持久对象原生数据库查询语言
设计白皮书

William R. Cook
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-0233, U.S.A.
wcook@cs.utexas.edu

Carl Rosenberger
db4objects Inc.
1900 South Norfolk Street
San Mateo, CA, 94403, U.S.A.
carl@db4o.com

2005 年 8 月 23 日

摘要

大部分 Java 和 .NET 持久架构提供的接口在执行查询时必须以架构特定的查询语言书写。这些接口是基于字符串的：查询语句被定义在字符串中，并通过持久引擎进行解释。基于字符串的查询接口对程序员的生产力有相当大的负面影响。对于像编译时的类型检查、自动对齐、重构，这些开发环境特性，查询语言是不可用的。程序员必须用两种语言开展工作：程序实现语言和数据库查询语言。本文介绍原生数据库查询语言，以简练且类型安全的方式直接使用 Java 和 C# 方法表达查询。探讨了原生数据库查询语言设计并提供了概括性的实现和优化方面的议题。同时，本文也探讨了目前原生数据库查询语言设计的优势和劣势。

1 介绍

当今的对象数据库和对象关系映射（ORM）工具在对象持久化做出了巨大的成就，让开发者能很自然的进行对象持久化，而在面向对象程序中的查询语言看起来有些不协调。这些查询语言用单一的字符串表达，或利用对象视图把分散的字符串组合起来。让我们看一小段例子。

本文中所有例子，我们都使用下面的类：

```
// Java
public class Student {
   private String name;
```

```
    private int age;
    public String getName(){
        return name;
    }
    public int getAge(){
        return age;
    }
}

// C#
public class Student {
    private string name;
    private int age;
    public string Name {
        get { return name; }
    }
    public int Age {
        get{ return age; }
    }
 }
```

怎样利用现有的对象查询语言或 API 找到“年龄小于 20 岁的所有学生”？

OQL [8, 1]

```
String oql =
    "select * from student in AllStudents where student.age < 20";
OQLQuery query = new OQLQuery(oql);
Object students = query.execute();
```

JDOQL [7, 9]

```
Query query =
    persistenceManager.newQuery(Student.class, "age < 20");
Collection students = (Collection)query.execute();
```

db4o SODA，使用 C# [4]

```
Query query = database.Query();
query.Constrain(typeof(Student));
query.Descend("age").Constrain(20).Smaller();
IList students = query.Execute();
```

上面的方法都存在一些普遍问题：

- 现代集成开发环境（IDEs）不会检查内嵌字符串的语义和语法错误。在上面所有查询语句中，age 字段和数值 20 都被认为是数字类型，但是没有一个 IDE 或编译器能检查其实际正确性。如果开发者混淆了查询代码——比如，改变了 age 字段的名字或类型，将导致——上面所有的查询语句在运行时报错，而不会在编译时提示。

- 由于现代 IDEs 不能重构出现在字符串中的字段名，重构行为将导致类模型和查询字符串不同步。假设由于公司采取的编码规则不同 Student 类的字段名 age 变成了 _age。现在，已有 age 查询语句都报错了，我们只好手工修改。

- 现代敏捷开发技术鼓励不断进行重构来维持清晰和与时俱进的类模型，以便准确重现不断演进的域模型。如果查询代码难于维护，它会延迟决定重构的时间并不可避免的引入低质量代码。

- 所有列出的查询操作都依赖 Student 类的私有实现

  student.age

  而不是使用它的公共接口

  student.getAge() / student.Age

  因此他们都破坏了面向对象封装规则，违反接口和实现应该分离的面向对象法则。

- 开发者经常需要在实现语言和查询语言间切换上下文。查询语句无法使用已有的程序实现语言代码。

- 没有明确支持创建可复用的查询组件。一次复杂查询由查询字符串连接而成，但是程序语言的复用特性（方法调用、多态、重载）却没有一样能够用来改善易用性。传递参数到基于字符串的查询显得有些笨拙且容易出错。

