



Red Hat Data Grid 8.2

Data Grid の設定

Data Grid の機能の有効化および調整

Red Hat Data Grid 8.2 Data Grid の設定

Data Grid の機能の有効化および調整

法律上の通知

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

ビジネス要件に合わせて、プログラムを用いて、または宣言的に Data Grid デプロイメントを設定します。

目次

RED HAT DATA GRID	3
DATA GRID のドキュメント	4
DATA GRID のダウンロード	5
多様性を受け入れるオープンソースの強化	6
第1章 DATA GRID キャッシュ	7
1.1. キャッシュインターフェイス	7
1.2. キャッシュマネージャー	7
1.3. キャッシュコンテナー	7
1.4. キャッシュモード	8
第2章 ローカルキャッシュ	10
2.1. シンプルキャッシュ	10
第3章 クラスター化されたキャッシュ	12
3.1. インバリデーションモード	12
3.2. レプリケートされたキャッシュ	13
3.3. 分散キャッシュ	14
3.4. 散在 (SCATTERED) キャッシュ	22
3.5. クラスター化されたキャッシュとの非同期通信	23
第4章 DATA GRID キャッシュの設定	25
4.1. 宣言型設定	25
4.2. DATA GRID の設定 API	27
4.3. キャッシュプログラムによる設定	28
第5章 メモリーの管理	31
5.1. エビクションおよびの有効期限の設定	31
5.2. オフヒープメモリー	39
第6章 統計、メトリクス、および JMX の設定	43
6.1. DATA GRID 統計の有効化	43
6.2. DATA GRID メトリクスの設定	43
6.3. JMX MBEAN を登録するための DATA GRID の設定	44
第7章 永続ストレージのセットアップ	47
7.1. DATA GRID キャッシュストア	47
7.2. キャッシュストア実装	54
7.3. キャッシュストア間の移行	68
第8章 パーティション処理の設定	76
8.1. パーティション処理	76

RED HAT DATA GRID

Data Grid は、高性能の分散型インメモリーデータストアです。

スキーマレスデータ構造

さまざまなオブジェクトをキーと値のペアとして格納する柔軟性があります。

グリッドベースのデータストレージ

クラスター間でデータを分散および複製するように設計されています。

エラスティックスケーリング

サービスを中断することなく、ノードの数を動的に調整して要件を満たします。

データの相互運用性

さまざまなエンドポイントからグリッド内のデータを保存、取得、およびクエリーします。

DATA GRID のドキュメント

Data Grid のドキュメントは、Red Hat カスタマーポータルで入手できます。

- [Data Grid 8.2 ドキュメント](#)
- [Data Grid 8.2 コンポーネントの詳細](#)
- [Data Grid 8.2 でサポートされる設定](#)
- [Data Grid 8 機能のサポート](#)
- [Data Grid で非推奨の機能](#)

DATA GRID のダウンロード

Red Hat カスタマーポータルで [Data Grid Software Downloads](#) にアクセスします。



注記

Data Grid ソフトウェアにアクセスしてダウンロードするには、Red Hat アカウントが必要です。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#) をご覧ください。

第1章 DATA GRID キャッシュ

Data Grid キャッシュは、以下のようなユースケースに合わせて、柔軟なインメモリーデータストアを提供します。

- 高速のローカルキャッシュでアプリケーションパフォーマンスを強化。
- 書き込み操作のボリュームを減らすことでデータベースの最適化。
- クラスタ間で一貫したデータの復元力と耐久性を提供。

1.1. キャッシュインターフェイス

Cache<K,V> は、Data Grid の中央インターフェイスであり、**java.util.concurrent.ConcurrentMap** を拡張します。

キャッシュエントリは、単純な文字列からより複雑なオブジェクトまで、幅広いデータ型をサポートする **key:value** 形式の同時データ構造です。

1.2. キャッシュマネージャー

Data Grid は、ローカルまたはクラスター化されたキャッシュを作成、変更、および管理できる **CacheManager** インターフェイスを提供します。キャッシュマネージャーは、Data Grid キャッシュを使用する際の開始点です。

CacheManager 実装には 2 種類あります。

EmbeddedCacheManager

クライアントアプリケーションと同じ Java Virtual Machine (JVM) 内で Data Grid を実行する場合のキャッシュのエントリーポイント (ライブラリーモードとも呼ばれます)。

RemoteCacheManager

Data Grid を独自の JVM でリモートサーバーとして実行する際のキャッシュのエントリーポイント。実行を開始すると、**RemoteCacheManager** は Data Grid サーバーの Hot Rod エンドポイントへの永続的な TCP 接続を確立します。



注記

埋め込みおよびリモートの **CacheManager** 実装は、一部のメソッドとプロパティを共有します。ただし、セマンティックの違いは **EmbeddedCacheManager** と **RemoteCacheManager** の間に存在します。

1.3. キャッシュコンテナ

キャッシュコンテナは、Cache Manager が制御する 1 つ以上のローカルキャッシュまたはクラスター化されたキャッシュを宣言します。

キャッシュコンテナの宣言

```
<cache-container name="clustered" default-cache="default">
  <!-- Cache Manager configuration goes here. -->
</cache-container>
```

1.4. キャッシュモード

ヒント

Data Grid の Cache Manager は、異なるモードを使用する複数のキャッシュを作成および制御できます。たとえば、ローカルキャッシュと同じ Cache Manager を使用し、キャッシュをインバリデーションモードで分散し、キャッシュを分散できます。

ローカルキャッシュ

Data Grid は単一ノードとして実行され、キャッシュエントリーに対して読み取り操作または書き込み操作を複製しません。

クラスター化されたキャッシュ

同じネットワークで実行している Data Grid インスタンスは、自動的に相互を検出し、キャッシュ操作を処理するためにクラスターを形成することができます。

インバリデーションモード

クラスター全体でキャッシュエントリーを複製する代わりに、Data Grid は操作によってキャッシュのエントリーが変更するたびに、すべてのノードから古いデータを削除します。Data Grid は、ローカルの読み取り操作のみを実行します。

レプリケートされたキャッシュ

Data Grid は、すべてのノードで各キャッシュエントリーを複製し、ローカルの読み取り操作のみを実行します。

分散キャッシュ

Data Grid は、ノードのサブセット全体でキャッシュエントリーを保存し、エントリーを固定所有者ノードに割り当てます。Data Grid は所有者ノードから読み取り操作を要求し、正しい値を返すようにします。

散在 (scattered) キャッシュ

Data Grid は、ノードのサブセット全体でキャッシュエントリーを保存します。デフォルトでは、Data Grid はプライマリーの所有者とバックアップ所有者を散在キャッシュの各キャッシュエントリーに割り当てます。Data Grid は分散キャッシュと同じ方法でプライマリー所有者を割り当てますが、バックアップ所有者は常に書き込み操作を開始するノードになります。Data Grid は、少なくとも 1 つの所有者ノードから読み取り操作を要求し、正しい値を返すようにします。

1.4.1. キャッシュモードの比較

選択するキャッシュモードは、データに必要な数量と保証によって異なります。

以下の表は、キャッシュモードの主な相違点をまとめています。

キャッシュモード	クラスター化？	読み取りパフォーマンス	書き込みパフォーマンス	容量	可用性	機能
Local	いいえ	高(ローカル)	高(ローカル)	単一ノード	単一ノード	完了

キャッシュ モード	クラスター 化？	読み取りパ フォーマンス	書き込みパ フォーマンス	容量	可用性	機能
Simple (単 純)	いいえ	最高(ロー カル)	最高(ロー カル)	単一ノード	単一ノード	Partial: ト ランザク ション、永 続性、また はインデッ クスなし。
Invalidation (無効化)	Yes	高(ローカ ル)	低い(すべ てのノー ド、デー タなし)	単一ノード	単一ノード	部分的: イン デックス 化なし
Replicated (レプリ ケート)	Yes	高(ローカ ル)	最低(すべ てのノー ド)	最小のノード	全ノード	完了
Distributed (分散)	Yes	メディ ア(所有者)	メディ ア(所有者 ノード)	所有者数で区分 されたすべての ノード容量の合 計。	所有者ノー ド	完了
Scattered(散在)	Yes	中(プライ マリー)	さらに高 (単一 RPC)	2 で除算された すべてのノード 容量の合計。	所有者ノー ド	部分的: ト ランザク ションがあ りません。

第2章 ローカルキャッシュ

Data Grid はクラスターモードで特に興味深いものですが、非常に有能なローカルモードも提供します。このモードでは、**ConcurrentHashMap** と同様の単純なインメモリーデータキャッシュとして機能します。

ただし、マップではなくローカルキャッシュを使用するのはなぜでしょうか。キャッシュは、単純なマップよりも多くの機能を提供します。これには、永続ストアへのライトスルーおよびライトビハインド、エントリーのエビクション、メモリー不足や有効期限の発生を防ぐことができます。

Data Grid の **Cache** インターフェイスは、JDK の **ConcurrentMap** を拡張し、マップから DataGrid への移行を簡単にします。

Data Grid キャッシュは、既存のトランザクションマネージャーと統合するか、別のトランザクションマネージャーを実行するトランザクションもサポートします。ローカルキャッシュトランザクションには、2つの選択肢があります。

1. ロックのタイミング。悲観的ロックは、書き込み操作時、またはユーザーが **AdvancedCache.lock(keys)** を明示的に呼び出したときにキーをロックします。楽観的ロックは、トランザクションのコミット中にのみキーをロックし、代わりに、現在のトランザクションがキーを読み取った後に別のトランザクションが同じキーを変更した場合は、コミット時に **WriteSkewCheckException** を出力します。
2. 分離レベル **read-committed** および **repeatable read** をサポートします。

2.1. シンプルキャッシュ

従来のローカルキャッシュは、クラスター化されたキャッシュと同じアーキテクチャーを使用します。つまり、インターセプタースタックを使用します。これにより、多くの実装を再利用できます。ただし、高度な機能が不要でパフォーマンスがより重要な場合は、インターセプタースタックを削除して、単純なキャッシュを使用できます。

そのため、どの機能も削除されますか。設定の観点からは、簡単なキャッシュは以下に対応していません。

- トランザクションと呼び出しバッチ処理
- 永続性 (キャッシュストアおよびローダー)
- カスタムインターセプター (インターセプタースタックなし)
- インデックス化
- トランスコーディング
- バイナリーとして保存 (ローカルキャッシュに非常に便利です)

API パースペクティブから、これらの機能は例外を出力します。

- カスタムインターセプターの追加
- 分散済みエグゼキューターフレームワーク

そして、何が残っていますか。

- 基本的なマップのような API

- キャッシュリスナー (ローカルリスナー)
- 有効期限
- eviction
- security
- JMX アクセス
- 統計 (ただし、最大のパフォーマンスを得るには、statistics-available=false を使用してこれをオフにすることが推奨されます)

宣言型設定

```
<local-cache name="mySimpleCache" simple-cache="true">  
  <!-- Additional cache configuration goes here. -->  
</local-cache>
```

プログラムによる設定

```
DefaultCacheManager cm = getCacheManager();  
ConfigurationBuilder builder = new ConfigurationBuilder().simpleCache(true);  
cm.defineConfiguration("mySimpleCache", builder.build());  
Cache cache = cm.getCache("mySimpleCache");
```

サポートされていない機能に対する簡単なキャッシュチェック。たとえばトランザクションを使用するよう設定すると、設定検証によって例外が出力されます。

第3章 クラスター化されたキャッシュ

クラスター化されたキャッシュは、ネットワークを介してデータを渡すためのトランスポート層として JGroups テクノロジーを使用して、複数の Data Grid ノードにまたがってデータを格納します。

3.1. インバリデーションモード

Data Grid をインバリデーションモードで使用して、大量の読み取り操作を実行するシステムを最適化できます。良い例は、インバリデーションを使用して、状態の変化が発生したときに大量のデータベース書き込みを防ぐことです。

このキャッシュモードは、データベースなどのデータ用に別の永続的なストアがあり、読み取りが多いシステムで最適化として Data Grid を使用している場合にのみ意味があり、読み取りごとにデータベースにアクセスするのを防ぎます。キャッシュがインバリデーション用に設定されている場合は、データをキャッシュに変更するたびに、クラスター内の他のキャッシュは、データが古いため、メモリーおよびローカルストアから削除される必要があることを通知するメッセージを受信します。

図3.1 インバリデーションモード

アプリケーションは外部ストアから値を読み取り、他のノードから削除せずにローカルキャッシュに書き込む場合があります。これを実行するには、**Cache.put(key, value)** の代わりに **Cache.putForExternalRead(key, value)** を呼び出す必要があります。

インバリデーションモードは、共有キャッシュストアと使用できます。書き込み操作は、共有ストアを更新し、他のノードメモリーから古い値を削除します。これには2つの利点があります。値全体を複製する場合に比べてインバリデーションメッセージが非常に小さいため、ネットワークトラフィックが最小限に抑えられます。また、クラスター内の他のキャッシュは、必要な場合にのみ、変更されたデータを遅延的に検索します。



注記

ローカルストアでインバリデーションモードを使用しないでください。インバリデーションメッセージはローカルストアのエントリーを削除せず、一部のノードが古い値を認識します。

インバリデーションキャッシュは、特別なキャッシュローダー (**ClusterLoader**) で設定することもできます。**ClusterLoader** が有効になっている場合、ローカルノードでキーが見つからない読み取り操作は、最初に他のすべてのノードからキーを要求し、ローカルのメモリーに保存します。特定の状況では古い値を保存するため、古くなった値の耐性がある場合にのみ使用します。

インバリデーションモードは、同期または非同期です。同期すると、クラスター内のすべてのノードが古い値をエビクトするまで、書き込みがブロックされます。非同期の場合、発信者はインバリデーションメッセージをブロードキャストしますが、応答を待ちません。つまり、発信者で書き込みが完了した

後も、他のノードはしばらくの間古い値を確認します。

トランザクションはインバリデーションメッセージをバッチするために使用できます。トランザクションはプライマリ所有者でキーロックを取得します。プライマリ所有者の割り当て方法の詳細は、[Key Ownership](#) セクションを参照してください。

- 悲観的ロックでは、各書き込みは、すべてのノードにブロードキャストされるロックメッセージをトリガーします。トランザクションのコミット中に、発信者は、影響を受けるすべてのキーを無効にし、ロックを解放する1フェーズの準備メッセージ (任意で fire-and-forget) をブロードキャストします。
- 楽観的ロックを使用すると、発信者は準備メッセージ、コミットメッセージ、およびロック解除メッセージ (任意) をブロードキャストします。1フェーズの準備またはロック解除メッセージのいずれかが fire-and-forget であり、最後のメッセージは常にロックを解放します。

3.2. レプリケートされたキャッシュ

任意のノードでレプリケートされたキャッシュに書き込まれたエントリは、クラスター内の他のすべてのノードに複製され、任意のノードからローカルで取得できます。レプリケートモードは、クラスター間で状態を共有するための迅速で簡単な方法を提供しますが、書き込みに必要なメッセージの数がクラスターサイズに比例してスケーリングするため、レプリケーションは実際には小さなクラスター (10 ノード未満) でのみ適切に実行します。Data Grid は、UDP マルチキャストを使用するように設定できます。これにより、この問題がある程度軽減されます。

各キーにはプライマリ所有者があり、一貫性を提供するためにデータコンテナの更新をシリアル化します。プライマリ所有者の割り当て方法の詳細は、[Key Ownership](#) セクションを参照してください。

図3.2 レプリケートモード

レプリケートモードは、同期または非同期にすることができます。

- 同期レプリケーションは、変更がクラスターのすべてのノードに正常に複製されるまで、呼び出し元 (`cache.put(key, value)` など) をブロックします。
- 非同期レプリケーションはバックグラウンドでレプリケーションを実行し、書き込み操作が即座に返されます。非同期レプリケーションは推奨されません。これは、通信エラーやリモートノードで発生したエラーは呼び出し元に報告されないためです。

トランザクションが有効になっていると、書き込み操作はプライマリ所有者によって複製されません。

- 悲観的ロックでは、各書き込みは、すべてのノードにブロードキャストされるロックメッセージをトリガーします。トランザクションのコミット時に、送信元は1フェーズの準備メッセージとロック解除メッセージ (任意) をブロードキャストします。1フェーズの準備またはロック

解除メッセージのいずれかが fire-and-forget になります。

- 楽観的ロックを使用すると、発信者は準備メッセージ、コミットメッセージ、およびロック解除メッセージ (任意) をブロードキャストします。ここで、1 フェーズの準備またはロック解除メッセージが fire-and-forget になります。

3.3. 分散キャッシュ

分散は、**numOwners** として設定された、キャッシュ内の任意のエントリーの固定数のコピーを保持しようとしています。これにより、キャッシュを線形にスケーリングし、ノードがクラスターに追加されるにつれて、より多くのデータを格納できます。

ノードがクラスターに参加およびクラスターから離脱すると、キーのコピー数が **numOwners** より多い場合と少ない場合があります。特に、**numOwners** ノードがすぐに連続して離れると、一部のエントリーが失われるため、分散キャッシュは、**numOwners - 1** ノードの障害を許容すると言われます。

コピー数は、パフォーマンスとデータの持続性を示すトレードオフを表します。維持するコピーが増えると、パフォーマンスは低くなりますが、サーバーやネットワークの障害によるデータ損失のリスクも低くなります。維持されるコピーの数に関係なく、分散は直線的にスケーリングされます。これは、Data Grid のスケーラビリティの鍵となります。

キーの所有者は、キーへの書き込みを調整する1つの**プライマリー所有者**と、0個以上の**バックアップ所有者**に分割されます。プライマリー所有者とバックアップ所有者の割り当て方法の詳細は、[キー所有権](#) セクションを参照してください。

図3.3 分散モード

読み取り操作はプライマリー所有者からの値を要求しますが、妥当な時間内に応答しない場合、バックアップの所有者からも値を要求します。(infinispan.stagger.delay システムプロパティ) は、リクエスト間の遅延を制御します。) キーがローカルキャッシュに存在する場合、読み取り操作には **0** メッセージが必要になる場合があります、すべての所有者が遅い場合は最大 **2 * numOwners** メッセージが必要になる場合があります。

また、書き込み操作は、最大 **2 * numOwners** メッセージで、送信元からプライマリー所有者、**numOwners - 1** メッセージ、プライマリーからバックアップへの1メッセージ、対応する ACK メッセージまでです。



注記

キャッシュトポロジの変更により、読み取りと書き込みの両方で再試行と追加メッセージが発生する可能性があります。

レプリケートモードと同様に、分散モードは同期または非同期にすることもできます。レプリケートされたモードでは、更新が失われる可能性があるため、非同期レプリケーションは推奨されません。更新

