Red Hat Data Grid 8.1

Data Grid 8 への移行

Data Grid 移行ガイド

Last Updated: 2023-12-15

Red Hat Data Grid 8.1 Data Grid 8 への移行

Data Grid 移行ガイド

法律上の通知

概要

本ガイドでは、以前のバージョンから Red Hat Data Grid 8 に正常に移行する際に役立つ詳細情報を提供します。

目次

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。

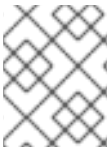
DATA GRID のドキュメント

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 8.1 ドキュメント
- Data Grid 8.1 コンポーネントの詳細
- Data Grid 8.1 でサポートされる設定
- Data Grid 8 機能のサポート
- Data Grid で非推奨の機能

DATA GRID のダウンロード

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



注記

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

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティ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、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社 の CTO、Chris Wright のメッセージ を参照してください。

第1章 DATA GRID 8

簡単な概要といくつかの基本事項を見て、Data Grid 8 への移行を始めましょう。

1.1. DATA GRID 8 への移行

Data Grid 8 では、サーバーデプロイメント用のまったく新しいアーキテクチャーなど、以前の DataGrid バージョンからの大幅な変更が導入されています。

これにより、既存の環境では移行の特定の側面がより困難になりますが、データグリッドチームは、これらの変更がデプロイメントの複雑さと管理オーバーヘッドを削減することでユーザーに利益をもたらすと考えています。

以前のバージョンと比較して、Data Grid 8 に移行すると、次のことが可能になります。

- コンテナープラットフォーム用に構築されたクラウドネイティブデザイン。
- メモリーフットプリントが軽くなり、全体的なリソース使用量が少なくなります。
- より速い開始時間。
- 攻撃対象領域を小さくすることでセキュリティーを強化します。
- Red Hat テクノロジーおよびソリューションとのより良い統合。

また、Data Grid 8 は、実績のある信頼できるオープンソーステクノロジーから構築された、可能な限り最高のメモリー内データストレージ機能を提供し続けます。

1.2. 移行パス

本ドキュメントは、Data Grid 7.3 から Data Grid 8 への移行に焦点を当てていますが、7.0.1 以降の 7.x バージョンにも引き続き適用できます。

Data Grid 6 からの移行を計画している場合、このドキュメントでは必要なものがすべて網羅されていない可能性があります。移行する前に、デプロイメントに固有のアドバイスについて Red Hat サポートに連絡する必要があります。

いつものように、このドキュメントを改善することでお手伝いできるかどうかお知らせください。

1.3. コンポーネントのダウンロード

Data Grid 8 の使用を開始するには、次のいずれかを行います。

- ベアメタルまたはその他のホスト環境に Data Grid をインストールする場合は、Red Hat カスタマーポータルからコンポーネントをダウンロードします。
- OpenShift で実行している場合は、Data Grid Operator サブスクリプションを作成します。

次の情報は、ベアメタルデプロイメントで利用可能なコンポーネントのダウンロードについて説明しています。これは、以前のバージョンの Data Grid とは異なります。

以下も参照してください。

- Data Grid on OpenShift Migration

- データグリッド 8 でサポートされている設定

Maven リポジトリー
Data Grid 8 は、以下のコンポーネントの Red Hat カスタマーポータルからの個別のダウンロードを提供しなくなりました。

- 以前のバージョンではライブラリーモードと呼ばれていた、カスタムアプリケーションで埋め込みキャッシュを作成するための Data Grid コアライブラリー。
- Hot Rod Java クライアント。
- **StoreMigrator** などのユーティリティー。

これらのコンポーネントをダウンロードとして利用できるようにする代わりに、Data Grid は Maven リポジトリーを介して Java アーティファクトを提供します。この変更は、Maven を使用して依存関係を一元管理できることを意味します。これにより、プロジェクト間の依存関係をより適切に制御できます。

顧客ポータルから Data Grid Maven リポジトリーをダウンロードするか、パブリック Red Hat Enterprise Maven リポジトリーから Data Grid 依存関係をプルできます。両方の方法の説明は、データグリッドのドキュメントに記載されています。

- Data Grid Maven リポジトリーの設定

データグリッドサーバー
Data Grid Server は、ダウンロードしてホストファイルシステムに抽出できるアーカイブとして配布されます。

アーカイブ配布には、次の最上位フォルダーが含まれています。

```
├── bin 1
├── boot 2
├── docs 3
├── lib 4
├── server 5
└── static 6
```

1. データグリッドサーバーとデータグリッドコマンドラインインターフェイス (CLI) を起動および管理するためのスクリプト。
2. ブートライブラリー。
3. Data Grid Server の設定と実行に役立つリソース。
4. Data Grid Server のランタイムライブラリー。このディレクトリーは、カスタムコードライブラリーではなく、内部コードのみを対象としていることに注意してください。
5. Data Grid Server インスタンスのルートディレクトリー。
6. Data Grid Console コンソールの静的リソース。

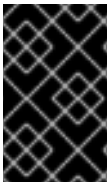
server ディレクトリーは、Data Grid Server インスタンスのルートディレクトリーで、カスタムコードライブラリー、設定ファイル、およびデータのサブディレクトリーが含まれています。

ファイルシステムとディストリビューションの内容の詳細については、**データグリッドサーバーガイド**を参照してください。

- Data Grid サーバーファイルシステム
- Data Grid サーバー README

JBossEAP のモジュール

Red Hat JBoss EAP (EAP) のモジュールを使用して、データグリッドキャッシング機能を EAP アプリケーションに組み込むことができます。



重要

EAP 7.4 では、アプリケーションは、データグリッドモジュールを個別にインストールすることなく、**infinispan** サブシステムを直接処理できます。EAP 7.4 GA がリリースされた後では、Data Grid はダウンロード用の EAP モジュールを提供しなくなります。

独自のデータグリッドモジュールを構築して使用する場合でも、Red Hat はサポートを提供します。ただし、Red Hat では、モジュールが次の理由から、EAP 7.4 で直接 Data Grid API を使用することを推奨します。

- EAP アプリケーション間で共有される集中管理されたデータグリッド設定を使用することはできません。
  モジュールを使用するには、アプリケーションの JAR または WAR 内に設定を格納する必要があります。
- 多くの場合、Java クラスのロードの問題が発生し、実装にデバッグと追加のオーバーヘッドが必要になります。

Data Grid が提供する EAP モジュールの詳細については、**Data Grid 開発者ガイド**を参照してください。

- Red Hat JBoss EAP のデータグリッドモジュール

Tomcat セッションクライアント

Tomcat セッションクライアントを使用すると、Apache Tomcat **org.apache.catalina.Manager** インターフェイスを介して、JBoss Web Server (JWS) アプリケーションからデータグリッドに HTTP セッションを外部化できます。

- JBoss Web Server (JWS) からデータグリッドへの HTTP セッションの外部化

Hot Rod Node.js クライアント

Hot Rod Node.js クライアントは、Data Grid Server クラスターで使用するためのリファレンス JavaScript 実装です。

- Hot RodNode.js クライアント API

ソースコード

各 Data Grid リリースのコンパイルされていないソースコード。

第2章 DATA GRID SERVER デプロイメントの移行

このセクションの詳細を確認して、Data Grid サーバーの正常な移行を計画および準備します。

2.1. DATA GRID SERVER 8

Data Grid Server 8 は次のとおりです。

- 最新のシステムアーキテクチャー向けに設計されています。
- コンテナープラットフォーム用に構築されています。
- Quarkus を使用したネイティブイメージのコンパイル用に最適化されています。

クラウドネイティブアーキテクチャーへの移行は、Data Grid Server 8 が RedHat JBoss Enterprise Application Platform (EAP) に基づいていないことを意味します。代わりに、Data Grid Server 8 は Netty プロジェクトのクライアント/サーバーフレームワークに基づいています。

EAP との統合によって提供された機能の多くは、Data Grid 8 に関連しなくなったか、変更されたため、この変更は以前のバージョンからの移行に影響します。

たとえば、サーバー設定の複雑さは以前のリリースと比較して大幅に軽減されていますが、既存の設定を新しいスキーマに適合させる必要があります。また、Data Grid 8 は、はるかにきめ細かい設定を実現できた以前のバージョンよりも、サーバー設定に関する規則を提供します。さらに、Data Grid Server はドメインモードを利用して設定を一元管理しなくなりました。

Data Grid チームは、これらの設定変更により、既存のクラスターを Data Grid 8 に移行するためにお客様に追加の労力がかかることを認識しています。

Red Hat OpenShift などのコンテナーオーケストレーションプラットフォームを使用して、Red Hat Ansible などの自動化エンジンとともにデータグリッドクラスターをプロビジョニングおよび管理し、データグリッド設定を管理することを推奨します。これらのテクノロジーは、データグリッドに固有のソリューションではなく、より汎用的で複数の異種システムに適しているという点で、より高い柔軟性を提供します。

Data Grid 8 への移行に関しては、Red Hat Ansible のようなソリューションが大規模な設定のデプロイメントに役立つことは注目に値します。ただし、そのツールは、既存のデータグリッド設定の実際の移行を必ずしも支援するとは限りません。

2.2. DATA GRID SERVER の設定

Data Grid は、利用可能なコンピューティングリソースをインテリジェントかつ効率的に利用できるようにするスケーラブルなデータレイヤーを提供します。Data Grid Server のデプロイメントでこれを実現するために、設定は動的と静的の 2 つのレイヤーに分けられます。

動的設定
動的設定は変更可能であり、実行時にキャッシュを作成し、クラスターにノードを追加したり、クラスターからノードを削除したりすると変更されます。

Data Grid Server クラスターをデプロイした後、Data Grid CLI、Data Grid Console、または Hot Rod および REST エンドポイントを介してキャッシュを作成します。Data Grid Server は、ノード間で分散されるクラスター状態の一部として、これらのキャッシュを永続的に保存します。参加している各ノードは、変更が発生すると Data Grid Server がすべてのノード間で自動的に同期する完全なクラスター状態を受け取ります。

静的設定
静的設定は不変であり、実行時に変更されません。

静的設定は、クラスタートランスポート、認証と暗号化、共有データソースなどの基盤となるメカニズムを設定するときに定義します。

デフォルトでは、Data Grid Server は静的設定に **$RHDG_HOME/server/conf/infinispan.xml** を使用します。

設定のルート要素は **infinispan** であり、次の 2 つの基本スキーマを宣言します。

```
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:11.0 https://infinispan.org/schemas/infinispan-config-11.0.xsd
                          urn:infinispan:server:11.0 https://infinispan.org/schemas/infinispan-server-11.0.xsd"
      xmlns="urn:infinispan:config:11.0"
      xmlns:server="urn:infinispan:server:11.0">
```

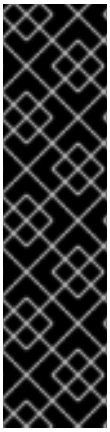
- **urn:infinispan:config** スキーマは、キャッシュコンテナーなどのコア Infinispan 機能の設定を検証します。
- **urn:infinispan:server** スキーマは、Data GridServer の設定を検証します。