- 嵌入的字符串可能受到注入攻击的威胁。

2 设计目标

我们能否用普通 Java 或 C# 语言来表达同样的查询呢？

```
// Java
student.getAge() < 20

// C#
student.Age < 20
```

开发者直接书写查询语句而不必考虑特定的查询语言或 API。IDE 可以及时帮助我们减少书写错误。查询语句将是完全类型安全的而且很容易获取 IDE 的重构特性。查询语句也是原生的、可测试的，运行时依靠内存里的集合类而不是后台数据库。

咋看起来，这种方法似乎不适合作为一种数据库查询机制。由于所有的候选对象都不得不从数据库中实例化，原生执行 Java/C# 代码依赖于完全包含某个类的所有存储对象，这会导致巨大的性能开销。这个问题的解决方案在 Cook 和 Rai[3] 出版的“Safe Query Objects”中：通过翻译成下面的持久化系统查询语言或 API（SQL[6]、OQL[1]、JDOQL[9]、EJBQL[11]、SODA[10]、等等），Java/C# 查询表达式的源代码或字节码能被分析和优化，另外还要利用索引和其他数据库优化手段。本文中，我们精炼了早期关于安全查询对象的理念，更简明和自然的定义了原生查询语言。我们将用 Java 和 .NET 语言的最新优势特性来进行综合查询检测，包括匿名类和委派。

因此，我们的原生查询语言目标是：

100% 的原生 查询语言应能用实现语言（Java 或 C#）完全表达，并完全遵循实现语言的语义。

100% 的面向对象 查询语言应可运行在自己的实现语言中，允许未经优化执行普通集合而不用自定义预处理。

100% 的类型安全 查询语言应能完全获取现代 IDE 的特性，比如语法检测、类型检测、重构，等等。

优化 为实现性能优化，它应该能把原生查询语言翻译成持久体系的查询语言或 API。能在编译期或载入期对源代码或字节码进行分析和翻译。

3 定义原生数据库查询语言 API

原生查询语言是什么样的呢？为了构建最小化设计，通过一次一个的增加设计特性来演进一个简单查询。并将使用 Java 和 C#（.NET 2.0）作为实现语言。

让我们从本文前面设计的类开始。假设想要查询“所有小于 20 岁且名字中包含‘f’字母的学生”。

1. 主要的查询表达式可由编程语言简单表述：

```
// Java
student.getAge() < 20 && student.getName().contains("f")

// C#
student.Age < 20 && student.Name.Contains("f")
```

2. 我们需要有传递 Student 对象到表达式的途径，并返回结果到查询处理器。可以定义 student 参数以及表达式返回一个布尔值达到目的：

```
// 伪 Java 代码
(Student student){
   return student.getAge() < 20
         && student.getName().contains("f");
}

// 伪 C# 代码
(Student student){
   return student.Age < 20
         && student.Name.Contains("f");
}
```

3. 现在我们必须包装上述部分并植入进对象，以便编程语言鉴定其合法性。现在可以传递对象到数据库引擎、集合、或其他查询处理器了。在 .NET 2.0 中，我们只需用委派。在 Java 中，需要一个命名方法，也就是类对象由方法传递。为实现这些要求，我们为方法选择名字，也就是类的名字。沿用这个范例，为 .NET 2.0 设置集合过滤。最后，类名是“Predicate”，方法名是“match”。

```
// Java
new Predicate(){
   public boolean match(Student student){
      return student.getAge() < 20
            && student.getName().contains("f");
   }
}

// C#
delegate(Student student){
   return student.Age < 20
         && student.Name.Contains("f");
}
```

4. 对于 .NET 2.0，我们已做好了最简单的查询接口设计。上面是一个合法对象。对于 Java，查询规范应该是标准的，为查询语句设计抽象基类: Predicate 类。

```
// Java
public abstract class Predicate <ExtentType> {
      public <ExtentType> Predicate (){}
      public abstract boolean match (ExtentType candidate);
}
```

