml` および `dynacache-remote-definition.xml` です。これらのファイルのコピーを作成して編集し、eXtreme Scale 動的キャッシュ・プロバイダーのスタンドアロン・コンテナーを起動する際に使用できるようにします。 **`dynacache-remote-deployment.xml`** ファイルの **numIntitialContainers** パラメーターは、実行中のコンテナー・プロセスの数に応じて設定する必要があります。

注: コンテナー・プロセスのセットには、リモート・トポロジーを使用するように構成されたすべての動的キャッシュ・インスタンスを処理するために十分な空きメモリーがあることが必要です。 **catalog.services.cluster** の同じ値または同等の値を共有する WebSphere Application Server プロセスはすべて、同一セットのスタンドアロン・コンテナーを使用する必要があります。コンテナーの数と、そのコンテナーが入るマシンの数は適切なサイズに設定される必要があります。詳細情報については、『キャパシティー・プランニングと高可用性』を参照してください。

次のコードは、eXtreme Scale 動的キャッシュ・プロバイダーのスタンドアロン・コンテナーを起動する UNIX® コマンド行入力の例を示しています。

```
startOgServer.sh container1 -objectGridFile ../dynacache/etc/dynacache-remote-objectgrid.xml -deploymentPolicyFile ../dynacache/etc/dynacache-remote-deployment.xml -catalogServiceEndpoints MyServer1.company.com:2809
```

キャパシティー・プランニングと高可用性

WebSphere Application Server にデプロイされた Java EE アプリケーションでは、動的キャッシュ API を使用できます。動的キャッシュは、ビジネス・データや生成された HTML をキャッシュに入れるために、または、データ複製サービス (DRS) を使用してセル内のキャッシュ・データを同期化するために利用できます。

概説

WebSphere eXtreme Scale 動的キャッシュ・プロバイダーで作成されたすべての動的キャッシュ・インスタンスは、デフォルトで、高い可用性を持ちます。高可用性のレベルおよびメモリー・コストは、使用されるトポロジーによって変わります。

組み込みトポロジーを使用している場合、キャッシュ・サイズは、1 つのサーバー・プロセス内の空きメモリーの量に制限され、各サーバー・プロセスはキャッシュの完全コピーを保管します。1 つのサーバー・プロセスが実行し続けている限

り、キャッシュは存続します。キャッシュ・データが失われるのは、キャッシュにアクセスするすべてのサーバーがシャットダウンされた場合のみです。

組み込み区画化トポロジーを使用するキャッシングの場合、キャッシュ・サイズの上限は、すべてのサーバー・プロセス内にある空きスペースの総計です。デフォルトでは、eXtreme Scale 動的キャッシュ・プロバイダーは各プライマリー断片ごとに 1 つの複製を使用します。したがって、各キャッシュ・データは 2 回ずつ保管されます。

組み込み区画化キャッシュの容量を判定するには、以下の式 A を使用してください。

式 A

F * C / (1 + R) = M

各部の意味は、次のとおりです。

- F = コンテナー・プロセス当たりの空きメモリー
- C = コンテナーの数
- R = 複製の数
- M = キャッシュの合計サイズ

各プロセスに 256 MB の使用可能なスペースがあり、サーバー・プロセスが合計 4 つある WebSphere Network Deployment グリッドの場合、それらの全サーバーにまたがるキャッシュ・インスタンスは 512 メガバイトまでのデータを保管できます。このモードでは、キャッシュは、1 つのサーバーが破損しても、データを失うことなく存続できます。また、最大 2 つのサーバーが順次シャットダウンしても、データを失うことはありません。この例では、上の式は次のようになります。

256mb * 4 コンテナー/ (1 プライマリー + 1 複製) = 512mb

リモート・トポロジーを使用するキャッシュのサイジング特性は、組み込み区画化トポロジーを使用するキャッシュと似ていますが、すべての eXtreme Scale コンテナー・プロセス内の使用可能なスペースの総量に制限されます。

リモート・トポロジーでは、複製の数を増やすことによって、メモリーのオーバーヘッドが余分にかかる代わりにアベイラビリティーのレベルを上げることが可能です。これは大部分の動的キャッシュ・アプリケーションでは不必要ですが、dynacache-remote-deployment.xml ファイルを編集して複製の数を増やすことができます。

以下の式 B と C を使用して、キャッシュの高可用性のために複製を追加することの影響を判定できます。

式 B

N = Minimum(T -1, R)

各部の意味は、次のとおりです。

- N = 同時に破損してもかまわないプロセスの数

- T = コンテナーの総数
- R = 複製の総数

式 C

Ceiling(T/ (1+N)) = m

各部の意味は、次のとおりです。

- T = コンテナーの総数
- N = 複製の総数
- m = キャッシュ・データをサポートするのに必要な最小コンテナー数

動的キャッシュ・プロバイダーでのパフォーマンス・チューニングについては、 75 ページの『動的キャッシュ・プロバイダーのチューニング』を参照してください。

キャッシュのサイズ見積もり

WebSphere eXtreme Scale 動的キャッシュ・プロバイダーを使用するアプリケーションをデプロイする前に、前のセクションに記述されている一般的な規則に、実動システムの環境データを組み合わせて検討する必要があります。まず最初に確定する必要がある数値は、コンテナー・プロセスの総数と、キャッシュ・データを保持するための各プロセス内の使用可能メモリーの量です。組み込みトポロジーを使用している場合、キャッシュ・コンテナーは WebSphere Application Server プロセスの内部の同一場所に配置され、キャッシュを共有する各サーバーごとにコンテナーが 1 つずつある状態になります。 キャッシングを使用可能にしていないアプリケーションと WebSphere Application Server のメモリー・オーバーヘッドを判定することが、プロセス内で使用可能なスペース量を計算する最良の方法です。これは、詳細ガーベッジ・コレクション・データを分析することで行えます。リモート・トポロジーを使用している場合、この情報は、新しく開始され、キャッシュ・データがまだ設定されたことのないスタンドアロン・コンテナーの、詳細ガーベッジ・コレクション出力を見れば分かります。 キャッシュ・データ用に使用可能な、プロセス当たりのスペース量を計算する際には、ガーベッジ・コレクション用にいくらかのヒープ・スペースを確保しておくことにも注意が必要です。コンテナー (WebSphere Application Server またはスタンドアロン) のオーバーヘッドに、キャッシュ用に予約されたサイズを加えた結果は、合計ヒープの 70 % 以下になっているべきです。

この情報を収集できたら、前述の式 A に値を挿入して、区画化されたキャッシュの最大サイズを判定してください。最大サイズが判明したら、次のステップは、サポートできるキャッシュ・エントリー総数を判定することです。これには、キャッシュ・エントリー当たりの平均サイズを決定することが必要です。これを行う簡単な方法は、カスタマー・オブジェクトのサイズに 10% 追加することです。動的キャッシュを使用している場合のキャッシュ・エントリーのサイズ見積もりについて詳しくは、動的キャッシュおよびデータ複製サービスのチューニング・ガイドを参照してください。

圧縮が有効にされている場合、圧縮はカスタマー・オブジェクトのサイズには影響しますが、キャッシング・システムのオーバーヘッドには影響しません。圧縮を使用している場合のキャッシュ・オブジェクトのサイズを判定するには、以下の式を使用します。

S = O * C + O * 0.10

各部の意味は、次のとおりです。

- S = キャッシュ・オブジェクトの平均サイズ
- O = 圧縮されていないカスタマー・オブジェクトの平均サイズ
- C = 分数で表した圧縮率

つまり、2 を 1 にする場合の圧縮率は 1/2 = 0.50 です。これは小さいほど良い値です。保管されるオブジェクトが通常の POJO で、大部分がプリミティブ型の場合、圧縮率を 0.60 から 0.70 に想定してください。キャッシュされるオブジェクトが、サーブレット、JSP、または WebServices オブジェクトの場合、圧縮率を判定する最適の方法は、代表的なサンプルを ZIP 圧縮ユーティリティーで圧縮することです。これが不可能な場合、このタイプのデータの一般的な圧縮率は 0.2 から 0.35 です。

次に、この情報を使用して、サポートできるキャッシュ・エントリーの総数を判定します。以下の式 D を使用してください。

式 D

T = S / A

各部の意味は、次のとおりです。

- T = キャッシュ・エントリーの総数
- S = 式 A を使用して算出され、キャッシュ・データ用に使用可能な合計サイズ
- A = 各キャッシュ・エントリーの平均サイズ

最後に、動的キャッシュ・インスタンスにキャッシュ・サイズを設定して、この制限を強制する必要があります。WebSphere eXtreme Scale 動的キャッシュ・プロバイダーは、この点で、デフォルトの動的キャッシュ・プロバイダーと異なります。以下の式を使用して、動的キャッシュ・インスタンスのキャッシュ・サイズに設定する値を判定してください。以下の式 E を使用してください。

式 E

Cs = Ts / Np

各部の意味は、次のとおりです。

- Ts = キャッシュの合計目標サイズ
- Cs = 動的キャッシュ・インスタンスに設定するキャッシュ・サイズ設定値
- Np = 区画の数。デフォルトは 47 です。

キャッシュ・インスタンスを共有する各サーバーで、動的キャッシュ・インスタンスのサイズを、式 E で計算した値に設定してください。

動的キャッシュ・プロバイダーのチューニング

WebSphere eXtreme Scale 動的キャッシュ・プロバイダーは、パフォーマンスのチューニングのために以下の構成パラメーターをサポートします。

このタスクについて

- **com.ibm.websphere.xs.dynacache.ignore_value_in_change_event**: 動的キャッシュ・プロバイダーで変更イベント・リスナーを登録し、ChangeEvent インスタンスを生成すると、値が ChangeEvent 内に入るようにするためのキャッシュ・エントリーのデシリアライズに関連したオーバーヘッドがあります。キャッシュ・インスタンスでこのオプション・パラメーターを `true` に設定すると、ChangeEvents の生成時にキャッシュ・エントリーのデシリアライゼーションがスキップされます。戻される値は、除去操作の場合の NULL か、または、シリアライズ形式でオブジェクトを含むバイト配列のいずれかになります。InvalidationEvent インスタンスには、同じようなパフォーマンスに不利な条件があり、これは com.ibm.ws.cache.CacheConfig.ignoreValueInInvalidationEvent を `true` に設定することで回避できます。
- **com.ibm.websphere.xs.dynacache.disable_recursive_invalidate**: 動的キャッシュの無効化は、デフォルトで再帰的に行われます。WebSphere eXtreme Scale の動的キャッシュ・プロバイダーを使用すると、組み込みの区画化リモート・トポロジーにおける再帰的な無効化に伴い、さらなるオーバーヘッドが発生します。キャッシュで再帰的な無効化の必要がない場合は、このパラメーターを `true` に設定してオーバーヘッドを回避してください。
- **com.ibm.websphere.xs.dynacache.enable_compression**: eXtreme Scale 動的キャッシュ・プロバイダーは、キャッシュの密度を上げるために、メモリー内のキャッシュ・エントリーを圧縮できます。これにより、読み取り操作と書き込み操作で追加のオーバーヘッドが生じますが、サーブレット・キャッシングなどのアプリケーションにおいては、相当量のメモリーを節約できます。

第 4 章 スケーラビリティー

WebSphere eXtreme Scale は、区画に分割されたデータの使用を通じて拡張可能です。そして各コンテナーは互いに独立しているため、何千というコンテナーに拡張できます。

WebSphere eXtreme Scale は、データ・セットを、プロセス間で (実行時にはマシン間でさえ) 移動できる異なる区画に分割します。例えば、4 つのサーバーのデプロイメントから始め、その後キャッシュに対する要求が増えるに従って 10 個のサーバーのデプロイメントに拡張することができます。ちょうど、垂直スケーラビリティーに備えて物理マシンや処理装置をさらに追加できるように、eXtreme Scale の弾力性のあるスケーリング能力を区画化によって水平方向に拡張することができます。これは、データ・グリッドである eXtreme Scale に対立するものとしてのメモリー内データベース (IMDB) とは別の大きな相違点です (IMDB は垂直方向にしか拡張できないため)。

さらに、WebSphere eXtreme Scale により、一連の API を使用して、区画化され、さらに必要に応じて分散されたデータにトランザクション・アクセスをすることができます。パフォーマンスに関しては、キャッシュとの対話のために行う選択が、キャッシュを可用性の面で管理する機能と同じくらい重要です。

注: スケーラビリティーは、コンテナー同士が互いに通信しているときは使用不可です。可用性管理、つまりコア・グループ化のプロトコルは、$O(N^2)$ ハートビートおよびビュー保守アルゴリズムですが、コア・グループ・メンバーの数を 20 個に維持することにより、このプロトコルの負担は軽減されます。複製は断片間でのみ対等です。

区画化

区画化は、 WebSphere eXtreme Scale がアプリケーションをスケールアウトするのに使用するメカニズムです。区画化により、アプリケーション状態は各パーツがいくつかの完全なインスタンス・データを含むパーツに分離されます。区画化は、新磁気ディスク制御機構 (RAID) ストライピングと同様のものではありませんが、各インスタンスをすべてのストライプ間でスライスします。各区画により、個別エントリーの完全なデータがホスティングされます。区画化は、非常に有効なスケーリング手段ですが、すべてのアプリケーションに適用できるわけではありません。複数の大規模なデータ集合にわたってトランザクションの保証を必要とするアプリケーションをスケールアウトすることはできず、効果的に区画化することもできません。したがって、eXtreme Scale は、区画をまたがる 2 フェーズ・コミットを現在のところサポートしていません。

重要: 区画数を選択する際は慎重に行ってください。デプロイメント・ポリシーで定義される区画の数は、アプリケーションが拡張できるコンテナーの数に直接影響を与えます。各区画は、プライマリー断片および構成済みの数の複製断片から構成されます。公式 `(Number_Partitions*(1 + Number_Replicas))` は、単一のアプリケーションを拡張するのに使用できるコンテナーの数を表します。

分散クライアント

WebSphere eXtreme Scale クライアント・プロトコルは、非常に多くのクライアントをサポートします。接続を複数のコンテナー間に広げることができるため、すべてのクライアントは常にすべての区画の対象となるわけではないと仮定すれば、区画化ストラテジーは役に立ちます。クライアントは区画に直接接続されるため、待ち時間は 1 つの転送接続に制限されます。

区画化

Java 仮想マシン (JVM) に大量のデータを保管するには、区画化を使用します。データを区画化するには、アプリケーション指定のスキームを使用してデータを分割します。

区画の使用

1 つのグリッドは多数の区画、つまり必要ならば数千個の区画を持つことができます。グリッドは、区画の数に区画ごとの断片の数を掛けた結果の大きさまで拡張できます。例えば、16 個の区画があり、各区画にはプライマリーと複製がそれぞれ 1 つずつ、つまり 2 つの断片があるとすれば、潜在的には 32 個の Java 仮想マシンまで拡張できることになります。この場合、JVM ごとに 1 つの断片が定義されます。使用する可能性が高い Java 仮想マシンの予定数に基づいて、適切な区画数を選択する必要があります。断片が 1 つ増すごとにシステムのためのプロセッサーおよびメモリーの使用量が増加します。システムは、使用可能な Java 仮想マシンの数に応じてこのオーバーヘッドを処理するように拡張される設計になっています。

アプリケーションが 4 つのコンテナー Java 仮想マシンのグリッドで実行される場合は、アプリケーションが数千もの区画を使用しないようにしてください。アプリケーションは、それぞれのコンテナー JVM について、適切な数の断片を持つよう構成する必要があります。例えば、2 つの断片を持つ 2000 の区画が 4 つのコンテナー Java 仮想マシンで稼働するという構成は適切ではありません。このような構成では、4 つのコンテナー Java 仮想マシンに 4000 個の断片が配置されるか、またはコンテナー JVM ごとに 1000 個の断片が配置されることになります。

予定される各コンテナー JVM の断片を 10 未満に構成することをお勧めします。それでもなお、この構成では、コンテナー JVM ごとの断片の数を適切に保ちながら、初期構成の 10 倍の弾力性のある拡張を行える可能性があります。

次のような拡張例を検討してください。現在、6 台のコンピューターがあり、コンピューターごとに 2 つのコンテナー Java 仮想マシンがあるとします。今後 3 年間で、コンピューターを 20 台まで増やす予定です。コンピューターが 20 台ある場合、コンテナー Java 仮想マシンは 40 になりますが、余裕を持って 60 を選択することにします。コンテナー JVM ごとに 4 つの断片が必要です。60 のコンテナーでは、断片は合計 240 になります。区画ごとにプライマリーとレプリカがあるとすると、120 の区画が必要になります。この例は、240 を 12 のコンテナー Java 仮想マシンで除算すること、つまり後でコンピューターを 20 台まで拡張できるようにするため、初期デプロイメント時にコンテナー JVM ごとに 20 の断片を想定することを示しています。

エンティティーおよび区画化

エンティティー・マネージャー・エンティティーは、サーバーのエンティティーを処理するクライアント向けに最適化されています。マップ・セットに対するサーバーのエンティティー・スキーマで、単一のルート・エンティティーを指定できます。クライアントは、このルート・エンティティーを介してすべてのエンティティーにアクセスする必要があります。それで、エンティティー・マネージャーは、同一区画のそのルートから関連エンティティーを検出することができ、関連マップには共通キーは必要ありません。ルート・エンティティーは単一区画とのアフィニティーを確立します。この区画は、アフィニティーの確立後、トランザクション内のすべてのエンティティーの取り出しに使用されます。このアフィニティーは、関連マップが共通キーを必要としないため、メモリーを節約できます。ルート・エンティティーは、以下の例に示すような変更したエンティティー・アノテーションで指定する必要があります。

```
@Entity(schemaRoot=true)
```

このエンティティーを使用してオブジェクト・グラフのルートを検出することができます。すべての子エンティティーはルートと同一の区画にあると仮定されます。このグラフ内の子エンティティーへのアクセスは、ルート・エンティティーのクライアントからのみ可能です。区画化環境では、eXtreme Scale クライアントを使用してサーバーと通信するときに、常にルート・エンティティーが必要です。クライアントごとに定義できるルート・エンティティー・タイプは、1 つのみです。Extreme Transaction Processing (XTP) スタイルの ObjectGrid を使用している場合は、区画とのすべての通信が、クライアントおよびサーバー機構ではなく、直接のローカル・アクセスによって実行されるため、ルート・エンティティーは必要ありません。

配置と区画

WebSphere eXtreme Scale には 2 つの配置ストラテジー (固定区画配置ストラテジーとコンテナーごとの配置ストラテジー) があります。配置ストラテジーは区画の配置とデータの分散に影響があります。

固定区画配置

配置ストラテジーはデプロイメント・ポリシー XML ファイルで設定することができます。デフォルトの配置ストラテジーは固定区画配置で、これは FIXED_PARTITION 設定で使用可能になります。使用可能なコンテナーに配置されるプライマリー断片の数と numberOfPartitions エレメントで構成した区画の数が等しくなります。複製を構成した場合は、配置される断片の最小合計数は次の式によって定義されます。((1 プライマリー断片 + 同期断片の最小数) * 定義されている区画の数)。配置される断片の最大合計数は次の式によって定義されます。 ((1 プライマリー断片 + 同期断片の最大数 + 非同期断片の最大数) * 区画数)。 WebSphere eXtreme Scale のデプロイメントにより、これらの断片は使用可能なコンテナーに拡散されます。各マップのキーは、定義した合計区画数に基づいて、割り当てられた区画にハッシュされます。フェイルオーバーやサーバー変更のために区画が移動された場合でも、これらのキーは同じ区画にハッシュされます。

例えば、numberPartitions 値が 6 で、MapSet1 の minSync 値が 1 である場合は、6 個の区画のそれぞれが同期複製を必要とするため、そのマップ・セットの合計断片数は 12 となります。コンテナーが 3 つ開始されると、WebSphere eXtreme Scale

は MapSet1 用にコンテナーごとに 4 個の断片を配置します。

コンテナーごとの配置

代替配置ストラテジーはコンテナーごとの配置です。これは、デプロイメント XML ファイルのマップ・セット・エレメントにある placementStrategy に対する PER_CONTAINER 設定で使用可能になります。このストラテジーでは、各新規コンテナーに配置されたプライマリー断片の数と構成した区画の数 (*P*) が等しくなります。 WebSphere eXtreme Scale デプロイメント環境では、残っているコンテナーごとに各区画の *P* 個の複製が配置されます。コンテナーごとの配置を使用していると、numInitialContainers 設定は無視されます。区画はコンテナーの増大につれて大きくなります。このストラテジーでは、マップのキーは特定の区画に固定されません。クライアントはある区画に経路指定して、ランダム・プライマリーを使用します。再度キーの検出に使用される同じセッションに再接続したいクライアントがあると、そのクライアントはセッション・ハンドルを使用しなければなりません。

詳しくは、「*プログラミング・ガイド*」に記載されている経路指定のための SessionHandle の使用に関するトピックを参照してください。

フェイルオーバーまたはサーバーが停止された場合、WebSphere eXtreme Scale 環境はプライマリー断片を (まだそこにデータが入っていれば) コンテナーごとの配置ストラテジーに従って移動します。空の断片は廃棄されます。コンテナーごとのストラテジーでは、すべてのコンテナーについて新しいプライマリー断片が配置されるため、古いプライマリー断片は保存されません。

PartitionableKey インターフェース

eXtreme Scale 構成が固定区画配置ストラテジーを使用しているとき、この構成は区画のキーのハッシュに応じて値の挿入、取得、更新、または除去を行います。このキーで hashCode メソッドが呼び出され、カスタム・キーが作成される場合は、hashCode メソッドが明確に定義されていなければなりません。ただし、PartitionableKey インターフェースを使用するもう 1 つのオプションがあります。PartitionableKey インターフェースがあれば、キー以外のオブジェクトを使用して区画にハッシュすることができます。

PartitionableKey インターフェースは、複数のマップが存在し、かつコミットしたデータが関連付けられ、したがって同じ区画に配置されなければならないような状況で使用することができます。WebSphere eXtreme Scale は、複数のマップ・トランザクションが複数の区画にまたがる場合は、これらのトランザクションがコミットされないようにするため、2 フェーズ・コミットをサポートしません。同じマップ・セット内の異なるマップにあるキーについて PartitionableKey が同じ区画にハッシュする場合は、トランザクションをまとめてコミットすることができます。

また、キーのグループを同じ区画に配置する必要があるが、必ずしも単一トランザクションのときでない場合にも PartitionableKey インターフェースを使用することができます。ロケーション、部門、ドメイン・タイプ、またはその他のタイプの ID に基づいてキーをハッシュする必要がある場合は、子キーに親 PartitionableKey を与えることができます。

例えば、従業員はその所属する部門と同じ区画にハッシュする必要があります。各従業員キーは部門マップに属する PartitionableKey オブジェクトを持ちます。そうすると、従業員と部門の両方が同じ区画にハッシュされます。

PartitionableKey インターフェースには ibmGetPartition というメソッドが 1 つあります。このメソッドから戻されたオブジェクトは hashCode メソッドを実装する必要があります。代替 hashCode を使用したために戻された結果は区画のキーを経路指定するために使用されます。

単一区画トランザクションおよびクロスグリッド区画トランザクション

WebSphere eXtreme Scale とリレーショナル・データベースやメモリー内データベースなどの従来のデータ・ストレージ・ソリューションとの間の主な相違は、キャッシュの直線的な増加を可能にする区画化を使用することにあります。考慮すべき重要なトランザクションのタイプに、単一区間トランザクションと各区画 (クロスグリッド) トランザクションがあります。

一般的に、以下で説明するようにキャッシュとの対話は、単一区間トランザクションまたはクロスグリッド・トランザクションとして分類できます。

単一区間トランザクション

単一区間トランザクションは、WebSphere eXtreme Scale によってホストされるキャッシュと対話する場合に適した方法です。単一区画に制限されている場合のトランザクションは、デフォルトで単一の Java 仮想マシン、すなわち単一のサーバー・コンピューターに制限されます。サーバーは、こうしたトランザクションを毎秒 *M* 個実行することができるので、*N* 台のコンピューターがある場合は、毎秒 `M*N` 個のトランザクションを実行できます。ビジネスが拡大し、毎秒こうしたトランザクションを 2 倍の数実行する必要性が出てきた場合、さらにコンピューターを購入して *N* を 2 倍にすることができます。これにより、アプリケーションを変更したり、ハードウェアをアップグレードしたり、さらにはアプリケーションをオフラインにしたりすることさえなく、容量ニーズを満たすことができます。

単一区間トランザクションは、キャッシュの拡大をかなり大幅に行えるようになっているほか、キャッシュの可用性を最大限に引き出します。各トランザクションは、1 台のコンピューターのみに依存します。他の (`N-1`) 台のコンピューターのいずれかに障害が起こっても、このトランザクションの成否および応答時間には影響しません。したがって、100 台のコンピューター (サーバー) を稼働していて、そのうち 1 台に障害が生じても、そのサーバーに障害が生じた時点で進行中であった 1 パーセントのトランザクションしかロールバックされません。サーバーの障害後、WebSphere eXtreme Scale は、障害を起こしたサーバーによってホストされる区画を他の 99 台のコンピューターに再配置します。これは短時間の処理であり、この操作の完了前であれば、この時間内に他の 99 台のコンピューターはトランザクションを完了できます。再配置される区画に関係するトランザクションしか、ブロックされません。フェイルオーバー・プロセスが完了すると、キャッシュは、元のスループット量の 99 パーセントで完全に操作可能状態で引き続き稼働できるようになります。障害のあるサーバーが交換されて、グリッドに戻されると、キャッシュは 100 パーセントのスループット量に戻ります。

クロスグリッド・トランザクション

パフォーマンス、可用性、およびスケーラビリティーの面では、クロスグリッド・トランザクションは、単一区間トランザクションの対極にあります。クロスグリッド・トランザクションは、すべての区画、つまり構成内のすべてのコンピューターにアクセスします。グリッド内の各コンピューターは、ある種のデータを検索して、その結果を戻すように求められます。トランザクションは、すべてのコンピューターが応答するまで完了できません。したがってグリッド全体のスループットは、最低速のコンピューターによって制限されます。コンピューターを追加しても、最低速のコンピューターの処理速度が増すわけではなく、キャッシュのスループットは改善しません。

クロスグリッド・トランザクションは、可用性についても同じ影響を及ぼします。先の例を拡大すると、100 台のサーバーが稼働していて、そのうち 1 台に障害が生じたとすると、そのサーバーに障害が生じた時点で進行中であったトランザクションの 100 パーセントがロールバックされます。サーバーの障害後、WebSphere eXtreme Scale は、このサーバーによってホストされる区画を他の 99 台のコンピューターに再配置する処理を開始します。この時間の間、フェイルオーバー・プロセスが完了するまでは、グリッドは、該当するトランザクションをどれも処理できなくなります。フェイルオーバー・プロセスが完了すると、キャッシュは、続行できるようになりますが、容量は減少します。グリッド内の各コンピューターが 10 個の区画をサービスしていた場合、残りの 99 台のコンピューターのうち 10 台は、フェイルオーバー・プロセスの一部として少なくとも 1 つの余分の区画を受け取ることになります。余分の区画を 1 つ追加すると、該当コンピューターのワークロードは 10 パーセント以上増えます。グリッドのスループットは、クロスグリッド・トランザクション内の最低速のコンピューターのスループットに制限されるので、平均して、スループットは 10 パーセント減少します。

WebSphere eXtreme Scale のような高可用性の分散オブジェクト・キャッシュでのスケールアウトの場合は、単一区間トランザクションのほうがクロスグリッド・トランザクションよりも適しています。こうした種類のシステムのパフォーマンスを最大限にするには、従来のリレーショナルの方法論とは異なる手法を使用する必要がありますが、クロスグリッド・トランザクションをスケーラブルな単一区間トランザクションに変えることができます。

スケーラブル・データ・モデルのビルドのベスト・プラクティス

WebSphere eXtreme Scale のような製品でのスケーラブル・アプリケーションをビルドする際のベスト・プラクティスには、基礎プリンシパルと実装ヒントという 2 つのカテゴリーがあります。基礎プリンシパルは、データ自体の設計に取り込む必要がある中心的なアイデアです。こうした原則を守らないアプリケーションは、たとえそのメインライン・トランザクションに対しても、適切に拡大できる可能性が低くなります。実装ヒントは、スケーラブル・データ・モデルの本来は一般プリンシパルに従って適切に設計されたアプリケーション内の問題のあるトランザクションに適用されます。

基本原則

スケーラビリティーを最適化する重要な手段の一部として、基本的な概念または原則を考慮する必要があります。

正規化に代わる重複

WebSphere eXtreme Scale のような製品の場合、その製品が多数のコンピューター間でデータを展開できるように設計されているということを念頭に入れておくことが重要です。ほとんどまたはすべてのトランザクションを単一区画で完全なものとすることが目標である場合は、データ・モデル設計で、トランザクションが必要とする可能性のあるすべてのデータがその区画に存在するようにする必要があります。ほとんどの場合、データを複製することによってのみ、この目標を実現できます。

例えば、メッセージ・ボードのようなアプリケーションを考えてみます。メッセージ・ボードの 2 つの極めて重要なトランザクションとして、一定のユーザーからのすべてのポスト・メッセージを表示するものと、特定のトピックに関するすべてのポスト・メッセージを表示するものがあります。まずこうしたトランザクションがユーザー・レコード、トピック・レコード、さらに実際のテキストが含まれるポスト・レコードを含む正規化されたデータ・モデルをどのように扱うかを考えてみます。ポスト・メッセージがユーザー・レコードによって区画に分割されている場合、トピックを表示することは、クロスグリッド・トランザクションとなります。またその逆もいえます。トピックおよびユーザーは、多対多の関係を持っているので一緒に区画に分割することはできません。

このメッセージ・ボードの拡大を行う最善の策は、ポスト・メッセージを複製して、トピック・レコードを持つコピーを 1 つ、ユーザー・レコードを持つコピーを 1 つ保存することです。この結果、ユーザーからのポスト・メッセージを表示することは単一区間トランザクションとなり、トピックに関するポスト・メッセージを表示することは単一区間トランザクションとなり、ポスト・メッセージを更新または削除することは、2 区画トランザクションとなります。グリッド内のコンピューターの数が増えるにつれ、これら 3 つのトランザクションがすべて直線的に拡大します。

リソースに代わるスケーラビリティー

非正規化されたデータ・モデルを考慮する場合に克服すべき最大の障害は、こうしたモデルがリソースに与える影響です。ある種のデータのコピーを 2 つ、3 つ、またはそれ以上保持すると、利用される資源が多すぎるように見える場合があります。こうしたシナリオに直面したら、ハードウェア・リソースが年々低価格になっているという事実を思い出してください。第 2 に (さらに重要)、WebSphere eXtreme Scaleは、追加資源のデプロイに関連した隠れコストを削減します。

メガバイトやプロセッサーといったコンピューター関連ではなく、コスト関連でリソースを測定してください。正規化された関係データを扱うデータ・ストアは、一般的に同じコンピューターに存在する必要があります。こうしたコロケーションの必要性から、いくつか小型コンピューターを購入するのではなく、1 台の大型の企業向けコンピューターを購入したほうがよいという結果が導かれます。ただし企業向けハードウェアの場合、通常では、毎秒 100 万のトランザクションの実行が可能な 1 台のコンピューターを使用するほうが、それぞれ毎秒 10 万のトランザクションの実行が可能な 10 台のコンピューターを結合した場合よりコストがかなりかかることは珍しいことではありません。

リソースを追加する際のビジネス・コストも存在します。ビジネスが成長していくと、結果的に容量不足となります。容量不足となると、より大型の高速コンピューターに移行する際にシャットダウンが必要になるか、切り替え可能な第 2 の実稼働環境の作成が必要になります。いずれにせよ、ビジネス損失が発生するか、遷移期間にほぼ 2 倍の容量の維持が必要になるという形で追加コストが発生します。

WebSphere eXtreme Scale を使用すると、容量追加のためにアプリケーションをシャットダウンする必要がなくなります。ビジネスで翌年に 10 パーセントの追加容量が必要になることが見込まれた場合、グリッド内のコンピューターの数を 10 パーセント増加します。このパーセンテージ分の増加の際に、アプリケーション・ダウン時間もなく、超過容量の購入の必要もありません。

データ形式変更の防止

WebSphere eXtreme Scale を使用している場合、データは、ビジネス・ロジックで直接消費可能な形式で保管されます。データをよりプリミティブな形式に分解することには、コストがかかります。データの書き込みおよび読み取り時に、変換を実行する必要があります。リレーショナル・データベースを使用する場合、データが最終的にディスクにパーシストされることがごく頻繁に行われるため、この変換は必要に応じて実行されますが、WebSphere eXtreme Scale を使用すると、こうした変換を実行する必要がなくなります。データは大部分メモリーに保管されるため、アプリケーションが必要とするそのままの形式で保管することができます。

この単純な規則に従うと、最初の原則に従ってデータを非正規化するのに役立ちます。ビジネス・データ用の最も一般的なタイプの変換は、正規化されたデータをアプリケーションのニーズに合う結果セットに変えるために必要な JOIN 演算です。データを正しい形式で保管すると、暗黙的にこうした JOIN 演算の実行が避けられ、非正規化されたデータ・モデルが作成されます。

未結合照会の除去

いくらデータを適切に構成しても、未結合照会は正しく拡張されません。例えば、値でソートされたすべての項目のリストを要求するようなトランザクションは使用しないでください。こうしたトランザクションは、はじめのうち合計項目数が 1000 であると、機能するかもしれませんが、合計項目数が 1000 万に達すると、トランザクションは 1000 万すべての項目を戻します。このトランザクションを実行した場合、最も考えられる 2 つの結果は、トランザクションのタイムアウトになるか、クライアントにメモリー不足エラーが発生するかのいずれかです。

