第 15 章

高级查询选项

本章内容

- 利用Criteria和Example API查询
- 嵌入原生的SQL查询
- 集合过滤器
- 可选的查询结果高速缓存

本章阐述你可能认为是可选的或者高级的所有查询选项。每当你通过编程创建更复杂的查询时，将需要本章的第一个主题：Criteria查询接口。这个API比给HQL和JPA QL通过编程生成的查询字符串更方便且更优雅。不幸的是，它也只可作为一个原生的Hibernate API使用，Java Persistence（还）没有标准化编程式的查询接口。

Hibernate和Java Persistence都支持用原生的SQL编写的查询。你可以在Java源代码中嵌入SQL和存储过程调用，或者把它们外部化到映射元数据。Hibernate可以执行你的SQL，并把结果集转变为更方便的对象，这取决于你的映射。

集合的过滤是Hibernate的一项简单便利特性——你不会经常用它。例如，如果想要在一个集合中获得对象的一个子集，它就会帮助你用一个简单的API调用和查询片段，取代更繁杂的查询。

最后，将讨论可选的查询结果高速缓存——我们已经说过，它并不是在所有的情况下都有用，因此将更深入地讨论一下高速缓存查询结果的好处，以及什么时候完美地启用这项特性。

从按条件查询和按示例查询开始。

15.1 利用条件和示例查询

Criteria和Example API都只可在Hibernate中使用；Java Persistence没有标准化这些接口。如前所述，似乎其他的供应商（不仅仅Hibernate）也支持一个类似的扩展接口，并且标准的未来版本将包括这项功能。

当查询变得更加复杂时，带有通过编程生成的条件和示例对象的查询经常是首选的解决方案。如果你必须在运行时创建查询，这尤其正确。想象你必须在应用程序中实现一个搜索掩码，利用用户可以启用的许多复选框、输入字段和开关。必须从用户的选择中创建一个数据库查询。

完成这项工作的传统方法是通过串联创建一个查询字符串，或者可能编写一个查询构建器（query builder），它可以为你构建SQL查询字符串。如果试图在这个场景中使用HQL或者JPA QL，也会遇到同样的问题。

Criteria和Example接口允许通过以正确的顺序创建和组合对象来编程式地构建查询。现在介绍如何利用这些API，以及如何表达选择、限制、联结和投影。假设你已经阅读了前一章，并且知道这些操作如何转变成SQL。即使你决定使用Criteria和Example API作为编写查询的主要方法，也要记住，由于HQL和JPA QL本来就是基于字符串的，因此始终更加灵活。

从一些基本的选择和限制示例开始吧。

15.1.1 基本的条件查询

最简单的条件查询看起来像这样：

```
session.createCriteria(Item.class);
```

它获取Item类的所有持久化实例。这也称作条件查询的根实体（root entity）。

条件查询也支持多态：

```
session.createCriteria(BillingDetails.class);
```

这个查询返回BillingDetails及其子类的实例。同样地，下列条件查询返回所有持久化对象：

```
session.createCriteria(java.lang.Object.class);
```

Criteria接口也支持通过addOrder()方法和Order条件给结果排序：

```
session.createCriteria(User.class)
        .addOrder( Order.asc("lastname") )
        .addOrder( Order.asc("firstname") );
```

你不必用开着的Session创建条件对象，DetachedCriteria可以被实例化，之后附加到Session用于执行（或者附加到另一个Criteria作为子查询）：

```
DetachedCriteria crit =
 DetachedCriteria.forClass(User.class)
             .addOrder( Order.asc("lastname") )
             .addOrder( Order.asc("firstname") );

List result = crit.getExecutableCriteria(session).list();
```

通常，你想要限制结果，并且不获取类的所有实例。

1. 应用限制

对于条件查询来说，你必须构建Criterion对象来表达约束。Restrictions类给内建的Criterion类型提供工厂方法。我们来用特定的电子邮件地址搜索User对象：

```
Criterion emailEq = Restrictions.eq("email", "foo@hibernate.org");
Criteria crit = session.createCriteria(User.class);
crit.add(emailEq);
User user = (User) crit.uniqueResult();
```

你创建了Criterion，表示对一个同一性比较的限制，并把它添加到Criteria。这个eq()方法有两个参数：第一个是属性名，然后是应该被比较的值。属性名始终作为一个字符串给定，

记住，这个名称在领域模型重构期间可以改变，并且你必须手工更新任何预设的条件查询。还要注意，条件接口不支持显式的参数绑定，因为没有必要。在前一个例子中，你绑定了字符串"foo@hibernate.org"到查询中。你可以绑定任何java.lang.Object，并让Hibernate确定如何处理它。uniqueResult()方法执行查询，并正好返回一个对象作为结果——你必须正确地转换（cast）它。

利用方法链编写这个通常不会太冗长：

```
User user =
  (User) session.createCriteria(User.class)
                .add(Restrictions.eq("email", "foo@hibernate.org"))
                .uniqueResult();
```

很显然，如果条件查询变得更加复杂时会更难以阅读——这是首选它们用于动态和编程查询生成的一个很好的理由，而给预设的查询使用外部化的HQL和JPA QL。JDK 5.0的一项新特性是静态导入（static import），它有助于使条件查询更易于阅读。例如，通过添加：

```
import static org.hibernate.criterion.Restrictions.*;
```

可以将这个条件查询限制代码缩写为：

```
User user =
  (User) session.createCriteria(User.class)
                .add( eq("email", "foo@hibernate.org") )
                .uniqueResult();
```

获得Criterion的另一种可选方法是Property对象——本节稍后讨论投影时，它将会更有用：

```
session.createCriteria(User.class)
        .add( Property.forName("email").eq("foo@hibernate.org") );
```

也可以用一般的点号命名一个组件的属性：

```
session.createCriteria(User.class)
        .add( Restrictions.eq("homeAddress.street", "Foo"));
```

Criteria API和org.hibernate.criterion包除了提供eq()之外，还提供许多其他的操作符，可以用来构造更复杂的表达式。

2. 创建比较表达式

所有一般的SQL（和HQL、JPA QL）比较操作符也可以通过Restrictions类使用：

```
Criterion restriction =
 Restrictions.between("amount",
                new BigDecimal(100),
                new BigDecimal(200) );
session.createCriteria(Bid.class).add(restriction);

session.createCriteria(Bid.class)
        .add( Restrictions.gt("amount", new BigDecimal(100) ) );

String[] emails = { "foo@hibernate.org", "bar@hibernate.org" };
session.createCriteria(User.class)
        .add( Restrictions.in("email", emails) );
```

三重逻辑操作符也可用；这个查询返回没有电子邮件地址的所有用户：

```
session.createCriteria(User.class)
        .add( Restrictions.isNull("email") );
```

还需要能够找到的确有电子邮件地址的用户：

```
session.createCriteria(User.class)
        .add( Restrictions.isNotNull("email") );
```

也可以用isEmpty()、isNotEmpty()或者它的实际大小测试集合：

```
session.createCriteria(Item.class)
        .add( Restrictions.isEmpty("bids"));

session.createCriteria(Item.class)
        .add( Restrictions.sizeGt("bids", 3));
```

或者可以比较两个属性：

```
session.createCriteria(User.class)
        .add( Restrictions.eqProperty("firstname", "username") );
```