遵循泛型约定，我们还得稍微修改，为 Java 查询对象增加扩展类型。

```
new Predicate <Student> () {
   public boolean match(Student student){
      return student.getAge() < 20
            && student.getName().contains("f");
    }
}
```

5. 尽管上面已完成概念性的东西，我们还是要提供完整的例子来诠释 API。尤其要说明依赖数据库的查询是怎样的，以便和本文中给出的基于字符串的范例做对比。

```
// Java
List <Student> students = database.query <Student> (
   new Predicate <Student> () {
      public boolean match(Student student){
         return student.getAge() < 20
            && student.getName().contains("f");
      }
   });

// C#
IList <Student> students = database.Query <Student> (
   delegate(Student student){
   return student.Age < 20
      && student.Name.Contains("f");
   });
```

上述例子阐述了核心概念。通过 Java 中匿名类和 .NET 中委托的支持，我们精确的反映了 Cook/Rai 关于安全查询[3]的概念。表现出来的是更加简练和直接的查询描述。

API 逐步增加所有的必要部分，可以让我们在 Java 和 C# 中找出最自然高效的表达查询的方式。额外的，像参数化和动态查询，可用相似的手法包含进原生数据库查询语言中。

到这里，我们已克服现有的基于字符串的查询语言的缺点并提高了生产力、健壮性、可维护性和性能。

4 详细规范

只有在实践体验之后，一份最终和详尽的原生数据库查询规范才能成文。因此本章节只在做些臆断。我们应该指出我们所了解到的精华以及原生查询手段所遇到的问题和解决办法。

4.1 优化

单独谈 API，原生查询不是第一个。如果没有优化，我们只能提供“最简单的靠单一方法运行一个类的所有实例并返回布尔值的概念”。著名接口：Smalltalk 80 包含基于 predicate[5, 2] 从集合中选择条目的方法。

优化是原生查询中的关键组建。用户书写原生查询表达式，数据库端以同等性能的基于字符串的查询语句执行，这些我们在前面的介绍中讨论过。

尽管原生查询的核心概念很简单，提供高效解决方案的工作很重要。生成的查询表达式代码必须被分析和转换成等价的数据库查询语言格式。不必在原生查询时转换所有代码。如果优化器不能处理查询表达式中部分或全部代码，这些代码最终会回退到实际对象的实例中，并直接运行查询表达式代码, 或部分代码, 真实对象在查询后回到中间值。由于这一过程可能很慢，它将有助于为开发者提供在开发时提供反馈。这种反馈可能包含优化器怎样“理解”查询表达式，为表达式创建潜在优化计划。这将有助于开发者参照最佳优化语法调整他们的开发风格，让开发者提供关于改善优化的反馈。

优化实际上是怎样工作的呢？在编译或载入期，增强器（一个分离应用程序，或是编译器或载入器的“插件”）将检测所有原生查询表达式的源代码或字节码，并在数据库引擎生成高效格式的附加代码。在运行时，被替代的代码将被替换成之前的查询表达式。当增加优化器来编辑、构建或者两者都是，这种机制对开发者是透明的。

我们的伙伴表示疑问，是否可得到满意的优化。由于原生查询格式和数据库格式都定义好了，因此开发优化器是一个长期的任务，但是我们非常乐观的估计会取得好成绩的。首先 Cook/Rai [3] 构建了 JDO 映射实现，这一点非常令人鼓舞。db4objects 今天[4]已经构建首个未优化的 db4o 原生查询预览产品，并计划在 2005 年 11 月构建原生查询优化产品 5.0 版本。

