EJB3 持久化规范

正式版

卫建军

2008/1/6

译者序

Java 是当前 IT 领域中比较流行的技术之一。J2EE 是当前比较流行的企业级应用架构。本人一直致力于 J2EE 架构的学习和研究，但是总是对英文文档有不可言语的恐惧。我想很多 J2EE 爱好者和我有同样的感觉。这样就影响了我们深入学习 J2EE 原始规范的兴趣。但是 J2EE 原始的规范文档对我们深入理解 J2EE 有很大的帮助，因为它阐述了规范的来龙去脉，以及违反了规范会造成什么样的影响。了解了这些缘由和影响，会使我们对 J2EE 架构有更深层次的理解。这也是我翻译该规范的动力所在。

由于本人的英语水平有限，翻译中难免会出现错误和拗口之处，请大家多多指教。

这次主要翻译的规范有《EJB3 规范简化版》、《J2EE5.0 规范》、《EJB 核心规范》、《EJB3 持久化规范》和《JMS1.1 规范》。希望对大家有所帮助。

卫建军
2008-1-5 于北京

目录

1 实体

实体是一个可持久化的域对象。主要的程序产物是实体类，实体类可以使用辅助类作为帮助类或者用于呈现实体的状态。

1.1 实体类的要求

- 实体类必须用 entity 标识符来声明，或者在配制文件中指明某个类为实体类。
- 实体类必须有一个无参数的构造器。它也可以有其他的构造器。这个无参数的构造器必须是 public 或 protected 的。
- 实体类必须是最高层的类。枚举或接口不应当被指派为实体。
- 如果实体实例作为一个脱管对象按值传递（如通过一个远程接口），则实体类必须实现 serializable 接口。
- 实体类不允许是 final 的，它的所有方法都不允许是 final 的。
- 实体支持继承，多义关联，多义查询。
- 实体类可以是抽象的，也可以是具体的。实体可以继承非实体类，非实体类也可以继承实体类。
- 实体的持久化状态由它的实例变量代表，这些变量和 JavaBean 的属性一样的。实体可以通过它自己的实体方法直接获得实例变量，但是实体的客户端必须通过获取变量的方法（getter/setter）来获取变量。实例变量必须是 private，protected 或包内可见的。

1.1.1 持久化字段和属性

持久化提供商运行时可以通过它的 setter/getter 方法或实例变量获得实体的持久化状态。一个单一的获取类型（字段或属性获取）应用到实体的层级上。当使用注解符时，在实体的持久化字段或持久化属性上的映射注解符指明获取类型是基于字段还是基于属性：

- 如果声明实体是基于字段获取，那么持久化提供商运行时（指持久化实

现的运行环境。在 Java EE 环境中，可以是 Java EE 容器或者是第三方持久化提供商实现）直接获取实例变量。并且持久化所有 non-transient 实例变量（没有用 Transient 标识符标识的变量）。当使用基于字段获取式，O/R 映射注解符注解实例变量。

- 如果声明实体是基于属性获取，那么持久化提供商运行时通过 getter/setter 方法获取实例变量，并且持久化所有 non-transient 实例变量（没有用 Transient 标识符标识的变量）。所有的属性获取方法必须是 public 或 protected。当使用基于属性获取，O/R 映射的注解符注解在 getter 方法上。（注解符不必应用到 setter 方法上）
- 映射注解符不能应用到那些是 transient 或 Transient 的字段或属性上。
- 如果在字段和属性都使用了映射注解符或如果在类层次内 XML 配置符指明使用不同的获取类型，那么持久化提供商运行时的行为没有规定。

当使用持久化属性时，对持久化属性要求实体遵循 JavaBean 的读写属性的方法约定。

在这种情况下，实体的类型 T 的每一个持久化属性 property，都有一个 getter 方法 getProperty 和一个 setter 方法 setProperty，对于 boolean 值则为 isProperty 或者为 getter。（特殊情况下，如果 getX 是 getter 方法的名称，且 setX 是 setter 方法的名称，那么持久化属性的名称由 java.beans.Introspector.decapitalize(X)的结果决定）

对于单值的持久化属性，这些方法如下形式：

- T getProperty()
- void setProperty(T t)

值是集合的持久化字段和属性必须根据下列集合值接口定义，而不管实体类是否遵守 JavaBean 的方法规定，也不管是否使用基于字段还是基于属性获取：java.util.Collection，java.util.List（除非使用 OrderBy 结构且列表的改变遵循指定的顺序，否则可移植的应用不应当期望在多个持久化上下文间列表的顺序相同。否则列表的顺序不被持久化），java.util.Set，java.util.Map（它的实现类型可以被应用用于在实体被持久化之前实例化字段或属性；一旦实体变成受管理实体或脱

管实体，随后必须通过这个接口类型获取实体状态）。

对于集合类型的字段和属性，在上述的方法形式中的类型 T 必须是上述集合接口类型中的一个。也可以使用这些集合类型的泛型变量（如：Set<Order>）。

为了返回和设置实例的持久化状态，实例的属性获取方法内可以包含别的业务逻辑，如执行验证。当使用基于属性的获取类型时，持久化运行环境才会执行这些业务逻辑。

注意，当使用基于属性的获取时，在获取方法内增加的业务逻辑应当增加警告。当加载或存储持久化状态时，没有规定持久化提供商调用这些方法的顺序。因此在这些方法内包含的逻辑不能依赖于特定的调用顺序。

如果使用基于属性的获取，并且指定为懒惰获取数据，那么可移植的应用在持久化提供商获取实体状态之前，不应当直接用受管理实体实例的属性方法来获取实体的状态。（懒惰获取数据是持久化提供商的缺省获取方式，可以通过 Basic，OneToOne，ManyToOne 和 ManyToMany 注解符指定，用 XML 是等价的。参看第 8 章）

在属性获取方法内抛出的运行时异常会引起事务回滚。当持久化运行时环境使用这些属性获取方法加载或存储实例状态时，如果抛出应用异常，则会引起持久化运行环境回滚事务，并且会抛出封装了应用异常的 PersitenceException。

实体的子类可以覆盖它的属性获取方法。然而，可移植的应用不需要覆盖应用在父类持久化字段和属性上的 O/R 影射元数据。

实体类的持久化字段和属性可以是原始类型、java.lang.String、也可以是其他可序列化类型（包括原始类型的封装类型，java.math.BigInteger，java.math.BigDecimal，java.util.Date，java.util.Calendar（注意 Calendar 类型的字段必须被完全初始化），java.sql.Date，java.sql.Time，java.sql.Timestamp，用户自定义类型，byte[]，Byte[]，char[]和 Character[]）、enum、实体类型和/或实体类型的集合、以及可嵌入类（参见 1.1.5）。

O/R 影射元数据可以用来客户化 O/R 影射、实体状态和实体间关系的加载和存储。参见第 8 章。

1.1.2 举例

```
@Entity
public class Customer implements Serializable {

  private Long id;

  private String name;

  private Address address;

  private Collection<Order> orders = new HashSet();

  private Set<PhoneNumber> phones = new HashSet();

  // No-arg constructor
  public Customer() {}

  @Id    // property access is used
  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public Address getAddress() {
    return address;
  }

  public void setAddress(Address address) {
    this.address = address;
  }

@OneToMany
  public Collection<Order> getOrders() {
    return orders;
  }

  public void setOrders(Collection<Order> orders) {
    this.orders = orders;
  }
```

```
@ManyToMany
public Set<PhoneNumber> getPhones() {
  return phones;
}

public void setPhones(Set<PhoneNumber> phones) {
  this.phones = phones;
}

   // Business method to add a phone number to the customer
   public void addPhone(PhoneNumber phone) {
     this.getPhones().add(phone);
     // Update the phone entity instance to refer to this customer
     phone.addCustomer(this);
   }
 }
```

1.1.3 创建实体实例

通过 new 操作符创建实体实例。当第一次创建实体实例时，这个实例是非持久化的。通过 EntityManager 的 API 可以将实例持久化。实体实例的生命周期在 2.2 章节中描述。

1.1.4 主键和实体唯一标识

每一个实体必须有一个主键。

主键必须定义在实体层次的根实体上或者定义在实体层次的被映射的超类上。在实体层次内只能定义主键一次。

一个简单主键（非组合主键）必须对应于实体类的一个单一持久化字段或属性。Id 注解符用于声明一个简单主键。参见 8.1.8 章节。

一个组合主键必须对应于一个单一持久化字段或属性，或者对应于一组持久化字段或属性。即必须定义一个主键类来代表组合主键。组合主键通常在这种情况下发生：当映射逻辑数据库时，数据库的主键是由几列组合而成的。EmbededId 和 IdClass 注解符用于声明组合主键。参见 8.1.14 和 8.1.15 章节。

主键（或字段、或组合主键的属性）的类型必须是：任何原始类型、任何原始类型的封装类型、java.lang.String、java.util.Date、java.sql.Date。然而，通常情况下，复杂的数字类型（如 float 类型）不要用作主键。主键不是上述类型的应

用将是不可移植的。如果使用生成的主键，只有整型类型是可移植的。如果使用 java.util.Date 作为主键字段或属性，那么临时类型应当作为 DATE。

主键类的获取类型（基于属性或基于字段）由实体的获取类型决定。

组合主键需遵循以下规则：

- 主键类必须是 public 的，并且有一个 public 的无参数构造器。
- 若 access=PROPERTY，则主键类的属性必须是 public 的或 protected 的。
- 主键类必须是可序列化的。
- 主键类必须定义 equals 和 hashCode 方法。值相等的语义必须和数据库中主键相等的语义一致。（即主键相等就认为是值相等）
- 组合主键必须被映射为可嵌入类（参见 8.1.14 章节），也可以被映射为实体类的多个字段或属性（参见 8.1.15 章节）。
- 如果组合主键类被映射到实体类的多个字段或属性，那么主键类的字段或属性的名称必须和实体类的相应字段或属性的名称和类型一致。

应用程序不能够改变主键的值（这包括不能改变作为主键或组合主键元素的可变类型的值）。规范中没有定以发生改变时的处理方式（实现可以但不要求抛出一个异常。可移植的应用不应当依赖这些特殊的行为）。

1.1.5 可嵌入类

实体类可以用别的类来代表实体状态。这些类实例没有可持久化的唯一标识。相反，他们仅仅作为他们所属实体的被嵌入对象。这些对象绝对是属于他们的属主实体类，而不会被其他实体类共享。共享这些被嵌入对象会造成语义混淆，因为这些对象没有唯一标识符，他们和实体实例一起才能构成完整的数据库映射（在以后的版本中可能会支持被嵌入对象的集合、可嵌入类的多义和继承）。

对可嵌入类的其他要求在 8.1.34 章节中描述。

1.1.6 对非关系字段或属性的映射缺省值

如果一个可持久化字段或属性（非关系属性）没有用第 8 章定义的映射注解

符注解的话（或者在 XML 中也没有等价的信息），则需要按顺序应用下面的规则：

- 如果是一个类，而且这个类用@Embeddable 注解，则它被映射的方式和用 Embedded 注解符注解它的字段或属性的方式是一样的。参见 8.1.34 和 8.1.35 章节。
- 如果字段或属性的类型是下面类型中的任意一个时，则它被映射的方式和它被用 Basic 注解的方式是一样的。类型有：原始类型、原始类型的封装类型、java.lang.String、java.math.BigInteger、java.math.BigDecimal、java.util.Date 、 java.util.Calendar 、 java.sql.Date 、 java.sql.Time 、 java.sql.Timestamp、byte[]、Byte[]、char[]、Character[]、enums、以及任何实现了序列化接口的类型。参见 8.1.18 和 8.1.21 章节。

如果没有使用标识符，而且也没有应用上述规则的话，就是错误的。

1.1.7 实体关系

实体间的关系可以是一对一、一对多、多对一、多对多的。关系是多义的。

如果在两个实体间有关系，则在持久化属性或引用实体的实例变量上必须指定关系模型注解符。关系注解符有：OneToOne、OneToMany、ManyToOne、ManyToMany。对没有指定目标类型关系（如，java 的泛型类型没有应用到集合类上），必须指定关系的目标实体。

这些注解符反映了在关系数据库模型中的实际情况。使用关系模型标识符使得与关系数据库相关的 O/R 映射完全缺省，更加易于配置。这在 1.1.8 章节中详细描述。

关系可以是双向的或单向的。双向关系既有所有方也有被属方。单向关系只有所有边。所有边决定了数据库中对关系的更新。详见 2.2.3。

下面的规则应用于双向关系：

- 双向关系的反向边必须用 OneToOne、OneToMany 或 ManyToMany 的 mappedBy 元素指向它的所有边。mappedBy 元素指明了实体的哪一个字段或属性是关系的拥有者。

- 双向关系中一对多或多对一的多边必须是关系的拥有边，因此在 ManyToOne 注解中不能指明 mappedBy 元素。
- 对一对一关双向关系，拥有关系边放在有外键的一边。
- 对多对多双向关系，放在哪边都是可以的。

关系模型注解限制了 cascade=REMOVE 规则的使用。cascade=REMOVE 规则只应当应用在 OneToOne 或 OneToMany 关系中。cascade=REMOVE 应用到其他关系中将是不可移植的。

其他的映射注解（如，列和表的映射注解）可以被指定用于重载或进一步改进缺省的映射，在 1.1.8 中描述。例如，外键映射可以用于单向一对多关系的映射。这种 Schema 级的映射注解符必须在关系的所有边指定。例如，如果指明多对一关系映射，那么就不允许在外键上为关系指定唯一主键限制。

持久化供应商处理关系的对象关系映射，包括关系的加载和存储到数据库中，以及处理数据库中关系的引用完整性（例如，通过外键约束）。

注意，维护运行时关系的一致性是应用的责任——例如，当应用在运行时更新关系时，要保证双向关系的“one”和“many”边互相一致。

如果从数据库获取的实体的多值关系没有关联实体，那么持久化提供商有责任返回一个空的集合作为关系的值。

1.1.8 关系映射缺省值

这一节定义用于 OneToOne、OneToMany、ManyToOne 和 ManyToMany 关系模型注解符的映射缺省值。当使用 XML 配置文件声明关系时，使用相同的映射缺省值。

1.1.8.1双向 OneToOne 关系

假定：

实体 A 引用实体 B 的一个实例。

实体 B 引用一个实体 A 的一个实例。

实体 A 作为关系的拥有者。

将使用以下的缺省映射：

实体 A 被映射到名字为 A 的表。

实体 B 被映射到名字为 B 的表。

表 A 有一个指向表 B 的外键。外键的列名格式为：实体 A 中关系字段的名称_表 B 中主键列的名称。外键列的类型和表 B 主键的类型相同，并且有一个唯一主键约束。

举例说明：

```
@Entity
public class Employee {
   private Cubicle assignedCubicle;

   @OneToOne
   public Cubicle getAssignedCubicle() {
     return assignedCubicle;
   }
   public void setAssignedCubicle(Cubicle cubicle) {
     this.assignedCubicle = cubicle;
   }
  ...
}
```

```
@Entity
public class Cubicle {
    private Employee residentEmployee;

    @OneToOne(mappedBy="assignedCubicle")
    public Employee getResidentEmployee() {
      return residentEmployee;
    }
    public void setResidentEmployee(Employee employee) {
      this.residentEmployee = employee;
    }
  ...
}
```

在上例中：

实体 Employee 引用一个实体 Cubicle 的实例。

实体 Cubicle 引用一个实体 Employee 的实例。

实体 Employee 是关系的拥有者。

将使用以下的缺省映射：

实体 Employee 被映射为名字为 EMPLOYEE 的表。

实体 Cubicle 被映射为名字为 CUBICLE 的表。

表 EMPLOYEE 有一个到表 CUBICLE 的外键。外键列名为 ASSIGNEDCUBICLE_<PK of CUBICLE>，<PK of CUBICLE>是表 CUBICLE 的主键列名。外键列和 CUBICLE 的主键有相同的类型，且有一个唯一主键约束。

1.1.8.2双向 ManyToOne/OneToMany 关系

假定：

实体 A 引用一个实体 B 的实例。

实体 B 引用一个实体 A 的集合。

那么，实体 A 必须是关系的拥有者。

将使用以下的缺省映射：

实体 A 映射成表 A。

实体 B 映射成表 B。

表 A 有一个到表 B 的外键。外键名的格式是实体 A 中关系属性的名称_表 B 的主键名称。外键和表 B 的主键具有相同的类型。

```
@Entity
public class Employee {
   private Department department;

   @ManyToOne
   public Department getDepartment() {
     return department;
   }
   public void setDepartment(Department department) {
     this.department = department;
   }
  ...
}
```

```
@Entity
public class Department {
   private Collection<Employee> employees = new HashSet();

   @OneToMany(mappedBy="department")
   public Collection<Employee> getEmployees() {
     return employees;
   }

   public void setEmployees(Collection<Employee> employees) {
     this.employees = employees;
   }
  ...
}
```

在上例中：

实体 Employee 引用了一个实体 Department 实例。

实体 Department 引用了一个 Employee 的集合。

实体 Employee 是关系的拥有者。

将使用以下的缺省映射：

实体 Employee 映射成表名为 EMPLOYEE 的表。

实体 Department 映射成表明为 DEPARTMENT 的表。

表 EMPLOYEE 有一个指向 DEPARTMENT 的外键。外键名为 DEPARTMENT_<PK of DEPARTMENT>。外键和 DEPARTMENT 的主键有相同的类型。

1.1.8.3单向单值关系

假定：

实体 A 引用一个实体 B 的实例。

实体 B 没有引用实体 A。

单向关系只有一个拥有者边，在上例中只能是实体 A。

单向单值关系模型可以作为单向 OneToOne 或单向 ManyToOne 关系。

1.1.8.3.1 单向 OneToOne 关系

将应用以下缺省的映射规则：

实体 A 映射成名称为 A 的表。

实体 B 映射成名称为 B 的表。

表 A 有一个指向表 B 的外键。外键名称的格式为：实体 A 的关系字段名称_表 B 的主键名称。外键和表 B 的主键具有相同的类型和约束条件。

举例如下：

```
@Entity
public class Employee {
   private TravelProfile profile;

   @OneToOne
   public TravelProfile getProfile() {
     return profile;
   }
   public void setProfile(TravelProfile profile) {
     this.profile = profile;
   }
  ...
}


@Entity
public class TravelProfile {
   ...
}
```

上例中：

实体 Employee 引用一个实体 TravelProfile 的实例。

实体 TravelProfile 没有引用 Employee。

实体 Employee 是关系的拥有者。

将应用以下映射规则：

实体 Employee 映射为名称为 EMPLOYEE 的表。

实体 TravelProfile 映射为名称为 TRAVELPROFILE 的表。

表 EMPLOYEE 有一个指向 TRAVELPROFILE 的外键。外键的列名为 PROFILE_<PK of TRVELPROFILE>，<PK of TRVELPROFILE> 是标

TARAVELPROFILE 的主键名称。外键和表 TARAVELPROFILE 的主键有相同的类型和唯一性约束条件。

1.1.8.3.2 单向 ManyToOne 关系

将使用以下缺省的映射规则:

实体 A 映射成表 A。

实体 B 映射成表 B。

表 A 有一个指向表 B 的外键。外键的列名的格式为：实体 A 的关系字段名称_表 B 的主键名称。外键和表 B 的主键具有相同的类型。

举例如下：

```
@Entity
public class Employee {
   private Address address;

   @ManyToOne
   public Address getAddress() {
     return address;
   }
   public void setAddress(Address address) {
     this.address = address;
   }
  ...
}

@Entity
public class Address {
   ...
}
```

在上例中：

实体 Employee 映射成表 EMPLOYEE。

实体 Address 映射成表 ADDRESS。

表 EMPLOYEE 有一个指向表 ADDRESS 的外键。外键列名为 ADDRESS_<PK of ADDRESS>，<PK of ADDRESS>是表 ADDRESS 的主键名称。外键和表 ADDRESS 的主键有相同的类型。

1.1.8.4双向 ManyToMany 关系

假定：

实体 A 引用一个实体 B 的集合。

实体 B 引用一个实体 A 的集合。

实体 A 是关系的拥有者。

将使用以下映射规则：

实体 A 映射成表 A。

实体 B 映射成表 B。

有一个关联表 A_B（关系拥有者的名称放在前）。关联表有两个外键列。一个外键指向表 A，这个外键和表 A 的主键具有相同的类型。外键的名称格式为：实体 B 中指向实体 A 的字段的名称_表 A 的主键名称。另外一个外键指向表 B，并且和表 B 的主键具有相同的类型。这个外键的名称为：实体 A 中指向实体 B 的字段的名称_表 B 的主键名称。

举例如下：

```
@Entity
public class Project {
   private Collection<Employee> employees;

   @ManyToMany
   public Collection<Employee> getEmployees() {
     return employees;
   }

   public void setEmployees(Collection<Employee> employees) {
     this.employees = employees;
   }
   ...
}

@Entity
public class Employee {
   private Collection<Project> projects;

   @ManyToMany(mappedBy="employees")
   public Collection<Project> getProjects() {
     return projects;
   }

   public void setProjects(Collection<Project> projects) {
     this.projects = projects;
   }
  ...
}
```

在上例中：

实体 Project 引用一个实体 Employee 的集合。

实体 Employee 引用一个实体 Project 的集合。

实体 Project 是关系的拥有者。

将使用下面的映射规则：

实体 Project 映射成表 Project。

实体 Employee 映射成表 EMPLOYEE。

有一个关联表 PROJECT_EMPLOYEE。关联表有两个外键，一个是 PROJECTS_<PK of PROJECT>，一个是 EMPLOYEES_<PK of EMPLOYEE>。

1.1.8.5单向多值关系

假定：

实体 A 引用一个实体 B 的集合。

实体 B 没有引用实体 A。

单向关系只有一个拥有者边，在这个情况下必须是实体 A。

单向多值关系关系模型可以作为单向 OneToMany 或单向 ManyToMany 关系。

1.1.8.5.1 单向 OneToMany 关系

将使用以下映射规则：

实体 A 映射成表 A。

实体 B 映射成表 B。

有一个关系表 A_B。这个关联表有两个外键列。一个外键指向表 A，这个外键和表 A 的主键具有相同的类型。外键的名称格式为：实体 B 中指向实体 A 的字段的名称_表 A 的主键名称。另外一个外键指向表 B，并且和表 B 的主键具有相同的类型。这个外键的名称为：实体 A 中指向实体 B 的字段的名称_表 B 的主键名称。

举例如下：

```
@Entity
public class Employee {
   private Collection<AnnualReview> annualReviews;

   @OneToMany
   public Collection<AnnualReview> getAnnualReviews() {
     return annualReviews;
   }

   public void setAnnualReviews(Collection<AnnualReview> annualRe-
views) {
     this.annualReviews = annualReviews;
   }
  ...
}


@Entity
public class AnnualReview {
   ...
}
```

在上例中：

实体 Employee 引用一个实体 AnnualReview 的集合。

实体 AnnualReview 没有引用 Employee。

有一个关联表 EMPLOYEE_ANNUALREVIEW。这个关联表有两个外键：一个是 EMPLOYEE_<PK of EMPLOYEE>，一个是 ANNUALREVIEWS_<PK of ANNUALREVIEW>。

1.1.8.5.2 单向 ManyToMany 关系

将使用以下映射规则：

实体 A 映射成表 A。

实体 B 映射成表 B。

有一个关系表 A_B。这个关联表有两个外键列。一个外键指向表 A，这个外键和表 A 的主键具有相同的类型。外键的名称格式为：实体 B 中指向实体 A 的字段的名称_表 A 的主键名称。另外一个外键指向表 B，并且和表 B 的主键具有相同的类型。这个外键的名称为：实体 A 中指向实体 B 的字段的名称_表 B 的主键名称。

举例如下：

```
@Entity
public class Employee {
   private Collection<Patent> patents;

   @ManyToMany
   public Collection<Patent> getPatents() {
     return patents;
   }

   public void setPatents(Collection<Patent> patents) {
     this.patents = patents;
   }
   ...
}


@Entity
public class Patent {
   ...
}
```

在上例中：

实体 Employee 引用一个实体 Patent 的集合。

实体 Patent 没有引用实体 Employee。

实体 Employee 是关系的拥有者。

将使用以下映射规则：

实体 Employee 映射成表 EMPLOYEE。

实体 Patent 映射成表 PATENT。

用一个关联表 EMPLOYEE_PATENT。关联表中有两个外键：一个是 EMPLOYEE_<PK of EMPLOYEE>，一个是 PATENTS_<PK of PATENT>。

1.1.9 继承

一个实体可以继承自另外一个实体。实体支持继承、多态关联和多态查询。

抽象类和具体类都可以是实体，都可以用 Entity 注解，都可以映射为实体，都可以作为实体被查询。

实体可以继承自非实体类，非实体类也可以继承实体类。

当一个实体继承另一个实体时，他们的主键类型必须一致。

上边这些概念在以下章节中会有进一步的描述。

1.1.9.1抽象实体类

抽象类可以作为实体。抽象实体和具体实体的差别仅仅是抽象实体不能直接实例化。抽象实体可以作为实体被映射，也可以是查询的目标（将取出它的具体子类的实例）。

抽象实体用 Entity 注解声明或者在 XML 配置文件中声明作为实体。

下例中解释了在实体继承层次中使用抽象实体类。

例子：抽象类作为实体

```
@Entity
@Table(name="EMP")
@Inheritance(strategy=JOINED)
public abstract class Employee {
     @Id protected Integer empId;
     @Version protected Integer version;
     @ManyToOne protected Address address;
     ...
}


@Entity
@Table(name="FT_EMP")
@DiscriminatorValue("FT")
@PrimaryKeyJoinColumn(name="FT_EMPID")
public class FullTimeEmployee extends Employee {

    // Inherit empId, but mapped in this class to FT_EMP.FT_EMPID
    // Inherit version mapped to EMP.VERSION
    // Inherit address mapped to EMP.ADDRESS fk

     // Defaults to FT_EMP.SALARY
     protected Integer salary;
     ...
}


@Entity
@Table(name="PT_EMP")
@DiscriminatorValue("PT")
// PK field is PT_EMP.EMPID due to PrimaryKeyJoinColumn default
public class PartTimeEmployee extends Employee {
     protected Float hourlyWage;
     ...
}
```

1.1.9.2被映射超类

一个实体可以继承一个提供了持久化实体状态和映射信息的超类，但它自己不是实体。典型地，这样的父类用于定义多个实体公共的状态和映射信息。

被映射的超类是不可查询的，且不能作为 EntityManager 或 Query 操作的参数。被映射超类不能作为持久化关系的目标对象。

抽象类和具体类都可以是被映射超类。MappedSuperclass 注解符（或 mapped-superclass XML 配置元素）用于指派被映射超类。

指派为 MappedSuperclass 的类没有单独的表。他的映射信息应用到继承它的实体上。

由于被映射超类不存在表，所以除了被映射超类的映射只应用到它的子类之外，指派为 MappedSuperclass 的类可以用和实体同样的方式被映射。当映射应用到子类时，将在子类表的上下文中应用继承的映射。在子类中可以使用 AttributeOverride 和 AssociationOverride 注解符（或 XML 中对等的元素）来重载继承的映射。

所有其他的实体映射缺省值都可以应用到被映射超类。

下面的例子解释了作为被映射超类的具体类的定义。

例子：具体类作为被映射超类

```
@MappedSuperclass
public class Employee {

    @Id protected Integer empId;
    @Version protected Integer version;
    @ManyToOne @JoinColumn(name="ADDR")
    protected Address address;

    public Integer getEmpId() { ... }
    public void setEmpId(Integer id) { ... }
    public Address getAddress() { ... }
    public void setAddress(Address addr) { ... }
}

// Default table is FTEMPLOYEE table
@Entity
public class FTEmployee extends Employee {

    // Inherited empId field mapped to FTEMPLOYEE.EMPID
    // Inherited version field mapped to FTEMPLOYEE.VERSION
    // Inherited address field mapped to FTEMPLOYEE.ADDR fk

    // Defaults to FTEMPLOYEE.SALARY
    protected Integer salary;

    public FTEmployee() {}

    public Integer getSalary() { ... }
    public void setSalary(Integer salary) { ... }
}

@Entity @Table(name="PT_EMP")
@AssociationOverride(name="address",
        joincolumns=@JoinColumn(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {

    // Inherited empId field mapped to PT_EMP.EMPID
    // Inherited version field mapped to PT_EMP.VERSION
    // address field mapping overridden to PT_EMP.ADDR_ID fk
    @Column(name="WAGE")
    protected Float hourlyWage;

    public PartTimeEmployee() {}

    public Float getHourlyWage() { ... }
    public void setHourlyWage(Float wage) { ... }
}
```

1.1.9.3在实体继承层中的非实体类

实体可以继承自非实体类，这个非实体类可以是具体类，也可以是抽象类。（超类可以不是可嵌入类或 id 类）

非实体超类仅仅作为行为继承。非实体超类的状态不会被持久化。在实体类中任何继承自非实体超类的状态都是不持久化的。非持久状态不被 EntityManager 管理（如使用事务范围的持久化上下文，则不要求对它跨事务保留）。在这些超类上的注解符都会被忽略。

非实体类不能作为 EntityManager 或 Query 接口方法的参数，也不能携带映射信息。

下面的例子描述了非实体类作为实体类的超类。

```
public class Cart {

    // This state is transient
    Integer operationCount;

    public Cart() { operationCount = 0; }
    public Integer getOperationCount() { return operationCount; }
    public void incrementOperationCount() { operationCount++; }
}

@Entity
public class ShoppingCart extends Cart {

    Collection<Item> items = new Vector<Item>();

    public ShoppingCart() { super(); }

     ...

    @OneToMany
    public Collection<Item> getItems() { return items; }
    public void addItem(Item item) {
        items.add(item);
        incrementOperationCount();
    }
}
```

1.1.10 继承映射策略

通过元数据来指定类层次的映射。

当映射一个类或类层次到数据库的 Schema 时，有以下三个基本策略：

- 每一个类层次对应一个数据库表。（父类和子类共用一个数据库表）
- 每一个具体的实体类对应一个数据库表。
- 子类中的字段映射到一个单独的数据库表，父类的公共字段映射成一个单独的数据库表，在实例化子类时把父类的表 jion 过来。

规范要求实现支持第一个策略和第三个策略。

在这个版本中，每个具体的类对应一个表策略是可选的，在后续的版本中将会要求支持其他的策略。

在这个版本中对继承映射策略的组合支持不做要求。

1.1.10.1 每个类层次对应一个表

在这个策略中，类层次内的所有类映射到同一个表内。表内有一个列作为“区分列”，也就说，这个区分类用于标识当前行是属于哪个子类。

优点：对实体间的多态关系和跨类层次的查询支持的比较好。

缺点：要求子类的列必须是可空的。

1.1.10.2 每个具体类对应一个表

这个策略是每个类映射成一个单独的数据库表。类的所有属性，包括从父类继承的属性，都映射成这个数据库表中的列。

这个策略的缺点是：

- 不能支持多态关系。
- 当跨类层次查询时，需要 SQL UNION 查询（或者每个子类都要有一个单独的 SQL 查询）。

1.1.10.3 关联子类策略

在这个策略中，类层次的根类对应一个单独的表。每个子类对应一个自己的表，这些表中只包含子类的字段和它的主键列。子类表的主键列作为指向超类表主键的外键。

优点：对实体间的多态关系支持的比较好。

缺点：当实例化一个子类时需要多个表间关联。当类的层次较深时，这可能造成很大的性能问题。跨类层次的查询也要求关联。

2 实体操作

本章介绍如何用 EntityManager API 来管理实体实例的生命周期和使用 Query API 来查询和获取实体和它的持久化状态。

2.1 EntityManager

EntityManager 关联一个持久化上下文。持久化上下文是一个实体实例的集合，在这个集合中任何持久化实体标识只有唯一一个实体实例。在持久化上下文中管理实体实例和它们的生命周期。EntityManager 接口定义了用于和持久化上下文进行交互的方法。EntityManager API 用于创建和清除持久化实体实例，根据实体的主键标识查找实体，以及进行跨实体查询。

实体集合由持久化单元定义的给定的 EntityManager 实例管理。持久化单元定义了与当前应用相关的或由应用分组的所有类，在持久化单元内的类需要映射到同一个数据库中。

在本节中定义了 EntityManager 的接口。实体的生命周期在 2.2 节中描述。EntityManager 和持久化单元之间的关系在第 2.3 节描述，进一步的详细信息在第 4 章中。2.3 节描述实体监听器和生命周期回调方法。2.6 节描述 Query 接口。在第 5 章中描述持久化单元。