条件查询接口对字符串匹配也有特殊的支持。

3. 字符串匹配

对于条件查询，通配符搜索可以使用与HQL和JAP QL相同的通配符符号（%和_），或者指定MatchMode。MatchMode是不用字符串操作来表达子字符串匹配的一种方便的方法。这两个查询是一样的：

```
session.createCriteria(User.class)
        .add( Restrictions.like("username", "G%") );

session.createCriteria(User.class)
        .add( Restrictions.like("username", "G", MatchMode.START) );
```

得到支持的MatchMode是START、END、ANYWHERE和EXACT。

你还经常想要执行不区分大小写的字符串匹配。求助于函数的时候，比如HQL或者JPA QL中的LOWRE()，就可以依赖Criteria API的一种方法：

```
session.createCriteria(User.class)
        .add( Restrictions.eq("username", "foo").ignoreCase() );
```

可以把表达式与逻辑操作符组合起来。

4. 组合表达式和逻辑操作符

如果把多个Criterion实例添加到同一个Criteria实例，它们被联结起来应用（利用and）：

```
session.createCriteria(User.class)
        .add( Restrictions.like("firstname", "G%") )
        .add( Restrictions.like("lastname", "K%") );
```

如果需要分开（or），有两种方法。第一种是把Restrictions.or()和Restrictions.and()一起使用：

```
session.createCriteria(User.class)
        .add(
            Restrictions.or(
                Restrictions.and(
```

```
                    Restrictions.like("firstname", "G%"),
                    Restrictions.like("lastname", "K%")
                ),
                Restrictions.in("email", emails)
            )
        );
```

第二种方法是把Restrictions.disjunction()和Restrictions.conjunction()一起使用：

```
session.createCriteria(User.class)
        .add( Restrictions.disjunction()
            .add( Restrictions.conjunction()
                .add( Restrictions.like("firstname", "G%") )
                .add( Restrictions.like("lastname", "K%") )
            )
            .add( Restrictions.in("email", emails) )
        );
```

我们认为这两种方法都很难懂，即使花上5分钟努力给它们调整格式以达到最大限度的可读性。JDK 5.0静态导入可以帮助明显改善可读性，但是即使如此，除非你正在凭空构建查询，否则HQL或者JPA QL字符串更容易理解。

你可能已经注意到许多标准的比较操作符（<、>、=等）都内建在Criteria API中，但是漏了某些操作符。例如，任何算术操作符（如加和减）都没有直接得到支持。

另一个问题是函数调用。Criteria只给最常见的案例（如字符串区分大小写）匹配内建函数。另一方面，HQL允许你在WHERE子句中调用任意的SQL函数。

Criteria API有着类似的工具：可以添加任意的SQL表达式作为Criterion。

5. 添加任意的SQL表达式

假设你想要测试一个字符串的长度，并相应地限制查询结果。Criteria API没有相当于SQL、HQL或者JPA QL中的LENGTH()函数。

但是，可以把一个简单的SQL函数表达式添加到你的Criteria：

```
session.createCriteria(User.class)
    .add( Restrictions.sqlRestriction(
            "length({alias}.PASSWORD) < ?",
            5,
            Hibernate.INTEGER
           )
    );
```

这个查询返回密码少于5个字符的所有User对象。最终的SQL中任何表别名都要加上{alias}占位符作为前缀；它总是指向根实体所映射到的表（这个例子里是USERS）。你还用了一个位置参数（这个API不支持具名参数），并指定它的类型为Hibernate.INTEGER。不用单个的绑定参数和类型，而是还可以使用支持参数和类型数组的sqlRestriction()方法的重载版本。

这个工具很强大——例如，可以用量词添加一个SQL WHERE子句子查询：

```
session.createCriteria(Item.class)
    .add( Restrictions.sqlRestriction(
            "'100' > all" +
            " ( select b.AMOUNT from BID b" +
```

```
        "    where b.ITEM_ID = {alias}.ITEM_ID )"
      )
);
```

这个查询返回出价不大于100的所有Item对象。

（Hibernate条件查询系统是可扩展的：也可以把LENGTH() SQL函数包装在自己的Criterion接口的实现中。）

最后，可以编写包括子查询的条件查询。

6. 编写子查询

条件查询中的子查询是一个WHERE子句查询。就像在HQL、JPA QL和SQL中一样，子查询的结果可能包含单行或者多行。通常情况下，返回单行的子查询执行统计。

下列子查询返回某位用户出售的货品总量；外部查询返回出售货品超过10件的所有用户：

```
DetachedCriteria subquery =
                    DetachedCriteria.forClass(Item.class, "i");

subquery.add( Restrictions.eqProperty("i.seller.id", "u.id"))
        .add( Restrictions.isNotNull("i.successfulBid") )
        .setProjection( Property.forName("i.id").count() );

Criteria criteria = session.createCriteria(User.class, "u")
                           .add( Subqueries.lt(10, subquery) );
```

这是一个相关子查询。DetachedCriteria指向别名u；这个别名在外部查询中声明。注意，外部查询使用了小于操作符，因为子查询是右操作数。还要注意i.seller.id没有生成联结，因为SELLER_ID是ITEM表中的一个列，它是脱管条件的根实体。

让我们进入下一个有关条件查询的主题：联结和动态抓取。

15.1.2 联结和动态抓取

就像在HQL和JPA QL中一样，对于你为什么要表达联结可能着不同的理由。首先，你可能想要用一个联结通过被联结类的某些属性来限制结果。例如，你可能想要获取某位特定的用户出售的所有Item实例。

当然，你也可能想要用联结来动态抓取被关联的对象或者集合，就像在HQL和JPA QL中使用fetch关键字一样。在条件查询中，也可以通过FetchMode使用相同的选项。

先来看看普通联结，以及如何表达涉及被关联类的限制。

1. 给限制联结关联

在Criteria API中表达联结有两种方法；因此你有两种给限制使用别名的方法。第一种是Criteria接口的createCriteria()方法。这个方法一般意味着你可以嵌套对createCriteria()的调用：

```
Criteria itemCriteria = session.createCriteria(Item.class);
itemCriteria.add(
    Restrictions.like("description",
                      "Foo",
                      MatchMode.ANYWHERE)
);
```

```
Criteria bidCriteria = itemCriteria.createCriteria("bids");
bidCriteria.add( Restrictions.gt( "amount", new BigDecimal(99) ) );

List result = itemCriteria.list();
```

通常编写查询如下（方法链）：

```
List result =
  session.createCriteria(Item.class)
   .add( Restrictions.like("description",
                           "Foo",
                           MatchMode.ANYWHERE)
   )
   .createCriteria("bids")
    .add( Restrictions.gt("amount", new BigDecimal(99) ) )
   .list();
```

给Item的bids创建Criteria，导致了两个类的表之间的内部联结。注意，你可以在任何一个Criteria实例中调用list()而不改变查询结果。嵌套条件不仅适用于集合（如bids），还可用于单值的关联（如seller）：

```
List result =
  session.createCriteria(Item.class)
   .createCriteria("seller")
    .add( Restrictions.like("email", "%@hibernate.org") )
   .list();
```

这个查询返回具有特定电子邮件地址模式的用户出售的所有货品。

用Criteria API表达内部联结的第二种方法是为被联结的实体分配一个别名：

```
session.createCriteria(Item.class)
        .createAlias("bids", "b")
        .add( Restrictions.like("description", "%Foo%") )
        .add( Restrictions.gt("b.amount", new BigDecimal(99) ) );
```