キャッシュコンテナーの設定
cache-container 要素を使用して、キャッシュライフサイクルを管理するメカニズムを提供する **CacheManager** インターフェーイスを設定します。

```
<!-- Creates a cache manager named "default" that exports statistics. -->
<cache-container name="default"
                 statistics="true">
  <!-- Defines cluster transport properties, including the cluster name. -->
  <!-- Uses the default TCP stack for inter-cluster communication. -->
  <transport cluster="${infinispan.cluster.name}"
             stack="${infinispan.cluster.stack:tcp}"
             node-name="${infinispan.node.name:}"/>
</cache-container>
```

cache-container 要素は、次の設定要素も保持できます。

- キャッシュマネージャーの **security**。
- MicroProfile 互換性メトリックの **metrics**。
- JMX の監視と管理のための **jmx**。



重要

以前のバージョンでは、データグリッド設定で複数の **cache-container** 要素を定義して、さまざまなエンドポイントでキャッシュコンテナーを公開できました。

Data Grid 8 では、Data Grid CLI とコンソールがクラスターごとに 1 つのキャッシュマネージャーしか処理できないため、複数のキャッシュコンテナーを設定しないでください。ただし、必要に応じて、キャッシュコンテナーの名前をデフォルトよりも環境にとって意味のある名前に変更できます。

キャッシュマネージャーが相互に干渉しないように、マルチテナンシーを実現するには、個別の Data Grid クラスターを使用する必要があります。

サーバー設定
server 要素を使用して、基盤となる Data Grid Server メカニズムを設定します。

```
<server>
 <interfaces>
  <interface name="public"> 1
   <inet-address value="${infinispan.bind.address:127.0.0.1}"/> 2
  </interface>
 </interfaces>

 <socket-bindings default-interface="public" 3
                 port-offset="${infinispan.socket.binding.port-offset:0}"> 4
  <socket-binding name="default" 5
                  port="${infinispan.bind.port:11222}"/> 6
  <socket-binding name="memcached" port="11221"/> 7
 </socket-bindings>

 <security>
   <security-realms> 8
     <security-realm name="default"> 9
       <server-identities> 10
         <ssl>
           <keystore path="application.keystore" 11
                     keystore-password="password"
                     alias="server"
                     key-password="password"
                     generate-self-signed-certificate-host="localhost"/>
         </ssl>
       </server-identities>
       <properties-realm groups-attribute="Roles"> 12
         <user-properties path="users.properties" 13
                          relative-to="infinispan.server.config.path"
                          plain-text="true"/> 14
         <group-properties path="groups.properties" 15
                           relative-to="infinispan.server.config.path" />
       </properties-realm>
     </security-realm>
   </security-realms>
 </security>
```

```
    <endpoints socket-binding="default" security-realm="default" /> (16)
</server>
```

(1) サーバーをネットワーク上で利用できるようにする public という名前のインターフェイスを作成します。

(2) パブリックインターフェイスに **127.0.0.1** ループバックアドレスを使用します。

(3) パブリックインターフェイスを、Data Grid Server エンドポイントが着信クライアント接続をリッスンするネットワークポートにバインドします。

(4) ネットワークポートのオフセットを **0** に指定します。

(5) default という名前のソケットバインディングを作成します。

(6) ソケットバインディング用の ポート **11222** を指定します。

(7) ポート **11221** に Memcached コネクターのソケットバインディングを作成します。

(8) エンドポイントをネットワーク侵入から保護するセキュリティーレルムを定義します。

(9) default という名前のセキュリティーレルムを作成します。

(10) ID 検証用に SSL/TLS キーストアを設定します。

(11) サーバー証明書を含むキーストアを指定します。

(12) プロパティーファイルを使用して、ユーザーをロールにマップするユーザーとグループを定義するように、デフォルトのセキュリティーレルムを設定します。

(13) データグリッドユーザーを含むプロパティーファイルに名前を付けます。

(14) **users.properties** ファイルの内容をプレーンテキストとして保存することを指定します。

(15) データグリッドユーザーをロールにマップするプロパティーファイルに名前を付けます。

(16) Hot Rod および REST コネクターでエンドポイントを設定します。

この例では、Data Grid 8.2 のデフォルトである暗黙の **hotrod-connector** と **rest-connector** 要素を示しています。
8.0 および 8.1 の Data Grid Server 設定は、明示的に宣言された Hot Rod および REST コネクターを使用します。

関連情報

- Data Grid Server ガイド
- Data Grid Server リファレンス

2.3. DATA GRID SERVER のエンドポイントとネットワーク設定

このセクションでは、以前のバージョンから移行する場合の Data Grid Server エンドポイントとネットワーク設定について説明します。

Data Grid 8 は、単一のネットワークインターフェイスとポートを使用してネットワーク上のエンドポイントを公開することにより、サーバーエンドポイントの設定を簡素化します。

2.3.1. インターフェイス

インターフェイスは、エンドポイントをネットワークの場所にバインドします。

Data Grid Server 7.x ネットワークインターフェイスの設定
Data Grid 7.x では、サーバー設定でさまざまなインターフェイスを使用して、管理アクセスと管理アクセスをキャッシュアクセスから分離していました。

```
<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

Data Grid Server 8 ネットワークインターフェイスの設定
Data Grid 8 には、管理アクセスと管理アクセス、およびキャッシュアクセス用のすべてのクライアント接続用に 1 つのネットワークインターフェイスがあります。

```
<interfaces>
  <interface name="public">
    <inet-address value="${infinispan.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

2.3.2. ソケットバインディング

ソケットバインディングは、エンドポイントがクライアント接続をリッスンするポートにネットワークインターフェイスをマップします。

Data Grid Server 7.x ソケットバインディングの設定
Data Grid 7.x では、サーバー設定は、管理コンソール用の **9990** やネイティブ管理プロトコル用のポート **9999** など、管理と管理に一意のポートを使用していました。古いバージョンでは、外部 Hot Rod アクセス用の **11222** や REST 用の **8080** など、エンドポイントごとに一意のポートも使用されていました。

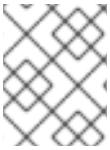
```
<socket-binding-group name="standard-sockets" default-interface="public" port-
offset="${jboss.socket.binding.port-offset:0}">
  <socket-binding name="management-http" interface="management"
port="${jboss.management.http.port:9990}"/>
  <socket-binding name="management-https" interface="management"
port="${jboss.management.https.port:9993}"/>
  <socket-binding name="hotrod" port="11222"/>
  <socket-binding name="hotrod-internal" port="11223"/>
  <socket-binding name="hotrod-multi-tenancy" port="11224"/>
  <socket-binding name="memcached" port="11211"/>
  <socket-binding name="rest" port="8080"/>
  ...
</socket-binding-group>
```

Data Grid Server 8 の単一ポート設定
Data Grid 8 は、単一のポートを使用してサーバーへのすべての接続を処理します。Hot Rod クライアント、REST クライアント、Data Grid CLI、および Data **GridConsole は** すべてポート 11222 を使用します。

```
<socket-bindings default-interface="public"
                 port-offset="${infinispan.socket.binding.port-offset:0}">
  <socket-binding name="default" port="${infinispan.bind.port:11222}"/>
  <socket-binding name="memcached" port="11221"/>
</socket-bindings>
```

2.3.3. エンドポイント

エンドポイントは、リモートクライアント接続をリッスンし、Hot Rod や HTTP (REST) などのプロトコルを介して要求を処理します。



注記

Data Grid CLI は、すべてのキャッシュおよび管理操作に REST エンドポイントを使用します。

Data Grid Server 7.x エンドポイントサブシステム
Data Grid 7.x では、**endpoint** サブシステムを使用して、HotRod および REST エンドポイントのコネクターを設定できます。

```
<subsystem xmlns="urn:infinispan:server:endpoint:9.4">
  <hotrod-connector socket-binding="hotrod" cache-container="local">
    <topology-state-transfer lazy-retrieval="false" lock-timeout="1000" replication-timeout="5000"/>
  </hotrod-connector>
  <rest-connector socket-binding="rest" cache-container="local">
    <authentication security-realm="ApplicationRealm" auth-method="BASIC"/>
  </rest-connector>
</subsystem>
```

Data Grid Server 8 エンドポイント設定
Data Grid 8 は、**endpoint** サブシステムを **endpoint** 要素に置き換えます。**hotrod-connector** と **rest-connector** の設定要素と属性は、以前のバージョンと同じです。

```
<endpoints socket-binding="default" security-realm="default">
  <hotrod-connector name="hotrod"/>
  <rest-connector name="rest"/>
</endpoints>
```

関連情報

- Data Grid Server ガイド

2.4. DATA GRID SERVER のセキュリティー

Data Grid Server のセキュリティーは、認証と暗号化を設定して、ネットワーク攻撃を防ぎ、データを保護します。

2.4.1. セキュリティーレルム

Data Grid 8 では、セキュリティーレルムは暗黙の設定オプションを提供します。これは、以前のバージョンほど多くの設定を提供する必要がないことを意味します。たとえば、Kerberos レルムを定義すると、Kerberos 機能を利用できます。トラストストアを追加すると、証明書認証を取得します。

Data Grid 7.x には、2 つのデフォルトのセキュリティーレルムがありました。

- **ManagementRealm** は管理 API を保護します。
- **ApplicationRealm** は、エンドポイントとリモートクライアント接続を保護します。

一方、Data Grid 8 は、HotRod および REST エンドポイントに使用できる複数の異なるセキュリティーレルムを定義できる **security** 要素を提供します。

```
<security>
  <security-realms>
    ...
  </security-realms>
</security>
```

サポートされているセキュリティーレルム

- プロパティーレルムは、プロパティーファイル **users.properties** と **groups.properties** を使用して、データグリッドにアクセスできるユーザーとグループを定義します。
- LDAP レルムは、OpenLDAP、Red Hat Directory Server、Apache Directory Server、Microsoft Active Directory などの LDAP サーバーに接続して、ユーザーを認証し、メンバーシップ情報を取得します。
- トラストストアレルムは、データグリッドへのアクセスが許可されているすべてのクライアントの公開証明書を含むキーストアを使用します。
- トークンレルムは外部サービスを使用してトークンを検証し、Red Hat SSO などの RFC-7662 (OAuth2 トークンイントロスペクション) と互換性のあるプロバイダーを必要とします。

2.4.2. サーバー ID

サーバー ID は、証明書チェーンを使用して、データグリッドサーバー ID をリモートクライアントに証明します。

Data Grid 8 は、以前のバージョンと同じ設定を使用して SSL ID を定義しますが、使いやすさが向上しています。

- セキュリティーレルムに SSLID が含まれている場合、Data Grid はそのセキュリティーレルムを使用するエンドポイントの暗号化を自動的に有効にします。
- テストおよび開発環境の場合、Data Grid には、起動時にキーストアを自動的に生成する **generate-self-signed-certificate-host** 属性が含まれています。

```
<security-realm name="default">
  <server-identities>
    <ssl>
      <keystore path="..."
                relative-to="..."
                keystore-password="..."
```

```
            alias="..."
            key-password="..."
            generate-self-signed-certificate-host="..."/>
    </ssl>
  </server-identities>
  ...
<security-realm>
```

