6.1.2.2 有连接表的单向 1－1 关联

虽然这种情况很少见，但 Hibernate 同样允许这样采用连接表映射单向 1－1 关联。有连接表的 1－1 关联同样需要显式使用@JoinTable 映射连接表。除此之外，由于此处的@JoinTable 映射的连接表维护的是 1－1 关联，因此程序需要为@JoinTable 中 joinColumns 属性映射的外键列增加 unique=true，也为 inverseJoinColumns 属性映射的外键列增加 unique=true。

下面 Person 类使用了@OneToOne 和@JoinTable 修饰代表关联实体的 Address 类型的属性。

程序清单：codes\06\6.1\unidirectional\1-1withjointable\src\ org\crazyit\app\domain\Person.java

```
@Entity
@Table(name="person_inf")
public class Person
{
    // 标识属性
    @Id @Column(name="person_id")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private int age;
    // 定义该 Person 实体关联的 Address 实体
    @OneToOne(targetEntity=Address.class)
    @JoinTable(name="person_address",
        joinColumns=@JoinColumn(name="person_id"
            , referencedColumnName="person_id" , unique=true),
        inverseJoinColumns=@JoinColumn(name="address_id"
            , referencedColumnName="address_id", unique=true)
    )
    private Address address;
    // 省略所有的 getter、setter 方法
    ...
}
```

上面粗体字注解使用@JoinTable 注解时为连接表中的两个外键列都增加了 unique=true，这就保证了一个 Address 实体最多只能关联一个 Person 实体。

6.1.7.2 有连接表的双向 1－1 关联

采用连接表的双向 1－1 关联是相当罕见的情形，映射相当复杂，数据模型烦琐。通常不推荐使用这种策略。

有连接表的双向 1－1 关联需要在两端分别使用@OneToOne 修饰代表关联实体的属性，并在一端都使用@JoinTable 显式映射连接表，由于此时是 1－1 关联，因此使用@JoinTable 注解时指定的@JoinColumn 都需要增加 unique=true；另一端的@OneToOne 注解上指定 mappedBy 属性，表明这一段不再管理关联关系。

下面示例让 Person 实体类与 Address 类保留双向 1－1 关联关系，并让 Person 一端不再管理关联关系，因此 Person 一端的@OneToOne 需要指定 mappedBy 属性。

下面是双向 1－1 关联的 Person 实体类的源代码，程序使用了带 mappedBy 属性的@OneToOne 修饰代表关联实体的属性。

程序清单：codes\06\6.1\bidirectional\1-1jointable\src\org\crazyit\app\domain\Person.java

```
@Entity
@Table(name="person_inf")
public class Person
{
    // 标识属性
    @Id @Column(name="person_id")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private int age;
    // 定义该 Person 实体关联的 Address 实体
```

```
    @OneToOne(targetEntity=Address.class, mappedBy="person")
    private Address address;
    // 省略所有的 setter、getter 方法
    ...
}
```

下面是双向 1－1 关联的 Address 实体类的源代码，程序通常使用@OneToOne、@JoinTable 修饰代表关联实体的属性。

程序清单：codes\06\6.1\bidirectional\1-1jointable\src\org\crazyit\app\domain\Address.java

```
@Entity
@Table(name="address_inf")
public class Address
{
    // 标识属性
    @Id @Column(name="address_id")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int addressId;
    // 定义地址详细信息的成员变量
    private String addressDetail;
    // 定义该 Address 实体关联的 Person 实体
    @OneToOne(targetEntity=Person.class)
    // 映射底层连接表，表名为 person_address
    @JoinTable(name="person_address",
        // 映射连接表的外键列，增加 unique=true 表明是 1-1 关联
        joinColumns=@JoinColumn(name="address_id"
            , referencedColumnName="address_id", unique=true),
        // 映射连接表的外键列，增加 unique=true 表明是 1-1 关联
        inverseJoinColumns=@JoinColumn(name="person_id"
            , referencedColumnName="person_id" , unique=true)
    )
    private Person person;
    // 无参数的构造器
    public Address()
    {
    }
    // 初始化全部成员变量的构造器
    public Address(String addressDetail)
    {
        this.addressDetail = addressDetail;
    }
    // 省略所有的 setter、getter 方法
    ...
}
```

