JBoss Enterprise Application Platform 5

Hibernate Search リファレンスガイド

JBoss Enterprise Application Platform 5 向け

エディッション 5.1.2

Last Updated: 2017-10-26

JBoss Enterprise Application Platform 5 Hibernate Search リファレンスガイド

JBoss Enterprise Application Platform 5 向け
エディッション 5.1.2

Red Hat ドキュメンテーショングループ

法律上の通知

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

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, 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 Software Collections 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

JBoss Enterprise Application Platform 5 およびそのパッチリリース向けの Hibernate Search リファレンスガイドです。

目次

第1章 はじめに **4**

1.1. システム要求 4
1.2. MAVEN の使用 4
1.3. 設定 5
1.4. インデックス化 9
1.5. 検索 10
1.6. アナライザ 11
1.7. 次の作業 12

第2章 アーキテクチャ **13**

2.1. 概要 13
2.2. バックエンド 13
2.2.1. バックエンドタイプ 14
2.2.1.1. Lucene 14
2.2.1.2. JMS 14
2.2.2. 作業の実行 15
2.2.2.1. 同期 15
2.2.2.2. 非同期 15
2.3. リーダー方針 15
2.3.1. 共有 15
2.3.2. 非共有 15
2.3.3. カスタム 16

第3章 設定 **17**

3.1. DIRECTORY 構成 17
3.2. インデックスの分割 19
3.3. インデックスの共有 (同じディレクトリに 2 つのエンティティ) 21
3.4. ワーカーの設定 21
3.5. JMS マスター/スレーブの設定 22
3.5.1. スレーブノード 22
3.5.2. マスターノード 23
3.6. リーダー方針の設定 24
3.7. HIBERNATE SEARCH および自動インデックスを有効化 25
3.7.1. Hibernate Search の有効化 25
3.7.2. 自動インデックス化 26
3.8. LUCENE インデックス化パフォーマンスのチューニング 26
3.9. LOCKFACTORY の設定 30

第4章 インデックス構造にエンティティをマッピング **32**

4.1. エンティティのマッピング 32
4.1.1. 基本的なマッピング 32
4.1.2. プロパティを複数回マッピング 34
4.1.3. 組込みおよび関連付けられたオブジェクト 35
4.1.4. ブーストファクタ 38
4.1.5. 動的ブーストファクタ 39
4.1.6. アナライザ 40
4.1.6.1. アナライザ定義 41
4.1.6.2. 利用可能なアナライザ 43
4.1.6.3. アナライザ判別子 (実験段階) 44
4.1.6.4. アナライザの取得 45
4.2. プロパティ／フィールドのブリッジ 46
4.2.1. ビルトインのブリッジ 46

4.2.2. カスタムのブリッジ 47
4.2.2.1. StringBridge 48
4.2.2.2. FieldBridge 50
4.2.2.3. ClassBridge 51
4.3. 独自の ID の提供 52
4.3.1. ProvidedId アノテーション 53

第5章 クエリ 54
5.1. クエリの構築 55
5.1.1. Lucene クエリの構築 55
5.1.2. Hibernate Search クエリの構築 55
5.1.2.1. 概要 55
5.1.2.2. ページ処理 56
5.1.2.3. ソート 56
5.1.2.4. フェッチ方針 56
5.1.2.5. プロジェクション 57
5.2. 結果の取得 58
5.2.1. パフォーマンスに関する考慮事項 58
5.2.2. 結果サイズ 59
5.2.3. ResultTransformer 59
5.2.4. 結果の理解 60
5.3. フィルタ 61
5.4. クエリプロセスの最適化 65
5.5. ネイティブ LUCENE クエリ 65

第6章 手動インデックス化 66
6.1. インデックス化 66
6.2. パージ 67

第7章 インデックスの最適化 69
7.1. 自動最適化 69
7.2. 手動最適化 69
7.3. 最適化の調整 70

第8章 高度な機能 71
8.1. SEARCHFACTORY 71
8.2. LUCENE ディレクトリへのアクセス 71
8.3. INDEXREADER の使用 71
8.4. LUCENE のスコア式のカスタマイズ 72

付録A 改訂履歴 74

第1章 はじめに

Hibernate Search へようこそ! この章では、Hibernate Search を既存の Hibernate 対応アプリケーションに統合するのに必要な最初の手順について説明します。Hibernate を初めて使用する場合は、 ここ から始めることを推奨します。

1.1. システム要求

表1.1 システム要求

Java ランタイム	JDK または JRE バージョン **5** 以上。Java Runtime for Windows/Linux/Solaris はここからダウンロードできます。
Hibernate Search	Hibernate Search ディストリビューションの **`lib`** ディレクトリにある **`hibernate-search.jar`** とすべてのランタイム依存関係。必要な依存関係を理解するには、lib ディレクトリにある **`README.txt`** を参照してください。
Hibernate Core	この手順は Hibernate 3.3.x に対してテストされました。**`hibernate-core.jar`** とディストリビューションの **`lib`** ディレクトリにある推移的な依存関係が必要です。最小ラインタイム要件を調べるには、**`lib`** ディレクトリにある **`README.txt`** を参照してください。
Hibernate Annotations	Hibernate Search は Hibernate Annotations なしで使用できますが、以下の手順では基本的なエンティティ設定 (**@Entity, @Id, @OneToMany,...**) のために Hibernate Annotations を使用します。設定のこの部分は xml またはコードでも記述できます。ただし、Hibernate Search 自体は、別の設定が存在しない独自のアノテーションセット (**@Indexed, @DocumentId, @Field,...**) を持ちます。チュートリアルは Hibernate Annotations のバージョン 3.4.x に対してテストされました。

すべての依存関係は Hibernate ダウンロードサイトからダウンロードできます。また、Hibernate Compatibility Matrix に対して依存関係のバージョンを検証することもできます。

1.2. MAVEN の使用

すべての依存関係を手動で管理する代わりに、maven ユーザーは JBoss maven repository を使用できます。JBoss レポジトリ url を **`pom.xml`** または **`settings.xml`** の **repositories** セクションに追加します。

例1.1 `settings.xml` への JBoss maven レポジトリの追加

```
<repository>
  <id>repository.jboss.org</id>
  <name>JBoss Maven Repository</name>
```

```
  <url>http://repository.jboss.org/maven2</url>
  <layout>default</layout>
</repository>
```

次に、以下の依存関係を pom.xml に追加します。

例1.2 Hibernate Search 用の Maven 依存関係

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search</artifactId>
   <version>3.1.0.GA</version>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-annotations</artifactId>
   <version>3.4.0.GA</version>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>3.4.0.GA</version>
</dependency>
<dependency>
   <groupId>org.apache.solr</groupId>
   <artifactId>solr-common</artifactId>
   <version>1.3.0</version>
</dependency>
<dependency>
   <groupId>org.apache.solr</groupId>
   <artifactId>solr-core</artifactId>
   <version>1.3.0</version>
</dependency>
<dependency>
   <groupId>org.apache.lucene</groupId>
   <artifactId>lucene-snowball</artifactId>
   <version>2.4.0</version>
</dependency>
```

すべての依存関係が必要なわけではありません。**hibernate-search** 依存関係のみが必須です。この依存関係は (必要な推移的な依存関係はとともに) Hibernate Search を使用するのに必要なすべてのクラスを含みます。**hibernate-annotations** は、このチュートリアルで行うようにアノテーションを使用してドメインモデルを設定する場合のみ必要です。ただし、Hibernate Annotations を使用しない場合であっても、hibernate-search jar ファイルに同梱される Hibernate Search 固有のアノテーションを使用して Lucene インデックスを設定する必要があります。現在、Hibernate Search に利用可能な XML 設定はありません。**hibernate-entitymanager** は、Hibernate Search を JPA とともに使用する場合に必要です。Solr 依存関係は、Solr のアナライザフレームワークを使用する場合に必要です。この詳細については、後ほど述べます。最後に、**`lucene-snowball`** 依存関係は、Lucene のスノーボールステマーを使用する場合に必要です。

1.3. 設定

必要なすべての依存関係をダウンロードし、アプリケーションに追加したら、いくつのプロパティを hibernate 設定ファイルに追加する必要があります。Hibernate を直接使用している場合、これは **hibernate.properties** または **hibernate.cfg.xml** で実行できます。JPA から Hibernate を使用している場合は、プロパティを **persistence.xml** に追加することもできます。標準的な使用では、ほとんどのプロパティが適切なデフォルト値を提供します。サンプルの **persistence.xml** 設定は以下のようになります。

例1.3 hibernate.properties、hibernate.cfg.xml、または persistence.xml に追加される基本的な設定オプション

```
...
<property name="hibernate.search.default.directory_provider"
   value="org.hibernate.search.store.FSDirectoryProvider"/>

<property name="hibernate.search.default.indexBase"
value="/var/lucene/indexes"/>
...
```

最初に、使用する **DirectoryProvider** を Hibernate Search に指示する必要があります。これは、**hibernate.search.default.directory_provider** プロパティを設定することにより達成できます。Apache Lucene には インデックスファイルを格納する **Directory** の概念があります。Hibernate Search は **DirectoryProvider** を使用して Lucene **Directory** インスタンスの初期化と設定を処理します。このチュートリアルでは、**FSDirectoryProvider** という名前の **DirectoryProvider** のサブクラスを使用します。これにより、Hibernate Search によって作成された Lucene インデックスを物理的に検査できます (たとえば、Luke を使用)。稼働する設定がある場合は、他のディレクトリプロバイダを実験できます (「Directory 構成」 を参照)。ディレクトリプロバイダの次は、 **hibernate.search.default.indexBase** を使用してすべてのインデックスに対してデフォルトのルートディレクトリを指定する必要があります。

アプリケーションに Hibernate により管理されたクラス **example.Book** と **example.Author** が含まれ、データベースに含まれた書籍を検索するためにフリーテキスト検索機能をアプリケーションに追加したいとします。

例1.4 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 Book と Author の例

```
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
    ...
}
```

これを達成するには、いくつかのアノテーションを **Book** および **Author** クラスに追加する必要があります。最初のアノテーション **@Indexed** は **Book** をインデックス化可能としてマークします。設計により、Hibernate Search は特定のエンティティのインデックスの単一性を確保するためにインデックスに非トークン化 id を格納する必要があります。**@DocumentId** は、このために使用するプロパティをマークし、ほとんどの場合はデータベース一次キーと同じです。実際には、Hibernate Search の 3.1.0 リリース以降、**@DocumentId** はオプションです (**@Id** アノテーションが存在します)。

次に、検索可能にするフィールドをマークする必要があります。まず、**title** と **subtitle** を **@Field** でアノテートします。パラメータ **index=Index.TOKENIZED** により、テキストはデフォルトの Lucene アナライザを使用してトークン化されます。通常は、トークン化とは文を個々の単語に分割し、場合によっては **'a'** や '**the**' などの共通の単語を除外することを意味します。アナライザについては、後ほど詳しく説明します。 **@Field**、 **store=Store.NO** 内で指定する2 つ目のパラメータは、実際のデータがインデックスに格納されないようにします。このデータがインデックスに格納されるかどうかは、その検索機能と関係ありません。Lucene の観点から、インデックスの作成後はデータを保持する必要がありません。これを格納する利点は、プロジェクション (「プロジェクション」) を使用して取得できることです。

プロジェクションがない場合、Hibernate Search はクエリ基準に一致するエンティティのデータベース ID を見つけるためにデフォルトで Lucene クエリを実行し、これらの ID を使用して管理されたオブジェクトをデータベースから取得します。プロジェクションを使用するかしないかの決定は、ケースバイケースで行う必要があります。管理されたオブジェクトを返すためデフォルトの動作 **Store.NO** が推奨されます (プロジェクションはオブジェクトアレイのみ返します)。

次に、**Book** クラスのアノテートに戻ってみましょう。まだ説明してないアノテーションは **@DateBridge** です。このアノテーションは、Hibernate Search のビルトインフィールドブリッジの 1 つです。Lucene インデックスは純粋に文字列ベースです。このため、Hibernate Search はインデックス化されたフィールドのデータタイプを文字列に変換する必要があります (また、文字列をフィールド

のデータタイプに変換する必要があります)。事前に定義されたさまざまなブリッジ (**java.util.Date** を指定されたレゾリューションを持つ **String** に変換する **DateBridge** を含む) が提供されます。詳細については、「プロパティ／フィールドのブリッジ」 を参照してください。

最後に **@IndexedEmbedded.** が残りました。このアノテーションは関連付けられたエンティティ (**@ManyToMany**、**@*ToOne**、および **@Embedded**) を所有エンティティの一部として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何も知らないフラットなデータ構造であるため必要です。著者名が検索可能になるように、著者名は書籍自体の一部としてインデックス化する必要があります。また、**@IndexedEmbedded** 上で、**@Indexed** を使用してインデックスに含める関連付けられたエンティティのすべてのフィールドをマークする必要があります。詳細については、「組込みおよび関連付けられたオブジェクト」 を参照してください。

この時点でこれらの設定は十分なはずです。エンティティマッピングの詳細については、「エンティティのマッピング」 を参照してください。

例1.5 Hibernate Search アノテーション追加後のサンプルエンティティ

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

  @Id
  @GeneratedValue
  @DocumentId
  private Integer id;

  @Field(index=Index.TOKENIZED, store=Store.NO)
  private String title;

  @Field(index=Index.TOKENIZED, store=Store.NO)
  private String subtitle;

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

 @Field(index = Index.UN_TOKENIZED, store = Store.YES)
@DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;

  public Book() {
  }

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

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

  @Id
```

```
    @GeneratedValue
    private Integer id;

    @Field(index=Index.TOKENIZED, store=Store.NO)
    private String name;

    public Author() {
    }

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

1.4. インデックス化

Hibernate Search は Hibernate Core により永続化、更新、または削除された各エンティティを透過的にインデックス化します。ただし、Lucene インデックスにデータベースにすでに存在するデータを入力するためにイニシャルインデックス化をトリガする必要があります。上記のプロパティとアノテーションを追加したら、書式のイニシャルバッチインデックスをトリガします。これを行うには、以下のコード断片のいずれかを使用します (6章 *手動インデックス化* も参照)。

例1.6 Hibernate Session を使用してデータをインデックス化

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

List books = session.createQuery("from Book as book").list();
for (Book book : books) {
    fullTextSession.index(book);
}

tx.commit(); //index is written at commit time
```

例1.7 JPA を使用してデータをインデックス化

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

List books = em.createQuery("select book from Book as
book").getResultList();
for (Book book : books) {
    fullTextEntityManager.index(book);
}

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

上記のコードの実行後に、Lucene インデックスが **/var/lucene/indexes/example.Book** 下にあるはずです。Luke を使用してこのインデックスを確認します。これにより、Hibernate Search がどのように動作するかを理解できます。

