SpringBoot-JPA-Hibernate复习

JPA

JPA是什么

JPA（Java Persistence API）Java持久化API，是一套Sun公司官方制定的ORM方案，是规范，是标准 ，sun公司自己并没有实现，目前常见主流的JPA框架实现者有：Hibernate （JBoos）、EclipseLink（Eclipse社区）、OpenJPA （Apache基金会）

JPA通过JDK 5.0注解或XML描述对象-关系表的映射关系，并将运行期的实体对象持久化到数据库中。

JPA和Hibernate的关系

- JPA 是 hibernate 的一个抽象（就像JDBC和JDBC驱动的关系）
- JPA 是规范：JPA 本质上就是一种 ORM 规范，不是ORM 框架 —— 因为 JPA 并未提供 ORM 实现，它只是制订了一些规范，提供了一些编程的 API 接口，但具体实现则由 ORM 厂商提供实现
- Hibernate 是实现：Hibernate 除了作为 ORM 框架之外，它也是一种 JPA 实现。从功能上来说， JPA 是 Hibernate 功能的一个子集

JPA主要包括 3方面的技术

- ORM 映射元数据：主要是使用注解的方式来进行映射。JPA 支持 XML 和 JDK 5.0 注解两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中。
- JPA 的 API：用来操作实体对象，执行CRUD操作，框架在后台完成所有的事情，开发者从繁琐的 JDBC和 SQL 代码中解脱出来。
- 查询语言（JPQL）：这是持久化操作中很重要的一个方面，通过面向对象而非面向数据库的查询语言查询数据，避免程序和具体的 SQL 紧密耦合。

JPA 基本注解

JPA 基本注解：@Entity， @Table， @Id， @GeneratedValue，@Column， @Basic，@Transient，@Temporal等

@Entity

@Entity 标注用于实体类声明语句之前，指出该Java 类为实体类，将映射到指定的数据库表。如声明一个实体类 Customer，它将映射到数据库中的 customer 表上。

@Table

当实体类与其映射的数据库表名不同名时需要使用 @Table 标注说明，该标注与 @Entity 标注并列使用，置于实体类声明语句之前，可写于单独语句行，也可与声明语句同行。 @Table 标注的常用选项是 name，用于指明数据库的表名 @Table 标注还有一个两个选项 catalog 和 schema 用于设置表所属的数据库目录或模式，通常为数据库名。uniqueConstraints 选项用于设置约束条件，通常不须设置。

@Id

@Id 标注用于声明一个实体类的属性映射为数据库的主键列。该属性通常置于属性声明语句之前，可与声明语句同行，也可写在单独行上。 @Id 标注也可置于属性的getter方法之前。

@GeneratedValue

@GeneratedValue 用于 标注主键的生成策略 ，通过 strategy 属性指定。默认情况下，JPA 自动选择一个最适合底层数据库的主键生成策略：SqlServer 对应 identity，MySQL 对应 auto increment。

在 javax.persistence.GenerationType 中定义了以下4种可供选择的策略：

> - IDENTITY：采用数据库 ID自增长的方式来自增主键字段，一般用于MySQL和SQLServer，Oracle 不支持这种方式；
> - AUTO： JPA自动选择合适的策略，是默认选项；
> - SEQUENCE：通过序列产生主键，通过 @SequenceGenerator 注解指定序列名，一般用于Oracle和DB2，MySql 不支持这种方式
> - TABLE：通过表产生主键，框架借由表模拟序列产生主键，使用该策略可以使应用更易于数据库移植。

@Basic

@Basic 表示一个简单的属性到数据库表的字段的映射,对于没有任何标注的 getXxxx() 方法,默认即为@Basic

- fetch: 表示该属性的读取策略,有 EAGER 和 LAZY 两种,分别表示主支抓取和延迟加载,默认为 EAGER.
- optional: 表示该属性是否允许为null, 默认为true

@Column

当 实体的属性与其映射的数据库表的列不同名时 需要使用@Column 标注说明，该属性通常置于实体的属性声明语句之前，还可与 @Id 标注一起使用。 @Column 标注的常用属性是 name，用于设置映射数据库表的列名。此外，该标注还包含其它多个属性，如：unique 、nullable、length等。 @Column 标注的 columnDefinition 属性: 表示该字段在数据库中的实际类型. 通常 ORM 框架可以根据属性类型自动判断数据库中字段的类型,但是对于Date类型仍无法确定数据库中字段类型究竟是DATE，TIME还是TIMESTAMP @Column 标注也可置于属性的getter方法之前

