1 JPA 批注参考

版本： 5/12/06

作为 Java 企业版 5 (Java EE 5) Enterprise Java Bean (EJB) 3.0 规范的组成部分，Java 持续性 API (JPA) 显著简化了 EJB 持续性并提供了一个对象关系映射方法，该方法使您可以采用声明方式定义如何通过一种标准的可移植方式（在 Java EE 5 应用服务器内部以及 Java 标准版 (Java SE) 5 应用程序中的 EJB 容器外部均可使用）将 Java 对象映射到关系数据库表。

在 JPA 之前，Java EE 应用程序将持续类表示为容器管理的实体 bean。使用 JPA，您可以将任何普通的旧式 Java 对象 (POJO) 类指定为 JPA 实体：一个应使用 JPA 持续性提供程序的服务将其非临时字段持久保存到关系数据库（在 Java EE EJB 容器的内部或在简单 Java SE 应用程序中的 EJB 容器的外部）的 Java 对象。

使用 JPA 时，可以使用批注配置实体的 JPA 行为。批注是一种使用元数据修饰 Java 源代码的简单表达方法，它编译为相应的 Java 类文件，以便在运行时由 JPA 持续性提供程序解释以管理 JPA 行为。

例如，要将 Java 类指定为 JPA 实体，请使用 @Entity 批注，如下所示：

```
@Entity
public class Employee implements Serializable {
    ...
}
```

您可以有选择地使用批注来修饰实体类以覆盖默认值。这称作*按异常进行配置* (configuration by exception)。

本参考广泛引用了 JSR-220 Enterprise JavaBean 版本 3.0 Java 持续性 API 规范，以按类别汇总批注信息（请参阅表 1-1），并解释了何时以及如何使用这些批注来自定义 JPA 行为，以满足应用程序的需要。

有关详细信息，请参阅：

- 批注索引
- 完整的 JPA 批注 Javadoc

表 1-1 按类别划分的 JPA 批注

类别	说明	批注
实体	默认情况下，JPA 持续性提供程序假设 Java 类是非持续类，并且仅当使用此批注对其进行修饰的情况下才可用于 JPA 服务。 使用此批注将普通的旧式 Java 对象 (POJO) 类指定为实体，以便可以将它用于 JPA 服务。 要将类用于 JPA 服务，必须将该类指定为 JPA 实体（使用此批注或 `orm.xml` 文件）。	@Entity

数据库模式属性	**默认情况下，JPA 持续性提供程序假设实体名称对应于同名的数据库表，且实体的数据成员名称对应于同名的数据库列。** **使用这些批注覆盖此默认行为，并微调对象模型与数据模型之间的关系。**	@Table @SecondaryTable @SecondaryTables @Column @JoinColumn @JoinColumns @PrimaryKeyJoinColumn @PrimaryKeyJoinColumns @JoinTable @UniqueConstraint
身份	**默认情况下，JPA 持续性提供程序假设每个实体必须至少有一个用作主键的字段或属性。** **使用这些批注指定以下项之一：** • **一个** `@Id` • **多个** `@Id` **和一个** `@IdClass` • **一个** `@EmbeddedId` **还可以使用这些批注微调数据库维护实体身份的方式。**	@Id @IdClass @EmbeddedId @GeneratedValue @SequenceGenerator @TableGenerator
直接映射	**默认情况下，JPA 持续性提供程序为大多数 Java 基元类型、基元类型的包装程序以及** `enums` **自动配置一个** `Basic` **映射。** **使用这些批注微调数据库实现这些映射的方式。**	@Basic @Enumerated @Temporal @Lob @Transient

关系映射	JPA 持续性提供程序要求您显式映射关系。 使用这些批注指定实体关系的类型和特征，以微调数据库实现这些关系的方式。	@OneToOne @ManyToOne @OneToMany @ManyToMany @MapKey @OrderBy
组合	默认情况下，JPA 持续性提供程序假设每个实体均映射到它自己的表。 使用这些批注覆盖其他实体拥有的此种实体行为。	@Embeddable @Embedded @AttributeOverride @AttributeOverrides @AssociationOverride @AssociationOverrides
继承	默认情况下，JPA 持续性提供程序假设所有持久字段均由一个实体类定义。 如果实体类继承了一个或多个超类中的某些或所有持续字段，则使用这些批注。	@Inheritance @DiscriminatorColumn @DiscriminatorValue @MappedSuperclass @AssociationOverride @AssociationOverrides @AttributeOverride @AttributeOverrides
锁定	默认情况下，JPA 持续性提供程序假设应用程序负责数据一致性。 使用此批注启用 JPA 管理的乐观锁定（推荐使用）。	@Version

生命周期回调事件	默认情况下，JPA 持续性提供程序处理所有持续性操作。 如果您要在实体生命周期内随时调用自定义逻辑，请使用这些批注将实体方法与 JPA 生命周期事件关联。图 1-1 演示了这些生命周期事件之间的关系。	@PrePersist @PostPersist @PreRemove @PostRemove @PreUpdate @PostUpdate @PostLoad @EntityListeners @ExcludeDefaultListeners @ExcludeSuperclassListeners
实体管理器	在使用 JPA 持续性提供程序的应用程序中，您可以使用 `EntityManager` 实例执行所有持续性操作（创建、读取、更新和删除）。 使用这些批注将实体与实体管理器关联并自定义实体管理器的环境。	@PersistenceUnit @PersistenceUnits @PersistenceContext @PersistenceContexts @PersistenceProperty
查询	在使用 JPA 持续性提供程序的应用程序中，可以使用实体管理器动态创建和执行查询，也可以预定义查询并在运行时按名称执行它们。 使用这些批注预定义查询并管理它们的结果集。	@NamedQuery @NamedQueries @NamedNativeQuery @NamedNativeQueries @QueryHint @ColumnResult @EntityResult @FieldResult @SqlResultSetMapping

		@SqlResultSetMappings

@AssociationOverride

默认情况下，JPA 持续性提供程序自动假设子类继承超类中定义的持久属性及其关联映射。

如果继承的列定义对实体不正确（例如，如果继承的列名与已经存在的数据模型不兼容或作为数据库中的列名无效），请使用 `@AssociationOverride` 批注自定义从 @MappedSuperclass 或 @Embeddable 继承的 @OneToOne 或 @ManyToOne 映射，以更改与字段或属性关联的 @JoinColumn。

如果有多个要进行的 `@AssociationOverride` 更改，则必须使用 @AssociationOverrides。

要自定义基本映射以更改它的 @Column，请使用 @AttributeOverride。

表 1-4 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-2 @AssociationOverride 属性

属性	必需	说明
`joinColumns`	✔	要指定映射到持久属性的连接列，请将 `joinColums` 设置为 `JoinColumn` 实例的数组（请参阅 @JoinColumn）。 映射类型将与可嵌套类或映射的超类中定义的类型相同。
`name`	✔	如果使用了基于属性的访问，则映射的为嵌入对象中的属性名称，如果使用了基于字段的访问，则映射的为字段名称。

示例 1-4 显示了示例 1-5 中的实体扩展的 @MappedSuperclass。示例 1-5 显示了如何在实体子类中使用 `@AssociationOverride` 覆盖 @MappedSuperclass `Employee` 中定义（默认情况下）的 @JoinColumn 以便关联到 `Address`。

如果使用 `@AssociationOverride`，则 `Employee` 表包含以下列：

- `ID`
- `VERSION`
- `ADDR_ID`
- `WAGE`

如果不使用 `@AssociationOverride`，则 `Employee` 表包含以下列：

- `ID`
- `VERSION`
- `ADDRESS`

- WAGE

示例 1-1 @MappedSuperclass

```
@MappedSuperclass
public class Employee {
@Id protected Integer id;
@Version protected Integer version;
@ManyToOne protected Address address;
    ...
}
```

示例 1-2 @AssociationOverride

```
@Entity@AssociationOverride(name="address", joinColumns=@JoinColumn(name="ADDR_ID"))public
class PartTimeEmployee extends Employee {    @Column(name="WAGE")    protected Float
hourlyWage;
    ...
}
```

@AssociationOverrides

如果需要指定多个 @AssociationOverride，则**必需使用一个** `@AssociationOverrides` **批注指定所有关**联**覆盖**。

表 1-5 **列出了此批注的属性。有关更多**详细**信息，**请**参**阅 API。

表 1-3 @AssociationOverrides 属性

属性	必需	说明
`value`	✔	**要指定两个或更多覆盖，**请**将** `value` 设**置**为 `AssociationOverride` 实**例的数**组（请**参**阅 @AssociationOverride）。

示例 1-6 显**示了如何使用此批注指定两个关**联**覆盖**。

示例 1-3 @AssociationOverrides

```
@Entity
@AssociationOverrides({
@AssociationOverride(name="address", joinColumn=@Column(name="ADDR_ID")),
@AssociationOverride(name="id", joinColumn=@Column(name="PTID"))
})
public class PartTimeEmployee extends Employee {
@Column(name="WAGE")
protected Float hourlyWage;
    ...
}
```

@AttributeOverride

默认**情况下，**JPA **持**续**性提供程序自**动**假**设**子类**继**承超类中定**义**的持久属性及其基本映射**。

如果针对实**体**继**承的列定**义**不正确，**请**使用** `@AttributeOverride` **批注自定**义**一个从** @MappedSuperclass **或** @Embeddable **继承的基本映射以更改与字段或属性关**联**的** @Column。**（例如，如果**继**承的列名与事先存在的数据模型不兼容，或者作**为**数据**库**中的列名无效）**。

如果有多个要进行的 @AttributeOverride 更改，则必须使用 @AttributeOverrides。

要自定义关联映射以更改它的 @JoinColumn，请使用 @AssociationOverride。

表 1-4 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-4 @AttributeOverride 属性

属性	必需	说明
column	✔	映射到持久属性的 @Column。映射类型将与可嵌套类或映射超类中定义的类型相同。
name	✔	如果使用了基于属性的访问，则映射的为嵌入对象中的属性名称，如果使用了基于字段的访问，则映射的为字段名称。

示例 1-4 显示了示例 1-5 中的实体扩展的 @MappedSuperclass。示例 1-5 显示了如何使用实体子类中的 @AttributeOverride 覆盖 @MappedSuperclass Employee 中定义（默认情况下）的 @Column，以便基本映射到 Address。

如果使用 @AttributeOverride，则 Employee 表包含以下列：

- ID
- VERSION
- ADDR_STRING
- WAGE

如果不使用 @AttributeOverride，则 Employee 表包含以下列：

- ID
- VERSION
- ADDRESS
- WAGE

示例 1-4 @MappedSuperclass

```
@MappedSuperclass
public class Employee {
@Id protected Integer id;
@Version protected Integer version;
protected String address;
    ...
}
```

示例 1-5 @AttributeOverride

```
@Entity
@AttributeOverride(name="address", column=@Column(name="ADDR_STRING"))
public class PartTimeEmployee extends Employee {
@Column(name="WAGE")
protected Float hourlyWage;
    ...
}
```

@AttributeOverrides

如果需要指定多个 @AttributeOverride，则必需使用一个 `@AttributeOverrides` 批注指定所有属性覆盖。

表 1-5 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-5 @AttributeOverrides 属性

属性	必需	说明
`value`	✔	要指定两个或更多属性覆盖，请将 `value` 设置为 `AttributeOverride` 实例的数组（请参阅 @AttributeOverride）。

示例 1-6 显示了如何使用此批注指定两个属性覆盖。

示例 1-6 @AttributeOverrides

```
@Entity
@AttributeOverrides({
@AttributeOverride(name="address", column=@Column(name="ADDR_ID")),
@AttributeOverride(name="id", column=@Column(name="PTID"))
})
public class PartTimeEmployee extends Employee {

@Column(name="WAGE")
protected Float hourlyWage;

public PartTimeEmployee() {
        ...
    }

public Float getHourlyWage() {
        ...
    }

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

@Basic

默认情况下，JPA 持续性提供程序为大多数 Java 基元类型、基元类型的包装程序以及枚举自动配置一个 `@Basic` 映射。

使用 `@Basic` 批注：

- 将获取类型配置为 `LAZY`

- 如果空值不适合于应用程序，则将映射配置为禁止空值（针对非基元类型）

表 1-6 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-6 @Basic 属性

属性	必需	说明
`fetch`		默认值：`FetchType.EAGER`。 默认情况下，JPA 持续性提供程序使用获取类型 `EAGER`：这将要求持续性提供程序运行时必须迫切获取数据。 如果这不适合于应用程序或特定的持久字段，请将 `fetch` 设置为 `FetchType.LAZY`：这将提示持续性提供程序在首次访问数据（如果可以）时应不急于获取数据。
`optional`		默认值：`true`。 默认情况下，JPA 持续性提供程序假设所有（非基元）字段和属性的值可以为空。 如果这并不适合于您的应用程序，请将 `optional` 设置为 `false`。

示例 1-7 显示了如何使用此批注为基本映射指定获取类型 `LAZY`。

示例 1-7 @Basic

```
@Entity
public class Employee implements Serializable {
    ...
@Basic(fetch=LAZY)
protected String getName() {
return name;
    }
    ...
}
```

@Column

默认情况下，JPA 持续性提供程序假设每个实体的持久字段存储在其名称与持久字段的名称相匹配的数据库表列中。

使用 `@Column` 批注：

- 将持久字段与其他名称关联（如果默认列名难于处理、与事先存在的数据模型不兼容或作为数据库中的列名无效）
- 将持久字段与辅助表中的列关联（请参阅 @SecondaryTable）
- 微调数据库中列的特征

表 1-7 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-7 @Column 属性

属性	必需	说明

columnDefinition		默认值：空 String。 默认情况下，JPA 使用最少量 SQL 创建一个数据库表列。 如果需要使用更多指定选项创建的列，请将 columnDefinition 设置为在针对列生成 DDL 时希望 JPA 使用的 SQL 片断。 注意：捕获批注中的 DDL 信息时，某些 JPA 持续性提供程序可以在生成数据库模式时使用此 DDL。例如，请参阅"用于 Java2DB 模式生成的 TopLink JPA 扩展"。
insertable		默认值：true。 默认情况下，JPA 持续性提供程序假设所有列始终包含在 SQL INSERT 语句中。 如果该列不应包含在这些语句中，请将 insertable 设置为 false。
length		默认值： 255 默认情况下，JPA 持续性提供程序假设所有列在用于保存 String 值时的最大长度为 255 个字符。 如果该列不适合于您的应用程序或数据库，请将 length 设置为适合于您的数据库列的 int 值。
name		默认值：JPA 持续性提供程序假设实体的每个持久字段都存储在其名称与持久字段或属性的名称相匹配的数据库表列中。 要指定其他列名，请将 name 设置为所需的 String 列名。
nullable		默认值：true。 默认情况下，JPA 持续性提供程序假设允许所有列包含空值。 如果不允许该列包含空值，请将 nullable 设置为 false。
precision		默认值： 0. 默认情况下，JPA 持续性提供程序假设所有列在用于保存十进制（精确数字）值时的精度为 0。 如果该精度不适合于您的应用程序或数据库，请将 precision 设置为相应的 int 精度。
scale		默认值： 0. 默认情况下，JPA 持续性提供程序假设所有列在用于保存十进制（精确数字）值时的伸缩度为 0。 如果该伸缩度不适合于您的应用程序或数据库，请将 scale 设置为相应的 int 精度。
table		默认值：JPA 持续性提供程序假设实体的所有持久字段都存储到一个其名称为实体名称的数据库表中（请参阅 @Table）。 如果该列与辅助表关联（请参阅 @SecondaryTable），请将 name 设置为相应辅助表名称的 String 名称，如示例 1-8 所示。

unique		默认值：false。 默认情况下，JPA 持续性提供程序假设允许所有列包含重复值。 如果不允许该列包含重复值，请将 unique 设置为 true。设置为 true 时，这相当于在表级别使用 @UniqueConstraint。
updatable		默认值：true。 默认情况下，JPA 持续性提供程序假设列始终包含在 SQL UPDATE 语句中。 如果该列不应包含在这些语句中，请将 updatable 设置为 false。

示例 1-8 显示了如何使用此批注使 JPA 将 empId 持久保存到辅助表 EMP_HR 中的列 EMP_NUM。默认情况下，JPA 将 empName 持久保存到主表 Employee 中的列 empName。

示例 1-8 @Column