4.2 约束

理想情况下，任何代码都可以出现在查询表达式中。通过实践，约束是要保证一个稳定的环境，并安装在一个有资源限制的环境中。我们建议：

变量 查询表达式中的变量声明应该是合法的。

对象创建 临时对象本质上是为复杂查询准备的，所以它们的构建应该被查询表达式所支持。

静态调用 静态调用是 OO 语言概念的一部分，所以它们也应是合法的。

少量界面 查询表达式宗旨是快速。它们不应该受 GUI 的影响。

线程 查询表达式将被大量触发。因此不允许创建线程。

安全约束 由于查询表达式实际上可能以服务器上的真实对象执行，因此它们应被约束在一定范围内。它可以允许或禁止某些表空间/包中的方法执行以及对象创建。

只读 不能在运行中的查询代码上修改持久对象。这种限制保证了结果复用以及规范之外的事务性保证。

超时 考虑到对资源使用的限制，数据库引擎会选择让长时间运行的查询超时。超时的配置不是原生查询规范的一部分，但还是推荐实现它。

内存限制 内存限制可设计为超时。一个可配置的内存上限可限制每个查询表达式，推荐实现这一特性。

未定义的行为 如果规范没有明确的限制，那么所有的构造函数都是允许的。

4.3 异常

查询表达式发生任何错误，流程都应该继续执行。抛出未捕获异常的查询表达式应该被处理成只返回 false（查询失败）。同时还要有让开发者能发现、跟踪异常的机制。我们建议最终的实现应该支持异常回调和异常日志机制。

4.4 排序

返回排序对象也应由原生代码定义。给出一个精确的定义超出了本文的范围，但是可以简单列举一下。使用 Java Comparator：

```
// Java
List <Student> students = database.query <Student> (
   new Predicate <Student> () {
      public boolean match(Student student){
         return student.getAge() < 20 && student.getName().contains("f");
      }
});
Collections.sort(students, new Comparator <Student>(){
   public int compare(Student student1, Student student2) {
      return student1.getAge() - student2.getAge();
  }
});
```

上述代码应该可运行在有和没有优化处理器的环境中。数据库服务器上，通过调用数据库引擎的排序函数，查询和排序都可被优化为一步完成。

5 结论

有充分的理由把原生查询语言作为一种主流标准。正如我们讲到的，原生查询语言克服了基于字符串 API 的缺点。只有在实践之后才能完全体会到原生查询语言的潜力。优势如下：

强大 查询语言可利用标准的面向对象编程技术。

生产率 原生数据库查询语言享有先进开发手段的优点，包括，静态类型检测，重构以及自动对齐。

标准 由于编程语言规范已定义好标准，原生数据库查询语言可 100% 保证跨不同数据库实现的兼容性。

效率 原生数据库查询语言可自动编译成传统查询语言或 APIs，以便能支持已有的高性能数据库引擎。

简单 正如前面提到的，原生数据库查询语言 API 只能是一个类一个方法。因此，原生数据库查询语言非常易学，也很容易定义。应该把它们作为 JSR 提交给 JCP(Java Community Process)。

感谢

首先要感谢 Johan Strandler 在 TheServerSide 提交的文章，使得两个作者能一起合作，Patrick Romer 起草了本文的首个草案，Rodrigo B. de Oliveira 贡献了 .NET 委派语法，Klaus Wuestefeld 建议了“原生数据库查询语言”术语，Roberto Zicari、Rick Grehan 以及 Dave Orme 校对了本文草案，以上所有参与者都为本文有争议部分达成共识而努力。

参考文献

[1] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann, January 2000.

[2] W. Cook. Interfaces and specifications for the Smalltalk collection classes. In *OOPSLA*, 1992.

[3] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 97{106. ACM, 2005.

[4] db4objects web site. http://www.db4o.com.

[5] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[6] ISO/IEC. Information technology - database languages - SQL - part 3: Call-level interface (SQL/CLI). Technical Report 9075-3:2003,

ISO/IEC, 2003.

[7] JDO web site. http://java.sun.com/products/jdo.

[8] ODMG web site. http://www.odmg.org.

[9] C. Russell. *Java Data Objects (JDO) Specification JSR-12.* Sun Microsystems, 2003.

[10] SODA - Simple Object Database Access. http://sourceforge.net/projects/sodaquery .

[11] Sun Microsystems. *Enterprise Java Beans Specification, version 2.1*. 2002. http://java.sun.com/j2ee/docs.html.