Red Hat JBoss Enterprise Application Platform 7.3

Hibernate アプリケーションの開発

Red Hat JBoss Enterprise Application Platform 向けの Jakarta Persistence API (JPA) または Hibernate アプリケーションを開発およびデプロイする開発者および管理者のための手順と情報。

Last Updated: 2023-09-02

Red Hat JBoss Enterprise Application Platform 7.3 Hibernate アプリケーションの開発

Red Hat JBoss Enterprise Application Platform 向けの Jakarta Persistence API (JPA) または Hibernate アプリケーションを開発およびデプロイする開発者および管理者のための手順と情報。

法律上の通知

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

本書は、Red Hat JBoss Enterprise Application Platform で Jakarta Persistence/Hibernate アプリケーションの開発とデプロイを行う開発者と管理者に情報を提供します。

目次

第1章 はじめに 3
1.1. HIBERNATE CORE 3
1.2. HIBERNATE ENTITYMANAGER 3

第2章 HIBERNATE の設定 5
2.1. HIBERNATE の設定 5
2.2. 2 次キャッシュ 5

第3章 HIBERNATE アノテーション 7
3.1. HIBERNATE アノテーション 7

第4章 HIBERNATE QUERY 言語 15
4.1. HIBERNATE QUERY 言語について 15
4.2. HQL ステートメントについて 15
4.3. HQL の順序について 19
4.4. コレクションメンバーの参照について 20
4.5. 修飾パス式について 21
4.6. HQL 関数について 22
4.7. 動的インスタンス化について 23
4.8. HQL 述語について 24
4.9. 関連比較について 27
4.10. バイトコードの機能強化 28

第5章 HIBERNATE サービス 30
5.1. HIBERNATE サービスについて 30
5.2. サービス契約について 30
5.3. サービス依存関係の種類 30

第6章 HIBERNATE ENVERS 36
6.1. HIBERNATE ENVERS について 36
6.2. 永続クラスの監査について 36
6.3. 監査ストラテジー 36
6.4. CONFIGURATION (設定) 39
6.5. 監査情報のクエリー 42
6.6. パフォーマンスチューニング 46

第7章 HIBERNATE SEARCH 49
7.1. HIBERNATE SEARCH の使用 49
7.2. CONFIGURATION (設定) 53
7.3. アプリケーションの HIBERNATE SEARCH 73
7.4. インデックス構造へのエンティティーのマッピング 79
7.5. HIBERNATE SEARCH を使用した LUCENE クエリーの実行 104
7.6. 手動によるインデックスの変更 131
7.7. インデックスの最適化 135
7.8. 高度な機能 137
7.9. モニタリング 142

付録A リファレンス資料 143
A.1. HIBERNATE プロパティー 143

第1章 はじめに

1.1. HIBERNATE CORE

Hibernate Core は、Java 言語用のオブジェクト関係マッピングフレームワークです。これにより、オブジェクト指向のドメインモデルをリレーショナルデータベースにマップするフレームワークが提供されるため、アプリケーションはデータベースとの直接的な対話を回避できます。Hibernate は、直接の永続データベースアクセスを高レベルのオブジェクト処理関数に置き換えることで、オブジェクト関係の不一致の問題を解決します。

1.2. HIBERNATE ENTITYMANAGER

Hibernate EntityManager は、Jakarta Persistence 2.2 仕様で定義されているプログラミングインターフェイスおよびライフサイクルルールを実装します。Hibernate Annotations とともに、このラッパーは成熟した Hibernate Core の上に完全な (およびスタンドアロン) Jakarta Persistence ソリューションを実装します。プロジェクトのビジネスニーズおよび技術上のニーズに応じて、Jakarta Persistence プログラミングインターフェイスやライフサイクル、あるいは純粋なネイティブ Hibernate Core でさえすべて組み合わせて使用できます。いつでも、Hibernate ネイティブ API、または必要に応じてネイティブ JDBC および SQL にフォールバックできます。JBoss EAP には完全な Jakarta 永続ソリューションが提供されます。

JBoss EAP 7.3 リリースは、Jakarta EE 8 で定義されている Jakarta Persistence 2.2 specification に準拠しています。

また、Hibernate によってこの仕様に追加機能が提供されます。Jakarta Persistence および JBoss EAP を初めて使用する場合は、JBoss EAP に同梱される **bean-validation**、**greeter**、および **kitchensink** クイックスタートを参照してください。

Jakarta Persistence は、Jakarta Enterprise Beans 3 または、より新しい Jakarta Contexts and Dependency Injection などのコンテナーや、特定コンテナーの外部で実行するスタンドアロン Java SE アプリケーションで利用できます。以下のプログラミングインターフェイスおよびアーティファクトは、両方の環境で利用できます。

重要

Hibernate でセキュリティーマネージャーの使用を計画している場合は、**EntityManagerFactory** が JBoss EAP サーバーによってブートストラップされている場合のみ Hibernate がサポートすることに注意してください。**EntityManagerFactory** または **SessionFactory** がアプリケーションによってブートストラップされている場合はサポートされません。

EntityManagerFactory

エンティティーマネージャーファクトリーはエンティティーマネージャーインスタンスを提供し、すべてのインスタンスは同じデータベースに接続するよう設定され、特定の実装で定義されたのと同じデフォルト設定を使用します。複数のデータストアにアクセスするための複数のエンティティーマネージャーファクトリーを準備できます。このインターフェイスは、ネイティブ Hibernate の SessionFactory と似ています。

EntityManager

EntityManager API は、特定の作業単位でデータベースにアクセスするために使用されます。これは、永続的なエンティティーインスタンスの作成および削除、それらのプライマリーキーアイデンティティーによるエンティティーの検索、およびすべてのエンティティーのクエリーに使用されます。このインターフェイスは、Hibernate の Session と似ています。

永続コンテキスト

永続コンテキストは、すべての永続エンティティーアイデンティティーに一意のエンティティーインスタンスがあるエンティティーインスタンスのセットです。永続コンテキスト内では、エンティティーインスタンスとそのライフサイクルは特定のエンティティーマネージャーによって管理されます。このコンテキストの範囲は、トランザクションまたは作業の拡張単位のいずれかになります。

永続ユニット

指定のエンティティーマネージャーで管理できるエンティティータイプのセットは永続ユニットによって定義されます。永続ユニットは、アプリケーションに関連する、またはアプリケーションによってグループ化されるすべてのクラスのセットを定義します。これらのクラスは、単一データストアへのマッピングに共存する必要があります。

コンテナー管理のエンティティーマネージャーー

コンテナーがライフサイクルを管理するエンティティーマネージャー。

例: アプリケーション管理のエンティティーマネージャー

ライフサイクルがアプリケーションによって管理されているエンティティーマネージャー。

Jakarta Transactions エンティティーマネージャー

Jakarta Transactions トランザクションに関連するエンティティーマネージャー。

Resource-local エンティティーマネージャー

リソーストランザクションを使用するエンティティーマネージャー (Jakarta Transactions トランザクションではない)。

その他のリソース

- クイックスタートをダウンロードし、実行する方法は、JBoss EAP スタートガイド のクイックスタートサンプルの使用を参照してください。
- セキュリティーマネージャーに関する詳細は、サーバーセキュリティーの設定方法 の**Java Security Manager**を参照してください。

第2章 HIBERNATE の設定

2.1. HIBERNATE の設定

アプリケーションサーバーとスタンドアロンアプリケーションの両方のエンティティーマネージャーの設定は永続アーカイブにあります。永続アーカイブは、**META-INF/** フォルダーにある **persistence.xml** ファイルを定義する必要がある JAR ファイルです。

persistence.xml ファイルを使用してデータベースに接続できます。これには、以下の方法があります。

- JBoss EAP の **datasources** サブシステムで設定されたデータソースを指定します。**jta-data-source** は、この永続ユニットマップ先となるデータソースの Java Naming and Directory Interface 名を参照します。ここでの **java:jboss/datasources/ExampleDS** は、JBoss EAP に埋め込まれた **H2 DB** を指しています。

  persistence.xml ファイルの object-logical -mapping の例

  ```
  <persistence>
    <persistence-unit name="myapp">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
      <properties>
        ... ...
      </properties>
    </persistence-unit>
  </persistence>
  ```

- 接続プロパティーを指定して **persistence.xml** ファイルを明示的に設定します。

  persistence.xml ファイルで接続プロパティーを指定する例

  ```
  <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
  <property name="javax.persistence.jdbc.user" value="sa"/>
  <property name="javax.persistence.jdbc.password" value=""/>
  <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:."/>
  ```

  接続プロパティーの完全なリストは、**persistence.xml** の Connection Properties Configurable を参照してください。

ランタイム時に Hibernate の動作を制御するプロパティーは複数あります。すべてはオプションで、適切なデフォルト値を持ちます。これらの Hibernate プロパティーはすべて **persistence.xml** で使用されます。設定可能な Hibernate プロパティーの完全リストは、Hibernate Properties を参照してください。

2.2. 2 次キャッシュ

2.2.1. 2 次キャッシュ

2 次キャッシュとは、アプリケーションセッション外部で永続化された情報を保持するローカルデータストアのことです。このキャッシュは永続プロバイダーにより管理され、アプリケーションとデータを分離することでランタイムを改善します。

JBoss EAP では、以下の目的のためにキャッシュがサポートされます。

- Web セッションのクラスターリング
- ステートフルセッション Bean のクラスターリング
- SSO クラスターリング
- Hibernate 2 次キャッシュ
- Jakarta Persistence 2 次キャッシュ

警告

各キャッシュコンテナーは **repl** および **dist** キャッシュを定義します。これらのキャッシュは、ユーザーアプリケーションで直接使用しないでください。

2.2.2. Hibernate の 2 次キャッシュの設定

Hibernate の 2 次キャッシュとして機能する Infinispan の設定は、以下の 2 ついずれかの方法で実行できます。

- JBoss EAP **Development Guide**で説明されているように **persistence.xml** ファイルを使用して Jakarta Persistence アプリケーションを介して 2 次キャッシュを設定することが推奨されます。
- または、以下で説明されているように **hibernate.cfg.xml** ファイルを使用して、Hibernate ネイティブアプリケーションで 2 次キャッシュを設定できます。

Hibernate ネイティブアプリケーションを使用した Hibernate 用 2 次キャッシュの設定

1. デプロイメントのクラスパスに **hibernate.cfg.xml** ファイルを作成します。
2. 以下の XML を **hibernate.cfg.xml** ファイルに追加します。XML は **<session-factory>** タグ内になければなりません。

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property
name="hibernate.cache.region.factory_class">org.jboss.as.jpa.hibernate5.infinispan.InfinispanRegionFactory</property>
```

3. アプリケーション内で Hibernate ネイティブ API を使用するには、以下の依存関係を **MANIFEST.MF** ファイルに追加する必要があります。

```
Dependencies: org.infinispan,org.hibernate
```

第3章 HIBERNATE アノテーション

3.1. HIBERNATE アノテーション

Org.hibernate.annotations パッケージには、標準の Jakarta Persistence アノテーションの上に Hibernate によって提供されるアノテーションが含まれます。

表3.1 一般的なアノテーション

アノテーション	説明
Check	クラス、プロパティー、またはコレクションレベルで定義できる任意の SQL チェック制約。
Immutable	エンティティーまたはコレクションにイミュータブルとしてマークを付けます。アノテーションがないということは、要素が変更されたことを意味します。 イミュータブルなエンティティーはアプリケーションによって更新されない可能性があります。イミュータブルなエンティティーの更新は無視されますが、例外は発生しません。 **@Immutable** をコレクションに配置すると、コレクションがイミュータブルな状態になり、コレクションへの追加および削除は許可されません。この場合、**HibernateException** が発生します。

表3.2 キャッシュエンティティー

アノテーション	説明
Cache	キャッシュストラテジーをルートエンティティーまたはコレクションに追加します。

表3.3 コレクション関連のアノテーション

アノテーション	説明
MapKeyType	永続マップのキーのタイプを定義します。
ManyToAny	異なるエンティティータイプを参照する **ToMany** 関連付けを定義します。エンティティータイプのマッチングは、メタデータの識別子コラムを使用して行われます。このようなマッピングはマージのみにする必要があります。
OrderBy	SQL 順序付け (HQL の順序ではなく) を使用してコレクションを順序付けます。

アノテーション	説明
OnDelete	コレクション、アレイ、および結合されたサブクラスの削除で使用するストラテジー。現在、セカンダリーテーブルの **OnDelete** はサポートされていません。
Persister	カスタム永続機能を指定します。
Sort	コレクションのソート (Java レベルのソート)。
Where	コレクションのエンティティーまたはターゲットエンティティーに追加する要素。レシピは SQL で記述されます。
WhereJoinTable	コレクションジョインテーブルに追加する部分。レシピは SQL で記述されます。

表3.4 CRUD 操作のためのカスタム SQL

アノテーション	説明
Loader	Hibernate デフォルトの **FIND** メソッドを上書きします。
SQLDelete	Hibernate デフォルトの **DELETE** メソッドを上書きします。
SQLDeleteAll	Hibernate のデフォルト **DELETE ALL** メソッドを上書きします。
SQLInsert	Hibernate のデフォルト **INSERT INTO** メソッドを上書きします。
SQLUpdate	Hibernate デフォルトの **UPDATE** メソッドを上書きします。
Subselect	イミュータブルで読み取り専用のエンティティーを指定の SQL サブ選択式にマッピングします。
Synchronization	自動フラッシュが正しく実行され、派生エンティティーに対するクエリーが古いデータを返さないようにします。大半は **Subselect** とともに使用されます。

表3.5 エンティティー

アノテーション	説明
Cascade	関連付けにカスケードストラテジーを適用します。
Entity	標準の **@Entity** で定義された以外のメタデータが必要になる可能性のある追加のメタデータを追加します。 ● **mutable**: このエンティティーが変更可能であるかどうか ● **dynamicInsert** : 動的な SQL を許可する ● **DynamicUpdate**: 更新のための動的 SQL を許可する ● **selectBeforeUpdate**: オブジェクトが実際に変更されたことが明らかでない場合、Hibernate が SQL UPDATE を実行しないように指定します。 ● **polymorphism**: エンティティーポリモーフィズムは PolymorphismType.IMPLICIT (デフォルト) または PolymorphismType.EXPLICIT のいずれか ● **optimisticLock**: 楽観的ロックストラテジー (OptimisticLockType.VERSION、OptimisticLockType.NONE、OptimisticLockType.DIRTY または OptimisticLockType.ALL) **注記** アノテーション Entity は非推奨となり、今後のリリースで削除される予定です。個別の属性または値はアノテーションになります。
Polymorphism	ポリモーフィズム Hibernate のタイプを定義するために使用されます。
Proxy	特定のクラスのレイジーおよびプロキシー設定。
Table	プライマリーまたはセカンダリーのテーブルに情報を補完します。
Tables	Table の複数形のアノテーション。
Target	明示的なターゲットを定義し、反映および汎用的な解決を回避します。

アノテーション	説明
Tuplizer	エンティティーまたはコンポーネントの tuplizer を定義します。
Tuplizers	エンティティーまたはコンポーネントの一連の tuplizer を定義します。

表3.6 フェッチ

アノテーション	説明
BatchSize	SQL 読み込みのバッチサイズ。
FetchProfile	フェッチストラテジープロファイルを定義します。
FetchProfiles	**@FetchProfile** の複数形のアノテーション。
LazyGroup	エンティティー属性が同じグループに属する他のすべての属性と共にフェッチされる必要があることを指定します。エンティティー属性をレイジーに読み込むには、バイトコード拡張が必要です。デフォルトでは、すべての非コレクション属性は **DEFAULT** という名前のグループに読み込まれます。このアノテーションを使用すると、グループ内の属性にアクセスするときに異なる属性のグループを同時に初期化できます。

表3.7 フィルター

アノテーション	説明
Filter	フィルターをエンティティーまたはコレクションのターゲットエンティティーに追加します。
FilterDef	フィルター定義。
FilterDefs	フィルター定義の配列。
FilterJoinTable	フィルターを結合テーブルコレクションに追加します。
FilterJoinTables	複数の **@FilterJoinTable** をコレクションに追加します。
フィルター	複数の **@Filter** を追加します。

アノテーション	説明
ParamDef	パラメーター定義。

表3.8 プライマリーキー

アノテーション	説明
Generated	このアノテーション付きプロパティーはデータベースによって生成されます。
GenericGenerator	タイプ解除された方法で、あらゆる Hibernate generator を記述するジェネレーターアノテーション。
GenericGenerators	汎用ジェネレーター定義のアレイ。
NaturalId	プロパティーがエンティティーの純粋な ID の一部であることを指定します。
Parameter	キー/値のパターン。
RowId	Hibernate の **ROWID** マッピング機能のサポート。

表3.9 継承

アノテーション	説明
DiscriminatorFormula	ルートエンティティーに配置されるイデンティターの式。
DiscriminatorOptions	Hibernate 固有の識別子プロパティーを表示するオプションのアノテーションです。
MetaValue	指定の識別子値を対応するエンティティータイプにマッピングします。

表3.10 JP-QL/HQL クエリーのマッピング

アノテーション	説明
NamedNativeQueries	**NamedNativeQueries** を拡張して、Hibernate NamedNativeQuery オブジェクトを保持します。
NamedNativeQuery	Hibernate 機能を使用した **NamedNativeQuery** の拡張

アノテーション	説明
NamedQueries	**NamedQuery** を拡張して Hibernate **NamedQuery** オブジェクトを保持します。
NamedQuery	Hibernate 機能を使用して **NamedQuery** を拡張します。

表3.11 簡単なプロパティーのマッピング

アノテーション	説明
AccessType	プロパティーアクセスタイプ。
Columns	列のアレイをサポートします。コンポーネントのユーザータイプマッピングに便利です。
ColumnTransformer	値をコラムから読み取り、コラムに書き込むために使用するカスタムの SQL 式。クエリーとともにオブジェクトの直接読み込み/保存に使用します。書き込み式には、値に対して 1 つの '?' プレースホルダーが必ず含まれている必要があります。
ColumnTransformers	**@ColumnTransformer** の複数のアノテーション複数のコラムがこの動作を使用している場合に便利です。

表3.12 プロパティー

アノテーション	説明
Formula	大半の場所で、**@Column** の代わりとして使用するために使用されます。式は有効な SQL フラグメントである必要があります。
Index	データベースインデックスを定義します。
JoinFormula	大半の場所で、**@JoinColumn** の代わりとして使用するために使用されます。式は有効な SQL フラグメントである必要があります。
Parent	プロパティーを所有者 (通常は所有するエンティティー) へのポインターとして参照します。
Type	Hibernate タイプ。
TypeDef	Hibernate タイプ定義。

アノテーション	説明
TypeDefs	Hibernate タイプ定義の配列。

表3.13 単一の関連付け関連のアノテーション

アノテーション	説明
Any	**複数のエンティティータイプを参照するトール関連付けを定義します**。一致したエンティティータイプのマッチングは、メタデータの識別子コラムを使用して行われます。このようなマッピングはマージのみにする必要があります。
AnyMetaDef	**@Any** および **@ManyToAny** メタデータを定義します。
AnyMetaDefs	**@Any** および **@ManyToAny** のメタデータセットを定義します。エンティティーレベルまたはパッケージレベルで定義できます。
Fetch	指定の関連付けに使用されるフェッチストラテジーを定義します。
LazyCollection	コレクションのレイジーステータスを定義します。
LazyToOne	ToOne 関連づけのレイジーステータスを定義します (**OneToOne** または **ManyToOne**)。
NotFound	要素が関連上に見つからない場合に行うアクション。

表3.14 楽観的ロック

アノテーション	説明
OptimisticLock	アノテーションが付けられたプロパティーの変更によってエンティティーバージョンの増分がトリガーされるかどうか。アノテーションが存在しない場合は、プロパティーが楽観的ロックストラテジーに関与します (デフォルト)。
OptimisticLocking	エンティティーに適用される楽観的ロックのスタイルを定義するために使用されます。階層では、ルートエンティティーでのみ有効です。

アノテーション	説明
Source	Version および timestamp version プロパティーと併せてのオプションのアノテーションです。アノテーションの値で、タイムスタンプの生成先が決まります。

第4章 HIBERNATE QUERY 言語

4.1. HIBERNATE QUERY 言語について

Java Persistence クエリー言語の概要

Java Persistence Query Language (JPQL) は、Java Persistence API 仕様の一部として定義されるプラットフォームに依存しないオブジェクト指向のクエリー言語です。Java Persistence クエリー言語と同等の Jakarta Persistence クエリー言語は Jakarta Persistence クエリー言語であり、Jakarta Persistence 仕様 の一部として定義されています。

Java Persistence クエリー言語は、リレーショナルデータベースに保存されているエンティティーに対してクエリーを行うために使用されます。これは SQL に大きく影響を受けています。また、そのクエリーは構文の SQL クエリーに類似しますが、データベーステーブルと直接連携するのではなく Java Persistence API エンティティーオブジェクトに対して動作します。

HQL の概要

Hibernate Query Language (HQL) は、SQL と同様のパワフルなクエリー言語です。ただし、SQL と比較すると、HQL は完全にオブジェクト指向で、継承、ポリモーフィズム、関連付けなどの概念を認識します。

Hql は、Java Persistence クエリー言語のスーパーセットです。HQL クエリーは常に有効な Java Persistence クエリー言語クエリーではありませんが、Java Persistence クエリー言語のクエリーは常に有効な HQL クエリーになります。

HQL および Java Persistence クエリー言語は、クエリー操作を実行するためのタイプセーフでない方法です。条件クエリーは、クエリーにタイプセーフなアプローチを提供します。

4.2. HQL ステートメントについて

HQL および Java Persistence クエリー言語は、**SELECT**、**UPDATE**、および **DELETE** ステートメントの両方を許可します。さらに HQL では、**SQL INSERT-SELECT** と似た形式の **INSERT** ステートメントを使用できます。

以下の表は、さまざまな HQL ステートメントの Backus-Naur Form (BNF) 表記の構文を示しています。

表4.1 HQL ステートメント

ステートメント	説明
SELECT	HQL における **SELECT** ステートメントの BNF は以下のとおりです。 select_statement :: = [select_clause] from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]

ステートメント	説明
UPDATE	HQL の **UPDATE** ステートメントの BNF は、Java Persistence クエリー言語と同じになります。 update_statement ::= update_clause [where_clause] update_clause ::= UPDATE entity_name [[AS] identification_variable] SET update_item {, update_item}* update_item ::= [identification_variable.] {state_field | single_valued_object_field} = new_value new_value ::= scalar_expression | simple_entity_expression | NULL
DELETE	HQL の **DELETE** ステートメントの BNF は、Java Persistence クエリー言語と同じになります。 delete_statement ::= delete_clause [where_clause] delete_clause ::= DELETE FROM entity_name [[AS] identification_variable]
INSERT	HQL の **INSERT** ステートメントの BNF は以下の通りです。 insert_statement ::= insert_clause select_statement insert_clause ::= INSERT INTO entity_name (attribute_list) attribute_list ::= state_field[, state_field]* これと同等の Java Persistence クエリー言語はありません。

警告

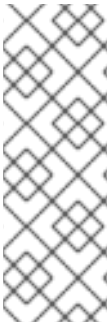
Hibernate では、DML (Data Manipulation Language) を使用して、HQL (Hibernate Query Language) を介して、マップされたデータベースにへのデータの直接挿入、更新、および削除を一括で行うことができます。

複数のオブジェクトまたは関係マッピングに違反し、オブジェクト状態に影響を与える可能性があります。オブジェクト状態は、基盤のデータベースで実行される操作によって、メモリー内オブジェクトの状態が影響を受けることはありません。インメモリーデータは、インメモリーデータが使用される場合は注意して使用する必要があります。

UPDATE ステートメントおよび DELETE ステートメント

UPDATE ステートメントおよび **DELETE** ステートメントの擬似構文は次のとおりです。

(UPDATE | DELETE) FROM?EntityName (WHERE where_conditions)?.



注記

FROM キーワードおよび **WHERE** Clause はオプションです。**FROM** 句は、残りのクエリーで使用できるオブジェクトモデルタイプのスコープを定義します。また、残りのクエリーで使用できるすべての識別変数も定義します。**WHERE** 句では、返されるインスタンスの一覧を絞り込むことができます。

UPDATE ステートメントまたは **DELETE** ステートメントの実行結果は、実際に影響を受ける (更新または削除される) 行数です。

例: 一括更新ステートメント

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Company set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

例: 一括削除ステートメント

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Company where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
```

```
        .executeUpdate();
tx.commit();
session.close();
```

Query.executeUpdate() メソッドによって返される **int** 値は、オペレーションの影響を受けた、データベース内のエンティティーの数を示します。

内部では、複数の SQL ステートメントを使用して、DML **Update** または **Delete** 更新リクエストまたは削除リクエストに応答して操作を実行する場合があります。これは、テーブルと、更新または削除が必要な結合テーブルの間に存在する関係が原因です。

たとえば、上記の例のように delete ステートメントを実行すると、**oldName** という名前の企業用の **Company** テーブルだけでなく、結合したテーブルに対しても削除が実行されます。したがって、**Employee** テーブルと多対多関係の双方向の **Company** テーブルは、前回の例の実行に成功すると、一致する結合テーブル **Company_Employee** から行を失います。

上記の **deletedEntries** 値には、この操作によって影響を受けるすべての行の数 (結合テーブルの行を含む) が含まれます。

重要

一括更新や削除の操作の実行時には、アクティブな永続コンテキストでデータベースとエンティティー間で不整合が生じる可能性があるため注意が必要です。一般的に、一括更新および削除操作は、新しい永続コンテキストのトランザクション内で、またはこのような操作の影響を受ける可能性のある状態を持つエンティティーをフェッチまたはアクセスする前にのみ実行する必要があります。

INSERT ステートメントについて

HQL は、**INSERT** ステートメントを定義する機能を追加します。これと同等の Java Persistence クエリー言語はありません。HQL **INSERT** ステートメントの Backus-Naur Form (BNF) は以下の通りです。

```
insert_statement ::= insert_clause select_statement

insert_clause ::= INSERT INTO entity_name (attribute_list)

attribute_list ::= state_field[, state_field ]*
```

tribute_list は、SQL **INSERT** ステートメントのコラム指定に似ています。マッピングされた継承に関連するエンティティーでは、名前付きエンティティーに直接定義された属性のみが **attribute_list** で使用できます。スーパークラスプロパティーは許可されておらず、サブクラスプロパティーは意味を成しません。つまり、**INSERT** ステートメントは本質的にはポリモーフィックではありません。

警告

select_statement には、有効な HQL 選択クエリーを使用できます。この場合、戻り値のタイプは挿入で想定されるタイプと一致する必要があります。現在、これは、チェックによるデータベースへの再参照を許可するのではなく、クエリーのコンパイル時に確認されます。これにより、Hibernate Types が **equal** とは逆に **equivalent** であるため、問題が発生する可能性があります。たとえば、これにより、データベースが区別を行わなかったり、変換を処理できない場合でも、**org.hibernate.type.DateType** としてマップされた属性と **org.hibernate.type.TimestampType** として定義された属性間で不一致の問題が発生する可能性があります。

id 属性には、insert ステートメントに 2 つのオプションを付与します。 **attribute_list** で **id** プロパティーを明示的に指定することができます。この場合、その値は、対応する select 式から取得されるか、生成された値が使用される場合は **attribute_list** から省略されます。後者のオプションは、データベースで (in the database) で動作する **ID** ジェネレーターを使用している場合にのみ利用できます。メモリー内 (in memory) タイプジェネレーターでこのオプションを使用しようとすると、解析中に例外が発生します。

ロック属性の場合、insert ステートメントでは、以下のいずれかのオプションが再度提供されます。**attribute_list** に属性を指定できます。この場合、その値は対応する選択式から取得されます。あるいは、**attribute_list** から除外できます。その場合は、対応する **org.hibernate.type.VersionType** で定義される **seed value** が使用されます。

例: INSERT Query ステートメント

```
String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c
where ...";
int createdEntities = s.createQuery(hqlInsert).executeUpdate();
```

例: Bulk Insert ステートメント

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Account (id, name) select c.id, c.name from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

SELECT ステートメントを使用して **id** 属性の値を指定しない場合、基礎となるデータベースが自動生成キーをサポートする限り、識別子が生成されます。この一括挿入操作の戻り値は、データベースで実際に作成されたエントリー数です。

4.3. HQL の順序について

クエリーの結果を順序付けすることもできます。**ORDER BY** は、結果の順序付けに使用される選択した値を指定するために使用されます。順序単位で有効とみなされる式のタイプには、以下が含まれます。

- ステートフィールド
- コンポーネント/組み込み可能な属性
- 算術演算、関数などのスカラー式。
- 前の式タイプに対して select 句で宣言された識別変数。

HQL は、order-by 句で参照されるすべての値が select 句内で命名されることを必須としませんが、Java Persistence クエリー言語で必要になります。データベースの移植性を求めるアプリケーションは、すべてのデータベースが、選択節で参照されていない order-by 句での参照値に対応しているわけではないことに注意してください。

order-by の個々の式は、必要な順序を示すために **ASC** (昇順) または **DESC** (下記) で修飾できます。

例: Order By

```
// legal because p.name is implicitly part of p
select p
from Person p
order by p.name

select c.id, sum( o.total ) as t
from Order o
   inner join o.customer c
group by c.id
order by t
```

4.4. コレクションメンバーの参照について

コレクション値の関連付けへの参照は、実際にはそのコレクションの値を参照します。

例: コレクション参照

```
select c
from Customer c
   join c.orders o
   join o.lineItems l
   join l.product p
where o.status = 'pending'
  and p.status = 'backorder'

// alternate syntax
select c
from Customer c,
   in(c.orders) o,
   in(o.lineItems) l
   join l.product p
where o.status = 'pending'
  and p.status = 'backorder'
```

この例では、識別変数 **o** は実際には、Customer#orders 関連付けの要素タイプであるオブジェクトモデルタイプ Order を参照します。

この例では、**IN** 構文を使用してコレクションの関連付けジョインを指定する代替構文も示しています。両方の形式は同じです。アプリケーションの選択を選択するのは、単に好みの問題です。

4.5. 修飾パス式について

これまでは、コレクション値関連づけは、コレクションの値を実際に参照していると言及されました。コレクションのタイプによっては、明示的な修飾式のセットも利用できます。

表4.2 修飾パス式

式	説明
VALUE	コレクション値を参照します。修飾子を指定しないのと同じです。意図を明示的に表示するのに役立ちます。いずれかのタイプのコレクションと値の参照に有効です。
INDEX	HQL ルールによると、これは、javax.persistence.OrderColumn アノテーションを指定して Map キーまたは List の場所 (別名 OrderColumn 値) を参照する Maps と List の両方に対して有効です。ただし、Java Persistence クエリー言語は、これを List ケースで使用するために予約し、MAP の場合に **KEY** を追加します。Jakarta Persistence プロバイダーの移植性に関心のあるアプリケーションは、この区別を認識する必要があります。
KEY	Maps にのみ有効です。マップのキーを参照します。キー自体がエンティティーである場合は、さらに移動することができます。
ENTRY	Maps にのみ有効です。Map の論理 java.util.Map.Entry タプル (キーと値の組み合わせ) を参照します。**ENTRY** はターミナルパスとしてのみ有効で、選択でのみ有効です。

例: 修飾コレクション参照

```
// Product.images is a Map<String,String> : key = a name, value = file path

// select all the image file paths (the map value) for Product#123
select i
from Product p
    join p.images i
where p.id = 123

// same as above
select value(i)
from Product p
    join p.images i
where p.id = 123
```

```
// select all the image names (the map key) for Product#123
select key(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names and file paths (the 'Map.Entry') for Product#123
select entry(i)
from Product p
    join p.images i
where p.id = 123

// total the value of the initial line items for all orders for a customer
select sum( li.amount )
from Customer c
        join c.orders o
        join o.lineItems li
where c.id = 123
  and index(li) = 1
```

4.6. HQL 関数について

HQL は、使用中の基盤のデータベースに関係なく利用可能な標準の関数を定義します。HQL は、ダイアレクトおよびアプリケーションで定義される追加の機能を理解することもできます。

4.6.1. HQL の標準化機能

以下の関数は、使用中の基盤のデータベースに関係なく HQL で利用できます。

表4.3 HQL の標準化機能

関数	説明
BIT_LENGTH	バイナリーデータの長さを返します。
CAST	SQL キャストを実行します。cast ターゲットは、使用する Hibernate マッピングタイプに名前を付ける必要があります。
EXTRACT	Datetime 値に対して SQL の抽出を実行します。展開により、年などの日付/ 時間値の一部が返されます。以下の省略形式を参照してください。
SECOND	2 つ目を抽出するための省略された抽出形式。
MINUTE	分を抽出するための省略された抽出形式。
HOUR	時を抽出するための省略された抽出形式。
DAY	日を抽出するための省略された抽出形式。

関数	説明
MONTH	月を抽出するための省略された抽出形式。
YEAR	年を抽出するための省略された抽出形式。
STR	値を文字データとしてキャストするための省略形式。

4.6.2. HQL 非標準化機能

Hibernate ダイアレクトは、その特定のデータベース製品で利用できると認識されている追加の関数を登録できます。これらは、そのデータベースまたはダイアレクトを使用する場合にのみ利用できます。データベースの移植性を目的とするアプリケーションは、このカテゴリーで関数を使用することを避ける必要があります。

アプリケーション開発者は、独自の機能のセットも提供します。これは、通常は、カスタムの SQL 関数、または SQL のスニペットのエイリアスを表します。このような関数宣言は、**org.hibernate.cfg.Configuration** の **addSqlFunction** メソッドを使用して作成されます。

4.6.3. 連結操作

HQL は、連結 (**CONCAT**) 関数のサポートのほかに連結演算子を定義します。これは Java Persistence クエリー言語では定義されないため、移植可能なアプリケーションは使用しないようにしてください。連結演算子は、SQL 連結演算子 (||) から取得されます。

例: 連結操作の例

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

4.7. 動的インスタンス化について

select 句でのみ有効な特定の式タイプがあります。Hibernate では、これを動的なインスタンス化 (dynamic instantiation) と呼びます。Java Persistence クエリー言語はこの機能の一部をサポートしており、それをコンストラクター式を呼びます。

例: 動的インスタンス化の例 - コンストラクター

```
select new Family( mother, mate, offspr )
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

ここでは Object[] を扱うのではなく、クエリーの結果として返される型安全な java オブジェクトの値をラップします。クラス参照は完全修飾する必要があり、一致するコンストラクターが必要です。

ここでクラスをマッピングする必要はありません。エンティティーを表す場合、生成されるインスタンスは NEW 状態 (管理されない) で返されます。

これは、Java Persistence クエリー言語もサポートします。HQL は、追加の動的なインスタンス化機能に対応しています。まず、クエリーはスカラー結果に対して Object[] ではなく List を返すように指定できます。

例: 動的インスタンス化の例 - リスト

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

このクエリーの結果は List<Object[]> ではなく List<List> になります。

HQL は、マップでのスカラー結果のラッピングもサポートします。

例: 動的インスタンス化の例 - マップ

```
select new map( mother as mother, offspr as offspr, mate as mate )
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr

select new map( max(c.bodyWeight) as max, min(c.bodyWeight) as min, count(*) as n )
from Cat cxt
```

このクエリーの結果は List<Object[]> ではなく List<Map<String,Object>> になります。マップのキーは、選択式に対して提供されたエイリアスによって定義されます。

4.8. HQL 述語について

述語は、**where** 句、**having** 句、検索 CASE 式の基礎を形成します。これらは式であり、通常は **TRUE** または **FALSE** に解決されます。ただし、NULL 値が関係するブール値比較は一般的に **UNKNOWN** に解決します。

HQL の述語

- Null 述語
  Null の値を確認します。基本的な属性参照、エンティティーの参照、およびパラメーターに適用できます。さらに HQL を使用すると、コンポーネント/ 組み込み可能なタイプに適用できます。

  例: NULL チェック

  ```
  // select everyone with an associated address
  select p
  from Person p
  where p.address is not null

  // select everyone without an associated address
  select p
  from Person p
    where p.address is null
  ```

- 述語と同様

文字列値に対して、同様の比較を実行します。構文は以下のとおりです。

```
like_expression ::=
      string_expression
      [NOT] LIKE pattern_value
      [ESCAPE escape_character]
```

セマンティクスは、式のような SQL のことに従います。**paattern_value** は、**string_expression** で一致を試みるパターンです。SQL と同様に、**pattern_value** はワイルドカードに _ (アンダースコア) および **%** (パーセント) を使用できます。意味は同じです。_ は、単一の文字に一致します。**%** は任意の数の文字に一致します。

オプションの **escape_character** は、**pattern_value** の _ と **%** の特別な意味をエスケープするために使用されるエスケープ文字を指定するために使用されます。これは、_ または **%** のいずれかを含むパターンを検索する必要がある場合に便利です。

例: LIKE 述語

```
select p
from Person p
where p.name like '%Schmidt'

select p
from Person p
where p.name not like 'Jingleheimmer%'

// find any with name starting with "sp_"
select sp
from StoredProcedureMetadata sp
where sp.name like 'sp|_%' escape '|'
```

- BETWEEN 演算子
  SQL **BETWEEN** 式と同様です。値が 2 つの他の値の範囲内にあるかどうかの評価を実行します。すべてのオペランドは、同等の型を持つ必要があります。

例: BETWEEN 演算子

