



Red Hat Data Grid 8.3

Data Grid キャッシュの設定

Data Grid キャッシュを設定してデプロイメントをカスタマイズする

Red Hat Data Grid 8.3 Data Grid キャッシュの設定

Data Grid キャッシュを設定してデプロイメントをカスタマイズする

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

ビジネス要件に合った機能を使用するように、Data Grid のデプロイメントを設定します。

目次

RED HAT DATA GRID	4
DATA GRID のドキュメント	5
DATA GRID のダウンロード	6
多様性を受け入れるオープンソースの強化	7
RED HAT ドキュメントへのフィードバック (英語のみ)	8
第1章 DATA GRID キャッシュ	9
1.1. キャッシュ API	9
1.2. キャッシュマネージャー	9
1.3. キャッシュモード	9
1.4. ローカルキャッシュ	11
第2章 クラスター化されたキャッシュ	13
2.1. レプリケートされたキャッシュ	13
2.2. 分散キャッシュ	14
2.3. INVALIDATION(インバリデーション) キャッシュ	25
2.4. 散在 (SCATTERED) キャッシュ	26
2.5. 非同期のレプリケーション	27
2.6. 初期クラスターサイズの設定	28
第3章 DATA GRID キャッシュの設定	30
3.1. 宣言型キャッシュの設定	30
3.2. キャッシュテンプレートの追加	37
3.3. リモートキャッシュの作成	42
3.4. 組み込みキャッシュの作成	46
第4章 DATA GRID 統計および JMX 監視の有効化および設定	48
4.1. DATA GRID メトリクスの設定	48
4.2. JMX MBEAN の登録	49
第5章 JVM メモリ使用量の設定	53
5.1. デフォルトのメモリ設定	53
5.2. エビクションと有効期限	53
5.3. DATA GRID キャッシュを使用したエビクション	54
5.4. ライフスパンと最大アイドル期間の有効期限	59
5.5. JVM ヒープおよびオフヒープメモリ	63
第6章 CONFIGURING PERSISTENT STORAGE	66
6.1. パッシベーション	66
6.2. ライトスルーキャッシュストア	67
6.3. WRITE-BEHIND キャッシュストア	68
6.4. セグメント化されたキャッシュストア	70
6.5. 共有キャッシュストア	70
6.6. 永続キャッシュストアを使用するトランザクション	71
6.7. グローバルの永続的な場所	72
6.8. ファイルベースのキャッシュストア	74
6.9. JDBC 接続ファクトリー	79
6.10. SQL キャッシュストア	88
6.11. JDBC 文字列ベースのキャッシュストア	100
6.12. ROCKSDB のキャッシュストア	104

6.13. リモートキャッシュストア	107
6.14. JPA キャッシュストア	109
6.15. クラスターキャッシュローダー	111
6.16. カスタムキャッシュストア実装の作成	113
6.17. キャッシュストア間のデータの移行	115
第7章 ネットワークパーティションを処理するための DATA GRID 設定	123
7.1. クラスターおよびネットワークパーティションの分割	123
7.2. キャッシュの可用性およびデグレードモード	124
7.3. パーティション処理の設定	127
7.4. パーティション処理ストラテジー	129
7.5. マージポリシー	129
7.6. カスタムマージポリシーの設定	130
7.7. 組み込みキャッシュでのパーティションの手動マージ	132
第8章 ユーザーロールとパーミッションの設定	133
8.1. セキュリティー認証	133
8.2. アクセス制御リスト (ACL) キャッシュ	137
8.3. ロールおよびパーミッションのカスタマイズ	138
8.4. セキュリティー承認によるキャッシュの設定	140
8.5. セキュリティー承認の無効化	142

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 8.3 ドキュメント](#)
- [Data Grid 8.3 コンポーネントの詳細](#)
- [Data Grid 8.3 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](#) を参照してください。

RED HAT ドキュメントへのフィードバック (英語のみ)

弊社の技術的な内容についてのフィードバックに感謝します。ご意見をお聞かせください。コメントの追加、Insights の提供、誤字の修正、および質問を行う必要がある場合は、ドキュメントで直接行うこともできます。



注記

Red Hat アカウントがあり、カスタマーポータルにログインしている必要があります。

カスタマーポータルからドキュメントのフィードバックを送信するには、以下の手順を実施します。

1. **Multi-page HTML** 形式を選択します。
2. ドキュメントの右上にある **Feedback** ボタンをクリックします。
3. フィードバックを提供するテキストのセクションを強調表示します。
4. ハイライトされたテキストの横にある **Add Feedback** ダイアログをクリックします。
5. ページの右側のテキストボックスにフィードバックを入力し、**送信** をクリックします。

フィードバックを送信すると、自動的に問題の追跡が作成されます。**Submit** をクリックすると表示されるリンクを開き、問題の監視を開始するか、さらにコメントを追加します。

貴重なフィードバックにご協力いただきありがとうございます。

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のパフォーマンスを向上させる。
- 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を最適化します。
- クラスタ全体で一貫したデータに対する回復性および持続性を提供します。

1.1. キャッシュ API

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

CacheManager API は、Data Grid キャッシュと対話するための開始点です。キャッシュマネージャーはキャッシュのライフサイクルを制御し、キャッシュインスタンスの作成、変更、および削除を行います。

Data Grid は、2つの **CacheManager** 実装を提供します。

EmbeddedCacheManager

クライアントアプリケーションと同じ Java 仮想マシン (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。

RemoteCacheManager

独自の JVM で Data Grid Server を実行する場合のキャッシュのエントリーポイント。**RemoteCacheManager** をインスタンス化すると、Hot Rod エンドポイントを使用して Data Grid Server 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. キャッシュモード

ヒント

Data Grid キャッシュマネージャーは、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、分散キャッシュ、およびインバリデーションモードでのキャッシュに、同じキャッシュマネージャーを使用することができます。

Local

Data Grid は単一ノードとして実行され、キャッシュエントリ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。

Replicated (レプリケート)

Data Grid は、クラスター内のすべてのノードのすべてのキャッシュエントリーを複製し、ローカル読み取り操作のみを実行します。

Distributed (分散)

Data Grid は、クラスター内のノードのサブセットでキャッシュエントリーを複製し、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。

Invalidation (無効化)

Data Grid は、操作のキャッシュ内のエントリーが変更されるたびに、すべてのノードから古いデータをエビクトします。Data Grid は、ローカルの読み取り操作のみを実行します。

Scattered(散在)

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 scattered 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。

1.3.1. 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
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

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
Distributed (分散)	Yes	メディア ア(所有者)	メディア 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。

1.4. ローカルキャッシュ

Data Grid は、**ConcurrentHashMap** に似たローカルキャッシュモードを提供します。

キャッシュは、永続ストレージやエビクションや有効期限などの管理機能など、単純なマップよりも多くの機能を提供します。

Data Grid **Cache** API は Java の **ConcurrentMap** API を拡張し、マップから Data Grid キャッシュへの移行を容易にします。

ローカルキャッシュの設定

XML

```
<local-cache name="mycache"
  statistics="true">
  <encoding media-type="application/x-protostream"/>
</local-cache>
```

JSON

```
{
  "local-cache": {
    "name": "mycache",
    "statistics": "true",
    "encoding": {
      "media-type": "application/x-protostream"
    }
  }
}
```

YAML

```
localCache:
  name: "mycache"
  statistics: "true"
```

```
encoding:  
  mediaType: "application/x-protostream"
```

1.4.1. 単純なキャッシュ

単純なキャッシュは、以下の機能のサポートを無効にするローカルキャッシュのタイプです。

- トランザクションと呼び出しのバッチ処理
- 永続ストレージ
- カスタムインターセプター
- インデックス化
- トランダング

ただし、有効期限、エビクション、統計、およびセキュリティー機能などの単純なキャッシュで他の Data Grid 機能を使用できます。単純なキャッシュと互換性がない機能を設定すると、Data Grid は例外を出力します。

単純なキャッシュ設定

XML

```
<local-cache simple-cache="true" />
```

JSON

```
{  
  "local-cache": {  
    "simple-cache": "true"  
  }  
}
```

YAML

```
localCache:  
  simpleCache: "true"
```


第2章 クラスター化されたキャッシュ

ノード間でデータをレプリケートする Data Grid クラスターで組み込みキャッシュおよびリモートキャッシュを作成できます。

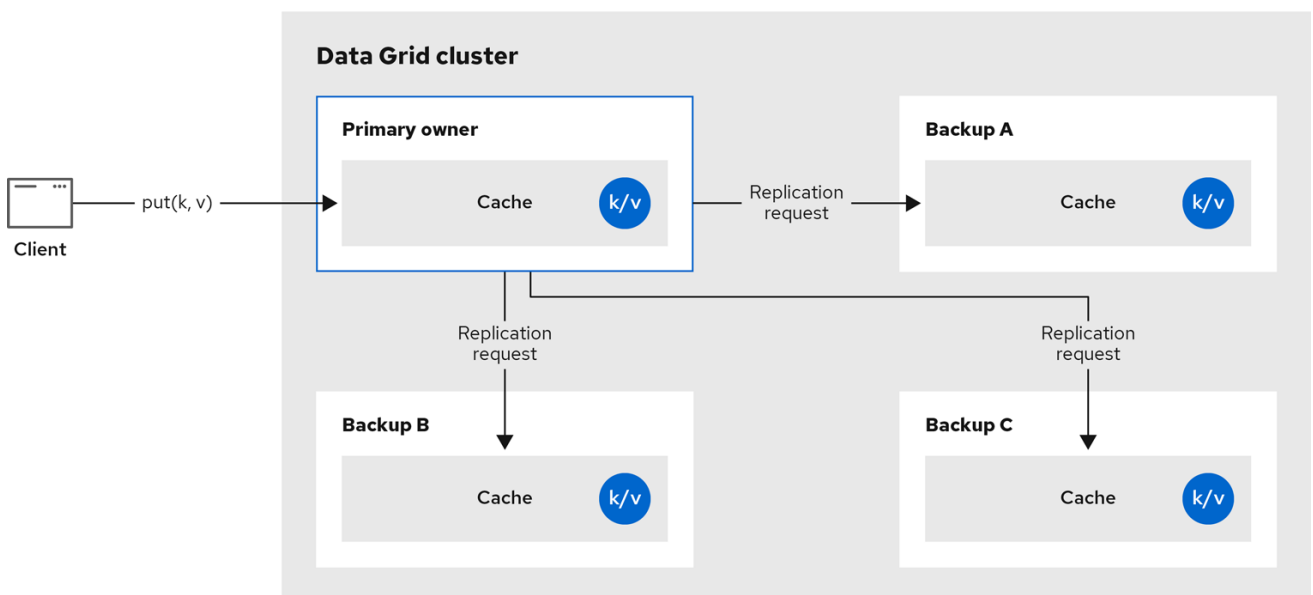
2.1. レプリケートされたキャッシュ

Data Grid は、キャッシュ内のすべてのエントリをクラスター内のすべてのノードに複製します。各ノードはローカルに読み取り操作を実行できます。

レプリケートされたキャッシュは、クラスター全体で状態をすばやく簡単に共有する方法を提供しますが、10 未満のノードのクラスターに適しています。レプリケーション要求の数はクラスター内のノード数を線形にスケールするため、大規模なクラスターでレプリケートされたキャッシュを使用するとパフォーマンスが低下します。ただし、レプリケーション要求に UDP マルチキャストを使用すると、パフォーマンスを向上させることができます。

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。

図2.1レプリケートされたキャッシュ



184_Data_Grid_0921

同期または非同期のレプリケーション

- 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

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。

2.2. 分散キャッシュ

Data Grid は、**numOwners** として設定された、キャッシュ内のエントリーの固定数のコピーを維持しようとします。これにより、分散キャッシュを線形にスケールでき、ノードをクラスターに追加する際により多くのデータを保存できます。

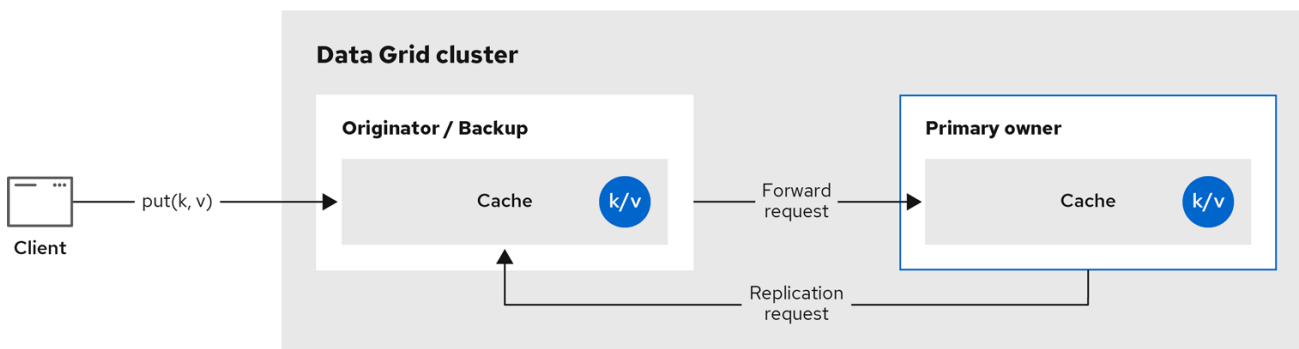
ノードがクラスターに参加およびクラスターから離脱すると、キーのコピー数が **numOwners** より多い場合と少ない場合があります。特に、**numOwners** ノードがすぐに連続して離れると、一部のエントリーが失われるため、分散キャッシュは、**numOwners - 1** ノードの障害を許容すると言われます。

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喪失のリスクも低くなります。

Data Grid は、キーの所有者を1つの **プライマリー所有者** に分割します。これにより、キーへの書き込みが行われ、ゼロ以上の **バックアップ所有者** が調整されます。

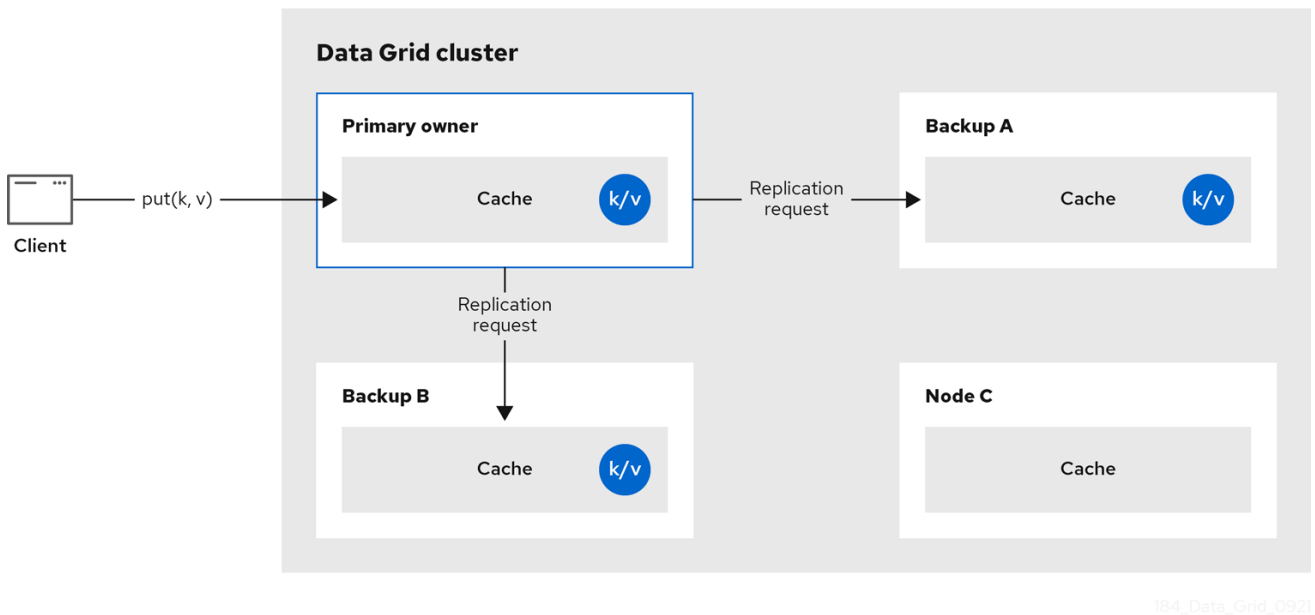
以下の図は、クライアントがバックアップ所有者に送信する書き込み操作を示しています。この場合、バックアップノードはプライマリー所有者に書き込みを転送し、書き込みをバックアップに複製します。

図2.2 クラスターのレプリケーション



184_Data_Grid_0921

図2.3 分散キャッシュ



読み取り操作

読み取り操作は、プライマリー所有者から値を要求します。プライマリー所有者が妥当な時間内に応答しない場合は、Data Grid はバックアップの所有者から値も要求します。

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。

書き込み操作

書き込み操作により、最大 **2 * numOwners** メッセージが生成されます。発信者からプライマリー所有者への1つのメッセージ。プライマリー所有者からバックアップノードへの **numOwners - 1** メッセージと、対応する確認応答メッセージ。



注記

キャッシュトポロジーの変更により、読み取り操作と書き込み操作の両方に対して再試行が行われる可能性があります。

同期または非同期のレプリケーション

更新を失う可能性があるため、非同期のレプリケーションは推奨されません。更新の喪失に加えて、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、その後に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

トランザクション分散キャッシュは、影響を受けるノードにのみロック/prepare/commit/unlock メッセージを送信します。つまり、トランザクションの影響を受ける1つの鍵を所有するすべてのノードを意味します。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。

2.2.1. 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では、シリアル化/スナップショットの分離はサポートしません。

たとえば、スレッドは1つの配置リクエストを行います。

```
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
```

理由は、プライマリー所有者の返信速度が速いかによって、読み取りはどの所有者からでも値を返すことができるからです。書き込みは、すべての所有者全体で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。

2.2.2. キーの所有権

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の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。

一貫性のあるハッシュファクトリーの実装	説明
SyncConsistentHashFactory	<p>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。</p> <p>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。</p>

一貫性のあるハッシュファクトリーの実装	説明
TopologyAwareSyncConsistentHashFactory	SyncConsistentHashFactory と同等ですが、データのバックアップコピーがプライマリ所有者とは異なるノードに保存されるように、トポロジ間でデータを分散するためにサーバーヒントで使われます。これは、サーバーヒントによるデフォルトの一貫性のあるハッシュ実装です。
DefaultConsistentHashFactory	SyncConsistentHashFactory よりも均等に分散を行います、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。
TopologyAwareConsistentHashFactory	DefaultConsistentHashFactory と同等ですが、データのバックアップコピーがプライマリ所有者とは異なるノードに保存されるように、トポロジ間でデータを分散するためにサーバーヒントで使われます。
ReplicatedConsistentHashFactory	レプリケートされたキャッシュの実装に内部で使われます。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。

ハッシュ設定

組み込みキャッシュのみを使用して、カスタムを含む **ConsistentHashFactory** 実装を設定できます。

XML

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

ConfigurationBuilder

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

関連情報

- [KeyPartitioner](#)

2.2.3. 容量要素

容量係数は、クラスター内の各ノードで利用可能なリソースに基づいてセグメント数を割り当てます。

ノードの容量係数は、ノードがプライマリー所有者とバックアップ所有者の両方であるセグメントに適用されます。つまり、容量係数は、ノードがクラスター内の他のノードと比較した合計容量です。

デフォルト値は **1** です。つまり、クラスターのすべてのノードに同じ容量があり、Data Grid はクラスターのすべてのノードに同じ数のセグメントを割り当てます。

ただし、ノードにさまざまなメモリー量がある場合は、Data Grid ハッシュアルゴリズムが各ノードの容量で重み付けする多数のセグメントを割り当てるように、キャパシティ係数を設定することができます。

容量係数の設定の値は正の値で、1.5 などの分数になります。容量係数 **0** を設定することもできますが、クラスターを一時的に参加するノードのみに推奨されます。代わりに、容量設定を使用することが推奨されます。

2.2.3.1. ゼロ容量ノード

すべてのキャッシュ、ユーザー定義のキャッシュ、および内部キャッシュに対して容量係数が **0** であるノードを設定できます。ゼロ容量ノードを定義する場合、ノードにはデータを保持しません。

ゼロ容量ノードの設定

XML

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

JSON

```
{
  "infinispan": {
    "cache-container": {
      "zero-capacity-node": "true"
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    zeroCapacityNode: "true"
```

