Red Hat Data Grid 8.5

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

Last Updated: 2024-08-12

Red Hat Data Grid 8.5 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

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

第1章 DATA GRID キャッシュ 8
1.1. CACHE API 8
1.2. CACHE MANAGER 8
1.3. キャッシュモード 8
1.4. ローカルキャッシュ 10

第2章 クラスター化されたキャッシュ 12
2.1. レプリケートされたキャッシュ 12
2.2. 分散キャッシュ 13
2.3. インバリデーションキャッシュ 24
2.4. 非同期のレプリケーション 25
2.5. 初期クラスターサイズの設定 26

第3章 DATA GRID キャッシュの設定 28
3.1. 宣言型キャッシュの設定 28
3.2. キャッシュテンプレートの追加 35
3.3. キャッシュエイリアス 40
3.4. リモートキャッシュの作成 41
3.5. 組み込みキャッシュの作成 45

第4章 DATA GRID 統計および JMX 監視の有効化および設定 51
4.1. DATA GRID メトリックの設定 51
4.2. JMX MBEAN の登録 52
4.3. 状態遷移操作中のメトリクスのエクスポート 55
4.4. クロスサイトレプリケーションのステータスの監視 56

第5章 JVM メモリー使用量の設定 61
5.1. デフォルトのメモリー設定 61
5.2. エビクションと有効期限 61
5.3. DATA GRID キャッシュを使用したエビクション 62
5.4. ライフスパンと最大アイドル期間の有効期限 67
5.5. JVM ヒープおよびオフヒープメモリー 71

第6章 CONFIGURING PERSISTENT STORAGE 74
6.1. パッシベーション 74
6.2. ライトスルーキャッシュストア 75
6.3. WRITE-BEHIND キャッシュストア 76
6.4. セグメント化されたキャッシュストア 78
6.5. 共有キャッシュストア 78
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 79
6.7. グローバルの永続的な場所 79
6.8. ファイルベースのキャッシュストア 82
6.9. JDBC 接続ファクトリー 87
6.10. SQL キャッシュストア 96
6.11. JDBC 文字列ベースのキャッシュストア 110
6.12. ROCKSDB のキャッシュストア 113

6.13. リモートキャッシュストア 116
6.14. クラスターキャッシュローダー 120
6.15. カスタムキャッシュストア実装の作成 121
6.16. キャッシュストア間のデータの移行 123

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 130
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 130
7.2. キャッシュの可用性およびデグレードモード 131
7.3. パーティション処理の設定 134
7.4. パーティション処理ストラテジー 136
7.5. マージポリシー 136
7.6. カスタムマージポリシーの設定 137
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 139

第8章 ロールベースアクセス制御によるセキュリティー承認 140
8.1. DATA GRID のユーザーロールと権限 140
8.2. セキュリティー承認によるキャッシュの設定 148

第9章 トランザクションの設定 151
9.1. トランザクション 151

第10章 ロックと同時実行の設定 162
10.1. ロックおよび同時実行 162

第11章 クラスター化されたカウンターの使用 165
11.1. クラスター化カウンター 165

第12章 リスナーおよび通知 175
12.1. リスナーおよび通知 175
12.2. キャッシュレベルの通知 175
12.3. CACHE MANAGER 通知 177
12.4. イベントの同期 178

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。

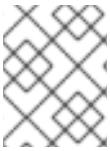
DATA GRID のドキュメント

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 8.5 ドキュメント
- Data Grid 8.5 コンポーネントの詳細
- Data Grid 8.5 でサポートされる構成
- 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 プロパティー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、用語の置き換えは、今後の複数のリリースにわたって段階的に実施されます。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第1章 DATA GRID キャッシュ

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

- 高速のローカルキャッシュでアプリケーション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。
- クラスター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. CACHE 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ーは、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. CACHE MANAGER

CacheManager API は、Data Grid と対話するためのエントリーポイントです。Cache Manager 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。Cache Manager は、ノード間でコードを実行する機能と共に、クラスターの管理および監視も提供します。

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ーを共有します。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 の Cache Manager は、異なるモードを使用する複数のキャッシュを作成および制御できます。たとえば、ローカルキャッシュ、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 Cache Manager 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ーに対して読み取り操作または書き込み操作を複製しません。

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。

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**
Distributed (分散)	**Yes**	**メディア**(所有者)	**メディア**(所有者ノード)	所有者数で区分されたすべてのノード容量の合計。	**所有者ノード**	**完了**

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
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。

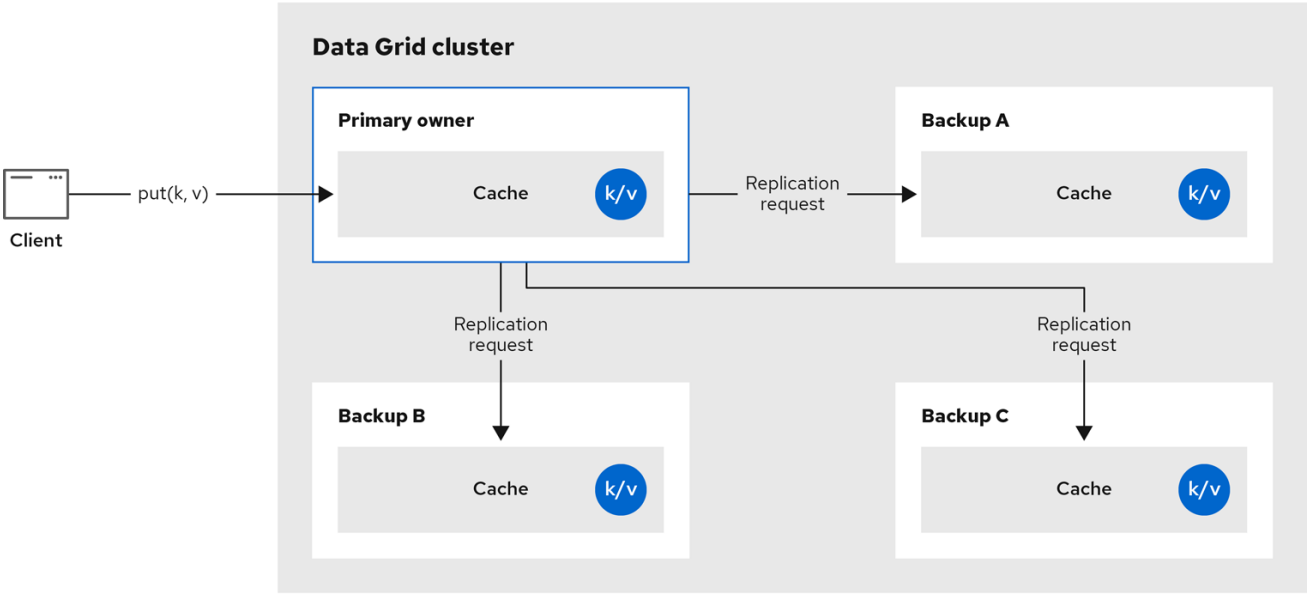
2.1. レプリケートされたキャッシュ

Data Grid は、キャッシュ内のすべてのエントリー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ー所有者があり、一貫性を提供するためにデータコンテナー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



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ー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

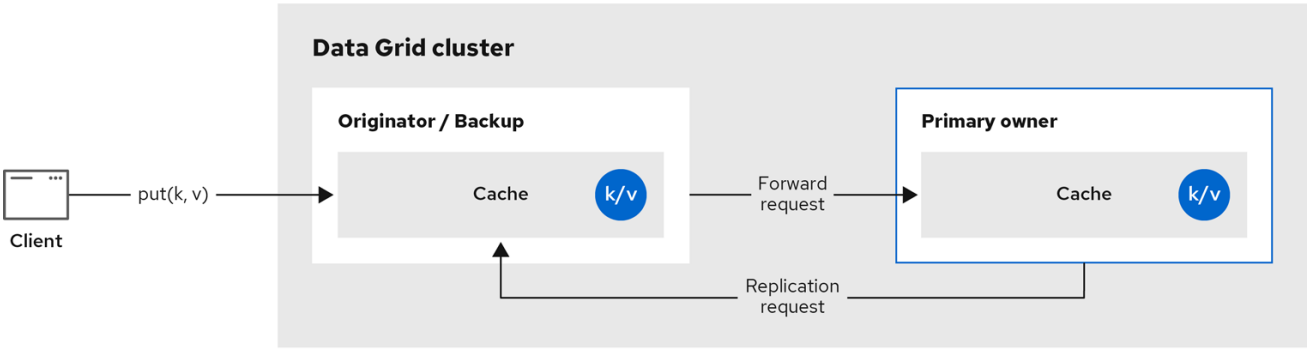
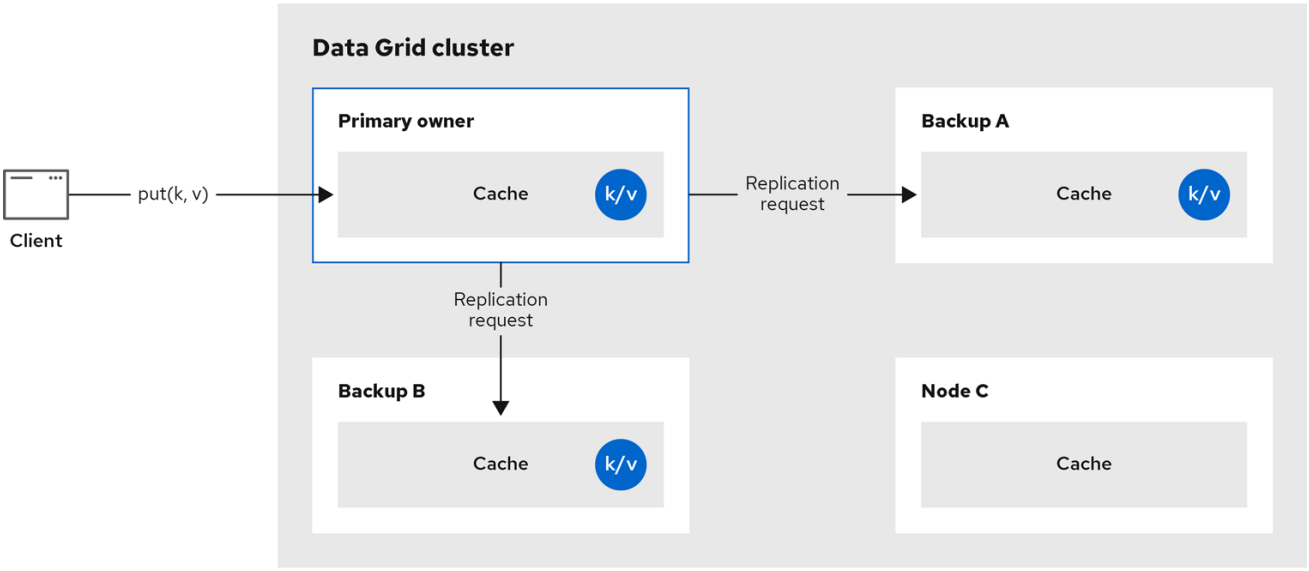


図2.3 分散キャッシュ



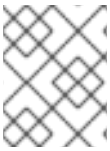
読み取り操作

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、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リストに割り当てます。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。リスト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ーが変更するると、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 プロパティーでは、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ーの変更に関係なく、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ー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	一貫性のあるハッシュ に基づくアルゴリズムを使用します。サーバーヒントを無効にした場合は、デフォルトで選択されています。 この実装では、クラスターが対称である限り、すべてのキャッシュの同じノードに常にキーが割り当てられます。つまり、すべてのキャッシュがすべてのノードで実行します。この実装には、負荷の分散が若干不均等であるため、負のポイントが若干異なります。また、参加または脱退時に厳密に必要な数よりも多くのセグメントを移動します。

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ー所有者とは異なるノードに保存されるように、トポロジー間でデータを分散するためにサーバーヒントで使用されます。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が、1つの欠点があります。ノードがクラスターに参加する順序によって、どのノードがどのセグメントを所有するかが決まります。その結果、キーは異なるキャッシュ内の異なるノードに割り当てられる可能性があります。
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ー所有者とは異なるノードに保存されるように、トポロジー間でデータを分散するためにサーバーヒントで使用されます。
ReplicatedConsistentHashFactory	レプリケートされたキャッシュの実装に内部で使用されます。このアルゴリズムは分散キャッシュで明示的に選択しないでください。

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、キャパシティー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

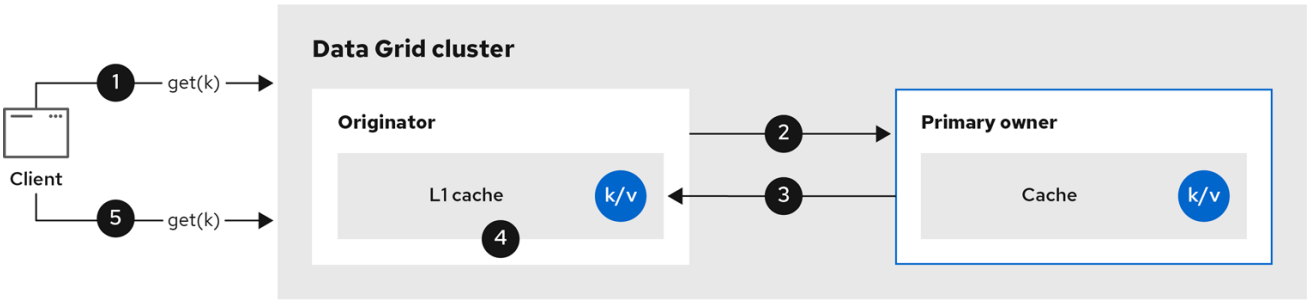
```
new GlobalConfigurationBuilder().zeroCapacityNode(true);
```

2.2.4. レベル 1(L1) キャッシュ

Data Grid ノードは、clsuter の別のノードからエントリーを取得すると、ローカルレプリカを作成します。L1 キャッシュは、プライマリー所有者ノードでエントリー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



"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ー所有者であるエントリーを読み取ります。
2. 元のノードは読み取り操作をプライマリー所有者に転送します。
3. プライマリー所有者はキー/値エントリーを返します。
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、プライマリー所有者に転送するのではなく、ローカルエントリーを返します。

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、エントリーが変更されたときにプライマリー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ーと同様に、ローカルレプリカまたは L1 エントリー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かを確認し、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、使用する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** パラメーターを持ち、これは非同期キー生成に使用されます (呼び出し元のスレッドでは処理されません)。ユーザーは、この **Executor** のシャットダウンを処理します。

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

KeyAffinityService がそれ自体で停止する唯一の状況は、登録済みの Cache Manager 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。主なアフィニティーサービスはこれらのトポロジーの変更を追跡し、現在別のノードにマップされるキーを返しませんが、先に生成したキーに関しては何も実行しません。

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

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

2.2.7. グループ化 API

キーアフィニティー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。

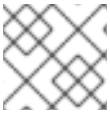
Grouping API を使用するには、グループを有効にする必要があります。

```
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推奨します。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推奨します。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>
```

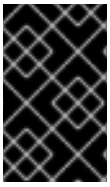
高度な API
AdvancedCache には、グループ固有のメソッドが 2 つあります。

- **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内のすべてのキーを削除します。

どちらのメソッドもデータコンテナー全体とストア (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. インバリデーションキャッシュ

Data Grid のインバリデーションキャッシュモードは、共有の永続データストアに対して大量の読み取り操作を実行するシステムを最適化することを目的としています。インバリデーションモードを使用すると、状態の変更が発生したときのデータベースへの書き込み回数を減らすことができます。



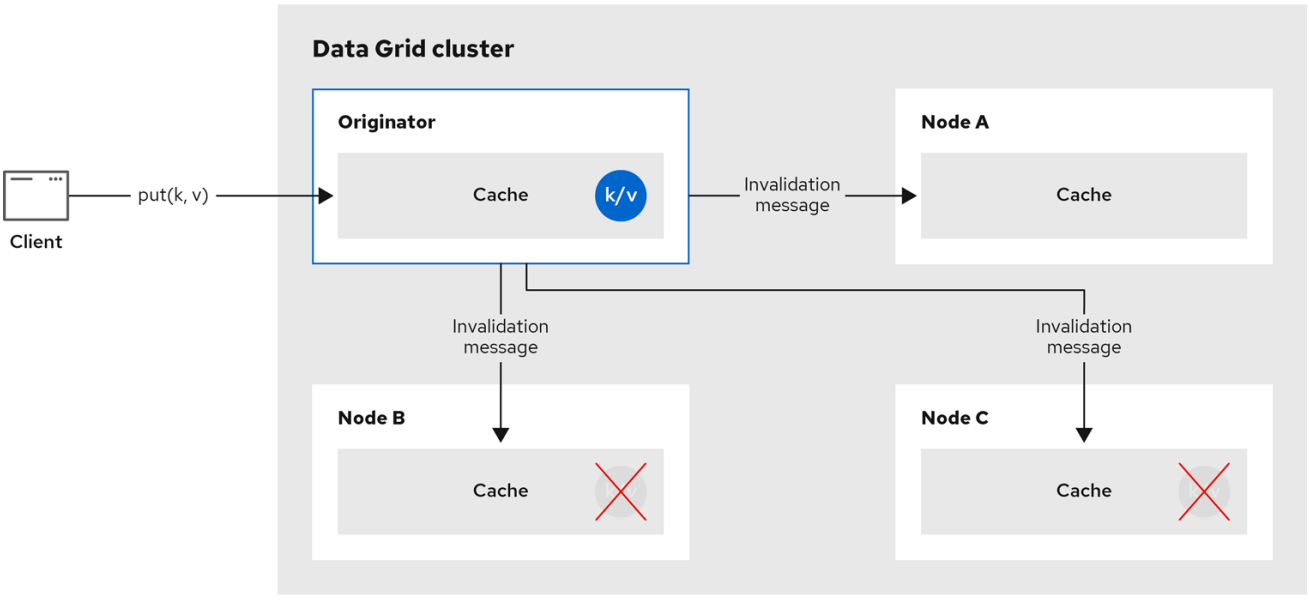
重要

インバリデーションキャッシュモードは、Data Grid リモートデプロイメントでは非推奨です。インバリデーションキャッシュモードは、共有キャッシュストアにのみ保存される埋め込みキャッシュで使用してください。

