WebSphere® eXtreme Scale バージョン 7.1
製品概要

WebSphere eXtreme Scale 製品概要

IBM

本書は、WebSphere eXtreme Scale のバージョン 7、リリース 1、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 Version7.1
Product Overview
WebSphere eXtreme Scale Product
Overview

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 2010.7

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。

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ます。 WebSphere eXtreme Scale を使用すると、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も手に入れることができます。

分散オブジェクト・キャッシュを使用することによって、弾力性のあるスケーラビリティーが可能になります。 弾力性のあるスケーラビリティーによって、データ・グリッドはそれ自体をモニターし、管理します。 データ・グリッドは、サーバーの追加と除去によってトポロジーのスケールアウトまたはスケールインを実行できます。これにより、必要に応じてメモリー、ネットワーク・スループット、および処理能力の増加や減少を行います。 スケールアウト処理が開始すると、データ・グリッドの実行中に、再始動を必要とせずに、キャパシティーが追加されます。 逆に、スケールイン処理は即時にキャパシティーを除去します。 データ・グリッドは、障害から自動的にリカバリーすることで自己修復も行います。

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、コンストレインド・ツリー・スキーマに相当する区画に分割します。 コンストレインド・ツリー・スキーマは、エンティティー間のリレーションシップを記述します。 区画を使用している場合、エンティティー・リレーションシップはツリー・データ構造をモデル化します。このツリーの先頭がルート・エンティティーで、区画化される唯一のエンティティーです。 ルート・エンティティーの他の子はすべて、ルート・エンティティーと同一の区画に保管されています。それぞれの区画は、プライマリー・コピーまたは断片として存在します。区画には、データのバックアップ用複製断片も含まれます。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。

後書きキャッシュ機能により、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ます。 このフロントエンド・キャッシュを使用して、データベースの負荷と競合を減らしながら、スループットが増加します。 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 のクライアントとデータ・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データ・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、データ・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。

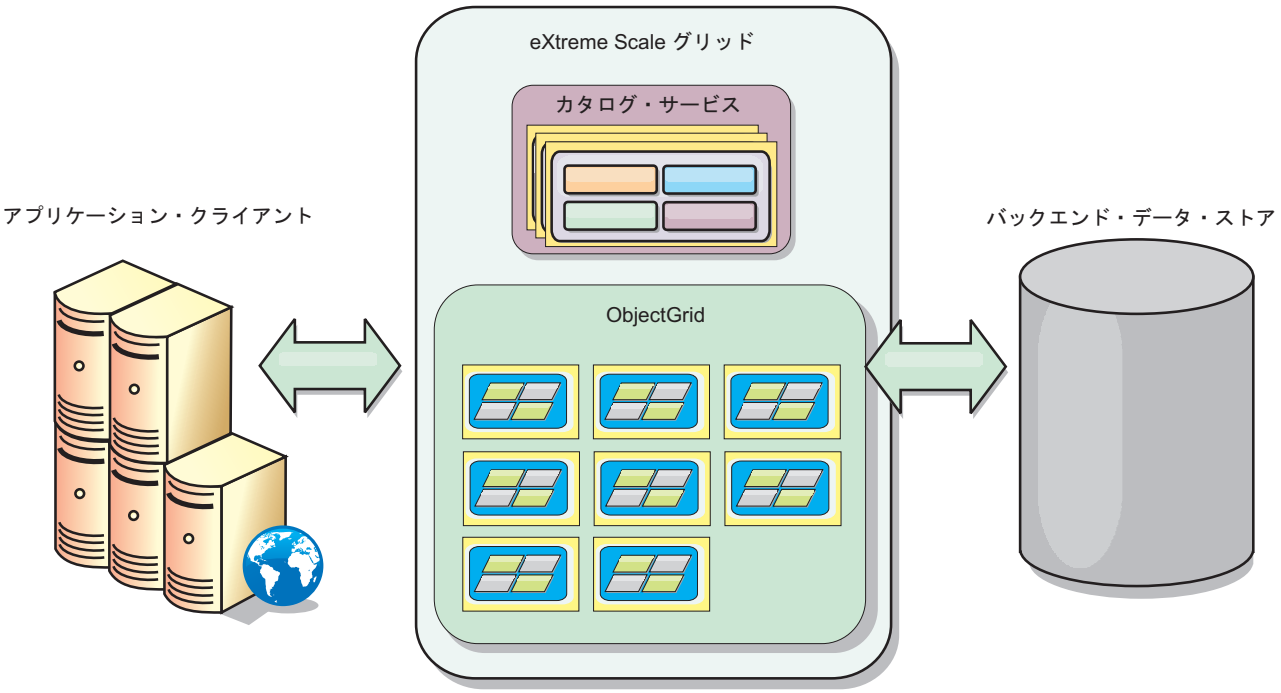


図 1. 概要レベルのトポロジー

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。データ・グリッドは、アプリケーションやアプリケーション・サーバーのメモリーまたはアドレス・スペースによる制

限はなく、それらに影響を与えることはありません。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

データ・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。データ・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) と一緒に保管します。その後、キーを使用して値が取り出されます。eXtreme Scale では、マップはこのようなキーと値のペアのエントリーで構成されます。

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 Java 仮想マシンまたはサーバーを使用する大容量の分散キャッシュにいたるまで、多くのデータ・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ます。 `SELECT ... FROM ... WHERE` ステートメントを使用する SQL に似た照会言語を使用して、こ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ち、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という例が考えられます。 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 Scale1 は、キャッシュにエンティティーを保管するための、EntityManager プログラミング・インターフェースもサポートします。 このプログラミング・インターフェースは、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 インターフェース上で 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。エンティティーをデータ・グリッドにパーシストすること、またはデータ・グリッドから除去することが 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 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。

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 用語ではデータ・グリッドまたは ObjectGrid とも呼ばれる) に対して、データ・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) メソッドでキャッシュに値を入れ、次に map.get(key) メソッドで値を取り出します。

使用する WebSphere eXtreme Scale アプリケーションを設計する際に利用できるベスト・プラクティスについては、developerWorks®: ハイパフォーマンスで高い回復力を持つ WebSphere eXtreme Scale アプリケーションを作成するための原則とベスト・プラクティス (Principles and best practices for building high performing and highly resilient WebSphere eXtreme Scale application) の記事を参照してください。

このリリースの新機能と非推奨機能

WebSphere eXtreme Scale には、動的キャッシュとの統合、バイト配列マップなど、バージョン 7.1 の多くの新たなフィーチャーが含まれています。

WebSphere eXtreme Scale バージョン 7.1 の新機能

表 1. WebSphere eXtreme Scale バージョン 7.1 の新フィーチャー

機能	説明
DB2® クライアント情報の統合	eXtreme Scale JPA ローダー・プラグインを DB2 と統合し、DB2 をバックエンド・データベースとして使用すると、WebSphere eXtreme Scale 情報 (ユーザー名、ワークステーション名、アプリケーション名、およびアカウンティング情報) を DB2 パフォーマンス・モニターで使用できるようになります。このフィーチャーにより、DB2 に対するクライアント情報の構成を使用可能および使用不可にすることができます。この機能は、デフォルトでは使用不可です。詳しくは、44 ページの『ローダー』を参照してください。
カタログ・サービス・ドメインの構成	カタログ・サービス・ドメインは、WebSphere Application Server の管理コンソールまたは管理用タスクを使用して構成することができます。カタログ・サービス・ドメインはグループを定義します。詳しくは、「*管理ガイド*」のカタログ・サービス・ドメインの作成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 7.1 の新フィーチャー (続き)

機能	説明
マルチ・マスター複製	複数のデータ・センターを非同期にリンクさせ、データ・センターによるデータへのローカル・アクセスを可能にし、高可用性を維持することができます。詳しくは、25 ページの『マルチ・マスター・グリッド複製トポロジー (AP)』を参照してください。
最終更新時間 TTL Evictor	TTL Evictor が更新され、エントリーが更新された時間のトラッキングが可能になり、存続時間 Evictor が拡張されました。詳しくは、「*プログラミング・ガイド*」の TimeToLive (TTL) Evictor に関する情報を参照してください。
メモリー内グリッドの usedBytes 統計	BackingMap 内のキャッシュ・エントリーが使用するメモリー量は、すべての統計プロバイダーを使用してトラッキングできます。詳しくは、「*管理ガイド*」のキャッシュ・メモリー消費量の見積もりに関する情報を参照してください。
動的統計	統計はオンデマンドで使用可能にしたり使用不可にしたりできます。詳しくは、「*管理ガイド*」の MBean を使用したモニターに関する情報を参照してください。
コンソールのモニター	グラフィカル・モニター・コンソールは、WebSphere eXtreme Scale サーバー統計に関する現在のビューおよび履歴のビューを提供します。詳しくは、「*管理ガイド*」の Web コンソールに関する情報を参照してください。
改善された HTTP セッション・マネージャー	HTTP セッション・マネージャーの構成が簡単になりました。HTTP セッション・マネージャーの構成が WebSphere Application Server 管理コンソールでできるようになりました。詳しくは、「*管理ガイド*」の HTTP セッション・マネージャーの構成に関する情報を参照してください。
マルチホームのクライアント・サポート	クライアントは特定のネットワーク・アダプターを使用するよう構成することができます。詳しくは、「*管理ガイド*」のクライアント・プロパティー・ファイルに関する情報を参照してください。
ISA Lite	IBM® Support Assistant Lite for WebSphere eXtreme Scale は、問題判別シナリオのための自動データ収集および症状分析支援を提供します。詳しくは、「*管理ガイド*」の IBM Support Assistant for WebSphere eXtreme Scale に関する情報を参照してください。
REST	REST データ・サービスは、非 Java クライアントに Open Data Protocol (OData) に対応した eXtreme Scale データへのアクセスを提供し、Microsoft® WCF Data Services との完全な互換性を実現します。詳しくは、185 ページの『第 8 章 REST データ・サービスの概要』を参照してください。
クライアント単独のインストール	WebSphere eXtreme Scale クライアントを独立してインストールできるようになりました。これにより、WebSphere eXtreme Scale アプリケーションのインストール占有スペースが減ります。詳しくは、「*管理ガイド*」の WebSphere eXtreme Scale のインストールとデプロイメント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
	WebSphere Application Server 管理コンソールでカタログ・サービス・ドメインを作成します。これにより、カスタム・プロパティーを使用する場合と同じ構成が作成されます。詳しくは、「*管理ガイド*」のカタログ・サービス・ドメインの作成に関する情報を参照してください。
	代わりに、CatalogServiceManagementMBean を使用します。
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。

表 2. 非推奨機能 (続き)

非推奨機能	推奨されるマイグレーション・アクション
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。

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。

アプリケーション・デプロイメントの概要

WebSphere eXtreme Scaleを実稼働環境で使用する前に、デプロイメントを最適化するための以下の問題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?

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 Java 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。

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。

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。

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。

第 2 章 キャッシュの概要

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、データ・グリッド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ュ概念やシナリオを説明し、データ・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解説します。

キャッシング・アーキテクチャー: マップ、コンテナー、クライアント、およびカタログ

WebSphere eXtreme Scale を使用して、アーキテクチャーはローカルのメモリー内でのデータ・キャッシング、または分散クライアント/サーバーでのデータ・キャッシングを使用できます。

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。

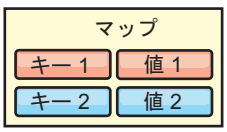


図 2. マップ

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。

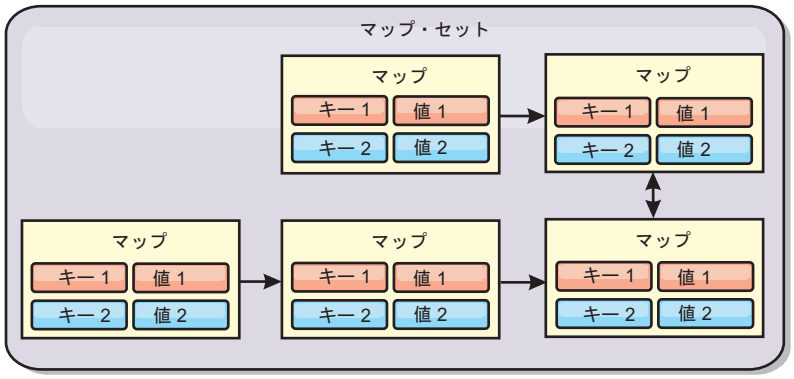


図 3. マップ・セット

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関係を記述したメタデータです。

WebSphere 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タイプ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。WebSphere 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。

WebSphere 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。WebSphere 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。

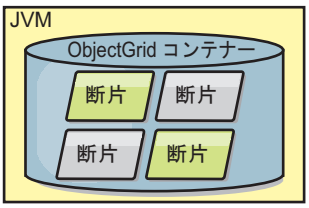


図 4. コンテナー

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。WebSphere 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でデータ・セット全体を eXtreme Scale が再ハッシュすることはできません。 一般に、区画の数は多めに見積もって構いません。

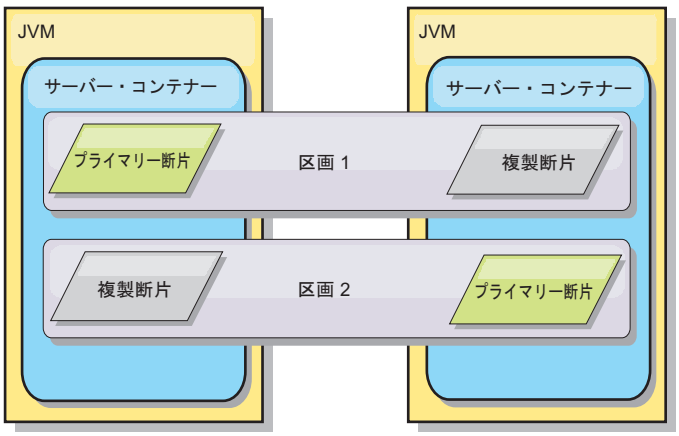


図 5. 区画

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。

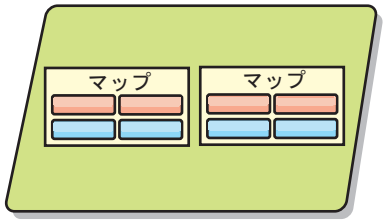


図 6. 断片

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。

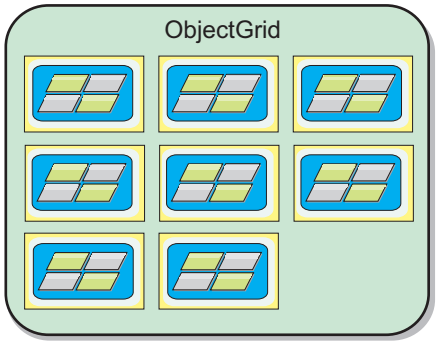


図 7. ObjectGrid

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、動的カタログ・サービスは、データをホスティングする適切なサーバーにクライアントを経路指定します。 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、複数の eXtreme Scale クライアントまたはサーバーを管理することができます。
- 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。

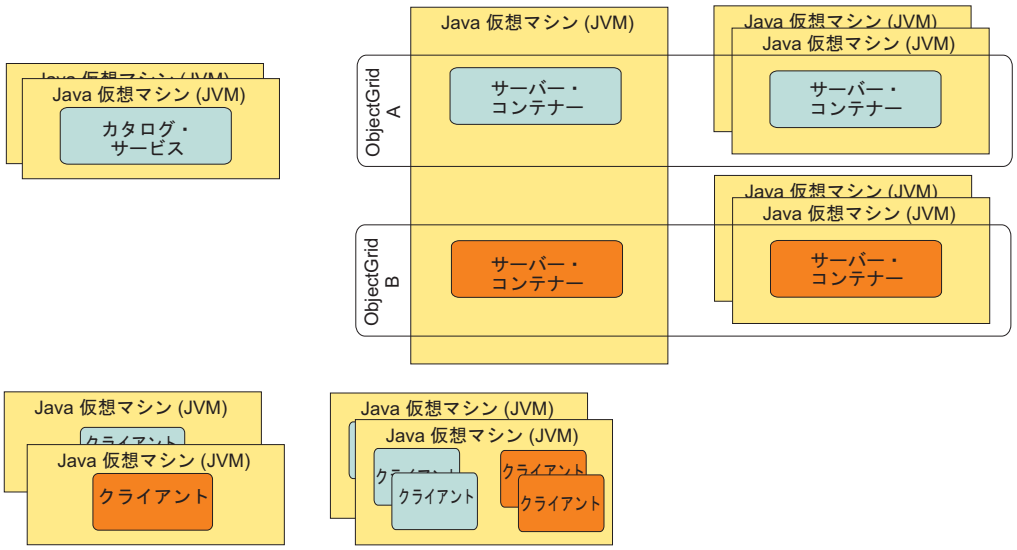


図 8. 可能なトポロジー

カタログ・サービス (カタログ・サーバー)

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。

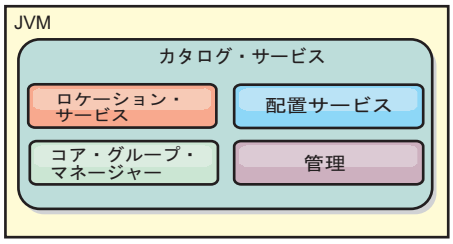


図 9. カタログ・サービス

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ます。 N 個の中の 1 つのポリシーが使用されるため、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・ドメインを構成します。カタログ・サービス・ドメインは、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。

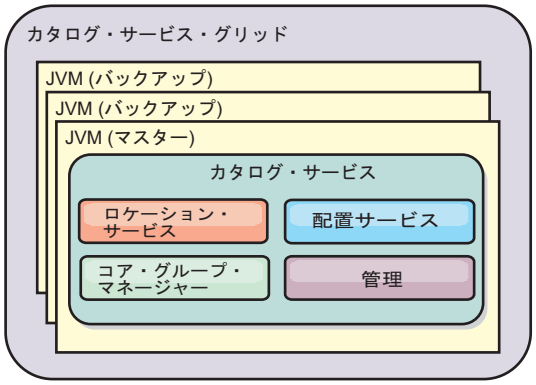


図 10. カタログ・サービス・ドメイン

キャッシング・トポロジー: メモリー内のキャッシングおよび分散キャッシング

WebSphere eXtreme Scale を使用して、アーキテクチャーはローカルのメモリー内でのデータ・キャッシング、または分散クライアント/サーバーでのデータ・キャッシングを使用できます。

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。

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 (非分散型の) 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。 ローカルのケースは、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。

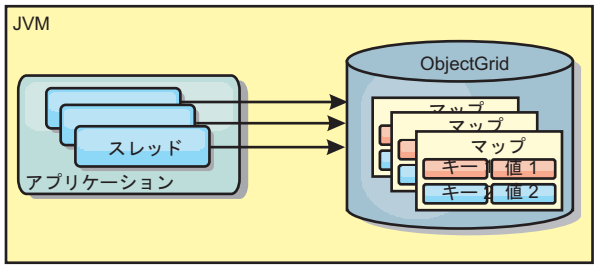


図 11. ローカルのメモリー内のキャッシュ・シナリオ

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 キャッシュでは、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、確実にキャッシュが同期されるようにする必要があります。 そのために、JMS を使用するピア複製キャッシュを使用可能にしてください。

WebSphere eXtreme Scale には、ピア ObjectGrid 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。

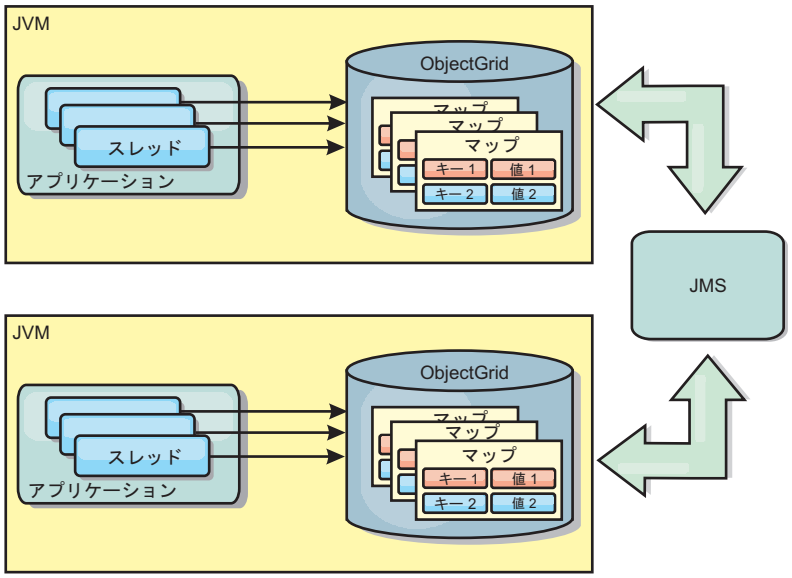


図 12. JMS によって変更が伝搬されるピア複製キャッシュ

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。

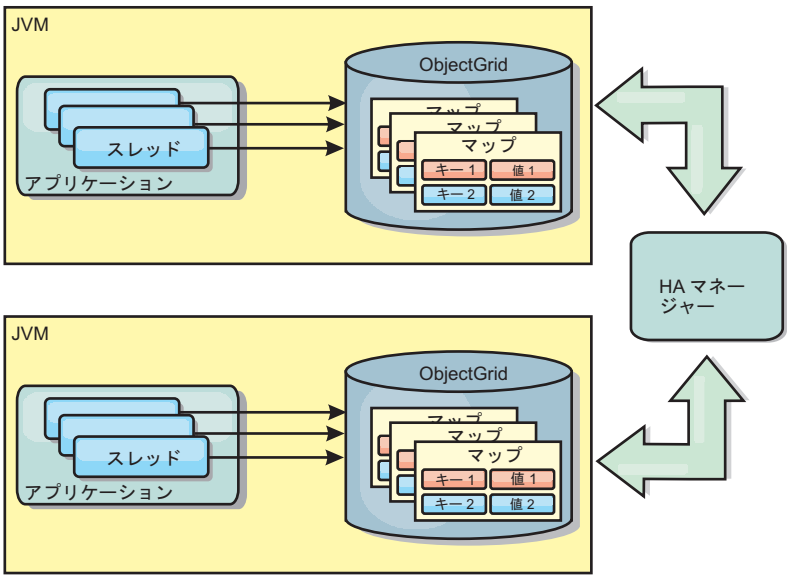


図 13. HA マネージャーによって変更が伝搬されるピア複製キャッシュ

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。共用キャッシュにより、データベースを構成する必要がなくなります。

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。

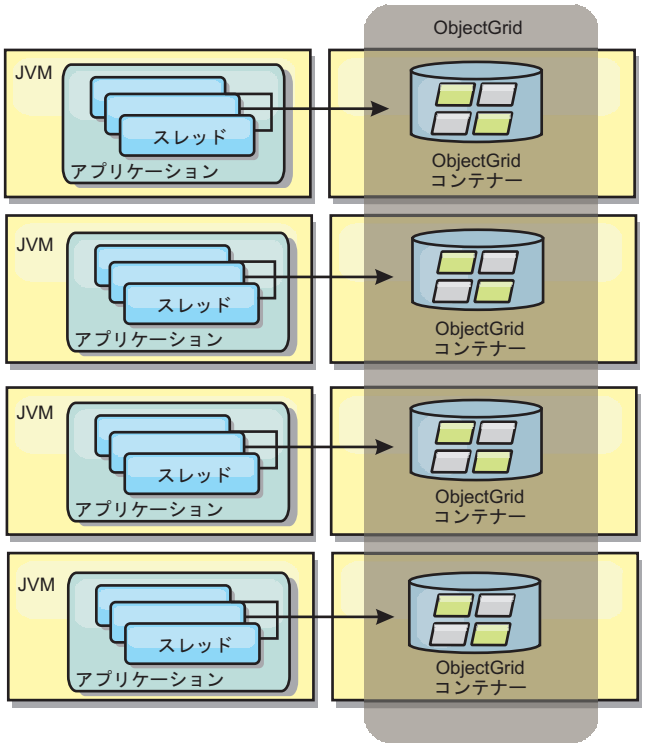


図 14. 分散キャッシュ

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。

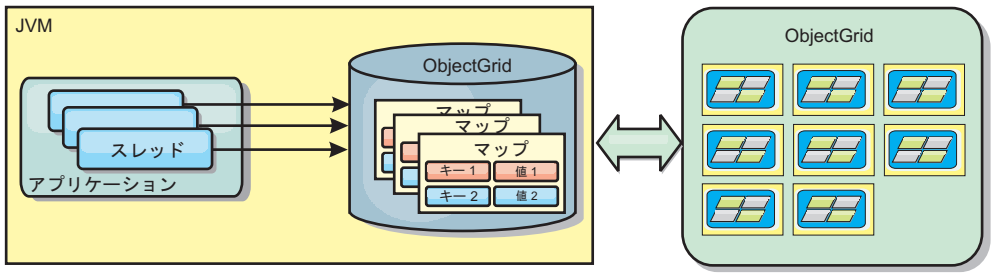


図 15. ニア・キャッシュ

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に関するトピック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。

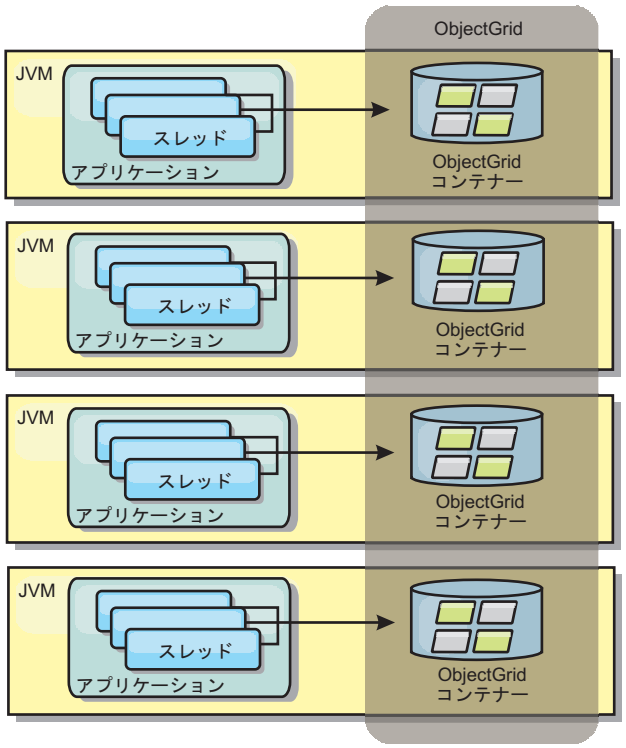


図 16. 組み込みキャッシュ

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・ローダーを使用しているため、アプリケーションのデプロイメントが簡単になります。
- 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。

マルチ・マスター・グリッド複製トポロジー (AP)

マルチ・マスター非同期複製機能を使用すると、2 つ以上のグリッドを、他のグリッドの正確なミラーにすることができます。 このミラーリングは、非同期複製を使用し、グリッドをまとめて接続するリンク間で実行されます。 各グリッドは完全に独立した「ドメイン」内でホストされ、独自のカタログ・サービス、コンテナー・サーバー、および固有のドメイン・ネームを所有します。 マルチ・マスター非同期複製機能では、これらのドメインの集合を相互接続するリンクを使用し、リンク上の複製を使用してドメインを同期させることができます。eXtreme Scale では、ドメイン間のリンクの定義はユーザーに任されているため、ほとんどのトポロジーを構成できます。

ドメイン: 固有の特性を持つグリッド

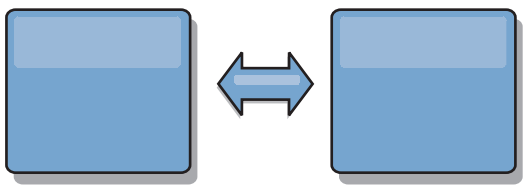
マルチ・マスター複製トポロジーで使用されるグリッドは、ドメインと呼ばれます。各ドメインは、以下の特性を持つ必要があります。

- 固有のドメイン・ネームを持つ専用カタログ・サービスがある
- ドメイン内の他のグリッドと同じグリッド名である
- ドメイン内の他のグリッドと同じ数の区画がある
- FIXED_PARTITION グリッドである (PER_CONTAINER グリッドは複製不可)
-
- ドメイン内の他のグリッドと同じデータ・タイプが複製される
- ドメイン内の他のグリッドと同じマップ・セット名、マップ名、および動的マップ・テンプレートがある

トポロジーのドメインが開始された後、上記の特性を持つ任意のグリッドが複製されます。 グリッドの複製ポリシーは無視されることに注意してください。

ドメインを接続するリンク

複製グリッドのインフラストラクチャーは、ドメイン間を双方向のリンクで接続したドメインのグラフです。 リンクによって、2 つのドメインはデータ変更を交換することができます。例えば、最も単純なトポロジーはドメイン間に単一のリンクを持つ 1 対のドメインです。 ドメインの名前は、左から「A」で始まって次は「B」というように付けられます。 リンクは、遠距離にわたる広域ネットワーク (WAN) を経由する場合もあります。 リンクが中断した場合は、いずれのドメインでもデータに対する変更を行うことができます。 その変更は後で、リンクがドメインを再接続すると調整されます。 ネットワーク接続が中断されると、リンクは自動的に再接続しようとします。

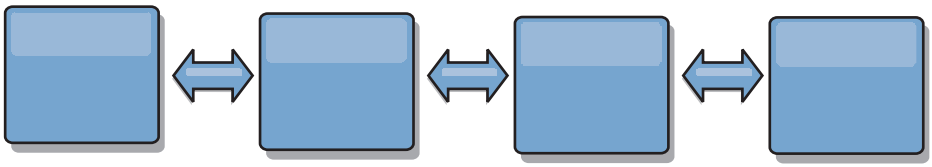


リンクのセットアップ後、eXtreme Scale は、まずすべてのドメインを同一にしようとします。そして、任意のドメインで変更が行われると同一状態を維持しようとします。 eXtreme Scale の目標は、各ドメインがリンクで接続されたすべての他のド

メインの正確なミラーになることです。 ドメイン間の複製リンクは、1 つのドメインで行われたすべての変更を確実に他のドメインにコピーするのに役立ちます。

ライン・トポロジー

ライン・トポロジーは最も単純なトポロジーの 1 つですが、かなりのリンク品質を実証します。 まず、変更を受信するために、ドメインは直接すべての他のドメインに接続する必要はありません。 ドメイン B はドメイン A から変更をプルします。ドメイン C は、ドメイン A と C を接続するドメイン B を介してドメイン A から変更を受信します。同様に、ドメイン D はドメイン C を介して他のドメインから変更を受信します。この機能によって、変更のソースから変更を配布する負荷が分散されます。



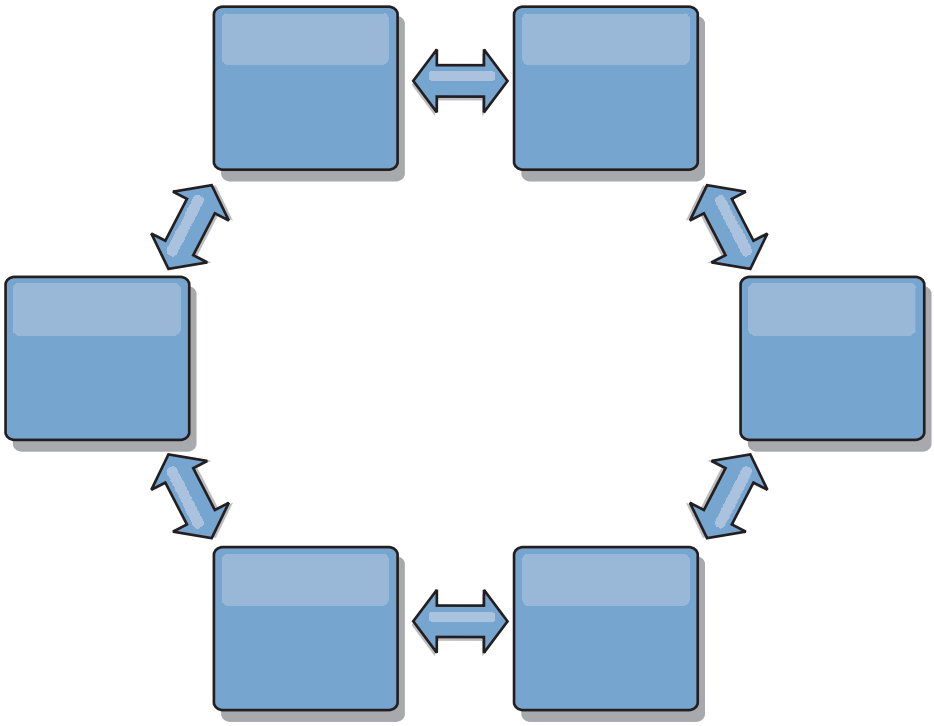
ドメイン C に障害が起こった場合、以下のイベントの発生が考えられることに注意してください。

1. ドメイン D は、ドメイン C が再開されるまで孤立します。
2. ドメイン C は、ドメイン A のコピーであるドメイン B と自分自身を同期させます。
3. ドメイン D は、ドメイン D が孤立していた間 (ドメイン C がダウンしていた間) にドメイン A と B で発生した変更を、ドメイン C を使って自分自身と同期させます。

最後に、ドメイン A、B、C、および D はすべて、他のドメインと再び同一になります。

リング・トポロジー

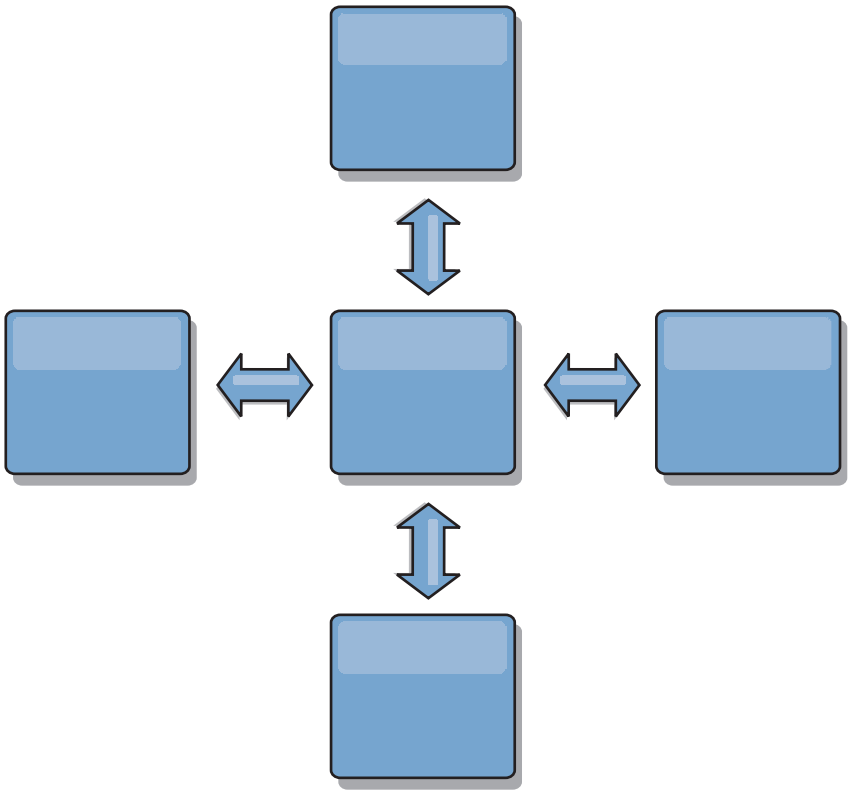
リング・トポロジーは、より回復力のあるトポロジーの例です。 1 つのドメインまたは単一リンクに障害が起こっても、残存しているドメインは、障害を避けてリング内を伝わる変更をそのまま取得できます。 各ドメインには、他のドメインへつながる 2 つのリンクがあります。 リング・トポロジーの大きさには関係なく、各ドメインは最大 2 つのリンクを持ちます。すべてのドメインがすべての変更を見るまで、特定のドメインからの変更をいくつものドメインの間で伝えなければならない場合があるため、変更を伝搬する待ち時間は長くなる可能性があります。 ライン・トポロジーにも同じ問題があります。