从上面两个关联实体的粗体字注解来看，Person 一端的@OneToOne 指定了 mappedBy 属性，因此这一端的关联实体不能指定@JoinTable，这一端也不控制关联关系。而 Address 一端则显式使用@JoinTable 映射了连接表，因此这一端可控制关联关系。

6.5 条件查询

条件查询是更具面向对象特色的数据查询方式。条件查询通过如下三个类完成。

- Criteria：代表一次查询。
- Criterion：代表一个查询条件。
- Restrictions：产生查询条件的工具类。

执行条件查询的步骤如下。

1. 获得 Hibernate 的 Session 对象。
2. 以 Session 对象创建 Criteria 对象。
3. 使用 Restrictions 的静态方法创建 Criterion 查询条件。
4. 向 Criteria 查询中添加 Criterion 查询条件。

5 执行 Criteria 的 list()或 uniqueResult()方法返回结果集。

看下面的条件查询的示例程序。

程序清单：codes\06\6.5\criteria\src\lee\CriteriaTest.java

```
public class CriteriaTest
{
    public static void main(String[] args)
    {
        CriteriaTest criteriaTest = new CriteriaTest();
        criteriaTest.query();
        HibernateUtil.sessionFactory.close();
    }
    private void query()
    {
        // 打开 Session 和事务
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
        // 使用 createCriteria 开始条件查询
        List list = session.createCriteria(Student.class)
            // 根据 Student 的属性进行过滤数据
            .add( Restrictions.gt("name" , "a"))
            .list();
        System.out.println("=====简单条件查询获取所有学生记录=====");
        for(Object obj : list)
        {
            Student s = (Student)obj;
            System.out.println(s.getName());
            Set<Enrolment> enrolments = s.getEnrolments();
            System.out.println("=====获取-" + s.getName()
                + "-的选课记录=====");
            for(Enrolment e : enrolments)
            {
                System.out.println(e.getCourse().getName());
            }
        }
        tx.commit();
        HibernateUtil.closeSession();
    }
}
```

提示：

在执行该程序之前，应该先将本书光盘 codes\06\6.5\criteria 路径下的 data.sql 脚本导入 MySQL 数据库，这样保证查询的数据表中已有数据。

在条件查询中，Criteria 接口代表一次查询，该查询本身不具备任何的数据筛选功能，Session 调用 createCriteria(Class clazz)方法对某个持久化类创建条件查询实例。上面程序中第一行粗体字代码调用 createCriteria(Student.class)表明创建针对 Student 实体的查询。

Criteria 对象并不具备任何数据筛选功能，但程序可以通过向 Criteria 对象中组合多个 Criterion（每个 Criterion 对象代表一个过滤条件）来实现数据过滤。上面程序中第二行粗体字代码调用 Criteria 的 add(Criterion c)方法添加了一个查询条件：Student 实体的 name 属性大于字符串"a"（在 SQL 语句中，字符串之间可以比较大小）。

Criteria 包含如下两个方法。

- Criteria setFirstResult(int firstResult)：设置查询返回的第一行记录。
- Criteria setMaxResults(int maxResults)：设置查询返回的记录数。

这两个方法与 Query 的两个类似方法的功能一样，都用于完成查询分页。

而 Criteria 还包含如下常用方法。

- Criteria add(Criterion criterion)：增加查询条件。

- Criteria addOrder(Order order)：增加排序规则。
- List list()：返回结果集。

Criterion 接口代表一个查询条件，该查询条件由 Restrictions 负责产生。Restrictions 是专门用于产生查询条件的工具类，它的方法大部分都是静态方法，常用的方法如下。