最善のオプションは、上位 10 または 20 の項目だけが戻されるように、ビジネス・ロジックを変更することです。このロジック変更によって、キャッシュ内の項目数に関係なく、トランザクションのサイズが管理可能な程度に保たれます。

スキーマの定義

データの正規化の主な利点は、データベース・システムが状況の背後にあるデータの整合性を考慮できることです。データがスケーラビリティーのために非正規化されると、この自動データ整合性管理は存在しなくなります。デ

ータの整合性を保証するために、アプリケーション層で機能できるか、分散グリッドに対するプラグインとして機能できるデータ・モデルを実装する必要があります。

メッセージ・ボードの例を考えてみます。トランザクションがトピックからポスト・メッセージを除去した場合、ユーザー・レコード上の重複するポスト・メッセージを除去する必要があります。データ・モデルがなくても、開発者は、トピックからポスト・メッセージを除去し、さらに確実にユーザー・レコードからそのポスト・メッセージを除去するアプリケーション・コードを作成することができます。ただし、仮に開発者がキャッシュと直接に対話する代わりにデータ・モデルを使用していたとしても、データ・モデル上の removePost メソッドによって、ポスト・メッセージからユーザー ID を抜き出して、ユーザー・レコードを検索し、この状況の背後にある重複ポスト・メッセージを除去することができます。

あるいは、実際の区画で実行し、トピックの変更を検出して、ユーザー・レコードを自動的に調整するリスナーを実装することができます。リスナーは、役に立ちます。区画がユーザー・レコードを持つようになった場合に、ユーザー・レコードの調整がローカルで可能になるか、ユーザー・レコードが異なる区画にあっても、トランザクションがクライアントとサーバーの間ではなく、サーバー間で実行されるためです。サーバー間のネットワーク接続のほうが、クライアントとサーバーの間のネットワーク接続よりも高速である可能性があります。

競合の防止

グローバル・カウンターを持つようなシナリオは避けてください。1 つのレコードが残りのレコードと比べて極端に多く使用されている場合は、グリッドは拡張されません。グリッドのパフォーマンスは、この特定のレコードを保持するコンピューターのパフォーマンスによって制限されています。

このような状態では、そのレコードを区画単位で管理できるように分割してみてください。例えば、分散キャッシュ内の合計項目数を戻すトランザクションを考えます。すべての挿入および除去操作で増大する単一のレコードにアクセスする代わりに、各区画のリスナーに挿入および除去操作を追跡させます。このリスナーによるトラッキングを使用すると、挿入および除去を単一区間操作とすることができます。

カウンターの読み取りはクロスグリッド操作となりますが、ほとんどの場合、それは元々クロスグリッド操作と同じく非効率的です。そのパフォーマンスがレコードをホストするコンピューターのパフォーマンスと関係しているためです。

実装ヒント

最善のスケーラビリティーを達成するには、以下のヒントも考慮してください。

逆引き索引の使用

顧客レコードが顧客 ID 番号に基づいて区画化されるような適切に非正規化されたデータ・モデルを考えます。この区画化方法は論理的な選択といえます。顧客レコードによって実行されるほぼすべてのビジネス・オペレーションは、顧客 ID 番号を使用するからです。ただし、顧客 ID 番号を使用しな

い重要なトランザクションに、ログイン・トランザクションがあります。ログインには顧客 ID 番号よりもユーザー名や電子メール・アドレスが使用されるほうが一般的です。

ログイン・シナリオの簡単な方法は、顧客レコードを見つけるためにクロスグリッド・トランザクションを使用することです。先に説明したように、この方法は拡張されません。

次のオプションとして、ユーザー名または電子メールに基づいて区画化することがあります。このオプションは、顧客 ID に基づくすべての操作がクロスグリッド・トランザクションとなるので、実用的でありません。またサイトのユーザーがユーザー名や電子メール・アドレスを変更したい場合もあります。WebSphere eXtreme Scale のような製品は、データをその不変性の維持のために区画化するのに使用される値を必要とします。

適切な解決方法として、逆引き索引を使用することができます。WebSphere eXtreme Scale を使用すると、すべてのユーザー・レコードを保持するキャッシュと同じ分散グリッドにキャッシュを作成できます。このキャッシュは、高可用性で、区画化され、しかもスケーラブルです。このキャッシュは、ユーザー名または電子メール・アドレスを顧客 ID にマップするために使用できます。このキャッシュでは、ログインは、クロスグリッド操作ではなく 2 区画操作となります。このシナリオは単一区間トランザクションほどよくはありませんが、コンピューターの数が増えるにつれ、スループットが直線的に増加します。

書き込み時の計算

平均や合計などの一般的な計算値は、作成にコストがかかることがあります。こうした操作には、通常膨大な数の項目を読み取る必要があるためです。ほとんどのアプリケーションでは、読み取りのほうが書き込みよりも一般的であるため、こうした値を書き込み時に計算し、結果をキャッシュに保管するほうが効率的です。これにより、読み取り操作は高速になり、よりスケーラブルになります。

オプション・フィールド

業務内容、自宅住所、および電話番号を保持するユーザー・レコードを考えます。これらすべてが定義されているユーザーもいれば、まったく定義されていないユーザーもいれば、一部が定義されているユーザーもいます。データが正規化されていると、ユーザー・テーブルおよび電話番号テーブルが存在することになります。一定ユーザーの電話番号は、この 2 つのテーブル間の JOIN 操作を使用して検出できます。

このレコードを非正規化する場合、データの重複は必要ありません。ほとんどのユーザーが電話番号を共有しないためです。代わりに、ユーザー・レコードで空スロットを使用できるようになっている必要があります。電話番号テーブルを使用する代わりに、各ユーザー・レコードに電話番号タイプごとに 1 つずつ 3 つの属性を追加します。この属性の追加により、JOIN 操作がなくなり、ユーザーの電話番号検索が単一区間操作となります。

多対多関係の配置

製品とその販売店を追跡するアプリケーションを考えてみます。1 つの製品が多くの店舗で販売され、1 つの店舗で多くの製品が販売されます。このア

プリケーションが 50 の大規模小売業者を追跡するものとします。各製品が最大 50 の店舗で販売され、それぞれの店舗で何千もの製品が販売されます。

各店舗エンティティー内に製品リストを保持する (配置 B) 代わりに、製品エンティティーの内部に店舗リストを保持します (配置 A)。このアプリケーションが実行する必要があるトランザクションの一部を見ると、配置 A がよりスケーラブルである理由が明らかになります。

まず更新に注目します。配置 A では、店舗の在庫から製品を除去する場合、製品エンティティーがロックされます。グリッドに 10000 の製品が保持されている場合、グリッドの 1/10000 しか更新の実行をロックする必要がありません。配置 B では、グリッドには 50 の店舗しか含まれていないので、更新を完了するには、グリッドの 1/50 をロックする必要があります。これらは両方とも単一区間操作と考えることができますが、配置 A のほうがより効率よくスケールアウトされます。

現在、配置 A による読み取りを考えていますから、トランザクションで少量のデータのみが転送されるため、製品の販売店舗の検索は拡張され、高速な単一区間トランザクションとなります。配置 B では、製品が店舗で販売されているかどうかを確認するために、各店舗エンティティーにアクセスする必要があるため、このトランザクションはクロスグリッド・トランザクションになります。これは、配置 A では多大なパフォーマンス上の利点となって現れます。

正規化されたデータによる拡張

クロスグリッド・トランザクションの正当な使用法の 1 つにデータ処理の拡張があります。グリッドに 5 台のコンピューターがあり、各コンピューターについて約 100,000 のレコード全部をソートするクロスグリッド・トランザクションがディスパッチされると、そのトランザクションは全体で 500,000 個のレコードをソートします。グリッド内の最低速のコンピューターが毎秒これらのトランザクションのうちの 10 個を実行できる場合、グリッドは全体で毎秒 5,000,000 レコードをソートできます。グリッド内のデータが 2 倍になると、各コンピューターは全体で 200,000 個のレコードをソートする必要があり、各トランザクションは全体で 1,000,000 個のレコードをソートします。このデータの増加は、最低速のコンピューターのスループットを毎秒 5 トランザクションに減少させるので、グリッドのスループットは毎秒 5 トランザクションに減少します。それでもグリッドは全体で毎秒 5,000,000 レコードをソートします。

このシナリオでは、コンピューターの数を 2 倍にすると、各コンピューターは元の 100,000 レコードのソートという負荷状態に戻るので、最低速のコンピューターは、これらのトランザクションを毎秒 10 個で処理できるようになります。グリッドのスループットは、毎秒 10 要求という同じ状態ですが、現在では各トランザクションは 1,000,000 レコードを処理するので、処理するレコードに関してはグリッドの容量は毎秒 10,000,000 レコードと 2 倍になります。

ユーザー数の増加に合わせてインターネットとスループットの規模を拡大するため、データ処理に関して両方を拡張する必要のある検索エンジンなどのアプリケーションでは、グリッド間の要求のラウンドロビンを備えた複数のグリッドを作成する必要があります。スループットを拡大する必要がある場

合、要求をサービスするために、コンピューターを追加し、別のグリッドを追加します。データ処理を拡大する必要がある場合、コンピューターを追加して、グリッド数を一定に保ちます。

第 5 章 可用性

高可用性を備えている WebSphere eXtreme Scale は、予備を提供し、障害も検出できます。

WebSphere eXtreme Scale は、 Java 仮想マシン のグリッドを自己編成して、ゆるやかに連合する 1 つのツリーにします。そのツリーのルートにはカタログ・サービスが置かれ、ツリーのリーフ部分にはコンテナーを保持するコア・グループが置かれます。詳しくは、9 ページの『アーキテクチャーおよびトポロジー』のトピックを参照してください。

各コア・グループはカタログ・サービスによって自動的に作成され、約 20 個のサーバーからなるグループに入れられます。コア・グループ・メンバーは、グループ内の他メンバーのヘルス・モニタリングを提供します。また、各コア・グループは、カタログ・サービスにグループ情報を伝達するためのリーダーとして 1 つのメンバーを選びます。コア・グループのサイズを制限することにより、良好なヘルス・モニタリングおよび高度にスケーラブルな環境を維持できます。

注: コア・グループ・サイズを変更できる WebSphere Application Server 環境では、eXtreme Scale は、コア・グループあたり 50 を超えるメンバー数はサポートしません。

障害

プロセスに障害が起きるのには、いくつかの場合があります。何らかのリソース限界に達したり (例えば、最大ヒープ・サイズ)、プロセス制御ロジックがプロセスを強制終了したといった理由で、プロセスに障害が起こることがあります。オペレーティング・システムに障害が起きると、システム上で実行中のすべてのプロセスが失われます。ネットワーク・インターフェース・カード (NIC) などの頻繁には障害が起きないハードウェアに障害が起きると、オペレーティング・システムがネットワークから切断されます。さらに多くのポイントで障害が起きると、プロセスが使用不可になります。このような状況において、これらすべての障害は、プロセス障害または接続不良の 2 つの障害タイプのうちのいずれかに分類されます。

プロセス障害

WebSphere eXtreme Scale は、プロセス障害に非常に迅速に対応します。プロセスに障害が起きると、オペレーティング・システムは、プロセスが使用していたリソースの残りすべてをクリーンアップする必要が生じます。このクリーンアップには、ポートの割り当ておよび接続が含まれています。プロセスに障害が起きると、信号はそのプロセスによって使用されていた接続を通して送信され、各接続がクローズされます。これらの信号を使用し、障害の起きたプロセスに接続されている他のプロセスによって即時にプロセス障害が検出されます。

接続不良

オペレーティング・システムが切断されると、接続不良が発生します。その結果として、オペレーティング・システムは信号を他のプロセスに送信することができな

くなります。接続不良が発生する理由はいくつかありますが、それらの理由は、ホスト障害と孤立化の 2 つのカテゴリーに分割されます。

ホスト障害

マシンの電源コンセントのプラグが抜かれると、即座に動作しなくなります。

孤立化

このシナリオは、使用不可であると見なされたプロセスが実際には使用不可ではないため、ソフトウェアが正しく対処することが最も難しい障害の状態です。基本的に、システムにはサーバーまたは他のプロセスが失敗したように見えるが、実際は正常に実行しているという状況です。

eXtreme Scale コンテナー障害

コンテナー障害は、通常コア・グループ・メカニズムを通してピア・コンテナーによって発見されます。コンテナーまたはコンテナーのセットに障害が起きると、カタログ・サービスにより、そのコンテナーにホスティングされていた断片が移行されます。カタログ・サービスにより、非同期のレプリカに移行する前に最初に同期複製が検索されます。プライマリー断片が新規ホスト・コンテナーに移行された後で、カタログ・サービスにより、欠落しているレプリカの新規ホスト・コンテナーが検索されます。

注: コンテナー孤立化 - コンテナーが使用不可であることが検出されると、カタログ・サービスによりコンテナーの断片がコンテナーから移行されます。これらのコンテナーが使用可能になると、カタログ・サービスは、通常の開始フローのときと同じように、これらのコンテナーを配置対象とみなします。

コンテナー・フェイルオーバー検出までの待ち時間

障害は、ソフトとハードの障害に分類されます。ソフト障害の原因は、一般にプロセスの障害です。そのような障害はオペレーティング・システムによって検出されます。オペレーティング・システムでは、ネットワーク・ソケットなどの使用されたリソースを非常に迅速にリカバリーできます。ソフト障害の場合、標準的な障害検出までの時間は、1 秒未満です。ハード障害の場合、デフォルトのハートビート調整を使用すると、検出まで最長 200 秒かかることもあります。そのような障害は、物理的なマシンの破損、ネットワーク・ケーブル切断、オペレーティング・システム障害などです。したがって、eXtreme Scale がハード障害を検出するには、構成可能なハートビートに頼らざるを得ません。ハード障害を検出するまでの時間を短縮する方法について詳しくは、118 ページの『フェイルオーバー検出のタイプ』を参照してください。

カタログ・サービス障害

カタログ・サービス・グリッドは、1 つの eXtreme Scale グリッドなので、コンテナー障害プロセスと同じようにコア・グループ・メカニズムも使用します。主な相違点は、カタログ・サービス・グリッドでは、コンテナーに使用するカタログ・サービス・アルゴリズムの代わりに、プライマリー断片の定義にピア選択プロセスを使用する点です。

配置サービスおよびコア・グループ化サービスは多くのサービスのうちの 1 つであり、ロケーション・サービスおよび管理はすべての場所で実行されているということに注意してください。配置サービスおよびコア・グループ化サービスは、システムをレイアウトする必要があるため別のものです。ロケーション・サービスおよび管理は読み取り専用サービスであり、スケーラビリティーを提供するためにあらゆる場所に存在します。

カタログ・サービスは複製を使用して、独自の障害限界を設定します。カタログ・サービス・プロセスに障害が起きると、サービスが再始動され、システムを必要なレベルの可用性に復元します。カタログ・サービスをホスティングしているすべてのプロセスで障害が起こると、eXtreme Scale から重要なデータが失われます。この障害により、すべてのコンテナーの再始動が必要になります。カタログ・サービスは多くのプロセスで実行されているため、この障害は起きる可能性のないイベントです。ただし、単一のボックスですべてのプロセスを実行している場合は、単一のブレード・シャーシ内、または単一のネットワーク・スイッチで、障害が起きる可能性があります。カタログ・サービスをホスティングしているボックスから共通の障害モードを除去して、障害が起きる可能性を減らします。

複数のコンテナー障害

プロセスが失われるとプライマリーおよびレプリカの両方が失われるため、レプリカはプライマリーとして同じプロセスに配置するべきではありません。デプロイメント・ポリシーにより、カタログ・サービスがレプリカをプライマリーと同じマシンに配置できるかどうかを判別するのに使用する開発モードのブール値属性が定義されます。単一マシンの開発環境においては、2 つのコンテナーを所有でき、それらの間でレプリカを生成できます。しかし、実動環境では、そのホストを失うことにより両方のコンテナーを失うことになるため、単一マシンの使用では不十分です。単一マシンでの開発モードと複数のマシンを使用する実動モード間でモードを変更するには、デプロイメント・ポリシー構成ファイルで開発モードを無効にします。

可用性向上のための複製

複製は、耐障害性を強化し、分散 eXtreme Scale トポロジーのパフォーマンスを向上させます。

複製は、BackingMap を MapSet に関連付けることで使用可能になります。

MapSet は、区画キーによってカテゴリー化されるマップの集まりです。この区画キーは、個別マップのキーから、そのハッシュ・モジュロを取って区画数とすることで派生します。つまり、MapSet 内の 1 つのマップ・グループが区画キー X を持つとすると、それらのマップはグリッド内の対応する区画 X に保管されます。別のグループが区画キー Y を持つとすると、それらのマップはすべて区画 Y に保管されます。以下同様です。また、マップ内のデータは、MapSet に定義されたポリシーに基づいて複製されます。これは、分散 eXtreme Scale トポロジーのみに使用されます (ローカル・インスタンスの場合は不要です)。

詳しくは、 78 ページの『区画化』 を参照してください。

MapSet は、それらが持つ区画の数および複製ポリシーを割り当てられます。MapSet 複製構成は、MapSet がプライマリー断片に加えて持つことになる同期および非同期の複製断片の数を示すだけです。例えば、1 つの同期複製と 1 つの非同期複製が存在することになる場合、MapSet に割り当てられたすべての BackingMap は、それぞれ eXtreme Scale の使用可能なコンテナー・セット内に自動的に配布される複製断片を持ちます。また MapSet 複製構成により、クライアントは同期複製されたサーバーからデータを読み取れるようになります。これにより、読み取り要求の負荷を eXtreme Scale 内のその他のサーバーにも分散することができます。複製は、BackingMap のプリロード時にプログラミング・モデルに影響するだけです。

さまざまな構成オプションについて詳しくは、以下を参照してください。

マップのプリロード

マップはローダーに関連付けることができます。ローダーは、オブジェクトがマップに見つからない場合 (キャッシュ・ミスの場合) に、そのオブジェクトをフェッチするためにも、またトランザクションのコミット時に変更をバックエンドに書き込むためにも使用されます。ローダーは、マップへのデータのプリロードに使用することもできます。Loader インターフェースの preloadMap メソッドは、MapSet 内のその対応する区画がプライマリーとなると、各マップで呼び出されます。preloadMap メソッドは、レプリカでは呼び出されません。このメソッドは、提供されたセッションを使用して、対象となる参照データのすべてをバックエンドからマップにロードしようとします。関係するマップは、preloadMap メソッドに渡される BackingMap 引数によって識別されます。

```
void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
```

区画に分割された MapSet でのプリロード

マップは、N 個の区画に分割することができます。したがってマップは、複数のサーバーに渡ってストライプすることができます。この場合、各エントリーは、これらのサーバーのうちの 1 つにのみ保管されているキーによって識別されます。アプリケーションは、マップのすべてのエントリーを保持する場合に単一 JVMのヒープ・サイズによる制限を受けなくなるため、非常に大きいマップを eXtreme Scale に保持できるようになります。Loader インターフェースの preloadMap メソッドがプリロードされるアプリケーションは、それがプリロードするデータのサブセットを識別する必要があります。常に、固定数の区画が存在します。この数を判別するには、以下のコード例を使用してください。

```
int numPartitions = backingMap.getPartitionManager().getNumOfPartitions();
int myPartition = backingMap.getPartitionId();
```

このコード例は、データベースからプリロードするデータのサブセットを、アプリケーションがどのように識別できるかを示しています。アプリケーションは、マップが最初に区画に分割されていない場合でも、これらのメソッドを常に使用しなければなりません。これらのメソッドによって柔軟性が実現されます。管理者が後でマップを区画に分割した場合でも、ローダーは正常に機能し続けます。

アプリケーションは、バックエンドから *myPartition* サブセットを検索する照会を発行する必要があります。テーブル内のデータを簡単に区画に分割できるなんらかの自然な照会がある場合を除き、データベースが使用される場合は、所定レコードの区画 ID の列を持つ方が、処理が容易である可能性があります。

パフォーマンス

プリロードの実装では、複数のオブジェクトを単一トランザクションでマップに保管して、データをバックエンドからマップにコピーします。トランザクションごとに保管されるレコードの最適数は、複雑さやサイズなど、いくつかの要因によって決まります。例えば、トランザクションに 100 エントリーを超えるブロックが含まれると、以後は、エントリーの数を増やすに従ってパフォーマンス利益が減少していきます。最適数を知るためには、まず 100 エントリーから始めて、徐々に数を増やしていきます。これをパフォーマンス利益がゼロに減少するまで続けます。 トランザクションが大きいほど、複製パフォーマンスが向上します。ただし、プライマリーのみがプリロード・コードを実行することに注意してください。プリロードされたデータは、プライマリーから、オンラインになっているすべてのレプリカに複製されます。

MapSets のプリロード

アプリケーションが複数のマップを持つ MapSet を使用する場合、各マップはそれぞれ独自のローダーを持ちます。各ローダーに、プリロード・メソッドがあります。各マップは、eXtreme Scale によって順次にロードされます。1 つのマップをプリロード・マップに指定して全マップをプリロードすると、より効率的になる可能性があります。このプロセスは、アプリケーション規則です。例えば、部門と従業員という 2 つのマップが、部門マップと従業員マップの両方をプリロードするために、部門 Loader を使用するとします。このプロシージャーにより、トランザクション上、アプリケーションで部門が必要な場合、その部門の従業員がキャッシュされます。部門 Loader が部門をバックエンドからプリロードするときに、その部門の従業員もフェッチします。その後で、部門オブジェクトとそれに関連する従業員オブジェクトが、単一のトランザクションを使用して、マップに追加されます。

リカバリー可能なプリロード

非常に大きいデータ・セットをキャッシュする必要がある場合があります。このデータのプリロードは、非常に時間がかかる可能性があります。アプリケーションがオンラインになる前に、プリロードを完了しなければならない場合もあります。プリロードをリカバリー可能にすると、便利です。100 万個のレコードをプリロードする必要があるとします。プライマリーがこれらのレコードをプリロードし、800,000 件目のレコードの時点でプライマリーが失敗するとします。通常、新規プライマリーとして選択されたレプリカは、複製状態をクリアして、最初からプリロードを開始します。eXtreme Scale では、ReplicaPreloadController インターフェースを使用できます。アプリケーションのローダーで、ReplicaPreloadController インターフェースを実装する必要が生じることもあります。この例では、単一メソッド `Status checkPreloadStatus(Session session, BackingMap bmap);` をローダーに追加します。Loader インターフェースのプリロード・メソッドが正常に呼び出されるためには、このメソッドが eXtreme Scale ランタイムによって呼び出されます。レプリカがプライマリーにプロモートされると、常に eXtreme Scale がこのメソッド (Status) の結果をテストして、その振る舞いを決定します。

表 6. 状況値および応答

返される状況値	eXtreme Scale の応答
Status.PRELOADED_ALREADY	この状況値は、マップが完全にプリロードされていることを示しているため、eXtreme Scale はプリロード・メソッドをまったく呼び出しません。

表 6. 状況値および応答 (続き)

返される状況値	eXtreme Scale の応答
Status.FULL_PRELOAD_NEEDED	eXtreme Scale はマップをクリアし、プリロード・メソッドを正常に呼び出します。
Status.PARTIAL_PRELOAD_NEEDED	eXtreme Scale は、マップを現状のままにして、プリロードを呼び出します。このストラテジーによって、アプリケーション・ローダーは、この時点以降プリロードを継続することができます。

プライマリーは、マップのプリロード中、返す必要のある状況をレプリカ側で判別できるように、複製中の MapSet 内のマップに必ず何らかの状態を残す必要があります。RecoveryMap などと呼ばれる追加のマップを使用することができます。マップがプリロード中のデータで一貫して複製されるようにするため、この RecoveryMap は、プリロード中の同じ MapSet の一部である必要があります。推奨の実装は、以下のとおりです。

プリロードがレコードの各ブロックをコミットすると、プロセスも、RecoveryMap 内のカウンターまたは値をそのトランザクションの一部として更新します。プリロードされたデータと RecoveryMap データは、レプリカにアトミックに複製されます。レプリカがプライマリーに格上げされると、RecoveryMap をチェックして何が起こったかを確認できるようになります。

RecoveryMap は、状態キーを持つ単一エントリーを保持できます。このキーに対するオブジェクトが存在しない場合には、完全なプリロードが必要となります (checkPreloadStatus は FULL_PRELOAD_NEEDED を返します)。この状態キーに対するオブジェクトが存在し、値が COMPLETE の場合は、プリロードが完了し、checkPreloadStatus メソッドで PRELOADED_ALREADY が返されます。これ以外の場合、値オブジェクトは、プリロードを再開する場所を示し、checkPreloadStatus メソッドは PARTIAL_PRELOAD_NEEDED を返します。ローダーは、プリロードが呼び出されたときにローダーに開始点がわかるように、ローダーのインスタンス変数にリカバリー・ポイントを保管できます。また、各マップが個別にプリロードされる場合、RecoveryMap もマップごとにエントリーを保持できます。

Loader での同期複製モードにおけるリカバリーの処理

eXtreme Scale ランタイムは、プライマリーが失敗したときにコミット済みデータを失わないよう設計されています。次のセクションでは、使用されるアルゴリズムについて説明します。これらのアルゴリズムは、複製グループが同期複製を使用する場合にのみ適用されます。ローダーはオプションです。

eXtreme Scale ランタイムは、すべての変更がプライマリーからレプリカに同期複製されるように構成することができます。同期複製が配置されると、その同期複製は、プライマリー断片にある既存データのコピーを受け取ります。この間もプライマリーはトランザクションを受け取り続け、受け取ったトランザクションを非同期に複製にコピーします。複製はこの時点ではオンラインであるとは見なされません。

複製がプライマリーに追いついた後、複製はピア・モードに入り、複製の同期生成が始まります。プライマリーでコミットされたトランザクションはすべて同期複製

に送信され、プライマリーは各複製からの応答を待ちます。ローダーを使用する、プライマリーでの同期コミット・シーケンスは、以下の一連のステップのようになります。

表 7. プライマリーでのコミット・シーケンス

Loader を使用する場合のステップ	**Loader を使用しない場合のステップ**
エントリーのロックを取得します。	同じ
変更をローダーにフラッシュします。	操作しない
キャッシュに変更を保存します。	同じ
変更をレプリカに送信し、確認通知を待機します。	同じ
TransactionCallback プラグインでローダーをコミットします。	プラグイン・コミットが呼び出されますが、何も実行しません。
エントリーのロックを解除します。	同じ

変更がレプリカに送信された後、ローダーにコミットされることに注意してください。変更がレプリカでコミットされる条件を判別するには、このシーケンスを訂正します。初期化時に、以下のようにプライマリーで tx リストを初期化します。

```
CommitedTx = {}, RolledBackTx = {}
```

同期コミットの処理中に、以下のシーケンスを使用します。

表 8. 同期コミット処理

Loader を使用する場合のステップ	**Loader を使用しない場合のステップ**
エントリーのロックを取得します。	同じ
変更をローダーにフラッシュします。	操作しない
キャッシュに変更を保存します。	同じ
コミット済みトランザクションで変更を送信し、トランザクションをレプリカにロールバックし、肯定応答を待機します。	同じ
コミット済みトランザクションおよびロールバック済みトランザクションのリストをクリアします。	同じ
TransactionCallBack プラグインでローダーをコミットします。	TransactionCallBack プラグイン・コミットがやはり呼び出されますが、通常、何も行われません。
コミットが成功した場合、トランザクションがコミット済みトランザクションに追加され、成功しなかった場合はロールバック済みトランザクションに追加されます。	操作しない
エントリーのロックを解除します。	同じ

レプリカ処理の場合、以下のシーケンスを使用します。

1. レプリカが変更されます。
2. コミット済みトランザクション・リスト内のすべての受信済みトランザクションをコミットします。

3. ロールバック済みトランザクション・リスト内のすべての受信済みトランザクションをロールバックします。
4. トランザクションまたはセッションを開始します。
5. トランザクションまたはセッションに変更を適用します。
6. 保留リストにトランザクションまたはセッションを保存します。
7. 応答を返信します。

レプリカがレプリカ・モードである間は、レプリカ上でローダーによる相互作用が行われないことに注意してください。プライマリーは、すべての変更を Loader を介してプッシュする必要があります。レプリカは変更を行いません。このアルゴリズムの副次作用は、レプリカに常にトランザクションがあるが、次のプライマリー・トランザクションによってこれらのトランザクションのコミット状況が送信されるまで、コミットされないことです。その場合には、トランザクションはレプリカ上でコミットまたはロールバックされます。このようになるまでは、トランザクションはコミットされません。短い時間 (数秒) 後にトランザクションの結果が送信されるようなタイマーをプライマリーに追加することができます。このタイマーは、その時刻ウィンドウに対する失効性を制限しますが、除去はしません。こうした失効性は、レプリカ読み取りモードを使用する場合のみの問題です。それ以外の点では、失効性は、アプリケーションに影響を与えません。

プライマリーが失敗した場合、プライマリーでコミットまたはロールバックされたトランザクションがいくつかある可能性がありますが、これらの結果が含まれるメッセージがレプリカに到達しませんでした。レプリカが新規プライマリーにプロモートされる際の最初のアクションの 1 つは、この状態に対処することです。保留中の各トランザクションは、新規プライマリーのマップ・セットに対して再処理されます。ローダーがある場合は、そのローダーに各トランザクションが送られます。これらのトランザクションには、厳密な先入れ先出し法 (FIFO) 順序が適用されます。失敗したトランザクションは無視されます。例えば、3 つのトランザクション A、B、および C が保留中の場合、A はコミットし、B はロールバックし、C もコミットする可能性があります。1 つのトランザクションが他のトランザクションに影響を与えることはありません。これらのトランザクションは独立したものと見なされます。

ローダーで使用されるロジックは、フェイルオーバー・リカバリー・モードと通常モードの場合では若干異なることがあります。ローダーがフェイルオーバー・リカバリー・モードであるときは、ReplicaPreloadController インターフェースを実装することで容易に識別できます。checkPreloadStatus メソッドは、フェイルオーバー・リカバリーが完了した場合にのみ呼び出されます。このため、Loader インターフェースの apply メソッドが checkPreloadStatus メソッドより前に呼び出される場合は、リカバリー・トランザクションになります。checkPreloadStatus メソッドが呼び出されると、フェイルオーバー・リカバリーが完了します。

レプリカ間のロード・バランシング

特に構成されていない限り、eXtreme Scale は、すべての読み取り要求と書き込み要求を指定された複製グループのプライマリー・サーバーに送信します。プライマリーは、クライアントからのすべての要求にサービスを提供する必要があります。読み取り要求をプライマリーのレプリカに送信できるようにするとよいでしょう。読み取り要求をレプリカに送信することにより、読み取り要求の負荷を複数の Java 仮

想マシン (JVM) で共有できるようになります。ただし、読み取り要求のためにレプリカを使用すると、応答が不整合になる可能性があります。

レプリカ間のロード・バランシングは、通常、クライアントが常時変更されるデータをキャッシュしているか、またはクライアントがペシミスティック・ロックを使用している場合にのみ使用されます。

データが絶えず変更され、そのためクライアントのニア・キャッシュで無効化された場合は、結果としてクライアントからプライマリーへの get 要求率が比較的高くなります。同様に、ペシミスティック・ロック・モードでは、ローカル・キャッシュが存在しないため、すべての要求がプライマリーに送信されます。

データが比較的静的であるか、またはペシミスティック・モードが使用されていない場合には、読み取り要求をレプリカへ送信しても、パフォーマンスにそれほど大きな影響を与えません。データで満杯のキャッシュを持つクライアントからの get 要求の頻度は、高くありません。

クライアントが始動されたばかりのときには、ニア・キャッシュは空です。空のキャッシュに対するキャッシュ要求は、プライマリーに転送されます。時間が経過してクライアント・キャッシュにデータが入れられると、要求ロードは除去されます。数多くのクライアントが同時に始動される場合には、ロードは大きくなる可能性があるため、パフォーマンス上、レプリカ読み取りを選択するほうが適切な場合があります。

クライアント・サイドの複製

eXtreme Scale により、非同期複製を使用して、サーバー・マップを 1 つ以上のクライアントに複製することができます。クライアントは ClientReplicableMap.enableClientReplication メソッドを使用して、サーバー・サイド・マップのローカルの読み取り専用コピーを要求できます。

```
void enableClientReplication(Mode mode, int[] partitions, ReplicationMapListener listener) throws ObjectGridException;
```

最初のパラメーターは複製モードです。このモードには、連続複製またはスナップショット複製を指定できます。2 番目のパラメーターは、データの複製元の区画を表す区画 ID の配列です。この値がヌルの場合、または空の配列の場合、データはすべての区画から複製されます。最後のパラメーターは、クライアント複製イベントを受信するためのリスナーです。詳しくは、API 資料の ClientReplicableMap および ReplicationMapListener を参照してください。

複製が有効になると、サーバーはクライアントへのマップの複製を開始します。結局のところ、クライアントは、どの時点においてもわずか数トランザクションでサーバーに到達します。

複製アーキテクチャー

eXtreme Scale を使用すると、メモリー内のデータベースまたは断片を、Java 仮想マシン (JVM) 相互間で複製することができます。断片は、コンテナー上に配置された区画を表します。異なる区画を表す複数の断片が、単一のコンテナー上に存在することができます。複製を実行するために、プライマリー断片と複製断片が各区画に存在しています。複製断片は、同期または非同期のいずれかです。複製断片のタ

