Динамические типобезопасные запросы в JPA 2.0

Узнайте, как интерфейс Criteria позволяет создавать динамические запросы и снижает риск возникновения ошибок на этапе выполнения

Опубликовано 23.03.2011

ТЕОРИЯ до 11 листа

С момента своего выпуска в 2006 г. JPA пользуется популярностью в среде разработчиков Java-приложений. Выход следующей версии спецификации – 2.0 (JSR 317) – намечен на конец 2009 г. (см. раздел [Ресурсы](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#artrelatedtopics)). Одной из ключевых новинок в JPA 2.0 должен стать API под названием Criteria, придающий языку Java уникальные возможности: он предоставляет средства для написания запросов, которые могут быть проверены на корректность на этапе компиляции. Кроме того, Criteria позволяет динамически формировать запросы на этапе выполнения приложения.

В этой статье рассказывается о самом API Criteria и тесно связанном с ним понятии *метамодели* (metamodel). В частности, вы узнаете о том, как использовать Criteria для создания запросов, корректность которых, в отличие от обычных строковых запросов на языке JPQL (Java Persistence Query Language), может быть проверена компилятором, что позволяет снизить риск возникновения ошибок на этапе выполнения приложения. Кроме того, вы увидите преимущества программных средств для формирования запросов перед фиксированным синтаксисом JPQL на примерах, в которых используются шаблоны и функции базы данных. Данная статья рассчитана на читателей, знакомых с языком программирования Java и основными компонентами JPA, такими как EntityManagerFactory и EntityManager.

Что не так с запросами JPQL?

В состав JPA 1.0 входит JPQL – мощный язык для написания запросов, который явился основной причиной высокой популярности JPA. Однако JPQL имеет определенные ограничения, поскольку позволяет создавать только строковые запросы с использованием фиксированного набора синтаксических конструкций. Одно из основных ограничений JPQL продемонстрировано в листинге 1, в котором представлен запрос для выборки списка экземпляров класса Person, чей возраст превышает 20 лет.

Листинг 1. Пример простого (и ошибочного) запроса JPQL

|  |  |
| --- | --- |
| 1  2  3  4 | EntityManager em = ...;  String jpql = "select p from Person where p.age > 20";  Query query = em.createQuery(jpql);  List result = query.getResultList(); |

Этот простой пример иллюстрирует следующие ключевые аспекты модели выполнения запросов, использующейся в JPA 1.0:

* запросы на языке JPQL задаются в виде строк (экземпляр String в строке 2);
* для создания объекта, представляющего собой *готовый к выполнению* запрос, на основе строки на языке JPQL используется класс-фабрика EntityManager (строка 3);
* результатом выполнения запроса является нетипизированный список (экземпляр java.util.List).

В этом простом примере кроется серьезная ошибка. Самый неприятный момент заключается в том, что код будет успешно скомпилирован, однако сгенерирует исключение на этапе выполнения, поскольку JPQL-строка запроса синтаксически некорректна. Правильный вариант выглядит следующим образом:

|  |  |
| --- | --- |
| 1 | String jpql = "select p from Person p where p.age > 20"; |

К сожалению, эта ошибка не может быть выявлена на этапе компиляции и возникнет только при выполнении третьей или четвертой строки листинга (в зависимости от того, в какой момент провайдер JPA будет разбирать строку в соответствии с синтаксисом JPQL: при создании запроса или его исполнении).

Преимущества типобезопасных запросов

Одним из ключевых преимуществ API Criteria является то, что он не позволяет создавать синтаксически некорректные запросы. В листинге 2 показан модифицированный вариант [листинга 1](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing1) с применением интерфейса CriteriaQuery.

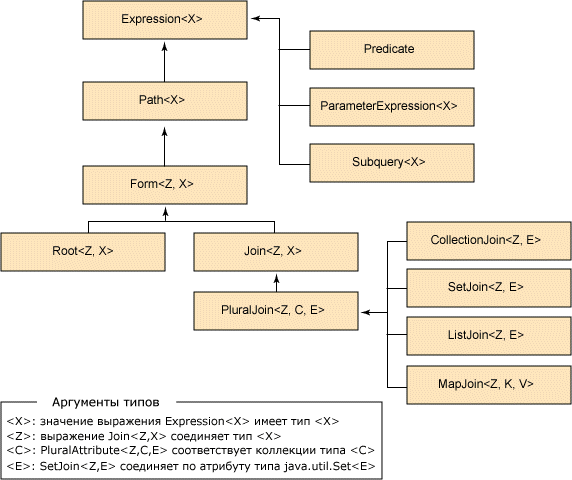
Листинг 2. Основные действия по созданию экземпляра CriteriaQuery

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | EntityManager em = ...  QueryBuilder qb = em.getQueryBuilder();  CriteriaQuery<Person> c = qb.createQuery(Person.class);  Root<Person> p = c.from(Person.class);  Predicate condition = qb.gt(p.get(Person\_.age), 20);  c.where(condition);  TypedQuery<Person> q = em.createQuery(c);  List<Person> result = q.getResultList(); |

В [листинге 2](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing2) демонстрируются основные принципы использования интерфейса Criteria и его базовые компоненты.

* В первой строке с помощью одного из поддерживаемых механизмов получается ссылка на экземпляр EntityManager.
* Во второй строке EntityManager создает экземпляр QueryBuilder, который представляет собой фабрику для инстанциирования CriteriaQuery.
* В третьей строке фабрика QueryBuilder создает типизированный экземпляр CriteriaQuery. Аргумент типа задает тип объектов, которые будут возвращены в результате выполнения запроса, представленного в виде экземпляра CriteriaQuery. При формировании запроса можно использовать различные варианты типов: от классов хранимых объектов, например, Person.class, до менее специализированных типов, таких как Object[].
* В четвертой строке для экземпляра CriteriaQuery задаются *выражения запроса* (query expressions). Полное описание каждого экземпляра CriteriaQuery представляет собой дерево, узлами которого являются подобные выражения. Иерархия типов выражений, поддерживаемых в Criteria API, показана на рисунке 1.

Рисунок 1. Иерархия интерфейсов, представляющих различные типы выражений в запросах



Вначале при помощи метода from указывается, что поиск будет осуществляться среди хранимых объектов типа Person.class. Данный метод возвращает экземпляр типа Root<Person>. Root представляет собой выражение, описывающее область определения некоторого множества сохраняемых объектов. Выражение Root<T> фактически означает, что данный запрос должен выполняться над всеми объектами типа T, тем самым напоминая выражение FROM в запросах SQL и JPQL. Обратите внимание, что выражение Root<Person> (как и любое выражение в запросе) является типизированным, причем параметр определяет тип значения выражения. Таким образом, тип Root<Person> определяет выражение, результатом вычисления которого будет экземпляр типа Person.class.