インバリデーションキャッシュモードが効果を発揮するのは、データベースなどの永続的なデータストアがあり、読み取りが多いシステムで読み取りごとにデータベースにアクセスするのを防ぐために Data Grid を最適化手法として使用する場合だけです。

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、キャッシュ内のデータが変更されるたびに、クラスター内の他のキャッシュにメッセージが送信され、データが古くなったことと、メモリーからデータを削除する必要があることが通知されます。インバリデーションメッセージにより、他のノードのメモリーから古い値が削除されます。このメッセージは、値全体を複製する場合に比べて非常に小さくなります。また、クラスター内の他のキャッシュが、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。共有ストアへの更新は、通常、ユーザーアプリケーションコードまたは Hibernate によって処理されます。

図2.5 インバリデーションキャッシュ



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。

重要

インバリデーションモードは、すべてのノードが同じデータにアクセスできる共有ストアにのみ適しています。永続ストアなしでインバリデーションモードを使用することは実用的ではありません。ノード間の一貫性を保つために、更新された値を共有ストアから読み取る必要があるためです。

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

インバリデーションキャッシュは、特別なキャッシュローダー (**ClusterLoader**) で設定することもできます。**ClusterLoader** が有効になっている場合、ローカルノードでキーが見つからない読み取り操作は、最初に他のすべてのノードからキーを要求し、ローカルのメモリーに保存します。これにより、古い値が保存される可能性があるため、古い値に対する許容度が高い場合にのみ使用してください。

同期または非同期のレプリケーション

同期の場合、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操作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. 非同期のレプリケーション

すべてのクラスター化された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推奨しません。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。

2.4.1. 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ーが変更された場合。トポロジーの変更により、Data Grid は書き込み操作を再試行します。また、再試行操作の戻り値は信頼性がありません。

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

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

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

これを実装する適切な方法として、**cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(k)** を使用します。

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

2.5. 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ないことを意味します。

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. Cache Manager 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ーまたは 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、JSON、および YAML 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、セキュリティーレルム、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 は例外を出力します。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。簡潔なファイル名を書きます。

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"/>
    <indexed-entities>
      <indexed-entity>org.infinispan.Person</indexed-entity>
    </indexed-entities>
  </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"
          },
          "indexed-entities": [
            "org.infinispan.Person"
          ]
    },
    "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"
    indexedEntities:
      - "org.infinispan.Person"
  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"/>
    <indexed-entities>
      <indexed-entity>org.infinispan.Person</indexed-entity>
    </indexed-entities>
  </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"
        },
      "indexed-entities": [
        "org.infinispan.Person"
      ]
    },
    "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"
    indexedEntities:
      - "org.infinispan.Person"
  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:14.0 https://infinispan.org/schemas/infinispan-config-
14.0.xsd
                          urn:infinispan:server:14.0 https://infinispan.org/schemas/infinispan-server-14.0.xsd"
      xmlns="urn:infinispan:config:14.0"
      xmlns:server="urn:infinispan:server:14.0">
  <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>
```

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"
            }
          }
        }
      }
    }
  }
}
```

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"
```

関連情報

- Data Grid 設定スキーマ参照
- infinispan-config-14.0.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 Cache Manager 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

- Cache Manager 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、プロパティーを上書きできます。

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

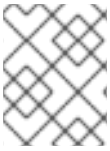
```
{
  "infinispan" : {
    "cache-container" : {
      "caches" : {
        "base-template" : {
          "distributed-cache-configuration" : {
            "expiration" : {
              "lifespan" : "5000"
            }
          }
        },
        "extended-template" : {
          "distributed-cache-configuration"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

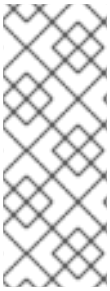
YAML

```
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** に追加する必要もあります。

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使用できる **infinispan-config-fragment-14.0.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:14.0 https://infinispan.org/schemas/infinispan-config-fragment-14.0.xsd"
             xmlns="urn:infinispan:config:14.0"
             name="mycache"/>
```

関連情報

- XInclude 仕様

3.3. キャッシュエイリアス

異なる名前を使用してキャッシュにアクセスするには、キャッシュにエイリアスを追加します。

ヒント

キャッシュ設定の **aliases** 属性は、ランタイム時に更新可能です。異なるキャッシュにエイリアスを再割り当てして、キャッシュの内容をすばやく切り替えます。

XML

```
<distributed-cache name="acache" aliases="0 anothername"/>
```

JSON

-

```
{
  "distributed-cache": {
    "aliases": ["0", "anothername"]
  }
}
```

YAML

```
distributedCache:
  aliases:
    - "0"
    - "anothername"
```

3.4. 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

3.4.1. 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ーで **describe** コマンドを使用します。

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

REST

任意のブラウザーで **127.0.0.1:11222/rest/v2/container/** 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: ~
```

3.4.2. 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。

3.4.3. 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 を起動します。

```
bin/cli.sh
```

2. **connect** コマンドを実行し、プロンプトが表示されたらユーザー名とパスワードを入力します。
3. **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リスト表示します。

```
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
```

3.4.4. 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#、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については、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

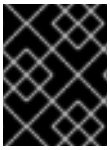
- **ConfigurationBuilder** の一部として **remoteCache()** メソッドを呼び出します。
- クラスパスの **hotrod-client.properties** ファイルで **configuration** または **configuration_uri** プロパティーを設定します。

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

3.4.5. 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

3.5. 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。

3.5.1. 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 リポジトリーから 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>
```

3.5.2. 組み込みキャッシュの作成と使用

Data Grid は、Cache Manager を制御する **GlobalConfigurationBuilder** API と、キャッシュを設定する **ConfigurationBuilder** 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. **CacheManager** を初期化します。

注記

キャッシュを作成する前に、必ず **cacheManager.start()** メソッドを呼び出して **CacheManager** を初期化する必要があります。デフォルトのコンストラクターはこれを行いますが、オーバーロードされたバージョンのコンストラクターはこれを行いません。

Cache Manager も重量のあるオブジェクトであり、Data Grid では、JVM ごとに 1 つのインスタンスのみをインスタンス化することを推奨します。

2. **ConfigurationBuilder** API を使用して、キャッシュ設定を定義します。
3. **getCache()**、**createCache()**、または **getOrCreateCache()** メソッドで、キャッシュを取得します。
   Data Grid では、**getOrCreateCache()** メソッドを使用することを推奨します。これは、すべてのノードでキャッシュを作成するか、既存のキャッシュを返すためです。
4. 必要であれば、キャッシュが再起動しても大丈夫なように、**PERMANENT** フラグを使用します。
5. **cacheManager.stop()** メソッドを呼び出して **CacheManager** を停止し、JVM リソースを解放してキャッシュを正常にシャットダウンします。

```
// 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());
// Stop the Cache Manager.
cacheManager.stop();
```

getCache() メソッド

以下のように、キャッシュを取得するために **getCache(String)** メソッドを呼び出します。

```
Cache<String, String> myCache = manager.getCache("myCache");
```

上記の操作は、**myCache** という名前のキャッシュがまだ存在しない場合は作成し、それを返します。

getCache() メソッドを使用すると、メソッドを呼び出すノードにのみキャッシュが作成されます。つまり、クラスター全体の各ノードで呼び出す必要のあるローカル操作を実行します。通常、複数のノードにまたがってデプロイされたアプリケーションは、初期化中にキャッシュを取得して、キャッシュが**対称**であり、各ノードに存在することを確認します。

createCache() メソッド

createCache() メソッドを呼び出して、クラスター全体でキャッシュを動的に作成します。

```
Cache<String, String> myCache = manager.administration().createCache("myCache",
"myTemplate");
```

上記の操作では、後でクラスターに参加するすべてのノードにキャッシュが自動的に作成されます。

createCache() メソッドを使用して作成するキャッシュは、デフォルトでは一時的です。クラスター全体がシャットダウンした場合、再起動時にキャッシュが自動的に再作成されることはありません。

PERMANENT フラグ

PERMANENT フラグを使用して、キャッシュが再起動後も存続できるようにします。

```
Cache<String, String> myCache =
manager.administration().withFlags(AdminFlag.PERMANENT).createCache("myCache",
"myTemplate");
```

PERMANENT フラグを有効にするには、グローバルの状態を有効にし、設定ストレージプロバイダーを設定する必要があります。

設定ストレージプロバイダーの詳細は、GlobalStateConfigurationBuilder#configurationStorage() を参照してください。

関連情報

- EmbeddedCacheManager
- EmbeddedCacheManager Configuration
- org.infinispan.configuration.global.GlobalConfiguration
- org.infinispan.configuration.cache.ConfigurationBuilder

3.5.3. Cache API

Data Grid は、JDK の ConcurrentMap インターフェイスによって公開されるアトミックメカニズムを含む、エントリーを追加、取得、および削除するための簡単なメソッドを公開する Cache インターフェイスを提供します。使用されるキャッシュモードに基づいて、これらのメソッドを呼び出すと、リモートノードにエントリーを複製したり、リモートノードからエントリーを検索することやキャッシュストアからエントリーを検索することなど、数多くのことが発生します。

単純な使用の場合、Cache API の使用は JDK Map API の使用と違いがないはずです。したがって、マップに基づく単純なインメモリーキャッシュから Data Grid のキャッシュへの移行は簡単になります。

特定のマップメソッドのパフォーマンスに関する懸念

size()、values()、keySet()、および entrySet() など、マップで公開される特定のメソッドは、Data Grid と使用すると特定のパフォーマンスに影響します。**keySet** の特定のメソッドである **values** および **entrySet** を使用できます。詳細については、Javadoc を参照してください。

これらの操作をグローバルに実行しようとすると、パフォーマンスに大きな影響を及ぼし、スケーラビリティーのボトルネックにもなります。そのため、これらの方法は情報またはデバッグの目的でのみ使用してください。

withFlags() メソッドで特定のフラグを使用すると、これらの問題の一部を軽減できる点に留意してください。詳細は、各メソッドのドキュメントを参照してください

Mortal および Immortal データ

単にエントリーを格納するだけでなく、Data Grid のキャッシュ API を使用すると、期限付き情報をデータに添付できます。たとえば、単に put(key, value) を使用すると、**immortal** エントリーが作成されます。このエントリーは削除されるまで (またはメモリー不足にならないようにメモリーからエビクトされるまで)、いつまでもキャッシュに存在します。ただし、put(key, value, lifespan, timeunit) を使用してデータをキャッシュに格納すると、有効期限が固定され、その有効期限が過ぎると期限切れになるエントリーである、**mortal** エントリーが作成されます。

Data Grid は、**lifespan**の他に、有効期限を決定する追加のメトリックとして**maxIdle**もサポートします。lifespans または maxIdles の任意の組み合わせを使用できます。

putForExternalRead 操作

Data Grid の Cache クラスには、putForExternalRead と呼ばれる異なる 'put' 操作が含まれます。この操作は、他の場所で保持されるデータの一時キャッシュとして Data Grid が使用される場合に特に便利です。読み取りが非常に多い場合、キャッシュは単に最適化のために行われ、妨害するものではないため、キャッシュの競合によって実際のトランザクションが遅延してはなりません。

これを実現するため、キーがキャッシュ内に存在しない場合にのみ動作する put 呼び出しとして**putForExternalRead()**が動作し、別のスレッドが同じキーを同時に格納しようとすると、通知なしに即座に失敗します。このシナリオでは、データのキャッシュはシステムを最適化する方法で、キャッシングの失敗が実行中のトランザクションに影響するのは望ましくないため、失敗の処理方法が異なります。成功したかどうかに関わらず、ロックを待たず、読み出し元に即座に返されるため、**putForExternalRead()**は高速な操作とみなされます。

この操作の使用方法を理解するために、基本的な例を見てみましょう。PersonId によって入力される Person インスタンスのキャッシュを想像してください。このデータは個別のデータストアから入力されます。以下のコードは、この例のコンテキスト内で putForExternalRead を使用する最も一般的なパターンを示しています。

```
// Id of the person to look up, provided by the application
PersonId id = ...;

// Get a reference to the cache where person instances will be stored
Cache<PersonId, Person> cache = ...;

// First, check whether the cache contains the person instance
// associated with with the given id
Person cachedPerson = cache.get(id);

if (cachedPerson == null) {
   // The person is not cached yet, so query the data store with the id
   Person person = dataStore.lookup(id);

   // Cache the person along with the id so that future requests can
   // retrieve it from memory rather than going to the data store
   cache.putForExternalRead(id, person);
} else {
```

```
    // The person was found in the cache, so return it to the application
    return cachedPerson;
}
```

putForExternalRead は、アプリケーションの実行元 (Person のアドレスを変更するトランザクションからなど) となる新しい Person インスタンスでキャッシュを更新するメカニズムとして使用しないでください。キャッシュされた値を更新する場合は、標準の put 操作を使用してください。使用しないと、破損したデータをキャッシュする可能性が高くなります。

3.5.3.1. AdvancedCache API

簡単な Cache インターフェイスの他に、Data Grid はエクステンション作成者向けに AdvancedCache インターフェイスを提供します。AdvancedCache は、特定の内部コンポーネントにアクセスし、フラグを適用して特定のキャッシュメソッドのデフォルト動作を変更する機能を提供します。次のコードスニペットは、AdvancedCache を取得する方法を示しています。

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

3.5.3.1.1. フラグ

フラグは通常のキャッシュメソッドに適用され、特定のメソッドの動作を変更します。利用可能なフラグの一覧と、その効果については、Flag 列挙を参照してください。フラグは、AdvancedCache.withFlags() を使用して適用されます。このビルダーメソッドを使用して、キャッシュ呼び出しに任意の数のフラグを適用できます。次に例を示します。

```
advancedCache.withFlags(Flag.CACHE_MODE_LOCAL, Flag.SKIP_LOCKING)
   .withFlags(Flag.FORCE_SYNCHRONOUS)
   .put("hello", "world");
```

3.5.3.2. Asynchronous API

Cache.put()、Cache.remove() などの同期 API メソッドの他に、Data Grid には非同期のノンブロッキング API も含まれ、同じ結果をノンブロッキング方式で達成できます。

これらのメソッドの名前は、ブロックメソッドと同様の名前が付けられ、"Async" が追加されます。例: Cache.putAsync()、Cache.removeAsync() など。 これらの非同期のメソッドは、操作の実際の結果が含まれる CompletableFuture を返します。

たとえば、**Cache<String, String>** としてパラメーター化されたキャッシュでは、**Cache.put(String key, String value)** は **String** を返し、**Cache.putAsync(String key, String value)** は **CompletableFuture<String>** を返します。

3.5.3.2.1. このような API を使用する理由

ノンブロッキング API は、通信の失敗や例外を処理する機能を備えており、同期通信の保証をすべて提供するという点で強力なもので、呼び出しが完了するまでブロックする必要がありません。 これにより、システムで並列処理をより有効に活用できます。 以下に例を示します。

```
Set<CompletableFuture<?>> futures = new HashSet<>();
futures.add(cache.putAsync(key1, value1)); // does not block
futures.add(cache.putAsync(key2, value2)); // does not block
futures.add(cache.putAsync(key3, value3)); // does not block

// the remote calls for the 3 puts will effectively be executed
```

```
// in parallel, particularly useful if running in distributed mode
// and the 3 keys would typically be pushed to 3 different nodes
// in the cluster

// check that the puts completed successfully
for (CompletableFuture<?> f: futures) f.get();
```

3.5.3.2.2. 実際に非同期で発生するプロセス

Data Grid には、通常の書き込み操作の重要なパスにあると見なされる 4 つの項目があります。これらの項目をコスト順に示します。

- ネットワークコール
- マーシャリング
- キャッシュストアへの書き込み (オプション)
- ロック

非同期メソッドを使用すると、ネットワーク呼び出しとマーシャリングがクリティカルパスから除外されます。 ただし、さまざまな技術的な理由により、キャッシュストアへの書き込みとロックの取得は、呼び出し元のスレッドで引き続き発生します。

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 メトリックの設定

Data Grid は、あらゆる監視システム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。

注記

Data Grid メトリックは、**vendor** スコープで提供されます。JVM に関連するメトリックは **base** スコープで提供されます。

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ーに追加します。
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。