```
select p
from Customer c
   join c.paymentHistory p
where c.id = 123
  and index(p) between 0 and 9

select c
from Customer c
where c.president.dateOfBirth
       between {d '1945-01-01'}
           and {d '1965-01-01'}

select o
from Order o
where o.total between 500 and 5000
```

```
select p
from Person p
where p.name between 'A' and 'E'
```

- IN 述語
  IN 述語は、特定の値が値の一覧にあることを確認します。構文は次のとおりです。

```
in_expression ::= single_valued_expression
        [NOT] IN single_valued_list

single_valued_list ::= constructor_expression |
        (subquery) |
        collection_valued_input_parameter

constructor_expression ::= (expression[, expression]*)
```

single_valued_expression のタイプと **single_valued_list** の個別の値は一致している必要があります。Java Persistence クエリー言語は、ここで有効なタイプを string、数値、日付、時刻、タイムスタンプ、および列挙型に制限します。Java Persistence クエリー言語では、**single_valued_expression** は以下のみを参照できます。

- 状態フィールド (state fields) は、単純な属性の用語です。具体的には、関連付けおよびコンポーネント/組み込み属性が除外されます。

- エンティティー型式。
  HQL では、**single_valued_expression** はより幅広い式タイプを参照することができます。1 価の関連づけが許可されています。コンポーネント/組み込み属性も同様です。ただし、この機能は基盤のデータベースのタプルのサポートレベルまたはロー値のコンストラクター構文 (row value constructor syntax) によって異なります。さらに、HQL では値のタイプは制限されませんが、アプリケーション開発者は、基盤のデータベースベンダーに基づいてサポートが制限される可能性があることに注意する必要があります。これは、Java Persistence クエリー言語の制限の主な理由です。

  値の一覧は多数の異なるソースから取得できます。**constructor_expression** および **collection_valued_input_parameter** では、値の一覧は空にすることはできません。少なくとも 1 つの値が含まれている必要があります。

  例: IN 述語

```
select p
from Payment p
where type(p) in (CreditCardPayment, WireTransferPayment)

select c
from Customer c
where c.hqAddress.state in ('TX', 'OK', 'LA', 'NM')

select c
from Customer c
where c.hqAddress.state in ?

select c
from Customer c
where c.hqAddress.state in (
    select dm.state
```

```
    from DeliveryMetadata dm
    where dm.salesTax is not null
)

// Not Java Persistence query language compliant!
select c
from Customer c
where c.name in (
    ('John','Doe'),
    ('Jane','Doe')
)

// Not Java Persistence query language compliant!
select c
from Customer c
where c.chiefExecutive in (
    select p
    from Person p
    where ...
)
```

4.9. 関連比較について

比較には、等号の =、>、>=、<、⇐、<> のいずれかが関係します。HQL は、<> と比べる比較演算子として != も定義します。オペランドは同じタイプである必要があります。

例: 比較例

```
// numeric comparison
select c
from Customer c
where c.chiefExecutive.age < 30

// string comparison
select c
from Customer c
where c.name = 'Acme'

// datetime comparison
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

// enum comparison
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

// boolean comparison
select c
from Customer c
where c.sendEmail = true

// entity type comparison
select p
```

```
from Payment p
where type(p) = WireTransferPayment

// entity value comparison
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

また、比較には **ALL**、**ANY**、**SOME** などの修飾子を含めることもできます。**some** および **ANY** は同義語です。

ALL 修飾子は、サブクエリーの結果のすべての値に対して比較が true の場合に true に解決します。これは、値が空の場合は false に解決します。

例: すべてのサブクエリー比較修飾子の例

```
// select all players that scored at least 3 points
// in every game.
select p
from Player p
where 3 > all (
   select spg.points
   from StatsPerGame spg
   where spg.player = p
)
```

ANY/**SOME** 修飾子は、サブクエリーの結果の値いずれかの比較が true の場合に true に解決します。これは、値が空の場合は false に解決します。

4.10. バイトコードの機能強化

4.10.1. Lazy 属性の読み込み

Lazy 属性の読み込みはバイトコードの拡張機能です。これにより、データベースからフェッチする際にエンティティーの特定の部分のみがロードされ、残りの部分もロードする必要があることを Hibernate に指示できます。これは、エンティティーの状態が必要に応じて一度に読み込まれるエンティティーを中心とするレイジーロードのプロキシーベースの概念とは異なります。バイトコード拡張では、必要に応じて個別の属性または属性グループが読み込まれます。

レイジー属性は一緒にロードするように指定できます。これは **lazy group** と呼ばれます。デフォルトでは、すべての単数属性が単一のグループに含まれます。あるレイジーな singular 属性にアクセスすると、レイジーなすべての singular 属性が読み込まれます。レイジー singular グループに反して、レイジーな複数の属性はそれぞれ個別のレイジーグループです。この動作は、**@org.hibernate.annotations.LazyGroup** アノテーションで明示的に制御できます。

```
@Entity
public class Customer {

   @Id
   private Integer id;

   private String name;

   @Basic( fetch = FetchType.LAZY )
```

```
    private UUID accountsPayableXrefId;

    @Lob
    @Basic( fetch = FetchType.LAZY )
    @LazyGroup( "lobs" )
    private Blob image;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public UUID getAccountsPayableXrefId() {
        return accountsPayableXrefId;
    }

    public void setAccountsPayableXrefId(UUID accountsPayableXrefId) {
        this.accountsPayableXrefId = accountsPayableXrefId;
    }

    public Blob getImage() {
        return image;
    }

    public void setImage(Blob image) {
        this.image = image;
    }
}
```

上記の例では、**accountsPayableXrefId** と **image** の 2 つのレイジー属性があります。これらの各属性は、異なるフェッチグループの一部です。**accountsPayableXrefId** 属性はデフォルトのフェッチグループの一部です。**accountsPayableXrefId** へのアクセスは、**image** 属性の読み込みを強制しません (逆の場合もまた然り)。

第5章 HIBERNATE サービス

5.1. HIBERNATE サービスについて

サービスは、Hibernate を提供するクラスで、さまざまなタイプの機能のプラグ可能な実装を提供します。特に、特定のサービスコントラクトインターフェイスの実装です。このインターフェイスはサービ出力ルとして知られています。実装クラスはサービス実装として知られています。一般的に、ユーザーはすべての標準サービ出力ル (上書き) の代替の実装をプラグインできます。また、サービ出力ルのベースセット (拡張) 以外に追加のサービスを定義することもできます。

5.2. サービス契約について

サービスの基本的な要件は、org.hibernate.service.Service のマーカーインターフェイスを実装することです。Hibernate は、基本的なタイプの安全のためにこの内部で使用します。

オプションとして、サービスは org.hibernate.service.spi.Startable および org.hibernate.service.spi.Stoppable インターフェイスを実装し、開始および停止の通知を受信することもできます。別のオプションのサービスコントラクトは org.hibernate.service.spi.Manageable で、Jakaarta Management 統合が有効になっていると、Jakarta Management でサービスが管理可能であるとマークされます。

5.3. サービス依存関係の種類

サービスは、以下の方法のいずれかを使用して他のサービスの依存関係を宣言できます。

@org.hibernate.service.spi.InjectService

単一のパラメーターを受け入れ、**@InjectService** でアノテーションが付けられたサービス実装クラスのメソッドは 、別のサービスの挿入を要求していると見なされます。
デフォルトでは、method パラメーターのタイプは、インジェクトされるサービ出力ルになることが想定されます。パラメーターのタイプがサービ出力ルと異なる場合は、**InjectService** の **serviceRole** 属性を使用してロールに明示的な名前を付ける必要があります。

デフォルトでは、インジェクトされたサービスは必須とみなされ、名前付き依存サービスがないと、起動に失敗します。インジェクトされるサービスがオプションの場合、**InjectService** の必要な属性は **false** として宣言される必要があります。デフォルトは **true** です。

org.hibernate.service.spi.ServiceRegistryAwareService

次のアプローチは、単一の **injectServices** メソッドを宣言する、オプションのサービスインターフェイス **org.hibernate.service.spi.ServiceRegistryAwareService** をサービスが実装するプルアプローチです。
Hibernate は起動時に、**org.hibernate.service.ServiceRegistry** 自体をこのインターフェイスを実装するサービスに挿入します。その後、**ServiceRegistry** 参照を使用して必要な追加サービスを見つけることができます。

5.3.1. サービスレジストリー

5.3.1.1. ServiceRegistry について

中央のサービス API はサービス自体以外に、org.hibernate.service.ServiceRegistry インターフェイスです。サービスレジストリーの主な目的は、サービスへのアクセスを保持し、管理し、提供することです。

サービスレジストリーは階層的です。あるレジストリーのサービスは、同じレジストリーのサービスや任意の親レジストリーに依存して使用できます。

org.hibernate.service.ServiceRegistryBuilder を使用して org.hibernate.service.ServiceRegistry インスタンスをビルドします。

ServiceRegistryBuilder を使用した ServiceRegistry の作成例

```
ServiceRegistryBuilder registryBuilder =
    new ServiceRegistryBuilder( bootstrapServiceRegistry );
    ServiceRegistry serviceRegistry = registryBuilder.buildServiceRegistry();
```

5.3.2. カスタムサービス

5.3.2.1. カスタムサービスについて

org.hibernate.service.ServiceRegistry を構築すると、不変とみなされます。このサービス自体は再設定を受け入れる可能性がありますが、ここでの不変性はサービスの追加または置き換えを意味します。そのため、**org.hibernate.service.ServiceRegistryBuilder** で利用できる別のロールは、これから生成される **org.hibernate.service.ServiceRegistry** に含まれるサービスの調整を許可することです。

カスタムサービスについて **org.hibernate.service.ServiceRegistryBuilder** に指示する方法は 2 つあります。

- **org.hibernate.service.spi.BasicServiceInitiator** クラスを実装し、サービスクラスのオンデマンド構築を制御し、**addInitiator** メソッドを使用して **org.hibernate.service.ServiceRegistryBuilder** に追加します。
- サービスクラスをインスタンス化し、**addService** メソッドを使用して、**org.hibernate.service.ServiceRegistryBuilder** に追加します。

新規サービ出力ルの追加やサービス実装の置き換えなどサービスの上書きなど、レジストリーの拡張にはいずれかのアプローチが有効です。

例: ServiceRegistryBuilder を使用して、既存のサービスをカスタムサービスに置き換える

```
ServiceRegistryBuilder registryBuilder =
    new ServiceRegistryBuilder(bootstrapServiceRegistry);
registryBuilder.addService(JdbcServices.class, new MyCustomJdbcService());
ServiceRegistry serviceRegistry = registryBuilder.buildServiceRegistry();

public class MyCustomJdbcService implements JdbcServices{

    @Override
    public ConnectionProvider getConnectionProvider() {
        return null;
    }

    @Override
    public Dialect getDialect() {
        return null;
    }

    @Override
    public SqlStatementLogger getSqlStatementLogger() {
```

```
        return null;
    }

    @Override
    public SqlExceptionHelper getSqlExceptionHelper() {
        return null;
    }

    @Override
    public ExtractedDatabaseMetaData getExtractedMetaDataSupport() {
        return null;
    }

    @Override
    public LobCreator getLobCreator(LobCreationContext lobCreationContext) {
        return null;
    }

    @Override
    public ResultSetWrapper getResultSetWrapper() {
        return null;
    }
}
```

5.3.3. Boot-Strap レジストリー

5.3.3.1. Boot-strap レジストリーについて

boot-strap レジストリーは、ほとんどの機能を機能させるために絶対に利用できるようにする必要のあるサービスを保持します。ここでの主要なサービスは、**ClassLoaderService** です。これは最適な例です。設定ファイルを解決する場合でも、クラスローディングサービス (例: リソース検索) へのアクセスが必要になります。これは、通常の使用時に親ではなく、ルートレジストリーです。

boot-strap レジストリーのインスタンスは、**org.hibernate.service.BootstrapServiceRegistryBuilder** クラスを使用して作成されます。

BootstrapServiceRegistryBuilder の使用

例: BootstrapServiceRegistryBuilder の使用

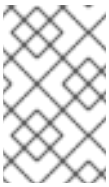
```
BootstrapServiceRegistry bootstrapServiceRegistry =
    new BootstrapServiceRegistryBuilder()
    // pass in org.hibernate.integrator.spi.Integrator instances which are not
    // auto-discovered (for whatever reason) but which should be included
    .with(anExplicitIntegrator)
    // pass in a class loader that Hibernate should use to load application classes
    .with(anExplicitClassLoaderForApplicationClasses)
    // pass in a class loader that Hibernate should use to load resources
    .with(anExplicitClassLoaderForResources)
    // see BootstrapServiceRegistryBuilder for rest of available methods
    ...
    // finally, build the bootstrap registry with all the above options
    .build();
```

5.3.3.2. BootstrapRegistry サービス

org.hibernate.service.classloading.spi.ClassLoaderService

Hibernate はクラスローダーと対話する必要があります。ただし、Hibernate または任意のライブラリーと対話する方法は、アプリケーションをホストしているランタイム環境によって異なります。アプリケーションサーバー、OSGi コンテナー、およびその他のモジュラークラスローディングシステムでは、非常に特殊なクラスローディング要件が課せられています。このサービスは、この環境的な複雑性からの抽象化を Hibernate に提供します。また、同様に重要な点として、単一スワップ可能なコンポーネントでこれを実行します。
クラスローダーとの対話では、Hibernate には以下の機能が必要になります。

- アプリケーションクラスを検索する機能
- インテグレーションクラスを検索する機能
- プロパティーファイルや XML ファイルなどのリソースを検索する機能
- **java.util.ServiceLoader** をロードする機能



注記

現在、アプリケーションクラスをロードする機能と、統合クラスをロードする機能が、サービス上の単一の **load class** 機能に統合されています。これは、今後のリリースで変更される可能性があります。

org.hibernate.integrator.spi.IntegratorService

アプリケーション、アドオン、およびその他のモジュールは Hibernate と統合する必要があります。前述の方法では、各モジュールの登録を調整するために、通常アプリケーションなどのコンポーネントが必要でした。この登録は、各モジュールのインテグレーターに代わって実施されました。
このサービスは、検出機能に重点を置いています。**org.hibernate.integrator.spi.Integrator** のコントラクトの実装を検出するために、**org.hibernate.service.classloading.spi.ClassLoaderService** が提供する標準の Java **java.util.ServiceLoader** 機能を利用します。

インテグレーターは、単に **/META-INF/services/org.hibernate.integrator.spi.Integrator** という名前のファイルを定義し、これをクラスパスで利用できるようにします。

このファイルは、**java.util.ServiceLoader** メカニズムによって使用されます。これは、1行ずつ **org.hibernate.integrator.spi.Integrator** インターフェイスを実装するクラスの完全修飾名を一覧表示します。

5.3.4. SessionFactory レジストリー

すべてのレジストリータイプのインスタンスを指定の **org.hibernate.SessionFactory** のターゲットとして扱うのがベストプラクティスですが、このグループのサービスのインスタンスは単一の **org.hibernate.SessionFactory** に明示的に所属します。

相違点は、開始タイミングに関する事項です。通常、**org.hibernate.SessionFactory** にアクセスするにはアクセスが必要になります。この特別なレジストリーは **org.hibernate.service.spi.SessionFactoryServiceRegistry** です。

5.3.4.1. SessionFactory サービス

org.hibernate.event.service.spi.EventListenerRegistry

説明

イベントリスナーを管理するためのサービスです。

イニシエーター

org.hibernate.event.service.internal.EventListenerServiceInitiator

実装

org.hibernate.event.service.internal.EventListenerRegistryImpl

5.3.5. インテグレーター

org.hibernate.integrator.spi.Integrator は、開発者が機能している **SessionFactory** をビルドするプロセスにフックできるようにする簡単な方法を提供することを目的としています。**org.hibernate.integrator.spi.Integrator** インターフェイスは関連する以下のメソッドを定義します。

- **integrate** では、ビルドプロセスへのフックが可能になります。
- **disintegrate** では、**SessionFactory** をシャットダウンできます。

注記

統合のオーバーロード形式である **org.hibernate.integrator.spi.Integrator** で定義されている 3 つ目のメソッドがあります。これは、**org.hibernate.cfg.Configuration** ではなく、**org.hibernate.metamodel.source.MetadataImplementor** を受け入れます。

IntegratorService で利用できる検出アプローチに加え、**BootstrapServiceRegistry** のビルド時にアプリケーションが手動でインテグレーター実装を登録できます。

5.3.5.1. インテグレーターのユースケース

org.hibernate.integrator.spi.Integrator の主なユースケースは、イベントリスナーを登録し、サービスを提供します。**org.hibernate.integrator.spi.ServiceContributingIntegrator** を参照してください。

例: イベントリスナーの登録

```
public class MyIntegrator implements org.hibernate.integrator.spi.Integrator {

   public void integrate(
         Configuration configuration,
         SessionFactoryImplementor sessionFactory,
         SessionFactoryServiceRegistry serviceRegistry) {
      // As you might expect, an EventListenerRegistry is the thing with which event listeners are
registered  It is a
      // service so we look it up using the service registry
      final EventListenerRegistry eventListenerRegistry =
serviceRegistry.getService(EventListenerRegistry.class);

      // If you wish to have custom determination and handling of "duplicate" listeners, you would have
to add an
      // implementation of the org.hibernate.event.service.spi.DuplicationStrategy contract like this
      eventListenerRegistry.addDuplicationStrategy(myDuplicationStrategy);
```

```
        // EventListenerRegistry defines 3 ways to register listeners:
        //     1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners(EventType.AUTO_FLUSH, myCompleteSetOfListeners);
        //     2) This form adds the specified listener(s) to the beginning of the listener chain
        eventListenerRegistry.prependListeners(EventType.AUTO_FLUSH,
myListenersToBeCalledFirst);
        //     3) This form adds the specified listener(s) to the end of the listener chain
        eventListenerRegistry.appendListeners(EventType.AUTO_FLUSH,
myListenersToBeCalledLast);
    }
}
```

第6章 HIBERNATE ENVERS

6.1. HIBERNATE ENVERS について

Hibernate Enver は監査およびバージョン管理システムであり、JBoss EAP に永続クラスへのこれまでの変更を追跡する手段を提供します。監査テーブルは **@Audited** アノテーションが付けられたエンティティーに対して作成されています。これは、エンティティーに加えられた変更の履歴が保存されます。その後、データを取得し、クエリーできます。

Envers では、開発者は以下を行うことができます。

- Jakarta Persistence 仕様によって定義されたすべてのマッピングの監査
- Jakarta Persistence 仕様を拡張するすべての hibernate マッピングの監査
- ネイティブ Hibernate API を使用する、または、これによってマッピングされるエンティティーの監査
- リビジョンエンティティーを使用した各リビジョンのデータのログ記録
- 履歴データのクエリー

6.2. 永続クラスの監査について

永続クラスの監査は、Hibernate 環境と **@Audited** アノテーションで JBoss EAP で実行されます。アノテーションがクラスに適用されると、エンティティーの改訂履歴を保存するテーブルが作成されます。

クラスに変更が行われるたびに、エントリーが監査テーブルに追加されます。エントリーにはクラスへの変更が含まれ、リビジョン番号が指定されます。これは、変更をロールバックしたり、以前のリビジョンを表示したりできることを意味します。

6.3. 監査ストラテジー

6.3.1. 監査ストラテジーについて

監査ストラテジーは、監査情報の永続化、クエリー、および保存方法を定義します。Hibernate Evers には現在、以下の 2 つの監査ストラテジーがあります。

デフォルトの監査ストラテジー

- このストラテジーは、開始リビジョンとともに監査データの保存を永続化します。監査されたテーブルで挿入、更新、削除される行ごとに、有効性の開始リビジョンとともに、監査テーブルに 1 つ以上の行を挿入します。
- 監査テーブルの行は、挿入後には更新されません。監査情報のクエリーはサブクエリーを使用して、監査テーブルの適用可能な行を選択します。これはスピードが遅く、インデックス作成が困難です。

有効性監査ストラテジー

- このストラテジーは、監査上の開始リビジョンと最後のリビジョンを保存します。監査されたテーブルで挿入、更新、削除される行ごとに、有効性の開始リビジョンとともに、監査テーブルに 1 つ以上の行を挿入します。

- 同時に、以前の監査行 (利用可能な場合) の終了リビジョンフィールドは、このリビジョンに設定されます。監査情報のクエリーは、サブクエリーの代わりに、**between start and end revision** の間で使用できます。つまり、追加の更新により、監査情報の永続化は多少遅くなりますが、監査情報の取得はかなり速くなります。

- これは、インデックスを追加して改善することもできます。

監査の詳細は、永続クラスの監査について を参照してください。アプリケーションの監査ストラテジーを設定するには、Set the Auditing Strategy を参照してください。

6.3.2. 監査ストラテジーの設定

JBoss EAP では、以下の 2 つの監査ストラテジーがサポートされています。

- デフォルトの監査ストラテジー

- 有効性監査ストラテジー

監査ストラテジーの定義
アプリケーションの **persistence.xml** ファイルで **org.hibernate.envers.audit_strategy** プロパティーを設定します。プロパティーが **persistence.xml** ファイルで設定されていない場合は、デフォルトの監査ストラテジーが使用されます。

デフォルトの監査ストラテジーの設定

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.DefaultAuditStrategy"/>
```

有効性監査ストラテジーの設定

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.ValidityAuditStrategy"/>
```

6.3.3. Jakarta Persistence エンティティーへの監査サポートの追加

手順

JBoss EAP は Hibernate Envers を介してエンティティー監査を使用し、永続クラスのこれまでの変更を追跡します。ここでは、Jakarta Persistence エンティティーの監査サポートの追加について説明します。

Jakarta Persistence エンティティーへの監査サポートの追加

1. デプロイメントに適した利用可能な監査パラメーターを設定します。詳細は Configure Envers Parameters を参照してください。

2. 監査する Jakarta Persistence エンティティーを開きます。

3. **org.hibernate.envers.Audited** インターフェイスをインポートします。

4. 監査する各フィールドまたはプロパティーに **@Audited** アノテーションを適用するか、クラス全体に対して 1 度適用します。

   例: 2 つのフィールドの監査

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
public class Person {
    @Id
    @GeneratedValue
    private int id;

    @Audited
    private String name;

    private String surname;

    @ManyToOne
    @Audited
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

例: クラス全体の監査

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
@Audited
public class Person {
    @Id
    @GeneratedValue
    private int id;

    private String name;

    private String surname;

    @ManyToOne
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

Jakarta Persistence エンティティーが監査用に設定されている場合は、履歴の変更を保存するために **_AUD** というテーブルが作成されます。

6.4. CONFIGURATION (設定)

6.4.1. Envers パラメーターの設定

JBoss EAP は About Hibernate Envers を介してエンティティー監査を使用して、永続クラスのこれまでの変更を追跡します。

利用可能な **Envers** パラメーターの設定

1. アプリケーションの **persistence.xml** ファイルを開きます。
2. 必要に応じて Envers プロパティーを追加、削除、または設定します。利用可能なプロパティーの一覧は、Envers Configuration Properties を参照してください。

   例: Envers パラメーター

```
<persistence-unit name="mypc">
  <description>Persistence Unit.</description>
  <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create-drop" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.cache.use_second_level_cache" value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
    <property name="hibernate.generate_statistics" value="true" />
    <property name="org.hibernate.envers.versionsTableSuffix" value="_V" />
    <property name="org.hibernate.envers.revisionFieldName" value="ver_rev" />
  </properties>
</persistence-unit>
```

6.4.2. ランタイム時の監査の有効化または無効化

ランタイム時のエンティティーバージョンの監査の有効化または無効化

1. **AuditEventListener** クラスをサブクラス化します。
2. Hibernate イベントで呼び出される以下のメソッドを上書きします。
   - **onPostInsert**
   - **onPostUpdate**
   - **onPostDelete**
   - **onPreUpdateCollection**
   - **onPreRemoveCollection**
   - **onPostRecreateCollection**
3. イベントのリスナーとしてサブクラスを指定します。
4. 変更を監査する必要があるかどうかを判断します。

5. 変更を監査する必要がある場合には、スーパークラスに呼び出しを渡します。

6.4.3. 条件監査の設定

Hibernate Envers は、一連のイベントリスナーを使用して、さまざまな Hibernate イベントに対する応答で監査データを永続化します。Envers JAR がクラスパスにある場合、これらのリスナーは自動的に登録されます。

条件監査の実装

1. **persistence.xml** ファイルで **hibernate.listeners.envers.autoRegister** を false に設定します。

2. 上書きされる各イベントリスナーをサブクラス化します。条件付き監査ロジックをサブクラスに配置し、監査を実行する必要がある場合は super メソッドを呼び出します。

3. **org.hibernate.envers.event.EnversIntegrator** に類似した、**org.hibernate.integrator.spi.Integrator** のカスタム実装を作成します。デフォルトのクラスではなく、ステップ 2 で作成したイベントリスナーサブクラスを使用します。

4. **META-INF/services/org.hibernate.integrator.spi.Integrator** ファイルを JAR に追加します。このファイルには、インターフェイスを実装するクラスの完全修飾名が含まれている必要があります。

6.4.4. Envers 設定プロパティー

表6.1 エンティティーデータのバージョン管理設定パラメーター

プロパティー名	デフォルト値	説明
org.hibernate.envers.audit_table_prefix		監査されたエンティティーの名前の前に付加される文字列。監査情報を保持するエンティティーの名前を作成します。
org.hibernate.envers.audit_table_suffix	_AUD	監査されたエンティティーの名前に付加される文字列。監査情報を保持するエンティティーの名前を作成します。たとえば、テーブル名が **Person** のエンティティーが監査されると、Envers は、履歴データを格納する **Person_AUD** と呼ばれるテーブルを生成します。
org.hibernate.envers.revision_field_name	REV	改訂番号を保持する監査エンティティーのフィールド名。
org.hibernate.envers.revision_type_field_name	REVTYPE	改訂の種類を保持する監査エンティティーのフィールド名。現在の改訂の種類には、挿入を行う **add**、変更を行う **mod**、削除を行う **del** があります。

プロパティー名	デフォルト値	説明
org.hibernate.envers.revision_on_collection_change	true	このプロパティーは、所有の変更のない関係フィールドがある場合にリビジョンを生成するかどうかを決定します。これは、一対多数の関係のコレクションであるか、一対一の関係で **mappedBy** 属性を使用するフィールドになります。
org.hibernate.envers.do_not_audit_optimistic_locking_field	true	true の場合、(**@Version** でアノテーション付けされた) 楽観的ロックに使用されるプロパティーは自動的に監査から除外されます。
org.hibernate.envers.store_data_at_delete	false	このプロパティーは、ID のみではなく、エンティティーデータを削除するときに、エンティティーデータをリビジョンに保存するかどうかを定義します。その他のすべてのプロパティーは null とマークされます。通常、データは最後から 2 番目のバージョンにあるため、これは必須ではありません。ただし時折、最後のリビジョンでアクセスすることが簡単で、より効率的になる場合があります。ただし、これは、削除前に含まれるエンティティーのデータが 2 回保存されることを意味します。
org.hibernate.envers.default_schema	null (通常のテーブルと同じ)	監査テーブルに使用されるデフォルトのスキーマ名。**@AuditTable(schema="...")** アノテーションを使用して上書き可能。これがない場合、スキーマは通常のテーブルのスキーマと同じになります。
org.hibernate.envers.default_catalog	null (通常のテーブルと同じ)	監査テーブルに使用するデフォルトのカタログ名。**@AuditTable(catalog="...")** アノテーションを使用して上書き可能。存在しない場合、カタログは通常のテーブルのカタログと同じになります。
org.hibernate.envers.audit_strategy	**org.hibernate.envers.strategy.DefaultAuditStrategy**	このプロパティーは、監査データの永続化時に使用する必要のある監査ストラテジーを定義します。デフォルトでは、エンティティーが変更されたリビジョンのみが保存されます。または、**org.hibernate.envers.strategy.ValidityAuditStrategy** では、開始リビジョンと終了リビジョンの両方を保存します。これらを組み合わせて、監査行が有効であったタイミングを定義します。

プロパティー名	デフォルト値	説明
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	REVEND	監査エンティティーで終了リビジョン番号を保持するコラムの名前。このプロパティーは、妥当な監査ストラテジーが使用される場合にのみ有効です。
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	false	このプロパティーは、終了リビジョン自体に加えて、データが最後に有効であった終了リビジョンのタイムスタンプも保存すべきかどうかを定義します。これは、テーブルパーティションを使用して、リレーショナルデータベースから古い監査レコードをパージする場合に便利です。パーティショニングには、テーブルに存在するコラムが必要です。このプロパティーは **ValidityAuditStrategy** が使用される場合にのみ評価されます。
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEND_TSTMP	データがまだ有効であった時点での終了リビジョンのタイムスタンプのコラム名。**ValidityAuditStrategy** が使用され **org.hibernate.envers.audit_strategy_validity_store_revend_timestamp** が true に評価します。

6.5. 監査情報のクエリー

6.5.1. クエリーを介した監査情報の取得

Hibernate Enver は、クエリーを介して監査情報を取得する機能を提供します。

注記

監査されたデータでのクエリーは、関連する subselect を伴うため、多くの場合、**live** データに対するクエリーよりもはるかに遅くなります。

所定リビジョンでのクラスのエンティティーのクエリー

このタイプのクエリーのエントリーポイントは以下の通りです。

```
AuditQuery query = getAuditReader()
    .createQuery()
    .forEntitiesAtRevision(MyEntity.class, revisionNumber);
```

制約は **AuditEntity** ファクトリークラスを使用して指定できます。以下のクエリーは、**name** プロパティーが **John** と同等のエンティティーのみを選択します。

```
query.add(AuditEntity.property("name").eq("John"));
```

以下のクエリーは、特定のエンティティーに関連するエンティティーのみを選択します。

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

その結果は、順序付け、制限でき、集計と予測 (グループ化を除く) セットを設定することができます。以下の例は、フルクエリーです。

```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision(Person.class, 12)
    .addOrder(AuditEntity.property("surname").desc())
    .add(AuditEntity.relatedId("address").eq(addressId))
    .setFirstResult(4)
    .setMaxResults(2)
    .getResultList();
```

所定クラスエンティティーが変更されるクエリーリビジョン

このタイプのクエリーのエントリーポイントは以下の通りです。

```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true);
```

このクエリーには、前述の例と同じように制約を追加できます。このクエリーには、追加の方法が考えられます。

AuditEntity.revisionNumber()

監査済みエンティティーが変更されたリビジョン番号の制約、展開、および順序を指定します。

AuditEntity.revisionProperty(propertyName)

監査済みエンティティーが変更されたリビジョンに対応する、リビジョンエンティティーのプロパティーに対する制約、調整および順序付けを指定します。

AuditEntity.revisionType()

リビジョンのタイプ (ADD、MOD、DEL) へのアクセスを提供します。

クエリー結果は必要に応じて調整できます。以下のクエリーは、リビジョン番号 42 の後に **entityId** が変更された **MyEntity** クラスのエンティティーが変更する最小リビジョン番号を選択します。

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.id().eq(entityId))
    .add(AuditEntity.revisionNumber().gt(42))
    .getSingleResult();
```

リビジョンのクエリーは、プロパティーを最小限に抑えたり、最大値にしたりすることもできます。以下のクエリーは、特定のエンティティーの **actualDate** の値が指定の値よりも大きくなるリビジョンを選択しますが、可能な限り小さくなります。

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    // We are only interested in the first revision
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.property("actualDate").minimize()
```

```
        .add(AuditEntity.property("actualDate").ge(givenDate))
        .add(AuditEntity.id().eq(givenEntityId)))
    .getSingleResult();
```

minimize() および **maximize()** のメソッドは基準を返します。これは、制約を追加でき、maximized/minimized プロパティーを持つエンティティーによって満たされる必要があります。

クエリーの作成時に、2 つのブール値パラメーターが渡されます。

selectEntitiesOnly

このパラメーターは、明示的な子が設定されていない場合のみ有効です。
true の場合、クエリーの結果は、指定された制約を満たすリビジョンで変更されたエンティティーのリストになります。
false の場合、結果は 3 つの要素の配列のリストになります。最初の要素は変更されたエンティティーインスタンスになります。次は、リビジョンデータを含むエンティティーです。カスタムエンティティーが使用されていない場合、**DefaultRevisionEntity** のインスタンスになります。3 番目の要素アレイはリビジョンのタイプ (ADD、MOD、DEL) になります。

selectDeletedEntities

このパラメーターは、エンティティーが削除されたリビジョンを結果に含める必要があるかどうかを指定します。true の場合、エンティティーにはリビジョンタイプ **DEL** が指定され、id を除くすべてのフィールドの値は **null** になります。

所定プロパティーを変更するエンティティーのクエリーの修正

以下のクエリーは、指定の id を持つ **MyEntity** のすべてのリビジョンを返します。ここでは、**realDate** プロパティーが変更になりました。

```
AuditQuery query = getAuditReader().createQuery()
  .forRevisionsOfEntity(MyEntity.class, false, true)
  .add(AuditEntity.id().eq(id));
  .add(AuditEntity.property("actualDate").hasChanged())
```

hasChanged 条件は、追加の基準と組み合わせることができます。以下のクエリーは、revisionNumber の生成時に **MyEntity** の水平スライスを返します。これは **prop1** が変更されたリビジョンに限定されますが、**prop2** には適用されません。

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

作成されたセットには、revisionNumber よりも少ない数字のリビジョンも含まれます。これは、このクエリーを Return all **MyEntities** changed in revisionNumber with **prop1** modified and **prop2** untouched として読み取ることができないことを意味します。

以下のクエリーは、**forEntitiesModifiedAtRevision** クエリーを使用して、この結果が返される方法を示しています。

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesModifiedAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

指定リビジョンでのクエリーエンティティーの変更

以下の例は、特定のリビジョンで変更されたエンティティーの基本クエリーを示しています。指定されたリビジョンで変更されたエンティティー名と対応する Java クラスを取得することができます。

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()
    .getCrossTypeRevisionChangesReader().findEntityTypes(revisionNumber);
```

org.hibernate.envers.CrossTypeRevisionChangesReader からもアクセス可能なクエリーは多数あります。

List<Object> findEntities(Number)

指定のリビジョンで変更された (追加、更新、削除された) すべての監査済みエンティティーのスナップショットを返します。**n+1** SQL クエリーを実行します。**n** は、指定されたリビジョン内で変更された複数の異なるエンティティークラスです。

List<Object> findEntities(Number, RevisionType)

変更タイプでフィルターされた特定のリビジョンで変更 (追加、更新、または削除) されたすべての監査済みエンティティーのスナップショットを返します。**n+1** SQL クエリーを実行します。**n** は、指定されたリビジョン内で変更された複数の異なるエンティティークラスです。Map<RevisionType, List<Object>>

findEntitiesGroupByRevisionType(Number)

変更操作 (追加、更新、または削除など) でグループ化されたエンティティースナップショットの一覧が含まれるマップを返します。**3n+1** SQL クエリーを実行します。**n** は、指定されたリビジョン内で変更された複数の異なるエンティティークラスです。

6.5.2. 参照されるエンティティーのプロパティーを使用したエンティティーの関連付けのトラバース

参照されるエンティティーのプロパティーを使用して、クエリー内のエンティティーをトラバースできます。これにより、1 対 1 および多対 1 の関連付けをクエリーできます。

以下の例は、クエリーでエンティティーをトラバースする方法の一部を示しています。

- リビジョン番号 1 では、所有者が 20 歳であるか、住所番号 30 に居住している車を見つけます。そして、車のメーカーで設定した結果の順序づけを行います。

```
List<Car> resultList = auditReader.createQuery()
            .forEntitiesAtRevision( Car.class, 1 )
            .traverseRelation( "owner", JoinType.INNER, "p" )
            .traverseRelation( "address", JoinType.INNER, "a" )
            .up().up().add( AuditEntity.disjunction().add(AuditEntity.property( "p", "age" )
                .eq( 20 ) ).add( AuditEntity.property( "a", "number" ).eq( 30 ) ) )
            .addOrder( AuditEntity.property( "make" ).asc() ).getResultList();
```

- リビジョン番号 1 では、所有者の年齢と住所番号が同じ車を見つけます。

```
Car result = (Car) auditReader.createQuery()
            .forEntitiesAtRevision( Car.class, 1 )
            .traverseRelation( "owner", JoinType.INNER, "p" )
            .traverseRelation( "address", JoinType.INNER, "a" )
            .up().up().add(AuditEntity.property( "p", "age" )
                .eqProperty( "a", "number" ) ).getSingleResult();
```

- リビジョン番号 1 では、所有者の年齢が 20 歳であるか、所有者がない車すべてを見つけます。

```
List<Car> resultList = auditReader.createQuery()
            .forEntitiesAtRevision( Car.class, 1 )
            .traverseRelation( "owner", JoinType.LEFT, "p" )
            .up().add( AuditEntity.or( AuditEntity.property( "p", "age").eq( 20 ),
                AuditEntity.relatedId( "owner" ).eq( null ) ) )
            .addOrder( AuditEntity.property( "make" ).asc() ).getResultList();
```

- リビジョン番号 1 では、メーカーが car3 であり、所有者の年齢が 30 歳であるか、所有者のいない車すべてを見つけます。