* В пятой строке формируется предикат (экземпляр класса Predicate).  Predicate представляет собой часто использующуюся разновидность выражений, результатом вычисления которых может быть true или false. Предикаты формируются при помощи класса QueryBuilder, который выполняет функции фабрики по отношению не только к запросам, но и к выражениям. QueryBuilder предоставляет методы для создания выражений всех видов, поддерживаемых стандартным языком JPQL, а также нескольких дополнительных разновидностей. В [листинге 2](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing2) показан пример использования QueryBuilder для создания выражения, которое принимает значение true в случае, если его первый аргумент численно больше второго. Метод gt имеет следующую сигнатуру:

|  |  |
| --- | --- |
| 1 | Predicate gt(Expression<? extends Number> x, Number y); |

* Этот метод является наглядной демонстрацией возможностей API, который, используя средства строго типизированного языка, такого как Java, позволяет создавать исключительно корректные выражения. Сигнатура метода недвусмысленно говорит том, что выражения типа Number могут сравниваться только с другими выражениями того же типа (но не, например, типа String).

|  |  |
| --- | --- |
| 1 | Predicate condition = qb.gt(p.get(Person\_.age), 20); |

* Однако пятая строка этим не ограничивается. Обратите внимание на первый параметр метода qb.gt(): p.get(Person\_.age),, в котором p — это ранее созданное выражение типа Root<Person>. В данном случае p.get(Person\_.age) является примером *выражения пути* (path expression). Подобные выражения определяют путь от корневого выражения, проходящий через один или несколько сохраняемых атрибутов. В данном примере p.get(Person\_.age) представляет собой путь от корневого выражения p через атрибут age класса Person. Возможно, запись Person\_.age покажется вам необычной, но пока просто запомните, что она означает "атрибут age класса Person". Этот вопрос будет более подробно рассмотрен ниже, в процессе обсуждения API Metamodel, который также вошел в состав JPA 2.0.
* Как уже было сказано, все выражения в запросах являются типизированными, причем параметры типов определяют тип их значений. Таким образом, если типом атрибута age для Person является Integer (или int), значение выражения p.get(Person\_.age) будет иметь тип Integer. Безопасность типов является неотъемлемым свойством данного API, поэтому компилятор будет автоматически сигнализировать об ошибке при попытках создания выражений, описывающих некорректные сравнения, например:

|  |  |
| --- | --- |
| 1 | Predicate condition = qb.gt(p.get(Person\_.age, "xyz")); |

* В строке 6 созданный выше предикат задается для экземпляра CriteriaQuery аналогично блоку WHERE в SQL.
* В строке 7 EntityManager создает готовый к выполнению экземпляр запроса на основе объекта типа CriteriaQuery. Этот шаг аналогичен формированию исполняемых запросов из строк на языке JPQL. Однако, поскольку CriteriaQuery заключает в себе более подробную информацию о типах, исполняемый запрос оказывается объектом типа TypedQuery, являющегося наследником javax.persistence.Query. Как следует из названия, тип TypedQuery несет в себе информацию о типе объектов, которые будут выбраны запросом. Этот интерфейс определен следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | public interface TypedQuery<T> extends Query {               List<T> getResultList();  } |

* При этом его нетипизированный родительский интерфейс выглядит так:

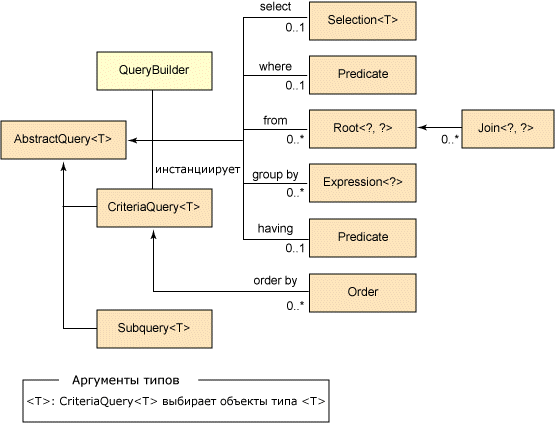
|  |  |
| --- | --- |
| 1  2  3 | public interface Query {  List getResultList();  } |

* Как и следовало ожидать, результаты данного запроса будут иметь тип Person.class, который был задан в строке 3 при формировании экземпляра CriteriaQuery средствами QueryBuilder.
* В строке 8, при выполнении запроса и получении списка результатов, наконец, становятся очевидными преимущества обладания информацией о типах. Заранее известно, что результатом запроса будет являться список объектов типа Person, благодаря чему удается избавиться от лишнего оператора приведения типа в процессе перебора элементов. Это также минимизирует риск возникновения исключения типа ClassCastException на этапе выполнения приложения.

Итак, подведем итоги рассмотрения простого примера, показанного в [листинге 2](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing2).

* CriteriaQuery представляет собой дерево, состоящее из узлов-выражений, при помощи которых задаются выражения, аналогичные блокам FROM, WHERE и ORDER BY в классических языках запросов. Поддерживаемые типы выражений показаны на рисунке 2.

Рисунок 2. Интерфейс CriteriaQuery поддерживает все типы выражений, соответствующие компонентам классических запросов



* Все выражения, использующиеся в запросах, являются типизированными. Примерами часто используемых выражений являются следующие:
  + Root<T> – представляет собой аналог блока FROM;
  + Predicate – результатом всегда является булево значение (true или false). Кстати говоря, интерфейс этих выражений определяется следующим образом: interface Predicate extends Expression<Boolean>;
  + Path<T> – описывает путь к сохраняемому атрибуту начиная от корневого выражения Root<*?*>. Само выражение Root<T> представляет собой частный случай Path<T>, не имеющий родительского выражения.
* QueryBuilder выполняет функции фабрики по созданию экземпляров CriteriaQuery, а также всех видов выражений запросов.
* На основе экземпляров CriteriaQuery создаются готовые к выполнению запросы, при этом вся информация о типах сохраняется. Это позволяет перебирать объекты полученного списка без приведения типов на этапе выполнения приложения.

Метамодель множества сохраняемых классов

В [листинге 2](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing2) нам встретилась необычная конструкция — Person\_.age, которая обозначает сохраняемый атрибут age класса Person. В том примере данная конструкция используется для формирования выражения, представляющего собой путь от корневого выражения p типа Root<Person> к атрибуту age при помощи вызова p.get(Person\_.age). Person\_.age является статическим полем в классе Person\_, который, в свою очередь, является примером *статического, инстанциированного, канонического*класса метамодели, соответствующего сохраняемому классу Person.

Класс метамодели содержит метаданные, относящиеся к сохраняемому классу. Он называется *каноническим*, если описывает метаинформацию о сохраняемом классе в точном соответствии со спецификацией JPA 2.0. Канонический класс метамодели является *статическим*, если все его переменные-члены объявлены как static и public. Примером такой переменной может служить статическое поле Person\_.age. *Инстанциирование* канонического класса заключается в генерации конкретного класса Person\_.java и включении его в исходный код приложения на этапе компиляции. Подобное инстанциирование позволяет обращаться к атрибутам класса Person в момент компиляции, обеспечивая тем самым строгий контроль типов.

Класс метамодели Person\_ представляет собой вариант хранения метаинформации о классе Person, являющийся альтернативным по отношению к часто используемому (возможно, даже слишком часто) механизму рефлексии Java (Java Reflection API). При этом между ними есть одно принципиальное различие: обращение к метаданным объекта Person.class, полученным при помощи рефлексии, не может контролироваться компилятором. Например, механизм рефлексии позволяет обратиться к полю age в Person.class следующим образом:

|  |  |
| --- | --- |
| 1 | Field field = Person.class.getField("age"); |

Однако подобный подход имеет недостатки, которые напоминают проблемы, связанные со строковыми запросами на языке JPQL и продемонстрированные в [листинге 1](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing1). Компиляция этого фрагмента кода проходит успешно, однако компилятор не может гарантировать, что при его выполнении не возникнет ошибок. В частности, к ошибке может привести обыкновенная опечатка. Таким образом, механизм рефлексии непригоден для обеспечения типобезопасности в запросах JPA 2.0.