メトリックの設定

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"
```

関連情報

- Micrometer Prometheus

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ーに追加し、**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 Server セキュリティーレルムの 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ルムを使用して JMX 固有の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。Data Grid は、グローバル **ADMIN** および **MONITOR** 権限を JMX **controlRole** および **monitorRole** ロールに自動的にマップします。

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 Server 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ーを Data Grid Server 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

- セキュリティーレルムの作成

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 の詳細なリストおよび説明、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 Cache Manager 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

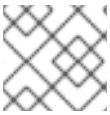
```
infinispan:
  cacheContainer:
    statistics: "true"
    jmx:
      enabled: "true"
      domain: "example.com"
      mbeanServerLookup: "com.example.MyMBeanServerLookup"
```

4.3. 状態遷移操作中のメトリクスのエクスポート

Data Grid がノード間で再配布するクラスター化されたキャッシュのタイムメトリクスをエクスポートできます。

ノードがクラスターに参加したりクラスターから離れたりするなど、クラスター化されたキャッシュトポロジーが変更されると、状態遷移操作が発生します。状態遷移操作中に、Data Grid は各キャッシュからメトリクスをエクスポートするため、キャッシュのステータスを判断できます。状態遷移は、Data

Grid が各キャッシュからメトリクスをエクスポートできるように、属性をプロパティーとして公開します。



注記

インバリデーションモードでは状態遷移操作を実行できません。

Data Grid は、REST API および JMX API と互換性のある時間メトリクスを生成します。

前提条件

- Data Grid メトリクスを設定します。
- 埋め込みキャッシュやリモートキャッシュなど、キャッシュタイプのメトリクスを有効にします。
- クラスター化されたキャッシュトポロジーを変更して、状態遷移操作を開始します。

手順

- 以下の方法のいずれかを選択します。
  - REST API を使用してメトリクスを収集するように Data Grid を設定します。
  - JMX API を使用してメトリクスを収集するように Data Grid を設定します。

関連情報

- Data Grid 統計と JMX モニタリング (Data Grid キャッシュ) の有効化と設定
- **StateTransferManager** (Data Grid 14.0 API)

4.4. クロスサイトレプリケーションのステータスの監視

バックアップの場所のサイトステータスを監視して、サイト間の通信の中断を検出します。リモートサイトのステータスが **offline** に変わると、Data Grid はバックアップの場所へのデータのレプリケーションを停止します。データが同期しなくなるため、クラスターをオンラインに戻す前に不整合を修正する必要があります。

問題を早期に検出するには、クロスサイトのイベントの監視が必要です。以下の監視ストラテジーのいずれかを使用します。

- REST API を使用したクロスサイトレプリケーションの監視
- Prometheus メトリクスまたはその他のモニタリングシステムを使用したクロスサイトレプリケーションの監視

REST API を使用したクロスサイトレプリケーションの監視

REST エンドポイントを使用して、すべてのキャッシュのクロスサイトレプリケーションのステータスを監視します。カスタムスクリプトを実装して REST エンドポイントをポーリングするか、以下の例を使用できます。

前提条件

- クロスサイトレプリケーションを有効にする。

手順

1. REST エンドポイントをポーリングするスクリプトを実装します。
   以下の例は、Python スクリプトを使用して、5 秒ごとにサイトのステータスをポーリングする方法を示しています。

```
#!/usr/bin/python3
import time
import requests
from requests.auth import HTTPDigestAuth


class InfinispanConnection:

    def __init__(self, server: str = 'http://localhost:11222', cache_manager: str = 'default',
                 auth: tuple = ('admin', 'change_me')) -> None:
        super().__init__()
        self.__url = f'{server}/rest/v2/container/x-site/backups/'
        self.__auth = auth
        self.__headers = {
            'accept': 'application/json'
        }

    def get_sites_status(self):
        try:
            rsp = requests.get(self.__url, headers=self.__headers, auth=HTTPDigestAuth(self.__auth[0],
self.__auth[1]))
            if rsp.status_code != 200:
                return None
            return rsp.json()
        except:
            return None


# Specify credentials for Data Grid user with permission to access the REST endpoint
USERNAME = 'admin'
PASSWORD = 'change_me'
# Set an interval between cross-site status checks
POLL_INTERVAL_SEC = 5
# Provide a list of servers
SERVERS = [
    InfinispanConnection('http://127.0.0.1:11222', auth=(USERNAME, PASSWORD)),
    InfinispanConnection('http://127.0.0.1:12222', auth=(USERNAME, PASSWORD))
]
#Specify the names of remote sites
REMOTE_SITES = [
    'nyc'
]
#Provide a list of caches to monitor
CACHES = [
    'work',
    'sessions'
]


def on_event(site: str, cache: str, old_status: str, new_status: str):
```

```
    # TODO implement your handling code here
    print(f'site={site} cache={cache} Status changed {old_status} -> {new_status}')


def __handle_mixed_state(state: dict, site: str, site_status: dict):
    if site not in state:
        state[site] = {c: 'online' if c in site_status['online'] else 'offline' for c in CACHES}
        return

    for cache in CACHES:
        __update_cache_state(state, site, cache, 'online' if cache in site_status['online'] else 'offline')


def __handle_online_or_offline_state(state: dict, site: str, new_status: str):
    if site not in state:
        state[site] = {c: new_status for c in CACHES}
        return

    for cache in CACHES:
        __update_cache_state(state, site, cache, new_status)


def __update_cache_state(state: dict, site: str, cache: str, new_status: str):
    old_status = state[site].get(cache)
    if old_status != new_status:
        on_event(site, cache, old_status, new_status)
        state[site][cache] = new_status


def update_state(state: dict):
    rsp = None
    for conn in SERVERS:
        rsp = conn.get_sites_status()
        if rsp:
            break
    if rsp is None:
        print('Unable to fetch site status from any server')
        return

    for site in REMOTE_SITES:
        site_status = rsp.get(site, {})
        new_status = site_status.get('status')
        if new_status == 'mixed':
            __handle_mixed_state(state, site, site_status)
        else:
            __handle_online_or_offline_state(state, site, new_status)


if __name__ == '__main__':
    _state = {}
    while True:
        update_state(_state)
        time.sleep(POLL_INTERVAL_SEC)
```

サイトのステータスが **online** から **offline** に変わったり、その逆の場合、関数 **on_event** が呼び出されます。

このスクリプトを使用する場合は、以下の変数を指定する必要があります。

- **USERNAME** および **PASSWORD**: REST エンドポイントにアクセスする権限を持つ Data Grid ユーザーのユーザー名およびパスワード。
- **POLL_INTERVAL_SEC**: ポーリング間の秒数。
- **SERVERS**: このサイトの Data Grid Server の一覧。このスクリプトには必要なのは、有効な応答が 1 つだけですが、フェイルオーバーを許可するためにリストが提供されています。
- **REMOTE_SITES**: これらのサーバー上で監視するリモートサイトのリスト。
- **CACHES** : 監視するキャッシュ名のリスト。

関連情報

- REST API: バックアップの場所のステータスの取得

Prometheus メトリクスを使用したクロスサイトレプリケーションの監視

Prometheus およびその他の監視システムを使用すると、サイトのステータスが **offline** に変化したことを検出するアラートを設定できます。

ヒント

クロスサイトレイテンシーメトリックを監視すると、発生する可能性のある問題を検出しやすくなります。

前提条件

- クロスサイトレプリケーションを有効にする。

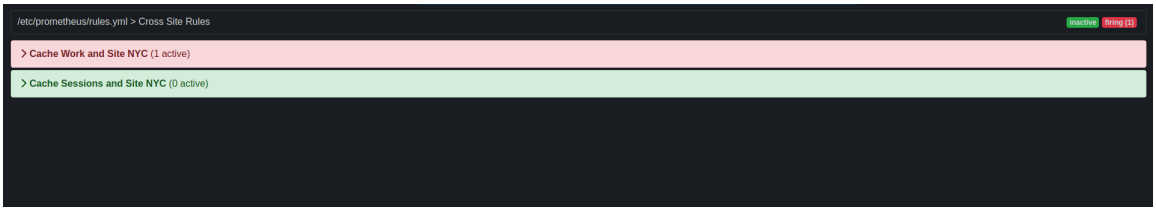
手順

1. Data Grid メトリクスを設定します。
2. Prometheus メトリクス形式を使用してアラートルールを設定します。
   - サイトの状態については、**online** の場合は **1** を、**offline** の場合は **0** を使用します。
   - **expr** フィールドには、**vendor_cache_manager_default_cache_<cache name>_x_site_admin_<site name>_status** の形式を使用します。
     以下の例では、Prometheus は、**NYC** サイトが **work** または **sessions** という名前のキャッシュに対して **オフライン** になると警告を表示します。

```
groups:
- name: Cross Site Rules
  rules:
  - alert: Cache Work and Site NYC
    expr: vendor_cache_manager_default_cache_work_x_site_admin_nyc_status == 0
  - alert: Cache Sessions and Site NYC
    expr: vendor_cache_manager_default_cache_sessions_x_site_admin_nyc_status ==
0
```

次の図は、NYC サイトがキャッシュ **work** を行うために **offline** であるという警告を示しています。

図4.1 Prometheus 警告



関連情報

- Data Grid メトリックの設定
- Prometheus Alerting Overview
- Grafana Alerting Documentation
- Openshift Managing Alerts

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ー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、コンテナー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 がデータコンテナー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。

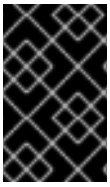
✓ エントリーを手動でエビクトし、有効期限を手動でトリガーできます。

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、データコンテナー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ーから破棄し、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 はデータコンテナー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 はデータコンテナー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

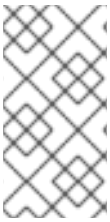
Data Grid エビクションを設定する場合は、以下を指定します。

- データコンテナーの最大サイズ。
- 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 Least Recently 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、測定の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);
```

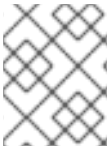
関連情報

- 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、データコンテナー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** 設定プロパティーの値を使用します。
- 最大アイドルは、**last used** タイムスタンプと **max-idle** 設定プロパティーの値を使用します。

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、データコンテナーから削除します。**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、定期的に実行さ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します。

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実行する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。

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

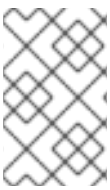
```
replicatedCache:
  expiration:
    lifespan: "5000"
    maxIdle: "1000"
```

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

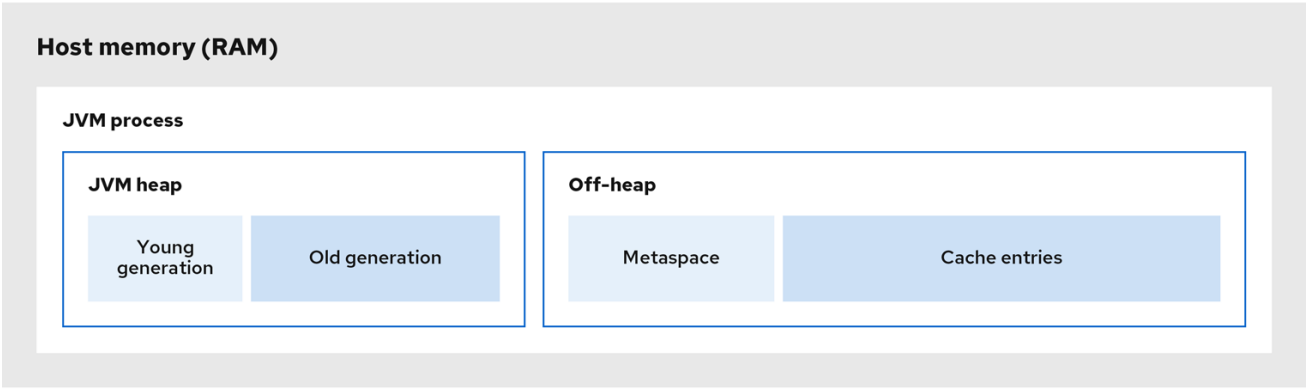
- org.infinispan.configuration.cache.ExpirationConfigurationBuilder
- org.infinispan.expiration.ExpirationManager

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



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、状態遷移操作は、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、データコンテナーに 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" : "OFF_HEAP",
      "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、再起動後も存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

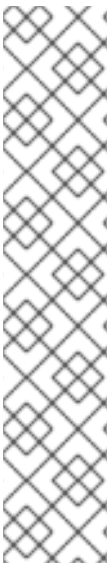
エビクションとパッシベーションの手法を使用すると、Data Grid は頻繁に使用されるデータのみをメモリー内に保持し、古いエントリーを永続ストレージに書き込みます。

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。そのため、パッシベーションにより、コストがかかる可能性のある不必要な永続ストレージへの書き込みが防止されます。

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と、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 は、エビクトされたエントリーへのアクセス要求を受け取ると、エントリーをキャッシュストアからメモリーに遅延ロードし、値をストア内に保持したまま、エントリーがアクティブ化されたことを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

パッシベーションが無効

メモリーのデータへの書き込みにより、永続ストレージが書き込まれます。

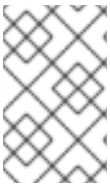
Data Grid がデータをメモリーからエビクトする場合、永続ストレージのデータにはメモリーからエビクトされるエントリーが含まれます。このようにして、永続ストレージはインメモリーキャッシュのスーパーセットになります。クラッシュ後もストアを再度読み取ることができるため、この設定は最大限の整合性が必要な場合に推奨されます。

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

パッシベーションが有効

Data Grid は、メモリーからデータをエビクトとするとき、エントリーが削除されたとき、またはノードをシャットダウンする場合にのみ、永続ストレージにデータを追加します。

Data Grid がエントリーをアクティブ化すると、メモリー内のデータが復元されますが、ストア内のデータはそのまま保持されます。これにより、ストアがない場合と同じくらい書き込みが高速になり、整合性も維持されます。エントリーが作成または更新されると、メモリー内のみが更新されるため、ストアが当面の間古い状態になります。



注記

ストアも共有として設定されている場合、パッシベーションはサポートされません。これは、書き込みがエビクトされるタイミングと読み取りがエビクトされるタイミングによっては、エントリーがノード間で同期されなくなる可能性があるためです。

データの整合性を確保するには、共有されていないストアで常に **purgeOnStartup** を有効にする必要があります。これは、パッシベーションが有効または無効な場合の両方に当てはまります。ストアがダウンしている間に古いエントリーを保持し、後でそれを復活させることができるためです。

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	パッシベーションが無効	パッシベーションが有効
k1 を挿入します。	**Memory:** k1 **Disk:** k1	**Memory:** k1 **Disk:** -
k2 を挿入します。	**Memory:** k1, k2 **Disk:** k1, k2	**Memory:** k1, k2 **Disk:** -
エビクションスレッドが実行され、k1 をエビクトします。	**Memory:** k2 **Disk:** k1, k2	**Memory:** k2 **Disk:** k1
k1 の読み取り	**Memory:** k1, k2 **Disk:** k1, k2	**Memory:** k1, k2 **Disk:** 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。	**Memory:** k1 **Disk:** k1, k2	**Memory:** k1 **Disk:** k1, k2
k2 を削除します。	**Memory:** k1 **Disk:** k1	**Memory:** k1 **Disk:** 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ーを更新すると、**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>
      <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:14.0"
                      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と、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イ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。

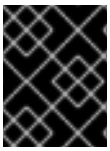
セグメント化されたキャッシュストアを追加した後に設定の **numSegments** パラメーターを変更すると、Data Grid はそのキャッシュストアからデータを読み取ることができません。

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では、永続ストレージから古いエントリーが読み込まれないように、"purge on startup" を使用する必要があります。

- 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

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"
         purge="true"/>
</persistence>
```

共有キャッシュストア

```
<persistence>
  <store shared="true"
         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。

Data Grid はファイルロックを使用して、グローバル永続化ロケーションへの同時アクセスを阻止します。ロックは起動時に取得され、ノードのシャットダウン時に解放されます。ダングリングロックファイルが存在する場合、クラッシュまたは外部終了が原因で、ノードが正常にシャットダウンされなかったことを示します。デフォルト設定では、データ破損を回避するために、Data Grid は次のメッセージを表示して起動を拒否します。

```
ISPN000693: Dangling lock file '%s' in persistent global state, probably left behind by an unclean shutdown
```

グローバル状態の **unclean-shutdown-action** 設定を次のいずれかに設定することで、この動作を変更できます。

- **FAIL**: 永続的なグローバル状態でダングリングロックファイルが見つかった場合、Cache Manager の起動を阻止します。これがデフォルトの動作です。
- **PURGE**: 永続的なグローバル状態にダングリングロックファイルが見つかった場合、永続グローバル状態をクリアします。
- **IGNORE**: 永続的なグローバル状態におけるダングリングロックファイルの存在を無視します。

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。

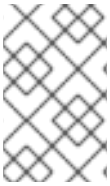
クラスター化された組み込みキャッシュ (レプリケートまたは分散など) の場合は、クラスタートポロジーを復元するためにグローバルの永続的な場所を常に有効にし、設定する必要があります。

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ーです。

   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。

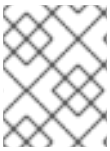
また、共有ファイルシステムでトランザクションキャッシュを使用しようとすると、コミットフェーズでファイルに書き込む際に修復不能な障害が発生する可能性があります。

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。



注記

クラスター化されたキャッシュで SoftIndexFileStore を使用すると、起動時のパージが有効になり、古いエントリーの復活を防ぐことができます。

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。

SoftIndexFileStore は、設定されたキャッシュセグメントごとに B+ ツリーを作成します。これにより、要素の数が限られるため追加の「インデックス」が提供され、インデックス更新の追加の並列処理が実現します。現在は、キャッシュセグメント数の 16 分の 1 に基づいて並列処理の数が許可されています。

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。

ファイル制限

SoftIndexFileStore は、特定の時点で、2 個のファイルと設定された openFilesLimit 個のファイルを使用します。追加の 2 つのファイルポインターのうち、1 つは新しく更新されたデータ用のログアペンダー用に予約されています。もう 1 つは、圧縮されたエントリーを新しいファイルに書き込むコンパクター用に予約されています。

インデックス作成用に割り当てられた開いているファイルの数は、設定された openFilesLimit の合計数の 10 分の 1 です。この数値の最小値は、1 かキャッシュセグメントの数です。設定された制限から引いて残った数は、開いているデータファイル自体に割り当てられます。

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。追加ログは直接セグメント化されず、セグメント化はインデックスによって直接処理されます。

有効期限

SoftIndexFileStore は、期限切れのエントリーとその要件を完全にサポートしています。

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。このファイルで使用可能な領域は、エントリーに収まることができる場合に限り使用されます。同様に、メモリーからすべてのエントリーを削除すると、単一のファイルストアのサイズが減少したり、デフラグしたりしません。