```
List<Car> resultList = auditReader.createQuery()
            .forEntitiesAtRevision( Car.class, 1 )
            .traverseRelation( "owner", JoinType.LEFT, "p" )
            .up().add( AuditEntity.and( AuditEntity.property( "make" ).eq( "car3" ),
AuditEntity.property( "p", "age" ).eq( 30 ) ) )
            .getResultList();
```

- リビジョン番号 1 では、メーカーが car3 であるか、所有者の年齢が 10 歳であるか、あるいは所有者がいない車すべてを見つけます。

```
List<Car> resultList = auditReader.createQuery()
            .forEntitiesAtRevision( Car.class, 1 )
            .traverseRelation( "owner", JoinType.LEFT, "p" )
            .up().add( AuditEntity.or( AuditEntity.property( "make" ).eq( "car3" ),
AuditEntity.property( "p", "age" ).eq( 10 ) ) )
            .getResultList();
```

6.6. パフォーマンスチューニング

6.6.1. 他のバッチロードアルゴリズム

Hibernate では、join、select、subselect、および batch の 4 つののフェッチストラテジーのいずれかを使用して関連付けのデータを読み込むことができます。これら 4 つのストラテジーにおいて、バッチローディングでは、select フェッチの最適化ストラテジーであるため、これらのストラテジーの外で、パフォーマンスを大幅に向上させることができます。このストラテジーでは、Hibernate はプライマリーキーまたは外部キーのリストを指定して、単一の SELECT ステートメントでエンティティーインスタンスまたはコレクションのバッチを取得します。batch フェッチは、レイジー select フェッチストラテジーの最適化です。

batch フェッチの設定方法は、クラス当たりのレベルまたはコレクション当たりのレベルです。

- クラス当たりのレベル
  Hibernate がクラス当たりのレベルでデータを読み込む場合は、クエリーの実行時にロードする関連付けのバッチサイズが必要になります。たとえば、ランタイム時に、セッションに 30 個のインスタンスの **car** オブジェクトがロードされているとします。各 **car** オブジェクトは **owner** オブジェクトに属します。すべての **car** オブジェクトを繰り返し処理し、所有者を要求する場合、**lazy** 読み込みでは、Hibernate は所有者ごとに 30 個の select ステートメントを発行します。これはパフォーマンスのボトルネックです。

  代わりに、Hibernate に対して、クエリーを経由する前に所有者の次のバッチのデータを事前に読み込むように指示することもできます。**owner** オブジェクトがクエリーされると、Hibernate は同じ SELECT ステートメントで多くのこれらのオブジェクトをクエリーします。

事前にクエリーする **owner** オブジェクトの数は、設定時に指定された **batch-size** パラメーターによって異なります。

```
<class name="owner" batch-size="10"></class>
```

これにより、Hibernate は、後で必要と予想される 10 個以上の **owner** オブジェクトをクエリーするようになります。ユーザーが **car A** の **owner** にクエリーすると、**car B** の **owner** はすでにバッチロードの一部として読み込まれている可能性があります。ユーザーがデータベースに移動 (および SELECT ステートメントを実行) する代わりに、実際に **car B** の **owner** が必要な場合は、現在のセッションから値を取得できます。

batch-size パラメーターに加えて、Hibernate 4.2.0 ではバッチロードのパフォーマンスを強化する新しい設定項目が導入されました。設定項目は **Batch Fetchstyle** 設定と呼ばれ、**hibernate.batch_fetch_style** パラメーターによって指定されます。

LEGACY、PADDED、DYNAMIC といった異なるバッチフェッチスタイルがサポートされています。使用するスタイルを指定するには、**org.hibernate.cfg.AvailableSettings#BATCH_FETCH_STYLE** を使用します。

- LEGACY: 読み込みのレガシースタイルでは、**ArrayHelper.getBatchSizes(int)** に基づく事前ビルドされたバッチサイズセットが使用されます。バッチは、既存のバッチ可能な識別子の数からの次に小さな事前ビルドされたバッチサイズを使用してロードされます。
  上記の例で、**batch-size** が 30 の場合、事前ビルドされたバッチサイズは [30, 15, 10, 9, 8, 7,..., 1] になります。ロード 29 識別子のバッチ処理を試みると、15、10、および 4 のバッチが発生します。対応する SQL クエリーは 3 つあり、各クエリーはデータベースから 15、10、および 4 の所有者 (owner) を読み込みます。

- PADDED - PADDED は、バッチローディングの LEGACY スタイルに似ています。依然として事前ビルドされたバッチサイズを使用していますが、次に大きなバッチサイズを使用し、追加の識別子プレースホルダーをパディングします。
  上記の例と同様に、30 個の owner オブジェクトを初期化する場合、データベースに対してクエリーが実行されるのは 1 つのみとなります。

  ただし、29 個の owner オブジェクトが初期化される場合でも、Hibernate は依然としてバッチサイズ 30 の SQL select ステートメントのみを実行し、識別子が連続する追加スペースがパディングされます。

- Dynamic - バッチサイズの制限に準拠していますが、このスタイルのバッチロードは、実際に読み込まれるオブジェクト数を使用して SQL SELECT ステートメントを動的に構築します。
  たとえば、owner オブジェクトが 30 個で、最大バッチサイズが 30 の場合、30 個の owner オブジェクトを取得する呼び出しは、1 つの SQL SELECT ステートメントになります。35 個を取得する呼び出しは、バッチサイズ 30 と 5 の 2 つの SQL ステートメントになります。Hibernate は、必要な数である 5 を維持するために 第 2 の SQL ステートメントを動的に変更します。また、バッチサイズは制限の 30 のままに保持します。これは、PADDED バージョンとは異なります。第 2 の SQL は PADDED されません。LEGACY スタイルとは異なり、第 2 の SQL ステートメントには固定サイズがなく、次の SQL は動的に作成されます。

  クエリーが 30 個の識別子を下回る場合、このスタイルは要求された識別子の数のみを動的に読み込みます。

- コレクション当たりのレベル
  Hibernate では、上記の各クラス当たりのセクションにリストされているバッチフェッチサイズとスタイルを採用するロードコレクションのバッチ処理も可能です。

前のセクションで使用した例を戻すには、各 **owner** オブジェクトが所有するすべての **car** オブジェクトをロードする必要があることを考慮してください。10 個の **owner** オブジェクトが現行セッションでロードされ、すべての owner で 反復すると、10 個の SELECT ステートメントが生成されます (**getCars()** メソッドへの呼び出しごと)。Owner のマッピングで cars コレクションのバッチフェッチを有効にすると、Hibernate は以下のようにこれらのコレクションの事前フェッチを実行できます。

```
<class name="Owner"><set name="cars" batch-size="5"></set></class>
```

そのため、バッチサイズが 10 個で、レガシーバッチスタイルを使用して つのコレクションをロードすると、Hibernate は つの SELECT ステートメントを実行し、各コレクションを取得します。

6.6.2. 変更不可のデータのオブジェクト参照の 2 次レベルのキャッシング

Hibernate は、パフォーマンスを改善するために自動的にデータをメモリーにキャッシュします。これは、特にほとんど変更されないデータに対して、必要なデータベースルックアップ回数を減らすインメモリーキャッシュによって実現されます。

Hibernate は以下の種類のキャッシュを維持します。1 次キャッシュとも呼ばれるプライマリーキャッシュは必須です。このキャッシュは現行セッションと関連付けられ、すべてのリクエストがそのセッションを通過する必要があります。セカンダリーキャッシュとも呼ばれる 2 次キャッシュは任意で、プライマリーキャッシュが参照された後にのみ参照されます。

データは、最初にデータを状態アレイにアセンブルすることで 2 次キャッシュに保存されます。このアレイはディープコピーされ、そのディープコピーがキャッシュに配置されます。その逆は、キャッシュからの読み取りに対して行われます。これは、変更 (変更不能データ) が変更できないデータには適切に機能しますが、変更できないデータには効率的ではありません。

データのディープコピーは、メモリー使用量と処理速度に関する負荷のかかる操作です。大きなデータセットでは、メモリーと処理速度がパフォーマンス制限の要因となります。Hibernate では、変更不能なデータがコピーされるのではなく、参照されるように指定できます。Hibernate は、データセット全体をコピーする代わりに、キャッシュのデータへの参照を保管できるようになりました。

これには、設定 **hibernate.cache.use_reference_entries** の値を **true** に変更します。デフォルトでは、**hibernate.cache.use_reference_entries** は **false** に設定されています。

hibernate.cache.use_reference_entries が **true** に設定されている場合は、関連付けのない変更不能なデータオブジェクトは 2 次キャッシュにコピーされず、その参照のみが保存されます。

警告

hibernate.cache.use_reference_entries が **true** に設定されると、関連付けのある変更不能なデータオブジェクトが 2 次キャッシュにディープコピーされます。

第7章 HIBERNATE SEARCH

7.1. HIBERNATE SEARCH の使用

7.1.1. Hibernate Search について

Hibernate Search は、Hibernate アプリケーションに完全なテキスト検索機能を提供します。これは、全文、あいまい、位置情報検索などの SQL ベースのソリューションには適していないアプリケーションの検索に適しています。Hibernate Search は Apache Lucene を全文検索エンジンとして使用しますが、メンテナンスのオーバーヘッドを最小限に抑えるように設計されています。設定が完了したら、インデックス、クラスターリング、およびデータ同期は透過的に維持されるため、ビジネス要件を満たすことができます。

注記

以前のリリースの JBoss EAP には Hibernate 4.2 および Hibernate Search 4.6 が含まれていました。JBoss EAP 7 には Hibernate 5 と Hibernate Search 5.5 が含まれています。

Hibernate Search 5.5 は Java 7 と機能し、Lucene 5.3.x 上に構築されるようになりました。ネイティブ Lucene API を使用している場合は、必ずこのバージョンに合わせてください。

7.1.2. Hibernate Search の概要

Hibernate Search はインデックスコンポーネントとインデックス検索コンポーネントで設定されており、どちらも Apache Lucene によってサポートされます。エンティティーがデータベースから挿入、更新、または削除されるたびに、Hibernate Search は Hibernate イベントシステムからこのイベントを追跡し、インデックスの更新をスケジュール設定します。これらの更新はすべて、Apache Lucene API と直接対話せずに処理されます。その代わりに、基盤の Lucene インデックスとの対話は **IndexManager** で処理されます。デフォルトでは、IndexManager と Lucene インデックスの間に 1 対 1 の関係があります。IndexManager は、選択した **back end**、**reader strategy** および **DirectoryProvider** を含む特定のインデックス設定を抽象化します。

インデックスが作成されると、基盤の Lucene インフラストラクチャーを処理するのではなく、エンティティーを検索し、管理エンティティーのリストを返すことができます。同じ永続コンテキストが Hibernate と Hibernate Search 間で共有されます。**FullTextSession** クラスは Hibernate **Session** クラスの上に構築されるため、アプリケーションコードで HQL、Java Persistence クエリー言語 (JPQL)、またはネイティブクエリーと同じように統一された **org.hibernate.Query** または **javax.persistence.Query** API を使用できます。

トランザクションバッチモードは、Java Naming および Directory Interface ベースであるかに関わらず、すべての操作に推奨されます。

注記

Java Naming and Directory Interface または Jakarta Transactions に関わらず、データベースと Hibernate Search の両方に対して、トランザクションで操作を実行することが推奨されます。

注記

Hibernate Search は、アトミック会話として知られる Hibernate または EntityManager の長い会話パターンで完全に機能します。

7.1.3. ディレクトリープロバイダーについて

Hibernate Search インフラストラクチャーの一部である Apache Lucene には、インデックスの保管にディレクトリーという概念があります。Hibernate Search は、**Directory Provider** 経由で Lucene Directory インスタンスの初期化および設定を処理します。

Directory_provider プロパティーは、インデックスを保存するために使用するディレクトリープロバイダーを指定します。デフォルトのファイルシステムディレクトリープロバイダーは **filesystem** で、ローカルファイルシステムを使用してインデックスを格納します。

7.1.4. Worker について

Lucene インデックスの更新は、Hibernate Search **Worker** によって処理されます。これは、コンテキストが終了すると、すべてのエンティティーの変更を受け取り、コンテキストでキューに入れてそれらをキューに適用します。最も一般的なコンテキストはトランザクションですが、エンティティーの変更数または他のアプリケーションイベントの数に依存する可能性があります。

効率性を向上するために、対話はバッチ化され、通常はコンテキストの終了時に適用されます。トランザクション以外では、インデックスの更新操作は実際のデータベース操作の直後に実行されます。継続中のトランザクションの場合、インデックス更新操作はトランザクションのコミットフェーズに対してスケジュール設定され、トランザクションのロールバック時に破棄されます。worker は特定のバッチサイズ制限で設定でき、その後、インデックスはコンテキストに関係なく実行されます。

この手法では、インデックス更新を処理するのに以下のようなメリットがあります。

- パフォーマンス: 操作がバッチで実行されると、Lucene インデックスのパフォーマンスが向上しました。
- ACIDity: 実行されるワークはデータベーストランザクションによって実行されたものと同じスコーピングを持ち、トランザクションがコミットされた場合にのみ実行されます。これは、厳密な意味では ACID ではありませんが、ACID の動作では、ソースからいつでも再構築できるため、完全なテキスト検索インデックスにはあまり役に立ちません。

no scope と transactional という 2 つのバッチモードは、自動コミットとトランザクション動作と同等です。パフォーマンスの観点からは、**transactional** モードが推奨されます。スコープの選択は透過的に行われます。Hibernate Search はトランザクションの存在を検出し、スコーピングを調整します。

7.1.5. バックエンド設定と操作

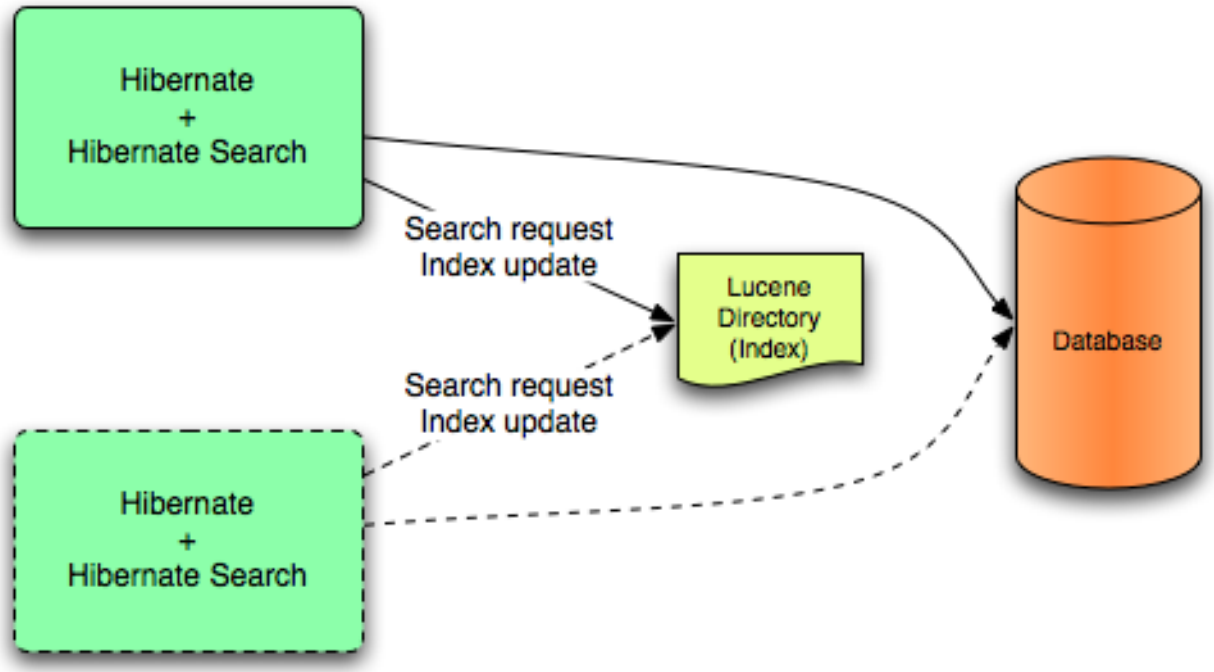
7.1.5.1. バックエンド

Hibernate Search は、さまざまなバックエンドを使用してワークのバッチ処理を行います。バックエンドは、設定オプション **default.worker.backend** に制限されません。このプロパティーは、バックエンド設定の一部である **BackendQueueProcessor** インターフェイスの実装を指定します。Jakarta Messaging バックエンドなどのバックエンドを設定するには、追加の設定が必要です。

7.1.5.2. Lucene

Lucene モードでは、ノードのすべてのインデックス更新は、ディレクトリープロバイダーを使用して同じノードから Lucene ディレクトリーに対して実行されます。このモードは、クラスター化されていない環境や、共有ディレクトリーストアを持つクラスター環境で使用します。

図7.1 Lucene バックエンド設定

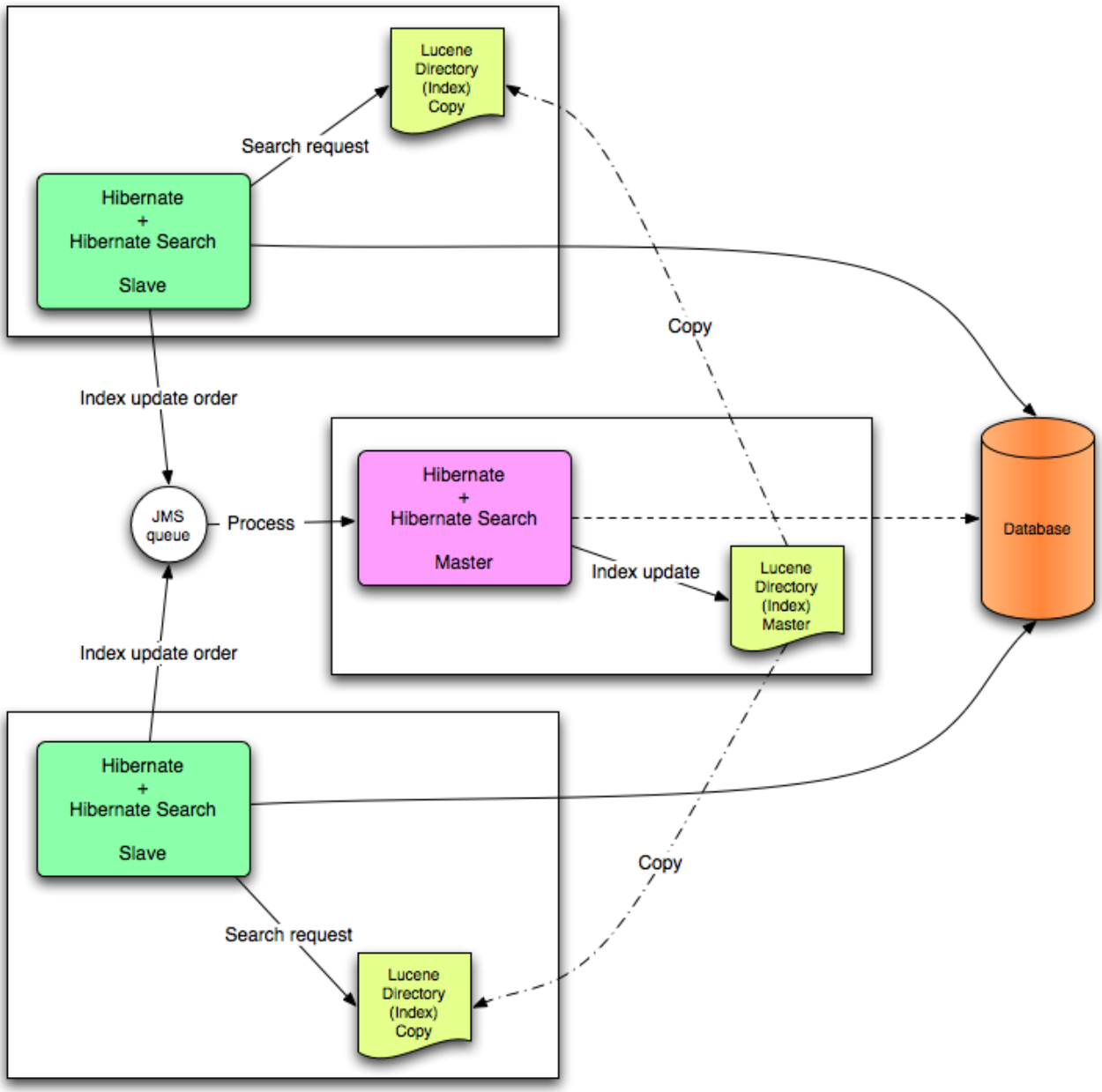


Lucene モードは、ディレクトリーがロックストラテジーを管理するクラスター化またはクラスター化されたアプリケーションをターゲットに設定します。Lucene モードの主な利点は、Lucene クエリーの変更の単純化と即時表示です。Near Real Time (NRT) バックエンドは、クラスター化されていないインデックス設定や共有されていないインデックス設定の代替バックエンドです。

7.1.5.3. Jakarta Messaging

ノードのインデックス更新は Jakarta Messaging キューに送信されます。一意のリーダーはキューを処理し、マスターインデックスを更新します。マスターインデックスはその後、スレーブコピーに定期的に複製され、マスターとスレーブのパターンを確立します。マスターは Lucene インデックスの更新を行います。スレーブは読み取りおよび書き込み操作を受け入れますが、ローカルインデックスコピーの読み取り操作を処理します。マスターは Lucene インデックスの更新のみを行います。更新操作にローカルの変更を適用するのは、マスターのみです。

図7.2 Jakarta Messaging Service バックエンドの設定



このモードは、スループットが重要で、インデックス更新の遅延が許容できるクラスター化された環境をターゲットとします。Jakarta Messaging プロバイダーは信頼性を確保し、スレーブを使用してローカルインデックスコピーを変更します。

7.1.6. リーダーストラテジー

クエリーの実行時に、Hibernate Search はリーダーストラテジーを使用して Apache Lucene インデックスと対話します。頻繁な更新、多くの読み取り、非同期のインデックス更新など、アプリケーションのプロファイルに基づいてリーダーストラテジーを選択します。

7.1.6.1. Shared ストラテジー

shared ストラテジーを使用すると、Hibernate Search は、**IndexReader** が更新され続ける場合に、複数のクエリーおよびスレッドにわたる特定の Lucene インデックスに対して同じ **IndexReader** を共有します。**IndexReader** が更新されない場合は、新しい IndexReader が開かれて提供されます。各

IndexReader は複数の **SegmentReaders** で設定されています。共有ストラテジーは、前回開いた後に修正または作成されたセグメントを分離し、以前のインスタンスのすでにロードされているセグメントを共有します。これはデフォルトのストラテジーです。

7.1.6.2. Not-shared ストラテジー

non-shared ストラテジーを使用すると、クエリーが実行されるたびに **Lucene IndexReader** が開かれます。**IndexReader** の開始と起動は、高価な操作です。そのため、各クエリー実行で **IndexReader** を開くことは効率的なストラテジーではありません。

7.1.6.3. カスタムリーダーストラテジー

org.hibernate.search.reader.ReaderProvider の実装を使用して、カスタムリーダーストラテジーを作成できます。実装はスレッドセーフである必要があります。

7.2. CONFIGURATION (設定)

7.2.1. 最小設定

Hibernate Search は、設定および操作の柔軟性を提供するように設計されており、ほとんどのユースケースに合わせてデフォルト値を慎重に選択しています。少なくとも、Directory Provider とそのプロパティーを設定する必要があります。デフォルトの Directory Provider は **filesystem** で、インデックスストレージにローカルファイルシステムを使用します。利用可能な Directory Providers およびその設定の詳細は、DirectoryProvider Configuration を参照してください。

Hibernate を直接使用する場合は、DirectoryProvider などの設定を、設定ファイル (**hibernate.properties** または **hibernate.cfg.xml**) で設定する必要があります。Jakarta Persistence 経由で Hibernate を使用している場合、設定ファイルは **persistence.xml** になります。

その他のリソース

- 利用可能な Directory Providers およびその設定の詳細は、DirectoryProvider Configuration を参照してください。

7.2.2. IndexManager の設定

Hibernate Search は、このインターフェイスにいくつかの実装を提供します。

- **directory-based**: Lucene **Directory** 抽象化を使用してインデックスファイルを管理するデフォルトの実装。
- **near-real-time**: コミット時にディスクへの書き込みをフラッシュしないようにします。このインデックスマネージャーは **Directory** ベースでもありますが、Lucene のほぼリアルタイムの NRT 機能を使用します。

デフォルト以外の IndexManager を指定するには、以下のプロパティーを指定します。

```
hibernate.search.[default|<indexname>].indexmanager = near-real-time
```

7.2.2.1. Directory-based

Directory-based 実装はデフォルトの **IndexManager** 実装です。これは高度な設定が可能で、リーダーストラテジー、バックエンド、およびディレクトリープロバイダーに個別の設定を可能にします。

7.2.2.2. Near Real Time

NRTIndexManager はデフォルトの **IndexManager** の拡張機能で、低レイテンシーインデックス書き込みに Lucene NRT、Near Real Time 機能を活用します。ただし、**lucene** 以外の代替のバックエンドの設定を無視し、**Directory** で排他的な書き込みロックを取得します。

IndexWriter は、低レイテンシーを提供するため、ディスクへの変更をすべてフラッシュしません。クエリーは、フラッシュされていないインデックスライターバッファーから更新された状態を読み取ることができます。ただし、これは、**IndexWriter** が強制終了したり、アプリケーションがクラッシュした場合に、インデックスを再構築する必要があるように更新が失われる可能性があることを意味します。

Near Real Time 設定は、上記のデメリットとマスターノードを個別に設定できるため、データが限定されたクラスター化されていない Web サイトに対して推奨されます。

7.2.2.3. Custom

カスタム実装の完全修飾クラス名を指定して、カスタマイズされた **IndexManager** を設定します。以下のように、実装に no-argument コンストラクターを設定します。

```
[default|<indexname>].indexmanager = my.corp.myapp.CustomIndexManager
```

カスタムインデックスマネージャーの実装には、デフォルトの実装と同じコンポーネントは必要ありません。たとえば、**Directory** インターフェイスを公開しないリモートインデックスサービスに委譲します 。

7.2.3. DirectoryProvider の設定

DirectoryProvider は Lucene **Directory** に関する Hibernate Search の抽象化で、基盤の Lucene リソースの設定および初期化を処理します。Directory Providers and Their Properties では Hibernate Search で利用可能なディレクトリープロバイダーの一覧と、対応するオプションを表示しています。

インデックス化された各エンティティーは Lucene インデックスに関連付けられます (複数のエンティティーが同じインデックスを共有している場合を除きます)。インデックスの名前は、**@Indexed** アノテーションの **index** プロパティーによって指定されます。**Index** プロパティーが指定されていない場合は、インデックスされたクラスの完全修飾名が名前として使用されます (推奨)。

DirectoryProvider および追加オプションは、接頭辞 **hibernate.search.<indexname>** を使用して設定できます。名前 **default** (**hibernate.search.default**) は予約されており、すべてのインデックスに適用されるプロパティーを定義するために使用できます。Configuring Directory Providers では、**hibernate.search.default.directory_provider** を使用してデフォルトのディレクトリープロバイダーをファイルシステムに設定する方法を示しています。その後に、**hibernate.search.default.indexBase** がインデックスのデフォルトのベースディレクトリーを設定します。その結果、エンティティー **Status** のインデックスが **/usr/lucene/indexes/org.hibernate.example.Status** に作成されます。

ただし、**Rule** エンティティーのインデックスは、メモリー内ディレクトリーを使用します。これは、このエンティティーのデフォルトのディレクトリープロバイダーが **hibernate.search.Rules.directory_provider** プロパティーによって上書きされるためです。

最後に、**Action** エンティティーは 、 **hibernate.search.Actions.directory_provider** で指定されるカスタムディレクトリープロバイダー **CustomDirectoryProvider** を使用します。

インデックス名の指定

```
package org.hibernate.example;
```

```
@Indexed
public class Status { ... }

@Indexed(index="Rules")
public class Rule { ... }

@Indexed(index="Actions")
public class Action { ... }
```

ディレクトリープロバイダーの設定

```
hibernate.search.default.directory_provider = filesystem
hibernate.search.default.indexBase=/usr/lucene/indexes
hibernate.search.Rules.directory_provider = ram
hibernate.search.Actions.directory_provider = com.acme.hibernate.CustomDirectoryProvider
```

注記

上記の設定スキームを使用すると、ディレクトリープロバイダーやベースディレクトリーなどの一般的なルールを簡単に定義でき、これらのデフォルトをインデックスごとに後で上書きできます。

ディレクトリープロバイダーおよびそれらのプロパティー

ram

None

Filesystem

ファイルシステムベースのディレクトリー使用されるディレクトリーは <indexBase> /< indexName > です。

- **indexBase**: ベースディレクトリー
- **indexName**: @Indexed.index を上書き (シャード化されたインデックスに便利です)
- **locking_strategy**: オプション。LockFactory Configuration を参照してください。
- **filesystem_access_type**: この **DirectoryProvider** で使用される **FSDirectory** 実装の正確なタイプを判別できます。許可される値は **auto** (デフォルト値。Windows 以外のシステムでは **NIOFSDirectory**、Windwos では **SimpleFSDirectory**)、**simple (SimpleFSDirectory)**、**nio (NIOFSDirectory)**、**mmap (MMapDirectory)** を選択します。この設定を変更する前に、これらのディレクトリー実装に関する Java ドキュメントを参照してください。**NIOFSDirectory** または **MMapDirectory** は、パフォーマンスを大幅に向上させる可能性がありますが、問題もあります。

filesystem-master

ファイルシステムベースのディレクトリー**filesystem** と類似しています。また、定期的にインデックスをソースディレクトリー (コピーディレクトリー) にコピーします。
更新期間に推奨される値は (最低 50%)、情報をコピーする時間 (デフォルトは 3600 秒 - 60 分) です。

コピーは、増分コピーメカニズムをベースにしているため、コピーの平均時間が短縮されることに注意してください。

DirectoryProvider は通常、Jakarta Messaging バックエンドクラスターのマスターノードで使用されます。

buffer_size_on_copy 最適化は、ご使用のオペレーティングシステムと利用可能な RAM によって異なります。多くユーザーは 16 から 64MB までの値を使用して良好な結果を報告しています。

- **indexBase**: ベースディレクトリー
- **indexName**: @Indexed.index を上書き (シャード化されたインデックスに便利です)
- **sourceBase**: ソース (コピー) のベースディレクトリー。
- **source**: ソースディレクトリーの接尾辞 (デフォルトは **@Indexed.index)**。実際のソースのディレクトリー名は **<sourceBase>/<source>** です。
- **refresh**: 更新の間隔 (秒単位) です (コピーは更新の秒数ごとに行われます)。次の **refresh** 期間が経過してもコピーが進行中であると、次のコピー操作はスキップされます。
- **buffer_size_on_copy**: 単一の低レベルのコピー命令で移動するメガージの量。デフォルトは 16MB です。
- **locking_strategy**: オプション。LockFactory Configuration を参照してください。
- **filesystem_access_type**: この **DirectoryProvider** で使用される **FSDirectory** 実装の正確なタイプを判別できます。許可される値は **auto** (デフォルト値。Windows 以外のシステムでは **NIOFSDirectory**、Windwos では **SimpleFSDirectory**)、**simple (SimpleFSDirectory)**、**nio (NIOFSDirectory)**、**mmap (MMapDirectory)** を選択します。この設定を変更する前に、これらのディレクトリー実装に関する Java ドキュメントを参照してください。**NIOFSDirectory** または **MMapDirectory** ではパフォーマンスが大幅に向上しますが、注意が必要な問題もあります。

filesystem-slave

ファイルシステムベースのディレクトリー**filesystem** と似ていますが、マスターバージョン (ソース) を定期的に取得します。ロックおよび一貫性のない検索結果を避けるため、2 つのローカルコピーが維持されます。
更新期間に推奨される値は (最低 50%)、情報をコピーする時間 (デフォルトは 3600 秒 - 60 分) です。

コピーは、増分コピーメカニズムをベースにしているため、コピーの平均時間が短縮されることに注意してください。**refresh** 期間が経過してもコピーが進行中であると、次のコピー操作はスキップされます。

DirectoryProvider は通常、Jakarta Messaging バックエンドを使用するスレーブノードに使用されます。

buffer_size_on_copy 最適化は、ご使用のオペレーティングシステムと利用可能な RAM によって異なります。多くユーザーは 16 から 64MB までの値を使用して良好な結果を報告しています。

- **indexBase**: ベースディレクトリー
- **indexName**: @Indexed.index を上書き (シャード化されたインデックスに便利です)
- **sourceBase**: ソース (コピー) のベースディレクトリー。
- **source**: ソースディレクトリーの接尾辞 (デフォルトは **@Indexed.index)**。実際のソースのディレクトリー名は **<sourceBase>/<source>** です。

- **refresh**: 更新の間隔 (秒単位) です (コピーは更新の秒数ごとに行われます)。
- **buffer_size_on_copy**: 単一の低レベルのコピー命令で移動するメガージの量。デフォルトは 16MB です。
- **locking_strategy**: オプション。LockFactory Configuration を参照してください。
- **retry_marker_lookup**: オプションで、デフォルトは 0 です。Hibernate Search がソースディレクトリーのマーカーファイルをチェックする回数を定義します。試行ごとに 5 秒待機します。
- **retry_initialize_period**: オプション。再試行初期化機能を有効にするために整数値を秒単位で設定します。スレーブがマスターインデックスを検出できない場合、アプリケーションがバックグラウンドで起動しないようにせずに再試行します。インデックスの初期化前に実行された FULLTEXT クエリーはブロックされず、空の結果が返されます。オプションを有効にしない、または明示的にゼロに設定すると、再試行タイマーのスケジュール設定ではなく、例外により失敗します。無効なインデックスなしでアプリケーションが起動しないようにし、初期化のタイムアウトを制御するには、代わりに **retry_marker_lookup** を参照してください。
- **filesystem_access_type**: この **DirectoryProvider** で使用される **FSDirectory** 実装の正確なタイプを判別できます。許可される値は auto (デフォルト値。Windows 以外のシステムでは **NIOFSDirectory**、Windwos では **SimpleFSDirectory**)、**simple (SimpleFSDirectory)**、**nio (NIOFSDirectory)**、**mmap (MMapDirectory)** を選択します。この設定を変更する前に、これらのディレクトリー実装に関する Java ドキュメントを参照してください。**NIOFSDirectory** または **MMapDirectory** はパフォーマンスを大幅に向上させる可能性がありますが、問題にも認識する必要があります。

注記

ビルトインディレクトリープロバイダーがニーズに適さない場合は、**org.hibernate.store.DirectoryProvider** インターフェイスを実装して独自のディレクトリープロバイダーを作成することができます。この場合、プロバイダーの完全修飾クラス名を **directory_provider** プロパティーに渡します。接頭辞 **hibernate.search.<indexname>** を使用して追加のプロパティーを渡すことができます。

7.2.4. Worker 設定

workder 設定では、Hibernate Search が Lucene と対話する方法を詳細化することができます。複数のアーキテクチャーコンポーネントと可能な拡張ポイントが存在します。詳しく見てみましょう。

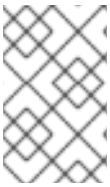
ワーカー設定を使用して、Infinispan クエリーが Lucene と対話する方法を詳細化します。この設定には、いくつかのアーキテクチャーコンポーネントおよび可能な拡張ポイントを使用できます。

まず、**Worker** があります。**Worker** インターフェイスの実装は、すべてのエンティティーの変更を受け取り、コンテキストによってそれらをキューに登録し、コンテキストの終了時に適用します。特に ORM との接続で最も直感的なコンテキストはトランザクションです。このため、はデフォルトで **TransactionalWorker** を使用してトランザクションごとにすべての変更をスコープします。ただし、エンティティーの変更数やその他のアプリケーションライフサイクルイベントなどによってコンテキストが異なるシナリオを想定できます。

表7.1 スコープの設定

プロパティー	説明
hibernate.search.worker.scope	使用する **Worker** 実装の完全修飾クラス名。このプロパティーが設定されていない場合や、空または **transaction** の場合は、デフォルトの **TransactionalWorker** が使用されます。
hibernate.search.worker.*	接頭辞 **hibernate.search.worker** が付いたすべての設定プロパティーは初期化中にワーカーに渡されます。これにより、カスタムのワーカー固有のパラメーターを追加できます。
hibernate.search.worker.batch_size	コンテキストごとにバッチ処理されるインデックス操作の最大数を定義します。制限に達すると、コンテキストが終了していなくてもインデックスがトリガーされます。このプロパティーは、**Worker** 実装がキュー待ちの作業を BatchedQueueingProcessor (**TransactionalWorker** の動作) に委任する場合にのみ動作します。

コンテキストが終了すると、インデックスの変更を準備して適用します。これは、新規スレッド内で同期または非同期に実行できます。同期更新には、常にインデックスがデータベースと同期しているという利点があります。一方、非同期の更新は、ユーザーの応答時間を最小限に抑えるのに役立ちます。欠点は、データベースとインデックスの状態間で不一致が生じる可能性があることです。



注記

以下のオプションはインデックスごとに異なる場合があります。実際には、indexName 接頭辞が必要になるか、**default** を使用してすべてのインデックスのデフォルト値を設定する必要があります。

表7.2 実行設定

プロパティー	説明
hibernate.search.<indexName>.worker.execution	**sync**: 同期実行 (デフォルト) **async**: 非同期実行
hibernate.search.<indexName>.worker.thread_pool.size	バックエンドは、スレッドプールを使用して、同じトランザクションコンテキスト (またはバッチ) から更新を並行して適用することができます。デフォルト値は 1 です。トランザクションごとに多数の操作がある場合は、大きな値を試すことができます。
hibernate.search.<indexName>.worker.buffer_queue.max	スレッドプールが不足している場合は、ワークキューの最大数を定義します。非同期実行のみに便利です。デフォルトは infinite です。制限に達すると、ワークはメインスレッドによって行われます。

現在、いずれの実行モードであっても、すべての作業は同じ仮想マシン内で行われます。単一仮想マシ

ンの作業合計量は変更されていません。常に、より適切なアプローチ (つまり委任) があります。**hibernate.search.default.worker.backend** を設定すると、インデックス作業を別のサーバーに送信できます。このオプションも、インデックスごとに異なる方法で設定できます。

表7.3 バックエンドの設定

プロパティー	説明
hibernate.search.<indexName>.worker.backend	**lucene**: 同じ仮想マシンでインデックスを更新するデフォルトのバックエンド。プロパティーが定義されていないか、空の場合に使用されます。 **jms**: Java Messaging Service バックエンド。インデックスの更新は Java Messaging Service キューに送信され、インデックスマスターによって処理されます。その他の設定オプションや、この設定の詳細は、 Java Messaging Service Back-end Configuration を参照してください。 **BlackHole**: 主にテスト/開発者の設定で、すべてのインデックス作業を無視します。 **BackendQueueProcessor** を実装するクラスの完全修飾名を指定することもできます。これにより、独自の通信層を実装することができます。この実装は、実行時にインデックスワークを処理する **Runnable** インスタンスを返します。

1. Java Messaging サービスのバックエンド設定

プロパティー	説明
hibernate.search.<indexName>.worker.jndi.*	必要に応じて InitialContext を開始する Java Naming および Directory Interface プロパティーを定義します。Java Naming and Directory Interface は、Java Messaging サービスのバックエンドによってのみ使用されます。
hibernate.search.<indexName>.worker.jms.connection_factory	Java Messaging Service バックエンドには必須です。Java Messaging Service 接続ファクトリーを (Red Hat JBoss Enterprise Application Platform のデフォルトでは /**ConnectionFactory**) から検索する Java Naming および Directory Interface 名を定義します。
hibernate.search.<indexName>.worker.jms.queue	Java Messaging Service バックエンドには必須です。Java Messaging Service キューを検索する Java Naming and Directory Interface 名を定義します。キューはワークメッセージをポストするために使用されます。

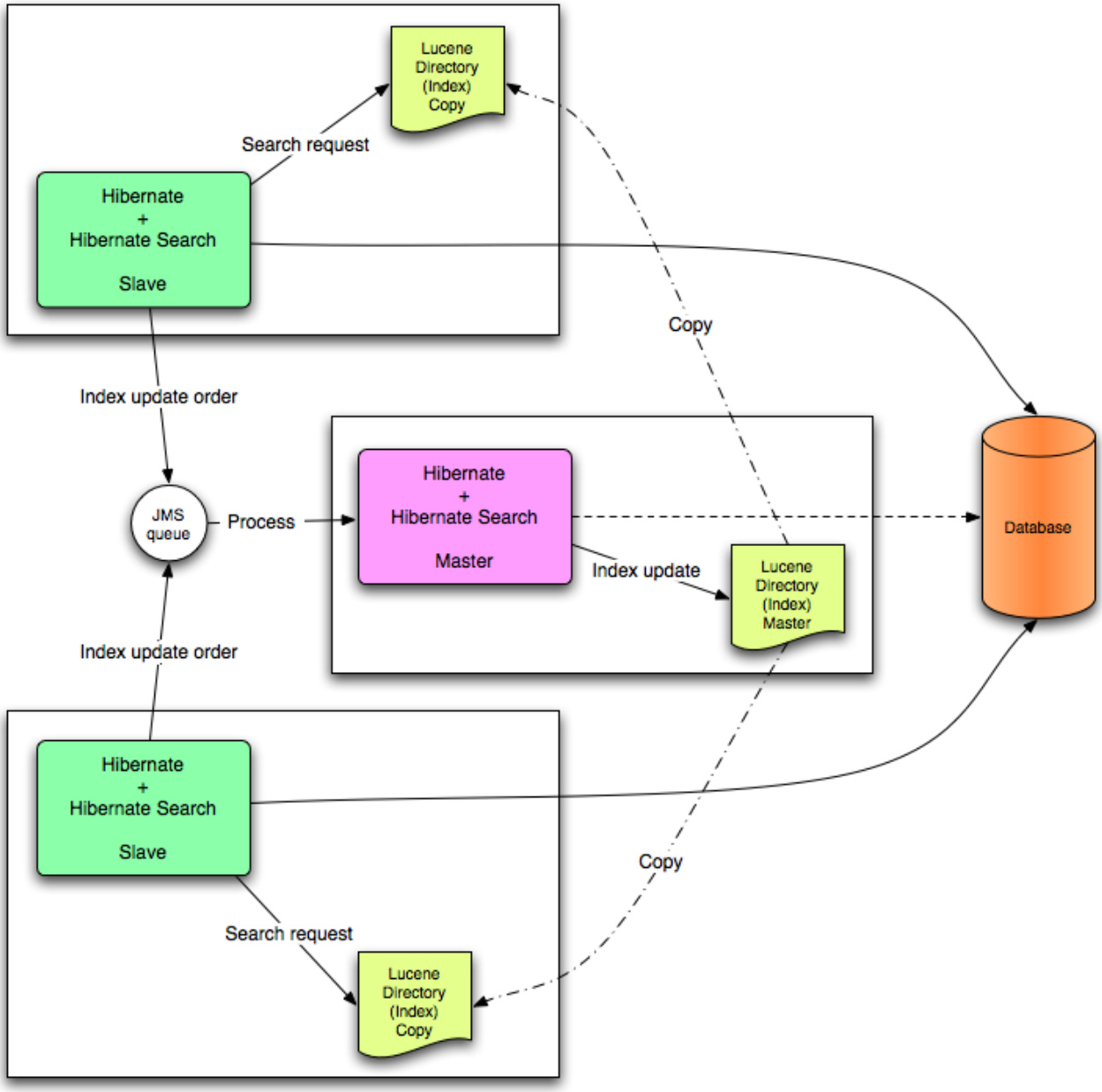
警告

おそらく、表示されるプロパティーの一部は関連付けられるため、プロパティー値のすべての組み合わせが適切であるとは限りません。実際には、機能以外の設定を行うことができます。これは、特に、ここに示されるインターフェイスの独自の実装を提供する場合が該当します。独自の **Worker** または **BackendQueueProcessor** 実装を作成する前に、既存のコードを調査してください。

7.2.4.1. Jakarta Messaging マスター/スレーブバックエンド

このセクションでは、マスター/スレーブの Hibernate Search アーキテクチャーの設定方法について説明します。

図7.3 Jakarta Messaging バックエンドの設定



7.2.4.2. スレーブノード

すべてのインデックス更新操作は Jakarta Messaging キューに送信されます。インデックスクエリー操作は、ローカルインデックスコピーで実行されます。

Jakarta Messaging スレーブの設定