1.5. 検索

次に最初の検索を実行します。一般的な方法は、ネイティブの Lucene クエリを作成し、Hibernate API から使い慣れているすべての機能を取得するためにこのクエリを org.hibernate.Query にラップします。以下のコードはインデックス化されたフィールドに対してクエリを準備して実行し、**Book** のリストを返します。

例1.8 Hibernate Session を使用して検索を作成および実行

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

// create native Lucene query
String[] fields = new String[]{"title", "subtitle", "authors.name",
"publicationDate"};
MultiFieldQueryParser parser = new MultiFieldQueryParser(fields, new
StandardAnalyzer());
org.apache.lucene.search.Query query = parser.parse( "Java rocks!" );

// 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1.9 JPA を使用して検索を作成および実行

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

// create native Lucene query
String[] fields = new String[]{"title", "subtitle", "authors.name",
"publicationDate"};
MultiFieldQueryParser parser = new MultiFieldQueryParser(fields, new
StandardAnalyzer());
org.apache.lucene.search.Query query = parser.parse( "Java rocks!" );

// 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();
```

1.6. アナライザ

次にもう少し面白いことをしてみましょう。インデックス化された書籍タイトルのいずれかが "Refactoring: Improving the Design of Existing Code" であり、"refactor"、"refactors"、"refactored"、および "refactoring" のすべてのクエリでヒットしたいとします。Lucene では、これは**インデックス作成中**と検索処理中にワードステミングを適用するアナライザクラスを選択して実現できます。Hibernate Search は、使用するアナライザを設定する複数の方法を提供します (「アナライザ」 を参照)。

- 設定ファイルでの **`hibernate.search.analyzer`** プロパティの設定。指定されたクラスはデフォルトのアナライザです。
- エンティティレベルでの **`@Analyzer`** アノテーションの設定。
- フィールドレベルでの **`@Analyzer`** アノテーションの設定。

`@Analyzer` アノテーションを使用する場合は、使用するアナライザの完全修飾クラス名を指定したり、**`@AnalyzerDef`** アノテーションにより定義されたアナライザ定義を参照したりできます。アナライザ定義を参照する場合は、ファクトリを使用する Solr アナライザフレームワークが使用されます。利用可能なファクトリクラスの詳細については、Solr JavaDoc または Solr Wiki の対応するセクションを参照してください。選択されたファクトリに応じて、Solr 依存関係の上部に追加のライブラリが必要になることがあります。たとえば、**`PhoneticFilterFactory`** は commons-codec に依存します。

以下の例では、**`StandardTokenizerFactory`** が使用され、その後に **`LowerCaseFilterFactory`** と **`SnowballPorterFilterFactory`** の 2 つのフィルタファクトリが続きます。標準的なトークナイザは単語を句読点とハイフンで分割します (電子メールアドレスとインターネットホスト名はそのまま保持されます)。これは優れた汎用的なトークナイザです。小文字のフィルタは各トークンの文字を小文字にし、スノーボールフィルタは言語固有のステミングを適用します。

一般的に、Solr フレームワークを使用する場合は、最初にトークナイザを使用し、次に任意の数のフィルタを使用する必要があります。

例1.10 `@AnalyzerDef` および Solr フレームワークを使用してアナライザを定義および使用

```
package example;
...
@Entity
@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 {

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

```
    @Field(index=Index.TOKENIZED, store=Store.NO)
    @Analyzer(definition = "customanalyzer")
    private String title;

    @Field(index=Index.TOKENIZED, store=Store.NO)
    @Analyzer(definition = "customanalyzer")
    private String subtitle;

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

    @Field(index = Index.UN_TOKENIZED, store = Store.YES)
    @DateBridge(resolution = Resolution.DAY)
    private Date publicationDate;

    public Book() {
    }

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

1.7. 次の作業

上記のパラグラフでは、Hibernate Search の概要について説明しました。maven アーキタイププラグインと以下のコマンドを使用すると、このチュートリアルのサンプルコードが入力された初期実行可能な maven プロジェクト構造を作成できます。

例1.11 maven アーキタイプを使用してチュートリアルソースを作成

```
mvn archetype:create \
    -DarchetypeGroupId=org.hibernate \
    -DarchetypeArtifactId=hibernate-search-quickstart \
    -DarchetypeVersion=3.1.0.GA \
    -DgroupId=my.company -DartifactId=quickstart
```

maven プロジェクトを使用すると、サンプルを実行し、ファイルシステムベースのインデックスを検査し、管理されたオブジェクトを検索および取得できます。**mvn package** を実行するだけで、ソースをコンパイルし、ユニットテストを実行できます。

このチュートリアルの後は、Hibernate Search の全体的なアーキテクチャ (2章 アーキテクチャ) と基本的な機能について詳しく説明します。このチュートリアルで簡単にしか触れられていないトピックはアナライザ設定 (「アナライザ」) とフィールドブリッジ (「プロパティ／フィールドのブリッジ」) です。これら 2 つの重要な機能は細かい単位のインデックス化が必要です。より高度なトピックでは、クラスタリング (「JMS マスター/スレーブの設定」) と大規模なインデックス処理 (「インデックスの分割」) について説明します。

第2章 アーキテクチャ

2.1. 概要

Hibernate Search はインデックス化コンポーネントとインデックス検索コンポーネントで構成されます。 いずれも Apache Lucene で支えられています。

エンティティがデータベースに対して挿入、更新または削除されるたびに、Hibernate Search はこのイベントを (Hibernate イベントシステムを使用して) 追跡し、インデックス更新をスケジュールします。すべてのインデックス更新は Apache Lucene API を使用せずに処理されます (「Hibernate Search および自動インデックスを有効化」 を参照)。

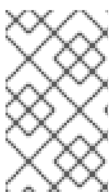
Apache Lucene のインデックスと交信するために、 Hibernate Search には **`DirectoryProvider`** に関する概念があります。 ディレクトリプロバイダは特定の Lucene **`Directory`** タイプを管理します。ディレクトリプロバイダを設定してディレクトリターゲットを調整することができます (「Directory 構成」 を参照)。

また、Hibernate Search は Lucene インデックスを使用してエンティティを検索し管理エンティティのリストを返すのでめんどうなオブジェクトと Lucene ドキュメントのマッピングを行わなくてすみます。 同じ永続コンテキストは Hibernate と Hibernate Search で共有されます。実際には、**`FullTextSession`** は Hibernate Session 上に構築されるため、アプリケーションコードは統一された **`org.hibernate.Query`** または **`javax.persistence.Query`** を HQL、JPA-QL、 またはネイティブのクエリが行うのと全く同じ方法で使用できます。

効率を上げるために、Hibernate Search は、Lucene インデックスとの書き込み対話をバッチ処理します。期待される範囲に応じて 2 つのバッチ処理タイプが存在します。トランザクション外で、実際のデータベース操作の直後にインデックス更新操作が実行されます。この範囲は実際には設定されず、バッチ処理も実行されません。ただし、データベースと Hibernate Search のいずれの場合でもトランザクション内でオペレーションを実行することをお勧めします (JDBC または JTA)。 トランザクション内の場合、 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され、トランザクションがロールバックする場合は破棄されます。バッチ処理範囲はトランザクションです。以下の 2 つの即時的な利点があります。

- パフォーマンス: バッチで操作を実行したとき Lucene のインデックス化は向上します。
- ACIDity: 実行される作業はデータベーストランザクションにより実行される作業と同じ範囲を持ち、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、厳密には ACID ではありませんが、ACID の動作は完全テキスト検索インデックスにはほとんど役に立ちません (インデックスをソースからいつでも再構築できるため)。

これら 2 つの範囲は (有名な) オートコミットとトランザクション動作の関係と同じと考えることができます (範囲なしとトランザクション)。パフォーマンスの観点からは、**トランザクション**モードが推奨されます。範囲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、範囲を調整します。



注記

Hibernate Search は Hibernate/EntityManager の長い会話 (アトミック会話) で適切に動作します。また、ユーザーの要求に応じて、範囲の追加が考慮されます (接続可能メカニズムがすでに動作)。

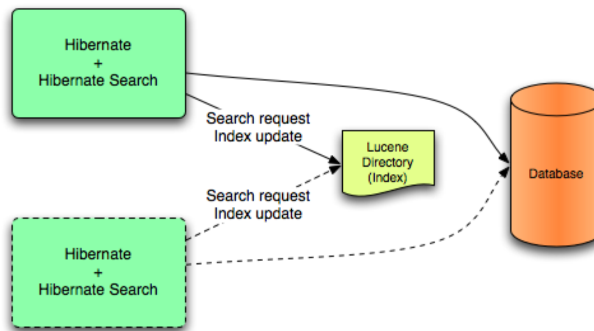
2.2. バックエンド

Hibernate Search では、範囲作業を異なるバックエンドにより処理できます。2 つの異なるシナリオ向けに 2 つのバックエンドがデフォルトで提供されます。

2.2.1. バックエンドタイプ

2.2.1.1. Lucene

このモードでは、特定のノード (JVM) のすべてのインデックス更新操作が同じノードにより Lucene ディレクトリに対して (ディレクトリプロバイダを使用して) 実行されます。このモードは、通常非クラスタ環境またはディレクトリストアが共有されたクラスタ環境で使用されます。



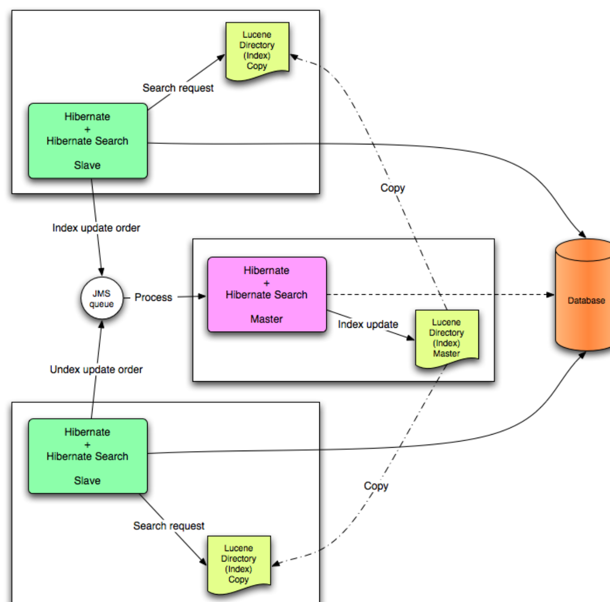
Lucene バックエンド設定。

このモードは非クラスタアプリケーションまたは、ディレクトリがロック方針を管理するクラスタアプリケーションを対象とします。

主な利点は Lucene クエリの単純さと変更の即時確認です (一部のアプリケーションの要件)。

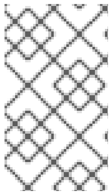
2.2.1.2. JMS

特定のノードに適用されたすべてのインデックス更新操作は JMS 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通常、スレーブコピーにレプリケートされます。これは、マスター/スレーブパターンとして知られています。マスターは Lucene インデックスのみを管理します。スレーブは読み取りおよび書き込み操作を受け取ることができます。ただし、ローカルインデックスコピーの読み取り操作のみが処理され、更新操作はマスターに委譲されます。



JMS バックエンド構成

このモードはスループットが重要なクラスタ環境を対象とし、インデックス更新遅延を受け入れることができます。安定性は JMS プロバイダにより、スレーブをインデックスのローカルコピーで動作することによって確保されます。



注記

Hibernate Search は拡張可能なアーキテクチャです。他のサードパーティバックエンドについてご意見がある場合は、 **`hibernate-dev@lists.jboss.org`** までご連絡ください。

2.2.2. 作業の実行

インデックス化作業 (バックエンドにより実行される) はトランザクションコミットにより同期的または非同期的に実行できます (トランザクション外の場合は更新操作)。

2.2.2.1. 同期

これは、トランザクションコミットに合わせてバックエンド作業が実行されるセーフモードです。同時性が高い環境では、これによりスループットの制限が発生し (Apache Lucene ロックメカニズムが原因)、バックエンドがトランザクションプロセスよりも大幅に低速であり、たくさんの IO 操作が関係する場合にシステム応答時間を増加できます。

2.2.2.2. 非同期

このモードはバックエンドにより実行された作業を異なるスレッドに委譲します。この結果、スループットと応答時間がバックエンドパフォーマンスから (ある程度) 切り離されます。欠点は、トランザクションコミット時とインデックス更新時の間に小さな遅延が発生し、スレッド管理に対応するために小さなオーバーヘッドが発生することです。

パフォーマンスの問題が発生し、適切なベンチマークを設定した後は、同期実行を最初に使用し、非同期実行を評価することが推奨されます (つまり、まったく非現実的な方法でシステムを使用)。

2.3. リーダー方針

クエリを実行する場合、Hibernate Search はリーダー方針を使用して Apache Lucene インデックスと対話します。リーダー方針の選択は、アプリケーションのプロファイル (頻繁な更新、ほとんどが読み取り、非同期インデックス更新など) に基づきます。「リーダー方針の設定」 も参照してください。

2.3.1. 共有

この方針では、**`IndexReader`** が最新の場合に、Hibernate Search が特定の Lucene に対して複数のクエリとスレッドで同じ **`IndexReader`** を共有します。**`IndexReader`** が最新でない場合は、新しい **`IndexReader`** がオープンされ、提供されます。各 **`IndexReader`** は複数の **`SegmentReader`** から構成されます。この方針は最後のオープン後に変更または作成されたセグメントのみを再オープンし、以前のインスタンスからすでにロードされたセグメントを共有します。この方針はデフォルトです。

この方針の名前は **`shared`** です。

2.3.2. 非共有

クエリが実行されるたびに、Lucene **`IndexReader`** がオープンされます。**`IndexReader`** のオープンやウォームアップは比較的コストが高い操作であるため、この方針は最も効率的とはいえません。

この方針の名前は **`not-shared`** です。

2.3.3. カスタム

`org.hibernate.search.reader.ReaderProvider` を実装することにより、アプリケーションのニーズを満たす独自のリーダー方針を記述できます。この実装はスレッドセーフである必要があります。

第3章 設定

3.1. DIRECTORY 構成

Apache Lucene には、インデックスファイルを保存する **Directory** の表記が存在します。**Directory** 実装はカスタマイズできますが、Lucene はファイルシステム (**FSDirectoryProvider**) とメモリ内 (**RAMDirectoryProvider**) 実装でバンドルされます。**DirectoryProvider** は Lucene **Directory** を Hibernate Search により抽象化し、基礎となる Lucene リソースの設定と初期化を処理します。表3.1「ビルトインの Directory Provider 一覧」 は Hibernate Search でバンドルされるディレクトリプロバイダのリストを示します。

表3.1 ビルトインの Directory Provider 一覧

クラス	説明	プロパティ
org.hibernate.search.store.RAMDirectoryProvider	メモリベースのディレクトリ。ディレクトリは **@Indexed.index** エレメントにより一意に (同じデプロイメント単位で) 識別されます。	none
org.hibernate.search.store.FSDirectoryProvider	ファイルシステムベースのディレクトリ。使用されるディレクトリは <indexBase>/< indexName > です。	**indexBase** : ベースディレクトリ **indexName**: @Indexed.index をオーバーライドします (共有されたインデックスの場合に役に立ちます)。 **locking_strategy** : オプション。「LockFactory の設定」を参照してください。

クラス	説明	プロパティ
org.hibernate.search.store.FSMasterDirectoryProvider	ファイルシステムベースのディレクトリ。FSDirectoryProvider と同様。インデックスを定期的にソースディレクトリ (コピーディレクトリとも呼ばれます) にコピーします。 更新期間の推奨値は情報をコピーする時間よりも大きい (最低) 50% です (デフォルトの 3600 秒 - 60 分)。 コピーは平均コピー時間を短縮する差分コピーメカニズムに基づいています。 DirectoryProvider は通常、JMS バックエンドクラスタのマスターノードで使用されます。 **buffer_size_on_copy** の最適化は、オペレーティングシステムと利用可能な RAM に依存します。16 〜 64MB の値を使用した場合に良い結果が得られると報告したユーザーがほとんどです。	**indexBase**: ベースのディレクトリ **indexName**: @Indexed.index をオーバーライドします (共有されたインデックスの場合に役に立ちます)。 **sourceBase**: ソース (コピー) ベースディレクトリ **source**: ソースディレクトリのサフィックス (デフォルト値は **@Indexed.index**)。実際のソースディレクトリ名は **<sourceBase>/<source>** です。 **refresh**: 秒単位の更新期間 (更新期間ごとにコピーが行われます) **buffer_size_on_copy**: 単一ローレベルコピー命令で移動する MegaBytes の量。デフォルト値は 16MB です。 **locking_strategy** : オプション。「LockFactory の設定」を参照してください。
org.hibernate.search.store.FSSlaveDirectoryProvider	ファイルシステムベースのディレクトリ。FSDirectoryProvider と似ていますが通常でマスターバージョン (ソース) を取得します。ロックと不整合な検索結果を回避するために、2 つのローカルコピーが保持されます。 更新期間の推奨値は情報をコピーする時間よりも大きい (最低) 50% です (デフォルトの 3600 秒 - 60 分)。 コピーは平均コピー時間を短縮する差分コピーメカニズムに基づいています。 DirectoryProvider は通常、JMS バックエンドを使用したスレーブノードで使用されます。 **buffer_size_on_copy** の最適化は、オペレーティングシステムと利用可能な RAM に依存します。16 〜 64MB の値を使用した場合に良い結果が得られると報告したユーザーがほとんどです。	**indexBase**: ベースのディレクトリ **indexName**: @Indexed.index をオーバーライドします (共有されたインデックスの場合に役に立ちます)。 **sourceBase**: ソース (コピー) ベースディレクトリ **source**: ソースディレクトリのサフィックス (デフォルト値は **@Indexed.index**)。実際のソースディレクトリ名は **<sourceBase>/<source>** です。 **refresh**: 秒単位の更新期間 (更新期間ごとにコピーが行われます) **buffer_size_on_copy**: 単一ローレベルコピー命令で移動する MegaBytes の量。デフォルト値は 16MB です。 **locking_strategy** : オプション。「LockFactory の設定」を参照してください。

ビルトインのディレクトリプロバイダがニーズに適合しない場合は **org.hibernate.store.DirectoryProvider** インターフェースを実装することで独自のディレクトリプロバイダを記述することができます。

インデックス化された各エンティティは Lucene インデックスに関連付けられます (インデックスは複数のエンティティで共有可能ですが通常は不要です)。 **hibernate.search.***indexname* でプレフィックスされるプロパティでインデックスを設定することができます。 全インデックスに継承されるデフォルトのプロパティはプレフィックスの **hibernate.search.default.** を使用して定義することができます。

特定のインデックスのディレクトリプロバイダを定義するには **hibernate.search.*indexname*.directory_provider** を使用します。

例3.1 ディレクトリプロバイダの設定