の損失に加えて、非同期の分散キャッシュは、スレッドがキーに書き込むときに古い値を確認し、その後同じキーをすぐに読み取ることもできます。

トランザクション分散キャッシュは、トランザクション複製キャッシュと同じ種類のメッセージを使用しますが、ロック/準備/コミット/ロック解除メッセージは、クラスター内のすべてのノードにブロードキャストされるのではなく、**影響を受けるノード**(トランザクションの影響を受ける少なくとも1つのキーを所有するすべてのノード)にのみ送信されます。最適化として、トランザクションが単一のキーに書き込み、送信元がキーの主な所有者である場合、ロックメッセージは複製されません。

3.3.1. 読み取りの一貫性

同期レプリケーションを使用しても、分散キャッシュは線形化できません。(トランザクションキャッシュの場合は、シリアル化/スナップショットの分離に対応していないとします。)1つのスレッドで1つの put を実行できます。

```
cache.get(k) -> v1
cache.put(k, v2)
cache.get(k) -> v2
```

ただし、別のスレッドでは、異なる順序で値が表示される場合があります。

```
cache.get(k) -> v2
cache.get(k) -> v1
```

その理由は、プライマリー所有者が応答する速度に応じて、クラスター内のすべてのノードが**任意の**所有者から値を返す可能性があるためです。書き込みはすべての所有者にわたってアトミックではありません。実際、プライマリーはバックアップから確認を受け取った後にのみ更新をコミットします。プライマリーがバックアップからの確認メッセージを待機している間、バックアップからの読み取りには新しい値が表示されますが、プライマリーからの読み取りには古い値が表示されます。

3.3.2. キーの所有者

分散キャッシュは、エントリーを固定数のセグメントに分割し、各セグメントを所有者ノードの一覧に割り当てます。レプリケートされたキャッシュは同じで、すべてのノードが所有者である場合を除きます。

所有者リストの最初のノードは**プライマリー所有者**です。一覧のその他のノードは**バックアップの所有者**です。キャッシュトポロジが変更されると、ノードがクラスターに参加またはクラスターから離脱するため、セグメント所有権テーブルがすべてのノードにブロードキャストされます。これにより、ノードはマルチキャスト要求を行ったり、各キーのメタデータを維持したりすることなく、キーを見つけることができます。

numSegments プロパティでは、利用可能なセグメントの数を設定します。ただし、クラスターが再起動しない限り、セグメントの数は変更できません。

同様に、キーからセグメントのマッピングは変更できません。鍵は、クラスタートポロジの変更に関係なく、常に同じセグメントにマップする必要があります。キーからセグメントのマッピングは、クラスタートポロジの変更時に移動する必要があるセグメント数を最小限に抑える一方で、各ノードに割り当てられたセグメント数を均等に分散することが重要になります。

[KeyPartitioner](#) を設定するか、[Grouping API](#) を使用して key-to-segment マッピングをカスタマイズできます。

ただし、Data Grid は以下の実装を提供します。

SyncConsistentHashFactory

[一貫性のあるハッシュ](#)に基づくアルゴリズムを使用します。サーバーヒントを無効にした場合は、デフォルトで選択されています。

この実装では、クラスターが対称である限り、すべてのキャッシュの同じノードに常にキーが割り当てられます。つまり、すべてのキャッシュがすべてのノードで実行します。この実装には、負荷の分散が若干不均等であるため、負のポイントが若干異なります。また、参加または脱退時に厳密に必要な数よりも多くのセグメントを移動します。

TopologyAwareSyncConsistentHashFactory

SyncConsistentHashFactory に似ていますが、[サーバーヒント](#)に適合しています。サーバーヒントが有効な場合にデフォルトで選択されます。

DefaultConsistentHashFactory

SyncConsistentHashFactory よりも均等に分散を行いますが、1つの欠点があります。ノードがクラスターに参加する順序によって、どのノードがどのセグメントを所有するかが決まります。その結果、キーは異なるキャッシュ内の異なるノードに割り当てられる可能性があります。

サーバーヒントを無効にした状態で、バージョン 5.2 からバージョン 8.1 のデフォルトでした。

TopologyAwareConsistentHashFactory

DefaultConsistentHashFactory に似ていますが、[サーバーヒント](#)に適合しています。

サーバーヒントが有効なバージョン 5.2 から 8.1 へのデフォルトでした。

ReplicatedConsistentHashFactory

レプリケートされたキャッシュの実装に内部で使用されます。このアルゴリズムは分散キャッシュで明示的に選択しないでください。

3.3.2.1. 容量ファクト

設備利用率は、ノードで使用可能なリソースに基づいてセグメントからノードへのマッピングを割り当てます。

容量係数を設定するには、負でない数を指定し、Data Grid ハッシュアルゴリズムにより、各ノードに容量係数で重み付けされた負荷が割り当てられます (プライマリ所有者とバックアップ所有者の両方として)。

たとえば、nodeA には、同じ Data Grid クラスター内の nodeB の 2 倍のメモリーがあります。この場合、**capacityFactor** を値 **2** に設定すると、nodeA に 2 倍の数のセグメントを割り当てるように Data Grid が設定されます。

容量係数を **0** に設定することは可能ですが、ノードが有用なデータ所有者として十分な長さでクラスターに参加していない場合にのみ推奨されます。

3.3.3. ゼロ容量ノード

各キャッシュ、ユーザー定義キャッシュ、および内部キャッシュに対して容量係数が **0** であるノード全体の設定が必要になる場合があります。ゼロの容量ノードを定義する場合、ノードはデータを保持しません。これは、ゼロ容量ノードを宣言します。

```
<cache-container zero-capacity-node="true" />
```

```
new GlobalConfigurationBuilder().zeroCapacityNode(true);
```

3.3.4. ハッシュ設定

これは、XML を使用して、ハッシュを宣言的に設定する方法です。

```
<distributed-cache name="distributedCache" owners="2" segments="100" capacity-factor="2" />
```

Java では、プログラムを用いてこの方法で設定できます。

```
Configuration c = new ConfigurationBuilder()
    .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .hash()
    .numOwners(2)
    .numSegments(100)
    .capacityFactor(2)
    .build();
```

3.3.5. 初期クラスターサイズ

トポロジの変更(つまり、実行時にノードが追加/削除される)の処理における Data Grid の非常に動的な性質は、通常、ノードが開始する前に他のノードの存在を待たないことを意味します。これは非常に柔軟性がありますが、キャッシュの開始前に、特定の数のノードがクラスターに参加する必要があるアプリケーションには適切ではない場合があります。このため、キャッシュの初期化に進む前に、クラスターに参加するノードの数を指定できます。これには、**initialClusterSize** および **initialClusterTimeout** トランSPORTプロパティを使用します。宣言型 XML 設定:

```
<transport initial-cluster-size="4" initial-cluster-timeout="30000" />
```

プログラムによる Java 設定:

```
GlobalConfiguration global = new GlobalConfigurationBuilder()
    .transport()
    .initialClusterSize(4)
    .initialClusterTimeout(30000, TimeUnit.MILLISECONDS)
    .build();
```

上記の設定は、初期化前に 4 つのノードがクラスターに参加するのを待機します。指定されたタイムアウト内に最初のノードが表示されない場合は、キャッシュマネージャーが起動に失敗します。

3.3.6. L1 キャッシュ

L1 が有効になっている場合、ノードはリモート読み取りの結果を短期間(設定可能、デフォルトでは 10 分)ローカルに保持し、ルックアップを繰り返すと、所有者に再度尋ねるのではなく、ローカルの L1 値が返されます。

図3.4 L1 キャッシュ

L1 キャッシュは無料ではありません。有効にするとコストがかかり、このコストは、すべてのエントリー更新でインバリデーションメッセージをすべてのノードにブロードキャストする必要があることです。L1 エントリーは、キャッシュが最大サイズで設定されている場合に他のエントリーと同様にエビクトできます。L1 を有効にすると、非ローカルキーの繰り返される読み取りのパフォーマンスが向上しますが、書き込みが遅くなり、メモリー消費量がある程度増加します。

L1 キャッシュが正しいか。正しいアプローチとして、L1 を有効にしない状態でアプリケーションをベンチマークし、アクセスパターンに最も適した動作を確認できます。

3.3.7. サーバーヒント

以下のトポロジーヒントを指定できます。

マシン

これは、複数の JVM インスタンスが同じノードで実行されている場合、または複数の仮想マシンが同じ物理マシンで実行している場合でも、おそらく最も便利です。

ラック

大規模なクラスターでは、同じラックにあるノードでハードウェアまたはネットワークの障害が同時に発生する可能性が高くなります。

サイト

一部のクラスターでは、復元力を高めるために、複数の物理的な場所にノードが存在する場合があります。2 つ以上のデータセンターにまたがる必要のあるクラスターには、クロスサイトレプリケーションも別の方法であることに注意してください。

上記のすべては任意です。指定した場合、ディストリビューショナルアルゴリズムは、できるだけ多くのサイト、ラック、およびマシン全体に各セグメントの所有権を分散しようとします。

3.3.7.1. 設定

ヒントは、トランスポートレベルで設定されます。

```
<transport
  cluster="MyCluster"
  machine="LinuxServer01"
  rack="Rack01"
  site="US-WestCoast" />
```

3.3.8. キーアフィニティーサービス

分散キャッシュでは、不透明なアルゴリズムを使用してノードのリストにキーが割り当てられます。計

算を逆にし、特定のノードにマップする鍵を生成する簡単な方法はありません。ただし、一連の(疑似)ランダムキーを生成し、それらのプライマリ所有者が何であることを確認し、特定のノードへのキーマッピングが必要なときにアプリケーションに渡すことができます。

3.3.8.1. API

以下のコードスニペットは、このサービスへの参照を取得し、使用方法を示しています。

```
// 1. Obtain a reference to a cache
Cache cache = ...
Address address = cache.getCacheManager().getAddress();

// 2. Create the affinity service
KeyAffinityService keyAffinityService = KeyAffinityServiceFactory.newLocalKeyAffinityService(
    cache,
    new RndKeyGenerator(),
    Executors.newSingleThreadExecutor(),
    100);

// 3. Obtain a key for which the local node is the primary owner
Object localKey = keyAffinityService.getKeyForAddress(address);

// 4. Insert the key in the cache
cache.put(localKey, "yourValue");
```

サービスはステップ 2 で開始します。この時点以降、サービスは提供されたエグゼキューターを使用してキーを生成してキューに入れます。ステップ 3 では、サービスから鍵を取得し、手順 4 ではそれを使用します。

3.3.8.2. ライフサイクル

KeyAffinityService は **ライフサイクル** を拡張し、停止と (再) 起動を可能にします。

```
public interface Lifecycle {
    void start();
    void stop();
}
```

サービスは **KeyAffinityServiceFactory** でインスタンス化されます。ファクトリーメソッドはすべて **Executor** パラメーターを持ち、これは非同期キー生成に使用されます (呼び出し元のスレッドでは処理されません)。ユーザーは、この **Executor** のシャットダウンを処理します。

KeyAffinityService が起動したら、明示的に停止する必要があります。これにより、バックグラウンドキーの生成が停止し、保持されている他のリソースが解放されます。

KeyAffinityService がそれ自体で停止する唯一の状況は、登録済みのキャッシュマネージャーがシャットダウンした時です。

3.3.8.3. トポロジーの変更

キャッシュトポロジーが変更すると (つまり、ノードがクラスターに参加またはクラスターから離脱する)、**KeyAffinityService** によって生成されたキーの所有権が変更される可能性があります。主なアフィニティーサービスはこれらのトポロジーの変更を追跡し、現在別のノードにマップされるキーを返しませんが、先に生成したキーに関しては何も実行しません。

そのため、アプリケーションは **KeyAffinityService** を純粋に最適化として処理し、正確性のために生成されたキーの場所に依存しないようにしてください。

特に、アプリケーションは、同じアドレスが常に一緒に配置されるように、**KeyAffinityService** によって生成されたキーに依存するべきではありません。キーのコロケーションは、[Grouping API](#) によってのみ提供されます。

3.3.8.4. Grouping API

[キーアフィニティサービス](#) を補完する Grouping API を使用すると、実際のノードを選択することなく、同じノードにエントリーのグループを同じ場所にコロケートできます。

3.3.8.5. 仕組み

デフォルトでは、キーのセグメントはキーの **hashCode()** を使用して計算されます。Grouping API を使用する場合、Data Grid はグループのセグメントを計算し、それをキーのセグメントとして使用します。セグメントがノードにマップされる方法についての詳細は、[キーの所有権](#) セクションを参照してください。

Grouping API を使用している場合は、他のノードに問い合わせることなく、すべてのノードが全キーの所有者を計算できることが重要です。このため、グループは手動で指定できません。グループは、エントリーに固有 (キークラスによって生成される) または外部 (外部関数によって生成される) のいずれかです。

3.3.8.6. Grouping API を使用する方法

まず、グループを有効にする必要があります。プログラムを用いて Data Grid を設定している場合は、以下を呼び出します。

```
Configuration c = new ConfigurationBuilder()
    .clustering().hash().groups().enabled()
    .build();
```

または、XML を使用している場合は、以下を行います。

```
<distributed-cache>
  <groups enabled="true"/>
</distributed-cache>
```

キークラスを制御できる場合 (クラス定義を変更できるが、変更不可能なライブラリーの一部ではない)、組み込みグループを使用することをお勧めします。intrinsic グループは、**@Group** アノテーションをメソッドに追加して指定します。以下に例を示します。

```
class User {
    ...
    String office;
    ...

    public int hashCode() {
        // Defines the hash for the key, normally used to determine location
        ...
    }

    // Override the location by specifying a group
```



```
// All keys in the same group end up with the same owners
@Group
public String getOffice() {
    return office;
}
}
```



注記

group メソッドは **String** を返す必要があります。

キークラスを制御できない場合、またはグループの決定がキークラスと直交する懸念事項である場合は、外部グループを使用することをお勧めします。外部グループは、**Grouper** インターフェイスを実装することによって指定されます。

```
public interface Grouper<T> {
    String computeGroup(T key, String group);

    Class<T> getKeyType();
}
```

同じキータイプに対して複数の **Grouper** クラスが設定されている場合は、それらすべてが呼び出され、前のクラスで計算された値を受け取ります。キークラスにも **@Group** アノテーションがある場合、最初の **Grouper** はアノテーション付きのメソッドによって計算されたグループを受信します。これにより、組み込みグループを使用するときに、グループをさらに細かく制御できます。**Grouper** 実装の例を見てみましょう。

```
public class KXGrouper implements Grouper<String> {

    // The pattern requires a String key, of length 2, where the first character is
    // "k" and the second character is a digit. We take that digit, and perform
    // modular arithmetic on it to assign it to group "0" or group "1".
    private static Pattern kPattern = Pattern.compile("(^k)(<a>\\d</a>)$");

    public String computeGroup(String key, String group) {
        Matcher matcher = kPattern.matcher(key);
        if (matcher.matches()) {
            String g = Integer.parseInt(matcher.group(2)) % 2 + "";
            return g;
        } else {
            return null;
        }
    }

    public Class<String> getKeyType() {
        return String.class;
    }
}
```

Grouper 実装は、キャッシュ設定で明示的に登録する必要があります。プログラムを用いて Data Grid を設定している場合は、以下を行います。

```
Configuration c = new ConfigurationBuilder()
    .clustering().hash().groups().enabled().addGroupouer(new KXGrouper())
    .build();
```

または、XML を使用している場合は、以下を行います。

```
<distributed-cache>
  <groups enabled="true">
    <grouper class="com.acme.KXGrouper" />
  </groups>
</distributed-cache>
```

3.3.8.7. 高度なインターフェイス

AdvancedCache には、グループ固有のメソッドが2つあります。

`getGroup(groupName)`

グループに属するキャッシュ内のすべてのキーを取得します。

`removeGroup(groupName)`

グループに属するキャッシュ内のすべてのキーを削除します。

どちらのメソッドもデータコンテナ全体とストア (存在する場合) を繰り返し処理するため、キャッシュに多くの小規模なグループが含まれる場合に処理が遅くなる可能性があります。

3.4. 散在 (SCATTERED) キャッシュ

散在 (scattered) モードは、クラスターの線形のスケーリングを可能にするため、Distribution モードと非常に似ています。データの2つのコピーを (numOwners=2 の分散モードとして) 維持することにより、単一ノードの障害を許容します。分散とは異なり、データの場所は固定されていません。同じ Consistent Hash アルゴリズムを使用してプライマリー所有者を特定しますが、バックアップコピーは前回データを書き込んだノードに保存されます。書き込みがプライマリー所有者で行われる場合、バックアップコピーは他のノードに保存されます (このコピーの正確な場所は重要ではありません)。

これには、任意の書き込みに対して単一のリモートプロシージャコール (RPC) という利点があります (配布モードには1つまたは2つの RPC が必要です) が、読み取りは常にプライマリー所有者をターゲットにする必要があります。これにより書き込みが高速になりますが、読み取り速度が遅い可能性があるため、このモードは書き込み集約型アプリケーションに適しています。

複数のバックアップコピーを保存すると、メモリー消費が若干高くなります。古いバックアップコピーを削除するために、インバリデーションメッセージがクラスターでブロードキャストされ、オーバーヘッドが発生します。これにより、非常に大きなクラスターでは分散モードのパフォーマンスが低下します (この動作は将来最適化される可能性があります)。

ノードがクラッシュすると、プライマリーコピーが失われる可能性があります。そのため、クラスターはバックアップを調整し、最後に書き込まれたバックアップコピーを見つける必要があります。このプロセスにより、状態遷移時によりネットワークトラフィックが上がります。

データのライターもバックアップであるため、トランスポートレベルでマシン/ラック/サイト ID を指定しても、クラスターは同じマシン/ラック/サイトで複数の失敗に対して回復性を備えません。

現在、トランザクションキャッシュで分散モードを使用することはできません。非同期レプリケーションはサポートされないため、代わりに非同期キャッシュ API を使用してください。機能コマンドはいずれも実装されていませんが、早い段階で追加される予定です。

キャッシュは、他のキャッシュモードと同様に設定されます。以下は宣言型設定の例です。

```
<scattered-cache name="scatteredCache" />
```

Java では、プログラムを用いてこの方法で設定できます。

```
Configuration c = new ConfigurationBuilder()
    .clustering().cacheMode(CacheMode.SCATTERED_SYNC)
    .build();
```

サーバーは通常 Hot Rod プロトコルを介してアクセスされるため、分散モードはサーバー設定では公開されません。このプロトコルは、書き込み用のプライマリ所有者を自動的に選択し (2 所有者を持つ分散モード)、クラスター内で単一の RPC が必要になります。したがって、分散キャッシュはパフォーマンス上の利点をもたらしません。

3.5. クラスター化されたキャッシュとの非同期通信

3.5.1. 非同期通信

すべてのクラスター化キャッシュモードは、`<replicated-cache/>`、`<distributed-cache>`、または `<invalidation-cache/>` 要素上で `mode="ASYNC"` 属性と非同期通信を使用するように設定できます。