リングの中心に置いたルート・ドメインを使った、より洗練されたリング・トポロジーを想像してください。 ルート・ドメインは中央クリアリングハウスとして機能し、他のドメインはルート・ドメインで発生する変更に対してリモート・クリアリングハウスとして機能します。 ルート・ドメインはドメイン間の変更をアービトレーションすることができます。ルート・ドメインを囲む複数のリングがリング・トポロジーに含まれている場合、ルート・ドメインは最も内側にあるリング内のドメイン間の変更のみをアービトレーションすることができます。ただし、アービトレーションの結果は他のリングのドメインにも広がります。

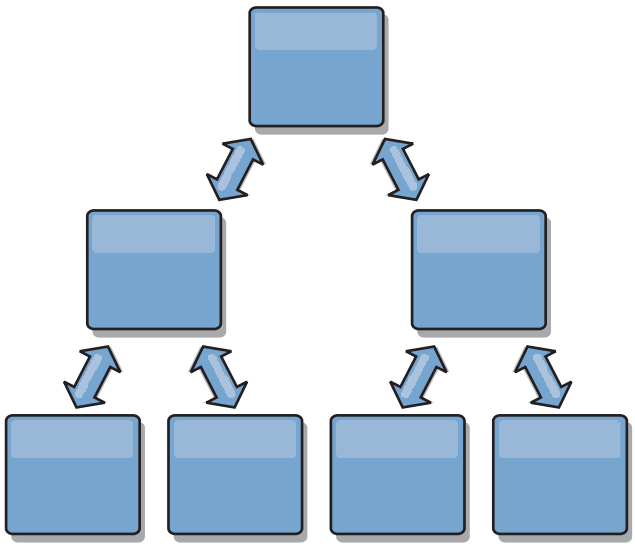
ハブ・アンド・スポーク・トポロジー

ハブ・アンド・スポーク・トポロジーでは、待ち時間が改善されます。したがって、変更は最大 1 つの中間ドメイン (ハブ) に伝わりますが、他の問題が出てきます。 このトポロジーにはハブとして機能する中央ドメインがあります。 この「ハブ・ドメイン」は、リンクを使用してすべての「スポーク・ドメイン」に接続されます。 明らかに、ドメイン間に変更を配布する負担はハブにかかります。 ハブは衝突のクリアリングハウスとして機能し、あるシナリオではセットアップが重要になってきます。 更新速度の速い環境では、ハブは、それについて行けるようにスポークよりも多くのハードウェア上で稼働する必要があります。 eXtreme Scale は、直線的に拡大するように設計されています。つまり、問題なく、必要に応じてハブをさらに大きくすることができます。 ただし、ハブに障害が起こった場合は、変更はハブが再始動するまで配布されません。 スポーク・ドメイン上の変更は、ハブが再接続された後に配布されます。



ツリー・トポロジー

最後のもう 1 つのトポロジーの例は、非循環有向ツリーです。 非循環とは、循環やループがないという意味です。有向とは、リンクが親と子の間にのみ存在するという意味です。この構成は、すべての可能なスポークに中央ハブを接続することが実用的ではないほどドメインをたくさん持つトポロジーの場合、あるいは、ルート・ドメインを更新することなく子ドメインを追加する機能を必要となるトポロジーの場合に役立ちます。



このトポロジーもルート・ドメインに中央クリアリングハウスを持つことができますが、第 2 レベルは、自分より下のドメインで発生する変更に対してリモート・ク

リアリングハウスとして機能することができます。 ルート・ドメインは、第 2 レベルにあるドメイン間の変更のみをアービトレーションすることができます。N 進ツリーも可能です。 N 進ツリーには 各レベルに N 個の子があります。 各ドメインは、N 個ずつ展開します。

トポロジー設計におけるアービトレーションの考慮事項

同じレコードが 2 個所で同時に変更される可能性がある場合には、変更の競合が生じることがあります。各ドメインが、同程度の CPU、メモリー、ネットワーク・リソースを持つようにセットアップしてください。変更の衝突処理 (アービトレーション) を実行しているドメインは、他のドメインよりも多くのリソースを使用することに気付くことがあります。 衝突は、自動的に検出されます。 衝突は、以下の 2 つのうちの 1 つのメカニズムを使って解決されます。

- **デフォルトの衝突アービター。** デフォルトのプロトコルは、字句的に最も小さい名前の付いたドメインからの変更を使用します。 例えば、ドメイン A と B によってレコードの競合が生じる場合には、ドメイン B の変更は無視されます。ドメイン A はそのバージョンを保持し、ドメイン B のレコードはドメイン A からのレコードに一致するように変更されます。これは、ユーザーやセッションが正常にバインドされているアプリケーション、またはユーザーやセッションがグリッドの 1 つにアフィニティーを持つ対象となるアプリケーションにも同様に適用されます。
- **カスタムの衝突アービター。** アプリケーションはカスタム・アービターを提供することができます。ドメインは、衝突を検出するとアービターを呼び出します。「優れた」カスタム・アービターの作成について詳しくは、マルチ・マスター複製用カスタム・アービターの作成を参照してください。

衝突が起こる可能性のあるトポロジーに対しては、ハブ・アンド・スポーク・トポロジーまたはツリー・トポロジーの使用を検討してください。 これらの 2 つのトポロジーは、以下のような場合に発生する可能性のある、エンドレスな衝突の回避につながります。

1. 複数のドメインで衝突が発生します。
2. 各ドメインが衝突をローカルで解決し、改訂を生成します。
3. 改訂が衝突し、その結果、改訂の改訂をもたらします。
4. このように、同期を取ろうとするさまざまなドメイン間に改訂が伝搬していきます。

エンドレスな衝突を回避するには、ドメインのサブセット用衝突ハンドラーとして特定のドメイン - *アービトレーション・ドメイン* – を選択してください。 例えば、ハブ・アンド・スポーク・トポロジーはハブを衝突ハンドラーとして使用する場合があります。 スポーク衝突ハンドラーは、スポーク・ドメインで検出された衝突を無視します。 ハブ・ドメインは改訂を作成し、制御できない衝突改訂を防ぎます。 衝突を処理するように割り当てられたドメインは、衝突の解決に責任を持つすべてのドメインにリンクしていなければなりません。 ツリー・トポロジーでは、内部の親ドメインが自分の直接の子の衝突を解決します。 対照的に、リング・トポロジーを使用すると、リング内の 1 つのドメインをリゾルバーとして指定することはできません。

次の表に、さまざまなトポロジーと互換性のあるアービトレーション・アプローチをまとめました。

表 3. アービトレーション・アプローチ： この表は、アプリケーション・アービトレーションがさまざまなトポロジーと互換性があるかどうかについて記述します。

トポロジー	アプリケーション・アービトレーションとの互換性	注
2 つのドメインのライン	あり	1 つのドメインをアービターとして選択します。
3 つのドメインのライン	あり	真ん中のドメインがアービターでなければなりません。 真ん中のドメインが、単純なハブ・アンド・スポーク・トポロジーのハブだと考えてください。
3 つより多いドメインのライン	なし	アプリケーション・アービトレーションはサポートされません。
N 個のスポークを持つハブ	あり	すべてのスポークへのリンクを持つハブがアービトレーション・ドメインでなければなりません。
N 個のドメインのリング	なし	アプリケーション・アービトレーションはサポートされません。
非循環有向ツリー (N 進ツリー)	あり	すべてのルート・ノードは、自分の直接の子孫のみをアービトレーションする必要があります。

トポロジー設計におけるリンクの考慮事項

変更待ち時間、フォールト・トレランス、およびパフォーマンス特性におけるトレードオフを最適化している間、トポロジーにはリンクの最小数が含まれているのが理想的です。

- **変更待ち時間**

  変更待ち時間は、変更が特定のドメインに到着する前に経由しなければならない中間ドメインの数によって決まります。

  トポロジーが、すべてのドメインを他のすべてのドメインにリンクすることによって中間ドメインを除去すれば、トポロジーの変更待ち時間は最善になります。ただし、ドメインはそのリンク数に比例して複製作業を実行しなければなりません。 大規模トポロジーの場合、非常に多くのリンクが定義され、管理が負担になることが考えられます。

  変更が他のドメインにコピーされる速度は、以下の追加要因によって異なります。

  – ソース・ドメイン上の CPU とネットワーク帯域幅
  – ソース・ドメインとターゲット・ドメインの間の中間ドメイン数とリンク数
  – ソース・ドメイン、ターゲット・ドメイン、および中間ドメインで使用可能な CPU とネットワーク・リソース
- **フォールト・トレランス**

フォールト・トレランスは、変更の複製のために、2 つのドメイン間に存在するパス数によって決定します。

ドメイン間に単一リンクしか存在しない場合、リンクに障害が起こると変更は伝搬されません。 1 つのドメインから別のドメインへの単一リンクが中間ドメインを経由する場合、いずれかの中間ドメインがダウンすると変更は伝搬されません。

3 つのドメイン A、B、および C を持つライン・トポロジーを考えてみます。

```
A <-> B <-> C
```

以下のいくつかの状態のままであれば、ドメイン C は A からの変更はまったく見えません。

– ドメイン A が稼働中でドメイン B がダウン

– A と B の間のリンクがダウン

– B と C の間のリンクがダウン

対照的に、リング・トポロジーでは、各ドメインはいずれかの方向から変更をプルすることができます。

```
A <-> B <-> C <-> A に戻る
```

例えば、ドメイン B がダウンしている場合、ドメイン C は引き続き変更を直接ドメイン A からプルできます。

ハブ・アンド・スポークの設計は、すべての変更がハブを介してプッシュされるので、ダウンしているハブの影響を受けやすくなります。 しかし、単一ドメインは、WAN や物理データ・センターの問題などの、コースに障害が起こる可能性のある完全なフォールト・トレラントなグリッドのままだと覚えていることは価値があります。

- **パフォーマンス**

ドメイン上に定義されるリンク数は、パフォーマンスに影響します。リンクが多いと使われるリソースも多くなり、結果的に複製のパフォーマンスが落ちる場合もあります。 他のドメインを介してドメイン A の変更をプルする機能は、そのトランザクションをどこにでも複製するドメイン A の負荷を効果的に軽減します。 *ドメイン上の変更配布の負荷は、ドメインが使用するリンクの数に制限されます。 トポロジー内にあるドメイン数には関係ありません。* このプロパティーによって、スケーラビリティーが提供され、変更配布の負荷は、単一ドメインに負荷をかけるのではなく、トポロジー内の複数のドメインによって共有することができます。

1 つのドメインが他のドメインを介して間接的に変更をプルできます。 5 つのドメインを持つライン・トポロジーを考えてみます。

```
A <=> B <=> C <=> D <=> E
```

– A は、B、C、D、および E から B を介して変更をプルします。

– B は、A と C からは直接、D と E からは C を介して変更をプルします。

– C は、B と D からは直接、A からは B を介して、E からは D を介して変更をプルします。

– D は、C と E からは直接、A と B からは C を介して変更をプルします。

– E は、D からは直接、A、B、および C からは D を介して変更をプルします。

ドメイン A および E は、それぞれ単一ドメインへのリンクのみを持っているので、配布の負荷は最も低くなります。 ドメイン B、C、および D はそれぞれ 2 つのドメインへのリンクを持っているので、ドメイン B、C、および D 上の配布の負荷は、ドメイン A および E 上の負荷の 2 倍になります。負荷は、トポロジー内の全体のドメイン数ではなく、各ドメインのリンク数によって異なるため、この負荷の分散は、ラインに 1000 ドメインを含んだとしても一定のままです。

パフォーマンスの考慮事項

マルチ・マスター複製トポロジーを使う際は、以下の制限を考慮してください。

- **変更配布の調整** (前述のとおり)
- **複製の待ち時間** (前述のとおり)
- **複製リンクのパフォーマンス** eXtreme Scale は、任意の一対の JVM 間で、単一の TCP/IP ソケットを作成します。それらの JVM 間のトラフィックはすべてそのソケット上で発生し、マルチ・マスター複製も含まれます。 ドメインは少なくとも N 個のコンテナー JVM でホストされ、少なくとも N 個の TCP リンクをピア・ドメインに提供しているため、コンテナー数をより多く持つドメインには、より高い複製のパフォーマンス・レベルがあります。 より多くのコンテナーとは、より多くの CPU とネットワーク・リソースを意味します。
- **TCP スライディング・ウィンドウのチューニングおよび RFC 1323** リンクの両端の RFC 1323 サポートを使用可能にすると、より多くのデータが往復でき、この結果、より高いスループットが実現されます。 この技法は、約 16,000 の要因でウィンドウの容量を拡張します。

  TCP ソケットが、スライディング・ウィンドウのメカニズムを使用して大量データのフローを制御することを思い出してください。これは通常、往復のインターバルのソケットを 64 KB に制限します。 往復のインターバルが 100 ミリ秒の場合、追加チューニングをすることなく帯域幅は 640 KB/秒に制限されます。 リンクで使用可能な帯域幅を完全に使用する場合は、オペレーティング・システムに固有のチューニングが必要になることがあります。 ほとんどのオペレーティング・システムにはチューニング・パラメーターがあり、高度な待ち時間リンクのスループットを向上させる RFC 1323 オプションも含まれます。

  以下の複数の要因が複製のパフォーマンスに影響する可能性があります。

  – eXtreme Scale が変更をプルする速度。
  – eXtreme Scale がプル複製要求をサービスする速度。
  – スライディング・ウィンドウの容量。
  – リンクの両端のネットワーク・バッファーをチューニングすると、eXtreme Scaleは、可能な限り速くソケット上の変更をプルできます。
- **オブジェクト・シリアライゼーション** すべてのデータはシリアライズ可能でなければなりません。 ドメインが COPY_TO_BYTES を使用していない場合、そのドメインは Java のシリアライゼーションまたは ObjectTransformers を使用してシリアライゼーション・パフォーマンスを最適化する必要があります。

- **圧縮** eXtreme Scale は、デフォルトでドメイン間で送信されるすべてのデータを圧縮します。 現行リリースにおいて、圧縮を無効にするオプションはありません。
- **メモリー・チューニング** *マルチ・マスター複製トポロジーのメモリー使用量は、トポロジー内のドメイン数とはほとんど関係ありません。*

  マルチ・マスター複製を使用可能にすると、バージョン管理を扱うマップ・エントリーごとに一定のオーバーヘッドが追加されます。 各コンテナーはトポロジー内の各ドメインの一定量のデータも追跡します。 2 つのドメインを持つトポロジーは、50 ドメインを持つトポロジーとほぼ同じメモリーを使用します。 eXtreme Scale は、その実装環境のリプレイ・ログや類似のキューを使用しません。すなわち、複製リンクがかなりの期間使用できない場合、データ構造のサイズは増大せず、リンクが再始動するときに複製が再開されるのを待ちます。

FIXED_PARTITION の複数データ・センター

現在、複数のデータ・センター間で FIXED_PARTITION グリッドを使用できます。各データ・センターには、マルチ・マスター複製用語で、独自のドメインが必要です。 各データ・センターは、ローカル・ドメインからのデータの読み取り、およびローカル・ドメインへのデータの書き込みができます。 これらの変更は、定義したリンクを使って他のデータ・センターに伝搬されます。

完全複製クライアント

このトポロジー変化には、ハブとして稼働する 1 対の eXtreme Scale サーバーが含まれます。 各クライアントは、クライアント JVM のカタログを使って、必要なものを完備した単一コンテナー・グリッドを作成します。 クライアントは、そのグリッドを使用してハブ・カタログに接続します。これにより、クライアントはハブへの接続を取得すると、すぐにハブと同期するようになります。

クライアントによって行われた変更は、クライアントに対してローカルで、非同期でハブに複製されます。 ハブはアービトレーション・ドメインとして機能し、すべての接続されたクライアントに変更を配布します。完全複製クライアントのトポロジーは、OpenJPA などのオブジェクト・リレーショナル・マッパーに適した L2 キャッシュを提供します。変更はハブを介してクライアント JVM 間に迅速に配布されます。 キャッシュ・サイズをクライアントの使用可能なヒープ・スペース内に含むことができる限り、このトポロジーは L2 のこのスタイルに適したアーキテクチャーです。

必要であれば、複数の区画を使用して、複数の JVM 上にハブ・ドメインを拡張します。 すべてのデータはまだ単一のクライアント JVM に収まらなければならないため、複数の区画を使用してハブの容量を増加させ、変更の配布とアービトレーションを行いますが、単一ドメインの容量は変更しません。

制限

マルチ・マスター複製トポロジーを使うかどうか、およびその使用方法について決定する際は、以下の制限を考慮してください。

- **複数ドメインを使ったクラス・ローダーの構成には気をつけてください**

ドメインは、キーおよび値として使用されるクラスすべてへのアクセス権限を持たなければなりません。すべての依存関係は、すべてのドメインのグリッド・コンテナー JVM に対するすべてのクラスパスに反映されなければなりません。CollisionArbiter プラグインがキャッシュ・エントリーの値を取得する場合、その値に対するクラスはアービターを呼び出すドメインに存在しなければなりません。

- **ローダーの使用はお勧めしません**

  ローダーを使用して、グリッドとデータベースの間の変更をインターフェースで連結することができます。 トポロジー内のすべてのグリッド (ドメイン) が、地理的に同じデータベースに連結されているということは考えられません。 WAN の待ち時間および他の要因で、このユース・ケースが望ましくない状態になる場合があります。

  グリッドのプリロードは、設計に慎重を要する別の問題です。 通常、グリッドは再始動すると、もう一度プリロードされます。 マルチ・マスター複製の使用時は、プリロードは必要ない、または望ましくさえありません。 ドメインは、オンラインになると、すぐに自動的に自分がリンクされているドメインの内容とともに自分自身を再ロードします。 結果的に、マルチ・マスター複製トポロジー内のドメインであるグリッドの手動プリロードを開始する必要はありません。

  ローダーは、通常、挿入規則および更新規則に従います。 マルチ・マスター複製を使用すると、挿入はマージとして扱う必要があります。 ドメインの再始動後にリモート側でデータがプルされている場合、既存データはローカル・ドメインに「挿入」されます。 このデータは既にローカル・データベースにあると考えられるため、標準的な挿入はデータベースの重複キー例外で失敗します。 代わりに、マージ・セマンティクスを使用してください。

  eXtreme Scale を構成し、Loader プラグインでプリロード・メソッドを使用して断片ベースのプリロードを行うことができます。この技法をマルチ・マスター複製トポロジーで使用しないでください。代わりに、トポロジーを始動するとき (最初に) は、クライアント・ベースのプリロードを使用します。トポロジー内の他のドメインに保管されたものの現行コピーを使用して、マルチ・マスター・トポロジーが再始動したドメインをリフレッシュできるようにします。ドメインが開始した後、マルチ・マスター・トポロジーは責任を持ってそれらのドメインを同期させます。

- **EntityManager はサポートされません**

  エンティティー・マップを含むマップ・セットは、ドメインを介して複製されません。

- **バイト配列マップはサポートされません**

  COPY_TO_BYTES で構成されたマップを含むマップ・セットは、ドメインを介して複製されません。

- **後書きはサポートされません**

  後書きサポートで構成されたマップを含むマップ・セットは、ドメインを介して複製されません。

データベース統合: 後書き、インライン、およびサイド・キャッシング

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。

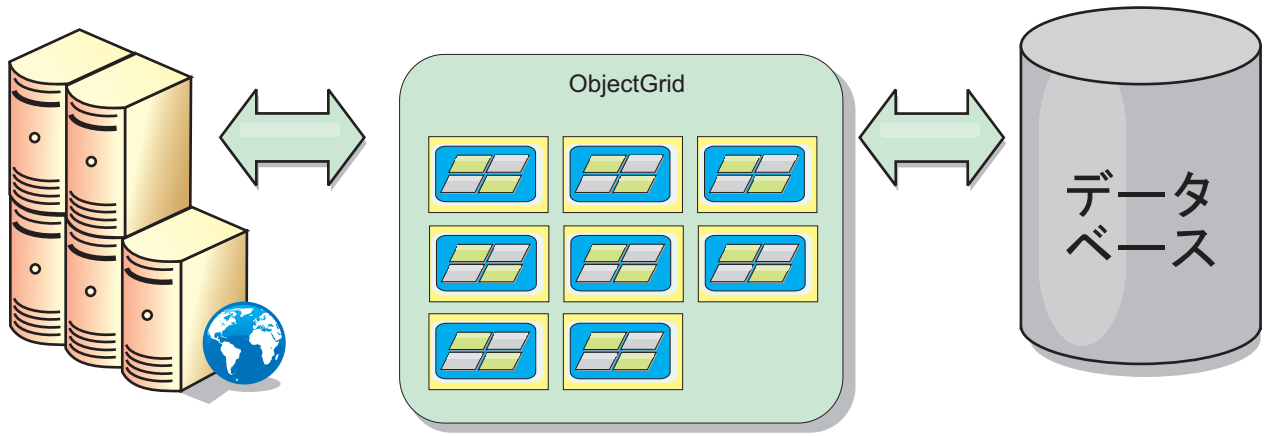


図 17. データベース・バッファーとしての ObjectGrid

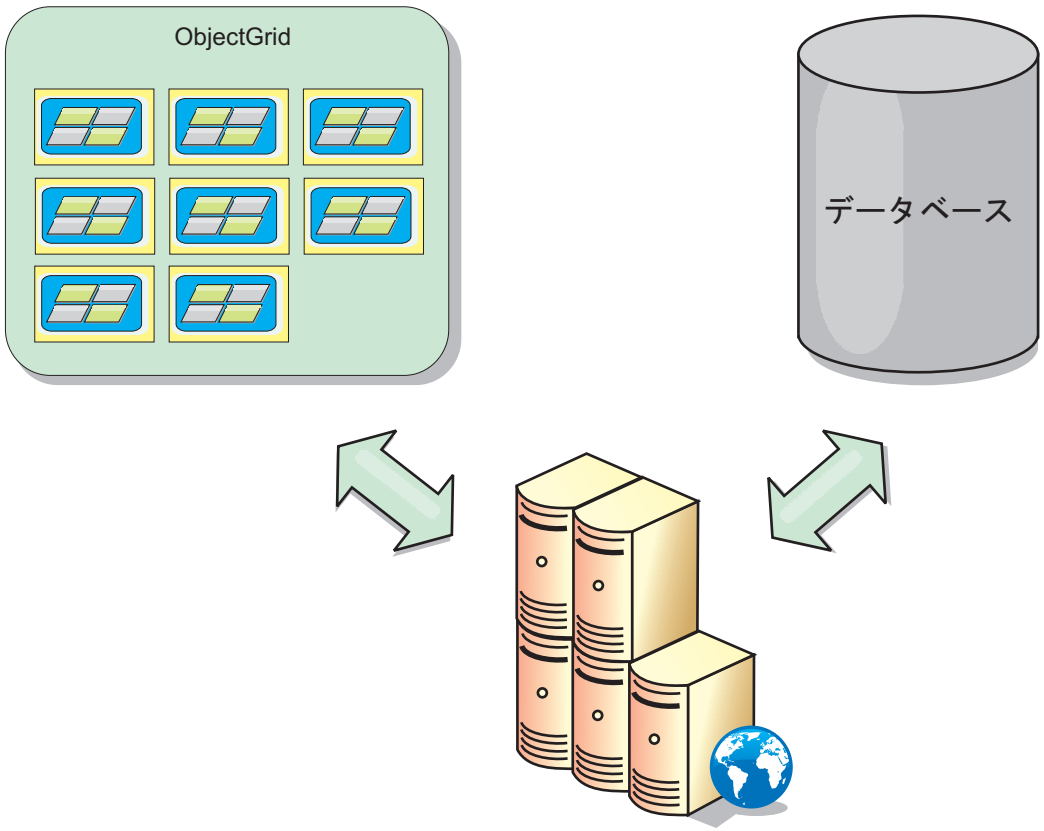


図 18. サイド・キャッシュとしての ObjectGrid

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

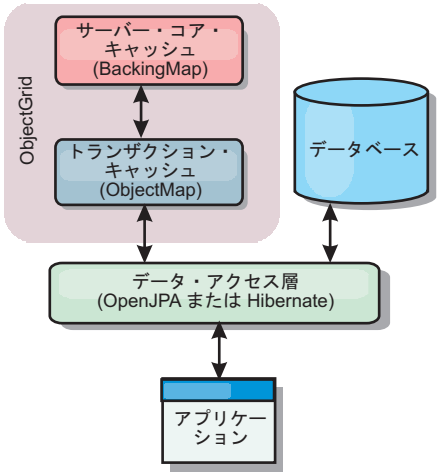


図 19. サイド・キャッシュ

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。

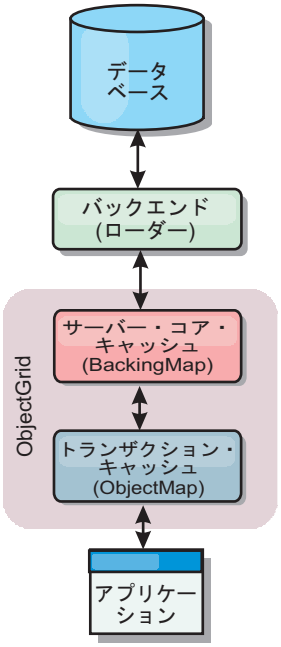


図 20. インライン・キャッシュ

インライン・キャッシング

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。

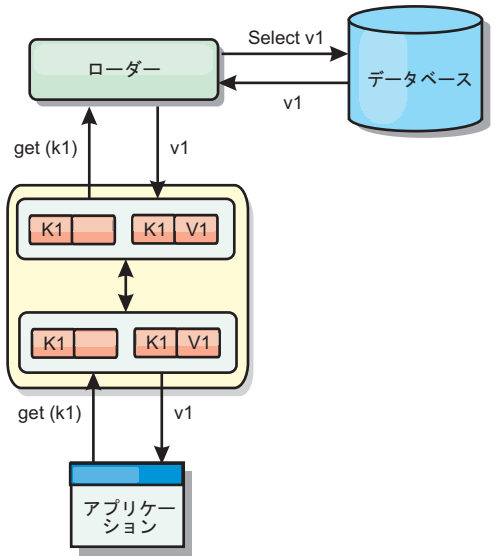


図 21. リードスルー・キャッシング

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。

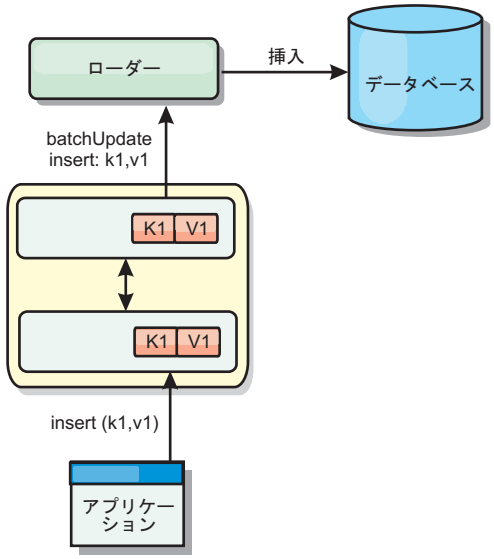


図 22. ライトスルー・キャッシング

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、『後書きキャッシング』を参照してください。

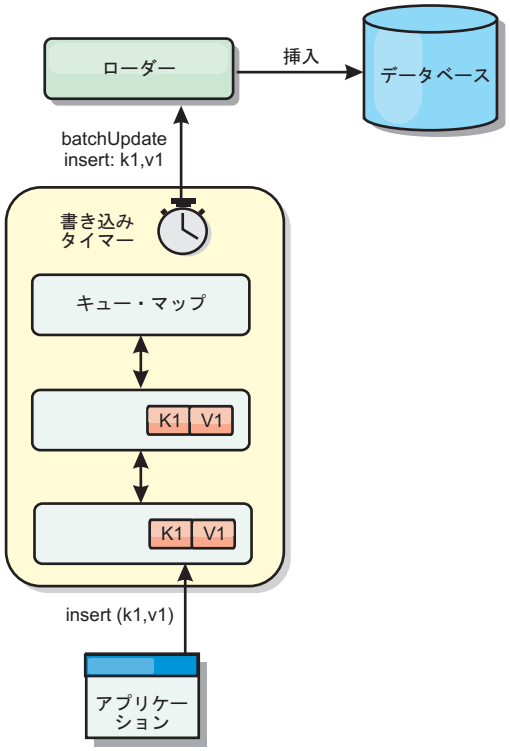


図 23. 後書きキャッシング

詳しくは、『後書きキャッシング』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、バックエンドとして使用している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)、またはエントリー・ベースの遅延 (例えば、1000 エントリー) 後に実行されます。

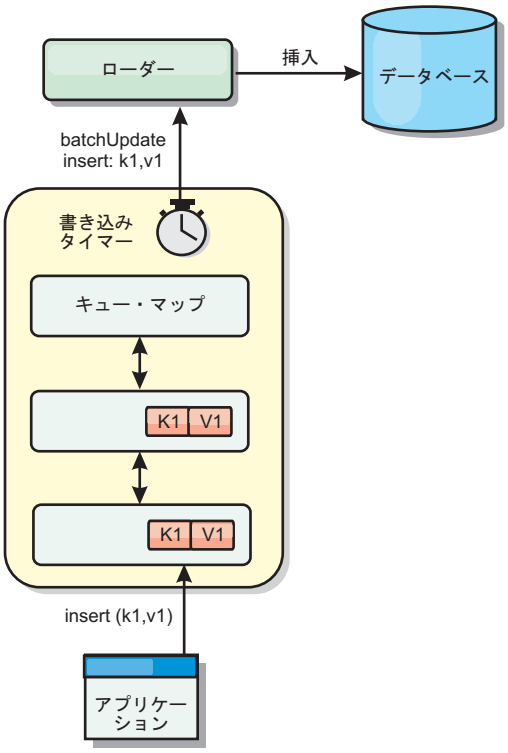


図 24. 後書きキャッシング

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キュー・マップから定期的に除去して、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と、以下のような利点があります。

- **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・パフォーマンスの改善: データがバックエンドと同期されるのをトランザクションが待機する必要がないので、個別の eXtreme Scale トランザクション時間が削減されます。

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ことは簡単ですが、後書きサポートを扱うアプリケーションを設計する際には、注意すべき考慮事項があります。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 更新は、1 つのバックエンド・トランザクションでバックエンドに対して更新され、マップ Map2 に対する key2 更新は、異なる後書きスレッドにより別のバックエンド・トランザクションでバックエンドに対して更新されます。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・マップのロックを取得する操作には 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 ロックが取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 ロックが取得されます。
- 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に、キュー・マップ内の同じキーに対して S ロックが取得されます。コミット時間の間、同じトランザクションを使用するキュー・マップ内のキーに対して X ロックが設定されます。

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。これは、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している場合に発生することがあります。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。

- LoaderException が後書きローダーによってキャッチされると、重複キー障害などのある種のデータ障害のため、後書きローダーはそれを障害とみなします。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。

- LoaderNotAvailableException が後書きローダーによってキャッチされると、データベース・エンドに接続できない (例えば、データベース・バックエンドがダウンしている、データベース接続が使用可能でない、ネットワークがダウンしているなど) ため、後書きローダーはそれを障害とみなします。後書きローダーは 15 秒待ってから、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。後書きローダーでキューに入れられたすべてのレコードは、失敗更新レコードとなります。このような場合、バックエンド障害分離の目的が果たせなくなります。

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・スループットも増加します。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。

予想される使用パターンおよび環境に基づいて、最大更新数または最大更新時間を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

ローダー

eXtreme Scale ローダー・プラグインによって、eXtreme Scale マップは、通常は同じシステムまたは別のシステム上の永続ストアに保管されるデータのメモリー・キャッシュの働きをすることができます。 通常、データベースまたはファイル・システムは永続ストアとして使用されます。リモート Java 仮想マシン (JVM) もデータのソースとして使用でき、eXtreme Scale を使用してハブ・ベースのキャッシュを構築できます。ローダーには、永続ストアとの間でデータの読み取りおよび書き込みを行うロジックが備わっています。

概説

ローダーは、変更がバックアップ・マップに対して行われた場合、または、バックアップ・マップがデータ要求を満足できない (キャッシュ・ミス) 場合に呼び出されるバックアップ・マップ・プラグインです。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。

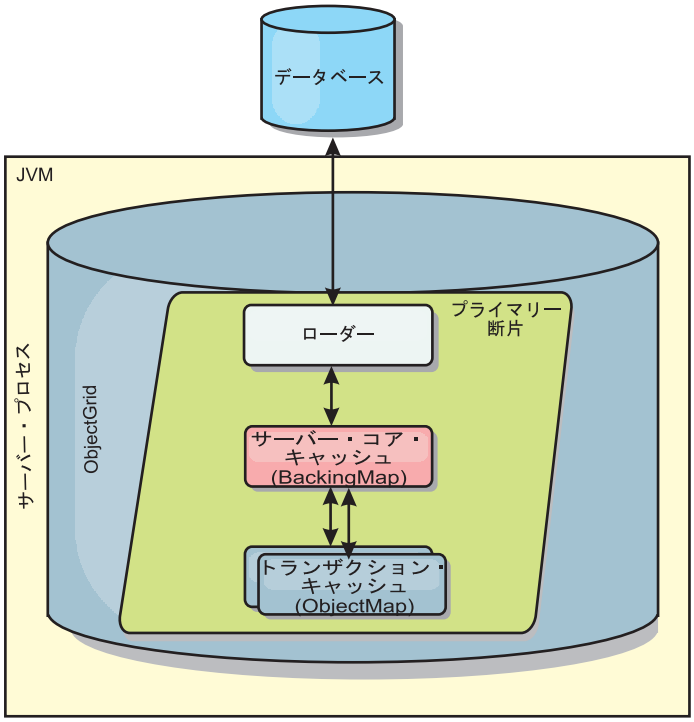


図 25. ローダー

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。

2 つの組み込みローダーにより、リレーショナル・データベース・バックエンドとの統合が非常に単純化されます。JPA ローダーは、Java Persistence API (JPA) 仕様の OpenJPA および Hibernate 実装の両方のオブジェクト関係マッピング (ORM) 機

能を使用します。詳しくは、「*製品概要*」で JPA ローダーに関する説明を参照してください。

ローダーの使用

ローダーを BackingMap 構成に追加するには、プログラマチック構成または XML 構成を使用します。ローダーには、バックアップ・マップとの間で以下のような関係があります。

