<http://developer.51cto.com/art/200911/162722.htm>

2009-11-13 09:24 Pinaki Poddar IBMDW 字号：**[T](javascript:setfont(12);" \t "_self)** | [**T**](javascript:setfont(16);)

[一键收藏，随时查看，分享好友！](javascript:favorBox('open');)

JPA 2.0引入了 Criteria API，这个 API 首次将类型安全查询引入到 Java 应用程序中，并为在运行时动态地构造查询提供一种机制。本文介绍如何使用 Criteria API 和与之密切相关的 Metamodel API 编写动态的类型安全查询。

AD：[51CTO 网+ 第十二期沙龙：大话数据之美\_如何用数据驱动用户体验](http://mobile.51cto.com/mobile/mdsa12/" \t "_blank)

自从 JPA 于 2006 年首次被引入之后，它就得到了 Java 开发社区的广泛支持。该规范的下一个主要更新 —— 2.0 版本 (JSR 317) —— 将在 2009 年年底完成。JPA 2.0 引入的关键特性之一就是 Criteria API，它为 Java 语言带来了一种独特的能力：开发一种 Java 编译器可以在运行时验证其正确性的查询。Criteria API 还提供一个能够在运行时动态地构建查询的机制。

* [Spring+JPA，下一个人气组合？](http://developer.51cto.com/art/200906/126364.htm)
* [JPA与Hibernate的优缺点](http://developer.51cto.com/art/200906/130500.htm)
* [Java开发三剑客JSF2.0、EJB3.1、JPA2.0现](http://developer.51cto.com/art/200906/130976.htm)
* [JPA是什么和Java EE对象持久化标准浅析](http://developer.51cto.com/art/200906/131699.htm)
* [JPA规范标准及优势浅析](http://developer.51cto.com/art/200906/131700.htm)

本文将介绍 Criteria API 和与之密切相关的 *元模型（metamodel）*概念。您将学习如何使用 Criteria API 开发 Java 编译器能够检查其正确性的查询，从而减少运行时错误，这种查询优于传统的基于字符串的 Java Persistence Query Language (JPQL) 查询。借助使用数据库函数或匹配模板实例的样例查询，我将演示编程式查询构造机制的强大威力，并将其与使用预定义语法的 JPQL 查询进行对比。本文假设您具备基础的 Java 语言编程知识，并了解常见的 JPA 使用，比如 EntityManagerFactory 或 EntityManager。

**JPQL 查询有什么缺陷？**

JPA 1.0 引进了 JPQL，这是一种强大的查询语言，它在很大程度上导致了 JPA 的流行。不过，基于字符串并使用有限语法的 JPQL 存在一些限制。要理解 JPQL 的主要限制之一，请查看清单 1 中的简单代码片段，它通过执行 JPQL 查询选择年龄大于 20 岁的 Person 列表：

**清单 1. 一个简单（并且错误）的 JPQL 查询**

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

这个基础的例子显示了 JPA 1.0 中的查询执行模型的以下关键方面：

* JPQL 查询被指定为一个 String（第 2 行）。
* EntityManager 是构造一个包含给定 JPQL 字符串的*可执行* 查询实例的工厂（第 3 行）。
* 查询执行的结果包含无类型的 java.util.List 的元素。

但是这个简单的例子有一个验证的错误。该代码能够顺利通过编译，但将在运行时*失败*，因为该 JPQL 查询字符串的语法有误。清单 1 的第 2 行的正确语法为：

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

不幸的是，Java 编译器不能发现此类错误。在运行时，该错误将出现在第 3 或第 4 行（具体行数取决于 JPA 提供者是否在查询构造或执行期间根据 JPQL 语法解析 JPQL 字符串）。

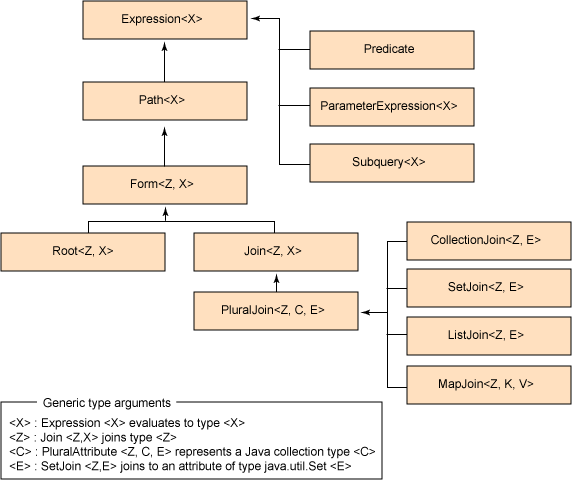
**类型安全查询如何提供帮助？**

Criteria API 的最大优势之一就是禁止构造语法错误的查询。清单 2 使用 CriteriaQuery接口重新编写了 清单 1 中的 JPQL 查询：

**清单 2. 编写 CriteriaQuery 的基本步骤**

|  |
| --- |
| **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(); |

清单 2 展示了 Criteria API 的核心构造及其基本使用：

* 第 1 行通过几种可用方法之一获取一个 EntityManager 实例。
* 在第 2 行，EntityManager 创建 QueryBuilder 的一个实例。QueryBuilder 是 CriteriaQuery 的工厂。
* 在第 3 行，QueryBuilder 工厂构造一个 CriteriaQuery 实例。CriteriaQuery 被赋予泛型类型。泛型参数声明 CriteriaQuery 在执行时返回的结果的类型。在构造 CriteriaQuery 时，您可以提供各种结果类型参数 —— 从持久化实体（比如 Person.class）到形式更加灵活的 Object[]。
* 第 4 行在 CriteriaQuery 实例上设置了*查询表达式*。查询表达式是在一个树中组装的核心单元或节点，用于指定 CriteriaQuery。图 1 显示了在 Criteria API 中定义的查询表达式的层次结构：  
  **图 1. 查询表达式中的接口层次结构**  
  