非同期通信では、送信元ノードは操作のステータスについて他のノードから確認応答を受け取ることはありません。そのため、他のノードで成功したかどうかを確認する方法はありません。

非同期通信はデータに不整合を引き起こす可能性があり、結果を推論するのが難しいため、一般的に非同期通信はお勧めしません。ただし、速度が一貫性よりも重要であり、このようなケースでオプションが利用できる場合があります。

3.5.2. Asynchronous API

非同期 API を使用すると、ユーザースレッドをブロックしなくても同期通信を使用できます。

注意点が1つあります。非同期操作はプログラムの順序を保持しません。スレッドが `cache.putAsync(k, v1); cache.putAsync(k, v2)` を呼び出す場合の `k` の最終的な値は `v1` または `v2` のいずれかになります。非同期通信を使用する場合の利点は、最終的な値をあるノードで `v1` にして別のノードで `v2` にすることができないことです。

3.5.3. 非同期通信で値を返す

Cache インターフェイスは `java.util.Map` を拡張するため、`put(key, value)` や `remove(key)` などの書き込みメソッドはデフォルトで以前の値を返します。

戻り値が正しくないことがあります。

1. `Flag.IGNORE_RETURN_VALUE`、`Flag.SKIP_REMOTE_LOOKUP`、または `Flag.SKIP_CACHE_LOAD` で `AdvancedCache.withFlags()` を使用する場合。
2. キャッシュに `unreliable-return-values="true"` が設定されている場合。
3. 非同期通信を使用する場合。

4. 同じキーへの同時書き込みが複数あり、キャッシュトポロジーが変更された場合。トポロジーの変更により、Data Grid は書き込み操作を再試行します。また、再試行操作の戻り値は信頼性がありません。

トランザクションキャッシュは、3 と 4 の場合、正しい以前の値を返します。しかし、トランザクションキャッシュには gotcha: in distributed モードもあり、read-committed 分離レベルは、繰り返し可能な読み取りとして実装されます。つまり、この"double-checked locking"例は機能しません。

```
Cache cache = ...
TransactionManager tm = ...

tm.begin();
try {
    Integer v1 = cache.get(k);
    // Increment the value
    Integer v2 = cache.put(k, v1 + 1);
    if (Objects.equals(v1, v2) {
        // success
    } else {
        // retry
    }
} finally {
    tm.commit();
}
```

これを実装する適切な方法とし

て、**cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(k)** を使用します。

最適化されたロックを持つキャッシュでは、書き込みは古い以前の値を返すことができます。書き込み skew チェックでは、古い値を回避できます。

第4章 DATA GRID キャッシュの設定

Data Grid を使用すると、プログラムを使用してキャッシュするプロパティとオプションを定義できます。

宣言型設定は、Data Grid スキーマに準拠する XML ファイルを使用します。一方、プログラムによる設定は Data Grid API を使用します。

ほとんどの場合、キャッシュ定義の開始点として宣言型設定を使用します。実行時に、プログラムでキャッシュを設定して、設定を調整したり、追加のプロパティを指定したりできます。しかし、Data Grid は柔軟性があるため、宣言的、プログラムによる、またはこの2つの組み合わせを選択できます。

4.1. 宣言型設定

宣言型設定はスキーマに準拠し、XML または JSON 形式のファイルで定義されます。

以下の例は、Data Grid 設定の基本構造を示しています。

```
<infinispan>
  <!-- Defines properties for all caches within the container and optionally names a default cache. -->
  <cache-container default-cache="local">
    <!-- Configures transport properties for clustered cache modes. -->
    <!-- Specifies the default JGroups UDP stack and names the cluster. -->
    <transport stack="udp" cluster="mycluster"/>
    <!-- Configures a local cache. -->
    <local-cache name="local"/>
    <!-- Configures an invalidation cache. -->
    <invalidation-cache name="invalidation"/>
    <!-- Configures a replicated cache. -->
    <replicated-cache name="replicated"/>
    <!-- Configures a distributed cache. -->
    <distributed-cache name="distributed"/>
  </cache-container>
</infinispan>
```

参照資料

- [Data Grid 8.2 Configuration Schema](#)
- [infinispan-config-8.2.xsd](#)

4.1.1. キャッシュテンプレート

Data Grid を使用すると、キャッシュ設定の作成に使用できるテンプレートを定義できます。

たとえば、以下の設定にはキャッシュテンプレートが含まれます。

```
<infinispan>
  <!-- Specifies the cache named "local" as the default. -->
  <cache-container default-cache="local">
    <!-- Adds a cache template for local caches. -->
    <local-cache-configuration name="local-template">
      <expiration interval="10000" lifespan="10" max-idle="10"/>
    </local-cache-configuration>
  </cache-container>
</infinispan>
```

```

</local-cache-configuration>
</cache-container>
</infinispan>

```

設定テンプレートをを使用した継承

設定テンプレートは、他のテンプレートから継承して設定を拡張し、上書きすることもできます。



注記

キャッシュテンプレートの継承は階層的です。親から継承する子設定テンプレートの場合は、親テンプレートの後に追加する必要があります。

以下は、テンプレートの継承例です。

```

<infinispan>
  <cache-container>
    <!-- Defines a cache template named "base-template". -->
    <local-cache-configuration name="base-template">
      <expiration interval="10000" lifespan="10" max-idle="10"/>
    </local-cache-configuration>
    <!-- Defines a cache template named "extended-template" that inherits settings from "base-template". -->
    <local-cache-configuration name="extended-template"
      configuration="base-template">
      <expiration lifespan="20"/>
      <memory max-size="2GB" />
    </local-cache-configuration>
  </cache-container>
</infinispan>

```



重要

設定のテンプレートの継承は、**property** などの複数の値を持つ要素について加算されます。結果として作成される子設定は親設定から値がマージされます。

たとえば、親設定のマージの **<property value_x="foo" />** は子設定の **<property value_y="bar" />** とマージされ、結果として **<property value_x="foo" value_y="bar" />** になります。

4.1.2. キャッシュ設定ワイルドカード

ワイルドカードを使用してキャッシュ定義と設定テンプレートを照合することができます。

```

<infinispan>
  <cache-container>
    <!-- Uses the ``*`` wildcard to match any cache names that start with "basecache". -->
    <local-cache-configuration name="basecache*">
      <expiration interval="10500" lifespan="11" max-idle="11"/>
    </local-cache-configuration>
    <!-- Adds local caches that use the "basecache*" configuration. -->
    <local-cache name="basecache-1"/>

```

```
<local-cache name="basecache-2"/>
</cache-container>
</infinispan>
```



注記

キャッシュ名が複数のワイルドカードと一致する場合は、Data Grid は例外を出力します。

4.1.3. 複数の設定ファイル

Data Grid は、複数のファイルに設定を分割する XML 包含 (XInclude) をサポートします。

```
<infinispan xmlns:xi="http://www.w3.org/2001/XInclude">
  <cache-container default-cache="cache-1">
    <!-- Includes a local.xml file that contains a cache configuration. -->
    <xi:include href="local.xml" />
  </cache-container>
</infinispan>
```

含まれるフラグメントにスキーマを使用する場合は、**infinispan-config-fragment-12.1.xsd** スキーマを使用します。

```
<local-cache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:infinispan:config:12.1 https://infinispan.org/schemas/infinispan-
config-fragment-12.1.xsd"
  xmlns="urn:infinispan:config:12.1"
  name="mycache"/>
```



注記

Data Grid 設定は、XInclude 仕様の最小サポートのみを提供します。たとえば、**xpointer** 属性、**xi:fallback** 要素、テキスト処理、またはコンテンツネゴシエーションは使用できません。

参照資料

[XInclude 仕様](#)

4.2. DATA GRID の設定 API

プログラムで Data Grid を設定します。

グローバル設定

GlobalConfiguration クラスを使用して、Cache Manager 下のすべてのキャッシュに設定を適用します。

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .cacheContainer().statistics(true) ❶
  .metrics().gauges(true).histograms(true) ❷
  .jmx().enable() ❸
  .build();
```


- ① キャッシュマネージャーの統計を有効にします。
- ② **metrics** エンドポイント経由で統計をエクスポートします。
- ③ JMX MBean を使用して統計をエクスポートします。

参考資料:

- [org.infinispan.configuration.global.GlobalConfiguration](https://www.infinispan.org/documentation/8.2/en/configuration/global/GlobalConfiguration)
- [Global Configuration API](#).

キャッシュ設定

ConfigurationBuilder クラスを使用してキャッシュを設定します。

```
ConfigurationBuilder builder = new ConfigurationBuilder();
    builder.clustering() ①
        .cacheMode(CacheMode.DIST_SYNC) ②
        .l1().lifespan(25000L) ③
        .hash().numOwners(3) ④
        .statistics().enable(); ⑤
Configuration cfg = builder.build();
```

- ① キャッシュクラスタリングを有効にします。
- ② 分散同期キャッシュモードを使用します。
- ③ L1 キャッシュのエントリーの最大有効期間を設定します。
- ④ キャッシュエントリーごとに 3 つのクラスター全体のレプリカを設定します。
- ⑤ キャッシュ統計を有効にします。

参考資料:

- [org.infinispan.configuration.cache.ConfigurationBuilder](https://www.infinispan.org/documentation/8.2/en/configuration/cache/ConfigurationBuilder).

4.3. キャッシュプログラムによる設定

Cache Manager でキャッシュ設定を定義します。



注記

このセクションの例では、クライアントと同じ JVM で実行する Cache Manager である **EmbeddedCacheManager** を使用します。

HotRod クライアントでキャッシュをリモートで設定するには、**RemoteCacheManager** を使用します。詳細は、HotRod のドキュメントを参照してください。

新しいキャッシュインスタンスの設定

以下の例では、新しいキャッシュインスタンスを設定します。


```

EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Cache defaultCache = manager.getCache();
Configuration c = new ConfigurationBuilder().clustering() ❶
    .cacheMode(CacheMode.REPL_SYNC) ❷
    .build();

String newCacheName = "replicatedCache";
manager.defineConfiguration(newCacheName, c); ❸
Cache<String, String> cache = manager.getCache(newCacheName);

```

- ❶ 新しい設定オブジェクトを作成します。
- ❷ 分散、同期キャッシュモードを指定します。
- ❸ Configuration オブジェクトで "replicatedCache" という名前の新しいキャッシュを定義します。

既存設定からの新規キャッシュの作成

以下の例では、既存のキャッシュ設定から新しいキャッシュ設定を作成します。

```

EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Configuration dcc = manager.getDefaultCacheConfiguration(); ❶
Configuration c = new ConfigurationBuilder().read(dcc) ❷
    .clustering()
    .cacheMode(CacheMode.DIST_SYNC) ❸
    .l1()
    .lifespan(60000L) ❹
    .build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c); ❺
Cache<String, String> cache = manager.getCache(newCacheName);

```

- ❶ Cache Manager からデフォルトのキャッシュ設定を返します。この例では、**infinispan-prod.xml** はレプリケートされたキャッシュをデフォルトとして定義します。
- ❷ デフォルトのキャッシュ設定をベースとして使用する新規の Configuration オブジェクトを作成します。
- ❸ 分散、同期キャッシュモードを指定します。
- ❹ L1 ライフスパン設定を追加します。
- ❺ Configuration オブジェクトで "distributedWithL1" という名前の新しいキャッシュを定義します。

```

EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Configuration rc = manager.getCacheConfiguration("replicatedCache"); ❶
Configuration c = new ConfigurationBuilder().read(rc)
    .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .l1()
    .lifespan(60000L)
    .build();

```

```
String newCacheName = "distributedWithL1";  
manager.defineConfiguration(newCacheName, c);  
Cache<String, String> cache = manager.getCache(newCacheName);
```

- 1 "replicatedCache" という名前のキャッシュ設定をベースとして使用します。

参照資料

- [CacheManager package summary](#)
- [org.infinispan.configuration.cache.ConfigurationBuilder](#)
- [org.infinispan.manager.EmbeddedCacheManager](#)
- [HotRod Java Client Guide](#)
- [org.infinispan.client.hotrod.configuration.ConfigurationBuilder](#)
- [org.infinispan.client.hotrod.RemoteCacheManager](#)

第5章 メモリーの管理

Data Grid がエントリーを保存するデータコンテナを設定します。キャッシュエントリーのエンコーディングを指定し、データをオフヒープメモリーに保存し、エビクションまたは有効期限ポリシーを使用してアクティブなエントリーのみをメモリーに維持します。

5.1. エビクションおよびの有効期限の設定

エビクションと有効期限は、古い未使用のエントリーを削除してデータコンテナをクリーンアップするための2つの戦略です。エビクションと有効期限は同じですが、重要な違いがいくつかあります。

- ✓ エビクションを使用すると、コンテナが設定されたしきい値より大きくなったときにエントリーを削除することで、Data Grid がデータコンテナのサイズを制御できます。
- ✓ 有効期限により、エントリーの存在が制限されます。Data Grid はスケジューラーを使用して、期限切れのエントリーを定期的に削除します。有効期限が切れていても削除されていないエントリーは、アクセスするとすぐに削除されます。この場合、期限切れのエントリーに対する `get()` 呼び出しは、"null" 値を返します。
- ✓ エビクションは Data Grid ノードのローカルです。
- ✓ 有効期限は Data Grid クラスター全体で実行されます。
- ✓ エビクションと有効期限を一緒に使用することも、個別に使用できます。
- ✓ `infinispan.xml` でエビクションおよび有効期限を宣言型で設定し、エントリーのキャッシュ全体のデフォルトを適用できます。
- ✓ 特定のエントリーの有効期限設定を明示的に定義できますが、エントリーごとにエビクションを定義することはできません。
- ✓ エントリーを手動でエビクトし、有効期限を手動でトリガーできます。

5.1.1. エビクション

エビクションにより、メモリーからエントリーを削除して、データコンテナのサイズを制御できます。エビクションは、一度に1つのエントリーをデータコンテナから破棄し、そのエントリーが実行するノードに対してローカルにあります。



重要

エビクションはメモリーからエントリーを削除しますが、永続的なキャッシュストアからは削除しません。Data Grid はそれらをエビクトし、データの一貫性を防ぐために、永続キャッシュストアを設定する必要があります。

Data Grid エビクションは、以下の2つの設定に依存します。

- データコンテナの最大サイズ。
- エントリーを削除するストラテジー。

データコンテナのサイズ

Data Grid を使用すると、エントリーを Java ヒープまたはネイティブメモリー (off-heap) に保存し、データコンテナに最大サイズを設定できます。

次の 2 つの方法のいずれかでデータコンテナの最大サイズを設定します。

- エントリーの合計数 (**max-count**)。
- メモリーの最大量 (**max-size**)。
メモリー量に基づいてエビクションを実行するには、バイト単位で最大サイズを定義します。
このため、**application/x-protostream** などのバイナリストレージ形式でエントリーをエンコードする必要があります。

キャッシュエントリーのエビクト

memory を設定する場合、Data Grid はデータコンテナの現在のメモリー使用量を概算します。エントリーが追加または変更されると、Data Grid はデータコンテナの現在のメモリー使用量を最大サイズと比較します。サイズが最大値を超えると、Data Grid はエビクションを実行します。

エビクションは、最大サイズを超えるエントリーを追加するスレッドですぐに行われます。

例として以下の設定を見てみましょう。

```
<memory max-count="50"/>
```

この場合、キャッシュの合計は 50 個になります。キャッシュがエントリーの合計数に達すると、書き込み操作によって Data Grid がトリガーされ、エビクションが実行されます。

エビクションストラテジー

ストラテジーは、Data Grid のエビクションを実行する方法を制御します。エビクションは手動で実行することも、Data Grid を以下のいずれかを実行するように設定したりできます。

- 古いエントリーを削除して、新しいエントリー用の領域を作成します。
- **ContainerFullException** を出力し、新規エントリーが作成されないようにします。
例外エビクションストラテジーは、2 フェーズコミットを使用するトランザクションキャッシュでのみ動作しますが、1 フェーズコミットまたは同期の最適化とは関係しません。



注記

Data Grid には、TinyLFU と呼ばれる Least Frequently Used (LFU) キャッシュ置換アルゴリズムのバリエーションを実装する Caffeine キャッシングライブラリーが含まれています。オフヒープストレージの場合、Data Grid は LeastRecent Used (LRU) アルゴリズムのカスタム実装を使用します。

References

- [Caffeine](#)
- [永続ストレージのセットアップ](#)

5.1.1.1. Data Grid キャッシュのエントリー数の合計設定

キャッシュエントリーのデータコンテナのサイズを合計エントリー数に制限します。

手順

1. 適切なストレージ形式で Data Grid キャッシュエンコーディングを設定します。

2. Data Grid がエビクションを実行する前にキャッシュを含めることができるエントリーの合計数を指定します。
 - 宣言型: **max-count** 属性を設定します。
 - Programmatic: **maxCount()** メソッドを呼び出します。
3. Data Grid がエントリーを削除する方法を制御するようにエビクションストラテジーを設定します。
 - declarative: **when-full** 属性を設定します。
 - プログラム: **whenFull()** メソッドを呼び出します。

宣言型設定

```
<local-cache name="maximum_count">
  <encoding media-type="application/x-protostream"/>
  <memory max-count="500" when-full="REMOVE"/>
</local-cache>
```

プログラムによる設定

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
  .encoding()
  .mediaType("application/x-protostream")
  .memory()
  .maxCount(500)
  .whenFull(EvictionStrategy.REMOVE)
  .build();
```

関連情報

- [Data Grid Configuration Schema Reference](#)
- [org.infinispan.configuration.cache.MemoryConfigurationBuilder](#)

5.1.1.2. Data Grid キャッシュの最大メモリー容量の設定

キャッシュエントリーのデータコンテナのサイズを最大メモリー量に制限します。

手順

1. バイナリーエンコーディングをサポートするストレージ形式を使用するように Data Grid キャッシュを設定します。
バイナリストレージ形式を使用して、メモリーの最大量に基づいてエビクションを実行する必要があります。
2. Data Grid がエビクションを実行する前にキャッシュが使用できるメモリーの最大量をバイト単位で設定します。
 - Declarativ 宣言型: **max-size** 属性を設定します。

- プログラム: **maxSize()** メソッドを使用します。
3. 任意で、測定バイト単位を指定します。デフォルトは B(バイト単位) です。サポートされるユニットの設定スキーマを参照してください。
 4. Data Grid がエントリーを削除する方法を制御するようにエビクションストラテジーを設定します。
 - declarative: **when-full** 属性を設定します。
 - プログラム: **whenFull()** メソッドを使用します。

宣言型設定

```
<local-cache name="maximum_size">
  <encoding media-type="application/x-protostream"/>
  <memory max-size="1.5GB" when-full="REMOVE"/>
</local-cache>
```