- 1 つのバックアップ・マップは 1 つのローダーしか持てない。
- クライアント・バックアップ・マップ (ニア・キャッシュ) はローダーを持てない。
- 1 つのローダー定義を複数のバックアップ・マップに適用できるが、各バックアップ・マップは独自のローダー・インスタンスを持つ。

詳しくは、*製品概要*のローダーの作成に関するトピックを参照してください。

ローダーのプラグインの XML 構成アプローチ

アプリケーションが提供するローダーは、XML ファイルを使用して接続できます。以下の例は、「MyLoader」ローダーを「map1」バックアップ・マップに接続する方法を示しています。ローダーの className、データベース名と接続詳細、および分離レベル・プロパティーを指定する必要があります。完全なローダー・クラス名ではなく、プリローダー・クラス名を指定して、プリローダーだけを使用している場合、同じ XML 構造を使用できます。

XML を使用したローダー構成

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
    <objectGrid name="grid">
        <backingMap name="map1" pluginCollectionRef="map1" lockStrategy="OPTIMISTIC" />
    </objectGrid>
</objectGrids>
<backingMapPluginCollections>
    <backingMapPluginCollection id="map1">
        <bean id="Loader" className="com.myapplication.MyLoader">
            <property name="dataBaseName"
                      type="java.lang.String"
                      value="testdb"
                      description="database name" />
            <property name="isolationLevel"
                      type="java.lang.String"
                      value="read committed"
                      description="iso level" />
        </bean>
    </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

ローダーのプログラマチックなプラグイン

プログラマチックな構成は、ローカルなメモリー内グリッドでのみ使用できます。以下のコード・スニペットは、ObjectGrid API を使用してアプリケーションが提供するローダーを map1 のバックアップ・マップに接続する方法を示しています。