2.1.1 EntityManager 接口

```
package javax.persistence;

/**
 * Interface used to interact with the persistence context.
 */
public interface EntityManager {

    /**
     * Make an instance managed and persistent.
     * @param entity
     * @throws EntityExistsException if the entity already exists.
     * (The EntityExistsException may be thrown when the persist
     * operation is invoked, or the EntityExistsException or
     * another PersistenceException may be thrown at flush or
     * commit time.)
     * @throws IllegalArgumentException if not an entity
     * @throws TransactionRequiredException if invoked on a
     * container-managed entity manager of type
     * PersistenceContextType.TRANSACTION and there is
     * no transaction.
     */
    public void persist(Object entity);

    /**
     * Merge the state of the given entity into the
     * current persistence context.
     * @param entity
     * @return the instance that the state was merged to
     * @throws IllegalArgumentException if instance is not an
     *                 entity or is a removed entity
     * @throws TransactionRequiredException if invoked on a
     * container-managed entity manager of type
     * PersistenceContextType.TRANSACTION and there is
     * no transaction.
     */
    public <T> T merge(T entity);

    /**
     * Remove the entity instance.
     * @param entity
     * @throws IllegalArgumentException if not an entity
     *                 or if a detached entity
     * @throws TransactionRequiredException if invoked on a
     * container-managed entity manager of type
     * PersistenceContextType.TRANSACTION and there is
     * no transaction.
     */
    public void remove(Object entity);

    /**
     * Find by primary key.
     * @param entityClass
     * @param primaryKey
     * @return the found entity instance or null
     *                if the entity does not exist
     * @throws IllegalArgumentException if the first argument does
     *                    not denote an entity type or the second
```

```
     *                      argument is not a valid type for that
     *                      entity's primary key
     */
    public <T> T find(Class<T> entityClass, Object primaryKey);


    /**
     * Get an instance, whose state may be lazily fetched.
     * If the requested instance does not exist in the database,
     * the EntityNotFoundException is thrown when the instance
     * state is first accessed. (The persistence provider runtime is
     * permitted to throw the EntityNotFoundException when
     * getReference is called.)
     * The application should not expect that the instance state will
     * be available upon detachment, unless it was accessed by the
     * application while the entity manager was open.
     * @param entityClass
     * @param primaryKey
     * @return the found entity instance
     * @throws IllegalArgumentException if the first argument does
     *                      not denote an entity type or the second
     *                      argument is not a valid type for that
     *                      entity's primary key
     * @throws EntityNotFoundException if the entity state
     *                      cannot be accessed
     */
    public <T> T getReference(Class<T> entityClass, Object prima-
ey);

    /**
     * Synchronize the persistence context to the
     * underlying database.
     * @throws TransactionRequiredException if there is
     *                  no transaction
     * @throws PersistenceException if the flush fails
     */
    public void flush();

    /**
     * Set the flush mode that applies to all objects contained
     * in the persistence context.
     * @param flushMode
     */
    public void setFlushMode(FlushModeType flushMode);

    /**
     * Get the flush mode that applies to all objects contained
     * in the persistence context.
     * @return flushMode
     */
    public FlushModeType getFlushMode();

    /**
     * Set the lock mode for an entity object contained
     * in the persistence context.
     * @param entity
     * @param lockMode
     * @throws PersistenceException if an unsupported lock call
     *             is made
```

```
     * @throws IllegalArgumentException if the instance is not
     *           an entity or is a detached entity
     * @throws TransactionRequiredException if there is no
     *           transaction
     */
    public void lock(Object entity, LockModeType lockMode);

    /**
     * Refresh the state of the instance from the database,
     * overwriting changes made to the entity, if any.
     * @param entity
     * @throws IllegalArgumentException if not an entity
     *                  or entity is not managed
     * @throws TransactionRequiredException if invoked on a
     * container-managed entity manager of type
     * PersistenceContextType.TRANSACTION and there is
     * no transaction.
     * @throws EntityNotFoundException if the entity no longer
     *                  exists in the database
     */
    public void refresh(Object entity);

    /**
     * Clear the persistence context, causing all managed
     * entities to become detached. Changes made to entities that
     * have not been flushed to the database will not be
     * persisted.
     */
    public void clear();

    /**
     * Check if the instance belongs to the current persistence
     * context.
     * @param entity
     * @return
     * @throws IllegalArgumentException if not an entity
     */
    public boolean contains(Object entity);

    /**
     * Create an instance of Query for executing a
     * Java Persistence query language statement.
     * @param qlString a Java Persistence query string
     * @return the new query instance
     * @throws IllegalArgumentException if query string is not valid
     */
    public Query createQuery(String qlString);

    /**
     * Create an instance of Query for executing a
     * named query (in the Java Persistence query language
     * or in native SQL).
     * @param name the name of a query defined in metadata
     * @return the new query instance
     * @throws IllegalArgumentException if a query has not been
     *       defined with the given name
     */
    public Query createNamedQuery(String name);
```

```
    /**
     * Create an instance of Query for executing
     * a native SQL statement, e.g., for update or delete.
     * @param sqlString a native SQL query string
     * @return the new query instance
     */
    public Query createNativeQuery(String sqlString);

    /**
     * Create an instance of Query for executing
     * a native SQL query.
     * @param sqlString a native SQL query string
     * @param resultClass the class of the resulting instance(s)
     * @return the new query instance
     */
    public Query createNativeQuery(String sqlString, Class result-
Class);

    /**
     * Create an instance of Query for executing
     * a native SQL query.
     * @param sqlString a native SQL query string
     * @param resultSetMapping the name of the result set mapping
     * @return the new query instance
     */
    public Query createNativeQuery(String sqlString, String result-
SetMapping);

    /**
     * Indicate to the EntityManager that a JTA transaction is
     * active. This method should be called on a JTA application
     * managed EntityManager that was created outside the scope
     * of the active transaction to associate it with the current
     * JTA transaction.
     * @throws TransactionRequiredException if there is
     * no transaction.
     */
    public void joinTransaction();

    /**
     * Return the underlying provider object for the EntityManager,
     * if available. The result of this method is implementation
     * specific.
     * /
    public Object getDelegate();

    /**
     * Close an application-managed EntityManager.
     * After the close method has been invoked, all methods
     * on the EntityManager instance and any Query objects obtained
     * from it will throw the IllegalStateException except
     * for getTransaction and isOpen (which will return false).
     * If this method is called when the EntityManager is
     * associated with an active transaction, the persistence
     * context remains managed until the transaction completes.
     * @throws IllegalStateException if the EntityManager
     * is container-managed.
     */
    public void close();
```

```
    /**
     * Determine whether the EntityManager is open.
     * @return true until the EntityManager has been closed.
     */
    public boolean isOpen();

    /**
     * Return the resource-level transaction object.
     * The EntityTransaction instance may be used serially to
     * begin and commit multiple transactions.
     * @return EntityTransaction instance
     * @throws IllegalStateException if invoked on a JTA
     *         EntityManager.
     */
    public EntityTransaction getTransaction();

}
```

当使用带有事务范围的持久化上下文的实体管理器时，Perist，merge，remove，flush 和 refresh 方法必须在事务上下文中调用。如果没有事务上下文，则抛出 javax.persistence.TransactionRequiredException。

Find 和 gerReference 方法不要求在事务上下文中调用。如果使用带有事务范围的持久化上下文的实体管理器，结果实体将是脱管的；如果使用带有扩展的持久化上下文的实体管理器，那么结果实体将受管理。参加 2.3 节在事务外使用实体管理器。

当实体管理器打开时，从实体管理器得到的 Query 和 EntityTransaction 对象是有效的。

如果传入 createQuery 方法的参数不是有效的 Java 持久化查询语句，那么抛出 IllegalArgumentException 异常，或者执行查询失败。如果本地查询不是使用的数据库的有效查询或者结果集和查询的结果不匹配，那么查询执行失败并且在执行查询的时候抛出 PersistenceException。PersistenceException 应当封装后台的数据库异常。

如果 EntityManager 接口的方法抛出运行时异常，则引起当前事务回滚。

Close、isOpen、joinTransaction 和 getTransaction 用于管理应用管理的实体管理器和它的生命周期。可以参考 4.2.2 章节“获取应用管理的实体管理器”。

2.1.2 使用 EntityManager API 的例子

```
@Stateless public class OrderEntryBean implements OrderEntry {

  @PersistenceContext EntityManager em;

  public void enterOrder(int custID, Order newOrder) {
     Customer cust = em.find(Customer.class, custID);
     cust.getOrders().add(newOrder);
     newOrder.setCustomer(cust);
   }
}
```

2.2 实体实例的生命周期

这章描述 EntityManager 用于管理实体实例生命周期的方法。实体实例可以是 new、managed、detached、或者 removed。

- 新实体实例没有持久化唯一标识，还没有和持久化上下文建立关联。
- 受管理的实体实例具有持久化标识，和一个持久化上下文建立了关联。
- 脱管的实体实例具有持久化标识，但没有或不再和持久化上下文有关联。
- 移除的实体实例具有持久化标识，和持久化上下文有关联，它将被从数据库中删除。

下面的章节描述了生命周期操作对实体产生的影响。cascade 注解元素用于迁移这些影响到相关的实体上。Cascade 典型的应用在父子关系中。

2.2.1 持久化实体实例

通过调用 pesist 方法或者通过层级产生的持久化可以使新实体变成受管理的和持久化的。

下面的描述就是持久化操作语义应用到实体 X 的效果：

- 如果 X 是一个新实体，那么他变成受管理的。实体 X 将在事务提交之前/时或执行 flush 操作后保存到数据库中。
- 如果 X 是已经存在的受管理的实体，那么它将被持久化操作忽略。然而，如果 X 到它引用的实体间的关系被注解为 cascade=PERSIST 或 cascade=ALL，那么持久化操作将会传递到 X 引用的实体。在 XML 配

置文件中同样配置也是可以的。

- 如果 X 是一个 removed 实体，它变成受管理的实体。
- 如果 X 是脱管的，那么执行持久化操作就会抛出 EntityExistsException，或者在 flush 或提交时抛出 EntityExistsException 或 PersistenceException。
- 对于 X 引用的所有实体 Y，如果指向 Y 的关系（字段）被注解为 cascade=PERSIST 或 cascade=ALL，那么持久化操作应用到 Y。

2.2.2 Removal

当调用 remove 方法或被 remove 方法层级到受管理的实体时，受管理的实例变成可删除的。

Remove 操作应用到实体 X 上的语义如下所述：

- 如果 X 是新实体，那么 remove 操作将忽略它。然而，如果 X 到它引用的实体间的关系被注解为 cascade=REMOVE 或 cascade=ALL，那么 remove 操作将会传递到 X 引用的实体。在 XML 配置文件中同样配置也是可以的。
- 如果 X 是受管理的，那么 remove 操作将使它变成可删除的。如果 X 到它引用的实体间的关系被注解为 cascade=REMOVE 或 cascade=ALL，那么 remove 操作将会传递到 X 引用的实体。
- 如果 X 是脱管的，那么执行 remove 操作就会抛出 IllegalArgumentException（或者事务提交失败）。
- 如果 X 是可删除实体，那么它被 remove 操作忽略。
- 可删除实体 X 将在事务提交时或在 flush 操作执行后从数据库中删除。

在实体被删除后，它的状态（除了生成的状态）将是 remove 操作被调用时刻的状态。

2.2.3 数据库同步

持久化实体的状态在事务提交时被同步到数据库中。同步会引起对持久化实

体及其关联的实体的更新持久化到数据库中。

对实体状态的更新既包括新值赋予实体的持久化属性或字段，也包括持久化属性或字段值的改变。

与数据库同步不会刷新受管理实体，除非显式调用实体的 refresh 方法。

受管理实体间的双向关系将根据关系的持有者所持有的引用进行持久化。开发者有责任保证在实体发生改变时内存中指向关系拥有者的引用和指向反向边的引用保持一致。在单向一对一和一对多关系的情况下，开发者要保证关系的语义是恒定的（如果唯一约束（例如在 1.1.8.3.1 和 1.1.8.5.1 章节中描述的缺省映射）没有应用到 O/R 映射的定义上就会产生问题）。

保证关系的反向边改变时，关系的拥有者边也作相应改变是非常重要的，以便保证这些改变在被同步到数据库时不会丢失。

当事务是活动时，持久化提供商运行时也可以在其他时间执行数据库同步操作。Flush 方法可以用于强制同步，它应用到那些和持久化上下文建立关联的实体上。EntityManager 和 Query 的 setFlushMode 方法用于控制同步语义。FlushModeType.AUTO 在 2.6.2 章节中定义。如果指定为 FlushModeType.COMMIT，那么在事务提交时执行 flush；持久化供应商可以但不要求在其他时间执行 flush。如果没有活动事务，那么持久化供应商不必执行 flush。

Flush 操作应用到实体 X 上的语义如下所述：

- 如果 X 是受管理实体，它被同步到数据库。
  - 对于 X 引用的所有实体 Y，如果指向 Y 的关系被注解为 cascade=PERSIST 或 cascade=ALL，则持久化也应用到 Y。
  - 对于 X 引用的所有实体 Y，如果指向 Y 的关系没有被注解为 cascade=PERSIST 或 cascade=ALL，那么：
    - 如果 Y 是 new 或者 removed，则执行 flush 操作时抛出 IllegalStateExcpetion 异常，并且回滚事务，或者事务提交失败。
    - 如果 Y 是脱管的，同步的语义依赖于关系的拥有者。如果 X 拥有关系，对关系的任何改变都会同步到数据库；否则，如果 Y 是关系的拥有者，如何处理没有规定。

- 如果 X 是将删除的实体，那么它将从数据库中删除。不需要层级设置。

2.2.4 脱管实体

如果使用事务范围的容器管理的实体管理器（参见 2.3），那么提交事务、回滚事务、清除持久化上下文、关闭实体管理器、以及序列化实体或按值传递一个实体（如在分布式应用中，通过远程接口传递等）都会产生脱管实体。

脱管实体实例一直存在于持久化上下文的外部，它们的状态不再保证和数据库一致。

在持久化上下文结束后，应用可以获取可以获取的脱管实体实例的可读取状态。可读取的状态包括：

- 标志不为 fetch=LAZY 的任何可持久化字段或属性。
- 任何可以被应用获取的字段或属性。

如果持久化字段或属性是关系字段，那么只有关联实例是可以获取时，才可以安全地获取关联的实例的状态。可获取的实例包括：

- 任何用 find()方法获取的实体实例。
- 任何用查询获取的或显式地在 FETCH JOIN 语句中要求的实体实例。
- 任何可被应用获取的实体实例，这些实体实例的变量持有非主键的持久化状态。
- 任何可以通过标记为 fetch=EAGER（即非 LAZY 方式）的迁移关系从其他实体实例间接获取的实体实例。

2.2.4.1合并（Merge）脱管实体状态

Merge 操作使得脱管实体转变成被 EntityManager 管理的持久化实体。

Merge 操作的语义用实体 X 说明如下：

- 如果 X 是脱管实体，那么它被复制到一个和它具有相同标识的预先存在的受管理实体实例 X’，或者创建一个新的 X 的复制品。
- 如果 X 是新实体实例，那么会创建一个新的受管理的实体实例 X’，并

且 X 的状态会被复制到 X’。

- 如果 X 是一个被删除实体实例，在 merge 操作中将抛出 IllegalArgumentException（或者事务提交失败）。
- 如果 X 是受管理实体，merge 操作将忽略它，但是，merge 操作将被层级传递到 X 引用的标志有 cascade=MERGE 或 cascade=ALL 实体上。
- 对所有 X 引用的标志为 cascade=MERGE 或 cascade=ALL 的实体，Y 将被迭代成 Y’。对 X 引用的这些 Y，X’将引用相应的 Y’。（注意，如果 X 是受管理的，那么 X 和 X’是同一个对象）。
- 如果 X 引用的所有 Y 没有标志为 cascade=MERGE 或 cascade=ALL，那么 X’会产生一个对受管理实体 Y’的引用，这个 Y’与 Y 有相同的标识。

持久化提供商不必 merge 标记为 LAZY 且还没有被读取的字段，在 merge 时，必须忽略这些字段。

持久化运行时实现在 merge 时或/和在 flush 或在提交时，必须检查任何被实体使用的 Version 列。没有 Version 列时，在 merge 时不需要执行检查。

2.2.4.2脱管实体和懒惰加载

当使用懒惰的属性或字段和/或关系时，不同的提供商序列化实体和 merge 实体到持久化上下文可以不同。

供应商需要支持在分离的 JVM 间对脱管实体实例的序列化和反序列化，以及 merge（这些实例可以包含还没有加载的懒惰的属性或字段和/或关系），在这些 JVM 中，运行时实例既可以获取实体类，也可以获取持久化实现的类。

当多个供应商之间协作时，应用不要使用懒惰加载。

2.2.5 管理实体实例

应用应当确保实例只在一个持久化上下文中。没有规定同一个 Java 实例在多个持久化上下文中如何处理。

Contains()方法用于确定一个实体实例是否被当前的持久化上下文管理。

在下列情况下，Cotains 方法返回 true：

- 如果实体已经从数据库中取出，但还没有被删除或脱管。
- 如果实体实例是新的，并且已经调用 persist 方法，或者持久化操作层级到这个实体。

在下列情况下，contains 方法返回 false：

- 如果实例已经脱管。
- 如果已经调用 remove 方法，或者 remove 操作层级到这个实体上。
- 如果实体是新的，但 persist 方法还没有被调用，或者 persist 方法还没有层级到这个实体上。

注意：contains 方法会立即知道 perist 或 remove 方法的层级影响，但是对数据库的真正的插入、删除在事务的最后才真正产生。

2.3 持久化上下文的生命周期

容器管理的持久化上下文的生命周期既可以是在事务的范围内（事务范围的持久化上下文），也可以超出一个单个事务的范围（扩展的持久化上下文）。PersistenceContextType 为容器管理的实体管理器枚举了持久化上下文的生命周期范围。当 EntityManager 被创建时（显式地，或通过注入，或通过 JNDI）定义持久化上下文的生命周期范围。参见 4.6 章节。

```
public enum PersistenceContextType {
  TRANSACTION,
  EXTENDED
}
```

缺省情况下，容器管理的实体管理器的持久化上下文的生命周期和事务的范围一致，即它的类型是 PersistenceContextType.TRANSACTION。

当使用扩展的持久化上下文时，扩展的持久化上下文的生命周期从 EntityManager 实例被创建开始直到 EntityManager 被关闭。这个持久化上下文可以跨越 EntityManager 的多个事务和非事务调用。当 EntityManager 在事务的范围内被调用时或当绑定了扩展的持久化上下文的有状态会话 bean 在事务的范围内被调用时，当前事务征用扩展的持久化上下文。

具有扩展的持久化上下文的 EntityManager 在事务提交后维护对实体对象的引用。这些对象仍然被 EntityManager 管理，并且它们可以在事务之间作为受管理对象被更新（注意，当开始新事务时，在扩展的持久化上下文中的受管理对象不会从数据库中被再次加载）。来自在扩展的持久化上下文中的受管理对象的迁移会产生一个或多个其他受管理对象，而不管事务是否是活动的。

当使用具有扩展的持久化上下文的 EntityManager 时，persist、remove、merge 和 refresh 操作都可以被调用而不管事务是否是活动的。当扩展的持久化上下文在事务中被征用且事务提交时，这些操作的影响才被提交。

应用管理的实体管理器的持久化上下文的范围是可扩展的。应用负责管理这个持久化上下文的生命周期。

扩展的持久化上下文在 4.7 章节中进一步描述。

2.3.1 事务提交

当事务提交时，在事务范围的持久化上下文中受管理实体变成脱管的；在扩展的持久化上下文中的受管理实体仍然受管理。

2.3.2 事务回滚

对事务范围的持久化上下文和扩展的持久化上下文，事务回滚引起所有已经存在的受管理实例和被删除的实例（这些实例在事务开始时被持久化到数据库中）变成脱管的。实例的状态将是事务回滚点的状态。典型地，事务回滚引起持久化上下文在回滚点处于不一致状态。特殊情况下，版本属性的状态和生成的状态（例如，生成的主键）可以是不一致的。因此，可以用和其他的脱管实体对象使用的方式一样的方式不重用以前被持久化上下文管理的实例（包括在事务中被持久化的新实例）——例如，不能传递到 merge 操作时（没有规定在事务回滚后那些没有被持久化到数据库的实例是作为新实例还是作为脱管实例。这由实现决定）。

2.4 乐观锁和并发

规范假定使用乐观锁。这假定实现将使用 read-committed 隔离（或提供商提供的隔离，在这个隔离中不使用长时间读取锁）来获取持久化单元映射的数据库，并且典型地只在调用 flush 方法时回写到数据库——通过 flush 模式来确定应用或持久化提供上运行时是否显式调用 flush 方法。如果事务是活动的，符合本规范的实现可以立即回写到数据库（例如，更新、创建和/或删除受管理实体），然而，实现是否通过配置来决定这种立即写数据库不在本规范的范围。配置设定乐观锁模式在 2.4.3 章节中描述。实际上，使用乐观锁的应用可以要求数据库隔离级别高于 read-committed。然而设定数据库隔离级别不在本规范范围之中。

2.4.1 乐观锁

乐观锁是一种用于保证更新到数据库的数据和实体的状态一致的一种技术。这种技术只有当没有并发事务更新实体状态的数据时（由于实体状态只被读取）使用。这保证了更新和删除的数据和数据库目前的状态一致，并且保证不会丢失并发更新。引起违反这样的约束的事务将抛出 OptimistaicLockException，并且引起事务回滚。

希望使用乐观锁的可移植应用必须为实体指定 Version 属性——例如，用 Version 注解符注解持久化属性或字段或在 XML 配置文件中将属性指定为 version 属性。强烈建议应用为那些可能被从断链的状态并发读取或合并的所有实体使用乐观锁。使用乐观锁失败可能导致实体状态不一致，丢失更新和其他状态混乱。如果乐观锁没有作为实体状态的一部分，那么应用必须承担维护数据一致性的责任。

2.4.2 版本属性

持久化提供商用 Version 字段或属性执行乐观锁。持久化提供商在执行实体实例生命周期操作的过程中获取或设定 Version 字段或属性。如果实体有一个映射为 Version 的属性或字段，那么实体自动地可以使用乐观锁。

实体可以获取版本属性或字段的状态，也可以为应用暴露获取版本的方法，但是不能更改版本的值（然而，批更新语句可以设置版本属性的值。参见 3.10）。只有持久化提供商可以设置或更新版本属性的值。

当实体被写到数据库时，持久化提供商运行时更新版本属性。所有的非关系字段和属性，以及实体拥有的所有关系都包括在版本检查中。

当实体被合并时，merge 操作的实现必须检查版本属性，如果发现要被 merge 的对象是实体的过时拷贝，那么抛出 OptimisticLockException——例如，由于实体变成脱管实体，这个实体已经被更新。依赖实现使用的策略，可能在 flush 被调用时或提交时才抛出异常，不管哪一个先发生。

当执行乐观锁检查时，要求持久化提供商只使用版本属性。持久化提供商可以还可以提供额外的机制。然而，实现额外的机制不在本规范规定之内（这些额外机制可能在这个规范的未来版本中标准化）。

如果只有一部分实体包含版本属性，要求持久化提供商检查指定了版本属性的实体。但不保证对象图的一致性，没有指定版本属性的实体不会在完成之前停止操作。

2.4.3 锁模式

除了上面描述的语义外，通过 EntityManager 的 lock 方法可以进一步指定锁模式。

定义了两个锁模式：READ 和 WRITE：

```
public enum LockModeType
{
    READ,
    WRITE
}
```

请求锁类型 LockModeType.READ 和 LockModeType.WRITE 的语义如下：

如果事务 T1 在版本对象上调用 lock(entity, LockModeType.READ)，那么实体管理器必须保证不要发生下面的现象：

- P1(脏读)：事务 T1 更改一行。在 T1 提交或回滚之前，另一个事务 T2 读这一行并且获得更改后的值。事务 T2 最终成功提交，不管 T1 是否提

交或回滚，也不管是在 T2 提交之前还是之后提交或回滚。

- P2（非重复读）：事务 T1 读取一行。在 T1 提交之前，事务 T2 更改或删除了那一行。两个事务最终都提交成功。

这通常由实体管理器完成在后台数据库行上加锁。这样的锁可以被立刻得到（一直保持到提交完成），或者可以在提交时获得这样的锁（即使这样，也必须保持到提交完成）。只要能避免上述现象发生，任何支持重复读的实现都是允许的。

不要求持久化实现支持对非版本对象调用 lock(entity, LockModeType.READ)。当实现不支持这种调用时，它必须抛出 PeristenceException。当支持这种调用时，不管是否是版本或非版本对象，LockModeType.READ 必须总是阻止 P1 和 P2 现象的发生。对非版本对象调用 lock(entity, LockModeType.READ)的应用将是不可移植的。

如果事务 T1 在版本对象上调用 lock(entity, LockModeType.WRITE)，实体管理器必须避免现象 P1 和 P2 发生（对 LockModeType.READ 同样）和也必须强迫对实体的版本列进行增量更新。强迫的版本更新可以立即执行，也可以延迟到 flush 或 commit 时。如果实体在延迟的版本更新被应用之前被删除，则由于后台的数据库行不再存在，因此强迫的版本更新被忽略。

不要求持久化实现支持在非版本对象上调用 lock(entity, LockModeType.WRITE)。当实现不支持这种调用时，它必须抛出 PeristenceException。当支持这种调用时，不管是否是版本或非版本对象，LockModeType.WRITE 必须总是阻止 P1 和 P2 现象的发生。对非版本对象来说，LockModeType.WRITE 是否有附加的行为由提供商决定。对非版本对象调用 lock(entity, LockModeType.READ)的应用将是不可移植的。

对版本对象，允许实现在请求 LockModeType.READ 的地方使用 LockModeType.WRITE，但不能反过来。

否则，如果版本对象被更新或删除，那么实现必须保证满足 LockModeType.WRITE 的要求，即使没有显式调用 EntityManager.lock。

对可移植应用来讲，应用不应当依赖特定供应商的配置来通过其他方式保证

（非 EntityManager.lock）重复读。然而，应当注意：如果实现在前面已经获得数据库行上的悲观锁，那么实现可以忽略也可以不忽略在代表这些行的实体对象上的 lock(entity, LockModeType.READ)调用。

2.4.4 OptimisticLockException

当用 flush 模式来设置一致性时，实现可以延迟写数据库到事务结束时。在这种情况下，乐观锁检查可以一直到提交时发生，并且可以在提交完成之前的阶段内抛出 OptimisticLockException。如果应用必须捕捉或处理 OptimisticLockException，那么应用必须应当使用 flush 模式来迫使发生写数据库。这样，应用就可以捕捉和处理乐观锁异常。

OptimisticLockException 提供了一个返回引起异常的对象的 API。不能保证每次抛出异常时都呈现这个对象引用，但无论什么时候持久化提供商都能提供它。应用不能依赖这个可以获得的对象。

在一些情况下，OptimisticLockException 可以被抛出，并且被其他异常封装，例如在穿越 VM 边界时，封装为 RemoteException。在封装的异常内引用的实体应当是可序列化的。

OptimisticLockException 总是引起事务回滚。

在新事务内刷新或重新加载对象然后再使用这个事务可能会引起 OptimisticLockException。

2.5 实体监听器和回调方法

一个方法可以被声明成回调方法，以便它可以接收实体寿命周期事件的通知。

生命周期回调方法可以定义在实体类、被映射的超类或者和实体/被映射超类关联的实体监听器上。实体监听器类的方法用于响应实体的生命周期事件。在实体类或被映射超类上可以定义多个实体监听器类。

缺省的实体监听器（应用到持久化单元内的所有实体上的实体监听器）可以用 XML 配置文件指定。

用元语注解符或 XML 配置符定义生命周期回调方法和实体监听器类。当使用注解符时，可以在实体类或被映射超类上用 EntityListeners 注解符定义一到多个实体监听器类。如果指定多个实体监听器类，他们的调用顺序由它们在 EntityListeners 中的顺序决定。XML 配置符是指定实体监听器类调用顺序的可选方案或用于覆盖注解符内指定的顺序。

在实体类、被映射的超类或监听器类上可以指定任何注解符的子集或组合。对同一个生命周期事件，一个单独的类不可以有多个生命周期回调方法。但同一个方法可以用于多个回调事件。

可以在类层次上的多个实体类和被映射超类上直接定义监听器类和/或生命周期方法。2.5.4 章节描述了在这种情况下方法调用顺序的规则。

实体监听器类必须有一个 public 的无参构造器。

实体监听器类是无状态的。实体监听器类的生命周期没有规定。

回调需遵循以下规则：

- 回调方法可以抛出 unchecked/运行时异常。在事务内执行的回调方法抛出的运行时异常引起事务回滚。
- 回调可以调用 JNDI、JDBC、JMS 和企业 Bean。
- 通常情况下，可移植应用不应当在生命周期回调方法内调用 EntityManager 或 Query 操作、获取其他实体实例或更改关系。（这些操作的语义在规范未来的版本中可能被标准化）

当在 Java EE 环境内调用时，回调监听器共享将要调用的组件的企业命名上下文，并且回调方法在将要调用的组件的事务和安全上下文中被调用。（例如，如果事务属性是 RequiredsNew 的会话 bean 的业务方法正常终止引起的事务提交，那么将在组件的命名上下文中、事务上下文和安全上下文中执行 PostPersist 和 PostRemove 回调方法）

2.5.1 生命周期回调方法

实体生命周期回调方法可以定义在实体监听器类上，也可以直接定义在实体类或被映射超类上。

生命周期回调方法用注解符指派调用它们的回调事件或用 XML 配置符指派它们对应的回调事件。

用于实体类或被映射超类的回调方法的注解符和用于实体监听器类中的回调方法的注解符是一样的。然而私有方法的符号是不同的。

在实体或被映射超类内定义的回调方法形式如下：

```
public void <METHOD>()
```

在监听器内定义的回调方法形式为：

```
public void <METHOD>(Object)
```

Object 参数是回调方法被调用的实体实例，它可以是被声明为真正的实体类型。

回调方法的可见性可以是 public，private，protected 或包级可见，但不应当是 static 或 final。

下面的注解符用于指派对应生命周期事件的回调方法。

- PerPersist
- PostPersit
- PreRemove
- PostRemove
- PreUpdate
- PostUpdate
- PostLoad

2.5.2 实体的生命周期回调方法的语义

PrePersist 和 PreRemove 回调方法在 EntityManger 的 persist 和 remove 方法被执行之前调用。对那些已经执行 merge 操作并且产生新的受管理实例的实体来说，PrePersist 回调方法会在实体的状态复制到受管理实例之后被调用。PrePersist 和 PreRemove 方法也会在所有被层级到的实体上调用。PrePersist 和 PreRemove 方法总是作为 persist、merge 和 remove 操作的一部分被同步调用。

PostPersist 和 PostRemove 回调方法在 EntityManger 的 persist 和 remove 方法被执行之后调用。这些方法也会在所有被层级到的实体上调用。PostPersist 和 PostRemove 方法将在数据库的插入和删除操作被执行之后被调用。可以在 persist、merge 或 remove 操作执行之后直接调用上述的数据库操作，也可以在 flush 操作被调用后直接调用（这可能在事务结束时）。可以在 PostPersist 方法内获得生成的主键值。

PreUpdate 和 PostUpdate 回调方法在数据库更新操作之前和之后调用。调用发生在实体状态被更新到数据库时或状态被 flush 到数据库时（在事务的结束时刻）。

注意：当实体在一个事务内先持久化，接着被更新时，或者在一个事务内先被更改，然后被删除时，是否发生 PreUpdate 和 PostUpdate 回调由具体的实现决定。可移植的应用不应该依赖这些行为。

PostLoad 方法在实体从数据库中加载到当前持久化上下文中时，或者在 refresh 操作执行之后被调用。PostLoad 方法在查询结果返回或获取之前，或者在关系被层级加载之前调用。

是否在生命周期事件层级的前面或后面调用回调方法依赖于具体的实现。应用不应当依赖于这个顺序。

2.5.3 举例

```
@Entity
@EntityListeners(com.acme.AlertMonitor.class)
public class Account {

    Long accountId;
    Integer balance;
    boolean preferred;

    @Id
    public Long getAccountId() { ... }
    ...
    public Integer getBalance() { ... }
    ...
    @Transient // because status depends upon non-persistent context
    public boolean isPreferred() { ... }
    ...

    public void deposit(Integer amount) { ... }
    public Integer withdraw(Integer amount) throws NSFException { ... }

    @PrePersist
    protected void validateCreate() {
        if (getBalance() < MIN_REQUIRED_BALANCE)
        throw new AccountException("Insufficient balance to open an
account");
    }

    @PostLoad
    protected void adjustPreferredStatus() {
        preferred =
            (getBalance() >= AccountManager.getPreferredStatu-
sLevel());
    }
}

public class AlertMonitor {

    @PostPersist
    public void newAccountAlert(Account acct) {
        Alerts.sendMarketingInfo(acct.getAccountId(), acct.getBal-
ance());
    }
}
```

2.5.4 为一个实体生命周期事件定义多个生命周期回调方法

如果一个实体生命周期事件上定义多个生命周期回调方法，那么这些方法的调用顺序如下。

任何情况下，缺省的监听器是第一个被调用，顺序按照在 XML 描述文件中定义的顺序。除非显式的通过 ExcludeDefaultListeners 注释符或 exclude-default-listeners 元素排除缺省监听器，缺省的监听器都将应用到持久化单元的所有实体上。

定义在实体监听器类或被映射的超类上的生命周期回调方法的调用顺序和

在 EntityListeners 注释符中指定的实体监听器类的顺序一致。

如果在继承层次中的多个类（实体类和/或被映射的超类）都定义了实体监听器，那么定义在超类上的实体监听器类先于定义在子类上的实体监听器类。ExcludeSuperclassListeners 注释符或 exclude-superclass-listeners 元素用于排除定义在超类或被映射的超类上的实体监听器。被上述元素排除掉的监听器将不会包含在实体类及其子类中（注：被排除的监听器可以通过显式的在 EntityListeners 注释符或 entity-listeners 元素中指定来被重新引入）。ExcludeSuperclassListeners 注释符或 exclude-superclass-listeners 元素不会排除将缺省的实体监听器。

如果对同一个生命周期事件的一个生命周期回调方法也被指定到实体类和/或一个或多个实体或被映射的超类上，那么在实体类和/或超类上指定的回调方法在其他回调方法之后被调用，且超类的首先被调用。一个类可以重载父类的相同类型的回调方法，且在这种情况下，重载的方法不会被调用（注：注意如果一个方法重载了父类的回调方法，但指定了不同的生命周期事件或不是一个生命周期回调方法，那么这个重载的方法将被调用）。