2.4.3. エンドポイント認証メカニズム

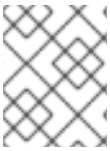
Hot Rod および REST エンドポイントは、SASL または HTTP メカニズムを使用してクライアント接続を認証します。

Data Grid 8 は、Data Grid 7.x 以前と同じ **authentication** 要素を **hotrod-connector** および **rest-connector** 設定に使用します。

```
<hotrod-connector name="hotrod">
  <authentication>
    <sasl mechanisms="..." server-name="..."/>
  </authentication>
</hotrod-connector>
<rest-connector name="rest">
  <authentication>
    <mechanisms="..." server-principal="..."/>
  </authentication>
</rest-connector>
```

以前のバージョンとの主な違いの 1 つは、Data Grid 8 がエンドポイントの追加の認証メカニズムをサポートしていることです。

Hot Rod SASL 認証メカニズム
Hot Rod クライアントは、**DIGEST-MD5** ではなく **SCRAM-SHA-512** をデフォルトの認証メカニズムとして使用するようになりました。



注記

プロパティーセキュリティーレルムを使用する場合は、**PLAIN** 認証メカニズムを使用する必要があります。

認証メカニズム	説明	関連する詳細
PLAIN	プレーンテキスト形式の認証情報を使用します。**PLAIN** 認証は、暗号化された接続でのみ使用する必要があります。	**Basic** HTTP メカニズムに似ています。

認証メカニズム	説明	関連する詳細
DIGEST-*	ハッシュアルゴリズムとナンス値を使用します。ホットロッドコネクターは、強度の順に、**DIGEST-MD5**、**DIGEST-SHA**、**DIGEST-SHA-256**、**DIGEST-SHA-384**、および **DIGEST-SHA-512** ハッシュアルゴリズムをサポートします。	**Digest** HTTP メカニズムに似ています。
SCRAM-*	ハッシュアルゴリズムとナンス値に加えて**ソルト**値を使用します。ホットロッドコネクターは、**SCRAM-SHA**、**SCRAM-SHA-256**、**SCRAM-SHA-384**、および **SCRAM-SHA-512** ハッシュアルゴリズムを強度順にサポートします。	**Digest** HTTP メカニズムに似ています。
GSSAPI	Kerberos チケットを使用し、Kerberos ドメインコントローラーが必要です。対応する **Kerberos** サーバー ID をレルム設定に追加する必要があります。ほとんどの場合、ユーザーメンバーシップ情報を提供するために **ldap-realm** も指定します。	**SPNEGO** HTTP メカニズムに似ています。
GS2-KRB5	Kerberos チケットを使用し、Kerberos ドメインコントローラーが必要です。対応する **Kerberos** サーバー ID をレルム設定に追加する必要があります。ほとんどの場合、ユーザーメンバーシップ情報を提供するために **ldap-realm** も指定します。	**SPNEGO** HTTP メカニズムに似ています。
EXTERNAL	クライアント証明書を使用します。	**CLIENT_CERTHTTP** メカニズムに似ています。
OAUTHBEARER	OAuth トークンを使用し、**token-realm** 設定が必要です。	**BEARER_TOKEN** HTTP メカニズムに似ています。

HTTP (REST) 認証メカニズム

認証メカニズム	説明	関連する詳細

認証メカニズム	説明	関連する詳細
BASIC	プレーンテキスト形式の認証情報を使用します。暗号化された接続でのみ **BASIC** 認証を使用する必要があります。	HTTP **Basic** HTTP 認証方式に対応し、**PLAIN** SASL メカニズムと同様です。
DIGEST	ハッシュアルゴリズムとナンス値を使用します。REST コネクターは、**SHA-512**、**SHA-256**、および **MD5** ハッシュアルゴリズムをサポートします。	**Digest** HTTP 認証スキームに対応し、**DIGEST-*** SASL メカニズムに似ています。
SPNEGO	Kerberos チケットを使用し、Kerberos ドメインコントローラーが必要です。対応する **Kerberos** サーバー ID をレルム設定に追加する必要があります。ほとんどの場合、ユーザーメンバーシップ情報を提供するために **ldap-realm** も指定します。	**Negotiate** HTTP 認証スキームに対応し、**GSSAPI** および **GS2-KRB5SASL** メカニズムに類似しています。
BEARER_TOKEN	OAuth トークンを使用し、**token-realm** 設定が必要です。	**Bearer** HTTP 認証スキームに対応し、**OAUTHBEARERSASL** メカニズムに似ています。
CLIENT_CERT	クライアント証明書を使用します。	**EXTERNAL** SASL メカニズムに似ています。

2.4.4. EAP アプリケーションの認証

EAP アプリケーションクラスパスの **hotrod-client.properties** にクレデンシャルを追加して、次の方法でデータグリッドで認証できるようになりました。

- リモートキャッシュコンテナー (**remote-cache-container**)
- リモートストア (**remote-store**)
- EAP モジュール

2.4.5. ロギング

Data Grid は、JBoss Log Manager に基づいていた以前のバージョンのロギングサブシステムの代わりに Apache Log4j2 を使用します。

デフォルトでは、DataGrid はログメッセージを次のディレクトリーに書き込みます。
$RHDG_HOME/${infinispan.server.root}/log

server.log はデフォルトのログファイルです。

アクセスログ

以前のバージョンでは、Data Grid にキャッシュのセキュリティーログを監査するためのロガーが含まれていました。

```
<authorization audit-logger="org.infinispan.security.impl.DefaultAuditLogger">
```

Data Grid 8 は、この監査ロガーを提供しなくなりました。

ただし、Hot Rod エンドポイントと REST エンドポイントのログカテゴリーを使用できます。

- **org.infinispan.HOTROD_ACCESS_LOG**
- **org.infinispan.REST_ACCESS_LOG**

関連情報

- Data Grid Server ガイド

2.5. DATA GRID SERVER エンドポイントの分離

以前のバージョンから移行する場合、既存の設定に一致するように、Data Grid エンドポイントに異なるネットワークロケーションを作成できます。ただし、Data Grid アーキテクチャーが変更され、すべてのクライアント接続に単一のポートを使用するようになったため、以前のバージョンのすべてのオプションを使用できるわけではありません。

重要

Data Grid CLI やコンソールなどの管理ツールは REST API を使用します。Data Grid CLI とコンソールを無効にせずに、エンドポイント設定から REST API を削除することはできません。同様に、REST エンドポイントを分離して、キャッシュアクセスと管理アクセスに異なるポートまたはソケットバインディングを使用することはできません。

手順

1. REST エンドポイントと Hot Rod エンドポイントに別々のネットワークインターフェイスを定義します。
   たとえば、Hot Rod エンドポイントを外部に公開するための "public" インターフェイスと、アクセスが制限されているネットワークロケーション上の REST エンドポイントを公開するための "private" インターフェイスを定義します。

   ```
   <interfaces>
     <interface name="public">
       <inet-address value="${infinispan.bind.address:198.51.100.0}"/>
     </interface>
     <interface name="private">
       <inet-address value="${infinispan.bind.address:192.0.2.0}"/>
     </interface>
   </interfaces>
   ```

   この設定により、以下が作成されます。

   - **198.51.100.0** IP アドレスを持つ "public" インターフェイス。
   - **192.0.2.0** IP アドレスを持つ "private" インターフェイス。

2. 次の例のように、エンドポイントに個別のソケットバインディングを設定します。

```
<socket-bindings default-interface="private"
                 port-offset="${infinispan.socket.binding.port-offset:0}">
  <socket-binding name="default"
                  port="${infinispan.bind.port:8080}"/>
  <socket-binding name="hotrod"
                  interface="public"
                  port="11222"/>
</socket-bindings>
```

この例では、

- プライベートインターフェイスをソケットバインディングのデフォルトとして設定します。
- ポート **8080** を使用するデフォルトのソケットバインディングを作成します。
- パブリックインターフェイスとポート **11222** を使用する hotrod ソケットバインディングを作成します。

3. 次に、エンドポイント用に個別のセキュリティーレルムを作成します。

```
<security>
  <security-realms>
    <security-realm name="truststore">
      <server-identities>
        <ssl>
          <keystore path="server.p12"
                    relative-to="infinispan.server.config.path"
                    keystore-password="secret"
                    alias="server"/>
        </ssl>
      </server-identities>
      <truststore-realm path="trust.p12"
                        relative-to="infinispan.server.config.path"
                        keystore-password="secret"/>
    </security-realm>
    <security-realm name="kerberos">
      <server-identities>
        <kerberos keytab-path="http.keytab"
                  principal="HTTP/localhost@INFINISPAN.ORG"
                  required="true"/>
      </server-identities>
    </security-realm>
  </security-realms>
</security>
```

この例では、

- トラストストアのセキュリティーレルムを設定します。
- Kerberos セキュリティーレルムを設定します。

4. 次のようにエンドポイントを設定します。

```
<endpoints socket-binding="default"
           security-realm="kerberos">
```

```
    <hotrod-connector name="hotrod"
                      socket-binding="hotrod"
                      security-realm="truststore"/>
    <rest-connector name="rest"/>
  </endpoints>
```

5. Data Grid Server を起動します。
   ログには、エンドポイントがクライアント接続を受け入れるネットワークの場所を示す次のメッセージが含まれます。

```
[org.infinispan.SERVER] ISPN080004: Protocol HotRod listening on 198.51.100.0:11222
[org.infinispan.SERVER] ISPN080004: Protocol SINGLE_PORT listening on 192.0.2.0:8080
[org.infinispan.SERVER] ISPN080034: Server '<hostname>' listening on http://192.0.2.0:8080
```

次のステップ

1. **http://192.0.2.0:8080** の任意のブラウザーから Data Grid Console にアクセスします
2. カスタムの場所で接続するように Data Grid CLI を設定します。次に例を示します。

```
$ bin/cli.sh -c http://192.0.2.0:8080
```

2.6. DATA GRID SERVER 共有データソース

Data Grid 7.x JDBC キャッシュストアは、**PooledConnectionFactory** を使用してデータベース接続を取得できます。

Data Grid 8 を使用すると、サーバー設定でマネージドデータソースを作成して、JDBC キャッシュストアを使用した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できます。

データソース設定は、次の 2 つのセクションで設定されています。

- データベースへの接続方法を定義する **connection factory**。
- 接続をプールして再利用する方法を定義し、Agroal に基づく **connection pool**。

最初にサーバー設定でデータソース接続ファクトリーと接続プールを定義し、次にそれを JDBC キャッシュストア設定に追加します。

JDBC キャッシュストアの移行の詳細については、このドキュメントの**キャッシュストアの移行**セクションを参照してください。

関連情報

- Data Grid Server ガイド

2.7. DATA GRID SERVER の JMX とメトリック

Data Grid 8 は、Prometheus などのメトリックツールと統合するために JMX と **/metrics** エンドポイントの両方を介してメトリックを公開します。