Programmatic configuration of a Loader

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
```

```
ObjectGrid og = ogManager.createObjectGrid( "grid" );
BackingMap bm = og.defineMap( "map1" );
MyLoader loader = new MyLoader();
loader.setDataBaseName("testdb");
loader.setIsolationLevel("read committed");
bm.setLoader( loader );
```

このスニペットでは、 MyLoader クラスは、com.ibm.websphere.objectgrid.plugins.Loader インターフェースを実装するアプリケーション提供のクラスであることが前提になります。 ObjectGrid の初期化後は、ローダーとバックアップ・マップとの関連付けを変更できないので、呼び出されている ObjectGrid インターフェースの initialize メソッドを起動する前にコードを実行する必要があります。初期化が起こった後に setLoader メソッドが呼び出された場合、IllegalStateException 例外が発生します。

アプリケーションが提供する Loader には、set プロパティーがあります。例では、MyLoader ローダーを使用して、リレーショナル・データベースの表からデータを読み書きします。ローダーにより、データベースの名前と SQL 分離レベルが指定されることが必要です。 MyLoader ローダーには、setDataBaseName メソッドと setIsolationLevel メソッドがあり、アプリケーションはこれらのメソッドを使用してこれら 2 つの Loader プロパティーを設定できます。

- ユーザー「bob」は eXtreme Scale ユーザーとして認証されています。アプリケーションは、「DB2Hibernate」というパーシスタンス・ユニット名を使用する「mygrid」という名前のグリッドにアクセスしています。 コンテナー・サーバーは「XS_Server1」です。結果情報は次のようになります。
  - **ユーザー**=bob
  - **ワークステーション名**=XS_Server1,192.168.1.101
  - **アプリケーション名**=mygrid,DB2Hibernate
  - **アカウント情報**=1, DEFAULT,FE7954BD-0126-4000-E000-2298094151DB,com.ibm.db2.jcc.t4.b@71787178
- ユーザー「bob」は WAS トークンを使用して認証されています。 アプリケーションは、「DB2OpenJPA」というパーシスタンス・ユニット名を使用する「mygrid」という名前のグリッドにアクセスしています。 コンテナー・サーバーは「XS_Server2」です。 結果情報は次のようになります。
  - **ユーザー**

    =acme.principal.UserPrincipal[Bob],acme.principal.GroupPrincipal[admin]
  - **ワークステーション名**=XS_Server2,192.168.1.102
  - **アプリケーション名**=mygrid,DB2OpenJPA
  - **アカウント情報**=188,DEFAULT,FE72BC63-0126-4000-E000-851C092A4E33,com.ibm.ws.rsadapter.jdbc.WSJccSQLJConnection@2b432b43

データベース・アクセスをモニターする方法については、DB2 Performance Expert を参照してください。

データのプリロードおよびウォームアップ

ローダーのユーザーを組み込む多くのシナリオで、グリッドをデータと一緒にプリロードして準備しておくことができます。

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。

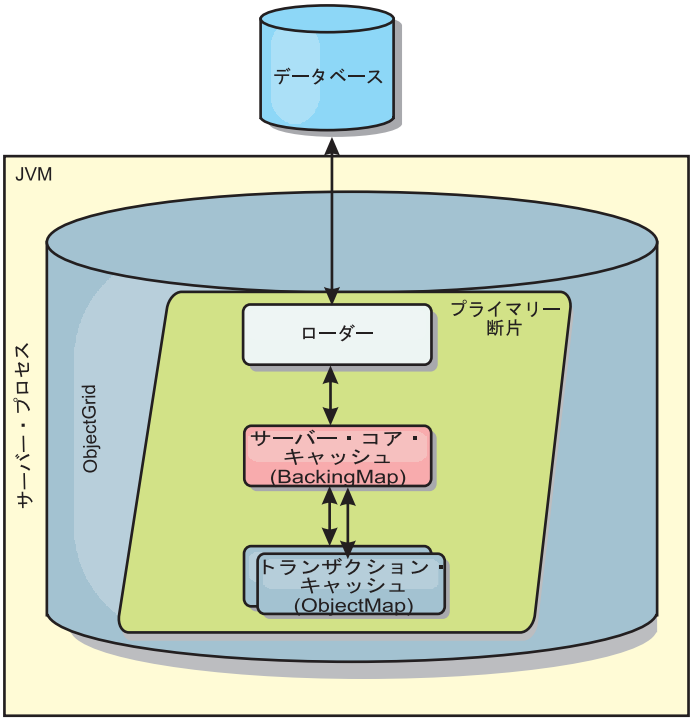


図 26. ローダー・プラグイン

詳しくは、「*プログラミング・ガイド*」でローダーの使用に関する説明を参照してください。

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。

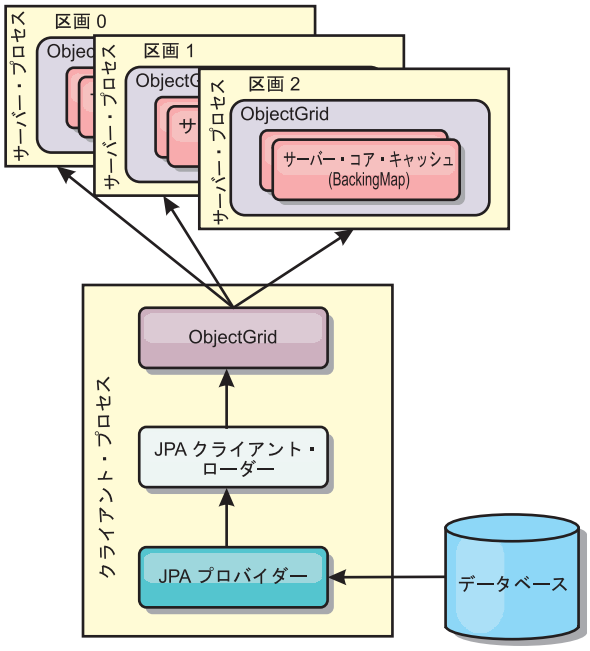


図 27. クライアント・ローダー

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。

複製された eXtreme Scale 用のローダーの実装例については、「*プログラミング・ガイド*」で、複製プリロード・コントローラーを使用するローダーの作成に関する説明を参照してください。

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 4. 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。
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 5. 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 6. 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

表 6. 同期コミット処理 (続き)

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**
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。

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。

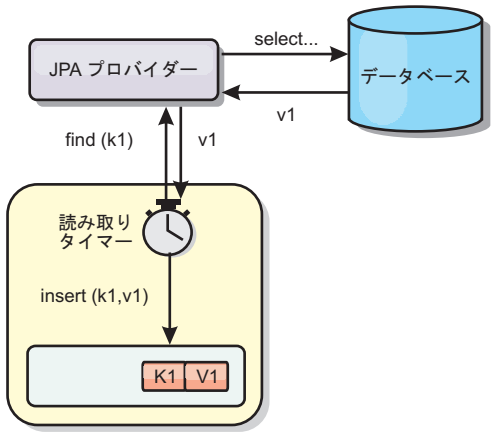


図 28. 定期的リフレッシュ

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。

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。

失効したキャッシュ・データの無効化

クライアントが失効データを表示する可能性のある時間枠を減らすには、イベント・ベースの無効化メカニズムまたはプログラマチックな無効化メカニズムが使用できます。

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・キャッシュが変更し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。 この通知タイプによって、クライアントが失効データを表示する時間を短縮します。

イベント・ベースの無効化は、一般的には以下の 3 つのコンポーネントで構成されます。

- **イベント・キュー:** イベント・キューには、データ変更イベントが保管されます。 データ変更イベントを管理できるのであれば、イベント・キューは JMS キュー、データベース、メモリー内の FIFO キュー、またはすべての種類のマニフェストの可能性があります。
- **イベント・パブリッシャー:** イベント・パブリッシャーは、データ変更イベントをイベント・キューにパブリッシュします。 イベント・パブリッシャーは、通常、作成されたアプリケーションまたは eXtreme Scale プラグインの実装です。イベント・パブリッシャーは、いつデータが変更されたかを知っています。あるいはイベント・パブリッシャーがデータ自体を変更します。トランザクションがコミットすると、変更されたデータに対してイベントが生成され、イベント・パブリッシャーはこれらのイベントをイベント・キューにパブリッシュします。
- **イベント・コンシューマー:** イベント・コンシューマーは、データ変更イベントをコンシュームします。 イベント・コンシューマーは、通常アプリケーションで、ターゲット・グリッド・データが他のグリッドからの最新の変更を使用して更新されることを確認します。 このイベント・コンシューマーは、イベント・キューと対話をして最新のデータ変更を取得し、ターゲット・グリッドのデータ変更を適用します。 イベント・コンシューマーは eXtreme Scale API を使用して、失効データを無効にしたり、グリッドを最新データで更新することができます。

例えば、JMSObjectGridEventListener にはクライアント/サーバー・モデルのオプションがあり、そのイベント・キューは指定された JMS 宛先です。 すべてのサーバー・プロセスがイベント・パブリッシャーです。 トランザクションがコミットすると、サーバーはデータ変更を取得し、それを指定された JMS 宛先にパブリッシュします。 すべてのクライアント・プロセスがイベント・コンシューマーです。 指定された JMS 宛先からデータ変更を受信し、その変更をクライアントのニア・キャッシュに適用します。

詳しくは、「*管理ガイド*」で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。

プログラマチックな無効化

WebSphere 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。

他の手法と一緒にプログラマチックな無効化を使用して、データをいつ無効にするかを決定します。 例えば、この無効化メソッドは、イベント・ベースの無効化メカニズムを使用してデータ変更イベントを受信し、API を使用して失効データを無効にします。

索引付け

MapIndexPlugin は、BackingMap 上にいくつかの索引を作成して、非キー・データ・アクセスをサポートするために使用します。

索引のタイプおよび構成

索引付けフィーチャーは、MapIndexPlugin と表されるか、または略して Index で表されます。Index は BackingMap プラグインです。BackingMap では、各索引プラグインが索引構成規則に従っている限り、複数の索引プラグインを構成できます。

索引付けフィーチャーは、1 つ以上の索引を BackingMap に作成する場合に使用できます。1 つの索引は、BackingMap 内の 1 つのオブジェクトの 1 つの属性または属性のリストから作成されます。このフィーチャーにより、アプリケーションはより迅速に特定のオブジェクトを見つけることができます。索引付けフィーチャーを使用すると、アプリケーションは特定の値を持つオブジェクトや、ある範囲の索引属性値内にあるオブジェクトを見つけることができます。

可能な索引付けには、静的および動的という 2 つのタイプがあります。静的索引付けの場合、ObjectGrid インスタンスを初期化する前に、BackingMap に索引プラグインを構成する必要があります。この構成を行うには、BackingMap を XML で構成するか、またはプログラマチックに構成します。静的索引付けでは、まず最初に、ObjectGrid の初期化中に索引を作成します。索引は常に BackingMap に同期しており、いつでも使用できる準備ができています。静的索引付けプロセスが既に開始している場合、索引は、eXtreme Scale トランザクション管理プロセスの一環として保守されます。トランザクションが変更をコミットすると、それらの変更は静的索引も更新し、トランザクションがロールバックされれば索引の変更もロールバックされます。

動的索引付けの場合は、索引を含む ObjectGrid インスタンスの初期化の前または後に、BackingMap に索引を作成することができます。動的索引付けプロセスのライフサイクルはアプリケーションによって制御されるので、不要になったら動的索引を

削除することができます。アプリケーションが動的索引を作成する場合は、索引作成プロセスを完了するまでに時間がかかるために、その索引をすぐに使用できないことがあります。この時間は索引付けされるデータの量に依存するので、特定の索引付けイベントが発生したときにそのことを通知してもらいたいアプリケーションのために、DynamicIndexCallback インターフェースが提供されています。これらのイベントには、準備完了、エラー、および破棄があります。アプリケーションは、このコールバック・インターフェースを実装し、動的索引付けプロセスに登録できます。

BackingMap に索引プラグインが構成されている場合、対応する ObjectMap からアプリケーション索引プロキシー・オブジェクトを取得することができます。ObjectMap の getIndex メソッドを呼び出し、索引プラグインの名前を渡すと、索引プロキシー・オブジェクトが戻されます。索引プロキシー・オブジェクトを適切なアプリケーション索引インターフェース (MapIndex、MapRangeIndex、またはカスタマイズされた索引インターフェースなど) にキャストする必要があります。索引プロキシー・オブジェクトを取得したら、アプリケーション索引インターフェースで定義されたメソッドを使用して、キャッシュされたオブジェクトを検出することができます。

次のリストに、索引付けの使用手順をまとめます。

- 静的または動的索引プラグインを BackingMap に追加します。
- ObjectMap の getIndex メソッドを発行して、アプリケーション索引プロキシー・オブジェクトを取得します。
- MapIndex、MapRangeIndex またはカスタマイズされた索引インターフェースなどの適切なアプリケーション索引インターフェースに、索引プロキシー・オブジェクトをキャストします。
- アプリケーション索引インターフェースで定義されたメソッドを使用して、キャッシュされたオブジェクトを検出します。

HashIndex クラスは、組み込みアプリケーション索引インターフェースである MapIndex と MapRangeIndex の両方をサポートすることのできる組み込み索引プラグイン実装です。ユーザー独自の索引を作成することもできます。HashIndex を静的索引または動的索引として BackingMap に追加して、MapIndex または MapRangeIndex の索引プロキシー・オブジェクトを取得し、その索引プロキシー・オブジェクトを使用してキャッシュ・オブジェクトを検索することができます。

HashIndex の構成について詳しくは、HashIndex の構成を参照してください。

ユーザー独自の索引プラグインの作成について詳しくは、「*プログラミング・ガイド*」内の索引プラグイン作成に関する説明を参照してください。

索引付けの使用方法について詳しくは、「*プログラミング・ガイド*」内の非キー・データ・アクセスでの索引付けの使用に関する説明および複合 HashIndexを参照してください。

データ品質に関する考慮事項

索引照会メソッドの結果が表わすのは、特定の時刻におけるデータのスナップショットのみです。結果がアプリケーションに戻された後には、データ・エントリーに

対するロックは取得されません。アプリケーションは、戻されたデータ・セットに対してデータ更新が発生する可能性があることに注意する必要があります。例えば、アプリケーションは MapIndex の findAll メソッドを実行して、キャッシュされたオブジェクトのキーを取得します。戻されたこのキー・オブジェクトは、キャッシュ内のデータ項目に関連付けられています。アプリケーションは、キー・オブジェクトを提供することにより、ObjectMap に対して get メソッドを実行して、オブジェクトを検出できるようになっている必要があります。get メソッドが呼び出される直前に、別のトランザクションがキャッシュからそのデータ・オブジェクトを削除した場合、戻される結果はヌルです。

索引付けのパフォーマンスに関する考慮事項

索引付けフィーチャーの主な目的の 1 つは、BackingMap の全体的なパフォーマンスを改善することです。索引付けの使い方が不適切な場合は、アプリケーションのパフォーマンスが低下する可能性があります。このフィーチャーを使用する前に、次の要因について検討します。

- **並行書き込みトランザクションの数:** 索引処理は、トランザクションが BackingMap にデータを書き込むたびに起こりえます。アプリケーションが索引照会操作を試行しているときに、多くのトランザクションがデータをマップに書き込んでいると、パフォーマンスが低下します。
- **照会操作で戻される結果セットのサイズ:** 結果セットのサイズが大きくなるにつれて、照会のパフォーマンスは低下します。結果セットのサイズが BackingMap の 15% 以上になるとパフォーマンスは低下する傾向にあります。
- **同じ BackingMap に作成される索引の数:** 各索引がシステム・リソースを消費します。BackingMap に作成される索引の数が増えると、パフォーマンスは低下します。

索引付け機能は、BackingMap パフォーマンスを大幅に改善できることがあります。理想的なケースは、BackingMap の大部分の操作が読み取りであり、照会の結果セットが BackingMap エントリーのわずかな割合に過ぎず、ごく少数の索引が BackingMap に対して作成される場合です。

Java オブジェクトのキャッシング概念

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、いくつかの API を使用して eXtreme Scale グリッドと対話することができます。

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、 11 ページの『キャッシング・アーキテクチャー: マップ、コンテナー、クライアント、およびカタログ』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、クラスパスにクラスを定義する必要があります。

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 hashCode 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。これらのプロセスはデータをシリアライズします。つまり、クライアント・プロセスとサーバー・プロセスの間でデータを移動させるために、(Java オブジェクト・インスタンス形式の) データをバイトに変換し、必要に応じて再びオブジェクトに戻します。 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。

詳しくは、キャッシュ・オブジェクトのシリアライズおよびコピーのためのプラグインおよびObjectTransformer インターフェースのベスト・プラクティスを参照してください。

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。

詳しくは、「*プログラミング・ガイド*」で CopyMode メソッドのベスト・プラクティスを参照してください。

異なる時間帯のデータの挿入

カレンダー属性、java.util.Date 属性、およびタイム・スタンプ属性でデータを ObjectGrid に挿入する場合、特にさまざまな時間帯の複数のサーバーにデプロイするときには、これらの日時属性が同じ時間帯を基に作成されるようにする必要があります。 同じ時間帯を基にした日時オブジェクトを使用すれば、アプリケーションの時間帯の問題はなくなり、データはカレンダー述部、java.util.Date 述部、タイム・スタンプ述部によって照会が可能です。

日時オブジェクトの作成時に明示的に時間帯を指定しないと、Java はローカル時間帯を使用し、クライアントとサーバーで日時値が不整合になる場合があります。

分散デプロイメントの例を考えてみます。client1 は時間帯 [GMT-0] にあり、client2 は [GMT-6] にあります。どちらも java.util.Date オブジェクトを値「1999-12-31 06:00:00」で作ろうとしています。次に、client1 は java.util.Date オブジェクトを値「1999-12-31 06:00:00 [GMT-0]」で作成し、client2 は java.util.Date オブジェクトを値「1999-12-31 06:00:00 [GMT-6]」で作成します。 時間帯が異なるため、両方の java.util.Date オブジェクトは等しくありません。 異なる時間帯のサーバーに存在する区画にデータをプリロードする際に、ローカル時間帯を使用して日時オブジェクトを作成していると同じような問題が起こります。

前述の問題を避けるため、カレンダー・オブジェクト、java.util.Date オブジェクト、およびタイム・スタンプ・オブジェクトを作成するための基本の時間帯として [GMT-0] などの時間帯をアプリケーションは選択することができます。

詳しくは、「*プログラミング・ガイド*」の複数の時間帯でのデータ照会に関するトピックを参照してください。

第 3 章 キャッシュ統合の概要: JPA、セッション、および動的キャッシング

多用途性や信頼性などを持ちながら実行する能力を WebSphere eXtreme Scale に付与する重大な要素は、キャッシング概念のアプリケーションです。それは、ほとんどすべてのデプロイメント環境で、データのパーシスタンスや再収集を最適化します。

JPA ローダー

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使用できます。

eXtreme Scale と一緒に Java Persistence API (JPA) ローダー・プラグイン実装を使用すると、選択されたローダーがサポートする任意のデータベースと対話することができます。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。

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。

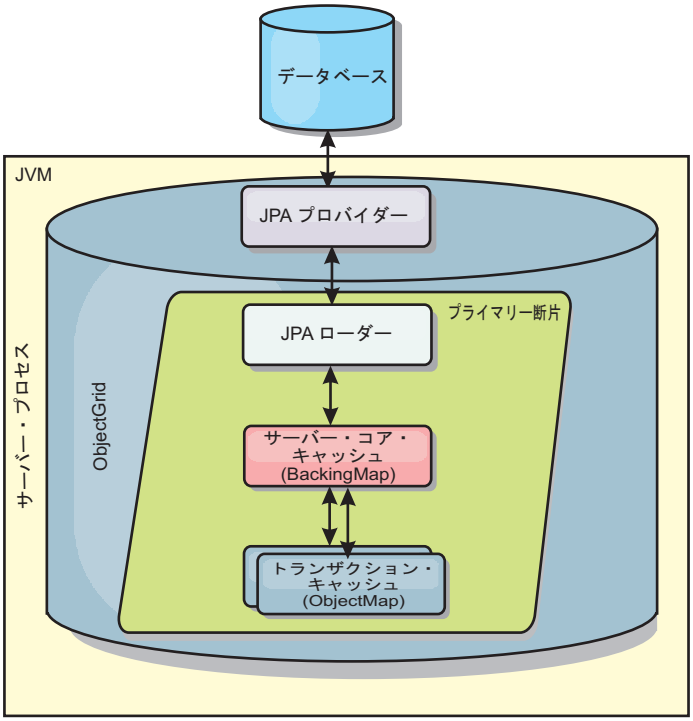


図 29. JPA ローダー・アーキテクチャー

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、*プログラミング・ガイド*の JPAEntityLoader プラグインに関する説明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。

構成およびプログラミング

JPA ローダーの構成について詳しくは、「*管理ガイド*」で JPA ローダーの構成に関する説明を参照してください。JPA ローダーのプログラミングについて詳しくは、*プログラミング・ガイド*を参照してください。

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。

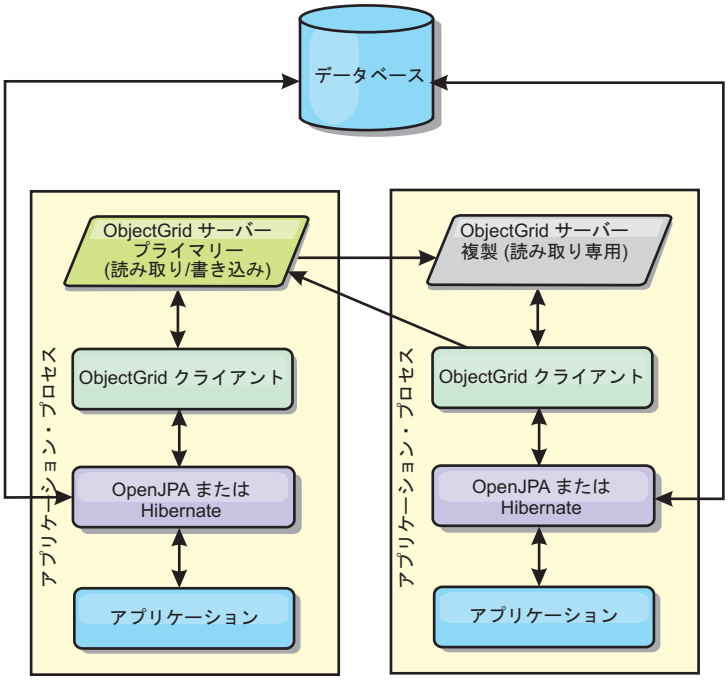


図 30. JPA 組み込みトポロジー

利点:

- 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。

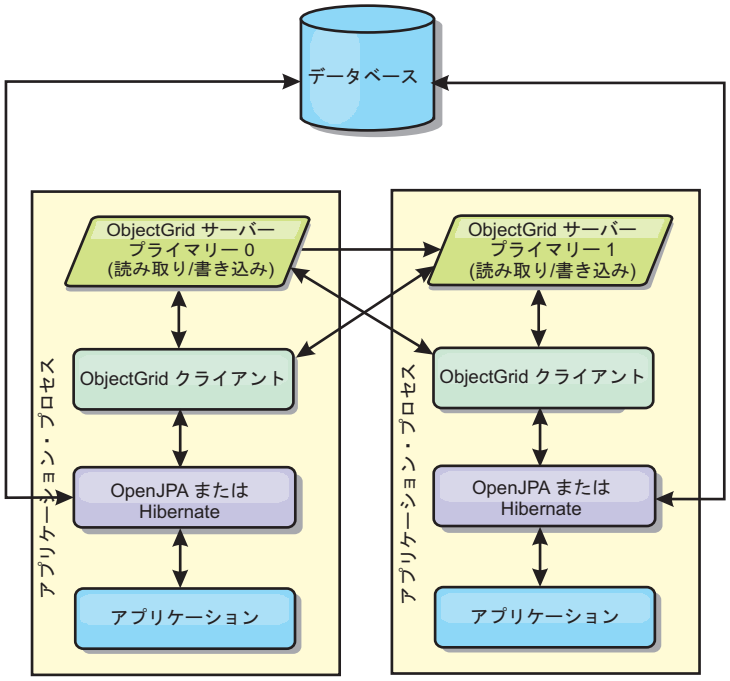


図 31. JPA 組み込みの区画化トポロジー

利点:

- 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 仮想マシンの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。

選択された構成のパフォーマンスを最大化するには、「*管理ガイド*」の『メモリー・サイズ設定および区画数の計算』を参照してください。

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、複製された eXtreme Scale グリッドをデプロイすることによって、個別のプロセスへデータ配布するという利点が生かせます。 前のセクションで説明した組み込み構成および組み込み区画化構成とは対照的に、リモート・グリッドを管理する場合は、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する必要があります。 eXtreme Scale グリッド・デプロイメントの管理について詳しくは、デプロイメント環境のモニターを参照してください。

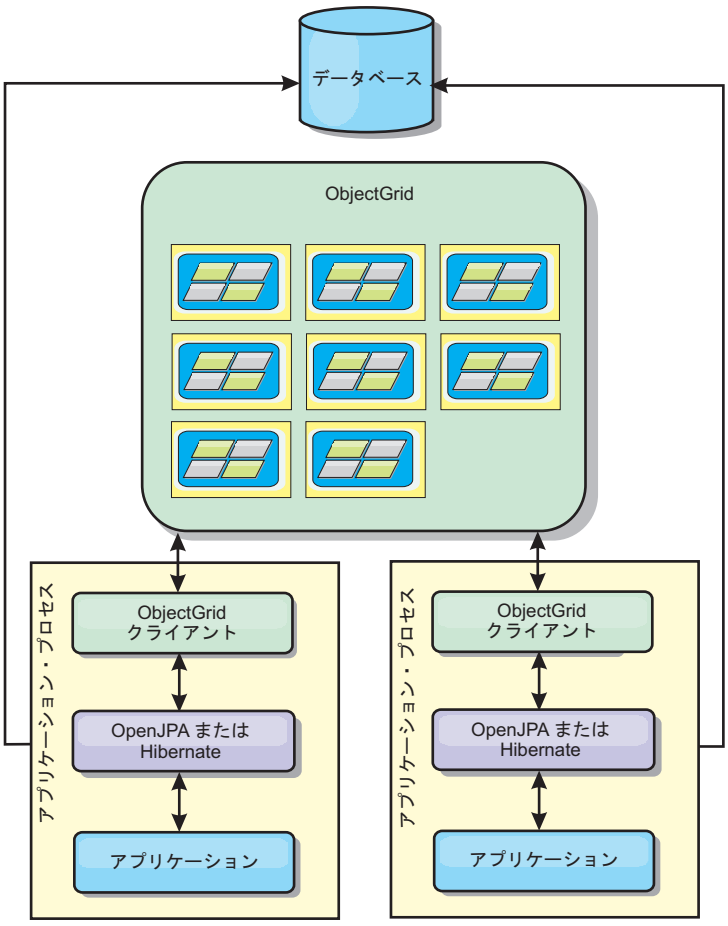


図 32. JPA リモート・トポロジー

利点:

- 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。

構成

JPA キャッシュ・プラグインの構成について詳しくは、*プログラミング・ガイド* のプラグイン・セクションを参照してください。

HTTP セッション管理

WebSphere eXtreme Scale に付属のセッション複製マネージャーは、アプリケーション・サーバーのデフォルト・セッション・マネージャーと連動し、ユーザー・セッション・データの高可用性をサポートするために、あるプロセスから別のプロセスへセッション・データを複製することができます。

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複製機能を提供します。 セッション複製マネージャーは、Web コンテナーのセッション・マネージャーと連動して HTTP セッションを作成し、そのアプリケーションに関連付けられた 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。
- 複数のデータ・センターまたは複数のゾーン。

セッション・マネージャーの動作

セッション複製マネージャーは、標準セッション・リスナーを使用して、セッション・データの変更点で listen し、そのセッション・データをローカルまたはリモートの ObjectGrid インスタンス内に永続化します。 セッション・データは、ローカルまたはリモートの ObjectGrid インスタンスから標準サーブレットを介して要求パスに再ロードされます。 WebSphere eXtreme Scale に付属のツールを使用して、セッション・リスナーおよびサーブレット・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、これらのリスナーおよびフィルターを、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。

このセッション複製マネージャーは、各ベンダーの Web コンテナーのセッション・マネージャーと連動し、Java 仮想マシン間でセッション・データを複製します。元のサーバーが停止したとき、ユーザーはセッション・データを他のサーバーから取得することができます。

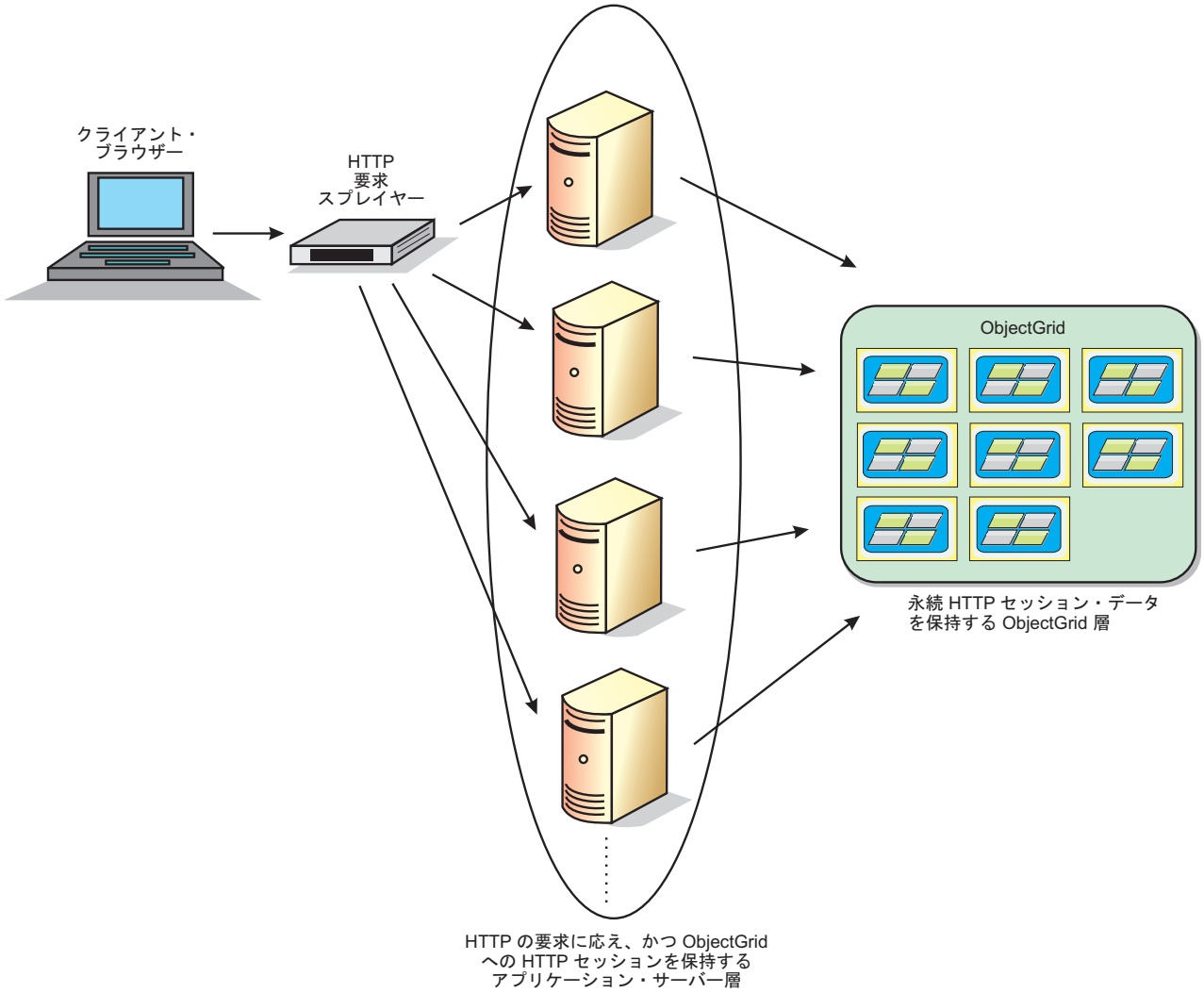


図 33. リモート・コンテナー構成を含む HTTP セッション管理トポロジー

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。このシナリオは、アフィニティーを持って実行され、パフォーマンスが重要な場合に適しています。

- **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の eXtreme Scale サーバー・グリッドと通信します。このシナリオは、Web コンテナー層にセッション・データを保管するメモリーがない場合に適しています。 セッション・デー

タは別の層へ負荷を軽減され、その結果 Web コンテナー層のメモリー使用量は下がりますが、データのリモート・ロケーションのために待ち時間はより大きくなります。

汎用組み込みコンテナーの開始

objectGridType プロパティーが EMBEDDED に設定されている場合、eXtreme Scale は、Web コンテナーがセッション・リスナーまたはサーブレット・フィルターを初期化する際に、任意のアプリケーション・サーバー・プロセス内で自動的に組み込み ObjectGrid コンテナーを開始します。詳しくは、サーブレット・コンテキスト初期化パラメーターを参照してください。

eXtreme Scale バージョン 7.1 を使用して ObjectGrid.xml ファイルおよび objectGridDeployment.xml ファイルを Web アプリケーションの WAR ファイルまたは EAR ファイルへパッケージする必要はありません。デフォルトの ObjectGrid.xml ファイルおよび objectGridDeployment.xml ファイルは製品の JAR へパッケージされています。動的マップは、さまざまな Web アプリケーション・コンテキストに対してデフォルトで作成されます。 静的な eXtreme Scale マップは引き続きサポートされます。

組み込み ObjectGrid コンテナーを開始するためのこのアプローチは、どのタイプのアプリケーション・サーバーにも適用されます。WebSphere Application Server コンポーネントまたは WebSphere Application Server Community Edition GBean を組み込むアプローチは推奨されません。

リスナー・ベースのセッション複製マネージャー

WebSphere eXtreme Scale に付属の eXtreme Scale セッション複製マネージャーは、アプリケーション・サーバーのデフォルト・セッション・マネージャーと連動し、ユーザー・セッション・データの高可用性をサポートするために、あるプロセスから別のプロセスへセッション・データを複製することができます。

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複製機能を提供します。 セッション複製マネージャーは、Web コンテナーのセッション・マネージャーと連動して HTTP セッションを作成し、そのアプリケーションに関連付けられた 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。
- 複数のデータ・センターまたは複数のゾーン。

セッション・マネージャーの詳細

セッション複製マネージャーは、標準セッション・リスナーを使用して、セッション・データの変更点で listen し、そのセッション・データをローカルまたはリモートの ObjectGrid インスタンス内に永続化します。 セッション・データは、ローカルまたはリモートの ObjectGrid インスタンスから標準サーブレットを介して要求パスに再ロードされます。 WebSphere eXtreme Scale に付属のツールを使用して、セッション・リスナーおよびサーブレット・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、これらのリスナーおよびフィルターを、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。

セッション複製マネージャーは、各ベンダーの基本セッション・マネージャーと連動し、アプリケーションのセッション・データを複製します。 次の考慮事項に注意してください。

- パフォーマンス要件および使用するデータ・サイズに応じて、組み込み ObjectGrid コンテナーまたはリモート ObjectGrid コンテナーを選択します。 組み込みシナリオを使用すると、最も簡単に構成でき、パフォーマンスも向上します。

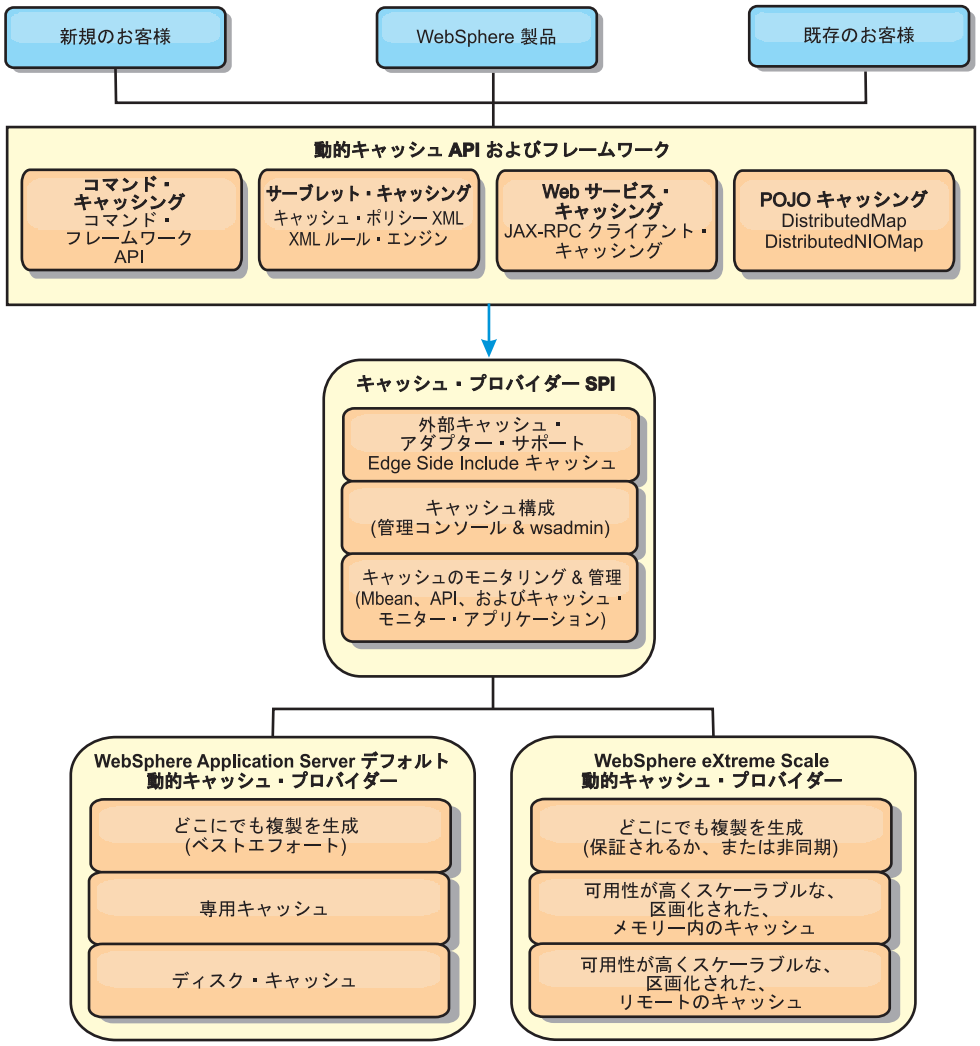
- ユーザーのデータ・サイズに応じて、Web コンテナーごとのリモート ObjectGrid コンテナーの数を選択します。
- 属性のユーザー・データ数とサイズ、および変更頻度に応じて、セッション・データ全体を一緒に保管するか、あるいは各属性を別々に保管するかを選択します。
- 複製間隔を選択します。 複製間隔は、あまり短くないほうがパフォーマンスは向上します。
- ローカル・メモリー・サイズとパフォーマンスのバランスを保つようなセッション・テーブルのサイズを選択します。 この製品は、ローカル・セッション・キャッシュの最大サイズに達すると、セッション・ユーザー・データの負荷を軽減します。
- この製品は、サーブレット仕様に従ってアプリケーション・コンテキスト内の HTTP セッションをサポートし、セキュリティー問題および使用属性の名前の競合問題が起こらないようにします。 singleton クラスによって、アプリケーション・コンテキスト全体でセッションを共有することができます。
- セッション複製マネージャーは、セッション・データの変更の listen や保管されたセッション・データの再ロードをオンデマンドで行うことによって、高可用性のためにセッション・データを複製します。 これは、各ベンダーの Web コンテナーの基本セッション・マネージャーを再利用することで実行されます。 セッションは、さまざまなネイティブ・セッション ID で結合されます。 セッション・データは、ObjectGrid インスタンスからセッション・データを再ロードすることで、ある Web コンテナーから別の Web コンテナーにフェイルオーバーされます。 セッション ID および作成時間はフェイルオーバーの前後で異なる場合があります。

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・プロバイダーは、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。



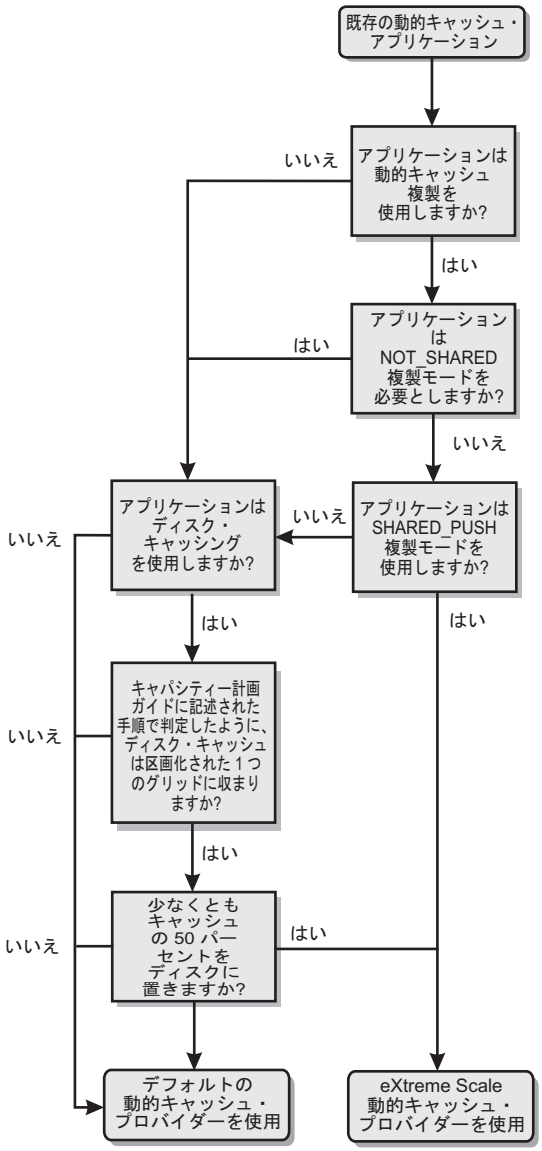
動的キャッシュ・プロバイダーのインストールと構成の手順については、WebSphere eXtreme Scale の動的キャッシュ・プロバイダーの構成を参照してください。

WebSphere eXtreme Scale の利用方法の決定

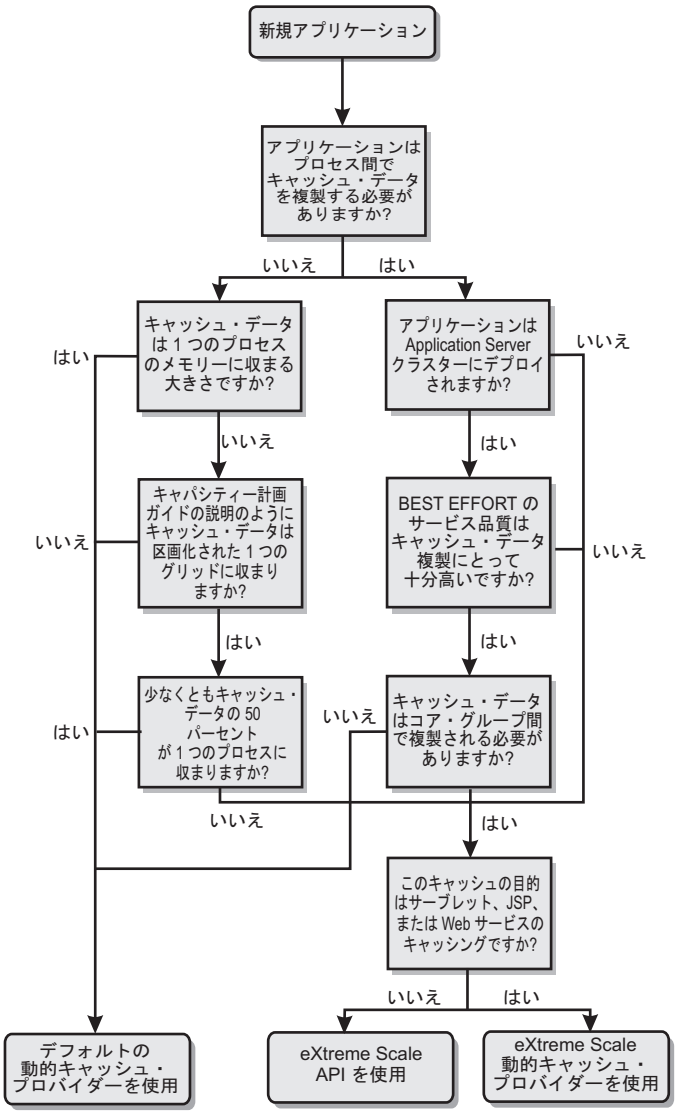
WebSphere eXtreme Scale で使用可能なフィーチャーによって動的キャッシュ API の分散機能は大幅に強化され、デフォルト動的キャッシュ・エンジンおよびデータ複製サービスで提供される機能を超えるものになっています。eXtreme Scale を使用することにより、複数のサーバー間で単に複製し、同期化するだけでなく、サーバー間で本当に分散したキャッシュを作成できます。さらに、eXtreme Scale キャッシュは、トランザクション・ベースであり、可用性がとても高く、動的キャッシュ・サービスに関して各サーバーが同じ内容を参照することを保証します。WebSphere eXtreme 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 7. 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 7. 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 8. 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 9. 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かについて詳しくは、 「*管理ガイド*」でデプロイメント構成の情報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。

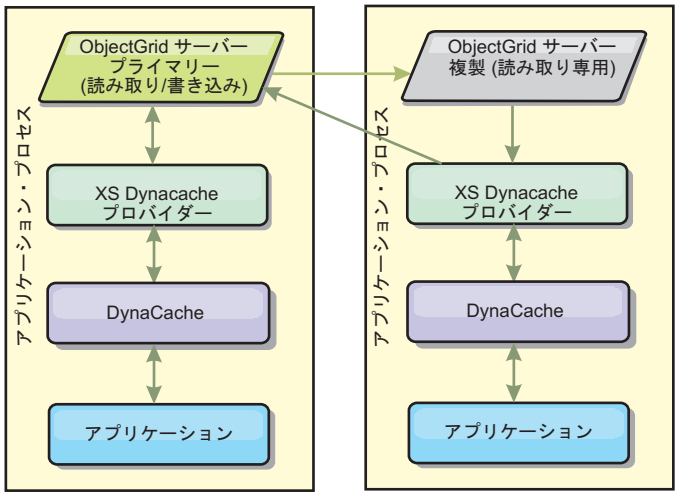
トポロジーのタイプ

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

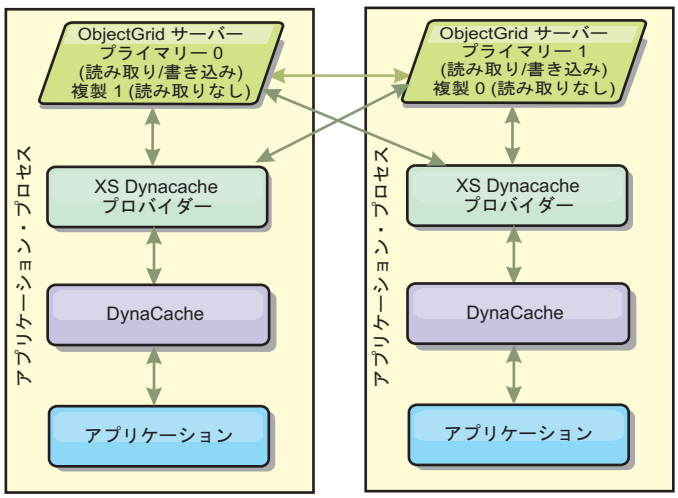
組み込みトポロジーは、デフォルトの動的キャッシュおよび DRS プロバイダーに似ています。組み込みトポロジーで作成された分散キャッシュ・インスタンスは、動的キャッシュ・サービスにアクセスする各 eXtreme Scale プロセスの内部でキャッシュの完全コピーを保持するので、すべての読み取り操作をローカルで実行できます。すべての書き込み操作は、シングル・サーバー・プロセスを完了し、そのプロセスでトランザクションのロックが管理された後で残りのサーバーに複製されます。したがって、このトポロジーは、キャッシュ読み取り操作がキャッシュ書き込み操作よりもずっと多いワークロードに向いています。

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

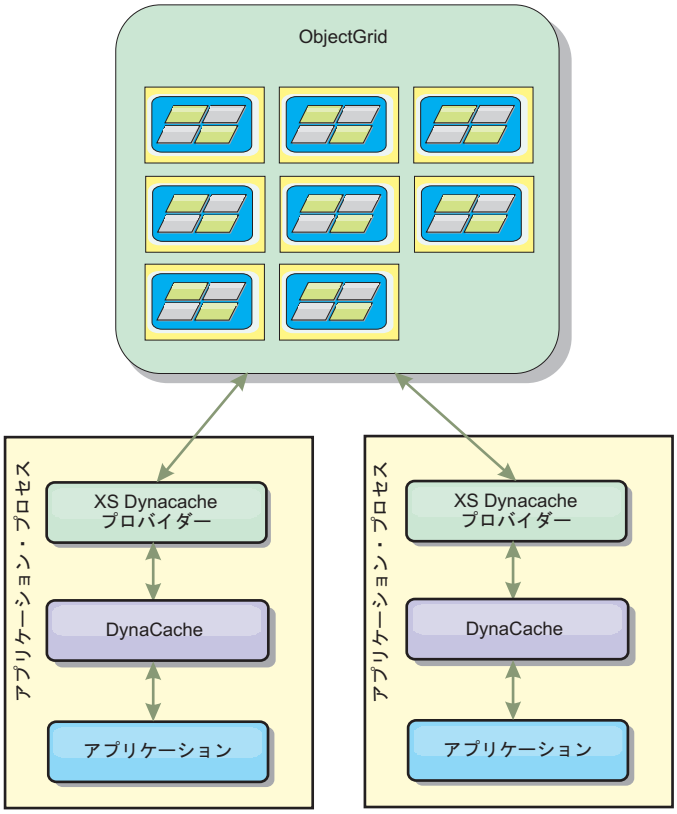
キャッシュ書き込みが読み取りと同じくらいか、より頻繁に発生するようなワークロードの場合、組み込み区画化トポロジーまたはリモート・トポロジーが推奨されます。組み込み区画化トポロジーは、キャッシュにアクセスする WebSphere Application Server プロセスの内部でキャッシュ・データのすべてを保持します。しかし、各プロセスが保管するのは、キャッシュ・データの一部分のみです。この「区画」に置かれたデータに対するすべての読み取りおよび書き込みは、そのプロセスを実行します。したがって、キャッシュに対するほとんどの要求はリモート・プロシージャー・コールで達成されることになります。その結果、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、リモート・トポロジー、組み込みトポロジー、または組み込み区画化トポロジーを使用するキャッシュに対してのみサポートされます。 これは、グリッド全体でキャッシュが消費する Java ヒープ・スペースのメガバイト数を報告します。 この統計は、プライマ

リー区画に対するヒープ使用量のみを報告します。レプリカを考慮してください。 リモート・トポロジーおよび組み込み区画化トポロジーのデフォルト設定は 1 つの非同期レプリカであるため、キャッシュのメモリー消費量を正確に知るには、この数値を 2 倍してください。

- **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 Application Server 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、WebSphere eXtreme Scale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

キャパシティー・プランニングと高可用性 (動的キャッシング)

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、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。

第 4 章 スケーラビリティーの概念の概要

スケーラビリティーによって、WebSphere eXtreme Scale のデプロイメント内のデータを、ユーザーの構成選択に基づいて一連のサーバー (コンテナー) に配布することができます。

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、必要ならば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。

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。

グリッド、区画、および断片

eXtreme Scale 分散グリッドは、区画に分割されています。各区画は、データの排他的なサブセットを保持します。1 つの区画は 1 つ以上の断片 (プライマリー断片と複製断片) から構成されます。区画に必ず複製断片を置く必要はありませんが、複製断片は高可用性をもたらします。デプロイメントが独立したメモリー内のデー

タ・グリッドであれ、メモリー内のデータベース処理スペースであれ、eXtreme Scale でのデータ・アクセスは、断片化の概念に大きく依存します。

1 つの区画のデータは、実行時、断片の集合に保管されます。この断片の集合には、プライマリー断片と、可能性として 1 つ以上の複製断片が含まれます。断片は、eXtreme Scale が Java 仮想マシン に対して追加または除去できる最小の単位です。

配置ストラテジーには、FIXED_PARTITIONS (デフォルト) と PER_CONTAINER の 2 つがあります。以下では、FIXED_PARTITIONS 戦略の使用に焦点を当てて説明します。

断片数

環境にレプリカなしの 1,000,000 のオブジェクトを保持する 10 の区画があるとすると、それぞれ 100,000 のオブジェクトを保管する 10 の断片が存在することになります。このシナリオにレプリカを 1 つ追加すると、各区画には追加の断片が存在することになります。この場合、20 の断片、すなわち、10 のプライマリー断片と 10 の複製断片が存在することになります。この場合もやはり各断片には、100,000 のオブジェクトが保管されます。各区画は、1 つのプライマリー断片と 1 つ以上 (N) の複製断片で構成されます。最適な断片数を決定することは、非常に重要です。少数の断片で構成すると、データが断片間に均等に配分されず、メモリー不足エラーやプロセッサーの過負荷問題が生じることになります。各 JVM あたり、少なくとも 10 の断片になるように見積もってください。当初、グリッドをデプロイする場合、可能性として多数の区画を使用することになります。

JVM あたりの断片数

シナリオ: 少数の JVM あたりの断片数

データは、断片単位を使用して JVM に対して追加および除去されます。断片がさらに小さく分割されることはありません。10 GB のデータがあり、このデータを保持するために 20 の断片が存在しているとすると、各断片には、平均 500 MB のデータが保持されていることになります。9 つの Java 仮想マシンがグリッドをホストしている場合、各 JVM には、平均して 2 つの断片を持ちます。20 は 9 で割りきれませんから、いくつかの Java 仮想マシン は次の配分のように 3 つの断片を持つことになります。

- 2 つの断片を持つ Java 仮想マシン が 7 つ
- 3 つの断片を持つ Java 仮想マシン が 2 つ

各断片には 500 MB のデータが保持されていますから、データ配分は均等ではありません。2 つの断片を持つ 7 つの Java 仮想マシン は、それぞれ 1 GB のデータをホストすることになります。3 つの断片を持つ 2 つの Java 仮想マシン には、50% 多い 1.5 GB のデータが設定され、メモリー負担がずっと大きくなります。これら 2 つの Java 仮想マシン は、3 つの断片をホスティングしているため、そのデータに対しても 50% 多い要求を受信することになります。したがって、各 JVM あたりに少数の断片数が設定される場合は、不均衡が生じます。パフォーマンスを改善するためには、各 JVM あたりの断片数を大きくします。

シナリオ: JVM あたりの断片数を大きくする

このシナリオでは、断片数をより大きくすることを考えます。このシナリオでは、10 GB のデータをホスティングする 9 つの Java 仮想マシンに、101 の断片があるとします。この場合、各断片には 99 MB のデータが保持されます。Java 仮想マシン には、次のように断片が配分されます。

- 11 の断片を持つ Java 仮想マシン が 7 つ
- 12 の断片を持つ Java 仮想マシン が 2 つ

12 の断片を持つ 2 つの Java 仮想マシン は、99 MB のデータだけ他の断片より多くなりますが、これは 9% の違いに相当します。このシナリオの方が、少数の断片によるシナリオの 50% の違いに比べてはるかに均等に配分されています。プロセッサー使用の観点から見れば、12 の断片を持つ 2 つの Java 仮想マシン には、11 の断片を持つ 7 つの Java 仮想マシン に比べてわずか 9% 多くの作業が割り当てられることになります。各 JVM 中の断片数を大きくすることにより、データおよびプロセッサー使用が、平等、均等に配分されることになります。

システムを作成する場合、あるいは、システムが計画期間で最大数の Java 仮想マシンを実行している場合、最大サイズのシナリオとして各 JVM あたり 10 の断片を使用するようにしてください。

追加の配置要因

区画数、配置ストラテジー、およびレプリカの数とタイプは、デプロイメント・ポリシーで設定されます。 配置される断片数は、ユーザーが定義したデプロイメント・ポリシーによって決まります。
numInitialContainers、minSyncReplicas、developmentMode、maxSyncReplicas および maxAsyncReplicas は、区画とレプリカが配置される場所と時間に影響します。サーバーの初期始動で maxSyncReplicas および maxAsyncReplicas の配置が許可されない場合、後で追加サーバーを始動させるならば、追加のレプリカを配置することもあります。 JVM ごとの断片数を計画する場合、レプリカを含む断片の最大数は、要求されるレプリカの最大数をサポートするのに十分な JVM が開始されているかどうかによって異なります。 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。developmentMode が false の場合、プライマリーとレプリカは同じマシン上に配置されません。

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。WebSphere eXtreme Scale では、区画化によってスケーラビリティーと可用性の両方が向上します。

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。

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。

ObjectMap および区画化

デフォルトの FIXED_PARTITION 配置ストラテジーでは、マップは区画間で分割され、キーは異なる区画にハッシュされます。 クライアントは、どの区画にキーが属

しているのかを知る必要はありません。 mapSet に複数のマップがある場合、マップは別々のトランザクションでコミットされるはずです。

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。オブジェクト・グラフは、1 つ以上のエンティティー間のリレーションシップを定義します。 リンクされた各エンティティーは同じ区画に解決される必要があります。 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。オブジェクト・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、固定区画とコンテナーごとの、2 つの配置ストラテジーがあります。 配置ストラテジーの選択は、デプロイメント構成が区画をどのようにリモート・グリッド内に配置するかに影響します。

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。

WebSphere eXtreme Scale では、「標準的」配置ストラテジーと称される、区画の 1 つにマップのキーをハッシュする固定区画を使用した方法の代替として、コンテナーごとの配置が可能です。 コンテナーごとの場合 (PER_CONTAINER で設定)、デプロイメントは区画を一連のオンライン・コンテナー・サーバーに配置し、コンテナーがサーバー・グリッドに追加またはサーバー・グリッドから削除されるのに合わせて、自動的にスケールアウトまたはスケールインします。 固定区画を使用した方法のグリッドは、キー・ベースのグリッドに使用すると効果があり、アプリケーションはキー・オブジェクトを使用してグリッドのデータを検索します。 次に、代替方法について説明します。

コンテナーごとのグリッドの例

PER_CONTAINER グリッドはさまざまです。 デプロイメント XML ファイルの placementPolicy 属性で、グリッドが PER_CONTAINER を使用するように指定します。 グリッド内の区画が合計でいくつ必要なのかを構成する代わりに、開始するコンテナーごとに区画がいくつ必要なのかを指定します。

例えば、コンテナーごとの区画数を 5 に設定する場合、コンテナーを 1 つ開始すると、eXtreme Scale は 5 つの新しい匿名プライマリー区画をそのコンテナーに作成し、必要なレプリカを既にデプロイ済みの他のコンテナーに作成します。

以下は、グリッドが拡張していくに従い、コンテナーごとの環境で可能性のあるシーケンスです。

1. 5 つのプライマリー (P0 から P4) をホスティングしているコンテナー C0 を開始します。
   - C0 ホスト: P0、P1、P2、P3、P4。
2. さらに 5 つのプライマリー (P5 から P9) をホスティングしているコンテナー C1 を開始します。 コンテナー上でレプリカのバランスが取られます。
   - C0 ホスト: P0、P1、P2、P3、P4、R5、R6、R7、R8、R9。
   - C1 ホスト: P5、P6、P7、P8、P9、R0、R1、R2、R3、R4。
3. さらに 5 つのプライマリー (P10 から P14) をホスティングしているコンテナー C2 を開始します。 さらにレプリカのバランスが取られます。
   - C0 ホスト: P0、P1、P2、P3、P4、R7、R8、R9、R10、R11、R12。
   - C1 ホスト: P5、P6、P7、P8、P9、R2、R3、R4、R13、R14。
   - C2 ホスト: P10、P11、P12、P13、P14、R5、R6、R0、R1。

さらに多くのコンテナーが開始される間このパターンが続き、5 つの新しいプライマリー区画が毎回作成されて、グリッド内の使用可能なコンテナー上で再度レプリカのバランスが取られます。

注: WebSphere eXtreme Scale は PER_CONTAINER ストラテジーを使用する場合、プライマリーは移動せず、レプリカのみを移動します。

区画番号は任意でキーと無関係なため、キー・ベースのルーティングは使用できないことに注意してください。 コンテナーが停止するとそのコンテナー用に作成された区画 ID は使用されなくなるので、区画 ID の間に抜けができます。 例では、コンテナー C2 に障害が起こると区画 P5 から P9 がなくなり、P0 から P4 と P10 から P14 のみが残るため、キー・ベースのハッシュは不可能です。

コンテナーの障害の影響を考慮すれば、5 あるいはさらに適当な 10 などの数字をコンテナーごとの区画数に使用するのが最も適切に機能します。 グリッド中に断片を均等にホスティングするという負荷を分散するには、各コンテナーに対して複数の区画が必要です。 コンテナーごとの区画が単一だった場合、コンテナーに障害が起こると 1 つのコンテナー (対応するレプリカ断片をホスティングするコンテナー) だけで失われたプライマリーの負荷をすべて引き受けなければなりません。 この場合、負荷はコンテナーに対して直ちに 2 倍になります。 ただし、コンテナーごとに 5 つの区画があれば、5 つのコンテナーが失われたコンテナーの負荷を受け取り、各コンテナーへの影響は 80 パーセント減少します。 コンテナーごとに複数の区画を使用すると、各コンテナーへの実質的な影響の可能性は一般的に低くなります。 さらに直接的に、コンテナーが予想外に急増するケースを考えてください。コンテナーの複製の負荷は、1 つだけではなく 5 つのコンテナーに分散されます。

コンテナーごとのポリシーの使用

いくつかのシナリオでは、HTTP セッション複製またはアプリケーション・セッション状態などの場合、コンテナーごとのストラテジーは理想的な構成だと考えられています。 そのような場合、HTTP ルーターはセッションをサーブレット・コンテナーに割り当てます。 サーブレット・コンテナーは HTTP セッションを作成する必要があり、5 つのローカル・プライマリー区画のうちの 1 つをそのセッション用に選択します。 その後、選択された区画の「ID」は Cookie に保管されます。 これでサーブレット・コンテナーは、セッション状態へのローカル・アクセス権限を持ち、セッション・アフィニティーが保持されている限り、この要求に対するデータへのアクセス待ち時間はゼロになることを意味します。 さらに、eXtreme Scale は、すべての変更を区画に複製します。

実際には、コンテナーごとに複数区画を持つ場合の悪影響に注意してください (再度 5 つの例を使用します)。 当然、新たに開始した各コンテナーには、さらに 5 つのプライマリー区画と、さらに 5 つのレプリカがあります。 時間とともに、さらに区画が作成され、移動または破棄されないはずです。しかし、これは実際にコンテナーが振る舞う様子ではありません。 コンテナーは、開始すると 5 つのプライマリー断片をホストします。これは「ホーム」プライマリーと呼ばれ、それらを作成したそれぞれのコンテナー上に存在します。 コンテナーに障害が起こると、レプリカがプライマリーになり、eXtreme Scale はさらに 5 つのレプリカを作成し、高可用性を保持します (自動修復を使用不可にしていない場合)。新規プライマリーはそれを作成したコンテナーとは別のコンテナーにあり、「外部」プライマリーと呼ばれます。 アプリケーションは、新規状態または新規セッションを外部プライマリーには決して配置しません。 最終的に、外部プライマリーはエントリーを持たず、eXtreme Scale は外部プライマリーとその関連レプリカを自動的に削除します。外部プライマリーの目的は、既存のセッション (新しいセッションではなく) を引き続き使用可能にしておくことです。

クライアントは、キーを利用しないグリッドとも対話することができます。 クライアントは、ただトランザクションを開始し、データをどのキーとも無関係のグリッドに保管します。クライアントは、必要なときに同じ区画と対話できるようにするシリアライズ可能ハンドル、SessionHandle オブジェクトをセッションに要求します。 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。 WebSphere eXtreme Scale は、ホーム・プライマリー区画のリストからクライアント用の区画を選択します。外部プライマリー区画は戻されません。 SessionHandle は、例えば HTTP Cookie でシリアライズされ、後で Cookie を 元の SessionHandle に変換することができます。 次に WebSphere eXtreme Scale API は SessionHandle を使用して、再度同じ区画にバインドされたセッションを取得します。

注: エージェントを使用して PER_CONTAINER グリッドと対話することはできません。

利点

コンテナーごとのクライアントは、データをグリッド内の場所に保管し、データへのハンドルを取得し、そのハンドルを使用してデータに再びアクセスするため、上記の説明は通常の FIXED_PARTITION またはハッシュ・グリッドと異なります。固定区画の場合のような、アプリケーション提供のキーはありません。

デプロイメントは、各セッションごとに新規区画を作りません。 そのため、コンテナーごとのデプロイメントでは、データを区画に保管するために使用されるキーは、その区画内で固有でなければなりません。 例えば、クライアントが固有の SessionID を生成し、それをキーとして使用してその区画内のマップの情報を検索するという例が考えられます。 複数のクライアント・セッションが同じ区画と対話するため、アプリケーションは固有のキーを使用してセッション・データを特定の各区画に保管する必要があります。

上記の例では 5 つの区画を使用しましたが、objectgrid XML ファイルの numberOfPartitions パラメーターを使用すると、必要に応じて区画を指定することができます。グリッドごとではなく、設定はコンテナーごとです。 (レプリカの数は、固定区画ポリシーの場合と同じ方法で指定されます。)

コンテナーごとのポリシーは、複数ゾーンでも使用できます。 可能であれば、eXtreme Scale は、プライマリーがそのクライアントと同じゾーンにある区画に SessionHandle を返します。クライアントは、コンテナーのパラメーターとして、または API を使用してゾーンを指定できます。 クライアント・ゾーン ID は `serverproperties` または `clientproperties` を使用して設定することができます。

グリッドの PER_CONTAINER ストラテジーは、データベース指向データではなく、会話型状態を保管するアプリケーションに適しています。データにアクセスするキーは会話 ID で、特定のデータベース・レコードには関係しません。 このストラテジーにより、パフォーマンスはさらに向上し (例えばプライマリー区画をサーブレットと連結できるので)、構成はさらに容易になります (区画およびコンテナーを計算する必要はありません)。

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、基本原則と実装ヒントという 2 つのカテゴリーがあります。基本原則は、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的な原則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エントリー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、通常膨大な数のエントリー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。

ユニットまたはポッドでの拡張

このトピックではスケーラビリティーについて説明しますが、JVM の X 番にグリッドをスケールアウトする方法など、一般的なやり方については説明しません。 代わりにここでは、操作、計画、およびリスク管理の観点からスケーラビリティーについて説明します。 これらの要因は、通常、従来の製品のスケーラビリティーに関する考慮事項よりも重要ですが、残念ながら無視されがちです。 高可用性なシステムを構築するには、スケーラビリティーに関して両方のタイプを考慮して、信頼できるデプロイメント・プロセスで eXtreme Scale をデプロイする必要があります。

大きな単一グリッドのデプロイ

テストによって、eXtreme Scale が 1000 を超す JVM にスケールアウトできることが検証されています。 このようなテストでは、単一グリッドを多数のコンピューターにデプロイするようなアプリケーションの構築を勧められます。そのようなアプリケーションの構築は可能ですが、以下のようないくつかの理由により推奨されません。

1. **予算の問題:** ご使用の環境では現実的に 1000 のサーバー・グリッドをテストできません。 ただし、予算の理由を考慮して、より小さなグリッドをテストすることはできますので、特にそのような多数のサーバーの場合、2 倍のハードウェアを購入する必要はありません。
2. **異なるアプリケーションのバージョン:** スレッドをそれぞれテストするために多数のコンピューターを必要とするのは実用的ではありません。 実稼働環境で行うのと同じ要因をテストしないことがリスクになります。
3. **データ損失:** データベースを単一ハード・ディスクで実行することは信用性が高くありません。 ハード・ディスクに問題が発生すると、データ損失の原因になります。成長するアプリケーションを単一グリッドで実行するのも同様です。

ご使用の環境およびご使用のアプリケーションにバグがある可能性があります。したがって、すべてのデータを大きな単一システムに配置することによって、大量のデータの損失につながることがあります。

グリッドの分割

アプリケーション・グリッドをポッド (ユニット) に分割することは、より信頼性が高くなります。 ポッドとは、同種のアプリケーション・スタックを実行するサーバーの一群です。 ポッドのサイズは任意ですが、約 20 台のコンピューターで構成されるのが理想的です。 単一グリッドに 500 台のコンピューターがあるよりも、20 台のコンピューターの 25 ポッドにしてください。 単一の種類のアプリケーション・スタックは、指定されたポッドで実行する必要がありますが、異なるポッドが独自の種類のアプリケーション・スタックを持つことは構いません。

一般的に、アプリケーション・スタックは以下のコンポーネント・レベルを考慮します。

- オペレーティング・システム
- ハードウェア
- JVM
- eXtreme Scale のバージョン
- アプリケーション
- その他の必要なコンポーネント

ポッドは、テストに都合のいいようにサイズ変更したデプロイメント・ユニットです。数百台のサーバーでテストを行う代わりに、20 台のサーバーで行うのはより実用的です。 この場合、実動環境と同じ構成を引き続きテストしていきます。 実動環境では、1 つのポッドを構成する、20 台のサーバーの最大サイズでグリッドが使用されます。 1 つのポッドにストレス・テストをかけ、そのキャパシティー、ユーザー数、データ量、およびトランザクション・スループットを判別できます。 これによって、より簡単に計画が立てやすく、予測可能なコストで予測可能な拡張を行うという規格に従うことができます。

ポッド・ベースの環境の設定

別のケースでは、ポッドには必ずしも 20 台のサーバーが必要というわけではありません。 ポッドのサイズの目的は、実用的なテストのためです。実動環境でポッドに問題が発生しても、影響を受ける一部のトランザクションが耐えられるように、ポッドのサイズは十分に小さくしてください。

バグは 1 つのポッドにしか影響しないのが理想的です。 前の例で、バグは 100 パーセントではなく、アプリケーション・トランザクションの 4 パーセントにしか影響を与えません。 さらに、一度に 1 つのポッドをロールアウトできるので、アップグレードはより簡単です。 これにより、シナリオは単純化されるため、ポッドへのアップグレードで問題が生じた場合、ユーザーはそのポッドを切り替えて前のレベルに戻すことができます。 アップグレードには、アプリケーションに対する変更、アプリケーション・スタックに対する変更、またはシステム更新を含みます。問題診断をより正確に行うために、アップグレードではできる限り、スタックのエレメントを一度に 1 つだけ変更するようにしてください。

ポッドを使用する環境を実装するには、ポッドにソフトウェアのアップグレードがあった場合に上下に互換性のあるルーティング層がポッドの上に必要です。 また、どのポッドが何のデータを持っているかについての情報を含むディレクトリーを作成する必要もあります。 (このために、できれば後書きシナリオを使って、別の eXtreme Scale グリッドをその後ろのデータベースと一緒に使用してください。) これは、2 層の解決策を生み出します。 層 1 はディレクトリーで、特定のトランザクションを処理するポッドを検索するために使用されます。 層 2 はポッド自体で構成されます。 層 1 がポッドを識別すると、セットアップは各トランザクションをポッド内の適切なサーバーに送付します。このサーバーは、通常、トランザクションが使用するデータ用区画を保持するサーバーです。 さらに、必要であれば層 1 でニア・キャッシュを使用して、適切なポッドの検索に関連する影響を小さくすることができます。

ポッドの使用は、単一グリッドを持つよりも少し複雑ですが、操作、テスト、および信頼性の面で改善されるため、スケーラビリティーのテストの重要な一部になっています。

第 5 章 可用性の概要

高可用性

高可用性を備えている WebSphere eXtreme Scale は、信頼できるデータの冗長性を備え、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、 11 ページの『キャッシング・アーキテクチャー: マップ、コンテナー、クライアント、およびカタログ』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、 122 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、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・ドメインでは、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 N 個の中の 1 つのサービスであることに注意してください。 N 個の中の 1 つのサービスは、高可用性グループのメンバーの 1 つで実行されます。 ロケーション・サービスおよび管理は、高可用性グループのすべてのメンバーで実行されています。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、 97 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。

複製された eXtreme Scale 用のローダーの実装例については、「*プログラミング・ガイド*」で、複製プリロード・コントローラーを使用するローダーの作成に関する説明を参照してください。

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 10. 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。
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 11. 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 12. 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。

フェイルオーバー検出のタイプ

WebSphere eXtreme Scale は、障害を確実に検出できます。

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す。また、この検出機能により、通常 1 秒足らずで、こうしたタイプの障害から回復できます。
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検出します。

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。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即時に送信され、各接続がクローズされます。

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

ホスト・マシンの電源が切断されると、ホスト・マシンは瞬時に使用不可になります。

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。

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・ドメインでは、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 N 個の中の 1 つのサービスであることに注意してください。 N 個の中の 1 つのサービスは、高可用性グループのメンバーの 1 つで実行されます。 ロケーション・サービスおよび管理は、高可用性グループのすべてのメンバーで実行されています。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。

表 13. 障害のディスカバリーおよび復旧の要約

ロス・タイプ	ディスカバリー (検出) メカニズム	復旧メソッド
プロセス・ロス	入出力	再始動
サーバー・ロス	ハートビート	再始動
ネットワーク障害	ハートビート	ネットワークおよび接続の再確立
サーバー・サイド・ハング	ハートビート	サーバーの停止および再始動
サーバー・ビジー	ハートビート	サーバーが使用可能になるまで待機

高可用性カタログ・サービス

カタログ・サービス・ドメインは、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・サービスをカタログ・サービス・ドメインにクラスター化する必要があります。

カタログ・サービス・ドメインのコンポーネント

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・サービス・ドメインのルーティング・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・サービス・ドメインのルーティング・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・サービス・ドメインは、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・サービス・ドメインのルーティング・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・サービス・ドメインのデプロイメント

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 デプロイメント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。配置サービスは、データ・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、サービスのインスタンスは 1 つだけが実行されます。 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・サービス・ドメインが開始されると、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・サービス・ドメイン構成を実行時に変更することはできないので、カタログ・サービス・ドメイン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・サービス・ドメインの開始

カタログ・サービス・ドメインの作成について詳しくは、*管理ガイド*にあるカタログ・サービス・ドメインの開始についての説明を参照してください。

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・サービス・ドメインへの接続

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・サービス・ドメインに接続できます。

-  (非推奨) 過去のリリースでは、カスタム・プロパティーを作成することによって、カタログ・サービスをカタログ・サービス・ドメインに接続しました。 このプロパティーをそのまま使用することはできますが、推奨されません。 このカスタム・プロパティーについて詳しくは、 「*管理ガイド*」にある WebSphere Application Server でのカタログ・サービス・プロセスの開始に関する情報を参照してください。

  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。

詳しくは、『カタログ・サーバー・クォーラム』を参照してください。

カタログ・サーバー・クォーラム

クォーラムとは、データ・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、クォーラムの値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、WebSphere 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。サーバーのグループに対するゾーンの例として、上の箇条書きの中で説明したデータ・センター、エリア・ネットワーク、ビル、ビルの 1 フロアなどがあります。
- **ハートビート:** ハートビートすることは、指定された Java 仮想マシン (JVM) が稼働中かどうかを判別するために使用されるメカニズムです。

トポロジー

このセクションでは、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 の要素をホスティングする Java 仮想マシン (JVM) が使用する IP アドレスとポートの間をすべてのトラフィックが流れることを許可するよう、すべてのファイアウォールに要求します。

接続された LAN

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

カタログ・サービス・ドメインとコンテナー・サーバー

カタログ・サービス・ドメインとは類似した JVM の集合をいいます。カタログ・サービス・ドメインはカタログ・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・ドメインはフル・クォーラム・メカニズムを使用します。このフル・クォーラム・メカニズムにより、データ・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データ・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/stopOgServer 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・ドメイン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・ドメインのハートビート

カタログ・サービス・ドメイン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 コマンド照会では不可視です。コンテナー JVM のログのメッセージは、コンテナー JVM が失敗したことを示します。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・ドメインは 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・ドメインに再参加し、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・ドメインに再参加します。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・ドメインに再参加し、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・ドメイン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。

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
```

