* hibernate 注释说明：

* @Entity —— 将一个类声明为一个实体 bean(即一个持久化 POJO 类)

* @Id —— 注解声明了该实体 bean 的标识属性（对应表中的主键）。

* @Table —— 注解声明了该实体 bean 映射指定的表（table）,目录（catalog）和 schema 的名字

* @Column —— 注解声明了属性到列的映射。该注解有如下的属性

* name 可选，列名（默认值是属性名）

* unique 可选，是否在该列上设置唯一约束（默认值 false）

* nullable 可选，是否设置该列的值可以为空（默认值 false）

* insertable 可选，该列是否作为生成的 insert 语句中的一个列（默认值 true）

* updatable 可选，该列是否作为生成的 update 语句中的一个列（默认值 true）

* columnDefinition 可选，为这个特定列覆盖 sql ddl 片段（这可能导致无法在不同数据库间移植）

* table 可选，定义对应的表（默认为主表）

* length 可选，列长度（默认值 255）

* precision 可选，列十进制精度（decimal precision)(默认值 0）

* scale 可选，如果列十进制数值范围（decimal scale）可用，在此设置（默认值 0）

* @GeneratedValue —— 注解声明了主键的生成策略。该注解有如下属性

* strategy 指定生成的策略（JPA 定义的），这是一个 GenerationType。默认是 GenerationType. AUTO

* GenerationType.AUTO 主键由程序控制

* GenerationType.TABLE 使用一个特定的数据库表格来保存主键

* GenerationType.IDENTITY 主键由数据库自动生成（主要是自动增长类型）

* GenerationType.SEQUENCE 根据底层数据库的序列来生成主键，条件是数据库支持序列。（这个值要与 generator 一起使用）

* generator 指定生成主键使用的生成器（可能是 orcale 中的序列）。

* @SequenceGenerator —— 注解声明了一个数据库序列。该注解有如下属性

* name 表示该表主键生成策略名称，它被引用在@GeneratedValue 中设置的“gernerator”值中

* sequenceName 表示生成策略用到的数据库序列名称。

* initialValue 表示主键初始值，默认为 0.

* allocationSize 每次主键值增加的大小，例如设置成 1，则表示每次创建新记录后自动加 1，默认为 50.

* @GenericGenerator —— 注解声明了一个 hibernate 的主键生成策略。支持十三种策略。该注解有如下属性

* name 指定生成器名称

* strategy 指定具体生成器的类名（指定生成策略）。

* parameters 得到 strategy 指定的具体生成器所用到的参数。

* 其十三种策略（strategy 属性的值）如下：

* 1.native 对于 orcale 采用 Sequence 方式，对于 MySQL 和 SQL Server 采用 identity(处境主键生成机制)，

* native 就是将主键的生成工作将由数据库完成，hibernate 不管（很常用）

* 例：
@GeneratedValue(generator = "paymentableGenerator")

* @GenericGenerator(name = "paymentableGenerator", strategy = "native")

* 2.uuid 采用 128 位的 uuid 算法生成主键，uuid 被编码为一个 32 位 16 进制数字的字符串。占用空间大（字符串类型）。

* 例：
@GeneratedValue(generator = "paymentableGenerator")

* @GenericGenerator(name = "paymentableGenerator", strategy = "uuid")

* 3.hilo 要在数据库中建立一张额外的表，默认表名为 hibernate_unque_key，默认字段为 integer 类型，名称是 next_hi（比较少用）

* 例：
@GeneratedValue(generator = "paymentableGenerator")

* @GenericGenerator(name = "paymentableGenerator", strategy = "hilo")

* 4.assigned 在插入数据的时候主键由程序处理（很常用），这是<generator>元素没有指定时的默认生成策略。等同于 JPA 中的 AUTO。

* 例：
@GeneratedValue(generator = "paymentableGenerator")

* @GenericGenerator(name = "paymentableGenerator", strategy = "assigned")

* 5.identity 使用 SQL Server 和 MySQL 的自增字段，这个方法不能放到 Oracle 中，Oracle 不支持自增字段，要设定 sequence(MySQL 和 SQL Server 中很常用)。等同于 JPA 中的 IDENTITY

*　　例：
@GeneratedValue(generator = "paymentableGenerator")

*　　@GenericGenerator(name = "paymentableGenerator", strategy = "identity")

*　　6.select 使用触发器生成主键（主要用于早期的数据库主键生成机制，少用）

*　　例：
@GeneratedValue(generator = "paymentableGenerator")

*　　@GenericGenerator(name = "paymentableGenerator", strategy = "select")

*　　7.sequence 调用谨慎数据库的序列来生成主键，要设定序列名，不然 hibernate 无法找到。

*　　例：
@GeneratedValue(generator = "paymentableGenerator")

*　　@GenericGenerator(name = "paymentableGenerator", strategy = "sequence",

*　　parameters = { @Parameter(name = "sequence", value = "seq_payablemoney") })

*　　8.seqhilo 通过 hilo 算法实现，但是主键历史保存在 Sequence 中，适用于支持 Sequence 的数据库，如 Orcale(比较少用)

*　　例：
@GeneratedValue(generator = "paymentableGenerator")

*　　@GenericGenerator(name = "paymentableGenerator", strategy = "seqhilo",

*　　parameters = { @Parameter(name = "max_lo", value = "5") })

*　　9.increnment　插入数据的时候 hibernate 会给主键添加一个自增的主键，但是一个 hibernate 实例就维护一个计数器，所以在多个实例运行的时候不能使用这个方法。