イプと配置は、eXtreme Scale により、同期断片および非同期断片の最小数と最大数を指定するデプロイメント・ポリシーを使用して決定されます。

断片タイプ

複製の生成では、次の 3 つのタイプの断片が使用されます。

- プライマリー
- 同期複製
- 非同期複製

プライマリー断片は、挿入、更新、および除去の各操作をすべて受信します。プライマリー断片は、レプリカの追加と除去を行い、データをレプリカに対して複製し、トランザクションのコミットとロールバックを管理します。

同期複製は、プライマリーと同じ状態を保持します。プライマリーがデータを同期複製に対して複製する場合、トランザクションは、同期複製上でコミットするまで、コミットされません。

非同期複製は、プライマリーと同じ状態である場合も、同じ状態でない場合もあります。プライマリーがデータを非同期複製に対して複製する場合、プライマリーは、非同期複製がコミットするのを待機しません。非同期複製は、プライマリーから送信されたトランザクションの順序はそのまま保持します。

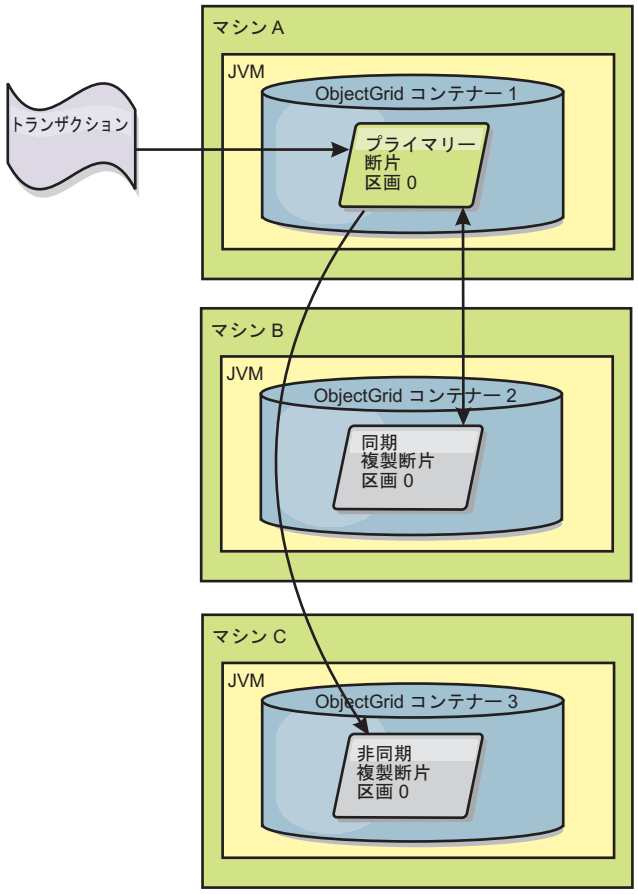


図 33. プライマリー断片と複製断片との間の通信パス

複製のメモリー・コスト

レプリカをオンラインに移す場合は、既存のデータをプライマリーからコピーするため、チェックポイント・マップを作成する必要があります。しかし、プライマリーでの主要なメモリー・コストは、チェックポイント後に変更される各エントリーに対するメモリーです。チェックポイント後にデータが大量に変更される場合、この操作には、変更されたエントリーの数に比例するメモリーの量がコストとしてかかります。チェックポイントがレプリカにコピーされた後、変更は、チェックポイントにマージされて戻ります。変更されたエントリーは、必要なすべてのレプリカに送信されるまで解放されません。

最小同期複製断片数

プライマリーは、データのコミットを準備するときに、トランザクションのコミットを断定した同期複製断片の数を検査します。トランザクションがレプリカに対して通常に処理を行う場合は、コミットを断定します。同期複製に何らかの異常がある場合は、コミットしないことを断定します。プライマリーがコミットする前に、コミットを断定している同期複製断片の数が、デプロイメント・ポリシーの minSyncReplica 設定に適合している必要があります。コミットを断定している同期複製断片の数が小さ過ぎる場合、プライマリーはトランザクションをコミットせず、エラーとなります。このアクションにより、正しいデータには、必要な数の同

期複製が必ず使用可能となります。エラーを検出した同期複製は、再登録して、その状態を修正します。再登録について詳しくは、複製断片のリカバリーを参照してください。

コミットを断定した同期複製の数が少な過ぎる場合、プライマリーは、ReplicationVotedToRollbackTransactionException エラーをスローします。

複製およびローダー

通常、プライマリー断片は、変更を、ローダーを介して同期的にデータベースに書き込みます。ローダーとデータベースは、常に同期しています。プライマリーが複製断片にフェイルオーバーする場合、データベースとローダーは同期しない場合があります。以下に例を示します。

- プライマリーは、トランザクションをレプリカに送信してから、データベースに対してコミットする前に、失敗する場合があります。
- プライマリーは、データベースに対してコミットしてから、レプリカに送信する前に、失敗する場合があります。

どちらの場合も、レプリカが、1 トランザクションだけデータベースの前に、または 1 トランザクションだけデータベースの後ろに移動します。この状態は許容されません。eXtreme Scale は、ローダー実装に特殊なプロトコルと契約を使用して、2 フェーズ・コミットを使用せずにこの問題を解決します。プロトコルは、次のようになります。

プライマリー・サイド

- トランザクションを、前のトランザクション結果と一緒に送信します。
- データベースに書き込み、トランザクションのコミットを試行します。
- データベースがコミットする場合、eXtreme Scale 上でコミットします。データベースがコミットしない場合には、トランザクションをロールバックします。
- 結果を記録します。

レプリカ・サイド

- トランザクションを受信し、それをバッファーに入れます。
- すべての結果について、トランザクションと一緒に送信し、バッファーに入れられたすべてのトランザクションをコミットし、ロールバックされたすべてのトランザクションを廃棄します。

フェイルオーバーする場合のレプリカ・サイド

- バッファーに入れられたすべてのトランザクションについて、トランザクションをローダーに提供し、ローダーはトランザクションのコミットを試行します。
- 各トランザクションがべき等になるようにローダーを作成する必要があります。
- トランザクションが既にデータベース内にある場合は、ローダーはいかなる操作も行いません。
- トランザクションがデータベース内にない場合には、ローダーはトランザクションを適用します。
- トランザクションがすべて処理されてから、新しいプライマリーが要求のサービス提供を開始できます。

このプロトコルにより、データベースは、確実に新規プライマリー状態と同じレベルとなります。

断片割り振り: プライマリーおよびレプリカ

カタログ・サービスは断片を配置します。各 ObjectGrid には複数の区画があります。区画ごとに、プライマリー断片、およびオプションの複製断片セットがあります。カタログ・サービスは、同じコンテナーに同じ区画のレプリカおよびプライマリー断片を配置しません。また、構成が開発モードでない限り、レプリカおよびプライマリー断片を同じ IP アドレスを持つコンテナーに配置しません。カタログ・サービスは、断片タイプが使用可能なコンテナーに均等に分散されるように、そのバランスを取ります。

新しいコンテナーが開始すると、eXtreme Scale は、比較的負荷の多いコンテナーから断片を取り出して、この新しい空のコンテナーに入れます。この振る舞いにより、eXtreme Scale は、その根幹をなす弾力性を実現、維持します。弾力性は、水平スケーリング (スケールアウトとスケールインの両方) における強力な能力となって現れます。

スケールアウト

スケールアウトとは、追加の Java 仮想マシン またはコンテナーが eXtreme Scale グリッドに追加された場合に、eXtreme Scale が既存の断片 (プライマリーまたは複製) を古い JVM のセットから新しいセットに移動しようとすることです。この移動により、グリッドを拡張して、新規に追加された JVM のプロセッサー、ネットワーク、およびメモリーを利用できるようになります。また、この移動は、グリッドのバランスを取り、グリッド内の各 JVM がホストするデータ量が等しくなるようにします。グリッドの拡張にともなって、グリッド全体のうち各サーバーがホストするサブセットは小さくなります。eXtreme Scale は、区画間でデータが均等に分散していると想定します。この拡張により、スケールアウトが可能になります。

スケールイン

スケールインとは、JVM の 1 つに障害が起こった場合に、そのダメージを eXtreme Scale が修復しようとすることです。障害が発生した JVM にレプリカがあった場合、eXtreme Scale は、残っている JVM のレプリカを新規に作成して、失われたレプリカと置き換えます。障害が発生した JVM にプライマリーがあった場合、eXtreme Scale は、残りの中から最適なレプリカを見つけて、そのレプリカを新規プライマリーとしてプロモートします。次に、eXtreme Scale は、障害が起きていないサーバー上に新しい複製を作成して、プロモートした複製と置き換えます。スケーラビリティーを保つため、サーバーに障害が起こっても eXtreme Scale は区画の複製数を維持します。

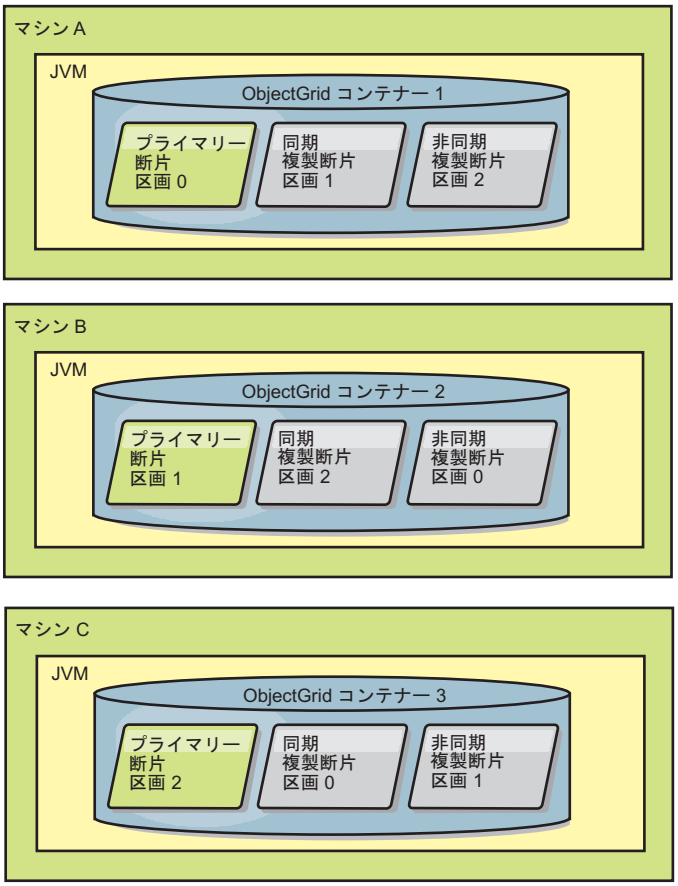


図 34. 区画が 3 つあり、minSyncReplicas 値を 1 に、maxSyncReplicas 値を 1 に、 maxAsyncReplicas 値を 1 にするというデプロイメント方針での ObjectGrid マップ・セットの配置

ライフサイクル、リカバリー、および障害イベント

断片は、さまざまな状態とイベントを経由して複製をサポートします。断片のライフサイクルには、オンラインになること、ランタイム、シャットダウン、フェイルオーバー、およびエラー処理があります。サーバー状態変更を処理するため、複製断片はプライマリー断片にプロモート可能です。

ライフサイクル・イベント

プライマリー断片と複製断片は、配置されて開始されると、一連のイベントを経由してオンラインとなり、listen モードとなります。

プライマリー断片

カタログ・サービスは、プライマリー断片を区画に配置します。カタログ・サービスは、プライマリー断片のロケーションの平衡化、およびプライマリー断片に対するフェイルオーバーの開始の作業も行います。

ある断片がプライマリー断片になると、このプライマリー断片はカタログ・サービスからレプリカのリストを受信します。新規のプライマリー断片は、レプリカ・グループを作成し、すべてのレプリカを登録します。

プライマリーが作動可能となると、「ビジネス用にオープン」メッセージが、プライマリーの実行されているコンテナーの SystemOut.log ファイルに表示されます。オープン・メッセージ、つまり CWOBJ1511I メッセージには、開始したプライマリー断片のマップ名、マップ・セット名、および区画番号がリストされます。

```
CWOBJ1511I: mapName:mapSetName:partitionNumber (primary) is open for business.
```

カタログ・サービスが断片をどのように配置するかに関する情報については、 101 ページの『断片割り振り: プライマリーおよびレプリカ』を参照してください。

複製断片

複製断片は、問題を検出した場合を除き、主にプライマリー断片に制御されます。通常のライフサイクルでは、プライマリー断片が、複製断片を配置、登録、および登録抹消します。

プライマリー断片が複製断片を初期化すると、メッセージでログが表示されます。このログには、レプリカが実行されている場所が記述され、複製断片が使用可能であることが示されます。オープン・メッセージ、つまり CWOBJ1511I メッセージには、複製断片のマップ名、マップ・セット名、および区画番号がリストされます。このメッセージは、次のとおりです。

```
CWOBJ1511I: mapName:mapSetName:partitionNumber (synchronous replica) is open for business.
```

または

```
CWOBJ1511I: mapName:mapSetName:partitionNumber (asynchronous replica) is open for business.
```

複製断片が最初に開始するときは、まだピア・モードにはなっていません。複製断片がピア・モードになっていると、複製断片は、データがプライマリーに着信するときにプライマリーからデータを受信します。ピア・モードに入る前に、複製断片には、プライマリー断片上のすべての既存データのコピーが必要となります。

プライマリー断片上のデータのコピーを取得するため、同期および非同期の両複製断片は、同じ開始シーケンスを実行します。複製断片の登録中に、プライマリー断片は、現行データのチェックポイントを作成します。プライマリー断片上のデータは、Copy-on-Write 状態です。複製断片に送信される現行データは決して変更されませんが、新規トランザクションがプライマリー上のレコードを更新するときは常に、Copy-on-Write が実行されます。プライマリー断片は、複製断片に送信されるデータを変更することなく、処理を続行できます。プライマリー断片は、チェックポイント以降に行われた変更のリストを保持します。

チェックポイント・データは、レプリカへプッシュされます。チェックポイントがレプリカに到着した後、チェックポイントのメモリーは解放されます。チェックポイントの作成以降の変更は、マージされます。変更のリストも、複製断片へプッシュされます。変更がプッシュされると同時に変更のメモリーが解放されます。

レプリカは、チェックポイント・フェーズを完了した後、ピア・モードに切り替わり、プライマリーがデータを受信すると同時にデータの受信を開始します。この時点で、複製断片は、同期または非同期のいずれかの複製断片として振る舞うことを開始します。

複製断片がピア・モードに達すると、複製断片は、メッセージをレプリカ用の SystemOut.log ファイルに出力します。

```
CWOBJ1526I: Replica objectGridName:mapsetName:partitionNumber:mapName entering peer
mode after X seconds.
```

所要時間は、複製断片が、プライマリー断片から最初のデータをすべて取得するのに要した時間の長さを指します。これには、チェックポイント・データ、およびチェックポイント・コピー中に行われた追加の変更がすべて含まれます。プライマリー断片によって複製される既存のデータが存在しない場合、所要時間は、ゼロまたは非常に低く表示されます。

同期複製断片

新規レプリカが同期複製断片である場合、プライマリー断片は、トランザクションがプライマリーでコミットするたびに、要求または応答を開始するようになります。プライマリー断片は、複製断片がデータを受け取ったと応答するまで待機します。同期複製断片のデータは、プライマリー断片のデータと同じレベルに保たれます。

非同期複製断片

新規レプリカが非同期複製断片である場合、プライマリー断片は、データをレプリカに送信しますが、応答を待機しません。非同期複製は、プライマリーから送信されたデータを番号順に配列、適用します。非同期複製断片のデータは、プライマリー断片のデータと同じレベルに保たれるという保証はありません。

すべての複製断片でのピア・モード

ピア・モードの場合、すべての複製断片がトランザクションの変更を受信してから、トランザクションのメモリーがプライマリー断片上で解放されます。挿入、更新、除去など、データを変更するトランザクション中、複製断片は、データをプライマリー断片からのみ受信します。複製断片は、プライマリー断片上のデータを読み取るために接続されることはありません。

リカバリー・イベント

複製は、障害およびエラー・イベントからリカバリーするように設計されています。あるプライマリー断片が失敗すると、別のレプリカが引き継ぎます。エラーが複製断片上にある場合、複製断片は、リカバリーを試行します。カタログ・サービスは、新規プライマリー断片または新規複製断片の配置とトランザクションを制御します。

複製断片がプライマリー断片となる

複製断片は、2 つの理由でプライマリー断片となります。プライマリー断片が停止または失敗した場合と、バランスを取る決定が行われて、前のプライマリー断片を新規ロケーションに移動した場合です。

カタログ・サービスは、新規プライマリー断片を、既存の同期複製断片から選択します。新規プライマリー断片は、すべての既存レプリカを登録し、トランザクションを新規プライマリー断片として受け入れます。既存の複製断片に正しいレベルのデータが存在する場合、現行データは、複製断片が新規プライマリー断片に登録されると同時に保存されます。非同期複製断片が遅れている場合、非同期複製断片

は、データの新しいコピーを受信し、登録します。

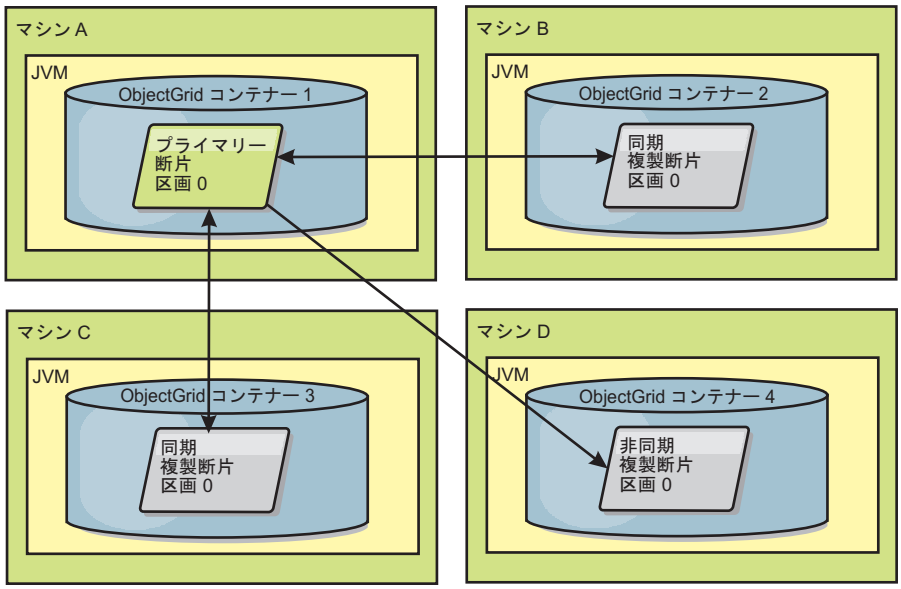


図 35. partition0 区画用の ObjectGrid マップ・セットの配置例。これは、minSyncReplicas 値を 1 に、maxSyncReplicas 値を 2 に、maxAsyncReplicas 値を 1 にするというデプロイメント方針です。

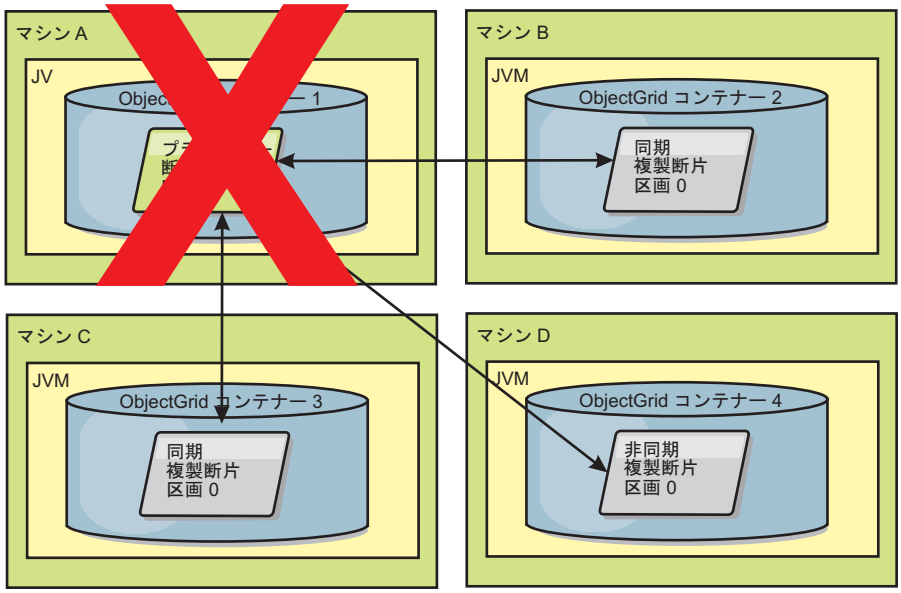


図 36. プライマリー断片のコンテナーに障害が起こる

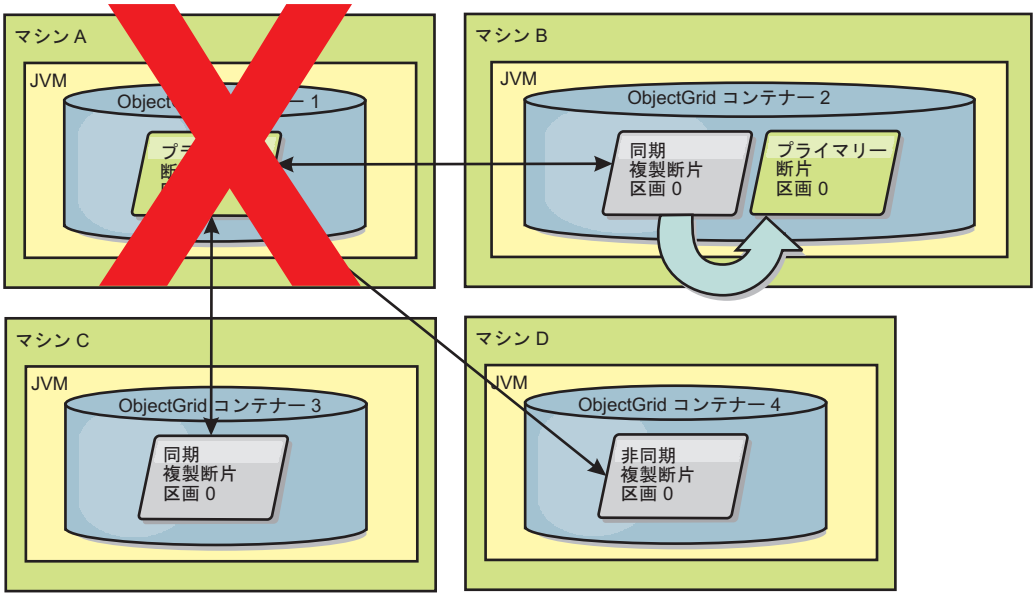


図 37. ObjectGrid コンテナー 2 にある同期複製断片がプライマリー断片になる

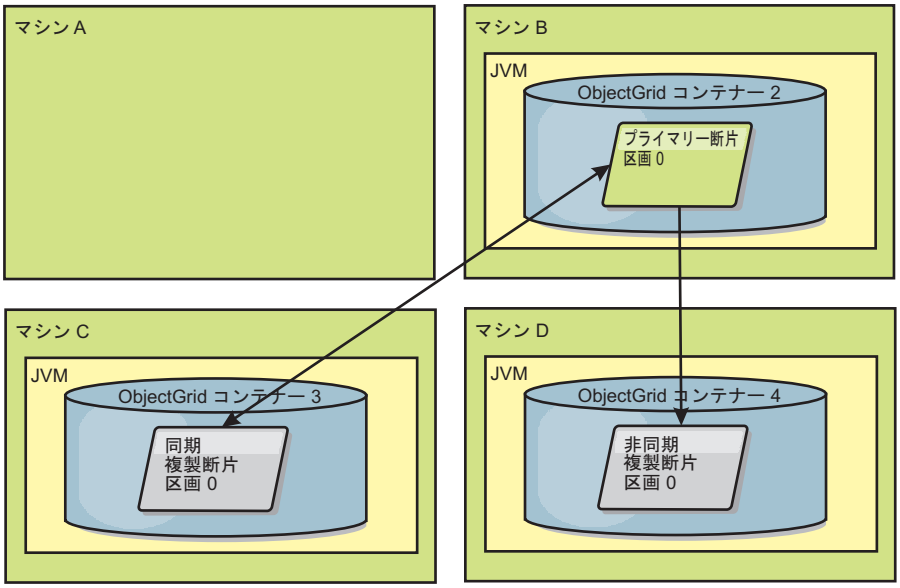


図 38. マシン B にプライマリー断片が含まれています。自動修復モードがどのように設定されているか、およびコンテナーが使用可能かどうかに基づいて、新しい同期複製断片がマシンに配置されるかどうかが決まります。

複製断片のリカバリー

複製断片は、プライマリー断片によって制御されます。ただし、複製断片は、問題を検出すると、登録イベントをトリガーして、データの状態を訂正することができます。レプリカは、現行データをクリアして、新しいコピーをプライマリーから取得します。

複製断片が登録イベントを開始すると、そのレプリカはログ・メッセージを出力します。

```
CWOBJ1524I: Replica listener
objectGridName:mapSetName:partition must re-register with the primary.
Reason: Exception listed
```

トランザクションの処理中に複製断片上でエラーが発生した場合には、その複製断片は不明な状態です。トランザクションはプライマリー断片上で正常に処理されましたが、レプリカ上で何らかの異常が発生しました。この状態を訂正するため、レプリカは再登録イベントを開始します。プライマリーからのデータの新しいコピーを使用して、複製断片は続行できます。同じ問題が再発生する場合、複製断片は、連続して再登録を行いません。詳しくは、『障害イベント』 を参照してください。

障害イベント

レプリカは、リカバリーできないエラー状態を検出した場合、データの複製を停止することがあります。

多すぎる登録試行

レプリカがデータを正常にコミットせずに、登録を複数回トリガーする場合、レプリカは停止します。停止により、レプリカがエンドレス登録ループに入ることが防止されます。デフォルトでは、複製断片は、登録を連続して 3 回試行してから、停止します。

複製断片が何度も登録しすぎる場合、レプリカは、次のメッセージをログに出力します。

```
CWOBJ1537E: objectGridName:mapSetName:partition exceeded the maximum number
of times to reregister (timesAllowed) without successful transactions..
```

レプリカが再登録によってリカバリーできない場合は、複製断片に関連するトランザクションに、広範囲な問題が存在する可能性があります。トランザクションからのキーまたは値の展開中にエラーが発生する場合、クラスパス上のリソースが欠落しているという問題が考えられます。

ピア・モードに入る際の障害

プライマリー (チェックポイント・データ) からの既存のバルク・データの処理中に、レプリカがピア・モードに入ろうとしてエラーが発生した場合、レプリカはシャットダウンします。シャットダウンは、レプリカが正しくない初期データで開始するのを防止します。レプリカは、再登録すれば、プライマリーから同じデータを受信するため、再試行はしません。

複製断片がピア・モードに入ることに失敗した場合、複製断片は、次のメッセージをログに出力します。

```
CWOBJ1527W Replica objectGridName:mapSetName:partition:mapName failed to enter peer mode after numSeconds seconds.
```

レプリカがピア・モードに入ることに失敗した理由を説明する追加のメッセージがログに表示されます。

再登録またはピア・モード障害後のリカバリー

レプリカが再登録できないか、ピア・モードに入ることができない場合、そのレプリカは、新規配置イベントが発生するまで非アクティブ状態になります。新規配置

イベントは、新規サーバーの開始または停止である場合があります。配置イベントは、PlacementServiceMBean Mbean で triggerPlacement メソッドを使用して開始することもできます。

レプリカからの読み取り

クライアントがプライマリー断片のみに制限されず複製からの読み取りも許可されるようにマップ・セットを構成することができます。

障害に備えて複製を単に潜在的なプライマリー以上のものにできれば便利なときがよくあります。例えば、MapSet の replicaReadEnabled オプションを true に設定すれば、読み取り操作が複製に経路指定されるようにマップ・セットを構成することができます。デフォルト設定は false です。

MapSet エレメントについて詳しくは、「*管理ガイド*」に記載されているデプロイメント・ポリシー記述子 XML ファイルに関するトピックを参照してください。

複製の読み取りを使用可能にすると、読み取り要求がより多くの Java™ 仮想マシンに拡散されるので、パフォーマンスが向上します。このオプションが使用可能になっていないと、ObjectMap.get メソッドや Query.getResultIterator メソッドなどの読み取り要求はすべてプライマリーに経路指定されます。 replicaReadEnabled が true に設定されているときには、get 要求が不整合データを戻す可能性があるため、このオプションを使用しているアプリケーションはこの可能性を許容できるようにする必要があります。ただし、キャッシュ・ミスは起こりません。データが複製上にないと、get 要求はプライマリーにリダイレクトされ、再試行されます。

replicaReadEnabled オプションは、同期複製および非同期複製の両方と一緒に使用できます。

レプリカ配置のためのゾーンの使用

ゾーン・サポートは、データ・センター間での断片配置のための高度な構成を可能にします。この機能により、少数のオプションの配置ルールを使用して、何千という区画のグリッドを簡単に管理することができます。データ・センターは、ゾーン・ルールによる構成に従って、ビルの別フロア、別ビル、または異なる都市にさえ配置することも、またはその他の区分に配置することができます。

ゾーンの柔軟性

ゾーンに断片を配置することができます。この機能により、eXtreme Scale がグリッド上に断片をどのように配置するかをさらに制御できるようになります。eXtreme Scale サーバーをホストする Java 仮想マシンは、ゾーン ID によってタグ付けすることができます。現在、デプロイメント・ファイルには 1 つ以上のゾーン・ルールを含めることができます。これらのゾーン・ルールは、断片タイプに関連付けられます。このことを説明する最善の方法として、いくつか例を挙げ、詳しく説明します。

配置ゾーンは、高度なトポロジーを構成するために、eXtreme Scale がプライマリーとレプリカの割り当てをどのように達成するかを制御します。

Java 仮想マシンは、複数のアクティブ eXtreme Scale サーバーを持つことができます。コンテナーは、単一の eXtreme Scale の複数の断片をホストできます。

この機能を利用すると、レプリカとプライマリーを異なるロケーションまたはゾーンに配置して、より優れた高可用性を実現することができます。通常、eXtreme Scale は、同じ IP アドレスでプライマリー断片と複製断片を Java 仮想マシンに配置することはありません。この単純なルールにより、一般的には、2 つの eXtreme Scale サーバーが同じ物理コンピューターに配置されることがなくなります。ただし、より柔軟なメカニズムが必要になる場合もあります。例えば、2 つのブレード・シャーシを使用していて、プライマリーを両方のシャーシ間でストライプし、それぞれのプライマリーのレプリカがそのプライマリーの他方のシャーシに配置されるようにしたい場合があります。

ストライプ・プライマリーは、プライマリーが各ゾーンに配置され、その各プライマリーの複製が対向ゾーンに配置されることを意味します。例えば、プライマリー 0 は zoneA に入り、同期複製 0 は zoneB に入ります。プライマリー 1 は zoneB に入り、同期複製 1 は zoneA に入ります。

この場合、シャーシ名がゾーン名となります。代替方法として、ビルのフロアの後にゾーンを命名し、ゾーンを使用して、同じデータのプライマリーとレプリカが別フロアに配置されるようにすることができます。ビルおよびデータ・センターでも可能です。データ・センター間でデータが適切に複製されるようにするためのメカニズムとしてゾーンを使用し、データ・センター全体に渡るテストが実行されています。eXtreme Scale の HTTP セッション・マネージャーを使用している場合も、ゾーンを使用することができます。このフィーチャーを使用すると、1 つの Web アプリケーションを 3 つのデータ・センターに渡ってデプロイできます。これにより、ユーザーの HTTP セッションがデータ・センター間で複製され、1 つのデータ・センター全体が障害を起こした場合にも、セッションを回復できるようになります。

WebSphere eXtreme Scale は、複数のデータ・センターに渡る大きなグリッドを管理する必要性を配慮されています。これにより、同一区画のバックアップとプライマリーを必要に応じて異なるデータ・センターに配置することが可能です。すべてのプライマリーをデータ・センター 1 に、すべてのレプリカをデータ・センター 2 に配置したり、両方のデータ・センター間でプライマリーとレプリカをラウンドロビンしたりすることができます。ルールは柔軟であり、さまざまなシナリオが考えられます。また eXtreme Scale は数千ものサーバーを管理することができます。これにより、データ・センター認識による完全な自動配置を同時に使用することによって、管理の観点から手ごろな大規模グリッドの作成が可能です。管理者は、簡単かつ効果的に実行するものを指定できます。

管理者の場合、配置ゾーンを使用して、プライマリー断片と複製断片の配置場所を制御できます。これにより、高性能でかつ高可用性のトポロジーのセットアップが可能になります。前述したように、eXtreme Scale プロセスのいずれの論理グループに対してもゾーンを定義できます。これらのゾーンは、データ・センター、データ・センターのフロア、ブレード・シャーシなど、物理ワークステーション・ロケーションに対応付けることができます。ゾーン間でデータをストライプすることができます。これにより、可用性が増します。またホット・スタンバイが必要な場合に、プライマリーとレプリカを別々のゾーンに分割することができます。

eXtreme Scale サーバーの WebSphere Extended Deployment を使用しないゾーンとの関連付け

eXtreme Scale が Java Standard Edition で使用されるか、あるいは、 WebSphere Extended Deployment バージョン 6.1 をベースにしていないアプリケーション・サーバーで使用される場合、断片コンテナーである JVM は、以下の手法を使用して、ゾーンに関連付けられます。

startOgServer スクリプトを使用するアプリケーション

startOgServer スクリプトは、既存のサーバーに組み込まれない eXtreme Scale アプリケーションを開始するために使用されます。**-zone** パラメーターを使用すると、サーバー内のすべてのコンテナーに使用するゾーンを指定できます。