トランスポート・セキュリティーの考慮事項

データ・センターは、通常、地理的に異なるロケーションにデプロイされるため、ユーザーは、安全上の理由からデータ・センター間のトランスポート・セキュリティーを使用可能にしたい場合があります。

「*管理ガイド*」のトランスポート層セキュリティーについて参照してください。

レプリカおよび断片

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。各区画にはインスタンスがあり、そのインスタンスは、プライマリー断片と構成可能な複数の複製断片です。 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
- 同期レプリカ
- 非同期レプリカ

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。

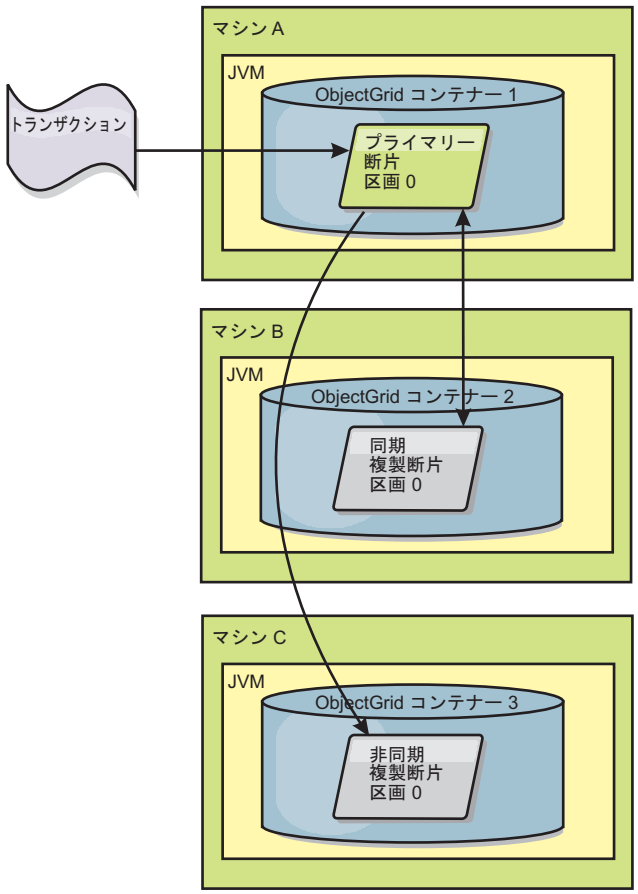


図 34. プライマリー断片と複製断片との間の通信パス

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、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って、断片を割り振ります。

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。

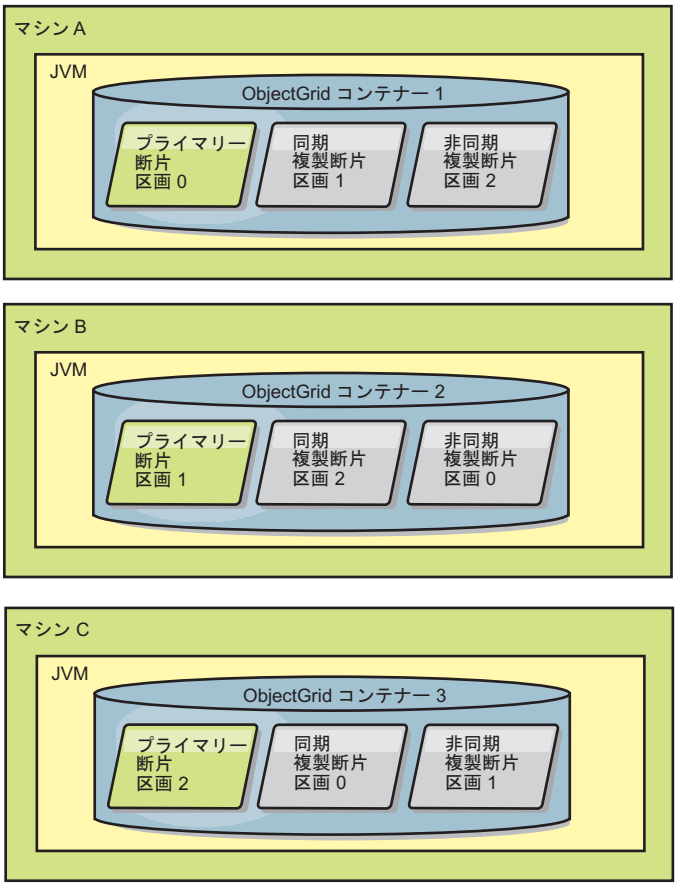


図 35. 区画が 3 つあり、minSyncReplicas 値を 1 に、maxSyncReplicas 値を 1 に、 maxAsyncReplicas 値を 1 にするというデプロイメント方針での ObjectGrid マップ・セットの配置

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。

レプリカ間のロード・バランシング

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。

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。

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、 138 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.
```

非同期複製断片: 非同期複製断片はデータを求めてプライマリーをポーリングします。 レプリカは、プライマリーからデータを受信しないと、それはレプリカがプライマリーに追いついたことを意味するため、自動的にポーリング時間を調整します。 プライマリーに障害が起こったことを示すエラーを受け取った場合、またはネットワークに問題があった場合も、レプリカは調整します。

非同期レプリカが複製を開始すると、非同期レプリカは、以下のメッセージをレプリカ用の SystemOut.log ファイルに出力します。 このメッセージは、1 回の CWOBJ1511 メッセージにつき複数回出力される可能性があります。 レプリカが別のプライマリーに接続する場合、あるいはテンプレート・マップが追加された場合、このメッセージは再度出力されます。

```
CWOBJ1543I: The asynchronous replica objectGridName:mapsetName:partitionNumber started or
Replicating for maps: [mapName]
```

同期複製断片: 同期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。

同期レプリカは、非同期レプリカと同様に、データを求めてポーリングすることでプライマリー断片からデータをコピーします。 同期レプリカは、既存データをプライマリーからコピーする際、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。

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。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。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。このメッセージは、1 回の CWOBJ1511 メッセージにつき複数回出力される場合があります。レプリカが別のプライマリーに接続する場合、あるいはテンプレート・マップが追加された場合、このメッセージは再度出力されます。

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

同期複製断片がピア・モードになっていると、プライマリー断片はすべてのピア・モードの同期レプリカにトランザクションを複製しなければなりません。 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。同期レプリカの最小数または minSync をデプロイメント・ポリシーで設定している場合、同期レプリカの最小数がコミットに賛成してからトランザクションはプライマリーで正常にコミットできます。

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。プライマリーを移動させる必要があり、レプリカがない場合は、一時レプリカを配置して遷移を完了します。 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。非同期レプリカは新規プライマリーに対してポーリングします。

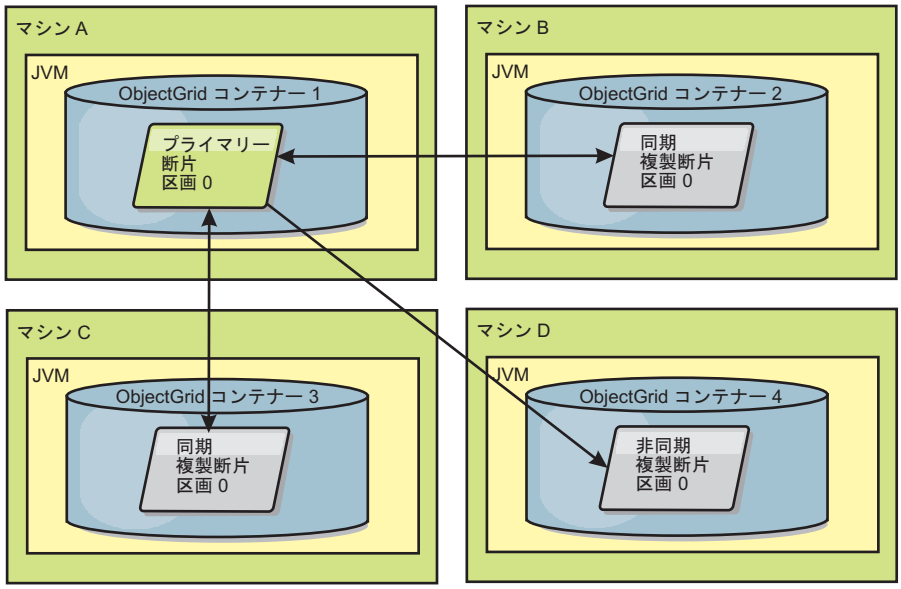


図 36. partition0 区画用の ObjectGrid マップ・セットの配置例。これは、minSyncReplicas 値を 1 に、maxSyncReplicas 値を 2 に、maxAsyncReplicas 値を 1 にするというデプロイメント方針です。

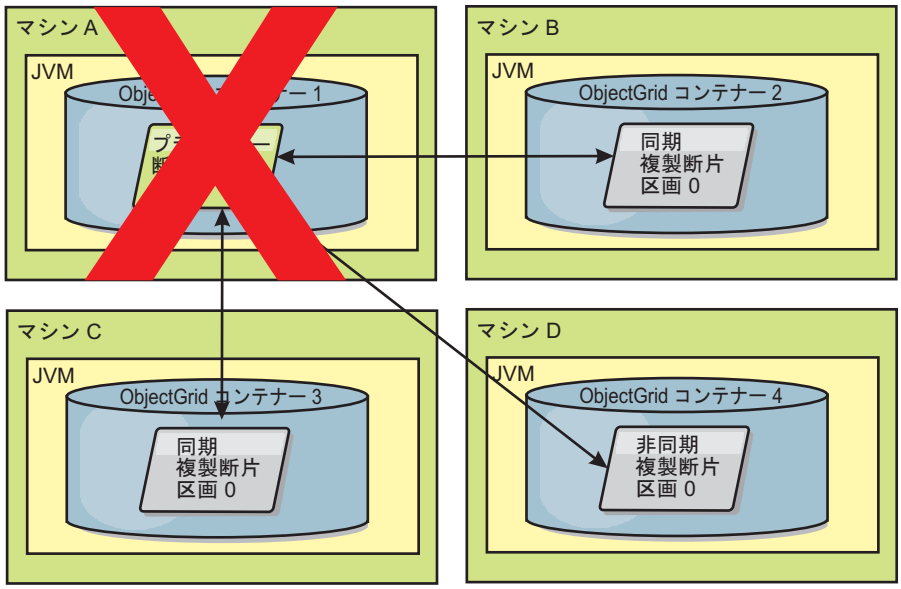


図 37. プライマリー断片のコンテナーに障害が起こる

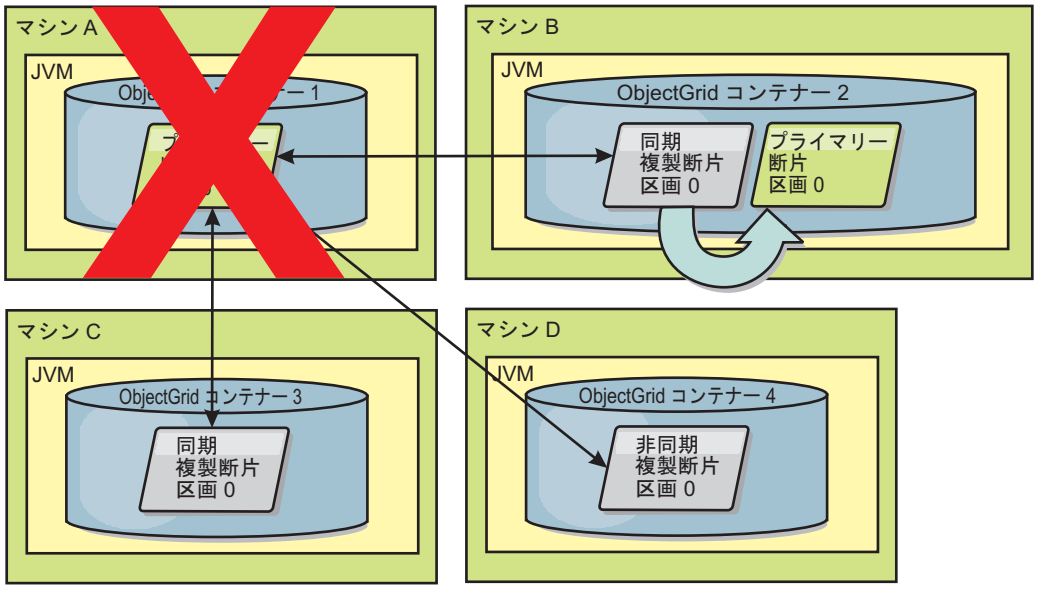


図 38. ObjectGrid コンテナー 2 にある同期複製断片がプライマリー断片になる

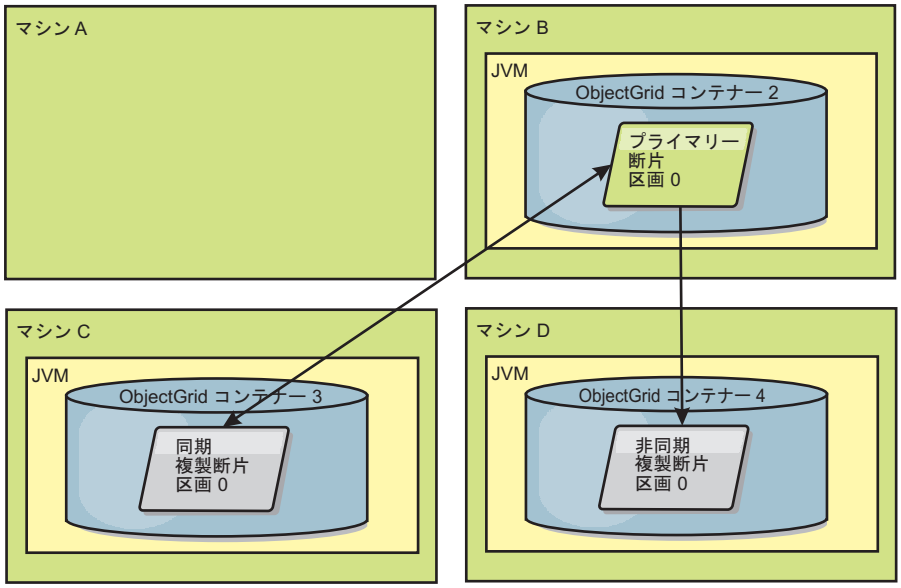


図 39. マシン B にプライマリー断片が含まれています。自動修復モードがどのように設定されているか、およびコンテナーが使用可能かどうかに基づいて、新しい同期複製断片がマシンに配置されるかどうかが決まります。

複製断片のリカバリー

同期複製断片は、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。

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。

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。

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。

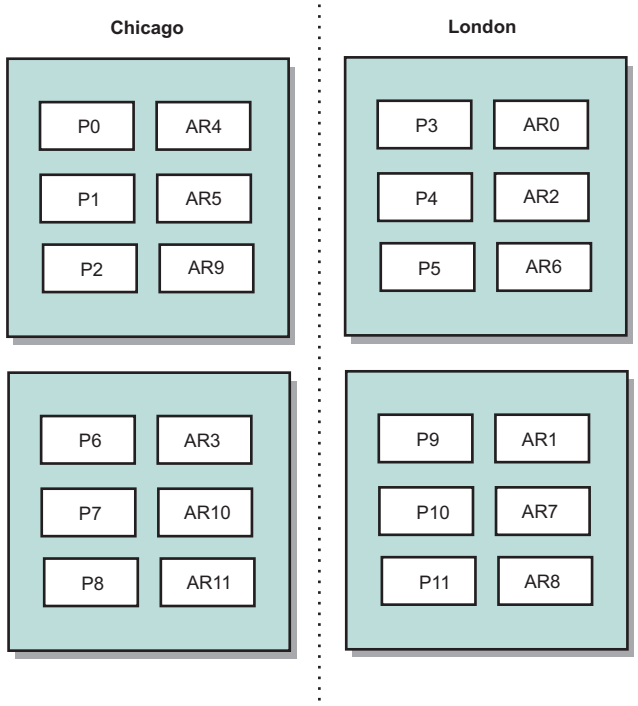


図 40. ゾーン内のプライマリーと複製

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。

マルチ・マスター・グリッド複製トポロジー (AP)

マルチ・マスター非同期複製機能を使用すると、2 つ以上のグリッドを、他のグリッドの正確なミラーにすることができます。 このミラーリングは、非同期複製を使用し、グリッドをまとめて接続するリンク間で実行されます。 各グリッドは完全に独立した「ドメイン」内でホストされ、独自のカタログ・サービス、コンテナー・サーバー、および固有のドメイン・ネームを所有します。 マルチ・マスター非同期複製機能では、これらのドメインの集合を相互接続するリンクを使用し、リンク上の複製を使用してドメインを同期させることができます。eXtreme Scale では、ドメイン間のリンクの定義はユーザーに任されているため、ほとんどのトポロジーを構成できます。

ドメイン: 固有の特性を持つグリッド

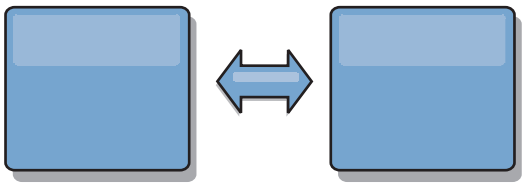
マルチ・マスター複製トポロジーで使用されるグリッドは、*ドメイン*と呼ばれます。各ドメインは、以下の特性を持つ必要があります。

- 固有のドメイン・ネームを持つ専用カタログ・サービスがある
- ドメイン内の他のグリッドと同じグリッド名である
- ドメイン内の他のグリッドと同じ数の区画がある
- FIXED_PARTITION グリッドである (PER_CONTAINER グリッドは複製不可)
-
- ドメイン内の他のグリッドと同じデータ・タイプが複製される
- ドメイン内の他のグリッドと同じマップ・セット名、マップ名、および動的マップ・テンプレートがある

トポロジーのドメインが開始された後、上記の特性を持つ任意のグリッドが複製されます。 グリッドの複製ポリシーは無視されることに注意してください。

ドメインを接続するリンク

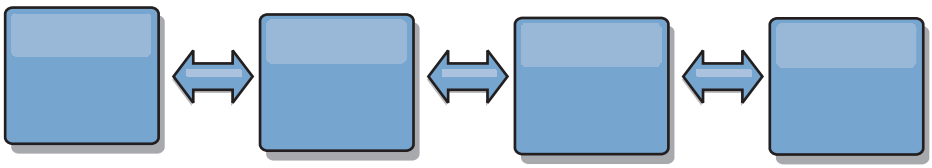
複製グリッドのインフラストラクチャーは、ドメイン間を双方向のリンクで接続したドメインのグラフです。 リンクによって、2 つのドメインはデータ変更を交換することができます。例えば、最も単純なトポロジーはドメイン間に単一のリンクを持つ 1 対のドメインです。 ドメインの名前は、左から「A」で始まって次は「B」というように付けられます。 リンクは、遠距離にわたる広域ネットワーク (WAN) を経由する場合もあります。 リンクが中断した場合は、いずれのドメインでもデータに対する変更を行うことができます。 その変更は後で、リンクがドメインを再接続すると調整されます。 ネットワーク接続が中断されると、リンクは自動的に再接続しようとします。



リンクのセットアップ後、eXtreme Scale は、まずすべてのドメインを同一にしようとします。そして、任意のドメインで変更が行われると同一状態を維持しようとします。 eXtreme Scale の目標は、各ドメインがリンクで接続されたすべての他のドメインの正確なミラーになることです。 ドメイン間の複製リンクは、1 つのドメインで行われたすべての変更を確実に他のドメインにコピーするのに役立ちます。

ライン・トポロジー

ライン・トポロジーは最も単純なトポロジーの 1 つですが、かなりのリンク品質を実証します。 まず、変更を受信するために、ドメインは直接すべての他のドメインに接続する必要はありません。 ドメイン B はドメイン A から変更をプルします。ドメイン C は、ドメイン A と C を接続するドメイン B を介してドメイン A から変更を受信します。同様に、ドメイン D はドメイン C を介して他のドメインから変更を受信します。この機能によって、変更のソースから変更を配布する負荷が分散されます。



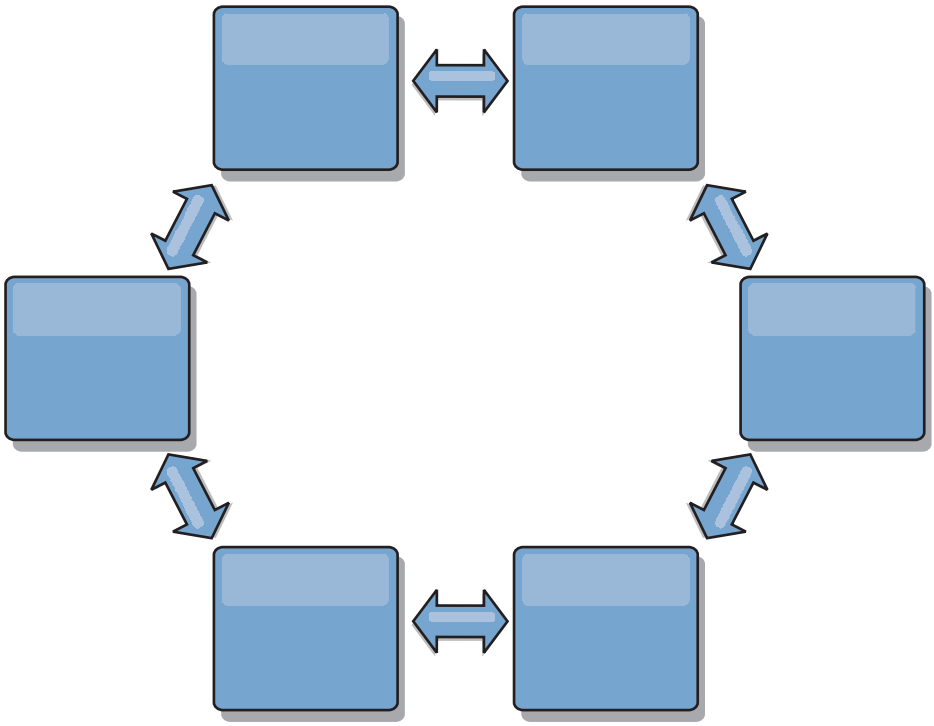
ドメイン C に障害が起こった場合、以下のイベントの発生が考えられることに注意してください。

1. ドメイン D は、ドメイン C が再開されるまで孤立します。
2. ドメイン C は、ドメイン A のコピーであるドメイン B と自分自身を同期させます。
3. ドメイン D は、ドメイン D が孤立していた間 (ドメイン C がダウンしていた間) にドメイン A と B で発生した変更を、ドメイン C を使って自分自身と同期させます。

最後に、ドメイン A、B、C、および D はすべて、他のドメインと再び同一になります。

リング・トポロジー

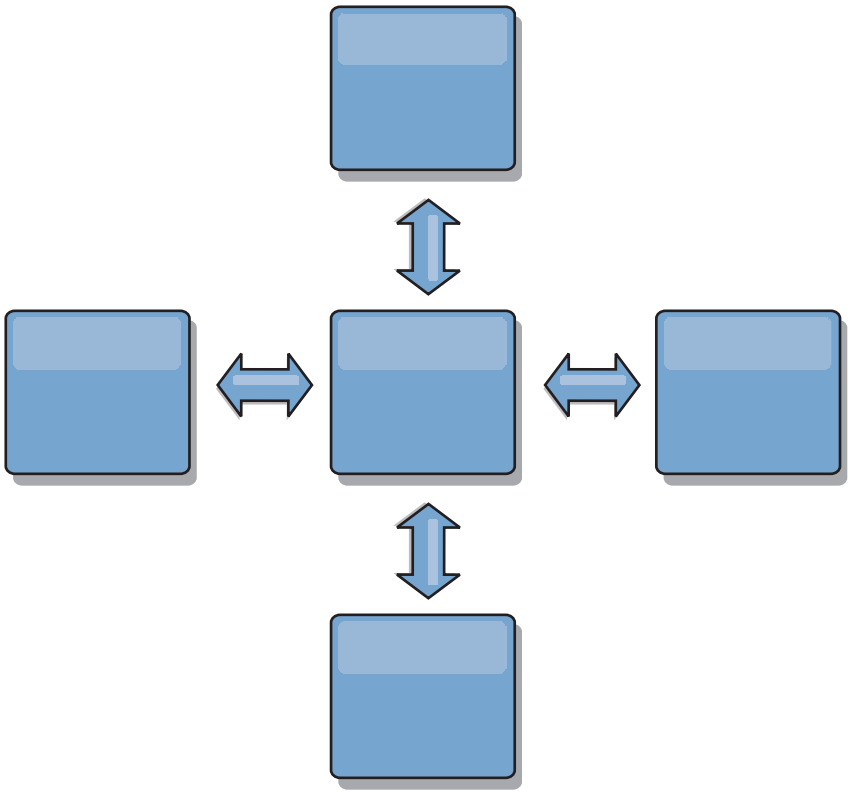
リング・トポロジーは、より回復力のあるトポロジーの例です。 1 つのドメインまたは単一リンクに障害が起こっても、残存しているドメインは、障害を避けてリング内を伝わる変更をそのまま取得できます。 各ドメインには、他のドメインへつながる 2 つのリンクがあります。 リング・トポロジーの大きさには関係なく、各ドメインは最大 2 つのリンクを持ちます。すべてのドメインがすべての変更を見るまで、特定のドメインからの変更をいくつものドメインの間で伝えなければならない場合があるため、変更を伝搬する待ち時間は長くなる可能性があります。 ライン・トポロジーにも同じ問題があります。



リングの中心に置いたルート・ドメインを使った、より洗練されたリング・トポロジーを想像してください。 ルート・ドメインは中央クリアリングハウスとして機能し、他のドメインはルート・ドメインで発生する変更に対してリモート・クリアリングハウスとして機能します。 ルート・ドメインはドメイン間の変更をアービトレーションすることができます。ルート・ドメインを囲む複数のリングがリング・トポロジーに含まれている場合、ルート・ドメインは最も内側にあるリング内のドメイン間の変更のみをアービトレーションすることができます。ただし、アービトレーションの結果は他のリングのドメインにも広がります。

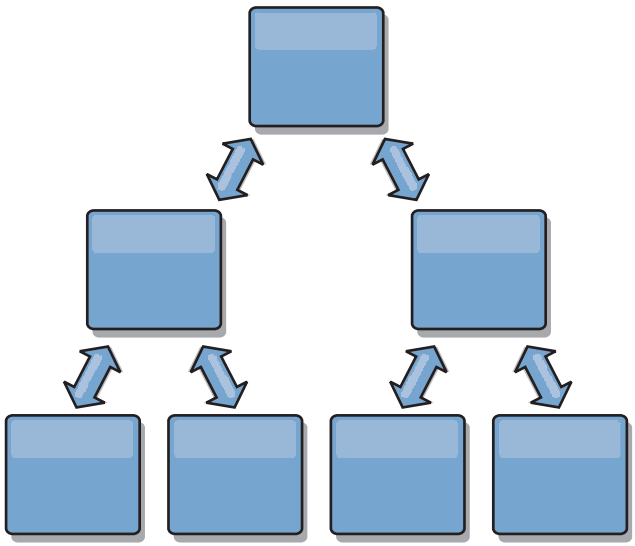
ハブ・アンド・スポーク・トポロジー

ハブ・アンド・スポーク・トポロジーでは、待ち時間が改善されます。したがって、変更は最大 1 つの中間ドメイン (ハブ) に伝わりますが、他の問題が出てきます。 このトポロジーにはハブとして機能する中央ドメインがあります。 この「ハブ・ドメイン」は、リンクを使用してすべての「スポーク・ドメイン」に接続されます。 明らかに、ドメイン間に変更を配布する負担はハブにかかります。 ハブは衝突のクリアリングハウスとして機能し、あるシナリオではセットアップが重要になってきます。 更新速度の速い環境では、ハブは、それについて行けるようにスポークよりも多くのハードウェア上で稼働する必要があります。 eXtreme Scale は、直線的に拡大するように設計されています。つまり、問題なく、必要に応じてハブをさらに大きくすることができます。 ただし、ハブに障害が起こった場合は、変更はハブが再始動するまで配布されません。 スポーク・ドメイン上の変更は、ハブが再接続された後に配布されます。



ツリー・トポロジー

最後のもう 1 つのトポロジーの例は、非循環有向ツリーです。 非循環とは、循環やループがないという意味です。有向とは、リンクが親と子の間にのみ存在するという意味です。この構成は、すべての可能なスポークに中央ハブを接続することが実用的ではないほどドメインをたくさん持つトポロジーの場合、あるいは、ルート・ドメインを更新することなく子ドメインを追加する機能を必要となるトポロジーの場合に役立ちます。



このトポロジーもルート・ドメインに中央クリアリングハウスを持つことができますが、第 2 レベルは、自分より下のドメインで発生する変更に対してリモート・ク

リアリングハウスとして機能することができます。 ルート・ドメインは、第 2 レベルにあるドメイン間の変更のみをアービトレーションすることができます。N 進ツリーも可能です。 N 進ツリーには 各レベルに N 個の子があります。 各ドメインは、N 個ずつ展開します。

トポロジー設計におけるアービトレーションの考慮事項

同じレコードが 2 個所で同時に変更される可能性がある場合には、変更の競合が生じることがあります。各ドメインが、同程度の CPU、メモリー、ネットワーク・リソースを持つようにセットアップしてください。変更の衝突処理 (アービトレーション) を実行しているドメインは、他のドメインよりも多くのリソースを使用することに気付くことがあります。 衝突は、自動的に検出されます。 衝突は、以下の 2 つのうちの 1 つのメカニズムを使って解決されます。

- **デフォルトの衝突アービター。** デフォルトのプロトコルは、字句的に最も小さい名前の付いたドメインからの変更を使用します。 例えば、ドメイン A と B によってレコードの競合が生じる場合には、ドメイン B の変更は無視されます。ドメイン A はそのバージョンを保持し、ドメイン B のレコードはドメイン A からのレコードに一致するように変更されます。これは、ユーザーやセッションが正常にバインドされているアプリケーション、またはユーザーやセッションがグリッドの 1 つにアフィニティーを持つ対象となるアプリケーションにも同様に適用されます。
- **カスタムの衝突アービター。** アプリケーションはカスタム・アービターを提供することができます。ドメインは、衝突を検出するとアービターを呼び出します。「優れた」カスタム・アービターの作成について詳しくは、マルチ・マスター複製用カスタム・アービターの作成を参照してください。

衝突が起こる可能性のあるトポロジーに対しては、ハブ・アンド・スポーク・トポロジーまたはツリー・トポロジーの使用を検討してください。 これらの 2 つのトポロジーは、以下のような場合に発生する可能性のある、エンドレスな衝突の回避につながります。

1. 複数のドメインで衝突が発生します。
2. 各ドメインが衝突をローカルで解決し、改訂を生成します。
3. 改訂が衝突し、その結果、改訂の改訂をもたらします。
4. このように、同期を取ろうとするさまざまなドメイン間に改訂が伝搬していきます。

エンドレスな衝突を回避するには、ドメインのサブセット用衝突ハンドラーとして特定のドメイン - *アービトレーション・ドメイン* – を選択してください。 例えば、ハブ・アンド・スポーク・トポロジーはハブを衝突ハンドラーとして使用する場合があります。 スポーク衝突ハンドラーは、スポーク・ドメインで検出された衝突を無視します。 ハブ・ドメインは改訂を作成し、制御できない衝突改訂を防ぎます。 衝突を処理するように割り当てられたドメインは、衝突の解決に責任を持つすべてのドメインにリンクしていなければなりません。 ツリー・トポロジーでは、内部の親ドメインが自分の直接の子の衝突を解決します。 対照的に、リング・トポロジーを使用すると、リング内の 1 つのドメインをリゾルバーとして指定することはできません。

次の表に、さまざまなトポロジーと互換性のあるアービトレーション・アプローチをまとめました。

表 14. アービトレーション・アプローチ： この表は、アプリケーション・アービトレーションがさまざまなトポロジーと互換性があるかどうかについて記述します。

トポロジー	アプリケーション・アービトレーションとの互換性	注
2 つのドメインのライン	あり	1 つのドメインをアービターとして選択します。
3 つのドメインのライン	あり	真ん中のドメインがアービターでなければなりません。 真ん中のドメインが、単純なハブ・アンド・スポーク・トポロジーのハブだと考えてください。
3 つより多いドメインのライン	なし	アプリケーション・アービトレーションはサポートされません。
N 個のスポークを持つハブ	あり	すべてのスポークへのリンクを持つハブがアービトレーション・ドメインでなければなりません。
N 個のドメインのリング	なし	アプリケーション・アービトレーションはサポートされません。
非循環有向ツリー (N 進ツリー)	あり	すべてのルート・ノードは、自分の直接の子孫のみをアービトレーションする必要があります。

トポロジー設計におけるリンクの考慮事項

変更待ち時間、フォールト・トレランス、およびパフォーマンス特性におけるトレードオフを最適化している間、トポロジーにはリンクの最小数が含まれているのが理想的です。

- **変更待ち時間**

  変更待ち時間は、変更が特定のドメインに到着する前に経由しなければならない中間ドメインの数によって決まります。

  トポロジーが、すべてのドメインを他のすべてのドメインにリンクすることによって中間ドメインを除去すれば、トポロジーの変更待ち時間は最善になります。ただし、ドメインはそのリンク数に比例して複製作業を実行しなければなりません。 大規模トポロジーの場合、非常に多くのリンクが定義され、管理が負担になることが考えられます。

  変更が他のドメインにコピーされる速度は、以下の追加要因によって異なります。

  – ソース・ドメイン上の CPU とネットワーク帯域幅
  – ソース・ドメインとターゲット・ドメインの間の中間ドメイン数とリンク数
  – ソース・ドメイン、ターゲット・ドメイン、および中間ドメインで使用可能な CPU とネットワーク・リソース
- **フォールト・トレランス**

フォールト・トレランスは、変更の複製のために、2 つのドメイン間に存在するパス数によって決定します。

ドメイン間に単一リンクしか存在しない場合、リンクに障害が起こると変更は伝搬されません。 1 つのドメインから別のドメインへの単一リンクが中間ドメインを経由する場合、いずれかの中間ドメインがダウンすると変更は伝搬されません。

3 つのドメイン A、B、および C を持つライン・トポロジーを考えてみます。