ConfigurationBuilder

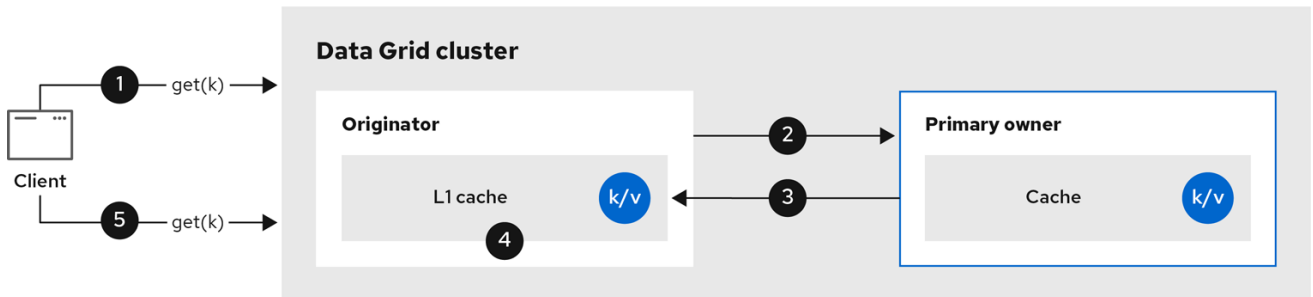
```
new GlobalConfigurationBuilder().zeroCapacityNode(true);
```

2.2.4. レベル 1(L1) キャッシュ

Data Grid ノードは、cluster の別のノードからエントリを取得すると、ローカルレプリカを作成します。L1 キャッシュは、プライマリ所有者ノードでエントリを繰り返し検索せずに、パフォーマンスを追加します。

以下の図は、L1 キャッシュの動作を示しています。

図2.4 L1 cache



184_Data_Grid_0921

L1 cache の図では、以下のようになります。

1. クライアントは **cache.get()** を呼び出して、クラスター内の別のノードがプライマリ所有者であるエントリを読み取ります。
2. 元のノードは読み取り操作をプライマリ所有者に転送します。
3. プライマリ所有者はキー/値エントリを返します。
4. 元のノードはローカルコピーを作成します。
5. 後続の **cache.get()** 呼び出しは、プライマリ所有者に転送するのではなく、ローカルエントリを返します。

L1キャッシュパフォーマンス

L1 を有効にすると読み取り操作のパフォーマンスが改善されますが、エントリが変更されたときにプライマリ所有者ノードがメッセージをブロードキャストする必要があります。これにより、Data Grid はクラスター全体で古くなったレプリカをすべて削除します。ただし、これにより書き込み操作のパフォーマンスが低下し、メモリー使用量が増大し、キャッシュの全体的な容量が削減されます。



注記

Data Grid は、他のキャッシュエントリと同様に、ローカルレプリカまたは L1 エントリをエビクトして期限切れにします。

L1キャッシュ設定

XML

```

<distributed-cache l1-lifespan="5000"
    l1-cleanup-interval="60000">
</distributed-cache>

```

JSON

```
{
  "distributed-cache": {
    "l1-lifespan": "5000",
    "l1-cleanup-interval": "60000"
  }
}
```

YAML

```
distributedCache:
  l1Lifespan: "5000"
  l1-cleanup-interval: "60000"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC)
    .l1()
    .lifespan(5000, TimeUnit.MILLISECONDS)
    .cleanupTaskFrequency(60000, TimeUnit.MILLISECONDS);
```

2.2.5. サーバーヒント

サーバーヒントは、できるだけ多くのサーバー、ラック、およびデータセンター間でエントリーをレプリケートすることで、分散キャッシュのデータ可用性を高めます。



注記

サーバーのヒントは、分散キャッシュにのみ適用されます。

Data Grid がデータのコピーを配布する場合は、サイト、ラック、マシン、およびノードの優先順位に従います。すべての設定属性はオプションです。たとえば、ラック ID のみを指定する場合、Data Grid はコピーを別のラックおよびノードに分散します。

サーバーのヒントは、キャッシュのセグメント数が少なすぎる場合に必要以上のセグメントを移動し、クラスターのリバランス操作に影響を与える可能性があります。

ヒント

複数のデータセンターにおけるクラスターの代替は、クロスサイトレプリケーションです。

サーバーヒントの設定

XML

```
<cache-container>
  <transport cluster="MyCluster"
    machine="LinuxServer01"
```



```

        rack="Rack01"
        site="US-WestCoast"/>
</cache-container>

```

JSON

```

{
  "infinispan" : {
    "cache-container" : {
      "transport" : {
        "cluster" : "MyCluster",
        "machine" : "LinuxServer01",
        "rack" : "Rack01",
        "site" : "US-WestCoast"
      }
    }
  }
}

```

YAML

```

cacheContainer:
  transport:
    cluster: "MyCluster"
    machine: "LinuxServer01"
    rack: "Rack01"
    site: "US-WestCoast"

```

GlobalConfigurationBuilder

```

GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder()
    .transport()
    .clusterName("MyCluster")
    .machineId("LinuxServer01")
    .rackId("Rack01")
    .siteId("US-WestCoast");

```

関連情報

- [org.infinispan.configuration.global.TransportConfigurationBuilder](#)

2.2.6. 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、Data Grid は一連の (疑似) ランダムキーを生成し、それらのプライマリー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。

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。

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。

トポロジーの変更

キャッシュトポロジーが変更すると、**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、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。

2.2.7. グループ化 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。

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。

Grouping API が使用されている場合、すべてのノードが他のノードと通信せずにすべての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。

Grouping API を使用するには、グループを有効にする必要があります。

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

```
<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。侵入グループは、**@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

```

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().addGrouper(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.example.KXGrouper" />
  </groups>
</distributed-cache>

```

高度な API

AdvancedCache には、グループ固有のメソッドが2つあります。

- **getGroup(groupName)** は、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)** は、グループに属するキャッシュにあるすべてのキーを削除します。

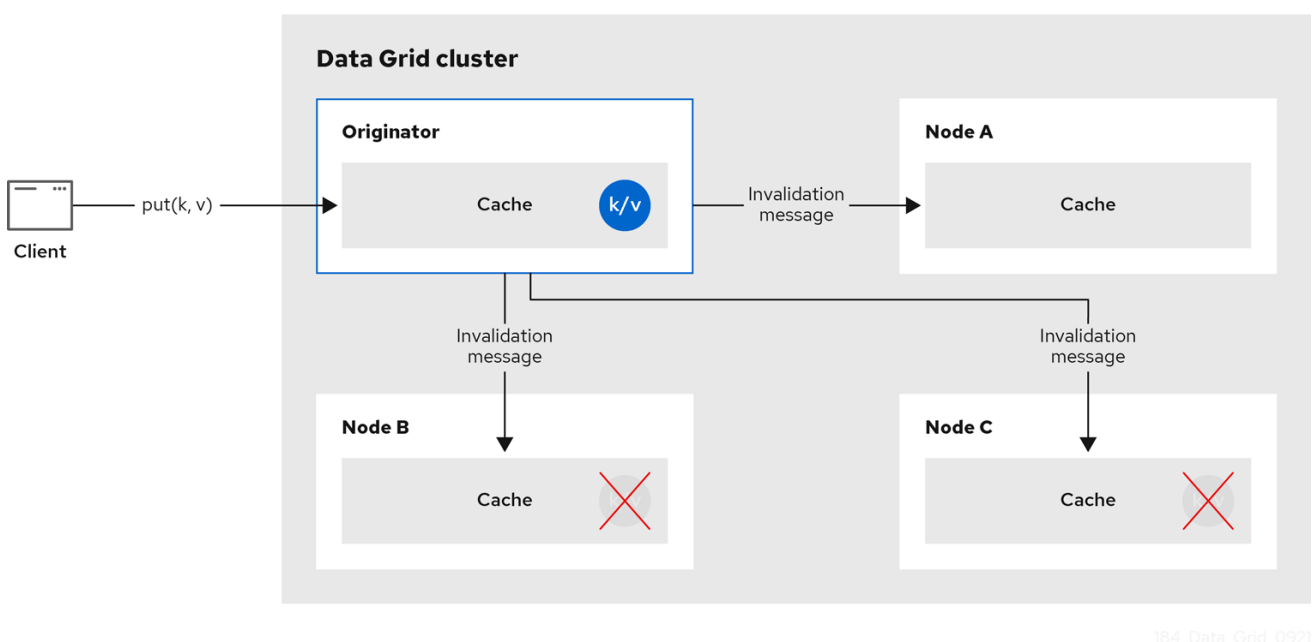
どちらのメソッドもデータコンテナ全体とストア (存在する場合) を繰り返し処理するため、キャッシュに多くの小規模なグループが含まれる場合に処理が遅くなる可能性があります。

2.3. INVALIDATION(インバリデーション) キャッシュ

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2.5 Invalidation cache



IS4_Data_Grid_0921

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。



重要

ローカル、共有されていないキャッシュストアでは invalidation 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。

同期または非同期のレプリケーション

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、元のブロードキャストは無効化メッセージを無効にしますが、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

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ー所有者でキーロックを取得します。

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。

2.4. 散在 (SCATTERED) キャッシュ

散在 (scattered) キャッシュは、クラスターの線形のスケールリングが可能であるため、分散キャッシュと非常に似ています。散在 (scattered) キャッシュにより、データ (**numOwners=2**) の2つのコピーを維持することで、単一ノードの障害が可能になります。分散キャッシュとは異なり、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、すべての書き込み(分散キャッシュ)に単一の Remote Procedure Call(RPC) を活用しますが、読み取りは常にプライマリー所有者を対象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、多数のノードを持つクラスターでの散在キャッシュのパフォーマンスが低下します。

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。

データの書き込み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 を指定していても、同じマシン/ラック/サイトの障害に対して、クラスターが回復性を持つことができません。



注記

トランザクションまたは非同期レプリケーションでは、散財キャッシュを使用することはできません。

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" />
```

```
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。

2.5. 非同期のレプリケーション

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。

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。

2.5.1. 非同期レプリケーションのある戻り値

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が変更された場合。トポロジの変更により、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。

2.6. 初期クラスターサイズの設定

Data Grid は、クラスタートポロジーの変更を動的に処理します。これは、Data Grid がキャッシュを初期化する前に、他のノードがクラスターに参加する必要があることを意味します。

キャッシュの開始前にアプリケーションがクラスター内の特定のノードを必要とする場合は、初期クラスターサイズをトランスポートの一部として設定できます。

手順

1. Data Grid 設定を開いて編集します。
2. キャッシュの開始前に必要なノードの最小数を **initial-cluster-size** 属性または **initialClusterSize()** メソッドで設定します。
3. キャッシュマネージャーが **initial-cluster-timeout** 属性または **initialClusterTimeout()** メソッドで開始しないまでの時間をミリ秒単位で設定します。
4. Data Grid 設定を保存して閉じます。

初期クラスターサイズの設定

XML

```

<infinispan>
  <cache-container>
    <transport initial-cluster-size="4"
      initial-cluster-timeout="30000" />
  </cache-container>
</infinispan>

```

JSON


```
{
  "infinispan" : {
    "cache-container" : {
      "transport" : {
        "initial-cluster-size" : "4",
        "initial-cluster-timeout" : "30000"
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    transport:
      initialClusterSize: "4"
      initialClusterTimeout: "30000"
```

ConfigurationBuilder

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

第3章 DATA GRID キャッシュの設定

キャッシュ設定は、Data Grid がデータを保存する方法を制御します。

キャッシュ設定の一部として、使用するキャッシュモードを宣言します。たとえば、Data Grid クラスターがレプリケートされたキャッシュまたは分散キャッシュを使用するように設定できます。

設定は、キャッシュの特性も定義し、データの処理時に使用する Data Grid 機能を有効にします。たとえば、エントリーがモビリティまたは immortal である場合は、Data Grid がキャッシュ内のエントリーをエンコードする方法、レプリケーション要求がノード間で同期または非同期に行われるかを設定できます。

3.1. 宣言型キャッシュの設定

Data Grid スキーマに従って、XML または JSON 形式でキャッシュを宣言型で設定できます。

宣言型キャッシュ設定には、プログラムによる設定と比較して、以下のような利点があります。

移植性

埋め込みキャッシュおよびリモートキャッシュを作成するために使用できるスタンドアロンファイルに各設定を定義します。

宣言型設定を使用して、OpenShift で実行しているクラスターの Data Grid Operator でキャッシュを作成することもできます。

簡素化

マークアップ言語をプログラミング言語とは別に維持します。

たとえば、リモートキャッシュを作成するには、通常、複雑な XML を Java コードに直接追加しないことが推奨されます。



注記

Data Grid Server 設定は **infinispan.xml** を拡張し、クラスタートランスポートメカニズム、セキュリティレール、およびエンドポイント設定が含まれます。Data Grid Server 設定の一部としてキャッシュを宣言する場合、Ansible または Chef などの管理ツールを使用して、クラスター全体で同期する必要があります。

Data Grid クラスター全体でリモートキャッシュを動的に同期するには、実行時に作成します。

3.1.1. キャッシュ設定

XML、JSON、および YAML 形式で宣言型キャッシュ設定を作成できます。

すべての宣言型キャッシュは Data Grid スキーマに準拠する必要があります。JSON 形式の設定は XML 設定の構造に従う必要があります。要素がオブジェクトに対応し、属性はフィールドに対応します。



重要

Data Grid では、キャッシュ名またはキャッシュテンプレート名の文字数を最大 **255** 文字に制限しています。この文字制限を超えると、Data Grid サーバーは例外メッセージを発行せずに突然停止する場合があります。簡潔なキャッシュ名とキャッシュテンプレート名を記述します。



重要

ファイルシステムによってファイル名の長さに制限が設定される場合があるため、キャッシュの名前がこの制限を超えないようにしてください。キャッシュ名がファイルシステムの命名制限を超えると、そのキャッシュに対する一般的な操作または初期化操作が失敗する可能性があります。簡潔なキャッシュ名とキャッシュテンプレート名を記述します。

分散キャッシュ

XML

```
<distributed-cache owners="2"
    segments="256"
    capacity-factor="1.0"
    l1-lifespan="5000"
    mode="SYNC"
    statistics="true">
  <encoding media-type="application/x-protostream"/>
  <locking isolation="REPEATABLE_READ"/>
  <transaction mode="FULL_XA"
    locking="OPTIMISTIC"/>
  <expiration lifespan="5000"
    max-idle="1000" />
  <memory max-count="1000000"
    when-full="REMOVE"/>
  <indexing enabled="true"
    storage="local-heap">
    <index-reader refresh-interval="1000"/>
  </indexing>
  <partition-handling when-split="ALLOW_READ_WRITES"
    merge-policy="PREFERRED_NON_NULL"/>
  <persistence passivation="false">
    <!-- Persistent storage configuration. -->
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "mode": "SYNC",
    "owners": "2",
    "segments": "256",
    "capacity-factor": "1.0",
    "l1-lifespan": "5000",
    "statistics": "true",
    "encoding": {
      "media-type": "application/x-protostream"
    },
    "locking": {
      "isolation": "REPEATABLE_READ"
    },
    "transaction": {
```

```

    "mode": "FULL_XA",
    "locking": "OPTIMISTIC"
  },
  "expiration" : {
    "lifespan" : "5000",
    "max-idle" : "1000"
  },
  "memory": {
    "max-count": "1000000",
    "when-full": "REMOVE"
  },
  "indexing" : {
    "enabled" : true,
    "storage" : "local-heap",
    "index-reader" : {
      "refresh-interval" : "1000"
    }
  },
  "partition-handling" : {
    "when-split" : "ALLOW_READ_WRITES",
    "merge-policy" : "PREFERRED_NON_NULL"
  },
  "persistence" : {
    "passivation" : false
  }
}

```

YAML

```

distributedCache:
  mode: "SYNC"
  owners: "2"
  segments: "256"
  capacityFactor: "1.0"
  l1Lifespan: "5000"
  statistics: "true"
  encoding:
    mediaType: "application/x-protostream"
  locking:
    isolation: "REPEATABLE_READ"
  transaction:
    mode: "FULL_XA"
    locking: "OPTIMISTIC"
  expiration:
    lifespan: "5000"
    maxIdle: "1000"
  memory:
    maxCount: "1000000"
    whenFull: "REMOVE"
  indexing:
    enabled: "true"
    storage: "local-heap"
  indexReader:
    refreshInterval: "1000"

```

```

partitionHandling:
  whenSplit: "ALLOW_READ_WRITES"
  mergePolicy: "PREFERRED_NON_NULL"
persistence:
  passivation: "false"
# Persistent storage configuration.

```

レプリケートされたキャッシュ

XML

```

<replicated-cache segments="256"
  mode="SYNC"
  statistics="true">
  <encoding media-type="application/x-protostream"/>
  <locking isolation="REPEATABLE_READ"/>
  <transaction mode="FULL_XA"
    locking="OPTIMISTIC"/>
  <expiration lifespan="5000"
    max-idle="1000" />
  <memory max-count="1000000"
    when-full="REMOVE"/>
  <indexing enabled="true"
    storage="local-heap">
    <index-reader refresh-interval="1000"/>
  </indexing>
  <partition-handling when-split="ALLOW_READ_WRITES"
    merge-policy="PREFERRED_NON_NULL"/>
  <persistence passivation="false">
    <!-- Persistent storage configuration. -->
  </persistence>
</replicated-cache>

```

JSON

```

{
  "replicated-cache": {
    "mode": "SYNC",
    "segments": "256",
    "statistics": "true",
    "encoding": {
      "media-type": "application/x-protostream"
    },
    "locking": {
      "isolation": "REPEATABLE_READ"
    },
    "transaction": {
      "mode": "FULL_XA",
      "locking": "OPTIMISTIC"
    },
    "expiration" : {
      "lifespan" : "5000",
      "max-idle" : "1000"
    },
  },
}

```

```

    "memory": {
      "max-count": "1000000",
      "when-full": "REMOVE"
    },
    "indexing" : {
      "enabled" : true,
      "storage" : "local-heap",
      "index-reader" : {
        "refresh-interval" : "1000"
      }
    },
    "partition-handling" : {
      "when-split" : "ALLOW_READ_WRITES",
      "merge-policy" : "PREFERRED_NON_NULL"
    },
    "persistence" : {
      "passivation" : false
    }
  }
}

```

YAML

```

replicatedCache:
  mode: "SYNC"
  segments: "256"
  statistics: "true"
  encoding:
    mediaType: "application/x-protostream"
  locking:
    isolation: "REPEATABLE_READ"
  transaction:
    mode: "FULL_XA"
    locking: "OPTIMISTIC"
  expiration:
    lifespan: "5000"
    maxIdle: "1000"
  memory:
    maxCount: "1000000"
    whenFull: "REMOVE"
  indexing:
    enabled: "true"
    storage: "local-heap"
    indexReader:
      refreshInterval: "1000"
  partitionHandling:
    whenSplit: "ALLOW_READ_WRITES"
    mergePolicy: "PREFERRED_NON_NULL"
  persistence:
    passivation: "false"
    # Persistent storage configuration.

```

複数のキャッシュ

XML

```

<infinispan
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:infinispan:config:13.0 https://infinispan.org/schemas/infinispan-config-13.0.xsd
                        urn:infinispan:server:13.0 https://infinispan.org/schemas/infinispan-server-13.0.xsd"
  xmlns="urn:infinispan:config:13.0"
  xmlns:server="urn:infinispan:server:13.0">
  <cache-container name="default"
    statistics="true">
    <distributed-cache name="mycacheone"
      mode="ASYNC"
      statistics="true">
      <encoding media-type="application/x-protostream"/>
      <expiration lifespan="300000"/>
      <memory max-size="400MB"
        when-full="REMOVE"/>
    </distributed-cache>
    <distributed-cache name="mycachetwo"
      mode="SYNC"
      statistics="true">
      <encoding media-type="application/x-protostream"/>
      <expiration lifespan="300000"/>
      <memory max-size="400MB"
        when-full="REMOVE"/>
    </distributed-cache>
  </cache-container>
</infinispan>

```

YAML

```

infinispan:
  cacheContainer:
    name: "default"
    statistics: "true"
  caches:
    mycacheone:
      distributedCache:
        mode: "ASYNC"
        statistics: "true"
        encoding:
          mediaType: "application/x-protostream"
        expiration:
          lifespan: "300000"
        memory:
          maxSize: "400MB"
          whenFull: "REMOVE"
    mycachetwo:
      distributedCache:
        mode: "SYNC"
        statistics: "true"
        encoding:
          mediaType: "application/x-protostream"
        expiration:
          lifespan: "300000"

```