*　　例：
@GeneratedValue(generator = "paymentableGenerator")

*　　@GenericGenerator(name = "paymentableGenerator", strategy = "increnment")

*　　10.foreign　使用另一个相关的对象的主键。通常和<one-to-one>联合起来使用。

*　　例：@Id

*　　@GeneratedValue(generator = "idGenerator")

*　　@GenericGenerator(name = "idGenerator", strategy = "foreign",

*　　parameters = { @Parameter(name = "property", value = "info") })

*　　Integer id;

* @OneToOne

* EmployeeInfo info;

*　　11.guid　采用数据库底层的 guid 算法机制，对应 MySQL 的 uuid()函数，SQL Server 的 newid()函数，ORCALE 的 rawtohex(sys_guid())函数等

*　　例：
@GeneratedValue(generator = "paymentableGenerator")

*　　@GenericGenerator(name = "paymentableGenerator", strategy = "guid")

*　　12.uuid.hex　看 uudi,建议用 uuid 替换

*　　例：
@GeneratedValue(generator = "paymentableGenerator")

*　　@GenericGenerator(name = "paymentableGenerator", strategy = "uuid.hex")

*　　13.sequence-identity sequence 策略的扩展，采用立即检索策略来获取 sequence 值，需要 JDBC3.0 和 JDK4 以上（含 1.4）版本

*　　例：
@GeneratedValue(generator = "paymentableGenerator")

*　　@GenericGenerator(name = "paymentableGenerator", strategy = "sequence-identity",

*　　parameters = { @Parameter(name = "sequence", value = "seq_payablemoney") })

*

* @OneToOne 设置一对一个关联。cascade 属性有五个值(只有 CascadeType.ALL 好用？很奇怪)，分别是 CascadeType.PERSIST(级联新建)，CascadeType.REMOVE（级联删除），CascadeType.REFRESH（级联刷新），CascadeType.MERGE（级联更新），CascadeType.ALL（全部四项）

*　　方法一

*　　主表：　?@OneToOne(cascade = CascadeType.ALL)

*　　@PrimaryKeyJoinColumn

*　　public 从表类 get 从表类(){return 从表对象}

*　　从表：没有主表类。

*　　注意：这种方法要求主表与从表的主键值想对应。

*　　方法二

*　　主表：?@OneToOne(cascade = CascadeType.ALL)

*　　@JoinColumn(name="主表外键")　//这里指定的是数据库中的外键字段。

*　　public　从表类　get从表类(){return　从表类}

*　　从表：@OneToOne(mappedBy　=　"主表类中的从表属性")//例主表User中有一个从表属性是Heart类型的heart，这里就填heart

*　　public　主表类　get主表类(){return　主表对象}

*　　注意：@JoinColumn是可选的。默认值是从表变量名+"_"+从表的主键（注意，这里加的是主键。而不是主键对应的变量）。

*　方法三

*　　主表：@OneToOne(cascade=CascadeType.ALL)

*　　@JoinTable(　name="关联表名",

*　joinColumns　=　@JoinColumn(name="主表外键"),

*　inverseJoinColumns　=　@JoinColumns(name="从表外键")

*　　)

*　　从表：@OneToOne(mappedBy　=　"主表类中的从表属性")//例主表User中有一个从表属性是Heart类型的heart，这里就填heart

*　　public　主表类　get主表类(){return　主表对象}

*　@ManyToOne　设置多对一关联

*　　方法一

*　　@ManyToOne(cascade={CasCadeType.PERSIST,CascadeType.MERGE})

*　　@JoinColumn(name="外键")

*　　public　主表类　get 主表类(){return　主表对象}

*　　方法二

*　　@ManyToOne(cascade={CascadeType.PERSIST,CascadeType.MERGE})

*　　@JoinTable(name="关联表名",

*　　joinColumns　=　@JoinColumn(name="主表外键"),

*　　inverseJoinColumns　=　@JoinColumns(name="从表外键")

*　　)

*　@OneToMany　设置一对多关联。cascade 属性指定关联级别,参考 @OneToOne 中的说明。fetch 指定是否延迟加载，值为 FetchType.LAZY 表示延迟，为 FetchType.EAGER 表示立即加载

*　　方法一　　使用这种配置，在为“一端”添加“多端”时，不会修改“多端”的外键。在“一端”加载时，不会得到“多端”。如果使用延迟加载，在读“多端”列表时会出异常，立即加载在得到多端时，是一个空集合（集合元素为 0）。

*　　“一端”配置

*　　@OneToMany(mappedBy="“多端”的属性")

*　　public　List<“多端”类>　get“多端”列表(){return　“多端”列表}

*　　“多端”配置参考@ManyToOne.

*　　方法二

*　　“一端”配置

*　　@OneToMany(mappedBy="“多端”的属性")

*　　@MapKey(name="“多端”做为 Key 的属性")

*　　public　Map<“多端”做为 Key 的属性的类,主表类>　get“多端”列表（）{return　“多端”列表}

*　　“多端”配置参考@ManyToOne.

*　　方法三　使用这种配置，在为“一端”添加“多端”时，可以修改“多端”的外键。

*　　“一端”配置

*　　@OneToMany

*　　@JoinColumn(name=″“多端”外键″)

*　　public　List<“多端”类>　get“多端”列表(){return　“多端”列表}

*　　“多端”配置参考@ManyToOne.