```
@Entity
@SecondaryTable(name="EMP_HR")
public class Employee implements Serializable {
    ...
@Column(name="EMP_NUM", table="EMP_HR")
private Long empId;

private String empName;
    ...
}
```

@ColumnResult

执行 @NamedNativeQuery 时，它可以返回实体（包括不同类型的实体）、标量值或实体和标量值的组合。

使用 @ColumnResult 批注返回标量值。标量类型由您在 @ColumnResult 中标识的列类型确定。

有关详细信息，另请参阅 @EntityResult、@FieldResult 和 @SqlResultSetMapping。

表 1-8 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-8 @ColumnResult 属性

属性	必需	说明
name	✔	在原生 SQL 查询的 SELECT 语句中将 name 设置为列名的 String 等效形式。如果在 SELECT 中使用列别名（AS 语句），则将 name 设置为列别名。

示例 1-9 显示了如何使用此批注将 Item（请参阅示例 1-10）标量 name 包含在结果列表（请参阅示例 1-11）中。在该示例中，结果列表将为 Object 数组的 List，如：{[Order, "Shoes"], [Order, "Socks"], ...}。

示例 1-9 使用 @ColumnResult 的 Order 实体

```
@SqlResultSetMapping(
```

```
name="OrderResults",
entities={
@EntityResult(
entityClass=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"
        )
    }
)
@Entity
public class Order {
@Id
protected int id;
protected long quantity;
protected Item item;
    ...
}
```

示例 1-10 Item 实体

```
@Entity
public class Item {
@Id
protected int id;
protected String name;
    ...
}
```

示例 1-11 结合使用 @SqlResultSetMapping 与 @ColumnResult 的原生查询

```
Query q = entityManager.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"
);

List resultList = q.getResultList();
// List of Object arrays:{[Order, "Shoes"], [Order, "Socks"], ...}
```

@DiscriminatorColumn

默认情况下，当 @Inheritance 属性策略为 `InheritanceType.SINGLE_TABLE` 或 `JOINED` 时，JPA 持续性提供程序将创建一个名为 `DTYPE` 的标识符列以区分继承层次中的类。

使用 `@DiscriminatorColumn` 批注：

- **指定一个标识符列名（如果数据模型中的列名不是默认列名 `DTYPE`）。**
- **指定一个适用于应用程序或事先存在的数据模型的标识符列长度**

- 微调数据库中的标识符列的特征

表 1-9 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-9 @DiscriminatorColumn 属性

属性	必需	说明
`columnDefinition`		默认值：空 `String`。 默认情况下，JPA 持续性提供程序使用最少量 SQL 创建一个数据库表列。 如果需要使用更多指定选项创建的列，请将 `columnDefinition` 设置为在针对列生成 DDL 时希望 JPA 使用的 SQL 片断。
`discriminatorType`		默认值：`DiscriminatorType.STRING`。 默认情况下，JPA 持续性提供程序假设标识符类型为 `String`。 如果要使用其他类型，请将 `discriminatorType` 设置为 `DiscriminatorType.CHAR` 或 `DiscriminatorType.INTEGER`。 您的 @DiscriminatorValue 必须符合此类型。
`length`		默认值： 31 默认情况下，JPA 持续性提供程序假设标识符列在用于保存 `String` 值时的最大长度为 255 个字符。 如果该列不适合于您的应用程序或数据库，请将 `length` 设置为适合于您的数据库列的 `int` 值。 您的 @DiscriminatorValue 必须符合此长度。
`name`		默认值：JPA 持续性提供程序假设标识符列名为“`DTYPE`”。 要指定其他列名，请将 `name` 设置为所需的 `String` 列名。

示例 1-12 显示了如何使用此批注指定一个名为 `DISC`、类型为 `STRING`、长度为 20 的标识符列。在本示例中，该类的 @DiscriminatorValue 指定为 `CUST`。示例 1-13 中的子类将它自己的 @DiscriminatorValue 指定为 `VIP`。在 `Customer` 和 `ValuedCustomer` 中，@DiscriminatorValue 的值必须可以转换为由 @DiscriminatorColumn 属性 `discriminatorType` 指定的类型，并且必须符合 @DiscriminatorColumn 属性 `length`。

示例 1-12 @DiscriminatorColumn 和 @DiscriminatorValue — 根类

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="DISC", discriminatorType=STRING, length=20)
@DiscriminatorValue(value-"CUST")
public class Customer {
    ...
}
```

示例 1-13 @DiscriminatorValue — 子类

```
@Entity
@DiscriminatorValue(value="VIP")
public class ValuedCustomer extends Customer {
    ...
}
```

@DiscriminatorValue

默认情况下，当 @Inheritance 属性策略为 `InheritanceType.SINGLE_TABLE` 或 `JOINED` 时，JPA 持续性提供程序使用 @DiscriminatorColumn 按实体名称区分继承层次中的类（请参阅 @Entity）。

使用 `@DiscriminatorValue` 批注指定用于区分此继承层次中的实体的标识符值：

- 如果实体名称不适合于此应用程序
- 匹配现有的数据库模式

表 1-10 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-10 @DiscriminatorValue 属性

属性	必需	说明
`value`	✔	将 `value` 设置为符合 @DiscriminatorColumn 属性 `discriminatorType` 和 `length` 的标识符值的 `String` 等效形式。

示例 1-14 显示了如何使用此批注指定一个名为 `DISC`、类型为 `STRING`、长度为 20 的标识符列。在本示例中，该类的 @DiscriminatorValue 指定为 `CUST`。示例 1-15 中的子类将它自己的 @DiscriminatorValue 指定为 `VIP`。在 `Customer` 和 `ValuedCustomer` 中，@DiscriminatorValue 的值必须可以转换为由 @DiscriminatorColumn 属性 `discriminatorType` 指定的类型，并且必须符合 @DiscriminatorColumn 属性 `length`。

示例 1-14 @DiscriminatorColumn 和 @DiscriminatorValue — 根类

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="DISC", discriminatorType=STRING, length=20)
@DiscriminatorValue(value-"CUST")
public class Customer {
    ...
}
```

示例 1-15 @DiscriminatorValue — 子类

```
@Entity
@DiscriminatorValue(value="VIP")
public class ValuedCustomer extends Customer {
    ...
}
```

@Embeddable

默认情况下，JPA 持续性提供程序假设每个实体均持久保存到它自己的数据库表。

使用 `@Embeddable` 批注指定一个类，该类的实例存储为拥有实体的固有部分并共享该实体的身份。嵌入对象的每个持久属性或字段都将映射到实体的数据库表。

此批注没有属性。有关更多详细信息，请参阅 API。

示例 1-16 显示了如何使用此批注指定：类 `EmploymentPeriod` 在用作批注为 @Embedded 的持久字段的类型时可以嵌套到实体中（请参阅示例 1-17）

示例 1-16 @Embeddable

```
@Embeddable
public class EmploymentPeriod {
java.util.Date startDate;
java.util.Date endDate;
    ...
}
```

@Embedded

默认情况下，JPA 持续性提供程序假设每个实体均持久保存到它自己的数据库表。

使用 `@Embedded` 批注指定一个持久字段，该字段的 @Embeddable 类型可以存储为拥有实体的固有部分，并共享该实体的身份。嵌入对象的每个持久属性或字段均映射到拥有实体的数据库表。

可以结合使用 `@Embedded` 和 @Embeddable 以建立严格所有权关系的模型，以便在删除了拥有对象的情况下还将删除被拥有的对象。

嵌入的对象不应映射到多个表。

默认情况下，@Embeddable 类中指定的列定义（请参阅 @Column）适用于 `@Embedded` 类。如果要覆盖这些列定义，请使用 @AttributeOverride。

此批注没有属性。有关更多详细信息，请参阅 API。

示例 1-17 显示了如何使用该批注指定：@Embeddable 类 `EmploymentPeriod`（请参阅示例 1-16）可以使用指定的属性覆盖（请参阅 @AttributeOverride）嵌入到实体类中。如果不需要属性覆盖，则可以完全忽略 `@Embedded` 批注：JPA 持续性提供程序将推断出 `EmploymentPeriod` 是从它的 @Embeddable 批注进行嵌套。

示例 1-17 @Embedded

```
@Entity
public class Employee implements Serializable {
    ...
@Embedded
@AttributeOverrides({
@AttributeOverride(name="startDate", column=@Column("EMP_START")),
@AttributeOverride(name="endDate", column=@Column("EMP_END"))
    )
public EmploymentPeriod getEmploymentPeriod() {
        ...
    }
    ...
}
```

@EmbeddedId

使用 `@EmbeddedId` 批注指定一个由实体拥有的可嵌入复合主键类（通常由两个或更多基元类型或 JDK 对象类型组成）。从原有数据库映射时（此时数据库键由多列组成），通常将出现复合主键。

复合主键类具有下列特征：

- 它是一个普通的旧式 Java 对象 (POJO) 类。
- 它必须为 public，并且必须有一个 public 无参数构造函数。
- 如果使用基于属性的访问，则主键类的属性必须为 public 或 protected。
- 它必须是可序列化的。
- 它必须定义 `equals` 和 `hashCode` 方法。

  这些方法的值相等性的语义必须与键映射到的数据库类型的数据库相等性一致。

或者，您可以使复合主键类成为非嵌入类（请参阅 @IdClass）。

此批注没有属性。有关更多详细信息，请参阅 API。

示例 1-18 显示了一个批注为 @Embeddable 的典型复合主键类。示例1-19 显示了如何使用可嵌入的复合主键类（使用 `@EmbeddedId` 批注）配置一个实体。

示例 1-18 可嵌入复合主键类

```
@Embeddable
public class EmployeePK implements Serializable
{
private String name;
private long id;

public EmployeePK()
    {
    }

public String getName()
    {
return name;
    }

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

public long getId()
    {
return id;
    }

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

public int hashCode()
```

```
    {
return (int) name.hashCode() + id;
    }

public boolean equals(Object obj)
    {
if (obj == this) return true;
if (!(obj instanceof EmployeePK)) return false;
if (obj == null) return false;
EmployeePK pk = (EmployeePK) obj;
return pk.id == id && pk.name.equals(name);
    }
}
```

示例 1-19 @EmbeddedId

```
@Entity
public class Employee implements Serializable
{
EmployeePK primaryKey;

public Employee()
    {
    }

@EmbeddedId
public EmployeePK getPrimaryKey()
    {
return primaryKey;
    }

public void setPrimaryKey(EmployeePK pk)
    {
primaryKey = pk;
    }

    ...
}
```

@Entity

使用 `@Entity` **批注将普通的旧式** Java **对象** (POJO) **类指定**为实**体，并使其可用于** JPA **服**务。**必**须**将** POJO **类指定**为实**体，然后才可以使用任何其他** JPA **批注**。

表 1-11 **列出了此批注的属性。有关更多**详细**信息，**请**参**阅** API。

表 1-11 @Entity 属性

属性	必需	说明
`name`		**默**认值：JPA **持**续**性提供程序假**设实**体名称是**实**体类的名称。在**示例 1-20 **中，默**认 `name` 为“`Employee`”。 **如果**实**体类名**难**于**处**理、是一个保留字、与事先存在的数据模型不兼容或作**为**数据**库**中的表名无效，**请**将 `name` 设**置**为**其他 `String` 值。

示例 1-20 显**示了**该**批注的用法**。

示例 1-20 @Entity

```
@Entity
public class Employee implements Serializable {
    ...
}
```

@EntityListeners

可以使用生命周期批注（请参阅生命周期事件批注）指定实体中的方法，这些方法在指定的生命周期事件发生时执行您的逻辑。

使用 `@EntityListeners` 批注将一个或多个实体监听程序类与 @Entity 或 @MappedSuperclass 关联，条件是您需要在指定的生命周期事件发生时执行逻辑，以及：

- 不希望在实体 API 中公开生命周期监听程序方法。
- 要在不同的实体类型之间共享生命周期监听程序逻辑。

当实体或子类上发生生命周期事件时，JPA 持续性提供程序将按监听程序定义的顺序通知每个实体监听程序，并调用使用相应的生命周期事件类型进行批注的实体监听程序方法（如果有）。

实体监听程序类具有以下特征：

- 它是一个普通的旧式 Java 对象 (POJO) 类
- 它有一个或多个具有以下签名的回调方法：

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

  可以指定参数类型 `Object`，或实体监听程序将与其关联的实体类的类型。

- 它用一个或多个生命周期事件批注对每个回调方法进行批注。

  一个生命周期事件只能与一个回调监听程序方法关联，但某个给定的回调监听程序方法可以与多个生命周期事件关联。

如果使用实体监听程序，则可以管理哪些实体监听程序使用 @ExcludeDefaultListeners 和 @ExcludeSuperclassListeners 调用。

表 1-12 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-12 @EntityListeners 属性

属性	必需	说明
`value`	✔	要为 @Entity 或 @MappedSuperclass 指定实体监听程序类的列表，请将 `value` 设置为实体监听程序类的 `Class` 数组。

示例 1-21 显示了如何使用此批注将实体监听程序类 `EmployeePersistListener`（请参阅示例 1-22）和 `EmployeeRemoveListener`（请参阅示例 1-23）与实体 `Employee` 关联。示例 1-23 显示了您可以将多个生命周期事件与给定的实体监听程序类方法关联，但任何给定的生命周期事件只能在实体监听程序类中出现一次。

示例 1-21 @EntityListeners

```
@Entity
@EntityListeners(value={EmployeePersistListner.class, EmployeeRemoveListener.class})
public class Employee implements Serializable {
    ...
}
```

示例 1-22 EmployeePersistListener

```
public class EmployeePersistListener {
@PrePersist
employeePrePersist(Object employee) {
    ...
    }
    ...
}
```

示例 1-23 EmployeeRemoveListener

```
public class EmployeeRemoveListener {
@PreRemove
@PostRemove
employeePreRemove(Object employee) {
    ...
    }
    ...
}
```

@EntityResult

执行 @NamedNativeQuery 时，它可以返回实体（包括不同类型的实体）、标量值或实体和标量值的组合。

使用 `@EntityResult` 批注返回实体。

有关详细信息，另请参阅 @ColumnResult、@FieldResult 和 @SqlResultSetMapping。

表 1-8 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-13 @EntityResult 属性

属性	必需	说明
`entityClass`	✔	将 `entityClass` 设置为由 `SELECT` 语句返回的实体的 `Class`。
`discriminatorColumn`		默认值：空 `String`。 默认情况下，JPA 持续性提供程序假设 `SELECT` 语句中不包含标识符列（请参阅 @Inheritance）。 如果在 `SELECT` 语句中使用标识符列，请将 `discriminatorColumn` 设置为所使用的 `String` 列名。

fields		默认值：空 FieldResult 数组。 默认情况下，JPA 持续性提供程序假设 SELECT 语句包含与返回的实体的所有字段或属性相对应的所有列，且 SELECT 语句中的列名对应于字段或属性名（未使用 AS 语句）。 如果 SELECT 语句只包含某些与返回的实体的字段或属性相对应的列，或 SELECT 语句中的列名并不对应于字段或属性名（使用了 AS 语句），请将 fields 设置为 @FieldResult 的数组，SELECT 语句中的每一列一个 @FieldResult。

示例 1-24 显示了如何使用此批注将 Order 和 Item（请参阅示例 1-25）实体包含在结果列表（请参阅示例 1-26）中。在该示例中，结果列表将为 Object 数组的 List，如：{[Order, Item], [Order, Item], ...}。

示例 1-24 使用 @EntityResult 的 Order 实体