プログラムによる設定

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
  .encoding()
  .mediaType("application/x-protostream")
  .memory()
  .maxSize("1.5GB")
  .whenFull(EvictionStrategy.REMOVE)
  .build();
```

関連情報

- [Data Grid Configuration Schema Reference](#)
- [org.infinispan.configuration.cache.EncodingConfiguration](#)
- [org.infinispan.configuration.cache.MemoryConfigurationBuilder](#)
- [キャッシュのエンコードとマーシャリング](#)

5.1.1.3. エビクションの例

エビクションをキャッシュ定義の一部として設定します。

デフォルトのメモリー設定

エビクションは有効になっていません。これはデフォルト設定です。Data Grid は、キャッシュエントリーを JVM ヒープのオブジェクトとして保存します。

```
<distributed-cache name="default_memory">
  <memory />
</distributed-cache>
```

エントリーの合計数に基くエビクション

エントリーの合計数に基づいたエビクション

Data Grid は、キャッシュエントリーを JVM ヒープのオブジェクトとして保存します。エビクションは、データコンテナに 100 エントリーがあり、Data Grid が新規エントリーを作成する要求を取得すると発生します。

```
<distributed-cache name="total_number">
  <memory max-count="100"/>
</distributed-cache>
```

エビクションベースの最大サイズ (バイト単位)

Data Grid は、キャッシュエントリーを **byte[]** 配列として保存します (例: **application/x-protostream** 形式)。

以下の例では、データコンテナのサイズが 500 MB(メガバイト) に達すると、Data Grid はエビクションを実行し、新規エントリーを作成する要求を取得します。

```
<distributed-cache name="binary_storage">
  <!-- Encodes the cache with a binary storage format. -->
  <encoding media-type="application/x-protostream"/>
  <!-- Bounds the data container to a maximum size in MB (megabytes). -->
  <memory max-size="500 MB"/>
</distributed-cache>
```

オフヒープストレージ

Data Grid は、キャッシュエントリーをバイトとしてネイティブメモリーに保存します。エビクションは、データコンテナに 100 エントリーがあり、Data Grid が新規エントリーを作成する要求を取得すると発生します。

```
<distributed-cache name="off_heap">
  <memory storage="OFF_HEAP" max-count="100"/>
</distributed-cache>
```

例外ストラテジーを使用したオフヒープストレージ

Data Grid は、キャッシュエントリーをバイトとしてネイティブメモリーに保存します。データコンテナに 100 エントリーがあり、Data Grid が新規エントリーを作成する要求を取得すると、例外が出力され、新しいエントリーは許可されません。

```
<distributed-cache name="eviction_exception">
  <memory storage="OFF_HEAP" max-count="100" when-full="EXCEPTION"/>
</distributed-cache>
```

手動エビクション

Data Grid は、キャッシュエントリーを JVM ヒープのオブジェクトとして保存します。エビクションは有効になっていませんが、**evict()** メソッドを使用して手動で実行されます。

ヒント

この設定は、パッシベーションを有効にし、エビクションを設定しない場合に警告メッセージを防ぎます。

```
<distributed-cache name="eviction_manual">
  <memory when-full="MANUAL"/>
</distributed-cache>
```

エビクションによるパッシベーション

パッシベーションは、Data Grid がエントリーをエビクトする際にキャッシュストアにデータを永続化します。パッシベーションを有効にすると、以下のようにエビクションを常に有効にする必要があります。

```
<distributed-cache name="passivation">
  <persistence passivation="true">
    <!-- Persistence configuration goes here. -->
  </persistence>
  <memory max-count="100"/>
</distributed-cache>
```

References

- [パッシベーション](#)

5.1.2. 有効期限

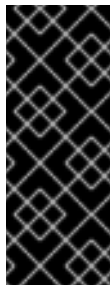
以下のいずれかの時間制限に達すると、キャッシュからエントリーが削除されます。

有効期間

エントリーが存在することができる最大時間を設定します。

最大アイドル

エントリーがアイドル状態のままになる期間を指定します。エントリーに対して操作が行われない場合は、アイドル状態になります。



重要

現在、最大アイドルの有効期限は永続キャッシュストアによるキャッシュ設定をサポートしません。

例外ベースのエビクションポリシーで有効期限を使用する場合、期限切れの状態ではなく、データコンテナのサイズに対して削除されないエントリーはキャッシュ数から削除されています。

5.1.2.1. 有効期限の仕組み

有効期限を設定する場合、Data Grid はエントリーが期限切れになるタイミングを決定するメタデータを持つキーを保存します。

- 有効期限は、**creation** タイムスタンプと **lifespan** 設定プロパティの値を使用します。
- 最大アイドルは、**last used** タイムスタンプと **max-idle** 設定プロパティの値を使用します。

Data Grid は、有効期限または最大アイドルメタデータが設定されているかどうかを確認し、値と現在の時間を比較します。

(**creation + lifespan < currentTime**) または (**lastUsed + maxIdle < currentTime**) の場合、Data Grid はエントリーが期限切れであると検出します。

有効期限は、有効期限リーパーによってエントリーがアクセスまたは検出されるたびに発生します。

たとえば、**k1** は最大アイドル時間に到達し、クライアントは **Cache.get(k1)** 要求を作成します。この場合、Data Grid はエントリーが期限切れであることを検出し、データコンテナから削除します。**Cache.get()** は **null** を返します。

Data Grid はキャッシュストアのエントリーも期限切れになりますが、ライフサイクルの有効期限のみになります。最大アイドル有効期限はキャッシュストアでは機能しません。キャッシュローダーの場合、ローダーは外部ストレージからしか読み取ることができないため、Data Grid はエントリーを期限切れにすることはできません。



注記

Data Grid は、期限切れのメタデータを、**long** プリミティブデータタイプとしてキャッシュエントリーに追加します。これにより、32 バイトだけにキーのサイズが増える可能性があります。

5.1.2.2. 有効期限リーパー

Data Grid は、定期的に行われるリーパースレッドを使用して、期限切れのエントリーを検出して削除します。有効期限により、アクセスされなくなった期限切れのエントリーが確実に削除されるようになります。

Data Grid の **ExpirationManager** インターフェイスは、有効期限リーパーを処理し、**processExpiration()** メソッドを公開します。

場合によっては、**processExpiration()** を呼び出すことで、有効期限リーパーを無効にし、エントリーを手動で期限切れにすることができます。たとえば、メンテナンススレッドが定期的に行うカスタムアプリケーションでローカルキャッシュモードを使用している場合です。



重要

クラスター化されたキャッシュモードを使用する場合は、有効期限リーパーを無効にしないでください。

キャッシュストアを使用する場合は、Data Grid は常に有効期限のリーパーを使用します。この場合、無効にすることはできません。

参照資料

- [org.infinispan.configuration.cache.ExpirationConfigurationBuilder](https://infinispan.org/docs/stable/configuration/cache/ExpirationConfigurationBuilder.html)
- [org.infinispan.expiration.ExpirationManager](https://infinispan.org/docs/stable/expiration/ExpirationManager.html)

5.1.2.3. 最大アイドルキャッシュおよびクラスター化キャッシュ

最大アイドル有効期限はキャッシュエントリーの最後のアクセス時間に依存するため、クラスター化されたキャッシュモードにはいくつかの制限があります。

有効期限が切れると、キャッシュエントリーの作成時間は、クラスター化されたキャッシュ全体で一貫した値を提供します。たとえば、**k1** の作成時間は、常にすべてのノードで同じです。

クラスター化されたキャッシュを使用した最大アイドル有効期限のため、エントリーに対する最終アクセス時間は、常にすべてのノードで同じではありません。クラスター全体で相対アクセス時間が同じになるように、Data Grid はキーへのアクセス時に、すべての所有者に touch コマンドを送信します。

Data Grid が送信する touch コマンドには、以下の考慮事項があります。

- **Cache.get()** リクエストは、すべての touch コマンドが完了するまで返されません。この同期動作により、クライアント要求のレイテンシーが長くなります。
- touch コマンドは、すべての所有者のキャッシュエントリーの recently accessed メタデータも更新します。
- scattered キャッシュモードの場合、Data Grid はプライマリーおよびバックアップ所有者のみではなく、touch コマンドをすべてのノードに送信します。

関連情報

- 最大アイドル有効期限はインバリデーションモードでは機能しません。
- クラスター化されたキャッシュでの反復は、最大アイドル時間制限を超過した期限切れのエントリーを返すことができます。この動作は、反復中にリモート呼び出しが実行されないため、パフォーマンスが向上します。また、繰り返しは期限切れのエントリーを更新しないことに注意してください。

5.1.2.4. 有効期限の例

Data Grid を設定してエントリーを期限切れにすると、以下のアイドル期間と最大アイドル時間を設定できます。

- キャッシュ内のすべてのエントリー (キャッシュ全体) **infinispan.xml** でキャッシュ全体の有効期限を設定できます。または、**ConfigurationBuilder** を使用してプログラムで設定できます。
- エントリーごとに優先され、キャッシュ全体の有効期限の値よりも優先されます。エントリーの作成時に特定のエントリーの有効期限を設定します。



注記

キャッシュエントリーの lifespan と最大アイドル時間の値を明示的に定義すると、Data Grid はキャッシュエントリーとともにクラスター全体でこれらの値を複製します。同様に、キャッシュストアを設定する場合は、Data Grid はエントリーと共に有効期限の値を永続化します。

すべてのキャッシュエントリーの有効期限の設定

2 秒後にすべてのキャッシュエントリーを期限切れにします。

```
<expiration lifespan="2000" />
```

最後のアクセス時間後にすべてのキャッシュエントリーを 1 秒後に期限切れにします。

```
<expiration max-idle="1000" />
```

interval 属性を使用して期限切れリパーを無効にし、最後のアクセス時刻の 1 秒後にエントリーを手動で期限切れにします。

```
<expiration max-idle="1000" interval="-1" />
```

最後のアクセス時間から 5 秒または 1 秒後にすべてのキャッシュエントリーの有効期限が切れるので、これは常に最初に発生します。

```
<expiration lifespan="5000" max-idle="1000" />
```

キャッシュエントリーの作成時に有効期限の設定

次の例は、キャッシュエントリーを作成するときにライフスパンと最大アイドル値を設定する方法を示しています。

```
// Use the cache-wide expiration configuration.
cache.put("pinot noir", pinotNoirPrice); ❶

// Define a lifespan value of 2.
cache.put("chardonnay", chardonnayPrice, 2, TimeUnit.SECONDS); ❷

// Define a lifespan value of -1 (disabled) and a max-idle value of 1.
cache.put("pinot grigio", pinotGrigioPrice,
    -1, TimeUnit.SECONDS, 1, TimeUnit.SECONDS); ❸

// Define a lifespan value of 5 and a max-idle value of 1.
cache.put("riesling", rieslingPrice,
    5, TimeUnit.SECONDS, 1, TimeUnit.SECONDS); ❹
```

Data Grid 設定ですべてのエントリーの有効期間値が **1000** と定義されている場合は、先行する **Cache.put()** 要求により、エントリーが期限切れになります。

- ❶ 1 秒後。
- ❷ 2 秒後。
- ❸ 最後のアクセス時間から 1 秒後。
- ❹ 最後のアクセス時間の 5 秒後または 1 秒後のいずれか早い方。

参照資料

- [Data Grid 設定スキーマ](#)
- [org.infinispan.configuration.cache.ExpirationConfigurationBuilder](#)
- [org.infinispan.expiration.ExpirationManager](#)

5.2. オフヒープメモリー

Data Grid は、キャッシュエントリーを、デフォルトで JVM ヒープメモリーに保存します。Data Grid は、オフヒープストレージを使用するように設定できます。つまり、データによって、管理された JVM メモリー領域外のネイティブメモリーに以下の利点があります。

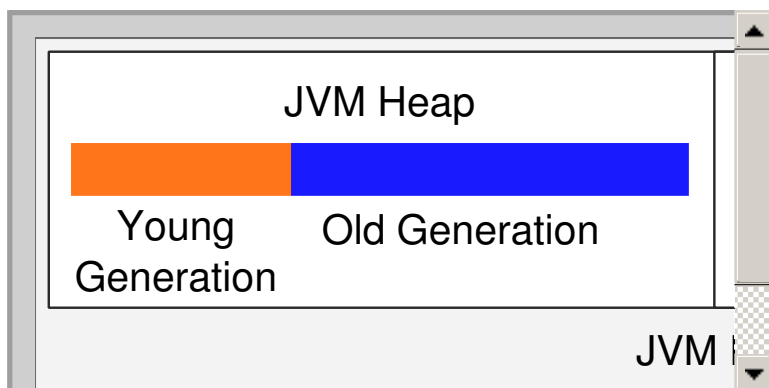
- 同じデータ量に JVM ヒープメモリーよりも少ないメモリーを使用します。

- Garbage Collector (GC) の実行を回避するために、JVM 全体のパフォーマンスを改善できます。

しかし、オフヒープストレージにはトレードオフがあります。たとえば、JVM ヒープダンプには、オフヒープメモリに保存されたエントリは表示されません。

以下の図は、Data Grid が実行されている JVM プロセスのメモリ領域を示しています。

図5.1 JVM メモリー領域



JVM ヒープメモリー

ヒープは、参照される Java オブジェクトや他のアプリケーションデータをメモリに維持するのに役立つ新しい世代と古い世代に分けられます。GC プロセスは、到達不能オブジェクトから領域を回収し、新しい生成メモリープールでより頻繁に実行します。

Data Grid がキャッシュエントリを JVM ヒープメモリーに保存すると、キャッシュへのデータ追加を開始するため、GC の実行が完了するまで時間がかかる場合があります。GC は集中的なプロセスであるため、実行が長く頻繁になると、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。

オフヒープメモリー

オフヒープメモリーは、JVM メモリー管理以外のネイティブで利用可能なシステムメモリーです。JVM メモリーの領域の図には、クラスメタデータを保持し、ネイティブメモリーから割り当てられるメタスペースメモリープールが表示されます。この図は、Data Grid キャッシュエントリを保持するネイティブメモリーのセクションも含まれています。

データのオフヒープの保存

オフヒープキャッシュにエントリを追加すると、Data Grid はネイティブメモリーをデータに動的に割り当てます。

Data Grid は、各キーのシリアル化された **byte []** を、標準の Java **HashMap** と同様のバケットにハッシュ値を持ちます。バケットには、Data Grid がオフヒープメモリーに保存するエントリの検索に使用するアドレスポインターが含まれます。

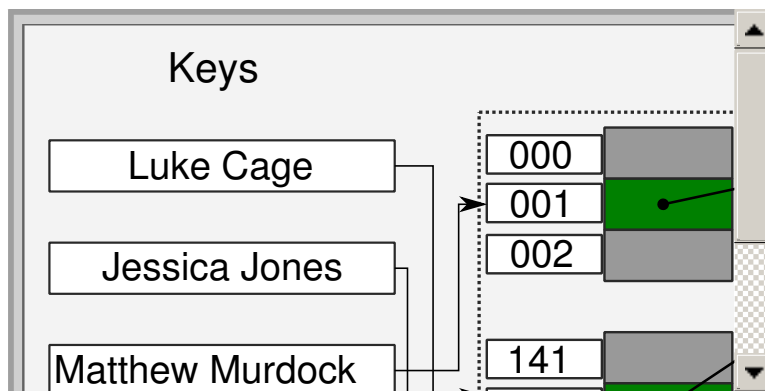


注記

Data Grid は、オブジェクトインスタンスではなく、各オブジェクトのシリアル化された **byte[]** 表現を使用して、オフヒープストレージで Java オブジェクトの等価性を決定します。

次の図は、名前付きのキーのセット、各キーのハッシュ、アドレスポインターのバケット配列、および名前と電話番号付きのエントリを示しています。

図5.2 キー用のメモリアドレスポインター



キーハッシュが衝突すると、データグリッドはエンTRIESをリンクします。**William Clay** と **Luke Cage** キーが同じハッシュを持っている場合、キーのメモリアドレスポインター ダイアグラムで、その後、キャッシュに追加する最初のエンTRIESは、バケットの最初の要素です。



注記

Data Grid はキャッシュエンTRIESをネイティブメモリーに保存する場合でも、ランタイム操作にはこれらのオブジェクトの JVM ヒープ表現が必要です。たとえば、**cache.get()** 操作は、返される前にオブジェクトをヒープメモリーに読み取ります。同様に、状態転送操作は、オブジェクトのサブセットが実行している間は、それらをヒープメモリーに保持します。

メモリーのオーバーヘッド

メモリーのオーバーヘッドは、Data Grid がエンTRIESを保存するために使用する追加のメモリーです。オフヒープストレージの場合、Data Grid はキャッシュの各エンTRIESに 25 バイトを使用します。

エビクションを使用して制限付きのオフヒープデータコンテナを作成すると、Data Grid がキャッシュ内のエンTRIESを追跡してエビクションを実行するための追加のリンクリストを作成するため、メモリーオーバーヘッドが合計 61 バイトに増加します。

データの整合性

Data Grid は、ロックの配列を使用して、オフヒープアドレス空間を保護します。ロックの数は、コア数に 2 倍になり、その後、最も近い 2 の累乗に丸められます。これにより、書き込み操作が読み取り操作をブロックしないように、**ReadWriteLock** インスタンスの配分も存在します。

5.2.1. オフヒープメモリーの使用

JVM ヒープ領域以外のネイティブメモリーにキャッシュエンTRIESを保存するように Data Grid を設定します。

手順

1. オフヒープメモリーを使用するようにキャッシュ設定を変更します。
 - 宣言型: **storage="OFF_HEAP"** 属性を **memory** 要素に追加します。
 - プログラム: **MemoryConfigurationBuilder** クラスの **storage(OFF_HEAP)** メソッドを呼び出します。
2. **max-count** または **max-size** エビクションのいずれかを設定して、キャッシュが使用できるオフヒープメモリーの容量を制限します。

サイズがバインドされたオフヒープメモリー

宣言型設定

```
<distributed-cache name="off_heap" mode="SYNC">  
  <memory storage="OFF_HEAP"  
    max-size="1.5GB"  
    when-full="REMOVE"/>  
</distributed-cache>
```

プログラムによる設定

```
ConfigurationBuilder cfg = new ConfigurationBuilder();  
  
cfg  
  .memory()  
  .storage(StorageType.OFF_HEAP)  
  .maxSize("1.5GB")  
  .whenFull(EvictionStrategy.REMOVE)  
  .build();
```

エントリーの合計数によってバインドされているオフヒープメモリー

宣言型設定

```
<distributed-cache name="off_heap" mode="SYNC">  
  <memory storage="OFF_HEAP"  
    max-count="500"  
    when-full="REMOVE"/>  
</distributed-cache>
```

プログラムによる設定