@Transient

表示该属性并非一个到数据库表的字段的映射,ORM框架将忽略该属性 如果一个属性并非数据库表的字段映射,就务必将其标示为@Transient,否则ORM框架默认其注解为@Basic

@Temporal

在核心的 Java API 中并没有定义 Date 类型的精度(temporal precision). 而在数据库中,表示 Date 类型的数据有 DATE，TIME 和 TIMESTAMP 三种精度(即单纯的日期,时间,或者两者 兼备).在进行属性映射时可使用@Temporal注解来调整精度

使用JPA持久化对象的步骤

persistence.xml核心配置文件

位置应该是classpath下的META-INF目录下

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
```

```
    <!--配置persistence-unit节点
        持久化单元name:持久化单元名称
        transection-type:事务管理方式
             JTA:分布式事务管理（多个数据库）
             RESOURCE_LOCAL 本地事务管理（一个数据库）-->
    <persistence-unit name="myjpa" transaction-type="RESOURCE_LOCAL">
        <!--jpA实现方式HibernatePersistenceProvider-->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.yan.entity.UserBean</class>
        <!--数据库信息-->
        <properties>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="root"/>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql:///test?
serverTimezone=UTC"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
   <!--可选配置：配置jpa实现方式的配置
        自动创建数据库表:
                    create创建数据表如果有表先删后建
                    create-drop: 也表示创建，只不过再系统关闭前执行一下drop
                    update  创建表如果有表不创建表
                    validate: 启动时验证现有schema与你配置的hibernate是否一致，如果不一致就抛出异
常，并不做更新 -->
            <property name="hibernate.show_sql" value="true"/>   显示SQL
            <property name="hibernate.hbm2ddl.auto" value="create"/>
        </properties>
     </persistence-unit>
</persistence>
```

数据库连接相关配置

```
<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
<property name="javax.persistence.jdbc.url" value="jdbc:mysql:///test"/>
<property name="javax.persistence.jdbc.user" value="root"/>
<property name="javax.persistence.jdbc.password" values="123456"/>
```

指定JPA使用哪个持久化框架，以及配置框架的基本属性

```
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.hbm2ddl.auto" value="update"/>
```

在文件中配置持久化单元

```
<class>com.yan.UserBean</class>
```

使用API完成增、删、改、查操作

创建EntityManagerFactory

```
EntityManagerFactory fac=Persistence.createEntityManagerFactory("yan");
```

创建EntityManager

```
EntityManager em=fac.createEntityManager();
```

开启事务

```
EntityTransaction tx=em.getTransaction();
tx.begin();
```

进行持久化操作

```
UserBean user=new UserBean();
em.persist(user);
```

提交事务

```
tx.commit();
```

关闭操作

JPA相关接口和类

Persistence

通过createEntityManagerFactory的静态方法，获取EntityManagerFactory实体管理器工厂实例。属于即用即丢型对象，最佳的生命周期范围为方法内部

带有一个参数：以JPA配置文件presistence.xml中持久化单元名为参数。

EntityManagerFactory实体管理器工厂

作用：用来创建EntityManager实例。是一个线程安全的、重量级长生命周期对象，一般针对一个数据库只创建一次，最佳的软件开发实践为单例模式

- createEntityManager()：用于创建实体管理器对象实例
- isOpen()：检查 EntityManagerFactory 是否处于打开状态。实体管理器工厂创建后一直处于打开状态，除非调用close()方法将其关闭。
- close()：关闭 EntityManagerFactory 。 EntityManagerFactory 关闭后将释放所有资源，isOpen()方法测试将返回 false，其它方法将不能调用，否则将导致IllegalStateException异常

```
public class JpaFactory {
    private static EntityManagerFactory factory;
    private static String PERSISTENCE_UNIT_NAME = "yan1";
```

```
4
5       private JpaFactory() {
6       }
7
8       static {
9           buildFactory();
10      }
11
12      public static EntityManagerFactory getFactory() {
13          return factory;
14      }
15
16      private static void buildFactory() {
17          if (factory == null)
18              factory = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
19      }
20  }
21
```

EntityManager实体管理器

作用：提供最基本的CRUD操作，是完成持久化操作的核心，即java实体类对象经过EntityManager变为持久化对象。是一个轻量级的线程不安全的短生命周期对象，最佳的软件开发实践采用ThreadLocal和OpenSessionInViewFilter模式进行管理，是一个数据库连接对象的浅封装，必须保证及时关闭 EntityManager：在一组实体类与底层数据源之间进行 O/R 映射的管理。

修改工具类添加对实体管理器的管理