```
@SqlResultSetMapping(
name="OrderResults",
entities={
@EntityResult(
entityClass=Order.class,
fields={
@FieldResult(name="id",       column="order_id"),
@FieldResult(name="quantity", column="order_quantity"),
@FieldResult(name="item",     column="order_item")
            }
        ),
@EntityResult(
entityClass=Item.class,
fields={
@FieldResult(name="id",       column="item_id"),
@FieldResult(name="name",     column="item_name"),
            }
        )
    }
)
@Entity
public class Order {
@Id
protected int id;
protected long quantity;
protected Item item;
    ...
}
```

示例 1-25 Item 实体

```
@Entity
public class Item {
@Id
protected int id;
protected String name;
    ...
}
```

示例 1-26 结合使用 @SqlResultSetMapping 与 @EntityResult 的原生查询

```
Query q = entityManager.createNativeQuery(
"SELECT o.id       AS order_id, " +
"o.quantity AS order_quantity, " +
"o.item     AS order_item, " +
```

```
"i.id       AS item_id, " +
"i.name     AS item_name, " +
"FROM Order o, Item i " +
"WHERE (order_quantity > 25) AND (order_item = i.id)",
"OrderResults"
);

List resultList = q.getResultList();
// List of Object arrays:{[Order, Item], [Order, Item], ...}
```

@Enumerated

默认情况下，JPA 持续性提供程序持久保存枚举常量的序数值。

使用 `@Enumerated` 批注指定在 String 值适合应用程序要求或与现有数据库模式匹配的情况下，JPA 持续性提供程序是否应持久保存枚举常量的序数值或 `String` 值。

该批注可以与 @Basic 一起使用。

表 1-14 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-14 @Enumerated 属性

属性	必需	说明
`value`		默认值：`EnumType.ORDINAL`。 默认情况下，JPA 持续性提供程序假设对于映射到枚举常量的属性或字段，应持久保存序数值。在示例 1-28 中，当持久保存 `Employee` 时，`EmployeeStatus` 的序数值将写入数据库。 如果需要持久保存的枚举常量的 String 值，请将 `value` 设置为 `EnumType.STRING`。

根据示例 1-27 中的枚举常量，示例 1-28 显示了如何使用此批注指定在持久保存 `Employee` 时应将 `SalaryRate` 的 `String` 值写入数据库。默认情况下，会将 `EmployeeStatus` 的序数值写入数据库。

示例 1-27 枚举常量

```
public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}
public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}
```

示例 1-28 @Enumerated

```
@Entity
public class Employee {
    ...
public EmployeeStatus getStatus() {
    ...
    }

@Enumerated(STRING)
public SalaryRate getPayScale() {
    ...
    }
    ...
}
```

@ExcludeDefaultListeners

默认监听程序是 orm.xml 文件中指定的一个生命周期事件监听程序类，该类应用于持续性单元（请参阅 @PersistenceUnit）中的所有实体。在调用任何其他实体监听程序（请参阅 @EntityListeners）之前，JPA 持续性提供程序首先按照 orm.xml 文件中定义的顺序调用默认监听程序（如果有）。

如果默认监听程序行为不适用，请使用 @ExcludeDefaultListeners 批注覆盖（并阻止）针对给定 @Entity 或 @MappedSuperclass 执行的默认监听程序。

此批注没有属性。有关更多详细信息，请参阅 API。

示例 1-29 显示了如何使用此批注指定不应对 Employee 实体执行默认监听程序。

示例 1-29 @ExcludeDefaultListeners

```
@Entity
@ExcludeDefaultListeners
public class Employee implements Serializable {
    ...
}
```

@ExcludeSuperclassListeners

如果继承层次中的 @Entity 和 @MappedSuperclass 类定义了 @EntityListeners，则默认情况下，JPA 持续性提供程序将在调用子类监听程序之前调用超类监听程序。

如果超类监听程序行为不适用，则使用 @ExcludeSuperclassListeners 批注覆盖（并阻止）针对给定 @Entity 或 @MappedSuperclass 执行的超类监听程序。

@ExcludeSuperclassListeners 批注不影响默认监听程序（请参阅 @ExcludeDefaultListeners）。

此批注没有属性。有关更多详细信息，请参阅 API。

示例 1-29 显示了如何使用此批注指定不应对 PartTimeEmployee 实体执行超类监听程序 EmployeeListener，而是执行默认监听程序和子类监听程序 PartTimeEmployeeListener1 和 PartTimeEmployeeListener2。

示例 1-30 超类级别的实体监听程序

```
@MappedSuperclass
@EntityListeners(value={EmployeeListener.class})
public class Employee {
    ...
}
```

示例 1-31 子类级别的 @ExcludeSuperclassListeners

```
@Entity
@ExcludeSuperclassListeners
@EntityListners(value={PartTimeEmployeeListener1.class, PartTimeEmployeeListener2.class})
public class PartTimeEmployee extends Employee {
    ...
}
```

@FieldResult

执行 @NamedNativeQuery 时，它可以返回实体（包括不同类型的实体）、标量值或实体和标量值的组合。

默认情况下，JPA 持续性提供程序假设在使用 @EntityResult 返回实体时，SELECT 语句将包含与返回的实体的所有字段或属性相对应的所有列，且 SELECT 语句中的列名对应于字段或属性名（未使用 AS 语句）。

如果 SELECT 语句只包含某些与返回的实体的字段或属性相对应的列，或 SELECT 语句中的列名并不对应于字段或属性名（使用了 AS 语句），则在使用 @EntityResult 返回实体时，请使用 @FieldResult 批注将 SELECT 语句中的列映射到字段或属性。

有关详细信息，另请参阅 @ColumnResult 和 @SqlResultSetMapping。

表 1-15 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-15 @FieldResult 属性

属性	必需	说明
column	✔	将 column 设置为 SELECT 语句中使用的列的 String 名称。如果在 SELECT 中使用列别名（AS 语句），请将 column 设置为列别名。
name	✔	将 name 设置为实体的字段或属性名（作为 String），该名称对应于 column 属性指定的列名。

示例 1-32 显示了如何使用此批注将 Order 和 Item（请参阅示例 1-33）实体包含在结果列表（请参阅示例 1-34）中。在该示例中，结果列表将为 Object 数组的 List，如：{[Order, Item], [Order, Item], ...}。

示例 1-32 使用 @EntityResult 和 @FieldResult 的 Order 实体

```
@SqlResultSetMapping(
name="OrderResults",
entities={
@EntityResult(
entityClass=Order.class,
fields={
@FieldResult(name="id",       column="order_id"),
@FieldResult(name="quantity", column="order_quantity"),
@FieldResult(name="item",     column="order_item")
             }
         ),
@EntityResult(
entityClass=Item.class,
fields={
@FieldResult(name="id",       column="item_id"),
@FieldResult(name="name",     column="item_name"),
             }
         )
    }
)
@Entity
public class Order {
@Id
protected int id;
protected long quantity;
protected Item item;
    ...
}
```

示例 1-33 Item 实体

```
@Entity
public class Item {
@Id
protected int id;
protected String name;
    ...
}
```

示例 1-34 结合使用 @SqlResultSetMapping 与 @EntityResult 的原生查询

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

List resultList = q.getResultList();
// List of Object arrays:{[Order, Item], [Order, Item], ...}
```

@GeneratedValue

默认情况下，JPA 持续性提供程序管理为实体主键提供的唯一标识符（请参阅 @Id）。

如果要微调此机制以实现以下目的，请使用 `@GeneratedValue` 批注：

- 如果您感觉另一个生成器类型更适合于数据库或应用，则覆盖持续性提供程序为数据库选择的身份值生成的类型
- 如果此名称难于处理、是一个保留字、与事先存在的数据模型不兼容或作为数据库中的主键生成器名称无效，则覆盖持续性提供程序选择的主键生成器名称

表 1-16 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-16 @GeneratedValue 属性

属性	必需	说明
`generator`		默认值：JPA 持续性提供程序为它选择的主键生成器分配一个名称。 如果该名称难于处理、是一个保留字、与事先存在的数据模型不兼容或作为数据库中的主键生成器名称无效，则将 `generator` 设置为要使用的 `String` 生成器名称。

strategy		默认值：GenerationType.AUTO。 默认情况下，JPA 持续性提供程序选择最适合于基础数据库的主键生成器类型。 如果您感觉另一个生成器类型更适合于数据库或应用程序，请将 strategy 设置为所需的 GeneratorType： • IDENTITY — 指定持续性提供程序使用数据库身份列 • AUTO — 指定持续性提供程序应选择一个最适合于基础数据库的主键生成器。 • SEQUENCE — 指定持续性提供程序使用数据库序列（请参阅 @SequenceGenerator） • TABLE — 指定持续性提供程序为使用基础数据库表的实体分配主键以确保唯一性（请参阅 @TableGenerator）

示例 1-35 显示了如何使用此批注指示持续性提供程序使用名为 CUST_SEQ、类型为 GeneratorType.SEQUENCE 的主键生成器。

示例 1-35 @GeneratedValue

```
@Entity
public class Employee implements Serializable {
    ...
@Id
@GeneratedValue(strategy=SEQUENCE, generator="CUST_SEQ")
@Column(name="CUST_ID")
public Long getId() {
return id;
    }
    ...
}
```

@Id

使用 @Id 批注将一个或多个持久字段或属性指定为实体的主键。

对于每个实体，必须至少指定以下项之一：

- 一个 @Id
- 多个 @Id 和一个 @IdClass（对于复合主键）
- 一个 @EmbeddedId

此批注没有属性。有关更多详细信息，请参阅 API。

默认情况下，JPA 持续性提供程序选择最合适的主键生成器（请参阅 @GeneratedValue）并负责管理主键值：您不必采取任何进一步的操作。如果要使用 JPA 持续性提供程序的默认键生成机制，则不必采取任何进一步的操作。

示例 1-36 显示了如何使用此批注将持久字段 empID 指定为 Employee 表的主键。

示例 1-36 @Id

```
@Entity
public class Employee implements Serializable {
@Id
private int empID;
    ...
}
```

@IdClass

使用 `@IdClass` 批注为实体指定一个复合主键类（通常由两个或更多基元类型或 JDK 对象类型组成）。从原有数据库映射时（此时数据库键由多列组成），通常将出现复合主键。

复合主键类具有下列特征：

- 它是一个普通的旧式 Java 对象 (POJO) 类。
- 它必须为 public，并且必须有一个 public 无参数构造函数。
- 如果使用基于属性的访问，则主键类的属性必须为 public 或 protected。
- 它必须是可序列化的。
- 它必须定义 `equals` 和 `hashCode` 方法。

  这些方法的值相等性的语义必须与键映射到的数据库类型的数据库相等性一致。
- 它的字段或属性的类型和名称必须与使用 @Id 进行批注的实体主键字段或属性的类型和名称相对应。

或者，您可以使复合主键类成为由实体拥有的嵌入类（请参阅 @EmbeddedId）。

表 1-17 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-17 @IdClass 属性

属性	必需	说明
`value`	✔	要指定复合主键类，请将 `value` 设置为所需的 `Class`（请参阅 @AttributeOverride）。

示例 1-37 显示了一个非嵌入的复合主键类。在该类中，字段 `empName` 和 `birthDay` 的名称和类型必须对应于实体类中属性的名称和类型。示例 1-38 显示了如何使用这个非嵌入的复合主键类（使用 `@IdClass` 批注）配置 EJB 3.0 实体。由于实体类字段 `empName` 和 `birthDay` 在主键中使用，因此还必须使用 @Id 批注对其进行批注。

示例 1-37 非嵌入的复合主键类

```
public class EmployeePK implements Serializable
{
private String empName;
private Date birthDay;

public EmployeePK()
```

```
    {
    }

public String getName()
    {
return empName;
    }

public void setName(String name)
    {
empName = name;
    }

public long getDateOfBirth()
    {
return birthDay;
    }

public void setDateOfBirth(Date date)
    {
birthDay = date;
    }

public int hashCode()
    {
return (int) empName.hashCode();
    }

public boolean equals(Object obj)
    {
if (obj == this) return true;
if (!(obj instanceof EmployeePK)) return false;
if (obj == null) return false;
EmployeePK pk = (EmployeePK) obj;
return pk.birthDay == birthDay && pk.empName.equals(empName);
    }
}
```

示例 1-38 @IdClass

```
@IdClass(EmployeePK.class)
@Entity
public class Employee
{
@Id String empName;
@Id Date birthDay;
...
}
```

@Inheritance

默认情况下，JPA 持续性提供程序自动管理继承层次中实体的持续性。

使用 `@Inheritance` 批注自定义持续性提供程序的继承层次支持，以提高应用程序性能或匹配现有的数据模型。

表 1-18 **列出了此批注的属性。有关更多详细信息，请参阅** API。

表 1-18 @Inheritance 属性

属性	必需	说明

strategy		默认值：InheritanceType.SINGLE_TABLE。 默认情况下，JPA 持续性提供程序假设层次中的所有类均映射到一个由表的标识符列（请参阅 @DiscriminatorColumn）中的标识符值（请参阅 @DiscriminatorValue）区分的表。 如果这并不适合于应用程序，或者如果必须匹配现有的数据模型，请将 strategy 设置为所需的 InheritanceType： • SINGLE_TABLE[Foot?1?] — 层次中的所有类均映射到一个表。该表有一个标识符列（请参阅 @DiscriminatorColumn），它的值（请参阅 @DiscriminatorValue）标识由行表示的实例所属的特定子类。 • TABLE_PER_CLASS — 每个类均映射到单独的表。该类的所有属性（包括继承的属性）映射到该类的表列。 • JOINED — 类层次的根由一个表表示，而每个子类由单独的表表示。每个子类表只包含特定于该子类的那些字段（而非从其超类继承的字段）和主键列，这些主键列用作超类表主键的外键。

Footnote?1?该选项为跨类层次的实体和查询之间的多态关系提供了最佳支持。该选项的缺点包括需要生成应为 NOT NULL 的可空列。

示例 1-39 显示了如何使用此批注指定 Customer 的所有子类将使用 InheritanceType.JOINED。示例 1-40 中的子类将映射到它自己的表（该表针对 ValuedCustomer 的每个持久属性包含一列）和一个外键列（包含 Customer 表的主键）。

示例 1-39 @Inheritance — 使用 JOINED 的根类

```
@Entity@Inheritance(strategy=JOINED)public class Customer {
@Id
private int customerId;
    ...
}
```

示例 1-40 @Inheritance — 使用 JOINED 的子类

```
@Entity
public class ValuedCustomer extends Customer {
    ...
}
```