```
memory:
  maxSize: "400MB"
  whenFull: "REMOVE"
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "name" : "default",
      "statistics" : "true",
      "caches" : {
        "mycacheone" : {
          "distributed-cache" : {
            "mode": "ASYNC",
            "statistics": "true",
            "encoding": {
              "media-type": "application/x-protostream"
            },
            "expiration" : {
              "lifespan" : "300000"
            },
            "memory": {
              "max-size": "400MB",
              "when-full": "REMOVE"
            }
          }
        },
        "mycachetwo" : {
          "distributed-cache" : {
            "mode": "SYNC",
            "statistics": "true",
            "encoding": {
              "media-type": "application/x-protostream"
            },
            "expiration" : {
              "lifespan" : "300000"
            },
            "memory": {
              "max-size": "400MB",
              "when-full": "REMOVE"
            }
          }
        }
      }
    }
  }
}
```

関連情報

- [Data Grid 設定スキーマ参照](#)
- [infinispan-config-8.3.xsd](#)

3.2. キャッシュテンプレートの追加

Data Grid スキーマには、テンプレートの作成に使用できる ***-cache-configuration** 要素が含まれます。その後、同じ設定を複数回使用して、オンデマンドでキャッシュを作成することができます。

手順

1. Data Grid 設定を開いて編集します。
2. 適切な ***-cache-configuration** 要素またはオブジェクトをキャッシュマネージャーに追加します。
3. Data Grid 設定を保存して閉じます。

キャッシュテンプレートの例

XML

```
<infinispan>
  <cache-container>
    <distributed-cache-configuration name="my-dist-template"
      mode="SYNC"
      statistics="true">
      <encoding media-type="application/x-protostream"/>
      <memory max-count="1000000"
        when-full="REMOVE"/>
      <expiration lifespan="5000"
        max-idle="1000"/>
    </distributed-cache-configuration>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan": {
    "cache-container": {
      "distributed-cache-configuration": {
        "name": "my-dist-template",
        "mode": "SYNC",
        "statistics": "true",
        "encoding": {
          "media-type": "application/x-protostream"
        },
        "expiration": {
          "lifespan": "5000",
          "max-idle": "1000"
        },
        "memory": {
          "max-count": "1000000",
          "when-full": "REMOVE"
        }
      }
    }
  }
}
```

```
}
}
}
```

YAML

```
infinispan:
  cacheContainer:
    distributedCacheConfiguration:
      name: "my-dist-template"
      mode: "SYNC"
      statistics: "true"
      encoding:
        mediaType: "application/x-protostream"
      expiration:
        lifespan: "5000"
        maxIdle: "1000"
      memory:
        maxCount: "1000000"
        whenFull: "REMOVE"
```

3.2.1. テンプレートからのキャッシュの作成

設定テンプレートからキャッシュを作成します。

ヒント

リモートキャッシュのテンプレートは、Data Grid コンソールの **Cache templates** メニューから利用できます。

前提条件

- キャッシュマネージャーに少なくとも1つのキャッシュテンプレートを追加します。

手順

1. Data Grid 設定を開いて編集します。
2. キャッシュが **configuration** 属性またはフィールドを継承するテンプレートを指定します。
3. Data Grid 設定を保存して閉じます。

テンプレートから継承されたキャッシュ設定

XML

```
<distributed-cache configuration="my-dist-template" />
```

JSON

```
{
  "distributed-cache": {
```

```

    "configuration": "my-dist-template"
  }
}

```

YAML

```

distributedCache:
  configuration: "my-dist-template"

```

3.2.2. キャッシュテンプレートの継承

キャッシュ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、上書きすることができます。

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。

さらに、複数の値を持つ要素にはテンプレート継承が追加されます。別のテンプレートから継承するキャッシュは、そのテンプレートから値をマージし、プロパティを上書きできます。

テンプレート継承の例

XML

```

<infinispan>
  <cache-container>
    <distributed-cache-configuration name="base-template">
      <expiration lifespan="5000"/>
    </distributed-cache-configuration>
    <distributed-cache-configuration name="extended-template"
      configuration="base-template">
      <encoding media-type="application/x-protostream"/>
      <expiration lifespan="10000"
        max-idle="1000"/>
    </distributed-cache-configuration>
  </cache-container>
</infinispan>

```

JSON

```

{
  "infinispan": {
    "cache-container": {
      "caches": {
        "base-template": {
          "distributed-cache-configuration": {
            "expiration": {
              "lifespan": "5000"
            }
          }
        },
        "extended-template": {
          "distributed-cache-configuration": {

```

```

        "configuration" : "base-template",
        "encoding": {
            "media-type": "application/x-protostream"
        },
        "expiration" : {
            "lifespan" : "10000",
            "max-idle" : "1000"
        }
    }
}
}
}
}
}
}
}
}
}
}

```

YAML

```

infinispan:
  cacheContainer:
    caches:
      base-template:
        distributedCacheConfiguration:
          expiration:
            lifespan: "5000"
      extended-template:
        distributedCacheConfiguration:
          configuration: "base-template"
          encoding:
            mediaType: "application/x-protostream"
          expiration:
            lifespan: "10000"
            maxIdle: "1000"

```

3.2.3. キャッシュテンプレートのワイルドカード

ワイルドカードをキャッシュ設定テンプレート名に追加できます。名前がワイルドカードに一致するキャッシュを作成すると、Data Grid は設定テンプレートを適用します。



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。

テンプレートワイルドカードの例

XML

```

<infinispan>
  <cache-container>
    <distributed-cache-configuration name="async-dist-cache-*"
      mode="ASYNC"
      statistics="true">
    <encoding media-type="application/x-protostream"/>

```

```

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

```

JSON

```

{
  "infinispan" : {
    "cache-container" : {
      "distributed-cache-configuration" : {
        "name" : "async-dist-cache-*",
        "mode": "ASYNC",
        "statistics": "true",
        "encoding": {
          "media-type": "application/x-protostream"
        }
      }
    }
  }
}

```

YAML

```

infinispan:
  cacheContainer:
    distributedCacheConfiguration:
      name: "async-dist-cache-*"
      mode: "ASYNC"
      statistics: "true"
      encoding:
        mediaType: "application/x-protostream"

```

上記の例では、async-dist-cache-prod という名前のキャッシュを作成する場合、Data Grid は **async-dist-cache-*** テンプレートの設定を使用します。

3.2.4. 複数の XML ファイルからのキャッシュテンプレート

キャッシュ設定テンプレートを複数の XML ファイルに分割して、粒度を柔軟に参照し、XML 包含 (XInclude) で参照します。



注記

Data Grid は、XInclude 仕様の最小限のサポートを提供します。つまり、**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を使用できません。

また、XInclude を使用するには **xmlns:xi="http://www.w3.org/2001/XInclude"** namespace を **infinispan.xml** に追加する必要があります。

Xinclude キャッシュテンプレート

```

<infinispan xmlns:xi="http://www.w3.org/2001/XInclude">

```

```
<cache-container default-cache="cache-1">
  <!-- References files that contain cache configuration templates. -->
  <xi:include href="distributed-cache-template.xml" />
  <xi:include href="replicated-cache-template.xml" />
</cache-container>
</infinispan>
```

Data Grid は、設定フラグメントで利用できる **infinispan-config-fragment-13.0.xsd** スキーマも提供します。

設定フラグメントスキーマ

```
<local-cache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:infinispan:config:13.0 https://infinispan.org/schemas/infinispan-
config-fragment-13.0.xsd"
  xmlns="urn:infinispan:config:13.0"
  name="mycache"/>
```

関連情報

- [XInclude 仕様](#)

3.3. リモートキャッシュの作成

ランタイム時にリモートキャッシュを作成すると、Data Grid Server はクラスター全体で設定を同期し、全ノードがコピーを持つようにします。このため、常に以下のメカニズムを使用してリモートキャッシュを動的に作成する必要があります。

- Data Grid コンソール
- Data Grid コマンドラインインターフェイス (CLI)
- Hot Rod または HTTP クライアント

3.3.1. デフォルトの Cache Manager

Data Grid Server は、リモートキャッシュのライフサイクルを制御するデフォルトの Cache Manager を提供します。Data Grid Server を起動すると、Cache Manager が自動的にインスタンス化されるため、リモートキャッシュや Protobuf スキーマなどの他のリソースを作成および削除できます。

Data Grid Server を起動してユーザー認証情報を追加したら、Cache Manager の詳細を表示し、Data Grid コンソールからクラスター情報を取得できます。

- 任意のブラウザーで **127.0.0.1:11222** を開きます。

コマンドラインインターフェイス (CLI) または REST API を使用して Cache Manager に関する情報を取得することもできます。

CLI

デフォルトのコンテナで **describe** コマンドを使用します。

```
[//containers/default]> describe
```

REST

任意のブラウザで **127.0.0.1:11222/rest/v2/cache-managers/default/** を開きます。

デフォルトの Cache Manager の設定

XML

```
<infinispan>
  <!-- Creates a Cache Manager named "default" and enables metrics. -->
  <cache-container name="default"
    statistics="true">
    <!-- Adds cluster transport that uses the default JGroups TCP stack. -->
    <transport cluster="${infinispan.cluster.name:cluster}"
      stack="${infinispan.cluster.stack:tcp}"
      node-name="${infinispan.node.name:}" />
    <!-- Requires user permission to access caches and perform operations. -->
    <security>
      <authorization/>
    </security>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "jgroups" : {
      "transport" : "org.infinispan.remoting.transport.jgroups.JGroupsTransport"
    },
    "cache-container" : {
      "name" : "default",
      "statistics" : "true",
      "transport" : {
        "cluster" : "cluster",
        "node-name" : "",
        "stack" : "tcp"
      },
      "security" : {
        "authorization" : {}
      }
    }
  }
}
```

YAML

```
infinispan:
  jgroups:
    transport: "org.infinispan.remoting.transport.jgroups.JGroupsTransport"
  cacheContainer:
    name: "default"
    statistics: "true"
    transport:
      cluster: "cluster"
```

```
nodeName: ""  
stack: "tcp"  
security:  
authorization: ~
```

3.3.2. Data Grid コンソールを使用したキャッシュの作成

Data Grid コンソールを使用して、任意の Web ブラウザーから直感的なビジュアルインターフェイスでリモートキャッシュを作成します。

前提条件

- **admin** パーミッションを持つ Data Grid ユーザーを作成します。
- 1つ以上の Data Grid Server インスタンスを起動します。
- Data Grid キャッシュ設定があります。

手順

1. 任意のブラウザーで **127.0.0.1:11222/console/** を開きます。
2. **Create Cache** を選択し、プロセスを Data Grid コンソールガイドの手順に従ってください。

3.3.3. Data Grid CLI を使用したリモートキャッシュの作成

Data Grid コマンドラインインターフェイス (CLI) を使用して、Data Grid Server にリモートキャッシュを追加します。

前提条件

- **admin** パーミッションを持つ Data Grid ユーザーを作成します。
- 1つ以上の Data Grid Server インスタンスを起動します。
- Data Grid キャッシュ設定があります。

手順

1. CLI を起動し、プロンプトが表示されたら認証情報を入力します。

```
bin/cli.sh
```

2. **create cache** コマンドを使用してリモートキャッシュを作成します。
たとえば、以下のように **mycache.xml** という名前のファイルから"mycache"という名前のキャッシュを作成します。

```
create cache --file=mycache.xml mycache
```

検証

1. **ls** コマンドを使用して、すべてのリモートキャッシュを一覧表示します。


```
ls caches
mycache
```

2. **describe** コマンドでキャッシュ設定を表示します。

```
describe caches/mycache
```

3.3.4. Hot Rod クライアントからのリモートキャッシュの作成

Data Grid Hot Rod API を使用して、Java、C++、.NET/C{hash}、JS クライアントなどから Data Grid Server にリモートキャッシュを作成します。

この手順では、最初のアクセスでリモートキャッシュを作成する Hot Rod Java クライアントを使用する方法を示します。他の Hot Rod クライアントのコード例は、[Data Grid Tutorials](#) を参照してください。

前提条件

- **admin** パーミッションを持つ Data Grid ユーザーを作成します。
- 1つ以上の Data Grid Server インスタンスを起動します。
- Data Grid キャッシュ設定があります。

手順

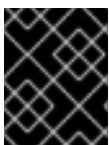
- **ConfigurationBuilder** の一部として **remoteCache()** メソッドを呼び出します。
- クラスパスの **hotrod-client.properties** ファイルで **configuration** または **configuration_uri** プロパティを設定します。

ConfigurationBuilder

```
File file = new File("path/to/infinispan.xml")
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.remoteCache("another-cache")
    .configuration("<distributed-cache name=\"another-cache\"/>");
builder.remoteCache("my.other.cache")
    .configurationURI(file.toURI());
```

hotrod-client.properties

```
infinispan.client.hotrod.cache.another-cache.configuration=<distributed-cache name=\"another-cache\"/>
infinispan.client.hotrod.cache.[my.other.cache].configuration_uri=file:///path/to/infinispan.xml
```



重要

リモートキャッシュの名前に **.** が含まれる場合は、**hotrod-client.properties** ファイルを使用する場合は角括弧で囲む必要があります。

関連情報

- [Hot Rod Client Configuration](#)
- [org.infinispan.client.hotrod.configuration.RemoteCacheConfigurationBuilder](#)

3.3.5. REST API を使用したリモートキャッシュの作成

Data Grid REST API を使用して、適切な HTTP クライアントから Data Grid Server でリモートキャッシュを作成します。

前提条件

- **admin** パーミッションを持つ Data Grid ユーザーを作成します。
- 1つ以上の Data Grid Server インスタンスを起動します。
- Data Grid キャッシュ設定があります。

手順

- ペイロードにキャッシュ設定を指定して `/rest/v2/caches/<cache_name>` に **POST** 要求を呼び出します。

関連情報

- [Creating and Managing Caches with the REST API](#)

3.4. 組み込みキャッシュの作成

Data Grid は、プログラムを使用して Cache Manager と組み込みキャッシュライフサイクルの両方を制御できる **EmbeddedCacheManager** API を提供します。

3.4.1. Data Grid のプロジェクトへの追加

Data Grid をプロジェクトに追加して、アプリケーションで組み込みキャッシュを作成します。

前提条件

- Maven リポジトリから Data Grid アーティファクトを取得するようにプロジェクトを設定します。

手順

- 以下のように、**infinispan-core** アーティファクトを **pom.xml** の依存関係として追加します。

```
<dependencies>
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-core</artifactId>
</dependency>
</dependencies>
```

3.4.2. 組み込みキャッシュの設定

Data Grid は、キャッシュマネージャーと、埋め込みキャッシュを設定する **ConfigurationBuilder** API を制御する **GlobalConfigurationBuilder** API を提供します。

前提条件

- **infinispan-core** アーティファクトを **pom.xml** の依存関係として追加します。

手順

1. デフォルトのキャッシュマネージャーを初期化し、埋め込みキャッシュを追加できます。
2. **ConfigurationBuilder** API を使用して、埋め込みキャッシュを1つ以上追加します。
3. クラスターのすべてのノードで組み込みキャッシュを作成するか、すでに存在するキャッシュを返す **getOrCreateCache()** メソッドを呼び出します。

```
// Set up a clustered cache manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();
// Initialize the default cache manager.
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());
// Create a distributed cache with synchronous replication.
ConfigurationBuilder builder = new ConfigurationBuilder();
    builder.clustering().cacheMode(CacheMode.DIST_SYNC);
// Obtain a volatile cache.
Cache<String, String> cache =
cacheManager.administration().withFlags(CacheContainerAdmin.AdminFlag.VOLATILE).getOrCreateC
ache("myCache", builder.build());
```

関連資料

- [EmbeddedCacheManager](#)
- [EmbeddedCacheManager Configuration](#)
- [org.infinispan.configuration.global.GlobalConfiguration](#)
- [org.infinispan.configuration.cache.ConfigurationBuilder](#)

第4章 DATA GRID 統計および JMX 監視の有効化および設定

Data Grid は、JMX MBean をエクスポートしたり、Cache Manager およびキャッシュ統計を提供できます。

4.1. DATA GRID メトリクスの設定

Data Grid は、MicroProfile Metrics API と互換性のあるメトリクスを生成します。

- 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。
- 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。

デフォルトでは、Data Grid は統計を有効にするとゲージを生成しますが、ヒストグラムを生成するように設定することもできます。

手順

1. Data Grid 設定を開いて編集します。
2. **metrics** 要素またはオブジェクトをキャッシュコンテナに追加します。
3. **gauges** 属性またはフィールドを使用してゲージを有効または無効にします。
4. **histograms** 属性またはフィールドでヒストグラムを有効または無効にします。
5. クライアント設定を保存して閉じます。

メトリクスの設定

XML

```
<infinispan>
  <cache-container statistics="true">
    <metrics gauges="true"
      histograms="true" />
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan": {
    "cache-container": {
      "statistics": "true",
      "metrics": {
        "gauges": "true",
        "histograms": "true"
      }
    }
  }
}
```

YAML

```

infinispan:
  cacheContainer:
    statistics: "true"
  metrics:
    gauges: "true"
    histograms: "true"

```

関連資料

- [Eclipse MicroProfile Metrics](#)

4.2. JMX MBEAN の登録

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。統計を有効にする必要もあります。そうしないと、Data Grid は JMX MBean のすべての統計属性に **0** 値を提供します。

手順

1. Data Grid 設定を開いて編集します。
2. **jmx** 要素またはオブジェクトをキャッシュコンテナに追加し、**enabled** 属性またはフィールドの値として **true** を指定します。
3. **domain** 属性またはフィールドを追加し、必要に応じて JMX MBean が公開されるドメインを指定します。
4. クライアント設定を保存して閉じます。

JMX の設定

XML

```

<infinispan>
  <cache-container statistics="true">
    <jmx enabled="true"
      domain="example.com"/>
  </cache-container>
</infinispan>

```

JSON

```

{
  "infinispan": {
    "cache-container": {
      "statistics": "true",
      "jmx": {
        "enabled": "true",
        "domain": "example.com"
      }
    }
  }
}

```

```
}
}
}
```

YAML

```
infinispan:
  cacheContainer:
    statistics: "true"
  jmx:
    enabled: "true"
    domain: "example.com"
```

4.2.1. JMX リモートポートの有効化

一意のリモート JMX ポートを提供し、JMXServiceURL 形式の接続を介して Data Grid MBean を公開します。

次のいずれかの方法を使用して、リモート JMX ポートを有効にできます。

- Data Grid サーバーセキュリティーレلمの1つに対する認証を必要とするリモート JMX ポートを有効にします。
- 標準の Java 管理設定オプションを使用して、手動でリモート JMX ポートを有効にします。

前提条件

- 認証付きのリモート JMX の場合、デフォルトのセキュリティーレلمを使用してユーザーロールを定義します。ユーザーが JMX リソースにアクセスするには、読み取り/書き込みアクセス権を持つ **controlRole** または読み取り専用アクセス権を持つ **monitorRole** が必要です。

手順

次のいずれかの方法を使用して、リモート JMX ポートを有効にして Data Grid サーバーを起動します:

* ポート **9999** を介してリモート JMX を有効にします。

+

```
bin/server.sh --jmx 9999
```

+



警告

SSL を無効にしてリモート JMX を使用することは、本番環境向けではありません。

- 起動時に以下のシステムプロパティを Data Grid サーバーに渡します。

```
bin/server.sh -Dcom.sun.management.jmxremote.port=9999 -  
Dcom.sun.management.jmxremote.authenticate=false -  
Dcom.sun.management.jmxremote.ssl=false
```



警告

認証または SSL なしでリモート JMX を有効にすることは安全ではなく、どのような環境でも推奨されません。認証と SSL を無効にすると、権限のないユーザーがサーバーに接続し、そこでホストされているデータにアクセスできるようになります。

関連情報

- [セキュリティレールの作成](#)

4.2.2. 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](#)

4.2.3. 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。

前提条件

- **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。
- JMX MBean を登録するように Data Grid を設定します。

手順

1. Data Grid 設定を開いて編集します。

2. **mbean-server-lookup** 属性またはフィールドをキャッシュマネージャーの JMX 設定に追加します。
3. **MBeanServerLookup** 実装の完全修飾名 (FQN) を指定します。
4. クライアント設定を保存して閉じます。

JMX MBean サーバールックアップの設定

XML

```
<infinispan>
  <cache-container statistics="true">
    <jmx enabled="true"
      domain="example.com"
      mbean-server-lookup="com.example.MyMBeanServerLookup"/>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "jmx" : {
        "enabled" : "true",
        "domain" : "example.com",
        "mbean-server-lookup" : "com.example.MyMBeanServerLookup"
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    statistics: "true"
  jmx:
    enabled: "true"
    domain: "example.com"
    mbeanServerLookup: "com.example.MyMBeanServerLookup"
```


第5章 JVM メモリー使用量の設定

以下を行って、Data Grid がデータを JVM メモリーに保存する方法を制御します。

- キャッシュからデータを自動的に削除するエビクションを使用した JVM メモリー使用量の管理
- ライフスパンを追加し、エントリー失効させるために最大アイドル時間を追加し、古くなったデータを防ぎます。
- データをオフヒープのネイティブメモリーに保存するための Data Grid の設定

5.1. デフォルトのメモリー設定

デフォルトでは、Data Grid はキャッシュエントリーを JVM ヒープにオブジェクトとして保存します。時間の経過とともに、アプリケーションによってエントリーが追加されると、キャッシュのサイズが JVM で利用可能なメモリー量を超える可能性があります。同様に、Data Grid がプライマリーデータストアではない場合、エントリーに古いデータが含まれることを意味します。

XML