```
### slave configuration

## DirectoryProvider
# (remote) master location
hibernate.search.default.sourceBase = /mnt/mastervolume/lucenedirs/mastercopy

# local copy location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider = filesystem-slave

## Back-end configuration
hibernate.search.default.worker.backend = jms
hibernate.search.default.worker.jms.connection_factory = /ConnectionFactory
hibernate.search.default.worker.jms.queue = queue/hibernatesearch
#optional jndi configuration (check your Jakarta Messaging provider for more information)

## Optional asynchronous execution strategy
# hibernate.search.default.worker.execution = async
# hibernate.search.default.worker.thread_pool.size = 2
# hibernate.search.default.worker.buffer_queue.max = 50
```

注記

ファイルシステムローカルコピーは、より高速な検索結果を得るために推奨されます。

7.2.4.3. マスターノード

すべてのインデックス更新操作は Jakarta Messaging キューから取得され、実行されます。マスターインデックスは定期的にコピーされます。

Jakarta Messaging キューのインデックス更新操作は実行され、マスターインデックスは定期的にコピーされます。

Jakarta Messaging Service Master 設定

```
### master configuration

## DirectoryProvider
# (remote) master location where information is copied to
hibernate.search.default.sourceBase = /mnt/mastervolume/lucenedirs/mastercopy

# local master location
hibernate.search.default.indexBase = /Users/prod/lucenedirs
```

```
# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider = filesystem-master

## Back-end configuration
#Back-end is the default for Lucene
```

Hibernate Search フレームワークの設定に加えて、Jakarta Messaging を介してインデックスを処理するようメッセージ駆動 Bean を作成し、設定する必要があります。

メッセージ駆動型 Bean がインデックスキューを処理する

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationType",
                              propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination",
                              propertyValue="queue/hibernatesearch"),
    @ActivationConfigProperty(propertyName="DLQMaxResent", propertyValue="1")
  } )
public class MDBSearchController extends AbstractJMSHibernateSearchController
                               implements MessageListener {
   @PersistenceContext EntityManager em;

   //method retrieving the appropriate session
   protected Session getSession() {
      return (Session) em.getDelegate();
   }

   //potentially close the session opened in #getSession(), not needed here
   protected void cleanSessionIfNeeded(Session session)
   }
}
```

この例では、Hibernate Search ソースコードで利用可能な抽象 Jakarta Messaging コントローラークラスから継承し、Jakarta EE MDB を実装します。この実装は例として提供されており、Jakarta EE 以外のメッセージ駆動 Bean を利用するように調整できます。

7.2.5. Lucene インデックスのチューニング

7.2.5.1. Lucene インデックスのパフォーマンスチューニング

Hibernate Search は、**mergeFactor**、**maxMergeDocs**、**maxBufferedDocs** など、基礎となる Lucene **IndexWriter** に渡される一連のパラメーターを指定することで Lucence インデクスパフォーマンスを調整するのに使用されます。これらのパラメーターは、インデックスベースまたはシャードごとに、すべてのインデックスに適用されるデフォルト値として指定します。

各種ユースケースに合わせて調整できる低レベルの **IndexWriter** 設定がいくつかあります。これらのパラメーターは、**indexwriter** キーワードでグループ化されます。

```
hibernate.search.[default|<indexname>].indexwriter.<parameter_name>
```

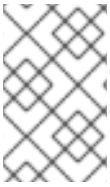
特定のシャード設定の **indexwriter** 値に値が設定されていないと、Hibernate Search は index セクションをチェックしてから default セクションをチェックします。

以下の表の設定により、**Animal** インデックスの 2 つ目のシャードにこれらの設定が適用されます。

- **max_merge_docs** = 10
- **merge_factor** = 20
- **ram_buffer_size** = 64MB
- **term_index_interval** = Lucene default

他のすべての値は、Lucene で定義されたデフォルトを使用します。

デフォルトでは、すべての値は Lucene の独自のデフォルトのままにします。Indexing Performance and Behavior Properties に一覧表示される値は、使用している Lucene のバージョンに応じて異なります。上記の値はバージョン **2.4** に相対的です。



注記

Hibernate Search の以前のバージョンには **batch** および **transaction** プロパティーの概念がありました。バックエンドは常に同じ設定を使用して機能するため、これは変わりました。

表7.4 パフォーマンスおよび動作プロパティーのインデックス作成

プロパティー	説明	デフォルト値
hibernate.search.[default|<indexname>].exclusive_index_use	他のプロセスが同じインデックスに書き込む必要がない場合は **true** に設定します。これにより、Hibernate Search がインデックスの排他モードで動作し、インデックスへの変更を書き込む際にパフォーマンスが向上します。	**true** (パフォーマンスの向上、シャットダウン時のみロックされる)
hibernate.search.[default|<indexname>].max_queue_length	各インデックスには、インデックスに適用される更新が含まれる個別の pipeline があります。このキューが満杯になると、ブロック操作になります。**worker.execution** が **async** として設定されていない限り、この設定の設定は意味がありません。	**1000**
hibernate.search.[default|<indexname>].indexwriter.max_buffered_delete_terms	バッファーインメモリー削除の条件が適用され、フラッシュされるまでに最低限必要な削除用語を決定します。時点でバッファーされたドキュメントがメモリーにある場合、ドキュメントはマージされ、新しいセグメントが作成されます。	Disabled (RAM 使用率によるフラッシュ)
hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs	インデックス作成中にメモリーでバッファーされたドキュメントの量を制御します。RAM が大きいほど消費されます。	Disabled (RAM 使用率によるフラッシュ)

プロパティー	説明	デフォルト値
hibernate.search.[default|<indexname>].indexwriter.max_merge_docs	セグメント内で許容されるドキュメントの最大数を定義します。値を小さくすると、頻繁に変更されるインデックスでのパフォーマンスが向上します。値を大きくすると、インデックスが頻繁に変更されない場合に検索パフォーマンスが向上します。	Unlimited (Integer.MAX_VALUE)
hibernate.search.[default|<indexname>].indexwriter.merge_factor	セグメントマージの頻度とサイズを制御します。 挿入時にセグメントインデックスをマージする頻度を決定します。値を小さくすると、インデックス処理時に使用される RAM が少なくなり、最適化されていないインデックスの検索速度が速くなりますが、インデックスの速度は遅くなります。値が大きいと、インデックス時により多くの RAM が使用され、最適化されていないインデックスの検索速度が遅くなるため、インデックスがより速くなります。そのため、大きな値 (> 10) はバッチインデックスの作成に最も適しており、対話的に維持されるインデックスに小さい値 (< 10) が適しています。この値は 2 未満にしないでください。	10
hibernate.search.[default|<indexname>].indexwriter.merge_min_size	セグメントマージの頻度とサイズを制御します。このサイズより小さいセグメント (MB 単位) は常に次のセグメントマージ操作の対象となります。このパラメーターを高く設定しすぎると、マージ操作が頻繁に行われない場合でも、処理にコストがかかる可能性があります。**org.apache.lucene.index.LogDocMergePolicy.minMergeSize** も参照してください。	0 MB (実際 ~1K)
hibernate.search.[default|<indexname>].indexwriter.merge_max_size	セグメントマージの頻度とサイズを制御します。 このサイズより大きいセグメント (MB 単位) は、大きなセグメントにマージされることはありません。 これにより、メモリー要件が減り、一部のマージ操作が最適な検索速度で回避されます。インデックスの最適化時にこの値は無視されます。 **org.apache.lucene.index.LogDocMergePolicy.maxMergeSize** も参照してください。	無制限
hibernate.search.[default|<indexname>].indexwriter.merge_max_optimize_size	セグメントマージの頻度とサイズを制御します。 このサイズよりも大きい (MB 単位) セグメントは、インデックスの最適化中にも大きなセグメントではマージされません (**merge_max_size** 設定も参照)。 **org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptimize** に適用されます。	無制限

プロパティー	説明	デフォルト値
hibernate.search.[default|<indexname>].indexwriter.merge_calibrate_by_deletes	セグメントマージの頻度とサイズを制御します。 **false** に設定すると、マージポリシーを見積もるときに削除されたドキュメントを考慮しません。 **org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes** に適用されます。	**true**
hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size	ドキュメントバッファー専用の RAM の容量 (MB 単位) を制御します。Max_buffered_docs を一緒に使用すると、いずれのイベントに対してもフラッシュが発生します。 通常、インデックスのパフォーマンスを上げるには、ドキュメント数ではなく、RAM 使用率によってフラッシュし、RAM バッファーをできるだけ大きく使用するのが最良の方法です。	16 MB
hibernate.search.[default|<indexname>].indexwriter.term_index_interval	インデックス化された期間の間隔を設定します。 値が大きくなると、IndexReader では使用されるメモリーが少なくなります。しかし、ランダムアクセスが遅くなります。値が小さいと、IndexReader の方がより多くのメモリーが使用され、用語へのランダムアクセスが速くなります。詳細は、Lucene ドキュメントを参照してください。	128
hibernate.search.[default|<indexname>].indexwriter.use_compound_file	複合ファイル形式を使用すると、使用するファイル記述子が少なくなります。欠点は、インデックス作成にかかる時間と一時ディスク容量が多いことです。インデックス処理時間を短縮するために、このパラメーターを **false** に設定できますが、**mergeFactor** が大きくなるとファイル記述子が不足する可能性があります。 ブール値のパラメーター。**true** または **false** を使用します。このオプションのデフォルト値は **true** です。	true

プロパティー	説明	デフォルト値
hibernate.search.enable_dirty_check	すべてのエンティティーの変更に Lucene インデックスの更新が必要であるわけではありません。更新されたエンティティープロパティー (dirty プロパティー) すべてがインデックス化されない場合、Hibernate Search はインデックス変更プロセスを省略します。 各更新イベントで呼び出す必要があるカスタムの **FieldBridges** を使用する場合は、このオプションを無効にします (フィールドブリッジを設定するプロパティーに変更がありません)。 この最適化は、**@ClassBridge** または **@DynamicBoost** を使用するクラスには適用されません。 ブール値のパラメーター。**true** または **false** を使用します。このオプションのデフォルト値は **true** です。	true

警告

blackhole バックエンドは 、インデックスのボトルネックを特定するツールとしてのみ、実稼働環境で使用することは意図されていません。

7.2.5.2. Lucene IndexWriter

各種ユースケースに合わせて調整できる低レベルの **IndexWriter** 設定がいくつかあります。これらのパラメーターは、**indexwriter** キーワードでグループ化されます。

```
default.<indexname>.indexwriter.<parameter_name>
```

シャード設定で **indexwriter** の値が設定されていない場合、Hibernate Search は index セクションを参照し、デフォルトのセクションを探します。

7.2.5.3. パフォーマンスオプションの設定

以下の設定では、これらの設定が **Animal** インデックスの 2 つ目のシャードに適用されます。

パフォーマンスオプションの設定例

```
default.Animals.2.indexwriter.max_merge_docs = 10
default.Animals.2.indexwriter.merge_factor = 20
default.Animals.2.indexwriter.term_index_interval = default
default.indexwriter.max_merge_docs = 100
default.indexwriter.ram_buffer_size = 64
```

- **max_merge_docs** = 10
- **merge_factor** = 20
- **ram_buffer_size** = 64MB
- **term_index_interval** = Lucene default

他のすべての値は、Lucene で定義されたデフォルトを使用します。

Lucene のデフォルト値が Hibernate Search のデフォルト設定です。したがって、以下の表に記載する値は、使用される Lucene のバージョンによって異なります。上記の値はバージョン **2.4** に相対的です。Lucene インデックスのパフォーマンスに関する詳細は、Lucene のドキュメントを参照してください。

注記

バックエンドは常に同じ設定を使用して動作します。

表7.5 パフォーマンスおよび動作プロパティーのインデックス作成

プロパティー	説明	デフォルト値
default.<indexname>.exclusive_index_use	他のプロセスが同じインデックスに書き込む必要がない場合は **true** に設定します。これにより、Hibernate Search がインデックスの排他モードで動作し、インデックスへの変更を書き込む際にパフォーマンスが向上します。	**true** (パフォーマンスの向上、シャットダウン時のみロックされる)
default.<indexname>.max_queue_length	各インデックスには、インデックスに適用される更新が含まれる個別の pipeline があります。このキューが満杯になると、ブロック操作になります。**worker.execution** が **async** として設定されていない限り、この設定の設定は意味がありません。	**1000**
default.<indexname>.indexwriter.max_buffered_delete_terms	バッファーインメモリー削除の条件が適用され、フラッシュされるまでに最低限必要な削除用語を決定します。時点でバッファーされたドキュメントがメモリーにある場合、ドキュメントはマージされ、新しいセグメントが作成されます。	Disabled (RAM 使用率によるフラッシュ)
default.<indexname>.indexwriter.max_buffered_docs	インデックス作成中にメモリーでバッファーされたドキュメントの量を制御します。RAM が大きいほど消費されます。	Disabled (RAM 使用率によるフラッシュ)
default.<indexname>.indexwriter.max_merge_docs	セグメント内で許容されるドキュメントの最大数を定義します。値を小さくすると、頻繁に変更されるインデックスでのパフォーマンスが向上します。値を大きくすると、インデックスが頻繁に変更されない場合に検索パフォーマンスが向上します。	Unlimited (Integer.MAX_VALUE)

プロパティー	説明	デフォルト値
default.<indexname>.indexwriter.merge_factor	セグメントマージの頻度とサイズを制御します。 挿入時にセグメントインデックスをマージする頻度を決定します。値を小さくすると、インデックス処理時に使用される RAM が少なくなり、最適化されていないインデックスの検索速度が速くなりますが、インデックスの速度は遅くなります。値が大きいと、インデックス時により多くの RAM が使用され、最適化されていないインデックスの検索速度が遅くなるため、インデックスがより速くなります。そのため、大きな値 (> 10) はバッチインデックスの作成に最も適しており、対話的に維持されるインデックスに小さい値 (< 10) が適しています。この値は 2 未満にしないでください。	10
default.<indexname>.indexwriter.merge_min_size	セグメントマージの頻度とサイズを制御します。 このサイズより小さいセグメント (MB 単位) は常に次のセグメントマージ操作の対象となります。 このパラメーターを高く設定しすぎると、マージ操作が頻繁に行われない場合でも、処理にコストがかかる可能性があります。 **org.apache.lucene.index.LogDocMergePolicy.minMergeSize** も参照してください。	0 MB (実際 ~1K)
default.<indexname>.indexwriter.merge_max_size	セグメントマージの頻度とサイズを制御します。 このサイズより大きいセグメント (MB 単位) は、大きなセグメントにマージされることはありません。 これにより、メモリー要件が減り、一部のマージ操作が最適な検索速度で回避されます。インデックスの最適化時にこの値は無視されます。 **org.apache.lucene.index.LogDocMergePolicy.maxMergeSize** も参照してください。	無制限
default.<indexname>.indexwriter.merge_max_optimize_size	セグメントマージの頻度とサイズを制御します。 このサイズよりも大きい (MB 単位) セグメントは、インデックスの最適化中にも大きなセグメントではマージされません (**merge_max_size** 設定も参照)。 **org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptimize** に適用されます。	無制限

プロパティー	説明	デフォルト値
default.<indexname>.indexwriter.merge_calibrate_by_deletes	セグメントマージの頻度とサイズを制御します。 **false** に設定すると、マージポリシーを見積もるときに削除されたドキュメントを考慮しません。 **org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes** に適用されます。	**true**
default.<indexname>.indexwriter.ram_buffer_size	ドキュメントバッファー専用の RAM の容量 (MB 単位) を制御します。Max_buffered_docs を一緒に使用すると、いずれのイベントに対してもフラッシュが発生します。 通常、インデックスのパフォーマンスを上げるには、ドキュメント数ではなく、RAM 使用率によってフラッシュし、RAM バッファーをできるだけ大きく使用するのが最良の方法です。	16 MB
default.<indexname>.indexwriter.term_index_interval	インデックス化された期間の間隔を設定します。 大きな値を設定すると、IndexReader が使用するメモリーは少なくなりますが、ランダムアクセスの速度が遅くなります。値を小さくすると、IndexReader によりより多くのメモリーが使用され、用語へのランダムアクセスが速くなります。詳細は、Lucene ドキュメントを参照してください。	128
default.<indexname>.indexwriter.use_compound_file	複合ファイル形式を使用すると、使用するファイル記述子が少なくなります。欠点は、インデックス作成にかかる時間と一時ディスク容量が多いことです。インデックス処理時間を短縮するために、このパラメーターを **false** に設定できますが、**mergeFactor** が大きくなるとファイル記述子が不足する可能性があります。 ブール値のパラメーター。**true** または **false** を使用します。このオプションのデフォルト値は **true** です。	true

プロパティー	説明	デフォルト値
default.enable_dirty_check	すべてのエンティティーの変更に Lucene インデックスの更新が必要であるわけではありません。更新されたエンティティープロパティー (dirty プロパティー) すべてがインデックス化されない場合、Hibernate Search はインデックス変更プロセスを省略します。 各更新イベントで呼び出す必要があるカスタムの **FieldBridges** を使用する場合は、このオプションを無効にします (フィールドブリッジを設定するプロパティーに変更がありません)。 この最適化は、**@ClassBridge** または **@DynamicBoost** を使用するクラスには適用されません。 ブール値のパラメーター。**true** または **false** を使用します。このオプションのデフォルト値は **true** です。	true

7.2.5.4. インデックス速度の調整

アーキテクチャーが許可する場合、インデックスの書き込み効率を向上させるために **default.exclusive_index_use=true** のままにします。

インデックスの速度を調整する場合は、最初にオブジェクトの読み込みの最適化に焦点を合わせ、次にインデックスプロセスのチューニングの基準として達成するタイミングを使用することが推奨されます。**blackhole** をワーカーバックエンドとして設定し、インデックスルーチンを開始します。このバックエンドは、Hibernate Search を無効にしません。インデックスに必要な変更セットが生成されますが、それらをインデックスにフラッシュするのではなく破棄します。**hibernate.search.indexing_strategy** を **manual** に設定するのとは対照的に、**blackhole** を使用すると、関連するエンティティーも再インデックス化されるため、データベースがより多くのデータを読み込む可能性があります。

```
hibernate.search.[default|<indexname>].worker.backend blackhole
```

警告

blackhole バックエンドは 、インデックスのボトルネックを特定する診断ツールとしてのみ、実稼働環境で使用することは意図されていません。

7.2.5.5. コントロールセグメントサイズ

以下のオプションは、作成されるセグメントの最大サイズを設定します。

- **merge_max_size**

- **merge_max_optimize_size**
- **merge_calibrate_by_deletes**

コントロールセグメントサイズ

```
//to be fairly confident no files grow above 15MB, use:
hibernate.search.default.indexwriter.ram_buffer_size = 10
hibernate.search.default.indexwriter.merge_max_optimize_size = 7
hibernate.search.default.indexwriter.merge_max_size = 7
```

マージセグメントが 2 つの大きなセグメントに結合するため、マージ操作の **max_size** はハード制限セグメントサイズの半分未満に設定します。

新しいセグメントは、当初は予想よりも大きくなる場合がありますが、セグメントが **ram_buffer_size** よりも大きく作成されることは決してありません。このしきい値は予測としてチェックされます。

7.2.6. LockFactory 設定

Lucene ディレクトリーは、Hibernate Search によって管理される各インデックスの **LockingFactory** を介してカスタムロックストラテジーで設定できます。

一部のロックストラテジーには、ファイルシステムレベルのロックが必要で、RAM ベースのインデックスで使用できます。このストラテジーを使用する場合は、**IndexBase** 設定オプションを指定して、ロックマーカーファイルを保存するファイルシステムの場所を指定する必要があります。

ロックファクトリーを選択するには、**hibernate.search.<index>.locking_strategy** オプションを以下のオプションのいずれかに設定します。

- ^simple
- native
- single
- none

表7.6 利用可能な **LockFactory** 実装の一覧

名前	クラス	説明
LockFactoryConfiguration **simple**	org.apache.lucene.store.SimpleFSLockFactory	Java の File API に基づく安全な実装では、マーカーファイルを作成してインデックスの使用をマークします。 何らかの理由でアプリケーションを強制終了する必要がある場合には、このファイルを削除してから再起動する必要があります。

名前	クラス	説明
native	org.apache.lucene.store.NativeFSLockFactory	**simple** と同様に、マーカーファイルを作成してインデックスの使用をマークします。しかし、これはネイティブの OS ファイルロックを使用しているため、JVM が終了してもロックはクリーンアップされます。 この実装では、NFS で既知の問題があるため、ネットワーク共有で使用しないでください。 **native** は、**filesystem**、**filesystem-master**、**filesystem-slave** ディレクトリープロバイダーのデフォルトの実装です。
single	org.apache.lucene.store.SingleInstanceLockFactory	この LockFactory はファイルマーカーを使用しませんが、メモリーに保持される Java オブジェクトロックであるため、インデックスが他のプロセスで共有されないことが確実な場合にのみ使用できます。 これは、**ram** ディレクトリープロバイダーのデフォルト実装です。
none	org.apache.lucene.store.NoLockFactory	このインデックスへの変更は、ロックによって調整されません。

以下は、ロックストラテジーの設定例です。

```
hibernate.search.default.locking_strategy = simple
hibernate.search.Animals.locking_strategy = native
hibernate.search.Books.locking_strategy = org.custom.components.MyLockingFactory
```

7.2.7. インデックス形式の互換性

Hibernate Search では現在、アプリケーションを新しいバージョンへの移植を容易にする後方互換性の API やツールは提供されていません。API は、インデックスの書き込みおよび検索に Apache Lucene を使用します。インデックス形式の更新が必要になる場合があります。この場合、Lucene が古い形式を読み取れない場合は、データのインデックスを再作成する必要があります。

警告

インデックス形式を更新する前にインデックスをバックアップします。

Hibernate Search は、**hibernate.search.lucene_version** 設定プロパティーを公開します。このプロパティーは、古いバージョンの Lucene に定義されている動作に準拠するように Analyzers およびその他の Lucene クラスに指示します。**lucene-core.jar** に含まれる **org.apache.lucene.util.Version** も参照し

てください。このオプションを指定しないと、Hibernate Search はバージョンのデフォルトを使用するよう Lucene に指示します。使用されるバージョンは、アップグレード時に自動的に変更されないように設定に明示的に定義することが推奨されます。アップグレード後、必要に応じて設定値を明示的に更新できます。

Lucene 3.0 の Created Index と互換性を持たせるようにアナライザーを強制する

```
hibernate.search.lucene_version = LUCENE_30
```

設定した **SearchFactory** はグローバルであり、関連するパラメーターが含まれるすべての Lucene API に影響します。Lucene が使用され、Hibernate Search がバイパスされる場合には、一貫した結果を得るために同じ値を適用してください。

7.3. アプリケーションの HIBERNATE SEARCH

7.3.1. Hibernate Search を使用する最初の手順

アプリケーションの Hibernate Search の使用を開始するには、以下のトピックに従います。

- Maven を使用した Hibernate Search の有効化
- インデックス化
- 検索
- アナライザー

7.3.2. Maven を使用した Hibernate Search の有効化

Maven プロジェクトで以下の設定を使用し、**hibernate-search-orm** 依存関係を追加します。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-search-orm</artifactId>
      <version>5.5.1.Final-redhat-1</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

7.3.3. アノテーションの追加

ここでは、書籍の詳細を含むデータベースの例を見てみましょう。アプリケーションに Hibernate 管理クラス **example.Book** および **example.Author** が含まれ、アプリケーションにフリーテキスト検索機能を追加して、書籍の検索を有効にする場合。

例: Hibernate Search 固有のアノテーションを追加する前のエンティティーブックおよび作成者

```
package example;
...
@Entity
public class Book {

  @Id
  @GeneratedValue
  private Integer id;

  private String title;

  private String subtitle;

  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  private Date publicationDate;

  public Book() {}

  // standard getters/setters follow here
  ...
}
```

```
package example;
...
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {}

  // standard getters/setters follow here
  ...
}
```

これを行うには、Book クラスおよび Author クラスにアノテーションをいくつか追加する必要があります。最初のアノテーション **@Indexed** は Book にインデックス可能 (indexable) のマークを付けます。デザイン上、Hibernate Search は未認証の ID をインデックスに保存し、特定のエンティティーのインデックスの一意性を確保します。**@DocumentId** は、この目的に使用するプロパティーをマークし、ほとんどの場合はデータベースのプライマリーキーと同じです。**@DocumentId** アノテーションは、**@Id** アノテーションが存在する場合は任意です。

次に、検索可能にするフィールドには、そのようにマークを付ける必要があります。この例では、**title** と **subtitle** で始まります。両方に、**@Field** アノテーションを付けます。パラメーター **index=Index.YES** は、テキストが確実にインデックス化されるようにします。一方、**analyze=Analyze.YES** は、デフォルトの Lucene アナライザーを使用してテキストが分析されるようにします。通常、分析とは個別の単語に文章を分け、**'a'** や 「the」.などの一般的な単語を除外できる可能性があることを意味します。アナライザーについてもう少し後ほど説明します。**@Field** で指定する第 3 のパラメーター **store=Store.NO** は、実際のデータがインデックスに保存されないようにします。このデータがインデックスに保存されているかどうかや、そのデータを検索する機能がないかどうかです。Lucene の観点からは、インデックスが作成されてからデータを保持する必要はありません。これを保存する利点は、projections を介してそれを取得することです。

Hibernate Search は projections なしで、クエリー基準に一致するエンティティーのデータベース識別子を見つけるためにデフォルトで Lucene クエリーを実行し、これらの識別子を使用してデータベースから管理オブジェクトを取得します。projection については、状況に応じてまたは対照的に決定する必要があります。管理オブジェクトを返す一方で、オブジェクト配列のみを返すため、デフォルトの動作が推奨されます。**Index=Index.YES**, **analyze=Analyze.YES** および **store=Store.NO** はこれらのパラメーターのデフォルト値であるため、省略できることに注意してください。

まだ説明されていない別のアノテーションは **@DateBridge** です。このアノテーションは、Hibernate Search の組み込みフィールドブリッジのいずれかになります。Lucene インデックスは文字列ベースです。このため、Hibernate Search はインデックス化されたフィールドの値タイプを文字列 (またはその逆) に変換する必要があります。事前定義されたブリッジの範囲が提供されます。これには、java.util.Date を指定の解決が含まれる文字列に変換する DateBridge が含まれます。詳細は Bridges を参照してください。

これにより、**@IndexedEmbedded** が残されます。このアノテーションは、所有するエンティティーの一部として関連するエンティティー (@**ManyToMany**、**Gatewaytoone**、**@Embedded**、および **@ElementCollection**) をインデックス化するために使用されます。これは、Lucene インデックスドキュメントがオブジェクト関係について不明なフラットデータ構造であるため必要になります。作成者の名前を確実に検索できるようにするには、名前を書籍の一部としてインデックス化する必要があります。**@IndexedEmbedded** のほかに、インデックスに含める関連エンティティーのフィールドすべてを **@Indexed** でマークする必要があります。詳細は、Embedded and Associated Objects を参照してください。

現在、これらの設定で十分です。エンティティーマッピングの詳細は、Mapping an Entity を参照してください。

例: Hibernate Search アノテーションの追加後のエンティティー

```
package example;
...
@Entity

public class Book {

  @Id
  @GeneratedValue
  private Integer id;

  private String title;

  private String subtitle;

  @Field(index = Index.YES, analyze=Analyze.NO, store = Store.YES)
  @DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;
```

```
  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

```
package example;
...
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {
  }

  // standard getters/setters follow here
  ...
}
```

7.3.4. インデックス化

Hibernate Search は、Hibernate Core を介して永続化、更新、または削除されるすべてのエンティティーを透過的にインデックス化します。ただし、データベースにすでに存在するデータの初期 Lucene インデックスを作成する必要があります。上記のプロパティーとアノテーションを追加したら、book の初期バッチインデックスをトリガーする時間になります。これは、以下のコードスニペットのいずれかを使用して実行できます (参照)。

例: Hibernate Session を使用したデータのインデックス化

```
FullTextSession fullTextSession = org.hibernate.search.Search.getFullTextSession(session);
fullTextSession.createIndexer().startAndWait();
```

例: Jakarta Persistence を使用したデータのインデックス化

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
fullTextEntityManager.createIndexer().startAndWait();
```

上記のコードを実行した後は、**/var/lucene/indexes/example.Book** に Lucene インデックスが表示されるはずです。このインデックスを Luke で検査すると Hibernate Search の仕組みを理解するのに役立ちます。

7.3.5. 検索

検索を実行するには、Lucene API または Hibernate Search query DSL のいずれかを使用して Lucene クエリーを作成します。org.hibernate.Query にクエリーをラップし、Hibernate API から必要な機能を取得します。以下のコードは、インデックスフィールドに対するクエリーを準備します。コードを実行すると、書籍の一覧が返されます。

例: Hibernate Search セッションを使用した検索の作成および実行

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();

// create native Lucene query using the query DSL
// alternatively you can write the Lucene query using the Lucene query parser
// or the Lucene programmatic API. The Hibernate Search DSL is recommended though
QueryBuilder qb = fullTextSession.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
  .keyword()
  .onFields("title", "subtitle", "authors.name", "publicationDate")
  .matching("Java rocks!")
  .createQuery();

// wrap Lucene query in a org.hibernate.Query
org.hibernate.Query hibQuery =
    fullTextSession.createFullTextQuery(query, Book.class);

// execute search
List result = hibQuery.list();

tx.commit();
session.close();
```

例: Jakarta Persistence を使用した検索の作成および実行

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
em.getTransaction().begin();

// create native Lucene query using the query DSL
// alternatively you can write the Lucene query using the Lucene query parser
// or the Lucene programmatic API. The Hibernate Search DSL is recommended though
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
  .keyword()
  .onFields("title", "subtitle", "authors.name", "publicationDate")
  .matching("Java rocks!")
  .createQuery();

// wrap Lucene query in a javax.persistence.Query
javax.persistence.Query persistenceQuery =
    fullTextEntityManager.createFullTextQuery(query, Book.class);
```

```
// execute search
List result = persistenceQuery.getResultList();

em.getTransaction().commit();
em.close();
```