```
A <-> B <-> C
```

以下のいくつかの状態のままであれば、ドメイン C は A からの変更はまったく見えません。

– ドメイン A が稼働中でドメイン B がダウン

– A と B の間のリンクがダウン

– B と C の間のリンクがダウン

対照的に、リング・トポロジーでは、各ドメインはいずれかの方向から変更をプルすることができます。

```
A <-> B <-> C <-> A に戻る
```

例えば、ドメイン B がダウンしている場合、ドメイン C は引き続き変更を直接ドメイン A からプルできます。

ハブ・アンド・スポークの設計は、すべての変更がハブを介してプッシュされるので、ダウンしているハブの影響を受けやすくなります。 しかし、単一ドメインは、WAN や物理データ・センターの問題などの、コースに障害が起こる可能性のある完全なフォールト・トレラントなグリッドのままだと覚えていることは価値があります。

- **パフォーマンス**

ドメイン上に定義されるリンク数は、パフォーマンスに影響します。リンクが多いと使われるリソースも多くなり、結果的に複製のパフォーマンスが落ちる場合もあります。 他のドメインを介してドメイン A の変更をプルする機能は、そのトランザクションをどこにでも複製するドメイン A の負荷を効果的に軽減します。 *ドメイン上の変更配布の負荷は、ドメインが使用するリンクの数に制限されます。 トポロジー内にあるドメイン数には関係ありません。* このプロパティーによって、スケーラビリティーが提供され、変更配布の負荷は、単一ドメインに負荷をかけるのではなく、トポロジー内の複数のドメインによって共有することができます。

1 つのドメインが他のドメインを介して間接的に変更をプルできます。 5 つのドメインを持つライン・トポロジーを考えてみます。

```
A <=> B <=> C <=> D <=> E
```

– A は、B、C、D、および E から B を介して変更をプルします。

– B は、A と C からは直接、D と E からは C を介して変更をプルします。

– C は、B と D からは直接、A からは B を介して、E からは D を介して変更をプルします。

– D は、C と E からは直接、A と B からは C を介して変更をプルします。

– E は、D からは直接、A、B、および C からは D を介して変更をプルします。

ドメイン A および E は、それぞれ単一ドメインへのリンクのみを持っているので、配布の負荷は最も低くなります。 ドメイン B、C、および D はそれぞれ 2 つのドメインへのリンクを持っているので、ドメイン B、C、および D 上の配布の負荷は、ドメイン A および E 上の負荷の 2 倍になります。負荷は、トポロジー内の全体のドメイン数ではなく、各ドメインのリンク数によって異なるため、この負荷の分散は、ラインに 1000 ドメインを含んだとしても一定のままです。

パフォーマンスの考慮事項

マルチ・マスター複製トポロジーを使う際は、以下の制限を考慮してください。

- **変更配布の調整** (前述のとおり)
- **複製の待ち時間** (前述のとおり)
- **複製リンクのパフォーマンス** eXtreme Scale は、任意の一対の JVM 間で、単一の TCP/IP ソケットを作成します。それらの JVM 間のトラフィックはすべてそのソケット上で発生し、マルチ・マスター複製も含まれます。 ドメインは少なくとも N 個のコンテナー JVM でホストされ、少なくとも N 個の TCP リンクをピア・ドメインに提供しているため、コンテナー数をより多く持つドメインには、より高い複製のパフォーマンス・レベルがあります。 より多くのコンテナーとは、より多くの CPU とネットワーク・リソースを意味します。
- **TCP スライディング・ウィンドウのチューニングおよび RFC 1323** リンクの両端の RFC 1323 サポートを使用可能にすると、より多くのデータが往復でき、この結果、より高いスループットが実現されます。 この技法は、約 16,000 の要因でウィンドウの容量を拡張します。

  TCP ソケットが、スライディング・ウィンドウのメカニズムを使用して大量データのフローを制御することを思い出してください。これは通常、往復のインターバルのソケットを 64 KB に制限します。 往復のインターバルが 100 ミリ秒の場合、追加チューニングをすることなく帯域幅は 640 KB/秒に制限されます。 リンクで使用可能な帯域幅を完全に使用する場合は、オペレーティング・システムに固有のチューニングが必要になることがあります。 ほとんどのオペレーティング・システムにはチューニング・パラメーターがあり、高度な待ち時間リンクのスループットを向上させる RFC 1323 オプションも含まれます。

  以下の複数の要因が複製のパフォーマンスに影響する可能性があります。

  – eXtreme Scale が変更をプルする速度。
  – eXtreme Scale がプル複製要求をサービスする速度。
  – スライディング・ウィンドウの容量。
  – リンクの両端のネットワーク・バッファーをチューニングすると、eXtreme Scaleは、可能な限り速くソケット上の変更をプルできます。
- **オブジェクト・シリアライゼーション** すべてのデータはシリアライズ可能でなければなりません。 ドメインが COPY_TO_BYTES を使用していない場合、そのドメインは Java のシリアライゼーションまたは ObjectTransformers を使用してシリアライゼーション・パフォーマンスを最適化する必要があります。

- **圧縮** eXtreme Scale は、デフォルトでドメイン間で送信されるすべてのデータを圧縮します。 現行リリースにおいて、圧縮を無効にするオプションはありません。
- **メモリー・チューニング** *マルチ・マスター複製トポロジーのメモリー使用量は、トポロジー内のドメイン数とはほとんど関係ありません。*

  マルチ・マスター複製を使用可能にすると、バージョン管理を扱うマップ・エントリーごとに一定のオーバーヘッドが追加されます。 各コンテナーはトポロジー内の各ドメインの一定量のデータも追跡します。 2 つのドメインを持つトポロジーは、50 ドメインを持つトポロジーとほぼ同じメモリーを使用します。 eXtreme Scale は、その実装環境のリプレイ・ログや類似のキューを使用しません。すなわち、複製リンクがかなりの期間使用できない場合、データ構造のサイズは増大せず、リンクが再始動するときに複製が再開されるのを待ちます。

FIXED_PARTITION の複数データ・センター

現在、複数のデータ・センター間で FIXED_PARTITION グリッドを使用できます。各データ・センターには、マルチ・マスター複製用語で、独自のドメインが必要です。 各データ・センターは、ローカル・ドメインからのデータの読み取り、およびローカル・ドメインへのデータの書き込みができます。 これらの変更は、定義したリンクを使って他のデータ・センターに伝搬されます。

完全複製クライアント

このトポロジー変化には、ハブとして稼働する 1 対の eXtreme Scale サーバーが含まれます。 各クライアントは、クライアント JVM のカタログを使って、必要なものを完備した単一コンテナー・グリッドを作成します。 クライアントは、そのグリッドを使用してハブ・カタログに接続します。これにより、クライアントはハブへの接続を取得すると、すぐにハブと同期するようになります。

クライアントによって行われた変更は、クライアントに対してローカルで、非同期でハブに複製されます。 ハブはアービトレーション・ドメインとして機能し、すべての接続されたクライアントに変更を配布します。完全複製クライアントのトポロジーは、OpenJPA などのオブジェクト・リレーショナル・マッパーに適した L2 キャッシュを提供します。変更はハブを介してクライアント JVM 間に迅速に配布されます。 キャッシュ・サイズをクライアントの使用可能なヒープ・スペース内に含むことができる限り、このトポロジーは L2 のこのスタイルに適したアーキテクチャーです。

必要であれば、複数の区画を使用して、複数の JVM 上にハブ・ドメインを拡張します。 すべてのデータはまだ単一のクライアント JVM に収まらなければならないため、複数の区画を使用してハブの容量を増加させ、変更の配布とアービトレーションを行いますが、単一ドメインの容量は変更しません。

制限

マルチ・マスター複製トポロジーを使うかどうか、およびその使用方法について決定する際は、以下の制限を考慮してください。

- **複数ドメインを使ったクラス・ローダーの構成には気をつけてください**

ドメインは、キーおよび値として使用されるクラスすべてへのアクセス権限を持たなければなりません。すべての依存関係は、すべてのドメインのグリッド・コンテナー JVM に対するすべてのクラスパスに反映されなければなりません。CollisionArbiter プラグインがキャッシュ・エントリーの値を取得する場合、その値に対するクラスはアービターを呼び出すドメインに存在しなければなりません。

- **ローダーの使用はお勧めしません**

  ローダーを使用して、グリッドとデータベースの間の変更をインターフェースで連結することができます。 トポロジー内のすべてのグリッド (ドメイン) が、地理的に同じデータベースに連結されているということは考えられません。 WAN の待ち時間および他の要因で、このユース・ケースが望ましくない状態になる場合があります。

  グリッドのプリロードは、設計に慎重を要する別の問題です。 通常、グリッドは再始動すると、もう一度プリロードされます。 マルチ・マスター複製の使用時は、プリロードは必要ない、または望ましくさえありません。 ドメインは、オンラインになると、すぐに自動的に自分がリンクされているドメインの内容とともに自分自身を再ロードします。 結果的に、マルチ・マスター複製トポロジー内のドメインであるグリッドの手動プリロードを開始する必要はありません。

  ローダーは、通常、挿入規則および更新規則に従います。 マルチ・マスター複製を使用すると、挿入はマージとして扱う必要があります。 ドメインの再始動後にリモート側でデータがプルされている場合、既存データはローカル・ドメインに「挿入」されます。 このデータは既にローカル・データベースにあると考えられるため、標準的な挿入はデータベースの重複キー例外で失敗します。 代わりに、マージ・セマンティクスを使用してください。

  eXtreme Scale を構成し、Loader プラグインでプリロード・メソッドを使用して断片ベースのプリロードを行うことができます。この技法をマルチ・マスター複製トポロジーで使用しないでください。代わりに、トポロジーを始動するとき (最初に) は、クライアント・ベースのプリロードを使用します。トポロジー内の他のドメインに保管されたものの現行コピーを使用して、マルチ・マスター・トポロジーが再始動したドメインをリフレッシュできるようにします。ドメインが開始した後、マルチ・マスター・トポロジーは責任を持ってそれらのドメインを同期させます。

- **EntityManager はサポートされません**

  エンティティー・マップを含むマップ・セットは、ドメインを介して複製されません。

- **バイト配列マップはサポートされません**

  COPY_TO_BYTES で構成されたマップを含むマップ・セットは、ドメインを介して複製されません。

- **後書きはサポートされません**

  後書きサポートで構成されたマップを含むマップ・セットは、ドメインを介して複製されません。

トランザクション変更の配布用 JMS

異なる層間、または混合プラットフォーム上の環境間で、トランザクションの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

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。

複製のためのマップ・セット

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、 97 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。

第 6 章 トランザクション処理の概要

セッションとトランザクションの処理

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。

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

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。

トランザクションの利点

トランザクションを使用するのは、以下の理由からです。

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。
- コミット時に複数の変更をアトミック単位で適用する
- 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・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。

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。

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。

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。

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。

マップ・エントリー・ロック

ObjectGrid BackingMap は、マップに対して複数のロック・ストラテジーをサポートし、キャッシュ・エントリーの整合性を維持します。

各 BackingMap は、以下のロックのストラテジーの 1 つを使用するよう構成できます。

1. オプティミスティック・ロック・モード
2. ペシミスティック・ロック・モード
3. なし

デフォルトのロック・ストラテジーは、OPTIMISTIC です。データの変更が頻繁でない場合は、このオプティミスティック・ロックを使用します。データがキャッシュから読み取られ、トランザクションにコピーされる間、ロックは短期間だけ保持されます。トランザクション・キャッシュがメイン・キャッシュと同期されると、更新されたあらゆるキャッシュ・オブジェクトが元のバージョンに対してチェックされます。チェックが失敗すると、トランザクションはロールバックされ、OptimisticCollisionException 例外となります。

ペシミスティック・ロック・ストラテジーは、キャッシュ・エントリーに対してロックを取得するため、データが頻繁に変更される場合に使用するようにしてください。キャッシュ・エントリーが読み取られる場合は、必ずロックが取得され、トランザクションが完了するまでロックが条件付きで保持されます。ロックによっては、セッションのトランザクション分離レベルを使用して、その期間を調整することができます。

データがまったく更新されないか、静止期間のみに更新されるため、ロックが必要ない場合は、NONE ロック・ストラテジーを使用すれば、ロックを使用不可にすることができます。このストラテジーは、ロック・マネージャーを必要としないため、非常に高速です。NONE ロック・ストラテジーは、ルックアップ表または読み取り専用のマップの場合に理想的です。

ロック・ストラテジーについて詳しくは、*製品概要*のロック・ストラテジーに関する説明を参照してください。

ロック・ストラテジーの指定

以下の例は、map1、map2、および map3 の BackingMap 上にロック・ストラテジーを設定する方法を示しており、各マップは異なるロック・ストラテジーを使用しています。最初のスニペットは、ロック・ストラテジー構成に XML を使用する方法を示し、2 番目のスニペットはプログラマチックな方法を示しています。

XML を用いた方法

BackingMap configuration - XML example

```
<?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="test">
            <backingMap name="map1"
                lockStrategy="PESSIMISTIC" numberOfLockBuckets="31"/>
            <backingMap name="map2"
                lockStrategy="OPTIMISTIC" numberOfLockBuckets="409"/>
            <backingMap name="map3"
                lockStrategy="NONE"/>
        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

プログラマチックな方法

BackingMap configuration - programmatic example

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
...
ObjectGrid og =
 ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("map1");
bm.setLockStrategy( LockStrategy.PESSIMISTIC );
bm.setNumberOfLockBuckets(31);
bm = og.defineMap("map2");
bm.setNumberOfLockBuckets(409);
bm.setLockStrategy( LockStrategy.OPTIMISTIC );
bm = og.defineMap("map3");
bm.setLockStrategy( LockStrategy.NONE );
```

java.lang.IllegalStateException 例外を避けるには、ローカル ObjectGrid インスタンスで initialize メソッドまたは getSession メソッドを使用する前に setLockStrategy メソッドを呼び出す必要があります。

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

ロック・マネージャー構成

ロック・ストラテジーに OPTIMISTIC または PESSIMISTIC が使用されている場合は、BackingMap に対してロック・マネージャーが作成されます。ロック・マネージャーは、ハッシュ・マップを使用して、1 つ以上のトランザクションによってロックされるエントリーを追跡します。 ハッシュ・マップに多くのマップ・エントリーが存在する場合、ロック・バケットが多いほど、パフォーマンスが良好になる可能性が高くなります。バケット数が増えるにつれて、Java 同期の衝突のリスクは下がります。またロック・バケットを増やすことが、並行性の増大につながります。前の例では、特定の BackingMap インスタンスに使用するロック・バケットの数をアプリケーションでどのように設定できるかを示しています。

java.lang.IllegalStateException 例外を避けるには、ObjectGrid インスタンスで initialize メソッドまたは getSession メソッドを呼び出す前に setNumberOfLockBuckets メソッドを呼び出す必要があります。setNumberOfLockBuckets メソッド・パラメーターは、使用するロック・バケットの数を指定する Java プリミティブ整数です。素数を使用すると、ロック・バケット上のマップ・エントリーの一様分布が可能になります。最良のパフォーマンスを得るために適した開始点は、BackingMap エントリーの予想される数のおよそ 10 パーセントにロック・バケットの数を設定することです。

LockDeadlockException

以下は、例外の catch を示すコード例で、結果のメッセージが表示されます。

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

結果は次のとおりです。

```
com.ibm.websphere.objectgrid.plugins.LockDeadlockException: _Message
```

このメッセージは、例外が作成されてスローされるときに、パラメーターとして渡されるストリングを表します。

例外の原因

最も一般的なタイプのデッドロック例外は、ペシミスティック・ロック・ストラテジーを使用しているときに起こり、2 つの別々のクライアントは特定のオブジェクトの共有ロックをそれぞれ所有しています。 その後、どちらのクライアントも、そのオブジェクトの排他ロックへプロモートしようとします。 次の図は、スローされる例外の原因となるトランザクション・ブロックを含めて、このような状況を示しています。

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

これは、例外発生時にプログラム内で何が起こっているかを抽象的に表しています。 同じ ObjectMap を更新するスレッドを多く使用するアプリケーションでは、このような状況が起こる可能性があります。 以下は、前の図で示したように、トランザクション・コード・ブロックを実行している 2 つのクライアントの例です。

考えられる解決策

多数のスレッドが特定のマップでトランザクションを開始すると、図 1 に示されるような状況に遭遇する可能性もあります。 この場合、例外がスローされ、プログラムがハングしないようにします。 自分自身にも通知し、原因をさらに詳しく知るためにコードを catch ブロックに追加することができます。 この例外はペシミスティック・ロック・ストラテジーでのみ見られるため、1 つの簡単な解決策として、単にオプティミスティック・ロック・ストラテジーを使用することが挙げられます。ただし、ペシミスティック・ロック・ストラテジーが必要な場合には、get メソッドの代わりに getForUpdate メソッドを使用することができます。 これにより、前述した状況で例外を受け取ることがなくなります。

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の問題を考慮する必要があります。

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取得し、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" />
```

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・モードを自動的に取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・モードが取得されます。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取得され、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、169 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。

詳しくは、「*プログラミング・ガイド*」のロックの処理に関するトピック、または「*管理ガイド*」のマップ・エントリー・ロックに関するトピックを参照してください。

トランザクション変更の配布用 JMS

異なる層間、または混合プラットフォーム上の環境間で、トランザクションの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

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。

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、基本原則と実装ヒントという 2 つのカテゴリーがあります。基本原則は、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的な原則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エントリー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、通常膨大な数のエントリー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 7 章 セキュリティーの概要

WebSphere eXtreme Scale はデータ・アクセスを保護し、外部セキュリティー・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。

関連タスク

208 ページの『Java SE セキュリティー・チュートリアル: 概要』
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。

第 8 章 REST データ・サービスの概要

WebSphere eXtreme Scale REST データ・サービスは、Microsoft WCF Data Services (正式には ADO.NET Data Services) と互換性があり、Open Data Protocol (OData) を実装する Java HTTP サービスです。Microsoft WCF Data Services は、Visual Studio 2008 SP1 および .NET Framework 3.5 SP1 を使用する場合、この仕様と互換性があります。

互換性の要件

REST データ・サービスは、HTTP クライアントをデータ・グリッドにアクセスできるようにします。REST データ・サービスは、Microsoft .NET Framework 3.5 SP1 で提供される WCF Data Services サポートと互換性があります。Microsoft Visual Studio 2008 SP1 で提供されるリッチ・ツールを使用して RESTful アプリケーションを開発することができます。 この図では、WCF Data Services がクライアントおよびデータベースとどのように対話をするのかについて、概要が示されています。

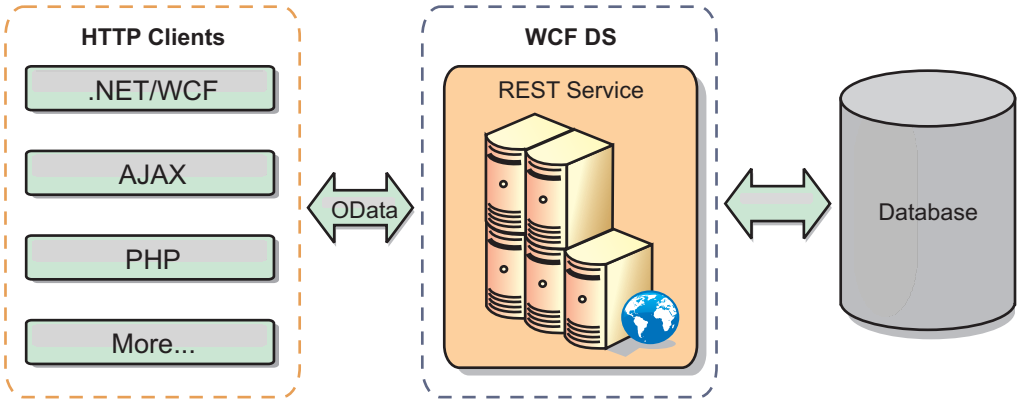


図 41. Microsoft WCF Data Services

WebSphere eXtreme Scale には Java クライアント用の機能の豊富な API セットが含まれています。 次の図で示されているように、REST データ・サービスは HTTP クライアントと WebSphere eXtreme Scale グリッドの間のゲートウェイで、WebSphere eXtreme Scale クライアントを介してグリッドと通信します。REST データ・サービスは Java サーブレットで、これにより、WebSphere Application Server などの共通 Java Platform, Enterprise Edition (JEE) プラットフォームに対する柔軟なデプロイメントが可能です。REST データ・サービスは、WebSphere eXtreme Scale Java API を使用して WebSphere eXtreme Scale グリッドと通信します。WCF Data Services クライアントまたはその他のクライアントが HTTP および XML と通信することができます。

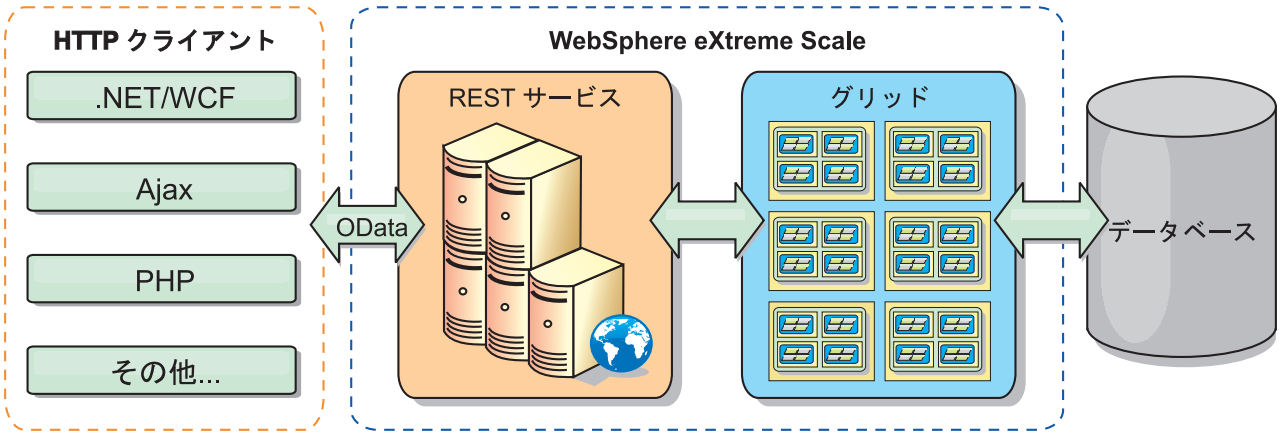


図 42. WebSphere eXtreme Scale REST データ・サービス

WCF Data Services について詳しくは、 229 ページの『REST データ・サービスのサンプルとチュートリアル』 を参照するか、以下のリンクを使用してください。

- MicrosoftWCF Data Services デベロッパー・センター (Microsoft WCF Data Services Developer Center)
- MSDN の ADO.NET Data Services の概要 (ADO.NET Data Services overview on MSDN)
- 「ADO.NET Data Service を使用する」ホワイトペーパー (Whitepaper: Using ADO.NET Data Services)
- Atom Publishing Protocol: データ・サービス URI およびペイロード拡張 (Atom Publish Protocol: Data Services URI and Payload Extensions)
- 概念スキーマ定義ファイル形式 (Conceptual Schema Definition File Format)
- データ・サービス・パッケージング形式のエンティティー・データ・モデル (Entity Data Model for Data Services Packaging Format)
- OData プロトコル (Open Data Protocol)
- OData プロトコルのよくある質問 (Open Data Protocol FAQ)

フィーチャー

このバージョンの eXtreme Scale REST データ・サービスは、以下のフィーチャーをサポートします。

- WCF Data Services エンティティーとしての eXtreme Scale EntityManager API エンティティーの自動モデリングには、以下のサポートが組み込まれます。
  - Java データ型の Entity Data Model 型への変換
  - エンティティー・アソシエーションのサポート
  - 区画に分割されたデータ・グリッドに必要なスキーマ・ルートおよびキー・アソシエーションのサポート

  詳しくは、エンティティー・モデルを参照してください。
- Atom Publishing Protocol (AtomPub または APP) XML および JavaScript™ Object Notation (JSON) データ・ペイロード形式。

- それぞれの HTTP 要求メソッドを使用する作成、読み取り、更新、および削除 (CRUD) 操作である、POST、GET、PUT および DELETE。 さらに、Microsoft 拡張機能の MERGE がサポートされます。
- フィルターを使用した単純照会
- バッチ検索および変更設定要求
- 高可用性のための、区画に分割されたグリッドのサポート
- eXtreme Scale EntityManager API クライアントとのインターオペラビリティー
- 標準 JEE Web サーバーのサポート
- オプティミスティック並行性
- REST データ・サービスと eXtreme Scale データ・グリッドの間のユーザー許可およびユーザー認証

既知の問題と制限

- トンネル要求はサポートされません。

関連タスク

229 ページの『REST データ・サービスのサンプルとチュートリアル』
このトピックでは、WebSphere eXtreme Scale REST データ・サービスの使用を迅速に開始する方法について説明します。 WebSphere Application Server バージョン 7.0、WebSphere Application Server Community Edition、および Apache Tomcat を対象に説明します。

第 9 章 Spring Framework の統合の概要

Spring は、Java アプリケーションの開発によく使用されるフレームワークです。WebSphere eXtreme Scale では、 Spring を使用して eXtreme Scale トランザクションを管理し、デプロイされたメモリー内データ・グリッドに含まれるクライアントおよびサーバーの構成を行うことがサポートされています。

Spring 管理ネイティブ・トランザクション

Spring は、 Java Platform, Enterprise Edition アプリケーション・サーバーに似たコンテナー管理トランザクションを提供します。しかし、Spring メカニズムはさまざまな実装環境でプラグ可能です。WebSphere eXtreme Scale が提供するトランザクション・マネージャー統合は、Spring が ObjectGrid トランザクションのライフサイクルを管理することを可能にします。詳しくは、「*プログラミング・ガイド*」のネイティブ・トランザクションに関する説明を参照してください。

Spring 管理拡張 Bean および名前空間のサポート

また、eXtreme Scale が Spring と統合されることによって、拡張ポイントまたはプラグイン用に Spring スタイルの Bean を定義することが可能になります。この機能によって、拡張ポイントの構成の柔軟性が高まり、洗練された構成ができるようになります。

Spring 管理の拡張 Bean に加えて、 eXtreme Scale は、「objectgrid」という名前の Spring 名前空間を提供します。Bean および組み込みの実装がこの名前空間に事前定義されていて、ユーザーが eXtreme Scale をより簡単に構成できるようになっています。これらのトピックに関する詳しい説明と、Spring 構成を使用して eXtreme Scale コンテナー・サーバーを開始する方法の例については、Spring 拡張 Bean および名前空間のサポートを参照してください。

断片有効範囲サポート

従来のスタイルの Spring 構成では、 ObjectGrid Bean は singleton タイプかプロトタイプ・タイプのどちらかです。ObjectGrid は、「断片」有効範囲と呼ばれる新しい有効範囲もサポートします。Bean が断片有効範囲と定義されている場合、断片当たり 1 つの Bean のみが作成されます。同じ断片内でその Bean 定義に一致する ID を持つ Bean に対する要求はすべて、その 1 つの特定の Bean インスタンスが Spring コンテナーによって戻される結果になります。

以下の例に示す com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl Bean の定義では、有効範囲が断片であると設定されています。したがって、断片当たり、JPAPropFactoryImpl クラスの 1 つのインスタンスのみが作成されます。

```
<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl" scope="shard" />
```

Spring Web Flow

Spring Web Flow は、デフォルトではセッション状態を HTTP セッションに保管します。Web アプリケーションがセッション管理に eXtreme Scale を使用するよう構

成されている場合、この状態を保管するために Spring によってそれが自動的に使用され、セッションと同じ方法でフォールト・トレラントにされます。

パッケージ化

eXtreme Scale Spring 拡張は `ogspring.jar` ファイルに入っています。Spring サポートが正しく機能するためには、この Java アーカイブ (JAR) ファイルがクラスパスになければなりません。 WebSphere Extended Deployment で実行している JEE アプリケーションが WebSphere Application Server Network Deployment を拡張した場合、そのアプリケーションは `spring.jar` ファイルおよびその関連ファイルをエンタープライズ・アーカイブ (EAR) モジュールに入れる必要があります。同じ場所に `ogspring.jar` ファイルも入れる必要があります。