```
1   private static final ThreadLocal<EntityManager> ems=new ThreadLocal<>();
2
3   public static EntityManager openEntityManager() {
4           EntityManager res = ems.get();
5           if(res==null) {
6               res=factory.createEntityManager();
7               ems.set(res);
8           }
9           return res;
10  }
11
12  public static void closeEntityManager() {
13          EntityManager em = ems.get();
14          ems.set(null);
15          if (em != null)
16              em.close();
17  }
```

添加过滤器保证entityManager及时关闭

```
1   @Slf4j
2   public class OpenEntityManagerFilter implements Filter {
3
        @Override
```

```
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
            throws IOException, ServletException {
        try {
            chain.doFilter(request, response);
        } catch (Exception e) {
            log.debug(e.getMessage());
            throw new ServletException(e);
        } finally {
            JpaFactory.closeEntityManager();  //保证及时关闭，至于打开的时机取决于调用DAO的时机
        }
    }
}
```

find(Class entityClass,Object primaryKey)

作用：获得指定ID的实体类对象，若不存在则返回null。

```
public class RoleDaoImpl implements IRoleDao {
    @Override
    public Role loadById(Long id) {
        EntityManager em=JpaFactory.openEntityManager();
        return em.find(Role.class, id);
    }
}
```

针对返回null的处理，不是必须的

```
public class RoleDaoImpl implements IRoleDao {
    @Override
    public Optional<Role> loadById(Long id) {
        Optional<Role> res = Optional.empty();
        EntityManager em = JpaFactory.openEntityManager();
        Role temp= em.find(Role.class, id);
        if(temp!=null)
            res=Optional.of(temp);
        return res;
    }
}
```

getReference (Class entityClass,Object primaryKey)

作用：类似于懒加载，首先获得实体类的代理对象，只用在用到时才加载。所以如果此 OID在数据库不存在，getReference()不会返回 null 值, 而是抛出异常EntityNotFoundException

```
public void delete(Long id) {
        EntityManager em = JpaFactory.openEntityManager();
        Role temp=em.getReference(Role.class, id);
        em.remove(temp);

}
```

persist (Object entity)

作用：使对象由临时状态变为持久化状态。

- 如果传入 persist() 方法的 Entity 对象已经处于持久化状态，则persist() 方法什么都不做。
- 如果对删除状态的 Entity 进行 persist() 操作，会转换为持久化状态。
- 如果对游离状态【这个对象已经在数据库中存在，但是并没有和EntityManager建立关系】的实体执行 persist() 操作，可能会在 persist() 方法抛出 EntityExistException(也有可能是在flush或事务提交后抛出)

```
    public void save(Role role) {
        EntityManager em = JpaFactory.openEntityManager();
        em.persist(role);
    }
```

remove (Object entity)

作用：删除实例，同时会删除相关联的数据库记录。

merge (T entity)

作用：merge() 用于处理 Entity 的同步。即数据库的插入和更新操作。如果对象中有id值则执行修改操作，否则执行插入操作

```
    public void update(Role role) {
        EntityManager em = JpaFactory.openEntityManager();
        em.merge(role);
    }
```

EntityTransaction实体事务管理器

用于针对底层的具体数据库事务进行封装，属于一个轻量级、短生命周期、线程不安全的对象，最佳编程实践为方法体内或者使用ThreadLocal进行管理。因为事务管理器是绑定EntityManager对象的，所以一般不使用ThreadLocal再进行管理

```
public static void beginTransaction() {
        EntityManager em=openEntityManager();
        em.getTransaction().begin();
}
public static void commitTransaction() {
        EntityManager em=openEntityManager();
        em.getTransaction().commit();
}
public static void rollbakTransaction() {
        EntityManager em=openEntityManager();
        em.getTransaction().rollback();
}
```

begin()启动一个事务

用于启动一个事务，此后的多个数据库操作将作为整体被提交或撤消。若这时事务已启动则会抛出 IllegalStateException 异常。

commit()提交事务

用于提交当前事务。即将事务启动以后的所有数据库更新操作持久化至数据库中。

rollback()回滚事务

撤消(回滚)当前事务。即撤消事务启动后的所有数据库更新操作，从而不对数据库产生影响。

在一般的web应用中采用的方式为一个请求对应一个事务，所以修改Filter