7.3.6. アナライザー

インデックス化された Book エンティティーのタイトルが **Refactoring: Improving the Design of Existing Code** で、**refactor**、**refactors**、**refactored**、**refactoring** のクエリーに必要だとします。インデックス化および検索を行う際にワードステミングを適用する Lucene でアナライザークラスを選択します。Hibernate Search は、アナライザーを設定する方法を複数提供します (詳細は Default Analyzer and Analyzer by Class 参照してください)。

- 設定ファイルに **analyzer** プロパティーを設定します。指定されたクラスがデフォルトのアナライザーになります。
- エンティティーレベルで **@Analyzer** アノテーションを設定します。
- フィールドレベルで **@Analyzer** アノテーションを設定します。

完全修飾クラス名または使用するアナライザーを指定するか、**@Analyzer** アノテーションとともに **@AnalyzerDef** アノテーションで定義されているアナライザーを確認します。Solr アナライザーフレームワークとそのファクトリーは、後者のオプションに使用されます。ファクトリークラスの詳細は、Solr JavaDoc を参照するか、Solr Wiki の該当するセクションを参照してください。

この例では、StandardTokenizerFactory が、LowerCaseFilterFactory と SnowballPorterFilterFactory の 2 つのフィルターファクトリーによって使用されています。トークンライザーは、英数字とハイフンで単語を分割しますが、メールアドレスとインターネットのホスト名は維持します。標準トークンは、これおよびその他の一般的な操作に適しています。小文字フィルターはトークンのすべての文字を小文字に変換し、snowball フィルターは言語固有のステミングを適用します。

Solr フレームワークを使用する場合は、任意の数のフィルターでトークンを使用します。

例: @AnalyzerDef および Solr Framework を使用した Analyzer の定義および使用

```
@Indexed
@AnalyzerDef(
  name = "customanalyzer",
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = SnowballPorterFilterFactory.class,
      params = { @Parameter(name = "language", value = "English") })
 })
public class Book implements Serializable {

  @Field
  @Analyzer(definition = "customanalyzer")
  private String title;

  @Field
  @Analyzer(definition = "customanalyzer")
  private String subtitle;

  @IndexedEmbedded
```

```
  private Set authors = new HashSet();

  @Field(index = Index.YES, analyze = Analyze.NO, store = Store.YES)
  @DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

@AnalyzerDef を使用してアナライザーを定義し、@Analyzer を使用してエンティティーおよびプロパティーに適用します。この例では、**customanalyzer** は定義されますが、エンティティーには適用されません。アナライザーは **title** および **subtitle** プロパティーのみに適用されます。アナライザーの定義はグローバルです。エンティティーのアナライザーを定義し、必要に応じて他のエンティティーの定義を再利用します。

7.4. インデックス構造へのエンティティーのマッピング

7.4.1. エンティティーのマッピング

エンティティーのインデックス化に必要なすべてのメタデータ情報はアノテーションで記述されるため、XML マッピングファイルは必要ありません。基本的な Hibernate 設定には Hibernate マッピングファイルも使用できますが、Hibernate Search 固有の設定はアノテーションで表現する必要があります。

7.4.1.1. 基本的なマッピング

エンティティーのマッピングに最も一般的に使用されるアノテーションから始めましょう。

Lucene ベースの Query API は、以下の共通アノテーションを使用してエンティティーをマッピングします。

- @Indexed
- @Field
- @NumericField
- @Id

7.4.1.2. @Indexed

ほとんどの場合、永続クラスをインデックス化可能として宣言する必要があります。これは、クラスに **@Indexed** アノテーションを付けることで行われます (**@Indexed** アノテーションが付いていないすべてのエンティティーはインデックスプロセスによって無視されます)。

```
@Entity
@Indexed
public class Essay {
...
}
```

オプションで @Indexed アノテーションの **index** 属性を指定して、インデックスのデフォルト名を変更できます。

7.4.1.3. @Field

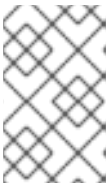
エンティティーのプロパティー (または属性) ごとに、インデックス化方法を記述する機能があります。デフォルト (アノテーションなし) は、インデックスプロセスによってプロパティーが無視されることを意味します。

注記

Hibernate Search 5 以前は、**@NumericField** で明示的に要求された場合に限り、数値フィールドエンコーディングが選択されていました。Hibernate Search 5 では、このエンコーディングは数値タイプに自動的に選択されます。数値エンコーディングを回避するには、非数値フィールドブリッジ **@Field.bridge** または **@FieldBridge** を明示的に指定できます。パッケージ **org.hibernate.search.bridge.builtin** には、**org.hibernate.search.bridge.builtin.IntegerBridge** などの数を文字列としてエンコードする一連のブリッジが含まれます。

@Field はプロパティーをインデックスとして宣言し、以下の属性のいずれかを設定してインデックスプロセスの複数の側面を設定できます。

- **name**: プロパティーがどの名前下で、Lucene Document に保存されるべきかを説明します。デフォルト値はプロパティー名です (続く JavaBeans 規則)。
- **store**: プロパティーが Lucene インデックスに保存されているかどうかを示します。**Store.YES** の値 (インデックスに多くの領域が必要ですが projection を許可) を保存するか、これを圧縮方式 **Store.COMPRESS** で保存、あるいは、**Store.NO** を回避することができます (デフォルト値)。プロパティーが保存されると、Lucene ドキュメントから元の値を取得できます。これは、要素がインデックス化されるかどうかに関連しません。
- **index**: プロパティーがインデックス化されるかどうかを示します。異なる値は **Index.NO** で、これはインデックス化されず、クエリーおよび **Index.YES** によって見つからりません。これは、要素がインデックス化され、検索可能であることを意味します。デフォルト値は **Index.YES** です。**Index.NO** は、プロパティーの検索が不可能であるものの、利用できる必要がある場合に便利です。

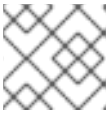
  注記

  Index.NO を **Analyze.YES** または **Norms.YES** と組み合わせると有用なわけではありません。これは、**analyze** と **norms** と解析のプロパティーのインデックス作成が必要ないためです。

- **analyze**: プロパティーが分析されたかどうか (**Analyze.YES**) または (**Analyze.NO**) を判断します。デフォルト値は **Analyze.YES** です。

  注記

  プロパティーを分析するかどうかは、要素をそのまま 0 検索する場合と、含まれる単語で検索するかによって異なります。テキストフィールドを分析することは理にかなっていますが、日付フィールドは分析しません。



注記

ソートに使用されるフィールドは、分析できません。

- **norms**: インデックス時間の改善情報を保存する必要があるかどうか (**Norms.YES**) または (**Norms.NO**) を示します。これを保存しないと、大量のメモリーを節約できますが、インデックスの時間が改善する情報は提供されません。デフォルト値は **Norms.YES** です。

- **termVector**: 用語と周波数のペア (term-frequency) のコレクションについて説明しています。この属性により、インデックス作成中にドキュメント内にベクターを保存することができます。デフォルト値は **TermVector.NO** です。
この属性の異なる値は次のとおりです。

値	定義
TermVector.YES	各ドキュメントの Term Vectors を保存します。これにより、同期されたアレイが作成され、これらはドキュメント用語が含まれ、他は用語の周波数が含まれます。
TermVector.NO	Term Vector は保存しないでください。
TermVector.WITH_OFFSETS	Term Verctor およびトークンオフセット情報を保存します。これは TermVector.YES と同様で、用語の開始および終了オフセット位置情報が含まれます。
TermVector.WITH_POSITIONS	Term Verctor およびトークン位置情報を保存します。これは TermVector.YES と同じですが、ドキュメント内の各用語の特徴も含まれます。
TermVector.WITH_POSITION_OFFSETS	Term Vector 、トークンの位置、およびオフセット情報を格納します。これは、YES、WITH_OFFSETS、および WITH_POSITIONS の組み合わせです。

- **indexNullAs**: デフォルトの null 値ごとに無視され、インデックスは作成されません。ただし、**indexNullAs** を使用すると、**null** 値のトークンとして挿入される文字列を指定できます。デフォルトでは、この値は **Field.DO_NOT_INDEX_NULL** に設定され、**null** 値がインデックス化されないことを示します。この値を **Field.DEFAULT_NULL_TOKEN** に設定すると、デフォルトの **null** トークンが使用されます。このデフォルトの **null** トークンは、**hibernate.search.default_null_token** を使用して設定に指定できます。このプロパティーを設定せず、**Field.DEFAULT_NULL_TOKEN** を指定すると、文字列 "null" がデフォルトとして使用されます。

注記

indexNullAs パラメーターを使用する場合は、検索クエリーで同じトークンを使用して **null** 値を検索することが重要です。また、この機能は、非分析フィールド (**analyze.NO**) でのみ使用することが推奨されます。

警告

カスタムの FieldBridge または TwoWayFieldBridge を実装する場合、開発者は null 値のインデックス化を処理することになります (LuceneOptions.indexRulesAs() の Java ドキュメントを参照)。

7.4.1.4. @NumericField

@Field には @NumericField というコンパニオンアノテーションがあり、@Field または @DocumentId と同じスコープで指定できます。このプロパティーは、Integer、Long、Float、および LastName プロパティーに指定できます。インデックスの作成時に、値は Trie 構造を使用してインデックス化されます。プロパティーを数字フィールドとしてインデックス化すると、標準的な @Field プロパティーに対して同じクエリーを実行するよりも、効率的な範囲クエリーとソートが可能になります。@NumericField アノテーションは以下のパラメーターを受け入れます。

値	定義
forField	(オプション) 数値としてインデックス化される関連 @Field の名前を指定します。@Field 宣言を超えるプロパティーが含まれる場合にのみ必須となります。
precisionStep	(オプション) インデックスに Trie 構造を格納する方法を変更します。precisionStes を小さくすると、ディスク領域の使用率が高くなり、範囲やソートのクエリーが速くなります。値が大きいほど使用領域が少なくなり、クエリーのパフォーマンスは通常の @Fields の範囲のクエリーに近づくことになります。デフォルト値は 4 です。

@NumericField は、LOCATION、Long、Integer、および Float のみに対応しています。他の数値タイプには Lucene で同様の機能を活用できないため、残りのタイプはデフォルトまたはカスタムの TwoWayFieldBridge で文字列エンコーディングを使用する必要があります。

タイプ変換中に概算を処理できると仮定した場合は、カスタムの NumericFieldBridge を使用することができます。

例: カスタムの NumericFieldBridge の定義

```
public class BigDecimalNumericFieldBridge extends NumericFieldBridge {
  private static final BigDecimal storeFactor = BigDecimal.valueOf(100);

  @Override
  public void set(String name, Object value, Document document, LuceneOptions luceneOptions) {
    if ( value != null ) {
      BigDecimal decimalValue = (BigDecimal) value;
      Long indexedValue = Long.valueOf( decimalValue.multiply( storeFactor ).longValue() );
      luceneOptions.addNumericFieldToDocument( name, indexedValue, document );
    }
  }

   @Override
```

```
    public Object get(String name, Document document) {
        String fromLucene = document.get( name );
        BigDecimal storedBigDecimal = new BigDecimal( fromLucene );
        return storedBigDecimal.divide( storeFactor );
    }

}
```

7.4.1.5. @Id

最後に、エンティティーの **id** (identifier) プロパティーは、特定のエンティティーのインデックスを一意に保つために Hibernate Search で使用される特別なプロパティーです。設計上、**id** は保存する必要があり、トークン化しないでください。プロパティーをインデックス識別子としてマークするには、**@DocumentId** アノテーションを使用します。Jakarta Persistence を使用し、@Id を指定すると @DocumentId を省略できます。選択したエンティティー ID は、ドキュメント識別子として使用されます。

Infinispan Query はエンティティーの **id** プロパティーを使用して、インデックスが一意に識別されるようにします。設計上、ID は保存されます。トークンに変換しないでください。プロパティーにインデックス ID のマークを付けるには、**@DocumentId** アノテーションを使用します。

例: インデックス化されたプロパティーの指定

```
@Entity
@Indexed
public class Essay {
    ...
    @Id
    @DocumentId
    public Long getId() { return id; }

    @Field(name="Abstract", store=Store.YES)
    public String getSummary() { return summary; }

    @Lob
    @Field
    public String getText() { return text; }

    @Field @NumericField( precisionStep = 6)
    public float getGrade() { return grade; }
}
```

上記の例では、**id**、**Abstract**、**text**、**grade** のフィールドを持つインデックスを定義します。デフォルトでは、JavaBean 仕様にしたがってフィールド名は大文字では表示されないことに注意してください。**Grade** フィールドは、デフォルトよりも若干精度の高いステップで数字としてアノテーションが付けられます。

7.4.1.6. 複数回のプロパティーのマッピング

若干異なるインデックスストラテジーで、インデックスごとにプロパティーを複数回マップする必要がある場合があります。たとえば、フィールド別にクエリーを並び替えるには、フィールドを分析解除する必要があります。このプロパティーの単語で検索し、これをソートするには、分析後と未分析のときにインデックス付けする必要があります。これは、@Fields を使用することで可能です。

例: @Fields を使用した複数時間のプロパティーのマップ

```
@Entity
@Indexed(index = "Book" )
public class Book {
   @Fields( {
         @Field,
         @Field(name = "summary_forSort", analyze = Analyze.NO, store = Store.YES)
         } )
   public String getSummary() {
      return summary;
   }
   ...
}
```

この例では、フィールドの **summary** は 2 回インデックス化されます。これはトークン化方式の **summary**、非トークン化方式の **summary_forSort** で行われます。

7.4.1.7. 埋め込みおよび関連オブジェクト

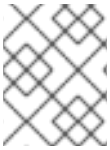
関連オブジェクトおよび組み込みオブジェクトは、ルートエンティティーインデックスの一部としてインデックス化できます。これは、関連するオブジェクトのプロパティーに基づいて特定のエンティティーを検索する場合に役に立ちます。関連する都市が Atlanta である場所を返すことを目的としています (Lucene クエリーパーサー言語で、**address.city:Atlanta** に変換されます)。場所フィールドは、**Place** インデックスでインデックス化されます。**Placement** インデックスドキュメントには、クエリー可能な **address.id**、**address.street**、**address.city** フィールドも含まれます。

例: 関連付けのインデックス化

```
@Entity
@Indexed
public class Place {
   @Id
   @GeneratedValue
   @DocumentId
   private Long id;

   @Field
   private String name;

   @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
   @IndexedEmbedded
   private Address address;
   ....
}

@Entity
public class Address {
   @Id
   @GeneratedValue
   private Long id;

   @Field
   private String street;

   @Field
   private String city;
```

```
    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}
```

@IndexedEmbedded 技術を使用すると、データは Lucene インデックスで非正規化されるため、Hibernate Search は Place オブジェクトのすべての変更と、インデックスを最新の状態に保つため Address オブジェクトの変更を認識する必要があります。Lucene ドキュメントが Address の変更時に更新されるようにするには、双方向関係の反対側に **@ContainedIn** のマークを付けます。



注記

@ContainedIn はエンティティーを参照する関連付けや、組み込み (コレクション) オブジェクトを参照する関連付けで役立ちます。

この例を展開するために、以下の例では **@IndexedEmbedded** のネスト化を示しています。

例: @IndexedEmbedded および @ContainedIn のネスト化された使用方法

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field
    private String street;

    @Field
    private String city;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_")
    private Owner ownedBy;

    @ContainedIn
    @OneToMany(mappedBy="address")
```

```
    private Set<Place> places;
    ...
}

@Embeddable
public class Owner {
    @Field
    private String name;
    ...
}
```

@*ToMany, **@*ToOne** および **@Embedded** 属性は、**@IndexedEmbedded** アノテーションを付けることができます。その後、関連クラスの属性が主なエンティティーインデックスに追加されます。インデックスには以下のフィールドが含まれます。

- id
- name
- address.street
- address.city
- address.ownedBy_name

デフォルトの接頭辞は **propertyName.** で、従来のオブジェクトナビゲーション規則に従います。**ownedBy** プロパティーに示されるように、**prefix** 属性を使用して上書きできます。

注記

接頭辞を空の文字列に設定することはできません。

depth プロパティーは、オブジェクトグラフにクラス (インスタンスではない) の cyclic 依存関係が含まれるときに必要になります。たとえば、Owner が Place をポイントする場合です。Hibernate Search は、予想される深さに達すると (またはオブジェクトグラフの境界に到達する)、インデックス化された組み込み属性を含まなくなります。自己参照を持つクラスは、cyclic 依存関係の例です。この例では、**depth** が 1 に設定されているため、Owner の **@IndexedEmbedded** 属性は無視されます。

オブジェクト関連付けに **@IndexedEmbedded** を使用すると、以下のようなクエリーを表現できます (Lucene のクエリー構文を使用)。

- 名前に JBoss が含まれ、住所の都市が Atlanta の場所を返します。Lucene クエリーでは、以下のようになります。

  ```
  +name:jboss +address.city:atlanta
  ```

- 名前に JBoss が含まれ、所有者名に Joe が含まれる場所を返します。Lucene クエリーでは、以下のようになります。

  ```
  +name:jboss +address.ownedBy_name:joe
  ```

この動作は、より効率的な方法 (データの重複が犠牲となる) でのリレーショナルジョイン操作の操作に似ています。初期状態の Lucene インデックスには関連付けの概念がないため、join 操作が存在しないことに注意してください。これは、完全なテキストインデックスの速度と機能が充実した状態で、リレーショナルデータベースを維持するのに役立ちます。

注記

関連付けられたオブジェクトは、**@Indexed** にすることができます (ただし、必須ではありません)。

@IndexedEmbedded がエンティティーを参照する場合、関連付けには指向性が必要で、反対側にはアノテーション **@ContainedIn** を付ける必要があります (前述の例を参照)。これがない場合、Hibernate Search は関連エンティティーの更新時にルートインデックスを更新することはできません (この例では、関連付けられた Address インスタンスの更新時に **Place** インデックスドキュメントを更新する必要があります)。

@IndexedEmbedded アノテーションが付けられたオブジェクトタイプは、Hibernate および Hibernate Search によってターゲットに設定されたオブジェクトタイプではない場合があります。これは、インターフェイスが実装の代わりに使用される場合にとくに当てはまります。このため、**targetElement** パラメーターを使用して、Hibernate Search がターゲットとするオブジェクトタイプを上書きできます。

例: @IndexedEmbedded の targetElement プロパティーの使用

```
@Entity
@Indexed
public class Address {
   @Id
   @GeneratedValue
   @DocumentId
   private Long id;

   @Field
   private String street;

   @IndexedEmbedded(depth = 1, prefix = "ownedBy_", )
   @Target(Owner.class)
   private Person ownedBy;
   ...
}

@Embeddable
public class Owner implements Person { ... }
```

7.4.1.8. 特定のパスへのオブジェクト埋め込みの制限

@IndexedEmbedded アノテーションは、属性 includePaths も提供します。これは、デプスの代わりとして使用することも、属性と組み合わせることもできます。

デプスのみを使用すると、埋め込み型のインデックス設定されたフィールドはすべて、同じデプスで再帰的に追加されます。これにより、他のフィールドもすべて追加せずに特定のパスのみを選択することが困難になります。これは必須ではありません。

不要な読み込みおよびインデックスエンティティーを回避するには、必要なパスを正確に指定することができます。通常のアプリケーションでは、パスごとに異なるデプスが必要な場合や、以下の例のようにパスを明示的に指定する必要がある場合があります。

例: @IndexedEmbedded の includePaths プロパティーの使用

```
@Entity
```

```
@Indexed
public class Person {

  @Id
  public int getId() {
    return id;
  }

  @Field
  public String getName() {
    return name;
  }

  @Field
  public String getSurname() {
    return surname;
  }

  @OneToMany
  @IndexedEmbedded(includePaths = { "name" })
  public Set<Person> getParents() {
    return parents;
  }

  @ContainedIn
  @ManyToOne
  public Human getChild() {
    return child;
  }

   ...//other fields omitted
```

上記の例のようにマッピングを使用すると、名前やサムネーム、**name** や **surname**、親の **name** で Person の検索 w お行うことができます。親の **surname** をインデックス化しないため、親の surname を検索することはできません。ただし、インデックス作成を迅速化し、スペースを節約して、全体的なパフォーマンスを向上させることができます。

@IndexedEmbeddedincludePaths には、通常はデプスの制限された値を指定するのに加え、指定されたパスが含まれます。IncludePaths を使用し、デプスを未定義のままにすると、depth=0 を設定するのと同等の動作になります。含まれるパスのみがインデックス化されます。

例: @IndexedEmbedded の includePaths プロパティーの使用

```
@Entity
@Indexed
public class Human {

  @Id
  public int getId() {
    return id;
  }

  @Field
  public String getName() {
    return name;
  }
```

```
@Field
public String getSurname() {
  return surname;
}

@OneToMany
@IndexedEmbedded(depth = 2, includePaths = { "parents.parents.name" })
public Set<Human> getParents() {
  return parents;
}

@ContainedIn
@ManyToOne
public Human getChild() {
  return child;
}

 ...//other fields omitted
```

上記の例では、すべての人間 (human) の名前と surname 属性にインデックスが付けられます。また、depth 属性が原因で、親の名前と surname (姓) も再帰的に 2 行目にインデックス化されます。人が直接、自身の親、または親の名前で検索することができます。第 2 レベル以外では、姓 (surname) ではなく、もう 1 レベル (名前のみ) をインデックス化します。

これにより、インデックスに以下のフィールドが生成されます。

- **id**: プライマリーキーとして
- **_hibernate_class**: エンティティータイプを保存
- **name**: 直接フィールドとして
- **surname**: 直接フィールドとして
- **parents.name**: デプス 1 の埋め込みフィールドとして
- **parents.surname**: デプス 1 の埋め込みフィールドとして
- **parents.parents.name**: デプス 2 の埋め込みフィールドとして
- **parents.parents.surname**: デプス 2 の埋め込みフィールドとして
- **parents.parents.parents.name**: includePaths で指定される追加パスとして最初の **parents** はフィールド名から推測され、残りのパスは includePaths の属性です。

インデックス化されたパスを明示的に制御することは、必要なクエリーを最初に定義してアプリケーションを設計する場合に容易になる可能性があります。この時点では、どのフィールドが必要かを正確に把握している可能性もあります。その他のどのフィールドがユースケースを実装する必要はありません。

7.4.2. Boosting

Lucene には **boosting** の概念があります。そのため、特定のドキュメントやフィールドに対して、他のドキュメントまたはフィールドよりも重要性を高くするか、低いものにすることができます。Lucene は、インデックスと検索時間の改善を区別します。以下のセクションでは、Hibernate Search

を使用してインデックス時間を改善する方法を説明します。

7.4.2.1. 静的インデックスの時間の改善

インデックス化されたクラスまたはプロパティーの静的なブースト値を定義するには、**@Boost** アノテーションを使用できます。このアノテーションは、**@Field** 内で使用することも、メソッドまたはクラスレベルで直接指定することもできます。

例: @Boost を使用するさまざまな方法

```
@Entity
@Indexed

public class Essay {
    ...

    @Id
    @DocumentId
    public Long getId() { return id; }

    @Field(name="Abstract", store=Store.YES, boost=@Boost(2f))
    @Boost(1.5f)
    public String getSummary() { return summary; }

    @Lob
    @Field(boost=@Boost(1.2f))
    public String getText() { return text; }

    @Field
    public String getISBN() { return isbn; }
}
```

上記の例では、Essay が検索一覧の最上位に到達する可能性は 1.7 で乗算されます。summary フィールドは、isbn フィールドに比べて 3.0 重要になります (2 * 1.5。これは、isbn フィールドよりも @Field.boost および @Boost が累積的であるためです)。Text フィールドは isbn フィールドよりも 1.2 倍重要になります。この説明は、最も厳格な条件では間違っていますが、あらゆる実用的な目的において、実用性に十分に近い点に注意してください。

7.4.2.2. 動的インデックス時間の改善

Static Index Time Boosting (静的インデックスの時間の改善) で使用される **@Boost** アノテーションは、ランタイム時にインデックス化されたエンティティーの状態に依存しない静的なブーディング係数を定義します。ただし、改善要因がエンティティーの実際の状態に依存する可能性があるユースケースがあります。この場合は、**@DynamicBoost** アノテーションと付随するカスタム BoostStrategy を使用できます。

例: 動的ブースト

```
public enum PersonType {
    NORMAL,
    VIP
}

@Entity
@Indexed
```

```
@DynamicBoost(impl = VIPBoostStrategy.class)
public class Person {
    private PersonType type;

    // ....
}

public class VIPBoostStrategy implements BoostStrategy {
    public float defineBoost(Object value) {
        Person person = ( Person ) value;
        if ( person.getType().equals( PersonType.VIP ) ) {
            return 2.0f;
        }
        else {
            return 1.0f;
        }
    }
}
```

上記の例では、動的ブーストは、インデックス処理時に使用される BoostStrategy インターフェイスの実装として VIPBoostStrategy を指定するクラスレベルで定義されます。**@DynamicBoost** は、クラスまたはフィールドレベルのいずれかで配置できます。アノテーションの配置に応じて、エンティティー全体が defineBoost メソッドに渡されるか、アノテーションが付いたフィールド/プロパティー値のみに渡されます。渡されたオブジェクトを正しいタイプにキャストするのはユーザー自身です。この例では、VIP ユーザーのすべてのインデックス化された値が通常の人の値と同じくらい重要になります。

注記

指定された BoostStrategy 実装は、パブリックの no-arg コンストラクターを定義する必要があります。

当然ながら、エンティティーで **@Boost** と **@DynamicBoost** DynamicBoost アノテーションを混在させることができます。定義されたすべてのブースター要素は累積的です。

7.4.3. 分析

Analysis、テキストを 単一の用語 (単語) に変換するプロセスであり、フルテキスト検索エンジンの主な機能として見なされます。Lucene は、Analyzers の概念を使用してこのプロセスを制御します。以下のセクションでは、Hibernate Search がアナライザーを設定するために提供する複数の方法について説明します。

7.4.3.1. デフォルトの Analyzer とクラスによる Analyzer

トークン化されたフィールドのインデックス化に使用されるデフォルトのアナライザークラスは、**hibernate.search.analyzer** プロパティーで設定できます。このプロパティーのデフォルト値は **org.apache.lucene.analysis.standard.StandardAnalyzer** です。

また、エンティティーやプロパティー、さらには @Field ごとにアナライザークラスを定義することもできます (複数のフィールドが単一のプロパティーからインデックス化される場合に便利です)。

例: @Analyzer の異なる使用

```
@Entity
@Indexed
```

```
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {
    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;

    @Field
    private String name;

    @Field
    @Analyzer(impl = PropertyAnalyzer.class)
    private String summary;

    @Field(analyzer = @Analyzer(impl = FieldAnalyzer.class)
    private String body;
    ...
}
```

この例では、EntityAnalyzer を使用して、PropertyAnalyzer と FieldAnalyzer でインデックス化される **summary** と **body** を除き トークン化されたプロパティー (**name**) をインデックス化します。

警告

同じエンティティーで異なるアナライザーを組み合わせることは多くの場合で適切ではありません。特にクエリー全体に同じパーサーを使用する QueryParser を使用している場合、クエリーの構築がより複雑になり、結果の予測が難しくなります。経験上、特定のフィールドでは、同じアナライザーを使用してインデックス付けとクエリーを実行する必要があります。

7.4.3.2. Named Analyzers

アナライザーの処理は非常に複雑になる可能性があります。このため、Hibernate Search にはアナライザ定義の概念が導入されました。アナライザ定義は、@Analyzer 宣言の多くで再利用でき、以下で設定されています。

- **a name:** 定義を参照するために使用される一意な文字列
- **a list of char filters:**各文字型フィルターは、トークン化の前に入力文字を事前処理します。文字型フィルターでは、文字の追加、変更、削除ができます。一般的な使用方法として、文字を正規化する方法があります。
- **a tokenizer:** 入力ストリームを個別の単語にトークン化します。
- **a list of filters:**各フィルターは、単語の削除や変更を行い、トークンライザーが提供するストリームに単語を追加することさえあります。

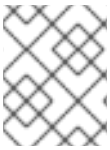
タスクの分離: 文字型フィルターのリストとトークンライザー、およびフィルターの一覧が続きます。これにより、個々のコンポーネントを簡単に再利用でき、(Lego など) 非常に柔軟な方法でカスタマイズされたアナライザーを構築することができます。一般的に、文字型フィルターは文字入力で一部の事

前処理を実行し、Tokenizer は、文字入力を TokenFilters によってさらに処理されるトークンに変換してトークン処理を開始します。Hibernate Search は Solr Analyzer フレームワークを使用してこのインフラストラクチャーをサポートします。

以下に記載されている具体的な例を見てみましょう。まず、文字型フィルターはファクトリーによって定義されます。この例では、マッピング文字型フィルターが使用され、マッピングファイルに指定されたルールに基づいて入力内の文字が置き換えられます。次にトークナイザーを定義します。この例では、標準トークナイザーを使用します。最後に同じように重要に、のフィルターの一覧がファクトリーによって定義されます。この例では、StopFilter フィルターは、専用の用語プロパティーファイルを読み取ります。フィルターはケースを無視することも予想されます。

例: @AnalyzerDef および Solr フレームワーク

```
@AnalyzerDef(name="customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value = "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
    })
  },
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
      @Parameter(name="words",
        value= "org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
      @Parameter(name="ignoreCase", value="true")
    })
})
public class Team {
    ...
}
```



注記

フィルターと文字型フィルターは、@AnalyzerDef アノテーションで定義された順序で適用されます。順序は重要です。

一部のトークナイザー、トークンフィルター、または文字型フィルターは、設定ファイルやメタデータファイルなどのリソースを読み込みます。これは、ストップフィルターと同意フィルターの例になります。リソース文字セットが仮想マシンのデフォルトを使用していない場合は、**resource_charset** パラメーターを追加して明示的に指定できます。

例: 特定の文字セットを使用したプロパティーファイルの読み込み

```
@AnalyzerDef(name="customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value = "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
    })
  },
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
```

```
  filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
      @Parameter(name="words",
        value= "org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
      @Parameter(name="resource_charset", value = "UTF-16BE"),
      @Parameter(name="ignoreCase", value="true")
  })
})
public class Team {
    ...
}
```

定義が完了すると、以下の例のように @Analyzer 宣言でアナライザーの定義を再利用できます。

例: 名前でのアナライザーの参照

```
@Entity
@Indexed
@AnalyzerDef(name="customanalyzer", ... )
public class Team {
    @Id
    @DocumentId
    @GeneratedValue
    private Integer id;

    @Field
    private String name;

    @Field
    private String location;

    @Field
    @Analyzer(definition = "customanalyzer")
    private String description;
}
```

@AnalyzerDef で宣言された Analyzer インスタンスは、SearchFactory の名前でも利用できます.これは、クエリーを構築する際に非常に便利です。

```
Analyzer analyzer = fullTextSession.getSearchFactory().getAnalyzer("customanalyzer");
```

クエリー内のフィールドは、共通の言語を伝えるためにフィールドのインデックス作成に使用するものと同じアナライザーで分析する必要があります。クエリーとインデックス作成プロセス間で同じトークンが再利用されます。このルールには例外がありますが、ほとんどの場合に当てはまります。実際に何を実行しているのかが分からない限り、それに従ってください。

7.4.3.3. 利用可能なアナライザー

Solr および Lucene には、多くの便利なデフォルトの文字型フィルター、トークンサイザー、およびフィルターがあります。文字型フィルターファクトリー、トークン化ファクトリー、およびフィルターファクトリーの完全な一覧は、http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters にあります。それらのいくつかをチェックしてください。

表7.7 利用可能な文字型フィルター

ファクトリー	説明	パラメーター
MappingCharFilterFactory	リソースファイルで指定されたマッピングに基づいて、1文字または複数の文字を1文字または複数の文字に置き換えます。	**mapping**: "á" ⇒ "a"、"ñ" ⇒ "n"、"ø" ⇒ "o" の形式で、マッピングを含むリソースファイルにを対象とします。
HTMLStripCharFilterFactory	HTML 標準のタグを削除し、テキストを保持します。	none

表7.8 利用可能なトークナイザー

ファクトリー	説明	パラメーター
StandardTokenizerFactory	Lucene StandardTokenizer の使用	none
HTMLStripCharFilterFactory	HTML タグを削除し、テキストを保持して StandardTokenizer に渡します。	none
PatternTokenizerFactory	指定された正規表現パターンでテキストを区切ります。	**pattern**: トークン化に使用する正規表現 **group**: トークンに抽出するパターングループを示します。

表7.9 利用可能なフィルター

ファクトリー	説明	パラメーター
StandardFilterFactory	略語および単語からドットを削除する	none
LowerCaseFilterFactory	すべての単語を小文字にします	none
StopFilterFactory	ストップワードの一覧に一致する単語 (トークン) を削除します。	**words**: ストップワードを含むリソースファイルを参照します。 **ignoreCase**: ストップワードを比較する際に大文字と小文字が無視される必要がある場合は true、そうでない場合は false を設定します。

ファクトリー	説明	パラメーター
SnowballPorterFilterFactory	特定の言語で、単語を語根に減らします (例: protect、protects、protection は同じ語根を共有)。このようなフィルターを使用すると、関連する単語を検索できます。	**language**: デンマーク語、オランダ語、英語、フィンランド語、フランス語、ドイツ語、イタリア語、ノルウェー語、ポルトガル語、ロシア語、韓国語、スウェーデン語など

IDE で **org.apache.lucene.analysis.TokenizerFactory** and **org.apache.lucene.analysis.TokenFilterFactory** のすべての実装を確認して、利用可能な実装を確認することが推奨されます。

7.4.3.4. 動的アナライザーの選択

現時点で、アナライザーを指定する方法はすべて静的でした。ただし、インデックスを作成するエンティティーの現在の状態 (たとえば多言語アプリケーション) に応じてアナライザーを選択すると便利なユースケースがあります。たとえば、BlogEntry クラスの場合、アナライザーはエントリーの言語プロパティーに依存する可能性があります。このプロパティーによっては、実際のテキストにインデックスを付けるために正しい言語固有のスチーマーを選択する必要があります。

この動的アナライザーを有効にするために、Hibernate Search で AnalyzerDiscriminator アノテーションが導入されました。以下の例は、このアノテーションの使用方法を示しています。

例: @AnalyzerDiscriminator の使用

```
@Entity
@Indexed
@AnalyzerDefs({
  @AnalyzerDef(name = "en",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = EnglishPorterFilterFactory.class
      )
    }),
  @AnalyzerDef(name = "de",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = GermanStemFilterFactory.class)
    })
})
public class BlogEntry {

    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;
```

```
    @Field
    @AnalyzerDiscriminator(impl = LanguageDiscriminator.class)
    private String language;

    @Field
    private String text;

    private Set<BlogEntry> references;

    // standard getter/setter
    ...
}
```

```
public class LanguageDiscriminator implements Discriminator {

    public String getAnalyzerDefinitionName(Object value, Object entity, String field) {
        if ( value == null || !( entity instanceof BlogEntry ) ) {
            return null;
        }
        return (String) value;

    }
}
```

@AnalyzerDiscriminator を使用するための前提条件は、動的に使用されるすべてのアナライザーが **@AnalyzerDef** 定義で事前定義されることです 。このような場合、クラスまたはアナライザーを動的に選択するエンティティーの特定のプロパティーに **@AnalyzerDiscriminator** アノテーションを配置することができます。**AnalyzerDiscriminator** の **impl** パラメーターを使用すると、Dicriminator インターフェイスの具体的な実装を指定できます。このインターフェイスの実装は、ユーザー自身が提供する必要があります。実装が必要な唯一の方法は **getAnalyzerDefinitionName()** で、これは Lucene ドキュメントに追加された各フィールドに対して呼び出されます。インデックスを取得するエンティティーもインターフェイスメソッドに渡されます。**value** パラメーターは、**AnalyzerDiscriminator** がクラスレベルではなくプロパティーレベルに配置されている場合にのみ設定されます。この場合、値はこのプロパティーの現在の値を表します。

Discriminator インターフェイスの実装では、既存のアナライザー定義の名前を返し、デフォルトのアナライザーが上書きされない場合は null を返します。上記の例では、言語パラメーターは **@AnalyzerDefs** で指定された名前に一致する 'de' または 'en' であることを仮定しています 。

7.4.3.5. アナライザーの取得

複数のアナライザーがドメインモデルで使用される場合、スチーミングや概算などの利点を活かすために、アナライザーを取得することができます。この場合、同じアナライザーを使用してクエリーを作成します。または、正しいアナライザーを自動的に選択する Hibernate Search クエリー DSL を使用します。以下を参照してください。

Lucene プログラム API または Lucene クエリーパーサーのいずれを使用している場合も、特定のエンティティーのスコープ分析を取得することができます。スコープ指定のアナライザーは、インデックス化されたフィールドに応じて適切なアナライザーを適用するアナライザーです。複数のアナライザーを特定のエンティティーに定義でき、それぞれが個別のフィールドで作業することに注意してください。スコープ指定のアナライザーは、すべてのアナライザーをコンテキスト認識のアナライザーに統合します。理論はビットが複雑であるように見えますが、クエリーで正しいアナライザーを使用することは非常に簡単です。