```
<distributed-cache>
  <memory storage="HEAP"/>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "memory": {
      "storage": "HEAP"
    }
  }
}
```

YAML

```
distributedCache:
  memory:
    storage: "HEAP"
```

5.2. エビクションと有効期限

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。

5.3. DATA GRID キャッシュを使用したエビクション

エビクションを使用すると、以下の 2 つの方法のいずれかで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。

- エントリーの合計数 (**max-count**)。
- メモリーの最大量 (**max-size**)。

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 がエビクトした後もエントリーが利用可能な状態を維持し、データの一貫性を防ぐためには、永続ストレージを設定する必要があります。

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。

5.3.1. エビクションストラテジー

Data Grid エビクションを設定する場合は、以下を指定します。

- データコンテナの最大サイズ。
- キャッシュがしきい値に達するとエントリーを削除するストラテジー。

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。

エビクションストラテジーについての詳細は、スキーマ参照を参照してください。



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。

関連情報

- [Caffeine](#)
- [Data Grid 設定スキーマ参照](#)

5.3.2. 最大カウントエビクションの設定

Data Grid キャッシュのサイズをエントリーの合計数に制限します。

手順

1. Data Grid 設定を開いて編集します。
2. Data Grid が **max-count** 属性または **maxCount()** メソッドのいずれかでエビクションを実行する前のキャッシュに含まれるエントリーの合計数を指定します。
3. 以下のいずれかをエビクションストラテジーとして設定し、Data Grid が **when-full** 属性または **whenFull()** メソッドを持つエントリーを削除する方法を制御します。
 - **REMOVE** Data Grid はエビクションを実行します。これはデフォルトのストラテジーです。
 - **MANUAL** 組み込みキャッシュのエビクションを手動で実行します。
 - **EXCEPTION** Data Grid は、エントリーをエビクトするのではなく、例外を出力します。
4. Data Grid 設定を保存して閉じます。

最大数エビクション

以下の例では、キャッシュに合計 500 エントリーが含まれ、新しいエントリーが作成されると、Data Grid はエントリーを削除します。

XML

```
<distributed-cache>
  <memory max-count="500" when-full="REMOVE"/>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "memory": {
      "max-count": "500",
```

```

    "when-full" : "REMOVE"
  }
}
}

```

YAML

```

distributedCache:
  memory:
    maxCount: "500"
    whenFull: "REMOVE"

```

ConfigurationBuilder

```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.memory().maxCount(500).whenFull(EvictionStrategy.REMOVE);

```

関連情報

- [Data Grid 設定スキーマ参照](#)
- [org.infinispan.configuration.cache.MemoryConfigurationBuilder](#)

5.3.3. 最大サイズのエビクションの設定

Data Grid キャッシュのサイズを最大メモリー容量に制限します。

手順

1. Data Grid 設定を開いて編集します。
2. **application/x-protostream** をキャッシュエンコーディングのメディアタイプとして指定します。
最大サイズのエビクションを使用するには、バイナリーメディアタイプを指定する必要があります。
3. Data Grid が **max-size** 属性または **maxSize()** メソッドで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 (バイト単位) を設定します。
4. 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。
5. 以下のいずれかをエビクションストラテジーとして設定し、Data Grid が **when-full** 属性または **whenFull()** メソッドを持つエントリーのいずれかを削除する方法を制御します。
 - **REMOVE** Data Grid はエビクションを実行します。これはデフォルトのストラテジーです。
 - **MANUAL** 組み込みキャッシュのエビクションを手動で実行します。
 - **EXCEPTION** Data Grid は、エントリーをエビクトするのではなく、例外を出力します。
6. Data Grid 設定を保存して閉じます。

最大サイズのエビクション

以下の例では、キャッシュのサイズが 1.5 GB(ギガバイト) に達すると、Data Grid はエントリーを削除します。

XML

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

JSON

```
{
  "distributed-cache" : {
    "encoding" : {
      "media-type" : "application/x-protostream"
    },
    "memory" : {
      "max-size" : "1.5GB",
      "when-full" : "REMOVE"
    }
  }
}
```

YAML

```
distributedCache:
  encoding:
    mediaType: "application/x-protostream"
  memory:
    maxSize: "1.5GB"
    whenFull: "REMOVE"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.encoding().mediaType("application/x-protostream")
    .memory()
    .maxSize("1.5GB")
    .whenFull(EvictionStrategy.REMOVE);
```

関連情報

- [Data Grid 設定スキーマ参照](#)
- [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](#)

5.3.4. 手動エビクション

手動エビクションストラテジーを選択する場合、Data Grid はエビクションを実行しません。これは、**evict()** メソッドで手動で行う必要があります。

組み込みキャッシュでのみ手動のエビクションを使用する必要があります。リモートキャッシュの場合は、**REMOVE** または **EXCEPTION** エビクションストラテジーで Data Grid を常に設定する必要があります。



注記

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。

XML

```
<distributed-cache>
  <memory max-count="500" when-full="MANUAL"/>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "memory": {
      "max-count": "500",
      "when-full": "MANUAL"
    }
  }
}
```

YAML

```
distributedCache:
  memory:
    maxCount: "500"
    whenFull: "MANUAL"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.encoding().mediaType("application/x-protostream")
    .memory()
    .maxSize("1.5GB")
    .whenFull(EvictionStrategy.REMOVE);
```

5.3.5. 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。以下の例のように、エビクションをパッシベーションを有効にする場合は、常にエビクションを有効にする必要があります。

XML

```
<distributed-cache>
  <persistence passivation="true">
    <!-- Persistent storage configuration. -->
  </persistence>
  <memory max-count="100"/>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "memory": {
      "max-count": "100"
    },
    "persistence": {
      "passivation": true
    }
  }
}
```

YAML

```
distributedCache:
  memory:
    maxCount: "100"
  persistence:
    passivation: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.memory().maxCount(100);
builder.persistence().passivation(true); //Persistent storage configuration
```

5.4. ライフスパンと最大アイドル期間の有効期限

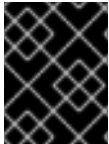
有効期限は、以下の時間制限のいずれかに到達すると、キャッシュからエントリーを削除するように Data Grid を設定します。

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、アイドルの最大有効期限は永続ストレージのキャッシュをサポートしていません。



注記

EXCEPTION エビクションストラテジーで `expiration` および `eviction` を使用する場合に、有効期限が切れているが、キャッシュから削除されないエントリーは、データコンテナのサイズに対してカウントされます。

5.4.1. 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(k1)** リクエストは **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

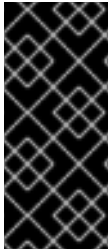
Data Grid は、期限切れのメタデータを、**long** プリミティブデータタイプとしてキャッシュエントリーに追加します。これにより、32 バイトだけにキーのサイズが増える可能性があります。

5.4.2. 有効期限のリーパー

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。

関連情報

- [org.infinispan.configuration.cache.ExpirationConfigurationBuilder](#)
- [org.infinispan.expiration.ExpirationManager](#)

5.4.3. アイドルおよびクラスター化されたキャッシュの最大数

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。

5.4.4. キャッシュのライフサイクルと最大アイドル時間の設定

キャッシュ内のすべてのエントリーの有効期間と最大アイドル時間を設定します。

手順

1. Data Grid 設定を開いて編集します。
2. **lifespan** 属性または **lifespan()** メソッドでエントリーがキャッシュ内に留まることができる期間をミリ秒単位で指定します。

3. **max-idle** 属性または **maxIdle()** メソッドを使用した最後のアクセス後にエントリーがアイドル状態でいられる期間をミリ秒単位で指定します。
4. Data Grid 設定を保存して閉じます。

Data Grid キャッシュの有効期限

以下の例では、Data Grid は、最後のアクセス時間が過ぎてから 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ようにします。

XML

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

JSON

```
{
  "replicated-cache": {
    "expiration": {
      "lifespan": "5000",
      "max-idle": "1000"
    }
  }
}
```

YAML

```
replicatedCache:
  expiration:
    lifespan: "5000"
    maxIdle: "1000"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.expiration().lifespan(5000, TimeUnit.MILLISECONDS)
    .maxIdle(1000, TimeUnit.MILLISECONDS);
```

5.4.5. エントリーごとのライフスパンおよび最大アイドル時間の設定

各エントリーの `lifespan` と `maximum idle times` を指定します。ライフスパンとアイドル時間をエントリーに追加する場合、これらの値はキャッシュの有効期限の設定よりも優先されます。



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、Data Grid は有効期限の値とエントリーを永続ストレージに書き込みます。

手順

- リモートキャッシュでは、Data Grid コンソールを使用して、有効期限とアイドル時間を対話的にエントリーに追加できます。
Data Grid コマンドラインインターフェイス (CLI) で、**put** コマンドで **--max-idle=** および **--ttl=** 引数を使用します。
- リモートキャッシュと組み込みキャッシュの両方に対して、**cache.put()** 呼び出しでライフスパンと最大アイドル時間を追加できます。

```
//Lifespan of 5 seconds.
//Maximum idle time of 1 second.
cache.put("hello", "world", 5, TimeUnit.SECONDS, 1, TimeUnit.SECONDS);

//Lifespan is disabled with a value of -1.
//Maximum idle time of 1 second.
cache.put("hello", "world", -1, TimeUnit.SECONDS, 1, TimeUnit.SECONDS);
```

関連情報

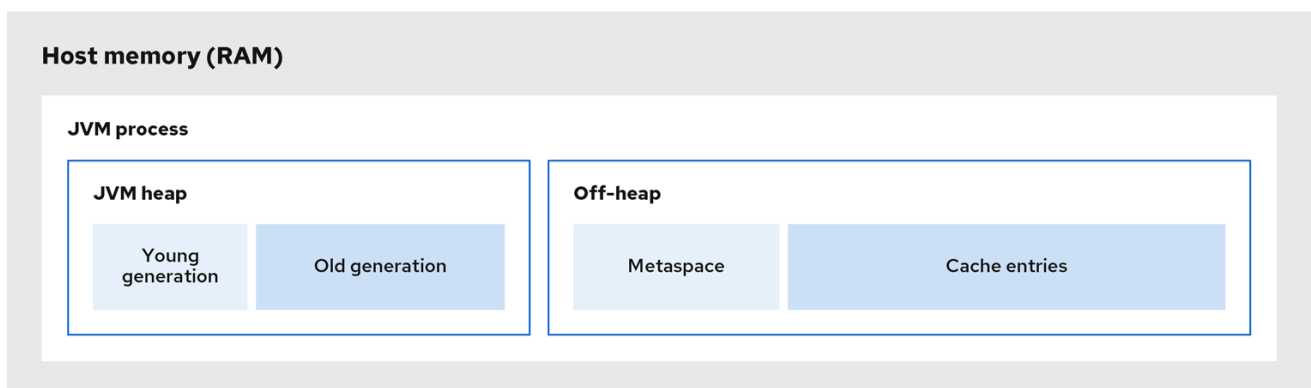
- [org.infinispan.configuration.cache.ExpirationConfigurationBuilder](#)
- [org.infinispan.expiration.ExpirationManager](#)

5.5. JVM ヒープおよび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、管理された JVM メモリー領域外のネイティブメモリーが、データで占有されます。

以下の図は、Data Grid が実行している JVM プロセスのメモリー領域を簡略して示しています。

図5.1 JVM メモリー領域



184_Data_Grid_0921

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ー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ーを 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ー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。

オフヒープメモリー

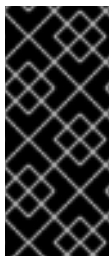
- エントリーごとに少ないメモリーを使用します。
- Garbage Collector (GC) の実行を回避するために、JVM 全体のパフォーマンスを改善します。

ただし、JVM ヒープダンプはオフヒープメモリーに保存されているエントリーを表示しない点が短所です。

5.5.1. オフヒープデータストレージ

オフヒープキャッシュにエントリーを追加すると、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ーの検索に使用するアドレスポインターが含まれます。



重要

Data Grid はキャッシュエントリー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。

オブジェクトの等価性

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ライズされた **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。

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、その後に最も近い 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。

5.5.2. オフヒープメモリーの設定

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トリーを保存するように Data Grid を設定します。

手順

1. Data Grid 設定を開いて編集します。
2. **OFF_HEAP** を **storage** 属性または **storage()** メソッドの値として設定します。
3. エビクションを設定して、キャッシュのサイズに境界を設定します。

4. Data Grid 設定を保存して閉じます。

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。

XML

```
<replicated-cache>
  <memory storage="OFF_HEAP" max-count="500"/>
</replicated-cache>
```

JSON

```
{
  "replicated-cache": {
    "memory": {
      "storage": "OBJECT",
      "max-count": "500"
    }
  }
}
```

YAML

```
replicatedCache:
  memory:
    storage: "OFF_HEAP"
    maxCount: "500"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.memory().storage(StorageType.OFF_HEAP).maxCount(500);
```

関連情報

- [Data Grid 設定スキーマ参照](#)
- [org.infinispan.configuration.cache.MemoryConfigurationBuilder](#)

第6章 CONFIGURING PERSISTENT STORAGE

Data Grid は、キャッシュストアとローダーを使用して永続ストレージと対話します。

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。

6.1. 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。

6.1.1. パッシベーションの仕組み

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ーは、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ベートされたエントリー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、以前にパッシベートされたエントリーは、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

6.2. ライトスルーキャッシュストア

ライトスルーは、メモリーへの書き込みと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、キャッシュおよびキャッシュストアが同時に更新され、キャッシュストアが常にキャッシュと一致していることです。

ただし、ライトスルーモードは、キャッシュ操作にレイテンシーを直接追加する必要があるため、パフォーマンスが低下する可能性があります。

ライトスルー設定

Data Grid は、キャッシュに write-behind 設定を明示的に追加しない限り、ライトスルーモードを使用します。write-through モードを設定する別の要素またはメソッドはありません。

たとえば、以下の設定は、ライトスルーモードを暗黙的に使用するキャッシュにファイルベースのストアを追加します。

```
<distributed-cache>
  <persistence passivation="false">
    <file-store fetch-state="true">
      <index path="path/to/index" />
      <data path="path/to/data" />
    </file-store>
  </persistence>
</distributed-cache>
```

6.3. WRITE-BEHIND キャッシュストア

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であ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、キャッシュ操作で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つ必要がないため、ライトスルーモードよりもパフォーマンス上の利点があります。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

XML


```

<distributed-cache>
  <persistence>
    <table-jdbc-store xmlns="urn:infinispan:config:store:sql:13.0"
      dialect="H2"
      shared="true"
      table-name="books">
      <connection-pool connection-url="jdbc:h2:mem:infinispan"
        username="sa"
        password="changeme"
        driver="org.h2.Driver"/>
      <write-behind modification-queue-size="2048"
        fail-silently="true"/>
    </table-jdbc-store>
  </persistence>
</distributed-cache>

```

JSON

```

{
  "distributed-cache": {
    "persistence" : {
      "table-jdbc-store": {
        "dialect": "H2",
        "shared": "true",
        "table-name": "books",
        "connection-pool": {
          "connection-url": "jdbc:h2:mem:infinispan",
          "driver": "org.h2.Driver",
          "username": "sa",
          "password": "changeme"
        },
        "write-behind" : {
          "modification-queue-size" : "2048",
          "fail-silently" : true
        }
      }
    }
  }
}

```

YAML

```

distributedCache:
  persistence:
    tableJdbcStore:
      dialect: "H2"
      shared: "true"
      tableName: "books"
    connectionPool:
      connectionUrl: "jdbc:h2:mem:infinispan"
      driver: "org.h2.Driver"
      username: "sa"
      password: "changeme"

```

```
writeBehind:
  modificationQueueSize: "2048"
  failSilently: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
  .async()
  .modificationQueueSize(2048)
  .failSilently(true);
```

サイレント失敗

write-behind 設定には、キャッシュストアが利用できない場合や、変更キューが満杯になったときに何が発生するかを制御する **fail-silently** のパラメーターが含まれます。

- **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ため、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。

6.4. 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。

6.5. 共有キャッシュストア

Data Grid キャッシュストアは、特定ノードにローカルにすることも、クラスターのすべてのノードで共有したりできます。デフォルトでは、キャッシュストアはローカル (**shared="false"**) です。

- ローカルキャッシュストアは各ノードに一意です。たとえば、ホストファイルシステムにデータを保持するファイルベースのキャッシュストアなどです。
ローカルキャッシュストアは、起動時に状態を取得し、パージし、永続ストレージから古いエントリーを読み込まないようにすることができます。
- 共有キャッシュストアを使用すると、複数のノードが同じ永続ストレージを使用できます。たとえば、複数のノードが同じデータベースにアクセスできるようにする JDBC キャッシュストアなどです。
共有キャッシュストアは、毎回の変更に対して書き込み操作を実行するバックアップノードではなく、プライマリー所有者のみが永続ストレージへの書き込むようにします。



重要

起動時に状態を取得し、パージするように、共有キャッシュストアを設定しないでください。共有キャッシュストアで状態を取得すると、パフォーマンスの問題が発生し、クラスターの起動時間が長くなります。パージするとデータが削除されます。これは、永続ストレージの望ましい動作ではありません。

ローカルキャッシュストア

```
<persistence>
  <store shared="false"
    fetch-state="true"
    purge="true"/>
</persistence>
```

共有キャッシュストア

```
<persistence>
  <store shared="true"
    fetch-state="false"
    purge="false"/>
</persistence>
```

関連情報

- [Data Grid 設定スキーマ](#)

6.6. 永続キャッシュストアを使用するトランザクション

Data Grid は、JDBC ベースのキャッシュストアでのみトランザクション操作をサポートします。キャッシュをトランザクションとして設定するには、**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。

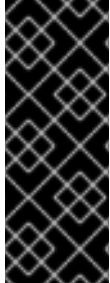
他のすべてのキャッシュストアでは、Data Grid はトランザクション操作でリストキャッシュローダーをエンリスト化しません。これにより、メモリー内のデータの変更中にトランザクションが正常に実行されたものの、キャッシュストアのデータへの変更が完全に適用されない場合は、データの不整合が生じる可能性があります。このような場合、キャッシュストアでは手動リカバリーができません。

6.7. グローバルの永続的な場所

Data Grid は、再起動後にクラスタートポロジーおよびキャッシュされたデータを復元できるようにグローバル状態を保持します。

リモートキャッシュ

Data Grid Server は、クラスターの状態を **\$RHDG_HOME/server/data** ディレクトリーに保存します。



重要

server/data ディレクトリーまたはその内容を削除または変更しないでください。Data Grid は、サーバーインスタンスを再起動すると、このディレクトリーからクラスターの状態を復元します。

デフォルト設定を変更したり、**server/data** ディレクトリーを直接変更すると、予期しない動作が発生し、データが失われる可能性があります。

組み込みキャッシュ

Data Grid は、グローバルな永続的な場所として **user.dir** システムプロパティーにデフォルト設定されます。ほとんどの場合、これはアプリケーションが開始するディレクトリーです。

クラスター化された組み込みキャッシュ (レプリケートまたは分散など) の場合は、クラスタートポロジーを復元するためにグローバルの永続的な場所を常に有効にし、設定する必要があります。

グローバルの永続的な場所外にあるファイルベースのキャッシュストアの絶対パスを設定しないでください。この場合、Data Grid は以下の例外をログに書き込みます。

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

6.7.1. グローバルの永続的な場所の設定

Data Grid がクラスター化された組み込みキャッシュのグローバル状態を保存する場所を有効にして設定します。



注記

Data Grid Server は、グローバル永続性を有効にし、デフォルトの場所を設定します。グローバル永続性を無効にしたり、リモートキャッシュのデフォルト設定を変更したりしないでください。

前提条件

- Data Grid をプロジェクトに追加します。

手順

1. 以下のいずれかの方法でグローバル状態を有効にします。
 - **global-state** 要素を Data Grid 設定に追加します。
 - **GlobalConfigurationBuilder** API で **globalState().enable()** メソッドを呼び出します。
2. グローバルの永続的な場所は各ノードに一意であるか、クラスター間で共有されるかどうかを定義します。

ロケーションのタイプ	設定
各ノードに一意	persistent-location 要素または persistentLocation() メソッド
クラスター間で共有される	shared-persistent-location 要素または sharedPersistentLocation(String) メソッド

3. Data Grid がクラスターの状態を保存するパスを設定します。
たとえば、ファイルベースのキャッシュストアは、パスはホストファイルシステムのディレクトリです。

値は以下のとおりです。

- ルートを含む完全な場所を含む絶対的な場所が含まれます。
 - ルートの場所と相対的です。
4. パスに相対値を指定する場合は、ルートロケーションに解決するシステムプロパティーも指定する必要があります。
たとえば、**global/state** をパスとして設定する Linux ホストシステムでは、以下のようになります。また、**/opt/data** ルートロケーションに解決する **my.data** プロパティーも設定します。この場合、Data Grid はグローバルの永続的な場所として **/opt/data/global/state** を使用します。

グローバルの永続的な場所設定

XML

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

JSON