```
@Slf4j
public class OpenEntityManagerFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
            throws IOException, ServletException {
        try {
            JpaFactory.beginTransaction();
            chain.doFilter(request, response);
            JpaFactory.commitTransaction();
        } catch (Exception e) {
            JpaFactory.rollbakTransaction();
            log.debug(e.getMessage());
            throw new ServletException(e);
        } finally {
            JpaFactory.closeEntityManager();
        }
    }
}
```

Hibernate

注意：hibernate独立使用的前提是优化，一般情况下互联网公司采用的持久层框架是MyBatis，只有老牌的软件公司才会使用Hibernate(JPA)。

从性能的角度上说JDBC>MyBatis>>Hibernate，从开发的角度上说JDBC<MyBatis<<Hibernate。MyBatis实际上是一种半自动化的ORM框架，是以编写SQL语句的代价换取高灵活性

什么是Hibernate框架

Hibernate 是一个开源ORM框架，它是对象关联关系映射的框架，它对 JDBC 做了轻量级的封装，使 java 程序员可以使用面向对象的思想来操纵关系型数据库(RDBMS关系型数据库管理系统)

为什么要用 Hibernate

- 对 JDBC 访问数据库的代码做了封装，大大简化了数据访问层繁琐的重复性代码【有过度封装的嫌疑，优化比较困难,类似LayUI】
- Hibernate 是一个基于 JDBC 的主流持久化框架，是一个优秀的 ORM 实现。它很大程度的简化了 DAO 层的编码工作

- Hibernate 使用 Java 反射机制，而不是字节码增强程序来实现透明性
- Hibernate 的性能非常好，因为它是个轻量级框架。映射的灵活性很出色。它支持各种关系数据库，从一对一到多对多的各种复杂关系

什么是ORM

对象关系映射（Object Relational Mapping，简称ORM）是一种为了解决面向对象与关系数据库存在的互不匹配现象的技术。简单的说，ORM 是通过使用描述对象和数据库之间映射的元数据，将 java 程序中的对象自动持久化到关系数据库中。

主要解决关系模型和对象模型的阻抗不匹配性

本质上就是将数据从一种形式转换到另外一种形式。 这也同时暗示者额外的执行开销；然而，如果 ORM 作为一种中间件实现，则会有很多机会做优化，而这些在手写的持久层并不存在

Hibernate开发

Hibernate的相关配置文件

首先要编写Hibernate的相关配置文件，Hibernate的相关配置文件分为两种：

- xxx.hbm.xml：它主要是用于描述类与数据库中的表的映射关系。映射元文件的定义可以采用xml或者JPA（Hibernate）注解
- hibernate.cfg.xml：它是Hibernate框架的核心配置文件。

映射配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.yan.entity">
    <!--
        name：即实体类的全名
        table：映射到数据库里面的那个表的名称
        catalog：数据库的名称
     -->
    <class name="Customer" table="t_customer" catalog="test">
        <!-- class下必须要有一个id的子元素 -->
        <!-- id是用于描述主键的 -->
        <id name="id" column="id">
            <!-- 主键生成策略，hibernate提供了至少11种主键生成策略 -->
            <generator class="native"></generator>
        </id>
        <!--
            使用property来描述属性与字段的对应关系
            如果length忽略不写，且你的表是自动创建这种方案，那么length的默认长度是255
        -->
        <property name="name" column="name" length="20"></property>
        <property name="address" column="address" length="50"></property>
    </class>

</hibernate-mapping>
```

核心配置文件hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
    <!--数据库连接池相关配置-->
        <!-- 配置关于数据库连接的四个项：driverClass  url username password -->
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql:///hibernateTest</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">yezi</property>
    <!--hibernate运行时常量的配置 -->
        <!-- 可以将向数据库发送的SQL语句显示出来 -->
        <property name="hibernate.show_sql">true</property>
        <!-- 格式化SQL语句 -->
        <property name="hibernate.format_sql">true</property>
        <!-- hibernate的方言 -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <!--注册映射元文件 -->
        <!-- 配置hibernate的映射文件所在的位置 -->
        <mapping resource="com/yan/entity/Customer.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