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"
```

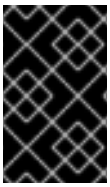
ConfigurationBuilder

```
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"/>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence" : {
      "passivation" : true,
      "single-file-store" : {
```

```
        "shared" : false,
        "preload" : true
      }
    }
  }
}
```

YAML

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

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().passivation(true)
       .addStore(SingleFileStoreConfigurationBuilder.class)
          .shared(false)
          .preload(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. マネージドデータソースの設定

Data Grid Server 設定の一部としてマネージド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、キャッシュ内のマネージドデータソースの JDNI 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。

  ヒント

  Data Grid コマンドラインインターフェイス (CLI) で **install** コマンドを使用して、必要なドライバーを **server/lib** ディレクトリーにダウンロードします。以下に例を示します。

  ```
  install org.postgresql:postgresql:42.4.3
  ```

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ーで定義します。
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リスト表示し、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
```

マネージドデータソースの設定

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <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

マネージドデータソースを 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

- マネージド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. マネージドデータソースの 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ー

Data Grid Server 設定で、マネージドデータソースの JDBC 接続プールを調整できます。

プロパティー	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。
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 プロパティーを使用した JDBC 接続プールの設定

プロパティー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.*** プロパティーで 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ー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ト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。

警告

SQL キャッシュストアは、既存のデータベーステーブルで使用することを目的としています。したがって、有効期限、セグメント、バージョン管理メタデータなどのメタデータは保存されません。バージョンストレージがないため、SQL ストアは楽観的トランザクションキャッシュと非同期クロスサイトレプリケーションをサポートしません。この制限は、Hot Rod のバージョン管理された操作にも適用されます。

ヒント

SQL キャッシュストアが読み取り専用として設定されている場合は、SQL キャッシュストアとともに有効期限を使用します。有効期限により、古い値がメモリーから削除され、キャッシュがデータベースから値を再度取得し、新たにキャッシュします。

関連情報

- DatabaseType Enum サポートされているデータベースのダイアレクトのリスト
- 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、マネージドデータソースを 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 プロセッサーで 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** ディレクトリーにコピーします。

     ヒント

     Data Grid コマンドラインインターフェイス (CLI) で **install** コマンドを使用して、必要なドライバーを **server/lib** ディレクトリーにダウンロードします。以下に例を示します。

     ```
     install org.postgresql:postgresql:42.4.3
     ```

   - 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

   ```
   table-jdbc-store xmlns="urn:infinispan:config:store:sql:14.0"
   ```

   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。

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

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:14.0"
                      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、マネージドデータソースを 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 プロセッサーで 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** ディレクトリーにコピーします。

ヒント

Data Grid コマンドラインインターフェイス (CLI) で **install** コマンドを使用して、必要なドライバーを **server/lib** ディレクトリーにダウンロードします。以下に例を示します。

```
install org.postgresql:postgresql:42.4.3
```

- 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>
```

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

宣言的

```
query-jdbc-store xmlns="urn:infinispan:config:store:sql:14.0"
```

プログラマティック

```
persistence().addStore(QueriesJdbcStoreConfigurationBuilder.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。
6. 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。

クエリーステートメント	説明
SIZE	キャッシュ内のエントリー数をカウントします。
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
次の例は、"person" テーブルと "address" テーブルの SQL データ定義言語 (DDL) ステートメントを示しています。例で説明されているデータ型は、PostgreSQL データベースに対してのみ有効です。

person テーブルの SQL ステートメント

```
CREATE TABLE Person (
  name VARCHAR(255) NOT NULL,
  picture BYTEA,
  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。

"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];
}
```

"person" テーブルの Protobuf スキーマ

```
package com.example;
```

```
import "/path/to/address.proto";

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];
}
```

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:sql:14.0"
                      dialect="POSTGRES"
                      shared="true" key-columns="name">
          <connection-pool driver="org.postgresql.Driver"
                           connection-url="jdbc:postgresql://localhost:5432/postgres"
                           username="postgres"
                           password="changeme"/>
      <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 = :name AND t2.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"/>
    </query-jdbc-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
```

```
  "distributed-cache": {
    "persistence": {
      "query-jdbc-store": {
        "dialect": "POSTGRES",
        "shared": "true",
        "key-columns": "name",
        "connection-pool": {
          "username": "postgres",
          "password": "changeme",
          "driver": "org.postgresql.Driver",
          "connection-url": "jdbc:postgresql://localhost:5432/postgres"
        },
        "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 = :name AND t2.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"
      connectionPool:
        username: "postgres"
        password: "changeme"
        driver: "org.postgresql.Driver"
        connectionUrl: "jdbc:postgresql://localhost:5432/postgres"
      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 = :name AND t2.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 = :name AND t2.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。

```
ISPN008064: 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ーを保存するために使用されるデータテーブルの他に、ストアはメタデータを保存するための **_META** テーブルも作成します。この表は、既存のデータベースコンテンツが現在の Data Grid バージョンおよび設定と互換性があることを確認するために使用されます。

注記

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

セグメンテーション

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

関連情報

- 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

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:14.0"
     ```

   - 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ア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ーを随時設定します。
   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ーを保存するデータベーステーブルを追加します。

重要

不適切なデータ型を指定して **string-keyed-jdbc-store** を設定すると、キャッシュエントリーのロードまたは保存中に例外が発生する可能性があります。詳細と、Data Grid リリースの一部としてテストされているデータ型の一覧は、Tested database settings for Data Grid string-keyed-jdbc-store persistence (Login required) を参照してください。

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:14.0"
                             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、データをメモリーに主要なキャッシュストアを提供します。他のデータベースには、Data Grid がメモリーから失効するエントリーを保持します。

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**

設定プロパティー

任意で、以下のように設定でプロパティーを設定します。

- RocksDB データベースを調整およびチューニングするために、プロパティーの前に **database** 接頭辞を追加します。
- プロパティーの前に **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:kSnappyCompression: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:14.0"
                   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("raytsang", new User(...));
```

プロパティーを持つ 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めません。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。

共有リモートキャッシュコンテナー

各リモートキャッシュストアは専用のリモート Cache Manager を作成します。複数のリモートストアが同じサーバーに接続している場合、リソースが浪費されます。**remote-cache-containers** 設定を使用して共有リモート Cache Manager を作成し、**remote-store** 定義内で名前で参照します。

ヒント

remote-cache-container が 1 つだけ存在する場合、明示的に指定されていないすべてのリモートキャッシュストアによってデフォルトで使用されます。

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。

共有リモートコンテナーを使用したリモートキャッシュストアの設定

XML

```
<infinispan>

   <remote-cache-containers>
      <remote-cache-container uri="hotrod://one,two:12111?max-active=10&amp;exhausted-
action=CREATE_NEW"/>
   </remote-cache-containers>

   <cache-container>
      <distributed-cache>
         <persistence>
            <remote-store xmlns="urn:infinispan:config:store:remote:14.0"
                          cache="mycache"
                          raw-values="true"
            />
         </persistence>
      </distributed-cache>
   </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan": {
    "remote-cache-containers": [
      {
        "uri": "hotrod://one,two:12111?max-active=10&exhausted-action=CREATE_NEW"
      }
    ],
    "cache-container": {
      "caches": {
        "mycache": {
          "distributed-cache": {
```

```
            "remote-store": {
              "cache": "mycache",
              "raw-values": "true"
            }
          }
        }
      }
    }
  }
}
```

YAML

```
infinispan:
  remoteCacheContainers:
    - uri: "hotrod://one,two:12111?max-active=10&exhausted-action=CREATE_NEW"
  cacheContainer:
    caches:
      mycache:
        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)
      .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();
```

プライベートリモートコンテナーを使用した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:14.0"
                 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)
      .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. 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**)。
- 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

6.15. 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。

6.15.1. 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。

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

6.15.2. 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。

6.15.3. 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();
```

6.15.4. 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** ディレクトリーに追加します。

6.16. 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ーを提供します。

6.16.1. 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 CLI **migrate store** コマンドを提供します。

ストアマイグレーターは、以前のバージョンの Data Grid のキャッシュストアをソースとして取得し、キャッシュストアの実装をターゲットとして使用します。

ストアマイグレーターを実行すると、**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。ストアマイグレーターは、ソースストアからメモリーにエントリーをロードし、それらをターゲットキャッシュに配置します。

ストアマイグレーターを使用すると、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 文字列ベースのキャッシュストアから、SIFS キャッシュストアに移行できます。

重要

ストアマイグレーターは、セグメント化されたキャッシュストアから以下へは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。

6.16.2. キャッシュストアマイグレーターの設定

migrator.properties ファイルを使用して、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ーを設定します。

手順

1. **migrator.properties** ファイルを作成します。
2. **migrator.properties** ファイルを使用して、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ーを設定します。
   a. **source.** 接頭辞をソースキャッシュストアの全設定プロパティーに追加します。

ソースキャッシュストアの例

```
source.type=SOFT_INDEX_FILE_STORE
source.cache_name=myCache
source.location=/path/to/source/sifs
source.version=<version>
```

重要

セグメント化されたキャッシュストアからデータを移行するには、**source.segment_count** プロパティーを使用してセグメント数も設定する必要があります。セグメント数は、Data Grid 設定の **clustering.hash.numSegments** と一致させる必要があります。キャッシュストアのセグメント数が対応するキャッシュのセグメント数と一致しないと、Data Grid が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。

b. **target.** 接頭辞をターゲットキャッシュストアの全設定プロパティーに追加します。

ターゲットキャッシュストアの例

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

6.16.2.1. キャッシュストアマイグレーターの設定プロパティー

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

表6.2 キャッシュストアタイププロパティー

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

表6.3 一般的なプロパティー

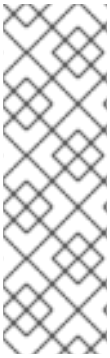
プロパティー	説明	値の例	必須/オプション
cache_name	バックアップするキャッシュの名前。	**.cache_name=myCache**	必須

プロパティー	説明	値の例	必須/オプション
segment_count	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。 セグメント数は、Data Grid 設定の **clustering.hash.numSegments** と一致させる必要があります。キャッシュストアのセグメント数が対応するキャッシュのセグメント数と一致しないと、Data Grid が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。	**.segment_count=256**	任意
marshaller.class	カスタムマーシャラ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.allow-list.classes**
デシリアライズが許可される完全修飾クラス名のコンマ区切りリストを指定します。	任意	**marshaller.allow-list.regexps**	どのクラスのデシリアライズを許可するかを決定する正規表現のコンマ区切りリストを指定します。
任意	**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。	任意

表6.4 JDBC プロパティー

プロパティー	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

プロパティー	説明	必須/オプション
version	ソースキャッシュストアのマーシャラーバージョンを指定します。 以下のいずれかの値を設定します。 * Data Grid 7.2.x の場合は **8** * Data Grid 7.3.x の場合は **9** * Data Grid 8.0.x の場合は **10** * Data Grid 8.1.x の場合は **11** * Data Grid 8.2.x の場合は **12** * Data Grid 8.3.x の場合は **13**	ソースストアにのみ必要です。
connection_pool.connection_url	JDBC 接続 URL を指定します。	必須
connection_pool.driver_class	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.disable_upsert	データベース upsert を無効にします。	任意
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。	任意
table.string.table_name_prefix	テーブル名の追加接頭辞を指定します。	任意
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。	任意



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、以下のプロパティーで **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.infinispan.commons.marshall.JavaSerializationMarshaller
target.marshaller.allow-list.classes=org.example.Person,org.example.Animal
target.marshaller.allow-list.regexps="org.another.example.*"
target.marshaller.externalizers=25:Externalizer1,org.example.Externalizer2
target.connection_pool.connection_url=jdbc:postgresql:postgres
target.connection_pool.driver_class=org.postrgesql.Driver
target.connection_pool.username=postgres
target.connection_pool.password=redhat
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 プロパティー

プロパティー	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。	任意

```
# 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 プロパティー

プロパティー	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 プロパティー

プロパティー	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.location=path/to/sifs/index
```

6.16.3. Data Grid キャッシュストアの移行

ストアマイグレーターを実行して、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

- Data Grid CLI を入手している。
- 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** ファイルを作成している。

手順

- **migrate store -p /path/to/migrator.properties** CLI コマンドを実行します。

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[2|3].timeout + FD_ALL[2|3].interval + VERIFY_SUSPECT[2].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。

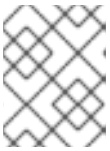
- 1 つ以上のセグメントですべての所有者が失われている。
  これは、分散キャッシュの所有者の数と同じか、それ以上の数のノードがクラスターを離れている場合に生じます。

- 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ー (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ー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。

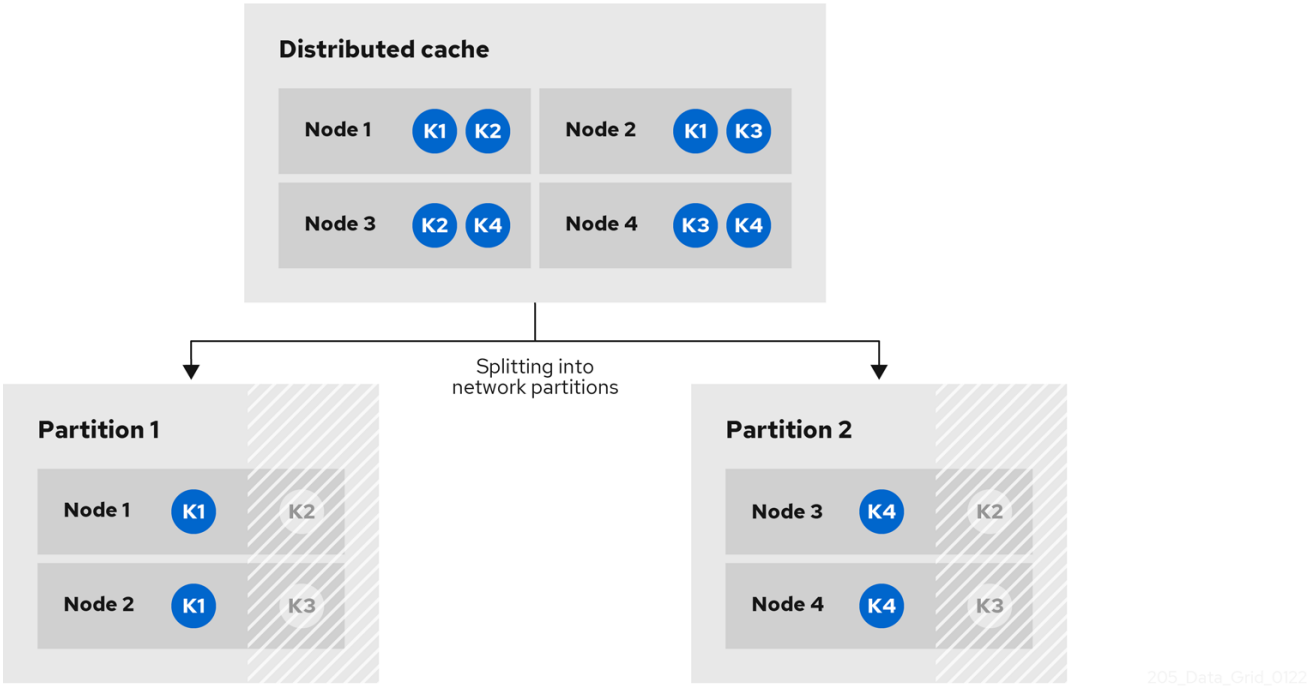
7.2.1. 低下したキャッシュのリカバリー例

このトピックでは、Data Grid が **DENY_READ_WRITES** パーティション処理ストラテジー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。

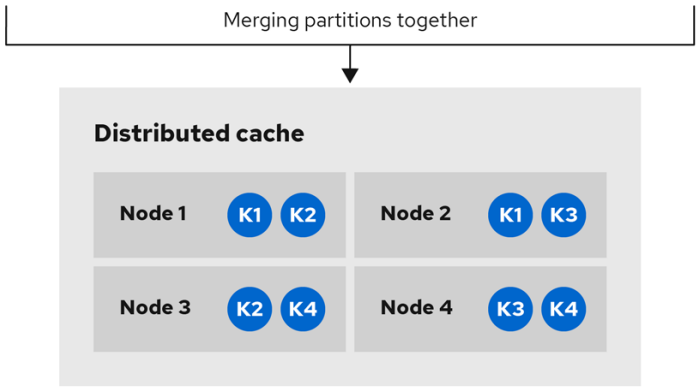
たとえば、Data Grid クラスターには 4 つのノードがあり、各エントリーに対してレプリカが 2 つある分散キャッシュが含まれています (**owners=2**)。キャッシュには、**k1**、**k2**、**k3**、および **k4** の 4 つのエントリーがあります。

DENY_READ_WRITES ストラテジーでは、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。



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。



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/container/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ム的に取得します。

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。

- Data Grid コンソールを開き、**Make available** にするオプションを選択します。
- REST API でリモートキャッシュの可用性を変更します。