对于单值关联seller中的限制也一样：

```
session.createCriteria(Item.class)
        .createAlias("seller", "s")
        .add( Restrictions.like("s.email", "%hibernate.org" ) );
```

这种方法不用Criteria的第二实例；它一般与你在HQL/JPA QL语句的FROM子句中编写的别名分配机制一样。然后，被联结实体的属性必须通过在createAlias()方法中分配的别名来限定，如s.email。条件查询（Item）的根实体的属性可以被引用，而不用完全限定的别名，或者利用别名"this"：

```
session.createCriteria(Item.class)
        .createAlias("bids", "b")
        .add( Restrictions.like("this.description", "%Foo%") )
        .add( Restrictions.gt("b.amount", new BigDecimal(99) ) );
```

最后注意，在编写本书之时，只有包含了对实体的引用的被关联实体或者集合的联结（一对多和多对多），在Hibernate中通过Criteria API得到支持。下列示例试图联结组件的一个集合：

```
session.createCriteria(Item.class)
        .createAlias("images", "img")
        .add( Restrictions.gt("img.sizeX", 320 ) );
```

Hibernate失败，并有一个异常告诉你：你想要使用别名的那个属性不表示一个实体关联。我们认为在你读到本书时，这项特性可能已经实现了。

另一种语法也是无效的，但你可能有兴趣一试，它是包含点号的单值关联的一个隐式联结：

```
session.createCriteria(Item.class)
        .add( Restrictions.like("seller.email", "%hibernate.org") );
```

“seller.email”字符串不是一个属性或者组件的属性路径。创建一个别名或者嵌套的Criteria对象来联结这个实体关联。

我们来讨论被关联对象和集合的动态抓取。

2. 通过条件查询动态抓取

在HQL和JPA QL中，你用join fetch操作即时填入一个集合，或者初始化一个被映射为延迟否则将被代理的对象。可以用Criteria API完成同样的事：

```
session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .add( Restrictions.like("description", "%Foo%") );
```

这个查询返回所有包含特定集合的Item实例，并给每个Item即时加载bids集合。

FetchMode.JOIN通过SQL外部联结启用即时抓取。如果想要使用内部联结（很少用，因为它不返回没有出价的货品），可以强制它：

```
session.createCriteria(Item.class)
        .createAlias("bids", "b", CriteriaSpecification.INNER_JOIN)
        .setFetchMode("b", FetchMode.JOIN)
        .add( Restrictions.like("description", "%Foo%") );
```

也可以预抓取多对一和一对一关联：

```
session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .setFetchMode("seller", FetchMode.JOIN)
        .add( Restrictions.like("description", "%Foo%") );
```

但是小心。HQL和JPA QL中的警告在这里也同样适用：并行即时抓取不止一个集合（如bids和images），会造成一个SQL笛卡儿积，它或许比两个单独的查询更慢。如果你给集合使用即时抓取，那么给限制分页结果集也在内存中完成。

然而，通过Criteria和FetchMode的动态抓取与HQL和JPA QL中的稍有不同：Criteria查询不会忽略在映射元数据中定义的全局抓取策略。例如，如果利用fetch="join"或者FetchType.EAGER映射bids集合，下列查询就会造成ITEM和BID表的一个外部联结：

```
session.createCriteria(Item.class)
        .add( Restrictions.like("description", "%Foo%") );
```

被返回的Item实例初始化了它们的bids集合，并完全加载。这种情形不会发生在HQL或者JPA QL中，除非你用LEFT JOIN FETCH进行手工查询（或者，当然要映射集合为lazy="false"，这样造成第二个SQL查询）。

因而，条件查询可能返回对根实体的独特实例的重复引用，即使你没有在查询中给集合应用FetchMode.JOIN。最后一个查询实例可能返回几百个Item引用，即使你在数据库中只有十几个。

记住我们在14.3.1节中的讨论，并再看一下图14-3中的SQL语句和结果集。

可以在结果List中移除重复的引用，通过把它包装在LinkedHashSet中（常规的HashSet不保存查询结果的顺序）。在HQL和JPA QL中，也可以使用DISTINCT关键字；然而，在Criteria中，没有它的直接等价物。这就是ResultTransformer变得有用的地方。

3．应用结果转换器

结果转换器可以被应用到查询结果，以便你可以用自己的过程而不是Hibernate的默认行为过滤或者封送结果。Hibernate的默认行为是你可以替换和（或）定制的一组默认转换器。

默认情况下，所有的条件查询都只返回根实体的实例：

```
List result = session.createCriteria(Item.class)
                     .setFetchMode("bids", FetchMode.JOIN)
                     .setResultTransformer(Criteria.ROOT_ENTITY)
                     .list();

Set distinctResult = new LinkedHashSet(result);
```

Criteria.ROOT_ENTITY是org.hibernate.transform.ResultTransformer接口的默认实现。前一个查询无论是否使用这个转换器设置都生成相同的结果。它返回所有Item实例并初始化它们的bids集合。List可能（取决于每个Item的Bid数量）包含重复的Item引用。

另一种方法是，可以应用一个不同的转换器：

```
List distinctResult =
    session.createCriteria(Item.class)
           .setFetchMode("bids", FetchMode.JOIN)
           .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)
           .list();
```

现在Hibernate在返回结果之前过滤掉重复的根实体引用——如果使用DISTINCT关键字，这实际上与在HQL或者JPA QL中发生的过滤是一样的。

如果想要在联结查询中获取有别名的实体，结果转换器也很有用：

```
Criteria crit =
        session.createCriteria(Item.class)
               .createAlias("bids", "b")
               .createAlias("seller", "s")
               .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);

List result = crit.list();
for (Object aResult : result) {
    Map map = (Map) aResult;
    Item item   = (Item) map.get(Criteria.ROOT_ALIAS);
    Bid  bid    = (Bid)  map.get("b");
    User seller = (User) map.get("s");
    ...
}
```

首先，创建条件查询，它用Item的bids和seller关联来联结Item。这是一个跨三张表的SQL内部联结。在SQL中，这个查询的结果是一张表，在这里每一个结果行都包含货品、出价和用户数据——几乎与图14-2中所示的一样。通过默认的转换器，Hibernate只返回Item实例。并且它通过DISTINCT_ROOT_ENTITY转换器，过滤掉重复的Item引用。似乎没有任何明显的选项——你真

正想要的是在一个映射中返回所有的信息。ALIAS_TO_ENTITY_MAP转换器可以把SQL结果封送到Map实例的一个集合。每个Map都有3个条目：Item、Bid和User。所有的结果数据都得到保存，并且可以在应用程序中被访问。(Criteria.ROOT_ALIAS是"this"的快捷方式。)

这最后一种转换器的好用例很少。注意，也可以实现自己的org.hibernate.transform.ResultTransformer。而且，HQL和原生的SQL查询也支持ResultTransformer：

```
Query q = session.createQuery(
    "select i.id as itemId," +
    "       i.description as desc," +
    "       i.initialPrice as price from Item i");
q.setResultTransformer( Transformers.aliasToBean(ItemDTO.class) );
```

这个查询现在返回ItemDTO实例的一个集合，并且这个bean的属性也通过setItemId()、setDesc()和setPrice()等设置方法被设值。

定义要从查询中返回什么数据的一种更常用的方法是投影。Hibernate条件支持SELECT子句的等价物，用于简单的投影、统计和分组。

15.1.3 投影和报表查询

在HQL、JPA QL和SQL中，编写SELECT子句给一个特定的查询定义投影。Criteria API也支持投影，当然是通过编程而不是基于字符串。可以选择在查询结果中具体需要哪些对象或者对象的哪些属性，以及可能想要如何对报表进行统计和分组结果。