/metrics エンドポイントは以下を提供します。

- JVM の稼働時間やキャッシュ操作の平均秒数などの値を返すゲージ。

- 読み取り、書き込み、および削除操作にかかる時間をパーセンタイルで示すヒストグラム。

以前のバージョンでは、Prometheus メトリックは、ネイティブでサポートされるのではなく、JMX メトリックをマップするエージェントによって収集されていました。

以前のバージョンの DataGrid は、JBoss Operations Network (JON) プラグインを使用して、メトリックを取得し、操作を実行していました。Data Grid 8 は JON プラグインを使用しなくなりました。

Data Grid 8 は、JMX と Prometheus のメトリックをキャッシュマネージャーとキャッシュレベルの設定に分離します。

```
<cache-container name="default"
                 statistics="true"> 1
  <jmx enabled="true" /> 2
</cache-container>
```

1 キャッシュマネージャーの統計を有効にします。これはデフォルトになります。

2 すべての統計と操作を含む JMX MBean をエクスポートします。

```
<distributed-cache name="mycache" statistics="true" /> 1
```

1 キャッシュの統計を有効にします。

関連情報

- Data Grid Server ガイド

2.8. DATA GRID SERVER のチートシート

次のコマンドと例は、Data GridServer を操作するためのクイックリファレンスとして使用してください。

サーバーインスタンスの起動

- Linux

```
$ bin/server.sh
```

- Microsoft Windows

```
$ bin\server.bat
```

CLI の開始

- Linux

```
$ bin/cli.sh
```

- Microsoft Windows

```
$ bin\cli.bat
```

ユーザーの作成

- Linux

```
$ bin/cli.sh user create myuser -p "qwer1234!"
```

- Microsoft Windows

```
$ bin\cli.bat user create myuser -p "qwer1234!"
```

サーバーインスタンスの停止

- 単一サーバーインスタンス

```
[//containers/default]> shutdown server $hostname
```

- クラスター全体

```
[//containers/default]> shutdown cluster
```

使用可能なコマンドオプションのリスト表示

-h フラグを使用して、サーバーを実行するために使用可能なコマンドオプションをリスト表示します。

- Linux

```
$ bin/server.sh -h
```

- Microsoft Windows

```
$ bin\server.bat -h
```

7.x から 8 のリファレンス

7.x	8.x
./standalone.sh -c clustered.xml	**./server.sh**
./standalone.sh	**./server.sh -c infinispan-local.xml**
-Djboss.default.multicast.address=234.99.54.20	**-Djgroups.mcast_addr=234.99.54.20**
-Djboss.bind.address=172.18.1.13	**-Djgroups.bind.address=172.18.1.13**
-Djboss.default.jgroups.stack=udp	**-j udp**

関連情報

- Data Grid Server ガイド

第3章 DATA GRI 設定の移行

Data Grid 8 への移行に影響するデータグリッド設定の変更を見つけます。

3.1. DATA GRID キャッシュの設定

Data Grid 8 は、デフォルトで空のキャッシュコンテナーを提供します。Data Grid を起動すると、キャッシュマネージャーがインスタンス化されるため、実行時にキャッシュを作成できます。

ただし、以前のバージョンと比較すると、すぐに使用できるデフォルトのキャッシュはありません。

Data Grid 8 では、**CacheContainerAdmin** API を介して作成するキャッシュは、クラスターの再起動後も存続することを保証するために永続的です。

永続的なキャッシュ

```
.administration()
  .withFlags(AdminFlag.PERMANENT) 1
  .getOrCreateCache("myPermanentCache", "org.infinispan.DIST_SYNC");
```

1 **AdminFlag.PERMANENT** はデフォルトで有効になっており、キャッシュが再起動後も存続するようになっています。

キャッシュを作成するときにこのフラグを設定する必要はありません。ただし、データが再起動後も存続するためには、データグリッドに永続ストレージを個別に追加する必要があります。次に例を示します。

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
  .addSingleFileStore()
  .location("/tmp/myDataStore")
  .maxEntries(5000);
```

揮発性キャッシュ

```
.administration()
  .withFlags(AdminFlag.VOLATILE) 1
  .getOrCreateCache("myTemporaryCache", "org.infinispan.DIST_SYNC"); 2
```

1 データグリッドの再起動時にキャッシュが失われるように **VOLATILE** フラグを設定します。

2 myTemporaryCache という名前のキャッシュを返すか、**DIST_SYN**C テンプレートを使用してキャッシュを作成します。

Data Grid 8 は、推奨設定でキャッシュを作成するために使用できるサーバーインストール用のキャッシュテンプレートを提供します。

使用可能なキャッシュテンプレートのリストは、次のように取得できます。

- CLI で **Tab** のオートコンプリートを使用します。

```
[//containers/default]> create cache --template=
```

- REST API を使用します。

```
GET 127.0.0.1:11222/rest/v2/cache-managers/default/cache-configs/templates
```

3.1.1. キャッシュエンコーディング

リモートキャッシュを作成するときは、キーと値の MediaType を設定する必要があります。MediaType を設定すると、データのストレージ形式が保証されます。

キャッシュをエンコードするには、設定で MediaType を指定します。他の要件がない限り、ProtoStream を使用する必要があります。これは、言語に依存しない下位互換性のある形式でデータを格納します。

<encoding media-type="application/x-protostream"/>

エンコーディングを使用した分散キャッシュ設定

```
<infinispan>
   <cache-container>
      <distributed-cache name="myCache" mode="SYNC">
        <encoding media-type="application/x-protostream"/>
         ...
      </distributed-cache>
   </cache-container>
</infinispan>
```

リモートキャッシュをエンコードしない場合、Data Grid Server は次のメッセージをログに記録します。

```
WARN  (main) [org.infinispan.encoding.impl.StorageConfigurationManager] ISPN000599: Configuration for cache 'mycache' does not define the encoding for keys or values. If you use operations that require data conversion or queries, you should configure the cache with a specific MediaType for keys or values.
```

将来のバージョンでは、データ変換が行われる操作にキャッシュエンコーディングが必要になります。たとえば、キャッシュのインデックス作成とデータコンテナーの検索、リモートタスクの実行、Hot Rod および REST エンドポイントからのさまざまな形式でのデータの読み取りと書き込み、リモートフィルター、コンバーター、リスナーの使用などです。

3.1.2. キャッシュヘルスステータス

Data Grid 7.x には、クラスターのヘルスステータスとクラスター内のキャッシュを返すヘルスチェック API が含まれています。

Data Grid 8 は Health API も提供します。組み込みおよびサーバーインストールの場合、次の MBean を使用して JMX 経由で Health API にアクセスできます。

```
org.infinispan:type=CacheManager,name="default",component=CacheContainerHealth
```

Data Grid Server は、REST エンドポイントと Data Grid Console を介して Health API も公開します。

表3.1 ヘルスステータス

7.x	8.x	説明
HEALTHY	**HEALTHY**	キャッシュが期待どおりに動作していることを示します。
Rebalancing	**HEALTHY_REBALANCING**	キャッシュがリバランス状態にあることを示しますが、それ以外場合は想定どおりに動作します。
Unhealthy	**DEGRADED**	キャッシュが期待どおりに動作しておらず、トラブルシューティングが必要な可能性があることを示します。

関連情報

- Data Grid キャッシュの設定

3.1.3. Data Grid 8.1 設定スキーマへの変更

このトピックでは、8.0 から 8.1 までの Data Grid 設定スキーマへの変更をリスト表示します。

新規および変更された要素と属性

- **stack** は、インライン JGroups スタック定義のサポートを追加します。
- **stack.combine** 属性と **stack.position** 属性を使用すると、JGroups スタック定義をオーバーライドおよび変更できます。
- **metric** 使用すると、Data Grid が Eclipse Micro Profile Metrics API と互換性のあるメトリックをエクスポートする方法を設定できます。
- **context-initializer** を使用すると、ユーザータイプの Protostream ベースのマーシャラーを初期化する **SerializationContextInitializer** 実装を指定できます。
- **key-transformers** を使用すると、Lucene でインデックスを作成するためにカスタムキーを文字列に変換するトランスフォーマーを登録できます。
- **statistics** はデフォルトで false になりました。

非推奨の要素と属性
次の要素と属性は非推奨になりました。

- **off-heap** 要素の **address-count** 属性。
- **transaction** 要素の **protocol** 属性。
- **jmx** 要素の **duplicate-domains** 属性。
- **advanced-externalizer**
- **custom-interceptors**
- **state-transfer-executor**

- **transaction-protocol**

削除された要素と属性
次の要素と属性は以前のリリースで非推奨になり、現在は削除されています。

- **deadlock-detection-spin**
- **compatibility**
- **write-skew**
- **versioning**
- **data-container**
- **eviction**
- **eviction-thread-policy**

3.2. エビクション設定

Data Grid 8 は、以前のバージョンと比較して、エビクション設定を簡素化します。ただし、エビクション設定は、データグリッドのさまざまなバージョン間で多数の変更が加えられているため、移行が簡単ではない可能性があります。

注記

Data Grid 7.2 以降、**memory** 要素が設定内の **eviction** 要素に置き換わります。このセクションでは、**memory** 要素のみを使用したエビクション設定について説明します。**eviction** 要素を使用する設定の移行については、Data Grid 7.2 のドキュメントを参照してください。

3.2.1. ストレージタイプ

Data Grid では、次のオプションを使用して、エントリーをメモリーに保存する方法を制御できます。

- オブジェクトを JVM ヒープメモリーに格納します。
- バイトをネイティブメモリー (オフヒープ) に格納します。
- バイトを JVM ヒープメモリーに格納します。

Data Grid 8 の変更
以前の 7.x バージョンおよび 8.0 では、**object**、**binary**、および **off-heap** 要素を使用してストレージタイプを設定していました。

Data Grid 8.1 以降では、**storage** 属性を使用して、オブジェクトを JVM ヒープメモリーに格納するか、バイトとしてオフヒープメモリーに格納します。

JVM ヒープメモリーにバイトを格納するには、**encoding** 要素を使用して、データのバイナリーストレージ形式を指定します。

Data Grid 7.x	Data Grid 8
<memory><object /></memory>	**<memory />**
<memory><off-heap /></memory>	**<memory storage="OFF_HEAP" />**
<memory><binary /></memory>	**<encoding media-type="..." />**

Data Grid 8 のオブジェクトストレージ
デフォルトでは、Data Grid 8.1 はオブジェクトストレージ (JVM ヒープ) を使用します。