Для обеспечения типобезопасности запросов необходим механизм, позволяющий обращаться к сохраняемому атрибуту age класса Person таким способом, который может быть проверен на корректность компилятором. Решение, предлагаемое в JPA 2.0, предоставляет такую возможность путем инстанциирования класса метамодели Person\_, который содержит статические аналоги всех сохраняемых атрибутов класса Person.

Как правило, обсуждения метаинформации (или даже мета-метаинформации) оказываются весьма скучными, поэтому мы рассмотрим конкретный пример класса метамодели, соответствующего простому сохраняемому Java-классу — domain.Person, код которого приведен в листинге 3.

Листинг 3. Пример простого сохраняемого класса

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package domain;  @Entity  public class Person {    @Id    private long ssn;    private string name;    private int age;      // открытые get- и set-методы    public String getName() {...}  } |

Этот класс является классическим примером POJO (простого Java-класса), содержащего аннотации, в частности, @Entity и @Id , которые необходимы провайдеру JPA для управления сохраняемыми экземплярами.

Данному классу соответствует статический канонический класс метамодели, показанный в листинге 4.

Листинг 4. Пример простого канонического класса метамодели

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | package domain;  import javax.persistence.metamodel.SingularAttribute;    @javax.persistence.metamodel.StaticMetamodel(domain.Person.class)    public class Person\_ {    public static volatile SingularAttribute<Person,Long> ssn;    public static volatile SingularAttribute<Person,String> name;    public static volatile SingularAttribute<Person,Integer> age;  } |

Все сохраняемые атрибуты класса domain.Person присутствуют в классе метамодели в виде открытых статических переменных-членов типа SingularAttribute<Person,*?*>. Благодаря этому можно ссылаться на атрибуты domain.Person (например, age) не через механизм рефлексии, а непосредственно на этапе компиляции, используя соответствующие статические члены, в частности, Person\_.age. В этом случае компилятор может проверить совместимость типов, поскольку он знает объявленный тип атрибута age. Подобный контроль типов был продемонстрирован выше на примере QueryBuilder.gt(p.get(Person\_.age), "xyz"). Попытка формирования такого выражения приведет к ошибке компиляции, поскольку, проанализировав сигнатуру метода QueryBuilder.gt(..)  и тип Person\_.age, компилятор сделает вывод, что атрибут age класса Person имеет числовой тип и, следовательно, не может сравниваться с объектами типа String.

Необходимо также отметить ряд других важных моментов:

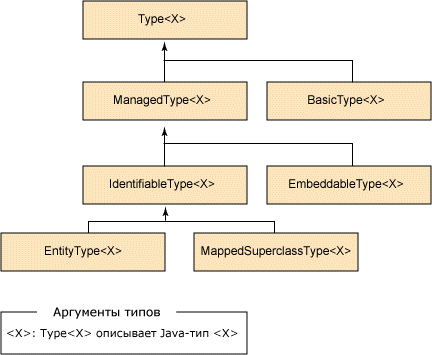
* Атрибут метамодели Person\_.age имеет тип javax.persistence.metamodel.SingularAttribute. SingularAttribute – это один из интерфейсов, определенных в рассматриваемом в следующем разделе API Metamodel, который входит в состав JPA. Аргументы типа SingularAttribute<Person, Integer> определяют класс, к которому принадлежит сохраняемый атрибут, и тип самого сохраняемого атрибута.
* Класс метамодели отмечен аннотацией @StaticMetamodel(domain.Person.class), которая указывает на то, что он соответствует исходной сохраняемой сущности domain.Person.

API Metamodel

Как было сказано выше, классы метамодели служат для описания сохраняемых классов модели приложения. Аналогично механизму рефлексии, в котором для описания компонентов java.lang.Class используются специальные интерфейсы, такие как java.lang.reflect.Field или java.lang.reflect.Method , в API Metamodel JPA для описания типов и атрибутов класса метамодели также применяются дополнительные интерфейсы, в частности,SingularAttribute и PluralAttribute,.

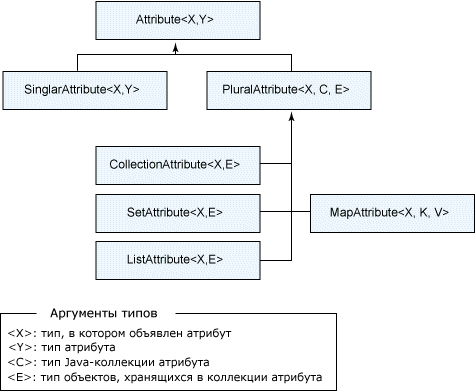
Набор интерфейсов, использующихся в API Metamodel для описания типов, показан на рисунке 3.

Рисунок 3. Иерархия интерфейсов API Metamodel для описания сохраняемых типов



На рисунке 4 показаны интерфейсы API Metamodel, которые используются для описания атрибутов.

Рисунок 4. Иерархия интерфейсов API Metamodel для описания сохраняемых атрибутов



Интерфейсы API Metamodel в JPA являются более специализированными, чем те, которые используются в механизме рефлексии Java Reflection API. Это необходимо для представления расширенной метаинформации о сохраняемых классах. Например, в Java Reflection API все типы Java представляются в виде объектов класса java.lang.Class, т. е. не делается особых различий между такими понятиями, как класс, абстрактный класс и интерфейс. Разумеется, экземпляры Class предоставляют информацию о том, являются они интерфейсами или, например, абстрактными классами, однако это далеко не то же самое, что использование различных определений для задания интерфейсов и классов.

Java Reflection API появился на заре Java и в то время представлял для языка программирования общего назначения вполне новаторское решение. Однако с годами понимание области применения и возможностей строго типизированных систем развивалось, что в итоге привело к появлению в JPA API Metamodel, который реализует эти возможности, вводя строгую типизацию для сохраняемых сущностей. Например, сохраняемые сущности делятся на семантические категории – MappedSuperClass, Entity и Embeddable. До JPA 2.0 принадлежность классов этим категориям задавалась при помощи аннотаций в их определениях, в то время как API Metamodel предоставляет специализированные интерфейсы MappedSuperclassType, EntityType и EmbeddableType в пакете javax.persistence.metamodel. Это позволяет подчеркнуть семантические особенности типов. Аналогичным образом сохраняемые атрибуты различаются на уровне определений типов благодаря таким интерфейсам, как SingularAttribute, CollectionAttribute и MapAttribute.

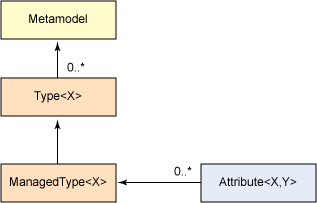
Подобные специализированные интерфейсы метамодели дают не только эстетические преимущества, но и помогают формировать типобезопасные запросы, снижая тем самым риск возникновения ошибок на этапе выполнения. Некоторые из этих преимуществ были продемонстрированы ранее на примерах, но еще ярче они будут видны при создании запросов с операторами соединения (join) при помощи CriteriaQuery.