1. 简单的投影列表

下列条件查询只返回仍然处在拍卖中的Item实例的标识符值：

```
session.createCriteria(Item.class)
        .add( Restrictions.gt("endDate", new Date()) )
        .setProjection( Projections.id() );
```

Criteria中的setProjection()方法，接受任何单个的被投影的属性（如在前一个示例中一样），或者接受要包括在结果中的几个属性的一个列表：

```
session.createCriteria(Item.class)
        .setProjection( Projections.projectionList()
            .add( Projections.id() )
            .add( Projections.property("description") )
            .add( Projections.property("initialPrice") )
        );
```

这个查询返回Object[]的一个List，就像HQL或者JPA QL会有一个相当的SELECT子句一样。给投影指定属性的另一种可选的方法是Property类：

```
session.createCriteria(Item.class)
        .setProjection( Projections.projectionList()
            .add( Property.forName("id") )
            .add( Property.forName("description") )
            .add( Property.forName("initialPrice") )
        );
```

在HQL和JPA QL中，可以通过SELECT NEW操作使用动态的实例化，并返回定制对象（而不是Object[]）的一个集合。Hibernate给条件查询捆绑了ResultTransformer，它可以完成几乎相同的工作（事实上，它更加灵活）。下列查询返回与前一个示例一样的结果，但被包在数据迁移对象中：

```
session.createCriteria(Item.class)
        .setProjection( Projections.projectionList()
            .add( Projections.id()
                        .as("itemId") )
    .add( Projections.property("description")
                        .as("itemDescription") )
    .add( Projections.property("initialPrice")
                        .as("itemInitialPrice") )
).setResultTransformer(
    new AliasToBeanResultTransformer(ItemPriceSummary.class)
);
```

ItemPriceSummary是一个简单的Java bean，包含设置方法或者具名itemId、itemDescription和itemInitialPrice的公共字段。它不一定是被映射的持久化类；只有属性/字段名称必须与分配到条件查询中被投影属性的别名相匹配。别名通过as()方法进行分配（可以把它当作SQL SELECT中AS关键字的等价物）。结果转换器调用设置方法或者直接填充字段，并返回ItemPriceSummary对象的一个集合。

我们来用条件进行更复杂的投影，涉及统计和分组。

2. 统计和分组

一般的统计函数和分组选项在条件查询中也可以用。有一种简单的方法统计结果中的行数：

```
session.createCriteria(Item.class)
        .setProjection( Projections.rowCount() );
```

提示　获取分页统计总数——在实际的应用程序中，你经常必须允许用户通过列表进行分页，并同时通知他们列表中总共有多少件货品。获得总数的一种方法是执行rowCount()的Criteria查询。你不用编写这个额外的查询，而是执行与利用scroll()给列表获取数据相同的Criteria。然后调用last()和getRowNumber()来跳到并得到最后一行的行数。这个数字加上一就是你所列的对象数量了。别忘了关闭光标。如果你正在使用现有的DetachedCriteria对象，并且不想重复和操作它的投影来执行rowCount()，这种方法尤其有用。它也适用于HQL或者SQL查询。

更加复杂的统计则使用统计函数。下列查询查找出价的数量，并且平均每位用户所进行的出价总金额：

```
session.createCriteria(Bid.class)
        .createAlias("bidder", "u")
        .setProjection( Projections.projectionList()
```

```
        .add( Property.forName("u.id").group() )
        .add( Property.forName("u.username").group() )
        .add( Property.forName("id").count())
        .add( Property.forName("amount").avg() )
    );
```

这个查询返回包含四个字段的Object[]的一个集合：用户id、登录名、出价数以及平均出价金额。记住，你可以再次给动态的实例化使用结果转换器，并返回数据转换器对象，而不是Object[]。产生同样结果的另一种可选方法如下：

```
session.createCriteria(Bid.class)
        .createAlias("bidder", "u")
        .setProjection( Projections.projectionList()
          .add( Projections.groupProperty("u.id") )
          .add( Projections.groupProperty("u.username") )
          .add( Projections.count("id") )
          .add( Projections.avg("amount") )
        );
```

喜欢哪种语法只是个人的偏好问题。更复杂的示例则把别名应用到被统计和被分组的属性，用于给结果排序：

```
session.createCriteria(Bid.class)
        .createAlias("bidder", "u")
        .setProjection( Projections.projectionList()
          .add( Projections.groupProperty("u.id") )
          .add( Projections.groupProperty("u.username").as("uname") )
          .add( Projections.count("id") )
          .add( Projections.avg("amount") )
        )
        .addOrder( Order.asc("uname") );
```

在编写本书之时，对HAVING的支持和对被统计结果的限制，在Hibernate条件查询中还不可用。这或许会增加到未来的版本中。

可以把原生的SQL表达式添加到条件查询中的限制，同样的特性也可用于投影。

3. 利用SQL投影

SQL投影是被添加到生成的SQL SELECT子句的一个任意片段。下列查询产生统计和分组，就像在前面的示例中一样，但是还添加了一个额外的值到结果中（货品的数量）：

```
String sqlFragment =
    "(select count(*) from ITEM i where i.ITEM_ID = ITEM_ID)" +
    " as numOfItems";

session.createCriteria(Bid.class)
        .createAlias("bidder", "u")
        .setProjection( Projections.projectionList()
            .add( Projections.groupProperty("u.id") )
            .add( Projections.groupProperty("u.username") )
            .add( Projections.count("id") )
            .add( Projections.avg("amount) )
            .add( Projections.sqlProjection(
                    sqlFragment,
                    new String[] { "numOfItems" },
```

```
                    new Type[] { Hibernate.LONG }
                )
            )
        );
```

生成的SQL如下：

```
select
        u.USER_ID,
        u.USERNAME,
        count(BID_ID),
        avg(BID_AMOUNT),
        (select
            count(*)
        from
            ITEM i
        where
            i.ITEM_ID = ITEM_ID) as numOfItems
    from
        BID
    inner join
        USERS u
            on BIDDER_ID = u.USER_ID
    group by
        u.USER_ID,
        u.USERNAME
```

这个SQL片段被嵌入到SELECT子句中。它可以包含数据库管理系统支持的任意表达式和函数调用。任何未限定列名（如ITEM_ID）都指向条件根实体（BID）的表。必须告诉Hibernate返回的SQL投影的别名numOfItems，以及它的Hibernate值映射类型Hibernate.LONG。

Criteria API的真正威力在于把任意的Criterion与示例对象组合的可能。这项特性就是大家所知的按示例查询。

15.1.4　按示例查询

通过（根据用户输入组合几个可选的条件），编程式地构建条件查询还是很常见的。例如，系统管理员可能希望按名或者姓的任何组合搜索用户，并获取按用户名排序的结果。

利用HQL或者JPA QL，你可以用字符串操作构建查询：

```
public List findUsers(String firstname,
                      String lastname) {

    StringBuffer queryString = new StringBuffer();
    boolean conditionFound = false;

    if (firstname != null) {
        queryString.append("lower(u.firstname) like :firstname ");
        conditionFound=true;
    }
    if (lastname != null) {
        if (conditionFound) queryString.append("and ");
        queryString.append("lower(u.lastname) like :lastname ");
        conditionFound=true;
```

```
    }

    String fromClause = conditionFound ?
                            "from User u where " :
                            "from User u ";

    queryString.insert(0, fromClause).append("order by u.username");

    Query query = getSession()
                      .createQuery( queryString.toString() );

    if (firstname != null)
        query.setString( "firstName",
                         '%' + firstname.toLowerCase() + '%' );
    if (lastname != null)
        query.setString( "lastName",
                         '%' + lastname.toLowerCase() + '%' );

    return query.list();
}
```