```
<distributed-cache>
  <memory />
</distributed-cache>
```

storage="HEAP" を明示的に設定して、データをオブジェクトとして JVM ヒープメモリーに格納することもできます。

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

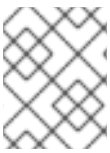
Data Grid 8 のオフヒープストレージ
データをバイトとしてネイティブメモリーに保存するには、**storage** 属性の値として **"OFF_HEAP"** を設定します。

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

オフヒープアドレス数
以前のバージョンでは、**offheap** の **address-count** 属性を使用すると、衝突を回避するためにハッシュマップで使用可能なポインターの数を指定できます。Data Grid 8.1 では、**address-count** は使用されなくなり、オフヒープメモリーは衝突を回避するために動的にサイズ変更されます。

Data Grid 8 のバイナリーストレージ
encording 要素を使用して、キャッシュエントリーのバイナリーストレージ形式を指定します。

```
<distributed-cache>
  <!--Configure MediaType for entries with binary formats.-->
  <encoding media-type="application/x-protostream"/>
  <memory ... />
</distributed-cache>
```



注記

この変更の結果、Data Grid は、**byte[]** と混合されたプリミティブと String を格納しなくなり、**byte[]** のみを格納します。

3.2.2. エビクションしきい値

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。

Data Grid 7. x および 8.0 では、キャッシュ内のエントリーの最大制限を定義する 2 つのエビクションタイプを指定します。

- **COUNT** は、キャッシュ内のエントリーの数を測定します。
- **MEMORY** は、キャッシュ内のすべてのエントリーが占めるメモリーの量を測定します。

設定した設定に応じて、メモリーの数または合計量のいずれかが最大値を超えると、Data Grid は未使用のエントリーを削除します。

Data Grid 7.x および 8.0 も長いようなデータコンテナーのサイズを規定する **size** 属性を使用します。設定するストレージの種類に応じて、エントリーの数またはメモリーの量が **size** 属性の値を超えると、エビクションが発生します。

Data Grid 8.1 では、**size** 属性は **COUNT** および **MEMORY** とともに非推奨になりました。代わりに、次の 2 つの方法のいずれかでデータコンテナーの最大サイズを設定します。

- **max-count** 属性を持つエントリーの総数。
- **max-size** 属性を持つメモリーの最大量 (バイト単位)。

エントリーの総数に基づくエビクション

```
<distributed-cache>
  <memory max-count="..." />
</distributed-cache>
```

最大メモリー量に基づくエビクション

```
<distributed-cache>
  <memory max-size="..." />
</distributed-cache>
```

3.2.3. エビクションストラテジー

エビクション戦略は、Data Grid がエビクションを実行する方法を制御します。

Data Grid 7.x および 8.0 では、**strategy** 属性を使用して次のエビクション戦略のいずれかを設定できます。

ストラテジー	説明
NONE	Data Grid はエントリーを削除しません。これは、エビクションを設定しない限り、デフォルト設定です。
REMOVE	Data Grid は、キャッシュが設定されたサイズを超えないように、メモリーからエントリーを削除します。これは、エビクションを設定するときのデフォルト設定です。

ストラテジー	説明
MANUAL	Data Grid は削除を実行しません。削除は、**evict()** メソッドを **Cache** API から呼び出すことによって手動で行われます。
EXCEPTION	Data Grid は、設定されたサイズを超える場合、キャッシュに新しいエントリーを書き込みません。キャッシュに新しいエントリーを書き込む代わりに、Data Grid は **ContainerFullException** 出力します。

Data Grid 8.1 では、以前のバージョンと同じ戦略を使用できます。ただし、**strategy** 属性は **when-full** 属性に置き換えられます。

```
<distributed-cache>
  <memory when-full="<eviction_strategy>" />
</distributed-cache>
```

エビクションアルゴリズム
Data Grid 7.2 では、エビクションアルゴリズムを設定する機能は、Low Inter-Reference Recency Set (LIRS) とともに非推奨になりました。

バージョン 7.2 以降、Data Grid には、TinyLFU と呼ばれる Least Frequently Used (LFU) キャッシュ置換アルゴリズムのバリエーションを実装する Caffeine キャッシングライブラリーが含まれています。オフヒープストレージの場合、Data Grid は LeastRecent Used (LRU) アルゴリズムのカスタム実装を使用します。

3.2.4. エビクション設定の比較

異なる Data Grid バージョン間でエビクション設定を比較します。

オブジェクトストレージとエントリー数の排除
7.2 から 8.0

```
<memory>
  <object size="1000000" eviction="COUNT" strategy="REMOVE"/>
</memory>
```

8.1

```
<memory max-count="1MB" when-full="REMOVE"/>
```

オブジェクトストレージとメモリー量の排除
7.2 から 8.0

```
<memory>
  <object size="1000000" eviction="MEMORY" strategy="MANUAL"/>
</memory>
```

8.1

```
<memory max-size="1MB" when-full="MANUAL"/>
```

バイナリーストレージとエントリー数の排除
7.2 から 8.0

```
<memory>
  <binary size="500000000" eviction="MEMORY" strategy="EXCEPTION"/>
</memory>
```

8.1

```
<cache>
   <encoding media-type="application/x-protostream"/>
   <memory max-size="500 MB" when-full="EXCEPTION"/>
</cache>
```

バイナリーストレージとメモリー量の排除
7.2 から 8.0

```
<memory>
  <binary size="500000000" eviction="COUNT" strategy="MANUAL"/>
</memory>
```

8.1

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

オフヒープストレージとエントリー数の排除
7.2 から 8.0

```
<memory>
  <off-heap size="10000000" eviction="COUNT"/>
</memory>
```

8.1

```
<memory storage="OFF_HEAP" max-count="10MB"/>
```

オフヒープストレージとメモリー量の排除
7.2 から 8.0

```
<memory>
  <off-heap size="1000000000" eviction="MEMORY"/>
</memory>
```

8.1

```
<memory storage="OFF_HEAP" max-size="1GB"/>
```

関連情報

- Configuring Data Grid caches

- RHDG 7.3 以降の新しいエビクションポリシー TinyLFU (Red Hat ナレッジベース)
- Data Grid 7.2 の製品ドキュメント

3.3. 有効期限の設定

有効期限は、ライフスパンまたは最大アイドル時間に基づいてキャッシュからエントリーを削除します。

設定を Data Grid 7.x から 8 に移行する場合、有効期限のために行う必要のある変更はありません。設定は同じままです。

ライフスパンの満了

```
<expiration lifespan="1000" />
```

最大アイドル有効期限

```
<expiration max-idle="1000" interval="120000" />
```

Data Grid 7.2 以前の場合、クラスター化されたキャッシュで **max-idle** を使用すると、技術的な制限があり、パフォーマンスが低下しました。

Data Grid 7.3 以降、クライアントが **max-idle** 有効期限値を持つエントリーを読み取ると、Data Grid はクラスター化されたキャッシュ内のすべての所有者にタッチコマンドを送信します。これにより、エントリーの相対アクセス時間がクラスター全体で同じになります。

Data Grid 8 は、クラスター間で **max-idle** の有効期限について同じタッチコマンドを送信します。ただし、**max-idle** の使用を開始する前に、考慮すべき技術的な考慮事項がいくつかあります。有効期限の仕組みの詳細と、タッチコマンドがクラスター化されたキャッシュのパフォーマンスにどのように影響するかを確認するには、**Configuring Data Grid caches** を参照してください。

関連情報

- Configuring Data Grid caches

3.4. 永続キャッシュストア

Data Grid 7.x と比較すると、Data Grid 8 のキャッシュストア設定にいくつかの変更があります。

永続性 SPI
Data Grid 8.1 では、キャッシュストア用の **NonBlockingStore** インターフェイスが導入されています。**NonBlockingStore** SPI は、呼び出し元のスレッドをブロックしてはならないメソッドを公開します。

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

ブロッキング操作を使用するカスタムキャッシュストア実装の場合、Data Grid は それらの操作を処理するための **BlockingManager** ユーティリティークラスを提供します。

NonBlockingStore インターフェイスの導入により、以下のインターフェイスは非推奨になります。

- **CacheLoader**

- **CacheWriter**
- **AdvancedCacheLoader**
- **AdvancedCacheWriter**

カスタムキャッシュストア

Data Grid 8 では、以前のバージョンと同様に、**store** 要素を使用してカスタムキャッシュストアを設定できます。

次の変更が適用されます。

- **singleton** 属性が削除されます。代わりに **shared=true** を使用してください。
- **segmented** 属性が追加され、デフォルトで **true** になります。

セグメント化されたキャッシュストア

Data Grid 8 の時点で、キャッシュストア設定はデフォルトで **segmented="true"** になり、次のキャッシュストア要素に適用されます。

- **store**
- **file-store**
- **string-keyed-jdbc-store**
- **jpa-store**
- **remote-store**
- **rocksdb-store**
- **soft-index-file-store**

単一ファイルキャッシュストア

シングルファイルキャッシュストアの **relative-to** 属性は、Data Grid 8 削除されています。キャッシュストア設定にこの属性が含まれている場合、Data Grid はそれを無視し、**path** 属性のみを使用してストアの場所を設定します。

JDBC キャッシュストア

JDBC キャッシュストアには、**xlmns** 名前空間宣言を含める必要があります。これは一部の Data Grid 7.x バージョンでは必要ありませんでした。

```
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:11.0" shared="true">
    ...
</persistence>
```

JDBC 接続ファクトリー

Data Grid 7.x JDBC キャッシュストアは、次の **ConnectionFactory** 実装を使用してデータベース接続を取得できます。

- **ManagedConnectionFactory**
- **SimpleConnectionFactory**
- **PooledConnectionFactory**

Data Grid 8 は、Red Hat JBoss EAP と同じ Agroal に基づく接続ファクトリーを使用してデータベースに接続するようになりました。**c3p0.properties** および **hikari.properties** ファイルを使用できなくなりました。

セグメンテーション
今デフォルトでセグメンテーションを可能に JDBC 文字列ベースのキャッシュストアの設定は、次のプログラムの例のように 、**segmentColumnName** と **segmentColumnType** パラメーターを含める必要があります。

MySQL の例

```
builder.table()
      .tableNamePrefix("ISPN")
      .idColumnName("ID_COLUMN").idColumnType(“VARCHAR(255)”)
      .dataColumnName("DATA_COLUMN").dataColumnType(“VARBINARY(1000)”)
      .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType(“BIGINT”)
      .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INTEGER")
```

PostgreSQL Example