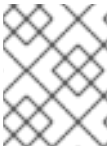
注記

子エンティティーにプログラムによるマッピングを使用する場合は、子エンティティーによって定義されるフィールドのみを表示できます。親エンティティーから継承されたフィールドまたはメソッド (@MappedSuperclass でアノテーション付け) は設定できません。親エンティティーから継承したプロパティーを設定するには、子エンティティーのプロパティーを上書きするか、親エンティティーのプログラムによるマッピングを作成します。これは、子エンティティーで定義されていない場合に親エンティティーのフィールドとメソッドにアノテーションを付けることができないアノテーションの使用に影響します。

例: 完全テキストクエリーの構築時のスコープ付きテナントの使用

```
org.apache.lucene.queryParser.QueryParser parser = new QueryParser(
    "title",
    fullTextSession.getSearchFactory().getAnalyzer( Song.class )
);

org.apache.lucene.search.Query luceneQuery =
    parser.parse( "title:sky Or title_stemmed:diamond" );

org.hibernate.Query fullTextQuery =
    fullTextSession.createFullTextQuery( luceneQuery, Song.class );

List result = fullTextQuery.list(); //return a list of managed objects
```

上記の例では、song タイトルが 2 つのフィールドでインデックス化されています。**title** フィールドでは標準アナライザーが使用され、**title_stemmed** フィールドには、スチーミングアナライザーが使用されます。検索ファクトリーによって提供されるアナライザーを使用すると、クエリーはターゲットフィールドに応じて適切なアナライザーを使用します。



注記

searchFactory.getAnalyzer(String) を使用して定義名で @AnalyzerDef を介して定義したアナライザーを取得することもできます。

7.4.4. ブリッジ

エンティティーの基本的なマッピングを検討する際に、最も重要なファクトは無視されていました。Lucene では、すべてのインデックスフィールドは文字列として表現する必要があります。**@Field** アノテーションが付けられたエンティティープロパティーはすべて、インデックス化される文字列に変換する必要があります。これまでに言及していない理由は、Hibernate Search のほとんどのプロパティーでは、組み込みブリッジのセットにより翻訳ジョブが実行されるためです。ただし、場合によっては、翻訳プロセスをより細かく制御する必要があります。

7.4.4.1. ビルトインブリッジ

Hibernate Search には、Java プロパティータイプとその完全なテキスト表現間の組み込みブリッジのセットがバンドルされています。

null

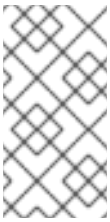
デフォルトの **null** 要素ごとにインデックス化されません。Lucene は null 要素をサポートしません。ただし、状況によっては、**null** 値を表すカスタムトークンを追加すると便利です。詳細はを参照してください。

java.lang.String

文字列は short、Short、integer、Integer、long、Long、float、Float、double としてインデックス化されます。

Double、BigInteger、BigDecimal

数値は文字列表現に変換されます。数値は Lucene (範囲指定のクエリーで使用される) によって追加されず、パディングする必要があることに注意してください。



注記

Range クエリーの使用には欠点があります。別の方法は、結果クエリーを適切な範囲に対してフィルターするフィルタークエリーを使用することです。Hibernate Search は、カスタム Custom Bridges で説明されているように、カスタム StringBridge を使用できます。

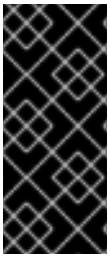
java.util.Date

日付は yyyyMMddHHmmssSSS として GMT 時間 (EST 2006 年 11 月 7 日 4:03 PM 12 秒の場合は 200611072203012) として保存されます。内部形式にこだわる必要はありません。TermRangeQuery を使用する場合は、グリニッジ標準時 (GMT) で日付を示す必要があることを把握しておくことが重要です。
通常は、ミリ秒単位までの日付を保存する必要はありません。**@DateBridge** は、インデックス に保存する適切な分解能を定義します ((**@DateBridge(resolution=Resolution.DAY)**))。日付のパターンは、それに応じて切り捨てられます。

```
@Entity
@Indexed
public class Meeting {
   @Field(analyze=Analyze.NO)

   private Date date;
   ...
```

警告

分解能が **MILLISECOND** 未満の日付は、**@DocumentId** にはできません。



重要

デフォルトの Date ブリッジは Lucene の DateTools を使用して、文字列から文字列に変換します。これは、すべての日付が GMT 時間で表されることを意味します。固定タイムゾーンに日付を保存する必要がある場合は、カスタムの日付ブリッジを実装する必要があります。日付のインデックス作成および検索に関するアプリケーションの要件を理解している。

java.net.URI、java.net.URL

URI および URL は文字列表現に変換されます。

java.lang.Class

クラスは完全修飾クラス名に変換されます。スレッドコンテキストクラスローダーは、クラスがリハイドレートされる際に使用されます。

7.4.4.2. カスタムブリッジ

Hibernate Search の組み込みブリッジは一部のプロパティータイプに対応していない場合や、ブリッジが使用する String 表現が要件を満たさない場合があります。以下では、この問題に対する複数の解決策を説明します。

7.4.4.2.1. StringBridge

最も単純なカスタムソリューションは、Hibernate Search に予想される Object を String ブリッジに実装を提供することです。これを実行するには、**org.hibernate.search.bridge.StringBridge** インターフェイスを実装する必要があります。すべての実装は同時に使用されるため、スレッドセーフである必要があります。

例: カスタム StringBridge 実装

```
/**
 * Padding Integer bridge.
 * All numbers will be padded with 0 to match 5 digits
 *
 * @author Emmanuel Bernard
 */
public class PaddedIntegerBridge implements StringBridge {

    private int PADDING = 5;

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > PADDING)
            throw new IllegalArgumentException( "Try to pad on a number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < PADDING ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}
```

前の例で定義された文字列ブリッジでは、**@FieldBridge** アノテーションにより、どのプロパティーまたはフィールドもこのブリッジを使用できます。

```
@FieldBridge(impl = PaddedIntegerBridge.class)
private Integer length;
```

7.4.4.2.2. パラメーター化されたブリッジ

また、パラメーターをブリッジ実装に渡すと柔軟性が向上します。以下の例は ParameterizedBridge インターフェイスを実装し、パラメーターは **@FieldBridge** アノテーションを介して渡されます。

例: ブリッジ実装にパラメーターを渡す

```
public class PaddedIntegerBridge implements StringBridge, ParameterizedBridge {
```

```
    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map<String,String> parameters) {
        String padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = Integer.parseInt( padding );
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < padding ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}


//property
@FieldBridge(impl = PaddedIntegerBridge.class,
             params = @Parameter(name="padding", value="10")
            )
private Integer length;
```

ParameterizedBridge インターフェイスは、**StringBridge**、**TwoWayStringBridge**、**FieldBridge** 実装で実装できます。

すべての実装はスレッドセーフである必要がありますが、パラメーターは初期化時に設定されるため、この段階で特別な注意をする必要はありません。

7.4.4.2.3. Type Aware Bridge

これは、ブリッジが適用されるタイプを取得すると便利です。

- field/getter-level ブリッジのプロパティーの戻りタイプ。
- クラスレベルのブリッジのクラスターイプ。

例として、カスタム方式で列挙に対応するブリッジがありますが、実際の列挙型にアクセスする必要があります。AppliedOnTypeAwareBridge を実装するブリッジは、挿入時にブリッジが適用されるタイプを取得します。パラメーターと同様に、インジェクトされるタイプには、スレッドセーフティーに関する特別な作業は必要ありません。

7.4.4.2.4. Two-Way Bridge

id プロパティー (**@DocumentId** アノテーション付き) でブリッジ実装を使用する必要がある場合は、TwoWayStringBridge という名前の拡張バージョン **StringBridge** を使用する必要があります。Hibernate Search は、識別子の文字列表現を読み取り、そこからオブジェクトを生成する必要があります。**@FieldBridge** アノテーションの使用方法には違いがありません。

例: id プロパティーに使用できる TwoWayStringBridge の実装

■

```
public class PaddedIntegerBridge implements TwoWayStringBridge, ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < padding ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }

    public Object stringToObject(String stringValue) {
        return new Integer(stringValue);
    }
}

//id property
@DocumentId
@FieldBridge(impl = PaddedIntegerBridge.class,
             params = @Parameter(name="padding", value="10")
private Integer id;
```

重要

双方向プロセスがべき等 (例: object = stringToObject(objectToString(object))) であることが重要です。

7.4.4.2.5. FieldBridge

場合によっては、プロパティーを Lucene インデックスにマッピングするときに、単純なオブジェクトから文字列への変換が必要になります。可能な限り柔軟性を持たせるために、FieldBridge としてブリッジを実装することもできます。このインターフェイスは、プロパティーの値を提供し、Lucene ドキュメント内で希望する方法でマッピングできるようにします。たとえば、プロパティーを異なるドキュメントフィールドに保存できます。インターフェイスの概念は Hibernate UserTypes と非常に似ています。

例: FieldBridge インターフェイスの実装

```
/**
 * Store the date in 3 different fields - year, month, day - to ease Range Query per
 * year, month or day (eg get all the elements of December for the last 5 years).
 * @author Emmanuel Bernard
 */
public class DateSplitBridge implements FieldBridge {
    private final static TimeZone GMT = TimeZone.getTimeZone("GMT");
```

```
    public void set(String name, Object value, Document document, LuceneOptions luceneOptions) {
        Date date = (Date) value;
        Calendar cal = GregorianCalendar.getInstance(GMT);
        cal.setTime(date);
        int year = cal.get(Calendar.YEAR);
        int month = cal.get(Calendar.MONTH) + 1;
        int day = cal.get(Calendar.DAY_OF_MONTH);

        // set year
        luceneOptions.addFieldToDocument(
            name + ".year",
            String.valueOf( year ),
            document );

        // set month and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".month",
            month < 10 ? "0" : "" + String.valueOf( month ),
            document );

        // set day and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".day",
            day < 10 ? "0" : "" + String.valueOf( day ),
            document );
    }
}

//property
@FieldBridge(impl = DateSplitBridge.class)
private Date date;
```

上記の例では、フィールドはドキュメントに直接追加されません。代わりに、追加が LuceneOptions ヘルパーシステムに委任されます。このヘルパーは、**Store** や **TermVector** などの **@Field** で選択したオプションを適用するか、選択した @Boost 値を適用します。**COMPRESS** 実装の複雑性をカプセル化することは特に便利です。ドキュメントにフィールドを追加するために LuceneOptions に委譲することが推奨されますが、必要であれば、ドキュメントを直接編集して、LuceneOptions を無視しても問題ありません。

注記

LuceneOptions などのクラスは、アプリケーションを Lucene API の変更から保護し、コードを簡素化するために作成されます。可能な場合はこれを使用しますが、さらに柔軟性が必要な場合は、使用しなくても問題ありません。

7.4.4.2.6. ClassBridge

特定のエンティティーの複数のプロパティーを組み合わせて、これを特定の方法で Lucene インデックスにインデックスを設定すると便利です。**@ClassBridge** および **@ClassBridges** アノテーションは、プロパティーレベルではなくクラスレベルで定義できます。この場合、カスタムフィールドブリッジ実装は、特定のプロパティーの代わりに、エンティティーインスタンスを値のパラメーターとして受信します。以下の例では示していませんが、**@ClassBridge** は Basic Mapping セクションで説明されている **termVector** 属性をサポートします。

例: クラスブリッジの実装

```
@Entity
@Indexed
(name="branchnetwork",
            store=Store.YES,
            impl = CatFieldsClassBridge.class,
            params = @Parameter( name="sepChar", value=" " ) )
public class Department {
   private int id;
   private String network;
   private String branchHead;
   private String branch;
   private Integer maxEmployees
   ...
}

public class CatFieldsClassBridge implements FieldBridge, ParameterizedBridge {
   private String sepChar;

   public void setParameterValues(Map parameters) {
      this.sepChar = (String) parameters.get( "sepChar" );
   }

   public void set( String name, Object value, Document document, LuceneOptions luceneOptions) {
      // In this particular class the name of the new field was passed
      // from the name field of the ClassBridge Annotation. This is not
      // a requirement. It just works that way in this instance. The
      // actual name could be supplied by hard coding it below.
      Department dep = (Department) value;
      String fieldValue1 = dep.getBranch();
      if ( fieldValue1 == null ) {
         fieldValue1 = "";
      }
      String fieldValue2 = dep.getNetwork();
      if ( fieldValue2 == null ) {
         fieldValue2 = "";
      }
      String fieldValue = fieldValue1 + sepChar + fieldValue2;
      Field field = new Field( name, fieldValue, luceneOptions.getStore(),
         luceneOptions.getIndex(), luceneOptions.getTermVector() );
      field.setBoost( luceneOptions.getBoost() );
      document.add( field );
   }
}
```

この例では、特定の **CatMissionClassBridge** が **department** インスタンスに適用され、フィールドブリッジはブランチとネットワークの両方を連結し、この連結をインデックス化します。

7.5. HIBERNATE SEARCH を使用した LUCENE クエリーの実行

手順

Hibernate Search は Lucene クエリーを実行し、InfinispanHibernate セッションによって管理されるドメインオブジェクトを取得できます。この検索は、Hibernate パラダイムを離れることなく Lucene の

機能を提供し、HQL、基準クエリー、ネイティブ SQL クエリーなど、Hibernate の従来の検索メカニズムに別の特性を与えます。

クエリーの準備と実行は、以下の 4 つの手順で設定されます。

- FullTextSession の作成
- Hibernate QueryHibernate Search クエリー DSL (推奨) または Lucene Query API を使用した Lucene クエリーの作成
- org.hibernate.Query を使用した Lucene クエリーのラップ
- サンプルの list() または scroll() を呼び出して検索を実行すると、検索が実行されます。

クエリー機能にアクセスするには、FullTextSession を使用します。この検索固有のセッションは、クエリーおよびインデックス機能を提供するために通常の org.hibernate.Session をラップします。

例: FullTextSession の作成

```
Session session = sessionFactory.openSession();
...
FullTextSession fullTextSession = Search.getFullTextSession(session);
```

FullTextSession を使用して、Hibernate Search クエリー DSL またはネイティブの Lucene クエリーのいずれかでフルテキストクエリーを構築します。

Hibernate Search クエリー DSL を使用する場合は、以下のコードを使用します。

```
final QueryBuilder b = fullTextSession.getSearchFactory().buildQueryBuilder().forEntity( Myth.class
).get();

org.apache.lucene.search.Query luceneQuery =
   b.keyword()
      .onField("history").boostedTo(3)
      .matching("storm")
      .createQuery();

org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery );
List result = fullTextQuery.list(); //return a list of managed objects
```

または、Lucene クエリーパーサーまたは Lucene プログラム API を使用して Lucene クエリーを書き込みます。

例: QueryParser を使用した Lucene クエリーの作成

```
SearchFactory searchFactory = fullTextSession.getSearchFactory();
org.apache.lucene.queryParser.QueryParser parser =
   new QueryParser("title", searchFactory.getAnalyzer(Myth.class) );
try {
   org.apache.lucene.search.Query luceneQuery = parser.parse( "history:storm^3" );
}
catch (ParseException e) {
   //handle parsing failure
}
```

```
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(luceneQuery);
List result = fullTextQuery.list(); //return a list of managed objects
```

Lucene クエリーに構築された Hibernate クエリーは org.hibernate.Query です。このクエリーは、HQL (Hibernate Query Language)、Native、および Criteria などの他の Hibernate クエリー機能と同じパラダイムに残ります。クエリーで list()、uniqueResult()、repeat()、および scroll () などのメソッドを使用します。

Hibernate Jakarta Persistence では、同じ拡張機能を利用できます。

例: Jakarta Persistence を使用した検索クエリーの作成

```
EntityManager em = entityManagerFactory.createEntityManager();

FullTextEntityManager fullTextEntityManager =
   org.hibernate.search.jpa.Search.getFullTextEntityManager(em);

...
final QueryBuilder b = fullTextEntityManager.getSearchFactory()
   .buildQueryBuilder().forEntity( Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
   b.keyword()
      .onField("history").boostedTo(3)
      .matching("storm")
      .createQuery();

javax.persistence.Query fullTextQuery = fullTextEntityManager.createFullTextQuery( luceneQuery );

List result = fullTextQuery.getResultList(); //return a list of managed objects
```

注記

これらの例では、Hibernate API が使われています。**FullTextQuery** の取得方法を調整すると、同じ例を Jakarta Persistence で記述することもできます。

7.5.1. クエリーの構築

Hibernate Search クエリーは Lucene クエリーにビルドされるため、ユーザーはすべての Lucene クエリータイプを使用できます。クエリーを構築すると、Hibernate Search はクエリー操作 API として org.hibernate.Query を使用して追加のクエリー処理を行います。

7.5.1.1. Lucene API を使用した Lucene クエリーの構築

Lucene API では、クエリーパーサー (簡単なクエリー) または Lucene プログラム API (複雑なクエリー) のいずれかを使用します。Lucene クエリーの構築は、Hibernate Search ドキュメントの範囲外です。詳細は、オンラインの Lucene ドキュメント、または**Lucene in Action** または **Hibernate Search in Action** を参照してください。

7.5.1.2. Lucene クエリーの構築

Lucene プログラム API は、フルテキストクエリーを有効にします。ただし、Lucene プログラム API を使用する場合は、パラメーターを同等の文字列に変換する必要があります。また、正しいアナライ

ザーを正しいフィールドに適用する必要もあります。たとえば、ngram アナライザーは、特定の単語に対するトークンとして複数の ngrams を使用するため、そのように検索する必要があります。このタスクには QueryBuilder を使用することが推奨されます。

Hibernate Search のクエリー API は変動しており、以下の主要な特徴があります。

- メソッド名は英語です。そのため、API 操作は、一連の英語のフレーズおよび命令として読み取り、理解することができます。
- IDE オートコンプリートを使用します。これは、現在の入力接頭辞の完了を容易にし、ユーザーが適切なオプションを選択できるようにします。
- 多くの場合、チェーンメソッドパターンを使用します。
- API 操作を簡単に使用でき、読み取ることができます。

API を使用するには、まず、指定の **indexedentitytype** に割り当てられるクエリービルダーを作成します。この QueryBuilder は、使用するアナライザーと、適用するフィールドブリッジを認識します。複数の QueryBuilder (クエリーのルートに関連するエンティティータイプごとに 1 つ) を作成できます。QueryBuilder は SearchFactory から派生します。

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder().forEntity( Myth.class ).get();
```

特定のフィールドに使用されるアナライザーも上書きできます。

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder()
    .forEntity( Myth.class )
        .overridesForField("history","stem_analyzer_definition")
    .get();
```

クエリービルダーを使用して Lucene クエリーをビルドできるようになりました。Lucene プログラム API を使用してアセンブルされた Lucene のクエリーパーサーまたはクエリーオブジェクトを使用して生成されたカスタマイズされたクエリーは、Hibernate Search DSL とともに使用されます。

7.5.1.3. キーワードのクエリー

以下の例は、特定の単語を検索する方法を示しています。

```
Query luceneQuery = mythQB.keyword().onField("history").matching("storm").createQuery();
```

表7.10 キーワードクエリーパラメーター

Parameter	説明
keyword()	特定の単語を検索するには、このパラメーターを使用します。
onField()	このパラメーターを使用して、単語を検索する lucene フィールドを指定します。
matching()	このパラメーターを使用して、検索文字列の一致を指定します。
createQuery()	Lucene クエリーオブジェクトを作成します。

- storm という値が **history** FieldBridge から渡されます。これは、数値または日付が必要な場合に便利です。
- フィールドブリッジの値は、**history** フィールドインデックス化に使用されるアナライザーに渡されます。これにより、クエリーはインデックス (小文字、ngram、スチミングなど) よりも、同じ用語変換を使用します。分析プロセスで指定の単語が複数生成されると、ブールクエリーが **SHOULD** 論理 (およびそ **OR** 論理) とともに使用されます。

タイプ文字列ではないプロパティーを検索します。

```
@Indexed
public class Myth {
  @Field(analyze = Analyze.NO)
  @DateBridge(resolution = Resolution.YEAR)
  public Date getCreationDate() { return creationDate; }
  public Date setCreationDate(Date creationDate) { this.creationDate = creationDate; }
  private Date creationDate;

  ...
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword().onField("creationDate").matching(birthdate).createQuery();
```

注記

プレーンの Lucene では、Date オブジェクトは文字列表現 (この場合は年) に変換する必要がありました。

この変換は、FieldBridge に objectToString メソッド (およびすべての組み込み FieldBridge 実装) がある場合、すべてのオブジェクトに対して機能します。

以下の例では、ngram アナライザーを使用するフィールドを検索します。ngram アナライザーは単語の ngrams インデックスを連続させるため、ユーザーの誤字を防ぎます。たとえば、hibernate という単語の 3 グラムは hib、ibe、ber、ern、rna、nat、ate です。

```
@AnalyzerDef(name = "ngram",
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class ),
  filters = {
    @TokenFilterDef(factory = StandardFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class),
    @TokenFilterDef(factory = NGramFilterFactory.class,
      params = {
        @Parameter(name = "minGramSize", value = "3"),
        @Parameter(name = "maxGramSize", value = "3") } )
  }
)

public class Myth {
  @Field(analyzer=@Analyzer(definition="ngram")
  public String getName() { return name; }
  public String setName(String name) { this.name = name; }
  private String name;
```

```
  ...
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword().onField("name").matching("Sisiphus")
   .createQuery();
```

一致する単語 Sisiphus は小文字になり、3 つのグラム (sis、isi、sip、iph、phu、hus) に分けられます。これらの各 ngram はクエリーの一部になります。ユーザーは、Sysiphus myth (**y**) を見つけることができます。ユーザーに透過的に実行されます。

注記

ユーザーが特定のフィールドでフィールドブリッジまたはアナライザーを使用しない場合は、ignoreAnalyzer() 関数または ignoreFieldBridge() 関数を呼び出しできます。

同一フィールドで複数の使用可能な単語を検索するには、それらすべてを一致するシーケンスに追加します。

```
//search document with storm or lightning in their history
Query luceneQuery =
   mythQB.keyword().onField("history").matching("storm lightning").createQuery();
```

複数のフィールドで同じ単語を検索するには、onField メソッドを使用します。

```
Query luceneQuery = mythQB
   .keyword()
   .onFields("history","description","name")
   .matching("storm")
   .createQuery();
```

場合によっては、同じ用語を検索する場合でも、あるフィールドを別のフィールドとは異なる方法で処理する必要があります。その場合は、andField() メソッドを使用します。

```
Query luceneQuery = mythQB.keyword()
   .onField("history")
   .andField("name")
     .boostedTo(5)
   .andField("description")
   .matching("storm")
   .createQuery();
```

上記の例では、フィールド名のみが 5 に改善されています。

7.5.1.4. Fuzzy クエリー

Levenshtein 距離アルゴリズムに基づく fuzzy クエリーを実行するには、**keyword** クエリーから始め、**fuzzy** フラグを追加します。

```
Query luceneQuery = mythQB
   .keyword()
     .fuzzy()
       .withThreshold( .8f )
```

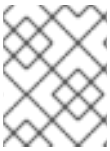
```
        .withPrefixLength( 1 )
    .onField("history")
    .matching("starm")
    .createQuery();
```

threshold は、両方の用語で照合が考慮される制限です。0 から 1 までの小数で、デフォルト値は 0.5 です。**prefixLength** は、fuzzyness で無視される接頭辞の長さです。デフォルト値は 0 ですが、多数の異なる用語を含むインデックスにはゼロ以外の値が推奨されます。

7.5.1.5. ワイルドカードクエリー

ワイルドカードクエリーは、単語の一部のみが認識される状況で役に立ちます。**?** は単一文字で、* は複数文字を表します。パフォーマンス維持のために、クエリーは **?** または * で開始しないことが推奨されます。

```
Query luceneQuery = mythQB
    .keyword()
      .wildcard()
    .onField("history")
    .matching("sto*")
    .createQuery();
```



注記

ワイルドカードクエリーは、一致する用語にアナライザーを適用しません。経験のある * または **?** のリスクが高すぎます。

7.5.1.6. フレーズクエリー

これまで、単語または単語セットを見てきましたが、ユーザーは正確な単語または概算した単語を検索することもできます。これを行うには、phrase() を使用します。

```
Query luceneQuery = mythQB
    .phrase()
    .onField("history")
    .sentence("Thou shalt not kill")
    .createQuery();
```

おおよその文は、slop 係数を追加することで検索できます。slop 係数は、文内で許可される他の単語の数を表します。これは、within または near 演算子のように機能します。

```
Query luceneQuery = mythQB
    .phrase()
      .withSlop(3)
    .onField("history")
    .sentence("Thou kill")
    .createQuery();
```

7.5.1.7. 範囲クエリー

範囲クエリーは、指定された範囲内 (含まれるかどうか) または特定の範囲を下回る、もしくは上回る値を検索します。

```
//look for 0 <= starred < 3
Query luceneQuery = mythQB
    .range()
    .onField("starred")
    .from(0).to(3).excludeLimit()
    .createQuery();

//look for myths strictly BC
Date beforeChrist = ...;
Query luceneQuery = mythQB
    .range()
    .onField("creationDate")
    .below(beforeChrist).excludeLimit()
    .createQuery();
```

7.5.1.8. クエリーの統合

クエリーを組み合わせてより複雑なクエリーを作成できます。以下の集計演算子を使用できます。

- **SHOULD**: クエリーには、サブクエリーの一致する要素が含まれる必要があります。
- **MUST**: クエリーには、サブクエリーの一致する要素が含まれる必要があります。
- **MUST NOT**: クエリーには、サブクエリーの一致する要素を含めないでください。

サブクエリーは、ブール値クエリー自体を含む任意の Lucene クエリーにすることができます。

例: SHOULD クエリー

```
//look for popular myths that are preferably urban
Query luceneQuery = mythQB
    .bool()
      .should( mythQB.keyword().onField("description").matching("urban").createQuery() )
      .must( mythQB.range().onField("starred").above(4).createQuery() )
    .createQuery();
```

例: MUST クエリー

```
//look for popular urban myths
Query luceneQuery = mythQB
    .bool()
      .must( mythQB.keyword().onField("description").matching("urban").createQuery() )
      .must( mythQB.range().onField("starred").above(4).createQuery() )
    .createQuery();
```

例: MUST NOT クエリー

```
//look for popular modern myths that are not urban
Date twentiethCentury = ...;
Query luceneQuery = mythQB
    .bool()
      .must( mythQB.keyword().onField("description").matching("urban").createQuery() )
        .not()
      .must( mythQB.range().onField("starred").above(4).createQuery() )
```

```
      .must( mythQB
        .range()
        .onField("creationDate")
        .above(twentiethCentury)
        .createQuery() )
    .createQuery();
```

7.5.1.9. クエリーオプション

Hibernate Search クエリー DSL は使いやすく、読みやすいクエリー API です。Lucene クエリーを受け入れて生成すると、DSL で対応していないクエリータイプを組み込むことができます。

以下は、クエリータイプおよびフィールドのクエリーオプションの要約です。

- **boostedTo** (クエリータイプおよびフィールド上) は、クエリー全体または特定のフィールドを指定された係数に改善します。
- **constantScore** (クエリー上) では、クエリーに一致するすべての結果の、定数スコアがブーストと等しくなります。
- **filteredBy(Filter)** (クエリー上) は、Filter インスタンスを使用してクエリー結果をフィルターします。
- **ignoreAnalyzer** (フィールド上) は、このフィールドを処理するときにアナライザーを無視します。
- **ignoreFieldBridge** (フィールド上) は、このフィールドを処理するときにフィールドブリッジを無視します。

例: クエリーオプションの組み合わせ

```
Query luceneQuery = mythQB
    .bool()
      .should( mythQB.keyword().onField("description").matching("urban").createQuery() )
      .should( mythQB
        .keyword()
        .onField("name")
          .boostedTo(3)
          .ignoreAnalyzer()
        .matching("urban").createQuery() )
      .must( mythQB
        .range()
          .boostedTo(5).withConstantScore()
        .onField("starred").above(4).createQuery() )
    .createQuery();
```

7.5.1.10. Hibernate Search クエリーの構築

7.5.1.10.1. 一般性

Lucene クエリーの構築後に、Hibernate クエリー内にラップします。クエリーはインデックス化されたすべてのエンティティーを検索し、明示的に設定しない限り、インデックス化されたクラスのすべてのタイプを返します。

例: Hibernate クエリーでの Lucene クエリーのラップ

```
FullTextSession fullTextSession = Search.getFullTextSession( session );
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery );
```

パフォーマンスを改善するために、返されたタイプを以下のように制限します。

例: エンティティータイプによる検索結果のフィルター

```
fullTextQuery = fullTextSession
    .createFullTextQuery( luceneQuery, Customer.class );

// or

fullTextQuery = fullTextSession
    .createFullTextQuery( luceneQuery, Item.class, Actor.class );
```

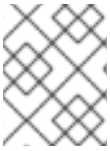
次の例の最初の部分は、一致する Customers のみを返します。同じ例の次の部分は、一致する Actors と item を返します。タイプ制限はポリモーフィックです。そのため、2 つのサブクラス Sales man と、ベースクラス Person の Customer は、結果タイプに基づいてフィルタリングする Person.class を指定します。

7.5.1.10.2. ページネーション

パフォーマンスの低下を回避するには、クエリーごとに返されたオブジェクトの数を制限することが推奨されます。あるページから別のページに移動するユーザーは、非常に一般的なユースケースです。ページネーションを定義する方法は、プレーンの HQL または基準クエリーでのページネーションの定義と似ています。

例: 検索クエリーのページネーションの定義

```
org.hibernate.Query fullTextQuery =
    fullTextSession.createFullTextQuery( luceneQuery, Customer.class );
fullTextQuery.setFirstResult(15); //start from the 15th element
fullTextQuery.setMaxResults(10); //return 10 elements
```



注記

fulltextQuery.getResultSize() によるページネーションに関係なく、一致する要素の合計数を取得できます。

7.5.1.10.3. ソート

Apache Lucene には、柔軟で強力な結果ソートメカニズムが含まれています。デフォルトの並び替えは関連により行われ、さまざまなユースケースに適しています。Lucene Sort オブジェクトを使用し、他のプロパティーでソートするようにソートメカニズムを変更できます。

例: Lucene ソートの指定

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( query, Book.class );
org.apache.lucene.search.Sort sort = new Sort(
    new SortField("title", SortField.STRING));

List results = query.list();
```

注記

ソートに使用されるフィールドは、トークン化できません。トークン化についての詳細は、@Field を参照してください 。

7.5.1.10.4. ストラテジーの取得

Hibernate Search は、戻り値の型が単一のクラスに制限されている場合に、単一のクエリーを使用してオブジェクトをロードします。Hibernate Search は、ドメインモデルで定義された静的なフェッチストラテジーによって制限されます。以下のように、特定のユースケースのフェッチストラテジーを微調整すると便利です。

例: クエリーでの FetchMode の指定

```
Criteria criteria =
    s.createCriteria( Book.class ).setFetchMode( "authors", FetchMode.JOIN );
s.createFullTextQuery( luceneQuery ).setCriteriaQuery( criteria );
```

この例では、クエリーは LuceneQuery に一致するすべての Books を返します。Authors コレクションは、SQL の外部結合を使用して同じクエリーからロードされます。

条件クエリー定義では、タイプは提供された基準クエリーに基づいて推測されます。そのため、返されたエンティティータイプを制限する必要はありません。

重要

フェッチモードは、唯一の調整可能なプロパティーです。getResultSize() は制限のある Criteria とともに使用された場合に SearchException を出力するため、Criteria クエリーで制限 (完全な句) を使用しないでください。

複数のエンティティーが予想される場合は、**setGatewayQuery** を使用しないでください。

7.5.1.10.5. プロジェクション

場合によっては、プロパティーの小さなサブセットのみが必要となります。Hibernate Search を使用して、以下のようにプロパティーのサブセットを取得します。

Hibernate Search は Lucene インデックスからプロパティーを抽出し、それらをオブジェクト表現に変換し、Object[] のリストを返します。プロジェクションは、データベースのラウンドトリップが長くなるのを防ぎます。ただし、以下の制限があります。

- 予測されるプロパティーはインデックス (**@Field(store=Store.YES)**) に保存され、インデックスサイズが増えます。
- 予測されるプロパティーは、org.hibernate.search.bridge.TwoWayFieldBridge または **org.hibernate.search.bridge.TwoWayStringBridge** を実装する **FieldBridge** を使用する必要があり、後者はより単純なバージョンになります。

  注記

  Hibernate Search の組み込みタイプはすべて双方向です。

- インデックス化されたエンティティーまたはその埋め込み関連の簡単なプロパティーのみを展開できます。したがって、埋め込みエンティティー全体を展開できません。

- @IndexedEmbedded でインデックス化されるコレクションやマップで機能しません。

Lucene は、クエリー結果に関するメタデータ情報を提供します。インジェクト定数を使用してメタデータを取得します。

例: メタデータの取得へのプロジェクションの使用

```
org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
query.;
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
float score = firstResult[0];
Book book = firstResult[1];
String authorName = firstResult[2];
```

フィールドは、以下のプロジェクションと組み合わせることができます。

- **FullTextQuery.THIS:** 初期化され管理されたエンティティーを返します (展開されていないクエリーが実行される場合)。
- **FullTextQuery.DOCUMENT:** 展開されるオブジェクトに関連する Lucene ドキュメントを返します。
- **FullTextQuery.OBJECT_CLASS:** インデックス化されたエンティティーのクラスを返します。
- **FullTextQuery.SCORE:** クエリーのドキュメントスコアを返します。スコアは、あるクエリーの結果を別のクエリーに対する比較には便利ですが、異なるクエリーの結果を比較する場合に有用ではありません。
- **FullTextQuery.ID:** 予測されるオブジェクトの ID プロパティー値。
- **FullTextQuery.DOCUMENT_ID:** Lucene ドキュメント ID。この値を Lucene ドキュメント ID として使用すると、異なる 2 つの IndexReader を開くたびに変更される可能性があります。
- **FullTextQuery.explanation:** 指定のクエリーの一致するオブジェクト/ドキュメントの Lucene Explanation オブジェクトを返します。これは、大量のデータを取得するのには適していません。通常、実行の説明は、一致する要素ごとに Lucene クエリー全体を実行することを意味します。そのため、展開が推奨されます。

7.5.1.10.6. オブジェクト初期化ストラテジーのカスタマイズ

デフォルトでは、Hibernate Search は最適なストラテジーを使用して、完全なテキストクエリーに一致するエンティティーを初期化します。必要なエンティティーを取得するためにクエリーを実行します。このアプローチでは、取得したエンティティーが永続コンテキスト (セッション) または 2 次レベルキャッシュにほとんど存在しないデータベースのストライプが最小限に抑えられます。

2 次キャッシュにエンティティーが存在する場合は、データベースオブジェクトを取得する前に、Hibernate Search が強制的にキャッシュを調べます。

例: クエリーを使用する前の 2 次キャッシュのチェック

```
FullTextQuery query = session.createFullTextQuery(luceneQuery, User.class);
query.initializeObjectWith(
    ObjectLookupMethod.SECOND_LEVEL_CACHE,
```

```
    DatabaseRetrievalMethod.QUERY
);
```

ObjectLookupMethod は、オブジェクトをデータベースから取得せずに簡単にアクセスできるかどうかを確認するストラテジーを定義します。その他のオプションは以下のとおりです。

- **ObjectLookupMethod.PERSISTENCE_CONTEXT** は、一致する多くのエンティティーが永続コンテキストにすでにロードされている場合に使用されます (Session または EntityManager にロードされている場合)。
- **ObjectLookupMethod.SECOND_LEVEL_CACHE** は永続コンテキストをチェックし、2 次キャッシュを確認します。

2 次キャッシュで検索するには、以下を設定します。

- 2 次キャッシュを正しく設定およびアクティブ化します。
- 関連するエンティティーの 2 次キャッシュを有効にします。これは、@Cacheable などのアノテーションを使用してを行います。
- Session、EntityManager、または Query のいずれかの 2 次キャッシュ読み取りアクセスを有効にします。Hibernate ネイティブ API では **CacheMode.NORMAL** を使用し、Jakarta Persistence では **CacheRetrieveMode.USE** を使用します。

警告

2 次キャッシュ実装が Infinispan でない場合、ObjectLookupMethod.SECOND_LEVEL_CACHE は使用しないでください。他の 2 次レベルのキャッシュプロバイダーはこのオペレーションを効率的に実装しません。

DatabaseRetrievalMethod を使用して、以下のようにデータベースからオブジェクトを読み込む方法をカスタマイズします。

- **QUERY** (デフォルト) はクエリーのセットを使用して、複数のオブジェクトを各バッチに読み込みます。このアプローチが推奨されます。
- **find_BY_ID** は **Session.get** または **EntityManager.find** セマンティックを使用して一度にオブジェクトをロードします。これは、Hibernate Core がエンティティーをバッチでロードできるようにする、エンティティーにバッチサイズが設定されている場合に推奨されます。

7.5.1.10.7. クエリー時間の制限

Hibernate Guide で、以下のようにクエリーにかかる時間を制限します。

- 制限を指定して受信する際に例外を発生させます。
- 時間制限が発生したときに取得する結果の数に制限します。

7.5.1.10.8. 時間制限の例外発生

クエリー使用する時間が定義した時間を超える場合は、QueryTimeoutException が発生します (プログラム API に応じた、org.hibernate.QueryTimeoutException または javax.persistence.QueryTimeoutException)。

ネイティブの Hibernate API を使用する際に制限を定義するには、以下のいずれかの方法を使用します。

例: クエリー実行でのタイムアウトの定義

