Hypersistence Optimizer – Kann die Optimierung der Datenzugriffsschicht einfach sein?

Hinter den meisten Anwendungen und Websites befindet sich eine Datenquelle (z. B. eine Datenbank oder andere Dateien), in der alle Arten von Informationen gespeichert werden, die für das ordnungsgemäße Funktionieren des Systems erforderlich sind. Somit ist die Kommunikation der Anwendung mit der Datenquelle ein Schlüsselelement des Systems. Für diese Kommunikation ist die Datenzugriffsschicht verantwortlich.

In der Java-Welt ist die objektrelationale Abbildung eine beliebte Methode zur Implementierung dieser Schicht. Die objektrelationale Abbildung besteht darin, die objektorientierte Architektur des Systems in einer relationalen Datenbank wiederzugeben. Das beliebteste Framework für die objektrelationale Abbildung ist Hibernate, eine Implementierung des JPA-Standards. Hibernate ist ein complexes und leistungsstarkes Tool und daher nicht einfach zu benutzen. Während der Konfiguration und Implementierung können viele Fehler gemacht werden, die sich erheblich negativ auf die Leistung des gebauten Systems auswirken können. Das Finden von Fehlern und die anschließende Optimierung können sehr zeitaufwändig sein. Abhilfe schafft bei diesem Problem der Hypersistence Optimizer.

*„Imagine having a tool that can **automatically** detect if you are using JPA and Hibernate properly.”* – Hypersistence Optimizer

Hypersistence Optimizer – Was ist das?

Hypersistence Optimizer ist ein Tool zur automatischen Analyse der Datenzugriffsschicht, das eine Vielzahl von Hibernate-Versionen unterstützt (v.3.3 - 5.4). Der Analyseprozess umfasst die

Konfiguration von Hibernate, die Entitätszuordnung, Datenbankabfragen und den Persistance Context. Basierend auf der Analyse erhalten wir einen detaillierten Bericht über die in der Anwendung gefundenen Probleme. Ein solcher Bericht enthält unter anderem Informationen über den Schweregrad des gefundenen Problems, seinen Lokalisation, eine kurze Beschreibung des Problems und einen Link zu einer detaillierteren Erläuterung mit Tipps und Beispielen. Das Tool wurde von Hypersistence erstellt und wird ständig weiterentwickelt. Derzeit kann das Tool über 50 verschiedene Arten von Fehlern erkennen. Den Hypersistence Optimizer gibt es in zwei Versionen: Der Voll -und Testversion. Die Testversion ist kostenlos, aber im Vergleich zur Vollversion sehr begrenzt. Die Testversion erfasst nur die Art und die Menge der Fehler und gibt nicht an, wo diese Fehler auftreten und wie mit ihnen umgegangen werden soll. Bevor Sie die Vollversion kaufen, sollten Sie anhand der Testversion prüfen, ob die erstellte Anwendung Fehler enthält.

Die ersten Schritte

Der Installationsvorgang besteht aus wenigen einfachen Schritten. Stellen Sie vor der Installation sicher, dass Maven auf dem Computer installiert ist, der für die Installation des Tools erforderlich ist. Führen Sie nach dem Herunterladen und Entpacken des Archivs die Batchdatei "maven-install" aus, die sich im extrahierten Ordner befindet. Hypersistence Optimizer wird dem Maven-Repository hinzugefügt. Der nächste Schritt, ist das Hinzufügen von der Abhängigkeit in die pom.xml Datei im Projekt:

```
1.  <dependency>
2.      <groupId>io.hypersistence</groupId>
3.      <artifactId>hypersistence-optimizer</artifactId>
4.      <version>2.1.1</version>
5.  </dependency>
```

Nach diesem Schritt ist das Tool einsatzbereit. Die Analyse wird auf der Ebene der Unit-Test ausgeführt. Erstellen Sie einen Test und initialisieren Sie den Optimizer. Es kann so aussehen:

```
1.   class HypersistenceTest {
2.
3.       @PersistenceUnit
4.       private EntityManagerFactory entityManagerFactory;
5.
6.       @PostConstruct
7.       public void init() {
8.           new HypersistenceOptimizer(new JpaConfig(entityManagerFactory));
9.       }
10.
11.      @Test
12.      void someTest() {
13.      }
14.  }
```