```
builder.table()
      .tableNamePrefix("ISPN")
      .idColumnName("ID_COLUMN").idColumnType(“VARCHAR(255)”)
      .dataColumnName("DATA_COLUMN").dataColumnType(“BYTEA”)
      .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT”)
      .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INTEGER");
```

ライトビハインド
Write Behind モードの **thread-pool-size** 属性は、Data Grid 8 で削除されています。

キャッシュストアとローダーの削除
Data Grid 7.3 は、Data Grid 8 で使用できなくなった次のキャッシュストアとローダーを非推奨にします。

- Cassandra キャッシュストア
- REST キャッシュストア
- LevelDB キャッシュストア
- CLI キャッシュローダー

キャッシュストアマイグレーター
以前のバージョンのデータグリッドのキャッシュストアは、データグリッド 8 と互換性のないバイナリー形式でデータを保存します。

StoreMigrator ユーティリティーを使用して、永続キャッシュストア内のデータを Data Grid 8 に移行します。

3.5. DATA GRID クラスタートランスポート

Data Grid は、JGroups テクノロジーを使用して、クラスター化されたノード間の通信を処理します。

JGroups スタック設定要素と属性は、以前のデータグリッドバージョンから大幅に変更されていません。

以前のバージョンと同様に、Data Grid は、ネットワーク要件に最適化されたカスタムクラストランスポート設定を構築するための開始点として使用できる事前設定済みの JGroups スタックを提供します。同様に、Data Grid は、外部 XML ファイルで定義された JGroups スタックを **infinispan.xml** に追加する機能を提供します。

Data Grid 8 は、クラストランスポート設定を容易にするための使いやすさの向上をもたらしました。

- インラインスタックを使用すると、**jgroups** 要素を使用して **infinispan.xml** 内で直接 JGroups スタックを設定できます。
- **infinispan.xml** 内で JGroups スキーマを宣言します。
- UDP および TCP プロトコル用に事前設定された JGroups スタック。
- JGroups スタックを拡張して、特定のプロトコルとプロパティーを調整できるようにする継承属性。

```
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:11.0 https://infinispan.org/schemas/infinispan-config-11.0.xsd
                          urn:infinispan:server:11.0 https://infinispan.org/schemas/infinispan-server-11.0.xsd
                          urn:org:jgroups http://www.jgroups.org/schema/jgroups-4.2.xsd" 1
      xmlns="urn:infinispan:config:11.0"
      xmlns:server="urn:infinispan:server:11.0">

  <jgroups> 2
    <stack name="xsite" extends="udp"> 3
      <relay.RELAY2 site="LON" xmlns="urn:org:jgroups"/>
      <remote-sites default-stack="tcp">
        <remote-site name="LON"/>
        <remote-site name="NYC"/>
      </remote-sites>
    </stack>
  </jgroups>

  <cache-container ...>
  ...
</infinispan>
```

1. **infinispan.xml** 内で JGroups 4.2 スキーマを宣言します。
2. カスタムスタック定義を含む JGroups 要素を追加します。
3. クロスサイトレプリケーション用の JGroups プロトコルスタックを定義します。

3.5.1. トランスポートセキュリティー

以前のバージョンと同様に、Data Grid 8 は JGroups SYM_ENCRYPT および ASYM_ENCRYPT プロトコルを使用してクラスター通信を暗号化します。

ノード認証

Data Grid 7.x では、JGroups SASL プロトコルにより、ノードは組み込みサーバーとリモートサーバーの両方のインストールでセキュリティーレルムに対して認証できます。

Data Grid 8 以降、セキュリティーレルムに対してノード認証を設定することはできません。同様に、Data Grid 8 は、クラスター化されたノードの認証に JGroups AUTH プロトコルを使用することを推奨していません。

ただし、組み込みデータグリッドのインストールでは、JGroups クラスタートランスポートに **jgroups** 要素の一部として SASL 設定が含まれます。以前のバージョンと同様に、SASL 設定は、ノード認証に必要な特定の情報を取得するために、**CallbackHandlers** などの JAAS 概念に依存しています。

関連情報

- Data Grid Server ガイド
- Using Embedded Data Grid Caches
- Data Grid セキュリティーガイド

3.6. DATA GRID 認証

Data Grid は、ロールベースのアクセス制御 (RBAC) を使用してデータへのアクセスを制限し、クラスターの暗号化を使用してノード間の通信を保護します。

キャッシュマネージャーの承認

```
<infinispan>
  <cache-container default-cache="secured" name="secured">
    <security>
      <authorization> 1
        <identity-role-mapper /> 2
        <role name="admin" permissions="ALL" /> 3
        <role name="reader" permissions="READ" />
        <role name="writer" permissions="WRITE" />
        <role name="supervisor" permissions="READ WRITE EXEC"/>
      </authorization>
    </security>
  </cache-container>
</infinispan>
```

1. キャッシュマネージャーのライフサイクルを制御するには、ユーザー権限が必要です。
2. ロールにプリンシパルをマップ **PrincipalRoleMapper** の実装を指定します。
3. 一連のロールと関連する権限を定義します。

暗黙的なキャッシュ認証

Data Grid 8 は、キャッシュが **cache-container** から承認設定を継承できるようにすることで使いやすさを向上させるため、各キャッシュのロールとパーミッションを明示的に設定する必要はありません。

```
<local-cache name="secured">
  <security>
    <authorization/> 1
```

```
  </security>
</local-cache>
```

1 キャッシュコンテナーで定義されたロールと権限を使用します。

関連情報

- Data Grid セキュリティーガイド

第4章 DATA GRID 8 API への移行

Data Grid 8 への移行に影響を与える Data Grid API への変更を見つけます。

API の非推奨と削除

このセクションの詳細に加えて、API の非推奨と削除も確認する必要があります。

データグリッドの非推奨の機能と機能 (Red Hat ナレッジベース) を参照してください。

4.1. REST API

Data Grid 7.x は、Data Grid 8 で REST API v2 に置き換えられた REST API v1 を使用していました。

REST API v2 のデフォルトのコンテキストパスは **<server_hostname>:11222/rest/v2/** です。REST API v2 を使用するには、クライアントまたはスクリプトを更新する必要があります。

performAsync ヘッダーも REST エンドポイントから削除されました。REST エンドポイントで非同期操作を実行するクライアントは、ブロックを回避するために、自分の側で要求と応答を管理する必要があります。

REST 操作の **PUT**、**POST**、および **DELETE** メソッドは、リクエストがリソースを返さない場合、 **200** ではなくステータス **204** (コンテンツなし) を返すようになりました。

関連情報

- Data Grid REST API

4.2. クエリー API

Data Grid 8 は、より使いやすく、より軽量なデザインの更新された Query API をもたらします。Data Grid 7.x と比較して、分散キャッシュ内の値全体を検索すると、より効率的なクエリーパフォーマンスが得られ、より良い結果が得られます。

注記

Data Grid 8 Query API はかなりのリファクタリングを経ているため、現在非推奨となっているいくつかの機能と機能リソースがあります。

このトピックでは、以前のバージョンから移行するときに設定に加える必要のある変更に焦点を当てます。これらの変更には、廃止されたすべてのインターフェイス、メソッド、またはその他の設定を削除する計画を含める必要があります。

非推奨の機能の完全なリストについては、Data Grid の非推奨と削除 (Red Hat ナレッジベース) を参照してください。

Data Grid キャッシュのインデックス作成

Data Grid Lucene Directory、**InfinispanIndexManager** および **AffinityIndexManager** インデックスマネージャー、および Hibernate Search 用の Infinispan Directory プロバイダーは、8.0 で非推奨になり、8.1 で削除されました。

auto-config 属性は 8.1 で非推奨になり、削除される予定です。

インデックスモード設定を設定する **index()** メソッドは非推奨になりました。設定でインデックス作成を有効にすると、Data Grid はインデックス作成を管理するための最良の方法を自動的に選択します。

重要

いくつかのインデックス設定値はサポートされなくなり、それらを含めると致命的な設定エラーが発生します。

設定に次の変更を加える必要があります。

- **.indexing().index(Index.NONE)** を **indexing().enabled(false)** に変更
- 他のすべての列挙値を次のように変更します。**indexing().enabled(true)**

宣言的に、設定に他のインデックス設定要素が含まれている場合は、**enabled="true"** を指定する必要はありません。ただし、プログラムでインデックスを設定する場合は、**enabled()** メソッドを呼び出す必要があります。同様に、JSON 形式のデータグリッド設定では、インデックスを明示的に有効にする必要があります。次に例を示します。

```
"indexing": {
    "enabled": "true"
    ...
    },
```

インデックス付きタイプ

インデックス設定ですべてのインデックス付きタイプを宣言する必要があります。宣言されていないタイプがインデックス付きキャッシュで使用される場合、Data Grid は警告メッセージをログに記録します。この要件は、Java クラスと Protobuf タイプの両方に適用されます。

Data Grid 8 でのインデックス作成の有効化

- 宣言的に

```
<distributed-cache name="my-cache">
  <indexing>
    <indexed-entities>
      <indexed-entity>com.acme.query.test.Car</indexed-entity>
      <indexed-entity>com.acme.query.test.Truck</indexed-entity>
    </indexed-entities>
  </indexing>
</distributed-cache>
```

- プログラムで

```
import org.infinispan.configuration.cache.*;