API を使用するコンテナーを開始する場合のゾーンの指定

WebSphere Extended Deployment ノードのゾーンとの関連付け

eXtreme Scale をWebSphere Extended Deployment Java Platform, Enterprise Edition アプリケーションで使用している場合、WebSphere Extended Deployment ノード・グループ・フィーチャーを活用して、クラスター・メンバーの Java 仮想マシンを特定のゾーンに自動的に配置できます。 JVM は、単一ゾーンのメンバーとすることができます。そのようなノード・グループのメンバーのノード上で稼働しているサーバーは、ノード・グループ名で指定されるゾーンに組み込まれます。これらのクラスター・メンバーが 2 つ以上のこうしたノード・グループのメンバーとならないようにする必要があります。クラスター・メンバー JVM は、始動時にのみゾーン・メンバーシップをチェックします。新規ノード・グループを追加するか、メンバーシップを変更した場合、それは新たに始動されるか、再始動される Java 仮想マシンのみに影響します。

ゾーン・ルール

eXtreme Scale 区画には、1 つのプライマリー断片とゼロ個以上の複製断片があります。この例の場合、これらの断片について以下の命名規則を考慮してください。P はプライマリー断片で、S は同期複製で、A は非同期複製です。ゾーン・ルールは以下の 3 つの部分からなります。

- ルール名
- ゾーンのリスト
- 包含または排他フラグ

ゾーン・ルールは、断片を配置できるゾーンのセットを指定します。包含フラグは、1 つの断片がリストからゾーンに配置されると、他のすべての断片もそのゾーンに配置されることを示します。排他設定は、区画の各断片がゾーン・リストの異なるゾーンに配置されることを示します。例えば、排他設定を使用する場合、3 つの断片 (プライマリーと 2 つの同期複製) がある場合は、ゾーン・リストに 3 つのゾーンがなければならないということです。

各断片は、1 つのゾーン・ルールに関連付けることができます。ゾーン・ルールは、2 つの断片間で共有できます。ルールが共有される場合、包含または排他フラグは、1 つのルールを共有するすべてのタイプの断片に拡張されます。

例

さまざまなシナリオおよびそのシナリオを実装するためのデプロイメント構成を示す一連の例は、以下のとおりです。

ゾーン間でプライマリーとレプリカをストライピングする

3 つのブレード・シャーシがあり、3 つすべてにプライマリーを分散し、1 つの同期複製をプライマリー以外のシャーシに配置するものとします。各シャーシをシャーシ名 ALPHA、BETA、および GAMMA を持つゾーンとして定義します。デプロイメント XML の例は以下のとおりです。

```
<?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="library">
   <mapSet name="ms1" numberOfPartitions="37" minSyncReplicas="1"
    maxSyncReplicas="1" maxAsyncReplicas="0">
   <map ref="book" />
   <zoneMetadata>
    <shardMapping shard="P" zoneRuleRef="stripeZone"/>
    <shardMapping shard="S" zoneRuleRef="stripeZone"/>
    <zoneRule name ="stripeZone" exclusivePlacement="true" >
     <zone name="ALPHA" />
     <zone name="BETA" />
     <zone name="GAMMA" />
    </zoneRule>
   </zoneMetadata>
  </mapSet>
 </objectgridDeployment>
</deploymentPolicy>
```

このデプロイメント XML には、「book」という名前の 1 つのマップを持つ「library」という名前のグリッドが含まれます。さらに、1 つの同期複製を持つ 4 つの区画を使用します。zone metadata 文節は、1 つのゾーン・ルールの定義およびゾーン・ルールと断片の関連付けを示します。プライマリー断片と同期断片は、ともにゾーン・ルール「stripeZone」に関連付けられます。このゾーン・ルールでは 3 つのゾーンがすべて含まれ、排他配置が使用されるようになっています。このルールは、区画 0 のプライマリーが ALPHA に配置されると、区画 0 のレプリカは BETA か GAMMA のいずれかに配置されることになります。他の区画のプライマリーは他のゾーンに配置され、レプリカが同様に配置されます。

非同期複製がプライマリーや同期複製と異なるゾーンにある

この例では、2 つのビルがあり、その間の接続は待ち時間が長いものとします。すべてのシナリオでデータ損失のない高可用性が保たれるようにします。ただし、ビル間の同期複製によるパフォーマンス・インパクトのためトレードオフが生じます。このため、一方のビルに同期複製があり、他方のビルに非同期複製があるようなプライマリーが必要になります。通常では、障害は、大規模な問題ではなく、JVM の破損やコンピューター障害です。このトポロジーを使用すると、データ損失なしに通常の障害を切り抜けることができます。ビルがなくなることは非常にまれなことであるため、その場合でもある程度のデータ損失は許容範囲内に収まります。それぞれのビルに 1 つずつ、合計 2 つのゾーンを作成できます。デプロイメント XML ファイルは以下のようになります。

```
<?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="library">
  <mapSet name="ms1" numberOfPartitions="13" minSyncReplicas="1"
```

```
   maxSyncReplicas="1" maxAsyncReplicas="1">
   <map ref="book" />
   <zoneMetadata>
    <shardMapping shard="P" zoneRuleRef="primarySync"/>
    <shardMapping shard="S" zoneRuleRef="primarySync"/>
    <shardMapping shard="A" zoneRuleRef="aysnc"/>
    <zoneRule name ="primarySync" exclusivePlacement="false" >
      <zone name="BldA" />
      <zone name="BldB" />
    </zoneRule>
    <zoneRule name="aysnc" exclusivePlacement="true">
      <zone name="BldA" />
      <zone name="BldB" />
    </zoneRule>
   </zoneMetadata>
  </mapSet>
 </objectgridDeployment>
</deploymentPolicy>
```

プライマリーと同期複製は、排他フラグ設定 false で primaySync ゾーン・ルールを共有します。このため、プライマリーか同期のいずれかがゾーンに配置されると、他方も同じゾーンに配置されます。非同期複製は、primarySync ゾーン・ルールと同じゾーンで第 2 のゾーン・ルールを使用しますが、true に設定された **exclusivePlacement** 属性を使用します。この属性は、断片を同じ区画の別の断片があるゾーンには配置できないことを示します。結果的に、非同期複製は、プライマリーまたは同期複製と同じゾーンに配置されません。

すべてのプライマリーを 1 つのゾーンに配置し、すべてのレプリカを別のゾーンに配置する

ここでは、すべてのプライマリーが特定のゾーンに配置され、すべてのレプリカが別のゾーンに配置されます。これにより、1 つのプライマリーと 1 つの非同期複製を持つことになります。すべてのレプリカはゾーン A に、プライマリーはゾーン B に配置されます。

```
<?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="library">
  <mapSet name="ms1" numberOfPartitions="13" minSyncReplicas="0"
   maxSyncReplicas="0" maxAsyncReplicas="1">
   <map ref="book" />
   <zoneMetadata>
    <shardMapping shard="P" zoneRuleRef="primaryRule"/>
    <shardMapping shard="A" zoneRuleRef="replicaRule"/>
    <zoneRule name ="primaryRule">
     <zone name="A" />
    </zoneRule>
    <zoneRule name="replicaRule">
     <zone name="B" />
      </zoneRule>
     </zoneMetadata>
    </mapSet>
  </objectgridDeployment>
</deploymentPolicy>
```

ここには、プライマリー用に 1 つ (P)、レプリカ用に 1 つ (A) という 2 つのルールがあります。

広域ネットワーク (WAN) 上のゾーン

低速のネットワーク相互接続を使用する複数のビルまたはデータ・センターにまたがって、1 つの eXtreme Scale をデプロイしたい場合があります。ネットワーク接続が低速になれば、それだけ処理能力が低下し、接続待ち時間が長くなります。このモードでは、ネットワーク輻輳やその他の要因のために、ネットワーク分割の可能性も増します。eXtreme Scale は、以下の方法でこの厳しい環境に対処します。

ゾーン間のハートビート処理の制限

コア・グループにグループ化された Java 仮想マシンは、互いにハートビートを実行します。カタログ・サービスが Java 仮想マシンをグループに編成した場合、こうしたグループはゾーンをまたぎません。そのグループ内のリーダーがメンバーシップ情報をカタログ・サービスにプッシュします。カタログ・サービスは報告された障害を検証してから、アクションを実行します。問題のある Java 仮想マシンとの接続を試みて、この処理を実行します。カタログ・サービスは、誤障害検出を認めた場合、何のアクションも実行しません。コア・グループ区画が短時間で回復するためです。

またカタログ・サービスは、定期的にコア・グループ・リーダーに低速でハートビートを行い、コア・グループ分離の症状を処理します。

グリッドのタイ・ブレーカーとしてのカタログ・サービス

カタログ・サービスは、eXtreme Scale グリッド用のタイ・ブレーカーです。カタログ・サービスがそのグリッドに関係する他の要因とは無関係に作動することが絶対不可欠です。カタログ・サービスは、Java 仮想マシンの固定セットに対して実行し、選択したプライマリーからそのセット内の他のすべての Java 仮想マシンにデータを複製します。カタログ・サービスは、グリッドから分離される可能性が少なくなるように物理ゾーンまたはデータ・センター間に分散する必要があります。そうすると、予想した障害シナリオを切り抜けられるようになります。

カタログ・サービスは、べき等操作または回復可能操作を使用して、グリッド内のコンテナー Java 仮想マシンと通信します。この通信は、IIOP を使用して行われます。カタログ・サービスでのすべての状態変更は、カタログ・サービスをホスティングする現行メンバー間に同期的に複製されます。この複製が成功するのは、大多数の Java 仮想マシンが変更を受け入れた場合のみです。つまり、カタログ・サービスが区画に分割されている場合、大多数区画のみが変更をコミットできるということです。コマンドを作成した状態変更がコミットされた場合、サービス・プライマリーのみがコマンドをコンテナー Java 仮想マシンに送信します。つまり、少数側の区画は、コンテナーにその状態を転送したり、コマンドを発行したりできないということです。

カタログ・サービスが区画に分割されても、グリッドの機能が停止することはありません。グリッドは、クライアント要求を受け入れ、操作を実行します。大多数側のカタログ・サービス区画がない場合、グリッドの障害は、カタログ・サービスで多数を占める区画が発生するまで回復されません。この結果、障害が発生した場合に回復が長時間遅延されると、カタログ・サービスで大多数区画が生じるまで、一定の区画がオフラインになります。

コア・グループ・リーダーの Java 仮想マシンは、メンバーシップ変更をカタログ・サービスに報告します。カタログ・サービスは区画に分割されている場合、カタログ・サービスの更新された経路テーブルをプッシュ・バックします。少数側の区画のこうした経路テーブルには、プライマリーのロケーションは含まれません。リーダーが、考えられるすべてのカタログ・サービス Java 仮想マシンに渡って反復し、プライマリー区画を検出することが必要になります。これは、区画が解決されるのを待機中に、定期的に行われる必要があります。リーダーがプライマリーを持つ経

路テーブルを受信すると、保留中のアクションが大多数側のカタログ・サービスのプライマリーによって処理されることになります。

コア・グループが一定時間、カタログ・サービスのプライマリーと接続できない場合、そのコア・グループがグリッドの残りから物理的に切断されているか (少数側のカタログ・サービス区画によるものと考えられます)、カタログ・サービスが少数側の区画でスタック状態になっています。その違いを区別することは不可能です。大多数側のカタログ・サービス区画がある場合、そのカタログ・サービスは、コア・グループの切断という明白なロスから回復している可能性があります。これは、同じ区画に 2 つのプライマリー (既存の古いプライマリーとネットワークの残りにある新規プライマリー) が発生する結果になる場合があります。古いプライマリーとのネットワーク切断状態にある大多数側のカタログ・サービス区画には、古いプライマリーを強制終了する手段はありません。カタログ・サービスが回復し、切断されたコア・グループが新規プライマリーをディスカバーすると、カタログ・サービスは、2 つのプライマリーが存在することを通知します。さらに、前に切断されたコア・グループにすべての断片を削除するように指示します。これにより、バランシングが実行されます。

カタログ・サービスが 2 つの少数側の区画または 1 つの存続する少数側の区画に区画化されている場合、回復にはユーザーの介入が必要になります。Java Management Extensions (JMX) コマンドで、少数側の単一区画でアクションの実行が可能であることを指定する必要があります。他の少数側の区画は、確実に停止されるようにする必要があります。

優先ゾーン・ルーティング

優先ゾーン・ルーティングを使用すると、eXtreme Scale は、ユーザーの優先指定に基づくゾーンにトランザクションを直接送信できます。

WebSphere eXtreme Scale を使用すると、ObjectGrid の断片が配置されている場所に対してかなりの制御を行うことができます。いくつかの基本的なシナリオ、および、それに合わせたデプロイメント・ポリシーの構成方法について詳しくは、108 ページの『レプリカ配置のためのゾーンの使用』を参照してください。

優先ゾーン・ルーティングにより、eXtreme Scale クライアントは特定のゾーンあるいはゾーンのセットに対する偏向を指定できます。したがって、eXtreme Scale は、クライアント・トランザクションの経路指定をまずは優先ゾーンに試みてから、他のゾーンに経路指定します。

優先ゾーン・ルーティングの要件

優先ゾーン・ルーティングを試行する前に考慮すべき要素がいくつかあります。アプリケーションで、シナリオの要件を満たすことができることを確認してください。

優先ゾーン・ルーティングを活用するためには、コンテナーごとの区画の配置が必要です。この配置ストラテジーはセッション・データを ObjectGrid に保管しようとしているアプリケーションには最適です。 WebSphere eXtreme Scale のデフォルトの区画配置ストラテジーは固定区画です。トランザクションのコミット時にキーがハッシュされて、固定区画配置を使用する際に、マップのキー値ペアがどの区画に納められるかが決まります。

コンテナーごとの配置により、データは、トランザクション・コミット時に SessionHandle を介してランダムに区画に割り当てられます。 ObjectGrid からデータを取得するためには、この SessionHandle を再構成できなければなりません。

ゾーンを使用するとプライマリーと複製が入るドメイン内の場所に対する制御を強化できるため、複数ゾーン・デプロイメントは、データが物理的に複数のロケーションに存在する場合に有効です。地理的にプライマリーと複製を分離させることは、1 つのデータ・センターの壊滅的な損失でも、データのアベイラビリティーには影響を与えないことを確実にする手段です。

データが複数ゾーン・トポロジーに散在される場合、クライアントもそのトポロジーに散在されると考えられます。クライアントをそれぞれのローカル・ゾーンあるいはデータ・センターにルーティングすることには、ネットワーク待ち時間の削減という明確なパフォーマンス上の利点があります。可能であれば、このシナリオを活用してください。

優先ゾーン・ルーティングのトポロジーの構成

次のシナリオを考えてみます。データ・センターが 2 つ、Chicago と London にあります。クライアントの応答時間を最小にするために、クライアントはローカル・データ・センターに対してデータの読み取りと書き込みを行うようにします。

トランザクションが各ロケーションでローカルに書き込みが行われるためには、ObjectGrid のプライマリー断片は各データ・センターに配置される必要があります。さらに、クライアントは、ローカル・ゾーンへ経路指定するために個々のゾーンについて把握している必要があります。

コンテナーごとの配置により、新しいプライマリー断片は、開始された各コンテナーに配置されます。複製は、デプロイメント・ポリシーにより指定されたゾーンと配置のルールにしたがって配置されます。デフォルトで、複製は、プライマリーとは異なるゾーンに配置されます。このシナリオでは、以下のデプロイメント・ポリシーを考えてみます。

```
<?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="universe">
  <mapSet name="mapSet1" placementStrategy="PER_CONTAINER"
   numberOfPartitions="3" maxAsyncReplicas="1">
   <map ref="planet" />
  </mapSet>
 </objectgridDeployment>
</deploymentPolicy>
```

デプロイメント・ポリシーを使用して開始される各コンテナーで、3 つの新規プライマリーを受け取ります。各プライマリーには、非同期複製が 1 つ作成されます。各コンテナーを適切なゾーン名で開始します。コンテナーを startOgServer スクリプトを使用して起動する場合、-zone パラメーターを使用します。

Chicago のコンテナー・サーバーの場合、以下のようにします。

- UNIX Linux

```
startOgServer.sh s1 -objectGridFile ../xml/universeGrid.xml
-deploymentPolicyFile ../xml/universeDp.xml
-catalogServiceEndpoints MyServer1.company.com:2809
-zone Chicago
```

- Windows

```
startOgServer.bat s1 -objectGridFile ../xml/universeGrid.xml
-deploymentPolicyFile ../xml/universeDp.xml
-catalogServiceEndpoints MyServer1.company.com:2809
-zone Chicago
```

ご使用のコンテナーが WebSphere Application Server で稼働している場合、ノード・グループを作成してそれに「ReplicationZone」という接頭部を付けた名前を指定します。そのようにしたノード・グループ内のノードで稼働中のサーバーは、適切なゾーンに配置されます。例えば、Chicago ノードで実行中のサーバーは、「ReplicationZoneChicago」という名前のノード・グループに含まれる可能性があります。 108 ページの『レプリカ配置のためのゾーンの使用』 の「WebSphere Extended Deployment ノードのゾーンとの関連付け」を参照してください。

Chicago ゾーンのプライマリーは、その複製が London ゾーンに入ります。 London ゾーンのプライマリーは、その複製が Chicago ゾーンに入ります。

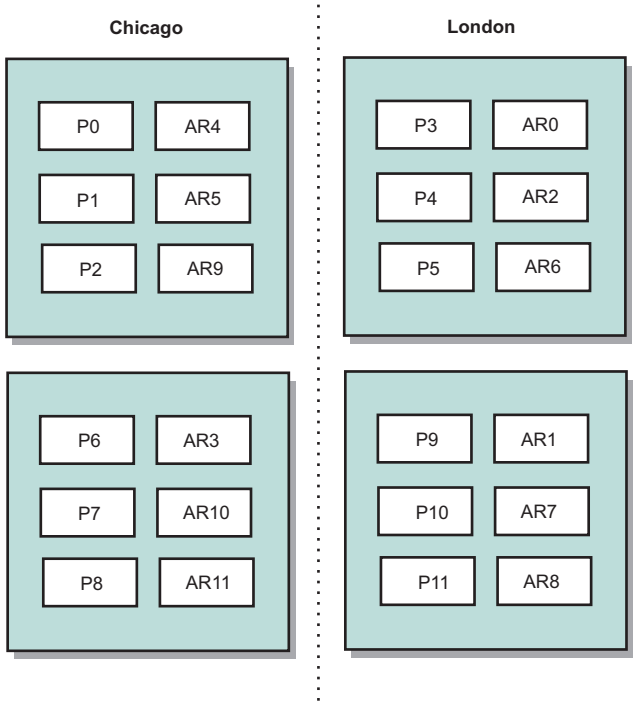


図 39. ゾーン内のプライマリーと複製

クライアントの優先ゾーンを設定します。この情報は、複数の異なる方法のいずれかで提供することができます。最もわかりやすい方法は、クライアント・プロパティー・ファイルをクライアント JVM に提供することです。objectGridClient.properties という名前のファイルを作成し、それが確実にクラスパスに入るようにします。 詳しくは、「管理ガイド」のクライアント・プロパティー・ファイルに関するトピックを参照してください。

このファイルに preferZones プロパティーを組み込みます。このプロパティー値を適切なゾーンに設定します。 Chicago のクライアントは objectGridClient.properties ファイルに以下を含みます。

```
preferZones=Chicago
```

London のクライアントのプロパティー・ファイルには、以下が含まれます。

```
preferZones=London
```

このプロパティーにより、各クライアントがトランザクションを可能な限りそのローカル・ゾーンに経路指定するように指示します。ローカル・ゾーンのプライマリーに挿入されるデータは、非同期で外部のゾーンに複製されます。

SessionHandle を使用してローカル・ゾーンに経路指定する

コンテナーごとの配置ストラテジーでは、ObjectGrid 内のキー値ペアのロケーションを決定する場合にハッシュ・ベースのアルゴリズムを使用しません。この配置ストラテジーの使用時は、ObjectGrid では、トランザクションが正しいロケーションに確実に経路指定されるように SessionHandles を活用する必要があります。トランザクションがコミットされると、SessionHandle が Session にバインドされます (まだ設定されていない場合)。また、トランザクションをコミットする前に Session.getSessionHandle を呼び出すことによって SessionHandle を Session にバインドすることができます。以下のコード・スニペットは、トランザクションのコミット前に SessionHandle がバインドされる場合を示しています。

```
Session ogSession = objectGrid.getSession();

// binding the SessionHandle
SessionHandle sessionHandle = ogSession.getSessionHandle();

ogSession.begin();
ObjectMap map = ogSession.getMap("planet");
map.insert("planet1", "mercury");

// tran is routed to partition specified by SessionHandle
ogSession.commit();
```

上記のコードは、Chicago データ・センターのクライアントで実行されると想定します。このクライアントの preferZones 属性は Chicago に設定されているため、このトランザクションは、Chicago ゾーンのプライマリー区画 0、1、2、6、7、または 8 のいずれかに経路指定されることになります。

この SessionHandle は、このコミット済みデータを保管する区画に戻るパスになります。コミット済みデータを含む区画に戻るためには、SessionHandle が再利用または再構成されて、Session に対して設定される必要があります。

```
ogSession.setSessionHandle(sessionHandle);
ogSession.begin();

// value returned will be "mercury"
String value = map.get("planet1");
ogSession.commit();
```

このトランザクションは、挿入トランザクション時に作成された SessionHandle を再利用するため、取得トランザクションは挿入済みデータを含む区画に対して経路指定されます。 SessionHandle が設定されていないと、このトランザクションは、以前に挿入されたデータを取得できません。

コンテナーおよびゾーン障害がゾーン・ベースのルーティングにどのように影響するか

preferZones プロパティーが設定されているクライアントでは、そのトランザクションのすべてが、通常の環境下で指定のゾーン (複数可) に経路指定されます。ただし、コンテナーの損失があると、外部ゾーンの複製がプライマリーにレベル上げされます。ローカル・ゾーンの区画に既に経路指定されていたクライアントは、以前に挿入されたデータを取得するために強制的に外部ゾーンに入れられる可能性があります。

次のシナリオを考えてみます。 Chicago ゾーンの 1 コンテナーが損失したとします。このコンテナーには区画 0、1、および 2 のプライマリーが既に格納されています。 London ゾーンでこれらの区画の複製をホスティングしていたため、これらの区画の新しいプライマリーは London ゾーンに行きます。

フェイルオーバーになった区画のいずれかを位置指定する SessionHandle を使用している Chicago クライアントは、今度は London に経路指定されます。新しい SessionHandle を使用する Chicago クライアントは、Chicago ベースのプライマリーに経路指定されます。

同様に、Chicago ゾーン全体が損失した場合、London ゾーンのすべての複製がプライマリーになります。この場合、すべての Chicago クライアントは、そのトランザクションを London に経路指定します。

フェイルオーバー検出のタイプ

WebSphere eXtreme Scale は、障害検出に 2 つの方法を用います。

ハートビート処理

1. Java 仮想マシン間ではソケットがオープン状態のままなので、ソケットが予想外に閉じると、この予想外のクローズはピア Java 仮想マシンの障害として検出されます。この検出機能は、Java 仮想マシンが極端に早く終了したなどの障害事例をキャッチし、また、こうしたタイプの障害からの回復を通常 1 秒以内に可能にします。
2. その他のタイプの障害には、オペレーティング・システム・パニック、物理サーバー障害、ネットワーク障害などがあります。こうした障害は、ハートビート処理を使用して対処されます。

プロセスのペア間では定期的にハートビートが送信されます。一定数のハートビートが欠落すると、障害とみなされます。この方法は、N*M 秒で障害を検出します。N は欠落ハートビートの数で、M はハートビートの設定間隔です。直接に M と N を指定することはサポートされておらず、代わりにスライダー・メカニズムを使用してテストされる一定範囲の M と N の組み合わせを指定できるようになっています。

障害

プロセスに障害が起きるのには、いくつかの場合があります。何らかのリソース限界に達したり (例えば、最大ヒープ・サイズ)、プロセス制御ロジックがプロセスを強制終了したといった理由で、プロセスに障害が起こることがあります。オペレーティング・システムに障害が起きると、システム上で実行中のすべてのプロセスが

失われます。ネットワーク・インターフェース・カード (NIC) などの頻繁には障害が起きないハードウェアに障害が起きると、オペレーティング・システムがネットワークから切断されます。さらに多くのポイントで障害が起きると、プロセスが使用不可になります。このような状況において、これらすべての障害は、プロセス障害または接続不良の 2 つの障害タイプのうちのいずれかに分類されます。

プロセス障害

WebSphere eXtreme Scale は、プロセス障害に非常に迅速に反応します。プロセスに障害が起きると、オペレーティング・システムは、プロセスが使用していたリソースの残りすべてをクリーンアップする必要が生じます。このクリーンアップには、ポートの割り当ておよび接続が含まれています。プロセスに障害が起きると、信号はそのプロセスによって使用されていた接続を通して送信され、各接続がクローズされます。これらの信号を使用し、障害の起きたプロセスに接続されている他のプロセスによって即時にプロセス障害が検出されます。

接続不良

オペレーティング・システムが切断されると、接続不良が発生します。その結果として、オペレーティング・システムは信号を他のプロセスに送信することができなくなります。接続不良が発生する理由はいくつかありますが、それらの理由は、ホスト障害と孤立化の 2 つのカテゴリーに分割されます。

ホスト障害

マシンの電源コンセントのプラグが抜かれると、即座に動作しなくなります。

孤立化

このシナリオは、使用不可であると見なされたプロセスが実際には使用不可ではないため、ソフトウェアが正しく対処することが最も難しい障害の状態です。

コンテナー障害

コンテナー障害は、通常コア・グループ・メカニズムを通してピア・コンテナーによって発見されます。コンテナーまたはコンテナーのセットに障害が起きると、カタログ・サービスにより、そのコンテナーにホスティングされていた断片が移行されます。カタログ・サービスにより、非同期のレプリカに移行する前に最初に同期複製が検索されます。プライマリー断片が新規ホスト・コンテナーに移行された後で、カタログ・サービスにより、欠落しているレプリカの新規ホスト・コンテナーが検索されます。

注: コンテナー孤立化 - コンテナーが使用不可であることが検出されると、カタログ・サービスによりコンテナーの断片がコンテナーから移行されます。これらのコンテナーが使用可能になると、カタログ・サービスは、通常の開始フローの場合と同じように、これらのコンテナーを配置可能とみなします。

コンテナー・フェイルオーバー検出までの待ち時間

障害は、ソフトとハードの障害に分類されます。ソフト障害の原因は、一般にプロセスの障害です。そのような障害はオペレーティング・システムによって検出されます。オペレーティング・システムでは、ネットワーク・ソケットなどの使用され

たリソースを非常に迅速にリカバリーできます。ソフト障害の場合、標準的な障害検出までの時間は、1 秒未満です。ハード障害の場合、デフォルトのハートビート調整を使用すると、検出まで最長 200 秒かかることもあります。そのような障害は、物理的なマシンの破損、ネットワーク・ケーブル切断、オペレーティング・システム障害などです。したがって、eXtreme Scale がハード障害を検出するには、構成可能なハートビートに頼らざるを得ません。ハード障害を検出するまでの時間を短縮する方法について詳しくは、 118 ページの『フェイルオーバー検出のタイプ』を参照してください。

複数のコンテナー障害

プロセスが失われるとプライマリーおよびレプリカの両方が失われるため、レプリカはプライマリーとして同じプロセスに配置するべきではありません。デプロイメント・ポリシーにより、カタログ・サービスがレプリカをプライマリーとして同じマシンに配置できるかどうかを判別するのに使用する属性が定義されます。単一マシンの開発環境においては、2 つのコンテナーを所有でき、それらの間でレプリカを生成できます。しかし、実動環境では、そのホストを失うことにより両方のコンテナーを失うことになるため、単一マシンの使用では不十分です。単一マシンでの開発モードと複数のマシンを使用する実動モード間でモードを変更するには、デプロイメント・ポリシー構成ファイルで開発モードを無効にします。

カタログ・サービス障害

カタログ・サービス・グリッドは、eXtreme Scale グリッドであるため、これもコンテナー障害プロセスと同じ方法でコア・グループ化のメカニズムを使用します。主な相違点は、カタログ・サービス・グリッドでは、コンテナーに使用するカタログ・サービス・アルゴリズムの代わりに、プライマリー断片の定義にピア選択プロセスを使用する点です。

配置サービスおよびコア・グループ化サービスは多くのサービスのうちの 1 つであり、ロケーション・サービスおよび管理はすべての場所で実行されているということに注意してください。配置サービスおよびコア・グループ化サービスは、システムをレイアウトする必要があるため別のものです。ロケーション・サービスおよび管理は読み取り専用サービスであり、スケーラビリティーを提供するためにあらゆる場所に存在します。

カタログ・サービスは複製を使用して、独自の障害限界を設定します。カタログ・サービス・プロセスに障害が起きると、サービスが再始動され、システムを必要なレベルの可用性に復元します。カタログ・サービスをホスティングしているすべてのプロセスで障害が起こると、eXtreme Scale から重要なデータが失われます。この障害により、すべてのコンテナーの再始動が必要になります。カタログ・サービスは多くのプロセスで実行されているため、この障害は起きる可能性のないイベントです。ただし、単一のボックスですべてのプロセスを実行している場合は、単一のブレード・シャーシ内、または単一のネットワーク・スイッチで、障害が起きる可能性があります。カタログ・サービスをホスティングしているボックスから共通の障害モードを除去して、障害が起きる可能性を減らします。

高可用性カタログ・サービス

カタログ・サービスは、使用しているカタログ・サーバーのグリッドであり、eXtreme Scale 環境内のすべてのコンテナーのトポロジー情報を保持します。カタログ・サービスは、すべてのクライアントの平衡化とルーティングを制御します。eXtreme Scale をメモリー内のデータベース処理スペースとしてデプロイするには、高可用性を実現するためにカタログ・サービスをグリッドにクラスター化する必要があります。

カタログ・サービスのコンポーネント

複数のカタログ・サーバーが始動すると、サーバーのいずれか 1 つがマスター・カタログ・サーバーとして選択されます。マスター・カタログ・サーバーは、Internet Inter-ORB Protocol (IIOP) ハートビートを受信し、カタログ・サービスまたはコンテナーの変更に応じて、システム・データの変更を処理します。

クライアントがいずれかのカタログ・サーバーにアクセスすると、カタログ・サーバー・グリッドのルーティング・テーブルは、共通オブジェクト・リクエスト・ブローカー・アーキテクチャー (CORBA) サービス・コンテキストを通してクライアントに伝搬されます。

少なくとも 3 つのカタログ・サーバーを構成します。構成がゾーンに分かれている場合、ゾーンごとに 1 つずつカタログ・サーバーを構成することができます。

eXtreme Scale サーバーおよびコンテナーが、カタログ・サーバーの 1 つにアクセスすると、カタログ・サーバー・グリッドのルーティング・テーブルが、CORBA サービス・コンテキストを通して eXtreme Scale サーバーおよびコンテナーにも伝搬されます。また、アクセスされたカタログ・サーバーがその時点でマスター・カタログ・サーバーでなかった場合、要求は現行マスター・カタログ・サーバーに自動的に転送され、カタログ・サーバーのルーティング・テーブルも更新されます。

注: カタログ・サーバー・グリッドとコンテナー・サーバー・グリッドは、まったく別のものです。カタログ・サーバー・グリッドは、システム・データの高可用性のためのものです。コンテナー・グリッドは、ユーザーのデータの高可用性、スケーラビリティー、およびワークロード管理を目的としています。したがって、カタログ・サーバー・グリッドのルーティング・テーブルとサーバー・グリッド断片のルーティング・テーブルという 2 つの異なるルーティング・テーブルが存在します。

カタログの責務は、一連のサービスに分割されます。コア・グループ・マネージャーは、ヘルスをモニターするためのピアのグループ化を実行し、配置サービスは、割り振りを行い、管理サービスは、管理のためのアクセスを提供し、ロケーション・サービスは局所性を管理します。

カタログ・グリッド・デプロイメント

コア・グループ・マネージャー

カタログ・サービスは、高可用性マネージャー (HA マネージャー) を使用して、可用性モニタリングのためにプロセスをグループ化します。各プロセス・グループが、コア・グループです。eXtreme Scale では、コア・グループ・マネージャーが動

的にプロセスをグループ化します。これらのプロセスは、スケーラビリティーのために小さく維持されます。各コア・グループはそれぞれリーダーを選出します。リーダーには、個々のメンバーに障害が発生したときに状況をコア・グループ・マネージャーに送信するという責任が追加されます。同じ状況のメカニズムは、グループのすべてのメンバーで障害が起こったときに (これにより、リーダーとの通信に障害が発生します)、それを検出するために使用されます。