在示例 1-41 中，默认情况下，InheritanceType.SINGLE_TABLE 应用于 Customer 及其所有子类。在该示例中，默认标识符表列 DTYPE（请参阅 @DiscriminatorColumn）指定为具有标识符类型 INTEGER，且 Customer 的 @DiscriminatorValue 指定为 1。示例 1-42 显示了如何将子类 ValuedCustomer 的标识符值指定为 2。在该示例中，Customer 和 ValuedCustomer 的所有持久属性将映射到一个表。

示例 1-41 @Inheritance — 指定其标识符列的根类

```
@Entity
@DiscriminatorColumn(discriminatorType=DiscriminatorType.INTEGER)
@DiscriminatorValue(value="1")
public class Customer {
    ...
}
```

示例 1-42 @Inheritance — 指定其标识符值的子类

```
@Entity
@DiscriminatorValue(value="2")
public class ValuedCustomer extends Customer {
    ...
}
```

@JoinColumn

默认情况下，在实体关联中，JPA 持续性提供程序使用一个基于现有名称（如字段或属性名称）的数据库模式，以便它可以自动确定要使用的单个连接列（包含外键的列）。

在以下条件下使用 `@JoinColumn` 批注：

- 默认连接列名称难于处理、是一个保留字、与预先存在的数据模型不兼容或作为数据库中的列名无效
- 您需要使用外部表中的列（非主键列）进行连接
- 您想要使用两个或更多连接列（请参阅 @JoinColumns）
- 您想要使用一个连接表（请参阅 @JoinTable）

表 1-19 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-19 @JoinColumn 属性

属性	必需	说明
`columnDefinition`		默认值：空 `String`。 JPA 使用最少量 SQL 创建一个数据库表列。 如果需要使用更多指定选项创建列，请将 `columnDefinition` 设置为在针对列生成 DDL 时希望 JPA 使用的 `String` SQL 片断。
`insertable`		默认值：`true`。 默认情况下，JPA 持续性提供程序假设它可以插入到所有表列中。 如果该列为只读，请将 `insertable` 设置为 `false`。
`name`		默认值：如果使用一个连接列，则 JPA 持续性提供程序假设外键列的名称是以下名称的连接： • 引用关系属性的名称 +"_"+ 被引用的主键列的名称。 • 引用实体的字段名称 +"_"+ 被引用的主键列的名称。 如果实体中没有这样的引用关系属性或字段（请参阅 @JoinTable），则连接列名称格式化为以下名称的连接：实体名称 +"_"+ 被引用的主键列的名称。 这是外键列的名称。如果连接针对"一对一"或"多对一"实体关系，则该列位于源实体的表中。如果连接针对"多对多"实体关系，则该列位于连接表（请参阅 @JoinTable）中。

		如果连接列名难于处理、是一个保留字、与预先存在的数据模型不兼容或作为数据库中的列名无效，请将 `name` 设置为所需的 `String` 列名。
`nullable`		默认值：`true`。 默认情况下，JPA 持续性提供程序假设允许所有列包含空值。 如果不允许该列包含空值，请将 `nullable` 设置为 `false`。
`referencedColumnName`		默认值：如果使用一个连接列，则 JPA 持续性提供程序假设在实体关系中，被引用的列名是被引用的主键列的名称。 如果在连接表（请参阅 @JoinTable）中使用，则被引用的键列位于拥有实体（如果连接是反向连接定义的一部分，则为反向实体）的实体表中。 要指定其他列名，请将 `referencedColumnName` 设置为所需的 `String` 列名。
`table`		默认值：JPA 持续性提供程序假设实体的所有持久字段存储到一个名称为实体类名称的数据库表中（请参阅 @Table）。 如果该列与辅助表关联（请参阅 @SecondaryTable），请将 `name` 设置为相应辅助表名称的 `String` 名称，如示例 1-8 所示。
`unique`		默认值：`false`。 默认情况下，JPA 持续性提供程序假设允许所有列包含重复值。 如果不允许该列包含重复值，请将 `unique` 设置为 `true`。
`updatable`		默认值：`true`。 默认情况下，JPA 持续性提供程序假设它可以更新所有表列。 如果该列为只读，则将 `updatable` 设置为 `false`

示例 1-43 显示了如何使用此批注使 JPA 将数据库表 `Employee` 列 `ADDR_ID` 用作连接列。

示例 1-43 @JoinColumn

```
@Entity
public class Employee implements Serializable {
    ...
@ManyToOne
@JoinColumn(name="ADDR_ID")
public Address getAddress() {
return address;
    }
}
```

@JoinColumns

默认情况下，在实体关联中，JPA 持续性提供程序假设使用一个连接列。

如果要指定两个或更多连接列（即复合主键），请使用 `@JoinColumns` 批注。

表 1-20 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-20 @JoinColumns 属性

属性	必需	说明
value	✔	要指定两个或更多连接列，请将 value 设置为 JoinColumn 实例数组（请参阅 @JoinColumn）。

示例 1-44 显示了如何使用此批注指定两个连接列的名称：Employee 表中的 ADDR_ID（其中包含 Address 表列 ID 中的外键值）以及 Employee 表中的 ADDR_ZIP（其中包含 Address 表列 ZIP 中的外键值）。

示例 1-44 @JoinColumns

```
@Entity
public class Employee implements Serializable {
    ...
@ManyToOne
@JoinColumns({
@JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
@JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
    })
public Address getAddress() {
return address;
    }
    ...
}
```

@JoinTable

默认情况下，JPA 持续性提供程序在映射多对多关联（或在单向的一对多关联中）的拥有方上的实体关联时使用一个连接表。连接表名称及其列名均在默认情况下指定，且 JPA 持续性提供程序假设：在关系的拥有方上的实体主表中，每个主键列有一个连接列。

如果您需要执行以下操作，请使用 @JoinTable 批注：

- 由于默认名称难于处理、是一个保留字、与预先存在的数据模型不兼容或作为数据库中的表名无效而更改连接表的名称
- 由于默认名称难于处理、是一个保留字、与预先存在的数据模型不兼容或作为数据库中的列名无效而更改连接表的列名称
- 使用特定目录或模式配置连接表
- 使用唯一约束配置一个或多个连接表列
- 每个实体使用多个连接列

表 1-21 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-21 @JoinTable 属性

属性	必需	说明

catalog		默认值：空 String。 默认情况下，JPA 使用任何适用于数据库的默认目录。 如果默认目录不适合于应用程序，请将 catalog 设置为要使用的 String 目录名。
inverseJoinColumns		默认值：JoinColumn 的空数组。 默认情况下，JPA 持续性提供程序假设关联的被拥有方（或另一方）上有一个连接列：被拥有实体的主键列。JPA 通过连接被拥有实体的名称 +"_"+ 被引用的主键列的名称来命名该列。 如果这样的列名难于处理、是一个保留字、与预先存在的数据模型不兼容，或者如果要指定多个连接列，则将 joinColumns 设置为 JoinColumn（请参阅 @JoinColumn）的一个或多个实例。
joinColumns		默认值：JoinColumn 的空数组。 默认情况下，JPA 持续性提供程序假设：拥有实体的每个主键列都有一个连接列。该持续性提供程序通过连接拥有实体的名称+"_"+ 被引用主键列的名称来命名这些列。 如果这样的列名难于处理、是一个保留字、与预先存在的数据模型不兼容，或者如果要指定多个连接列，则将 joinColumns 设置为 JoinColumn（请参阅 @JoinColumn）的一个或多个实例。
name		默认值：JPA 持续性提供程序通过使用下划线连接关联主表（拥有方优先）的表名来命名连接表。 如果这样的连接表难于处理、是一个保留字或与预先存在的数据模型不兼容，则将 name 设置为相应的连接表名。在示例 1-45 中，JPA 使用名为 EJB_PROJ_EMP 的连接表。
schema		默认值：空 String。 默认情况下，JPA 使用任何适用于数据库的默认模式。 如果默认模式不适合于应用程序，则将 schema 设置为要使用的 String 模式名。
uniqueConstraints		默认值：UniqueConstraint 的空数组。 默认情况下，JPA 持续性提供程序假设连接表中的任何列均没有唯一约束。 如果唯一约束应用于该表中的一列或多列，则将 uniqueContraints 设置为一个或多个 UniqueConstraint 实例的数组。有关详细信息，请参阅 @UniqueConstraint。

示例 1-45 显示了如何使用此批注为 Employee 与 Project 之间实体的多对多关系指定一个名为 EMP_PROJ_EMP 的连接表。连接表中有两列：EMP_ID 和 PROJ_ID。EMP_ID 列包含其主键列（被引用列）名为 ID 的 Employee 表中的主键值。PROJ_ID 列包含其主键列（被引用列）也名为 ID 的 Project 表中的主键值。

示例 1-45 @JoinTable

```
@Entity
public class Employee implements Serializable {
    ...
@ManyToMany
@JoinTable(
name="EJB_PROJ_EMP",
```

```
joinColumns=@JoinColumn(name="EMP_ID", referencedColumnName="ID"),
inverseJoinColumns=@JoinColumn(name="PROJ_ID", referencedColumnName="ID")
    )
public Collection getProjects() {
return projects;
    }
    ...
}
```

@Lob

默认情况下，JPA 持续性提供程序假设所有持久数据均可以表示为典型的数据库数据类型。

结合使用 `@Lob` 批注与 @Basic 映射，以指定持久属性或字段应作为大型对象持久保存到数据库支持的大型对象类型。

Lob 可以是二进制类型或字符类型。持续性提供程序从持久字段或属性的类型推断出 Lob 类型。

对于基于字符串和字符的类型，默认值为 Clob。在所有其他情况下，默认值为 Blob。

还可以使用 @Column 属性 `columnDefinition` 进一步改进 `Lob` 类型。

此批注没有属性。有关更多详细信息，请参阅 API。

示例 1-46 显示了如何使用此批注指定持久字段 `pic` 应作为 `Blob` 进行持久保存。

示例 1-46 @Lob

```
@Entity
public class Employee implements Serializable {
    ...
@Lob
@Basic(fetch=LAZY)
@Column(name="EMP_PIC", columnDefinition="BLOB NOT NULL")
protected byte[] pic;
    ...
}
```

@ManyToMany

默认情况下，JPA 为具有多对多多重性的为多值关联自动定义一个 `@ManyToMany` 映射。

使用 `@ManyToMany` 批注：

- 将获取类型配置为 `LAZY`
- 如果空值不适合于应用程序，则将映射配置为禁止空值（针对非基元类型）
- 由于所使用的 `Collection` 不是使用一般参数定义的，因此配置关联的目标实体
- 配置必须层叠到关联目标的操作：例如，如果删除了拥有实体，则确保还删除关联的目标
- 配置由持续性提供程序使用的连接表的详细信息（请参阅 @JoinTable）

表 1-22 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-22 @ManyToMany 属性

属性	必需	说明
cascade		默认值：CascadeType 的空数组。 默认情况下，JPA 不会将任何持续性操作层叠到关联的目标。 如果希望某些或所有持续性操作层叠到关联的目标，请将 cascade 设置为一个或多个 CascadeType 实例，其中包括： • ALL — 针对拥有实体执行的任何持续性操作均层叠到关联的目标。 • MERGE — 如果合并了拥有实体，则将 merge 层叠到关联的目标。 • PERSIST — 如果持久保存拥有实体，则将 persist 层叠到关联的目标。 • REFRESH — 如果刷新了拥有实体，则 refresh 为关联的层叠目标。 • REMOVE — 如果删除了拥有实体，则还删除关联的目标。
fetch		默认值：FetchType.EAGER。 默认情况下，JPA 持续性提供程序使用获取类型 EAGER：这将要求持续性提供程序运行时必须迫切获取数据。 如果这不适合于应用程序或特定的持久字段，请将 fetch 设置为 FetchType.LAZY：这将提示持续性提供程序在首次访问数据（如果可以）时应不急于获取数据。
mappedBy		默认值：如果关系是单向的，则 JPA 持续性提供程序确定拥有该关系的字段。 如果关系是双向的，则将关联的反向（非拥有）一方上的 mappedBy 属性设置为拥有该关系的字段或属性的名称（如示例 1-48 所示）。
targetEntity		默认值：使用一般参数定义的 Collection 的参数化类型。 默认情况下，如果使用通过一般参数定义的 Collection，则持续性提供程序将从被引用的对象类型推断出关联的目标实体。 如果 Collection 不使用一般参数，则必须指定作为关联目标的实体类：将关联拥有方上的 targetEntity 元素设置为作为关系目标的实体的 Class。

示例 1-47 和示例 1-48 显示了如何使用此批注在使用一般参数的 Customer 和 PhoneNumber 之间配置一个多对多映射。

示例 1-47 @ManyToMany — 使用一般参数的 Customer 类

```
@Entity
public class Customer implements Serializable {
    ...
@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;
    }
    ...
}
```

示例 1-48 @ManyToMany — 使用一般参数的 PhoneNumber 类

```
@Entity
public class PhoneNumber implements Serializable {
    ...
@ManyToMany(mappedBy="phones")
public Set<Customer> getCustomers() {
return customers;
    }
    ...
}
```

@ManyToOne

默认情况下，JPA 为指向具有多对一多重性的其他实体类的单值关联自动定义一个 `ManyToOne` 映射。

使用 `@ManyToOne` 批注：

- 将获取类型配置为 `LAZY`
- 如果空值不适合于应用程序，则将映射配置为禁止空值（针对非基元类型）
- 配置关联的目标实体（如果无法从被引用的对象类型推断出它）
- 配置必须层叠到关联目标的操作：例如，如果删除了拥有实体，则确保还删除关联的目标

表 1-23 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-23 @ManyToOne 属性

属性	必需	说明
`cascade`		默认值：`CascadeType` 的空数组。 默认情况下，JPA 不会将任何持续性操作层叠到关联的目标。 如果希望某些或所有持续性操作层叠到关联的目标，请将 `cascade` 设置为一个或多个 `CascadeType` 实例，其中包括： • `ALL` — 针对拥有实体执行的任何持续性操作均层叠到关联的目标。 • `MERGE` — 如果合并了拥有实体，则将 merge 层叠到关联的目标。 • `PERSIST` — 如果持久保存拥有实体，则将 persist 层叠到关联的目标。

		• REFRESH — 如果刷新了拥有实体，则 refresh 为关联的层叠目标。 • REMOVE — 如果删除了拥有实体，则还删除关联的目标。
fetch		默认值：FetchType.EAGER。 默认情况下，JPA 持续性提供程序使用获取类型 EAGER：这将要求持续性提供程序运行时必须迫切获取数据。 如果这不适合于应用程序或特定的持久字段，请将 fetch 设置为 FetchType.LAZY：这将提示持续性提供程序在首次访问数据（如果可以）时应不急于获取数据。
optional		默认值：true。 默认情况下，JPA 持续性提供程序假设所有（非基元）字段和属性的值可以为空。 如果这并不适合于您的应用程序，请将 optional 设置为 false。
targetEntity		默认值：JPA 持续性提供程序从被引用的对象类型推断出关联的目标实体 如果持续性提供程序无法推断出目标实体的类型，则将关联拥有方上的 targetEntity 元素设置为作为关系目标的实体的 Class。

示例 1-49 显示了如何使用此批注在使用一般参数的 Customer（被拥有方）和 Order（拥有方）之间配置一个多对一映射。

示例 1-49 @ManyToOne

```
@Entity
public class Order implements Serializable {
    ...
@ManyToOne(optional=false)
@JoinColumn(name="CUST_ID", nullable=false, updatable=false)
public Customer getCustomer() {
return customer;
    }
    ...
}
```