ConfigurationBuilder config=new ConfigurationBuilder();
config.indexing().enable().addIndexedEntity(Car.class).addIndexedEntity(Truck.class);
```

Querying values in caches

org.infinispan.query.SearchManager インターフェイスは Data Grid 8 で非推奨になり、Lucene および HibernateSearch のネイティブオブジェクトをサポートしなくなりました。

削除されたメソッド

- Lucene クエリーを受け取る **.getQuery()** メソッド。代わりに、**org.infinispan.query.Search** エントリーポイントから Ickle クエリーを取得する別のメソッドを使用してください。

同様に、**.getQuery()** の呼び出し時に、複数のターゲットエンティティークラスを指定できなくなりました。Ickle クエリー文字列は、代わりにエンティティーを提供します。

- Hibernate Search クエリーを直接構築する **.buildQueryBuilderForClass()**。代わりに Ickle クエリーを使用してください。

org.infinispan.query.CacheQuery インターフェイスも非推奨になりました。代わりに、**org.infinispan.query.dsl.Query** インターフェイスを **Search.getQueryFactory()** メソッドから取得する必要があります。

org.infinispan.query.dsl.Query のインスタンスはクエリー結果をキャッシュしなくなり、**list()** などのメソッドを呼び出すときにクエリーを再実行することができることに注意してください。

エンティティーマッピング

今後は、すべての場合において、**@SortableField** で並べ替えが必要なフィールドにアノテーションを付ける必要があります。

関連情報

- Data Grid Query API
- Data Grid の非推奨と削除

第5章 アプリケーションの DATA GRID 8 への移行

5.1. DATA GRID 8 でのマーシャリング

マーシャリング機能は、Data Grid 8 で大幅にリファクタリングされ、内部オブジェクトとユーザーオブジェクトを分離します。

Data Grid が内部クラスのマーシャリングを処理するようになったため、埋め込みキャッシュまたはリモートキャッシュを使用してマーシャラーを設定するときに、これらの内部クラスを処理する必要がなくなりました。

5.1.1. ProtoStream マーシャリング

デフォルトでは、Data Grid 8 は ProtoStream API を使用して、言語に依存しない下位互換性のある形式である Protocol Buffers としてデータをマーシャリングします。

Protobuf エンコーディングはスキーマ定義のフォーマットであり、現在多くのアプリケーションのデフォルト標準であり、Data Grid 7 のデフォルトであった JBoss Marshalling と比較してデータをトランスコードする際の柔軟性が高くなっています。

ProtoStream マーシャラーは Protobuf 形式に基づいているため、DataGrid は最初に Java オブジェクトに変換せずに他のエンコーディングに変換できます。JBoss Marshalling を使用する場合、他の形式に変換する前に、キーと値を Java オブジェクトに変換する必要があります。

Data Grid 8 への移行の一環として、Java クラスに ProtoStream マーシャリングの使用を開始する必要があります。

高レベルから、ProtoStream マーシャラーを使用するには、ProtoStream プロセッサーを使用して **SerializationContextInitializer** 実装を生成します。まず、**@Proto** アノテーションを Java クラスに追加してから、Data Grid が提供する ProtoStream プロセッサーを使用して、以下を含むシリアル化コンテキストを生成します。

- Java オブジェクトの構造化表現を Protobuf メッセージタイプとして提供する **.proto** スキーマ。
- Java オブジェクトを Protobuf 形式にエンコードするための Marshaller の実装。

埋め込みキャッシュとリモートキャッシュのどちらを使用するかに応じて、Data Grid は **SerializationContextInitializer** 実装を自動的に登録できます。

Data Grid サーバーとのマーシャリング
リモートキャッシュには Protobuf エンコーディングのみを使用し、カスタムタイプには ProtoStream マーシャラーを組み合わせて使用する必要があります。

JBoss マーシャリングなどの他のマーシャラー実装では、Data Grid CLI、Data Grid Console、または Ickle クエリーと互換性のない異なるキャッシュエンコーディングを使用する必要があります。

キャッシュストアと ProtoStream
Data Grid 7.x では、キャッシュストアに永続化するデータは、Data Grid 8 の ProtoStream マーシャラーと互換性がありません。データを **DataGrid7.x** キャッシュストアから Data Grid 8 キャッシュストアに移行するには、StoreMigrator ユーティリティーを使用する必要があります。

5.1.2. 代替マーシャラーの実装

Data Grid は、ProtoStream の代替マーシャラー実装を提供し、古いバージョンからの移行を容易にします。これらの代替マーシャラーは、ProtoStream マーシャリングに移行する際の暫定的な解決策としてのみ使用する必要があります。

注記

新しいプロジェクトの場合、将来のアップグレードや移行に関する問題を回避するために、ProtoStream マーシャリングのみを使用することを強く推奨します。

JBoss marshalling
Data Grid 7 では、JBoss Marshalling がデフォルトのマーシャラーです。Data Grid 8 では、Proto Stream マーシャリングがデフォルトです。

注記

Java シリアル化を使用するクライアント要件がある場合は、JBoss Marshalling の代わりに **JavaSerializationMarshaller** を使用する必要があります。

Data Grid 8 への移行中に一時的な解決策として JBoss Marshalling を使用する必要がある場合は、以下を実行します。

埋め込みキャッシュ

1. **infinispan-jboss-marshalling** 依存関係をクラスパスに追加します。
2. 次に、**JBossUserMarshaller** を使用するようにデータグリッドを設定します。

```
<serialization marshaller="org.infinispan.jboss.marshalling.core.JBossUserMarshaller"/>
```

3. Data Grid が逆シリアル化を許可するクラスのリストにクラスを追加します。

リモートキャッシュ

Data GridServer は JBossMarshalling をサポートしておらず、**infinispan-jboss-marshalling** モジュールがクラスパス上にある場合、**GenericJBossMarshaller** は自動的に設定されなくなりました。

次のように、JBoss Marshalling を使用するように Hot Rod Java クライアントを設定する必要があります。

- **RemoteCacheManager**

```
.marshaller("org.infinispan.jboss.marshalling.commons.GenericJBossMarshaller");
```

- **hotrod-client.properties**

```
infinispan.client.hotrod.marshaller = GenericJBossMarshaller
```

関連情報

- キャッシュのエンコードとマーシャリング

5.2. AUTOPROTOSCHEMABUILDER アノテーションへのアプリケーションの移行

以前のバージョンの DataGrid は、ProtoStream API の **MessageMarshaller** インターフェイスを使用してマーシャリングを設定します。

MessageMarshaller API と **ProtoSchemaBuilder** アノテーションはどちらも、ProtoStream4.3.4 に対応する DataGrid 8.1.1 で非推奨になりました。

MessageMarshaller インターフェイスの使用には、以下のいずれかが含まれます。

- Protobuf スキーマを手動で作成します。
- **ProtoSchemaBuilder** アノテーションを Java クラスに追加してから、Protobuf スキーマを生成します。

ただし、ProtoStream マーシャリングを設定するためのこれらの手法は、 Data Grid 8.1.1 以降で使用可能な **AutoProtoSchemaBuilder** アノテーションほど効率的で信頼性が高くありません。**AutoProtoSchemaBuilder** アノテーションを Java クラスに追加し、Protobuf スキーマと関連するマーシャラーを含む **SerializationContextInitializer** 実装を生成するだけです。

Red Hat は、**AutoProtoSchemaBuilder** アノテーションの使用を開始して、ProtoStream マーシャラーから最良の結果を取得することを推奨します。

次のコード例は、アプリケーションを **MessageMarshaller** API から **AutoProtoSchemaBuilder** アノテーションに移行する方法を示しています。

5.2.1. 基本的な MessageMarshaller の実装

この例には、デフォルト以外のタイプを使用するいくつかのフィールドが含まれています。コードジェネレーターはデフォルトで **int** 型を使用するため、**text** フィールドの順序は異なり、**fixed32** フィールドは生成された Protobuf スキーマタイプと競合します。

SimpleEntry.java

```
public class SimpleEntry {

  private String description;
  private Collection<String> text;
  private int intDefault;
  private Integer fixed32;

  // public Getter, Setter, equals and HashCode methods omitted for brevity
}
```

SimpleEntryMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class SimpleEntryMarshaller implements MessageMarshaller<SimpleEntry> {

  @Override
  public void writeTo(ProtoStreamWriter writer, SimpleEntry testEntry) throws IOException {
    writer.writeString("description", testEntry.getDescription());
    writer.writeInt("intDefault", testEntry.getIntDefault());
```

```
    writer.writeInt("fix32", testEntry.getFixed32());
    writer.writeCollection("text", testEntry.getText(), String.class);
  }

  @Override
  public SimpleEntry readFrom(MessageMarshaller.ProtoStreamReader reader) throws IOException {
    SimpleEntry x = new SimpleEntry();

    x.setDescription(reader.readString("description"));
    x.setIntDefault(reader.readInt("intDefault"));
    x.setFixed32(reader.readInt("fix32"));
    x.setText(reader.readCollection("text", new LinkedList<String>(), String.class));

    return x;
  }
}
```

結果の Protobuf スキーマ

```
syntax = "proto2";

package example;

message SimpleEntry {
  required string description = 1;
  optional int32 intDefault = 2;
  optional fixed32 fix32 = 3;
  repeated string text = 4;
}
```

AutoProtoSchemaBuilder アノテーションに移行

SimpleEntry.java

```
import org.infinispan.protostream.annotations.ProtoField;
import org.infinispan.protostream.descriptors.Type;

public class SimpleEntry {

  private String description;
  private Collection<String> text;
  private int intDefault;
  private Integer fixed32;

  @ProtoField(number = 1)
  public String getDescription() {...}

  @ProtoField(number = 4, collectionImplementation = LinkedList.class)
  public Collection<String> getText() {...}

  @ProtoField(number = 2, defaultValue = "0")
  public int getIntDefault() {...}

  @ProtoField(number = 3, type = Type.FIXED32)
  public Integer getFixed32() {...}
```

```
  // public Getter, Setter, equals and HashCode methods and convenient constructors omitted for
brevity
}
```

SimpleEntryInitializer.java

```
import org.infinispan.protostream.GeneratedSchema;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;

@AutoProtoSchemaBuilder(includeClasses = { SimpleEntry.class }, schemaFileName =
"simple.proto", schemaFilePath = "proto", schemaPackageName = "example")
public interface SimpleEntryInitializer extends GeneratedSchema {
}
```

重要な所見

- Field 2 は、以前のバージョンの ProtoStream マーシャラーがチェックしなかった **int** として定義されています。
- Java **int** フィールドは null 許容ではないため、ProtoStream プロセッサーは失敗します。Java **int** フィールドは、**required** または **defaultValue** で初期化する必要があります。Java アプリケーションの観点からは、**int** フィールドは 0 で初期化されるため、put 操作で設定されるため、影響を与えることなく **defaultValue** を使用できます。**required** への変更は、常に存在する場合、保存されたデータの観点からは問題ではありませんが、さまざまなクライアントで問題が発生する可能性があります。
- 互換性を確立するため、Field 3 は、**Type.FIXED32** に明示的に設定する必要があります。
- **テキストコレクション**は、結果の Protobuf スキーマに対して正しい順序で設定する必要があります。

重要

Protobuf スキーマのテキストコレクションの順序は、移行の前後で同じである必要があります。同様に、移行中に **fixed32** タイプを設定する必要があります。

そうでない場合、クライアントアプリケーションは次の例外を出力し、起動に失敗する可能性があります。

```
Exception ( ISPN004034: Unable to unmarshall bytes )
```

また、キャッシュされたデータに不完全または不正確な結果が表示される場合もあります。

5.2.2. カスタムタイプを使用した MessageMarshaller の実装

このセクションでは、ProtoStream がネイティブに処理しないフィールドを含む **MessageMarshaller** 実装の移行例を示します。

次の例では **BigInteger** クラスを使用していますが、データグリッドアダプターやカスタムクラスを含め、すべてのクラスに適用されます。

注記

BigInteger クラスは不変であるため、引数のないコンストラクターはありません。

CustomTypeEntry.java

```
import java.math.BigInteger;

public class CustomTypeEntry {

  final String description;
  final BigInteger bigInt;

  // public Getter, Setter, equals and HashCode methods and convenient constructors omitted for
brevity
}
```

CustomTypeEntryMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class CustomTypeEntryMarshaller implements MessageMarshaller<CustomTypeEntry> {

  @Override
  public void writeTo(ProtoStreamWriter writer, CustomTypeEntry testEntry) throws IOException {
    writer.writeString("description", testEntry.description);
    writer.writeString("bigInt", testEntry.bigInt.toString());
  }

  @Override
  public CustomTypeEntry readFrom(MessageMarshaller.ProtoStreamReader reader) throws
IOException {
    final String desc = reader.readString("description");
    final BigInteger bInt = new BigInteger(reader.readString("bigInt"));

    return new CustomTypeEntry(desc, bInt);
  }
}
```

CustomTypeEntry.proto

```
syntax = "proto2";

package example;

message CustomTypeEntry {
   required string description = 1;
   required string bigInt = 2;
}
```

アダプタークラスを使用して移行されたコード

ProtoAdapter アノテーションを使用して、**MessageMarshaller** 実装で作成した Protobuf スキーマと互換性のある Protobuf スキーマを生成する方法で **CustomType** クラスをマーシャリングできます。

このアプローチでは、次のことができます。

- **CustomTypeEntry** クラスにアノテーションを追加してはなりません。
- **@ProtoAdapter** アノテーションを使用して Protobuf スキーマとマーシャラーの生成方法を制御する **CustomTypeEntryAdapter** クラスを作成します。
- **@AutoProtoSchemaBuilder** アノテーションとともに **CustomTypeEntryAdapter** クラスを含めます。

注記

AutoProtoSchemaBuilder アノテーションは **CustomTypeEntry** クラスを参照しないため、そのクラスに含まれるアノテーションはすべて無視されます。

次の例が示す **CustomTypeEntry** クラスの ProtoStream アノテーションを含む **CustomTypeEntryAdapter** クラス:

CustomTypeEntryAdapter.java