持久化提供者按照指定的顺序调用回调方法。如果前一个回调方法正常执行，那么持久化提供者执行下一个回调方法。

XML 描述符可以用于覆盖在注释符中指定的生命周期回调方法的执行顺序。

2.5.5 举例

为动物定义了几个实体类和监听器：

```
@Entity
public class Animal {
    ....
    @PostPersist
    protected void postPersistAnimal() {
        ....
    }
}

@Entity
@EntityListeners(PetListener.class)
public class Pet extends Animal {
    ....
}

@Entity
@EntityListeners({CatListener.class, CatListener2.class})
public class Cat extends Pet {
    ....
}

public class PetListener {
    @PostPersist
    protected void postPersistPetListenerMethod(Object pet) {
        ....
    }
}

public class CatListener {
    @PostPersist
    protected void postPersistCatListenerMethod(Object cat) {
        ....
    }
}

public class CatListener2 {
    @PostPersist
    protected void postPersistCatListener2Method(Object cat) {
        ....
    }
}
```

如果 PostPersist 事件发生在 Cat 实例上，则按顺序调研下面的方法：

```
postPersistPetListenerMethod
postPersistCatListenerMethod
postPersistCatListener2Method
postPersistAnimal
```

假定，SiameseCat 是 Cat 的子类：

```
@EntityListeners(SiameseCatListener.class)
@Entity
public class SiameseCat extends Cat {
  ...
  @PostPersist
  protected void postPersistSiameseCat() {
     ...
  }
}

public class SiameseCatListener {
    @PostPersist
    protected void postPersistSiameseCatListenerMethod(Object cat) {
        ....
    }
}
```

如果 PostPersist 事件发生在 SiameseCat 实例上，则按顺序执行下列方法：

```
postPersistPetListenerMethod
postPersistCatListenerMethod
postPersistCatListener2Method
postPersistSiameseCatListenerMethod
postPersistAnimal
postPersistSiameseCat
```

假定 SiameseCat 的定义改为下面的方式:

```
@EntityListeners(SiameseCatListener.class)
@Entity
public class SiameseCat extends Cat {
  ...
  @PostPersist
  protected void postPersistAnimal() {
     ...
  }
}
```

将按照下面的顺序调用下面的方法，其中 postPersistAnimal 是定义在 SiameseCat 类上的 PostPersist 方法：

```
postPersistPetListenerMethod
postPersistCatListenerMethod
postPersistCatListener2Method
postPersistSiameseCatListenerMethod
postPersistAnimal
```

2.5.6 异常

生命周期回调方法可以抛出运行时异常。在一个事务内执行的回调方法内抛出的运行时异常引起事务回滚。在抛出运行时异常后，其他的回调方法将不再被

调用。

2.5.7 在 XML 描述符中的回调监听器类和生命周期方法规范

XML 描述符是注释符的可选方案，或用于覆盖在注释符中指定的回调方法的调用顺序。

2.5.7.1回调监听器规范

Entity-listener 用于指定实体监听器类的监听器方法。通过 pre-persist，post-persist，per-remove，post-remove，pre-update，post-update 和/或 post-load 元素来指定监听器方法。

不管是用 XML 描述符还是与注释符组合使用，一个实体监听器类上最多只能有一个 pre-persist 方法，一个 post-persit 方法，一个 pre-remove 方法，一个 post-remove 方法，一个 pre-update 方法，一个 post-update 方法和/或 post-load 方法。

2.5.7.2实体监听器类绑定到实体的规范

Entity 或 mapped-superclass 元素的 Entity-listeners 子元素用于为单独的实体或被映射超类及其子类指定实体监听器类。

实体监听器绑定到实体类是递增的。绑定到实体或被映射超类的超类上的实体监听器类也应用到实体或被映射超类上。

Exclude-superclass-listener 元素指定不调用在父类上定义的监听器方法。

Exclude-default-listener 元素指定不调用的缺省监听器方法。

为一个实体或被映射超类显式列出排除的缺省或超类监听器会影响到实体或被映射超类及其子类。

在为一个生命周期事件指定多个回调方法的情况下，使用在 2.5.4 章节的调用顺序规则。

2.6 Query API

Query API 既可以用于静态查询（如，命名查询），也可以用于动态查询。Query API 致词后命名参数绑定和分页控制。

2.6.1 Query 接口

```
package javax.persistence;
import java.util.Calendar;
import java.util.Date;
import java.util.List;
/**
 * Interface used to control query execution.
 */
public interface Query {
/**
 * Execute a SELECT query and return the query results
 * as a List.
 * @return a list of the results
 * @throws IllegalStateException if called for a Java
 *Persistence query language UPDATE or DELETE statement
 */
public List getResultList();
/**
 * Execute a SELECT query that returns a single result.
 * @return the result
 * @throws NoResultException if there is no result
 * @throws NonUniqueResultException if more than one result
 * @throws IllegalStateException if called for a Java
```

```
  *Persistence query language UPDATE or DELETE statement
  */
public Object getSingleResult();
/**
  * Execute an update or delete statement.
  * @return the number of entities updated or deleted
  * @throws IllegalStateException if called for a Java
  *Persistence query language SELECT statement
  * @throws TransactionRequiredException if there is
  *                          no transaction
  */
public int executeUpdate();
/**
  * Set the maximum number of results to retrieve.
  * @param maxResult
  * @return the same query instance
  * @throws IllegalArgumentException if argument is negative
  */
public Query setMaxResults(int maxResult);
/**
  * Set the position of the first result to retrieve.
  * @param start position of the first result, numbered from 0
  * @return the same query instance
  * @throws IllegalArgumentException if argument is negative
  */
public Query setFirstResult(int startPosition);
/**
* Set an implementation-specific hint.
```

```
     * If the hint name is not recognized, it is silently ignored.
     * @param hintName
     * @param value
     * @return the same query instance
    *@throwsIllegalArgumentExceptioniftheSecondargumentisnot
     *valid for the implementation
     */
    public Query setHint(String hintName, Object value);
    /**
     * Bind an argument to a named parameter.
     * @param name the parameter name
     * @param value
     * @return the same query instance
     * @throws IllegalArgumentException if parameter name does not
     *correspond to parameter in query string
     *or argument is of incorrect type
     */
    public Query setParameter(String name, Object value);
    /**
     * Bind an instance of java.util.Date to a named parameter.
     * @param name
     * @param value
     * @param temporalType
     * @return the same query instance
     * @throws IllegalArgumentException if parameter name does not
     *correspond to parameter in query string
     */
    public Query setParameter(String name, Date value, TemporalType
```

```
temporalType);
/**
  * Bind an instance of java.util.Calendar to a named parameter.
  * @param name
  * @param value
  * @param temporalType
  * @return the same query instance
  * @throws IllegalArgumentException if parameter name does not
  *correspond to parameter in query string
  */
public Query setParameter(String name, Calendar value, Temporal-
Type temporalType);
/**
  * Bind an argument to a positional parameter.
  * @param position
  * @param value
  * @return the same query instance
  * @throws IllegalArgumentException if position does not
  *correspond to positional parameter of query
  *or argument is of incorrect type
  */
public Query setParameter(int position, Object value);
/**
  * Bind an instance of java.util.Date to a positional parameter.

* @param position
  * @param value
  * @param temporalType
```

```
  * @return the same query instance
  * @throws IllegalArgumentException if position does not
  *correspond to positional parameter of query
  */
publicQuerysetParameter(intposition,Datevalue,TemporalType
temporalType);
/**
  * Bind an instance of java.util.Calendar to a positional param-
eter.
  * @param position
  * @param value
  * @param temporalType
  * @return the same query instance
  * @throws IllegalArgumentException if position does not
  *correspond to positional parameter of query
  */
publicQuerysetParameter(intposition,Calendarvalue,Temporal-
Type temporalType);
/**
  * Set the flush mode type to be used for the query execution.
  * The flush mode type applies to the query regardless of the
  * flush mode type in use for the entity manager.
  * @param flushMode
  */
public Query setFlushMode(FlushModeType flushMode);
}
```

一个由多个 select 表达式组成 SELECT 语句的查询的结果是 Object[]。如果

SELECT 语句只有一个 select 表达式，查询结果是 Object。当使用本地 SQL 查询时，SQL 查询结果集映射（看 2.6.6 章节）决定了返回多少项（实体，标量值等）。如果返回多个项，查询结果是 Object[]。如果只返回一项或指定了结果的类，则查询结果是 Object。

如果参数的名称和查询语句内的命名参数不一致，或者如果指定的位置值和查询语句内的位置参数不一致，或者参数的类型不匹配，则抛出 IllegalArgumentException。可以在参数绑定的时候抛出异常，或者执行查询失败。

在涉及多个表连接的查询上使用 setMaxResults 或 setFirstResult 的影响没有定义。

除了 executeUpdate 方法外的查询方法不要求在一个事务上下文中调用。尤其 getResultList 和 getSingleReslut 方法不要求在一个事务上下文中调用。如果使用带事务范围持久化上下文的实体管理器，则查询的结果实体是托管的；如果使用带扩展持久化上下文的实体管理器，则查询结果实体会受到管理。参照第 4 章的在事务外部使用实体管理器和持久化上下文类型。

除了 NoResultException 和 NonUniqueResultException 以外其他的由 Query 接口的方法抛出的运行时异常都会引起当前事务回滚。

2.6.1.1举例

```
public List findWithName(String name) {
  return em.createQuery(
     "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

2.6.2 查询和 FlushMode

Flush 模式影响下列查询的结果。

当在事务内执行查询时，如果在查询对象上设置 FlushModeType.AUTO，或持久化上下文的 flush 模式被设置为 AUTO（缺省值）且 Query 对象上没有指定

flush 模式，那么持久化提供商必须保证查询操作能够知道对持久化上下文内实体的所有更新（这些更新可能影响查询结果）。持久化实现可以通过 flush 这些实体到数据库或通过别的方式实现这个目的。如果设置 FlushModeType.COMMIT，那么持久化上下文内实体的更新对查询的影响没有规定。

public enum FlushModeType {

COMMIT,

AUTO

}

如果没有活动的事务，那么持久化提供商不需要 flush 到数据库。

2.6.3 命名参数

命名参数是以“:”作为前缀的标识符。命名参数是大小写敏感的。

命名参数遵循在 3.4.1 中定义的标识符规则。命名参数可以在 EJB QL 中使用，但不能用在本地查询中。只有位置参数可以用于本地查询。

传入 Query API 的 setParameter 方法的参数名不包括前缀“:”。

2.6.4 命名查询

命名查询是一个静态查询表达式。命名查询可以用 EJB QL 语言定义，也可以用 SQL 语言定义。查询的名字在持久化单元内必须唯一。

下面是定义命名查询的例子：

```
@NamedQuery(
  name="findAllCustomersWithName",
  queryString="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

下面是使用命名查询的例子：

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
  .setParameter("custName", "Smith")
  .getResultList();
```

2.6.5 多态查询

缺省情况下，所有的查询都是多态的。也就是说查询语句内 FROM 语句不仅仅是显式指定的特定的实体类的实例，也可以是它的子类。查询返回的实例包含满足条件的子类的实例。(注：在将来的版本中考虑构造受限的查询多态机制)

例如，查询：

```
select avg(e.salary) from Employee e where e.salary > 80000
```

返回所有职员的平均薪水，包含 Employee 的子类，例如 Manager 和 Exempt。

2.6.6 SQL 查询

可以用本地 SQL 表述查询语句。本地 SQL 的查询结果可以由多个实体，分层级的值，或者两者的组合组成。查询返回的实体可以是不同的实体类型。

提供 SQL 查询是为了支持那些必须使用目标数据库本地 SQL 的情况（不能使用 EJB QL）。本地 SQL 查询不能在多个数据库间移植。

当 SQL 查询返回多个实体时，实体必须与在 SqlResultSetMapping 元语中定义的 SQL 语句的结果列一一对应。然后，持久化运行时可以通过结果集映射元语将 JDBC 结果映射到期望的对象上。可以参照第 7.3.3 章节了解 SqlResultSetMapping 元语注释符的定义和其它相关的注释符。

如果查询结果被限定到一个实体类的多个实体，那么可以使用更加简单的格式，且可以不使用 SqlResultSetMapping 元语。

下面的例子解释了用 createNativeQuery 方法动态的创建本地 SQL 查询，并且将一个指定了结果类型的实体类作为参数。

```
Query q = em.createNativeQuery(
          "SELECT o.id, o.quantity, o.item " +
          "FROM Order o, Item i " +
          "WHERE (o.item = i.id) AND (i.name = 'widget')",
        com.acme.Order.class);
```

当执行上面的语句后，将返回所有名字为“widget”的 Order 实体。也可以用 SqlResultSetMapping 获得相同的结果。

```
Query q = em.createNativeQuery(
              "SELECT o.id, o.quantity, o.item " +
              "FROM Order o, Item i " +
              "WHERE (o.item = i.id) AND (i.name = 'widget')",
              "WidgetOrderResults");
```

在这种情况下，可能要按下面的方式指定查询结果类型的元语。

```
@SqlResultSetMapping(name="WidgetOrderResults",
       entities=@EntityResult(entityClass=com.acme.Order.class))
```

下面的查询和 SqlResultSetMapping 解释了返回多个实体类型，并且假定使用缺省元语和缺省列名。

```
Query q = em.createNativeQuery(
  "SELECT o.id, o.quantity, o.item, i.id, i.name, i.description "+
  "FROM Order o, Item i " +
  "WHERE (o.quantity > 25) AND (o.item = i.id)",
  "OrderItemResults");


@SqlResultSetMapping(name="OrderItemResults",
    entities={
       @EntityResult(entityClass=com.acme.Order.class),
       @EntityResult(entityClass=com.acme.Item.class)
    })
```

当返回一个实体时，SQL 语句查询所有映射到实体对象的列。这应当包括关联其它实体的外键列。当返回的数据不足够（数据列没有定义的多）时，返回的结果没有规定。SQL 结果集映射不必将结果与实体的非持久化字段进行映射。

在 SQL 结果集映射注释符中使用的列名指的是在 SQL SELECT 语句中使用的列名。注意，如果 SQL 结果中有相同名字的列时，必须在 SQL SELECT 语句中使用列的别名。

下例是组合多个实体类型，它在 SQL 语句内使用了别名，并且将这些列名显式地映射到实体的字段上。FieldResult 注解用于定义这种映射。

```
Query q = em.createNativeQuery(
      "SELECT o.id AS order_id, " +
             "o.quantity AS order_quantity, " +
             "o.item AS order_item, " +
             "i.id, i.name, i.description " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item = i.id)",
      "OrderItemResults");

   @SqlResultSetMapping(name="OrderItemResults",
       entities={
          @EntityResult(entityClass=com.acme.Order.class, fields={
             @FieldResult(name="id", column="order_id"),
             @FieldResult(name="quantity", column="order_quantity"),
             @FieldResult(name="item", column="order_item")}),
          @EntityResult(entityClass=com.acme.Item.class)
   })
```

通过指定 ColumnResult 注释符，可以在查询结果内包含标量的结果类型。

```
Query q = em.createNativeQuery(
      "SELECT o.id AS order_id, " +
             "o.quantity AS order_quantity, " +
             "o.item AS order_item, " +
             "i.name AS item_name, " +
         "FROM Order o, Item i " +
         "WHERE (order_quantity > 25) AND (order_item = i.id)",
      "OrderResults");

   @SqlResultSetMapping(name="OrderResults",
      entities={
         @EntityResult(entityClass=com.acme.Order.class, fields={
            @FieldResult(name="id", column="order_id"),
            @FieldResult(name="quantity", column="order_quantity"),
            @FieldResult(name="item", column="order_item")})},
      columns={
         @ColumnResult(name="item_name")}
   )
```

当返回的实体类型是单值关系的拥有者且外键是组合外键时，应当为每个外键列使用 FieldResult 元素。FieldResult 元素必须使用“.”来表明哪个列映射到目标实体主键的哪个属性/字段。接下来描述的“.”形式除了组合主键或嵌入式主键外，在其他的用法中不要求支持。

如果目标实体有类型为 IdClass 的主键，那么使用：关系字段或属性的名字+“.”+目标实体主键字段或属性的名字的形式。正如在 8.1.15 章节所述，后者将用 Id 注释。

举例：

```
Query q = em.createNativeQuery(
      "SELECT o.id AS order_id, " +
            "o.quantity AS order_quantity, " +
            "o.item_id AS order_item_id, " +
            "o.item_name AS order_item_name, " +
            "i.id, i.name, i.description " +
      "FROM Order o, Item i " +
      "WHERE (order_quantity > 25) AND (order_item_id = i.id) AND
(order_item_name = i.name)",
            "OrderItemResults");
@SqlResultSetMapping(name="OrderItemResults",
```

```
    entities={
        @EntityResult(entityClass=com.acme.Order.class, fields={
            @FieldResult(name="id", column="order_id"),
            @FieldResult(name="quantity", column="order_quantity"),
            @FieldResult(name="item.id", column="order_item_id")}),
            @FieldResult(name="item.name",
column="order_item_name")}),
        @EntityResult(entityClass=com.acme.Item.class)
})
```

如果目标实体由一个类型为 EmbeddedId 的主键，那么使用：关系字段/属性的名字+“.”+主键的属性/字段名字（例如，注释为 EmbeddedId 的字段或属性的名字）+对应的被嵌入的主键类的属性或字段的名字的形式。

例子：

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item_id AS order_item_id, " +
        "o.item_name AS order_item_name, " +
        "i.id, i.name, i.description " +
    "FROM Order o, Item i " +
    "WHERE (order_quantity > 25) AND (order_item_id = i.id) AND
(order_item_name = i.name)",
        "OrderItemResults");
@SqlResultSetMapping(name="OrderItemResults",
    entities={
        @EntityResult(entityClass=com.acme.Order.class, fields={
            @FieldResult(name="id", column="order_id"),
            @FieldResult(name="quantity", column="order_quantity"),
```

```
        @FieldResult(name="item.itemPk.id",
column="order_item_id")}),
        @FieldResult(name="item.itemPk.name",
column="order_item_name")}),
    @EntityResult(entityClass=com.acme.Item.class)
})
```

用于组合外键的 FieldResult 元素与目标实体的 EmbeddedId 类结合使用。如果关系被 eagerly 加载，那么这种组合就可以用于读取实体。

没有定义本地查询如何使用命名参数。对于 SQL 查询，只有使用位置参数才能使应用可移植。

目前只支持单值关系的对连接查询。

2.7 异常汇总

下面是在规范中定义的所有异常：

2.7.1 PersistenceException

PersistenceException 是当出现问题时由持久化提供商抛出。它可以是调用的操作由于不期望的错误而没有完成(例如，持久化提供商不能打开数据库连接)。

在规范中定义的其他异常都是 PersistenceException 的子类。除了 NoReusltException 和 NoUniqueResultException 外的 PersistenceException 的所有实例都会引起当前活动事务的回滚。

2.7.2 TransactionRequiredException

当要求事务，但事务是不活动时抛出 TransactionRequiredException。

2.7.3 OptimisticLockException

当乐观锁发生冲突时抛出 OptimisticLockException。这个异常可以在 flush

或提交的 API 调用中被抛出。如果当前是活动事务，则回滚该事务。

2.7.4 RollbackException

当 EntityTransaction.commit 失败时抛出 RollbackException。

2.7.5 EntityExistsException

当执行 persist 操作但实体已经存在时抛出 EntityExistsException。EntityExistsException 可以在调用持久化操作时抛出，或者在提交时抛出 EntityExistsException 或另一个 PersistenceException 时抛出。

2.7.6 EntityNotFoundException

当通过 getReference 获取实体的引用但实体不存在时抛出 EntityNotFoundException。也可以在执行刷新操作时实体在数据库中已经不存在抛出。如果当前事务是活动的，则回滚事务。

2.7.7 NoResultException

当调用 Query.getSingleResult 但没有结果返回时抛出 NoResultException。这个异常不会引起当前活动事务的回滚。

2.7.8 NonUniqueResultException

当调用 Query.getSingleResult 但查询结果有多个时抛出 NonUniqueResultException。这个异常不会引起当前活动事务的回滚。

3 查询语言

Java 持久化查询语言用于根据实体和他们的字段定义查询。EJB QL 使得开发人员能够以跨平台的方式指定查询元语，而不依赖于特定的数据库。

Java 持久化查询语言是对 EJBQL（在《企业 JavaBean》中定义）的扩展。Java 持久化查询语言增加了操作，包括批更新和批删除，连接操作，GROUP BY，HAVING，投影（projection），以及子查询；并且支持动态查询和命名参数查询。整个语言既可以用在静态查询，也可以用于动态查询。

本章对 java 持久化查询语言做了全面的定义。

3.1 概述

Java 持久化查询语言是一个使用元语描述动态和静态查询的查询规范语言。它用于定义使用本规范中定义的实体和它们的持久化字段及其关系的查询。

Java 持久化查询语可以被编译成本地语言，如数据库的 SQL 或其他持久化存储的语言。这样，查询就可以方便地切换到数据库提供的本地语言，而不要需要在运行时切换。这样的话，查询的方法可以被很好的优化。

查询语言使用实体（包括关系）的抽象持久化 schema，并且基于数据模型定义操作和表达式。它使用类似 SQL 的语法基于实体抽象 schema 和它们的关系来查询对象或值。这样可以做到在实体被部署前可以解析和验证查询。

术语“抽象持久化 schema”指的是 java 持久化查询操作的持久化 schema 抽象（持久化实体，它们的状态，以及它们的关系）。使用这些持久化 schema 抽象的查询被转换成使用目标数据库 schema（实体对应的 schema）的查询。参见 3.3 章节。

查询可以定义在注释符中，也可以定义在 XML 描述符中。如果查询和实体定义在同一个持久化单元内，可以在一个查询内使用多个实体的抽象 schema。路径表达式可以遍历在同一个持久化单元内的关系。

一个持久化单元定义了应用相关的或分组的所有类，并且这些类必须被映射到同一个数据库中。

3.2 语句类型

一个 java 持久化语言语句可以是一个 select 语句、更新语句、或者是一个删

除语句。

本章把所有这些语句都称为“查询”。这样，在提到特殊的语句类型时，能区分出特殊的语句类型。

用 BNF 语法，查询语句被定义为：

QL_statement :: = select_statement | update_statement | delete_statement

一个持久化查询语言语句可以动态的构造，也可以静态的定义在注解符中或 XML 配置文件的元素内。

所有的查询语句都可以有参数。

3.2.1 Select 语句

一个 select 语句由以下语句组成：

- 一个 SELECT 语句，它决定了将要选择的对象或值的类型。
- 一个 FROM 语句，它提供了一些声明，这些声明用于指派域（译者注：领域对象）到查询语句内其他语句中特定的表达式上。
- 一个可选的 WHERE 语句，用于限定查询返回的结果。
- 一个可选的 GROUP BY 语句，它用于按照分组方式组织返回查询结果。
- 一个可选的 HAVING 语句，用于过滤组。
- 一个可选的 ORDER BY 语句，用于排序返回的结果。

用 BNF 语法，select 语句被定义为：

select_statement :: = select_clause from_clause [where_clause] [groupby_clause][having_clause] [orderby_clause]

一个 select 语句必须有一个 SELECT 和一个 FROM 语句。“[]”表示其内的语句是可选的。

3.2.2 Update 和 Delete 语句

更新和删除语句提供了对实体集合的批量操作。

用 BNF 语法，更新和删除操作定义为：

update_statement :: = update_clause [where_clause]

delete_statement :: = delete_clause [where_clause]

更新和删除语句决定了实体被更新还是被删除的类型。WHERE 语句用于限定更新或删除的范围。

更新和删除语句在 3.10 中作进一步的描述。

3.3 抽象 Schema 类型和查询域

Java 持久化语言是类型语言，他的每一个表达式都有一个类型。表达式的类型来自表达式的结构、标识变量声明的抽象 Schema 类型、可以与持久化字段和关系进行计算的类型、以及文字类型。

实体的抽象 Schema 类型来自实体类和由 java 注解符（或 XML 配置）提供的元语信息。

非正式情况下，实体的抽象 Schema 类型可以有以下特征：

- 对实体类的每一个持久化字段或者 getter 方法(对于持久化属性)来说，它有一个字段（“状态字段”），这个字段的抽象 Schema 类型和这个字段类型或 getter 方法的返回值的类型一致。
- 对实体类的每一个持久化关系字段或它的 getter 方法来说（对于持久化关系属性），它有一个字段（“关联字段”），这个关联字段的类型是关联实体的抽象 Schema 类型（或者，如果关系是一对多或多对多，那么它的类型是相关实体的抽象 Schema 类型的集合）。

抽象 Schema 类型是数据模型专有的类型。所以持久化提供商可以不实现它，否则需要持久化提供商需要具体化一个抽象 Schema 类型。

一个 EJB QL 查询的域由定义在同一个持久化单元内的所有实体的抽象 Schema 类型组成。

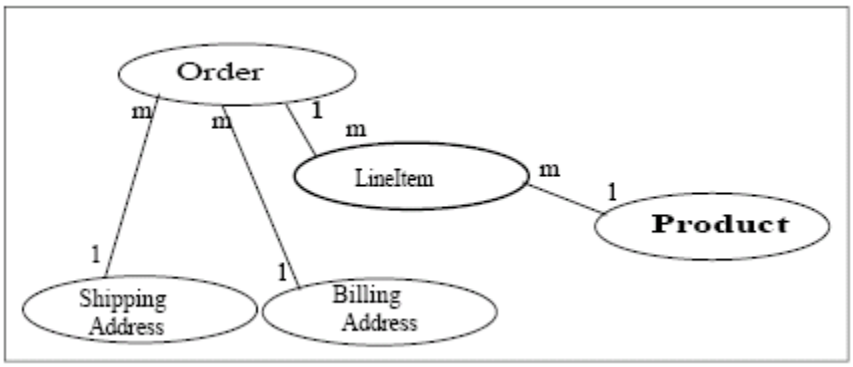
查询的域可以通过实体的关系 navigability 来限定。一个实体的抽象 Schema 类型的关系字段决定了 navigability。如果使用关系字段和他们的值，那么查询可以查询相关的实体，以及可以在查询中使用他们的抽象 Schema 类型。

3.3.1 命名

通过实体的名字可以在查询语句内指派实体。通过 Entity 注释符的 name 元素可以定义实体的名字（或者通过 XML 的 entity-name 元素），缺省是实体类的简单名字。实体的名字在持久化单元内必须唯一。

3.3.2 举例

这个例子假定应用开发者提供了几个实体类，分别是订单、产品、订单明细、托运地址和账单地址。他们的抽象 Schema 类型分别是 Order、Product、LineItem、ShippingAddress 和 BillingAddress。这些实体逻辑上在同一个持久化单元内，如下图所示：



上图的几个实体和抽象持久化 Schema 一起定义在同一个持久化单元内。

实体 ShippingAddress 和 BillingAddress 和 Order 之间都是一对多的关系。在 Order 和 LineItem 之间也是一对多的关系。实体 LineItem 和 Product 是多对一的关系。

查询订单的查询语句可以通过关联字段和状态字段 Order 和 LineItem 来遍历。查询所有带明细的订单可以写成下面这种形式：

```
SELECT DISTINCT o
FROM Order AS o JOIN o.lineItems AS l
WHERE l.shipped = FALSE
```

查询通过 Order 的关联字段 lineItems 去查找所有的订单明细，然后使用

LineItem 的 shipped 字段去过滤那些还没有完全被托运的订单。(注意：这个查询没有查询那些没有订单明细的订单)。

尽管预定了一些保留标识，如 DISTINCT、FROM、AS、JOIN、WHERE 和 FALSE，且在这个例子中用大写，但保留标识是不区分大小写的。

这个例子中的 SELECT 语句设定了查询的返回值是 Order 类型。

由于在同一个持久化单元定义了相关实体的抽象持久化 Schema，所以开发者也可以指定一个使用 Product 的抽象 Schema 类型来查询订单的查询。因此，就会使用到抽象 Schema 类型 Order 和 Product 的状态字段和关联字段。例如，如果抽象 Schema 类型 Product 有一个名字为 productType 的状态字段，那么一个查询可以用这个字段来定义查询订单的查询。这个查询可以用于查找产品类型为供应办公室产品的所有订单。这个查询可以如下：

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType =’office_supplies’
```

因为 Order 是通过 LineItem 关联到 Product 的，所以使用 lineItems 和 product 关联字段来定义查询。这个查询使用抽象 Schema 的名字 Order 来指定，Order 名字指出了查询使用的抽象 Schema 类型。通过抽象 Schema 类型 Order 和 LineItem 的关联字段 lineItems 和 product 实现了抽象 Schema 类型的连接。

3.4 FROM 子句和连接声明

通过声明标识变量，FROM 子句定义了查询的域。标识变量是声明在查询的 FROM 子句中的唯一标识。路径表达式可以用于限制查询域。

标识变量用于标识一个特定实体抽象 Schema 类型的所有实例。FROM 子句可以包含多个标识变量，这些表示变量用“,”分开。

```
from_clause ::=
FROM identification_variable_declaration
{, {identification_variable_declaration | collection_member_declaration}}*
identification_variable_declaration ::= range_variable_declaration { join |
```

fetch_join }*

range_variable_declaration ::= abstract_schema_name [

AS] identification_variable

join ::= join_spec join_association_path_expression [AS] identification_variable

fetch_join ::= join_specFETCHjoin_association_path_expression

join_association_path_expression ::= join_collection_valued_path_expression |

join_single_valued_association_path_expression

join_spec::= [LEFT[OUTER]|INNER]JOIN

collection_member_declaration ::=

IN (collection_valued_path_expression) [AS] identification_variable

下面的几节来讨论在 FROM 子句中使用的结构。

3.4.1 标识符

一个标识符是一个无限长的字符串。这个字符串必须以 java 标识符的开始字符开始，并且所有其他字符必须是 java 标识符其余部分允许的字符（译者注：就是符合 java 标识符的规范）。标识符的第一个字符可以是 Character.isJavaIdentifierStart()方法返回 true 的任何字符。包括下划线（_）和美元（$）符号。标识符的其他部分可以是 Character.isJavaIdentifierPart()方法返回 true 的任何字符。问号（？）被 java 持久化语言保留了。

以下是保留的标识符：SELECT, FROM, WHERE, UPDATE, DELETE,JOIN, OUTER, INNER, LEFT, GROUP, BY, HAVING, FETCH, DISTINCT, OBJECT, NULL, TRUE,FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, AS, UNKNOWN（将来使用，现在被保留）, EMPTY, MEMBER, OF, IS, AVG,MAX, MIN, SUM, COUNT, ORDER, BY, ASC, DESC, MOD, UPPER, LOWER, TRIM, POSITION,CHARACTER_LENGTH, CHAR_LENGTH, BIT_LENGTH, CURRENT_TIME, CURRENT_DATE,CURRENT_TIMESTAMP, NEW, EXISTS, ALL, ANY, SOME。

保留字不区分大小写。这些保留字不能用于声明表示变量标识符。

建议：SQL 的保留字也不要使用，因为这些保留字在将来版本中也可能成为这个规范的保留字。

3.4.2 标识变量

标识变量是一个在 FROM 子句中声明的有效的标识符。

所有的标识变量必须声明在 FROM 子句中，不能在其他子句中声明变量。

标识变量不能是保留字，或者与同一个持久化单元内的实体名称相同。

标识变量是大小写敏感的。

标识变量是声明变量的表达式的类型的值。例如，对于前面的查询：

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType =  'office_supplies'
```

在 FROM 子句中声明 o.lineItems l，标识变量 l 是直接从 Order 得到的 LineItem 的值。关联字段 lineItems 是抽象 Schema 类型 LineItem 实例的集合，并且标识变量 l 指向这个集合内的一个元素。l 的类型是 LineItem 的抽象 Schema 类型。

标识变量总是指向一个对单值的引用。有三种途径来声明标识变量：在范围变量声明内，在 join 语句内或者在一个集合成员声明内。标识变量在 FROM 子句内按照从左到右的顺序计算，且一个标识变量可以作为上一个标识变量的查询结果。

3.4.3 范围变量声明

声明标识变量为范围变量的语法类似于 SQL 的语法；不同的是，AS 关键字是可选的。

range_variable_declaration::=abstract_schema_name [AS] identification_variable

范围变量声明可以让开发者为多个对象指明一个根，这些对象可能无法通过

遍历获得。

为了通过比较一个实体抽象 Schema 类型的多个实例来获得查询结果，则需要在 FROM 子句内为同一个实体抽象 Schema 类型声明多个标识变量来确定范围。

下面的查询语句返回比 Joh Smith 订单数多的订单。这个例子说明了如何在 FROM 子句内使用不同的标识变量，这两个不同的标识变量的抽象 Schema 类型都是 Order。查询的 SELECT 语句决定了返回的结果是订单数比 John Smith 订单多的订单。

```
SELECT DISTINCT o1
FROM Order o1, Order o2
WHERE o1.quantity > o2.quantity AND
o2.customer.lastname =  'Smith'  AND
o2.customer.firstname=  'John'
```

3.4.4 路径表达式

标识变量后跟“.”加上状态字段或关联字段构成路径表达式。路径表达式的类型就是导航结果的类型。也就是说，状态字段或关联字段的类型就是路径表达式的类型。

依靠导航，指向关联字段的路径表达式可以被进一步组合。如果源路径表达式的结果是单值类型（不是集合），那么路径表达式可以由多个单值路径表达式组合而成。

路径表达式导航可以使用“inner join”语义来组合。也就是说，如果在路径表达式中的非终点关联字段的值不是 null，那么这个路径可以被认为是没有值，并且不影响查询结果。