```
ConfigurationBuilder cfg = new ConfigurationBuilder();  
  
cfg  
  .memory()  
  .storage(StorageType.OFF_HEAP)  
  .maxCount(500)  
  .whenFull(EvictionStrategy.REMOVE)  
  .build();
```

関連情報

- [Data Grid Configuration Schema Reference](#)
- [org.infinispan.configuration.cache.MemoryConfigurationBuilder](#)

第6章 統計、メトリクス、および JMX の設定

Data Grid が metrics エンドポイントに、または JMX MBean を介してエクスポートする統計を有効にします。JMX MBean を登録することで、管理操作を実施することもできます。

6.1. DATA GRID 統計の有効化

Cache Manager およびキャッシュの統計をエクスポートするように Data Grid を設定します。

手順

以下のいずれかの方法で、Data Grid 統計が有効化されるように設定を変更します。

- 宣言型: **statistics="true"** 属性を追加します。
- プログラマティック: **.statistics()** メソッドを呼び出します。

宣言型

```
<!-- Enables statistics for the Cache Manager. -->
<cache-container statistics="true">
  <!-- Enables statistics for the named cache. -->
  <local-cache name="mycache" statistics="true"/>
</cache-container>
```

プログラマティック

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    //Enables statistics for the Cache Manager.
    .cacheContainer().statistics(true)
    .build();

Configuration config = new ConfigurationBuilder()
    //Enables statistics for the named cache.
    .statistics().enable()
    .build();
```

6.2. DATA GRID メトリクスの設定

Data Grid を設定して、**metrics** エンドポイント経由でゲージとヒストグラムをエクスポートします。

手順

- 必要に応じて、**metrics** 設定でゲージとヒストグラムをオンまたはオフにします。

宣言型

```
<!-- Computes and collects statistics for the Cache Manager. -->
<cache-container statistics="true">
  <!-- Exports collected statistics as gauge and histogram metrics. -->
  <metrics gauges="true" histograms="true" />
</cache-container>
```


プログラマティック

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    //Computes and collects statistics for the Cache Manager.
    .statistics().enable()
    //Exports collected statistics as gauge and histogram metrics.
    .metrics().gauges(true).histograms(true)
    .build();
```

6.2.1. Data Grid メトリックス

Data Grid は Eclipse MicroProfile Metrics API と互換性があり、ゲージおよびヒストグラムメトリックスを生成することができます。

- Data Grid メトリックスは、**vendor** スコープで提供されます。JVM に関連するメトリックスは、Data Grid サーバーの **base** スコープで提供されます。
- ゲージは、書き込み操作または JVM アップタイムの平均数 (ナノ秒) などの値を指定します。ゲージはデフォルトで有効になっています。統計を有効にすると、Data Grid はゲージを自動的に生成します。
- ヒストグラムは、読み取り、書き込み、削除の時間などの操作実行時間の詳細を提供します。Data Grid では追加の計算が必要になるため、デフォルトではヒストグラムは有効になりません。

参照資料

- [Eclipse MicroProfile Metrics](#)

6.3. JMX MBEAN を登録するための DATA GRID の設定

Data Grid は、統計の収集と管理操作の実行に使用できる JMX MBean を登録できます。JMX とは別に統計を有効にする必要があります。そうしないと、Data Grid はすべての統計属性に **0** の値を提供します。

手順

以下のいずれかの方法でキャッシュコンテナ設定を変更し、JMX を有効にします。

- 宣言型: `<jmx enabled="true" />` 要素をキャッシュコンテナに追加します。
- プログラマティック: `.jmx().enable()` メソッドを呼び出します。

宣言型

```
<cache-container>
  <jmx enabled="true" />
</cache-container>
```

プログラマティック


```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .jmx().enable()
    .build();
```

6.3.1. 複数のキャッシュマネージャーの命名

同じ JVM 上で複数の Data Grid Cache Manager が実行する場合は、競合を防ぐために各 Cache Manager を一意に特定する必要があります。

手順

- 環境内の各キャッシュマネージャーを一意に識別します。

たとえば、以下の例では、キャッシュマネージャー名として Hibernate2LC を指定します。これにより、**org.infinispan:type=CacheManager,name="Hibernate2LC"** という名前の JMX MBean が作成されます。

宣言的に

```
<cache-container name="Hibernate2LC">
  <jmx enabled="true" />
  ...
</cache-container>
```

プログラムで

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .cacheManagerName("Hibernate2LC")
    .jmx().enable()
    .build();
```

参照資料

- [GlobalConfigurationBuilder](#)
- [Data Grid 設定スキーマ](#)

6.3.2. カスタム MBean サーバーでの MBean の登録

Data Grid には、カスタム MBeanServer インスタンスに MBean を登録するのに使用できる **MBeanServerLookup** インターフェイスが含まれています。

手順

1. **getMBeanServer()** メソッドがカスタム MBeanServer インスタンスを返すように **MBeanServerLookup** の実装を作成します。
2. 以下の例のように、クラスの完全修飾名で Data Grid を設定します。

宣言的に

```
<cache-container>
  <jmx enabled="true" mbean-server-lookup="com.acme.MyMBeanServerLookup" />
</cache-container>
```

プログラムで

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .jmx().enable().mBeanServerLookup(new com.acme.MyMBeanServerLookup())
    .build();
```

参照資料

- [Data Grid 設定スキーマ](#)
- [MBeanServerLookup](#)

6.3.3. Data Grid MBean

Data Grid は、管理可能なリソースを表す JMX MBean を公開します。

org.infinispan:type=Cache

キャッシュインスタンスに使用できる属性および操作。

org.infinispan:type=CacheManager

Data Grid キャッシュやクラスターのヘルス統計など、Cache Manager で使用できる属性および操作。

使用できる JMX MBean の詳細な一覧および説明、ならびに使用可能な操作および属性については、**Data Grid JMX Components** のドキュメントを参照してください。

関連情報

- [Data Grid JMX Components](#)

第7章 永続ストレージのセットアップ

Data Grid はインメモリーデータを外部ストレージに維持し、以下のようなデータを管理する追加機能を提供します。

持続性

キャッシュストアを追加すると、不揮発性ストレージにデータを永続化できるため、再起動後も継続することができます。

ライトスルーキャッシュ

Data Grid を永続ストレージの前のキャッシュレイヤーとして設定すると、Data Grid が外部ストレージとのすべての対話を処理するため、アプリケーションのデータアクセスが簡素化されます。

データオーバーフロー

エビクションとパッシベーションの手法を使用すると、Data Grid は頻繁に使用されるデータのみをメモリー内に保持し、古いエントリーを永続ストレージに書き込みます。

7.1. DATA GRID キャッシュストア

データグリッドを永続データソースに接続するキャッシュストアは、**NonBlockingStore** インターフェイスを実装します。

7.1.1. キャッシュストアの設定

チェーンの Data Grid キャッシュにキャッシュストアを追加します。宣言的またはプログラムのいずれかです。キャッシュ読み取り操作は、データの有効な null 要素を見つけるまで、設定された順序で各キャッシュストアを確認します。書き込み操作は、読み取り専用として設定したものを除いて、すべてのキャッシュストアに影響します。

手順

1. **persistence** パラメーターを使用して、キャッシュの永続層を設定します。
2. キャッシュストアがノードに対してローカルになるか、またはクラスター全体で共有されるかどうかを設定します。
shared 属性を宣言的に、または **shared(false)** メソッドをプログラムを使用して使用します。
3. その他のキャッシュストアプロパティを随時設定します。カスタムキャッシュストアには、**プロパティ** パラメーターを含めることもできます。



注記

キャッシュストアを共有または非共有 (ローカルのみ) として設定すると、設定する必要のあるパラメーターが決まります。キャッシュストア設定でパラメーターの組み合わせが正しくないと、データ損失やパフォーマンスの問題が発生することがあります。

たとえば、キャッシュストアがノードにローカルにある場合は、起動時に状態をフェッチし、ページする方が理にかなっています。しかし、キャッシュストアが共有されている場合は、起動時に状態をフェッチしたり、ページしたりしないでください。

ローカル (非共有) のファイルストア

```
<persistence passivation="false">
  <file-store shared="false"
    path="{java.io.tmpdir}">
    <write-behind modification-queue-size="123" />
  </file-store>
</persistence>
```

共有カスタムキャッシュストア

```
<local-cache name="myCustomStore">
  <persistence passivation="false">
    <!-- Specifies the fully qualified class name of a custom cache store. -->
    <store class="org.acme.CustomStore"
      shared="true"
      segmented="true">
      <write-behind modification-queue-size="123" />
    <!-- Sets system properties for the custom cache store. -->
    <property name="myProp">${system.property}</property>
  </store>
</persistence>
</local-cache>
```

単一ファイルストア

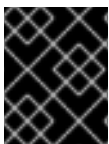
```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
  .passivation(false)
  .addSingleFileStore()
  .preload(true)
  .shared(false)
  .location(System.getProperty("java.io.tmpdir"))
  .async()
  .enabled(true)
```

参照資料

- [Data Grid 設定スキーマ](#)
- [Data Grid キャッシュストア実装](#)
- [カスタムキャッシュストアの作成](#)

7.1.2. ファイルベースのキャッシュストアのグローバル永続場所の設定

Data Grid は、グローバルファイルシステムの場所を使用して、データを永続ストレージに保存します。



重要

グローバルの永続場所は各 Data Grid インスタンスに固有の必要があります。複数のインスタンス間でデータを共有するには、共有の永続的な場所を使用します。

Data Grid サーバーは、**\$RHDG_HOME/server/data** ディレクトリーをグローバルの永続場所として使
用します。

カスタムアプリケーションと `global-state` に組み込まれたライブラリーとして Data Grid を使用してい
る場合、グローバル永続場所はデフォルトで **user.dir** システムプロパティーに設定されます。このシス
テムプロパティーは、通常、アプリケーションが開始するディレクトリーを使用します。適切な場所を
使用するようグローバル永続の場所を設定する必要があります。

宣言型設定

```
<cache-container default-cache="myCache">
  <global-state>
    <persistent-location path="example" relative-to="my.data"/>
  </global-state>
  ...
</cache-container>
```

```
new GlobalConfigurationBuilder().globalState().enable().persistentLocation("example", "my.data");
```

ファイルベースのキャッシュストアおよびグローバル永続の場所

ファイルベースのキャッシュストアを使用する場合は、必要に応じてストレージのファイルシステム
ディレクトリーを指定できます。絶対パスが宣言されない限り、ディレクトリーは常にグローバル永続
の場所と相対的になります。

たとえば、以下のようにグローバル永続の場所を設定します。

```
<global-state>
  <persistent-location path="/tmp/example" relative-to="my.data"/>
</global-state>
```

次に、以下のように **myDataStore** という名前のパスを使用する Single File キャッシュストアを設定し
ます。

```
<file-store path="myDataStore"/>
```

この場合、設定では **/tmp/example/myDataStore/myCache.dat** に Single File の File キャッシュストア
が作成されます。

グローバルの永続場所外にある絶対パスを設定しようとし、`global-state` が有効になっている場合は、
Data Grid は以下の例外を出力します。

```
ISPN000558: "The store location 'foo' is not a child of the global persistent location 'bar'"
```

参照資料

- [Data Grid 設定スキーマ](#)
- [org.infinispan.configuration.global.GlobalStateConfiguration](#)

7.1.3. パッシベーション

パッシベーションは、これらのエントリーをメモリーからエビクトする際に、ストアにエントリーを書
き込むように Data Grid を設定します。この方法により、パッシベーションは、インメモリーまたは

キャッシュストアのいずれかによって単一のエントリーのコピーのみが維持されるようになります。これにより、不要な書き込みや永続ストレージへのコストがかかる可能性があります。

アクティベーションとは、パッシベートされたエントリーにアクセスしようとする、キャッシュストアからメモリーにエントリーを復元するプロセスのことです。このため、パッシベーションを有効にすると、**CacheWriter** インターフェイスと **CacheLoader** インターフェイスの両方を実装するキャッシュストアを設定して、永続ストレージからエントリーを書き込み、読み込めるようにします。

Data Grid がキャッシュからエントリーをエビクトすると、エントリーがパッシベートされてからキャッシュストアにエントリーを保存するようキャッシュリスナーに通知します。Data Grid がエビクトされたエントリーへのアクセス要求を取得すると、キャッシュストアからエントリーをメモリーにロードし、エントリーがアクティブになるキャッシュリスナーに通知します。

注記

- パッシベーションは、Data Grid 設定の最初のキャッシュローダーを使用し、その他はすべて無視します。
- パッシベーションは以下ではサポートされません。
 - トランザクションストア。パッシベーションは、実際の Data Grid コミット境界の範囲外のストアからエントリーを作成し、削除します。
 - 共有ストア。共有キャッシュストアでは、他の所有者に対して常にストアにエントリーが存在する必要があります。そのため、エントリーを削除できないため、パッシベーションはサポートされません。

トランザクションストアまたは共有ストアでパッシベーションを有効にすると、Data Grid は例外を出力します。

7.1.3.1. パッシベーションおよびキャッシュストア

無効のパッシベーション

メモリーのデータへの書き込みにより、永続ストレージが書き込まれます。

Data Grid がデータをメモリーからエビクトする場合、永続ストレージのデータにはメモリーからエビクトされるエントリーが含まれます。このようにして、永続ストレージはインメモリーキャッシュのスーパーセットになります。

エビクションを設定しない場合、永続ストレージのデータはメモリーにデータのコピーを提供します。

有効のパッシベーション

Data Grid は、メモリーからデータをエビクトする場合にのみ、データを永続ストレージに追加します。

Data Grid がエントリーをアクティベートすると、メモリーのデータが復元され、永続ストレージからデータが削除されます。これにより、永続ストレージのメモリーおよびデータ内のデータは、データセット全体のサブセットを切り離し、2つの間の交差はありません。



注記

共有ストレージのエントリは、共有キャッシュストアの使用時に古くなる可能性があります。これは、Data Grid はアクティブ時に共有キャッシュストアからパッシブにされたエントリを削除しないため発生します。

値はメモリーで更新されますが、以前にパッシブにされたエントリは、古い値で永続ストレージに残ります。

以下の表は、一連の操作後のメモリーおよび永続ストレージのデータを示しています。

操作	無効のパッシベーション	有効のパッシベーション	共有キャッシュストアでパッシベーションが有効になっている
k1 を挿入します。	メモリー: k1 ディスク: k1	メモリー: k1 ディスク: -	メモリー: k1 ディスク: -
k2 を挿入します。	メモリー: k1, k2 ディスク: k1, k2	メモリー: k1, k2 ディスク: -	メモリー: k1, k2 ディスク: -
エビクションスレッドが実行され、k1 をエビクトします。	メモリー: k2 ディスク: k1, k2	メモリー: k2 ディスク: k1	メモリー: k2 ディスク: k1
k1 の読み取り	メモリー: k1, k2 ディスク: k1, k2	メモリー: k1, k2 ディスク: -	メモリー: k1, k2 ディスク: k1
エビクションスレッドが実行され、k2 をエビクトします。	メモリー: k1 ディスク: k1, k2	メモリー: k1 ディスク: k2	メモリー: k1 ディスク: k1, k2
k2 を削除します。	メモリー: k1 ディスク: k1	メモリー: k1 ディスク: -	メモリー: k1 ディスク: k1

7.1.4. キャッシュローダーおよびトランザクションキャッシュ

トランザクション操作をサポートする JDBC 文字列ベースのキャッシュのみ。キャッシュをトランザクションとして設定する場合は、**transactional=true** を設定して、永続ストレージのデータをメモリー内のデータと同期させる必要があります。

他のすべてのキャッシュストアでは、Data Grid はトランザクション操作でリストキャッシュローダーをエンリスト化しません。これにより、メモリー内のデータの変更中にトランザクションが正常に実行されたものの、キャッシュストアのデータへの変更が完全に適用されない場合は、データの不整合が生じる可能性があります。この場合、手動リカバリーはキャッシュストアでは機能しません。

参照資料

- [JDBC 文字列ベースのキャッシュストア](#)

7.1.5. セグメント化されたキャッシュストア

キャッシュストアは、キーがマップされるハッシュ空間セグメントにデータを編成できます。

セグメント化ストアは、一括操作の読み取りパフォーマンスを向上させます。たとえば、データ (**Cache.size**、**Cache.entrySet.stream**) をストリーミングし、キャッシュを事前読み込み、状態遷移操作を行います。

ただし、セグメントストアを使用すると、書き込み操作のパフォーマンスが低下する可能性があります。このパフォーマンス損失は、トランザクションまたはライトビハストアで実行可能なバッチ書き込み操作に対して特に該当します。このため、セグメント化されたストアを有効にする前に、書き込み操作のオーバーヘッドを評価する必要があります。書き込み操作のパフォーマンスが大幅に低下した場合、一括読み取り操作のパフォーマンスは許容できない場合があります。



重要

キャッシュストア用に設定するセグメント数は、**clustering.hash.numSegments** パラメーターと Data Grid 設定で定義するセグメントの数と一致している必要があります。

セグメント化されたキャッシュストアを追加した後に設定の **numSegments** パラメーターを変更すると、Data Grid はそのキャッシュストアからデータを読み取ることができません。

参照資料

[キーの所有者](#)

7.1.6. ファイルシステムベースのキャッシュストア

多くの場合、ファイルシステムベースのキャッシュストアは、サイズや時間の制限を超えた、メモリーからオーバーフローするデータのローカルキャッシュストアに適しています。



警告

ファイルシステムベースのキャッシュストアは、NFS、Microsoft Windows、Samba 共有などの共有ファイルシステムで使わないでください。共有ファイルシステムは、ファイルロック機能を提供しないため、データが破損する可能性があります。

同様に、共有ファイルシステムはトランザクションではありません。共有ファイルシステムとトランザクションキャッシュの使用を試みると、コミットフェーズでファイルへの書き込み時に回復不可能な障害が発生する可能性があります。

7.1.7. ライトスルー

書き込みは、メモリーへの書き込みおよびキャッシュストアへの書き込みが同期されるキャッシュ書き込みモードです。クライアントアプリケーションがキャッシュエントリを更新すると、**Cache.put()** を呼び出すと、Data Grid はキャッシュストアを更新するまで呼び出しを返しません。このキャッシュ書き込みモードを使用すると、クライアントスレッドの境界内にあるキャッシュストアに更新されます。

Write-through モードの主な利点は、キャッシュとキャッシュストアが同時に更新されるので、キャッシュストアが常にキャッシュと一貫性を持たせるようにします。

ただし、Write-Through モードは、キャッシュストアにアクセスして更新を行う必要がないため、キャッシュ操作のレイテンシーが直接追加されるため、パフォーマンスが低下する可能性があります。

キャッシュストアに Write-Behind モードを明示的に設定しない限り、Data Grid はデフォルトで Write-Through モードに設定されます。

ライトスルー設定