- static eq|ne|gt|ge|lt|le(String propertyName, Object value)：判断指定属性值是否等于|不等于|大于|大于等于|小于|小于等于指定值。
- static eq|ne|gt|ge|lt|leProperty(String propertyName, String otherPropertyName)：判断第一个属性（由 propertyName 参数确定）的值是否等于|不等于|大于|大于等于|小于|小于等于第二个属性（由 otherPropertyName 参数确定）的值。
- static Criterion allEq(Map propertyNameValues)：判断指定属性（由 Map 参数的 key 指定）和指定值（由 Map 参数的 value 指定）是否完全相等。
- static Criterion between(String propertyName, Object lo, Object hi)：判断属性值在某个值范围之内。
- static Criterion ilike(String propertyName, Object value)：判断属性值匹配某个字符串（不区分大小写）。
- static Criterion ilike(String propertyName, String value, MatchMode matchMode)：判断属性值匹配某个字符串（不区分大小写），并确定匹配模式。
- static Criterion like(String propertyName, Object value)：判断属性值匹配某个字符串（区分大小写）。
- static Criterion like(String propertyName, String value, MatchMode matchMode)：判断属性值匹配某个字符串（区分大小写），并确定匹配模式。
- static Criterion in(String propertyName, Collection values)：判断属性值在某个集合内。
- static Criterion in(String propertyName, Object[] values)：判断属性值是数组元素的其中之一。
- static Criterion isEmpty(String propertyName)：判断属性值是否为空。
- static Criterion isNotEmpty(String propertyName)：判断属性值是否不为空。
- static Criterion isNotNull(String propertyName)：判断属性值是否为空。
- static Criterion isNull(String propertyName)：判断属性值是否不为空。
- static Criterion not(Criterion expression)：对 Criterion 求否。
- static Criterion sizeEq(String propertyName, int size)。判断某个属性的元素个数是否与 size 相等。
- static Criterion sqlRestriction(String sql)：直接使用 SQL 语句作为筛选条件
- static Criterion sqlRestriction(String sql, Object[] values, Type[] types)：直接使用带参数占位符的 SQL 语句作为条件，并指定多个参数值。
- static Criterion sqlRestriction(String sql, Object value, Type type)：直接使用带参数占位符的 SQL 语句作为条件，并指定参数值。

Order 实例代表一个排序标准，Order 有如下静态方法。

- static Order asc(String propertyName)：根据 propertyName 属性升序排序。
- static Order desc(String propertyName)：根据 propertyName 属性降序排序。

6.5.1 关联和动态关联

如果需要使用关联实体的属性来增加查询条件，则应该对属性再次使用 createCriteria()方法。createCriteria()方法有如下重载版本。

- Criteria createCriteria(String associationPath)：使用默认的连接方式进行关联。
- Criteria createCriteria(String associationPath, JoinType joinType)：以 JoinType 指定的连接方式（支持 INNER_JOIN、LEFT_OUTER_JOIN、RIGHT_OUTER_JOIN、FULL_JOIN 等枚举值）进行关联。
- Criteria createCriteria(String associationPath, String alias)：该方法的功能与上面第 1 个方法的功能

基本相似，只是该方法允许为关联实体指定别名。

- Criteria createCriteria(String associationPath, String alias, JoinType joinType)：该方法的功能与上面第 2 个方法的功能基本相似，只是该方法允许为关联实体指定别名。
- Criteria createCriteria(String associationPath, String alias, JoinType joinType, Criterion withClause)：该方法的功能最强大，该方法既可为关联实体指定别名，也可指定连接类型，还可通过 withClause 指定自定义的连接条件——这可用于实现非等值连接。