```
import java.math.BigInteger;

import org.infinispan.protostream.annotations.ProtoAdapter;
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

@ProtoAdapter(CustomTypeEntry.class)
public class CustomTypeEntryAdapter {

  @ProtoFactory
  public CustomTypeEntry create(String description, String bigInt) {
    return new CustomTypeEntry(description, new BigInteger(bigInt));
  }

  @ProtoField(number = 1, required = true)
  public String getDescription(CustomTypeEntry t) {
    return t.description;
  }

  @ProtoField(number = 2, required = true)
  public String getBigInt(CustomTypeEntry t) {
    return t.bigInt.toString();
  }
}
```

次の例が示す **CustomTypeEntryAdapter** クラスを参照する **AutoProtoSchemaBuilder** アノテーションと t もに **SerializationContextInitializer** を示しています。

CustomTypeEntryInitializer.java

```
import org.infinispan.protostream.GeneratedSchema;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;

@AutoProtoSchemaBuilder(includeClasses = { CustomTypeEntryAdapter.class },
```

```
    schemaFileName = "custom.proto",
    schemaFilePath = "proto",
    schemaPackageName = "example")
public interface CustomTypeAdapterInitializer extends GeneratedSchema { }
```

アダプタークラスなしで移行されたコード
アダプタークラスを作成する代わりに、ProtoStream アノテーションを **CustomTypeEntry** クラスに直接追加できます。

重要

この例では、生成された Protobuf スキーマは、**BigInteger** が別個のメッセージであるため、**MessageMarshaller** インターフェイスを介して追加されたキャッシュ内のデータと互換性がありません。アダプターフィールドに同じ文字列が書き込まれていても、データのマーシャリングを解除することはできません。

次の例が示す直接 ProtoStream アノテーションを含む **CustomTypeEntry** クラス:

CustomTypeEntry.java

```
import java.math.BigInteger;

public class CustomTypeEntry {

  @ProtoField(number = 1)
  final String description;
  @ProtoField(number = 2)
  final BigInteger bigInt;

  @ProtoFactory
  public CustomTypeEntry(String description, BigInteger bigInt) {
    this.description = description;
    this.bigInt = bigInt;
  }

  // public Getter, Setter, equals and HashCode methods and convenient constructors omitted for
brevity
}
```

以下の例は、**CustomTypeEntry** および **BigIntegerAdapter** クラスを参照する **AutoProtoSchemaBuilder** アノテーションと共に **SerializationContextInitializer** を示しています。

CustomTypeEntryInitializer.java

```
import org.infinispan.protostream.GeneratedSchema;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;
import org.infinispan.protostream.types.java.math.BigIntegerAdapter;

@AutoProtoSchemaBuilder(includeClasses = { CustomTypeEntry.class,
  BigIntegerAdapter.class },
  schemaFileName = "customtype.proto",
```

```
    schemaFilePath = "proto",
    schemaPackageName = "example")
public interface CustomTypeInitializer extends GeneratedSchema { }
```

前述の **SerializationContextInitializer** 実装から Protobuf スキーマを生成すると、以下の Protobuf スキーマになります。

CustomTypeEntry.proto

```
syntax = "proto2";

package example;

message BigInteger {
   optional bytes bytes = 1;
}

message CustomTypeEntry {
   optional string description = 1;
   optional BigInteger bigInt = 2;
}
```

第6章 RED HAT OPENSHIFT での DATA GRID クラスターの移行

Red Hat OpenShift で実行されている Data Grid クラスターの移行の詳細を確認してください。

6.1. OPENSHIFT の DATA GRID

Data Grid 8 は、運用インテリジェンスを提供し、OpenShift に DataGrid をデプロイするための管理の複雑さを軽減する Data Grid Operator を導入しています。

Red Hat は、Data Grid Operator サブスクリプションを介してのみ、OpenShift 上の Data Grid 8 をサポートします。

Data Grid 8 では、Data Grid Operator は認証、クライアントキーストア、外部ネットワークアクセス、ロギングなどの Data Grid クラスターのほとんどの設定を処理します。

Data Grid サービスの作成

Data Grid 7.3 では、OpenShift で Data Grid クラスターを作成するための Cache サービスおよび Data Grid サービスが導入されました。

Data Grid 7.3 でこれらのサービスを作成するには、必要に応じてサービステンプレートをインポートしてから、テンプレートパラメーターと環境変数を使用してサービスを設定します。

7.3 でのキャッシュサービスノードの作成

```
$ oc new-app cache-service \
  -p APPLICATION_USER=${USERNAME} \
  -p APPLICATION_PASSWORD=${PASSWORD} \
  -p NUMBER_OF_INSTANCES=3 \
  -p REPLICATION_FACTOR=2
```

7.3 での Data Grid サービスノードの作成

```
$ oc new-app datagrid-service \
  -p APPLICATION_USER=${USERNAME} \
  -p APPLICATION_PASSWORD=${PASSWORD} \
  -p NUMBER_OF_INSTANCES=3
  -e AB_PROMETHEUS_ENABLE=true
```

Data Grid 8 でのサービスの作成

1. Data Grid Operator サブスクリプションを作成します。
2. **Infinispan** カスタムリソース (CR) を作成して、Data Grid クラスターをインスタンス化し、設定します。

```
apiVersion: infinispan.org/v1
kind: Infinispan
metadata:
  name: example-infinispan
spec:
  replicas: 2
  service:
    type: Cache 1
```

1 **spec.service.type** フィールドは、Cache サービスノードまたは Data Grid サービスノードを作成するかどうかを指定します。

6.1.1. コンテナーストレージ

Data Grid 7.3 サービスは、**/opt/datagrid/standalone/data** にマウントされたストレージボリュームを使用します。

Data Grid 8 サービスは、**/opt/infinispan/server/data** にマウントされた永続ボリューム要求を使用します。

6.1.2. Data Grid CLI

Data Grid 7.3 では、リモートシェルからのみ CLI にアクセスできます。Data Grid 7.3 CLI 経由で行った変更は Pod にバインドされ、再起動後は維持されませんでした。Data Grid 8 を使用すると、OpenShift でクラスターを使用して管理操作を実行したり、データを操作したりするための完全に機能するメカニズムとして CLI を使用できます。

6.1.3. Data Grid コンソール

Data Grid 7.3 は OpenShift のコンソールをサポートしませんでした。Data Grid 8 では、コンソールを使用して OpenShift で実行されているクラスターを監視し、管理操作を実行し、キャッシュをリモートで作成できます。

6.1.4. Data Grid のカスタマイズ

Data Grid 7.3 では、Source-to-Image(S2I) プロセスおよび **ConfigMap** API を使用して、OpenShift で実行される Data Grid サーバーイメージをカスタマイズできます。

Data Grid 8 では、Red Hat は、Red Hat Container Registry からの Data Grid イメージのカスタマイズをサポートしません。

Data Grid Operator は、OpenShift 上の Data Grid 8 クラスターのデプロイメントおよび管理を処理します。

その結果、カスタムを使用できません。

- 検出プロトコル
- 暗号化メカニズム (SYM_ENCRYPT または ASYM_ENCRYPT)
- 永続的なデータソース

Data Grid 8.0 および 8.1 では、Data Grid Operator では **JAR** ファイルや他のアーティファクトなどのカスタムコードをデプロイすることはできません。

6.1.5. デプロイメント設定テンプレート

Data Grid 7.3 で利用可能だったデプロイメント設定テンプレートおよび環境変数は、Data Grid 8 で削除されました。

第7章 キャッシュストア間のデータの移行

Data Grid は、キャッシュストア間で永続化されたデータを移行するための Java ユーティリティーを提供します。

Data Grid をアップグレードする場合、メジャーバージョン間の機能相違点は、キャッシュストア間の後方互換性を許可しません。**StoreMigrator** を使用してデータを変換し、ターゲットバージョンとの互換性を持つことができます。

たとえば、Data Grid 8.0 にアップグレードすると、デフォルトのマーシャラーが Protostream に変更になります。以前の Data Grid バージョンでは、キャッシュストアはバイナリー形式を使用し、マーシャリングする変更との互換性がありません。つまり、Data Grid 8.0 は、以前の Data Grid バージョンでキャッシュストアから読み込むことができません。

他の場合は、Data Grid のバージョンが、JDBC Mixed および Binary ストアなどのキャッシュストア実装を非推奨または削除します。このような場合は、**StoreMigrator** を使用して異なるキャッシュストア実装に変換できます。

7.1. キャッシュストアマイグレーション

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

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

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

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

重要

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

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

7.2. STORE MIGRATOR の取得

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

手順

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

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

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

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

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

7.3. ストア移行の設定

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

手順

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

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

3. **migrator.properties** でターゲットキャッシュストアを設定します。

   a. 以下の例のように、すべての設定プロパティーの先頭に **target.** を付けます。

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

7.3.1. 移行プロパティーの保存

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

表7.1 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タイプ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7.2 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	ストアがバックアップするキャッシュに名前を付けます。	**.cache_name=myCache**	必須

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を指定します。 セグメント数は、Data Grid 設定の **clustering.hash.numSegments** と一致する必要があります。 つまり、キャッシュストアのセグメント数は、対応するキャッシュのセグメント数と一致する必要があります。セグメントの数が同一でない場合、Data Grid はキャッシュストアからデータを読み込めません。	**.segment_count=256**	任意

表7.3 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
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.x の場合は **10**	ソースストアにのみ必要です。 例: **source.version=9**
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.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

プロパティー	説明	必須/オプション
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.major_version	データベースのメジャーバージョンを設定します。	Optional
db.minor_version	データベースのマイナーバージョンを設定します。	Optional
db.disable_upsert	データベース upsert を無効にします。	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.example.CustomMarshaller
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.major_version=9
target.db.minor_version=5
target.db.disable_upsert=false
target.db.disable_indexing=false
target.table.string.table_name_prefix=tablePrefix
target.table.string.id.name=id_column
target.table.string.data.name=datum_column
target.table.string.timestamp.name=timestamp_column
target.table.string.id.type=VARCHAR
target.table.string.data.type=bytea
target.table.string.timestamp.type=BIGINT
target.key_to_string_mapper=org.infinispan.persistence.keymappers.
DefaultTwoWayKey2StringMapper
```

表7.4 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7.5 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7.6 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
```

7.4. キャッシュストアの移行

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

前提条件

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

手順

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