这段代码十分冗长乏味，因此我们尝试一种不同的方法。利用你目前已经学过的Criteria API看起来很有希望进行简化：

```
public List findUsers(String firstname,
                      String lastname) {

    Criteria crit = getSession().createCriteria(User.class);

    if (firstname != null) {
        crit.add( Restrictions.ilike("firstname",
                                     firstname,
                                     MatchMode.ANYWHERE) );
    }
    if (lastname != null) {
        crit.add( Restrictions.ilike("lastname",
                                     lastname,
                                     MatchMode.ANYWHERE) );
    }

    crit.addOrder( Order.asc("username") );

    return crit.list();
}
```

这段代码更短一些。注意ilike()操作符执行不区分大小写的匹配。似乎毫无疑问这就是比较好的方法了。然而，对于利用许多可选的搜索条件的搜索屏幕来说，还有一种更好的方法。

随着你增加新的搜索条件，findUsers()的参数清单也在增加，捕捉可搜索的属性作为对象将会更好。因为所有的搜索属性都属于User类，为什么不把User的实例用于这一用途？

按示例查询（QBE）依赖这一思想。你通过一些被初始化的属性提供被查询类的实例，并且查询返回包含匹配的属性值的所有持久化实例。Hibernate实现QBE作为Criteria查询API的一部分：

```
public List findUsersByExample(User u) throws {

    Example exampleUser =
      Example.create(u)
```

```
                .ignoreCase()
                .enableLike(MatchMode.ANYWHERE)
                .excludeProperty("password");

        return getSession().createCriteria(User.class)
                           .add(exampleUser)
                           .list();
    }
```

对create()的调用给User的指定实例返回Example的一个新实例。ignoreCase()方法为所有字符串值属性把实例查询放进一个不区分大小写的模式中。对enableLike()的调用指定SQL Like操作符应该用于所有的字符串值属性，并指定MatchMode。最后，可以利用excludeProperty()从搜索中排除特定的属性。默认情况下，所有的值类型属性，除了标识符属性之外，都用在比较式中。

你已经进一步明显地简化了这段代码。关于Hibernate Example查询最好的方面是，Example只是一个一般的Criterion。可以随意混合和匹配按实例查询和按条件查询。

通过进一步限制搜索结果为包含未售Items的用户，看一下这是如何进行的。因此可以把Criteria添加到示例用户中，利用它的Items的items集合来约束结果：

```
public List findUsersByExample(User u){

    Example exampleUser =
      Example.create(u)
              .ignoreCase()
              .enableLike(MatchMode.ANYWHERE);

    return getSession().createCriteria(User.class)
              .add( exampleUser )
              .createCriteria("items")
                  .add( Restrictions.isNull("successfulBid") )
              .list();
}
```

更好的是，可以在同一个搜索中组合User属性和Item属性：

```
public List findUsersByExample(User u, Item i) {

    Example exampleUser =
      Example.create(u).ignoreCase().enableLike(MatchMode.ANYWHERE);

    Example exampleItem =
      Example.create(i).ignoreCase().enableLike(MatchMode.ANYWHERE);

    return getSession().createCriteria(User.class)
              .add( exampleUser )
              .createCriteria("items")
                  .add( exampleItem )
              .list();
}
```

现在，请你退回一步，并考虑利用手工编码的SQL/JDBC实现这个搜索屏幕需要多少代码。我们不想把它复制到这里，它会增加好多页。还要注意findUsersByExample()方法的客户端不需要知道关于Hibernate的任何事，它仍然可以给搜索创建复杂的条件。

如果HQL、JPA QL甚至Criteria和Example都不够强大，难以表达特定查询，你就必须退回到原生的SQL。

15.2 利用原生的 SQL 查询

HQL、JPA QL或者条件查询，都应该足够灵活，可以执行你喜欢的几乎任何查询。它们指向被映射的对象模式，因此，如果你的映射结果不出所料，Hibernate的查询就应该提供你喜欢的任何获取数据的功能。有几种例外。如果你想包括一个原生的SQL提示，来指示数据库管理系统查询优化器（例如，你需要自己编写SQL）。HQL、JPA QL和条件查询都没有用于这个的关键字。

另一方面，你不退回到手工的SQL查询，而是始终努力扩展内建的查询机制，并给特殊的操作包括支持。这用HQL和JPA QL更难以实现，因为你必须修改这些基于字符串语言的语法。扩展Criteria API和添加新方法或者新Criterion类很容易。查看一下org.hibernate.criterion包中的Hibernate源代码，它精心设计且制成了文档。

当无法扩展内建的查询工具或者防止不可移植的手工编写的SQL时，你应该首先考虑使用Hibernate原生的SQL查询选项，就是我们现在正在介绍的。记住，你总是可以退回到简单的JDBC Connection，并自己准备任何SQL语句。Hibernate的SQL选项允许你在Hibernate API中嵌入SQL语句，并让你从能让生活变得更轻松的额外服务中受益。

最重要的是，Hibernate可以处理你SQL查询的结果集。

15.2.1 自动的结果集处理

利用Hibernate API执行SQL语句的最大好处在于，自动把表格式的结果集封送到业务对象中去。下列SQL查询返回Category对象的一个集合：

```
List result = session.createSQLQuery("select * from CATEGORY")
                     .addEntity(Category.class)
                     .list();
```

Hibernate读取SQL查询的结果集，并尝试发现你的映射元数据中所定义的列名和类型。如果返回列CATEGORY_NAME，就把它映射到Category类的name属性，Hibernate就知道如何填充该属性，并最后返回完全加载的业务对象。

SQL查询中的*投影结果集中所有选中的列。因此自动的发现机制只适用于繁琐的查询，更加复杂的查询则需要显式的投影。下一个查询返回Item对象的一个集合：

```
session.createSQLQuery("select {i.*} from ITEM i" +
                       " join USERS u on i.SELLER_ID = u.USER_ID" +
                       " where u.USERNAME = :uname")
       .addEntity("i", Item.class)
       .setParameter("uname", "johndoe");
```

SQL SELECT子句包括一个占位符，它命名表别名为i，并把这张表的所有列投影到结果中。任何其他的表别名（例如被联结的USERS表，它只与限制相关）不包括在结果集中。现在用addEntity()告诉Hibernate，别名i的占位符是指填充Item实体类所需要的所有列。列名和类型再次由Hibernate在查询执行和结果封送期间自动推测。

甚至可以在原生的SQL查询中即时抓取被关联的对象和集合：

```
session.createSQLQuery("select {i.*}, {u.*} from ITEM i" +
                       " join USERS u on i.SELLER_ID = u.USER_ID" +
                       " where u.USERNAME = :uname")
        .addEntity("i", Item.class)
        .addJoin("u", "i.seller")
        .setParameter("uname", "johndoe");
```