第 10 章 チュートリアル、例、およびサンプル

WebSphere eXtreme Scale のさまざまなチュートリアル、例、およびサンプルが使用できます。

チュートリアル

以下のチュートリアルが現在使用可能です。

- ObjectMap チュートリアル
- 192 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: 概要』
-
-

例

以下のトピックには、WebSphere eXtreme Scale の主要なフィーチャーを示しています。

- 「*プログラミング・ガイド*」にある Data Grid API の例を参照
- 詳しくは、「*管理ガイド*」のローカル・デプロイメントの構成に関する説明を参照してください。

サンプル

WebSphere eXtreme Scale 製品の出荷時には、さまざまな環境における ObjectGrid API の使用法を示すサンプルが付随しています。

チュートリアルおよび例を伴う項目

表 15. フィーチャーごとの使用可能項目

項目	フィーチャー
グリッド対応アプリケーションの作成	ObjectMap API、EntityManager API、照会、エージェント、Java SE および EE、統計、区画化、管理/操作、Eclipse
スケーラブルなグリッド・スタイルのコンピューティングおよびデータ処理	EntityManager API、エージェント
スケーラブルで、回復力のある代替の高性能データベースの作成	ObjectMap API、複製、区画化、管理/操作、Eclipse
WebSphereeXtreme Scale 用の xsadmin の強化	管理
Redbook: User's Guide	すべてのトピック

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

手順

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。

手順

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 192 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、エンティティー・マネージャーで 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 43. 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。

手順

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。

手順

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。

手順

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。

手順

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。

手順

1. 『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. 204 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. 204 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. 207 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。

手順

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

204 ページの『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、202 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。

手順

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。

itemName フィールドにフィルターに掛ける照会が実行されると、照会エンジンは、定義された索引を自動的に使用します。索引を使用することで、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。
次のステップ

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、 204 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。

手順

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

207 ページの『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、204 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。

手順

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: 概要

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

手順

1. 『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. 213 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. 221 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. 225 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。

手順

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

209 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。

手順

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 213 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さまざまな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、「*プログラミング・ガイド*」のクライアント許可リファレンスを参照してください。

WebSphere 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、ObjectMap および JavaMap のメソッドにアクセスするための以下の許可ストリングを定義します。

- 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 ランタイムが、さまざまなメソッドの異なる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、Java 認証・承認サービス (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。

手順

1. **eXtreme Scale 許可を使用可能にします。** ObjectGrid で許可を使用可能にするには、 XML ファイルで、その特定の ObjectGrid の securityEnabled 属性を true に設定する必要があります。 ObjectGrid でセキュリティーを使用可能にするということは、許可を使用可能にするということです。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 ファイルを作成します。

   a. bin ディレクトリーに移動します。

      ```
      cd objectgridRoot/bin
      ```

   b. SimpleApp.xml ファイルを SecureSimpleApp.xml ファイルにコピーします。

      ```
      cp SimpleApp.xml SecureSimpleApp.xml
      ```

   c. SecureSimpleApp.xml ファイルを開いて、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. **許可ポリシーを定義します。** 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) を作成しました。この例では、ユーザー「cashier」はすべてのマップに対する読み取り許可のみを持ち、ユーザー「manager」はすべての許可を持ちます。 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。以下の ファイルは、セキュリティー・ディレクトリーに定義されます。

   ```
   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:

   - codebase "http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction" は、ObjectGrid 用の特別予約 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読み取り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すべての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. **アプリケーションを実行します。**

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。

T

c. 次に、以下のスクリプトを使用して、セキュア・コンテナー・サーバーを始動できます。 bin ディレクトリーから以下のスクリプトを実行します。

- 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 ファイルの代わりに、SecureSimpleApp.xml ファイルを使用します。

- 別の -Djava.security.auth.policy 引数を追加して、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。

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。

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. 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
```

この例外は、ユーザー「cashier」に書き込み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。

次のタスク

チュートリアルの次のステップを完了します。『Java SE セキュリティー・チュートリアル - ステップ 4』を参照してください。

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 221 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。

手順

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 221 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、 213 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。

REST データ・サービスのサンプルとチュートリアル

このトピックでは、WebSphere eXtreme Scale REST データ・サービスの使用を迅速に開始する方法について説明します。 WebSphere Application Server バージョン 7.0、WebSphere Application Server Community Edition、および Apache Tomcat を対象に説明します。

このタスクについて

付属のサンプルには、区画化された eXtreme Scale グリッドを実行するためのソース・コードおよびコンパイルされたバイナリーが含まれています。このサンプルでは、単純なグリッドを作成して、eXtreme Scale エンティティーを使用してデータをモデル化する方法を示します。また、Java または C# を使用したエンティティーの追加および照会を可能にする 2 つのコマンド行クライアント・アプリケーションを示します (図 1 を参照)。

サンプル Java クライアントは、eXtreme Scale Java EntityManager API を使用して、グリッド内のデータを永続化および照会します。このクライアントは、Eclipse またはコマンド行スクリプトを使用して実行できます。なお、サンプル Java クライアントでは REST データ・サービスについては説明されませんが、グリッド内のデータの更新は可能であるため、Web ブラウザーなどのクライアントでデータを読み取ることができます。サンプル Java クライアントおよび Web ブラウザー (図 1) は、REST データ・サービスを使用した HTTP クライアント、および同じ eXtreme Scale グリッドとその中のデータを使用した eXtreme Scale Java クライアントを示します。

サンプル Microsoft WCF Data Services C# クライアントは、.NET Framework を使用した REST データ・サービスを介して、eXtreme Scale グリッドと通信します。WCF Data Services クライアントは、グリッドを更新および照会するために使用で

きます。

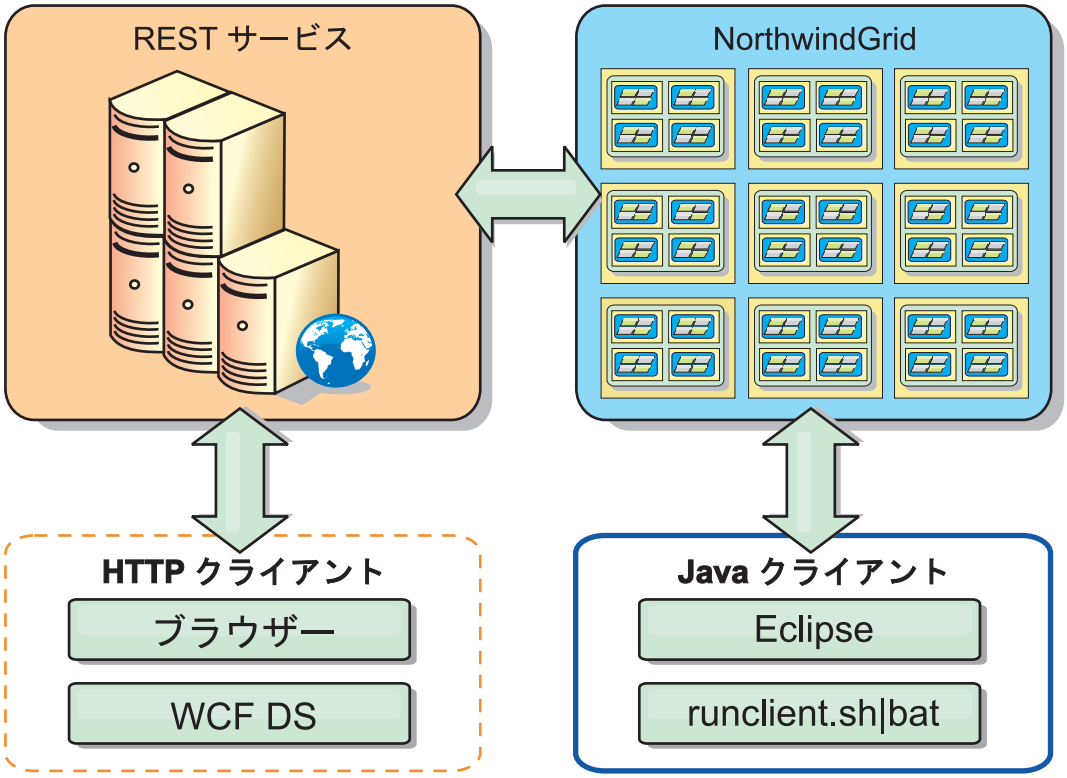


図 44. 開始用 (getting started) サンプル・トポロジー

手順

1. eXtreme Scale グリッドを構成および開始します。 232 ページの『REST データ・サービスの使用可能化』を参照してください。
2. Web ブラウザーで REST データ・サービスを構成および開始します。 243 ページの『REST データ・サービス用のアプリケーション・サーバーの構成』を参照してください。
3. クライアントを実行して、REST データ・サービスと対話します。以下の 2 つのオプションを使用できます。
   a. サンプル Java クライアントを実行して、EntityManager API でグリッドにデータを追加し、Web ブラウザーおよび eXtreme Scale REST データ・サービスでグリッド内のデータを照会します。 254 ページの『REST データ・サービスでの Java クライアントの使用』を参照してください。
   b. サンプル WCF Data Services C# クライアントを実行します。 256 ページの『REST データ・サービスでの Visual Studio 2008 WCF クライアント』を参照してください。

ディレクトリー規則

このトピックでは、*wxs_install_root* や *wxs_home* などの特殊ディレクトリーを参照する必要がある、多くの例およびコマンド行構文について説明します。これらのディレクトリーは以下のように定義されます。

wxs_install_root

wxs_install_root ディレクトリーは、WebSphere eXtreme Scale 製品ファイルがインストールされているルート・ディレクトリーです。これは、試用版の zip ファイルが解凍されたディレクトリー、または eXtreme Scale 製品がインストールされているディレクトリーになる可能性があります。

- 試用版を解凍した場合の例:

  ```
  /opt/IBM/WebSphere/eXtremeScale
  ```

- eXtreme Scale がスタンドアロン・ディレクトリーにインストールされている場合の例:

  ```
  /opt/IBM/eXtremeScale
  ```

- eXtreme Scale が WebSphere Application Server に統合されている場合の例:

  ```
  /opt/IBM/WebSphere/AppServer
  ```

wxs_home

wxs_home ディレクトリーは、WebSphere eXtreme Scale 製品のライブラリー、サンプル、およびコンポーネントのルート・ディレクトリーです。これは、試用版を解凍した場合は、*wxs_install_root* ディレクトリーと同じです。スタンドアロン・インストール済み環境の場合は、これは *wxs_install_root* 内の ObjectGrid サブディレクトリーになります。 WebSphere Application Server に統合されているインストール済み環境の場合は、これは、*wxs_install_root* 内の `optionalLibraries/ObjectGrid` ディレクトリーになります。

- 試用版を解凍した場合の例:

  ```
  /opt/IBM/WebSphere/eXtremeScale
  ```

- eXtreme Scale がスタンドアロン・ディレクトリーにインストールされている場合の例:

  ```
  /opt/IBM/eXtremeScale/ObjectGrid
  ```

- eXtreme Scale が WebSphere Application Server に統合されている場合の例:

  ```
  /opt/IBM/WebSphere/AppServer/optionalLibraries/ObjectGrid
  ```

was_root

was_root ディレクトリーは、WebSphere Application Server インストール済み環境のルート・ディレクトリーです。

```
/opt/IBM/WebSphere/AppServer
```

restservice_home

restservice_home ディレクトリーは、eXtreme Scale REST データ・サービスのライブラリーおよびサンプルが配置されるディレクトリーです。このディレクトリーは *restservice* という名前で、*wxs_home* ディレクトリー内のサブディレクトリーです。

- スタンドアロン・デプロイメントの場合の例:

  ```
  /opt/IBM/WebSphere/eXtremeScale/ObjectGrid/restservice
  ```

- WebSphere Application Server 統合デプロイメントの場合の例:

/opt/IBM/WebSphere/AppServer/optionalLibraries/ObjectGrid/restservice

tomcat_root

tomcat_root は、Apache Tomcat インストール済み環境のルート・ディレクトリーです。

/opt/tomcat5.5

wasce_root

wasce_root は、WebSphere Application Server Community Edition インストール済み環境のルート・ディレクトリーです。

/opt/IBM/WebSphere/AppServerCE

java_home

java_home は、Java Runtime Environment (JRE) インストール済み環境のルート・ディレクトリーです。

/opt/IBM/WebSphere/eXtremeScale/java

REST データ・サービスの使用可能化

REST データ・サービスは、WebSphere eXtreme Scale エンティティー・メタデータを表し、各エンティティーを EntitySet として表すことができます。

サンプル eXtreme Scale グリッドの開始

一般的に、REST データ・サービスを開始する前に、eXtreme Scale グリッドを開始します。以下のステップで、1 つの eXtreme Scale カタログ・サービス・プロセスおよび 2 つのコンテナー・サーバー・プロセスを開始します。

WebSphere eXtreme Scale は、3 つの異なる方法を使用してインストールできます。

- 試用インストール
- スタンドアロン・デプロイメント
- WebSphere Application Server 統合デプロイメント

eXtreme Scale のスケーラブル・データ・モデル

Microsoft Northwind サンプルは、Order Detail 表を使用して多対多のアソシエーションを Order と Product の間で確立します。

ADO.NET Entity Framework および Java Persistence API (JPA) などのオブジェクト・リレーショナル・マッピング仕様 (ORMs) は、エンティティーを使用する表やリレーションシップをマップすることができます。 ただし、このアーキテクチャーは拡張されません。すべてが同一マシン上、あるいはうまく機能するような高価なマシンのクラスター上に存在していなければなりません。

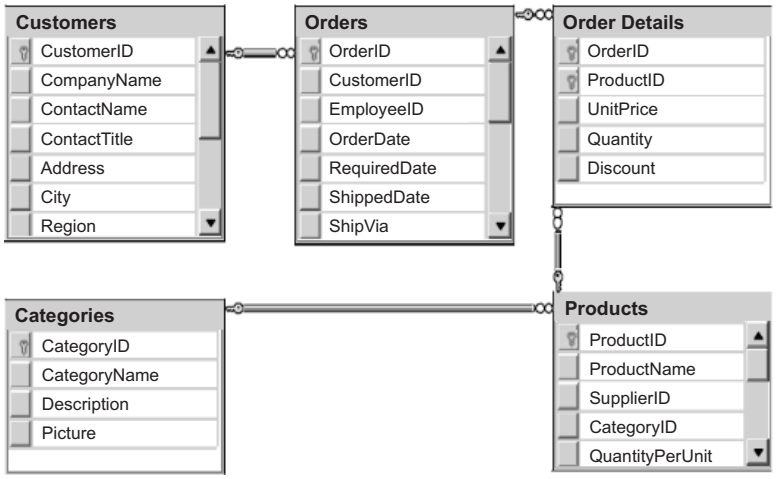


図 45. Microsoft SQL Server Northwind サンプルのスキーマ図

拡張が容易な種類のサンプルを作成するためには、各エンティティーまたは関連エンティティーのグループが単一キーを基にして区画に分割できるように、エンティティーをモデル化する必要があります。単一キーに基づいて区画を作成することで、複数の独立したサーバーに要求を分散させることができます。この構成を実現させるために、エンティティーを 2 つのツリーに分割しました。Customer および Order のツリーと、Product および Category のツリーです。このモデルでは、各ツリーは個別に区画に分割することができるため、異なる速度で、スケーラビリティーを高めながら成長できます。

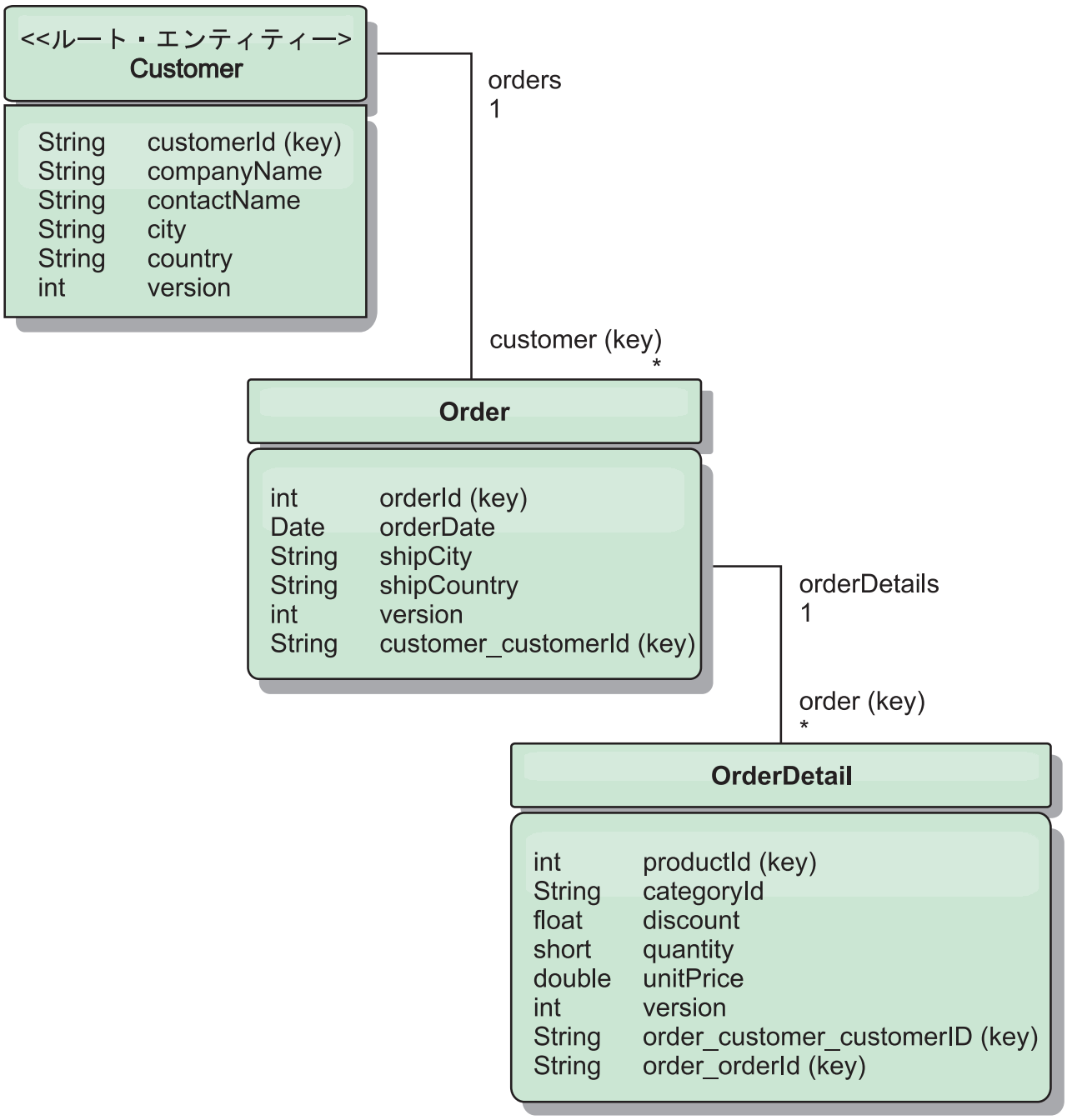


図 46. Customer および Order エンティティーのスキーマ図

例えば、Order と Product はいずれも固有な、別の整数をキーとして持っています。 つまり、Order 表と Product 表は本当にお互いに独立しています。 例えば、オーダー総数とカタログのサイズ (販売する製品数) の影響を考えてみます。 直感的に、多くの製品を持てば多くのオーダーを受けることを意味するようにも思えますが、これは必ずしもそうとは限りません。これが真実であれば、カタログにより多くの製品を追加するだけで、簡単に売り上げを伸ばすことができるでしょう。 オーダーと製品には、それぞれ独自の独立した表があります。 オーダーと製品がそれぞれ独自に別々のデータ・グリッドを持つように、この概念をさらに拡張することができます。独立したグリッドを使用すると、アプリケーションが拡張できるよう

に、各グリッドの個別のサイズの他に、区画数およびサーバー数を制御することができます。 カタログのサイズを 2 倍にすると、製品グリッドを 2 倍にする必要がありますが、オーダー・グリッドはおそらく変わりません。 オーダーの急増、または予測されるオーダーの急増に関しては、その逆が真実となります。

スキーマでは、Customer にはゼロ以上の Order があり、Order は 1 つの特定製品それぞれに明細行 (OrderDetail) があります。 Product は、各 OrderDetail で ID (製品キー) によって識別されます。単一グリッドは、Customer をグリッドのルート・エンティティーとして使用し、Customer、Order、および OrderDetails を保管します。Customer を ID で取得することができますが、Customer ID から始めて Order を取得しなければなりません。そのため、Customer ID は Order にそのキーの一部として追加されます。同様に、カスタマー ID およびオーダー ID は OrderDetail ID の一部です。

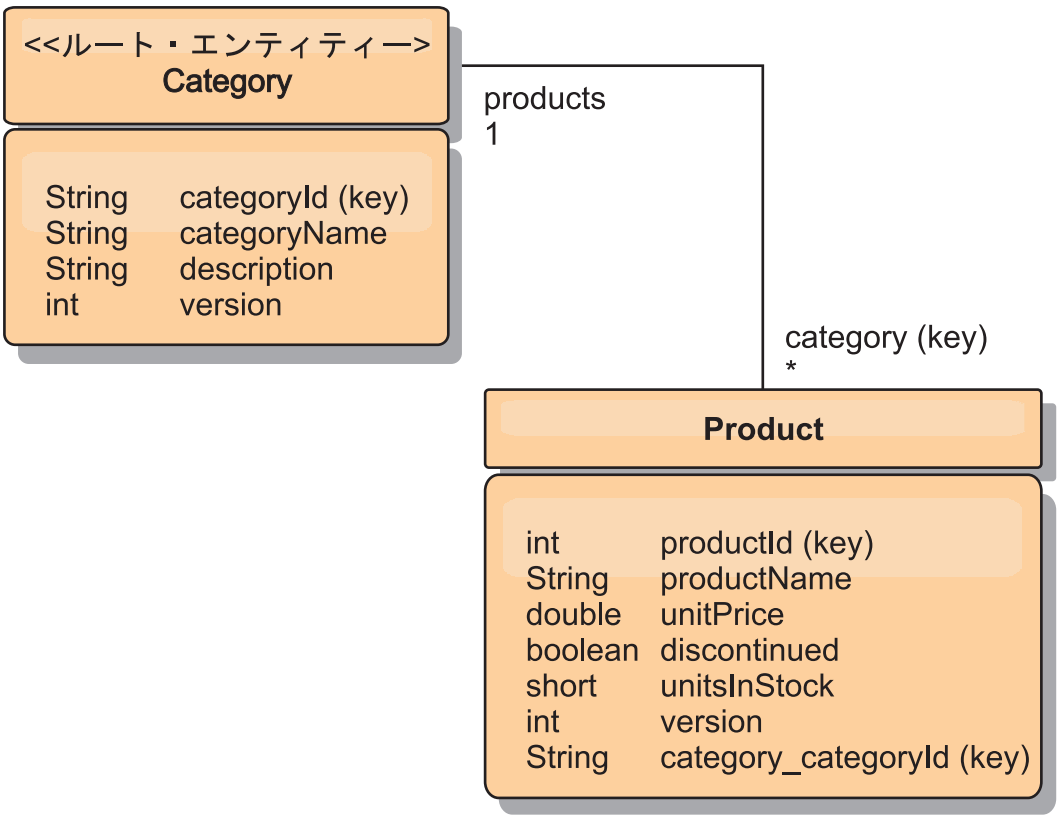


図 47. Category および Product エンティティーのスキーマ図

Category および Product スキーマでは、Category がスキーマ・ルートです。 このスキーマにより、カスタマーはカテゴリーごとに製品を照会することができます。キー・アソシエーションおよびその重要性のさらに詳細な情報については、『REST を使用したデータの取得および更新』 を参照してください。

REST を使用したデータの取得および更新

OData プロトコルは、すべてのエンティティーが正規化形式でアドレス指定できることを要求します。 つまり、各エンティティーは区画に分割されたルート・エンティティー (スキーマ・ルート) のキーを含んでいなければなりません。

以下は、子エンティティーをアドレス指定するためのルート・エンティティーからのアソシエーションの使用方法の例です。

```
/Customer('ACME')/order(100)
```

WCF Data Services では、子エンティティーは直接アドレス可能でなければなりません。つまり、スキーマ・ルートのキーは、次のように子のキーの一部でなければなりません。/Order(customer_customerId='ACME', orderId=100) これは、ルート・エンティティーへのアソシエーションの作成により実現され、ルート・エンティティーへの 1 対 1 または多対 1 のアソシエーションもキーとして識別されます。 エンティティーがキーの一部として組み込まれる場合、親エンティティーの属性はキー・プロパティーとして公開されます。

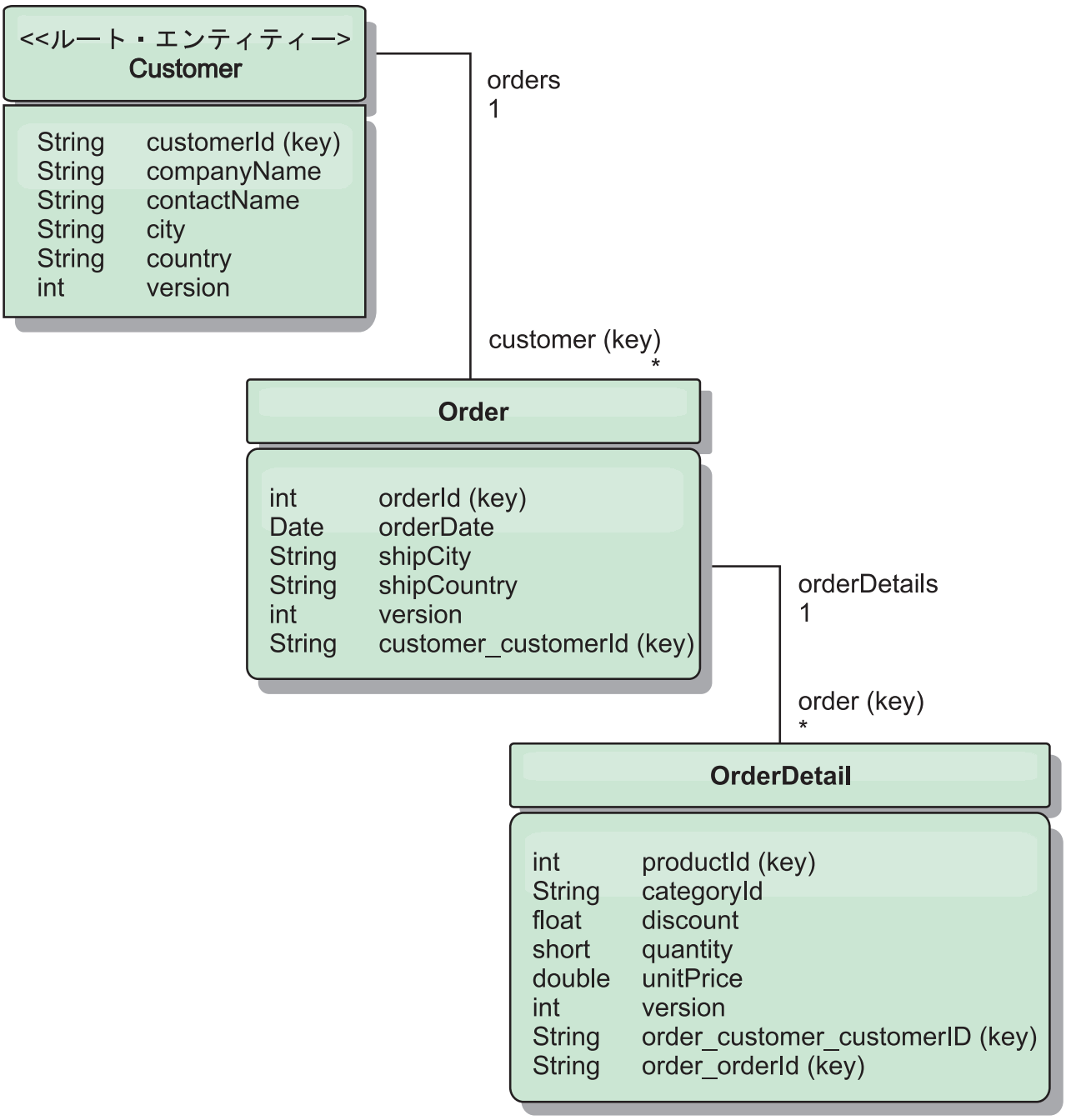


図 48. Customer および Order エンティティーのスキーマ図

Customer/Order エンティティーのスキーマ図で、どのように各エンティティーが Customer を使用して区画に分割されているかを示しています。Order エンティティーにはキーの一部として Customer が組み込まれるため、直接アクセスすることができます。 REST データ・サービスは、すべてのキー・アソシエーションを個別のプロパティーとして公開します。Order には customer_customerId があり、OrderDetail には order_customer_customerId および order_orderId があります。

EntityManager API を使用すると、Customer とオーダー ID を使用して Order を検索することができます。

```
transaction.begin();
// Look-up the Order using the Customer.  We only include the Id
// in the Customer class when building the OrderId key instance.
Order order = (Order) em.find(Order.class,
    new OrderId(100, new Customer('ACME')));
...
transaction.commit();
```

REST データ・サービスを使用する場合、どちらかの URL を使用して Order を取得することができます。

- `/Order(orderId=100, customer_customerId='ACME')`
- `/Customer('ACME')/orders?$filter=orderId eq 100`

カスタマー・キーは Customer エンティティーの属性名 (下線文字とカスタマー ID の属性名 customer_customerId) を使用してアドレス指定されています。

エンティティーには、非ルート・エンティティーのすべての上位エンティティーがルートへのキー・アソシエーションを持つ場合は、キーの一部として非ルート・エンティティーを組み込むこともできます。 この例では、OrderDetail には Order へのキー・アソシエーションがあり、Order にはルートの Customer エンティティーへのキー・アソシエーションがあります。 EntityManager API の使用は、次のとおりです。

```
transaction.begin();
// Construct an OrderDetailId key instance.  It includes
// The Order and Customer with only the keys set.
Customer customerACME = new Customer("ACME");
Order order100 = new Order(100, customerACME);
OrderDetailId orderDetailKey =
    new OrderDetailId(order100, "COMP");
OrderDetail orderDetail = (OrderDetail)
    em.find(OrderDetail.class, orderDetailKey);
...
```

REST データ・サービスは、OrderDetail に直接アドレスを指定することができます。

```
/OrderDetail(productId=500, order_customer_customerId='ACME', order_orderId =100)
```

OrderDetail エンティティーから Product エンティティーへのアソシエーションは分割され、Orders および Product インベントリーを個別に区画に分割することができます。 OrderDetail エンティティーは、強いリレーションシップの代わりにカテゴリーと製品 ID を保管します。 2 つのエンティティー・スキーマを切り離すと、一度に 1 つの区画だけがアクセスされます。

Category および Product スキーマは、図で示すように、ルート・エンティティーが Category で、各 Product には Category エンティティーへのアソシエーションがあることを表しています。Category エンティティーは Product ID に組み込まれています。 REST データ・サービスは、キー・プロパティーを公開します。 category_categoryId で Product に直接アドレスを指定できます。

Category はルート・エンティティーなので、区画に分割された環境で Product を検索するには、Category が認識されている必要があります。EntityManager API を使用すると、Product の検索前にトランザクションは Category エンティティーに pinned されなければなりません。

EntityManager API の使用は、次のとおりです。

```
transaction.begin();
// Create the Category root entity with only the key.  This
// allows us to construct a ProductId without needing to find
// The Category first.  The transaction is now pinned to the
// partition where Category "COMP" is stored.
Category cat = new Category("COMP");
Product product = (Product) em.find(Product.class,
    new ProductId(500, cat));
...
```

REST データ・サービスは、Product に直接アドレスを指定することができます。

```
/Product(productId=500, category_categoryId='COMP')
```

REST データ・サービスのスタンドアロン・グリッドの開始

以下のステップに従って、スタンドアロン eXtreme Scale デプロイメントの WebSphere eXtreme Scale REST サービス・サンプル・グリッドを開始します。

始める前に

以下のように、WebSphere eXtreme Scale の試用版または完全な製品をインストールします。

- スタンドアロン・バージョンの WebSphere eXtreme Scale 7.1 製品を インストールして、後続フィックスがある場合にはすべて適用します。
- WebSphere eXtreme Scale REST データ・サービスが含まれている WebSphere eXtreme Scale バージョン 7.1 の試用版をダウンロードして解凍します。

このタスクについて

WebSphere eXtreme Scale サンプル・グリッドを開始します。

手順

1. カタログ・サービス・プロセスを開始します。コマンド行または端末ウィンドウを開いて、以下のように、JAVA_HOME 環境変数を設定します。
   - Linux UNIX `export JAVA_HOME=java_home`
   - Windows `set JAVA_HOME=java_home`
2. `cd restservice_home/gettingstarted`
3. カタログ・サービス・プロセスを開始します。 eXtreme Scale セキュリティーなしで サービスを開始するには、以下のコマンドを使用します。
   - Linux UNIX `./runcat.sh`
   - Windows `runcat.bat`

   eXtreme Scale セキュリティーありでサービスを開始するには、以下のコマンドを使用します。
   - Linux UNIX `./runcat_secure.sh`
   - Windows `runcat_secure.bat`

4. 以下のように、2 つのコンテナー・サーバー・プロセスを開始します。もう 1 つコマンド行または端末ウィンドウを開いて、以下のように、JAVA_HOME 環境変数を設定します。
   - Linux UNIX `export JAVA_HOME=java_home`
   - Windows `set JAVA_HOME=java_home`
5. `cd restservice_home/gettingstarted`
6. 以下のように、コンテナー・サーバー・プロセスを開始します。

   eXtreme Scale セキュリティーなしでサーバーを始動するには、以下のコマンドを使用します。
   - Linux UNIX `./runcontainer.sh container0`
   - Windows `runcontainer.bat container0`

   eXtreme Scale セキュリティーありでサーバーを始動するには、以下のコマンドを使用します。
   - Linux UNIX `./runcontainer_secure.sh container0`
   - Windows `runcontainer_secure.bat container0`
7. もう 1 つコマンド行または端末ウィンドウを開いて、以下のように、JAVA_HOME 環境変数を設定します。
   - Linux UNIX `export JAVA_HOME=java_home`
   - Windows `set JAVA_HOME=java_home`
8. `cd restservice_home/gettingstarted`
9. 以下のように、2 番目のコンテナー・サーバー・プロセスを開始します。

   eXtreme Scale セキュリティーなしでサーバーを始動するには、以下のコマンドを使用します。
   - Linux UNIX `./runcontainer.sh container1`
   - Windows `runcontainer.bat container1`

   eXtreme Scale セキュリティーありでサーバーを始動するには、以下のコマンドを使用します。
   - Linux UNIX `./runcontainer_secure.sh container1`
   - Windows `runcontainer_secure.bat container1`

タスクの結果

eXtreme Scale コンテナーの準備ができるまで待機してから、次のステップに進みます。以下のメッセージが端末ウィンドウに表示されると、コンテナー・サーバーの準備ができています。