单值路径表达式和集合值路径表达式的语法如下所示：

```
single_valued_path_expression ::=
state_field_path_expression | single_valued_association_path_expression
state_field_path_expression ::=
{identification_variable                                                |
```

single_valued_association_path_expression}.state_field

single_valued_association_path_expression ::=

identification_variable.{single_valued_association_field.}*single_valued_association_field

collection_valued_path_expression ::=

identification_variable.{single_valued_association_field.}*collection_valued_association_field

state_field ::= {embedded_class_state_field.}*simple_state_field

Single_valued_association_field 用于指明在一对一或多对一关系中关联字段的名字。Single_valued_association_field 的类型和 single_valued_association_path_expression 都是都是关联实体的抽象 Schema 类型。

Collection_valued_association_field 用于指明在一对多或多对多关系中关联字段的名称。Collection_valued_association_field 的类型是关联实体的抽象 Schema 类型的值的集合。

Embedded_class_field 指实体中指向被嵌入类的状态字段的名字。

导航到关联实体则会产生关联实体的抽象 Schema 类型的值。

对状态字段内的终端路径表达式的计算会产生一个与这个状态字段的 Java 类型对应的抽象 Schema 类型。（就是会实例化一个这个字段的类型的对象）。

在语法上，不能使用由结果是集合的路径表达式组成的路径表达式。例如，如果 o 指向 Oreder，路径表达式 o.lineItems.product 就是错误的。由于导航到 lineItems 的结果是一个集合。为了处理这种导航，必须在 FROM 子句内声明一个标识变量指向 lineItems 集合内的元素，然后在 WHERE 子句内用另外一个路径表达式遍历这个元素。如下所示：

SELECT DISTINCT l.product FROM Order AS o, IN(o.lineItems) l

3.4.5 关联（Join）

内连接（inner join）是可以通过在 FROM 子句内使用笛卡尔积（即不写就是内连接），并且在 WHERE 子句内增加连接条件的方式来隐式地指定。如果不

使用条件限定，则是所有的笛卡尔积。

内连接通常用于关联条件不涉及到外键关联的情况（即关联条件中字段对应的字段属性间没有关联关系）。

例子：

select c from Customer c, Employee e where c.hatsize = e.shoesize

一般情况下，内连接的这种风格（也称为 theta-join）很少在实体内显式定义实体关系。

显式连接定义的语法如下：

join ::= join_spec join_association_path_expression [AS] identification_variable

fetch_join ::= join_spec FETCH join_association_path_expression

join_association_path_expression ::= join_collection_valued_path_expression |

join_single_valued_association_path_expression

join_spec::= [LEFT[OUTER]|INNER]JOIN

支持下述的内连接和外连接类型。

3.4.5.1内连接（关系连接）

内连接操作的语法如下：

[INNER]JOINjoin_association_path_expression [AS] identification_variable

例如，下面的查询基于客户和订单的关系进行连接。这种连接等同与数据库中的外键关系。

SELECT c FROM Customer c JOIN c.orders o WHERE c.status = 1

可以不用关键字 INNER，不用就是 INNER 连接。

SELECT c FROM Customer c INNER JOIN c.orders o WHERE c.status = 1

上面的语句等同于早期使用 IN 结构的查询语句（在 EJB2.0 中规定的），如下所示。下例是查询所有状态为 1 且至少有一个订单的客户：

SELECT OBJECT(c) FROM Customer c, IN(c.orders) o WHERE c.status = 1

3.4.5.2左外连接

LEFT JOIN 和 LEFT OUTER JOIN 是相同的。它们用于在连接条件内匹配的值缺少的情况下仍然可以查询到结果。

语法如下：

LEFT[OUTER] JOIN join_association_path_expression [AS] identification_variable

例如：

SELECT c FROM Customer c LEFT JOIN c.orders o WHERE c.status = 1

可以没有 OUTER 关键字。

SELECT c FROM Customer c LEFT OUTER JOIN c.orderso WHERE c.status=1

左连接一个重要的用途就是可以预先读取关联数据项作为查询的附加结果。这可以通过将 LEFT JOIN 指定为 FETCH JOIN 来实现。

3.4.5.3Fetch 连接

FETCH JOIN 用于在执行查询的时候顺便将关联数据项读取。FETCH JOIN 指定在实体和它的关联实体上。

语法如下：

fetch_join ::= [LEFT[OUTER]|INNER] JOIN FETCH join_association_path_expression

FETCH JOIN 语句的右边引用的关系必须属于查询结果的实体。FETCH JOIN 语句右边引用的实体不能有标识变量，因此，对隐式 fetched 的实体的引用不能出现在查询语句的其他地方。

下面的查询返会一个部门的集合。附带的，这些部门关联的雇员也将被查询出来，尽管在查询结果中没有显式的指定它们。被同时查出的雇员的持久化字段或属性也都被初始化。是否初始化雇员的关联属性需要根据 Employee 实体类的元语来决定。

SELECT d

FROM Department d LEFT JOIN FETCH d.employees

WHERE d.deptno = 1

除了连接操作右端指定的关联对象不在查询结果内或者在查询中没有被引用以外，Fetch 连接的语义与对应的内连接或外连接的语义相同。因此，例如，如果部门 l 由 5 个雇员，则上面的查询返回 5 个对部门 l 实体的引用。

3.4.6 集合成员声明

由 collection_member_declaration 声明的标识变量指向通过路径表达式导航获得的集合的值。这种路径表达式代表了一个涉及实体抽象 Schema 类型的关系字段的导航。由于一个路径表达式可以基于另外一个路径表达式，所以这种导航用于关联实体的关联字段。

用一个特定的操作来声明集合成员声明的标识变量，这个保留字是 IN。IN 操作的参数是值是集合的路径表达式。这个路径表达式将导航结果放入实体抽象 Schema 类型的值是集合的关系字段中。

声明集合成员标识变量的语法如下：

collection_member_declaration ::=

IN (collection_valued_path_expression) [AS] identification_variable

例如，查询：

SELECT DISTINCT o

FROM Order o JOIN o.lineItems l JOIN l.product p

WHERE p.productType = ‘office_supplies’

等同于下面用 IN 的查询：

SELECT DISTINCT o

FROM Order o, IN(o.lineItems) l

WHERE l.product.productType = ‘office_supplies’

在这个例子中，lineItems 是关联字段的名字，这个关联字段的值是 LineItem 实例的集合。标识变量 l 指向这个集合的成员，即一个单个 LineItem 实例。在这

个例子中，o 是 Order 的标识变量。

3.4.7 FROM 语句和 SQL

Java 持久化查询语言把 FROM 子句和 SQL 看作是相似的是因为声明的标识变量影响查询结果，即使它们没有在 WHERE 语句中使用。应用开发者应该谨慎定义标识变量，因为查询域可能会依赖声明的类型是否有值。

例如，下例的 FROM 语句用于查询所有有明细且产品存在的订单。如果在数据库中没有 Product 实例，那么查询域就是空的，同时也不会查询出订单。

SELECT o

FROM Order AS o, IN(o.lineItems) l, Product p

3.4.8 多义性

Java 持久化查询自动是多义性的。查询的 FROM 子句不仅指向显式指定的特定实体的实例，而且也指向它的子类。查询的结果包含满足条件的子类的实例。（注：这在 EJB2.0 中是不支持的，由于 EJB2.0 不支持继承。）

3.5 WHERE 语句

查询的 WHERE 语句由条件表达式组成，这些条件表达式用于选择满足表达式的对象或值。WHERE 语句用于限制选择语句的结果或者用于限制更新或删除操作的范围。

WHERE 语句的语法如下：

where_clause ::=WHERE conditional_expression

GROUP BY 用于根据实体的属性对值进行分组。HAVING 用于在组上进一步限制查询结果。

HAVING 的语法如下：

having_clause ::=HAVING conditional_expression

GROUP BY 和 HAVING 在 3.7 节中作进一步的讨论。

3.6 条件表达式

以下章节描述可以用在 WHERE 或 HAVING 语句内的条件表达式的语法结构。

在可移植的应用中，不要在条件表达式中使用那些映射为序列化形式或作为 blob 的状态字段。（注：不期望持久化实现能够在内存中执行带有这种字段的查询语句，而是要在数据库中执行）

3.6.1 语法

字符串是用单引号‘’括住的——例如，‘literal’。带有单引号的字符串需要用单引号转义——例如，‘literal’’s’。在查询中的字符串类似于 Java 的 String 使用 unicode 字符集编码。在查询的字符串语法中不支持 java 的转义符。

精确数值支持 java 的整型和 SQL 的整型数值语法。

近似数值支持 java 的浮点数以及 SQL 的近似数值语法。

枚举支持 java 的枚举语法。但必须指定枚举类。

为了和 java 语言规范保持一致，可以在数值后使用用适当的后缀来表示特定的数值类型。在本规范中不要求支持十六进制和八进制。

Booolean 是 TRUE 和 FALSE。

所有预定义的保留字都是不区分大小写的。

3.6.2 标识变量

正如在 3.4.2 章节中所述，所有在 SELECT 或 DELECT 语句内的 WHERE 或 HAVING 语句中使用的标识变量必须在 FROM 语句内声明。同样，在 UPDATE 语句内的 WHERE 语句中使用的标识变量也必须在 UPDATE 语句内声明。

在 WHERE 和 HAVING 语句内使用的标识变量必需是可量化的。意思是，这个标识变量代表的是实体抽象 Schema 类型实例或是一个集合的成员。标识变量从来都不会指向一个集合。

3.6.3 路径表达式

除了在 empty_collection_comparison_expression 、collection_memeber_expression 或作为 SIZE 操作的参数外，不能在 WHERE 和 HAVING 的条件表达式中使用 collection_valued_path_expression。

3.6.4 输入参数（入参）

可以使用位置和命名参数。在同一个查询语句内不可以同时使用位置参数和命名参数。

入参只能在查询语句的 WHERE 或 HAVING 语句内使用。

注意：如果入参的值是 null，涉及到入参的比较运算或数学运算将返回一个不知道的值。参见 3.11。

3.6.4.1位置参数

位置参数需遵循以下规则：

- 在整数前使用？表示入参。如，?1。
- 入参的序号从 1 开始。

注意：同一个参数可以在查询语句内使用多次，而且参数的使用顺序可以和位置参数的顺序不一致。

3.6.4.2命名参数

命名参数以“:”为前缀。它遵循在 3.4.1 章节中描述的规则。命名参数是大小写敏感的。

例如：

```
SELECT c
FROM Customer c
WHERE c.status = :stat
```

在 2.6.1 章节内讲述了绑定当命名参数的 API。

3.6.5 组合条件表达式

条件表达是可以由其他条件表达式、比较操作、逻辑操作、结果值是 boolean 值的路径表达式、布尔语法和布尔入参组成。

在比较表达式中可以使用算术表达式。算术表达式由其他的算术表达式、算术操作、结果值是数值的路径表达式和数值入参组成。

算术操作使用数值提升。

支持用“()”来改变表达式的计算顺序。

条件表达式的语法如下：

conditional_expression ::= conditional_term | conditional_expressionOR conditional_term

conditional_term ::= conditional_factor | conditional_termAND conditional_factor

conditional_factor ::= [NOT] conditional_primary

conditional_primary ::= simple_cond_expression | (conditional_expression)

simple_cond_expression ::=

comparison_expression |

between_expression |

like_expression |

in_expression |

null_comparison_expression |

empty_collection_comparison_expression |

collection_member_expression |

exists_expression

合计函数只能用在 HAVING 语句的条件表达式中，参见 3.7 章节。

3.6.6 操作符和操作优先级

下面列出的操作符的优先级是依次降低的：

- 导航操作符(.)
- 数学运算符
  - +，- 一元运算符
  - *，/
  - +，- 两元的加和减
- 比较运算符：=，>,>=,<,<=,<>,[NOT] BETWEEN,[NOT] LIKE,[NOT] IN,IS [NOT] NULL,IS [NOT] EMPTY,[NOT] MEMBER [OF]
- 逻辑运算符
  - NOT
  - AND
  - OR

下面的章节介绍在特定表达式内使用的操作符。

3.6.7 Between 表达式

在条件表达式内使用的[NOT] BETWEEN 操作符的语法如下：

arithmetic_expression [NOT]BETWEEN arithmetic_expressionAND arithmetic_expression |

string_expression [NOT]BETWEEN string_expressionAND string_expression |

datetime_expression [NOT]BETWEEN datetime_expressionAND datetime_expression

BETWEEN 表达式：x BETWEEN y AND z 等同于 y<=x AND x<=z。

对未知值和 NULL 值的规则在比较操作中同样适用。参加 3.11 章节。

例如：

p.age BETWEEN 15 and 19 is equivalent top.age >= 15 AND p.age <= 19

p.age NOT BETWEEN 15 and 19 is equivalent top.age < 15 OR p.age > 19

3.6.8 IN 表达式

在条件表达式中的[NOT] IN 操作符的语法如下：

in_expression ::=

state_field_path_expression [NOT]IN(in_item {, in_item}* | subquery)

in_item ::= literal | input_parameter

State_field_path_expression 必须有一个字符串、数值或枚举值。

literal 和 input_paramter 的值的类型必须和 State_field_path_expression 的抽象 Schema 类型在类型上相对应。（参见 3.12）

子查询的结果必须和 State_field_path_expression 的抽象 Schema 类型在类型上相对应。子查询在 3.6.15 章节“子查询”中描述。

例子：

o.country IN (‘UK’,’US’,’France’) is true for UK and false for Peru,and is equivalent to the expression (o.country=‘UK’) OR (o.country=‘US’) OR (o.country=’France’).

o.country NOT IN(‘UK’,’US’,’Frank’) is false for UK and true for Peru,and is equivalent to the expression NOT ((o.country=‘UK’) OR (o.country=‘US’ (o.country = ‘France’)).

为 IN 操作符定义的用逗号分开的值的列表中至少有一个元素。

如果在 IN 或 NOT IN 表达式内的 state_field_path_expression 的值是 NULL 或未知，那么整个表达式的值就是未知的。

3.6.9 Like 表达式

在条件表达式中的[NOT] LIKE 操作符的语法如下：

string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]

String_expression 必须是字符串。Pattern_value 是一个字符串或值是字符串的入参，在这个字符串中，“_”代表任意单个字符，“%”代表字符串（包括空串），其他字符的字符代表自己。Escape_character 是可选的，它是单个字符或值

是单个字符的入参（也就是，char 或 Character），用于转义在 pattern_value 内的下划线和百分号特殊字符。（参考《数据库语言 SQL.ANSI X3.135-1992 或 ISO/IEC 9075:1992》，它对这些规则有精确的定义）

例如：

- address.phone LIKE '12%3' is true for '123' '12993' and false for '1234'
- asentence.word LIKE 'l_se' is true for 'lose' and false for 'loose'
- aword.underscored LIKE '_%' ESCAPE '\' is true for '_foo' and false for 'bar'
- address.phone NOT LIKE '12%3' is false for '123' and '12993' and true for '1234'

如果 string_expression 或 pattern_value 的值是 NULL 或未知，那么 LIKE 表达式的值就是未知的。如果指定 escape_character 但它的值是 NULL，那么 LIKE 表达式的值就是未知的。

3.6.10 NULL 比较表达式

在条件表达式中的 [NOT] IS NULL 表达式的语法如下：

{single_valued_path_expression | input_parameter } IS [NOT] NULL

空比较表达式用于检查单值路径表达式或入参是否是 NULL 值。

3.6.11 EMPTY 集合比较表达式

在 empty_collection_comparison_expression 中使用 IS EMPTY 的语法如下：

collection_valued_path_expression IS [NOT] EMPTY

这个表达式用于检查值是集合的路径表达式的值集合是否是空的（没有元素）。

例如：

SELECT o

FROM Order o

WHERE o.lineItems IS EMPTY

如果在一个空集合比较表达式中的值是集合的路径表达式的值是未知的，那么空比较表达式的值就是未知的。

3.6.12 集合成员表达式

在 collection_member_expression 中使用 MEMBER OF（注：OF 是可选的）的语法如下：

entity_expression[NOT] MEMBER [OF]collection_valued_path_expression

entity_expression ::=

single_valued_association_path_expression | simple_entity_expression

simple_entity_expression ::=identification_variable |input_parameter

这个表达式用于检查指定的值是否是路径表达式集合值的成员。

如果路径表达式集合值是空集合，那么 MEMBER OF 表达式就是 FALSE，但 NOT MEMBER OF 表达式就是 TRUE。否则，如果集合值的路径表达式或单值关联字段路径表达式的值是 NULL 或未知，那么这个集合成员表达式的值也是未知的。

3.6.13 EXISTS 表达式

EXISTS 是一个谓词。只有当子查询结果由一个或多个值组成时，EXISTS 表达式的值是 TRUE，否则就是 FALSE。

语法如下：

exists_expression::= [NOT]EXISTS (subquery)

例如：

SELECT DISTINCT emp

FROM Employee emp

WHERE EXISTS (

```
SELECT spouseEmp
FROM Employee spouseEmp
WHERE EXISTS (
    SELECT spouseEmp
    FROM Employee spouseEmp
    WHERE spouseEmp = emp.spouse)
```

上例中查询的结果由所有职员的配偶也是职员的记录组成。

3.6.14 ALL 或 ANY 表达式

ALL 条件表达式是一个谓词。如果所有的值都在子查询的结果内或子查询的结果为空，那么 ALL 表达式就是 true。如果比较的结果只要有一行是 false，则 ALL 条件表达式就是 false，并且如果既不是 true 又不是 false，那就是未知。

ANY 条件表达式也是谓词。如果对子查询结果内的某些值的比较操作是 true ，那么 ANY 条件表达式就是 true。如果子查询的结果是空或与子查询结果的每一个值比较都是 false，那么 ANY 条件表达式的值就是 false，并且如果既不是 true 又不是 false，那就是未知。关键字 SOME 和 ANY 是同义词。

和 ALL 或 ANY 条件表达式一起使用的比较操作符有：=,<,<=,>,>=,<>。子查询的结果必须和比较操作符的其他参数的结果在类型上相对应。参见 3.12 章节。

ALL 或 ANY 表达式的语法如下：

```
all_or_any_expression ::= {ALL |ANY |SOME}(subquery)
```

例子：

```
SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
SELECT m.salary
FROM Manager m
WHERE m.department = emp.department)
```

3.6.15 子查询

可以在 WHERE 或 HAVING 语句内使用子查询。（注：子查询在这个规范版本中被限定到 WHERE 和 HAVING 语句。将在以后的版本中考虑在 FROM 子句中支持子查询）

子查询的语法如下：

subquery ::= simple_select_clause subquery_from_clause [where_clause]

[groupby_clause] [having_clause]

simple_select_clause ::=SELECT [DISTINCT] simple_select_expression

subquery_from_clause ::=

FROM subselect_identification_variable_declaration

{, subselect_identification_variable_declaration}*

subselect_identification_variable_declaration ::=

identification_variable_declaration |

association_path_expression [AS] identification_variable |

collection_member_declaration

simple_select_expression::=

single_valued_path_expression |

aggregate_expression |

identification_variable

例子：

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
   SELECT spouseEmp
   FROM Employee spouseEmp
   WHERE spouseEmp = emp.spouse)
SELECT c
```

```
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

注意：在某些上下文中一个子查询可以用于要求子查询是标量子查询（也就是说，产生一个单一结果）。在下面的例子中来解释说明，这个例子有一个数值比较操作：

```
SELECT goodCustomer
FROM Customer goodCustomer
WHERE goodCustomer.balanceOwed < (
   SELECT avg(c.balanceOwed) FROM Customer c)
```

3.6.16 函数表达式

Java 持久化查询语言有以下内置的函数，这些函数可以在 WHERE 或 HAVING 语句内使用。

如果函数表达式的参数值是 null 或未知，那么函数的值就是未知的。

3.6.16.1 字符串函数

```
functions_returning_strings ::=
CONCAT(string_primary, string_primary) |
SUBSTRING(string_primary,
simple_arithmetic_expression,simple_arithmetic_expression) |
TRIM(
[[trim_specification] [trim_character]FROM] string_primary) |
LOWER(string_primary) |
UPPER(string_primary)
trim_specification ::=LEADING | TRAILING | BOTH
functions_returning_numerics::=
LENGTH(string_primary)|
```

LOCATE(string_primary,string_primary[, simple_arithmetic_expression]) |

CONCAT 函数将两个字符串连接成一个字符串。

SUBSTRING 函数的第二个和第三个参数表示子串开始的位置和子串的长度。这两个参数都是整数。字符串的第一个位置是 1. SUBSTRING 返回一个子串。

TRIM 函数用于从字符串内剪切指定的字符。如果没有指定要剪切的字符，那么这个字符假定是空格。Trim_character 是可选的，它是一个只有一个字符的字符串或是一个字符（也就是 char 或 Character）（注：注意，不是所有的数据库都支持任意字符剪切，有的只支持空格，指定剪切字符可能使应用不可移植）。如果没有提供剪切规范（trim_specification），则假定是 BOTH。TRIM 函数返回被剪切后的字符串。

LOWER 和 UPPER 函数用于将字符串转换成大写或小写。它们返回转换后的字符串。

LOCATE 函数返回给定字符串在字符串中的位置，从给定的位置开始向后搜索。它以整数形式返回搜索到的第一个给定字符串的位置。第一个参数是用于定位的被搜索的字符串；第二个参数用于搜索的字符串；第三个参数是可选的，它指定从哪个位置开始搜索（缺省情况下，从被搜索的字符串的开始位置开始搜索）。字符串的第一个位置是 1. 如果没有搜索的要搜索的字符窜，则返回 0。（注：注意，不是所有的数据库都支持 LOCATE 的第三个参数；使用这个参数可能导致应用的不可移植性）

LENGTH 函数返回字符串的长度。

3.6.16.2 算术函数

functions_returning_numerics::=

ABS(simple_arithmetic_expression) |

SQRT(simple_arithmetic_expression)|

MOD(simple_arithmetic_expression, simple_arithmetic_expression) |

SIZE(collection_valued_path_expression)

ABS 函数有一个数值参数，返回一个和参数同样类型的数值（integer，float

或 double）。求绝对值。

SQRT 函数参数是数值参数，返回值是 double。

MOD 函数有两个整型参数，返回一个整数。

SIZE 函数返回一个整型值，集合内元素的个数。如果集合是空，那么 SIZE 函数返回 0.

在这些函数内的数值参数可以对应到 java 的对象数值类型，也对应 java 的原始数值类型。

3.6.16.3 时间函数

functions_returning_datetime:=

CURRENT_DATE |

CURRENT_TIME |

CURRENT_TIMESTAMP

时间函数返回数据库服务器的当前日期、时间和时间戳。

3.7 GROUP BY，HAVING

GROUP BY 用于根据一系列属性组合值。HAVING 用于在 GROUP BY 上进一步限定查询结果。

GROUP BY 和 HAVING 语句的语法如下：

groupby_clause ::=GROUP BYgroupby_item {, groupby_item}*

groupby_item ::= single_valued_path_expression | identification_variable

having_clause ::=HAVINGconditional_expression

如果查询同时有 WHERE 语句和 GROUP BY 语句，则查询过程就是，首先应用 WHERE 语句，然后组织组并根据 HAVING 语句过滤这些组。HAVING 语句将保留那些满足 HAVING 语句条件的组。

对于使用 GROUP BY 的 SELECT 语句的要求就是：出现在 SELECT 语句内的任何项（除了合计函数的参数）必须在 GROUP BY 中出现。当组织组时，null

值也被看作是一个组。

允许按实体进行分组。在这种情况下，实体必须不能包含已序列化的状态字段或 lob 字段。

HAVING 语句必须基于分组的项或使用分组项的合计函数指定查询条件。

如果没有使用 GROUP BY 和 HAVING，那么查询结果认为是一个组，并且 select 的结果列表只能由合计函数组成。不要求持久化实现支持没有 GROUP BY 但有 HAVING 的情况。可移植应用不应当依赖于没有 GROUP BY 但有 HAVING 的实现。

例如：

```
SELECT c.status, avg(c.filledOrderCount), count(c)
FROM Customer c
GROUP BY c.status
HAVING c.status IN (1, 2)

SELECT c.country, COUNT(c)
FROM Customer c
GROUP BY c.country
HAVING COUNT(c.country) > 3
```

3.8 SELECT 语句

SELECT 语句用于声明查询的结果。SELECT 语句可以返回多个值。

SELECT 语句可以包含一到多个下面的元素：单值变量或一个指向实体抽象 Schema 的单值标识变量，单值路径表达式，合计选择表达式和构造器表达式。

SELECT 的语法如下：

```
select_clause ::=SELECT [DISTINCT] select_expression {, select_expression}*
select_expression ::=
single_valued_path_expression |
aggregate_expression |
```

identification_variable |

OBJECT(identification_variable)|

constructor_expression

constructor_expression ::=

NEWconstructor_name(constructor_item {, constructor_item}*)

constructor_item ::= single_valued_path_expression | aggregate_expression

aggregate_expression ::=

{AVG |MAX |MIN |SUM }([DISTINCT] state_field_path_expression) |

COUNT ([DISTINCT] identification_variable | state_field_path_expression |

single_valued_association_path_expression)

例如：

```
SELECT c.id, c.status
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

注意，SELECT 语句必须被指定只返回单值表达式。下面的语句就是无效的：

```
SELECT o.lineItems FROM Order AS o
```

DISTINCT 关键字用于去除重复的查询结果。如果没有指定 DISTINCT，则不去掉重复值。

在 SELECT 语句中独立的标识变量可以不使用 OBJECT 操作限定。SELECT 语句不必使用 OBJECT 操作符去限定路径表达式。

3.8.1 SELECT 语句的结果类型

由查询语句内的 SELECT 语句指定的查询结果的类型就是实体的抽象 Schema 类型，状态字段类型，合计函数的结果，构造操作的结果，或者是这些类型的组合。

SELECT 语句的结果类型由 select_expressions 的结果类型定义。当在 SELECT 语句内使用多个 select_expressions 时，查询结果是 Object[]类型，并且在结果数组中元素的顺序和这些表达式在 SELECZT 语句中指定的顺序一致，同

时类型和 select_expressions 的类型相对应。

Select_expressions 的结果类型如下：

- 是一个 single_valued_path_expression，它是一个 state_field_path_expression。它的结果是一个对象，这个对象的类型和实体的状态字段的类型一致。如果这个状态字段的类型是原始类型，那么返回对应的对象类型。
- 是一个 single_valued_path_expression，它是一个 single_valued_associate_path_expression。它的结果是一个实体对象，这个对象的类型和实体对象的关系字段的类型或关系字段的子类型，这个类型由 O/R 映射确定。
- 是一个 Identification_variable 的结果类型。这个类型是标识变量对应的实体类型或它的子类型，由 O/R 映射确定。
- 是一个 aggreate_expression 的结果类型，这个类型在 3.8.4 章节中描述。
- 是一个 consturctor_expression 的结果类型，这个类型是类的构造器的类型。构造器内的参数的类型根据上述规则确定。

3.8.2 SELECT 语句内的构造器表达式

可以在 SELECT 列表内使用构造器，以便能一个或多个 java 实例。这个类可以不是实体或可以和数据库没有映射。但是，构造器名称必须是全称。

如果在 SELECT NEW 语句内使用实体类的名称，则查询的结果中实体实例的状态是 new。

```
SELECT NEW com.acme.example.CustomerDetails(c.id, c.status, o.count)
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

3.8.3 查询结果内的 Null 值

如果与关联字段或状态字段相对应的查询结果值是 null，那么查询方法的返

回值就是 null。IS NOT NULL 指令用于从查询结果内去除这些 null 值。

但是要注意，根据 java 的数字类型定义的状态字段类型在查询结果内不能产生出 NULL 值（即原始数值类型的值不会是 null）。返回这些字段的查询不必返回 null 值。

3.8.4 SELECT 语句内的合计函数

查询的结果可以是使用路径表达式的合计函数的结果。

在 SELECT 语句内可以使用以下合计函数：AVG，COUNT，MAX，MIN，SUM。

除了 COUNT 外，作为合计函数参数的路径表达式的最后必须是状态字段。作为 COUNT 函数参数的路径表达式的最后可以是状态字段，也可以是关联字段，还可以是标识变量。

SUM 和 AVG 函数的参数必须是数字。MAX 和 MIN 函数的参数的类型和可排序的状态字段类型一致（也就是，数字类型，字符串类型，字符类型或日期类型）。

包含在使用合计函数的查询结果内的 java 类型有：

- COUNT 返回 Long
- MAX，MIN 返回使用的状态字段的类型。
- AVG 返回 Double
- 当状态字段是整型（不是 BigInteger）时，SUM 返回 Long；当状态字段是浮点类型时，SUM 返回 Double；当状态字段类型是 BigInteger 时，SUM 返回 BigInteger；当状态字段类型是 BigDecimal 时，SUM 返回 BigDecimal。

如果使用 SUM，AVG，MAX 或 MIN，但没有参数值，那么合计函数的结果就是 NULL。

如果使用 COUNT，但没有参数值，那么合计的结果就是 0.

可以在合计函数的参数前加关键字 DISTINCT 来指定在合计之前去除重复值。（注：可以在 MAX 或 MIN 中使用 DISTINCT，但不影响结果）

在执行合计函数之前会将 Null 值去除，无论是否使用 DISTINCT 关键字。

3.8.5 例子

下面的例子返回平均的的订单数量：

```
SELECT AVG(o.quantity) FROM Order o
```

下面的查询返回 John Smith 订购的所有物品的总金额：

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = ‘Smith’ AND c.firstname = ‘John’
```

下面的查询返回订单总数：

```
SELECT COUNT(o)
FROM Order o
```

下面的查询计算 John Smith 的订单中所有给定价格的物品数量：

```
SELECT COUNT(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = ‘Smith’ AND c.firstname = ‘John’
```

注意：上面的查询等价于：

```
SELECT COUNT(l)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = ‘Smith’ AND c.firstname = ‘John’
AND l.price IS NOT NULL
```

3.9 ORDER BY 语句

ORDER BY 用于对查询结果或对象进行排序。

语法如下：

```
orderby_clause ::=ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression [ASC | DESC]
```

当在查询中使用 ORDER BY 语句时，SELECT 中的元素必须是下列元素之一：

- 标识变量 x，也可以表示为 OBJECT（x）
- 一个 single_valued_association_path_expression
- 一个 state_field_path_expression

对于前两种情况，每一个 orderby_item 必须是由 SELECT 语句返回的抽象 schema 值的可排序状态字段。对于第三种情况，orderby_item 必须和 SELECT 语句中的 state_path_expression 是同一抽象 schema 类型的相同字段。

例如，下面的前两个查询是正确的，但第三和第四两个是错误的：

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state =  ‘CA’
ORDER BY o.quantity, o.totalcost
SELECT o.quantity, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state =  ‘CA’
ORDER BY o.quantity, a.zipcode
```

下面的两个查询是不正确的，因为 orderby_item 不是 SELECT 中的元素。

```
SELECT p.product_name
FROM Order o JOIN o.lineItems l JOIN l.product p JOIN o.customer c
WHERE c.lastname =  ‘Smith’  AND c.firstname =  ‘John’
ORDER BY p.price

SELECT p.product_name
FROM Order o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname =  ‘Smith’  AND c.firstname =  ‘John’
ORDER BY o.quantity
```

如果指定多个 orderby_item，orderby_item 元素从左到右的顺序决定了排序

的优先顺序，因此，最左边的 orderby_item 的优先级最高。

关键字 ASC 指明使用升序排列；关键字 DESC 指明使用降序排列。缺省是升序排列。

对于 null 值的 SQL 排序规则是：所有的空值出现在非空值之前或所有的空值出现在非空值之后，但没有规定使用哪一个。

如果使用 ORDER BY，那么查询结果的顺序会被预先保存在查询方法的结果内。

3.10 批量更新和删除操作

批量更新和删除应用与单个实体类的实体(如果可能,可以包括它的子类)。在 FROM 或 UPDATE 语句中只能指定一个实体抽象 Schema 类型。

批量更新和删除的语法如下：

update_statement ::= update_clause [where_clause]

update_clause ::=UPDATE abstract_schema_name [[AS] identification_variable]

SET update_item {, update_item}*

update_item ::= [identification_variable.]{state_field | single_valued_association_field}=new_value

new_value ::=

simple_arithmetic_expression |

string_primary |

datetime_primary |

boolean_primary |

enum_primary

simple_entity_expression |

NULL

delete_statement ::= delete_clause [where_clause]

delete_clause ::=

DELETEFROM abstract_schema_name [[AS] identification_variable]

WHERE 语句的语法在 3.5 章节中描述。

删除操作只应用与指定类和它的子类的实体上，不会层级到相关的实体。

更新操作中指定的新值在类型上必须和它被赋予的状态字段的类型一致。

批量更新绕过乐观锁检查直接映射到数据库的更新操作上。可移植应用必须手动更新版本列的值，如果希望验证版本列的值，也需要手工验证。

持久化上下文不与批量更新和删除的结果同步。

执行批量更新或删除时应当非常小心，因为它们会引起数据库和活动持久化上下文中实体的不一致。通常情况下，批量更新和删除操作只应当在一个单独的事务中或在实体的开始处执行（在实体已经读取之前，实体的状态可能已经被这些操作改变了）。

例子：

```
DELETE
FROM Customer c
WHERE c.status =  ‘inactive’
DELETE
FROM Customer c
WHERE c.status =  ‘inactive’
  AND c.orders IS EMPTY
UPDATE customer c
SET c.status =  ‘outstanding’
WHERE c.balance < 10000
  AND 1000 > (SELECT COUNT(o)
                FROM customer cust JOIN cust.order o)
```

3.11 Null 值

当引用的目标对象在数据库中不存在时，它的值被认为是 NULL，SQL92 中 NULL 语法定义了包含 NULL 值的条件表达式的计算方法。

下面是这些语法的简要描述：

- 用 NULL 值进行的比较或算术运算的结果总是未知的值。
- 两个 NULL 值是不相等的，比较的结果是未知的。
- 对未知值进行比较或算术运算的结果仍然是未知的。
- IS NULL 和 IS NOT NULL 操作将 NULL 状态字段或单值关系字段的值转换为 TRUE 或 FALSE。
- Boolean 操作使用三个值逻辑，分别定义在 Table1，Table2 和 Table3.