操作

事实上JPA规范的定义是Hibernate作者参与的，他是EJB3项目组的主导者。两者之间差异性不大，区别在于类名称不一样

```
// 使用Hibernate的API来完成将Customer信息保存到mysql数据库中的操作---Persistence
Configuration config = new Configuration().configure(); // Hibernate框架加载hibernate.cfg.xml文件
SessionFactory sessionFactory = config.buildSessionFactory();  //EntityManagerFactory
Session session = sessionFactory.openSession(); // 相当于得到一个Connection  //EntityManager
        // 开启事务
session.beginTransaction();  //EntityTransaction
        // 操作
session.save(c);
        // 事务提交
session.getTransaction().commit();
session.close();
```

Hibernate执行原理

- 通过Configuration().configure();读取并解析hibernate.cfg.xml配置文件。
- 由hibernate.cfg.xml中的读取解析映射信息。

- 通过config.buildSessionFactory();得到sessionFactory。
- sessionFactory.openSession();得到session。
- session.beginTransaction();开启事务。
- persistent operate; 执行你自己的操作。
- session.getTransaction().commit();提交事务。
- 关闭session。
- 关闭sessionFactory。

Hibernate核心API

Configuration:Hibernate的配置对象：

Configuration类的作用是对Hibernate进行配置，以及对它进行启动。在Hibernate的启动过程中，Configuration类的实例首先定位映射文档的位置，读取这些配置，然后创建一个SessionFactory对象。虽然Configuration类在整个Hibernate项目中只扮演着一个很小的角色，但它是启动Hibernate时所遇到的第一个对象。

作用：加载核心配置文件。hibernate.properties和hibernate.cfg.xml

加载映射文件

ServiceRegistry 接口

是从Hibernate4开始新增方法，所有基于 Hibernate 的配置或者服务都必须统一向这个 ServiceRegistry 注册后才能生效

```
//创建 configuration 对象
Configuration configuration = new Configuration().configure();
//创建 serviceRegistry 对象,这是Hibernate4 新增了一个ServiceRegistry 接口，所有基于 Hibernate 的配置或者服务都必须统一向这个ServiceRegistry  注册后才能生效
ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildServiceRegistry();
//创建  sessionFactory 对象
SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

SessionFactory:Session工厂

SessionFactory内部维护了Hibernate的连接池和HIbernate的二级缓存。是线程安全的对象。一个项目创建一个对象即可

```
<!-- 配置C3P0连接池 -->
        <property name="connection.provider_class">org.hibernate.connection.C3P0ConnectionProvider</property>
        <!--在连接池中可用的数据库连接的最少数目 -->
        <property name="c3p0.min_size">5</property>
        <!--在连接池中所有数据库连接的最大数目  -->
        <property name="c3p0.max_size">20</property>
        <!--设定数据库连接的过期时间,以秒为单位,
        如果连接池中的某个数据库连接处于空闲状态的时间超过了timeout时间,就会从连接池中清除 -->
        <property name="c3p0.timeout">120</property>
         <!--每3000秒检查所有连接池中的空闲连接 以秒为单位-->

        <property name="c3p0.idle_test_period">3000</property>
```

Session：会话对象

类似Connection对象是连接对象，充当实体管理器功能，Session代表的是Hibernate与数据库的连接对象。非线程安全的。是与数据库交互的桥梁。

保存方法：

Serializable save(Object obj);

查询方法：

T get(Class c,Serializable id);

T load(Class c,Serializable id);

get方法和load方法的区别：

- get方法：

采用的是立即加载，执行到这行代码的时候就马上发送SQL语句去查询。 查询后返回的是真实对象本身。 查询一个找不到的对象的时候，返回null。

- load方法：

采用的是延迟加载，也称为lazy懒加载，执行到这行代码的时候，不会发送SQL语句，当真正使用这个对象的时候才会发送SQL语句。 查询后返回的是代理对象。javaassist-3.18.1-GA.jar 利用javassist技术产生的代理。 查询一个找不到的对象的时候，返回ObjectNotFoundException。

删除方法：

void delete(Object obj);

保存和更新：

void saveOrUpdate(Object obj);

查询数据库所有：