コア・グループ・マネージャーは完全に自動化されたサービスです。コンテナーを少数のサーバーからなるグループに編成する責務を持ち、そのグループは自動で緩やかに統合して ObjectGrid を形成します。コンテナーは、カタログ・サービスへの初回接続時、新規または既存のグループに割り当てられるまで待機します。eXtreme Scale はそうした多くのグループから構成されており、このグループ化は重要なスケーラビリティー・イネーブラーです。各グループは Java 仮想マシンで構成されるグループであり、ハートビートを使用して、他のグループの可用性をモニターします。これらのグループ・メンバーの 1 つがリーダーに選出され、リーダーには可用性情報をカタログ・サービスにリレーする責務が追加され、再割り振りとルート転送により障害に対処できるようにします。

配置サービス

カタログ・サービスは、使用可能なすべてのコンテナーでの断片の配置を管理します。物理リソース間でのリソース・バランスの維持は、配置サービスの担当です。配置サービスは、個々の断片をホスト・コンテナーに割り振る責任を担います。それは、グリッド内で N 個の中から 1 つ選ばれたサービスとして実行されるため、実行中のサービスのインスタンスは必ず 1 つとなります。そのインスタンスに障害が起こると、別のプロセスが選出され、それが引き継ぎます。予備のために、カタログ・サービスの状態は、カタログ・サービスをホスティングするすべてのサーバーに複製されます。

管理

カタログ・サービスは、システム管理のための論理的なエントリー・ポイントでもあります。カタログ・サービスは、Managed Bean (MBean) をホストし、サービスが管理しているすべてのサーバーの Java Management Extensions (JMX) URL を提供します。

ロケーション・サービス

ロケーション・サービスは、探しているアプリケーションをホストするコンテナーを検索しているクライアント、およびホストされるアプリケーションを配置サービスに登録しようとしているコンテナーを検索しているクライアントの両方に対し、タッチ・ポイントとしての役割を果たします。ロケーション・サービスは、この機能をスケールアウトするために、すべてのグリッド・メンバーで実行されます。

カタログ・サービスは、一般的に定常状態でアイドルになるロジックをホストします。その結果として、カタログ・サービスがスケーラビリティーに与える影響はごくわずかです。サービスは、同時に使用可能になる多数のコンテナーにサービスを提供するために作成されています。可用性のために、カタログ・サービスをグリッドに構成します。

計画

カタログ・グリッドが開始されると、グリッドのメンバーは相互にバインドされます。カタログ構成を実行時に変更することはできないので、カタログ・グリッドのトポロジーは慎重に計画してください。エラー防止のため、グリッドはできるだけ広範囲に分散させてください。

カタログ・サーバー・グリッドの開始

WebSphere Application Server に組み込まれている eXtreme Scale コンテナーのスタンドアロン・カタログ・グリッドへの接続

WebSphere Application Server 環境に組み込まれている eXtreme Scale コンテナーを構成して、スタンドアロン・カタログ・グリッドに接続できます。アプリケーション・サーバーをカタログ・グリッドに接続するのと同じプロパティーを使用します。しかし、プロパティーは、サーバーでカタログのライフサイクルを管理しません。その代わり、プロパティーは、コンテナーがリモート・カタログ・グリッドを見つけることができるようにします。プロパティーの設定については、 「*管理ガイド*」内の WebSphere Application Server 環境でのカタログ・サービス・プロセスの開始についての説明を参照してください。

注: サーバー名の競合: このプロパティーは、eXtreme Scale カタログ・サーバーの開始だけでなく、そこに接続する目的でも使用されるため、カタログ・サーバーの名前をどの WebSphere Application Server とも同じ名前にすることはできません。

カタログ・サーバー・クォーラム

クォーラムとは、グリッドに対して配置操作を実行するために必要なカタログ・サーバーの最小数のことです。この最小数は、クォーラムがオーバーライドされていない限り、カタログ・サーバーのフルセットです。

重要な用語

以下は、eXtreme Scale に関するクォーラムの考慮事項に関連する用語のリストです。

- ブラウン・アウト: ブラウン・アウトとは、1 つ以上のサーバー間における一時的な接続喪失のことです。
- ブラック・アウト: ブラック・アウトとは、1 つ以上のサーバー間における完全な接続喪失のことです。
- データ・センター: データ・センターとは、地理的に配置されたサーバーの一群のことで、通常はサーバー同士がローカル・エリア・ネットワーク (LAN) で接続されています。
- ゾーン: ゾーンとは、何らかの物理的特性を共有するサーバーを 1 つのグループにまとめるために使用される構成オプションのことです。例えば、あるデータ・センター内のサーバーはすべてあるゾーン内でマークすることができます。
- ハートビート: ハートビートとは、Java 仮想マシンが稼働中であるかどうかを確認するために Java 仮想マシンを ping することをいいます。

トポロジー

このセクションでは、IBM WebSphere eXtreme Scale が信頼性の低いコンポーネントを含むネットワークでどう稼働するかについて説明します。このようなネットワークの例としては、複数のデータ・センターにまたがるネットワークがあります。

IP アドレス・スペース

WebSphere eXtreme Scale では、ネットワーク上のアドレス可能なすべての要素がネットワーク上のアドレス可能な他のすべての要素にスムーズに接続できるようなネットワークが必要となります。つまり、WebSphere eXtreme Scale は、フラット IP アドレス・ネーミング・スペースを必要とするとともに、WebSphere eXtreme Scale の要素をホストする Java 仮想マシン (JVM) が使用する IP アドレスおよびポート間を流れるすべてのトラフィックに対するすべてのファイアウォールを必要とします。

接続された LAN

各 LAN には WebSphere eXtreme Scale の要件のゾーン ID が割り当てられます。WebSphere eXtreme は単一ゾーン内の JVM を活発にハートビートします。そして欠落ハートビートがあると、カタログ・サービスがクォーラムを持っている場合にのみフェイルオーバー・イベントが発生します。

カタログ・サービス・グリッドとコンテナー・サーバー

グリッドとは類似した JVM の集合をいいます。カタログ・サービスはカタログ・サーバーから成るグリッドで、そのサイズは固定されています。ただし、コンテナー・サーバーの数は動的です。コンテナー・サーバーはオンデマンドで追加したり除去したりすることができます。データ・センターが 3 つある構成では、WebSphere eXtreme Scale はデータ・センターごとにカタログ・サービス JVM を 1 つずつ必要とします。

カタログ・サービス・グリッドはフル・クォーラム・メカニズムを使用します。つまり、グリッドのすべてのメンバーが任意のアクションに合意する必要があります。

コンテナー・サーバー JVM はゾーン ID でタグ付けされます。コンテナー JVM のグリッドは JVM から成る小さいコア・グループに自動的に分割されます。 1 つのコア・グループには同じゾーンからの JVM のみが含まれます。いかなる時点でも、異なるゾーンからの JVM が同じコア・グループに存在することはありません。

コア・グループはそのメンバー JVM の障害の検出を活発に試みます。いかなる時点でも、コア・グループのコンテナー JVM は広域ネットワークと同様にリンクによって接続された複数の LAN にまたがってはなりません。つまり、コア・グループは異なるデータ・センターで実行されている同じゾーン内のコンテナーを持つことができません。

サーバー・ライフ・サイクル

カタログ・サーバーの始動

カタログ・サーバーは startOgServer コマンドを使用して始動されます。デフォルトではクォーラム・メカニズムが使用不可になっています。クォーラムを使用可能にするためには、startOgServer コマンドで -quorum 使用可能フラグを渡すか、または enableQuorum=true プロパティーをプロパティー・ファイルに追加してください。すべてのカタログ・サーバーに対して同じクォーラム設定を指定する必要があります。

```
# bin/startOgServer cat0 –serverProps objectGridServer.properties
```

objectGridServer.properties ファイル

```
catalogClusterEndPoints=cat0:cat0.domain.com:6600:6601,
cat1:cat1.domain.com:6600:6601
catalogServiceEndPoints= cat0.domain.com:2809, cat1.domain.com:2809
enableQuorum=true
```

コンテナー・サーバーの始動

コンテナー・サーバーは startOgServer コマンドを使用して始動されます。これらのサーバーは、複数のデータ・センターにまたがるグリッドを実行する際、ゾーン・タグを使用してサーバーが存在するデータ・センターを識別する必要があります。グリッド・サーバー上にゾーンを設定することにより、WebSphere eXtreme Scale はデータ・センターまでの範囲内にあるサーバーのヘルスをモニターし、データ・センター間のトラフィックを最小化することができます。

```
# bin/startOgServer gridA0 –serverProps objectGridServer.properties –
objectgridfile xml/objectgrid.xml –deploymentpolicyfile xml/
deploymentpolicy.xml
```

objectGridServer.properties ファイル

```
catalogServiceEndPoints= cat0.domain.com:2809, cat1.domain.com:2809
zoneName=ZoneA
```

グリッド・サーバーのシャットダウン

グリッド・サーバーは stopOgServer コマンドを使用して停止されます。保守のためにデータ・センター全体をシャットダウンする際は、そのゾーンに属するすべてのサーバーのリストを渡してください。そうすると、分解されているゾーンから残存しているゾーンへの滑らかな状態遷移が可能になります。

```
# bin/startOgServer gridA0,gridA1,gridA2 –catalogServiceEndPoints
cat0.domain.com:2809,cat1.domain.com:2809
```

障害検出

WebSphere eXtreme Scale は異常ソケット閉鎖イベントを通じてプロセス・デスを検出します。プロセスが終了すると、そのことが直ちにカタログ・サービスに知らされます。ブラック・アウトは欠落ハートビートを通じて検出されます。 WebSphere eXtreme Scale は、クォーラム実装を使用することで、データ・センターでのブラウン・アウト条件から防護します。

ハートビート実装

このセクションでは、WebSphere eXtreme Scale で活性検査がどのようにして実装されるかについて説明します。

コア・グループ・メンバーのハートビート

カタログ・サービスはコンテナー JVM を限られたサイズのコア・グループに配置します。コア・グループは、2 つの方法を使用して、そのメンバーの障害の検出を試みます。 JVM のソケットが閉じられていると、その JVM はデッド状態にあると見なされます。さらに、各メンバーは構成によって決定されたペースでこれらのソケットをハートビートします。ある JVM が構成された最大時間内にこれらのハートビートに応答しないと、その JVM はデッド状態にあると見なされます。

コア・グループのメンバーの中から常に 1 つだけがリーダーに選ばれます。コア・グループ・リーダー (CGL) は、コア・グループが活動中であることをカタログ・サービスに定期的に知らせるとともに、メンバーシップの変更をカタログ・サービスに報告する必要があります。メンバーシップの変更としては、JVM の障害や新しく追加された JVM のコア・グループへの参加などが考えられます。

コア・グループ・リーダーがカタログ・サービス・グリッドのメンバーと連絡できないと、コア・グループ・リーダーは再試行し続けます。

カタログ・サービス・グリッドのハートビート

カタログ・サービスは静的メンバーシップおよびクォーラム・メカニズムを持つ専用コア・グループに似ています。そして、通常のコア・グループと同じ方法で障害検出を行います。ただし、その振る舞いはクォーラム・ロジックを含むように変更されています。さらに、カタログ・サービスではそれほど活発でないハートビートの構成が使用されます。

コア・グループのハートビート

カタログ・サービスは、コンテナー・サーバーに障害が発生したとき、そのことを知る必要があります。各コア・グループは、コンテナー JVM の障害を突き止め、コア・グループ・リーダーを通じてこれをカタログ・サービスに報告する役割を担っています。コア・グループの全メンバーが完全に障害を起こす可能性もあります。コア・グループ全体で障害が起こった場合は、カタログ・サービスがその障害を検出しなければなりません。

カタログ・サービスがあるコンテナー JVM を障害と判断した後で、そのコンテナーが活動中と報告された場合は、このコンテナー JVM に対して WebSphere eXtreme Scale コンテナー・サーバーをシャットダウンするよう指示が出されます。この状態の JVM は xsadmin 照会では不可視になります。こうなったことを示すメッセージがコンテナー JVM のログに記録されます。これらの JVM は手動で再始動する必要があります。

クォーラム損失イベントが発生した場合は、クォーラムが再確立されるまでハートビートは中断されます。

カタログ・サービス・クォーラムの振る舞い

通常、カタログ・サービスのメンバーは完全な接続性を備えています。カタログ・サービス・グリッドは JVM の静的集合です。 WebSphere eXtreme Scale は、カタ

ログ・サービスのすべてのメンバーが常時オンラインであることを想定しています。カタログ・サービスは、カタログ・サービスがクォーラムを持っている間だけコンテナー・イベントに応答します。

カタログ・サービスがクォーラムを失うと、カタログ・サービスはクォーラムが再確立されるのを待ちます。カタログ・サービスは、クォーラムを持っていない間は、コンテナー・サーバーからのイベントを無視します。 WebSphere eXtreme Scale はクォーラムが再確立されることを想定しているため、コンテナー・サーバーはクォーラム不在の間にカタログ・サーバーが拒否した要求を再試行します。

次のメッセージはクォーラムが失われたことを示しています。このメッセージはご使用のカタログ・サービス・ログに入っています。

```
CWOBJ1254W: カタログ・サービスがクォーラムを待機しています。
```

WebSphere eXtreme Scale は、以下の理由によってクォーラムの損失を予想します。

- カタログ・サービス JVM メンバーの障害
- ネットワークのブラウン・アウト
- データ・センター損失

`stopOgServer` を使用してカタログ・サーバー・インスタンスを停止しても、システムはこのサーバー・インスタンスが停止したこと (これは JVM 障害やブラウン・アウトとは異なる) を知っているため、これがクォーラム損失の原因となることはありません。

JVM 障害によるクォーラム損失

障害を起こしたカタログ・サーバーはクォーラム損失の原因となります。そうなった場合は、できるだけ迅速にクォーラムがオーバーライドされるようにしてください。障害を起こしたカタログ・サービスは、クォーラムがオーバーライドされるまで、グリッドに再参加できません。

ネットワーク・ブラウン・アウトによるクォーラム損失

WebSphere eXtreme Scale はブラウン・アウトの可能性を予想できる設計になっています。ブラウン・アウトとは、データ・センター間の接続が一時的に失われた状態をいいます。これは通常、本質的に一過性のものであり、数秒あるいは数分足らずでブラウン・アウトは解消するはずです。ブラウン・アウト中、WebSphere eXtreme Scale は通常動作の維持を試みますが、ブラウン・アウトは 1 つの障害イベントと見なされます。この障害は修正されることが想定されており、WebSphere eXtreme Scale のアクションを必要とせずに通常動作が再開されます。

長時間に及ぶブラウン・アウトは、ユーザーの介入がある場合にのみブラック・アウトに分類できます。このイベントをブラック・アウトに分類するためには、ブラウン・アウトの一方の側でクォーラムをオーバーライドする必要があります。

カタログ・サービス JVM の循環

stopOgServer を使用してカタログ・サーバーが停止された場合は、クォーラムを持つサーバーが 1 つ減少します。つまり、残りのサーバーはまだクォーラムを持っています。このカタログ・サーバーを再始動すると、クォーラムは元の数に戻ります。

クォーラム損失の影響

クォーラムが失われると同時にコンテナー JVM が障害を起こした場合は、ブラウン・アウトが回復するまでリカバリーが行われないか、または (ブラック・アウトの場合) お客様がクォーラム・オーバーライド・コマンドを実行します。WebSphere eXtreme Scale はクォーラム損失イベントとコンテナー障害を二重障害と見なしますが、これはまれにしか起こりません。つまり、クォーラムが復元されてリカバリーが正常に行われるようになるまで、アプリケーションは障害を起こした JVM に保管されたデータへの書き込みアクセスを失う可能性があります。

同様に、クォーラム損失イベントの発生中にコンテナーを開始しようとしても、コンテナーは開始されません。

クォーラムの損失中に完全クライアント接続が許可されます。クォーラム損失イベント中にコンテナー障害も接続問題も起こらなければ、クライアントはコンテナー・サーバーと完全に対話することができます。

ブラウン・アウトが発生すると、クライアントによっては、ブラウン・アウトが解消されるまで、データのプライマリーまたは複製コピーにアクセスできない場合があります。

ブラウン・アウト・イベント中でもクライアントが少なくとも 1 つのカタログ・サービス JVM に到達できるように各データ・センターにはカタログ・サービス JVM が存在するはずなので、新規のクライアントを開始することができます。

クォーラムのリカバリー

何らかの理由によってクォーラムが失われた場合は、クォーラムが再確立される際、リカバリー・プロトコルが実行されます。クォーラム損失イベントが発生すると、コア・グループに対する活性検査はすべて中断され、障害報告も無視されます。クォーラムが元に戻ると、カタログ・サービスはすべてのコア・グループに対して活性検査を行い、直ちにそれらのメンバーシップを決定します。障害として報告されたコンテナー JVM でそれまでホストされていた断片は、いずれもこの時点でリカバリーされます。プライマリー断片が失われた場合は、残存している複製がプライマリーに昇格します。複製断片が失われた場合は、残存物の上に追加の複製が作成されます。

クォーラムのオーバーライド

これは、データ・センター障害が発生した場合にだけ使用するようにしてください。カタログ・サービス JVM の障害またはネットワークのブラウン・アウトのためにクォーラムが失われた場合は、カタログ・サービス JVM が再始動されるか、またはネットワークのブラウン・アウトが解消されると、リカバリーが自動的に行われます。

データ・センターの障害に通じているのは管理者のみです。 WebSphere eXtreme Scale はブラウン・アウトとブラック・アウトを同様に扱います。これらの障害が発生した場合は、クォーラムをオーバーライドする xsadmin コマンドを使用して eXtreme Scale 環境に通知する必要があります。そうすると、カタログ・サービスに対して、現在のメンバーシップでクォーラムが達成されて、完全リカバリーが行われると想定するように指示が出されます。クォーラム・オーバーライド・コマンドを発行したときは、障害のあるデータ・センター内の JVM が実際に障害を起こしていて、しかもリカバリーされないことを保証していることになります。

以下のリストは、クォーラムのオーバーライドに関するいくつかのシナリオを考慮したものです。 A、B、および C という 3 つのカタログ・サーバーがあるとします。

- ブラウン・アウト: C が一時的に分離されているブラウン・アウトがあるとします。カタログ・サービスはクォーラムを失い、ブラウン・アウトが解消されるのを待ちます。解消された時点で、C はカタログ・サービス・グリッドに再参加し、クォーラムが再確立されます。この間、アプリケーションはいかなる問題も検出しません。
- 一時障害: ここでは、C が障害を起こし、カタログ・サービスがクォーラムを失ったため、クォーラムをオーバーライドする必要があります。クォーラムが再確立されると、C を再始動することができます。 C は、再始動されたとき、カタログ・サービス・グリッドに再参加します。この間、アプリケーションはいかなる問題も検出しません。
- データ・センター障害: データ・センターが実際に障害を起こし、しかもネットワーク上で分離されていることを確認します。次に、xsadmin クォーラム・オーバーライド・コマンドを発行します。そうすると、残存している 2 つのデータ・センターが、障害を起こしたデータ・センターでホストされていた断片を置き換えることによって完全リカバリーを実行します。カタログ・サービスは現在、A および B のフル・クォーラムで実行しています。アプリケーションは、ブラック・アウトが始まってからクォーラムがオーバーライドされるまでの間に、遅延や例外を検出することがあります。クォーラムがオーバーライドされると、グリッドがリカバリーされ、通常動作が再開されます。
- データ・センター・リカバリー: 残存しているデータ・センターは、オーバーライドされたクォーラムで既に実行されています。 C を含むデータ・センターが再始動されたならば、そのデータ・センターにあるすべての JVM も再始動する必要があります。そうすると、C は既存のカタログ・サービス・グリッドに再参加し、クォーラムはユーザーの介入なしで通常状態に戻ります。
- データ・センターの障害およびブラウン・アウト: C を含むデータ・センターが障害を起こしました。残りのデータ・センターでは、クォーラムがオーバーライドされてリカバリーされます。 A と B の間でブラウン・アウトが発生した場合は、通常のブラウン・アウト・リカバリー・ルールが適用されます。ブラウン・アウトが解消されると、クォーラムが再確立され、クォーラム損失からの必要なリカバリーが実行されます。

コンテナーの振る舞い

このセクションでは、クォーラムが失われてからリカバリーされるまでの間、コンテナー・サーバー JVM がどのように振る舞うかについて説明します。

コンテナーは 1 つ以上の断片をホストします。断片は、特定の区画のプライマリーであるか複製であるか、そのいずれかです。カタログ・サービスが断片をコンテナーに割り当てると、カタログ・サービスから新しい指示が送られてくるまで、コンテナーはこの割り当てに従います。つまり、ブラウン・アウトのためにコンテナー内のプライマリー断片が複製断片と通信できない場合は、カタログ・サービスから新しい指示を受け取るまで、コンテナーは再試行し続けます。

ネットワーク・ブラウン・アウトが発生し、プライマリー断片が複製との通信を失うと、カタログ・サービスが新しい指示を出すまでコンテナーは接続を再試行します。

同期複製の振る舞い

接続が中断されている間でも、マップ・セットの minsync プロパティーと少なくとも同数の複製がオンラインである限り、プライマリーは新しいトランザクションを受け入れることができます。同期複製へのリンクが中断されているときにプライマリーで新しいトランザクションが処理された場合は、リンクが再確立された時点で、複製はクリアされ、プライマリーの現在の状態と再同期されます。

データ・センター間や WAN スタイルのリンクに対しては、同期複製を決して推奨しません。

非同期複製の振る舞い

接続が中断されている間でも、プライマリーは新しいトランザクションを受け入れることができます。プライマリーは変更内容を限界までバッファーに入れます。この限界に達する前に複製との接続が再確立された場合は、バッファーに入れられた変更内容を使用して複製が更新されます。限界に達すると、プライマリーはバッファーに入れられたリストを破棄します。そして複製が再接続されると、複製はクリアされて再同期されます。

クライアントの振る舞い

カタログ・サービス・グリッドがクォーラムを持っていてもいなくても、常時、クライアントはカタログ・サーバーに接続して、グリッドをブートストラップすることができます。クライアントは、経路テーブルを取得した上でグリッドと対話するために、カタログ・サーバー・インスタンスへの接続を試みます。ネットワークの接続性により、ネットワーク・セットアップが原因でクライアントが一部の区画と対話できなくなる場合があります。クライアントはローカル複製に接続してリモート・データを取得できますが、その場合はクライアントがそうするように構成されていなければなりません。クライアントは、該当するプライマリー区画が使用可能でない場合、データを更新できません。

SSL 使用不可

ObjectGrid 記述子 XML ファイルでトランザクション・タイムアウトを指定することを強くお勧めします。このタイムアウトを超えるまで、クライアントは 1 つのトランザクションを再試行します。タイムアウトが指定されていないと、クライアント・スレッドがブロックされる時間が不明なため、クライアントは無期限に再試行します。これは推奨できません。トランザクション・タイムアウトは予想される最大トランザクション時間の倍数に設定されるようにしてください。

さらに、orb.properties ファイルで ORB 要求タイムアウトを設定することを強くお勧めします。クライアントは、この最大時間が経過するまで、ブラウン・アウトされたソケットをブロックし続けます。要求が出されてからの経過時間がまだトランザクション・タイムアウトに満たない場合は、WebSphere eXtreme Scale が再び要求を試みます。合計経過時間がトランザクション・タイムアウトを超えるまで、クライアントは再試行し続けます。したがって、最大クライアント・ブロック時間は、トランザクション・タイムアウトに ORB 要求タイムアウトを加えたものとなります。

ORB 要求タイムアウトが指定されていないと、TCP スタックがブラウン・アウトされたソケットを閉じるまで、ORB がブロックされます。この時間は TCP スタックの調整によって異なり、TCP FIN_WAIT その他の関連パラメーター次第で数時間になる場合もあります。

xsadmin を含むクォーラム・コマンド

このセクションでは、クォーラムのさまざまな状況に役立つ xsadmin コマンドについて説明します。

クォーラム状況の照会

カタログ・サーバー・インスタンスのクォーラム状況は、xsadmin コマンドを使用して問い合わせることができます。

```
xsadmin –ch cathost –p 1099 –quorumstatus
```

起こり得る結果は 5 つあります。

- クォーラムが使用不可である: カタログ・サーバーがクォーラム使用不可モードで実行されています。これは開発モードまたは単一専用データ・センター・モードです。複数データ・センター構成の場合は、これをお勧めしません。
- クォーラムが使用可能で、かつカタログ・サーバーがクォーラムを持っている: クォーラムが使用可能で、しかもシステムが正常に機能しています。
- クォーラムは使用可能だが、カタログ・サーバーがクォーラムを待機している: クォーラムが使用可能で、かつクォーラムが失われています。
- クォーラムが使用可能で、かつクォーラムがオーバーライドされている: クォーラムが使用可能で、しかもクォーラムがオーバーライドされました。
- クォーラム状況が禁止である: ブラウン・アウトが発生したとき、カタログ・サーバーが 2 つの区画 (A および B) に分割され、カタログ・サーバー A のクォーラムがオーバーライドされました。ネットワーク区画は分解され、B 区画にあるサーバーが禁止されているため、JVM の再始動が必要です。この状況は、ブラウン・アウト中に B にあるカタログ JVM が再始動されてから、ブラウン・アウトが解消された場合にも起こります。

クォーラムのオーバーライド

xsadmin コマンドを使用してクォーラムをオーバーライドすることができます。残存しているカタログ・サーバー・インスタンスを使用することができます。ある残存物に対してクォーラムをオーバーライドする指示が出されると、それがすべての残存物に通知されます。これを行うための構文は次のとおりです。

```
xsadmin –ch cathost –p 1099 –overridequorum
```

診断コマンド

- クォーラム状況: 前のセクションで説明したとおりです。
- コア・グループ・リスト: これはすべてのコア・グループのリストを表示します。コア・グループのメンバーとリーダーが表示されます。

  ```
  xsadmin –ch cathost –p 1099 –coregroups
  ```
- サーバーの分解: このコマンドはグリッドからサーバーを手動で除去します。障害サーバーとして検出されたサーバーは自動的に除去されるので、これは通常不要ですが、このコマンドは IBM サポート・ヘルプのもとで使用するために提供されています。

  ```
  xsadmin –ch cathost –p 1099 –g Grid –teardown server1,server2,server3
  ```
- 経路テーブルの表示: このコマンドは、グリッドへの新しいクライアント接続をシミュレートすることによって、現在の経路テーブルを表示します。また、すべてのコンテナー・サーバーが経路テーブル内でのそれぞれのロール (どの区画のどのタイプの断片であるかなど) を認識していることを確認することによって、経路テーブルの妥当性検査も行います。

  ```
  xsadmin –ch cathost –p 1099 –g myGrid -routetable
  ```
- 未割り当て断片の表示: グリッドに配置できない断片がある場合は、これを使用してそれらの断片をリストすることができます。これは、配置サービスに配置を妨げる制約があるときのみ現れます。例えば、実動モードのとき、1 つの物理ボックスで JVM を始動した場合は、プライマリー断片のみを配置できます。 2 つ目のボックスで JVM が始動されるまで、複製は未割り当てとなります。配置サービスでは、プライマリー断片をホストしている JVM とは異なる IP アドレスを持つ JVM にのみ複製が配置されます。ゾーンに JVM が存在しない場合も、断片が未割り当てとなることがあります。

  ```
  xsadmin –ch cathost –p 1099 –g myGrid –unassigned
  ```
- トレースの設定: このコマンドは、xsadmin コマンドに対して指定されたフィルターと一致するすべての JVM について、トレースを設定します。この設定によって、別のコマンドが使用されるか、修正された JVM が障害を起こすか、または停止されるまでの間に、トレース設定のみが変更されます。

  ```
  xsadmin –ch cathost –p 1099 –g myGrid –fh host1 –settracespec
  ObjectGrid*=event=enabled
  ```

  これにより、指定されたホスト名 (この場合は host1) を持つボックスにあるすべての JVM に対して、トレースが使用可能となります。
- マップ・サイズの検査: マップ・サイズ・コマンドは、キー分布がキー内の断片に均等であることを確認するために役立ちます。他のコンテナーより著しく多いキーを持っているコンテナーがある場合は、キー・オブジェクトに対するハッシュ関数の分布が不完全である可能性があります。

  ```
  xsadmin -ch cathost -p 1099 -g myGrid -m myMapSet -mapsizes myMap
  ```

トランザクションの変更を配布する Java Message Service の使用

異なる層間、または混合プラットフォーム上の環境間で、変更を配布するために Java Message Service (JMS) を使用します。

JMS は、異なる層または混合しているプラットフォームの環境で配布された変更に理想的なプロトコルです。例えば、eXtreme Scale を使用するいくつかのアプリケーションが、IBM WebSphere Application Server Community Edition、Apache Geronimo、または Apache Tomcat にデプロイされていて、別のアプリケーションが WebSphere Application Server バージョン 6.x で実行しているとします。このような多様な環境における eXtreme Scale ピア間で配布される変更には、JMS が理想的です。HA マネージャーのメッセージ・トランスポートは非常に高速ですが、単一コア・グループに属する Java 仮想マシン にのみ変更を配布できます。 JMS はそれに比較すれば低速ですが、より広範囲で、多様なアプリケーション・クライアントのセットに ObjectGrid を共用させることができます。 JMS は、ファット Swing クライアントと、 WebSphere Extended Deployment にデプロイされているアプリケーションとの間で、ObjectGrid 内のデータを共用する場合に理想的です。

JMS を使用したトランザクションの変更の配布の例としては、組み込みの クライアント無効化メカニズムやピアツーピア複製メカニズムなどがあります。詳しくは、*管理ガイド*の JMS を使用したピアツーピア複製の構成に関する説明を参照してください。

JMS の実装

JMS は、ObjectGridEventListener として動作する Java オブジェクトを使用してトランザクションの変更を配布するために実装されます。このオブジェクトは、以下の 4 つの方法で状態を伝搬することができます。

1. 無効化: 除去、更新、または削除されるエントリーは、メッセージを受け取ると、すべてのピア Java 仮想マシンで除去されます。
2. 無効化の条件: ローカル・バージョンがパブリッシャーのバージョンと同じか、またはそれより古い場合のみ、エントリーが除去されます。
3. プッシュ: 除去、更新、削除または挿入されたエントリーは、JMS メッセージを受信する場合、すべてのピア Java 仮想マシンに追加または上書きされます。
4. プッシュ条件: ローカル・エントリーがパブリッシュされているバージョンより新しくない場合に、エントリーは受信サイドで更新または追加のみ行われます。

パブリッシュする変更の listen

プラグインは、ObjectGridEventListener インターフェースを実装し、transactionEnd イベントをインターセプトします。 eXtreme Scale がこのメソッドを呼び出す場合、プラグインはトランザクションによってタッチされる各マップの LogSequence リストを JMS メッセージに変換し、それをパブリッシュしようとします。プラグインは、すべてのマップまたはマップのサブセットの変更をパブリッシュするよう構成することができます。 LogSequence オブジェクトは、パブリッシュが使用可能なマップのために処理されます。 LogSequenceTransformer ObjectGrid クラスは、ストリームに対して各マップのフィルタリングされた LogSequence をシリアライズします。すべての LogSequences がストリームにシリアライズされたら、JMS ObjectMessage が作成され、既知のトピックにパブリッシュされます。

JMS メッセージの listen およびローカル ObjectGrid への適用

管理ガイド

同じプラグインはまた、既知のトピックにパブリッシュされるすべてのメッセージを受け取りながら、ループでスピンするスレッドを開始します。メッセージを受け取ると、LogSequenceTransformer クラスにメッセージ・コンテンツを渡します。このクラスでメッセージ・コンテンツは LogSequence オブジェクトのセットに変換されます。その後、ノー・ライトスルー・トランザクションが開始されます。各 LogSequence オブジェクトは Session.processLogSequence メソッドに提供され、その変更でローカル Map を更新します。 processLogSequence メソッドは、配布モードを理解しています。トランザクションはコミットされ、ローカル・キャッシュが変更を反映します。 JMS を使用してトランザクションの変更を配布する方法について詳しくは、*管理ガイド*の Java 仮想マシンのピア間での変更の配布に関する説明を参照してください。

第 6 章 セキュリティー

WebSphere eXtreme Scale は、分散キャッシング・システムです。アクセスを安全にしてデータを保護する場合、セキュリティーを有効にして、外部のセキュリティー・プロバイダーと統合することができます。

注: データベースなど、既存の非キャッシュ・データ・ストアでは、積極的に構成したり、有効にしたりする必要のない組み込みセキュリティー・フィーチャーがある可能性があります。ただし、eXtreme Scale でデータをキャッシュした後では、その結果として生じる、バックエンドのセキュリティー・フィーチャーが効力を持たなくなるような重要な状況を考慮する必要があります。eXtreme Scale セキュリティーを必要なレベルで構成すると、データの新しいキャッシュ・アーキテクチャーも保護できます。
以下に、eXtreme Scale セキュリティー機能について簡単に説明します。セキュリティーの構成について詳しくは、「*管理ガイド*」および「*プログラミング・ガイド*」を参照してください。

分散セキュリティーの基礎

分散 eXtreme Scale セキュリティーは、次の 3 つの主要概念に基づいています。

信頼できる認証
: 要求側 の ID を判別する能力。WebSphere eXtreme Scale は、クライアントとサーバー間の認証も、サーバー相互間の認証もともにサポートします。

許可
: 要求側にアクセス権を付与する許可を与える能力。WebSphere eXtreme Scale は、さまざまな操作に対しさまざまな許可をサポートします。

セキュア・トランスポート
: ネットワーク上での安全なデータ伝送。WebSphere eXtreme Scale は、Transport Layer Security/Secure Sockets Layer (TLS/SSL) プロトコルをサポートします。