Table1 AND 操作符的定义

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

Table2 OR 操作的定义

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Table3 NOT 操作的定义

NOT	
T	F
F	T
U	U

注意：Java 持久化语言定义了空串‘’，它不等于 NULL 值，只是字符串的长度为 0.然而，对某些数据库，NULL 值和空串没有严格区分。因此，应用开发

者不应当依赖涉及空串和NULL 值的比较语法。

3.12 相等和比较语法

只允许相似类型的值进行比较。相似类型是指两个类型是同一个 java 类型或者一个原始类型，另一个是原始封装类型那么两个类型（例如，int 和 Integer）。这个规则有一个特殊情况：比较那些遵循数值转换规则的数值类型总是有效的。除了数值类型外条件表达是不允许比较非相似的类型。

注意，允许类型为原始封装类型的状态字段和入参上使用算术操作和比较操作。

当同一个抽象 Schema 类型的两个实体有且只有相同的主键值时，那么他们才是相等的。

只要求支持对枚举进行相等或不等的比较。

3.13 举例

下面的例子解释 Java 持久化查询语言的语法和语义。这些例子基于 3.3.2 章节的例子。

3.13.1 简单查询

查询所有的订单：

SELECT o

FROM Order o

查找所有需要托运到加利福尼亚的订单：

SELECT o

FROM Order o

WHERE o.shippingAddress.state = ‘CA’

Find all states for which there are orders:

SELECT DISTINCT o.shippingAddress.state

FROM Order o

3.13.2 使用关系的查询

查询所有有订单明细的订单：

SELECT DISTINCT o

FROM Order o, IN(o.lineItems) l

注意，这个查询的结果不包括那些没有订单明细的订单。这个查询也可以写作：

```
SELECT o
FROM Order o
WHERE o.lineItems IS NOT EMPTY
```

查询所有没有订单明细的订单：

SELECT o

FROM Order o

WHERE o.lineItems IS NOT EMPTY

查询所有没有明细的订单：

SELECT o

FROM Order o

WHERE o.lineItems IS EMPTY

查询所有未确定的订单：

SELECT DISTINCT o

FROM Order o JOIN o.lineItems l

WHERE l.shipped = FALSE

查询那些托运地址和订单地址不同的订单。这个例子假定应用开发者使用两个不同的实体类型来描述托运地址和订单地址：

SELECT o

FROM Order o

WHERE

NOT (o.shippingAddress.state = o.billingAddress.state AND

o.shippingAddress.city = o.billingAddress.city AND

o.shippingAddress.street = o.billingAddress.street)

如果应用开发者在一个实体内使用两个不同的关系来表示托运地址和订单地址，上面的表达式就可以根据在 3.12 章节内定义的相等规则做简化。查询如下：

```
SELECT o
FROM Order o
WHERE o.shippingAddress <> o.billingAddress
```

这个查询检查是否同一个实体抽象 Schema 类型实例（用主键标识）通过两个不同的关系被关联到一个订单。

查询订购了书名为“Appling Enterprise JavaBean: Component-Base Development for the J2EE Platform”的书的所有订单：

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l
WHERE l.product.type =  ‘book ‘ AND
l.product.name =  ‘Applying Enterprise JavaBeans
Component-Based Development for the J2EE Platform’
```

3.13.3 使用输入参数的查询

下面的查询查找产品名称等于输入参数的订单：

```
SELECT DISTINCT o
FROM Order o, IN(o.lineItems) l
WHERE l.product.name = ?1
```

对于上面的查询，输入参数必须是和状态字段的类型一致，例如，一个字符串。

3.14 BNF

BNF 符号总览：

- { ... } grouping
- [...] optional constructs
- boldface keywords
- *zero or more
- |alternates

下面是 java 持久化查询语言的 BNF：

QL_statement ::= select_statement | update_statement | delete_statement

select_statement ::= select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]

update_statement ::= update_clause [where_clause]

delete_statement ::= delete_clause [where_clause]

from_clause ::=

FROM identification_variable_declaration

{, {identification_variable_declaration | collection_member_declaration}}*

identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*

range_variable_declaration ::= abstract_schema_name [AS] identification_variable

join ::= join_spec join_association_path_expression [AS] identification_variable

fetch_join ::= join_specFETCHjoin_association_path_expression

association_path_expression ::=

collection_valued_path_expression | single_valued_association_path_expression

join_spec::= [LEFT[OUTER]|INNER]JOIN

join_association_path_expression ::= join_collection_valued_path_expression |

join_single_valued_association_path_expression

join_collection_valued_path_expression::=

identification_variable.collection_valued_association_field

join_single_valued_association_path_expression::=

identification_variable.single_valued_association_field

collection_member_declaration ::=

IN (collection_valued_path_expression) [AS] identification_variable

single_valued_path_expression ::=

state_field_path_expression | single_valued_association_path_expression

state_field_path_expression ::=

{identification_variable | single_valued_association_path_expression}.state_field

single_valued_association_path_expression ::=

identification_variable.{single_valued_association_field.}* single_valued_association_field

collection_valued_path_expression ::=

identification_variable.{single_valued_association_field.}*collection_valued_association_field

state_field ::= {embedded_class_state_field.}*simple_state_field

update_clause ::=

UPDATE abstract_schema_name [[AS] identification_variable]

SET update_item {, update_item}*

update_item ::= [identification_variable.]{state_field | single_valued_association_field}=

new_value

new_value ::=

simple_arithmetic_expression |

string_primary |

datetime_primary |

boolean_primary |

enum_primary

simple_entity_expression |

NULL

delete_clause ::=DELETEFROM abstract_schema_name [[AS] identification_variable]

select_clause ::=SELECT [DISTINCT] select_expression {, select_expression}*

select_expression ::=

single_valued_path_expression |

aggregate_expression |

identification_variable |

OBJECT(identification_variable)|

constructor_expression

constructor_expression ::=

NEWconstructor_name(constructor_item {, constructor_item}*)

constructor_item ::= single_valued_path_expression | aggregate_expression

aggregate_expression ::=

{AVG |MAX |MIN |SUM }([DISTINCT] state_field_path_expression) |

COUNT ([DISTINCT] identification_variable | state_field_path_expression |

single_valued_association_path_expression)

where_clause ::=WHERE conditional_expression

groupby_clause ::=GROUP BYgroupby_item {, groupby_item}*

groupby_item ::= single_valued_path_expression | identification_variable

having_clause ::=HAVINGconditional_expression

orderby_clause ::=ORDER BYorderby_item {, orderby_item}*

orderby_item ::= state_field_path_expression [ASC |DESC]

subquery ::= simple_select_clause subquery_from_clause [where_clause]

[groupby_clause] [having_clause]

subquery_from_clause ::=

FROMsubselect_identification_variable_declaration

{, subselect_identification_variable_declaration}*

subselect_identification_variable_declaration ::=

identification_variable_declaration |

association_path_expression [AS] identification_variable |

collection_member_declaration

simple_select_clause ::=SELECT [DISTINCT] simple_select_expression

simple_select_expression::=

single_valued_path_expression |

aggregate_expression |

identification_variable

conditional_expression ::= conditional_term | conditional_expressionOR conditional_term

conditional_term ::= conditional_factor | conditional_termAND conditional_factor

conditional_factor ::= [NOT] conditional_primary

conditional_primary ::= simple_cond_expression |(conditional_expression)

simple_cond_expression ::=

comparison_expression |

between_expression |

like_expression |

in_expression |

null_comparison_expression |

empty_collection_comparison_expression |

collection_member_expression |

exists_expression

between_expression ::=

arithmetic_expression [NOT]BETWEEN

arithmetic_expressionAND arithmetic_expression |

string_expression [NOT]BETWEEN string_expressionAND string_expression |

datetime_expression [NOT]BETWEEN

datetime_expressionAND datetime_expression

in_expression ::=

state_field_path_expression [NOT]IN(in_item {, in_item}* | subquery)

in_item ::= literal | input_parameter

like_expression ::=

string_expression [NOT]LIKEpattern_value [ESCAPE escape_character]

null_comparison_expression ::=

IS[NOT] NULL

{single_valued_path_expression | input_parameter}

empty_collection_comparison_expression ::=

collection_valued_path_expressionIS [NOT] EMPTY

collection_member_expression ::= entity_expression

[NOT]MEMBER[OF]collection_valued_path_expression

exists_expression::= [NOT]EXISTS (subquery)

all_or_any_expression ::= {ALL |ANY |SOME}(subquery)

comparison_expression ::=

string_expressioncomparison_operator{string_expression|all_or_any_expression

}|

boolean_expression {=|<>} {boolean_expression | all_or_any_expression} |

enum_expression {=|<>} {enum_expression | all_or_any_expression} |

datetime_expression comparison_operator

{datetime_expression | all_or_any_expression} |

entity_expression {= |<> } {entity_expression | all_or_any_expression} |

arithmetic_expression comparison_operator

{arithmetic_expression | all_or_any_expression}

comparison_operator ::== |> |>= |< |<= |<>

arithmetic_expression ::= simple_arithmetic_expression |(subquery)

simple_arithmetic_expression ::=

arithmetic_term | simple_arithmetic_expression {+ |- } arithmetic_term

arithmetic_term ::= arithmetic_factor | arithmetic_term {* |/ } arithmetic_factor

arithmetic_factor ::= [{+ |-}] arithmetic_primary

arithmetic_primary ::=

state_field_path_expression |

numeric_literal |

(simple_arithmetic_expression) |

input_parameter |

functions_returning_numerics |

aggregate_expression

string_expression ::= string_primary |(subquery)

string_primary ::=

state_field_path_expression |

string_literal |

input_parameter |

functions_returning_strings |

aggregate_expression

datetime_expression ::= datetime_primary |(subquery)

datetime_primary ::=

state_field_path_expression |

input_parameter |

functions_returning_datetime |

aggregate_expression

boolean_expression ::= boolean_primary |(subquery)

boolean_primary ::=

state_field_path_expression |

boolean_literal |

input_parameter |

enum_expression ::= enum_primary |(subquery)

enum_primary ::=

state_field_path_expression |

enum_literal |

input_parameter |

entity_expression ::=

single_valued_association_path_expression | simple_entity_expression

simple_entity_expression ::=

identification_variable |

input_parameter

functions_returning_numerics::=

LENGTH(string_primary)|

LOCATE(string_primary,string_primary[, simple_arithmetic_expression]) |

ABS(

simple_arithmetic_expression) |

SQRT(simple_arithmetic_expression) |

MOD(

simple_arithmetic_expression, simple_arithmetic_expression) |

SIZE(

collection_valued_path_expression)

functions_returning_datetime ::=

CURRENT_DATE|

CURRENT_TIME |

CURRENT_TIMESTAMP

functions_returning_strings ::=

CONCAT(string_primary, string_primary) |

SUBSTRING(string_primary,

simple_arithmetic_expression,simple_arithmetic_expression)|

TRIM(

[[trim_specification] [trim_character]FROM] string_primary) |

LOWER(string_primary) |

UPPER(string_primary)

trim_specification ::=LEADING | TRAILING | BOTH

4 实体管理器和持久化上下文

4.1 持久化上下文

持久化上下文是一个受管理实体实例的集合，在持久化上下文中的每一个实体实例有一个唯一的标识。在持久化上下文中，实体实例及其生命周期都由实体管理器管理。

在 java EE 环境中，通常会在多个组件中使用一个 JTA 事务。这些组件经常需要在一个事务内获取同一个持久化上下文。为了在 Java EE 环境中满足实体管理器的这种用法，当一个实体管理器被注入到一个组件或直接通过 JNDI 搜索到管理器时，它的持久化上下文会自动的随着当前 JTA 事务一起传播，并且引用同一个持久化单元的 EntityManager 将提供获取当前 JTA 事务中的同一个持久化上下文。由 Java EE 容器进行的持久化上下文的传播避免了应用在组件间传递 EntityManager 实例。容器用这种方式管理持久化上下文的实体管理器称为容器管理的实体管理器。容器管理的实体管理器的生命周期由 Java EE 容器管理。

在 Java EE 环境中，很少有应用需要获取“独立的”持久化上下文——也就是说，不随着 JTA 事务而穿越多个 EntityManager 引用。而是，每创建一个实体管理器实例都会创建一个鼓励的持久化上下文，这个持久化上下文不能通过在同一个事务内的其他实体管理器获得。这种使用方式可以通过

EntityManagerFactory 的 createEntityManager 方法来实现。这种由应用创建和销毁持久化上下文的实体管理器称为应用管理的实体管理器。应用管理的实体管理器的生命周期由应用自己来管理。

要求在 Java EE 的 web 容器和 EJB 容器内都要支持容器管理的实体管理器和应用管理的实体管理器以及它们的持久化上下文。在 EJB 环境中，通常使用容器管理的实体管理器。

在 java SE 环境和 Java EE 应用客户端容器中，只要求支持应用管理的实体管理器。（注：注意不要求在应用客户端容器中支持 JTA）

4.2 获取 EntityManager

从实体管理器工厂中获取指向一个持久化上下文的实体管理器。

当使用容器管理的实体管理器（在 Java EE 环境中）时，应用不和实体管理器工厂交互。实体管理器直接通过依赖注入或 JNDI 获得，容器负责与实体管理器工厂交互。

当使用应用管理的实体管理器时，应用必须使用实体管理器工厂来管理实体管理器和持久化上下文的生命周期。

实体管理器可以不被多个线程间共享。实体管理器可以只在单线程方式下被获得。

4.2.1 在 Java EE 环境中获取 EntityManager

可以通过依赖注入或 JNDI 查找来获得容器管理的 EntityManager。容器管理持久化上下文的生命周期，以及透明的创建和管理实体管理器实例。

可以用 PersitenceContext 注释来注入 EntityManager。Type 元素指明是否使用事务范围的还是扩展的持久化上下文，这在 4.6 章节中有述。可选的 unitName 用于指派一个持久化单元（参看 7.4.2 章节）。

例如：

@PersistenceContext

```
EntityManager em;
@PersistenceContext(type=PersistenceContextType.EXTENDED)
EntityManager orderEM;
```

下面是使用 JNDI 查找的方式获取一个实体管理器：

```
@Stateless
@PersistenceContext(name="OrderEM")
public class MySessionBean implements MyInterface {
@Resource SessionContext ctx;
public void doSomething() {
EntityManager em = (EntityManager)ctx.lookup("OrderEM");
...
}
}
```

4.2.2 获取应用管理的 EntityManager

应用可以通过实体管理器工厂来获得 EntityManager。

用于获取一个应用管理的实体管理器的 EntityManagerFactory API 与是否是用于 Java EE 还是 Java SE 环境无关。

4.3 获取实体管理器工厂

EntityManagerFactory 接口供应用来创建应用管理的实体管理器。（注：也用于 Java EE 容器内部。参见 4.9 章节）

每个实体管理器工厂提供相同配置的实体管理器实例（例如，配置为连接到同一个数据库，使用由应用实现定义的相同的初始化设置，等等）。

在 JVM 中可以同时获取多个实体管理器工厂实例。（注：这可能是在使用多个数据库的情况下，因为在通常的配置中一个单一的实体管理器只和一个数据库通信。但是，每个持久化单元只有一个实体管理器工厂。）

EntityManagerFactory 的方法都是线程安全的。

4.3.1 在 Java EE 容器内获取实体管理器工厂

在 J2EE 容器内，EntityManagerFactory 可以用 PersistenceUnit 注释注入或者通过 JNDI 查找。可选的 unitName 元素用于指明使用哪个持久化单元（参见 7.4.2 章节）。

例如：

```
@PersistenceUnit
EntityManagerFactory emf;
```

4.3.2 在 Java SE 环境下获取实体管理器工厂

在 Java EE 容器环境外，javax.persistence.Persistence 类是提供获取实体管理器工厂的入口。应用通过调用这个类的 createEntityManagerFactory 方法创建实体管理器工厂，在 6.2.1 章节中进行描述。

例如：

```
EntityManagerFactory emf =
javax.persistence.Persistence.createEntityManagerFactory("Order");
EntityManager em = emf.createEntityManager();
```

4.4 EntityManagerFactory 接口

EntityManagerFactory 接口用于应用获取应用管理的实体管理器。当应用使用完实体管理器，并且/或者在应用关闭时，应用应当关闭实体管理器工厂。一旦 EntityManagerFactory 被关闭，它的所有实体管理器都被认为是关闭的。

```
public interface javax.persistence.EntityManagerFactory {
/**
  * Create a new EntityManager.
  * This method returns a new EntityManager instance each time
```

```
  * it is invoked.
  * The isOpen method will return true on the returned instance.
  */
public EntityManager createEntityManager();
/**
  * Create a new EntityManager with the specified Map of
  * properties.
  * This method returns a new EntityManager instance each time
  * it is invoked.
  * The isOpen method will return true on the returned instance.
  */
public EntityManager createEntityManager(Map map);
/**
  * Close the factory, releasing any resources that it holds.
  * After a factory instance is closed, all methods invoked on
  * it will throw an IllegalStateException, except for isOpen,
  * which will return false. Once an EntityManagerFactory has
  * been closed, all its entity managers are considered to be
  * in the closed state.
  */
public void close();
/**
  * Indicates whether the factory is open. Returns true
  * until the factory has been closed.
  */
public boolean isOpen();
}
```

传入 createEntityManager 方法的 Properties 可以包含任何提供商特有的属性，

提供商必须忽略不能被识别的属性。

提供商应当为这些属性使用自己的命名空间（例如，com.acme.persistence.logging），不应当那些使用 javax.persistence 和它的子空间。Javax.persistence 是本规范的保留空间。

4.5 控制事务

根据实体管理的事务类型，EntityManager 操作涉及的事务可以通过 JTA 或通过使用本地资源的 EntityTransaction API 来控制，它被映射成一个资源上的资源事务，这些资源成为实体管理器管理实体的基础。

由 JTA 管理事务的实体管理器称为 JTA 实体管理器。

由应用通过 EntityTransaction API 管理事务的实体管理器称为本地资源实体管理器。

容器管理的实体管理器一定是 JTA 实体管理器。JTA 实体管理器只用于 Java EE 容器。

应用管理的实体管理器可以是 JTA 实体管理器，也可以是本地资源实体管理器。

实体管理器用给定的事务类型来定义——或者是 JTA 或者是本地资源——同时，它的实体管理器工厂会在后台被配置和创建。参见 5.2.1.2 和 6.1.1 章节。

在 Java EE web 容器和 EJB 容器内要求支持 JTA 实体管理器和本地资源实体管理器。在 EJB 环境中，通常使用 JTA 实体管理器。通常情况下，在 Java SE 环境下只支持本地资源的实体管理器。

4.5.1 JTA EntityManager

通过 JTA 控制事务的实体管理器称为 JTA 实体管理器。JTA 实体管理器参与当前的 JTA 事务，它在实体管理器的外部开始和提交，并且广播到后台的资源管理器。

4.5.2 本地资源 EntityManager

实体管理器的事务是由应用通过 EntityTransaction API 控制的实体管理器称为本地资源的实体管理器。本地资源实体管理器事务通过持久化提供商被映射到资源事务。本地资源实体管理器可以使用服务器资源或本地资源来连接数据库，且不关心 JTA 事务的持久化，不管它是否是活动的。

4.5.2.1EntityTransaction 接口

EntityTransaction 接口用于控制在本地资源实体管理器上的资源事务。EntityManager.getTransaction 方法返回 EntityTransaction 接口。

当使用本地资源实体管理器且持久化提供商运行时抛出引起事务回滚的异常时，持久化提供上必须标记事务回滚。

如果 EnityTransaction.commit 操作失败，持久化提供商必须回滚事务。

```
public interface EntityTransaction {
/**
 * Start a resource transaction.
 * @throws IllegalStateException if isActive() is true.
 */
public void begin();
/**
 * Commit the current transaction, writing any unflushed
 * changes to the database.
 * @throws IllegalStateException if isActive() is false.
 * @throws RollbackException if the commit fails.
 */
public void commit();
/**
 * Roll back the current transaction.
```

```
  * @throws IllegalStateException if isActive() is false.
  * @throws PersistenceException if an unexpected error
  *condition is encountered.
  */
public void rollback();
/**
  * Mark the current transaction so that the only possible
  * outcome of the transaction is for the transaction to be
  * rolled back.
  * @throws IllegalStateException if isActive() is false.
  */
public void setRollbackOnly();
/**
  * Determine whether the current transaction has been marked
  * for rollback.
  * @throws IllegalStateException if isActive() is false.
  */
public boolean getRollbackOnly();
/**
  * Indicate whether a transaction is in progress.
  * @throws PersistenceException if an unexpected error
  *condition is encountered.
  */
public boolean isActive();
}
```

4.5.3 例子

下面的例子解释在 Java SE 环境下创建实体管理器工厂，以及用它来创建和

使用一个本地资源的实体管理器。

```
import javax.persistence.*;
public class PasswordChanger {
public static void main (String[] args) {
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("Order");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
User user = (User)em.createQuery
                ("SELECT u FROM User u WHERE u.name=:name AND
u.pass=:pass")
                .setParameter("name", args[0])
                .setParameter("pass", args[1])
                .getSingleResult();
if (user!=null)
user.setPassword(args[2]);
em.getTransaction().commit();
em.close();
emf.close ();
    }
}
```

4.6 容器管理的持久化上下文

当使用容器管理的持久化上下文时，持久化上下文的生命周期被自动地管理，且对应用是透明的，同时持久化上下文随着 JTA 事务传播。

容器管理的持久化上下文可以定义为单事务的生命期，也可以定义为跨多个事务的生命期，这在持久化上下文所在的 EntityManager 被创建时由 PersistenceContextType 来指定。本规范分别称这种持久化上下文为事务范围的持

久化上下文和扩展的持久化上下文。

持久化上下文的生命期由 PersistenceContext 注释符或在 XML 中用 persistence-context-ref 元素。缺省情况下使用事务范围的持久化上下文。

第 4.6.1 和 4.6.2 章节描述了事务范围和扩展的持久化上下文但是没有描述持久化上下文的传播。持久化上下文的传播在第 4.6.3 章节中描述。

持久化上下文总是和一个实体管理器工厂相关联。在以后章节中只要提到“持久化上下文”，就应该理解为“关联了一个实体管理器工厂的持久化上下文”。

4.6.1 容器管理的事务范围的持久化上下文

应用可以通过注入或在 JNDI 命名空间中 lookup 的方式获得一个带有事务范围的持久化上下文的实体管理器，这个持久化上下文绑定到 JTA 事务上。这个持久化上下文的类型缺省定义为 PersistenceContextType.TRANSACTION。

当在活动的 JTA 事务中调用容器管理的实体管理器时（注：特指当 EntityManager 接口的方法被调用时），开始一个新的持久化上下文，但这个持久化上下文还没有和 JTA 事务建立关联。在持久化上下文被创建后才和 JTA 事务建立关联。

当关联的 JTA 事务提交或回滚时，持久化上下文结束，并且由 EntityManager 管理的所有实体变成脱管的。

如果在事务范围外部调用 EntityManager，那么在方法调用的最后，所有从数据库加载的实体将立刻变成托管的。

4.6.2 容器管理的扩展持久化上下文

容器管理的扩展持久化上下文只能在有状态的会话 bean 中被初始化。它从被声明为依赖于一个类型为 PeristenceContextType.EXTEND 的实体管理器的有状态会话 bean 被创建开始存在，称为绑定到有状态会话 bean。通过 PersistenceContext 注释符或在 XML 中的 persistence-context-ref 元素来声明使用容器管理的扩展持久化上下文。

持久化上下文在有状态的会话 bean 的@Remote 方法完成后被容器关闭（或者在有状态会话 bean 被销毁后被关闭）。

4.6.2.1扩展持久化上下文的继承

如果一个有状态会话 bean 实例化一个有扩展持久化上下文的有状态会话 bean、，那么第一个有状态会话 bean 的扩展持久化上下文被第二个有状态会话 bean 继承并且绑定到它上，且这个规则迭代执行下去——不管这些有状态会话 bean 在创建时事务是否活动。

如果持久化上下文有状态会话 bean 继承，那么容器在所有继承它的有状态会话 bean 被删除或被销毁后才关闭该持久化上下文。

4.6.3 持久化上下文的传播

正如在 4.1 章节中所述，一个持久化上下文可以对应一到多个 JTA 实体管理器实例（它们都和同一个实体管理器关联——注：从不同的实体管理器工厂获得的实体管理器从不共享同一个持久化上下文）。

同 JTA 事务的传播一样，持久化上下文也在多个实体管理器实例间传播。

持久化上下文的传播只应用在本地环境中，不能进行远程传播。

4.6.3.1持久化上下文传播的要求

被容器跨越多个组件调用传播的持久化上下文要求如下。

1. 如果调用一个组件但没有 JTA 事务或 JTA 事务不传播，那么持久化上下文不传播。
   - 如果随后在组件内调用一个实体管理器，那么：
     - 调用定义为 PersistenceContextType.TRANSACTION 的实体管理器会引起产生一个新的持久化上下文（正如在 4.6.1 章节中所述）。
     - 调用定义为 PersistenceContextType.EXTEND 的实体管理器会引起已存在的扩展持久化上下文绑定到该组件上。

- 如果在 JTA 事务内调用实体管理器，那么持久化上下文将被绑定到 JTA 事务上。

2. 如果调用一个组件，且 JTA 事务被广播到该组件，那么：
   - 如果该组件是绑定了扩展持久化上下文的有状态会话 bean，并且在 JTA 事务上绑定了一个不同的持久化上下文，那么容器抛出 EJBException。
   - 否则，如果在 JTA 事务上绑定一个持久化上下文，那么传播并使用该持久化上下文。

4.6.4 例子

4.6.4.1容器管理的事务范围的持久化上下文

```
@Stateless
public class ShoppingCartImpl implements ShoppingCart {
@PersistenceContext EntityManager em;
public Order getOrder(Long id) {
return em.find(Order.class, id);
}
public Product getProduct(String name) {
return (Product) em.createQuery("select p from Product p
where p.name = :name")
.setParameter("name", name)
.getSingleResult();
}
  public LineItem createLineItem(Order order, Product product, int
quantity) {
LineItem li = new LineItem(order, product, quantity);
order.getLineItems().add(li);
```

```
em.persist(li);
return li;
}
}
```

4.6.4.2容器管理的扩展持久化上下文

```
@Stateful
@Transaction(REQUIRES_NEW)
public class ShoppingCartImpl implements ShoppingCart {
@PersistenceContext(type=EXTENDED)
EntityManager em;
private Order order;
private Product product;
public void initOrder(Long id) {
order = em.find(Order.class, id);
}
public void initProduct(String name) {
product = (Product) em.createQuery("select p from Product p
where p.name = :name")
.setParameter("name", name)
.getSingleResult();
}
public LineItem createLineItem(int quantity) {
LineItem li = new LineItem(order, product, quantity);
order.getLineItems().add(li);
return li;
}
}
```

4.7 应用管理的持久化上下文

当使用管理的久化上下文时，应用直接通过持久化提供商提供的实体管理器工厂进行交互来管理实体管理器的生命周期以及获得和销毁持久化上下文。

所有这些应用管理的持久化上下文在范围上是被扩展的，以及是跨越多个事务的。

EntityManager.close()和 isOpen 方法用于管理应用管理的实体管理器和它关联的持久化上下文的生命周期。

EntityManager.close()方法用于关闭一个操作一个实体管理器并是否它关联的持久化上下文和其他资源。在 close 方法被调用后，应用不能调用 EntityManager 实例上的任何方法，除了 getTransaction 和 isOpen，否则抛出 IllegalStateException。如果在事务活动时调用 close 方法，那么持久化上下文仍然被管理直到事务完成。

EntityManager.isOpen 方法表示实体管理器是否是 open 的。isOpen 方法在实体管理器被关闭前一直返回 true。

扩展的持久化上下文从使用 EntityManagerFactory.createEntityManager 创建实体管理器开始存在，直到通过 EntityManager.close 关闭实体管理器。从应用管理的实体管理器获得的扩展持久化上下文是独立的——不随着事务传播。

当使用 JTA 应用管理的实体管理器时，如果在 JTA 事务外创建实体管理器，那么应用负责调用 EntityManager.joinTransaction 来建立实体管理器与事务的的关联（如果希望建立关联）。

4.7.1 例子

4.7.1.1在无状态会话 bean 中使用应用管理的持久化上下文

```
/*
 * Container-managed transaction demarcation is used.
 * Session bean creates and closes an entity manager in
 * each business method.
```

```
 */
@Stateless
public class ShoppingCartImpl implements ShoppingCart {
@PersistenceUnit
private EntityManagerFactory emf;
public Order getOrder(Long id) {
EntityManager em = emf.createEntityManager();
Order order = (Order)em.find(Order.class, id);
em.close();
return order;
}
public Product getProduct() {
EntityManager em = emf.createEntityManager();
Product product = (Product) em.createQuery("select p from
Product p where p.name = :name")
.setParameter("name", name)
.getSingleResult();
em.close();
return product;
}
publicLineItemcreateLineItem(Orderorder,Productproduct,int
quantity) {
EntityManager em = emf.createEntityManager();
LineItem li = new LineItem(order, product, quantity);
order.getLineItems().add(li);
em.persist(li);
em.close();
return li; // remains managed until JTA transaction ends
```

```
}
}
```

4.7.1.2在无状态会话 bean 中使用应用管理的持久化上下文

```
/*
 * Container-managed transaction demarcation is used.
 * Session bean creates entity manager in PostConstruct
 * method and clears persistence context at the end of each
 * business method.
 */
@Stateless
public class ShoppingCartImpl implements ShoppingCart {
@PersistenceUnit
private EntityManagerFactory emf;
private EntityManager em;
@PostConstruct
public void init()
em = emf.createEntityManager();
}
public Order getOrder(Long id) {
Order order = (Order)em.find(Order.class, id);
em.clear(); // entities are detached
return order;
}
public Product getProduct() {
Product product = (Product) em.createQuery("select p from
Product p where p.name = :name")
.setParameter("name", name)
```

```
.getSingleResult();
em.clear();
return product;
}
publicLineItemcreateLineItem(Orderorder,Productproduct,int
quantity) {
em.joinTransaction();
LineItem li = new LineItem(order, product, quantity);
order.getLineItems().add(li);
em.persist(li);
// persistence context is flushed to database;
// all updates will be committed to database on tx commit
em.flush();
// entities in persistence context are detached
em.clear();
return li;
}
@PreDestroy
public void destroy()
em.close();
}
}
```

4.7.1.3在有状态会话 bean 中使用应用管理的持久化上下文

```
//Container-managed transaction demarcation is used
@Stateful
public class ShoppingCartImpl implements ShoppingCart {
@PersistenceUnit
```

```
private EntityManagerFactory emf;
private EntityManager em;
private Order order;
private Product product;
@PostConstruct
public void init() {
em = emf.createEntityManager();
}
public void initOrder(Long id) {
order = em.find(Order.class, id);
}
public void initProduct(String name) {
product = (Product) em.createQuery("select p from Product p
where p.name = :name")
.setParameter("name", name)
.getSingleResult();
}
public LineItem createLineItem(int quantity) {
em.joinTransaction();
LineItem li = new LineItem(order, product, quantity);
order.getLineItems().add(li);
return li;
}
@Remove
public void destroy() {
em.close();
}
}
```

4.7.1.4带有资源事务的应用管理的持久化上下文

```
// Usage in an ordinary Java class
public class ShoppingImpl {
private EntityManager em;
private EntityManagerFactory emf;
public ShoppingCart() {
emf = Persistence.createEntityManagerFactory("orderMgt");
em = emf.createEntityManager();
}
private Order order;
private Product product;
public void initOrder(Long id) {
order = em.find(Order.class, id);
}
public void initProduct(String name) {
product = (Product) em.createQuery("select p from Product p
where p.name = :name")
.setParameter("name", name)
.getSingleResult();
}
public LineItem createLineItem(int quantity) {
em.getTransaction().begin();
LineItem li = new LineItem(order, product, quantity);
order.getLineItems().add(li);
em.getTransaction().commit();
return li;
}
```