```
POST /rest/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ム的に変更します。

```
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

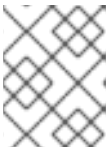
7.3. パーティション処理の設定

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定されます。

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。

ヒント

install コマンドを Data Grid コマンドラインインターフェイス (CLI) で使用して、JAR を **server/lib** ディレクトリーにダウンロードします。

2. Data Grid 設定を開いて編集します。
3. 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。
4. 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。
5. 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

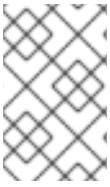
7.7. 組み込みキャッシュでのパーティションの手動マージ

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 ロールベースアクセス制御によるセキュリティー承認

ロールベースアクセス制御 (RBAC) 機能では、さまざまなパーミッションレベルを使用して、Data Grid とのユーザーの対話を制限します。

注記

リモートキャッシュまたは組み込みキャッシュに固有のユーザーの作成と承認の設定に関する詳細は、以下を参照してください。

- Configuring user roles and permissions with Data Grid Server
- Programmatically configuring user roles and permissions

8.1. DATA GRID のユーザーロールと権限

Data Grid には、キャッシュや Data Grid のリソースにアクセスするための権限をユーザーに提供するロールがいくつかあります。

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。

関連情報

- org.infinispan.security.AuthorizationPermission Enum
- Data Grid 設定スキーマ参照

8.1.1. パーミッション

ユーザーロールは、さまざまなアクセスレベルを持つパーミッションのセットです。

表8.1 Cache Manager のパーミッション

Permission	機能	説明
設定	**defineConfiguration**	新しいキャッシュ設定を定義します。
LISTEN	**addListener**	Cache Manager に対してリスナーを登録します。
ライフサイクル	**stop**	Cache Manager 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ー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 Cache Manager のアクセス許可が含まれます。

表8.2 キャッシュ権限

Permission	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。

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

8.1.2. ロールとパーミッションマッパー

Data Grid は、ユーザーをプリンシパルのコレクションとして実装します。プリンシパルは、ユーザー名などの個々のユーザー ID、またはユーザーが属するグループのいずれかを表します。内部的には、これらは **javax.security.auth.Subject** クラスで実装されます。

承認を有効にするには、プリンシパルをロール名にマップし、その後、ロール名を一連のパーミッションに展開する必要があります。

Data Grid には、セキュリティープリンシパルをロールに関連付ける **PrincipalRoleMapper** API と、ロールを特定のパーミッションに関連付ける **RolePermissionMapper** API が含まれています。

Data Grid は以下のロールおよびパーミッションのマッパーの実装を提供します。

クラスターロールマッパー

クラスターレジストリーにロールマッピングのプリンシパルを保存します。

クラスターパーミッションマッパー

ロールとパーミッションのマッピングをクラスターレジストリーに保存します。ユーザーのロールとパーミッションを動的に変更できます。

ID ロールマッパー

ロール名としてプリンシパル名を使用します。プリンシパル名のタイプまたはフォーマットはソースに依存します。たとえば、LDAP ディレクトリーでは、プリンシパル名を識別名 (DN) にすることができます。

コモンネームロールマッパー

ロール名としてコモンネーム (CN) を使用します。このロールマッパーは、識別名 (DN) を含む LDAP ディレクトリーまたはクライアント証明書で使用できます。たとえば、**cn=managers,ou=people,dc=example,dc=com** は、**managers** ロールにマッピングされます。

注記

デフォルトでは、プリンシパルとロールのマッピングは、グループを表すプリンシパルにのみ適用されます。ユーザープリンシパルのマッピングも実行するように Data Grid を設定できます。

8.1.2.1. Data Grid でのロールとパーミッションへのユーザーのマッピング

LDAP サーバーから DN のコレクションとして取得された次のユーザーを考えてみましょう。

```
CN=myapplication,OU=applications,DC=mycompany
CN=dataprocessors,OU=groups,DC=mycompany
CN=finance,OU=groups,DC=mycompany
```

コモンネームロールマッパー を使用すると、ユーザーは次のロールにマッピングされます。

```
dataprocessors
finance
```

Data Grid には次のロール定義があります。

```
dataprocessors: ALL_WRITE ALL_READ
finance: LISTEN
```

ユーザーには次のパーミッションが与えられます。

```
ALL_WRITE ALL_READ LISTEN
```

関連情報

- Data Grid Security API
- org.infinispan.security.PrincipalRoleMapper
- org.infinispan.security.RolePermissionMapper
- org.infinispan.security.mappers.IdentityRoleMapper
- org.infinispan.security.mappers.CommonNameRoleMapper

8.1.3. ロールマッパーの設定

Data Grid は、デフォルトでクラスターロールマッパーとクラスターパーミッションマッパーを有効にします。ロールマッピングに別の実装を使用するには、ロールマッパーを設定する必要があります。

手順

1. Data Grid 設定を開いて編集します。
2. ロールマッパーを、Cache Manager 設定のセキュリティー許可の一部として宣言します。
3. 変更を設定に保存します。

ロールマッパーの設定

XML

```
<cache-container>
  <security>
    <authorization>
      <common-name-role-mapper />
    </authorization>
  </security>
</cache-container>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "security" : {
        "authorization" : {
          "common-name-role-mapper": {}
        }
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    security:
      authorization:
        commonNameRoleMapper: ~
```

関連情報

- Data Grid 設定スキーマ参照

8.1.4. クラスターのロールと権限マッパーの設定

クラスターロールマッパーは、プリンシパルとロール間の動的なマッピングを維持します。クラスター権限マッパーは、動的なロール定義セットを維持します。どちらの場合も、マッピングはクラスターレジストリーに保存され、ランタイム時に CLI または REST API を使用して操作できます。

前提条件

- Data Grid に **ADMIN** 権限がある。
- Data Grid CLI を起動している。
- 実行中の Data Grid クラスターに接続している。

8.1.4.1. 新しいロールの作成

新しいロールを作成し、権限を設定します。

手順

- **user roles create** コマンドを使用してロールを作成します。以下に例を示します。

```
user roles create --permissions=ALL_READ,ALL_WRITE simple
```

検証

user roles ls コマンドでユーザーに付与したロールを一覧表示します。

```
user roles ls
["observer","application","admin","monitor","simple","deployer"]
```

user roles describe コマンドを使用してロールを説明します。

```
user roles describe simple
{
  "name" : "simple",
  "permissions" : [ "ALL_READ","ALL_WRITE" ]
}
```

8.1.4.2. ユーザーへのロール付与

ユーザーにロールを割り当て、キャッシュ操作や Data Grid のリソースとのやり取りを行う権限を付与します。

ヒント

同じロールを複数のユーザーに割り当て、それらの権限を一元管理する場合は、ユーザーではなくグループにロールを付与します。

前提条件

- Data Grid に **ADMIN** 権限がある。
- Data Grid ユーザーを作成すること。

手順

1. Data Grid への CLI 接続を作成します。
2. **user roles grant** コマンドでユーザーにロールを割り当てます。以下に例を示します。

```
user roles grant --roles=deployer katie
```

検証

user roles ls コマンドでユーザーに付与したロールを一覧表示します。

```
user roles ls katie
["deployer"]
```

8.1.4.3. クラスターロールマッパー名リライター

デフォルトでは、プリンシパル名とロール間の厳密な文字列等価性を使用してマッピングが実行されます。ルックアップを実行する前に、プリンシパル名にトランスフォーメーションを適用するようにクラスターロールマッパーを設定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. Cache Manager 設定のセキュリティー認可の一環として、クラスターロールマッパーの名前リライターを指定します。
3. 変更を設定に保存します。

プリンシパル名の形式は、セキュリティーレルムの種類に応じて異なります。

- プロパティーとトークンレルムは、単純な文字列を返す場合があります
- 信頼と LDAP レルムは X.500 形式の識別名を返す場合があります
- Kerberos レルムは **user@domain** 形式の名前を返す場合があります

次のいずれかのトランスフォーマーを使用して、名前を共通形式に正規化できます。

8.1.4.3.1. ケースプリンシパルトランスフォーマー

XML

```
<cache-container>
  <security>
    <authorization>
      <cluster-role-mapper>
        <name-rewriter>
          <case-principal-transformer uppercase="false"/>
        </name-rewriter>
      </cluster-role-mapper>
    </authorization>
  </security>
</cache-container>
```

JSON

```
{
  "cache-container": {
    "security": {
      "authorization": {
        "cluster-role-mapper": {
          "name-rewriter": {
            "case-principal-transformer": {}
          }
        }
      }
    }
  }
}
```

YAML

```
cacheContainer:
  security:
    authorization:
      clusterRoleMapper:
        nameRewriter:
          casePrincipalTransformer:
            uppercase: false
```

8.1.4.3.2. Regex プリンシパルトランスフォーマー

XML

```
<cache-container>
  <security>
    <authorization>
      <cluster-role-mapper>
        <name-rewriter>
          <regex-principal-transformer pattern="cn=([^,]+),.*" replacement="$1"/>
        </name-rewriter>
      </cluster-role-mapper>
    </authorization>
  </security>
</cache-container>
```

JSON

```
{
  "cache-container": {
    "security": {
      "authorization": {
        "cluster-role-mapper": {
          "name-rewriter": {
            "regex-principal-transformer": {
```

```
            "pattern": "cn=([^,]+),.*",
            "replacement": "$1"
          }
        }
      }
    }
  }
}
```

YAML

```
cacheContainer:
  security:
    authorization:
      clusterRoleMapper:
        nameRewriter:
          regexPrincipalTransformer:
            pattern: "cn=([^,]+),.*"
            replacement: "$1"
```

関連情報

- Data Grid 設定スキーマ参照

8.2. セキュリティー承認によるキャッシュの設定

キャッシュにセキュリティー承認を追加し、ロールベースアクセス制御 (RBAC) を実施することができます。これには、Data Grid ユーザーが、キャッシュ操作を実行するのに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

- Data Grid のユーザーを作成し、ロールを付与するか、グループに割り当てている。

手順

1. Data Grid 設定を開いて編集します。
2. 設定に **security** セクションを追加します。
3. **authorization** 要素を使用して、ユーザーがキャッシュ操作を実行するために必要なロールを指定します。
   Cache Manager で定義されたすべてのロールを暗黙的に追加するか、ロールのサブセットを明示的に定義できます。
4. 変更を設定に保存します。

暗黙的なロール設定
次の設定では、Cache Manager で定義されたすべてのロールが暗黙的に追加されます。

XML

```
<distributed-cache>
  <security>
    <authorization/>
  </security>
</distributed-cache>
```

JSON

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

YAML

```
distributedCache:
  security:
    authorization:
      enabled: true
```

明示的なロール設定
次の設定では、Cache Manager で定義されたロールのサブセットを明示的に追加しています。この場合、Data Grid は、設定されたロールのいずれかを持たないユーザーのキャッシュ操作を拒否します。

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"]
```

第9章 トランザクションの設定

分散システム上に存在するデータは、一時的なネットワークの停止やシステム障害、あるいは単純なヒューマンエラーによって生じるエラーなどの影響を受けやすくなっています。このような外部要素については制御できませんが、データの品質に深刻な影響を与える可能性があります。データ破損の影響は、顧客満足度の低下、サービスが利用できなくなるほどのコストのかかるシステムの再構築など多岐にわたります。

Data Grid は、ACID (atomicity, consistency, isolation, durability) トランザクションを実行して、キャッシュの状態が一貫していることを確認できます。

9.1. トランザクション

Data Grid は、JTA 準拠のトランザクションを使用し、参加するように設定できます。

または、トランザクションのサポートが無効になっている場合は、JDBC 呼び出しで自動コミットを使用する場合と同等になります。ここでは、すべての変更後に変更がレプリケートされる可能性があります (レプリケーションが有効な場合)。

すべてのキャッシュ操作で Data Grid は以下を行います。

1. スレッドに関連する現在の トランザクション を取得します。
2. トランザクションのコミットまたはロールバック時に通知されるように、XAResource をトランザクションマネージャーに登録します (登録されていない場合)。

これを実行するには、キャッシュに環境の TransactionManager への参照を提供する必要があります。これは通常、TransactionManagerLookup インターフェイスの実装のクラス名を使用してキャッシュを設定することで行います。キャッシュが起動すると、このクラスのインスタンスを作成し、**TransactionManager** への参照を返す **getTransactionManager()** メソッドを呼び出します。

Data Grid には複数のトランザクションマネージャールックアップクラスが同梱されます。

トランザクションマネージャールックアップの実装

- EmbeddedTransactionManagerLookup: これは、他の実装が利用できない場合に、組み込みモードのみに使用する必要がある基本的なトランザクションマネージャーを提供します。この実装は、同時トランザクションおよびリカバリーでは、重大な制限があります。
- JBossStandaloneJTAManagerLookup: スタンドアロン環境、または JBoss AS 7 以前、および WildFly 8、9、10 で Data Grid を実行している場合、トランザクションマネージャーのデフォルトとしてこれを選択します。このトランザクションは、**EmbeddedTransactionManager** の不足をすべて解消する JBoss Transactions をベースとした本格的なトランザクションマネージャーです。
- WildflyTransactionManagerLookup: WildFly 11 以降で Data Grid を実行している場合は、トランザクションマネージャーのデフォルトとしてこれを選択します。
- GenericTransactionManagerLookup: これは、最も一般的な Java EE アプリケーションサーバーでトランザクションマネージャーを見つけるルックアップクラスです。トランザクションマネージャーが見つからない場合は、**EmbeddedTransactionManager** がデフォルトの設定になります。

初期化すると、**TransactionManager** は **Cache** 自体から取得することもできます。

```
//the cache must have a transactionManagerLookupClass defined
Cache cache = cacheManager.getCache();

//equivalent with calling TransactionManagerLookup.getTransactionManager();
TransactionManager tm = cache.getAdvancedCache().getTransactionManager();
```

9.1.1. トランザクションの設定

トランザクションはキャッシュレベルで設定されます。以下はトランザクションの動作に影響する設定と、各設定属性の簡単な説明になります。

```
<locking
  isolation="READ_COMMITTED"/>
<transaction
  locking="OPTIMISTIC"
  auto-commit="true"
  complete-timeout="60000"
  mode="NONE"
  notifications="true"
  reaper-interval="30000"
  recovery-cache="__recoveryInfoCacheName__"
  stop-timeout="30000"
  transaction-manager-
lookup="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"/>
```

プログラムを使用する場合

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.locking()
   .isolationLevel(IsolationLevel.READ_COMMITTED);
builder.transaction()
   .lockingMode(LockingMode.OPTIMISTIC)
   .autoCommit(true)
   .completedTxTimeout(60000)
   .transactionMode(TransactionMode.NON_TRANSACTIONAL)
   .useSynchronization(false)
   .notifications(true)
   .reaperWakeUpInterval(30000)
   .cacheStopTimeout(30000)
   .transactionManagerLookup(new GenericTransactionManagerLookup())
   .recovery()
   .enabled(false)
   .recoveryInfoCacheName("__recoveryInfoCacheName__");
```

- **isolation** - 分離レベルを設定します。詳細は、分離レベル を参照してください。デフォルトは **REPEATABLE_READ** です。
- **locking** - キャッシュが楽観的または悲観的ロックを使用するかどうかを設定します。詳細は、トランザクションのロック を参照してください。デフォルトは **OPTIMISTIC** です。
- **auto-commit**: 有効にすると、ユーザーは 1 回の操作でトランザクションを手動で開始する必要はありません。トランザクションは自動的に起動およびコミットされます。デフォルトは **true** です。

- **complete-timeout** - 完了したトランザクションに関する情報を保持する期間 (ミリ秒単位)。デフォルトは **60000** です。
- **mode**: キャッシュがトランザクションかどうかを設定します。デフォルトは **NONE** です。利用可能なオプションは以下のとおりです。
  - **NONE** - 非トランザクションキャッシュ
  - **FULL_XA** - リカバリーが有効になっている XA トランザクションキャッシュリカバリーの詳細は、トランザクションリカバリー を参照してください。
  - **NON_DURABLE_XA** - リカバリーが無効になっている XA トランザクションキャッシュ。
  - **NON_XA** - XA の代わりに 同期化 を介して統合されたトランザクションキャッシュ。詳細は、同期の登録 のセクションを参照してください。
  - **BATCH**- バッチを使用して操作をグループ化するトランザクションキャッシュ。詳細は バッチ処理 のセクションを参照してください。
- **notifications** - キャッシュリスナーのトランザクションイベントを有効/無効にします。デフォルトは **true** です。
- **reaper-interval** - トランザクション完了情報をクリーンアップするスレッドが開始する間隔 (ミリ秒単位)。デフォルトは **30000** です。
- **recovery-cache** - リカバリー情報を保存するキャッシュ名を設定します。リカバリーの詳細は、トランザクションリカバリー を参照してください。デフォルトは **recoveryInfoCacheName** です。
- **stop-timeout** - キャッシュの停止時に進行中のトランザクションを待機する時間 (ミリ秒単位)。デフォルトは **30000** です。
- **transaction-manager-lookup** - **jakarta.transaction.TransactionManager** への参照を検索するクラスの完全修飾クラス名を設定します。デフォルトは **org.infinispan.transaction.lookup.GenericTransactionManagerLookup** です。

2 フェーズコミット (2PC) が Data Grid に実装される方法、およびロックが取得される方法についての詳細は、以下のセクションを参照してください。設定の詳細については、設定リファレンス を参照してください。

9.1.2. 分離レベル

Data Grid は、READ_COMMITTED および REPEATABLE_READ の 2 つの分離レベルを提供します。

これらの分離レベルは、リーダーが同時書き込みを確認するタイミングを決定し、**MVCCEntry** の異なるサブクラスを使用して内部的に実装されます。MVCCEntry では、状態がデータコンテナーにコミットされる方法が異なります。

以下は、Data Grid のコンテキストの **READ_COMMITTED** および **REPEATABLE_READ** の違いを理解する上で役立つ詳細な例です。**READ_COMMITTED** の場合、同じキーで連続して 2 つの読み取り呼び出しを行うと、キーが別のトランザクションによって更新され、2 つ目の読み取りによって新しい更新値が返されることがあります。

```
Thread1: tx1.begin()
Thread1: cache.get(k) // returns v
Thread2:                          tx2.begin()
Thread2:                          cache.get(k) // returns v
```