Область видимости на этапе выполнения

В широком смысле можно провести определенные параллели между классическими интерфейсами Java Reflection API и специализированными интерфейсами пакета javax.persistence.metamodel, служащими для представления метаинформации о сохраняемых классах. В продолжение этой аналогии можно ввести для интерфейсов метамодели аналог понятия области видимости времени выполнения (run-time scope). Область видимости объектов java.lang.Class ограничивается используемым загрузчиком классов (java.lang.ClassLoader), поэтому все экземпляры Java-классов, ссылающиеся друг на друга, должны находиться в области одного загрузчика. Другими словами, множество связанных классов является *строгим* (или *замкнутым*), что говорит о том, что если в классе A, находящемся в области загрузчика L, определить ссылку на объект класса B, который находится вне области загрузчика L, то будет сгенерировано исключение типа ClassNotFoundException или NoClassDefFoundError (подобная ситуация способна надолго лишить сна разработчика или инженера, отвечающего за развертывание приложения в среде с несколькими загрузчиками классов).

Это понятие области видимости на этапе выполнения, означающее замкнутость множества ссылающихся друг на друга классов, получило название *единицы хранения* (persistence unit) в JPA 1.0. Область каждой единицы хранения определяется путем перечисления всех сохраняемых классов в секции <class> в файле META-INF/persistence.xml. JPA 2.0 предоставляет доступ к определению этой области на этапе выполнения через интерфейс javax.persistence.metamodel.Metamodel, который содержит список всех сохраняемых классов, принадлежащих конкретной единице хранения (рисунок 5).

Рисунок 5. Интерфейс Metamodel представляет собой контейнер типов, составляющих единицу хранения



Этот интерфейс позволяет обращаться к элементам метамодели через соответствующие классы сохраняемых сущностей. Например, получить ссылку на класс метамодели, описывающий сохраняемый класс Person, можно следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | EntityManagerFactory emf = ...;  Metamodel metamodel = emf.getMetamodel();  EntityType<Person> pClass = metamodel.entity(Person.class); |

Несмотря на несколько отличающиеся принципы и стиль кода, этот подход аналогичен получению экземпляра Class по его имени через загрузчик классов.

|  |  |
| --- | --- |
| 1  2 | ClassLoader classloader =  Thread.currentThread().getContextClassLoader();  Class<?> clazz = classloader.loadClass("domain.Person"); |

Класс EntityType<Person> позволяет во время выполнения просматривать все сохраняемые атрибуты класса Person. Например, вызов в приложении метода pClass.getSingularAttribute("age", Integer.class) вернет экземпляр типа SingularAttribute<Person, Integer>, который эквивалентен статическому атрибуту Person\_.age инстанциированного канонического класса метамодели. Таким образом, компилятор получает доступ к метаданным о сохраняемых атрибутах, к которым во время выполнения можно обращаться через API Metamodel, благодаря инстанциированию статического канонического класса метамодели Person\_ .

API Metamodel не только обеспечивает доступ к элементам метамодели, описывающим тот или иной сохраняемый класс, но и позволяет получить список всех известных классов метамодели (метод Metamodel.getManagedTypes()). Наконец, через этот API можно получить доступ к классу метамодели на основе специализированной информации о сохраняемом классе. Примером такой возможности может служить вызов метода embeddable(Address.class), который возвращает объект типа EmbeddableType<Address>, являющегося дочерним интерфейсом ManagedType<>.

В JPA метаинформация о классах POJO также включает в себя специальные компоненты, касающиеся хранения экземпляров. К ним относятся данные о том, является ли класс вложенным (embedded), а также о том, какие поля составляют первичный ключ. Эта информация содержится в исходном коде классов в виде аннотаций (XML-дескрипторов), которые делятся на две основные категории: аннотации, относящиеся непосредственно к сохранению объектов (например, @Entity), и аннотации, описывающие отображение (например, @Table). В JPA 2.0 метамодель содержит *исключительно* аннотации первого типа. Другими словами, текущая версия API Metamodel позволяет узнавать, какие атрибуты классов являются сохраняемыми, но *не* позволяет определять, в каких столбцах таблиц базы данных они сохраняются.

Канонические и неканонические классы метамодели

Несмотря на то, что структура канонических статических классов метамодели полностью контролируется спецификацией JPA 2.0 (включая полностью квалифицированные имена классов и имена их статических членов), разработчики также могут создавать классы метамодели самостоятельно. Подобные метаклассы, созданные вручную, называются *неканоническими*. На данный момент они не очень подробно описаны в спецификации и могут быть не полностью переносимыми с одного провайдера JPA на другой. Возможно, вы заметили, что открытые статические поля канонической метамодели *объявлены*, но не инициализированы. Поскольку они объявлены, к ним можно обращаться в процессе создания запросов при помощи CriteriaQuery, однако на этапе выполнения им должны быть присвоены значения. За инициализацию статических полей канонических метаклассов отвечает провайдер JPA, однако эта ответственность не распространяется на их неканонические аналоги. Таким образом, приложениям, использующим неканонические метаклассы, приходится либо полагаться на специфическую функциональность провайдеров, либо реализовывать собственные механизмы для инициализации полей классов метамодели на этапе выполнения.

**Практические аспекты генерации кода**

Автоматическая генерация кода часто смущает разработчиков, тем более если речь идет о генерации исходного кода канонических метаклассов. Данные классы используются в процессе разработки, в частности, при компиляции на них могут ссылаться компоненты приложения, отвечающие за формирование выражений CriteriaQuery. В результате возникают следующие практические вопросы:

* Где должны располагаться сгенерированные файлы с исходным кодом метаклассов: в каталоге с первоначальным исходным кодом, в отдельном каталоге или внутри каталога, в который помещается скомпилированный код?
* Должен ли исходный код сгенерированных классов помещаться в систему управления конфигурированием приложения с контролем версий?
* Каким образом должно поддерживаться соответствие между классами модели (например, Person) и метамодели (соответственно Person\_)? В частности, какие действия необходимо выполнить после изменения Person.java, например, после добавления или переименования одного из сохраняемых атрибутов?

На момент написания данной статьи ответы на эти вопросы остаются на усмотрение разработчиков.

Обработка аннотаций и генерация метаклассов

Понятно, что, работая с большим числом сохраняемых классов, вам не захочется самостоятельно писать классы метамодели. Предполагается, что эту задачу *должен* взять на себя провайдер механизма персистентности. Подобная функциональность и механизмы генерации строго не регламентируются в спецификации, однако среди провайдеров JPA существует негласная договоренность, что провайдеры должны генерировать каноническую метамодель при помощи процессора аннотаций (Annotation Processor), который входит в состав компилятора Java 6. Реализация Apache OpenJPA включает в себя утилиту для генерации классов метамодели либо неявным образом, в момент компиляции исходного кода, либо через вызов специального скрипта. В более ранних версиях Java для этих целей использовался отдельный процессор аннотаций под названием apt, однако в Java 6 взаимодействие между компилятором и процессором аннотаций стало частью стандарта.