@MapKey

默认情况下，JPA 持续性提供程序假设关联实体的主键为 java.util.Map 类型的关联的 Map 键：

- 如果主键是批注为 @Id 的非复合主键，则该字段或属性的类型实例将用作 Map 键。
- 如果主键是批注为 @IdClass 的复合主键，则主键类的实例将用作 Map 键。

使用 @MapKey 批注：

- 将某个其他字段或属性指定为 Map 键（如果关联实体的主键不适合于应用程序）
- 指定一个嵌入的复合主键类（请参阅 @EmbeddedId）

指定的字段或属性必须具有唯一约束（请参阅 @UniqueConstraint）。

表 1-24 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-24 @MapKey 属性

属性	必需	说明
name		默认值：默认情况下，JPA 持续性提供程序将关联实体的主键作为 Map 键，以用于映射到非复合主键或复合主键（批注为 @IdClass）的 java.util.Map 的属性或字段。 如果要将某个其他字段或属性用作 Map 键，请将 name 设置为要使用的关联实体的 String 字段或属性名。

在示例 1-52 中，Project 对作为 Map 的 Employee 实例拥有一对多关系。示例 1-52 显示了如何使用 @MapKey 批注指定此 Map 的键为 Employee 字段 empPK，它是一个类型为 EmployeePK（请参阅示例 1-52）的嵌入式复合主键（请参阅示例 1-51）。

示例 1-50 使用 @MapKey 的 Project 实体

```
@Entitypublic class Project {    ...@OneToMany(mappedBy="project")    @MapKey
(name="empPK")    public Map<EmployeePK, Employee> getEmployees() {    ...
    }    ...}
```

示例 1-51 Employee 实体

```
@Entitypublic class Employee {    @EmbeddedId public EmployeePK getEmpPK() {     ...
}    ...    @ManyToOne    @JoinColumn(name="proj_id")    public Project getProject() {
    ...
    }...}
```

示例 1-52 EmployeePK 复合主键类

```
@Embeddablepublic class EmployeePK {    String name;    Date birthDate;}
```

@MappedSuperclass

默认情况下，JPA 持续性提供程序假设实体的所有持久字段均在该实体中定义。

使用 @MappedSuperclass 批注指定一个实体类从中继承持久字段的超类。当多个实体类共享通用的持久字段或属性时，这将是一个方便的模式。

您可以像对实体那样使用任何直接和关系映射批注（如 @Basic 和 @ManyToMany）对该超类的字段和属性进行批注，但由于没有针对该超类本身的表存在，因此这些映射只适用于它的子类。继承的持久字段或属性属于子类的表。

可以在子类中使用 @AttributeOverride 或 @AssociationOverride 批注来覆盖超类的映射配置。

该批注没有属性。有关更多详细信息，请参阅 API。

示例 1-53 显示了如何使用此批注将 Employee 指定为映射超类。示例 1-54 显示了如何扩展实体中的此超类，以及如何在实体类中使用 @AttributeOverride 以覆盖超类中设置的配置。

示例 1-53 @MappedSuperclass

```
@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) {
        ...
    }
}
```

示例 1-54 扩展 @MappedSuperclass

```
@Entity
@AttributeOverride(name="address", column=@Column(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {

@Column(name="WAGE")
protected Float hourlyWage;

public PartTimeEmployee() {
        ...
    }

public Float getHourlyWage() {
        ...
    }

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

@NamedNativeQueries

如果需要指定多个 @NamedNativeQuery**，则必须使用一个** `@NamedNativeQueries` **批注指定所有命名查询。**

表 1-5 **列出了此批注的属性。有关更多详细信息，请参阅** API**。**

表 1-25 @NamedNativeQueries 属性

属性	必需	说明

value	✔	要指定两个或更多属性覆盖，请将 value 设置为 NamedNativeQuery 实例数组（请参阅 @NamedNativeQuery）。

示例 1-6 显示了如何使用此批注指定两个命名原生查询。

示例 1-55 @NamedNativeQueries

```
@Entity
@NamedNativeQueries({
@NamedNativeQuery(
name="findAllPartTimeEmployees",
query="SELECT * FROM EMPLOYEE WHERE PRT_TIME=1"
    ),
@NamedNativeQuery(
name="findAllSeasonalEmployees",
query="SELECT * FROM EMPLOYEE WHERE SEASON=1"
    )
})
public class PartTimeEmployee extends Employee {
    ...
}
```

@NamedNativeQuery

在使用 JPA 持续性提供程序的应用程序中，可以使用实体管理器动态创建和执行查询，也可以预定义查询并在运行时按名称执行。

使用 @NamedNativeQuery 批注创建与 @Entity 或 @MappedSuperclass 关联的预定义查询，这些查询：

- 使用基础数据库的原生 SQL
- 经常被使用
- 比较复杂并且难于创建
- 可以在不同实体之间共享
- 返回实体、标量值或两者的组合（另请参阅 @ColumnResult、@EntityResult、@FieldResult 和 @SqlResultSetMapping）

如果有多个要定义的 @NamedNativeQuery，则必须使用 @NamedNativeQueries。

要预定义适合于任何数据库的可移植查询，请参阅 @NamedQuery。

表 1-6 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-26 @NamedNativeQuery 属性

属性	必需	说明
query	✔	要指定查询，请将 query 设置为 SQL 查询（作为 String）。 有关原生 SQL 查询语言的详细信息，请参阅数据库文档。

hints		默认值：空 QueryHint 数组。 默认情况下，JPA 持续性提供程序假设 SQL 查询应完全按照 query 属性提供的方式执行。 要微调查询的执行，可以选择将 hints 设置为一个 QueryHint 数组（请参阅 @QueryHint）。在执行时，EntityManager 将向基础数据库传递提示。
name	✔	要指定查询名称，请将 name 设置为所需的 String 名称。 这是您在运行时调用查询所使用的名称（请参阅示例 1-60）。
resultClass		默认值：JPA 持续性提供程序假设结果类是关联实体的 Class。 要指定结果类，请将 resultClass 设置为所需的 Class。
resultSetMapping		默认值：JPA 持续性提供程序假设原生 SQL 查询中的 SELECT 语句：返回一个类型的实体；包括与返回的实体的所有字段或属性相对应的所有列；并使用与字段或属性名称（未使用 AS 语句）相对应的列名。 要控制 JPA 持续性提供程序如何将 JDBC 结果集映射到实体字段或属性以及标量，请通过将 resultSetMapping 设置为所需的 @SqlResultSetMapping 的 String 名称来指定结果集映射。

示例 1-59 显示了如何使用 @NamedNativeQuery 批注定义一个使用基础数据库的原生 SQL 的查询。示例 1-60 显示了如何使用 EntityManager 获取此查询以及如何通过 Query 方法 getResultList 执行该查询。

示例 1-56 使用 @NamedNativeQuery 实现一个 Oracle 层次查询

```
@Entity
@NamedNativeQuery(
name="findAllEmployees",
query="SELECT * FROM EMPLOYEE"
)
public class Employee implements Serializable {
    ...
}
```

示例 1-57 执行一个命名原生查询

```
Query queryEmployees = em.createNamedQuery("findAllEmployees");
Collection employees = queryEmployees.getResultList();
```

@NamedQueries

如果需要指定多个 @NamedQuery，则必须使用一个 @NamedQueries 批注指定所有命名查询。

表 1-5 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-27 @NamedQueries 属性

属性	必需	说明

value	✔	要指定两个或更多属性覆盖，请将 value 设置为 NamedQuery 实例数组（请参阅 @NamedQuery）。

示例 1-6 显示了如何使用此批注指定两个命名查询。

示例 1-58 @NamedQueries

```
@Entity
@NamedQueries({
@NamedQuery(
name="findAllEmployeesByFirstName",
query="SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
    ),
@NamedQuery(
name="findAllEmployeesByLasttName",
query="SELECT OBJECT(emp) FROM Employee emp WHERE emp.lasstName = :lastname"
    )
})
public class PartTimeEmployee extends Employee {
    ...
}
```

@NamedQuery

在使用 JPA 持续性提供程序的应用程序中，可以使用实体管理器动态创建和执行查询，也可以预定义查询并在运行时按名称执行。

使用 @NamedQuery 批注创建与 @Entity 或 @MappedSuperclass 关联的预定义查询，这些查询：

- 使用 JPA 查询语言（请参阅 JSR-000220 Enterprise JavaBeans v3.0 规范，第 4 章）进行基于任何基础数据库的可移植执行
- 经常被使用
- 比较复杂并且难于创建
- 可以在不同实体之间共享
- 只返回实体（从不返回标量值），并只返回一个类型的实体

如果有多个要定义的 @NamedQuery，则必须使用 @NamedQueries。

要在已知的基础数据库中预定义原生 SQL 查询，请参阅 @NamedNativeQuery。使用原生 SQL 查询，您可以返回实体（包括不同类型的实体）、标量值或同时返回两者。

表 1-6 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-28 @NamedQuery 属性

属性	必需	说明

query	✔	要指定查询，请将 query 设置为 JPA 查询语言（作为 String）。 有关 JPA 查询语言的详细信息，请参阅 JSR-000220 Enterprise JavaBeans v.3.0 规范的第 4 章。
hints		默认值：空 QueryHint 数组。 默认情况下，JPA 持续性提供程序假设 SQL 查询应完全按照 query 属性提供的方式执行，而不管基础数据库如何。 如果您知道基础数据库在运行时的状态，则要微调查询的执行，可以选择将 hints 设置为 QueryHint 数组（请参阅 @QueryHint）。在执行时，EntityManager 将向基础数据库传递提示。
name	✔	要指定查询名称，请将 name 设置为查询名称（作为 String）。 这是您在运行时调用查询所使用的名称（请参阅示例 1-60）。

示例 1-59 显示了如何使用 @NamedQuery 批注定义一个JPA 查询语言查询，该查询使用名为 firstname 的参数。示例 1-60 显示了如何使用 EntityManager 获取此查询并使用 Query 方法 setParameter 设置 firstname 参数。

示例 1-59 使用 @NamedQuery 实现一个带参数的查询

```
@Entity
@NamedQuery(
name="findAllEmployeesByFirstName",
query="SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
)
public class Employee implements Serializable {
    ...
}
```

示例 1-60 执行命名查询

```
Query queryEmployeesByFirstName = em.createNamedQuery("findAllEmployeesByFirstName");
queryEmployeeByFirstName.setParameter("firstName", "John");
Collection employees = queryEmployessByFirstName.getResultList();
```

@OneToMany

默认情况下，JPA 为具有一对多多重性的多值关联定义一个 OneToMany 映射。

使用 @OneToMany 批注：

- 将获取类型配置为 LAZY
- 由于所使用的 Collection 不是使用一般参数定义的，因此配置关联的目标实体
- 配置必须层叠到关联目标的操作：例如，如果删除了拥有实体，则确保还删除关联的目标
- 配置持续性提供程序对单向一对多关系使用的连接表（请参阅 @JoinTable）的详细信息

表 1-29 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-29 @OneToMany 属性

属性	必需	说明
cascade		默认值：CascadeType 的空数组。 默认情况下，JPA 不会将任何持续性操作层叠到关联的目标。 如果希望某些或所有持续性操作层叠到关联的目标，请将 cascade 设置为一个或多个 CascadeType 实例，其中包括： • ALL - 针对拥有实体执行的任何持续性操作均层叠到关联的目标。 • MERGE - 如果合并了拥有实体，则将 merge 层叠到关联的目标。 • PERSIST - 如果持久保存拥有实体，则将 persist 层叠到关联的目标。 • REFRESH - 如果刷新了拥有实体，则 refresh 为关联的层叠目标。 • REMOVE - 如果删除了拥有实体，则还删除关联的目标。
fetch		默认值：FetchType.EAGER。 默认情况下，JPA 持续性提供程序使用获取类型 EAGER：它要求持续性提供程序运行时必须急性获取数据。 如果这不适合于应用程序或特定的持久字段，请将 fetch 设置为 FetchType.LAZY：它提示持续性提供程序在首次访问数据（如果可以）时应惰性获取数据。
mappedBy		默认值：如果关系是单向的，则该持续性提供程序确定拥有该关系的字段。 如果关系是双向的，则将关联相反（非拥有）方上的 mappedBy 元素设置为拥有此关系的字段或属性的名称（如示例 1-62 所示）。
targetEntity		默认值：使用一般参数定义的 Collection 的参数化类型。 默认情况下，如果使用通过一般参数定义的 Collection，则持续性提供程序从被引用的对象类型推断出关联的目标实体。 如果 Collection 不使用一般参数，则必须指定作为关联目标的实体类：将关联拥有方上的 targetEntity 元素设置为作为关系目标的实体的 Class。

示例 1-61 和示例 1-62 显示了如何使用此批注在使用一般参数的 Customer（被拥有方）和 Order（拥有方）之间配置一个一对多映射。

示例 1-61 @OneToMany - 使用一般参数的 Customer 类

```
@Entity
public class Customer implements Serializable {
    ...
@OneToMany(cascade=ALL, mappedBy="customer")
public Set<Order> getOrders() {
return orders;
    }
    ...
```

```
}
```

示例 1-62 @ManyToOne - 使用一般参数的 Order 类

```
@Entity
public class Customer implements Serializable {
    ...
@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
public Customer getCustomer() {
return customer;
    }
    ...
}
```

@OneToOne

默认情况下，JPA 为指向另一个具有一对一多重性的实体的单值关联定义一个 `OneToOne` 映射，并从被引用的对象类型推断出关联的目标实体。

使用 `@OneToOne` 批注：

- 将获取类型配置为 `LAZY`
- 如果空值不适合于应用程序，则将映射配置为禁止空值（针对非基元类型）
- 配置关联的目标实体（如果无法从被引用的对象类型推断出它）
- 配置必须层叠到关联目标的操作：例如，如果删除了拥有实体，则确保还删除关联的目标

表 1-30 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-30 @OneToOne 属性

属性	必需	说明
`cascade`		默认值：空 `CascadeType` 数组。 默认情况下，JPA 不会将任何持续性操作层叠到关联的目标。 如果希望某些或所有持续性操作层叠到关联的目标，请将 `cascade` 设置为一个或多个 `CascadeType` 实例，其中包括： • `ALL` - 针对拥有实体执行的任何持续性操作均层叠到关联的目标。 • `MERGE` - 如果合并了拥有实体，则将 merge 层叠到关联的目标。 • `PERSIST` - 如果持久保存拥有实体，则将 persist 层叠到关联的目标。 • `REFRESH` - 如果刷新了拥有实体，则 refresh 为关联的层叠目标。 • `REMOVE` - 如果删除了拥有实体，则还删除关联的目标。

fetch		默认值：FetchType.EAGER。 默认情况下，JPA 持续性提供程序使用获取类型 EAGER：它要求持续性提供程序运行时必须急性获取数据。 如果这不适合于应用程序或特定的持久字段，请将 fetch 设置为 FetchType.LAZY：它提示持续性提供程序在首次访问数据（如果可以）时应惰性获取数据。
mappedBy		默认值：JPA 持续性提供程序从被引用的对象类型推断出关联的目标实体 如果持续性提供程序无法推断关联的目标实体，则将关联的相反（非拥有）方上的 mappedBy 元素设置为拥有此关系的字段或属性的 String 名称（如示例 1-64）所示。
optional		默认值：true。 默认情况下，JPA 持续性提供程序假设所有（非基元）字段和属性的值可以为空。 如果这并不适合于您的应用程序，请将 optional 设置为 false。
targetEntity		默认值：JPA 持续性提供程序从被引用的对象类型推断出关联的目标实体 如果持续性提供程序无法推断出目标实体的类型，则将关联的拥有方上的 targetEntity 元素设置为作为关系目标的实体的 Class。

示例 1-63 和示例 1-64 显示了如何使用此批注在 Customer（拥有方）和 CustomerRecord（被拥有方）之间配置一个一对一映射。

示例 1-63 @OneToOne - Customer 类