```
Thread2:                          cache.put(k, v2)
Thread2:                          tx2.commit()
Thread1: cache.get(k) // returns v2!
Thread1: tx1.commit()
```

REPEATABLE_READ では、最終 get は引き続き **v** を返します。そのため、トランザクション内で同じキーを複数回取得する場合は、**REPEATABLE_READ** を使用する必要があります。

ただし、読み取りロックが **REPEATABLE_READ** に対しても取得されないため、この現象が発生する可能性があります。

```
cache.get("A") // returns 1
cache.get("B") // returns 1

Thread1: tx1.begin()
Thread1: cache.put("A", 2)
Thread1: cache.put("B", 2)
Thread2:                          tx2.begin()
Thread2:                          cache.get("A") // returns 1
Thread1: tx1.commit()
Thread2:                          cache.get("B") // returns 2
Thread2:                          tx2.commit()
```

9.1.3. トランザクションのロック

9.1.3.1. 悲観的なトランザクションキャッシュ

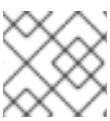
ロック取得の観点では、悲観的トランザクションはキーの書き込み時にキーのロックを取得します。

1. ロック要求がプライマリー所有者に送信されます (明示的なロック要求または操作のいずれか)。
2. プライマリーの所有者はロックの取得を試みます。
   a. 成功した場合は、正の応答が返されます。
   b. そうでない場合は、負の応答が送信され、トランザクションはロールバックされます。

たとえば、以下のようになります。

```
transactionManager.begin();
cache.put(k1,v1); //k1 is locked.
cache.remove(k2); //k2 is locked when this returns
transactionManager.commit();
```

cache.put(k1,v1) が返されると、**k1** はロックされ、クラスター内のどこかで実行中の他のトランザクションは、これに書き込むことができません。**k1** の読み取りは引き続き可能です。トランザクションの完了時に **k1** のロックが解放されます (コミットまたはロールバック)。



注記

条件付き操作の場合、検証はオリジネーターで実行されます。

9.1.3.2. 楽観的トランザクションキャッシュ

楽観的トランザクションロックはトランザクションの準備時に取得され、トランザクションのコミット (またはロールバック) まで保持されます。これは、書き込みでローカルロックを取得し、準備中にクラスターのロックが取得される 5.0 デフォルトロックモデルとは異なります。

1. 準備はすべての所有者に送信されます。
2. プライマリーの所有者は、必要なロックの取得を試みます。
   a. ロックに成功すると、書き込みのスキューチェックが実行されます。
   b. 書き込みスキューチェックが成功した場合 (または無効化された場合) は、正の応答を送信します。
   c. それ以外の場合は、負の応答が送信され、トランザクションはロールバックされます。

たとえば、以下のようになります。

```
transactionManager.begin();
cache.put(k1,v1);
cache.remove(k2);
transactionManager.commit(); //at prepare time, K1 and K2 is locked until committed/rolled back.
```

注記

条件付きコマンドの場合、検証は引き続きオリジネーターで実行されます。

9.1.3.3. 悲観的または楽観的トランザクションのどちらが必要か

ユースケースの観点からは、同時に実行されている複数のトランザクション間で多くの競合が **ない** 場合は、楽観的トランザクションを使用する必要があります。これは、読み取り時と、コミット時 (書き込みスキューチェックが有効) の間でデータが変更された場合に、楽観的トランザクションがロールバックするためです。

一方、キーでの競合が多く、トランザクションのロールバックがあまり望ましくない場合は、悲観的トランザクションの方が適している可能性があります。悲観的トランザクションは、その性質上、よりコストがかかります。各書き込み操作ではロックの取得に RPC が関係する可能性があります。

9.1.4. スキューの書き込み

書き込みスキューは、2 つのトランザクションが独立して同時に同じキーの読み取りと書き込みを行うときに発生します。書き込みスキューの結果、両方のトランザクションは同じキーに対して更新を正常にコミットしますが、値は異なります。

Data Grid は、書き込みスキューチェックを自動的に実行し、楽観的トランザクションで **REPEATABLE_READ** 分離レベルのデータの一貫性を確保します。これにより、Data Grid はトランザクションの 1 つを検出し、ロールバックできます。

LOCAL モードで動作する場合、書き込みスキューの確認は Java オブジェクト参照に依存して違いを比較します。これにより、書き込みスキューをチェックするための信頼性の高い技術が提供されます。

9.1.4.1. 悲観的トランザクションでのキーへの書き込みロックの強制

悲観的トランザクションでの書き込みスキューを回避するには、**Flag.FORCE_WRITE_LOCK** で読み取り時にキーをロックします。

注記

- トランザクション以外のキャッシュでは、**Flag.FORCE_WRITE_LOCK** は動作しません。**get()** 呼び出しは、キーの値を読み取りますが、ロックをリモートで取得しません。
- **Flag.FORCE_WRITE_LOCK** は、同じトランザクションでエンティティーが後で更新されるトランザクションと併用する必要があります。

Flag.FORCE_WRITE_LOCK の例については、以下のコードスニペットを比較してください。

```
// begin the transaction
if (!cache.getAdvancedCache().lock(key)) {
   // abort the transaction because the key was not locked
} else {
   cache.get(key);
   cache.put(key, value);
   // commit the transaction
}
```

```
// begin the transaction
try {
   // throws an exception if the key is not locked.
   cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(key);
   cache.put(key, value);
} catch (CacheException e) {
   // mark the transaction rollback-only
}
// commit or rollback the transaction
```

9.1.5. 例外への対処

CacheException (またはそのサブクラス) が JTA トランザクションの範囲内のキャッシュメソッドによってスローされる場合、トランザクションは自動的にロールバック対象としてマークされます。

9.1.6. 同期の登録

デフォルトでは、Data Grid は XAResource を介して、分散トランザクションの最初のクラス参加者として登録します。トランザクションの参加者として Data Grid が必要ではなく、ライフサイクル (準備、完了) によってのみ通知される状況があります (例: Data Grid が Hibernate で 2 次レベルキャッシュとして使用される場合など)。

Data Grid は、同期 を介したトランザクションのエンリストを許可します。これを有効にするには、**NON_XA** トランザクションモードを使用します。

Synchronization には、**TransactionManager** が 1PC で 2PC を最適化できるという利点があります。この場合、他の 1 つのリソースのみがそのトランザクションにエンリストされます (last resource commit optimization)。たとえば、Hibernate 2 次キャッシュ: Data Grid がコミット時よりも **XAResource** として **TransactionManager** に登録する場合、**TransactionManager** は 2 つの **XAResource** (キャッシュとデータベース) を認識し、この最適化を行いません。2 つのリソース間で調整する必要があるため、tx ログをディスクに書き込む必要があります。一方、Data Grid を **Synchronization** として登録すると、**TransactionManager** はディスクへのログの書き込みを省略します (パフォーマンスが向上)。

9.1.7. バッチ処理

バッチ処理は、トランザクションの原子性といくつかの特性を許可しますが、本格的な JTA または XA 機能は許可しません。多くの場合、バッチ処理は本格的なトランザクションよりもはるかに軽量で安価です。

ヒント

一般的には、トランザクションの参加者のみが Data Grid クラスターである場合に、バッチ処理 API を使用する必要があります。反対に、トランザクションに複数のシステムが必要な場合に、(**TransactionManager** に関連する)JTA トランザクションを使用する必要があります。たとえば、トランザクションの "Hello world!" を考慮すると、ある銀行口座から別の銀行口座にお金を転送します。両方の口座が Data Grid 内に保存されている場合は、バッチ処理を使用できます。ある口座がデータベースにあり、もう 1 つの口座が Data Grid の場合は、分散トランザクションが必要になります。

注記

バッチ処理を使用するためにトランザクションマネージャーを定義する必要は**ありません**。

9.1.7.1. API

バッチ処理を使用するようにキャッシュを設定したら、**Cache** で **startBatch()** と **endBatch()** を呼び出して使用します。例:

```
Cache cache = cacheManager.getCache();
// not using a batch
cache.put("key", "value"); // will replicate immediately

// using a batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true); // This will now replicate the modifications since the batch was started.

// a new batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false); // This will "discard" changes made in the batch
```

9.1.7.2. バッチ処理と JTA

裏では、バッチ機能が JTA トランザクションを開始し、そのスコープ内のすべての呼び出しがそれに関連付けられます。これには、内部 **TransactionManager** 実装が非常に簡単な (例: リカバリーなし) を使用します。バッチ処理では、以下を取得します。

1. 呼び出し中に取得したロックはバッチが完了するまで保持されます。
2. 変更はすべて、バッチ完了プロセスの一部として、クラスター内でバッチ内に複製されます。バッチの各更新のレプリケーションチャット数を減らします。

3. 同期のレプリケーションまたは無効化が使用された場合は、レプリケーション/無効化の失敗により、バッチがロールバックされます。

4. すべてのトランザクション関連の設定は、バッチ処理にも適用されます。

9.1.8. トランザクションリカバリー

リカバリーは XA トランザクションの機能であり、リソースの不測の事態、場合によってはトランザクションマネージャーの障害を対処し、それに応じてそのような状況から回復します。

9.1.8.1. リカバリーを使用するタイミング

外部データベースに保存されたアカウントから Data Grid に保管されたアカウントに転送される分散トランザクションについて考えてみましょう。**TransactionManager.commit()** が呼び出されると、両方のリソースが正常に完了します (第 1 フェーズ)。コミット (第 2) フェーズでは、データベースは、トランザクションマネージャーからコミットリクエストを受け取る前に、Data Grid の変更を問題なく適用します。この時点では、システムが一貫性のない状態です。お金は外部データベースのア口座から取得されますが、まだ Data Grid には表示されません (ロックは 2 フェーズコミットプロトコルの 2 番目のフェーズでのみリリースされます)。リカバリーはこの状況に対応することで、データベースと Data Grid の両方のデータが一貫した状態で終了します。

9.1.8.2. 仕組み

リカバリーはトランザクションマネージャーによって調整されます。トランザクションマネージャーは Data Grid と連携して、手動による介入が必要な未確定のトランザクションのリストを決定し、システム管理者に (電子メール、ログアラートなどを介して) 通知します。このプロセスはトランザクションマネージャーに固有のものですが、通常トランザクションマネージャーで設定が必要になります。

未確定のトランザクション ID を把握すると、システム管理者は Data Grid クラスターに接続し、トランザクションのコミットを再生したり、ロールバックを強制できるようになりました。Data Grid は、この JMX ツールを提供します。これは、トランザクションのリカバリーおよび調整セクション で広範囲に説明されています。

9.1.8.3. リカバリーの設定

Data Grid では、リカバリーはデフォルトでは**有効になっていません**。無効にすると、**TransactionManager** は Data Grid と動作しないため、インダウト状態のトランザクションを決定できません。トランザクションの設定 セクションでは、その設定を有効にする方法を示しています。

注記: **recovery-cache** 属性は必須ではなく、キャッシュごとに設定されます。

注記

リカバリーが機能するには、完全な XA トランザクションが必要であるため、**mode** を **FULL_XA** に設定する必要があります。

9.1.8.3.1. JMX サポートの有効化

リカバリー JMX サポートの管理に JMX を使用できるようにするには、明示的に有効にする必要があります。

9.1.8.4. リカバリーキャッシュ

未確定のトランザクションを追跡し、それらに応答できるようにするために、Data Grid は将来の使用

のためにすべてのトランザクション状態をキャッシュします。この状態は、未確定のトランザクションに対してのみ保持され、コミット/ロールバックフェーズが完了した後、正常に完了したトランザクションに対しては削除されます。

この未確定のトランザクションデータはローカルキャッシュ内に保持されます。これにより、データが大きくなりすぎた場合に、キャッシュローダーを介してこの情報をディスクにスワップするように設定できます。このキャッシュは、**recovery-cache** 設定属性を介して指定できます。指定のない場合は、Data Grid がローカルキャッシュを設定します。

リカバリーが有効になっているすべての Data Grid キャッシュ間で同じリカバリーキャッシュを共有することは可能です (必須ではありません)。デフォルトのリカバリーキャッシュがオーバーライドされた場合、指定のリカバリーキャッシュは、キャッシュ自体が使用するものとは異なるトランザクションマネージャーを返す TransactionManagerLookup を使用する必要があります。

9.1.8.5. トランザクションマネージャーとの統合

これはトランザクションマネージャーに固有のものですが、通常トランザクションマネージャーは **XAResource.recover()** を呼び出すために **XAResource** 実装への参照が必要になります。Data Grid **XAResource** の以下の API への参照を取得するには、以下を行います。