```
hibernate.search.default.directory_provider
org.hibernate.search.store.FSDirectoryProvider
hibernate.search.default.indexBase=/usr/lucene/indexes
hibernate.search.Rules.directory_provider
org.hibernate.search.store.RAMDirectoryProvider
```

以下に適用します。

例3.2 @Indexed の index パラメータを使用したインデックス名の指定

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

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

すると、 Status エンティティがインデックス化される **/usr/lucene/indexes/Status** 内にファイルシステムディレクトリを作成し、 Rule エンティティがインデックス化される **Rules** という名前のインメモリディレクトリを使用します。

ディレクトリプロバイダやベースディレクトリのような共通のルールを容易に定義して、 あとでこれらのデフォルト値をインデックスごとに上書きすることができます。

独自の **DirectoryProvider** を記述すると、 この設定メカニズムを利用することもできます。

3.2. インデックスの分割

巨大なインデックス (サイズ) が関係する一部の極端な状況では、特定のエンティティタイプのインデックスデータを複数の Lucene インデックスに分割することが必要になります。このソリューションの使用は、インデックスのサイズが巨大になり、インデックスの更新によりアプリケーションの稼働に影響がでるまで推奨されません。インデックス分割の主な欠点は 1 つの検索でより多くのファイルを開く必要があるため、検索が低速になることです。つまり、この作業は問題が発生するまで行わないでください。

この強い警告にも関わらず、Hibernate Search では特定のエンティティタイプを複数のサブインデックスにインデックス化できます。**IndexShardingStrategy** により、データは異なるサブインデックス

に分割されます。デフォルトでは、分割の数が設定されない限り、分割方針は有効になりません。分割の数を設定するには、次のプロパティを使用します。

例3.3 特定のインデックスに対して nbr_of_shards を指定することによりインデックスの分割を有効化

```
hibernate.search.<indexName>.sharding_strategy.nbr_of_shards 5
```

この場合は 5 つの異なる分割を使用します。

デフォルトの分割方針 (分割が設定された場合) では、id 文字列表記のハッシュ値 (Field Bridge により生成される) に従ってデータが分割されます。これにより非常にバランスの取れた分割が保証されます。方針は、**`IndexShardingStrategy`** を実装し、以下のプロパティを設定することにより変えることができます。

例3.4 カスタム分割方針の指定

```
hibernate.search.<indexName>.sharding_strategy
my.shardingstrategy.Implementation
```

「Directory 構成」 で示されたように、各分割は独立したディレクトリプロバイダ設定を持ちます。以前の例の DirectoryProvider デフォルト名は **`<indexName>.0`** ～ **`<indexName>.4`** です。つまり、各分割は独自のインデックスの名前の後に **`.`** (ドット) とインデックス番号が付いた名前を持ちます。

例3.5 サンプルのエンティティ `Animal` に対する分割設定の構成

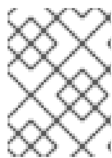
```
hibernate.search.default.indexBase /usr/lucene/indexes

hibernate.search.Animal.sharding_strategy.nbr_of_shards 5
hibernate.search.Animal.directory_provider
org.hibernate.search.store.FSDirectoryProvider
hibernate.search.Animal.0.indexName Animal00
hibernate.search.Animal.3.indexBase /usr/lucene/sharded
hibernate.search.Animal.3.indexName Animal03
```

この設定はデフォルトの id 文字列ハッシュ方針を使用し、Animal インデックスを 5 つのサブインデックスに分割します。すべてのサブインデックスは **`FSDirectoryProvider`** インスタンスであり、各サブインデックスが格納されるディレクトリは以下のとおりです。

- サブインデックス 0 用: /usr/lucene/indexes/Animal00 (共有された indexBase、上書きされた indexName)
- サブインデックス 1 用: /usr/lucene/indexes/Animal.1 (共有された indexBase、デフォルトの indexName)
- サブインデックス 2 用: /usr/lucene/indexes/Animal.2 (共有された indexBase、デフォルトの indexName)
- サブインデックス 3 用: /usr/lucene/shared/Animal03 (上書きされた indexBase、上書きされた indexName)

- サブインデックス 4 用: /usr/lucene/indexes/Animal.4 (共有された indexBase、デフォルトの indexName)

3.3. インデックスの共有 (同じディレクトリに 2 つのエンティティ)



注記

ここでは、このオプションが利用可能であることのみを示しています。実際には、インデックスの共有にはそれほど利点がありません。

複数のエンティティの情報を単一の Lucene インデックスに格納することは技術的には可能です。これを行うには 2 つの方法があります。

- 基礎となるディレクトリプロバイダが同じ物理インデックスディレクトリを参照するよう設定します。実際には、プロパティ **hibernate.search.[fully qualified entity name].indexName** に同じ値を設定します。例として、**Furniture** と **Animal** エンティティに対して同じインデックス (ディレクトリ) を使用します。例 “Animal” の両方のエンティティに対して **indexName** を設定します。両方のエンティティは Animal ディレクトリに格納されます。

```
hibernate.search.org.hibernate.search.test.shards.Furniture.indexNam
e = Aninal
hibernate.search.org.hibernate.search.test.shards.Animal.indexName =
Aninal
```

- 同じ値にマージする、エンティティの **@Indexed** アノテーションの **index** 属性を設定します。**Animal** のすべてのインスタンスとともに再び **Furniture** インスタンスを **Animal** インデックスでインデックス化する場合は、**Animal** クラスと **Furniture** クラスの両方で **@Indexed(index=”Animal”)** を指定します。

3.4. ワーカーの設定

ワーカー設定を使用して Hibernate Search が Lucene とどのように対話するかを調整できます。この作業は Lucene ディレクトリに対して実行したり、後で処理するために JMS キューに送信したりできます。Lucene ディレクトリに対して処理された場合、作業はトランザクションコミットに対して同期的または非同期的に処理できます。

ワーカーの設定は以下のプロパティを使用して定義できます。

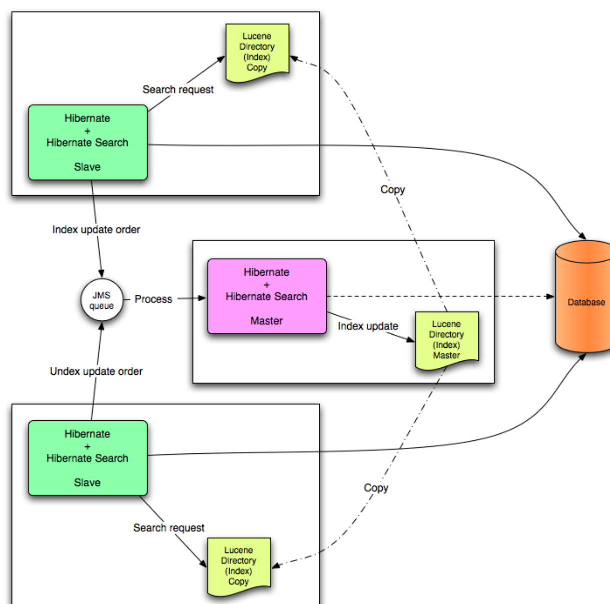
表3.2 ワーカーの設定

プロパティ	説明
hibernate.search.worker.backend	Apache Lucene バックエンドと JMS バックエンドはデフォルトでサポートされます。デフォルト値は **lucene** です。**jms** もサポートされます。
hibernate.search.worker.execution	同期および非同期実行をサポートします。デフォルト値は **sync** です。**async** もサポートされます。

hibernate.search.worker.thread_pool.size	プール内のスレッド数を定義します。非同期実行の場合のみ役に立ちます。デフォルト値は 1 です。
hibernate.search.worker.buffer_queue.max	スレッドポールが枯渇した場合に作業キューの最大数を定義します。非同期実行の場合のみ役に立ちます。デフォルト値は無限です。制限に達したら、作業は主なスレッドにより実行されます。
hibernate.search.worker.jndi.*	InitialContext を初期化する JNDI プロパティを定義します (必要な場合)。JNDI は JMS バックエンドのみにより使用されます。
hibernate.search.worker.jms.connection_factory	JMS バックエンドには必須。JMS 接続ファクトリを検索する JNDI 名を定義します (JBoss AS ではデフォルトで **/ConnectionFactory**)
hibernate.search.worker.jms.queue	JMS バックエンドには必須。JMS キューを検索する JNDI 名を定義します。このキューは作業メッセージを転記するために使用されます。

3.5. JMS マスター/スレーブの設定

この項では、Hibernate Search アーキテクチャにおけるMaster / Slaves 設定の方法について詳しく説明します。



JMS マスター/スレーブアーキテクチャの概要

3.5.1. スレーブノード

各インデックスの更新操作は JMS キューに送信されます。インデックスのクエリ操作はローカルインデックスコピーで実行されます。

例3.6 JMS 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 =
org.hibernate.search.store.FSSlaveDirectoryProvider

## Backend configuration
hibernate.search.worker.backend = jms
hibernate.search.worker.jms.connection_factory = /ConnectionFactory
hibernate.search.worker.jms.queue = queue/hibernatesearch
#optional jndi configuration (check your JMS provider for more
information)

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

高速な検索結果を実現するためにファイルシステムのローカルコピーが推奨されます。

更新期間は期待された時間コピーよりも大きくなる必要があります。

3.5.2. マスターノード

各インデックスの更新操作は JMS 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。

例3.7 JMS マスターの設定

```
### 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 =
```

```
org.hibernate.search.store.FSMasterDirectoryProvider

## Backend configuration
#Backend is the default lucene one
```

更新期間は期待された時間コピーよりも大きくなる必要があります。

Hibernate Search フレームワーク設定以外に、メッセージ駆動 Bean を記述し、JMS を使用してインデックス作業キューを処理するよう設定する必要があります。

例3.8 インデックスキューを処理するメッセージ駆動 Bean

```
@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)
    }
}
```

このサンプルでは、Hibernate Search ソースコードで利用可能な抽象 JMS コントローラクラスから継承し、JavaEE 5 MDB を実装します。この実装はサンプルとして提供され、非 Java EE メッセージ駆動 Bean を使用するよう調整できます (ただし、より複雑になります)。**`getSession()`** と **`cleanSessionIfNeeded()`** の詳細については、 **`AbstractJMSHibernateSearchController`** の javadoc を参照してください。

3.6. リーダー方針の設定

さまざまなリーダー方針が「リーダー方針」で説明されています。デフォルトの方針は以下のとおりです。

- **`shared`**: 複数のクエリでインデックスリーダーを共有します。この方針は最も効率的です。
- **`not-shared`**: 各クエリのインデックスリーダーを作成します。

デフォルトのリーダー方針は **`shared`** です。これは調整できます。

```
hibernate.search.reader.strategy = not-shared
```

このプロパティスイッチを **not-shared** 方針に追加します。

または、カスタムリーダー方針を使用する場合:

```
hibernate.search.reader.strategy = my.corp.myapp.CustomReaderProvider
```

ここで、**my.corp.myapp.CustomReaderProvider** はカスタム方針実装です。

3.7. HIBERNATE SEARCH および自動インデックスを有効化

3.7.1. Hibernate Search の有効化

Hibernate Annotations または Hibernate EntityManager を使用している場合、Hibernate Search はデフォルトで有効になります。何らかの理由でこれを無効にする必要がある場合は、**hibernate.search.autoregister_listeners** を false に設定します。リスナーが有効な場合は、エンティティがインデックス化されない場合であってもパフォーマンスの影響はありません。

Hibernate Core の Hibernate Search を有効化するには (つまり、Hibernate Annotations を使用しない場合)、以下の 6 つの Hibernate イベントに対して **FullTextIndexEventListener** を追加します。

例3.9 FullTextIndexEventListener を設定して Hibernate Search を明示的に有効にする

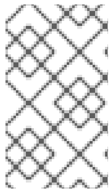
```
<hibernate-configuration>
     <session-factory>
        ...
        <event type="post-update"/>
            <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
        </event>
        <event type="post-insert"/>
            <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
        </event>
        <event type="post-delete"/>
            <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
        </event>
        <event type="post-collection-recreate"/>
            <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
        </event>
        <event type="post-collection-remove"/>
            <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
        </event>
        <event type="post-collection-update"/>
            <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

3.7.2. 自動インデックス化

デフォルトでは、Hibernate でオブジェクトが挿入、更新、または削除されるたびに、Hibernate Search が対応する Lucene インデックスを更新します。インデックスが読み取り専用の場合やインデックス更新がバッチで処理される場合 (*6章手動インデックス化* を参照) は、この機能を無効にすることが望まれることがあります。

イベントベースのインデックス化を無効にするには、次のように設定します。

```
hibernate.search.indexing_strategy manual
```



注記

ほとんどの場合、JMS バックエンドは、システムのすべての変更を追跡する軽量なイベントベースシステムと、異なるプロセスまたはマシンにより実行される重いインデックス化プロセスの 2 つの良い面を提供します。

3.8. LUCENE インデックス化パフォーマンスのチューニング

Hibernate Search では、基礎となる Lucene **IndexWriter** に渡されるパラメータセット (**mergeFactor**、**maxMergeDocs**、**maxBufferedDocs** など) を指定することにより Lucene インデックス化のパフォーマンスをチューニングできます。これらのパラメータは、すべてのインデックスに適用するデフォルト値、1 つのインデックスごとのデフォルト値、または 1 つの分割ごとのデフォルト値として指定できます。

使用状況に応じて異なるパフォーマンス設定に対して使用できる 2 つのパラメータセットが存在します。データベース変更によりトリガされたインデックス化操作の間に、パラメータは **transaction** キーワードによりグループ化されます。

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

インデックス化が **FullTextSession.index()** (*6章手動インデックス化* を参照) を使用して行われた場合、使用されるプロパティは **batch** キーワードによりグループ化されます。

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

対応する **.batch** プロパティが明示的に設定されない限り、値はデフォルトで **.transaction** プロパティに設定されます。特定の分割設定の **.batch** 値に対して値が設定されない場合、Hibernate Search は最初にインデックスセクション、次にデフォルトセクションを参照し、その後以下の順序で **.transaction** を検索します。

```
hibernate.search.Animals.2.indexwriter.transaction.max_merge_docs 10
hibernate.search.Animals.2.indexwriter.transaction.merge_factor 20
hibernate.search.default.indexwriter.batch.max_merge_docs 100
```

この設定は、Animals インデックスの共有された 2 つ目のものに適用される設定になります。

- **transaction.max_merge_docs** = 10
- **batch.max_merge_docs** = 100

- **transaction.merge_factor** = 20
- **batch.merge_factor** = 20

他のすべての値は Lucene で定義されたデフォルト値を使用します。

すべての値のデフォルト値は Lucene の独自のデフォルト値のままになるため、以下のテーブルのリストされた値は使用する Lucene のバージョンに依存します。示された値はバージョン **2.4** に相対的です。Lucene インデックス化のパフォーマンスの詳細については、Lucene ドキュメンテーションを参照してください。

表3.3 インデックス化パフォーマンスと動作プロパティのリスト

プロパティ	説明	デフォルト値
hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].max_buffered_delete_terms	バッファ化されたメモリ内の削除タームが適用され、フラッシュされる前に必要な削除タームの最小数を決定します。メモリ内にバッファ化されたドキュメントが存在する場合、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。	無効 (RAM の使用によりフラッシュ)
hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].max_buffered_docs	インデックス化の間にメモリ内にバッファ化されたドキュメントの量を制御します。量が多くなると、消費される RAM も多くなります。	無効 (RAM の使用によりフラッシュ)

プロパティ	説明	デフォルト値
`hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].max_field_length`	1 つのフィールドに対してインデックス化するタームの最大数。これにより、インデックス化に必要なメモリ量が制限され、データが非常に大きい場合にメモリ不足によりインデックス化プロセスがクラッシュしなくなります。この設定は、異なるタームの数ではなく稼働しているタームの数を参照します。 これにより大きいドキュメントが若干切り捨てられます (ドキュメントにあるすべてのタームがインデックスから除外されます)。ソースドキュメントが大きいことがわかっている場合は、期待されるサイズを収めることができるようこの値を十分に大きく設定してください。Integer.MAX_VALUE に設定した場合、唯一の制限はメモリです。ただし、OutOfMemoryError が発生します。 この値を **`transaction`** ではなく **`batch`** で設定する場合は、インデックス化モードに応じてインデックスの異なるデータ (および結果) を取得できます。	10000
`hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].max_merge_docs`	セグメントで許可されるドキュメントの最大数を定義します。大きい値はバッチインデックス化と高速な検索に最適です。小さい値はトランザクションインデックス化に最適です。	無制限 (Integer.MAX_VALUE)

プロパティ	説明	デフォルト値
hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].merge_factor	セグメントマージの頻度とサイズを制御します。 挿入が行われたとき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、インデックス化の間に使用される RAM の量が少なくなり、最適化されないインデックスの検索が高速になりますが、インデックス化の処理に時間がかかります。値を大きくすると、インデックス化の間に使用される RAM の量が多くなり、最適化されないインデックスの検索が低速になり、インデックス化が高速なります。したがって、バッチインデックス作成には大きい値 (> 10) が最適であり、対話的に保持するインデックスには小さい値 (< 10) が最適です。この値は 2 よりも小さくする必要があります。	10
hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].ram_buffer_size	ドキュメントバッファ専用の RAM の量 (MB 単位) を制御します。max_buffered_docs とともに使用すると、最初に発生したイベントに対してフラッシュが実行されます。 一般的に、高速なインデックス化パフォーマンスを実現するには、ドキュメント数の代わりに RAM の使用量によりフラッシュし、できるだけ大きい RAM バッファを使用することが最適です。	16 MB
hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].term_index_interval	上級者向け: インデックス化されたターム間の間隔を設定します。 値を大きくすると、IndexReader により使用されるメモリ量は小さくなりますが、タームへのランダムアクセスが低速になります。値を小さくすると、IndexReader により使用されるメモリ量は大きくなり、タームへのランダムアクセスが高速になります。詳細については、Lucene ドキュメンテーションを参照してください。	128

プロパティ	説明	デフォルト値
hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].use_compound_file	複合ファイル形式を使用する利点は使用されるファイル記述子が少なくなることです。欠点はインデックス化に時間がかかり、より多くの一時ディスク領域が必要になることです。インデックス化の時間を短縮するために、このパラメータを **false** に設定できますが、**mergeFactor** も大きい場合は、ファイル記述子が足りなくなることがあります。 ブール値パラメータの場合は、"**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

3.9. LOCKFACTORY の設定

Lucene Directories には、ほとんどの場合に使用できるデフォルトのロック方針がありますが、Hibernate Search により管理される各インデックスに対して使用する LockingFactory を指定できます。

これらのロック方針のいくつかは、ファイルシステムレベルのロックを必要とし、RAM ベースのインデックスに対しても使用できます。ただし、これは推奨されない方法であり、実践的ではありません。

ロックファクトリを選択するには、**hibernate.search.<index>.locking_strategy** オプションを **simple**、**native**、**single**、または **none** のいずれかに設定するか、あるいは**org.hibernate.search.store.LockFactoryFactory** の実装の完全修飾名に設定します。このインターフェイスを実装することにより、カスタム **org.apache.lucene.store.LockFactory** を提供できます。

表3.4 利用可能な LockFactory 実装のリスト

name	クラス	説明

name	クラス	説明
simple	org.apache.lucene.store.SimpleFSLockFactory	Java のファイル API をベースにした安全な実装。マーカーファイルを作成することにより、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終了する必要がある場合は、アプリケーションを再起動する前にこのファイルを削除する必要があります。 これは、**FSDirectoryProvider**、**FSMasterDirectoryProvider**、および **FSSlaveDirectoryProvider** のデフォルトの実装です。
native	org.apache.lucene.store.NativeFSLockFactory	**simple** と同様に、これもマーカーファイルを作成することによりインデックスの使用をマークします。ただし、アプリケーションがクラッシュした場合であってもロックがクリーンアップされるようにネイティブの OS ファイルロックが使用されます。 この実装には NFS に関する既知の問題が存在します。
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、メモリ内で保持される Java オブジェクトロックです。したがって、インデックスが他のプロセスにより共有されない場合のみ使用できます。 これは、**RAMDirectoryProvider** のデフォルトの実装です。
none	org.apache.lucene.store.NoLockFactory	このインデックスのすべての変更は、ロックにより調整されません。アプリケーションを慎重にテストし、何が行われるかを理解してください。

設定例:

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

第4章 インデックス構造にエンティティをマッピング

エンティティをインデックス化するために必要なすべてのメタデータ情報は、アノテーションを使用して定義されます。xml マッピングファイルで作業する必要はありません。実際は、現在 xml 設定オプションは利用できません (HSEARCH-210 を参照)。基本的な Hibernate 設定には Hibernate マッピングファイルを引き続き使用できますが、Hibernate Search 固有の設定はアノテーションを使用して指定する必要があります。

4.1. エンティティのマッピング

4.1.1. 基本的なマッピング

まず、 永続クラスをインデックス可能として宣言する必要があります。クラスに **@Indexed** アノテーションを付与してこれを行います (**@Indexed** アノテーションが付与されていないエンティティはすべてインデックス化のプロセスで無視されることになる)。

例4.1 @Indexed アノテーションを使用してクラスのインデックス化を可能にする

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...
}
```

index 属性は Lucene ディレクトリ名が何になるかを Hibernate に指示します (通常はファイルシステム上のディレクトリ)。設定ファイルで **hibernate.search.default.indexBase** プロパティを使用して、すべての Lucene インデックスにベースディレクトリを定義することが推奨されます。または、**hibernate.search.<index>.indexBase** (**<index>** はインデックス化されたエンティティの完全修飾クラス名) を指定してインデックス化された各エンティティに対してベースディレクトリを指定できます。各エンティティインスタンスは特定のインデックス (つまり Directory) の内側の Lucene **Document** で表されます。

エンティティの各プロパティ (または属性) に対してそれがどのようにインデックス化されるかを定義できます。デフォルトでは (つまりアノテーションなし) そのプロパティはインデックス化のプロセスで無視されます。**@Field** はプロパティをインデックス化されたものとして宣言します。Lucene ドキュメントにエレメントをインデックス化する場合はどのようにインデックス化するかを指定することができます。

- **name**: どの名前でプロパティが Lucene Document に格納されるかを指定します。 デフォルト値はそのプロパティ名です (JavaBeans 規則に準拠)。

- **store**: プロパティが Lucene インデックスに格納されるかされないかを指定します。**Store.YES** の値 (インデックス内の領域消費が多いが、プロジェクションを許可。詳細については、 「プロジェクション」 を参照)、 圧縮形式で格納する **Store.COMPRESS** (CPU の消費が多い)、 またはまったく格納しないようにする **Store.NO** (デフォルトの値) のいずれかにセットできます。 プロパティが格納される場合はその元の値を Lucene Document から検索できます。これはエレメントがインデックス化されるか否かとは関係ありません。

- index: どのようにエレメントのインデックス化を行うかと情報ストアのタイプを指定します。**Index.NO** (インデックス化なし、 クエリでは見つけられない)、 **Index.TOKENIZED** (プロパティの処理にアナライザを使用する)、 **Index.UN_TOKENISED** (アナライザ事前処理な

し)、 **`Index.NO_NORM`** (正常化データを格納しない)などの値があります。デフォルト値は **`TOKENIZED`** です。

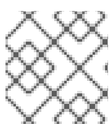
- termVector: 期間と頻度のペアのコレクションを定義します。この属性により、インデックス化の間に期間ベクタが格納され、ドキュメント内で期間ベクタを利用できるようになります。デフォルト値は TermVector.NO です。

  この属性の他の値は次のとおりです。

値	定義
TermVector.YES	各ドキュメントの期間ベクタを格納します。これにより、同期化された 2 つのアレイが生成されます。1 つはドキュメント期間を含み、もう 1 つは期間の頻度を含みます。
TermVector.NO	期間ベクタは格納しないでください。
TermVector.WITH_OFFSETS	期間ベクタとトークンオフセット情報を格納します。これは、TermVector.YES と同じです。また、期間の開始/終了オフセット位置情報も含まれます。
TermVector.WITH_POSITIONS	期間ベクタおよびトークン位置情報を格納します。これは、TermVector.YES と同じです。また、ドキュメント内の各期間の順序の位置が含まれます。
TermVector.WITH_POSITIONS_OFFSETS	期間ベクタ、トークン位置、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。

インデックスの元のデータを格納するかどうかは、インデックスクエリの結果をどのように使用するかに基づきます。通常の Hibernate Search の使用では、格納機能は必要ありません。ただし、一部のフィールドを格納して結果的にプロジェクションを行うことができます (詳細については、「プロジェクション」 を参照)。

プロパティをトークン化するかどうかは、エレメントをそのまま、または含まれるワードによって検索するかどうかに基づきます。テキストフィールドをトークン化することは適切ですが、日付フィールドをトークン化することは適切ではないことがあります。



注記

ソートに使用するフィールドはトークン化してはいけません。

最後に、エンティティの id プロパティは特定エンティティのインデックス単一性を確保するために Hibernate Search により使用される特殊なプロパティになります。 設計により、id を格納してトークン化しなければなりません。プロパティをインデックス id としてマークするには **`@DocumentId`** アノテーションを使用します。Hibernate Annotations を使用し、@Id を指定した場合は、@DocumentId を省略できます。選択されたエンティティ id はドキュメント id としても使用されます。

例4.2 インデックス化されたエンティティに @DocumentId ad @Field アノテーションを追加

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...

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

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

    @Lob
    @Field(index=Index.TOKENIZED)
    public String getText() { return text; }
}
```