```
Query luceneQuery = ...;
FullTextQuery query = fullTextSession.createFullTextQuery(luceneQuery, User.class);

//define the timeout in seconds
query.setTimeout(5);

//alternatively, define the timeout in any given time unit
query.setTimeout(450, TimeUnit.MILLISECONDS);

try {
    query.list();
}
catch (org.hibernate.QueryTimeoutException e) {
    //do something, too slow
}
```

getResultSize()、iterate()、および scroll() は、メソッド呼び出しの終了までタイムアウトを受け入れます。その結果、Iterable または ScrollableResults は、タイムアウトを無視します。また、explain() はこのタイムアウト期間を受け入れません。この方法は、デバッグに使用され、クエリーのパフォーマンス低下の原因をチェックします。

以下は、Jakarta Persistence を使用した実行時間を制限する標準的な方法です。

例: クエリー実行でのタイムアウトの定義

```
Query luceneQuery = ...;
FullTextQuery query = fullTextEM.createFullTextQuery(luceneQuery, User.class);

//define the timeout in milliseconds
query.setHint( "javax.persistence.query.timeout", 450 );

try {
    query.getResultList();
}
catch (javax.persistence.QueryTimeoutException e) {
    //do something, too slow
}
```

重要

サンプルコードは、クエリーが指定の結果量で停止することを保証しません。

7.5.2. 結果の取得

Hibernate クエリーの構築後、HQL または Criteria クエリーと同じように実行されます。同じ準仮想化とオブジェクトセマンティックが Lucene クエリーに適用さ
れ、**list()**、**uniqueResult()**、**iterate()**、**scroll()** などの一般的な操作を使用できます。

7.5.2.1. パフォーマンスに関する考慮事項

妥当な数の結果 (たとえば、ページネーションの使用) が想定され、それらすべてで動作することが予想される場合は、**list()** または **uniqueResult()** が推奨されます。**list()** は、エンティティー **batch-size** が正しく設定されている場合に最適に機能します。**list()**、**uniqueResult()**、**iterate()** を使用する場合は、Hibernate Search が Lucene Hits 要素 (ページネーション内) をすべて処理する必要があることに注意してください。

Lucene ドキュメントの負荷を最小限に抑える必要がある場合には、**scroll()** の方が適しています。完了したら、Lucene リソースを保持するため、Scrollable Mission オブジェクトを閉じることを忘れないでください。スクロールを使用することが予測されても、オブジェクトを一括して読み込む必要がある場合は、query.**setFetchSize()** を使用できます。オブジェクトにアクセスし、読み込まれていない場合、Hibernate Search は次の **fetchSize** オブジェクトをパスに読み込みます。

重要

ページネーションが、スクロールよりも好まれます。

7.5.2.2. 結果サイズ

一致するドキュメントの合計数を把握しておくと役に立つ場合があります。

- Google 検索で提供された、全体的な検索結果機能を提供たとえば、"約 888,000,000 件のうちの 1-10 のようになります。
- 高速なページネーションナビゲーションを実装する
- クエリーがゼロを返すか、十分な結果がない場合に概算を追加する複数ステップの検索エンジンを実装するには、以下を実行します。

当然ながら、一致するドキュメントをすべて取得することはできません。Hibernate Search を使用すると、ページネーションパラメーターに関係なく、一致するドキュメントの合計数を取得できます。さらに注意深く、単一のオブジェクト負荷をトリガーせずに一致する要素の数を取得できます。

例: クエリーの結果サイズの決定

```
org.hibernate.search.FullTextQuery query =
   s.createFullTextQuery( luceneQuery, Book.class );
//return the number of matching books without loading a single one
assert 3245 == ;

org.hibernate.search.FullTextQuery query =
   s.createFullTextQuery( luceneQuery, Book.class );
query.setMaxResult(10);
List results = query.list();
//return the total number of matching books regardless of pagination
assert 3245 == ;
```

注記

Google と同様に、インデックスがデータベースと完全に更新されていない場合は、結果の数は概算されます (例: 非同期クラスター)。

7.5.2.3. ResultTransformer

プロジェクション結果はオブジェクト配列として返されます。オブジェクトに使用されるデータ構造がアプリケーションの要件と一致しない場合は、ResultTransformer を適用します。ResultTransformer は、クエリーの実行後に必要なデータ構造を構築します。

例: プロジェクトでの ResultTransformer の使用

```
org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
query.setProjection( "title", "mainAuthor.name" );

query.setResultTransformer( new StaticAliasToBeanResultTransformer( BookView.class, "title",
"author" ) );
List<BookView> results = (List<BookView>) query.list();
for(BookView view : results) {
    log.info( "Book: " + view.getTitle() + ", " + view.getAuthor() );
}
```

ResultTransformer 実装の例は、ibernate Core codebase を参照してください。

7.5.2.4. 結果について

クエリーの結果が適切でない場合、**Luke** ツールは結果を理解する際に役立ちます。ただし、Hibernate Search を使用すると、所定の結果 (特定のクエリー内) の Lucene Explanation オブジェクトにアクセスできます。このクラスは Lucene ユーザーに非常に高度なものとみなされますが、オブジェクトの性質をよく理解することができます。特定の結果について Explanation オブジェクトにアクセスするには、以下のいずれかの方法があります。

- **fullTextQuery.explain(int)** メソッドを使用します。
- プロジェクションの使用

最初の方法は、ドキュメント ID をパラメーターとして取り、Explanation オブジェクトを返します。ドキュメント ID は、インジェクトと **FullTextQuery.DOCUMENT_ID** 定数を使用して取得できます。

警告

ドキュメント ID はエンティティー ID に関連しません。これらの概念を混同しないように注意してください。

次の方法では、**FullTextQuery.EXPLANATION** 定数を使用して Explanation オブジェクトをプロジェクトします。

例: プロジェクトを使用した Lucene の説明オブジェクトの取得

```
FullTextQuery ftQuery = s.createFullTextQuery( luceneQuery, Dvd.class )
        .setProjection(
            FullTextQuery.DOCUMENT_ID,
            ,
            FullTextQuery.THIS );
@SuppressWarnings("unchecked") List<Object[]> results = ftQuery.list();
for (Object[] result : results) {
    Explanation e = (Explanation) result[1];
    display( e.toString() );
}
```

Explanation オブジェクトは、Lucene クエリーを再度実行するときのように、必要とされる場合にのみ使用してください。

7.5.2.5. フィルター

Apache Lucene には、カスタムフィルタープロセスに応じてクエリーの結果をフィルタリングできる強力な機能があります。これは、特にフィルターをキャッシュして再利用できるため、追加のデータ制限を適用する非常に強力な方法です。ユースケースには以下が含まれます。

- セキュリティー
- 一時的なデータ (例: 先月のデータのみ表示)
- 予測フィルター (例: 検索は所定カテゴリーに限定される)

Hibernate Search は、透過的にキャッシュされるパラメーター可能な名前付きフィルターの概念を導入することで、この概念をさらにプッシュします。Hibernate Core フィルターの概念を熟知しているユーザーにとって、API は非常に似ています。

例: クエリーのフルテキストフィルターの有効化

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("bestDriver");
fullTextQuery.enableFullTextFilter("security").setParameter( "login", "andre" );
fullTextQuery.list(); //returns only best drivers where andre has credentials
```

この例では、クエリーの上に複数のフィルターを有効化しています。フィルターはいくつでも有効または無効にできます。

宣言フィルターは @FullTextFilterDef アノテーションを使用して実行されます。このアノテーションは、フィルターが後に適用されるクエリーに関係なく、**@Indexed** エンティティーに設定できます。これは、フィルター定義がグローバルであり、名前が一意である必要があることを意味します。同じ名前を持つ @FullTextFilterDef アノテーションが定義される場合は、SearchException が発生します。名前付きの各フィルターは、実際のフィルター実装を指定する必要があります。

例: フィルターの定義および実装

```
@FullTextFilterDefs( {
    @FullTextFilterDef(name = "bestDriver", impl = BestDriversFilter.class),
    @FullTextFilterDef(name = "security", impl = SecurityFilterFactory.class)
})
public class Driver { ... }
```

```
public class BestDriversFilter extends org.apache.lucene.search.Filter {

    public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
        OpenBitSet bitSet = new OpenBitSet( reader.maxDoc() );
        TermDocs termDocs = reader.termDocs( new Term( "score", "5" ) );
        while ( termDocs.next() ) {
            bitSet.set( termDocs.doc() );
        }
        return bitSet;
    }
}
```

BestDriversFilter は簡単な Lucene フィルターの例です。これにより、スコアが 5 のドライバーに設定された結果が減少します。この例では、指定したフィルターは **org.apache.lucene.search.Filter** を直接実装し、no-arg コンストラクターが含まれます。

フィルターの作成に追加のステップが必要な場合や、使用するフィルターに no-arg コンストラクターがない場合は、ファクトリーパターンを使用できます。

例: ファクトリーパターンを使用したフィルターの作成

```
@FullTextFilterDef(name = "bestDriver", impl = BestDriversFilterFactory.class)
public class Driver { ... }

public class BestDriversFilterFactory {

@Factory
    public Filter getFilter() {
        //some additional steps to cache the filter results per IndexReader
        Filter bestDriversFilter = new BestDriversFilter();
        return new CachingWrapperFilter(bestDriversFilter);
    }
}
```

Hibernate Search は **@Factory** アノテーションが付けられたメソッドを検索し、そのメソッドを使用してフィルターインスタンスを作成します。ファクトリーには no-arg コンストラクターが必要です。

Infinispan Query は @Factory アノテーションが付けられたメソッドを使用してフィルターインスタンスを構築します。ファクトリーには引数コンストラクターは使えません。

名前付きフィルターを使用すると、パラメーターをフィルターに渡すことができます。たとえば、セキュリティーフィルターを適用すると、適用するセキュリティーレベルが分かります。

例: 定義されたフィルターにパラメーターを渡す

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("security").setParameter( "level", 5 );
```

各パラメーター名は、フィルターや、ターゲットとなる名前付きフィルター定義のフィルターまたはフィルターファクトリーのいずれかに関連するセッターを持つ必要があります。

例: フィルター実装におけるパラメーターの使用

```
public class SecurityFilterFactory {
```

```
    private Integer level;

    /**
     * injected parameter
     */
    public void setLevel(Integer level) {
        this.level = level;
    }

    @Key public FilterKey getKey() {
        StandardFilterKey key = new StandardFilterKey();
        key.addParameter( level );
        return key;
    }

    @Factory
    public Filter getFilter() {
        Query query = new TermQuery( new Term("level", level.toString() ) );
        return new CachingWrapperFilter( new QueryWrapperFilter(query) );
    }
}
```

@Key アノテーションが付けられたメソッドは FilterKey オブジェクトを返すことに注意してください。返されたオブジェクトには特別なコントラクトを持ちます。キーオブジェクトは equals() / hashCode() を実装し、指定される Filter タイプが同一で、パラメーターのセットが同じである場合にのみ、両方の鍵が同じになるようにする必要があります。つまり、鍵の生成元となるフィルターが交換可能な場合、あるいは交換可能である場合にのみ、両方のフィルター鍵が同等になります。キーオブジェクトは、キャッシュメカニズムのキーとして使用されます。

@Key メソッドは以下の場合にのみ必要です。

- フィルターキャッシングシステムが有効になっている (デフォルトでは有効)。
- フィルターにはパラメーターがあります。

多くの場合、**StandardFilterKey** 実装を使用すれば十分です。これは、equals() / hashCode() 実装をそれぞれのパラメーター equals および hashcode メソッドに委譲します。

定義されたフィルターがデフォルトのキャッシュされ、キャッシュは、必要に応じてハード参照とソフト参照の組み合わせを使用してメモリーの破損を可能にします。ハード参照キャッシュは、最近使用されたフィルターを追跡し、必要に応じて SoftReferences に最も使用されるフィルターを変換します。ハード参照キャッシュの制限に達すると、追加のフィルターが SoftReferences としてキャッシュされます。ハード参照キャッシュのサイズを調整するには、**hibernate.search.filter.cache_strategy.size** (デフォルトは 128 に設定) を使用します。フィルターキャッシングの高度な使用のために、独自の FilterCachingStrategy を実装します。classname は **hibernate.search.filter.cache_strategy** によって定義されます。

このフィルターキャッシュメカニズムは、実際のフィルター結果をキャッシュするのと混同しないようにしてください。Lucene では、CachingWrapperFilter を中心に IndexReader を使用してフィルターをラッピングすることが一般的です。ラッパーは、**getDocIdSet(IndexReader reader)** メソッドから返される DocIdSet をキャッシュして、高価な再構築を防ぎます。リーダーは、開いた時点のインデックスの状態を効果的に表示するため、計算した DocIdSet が同じ IndexReader インスタンスに対してのみキャッシュ可能であることを示すことが重要です。ドキュメントリストは、開いている IndexReader 内では変更できません。ただし、別の、新しい IndexReader インスタンスの場合は、(別のインデックスから、または単にインデックスが変更されたため) 異なるファイルのセットで動作する可能性があるため、キャッシュされた DocIdSet を再計算する必要があります。

また、Hibernate Search はキャッシングのこの側面にも役立ちます。@FullTextFilterDef の **cache** フラグは、デフォルトでは **FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS** に設定されています。これにより、フィルターインスタンスを自動的にキャッシュし、CachingWrapperFilter の Hibernate 固有の実装に指定されたフィルターをラップします。このクラスの Lucene のバージョンとは対照的に、SoftReferences はハード参照数とともに使用されます (フィルターキャッシュについて参照)。ハード参照数は、**hibernate.search.filter.cache_docidresults.size** (デフォルトは 5 に設定) を使用して調整できます。ラッピング動作は、**the@FullTextFilterDef.cache** パラメーターを使用して制御できます。このパラメーターには、以下の異なる値があります。

値	定義
FilterCacheModeType.NONE	フィルターインスタンスがなく、Hibernate Search によって結果がキャッシュされません。フィルター呼び出しごとに、新しいフィルターインスタンスが作成されます。この設定は、データセットまたはメモリーが制限される環境を迅速に変更する際に役に立つことがあります。
FilterCacheModeType.INSTANCE_ONLY	フィルターインスタンスは、同時に Filter.getDocIdSet() 呼び出しでキャッシュされ、再利用されます。DocIdSet の結果はキャッシュされません。この設定は、フィルターが固有のキャッシングメカニズムを使用するか、アプリケーション固有のイベントにより DocIdSet をキャッシュする必要がない場合に動的にフィルターの結果が変更される場合に役立ちます。
FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS	フィルターインスタンスと DocIdSet 両方の結果がキャッシュされます。これがデフォルト値になります。

フィルターは以下の状況でキャッシュする必要があります。

- システムはターゲットエンティティーインデックスを頻繁に更新しません (つまり、IndexReader は頻繁に再利用されます)。
- Filter の DocIdSet の計算には、クエリーの実行にかかった時間と比較して負荷がかかります。

7.5.2.6. シャード化された環境でのフィルターの使用

シャード化された環境では、利用可能なシャードのサブセットでクエリーを実行することができます。これを行う方法は 2 つあります。

インデックスシャードのサブセットのクエリー

1. フィルター設定に応じて IndexManager のサブセットを選択するシャード化ストラテジーを作成します。
2. クエリー時にフィルターをアクティブにします。

例: インデックスシャードのサブセットのクエリー

この例では、**customer** フィルターがアクティブな場合は、クエリーが特定の顧客シャードに対して実行されます。

```
public class CustomerShardingStrategy implements IndexShardingStrategy {
```

```
// stored IndexManagers in an array indexed by customerID
private IndexManager[] indexManagers;

public void initialize(Properties properties, IndexManager[] indexManagers) {
  this.indexManagers = indexManagers;
}

public IndexManager[] getIndexManagersForAllShards() {
  return indexManagers;
}

public IndexManager getIndexManagerForAddition(
    Class<?> entity, Serializable id, String idInString, Document document) {
  Integer customerID = Integer.parseInt(document.getFieldable("customerID").stringValue());
  return indexManagers[customerID];
}

public IndexManager[] getIndexManagersForDeletion(
    Class<?> entity, Serializable id, String idInString) {
  return getIndexManagersForAllShards();
}

 /**
 * Optimization; don't search ALL shards and union the results; in this case, we
 * can be certain that all the data for a particular customer Filter is in a single
 * shard; simply return that shard by customerID.
 */
public IndexManager[] getIndexManagersForQuery(
    FullTextFilterImplementor[] filters) {
  FullTextFilter filter = getCustomerFilter(filters, "customer");
  if (filter == null) {
    return getIndexManagersForAllShards();
  }
  else {
    return new IndexManager[] { indexManagers[Integer.parseInt(
      filter.getParameter("customerID").toString())] };
  }
}

private FullTextFilter getCustomerFilter(FullTextFilterImplementor[] filters, String name) {
  for (FullTextFilterImplementor filter: filters) {
    if (filter.getName().equals(name)) return filter;
  }
  return null;
}
}
```

この例では、**custom** という名前のフィルターがある場合、この顧客専用のシャードのみがクエリーされ、それ以外の場合はすべてのシャードが返されます。所定のシャード化ストラテジーは、単一または複数のフィルターに対応し、それらのパラメーターに依存します。

次のステップでは、クエリー時にフィルターをアクティブにします。フィルターは、クエリーの後に Lucene 結果をフィルターする (定義されている) 通常のフィルターですが、シャード化ストラテジーにのみ渡される特殊フィルターを使用することができます (その他は無視されます)。

この機能を使用するには、フィルターの宣言時に ShardSensitiveOnlyFilter クラスを指定します。

```
@Indexed
@FullTextFilterDef(name="customer", impl=ShardSensitiveOnlyFilter.class)
public class Customer {
   ...
}

FullTextQuery query = ftEm.createFullTextQuery(luceneQuery, Customer.class);
query.enableFulltextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")
List<Customer> results = query.getResultList();
```

ShardSensitiveOnlyFilter を使用して Lucene フィルターを実装する必要はありません。シャード化された環境のクエリーを加速するには、これらのフィルターに対応するフィルターおよびシャード化ストラテジーを使用することが推奨されます。

7.5.3. ファセット

ファセット検索は、クエリーの結果を複数のカテゴリーに分割できる技術です。この分類には、各カテゴリーのヒット数の計算や、これらのファイン (カテゴリー) に基づいて検索結果をさらに制限する機能が含まれます。以下の例は、ファセットの例を示しています。ページのメイン部分に表示される検索結果が表示されます。ただし、左側のナビゲーションバーには、**Programming**、**Computer Science**、**Databases**、**Software**、**Web Development**、**Networking**、**Home Computing** のサブカテゴリーを持つ **Computers & Internet** カテゴリーが表示されています。各サブカテゴリーについて、主要な検索条件に合致し、それぞれのサブカテゴリーに属する Book の数が表示されます。**Computers & Internet** のカテゴリーの区分は、特定の検索ファセットです。その他には、平均的なカスタマーレビューなどが挙げられます。

ファセット検索は、クエリーの結果をカテゴリーに分割します。この分類は、各カテゴリーのヒット数の計算を含み、これらのファセット (カテゴリー) に基づいて検索結果をさらに制限します。以下の例では、ファセット検索結果がメインページに表示されます。

左側のナビゲーションバーには、カテゴリーとサブカテゴリーが表示されます。各サブカテゴリーについて、Book の数は主要な検索条件と一致し、それぞれのサブカテゴリーに属します。この Computers & Internet カテゴリーの区分は、特定の検索ファセットです。もう 1 つの例は、平均的なカスタマーレビューです。

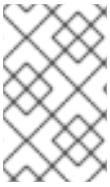
例: Amazon での Hibernate Search の検索

Hibernate Search の QueryBuilder クラスおよび FullTextQuery クラスは、ファセット API へのエントリーポイントです。前者は要求を作成し、後者は FacetManager にアクセスします。FacetManager はクエリーにファセット要求を適用し、検索結果を絞り込むために既存のクエリーに追加されるブックマークを選択します。この例では、以下の例のように Cd エンティティーを使用します。

例: エンティティー Cd

```
@Indexed
public class Cd {

    private int id;

    @Fields( {
        @Field,
        @Field(name = "name_un_analyzed", analyze = Analyze.NO)
    })
    private String name;

    @Field(analyze = Analyze.NO)
```

```
    @NumericField
    private int price;

    Field(analyze = Analyze.NO)
    @DateBridge(resolution = Resolution.YEAR)
    private Date releaseYear;

    @Field(analyze = Analyze.NO)
    private String label;


// setter/getter
...
```



注記

Hibernate Search 5.2 よりも前のバージョンでは、@Facet アノテーションを明示的に使用する必要がなくなりました。Hibernate Search 5.2 では、Lucene のネイティブファセッティング API を使用するために必要になりました。

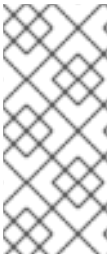
7.5.3.1. ファセット要求の作成

ファセット検索に対する最初のステップは、FacetingRequest を作成することです。現時点では、2 種類のセッティング要求がサポートされています。最初のタイプは **discrete faceting** と呼ばれ、次のタイプは **range faceting** 要求と呼ばれます。個別のファセットリクエストの場合は、ファセット (分類) を行うインデックスフィールドと、適用するファセットオプションを指定します。個別のファセッティング要求の例は、以下の例で確認できます。

例: 個別のファセット要求の作成

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
        .forEntity( Cd.class )
            .get();
FacetingRequest labelFacetingRequest = builder.facet()
    .name( "labelFaceting" )
    .onField( "label")
    .discrete()
    .orderedBy( FacetSortOrder.COUNT_DESC )
    .includeZeroCounts( false )
    .maxFacetCount( 1 )
    .createFacetingRequest();
```

このファセットリクエストを実行すると、インデックス設定された **label** フィールドの個別値ごとに Facet インスタンスが作成されます。Facet インスタンスは、元のクエリー結果内でこの特定のフィールドの値が発生する頻度を含む実際のフィールド値を記録します。orderdBy、includeZeroCounts および maxFacetCount は任意のオプションのパラメーターで、すべてのファセット要求に適用できます。ordersBy では、作成されたブックマークが返される順序を指定できます。デフォルトは **FacetSortOrder.COUNT_DESC** ですが、フィールドの値または範囲の指定順序でソートすることもできます。includeZeroCount は、結果内に含まれるブックマーク数 (デフォルトでは 0) を判断し、maxFacetCount により、返されるワイルドカードの最大数を制限できます。



注記

現時点では、ファセッティングを適用するためにインデックス化されたフィールドを満たす必要のあるいくつかの前提条件があります。インデックス付きプロパティーは String、Date、または Number および **null** の値を持つものは使用しないでください。さらに、プロパティーは **Analyze.NO** でインデックス化する必要があり、数値プロパティー @NumericField を指定する場合は指定する必要があります。

一定の範囲のファセットリクエストの作成は、ブックマークするフィールド値の範囲を指定する必要がある点以外は非常に似ています。一定の範囲のファセットリクエストは、複数の異なるレート範囲が指定されている場合に以下で確認できます。**below** および **above** は1度のみ指定できますが、**from** - **to** は必要なだけ指定できます。各範囲の境界は、excludeLimit で、範囲に含まれるかどうかを指定することもできます。

例: 範囲ファセット要求の作成

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
        .forEntity( Cd.class )
            .get();
FacetingRequest priceFacetingRequest = builder.facet()
    .name( "priceFaceting" )
    .onField( "price" )
    .range()
    .below( 1000 )
    .from( 1001 ).to( 1500 )
    .above( 1500 ).excludeLimit()
    .createFacetingRequest();
```

7.5.3.2. ファセット要求の適用

ファセット要求は、FullTextQuery クラスを介して取得できる FacetManager クラスでクエリーに適用されます。

ファセットリクエストはいくつでも有効にでき、ファセット要求名を指定して getFacets() で後から取得できます。また、名前を指定してファセット要求を無効にできる disableFaceting() メソッドもあります。

ファセット要求は、FullTextQuery から取得できる FacetManager を使用してクエリーに適用できます。

例: ファセット要求の適用

```
// create a fulltext query
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, Cd.class );

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cds
List<Cd> cds = fullTextQuery.list();
...
```

```
// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
...
```

getFacets() を使用して、ファセットリクエスト名を指定することで、複数のファセット名を取得できます。

disableFaceting() メソッドは、名前を指定してブックマーク要求を無効にします。

7.5.3.3. クエリー結果の制限

最後でも重要ですが、ドルダウン機能を実装するために、元のクエリーに追加の基準として返されたすべての Facets を適用できます。そのためには、FacetSelection を使用できます。FacetSelections は FacetManager 経由で利用可能で、クエリー基準としてファセットを選択できます。(selectFacets)、ファセットの制限を削除して、すべてのファセット制限 (deselectFacets) を削除し、現在選択しているすべてのファセット (getSelectedFacets) を取得します。以下のスニペットは例になります。

```
// create a fulltext query
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, clazz );

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cd
List<Cd> cds = fullTextQuery.list();
assertTrue(cds.size() == 10);

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
assertTrue(facets.get(0).getCount() == 2)

// apply first facet as additional search criteria
facetManager.getFacetGroup( "priceFaceting" ).selectFacets( facets.get( 0 ) );

// re-execute the query
cds = fullTextQuery.list();
assertTrue(cds.size() == 2);
```

7.5.4. クエリープロセスの最適化

クエリーのパフォーマンスは、以下の基準に依存します。

- Lucene クエリー。
- 読み込まれたオブジェクト数: ページネーション (常時) またはインデックス処理 (必要な場合) を使用します。
- Hibernate Search が Lucene リーダーと対話する方法: 適切なリーダーストラテジーを定義します。
- インデックスから頻繁に抽出された値をキャッシュします。詳細は、Caching Index Values: FieldCache を参照してください。

7.5.4.1. インデックス値のキャッシュ: FieldCache

Lucene インデックスの主な機能は、クエリーへの一致を識別することです。クエリーが実行された後に、結果が分析され、有用な情報が抽出される必要があります。通常、Hibernate Search はクラスターイプとプライマリーキーを抽出する必要があります。

インデックスから必要な値を抽出するには、パフォーマンス負荷がかかります。パフォーマンス負荷が極めて低く、気付かない場合もありますが、キャッシュを行うのに役立つ場合もあります。

この要件は、使用されている Projections によって異なります。これは、クラスターイプがクエリーコンテキストまたは他の方法で推測される可能性があるため、クラスターイプが不要なためです。

@CacheFromIndex アノテーションを使用すると、Hibernate Search に必要なメインのメタデータフィールドのキャッシュをさまざまな方法で試すことができます。

```
import static org.hibernate.search.annotations.FieldCacheType.CLASS;
import static org.hibernate.search.annotations.FieldCacheType.ID;

@Indexed
@CacheFromIndex( { CLASS, ID } )
public class Essay {
    ...
```

このアノテーションを使用してクラスターイプと ID をキャッシュできます。

- **CLASS**: Hibernate Search は Lucene FieldCache を使用して、インデックスからクラスターイプの抽出のパフォーマンスを改善します。
  この値はデフォルトで有効になっており、@CacheFromIndex アノテーションを指定しない場合に Hibernate Search が適用されます。

- **ID**: プライマリー識別子はキャッシュを使用します。これにより、パフォーマンスが最も高いクエリーが提供されますが、消費するメモリーが多くなり、パフォーマンスが低下する可能性があります。

注記

ウォームアップ後のパフォーマンスおよびメモリー消費の影響を測定します (一部のクエリーの実行)。パフォーマンスは、フィールドキャッシュを有効にすることによって改善される可能性がありますが、常に改善されるわけではありません。

FieldCache の使用は、以下の点を考慮してください。

- メモリー使用量: このキャッシュには、かなりメモリーがハングします。通常、CLASS キャッシュの要件は ID キャッシュの要件よりも低くなります。

- インデックスウォームアップ: フィールドキャッシュを使用する場合、新しいインデックスまたはセグメントの最初のクエリーは、キャッシュが有効になっていない場合よりも遅くなります。

一部のクエリーでは、クラスターイプは全く不要です。その場合、**CLASS** フィールドキャッシュを有効にしても使用されない可能性があります。たとえば、単一クラスをターゲットとしている場合は、返される値はすべてそのタイプのものになります (これは各クエリー実行時に評価されます)。

ID FieldCache を使用するには、ターゲットエンティティーの ID が TwoWayFieldBridge (すべてのブリッジの構築として) を使用し、特定のクエリーに読み込まれるすべてのタイプが id のフィールド名を使用し、同じタイプの ID が割り当てられている必要があります (これは各クエリー実行時に評価されま

す)。

7.6. 手動によるインデックスの変更

Hibernate Core が変更をデータベースに適用すると、Hibernate Search はこれらの変更を検出し、インデックスを自動的に更新します (eventListener が無効でない場合)。バックアップが復元したり、データが影響を受ける場合のように、Hibernate を使用せずにデータベースに変更が加えられることがあります。このような場合、Hibernate Search は Manual Index API を公開し、インデックスから単一のエンティティーを明示的に更新または削除したり、データベース全体のインデックスを再構築したり、特定のタイプへのすべての参照を削除したりします。

これらのメソッドはすべて Lucene Index のみに影響し、変更は適用されません。

7.6.1. インデックスへのインスタンスの追加

FullTextSession.index(T エンティティー) を使用すると、特定のオブジェクトインスタンスを直接インデックスに追加または更新できます。このエンティティーがすでにインデックス化されている場合は、インデックスが更新されます。インデックスへの変更は、トランザクションコミット時にのみ適用されます。

FullTextSession.index(T エンティティー) を使用してオブジェクトまたはインスタンスを直接インデックスに追加します。このインデックスは、エンティティーがインデックス化されると更新されます。Infinispan Query は、トランザクションのコミット中に変更をインデックスに適用します。

例: FullTextSession.index(T エンティティー) を使用したエンティティーのインデックス作成

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
Object customer = fullTextSession.load( Customer.class, 8 );
fullTextSession.index(customer);
tx.commit(); //index only updated at commit time
```

1 つのタイプのすべてのインスタンスを追加する場合や、すべてのインデックスタイプのインスタンスを追加する場合には、MassIndexer を使用することが推奨されます。詳細はを参照してください。

MassIndexer を使用して、タイプ (またはすべてのインデックス型) のすべてのインスタンスを追加します。詳細は、Using a MassIndexer を参照してください。

7.6.2. インデックスからのインスタンスの削除

データベースからエンティティーまたは特定タイプのすべてのエンティティーを物理的に削除しなくても、特定のタイプの単一のエンティティーまたはすべてのエンティティーを Lucene インデックスから削除できます。この操作はパージと呼ばれ、**FullTextSession** を介しても実行されます。

パージ操作では、データベースから物理的に削除せずに、単一つのエンティティーまたは特定タイプのすべてのエンティティーを Lucene インデックスから削除できます。この操作は FullTextSession を使用して実行されます。

例: インデックスからエンティティーの特定のインスタンスを削除

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
fullTextSession.purgeAll( Customer.class );
```

```
//optionally optimize the index
//fullTextSession.getSearchFactory().optimize( Customer.class );
tx.commit(); //index is updated at commit time
```

このような操作の後にインデックスを最適化することが推奨されます。

注記

メソッド index、purge、および purgeAll は FullTextEntityManager でも使用できます。

注記

すべての手動のインデックスメソッド (index、urge、および purgeAll) は、データベースではなくインデックスにのみ影響し、トランザクションであっても、トランザクションが正常にコミットされるまで適用されません。または flushToIndexes を利用します。

7.6.3. インデックスの再構築

エンティティーマッピングをインデックスに変更する場合は、インデックス全体を更新する必要があります。たとえば、別のアナライザーを使用して既存のフィールドにインデックスを付ける場合は、影響を受けるタイプのインデックスを再構築する必要があります。また、データベースが置き換えられた場合 (バックアップから復元した場合や、レガシーシステムからインポートした場合など) は、既存データからインデックスを再構築できます。Hibernate Search は以下から選択する主要なストラテジーを提供します。

インデクサーのエンティティーマッピングを変更する場合は、インデックス全体を更新する必要があります。たとえば、既存のフィールドを異なるアナライザーを使用してインデックス化する場合、インデックスは影響を受けるタイプに対して再構築する必要があります。

さらに、バックアップから復元するか、レガシーシステムからインポートすることによりデータベースが置き換えられる場合、インデックスは既存データから再構築する必要があります。Infinispan Query は、以下の主要なストラテジーを提供します。

- **FullTextSession.index()** をすべてのエンティティーで使用しながら、**FullTextSession.flushToIndexes()** を定期的に使用。
- **MassIndexer** を使用。

7.6.3.1. FlushToIndexes() の使用

このストラテジーは、既存のインデックスを削除してから、**FullTextSession.purgeAll()** および **FullTextSession.index()** を使用してすべてのエンティティーをインデックスに戻すことで設定されますが、メモリーと効率の制約があります。効率を最大限に高めるためにも、Hibernate Search はインデックス操作をバッチ処理し、コミット時に実行します。大量のデータをインデックス化する場合は、トランザクションがコミットされるまですべてのドキュメントがキューに保存されるため、メモリー消費について注意する必要があります。キューを定期的に空にしない場合は、**OutOfMemoryException** が発生する可能性があります。これには **fullTextSession.flushToIndexes()** を使用します。**FullTextSession.flushToIndexes()** が呼び出されるたびに (またはトランザクションがコミットされると)、バッチキューが処理され、すべてのインデックスの変更が適用されます。フラッシュ後は、変更をロールバックできないことに注意してください。

例: index() および flushToIndexes() を使用したインデックス再構築

```
fullTextSession.setFlushMode(FlushMode.MANUAL);
fullTextSession.setCacheMode(CacheMode.IGNORE);
```

```
transaction = fullTextSession.beginTransaction();
//Scrollable results will avoid loading too many objects in memory
ScrollableResults results = fullTextSession.createCriteria( Email.class )
    .setFetchSize(BATCH_SIZE)
    .scroll( ScrollMode.FORWARD_ONLY );
int index = 0;
while( results.next() ) {
    index++;
    fullTextSession.index( results.get(0) ); //index each element
    if (index % BATCH_SIZE == 0) {
        fullTextSession.flushToIndexes(); //apply changes to indexes
        fullTextSession.clear(); //free memory since the queue is processed
    }
}
transaction.commit();
```

注記

この明示的な API が優先され、より優れた制御が提供されるため、**hibernate.search.default.worker.batch_size** が非推奨となりました。

アプリケーションがメモリーが不足しないように、バッチサイズを使用するようにしてください.大規模なバッチサイズオブジェクトの方がデータベースより高速ですが、より多くのメモリーが必要になります。

7.6.3.2. MassIndexer の使用

Hibernate Search の MassIndexer は、複数の並列スレッドを使用してインデックスを再ビルドします。オプションで、リロードする必要のあるエンティティーを選択するか、すべてのエンティティーのインデックスを変更できます。このアプローチは、パフォーマンスを最大化するために最適化されていますが、アプリケーションをメンテナンスモードに設定する必要があります。MassIndexer がビジーな場合、インデックスのクエリーは推奨されません。

例: MassIndexer を使用したインデックスの再構築

```
fullTextSession.createIndexer().startAndWait();
```

これにより、インデックスが再構築され、インデックスが削除されてから、データベースからすべてのエンティティーが再読み込みされます。簡単に使用できますが、プロセスのスピードを上げるために調整を行うことが推奨されます。

警告

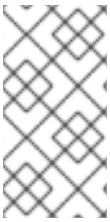
MassIndexer の進行中は、インデックスの内容未定義になります。MassIndexer が機能している間にクエリーを実行すると、一部の結果が失われる可能性が高くなります。

例: Tuned MassIndexer の使用