Wie wir sehen können, ist die Verwendung von Hypersistence Optimizer sehr einfach. Während der Initialisierung sollte nur der EntityManager angegeben werden, von dem das Tool die für die Analyse erforderlichen Metadaten abruft. Dies ist natürlich die grundlegendste Konfiguration. Die Konfigurationsoptionen sind umfangreicher. Sie können unter anderem festlegen, welcher Bereich der Datenzugriffsschicht analysiert werden soll (Konfiguration, Entitätszuordnung oder Datenbankabfragen), welche Ereignisse (Events) abgefangen werden sollen oder was mit dem erkannten Ereignissen geschehen soll. Jedes Problem das während der Analyse auftritt verursacht ein Ereignis, das sich auf die aktuell überprüfte Regel bezieht. Daher werden alle erkannten Probleme in Form von Ereignissen zurückgegeben, die einer bestimmten Regel und Beschreibung zugeordnet werden können. Standardmäßig analysiert das Tool alle Bereiche und erfasst alle aufgetretenen Ereignisse. Die Autoren geben eine detaillierte Beschreibung der Konfiguration, mit der Sie das Tool an Ihre Bedürfnisse anpassen können.

Der letzte Schritt besteht darin, die Standardkonfiguration von Hibernates SessionFactory mit den von den Erstellern bereitgestellten Decorator zu verpacken. Dieser Schritt ist erforderlich, wenn Ihre Datenbankabfragen in Echtzeit analysiert werden sollen. Wenn Sie Hibernate 5 verwenden, kopieren Sie einfach die Datei im Ordner ..\configs\META-INF\services in den Ordner ..\ src\ test\resources des aktuellen Projekts. Das auf diese Weise konfigurierte Tool funktioniert in der Testumgebung, ohne Probleme in der Produktionsumgebung zu verursachen. Für ältere Versionen von Hibernate muss man die SessionFactory manuell verpacken. Eine ausführliche Beschreibung finden Sie im mitgelieferten Konfigurationshandbuch.

Der Optimizer in Action

Nach dem Ausführen des Unit-Tests, in dem Hypersistence Optimizer konfiguriert wurde, wird ein Bericht in der Konsole angezeigt. Das Tool wurde an einer kleinen Anwendung zur Buchung von Firmenwagen getestet. Der erhaltene Bericht lautet wie folgt:

```
CRITICAL - IdentityGeneratorEvent - The [id] identifier attribute in the
[pl.fis.carrs.domain.model.Vehicle] entity uses the [IdentityGenerator] strategy, which prevents
Hibernate from enabling JDBC batch inserts. Consider using the SEQUENCE identifier strategy
instead. For more info about this event, check out this User Guide link -
https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#IdentityGeneratorEvent

MAJOR - EnumTypeStringEvent - The [fuelType] enum attribute in the
[pl.fis.carrs.domain.model.Vehicle] entity uses the EnumType.STRING strategy, which has a bigger
memory footprint than EnumType.ORDINAL. For more info about this event, check out this User
Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-
guide/#EnumTypeStringEvent

CRITICAL - BidirectionalSynchronizationEvent - The [reservations] bidirectional association in the
[pl.fis.carrs.domain.model.Vehicle] entity requires both ends to be synchronized. Consider adding
the [addReservation(pl.fis.carrs.domain.model.Reservation reservation)] and
[removeReservation(pl.fis.carrs.domain.model.Reservation reservation)] synchronization methods.
```

For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#BidirectionalSynchronizationEvent

CRITICAL - EagerFetchingEvent - The [vehicle] attribute in the [pl.fis.carrs.domain.model.Reservation] entity uses eager fetching. Consider using a lazy fetching which, not only that is more efficient, but it is way more flexible when it comes to fetching data. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#EagerFetchingEvent

MAJOR - SkipAutoCommitCheckEvent - You should set the [hibernate.connection.provider_disables_autocommit] configuration property to [true] while also making sure that the underlying DataSource is configured to disable the auto-commit flag whenever a new Connection is being acquired. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#SkipAutoCommitCheckEvent

CRITICAL - JdbcBatchSizeEvent - If you set the [hibernate.jdbc.batch_size] configuration property to a value greater than 1 (usually between 5 and 30), Hibernate can then execute SQL statements in batches, therefore reducing the number of database network roundtrips. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#JdbcBatchSizeEvent

CRITICAL - QueryPaginationCollectionFetchingEvent - You should set the [hibernate.query.fail_on_pagination_over_collection_fetch] configuration property to the value of [true], as Hibernate can then prevent in-memory pagination when join fetching a child entity collection. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#QueryPaginationCollectionFetchingEvent

MAJOR - QueryInClauseParameterPaddingEvent - You should set the [hibernate.query.in_clause_parameter_padding] configuration property to the value of [true], as Hibernate entity queries can then make better use of statement caching and fewer entity queries will have to be compiled while varying the number of parameters passed to the in query clause. For more info about this event, check out this User Guide link - https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#QueryInClauseParameterPaddingEvent