```
Query query = session.createQuery("from Customer");
List<Customer> customerList = query.list();
```

Transaction通用事务接口

代表一次原子操作，它具有数据库事务的概念。所有持久层都应该在事务管理下进行，即使是只读操作。

commit(): 提交相关联的session实例

rollback(): 撤销事务操作

Hibernate提供的5种查询方法

OID、HQL，QBC、NativeSQL、OGN

```
SQLQuery query=session.createSQLQuery("select * from t_users");
List<Object[]> list=query.list();

也可以
query.addEntity(User.class);
List<User> list=query.list();
```

对象状态与一级缓存

hibernate规定三种状态：瞬时态、持久态、脱管态。

- 瞬时态(临时态、自由态)：不存在持久化标识OID，尚未与Hibernate Session关联对象，被认为处于瞬时态，失去引用将被JVM回收
- 持久态：存在持久化标识OID，与当前session有关联，并且相关联的session没有关闭,并且事务未提交
- 脱管态(离线态、游离态)：存在持久化标识OID，但没有与当前session关联，脱管状态改变hibernate不能检测到

状态：

瞬时态：transient，session没有缓存对象，数据库也没有对应记录。

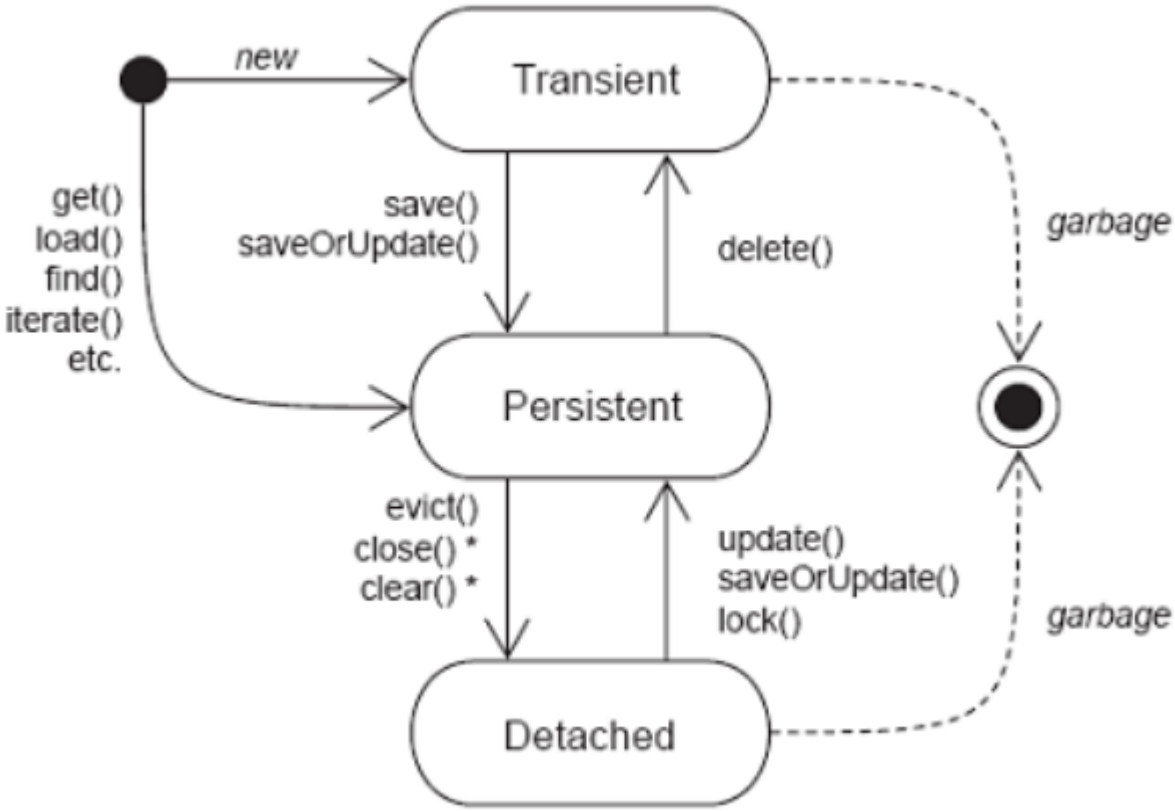
OID特点：没有值。

持久态：persistent，session缓存对象，数据库最终会有记录。（事务没有提交）

OID特点：有值。

脱管态：detached，session没有缓存对象，数据库有记录。

OID特点：有值。



Hibernate与Ibatis/MyBatis的区别

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，建立对象与数据库表的映射。是一个全自动的、完全面向对象的持久层框架。 Mybatis 是一个开源对象关系映射框架，原名：ibatis，2010年由谷歌接管以后更名。是一个半自动化的持久层框架。

两者区别

- 开发方面 在项目开发过程当中，就速度而言：hibernate 开发中，sql 语句已经被封装，直接可以使用，加快系统开发；Mybatis 属于半自动化，sql 需要手工完成，稍微繁琐；但是，凡事都不是绝对的，如果对于庞大复杂的系统项目来说，发杂语句较多，选择hibernate 就不是一个好方案。
- sql 优化方面 Hibernate 自动生成 sql，有些语句较为繁琐，会多消耗一些性能；Mybatis 手动编写 sql，可以避免不需要的查询，提高系统性能
- 对象管理 Hibernate 是完整的对象-关系映射的框架，开发工程中，无需过多关注底层实现，只要去管理对象即可；Mybatis 需要自行管理 映射关系
- 缓存方面
- 相同点： Hibernate 和 Mybatis 的二级缓存除了采用系统默认的缓存机制外，都可以通过实现你自己的缓存或为其他第三方缓存方案，创建适配器来完全覆盖缓存行为。
- 不同点： Hibernate 的二级缓存配置在 SessionFactory 生成的配置文件中进行详细配置，然后再在具体的表-对象映射中配置那种缓存。

  MyBatis 的二级缓存配置都是在每个具体的表-对象映射中进行详细配置，这样针对不同的表可以自定义不同的缓存机制。并且 Mybatis 可以在命名空间中共享相同的缓存配置和实例，通过 Cache-ref 来实现。

- 比较 Hibernate 具有良好的管理机制，用户不需要关注 SQL，如果二级缓存出现脏数据，系统会保存；Mybatis 在使用的时候要谨慎，避免缓存 CAche 的使用。