```
public void destroy() {
em.close();
emf.close();
}
}
```

4.8 对容器的要求

4.8.1 应用管理的持久化上下文

当使用应用管理的持久化上下文时，容器必须初始化实体管理器工厂并通过 JNDI 把它暴露给应用。容器可以使用内部的 API 取创建实体管理器工厂，也可以用 PersistenceProvider.createContainerEntityManagerFactory 来创建。但是要求容器支持第三方的持久化提供商，因此在这种情况下容器必须使用 PersistenceProvider.createContainerEntityManagerFactory 来创建实体管理器工厂，以及在关闭系统前使用 EntityManager.close 方法来销毁实体管理器工厂（如果在应用关闭前没有关闭它）。

4.8.2 容器管理的持久化上下文

容器负责管理容器管理的持久化上下文的生命周期，负责为 web 组件、会话 bean 和消息驱动 bean 注入 EntityManager，以及负责让使用者可以通过 JNDI 获取到 EntityManager 引用。

当容器使用第三方持久化提供商进行操作时，它使用在 4.9 章节中定义的协议来创建和销毁容器管理的持久化上下文。没有规定对每一个持久化上下文是否要创建一个新的实体管理器实例，或是否在某些时可以重用实体管理器。也没有规定容器如何维护持久化上下文和 JTA 事务直接的关系。

如果一个持久化上下文已经和一个 JTA 事务关联，那么容器在事务范围内为接下来的调用使用这个持久化上下文，使用规则遵循在 4.6.3 章节中定义的持久化上下文传播规则。

4.9 容器和持久化提供商之间的运行时协议

本章描述容器和持久化提供商之间的运行时协议。要求容器支持这些协议（当不使用第三方的持久化时不要求容器支持这些协议：容器可以使用这些 API 或使用自己的 API）。

4.9.1 容器的职责

如果 EntityManager 没有和 JTA 事务关联，则负责管理事务范围的持久化上下文。

- 当在 JTA 事务中的业务方法第一次调用设置为 PersistenceContextType.TANSACTION 的实体管理器时，容器通过调用 EntityManagerFactory.createEntityManager 来创建一个实体管理器实例。
- 在 JTA 事务完成后（提交或回滚），容器通过调用 EntityManager.close 方法来关闭实体管理器（注：容器可以用池来存储 EntityManager 而不是每次都创建然后关闭，从池中获取一个 EntityManager 然后调用它的 clear 方法）。

如果使用事务范围的持久化上下文且 EntityManger 的 persist、remove、merge 或 refresh 方法在没有活动事务的情况下被调用，则容器必须抛出 TransactionRequireException。

对设置为扩展持久化上下文的有状态会话 bean 来说：

- 当一个声明为依赖一个设置为 PersistenceContextType.EXTENDED 的实体管理器的有状态会话 bean 被创建时，容器调用 EntityManagerFactory.createEntityManager 方法创建一个实体管理器。
- 在有状态会话 bean 和其他所有继承同一个持久化上下文的有状态会话 bean 被清除后，容器通过调用 EntityManager.close 方法来关闭实体管理器。
- 当调用有状态会话 bean 的业务方法时，如果有状态会话 bean 使用容器管理的事务分割，并且实体管理器还没有和当前事务关联，那么容器将实体管理器和当前 JTA 事务关联起来并调用 EntityManager.joinTransaction 方法。

如果和 JTA 事务关联的持久化上下文不同，那么容器抛出 EJBException。

- 当调用有状态会话 bean 的业务方法时，如果有状态会话 bean 使用受管理的事务分割，并且在这个方法内开始一个 UserTransaction，那么容器将 JTA 事务和持久化上下文关联起来并调用 EntityManager.joinTransaction。

如果应用调用容器管理的实体管理器的 EntityManager.close 方法，那么容器必须抛出 IllegalStateException。

当容器创建实体管理器时，它可以通过使用 EntityManagerFactory.createEntityManager(Map map)方法传入一个属性 map。如果已经在 PersistenceContext 注释中指定了属性或者在 persistence-context-ref 元素中指定了属性，那么必须使用这个方法并且在 map 中必须包含那些已指定的属性。

4.9.2 提供商的责任

提供商不知道事务范围的和扩展的持久化上下文之间的区别。它为容器提供实体管理器，并且为事务注册同步通知。

- 当调用 EntityManagerFactory.createEntityManager 时，提供商必须创建并返回一个新的实体管理器。如果有活动的 JTA 事务，那么提供商必须基于 JTA 事务注册同步通知。
- 当调用 EntityManager.joinTransaction 时，如果前一个 joinTransaction 调用还没有被处理，那么提供商必须基于当前 JTA 事务注册同步通知。
- 当 JTA 事务提交时，提供商必须 flush 所有更改的实体状态到数据库中。
- 当 JTA 事务回滚时，提供商必须脱管所有被管理的实体。
- 当提供上抛出引起事务回滚的异常时，提供商必须将事务标记为回滚。
- 当调用 EntityManager.close 时，提供商应当释放所有的资源，这些资源可以在所有涉及实体管理器的过期的事务完成后被收集。如果实体管理器已经是关闭状态，提供商必须抛出 IllegalStateException。
- 当调用 EntityManager.clear 时，提供商必须脱管所有被管理的实体。

5 实体打包

本章描述持久化单元的打包。

5.1 持久化单元

一个持久化单元是一个逻辑组，它包括：

- 一个实体管理器工厂和和它的实体管理器，以及它们的配置信息。
- 受管理类的集合，包括在持久化单元内的和由实体管理器管理的类。
- 映射元数据（用元数据注解或/和 XML 的形式），它们指定了类到数据库的映射关系。

5.2 打包持久化单元

在 Java EE 环境下，EJB-JAR、WAR、EAR 或应用客户端 JAR 都可以定义持久化单元。在这些包内可以定义任意数量的持久化单元。

持久化单元可以打包到包含在 WAR 或 EAR 内的一个或多个 jar 文件中，类似于在 EJB-JAR 文件中的类或 WAR 的 classes 目录内的类，或和以下的组合方式。

持久化单元定义在 persistence.xml 文件中。放置 persistence.xml 的 jar 文件或目录的 META-INF 目录称为持久化单元的 root。在 Java EE 中，持久化单元的根可以是下述情况的其中一种：

- 一个 EJB-JAR 文件。
- WAR 文件的 WEB-INF/classes 目录。（注：持久化单元的根是 WEB-INF/classes 目录；因此 persistence.xml 包含在 WEB-INF/classes/META-INF 目录中）
- 在 WAR 文件的 WEB-INF/lib 目录中的 jar 文件
- 在 EAR 根目录内的 jar 文件
- 在 EAR 的 lib 目录内的 jar 文件
- 一个应用客户端 jar 文件

不要求包含持久化单元的 EJB-JAR 或 WAR 必须被打包进 EAR 中，除非这个持久化单元包含的持久化类包含在 EJB-JAR 或 WAR 中。参见 5.2.1.6 章节。

可以在 persistence.xml 文件中的同一范围内指派多个持久化单元。

所有定义在 Java EE EAR 层级的持久化类必须可以被其他所有的 Java EE 组件获得——例如，可以被应用的类加载器加载——因此，如果同一个实体被两个不同的 Java EE 组件引用（它们可以使用不同的持久化单元），那么被引用的类是同一个类。

在 Java SE 环境下，元数据映射文件、jar 文件、以及在下列章节中描述的类都可以被使用。为了保证 Java SE 应用的可移植性，必须显式列出受管理的在持久化单元内的持久化类。参见章节 5.2.1.6。

5.2.1 Persistence.xml 文件

Persistence.xml 文件定义持久化单元。它也用于指定包含在持久化单元内受管理的持久化类、这些类的对象关系映射信息、以及持久化单元、实体管理器和实体管理器工厂的配置信息。Persistence.xml 位于持久化单元根目录的 META-INF 目录内。这些信息可以由如下所述的容器或引用定义。

对象关系映射可以采用在受管理持久化类类上增加注解符的形式、采用在持久化单元根目录内放置一到多个 XML 文件的形式、在类路径内在持久化单元根目录外并且从 persistence.xml 中引用一到多个 XML 文件的形式、或者这些形式的组合。

受管理的持久化类既可以在持久化单元的根目录内，也可以通过引用指定——例如，通过命名应用类路径中的类、类包或映射 XML 文件（按照引用类的顺序）；还可以通过组合这些方式。参见 5.2.1.6 章节。

Persistence 元素由一到多个 persistence-unit 元素组成。

Persistence 元素由 name 和 transaction-type 属性和下面的子元素组成：description，provider，jta-data-source，non-jta-data-source，mapping-file，jar-file，class，exclude-unlisted-classes 和 properties。

Name 元素是必须的；其他的属性和元素都是可选的。它们的语义在下面的

各小节中描述。

举例：

```
<persistence>
<persistence-unit name="OrderManagement">
<description>
This unit manages orders and customers.
It does not rely on any vendor-specific features and can
therefore be deployed to any persistence provider.
</description>
<jta-data-source>jdbc/MyOrderDB</jta-data-source>
<mapping-file>ormap.xml</mapping-file>
<jar-file>MyOrderApp.jar</jar-file>
<class>com.widgets.Order</class>
<class>com.widgets.Customer</class>
</persistence-unit>
</persistence>
<persistence>
<persistence-unit name="OrderManagement2">
<description>
This unit manages inventory for auto parts.
It depends on features provided by the
com.acme.persistence implementation.
</description>
<provider>com.acme.AcmePersistence</provider>
<jta-data-source>jdbc/MyPartDB</jta-data-source>
<mapping-file>ormap2.xml</mapping-file>
<jar-file>MyPartsApp.jar</jar-file>
<properties>
```

```
<property name="com.acme.persistence.sql-logging"
value="on"/>
</properties>
</persistence-unit>
</persistence>
```

5.2.1.1Name

Name 属性用于定义持久化单元的名字。这个名字可以用于标识被 PersistenceContext 和 PersistenceUnit 注解符引用的持久化单元，以及在程序级的 API 中用于创建实体管理器工厂。

5.2.1.2Transaction-type

Tansaction-type 属性用于指定由实体管理器工厂提供的实体管理器是否必须是 JTA 实体管理器还是本地资源实体管理器。这个元素的值是 JTA 或 RESOURCE_LOCAL。JTA 的 transaction-type 假定提供的是 JTA 数据资源——也可以通过 jta-data-source 元素指定或者由容器提供。通常情况下，在 Java EE 环境中，RESOURCE_LOCAL 的 transaction-type 认为是要提供一个非 JTA 的数据资源。在 Java EE 环境下，如果没有指定 transaction-type，缺省是 JTA。在 Java SE 环境中，缺省是 RESOURCE_LOCAL。

5.2.1.3Description

Description 元素用于为持久化单元提供描述信息。

5.2.1.4Provider

Provieder 元素指定持久化提供者的 javax.persistence.spi.PersistenceFactory 类的名字。如果使用第三方的持久化实现，则必须指定 Provider 元素。

5.2.1.5Jta-data-source, non-jta-data-source

在 Java EE 环境中，Jta-data-source 和 non-jta-data-source 用于分别指定持久化提供商使用的 JTA 和/或 non-JTA 数据源的全局 JNDI 名称。如果两者都没有被指定，那么配置人员必须在配置时指定一个 JTA 数据源或者容器必须提供 JTA 数据源，同时创建对应的 JTA EntityManagerFactory。

这些元素在本地环境中命名数据源；这些名字的格式和指定名字的能力根据产品的不同而不同。

在 Java SE 环境下，可以使用这些元素或也可以通过其他方式来指定数据源信息——依赖提供商的要求。

5.2.1.6Mapping-file，jar-file，class，exclude-unlisted-classes

下面列出的类必须显式或隐式地被指明作为受管理持久化类，以便被包含在持久化单元内：实体类、可嵌入类和被映射的超类。

被持久化单元管理的持久化类的集合通过下面的方式来定义（注意，一个类可以在多个持久化单元内使用）：

- 一个或多个 O/R 映射 XML 文件
- 一个或多个 jar 文件，这些文件用于搜索类
- 一个显式的类列表
- 包含在持久化单元根中的注解为受管理的持久化类（除非指定了 exclude-unlisted-classes 元素）

O/R 映射 XML 文件包含了列在它内的类的映射信息。可以在持久化单元的根的 META-INF 目录内或在 persistence.xml 引用的任何 jar 文件的 META-INF 目录内指定名字为 orm.xml 的 O/R 映射文件，另外，其他的映射文件可以被 persistence-unit 元素的 mapping-file 元素引用，而且可以出现在类路径内的任何地方。Orm.xml 或其他映射文件作为资源被持久化提供者加载。如果指定一个映射文件，那么将使用在映射文件内定义的类和映射信息。如果指定多个映射文件（可能包括一个 orm.xml 文件），那么合并所有文件内的映射信息。没有规定在

一个持久化单元引用多个映射文件（包括任何 orm.xml 文件）时，是否覆盖给定类的映射信息。包含在任何由持久化单元引用的映射文件内的 O/R 映射信息必须在类级别上与其他的 O/R 映射文件的映射分离。

除了在 mapping-file 元素中指定映射文件外，也可以用 jar-file 元素替代性的指定一个或多个 jar 文件。如果指定了 jar 文件，这些 jar 文件用于查找受管理持久化类，然后在这些类中的映射注解都会被处理，或者用在这个规范内定义的缺省的映射注解来映射它们。这些 JAR 文件用相对应持久化单元的根的相对路径来指定（例如，utils/myUtils.jar）。

除了 JAR 文件和映射文件外，也会替代性的指定一个命名受管理持久化类的列表。在这些类中的任何映射元数据注解都会被处理，或者用缺省的映射注解来映射它们。Class 元素用于列出受管理持久化类。所有命名的受管理类必须在 Java SE 环境中被指定以确保可移植性。可移植的 Java SE 应用不应当依赖在这里描述的其他机制来指定持久化单元的受管理类。在 Java SE 环境中持久化提供上也可以要求实体类和受管理的类的集合必须被完全排列在 persistence.xml 中。

也会在持久化单元的根中查找注解的受管理持久化类，并且在这些类中的任何映射元数据注解都会被处理，或者用缺省的映射注解来映射它们。如果不打算把在持久化单元的跟的被注解的持久化类包含在持久化单元中，那么使用 exclude-unlisted-classes 元素。Exclude-unlisted-classes 不能用于 Java SE 环境。

由持久化单元管理的实体的结果集是来源于上述的四个地方，如果对一个类既有 XML 映射文件，又有映射元数据注解（或缺省注解），那么 XML 内的映射信息将覆盖注解信息。覆盖的最小可移植级别是在持久化字段或属性的级别上。

所有作为持久化单元一部分的类和/或 jar 必须在应用的类路径内；从 persistence.xml 文件引用的类或 jar 不要求它们在类路径内。

所有的类必须在类路径上，以确保来自不同的持久化单元的实体管理器能得到同一个类。

5.2.1.7Properties

Properties 元素用于指定特定供应商的属性，这些属性应用到持久化单元和

持久化单元的管理器工厂配置上。

如果持久化提供商不能识别属性（除了那些在规范中定义的以外），提供商必须忽略它。

提供商应当为属性使用提供商的命名空间（例如，com.acme.persistence.logging），使用 javax.persistence 命名空间和它的子空间的属性不要用于特定供应商的信息。命名空间 javax.persistence 被这个规范用于保留字。

5.2.1.8例子

下面的例子展示了一个 persistence.xml 的内容。

例 1：

<persistence-unit name="OrderManagement"/>

创建名字为 orderManagement 的持久化单元。

在持久化单元根中的任何被注解为受管理持久化类都被增加到受管理持久化类的列表中。如果 META-INF/orm.xml 文件存在，那么被这个文件引用的所有类，以及包含在这个文件内的映射信息都会用于指定上述内容。由于没有指定提供商，持久化单元假定可以在多供应商间兼容。由于没有指定事务类型，假定是 JTA。容器必须提供数据源（例如，它可以在应用配置中指定）；在 Java SE 环境中，可以通过其他方式来指定数据源。

例 2：

<persistence-unit name="OrderManagement2">

<mapping-file>mappings.xml</mapping-file>

</persistence-unit>

创建名字为 OrderManagement2 的持久化单元。在持久化单元根中的任何被注解的持久化类被增加到受管理类的列表中。Mappings.xml 存在于类路径中，在这个文件内的任何类和映射信息都会作用于上述类。如果 META-INF/orm.xml 文件存在，那么在它中定义的任何类和映射信息也会被使用。事务类型、数据源和提供者都和前面所述的一样。

例 3：

<persistence-unit name="OrderManagement3">

<jar-file>order.jar</jar-file>

<jar-file>order-supplemental.jar</jar-file>

</persistence-unit>

创建名字为 OrderManagement3 持久化单元。在持久化单元根中的任何被注解的持久化类被增加到受管理类的列表中。如果 META-INF/orm.xml 文件存在，那么在这个文件内的任何类和映射信息都会被使用。Order.jar 和 order-supplemental.jar 以及/或者在这些 jar 文件中的 orm.xml 文件中指定的类都会用于查找受管理持久化类和任何被注解为受管理的持久化类。事务类型、数据源和提供者都和前面所述的一样。

例 4：

<persistence-unit name="OrderManagement4"

transaction-type=RESOURCE_LOCAL>

<non-jta-data-source>jdbc/MyDB</jta-data-source>

<mapping-file>order-mappings.xml</mapping-file>

<exclude-unlisted-classes/>

<class>com.acme.Order</class>

<class>com.acme.Customer</class>

<class>com.acme.Item</class>

</persistence-unit>

创建名字为 OrderManagement4 的持久化单元。Order-mapping.xml 作为资源被读取，它引用的任何类和包含的映射信息都会被使用。有注解的 Order，Customer 和 Item 类也会被加载，而且也会被增加到受管理类的列表中。在持久化单元根中的其他类也会被添加到受管理类的列表中。持久化单元是在多个提供商之间是可移植的。将创建支持本地资源实体管理器的实体管理器工厂。必须使用 jdbc//MyDB 数据源。

例 5：

<persistence-unit name="OrderManagement5">

<provider>com.acme.AcmePersistence</provider>

<mapping-file>order1.xml</mapping-file>

<mapping-file>order2.xml</mapping-file>

<jar-file>order.jar</jar-file>

<jar-file>order-supplemental.jar</jar-file>

</persistence-unit>

创建名字为 OrderManagement5 的持久化单元。在持久化单元根中的任何被注解的持久化类被增加到受管理类的列表中。Order1.xml 和 order2.xml 作为资源被读取，它们引用的任何类和包含的映射信息都会被用于查找。Order.par 是类路径中的包含另外一个持久化单元的 jar 文件，然而 order-supplement.jar 只是一个库文件。这两个都被用于查找注解为受管理持久化类，并且在它们中发现的任何注解为受管理类和/或在这些 jar 文件中的 orm.xml（如果可能）中指定的类都会被增加到受管理类的列表中。必须使用 com.acme.AcmePersistence 提供商。

注意：包含在 order.jar 中的 persistence.xml 文件不用于和根是 order.jar 持久化单元内的类一起增大持久化单元 EM-5。

5.2.2 持久化单元的范围

EJB-JAR、WAR、应用客户端 jar 或者 EAR 都可以定义持久化单元。

当使用 unitName 注解符元素或 persistence-unit-name 配置描述符元素来引用一个持久化单元时，持久化单元的可视范围由定义的点来决定。定义在 EJB-JAR、WAR 或应用客户端 jar 层级上的持久化单元范围分别是 EJB-JAR、WAR 或客户端 jar，对定义在这些 jar 或 war 的组件是可见的。定义在 EAR 层级上的持久化单元对应用中所有的组件都是可见的。

但是，如果在 EJB-JAR、WAR 或在 EAR 中的应用 jar 中定义了相同名字的持久化单元，那么在 EAR 层级的持久化单元对定义在 EJB-JAR、WAR 或应用 jar 是不可见的，除非使用#加上一个不会引起歧义的路径来引用这个持久化单元。当使用#时，路径是相对于发起引用的应用组件的 jar 文件。例如语

法 ../lib/persistenceUnitRoot.jar#myPersistenceUnit 指向一个名字为 myPersistenceUnit（这个名字与在 persistence.xml 的 name 元素的值相同），并且持久化单元的根的相对路径名为../lib/persistenceUnitRoot.jar。在 unitName 注释符元素和 persistence-unit-name 配置描述符元素中都可以使用#来引用一个在 EAR 层级的持久化单元。

5.3 Persistence.xml Schema

本节提供了 persistence.xml 的 XML schema。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- persistence.xml schema -->
<xsd:schema targetNamespace="http://java.sun.com/xml/ns/persistence"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:persistence="http://java.sun.com/xml/ns/persistence"
   elementFormDefault="qualified"   attributeFormDefault="unqualified"
   version="1.0">
   <xsd:annotation>
      <xsd:documentation>
         @(#)persistence_1_0.xsd   1.0   Feb 9 2006
      </xsd:documentation>
   </xsd:annotation>
    <xsd:annotation>
       <xsd:documentation><![CDATA[
       This is the XML Schema for the persistence configuration file.
       The file must be named "META-INF/persistence.xml" in the
       persistence archive.
       Persistence configuration files must indicate
       the persistence schema by using the persistence namespace:
       http://java.sun.com/xml/ns/persistence
```

```
        and indicate the version of the schema by
        using the version element as shown below:
          <persistence xmlns="http://java.sun.com/xml/ns/persistence"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
              http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
            version="1.0">
              ...
          </persistence>
        ]]></xsd:documentation>
      </xsd:annotation>
      <xsd:simpleType name="versionType">
        <xsd:restriction base="xsd:token">
          <xsd:pattern value="[0-9]+(\.[0-9]+)*"/>
        </xsd:restriction>
      </xsd:simpleType>
      <!-- ********************************************** -->
      <xsd:element name="persistence">
        <xsd:complexType>
          <xsd:sequence>
            <!--   ****************************************************
-->
            <xsd:element name="persistence-unit"
                         minOccurs="0" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:annotation>
                  <xsd:documentation>
                    Configuration of a persistence unit.
```

```
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
        <!--
********************************************************* -->
            <xsd:element name="description" type="xsd:string"
                         minOccurs="0">
              <xsd:annotation>
                <xsd:documentation>
                  Textual description of this persistence unit.
                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>
            <!--
********************************************************* -->
            <xsd:element name="provider" type="xsd:string"
                         minOccurs="0">
              <xsd:annotation>
                <xsd:documentation>
                  Provider class that supplies EntityManagers for this
                  persistence unit.
                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>
            <!--
********************************************************* -->
            <xsd:element name="jta-data-source" type="xsd:string"
                         minOccurs="0">
```

```
                    <xsd:annotation>
                      <xsd:documentation>
    Thecontainer-specificnameoftheJTAdatasourcetouse.
                      </xsd:documentation>
                    </xsd:annotation>
                  </xsd:element>
                  <!--
********************************************************** -->
                  <xsd:element name="non-jta-data-source" type="xsd:string"
                               minOccurs="0">
                    <xsd:annotation>
                      <xsd:documentation>
    Thecontainer-specificnameofanon-JTAdatasourcetouse.
                      </xsd:documentation>
                    </xsd:annotation>
                  </xsd:element>
                  <!--
********************************************************** -->
                  <xsd:element name="mapping-file" type="xsd:string"
                               minOccurs="0" maxOccurs="unbounded">
                    <xsd:annotation>
                      <xsd:documentation>
    Filecontainingmappinginformation.Loadedasaresource
                        by the persistence provider.
                      </xsd:documentation>
                    </xsd:annotation>
                  </xsd:element>
                  <!--
```

```
********************************************************* -->
                    <xsd:element name="jar-file" type="xsd:string"
                                 minOccurs="0" maxOccurs="unbounded">
                      <xsd:annotation>
                        <xsd:documentation>
                          Jar file that should be scanned for entities.
                          Not applicable to Java SE persistence units.
                        </xsd:documentation>
                      </xsd:annotation>
                    </xsd:element>
                    <!--
********************************************************* -->
                    <xsd:element name="class" type="xsd:string"
                                 minOccurs="0" maxOccurs="unbounded">
                      <xsd:annotation>
                        <xsd:documentation>
                          Class to scan for annotations.  It should be annotated
                          with either @Entity, @Embeddable or
@MappedSuperclass.
                        </xsd:documentation>
                      </xsd:annotation>
                    </xsd:element>
                    <!--
********************************************************* -->
      <xsd:elementname="exclude-unlisted-classes"type="xsd:boolean"
                                 default="false" minOccurs="0">
                      <xsd:annotation>
                        <xsd:documentation>
```

```
                When set to true then only listed classes and jars will
      bescannedforpersistentclasses,otherwisetheenclosing
      jarordirectorywillalsobescanned.Notapplicableto
                Java SE persistence units.
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <!--
********************************************************** -->
          <xsd:element name="properties" minOccurs="0">
            <xsd:annotation>
              <xsd:documentation>
                A list of vendor-specific properties.
              </xsd:documentation>
            </xsd:annotation>
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="property"
                             minOccurs="0"
maxOccurs="unbounded">
                  <xsd:annotation>
                    <xsd:documentation>
                      A name-value pair.
                    </xsd:documentation>
                  </xsd:annotation>
                  <xsd:complexType>
                    <xsd:attribute name="name" type="xsd:string"
                                   use="required"/>
```

```
                        <xsd:attribute name="value" type="xsd:string"
                                       use="required"/>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:sequence>
<!--
********************************************************** -->
<xsd:attribute name="name" type="xsd:string" use="required">
    <xsd:annotation>
        <xsd:documentation>
            Name used in code to reference this persistence unit.
        </xsd:documentation>
    </xsd:annotation>
</xsd:attribute>
<!--
********************************************************** -->
<xsd:attribute name="transaction-type"
                        type="persistence:persistence-unit-transac-
    tion-type">
    <xsd:annotation>
        <xsd:documentation>
            Type of transactions used by EntityManagers from this
            persistence unit.
        </xsd:documentation>
    </xsd:annotation>
```

```
                </xsd:attribute>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
          <xsd:attribute name="version" type="persistence:versionType"
                         fixed="1.0" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <!-- ********************************************** -->
      <xsd:simpleType name="persistence-unit-transaction-type">
        <xsd:annotation>
          <xsd:documentation>
            public enum TransactionType { JTA, RESOURCE_LOCAL };
          </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:token">
          <xsd:enumeration value="JTA"/>
          <xsd:enumeration value="RESOURCE_LOCAL"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:schema>
```

6 配置和启动时容器和提供者之间的协议

本章定义配置和启动时对 Java EE 容器和持久化提供者的要求。

6.1 Java EE 配置

配置在 Java EE 容器中的每一个持久化单元有一个 persistence.xml 文件、任

意数量的映射文件和任意数量的类文件组成。

6.1.1 容器的责任

在配置期间容器负责扫描在 5.2 章节中指定的位置，找到 persistence.xml 文件并处理它。

当容器找到 persistence.xml 文件时，它处理它锁包含的持久化单元。容器必须根据 persistence_1_0.xsd 来验证 persistenc.xml 并报告所有的错误。没有在 persistence.xml 中指定的提供者或数据源信息必须在配置时提供或者有容器提供。当为持久化单元创建实体管理器工厂时，容器可以有选择的增加容器特有的属性并传给提供者。

一旦容器读取了持久化元数据，它为每一个配置的命名持久化单元指定 javax.persistence.spi.PersistenceProvider 实现类。它创建这个实现类的实例并调用这个类的 createContainerEntityManagerFactory 方法。元数据以 PersistneceUnitInfo 类的形式作为调用的一部分传递给持久化提供者。容器使用返回的工厂来创建容器管理的试听管理器。对每一个配置的持久化单元配置，只能有一个 EntityManagerFactory 来创建。从这个给定的工厂可以创建任意数量的实体管理器。

当持久化单元被重新配置时，容器应当调用前一个 EntityManagerFactory 实例的 close 方法并且再次使用 PersistenceUnitInfo 元数据作为参数调用 createContainerEntityManagerFactory 方法。

6.1.2 持久化提供者的责任

持久化提供者必须实现 PersistenceProvider SPI 并能够处理传给它的元数据，这些元数据在调用 createContainerEntityManagerFactory 方法时被传入。使用 PersistenceUnitInfo 元数据来创建 EntityManagerFactory 的实例。然后将工厂返回给容器。

持久化提供者处理在持久化单元内的受管理类的元数据注解符。

当持久化提供者获得一个 O/R 映射文件时，它要处理映射文件包含的定义信息。持久化提供者必须根据 orm_1_0.xsd 来验证映射信息并报告所有的验证错误。

在 Java SE 环境下，持久化提供者必须根据 persistence_1_0.xsd 验证 persistence.xml 文件并报告所有的验证错误。

6.2 Javax.persistence.spi.PersistenceProvider

接口 javax.persistence.spi.PersistenceProvider 由容器提供者实现。

在 Java EE 环境下，它被容器调用。在 Java SE 环境下，它被 javax.persistence.Persistence 类调用。应用不能使用 javax.persistence.spi.PersistenceProvider 的实现。

PersistenceProvider 类必须有一个 public 的无参构造器。

在 Java SE 环境下，在 createEntityManagerFactory 方法中使用的属性在 6.1.3.1 章节中作进一步的描述。

```
package javax.persistence.spi;
/**
* Interface implemented by the persistence provider.
* This interface is used to create an EntityManagerFactory.
* It is invoked by the container in Java EE environments and
* by the Persistence class in Java SE environments.
*/
public interface PersistenceProvider {
/**
 * Called by Persistence class when an EntityManagerFactory
 * is to be created.
 *
 * @param emName The name of the persistence unit
 * @param map A Map of properties for use by the
```

```
     * persistence provider. These properties may be used to
     * override the values of the corresponding elements in
     * the persistence.xml file or specify values for
     * properties not specified in the persistence.xml
     * (and may be null if no properties are specified).
     * @return EntityManagerFactory for the persistence unit,
     * or null if the provider is not the right provider
     */
    public EntityManagerFactory createEntityManagerFactory(StringemName, Map map);
    /**
     * Called by the container when an EntityManagerFactory
     * is to be created.
     *
     * @param info Metadata for use by the persistence provider
     * @return EntityManagerFactory for the persistence unit
     * specified by the metadata
     * @param map A Map of integration-level properties for use
     * by the persistence provider (may be null if no properties
     * are specified).
     */
     public EntityManagerFactory createContainerEntityManagerFactory(PersistenceUnitInfo info, Map map);
    }
```

6.2.1 持久化单元属性

持久化单元属性放在 createEntityManagerFactory(String,Map)方法的 Map 参数中传递给持久化提供者。这些属性对应于 persistence.xml 文件中的元素。当在

Map 中指定这些属性时，它们的值覆盖 persistence.xml 文件中对应元素的值，也会覆盖提供者可能已经使用的缺省值。

下面列出了本规范定义的属性：

- Javax.persistence.provider——对应于 persistence.xml 中的 provider 元素。参见 5.2.1.4。
- Javax.persistence.transactionType——对应于 persistence.xml 中的 transaction-type 元素。参见 5.2.1.2 章节。
- Javax.persistence.jtaDataSource——对应于 persistence.xml 中的 jta-data-source 元素。参见 5.2.1.5 章节。
- Javax.persistence.nonJtaDataSource ——对应于 persistence.xml 中的 non-jta-data-source 元素。参见 5.2.1.5 章节。

提供商自己的属性也可以包含在 map 中。提供商必须忽略不能识别的属性。

提供商应当使用自己的属性命名空间（例如：com.acme.persistence.logging）。提供商不能使用 javax.persistence 及其子空间。javax.persistence 命名空间由本规范保留。

6.3 Javax.persistence.spi.PersistenceUnitInfo 接口

```
import javax.sql.DataSource;
/**
 * Interface implemented by the container and used by the
 * persistence provider when creating an EntityManagerFactory.
 */
public interface PersistenceUnitInfo {
/**
 * @return The name of the persistence unit.
*Correspondstothenameattributeinthepersistence.xmlfile.
 */
public String getPersistenceUnitName();
/**
 * @return The fully qualified name of the persistence provider
 * implementation class.
```

```
  * Corresponds to the <provider> element in the persistence.xml
  * file.
  */
public String getPersistenceProviderClassName();
/**
  * @return The transaction type of the entity managers created
  * by the EntityManagerFactory.
  * The transaction type corresponds to the transaction-type
  * attribute in the persistence.xml file.
  */
public PersistenceUnitTransactionType getTransactionType();
/**
  * @return The JTA-enabled data source to be used by the
  * persistence provider.
  * The data source corresponds to the <jta-data-source>
  * element in the persistence.xml file or is provided at
  * deployment or by the container.
  */
public DataSource getJtaDataSource();
/**
  * @return The non-JTA-enabled data source to be used by the
  * persistence provider for accessing data outside a JTA
  * transaction.
*Thedatasourcecorrespondstothenamed<non-jta-data-source>
  * element in the persistence.xml file or provided at
  * deployment or by the container.
  */
public DataSource getNonJtaDataSource();
```

```
/**
  * @return The list of mapping file names that the persistence
  * provider must load to determine the mappings for the entity
  * classes. The mapping files must be in the standard XML
  * mapping format, be uniquely named and be resource-loadable
  * from the application classpath.
  * Each mapping file name corresponds to a <mapping-file>
  * element in the persistence.xml file.
*/
public List<String> getMappingFileNames();
/**
  * Returns a list of URLs for the jar files or exploded jar
  * file directories that the persistence provider must examine
  * for managed classes of the persistence unit. Each URL
  * corresponds to a named <jar-file> element in the
  * persistence.xml file. A URL will either be a file:
  * URL referring to a jar file or referring to a directory
  * that contains an exploded jar file, or some other URL from
  * which an InputStream in jar format can be obtained.
  *
  * @return a list of URL objects referring to jar files or
  * directories.
  */
public List<URL> getJarFileUrls();
/**
  * Returns the URL for the jar file or directory that is the
  * root of the persistence unit. (If the persistence unit is
  * rooted in the WEB-INF/classes directory, this will be the
```

```
  * URL of that directory.)
  * The URL will either be a file: URL referring to a jar file
  * or referring to a directory that contains an exploded jar
  * file, or some other URL from which an InputStream in jar
  * format can be obtained.
  *
  * @return a URL referring to a jar file or directory.
  */
public URL getPersistenceUnitRootUrl();
/**
  * @return The list of the names of the classes that the
  * persistence provider must add it to its set of managed
  * classes. Each name corresponds to a named <class> element
  * in the persistence.xml file.
  */
public List<String> getManagedClassNames();
/**
  * @return Whether classes in the root of the persistence
  * unit that have not been explicitly listed are to be
  * included in the set of managed classes.
  * This value corresponds to the <exclude-unlisted-classes>
  * element in the persistence.xml file.
  */
public boolean excludeUnlistedClasses();
/**
  * @return Properties object. Each property corresponds
  * to a <property> element in the persistence.xml file
  */
```

```
public Properties getProperties();
/**
  * @return ClassLoader that the provider may use to load any
  * classes, resources, or open URLs.
  */
public ClassLoader getClassLoader();
/**
  * Add a transformer supplied by the provider that will be
  * called for every new class definition or class redefinition
  * that gets loaded by the loader returned by the
  * PersistenceUnitInfo.getClassLoader method. The transformer
  * has no effect on the result returned by the
  * PersistenceUnitInfo.getNewTempClassLoader method.
*Classesareonlytransformedoncewithinthesameclassloading
  * scope, regardless of how many persistence units they may be
  * a part of.
  *
  * @param transformer A provider-supplied transformer that the
  * Container invokes at class-(re)definition time
  */
public void addTransformer(ClassTransformer transformer);
/**
  * Return a new instance of a ClassLoader that the provider
  * may use to temporarily load any classes, resources, or
  * open URLs. The scope and classpath of this loader is
  * exactly the same as that of the loader returned by
*PersistenceUnitInfo.getClassLoader.Noneoftheclassesloaded
  * by this class loader will be visible to application
```

```
 * components. The provider may only use this ClassLoader
 * within the scope of the createContainerEntityManagerFactory
 * call.
 *
 * @return Temporary ClassLoader with same visibility as current
 * loader
 */
public ClassLoader getNewTempClassLoader();
}
```