对比上面两个方法，不难发现两个方法的功能基本相似，只是第二个方法能显式指定连接方式。例如，如下查询方法。

程序清单：codes\06\6.5\criteria\src\lee\CriteriaTest.java

```
// 示范根据关联实体的属性过滤数据
private void queryWithJoin()
{
    // 打开 Session 和事务
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();
    // 使用 createCriteria 开始条件查询
    List list = session.createCriteria(Student.class)
        // 此处增加限制条件必须是 Student 实体存在的属性
        .add( Restrictions.gt("studentNumber" , 20050231))
        // 如果要增加对 Student 的关联类的属性限制
        // 则必须重新 createCriteria()
        // 如果此关联属性是集合，则只要集合里任意一个对象的属性满足下面条件即可
        .createCriteria("enrolments")
        .add( Restrictions.gt("semester" , 2))
        .list();
    System.out.println("=====关联条件查询获取所有学生记录=====");
    for (Object obj : list)
    {
        Student s = (Student)obj;
        System.out.println(s.getName());
        Set<Enrolment> enrolments = s.getEnrolments();
        System.out.println("=====获取-" + s.getName()
            + "-的选课记录=====");
        for(Enrolment e : enrolments)
        {
            System.out.println(e.getCourse().getName());
        }
    }
    tx.commit();
    HibernateUtil.closeSession();
}
```

正如从上面条件查询示例的 queryWithJoin()方法中的粗体字代码所看到的，上面的代码表示建立 Person 类的条件查询，第一个查询条件是直接过滤 Person 的属性，第二个查询条件则过滤 Person 的关联实体的属性，其中 enrolments 是 Person 类的关联实体，而 semester 则是 Enrolment 类的属性。值得注意的是，返回的并不是 Enrolment 对象的集合，而是 Person 对象的集合。

注意：

使用关联类的条件查询，依然是查询原有持久化类的实例，而不是查询被关联类的实例。

为了达到这种效果，也可使用 createAlias()方法来代替 createCriteria()方法。createAlias()方法同样有三个重载的版本。

- Criteria createAlias(String associationPath, String alias)：该方法的功能基本等同于 Criteria createCriteria(String associationPath, String alias)。
- Criteria createAlias(String associationPath, String alias, JoinType joinType)：该方法的功能基本等同

于 Criteria createCriteria(String associationPath, String alias, JoinType joinType)。

- Criteria createAlias(String associationPath, String alias, JoinType joinType, Criterion withClause)：该方法的功能基本等同于 Criteria createCriteria(String associationPath, String alias, JoinType joinType, Criterion withClause)。

因此，程序可以将上面的查询替换成如下形式：

```
List l = session.createCriteria(Student.class)
    .add( Restrictions.gt("studentNumber" , 20050231L))
    .createAlias("enrolments", "en")
    .add( Restrictions.gt("en.semester" , 2))
    .list();
```

createAlias()方法并不创建一个新的 Criteria 实例，它只是给关联实体（包括集合里包含的关联实体）起一个别名，让后面的过滤条件可根据该关联实体进行筛选。

在默认情况下，条件查询将根据持久化注解指定的延迟加载策略来加载关联实体，如果希望在条件查询中改变延迟加载策略（就像在 HQL 查询中使用 fetch 关键字一样），则可通过 Criteria 的 setFetchMode()方法来实现，该方法也接受一个 FetchMode 枚举类型的值，FetchMode 支持如下枚举值。

- DEFAULT：使用配置文件指定延迟加载策略处理。
- JOIN：使用外连接、预初始化所有的关联实体。
- SELECT：启用延迟加载，系统将使用单独的 select 语句来初始化关联实体。只有当真正访问关联实体的时候，才会执行第二条 select 语句。

如果想让程序在初始化 Student 对象时，也可以初始化 Student 关联的 Enrolment 实体，则可使用如下查询方法：

```
// 示范 FetchMode 的用法
private void queryWithFecth()
{
    // 打开 Session 和事务
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();
    // 使用 createCriteria 开始条件查询
    List list = session.createCriteria(Student.class)
        // 此处增加限制条件必须是 Student 实体存在的属性
        .add( Restrictions.gt("studentNumber" , 20050231))
        .setFetchMode("enrolments", FetchMode.JOIN)
        .list();
    tx.commit();
    HibernateUtil.closeSession();     // ①
    System.out.println("=====启用预初始化的条件查询获取所有学生记录=====");
    for (Object obj : list)
    {
        Student s = (Student)obj;
        System.out.println(s.getName());
        // Session 关闭后访问 Student 的关联实体
        Set<Enrolment> enrolments = s.getEnrolments();     // ②
        System.out.println(enrolments);
    }
}
```