認証

WebSphere eXtreme Scale は、分散クライアント・サーバー・フレームワークをサポートします。クライアント・サーバー・セキュリティー・インフラストラクチャーは、eXtreme Scale サーバーへのアクセスを安全にするために配置されています。例えば、認証が eXtreme Scale サーバーによって必要とされる場合、認証のためのクレデンシャルを eXtreme Scale クライアントがサーバーに提供する必要があります。これらのクレデンシャルは、ユーザー名とパスワードのペア、クライアント証明書、Kerberos チケット、またはクライアントとサーバーが合意した形式で示されたデータなどです。

許可

WebSphere eXtreme Scale の許可は、サブジェクトおよびアクセス権に基づいています。Java 認証・承認サービス (JAAS) を使用してアクセスを許可したり、Tivoli

Access Manager (TAM) などのカスタム・アプローチを接続して許可を処理したりできます。クライアントまたはグループに対しては、以下の許可を与えることができます。

マップ許可
マップに対して挿入、読み取り、更新、除去、または削除の操作を実行することを許可します。

ObjectGrid 許可
ObjectGrid オブジェクトに対してオブジェクト照会またはエンティティー照会およびストリーム照会を実行することを許可します。

DataGrid エージェント許可
DataGrid エージェントを ObjectGrid へデプロイすることを許可します。

サーバー・サイド・マップ許可
サーバー・マップをクライアント・サイドに複製すること、またはサーバー・マップに動的索引を作成することを許可します。

管理許可
管理タスクを実行することを許可します。

トランスポート・セキュリティー

クライアント・サーバー通信を保護するため、WebSphere eXtreme Scale は TLS/SSL をサポートします。これらのプロトコルは、eXtreme Scale クライアントとサーバー間のセキュア接続のための、認証性、保全性、および機密性を備えたトランスポート層セキュリティーを提供します。

グリッド・セキュリティー

セキュア環境では、サーバーは他のサーバーの認証性を確認できる必要があります。WebSphere eXtreme Scale は、この目的のために共有秘密ストリングのメカニズムを使用します。この秘密鍵のメカニズムは、共有パスワードと同様です。すべての eXtreme Scale サーバーは、共有秘密ストリングについて同意します。グリッドに加わるサーバーは、秘密ストリングを提示するよう求められます。参加しようとするサーバーの秘密ストリングがマスター・サーバーのものと一致すると、そのサーバーはグリッドに参加できます。一致しない場合、結合要求は拒否されます。

平文の機密事項の送信は保護されません。eXtreme Scale セキュリティー・インフラストラクチャーには、サーバーがこの機密事項を送信前に保護できるようにするため、SecureTokenManager プラグインが用意されています。セキュア操作の実装方法を選択できます。WebSphere eXtreme Scale は、セキュア操作が実装され、機密事項が暗号化され署名されるような実装を提供します。

動的デプロイメント・トポロジーでの Java Management Extensions (JMX) セキュリティー

JMX MBean セキュリティーは、すべてのバージョンの eXtreme Scale でサポートされています。カタログ・サーバー MBean およびコンテナー・サーバー MBean のクライアントを認証可能にして、MBean 操作へのアクセスを実施できるようになります。

ローカル eXtreme Scale セキュリティー

ローカル eXtreme Scale セキュリティーは、アプリケーションが ObjectGrid インスタンスを直接にインスタンス化して、使用するので、分散 eXtreme Scale モデルとは異なります。アプリケーションおよび eXtreme Scale インスタンスは、同じ Java 仮想マシン (JVM) 内にあります。このモデルにはクライアント/サーバーの概念が含まれていないので、認証はサポートされません。アプリケーションがそれ自身の認証を管理し、認証済みサブジェクト・オブジェクトを eXtreme Scale に渡す必要があります。ただし、ローカル eXtreme Scale プログラミング・モデルに使用される許可メカニズムは、クライアント/サーバー・モデルに使用されるものと同じです。

構成およびプログラミング

セキュリティーに関する構成とプログラミングについて詳しくは、「*管理ガイド*」および「*プログラミング・ガイド*」を参照してください。

第 7 章 トランザクション処理

WebSphere eXtreme Scale は、データとの相互作用のメカニズムとしてトランザクションを使用します。

セッションとトランザクション

データとの相互作用のために、アプリケーション内のスレッドは、独自の Session を必要とします。アプリケーションがスレッド上で ObjectGrid を使用する必要がある場合、ObjectGrid.getSession メソッドの 1 つを呼び出してスレッドを取得します。このセッションを使用すると、アプリケーションは ObjectGrid マップに保管されているデータの処理を行うことができます。

アプリケーションが Session オブジェクトを使用する場合、そのセッションはトランザクションのコンテキスト内にある必要があります。 Session オブジェクトに対する begin メソッド、commit メソッド、および rollback メソッドにより、トランザクションは、開始してコミット、あるいは開始してロールバックを行います。また、アプリケーションは自動コミット・モードで動作することも可能で、この場合、マップに対する操作が実行されるたびに、Session は自動的にトランザクションを開始してコミットします。自動コミット・モードでは複数の操作を単一トランザクションにグループ化することはできないため、複数操作のバッチを作成して単一トランザクションにする場合は、自動コミット・モードの方が時間がかかるオプションです。ただし、単一の操作しか含まないトランザクションの場合は、自動コミット・モードの方が速いオプションになります。

トランザクションの利点

トランザクションを使用して、以下の操作を行うことができます。

- 例外が発生した場合や、ビジネス・ロジックにより状態変更を元に戻す必要がある場合に、変更をロールバックします。
- データに対するロックの保持および解除を行い、コミット時に複数の変更をアトミック単位で適用します。
- 同時変更からスレッドを保護します。
- 変更に対するロックのライフサイクルを実装します。
- アトミック単位の複製を生成します。

トランザクション

トランザクションには、データ保管および操作に関して多くの利点があります。トランザクションを使用すれば、同時変更からグリッドを保護したり、複数の変更を 1 つの並行ユニットとして適用したり、データを複製したり、変更に対するロックのライフ・サイクルを実装したりすることができます。

トランザクションの概要

トランザクションを使用するのは、以下の理由からです。

- 同時変更からスレッドを保護する
- コミット時に複数の変更をアトミック単位で適用する
- 変更時にロックのライフ・サイクルを実装する
- 複製の単位の役割を果たす

トランザクションが開始すると、WebSphere eXtreme Scale は別の特別なマップを割り振って、そのトランザクションが使用するキーと値のペアの現在の変更またはコピーを保持します。通常、キーと値のペアにアクセスすると、アプリケーションがその値を受け取る前に、値のコピーが作成されます。その別のマップは、挿入、更新、取得、除去などの操作についてすべての変更を追跡します。キーは不変のものと見なされているため、コピーされません。ObjectTransformer オブジェクトを指定すると、このオブジェクトが値をコピーするために使用されます。 トランザクションがオプティミスティック・ロックを使用している場合は、トランザクションのコミット時に、以前の値のイメージも比較のために追跡されます。

トランザクションがロールバックされる場合、その別のマップの情報は破棄され、エントリーに対するロックは解除されます。トランザクションをコミットすると、変更がマップに適用され、ロックが解除されます。オプティミスティック・ロックが使用されている場合、eXtreme Scale は、以前のイメージ・バージョンの値とマップ内の値を比較します。トランザクションをコミットするには、これらの値が一致している必要があります。こうした比較によって複数バージョンのロック体系が可能になりますが、トランザクションがそのエントリーにアクセスすると、代わりに 2 つのコピーが作成されます。すべての値が再度コピーされ、新しいコピーがマップに保管されます。 WebSphere eXtreme Scale は、コミット後に値へのアプリケーション参照を変更するアプリケーションから自身を保護するために、このコピーを実行します。

情報の複数のコピーを使用しないようにできます。アプリケーションは、並行性を制限する代償としてオプティミスティック・ロックの代わりにペシミスティック・ロックを使用することで、コピーを節約できます。コミット後に値を変更しないことにアプリケーションが同意すれば、コミット時の値のコピーも回避することができます。

トランザクション・サイズ

トランザクションは、特に複製の場合には、大きいほど効果的です。ただし、大きなトランザクションの場合はエントリーのロックの保持時間が長くなるため、並行性に悪影響を及ぼします。大きなトランザクションを使用すると、複製のパフォーマンスが向上する場合があります。このパフォーマンスの向上は、マップを事前にロードする場合には重要です。さまざまなバッチ・サイズで実験を行い、使用するシナリオに最適なサイズを判別してください。

大きなトランザクションはローダーにとっても好都合です。 SQL バッチを実行できるローダーを使用している場合は、トランザクションによっては著しくパフォーマンスが向上する可能性があり、データベース側ではロードを著しく削減すること

ができます。このパフォーマンス向上は、ローダーの実装方法によって異なります。

CopyMode 属性

BackingMap または ObjectMap オブジェクトの CopyMode 属性を定義することで、コピーの数を調整することができます。コピー・モードには以下の値があります。

- COPY_ON_READ_AND_COMMIT
- COPY_ON_READ
- NO_COPY
- COPY_ON_WRITE
- COPY_TO_BYTES

COPY_ON_READ_AND_COMMIT がデフォルト値です。 COPY_ON_READ 値は、最初のデータ取得時にはコピーを行いますが、コミット時にはコピーを行いません。アプリケーションが、トランザクションのコミット後の値を変更しなければ、このモードが安全です。NO_COPY 値は、データをコピーしないため、読み取り専用データの場合のみ安全です。データが変更されない限り、分離目的でデータをコピーする必要はありません。

更新される可能性があるマップに NO_COPY 属性値を使用する場合は、注意が必要です。WebSphere eXtreme Scale は最初のタッチ時のコピーを使用して、トランザクションのロールバックを可能にします。アプリケーションはコピーを変更しただけなので、eXtreme Scale はそのコピーを破棄します。 NO_COPY 属性値が使用され、かつアプリケーションがコミットされた値を変更した場合は、ロールバックを完了することが不可能になります。 索引や複製はトランザクションのコミット時に更新されるため、コミット済みの値を変更すると、索引、複製などに問題が生じます。コミット済みのデータを変更してからトランザクションをロールバックした場合は、これによって実際にはまったくロールバックされないため、索引は更新されず、複製は行われません。他のスレッドは、コミットされていない変更を、ロックがあっても即時に参照することができます。読み取り専用マップ、または値を変更する前に適切なコピーを完了するアプリケーションの場合は、NO_COPY 属性値を使用してください。 NO_COPY 属性値を使用した場合に、データ保全性の問題で IBM サポートに連絡すると、コピー・モードを COPY_ON_READ_AND_COMMIT に設定して問題を再現するように求められます。

COPY_TO_BYTES 値は、マップ内の値をシリアライズ・フォームに保管します。eXtreme Scale は、読み取り時にシリアライズ・フォームからの値を拡張し、コミット時に値をシリアライズ・フォームに保管します。この方法によれば、読み取り時とコミット時の両方でコピーが行われます。

マップのデフォルトのコピー・モードは、BackingMap オブジェクトで構成することができます。さらに、トランザクションを開始する前に、ObjectMap.setCopyMode メソッドを使用してマップのコピー・モードを変更することができます。

`objectgrid.xml` ファイルにあり、指定のバックアップ・マップのコピー・モードを設定する方法を示すバックアップ・マップ・スニペットの例は以下のとおりです。この例では、`objectgrid/config` 名前空間として `cc` を使用しているものとします。

```
<cc:backingMap name="RuntimeLifespan" copyMode="NO_COPY"/>
```

詳しくは、「*プログラミング・ガイド*」に記載されている copyMode のベスト・プラクティスに関する情報を参照してください。

自動コミット・モード

アクティブに始動されたトランザクションがない場合は、アプリケーションが ObjectMap オブジェクトとの対話を行うと、アプリケーションの代わりに自動的に開始およびコミット操作が行われます。この自動的な開始およびコミット操作は役に立ちますが、ロールバックおよびロックが有効に機能する妨げとなります。トランザクションのサイズが小さすぎると、同期複製スピードに影響します。エンティティー・マネージャー・アプリケーションを使用している場合は、自動コミット・モードは使用しないでください。その理由は、EntityManager.find メソッドで検索されたオブジェクトが、そのメソッドが戻されると同時に管理不能となり、使用不可となるためです。

ロック・モードおよびトランザクション

ロックはトランザクションに束縛されます。以下のロック設定を指定することができます。

- **ロックなし**: ロック設定を使用しないと、実行は最速になります。読み取り専用データを使用していれば、ロックは必要ない場合があります。
- **ペシミスティック・ロック**: エントリーに対するロックを獲得し、コミット時までそのロックを保持します。このロック戦略は、スループットを低下させる代わりに、優れた一貫性を提供します。
- **オプティミスティック・ロック**: トランザクションがタッチするすべてのレコードの以前のイメージを取得して、トランザクションのコミット時に、そのイメージと現在のエントリーの値を比較します。エントリーの値が変更された場合、そのトランザクションはロールバックします。コミット時までロックは保持されません。このロック戦略は、ペシミスティック戦略よりも並行性において優れていますが、トランザクション・ロールバックのリスクがあり、エントリーのコピーを作成するためにメモリーを消費します。

BackingMap でロック戦略を設定します。各トランザクションのロック戦略を変更することはできません。XML ファイルを使用してマップに対してロック・モードを設定する方法を示す XML スニペットの例は以下のとおりです。この場合、cc は、objectgrid/config 名前空間用の名前空間であるとします。

```
<cc:backingMap name="RuntimeLifespan" lockStrategy="PESSIMISTIC" />
```

外部トランザクション・コーディネーター

通常、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。ただし、eXtreme Scale が組み込まれていると、トランザクションは、外部トランザクション・コーディネーターによって開始および終了する場合があります。外部トランザクション・コーディネーターを使用している場合は、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。 eXtreme Scale および 外部トランザクションの対話について詳しくは、*プログラミング・ガイド*を参照してください。WebSphere Application Server を使用している場合は、WebSphereTranscationCallback プラグインを使用できます。

WebSphere eXtreme Scale で使用可能なプラグインについて詳しくは、*プログラミング・ガイド*を参照してください。

ロック・ストラテジー

ロック・ストラテジーには、ペシミスティック、オプティミスティック、およびロックなしがあります。ロック・ストラテジーを選択する場合、各タイプの操作の比率、ローダーを使用するかどうかなどを知っておく必要があります。

ペシミスティック・ロック

ほかのロック・ストラテジーが可能でない場合は、マップの読み書きにペシミスティック・ロック・ストラテジーを使用します。 ObjectGrid マップがペシミスティック・ロック・ストラテジーを使用するように構成されている場合、トランザクションが最初に BackingMap からのエントリーを取得すると、マップ・エントリーのペシミスティック・トランザクション・ロックが取得されます。ペシミスティック・ロックは、アプリケーションがトランザクションを完了するまでは保留されます。通常の場合、ペシミスティック・ロック・ストラテジーは、以下の状態で使用されます。

- BackingMap がローダー付き、またはローダーなしで構成され、バージョン管理情報が使用可能でない場合。
- BackingMap が、並行処理制御について eXtreme Scale からの支援を必要とするアプリケーションによって直接使用されている場合。
- バージョン管理情報は使用できるが、更新トランザクションがバッキング・エントリー上で頻繁に衝突し、その結果、オプティミスティック更新が失敗する場合。

ペシミスティック・ロック・ストラテジーは、パフォーマンスとスケーラビリティーに最大のインパクトを与えるので、このストラテジーはほかのロック・ストラテジーが実行可能でないときのマップの読み取りと書き込みにのみ使用してください。例えば、こうした状態には、オプティミスティック更新の失敗が頻繁に発生する場合や、オプティミスティック障害からのリカバリーをアプリケーションが処理するには難しい場合が含まれます。

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・ストラテジーでは、並行して実行中に、2 つのトランザクションが同じマップ・エントリーを更新することはないと想定します。このことから、トランザクションのライフサイクル中、ロック・モードを保留する必要はありません。これは、複数のトランザクションがマップ・エントリーを並行して更新するとは考えられないためです。オプティミスティック・ロック・ストラテジーは通常、以下の場合に使用されます。

- BackingMap がローダー付き、またはローダーなしで構成され、バージョン管理情報が使用可能である場合。
- BackingMap のほとんどのトランザクションが読み取り操作を実行するトランザクションである場合。 BackingMap に対するエントリーの挿入、更新、または除去操作は、あまり行われません。

- BackingMap は、読み取りと比べてより頻繁に挿入、更新、または除去されるが、トランザクションは同じマップ・エントリー上でほとんど衝突しない場合。

ペシミスティック・ロック・ストラテジーと同様に、 ObjectMap インターフェース上のメソッドは、eXtreme Scale が、アクセス中のマップ・エントリーのロック・モードを自動的に獲得する方法を決定します。ただし、ペシミスティック・ストラテジーとオプティミスティック・ストラテジーの間には、以下のような違いがあります。

- ペシミスティック・ロック・ストラテジーと同様に、メソッドの呼び出しの際、get メソッドおよび getAll メソッドによって S ロック・モードが獲得されます。しかし、オプティミスティック・ロックを使用すると、 S ロック・モードはトランザクションが完了するまで保留されません。代わりに、S ロック・モードはメソッドがアプリケーションに戻す前に保留解除されます。ロック・モードの獲得の目的は、eXtreme Scale が、その他のトランザクションからのコミット済みデータのみが現行トランザクションに可視となるように保証できるようにすることです。 eXtreme Scale がそのデータがコミット済みであることを確認した後で、S ロック・モードは保留解除されます。コミット時に、オプティミスティック・バージョン管理チェックが実行され、現行トランザクションがその S ロック・モードを保留解除した後で、マップ・エントリーを変更したトランザクションが他にないことが確認されます。更新、無効化、または削除される前にマップからエントリーがフェッチされない場合、eXtreme Scale ランタイムによって、暗黙的にマップからエントリーがフェッチされます。この暗黙的な get 操作は、エントリーの変更が要求された時点における現行値を取得するために実行されます。
- ペシミスティック・ロック・ストラテジーとは異なり、 getForUpdate メソッドと getAllForUpdate メソッドは、オプティミスティック・ロック・ストラテジーが使用された場合には、 get メソッドと getAll メソッドと同様に処理されます。つまり、S ロック・モードはメソッドの開始時に獲得され、S ロック・モードはアプリケーションに戻る前に保留解除されます。

その他の ObjectMap メソッドは、すべてペシミスティック・ロック・ストラテジーの場合と同様に処理されます。つまり、commit メソッドが呼び出されると、挿入、更新、除去、タッチ、または無効化されたマップ・エントリー用に X ロック・モードが獲得され、トランザクションがコミット処理を完了するまで X ロック・モードが保留されます。

オプティミスティック・ロック・ストラテジーでは、並行して実行中のトランザクションが同じマップ・エントリーを更新することはないと想定します。この想定から、トランザクションの存続期間中、ロック・モードを保留する必要はありません。これは、複数のトランザクションがマップ・エントリーを並行して更新するとは考えられないためです。しかし、ロック・モードが保留されなかったため、現行トランザクションがその S ロック・モードを保留解除した後で、別の並行トランザクションがマップ・エントリーを更新する可能性があります。

この可能性に対処するため、eXtreme Scale はコミット時に X ロックを取得し、オプティミスティック・バージョン管理チェックを行って、現行トランザクションが BackingMap からマップ・エントリーを読み取って以降、他にマップ・エントリーを変更したトランザクションがないことを確認します。別のトランザクションがマップ・エントリーを変更した場合、バージョン・チェックは失敗し、

OptimisticCollisionException 例外が発生します。この例外により、現行トランザクションが強制的にロールバックされ、トランザクション全体がアプリケーションによって再試行されることになります。オプティミスティック・ロック・ストラテジーは、マップがほとんど既読で、同じマップ・エントリーに対する更新が起こる可能性が低い場合に非常に便利です。

ロックなし

BackingMap がロックなしストラテジーを使用するよう構成されている場合、マップ・エントリーのトランザクション・ロックは獲得されません。

ロックなしストラテジーは、アプリケーションが Enterprise JavaBeans™ (EJB) コンテナーなどのパーシスタンス・マネージャーである場合や、アプリケーションが Hibernate を使用して永続データを取得している場合に有効です。このシナリオでは、BackingMap はローダーを使用せずに構成され、パーシスタンス・マネージャーによってデータ・キャッシュとして使用されます。またこのシナリオでは、パーシスタンス・マネージャーにより、同じマップ・エントリーにアクセスするトランザクション間の並行性制御が提供されます。

WebSphere eXtreme Scale は、並行性制御のためにトランザクション・ロックを入手する必要はありません。これは、パーシスタンス・マネージャーが、コミットされた変更で ObjectGrid マップを更新する前にそのトランザクション・ロックをリリースしないことを前提としています。パーシスタンス・マネージャーがロックを解放する場合は、ペシミスティックまたはオプティミスティック・ロック・ストラテジーを使用しなければなりません。例えば、EJB コンテナーのパーシスタンス・マネージャーが、EJB コンテナー管理のトランザクション内でコミットされたデータで ObjectGrid Map を更新していると仮定します。ObjectGrid マップの更新が、パーシスタンス・マネージャーのトランザクション・ロックが解放される前に発生する場合、ロックなしストラテジーを使用することができます。パーシスタンス・マネージャーのトランザクション・ロックが解放された後で ObjectGrid マップ更新が発生する場合は、オプティミスティックまたはペシミスティックのいずれかのロック・ストラテジーを使用してください。

ロックなしストラテジーの使用が可能なもう 1 つのシナリオは、アプリケーションが BackingMap を直接使用し、ローダーがマップに対して構成されているときです。このシナリオでは、ローダーは、Java Database Connectivity (JDBC) または Hibernate のいずれかを使用してリレーショナル・データベース内のデータにアクセスすることによって、リレーショナル・データベース管理システム (RDBMS) によって提供される並行性制御サポートを使用します。ローダーの実装は、オプティミスティックまたはペシミスティックのいずれかの方法を使用できます。オプティミスティック・ロックまたはバージョン管理方法を使用するローダーは、大量の並行性およびパフォーマンスの達成を支援します。オプティミスティック・ロック手法の実装について詳しくは、「*管理ガイド*」内のローダー考慮事項に関する説明の OptimisticCallback セクションを参照してください。基礎となるバックエンドのペシミスティック・ロック・サポートを使用するローダーを使用する場合は、ローダー・インターフェースの get メソッドに渡される forUpdate パラメーターを使用することがあります。アプリケーションがデータを取得するために ObjectMap インターフェースの getForUpdate メソッドを使用した場合は、このパラメーターを true に設定します。ローダーはこのパラメーターを使用して、読み取り中の行のアップ

グレード可能なロックを要求するかどうかを判別できます。例えば、DB2® は、SQL の SELECT ステートメントに FOR UPDATE 節が含まれている場合、アップグレード可能なロックを獲得します。このアプローチは、 143 ページの『ペシミスティック・ロック』で説明されているのと同じデッドロック防止を提供します。

第 8 章 チュートリアル

チュートリアルは、特定の WebSphere eXtreme Scale 機能を使用し始める際に有効です。

エンティティー・マネージャーのチュートリアル: 概要

エンティティー・マネージャーのチュートリアルでは、WebSphere eXtreme Scale を使用して Web サイトのオーダー情報を格納する方法を示します。メモリー内のローカル eXtreme Scale を使用する、簡単な Java Platform, Standard Edition 5 アプリケーションを作成できます。エンティティーは Java SE 5 のアノテーションおよび汎用を使用します。

始める前に

チュートリアルを始める前に、以下の要件を満たしていることを確認してください。

- Java SE 5 が必要です。
- クラスパスに `objectgrid.jar` ファイルがなければなりません。

エンティティー・マネージャーのチュートリアル: エンティティー・クラスの作成

エンティティー・マネージャー・チュートリアルの最初のステップでは、エンティティー・クラスの作成、エンティティー・タイプの eXtreme Scale への登録、およびエンティティー・インスタンスのキャッシュへの格納によって、1 つのエンティティーを持つローカル ObjectGrid を作成する方法を示します。

このタスクについて

1. Order オブジェクトを作成します。 このオブジェクトを ObjectGrid エンティティーとして識別するには、@Entity アノテーションを追加します。このアノテーションを追加すると、オブジェクト内のシリアライズ可能な属性はすべて、属性のアノテーションを使用して属性をオーバーライドする場合を除いて、自動的に eXtreme Scale 内で保持されます。 orderNumber 属性には、この属性が 1 次キーであることを示す @Id というアノテーションが付けられています。 Order オブジェクトの例を次に示します。

   Order.java

   ```
   @Entity
   public class Order
   {
       @Id String orderNumber;
       Date date;
       String customerName;
       String itemName;
       int quantity;
       double price;
   }
   ```

2. eXtreme Scale Hello World アプリケーションを実行してエンティティー操作をデモンストレーションします。 次のプログラム例をスタンドアロン・モードで実行することで、エンティティー操作をデモンストレーションすることができます。このプログラムは、クラスパスに objectgrid.jar ファイルが追加されている Eclipse Java プロジェクトで使用します。 eXtreme Scale を使用する簡単な Hello world アプリケーションの例を次に示します。

Application.java

```
package emtutorial.basic.step1;

import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class Application
{

    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og =
    ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.registerEntities(new Class[] {Order.class});

        Session s = og.getSession();
        EntityManager em = s.getEntityManager();

        em.getTransaction().begin();

        Order o = new Order();
        o.customerName = "John Smith";
        o.date = new java.util.Date(System.currentTimeMillis());
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;

        em.persist(o);
        em.getTransaction().commit();

        em.getTransaction().begin();
        o = (Order)em.find(Order.class, "1");
        System.out.println("Found order for customer: " + o.customerName);
        em.getTransaction().commit();
    }
}
```

このアプリケーション例は以下の操作を実行します。

a. 自動的に生成された名前を持つローカル eXtreme Scale を初期化します。

b. API は必ずしも必要ではありませんが registerEntities API を使用して、エンティティー・クラスをアプリケーションに登録します。

c. セッションとそのセッションのエンティティー・マネージャーへの参照を取得します。

d. 各 eXtreme Scale Session を単一の EntityManager および EntityTransaction に関連付けます。これで EntityManager が使用されます。

e. registerEntities メソッドが Order という BackingMap オブジェクトを作成し、Order オブジェクトのメタデータをその BackingMap オブジェクトに関連付けます。このメタデータには、属性タイプと名前とともに、キー属性と非キー属性が含まれています。

f. トランザクションが開始し、Order インスタンスが作成されます。トランザクションには、いくつかの値が格納され、EntityManager.persist メソッドの使用によって永続化されます。このメソッドでは、関連付けられている ObjectGrid Map に組み込まれるまでエンティティーが待機していると認識されます。

g. 次に、トランザクションがコミットされ、エンティティーが ObjectMap に組み込まれます。

h. 別のトランザクションが作成され、キー 1 を使用して Order オブジェクトが取得されます。EntityManager.find メソッドでは型キャストが必要です。Java SE 1.4 以降の Java 仮想マシンで objectgrid.jar ファイルが確実に実行されるようにするために、Java SE 5 の汎用機能が使用されないからです。

エンティティー・マネージャーのチュートリアル: エンティティー・リレーションシップの形成

リレーションシップを持つ 2 つのエンティティー・クラスを作成し、それらのエンティティーを ObjectGrid に登録し、エンティティー・インスタンスをキャッシュに格納することで、エンティティー間の簡単なリレーションシップを作成します。

1. Customer エンティティーを作成します。このエンティティーは、カスタマーの情報を Order オブジェクトとは別に格納するために使用されます。 Customer エンティティーの例を次に示します。

   Customer.java

   ```
   @Entity
   public class Customer
   {
       @Id String id;
       String firstName;
       String surname;
       String address;
       String phoneNumber;
   }
   ```

   このクラスには、名前、住所、電話番号といった、カスタマーに関する情報が含まれます。

2. Order オブジェクトを作成します。このオブジェクトは 147 ページの『エンティティー・マネージャーのチュートリアル: エンティティー・クラスの作成』トピックの Order オブジェクトと類似しています。 Order オブジェクトの例を次に示します。

   Order.java

   ```
   @Entity
   public class Order
   {
       @Id String orderNumber;
       Date date;
       @ManyToOne(cascade=CascadeType.PERSIST) Customer customer;
       String itemName;
       int quantity;
       double price;
   }
   ```

   この例では、Customer オブジェクトへの参照が customerName 属性に取って代わります。この参照には多対 1 リレーションシップを示すアノテーションが付いています。多対 1 リレーションシップは各オーダーに 1 人のカスタマーがあることを示しますが、複数のオーダーが同じカスタマーを参照することもあります。カスケード・アノテーション修飾子は、EntityManager で Order オブジェクトを永続化させる場合に、Customer オブジェクトも永続化させる必要があることを示しています。カスケード永続化オプション (デフォルトのオプション) を設定しない場合は、Order オブジェクトとともに Customer オブジェクトを手動で永続化する必要があります。

3. エンティティーを使用して、ObjectGrid インスタンスのマップを定義します。各マップは特定のエンティティーに対して定義されています。1 つのエンティティーの名前は Order で、もう 1 つのエンティティーの名前は Customer です。 次のアプリケーション例は、カスタマー・オーダーの格納および取得方法を示しています。

Application.java

```
public class Application
{
    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og =
     ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.registerEntities(new Class[] {Order.class});

        Session s = og.getSession();
        EntityManager em = s.getEntityManager();

        em.getTransaction().begin();

        Customer cust = new Customer();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";

        Order o = new Order();
        o.customer = cust;
        o.date = new java.util.Date();
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;

        em.persist(o);
        em.getTransaction().commit();

        em.getTransaction().begin();
        o = (Order)em.find(Order.class, "1");
        System.out.println("Found order for customer: "
     + o.customer.firstName + " " + o.customer.surname);
        em.getTransaction().commit();
    }
}
```

このアプリケーションは、直前のステップにあるアプリケーション例と類似しています。前の例では、単一のクラス Order のみが登録されました。 WebSphere eXtreme Scale では、Customer エンティティーへの参照を検出して自動的に組み込むため、John Smith の Customer インスタンスが作成されると、新しい Order オブジェクトから参照されます。この結果として、新しいカスタマーは自動的に永続化されます。これは、2 つのオーダーの関係には、各オブジェクトの永続化を必要とするカスケード修飾子が組み込まれているためです。Order オブジェクトが見つかると、エンティティー・マネージャーでは、関連の Customer オブジェクトを自動的に検出し、このオブジェクトへの参照を挿入します。

エンティティー・マネージャーのチュートリアル: Order エンティティー・スキーマ

単一方向と双方向の両方の関係、順序リスト、および外部キー関係を使用して、4 つのエンティティー・クラスを作成します。エンティティーの永続化と検索には、EntityManager API を使用します。このチュートリアルの前の部分にある Order および Customer エンティティーを前提として、このチュートリアル・ステップでは、Item および OrderLine という 2 つのエンティティーをさらに追加します。

このタスクについて

図 40. Order エンティティー・スキーマ： Order エンティティーは、1 人のカスタマーへの参照と 0 個以上の OrderLine を持っています。各 OrderLine エンティティーは、単一の Item を参照し、オーダーされた数量を含みます。

1. Customer エンティティーを作成します。このエンティティーは、これまでの例と類似しています。

   Customer.java
   ```
   @Entity
   public class Customer
   {
       @Id String id;
       String firstName;
       String surname;
       String address;
       String phoneNumber;
   }
   ```

2. Item エンティティーを作成します。このエンティティーには、ストアのインベントリーにある製品の情報 (製品説明、数量、価格など) が保持されています。

   Item.java
   ```
   @Entity
   public class Item
   {
       @Id String id;
       String description;
       long quantityOnHand;
       double price;
   }
   ```

3. OrderLine エンティティーを作成します。各 Order は、オーダー内の各品目の数量を示す 0 個以上の OrderLine を持っています。 OrderLine のキーは、OrderLine を所有する Order とオーダー行に数値を割り当てる整数から構成される複合キーです。エンティティーのすべての関係にカスケード永続化修飾子を追加します。

   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;
   }
   ```

4. 最終の Order オブジェクトを作成します。このオブジェクトは、オーダーに対応した Customer と OrderLine オブジェクトの集合を参照します。

   Order.java
   ```
   @Entity
   public class Order
   {
       @Id String orderNumber;
       java.util.Date date;
       @ManyToOne(cascade=CascadeType.PERSIST) Customer customer;
       @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
      @OrderBy("lineNumber") List<OrderLine> lines;
   }
   ```

cascade ALL は、行に対する修飾子として使用されます。この修飾子は、PERSIST 操作と REMOVE 操作をカスケードするように EntityManager に指示します。例えば、Order エンティティーを永続化または削除すると、すべての OrderLine エンティティーも永続化または削除されます。

Order オブジェクトの行リストから OrderLine エンティティーを削除すると、参照は破損されます。ただし、OrderLine エンティティーはキャッシュからは削除されません。キャッシュからエンティティーを削除するには、EntityManager remove API を使用する必要があります。REMOVE 操作は、OrderLine から Customer エンティティーまたは Item エンティティーで使用されることはありません。したがって、OrderLine を削除するときに Order または Item を削除しても、Customer エンティティーは残ります。

mappedBy 修飾子は、ターゲット・エンティティーとの逆の関係を示しています。この修飾子は、ソース・エンティティーを参照するターゲット・エンティティーの属性、および 1 対 1 関係または多対多関係の所有側を指定します。通常、この修飾子は省略できます。ただし、WebSphere eXtreme Scale で自動的に検出できなかった場合、この修飾子を指定する必要があることを示すエラーが表示されます。 OrderLine エンティティーが、多対 1 関係にある型 Order 属性を 2 つ含む場合、通常はエラーが発生します。

@OrderBy アノテーションは、各 OrderLine エンティティーが行リストに表示される順序を指定します。このアノテーションを指定しない場合は、行は任意の順序で表示されます。ArrayList を指定すると、行が Order エンティティーに追加されて、順序が維持されますが、EntityManager では必ずしもこの順序が認識されるわけではありません。find メソッドを実行して、キャッシュから Order オブジェクトを取得する場合、ArrayList オブジェクトはリスト・オブジェクトにはなりません。

5. アプリケーションを作成します。以下の例は、最終の Order オブジェクトを示し、オーダーに対応した Customer と OrderLine オブジェクトの集まりを参照します。
   a. オーダー対象であり、管理エンティティーとなる Item を検索します。
   b. OrderLine を作成し、各 Item に付加します。
   c. Order を作成し、各 OrderLine とそのカスタマーに関連付けます。
   d. オーダーを永続化します。この場合、各 OrderLine も自動的に永続化されます。
   e. トランザクションをコミットします。各エンティティーが切り離され、エンティティーの状態がキャッシュと同期化されます。
   f. オーダー情報を出力します。OrderLine エンティティーは、OrderLine ID 別に自動的に分類されます。

```
Application.java

static public void main(String [] args)
        throws Exception
    {
        ...