```
<persistence passivation="false">
  <file-store fetch-state="true"
    read-only="false"
    purge="false" path="${java.io.tmpdir}"/>
</persistence>
```

参照資料

[Write-Behind](#)

7.1.8. Write-Behind

write-Behind は、メモリーへの書き込みが同期され、キャッシュストアへの書き込みが非同期になるキャッシュ書き込みモードです。

クライアントが書き込み要求を送信すると、Data Grid はこれらの操作を変更キューに追加します。Data Grid は、呼び出しスレッドがブロックされず、操作がすぐに完了しないように、キューに参加する際に操作を処理します。

変更キューの書き込み操作の数がキューのサイズを超えた場合、Data Grid はこれらの追加操作をキューに追加します。ただし、これらの操作は、すでにキューにある Data Grid が処理するまで完了しません。

たとえば、**Cache.putAsync** を呼び出すとすぐに戻り、変更キューが満杯でない場合はすぐに Stage も完了します。変更キューが満杯であったり、Data Grid が書き込み操作のバッチを処理している場合、**Cache.putAsync** は即座に戻り、Stage は後で完了します。

write-Behind モードは、基礎となるキャッシュストアへの更新が完了するまで待機する必要がないため、Write-Through モードよりもパフォーマンス上のメリットが得られます。ただし、キャッシュストアのデータは、変更キューが処理されるまでキャッシュ内のデータと一貫性がありません。このため、Write-Behind モードは、非共有およびローカルのファイルシステムベースのキャッシュストアなど、低レイテンシーでキャッシュストアに適しています。ここでは、キャッシュへの書き込みとキャッシュストアの書き込みの間隔が可能な限り小さくなります。

ライトビハインドの設定

```
<persistence passivation="false">
  <file-store fetch-state="true"
    read-only="false"
    purge="false" path="${java.io.tmpdir}"/>
  <write-behind modification-queue-size="123"
    fail-silently="true"/>
</file-store>
</persistence>
```

上記の設定例は **fail-silently** パラメーターを使用して、キャッシュストアが利用できないか、変更キューが満杯の場合に何が起こるかを制御します。

- **fail-silently="true"** の場合、Data Grid は WARN メッセージをログに記録し、書き込み操作を拒否します。
- **fail-silently="false"** の場合、書き込み操作中にキャッシュストアが利用できないことを検知すると、Data Grid は例外を出力します。同様に、変更キューがいっぱいになると、Data Grid は例外を出力します。
Data Grid の再起動および書き込み操作が変更キューに存在すると、データ喪失が発生する可能性があります。たとえば、キャッシュストアはオフラインになりますが、キャッシュストアが利用できないことを検知するのにかかる時間、フルではないため、変更キューへの書き込み操作が追加されます。Data Grid が再起動するか、キャッシュストアがオンラインに戻る前に利用できなくなると、永続化していないため、変更キューへの書き込み操作が失われます。

参照資料

[ライトスルー](#)

7.2. キャッシュストア実装

Data Grid は、使用できるキャッシュストア実装を複数提供します。または、カスタムキャッシュストアを指定することもできます。

7.2.1. クラスターキャッシュローダー

ClusterCacheLoader は、他の Data Grid クラスターメンバーからデータを取得しますが、データは永続化されません。つまり、**ClusterCacheLoader** はキャッシュストアではありません。

ClusterCacheLoader は、状態遷移へのブロック以外の部分を提供します。**ClusterCacheLoader** は、それらの鍵がローカルノードで利用できない場合に、他のノードからキーを取得します。これは、キャッシュコンテンツを後で読み込むのと似ています。

以下のポイントは **ClusterCacheLoader** にも適用されます。

- 事前読み込みは有効になりません (**preload=true**)。
- 永続状態の取得はサポートされていません (**fetch-state=true**)。
- セグメンテーションはサポートされていません。



警告

ClusterLoader は非推奨となり、今後のリリースで削除されます。

宣言型設定

```
<persistence>
  <cluster-loader remote-timeout="500"/>
</persistence>
```

プログラムによる設定

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
  .addClusterLoader()
  .remoteCallTimeout(500);
```

参照資料

- [Data Grid 設定スキーマ](#)
- [ClusterLoader](#)
- [ClusterLoaderConfiguration](#)

7.2.2. 単一ファイルキャッシュストア

Single File キャッシュストアである **SingleFileStore** は、ファイルにデータを永続化します。また、Data Grid は、キーと値がファイルに保存される間に、キーのインメモリーインデックスも維持されます。デフォルトでは、単一ファイルキャッシュストアはセグメント化されるため、Data Grid は各セグメントに個別のファイルを作成します。

SingleFileStore はキーのインメモリーインデックス、および値の場所を保持するため、キーのサイズと数に応じて追加のメモリーが必要です。このため、**SingleFileStore** は、鍵のサイズが大きいユースケースには推奨されません。

SingleFileStore が断片化される場合もあります。値のサイズが継続的に増加している場合は、単一ファイルで利用可能な領域は使用されませんが、エントリーはファイルの最後に追加されます。このファイルで利用可能な領域は、エントリーに収まることができる場合に限り使用されます。同様に、メモリーからすべてのエントリーを削除すると、単一のファイルストアのサイズが減少したり、デフラグしたりしません。

宣言型設定

```
<persistence>
  <file-store max-entries="5000"/>
</persistence>
```

プログラムによる設定

- 埋め込みデプロイメントの場合は、以下を実行します。

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
  .addSingleFileStore()
  .maxEntries(5000);
```

- サーバーデプロイメントの場合は、以下を実行します。

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
...

ConfigurationBuilder builder = new ConfigurationBuilder();
```

```
builder
    .remoteCache("mycache")
    .configuration("<distributed-cache name=\"mycache\"><persistence><file-store max-entries=\"5000\"></persistence></distributed-cache>");
```

セグメンテーション

単一ファイルキャッシュは、セグメンテーションをサポートし、セグメントごとに別のインスタンスを作成するので、設定するパスに複数のディレクトリーが作成されます。各ディレクトリーは、データマップ先のセグメントを表す数字です。

参照資料

- [グローバルの永続化の場所の設定](#)
- [Data Grid 設定スキーマ](#)
- [SingleFileStore](#)

7.2.3. JDBC 文字列ベースのキャッシュストア

JDBC 文字列ベースのキャッシュストアである **JdbcStringBasedStore** は JDBC ドライバーを使用して、基礎となるデータベースに値を読み込みおよび保存します。

JdbcStringBasedStore はテーブル内の独自の行に各エントリーを保存し、同時読み込みのスループットを高めます。**JdbcStringBasedStore** は、**key-to-string-mapper** インターフェイスを使用して各キーを **String** オブジェクトにマッピングする単純な 1 対 1 のマッピングも使用します。

Data Grid は、プリミティブタイプを処理する **DefaultTwoWayKey2StringMapper** のデフォルト実装を提供します。

キャッシュエントリーを保存するために使用されるデータテーブルの他に、ストアはメタデータを保存するための **_META** テーブルも作成します。この表は、既存のデータベースコンテンツが現在の Data Grid バージョンおよび設定と互換性があることを確認するために使用されます。



注記

デフォルトでは、Data Grid 共有は保存されません。つまり、クラスター内のすべてのノードが更新ごとに基盤のストアに書き込まれます。基盤のデータベースに一度書き込みを行う場合、JDBC ストアを共有として設定する必要があります。

セグメンテーション

JdbcStringBasedStore はデフォルトでセグメンテーションを使用し、エントリーが属するセグメントを表すためにデータベーステーブルの列を必要とします。

7.2.3.1. 接続ファクトリー

JdbcStringBasedStore は、**ConnectionFactory** 実装に依存してデータベースに接続します。

Data Grid は以下の **ConnectionFactory** 実装を提供します。

PooledConnectionFactoryConfigurationBuilder

PooledConnectionFactoryConfiguration で設定する Agroal に基づく接続ファクトリー。

または、以下の例のように、**org.infinispan.agroal.** の接頭辞が付けられた設定プロパティを指定することもできます。

```
org.infinispan.agroal.metricsEnabled=false

org.infinispan.agroal.minSize=10
org.infinispan.agroal.maxSize=100
org.infinispan.agroal.initialSize=20
org.infinispan.agroal.acquisitionTimeout_s=1
org.infinispan.agroal.validationTimeout_m=1
org.infinispan.agroal.leakTimeout_s=10
org.infinispan.agroal.reapTimeout_m=10

org.infinispan.agroal.metricsEnabled=false
org.infinispan.agroal.autoCommit=true
org.infinispan.agroal.jdbcTransactionIsolation=READ_COMMITTED
org.infinispan.agroal.jdbcUrl=jdbc:h2:mem:PooledConnectionFactoryTest;DB_CLOSE_DELAY=-1
org.infinispan.agroal.driverClassName=org.h2.Driver.class
org.infinispan.agroal.principal=sa
org.infinispan.agroal.credential=sa
```

その後、Data Grid が **PooledConnectionFactoryConfiguration.propertyFile** でプロパティファイルを使用するように設定します。



注記

サブレットコンテナのデプロイメントではなく、スタンドアロンデプロイメントで **PooledConnectionFactory** を使用する必要があります。

ManagedConnectionFactoryConfigurationBuilder

アプリケーションサーバーなどの管理環境で利用できる接続ファクトリー。この接続ファクトリーは JNDI ツリーで設定可能な場所を確認し、接続管理を **DataSource** に委譲できます。

SimpleConnectionFactoryConfigurationBuilder

呼び出しごとにデータベース接続を作成する接続ファクトリー。この接続ファクトリーは、テスト環境または開発環境にのみ使用してください。

参照資料

- [Agroal](#)
- [ConnectionFactoryConfigurationBuilder](#)
- [PooledConnectionFactoryConfigurationBuilder](#)
- [ManagedConnectionFactoryConfigurationBuilder](#)
- [SimpleConnectionFactoryConfigurationBuilder](#)

7.2.3.2. JDBC 文字列ベースのキャッシュストア設定

プログラムまたは宣言型で **JdbcStringBasedStore** を設定できます。

目次

- **PooledConnectionFactory** の使用

```
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:12.1" shared="true">
    <connection-pool connection-url="jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1"
      username="sa"
      driver="org.h2.Driver"/>
    <string-keyed-table drop-on-exit="true"
      prefix="ISPN_STRING_TABLE">
      <id-column name="ID_COLUMN" type="VARCHAR(255)" />
      <data-column name="DATA_COLUMN" type="BINARY" />
      <timestamp-column name="TIMESTAMP_COLUMN" type="BIGINT" />
      <segment-column name="SEGMENT_COLUMN" type="INT" />
    </string-keyed-table>
  </string-keyed-jdbc-store>
</persistence>
```

- **ManagedConnectionFactory** の使用

```
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:12.1" shared="true">
    <data-source jndi-url="java:/StringStoreWithManagedConnectionTest/DS" />
    <string-keyed-table drop-on-exit="true"
      create-on-start="true"
      prefix="ISPN_STRING_TABLE">
      <id-column name="ID_COLUMN" type="VARCHAR(255)" />
      <data-column name="DATA_COLUMN" type="BINARY" />
      <timestamp-column name="TIMESTAMP_COLUMN" type="BIGINT" />
      <segment-column name="SEGMENT_COLUMN" type="INT"/>
    </string-keyed-table>
  </string-keyed-jdbc-store>
</persistence>
```

プログラムによる設定

- **PooledConnectionFactory** の使用

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
  .fetchPersistentState(false)
  .ignoreModifications(false)
  .purgeOnStartup(false)
  .shared(true)
  .table()
    .dropOnExit(true)
    .createOnStart(true)
    .tableNamePrefix("ISPN_STRING_TABLE")
    .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
    .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
    .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INT")
  .connectionPool()
```

```
.connectionUrl("jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1")
.username("sa")
.driverClass("org.h2.Driver");
```

- **ManagedConnectionFactory** の使用

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .shared(true)
    .table()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_STRING_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
        .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
        .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INT")
    .dataSource()
        .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
```

参照資料

- [JDBC キャッシュストア設定スキーマ](#)
- [JdbcStringBasedStore](#)
- [JdbcStringBasedStoreConfiguration](#)

7.2.4. JPA キャッシュストア

JPA (Java Persistence API) キャッシュストア、**JpaStore** は正式なスキーマを使用してデータを永続化します。その後、他のアプリケーションは永続ストレージから読み取れ、Data Grid からデータを読み込むことができます。ただし、他のアプリケーションは、Data Grid と同時に永続ストレージを使用しないでください。

JpaStore を使用する場合は、以下を考慮する必要があります。

- キーはエンティティの ID である必要があります。値はエンティティオブジェクトにする必要があります。
- 1つの **@Id** または **@EmbeddedId** アノテーションのみが許可されます。
- **@GeneratedValue** アノテーションを使用した自動生成 ID はサポートされません。
- すべてのエントリは **immortal** として保存されます。
- **JpaStore** はセグメンテーションをサポートしていません。

宣言型設定

```
<local-cache name="vehicleCache">
  <persistence passivation="false">
```



```

<jpa-store xmlns="urn:infinispan:config:store:jpa:12.1"
  persistence-unit="org.infinispan.persistence.jpa.configurationTest"
  entity-class="org.infinispan.persistence.jpa.entity.Vehicle">
/>
</persistence>
</local-cache>

```

パラメーター	説明
persistence-unit	JPA エンティティークラスを含む JPA 設定ファイル persistence.xml で JPA 永続性ユニット名を指定します。
entity-class	このキャッシュに保存されることが想定される完全修飾 JPA エンティティークラス名を指定します。1 つのクラスのみが許可されます。

プログラムによる設定

```

Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(JpaStoreConfigurationBuilder.class)
    .persistenceUnitName("org.infinispan.loaders.jpa.configurationTest")
    .entityClass(User.class)
    .build();

```

パラメーター	説明
persistenceUnitName	JPA エンティティークラスを含む JPA 設定ファイル persistence.xml で JPA 永続性ユニット名を指定します。
entityClass	このキャッシュに保存されることが想定される完全修飾 JPA エンティティークラス名を指定します。1 つのクラスのみが許可されます。

参照資料

- [JPA キャッシュストア設定スキーマ](#)
- [JpaStore](#)
- [JpaStoreConfiguration](#)

7.2.4.1. JPA キャッシュストア使用例

このセクションでは、JPA キャッシュストアを使用する例を紹介します。

前提条件

- JPA エンティティをマーシャリングするように Data Grid を設定します。デフォルトでは、

DataGrid は Java オブジェクトのマーシャリングに ProtoStream を使用します。JPA エンティティをマーシャリングするには、**.proto** スキーマおよびマーシャラーを **SerializationContext** に登録する **SerializationContextInitializer** 実装を作成する必要があります。

手順

1. 永続性ユニット "myPersistenceUnit" を **persistence.xml** で定義します。

```
<persistence-unit name="myPersistenceUnit">
  <!-- Persistence configuration goes here. -->
</persistence-unit>
```

2. ユーザーエンティティクラスを作成します。

```
@Entity
public class User implements Serializable {
    @Id
    private String username;
    private String firstName;
    private String lastName;

    ...
}
```

3. JPA キャッシュストアで "usersCache" という名前のキャッシュを設定します。
キャッシュ usersCache を設定して JPA キャッシュストアを使用するように設定できます。これにより、データをキャッシュに配置すると、JPA 設定を基にデータがデータベースに永続化されます。

```
EmbeddedCacheManager cacheManager = ...;

Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(JpaStoreConfigurationBuilder.class)
    .persistenceUnitName("org.infinispan.loaders.jpa.configurationTest")
    .entityClass(User.class)
    .build();
cacheManager.defineCache("usersCache", cacheConfig);

Cache<String, User> usersCache = cacheManager.getCache("usersCache");
usersCache.put("raytsang", new User(...));
```

- JPA キャッシュストアを使用するキャッシュは、以下の例のように、1 種類のデータのみを保存できます。

```
Cache<String, User> usersCache = cacheManager.getCache("myJPACache");
// Cache is configured for the User entity class
usersCache.put("username", new User());
// Cannot configure caches to use another entity class with JPA cache stores
Cache<Integer, Teacher> teachersCache = cacheManager.getCache("myJPACache");
teachersCache.put(1, new Teacher());
// The put request does not work for the Teacher entity class
```

- **@EmbeddedId** アノテーションでは、以下の例のように複合キーを使用できます。

```

@Entity
public class Vehicle implements Serializable {
    @EmbeddedId
    private VehicleId id;
    private String color; ...
}

@Embeddable
public class VehicleId implements Serializable
{
    private String state;
    private String licensePlate;
    ...
}

```

関連資料

- [キャッシュのエンコードとマーシャリング](#)

7.2.5. リモートキャッシュストア

リモートキャッシュストア **RemoteStore** は Hot Rod プロトコルを使用して Data Grid クラスタにデータを保存します。

以下は、"one" および "two" という名前の 2 つの Data Grid Server インスタンスの "mycache" という名前のキャッシュにデータを保存する **RemoteStore** 設定例になります。



注記

リモートキャッシュストアを共有として設定している場合は、事前にデータを読み込みません。つまり、設定の **shared="true"** の場合は、**preload="false"** を設定する必要があります。

宣言型設定

```

<persistence>
  <remote-store xmlns="urn:infinispan:config:store:remote:12.1" cache="mycache" raw-
values="true">
    <remote-server host="one" port="12111" />
    <remote-server host="two" />
    <connection-pool max-active="10" exhausted-action="CREATE_NEW" />
    <write-behind />
  </remote-store>
</persistence>

```

プログラムによる設定

```

ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence().addStore(RemoteStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .remoteCacheName("mycache")

```

```

        .rawValues(true)
    .addServer()
        .host("one").port(12111)
    .addServer()
        .host("two")
    .connectionPool()
        .maxActive(10)
    .exhaustedAction(ExhaustedAction.CREATE_NEW)
    .async().enable();

```

セグメンテーション

RemoteStore はセグメンテーションをサポートし、セグメントごとにキーとエントリを公開できるため、一括操作がより効率的になります。ただし、セグメンテーションは Data Grid Hot Rod プロトコルバージョン 2.3 以降でのみ利用できます。



警告

RemoteStore のセグメンテーションを有効にすると、Data Grid サーバー設定で定義したセグメントの数が使用されます。

ソースキャッシュがセグメント化され、**RemoteStore** 以外のセグメントを使用する場合は、一括操作のために誤った値が返されます。この場合は、**RemoteStore** のセグメンテーションを無効にする必要があります。

参照資料

- [リモートキャッシュストア設定スキーマ](#)
- [RemoteStore](#)
- [RemoteStoreConfigurationBuilder](#)

7.2.6. RocksDB キャッシュストア

RocksDB は、同時環境のパフォーマンスと信頼性が高いキー/ファイルシステムベースのストレージを提供します。

RocksDB キャッシュストア **RocksDBStore** は、2つのデータベースを使用します。1つのデータベースは、データをメモリーに主要なキャッシュストアを提供します。他のデータベースには、Data Grid がメモリーから失効するエントリを保持します。