```
@Entity
public class Customer implements Serializable {
    ...
@OneToOne(optional=false)
@JoinColumn(name="CUSTREC_ID", unique=true, nullable=false, updatable=false)
public CustomerRecord getCustomerRecord() {
return customerRecord;
    }
    ...
}
```

示例 1-64 @OneToOne - CustomerRecord 类

```
@Entity
public class CustomerRecord implements Serializable {
    ...
@OneToOne(optional=false, mappedBy="customerRecord")
public Customer getCustomer() {
return customer;
    }
    ...
}
```

@OrderBy

默认情况下，JPA 持续性提供程序按关联实体的主键以升序顺序检索 Collection 关联的成员。

将 @OrderBy 批注与 @OneToMany 和 @ManyToMany 一起使用以便：

- 指定一个或多个作为排序依据的其他字段或属性
- 为每个这样的字段或属性名指定不同的排序（升序或降序）

表 1-31 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-31 @OrderBy 属性

属性	必需	说明
`value`		默认值：JPA 持续性提供程序按关联实体的主键以升序顺序检索 `Collection` 关联的成员。 如果要按某些其他字段或属性排序并指定了不同的排序，则将 `value` 设置为以下元素的逗号分隔列表："`property-or-field-name ASC|DESC`"（请参阅示例 1-65）。

示例 1-65 显示了如何使用 `@OrderBy` 批注指定 Project 方法 `getEmployees` 应按 `Employee` 字段 `lastname` 以升序顺序并按 `Employee` 字段 `seniority` 以降序顺序返回 `Employee` 的 `List`。示例 1-66 显示了默认情况下，`Employee` 方法 `getProjects` 按 `Employee` 主键 `empId` 以升序顺序返回 `List`。

示例 1-65 Project 实体

```
@Entity public class Project {
    ...
@ManyToMany
@OrderBy("lastname ASC", "seniority DESC")
public List<Employee> getEmployees() {
    ...
    };
    ...
}
```

示例 1-66 Employee 实体

```
@Entity public class Employee {
@Id
private int empId;
    ...
private String lastname;
    ...
private int seniority;
    ...
@ManyToMany(mappedBy="employees")
// By default, returns a List in ascending order by empId
public List<Project> getProjects() {
    ...
    };
    ...
}
```

@PersistenceContext

在使用 JPA 持续性提供程序的应用程序中，可以使用实体管理器执行所有持续性操作（创建、读取、更新和删除）。Java EE 应用程序使用相关性注入或在 JNDI 名称空间中直接查找实体管理器获取实体管理器。

使用 `@PersistenceContext` 批注获取实体管理器：

- 使用相关性注入
- 使用 JNDI 查找按名称进行
- 与特定的持续性单元关联（另请参阅 @PersistenceUnit）
- 具有扩展的持续性上下文
- 使用特定的持续性属性进行了自定义（请参阅 @PersistenceProperty）

如果有多个要指定的 `@PersistenceContext`，则必须使用 @PersistenceContexts。

表 1-32 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-32 @PersistenceContext 属性

属性	必需	说明
`name`		默认值：JPA 持续性提供程序检索默认实体管理器。 如果要注入或查找特定实体管理器，请将 `name` 设置为要使用的实体管理器的 `String` 名称。 对于相关性注入，不需要 `name`。 对于 JNDI 查找，必须将 `name` 设置为要在环境引用上下文中访问实体管理器所使用的名称。
`properties`		默认值：JPA 持续性提供程序假设实体管理器将使用默认属性。 如果要配置包含供应商特定属性的 JPA 持续性提供程序属性（例如，请参阅“TopLink JPA Persistence.xml 文件扩展”），请将 `properties` 设置为 @PersistenceProperty 实例数组。
`type`		默认值：`PersistenceContextType.TRANSACTION`。 默认情况下，JPA 持续性提供程序假设实体管理器是容器管理的，并且它们的持续性上下文的生命周期伸缩到一个事务：即，持续性上下文在事务启动时开始存在，并在事务提交时停止存在。 在以下条件下，将 `type` 设置为 `PersistenceContextType.EXTENDED`： • 您的持续性上下文是应用程序管理的 • 您希望扩展的持续性上下文在 `EntityManager` 实例从创建一直到关闭期间存在 • 您希望实体管理器在事务提交后维护对实体对象的引用 • 您希望调用 `EntityManager` 方法 `persist`、`remove`、`merge` 和 `refresh`，而不论事务是否处于活动状态

unitName		默认值：JPA 持续性提供程序检索默认持续性单元的默认实体管理器。 如果要注入或查找与特定持续性单元关联的实体管理器，则将 unitName 设置为所需的 String 持续性单元名称。或者，如果要指定 EntityManagerFactory 和持续性单元，则可以使用 @PersistenceUnit。 对于相关性注入，不需要 unitName。 对于 JNDI 查找，如果指定 unitName，则由 name 访问的实体管理器必须与此持续性单元关联。

示例 1-67 显示了如何使用此批注在无状态会话中注入实体管理器，示例 1-68 显示了如何使用此批注在 JNDI 中查找实体管理器。

示例 1-67 使用 @PersistenceContext 和相关性注入

```
@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);
    }
}
```

示例 1-68 使用 @PersistenceContext 和 JNDI 查找

```
@Stateless
public class OrderEntryBean implements OrderEntry {
@PersistenceContext(name="OrderEM")
EntityManager em;
public void enterOrder(int custID, Order newOrder) {
em = (EntityManager)ctx.lookup("OrderEM");
Customer cust = em.find(Customer.class, custID);
cust.getOrders().add(newOrder);
newOrder.setCustomer(cust);
    }
}
```

@PersistenceContexts

如果要指定多个 @PersistenceContext，则必须使用一个 @PersistenceContexts 批注指定所有持续性上下文。

表 1-33 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-33 @PersistenceContexts 属性

属性	必需	说明
value	✔	要指定两个或更多持续性上下文，请将 value 设置为 PersistenceContext 实例数组（请参阅 @PersistenceContext）。

示例 1-69 显示了如何使用此批注指定两个持续性上下文。

示例 1-69 @PersistenceContexts

```
@Stateless
public class OrderEntryBean implements OrderEntry {
@PersistenceContexts({
@PersistenceContext(name="OrderEM")
@PersistenceContext(name="ItemEM"),
    })
public void enterOrder(int custID, Order newOrder) {
EntityManager em = (EntityManager)ctx.lookup("OrderEM");
Customer cust = em.find(Customer.class, custID);
cust.getOrders().add(newOrder);
newOrder.setCustomer(cust);
    }
public void enterItem(int orderID, Item newItem) {
EntityManager em = (EntityManager)ctx.lookup("ItemEM");
    ...
    }
}
```

@PersistenceProperty

默认情况下，JPA 持续性提供程序假设您使用 @PersistenceContext 获取的实体管理器将使用默认属性。

使用 `@PersistenceProperty` 批注指定属性（包括供应商特定的属性），以便容器或持续性提供程序：

- 自定义实体管理器行为
- 利用供应商的 JPA 持续性提供程序实现中的特定特性

创建实体管理器时将属性传递给持续性提供程序。无法识别的属性被简单地忽略。

表 1-34 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-34 @PersistenceProperty 属性

属性	必需	说明
`name`	✔	要指定持续性属性的名称，请将 `name` 设置为 `String` 属性名。 有关持续性属性的详细信息，请参阅 JPA 持续性提供程序文档。
`value`	✔	要指定持续性属性的值，请将 `value` 设置为所需的 `String` 属性值。 有关持续性属性值的详细信息，请参阅 JPA 持续性提供程序文档。

示例 1-70 显示了如何使用 `@PersistenceProperty` 批注自定义查询以利用由 TopLink Essentials 提供的供应商 JPA 扩展：在该示例中，该属性确保在此持续性上下文中使用一个完整的 TopLink 缓存。有关详细信息，请参阅"TopLink JPA Persistence.xml 文件扩展"。

示例 1-70 @PersistenceProperty

```
@Stateless
public class OrderEntryBean implements OrderEntry {
@PersistenceContext(
properties={
@PersistenceProperty={name="toplink.cache.type.default", value="CacheType.Full"}
```

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

@PersistenceUnit

默认情况下，JPA 持续性提供程序使用与默认持续性单元或您使用 @PersistenceContext 属性 unitName 指定的持续性单元关联的默认 EntityManagerFactory。

使用 @PersistenceUnit 批注指定EntityManagerFactory，您希望 JPA 持续性提供程序使用它来：

- 获取指定的实体管理器
- 指定 EntityManagerFactory 和持续性单元

如果有多个要指定的 @PersistenceUnit，则必须使用 @PersistenceUnits。

表 1-34 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-35 @PersistenceUnit 属性

属性	必需	说明
name		默认值：JPA 持续性提供程序从默认 EntityManagerFactory 中获取它的 EntityManager 实例。 如果希望 JPA 持续性提供程序在注入或查找实体管理器时使用特定的 EntityManagerFactory，请将 name 设置为所需的实体管理器工厂的 String 名称。 对于相关性注入，不需要 name。 对于 JNDI 查找，必须将 name 设置为要在环境引用上下文中访问实体管理器所使用的名称。
unitName		默认值：JPA 持续性提供程序检索默认持续性单元的默认实体管理器。 如果要注入或查找与特定持续性单元关联的实体管理器，则将 unitName 设置为所需的 String 持续性单元名称。另请参阅 @PersistenceContext。 对于相关性注入，不需要 unitName。 对于 JNDI 查找，如果指定 unitName，则由 name 访问的 EntityManagerFactory 必须与此持续性单元关联。

示例 1-71 显示了如何使用 @PersistenceUnit 批注指定要使用的 EntityManagerFactory 的 JNDI 名称以及与该工厂关联的持续性单元名称。当 JPA 持续性提供程序使用 JNDI 获取一个使用持续性上下文 OrderEM 的实体管理器时，它将使用 JNDI 名称 OrderEMFactory 与持续性单元 OrderEMUnit 关联的 EntityManagerFactory。

示例 1-71 使用 @PersistenceUnit 指定工厂和单元

```
@Stateless
public class OrderEntryBean implements OrderEntry {
@PersistenceContext(name="OrderEM")
@PersistenceUnit(name="OrderEMFactory", unitName="OrderEMUnit")
EntityManager em;
public void enterOrder(int custID, Order newOrder) {
em = (EntityManager)ctx.lookup("OrderEM");
Customer cust = em.find(Customer.class, custID);
cust.getOrders().add(newOrder);
newOrder.setCustomer(cust);
    }
}
```

示例 1-72 显示了一个使用 @PersistenceContext 属性 `unitName` 仅指定持续性单元的其他方法。在该示例中，当 JPA 持续性提供程序使用 JNDI 获取一个使用持续性上下文 `OrderEM` 的实体管理器时，它将使用与持续性单元 `OrderEMUnit` 关联的默认 `EntityManagerFactory`。

示例 1-72 使用 @PersistenceContext 属性 unitName

```
@Stateless
public class OrderEntryBean implements OrderEntry {
@PersistenceContext(name="OrderEM", unitName="OrderEMUnit")
EntityManager em;
public void enterOrder(int custID, Order newOrder) {
em = (EntityManager)ctx.lookup("OrderEM");
Customer cust = em.find(Customer.class, custID);
cust.getOrders().add(newOrder);
newOrder.setCustomer(cust);
    }
}
```

@PersistenceUnits

如果要指定多个 @PersistenceUnit，则必须使用一个 `@PersistenceUnits` 批注指定所有持续性上下文。

表 1-36 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-36 @PersistenceUnits 属性

属性	必需	说明
`value`	✔	要指定两个或更多持续性单元，请将 `value` 设置为 `PersistenceUnit` 实例数组（请参阅 @PersistenceUnit）。

示例 1-73 显示了如何使用此批注指定两个持续性单元。在该示例中，@PersistenceContext 属性 `unitName` 和 `@PersistenceUnit` 属性 `unitName` 必须对应以便关联持续性上下文和持续性单元。

示例 1-73 @PersistenceUnits

```
@Stateless
public class OrderEntryBean implements OrderEntry {
@PersistenceContexts({
@PersistenceContext(name="OrderEM", unitName="OrderEMUnit")
@PersistenceContext(name="ItemEM", unitName="ItemEMUnit"),
    })
@PersistenceUnits({
@PersistenceUnit(name="OrderEMFactory", unitName="OrderEMUnit"),
```

```
@PersistenceUnit(name="ItemEMFactory", unitName="ItemEMUnit")
    })
public void enterOrder(int custID, Order newOrder) {
EntityManager em = (EntityManager)ctx.lookup("OrderEM");
Customer cust = em.find(Customer.class, custID);
cust.getOrders().add(newOrder);
newOrder.setCustomer(cust);
    }
public void enterItem(int orderID, Item newItem) {
EntityManager em = (EntityManager)ctx.lookup("ItemEM");
    ...
    }
}
```

@PrimaryKeyJoinColumn

默认情况下，当一个实体使用 `InheritanceType.JOINED`（请参阅 @Inheritance）扩展另一个实体时，JPA 持续性提供程序假设子类的外键列与超类主表的主键列同名。

使用 `@PrimaryKeyJoinColumn` 批注：

- 如果子类的外键列与该情形中超类的主表的主键列不同名
- 使用 @SecondaryTable 批注将辅助表连接到主表
- 在 @OneToOne 映射中，引用实体的主键用作被引用实体的外键。
- 使用复合外键（请参阅 @PrimaryKeyJoinColumns）

表 1-37 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-37 @PrimaryKeyJoinColumn 属性

属性	必需	说明
`columnDefinition`		默认值：空 `String`。 默认情况下，JPA 使用最少量 SQL 创建一个数据库表列。 如果需要使用更多指定选项创建的列，请将 `columnDefinition` 设置为在生成列的 DDL 时希望 JPA 使用的 `String` SQL 片断。 不要将此属性与 @OneToOne 映射一起使用。

name		默认值：JPA 持续性提供程序对当前表的主键列采用以下名称之一（取决于您使用该批注的方式）： • InheritanceType.JOINED（请参阅 @Inheritance）：与超类的主键列同名。 • @SecondaryTable 映射：与主表的主键列同名。 • @OneToOne 映射：与引用实体的表的主键列同名。 如果该名称难于处理、是一个保留字、与事先存在的数据模型不兼容或作为数据库中的列名无效，则将 name 设置为所需的 String 列名。
referencedColumnName		默认值：JPA 持续性提供程序对连接到的表的主键列采用以下名称之一（取决于您使用该批注的方式）： • InheritanceType.JOINED（请参阅 @Inheritance）：与超类的主表的主键列同名。 • @SecondaryTable 映射：与主表的主键列同名。 • @OneToOne 映射：与被引用实体的表的主键列同名。 如果该名称难于处理、是一个保留字、与预先存在的数据模型不兼容或作为数据库中的列名无效，请将 referencedColumnName 设置为所需的 String 列名。

示例 1-74 显示了一个实体基类 Customer，示例 1-75 显示了如何使用 @PrimaryKeyJoinColumn 在 ValuedCustomer（Customer 的一个子类）的主表中指定主键连接列 CUST_ID。

示例 1-74 @PrimaryKeyJoinColumn - InheritanceType.JOINED 超类

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=JOINED)
@DiscriminatorValue("CUST")
public class Customer {
    ...
}
```