- Hibernate 优势

  Hibernate 的 DAO 层开发比 MyBatis 简单，Mybatis 需要维护 SQL 和结果映射。

  Hibernate 对对象的维护和缓存要比 MyBatis 好，对增删改查的对象的维护要方便。

  Hibernate 数据库移植性很好，MyBatis 的数据库移植性不好，不同的数据库需要写不同 SQL。

  Hibernate 有更好的二级缓存机制，可以使用第三方缓存。MyBatis 本身提供的缓存机制不佳。
- Mybatis 优势

  MyBatis 可以进行更为细致的 SQL 优化，可以减少查询字段。

  MyBatis容易掌握，而Hibernate门槛较高。

一句话总结

- Mybatis：小巧、方便、高效、简单、直接、半自动化
- Hibernate：强大、方便、高效、复杂、间接、全自动化

什么是延迟加载

延迟加载机制是为了避免一些无谓的性能开销而提出来的，所谓延迟加载就是当在真正需要数据的时候，才真正执行数据加载操作。在Hibernate中提供了对实体对象的延迟加载以及对集合的延迟加载，另外在Hibernate3中还提供了对属性的延迟加载

什么是缓存

缓存是介于物理数据源与应用程序之间，是对数据库中的数据复制一份临时放在内存中的容器，其作用是为了减少应用程序对物理数据源访问的次数，从而提高了应用程序的运行性能。

Hibernate在进行读取数据的时候，根据缓存机制在相应的缓存中查询，如果在缓存中找到了需要的数据(我们把这称做"缓存命 中")，则就直接把命中的数据作为结果加以利用，避免了大量发送SQL语句到数据库查询的性能损耗。

缓存策略提供商：

- 提供了HashTable缓存，EHCache，OSCache，SwarmCache，jBoss Cathe2，这些缓存机制，其中EHCache，OSCache是不能用于集群环境（Cluster Safe）的，而SwarmCache，jBoss Cathe2是可以的。
- HashTable缓存主要是用来测试的，只能把对象放在内存中，EHCache，OSCache可以把对象放在内存（memory）中，也可以把对象放在硬盘（disk）上。

> EHCache（主要学习，支持本地缓存，支持分布式缓存）　可作为进程范围内的缓存, 存放数据的物理介质可以是内存或硬盘, 对 Hibernate 的查询缓存提供了支持。
>
> OSCache可作为进程范围内的缓存, 存放数据的物理介质可以是内存或硬盘, 提供了丰富的缓存数据过期策略, 对 Hibernate 的查询缓存提供了支持
>
> SwarmCache 可作为集群范围内的缓存, 但不支持 Hibernate 的查询缓存
>
> JBossCache 可作为集群范围内的缓存, 支持 Hibernate 的查询缓存

Hibernate缓存分类

- Session缓存（又称作事务缓存）：一级缓存，Hibernate内置的，不能卸除。

  缓存范围：缓存只能被当前Session对象访问。缓存的生命周期依赖于Session的生命周期，当Session被关闭后，缓存也就结束生命周期。

- SessionFactory缓存（又称作应用缓存）：二级缓存，使用第三方插件，可插拔。

  缓存范围：缓存被应用范围内的所有session共享,不同的Session可以共享。这些session有可能是并发访问缓存，因此必须对缓存进行更新。缓存的生命周期依赖于应用的生命周期，应用结束时，缓存也就结束了生命周期，二级缓存存在于应用程序范围。

一级缓存

数据放入缓存：

- save()。当session对象调用save()方法保存一个对象后，该对象会被放入到session的缓存中。
- get()和load()。当session对象调用get()或load()方法从数据库取出一个对象后，该对象也会被放入到session的缓存中。
- 使用HQL和QBC等从数据库中查询数据。