例4.2「インデックス化されたエンティティに @DocumentId ad @Field アノテーションを追加」 では、**id**、**Abstract**、**text** の 3 つのフィールドでインデックスを定義します。デフォルトではフィールド名は JavaBean の仕様に従い小文字に変換されます。

4.1.2. プロパティを複数回マッピング

場合によっては、1 つのインデックスに対してプロパティを複数回 (若干異なるインデックス方針に従って) マップする必要があります。たとえば、query by フィールドをソートするには、フィールドが **UN_TOKENIZED** である必要があります。このプロパティのワードで検索し、ソートする場合は、プロパティを 2 回インデックス化する必要があります (1 回はトークン化、もう 1 回はトークン化解除)。@Fields を使用すると、この目的を達成できます。

例4.3 @Fields を使用してプロパティを複数回マップ

```
@Entity
@Indexed(index = "Book" )
public class Book {
    @Fields( {
            @Field(index = Index.TOKENIZED),
            @Field(name = "summary_forSort", index =
Index.UN_TOKENIZED, store = Store.YES)
            } )
    public String getSummary() {
        return summary;
    }

    ...
}
```

例4.3「@Fields を使用してプロパティを複数回マップ」 では、フィールド **summary** は 2 回インデックス化されます（1 回はトークン化の方法で **summary** として、もう 1 回は非トークン化の方法で **summary_forSort** として）。@Field は @Fields が使用された場合に役に立つ 2 つの属性をサポート

します。

- アナライザ: プロパティごとではなくフィールドごとに @Analyzer アノテーションを定義します。
- ブリッジ: プロパティごとではなくフィールドごとに @FieldBridge アノテーションを定義します。

アナライザとフィールドブリッジの詳細については、以下を参照してください。

4.1.3. 組込みおよび関連付けられたオブジェクト

関連付けられたオブジェクトと組込みオブジェクトは、ルートエンティティ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、関連付けられたオブジェクトのプロパティに基づいて特定のエンティティを検索する場合に役に立ちます。以下の例の目的は関連付けられた都市が Atlanta である場所を返します (Lucene クエリパーサー言語では、**`address.city:Atlanta`** に変換されます)。

例4.4 関係をインデックス化するために @IndexedEmbedded を使用

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

    @Field( index = Index.TOKENIZED )
    private String name;

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

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

    @Field(index=Index.TOKENIZED)
    private String street;

    @Field(index=Index.TOKENIZED)
    private String city;

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

この例では、プレースフィールドが **Place** インデックスでインデックス化されます。**Place** インデックスドキュメントには、フィールド **address.id**、**address.street**、および **address.city** (問い合わせることが可能) が含まれます。これは、**@IndexedEmbedded** アノテーションにより有効化されます。

以下の点にご注意ください：**@IndexedEmbedded** テクニックを使用した場合、データは Lucene インデックスで非正規化されるため、Hibernate Search はインデックスを最新の状態に保つために **Place** オブジェクトと **Address** オブジェクトの変更を認識する必要があります。**Address** が変更したときに **Place** Lucene ドキュメントが更新されるように、**@ContainedIn** との双方向の関係の一方をマークする必要があります。

@ContainedIn は、組込みオブジェクト (オブジェクトコレクション) ではなくエンティティを参照する関係の場合のみ役に立ちます。

例をもう少し複雑にしてみましょう。

例4.5 @IndexedEmbedded と @ContainedIn のネストでの使用

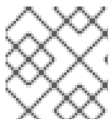
```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field( index = Index.TOKENIZED )
    private String name;

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

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

    @Field(index=Index.TOKENIZED)
    private String street;

    @Field(index=Index.TOKENIZED)
    private String city;

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

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

```
@Embeddable
public class Owner {
    @Field(index = Index.TOKENIZED)
    private String name;
   ...
}
```

@*ToMany, @*ToOne および **@Embedded** 属性は、 **@IndexedEmbedded** を使用してアノテートできます。関連付けられたクラスの属性は、主なエンティティインデックスに追加されます。前の例では、インデックスに以下のフィールドが含まれます。

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

デフォルトのプレフィックスは、従来のオブジェクトナビゲーション規則に基づき **propertyName.** となります。これは、**ownedBy** プロパティで示されたように **prefix** 属性を使用してオーバーライドできます。



注記

プレフィックスは空白の文字列として設定できません。

オブジェクトグラフにクラス (インスタンスではなく) の周期的な依存関係が含まれる場合は、 **depth** プロパティが必要です。たとえば、**Owner** が **Place** を参照する場合、Hibernate Search は期待された深さに達した後 (または、オブジェクトグラフ境界に達した後) にインデックス化された組込み属性を含めることを止めます。自己参照を持つクラスは周期的な依存関係の例です。例では、**depth** が 1 に設定されているため、Owner (存在する場合) の **@IndexedEmbedded** 属性は無視されます。

オブジェクト関係に **@IndexedEmbedded** を使用すると、以下のようにクエリを記述できます。

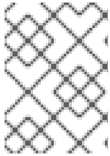
- 名前に JBoss が含まれ、住所の都市が Atlanta である場所を返します。Lucene クエリでは、以下のようになります。

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

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

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

つまり、これはリレーショナル結合操作をより効率的な方法で模倣しています (ただし、データ重複が問題となります)。デフォルトでは、Lucene インデックスには関係が含まれず、結合操作は単に存在しません。完全テキストインデックスの速度と機能の豊富さという点でメリットがある一方で、リレーショナルモデルを正規化することが役に立つ場合があります。



注記

関連付けられたオブジェクト自体が **@Indexed** である場合があります (ただし、これに限定されません)。

@IndexedEmbedded がエンティティを参照する場合、関係は両方向であり、一方を **@ContainedIn** でアノテートする必要があります (前の例で示されたように)。これに該当しない場合、Hibernate Search は関連付けられたエンティティが更新されたときにルートインデックスを更新できません (例では、関連付けられた **Address** インスタンスが更新されたときに、**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)。

場合によっては、**@IndexedEmbedded** でアノテートされたオブジェクトタイプが Hibernate と Hibernate Search が対象とするオブジェクトタイプではないことがあります。これは、実装の代わりにインターフェイスが使用される場合に特に当てはまります。このため、Hibernate Search が対象とするオブジェクトタイプは **targetElement** パラメータを使用してオーバーライドできます。

例4.6 @IndexedEmbedded の targetElement プロパティの使用

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

    @Field(index= Index.TOKENIZED)
    private String street;

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


    ...
}

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

4.1.4. ブーストファクタ

Lucene には **ブーストファクタ** という概念があります。インデックス化の処理中にあるフィールドまたはインデックス化エレメントを他のフィールドまたはエレメントよりも重視する方法です。**@Boost** を @Field、メソッド、またはクラスレベルで使用することができます。

例4.7 ブーストファクタを使用してインデックス化エレメントの重みをさまざまな方法で増やす

```
@Entity
@Indexed(index="indexes/essays")
@Boost(1.7f)
```

```
public class Essay {
    ...

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

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

    @Lob
    @Field(index=Index.TOKENIZED, boost=@Boost(1.2f))
    public String getText() { return text; }

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

}
```

ここの例では **Essay** が検索一覧のトップに到達する確率は 1.7 で乗じられ、**summary** フィールドは **isbn** フィールドより 3.0 (2 * 1.5 - **@Field.boost** とプロパティの **@Boost** は累積されます) 重要になります。**text** フィールドは **isbn** よりも 1.2 場合重要になります。 この説明は実際には誤りですが、説明としてはわかりやすく現実に近いものとなります。 Otis Gospodnetic および Erik Hatcher の『Lucene In Action』 または Lucene 関連のドキュメントを確認してください。

4.1.5. 動的ブーストファクタ

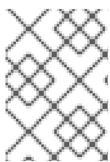
「ブーストファクタ」 で使用する **@Boost** アノテーションは、ランタイム時にインデックス化されたエンティティの状態から無関係の静的ブーストファクタを定義します。ただし、ブーストファクタが実際のエンティティの状態によって異なる使用事例もあります。この場合は、付随のカスタムの **BoostStrategy** とともに **@DynamicBoost** アノテーションを使用できます。

例4.8 動的ブーストの例

```
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;
            }
        }
    }
```

例4.8「動的ブーストの例」 では、動的ブーストは、インデックス化の時に使用される **BoostStrategy** インターフェースの実装として **VIPBoostStrategy** を指定するクラスレベルで定義されます。クラスレベルまたはフィールドレベルのどちらかに **@DynamicBoost** を配置することができます。アノテーションの配置により、エンティティ全体が **defineBoost** メソッドに渡されるか、アノテートされたフィールド／プロパティの値だけか異なります。渡されたオブジェクトを正しい種類にキャストするのはユーザー次第です。例では、重要人物が持つインデックス値はすべて、通常の人の値と比べ 2 倍重要になります。



注記

指定の **BoostStrategy** の実装は、引数なしのパブリックコンストラクタを定義します。

もちろん、エンティティの **@Boost** と **@DynamicBoost** アノテーションを適合することができます。すべての定義されたブーストファクタは、「ブーストファクタ」 に記載のとおり累積されます。

4.1.6. アナライザ

トークン化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クラスは **hibernate.search.analyzer** プロパティから設定することができます。 このプロパティのデフォルト値は **org.apache.lucene.analysis.standard.StandardAnalyzer** です。

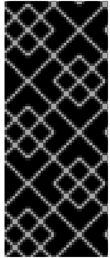
また、エンティティ、プロパティ、@Field (単一のプロパティから複数のフィールドをインデックス化する場合に役に立ちます) ごとにアナライザクラスを定義することもできます。

例4.9 アナライザを指定するさまざまな方法

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

    @Field(index = Index.TOKENIZED)
    private String name;

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

    @Field(index = Index.TOKENIZED, analyzer = @Analyzer(impl =
FieldAnalyzer.class)
    private String body;
```

```
    ...
}
```

この例では、すべてのトークン化プロパティ (**name** など) をインデックス化するために **EntityAnalyzer** が使用されます (それぞれ **PropertyAnalyzer** と **FieldAnalyzer** でインデックス化される **summary** と **body** を除く)。



重要

ほとんどの場合、同じエンティティに異なるアナライザを混在させることは推奨されません。これにより、クエリの構築が複雑になり、結果が (初心者にとって) 予測不可能になります (特に QueryParser (クエリ全体に対して同じアナライザを使用) を使用する場合)。原則として、あるフィールドに対して、インデックス化と問い合わせのためにアナライザを使用する必要があります。

4.1.6.1. アナライザ定義

アナライザの使用は非常に複雑になることがあるため、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、多くの **@Analyzer** 宣言によって再利用できます。アナライザ定義は以下のものから構成されます。

- 名前: 定義を参照するために使用される一意の文字列
- トークナイザ: 個々のワードに入力ストリームをトークン化する
- フィルタのリスト: 各フィルタはトークナイザにより提供されたストリームに対してワードを削除、変更、または場合によっては追加する

このようにタスクを分離することにより (トークナイザとフィルタのリスト)、各コンポーネントを簡単に再利用し、非常に柔軟に (レゴブロックのように) カスタマイズアナライザを構築できます。一般的に、**Tokenizer** は入力された文字をトークン (さらに **TokenFilter** によって処理される) に変換することによって分析プロセスを開始します。Hibernate Search は、Solr アナライザフレームワークを使用してこのインフラストラクチャをサポートします。アナライザ定義を使用するためにクラスパスに **solr-core.jar and solr-common.jar** を追加してください。スノーボールステマーも使用する場合は、**lucene-snowball.jar.** も含めてください。他の Solr アナライザはさらに多くのライブラリに依存することがあります。たとえば、**PhoneticFilterFactory** は commons-codec に依存します。Hibernate Search のディストリビューションは、これらの依存関係を **lib** ディレクトリで提供します。

例4.10 @AnalyzerDef および Solr フレームワーク

```
@AnalyzerDef(name="customanalyzer",
        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 {
    ...
}
```

トークナイザは、トークナイザを構築子、オプションのパラメータリストを使用するファクトリにより定義されます。この例では、標準的なトークナイザを使用します。フィルタはオプションのパラメータを使用してフィルタインスタンスを作成するファクトリにより定義されます。この例では、StopFilter フィルタが構築され、専用のワードプロパティファイルを読み取り、大文字と小文字を区別しません。パラメータのリストはトークナイザまたはフィルタファクトリに依存します。