这个SQL查询从两个表别名中投影两组列，并且使用两个占位符。占位符i仍然是指填充了由这个查询返回的Item实体对象的列。addJoin()方法告诉Hibernate，别名u是指可以用来立即填充每个Item被关联seller的列。

自动把结果集封送到业务对象，这并不是Hibernate中原生的SQL查询特性的唯一好处。即使你想获取的只是一个简单的标量值，也可以使用它。

15.2.2　获取标量值

标量值可以是任何Hibernate值类型。最常见的是字符串、数字或者时间戳。下列SQL查询返回货品数据：

```
List result = session.createSQLQuery("select * from ITEM").list();
```

这个查询的result是Object[]的一个List，实际上是一张表。每个数组中的每个字段都是标量类型即字符串、数字或者时间戳。除了包装在一个Object[]中之外，结果与类似的简单JDBC查询结果完全一样。这显然没有太大的用处，但是Hibernate API的其中一个好处在于它抛出unchecked exception，因此你不必把查询包装在try/catch块中，就像如果你调用JDBC API时所必须做的那样。

如果没有用*投影每件东西，就需要告诉Hibernate你想要从结果中返回哪些标量值：

```
session.createSQLQuery("select u.FIRSTNAME as fname from USERS u")
        .addScalar("fname");
```

addScalar()方法告诉Hibernate，你的fname SQL别名应该作为标量值返回，并且类型应该被自动推测。查询返回字符串的一个集合。这种自动的类型发现在大多数案例中都运行得很好，但是你有时可能想要显式地指定类型——例如，想要用UserType转换值的时候：

```
Properties params = new Properties();
params.put("enumClassname", "auction.model.Rating");

session.createSQLQuery(
        "select c.RATING as rating from COMMENTS c" +
        " where c.FROM_USER_ID = :uid"
    )
    .addScalar("rating",
               Hibernate.custom(StringEnumUserType.class, params) )
    .setParameter("uid", new Long(123));
```

首先，看一下SQL查询。它选择COMMENTS表的RATING列，并限制由特定的用户所做出的评价结果。假设数据库中的这个字段包含字符串值（例如EXCELLENT、OK或者BAD）。因此SQL查询的结果是字符串值。

你一般不会把这个映射为Java中简单的字符串，而是使用枚举，或者用定制的Hibernate UserType。5.3.7节这么做过，并创建过一个可以把SQL数据库中的字符串转变为Java中任何枚举实例的StringEnumUserType。它必须用enumClassname参数化，你想要它转换值到——这个例子中的auction.model.Rating。通过在查询中用addScalar()方法设置预设的定制类型，把它启用为处理结果的转换器，这样你取回了Rating对象的集合，而不是简单的字符串。

最后，可以在同一个原生的SQL查询中，混合标量结果和实体对象：

```
session.createSQLQuery(
        "select {i.*}, u.FIRSTNAME as fname from ITEM i" +
        " join USERS u on i.SELLER_ID = u.USER_ID" +
        " where u.USERNAME = :uname"
    )
    .addEntity("i", Item.class)
    .addScalar("fname")
    .setParameter("uname", "johndoe");
```

这个查询的结果仍然是Object[]的一个集合。每一个数组有两个字段：Item实例和字符串。

你或许会认同原生的SQL查询比HQL或者JPA QL语句更难以阅读，并且把它们隔离和外部化到映射元数据中似乎会更好一些。8.2.2节对存储过程查询这么做过。这里不再重复，因为存储过程查询和简单的SQL查询之间的唯一区别只在于调用的语法或者语句——封送和结果集映射选项是一样的。

Java Persistence标准化了JPA QL，并允许退回到原生的SQL。

15.2.3 Java Persistence 中的原生 SQL

Java Persistence通过在EntityManager中用createNativeQuery()方法支持原生的SQL查询。原生的SQL查询可以返回实体实例、标量值或者两者的混合。然而，不同于Hibernate，Java Persistence中的API利用映射元数据来定义结果集处理。我们来看一些实例。

简单的SQL查询不需要显式的结果集映射：

```
em.createNativeQuery("select * from CATEGORY", Category.class);
```

结果集被自动封送到Category实例的一个集合。注意，持久化引擎希望创建Category实例所需的所有列都通过查询返回，包括所有的属性、组件和外键列——否则就会抛出异常。列在结果集中按名称搜索。你可能必须在SQL中使用别名，返回与定义在实体映射元数据中相同的列名。

如果你的原生SQL查询返回多个实体类型或者标量类型，就需要应用显式的结果集映射。例如，返回Object[]集合的查询，在这里每一个数组索引0中都是Item实例，索引1中是User实例，可以写成下面这样：

```
em.createNativeQuery("select " +
    "i.ITEM_ID, i.ITEM_PRICE, u.USERNAME, u.EMAIL " +
    "from ITEM i join USERS u where i.SELLER_ID = u.USER_ID",
    "ItemSellerResult");
```

最后一个参数ItemSellerResult，是定义在元数据中的一个结果映射的名称（在类或者全局的JPA XML级别中）：

```
@SqlResultSetMappings({
    @SqlResultSetMapping(
        name = "ItemSellerResult",
        entities = {
            @EntityResult(entityClass = auction.model.Item.class),
            @EntityResult(entityClass = auction.model.User.class)
        }
    )
})
```

这个结果集映射对于我们介绍过的查询很可能不起作用——给自动映射记住这一点，实例化Item和User对象所需的所有列都必须在SQL查询中被返回。返回的4个列不可能只表示持久化的属性。为了便于举例，假设它们的确只表示持久化的属性，而你真正的问题在于结果集中的列名，它们与被映射列的名称不匹配。首先，把别名添加到SQL语句：

```
em.createNativeQuery("select " +
    "i.ITEM_ID as ITEM_ID, i.ITEM_PRICE as ITEM_PRICE, " +
    "u.USERNAME as USER_NAME, u.EMAIL as USER_EMAIL " +
    "from ITEM i join USERS u on i.SELLER_ID = u.USER_ID",
    "ItemSellerResult");
```

接下来，在结果集映射中使用@FieldResult来把别名映射到实体实例的字段：

```
@SqlResultSetMapping(
name = "ItemSellerResult",
entities = {
    @EntityResult(
        entityClass = auction.model.Item.class,
        fields = {
          @FieldResult(name = "id", column = "ITEM_ID"),
          @FieldResult(name = "initialPrice", column = "ITEM_PRICE")
        }),
    @EntityResult(
        entityClass = auction.model.User.class,
        fields = {
          @FieldResult(name = "username", column = "USER_NAME"),
          @FieldResult(name = "email", column = "USER_EMAIL")
        })
})
```

也可以返回标量类型结果。下列查询返回拍卖货品标识符和每件货品的出价数量：

```
em.createNativeQuery("select " +
    "i.ITEM_ID as ITEM_ID, count(b.*) as NUM_OF_BIDS " +
    "from ITEM i join BIDS b on i.ITEM_ID = b.ITEM_ID " +
    "group by ITEM_ID",
    "ItemBidResult");
```

这一次，结果集映射没有包含实体结果映射，而只有列：

```
@SqlResultSetMapping(
name = "ItemBidResult",
columns = {
    @ColumnResult(name = "ITEM_ID"),
    @ColumnResult(name = "NUM_OF_BIDS")
})
```

这个查询的结果是Object[]的一个集合，有两个字段，两者都是数字类型（最可能是long）。如果想要把实体和标量类型混合成为一个查询结果，就在@SqlResultSetMapping中组合entities和columns属性。