8 issues were found: 0 BLOCKER, 5 CRITICAL, 3 MAJOR, 0 MINOR

Basierend auf den im Bericht enthaltenen Informationen können wir beginnen, die Datenzugriffsschicht zu optimieren. Betrachten Sie beispielsweise den EagerFetchingEvent-Fehler, der als schwerwiegender Fehler eingestuft wird. Die Beziehung zwischen der Reservierung und dem Fahrzeug wurde als many-to-one definiert, es wurde jedoch keine Strategie zum Laden der Fahrzeugsammlung angegeben.

```
1.  @ManyToOne
2.  private Vehicle vehicle;
```

Bei Verwendung von many-to-one- und one-to-one-Beziehungen ist die Standard-Erfassungsstrategie FetchType.EAGER. Das EAGER Laden bedeutet, dass die Fahrzeugsammlung beim Herunterladen des übergeordneten Elements vollständig heruntergeladen wird, auch wenn wir sie überhaupt nicht benötigen. In diesem Fall wird empfohlen, die Strategie FetchType.LAZY zu verwenden, mit der die Sammlung bei Bedarf heruntergeladen wird. Der Code nach der Optimierung könnte folgendermaßen aussehen:

```
1. @ManyToOne (fetch = FetchType.LAZY)
2. @JoinColumn (name = "vehicle_id")
3. private Vehicle vehicle;
```

Hypersistence Optimizer erkennt potenzielle Bedrohungen in der Datenzugriffsschicht, die Performanceprobleme verursachen können (aber nicht müssen!). Dies sind jedoch nur Richtlinien, die berücksichtigt werden können, um tatsächliche oder potenzielle Probleme zu beseitigen. Spezifische Lösungen hängen von der Komplexität des gebauten Systems ab. Daher sollte jedes Problem einzeln betrachtet werden. Ein Beispiel wäre das EnumTypeStringEvent. In diesem Fall wird empfohlen, Aufzählungstypen in der Datenbank als EnumType.ORDINAL zu speichern. In der getesteten Anwendung werden die Aufzählungstypen als EnumType.STRING gespeichert. Wenn Sie von EnumType.STRING zu EnumType.ORDINAL wechseln, wird Speicherplatz gespart, aber die Lesbarkeit des Codes und die Lesbarkeit der Informationen in der Datenbank nehmen ab. Hier ist es Sache des Programmierers ob er den Code sauber halten oder bisschen Speicherplatz sparen möchte.

Der zweite Teil des Berichts enthält festgestellte Probleme im Zusammenhang mit den Datenbankabfragen (wird nur angezeigt, wenn Hibernate SessionFactory konfiguriert wurde):

```
CRITICAL - PaginationWithoutOrderByEvent - The [select v from VEHICLES v ] query uses pagination without an ORDER BY clause. Therefore, the result is not deterministic since SQL does not guarantee any particular ordering unless an ORDER BY clause is being used. For more info about this event, check out this User Guide link – https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#PaginationWithoutOrderByEvent

CRITICAL - PassDistinctThroughEvent - The [SELECT DISTINCT v FROM VEHICLES v left join v.reservations res where not exists… ] query uses DISTINCT to deduplicate the returned entities. However, without setting the [hibernate.query.passDistinctThrough] JPA query hint to [false], the underlying SQL statement will also contain DISTINCT, which will incur extra sorting and duplication removal execution stages. For more info about this event, check out this User Guide link – https://vladmihalcea.com/hypersistence-optimizer/docs/user-guide/#PassDistinctThroughEvent
```

In der aktuellen Version von Hypersistence Optimizer werden derzeit nur zwei Arten von Ereignissen bei der Datenbankabfrageanalyse erfasst: PaginationWithoutOrderByEvent und PassDistinctThroughEvent. Im Beispiel können wir sehen, dass selbst in einer so einfachen Abfrage

"select v from VEHICLES v " ein Fehler gemacht werden kann. Wenn Sie die Ergebnisse mit setMaxResults einschränken, müssen Sie Ihrer Abfrage die ORDER BY-Klausel hinzufügen.

Zusammenfassung

Das enorme Wissen und Engagement des Autors - Vlad Mihalcea als Sprecher der Programmierer für das Hibernate-Projekt, führte zur Schaffung eines großartigen Tools für die automatische Analyse der Datenzugriffsschicht, dem Hypersistence Optimizer. Dies spart Entwicklern viel Zeit und graues Haar, wenn sie nach den Ursachen für eine schlechte Performance der Anwendung suchen. Darüber hinaus können detaillierte Problembeschreibungen und Beispiele als Wissensbasis für jeden Programmierer dienen.

Autor: Denis Wiesner

Ingenieur, Java-Programmierer mit mehrjähriger Berufserfahrung im Backend und Frontend. Interessen: numerische Methoden, parallele Programmierung, neue Technologien.

#java #hibernate #jpa #orm #hypersistenceOptimizer #Optimierung