警告

フィルタは**@AnalyzerDef** アノテーションで定義された順序で適用されます。この順序についてよく考えてください。

定義されたアナライザ定義は、**@Analyzer** 宣言により実装クラスを宣言するのではなく定義名を使用して再利用できます。

例4.11 名前によるアナライザの参照

```
@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 により宣言されたアナライザインスタンスは **SearchFactory** の名前で利用できます。

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

これは、クエリを構築するときに非常に役に立ちます。クエリのフィールドは、フィールドをインデックス化するのに使用されたのと同じアナライザで分析する必要があります (したがって、これらは共通の「言語」を話します。クエリとインデックス化プロセス間で同じトークンが再利用されます)。このルールにはいくつかの例外がありますが、ほとんどの場合該当します。特に何を行うかがわかっていない限り、このルールに従ってください。

4.1.6.2. 利用可能なアナライザ

Solr と Lucene にはたくさんの役に立つデフォルトのトークナイザとフィルタが含まれます。トークナイザファクトリとフィルタファクトリの完全なリストは http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters で見つけることができます。これらのいくつかについて確認してみましょう。

表4.1 利用可能なトークナイザの一部

ファクトリ	説明	パラメータ
StandardTokenizerFactory	Lucene 標準トークナイザを使用する	none
HTMLStripStandardTokenizerFactory	HTML タグを削除し、テキストを保持し、StandardTokenizer に渡す	none

表4.2 利用可能なフィルタの一部

ファクトリ	説明	パラメータ
StandardFilterFactory	頭字語とワードからドットを削除する	none
LowerCaseFilterFactory	小文字のワード	none
StopFilterFactory	ストップワードのリストに一致するワード (トークン) を削除する	`words`: ストップワードを含むリソースファイルを参照する ignoreCase: ストップワードを比較するときに `case` を無視する場合は true、その他の場合は `false`
SnowballPorterFilterFactory	ワードを特定の言語の語幹に減らす (たとえば、protect、protects、protection は同じ語幹を共有する)。このようなフィルタを使用すると、関連ワードに一致する検索ができる。	`language`: Danish、Dutch、English、Finnish、French、German、Italian、Norwegian、Portuguese、Russian、Spanish、Swedish その他

ファクトリ	説明	パラメータ
ISOLatin1AccentFilterFactory	フランス語のような言語のアクセントを削除する	none

IDE の **org.apache.solr.analysis.TokenizerFactory**、**org.apache.solr.analysis.TokenFilterFactory** のすべての実装をチェックして利用可能な実装を確認することをお薦めします。

4.1.6.3. アナライザ判別子 (実験段階)

これまで説明したアナライザの指定方法はすべて静的でした。ただし、インデックス化するエンティティの現在の状態に応じてアナライザを選択することが役に立つ場合があります (マルチリンガルアプリケーションなど)。たとえば、**BlogEntry** クラスの場合、アナライザは、エンティティの言語プロパティに依存することがあります。このプロパティに応じて、実際のテキストをインデックス化するために適切な言語固有ステマーを選択する必要があります。

この動的アナライザ選択を有効にするために、Hibernate Search には **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4.12 アナライザを選択するための @AnalyzerDiscriminator の使用方法はエンティティステータスによって異なる

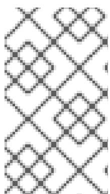
```
@Entity
@Indexed
@AnalyzerDefs({
  @AnalyzerDef(name = "en",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = EnglishPorterFilterFactory.class
      )
    }),
  @AnalyzerDef(name = "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 getAnanyzerDefinitionName(Object value, Object entity,
String field) {
        if ( value == null || !( entity instanceof Article ) ) {
            return null;
        }
        return (String) value;
    }
}
```

@AnalyzerDiscriminator を使用する前提条件は、使用されるすべてのアナライザが **@AnalyzerDef** 定義で事前に定義されていることです。これに該当する場合は、**@AnalyzerDiscriminator** アノテーションをクラス、またはアナライザを動的に選択するエンティティの特定のプロパティに配置できます。**AnalyzerDiscriminator** の **impl** パラメータを使用して、**Discriminator** インターフェイスの具体的な実装を指定します。実装する必要がある唯一のメソッドは、Lucene ドキュメントに追加される各フィールドに対して呼び出される **getAnanyzerDefinitionName()** です。インデックス化されるエンティティは、インターフェイスメソッドにも渡されます。**value** パラメータは、**AnalyzerDiscriminator** がクラスレベルではなくプロパティレベルに配置された場合にのみ設定されます。この場合、値はこのプロパティ現在の値を表します。

Discriminator インターフェイスの実装は、アナライザを動的に設定する場合は既存のアナライザ定義の名前、デフォルトのアナライザをオーバーライドしない場合は **null** を返す必要があります。示された例では、言語パラメータが **@AnalyzerDef** で指定された名前に一致する 'de' または 'en' のいずれかであることを前提とします。

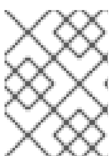


注記

@AnalyzerDiscriminator は現在まだ実験段階にあり、API は変わる可能性があります。この機能の利便性と使用感についてコミュニティからのフィードバックを募集しています。

4.1.6.4. アナライザの取得

インデックス作成中に、Hibernate Search は背後でアナライザを使用します。場合によっては、アナライザの取得が便利です。ドメインモデルが複数のアナライザを使用する場合 (ステミングのメリットを得るための音声近似の使用など)、クエリを構築する場合に同じアナライザを使用する必要があります。



注記

このルールに従わないこともできますが、その場合は理由が必要です。よくわからない場合は、同じアナライザを使用してください。

Hibernate Search によりインデックス作成時に使用される特定のエンティティに対してスコープ設定されたアナライザを取得できます。スコープ設定されたアナライザは、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を適用するアナライザです。複数のアナライザは特定のエンティティ上で定義できます (それぞれは個々のフィールド上で動作します)。スコープ設定されたアナライザはこれらすべてのアナライザを 1 つのコンテキスト認識アナライザに統一します。理論的に少し複雑ですが、クエリで適切なアナライザを使用することは非常に簡単です。

例4.13 完全テキストクエリの構築時にスコープ設定されたアナライザを使用

```
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、歌のタイトルは 2 つのフィールドでインデックス化されます。標準的なアナライザはフィールド **`title`** で使用され、ステミングアナライザはフィールド **`title_stemmed`** で使用されます。検索ファクトリにより提供されたアナライザを使用することにより、クエリは対象となるフィールドに応じて適切なアナライザを使用します。

クエリが複数のクエリを対象とし、標準的なアナライザを使用する場合、アナライザ定義を使用してそれを定義する必要があります。アナライザは、**`searchFactory.getAnalyzer(String)`** を使用して定義により取得できます。

4.2. プロパティ／フィールドのブリッジ

Lucene では、すべてのインデックスフィールドは String として表現する必要があります。このため、**`@Field`** でアノテートされたすべてのエンティティプロパティは String 形式でインデックス化する必要があります。プロパティのほとんどでは、Hibernate Search がビルトインのブリッジセットで変換作業を行ってくれます。 ただし、 変換プロセスを細かく制御する必要がある場合もあります。

4.2.1. ビルトインのブリッジ

Hibernate Search には Java プロパティタイプとその完全テキスト表現間のビルトインブリッジセットが同梱されます。

null

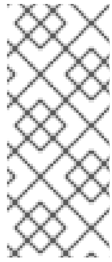
null エレメントはインデックス化されません。 Lucene は null エレメントに対応しないため意味がありません。

java.lang.String

String はそのままインデックス化されます。

short、 Short、 integer、 Integer、 long、 Long、 float、 Float、 double、 Double、 BigInteger、 BigDecimal

数値はその文字列表現に変換されます。 Lucene はそのままでは数値を比較できないため (つまり、対象のクエリ内で使用できない)、 パディイングする必要があります。



注記

Range クエリの使用は議論の余地があり欠点もあるため、 代替となる手段として Filter クエリの使用があります。 これは結果となるクエリを適切な範囲にフィルタします。

Hibernate Search はパッディングのメカニズムをサポートします。

java.util.Date

日付は GMT 時間で yyyyMMddHHmmssSSS のように格納されます (2006 年 11 月 7 日の 4:03PM 12 ミリ秒 EST の場合は 200611072203012)。 内部形式は気にする必要はありません。 重要なのは DateRange クエリを使用する場合にその日付が GMT 時間で表現されるということを認識しておくことです。

通常、 日付をミリ秒まで格納する必要がありません。 **@DateBridge** はインデックスに格納しようとしている適切なリゾリューションを定義します (**@DateBridge(resolution=Resolution.DAY)**)。 日付のパターンは定義に応じて切り捨てられます。

```
@Entity
@Indexed
public class Meeting {
    @Field(index=Index.UN_TOKENIZED)
    @DateBridge(resolution=Resolution.MINUTE)
    private Date date;
    ...
```

警告

MILLISECOND より低い単位の日付は **@DocumentId** にはなれません。

java.net.URI、java.net.URL

URI と URL は文字列形式に変換されます。

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、クラスがハイドレートされたときに使用されます。

4.2.2. カスタムのブリッジ

Hibernate Search のビルトインブリッジが使用するプロパティタイプの一部に対応しなかったり、 ブリッジにより使用される文字列表現が要件を満たさないことがります。以下のパラグラフはこの問題の複数のソリューションについて説明します。

4.2.2.1. StringBridge

もっともシンプルなカスタムのソリューションとして Hibernate Search に期待する **Object** と **String** のブリッジの実装を提供します。これを行うには **org.hibernate.search.bridge.StringBridge** インターフェー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4.14 独自の StringBridge の実装

```
/**
 * 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();
    }
}
```

すると、 いずれのプロパティまたはフィールドも **@FieldBridge** アノテーションによりこのブリッジを使用できるようになります。

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

柔軟性を向上させるためにパラメータをそのブリッジ実装に渡すことができます。 ブリッジ実装は **ParameterizedBridge** インターフェースを実装し、 そのパラメータは **@FieldBridge** アノテーションで渡されます。

例4.15 ブリッジ実装にパラメータを渡す

```
public class PaddedIntegerBridge implements StringBridge,
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();
    }
}


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

ParameterizedBridge インターフェースは **StringBridge**、 **TwoWayStringBridge**、**FieldBridge** の実装で実装可能です。

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、パラメータは初期化時に設定されるため、この段階では特に気を付ける必要はありません。

id プロパティにブリッジ実装を使用する予定がある場合 (つまり **@DocumentId** アノテーションを付与)、 若干拡張した **TwoWayStringBridge** という名前の **StringBridge** バージョン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違いはありません。

例4.16 id プロパティなどに使用できる 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;
```

2 方向プロセスがべき等となることが非常に重要です (つまり、 object = stringToObject(objectToString(object)))。

4.2.2.2. FieldBridge

プロパティを Lucene インデックスにマッピングする際に単純なオブジェクトから文字列への変換だけではなくそれ以上のものを必要とする場合があります。 柔軟性を最大限に引き出すためブリッジを **`FieldBridge`** として実装することもできます。 このインターフェースによりプロパティ値が提供され、Lucene **`Document`** で行いたい方法でマッピングすることができます。 このインターフェースは概念的には Hibernate **`UserType`** に非常によく似ています。

たとえば、 特定のプロパティを 2 種類のドキュメントフィールドに格納することができます。

例4.17 特定のプロパティを複数のドキュメントフィールドに格納するため FieldBridge インターフェースを実装

```
/**
 * 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
        Field field = new Field(name + ".year", String.valueOf(year),
            luceneOptions.getStore(), luceneOptions.getIndex(),
            luceneOptions.getTermVector());
        field.setBoost(luceneOptions.getBoost());
        document.add(field);

        // set month and pad it if needed
        field = new Field(name + ".month", month < 10 ? "0" : ""
            + String.valueOf(month), luceneOptions.getStore(),
            luceneOptions.getIndex(), luceneOptions.getTermVector());
        field.setBoost(luceneOptions.getBoost());
        document.add(field);

        // set day and pad it if needed
        field = new Field(name + ".day", day < 10 ? "0" : ""
            + String.valueOf(day), luceneOptions.getStore(),
            luceneOptions.getIndex(), luceneOptions.getTermVector());
        field.setBoost(luceneOptions.getBoost());
        document.add(field);
    }
}

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

4.2.2.3. ClassBridge

場合によっては、特定のエンティティの複数のプロパティを組み合わせ、この組み合わせを特定の方法で Lucene インデックスにインデックス化することが役に立つことがあります。**@ClassBridge** アノテーションと **@ClassBridge** アノテーションは (プロパティレベルではなく) 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、値パラメータとして特定のプロパティではなくエンティティインスタンスを受け取ります。この例では示されていませんが、**@ClassBridge** は項 「基本的なマッピング」 で説明された **termVector** 属性をサポートします。

例4.18 クラスブリッジの実装

```
@Entity
@Indexed
@ClassBridge(name="branchnetwork",
             index=Index.TOKENIZED,
             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 **CatFieldsClassBridge** 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、その連結をインデックス化します。

4.3. 独自の ID の提供

警告

ドキュメンテーションのこの部分は準備中です。

内部機能を拡張する場合は、Hibernate Search に独自の id を提供できます。インデックス化する Lucene に提供できるよう一意の値を生成する必要があります。これは、org.hibernate.search.Work オブジェクトを作成する場合に Hibernate Search に提供する必要があります。ドキュメント id はコンス

トラクタで必要です。

4.3.1. ProvidedId アノテーション

従来の Hibernate Search API と @DocumentId とは異なり、このアノテーションはフィールドではなくクラスで使用されます。また、このアノテーションを使用する場合は @ProvidedId にある bridge() を呼び出すことにより独自のブリッジ実装を提供することもできます。さらに、@ProvidedId でクラスをアノテートする場合は、サブクラスもアノテーションを取得します。ただし、これは、java.lang.annotations.@Inherited を使用して行われません。ただし、システムが破壊しないよう @DocumentId でこのアノテーションを使用しないでください。

例4.19 独自の id の提供

```
@ProvidedId (bridge = org.my.own.package.MyCustomBridge)
@Indexed
public class MyClass{
    @Field
    String MyString;
    ...
}
```

第5章 クエリ

Hibernate Search の 2 番目に重要な機能は、Lucene クエリを実行し、Hibernate セッションにより管理されたエンティティを取得する機能です (これにより、Hibernate のパラダイム内で Lucene の能力が提供され、Hibernate の従来の検索メカニズム (HQL、基準クエリ、ネイティブ SQL クエリ) に別の次元が提供されます。クエリの準備と実行は 4 つの単純な手順から構成されます。

- **`FullTextSession`** の作成
- Lucene クエリの作成
- **`org.hibernate.Query`** を使用した Lucene クエリのラップ
- **`list()`** や **`scroll()`** などを呼び出して検索を実行

クエリ機能にアクセスするには、**`FullTextSession`** を使用する必要があります。この検索固有のセッションは、通常の **`org.hibernate.Session`** をラップしてクエリ機能とインデックス機能を提供します。

例5.1 FullTextSession の作成

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

実際の検索機能は、以下の例で示されたネイティブの Lucene クエリに基づいて構築されます。

例5.2 Lucene クエリの作成

```
org.apache.lucene.queryParser.QueryParser parser =
    new QueryParser("title", new StopAnalyzer() );

org.apache.lucene.search.Query luceneQuery = parser.parse(
"summary:Festina Or brand:Seiko" );
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(
luceneQuery );
List result = fullTextQuery.list(); //return a list of managed objects
```

Lucene クエリ上に構築された Hibernate クエリは通常の **`org.hibernate.Query`** です。つまり、他の Hibernate クエリ機能 (HQL、Native、または Criteria) と同じパラダイムに属しています。通常の **`list()`**、**`uniqueResult()`**、**`iterate()`**、および **`scroll()`** メソッドを使用できます。

Hibernate の Java Persistence API (EJB 3.0 Persistence とも呼ばれます) を使用している場合は、同じ拡張機能が存在します。

例5.3 JPA API を使用した検索クエリの作成

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

FullTextEntityManager fullTextEntityManager =
```

```
org.hibernate.hibernate.search.jpa.Search.getFullTextEntityManager(em);

...
org.apache.lucene.queryParser.QueryParser parser =
    new QueryParser("title", new StopAnalyzer() );

org.apache.lucene.search.Query luceneQuery = parser.parse(
"summary:Festina Or brand:Seiko" );
javax.persistence.Query fullTextQuery =
fullTextEntityManager.createFullTextQuery( luceneQuery );

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

以下の例では、Hibernate API を使用します。ただし、同じ例は、Java Persistence API を使用して **`FullTextQuery`** が取得される方法を調整することによって簡単に書き換えることができます。

5.1. クエリの構築

Hibernate Search クエリは Lucene クエリの上部に構築され、実行する Lucene クエリのタイプはまったく関係ありません。ただし、構築された Hibernate Search は **`org.hibernate.Query`** を主なクエリ操作 API として使用してクエリ処理をラップします。

5.1.1. Lucene クエリの構築

Lucene クエリの実際の構築方法については、このドキュメンテーションの範囲外です。オンラインの Lucene ドキュメンテーションまたは『Lucene In Action』あるいは『Hibernate Search in Action』を参照してください。

5.1.2. Hibernate Search クエリの構築

5.1.2.1. 概要

Lucene クエリが構築されたら、Lucene クエリを Hibernate Query にラップする必要があります。

例5.4 Hibernate Query への Lucene クエリのラップ

```
FullTextSession fullTextSession = Search.getFullTextSession( session );
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(
luceneQuery );
```