最后，注意JPA规范并不要求原生的SQL查询支持具名参数绑定。Hibernate支持它。

接下来讨论另一项更为怪异但更方便的Hibernate特性（Java Persistence没有它的等价物）：集合过滤器（collection filter）。

15.3 过滤集合

你可能希望针对一个集合的所有元素执行一个查询。例如，你可能有一个Item，并希望按出价创建的时间获取对该特定货品的所有出价。为此可以映射一个排序的或者有序的集合，但是有一种更容易的选择。可以编写查询，你应该已经知道要如何进行了：

```
session.createQuery("from Bid b where b.item = :givenItem" +
                    " order by b.created asc")
       .setEntity("givenItem", item);
```

这个查询有效，因为出价和货品之间的关联是双向的，每一个Bid都知道它的Item。这个查询中没有任何联结，b.item指向BID表中的ITEM_ID列，因此你直接给比较设置了值。想象这个关联是单向的：Item有Bids的一个集合，但是没有从Bid到Item的反向关联。可以试试下列查询：

```
select b from Item i join i.bids b
    where i = :givenItem order by b.amount asc
```

这个查询的效率很低，它用了一个完全没有必要的联结。一种更好更优雅的解决方案是使用集合过滤器——可以应用到持久化集合（或者数组）的一种特殊查询。它常用于进一步限制或者排序结果。在一个已经加载的Item和它的出价集合中应用它：

```
List filteredCollection =
        session.createFilter( item.getBids(),
                              "order by this.created asc" ).list();
```

这个过滤器相当于本节的第一个查询，并生成相同的SQL。Session中的createFilter()方法用了两个参数：一个持久化的集合（不一定要被初始化）和一个HQL查询字符串。集合过滤器查询有一个隐式的FROM子句和一个隐式的WHERE条件。别名this暗指出价集合的元素。

Hibernate集合过滤器不是在内存中执行的。出价的集合可能在调用过滤器时没有被初始化，如果是这样，就保持未初始化。此外，不要把过滤器应用到瞬时的集合或者查询结果。它们只能应用到当前被添加到Hibernate持久化上下文的一个实体实例引用的持久化集合。术语“过滤器”有点误导，因为过滤的结果是一个全新的不同集合，并没有接触到原始的集合。

HQL查询唯一需要的子句是FROM子句。因为集合过滤器有一个隐式的FROM子句，下面是一个有效的过滤器：

```
List filteredCollection =
        session.createFilter( item.getBids(), "" ).list();
```

让大家（包括这项特性的设计者）惊喜的是，这个小小的过滤器很有用。可以用它给集合元素进行分页：

```
List filteredCollection =
    session.createFilter( item.getBids(), "" )
            .setFirstResult(50)
            .setMaxResults(100)
            .list();
```

但是，你通常会给分页查询使用ORDER BY。

即使在集合过滤器中不需要FROM子句，如果喜欢，你也可以放一个。集合过滤器甚至不需要返回正被过滤的集合的元素。下一个查询返回任何与指定集合中的一个类别同名的Category：

```
String filterString =
  "select other from Category other where this.name = other.name";

List result =
    session.createFilter( cat.getChildCategories(), filterString )
            .list();
```

下列查询返回对该货品有出价的Users的一个集合：

```
List result =
    session.createFilter( item.getBids(),
                          "select this.bidder" )
             .list();
```

下一个查询返回所有这些用户的出价（包括对其他货品的出价）：

```
List result =
    session.createFilter(
        item.getBids(),
        "select elements(this.bidder.bids)"
    ).list();
```

注意这个查询用了特殊的HQL elements()函数来投影一个集合的所有元素。

所有这些都很有趣，但是集合过滤器存在的最重要原因在于，允许应用程序获取集合的一些元素，而不用初始化整个集合。对于大集合而言，实现满意的性能很重要。下列查询获取一位用户在过去一周所做的所有出价：

```
List result =
    session.createFilter( user.getBids(),
                          "where this.created > :oneWeekAgo" )
            .setTimestamp("oneWeekAgo", oneWeekAgo)
            .list();
```

这段代码还是没有初始化User的bids集合。

无论用哪种语言或者哪个API编写的查询，都始终应该在你决定用可选的查询高速缓存给它们提速之前，把它们调优成能够按预期执行。

15.4　高速缓存查询结果

13.3节中已经讨论过二级高速缓存和Hibernate的一般高速缓存架构。你知道二级高速缓存是一个共享的数据高速缓存，每当你访问一个未加载的代理或者集合时，或者当你按标识符加载对象时（从二级高速缓存的观点来看，这些全部都是标识符查找），Hibernate就在这个高速缓存中通过查找尝试解析数据。另一方面，查询结果在默认情况下没有被高速缓存。

有些查询仍然使用二级高速缓存，这取决于你如何执行查询。例如，如果决定用iterate()执行查询（如前一章所述），只从数据库中获取实体的主键，并通过一级高速缓存查找实体数据，如果对特定的实体启用二级高速缓存，也通过二级高速缓存查找。我们还断定这个选项只有当二级高速缓存被启用时才有意义，因为列读取的优化一般不影响性能。

高速缓存查询结果是一个完全不同的问题。查询结果高速缓存在默认情况下是禁用的，并且每一个HQL、JPA QL、SQL和Criteria查询都始终先命中数据库。我们首先介绍如何启用查询结果高速缓存，以及它是如何工作的。然后讨论它为什么被禁用，以及为什么很少有查询从结果高速缓存中受益。

15.4.1 启用查询结果高速缓存

查询高速缓存必须用Hibernate配置属性来启用：

```
hibernate.cache.use_query_cache = true
```

然而，单独这个设置对于Hibernate高速缓存查询结果还不够。默认情况下，所有的查询都始终忽略高速缓存。为了给特定的查询启用查询高速缓存（允许它的结果被添加到高速缓存，并允许它从高速缓存中提取结果），那么使用org.hibernate.Query接口。

```
Query categoryByName =
    session.createQuery("from Category c where c.name = :name");
categoryByName.setString("name", categoryName);
categoryByName.setCacheable(true);
```

setCachable()方法启用了结果高速缓存。它在Criteria API上也可用。如果想给javax.persistence.Query启用结果高速缓存，就使用setHint("org.hibernate.cacheable", true)。

15.4.2 理解查询高速缓存

当某个查询被第一次执行时，它的结果被高速缓存在高速缓存区域——这个区域不同于你可能已经配置的任何其他实体或者集合高速缓存区域。这个区域的名称默认为org.hibernate.cache.QueryCache。

可以通过setCacheRegion()方法，给一个特定查询改变高速缓存区域：

```
Query categoryByName =
    session.createQuery("from Category c where c.name = :name");
categoryByName.setString("name", categoryName);
categoryByName.setCacheable(true);
categoryByName.setCacheRegion("my.Region");
```

这几乎没有必要，你只在需要一个不同的区域配置时，才会给某些查询使用不同的高速缓存区域——例如，在一个更细粒度的级别上限制查询高速缓存的内存消耗。

标准的查询结果高速缓存区域保存着SQL语句（包括所有绑定的参数）和每个SQL语句的结果集。但是，这不是完整的SQL结果集。如果结果集包含实体实例（前面的示例查询返回Category实例），那么只有标识符值被保存在结果集高速缓存中。当结果集被放进高速缓存区域