首先，将 CriteriaQuery 设置为*从* Person.class 查询。结果返回 Root< Person>实例 p。Root 是一个查询表达式，它表示持久化实体的范围。Root< T> 实际上表示：“对所有类型为 T 的实例计算这个查询。” 这类似于 JPQL 或 SQL 查询的 FROM子句。另外还需要注意，Root< Person> 是泛型的（实际上每个表达式都是泛型的）。类型参数就是表达式要计算的值的类型。因此 Root< Person> 表示一个对 Person.class进行计算的表达式。第 5 行构造一个 Predicate。Predicate 是计算结果为 true 或 false 的常见查询表达式形式。谓词由 QueryBuilder 构造，QueryBuilder 不仅是 CriteriaQuery 的工厂，同时也是查询表达式的工厂。QueryBuilder 包含构造传统 JPQL 语法支持的所有查询表达式的 API 方法，并且还包含额外的方法。在 清单 2 中，QueryBuilder 用于构造一个表达式，它将计算第一个表达式参数的值是否大于第二个参数的值。方法签名为：

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

这个方法签名是展示使用强类型语言（比如 Java）定义能够检查正确性并阻止错误的 API 的好例子。该方法签名指定，仅能将值为 Number 的表达式与另一个值也为 Number 的表达式进行比较（例如，不能与值为 String 的表达式进行比较）：

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

第 5 行有更多学问。注意 qb.gt() 方法的第一个输入参数：p.get(Person\_.age)，其中 p 是先前获得的 Root< Person> 表达式。p.get(Person\_.age) 是一个*路径表达式*。路径表达式是通过一个或多个持久化属性从根表达式进行导航得到的结果。因此，表达式 p.get(Person\_.age) 表示使用 Person 的 age 属性从根表达式 p 导航。您可能不明白 Person\_.age 是什么。您可以将其暂时看作一种表示 Person 的 age 属性的方法。我将在谈论 JPA 2.0 引入的新 Metamodel API 时详细解释 Person\_.age。

如前所述，每个查询表达式都是泛型的，以表示表达式计算的值的类型。如果 Person.class 中的 age 属性被声明为类型 Integer （或 int），则表达式 p.get(Person\_.age) 的计算结果的类型为 Integer。由于 API 中的类型安全继承，编辑器本身将对无意义的比较抛出错误，比如：

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

* 第 6 行在 CriteriaQuery 上将谓词设置为其 WHERE 子句。
* 在第 7 行中，EntityManager 创建一个可执行查询，其输入为 CriteriaQuery。这类似于构造一个输入为 JPQL 字符串的可执行查询。但是由于输入 CriteriaQuery 包含更多的类型信息，所以得到的结果是 TypedQuery，它是熟悉的 javax.persistence.Query 的一个扩展。如其名所示，TypedQuery 知道执行它返回的结果的类型。它是这样定义的：

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

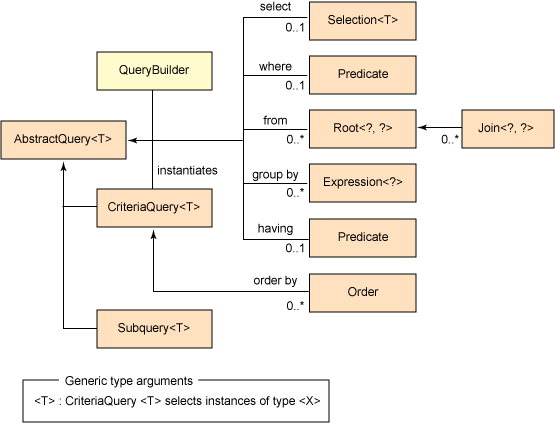
与对应的无类型超接口相反：

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

很明显，TypedQuery 结果具有相同的 Person.class 类型，该类型在构造输入 CriteriaQuery 时由 QueryBuilder 指定（第 3 行）。

* 在第 8 行中，当最终执行查询以获得结果列表时，携带的类型信息展示了其优势。得到的结果是带有类型的 Person 列表，从而使开发人员在遍历生成的元素时省去麻烦的强制类型转换（同时减少了 ClassCastException 运行时错误）。

现在归纳 清单 2 中的简单例子的基本方面：

* CriteriaQuery 是一个查询表达式节点树。在传统的基于字符串的查询语言中，这些表达式节点用于指定查询子句，比如 FROM、WHERE 和 ORDER BY。图 2 显示了与查询相关的子句：   
  **图 2. CriteriaQuery 封装了传统查询的子句**  
  
* 查询表达式被赋予泛型。一些典型的表达式是：
  + Root< T>，相当于一个 FROM 子句。
  + Predicate，其计算为布尔值 true 或 false（事实上，它被声明为 interface Predicate extends Expression< Boolean>）。
  + Path< T>，表示从 Root< *?*> 表达式导航到的持久化属性。Root< T> 是一个没有父类的特殊 Path< T>。
* QueryBuilder 是 CriteriaQuery 和各种查询表达式的工厂。
* CriteriaQuery 被传递给一个可执行查询并保留类型信息，这样可以直接访问选择列表的元素，而不需要任何运行时强制类型转换。