Если вы используете в качестве провайдера персистентности OpenJPA, для генерации классов метамодели достаточно просто добавить библиотеки классов OpenJPA в classpath компилятора при компиляции POJO:

|  |  |
| --- | --- |
| 1 | $ javac domain/Person.java |

При этом будет сгенерирован канонический класс метамодели Person\_. Он будет помещен в ту же директорию, что и Person.java, и скомпилирован параллельно с ним.

**Создание запросов**

До сих пор мы рассматривали компоненты интерфейса CriteriaQuery и связанные с ними классы метамодели. Теперь пришло время приступить непосредственно к написанию запросов с использованием API Criteria.

Функциональные выражения

Функциональные выражения применяют функцию к одному или нескольким аргументам, что приводит к созданию нового выражения. Тип результирующего выражения определяется как самой функцией, так и типами ее аргументов, которые, в свою очередь, могут быть примитивными значениями (литералами) или выражениями. При этом механизм контроля типов компилятора и сигнатуры методов API определяют, какие выражения являются корректными, а какие нет.

Рассмотрим выражение, вычисляющее среднее значение для единственного входного параметра. Пример его использования приведен в листинге 5, в котором CriteriaQuery применяется для подсчета среднего значения баланса по всем счетам (объектам типа Account).

Листинг 5. Пример использования функционального выражения в CriteriaQuery

|  |  |
| --- | --- |
| 1  2  3  4 | CriteriaQuery<Double> c = cb.createQuery(Double.class);  Root<Account> a = c.from(Account.class);    c.select(cb.avg(a.get(Account\_.balance))); |

Эквивалентный запрос на языке JPQL выглядит следующим образом:

|  |  |
| --- | --- |
| 1 | String jpql = "select avg(a.balance) from Account a"; |

В [листинге 5](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing5) выражение avg() создается с помощью фабрики QueryBuilder (переменная cb), а затем используется в блоке select() запроса.

Выражение запроса представляет собой конструктивный блок, из таких блоков собирается итоговый предикат, который будет использоваться для выборки данных. В примере, приведенном в [листинге 6](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing6), показано выражение Path, представляющее собой переход от экземпляра Account к его атрибуту balance. Затем это выражение используется в качестве входного аргумента в двух бинарных функциональных выражениях: greaterThan() и lessThan(), результатами которых являются предикаты (булевы выражения). Далее полученные булевы выражения объединяются методом and() в итоговый предикат поиска, который будет вычисляться в блоке where при выполнении запроса.

**Бесстыковый API**

Как видно из этого примера, методы Criteria часто возвращают экземпляры таких типов, которые могут немедленно использоваться в последующих вызовах сопутствующих методов. Интерфейсы, поддерживающие подобный интуитивный стиль программирования, получили название "бесстыковых API" (Fluent API).

Листинг 6. Пример формирования **предиката where()** в CriteriaQuery

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CriteriaQuery<Account> c = cb.createQuery(Account.class);  Root<Account> account = c.from(Account.class);  Path<Integer> balance = account.get(Account\_.balance);  c.where(cb.and         (cb.greaterThan(balance, 100),          cb.lessThan(balance, 200))); |

На JPQL данный запрос выглядел бы следующим образом:

|  |  |
| --- | --- |
| 1 | "select a from Account a where a.balance>100 and a.balance<200"; |

**Сложные предикаты**

Некоторые выражения, в частности, in(), могут вычисляться над различным числом аргументов (дочерних выражений). Пример показан в листинге 7.

Листинг 7. Пример выражения CriteriaQuery, принимающего множество аргументов

|  |  |
| --- | --- |
| 1  2  3  4  5 | CriteriaQuery<Account> c = cb.createQuery(Account.class);  Root<Account> root = c.from(Account.class);  Path<Person> owner = root.get(Account\_.owner);  Path<String> name = owner.get(Person\_.name);  c.where(cb.in(name).value("X").value("Y").value("Z")); |

запрос на JPQL:

|  |  |
| --- | --- |
| 1 | "select a from Account a where a.owner.name in ('X','Y','Z')"; |

В этом примере формируется выражение, обозначающее путь от счета к имени его владельца, которое затем используется в качестве входного аргумента для выражения in(). Значением in() является true в том случае, если входное выражение равно одному из переменного числа его аргументов. Эти аргументы задаются при помощи метода value() типа In<T>, который имеет следующую сигнатуру:

|  |  |
| --- | --- |
| 1 | In<T> value(T value); |

Обратите внимание на параметризацию типа In<T>, которая означает, что данное выражение может вычисляться только для аргументов типа T. Выражение пути, указывающее на имя владельца счета (Account), имеет тип String, поэтому оно может сравниваться только со строковыми аргументами – либо выражениями типа String, либо строковыми литералами.

Сравните запрос в [листинге 7](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing7) с равнозначным ему (правильным) запросом на JPQL:

|  |  |
| --- | --- |
| 1 | "select a from Account a where a.owner.name in ('X','Y','Z')"; |

Небольшая ошибка в тексте запроса JPQL не только останется незамеченной компилятором, но и может привести к неожиданным результатам при его выполнении. Пример:

|  |  |
| --- | --- |
| 1 | "select a from Account a where a.owner.name in (X, Y, Z)"; |

Операторы соединения

В листингах [6](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing6) и [7](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing7) запросы строились из нескольких выражений, но при этом они применялись к одной сущности и ее атрибутам. Однако часто приходится создавать запросы, включающие несколько сущностей, с использованием операторов *соединения*. Для этой цели в CriteriaQuery существуют параметризованные типы *выражений соединения*. Они имеют два типа-параметра: соединяемый тип и связываемый (bindable) тип атрибута, по которому осуществляется соединение. Например, допустим, что необходимо создать запрос для выборки всех объектов типа Customer (клиентов), которым пока не доставили сделанные ими заказы (экземпляры PurchaseOrder). Для этого необходимо соединение сущностей Customer и PurchaseOrder через атрибут Customer.orders, имеющий тип java.util.Set<PurchaseOrder>, как показано в листинге 8.

Листинг 8. Соединение по атрибуту, имеющему тип-коллекцию

|  |  |
| --- | --- |
| 1  2  3 | CriteriaQuery<Customer> q = cb.createQuery(Customer.class);  Root<Customer> c = q.from(Customer.class);  SetJoin<Customer, PurchaseOrder> o = c.join(Customer\_.orders); |

Выражение соединения, сформированное на основе корневого выражения  c и сохраняемого атрибута Customer.orders, имеет два типа-параметра: Customer и связываемый тип атрибута Customer.orders. Обратите внимание, что связываемый тип данного атрибута (PurchaseOrder) *отличается* от объявленного типа (java.util.Set<PurchaseOrder>). Кроме того, заслуживает внимания тот факт, что поскольку связываемый атрибут является множеством (java.util.Set), получившееся в итоге выражение связывания имеет тип SetJoin, являющийся дочерним интерфейсом Join. Подобные специализированные интерфейсы существуют и для других типов множественной связи, а именно CollectionJoin, ListJoin и MapJoin (эти интерфейсы показаны на [рисунке 1](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#fig1)). При этом в третьей строке [листинга 8](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing8) не требуется выполнять преобразование типов, поскольку CriteriaQuery и API Metamodel предоставляют отдельные методы join() для каждого из типов java.util.Collection, List, Set и Map.