示例 1-75 @PrimaryKeyJoinColumn - InheritanceType.JOINED 子类

```
@Entity
@Table(name="VCUST")
@DiscriminatorValue("VCUST")
@PrimaryKeyJoinColumn(name="CUST_ID")
public class ValuedCustomer extends Customer {
    ...
}
```

@PrimaryKeyJoinColumns

默认情况下，JPA 持续性提供程序假设每个实体有一个单列主键。

如果要指定一个由两个或更多列组成的主键，请使用 @PrimaryKeyJoinColumns 批注。

表 1-38 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-38 @PrimaryKeyJoinColumns 属性

属性	必需	说明
value	✔	要指定一个复合（多列）主键，请将 value 设置为 PrimaryKeyJoinColumn 实例的数组（请参阅 @PrimaryKeyJoinColumn）。

示例 1-76 显示了如何使用此批注指定一个由列 CUST_ID 和 CUST_TYPE 组成的复合主键。

示例 1-76 @PrimaryKeyJoinColumns

```
@Entity
@Table(name="VCUST")
@DiscriminatorValue("VCUST")
@PrimaryKeyJoinColumns({
@PrimaryKeyJoinColumn(name="CUST_ID",referencedColumnName="ID"),
@PrimaryKeyJoinColumn(name="CUST_TYPE",referencedColumnName="TYPE")
})
public class ValuedCustomer extends Customer {
    ...
}
```

@QueryHint

默认情况下，JPA 持续性提供程序假设 @NamedQuery 或 @NamedNativeQuery 应完全按照查询 String 指定的方式执行。

使用 @QueryHint 批注指定供应商特定的 JPA 查询扩展，以：

- 提高查询性能
- 利用供应商的 JPA 持续性提供程序实现中的特定特性

表 1-6 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-39 @QueryHint 属性

属性	必需	说明
name	✔	要指定提示名称，请将 name 设置为 String 提示名称。 有关提示的详细信息，请参阅 JPA 持续性提供程序文档。
value	✔	要指定提示的值，请将 value 设置为所需的 String 提示值。 有关提示值的详细信息，请参阅 JPA 持续性提供程序文档。

示例 1-77 显示了如何使用 @QueryHint 批注自定义查询以利用由 TopLink Essentials 提供的供应商 JPA 扩展：在该示例中，提示确保在执行查询时始终刷新 TopLink 缓存。有关详细信息，请参阅“TopLink JPA 查询提示扩展”。

示例 1-77 @QueryHint

```
@Entity
@NamedQuery(
name="findAllEmployees",
query="SELECT * FROM EMPLOYEE WHERE MGR=1"
hints={@QueryHint={name="toplink.refresh", value="true"}}
)
public class Employee implements Serializable {
    ...
}
```

@SecondaryTable

默认情况下，JPA 持续性提供程序假设实体的所有持久字段均存储到一个名称为实体名称的数据库表中：该表称作主表（请参阅 @Table）。

如果希望 JPA 分别将实体的某些持久字段持久保存到主表和其他数据库表，请使用 `@SecondaryTable` 批注将实体与其他数据库表关联。在该示例中，您使用 @Column 批注将实体的持久字段与表关联。

如果要将两个或更多辅助表与实体关联，则可以使用 @SecondaryTables。

表 1-40 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-40 @SecondaryTable 属性

属性	必需	说明
姓名	✔	如果实体使用辅助表，请将 `name` 设置为 `String` 表名。
`catalog`		默认值：JPA 持续性提供程序使用任何适用于数据库的默认目录。 如果默认目录不适合于应用程序，请将 `catalog` 设置为要使用的 `String` 目录名。
`pkJoinColumns`		默认值：JPA 持续性提供程序假设实体的数据库表中的任何列均不用于主键连接。 如果对该表中的主键连接使用一个或多个列，请将 `pkJoinColumns` 设置为一个或多个 `@PrimaryKeyJoinColumn` 实例的数组。有关详细信息，请参阅 @PrimaryKeyJoinColumn。
`schema`		默认值：JPA 持续性提供程序使用任何适用于数据库的默认模式。 如果默认模式不适合于应用程序，请将 `schema` 设置为要使用的 `String` 模式名。
`uniqueConstraints`		默认值：JPA 持续性提供程序假设实体的数据库表中的任何列均没有唯一约束。 如果唯一约束应用于该表中的一列或多列，请将 `uniqueContraints` 设置为一个或多个 `UniqueConstraint` 实例的数组。有关详细信息，请参阅 @UniqueConstraint。

示例 1-78 显示了如何使用此批注指定一个名为 `EMP_HR` 的辅助表。在该示例中，默认情况下，JPA 将实体持久字段 `empId` 持久保存到名为 `Employee` 的主表中的列 `empId`，并将 `empSalary` 持久保存到辅助表 `EMP_HR` 中的列 `empSalary`。有关详细信息，请参阅 @Column。

示例 1-78 @SecondaryTable

```
@Entity
@SecondaryTable(name="EMP_HR")
public class Employee implements Serializable {
    ...
private Long empId;

@Column(table="EMP_HR", name="EMP_SALARY"))
private Float empSalary;
    ...
}
```

@SecondaryTables

如果需要指定多个 @SecondaryTable，可以使用一个 `@SecondaryTables` 批注指定所有辅助表。

表 1-41 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-41 @SecondaryTables 属性

属性	必需	说明
`value`	✔	要指定两个或更多辅助表，请将 `value` 设置为 `SecondaryTable` 实例的数组（请参阅 @SecondaryTable）。

示例 1-79 显示了如何使用此批注指定两个名为 `EMP_HR` 和 `EMP_TR` 的辅助表。在该示例中，默认情况下，JPA 将实体持久字段 `empId` 持久保存到名为 `Employee` 的主表中的列 `empId`。JPA 将 `empSalary` 持久保存到辅助表 `EMP_HR` 中的列 `empSalary`，并将 `empClass` 持久保存到辅助表 `EMP_TR` 中的列 `EMP_HR`。有关详细信息，请参阅 @Column。

示例 1-79 @SecondaryTables

```
@Entity
@SecondaryTables({
@SecondaryTable(name="EMP_HR"),
@SecondaryTable(name="EMP_TR")
    })
public class Employee implements Serializable {
    ...
private Long empId;

@Column(table="EMP_HR", name="EMP_SALARY"))
private Float empSalary;

@Column(table="EMP_TR", name="EMP_CLASS"))
private Float empClass;
    ...
}
```

@SequenceGenerator

如果使用 @GeneratedValue 批注指定一个 `SEQUENCE` 类型的主键生成器，则可以使用 `@SequenceGenerator` 批注微调该主键生成器以：

- 更改分配大小以匹配应用程序要求或数据库性能参数
- 更改初始值以匹配现有的数据模型（例如，如果基于已经为其分配或保留了一组主键值的现有数据集构建）

- 使用现有数据模型中预定义的序列

表 1-42 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-42 @SequenceGenerator 属性

属性	必需	说明
`name`	✔	`SequenceGenerator` 的名称必须匹配其 `startegy` 设置为 `SEQUENCE` 的 `GeneratedValue` 的名称。
`allocationSize`		默认值： 50. 默认情况下，JPA 持续性提供程序使用的分配大小为 50。 如果此分配大小与应用程序要求或数据库性能参数不匹配，请将 `allocationSize` 设置为所需的 `int` 值。
`initialValue`		默认值： 0. 默认情况下，JPA 持续性提供程序假设持续性提供程序将所有主键值的起始值设置为 0。 如果这与现有数据模型不匹配，请将 `initialValue` 设置为所需的 `int` 值。
`sequenceName`		默认值：JPA 持续性提供程序分配它自己创建的序列名。 如果要使用事先存在或预定义的序列，请将 `sequenceName` 设置为所需的 `String` 名称。

示例 1-80 显示了如何使用此批注为名为 `CUST_SEQ` 的 `SEQUENCE` 主键生成器指定分配大小。

示例 1-80 @SequenceGenerator

```
@Entity
public class Employee implements Serializable {
    ...
@Id
@SequenceGenerator(name="CUST_SEQ", allocationSize=25)
@GeneratedValue(strategy=SEQUENCE, generator="CUST_SEQ")
@Column(name="CUST_ID")
public Long getId() {
return id;
    }
    ...
```

@SqlResultSetMapping

执行 @NamedNativeQuery 时，它可以返回实体（包括不同类型的实体）、标量值或实体和标量值的组合。

默认情况下（如示例 1-81 所示），JPA 持续性提供程序假设原生 SQL 查询中的 `SELECT` 语句：

- 返回一个实体类型
- 包含与返回的实体的所有字段或属性相对应的所有列

- 使用与字段或属性名（未使用 AS 语句）相对应的列名

示例 1-81 简单的原生 SQL 查询

```
Query q = entityManager.createNativeQuery(
"SELECT o.id, o.quantity, o.item " +
"FROM Order o, Item i " +
"WHERE (o.item = i.id) AND (i.name = "widget")",
Order.class
);
List resultList = q.getResultList();
// List of Order entity objects:{Order, Order, ...}
```

如果原生 SQL 查询满足以下条件，请使用 `@SqlResultSetMapping` 批注控制 JPA 持续性提供程序如何将 JDBC 结果集映射到实体字段或属性以及标量：

- 返回多个类型的实体
- 只返回标量值或实体和标量值的组合
- 使用列别名（AS 语句）

如果有多个 `@SqlResultSetMapping`，则必须使用 @SqlResultSetMappings。

表 1-8 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-43 @SqlResultSetMapping 属性

属性	必需	说明
`name`	✔	将 `name` 设置为此 `@SqlResultSetMapping` 的 `String` 名称。 这是用于将 `@SqlResultSetMapping` 与原生 SQL 查询关联的名称（请参阅示例 1-84）。
`columns`		默认值：空 `ColumnResult` 数组。 默认情况下，JPA 持续性提供程序假设 `SELECT` 语句只返回实体。 如果 `SELECT` 语句返回标量值，则将 `columns` 设置为 `ColumnResult` 实例的数组，每个标量结果一个 @ColumnResult。
`entities`		默认值：空 `EntityResult` 数组。 默认情况下，JPA 持续性提供程序假设 SELECT 语句返回一个类型的实体。 如果 SELECT 语句返回多个类型的实体，请将实体设置为 `EntityResult` 实例的数组，每个返回的实体类型一个 @EntityResult。

示例 1-82 显示了如何使用此批注将 `Order` 和 `Item`（请参阅示例 1-83）实体和标量 `name` 包含在结果列表（请参阅示例 1-84）中。在该示例中，结果列表将为 `Object` 数组的 `List`，如：`{[Order, Item, "Shoes"], [Order, Item, "Socks"], ...}`。

示例 1-82 使用 @SqlResultSetMapping 的 Order 实体

```
@SqlResultSetMapping(
name="OrderResults",
entities={
@EntityResult(
entityClass=Order.class,
fields={
@FieldResult(name="id",       column="order_id"),
@FieldResult(name="quantity", column="order_quantity"),
@FieldResult(name="item",     column="order_item")
            }
        ),
@EntityResult(
entityClass=Item.class,
fields={
@FieldResult(name="id",       column="item_id"),
@FieldResult(name="name",     column="item_name"),
            }
        )
    }
columns={
@ColumnResult(
name="item_name"
        )
    }
)
@Entity
public class Order {
@Id
protected int id;
protected long quantity;
protected Item item;
    ...
}
```

示例 1-83 Item 实体

```
@Entity
public class Item {
@Id
protected int id;
protected String name;
    ...
}
```

示例 1-84 将 @SqlResultSetMapping 与 @EntityResult 一起使用的原生查询

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

List resultList = q.getResultList();
// List of Object arrays:{[Order, Item, "Shoes"], [Order, Item, "Socks"], ...}
```

@SqlResultSetMappings

如果需要指定多个 @SqlResultSetMapping**，则必须使用一个** `@SqlResultSetMappings` **批注指定所有** SQL **结果集映**

射。

表 1-5 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-44 @SqlResultSetMappings 属性

属性	必需	说明
value	✔	要指定两个或更多 SQL 结果集映射，请将 value 设置为 @SqlResultSetMapping 实例的数组。

示例 1-85 显示了如何使用此批注指定两个 @SqlResultSetMapping 实例。

示例 1-85 @SqlResultSetMappings

```
SqlResultSetMappings({
@SqlResultSetMapping(
name="OrderItemItemNameResults",
entities={
@EntityResult(entityClass=Order.class),
@EntityResult(entityClass=Item.class)
        }
columns={
@ColumnResult(name="item_name")
        }
    ),
@SqlResultSetMapping(
name="OrderItemResults",
entities={
@EntityResult(entityClass=Order.class),
@EntityResult(entityClass=Item.class)
        }
    )
})
@Entity
public class Order {
@Id
protected int id;
protected long quantity;
protected Item item;
    ...
}
```

@Table

默认情况下，JPA 持续性提供程序假设实体的所有持久字段均存储到一个名称为实体名称的数据库表中（请参阅 @Entity）。

在以下条件下，使用 @Table 批注指定与实体关联的主表：

- 实体名称难于处理、是一个保留字、与预先存在的数据模型不兼容或作为数据库中的表名无效
- 需要控制表所属的目录或模式

如果希望 JPA 将某些字段持久保存到主表，而将其他字段持久保存到一个或多个辅助表，请参阅 @SecondaryTable。

表 1-45 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-45 @Table 属性

属性	必需	说明
`catalog`		默认值：JPA 持续性提供程序使用任何适用于数据库的默认目录。 如果默认目录不适合于应用程序，请将 `catalog` 设置为要使用的 `String` 目录名。
`name`		默认值：JPA 持续性提供程序假设实体的数据库表与实体类同名。在示例 1-86 中，默认 `name` 为 `Employee`。 如果实体类名难以处理、是一个保留字或与预先存在的数据模型不兼容，请将 `name` 设置为相应的数据表名称。在示例 1-86 中，JPA 将实体类 `Employee` 持久保存到名为 `EMP` 的数据库表中。
`schema`		默认值：JPA 持续性提供程序使用任何适用于数据库的默认模式。 如果默认模式不适合于应用程序，请将 `schema` 设置为要使用的 `String` 模式名。
`uniqueConstraints`		默认值：JPA 持续性提供程序假设实体的数据库表中的任何列均没有唯一约束。 如果唯一约束应用于该表中的一列或多列，请将 uniqueContraints 设置为一个或多个 `UniqueConstraint` 实例的数组。有关详细信息，请参阅 @UniqueConstraint。

示例 1-86 显示了如何使用此批注指定主表名。

示例 1-86 @Table