特に指定がない限りクエリはインデックス化されたすべてのエンティティに対して実行されます (場合によっては、インデックス化されたクラスのすべてのタイプが返されます)。パフォーマンスの観点から、返されるタイプを制限することが推奨されます。

例5.5 エンティティタイプによる検索結果のフィルタリング

```
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(
luceneQuery, Customer.class );
// or
```

```
fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery,
Item.class, Actor.class );
```

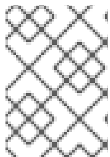
最初の例は一致する **Customer** だけを返し、2 つ目の例は一致する **Actor** と **Item** を返します。タイプの制限は完全にポリモーフィックです。つまり、基本クラス **Person** の、2 つのインデックス化されたサブクラス**Salesman** と **Customer** が存在する場合、結果タイプに基づいてフィルタするには **Person.class** を指定します。

5.1.2.2. ページ処理

パフォーマンス上の理由から、1 つのクエリに対して返されるオブジェクトの数を制限することが推奨されます。ユーザーがあるページから別のページに移動することは実際にはよくあることです。ページ処理を定義する方法は、プレーンな HQL または基準クエリでページ処理を定義する方法とまったく同じです。

例5.6 検索クエリのページ処理の定義

```
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() を使用すると、ページ処理に関係なく一致するエレメントの合計数を取得できます。

5.1.2.3. ソート

Apache Lucene を使用すると、結果を非常に柔軟かつ強力にソートできます。ほとんどの場合はデフォルトのソート (重要度別) が適切ですが、1 つまたは複数の他のプロパティによってソートすることもできます。これを行うには、Lucene Sort オブジェクトが Lucene ソート方針を適用するよう設定します。

例5.7 結果をソートするための Lucene Sort の指定

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( query,
Book.class );
org.apache.lucene.search.Sort sort = new Sort(new SortField("title"));
query.setSort(sort);
List results = query.list();
```

FullTextQuery インタフェースが **org.hibernate.Query** のサブインタフェースであることに気づかれるかもしれません。ソートに使用されるフィールドはトークン化しないよう注意してください。

5.1.2.4. フェッチ方針

戻り値のタイプを 1 つのクラスに制限する場合、Hibernate Search は単一のクエリを使用してオブジェクトをロードします。また、ドメインモデルで定義された静的なフェッチ方針が優先されます。

ただし、多くの場合、特定の使用例のためにフェッチ方針を調整することが役に立ちます。

例5.8 クエリでの FetchMode の指定

```
Criteria criteria = s.createCriteria( Book.class ).setFetchMode(
"authors", FetchMode.JOIN );
s.createFullTextQuery( luceneQuery ).setCriteriaQuery( criteria );
```

この例では、クエリは luceneQuery に一致するすべての本を返します。作者コレクションは SQL 外部結合を使用して同じクエリからロードされます。

基準クエリを定義する場合は、完全テキストセッションから Hibernate Search クエリを作成するときに返されるエンティティタイプを制限する必要はありません。タイプは基準クエリ自体から推測されます。フェッチモードのみを調整し、他の制限の適用を回避できます。

複数のエンティティが返されることが期待される場合は、**setCriteriaQuery** を使用できません。

5.1.2.5. プロジェクション

場合によっては、ドメインオブジェクト (グラフ) を返すことがやりすぎになることがあります。Hibernate Search では、プロパティのサブセットを返すことができます。

例5.9 完全なドメインオブジェクトを返す代わりにプロジェクションを使用

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
luceneQuery, Book.class );
query.setProjection( "id", "summary", "body", "mainAuthor.name" );
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
  Integer id = (Integer) firstResult[0];
  String summary = (String) firstResult[1];
  String body = (String) firstResult[2];
  String authorName = (String) firstResult[3];
```

Hibernate Search は Lucene インデックスからプロパティを抽出し、オブジェクト形式に再び変換します (**Object[]** のリストが返されます)。プロジェクションにより、潜在的なデータベースラウンドトリップ (クエリの応答時間が重要な場合に有用) が回避されますが、いくつかの制限が存在します。

- 予測されたプロパティはインデックス (**@Field(store=Store.YES)**) に格納する必要があります (これによりインデックスサイズが増加します)。
- 予測されたプロパティは、**org.hibernate.search.bridge.TwoWayFieldBridge** または **org.hibernate.search.bridge.TwoWayStringBridge** (単純なバージョン) を実装する **FieldBridge** を使用する必要があります。すべての Hibernate Search 組込みタイプは両方向です。
- インデックス化されたエンティティまたは組み込まれた関係の単純なプロパティのみを予測できます。つまり、組み込まれたエンティティ全体を予測できません。
- プロジェクションは、**@IndeedEmbedded** を使用してインデックス化されたコレクションまたはマップに対して機能しません。

プロジェクションは、別の場合にも役に立ちます。Lucene は結果に関するいくつかのメタデータ情報をユーザーに提供します。いくつかの特殊なプレースホルダーを使用することにより、プロジェクションメカニズムはこれらのメタデータを取得できます。

例5.10 メタデータを取得するためにプロジェクションを使用

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
luceneQuery, Book.class );
query.setProjection( FullTextQuery.SCORE, FullTextQuery.THIS,
"mainAuthor.name" );
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
  float score = (Float) firstResult[0];
  Book book = (Book) firstResult[1];
  String authorName = (String) firstResult[2];
```

通常のフィールドと特殊なプレースホルダーは混在させることができます。利用可能なプレースホルダーのリストは以下のとおりです。

- FullTextQuery.THIS: 初期化および管理されたエンティティを返します (予測されないクエリの場合と同様)。
- FullTextQuery.DOCUMENT: 予測されたオブジェクトに関連する Lucene Document を返します。
- FullTextQuery.OBJECT_CLASS: インデックス化されたエンティティのクラスを返します。
- FullTextQuery.SCORE: クエリのドキュメンテーションスコアを返します。スコアはある結果とクエリの他の結果を比較するのに役に立ちしますが、異なるクエリの結果を比較する場合は役に立ちません。
- FullTextQuery.ID: 予測されたオブジェクトの id プロパティ値。
- FullTextQuery.DOCUMENT_ID: Lucene ドキュメント id。Lucene ドキュメント id は 2 つの異なる IndexReader のオープン間で変わることがあることに注意してください (この機能は実験段階にあります)。
- FullTextQuery.EXPLANATION: 該当するクエリの一致するオブジェクト/ドキュメントに対する Lucene Explanation オブジェクトを返します。たくさんのデータを取得する場合は使用しないでください。通常、Explanation を返すコストは、一致するエレメントごとに Lucene クエリ全体を実行するコストと同じです。

5.2. 結果の取得

Hibernate Search クエリが構築された場合、Hibernate Search クエリの実行は HQL または Criteria クエリの実行とまったく変わりません。同じパラダイムとオブジェクトセマンティックが適用されます。**`list()`**、**`uniqueResult()`**、**`iterate()`**、**`scroll()`** などのすべての共通の操作が利用可能です。

5.2.1. パフォーマンスに関する考慮事項

適切な結果の数を期待し (たとえば、ページ処理を使用)、これらすべてを処理する場合は、**`list()`** または **`uniqueResult()`** が推奨されます。エンティティ **`batch-size`** が適切に設定された場合は、**`list()`** が最適です。**`list()`**、**`uniqueResult()`**、および **`iterate()`** を使用する場合、

Hibernate Search はすべての Lucene Hits エレメントを (ページ処理内で) 処理する必要があることに注意してください。

Lucene ドキュメントのロードを最小化する場合は、**scroll()** が適しています。作業を行ったら **ScrollableResults** オブジェクトをクローズすることを忘れないでください (クローズしないと、Lucene リソースが保持されます)。**scroll,** を使用し、オブジェクトをバッチでロードする場合は、**query.setFetchSize()** を使用できます。オブジェクトがアクセスされ、まだロードされていない場合、Hibernate Search は 1 つのパスで次の **fetchSize** をロードします。

スクロールよりもページ処理の方が推奨されます。

5.2.2. 結果サイズ

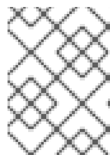
一致するドキュメントの合計数を知ることが役に立つ場合があります。

- Google のような機能用 (約 888,000,000 のうちの 1-10)
- 高速なページ処理ナビゲーションの実装
- マルチステップ検索エンジンの実装 (制限されたクエリが結果をまったく返さない、または十分な結果を返さない場合に近似値を追加)

もちろん、一致するすべてのドキュメントを取得することはコストが高すぎます。Hibernate Search では、ページ処理パラメータ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、オブジェクトをまったくロードせずに一致するエレメントの数を取得できます。

例5.11 クエリの結果サイズの決定

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
luceneQuery, Book.class );
assert 3245 == query.getResultSize(); //return the number of matching
books without loading a single one

org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
luceneQuery, Book.class );
query.setMaxResults(10);
List results = query.list();
assert 3245 == query.getResultSize(); //return the total number of
matching books regardless of pagination
```



注記

Google のように、インデックスがデータベースと完全に同期されていない場合 (非同期クラスタなど)、結果の数は近似になります。

5.2.3. ResultTransformer

特にプロジェクションを使用する場合、クエリによって返されたデータ構造 (この場合はオブジェクトアレイ) は常にアプリケーションのニーズを満たすとは限りません。対象となるデータ構造に一致するよう **ResultTransformer** 操作ポストクエリを適用できます。

例5.12 プロジェクションとともに ResultTransformer を使用

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
luceneQuery, Book.class );
query.setProjection( "title", "mainAuthor.name" );

query.setResultTransformer( new AliasToBeanResultTransformer(
BookView.class ) );
List<BookView> results = (List<BookView>) query.list();
for(BookView view : results) {
    log.info( "Book: " + view.getTitle() + ", " + view.getAuthor() );
}
```

ResultTransformer 実装のサンプルは Hibernate Core コードベースにあります。

5.2.4. 結果の理解

結果がクエリに表れたり、表れなかったりして不思議に思うことがあるかもしれません。Luke はこのことについて理解するのに優れたツールです。ただし、Hibernate Search は該当する結果に対して Lucene **Explanation** オブジェクトへのアクセスを提供します (該当するクエリで)。このクラスは Lucene ユーザーにとっては非常に高度なものですが、オブジェクトのスコアを理解するのに役に立ちます。該当する結果の Explanation オブジェクトにアクセスするには 2 つの方法があります。

- **fullTextQuery.explain(int)** メソッドを使用
- プロジェクションを使用

最初の方法では、ドキュメント id をパラメータとして取得し、Explanation オブジェクトを返します。ドキュメント id はプロジェクションと **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 id とは関係ありません。これらの 2 つを混同しないでください。

2 つ目の方法では、**FullTextQuery.EXPLANATION** 定数を使用して **Explanation** オブジェクトを予測します。

例5.13 プロジェクションを使用して Lucene Explanation オブジェクトを取得

```
FullTextQuery ftQuery = s.createFullTextQuery( luceneQuery, Dvd.class )
        .setProjection( FullTextQuery.DOCUMENT_ID,
FullTextQuery.EXPLANATION, FullTextQuery.THIS );
@SuppressWarnings("unchecked") List<Object[]> results = ftQuery.list();
for (Object[] result : results) {
    Explanation e = (Explanation) result[1];
    System.out.println( e.toString() );
}
```

Explanation オブジェクトの構築は非常にコストがかかることに注意してください。このコストは大体 Lucene クエリを再び実行するコストと同じです。Explanation オブジェクトが必要ない場合はこの作業を行わないでください。

5.3. フィルタ

Apache Lucene には、カスタムフィルタリングプロセスに従ってクエリ結果をフィルタできる強力な機能があります。これは、特にフィルタをキャッシュおよび再利用できるため、追加データ制限を適用するのに非常に強力です。一部の使用例は以下のとおりです。

- セキュリティ
- 一時データ (先月のデータのみを表示するなど)
- 生成フィルタ (該当するカテゴリに制限される検索など)
- その他

Hibernate Search は透過的にキャッシュされるパラメータ設定可能な名前付きフィルタの概念を導入することにより、このコンセプトを押し進めています。Hibernate Core フィルタの概念を知っているユーザーにとってこの API は非常に似ています。

例5.14 該当するクエリに対して完全テキストフィルタを有効化

```
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、クエリの上部に 2 つのフィルタが有効化されています。任意の数のフィルタを有効化 (または無効化) できます。

フィルタの宣言は **@FullTextFilterDef** アノテーションを使用して行われます。このアノテーションはフィルタが後で適用されるクエリに関係なく任意の **@Indexed** エンティティに対するものです。これは、フィルタ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暗黙的に示しています。**SearchException** は同じ名前を持つ 2 つの異なる **@FullTextFilterDef** アノテーションが定義された場合にスローされます。各名前付きフィルタは実際のフィルタ実装を指定する必要があります。

例5.15 フィルタの定義と実装

```
@Entity
@Indexed
@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、スコアが 5 のドライバに結果セットを削減する単純な Lucene フィルタの例です。この例では、指定されたフィルタが **org.apache.lucene.search.Filter** を直接実装し、引数を持たないコンストラクタを含みます。

フィルタ作成で追加のステップが必要な場合、または使用するフィルタが引数を持たないコンストラクタを含まない場合は、以下のファクトリパターンを使用できます。

例5.16 ファクトリパターンを使用したフィルタの作成

```
@Entity
@Indexed
@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** によりアノテートされたメソッドを探し、それを使用してフィルタインスタンスを構築します。ファクトリは引数を持たないコンストラクタを持つ必要があります。JBoss Seam に精通しているユーザーにとって、これはコンポーネントファクトリパターンに似ていますが、アノテーションは異なります。

名前付きフィルタはパラメータをフィルタに渡す必要がある場合に役に立ちます。たとえば、セキュリティフィルタは適用するセキュリティレベルを知りたいことがあります。

例5.17 定義されたフィルタへのパラメータの引き渡し

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("security").setParameter( "level", 5
);
```

■

各パラメータ名は、対象の名前付きフィルタ定義のフィルタまたはフィルタファクトリに対して関連付けられた設定メソッドを持つ必要があります。

例5.18 実際のフィルタ実装でのパラメータの使用

```
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)
);
    }
}
```

メソッドが **@Key** をアノテートし、**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は特別なコントラクトを持ちます。キーオブジェクトは、該当する **Filter** タイプが同じであり、パラメータのセットが同じである場合にのみ 2 つのキーが同じになるよう **equals()** / **hashcode()** を実装する必要があります。つまり、2 つのフィルタキーは、キーが生成されるフィルタを交換できる場合にのみ同じになります。キーオブジェクトはキャッシュメカニズムで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。

- フィルタキャッシュシステムを有効にした場合 (デフォルトで有効)
- フィルタがパラメータを持っている場合

ほとんどの場合、**StandardFilterKey** 実装を使用するだけで十分です。これにより、**equals()** / **hashcode()** 実装が同じ各パラメータとハードコードメソッドに委譲されます。