Созданные выражения соединения используются в запросах для формирования предикатов, которые вычисляются над связанными сущностями. Например, если требуется выбрать все объекты Customer, для которых существуют недоставленные заказы (PurchaseOrder), можно построить предикат путем перехода от соединенного выраженияo к атрибуту состояния заказа status, сравнения его со значением DELIVERED (доставлен) и выполнения логического отрицания:

|  |  |
| --- | --- |
| 1  2 | Predicate p = cb.equal(o.get(PurchaseOrder\_.status), Status.DELIVERED)          .negate(); |

При работе с выражениями соединения необходимо помнить, что вызов метода join() для некоторого выражения всегда возвращает новое выражение соединения (листинг 9).

Листинг 9. Каждый вызов метода join создает новое выражение

|  |  |
| --- | --- |
| 1  2  3 | SetJoin<Customer, PurchaseOrder> o1 = c.join(Customer\_.orders);  SetJoin<Customer, PurchaseOrder> o2 = c.join(Customer\_.orders);  assert o1 == o2; |

Результат проверки на равенство двух выражений, полученных в результате соединения одного и того же выражения c по одному и тому же атрибуту, будет отрицательным. Таким образом, если требуется создать предикат для выборки всех заказов, которые не были доставлены и сумма которых превышает $200, то следует соединить сущность PurchaseOrder с Customer только один раз, сохранить получившееся выражение связывания в локальной переменной, а затем использовать ее при построении предиката. Данная переменная будет эквивалентна табличной переменной (range variable) в JPQL.

Использование параметров

Вернемся к корректной версии запроса на JPQL, приведенной в начале статьи:

|  |  |
| --- | --- |
| 1 | String jpql = "select p from Person p where p.age > 20"; |

Хотя в запросах часто используются константы, это считается дурным тоном. Более правильный подход заключается в параметризации запросов, благодаря чему текст запросов разбирается и анализируется только один раз, после чего его можно кэшировать и использовать повторно. Следуя этому подходу, показанный выше запрос можно переписать с использованием именованного параметра следующим образом:

|  |  |
| --- | --- |
| 1 | String jpql = "select p from Person p where p.age > :age"; |

Перед выполнением параметризованного запроса выполняется подстановка значений параметров:

|  |  |
| --- | --- |
| 1  2 | Query query = em.createQuery(jpql).setParameter("age", 20);  List result = query.getResultList(); |

В JPQL параметры запросов задаются либо как именованные (например, :age), либо как позиционные (например, ?3), однако в обоих случаях они включаются непосредственно в текст запроса. В отличие от JPQL, в CriteriaQuery параметры выступают в роли отдельных выражений. Как и остальные выражения, они являются строго типизированными и создаются при помощи класса-фабрики QueryBuilder. Например, запрос, приведенный в [листинге 2](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing2), может быть параметризован, как показано в листинге 10.

Листинг 10. Использование параметров в CriteriaQuery

|  |  |
| --- | --- |
| 1  2  3  4  5 | ParameterExpression<Integer> age = qb.parameter(Integer.class);  Predicate condition = qb.gt(p.get(Person\_.age), age);  c.where(condition);  TypedQuery<Person> q = em.createQuery(c);  List<Person> result = q.setParameter(age, 20).getResultList(); |

Необходимо отметить принципиальное различие между параметрами в JPA 2.0 и JPQL: в первом случае выражение-параметр создается на основе явно указанной информации о типе. В примере, приведенном выше, типом является Integer, поэтому число 20 при подстановке расценивается как допустимое значение. Подобное использование информации о типах позволяет минимизировать риск ошибок при выполнении запроса, в частности, запрещая сравнение параметров с выражениями несовместимых типов, а также не допуская подстановку значений недопустимых типов. Подобный контроль типов параметров на этапе компиляции в запросах JPQL невозможен.

В [листинге 10](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing10) показано безымянное выражение-параметр, для которого значение подставляется напрямую. Для выражения можно задать имя, указав его вторым аргументом при создании выражения. После этого подстановку значения в запрос можно будет осуществлять по имени. Использование позиционных параметров невозможно, поскольку, в отличие от JPQL, понятие позиции параметра в запросе не имеет смысла в контексте CriteriaQuery, в котором запрос представляется в виде дерева выражений, а не строки.

Еще одной интересной особенностью параметров в JPA является то, что они не имеют собственного значения. Значения параметров определяются исключительно в контексте исполняемого запроса. Таким образом, можно создать два отдельных исполняемых запроса на основе одного экземпляра CriteriaQuery, подставив в каждый из них свое численное значение одного и того же выражения-параметра.

Проекция результатов

Как уже говорилось, тип результата, возвращаемого при исполнении CriteriaQuery, указывается в момент создания выражения запроса классом QueryBuilder. При этом сами требуемые результаты задаются в виде одного или нескольких *элементов проекции* (projection terms). Для этой цели интерфейс CriteriaQuery предоставляет следующие два метода:

|  |  |
| --- | --- |
| 1  2 | CriteriaQuery<T> select(Selection<? extends T> selection);  CriteriaQuery<T> multiselect(Selection<?>... selections); |

Наиболее простым и часто использующимся элементом проекции является сам класс, указываемый в момент создания запроса. Как показано в листинге 11, этот элемент может задаваться неявным образом.

Листинг 11. По умолчанию запросы CriteriaQuery осуществляют выборку объектов указанного типа

|  |  |
| --- | --- |
| 1  2  3 | CriteriaQuery<Account> q = cb.createQuery(Account.class);  Root<Account> account = q.from(Account.class);  List<Account> accounts = em.createQuery(q).getResultList(); |

В запросе на выборку объектов типа Account, показанном в [листинге 11](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing11) выбираемый элемент не задается явным образом, поэтому в этом качестве выступает сам класс Account. Пример запроса с явным объявлением выбираемого элемента приведен в листинге 12.

Листинг 12. Пример запроса CriteriaQuery с явным указанием выбираемого элемента

|  |  |
| --- | --- |
| 1  2  3  4 | CriteriaQuery<Account> q = cb.createQuery(Account.class);  Root<Account> account = q.from(Account.class);  q.select(account);  List<Account> accounts = em.createQuery(q).getResultList(); |

Проекция результатов запроса может отличаться от сохраняемого класса, указываемого при создании запроса. В этом случае можно использовать специальные конструкции, поддерживаемые интерфейсом QueryBuilder, для точного определения структуры результатов. Примеры этих конструкций приведены в листинге 13.

Листинг 13. Методы, служащие для представления результатов выборки

|  |  |
| --- | --- |
| 1  2  3 | <Y> CompoundSelection<Y> construct(Class<Y> result, Selection<?>... terms);      CompoundSelection<Object[]> array(Selection<?>... terms);      CompoundSelection<Tuple> tuple(Selection<?>... terms); |

Методы, показанные в [листинге 13](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing13), создают сложный элемент проекции, состоящий из выражений, которые могут участвовать в блоках select. Метод construct() создает экземпляр класса, переданного в качестве первого аргумента, передавая его конструктору набор значений, полученных в виде результатов запроса. Например, если несохраняемый класс CustomerDetails содержит конструктор, принимающий аргументы типа String и int, то CriteriaQuery может возвращать набор его экземпляров в качестве результата, создавая их на лету на основе набора объектов сохраняемого класса Customer, выбранных запросом. Пример приведен в листинге 14.