上面程序的粗体字代码将会预初始化 Student 关联的 enrolments 集合。由于上面的粗体字代码让程序查询 Student 实体时同步抓取了 Student 关联的 Enrolment 实体，因此虽然程序在①号代码处关闭了 Hibernate Session，但程序依然可以在②号代码处通过 Student 获取关联的 Enrolment 实体。如果将程序中粗体字代码注释掉，程序将会在②号代码处引发 LazyInitializationException 异常（延迟初始化异常）——导致该异常的典型原因就是，当程序试图获取延迟加载的关联实体或集合属性时，但 Session 的关闭导致无法加载到关联实体或集合属性，程序就会引发 LazyInitializationException 异常了。

6.5.2 投影、聚合和分组

投影运算实际上就是一种基于列的运算，通常用于投影到指定列（也就是过滤其他列，类似于 select

子句的作用），还可以完成 SQL 语句中常用的分组、组筛选等功能。

Hibernate 的条件过滤中使用 Projection 代表投影运算，Projection 是一个接口，而 Projections 作为 Projection 的工厂，负责生成 Projection 对象。

一旦产生了 Projection 对象之后，就可通过 Criteria 提供的 setProjection(Projection projection)方法来进行投影运算。从该方法上看，每个 Criteria 只能接受一个投影运算，似乎无法进行多个投影运算，但实际上 Hibernate 又提供了一个 ProjectionList 类，该类是 Projection 的子类，并可以包含多个投影运算，通过这种方式即可完成多个投影运算。

因此，一个条件查询的投影运算通常有如下程序结构：

```
List cats = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
    .add( Projections.rowCount() )
    .add( Projections.avg("weight") )
    .add( Projections.max("weight") )
    .add( Projections.min("weight") )
    .add( Projections.groupProperty("color") ))
    .addOrder( Order.asc("color") )
    .list();
```

从上面的粗体字代码可以看出，所谓投影运算实际上和 SQL 语句里的聚集函数、分组语句（group by 子句）有类似的功能。使用条件查询的投影运算时，不能使用显式的分组子句，但某些投影类型的实质就是分组投影，这些投影运算将出现在 SQL 的 group by 子句中——如上面的 groupProperty("color")投影。

在 Projections 工具类中提供了如下几个静态方法。

- AggregateProjection avg(String propertyName)：计算特定属性的平均值。类似于 avg 函数。
- CountProjection count(String propertyName)：统计查询结果在某列上的记录条数。类似于 count(column)函数。
- CountProjection countDistinct(String propertyName)：统计查询结果在某列上不重复的记录条数。类似于 count(distinct column)函数。
- PropertyProjection groupProperty(String propertyName)：将查询结果按某列上的值进行分组。类似于添加 group by 子句。
- AggregateProjection max(String propertyName)：统计查询结果在某列上的最大值。类似于 max 函数。
- AggregateProjection min(String propertyName)：统计查询结果在某列上的最小值。类似于 min 函数。
- Projection rowCount()：统计查询结果的记录条数。类似于 count(*)的功能。
- AggregateProjection sum(String propertyName)：统计查询结果在某列上的总和。类似于 sum 函数。