宣言型設定

```

<local-cache name="vehicleCache">
  <persistence>
    <rocksdb-store xmlns="urn:infinispan:config:store:rocksdb:12.1" path="rocksdb/data">
      <expiration path="rocksdb/expired"/>
    </rocksdb-store>
  </persistence>
</local-cache>

```

プログラムによる設定

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(RocksDBStoreConfigurationBuilder.class)
    .build();
EmbeddedCacheManager cacheManager = new DefaultCacheManager(cacheConfig);
```

```
Cache<String, User> usersCache = cacheManager.getCache("usersCache");
usersCache.put("raysang", new User(...));
```

```
Properties props = new Properties();
props.put("database.max_background_compactions", "2");
props.put("data.write_buffer_size", "512MB");
```

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(RocksDBStoreConfigurationBuilder.class)
    .location("rocksdb/data")
    .expiredLocation("rocksdb/expired")
    .properties(props)
    .build();
```

表7.1 RocksDBStore 設定パラメーター

パラメーター	説明
location	プライマリーキャッシュストアを提供する RocksDB データベースへのパスを指定します。ロケーションを設定しない場合には、自動的に作成されます。パスはグローバル永続の場所と相対的である必要があります。
expiredLocation	期限切れのデータのキャッシュストアを提供する RocksDB データベースへのパスを指定します。ロケーションを設定しない場合には、自動的に作成されます。パスはグローバル永続の場所と相対的である必要があります。
expiryQueueSize	期限切れのエントリーのためにインメモリーキューのサイズを設定します。キューがサイズに達すると、Data Grid は期限切れの RocksDB キャッシュストアにフラッシュします。
clearThreshold	RocksDB データベースを削除し、再初期化する (re-init) 前にエントリーの最大数を設定します。サイズが小さいキャッシュストアの場合、すべてのエントリーを繰り返し処理し、各エントリーを個別に削除すると、より高速な方法が得られます。

RocksDB チューニングパラメーター

以下の RocksDB チューニングパラメーターを指定することもできます。

- **compressionType**

- **blockSize**
- **cacheSize**

RocksDB 設定プロパティ

任意で、以下のように設定でプロパティを設定します。

- RocksDB データベースを調整およびチューニングするために、プロパティの前に **database** 接頭辞を追加します。
- プロパティの前に **data** 接頭辞を追加して、RocksDB がデータを格納する列ファミリーを設定します。

```
<property name="database.max_background_compactions">2</property>
<property name="data.write_buffer_size">64MB</property>
<property
name="data.compression_per_level">kNoCompression:kNoCompression:kNoCompression:kSnappyCo
mpression:kZSTD:kZSTD</property>
```

セグメンテーション

RocksDBStore はセグメンテーションをサポートし、セグメントごとに個別の列ファミリーを作成します。セグメント化された RocksDB キャッシュストアは、ルックアップのパフォーマンスと反復が改善されますが、書き込み操作のパフォーマンスに若干低下します。



注記

数百のセグメントを複数設定しないでください。RocksDB は、列ファミリーの数を無制限に指定するように設計されていません。セグメントが多すぎると、キャッシュストアの起動時間が大幅に増加します。

参照資料

- [RocksDB cache store configuration schema](#)
- [RocksDBStore](#)
- [RocksDBStoreConfiguration](#)
- [rocksdb.org](#)
- [RocksDB Tuning Guide](#)

7.2.7. Soft-Index ファイルストア

soft-Index ファイルキャッシュストア **SoftIndexFileStore** は、ローカルファイルベースのストレージを提供します。

SoftIndexFileStore は、Java のソフト参照を使用してインメモリーでキャッシュされる **B+ Tree** のバリエーションを使用する Java 実装です。**Index** と呼ばれる **B+ Tree** は、キャッシュストアが再起動するたびに削除および再ビルドされる単一のファイルにオフロードされます。

SoftIndexFileStore は、データを単一のファイルではなく一連のファイルに保存します。ファイルの使用量が 50% を下回ると、ファイルのエントリが別のファイルに上書きされ、ファイルは削除されます。

SoftIndexFileStore は、add-only メソッドで記述されたファイルのセットでデータを永続化します。このため、従来の磁気ディスクで **SoftIndexFileStore** を使用する場合は、エントリーのバーストを書き込むときにシークする必要はありません。

SoftIndexFileStore のほとんどの構造はバインドされるため、メモリー不足の例外は危険にさらされません。同時オープンファイルの制限を設定することもできます。

デフォルトでは、**Index** のノードのサイズは 4096 バイトに制限されます。このサイズはキーの長さも制限します。シリアル化されたキーの長さが正確になります。このため、ノードのサイズ (15 バイト) よりも長い鍵を使用することはできません。さらに、キーの長さは "short" として保存され、キーの長さは 32767 バイトに制限されます。**SoftIndexFileStore** は、シリアル化後にキーが長い場合に例外を出力します。

SoftIndexFileStore は期限切れのエントリーを検出できず、ファイルシステムで領域が過剰に使用されてしまう可能性があります。



注記

AdvancedStore.purgeExpired() は **SoftIndexFileStore** には実装されていません。

宣言型設定

```
<persistence>
  <soft-index-file-store xmlns="urn:infinispan:config:store:soft-index:12.1">
    <index path="testCache/index" />
    <data path="testCache/data" />
  </soft-index-file-store>
</persistence>
```

プログラムによる設定

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
  .addStore(SoftIndexFileStoreConfigurationBuilder.class)
  .indexLocation("testCache/index");
  .dataLocation("testCache/data")
```

セグメンテーション

soft-Index ファイルキャッシュは、サポートセグメントとセグメントごとに別のインスタンスを作成するので、設定するパスに複数のディレクトリーが作成されます。各ディレクトリーは、データマップ先のセグメントを表す数字です。

参照資料

- [Soft-Index File cache store configuration schema](#)
- [SoftIndexFileStore](#)
- [SoftIndexFileStoreConfiguration](#)

7.2.8. カスタムキャッシュストアの実装

Data Grid の永続 SPI を使用してカスタムキャッシュストアを作成できます。

7.2.8.1. Data Grid 永続性 SPI

Data Grid Service Provider Interface (SPI) は、**NonBlockingStore** インターフェイスを介して外部ストレージへの読み書き操作を有効にし、以下の機能を持ちます。

JCache 準拠のベンダー間での移植性

Data Grid は、ブロッキングコードを処理するアダプターを使用して、**NonBlockingStore** インターフェイスと **JSR-107** JCache 仕様間の互換性を維持します。

簡素化されたトランザクション統合

Data Grid はロックを自動的に処理するため、実装は永続ストアへの同時アクセスを調整する必要はありません。使用するロックモードによっては、通常、同じキーへの同時書き込みは発生しません。ただし、永続ストレージ上の操作は複数のスレッドから発信され、この動作を許容する実装を作成することを想定する必要があります。

並列反復

Data Grid を使用すると、複数のスレッドを持つ永続ストアのエントリを繰り返すことができます。

シリアル化の減少による CPU 使用率の削減

Data Grid は、リモートで送信できるシリアル化された形式で保存されたエントリを公開します。このため、Data Grid は永続ストレージから取得されたエントリをデシリアライズし、ネットワークに書き込む際に再びシリアライズする必要はありません。

参照資料

- [永続性 SPI](#)
- [NonBlockingStore](#)
- [JSR-107](#)

7.2.8.2. キャッシュストアの作成

NonBlockingStore インターフェイスを実装してカスタムキャッシュストアを作成します。

1. 適切な Data Grid の永続 SPI を実装します。
2. カスタム設定がある場合は、ストアクラスに **@ConfiguredBy** アノテーションを付けます。
3. 必要に応じてカスタムキャッシュストア設定およびビルダーを作成します。
 - a. **AbstractStoreConfiguration** および **AbstractStoreConfigurationBuilder** を拡張します。
 - b. オプションで以下のアノテーションをストア設定クラスに追加し、カスタム設定ビルダーが XML からキャッシュストア設定を解析できるようにします。
 - **@ConfigurationFor**
 - **@BuiltBy**
これらのアノテーションを追加しないと、**CustomStoreConfigurationBuilder** は **AbstractStoreConfiguration** で定義された共通のストア属性を解析し、追加の要素は無視されます。



注記

設定が **@ConfigurationFor** アノテーションを宣言しない場合は、Data Grid がキャッシュを初期化する際に警告メッセージがログに記録されません。

7.2.8.3. カスタムストアを使用するための Data Grid の設定

カスタムキャッシュストア実装を作成したら、使用する Data Grid を設定します。

宣言型設定

```
<local-cache name="customStoreExample">
  <persistence>
    <store class="org.infinispan.persistence.dummy.DummyInMemoryStore" />
  </persistence>
</local-cache>
```

プログラムによる設定

```
Configuration config = new ConfigurationBuilder()
    .persistence()
    .addStore(CustomStoreConfigurationBuilder.class)
    .build();
```

7.2.8.4. カスタムキャッシュストアのデプロイ

カスタムキャッシュストアを JAR ファイルにパッケージ化し、以下のように Data Grid サーバーにデプロイできます。

1. カスタムキャッシュストア実装を JAR ファイルにパッケージ化します。
2. JAR ファイルを Data Grid サーバーの **server/lib** ディレクトリーに追加します。

7.3. キャッシュストア間の移行

Data Grid は、あるキャッシュストアから別のキャッシュストアにデータを移行するユーティリティを提供します。

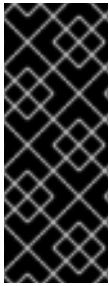
7.3.1. キャッシュストアマイグレーション

Data Grid は、最新の Data Grid キャッシュストア実装のデータを再作成する **StoreMigrator.java** ユーティリティを提供します。

StoreMigrator は以前のバージョンの Data Grid のキャッシュストアを取得し、キャッシュストア実装をターゲットとして使用します。

StoreMigrator を実行すると、**EmbeddedCacheManager** インターフェイスを使用して定義したキャッシュストアタイプでターゲットキャッシュが作成されます。**StoreMigrator** は、ソースストアからメモリーにエントリーを読み込み、それらをターゲットキャッシュに配置します。

StoreMigrator を使用すると、あるタイプのキャッシュストアから別のストアにデータを移行することもできます。たとえば、JDBC String ベースのキャッシュストアから Single File キャッシュストアに移行することができます。



重要

StoreMigrator は、セグメント化されたキャッシュストアから以下にデータを移行できません。

- 非セグメント化されたキャッシュストア。
- セグメント数が異なるセグメント化されたキャッシュストア。

7.3.2. Store Migrator の取得

StoreMigrator は、Data Grid ツールライブラリー **infinispan-tools** の一部として利用でき、Maven リポジトリに含まれます。

手順

- **StoreMigrator** の **pom.xml** を以下のように設定します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.infinispan.example</groupId>
  <artifactId>jdbc-migrator-example</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-tools</artifactId>
    </dependency>
    <!-- Additional dependencies -->
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.2.1</version>
        <executions>
          <execution>
            <goals>
              <goal>java</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <mainClass>org.infinispan.tools.store.migrator.StoreMigrator</mainClass>
          <arguments>
            <argument>path/to/migrator.properties</argument>
          </arguments>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

```

    </plugin>
  </plugins>
</build>
</project>

```

7.3.3. ストア移行の設定

ソースおよびターゲットのキャッシュストアのプロパティを **migrator.properties** ファイルに設定します。

手順

1. **migrator.properties** ファイルを作成します。
2. ソースキャッシュストアを **migrator.properties** に設定します。
 - a. 以下の例にあるように、すべての設定プロパティの先頭に **source.** を追加します。

```

source.type=SOFT_INDEX_FILE_STORE
source.cache_name=myCache
source.location=/path/to/source/sifs

```

3. **migrator.properties** でターゲットキャッシュストアを設定します。
 - a. 以下の例のように、すべての設定プロパティの先頭に **target.** を付けます。

```

target.type=SINGLE_FILE_STORE
target.cache_name=myCache
target.location=/path/to/target/sfs.dat

```

7.3.3.1. 移行プロパティの保存

ソースおよびターゲットのキャッシュストアを **StoreMigrator** プロパティで設定します。

表7.2 キャッシュストアタイププロパティ

プロパティ	説明	必須/オプション
-------	----	----------

プロパティ	説明	必須/オプション
type	<p>ソースまたはターゲットのキャッシュストアタイプのタイプを指定します。</p> <p>.type=JDBC_STRING</p> <p>.type=JDBC_BINARY</p> <p>.type=JDBC_MIXED</p> <p>.type=LEVELDB</p> <p>.type=ROCKSDB</p> <p>.type=SINGLE_FILE_STORE</p> <p>.type=SOFT_INDEX_FILE_STORE</p> <p>.type=JDBC_MIXED</p>	必須

表7.3 一般的なプロパティ

プロパティ	説明	値の例	必須/オプション
cache_name	ストアがバックアップするキャッシュに名前を付けます。	.cache_name=myCache	必須
segment_count	<p>セグメンテーションを使用できるターゲットキャッシュストアのセグメント数を指定します。</p> <p>セグメント数は、Data Grid 設定の clustering.hash.num Segments と一致する必要があります。</p> <p>つまり、キャッシュストアのセグメント数は、対応するキャッシュのセグメント数と一致する必要があります。セグメントの数が同一でない場合、Data Grid はキャッシュストアからデータを読み込めません。</p>	.segment_count=256	Optional

表7.4 JDBC プロパティ

プロパティ	説明	必須/オプション
dialect	基礎となるデータベースのダイアレクトを指定します。	必須
version	<p>ソースキャッシュストアのマーシャラーバージョンを指定します。以下のいずれかの値を設定します。</p> <p>* Data Grid 7.2.x の場合は 8</p> <p>* Data Grid 7.3.x の場合は 9</p> <p>* Data Grid 8.x の場合は 10</p>	<p>ソースストアにのみ必要です。</p> <p>例: source.version=9</p>
marshaller.class	カスタムマーシャークラスを指定します。	カスタムマーシャラーを使用する場合に必要です。
marshaller.externalizers	[id]:<Externalizer class> 形式で読み込むカスタム AdvancedExternalizer 実装のコンマ区切りリストを指定します。	Optional
connection_pool.connection_url	JDBC 接続 URL を指定します。	必須
connection_pool.driver_classes	JDBC ドライバーのクラスを指定します。	必須
connection_pool.username	データベースユーザー名を指定します。	必須
connection_pool.password	データベースユーザー名のパスワードを指定します。	必須
db.major_version	データベースのメジャーバージョンを設定します。	Optional
db.minor_version	データベースのマイナーバージョンを設定します。	Optional
db.disable_upsert	データベース upsert を無効にします。	Optional
db.disable_indexing	テーブルインデックスが作成されるかどうかを指定します。	Optional
table.string.table_name_prefix	テーブル名の追加接頭辞を指定します。	Optional

プロパティ	説明	必須/オプション
table.string. <id data timestamp>.name	列名を指定します。	必須
table.string. <id data timestamp>.type	列タイプを指定します。	必須
key_to_string_mapper	TwoWayKey2StringMapper クラスを指定します。	Optional

注記

Binary キャッシュストアから古い Data Grid バージョンの移行には、以下のプロパティで **table.string.*** を **table.binary.*** に変更します。

- **source.table.binary.table_name_prefix**
- **source.table.binary.<id|data|timestamp>.name**
- **source.table.binary.<id|data|timestamp>.type**

```
# Example configuration for migrating to a JDBC String-Based cache store
target.type=STRING
target.cache_name=myCache
target.dialect=POSTGRES
target.marshaller.class=org.example.CustomMarshaller
target.marshaller.externalizers=25:Externalizer1,org.example.Externalizer2
target.connection_pool.connection_url=jdbc:postgresql:postgres
target.connection_pool.driver_class=org.postgresql.Driver
target.connection_pool.username=postgres
target.connection_pool.password=redhat
target.db.major_version=9
target.db.minor_version=5
target.db.disable_upsert=false
target.db.disable_indexing=false
target.table.string.table_name_prefix=tablePrefix
target.table.string.id.name=id_column
target.table.string.data.name=datum_column
target.table.string.timestamp.name=timestamp_column
target.table.string.id.type=VARCHAR
target.table.string.data.type=bytea
target.table.string.timestamp.type=BIGINT
target.key_to_string_mapper=org.infinispan.persistence.keymappers.
DefaultTwoWayKey2StringMapper
```

表7.5 RocksDB プロパティ

プロパティ	説明	必須/オプション
-------	----	----------

プロパティ	説明	必須/オプション
location	データベースディレクトリーを設定します。	必須
圧縮	使用する圧縮タイプを指定します。	Optional

```
# Example configuration for migrating from a RocksDB cache store.
source.type=ROCKSDB
source.cache_name=myCache
source.location=/path/to/rocksdb/database
source.compression=SNAPPY
```

表7.6 SingleFileStore プロパティ

プロパティ	説明	必須/オプション
location	キャッシュストア .dat ファイルが含まれるディレクトリーを設定します。	必須

```
# Example configuration for migrating to a Single File cache store.
target.type=SINGLE_FILE_STORE
target.cache_name=myCache
target.location=/path/to/sfs.dat
```

表7.7 SoftIndexFileStore プロパティ

プロパティ	説明	値
必須/オプション	location	データベースディレクトリーを設定します。
必須	index_location	データベースインデックスディレクトリーを設定します。