Листинг 14. Представление результатов выборки в виде набора элементов класса при помощи метода construct()

|  |  |
| --- | --- |
| 1  2  3  4 | CriteriaQuery<CustomerDetails> q = cb.createQuery(CustomerDetails.class);  Root<Customer> c = q.from(Customer.class);  q.select(cb.construct(CustomerDetails.class,                c.get(Customer\_.name), c.get(Customer\_.age)); |

Несколько элементов проекции можно объединить в сложный элемент проекции, имеющий тип Object[] или Tuple.. В листинге 15 показан пример представления результатов запроса в виде Object[]:

Листинг 15. Представление результата запроса в виде списка объектов Object[]

|  |  |
| --- | --- |
| 1  2  3  4 | CriteriaQuery<Object[]> q = cb.createQuery(Object[].class);  Root<Customer> c = q.from(Customer.class);  q.select(cb.array(c.get(Customer\_.name), c.get(Customer\_.age));  List<Object[]> result = em.createQuery(q).getResultList(); |

Данный запрос возвращает список объектов-массивов типа Object[], содержащих два элемента каждый. Первым элементом является имя клиента (Customer.name), а вторым – возраст.

Tuple (кортеж) — это специальный интерфейс в JPA, служащий для представления строк базы данных. Объект типа Tuple представляет собой список экземпляров интерфейса TupleElement, который расширяют все интерфейсы выражений, встречающихся в запросах. К элементам кортежа можно обращаться в стиле JDBC (по индексу, начинающемуся с нуля), по псевдониму либо напрямую, через ссылку типа TupleElement. Пример представления результатов запроса в виде кортежей показан в листинге 16.

Листинг 16. Представление результатов выборки в виде кортежей

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CriteriaQuery<Tuple> q = cb.createTupleQuery();  Root<Customer> c = q.from(Customer.class);  TupleElement<String> tname = c.get(Customer\_.name).alias("name");  q.select(cb.tuple(tname, c.get(Customer\_.age).alias("age");  List<Tuple> result = em.createQuery(q).getResultList();  String name = result.get(0).get(name);  String age  = result.get(0).get(1); |

**Ограничения на уровень вложенности**

Теоретически можно определять сложные структуры результатов запросов, используя вложенные элементы проекции. Например, компонентами Tuple могут быть объекты типа Object[] или другие экземпляры Tuple. Однако в спецификации JPA 2.0 подобные иерархические структуры запрещены. Таким образом, выражения, передаваемые в качестве параметров в multiselect(), не могут представлять собой массивы или кортежи. Единственные сложные выражения, которые можно использовать внутри multiselect() - это члены, созданные при помощи метода construct(), которые представляют собой единичные значения объектных типов.

При этом в OpenJPA ограничение на использование вложенных элементов выборки не накладывается.

Данный запрос возвращает список объектов типа Tuple, содержащих по два элемента каждый, к которым можно обращаться либо по индексу, либо по имени TupleElement (если оно было присвоено), либо непосредственно через ссылку на TupleElement. Обратите внимание на вызов метода alias(), который используется для присвоения имен выражениям в запросе (учтите, что вызов этого метода создает копию выражения). Кроме того, в листинге 16 используется метод createTupleQuery() класса QueryBuilder, который эквивалентен вызову createQuery(Tuple.class).

API Criteria включает метод multiselect(), который сочетает в себе поведение описанных выше методов определения структуры результатов, используя информацию о типе выбираемых объектов, указанную при создании экземпляра CriteriaQuery. Данный метод интерпретирует переданные ему элементы, основываясь на типе результата, возвращаемого CriteriaQuery, и преобразует список результатов к нужному виду. Например, чтобы представить результаты запроса в виде списка объектов типа CustomerDetails (см. [листинг 14](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing14)) с использованием multiselect(), следует задать CustomerDetails в качестве типа CriteriaQuery, а затем вызвать multiselect() для списка выбираемых выражений, значения которых должны передаваться конструктору CustomerDetails при выполнении запроса. Пример приведен в листинге 17.

Листинг 17. Интерпретация выражений в методе multiselect() на основе типа результата

|  |  |
| --- | --- |
| 1  2  3 | CriteriaQuery<CustomerDetails> q = cb.createQuery(CustomerDetails.class);  Root<Customer> c = q.from(Customer.class);  q.multiselect(c.get(Customer\_.name), c.get(Customer\_.age)); |

Поскольку типом выбираемых запросом объектов является CustomerDetails, метод multiselect() интерпретирует свои входные параметры как аргументы соответствующего конструктора класса CustomerDetails. Если бы типом запроса был Tuple, то метод бы создавал экземпляры Tuple, используя тот же самый набор параметров. Пример приведен в листинге 18.

Листинг 18. Создание экземпляров Tuple методом multiselect()

|  |  |
| --- | --- |
| 1  2  3 | CriteriaQuery<Tuple> q = cb.createTupleQuery();  Root<Customer> c = q.from(Customer.class);  q.multiselect(c.get(Customer\_.name), c.get(Customer\_.age)); |

Еще интереснее поведение multiselect(), если в качестве типа выбираемых объектов указан Object, либо он не задан вообще. В этом случае результат зависит от числа выражений, переданных в multiselect(). Если такое выражение одно, в качестве результата возвращается оно само, в противном случае результат представляется в виде массива объектов (Object[]).

Дополнительные возможности

До этого момента основное внимание уделялось строгой типизации, лежащей в основе API Criteria и позволяющей минимизировать число синтаксических ошибок, которые могут возникать в строковых запросах на JPQL. Однако Criteria также имеет механизм для программного формирования запросов, который получил название *динамического* API для манипулирования запросами. Его возможности ограничиваются исключительно изобретательностью разработчиков. Ниже будут рассмотрены следующие примеры:

* использование слаботипизированной версии API для создания динамических запросов;
* расширение грамматики с использованием функции базы данных в качестве выражений в запросе;
* редактирование запроса с целью поиска в результатах;
* реализация "запроса по образцу", получившего особую популярность благодаря объектным базам данных.

Слабая типизация и динамические запросы

Поддержка строгой типизации в Criteria базируется на инстанциированных классах метамодели, доступных компилятору. Однако бывают ситуации, когда тип выбираемых объектов может быть определен только во время выполнения. Для решения этой проблемы существуют специальные версии методов Criteria, в которых, как и в Java Reflection API, обращение к сохраняемым атрибутам осуществляется по именам, а не при помощи ссылок на инстанциированные статические метаатрибуты. Эти методы позволяют динамически создавать запросы, жертвуя при этом проверкой типов на этапе компиляции. Пример приведен в листинге 19, в котором показан слаботипизированный вариант запроса из [листинга 6](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing6).

Листинг 19. Пример слаботипизированного запроса

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Class<Account> cls =Class.forName("domain.Account");  Metamodel model = em.getMetamodel();  EntityType<Account> entity = model.entity(cls);  CriteriaQuery<Account> c = cb.createQuery(cls);  Root<Account> account = c.from(entity);  Path<Integer> balance = account.<Integer>get("balance");  c.where(cb.and         (cb.greaterThan(balance, 100),          cb.lessThan(balance), 200))); |

Слаботипизированные методы API не могут возвращать объекты правильных параметризованных типов, поэтому компилятор будет выдавать предупреждения о небезопасных приведениях типов. Один из способов избавления от этих сообщений заключается в применении редко используемой возможности параметризованных типов в Java, а именно – параметризованных вызовов методов. Примером такого вызова может служить метод get(), использующийся для формирования выражения пути в [листинге 19](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing19).