Theenum javax.persistence.spi.PersistenceUnitTransactionType de?nes whether the entity managers created by the factory will be JTA or resource-local entity managers.

```
public enum PersistenceUnitTransactionType {
JTA,
RESOURCE_LOCAL
}
```

Javax.persistence.spi.ClassTransformer 接口由持久化提供者实现，为了提供者能实现在类加载时或在类重定义时转换实体和受管理类。

```
/**
 * A persistence provider supplies an instance of this
 * interface to the PersistenceUnitInfo.addTransformer
 * method. The supplied transformer instance will get
 * called to transform entity class files when they are
 * loaded or redefined. The transformation occurs before
 * the class is defined by the JVM.
 */
public interface ClassTransformer {
/**
```

```
  * Invoked when a class is being loaded or redefined.
  * The implementation of this method may transform the
  * supplied class file and return a new replacement class
  * file.
  *
  * @param loader The defining loader of the class to be
  * transformed, may be null if the bootstrap loader
  * @param className The name of the class in the internal form
  * of fully qualified class and interface names
  * @param classBeingRedefined If this is a redefine, the
  * class being redefined, otherwise null
  * @param protectionDomain The protection domain of the
  * class being defined or redefined
  * @param classfileBuffer The input byte buffer in class
  * file format - must not be modified
  * @return A well-formed class file buffer (the result of
  * the transform), or null if no transform is performed
  * @throws IllegalClassFormatException If the input does
  * not represent a well-formed class file
  */
byte[] transform(ClassLoader loader,
                 String className,
                 Class<?> classBeingRedefined,
                 ProtectionDomain protectionDomain,
                 byte[] classfileBuffer)
throws IllegalClassFormatException;
}
```

6.4 在 Java SE 环境下启动

在 Java SE 环境下，应用使用 Persistence.createEntityManagerFactory 方法来创建一个实体管理器工厂（注：在 Java EE 环境内可以支持使用这些 Java SE 的启动 API；但是不要求支持这种用法）。

提供者配置文件用于暴露提供者实现类给 Persistence 启动类，把作为候选提供者放在合适的位置以支持命名的持久化单元。

提供者通过创建一个名字为 javax.persistence.spi.PersistenceProvider 的文本文件并把它放在一个 jar 文件的 META-INF/services 目录下来提供提供者配置文件。文件的内容应当是实现了 javax.persistence.spi.PersistenceProvider 接口的类。

例如：

一个名字为 ACME 的持久化产品有一个名字为 acme.jar 的 JAR，它包含了持久化提供者的实现。这个 JAR 包含了提供者配置文件。

acme.jar

META-INF/services/javax.persistence.spi.PersistenceProvider

com.acme.PersistenceProvider

。。。

META-INF/services/javax.persistence.spi.PersistenceProvider 文件的内容只有实现类的名字：com.acme.PersistenceProvider。

持久化提供者的 jar 包和其他服务提供者一样可以用同一种方式安装或者被得到。例如，根据 JAR 文件规范扩展或者增加到应用的类路径。

Persistence 启动类会根据提供者配置文件定位所有的持久化提供者并依次调用他们的 createEntityManagerFactory()，直到正确的提供者返回 EntityManagerFactory。如果下面的结构返回 true，那么提供者就可以认为自己是匹配持久化单元的：

- 它的实现类已经在 persistence.xml 的 provider 元素中指定。
- 传入 createEntityManagerFactory 的 Map 中包含了 javax.persistence.provider 属性，并且这个属性的值是提供者的实现类。
- 在 persistence.xml 或属性 map 中没有为持久化单元指定提供者。

如果提供者和为命名的持久化单元不匹配，那么它在调用

createEntityManagerFactory 时必须返回 null。

6.4.1 Javax.persistence.Persistence 类

```
package javax.persistence;
import java.util.*;
...
/**
 * Bootstrap class that is used to obtain an
 * EntityManagerFactory.
 */
public class Persistence {
/**
 * Create and return an EntityManagerFactory for the
 * named persistence unit.
 *
 * @param persistenceUnitName The name of the persistence unit
 * @return The factory that creates EntityManagers configured
 * according to the specified persistence unit
 */
public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName) {...}
/**
 * Create and return an EntityManagerFactory for the
 * named persistence unit using the given properties.
 *
 * @param persistenceUnitName The name of the persistence unit
 * @param props Additional properties to use when creating the
 * factory. The values of these properties override any values
```

```
     * that may have been configured elsewhere.
     * @return The factory that creates EntityManagers configured
     * according to the specified persistence unit.
     */
    public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName, Map properties) {...}
    ...
    }
```

7 元数据注释符

本章和第 8 章定义本规范引入的元数据注释符。

在第 9 章定义的 XML Schema 为元数据注释符的使用提供了可选方案。

注释符都在 javax.persistence 包内。

7.1 Entity

Entity 注释符指明一个类是实体。这个注释符用于实体类。

name 注释符元素缺省是实体类的名字（不是全称）。名称用于在查询中引用这个实体。这个名称不能是 EJB QL 的保留字。

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Entity {
    String name() default "";
}
```

7.2 回调注释符

EntityListeners 注释符指定实体类和被影射的超类的回调监听器。EntityListeners 可以用于实体类和被映射的超类。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface EntityListeners {
   Class[] value();
}
```

ExcludeSuperClassListeners 注释符指定从实体类（或被映射超类）和它的子类中排除那些超类监听器。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface ExcludeSuperclassListeners {
}
```

ExcludeDefaultListeners 指定从实体类（或被映射超类）和它的子类中排除缺省监听器。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface ExcludeDefaultListeners {
}
```

下面的注释符用于指定对应生命周期事件的回调方法。这些注释符可以用于任意实体类、被映射超类或实体监听器类的方法。

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PrePersist {}
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostPersist {}
@Target({METHOD}) @Retention(RUNTIME)
public @interface PreRemove {}
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostRemove {}
@Target({METHOD}) @Retention(RUNTIME)
public @interface PreUpdate {}
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostUpdate {}
```

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostLoad {}
```

7.3 用于查询的注释符

7.3.1 NamedQuery

NamedQuery 注释符用于指定一个使用 Java 持久化查询语言的命名查询。当使用 EntityManager 的创建查询对象的方法时，元素 name 用于指向一个查询。NamedQuery 和 NamedQueries 注释符可以用于实体类和被映射超类。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedQuery {
String name();
String query();
QueryHint[] hints() default {};
}
@Target({}) @Retention(RUNTIME)
public @interface QueryHint {
String name();
String value();
}
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedQueries {
NamedQuery[] value ();
}
```

7.3.2 NamedNativeQuery

NamedNativeQuery 用于指定一个本地 SQL 命名查询。当使用 EntityManager 的创建查询对象的方法时，元素 name 用于指向那个查询。元素 resultClass 指向

结果类；元素 resultSetMapping 的值是定义在元数据中的 SqlResultSetMapping 的名字。NamedNativeQuery 和 NamedNativeQueries 可以用于实体或被映射超类。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedNativeQuery {
String name();
String query();
QueryHint[] hints() default {};
Class resultClass() default void.class;
StringresultSetMapping()default"";//nameofSqlResultSetMap-
ping
}
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedNativeQueries {
NamedNativeQuery[] value ();
}
```

7.3.3 用于 SQL 查询结果集映射的注释符

SqlResultMapping 注释符用于指定本地 SQL 查询结果的映射。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface SqlResultSetMapping {
String name();
EntityResult[] entities() default {};
ColumnResult[] columns() default {};
}
@Target({TYPE}) @Retention(RUNTIME)
public @interface SqlResultSetMappings {
SqlResultSetMapping[] value();
}
```

元素 name 是结果集映射的名字，这个名字用于在 Query API 的方法内引用这个结果集。元素 entities 和 columns 用于指定实体映射并对应结果值。

```
@Target({}) @Retention(RUNTIME)
public @interface EntityResult {
Class entityClass();
FieldResult[] fields() default {};
String discriminatorColumn() default "";
}
```

元素 entityClass 指定结果集的类。

元素 fields 用于映射 SELECT 中的列和实体的属性或字段。

元素 discriminatorColumn 用于指定 SELECT 中用于区分实体类型的列的列名（或别名）。（译者注：对于继承关系的映射，父类和子类映射到一个数据表的情况）。

```
@Target({}) @Retention(RUNTIME)
public @interface FieldResult {
String name();
String column();
}
```

元素 name 是类的持久化字段或属性的名字。

在这些注释符内使用的列名都指向 SELECT 语句中的列名——也就是说，如果可以的话就是列的别名。

```
@Target({}) @Retention(RUNTIME)
public @interface ColumnResult {
String name();
}
```

7.4 引用 EntityManager 和 EntityManagerFactory

这些注释符用于表达对实体管理器和实体管理器工厂的依赖。

7.4.1 PersistenceContext

PersistenceContext 用于表达依赖容器管理的实体管理器持久化上下文。

元素 name 指的是在环境的引用上下文内中可以获得的实体管理器，但当使用依赖注入时不需要使用 name。

可选元素 unitName 指向持久化单元的名字。如果指定了 unitName 元素，那么它的名字必须和在 JNDI 中的实体管理器的持久化单元的名字一致。

元素 type 指定使用的是事务范围的持久化上下文，还是扩展的持久化上下文。如果没有指定 type，则使用事务范围的持久化上下文。

可选的元素 properties 可以用于指定容器或持久化提供者的属性。特定提供商的属性可以包含在属性集中，当容器创建实体管理器时，这些属性被容器传递到持久化提供商。提供商不能识别的属性必须被忽略。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PersistenceContext{
String name() default "";
String unitName() default "";
PersistenceContextType type default TRANSACTION;
PersistenceProperty[] properties() default {};
}
public enum PersistenceContextType {
   TRANSACTION,
   EXTENDED
}
@Target({}) @Retention(RUNTIME)
public @interface PersistenceProperty {
String name();
String value();
}
```

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface PersistenceContexts{
   PersistenceContext[] value();
}
```

7.4.2 PersistenceUnit

PersistenceUnit 用于表达依赖的实体管理器工厂。

元素 name 指向可以在环境引用上下文获得的实体管理器工厂。当使用依赖注入时，不需要指定 name。

可选元素 unitName 指向在 persistence.xml 中定义的持久化单元的名字。如果指定 unitName，则可以在 JNDI 中获得的实体管理器工厂的持久化单元必须和这个名字一致。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PersistenceUnit{
String name() default "";
String unitName() default "";
}
@Target(TYPE) @Retention(RUNTIME)
public @interface PersistenceUnits{
   PersistenceUnit[] value();
}
```

8 O/R 映射的元数据

由应用表达的 O/R 映射的元数据表是应用域对象协议的一部分。

O/R 映射元数据是应用域对象协议的一部分。它表达了映射应用域的实体和关系到数据库的需求和期望。和应用域模型一致的基于数据库 Schema 书写的查询（特殊情况下是 SQL 查询）依赖于由 O/R 映射的元数据表达的映射关系。这

个规范的实现必须假定应用是依赖于 O/R 映射元数据，并且确保被这些映射表达的语义和需求是可观察到的。

允许但不要求规范的实现支持生成 DDL。可移植的应用不应当依赖于 DDL 生成。

8.1 O/R 映射的注解符

这些注解符在 javax.persistence 包内。

XML 元数据可以作为注解符的替换方案，或者可以用于重载或参数化注解符，这在第 10 章描述。

8.1.1 Table 注解符

Table 注解符指定注解实体的主表。从表可以用 SecondaryTable 或 SecondaryTables 来指定。

表 4 列出了指定 Table 注解符时会用到的注解符元素，以及这些元素的缺省值。

如果一个实体类没有指定 Table 注解符，那么将使用表 4 中定义的缺省值。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Table {
     String name() default "";
     String catalog() default "";
     String schema() default "";
     UniqueConstraint[] uniqueConstraints() default {};
}
```

Table 4 Table Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the table	Entity name
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect. These constraints apply in addition to any constraints specified by the Column and JoinColumn annotations and constraints entailed by primary key mappings.	No additional constraints

例如：

```
@Entity
@Table(name="CUST", schema="RECORDS")
public class Customer { ... }
```

8.1.2 SecondaryTable 注解符

SecondaryTable 用于指定实体类的从表。指定一个或多个从表说明实体数据被存在多个表中。

表 5 列出了可以在 SecondaryTable 注解符中指定的注解符元素以及这些元素的缺省值。

如果没有指定 SecondaryTable，那么就假定实体的所有持久化属性或字段都被映射到主表。如果没有指定主键关联列，那么关联列就假定为主表的主键列，并且和主键列有相同的名字和类型。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface SecondaryTable {
     String name();
     String catalog() default "";
     String schema() default "";
     PrimaryKeyJoinColumn[] pkJoinColumns() default {};
     UniqueConstraint[] uniqueConstraints() default {};
}
```

Table 5 SecondaryTable Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the table.	
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user
PrimaryKeyJoin-Column[]	pkJoinColumns	(Optional) The columns that are used to join with the primary table.	Column(s) of the same name(s) as the primary key column(s) in the primary table
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are typically only used if table generation is in effect. These constraints apply in addition to any constraints specified by the Column and Join-Column annotations and constraints entailed by primary key mappings.	No additional constraints

例 1：只有一个主键列的单一从表

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(name="CUST_DETAIL",
  pkJoinColumns=@PrimaryKeyJoinColumn(name="CUST_ID"))
public class Customer { ... }
```

例 2：有多个主键列的单一从表

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(name="CUST_DETAIL",
  pkJoinColumns={
    @PrimaryKeyJoinColumn(name="CUST_ID"),
    @PrimaryKeyJoinColumn(name="CUST_TYPE")})
public class Customer { ... }
```

8.1.3 SecondaryTables 注解符

SecondaryTables 用于指定多个从表。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface SecondaryTables {
    SecondaryTable[] value();
}
```

例 1：假定主键列和所有表的主键名称相同的多个从表。

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables({
    @SecondaryTable(name="EMP_DETAIL"),
    @SecondaryTable(name="EMP_HIST")
})
public class Employee { ... }
```

例 2：假定主键列和所有表的主键名称不相同的多个从表。

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables({
    @SecondaryTable(name="EMP_DETAIL",
        pkJoinColumns=@PrimaryKeyJoinColumn(name="EMPL_ID")),
    @SecondaryTable(name="EMP_HIST",
        pkJoinColumns=@PrimaryKeyJoinColumn(name="EMPLOYEE_ID"))
})
public class Employee { ... }
```

8.1.4 UniqueConstraint 注解符

UniqueConstraint 用于指定在为主表或从生成的 DDL 中的唯一约束。

表 6 列出了可以在 UniqueConstraint 中指定的注解符元素。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface UniqueConstraint {
     String[] columnNames();
}
```

Table 6 UniqueConstraint Annotation Elements

Type	Name	Description	Default
String[]	columnNames	(Required) An array of the column names that make up the constraint.	

例子：

```
@Entity
@Table(
     name="EMPLOYEE",
     uniqueConstraints=
          {@UniqueConstraint(columnNames={"EMP_ID", "EMP_NAME"})}
)
public class Employee { ... }
```

8.1.5 Column 注解符

Column 用于为持久化属性或字段指定映射列。

表 7 列出了可以在 Column 注解符内指定的注解符元素以及这些元素的缺省值。

如果没有指定 Column，那么使用表 7 内的缺省值。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
     String name() default "";
     boolean unique() default false;
     boolean nullable() default true;
     boolean insertable() default true;
     boolean updatable() default true;
     String columnDefinition() default "";
     String table() default "";
     int length() default 255;
     int precision() default 0; // decimal precision
     int scale() default 0; // decimal scale
}
```

Table 7 Column Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the column.	The property or field name
boolean	unique	(Optional) Whether the property is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint is only a single field. This constraint applies in addition to any constraint entailed by primary key mapping and to constraints specified at the table level.	false
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL to create a column of the inferred type.
String	table	(Optional) The name of the table that contains the column. If absent the column is assumed to be in the primary table.	Column is in primary table.
int	length	(Optional) The column length. (Applies only if a string-valued column is used.)	255
int	precision	(Optional) The precision for a decimal (exact numeric) column. (Applies only if a decimal column is used.)	0 (Value must be set by developer.)
int	scale	(Optional) The scale for a decimal (exact numeric) column. (Applies only if a decimal column is used.)	0

例 1：

```
@Column(name="DESC", nullable=false, length=512)
public String getDescription() { return description; }
```

例 2：

```
@Column(name="DESC",
    columnDefinition="CLOB NOT NULL",
    table="EMP_DETAIL")
@Lob
public String getDescription() { return description; }
```

例 3：

```
@Column(name="ORDER_COST", updatable=false, precision=12, scale=2)
public BigDecimal getCost() { return cost; }
```

8.1.6 JoinColumn 注解符

JoinColumn 用于为连接一个实体关联指定映射列。

表 8 列出了可以在 JoinColumn 注解符中指定的注解符元素以及它们的缺省值。

如果没有指定 JoinColumn 注解符，那么假定连接列是单一的列。连接列的缺省值如下所述。

注解符 name 元素定义外键列的名称。其余的注解符元素（除了 refernecedColumnName）指的是列，和 Column 注解符的语义一样。

如果有一个单一的连接列，且如果没有指定 name 元素，那么关联列的名字的格式如下：引用实体的引用关系属性或字段的名字+“_”+被引用的主键列的名字。如果在实体内没有这样的关系属性或字段（如使用关联表），那么关联列的名字的格式是：实体的名字+“_”+被引用的主键列的名字。

如果没有指定 refernecedColumnName，外键假定就是被引用表的主键。

如果有多个连接列，那么必须用 JonColumns 注解符为每一个连接列指定一个 JoinColumn。同时也必须为每一个 JoinColumn 指定元素 name 和 refernecedColumnName。

```
@Target({METHOD, FIELD})  @Retention(RUNTIME)
public @interface JoinColumn {
     String name() default "";
     String referencedColumnName() default "";
     boolean unique() default false;
     boolean nullable() default true;
     boolean insertable() default true;
     boolean updatable() default true;
     String columnDefinition() default "";
     String table() default "";
}
```

是否支持被引用列不是被应用表的主键在这个版本中是可选的，但在下一个版本中会要求至此。

Table 8 JoinColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the foreign key column. The table in which it is found depends upon the context. If the join is for a OneToOne or ManyToOne mapping, the foreign key column is in the table of the source entity. If the join is for a ManyToMany, the foreign key is in a join table.	(Default only applies if a single join column is used.) The concatenation of the following: the name of the referencing relationship property or field of the referencing entity; "_"; the name of the referenced primary key column. If there is no such referencing relationship property or field in the entity, the join column name is formed as the concatenation of the following: the name of the entity; "_"; the name of the referenced primary key column.
String	referencedColumnName	(Optional) The name of the column referenced by this foreign key column. When used with relationship mappings, the referenced column is in the table of the target entity. When used inside a JoinTable annotation, the referenced key column is in the entity table of the owning entity, or inverse entity if the join is part of the inverse join definition.	(Default only applies if single join column is being used.) The same name as the primary key column of the referenced table.
boolean	unique	(Optional) Whether the property is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint is only a single field. It is not necessary to explicitly specify this for a join column that corresponds to a primary key that is part of a foreign key.	false
boolean	nullable	(Optional) Whether the foreign key column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL for the column.

Type	Name	Description	Default
String	table	(Optional) The name of the table that contains the column. If a table is not specified, the column is assumed to be in the primary table of the applicable entity.	Column is in primary table.

例子：

```
@ManyToOne
@JoinColumn(name="ADDR_ID")
public Address getAddress() { return address; }
```

8.1.7 JoinColumns 注解符

通过 JoinColumns 注解符可以支持组合键。这可以为同一关系或表关联组合多个 JoinColumn。

当使用 JoinColumns 注解符时，必须为每一个 JoinColumn 指定元素 name 和 referencedColumnName。

```
@Target({TYPE, METHOD, FIELD})  @Retention(RUNTIME)
public @interface JoinColumns {
    JoinColumn[] value();
}
```

例子：

```
@ManyToOne
@JoinColumns({
    @JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
    @JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
})
public Address getAddress() { return address; }
```

8.1.8 Id 注解符

Id 注解符指定实体的主键属性或字段。Id 注解符可以用于实体或映射的超类上。

缺省情况下，属性的映射列的格式假定为主表的主键。如果没有指定 Column 注解符，那么主键列的名字假定和唯一标识属性或字段的名字是相同的。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Id {}
```

例子：

```
@Id
public Long getId() { return id; }
```

8.1.9 GeneratedValue 注解符

GeneratedValue 注解符提供主键的生成策略规范。GeneratedValue 可以应用到实体的主键属性或字段，也可以应用到和 Id 注解符关联的被映射的超类上。（可

移植的应用不应当在其他的字段上使用 GeneratedValue）

表 9 列出了可以在 GeneratedValue 中指定的注解符元素以及它们的缺省值。

主键生成的类型在 GenerationType 枚举中定义：

```
public enum GenerationType { TABLE, SEQUENCE, IDENTITY, AUTO };
```

- TABLE 策略指的是持久化提供商应当使用后台数据库表来指派唯一标识。
- SEQUENCE 和 IDENTITY 策略分别指的是使用数据库序列或唯一标识列。
- AUTO 策略指的是持久化提供商根据特定的数据库来确定一个合适的策略。AUTO 策略可以认为数据库资源存在，或者企图去创建它。提供商可以提供在不支持 Schema 生成或在运行时不能创建 Schema 资源的情况下如何去创建这样资源的文档。
- NONE 策略指的是不使用持久化提供商的主键生成，应用负责分派主键。

本规范没有定义这些策略的确切行为。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface GeneratedValue {
    GenerationType strategy() default AUTO;
    String generator() default "";
}
```

Table 9 GeneratedValue Annotation Elements

Type	Name	Description	Default
Generation-Type	strategy	(Optional) The primary key generation strategy that the persistence provider must use to generate the annotated entity primary key.	GenerationType.AUTO
String	generator	(Optional) The name of the primary key generator to use as specified in the SequenceGenerator or TableGenerator annotation.	Default id generator supplied by persistence provider.

例 1：

```
@Id
@GeneratedValue(strategy=SEQUENCE, generator="CUST_SEQ")
@Column(name="CUST_ID")
public Long getId() { return id; }
```

例 2：

```
@Id
@GeneratedValue(strategy=TABLE, generator="CUST_GEN")
@Column(name="CUST_ID")
Long id;
```

8.1.10 AttributeOverride 注解符

AttributeOverride 用于重载基本的（不管是显式的还是缺省的）属性/字段或 Id 属性/字段的映射。

AttributeOverride 可以用于一个继承超类的实体或一个被嵌入的字段/属性来重载定义在父类或可嵌入类的基本映射信息。如果没有指定 AttributeOverride，被映射的列名和原始映射的列名相同。

表 10 列出了可以在 AttributeOverride 注解符中指定的注解符元素。

column 元素指的是包含了这个注解符的类对应的数据库表中的列名。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AttributeOverride {
    String name();
    Column column();
}
```

Table 10 AttributeOverride Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the property whose mapping is being overridden if property-based access is being used, or the name of the field if field-based access is used.	
Column	column	(Required) The column that is being mapped to the persistent attribute. The mapping type will remain the same as is defined in the embeddable class or mapped superclass.	

例子：

```
@MappedSuperclass
public class Employee {

    @Id protected Integer id;
    @Version protected Integer version;
    protected String address;

    public Integer getId() { ... }
    public void setId(Integer id) { ... }
    public String getAddress() { ... }
    public void setAddress(String address) { ... }
}

@Entity
@AttributeOverride(name="address", column=@Column(name="ADDR"))
public class PartTimeEmployee extends Employee {
     // address field mapping overridden to ADDR
    protected Float wage();
    public Float getHourlyWage() { ... }
    public void setHourlyWage(Float wage) { ... }
}
```

8.1.11 AttributeOverrides 注解符

AttributeOverrides 用于重载多个属性或字段的映射。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AttributeOverrides {
  AttributeOverride[] value();
}
```

例子：

```
@Embedded
@AttributeOverrides({
     @AttributeOverride(name="startDate",
                        column=@Column(name="EMP_START")),
    @AttributeOverride(name="endDate",
                        column=@Column(name="EMP_END"))
})
public EmploymentPeriod getEmploymentPeriod() { ... }
```

8.1.12 AssociationOverride 注解符

AssociationOverride 用于重载表示实体关系的属性或字段的多对一或一对一映射。

AssociationOverride 可以用于一个继承超类的实体，在这个超类内定义了多对一或一对多的映射。如果没有指定 AssociationOverride，那么关联列被映射的

名字和原始映射相同。

表 11 列出了可以在 AssociationOverride 中指定的注解符元素。

元素 joinColumns 指的是包含这个注解符的类对应的数据库表中的列名。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AssociationOverride {
     String name();
     JoinColumn[] joinColumns();
}
```

Table 11 AssociationOverride Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the relationship property whose mapping is being overridden if property-based access is being used, or the name of the relationship field if field-based access is used.	
JoinColumn[]	joinColumns	(Required) The join column that is being mapped to the persistent attribute. The mapping type will remain the same as is defined in the mapped superclass.	

例子：

```
@MappedSuperclass
public class Employee {

    @Id protected Integer id;
    @Version protected Integer version;
    @ManyToOne
    protected Address address;

    public Integer getId() { ... }
    public void setId(Integer id) { ... }
    public Address getAddress() { ... }
    public void setAddress(Address address) { ... }
}

@Entity
@AssociationOverride(name="address",
                     joinColumns=@JoinColumn(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {
     // address field mapping overridden to ADDR_ID fk
    @Column(name="WAGE")
    protected Float hourlyWage;
    public Float getHourlyWage() { ... }
    public void setHourlyWage(Float wage) { ... }
}
```

8.1.13 AssociationOverrides 注解符

用于重载多个多对一或一对一关系属性或字段的映射。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AssociationOverrides {
  AssociationOverride[] value();
}
```

例子：

```
@MappedSuperclass
public class Employee {

    @Id protected Integer id;
    @Version protected Integer version;
    @ManyToOne protected Address address;
    @OneToOne protected Locker locker;

    public Integer getId() { ... }
    public void setId(Integer id) { ... }
    public Address getAddress() { ... }
    public void setAddress(Address address) { ... }
    public Locker getLocker() { ... }
    public void setLocker(Locker locker) { ... }

}

@Entity
@AssociationOverrides({
     @AssociationOverride(name="address",
                          joinColumns=@JoinColumn("ADDR_ID")),
    @AttributeOverride(name="locker",
                          joinColumns=@JoinColumn("LCKR_ID"))})

public PartTimeEmployee { ... }
```

8.1.14 EmbeddedId 注解符

EmbeddedId 用于表示一个实体类或超类的一个持久化属性或字段是组合主键，这个主键是一个可嵌入类。可嵌入类必须注解为 Embeddable。（注意，Id 注解符不使用在可嵌入类中）

当使用 EmbeddedId 时，只能有一个 EmbeddedId，且不能有 Id 注解符。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface EmbeddedId {}
```

例子：

```
@EmbeddedId
protected EmployeePK empPK;
```

8.1.15 IdClass 注解符

IdClass 用于为实体类或超类指定一个组合主键类，这个类映射到实体的多

个属性或字段。

在主键类中的主键字段或属性的名字和实体类中的主键字段或属性必须一一对应，并且他们的类型也必须是一致的。参考 1.1.4 章节，“主键和实体唯一标识”。

Id 注解符也必须应用到这些对应的字段或属性上。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface IdClass {
     Class value();
}
```

例子：

```
@IdClass(com.acme.EmployeePK.class)
@Entity
public class Employee {
     @Id String empName;
     @Id Date birthDay;
...
}
```

8.1.16 Transient 注解符

Transient 用于指定实体类、超类或可嵌入类的属性或字段不是持久化的。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Transient {}
```

例子：

```
@Entity
public class Employee {
     @Id int id;
     @Transient User currentUser;
...
}
```

8.1.17 Version 注解符

Version 指定实体类的版本属性作为乐观锁的值。这用于执行 merge 操作时确保完成性和乐观的并发控制。

每个类只有一个 Version 属性/字段；使用多个 Version 属性/字段的应用可能是不可移植的。

Version 属性应当被映射到实体类的主表上，映射到其他表上的应用是不可移植的。

通常情况下，被 Version 注解符指定的字段或属性不应当被应用更新。（特殊情况，参考 3.10）

Version 属性可以是以下类型：int，Integer，short，Short，long，Long，Timestamp。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Version {}
```

例子：

```
@Version
@Column("OPTLOCK")
protected int getVersionNum() { return versionNum; }
```

8.1.18 Basic 注解符

Basic 是映射到数据库列的最简单的类型。它可以应用到下列类型的任何持久化属性或实例变量上：java 原始类型及其包装类型，java.lang.String，java.math.BigInteger，java.math.BigDecimal，java.util.Date，java.util.Calendar，java.sql.Date，java.slq.Time，java.sql.Timestamp，byte[]，Byte[]，char[]，Character[]，枚举，和任何实现了序列化接口的类型。和 1.1.6 章节描述的一样，对这些类型的持久化字段或属性使用 Basic 注解符是可选的。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

表 12 列出了可以在 Basic 注解符内指定的注解符元素以及它们的缺省值。

FetchType 枚举定义了从数据库获取数据的策略：

```
public enum FetchType { LAZY, EAGER };
```

- EAGER 策略是持久化提供商运行时应当提供的饥渴提取数据策略。
- LAZY 策略是持久化提供商运行时的缺省提供的提取数据策略。

实现可以在缺省为 LAZY 提取策略时改变为饥渴提取数据。对 Basic 属性，懒惰提取只可以被应用到那些总是通过基于属性获取方法的属性上。

Optional 元素能被用作指定字段或属性的值是否可以为 null。这不考虑原始类型，原始类型被认为是必须有值的。

Table 12 Basic Annotation Elements

Type	Name	Description	Default
FetchType	fetch	(Optional) Whether the value of the field or property should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the value must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	EAGER
boolean	optional	(Optional) Whether the value of the field or property may be null. This is a hint and is disregarded for primitive types; it may be used in schema generation.	true

例 1：

```
@Basic
protected String name;
```

例 2：

```
@Basic(fetch=LAZY)
protected String getName() { return name; }
```

8.1.19 Lob 注解符

Lob 用于指定持久化字段或属性是个大对象类型。当映射到数据库的 Lob 类型时，可移植的应用应当使用 Lob 注解符。Lob 注解符可以和 Basic 注解符联合使用。Lob 类型可以是二进制类型，也可以是字符类型。Lob 类型从持久化字段或属性的类型推断出来，且除了字符串和基于字符的类型外，缺省值都是 Blob。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Lob {
}
```

例 1：

```
@Lob @Basic(fetch=EAGER)
@Column(name="REPORT")
protected String report;
```

例 2：

```
@Lob @Basic(fetch=LAZY)
@Column(name="EMP_PIC", columnDefinition="BLOB NOT NULL")
protected byte[] pic;
```

8.1.20 Temproal 注解符

Temproal 注解符必须用于指定 java.util.Date 和 java.util.Calendar 类型的持久化字段或属性。它只能用于指定这些类型的字段或属性。

Temproal 可以和 Basic 联合使用。

TemproalType 枚举定义这些临时类型的映射。

```
public enum TemporalType {
    DATE, //java.sql.Date
    TIME, //java.sql.Time
    TIMESTAMP //java.sql.Timestamp
}


@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Temporal {
    TemporalType value();
}
```

表 13 列出了可以在 Temporal 注解符内指定的注解符元素以及它们的缺省值。

Table 13 Temporal Annotation Elements

Type	Name	Description	Default
TemporalType	value	The type used in mapping java.util.Date or java.util.Calendar.	