下面的程序示范了如何通过投影运算来进行分组，使用聚集函数功能。

程序清单：codes\06\6.5\projection\src\lee\ProjectionTest.java

```
public class ProjectionTest
{
    public static void main(String[] args)
    {
        ProjectionTest pt = new ProjectionTest();
        pt.query();
        HibernateUtil.sessionFactory.close();
    }

private void query()
{
    // 打开 Session 和事务
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();
    // 使用 createCriteria 开始条件查询
    List list = session.createCriteria(Enrolment.class)
```

```
            .createAlias("student", "s")
            .setProjection(Projections.projectionList()
                // 统计记录条数
                .add(Projections.rowCount())
                // 统计选择该课程里最大的学生姓名
                .add(Projections.max("s.name"))
                // 按 course 进行分组
                .add(Projections.groupProperty("course"))
            ).list();
        for(Object ele : list)
        {
            Object[] objs = (Object[])ele;
            Course c = (Course)objs[2];
            System.out.println("=====<" + c.getName()
                + ">课程的选课统计=====");
            System.out.println("选课人数:" + objs[0]);
            System.out.println("选课的姓名最大的学生为：" + objs[1]);
        }
        tx.commit();
        HibernateUtil.closeSession();
    }
```

上面程序的粗体字代码为条件查询增加了分组功能，并通过 Projections 统计了每组的记录条数，统计 Student 名字最大的值。

从上面的程序可以看出，对于增加了投影运算后的条件查询，查询返回的结果是数组，数组的前 *N* 个元素依次是投影运算的结果，最后一个数组元素才是条件查询得到的实体。由此可见，投影运算的实质和 group by 子句、聚集函数的功能大致一致。

除此之外，如果希望对分组（投影）后属性进行排序，那就需要为投影运算指定一个别名。为投影运算指定别名有如下三种方法。

- 使用 Projections 的 alias()方法为指定投影指定别名。

Projections 的 alias()方法为指定 Projection 指定别名，并返回原 Projection 对象。一旦为指定 Projection 指定了别名，程序就可以根据该 Projection 别名来进行其他操作了，比如排序。条件查询示例如下：

```
    List list = session.createCriteria(Enrolment.class)
        .setProjection(Projections.projectionList()
        // 按 course 进行分组
        .add(Projections.groupProperty("course"))
        // 统计记录条数，并为统计结果指定别名 c
        .add(Projections.alias(Projections.rowCount(), "c")))
        .addOrder(Order.asc("c"))
        .list();
```

上面程序的粗体字代码示范了为 Projection 指定别名的方法。

- 使用 SimpleProjection 的 as()方法为自身指定别名。

如果条件查询所使用的投影运算是 SimpleProjection 及其子类，则可直接使用该投影对象的 as()方法来为自身指定别名。条件查询片段如下：

```
    List list = session.createCriteria(Enrolment.class)
        .setProjection(Projections.projectionList()
        // 按 course 进行分组，并为分组结果指定别名 c
        .add(Projections.groupProperty("course").as("c"))
        // 统计记录条数
        .add(Projections.rowCount()))
        .addOrder(Order.asc("c"))
        .list();
```

- 使用 ProjectionList 的 add()方法添加投影时指定别名。

ProjectionList 的 add()方法有两种重载形式：一种是直接添加一个投影；另一种是在添加投影时指定别名。条件查询片段如下：

```
    List list = session.createCriteria(Enrolment.class)
        .setProjection(Projections.projectionList()
```

```
        // 按 course 进行分组，指定别名为 c
        .add(Projections.groupProperty("course") , "c")
        // 统计每组记录的条数，指定别名为 rc
        .add(Projections.rowCount(), "rc"))
        .addOrder(Order.asc("rc"))
        .list();
```

除此之外，Hibernate 还提供了 Property 执行投影运算，Property 投影的作用类似于 SQL 语句中的 select，条件查询的结果只有被 Property 投影的列才会被选出。看如下的条件查询：

```
    // 使用 Property 只选出指定列
    List list = session.createCriteria(Student.class)
        .setProjection(Property.forName("name"))
        .list();
```

上面的条件查询执行的结果不再是 Student 对象集合，而是 name 属性所组成的集合。

使用 Property 还可根据指定列来过滤记录，看如下的条件查询：

```
    // 使用 Property 选出指定列，并根据指定列过滤数据
    List list = session.createCriteria(Enrolment.class)
        .createAlias("student", "s")
        .setProjection(Projections.projectionList()
        .add(Property.forName("course"))
        .add(Property.forName("s.name")))
        .add(Property.forName("s.name").eq("孙悟空"))      // ①
        .list();
```