```
{
  "infinispan": {
    "cache-container": {
      "global-state": {
        "persistent-location": {
          "path": "global/state",
          "relative-to": "my.data"
        }
      }
    }
  }
}
```

YAML

```
cacheContainer:
  globalState:
    persistentLocation:
      path: "global/state"
      relativeTo : "my.data"
```

GlobalConfigurationBuilder

```
new GlobalConfigurationBuilder().globalState()
    .enable()
    .persistentLocation("global/state", "my.data");
```

関連情報

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

6.8. ファイルベースのキャッシュストア

ファイルベースのキャッシュストアは、Data Grid が実行されているローカルホストのファイルシステムで永続ストレージを提供します。クラスター化されたキャッシュでは、ファイルベースのキャッシュストアは各 Data Grid ノードに固有のものです。



警告

NFS や Samba 共有などの共有ファイルシステムには、ファイルシステムベースのキャッシュストアを使用しないでください。これは、ファイルのロック機能やデータの破損が発生する可能性があるためです。

また、共有ファイルシステムでトランザクションキャッシュを使用しようとする、コミットフェーズでファイルに書き込む際に修復不能な障害が発生する可能性があります。

Soft-Index ファイルストア

SoftIndexFileStore は、ファイルベースのキャッシュストアのデフォルト実装で、追加のみのファイルのセットにデータを保存します。

追加のみのファイルの場合:

- 最大サイズに達すると、Data Grid は新しいファイルを作成し、書き込みを開始します。
- 使用率が 50% 未満の圧縮しきい値に達すると、Data Grid はエントリーを新しいファイルに上書きしてから、古いファイルを削除します。

B+ ツリー

パフォーマンスを改善するために、**SoftIndexFileStore** の追記のみのファイルは、ディスクおよびメモリーの両方に格納できる **B+ ツリー** を使用してインデックス化されます。インメモリーインデックスは Java ソフト参照を使用して、ガベージコレクション (GC) によって削除された場合に再構築してから再度要求できるようにします。

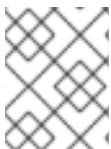
SoftIndexFileStore は Java ソフト参照を使用してインデックスをメモリーに維持するため、メモリー不足の例外を防ぐのに役立ちます。GC は、ディスクにフォールバックしながら、メモリーを過剰に消費する前にインデックスを削除します。

index 要素の **segments** 属性で宣言型で、あるいは **indexSegments()** メソッドでプログラマ的に、任意の数の B+ ツリーを設定できます。デフォルトでは、Data Grid は最大 16 の B+ ツリーを作成します。つまり、最大 16 個のインデックスを持つことができます。複数のインデックスがあると、インデックスへの同時書き込みからのボトルネックを防ぎ、Data Grid がメモリーに保持する必要のあるエントリーの数減らします。ソフトインデックスファイルストアを繰り返し処理するため、Data Grid はインデックスのすべてのエントリーを同時に読み取ります。

B+ ツリーの各エントリーはノードです。デフォルトでは、各ノードのサイズは 4096 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。

セグメンテーション

soft-index ファイルの場所は常にセグメント化されます。



注記

AdvancedStore.purgeExpired() メソッドは **SoftIndexFileStore** に実装されていません。

単一ファイルキャッシュストア



注記

単一ファイルキャッシュストアは非推奨となり、削除される予定です。

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、データ 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。

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、キーが大きく、または多数の値がある可能性のあるユースケースには推奨しません。

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、エントリーに収め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。

セグメンテーション

単一ファイルキャッシュストアは、デフォルトではセグメントごとに個別のインスタンスを持つセグメント化され、これにより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。

6.8.1. ファイルベースのキャッシュストアの設定

ファイルベースのキャッシュストアを Data Grid に追加し、ホストファイルシステムでデータを永続化します。

前提条件

- 組み込みキャッシュを設定する場合は、グローバルの状態を有効にし、グローバルの永続的な場所を設定します。

手順

1. **persistence** 要素をキャッシュ設定に追加します。
2. オプションで、データがメモリーからエビクトされる場合にのみ、ファイルベースのキャッシュストアに書き込む **passivation** 属性の値として **true** を指定します。
3. **file-store** 要素を含め、属性を適宜設定します。
4. **False** を **shared** 属性の値として指定します。
ファイルベースのキャッシュストアは、常に各 Data Grid インスタンスに固有のものである必要があります。クラスター全体で同じ永続を使用する場合は、JDBC 文字列ベースのキャッシュストアなどの共有ストレージを設定します。
5. **index** と **data** 要素を設定し、Data Grid がインデックスを作成し、データを格納する場所を指定します。
6. **write-behind** モードでキャッシュストアを設定する場合は、write-behind 要素を含めます。

ファイルベースのキャッシュストアの設定

XML

```
<distributed-cache>
  <persistence passivation="true">
    <file-store shared="false">
      <data path="data"/>
      <index path="index"/>
      <write-behind modification-queue-size="2048" />
    </file-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "passivation": true,
      "file-store": {
        "shared": false,
        "data": {
          "path": "data"
        },
        "index": {
          "path": "index"
        }
      }
    }
  }
}
```



```

    },
    "write-behind": {
      "modification-queue-size": "2048"
    }
  }
}
}
}

```

YAML

```

distributedCache:
  persistence:
    passivation: "true"
    fileStore:
      shared: "false"
      data:
        path: "data"
      index:
        path: "index"
    writeBehind:
      modificationQueueSize: "2048"

```

ConfigurationBuilder

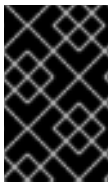
```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().passivation(true)
    .addSoftIndexFileStore()
    .shared(false)
    .dataLocation("data")
    .indexLocation("index")
    .modificationQueueSize(2048);

```

6.8.2. 単一ファイルキャッシュストアの設定

必要な場合は、Data Grid を設定して単一のファイルストアを作成することができます。



重要

単一ファイルストアは非推奨になりました。単一のファイルストアと比べ、ソフトインデックスファイルストアを使用すると、パフォーマンスとデータの整合性が向上します。

前提条件

- 組み込みキャッシュを設定する場合は、グローバルの状態を有効にし、グローバルの永続的な場所を設定します。

手順

1. **persistence** 要素をキャッシュ設定に追加します。

2. オプションで、データがメモリーからエビクトされる場合にのみ、ファイルベースのキャッシュストアに書き込む **passivation** 属性の値として **true** を指定します。
3. **single-file-store** 要素を含めます。
4. **False** を **shared** 属性の値として指定します。
5. その他の属性を必要に応じて設定します。
6. **write-behind** 要素を追加して、書き込みではなく、書き込みの背後でキャッシュストアを設定します。

単一ファイルキャッシュストアの設定

XML

```
<distributed-cache>
  <persistence passivation="true">
    <single-file-store shared="false"
      preload="true"
      fetch-state="true"/>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "passivation": true,
      "single-file-store": {
        "shared": false,
        "preload": true,
        "fetch-state": true
      }
    }
  }
}
```

YAML

```
distributedCache:
  persistence:
    passivation: "true"
  singleFileStore:
    shared: "false"
    preload: "true"
    fetchState: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().passivation(true)
```

```

.addStore(SingleFileStoreConfigurationBuilder.class)
  .shared(false)
  .preload(true)
  .fetchPersistentState(true);

```

6.9. JDBC 接続ファクトリー

Data Grid は、データベースへの接続を可能にするさまざまな **ConnectionFactory** 実装を提供します。SQL キャッシュストアと JDBC 文字列ベースのキャッシュストアで JDBC 接続を使用します。

接続プール

接続プールは、スタンドアロンの Data Grid デプロイメントに適していますが、Agroal をベースにしています。

XML

```

<distributed-cache>
  <persistence>
    <connection-pool connection-url="jdbc:h2:mem:infinispan;DB_CLOSE_DELAY=-1"
      username="sa"
      password="changeme"
      driver="org.h2.Driver"/>
  </persistence>
</distributed-cache>

```

JSON

```

{
  "distributed-cache": {
    "persistence": {
      "connection-pool": {
        "connection-url": "jdbc:h2:mem:infinispan_string_based",
        "driver": "org.h2.Driver",
        "username": "sa",
        "password": "changeme"
      }
    }
  }
}

```

YAML

```

distributedCache:
  persistence:
    connectionPool:
      connectionUrl: "jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1"
      driver: org.h2.Driver
      username: sa
      password: changeme

```

ConfigurationBuilder

■

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

管理データソース

データソース接続は、アプリケーションサーバーなどの管理環境に適しています。

XML

```
<distributed-cache>
  <persistence>
    <data-source jndi-url="java:/StringStoreWithManagedConnectionTest/DS" />
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "data-source": {
        "jndi-url": "java:/StringStoreWithManagedConnectionTest/DS"
      }
    }
  }
}
```

YAML

```
distributedCache:
  persistence:
    dataSource:
      jndiUrl: "java:/StringStoreWithManagedConnectionTest/DS"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
    .dataSource()
    .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
```

セキュアな接続

単純な接続ファクトリーは呼び出しごとにデータベース接続を作成し、テスト環境または開発環境での使用を目的としています。

XML

```
<distributed-cache>
```

```

<persistence>
  <simple-connection connection-url="jdbc:h2://localhost"
    username="sa"
    password="changeme"
    driver="org.h2.Driver"/>
</persistence>
</distributed-cache>

```

JSON

```

{
  "distributed-cache": {
    "persistence": {
      "simple-connection": {
        "connection-url": "jdbc:h2://localhost",
        "driver": "org.h2.Driver",
        "username": "sa",
        "password": "changeme"
      }
    }
  }
}

```

YAML

```

distributedCache:
  persistence:
    simpleConnection:
      connectionUrl: "jdbc:h2://localhost"
      driver: org.h2.Driver
      username: sa
      password: changeme

```

ConfigurationBuilder

```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
  .simpleConnection()
  .connectionUrl("jdbc:h2://localhost")
  .driverClass("org.h2.Driver")
  .username("admin")
  .password("changeme");

```

関連情報

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

6.9.1. 管理対象データソースの設定

Data Grid Server 設定の一部として管理対象データソースを作成し、JDBC データベース接続の接続プールとパフォーマンスを最適化します。その後、キャッシュ内の管理対象データソースの JNDI 名を指定して、デプロイメントの JDBC 接続設定を一元化できます。

前提条件

- データベースドライバーを、Data Grid Server インストールの **server/lib** ディレクトリーにコピーします。

手順

1. Data Grid Server 設定を開いて編集します。
2. **data-sources** セクションに新しい **data-source** を追加します。
3. **name** 属性またはフィールドでデータソースを一意に識別します。
4. **jndi-name** 属性またはフィールドを使用してデータソースの JNDI 名を指定します。

ヒント

JNDI 名を使用して、JDBC キャッシュストア設定でデータソースを指定します。

5. **true** を **statistics** 属性またはフィールドの値に設定し、**/metrics** エンドポイント経由でデータソースの統計を有効にします。
6. **connection-factory** セクションのデータソースへの接続方法を定義する JDBC ドライバーの詳細を提供します。
 - a. **driver** 属性またはフィールドを使用して、データベースドライバーの名前を指定します。
 - b. **url** 属性またはフィールドを使用して、JDBC 接続 URL を指定します。
 - c. **username** および **password** 属性またはフィールドを使用して、認証情報を指定します。
 - d. 必要に応じて他の設定を指定します。
7. Data Grid Server ノードが接続をプールして再利用する方法を、**connection-pool** セクションの接続プール調整プロパティで定義します。
8. 変更を設定に保存します。

検証

以下のように、Data Grid コマンドラインインターフェイス (CLI) を使用して、データソース接続をテストします。

1. CLI セッションを開始します。

```
bin/cli.sh
```

2. すべてのデータソースを一覧表示し、作成したデータソースが利用できることを確認します。

```
server datasource ls
```

3. データソース接続をテストします。

```
server datasource test my-datasource
```

管理対象データソースの設定

XML

```
<server xmlns="urn:infinispan:server:13.0">
  <data-sources>
    <!-- Defines a unique name for the datasource and JNDI name that you
         reference in JDBC cache store configuration.
         Enables statistics for the datasource, if required. -->
    <data-source name="ds"
      jndi-name="jdbc/postgres"
      statistics="true">
      <!-- Specifies the JDBC driver that creates connections. -->
      <connection-factory driver="org.postgresql.Driver"
        url="jdbc:postgresql://localhost:5432/postgres"
        username="postgres"
        password="changeme">
        <!-- Sets optional JDBC driver-specific connection properties. -->
        <connection-property name="name">value</connection-property>
      </connection-factory>
      <!-- Defines connection pool tuning properties. -->
      <connection-pool initial-size="1"
        max-size="10"
        min-size="3"
        background-validation="1000"
        idle-removal="1"
        blocking-timeout="1000"
        leak-detection="10000"/>
    </data-source>
  </data-sources>
</server>
```

JSON

```
{
  "server": {
    "data-sources": [{
      "name": "ds",
      "jndi-name": "jdbc/postgres",
      "statistics": true,
      "connection-factory": {
        "driver": "org.postgresql.Driver",
        "url": "jdbc:postgresql://localhost:5432/postgres",
        "username": "postgres",
        "password": "changeme",
        "connection-properties": {
          "name": "value"
        }
      },
      "connection-pool": {
```

```

        "initial-size": 1,
        "max-size": 10,
        "min-size": 3,
        "background-validation": 1000,
        "idle-removal": 1,
        "blocking-timeout": 1000,
        "leak-detection": 10000
    }
}
}
}

```

YAML

```

server:
  dataSources:
    - name: ds
      jndiName: 'jdbc/postgres'
      statistics: true
      connectionFactory:
        driver: "org.postgresql.Driver"
        url: "jdbc:postgresql://localhost:5432/postgres"
        username: "postgres"
        password: "changeme"
        connectionProperties:
          name: value
      connectionPool:
        initialSize: 1
        maxSize: 10
        minSize: 3
        backgroundValidation: 1000
        idleRemoval: 1
        blockingTimeout: 1000
        leakDetection: 10000

```

6.9.1.1. JNDI 名を使用したキャッシュの設定

管理対象データソースを Data Grid Server に追加するとき、JNDI 名を JDBC ベースのキャッシュストア設定に追加できます。

前提条件

- 管理対象データソースを使用した Data Grid Server の設定

手順

1. キャッシュ設定を開いて編集します。
2. **data-source** 要素またはフィールドを JDBC ベースのキャッシュストア設定に追加します。
3. 管理対象データソースの JNDI 名を **jndi-url** 属性の値として指定します。
4. JDBC ベースのキャッシュストアを適宜設定します。

5. 変更を設定に保存します。

キャッシュ設定の JNDI 名

XML

```
<distributed-cache>
  <persistence>
    <jdbc:string-keyed-jdbc-store>
      <!-- Specifies the JNDI name of a managed datasource on Data Grid Server. -->
      <jdbc:data-source jndi-url="jdbc/postgres"/>
      <jdbc:string-keyed-table drop-on-exit="true" create-on-start="true" prefix="TBL">
        <jdbc:id-column name="ID" type="VARCHAR(255)"/>
        <jdbc:data-column name="DATA" type="BYTEA"/>
        <jdbc:timestamp-column name="TS" type="BIGINT"/>
        <jdbc:segment-column name="S" type="INT"/>
      </jdbc:string-keyed-table>
    </jdbc:string-keyed-jdbc-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "string-keyed-jdbc-store": {
        "data-source": {
          "jndi-url": "jdbc/postgres"
        },
        "string-keyed-table": {
          "prefix": "TBL",
          "drop-on-exit": true,
          "create-on-start": true,
          "id-column": {
            "name": "ID",
            "type": "VARCHAR(255)"
          },
          "data-column": {
            "name": "DATA",
            "type": "BYTEA"
          },
          "timestamp-column": {
            "name": "TS",
            "type": "BIGINT"
          },
          "segment-column": {
            "name": "S",
            "type": "INT"
          }
        }
      }
    }
  }
}
```

```

    }
  }
}

```

YAML

```

distributedCache:
  persistence:
    stringKeyedJdbcStore:
      dataSource:
        jndi-url: "jdbc/postgres"
      stringKeyedTable:
        prefix: "TBL"
        dropOnExit: true
        createOnStart: true
      idColumn:
        name: "ID"
        type: "VARCHAR(255)"
      dataColumn:
        name: "DATA"
        type: "BYTEA"
      timestampColumn:
        name: "TS"
        type: "BIGINT"
      segmentColumn:
        name: "S"
        type: "INT"

```

6.9.1.2. 接続プールのチューニングプロパティ

Data Grid Server 設定で、管理対象データソースの JDBC 接続プールを調整できます。

プロパティ	説明
initial-size	プールが保持する最初の接続数。
max-size	プールの最大接続数。
min-size	プールが保持する必要のある接続の最小数。
blocking-timeout	例外が発生する前に、接続を待機している間にブロックする最大時間 (ミリ秒単位)。新しい接続の作成に非常に長い時間がかかる場合は、これによって例外が出力されることはありません。デフォルトは 0 です。これは、呼び出しが無期限に待機することを意味します。
background-validation	バックグラウンド検証の実行の間隔 (ミリ秒単位)。期間 0 は、この機能が無効化されていることを意味します。

プロパティ	説明
validate-on-acquisition	ミリ秒単位で指定された、この時間より長いアイドル状態の接続は、取得される前に検証されます (フォアグラウンド検証)。期間 0 は、この機能が無効化されていることを意味します。
idle-removal	削除される前の接続がアイドル状態でなくてはならない時間 (分単位)。
leak-detection	リーク警告の前に接続を保持しなければならない時間 (ミリ秒単位)。

6.9.2. Agroal プロパティを使用した JDBC 接続プールの設定

プロパティファイルを使用して、JDBC 文字列ベースのキャッシュストアにプールされた接続ファクトリーを設定できます。

手順

1. 以下の例のように、**org.infinispan.agroal.*** プロパティで JDBC 接続プール設定を指定します。

```
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
```

2. **properties-file** 属性または **PooledConnectionFactoryConfiguration.propertyFile()** メソッドでプロパティファイルを使用するように Data Grid を設定します。

XML

```
<connection-pool properties-file="path/to/agroal.properties"/>
```

JSON

```
"persistence": {
  "connection-pool": {
    "properties-file": "path/to/agroal.properties"
  }
}
```

YAML

```
persistence:
  connectionPool:
    propertiesFile: path/to/agroal.properties
```

ConfigurationBuilder

```
.connectionPool().propertyFile("path/to/agroal.properties")
```

関連情報

- [Agroal](#)

6.10. SQL キャッシュストア

SQL キャッシュストアを使用すると、既存のデータベーステーブルから Data Grid キャッシュを読み込むことができます。Data Grid は、2 種類の SQL キャッシュストアを提供します。

テーブル

Data Grid は、1つのデータベーステーブルからエントリーを読み込みます。

クエリー

Data Grid は SQL クエリーを使用して、単一または複数のデータベーステーブルからエントリーを読み込み (これらのテーブルのサブ列からの読み込みを含む)、挿入、更新、および削除操作を実行します。

ヒント

コードチュートリアルにアクセスして、SQL キャッシュストアの動作を試します。 [Persistence code tutorial with remote caches](#) を参照してください。

SQL テーブルとクエリースタアの両方は以下のようになります。

- 永続ストレージに対する読み取りおよび書き込み操作を許可します。
- 読み取り専用で、キャッシュローダーとして機能します。
- 単一のデータベース列または複数のデータベース列の複合に対応するキーと値をサポートします。
複合キーと値の場合は、キーと値を記述する Protobuf スキーマ (**.proto** ファイル) で Data Grid を指定する必要があります。Data Grid Server を使用すると、`schema` コマンドを使用して、Data Grid Console または Command Line Interface (CLI) から **schema** を追加できます。



注記

SQL キャッシュストアは有効期限またはセグメンテーションをサポートしません。

関連資料

- [DatabaseType Enum サポートされているデータベースのダイレクトのリスト](#)
- [Data Grid SQL store configuration reference](#)

6.10.1. キーおよび値のデータ型

Data Grid は、適切なデータ型を使用して、SQL キャッシュストアを介してデータベーステーブルの列からキーと値を読み込みます。以下の **CREATE** ステートメントは、**isbn** と **title** の 2 つの列が含まれる **books** という名前のテーブルを追加します。

2 列のデータベーステーブル