```
XAResource xar = cache.getAdvancedCache().getXAResource();
```

トランザクションを実行するプロセスとは異なるプロセスで復元を実行することが一般的です。

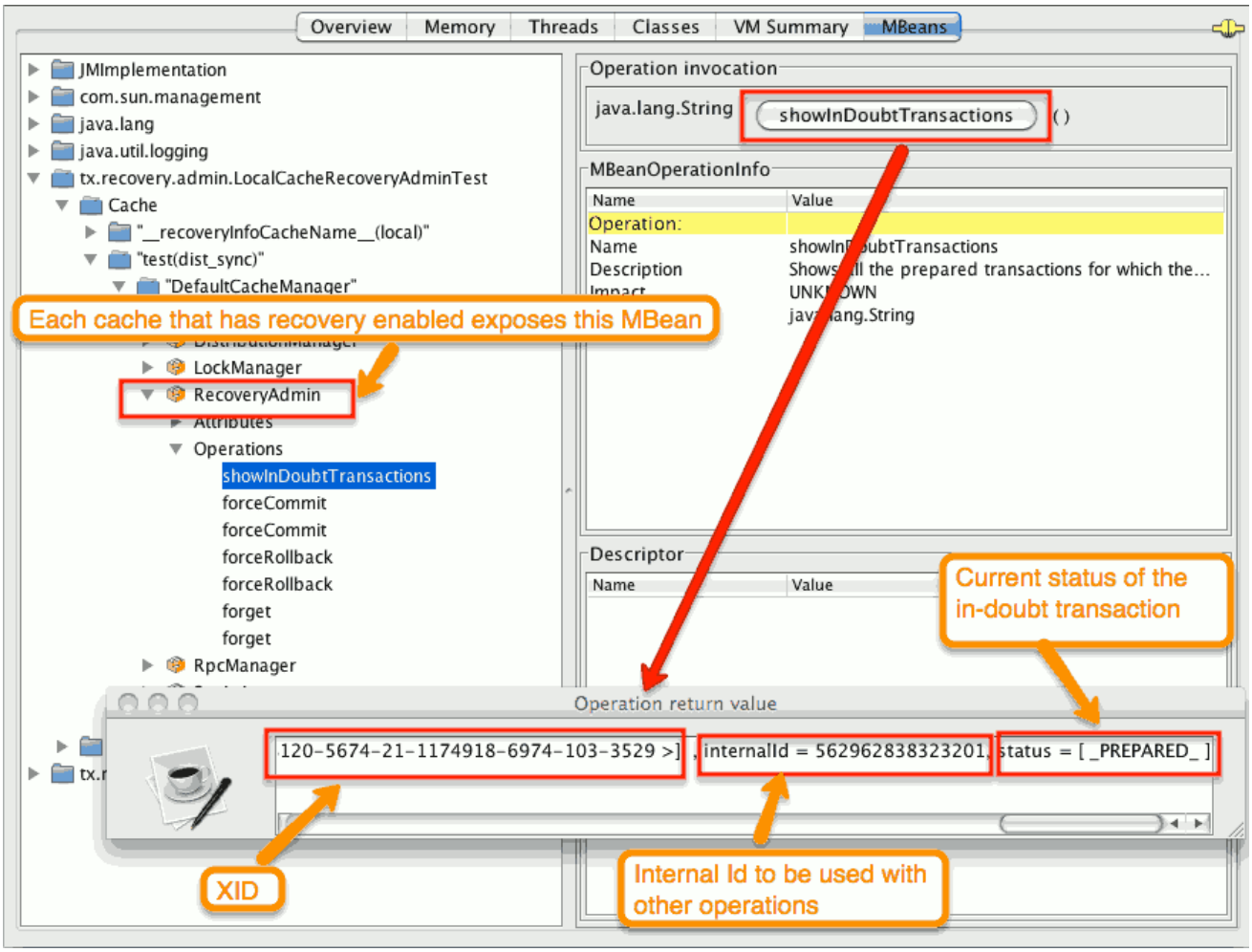
9.1.8.6. 調整

トランザクションマネージャーは、システム管理者に未確定のトランザクションについて独自の方法で通知します。この段階では、システム管理者がトランザクションの XID(バイトアレイ) を把握していることを前提としています。

通常のリカバリーフローは以下のとおりです。

- **ステップ 1**: システム管理者は、JMX を介して Data Grid サーバーに接続し、未確定のトランザクションをリスト表示します。以下のイメージは、未確定のトランザクションを持つ Data Grid ノードに接続する JConsole を示しています。

図9.1 未確定のトランザクションの表示

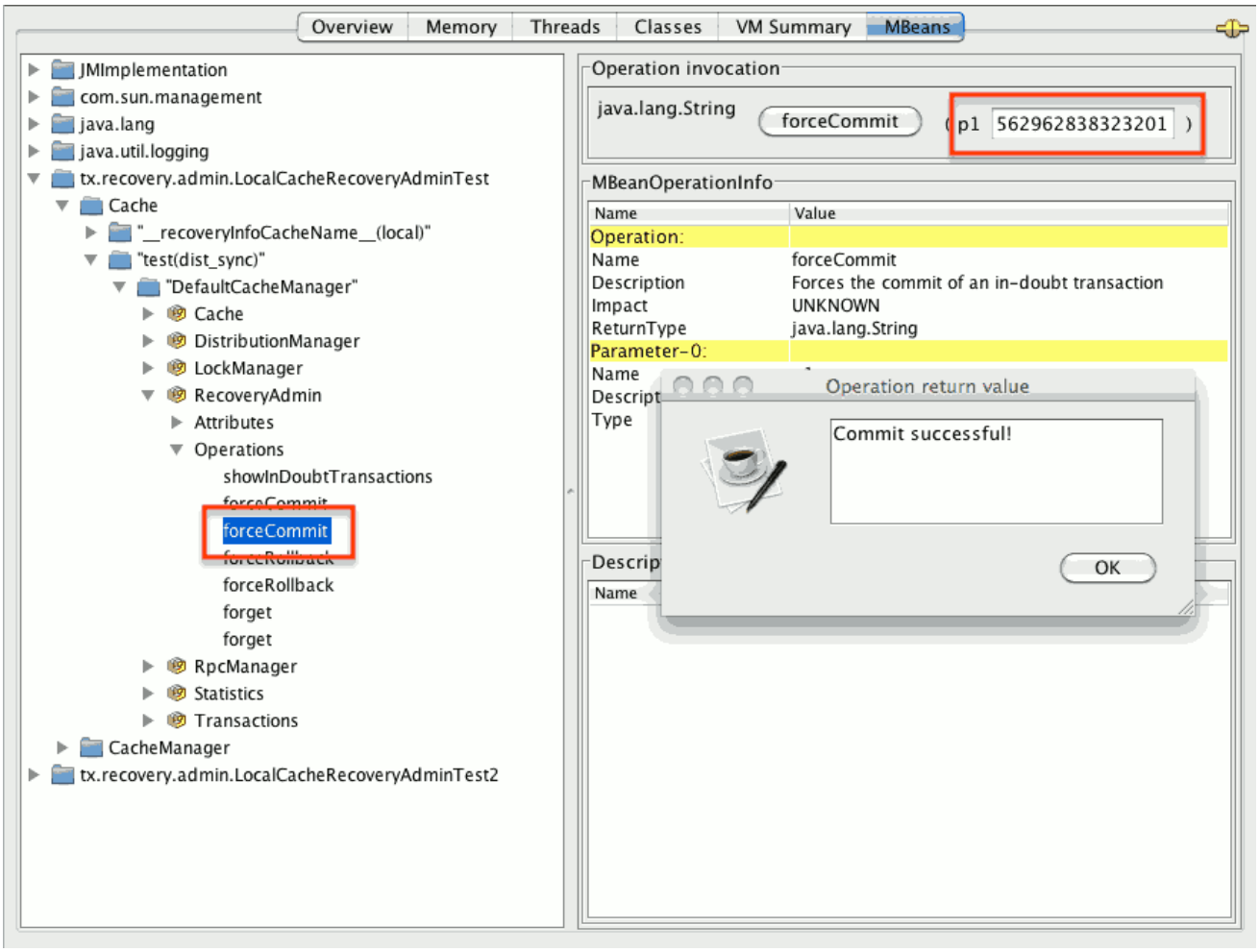


未確定の各トランザクションのステータスが表示されます (この例では "**PREPARED**" です)。status フィールドに複数の要素が存在する可能性があります。たとえば、トランザクションが特定ノードでコミットされていても、それらのノードでコミットされない場合は "PREPARED" および "COMMITTED" です。

- **ステップ 2**: システム管理者は、トランザクションマネージャーから受け取った XID を数字で表した Data Grid 内部 ID に視覚的にマッピングします。XID(バイトアレイ) は、JMX ツール (JConsole など) に渡して Data Grid 側で再アセンブルされるため、このステップが必要です。

- **ステップ 3**: システム管理者は、内部 ID に基づいて、対応する jmx 操作を介してトランザクションのコミット/ロールバックを強制的に実行します。以下のイメージは、内部 ID に基づいてトランザクションのコミットを強制することで取得します。

図9.2 コミットの強制



ヒント

上記のすべての JMX 操作は、トランザクションの発信場所に関係なく、任意のノードで実行できます。

9.1.8.6.1. XID に基づくコミット/ロールバックの強制

未確定のトランザクションのコミット/ロールバックの強制を行う XID ベースの JMX 操作も使用できます。これらのメソッドはトランザクションに関連する番号ではなく、XID を記述する byte[] アレイを受け取ります (前述のステップ 2 で説明)。これらは、たとえば、特定の未確定トランザクションの自動完了ジョブを設定する場合に役立ちます。このプロセスはトランザクションマネージャーのリカバリーにプラグインされ、トランザクションマネージャーの XID オブジェクトにアクセスできます。

第10章 ロックと同時実行の設定

Data Grid は、マルチバージョン同時実行制御 (MVCC) を使用して、共有データへのアクセスを改善します。

- 同時リーダーとライターの許可
- リーダーとライターが互いにブロックしない
- 書き込みスキューを検出して処理できる
- 内部ロックのストライピングが可能

10.1. ロックおよび同時実行

マルチバージョン同時実行制御 (MVCC) は、リレーショナルデータベースやその他のデータストアで一般的な同時実行スキームです。MVCC には、粗粒度の Java 同期や、共有データにアクセスするための JDK ロックに比べて、多くの利点があります。

Data Grid の MVCC 実装では、ロックと同期が最小限に抑えられており、可能な限り compare-and-swap などのロックフリー技術やロックフリーのデータ構造などに重点を置いています。これにより、マルチ CPU 環境とマルチコア環境の最適化に役立ちます。

特に、Data Grid の MVCC 実装はリーダーに対して高度に最適化されています。リーダースレッドは、エントリーの明示的なロックを取得せず、代わりに問題のエントリーを直接読み込みます。

一方、ライターは、書き込みロックを取得する必要があります。これにより、エントリーごとに 1 つの同時書き込みのみが保証されるため、同時ライターはキューイングしてエントリーを変更することになります。

同時読み取りを可能にするため、ライターはエントリーを **MVCCEntry** でラップして、変更する予定のエントリーのコピーを作成します。このコピーは、同時リーダーが部分的に変更された状態を認識できないようにします。書き込みが完了したら、**MVCCEntry.commit()** はデータコンテナーへの変更をフラッシュし、後続のリーダーに変更内容が反映されます。

10.1.1. クラスター化されたキャッシュおよびロック

Data Grid クラスターでは、プライマリー所有者ノードがキーをロックします。

非トランザクションキャッシュの場合、Data Grid は書き込み操作をキーのプライマリー所有者に転送して、ロックを試行できるようにします。次に、Data Grid は、他の所有者に書き込み操作を転送するか、キーをロックできない場合は例外を出力します。

注記

操作が条件付きで、プライマリー所有者で失敗した場合、Data Grid はこれを他の所有者には転送しません。

トランザクションキャッシュの場合、プライマリーの所有者は楽観的および悲観的ロックモードでキーをロックできます。Data Grid は、トランザクション間の同時読み取りを制御する異なる分離レベルもサポートします。

10.1.2. LockManager

LockManager は、書き込み用にエントリーをロックするコンポーネントです。**LockManager** は、**LockContainer** を使用して、ロックを検索、保持、作成します。**LockContainers** には、ロックストライピングをサポートするものと、エントリーごとに 1 つのロックをサポートするものの 2 つの大まかな特徴があります。

10.1.3. ロックストライピング

ロックストライピングでは、固定サイズの共有ロックコレクションをキャッシュ全体に使用する必要があり、ロックはエントリーのキーのハッシュコードに基づいてエントリーに割り当てられます。JDK の **ConcurrentHashMap** がロックを割り当てる方法と同様に、これにより、関連性のない可能性のあるエントリーが同じロックによってブロックされる代わりに、拡張性の高い固定オーバーヘッドのロックメカニズムが可能になります。

別の方法は、ロックストライピングを無効にすることです。これは、エントリーごとに **新しい** ロックが作成されることを意味します。このアプローチでは、スループットが高くなる **可能性** がありますが、追加のメモリー使用量やガベージコレクションのチャーンなどのコストがかかります。

デフォルトのロックストライピング設定

異なるキーのロックが同じロックストライプになってしまうとデッドロックが発生する可能性があるため、ロックストライピングはデフォルトで無効になっています。

ロックストライピングで使用される共有ロックコレクションのサイズは、**<locking />** 設定要素の **concurrencyLevel** 属性を使用して調整できます。

設定例:

```
<locking striping="false|true"/>
```

または、以下を実行します。

```
new ConfigurationBuilder().locking().useLockStriping(false|true);
```

10.1.4. 同時実行レベル

この同時実行レベルは、ストライプロックコンテナーのサイズを決定する他に、**DataContainer** の内部など、関連する JDK **ConcurrentHashMap** ベースのコレクションを調整するためにも使用されます。このパラメーターは、Data Grid でもまったく同じ方法で使用されているため、同時実行レベルの詳細については、JDK **ConcurrentHashMap** Javadocs を参照してください。

設定例:

```
<locking concurrency-level="32"/>
```

または、以下を実行します。

```
new ConfigurationBuilder().locking().concurrencyLevel(32);
```

10.1.5. ロックタイムアウト

ロックタイムアウトは、競合するロックを待つ時間 (ミリ秒単位) を指定します。

設定例:

```
<locking acquire-timeout="10000"/>
```

または、以下を実行します。

```
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10000);
//alternatively
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10, TimeUnit.SECONDS);
```

10.1.6. 一貫性

(すべての所有者がロックされているのとは対照的に) 単一の所有者がロックされるという事実により、次の一貫性の保証が失われることはありません。キー **K** がノード **{A, B}** に対してハッシュ化され、トランザクション **TX1** が、たとえば、**A** 上の **K** のロックを取得したとします。別のトランザクション **TX2** が **B** (またはその他のノード) 上で開始され、**TX2** が **K** のロックを試みる場合、ロックがすでに **TX1** によって保持されているため、タイムアウトでロックに失敗します。理由は、キー **K** のロックがトランザクションの発生場所に関係なく、常に、確定的に、クラスターの同じノードで取得されるからです。

10.1.7. データのバージョン管理

Data Grid は、simple と external の 2 つの形式のデータバージョン管理をサポートします。simple バージョン管理は、書き込みスキューチェックのトランザクションキャッシュで使用されます。

external バージョン管理は、Data Grid を Hibernate で使用する場合など、Data Grid 内のデータバージョン管理の外部ソースをカプセル化するために使用され、そのデータバージョン情報をデータベースから直接取得します。

このスキームでは、バージョンに渡すメカニズムが必要になり、オーバーロードされたバージョン **put()** および **putForExternalRead()** が、**AdvancedCache** で提供され、外部データバージョンを取り込みます。その後、これは **InvocationContext** に保管され、コミット時にエントリーに適用されます。

注記

external バージョン管理の場合、書き込みスキューチェックは実行できず、実行されません。

第11章 クラスター化されたカウンターの使用

Data Grid は、オブジェクトの数を記録するカウンターを提供し、クラスター内のすべてのノードに分散されます。

11.1. クラスター化カウンター

クラスター化されたカウンター は、Data Grid クラスターのすべてのノードで分散され、共有されるカウンターです。カウンターは異なる整合性レベル (strong および weak) を持つことができます。

strong/weak と一貫性のあるカウンターには個別のインターフェイスがありますが、どちらもその値の更新をサポートし、現在の値を返し、その値が更新されたときにイベントを提供します。このドキュメントでは、ユースケースに最適なものを選択する上で役立つ詳細を以下に示します。

11.1.1. インストールおよび設定

カウンターの使用を開始するには、Maven の **pom.xml** ファイルに依存関係を追加する必要があります。

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-counter</artifactId>
</dependency>
```

このカウンターは、本書で後述する **CounterManager** インターフェイスを介して、Data Grid 設定ファイルまたはオンデマンドを設定できます。**EmbeddedCacheManager** の起動時に、起動時に Data Grid 設定ファイルに設定されたカウンターが作成します。これらのカウンターは Eagerly で開始され、すべてのクラスターのノードで利用できます。

configuration.xml

```
<infinispan>
   <cache-container ...>
      <!-- To persist counters, you need to configure the global state. -->
      <global-state>
      <!-- Global state configuration goes here. -->
      </global-state>
      <!-- Cache configuration goes here. -->
       <counters xmlns="urn:infinispan:config:counters:14.0" num-owners="3"
reliability="CONSISTENT">
          <strong-counter name="c1" initial-value="1" storage="PERSISTENT"/>
          <strong-counter name="c2" initial-value="2" storage="VOLATILE" lower-bound="0"/>
          <strong-counter name="c3" initial-value="3" storage="PERSISTENT" upper-bound="5"/>
          <strong-counter name="c4" initial-value="4" storage="VOLATILE" lower-bound="0" upper-
bound="10"/>
          <strong-counter name="c5" initial-value="0" upper-bound="100" lifespan="60000"/>
          <weak-counter name="c6" initial-value="5" storage="PERSISTENT" concurrency-level="1"/>
      </counters>
   </cache-container>
</infinispan>
```

または、プログラムを使用して **GlobalConfigurationBuilder** で以下を行います。

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...;
CounterManagerConfigurationBuilder builder =
globalConfigurationBuilder.addModule(CounterManagerConfigurationBuilder.class);
builder.numOwner(3).reliability(Reliability.CONSISTENT);
builder.addStrongCounter().name("c1").initialValue(1).storage(Storage.PERSISTENT);
builder.addStrongCounter().name("c2").initialValue(2).lowerBound(0).storage(Storage.VOLATILE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PERSISTENT)
;
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).storage(Storag
e.VOLATILE);
builder.addStrongCounter().name("c5").initialValue(0).upperBound(100).lifespan(60000);
builder.addWeakCounter().name("c6").initialValue(5).concurrencyLevel(1).storage(Storage.PERSIST
ENT);
```

一方、このカウンターは、**EmbeddedCacheManager** を初期化した後にいつでも設定することができます。

```
CounterManager manager = ...;
manager.defineCounter("c1",
CounterConfiguration.builder(CounterType.UNBOUNDED_STRONG).initialValue(1).storage(Storage.
PERSISTENT).build());
manager.defineCounter("c2",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(2).lowerBound(0).stora
ge(Storage.VOLATILE).build());
manager.defineCounter("c3",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(3).upperBound(5).stor
age(Storage.PERSISTENT).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(4).lowerBound(0).uppe
rBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(0).upperBound(100).lif
espan(60000).build());
manager.defineCounter("c6",
CounterConfiguration.builder(CounterType.WEAK).initialValue(5).concurrencyLevel(1).storage(Storag
e.PERSISTENT).build());
```

注記

CounterConfiguration は変更できず、再利用できます。

カウンターが正常に設定されていると、**defineCounter()** メソッドは **true** を返します。そうでない場合は、**true** を返します。ただし、設定が無効な場合は、メソッドによって **CounterConfigurationException** が発生します。カウンターがすでに定義されているかを調べるには、**isDefined()** メソッドを使用します。

```
CounterManager manager = ...
if (!manager.isDefined("someCounter")) {
   manager.define("someCounter", ...);
}
```

関連情報

- Data Grid 設定スキーマ参照

11.1.1.1. カウンター名のリスト表示

定義されたすべてのカウンターをリスト表示するには、**CounterManager.getCounterNames()** メソッドは、クラスター全体で作成されたすべてのカウンター名のコレクションを返します。

11.1.2. CounterManager インターフェイス

CounterManager インターフェイスは、カウンターを定義、取得、および削除するエントリーポイントです。

埋め込みデプロイメント

CounterManager は **EmbeddedCacheManager** の作成を自動的にリッスンし、**EmbeddedCacheManager** ごとのインスタンスの登録を続行します。カウンター状態を保存し、デフォルトのカウンターの設定に必要なキャッシュを開始します。

CounterManager の取得は、以下の例のように **EmbeddedCounterManagerFactory.asCounterManager(EmbeddedCacheManager)** を呼び出すだけです。

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager manager = ...;

// retrieve the CounterManager
CounterManager counterManager =
EmbeddedCounterManagerFactory.asCounterManager(manager);
```

サーバーデプロイメント

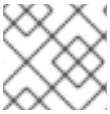
Hot Rod クライアントの場合、**CounterManager** は RemoteCacheManager に登録されており、以下のように取得できます。

```
// create or obtain your RemoteCacheManager
RemoteCacheManager manager = ...;

// retrieve the CounterManager
CounterManager counterManager = RemoteCounterManagerFactory.asCounterManager(manager);
```

11.1.2.1. CounterManager を介したカウンターの削除

Strong/WeakCounter と **CounterManager** でカウンターを削除するのに違いがあります。**CounterManager.remove(String)** は、クラスターからカウンター値を削除し、ローカルカウンターインスタンスのカウンターに登録されているすべてのリスナーを削除します。さらに、カウンターインスタンスは再利用可能ではなくなり、無効な結果が返される可能性があります。

一方で、**Strong/WeakCounter** を削除するとカウンター値のみが削除されます。インスタンスは引き続き再利用でき、リスナーは引き続き動作します。



注記

削除後にアクセスされると、カウンターは再作成されます。

11.1.3. カウンター

カウンターは、strong (**StrongCounter**) または weak(**WeakCounter**) になり、いずれも名前で識別されます。各インターフェイスには特定のインターフェイスがありますが、ロジック (つまり各操作により **CompletableFuture** が返される) を共有しているため、更新イベントが返され、初期値にリセットできます。

非同期 API を使用しない場合は、**sync()** メソッドを介して同期カウンターを返すことができます。API は同じですが、**CompletableFuture** の戻り値はありません。

以下のメソッドは、両方のインターフェイスに共通しています。

```
String getName();
CompletableFuture<Long> getValue();
CompletableFuture<Void> reset();
<T extends CounterListener> Handle<T> addListener(T listener);
CounterConfiguration getConfiguration();
CompletableFuture<Void> remove();
SyncStrongCounter sync(); //SyncWeakCounter for WeakCounter
```