```
fullTextSession
 .createIndexer( User.class )
 .batchSizeToLoadObjects( 25 )
 .cacheMode( CacheMode.NORMAL )
 .threadsToLoadObjects( 12 )
 .idFetchSize( 150 )
 .progressMonitor( monitor ) //a MassIndexerProgressMonitor implementation
 .startAndWait();
```

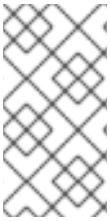
これにより、すべての User インスタンス (およびフラグ) のインデックスが再構築され、クエリーごとに 25 個のオブジェクトのバッチを使用して User インスタンスをロードするために、12 個の並列スレッドが作成されます。また、これら 12 個のスレッドが Lucene ドキュメントを出力するには、インデックス化された埋め込み関係およびカスタム **FieldBridges** または **ClassBridges** を処理する必要もあります。スレッドは、変換プロセス中に追加属性のレイジーローディングをトリガーします。そのため、並行して機能しているスレッドは多く必要になります。実際のインデックス書き込みで稼働しているスレッドの数は、各インデックスのバックエンド設定によって定義されます。

Cachemode を **CacheMode.IGNORE** (デフォルト) のままにすることが推奨されます。これは、ほとんどの場合でキャッシュが不要な追加オーバーヘッドになるためです。メインのエンティティーがインデックスに含まれる列挙のようなデータに関連する場合は、パフォーマンスを向上させる可能性があるため、データに応じて他の **CacheMode** を有効にすると便利です。



注記

最適なパフォーマンスを実現するために理想的なスレッド数は、全体的なアーキテクチャー、データベース設計、およびデータ値によって大きく異なります。すべての内部スレッドグループには意味のある名前が付けられているため、スレッドダンプ多くのの診断ツールで簡単に識別できます。



注記

MassIndexer はトランザクションを認識しないため、開始したり、コミットしたりする必要はありません。これはトランザクション処理ではないため、ユーザーが処理中にシステムを使用するようにすることは推奨されません。これは、ユーザーが結果を見つけられず、システム負荷が高すぎる可能性があるためです。

インデックス処理にかかる時間やメモリー消費量に影響を与える他のパラメーターには、以下が含まれます。

- **hibernate.search.[default|<indexname>].exclusive_index_use**
- **hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs**
- **hibernate.search.[default|<indexname>].indexwriter.max_merge_docs**
- **hibernate.search.[default|<indexname>].indexwriter.merge_factor**
- **hibernate.search.[default|<indexname>].indexwriter.merge_min_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_max_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_max_optimize_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_calibrate_by_deletes**

- **hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size**
- **hibernate.search.[default|<indexname>].indexwriter.term_index_interval**

以前のバージョンにも **max_field_length** がありましたが、これは Lucene から削除されました。**LimitTokenCountAnalyzer** を使用すると、同様の効果を得ることができます。

すべての **.indexwriter** パラメーターは Lucene 固有で、Hibernate Search はこれらのパラメーターを渡します。

MassIndexer は、前方のみのスクロール可能な結果を使用して、読み込まれるプライマリーキーを繰り返し処理しますが、MySQL の JDBC ドライバーはメモリーのすべての値を読み込みます。この最適化を回避するには、**idFetchSize** を **Integer.MIN_VALUE** に設定します。

7.7. インデックスの最適化

時折、Lucene インデックスを最適化する必要があります。このプロセスは基本的にはデフラグです。最適化がトリガーされるまで、Lucene は削除されたドキュメントのみをマークします。物理的には適用されません。最適化プロセス中に削除が適用され、Lucene ディレクトリー内のファイル数にも影響を及ぼします。

Lucene インデックスを最適化すると検索が高速になりますが、インデックス化 (更新) パフォーマンスには影響はありません。最適化中に検索を実行できますが、おそらく遅くなる可能性があります。インデックスの更新はすべて停止します。最適化のスケジュール設定が推奨されます。

Lucene インデックスを最適化すると検索が高速になりますが、インデックス更新のパフォーマンスには影響しません。検索は、最適化プロセス中に実行できますが、期待よりも遅くなります。すべてのインデックスの更新は、最適化中に保留されます。そのため、最適化のスケジュール設定を行うことが推奨されます。

- アイドル状態のシステムの場合、または検索が頻繁に行われない場合
- 多数のインデックス変更が適用された後。

MassIndexer は、デフォルトで、開始時と処理終了時にインデックスを最適化します。**MassIndexer.optimizeAfterPurge** および **MassIndexer.optimizeOnFinish** を使用して、このデフォルト動作を変更します。詳細は、Using a MassIndexer を参照してください。

7.7.1. 自動最適化

Hibernate Search は、以下のいずれかの後にインデックスを自動的に最適化できます。

Infinispan のクエリーは、以下の後にインデックスを自動的に最適化します。

- 一定量の操作 (挿入または削除)。
- 一定量のトランザクション。

インデックスの自動最適化の設定は、グローバルまたはインデックスごとに定義できます。

例: 自動最適化パラメーターの定義

```
hibernate.search.default.optimizer.operation_limit.max = 1000
hibernate.search.default.optimizer.transaction_limit.max = 100
hibernate.search.Animal.optimizer.transaction_limit.max = 50
```

最適化は、以下のいずれかの時点で **Animal** インデックスに対してトリガーされます。

- 追加および削除の数は **1000** に到達します。
- トランザクション数が **50** に達すると、**hibernate.search.Animal.optimizer.transaction_limit.max** が **hibernate.search.default.optimizer.transaction_limit.max** よりも優先されます。

これらのパラメーターが定義されていないと、最適化は自動的に処理されません。

OptimizerStrategy のデフォルト実装は、**org.hibernate.search.store.optimization.OptimizerStrategy** を実装して上書きし、**optimizer.implementation** プロパティーを実装の完全修飾名に設定して上書きできます。この実装は、インターフェイスを実装し、パブリッククラスであり、引数なしでパブリックコンストラクターを持つ必要があります。

例: カスタム OptimizerStrategy の読み込み

```
hibernate.search.default.optimizer.implementation = com.acme.worlddomination.SmartOptimizer
hibernate.search.default.optimizer.SomeOption = CustomConfigurationValue
hibernate.search.humans.optimizer.implementation = default
```

キーワード **default** を使用すると、Hibernate Search のデフォルト実装を選択できます。**.optimizer** キーセパレーター以降のすべてのプロパティーは、起動時に実装の初期化メソッドに渡されます。

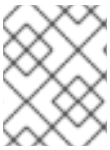
7.7.2. 手動の最適化

SearchFactory を使用して、Hibernate Search からプログラムで Lucene インデックスを最適化 (デフラグ) できます。

例: プログラムのインデックスの最適化

```
FullTextSession fullTextSession = Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();

searchFactory.optimize(Order.class);
// or
searchFactory.optimize();
```

最初の例は、Orders を保持している Lucene インデックスを最適化し、次の例はすべてのインデックスを最適化します。



注記

searchFactory.optimize() は Jakarta Messaging バックエンドには影響しません。マスターノードで最適化操作を適用する必要があります。

SearchFactory.optimize() は Jakarta Message バックエンドに影響を与えないため、マスターノードに適用されます。

7.7.3. 最適化の調整

Apache Lucene には、最適化の実行方法に影響を与えるいくつかのパラメーターがあります。Hibernate Search はこれらのパラメーターを公開します。

その他のインデックス最適化パラメーターには、以下が含まれます。

- **hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs**
- **hibernate.search.[default|<indexname>].indexwriter.max_merge_docs**
- **hibernate.search.[default|<indexname>].indexwriter.merge_factor**
- **hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size**
- **hibernate.search.[default|<indexname>].indexwriter.term_index_interval**

7.8. 高度な機能

7.8.1. SearchFactory へのアクセス

SearchFactory オブジェクトは、Hibernate Search の基礎となる Lucene リソースを追跡します。これは、Lucene にネイティブにアクセスする便利な方法です。**SearchFactory** は FullTextSession からアクセスできます。

例: SearchFactory へのアクセス

```
FullTextSession fullTextSession = Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

7.8.2. IndexReader の使用

Lucene のクエリーは IndexReader で実行されます。Hibernate Search は、パフォーマンスを最大化するためにインデックスリーダーをキャッシュするか、更新された IndexReader 軽減 I/O 操作を取得するために他の効率的なストラテジーを提供する場合があります。コードはこれらのキャッシュされたリソースにアクセスできますが、要件がいくつかあります。

例: IndexReader のアクセス

```
IndexReader reader = searchFactory.getIndexReaderAccessor().open(Order.class);
try {
  //perform read-only operations on the reader
}
finally {
  searchFactory.getIndexReaderAccessor().close(reader);
}
```

この例では、SearchFactory は (シャード化ストラテジーに関連して) このエンティティーのクエリーに必要なインデックスを判別します。各インデックスで設定された ReaderProvider を使用して、関連するすべてのインデックスの上に複合 **IndexReader** を返します。この IndexReader は複数のクライアント間で共有されるため、以下のルールに従う必要があります。

- IndexReader.close() を呼び出さず、必要に応じて readerProvider.closeReader(reader) を finally ブロックで使用します。
- 変更操作には、この IndexReader を使用しないでください (読み取り専用の IndexReader であり、このような試行によって例外が発生します)。

これらのルール以外に、特にネイティブな Lucene クエリーを行うため、IndexReader を自由に使用できます。共有 IndexReaders を使用すると、たとえばファイルシステムから直接開くよりも、ほとんどのクエリーがより効率的になります。

open (Class...types) メソッドの代替として、open (String...indexNames) を使用して、1 つ以上のインデックス名で渡すことができます。このストラテジーを使用して、シャードが使用される場合に、インデックスタイプに対してインデックスのサブセットを選択することもできます。

例: インデックス名による IndexReader へのアクセス

```
IndexReader reader = searchFactory.getIndexReaderAccessor().open("Products.1", "Products.3");
```

7.8.3. Lucene ディレクトリーへのアクセス

Directory は、インデックスストレージを表すために Lucene で使用される最も一般的な抽象化です。Hibernate Search は Lucene Directory と直接対話することはありませんが、これらの対話は IndexManager 経由で抽象化されます。インデックスが必ずしもディレクトリーによって実装される必要はありません。

インデックスが Directory として表示され、アクセスする必要がある場合は、IndexManager からディレクトリーへの参照を取得できます。IndexManager を DirectoryBasedIndexManager にキャストし、**getDirectoryProvider().getDirectory()** を使用して基礎となる Directory への参照を取得します。これは推奨されていません.代わりに IndexReader を使用することが推奨されます。

7.8.4. シャード化インデックス

場合によっては、特定のエンティティーのインデックス付きデータを複数の Lucene インデックスに分割 (シャード) すると役に立つことがあります。

警告

シャード化は、欠点が短所を上回った場合にのみ実装する必要があります。単一の検索用にシャードをすべて開く必要があるため、シャード化されたインデックスの検索は一般的に遅くなります。

シャード化のユースケースを以下に示します。

- 単一のインデックスが大きいと、インデックスの更新時間は遅くなります。
- 一般的な検索は、データが顧客、地域、またはアプリケーションによってセグメント化された場合など、インデックスのサブセットのみに一致します。

デフォルトでは、シャードの数が設定されていないとシャード化は有効になりません。これには、**hibernate.search.<indexName>.sharding_strategy.nbr_of_shards** プロパティーを使用します。

例: インデックスのシャード化

この例では、5 つのシャードが有効にされています。

```
hibernate.search.<indexName>.sharding_strategy.nbr_of_shards = 5
```

データをサブインデックスに分割するには、IndexShardingStrategy を使用します。デフォルトのシャード化ストラテジーは、(FieldBridge によって生成された)ID 文字列表現のハッシュ値に従ってデータを分割します。これにより、シャードが大幅に分散されます。カスタムの IndexShardingStrategy を実装して、デフォルトのストラテジーを置き換えることができます。カスタムストラテジーを使用するには、**hibernate.search.<indexName>.sharding_strategy** プロパティーを設定する必要があります。

例: カスタムシャードストラテジーの指定

```
hibernate.search.<indexName>.sharding_strategy = my.shardingstrategy.Implementation
```

IndexShardingStrategy プロパティーでは、クエリーを実行するシャードを選択して検索を最適化することもできます。フィルターをアクティベートすることで、シャード化ストラテジーはクエリー (IndexShardingStrategy.getIndexManagersForQuery) に回答するために使用されるシャードのサブセットを選択できるため、クエリーの実行が速くなります。

各シャードには独立した IndexManager があるため、異なるディレクトリープロバイダーおよびバックエンド設定を使用するように設定できます。以下の例の Animal エンティティーの IndexManager インデックス名は **Animal.0** から **Animal.4** です。つまり、各シャードには所有するインデックス名の後に **.** (ドット) とそのインデックス番号が設定されます。

例: エンティティー解析のシャード化設定

```
hibernate.search.default.indexBase = /usr/lucene/indexes
hibernate.search.Animal.sharding_strategy.nbr_of_shards = 5
hibernate.search.Animal.directory_provider = filesystem
hibernate.search.Animal.0.indexName = Animal00
hibernate.search.Animal.3.indexBase = /usr/lucene/sharded
hibernate.search.Animal.3.indexName = Animal03
```

上記の例では、設定はデフォルトの id 文字列ハッシュストラテジーを使用し、シャードは Animal インデックスを 5 つのサブインデックスに使用しています。すべてのサブインデックスはファイルシステムのインスタンスで、各サブインデックスが保存されるディレクトリーは、以下のようになります。

- sub-index 0: **/usr/lucene/indexes/Animal00** (共有 indexBase、indexName を上書き)
- sub-index 1: **/usr/lucene/indexes/Animal.1** (共有 indexBase、デフォルトの indexName)
- sub-index 2: **/usr/lucene/indexes/Animal.2** (共有 indexBase、デフォルトの indexName)
- sub-index 3: **/usr/lucene/shared/Animal03** (上書きされた indexBase、上書きされた indexName)
- sub-index 4: **/usr/lucene/indexes/Animal.4** (共有 indexBase、デフォルトの indexName)

IndexShardingStrategy を実装する場合は、任意のフィールドを使用してシャード化の選択を判断できます。deletion、**purge**、**purgeAll** などの操作を処理するには、すべてのフィールド値またはプライマリー識別子を読み取れずにインデックスを返す必要があることもあります。このような場合、すべてのインデックスが返されるため、削除操作は、削除されるドキュメントが含まれる可能性のあるすべてのインデックスに伝播されます。

7.8.5. Lucene のスコアリングカスタマイズ

Lucene を使用するとユーザーは、org.apache.lucene.search.Similarity を拡張して、そのフラグ式をカスタマイズできます。このクラスで定義された抽象メソッドは、以下の式の係数と一致し、ドキュメント d のクエリー q のスコアを計算します。

org.apache.lucene.search.Similarity を拡張して、Lucene の診断式をカスタマイズします。抽象メソッドは、以下のようにドキュメント **d** のクエリー **q** のスコアを計算するために使用される式と一致します。

```
*score(q,d) = coord(q,d) · queryNorm(q) · ∑ ~t in q~ ( tf(t in d) ·
idf(t) ^2^ · t.getBoost() · norm(t,d) )*
```

ファクター	説明
tf(t ind)	ドキュメント (d) の用語 (t) の周波数係数。
idf(t)	用語の頻度に関する記録。
coord(q,d)	指定されたドキュメントのクエリー用語がいくつあるかに基づくスコア要因。
queryNorm(q)	クエリー間でスコアを設定するために使用される正規化の要素。
t.getBoost()	フィールドブースト。
norm(t,d)	いくつかの (インデックス時間) ブーストおよび長さ要素をカプセル化します。

この式の詳細は、本書の範囲外です。詳細は、Similarity の Java ドキュメントを参照してください。

Hibernate Search では、Lucene の類似性の計算を修正する方法を利用できます。

最初に、プロパティー **hibernate.search.matchity** を使用して、Similarity 実装の完全に指定されたクラス名を指定すると、デフォルトの類似性を設定できます。デフォルト値は org.apache.lucene.search.DefaultSimilarity です。

また、**similarity** プロパティーを設定して特定のインデックスに使用される類似性を上書きすることもできます。

```
hibernate.search.default.similarity = my.custom.Similarity
```

最後に、**@Similarity** アノテーションを使用してクラスレベルのデフォルトの類似性を上書きできます。

```
@Entity
@Indexed
@Similarity(impl = DummySimilarity.class)
public class Book {
...
}
```

たとえば、ドキュメントに用語が表示される頻度は重要ではないと仮定します。用語が 1 回出現したドキュメントは、複数回見つかったドキュメントと同様にスコア付けされる必要があります。この場合、メソッド tf(float freq) のカスタム実装は 1.0 を返します。

警告

2 つのエンティティーが同じインデックスを共有する場合、同じ Similarity 実装を宣言する必要があります。同じクラス階層のクラスは常にインデックスを共有するため、シミュリティー実装の上書きはできません。

同様に、インデックス設定とクラスレベルの設定で類似性を定義することは、競合が発生するため意味がありません。このような設定は拒否されます。

7.8.6. 例外処理の設定

Hibernate Search では、インデックスプロセスでの例外の処理方法を設定できます。設定が指定されていない場合、デフォルトでは例外がログ出力に記録されます。以下のように例外ロギングメカニズムを明示的に宣言できます。

```
hibernate.search.error_handler = log
```

デフォルトの例外処理は、同期インデックスと非同期インデックスの両方で行われます。Hibernate Search は、デフォルトのエラー処理実装を上書きする簡単なメカニズムを提供します。

独自の実装を指定するには、**handle(ErrorContext context)** メソッドを提供する ErrorHandler インターフェイスを実装する必要があります。**ErrorContext** は、プライマリー **LuceneWork** インスタンス、**LuceneWork**、およびプライマリー例外により処理できなかった後続の LuceneWork インスタンスへの参照を提供します。

```
public interface ErrorContext {
  List<LuceneWork> getFailingOperations();
  LuceneWork getOperationAtFault();
  Throwable getThrowable();
  boolean hasErrors();
}
```

このエラーハンドラーを Hibernate Search に登録するには、設定プロパティーで ErrorHandler 実装の完全修飾クラス名を宣言する必要があります。

```
hibernate.search.error_handler = CustomerErrorHandler
```

7.8.7. Hibernate Search の無効化

Hibernate Search は必要に応じて部分的にまたは完全に無効にできます。Hibernate Search のインデックスは、たとえばインデックスが読み取り専用の場合や、自動ではなく手動でインデックスを実行したい場合など、無効にすることができます。Hibernate Search を完全に無効にして、インデックス設定や検索を阻止することもできます。

インデックスの無効化

Hibernate Search インデックスを無効にするには、**indexing_strategy** 設定オプションを **manual** に変更して JBoss EAP を再起動します。

```
hibernate.search.indexing_strategy = manual
```

Hibernate Search を完全に無効にする

Hibernate Search を完全に無効にするには、**autoregister_listeners** 設定オプションを **false** に変更してすべてのリスナーを無効にし、JBoss EAP を再起動します。

```
hibernate.search.autoregister_listeners = false
```

7.9. モニタリング

Hibernate Search は、**SearchFactory.getStatistics()** を経由した **Statistics** オブジェクトへのアクセスを提供します。たとえば、インデックス付けされるクラスや、インデックスに含まれるエンティティーの数を判別できます。この情報は、常に利用できます。ただし、設定で **hibernate.search.generate_statistics** プロパティーを指定すると、合計および平均の Lucene クエリーおよびオブジェクトの読み込みタイミングを収集することもできます。

Jakarta Management を使用した統計へのアクセス

Jakarta Management による統計へのアクセスを有効にするには、**hibernate.search.jmx_enabled** プロパティーを **true** に設定します。これにより、**StatisticsInfoMBean** Bean が自動的に登録され、**Statistics** オブジェクトを使用した統計へのアクセスが提供されます。設定によっては、**IndexingProgressMonitorMBean** Bean も登録できます。

インデックスのモニタリング

マスインデックサー API を使用する場合は、**IndexingProgressMonitorMBean** Bean を使用してインデックス作成の進捗をモニターできます。Bean は、インデックスが進行中の Jakarta Management にのみバインドされます。

注記

JConsole を使用して Jakarta Management bean にリモートでアクセスするには、システムプロパティー **com.sun.management.jmxremote** を **true** に設定します。

付録A リファレンス資料

A.1. HIBERNATE プロパティー

表A.1 **persistence.xml** ファイルで設定できる接続プロパティー

プロパティー名	値	説明
javax.persistence.jdbc.driver	**org.hsqldb.jdbcDriver**	使用する JDBC ドライバーのクラス名。
javax.persistence.jdbc.user	sa	ユーザー名
javax.persistence.jdbc.password		パスワード。
javax.persistence.jdbc.url	**jdbc:hsqldb:.**	JDBC 接続 URL。

表A.2 Hibernate 設定プロパティー

プロパティー名	説明
hibernate.dialect	Hibernate **org.hibernate. metaect.Dialect** のクラス名。Hibernate が特定の関係データベース向けに最適化された SQL を生成できるようにします。 ほとんどの場合、Hibernate は JDBC ドライバーによって返される JDBC メタデータに基づいて適切な **org.hibernate. metaect.Dialect** 実装を選択できます。
hibernate.show_sql	ブール値。すべての SQL ステートメントをコンソールに書き込みます。これは、ログカテゴリー **org.hibernate.SQL** を **debug** に設定する代わりになります。
hibernate.format_sql	ブール値。pretty print で、ログとコンソールに SQL を出力します。
hibernate.default_schema	生成された SQL で指定のスキーマ/テーブルスペースを持つ修飾されていないテーブル名。
hibernate.default_catalog	生成された SQL で指定のカタログと修飾されていないテーブル名を修飾します。
hibernate.session_factory_name	org.hibernate.SessionFactory は、作成後に Java Naming および Directory Interface でこの名前に自動的にバインドされます。たとえば、**jndi/composite/name**.Jpa です。

プロパティー名	説明
hibernate.max_fetch_depth	単独の関連 (one-to-one、many-to-one) の外部結合フェッチツリーの最大深さを設定します。**0** は、デフォルトの外部ジョインフェッチを無効にします。推奨の値は **0** から **3** の間です。
hibernate.default_batch_fetch_size	関連付けの Hibernate バッチフェッチのデフォルトサイズを設定します。推奨される値は **4**、**8**、**16** です。
hibernate.default_entity_mode	この **SessionFactory** か ら開かれたすべてのセッションに対するエンティティー表現のデフォルトモードを設定します。値には **dynamic-map**、**dom4j**、**pojo** が含まれます。
hibernate.order_updates	ブール値。Hibernate は、更新する製品のプライマリーキーの値で SQL 更新の順序を強制します。これにより、非常に同時になるシステムでは、トランザクションのデッドロックが少なくなります。
hibernate.generate_statistics	ブール値。有効にすると、Hibernate はパフォーマンスチューニングに役立つ統計を収集します。
hibernate.use_identifier_rollback	ブール値。有効な場合、オブジェクトが削除されると、生成された識別子プロパティーがデフォルト値にリセットされます。
hibernate.use_sql_comments	ブール値。有効にすると、Hibernate はデバッグを容易にするため、SQL 内でコメントを生成します。デフォルト値は **false** です。
hibernate.id.new_generator_mappings	ブール値。@GeneratedValue を使用する場合、このプロパティーには関係があります。これは、新しい IdentifierGenerator 実装が javax.persistence.GenerationType.AUTO、javax.persistence.GenerationType.TABLE および javax.persistence.GenerationType.SEQUENCE に使用されるかどうかを示します。デフォルト値は **true** です。

プロパティー名	説明
hibernate.ejb.naming_strategy	Hibernate EntityManager を使用する際に org.hibernate.cfg.NamingStrategy 実装を選択します。**Hibernate.ejb.naming_strategy** は Hibernate 5.0 でサポートされなくなりました。これを使用すると、非推奨のメッセージがログに記録され、split ImplicitNamingStrategy および PhysicalNamingStrategy が優先され、サポートされなくなったことを示すメッセージが削除されます。 アプリケーションが EntityManager を使用しない場合は、以下の手順に従って NamingStrategy: Hibernate Reference Documentation - Naming Strategies を設定します。 MetadataBuilder を使用したネイティブブートストラップおよび暗黙的な命名ストラテジーの適用例は、Hibernate 5.0 ドキュメントの http://docs.jboss.org/hibernate/orm/5.0/userguide/html_single/Hibernate_User_Guide.html#bootstrap-native-metadata を参照してください。物理的な命名ストラテジーは、**MetadataBuilder.applyPhysicalNamingStrategy()** を使用して適用できます。**org.hibernate.boot.MetadataBuilder**, see https://docs.jboss.org/hibernate/orm/5.0/javadocs/ を参照してください。
hibernate.implicit_naming_strategy	使用する **org.hibernate.boot.model.naming.ImplicitNamingStrategy** クラスを指定します。**hibernate.implicit_naming_strategy** は、ImplicitNamingStrategy を実装するカスタムクラスの設定にも使用できます。この設定には、以下の短縮名が定義されます。 ● **default** - **ImplicitNamingStrategyJpaCompliantImpl** ● **jpa** - **ImplicitNamingStrategyJpaCompliantImpl** ● **legacy-jpa** - **ImplicitNamingStrategyLegacyJpaImpl** ● **legacy-hbm** - **ImplicitNamingStrategyLegacyHbmImpl** ● **component-path** - **ImplicitNamingStrategyComponentPathImpl** デフォルト設定は、**default** の省略名の **ImplicitNamingStrategy** によって定義されます。デフォルト設定が空の場合、フォールバックは **ImplicitNamingStrategyJpaCompliantImpl** を使用します。

プロパティー名	説明
hibernate.physical_naming_strategy	データベースオブジェクト名に物理的な命名ルールを適用するためのプラグ可能なストラテジーコントラクト。使用する PhysicalNamingStrategy クラスを指定します。**PhysicalNamingStrategyStandardImpl** はデフォルトで使用されます。**hibernate.physical_naming_strategy** を使用して、PhysicalNamingStrategy を実装するカスタムクラスを設定することもできます。

重要

hibernate.id.new_generator_mappings の場合、新規アプリケーションはデフォルト値の **true** のままにする必要があります。Hibernate 3.3.x を使用した既存のアプリケーションは、シーケンスオブジェクトまたはテーブルベースのジェネレーターを使用し、後方互換性を維持するために **false** に変更する必要がある場合があります。

表A.3 Hibernate JDBC および接続プロパティー

プロパティー名	説明
hibernate.jdbc.fetch_size	JDBC フェッチサイズを決定するゼロ以外の値 (**Statement.setFetchSize()** を呼び出します)。
hibernate.jdbc.batch_size	ゼロ以外の値は、Hibernate による JDBC2 バッチ更新の使用を有効にします。推奨される値は **5** から **30** の間です。
hibernate.jdbc.batch_versioned_data	ブール値。JDBC ドライバーが **executeBatch()** から正しい行数を返す場合は、このプロパティーを **true** に設定します。その後、Hibernate は、自動的にバージョン付けされたデータにバッチ処理を使用します。デフォルト値は **false** です。
hibernate.jdbc.factory_class	カスタムの org.hibernate.jdbc.Batcher を選択します。ほとんどのアプリケーションでは、この設定プロパティーは必要ありません。
hibernate.jdbc.use_scrollable_resultset	ブール値。Hibernate による JDBC2 のスクロール可能な結果セットの使用を有効にします。このプロパティーは、ユーザー指定の JDBC 接続を使用する場合にのみ必要です。Hibernate は接続メタデータを使用します。
hibernate.jdbc.use_streams_for_binary	ブール値。これはシステムレベルのプロパティーです。**binary** または **serializable** を JDBC に書き込む際や、JDBC から読み込む際には、ストリームを使用します。

プロパティー名	説明
hibernate.jdbc.use_get_generated_keys	ブール値。JDBC3 の **PreparedStatement.getGeneratedKeys()** を使用して、挿入後にネイティブに生成されたキーを取得できます。JDBC3+ ドライバーおよび JRE1.4+ が必要です。JDBC ドライバーに、Hibernate 識別子ジェネレーターの問題がある場合は false に設定します。デフォルトでは、接続メタデータを使用してドライバー機能を判断しようとします。
hibernate.connection.provider_class	Hibernate への JDBC 接続を提供するカスタム org.hibernate.connection.ConnectionProvider のクラス名。
hibernate.connection.isolation	JDBC トランザクション分離レベルを設定します。java.sql.Connection で有意な値を確認しますが、ほとんどのデータベースはすべての分離レベルをサポートしておらず、追加の非標準分離を定義することに注意してください。標準値は **1**、**2**、**4**、**8** です。
hibernate.connection.autocommit	ブール値。このプロパティーの使用は推奨されません。JDBC プールされた接続の自動コミットを有効にします。
hibernate.connection.release_mode	Hibernate が JDBC 接続を解放するタイミングを指定します。デフォルトでは、明示的にセッションが閉じられるか、切断されるまで JDBC 接続が保持されます。デフォルト値 **auto** は Jakarta Transactions および CMT トランザクションストラテジーに **after_statement** を選択し、JDBC トランザクションストラテジーに **after_transaction** を選択します。 使用可能な値は auto (デフォルト)、on_close、**after_transaction**、**after_statement** です。 この設定は、**SessionFactory.openSession** によって返されるセッションにのみ影響します。**SessionFactory.getCurrentSession** から取得したセッションの場合、使用するために設定された **CurrentSessionContext** 実装はそのセッションの接続リリースモードを制御します。
hibernate.connection.**<propertyName>**	JDBC プロパティーを **<propertyName>** から **DriverManager.getConnection()** に渡します。

プロパティー名	説明
hibernate.jndi.**<propertyName>**	**<propertyName>** プロパティーを Java Naming and Directory Interface **InitialContextFactory** に渡します。

表A.4 Hibernate キャッシュプロパティー

プロパティー名	説明
hibernate.cache.region.factory_class	カスタム **CacheProvider** のクラス名。
hibernate.cache.use_minimal_puts	ブール値。書き込みを最小限に抑えるために 2 次レベルのキャッシュ操作を最適化し、読み取りの頻度を上げます。この設定はクラスター化されたキャッシュで最も便利なもので、Hibernate3 ではクラスター化されたキャッシュ実装に対してデフォルトで有効になっています。
hibernate.cache.use_query_cache	ブール値。クエリーキャッシュを有効にします。個別のクエリーをキャッシュ可能に設定する必要があります。
hibernate.cache.use_second_level_cache	ブール値。**<cache>** マッピングを指定するクラスに対してデフォルトで有効になっている 2 次レベルのキャッシュを完全に無効にするために使用されます。
hibernate.cache.query_cache_factory	カスタム **QueryCache** インターフェイスのクラス名。デフォルト値はビルトインの **StandardQueryCache** です。
hibernate.cache.region_prefix	2 次キャッシュリージョン名に使用する接頭辞。
hibernate.cache.use_structured_entries	ブール値。Hibernate が、よりヒューマンリーダブルな形式で、2 次レベルのキャッシュにデータを格納するように強制します。
hibernate.cache.default_cache_concurrency_strategy	@Cacheable または @Cache のいずれかが使用された場合に使用するデフォルトの org.hibernate.annotations.CacheConcurrencyStrategy の名前を付与するために使用される設定。**@Cache(strategy="..")** は、これを上書きするのに使用されます。

表A.5 Hibernate トランザクションプロパティー

プロパティー名	説明
hibernate.transaction.factory_class	Hibernate **Transaction** API で使用するための **TransactionFactory** のクラス名。デフォルトは **JDBCTransactionFactory**
jta.UserTransaction	アプリケーションサーバーから Jakarta Transactions **UserTransaction** を取得するために **JTATransactionFactory** によって使用される Java Naming and Directory Interface 名。
hibernate.transaction.manager_lookup_class	**TransactionManagerLookup** のクラス名。これは、JVM レベルキャッシングが有効な場合や、Jakarta Transactions 環境で hilo ジェネレーターを使用する場合に必要です。
hibernate.transaction.flush_before_completion	ブール値。有効にすると、トランザクションの完了前のフェーズでセッションが自動的にフラッシュされます。組み込みおよび自動セッションコンテキスト管理が推奨されます。
hibernate.transaction.auto_close_session	ブール値。有効にすると、トランザクションの完了フェーズ中にセッションが自動的に閉じられます。組み込みおよび自動セッションコンテキスト管理が推奨されます。

表A.6 その他の Hibernate プロパティー

プロパティー名	説明
hibernate.current_session_context_class	現在の **Session** セッションのスコープ設定にカスタムストラテジーを指定します。値には、**jta**、**thread**、**managed**、**custom.Class** が含まれます。
hibernate.query.factory_class	HQL パーサー実装 (**org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory** または **org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory**) を選択します。
hibernate.query.substitutions	Hibernate クエリーのトークンから SQL トークン (トークンは関数またはリテラル名) にマップするために使用されます。たとえば、**hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC**。

プロパティー名	説明
hibernate.query.conventional_java_constants	Java の定数が Java の命名規則に従うかどうかを示します。デフォルトは **false** です。既存のアプリケーションは、従来の Java 定数がアプリケーションで使用されている場合にのみ **true** に設定できます。 これを **true** に設定すると、エイリアスが Java 命名規則に準拠するかどうかを確認するだけで、Hibernate はエイリアスを Java の定数として処理する必要があるかどうかを判別できるため、パフォーマンスが大幅に向上します。 このプロパティーを **false** に設定すると、Hibernate はエイリアスをクラス (アプリケーションのオーバーヘッド) としてロードし、Java の定数として処理する必要があります。エイリアスがクラスとしてロードされない場合、Hibernate はエイリアスを Java の定数として処理します。
hibernate.hbm2ddl.auto	**SessionFactory** の作成時にスキーマをデータベースに自動的に検証またはエクスポートします。**Create-drop** で **SessionFactory** を明示的に閉じると、データベーススキーマがドロップされます。プロパティー値のオプションは **validate**、**update**、**create**、**create-drop** です。
hibernate.hbm2ddl.import_files	SessionFactory の作成時に実行される SQL Mission ステートメントが含まれるオプションのファイルのコンマ区切りの名前です。これは、テストまたはデモンストレーションに役立ちます。たとえば、AIA ステートメントを追加すると、デプロイ時にデータベースが最小限のデータセットでデータが投入されます。値の例は **/humans.sql,/dogs.sql** です。 ファイルの順序は重要です。以下のファイルのステートメントの前に特定ファイルのステートメントが実行されるためです。これらのステートメントは、たとえば **hibernate.hbm2ddl.auto** が **create** or **create-drop** に設定されている場合にのみ実行されます。
hibernate.hbm2ddl.import_files_sql_extractor	カスタム mportSqlCommandExtractor のクラス名デフォルトは、ビルトインの SingleLineSqlCommandExtractor です。これは、各インポートファイルから単一の SQL ステートメントを抽出する専用パーサーを実装する際に便利です。Hibernate は、複数の行 (各ステートメントの最後で必須のセミコロン) に散在する命令/コメントおよび引用符で囲まれた文字列をサポートする、複数の LinesLookupCommandLookupor も提供します。

プロパティー名	説明
hibernate.bytecode.use_reflection_optimizer	ブール値。これはシステムレベルのプロパティーで、**hibernate.cfg.xml** ファイルで設定することはできません。ランタイムの反映の代わりにバイトコード操作の使用を有効にします。リフレクションはトラブルシューティング時に役に立つことがあります。Hibernate には常に cglib または javassist のいずれかが必要です。
hibernate.bytecode.provider	javassist または cglib は、バイト操作エンジンとして使用できます。デフォルトは **javassist** です。値は **javassist** または **cglib** です。

表A.7 Hibernate SQL ダイアレクト (**hibernate.dialect**)

RDBMS	ダイアレクト
DB2	**org.hibernate.dialect.DB2Dialect**
DB2 AS/400	**org.hibernate.dialect.DB2400Dialect**
DB2 OS390	**org.hibernate.dialect.DB2390Dialect**
Adobird	**org.hibernate.dialect.FirebirdDialect**
FrontBase	**org.hibernate.dialect.FrontbaseDialect**
H2 Database	**org.hibernate.dialect.H2Dialect**
HypersonicSQL	**org.hibernate.dialect.HSQLDialect**
Informix	**org.hibernate.dialect.InformixDialect**
Ingres	**org.hibernate.dialect.IngresDialect**
Interbase	**org.hibernate.dialect.InterbaseDialect**
MariaDB 10	**org.hibernate.dialect.MariaDB10Dialect**
MariaDB Galera Cluster 10	**org.hibernate.dialect.MariaDB10Dialect**
Mckoi SQL	**org.hibernate.dialect.MckoiDialect**
Microsoft SQL Server 2000	**org.hibernate.dialect.SQLServerDialect**
Microsoft SQL Server 2005	**org.hibernate.dialect.SQLServer2005Dialect**

RDBMS	ダイアレクト
Microsoft SQL Server 2008	**org.hibernate.dialect.SQLServer2008Dialect**
Microsoft SQL Server 2012	**org.hibernate.dialect.SQLServer2012Dialect**
Microsoft SQL Server 2014	**org.hibernate.dialect.SQLServer2012Dialect**
Microsoft SQL Server 2016	**org.hibernate.dialect.SQLServer2012Dialect**
MySQL5	**org.hibernate.dialect.MySQL5Dialect**
MySQL5.5	**org.hibernate.dialect.MySQL55Dialect**
MySQL5.7	**org.hibernate.dialect.MySQL57Dialect**
Oracle (すべてのバージョン)	**org.hibernate.dialect.OracleDialect**
Oracle 9i	**org.hibernate.dialect.Oracle9iDialect**
Oracle 10g	**org.hibernate.dialect.Oracle10gDialect**
Oracle 11g	**org.hibernate.dialect.Oracle10gDialect**
Oracle 12c	**org.hibernate.dialect.Oracle12cDialect**
Pointbase	**org.hibernate.dialect.PointbaseDialect**
PostgreSQL	**org.hibernate.dialect.PostgreSQLDialect**
PostgreSQL 9.2	**org.hibernate.dialect.PostgreSQL9Dialect**
PostgreSQL 9.3	**org.hibernate.dialect.PostgreSQL9Dialect**
PostgreSQL 9.4	**org.hibernate.dialect.PostgreSQL94Dialect**
Postgres Plus Advanced Server	**org.hibernate.dialect.PostgresPlusDialect**
Progress	**org.hibernate.dialect.ProgressDialect**
SAP DB	**org.hibernate.dialect.SAPDBDialect**
Sybase	**org.hibernate.dialect.SybaseASE15Dialect**
Sybase 15.7	**org.hibernate.dialect.SybaseASE157Dialect**
Sybase 16	**org.hibernate.dialect.SybaseASE157Dialect**

RDBMS	ダイアレクト
Sybase Anywhere	**org.hibernate.dialect.SybaseAnywhereDialect**

重要

hibernate.dialect プロパティーは、アプリケーションデータベースの適切な **org.hibernate. denialect.Dialect** サブクラスに設定する必要があります。ダイアレクトが指定されている場合、Hibernate は他のプロパティーの一部に対して適切なデフォルトを使用します。これは、手動で指定する必要がないことを意味します。

Revised on 2023-01-28 12:52:19 +1000