```
@Entity
@Table(name="EMP")
public class Employee implements Serializable {
    ...
}
```

@TableGenerator

如果使用 @GeneratedValue 批注指定一个 `TABLE` 类型的主键生成器，可以使用 `@TableGenerator` 批注微调该主键生成器以：

- 由于名称难于处理、是一个保留字、与预先存在的数据模型不兼容或作为数据库中的表名无效而更改主键生成器的表名称
- 更改分配大小以匹配应用程序要求或数据库性能参数
- 更改初始值以匹配现有的数据模型（例如，如果基于已经为其分配或保留了一组主键值的现有数据集构建）
- 使用特定目录或模式配置主键生成器的表
- 在主键生成器表的一列或多列商配置一个唯一的约束

表 1-46 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-46 @TableGenerator 属性

属性	必需	说明
name	✔	SequenceGenerator 的名称必须匹配其 startegy 设置为 startegy 的 GeneratedValue 的名称。生成器名称的作用域对持续性单元是全局的（跨所有生成器类型）。
allocationSize		默认值： 50. 默认情况下，JPA 持续性提供程序使用的分配大小为 50。 如果此分配大小与应用程序要求或数据库性能参数不匹配，请将 allocationSize 设置为所需的 int 值。
catalog		默认值：JPA 持续性提供程序使用任何适用于数据库的默认目录。 如果默认目录不适合于应用程序，请将 catalog 设置为要使用的 String 目录名。
initialValue		默认值： 0. 默认情况下，JPA 持续性提供程序将所有主键值的起始值设置为 0。 如果这与现有数据模型不匹配，请将 initialValue 设置为所需的 int 值。
pkColumnName		默认值：JPA 持续性提供程序为生成器表中的主键列提供名称。 如果该名称不适合于应用程序，请将 pkColumnName 设置为所需的 String 名称。
pkColumnValue		默认值：JPA 持续性提供程序为生成器表中的主键列提供一个合适的主键值。 如果该值不适合于应用程序，请将 pkColumnValue 设置为所需的 String 值。
schema		默认值：JPA 持续性提供程序使用任何适用于数据库的默认模式。 如果默认模式不适合于应用程序，请将 schema 设置为要使用的 String 模式名。
table		默认值：JPA 持续性提供程序为存储生成的 ID 值的表提供了一个合适的名称。 如果默认表名不适合于应用程序，请将 table 设置为所需的 String 表名。
uniqueConstraints		默认值：JPA 持续性提供程序假设主键生成器表中的任何列均没有唯一约束。 如果唯一约束应用于该表中的一列或多列，则将 uniqueContraints 设置为一个或多个 UniqueConstraint 实例的数组。有关详细信息，请参阅 @UniqueConstraint。
valueColumnName		默认值：JPA 持续性提供程序为存储生成的 ID 值的列提供了一个合适的名称。 如果默认列名不适合于应用程序，请将 valueColumnName 设置为所需的 String 列名。

示例 1-87 显示了如何使用此批注为名为 empGen 的 TABLE 主键生成器指定分配大小。

示例 1-87 @TableGenerator

```
@Entity
public class Employee implements Serializable {
    ...
@Id
@TableGenerator(
name="empGen",
```

```
allocationSize=1
    )
@GeneratedValue(strategy=TABLE, generator="empGen")
@Column(name="CUST_ID")
public Long getId() {
return id;
    }
    ...
```

@Temporal

使用 `@Temporal` 批注指定 JPA 持续性提供程序应只为 `java.util.Date` 和 `java.util.Calendar` 类型的字段或属性持久保存的数据库类型。

该批注可以与 @Basic 一起使用。

表 1-14 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-47 @Temporal 属性

属性	必需	说明
`value`	✔	将 `value` 设置为与希望 JPA 持续性提供程序使用的数据库类型相对应的 `TemporalType`： • `DATE` - 等于 `java.sql.Date` • `TIME` - 等于 `java.sql.Time` • `TIMESTAMP` - 等于 `java.sql.Timestamp`

示例 1-88 显示了如何使用此批注指定 JPA 持续性提供程序应将 `java.util.Date` 字段 `startDate` 持久保存为 `DATE` (`java.sql.Date`) 数据库类型。

示例 1-88 @Temporal

```
@Entity
public class Employee {
    ...
@Temporal(DATE)    protected java.util.Date startDate;
    ...
}
```

@Transient

默认情况下，JPA 持续性提供程序假设实体的所有字段均为持久字段。

使用 `@Transient` 批注指定实体的非持久字段或属性，例如，一个在运行时使用但并非实体状态一部分的字段或属性。

JPA 持续性提供程序不会对批注为 `@Transient` 的属性或字段持久保存（或创建数据库模式）。

该批注可以与 @Entity、@MappedSuperclass 和 @Embeddable 一起使用。

该批注没有属性。有关更多详细信息，请参阅 API。

示例 1-89 显示了如何使用此批注将 Employee 字段 currentSession 指定为非持久字段。JPA 持续性提供程序将不持久保存该字段。

示例 1-89 @Transient

```
@Entitypublic class Employee {    @Id int id;    @Transient Session currentSession;    ...}
```

@UniqueConstraint

默认情况下，JPA 持续性提供程序假设所有列均可以包含重复值。

使用 @UniqueConstraint 批注指定将在为主表或辅助表生成的 DDL 中包含一个唯一约束。或者，您可以在列级别指定唯一约束（请参阅 @Column）。

表 1-48 列出了此批注的属性。有关更多详细信息，请参阅 API。

表 1-48 @UniqueConstraint 属性

属性	必需	说明
columnNames	✔	如果任何列均包含唯一约束，请将 columnNames 设置为 String 列名的数组。

示例 1-90 显示了如何使用此批注对主表 EMP 中的列 EMP_ID 和 EMP_NAME 指定一个唯一约束。

示例 1-90 使用唯一约束的 @Table

```
@Entity
@Table(
name="EMP",
uniqueConstraints={@UniqueConstraint(columnNames={"EMP_ID", "EMP_NAME"})}
)
public class Employee implements Serializable {
    ...
}
```

@Version

默认情况下，JPA 持续性提供程序假设应用程序负责数据一致性。

使用 @Version 批注通过指定用作其乐观锁定值的实体类的版本字段或属性来启用 JPA 管理的乐观锁定（推荐做法）。

选择版本字段或属性时，确保：

- 每个实体只有一个版本字段或属性
- 选择一个持久保存到主表的属性或字段（请参阅 @Table）
- 您的应用程序不修改版本属性或字段

此批注没有属性。有关更多详细信息，请参阅 API。

示例 1-91 显示了如何使用此批注将属性 getVersionNum 指定为乐观锁定值。在该示例中，该属性的列名设置为 OPTLOCK（请参阅 @Column），而非属性的默认列名。

示例 1-91 @Version

```
@Entity
public class Employee implements Serializable {
    ...
@Version
@Column(name="OPTLOCK")
protected int getVersionNum() {
return versionNum;
    }
    ...
}
```

生命周期事件批注

如果需要在生命周期事件期间执行自定义逻辑，请使用以下生命周期事件批注关联生命周期事件与回调方法：

- @PostLoad
- @PostPersist
- @PostRemove
- @PostUpdate
- @PrePersist
- @PreRemove
- @PreUpdate

图 1-1 演示了 JPA 支持的实体生命周期事件之间的关系。

可以直接对实体方法进行批注，也可以指定一个或多个实体监听程序类（请参阅 @EntityListeners）。

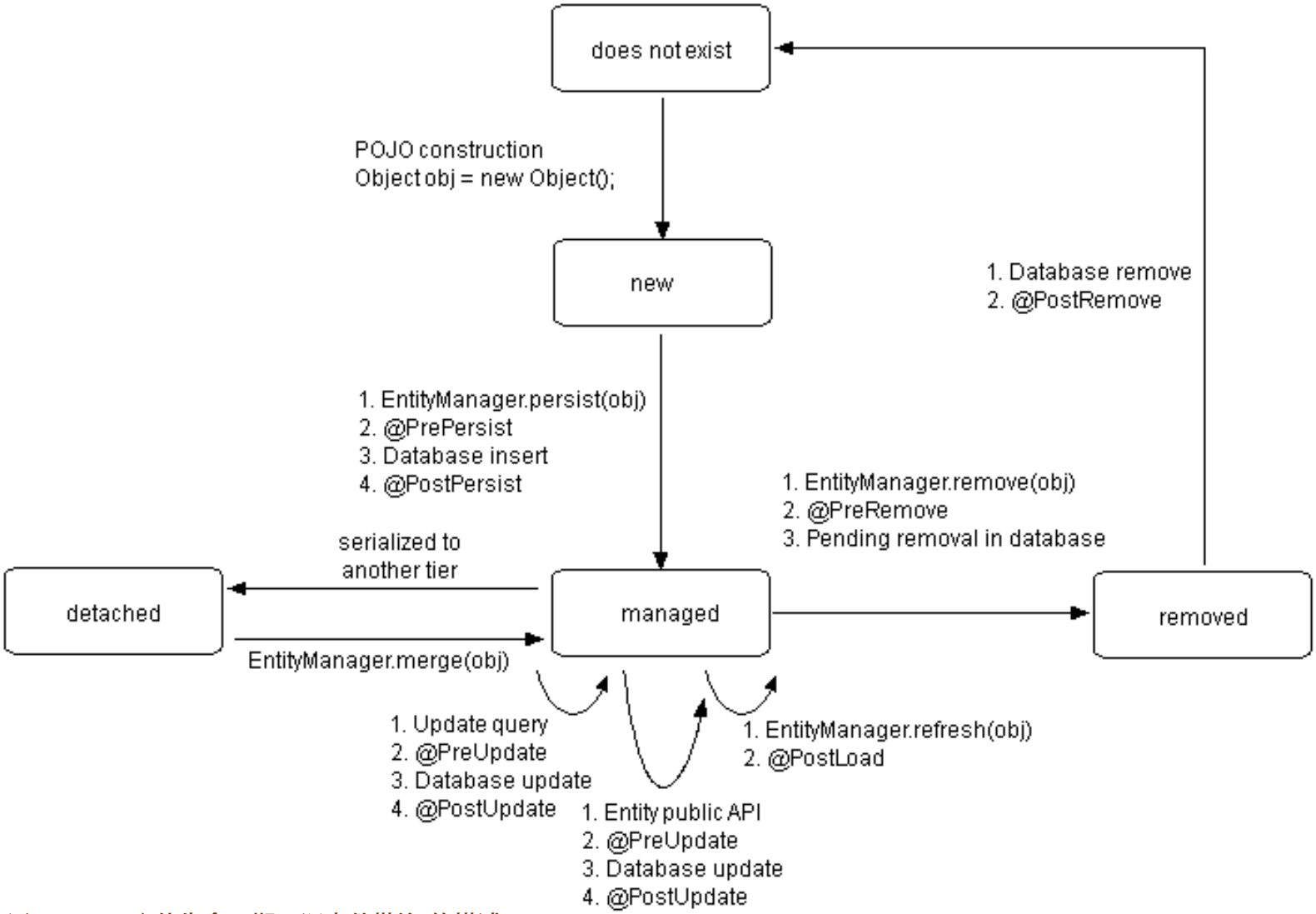
如果直接对实体方法进行批注，则该实体方法必须满足以下要求：

- 实体类方法必须具有以下签名：

  ```
  public int <MethodName>()
  ```

- 实体类方法可以有任何方法名称，只要它不以 `ejb` 开头。

*图 **1-1 JPA** 实体生命周期回调事件批注*



"图 1-1 JPA 实体生命周期回调事件批注"的描述"

@PostLoad

将实体加载到数据库的当前持续性上下文中后或在向其应用了刷新操作后，调用实体的 `@PostLoad` 方法。在返回或访问查询结果之前或在遍历关联之前调用该方法。

如果要在实体生命周期中的该点调用自定义逻辑，请使用 `@PostLoad` 批注。

此批注没有属性。有关更多详细信息，请参阅 API。

@PostPersist

在实体成为持久实体后，调用该实体的 `@PostPersist` 回调方法。对该操作层叠到的所有实体调用该方法。在数据库插入操作之后调用该方法。这些数据库操作可能在调用了持久操作之后立即发生，也可能在刷新操作（可能在事务结束时发生）发生之后立即发生。PostPersist 方法中提供了生成的主键值。

使用 `@PostPersist` 批注通知任何相关对象或更新直到插入对象时才可以访问的信息。

此批注没有属性。有关更多详细信息，请参阅 API。

@PostRemove

在删除实体后，调用该实体的 @PostRemove 回调方法。对该操作层叠到的所有实体调用该方法。在数据库删除操作之后调用该方法。这些数据库操作可能在调用了删除操作之后立即发生，也可能在刷新操作（可能在事务结束时发生）发生之后立即发生。

使用 @PostRemove 批注通知任何相关对象。

此批注没有属性。有关更多详细信息，请参阅 API。

@PostUpdate

在对实体数据进行了数据库更新操作后，调用实体的 @PostUpdate 回调方法。这些数据库操作可以在更新实体状态时发生，也可以在将状态刷新到数据库（位于事务结尾）时发生。注意，究竟此回调是在持久保存实体并随后在单个事务中修改实体时发生还是在修改了实体并随后在单个事务中删除实体时发生与实现相关。可移植应用程序不应依赖此行为。

如果要在实体生命周期的该点调用自定义逻辑，请使用 @PostUpdate 批注。

此批注没有属性。有关更多详细信息，请参阅 API。

@PrePersist

在执行给定实体的相应 EntityManager 持久操作之前，调用该实体的 @PrePersist 回调方法。对于向其应用了合并操作并导致创建新管理的实例的实体而言，在向其复制了实体状态后对管理的实例调用该方法。对该操作层叠到的所有实体调用该方法。

如果要在实体生命周期期间的该点调用自定义逻辑，请使用 @PrePersist 批注。

此批注没有属性。有关更多详细信息，请参阅 API。

@PreRemove

在针对给定实体执行相应的 EntityManager 删除操作之前，调用该给定实体的 @PreRemove 回调方法。对该操作层叠到的所有实体调用该方法。

如果要在实体生命周期中的该点调用自定义逻辑，请使用 @PreRemove 批注。

此批注没有属性。有关更多详细信息，请参阅 API。

@PreUpdate

在对实体数据进行数据库更新操作之前，调用实体的 @PreUpdate 回调方法。这些数据库操作可以在更新实体状态时发生，也可以在将状态刷新到数据库（可能位于事务结尾）时发生。请注意：此回调是否在持久保存实体并随后在单个事务中修改该实体时发生，均依赖于实施。可移植应用程序不应依赖此行为。

如果要在实体生命周期的该点调用自定义逻辑，请使用 @PreUpdate 批注。

此批注没有属性。有关更多详细信息，请参阅 API。

批注索引

- A

- @AssociationOverride
- @AssociationOverrides
- @AttributeOverride
- @AttributeOverrides

- **B**
  - @Basic
- **C**
  - @Column
  - @ColumnResult
- **D**
  - @DiscriminatorColumn
  - @DiscriminatorValue
- **E**
  - @Embeddable
  - @Embedded
  - @EmbeddedId
  - @Entity
  - @EntityListeners
  - @EntityResult
  - @Enumerated
  - @ExcludeDefaultListeners
  - @ExcludeSuperclassListeners
- **F**
  - @FieldResult
- **G**
  - @GeneratedValue
- **I**
  - @Id
  - @IdClass
  - @Inheritance
- **J**

- @JoinColumn
- @JoinColumns
- @JoinTable

- **L**
  - @Lob
- **M**
  - @ManyToMany
  - @ManyToOne
  - @MapKey
  - @MappedSuperclass
- **N**
  - @NamedNativeQueries
  - @NamedNativeQuery
  - @NamedQueries
  - @NamedQuery
- **O**
  - @OneToMany
  - @OneToOne
  - @OrderBy
- **P**
  - @PersistenceContext
  - @PersistenceContexts
  - @PersistenceProperty
  - @PersistenceUnit
  - @PersistenceUnits
  - @PrimaryKeyJoinColumn
  - @PrimaryKeyJoinColumns
- **Q**
  - @QueryHint
- **S**

- @SecondaryTable
- @SecondaryTables
- @SequenceGenerator
- @SqlResultSetMapping
- @SqlResultSetMappings

- **T**
  - @Table
  - @TableGenerator
  - @Temporal
  - @Transient
- **U**
  - @UniqueConstraint
- **V**
  - @Version