例子：

```
@Temporal(DATE)
protected java.util.Date endDate;
```

8.1.21 Enumerated 注解符

Enumerated 指定一个持久化字段应当作为枚举类型存储。Enumerated 可以和 Basic 注解符联合使用。

枚举可以被映射成字符串或整数。EnumType 定义了枚举类型映射。

```
public enum EnumType {
    ORDINAL,
    STRING
}
```

如果没有指定枚举类型或没有使用 Enumerated，那么枚举类型假定是 ORDINAL。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Enumerated {
    EnumType value() default ORDINAL;
}
```

表 14 列出了可以在 Enumerated 注解符中指定的注解符元素以及他们的缺省值。

Table 14 Enumerated Annotation Elements

Type	Name	Description	Default
EnumType	value	(Optional) The type used in mapping an enum type.	ORDINAL

例子：

```
public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}

public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}

@Entity public class Employee {
    ...
    public EmployeeStatus getStatus() {...}

    @Enumerated(STRING)
    public SalaryRate getPayScale() {...}
    ...
}
```

如果 status 属性被映射成整型列，并且 payscale 属性被映射成 varchar 类型的列，那么一个有 PART_TIME 的 status 和 JUNIOR 的 pay rate 将分别被存储为 STATUS=1 和 PAYSCALE=”JUNIOR”。

8.1.22 ManyToOne 注解符

ManyToOne 定义一个到有多对一关联的实体类的单值关系。由于可以从被引用的对象类型推断出目标实体，所以，通常情况下，不需要显式指定目标实体。

表 15 列出了 ManyToOne 注解符内可以指定的注解符元素以及它们的缺省值。

Cascade 元素指定会繁衍到关联实体的级联操作集合。级联操作的类型由 CascadeType 枚举定义：

```
public enum CascadeType { ALL, PERSIST, MERGE, REMOVE, REFRESH};
```

Cascade=ALL 等价于 cascade={PERSIST，MERGE，REMOVE，REFRESH}。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

当使用 EAGER 策略时，持久化提供商运行时应当是即时获取关联实体。LAZY 策略是缺省的策略。实现必须允许能将缺省策略改为 EAGER 策略。

Table 15 ManyToOne Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association.	The type of the field or property that stores the association.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	EAGER
boolean	optional	(Optional) Whether the association is optional. If set to false then a non-null relationship must always exist.	true

例子：

```
@ManyToOne(optional=false)
@JoinColumn(name="CUST_ID", nullable=false, updatable=false)
public Customer getCustomer() { return customer; }
```

8.1.23 OneToOne 注解符

OneToOne 定义了到另一个实体的单值关联，这个实体有一对一多样性。由于可以从被引用的对象类型推断出目标实体，所以，通常情况下，不需要显式指定目标实体。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
}
```

表 16 列出了可以在 OneToOne 注解符内指定的注解符元素。

Table 16 OneToOne Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association.	The type of the field or property that stores the association.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	EAGER
boolean	optional	(Optional) Whether the association is optional. If set to false then a non-null relationship must always exist.	true
String	mappedBy	(Optional) The field that owns the relationship. The mappedBy element is only specified on the inverse (non-owning) side of the association.	

Type	Name	Description	Default
CascadeType[]	cascade	(Optional) The operations that should be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Hint to the implementation as to whether the association should be lazy loaded or eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entity should be eagerly fetched.	EAGER
boolean	optional	(Optional) Whether the association is optional. If set to false then a non-null relationship must always exist.	true
String	mappedBy	(Optional) The field that owns the relationship. The mappedBy element is only specified on the inverse (non-owning) side of the association.	

例 1：使用外键列的一对一关联

在 Customer 类上：

```
@OneToOne(optional=false)
@JoinColumn(
    name="CUSTREC_ID", unique=true, nullable=false, updatable=false)
public CustomerRecord getCustomerRecord() { return customerRecord; }
```

在 CustomerRecord 类上：

```
@OneToOne(optional=false, mappedBy="customerRecord")
public Customer getCustomer() { return customer; }
```

例 2：假定源和目标共享同一个主键值的一对一关联。

在 Employee 类上：

```
@Entity
public class Employee {
     @Id Integer id;

     @OneToOne @PrimaryKeyJoinColumn
     EmployeeInfo info;
     ...
}
```

在 EmployeeInfo 类上：

```
@Entity
public class EmployeeInfo {
     @Id Integer id;
     ...
}
```

8.1.24 OneToMany 注解符

OneToMany 定义多值关联。

如果用泛型指定了集合内的元素类型，那么不需要指定关联目标实体类型，否则必须指定目标实体类。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

表 17 列出了可以在 OneToMany 注解符中指定的注解符元素以及它们的缺省值。

Table 17 OneToMany Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association. Optional only if the collection property is defined using Java generics. Must be specified otherwise.	The parameterized type of the collection when defined using generics.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the associated entities must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime.	LAZY
String	mappedBy	The field that owns the relationship. Required unless the relationship is unidirectional.	

单向的一对多关系的缺省 schema 层的映射使用一个关联表，在 1.1.8.5 章节中描述。单向一对多关系也可以用一对多外键映射来实现，然而，在这个版本中不要求支持。对一对多关系想使用外键映射策略的应用应当使用双向关系确保可移植性。

例 1：用泛型的一对多关系

在 Customer 类上：

```
@OneToMany(cascade=ALL, mappedBy="customer")
public Set<Order> getOrders() { return orders; }
```

在 Order 类型上：

```
@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
public Customer getCustomer() { return customer; }
```

例 2：不使用泛型的一对多关系

在 Customer 类中：

```
@OneToMany(targetEntity=com.acme.Order.class, cascade=ALL,
mappedBy="customer")
public Set getOrders() { return orders; }
```

在 Order 类中：

```
@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
public Customer getCustomer() { return customer; }
```

8.1.25 JoinTable 注解符

JoinTable 用于关系的映射。在多对多关系的拥有者端指定，或者在单向一对多关系指定。

表 18 列出可以在 JoinTable 注解符中指定的注解符元素以及它们的缺省值。

如果没有指定 JoinTable，则使用注解符元素的缺省值。

关联表的名字假定是：用下划线连接被关联主表的表名，拥有者的表名放在首位。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface JoinTable {
     String name() default "";
     String catalog() default "";
     String schema() default "";
     JoinColumn[] joinColumns() default {};
     JoinColumn[] inverseJoinColumns() default {};
     UniqueConstraint[] uniqueConstraints() default {};
}
```

Table 18 JoinTable Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the join table.	The concatenated names of the two associated primary entity tables, separated by an underscore.
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user.
JoinCol-umn[]	joinColumns	(Optional) The foreign key columns of the join table which reference the primary table of the entity owning the association (i.e. the owning side of the association).	The same defaults as for JoinColumn.
JoinCol-umn[]	inverseJoinColumns	(Optional) The foreign key columns of the join table which reference the primary table of the entity that does not own the association (i.e. the inverse side of the association).	The same defaults as for JoinColumn.

Type	Name	Description	Default
UniqueCon-straint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect.	No additional constraints

例子：

```
@JoinTable(
     name="CUST_PHONE",
     joinColumns=
          @JoinColumn(name="CUST_ID", referencedColumnName="ID"),
     inverseJoinColumns=
          @JoinColumn(name="PHONE_ID", referencedColumnName="ID")
)
```

8.1.26 ManyToMany 注解符

ManyToMany 定义多对多关系的多值关联。如果用泛型指定了集合内的元素类型，那么不需要指定关联目标实体类型，否则必须指定目标实体类。

每一个多对多关联有两端，拥有者端和非拥有者或反向端。在拥有者端指定关联表。如果关联是双向的，另一个端也可以被指派为拥有者端。

OneToMany 注解符的注解符元素同样可以应用到 ManyToMany 注解符。

表 17 列出了这些注解符元素以及它们的缺省值。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToMany {
     Class targetEntity() default void.class;
     CascadeType[] cascade() default {};
     FetchType fetch() default LAZY;
     String mappedBy() default "";
}
```

例 1：

在 Customer 类中：

```
@ManyToMany
@JoinTable(name="CUST_PHONES")
public Set<PhoneNumber> getPhones() { return phones; }
```

在 PhoneNumber 类中：

```
@ManyToMany(mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

例 2：

在 Customer 类中：

```
@ManyToMany(targetEntity=com.acme.PhoneNumber.class)
public Set getPhones() { return phones; }
```

在 PhoneNumber 类中：

```
@ManyToMany(targetEntity=com.acme.Customer.class, mappedBy="phones")
public Set getCustomers() { return customers; }
```

例 3：

在 Customer 类：

```
@ManyToMany
@JoinTable(
     name="CUST_PHONE",
     joinColumns=
          @JoinColumn(name="CUST_ID", referencedColumnName="ID"),
     inverseJoinColumns=
          @JoinColumn(name="PHONE_ID", referencedColumnName="ID")
)
public Set<PhoneNumber> getPhones() { return phones; }
```

在 PhoneNumber 类中：

```
@ManyToMany(mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

8.1.27 MapKey 注解符

MapKey 用于指定 java.util.Map 类型关联的 map 主键。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface MapKey {
  String name() default "";
}
```

元素 name 指派关联实体的持久化字段或属性的名称，这个关联实体用于做 map 主键。如果没有指定 name，将使用主键作为 map 主键。如果主键是组合主键，且映射为 IdClass，那么主键类的实例用作 map 主键。

如果一个不是主键的持久化字段或属性被用作 map 主键，那么它应当有一个唯一约束。

例 1：

```
@Entity
public class Department {
        ...
        @OneToMany(mappedBy="department")
        @MapKey(name="empId")
        public Map<Integer, Employee> getEmployees() {... }
        ...
}

@Entity
public class Employee {
        ...
        @Id Integer getEmpid() { ... }

        @ManyToOne
        @JoinColumn(name="dept_id")
        public Department getDepartment() { ... }
        ...
}
```

例 2：

```
@Entity
public class Department {
     ...
     @OneToMany(mappedBy="department")
     @MapKey(name="empPK")
     public Map<EmployeePK, Employee> getEmployees() {... }
     ...
 }
@Entity
public class Employee {
     @EmbeddedId public EmployeePK getEmpPK() { ... }
     ...
     @ManyToOne
@JoinColumn(name="dept_id")
     public Department getDepartment() { ... }
     ...
}

@Embeddable
public class EmployeePK {
    String name;
    Date bday;
}
```

8.1.28 OrderBy 注解符

OrderBy 用于指定在获取关联时确定关联值集合元素的顺序。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OrderBy {
  String value() default "";
}
```

排序元素的语法是一个 orderby_list，如下所示：

```
orderby_list::= orderby_item [,orderby_item]*
orderby_item::= property_or_field_name [ASC | DESC]
```

如果没有指定 ASC 还是 DESC，那么缺省是 ASC。

如果没有指定排序元素，那么缺省是按主键排序。

排序的属性或字段名必须和关联类的字段或属性名一一对应。在排序中使用的属性或字段必须和比较操作中支持的列一一对应。

例子：

```
@Entity public class Course {
 ...
 @ManyToMany
 @OrderBy("lastname ASC")
 public List<Student> getStudents() {...};
 ...
}

@Entity public class Student {
  ...
  @ManyToMany(mappedBy="students")
  @OrderBy // PK is assumed
  public List<Course> getCourses() {...};
  ...
}
```

8.1.29 Inheritance 注解符

Inheritance 定义实体类层次的继承策略。它在类层次的根上指定。

本规范不要求支持继承策略的组合。可移植的应用在实体层次中应当只使用一个单一继承策略。

三种继承策略是：

- 每个类层次是一个表。
- 每个类是一个表。
- 关联子类策略。

参考 2.1.10 章节了解更详细的继承策略。继承策略选项在 InheritanceType 枚举中定义：

```
public enum InheritanceType
     { SINGLE_TABLE, TABLE_PER_CLASS, JOINED };
```

在这个版本中是否支持 TABLE_PER_CLASS 策略是可选的。

如果没有指定 Inheritance 或没有指定继承类型，则使用 SINGLE_TABLE 策略。

表 19 列出了可以在 Inheritance 注解符中指定的注解符元素以及它们的缺省值。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Inheritance {
     InheritanceType strategy() default SINGLE_TABLE;
}
```

Table 19 Inheritance Annotation Elements

Type	Name	Description	Default
InheritanceType	strategy	(Optional) The inheritance strategy to use for the entity inheritance hierarchy.	InheritanceType.SINGLE_TABLE

例子：

```
@Entity
@Inheritance(strategy=JOINED)
public class Customer { ... }

@Entity
public class ValuedCustomer extends Customer { ... }
```

8.1.30 DiscrminatorColumn 注解符

对 SINGLE_TABLE 策略，典型地也是对 JOINED 策略，持久化提供商将使用一个类型区分列。DiscrminatorColumn 用于为 SINGLE_TABLE 和 JOINED 继承映射策略定义区分列。

继承映射策略和区分列只在类层次的根上指定，或者在子层次上使用不同的映射策略。

DiscrminatorColumn 可以被指定在实体类上（包括抽象实体类）。

如果没有指定 DiscrminatorColumn，并且还要求区分列，那么区分列的名称缺省为“DTYPE”，且区分列的类型为 STRING。

表 20 列出了可以在 DiscrminatorColumn 注解符中指定的注解符元素以及他们的缺省值。

支持的区分类型在 DiscriminatorType 枚举中定义：

```
public enum DiscriminatorType { STRING, CHAR, INTEGER };
```

如果在可选的 columnDefinition 元素中指定值，那么区分列的类型和区分器类型一致。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface DiscriminatorColumn {
     String name() default "DTYPE";
     DiscriminatorType discriminatorType() default STRING;
     String columnDefinition() default "";
     int length() default 31;
}
```

Table 20 DiscriminatorColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of column to be used for the discriminator.	"DTYPE"
Dis-crimi-natorTy pe	discriminator-Type	(Optional) The type of object/column to use as a class discriminator.	DiscriminatorType.STRING
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the discrimina-tor column.	Provider-generated SQL to create a column of the speci-fied discriminator type.
String	length	(Optional) The column length for String-based discriminator types. Ignored for other discriminator types.	31

例子：

```
@Entity
@Table(name="CUST")
@Inheritance
@DiscriminatorColumn(name="DISC", discriminatorType=STRING,length=20)
public class Customer { ... }

@Entity
public class ValuedCustomer extends Customer { ... }
```

8.1.31 DiscriminatorValue 注解符

DiscriminatorValue 用于指定区分列的值。DiscriminatorValue 只能用在具体的实体类上。如果没有指定 DiscriminatorValue，而且还使用了区分列，那么提供商特定的功能会用于生成一个代表这个实体类型的值。

继承策略和区分列只是在实体类层次根上或使用不同继承策略的子层次上指定。区分列的值如果没有缺省值，则应当为类层次上的每一个实体类指定。

表 21 列出了可以在 DiscriminatorValue 注解符内指定的注解符元素以及它们的缺省值。

区分列的值必须和被指定列（或缺省的区分列）的类型一致。如果区分列的类型是整型，那么它被指定的值必须可以被转化为整型值（如，“1”）。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface DiscriminatorValue {
    String value();
}
```

Table 21 DiscriminatorValueAnnotation Elements

Type	Name	Description	Default
String	value	(Optional) The value that indicates that the row is an entity of the annotated entity type.	If the DiscriminatorValue annotation is not specified, a provider-specific function to generate a value representing the entity type is used for the value of the discriminator column. If the Discriminator Type is STRING, the discriminator value default is the entity name.

例子：

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="DISC", discriminatorType=STRING,length=20)
@DiscriminatorValue("CUSTOMER")
public class Customer { ... }

@Entity
@DiscriminatorValue("VCUSTOMER")
public class ValuedCustomer extends Customer { ... }
```

8.1.32 PrimarykeyJoinColumn 注解符

PrimarykeyJoinColumn 指定用作外键（关联到另外一个表）的主键列。

在 JOINED 策略中，PrimarykeyJoinColumn 用于将实体子类的主表关联到父类的主表。它在 SecondaryTable 注解符中被用于将从表关联到主表；它可以被用在 OneToOne 映射中，在这个映射中引用实体的主键用作被引用实体的外键。

表 22 列出了可以在 PrimarykeyJoinColumn 注解符中指定的注解符元素以及它们的缺省值。

如果在 JOINED 映射策略中没有为子类指定 PrimarykeyJoinColumn 注解符，那么假定外键列的名字和父类主表的主键列相同。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PrimaryKeyJoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    String columnDefinition() default "";
}
```

Table 22 PrimaryKeyJoinColumn Annotation Elements

Type	Name	Description	Default
String	name	The name of the primary key column of the current table.	The same name as the primary key column of the primary table of the superclass (JOINED mapping strategy); the same name as the primary key column of the primary table (SecondaryTable mapping); or the same name as the primary key column for the table for the referencing entity (OneToOne mapping).
String	referencedColumnName	(Optional) The name of the primary key column of the table being joined to.	The same name as the primary key column of the primary table of the superclass (JOINED mapping strategy); the same name as the primary key column of the primary table (SecondaryTable mapping); or the same name as the primary key column of the table for the referenced entity (OneToOne mapping).
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column. This should not be specified for a OneToOne primary key association.	Generated SQL to create a column of the inferred type.

例子：Customer 和 valuedCustomer 子类

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=JOINED)
@DiscriminatorValue("CUST")
public class Customer { ... }

@Entity
@Table(name="VCUST")
@DiscriminatorValue("VCUST")
@PrimaryKeyJoinColumn(name="CUST_ID")
public class ValuedCustomer extends Customer { ... }
```

8.1.33 PrimaryKeyJoinColumns 注解符

通过 PrimaryKeyJoinColumns 可以支持组合主键。PrimaryKeyJoinColumns 组合了 PrimaryKeyJoinColumn。

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface PrimaryKeyJoinColumns {
PrimaryKeyJoinColumn[] value();
}
```

例子：ValuedCustomer 子类

```
@Entity
@Table(name="VCUST")
@DiscriminatorValue("VCUST")
@PrimaryKeyJoinColumns({
     @PrimaryKeyJoinColumn(name="CUST_ID",
          referencedColumnName="ID"),
     @PrimaryKeyJoinColumn(name="CUST_TYPE",
          referencedColumnName="TYPE")
})
public class ValuedCustomer extends Customer { ... }
```

例子：在 Employee 和 EmployeeInfo 类之间的 OneToOne 关系

```
public class EmpPK {
     public Integer id;
     public String name;
}

@Entity
@IdClass(com.acme.EmpPK.class)
public class Employee {

     @Id Integer id;
     @Id String name;

     @OneToOne
     @PrimaryKeyJoinColumns({
          @PrimaryKeyJoinColumn(name="ID", referencedColumn-
Name="EMP_ID"),
          @PrimaryKeyJoinColumn(name="NAME", referencedColumn-
Name="EMP_NAME")})
     EmployeeInfo info;

     ...
}

@Entity
@IdClass(com.acme.EmpPK.class)
public class EmployeeInfo {

     @Id @Column(name="EMP_ID")
     Integer id;
     @Id @Column(name="EMP_NAME")
     String name;

     ...
}
```

8.1.34 Embeddable 注解符

Embeddable 用于标记一个对象是作为一个实体内在的部分被存储，这个对象和实体共享唯一标识。嵌入对象的每个持久化属性或字段都被映射到数据库表。只有 Basic，Column、Lob、Temporal 和 Enumerated 映射注解符可以方便地用于

映射注解为 Embeddable 的类的持久化字段或属性。(Transient 可以用于指派可嵌入类的非持久化状态)

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Embeddable {
}
```

例子:

```
@Embeddable
public class EmploymentPeriod {
     java.util.Date startDate;
     java.util.Date endDate;
     ...
}
```

8.1.35 Embeded 注解符

Embeded 用于指定值为可嵌入类的实例的实体的持久化字段或属性。

AttributeOverride 和/或 AttributeOverrides 注解符可以用于重载声明在可嵌入类的列的映射，这个可嵌入类被映射成实体表。

不要求实现支持被嵌入对象被映射成多个表(例如，分成主表和从表或多个从表)。

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Embedded {}
```

例子:

```
@Embedded
@AttributeOverrides({
     @AttributeOverride(name="startDate",
                        column=@Column(name="EMP_START")),
     @AttributeOverride(name="endDate",
                        column=@Column(name="EMP_END"))
})
public EmploymentPeriod getEmploymentPeriod() { ... }
```

8.1.36 MappedSuperClass 注解符

MappedSuperClass 指派一个类，这个类的映射信息应用到它的子类上。被映射的超类没有单独的表。

由于被映射的超类没有数据库表，所以除了映射只应用到它的子类上外，用 MappedSuperClass 注解符指派的类可以用同样的方式被映射作为实体。当应用到子类时，继承的映射将应用在子类表的上下文中。在这样的子类中，可以用 AttributeOverride 来重载映射信息。

```
@Target(TYPE) @Retention(RUNTIME)
public @interface MappedSuperclass {}
```

8.1.37 SequenceGenerator 注解符

当指定 GeneratedValue 注解符的生成器元素时，SequenceGenerator 定义一个可以通过名字引用的主键生成器。生成器可以定义在类、或字段/属性上。序列生成器可以指定在实体类或注解字段/属性上。生成器名字的范围在持久化单元内是全局的（跨所有的生成器类型）。

表 23 列出了可以在 SequenceGenerator 注解符中指定的注解符元素以及它们的缺省值。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface SequenceGenerator {
    String name();
    String sequenceName() default "";
    int initialValue() default 1;
    int allocationSize() default 50;
}
```

Table 23 SequenceGenerator Annotation Elements

Type	Name	Description	Default
String	name	(Required) A unique generator name that can be referenced by one or more classes to be the generator for primary key values.	
String	sequenceName	(Optional) The name of the database sequence object from which to obtain primary key values.	A provider-chosen value
int	initialValue	(Optional) The value from which the sequence object is to start generating.	1
int	allocationSize	(Optional) The amount to increment by when allocating sequence numbers from the sequence.	50

例子：

```
@SequenceGenerator(name="EMP_SEQ", allocationSize=25)
```

8.1.38 TableGenerator 注解符

当指定 GeneratedValue 注解符的生成器元素时，TableGenerator 定义一个可以通过名字引用的主键生成器。生成器可以定义在类、或字段/属性上。序列生成器可以指定在实体类或注解字段/属性上。生成器名字的范围在持久化单元内是全局的（跨所有的生成器类型）。

表 24 列出了可以在 TableGenerator 注解符内指定的注解符元素以及它们的缺省值。

元素 table 指定了用于持久化提供商存储生成的 id 值的表的表名。实体类型典型地用它自己表中的行来生成 id 值。Id 值通常是正整数。

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface TableGenerator {
    String name();
    String table() default "";
    String catalog() default "";
    String schema() default "";
    String pkColumnName() default "";
    String valueColumnName() default "";
    String pkColumnValue() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
    UniqueConstraint[] uniqueConstraints() default {};
}
```

Table 24 TableGenerator Annotation Elements

Type	Name	Description	Default
String	name	(Required) A unique generator name that can be referenced by one or more classes to be the generator for id values.	
String	table	(Optional) Name of table that stores the generated id values.	Name is chosen by persistence provider
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user
String	pkColumnName	(Optional) Name of the primary key column in the table	A provider-chosen name
String	valueColumnName	(Optional) Name of the column that stores the last value generated.	A provider-chosen name
String	pkColumnValue	(Optional) The primary key value in the generator table that distinguishes this set of generated values from others that may be stored in the table.	A provider-chosen value to store in the primary key column of the generator table
int	initialValue	(Optional) The value used to initialize the column that stores the last value generated.	0
int	allocationSize	(Optional) The amount to increment by when allocating id numbers from the generator.	50
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect. These constraints apply in addition to primary key constraints .	No additional constraints

例 1：

```
@Entity public class Employee {
    ...
    @TableGenerator(
        name="empGen",
        table="ID_GEN",
        pkColumnName="GEN_KEY",
        valueColumnName="GEN_VALUE",
        pkColumnValue="EMP_ID",
        allocationSize=1)
    @Id
    @GeneratedValue(strategy=TABLE, generator="empGen")
    public int id;
    ...
}
```

例 2：

```
@Entity public class Address {
     ...
     @TableGenerator(
          name="addressGen",
          table="ID_GEN",
          pkColumnName="GEN_KEY",
          valueColumnName="GEN_VALUE",
          pkColumnValue="ADDR_ID")
     @Id
     @GeneratedValue(strategy=TABLE, generator="addressGen")
     public int id;
     ...
}
```

8.2 O/R 映射注解符使用样例

8.2.1 简单映射例子

```
@Entity
public class Customer {

    @Id @GeneratedValue(strategy=AUTO) Long id;
    @Version protected int version;
    @ManyToOne Address address;
    @Basic String description;
    @OneToMany(targetEntity=com.acme.Order.class,
               mappedBy="customer")
    Collection orders = new Vector();
    @ManyToMany(mappedBy="customers")
    Set<DeliveryService> serviceOptions = new HashSet();

    public Long getId() { return id; }

    public Address getAddress() { return address; }
    public void setAddress(Address addr) {
        this.address = addr;
    }

    public String getDescription() { return description; }
    public void setDescription(String desc) {
        this.description = desc;
    }

    public Collection getOrders() { return orders; }

    public Set<DeliveryService> getServiceOptions() {
        return serviceOptions;
    }
}


@Entity
public class Address {

    private Long id;
    private int version;
    private String street;

    @Id @GeneratedValue(strategy=AUTO)
    public Long getId() { return id; }
    protected void setId(Long id) { this.id = id; }

    @Version
    public int getVersion() { return version; }
    protected void setVersion(int version) {
        this.version = version;
    }
```

```
    public String getStreet() { return street; }
    public void setStreet(String street) {
        this.street = street;
    }
}


@Entity
public class Order {

    private Long id;
    private int version;
    private String itemName;
    private int quantity;
    private Customer cust;

    @Id @GeneratedValue(strategy=AUTO)
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    @Version
    protected int getVersion() { return version; }
    protected void setVersion(int version) {
        this.version = version;
    }

    public String getItemName() { return itemName; }
    public void setItemName(String itemName) {
        this.itemName = itemName;
    }

    public int getQuantity() { return quantity; }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    @ManyToOne
    public Customer getCustomer() { return cust; }
    public void setCustomer(Customer cust) {
        this.cust = cust;
    }
}


@Entity
@Table(name="DLVY_SVC")
public class DeliveryService {

    private String serviceName;
    private int priceCategory;
    private Collection customers;

    @Id
    public String getServiceName() { return serviceName; }
    public void setServiceName(String serviceName) {
        this.serviceName = serviceName;
    }

    public int getPriceCategory() { return priceCategory; }
```

```
    public void setPriceCategory(int priceCategory) {
        this.priceCategory = priceCategory;
    }

    @ManyToMany(targetEntity=com.acme.Customer.class)
    @JoinTable(name="CUST_DLVRY")
    public Collection getCustomers() { return customers; }
    public setCustomers(Collection customers) {
        this.customers = customers;
    }
}
```

8.2.2 复杂的例子

```
/***** Employee class *****/

@Entity
@Table(name="EMPL")
@SecondaryTable(name="EMP_SALARY",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="EMP_ID",
                      referencedColumnName="ID"))
public class Employee implements Serializable {

    private Long id;
    private int version;
    private String name;
    private Address address;
    private Collection phoneNumbers;
    private Collection<Project> projects;
    private Long salary;
    private EmploymentPeriod period;

    @Id @GeneratedValue(strategy=TABLE)
    public Integer getId() { return id; }
    protected void setId(Integer id) { this.id = id; }

    @Version
    @Column(name="EMP_VERSION", nullable=false)
    public int getVersion() { return version; }
    protected void setVersion(int version) {
        this.version = version;
    }

    @Column(name="EMP_NAME", length=80)
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @ManyToOne(cascade=PERSIST, optional=false)
    @JoinColumn(name="ADDR_ID",
                referencedColumnName="ID", nullable=false)
    public Address getAddress() { return address; }
    public void setAddress(Address address) {
        this.address = address;
    }

    @OneToMany(targetEntity=com.acme.PhoneNumber.class,
               cascade=ALL, mappedBy="employee")
    public Collection getPhoneNumbers() { return phoneNumbers; }
    public void setPhoneNumbers(Collection phoneNumbers) {
        this.phoneNumbers = phoneNumbers;
    }

    @ManyToMany(cascade=PERSIST, mappedBy="employees")
    @JoinTable(
         name="EMP_PROJ",
         joinColumns=@JoinColumn(
            name="EMP_ID", referencedColumnName="ID"),
         inverseJoinColumns=@JoinColumn(
            name="PROJ_ID", referencedColumnName="ID"))
    public Collection<Project> getProjects() { return projects; }
    public void setProjects(Collection<Project> projects) {
```

```
        this.projects = projects;
    }

    @Column(name="EMP_SAL", table="EMP_SALARY")
    public Long getSalary() { return salary; }
    public void setSalary(Long salary) {
        this.salary = salary;
    }

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="startDate",
            column=@Column(name="EMP_START")),
        @AttributeOverride(name="endDate",
            column=@Column(name="EMP_END"))
    })
    public EmploymentPeriod getEmploymentPeriod() {
        return period;
    }
    public void setEmploymentPeriod(EmploymentPeriod period) {
        this.period = period;
    }
}


/***** Address class *****/

@Entity
public class Address implements Serializable {

    private Integer id;
    private int version;
    private String street;
    private String city;

    @Id @GeneratedValue(strategy=IDENTITY)
    public Integer getId() { return id; }
    protected void setId(Integer id) { this.id = id; }

    @Version @Column(name="VERS", nullable=false)
    public int getVersion() { return version; }
    protected void setVersion(int version) {
        this.version = version;
    }

    @Column(name="RUE")
    public String getStreet() { return street; }
    public void setStreet(String street) {
        this.street = street;
    }

    @Column(name="VILLE")
    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }
}

/***** PhoneNumber class *****/

@Entity
```

```
@Table(name="PHONE")
public class PhoneNumber implements Serializable {

    private String number;
    private int phoneType;
    private Employee employee;

    @Id
    public String getNumber() { return number; }
    public void setNumber(String number) {
        this.number = number;
    }

    @Column(name="PTYPE")
    public int getPhonetype() { return phonetype; }
    public void setPhoneType(int phoneType) {
        this.phoneType = phoneType;
    }

    @ManyToOne(optional=false)
    @JoinColumn(name="EMP_ID", nullable=false)
    public Employee getEmployee() { return employee; }
    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
}


/***** Project class *****/

@Entity
@Inheritance(strategy=JOINED)
DiscriminatorValue("Proj")
@DiscriminatorColumn(name="DISC")
public class Project implements Serializable {

    private Integer projId;
    private int version;
    private String name;
    private Set<Employee> employees;

    @Id @GeneratedValue(strategy=TABLE)
    public Integer getId() { return projId; }
    protected void setId(Integer id) { this.projId = id; }

    @Version
    public int getVersion() { return version; }
    protected void setVersion(int version) { this.version = version; }

    @Column(name="PROJ_NAME")
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @ManyToMany(mappedBy="projects")
    public Set<Employee> getEmployees() { return employees; }
    public void setEmployees(Set<Employee> employees) {
        this.employees = employees;
    }
}
```

```
    /***** GovernmentProject subclass *****/

    @Entity
    @Table(name="GOVT_PROJECT")
    @DiscriminatorValue("GovtProj")
    @PrimaryKeyJoinColumn(name="GOV_PROJ_ID",
                          referencedColumnName="ID")
    public class GovernmentProject extends Project {

        private String fileInfo;

        @Column(name="INFO")
        public String getFileInfo() { return fileInfo; }
        public void setFileInfo(String fileInfo) {
            this.fileInfo = fileInfo;
        }
    }


    /***** CovertProject subclass *****/

    @Entity
    @Table(name="C_PROJECT")
    @DiscriminatorValue("CovProj")
    @PrimaryKeyJoinColumn(name="COV_PROJ_ID",
                          referencedColumnName="ID")
    public class CovertProject extends Project {

        private String classified;

        public CovertProject() { super(); }

        public CovertProject(String classified) {
            this();
            this.classified = classified;
        }

        @Column(updatable=false)
        public String getClassified() { return classified; }
        protected void setClassified(String classified) {
            this.classified = classified;
        }
    }

/***** EmploymentPeriod class *****/

@Embeddable
public class EmploymentPeriod implements Serializable {

    private Date start;
    private Date end;

    @Column(nullable=false)
    public Date getStartDate() { return start; }
    public void setStartDate(Date start) {
        this.start = start;



    }

    public Date getEndDate() { return end; }
    public void setEndDate(Date end) {
       this.end = end;
    }
}
```

9 XML Descriptor

XML 描述符打算用于可选方案和重载 java 元数据注解机制。

9.1 XML Schema

本节定义 XML 配置用于重载注解符的规则，以及 XML 元素作为 peristence-unit-defaluts，entity-mappings，entity，mapped-superclass 和 embedaable 元素的子元素的规则。

如果指定 peristence-unit-metadata 元素的子元素 xml-mapping-metadata-complete，持久化单元的所有映射元数据都要被包含在为持久化单元定义的 XML 映射文件内，并且忽略类上的注解符。当指定了 xml-mapping-metadata-complete 且 XML 元素被忽略，则使用缺省值。（如果指定了 xml-mapping-metadata-complete 元素，那么在实体、被映射超类和可嵌入元素指定的任何 metadata-complete 元素都会被忽略）

9.1.1 持久化单元缺省的子元素

9.1.1.1Schema

Schema 子元素应用到持久化单元内的所有的实体，表生成器和关联表。

entity-mappings 元素的任何 schema 子元素、在实体上的 Table 或 SecondaryTable 注解符内显式指定的任何 schema 元素或定义在 entity 元素内的任何 table 或 secondary-table 子元素上的任何 schema 属性、显式地在 TableGenerator 或 table-generator、显式地在 JoinTable 或 join-table 子元素中指定的 schema 元素都可以重载 scheam 子元素。

9.1.1.2Catalog

Catalog 子元素应用到持久化单元内的所有实体，表生成器和关联表。

Entity-mappings 元素的任何 Catalog 子元素、显式定义在实体上的 Table 或

SecondaryTable 注解符中指定的 catalog 元素或定义在 entity XML 元素的 table 或 secondary-table 子元素的任何 catalog 属性、显式定义在 JoinTable 或 join-table 子元素中任何 catalog 元素都可以被重载。

9.1.1.3Access

Access 子元素可以应用到持久化单元的所有受管理类。

通过使用在实体类的字段或属性上指定的映射信息、或通过 entity-mappings 元素的任何 access 子元素、或通过定义在 entity，mapped-superclass 或 embeddable XML 元素可以覆盖 access 子元素。

9.1.1.4Cascade-persist