        // Add some items to our inventory.
        em.getTransaction().begin();
        createItems(em);
```

```
        em.getTransaction().commit();

        // Create a new customer with the items in his cart.
        em.getTransaction().begin();
        Customer cust = createCustomer();
        em.persist(cust);

        // Create a new order and add an order line for each item.
        // Each line item is automatically persisted since the
    // Cascade=ALL option is set.
        Order order = createOrderFromItems(em, cust, "ORDER_1",
     new String[]{"1", "2"}, new int[]{1,3});
        em.persist(order);
        em.getTransaction().commit();

        // Print the order summary
        em.getTransaction().begin();
        order = (Order)em.find(Order.class, "ORDER_1");
        System.out.println(printOrderSummary(order));
        em.getTransaction().commit();
    }

    public static Customer createCustomer() {
        Customer cust = new Customer();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";
        return cust;
    }

    public static void createItems(EntityManager em) {
        Item item1 = new Item();
        item1.id = "1";
        item1.price = 9.99;
        item1.description = "Widget 1";
        item1.quantityOnHand = 4000;
        em.persist(item1);

        Item item2 = new Item();
        item2.id = "2";
        item2.price = 15.99;
        item2.description = "Widget 2";
        item2.quantityOnHand = 225;
        em.persist(item2);

    }

    public static Order createOrderFromItems(EntityManager em,
   Customer cust, String orderId, String[] itemIds, int[] qty) {

        Item[] items = getItems(em, itemIds);

        Order order = new Order();
        order.customer = cust;
        order.date = new java.util.Date();
        order.orderNumber = orderId;
        order.lines = new ArrayList<OrderLine>(items.length);
    for(int i=0;i<items.length;i++){
      OrderLine line = new OrderLine();
            line.lineNumber = i+1;
            line.item = items[i];
            line.price = line.item.price;
            line.quantity = qty[i];
            line.order = order;
            order.lines.add(line);
```

```
    }
    return order;
}

public static Item[] getItems(EntityManager em, String[] itemIds) {
    Item[] items = new Item[itemIds.length];
    for(int i=0;i<items.length;i++){
items[i] = (Item) em.find(Item.class, itemIds[i]);
    }
    return items;
}
```

次のステップでは、エンティティーを削除します。EntityManager インターフェースは、削除対象にするオブジェクトにマークを付ける remove メソッドを備えています。アプリケーションでは、remove メソッドを呼び出す前に、すべての関係のコレクションからエンティティーを削除する必要があります。最終ステップとして、参照を編集し、remove メソッド em.remove(object) を実行します。

エンティティー・マネージャーのチュートリアル: エントリーの更新

エンティティーを変更する場合は、インスタンスを検出し、インスタンスと参照先エンティティーを更新し、トランザクションをコミットできます。

エントリーを更新します。 以下の例は、Order インスタンスの検索方法、このインスタンスと参照先エンティティーの変更方法、およびトランザクションのコミット方法を示しています。

```
public static void updateCustomerOrder(EntityManager em) {
    em.getTransaction().begin();
    Order order = (Order) em.find(Order.class, "ORDER_1");
    processDiscount(order, 10);
    Customer cust = order.customer;
    cust.phoneNumber = "5075551234";
    em.getTransaction().commit();
}

public static void processDiscount(Order order, double discountPct) {
    for(OrderLine line : order.lines) {
        line.price = line.price * ((100-discountPct)/100);
    }
}
```

トランザクションをフラッシュすると、すべての管理エンティティーがキャッシュと同期化されます。トランザクションがコミットされると、フラッシュが自動的に実行されます。この場合は、Order が管理エンティティーとなります。
Order、Customer、および OrderLine から参照されるエンティティーも管理エンティティーとなります。トランザクションがフラッシュされるとき、各エンティティーは検査され、変更されているかどうかが判定されます。変更されているエンティティーは、キャッシュ内で更新されます。コミットまたはロールバックされてトランザクションが完了した後、エンティティーは切り離され、エンティティーで行われた変更はキャッシュに反映されません。

エンティティー・マネージャーのチュートリアル: 索引によるエントリーの更新と除去

索引を使用して、エンティティーを検索、更新、および除去することができます。

更新を使用してエンティティーを更新および除去します。 索引を使用して、エンティティーを検索、更新、および除去することができます。以下の例では、Order エ

ンティティー・クラスを更新して、@Index アノテーションを使用します。@Index アノテーションは、属性の範囲索引で作成するよう WebSphere eXtreme Scale に通知します。索引の名前は属性の名前と同じで、常に MapRangeIndex 索引型です。

Order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    @Index java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
    @OrderBy("lineNumber") List<OrderLine> lines;  }
```

以下の例では、直前にサブミットされたすべてのオーダーを取り消す方法を示しています。索引を使用してオーダーを検索し、オーダーの品目を在庫に戻し、オーダーおよびそれに関連する明細行をシステムから削除します。

```
public static void cancelOrdersUsingIndex(Session s)
  throws ObjectGridException {
    // Cancel all orders that were submitted 1 minute ago
    java.util.Date cancelTime = new
   java.util.Date(System.currentTimeMillis() - 60000);
    EntityManager em = s.getEntityManager();
    em.getTransaction().begin();
    MapRangeIndex dateIndex = (MapRangeIndex)
   s.getMap("Order").getIndex("date");
    Iterator<Tuple> orderKeys = dateIndex.findGreaterEqual(cancelTime);
  while(orderKeys.hasNext()) {
   Tuple orderKey = orderKeys.next();
   // Find the Order so we can remove it.
   Order curOrder = (Order) em.find(Order.class, orderKey);
   // Verify that the order was not updated by someone else.
   if(curOrder != null && curOrder.date.getTime() >= cancelTime.getTime()) {
    for(OrderLine line : curOrder.lines) {
     // Add the item back to the inventory.
     line.item.quantityOnHand += line.quantity;
     line.quantity = 0;
    }
   em.remove(curOrder);
   }
  }
 em.getTransaction().commit();
}
```

エンティティー・マネージャーのチュートリアル: 照会を使用したエントリーの更新と除去

照会を使用してエンティティーを更新および除去することができます。

照会を使用してエンティティーを更新および除去します。

Order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    @Index java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
   @OrderBy("lineNumber") List<OrderLine> lines;
}
```

Order エンティティー・クラスは前の例のものと同じです。照会ストリングが日付を使用してエンティティーを検索するため、このクラスは引き続き @Index アノテーションを提供します。照会エンジンは、索引が使用可能であるときは、索引を使用します。

```
public static void cancelOrdersUsingQuery(Session s) {
        // Cancel all orders that were submitted 1 minute ago
        java.util.Date cancelTime =
     new java.util.Date(System.currentTimeMillis() - 60000);
        EntityManager em = s.getEntityManager();
        em.getTransaction().begin();

        // Create a query that will find the order based on date.  Since
        // we have an index defined on the order date, the query
    // will automatically use it.
        Query query = em.createQuery("SELECT order FROM Order order
     WHERE order.date >= ?1");
        query.setParameter(1, cancelTime);
        Iterator<Order> orderIterator = query.getResultIterator();
    while(orderIterator.hasNext()) {
     Order order = orderIterator.next();
     // Verify that the order wasn't updated by someone else.
     // Since the query used an index, there was no lock on the row.
     if(order != null && order.date.getTime() >= cancelTime.getTime()) {
       for(OrderLine line : order.lines) {
        // Add the item back to the inventory.
        line.item.quantityOnHand += line.quantity;
        line.quantity = 0;
       }
       em.remove(order);
     }
   }
  em.getTransaction().commit();
}
```

前の例と同様、cancelOrdersUsingQuery メソッドの目的は、この 1 分間にサブミットされたすべてのオーダーを取り消すことです。オーダーを取り消すには、照会を使用してオーダーを検索し、オーダー内の品目を在庫に戻し、オーダーおよび関連の明細行をシステムから削除します。

ObjectQuery の解説

以下のステップにより、ある Web サイトのオーダー情報を保管できるローカルのメモリー内 ObjectGrid を開発できます。また、ObjectQuery を使用してグリッド内のデータを照会する方法を示します。

始める前に

クラスパスに必ず `objectgrid.jar` ファイルを入れてください。

このタスクについて

このチュートリアルの各ステップは、前のステップを基にしています。各ステップに従って、メモリー内のローカル ObjectGrid を使用する Java Platform, Standard Edition バージョン 1.4 (以降) のシンプルなアプリケーションをビルドします。

1. 157 ページの『ObjectQuery チュートリアル - ステップ 1』
   - ローカル ObjectGrid の作成方法
   - フィールド・アクセスを使用した単一オブジェクト用スキーマの定義方法
   - オブジェクトの保管方法
   - ObjectQuery を使用したオブジェクトの照会方法
2. 158 ページの『ObjectQuery チュートリアル - ステップ 2』
   - 照会で使用できる索引の作成方法
3. 159 ページの『ObjectQuery チュートリアル - ステップ 3』
   - 2 つの関連エンティティーを持つスキーマの作成方法
   - オブジェクト間に外部キー参照を持つオブジェクトを保管する方法

- JOIN で単一照会を使用したオブジェクトの照会方法

4. 161 ページの『ObjectQuery チュートリアル - ステップ 4』
   - 複数の関連エンティティーを持つスキーマの作成方法
   - フィールド・アクセスの代わりにメソッドまたはプロパティー・アクセスを使用する方法。

ObjectQuery チュートリアル - ステップ 1

以下のステップにより、ObjectMap API を使用して、オンライン・ショップのオーダー情報を保管するローカルのメモリー内 ObjectGrid を引き続き開発できます。マップのスキーマを定義し、そのマップに対して照会を実行します。

1. マップ・スキーマを持つ ObjectGrid を作成します。

   マップに対応した 1 つのマップ・スキーマを持つ ObjectGrid を作成して、オブジェクトをキャッシュに挿入し、後でシンプルな照会を使用してこのオブジェクトを検索します。

   OrderBean.java

```
public class OrderBean implements Serializable {
    String orderNumber;
    java.util.Date date;
    String customerName;
    String itemName;
    int quantity;
    double price;
}
```

2. 1 次キーを定義します。

   このコードは、OrderBean オブジェクトを示しています。キャッシュ内のすべてのオブジェクトは、(デフォルトで) シリアライズ可能でなければならないため、このオブジェクトは、java.io.Serializable インターフェースを実装します。

   orderNumber 属性は、オブジェクトの主キーです。次のプログラム例は、スタンドアロン・モードで実行できます。このチュートリアルは、objectgrid.jar ファイルがクラスパスに追加されている Eclipse Java プロジェクトで実行してください。

   Application.java

```
package querytutorial.basic.step1;

import java.util.Iterator;

import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.config.QueryConfig;
import com.ibm.websphere.objectgrid.config.QueryMapping;
import com.ibm.websphere.objectgrid.query.ObjectQuery;

public class Application
{

    static public void main(String [] args) throws Exception
    {
        ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.defineMap("Order");

        // Define the schema
        QueryConfig queryCfg = new QueryConfig();
        queryCfg.addQueryMapping(new QueryMapping("Order", OrderBean.class.getName(),
     "orderNumber", QueryMapping.FIELD_ACCESS));
```

```
        og.setQueryConfig(queryCfg);

        Session s = og.getSession();
        ObjectMap orderMap = s.getMap("Order");

        s.begin();
        OrderBean o = new OrderBean();
        o.customerName = "John Smith";
        o.date = new java.util.Date(System.currentTimeMillis());
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;
        orderMap.put(o.orderNumber, o);
        s.commit();

        s.begin();
        ObjectQuery query = s.createObjectQuery("SELECT o FROM Order o WHERE o.itemName='Widget'");
        Iterator result = query.getResultIterator();
        o = (OrderBean) result.next();
        System.out.println("Found order for customer: " + o.customerName);
        s.commit();
    }
}
```

この eXtreme Scale アプリケーションでは、最初に、自動的に生成される名前で、ローカル ObjectGrid が初期化されます。次に、このアプリケーションは、BackingMap および QueryConfig を作成します。この QueryConfig は、マップに関連付けられる Java 型、マップの 1 次キーとなるフィールド名、および、オブジェクト内のデータにアクセスする方法を定義します。次に、Session を取得して ObjectMap インスタンスを取得し、トランザクション内のマップに OrderBean オブジェクトを挿入します。

キャッシュ内にデータがコミットされた後、ObjectQuery でクラス内の任意のパーシスタント・フィールドを使用して、OrderBean を検索できます。パーシスタント・フィールドとは、一時的な修飾子を持たないフィールドのことです。BackingMap には索引を定義していないため、 ObjectQuery は、Java リフレクションを使用してマップ内の各オブジェクトをスキャンする必要があります。

次のタスク

『ObjectQuery チュートリアル - ステップ 2』 では、索引を使用して照会を最適化する方法について説明します。

ObjectQuery チュートリアル - ステップ 2

以下のステップにより、1 つのマップと索引を持つ ObjectGrid、およびマップに対応するスキーマを引き続き作成できます。次に、オブジェクトをキャッシュに挿入し、後でシンプルな照会を使用してオブジェクトを検索することができます。

始める前に

チュートリアルのこのステップを続行する前に、 157 ページの『ObjectQuery チュートリアル - ステップ 1』 を完了していなければなりません。

スキーマと索引

Application.java

```
// Create an index
    HashIndex idx= new HashIndex();
    idx.setName("theItemName");
    idx.setAttributeName("itemName");
```

```
    idx.setRangeIndex(true);
    idx.setFieldAccessAttribute(true);
    orderBMap.addMapIndexPlugin(idx);
}
```

索引は、以下のように設定された com.ibm.websphere.objectgrid.plugins.index.HashIndex インスタンスにする必要があります。

- Name は任意ですが、特定の BackingMap に対しては一意にする必要があります。
- AttributeName は、フィールドの名前か、またはクラスをイントロスペクトするために索引付けエンジンが使用する Bean のプロパティーの名前です。この場合は、索引を作成するフィールドの名前です。
- RangeIndex は常に true にする必要があります。
- FieldAccessAttribute は、照会スキーマの作成時に QueryMapping オブジェクトで設定された値と一致させる必要があります。この場合は、フィールドを使用して Java オブジェクトに直接アクセスします。

照会によって itemName フィールドにフィルター操作が実行されると、照会エンジンは自動的に索引を使用します。これにより、照会の実行速度が向上し、マップ・スキャンが不要になります。 次のステップでは、索引を使用して照会を最適化する方法について説明します。

ObjectQuery チュートリアル - ステップ 3

以下のステップにより、2 つのマップを持つ ObjectGrid、および関係を備えたマップのスキーマを作成し、オブジェクトをキャッシュに挿入し、後でシンプルな照会を使用してオブジェクトを検索することができます。

始める前に

このステップを続行する前に、158 ページの『ObjectQuery チュートリアル - ステップ 2』 を完了していなければなりません。

このタスクについて

この例では、2 つのマップがあり、それぞれのマップに 1 つの Java 型がマップされています。Order マップは OrderBean オブジェクトを持ち、Customer マップは CustomerBean オブジェクトを持っています。

複数のマップを 1 つの関係で定義します。

OrderBean.java

```
public class OrderBean implements Serializable {
    String orderNumber;
    java.util.Date date;
    String customerId;
    String itemName;
    int quantity;
    double price;
}
```

OrderBean には customerName はありません。代わりに customerId があり、これは CustomerBean オブジェクトと Customer マップの主キーです。

CustomerBean.java

```
public class CustomerBean implements Serializable{
    private static final long serialVersionUID = 1L;
    String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
```

この 2 つの型あるいは 2 つのマップの間の関係は次のとおりです。

Application.java

```
public class Application
{

    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.defineMap("Order");
        og.defineMap("Customer");

        // Define the schema
        QueryConfig queryCfg = new QueryConfig();
        queryCfg.addQueryMapping(new QueryMapping(
            "Order", OrderBean.class.getName(), "orderNumber", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryMapping(new QueryMapping(
            "Customer", CustomerBean.class.getName(), "id", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryRelationship(new QueryRelationship(
             OrderBean.class.getName(), CustomerBean.class.getName(), "customerId", null));
        og.setQueryConfig(queryCfg);

        Session s = og.getSession();
        ObjectMap orderMap = s.getMap("Order");
        ObjectMap custMap = s.getMap("Customer");

        s.begin();
        CustomerBean cust = new CustomerBean();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";
        custMap.insert(cust.id, cust);

        OrderBean o = new OrderBean();
        o.customerId = cust.id;
        o.date = new java.util.Date();
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;
        orderMap.insert(o.orderNumber, o);
        s.commit();

        s.begin();
        ObjectQuery query = s.createObjectQuery(
            "SELECT c FROM Order o JOIN o.customerId as c WHERE o.itemName='Widget'");
        Iterator result = query.getResultIterator();
        cust = (CustomerBean) result.next();
        System.out.println("Found order for customer: " + cust.firstName + " " + cust.surname);
        s.commit();
    }
}
```

ObjectGrid デプロイメント記述子の対応する XML は、以下のようになります。

```
<?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="CompanyGrid">
      <backingMap name="Order"/>
```

```
      <backingMap name="Customer"/>

      <querySchema>
         <mapSchemas>
           <mapSchema
             mapName="Order"
             valueClass="com.mycompany.OrderBean"
             primaryKeyField="orderNumber"
             accessType="FIELD"/>
           <mapSchema
             mapName="Customer"
             valueClass="com.mycompany.CustomerBean"
             primaryKeyField="id"
             accessType="FIELD"/>
         </mapSchemas>
         <relationships>
           <relationship
             source="com.mycompany.OrderBean"
             target="com.mycompany.CustomerBean"
             relationField="customerId"/>
         </relationships>
      </querySchema>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

次のタスク

『ObjectQuery チュートリアル - ステップ 4』。フィールドおよびプロパティー・アクセス・オブジェクトならびに追加の関係を組み込んで現在のステップを拡張します。

ObjectQuery チュートリアル - ステップ 4

以下のステップでは、4 つのマップを持った ObjectGrid、および複数の単一方向関係と双方向関係を備えたマップのスキーマを作成する方法を示します。次に、オブジェクトをキャッシュに挿入し、後で複数の照会を使用してオブジェクトを検索することができます。

始める前に

現在のステップを続行する前に、 159 ページの『ObjectQuery チュートリアル - ステップ 3』 を完了していなければなりません。

複数のマップ関係

OrderBean.java

```
public class OrderBean implements Serializable {
    String orderNumber;
    java.util.Date date;
    String customerId;
    String itemName;
    int quantity;
    double price;
}
```

前のステップと同様、OrderBean には customerName がありません。代わりに customerId があり、これは CustomerBean オブジェクトと Customer マップの主キーです。

CustomerBean.java

```
public class CustomerBean implements Serializable{
    private static final long serialVersionUID = 1L;
    String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
```

上で指定したクラスを作成したならば、下のアプリケーションを実行できます。

Application.java

```
public class Application
{

    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.defineMap("Order");
        og.defineMap("Customer");

        // Define the schema
        QueryConfig queryCfg = new QueryConfig();
        queryCfg.addQueryMapping(new QueryMapping(
            "Order", OrderBean.class.getName(), "orderNumber", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryMapping(new QueryMapping(
            "Customer", CustomerBean.class.getName(), "id", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryRelationship(new QueryRelationship(
             OrderBean.class.getName(), CustomerBean.class.getName(), "customerId", null));
        og.setQueryConfig(queryCfg);

        Session s = og.getSession();
        ObjectMap orderMap = s.getMap("Order");
        ObjectMap custMap = s.getMap("Customer");

        s.begin();
        CustomerBean cust = new CustomerBean();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";
        custMap.insert(cust.id, cust);

        OrderBean o = new OrderBean();
        o.customerId = cust.id;
        o.date = new java.util.Date();
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;
        orderMap.insert(o.orderNumber, o);
        s.commit();

        s.begin();
        ObjectQuery query = s.createObjectQuery(
            "SELECT c FROM Order o JOIN o.customerId as c WHERE o.itemName='Widget'");
        Iterator result = query.getResultIterator();
        cust = (CustomerBean) result.next();
        System.out.println("Found order for customer: " + cust.firstName + " " + cust.surname);
        s.commit();
    }
}
```

下の XML 構成 (ObjectGrid デプロイメント記述子にある) を使用することは、上のプログラマチック・アプローチと同等です。

```
<?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="og1">
      <backingMap name="Order"/>
      <backingMap name="Customer"/>
```

```
      <querySchema>
         <mapSchemas>
           <mapSchema
             mapName="Order"
             valueClass="com.mycompany.OrderBean"
             primaryKeyField="orderNumber"
             accessType="FIELD"/>
           <mapSchema
             mapName="Customer"
             valueClass="com.mycompany.CustomerBean"
             primaryKeyField="id"
             accessType="FIELD"/>
         </mapSchemas>
         <relationships>
           <relationship
             source="com.mycompany.OrderBean"
             target="com.mycompany.CustomerBean"
             relationField="customerId"/>
         </relationships>
       </querySchema>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

Java SE セキュリティー・チュートリアル - メインページ

以下のチュートリアルにより、Java Platform, Standard Edition 環境で分散 eXtreme Scale 環境を作成できます。

始める前に

分散 eXtreme Scale 構成の基本をよく理解している必要があります。

このタスクについて

このチュートリアルでは、カタログ・サーバー、コンテナー・サーバー、およびクライアントのすべてが Java SE 環境で実行されています。このチュートリアルの各ステップは直前のステップを踏まえて進行します。このステップを一つ一つ実行して、分散 eXtreme Scale を保護し、その保護された eXtreme Scale にアクセスするシンプルな Java SE アプリケーションを作成してください。

チュートリアルの開始

1. 164 ページの『Java SE セキュリティー・チュートリアル - ステップ 1』
   - 非セキュア・カタログ・サーバーの始動
   - 非セキュア・コンテナー・サーバーの始動
   - データにアクセスするクライアントの始動
   - xsadmin を使用してマップ・サイズを表示
   - サーバーの停止
2. 167 ページの『Java SE セキュリティー・チュートリアル - ステップ 2』
   - CredentialGenerator の使用
   - Authenticator の使用
   - セキュア・カタログ・サーバーの始動
   - セキュア・コンテナー・サーバーの始動

- 保護 ObjectGrid にアクセスするクライアントの始動
- xsadmin を使用してマップ・サイズを表示
- セキュア・サーバーの停止

3. 175 ページの『Java SE セキュリティー・チュートリアル - ステップ 3』
   - JAAS 許可ポリシーの使用
4. 179 ページの『Java SE セキュリティー・チュートリアル - ステップ 4』
   - 鍵ストアおよびトラストストアの作成
   - サーバーの SSL プロパティーの構成
   - クライアントの SSL プロパティーの構成
   - xsadmin を使用してマップ・サイズを表示
   - セキュア・サーバーの停止

Java SE セキュリティー・チュートリアル - ステップ 1

このトピックではシンプルで非セキュアなサンプル について説明します。また、利用可能な統合セキュリティーを強化するため、このチュートリアルのステップごとにセキュリティー機能を順次追加していきます。

始める前に

注: チュートリアルのこのステップで必要なファイルはすべて、次のセクションに示します。

サンプルの実行

次のスクリプトを使用してカタログ・サービスを始動します。カタログ・サービスの開始に関して詳しくは、 *管理ガイド* にあるカタログ・サービスの開始に関する情報を参照してください。

1. bin ディレクトリーに移動します。`cd objectgridRoot/bin`
2. catalogServer という名前のカタログ・サーバーを始動します。
   - UNIX Linux `startOgServer.sh catalogServer`
   - Windows `startOgServer.bat catalogServer`
3. bin ディレクトリー `cd objectgridRoot/bin` に移動します。
4. 次のスクリプトを使用して c0 という名前のコンテナー・サーバーを起動します。
   - UNIX Linux
     ```
     startOgServer.sh c0 -objectGridFile
     ../xml/SimpleApp.xml -deploymentPolicyFile ../xml/SimpleDP.xml
     -catalogServiceEndpoints localhost:2809
     ```
   - Windows
     ```
     startOgServer.bat c0 -objectGridFile ../xml/SimpleApp.xml -
     deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndpoints
     localhost:2809
     ```

例

コンテナー・サーバーの始動に関して詳しくは、 *管理ガイド* にあるコンテナー・プロセスの開始に関する情報を参照してください。

カタログ・サーバーとコンテナー・サーバーが始動されたならば、次のようにしてクライアントを起動します。

1. 再度、bin ディレクトリーに移動します。
2. java -classpath ../lib/objectgrid.jar;../applib/secsample.jar com.ibm.websphere.objectgrid.security.sample.guide.SimpleApp

secsample.jar ファイルには、SimpleApp クラスが含まれています。

このプログラムの出力は次のとおりです。

```
The customer name for ID 0001 is fName lName
```

また、xsadmin を使用して「accounting」グリッドのマップ・サイズを表示することもできます。

- ディレクトリー objectgridRoot/bin に移動します。
- 次のように、オプション -mapSizes を使用して xsadmin コマンドを実行します。
  - UNIX Linux xsadmin.sh -g accounting -m mapSet1 -mapSizes
  - Windows xsadmin.bat -g accounting -m mapSet1 -mapSizes

  以下の出力が表示されます。

```
This administrative utility is provided as a sample only and is not to be
considered a fully supported component of the WebSphere eXtreme Scale
product.

Connecting to Catalog service at localhost:1099

********** Displaying Results for Grid - accounting, MapSet - mapSet1
**********

*** Listing Maps for c0 ***

Map Name: customer Partition #: 0 Map Size: 1 Shard Type: Primary

Server Total: 1

Total Domain Count: 1
```

サーバーの停止

コンテナー・サーバー

以下のコマンドを使用してコンテナー・サーバー c0 を停止します。

UNIX Linux stopOgServer.sh c0 -catalogServiceEndPoints localhost:2809

Windows stopOgServer.bat c0 -catalogServiceEndPoints localhost:2809

以下のメッセージが出力されます。

CWOBJ2512I: ObjectGrid server c0 stopped.

カタログ・サーバー

以下のコマンドを使用して、カタログ・サーバーを停止できます。

UNIX Linux stopOgServer.sh catalogServer -catalogServiceEndPoints localhost:2809

Windows stopOgServer.bat catalogServer -catalogServiceEndPoints localhost:2809

カタログ・サーバーをシャットダウンすると、次のメッセージが表示されます。

CWOBJ2512I: ObjectGrid server catalogServer stopped.

必要なファイル

下記のファイルは、SimpleApp の Java クラスです。

SimpleApp.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 (C) COPYRIGHT International Business Machines Corp. 2007-2009
package com.ibm.websphere.objectgrid.security.sample.guide;

import com.ibm.websphere.objectgrid.ClientClusterContext;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;

public class SimpleApp {

    public static void main(String[] args) throws Exception {

        SimpleApp app = new SimpleApp();
        app.run(args);
    }

    /**
     * read and write the map
     * @throws Exception
     */
    protected void run(String[] args) throws Exception {
        ObjectGrid og = getObjectGrid(args);

        Session session = og.getSession();

        ObjectMap customerMap = session.getMap("customer");

        String customer = (String) customerMap.get("0001");

        if (customer == null) {
            customerMap.insert("0001", "fName lName");
        } else {
            customerMap.update("0001", "fName lName");
        }
        customer = (String) customerMap.get("0001");

        System.out.println("The customer name for ID 0001 is " + customer);
    }

    /**
     * Get the ObjectGrid
     * @return an ObjectGrid instance
     * @throws Exception
     */
    protected ObjectGrid getObjectGrid(String[] args) throws Exception {
        ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
```

```
        // Create an ObjectGrid
        ClientClusterContext ccContext = ogManager.connect("localhost:2809", null, null);
        ObjectGrid og = ogManager.getObjectGrid(ccContext, "accounting");

        return og;

    }
}
```

このクラスの getObjectGrid メソッドは、ObjectGrid を取得し、run メソッドは、カスタマー・マップからレコードを読み取り、値を更新します。

分散環境でこのサンプルを実行する場合、ObjectGrid 記述子 XML ファイル SimpleApp.xml およびデプロイメント XML ファイル SimpleDP.xml を作成します。以下の例で、これらのファイルを取り上げています。

SimpleApp.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="accounting">
            <backingMap name="customer" readOnly="false" copyKey="true"/>
        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

下記の XML ファイルはデプロイメント環境を構成します。

SimpleDP.xml

```
<?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="accounting">
  <mapSet name="mapSet1" numberOfPartitions="1" minSyncReplicas="0" maxSyncReplicas="2" maxAsyncReplicas="1">
   <map ref="customer"/>
  </mapSet>
 </objectgridDeployment>
</deploymentPolicy>
```

これは、「accounting」という 1 つの ObjectGrid インスタンスと「customer」という 1 つのマップ (mapSet「mapSet1」内にある) を含むシンプルな ObjectGrid 構成です。 SimpleDP.xml ファイルの特徴は、1 つの区画と 0 個の最小必要複製で構成される 1 つのマップ・セットです。

次のチュートリアル・ステップ

Java SE セキュリティー・チュートリアル - ステップ 2

前のステップに基づいて、以下のトピックでは、分散 eXtreme Scale 環境でクライアント認証を実装する方法を示します。

始める前に

164 ページの『Java SE セキュリティー・チュートリアル - ステップ 1』 を完了していなければなりません。

このタスクについて

クライアント認証が有効になっていると、クライアントは eXtreme Scale サーバーに接続する前に認証されます。このセクションでは、実例を示すサンプルのコードおよびスクリプトを含め、eXtreme Scale サーバー環境におけるクライアント認証の方法を明らかにします。

他のすべての認証メカニズムと同様に、最小の認証は以下のステップで構成されています。

1. 管理者は、認証を必須とするよう構成を変更します。
2. クライアントは、サーバーにクレデンシャルを提供します。
3. サーバーは、そのクレデンシャルをレジストリーに対して認証します。

1. **クライアント・クレデンシャル**

   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、クレデンシャル API 資料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。

   さらに、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、クライアント・クレデンシャルの生成に使用されます。これは、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、「CredentialGenerator API 資料」を参照してください。

   これら 2 つのインターフェースを eXtreme Scale クライアント・ランタイム対して実装することで、クライアント・クレデンシャルを取得することができます。

   このサンプルは、eXtreme Scale が提供する以下の 2 つのサンプル・プラグインの実装を使用します。

   ```
   com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential
   com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator
   ```

   これらのプラグインに関して詳しくは、「プログラミング・ガイド」にあるクライアント認証プログラミングのトピックを参照してください。

2. **サーバー・オーセンティケーター** 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。 eXtreme Scale サーバーは、要求を処

理する前に 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、「Authenticator API 資料」を参照してください。

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、実動には使用しないようにしてください)。詳しくは、「プログラミング・ガイド」にあるクライアント認証プログラミングのオーセンティケーター・プラグインに関するトピックを参照してください。

この KeyStoreLoginAuthenticator では KeyStoreLoginModule を使用し、JAAS ログイン・モジュール KeyStoreLogin を使用して鍵ストアでユーザーを認証します。鍵ストアは、KeyStoreLoginModule クラスに対するオプションとして構成できます。以下の例では、JAAS 構成ファイル og_jaas.config に構成された keyStoreLogin 別名について示しています。

```
KeyStoreLogin{
com.ibm.websphere.objectgrid.security.plugins.builtins.KeyStoreLoginModule required
     keyStoreFile="../security/sampleKS.jks" debug = true;
};
```

以下のコマンドでは、%OBJECTGRID_HOME%/security ディレクトリーに鍵ストア sampleKS.jks を作成し、パスワードとして sampleKS1 を使用します。また、アドミニストレーター・ユーザー、マネージャー・ユーザー、およびキャッシャー・ユーザーを表す 3 つのユーザー証明書が作成され、それぞれ独自のパスワードを使用します。

a. 次の eXtreme Scale ルート・ディレクトリーに移動します。

```
cd objectgridRoot
```

b. 「security」というディレクトリーを作成します。

```
mkdir security
```

c. 新規に作成した security ディレクトリーに移動します。

```
cd security
```

d. 鍵ツール (javaHOME/bin ディレクトリー内にある) を使用して、鍵ストア sampleKS.jks にユーザー「administator」をパスワード「administrator1」で作成します。

```
keytool -genkey -v -keystore ./sampleKS.jks -storepass sampleKS1
-alias administrator -keypass administrator1
-dname CN=administrator,O=acme,OU=OGSample -validity 10000
```

e. 鍵ツール (javaHOME/bin ディレクトリー内にある) を使用して、鍵ストア sampleKS.jks にユーザー「manager」をパスワード「manager1」で作成します。

```
keytool -genkey -v -keystore ./sampleKS.jks -storepass sampleKS1
-alias manager -keypass manager1
-dname CN=manager,O=acme,OU=OGSample -validity 10000
```

f. 鍵ツール (javaHOME/bin ディレクトリー内) を使用して、鍵ストア sampleKS.jks にユーザー「cashier」をパスワード「cashier1」で作成します。

```
keytool -genkey -v -keystore ./sampleKS.jks -storepass sampleKS1
-alias cashier -keypass cashier1 -dname CN=cashier,O=acme,OU=OGSample
-validity 10000
```