时，每个实体的数据列就被它丢弃了。因此，命中查询结果高速缓存意味着，对于前面的查询，Hibernate将找到一些Category标识符值。

高速缓存实体的状态，是二级高速缓存区域auction.model.Category（结合持久化上下文）的责任。这类似于iterate()的查询策略，如前所述。换句话说，如果你查询实体，并决定启用高速缓存，就要确保你也给这些实体启用了常规的二级高速缓存。如果没有，可能在启用查询高速缓存之后，以更多的数据库命中而告终。

如果高速缓存了不返回实体实例而只返回相同标量值（如货品名称和价格）的查询结果，这些值就被直接保存在查询结果高速缓存中。

如果查询结果高速缓存在Hibernate中被启用，另一个始终需要的高速缓存区域也出现了：org.hibernate.cache.UpdateTimestampsCache。这是Hibernate内部使用的一个高速缓存区域。

Hibernate用时间戳区域来决定被高速缓存的查询结果集是否为失效的。当你重新执行一个启用了高速缓存的查询时，Hibernate就在时间戳高速缓存中查找对被查询的（几张）表所做的最近插入、更新、或者删除的时间戳。如果找到的时间戳晚于高速缓存查询结果的时间戳，高速缓存结果就被丢弃，并产生一个新查询。这就有效地保证了：如果查询可能涉及的任何表中包含被更新的数据，Hibernate就不会使用被高速缓存的查询结果；因此高速缓存结果就可能是失效的。为了得到最佳结果，你应该配置时间戳区域，以便表的更新时间戳没有超出高速缓存期，而来自这些表的查询结果仍然被高速缓存在一个其他区域中。最容易的方法是在你二级高速缓存提供程序的配置中，关闭时间戳高速缓存区域的过期期限。

15.4.3　什么时候使用查询高速缓存

大多数查询都没有从结果高速缓存中受益。这可能有点出乎意料。毕竟，避免数据库命中听起来似乎总是件好事。比起对象导航或者按标识符获取而言，它之所以并非永远对任意的查询有用，有以下两个原因。

首先，必须问问，你会每隔多久重复执行同一个查询。毫无疑问，在你的应用程序中可能有很多已反复执行的查询（把完全相同的实参绑定到参数），以及自动生成相同的SQL语句。这种情况很罕见，但是当你确定某个查询被重复执行时，它就变成了结果高速缓存的一个很好的备选对象了。

其次，对于执行许多查询和少数插入、删除或者更新的应用程序来说，高速缓存查询可以提升性能和可伸缩性。另一方面，如果应用程序执行许多写入操作，查询高速缓存就没有得到有效的利用。当出现在被高速缓存的查询结果中的表的任何行有任何插入、更新或者删除时，Hibernate就会废弃掉被高速缓存的查询结果集。这意味着被高速缓存结果的寿命可能很短，即使重复执行查询，由于相同数据（相同表）的并发修改，也没有可以使用的高速缓存结果。

对于许多查询来说，查询结果高速缓存的好处是不存在的，至少没有达到所期待的影响程度。但是有一种特殊的查询可以从结果高速缓存中大大受益。

15.4.4 自然标识符高速缓存查找

假设你有一个包含自然键的实体。我们不讨论自然主键，而是讨论应用到你实体的单个或者复合属性的业务键（business key）。例如，如果用户的登录名称不可变，那么它可以是唯一的业务键。这是我们已经隔离作为好的equals()对象等同性子程序实现的最好的键。可以在9.2.3节中找到这种键的示例。

通常要把构成自然键的属性映射为Hibernate中的常规属性。可以在数据库级启用一个unique约束来表示这个键。例如，如果考虑User类，可能要用username和emailAddress构成实体的业务键：

```
<class name="User">
    <id name="id".../>

    <property name="username" unique-key="UNQ_USERKEY"/>
    <property name="emailAddress" unique-key="UNQ_USERKEY"/>
    ...

</class>
```

这个映射在跨越两列的数据库级上启用了一个唯一键约束。也假设业务键属性是不可变的。虽然这不太可能，因为你可能允许用户更新他们的电子邮件地址，但我们现在介绍的这项功能则只有当你处理不可变的业务键时才有意义。映射不可变性如下：

```
<class name="User">
    <id name="id".../>

    <property name="username"
              unique-key="UNQ_USERKEY"
              update="false"/>

    <property name="emailAddress"
              unique-key="UNQ_USERKEY"
              update="false"/>
    ...

</class>
```

或者，通过业务键利用高速缓存查找，可以用<natural-id>映射它：

```
<class name="User">
    <id name="id".../>

    <cache usage="read-write"/>

    <natural-id mutable="false">
        <property name="username"/>
        <property name="emailAddress"/>
    </natural-id>
    ...

</class>
```

这个分组自动启用跨越所有被组合属性的唯一键SQL约束的生成。如果mutable属性被设置成false，它也防止被映射列的更新。现在可以给高速缓存查找使用这个业务键：

```
Criteria crit = session.createCriteria(User.class);

crit.add( Restrictions.naturalId()
            .set("username", "johndoe")
            .set("emailAddress", "jd@hibernate.org")
        );
crit.setCacheable(true);

User result = (User) crit.uniqueResult();
```

这个条件查询根据业务键找到一个特定的用户对象。它通过业务键导致一个二级高速缓存查找——记住，这通常是一个通过主键的查找，并且可能只用于按主标识符的获取。业务键映射和Criteria API允许你通过业务键表达这个特殊的二级高速缓存查找。

在编写本书之时，还没有用于自然标识符映射的Hibernate扩展注解可用，并且HQL也不支持与此相当的关键字，用于通过业务键的查找。

从我们的观点来看，二级高速缓存是一项重要的特性，但它不是优化性能时的首选方案。查询设计中的错误，或者对象模型中没有必要的复杂部分，都无法通过一个“全部高速缓存”的方法得到改进。如果应用程序只用一个热高速缓存（即几个小时或者几天的运行时之后的完整高速缓存）以可接受的等级执行，那么就应该查找出严重的设计错误、损害性能的查询和n+1查询问题。在决定启用此处所述的任何一个查询高速缓存选项时，首先要按照13.2.5节中介绍的指导方针，检查和调优一下你的应用程序。

15.5　小结

在本章中，你已经通过Hibernate Criteria和Example API，有步骤地生成了查询。我们也讨论了嵌入的和外部化的SQL查询，以及如何自动地把SQL查询的结果集映射到更方便的业务对象。Java Persistence也支持原生的SQL，并标准化如何映射外部化的SQL查询的结果集。

最后，我们涵盖了查询结果高速缓存，并讨论了它为什么只在某些情况下有用。

表15-1展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表15-1　第15章中Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate支持一个强大的Criteria和Example API，用于有步骤的查询生成	即将面世的标准版本中将有一些QBC和QBE API
Hibernate通过结果集的自动封送，具有灵活的映射选项用于嵌入式和外部化的SQL查询	Java Persistence标准化SQL嵌入和映射，并支持结果集封送
Hibernate支持集合过滤器API	Java Persistence没有标准化集合过滤器API
Hibernate可以高速缓存查询结果	特定于Hibernate的查询提示可以用来高速缓存查询结果

第16章将把所有的知识点放在一起，并关注利用Hibernate、Java Persistence和EJB 3.0组件的应用程序的设计和架构。我们也将对Hibernate应用程序进行单元测试。