- **getName()** はカウンター名 (identifier) を返します。
- **getValue()** は現在のカウンターの値を返します。
- **reset()** により、カウンターの値を初期値にリセットできます。
- **addListener()** はリスナーを登録し、更新イベントを受信します。詳細については、通知およびイベント セクションをご覧ください。
- **getConfiguration()** はカウンターによって使用される設定を返します。
- **remove()** はクラスターからカウンター値を削除します。インスタンスは引き続き使用でき、リスナーが保持されます。
- **sync()** は同期カウンターを作成します。

注記

削除後にアクセスされると、カウンターは再作成されます。

11.1.3.1. StrongCounter インターフェイス: 一貫性または境界が明確になります。

strong カウンターは、Data Grid キャッシュに保存されている単一のキーを使用して、必要な一貫性を提供します。すべての更新は、その値を更新するためにキーロックの下で実行されます。一方、読み取りはロックを取得し、現在の値を読み取ります。さらに、このスキームではカウンター値をバインドでき、比較および設定/スワップなどのアトミック操作を提供できます。

StrongCounter は、**getStrongCounter()** メソッドを使用して **CounterManager** から取得することができます。たとえば、以下のようになります。

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getStrongCounter("my-counter");
```

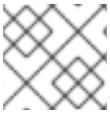
警告

すべての操作は単一のキーに到達するため、**StrongCounter** は競合レートが高くなります。

StrongCounter インターフェイスでは、以下のメソッドを追加します。

```
default CompletableFuture<Long> incrementAndGet() {
  return addAndGet(1L);
}

default CompletableFuture<Long> decrementAndGet() {
  return addAndGet(-1L);
}

CompletableFuture<Long> addAndGet(long delta);

CompletableFuture<Boolean> compareAndSet(long expect, long update);

CompletableFuture<Long> compareAndSwap(long expect, long update);
```

- **incrementAndGet()** はカウンターを 1 つずつ増分し、新しい値を返します。
- **decrementAndGet()** は、1 つずつカウンターをデクリメントし、新しい値を返します。
- **addAndGet()** は、delta をカウンターの値に追加し、新しい値を返します。
- **compareAndSet()** および **compareAndSwap()** は、現在の値が想定される場合にカウンターの値を設定します。



注記

CompletableFuture が完了すると、操作は完了とみなされます。

注記

compare-and-set と compare-and-swap の相違点は、操作に成功した場合に、compare-and-set は true を返しますが、compare-and-swap は前の値をか返すことです。戻り値が期待値と同じ場合は、compare-and-swap が正常になります。

11.1.3.1.1. バインドされた StrongCounter

バインドされている場合、上記の更新メソッドはすべて、下限または上限に達すると **CounterOutOfBoundsException** を出力します。例外には、どちら側にバインドが到達したかを確認するための次のメソッドがあります。

```
public boolean isUpperBoundReached();
public boolean isLowerBoundReached();
```

11.1.3.1.2. ユースケース

強力なカウンターは、次の使用例に適しています。

- 各更新後にカウンターの値が必要な場合 (例: クラスター単位の ID ジェネレーターまたはシーケンス)
- バインドされたカウンターが必要な場合は (例: レートリミッター)

11.1.3.1.3. 使用例

```
StrongCounter counter = counterManager.getStrongCounter("unbounded_counter");

// incrementing the counter
System.out.println("new value is " + counter.incrementAndGet().get());

// decrement the counter's value by 100 using the functional API
counter.addAndGet(-100).thenApply(v -> {
   System.out.println("new value is " + v);
   return null;
}).get();

// alternative, you can do some work while the counter is updated
CompletableFuture<Long> f = counter.addAndGet(10);
// ... do some work ...
System.out.println("new value is " + f.get());

// and then, check the current value
System.out.println("current value is " + counter.getValue().get());

// finally, reset to initial value
counter.reset().get();
System.out.println("current value is " + counter.getValue().get());

// or set to a new value if zero
System.out.println("compare and set succeeded? " + counter.compareAndSet(0, 1));
```

以下に、バインドされたカウンターを使用する別の例を示します。

```
StrongCounter counter = counterManager.getStrongCounter("bounded_counter");

// incrementing the counter
try {
   System.out.println("new value is " + counter.addAndGet(100).get());
} catch (ExecutionException e) {
   Throwable cause = e.getCause();
   if (cause instanceof CounterOutOfBoundsException) {
      if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
         System.out.println("ops, upper bound reached.");
      } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
         System.out.println("ops, lower bound reached.");
      }
   }
}
```

```
// now using the functional API
counter.addAndGet(-100).handle((v, throwable) -> {
   if (throwable != null) {
      Throwable cause = throwable.getCause();
      if (cause instanceof CounterOutOfBoundsException) {
         if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
            System.out.println("ops, upper bound reached.");
         } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
            System.out.println("ops, lower bound reached.");
         }
      }
      return null;
   }
   System.out.println("new value is " + v);
   return null;
}).get();
```

Compare-and-set と Compare-and-swap の比較例:

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue, newValue;
do {
   oldValue = counter.getValue().get();
   newValue = someLogic(oldValue);
} while (!counter.compareAndSet(oldValue, newValue).get());
```

compare-and-swap では、呼び出しカウンターの呼び出し (**counter.getValue()**) が 1 つ保存されます。

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue = counter.getValue().get();
long currentValue, newValue;
do {
   currentValue = oldValue;
   newValue = someLogic(oldValue);
} while ((oldValue = counter.compareAndSwap(oldValue, newValue).get()) != currentValue);
```

stong カウンターをレートリミッターとして使用するには、以下のように **upper-bound** パラメーターおよび **lifespan** パラメーターを設定します。

```
// 5 request per minute
CounterConfiguration configuration =
CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
      .upperBound(5)
      .lifespan(60000)
      .build();
counterManager.defineCounter("rate_limiter", configuration);
StrongCounter counter = counterManager.getStrongCounter("rate_limiter");

// on each operation, invoke
try {
   counter.incrementAndGet().get();
   // continue with operation
} catch (InterruptedException e) {
   Thread.currentThread().interrupt();
} catch (ExecutionException e) {
```

```
    if (e.getCause() instanceof CounterOutOfBoundsException) {
      // maximum rate. discard operation
      return;
    } else {
      // unexpected error, handling property
    }
  }
```

注記

lifespan パラメーターは実験的な機能で、今後のバージョンで削除される可能性があります。

11.1.3.2. WeakCounter インターフェイス: 速度が必要な場合

WeakCounter は、カウンターの値を Data Grid キャッシュの複数のキーに保存します作成されたキーの数は **concurrency-level** 属性によって設定されます。各キーはカウンターの値の一部の状態を保存し、同時に更新できます。**StrongCounter** よりも優れた点は、キャッシュの競合率が低いことです。一方、値の読み取りはよりコストが高く、バインドは許可されません。

警告

リセット操作は注意して行う必要があります。これは **アトミックではなく**、中間値を生成します。これらの値は、読み取り操作および登録されたリスナーによって確認できます。

WeakCounter は、**getWeakCounter()** メソッドを使用して **CounterManager** から取得できます。たとえば、以下のようになります。

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getWeakCounter("my-counter);
```

11.1.3.2.1. weak カウンターインターフェイス

WeakCounter は、以下のメソッドを追加します。

```
default CompletableFuture<Void> increment() {
  return add(1L);
}

default CompletableFuture<Void> decrement() {
  return add(-1L);
}

CompletableFuture<Void> add(long delta);
```

これらは `StrongCounter のメソッドと似ていますが、新しい値は返されません。

11.1.3.2.2. ユースケース

weak カウンターは、更新操作の結果が必要ない場合やカウンターの値があまり必要でないユースケースに最適です。統計の収集は、このようなユースケースの良い例になります。

11.1.3.2.3. 例

以下では、弱いカウンターの使用例を示します。

```
WeakCounter counter = counterManager.getWeakCounter("my_counter");

// increment the counter and check its result
counter.increment().get();
System.out.println("current value is " + counter.getValue());

CompletableFuture<Void> f = counter.add(-100);
//do some work
f.get(); //wait until finished
System.out.println("current value is " + counter.getValue().get());

//using the functional API
counter.reset().whenComplete((aVoid, throwable) -> System.out.println("Reset done " + (throwable
== null ? "successfully" : "unsuccessfully"))).get();
System.out.println("current value is " + counter.getValue().get());
```

11.1.4. 通知およびイベント

strong カウンターと weak カウンターの両方が、更新イベントを受信するためにリスナーをサポートします。リスナーは **CounterListener** を実装する必要があり、これを以下の方法で登録できます。

```
<T extends CounterListener> Handle<T> addListener(T listener);
```

CounterListener には以下のインターフェイスがあります。

```
public interface CounterListener {
  void onUpdate(CounterEvent entry);
}
```

返される **Handle** オブジェクトには、**CounterListener** が必要なくなったときに削除するという主な目的があります。また、処理している **CounterListener** インスタンスにアクセスできます。これには、以下のインターフェイスがあります。

```
public interface Handle<T extends CounterListener> {
  T getCounterListener();
  void remove();
}
```

最後に、**CounterEvent** には、以前と現在の値と状態があります。これには、以下のインターフェイスがあります。

```
public interface CounterEvent {
  long getOldValue();
  State getOldState();
```

```
    long getNewValue();
    State getNewState();
}
```

注記

状態は、非有界である strong カウンターおよび weak カウンターでは常に **State.VALID** になります。**State.LOWER_BOUND_REACHED** および **State.UPPER_BOUND_REACHED** は有界である strong カウンターのみに有効です。

警告

weak カウンター **reset()** 操作は、中間値で複数の通知をトリガーします。

第12章 リスナーおよび通知

Data Grid でリスナーを使用して、Cache Manager またはキャッシュのイベントが発生したときに通知を取得します。

12.1. リスナーおよび通知

Data Grid はリスナー API を提供し、クライアントはイベントが発生したときに登録して通知を受け取ることができます。このアノテーション駆動型 API は、キャッシュレベルイベントと Cache Manager レベルイベントの 2 つの異なるレベルに適用されます。

イベントは、リスナーにディスパッチされる通知をトリガーします。リスナーは @Listener アノテーションが付けられ、Listenable インターフェイスで定義されたメソッドを使用して登録された単純な POJO です。

Cache と CacheManager はどちらも Listenable を実装しています。つまり、リスナーをキャッシュまたは Cache Manager のいずれかにアタッチして、キャッシュレベルまたは Cache Manager レベルのいずれかの通知を受信できます。

たとえば、次のクラスは、新しいエントリーがキャッシュに追加されるたびに、ブロックしない方法で、一部の情報を出力するようにリスナーを定義します。

```
@Listener
public class PrintWhenAdded {
  Queue<CacheEntryCreatedEvent> events = new ConcurrentLinkedQueue<>();

  @CacheEntryCreated
  public CompletionStage<Void> print(CacheEntryCreatedEvent event) {
    events.add(event);
    return null;
  }

}
```

より包括的な例は、Javadocs for @Listener を参照してください。

12.2. キャッシュレベルの通知

キャッシュレベルのイベントはキャッシュごとに発生し、デフォルトでは、イベントが発生したノードでのみ発生します。分散キャッシュでは、これらのイベントは影響を受けるデータの所有者に対してのみ発生することに注意してください。キャッシュレベルのイベントの例としては、エントリーの追加、削除、変更などがあります。これらのイベントは、特定のキャッシュに登録されているリスナーへの通知をトリガーします。

すべてのキャッシュレベルの通知とそれぞれのメソッドレベルのアノテーションの包括的なリストについては、org.infinispan.notifications.cachelistener.annotation パッケージの Javadocs を参照してください。

注記

Data Grid で使用可能なキャッシュレベルの通知のリストについては、org.infinispan.notifications.cachelistener.annotation パッケージの Javadocs を参照してください。

クラスターリスナー
単一ノードでキャッシュイベントをリッスンすることが望ましい場合は、クラスターリスナーを使用する必要があります。

そのために必要なのは、リスナーがクラスター化されているというアノテーションを付けるよう設定することだけです。

```
@Listener (clustered = true)
public class MyClusterListener { .... }
```

クラスター化されていないリスナーからのクラスターリスナーには、いくつかの制限があります。

1. クラスターリスナー
   は、**@CacheEntryModified**、**@CacheEntryCreated**、**@CacheEntryRemoved**、および **@CacheEntryExpired** イベントのみをリッスンできます。これは、他のタイプのイベントは、このリスナーに対してリッスンされないことを意味することに注意してください。

2. ポストイベントのみがクラスターリスナーに送信され、プレイベントは無視されます。

イベントのフィルタリングおよび変換
リスナーがインストールされているノードで適用可能なすべてのイベントがリスナーに発生します。KeyFilter (キーのフィルタリングのみを許可) または CacheEventFilter (キー、古い値、古いメタデータ、新しい値、新しいメタデータ、コマンドの再実行の有無、イベントがイベント (isPre など) の前であるか、およびコマンドタイプのフィルターに使用) を使用して、どのイベントが発生したかを動的にフィルターできます。

この例で、イベントがキー **Only Me** のエントリーを変更したときにイベントのみを発生させる単純な **KeyFilter** を示しています。

```
public class SpecificKeyFilter implements KeyFilter<String> {
   private final String keyToAccept;

   public SpecificKeyFilter(String keyToAccept) {
     if (keyToAccept == null) {
       throw new NullPointerException();
     }
     this.keyToAccept = keyToAccept;
   }

   public boolean accept(String key) {
     return keyToAccept.equals(key);
   }
}

...
cache.addListener(listener, new SpecificKeyFilter("Only Me"));
...
```

これは、より効率的な方法で受信するイベントを制限したい場合に便利です。

また、イベントが発生する前に値を別の値に変換できるようにする CacheEventConverter も提供できます。これは、値の変換を行うコードをモジュール化するのに適しています。

注記

上記のフィルターとコンバーターは、クラスターリスナーと組み合わせて使用すると特に効果的です。これは、イベントがリッスンされているノードではなく、イベントが発生したノードでフィルタリングと変換が行われるためです。これにより、クラスター全体でイベントを複製する必要がない (フィルター)、またはペイロードを減らす (コンバーター) という利点があります。

初期状態のイベント
リスナーがインストールされると、完全にインストールされた後にのみイベントが通知されます。

リスナーの初回登録時にキャッシュコンテンツの現在の状態を取得することが望ましい場合があります。この場合、キャッシュ内の各要素の **@CacheEntryCreated** タイプのイベントが生成されます。この最初のフェーズで追加で生成されたイベントは、適切なイベントが発生するまでキューに置かれます。

注記

現時点では、これはクラスター化されたリスナーに対してのみ機能します。ISPN-4608 では、クラスター化されていないリスナーへの追加を説明しています。

重複イベント
トランザクションではないキャッシュで、重複したイベントを受け取ることが可能です。これは、put などの書き込み操作の実行中に、キーのプライマリー所有者がダウンした場合に可能になります。

Data Grid は、指定のキーの新規プライマリー所有者に put 操作を自動的に送信することで、put 操作を内部で修正しますが、最初に書き込みがバックアップに複製されたかどうかについては保証はありません。そのため、**CacheEntryCreatedEvent**、**CacheEntryModifiedEvent**、および **CacheEntryRemovedEvent** の書き込みイベントの 1 つ以上が、1 つの操作で送信される可能性があります。

複数のイベントが生成された場合、Data Grid は再試行コマンドによって生成されたイベントをマークし、変更の表示に注意を払いなくても、このイベントが発生したタイミングを把握できるようにします。

```
@Listener
public class MyRetryListener {
  @CacheEntryModified
  public void entryModified(CacheEntryModifiedEvent event) {
    if (event.isCommandRetried()) {
      // Do something
    }
  }
}
```

また、**CacheEventFilter** または **CacheEventConverter** を使用する場合、EventTypeには、再試行によりイベントが生成されたかを確認するために、**isRetry** メソッドが含まれます。

12.3. CACHE MANAGER 通知

Cache Manager レベルで発生するイベントはクラスター全体であり、単一の Cache Manager によって作成されたすべてのキャッシュに影響するイベントが含まれます。Cache Manager レベルのイベントの例として、クラスターに参加または退出するノード、または開始または停止するキャッシュがあります。

すべての Cache Manager 通知とそれぞれのメソッドレベルのアノテーションの包括的なリストは、org.infinispan.notifications.cachemanagerlistener.annotation パッケージ を参照してください。

12.4. イベントの同期

デフォルトでは、すべての非同期通知は通知スレッドプールにディスパッチされます。同期通知は、リスナーメソッドが完了するか (スレッドがブロックする原因となる)、または CompletionStage が完了するまで、操作の続行を遅らせます。または、リスナーに**非同期**としてアノテーションを付けることもできます。この場合、操作は即座に継続され、通知は通知スレッドプールで非同期に完了します。これには、以下のようにリスナーにアノテーションを付けます。

非同期リスナー

```
@Listener (sync = false)
public class MyAsyncListener {
  @CacheEntryCreated
  void listen(CacheEntryCreatedEvent event) { }
}
```

同期リスナーのブロック

```
@Listener
public class MySyncListener {
  @CacheEntryCreated
  void listen(CacheEntryCreatedEvent event) { }
}
```

ノンブロッキングリスナー

```
@Listener
public class MyNonBlockingListener {
  @CacheEntryCreated
  CompletionStage<Void> listen(CacheEntryCreatedEvent event) { }
}
```

非同期スレッドプール
このような非同期通知のディスパッチに使用されるスレッドプールを調整するには、設定ファイルの **<listener-executor />** XML 要素を使用します。