```
CREATE TABLE books (
  isbn NUMBER(13),
  title varchar(120)
  PRIMARY KEY(isbn)
);
```

SQL キャッシュストアでこのテーブルを使用すると、Data Grid は **isbn** 列をキーとして、**title** 列を値として使用してキャッシュに追加します。

関連情報

- [Data Grid SQL store configuration reference](#)

6.10.1.1. 複合キーと値

複合プライマリーキーまたは複合値を含むデータベーステーブルで SQL ストアを使用できます。

複合キーまたは値を使用するには、データ型を記述する Protobuf スキーマで Data Grid を指定する必要があります。また、SQL ストアに **schema** 設定を追加し、キーと値のメッセージ名を指定する必要があります。

ヒント

Data Grid は、ProtoStream プロセッサーで Protobuf スキーマを生成することを推奨します。次に、Data Grid コンソール、CLI、または REST API を使用してリモートキャッシュの Protobuf スキーマをアップロードできます。

複合値

以下のデータベーステーブルは、**title** および **author** 列の複合値を保持します。

```
CREATE TABLE books (
  isbn NUMBER(13),
  title varchar(120),
  author varchar(80)
  PRIMARY KEY(isbn)
);
```

Data Grid は、**isbn** 列をキーとして使用してキャッシュにエントリーを追加します。値の場合、Data Grid には **title** 列と **author** 列をマッピングする Protobuf スキーマが必要です。

```
package library;

message books_value {
  optional string title = 1;
  optional string author = 2;
}
```

複合キーと値

以下のデータベーステーブルは複合プライマリーキーと複合値を保持し、それぞれに 2 列があります。

```
CREATE TABLE books (
  isbn NUMBER(13),
  reprint INT,
  title varchar(120),
  author varchar(80)
  PRIMARY KEY(isbn, reprint)
);
```

キーと値の両方で、Data Grid には列をキーと値にマッピングする Protobuf スキーマが必要です。

```
package library;

message books_key {
  required string isbn = 1;
  required int32 reprint = 2;
}

message books_value {
  optional string title = 1;
  optional string author = 2;
}
```

関連情報

- [Cache encoding and marshalling: Generate Protobuf schema and register them with Data Grid](#)
- [Data Grid SQL store configuration reference](#)

6.10.1.2. 埋め込みキー

以下の例のように、Protobuf スキーマは値の中にキーを含めることができます。

組み込みキーを使用した Protobuf スキーマ

```
package library;

message books_key {
  required string isbn = 1;
  required int32 reprint = 2;
}
```

```

message books_value {
  required string isbn = 1;
  required string reprint = 2;
  optional string title = 3;
  optional string author = 4;
}

```

埋め込みキーを使用するには、SQL ストア設定に **embedded-key="true"** 属性または **embeddedKey(true)** メソッドを含める必要があります。

6.10.1.3. SQL から Protobuf タイプへ

以下の表に、SQL データ型のデフォルトを Protobuf データ型にマッピングしています。

SQL 型	Protobuf タイプ
int4	int32
int8	int64
float4	float
float8	double
numeric	double
bool	bool
char	string
varchar	string
text、tinytext、mediumtext、longtext	string
bytea、tinyblob、Blob、mediumblob、longblob	bytes

関連情報

- [キャッシュエンコーディングおよびマーシャリング](#)

6.10.2. データベーステーブルからの Data Grid キャッシュの読み込み

Data Grid にデータベーステーブルからデータを読み込ませる場合は、SQL テーブルキャッシュストアを設定に追加します。データベースに接続すると、Data Grid はテーブルからメタデータを使用して列名とデータタイプを検出します。また、Data Grid は、データベースのどの列がプライマリーキーの一部であるかを自動的に決定します。

前提条件

- JDBC 接続の詳細を把握している。
JDBC 接続ファクトリーを直接キャッシュ設定に追加できます。
実稼働環境でのリモートキャッシュでは、管理対象データソースを Data Grid Server 設定に追加し、キャッシュ設定で JNDI 名を指定する必要があります。
- 複合キーまたは複合値の Protobuf スキーマを生成し、スキーマを Data Grid に登録します。

ヒント

Data Grid は、ProtoStream プロセッサで Protobuf スキーマを生成することを推奨します。
リモートキャッシュでは、Data Grid コンソール、CLI、または REST API を使用してスキーマを追加して、スキーマを登録できます。

手順

1. データベースドライバーを Data Grid デプロイメントに追加します。
 - リモートキャッシュ: データベースドライバーを、Data Grid Server インストールの **server/lib** ディレクトリにコピーします。
 - 組み込みキャッシュ: **infinispan-cachestore-sql** 依存関係を **pom** ファイルに追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cachestore-sql</artifactId>
</dependency>
```

2. Data Grid 設定を開いて編集します。
3. SQL テーブルキャッシュストアを追加します。

宣言的 (Declarative)

```
table-jdbc-store xmlns="urn:infinispan:config:store:sql:13.0"
```

プログラマティック

```
persistence().addStore(TableJdbcStoreConfigurationBuilder.class)
```

4. データベースダイアレクトを **dialect=""** または **dialect()** のいずれかで指定します
(例: **dialect="H2"** または **dialect="postgres"**)。
5. 以下のように、必要なプロパティで SQL キャッシュストアを設定します。
 - クラスター全体で同じキャッシュストアを使用するには、**shared="true"** または **shared(true)** を設定します。
 - 読み取り専用のキャッシュストアを作成するには、**read-only="true"** または **.ignoreModifications(true)** を設定します。
6. **table-name="<database_table_name>"** または **table.name("<database_table_name>")** を使用して、キャッシュを読み込むデータベーステーブルに名前を付けます。

7. **schema** 要素または **.schemaJdbcConfigurationBuilder()** メソッドを追加し、複合キーまたは値の Protobuf スキーマ設定を追加します。
 - a. **package** 属性または **package()** メソッドを使用してパッケージ名を指定します。
 - b. **message-name** 属性または **messageName()** メソッドを使用して複合値を指定します。
 - c. **key-message-name** 属性または **keyMessageName()** メソッドを使用して複合キーを指定します。
 - d. スキーマで値内にキーが含まれている場合は、**embedded-key** 属性または **embeddedKey()** メソッドに **true** の値を設定します。
8. 変更を設定に保存します。

SQL テーブルストアの設定

以下の例では、Protobuf スキーマで定義された複合値を使用して、books という名前のデータベーステーブルから分散キャッシュを読み込みます。

XML

```
<distributed-cache>
  <persistence>
    <table-jdbc-store xmlns="urn:infinispan:config:store:sql:13.0"
      dialect="H2"
      shared="true"
      table-name="books">
      <schema message-name="books_value"
        package="library"/>
    </table-jdbc-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "table-jdbc-store": {
        "dialect": "H2",
        "shared": "true",
        "table-name": "books",
        "schema": {
          "message-name": "books_value",
          "package": "library"
        }
      }
    }
  }
}
```

YAML

```
distributedCache:
```

```

persistence:
  tableJdbcStore:
    dialect: "H2"
    shared: "true"
    tableName: "books"
    schema:
      messageName: "books_value"
      package: "library"

```

ConfigurationBuilder

```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(TableJdbcStoreConfigurationBuilder.class)
    .dialect(DatabaseType.H2)
    .shared("true")
    .tableName("books")
    .schemaJdbcConfigurationBuilder()
    .messageName("books_value")
    .packageName("library");

```

関連情報

- [Cache encoding and marshalling: Generate Protobuf schema and register them with Data Grid](#)
- [Persistence code tutorial with remote caches](#)
- [JDBC 接続ファクトリー](#)
- [DatabaseType Enum サポートされているデータベースのダイアレクトのリスト](#)
- [Data Grid SQL store configuration reference](#)

6.10.3. SQL クエリーを使用したデータのロードおよび操作の実行

SQL クエリーキャッシュストアを使用すると、データベーステーブルのサブ列からなど、複数のデータベーステーブルからキャッシュを読み込み、挿入、更新、および削除操作を実行することができます。

前提条件

- JDBC 接続の詳細を把握している。
JDBC 接続ファクトリーを直接キャッシュ設定に追加できます。
実稼働環境でのリモートキャッシュでは、管理対象データソースを Data Grid Server 設定に追加し、キャッシュ設定で JNDI 名を指定する必要があります。
- 複合キーまたは複合値の Protobuf スキーマを生成し、スキーマを Data Grid に登録します。

ヒント

Data Grid は、ProtoStream プロセッサで Protobuf スキーマを生成することを推奨します。リモートキャッシュでは、Data Grid コンソール、CLI、または REST API を使用してスキーマを追加して、スキーマを登録できます。

手順

- データベースドライバーを Data Grid デプロイメントに追加します。
 - リモートキャッシュ: データベースドライバーを、Data Grid Server インストールの **server/lib** ディレクトリーにコピーします。
 - 組み込みキャッシュ: **infinispan-cachestore-sql** 依存関係を **pom** ファイルに追加し、データベースドライバーがアプリケーションクラスパス上にあることを確認します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cachestore-sql</artifactId>
</dependency>
```

- Data Grid 設定を開いて編集します。
- SQL クエリーキャッシュストアを追加します。

宣言的 (Declarative)

```
query-jdbc-store xmlns="urn:infinispan:config:store:jdbc:13.0"
```

プログラマティック

```
persistence().addStore(QueriesJdbcStoreConfigurationBuilder.class)
```

- データベースダイアレクトを **dialect=""** または **dialect()** のいずれかで指定します (例: **dialect="H2"** または **dialect="postgres"**)。
- 以下のように、必要なプロパティーで SQL キャッシュストアを設定します。
 - クラスター全体で同じキャッシュストアを使用するには、**shared="true"** または **shared(true)** を設定します。
 - 読み取り専用のキャッシュストアを作成するには、**read-only="true"** または **.ignoreModifications(true)** を設定します。
- データでキャッシュを読み込み、**queries** 要素または **queries()** メソッドでデータベーステーブルを変更する SQL クエリーステートメントを定義します。

クエリーステートメント	説明
SELECT	単一のエントリーをキャッシュに読み込みます。ワイルドカードを使用できますが、キーのパラメーターを指定する必要があります。ラベル付きの式を使用できます。
SELECT ALL	複数のエントリーをキャッシュに読み込みます。返された列の数がキーと値の列数と一致する場合は、*ワイルドカードを使用できます。ラベル付きの式を使用できます。
SIZE	キャッシュ内のエントリー数をカウントします。

クエリーステートメント	説明
DELETE	キャッシュからエントリーを1つ削除します。
DELETE ALL	キャッシュからすべてのエントリーを削除します。
UPSERT	キャッシュのエントリーを修正します。

注記

DELETE、**DELETE ALL**、および **UPSERT** ステートメントは読み取り専用キャッシュストアには適用されませんが、キャッシュストアが変更を許可する場合は必要です。

DELETE ステートメントのパラメーターは、**SELECT** ステートメントのパラメーターに完全に一致する必要があります。

UPSERT ステートメントの変数には、**SELECT** および **SELECT ALL** ステートメントが返すものと同じ数の一意に名前が付けられた変数が必要です。たとえば、**SELECT** が **foo** と **bar** を返す場合、このステートメントは変数として **:foo** および **:bar** のみを取る必要があります。ただし、ステートメントに同じ名前の変数を複数回適用することができます。

SQL クエリーには、**JOIN**、**ON** およびデータベースがサポートするその他の句を含めることができます。

7. **schema** 要素または **.schemaJdbcConfigurationBuilder()** メソッドを追加し、複合キーまたは値の Protobuf スキーマ設定を追加します。
 - a. **package** 属性または **package()** メソッドを使用してパッケージ名を指定します。
 - b. **message-name** 属性または **messageName()** メソッドを使用して複合値を指定します。
 - c. **key-message-name** 属性または **keyMessageName()** メソッドを使用して複合キーを指定します。
 - d. スキーマで値内にキーが含まれている場合は、**embedded-key** 属性または **embeddedKey()** メソッドに **true** の値を設定します。
8. 変更を設定に保存します。

関連情報

- [Cache encoding and marshalling: Generate Protobuf schema and register them with Data Grid](#)
- [Persistence code tutorial with remote caches](#)
- [JDBC 接続ファクトリー](#)
- [DatabaseType Enum サポートされているデータベースのダイアレクトのリスト](#)
- [Data Grid SQL store configuration reference](#)

6.10.3.1. SQL クエリーストアの設定

このセクションでは、person と address の2つのデータベーステーブルのデータを含む分散キャッシュを読み込む SQL クエリーキャッシュストアの設定例を説明します。

SQL ステートメント

person および address テーブルの SQL データ定義言語 (DDL) ステートメントは以下のとおりです。

person テーブルの SQL ステートメント

```
CREATE TABLE Person (
  name VARCHAR(255) NOT NULL,
  picture VARBINARY(255),
  sex VARCHAR(255),
  birthdate TIMESTAMP,
  accepted_tos BOOLEAN,
  notused VARCHAR(255),
  PRIMARY KEY (name)
);
```

address テーブルの SQL ステートメント

```
CREATE TABLE Address (
  name VARCHAR(255) NOT NULL,
  street VARCHAR(255),
  city VARCHAR(255),
  zip INT,
  PRIMARY KEY (name)
);
```

Protobuf スキーマ

person および address テーブルの Protobuf スキーマは以下のとおりです。

person テーブルの Protobuf スキーマ

```
package com.example

enum Sex {
  FEMALE = 1;
  MALE = 2;
}

message Person {
  optional string name = 1;
  optional Address address = 2;
  optional bytes picture = 3;
  optional Sex sex = 4;
  optional fixed64 birthDate = 5 [default = 0];
  optional bool accepted_tos = 6 [default = false];
}
```

address テーブルの Protobuf スキーマ

■

```
package com.example

message Address {
    optional string street = 1;
    optional string city = 2 [default = "San Jose"];
    optional int32 zip = 3 [default = 0];
}
```

キャッシュ設定

以下の例では、**JOIN** 句を含む SQL クエリーを使用して、person テーブルおよび address テーブルから分散キャッシュを読み込みます。

XML

```
<distributed-cache>
  <persistence>
    <query-jdbc-store xmlns="urn:infinispan:config:store:jdbc:13.0"
      dialect="POSTGRES"
      shared="true">
      <queries key-columns="name">
        <select-single>SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos, t2.street,
t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name WHERE t1.name =
:name</select-single>
        <select-all>SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos, t2.street, t2.city,
t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name</select-all>
        <delete-single>DELETE FROM Person t1 WHERE t1.name = :name; DELETE FROM Address
t2 where t2.name = :name</delete-single>
        <delete-all>DELETE FROM Person; DELETE FROM Address</delete-all>
        <upsert>INSERT INTO Person (name, picture, sex, birthdate, accepted_tos) VALUES (:name,
:picture, :sex, :birthdate, :accepted_tos); INSERT INTO Address(name, street, city, zip) VALUES
(:name, :street, :city, :zip)</upsert>
        <size>SELECT COUNT(*) FROM Person</size>
      </queries>
      <schema message-name="Person"
        package="com.example"
        embedded-key="true"/>
    </query-jdbc-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "query-jdbc-store": {
        "dialect": "POSTGRES",
        "shared": "true",
        "key-columns": "name",
        "queries": {
          "select-single": "SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos, t2.street,
t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name WHERE t1.name = :name",
          "select-all": "SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos, t2.street, t2.city,
t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name",

```

```

    "delete-single": "DELETE FROM Person t1 WHERE t1.name = :name; DELETE FROM
Address t2 where t2.name = :name",
    "delete-all": "DELETE FROM Person; DELETE FROM Address",
    "upsert": "INSERT INTO Person (name, picture, sex, birthdate, accepted_tos) VALUES
(:name, :picture, :sex, :birthdate, :accepted_tos); INSERT INTO Address(name, street, city, zip)
VALUES (:name, :street, :city, :zip)",
    "size": "SELECT COUNT(*) FROM Person"
  },
  "schema": {
    "message-name": "Person",
    "package": "com.example",
    "embedded-key": "true"
  }
}
}
}
}
}

```

YAML

```

distributedCache:
  persistence:
    queryJdbcStore:
      dialect: "POSTGRES"
      shared: "true"
      keyColumns: "name"
      queries:
        selectSingle: "SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos, t2.street,
t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name WHERE t1.name = :name"
        selectAll: "SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos, t2.street, t2.city,
t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name"
        deleteSingle: "DELETE FROM Person t1 WHERE t1.name = :name; DELETE FROM Address t2
where t2.name = :name"
        deleteAll: "DELETE FROM Person; DELETE FROM Address"
        upsert: "INSERT INTO Person (name, picture, sex, birthdate, accepted_tos) VALUES (:name,
:picture, :sex, :birthdate, :accepted_tos); INSERT INTO Address(name, street, city, zip) VALUES
(:name, :street, :city, :zip)"
        size: "SELECT COUNT(*) FROM Person"
      schema:
        messageName: "Person"
        package: "com.example"
        embeddedKey: "true"

```

ConfigurationBuilder

```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(QueriesJdbcStoreConfigurationBuilder.class)
    .dialect(DatabaseType.POSTGRES)
    .shared("true")
    .keyColumns("name")
    .queriesJdbcConfigurationBuilder()
        .select("SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos, t2.street, t2.city,
t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name WHERE t1.name = :name")
        .selectAll("SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos, t2.street, t2.city,

```

```
t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name")
    .delete("DELETE FROM Person t1 WHERE t1.name = :name; DELETE FROM Address t2
where t2.name = :name")
    .deleteAll("DELETE FROM Person; DELETE FROM Address")
    .upsert("INSERT INTO Person (name, picture, sex, birthdate, accepted_tos) VALUES (:name,
:picture, :sex, :birthdate, :accepted_tos); INSERT INTO Address(name, street, city, zip) VALUES
(:name, :street, :city, :zip)")
    .size("SELECT COUNT(*) FROM Person")
    .schemaJdbcConfigurationBuilder()
    .messageName("Person")
    .packageName("com.example")
    .embeddedKey(true);
```

関連情報

- [Data Grid SQL store configuration reference](#)

6.10.4. SQL キャッシュストアに関するトラブルシューティング

SQL キャッシュストアに関する一般的な問題およびエラーと、そのトラブルシューティング方法を確認してください。

ISPNO08064: No primary keys found for table <table_name>, check case sensitivity

Data Grid は、以下の場合にこのメッセージをログに記録します。

- データベーステーブルが存在しない。
- データベーステーブル名は大文字と小文字が区別され、データベースプロバイダーに応じて、すべての小文字またはすべての大文字のいずれかである必要があります。
- データベーステーブルにプライマリーキーが定義されていない。

この問題を解決するには、以下を行う必要があります。

1. SQL キャッシュストア設定を確認し、既存のテーブルの名前を指定するようにしてください。
2. データベーステーブル名が大文字/小文字の要件に準拠することを確認します。
3. データベーステーブルに、適切な行を一意に識別するプライマリーキーがあることを確認します。

6.11. 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。

JDBC 文字列ベースのキャッシュストア:

- 各エントリーをテーブルに独自の行に保存し、同時ロードのスループットを向上させます。
- **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プする単純な 1 対 1 のマッピングを使用します。
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。

関連情報

- [DatabaseType Enum サポートされているデータベースのダイレクトのリスト](#)

6.11.1. JDBC 文字列ベースのキャッシュストアの設定

データベースに接続できる JDBC 文字列ベースのキャッシュストアで Data Grid キャッシュを設定します。

前提条件

- リモートキャッシュ: データベースドライバーを、Data Grid Server インストールの **server/lib** ディレクトリーにコピーします。
- 組み込みキャッシュ: **infinispan-cachestore-jdbc** 依存関係を **pom** ファイルに追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cachestore-jdbc</artifactId>
</dependency>
```

手順

1. 以下のいずれかの方法で JDBC 文字列ベースのキャッシュストア設定を作成します。
 - 宣言的に、**persistence** 要素またはフィールドを追加してから、以下のスキーマ名前空間で **string-keyed-jdbc-store** を追加します。

```
xmlns="urn:infinispan:config:store:jdbc:13.0"
```

- プログラムで以下のメソッドを **ConfigurationBuilder** に追加します。

```
persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
```

2. **dialect** 属性または **dialect()** メソッドのいずれかを使用してデータベースのダイレクトを指定します。
3. JDBC 文字列ベースのキャッシュストアのプロパティを随時設定します。

たとえば、キャッシュストアが共有属性または **shared** メソッドまたは **shared()** メソッドのいずれかで複数のキャッシュインスタンスと共有されるかどうかを指定します。