```
# Example configuration for migrating to a Soft-Index File cache store.
target.type=SOFT_INDEX_FILE_STORE
target.cache_name=myCache
target.location=path/to/sifs/database
target.index_location=path/to/sifs/index
```

7.3.4. キャッシュストアの移行

StoreMigrator を実行して、あるキャッシュストアから別のキャッシュストアにデータを移行します。

前提条件

- **infinispan-tools.jar** を取得します。
- ソースおよびターゲットのキャッシュストアを設定する **migrator.properties** ファイルを作成します。

手順

- ソースから **infinispan-tools.jar** をビルドする場合は、以下を実行します。
 1. JDBC ドライバーなどのソースおよびターゲットのデータベースの **infinispan-tools.jar** および依存関係をクラスパスに追加します。
 2. **migrator.properties** ファイルを **StoreMigrator** の引数として指定します。
- Maven リポジトリから **infinispan-tools.jar** をプルする場合は、以下のコマンドを実行します。
mvn exec:java

第8章 パーティション処理の設定

8.1. パーティション処理

Data Grid クラスターは、データが保存される複数のノードから構築されます。ノード障害がある状態でデータを失わないようにするために、Data Grid は複数のノードにわたり、Data Grid parlance の同じデータ -- キャッシュエントリーをコピーします。このレベルのデータ冗長性は、**numOwners** 設定属性を介して設定され、同時にクラッシュするノードが **numOwners** 未満である限り、DataGrid に使用可能なデータのコピーがあることを保証します。

ただし、**numOwners** を超えるノードがクラスターから消えるという壊滅的な状況が発生する可能性があります。

スプリットブレイン

たとえば、ルーターのクラッシュにより、クラスターは2つ以上のパーティションに分割されるか、個別に動作するサブクラスターに分割されます。このような場合、異なるパーティションから読み取り/書き込みを行う複数のクライアントが、同じキャッシュエントリーの異なるバージョンを参照し、多くのアプリケーションで問題があります。冗長ネットワークや [IP ボンディング](#) など、スプリットブレインが発生する可能性を軽減する方法があることに注意してください。これらは、問題の発生期間のみを縮小します。

numOwners ノードを順番にクラッシュ

numOwners ノードは迅速な成功でクラッシュし、Data Grid にクラッシュ間の状態を適切にリバランスする時間がない場合、結果は部分的なデータの損失になります。

このセクションで説明されているパーティション処理機能により、スプリットブレインが発生したときに、キャッシュで実行できる操作を設定できます。Data Grid は、複数のパーティション処理ストラテジーを提供します。これは、Brewer の [CAP 定理](#) の観点から、パーティションが存在する場合に可用性または一貫性が犠牲になるかどうかを決定します。以下は、指定されるストラテジーの一覧です。

ストラテジー	詳細	CAP
DENY_READ_WRITES	パーティションに特定のセグメントのすべての所有者がない場合、そのセグメント内のすべてのキーの読み取りと書き込みの両方が拒否されます。	一貫性
ALLOW_READS	このパーティションに存在する場合は特定のキーの読み取りを許可しますが、このパーティションにセグメントのすべての所有者が含まれている場合にのみ書き込みを許可します。これは、このパーティションで利用可能なものの、クライアントアプリケーションの観点から見えても決定論的ではない場合に一部のエントリーが読み取り可能であるため、一貫したアプローチです。	一貫性

ストラテジー	詳細	CAP
ALLOW_READ_WRITES	各パーティションのエントリーが分岐することを許可し、パーティションのマージ時に競合解決を試みます。	可用性

アプリケーションの要件によって、適切なストラテジーを決定する必要があります。たとえば、DENY_READ_WRITES は、高い整合性要件を持つアプリケーションにより適しています。つまり、システムからデータを読み取ったデータを正確にする必要があります。一方、Data Grid がベストエフォートキャッシュとして使用されている場合、パーティションは完全に許容できる可能性があり、一貫性よりも可用性を優先するため、ALLOW_READ_WRITES の方が適切な場合があります。

以下のセクションでは、各パーティション処理ストラテジーに対して Data Grid が [スプリットブレイン](#) と [連続した障害](#) を処理する方法を説明します。この後に、Data Grid が [マージポリシー](#) を介したパーティションマージ時に自動競合解決を可能にする方法を説明するセクションが続きます。最後に、[パーティション処理ストラテジーおよびマージポリシーの設定方法](#) を説明するセクションを説明します。

8.1.1. スプリットブレイン

スプリットブレインの状況では、各ネットワークパーティションは独自の JGroups ビューをインストールし、他のパーティションからノードを削除します。パーティションが互いに認識していないため、2つ以上のパーティションに分割されているかどうかを直接判断する方法はありません。代わりに、明示的な脱退メッセージを送信せずに1つ以上のノードが JGroups クラスターから消えたときに、クラスターが分割されたと想定します。

8.1.1.1. 分割ストラテジー

このセクションでは、スプリットブレインが発生したときに各パーティション処理ストラテジーがどのように動作するかを詳細に説明します。

8.1.1.1.1. ALLOW_READ_WRITES

各パーティションは、AVAILABLE モードで残りのすべてのパーティションで、独立したクラスターとして機能します。つまり、各パーティションはデータの一部のみを確認でき、各パーティションはキャッシュに競合する更新を書き込む可能性があることを意味します。これらの競合をマージすると、[ConflictManager](#) と設定された [EntryMergePolicy](#) を利用することで自動的に解決されます。

8.1.1.1.2. DENY_READ_WRITES

分割が検出されると、各パーティションは即座にリバランスを開始しませんが、まずは代わりに **DEGRADED** モードになるかどうかを確認します。

- 少なくとも1つのセグメントがすべての所有者を失った場合 (最後のリバランスが終了してから少なくとも `numOwners` ノードが残っていることを意味します)、パーティションは **DEGRADED** モードに入ります。
- 最新の安定したトポロジ** に、パーティションに単純なノード ($\text{floor}(\text{numNodes}/2) + 1$) が含まれていない場合は、パーティションも **DEGRADED** モードになります。
- それ以外の場合、パーティションは正常に機能し、リバランスを開始します。

安定したトポロジーは、リバランス操作が終了するたびに更新され、コーディネーターによって別のリバランスが不要であると判断されます。

これらのルールは、複数のパーティションが AVAILABLE モードになるようにし、他のパーティションが DEGRADED モードになるようにします。

パーティションが DEGRADED モードの場合、完全に所有されているキーへのアクセスのみを許可します。

- このパーティション内のノード上のコピーをすべて持つエントリーのリクエスト (読み取りおよび書き込み) は異常な状態です。
- 消えたノードによって部分的または完全に所有されているエントリーの要求は、**AvailabilityException** で拒否されます。

これにより、パーティションが同じキーに異なる値を書き込めないようにし (キャッシュの一貫性を保つ)、また、1つのパーティションが他のパーティションで更新されたキーを読み取れないようにすることができます (古いデータはありません)。

強調を行うには、**numOwners = 2** を使用した分散モードに設定されている初期クラスター **M = {A, B, C, D}** を検討してください。さらに、**owners(k1) = {A,B}**、**owners(k2) = {B,C}**、および **owners(k3) = {C,D}** となる 3つのキー **k1**、**k2**、および **k3**(キャッシュに存在するかどうかは不明) について考えてみます。次に、ネットワークは **N1 = {A, B}** と **N2 = {C, D}** の2つのパーティションに分割され、DEGRADED モードに入り、以下のように動作します。

- **N1** では、**k1** は読み取り/書き込みに使用でき、**k2**(部分的に所有) と **k3**(所有されていない) は使用できず、それらにアクセスすると、**AvailabilityException** が発生します。
- **N2** では **k1** および **k2** は読み取り/書き込みには使用できません。**k3** が利用できます。

パーティション処理プロセスの関連する要素として、スプリットブレインが発生すると、作成されるパーティションは、キーの所有権を算出するために元のセグメントマッピング (スプリットブレインの前に存在したもの) に依存します。**k1**、**k2**、または **k3** がすでに存在しているかどうかは重要で、それらの可用性は同じです。

さらに別の時点でネットワークが回復し、**N1** パーティションと **N2** パーティションがマージして最初のクラスター **M** に戻ると、**M** は劣化モードを終了し、再び完全に使用可能になります。このマージ操作中、**M** は再利用可能になったため、[ConflictManager](#) と設定された [EntryMergePolicy](#) を使用して、スプリットブレインの発生と検出の間に発生した可能性のある競合をチェックします。

別の例として、クラスターは **O1 = {A, B, C}** および **O2 = {D}** の2つのパーティションに分割して、パーティション **O1** が完全に使用可能になるようにすることができます (残りのメンバーのキャッシュエントリーのバランスを取り直します)。ただし、パーティション **O2** は分割を検出し、パフォーマンスが低下します。キーは完全に所有されていないため、**AvailabilityException** による読み取り操作または書き込み操作は拒否されます。

その後、**O1** および **O2** のパーティションが **M** にマージし直すと、[ConflictManager](#) は競合を解決しようとし、もう一度 **D** が完全に利用可能になります。

8.1.1.1.3. ALLOW_READS

パーティションは DENY_READ_WRITES と同じ方法で処理されます。ただし、パーティションが DEGRADED モードの場合に、部分的に所有されたキー上の読み取り操作は、**AvailabilityException** を出力しません。

8.1.1.2. 現在の制限

2つのパーティションは、分離して開始できます。マージしない限り、一貫性のないデータの読み込み、書き込みを行うことができます。今後は、カスタムの可用性ストラテジーを許可します (例: 特定のノードがクラスターの一部であるか、または外部のマシンにアクセスできるかどうかを確認など)。

8.1.2. 連続するノードが停止

前項で説明したように、Data Grid は、プロセス/マシンがクラッシュするため、またはネットワーク障害が原因でノードが JGroups ビューを残すかどうかを検出できません。ノードが JGroups クラスターを突然とすると、ネットワークの問題が原因でノードが JGroups クラスターを終了すると想定されます。

設定したコピー数 (**numOwners**) が1よりも大きい場合、クラスターは利用可能なままになり、クラッシュしたノードでデータの新しいレプリカを作成しようとします。ただし、リバランスプロセス中にその他のノードがクラッシュする可能性があります。短期間に **numOwners** ノードがクラッシュすると、一部のキャッシュエントリーがクラスターから完全に消える可能性があります。この場合、DENY_READ_WRITES または ALLOW_READS ストラテジーを有効にすると、(正しい)Data Grid はスプリットブレインのセクションに記載されているように、DEGRADED モードに入ります。

管理者は、**numOwners** を超えるノードを急速にシャットダウンして、それらのノードにのみ保存したデータが失われないようにすることもできます。管理者がノードを正常にシャットダウンすると、Data Grid はノードに戻ることができないことを認識します。ただし、クラスターは各ノードの残状況を追跡せず、それらのノードがクラッシュしたかのように、キャッシュが DEGRADED モードに入ります。

この段階では、クラスターが停止し、外部ソースからのデータを使用して再起動時にその状態を回復する方法はありません。クラスターは、**numOwners** の連続するノードの失敗を回避するために、適切な **numOwners** で設定されることが予想されます。したがって、この状況は非常に稀なはずで、アプリケーションがキャッシュ内の一部のデータを失うことが可能な場合、管理者は JMX 経由で可用性モードを AVAILABLE に戻すことができます。

8.1.3. 競合マネージャー

競合マネージャーは、ユーザーが特定のキーに保存されているすべてのレプリカ値を取得できるツールです。保存したレプリカの値が競合しているキャッシュエントリーのストリームをユーザーが処理できるようにする他にもあります。さらに、[EntryMergePolicy](#) インターフェイスの実装を利用することで、そのような競合を自動的に解決できます。

8.1.3.1. 競合の検出

競合は、指定のキーの保存された各値を取得することによって検出されます。競合マネージャーは、現在の整合性ハッシュによって定義された各キーの書き込み所有者から保存された値を取得します。保存した値の `.equals` メソッドは、すべての値が等しいかどうかを判断するために使用されます。すべての値が等しい場合は、キーに競合が発生しません。それ以外の場合は競合が発生しています。指定のノードにエントリーが存在しない場合は、`null` 値が返されることに注意してください。そのため、指定のキーに `null` 値と非 `null` 値が存在する場合に競合が発生している点に注意してください。

8.1.3.2. マージポリシー

特定の `CacheEntry` のレプリカ間で競合が発生した場合は、競合解決アルゴリズムを定義する必要があります。そのため、[EntryMergePolicy](#) インターフェイスを提供します。このインターフェイスは、単一のメソッド `merge` で設定され、その返された `CacheEntry` は、指定されたキーの `"resolved"` エントリーとして使用されます。`null` 以外の `CacheEntry` が返されると、このエントリー値はキャッシュのすべてのレプリカに `"put"` されます。ただし、マージの実装が `null` 値を返すと、競合するキーに関連付けられたすべてのレプリカがキャッシュから削除されます。

`merge` メソッドは、`"preferredEntry"` および `"otherEntries"` という2つのパラメーターを取ります。

パーティションのマージのコンテキストでは、希望の Entry がパーティションに保存されている CacheEntry のプライマリーレプリカで、パーティションには最も大きな topologyId を持つものと等しくなります。パーティションが重複している場合、つまりノード A は両方のパーティション {A}、{A,B,C} のトポロジーにあります。このパーティションでは、他のパーティションのトポロジーの背後に topologyId が高いため、{A} を選択します。パーティションのマージが発生しないと、"preferredEntry" は CacheEntry のプライマリーレプリカです。2 番目のパラメーター "otherEntries" は、競合が検出されたキーに関連付けられた他のすべてのエントリーのリストです。



注記

EntryMergePolicy::merge は、競合が検出されたときにのみ呼び出され、すべての CacheEntrys が同じ場合は呼び出されません。

現在、Data Grid は EntryMergePolicy の以下の実装を提供します。

ポリシー	詳細
MergePolicy.NONE(デフォルト)	競合を解決するための試行はありません。マイノリティパーティションでホストされているエントリーは削除され、このパーティションのノードはリバランスが開始されるまでデータを保持しません。この動作は、競合解決をサポートしない Infinispan の以前のバージョンと同等であることに注意してください。この場合、マイナーパーティションでホストされるエントリーに対する変更はすべて失われますが、リバランスが終了するとすべてのエントリーの一貫性が保たれます。
MergePolicy.PREFERRED_ALWAYS	常に "preferredEntry" を使用します。MergePolicy.NONE は、PREFERRED_ALWAYS とほぼ同等で、競合解決の実行によるパフォーマンスへの影響はありません。そのため、以下のシナリオが懸念されない限り、MergePolicy.NONE を選択する必要があります。DENY_READ_WRITES または DENY_READ ストラテジーを使用する場合は、パーティションが DEGRADED モードに入る場合にのみ、書き込み操作が部分的に完了するため、一貫性のない値が含まれるレプリカが作成されます。MergePolicy.PREFERRED_ALWAYS は対象の不整合を検出し、これを解決します。ただし、MergePolicy.NONE の場合、CacheEntry レプリカはクラスタのリバランス後に一貫性がありません。
MergePolicy.PREFERRED_NON_NULL	null 以外の場合は "preferredEntry" を使用します。それ以外の場合は、"otherEntries" の最初のエントリーを利用します。
MergePolicy.REMOVE_ALL	競合が検出されると、必ずキャッシュからキーを削除します。
完全修飾クラス名	マージのカスタム実装は、 カスタムマージポリシー を使用します。

8.1.4. 使用法

パーティションが ConflictManager をマージすると、設定された EntryMergePolicy の競合を自動的に解決しようとはしますが、アプリケーションが必要とする競合の有無を手動で検索または解決することもできます。

以下のコードは、EmbedCacheManager の ConflictManager を取得する方法、指定されたキーの全バージョンを取得する方法、および特定のキャッシュ全体で競合をチェックする方法を示しています。

```
EmbeddedCacheManager manager = new DefaultCacheManager("example-config.xml");
Cache<Integer, String> cache = manager.getCache("testCache");
ConflictManager<Integer, String> crm = ConflictManagerFactory.get(cache.getAdvancedCache());

// Get All Versions of Key
Map<Address, InternalCacheValue<String>> versions = crm.getAllVersions(1);

// Process conflicts stream and perform some operation on the cache
Stream<Map<Address, CacheEntry<Integer, String>>> conflicts = crm.getConflicts();
conflicts.forEach(map -> {
    CacheEntry<Integer, String> entry = map.values().iterator().next();
    Object conflictKey = entry.getKey();
    cache.remove(conflictKey);
});

// Detect and then resolve conflicts using the configured EntryMergePolicy
crm.resolveConflicts();

// Detect and then resolve conflicts using the passed EntryMergePolicy instance
crm.resolveConflicts((preferredEntry, otherEntries) -> preferredEntry);
```



注記

ConflictManager::getConflicts ストリームはエントリーごとに処理されますが、基礎となるスプリットは、セグメントごとにキャッシュエントリーを遅延読み込みしています。

8.1.5. パーティション処理の設定

キャッシュが分散またはレプリケートされない限り、パーティション処理の設定は無視されます。デフォルトのパーティション処理ストラテジーは ALLOW_READ_WRITES で、デフォルトの EntryMergePolicy は MergePolicies::PREFERRED_ALWAYS です。

```
<distributed-cache name="the-default-cache">
  <partition-handling when-split="ALLOW_READ_WRITES" merge-
policy="PREFERRED_NON_NULL"/>
</distributed-cache>
```

プログラムを使用しても同じことができます。

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
    .mergePolicy(MergePolicy.PREFERRED_ALWAYS);
```


8.1.5.1. カスタムマージポリシーの実装

EntryMergePolicy のカスタム実装を提供することもできます。

```
<distributed-cache name="mycache">
  <partition-handling when-split="ALLOW_READ_WRITES"
    merge-policy="org.example.CustomMergePolicy"/>
</distributed-cache>
```

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
    .mergePolicy(new CustomMergePolicy());
```

```
public class CustomMergePolicy implements EntryMergePolicy<String, String> {

    @Override
    public CacheEntry<String, String> merge(CacheEntry<String, String> preferredEntry,
        List<CacheEntry<String, String>> otherEntries) {
        // decide which entry should be used

        return the_solved_CacheEntry;
    }
}
```

8.1.5.2. Infinispan サーバーインスタンスへのカスタムマージポリシーのデプロイ

サーバーでカスタム EntryMergePolicy 実装を使用するには、実装クラスをサーバーにデプロイする必要があります。これは、java service-provider の規則を使用し、EntryMergePolicy 実装の完全修飾クラス名を含む META-INF/services/org.infinispan.conflict.EntryMergePolicy ファイルを持つ jar のクラスファイルをパッケージ化することで実現できます。

```
# list all necessary implementations of EntryMergePolicy with the full qualified name
org.example.CustomMergePolicy
```

カスタムマージポリシーをサーバー上で利用できるようにするには、ポリシーセマンティックが保存された Key/Value オブジェクトにアクセスする必要がある場合は、オブジェクトストレージを有効にする必要があります。これは、サーバーのキャッシュエントリはマーシャル形式に格納され、ポリシーに返された Key/Value オブジェクトは WrappedByteArray のインスタンスとなる可能性があるためです。ただし、カスタムポリシーがキャッシュエントリに関連付けられたメタデータだけに依存する場合、オブジェクトストレージは必要ではなく、リクエストごとのマーシャリングデータの追加のパフォーマンスコストが原因で (他の理由で必要としなければ) 回避する必要があります。最後に、提供されるマージポリシーのいずれかが使用されている場合は、オブジェクトストレージは必要ありません。

8.1.6. 監視および管理

キャッシュの可用性モードは、JMX で [Cache MBean](#) の属性として公開されます。属性は書き込み可能で、管理者はキャッシュを DEGRADED モードから AVAILABLE(一貫性のコスト) に強制的に移行することができます。

可用性モードは、[AdvancedCache](#) インターフェイスからもアクセスできます。

```
AdvancedCache ac = cache.getAdvancedCache();
```

```
// Read the availability  
boolean available = ac.getAvailability() == AvailabilityMode.AVAILABLE;  
  
// Change the availability  
if (!available) {  
    ac.setAvailability(AvailabilityMode.AVAILABLE);  
}
```