Использование функциональности базы данных в выражениях запросов

Несомненным преимуществом динамического подхода к созданию запросов является возможность расширения грамматики. Например, можно создавать выражения, выполняющие функции базы данных, при помощи метода function() интерфейса QueryBuilder.

|  |  |
| --- | --- |
| 1 | <T> Expression<T> function(String name, Class<T> type, Expression<?>...args); |

Метод function() создает выражение с указанным именем и типом, принимающее на вход произвольное (ноль или более) число аргументов. Это позволяет создавать запросы, в которых вычисляются функции, предоставляемые базой данных. Например, MySQL поддерживает функцию без параметров CURRENT\_USER(), которая возвращает имя пользователя и компьютера для учетной записи, использованной для аутентификации текущего подключения к серверу. Результат данной функции представляет собой строку в кодировке UTF-8. Пример использования CURRENT\_USER() приведен в листинге 20.

Листинг 20. Пример использования функции базы данных в CriteriaQuery

|  |  |
| --- | --- |
| 1  2  3  4  5 | CriteriaQuery<Tuple> q = cb.createTupleQuery();  Root<Customer> c = q.from(Customer.class);  Expression<String> currentUser =      cb.function("CURRENT\_USER", String.class, (Expression<?>[])null);  q.multiselect(currentUser, c.get(Customer\_.balanceOwed)); |

Необходимо отметить, что подобный запрос невозможно написать на JPQL, поскольку этот язык поддерживает только фиксированное число выражений, в то время как API для динамического создания запросов позволяет использовать произвольные выражения.

Редактируемые запросы

Экземпляры CriteriaQuery можно редактировать программным образом. При этом изменения могут касаться всех составляющих запроса, в том числе элементов проекции, предикатов выборки и выражений, задающих порядок сортировки результатов. Эту возможность можно использовать для реализации поиска среди результатов (search-within-result), при котором предикаты поиска последовательно уточняются путем добавления новых условий.

В листинге 21 показан пример создания запроса, упорядочивающего результаты по имени, который затем редактируется с целью добавления сортировки по почтовому индексу.

Листинг 21. Пример редактирования экземпляра CriteriaQuery

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CriteriaQuery<Person> c = cb.createQuery(Person.class);  Root<Person> p = c.from(Person.class);  c.orderBy(cb.asc(p.get(Person\_.name)));  List<Person> result = em.createQuery(c).getResultList();  // start editing  List<Order> orders = c.getOrderList();  List<Order> newOrders = new ArrayList<Order>(orders);  newOrders.add(cb.desc(p.get(Person\_.zipcode)));  c.orderBy(newOrders);  List<Person> result2 = em.createQuery(c).getResultList(); |

Все методы, служащие для задания выражения в запросах ( select(), where() и orderBy() ), очищают предыдущие версии выражений, заменяя их новыми. Кроме того, выражения-списки, возвращаемые такими методами, как getOrderList(), *не связаны* с самим запросом, т. е. добавление и удаление из них элементов никак не затронет объект CriteriaQuery; . Более того, некоторые провайдеры могут даже возвращать константные списки, предотвращая тем самым случайные ошибки. Таким образом, лучше копировать содержимое существующих списков перед тем, как добавлять новые выражения.

Выполнение запроса в памяти при использовании OpenJPA

Возможности OpenJPA позволяют повысить эффективность поиска среди результатов (см. пример в [листинге 21](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing21)) благодаря выполнению модифицированного запроса *в памяти*. Этот подход подразумевает, что результаты отредактированного запроса будут являться строгим подмножеством набора изначально выбранных объектов. OpenJPA предоставляет средства для выполнения запроса над заданным множеством объектов-кандидатов, поэтому в последней строке [листинга 21](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#listing21) достаточно просто передать результаты первоначального запроса.

|  |  |
| --- | --- |
| 1  2  3  4  5 | List<Person> result2 =    em.createQuery(c).    setCandidate    Collection(result).    getResultList(); |

Запросы по образцу

Еще одной привлекательной чертой динамического создания запросов является возможность сравнительно простой реализации принципа написания запросов "*по образцу*". Этот подход (query-by-example) был предложен IBM® Research в 1970 г. и часто приводится в качестве примера ранних попыток упрощения работы с программным обеспечением. Идея этого подхода заключается в том, что от пользователя не требуется указание точных предикатов для поиска данных. Вместо этого ему предлагается выбрать элементы данных, играющие роль шаблона ("образца"). На основе данного образца затем автоматически формируется набор предикатов, каждый из которых представляет собой операцию сравнения с атрибутом шаблона, имеющего непустое значение, отличное от значения по умолчанию. Данный набор предикатов вычисляется в процессе выполнения запроса, в результате чего выбираются все объекты, соответствующие шаблону. "Поиск по образцу" был кандидатом на включение в спецификацию JPA 2.0, однако в итоге в нее не вошел. Тем не менее, этот принцип поддерживается в OpenJPA при помощи расширенного интерфейса OpenJPAQueryBuilder (листинг 22).

Листинг 22. Реализация принципа "запрос по образцу" при помощи расширения интерфейса CriteriaQuery в OpenJPA

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CriteriaQuery<Employee> q = cb.createQuery(Employee.class);    Employee example = new Employee();  example.setSalary(10000);  example.setRating(1);    q.where(cb.qbe(q.from(Employee.class), example); |

Как видно из этого примера, расширение интерфейса QueryBuilder в OpenJPA предоставляет следующий метод:

|  |  |
| --- | --- |
| 1 | public <T> Predicate qbe(From<?, T> from, T template); |

Этот метод создает конъюнкцию предикатов на основе значений атрибутов переданного шаблонного экземпляра (образца). Например, запрос, показанный в листинге 22, выберет всех сотрудников (тип Employee), имеющих зарплату выше 10000 и рейтинг, равный 1. Созданием предикатов можно управлять, указывая список атрибутов, значения которых не должны участвовать в сравнениях, а также задавая способ сравнения строковых атрибутов. Ссылка на Java-документацию по расширениям CriteriaQuery, поддерживаемым в OpenJPA, находится в разделе [Ресурсы](https://www.ibm.com/developerworks/ru/library/j-typesafejpa/index.html#artrelatedtopics).

Заключение

В этой статье был рассмотрен входящий в JPA 2.0 API Criteria, представляющий собой механизм для создания динамических и типобезопасных запросов в Java. Экземпляры CriteriaQuery создаются на этапе выполнения в виде деревьев, содержащих строго типизированные выражения. Использование этих выражений было продемонстрировано на ряде примеров.

В статье была также подчеркнута ключевая роль нового API Metamodel и рассказано о том, как инстанциированные классы метамодели позволяют компилятору контролировать корректность запросов, тем самым снижая риск возникновения ошибок на этапе выполнения, характерных для синтаксически неверных запросов на языке JPQL. Кроме проверки синтаксической корректности, JPA 2.0 предоставляет средства для программного создания запросов, которые открывают такие мощные возможности, как использование функций базы данных, запросы по образцу, а также множество других, которые, я надеюсь, будут найдены читателями этой статьи.