4. Data Grid がデータベースに接続できるように、JDBC 接続ファクトリーを追加します。
5. キャッシュエントリを保存するデータベーステーブルを追加します。

JDBC 文字列ベースのキャッシュストアの設定

XML

```
<distributed-cache>
  <persistence>
    <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:13.0"
      dialect="H2">
      <connection-pool connection-url="jdbc:h2:mem:infinispan"
        username="sa"
        password="changeme"
        driver="org.h2.Driver"/>
      <string-keyed-table 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>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "string-keyed-jdbc-store": {
        "dialect": "H2",
        "string-keyed-table": {
          "prefix": "ISPN_STRING_TABLE",
          "create-on-start": true,
          "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"
    }
  },
  "connection-pool": {
    "connection-url": "jdbc:h2:mem:infinispan",
    "driver": "org.h2.Driver",
    "username": "sa",
    "password": "changeme"
  }
}
}
}
}

```

YAML

```

distributedCache:
  persistence:
    stringKeyedJdbcStore:
      dialect: "H2"
    stringKeyedTable:
      prefix: "ISPN_STRING_TABLE"
      createOnStart: true
      idColumn:
        name: "ID_COLUMN"
        type: "VARCHAR(255)"
      dataColumn:
        name: "DATA_COLUMN"
        type: "BINARY"
      timestampColumn:
        name: "TIMESTAMP_COLUMN"
        type: "BIGINT"
      segmentColumn:
        name: "SEGMENT_COLUMN"
        type: "INT"
    connectionPool:
      connectionUrl: "jdbc:h2:mem:infinispan"
      driver: "org.h2.Driver"
      username: "sa"
      password: "changeme"

```

ConfigurationBuilder

```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
    .dialect(DatabaseType.H2)
    .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")
.username("sa")
.password("changeme")
.driverClass("org.h2.Driver");

```

関連情報

- [JDBC 接続ファクトリー](#)

6.12. ROCKSDB のキャッシュストア

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 がメモリから失効するエントリーを保持します。

表6.1 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。

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**

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。

RocksDB キャッシュストアの設定

XML

```
<local-cache>
  <persistence>
    <rocksdb-store xmlns="urn:infinispan:config:store:rocksdb:13.0"
      path="rocksdb/data">
      <expiration path="rocksdb/expired"/>
    </rocksdb-store>
  </persistence>
</local-cache>
```

JSON

```
{
  "local-cache": {
    "persistence": {
      "rocksdb-store": {
        "path": "rocksdb/data",
```

```

    "expiration": {
      "path": "rocksdb/expired"
    }
  }
}
}
}
}

```

YAML

```

localCache:
  persistence:
    rocksdbStore:
      path: "rocksdb/data"
      expiration:
        path: "rocksdb/expired"

```

ConfigurationBuilder

```

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(...));

```

プロパティを持つ ConfigurationBuilder

```

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();

```

参照資料

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

6.13. 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 クラスターにデータを保存します。



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。

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ーを公開できるため、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。

リモートキャッシュストアの設定

XML

```
<distributed-cache>
  <persistence>
    <remote-store xmlns="urn:infinispan:config:store:remote:13.0"
      cache="mycache"
      raw-values="true">
      <remote-server host="one"
        port="12111" />
      <remote-server host="two" />
      <connection-pool max-active="10"
        exhausted-action="CREATE_NEW" />
    </remote-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
```

```

"remote-store": {
  "cache": "mycache",
  "raw-values": "true",
  "remote-server": [
    {
      "host": "one",
      "port": "12111"
    },
    {
      "host": "two"
    }
  ],
  "connection-pool": {
    "max-active": "10",
    "exhausted-action": "CREATE_NEW"
  }
}
}
}

```

YAML

```

distributedCache:
  remoteStore:
    cache: "mycache"
    rawValues: "true"
    remoteServer:
      - host: "one"
        port: "12111"
      - host: "two"
  connectionPool:
    maxActive: "10"
    exhaustedAction: "CREATE_NEW"

```

ConfigurationBuilder

```

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();

```

参照資料

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

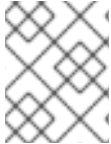
6.14. JPA キャッシュストア

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。

JPA キャッシュストアを使用するには、以下の点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。
- JPA のキャッシュストアは、セグメント化をサポートしていません。



注記

Data Grid キャッシュを組み込んだ JPA キャッシュストアのみを使用する必要があります。

JPA キャッシュストアの設定

XML

```
<local-cache name="vehicleCache">
  <persistence passivation="false">
    <jpa-store xmlns="urn:infinispan:config:store:jpa:13.0"
      persistence-unit="org.infinispan.persistence.jpa.configurationTest"
      entity-class="org.infinispan.persistence.jpa.entity.Vehicle">
    />
  </persistence>
</local-cache>
```

ConfigurationBuilder

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

設定パラメーター

宣言型	プログラマティック	詳細
persistence-unit	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。
entity-class	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。

関連資料

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

6.14.1. JPA キャッシュストアの例

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。

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ー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("raysang", 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](#)

6.15. 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。

**警告**

ClusterLoader は非推奨であり、将来のバージョンで削除される予定です。

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。

クラスターキャッシュブートローダーの設定を変更します。

XML

```
<distributed-cache>
  <persistence>
    <cluster-loader preload="true" remote-timeout="500"/>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "cluster-loader": {
        "preload": true,
        "remote-timeout": "500"
      }
    }
  }
}
```

YAML

```
distributedCache:
  persistence:
    clusterLoader:
      preload: "true"
      remoteTimeout: "500"
```

ConfigurationBuilder

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

関連情報

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

6.16. カスタムキャッシュストア実装の作成

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。

6.16.1. 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ー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ーを公開します。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ー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。

関連情報

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

6.16.2. キャッシュストアの作成

NonBlockingStore API の実装により、カスタムキャッシュストアを作成することができます。

手順

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す。

6.16.3. カスタムキャッシュストアの設定例

以下の例では、カスタムキャッシュストアの実装で Data Grid を設定する方法を示しています。

XML

```
<distributed-cache>
  <persistence>
    <store class="org.infinispan.persistence.example.MyInMemoryStore" />
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "store": {
        "class": "org.infinispan.persistence.example.MyInMemoryStore"
      }
    }
  }
}
```

YAML

```
distributedCache:
  persistence:
    store:
```

```
class: "org.infinispan.persistence.example.MyInMemoryStore"
```

ConfigurationBuilder

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

6.16.4. カスタムキャッシュストアの導入

作成したキャッシュストア実装を Data Grid Server で使用するには、JAR ファイルで提供する必要があります。

前提条件

- Data Grid Server が実行している場合は停止します。
Data Grid は起動時にのみ JAR ファイルを読み込みます。

手順

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 Server インストールの **server/lib** ディレクトリに追加します。

6.17. キャッシュストア間のデータの移行

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。

6.17.1. キャッシュストアマイグレーター

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TRIES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 RocksDB 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。

6.17.2. キャッシュストアマイグレーターの取得

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>

```

6.17.3. キャッシュストアマイグレーターの設定

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
source.version=<version>

```

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

```

6.17.3.1. キャッシュストアマイグレーターの設定プロパティ

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6.2 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6.3 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	<p>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。</p>	.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6.4 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ジャ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 8.0.x の場合は 10</p> <p>* Data Grid 8.1.x の場合は 11</p> <p>* Data Grid 8.2.x の場合は 12</p> <p>* Data Grid 8.3.x の場合は 13</p>	ソースストアにのみ必要です。
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

プロパティ	説明	必須/オプション
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
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6.5 RocksDB プロパティ

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6.6 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6.7 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
```

6.17.4. Data Grid 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7章 ネットワークパーティションを処理するための DATA GRID 設定

Data Grid クラスターは、ノードのサブセットが相互に分離されるネットワークパーティションに分割できます。この状態により、クラスターキャッシュの可用性や一貫性が失われます。Data Grid はクラッシュしたノードを自動的に検出し、競合を解決してキャッシュを1つにマージします。

7.1. クラスターおよびネットワークパーティションの分割

ネットワークパーティションは、ネットワークルーターがクラッシュした場合など、稼働中の環境でのエラー状態の結果です。クラスターがパーティションに分割されると、ノードはそのパーティションのノードのみが含まれる JGroups クラスタービューを作成します。この状態は、1つのパーティションのノードが、他のパーティションのノードとは独立して動作できることを意味します。

分割の検出

ネットワークパーティションを自動的に検出するために、Data Grid はデフォルトの JGroups スタックで **FD_ALL** プロトコルを使用して、ノードが突然クラスターから離れるタイミングを判断します。



注記

Data Grid は、ノードが突然離れる原因を検知できません。これは、ネットワーク障害の発生時だけでなく、ガベージコレクション (GC) が JVM を一時停止した場合など、その他の理由で発生する可能性があります。

Data Grid は、以下の時間が経過すると (ミリ秒単位)、ノードがクラッシュしたことを疑います。

```
FD_ALL.timeout + FD_ALL.interval + VERIFY_SUSPECT.timeout +
GMS.view_ack_collection_timeout
```

クラスターがネットワークパーティションに分割されていることを検出すると、Data Grid はキャッシュ操作処理のストラテジーを使用します。アプリケーションの要件に応じて Data Grid は以下を行うことができます。

- 可用性のために読み取りおよび書き込み操作を許可する
- 一貫性を保つために読み取りおよび書き込み操作を拒否する

パーティションのマージ

分割クラスターを修正するため、Data Grid はパーティションを1つにマージします。マージ時に、Data Grid はキャッシュエントリーの値に **.equals()** メソッドを使用して、競合が存在するかどうかを判断します。パーティションで見つかったレプリカ間の競合を解決するために、Data Grid は設定可能なマージポリシーを使用します。

7.1.1. 分割されたクラスター内のデータの一貫性

Data Grid クラスターをパーティションに分割させるネットワークの停止またはエラーにより、処理ストラテジーやマージポリシーに関係なく、データ喪失や一貫性の問題が発生する可能性があります。

分割と検出の間

分割が発生し、Data Grid が分割を検出する前にマイナーパーティションにあるノードで書き込み操作が行われた場合、マージ中に Data Grid がそのマイナーパーティションに状態を転送すると、その値が失われます。

すべてのパーティションが **DEGRADED** モードになっている場合、状態の転送は発生しないためその値は失われませんが、エントリーに一貫性のない値が含まれる可能性があります。分割が発生したときに進行中のトランザクションキャッシュの書き込み操作は、一部のノードでコミットされ他のノードでロールバックされる場合があるので、このケースでも一貫性のない値が生じます。

分割が発生し、Data Grid がそれを検出する間、まだ **DEGRADED** モードになっていないマイナーパーティションのキャッシュからの古い読み取りが生じる可能性があります。

マージ中

Data Grid がパーティションの削除を開始すると、ノードは一連のマージイベントでクラスターに再接続されます。このマージプロセスを完了する前に、一部のノードではトランザクションキャッシュでの書き込み操作が成功し、他のノードでは成功しない可能性があります。この場合、エントリーが更新されるまで、古い読み取りが発生する可能性があります。

7.2. キャッシュの可用性およびデグレードモード

DENY_READ_WRITES または **ALLOW_READS** パーティション処理ストラテジーのいずれかを使用するように設定すると、データの整合性を維持するために、Data Grid はキャッシュを **DEGRADED** モードに設定できます。

Data Grid は、以下の条件が満たされる場合に、パーティション内のキャッシュを **DEGRADED** モードに設定します。

- 1つ以上のセグメントですべての所有者が失われている。
これは、分散キャッシュの所有者の数と同じか、それ以上の数のノードがクラスターを離れている場合に生じます。
- パーティションに大多数のノードがない。
大多数のノードとは、最新の安定したトポロジ (クラスターのリバランス操作が最後に正常に完了した時) からのクラスター内のノード合計数の過半数です。

キャッシュが **DEGRADED** モードの場合、Data Grid は以下を行います。

- エントリーのすべてのレプリカが同じパーティションにある場合にのみ、読み取りおよび書き込み操作を許可する。
- パーティションにエントリーのすべてのレプリカが含まれていない場合は、読み取り操作および書き込み操作を拒否し、**AvailabilityException** を出力する。



注記

ALLOW_READS ストラテジーを使用すると、Data Grid は **DEGRADED** モードのキャッシュで読み取り操作を許可します。

DEGRADED モードは、異なるパーティションの同じキーに対して書き込み操作が行われないようにすることで一貫性を保証します。さらに、**DEGRADED** モードは、キーが1つのパーティションで更新され、別のパーティションで読み取られる場合に発生する古い読み取り操作を防ぎます。

すべてのパーティションが **DEGRADED** モードにある場合は、クラスターに最新の安定したトポロジからの過半数のノードが含まれ、各エントリーに少なくとも1つのレプリカがある場合にのみ、マージ後にキャッシュが再び利用可能になります。クラスターに各エントリーのレプリカが少なくとも1つあ

る場合、キーが失われることはなく、Data Grid はクラスターのリバランス中に所有者の数に基づいて新しいレプリカを作成できます。

あるパーティションで **DEGRADED** モードに設定されている間、場合によっては、別のパーティションのキャッシュを引き続き利用できます。これが生じると、利用可能なパーティションは通常通りにキャッシュ操作を続行し、Data Grid はそれらのノード間でデータのリバランスを試行します。キャッシュを1つにマージするために、Data Grid は必ず利用可能なパーティションから **DEGRADED** モードのパーティションに状態を転送します。

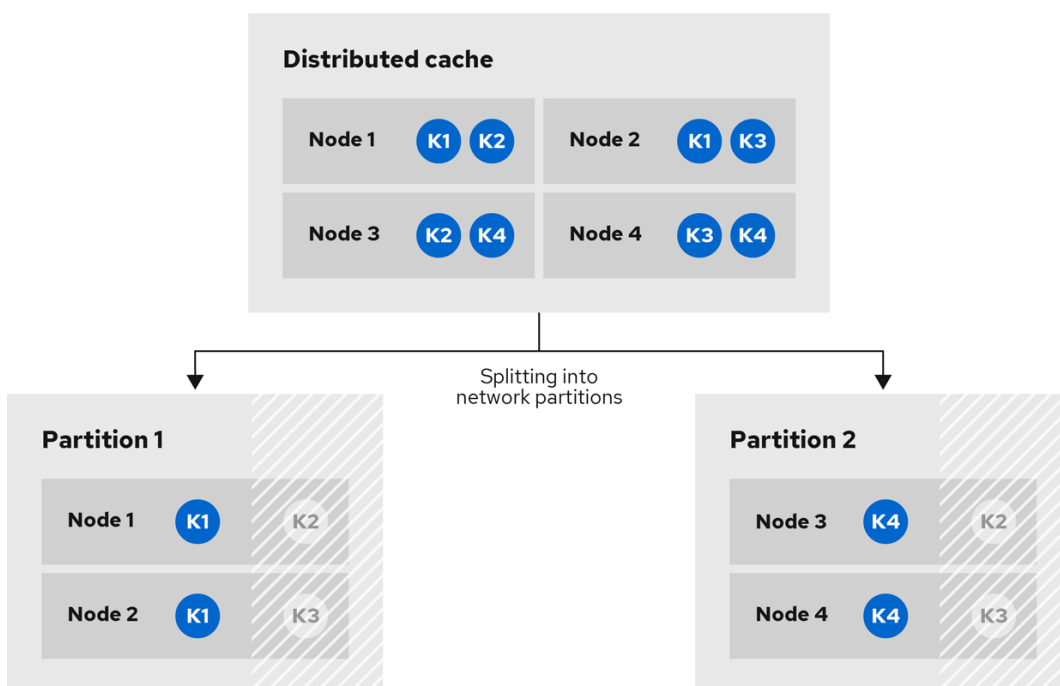
7.2.1. 低下したキャッシュのリカバリー例

このトピックでは、Data Grid が **DENY_READ_WRITES** パーティション処理戦略を使用するキャッシュを持つ分割されたクラスターからリカバリーする方法を示しています。

たとえば、Data Grid クラスターには4つのノードがあり、各エントリーに対してレプリカが2つある分散キャッシュが含まれています (**owners=2**)。キャッシュには、**k1**、**k2**、**k3**、および **k4** の4つのエントリーがあります。

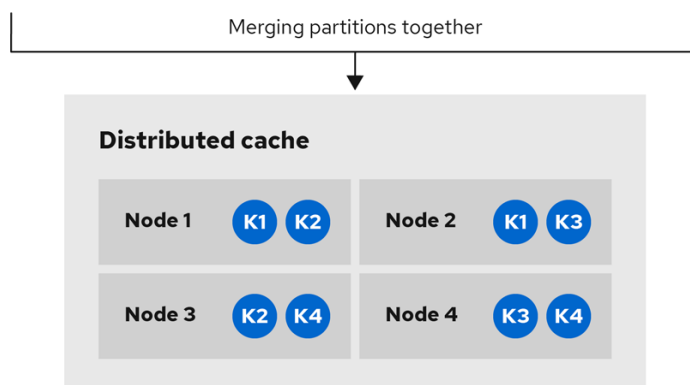
DENY_READ_WRITES 戦略では、クラスターがパーティションに分割されると、Data Grid はエントリーのすべてのレプリカが同じパーティションにある場合にのみキャッシュ操作を許可します。

以下の図では、キャッシュがパーティションに分割されている間、Data Grid はパーティション1上の **k1** およびパーティション2の **k4** の読み取りおよび書き込み操作を許可します。パーティション1またはパーティション2の **k2** と **k3** のにはレプリカが1つしかないため、Data Grid はこれらのエントリーの読み取りおよび書き込み操作を拒否します。



205_Data_Grid_0122

ネットワーク条件により、ノードが同じクラスタービューに再ジョインするのが許可されると、Data Grid は状態の送信なしにパーティションをマージし、通常のキャッシュ操作を回復します。



205_Data_Grid_0123

7.2.2. ネットワークパーティション時のキャッシュ可用性の確認

ネットワークパーティション時に、Data Grid クラスターのキャッシュが **AVAILABLE** モードまたは **DEGRADED** モードにあるかどうかを判別します。

Data Grid クラスターがパーティションに分割されると、それらのパーティションのノードは **DEGRADED** モードに入り、データの一貫性を確保できます。**DEGRADED** モードでは、クラスターは可用性の喪失につながるキャッシュ操作を許可しません

手順

以下のいずれかの方法で、ネットワークパーティションでクラスター化されたキャッシュの可用性を確認します。

- Data Grid ログで、クラスターが利用可能か、少なくとも1つのキャッシュが **DEGRADED** モードにあるかどうかを示す **ISPN100011** メッセージを探します。
- Data Grid Console または REST API を使用して、リモートキャッシュの可用性を取得します。
 - ブラウザーで Data Grid Console を開き、**Data Container** タブを選択し、**Health** 列で可用性のステータスを特定します。
 - REST API からキャッシュの健全性を取得します。

```
GET /rest/v2/cache-managers/<cacheManagerName>/health
```

- **AdvancedCache** API の **getAvailability()** メソッドを使用して、組み込みキャッシュの可用性をプログラマ的に取得します。

関連情報

- [REST API: Getting cluster health](#)
- [org.infinispan.AdvancedCache.getAvailability](#)
- [Enum AvailabilityMode](#)

7.2.3. キャッシュを使用できるようにする

キャッシュを強制的に **DEGRADED** モードから解除することで、キャッシュを読み取りおよび書き込み操作に利用できるようにします。



重要

デプロイメントでデータ喪失や不整合を許容できる場合にのみ、クラスターを強制的に **DEGRADED** モードから解除する必要があります。

手順

以下のいずれかの方法で、キャッシュを利用できるようにします。

- REST API でリモートキャッシュの可用性を変更します。

```
POST /v2/caches/<cacheName>?action=set-availability&availability=AVAILABLE
```

- **AdvancedCache** API で、組み込みキャッシュの可用性をプログラマ的に変更します。

```
AdvancedCache ac = cache.getAdvancedCache();
// Retrieve cache availability
boolean available = ac.getAvailability() == AvailabilityMode.AVAILABLE;
// Make the cache available
if (!available) {
    ac.setAvailability(AvailabilityMode.AVAILABLE);
}
```

関連情報

- [REST API: Setting cache availability](#)
- [org.infinispan.AdvancedCache](#)

7.3. パーティション処理の設定

パーティション処理ストラテジーとマージポリシーを使用するように Data Grid を設定し、ネットワークの問題が発生したときに分離されたクラスターを解決できるようにします。デフォルトでは、Data Grid はデータの一貫性を犠牲にして可用性を提供するストラテジーを使用します。ネットワークのパーティションによってクラスターが分割されると、クライアントは引き続きキャッシュで読み取りおよび書き込み操作を実行できます。