クライアント・セキュリティー構成は、クライアント・プロパティー・ファイルに構成されます。以下のコマンドを使用し、%OBJECTGRID_HOME%/security ディレクトリーにコピーを作成します。

a. security ディレクトリーに移動します。

```
cd objectgridRoot/security
```

b. sampleClient.properties ファイルを client.properties ファイルにコピーします。

```
cp ../properties/sampleClient.properties client.properties
```

以下のプロパティーは、security ディレクトリーにある client.properties ファイルで強調表示されます。

a. **securityEnabled:** securityEnabled を true (デフォルト値) に設定すると、認証を含むクライアント・セキュリティーが使用可能になります。

b. **credentialAuthentication:** credentialAuthentication を Supported (デフォルト値) に設定すると、クライアントでクレデンシャル認証がサポートされます。

c. **transportType:** transportType を TCP/IP に設定すると、SSL は使用されません。

d. **singleSignOnEnabled:** false (デフォルト値) に設定します。シングル・サインオンは使用不可になります。

3. **サーバー・セキュリティー構成**

サーバー・セキュリティー構成は、セキュリティー記述子 XML ファイルおよびサーバー・セキュリティー・プロパティー・ファイルで指定されます。セキュリティー記述子 XML ファイルは、すべてのサーバー (カタログ・サーバーおよびコンテナー・サーバーを含む) に共通するセキュリティー・プロパティーを記述します。プロパティーの例の 1 つは、ユーザー・レジストリーおよび認証メカニズムを表すオーセンティケーター構成です。

このサンプルで使用する security.xml ファイルを以下に示します。

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

 <security securityEnabled="true" loginSessionExpirationTime="300" >

        <authenticator className ="com.ibm.websphere.objectgrid.security.plugins.builtins.KeyStoreLoginAuthenticator">
```

```
        </authenticator>
    </security>

</securityConfig>
```

a. **securityEnabled:** true に設定し、認証を含むサーバー・セキュリティーを有効にします。

b. **loginSessionExpirationTime:** 値を 300 (デフォルト値) に設定します。

c. **authenticator:** 以下のように、オーセンティケーター・クラス KeyStoreLoginAuthenticator をクラスター XML ファイルに追加します。

```
<authenticator className ="com.ibm.websphere.objectgrid.security.plugins.builtins.KeyStoreLoginAuthenticator">
        </authenticator>
```

d. **credentialAuthentication:** credentialAuthentication 属性を Required に設定し、サーバーが認証を必要とするようにします。

security.xml ファイルに関する詳しい説明は、 *管理ガイド* にあるセキュリティー記述子 XML ファイルに関する情報を参照してください。

サーバーのプロパティー・ファイルを security ディレクトリーにコピーします。この時点で、このファイルを変更する必要はありません。

a. security ディレクトリーに移動します。

```
cd objectgridRoot/security
```

b. サンプル objectGrid の sampleServer.properties ファイルを properties ディレクトリーから新規の server.properties ファイルにコピーします。

```
cp ../properties/containerServer.properties server.properties
```

server.properties ファイルで以下の変更を行います。

a. **securityEnabled: securityEnabled** 属性を true に設定します。

b. **transportType: transportType** 属性を TCP/IP に設定します。すなわち、SSL は使用されません。

c. **secureTokenManagerType: secureTokenManagerType** 属性を none に設定します。これで、セキュア・トークン・マネージャーが構成されなくなります。

4. **セキュア・クライアント** 以下の例に示すように、クライアント・アプリケーションを確実にサーバーに接続します。

```
// 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 (C) COPYRIGHT International Business Machines Corp. 2007-2009
package com.ibm.websphere.objectgrid.security.sample.guide;

import com.ibm.websphere.objectgrid.ClientClusterContext;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.security.config.ClientSecurityConfiguration;
import com.ibm.websphere.objectgrid.security.config.ClientSecurityConfigurationFactory;
import com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator;
import com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator;

public class SecureSimpleApp extends SimpleApp {

    public static void main(String[] args) throws Exception {
```

```
        SecureSimpleApp app = new SecureSimpleApp();
        app.run(args);
    }

    /**
     * Get the ObjectGrid
     * @return an ObjectGrid instance
     * @throws Exception
     */
    protected ObjectGrid getObjectGrid(String[] args) throws Exception {
        ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
        ogManager.setTraceFileName("logs/client.log");
        ogManager.setTraceSpecification("ObjectGrid*=all=enabled:ORBRas=all=enabled");

        // Creates a ClientSecurityConfiguration object using the specified file
        ClientSecurityConfiguration clientSC = ClientSecurityConfigurationFactory
                .getClientSecurityConfiguration(args[0]);

        // Creates a CredentialGenerator using the passed-in user and password.
        CredentialGenerator credGen = new UserPasswordCredentialGenerator(args[1], args[2]);
        clientSC.setCredentialGenerator(credGen);

        // Create an ObjectGrid by connecting to the catalog server
        ClientClusterContext ccContext = ogManager.connect("localhost:2809", clientSC, null);
        ObjectGrid og = ogManager.getObjectGrid(ccContext, "accounting");

        return og;

    }
}
```

非セキュアのアプリケーションとは、以下の 3 つの点で異なります。

a. 構成済みの client.properties ファイルを受け渡すことにより、ClientSecurityConfiguration オブジェクトを作成しています。

b. 渡されたユーザー ID とパスワードを使用することにより、UserPasswordCredentialGenerator を作成しています。

c. カタログ・サーバーに接続し、ClientSecurityConfiguration オブジェクトを受け渡すことにより ClientClusterContext から ObjectGrid を取得しています。

5. **アプリケーションの実行**

アプリケーションを実行するには、カタログ・サーバーを開始します。以下のように -clusterFile および -serverProps コマンド行オプションを発行して、セキュリティー・プロパティーを受け渡します。

a. bin ディレクトリーに移動します。

```
cd objectgridRoot/bin
```

b. カタログ・サーバーを起動します。

- UNIX Linux

```
startOgServer.sh catalogServer -clusterSecurityFile ../security/security.xml
-serverProps ../security/server.properties -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
```

- Windows

```
startOgServer.bat catalogServer -clusterSecurityFile ../security/security.xml
-serverProps ../security/server.properties -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
```

次に、以下のスクリプトを使用し、セキュア・コンテナー・サーバーを起動します。

a. 再度、bin ディレクトリーに移動します。

```
cd objectgridRoot/bin
```

b. セキュア・コンテナー・サーバーを起動します。

- Linux UNIX

```
startOgServer.sh c0 -objectgridFile ../xml/SimpleApp.xml
-deploymentPolicyFile ../xml/SimpleDP.xml
-catalogServiceEndpoints localhost:2809
-serverProps ../security/server.properties
-jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config"
```

- Windows

```
startOgServer.bat c0 -objectgridFile ../xml/SimpleApp.xml
-deploymentPolicyFile ../xml/SimpleDP.xml
-catalogServiceEndpoints localhost:2809
-serverProps ../security/server.properties
-jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config"
```

-serverProps を発行するとサーバー・プロパティー・ファイルが渡されます。

サーバーの始動後に、以下のコマンドを使用してクライアントを起動します。

a. `cd objectgridRoot/bin`

b.

```
java -classpath ../lib/objectgrid.jar;../applib/secsample.jar
  com.ibm.websphere.objectgrid.security.sample.guide.SecureSimpleApp
  ../security/client.properties manager manager1
```

`secsample.jar` ファイルには、SimpleApp クラスが含まれています。

SecureSimpleApp は、以下のリストに示されている 3 つのパラメーターを使用します。

a. `../security/client.properties` ファイルは、クライアント・セキュリティー・プロパティー・ファイルです。

b. `manager` はユーザー ID です。

c. `manager1` はパスワードです。

クラスを発行すると、以下の出力が得られます。

```
The customer name for ID 0001 is fName lName.
```

また、xsadmin を使用して「accounting」グリッドのマップ・サイズを表示することもできます。

- ディレクトリー `objectgridRoot/bin` に移動します。
- 次のように、オプション -mapSizes を使用して `xsadmin` コマンドを実行します。
  - UNIX Linux `xsadmin.sh -g accounting -m mapSet1 -username manager -password manager1 -mapSizes`
  - Windows `xsadmin.bat -g accounting -m mapSet1 -username manager -password manager1 -mapSizes`

  以下の出力が表示されます。

```
This administrative utility is provided as a sample only and is not to
be considered a fully supported component of the WebSphere eXtreme
Scale product.
```

```
Connecting to Catalog service at localhost:1099

********** Displaying Results for Grid - accounting, MapSet - mapSet1
**********

*** Listing Maps for c0 ***

Map Name: customer Partition #: 0 Map Size: 1 Shard Type: Primary

Server Total: 1

Total Domain Count: 1
```

これで、stopOgServer コマンドを使用して、コンテナー・サーバーまたはカタログ・サービス・プロセスを停止できます。ただし、セキュリティー構成ファイルを指定する必要があります。サンプル・クライアント・プロパティー・ファイルは、以下の 2 つのプロパティーを定義して、ユーザー ID とパスワードのクレデンシャル (manager/manager1) を生成します。

```
credentialGeneratorClass=com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator
credentialGeneratorProps=manager manager1
```

次のコマンドを使用してコンテナー c0 を停止します。

- UNIX Linux `stopOgServer.sh c0 -catalogServiceEndPoints localhost:2809 -clientSecurityFile ..¥security¥client.properties`
- Windows `stopOgServer.bat c0 -catalogServiceEndPoints localhost:2809 -clientSecurityFile ..¥security¥client.properties`

-clientSecurityFile オプションを指定しないと、次のメッセージを伴う例外が表示されます。

```
>> SERVER (id=39132c79, host=9.10.86.47) TRACE START:

>> org.omg.CORBA.NO_PERMISSION: Server requires credential
authentication but there is no security context from the client. This
usually happens when the client does not pass a credential the server.

vmcid: 0x0

minor code: 0

completed: No
```

また、以下のコマンドを使用してカタログ・サーバーをシャットダウンすることもできます。ただし、チュートリアルの次のステップに続行する場合は、このカタログ・サーバーを実行させたままにしておいてかまいません。

- UNIX Linux `stopOgServer.sh catalogServer -catalogServiceEndPoints localhost:2809 -clientSecurityFile ..¥security¥client.properties`
- Windows `stopOgServer.bat catalogServer -catalogServiceEndPoints localhost:2809 -clientSecurityFile ..¥security¥client.properties`

カタログ・サーバーをシャットダウンすると、次の出力が表示されます。

```
CWOBJ2512I: ObjectGrid server catalogServer stopped
```

これで、認証を有効にすることにより、正常にシステムが部分的にセキュアになりました。サーバーを構成してユーザー・レジストリーをプラグインし、クライアントを構成してクライアント・クレデンシャルを提供するようにし、クライアント・プロパティー・ファイルおよびクラスター XML ファイルを変更して認証を有効にしています。

無効なパスワードを入力すると、ユーザー名およびパスワードが誤っていることを示す例外が表示されます。

クライアント認証について詳しくは、 *管理ガイド* にあるアプリケーション・クライアント認証に関する情報を参照してください。

次のチュートリアル・ステップ

Java SE セキュリティー・チュートリアル - ステップ 3

前のステップのようにクライアントを認証した後、 eXtreme Scale 許可メカニズムによりセキュリティー特権を付与することができます。

始める前に

このタスクを続行する前に 167 ページの『Java SE セキュリティー・チュートリアル - ステップ 2』 を完了している必要があります。

このタスクについて

このチュートリアルの前のステップでは、eXtreme Scale グリッドで認証を使用可能にする方法について説明しました。この結果として、非認証クライアントは、サーバーに接続することができず、システムに要求の実行依頼をすることができません。ただし、認証されている各クライアントは、ObjectGrid マップに格納されているデータの読み取り、書き込み、削除など、サーバーに対して同じアクセス権または特権を持っています。クライアントは、どのような照会でも実行できます。このセクションでは、eXtreme Scale 許可を使用してさまざまな認証済みユーザーに特権を付与する方法について説明します。

他の多くのシステムと同様、eXtreme Scale でもアクセス権ベースの許可メカニズムを採用しています。 WebSphere eXtreme Scale には、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。このトピックでは、MapPermission について説明します。許可の完全なカテゴリーについては、クライアント許可リファレンスを参照してください。

eXtreme Scale では、com.ibm.websphere.objectgrid.security.MapPermission クラスは eXtreme Scale リソース、特に ObjectMap インターフェースまたは JavaMap インターフェースのメソッドに対する許可を表しています。 WebSphere eXtreme Scale は、以下の許可ストリングを定義し、ObjectMap および JavaMap のメソッドにアクセスします。

- read: マップからデータを読み取る許可を与えます。

- write: マップのデータを更新する許可を与えます。
- insert: マップにデータを挿入する許可を与えます。
- remove: マップからデータを削除する許可を与えます。
- invalidate: マップからのデータを無効にする許可を与えます。
- all: read、write、insert、remove、および invalidate に対するすべての許可を与えます。

クライアントが ObjectMap または JavaMap のメソッドを呼び出すと許可が行われます。 eXtreme Scale ランタイムは、メソッドごとにそれぞれのマップ許可を検査します。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。

このチュートリアルでは、JAAS 許可を使用して、さまざまなユーザーに対する許可マップ・アクセスを付与する方法について説明します。

1. **eXtreme Scale 許可を使用可能に設定** ObjectGrid で許可を使用可能にするには、XML ファイルで特定の ObjectGrid に対して、securityEnabled を true に設定する必要があります。ObjectGrid でのセキュリティーとは、許可のことを表しています。以下のコマンドを使用して、セキュリティーが使用可能な新しい ObjectGrid XML を作成します。

   a. cd objectgridRoot/bin

   b. cp SimpleApp.xml SecureSimpleApp.xml

   次に、以下の XML に示すように、ObjectGrid レベルで securityEnabled=″true″ を追加します。

```
<?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" securityEnabled="true">
            <backingMap name="customer" readOnly="false" copyKey="true"/>
        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

2. **許可ポリシーの定義** 直前のステップに記載されているクライアントごとの認証のセクションで、鍵ストアに 3 人のユーザー (cashier、manager、および administrator) を作成したことを思い出してください。この例では、ユーザー「cashier」はすべてのマップに対する read 許可のみを持ち、ユーザー「manager」は all 許可を持つという点について説明します。 この例では、JAAS 許可が使用されます。JAAS 許可では許可ポリシー・ファイルを使用して、プリンシパルに許可を付与します。security ディレクトリーで以下の og_auth.policy ファイルが定義されます。

```
grant codebase "http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction"
    principal javax.security.auth.x500.X500Principal "CN=cashier,O=acme,OU=OGSample" {
    permission com.ibm.websphere.objectgrid.security.MapPermission "accounting.*", "read ";
};

grant codebase "http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction"
    principal javax.security.auth.x500.X500Principal "CN=manager,O=acme,OU=OGSample" {
    permission com.ibm.websphere.objectgrid.security.MapPermission "accounting.*", "all";
};
```

   注:

- コードベース「http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction」は、ObjectGrid 用に特別に予約されている URL です。プリンシパルに付与されているすべての ObjectGrid 許可では、この特別なコードベースを使用します。
- 1 番目の grant ステートメントでは、「read」マップ許可がプリンシパル「CN=cashier,O=acme,OU=OGSample」に付与されるので、cashier には、ObjectGrid アカウンティングのすべてのマップに対するマップ read 許可のみが付与されます。
- 2 番目の grant ステートメントでは「all」マップ許可がプリンシパル「CN=manager,O=acme,OU=OGSample」に付与されるので、manager には、ObjectGrid アカウンティングのマップに対する all 許可が付与されます。

これで、許可ポリシーを使用してサーバーを起動することができます。次のように標準の -D プロパティーを使用して JAAS 許可ポリシー・ファイルを設定することができます。-Djava.security.auth.policy=../security/ogAuth.policy

3. **アプリケーションの実行**

   上記のファイルを作成すると、アプリケーションを実行することができます。

   以下のコマンドを使用して、カタログ・サーバーを始動します。カタログ・サービスの開始に関して詳しくは、 *管理ガイド* にあるカタログ・サービスの開始に関する情報を参照してください。

   a. bin ディレクトリーに移動します。`cd objectgridRoot/bin`

   b. カタログ・サーバーを始動します。

      - UNIX Linux
      ```
      startOgServer.sh catalogServer
      -clusterSecurityFile ../security/security.xml -serverProps
      ../security/server.properties -jvmArgs
      -Djava.security.auth.login.config="../security/og_jaas.config"
      ```
      - Windows
      ```
      startOgServer.bat catalogServer -clusterSecurityFile
      ../security/security.xml -serverProps ../security/server.properties
      -jvmArgs -Djava.security.auth.login.config="../security/
      og_jaas.config"
      ```

      `security.xml` ファイルおよび `server.properties` ファイルは、このチュートリアルの前のステップで作成されています。

      次に、以下のスクリプトを使用して、セキュア・コンテナー・サーバーを起動します。

   c. 再度、bin ディレクトリーに移動します。

   d.

      - UNIX Linux
      ```
      # startOgServer.sh c0 -objectGridFile
      ../xml/SecureSimpleApp.xml -deploymentPolicyFile
      ../xml/SimpleDP.xml -catalogServiceEndpoints localhost:2809
      -serverProps ../security/server.properties -jvmArgs
      -Djava.security.auth.login.config="../security/og_jaas.config"
      -Djava.security.auth.policy="../security/og_auth.policy"
      ```

- Windows `startOgServer.bat c0 -objectGridFile ../xml/SecureSimpleApp.xml -deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndpoints localhost:2809 -serverProps ../security/server.properties -jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config" -Djava.security.auth.policy="../security/og_auth.policy"`

前のコンテナー・サーバー始動コマンドとの以下の違いに注意してください。

- `SimpleApp.xml` の代わりに `SecureSimpleApp.xml` を使用します。
- 別の `-Djava.security.auth.policy` を追加して、JAAS 許可ポリシー・ファイルをコンテナー・サーバー・プロセスに設定します。

次に、このチュートリアルの直前のステップで使用したのと同じコマンドを使用します。

a. bin ディレクトリーに移動します (同上)。

b. `java -classpath ../lib/objectgrid.jar;../applib/secsample.jar com.ibm.websphere.objectgrid.security.sample.guide.SecureSimpleApp ../security/client.properties manager manager1`

   ユーザー「manager」にはアカウンティング ObjectGrid のマップに対するすべての許可が付与されているため、アプリケーションは正しく実行されます。

   次に、ユーザー「manager」を使用する代わりにユーザー「cashier」を使用して、クライアント・アプリケーションを開始します。

c. 再度、bin ディレクトリーに移動します。

d. `java -classpath ../lib/objectgrid.jar;../applib/secsample.jar com.ibm.ws.objectgrid.security.sample.guide.SecureSimpleApp ../security/client.properties cashier cashier1`

以下の例外が発生します。

```
Exception in thread "P=387313:O=0:CT" com.ibm.websphere.objectgrid.TransactionException:
rolling back transaction, see caused by exception
 at com.ibm.ws.objectgrid.SessionImpl.rollbackPMapChanges(SessionImpl.java:1422)
  at com.ibm.ws.objectgrid.SessionImpl.commit(SessionImpl.java:1149)
  at com.ibm.ws.objectgrid.SessionImpl.mapPostInvoke(SessionImpl.java:2260)
  at com.ibm.ws.objectgrid.ObjectMapImpl.update(ObjectMapImpl.java:1062)
  at com.ibm.ws.objectgrid.security.sample.guide.SimpleApp.run(SimpleApp.java:42)
 at com.ibm.ws.objectgrid.security.sample.guide.SecureSimpleApp.main(SecureSimpleApp.java:27)
Caused by: com.ibm.websphere.objectgrid.ClientServerTransactionCallbackException:
   Client Services - received exception from remote server:
     com.ibm.websphere.objectgrid.TransactionException: transaction rolled back,
   see caused by Throwable
        at com.ibm.ws.objectgrid.client.RemoteTransactionCallbackImpl.processReadWriteResponse(
            RemoteTransactionCallbackImpl.java:1399)
        at com.ibm.ws.objectgrid.client.RemoteTransactionCallbackImpl.processReadWriteRequestAndResponse(
            RemoteTransactionCallbackImpl.java:2333)
        at com.ibm.ws.objectgrid.client.RemoteTransactionCallbackImpl.commit(RemoteTransactionCallbackImpl.java:557)
        at com.ibm.ws.objectgrid.SessionImpl.commit(SessionImpl.java:1079)
        ... 4 more
Caused by: com.ibm.websphere.objectgrid.TransactionException: transaction rolled back, see caused by Throwable
        at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processLogSequence(ServerCoreEventProcessor.java:1133)
        at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processReadWriteTransactionRequest
     (ServerCoreEventProcessor.java:910)
        at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processClientServerRequest(ServerCoreEventProcessor.java:1285)

        at com.ibm.ws.objectgrid.ShardImpl.processMessage(ShardImpl.java:515)
        at com.ibm.ws.objectgrid.partition.IDLShardPOA._invoke(IDLShardPOA.java:154)
```

```
        at com.ibm.CORBA.poa.POAServerDelegate.dispatchToServant(POAServerDelegate.java:396)
        at com.ibm.CORBA.poa.POAServerDelegate.internalDispatch(POAServerDelegate.java:331)
        at com.ibm.CORBA.poa.POAServerDelegate.dispatch(POAServerDelegate.java:253)
        at com.ibm.rmi.iiop.ORB.process(ORB.java:503)
        at com.ibm.CORBA.iiop.ORB.process(ORB.java:1553)
        at com.ibm.rmi.iiop.Connection.respondTo(Connection.java:2680)
        at com.ibm.rmi.iiop.Connection.doWork(Connection.java:2554)
        at com.ibm.rmi.iiop.WorkUnitImpl.doWork(WorkUnitImpl.java:62)
        at com.ibm.rmi.iiop.WorkerThread.run(ThreadPoolImpl.java:202)
        at java.lang.Thread.run(Thread.java:803)
Caused by: java.security.AccessControlException: Access denied (
   com.ibm.websphere.objectgrid.security.MapPermission accounting.customer write)
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:155)
        at com.ibm.ws.objectgrid.security.MapPermissionCheckAction.run(MapPermissionCheckAction.java:141)
        at java.security.AccessController.doPrivileged(AccessController.java:275)
        at javax.security.auth.Subject.doAsPrivileged(Subject.java:727)
        at com.ibm.ws.objectgrid.security.MapAuthorizer$1.run(MapAuthorizer.java:76)
        java.security.AccessController.doPrivileged(AccessController.java:242)
        at com.ibm.ws.objectgrid.security.MapAuthorizer.check(MapAuthorizer.java:66)
        at com.ibm.ws.objectgrid.security.SecuredObjectMapImpl.checkMapAuthorization(SecuredObjectMapImpl.java:429)
        at com.ibm.ws.objectgrid.security.SecuredObjectMapImpl.update(SecuredObjectMapImpl.java:490)
        at com.ibm.ws.objectgrid.SessionImpl.processLogSequence(SessionImpl.java:1913)
        at com.ibm.ws.objectgrid.SessionImpl.processLogSequence(SessionImpl.java:1805)
        at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processLogSequence(ServerCoreEventProcessor.java:1011)
        ... 14 more
```

これは、ユーザー「cashier」に write 許可が付与されていないため、map customer を更新できないことが原因です。

これで、システムは許可をサポートするようになりました。許可ポリシーを定義して、ユーザーごとに各種の許可を付与することができます。許可について詳しくは、 *プログラミング・ガイド* にあるアプリケーション・クライアント許可に関する情報を参照してください。

次のステップ

Java SE セキュリティー・チュートリアル - ステップ 4

以下のステップでは、ご使用環境のエンドポイント間の通信にセキュリティー層を使用可能にする方法について説明します。

始める前に

このタスクを続行する前に 175 ページの『Java SE セキュリティー・チュートリアル - ステップ 3』 を完了している必要があります。

このタスクについて

eXtreme Scale トポロジーは、ObjectGrid エンドポイント (クライアント、コンテナー・サーバー、およびカタログ・サーバー) 間のセキュア通信のために Transport Layer Security/Secure Sockets Layer (TLS/SSL) をサポートします。このチュートリアル・ステップでは、それ以前のステップに基づいてトランスポート・セキュリティーを使用可能にします。

1. **TLS/SSL 鍵および鍵ストアの作成**

   トランスポート・セキュリティーを使用可能にするためには、鍵ストアとトラストストアを作成する必要があります。この練習課題では、鍵ストアとトラストストアのペアのみを作成します。これらのストアは ObjectGrid クライアント、コンテナー・サーバー、およびカタログ・サーバーのために使用されるもので、JDK 鍵ツールを使用して作成されます。

- *鍵ストアに秘密鍵を作成します*

  ```
  keytool -genkey -alias ogsample -keystore key.jks -storetype JKS
  -keyalg rsa -dname "CN=ogsample, OU=Your Organizational Unit, O=Your
  Organization, L=Your City, S=Your State, C=Your Country" -storepass
  ogpass -keypass ogpass -validity 3650
  ```

  このコマンドを使用すると、「ogsample」という鍵を含む鍵ストア key.jks が作成されます。この鍵ストア key.jks は SSL 鍵ストアとして使用されます。
- *パブリック証明書をエクスポートします*

  ```
  keytool -export -alias ogsample -keystore key.jks -file temp.key
  -storepass ogpass
  ```

  このコマンドを使用すると、「ogsample」という鍵のパブリック証明書が抽出されて、ファイル temp.key に格納されます。
- *クライアントのパブリック証明書をトラストストアにインポートします*

  ```
  keytool -import -noprompt -alias ogsamplepublic -keystore trust.jks
  -file temp.key -storepass ogpass
  ```

  このコマンドを使用すると、パブリック証明書が鍵ストア trust.jks に追加されます。この trust.jks は SSL トラストストアとして使用されます。

2. **ObjectGrid プロパティー・ファイルを構成します**

   このステップでは、トランスポート・セキュリティーを使用可能にするように ObjectGrid プロパティー・ファイルを構成する必要があります。

   まず、key.jks ファイルと trust.jks ファイルを objectgridRoot/security ディレクトリーにコピーします。

   client.properties および server.properties ファイルで以下のプロパティーを設定します。

   ```
   transportType=SSL-Required

   alias=ogsample
   contextProvider=IBMJSSE2
   protocol=SSL
   keyStoreType=JKS
   keyStore=../security/key.jks
   keyStorePassword=ogpass
   trustStoreType=JKS
   trustStore=../security/trust.jks
   trustStorePassword=ogpass
   ```

   transportType: transportType の値は「SSL-Required」に設定されます。つまり、トランスポートに SSL が必要となります。したがって、すべての ObjectGrid エンドポイント (クライアント、カタログ・サーバー、およびコンテナー・サーバー) で SSL 構成が設定され、すべてのトランスポート通信が暗号化されます。

   その他のプロパティーは SSL 構成を設定するために使用されます。詳しい説明は、*管理ガイド* にある Transport Layer Security および Secure Sockets Layer に

関する情報を参照してください。必ずこのトピックの説明に従って、orb.properties ファイルを更新してください。

必ずこのページに従って、orb.properties ファイルを更新してください。

server.properties ファイルでは、別のプロパティー clientAuthentication を追加し、それを false に設定する必要があります。サーバー・サイドでは、クライアントを信頼する必要はありません。

```
clientAuthentication=false
```

3. **アプリケーションの実行**

使用するコマンドは 175 ページの『Java SE セキュリティー・チュートリアル - ステップ 3』 トピックのコマンドと同じです。

以下のコマンドを使用してカタログ・サーバーを始動します。

a. bin ディレクトリーに移動します。cd objectgridRoot/bin

b. カタログ・サーバーを始動します。

- Linux UNIX

```
startOgServer.sh catalogServer -clusterSecurityFile ../security/security.xml
-serverProps ../security/server.properties -JMXServicePort 11001
-jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config"
```

- Windows

```
startOgServer.bat catalogServer -clusterSecurityFile ../security/security.xml
-serverProps ../security/server.properties -JMXServicePort 11001 -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
```

security.xml ファイルおよび server.properties ファイルは、 167 ページの『Java SE セキュリティー・チュートリアル - ステップ 2』で作成されています。

-JMXServicePort オプションを使用して、サーバーの JMX ポートを明示的に指定してください。このオプションは、xsadmin コマンドを使用するために必要です。

セキュア ObjectGrid コンテナー・サーバーを実行します。

c. 再度、bin ディレクトリーに移動します。cd objectgridRoot/bin

d.

- Linux UNIX

```
startOgServer.sh c0 -objectGridFile ../xml/SecureSimpleApp.xml
-deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndpoints
localhost:2809 -serverProps ../security/server.properties
-JMXServicePort 11002 -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
-Djava.security.auth.policy="../security/og_auth.policy"
```

- Windows

```
startOgServer.bat c0 -objectGridFile ../xml/SecureSimpleApp.xml
-deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndpoints localhost:2809
-serverProps ../security/server.properties -JMXServicePort 11002
-jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config"
-Djava.security.auth.policy="../security/og_auth.policy"
```

前のコンテナー・サーバー始動コマンドとの以下の違いに注意してください。

- SimpleApp.xml の代わりに SecureSimpleApp.xml を使用します。
- 別の -Djava.security.auth.policy を追加して、JAAS 許可ポリシー・ファイルをコンテナー・サーバー・プロセスに設定します。

クライアント認証のために次のコマンドを実行します。

a. `cd objectgridRoot/bin`

b.

```
javaHome/java -classpath ../lib/objectgrid.jar;../applib/secsample.jar
com.ibm.websphere.objectgrid.security.sample.guide.SecureSimpleApp
../security/client.properties manager manager1
```

ユーザー「manager」にはアカウンティング ObjectGrid のすべてのマップに対する許可が付与されているため、アプリケーションは正常に実行されます。

また、xsadmin を使用して「accounting」グリッドのマップ・サイズを表示することもできます。

- ディレクトリー `objectgridRoot/bin` に移動します。
- 次のように、オプション -mapSizes を使用して `xsadmin` コマンドを実行します。

  _ UNIX Linux

  ```
  xsadmin.sh -g accounting -m mapSet1 -mapsizes -p 11001 -ssl
  -trustpath ..¥security¥trust.jks -trustpass ogpass -trusttype jks
  -username manager -password manager1
  ```

  _ Windows

  ```
  xsadmin.bat -g accounting -m mapSet1 -mapsizes -p 11001 -ssl
  -trustpath ..¥security¥trust.jks -trustpass ogpass -trusttype jks
  -username manager -password manager1
  ```

  ここで、-p 11001 を使用してカタログ・サービスの JMX ポートを指定することに注意してください。

  以下の出力が表示されます。

```
This administrative utility is provided as a sample only and is not to
be considered a fully supported component of the WebSphere eXtreme Scale product.
Connecting to Catalog service at localhost:1099
********** Displaying Results for Grid - accounting, MapSet - mapSet1 **********
*** Listing Maps for c0 ***
Map Name: customer Partition #: 0 Map Size: 1 Shard Type: Primary
Server Total: 1
Total Domain Count: 1
```

間違った鍵ストアを使用したアプリケーションの実行

鍵ストア内の秘密鍵のパブリック証明書がトラストストアに含まれていないと、鍵がトラステッド鍵でありえないことを示す例外が発生します。

このことを示すために、もう 1 つの鍵ストア key2.jks を作成します。

```
keytool -genkey -alias ogsample -keystore key2.jks -storetype JKS
-keyalg rsa -dname "CN=ogsample, OU=Your Organizational Unit, O=Your
Organization, L=Your City, S=Your State, C=Your Country" -storepass
ogpass -keypass ogpass -validity 3650
```

次に、server.properties を変更して、 keyStore が、この新規の鍵ストア key2.jks をポイントするようにします。

```
keyStore=../security/key2.jks
```

次のコマンドを実行してカタログ・サーバーを始動します。

a. bin ディレクトリーに移動します。`cd objectgridRoot/bin`

b. カタログ・サーバーを始動します。

Linux UNIX

```
startOgServer.sh c0 -objectGridFile ../xml/SecureSimpleApp.xml
-deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndpoints localhost:2809
-serverProps ../security/server.properties -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
-Djava.security.auth.policy="../security/og_auth.policy"
```

Windows

```
startOgServer.bat c0 -objectGridFile ../xml/SecureSimpleApp.xml
-deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndpoints localhost:2809
 -serverProps ../security/server.properties -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
-Djava.security.auth.policy="../security/og_auth.policy"
```

次の例外が表示されます。

```
Caused by: com.ibm.websphere.objectgrid.ObjectGridRPCException:
    com.ibm.websphere.objectgrid.ObjectGridRuntimeException:
        SSL connection fails and plain socket cannot be used.
```

最後に、key.jks ファイルを使用するように server.properties ファイルを元に戻します。

第 9 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]

[ハ行]

[マ行]

［ラ行］

E

H

J

T

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