以前に述べたように、定義されたフィルタはデフォルトでキャッシュされ、キャッシュは、必要な場合にメモリを廃棄できるようハード参照とソフト参照の組み合わせを使用します。ハード参照キャッシュは最後に使用されたフィルタを追跡し、必要な場合に使用頻度が最も低いフィルタを **SoftReferences** に変換します。ハード参照キャッシュの制限に達したら、追加のフィルタは **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ため

に、**hibernate.search.filter.cache_strategy.size** (デフォルトで 128) を使用します。フィルタキャッシュの上級ユーザーの場合は、独自の **FilterCachingStrategy** を実装できます。クラス名は **hibernate.search.filter.cache_strategy** により定義されます。

このフィルタキャッシュメカニズムを実際のフィルタ結果のキャッシュと混同しないでください。Lucene では、**CachingWrapperFilter** の周りで **IndexReader** を使用してフィルタをラップすることは一般的です。コストが高井再計算を回避するために、ラッパーは **getDocIdSet(IndexReader reader)** メソッドから返された **DocIdSet** をキャッシュします。計算された **DocIdSet** は同じ **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に注意してください (リーダーはインスタンスが開かれたときのインデックスの状態を表すため)。ドキュメントリストは開かれた **IndexReader** 内で変更できません。ただし、キャッシュされた **DocIdSet** を再計算する必要があるため、異なる/新しい **IndexReader** インスタンスは、**Document** の異なるセット上で動作します (異なるインデックスから、または単にインデックスが変更されたため)。

また、Hibernate Search はキャッシュのこの側面でも役に立ちます。デフォルトでは、**@FullTextFilterDef** の **cache** フラグは **FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS** に設定されます。この結果、フィルタインスタンスが自動的にキャッシュされ、**CachingWrapperFilter** の Hibernate 固有の実装の周りに指定されたフィルタがラップされます (**org.hibernate.search.filter.CachingWrapperFilter**)。このクラス **SoftReference** の Lucene のバージョンは、ハード参照数とともに使用されます (フィルタキャッシュの説明を参照)。ハード参照数は **hibernate.search.filter.cache_docidresults.size** (デフォルト値は 5) を使用して調整できます。ラップの動作は **@FullTextFilterDef.cache** パラメータを使用して制御できます。このパラメータには 3 つの異なる値が存在します。

値	定義
FilterCacheModeType.NONE	Hibernate Search によりフィルタインスタンスと結果がキャッシュされません。各フィルタコールごとに、新しいフィルタインスタンスが作成されます。この設定はすぐに変わるデータセットやメモリの制約が強い環境に役に立つことがあります。
FilterCacheModeType.INSTANCE_ONLY	フィルタインスタンスはキャッシュされ、同時 **Filter.getDocIdSet()** コールで再利用されます。**DocIdSet** の結果はキャッシュされません。この設定は、フィルタが独自のキャッシュメカニズムを使用したり、フィルタの結果がアプリケーション固有のイベントのため動的に変化したりする場合 (両方の場合に **DocIdSet** キャッシュは不必要になります) に役に立ちます。
FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS	フィルタインスタンスと **DocIdSet** 結果の両方はキャッシュされます。これはデフォルト値です。

なぜフィルタをキャッシュする必要があるのでしょうか? フィルタキャッシュが有用な場合は以下の 2 つです。

- システムが対象のエンティティインデックスを頻繁に更新しない (つまり、IndexReader が頻繁に再利用される)
- フィルタの DocIdSet を計算するのにコストがかかる (クエリを実行するのにかかる時間と比較)

5.4. クエリプロセスの最適化

クエリパフォーマンスは複数の条件に依存します。

- Lucene クエリ自体。この主題に関する資料を参照
- ロードされたオブジェクトの数。ページ処理をまたはプロジェクション (必要な場合) を使用
- Hibernate Search が Lucene リーダーとどのように対話するか。適切な「リーダー方針」を定義

5.5. ネイティブ LUCENE クエリ

Lucene のいくつかの固有の機能を使用する場合は、Lucene 固有のクエリを常に実行できます。詳細については、8章*高度な機能* を参照してください。

第6章 手動インデックス化

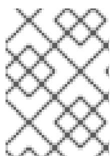
6.1. インデックス化

エンティティがデータベースに対して挿入または更新されない場合でもエンティティのインデックス化が役に立つ場合があります。 これは、たとえば、初めてインデックスを構築する場合に該当します。これは、**FullTextSession.index()** を使用して実行できます。

例6.1 FullTextSession.index() を使用したエンティティのインデックス化

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
    fullTextSession.index(customer);
}
tx.commit(); //index are written at commit time
```

効率を最大化するために、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。ただし、たくさんのデータをインデックス化する場合は、トランザクションのコミット時まですべてのドキュメントがキューに保持されるため、メモリ消費について気を付ける必要があります。**OutOfMemoryException** が発生することがあります。この例外を回避するには、**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場合)、バッチキューが処理され (メモリが解放される) すべてのインデックスの変更が適用されます。破棄された変更はロールバックできないことに注意してください。



注記

hibernate.search.worker.batch_size は廃止され、より適切に制御できるこの明示的な API に取って代わりました。

インデックス化時間とメモリ消費に影響を与える可能性がある他のパラメータは以下のとおりです。

- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_buffered_docs**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_field_length**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_merge_docs**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].merge_factor**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].ram_buffer_size**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].term_index_interval**

これらのパラメータは Lucene 固有であり、 Hibernate Search はこれらのパラメータを単に渡します。詳細については、「Lucene インデックス化パフォーマンスのチューニング」 を参照してください。

例6.2 特定のクラスを効率的にインデックス化 (インデックスの (再) 初期化に有用)

```
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(); //clear since the queue is processed
    }
}
transaction.commit();
```

アプリケーションのメモリが足りなくならないバッチサイズをい使用してみてください。

6.2. パージ

データベースから特定のタイプのエンティティを物理的に削除せずに 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** から実行されます。

例6.3 インデックスあkらエンティティの特定のインスタンスをパージ

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
    fullTextSession.purge( Customer.class, customer.getId() );
}
tx.commit(); //index are written at commit time
```

パージにより、Lucene インデックスから特定の id を持つエンティティが削除されますが、データベースは変更されません。

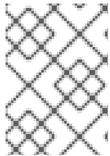
特定のタイプのすべてのエンティティを削除する必要がある場合は、**purgeAll** メソッドを使用できます。この操作はパラメータおよびすべてのサブタイプとして渡されたタイプのすべてのエンティティを削除します。

例6.4 インデックスからエンティティのすべてのインスタンスをパージ

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

```
fullTextSession.purgeAll( Customer.class );
//optionally optimize the index
//fullTextSession.getSearchFactory().optimize( Customer.class );
tx.commit(); //index are written at commit time
```

このような操作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** でも利用可能です。

第7章 インデックスの最適化

場合によっては、Lucene インデックスを最適化する必要があります。このプロセスは実際にはデフラグメンテーションです。最適化が実行されるまで、Lucene は削除されたドキュメントのみをマークし、物理的な削除は適用されません。最適化プロセスの間、削除は適用され、Lucene Directory の複数のファイルにも影響を与えます。

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) パフォーマンスは影響を受けません。最適化中に、検索は実行できますが、ほとんどの場合処理が遅くなります。すべてのインデックス更新は停止されます。最適化をスケジュールすることが推奨されます。

- アイドル状態のシステム上、または検索の頻度が低い場合
- 大量のインデックス変更後

7.1. 自動最適化

Hibernate Search は以下の処理後に自動的にインデックスを最適化できます。

- 特定の数の操作 (挿入、削除)
- または特定の数のトランザクション

自動インデックス最適化の設定は、グローバルレベルまたはインデックスごとに定義できます。

例7.1 自動最適化パラメータの定義

```
hibernate.search.default.optimizer.operation_limit.max = 1000
hibernate.search.default.optimizer.transaction_limit.max = 100
hibernate.search.Animal.optimizer.transaction_limit.max = 50
```

最適化は以下の場合すぐに **Animal** インデックスに対してトリガされます。

- 追加および削除の数が 1000 に達する
- トランザクションの数が 50 に達する (**hibernate.search.Animal.optimizer.transaction_limit.max** は **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が定義されていない場合、最適化は自動的に処理されません。

7.2. 手動最適化

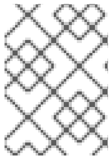
Hibernate Search の **SearchFactory** を使用して、Lucene インデックスをプログラムで最適化 (デフラグ) できます。

例7.2 プログラムによるインデックス最適化

```
FullTextSession fullTextSession =
Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

```
searchFactory.optimize(Order.class);
// or
searchFactory.optimize();
```

最初の例では **Order** を保持する Lucene インデックスを最適化します。2 つ例では、すべてのインデックスを最適化します。



注記

searchFactory.optimize() は JMS バックエンドに影響を与えます。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。

7.3. 最適化の調整

Apache Lucene は、最適化の実行方法に影響をい与えるいくつかのパラメータを持ちます。Hibernate Search はこれらのパラメータを公開します。

他のインデックス最適化パラメータは以下のとおりです。

- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_buffered_docs**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_field_length**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_merge_docs**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].merge_factor**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].ram_buffer_size**
- **hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].term_index_interval**

詳細については、「Lucene インデックス化パフォーマンスのチューニング」 を参照してください。

第8章 高度な機能

8.1. SEARCHFACTORY

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。また、このオブジェクトを使用することにより、Lucene にネイティブな方法で簡単にアクセスできます。**SearchFactory** は **FullTextSession** からアクセスできます。

例8.1 SearchFactory へのアクセス

```
FullTextSession fullTextSession =
Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

8.2. LUCENE ディレクトリへのアクセス

Lucene ディレクトリにはプレーンな Lucene を使用して常にアクセスできます。ディレクトリ構造は Hibernate Search を使用する場合でも使用しない場合でも変わりません。ただし、特定のいディレクトリにアクセスするにはいつくかの便利な方法があります。**SearchFactory** はインデックス化されたクラスごとに **DirectoryProvider** を追跡します。クラスが基礎となる同じインデックスディレクトリを共有する場合は、1 つのディレクトリプロバイダをインデックス化された複数のクラスで共有できます。通常は当てはまりませんが、インデックスが共有された場合、特定のエンティティは複数の **DirectoryProvider** を持つことができます (「インデックスの分割」 を参照)。

例8.2 Lucene Directory へのアクセス

```
DirectoryProvider[] provider =
searchFactory.getDirectoryProviders(Order.class);
org.apache.lucene.store.Directory directory =
provider[0].getDirectory();
```

この例では、ディレクトリは **Order** 情報を格納する lucene インデックスを参照します。取得された Lucene ディレクトリはクローズしないでください (これは Hibernate Search の責任です)。

8.3. INDEXREADER の使用

Lucene のクエリは **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すべてのインデックスリーダーをキャッシュします。コードはこのキャッシュされたリソースにアクセスできますが、 "good citizen" ルールに従う必要があります。

例8.3 IndexReader へのアクセス

```
DirectoryProvider orderProvider =
searchFactory.getDirectoryProviders(Order.class)[0];
DirectoryProvider clientProvider =
searchFactory.getDirectoryProviders(Client.class)[0];

ReaderProvider readerProvider = searchFactory.getReaderProvider();
IndexReader reader = readerProvider.openReader(orderProvider,
```

```
clientProvider);

try {
    //do read-only operations on the reader
}
finally {
    readerProvider.closeReader(reader);
}
```

ReaderProvider (「リーダー方針」 で説明) はディレクトリプロバイダにより参照されたインデックス上で IndexReader をオープンにします。この **IndexReader** は複数のクライアントで共有されるため、以下のルールに従う必要があります。

- indexReader.close() は呼び出さず、常に readerProvider.closeReader(reader) を呼び出します (理想的には Finally Block で)。
- 変更操作にはこの **IndexReader** を使用しないでください (例外が発生します)。読み取り/書き込みインデックスリーダーを使用する場合は、Lucene Directory オブジェクトからオープンします。

これらのルール以外に、特にネイティブクエリを実行するために IndexReader を自由に使用できます。共有された **IndexReader** を使用すると、ほとんどのクエリが効率的になります。

8.4. LUCENE のスコア式のカスタマイズ

Lucene では、ユーザーは **org.apache.lucene.search.Similarity** を拡張することによりスコア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ドキュメント d のクエリ q のスコアを計算する以下の式の係数に一致します。

$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t\ in\ q} (tf(t\ in\ d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d))$

係数	説明
tf(t ind)	ドキュメント (d) のターム (t)のターム頻度係数。
idf(t)	ドキュメントのタームの頻度を反転。
coord(q,d)	指定されたドキュメントで見付かったクエリタームの数に基づいたスコア係数。
queryNorm(q)	クエリ間のスコアを比較可能にするために使用する正規化係数。
t.getBoost()	フィールドブースト。
norm(t,d)	いくつかの (インデックス化時間) ブーストおよび長さ係数をカプセル化。

この式を詳しく説明することはこのマニュアルの範囲外です。詳細については、**Similarity** の Javadoc を参照してください。

Hibernate Search は、Lucene の近似値計算を変更する 2 つの方法を提供します。最初に、プロパティ **hibernate.search.similarity** を使用して **Similarity** 実装の完全修飾クラス名を指定してデフォルトの近似値を設定できます。デフォルト値は **org.apache.lucene.search.DefaultSimilarity** です。また、**@Similarity** アノテーションを使用してクラスレベルでデフォルトの近似値をオーバーライドできます。

```
@Entity
@Indexed
@Similarity(impl = DummySimilarity.class)
public class Book {
   ...
}
```

例として、ドキュメントにタームが現れる頻度は重要でないこととします。タームが 1 つのドキュメントのスコアは、複数のタームがあるドキュメントのスコアと同じになる必要があります。この場合、メソッド **tf(float freq)** のカスタム実装は 1.0 を返す必要があります。

付録A 改訂履歴

改訂 5.1.2-3.400 **2013-10-30** **Rüdiger Landmann**

Rebuild with publican 4.0.0

改訂 5.1.2-3 **2012-07-18** **Anthony Towns**

Rebuild for Publican 3.0

改訂 5.1.2-100 **Thu Dec 8 2011** **Jared Morgan**

JBoss Enterprise Application Platform 5.1.2 GAに対する変更を追加。本ガイド文書の変更に関する情報は、『リリースノート 5.1.2』を参照してください。

改訂 5.1.1-100 **Mon Jul 18 2011** **Jared Morgan**

JBoss Enterprise Application Platform 5.1.1 GAに対する変更を追加。本ガイド文書の変更に関する情報は、『リリースノート 5.1.1』を参照してください。

改訂 5.1.0-100 **Wed Sep 15 2010** **Laura Bailey**

新しいバージョニングの要件に沿ってバージョン番号を変更
JBoss Enterprise Application Platform 5.1.0.GA 向けに改訂