可用性よりも整合性が要求される場合は、クラスターがパーティションに分割されている間に、読み取りおよび書き込み操作を拒否するように Data Grid を設定することができます。または、読み取り操作を許可し、書き込み操作を拒否できます。また、カスタムのマージポリシー実装を指定して、Data Grid を設定し、要件に合わせたカスタムロジックで分割を解決することもできます。

前提条件

- レプリケートされたキャッシュまたは分散キャッシュのいずれかを作成できる Data Grid クラスターが必要です。



注記

パーティション処理の設定は、レプリケートされたキャッシュと分散キャッシュにのみ適用されます。

手順

1. Data Grid 設定を開いて編集します。
2. **partition-handling** 要素または **partitionHandling()** メソッドのいずれかを使用して、キャッシュにパーティション処理設定を追加します。
3. **when-split** 属性または **whenSplit()** メソッドを使用して、クラスターがパーティションに分割される際に Data Grid が使用するストラテジーを指定します。
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** であるため、キャッシュは利用可能のままです。ユースケースでキャッシュの可用性よりデータの整合性が要求される場合は、**DENY_READ_WRITES** ストラテジーを指定します。
4. **merge-policy** 属性または **mergePolicy()** メソッドで、パーティションをマージする際に Data Grid が競合するエントリーを解決するために使用するポリシーを指定します。
デフォルトでは、Data Grid はマージ時の競合を解決しません。
5. 変更を Data Grid の設定に保存します。

パーティション処理の設定

XML

```
<distributed-cache>
  <partition-handling when-split="DENY_READ_WRITES"
    merge-policy="PREFERRED_ALWAYS"/>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "partition-handling": {
      "when-split": "DENY_READ_WRITES",
      "merge-policy": "PREFERRED_ALWAYS"
    }
  }
}
```

YAML

```
distributedCache:
  partitionHandling:
    whenSplit: DENY_READ_WRITES
    mergePolicy: PREFERRED_ALWAYS
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC)
    .partitionHandling()
    .whenSplit(PartitionHandling.DENY_READ_WRITES)
    .mergePolicy(MergePolicy.PREFERRED_NON_NULL);
```

7.4. パーティション処理ストラテジー

パーティション処理ストラテジーは、クラスターの分割時に Data Grid が読み取りおよび書き込み操作を許可するかどうかを制御します。設定するストラテジーにより、キャッシュの可用性またはデータの整合性を優先するかどうかが決まります。

表7.1 パーティション処理ストラテジー

ストラテジー	説明	可用性または一貫性
ALLOW_READ_WRITES	クラスターがネットワークパーティションに分割されている間、Data Grid はキャッシュに対する読み取りおよび書き込み操作を許可します。各パーティションのノードは可用性を維持し、相互に独立して機能します。これは、デフォルトのパーティション処理ストラテジーです。	可用性
DENY_READ_WRITES	エントリーのすべてのレプリカがパーティションにある場合にのみ、Data Grid は読み取りおよび書き込み操作を許可します。パーティションにエントリーのすべてのレプリカが含まれていない場合、Data Grid はそのエントリーのキャッシュ操作を防ぎます。	一貫性
ALLOW_READS	パーティションにエントリーのすべてのレプリカが含まれない限り、Data Grid はエントリーに対する読み取り操作を許可し、書き込み操作を防ぎます。	読み取りが可能な一貫性

7.5. マージポリシー

マージポリシーは、クラスターパーティションを1つにまとめる際に Data Grid がレプリカ間の競合を解決する方法を制御します。Data Grid が提供するマージポリシーのいずれかを使用するか、**EntryMergePolicy** API のカスタム実装を作成できます。

表7.2 Data Grid のマージポリシー

マージポリシー	説明	留意事項
NONE	Data Grid は、分割されたクラスターをマージする際に競合を解決しません。これは、デフォルトのマージポリシーです。	ノードはプライマリーの所有者ではないセグメントをドロップするため、データが失われる可能性があります。

マージポリシー	説明	留意事項
PREFERRED_ALWAYS	Data Grid は、クラスター内の過半数のノードに存在する値を検出し、競合を解決するのに使用します。	Data Grid は、古い値を使用して競合を解決する可能性があります。エントリーが過半数のノードで利用可能な場合でも、少数派側のパーティションで最後の更新が行われる可能性があります。
PREFERRED_NON_NULL	Data Grid は、クラスター上で見つかった最初の null 以外の値を使用して競合を解決します。	Data Grid は削除されたエントリーを復元する場合があります。
REMOVE_ALL	Data Grid は、競合するすべてのエントリーをキャッシュから削除します。	分割されたクラスターをマージする際に、異なる値を持つエントリーが失われます。

7.6. カスタムマージポリシーの設定

ネットワークパーティションの処理時に **EntryMergePolicy** API のカスタム実装を使用するように Data Grid を設定します。

前提条件

- **EntryMergePolicy** API を実装している。

```
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 resolves the conflict

        return the_solved_CacheEntry;
    }
}
```

手順

1. リモートキャッシュを使用する場合は、マージポリシーの実装を Data Grid Server にデプロイします。
 - a. マージポリシーの完全修飾クラス名が含まれる **META-INF/services/org.infinispan.conflict.EntryMergePolicy** ファイルが含まれる JAR ファイルとしてクラスをパッケージ化します。

```
# List implementations of EntryMergePolicy with the full qualified class name
org.example.CustomMergePolicy
```

- b. JAR ファイルを **server/lib** ディレクトリーに追加します。
2. Data Grid 設定を開いて編集します。

- 必要に応じて **encoding** 要素または **encoding()** メソッドを使用して、適宜キャッシュエンコーディングを設定します。
リモートキャッシュでは、エントリーのマージ時に比較にオブジェクトメタデータのみを使用する場合は、メディアタイプとして **application/x-protostream** を使用できます。この場合、Data Grid はエントリーを **EntryMergePolicy** に **byte[]** として返します。

競合のマージ時にオブジェクト自体が必要な場合、キャッシュを **application/x-java-object** メディアタイプで設定する必要があります。この場合、関連する ProtoStream マーシャラーを Data Grid Server にデプロイし、クライアントが Protobuf エンコーディングを使用する場合に、オブジェクト変換に **byte[]** を実行できるようにする必要があります。
- パーティション処理設定の一部として **merge-policy** 属性または **mergePolicy()** メソッドを使用して、カスタムのマージポリシーを指定します。
- 変更を保存します。

カスタムマージポリシーの設定

XML

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

JSON

```
{
  "distributed-cache": {
    "partition-handling": {
      "when-split": "DENY_READ_WRITES",
      "merge-policy": "org.example.CustomMergePolicy"
    }
  }
}
```

YAML

```
distributedCache:
  partitionHandling:
    whenSplit: DENY_READ_WRITES
    mergePolicy: org.example.CustomMergePolicy
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC)
    .partitionHandling()
    .whenSplit(PartitionHandling.DENY_READ_WRITES)
    .mergePolicy(new CustomMergePolicy());
```

関連情報

- org.infinispan.conflict.EntryMergePolicy

7.7. 組み込みキャッシュでのパーティションの手動マージ

ネットワークパーティションの発生後に、組み込みキャッシュを手動でマージするために競合するエントリーを検出し、解決します。

手順

- 以下の例のように、**EmbeddedCacheManager** から **ConflictManager** を取得し、キャッシュ内の競合するエントリーを検出して解決します。

```
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 a 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、基礎となるスプリタレイターは、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ます。

第8章 ユーザーロールとパーミッションの設定

承認は、ユーザーがキャッシュにアクセスしたり、Data Grid リソースとやり取りしたりする前に、特定の権限を持つ必要があるセキュリティ機能です。読み取り専用アクセスから完全なスーパーユーザー特権まで、さまざまなレベルのパーミッションを提供するロールをユーザーに割り当てます。

8.1. セキュリティー-認証

Data Grid の認証は、ユーザーアクセスを制限することでデプロイメントを保護します。

ユーザーアプリケーションまたはクライアントは、Cache Manager またはキャッシュで操作を実行する前に、十分なパーミッションが割り当てられたロールに属している必要があります。

たとえば、特定のキャッシュインスタンスで承認を設定して、**Cache.get()** を呼び出すには、読み取り権限を持つロールを ID に割り当てる必要があります、**Cache.put()** を呼び出すには書き込み権限を持つロールが必要になるようにします。

このシナリオでは、**io** ロールが割り当てられたユーザーアプリケーションまたはクライアントがエントリーの書き込みを試みると、Data Grid はリクエストを拒否し、セキュリティ例外を出力します。**writer** ロールのあるユーザーアプリケーションまたはクライアントが書き込みリクエストを送信する場合、Data Grid は承認を検証し、後続の操作のためにトークンを発行します。

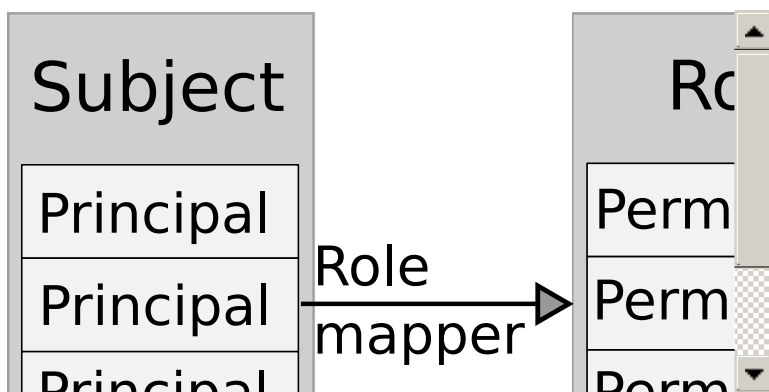
アイデンティティー

アイデンティティーは **java.security.Principal** タイプのセキュリティプリンシパルです。**javax.security.auth.Subject** クラスで実装されたサブジェクトは、セキュリティプリンシパルのグループを表します。つまり、サブジェクトはユーザーとそれが属するすべてのグループを表します。

ロールのアイデンティティー

Data Grid はロールマッパーを使用するため、セキュリティプリンシパルが1つ以上のパーミッションを割り当てるロールに対応します。

以下の図は、セキュリティプリンシパルがロールにどのように対応するかを示しています。



8.1.1. ユーザーロールとパーミッション

Data Grid には、データにアクセスして Data Grid リソースと対話するためのパーミッションをユーザーに付与するデフォルトのロールのセットが含まれています。

ClusterRoleMapper は、Data Grid がセキュリティプリンシパルを承認ロールに関連付けるために使用するデフォルトのメカニズムです。



重要

ClusterRoleMapper は、プリンシパル名をロール名に一致させます。**admin** という名前のユーザーは **admin** パーミッションを自動的に取得し、**deployer** という名前のユーザーは **deployer** パーミッションを取得する、というようになります。

ロール	パーミッション	説明
admin	ALL	Cache Manager ライフサイクルの制御など、すべてのパーミッションを持つスーパーユーザー。
deployer	ALL_READ、ALL_WRITE、LISTEN、EXEC、MONITOR、CREATE	application パーミッションに加えて、Data Grid リソースを作成および削除できます。
application	ALL_READ、ALL_WRITE、LISTEN、EXEC、MONITOR	observer パーミッションに加え、Data Grid リソースへの読み取りおよび書き込みアクセスがあります。また、イベントをリッスンし、サーバータスクおよびスクリプトを実行することもできます。
observer	ALL_READ、MONITOR	monitor パーミッションに加え、Data Grid リソースへの読み取りアクセスがあります。
monitor	MONITOR	JMX および metrics エンドポイント経由で統計を表示できます。

参照資料

- org.infinispan.security.AuthorizationPermission Enumeration
- [Data Grid 設定スキーマ参照](#)

8.1.2. パーミッション

承認ロールには、Data Grid へのアクセスレベルが異なるさまざまなパーミッションがあります。パーミッションを使用すると、Cache Manager とキャッシュの両方へのユーザーアクセスを制限できます。

8.1.2.1. Cache Manager のパーミッション

パーミッション	機能	説明
設定	defineConfiguration	新しいキャッシュ設定を定義します。

パーミッション	機能	説明
LISTEN	addListener	キャッシュマネージャーに対してリスナーを登録します。
ライフサイクル	stop	キャッシュマネージャーを停止します。
CREATE	createCache, removeCache	キャッシュ、カウンター、スキーマ、スクリプトなどのコンテナリソースを作成および削除することができます。
MONITOR	getStats	JMX 統計および metrics エンドポイントへのアクセスを許可します。
ALL	-	すべてのキャッシュマネージャーのアクセス許可が含まれます。

8.1.2.2. キャッシュ権限

パーミッション	機能	説明
READ	get, contains	キャッシュからエントリーを取得します。
WRITE	put, putIfAbsent, replace, remove, evict	キャッシュ内のデータの書き込み、置換、削除、エビクト。
EXEC	distexec, streams	キャッシュに対するコードの実行を許可します。
LISTEN	addListener	キャッシュに対してリスナーを登録します。
BULK_READ	keySet, values, entrySet, query	一括取得操作を実行します。
BULK_WRITE	clear, putAll	一括書き込み操作を実行します。
ライフサイクル	start, stop	キャッシュを開始および停止します。

パーミッション	機能	説明
ADMIN	getVersion, addInterceptor*, removeInterceptor, getInterceptorChain, getEvictionManager, getComponentRegistry, getDistributionManager, getAuthorizationManager, evict, getRpcManager, getCacheConfiguration, getCacheManager, getInvocationContextContainer, setAvailability, getDataContainer, getStats, getXAResource	基盤となるコンポーネントと内部構造へのアクセスを許可します。
MONITOR	getStats	JMX 統計および metrics エンドポイントへのアクセスを許可します。
ALL	-	すべてのキャッシュパーミッションが含まれます。
ALL_READ	-	READ パーミッションと BULK_READ パーミッションを組み合わせます。
ALL_WRITE	-	WRITE パーミッションと BULK_WRITE パーミッションを組み合わせます。

関連情報

- [Data Grid Security API](#)

8.1.3. ロールマッパー

Data Grid には、サブジェクトのセキュリティープリンシパルをユーザーに割り当てる承認ロールにマップする **PrincipalRoleMapper** API が含まれています。

8.1.3.1. クラスターのロールマッパー

ClusterRoleMapper は永続的にレプリケートされたキャッシュを使用して、デフォルトのロールおよびパーミッションのプリンシパルからロールへのマッピングを動的に保存します。

デフォルトでは、プリンシパル名をロール名として使用し、実行時にロールマッピングを変更するメソッドを公開する **org.infinispan.security.MutableRoleMapper** を実装します。

- Java クラス: **org.infinispan.security.mappers.ClusterRoleMapper**

- 宣言型設定: `<cluster-role-mapper />`

8.1.3.2. ID ロールマッパー

IdentityRoleMapper は、プリンシパル名をロール名として使用します。

- Java クラス: `org.infinispan.security.mappers.IdentityRoleMapper`
- 宣言型設定: `<identity-role-mapper />`

8.1.3.3. CommonName ロールマッパー

CommonNameRoleMapper は、プリンシパル名が識別名 (DN) の場合は Common Name (CN) をロール名として使用します。

たとえば、この DN (`cn=managers,ou=people,dc=example,dc=com`) は **managers** ロールにマッピングします。

- Java クラス: `org.infinispan.security.mappers.CommonRoleMapper`
- 宣言型設定: `<common-name-role-mapper />`

8.1.3.4. カスタムロールマッパー

カスタムロールマッパーは `org.infinispan.security.PrincipalRoleMapper` の実装です。

- 宣言型設定: `<custom-role-mapper class="my.custom.RoleMapper" />`

関連情報

- [Data Grid Security API](#)
- [org.infinispan.security.PrincipalRoleMapper](#)

8.2. アクセス制御リスト (ACL) キャッシュ

Data Grid は、パフォーマンスの最適化のために内部でユーザーに付与するロールをキャッシュします。ロールをユーザーに付与または拒否するたびに、Data Grid は ACL キャッシュをフラッシュして、ユーザーのパーミッションが正しく適用されていることを確認します。

必要に応じて、ACL キャッシュを無効にするか、**cache-size** および **cache-timeout** 属性を使用してこれを設定することができます。

XML

```
<infinispan>
  <cache-container name="acl-cache-configuration">
    <security cache-size="1000"
      cache-timeout="300000">
      <authorization/>
    </security>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan": {
    "cache-container": {
      "name": "acl-cache-configuration",
      "security": {
        "cache-size": "1000",
        "cache-timeout": "300000",
        "authorization": {}
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    name: "acl-cache-configuration"
  security:
    cache-size: "1000"
    cache-timeout: "300000"
    authorization: ~
```

関連情報

- [Data Grid 設定スキーマ参照](#)

8.3. ロールおよびパーミッションのカスタマイズ

Data Grid 設定の認証設定をカスタマイズして、異なるロールとパーミッションの組み合わせでロールマッパーを使用できます。

手順

1. Cache Manager 設定でロールマッパーとカスタムロールとパーミッションのセットを宣言します。
2. ユーザーロールに基づいてアクセスを制限するようにキャッシュの承認を設定します。

カスタムロールおよびパーミッションの設定

XML

```
<infinispan>
  <cache-container name="custom-authorization">
    <security>
      <authorization>
        <!-- Declare a role mapper that associates a security principal
             to each role. -->
        <identity-role-mapper />
        <!-- Specify user roles and corresponding permissions. -->
```

```

<role name="admin" permissions="ALL" />
<role name="reader" permissions="READ" />
<role name="writer" permissions="WRITE" />
<role name="supervisor" permissions="READ WRITE EXEC"/>
</authorization>
</security>
</cache-container>
</infinispan>

```

JSON

```

{
  "infinispan" : {
    "cache-container" : {
      "name" : "custom-authorization",
      "security" : {
        "authorization" : {
          "identity-role-mapper" : null,
          "roles" : {
            "reader" : {
              "role" : {
                "permissions" : "READ"
              }
            },
            "admin" : {
              "role" : {
                "permissions" : "ALL"
              }
            },
            "writer" : {
              "role" : {
                "permissions" : "WRITE"
              }
            },
            "supervisor" : {
              "role" : {
                "permissions" : "READ WRITE EXEC"
              }
            }
          }
        }
      }
    }
  }
}

```

YAML

```

infinispan:
  cacheContainer:
    name: "custom-authorization"
  security:
    authorization:
      identityRoleMapper: "null"

```

```

roles:
  reader:
    role:
    permissions:
      - "READ"
  admin:
    role:
    permissions:
      - "ALL"
  writer:
    role:
    permissions:
      - "WRITE"
  supervisor:
    role:
    permissions:
      - "READ"
      - "WRITE"
      - "EXEC"

```

8.4. セキュリティー承認によるキャッシュの設定

キャッシュ設定で承認を使用して、ユーザーアクセスを制限します。キャッシュエントリーの読み取りや書き込み、キャッシュの作成または削除を行う前に、ユーザーは十分なレベルのパーミッションを持つロールを持っている必要があります。

前提条件

- **authorization** 要素が **cache-container** 設定の **security** セクションに含まれていることを確認します。
Data Grid はデフォルトで Cache Manager でセキュリティ承認を有効にし、キャッシュのグローバルロールおよびパーミッションを提供します。
- 必要な場合は、Cache Manager 設定でカスタムロールとパーミッションを宣言します。

手順

1. キャッシュ設定を開いて編集します。
2. **authorization** 要素をキャッシュに追加し、ロールおよびパーミッションに基づいてユーザーアクセスを制限します。
3. 変更を設定に保存します。

認証設定

以下の設定は、デフォルトのロールおよびパーミッションで暗黙的な認証設定を使用する方法を示しています。

XML

```

<distributed-cache>
  <security>
    <!-- Inherit authorization settings from the cache-container. --> <authorization/>

```



```

</security>
</distributed-cache>

```

JSON

```

{
  "distributed-cache": {
    "security": {
      "authorization": {
        "enabled": true
      }
    }
  }
}

```

YAML

```

distributedCache:
  security:
    authorization:
      enabled: true

```

カスタムロールおよびパーミッション

XML

```

<distributed-cache>
  <security>
    <authorization roles="admin supervisor"/>
  </security>
</distributed-cache>

```

JSON

```

{
  "distributed-cache": {
    "security": {
      "authorization": {
        "enabled": true,
        "roles": ["admin", "supervisor"]
      }
    }
  }
}

```

YAML

```

distributedCache:
  security:
    authorization:

```

```
enabled: true  
roles: ["admin", "supervisor"]
```

8.5. セキュリティー承認の無効化

ローカル開発環境では、ユーザーがロールおよびパーミッションを必要としないように、承認を無効にできます。セキュリティ承認を無効にすると、すべてのユーザーがデータにアクセスでき、Data Grid リソースと対話できます。

手順

1. Data Grid 設定を開いて編集します。
2. Cache Manager **security** 設定から **authorization** 要素を削除します。
3. キャッシュから **authorization** 設定を削除します。
4. 変更を設定に保存します。