上面的条件查询稍微有点复杂，条件查询中粗体字代码用于创建一个 ProjectionList 对象，其中两个 add()方法都属于 ProjectionList 对象，表明选出 Enrolment 的 couse 属性、Enrolment 的 student 的 name 属性。①号粗体字代码的 Property 用于根据指定列过滤属性，即只选出 Enrolment 的 student 的 name 属性等于“孙悟空”的记录。执行上面的条件查询，将生成如下的 SQL 语句：

```
    select
        this_.course_code as y0_,
        s1_.name as y1_
    from
        enrolment_inf this_
    inner join
        student_inf s1_
            on this_.student_id=s1_.student_id
    where
        s1_.name=?
```

6.5.3 离线查询和子查询

条件查询的离线查询由 DetachedCriteria 来代表，DetachedCriteria 类允许在一个 Session 范围之外创建一个查询，并且可以使用任意 Session 来执行它。

使用 DetachedCriteria 来执行离线查询，通常使用如下方法来获得一个离线查询：

```
    // 创建指定持久化类的离线查询
    DetachedCriteria.forClass(Class entity)
```

除此之外，DetachedCriteria 还可代表子查询，当把 DetachedCriteria 传入 Criteria 中作为查询条件时，DetachedCriteria 就变成了子查询。条件实例包含子查询通过 Subqueries 或者 Property 来获得。

如下程序示范了使用 DetachedCriteria 进行离线查询和子查询的效果。

程序清单：codes\06\6.5\DetachedCriteria\src\lee\DetachedCriteriaTest.java

```
    public class DetachedCriteriaTest
    {
        public static void main(String[] args)
        {
            DetachedCriteriaTest pt = new DetachedCriteriaTest();
            pt.datached();
```

```
        pt.subQuery();
        HibernateUtil.sessionFactory.close();
    }
    // 执行离线查询
    private void datached()
    {
        // 定义一个离线查询
        DetachedCriteria query = DetachedCriteria
            .forClass(Student.class)
            .setProjection(Property.forName("name"));
        // 打开 Session 和事务
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
        // 执行离线查询
        List list = query.getExecutableCriteria(session)
            .list();
        // 遍历查询的结果
        for (Object obj : list)
        {
            System.out.println(obj);
        }
        tx.commit();
        HibernateUtil.closeSession();
    }
    // 执行子查询
    private void subQuery()
    {
        // 定义一个离线查询
        DetachedCriteria subQuery = DetachedCriteria
            .forClass(Student.class)
            .setProjection(Property.forName("name"));
        // 打开 Session 和事务
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
        // 执行子查询
        List list = session.createCriteria(Student.class)
            // 下面两行代码的作用相同，都示范了通过子查询添加查询条件
//          .add( Property.forName("name").in(subQuery))
            .add( Subqueries.propertyIn("name" , subQuery))
            .list();
        // 遍历查询的结果
        for (Object obj : list)
        {
            System.out.println(obj);
        }
        tx.commit();
        HibernateUtil.closeSession();
    }
}
```

从上面的程序来看，当创建一个 DetachedCriteria 对象之后，该对象到底被作为离线查询使用，还是作为子查询使用，与 DetachedCriteria 对象无关。

如果程序使用 Session 的 getExecutableCriteria()方法来执行 DetachedCriteria 对象，则它被当成离线查询使用；如果程序使用 Property 的系列方法（或 Subqueries 的系列类方法）来操作 DetachedCriteria 对象，则它被当成子查询使用。

Property 类提供了 eq()、eqAll()、ge()、geAll()、gt()、gtAll()、in 等系列方法，这些方法与 SQL 子查询中的运算符一一对应。除此之外，Subqueries 也提供了 q()、eqAll()、ge()、geAll()、gt()、gtAll()、in 等静态方法，这些方法与 Property 同名的实例方法的功能基本相似，在这种情况下， DetachedCriteria 被当成子查询使用。关于 Property 的实例方法、Subqueries 的类方法，请参考 Hibernate 相关 API 文档。