`CWOBJ1001I: ObjectGrid サーバー container_name は要求を処理する用意ができています。`

ここで、*container_name* は、開始したコンテナーの名前です。

WebSphere Application Server での REST データ・サービス・グリッドの開始

以下のステップに従って、WebSphere Application Server に統合された WebSphere eXtreme Scale デプロイメントのスタンドアロン WebSphere eXtreme Scale REST サービス・サンプル・データ・グリッドを開始します。WebSphere eXtreme Scale は WebSphere Application Server に統合されていますが、以下のステップでは、スタンドアロン WebSphere eXtreme Scale カタログ・サービス・プロセスおよびコンテナーが開始されます。

始める前に

(セキュリティーを使用不可にして) WebSphere eXtreme Scale バージョン 7.1 製品を WebSphere Application Server バージョン 7.0.0.5 以上のインストール・ディレクトリーにインストールし、少なくとも 1 つの Application Server プロファイルを拡張します。

このタスクについて

WebSphere eXtreme Scale サンプル・グリッドを開始します。

手順

1. カタログ・サービス・プロセスを開始します。コマンド行または端末ウィンドウを開いて、以下のように、JAVA_HOME 環境変数を設定します。
   - Linux UNIX `export JAVA_HOME=java_home`
   - Windows `set JAVA_HOME=java_home`

   ```
   cd restservice_home/gettingstarted
   ```
2. カタログ・サービス・プロセスを開始します。

   eXtreme Scale セキュリティーなしでサーバーを始動するには、以下のコマンドを使用します。
   - Linux UNIX `./runcat.sh`
   - Windows `runcat.bat`

   eXtreme Scale セキュリティーありでサーバーを始動するには、以下のコマンドを使用します。
   - Linux UNIX `./runcat_secure.sh`
   - Windows `runcat_secure.bat`
3. 以下のように、2 つのコンテナー・サーバー・プロセスを開始します。もう 1 つコマンド行または端末ウィンドウを開いて、以下のように、JAVA_HOME 環境変数を設定します。
   - Linux UNIX `export JAVA_HOME=java_home`
   - Windows `set JAVA_HOME=java_home`

4. 以下のように、コンテナー・サーバー・プロセスを開始します。

   eXtreme Scale セキュリティーなしでサーバーを始動するには、以下のコマンドを使用します。

   a. コマンド行ウィンドウを開きます。
   b. `cd restservice_home/gettingstarted`
   c. eXtreme Scale セキュリティーなしで サーバーを始動するには、以下のコマンドを使用します。
      - Linux UNIX `./runcontainer.sh container0`
      - Windows `runcontainer.bat container0`
   d. eXtreme Scale セキュリティーありでサーバーを始動するには、以下のコマンドを使用します。
      - Linux UNIX `./runcontainer_secure.sh container0`
      - Windows `runcontainer_secure.bat container0`

5. 以下のように、2 番目のコンテナー・サーバー・プロセスを開始します。
   a. コマンド行ウィンドウを開きます。
   b. `cd restservice_home/gettingstarted`
   c. eXtreme Scale セキュリティーなしで サーバーを始動するには、以下のコマンドを使用します。
      - Linux UNIX `./runcontainer.sh container1`
      - Windows `runcontainer.bat container1`
   d. eXtreme Scale セキュリティーありで サーバーを始動するには、以下のコマンドを使用します。
      - Linux UNIX `./runcontainer_secure.sh container1`
      - Windows `runcontainer_secure.bat container1`

タスクの結果

コンテナー・サーバーの準備ができるまで待機してから、次のステップに進みます。以下のメッセージが表示されると、コンテナー・サーバーの準備ができています。

`CWOBJ1001I: ObjectGrid サーバー` *container_name* `は要求を処理する用意ができています。`

ここで、*container_name* は前のステップで開始したコンテナーの名前です。

REST データ・サービス用のアプリケーション・サーバーの構成

WebSphere Application Server バージョン 7.0 での REST データ・サービスの開始

このトピックでは、WebSphere Application Server バージョン 7.0 を使用して eXtreme Scale REST データ・サービスを構成および開始する方法について説明します。

始める前に

サンプル eXtreme Scale グリッドが開始されていることを確認します。グリッドの開始方法の詳細については、232 ページの『REST データ・サービスの使用可能化』を参照してください。

手順

1. WebSphere Application Server for Developers バージョン 7.0 をダウンロードしてインストールします。

   制約事項: セキュリティーは使用可能にしないでください。
2. WebSphere Application Server バージョン 7.0 フィックスパック 5 以上をダウンロードしてインストールします。
3. 以下のように、WebSphere eXtreme Scale クライアント・ランタイム JAR の `wsogclient.jar` ファイルおよび REST データ・サービス構成 JAR またはディレクトリーをアプリケーション・サーバー・クラスパスに追加します。
   a. WebSphere Application Server 管理コンソールを開きます。
   b. 「**環境**」 → 「**共有ライブラリー**」にナビゲートします。
   c. 「**新規**」をクリックします。
   d. 以下の項目をフィールドに追加します。
      1) 名前: extremescale_client_v71
      2) クラスパス: `wxs_home/lib/wsogclient.jar`
   e. 「**OK**」をクリックします。
   f. 「**新規**」をクリックします。
   g. 以下の項目を該当するフィールドに追加します。
      1) 名前: `extremescale_gettingstarted_config`
      2) クラスパス:
         - `restservice_home/gettingstarted/restclient/bin`
         - `restservice_home/gettingstarted/common/bin`

      要確認: それぞれのパスを別個の行に追加します。
   h. 「**OK**」をクリックします。
   i. 変更をマスター構成に保存します。
4. 管理コンソールを使用して、REST データ・サービスの EAR ファイル `wxsrestservice.ear` を WebSphere Application Server にインストールします。
   a. WebSphere Application Server 管理コンソールを開きます。
   b. 「**アプリケーション**」 → 「**新規アプリケーション**」にナビゲートします。

c. restservice_home/lib/wxsrestservice.ear を参照して、ファイルを選択し、「**次へ**」をクリックします。

d. 詳細インストール・オプションを選択して、「**次へ**」をクリックします。

e. アプリケーション・セキュリティー警告画面で、「**続行**」をクリックします。

f. デフォルト・インストール・オプションを選択して、「**次へ**」をクリックします。

g. アプリケーションのマップ先サーバーを選択して、「**次へ**」をクリックします。

h. JSP 再ロード・ページで、デフォルトを使用し、「**次へ**」をクリックします。

i. 共有ライブラリーページで、wxsrestservice.war モジュールを、以下の定義された共有ライブラリーにマップします。

- extremescale_client_v71
- extremescale_ gettingstarted _config

j. マップ共有ライブラリー・リレーションシップ・ページで、デフォルトを使用し、「**次へ**」をクリックします。

k. マップ仮想ホスト・ページで、デフォルトを使用し、「**次へ**」をクリックします。

l. マップ・コンテキスト・ルート・ページで、コンテキスト・ルートを wxsrestservice に設定します。

m. 要約画面で、「**終了**」をクリックして、インストールを完了します。

n. 変更をマスター構成に保存します。

5. アプリケーション・サーバーおよび wxsrestservice eXtreme Scale REST データ・サーバー・アプリケーションを開始します。 アプリケーションの開始後に、アプリケーション・サーバーの SystemOut.log ファイルを確認して、以下のメッセージが表示されていることを確認します。CWOBJ4000I: WebSphere eXtreme Scale REST データ・サービスが開始されました。

6. 以下のように、REST データ・サービスが動作していることを確認します。

a. ブラウザーを開いて、http://localhost:9080/wxsrestservice/restservice/NorthwindGrid にナビゲートします。NorthwindGrid のサービス文書が表示されます。

b. http://localhost:9080/wxsrestservice/restservice/NorthwindGrid/$metadata にナビゲートします。Entity Model Data Extensions (EDMX) 文書が表示されます。

7. グリッド・プロセスを停止するには、それぞれのコマンド・ウィンドウで CTRL+C を使用します。

WebSphere Application Server 7.0 に統合された WebSphere eXtreme Scale での REST データ・サービスの開始

このトピックでは、WebSphere eXtreme Scale で統合および拡張されている WebSphere Application Server バージョン 7.0 を使用して eXtreme Scale REST データ・サービスを構成および開始する方法について説明します。

始める前に

サンプル・スタンドアロン eXtreme Scale グリッドが開始されていることを確認します。グリッドの開始方法の詳細については、232 ページの『REST データ・サービスの使用可能化』を参照してください。

このタスクについて

WebSphere Application Server を使用して WebSphere eXtreme Scale REST データ・サービスを開始するには、以下のステップに従います。

手順

1. 以下のように、WebSphere eXtreme Scale REST データ・サービスのサンプル構成 JAR をクラスパスに追加します。
   a. WebSphere 管理コンソールを開きます。
   b. 「環境」->「共有ライブラリー」にナビゲートします。
   c. 「新規」をクリックします。
   d. 以下の項目を該当するフィールドに追加します。
      1) 名前: extremescale_gettingstarted_config
      2) クラスパス
         - `restservice_home/gettingstarted/restclient/bin`
         - `restservice_home/gettingstarted/common/bin`

         要確認: 各パスは異なる行に記述する必要があります。
   e. 「**OK**」をクリックします。
   f. 変更をマスター構成に保存します。
2. WebSphere 管理コンソールを使用して、 REST データ・サービスの EAR ファイル wxsrestservice.ear を WebSphere Application Server にインストールします。
   a. WebSphere 管理コンソールを開きます。
   b. 「アプリケーション」->「新規アプリケーション」にナビゲートします。
   c. ファイル・システム上の `restservice_home/lib/wxsrestservice.ear` ファイルを参照します。ファイルを選択して、「**次へ**」をクリックします。
   d. 詳細インストール・オプションを選択して、「**次へ**」をクリックします。
   e. アプリケーション・セキュリティー警告画面で、「**続行**」をクリックします。
   f. デフォルト・インストール・オプションを選択して、「**次へ**」をクリックします。
   g. wxsrestservice.war モジュールのマップ先サーバーを選択して、「**次へ**」をクリックします。
   h. JSP 再ロード・ページで、デフォルトを使用し、「**次へ**」をクリックします。
   i. 共有ライブラリー・ページで、「wxsrestservice.war」モジュールを、ステップ 1 で定義した共有ライブラリー `extremescale_ gettingstarted _config` にマップします。

j. マップ共有ライブラリー・リレーションシップ・ページで、デフォルトを使用し、「**次へ**」をクリックします。

k. マップ仮想ホスト・ページで、デフォルトを使用し、「**次へ**」をクリックします。

l. マップ・コンテキスト・ルート・ページで、コンテキスト・ルートを wxsrestservice に設定します。

m. 要約画面で、「**終了**」をクリックして、インストールを完了します。

n. 変更をマスター構成に保存します。

3. eXtreme Scale セキュリティーを使用可能にして eXtreme Scale グリッドを始動した場合には、 `restservice_home/gettingstarted/restclient/bin/wxsRestService.properties` ファイルで以下のプロパティーを設定します。

```
ogClientPropertyFile=restservice_home/gettingstarted/security/security.ogclient.properties
```

4. アプリケーション・サーバーおよび「wxsrestservice」eXtreme Scale REST データ・サーバー・アプリケーションを開始します。

   アプリケーションの開始後に、アプリケーション・サーバーの SystemOut.log を確認して、以下のメッセージが表示されていることを確認します。`CWOBJ4000I: WebSphere eXtreme Scale REST データ・サービスが開始されました。`

5. 以下のように、REST データ・サービスが動作していることを確認します。

   a. ブラウザーを開いて、以下にナビゲートします。

      http://localhost:9080/wxsrestservice/restservice/NorthwindGrid

      NorthwindGrid のサービス文書が表示されます。

   b. 以下にナビゲートします。

      http://localhost:9080/wxsrestservice/restservice/NorthwindGrid/$metadata

      Entity Model Data Extensions (EDMX) 文書が表示されます。

6. グリッド・プロセスを停止するには、それぞれのコマンド・ウィンドウで CTRL+C を使用して、プロセスを停止します。

WebSphere Application Server Community Edition での REST データ・サービスの開始

このトピックでは、WebSphere Application Server Community Edition を使用して eXtreme Scale REST データ・サービスを構成および開始する方法について説明します。

始める前に

サンプル・データ・グリッドが開始されていることを確認します。グリッドの開始方法の詳細については、232 ページの『REST データ・サービスの使用可能化』を参照してください。

手順

1. WebSphere Application Server Community Edition バージョン 2.1.1.3 以降をダウンロードして、 `wasce_root` (`/opt/IBM/wasce` など) にインストールします。

2. 以下のコマンドを実行して、WebSphere Application Server Community Edition サーバーを始動します。
   - Linux UNIX `wasce_root/bin/startup.sh`
   - Windows `wasce_root/bin/startup.bat`
3. eXtreme Scale セキュリティーを使用可能にして eXtreme Scale グリッドを始動した場合には、 `restservice_home/gettingstarted/restclient/bin/wxsRestService.properties` ファイルで以下のプロパティーを設定します。

```
ogClientPropertyFile=restservice_home/gettingstarted/security/security.ogclient.properties
loginType=none
```

4. 以下のように、eXtreme Scale REST データ・サービスおよび提供サンプルを WebSphere Application Server Community Edition サーバーにインストールします。
   a. 以下のように、ObjectGrid クライアント・ランタイム JAR を WebSphere Application Server Community Edition リポジトリーに追加します。
      1) WebSphere Application Server Community Edition 管理コンソールを開いてログインします。

         ヒント: デフォルト URL は `http://localhost:8080/console` です。デフォルト・ユーザー IDは `system` で、パスワードは `manager` です。
      2) 「サービス」フォルダー内の「**リポジトリー**」をクリックします。
      3) 「**リポジトリーへのアーカイブの追加**」セクションで、入力テキスト・ボックスに以下を入力します。

表 16. リポジトリーへのアーカイブ

テキスト・ボックス	値
ファイル	wxs_home/lib/ogclient.jar
グループ	com.ibm.websphere.xs
成果物	ogclient
バージョン	7.0
タイプ	jar

      4) 「インストール」ボタンをクリックします。

         ヒント: 構成クラスおよびライブラリーの依存関係のさまざまな方法の詳細については、技術情報 Specifying external dependencies to applications running on WebSphere Application Server Community Edition を参照してください。
   b. REST データ・サービス・モジュール `wxsrestservice.war` ファイルを WebSphere Application Server Community Edition サーバーにデプロイします。
      1) 開始用 (getting started) サンプル・クラスパス・ディレクトリーへのパス依存関係を組み込むように、サンプルの `restservice_home/gettingstarted/wasce/geronimo-web.xml` デプロイメント XML ファイルを編集します。

2 つの開始用 (getting started) クライアント GBean の classesDirs パスを変更します。

- GettingStarted_Client_SharedLib GBean の「classesDirs」パスを restservice_home/gettingstarted/restclient/bin に設定する必要があります。
- GettingStarted_Common_SharedLib GBean の「classesDirs」パスを restservice_home/gettingstarted/common/bin に設定する必要があります。

2) WebSphere Application Server Community Edition 管理コンソールを開いてログインします。

ヒント: デフォルト URL は http://localhost:8080/console です。デフォルト・ユーザー IDは system で、パスワードは manager です。

3) 「**新規デプロイ**」をクリックします。

4) 「**新規アプリケーションのインストール**」ページで、テキスト・ボックスに以下の値を入力します。

表 17. インストール値

テキスト・ボックス	**値**
アーカイブ	restservice_home/lib/wxsrestservice.war
プラン	restservice_home/gettingstarted/wasce/geronimo-web.xml

5) 「インストール」ボタンをクリックします。

コンソール・ページに、アプリケーションが正常にインストールされて開始されたことが示されます。

6) WebSphere Application Server Community Edition システム出力ログまたはコンソールで以下のメッセージが存在することを確認して、REST データ・サービスが正常に開始されていることを検査します。

CWOBJ4000I: WebSphere eXtreme Scale REST データ・サービスが開始されました。

5. 以下のように、REST データ・サービスが動作していることを確認します。
   a. ブラウザー・ウィンドウで、リンク http://localhost:8080/wxsrestservice/restservice/NorthwindGrid を開きます。 NorthwindGrid グリッドのサービス文書が表示されます。
   b. ブラウザー・ウィンドウで、リンク http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/$metadata を開きます。 Entity Model Data Extensions (EDMX) 文書が表示されます。
6. グリッド・プロセスを停止するには、それぞれのコマンド・ウィンドウで CTRL+C を使用して、プロセスを停止します。
7. WebSphere Application Server Community Edition を停止するには、以下のコマンドを使用します。

- UNIX Linux `wasce_root/bin/shutdown.sh`
- Windows `wasce_root¥bin¥shutdown.bat`

ヒント: デフォルト・ユーザー IDは `system` で、パスワードは `manager` です。カスタム・ポートを使用する場合は、`-port` オプションを使用します。

Apache Tomcat での REST データ・サービスの開始

このトピックでは、Apache Tomcat バージョン 5.5 以上を使用して、eXtreme Scale REST データ・サービスを構成および開始する方法について説明します。

始める前に

サンプル eXtreme Scale グリッドが開始されていることを確認します。グリッドの開始方法の詳細については、232 ページの『REST データ・サービスの使用可能化』を参照してください。

手順

1. tomcat_root に Apache Tomcat バージョン 5.5 以上をダウンロードしてインストールします。例: `/opt/tomcat`
2. 以下のように、eXtreme Scale REST データ・サービスおよび提供サンプルを Tomcat サーバーにインストールします。
   a. Sun JRE または JDK を使用する場合は、IBM ORB を Tomcat にインストールする必要があります。
      - Tomcat バージョン 5.5 の場合

        すべての JAR ファイルを以下のようにコピーします。

        `wxs_home/lib/endorsed`

        から

        `tomcat_root/common/endorsed`
      - Tomcat バージョン 6.0 の場合
        1) 「endorsed」ディレクトリーを作成します。
           - UNIX Linux `mkdir tomcat_root/endorsed`
           - Windows `md tomcat_root/endorsed`
        2) すべての JAR ファイルを以下のようにコピーします。

           `wxs_home/lib/endorsed`

           から

           `tomcat_root/endorsed`

   b. REST データ・サービス・モジュール wxsrestservice.war を Tomcat サーバーにデプロイします。

      wxsrestservice.war ファイルを以下のようにコピーします。

restservice_home/lib

から

tomcat_root/webapps

c. ObjectGrid クライアント・ランタイム JAR およびアプリケーション JAR を Tomcat の共有クラスパスに追加します。

1) tomcat_root/conf/catalina.properties ファイルを編集します。

2) コンマ区切りリストの形式で、以下のパス名を shared.loader プロパティーの末尾に追加します。

- wxs_home/lib/ogclient.jar
- restservice_home/gettingstarted/restclient/bin
- restservice_home/gettingstarted/common/bin

重要: パス分離文字は、**スラッシュ**にする必要があります。

3. eXtreme Scale セキュリティーを使用可能にして eXtreme Scale グリッドを始動した場合には、 restservice_home/gettingstarted/restclient/bin/wxsRestService.properties ファイルで以下のプロパティーを設定します。

```
ogClientPropertyFile=restservice_home/gettingstarted/security/security.ogclient.properties
loginType=none
```

4. REST データ・サービスが含まれた Tomcat サーバーを始動します。

- Tomcat 5.5 を UNIX® または Windows® で、あるいは Tomcat 6.0 を UNIX で使用する場合:

a. cd tomcat_root/bin

b. 以下のように、サーバーを始動します。

– UNIX Linux ./catalina.sh run

– Windows catalina.bat run

c. コンソールに、Apache Tomcat のログが表示されます。 REST データ・サービスが正常に開始すると、管理コンソールに以下のメッセージが表示されます。

CWOBJ4000I: WebSphere eXtreme Scale REST データ・サービスが開始されました。

- Tomcat 6.0 を Windows で使用する場合:

a. cd tomcat_root/bin

b. tomcat6w.exe コマンドで、Apache Tomcat 6 構成ツールを開始します。

c. Apache Tomcat 6 のプロパティー・ウィンドウの「Start」ボタンをクリックして、Tomcat サーバーを始動します。

d. 以下のログを確認して、Tomcat サーバーが正常に始動していることを確認します。

– tomcat_root/bin/catalina.log

Tomcat サーバー・エンジンの状況を表示します。

– tomcat_root/bin/stdout.log

システム出力ログを表示します。

e. REST データ・サービスが正常に開始されていると、以下のメッセージがシステム出力ログに表示されます。 `CWOBJ4000I: WebSphere eXtreme Scale REST データ・サービスが開始されました。`

5. 以下のように、REST データ・サービスが動作していることを確認します。
   a. ブラウザーを開いて、以下にナビゲートします。

      http://localhost:8080/wxsrestservice/restservice/NorthwindGrid

      NorthwindGrid のサービス文書が表示されます。
   b. 以下にナビゲートします。

      http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/$metadata

      Entity Model Data Extensions (EDMX) 文書が表示されます。
6. グリッド・プロセスを停止するには、それぞれのコマンド・ウィンドウで CTRL+C を使用します。
7. Tomcat を停止するには、Tomcat を開始したウィンドウで CTRL+C を使用します。

REST データ・サービスでのブラウザーの使用

eXtreme Scale REST データ・サービスは、Web ブラウザーを使用した場合に、ATOM フィードをデフォルトで作成します。ATOM フィード・フォーマットは、古いブラウザーでは互換性がないことがあり、また、XML として表示できないデータとして解釈されることがあります。以下のトピックでは、ブラウザー内で ATOM フィードおよび XML を表示するように Internet Explorer バージョン 8 および Firefox バージョン 3 を構成する方法を詳細に説明します。

このタスクについて

eXtreme Scale REST データ・サービスは、Web ブラウザーを使用した場合に、ATOM フィードをデフォルトで作成します。ATOM フィード・フォーマットは、古いブラウザーでは互換性がないことがあり、また、XML として表示できないデータとして解釈されることがあります。古いブラウザーでは、ファイルをディスクに保存するように求められます。ファイルをダウンロードしてから、お好みの XML リーダーを使用して、ファイルを参照してください。生成された XML は表示用にフォーマット設定されていないため、すべてが 1 行で出力されます。ほとんどの XML 読み取りプログラム (Eclipse など) では、XML を可読フォーマットに再設定することができます。

最新のブラウザー (Microsoft Internet Explorer バージョン 8 や Firefox バージョン 3 など) では、ATOM XML ファイルをブラウザーでネイティブ表示できます。以下のトピックでは、ブラウザー内で ATOM フィードおよび XML を表示するように Internet Explorer バージョン 8 および Firefox バージョン 3 を構成する方法を詳細に説明します。

手順

Internet Explorer バージョン 8 の構成

- REST データ・サービスが生成する ATOM フィードを Internet Explorer で表示できるようにするには、以下のステップを使用します。
  1. 「**ツール**」 → 「**インターネット オプション**」をクリックします。
  2. 「**コンテンツ**」タブを選択します。
  3. 「**フィードと Web スライス**」セクションの「**設定**」ボタンをクリックします。
  4. 「フィードの読み取りビューを有効にする」ボックスのチェック・マークを外します。
  5. 「**OK**」をクリックして、ブラウザーに戻ります。
  6. Internet Explorer を再始動します。

Firefox バージョン 3 の構成

- Firefox では、コンテンツ・タイプが application/atom+xml のページは自動的には表示されません。ページの初回表示時に、Firefox でファイルを保存するように求められます。ページを表示するには、以下のように、Firefox でファイル自体を開きます。
  1. アプリケーション選択ダイアログ・ボックスで、「アプリケーションで開く」ラジオ・ボタンを選択して、「**参照**」ボタンをクリックします。
  2. Firefox インストール・ディレクトリーにナビゲートします。例: `C:¥Program Files¥Mozilla Firefox`
  3. `firefox.exe` を選択して、「**OK**」ボタンをクリックします。
  4. 「今後この種類のファイルは同様に処理する」チェック・ボックスにチェック・マークを付けます。
  5. 「**OK**」ボタンをクリックします。
  6. Firefox で、ATOM XML ページが新しいブラウザー・ウィンドウまたはタブで表示されます。
- Firefox は、ATOM フィードを可読フォーマットで自動的にレンダリングします。ただし、REST データ・サービスが作成するフィードには XML が含まれます。 Firefox では、フィード・レンダラーを使用不可にしない限り、XML を表示できません。 Internet Explorer とは異なり、Firefox では、ATOM フィード・レンダリング・プラグインを明示的に編集する必要があります。 ATOM フィードを XML ファイルとして読み取るように Firefox を構成するには、以下のステップに従います。
  1. ファイル `<firefoxInstallRoot>¥components¥FeedConverter.js` をテキスト・エディターで開きます。このパスで、`<firefoxInstallRoot>` は、Firefox がインストールされているルート・ディレクトリーです。

     Windows オペレーティング・システムの場合、デフォルト・ディレクトリーは `C:¥Program Files¥Mozilla Firefox` です。
  2. 以下のようなスニペットを探します。

     ```
     // show the feed page if it wasn't sniffed and we have a document,
     // or we have a document, title, and link or id
     if (result.doc && (!this._sniffed ||
         (result.doc.title && (result.doc.link || result.doc.id)))) {
     ```
  3. `if` および `result` で開始している 2 行の行頭に `//` (2 つのスラッシュ) を追加して、コメント化します。

4. スニペットにステートメント `if(0) {` を追加します。
5. 結果のテキストは以下のようになります。

```
// show the feed page if it wasn't sniffed and we have a document,
// or we have a document, title, and link or id
//if (result.doc && (!this._sniffed ||
//    (result.doc.title && (result.doc.link || result.doc.id)))) {
if(0) {
```

6. ファイルを保存します。
7. Firefox を再始動します。
8. これで、Firefox のブラウザーで、すべてのフィードを自動的に表示できるようになりました。

- いくつかの URL を試して、セットアップをテストします。

例

このセクションでは、eXtreme Scale REST に付属の開始用 (getting started) サンプルで追加されたデータを表示するために使用できる一部のサンプル URL について説明します。以下の URL を使用する前に、サンプル Java クライアントまたはサンプル Visual Studio WCF Data Services クライアントを使用して、デフォルト・データ・セットを eXtreme Scale サンプル・グリッドに追加します。

以下の例では、ポートは 8080 であると想定しています (ポートは可変)。別のアプリケーション・サーバーで REST データ・サービスを構成する方法の詳細については、セクションを参照してください。

- 「ACME」という ID の単一の顧客を表示:

  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('ACME')
  ```

- 「ACME」という顧客のすべてのオーダーを表示:

  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('ACME')/orders
  ```

- 顧客「ACME」およびオーダーを表示:

  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('ACME')?$expand=orders
  ```

- 顧客「ACME」のオーダー 1000 を表示:

  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=1000,customer_customerId='ACME')
  ```

- 顧客「ACME」のオーダー 1000 および関連付けられた Customer を表示:

  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Order(orderId=1000,customer_customerId='ACME')?$expand=customer
  ```

- 顧客「ACME」のオーダー 1000 および関連付けられた Customer および OrderDetails を表示:

  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Order(orderId=1000,customer_customerId='ACME')?$expand=customer,orderDetails
  ```

- 顧客「ACME」の 2009 年 10 月 (GMT) のすべてのオーダーを表示:

  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Customer(customerId='ACME')/orders?$filter=orderDate
  ge datetime'2009-10-01T00:00:00'
  and orderDate lt datetime'2009-11-01T00:00:00'
  ```

- 顧客「ACME」の 2009 年 10 月 (GMT) の最初の 3 つのオーダーおよび orderDetails を表示:

```
http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
Customer(customerId='ACME')/orders?$filter=orderDate
ge datetime'2009-10-01T00:00:00'
and orderDate lt datetime'2009-11-01T00:00:00'
&$orderby=orderDate&$top=3&$expand=orderDetails
```

REST データ・サービスでの Java クライアントの使用

Java クライアント・アプリケーションは eXtreme Scale EntityManager API を使用して、データをグリッドに挿入します。

このタスクについて

前のセクションでは、eXtreme Scale グリッドを作成して、eXtreme Scale REST データ・サービスを構成および開始する方法について説明しました。Java クライアント・アプリケーションは eXtreme Scale EntityManager API を使用して、データをグリッドに挿入します。REST インターフェースの使用方法については説明されていません。このクライアントの目的は、EntityManager API を使用して eXtreme Scale グリッドと対話して、グリッド内のデータを変更できるようにする方法を説明することです。 REST データ・サービスを使用してグリッド内のデータを表示するには、 Web ブラウザーを使用するか、Visual Studio 2008 クライアント・アプリケーションを使用します。

手順

eXtreme Scale グリッドにコンテンツを迅速に追加するには、以下のコマンドを実行します。

1. コマンド行または端末ウィンドウを開いて、以下のように、JAVA_HOME 環境変数を設定します。
   - Linux UNIX `export JAVA_HOME=java_home`
   - Windows `set JAVA_HOME=java_home`
2. `cd restservice_home/gettingstarted`
3. グリッドに何らかのデータを挿入します。挿入したデータは、後から Web ブラウザーおよび REST データ・サービスを使用して取得します。

   eXtreme Scale セキュリティーなしで グリッドを始動した場合には、以下のコマンドを使用します。

   - UNIX Linux `./runclient.sh load default`
   - Windows `runclient.bat load default`

   eXtreme Scale セキュリティーありで グリッドを始動した場合には、以下のコマンドを使用します。

   - UNIX Linux `./runclient_secure.sh load default`
   - Windows `runclient_secure.bat load default`

   Java クライアントの場合は、以下のコマンド構文を使用します。

   - UNIX Linux `runclient.sh command`

- Windows `runclient.bat command`

以下のコマンドが使用可能です。

- `load default`

  Customer、Category、および Product の各エンティティーの事前定義セットをグリッドにロードして、顧客ごとに Orders のランダム・セットを作成します。

- `load category categoryId categoryName firstProductId num_products`

  製品 Category および固定数の Product エンティティーをグリッドに作成します。 firstProductId パラメーターには、最初の製品の ID 番号を指定し、指定数の製品が作成されるまで、それ以降の製品に次の ID を割り当てます。

- `load customer companyCode contactNamecompanyName numOrders firstOrderIdshipCity maxItems discountPct`

  新規 Customer をグリッドにロードして、現在グリッドにロードされている任意のランダム製品の Order エンティティーの固定セットを作成します。 Order の数は、<numOrders> パラメーターを設定することで決定します。各 Order には、ランダム数の OrderDetail エンティティーが含まれます (最大数は <maxItems>)。

- `display customer companyCode`

  Customer エンティティー、および関連付けられた Order エンティティーと OrderDetail エンティティーを表示します。

- `display category categoryId`

  製品 Category エンティティーおよび関連付けられた Product エンティティーを表示します。

タスクの結果

- `runclient.bat load default`
- `runclient.bat load customer IBM "John Doe" "IBM Corporation" 5 5000 Rochester 5 0.05`
- `runclient.bat load category 5 "Household Items" 100 5`
- `runclient.bat display customer IBM`
- `runclient.bat display category 5`

Eclipse でのサンプル・グリッドおよび Java クライアントの実行およびビルド

REST データ・サービスの開始用 (getting started) サンプルは、Eclipse を使用して更新および拡張できます。 Eclipse 環境のセットアップ方法の詳細については、テキスト資料 `restservice_home/gettingstarted/ECLIPSE_README.txt` を参照してください。

WXSRestGettingStarted プロジェクトを Eclipse をインポートして正常にビルドすると、サンプルが自動的に再コンパイルされて、コンテナー・サーバーおよびクライアントを開始するために使用するスクリプト・ファイルでクラス・ファイルおよび

XML ファイルが自動的に選択されます。Web サーバーが Eclipse ビルド・ディレクトリーを自動的に読み取るように構成されているため、変更が行われると、REST データ・サービスでもその変更が自動的に検出されます。

重要: ソースまたは構成ファイルを変更する際には、eXtreme Scale コンテナー・サーバーと REST データ・サービス・アプリケーションの両方を再始動する必要があります。eXtreme Scale コンテナー・サーバーは、REST データ・サービス Web アプリケーションの前に始動する必要があります。

REST データ・サービスでの Visual Studio 2008 WCF クライアント

eXtreme Scale REST データ・サービス開始用 (getting started) サンプルには、eXtreme Scale REST データ・サービスと対話できる WCF Data Services クライアントが含まれています。サンプルは、C# のコマンド行アプリケーションとして作成されています。

ソフトウェア要件

WCF Data Services C# サンプル・クライアントには、以下が必要です。

- オペレーティング・システム
  - Microsoft Windows XP
  - Microsoft Windows Server 2003
  - Microsoft Windows Server 2008
  - Microsoft Windows Vista
- Microsoft Visual Studio 2008 (Service Pack 1 適用済み)

  ヒント: 追加のハードウェアおよびソフトウェアの要件については、上のリンクを参照してください。
- Microsoft .NET Framework 3.5 Service Pack 1
- Microsoft Support: .NET Framework 3.5 Service Pack 1 の更新が使用可能です

開始用 (getting started) クライアントのビルドおよび実行

WCF Data Services サンプル・クライアントには、サンプルを実行するための、Visual Studio 2008 のプロジェクトとソリューションおよびソース・コードが含まれています。サンプルを実行するには、Visual Studio 2008 にロードして、Windows 実行可能プログラムにコンパイルする必要があります。サンプルをビルドして実行するには、テキスト資料 `restservice_home/gettingstarted/VS2008_README.txt` を参照してください。

WCF Data Services C# クライアントのコマンド構文

Windows `WXSRESTGettingStarted.exe <サービス URL> <コマンド>`

<サービス URL> は、セクションで構成された eXtreme Scale REST データ・サービスの URL です。

以下のコマンドが使用可能です。

- `load default`

  Customer、Category、および Product の各エンティティーの事前定義セットをグリッドにロードして、顧客ごとに Orders のランダム・セットを作成します。
- `load category <categoryId> <categoryName> <firstProductId> <numProducts>`

  製品 Category および固定数の Product エンティティーをグリッドに作成します。 firstProductId パラメーターには、最初の製品の ID 番号を指定し、指定数の製品が作成されるまで、それ以降の製品に次の ID を割り当てます。
- `load customer <companyCode> <contactName> <companyName> <numOrders> <firstOrderId> <shipCity> <maxItems> <discountPct>`

  新規 Customer をグリッドにロードして、現在グリッドにロードされている任意のランダム製品の Order エンティティーの固定セットを作成します。 Order の数は、<numOrders> パラメーターを設定することで決定します。各 Order には、ランダム数の OrderDetail エンティティーが含まれます (最大数は <maxItems>)。
- `display customer <companyCode>`

  Customer エンティティー、および関連付けられた Order エンティティーと OrderDetail エンティティーを表示します。
- `display category <categoryId>`

  製品 Category エンティティーおよび関連付けられた Product エンティティーを表示します。
- `unload`

  「default load」コマンドを使用してロードされたすべてのエンティティーを削除します。

次に、さまざまなコマンドの例を示します。

- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid load default`
- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid load customer`
- `IBM "John Doe" "IBM Corporation" 5 5000 Rochester 5 0.05`
- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid load category 5 "Household Items" 100 5`
- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid display customer IBM`
- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid display category 5`

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。

- AIX®
- CICS®
- cloudscape
- DB2
- Domino®
- IBM
- Lotus®
- RACF®
- Redbooks®
- Tivoli
- WebSphere
- z/OS®

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。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

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

P

S

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