其原理是：在同一个Session里面，第一次调用get()方法， Hibernate先检索缓存中是否有该查找对象，发现没有，Hibernate发送SELECT语句到数据库中取出相应的对象，然后将该对象放入缓存中，以便下次使用，第二次调用get()方法，Hibernate先检索缓存中是否有该查找对象，发现正好有该查找对象，就从缓存中取出来，不再去数据库中检索，没有再次发送select语句。

数据从缓存中清除：

- evit()将指定的持久化对象从缓存中清除，释放对象所占用的内存资源，指定对象从持久化状态变为脱管状态，从而成为游离对象。
- clear()将缓存中的所有持久化对象清除，释放其占用的内存资源。

其他缓存操作：

- contains()判断指定的对象是否存在于缓存中。
- flush()刷新缓存区的内容，使之与数据库数据保持同步。

二级缓存

SessionFactory级别的缓存，可以跨越Session存在，可以被多个Session所共享

适合放到二级缓存中

- 经常被访问
- 改动不大
- 数量有限
- 不是很重要的数据，允许出现偶尔并发的数据。

这样的数据非常适合放到二级缓存中的。

二级缓存实现原理

Hibernate如何将数据库中的数据放入到二级缓存中？注意，你可以把缓存看做是一个Map对象，它的Key用于存储对象OID，Value用于存储POJO。首先，当我们使用Hibernate从数据库中查询出数据，获取检索的数据后，Hibernate将检索出来的对象的OID放入缓存中key 中，然后将具体的POJO放入value中，等待下一次再次向数据查询数据时，Hibernate根据你提供的OID先检索一级缓存，若有且配置了二级缓存，则检索二级缓存，如果还没有则才向数据库发送SQL语句，然后将查询出来的对象放入缓存中。

使用二级缓存

在主配置文件中hibernate.cfg.xml ：

```
<!-- 使用二级缓存 -->
<property name="hibernate.cache.use_second_level_cache">true</property>
<!--设置缓存的类型，设置缓存的提供商-->
<property
name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
```

```
hibernate.cache.use_second_level_cache=true
hibernate.cache.provider_class=org.hibernate.cache.EhCacheProvider
hibernate.generate_statistics=true
```

配置ehcache.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
缓存到硬盘的路径
   <diskStore path="d:/ehcache"></diskStore>
默认设置
  maxElementsInMemory ： 在内存中最大緩存的对象数量。
  eternal ： 缓存的对象是否永远不变。
  timeToIdleSeconds ：可以操作对象的时间。
  timeToLiveSeconds ：缓存中对象的生命周期，时间到后查询数据会从数据库中读取。
  overflowToDisk ：内存满了，是否要缓存到硬盘。
<defaultCache maxElementsInMemory="200" eternal="false" timeToIdleSeconds="50"
timeToLiveSeconds="60" overflowToDisk="true"></defaultCache>
</ehcache>
```

并发访问策略配置

在实体类中通过注解可以配置实用二级缓存

> @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)

```
<class name="com.yan.entity.Order" table="orders">
   <cache usage="read-only"/>
```

Spring data jpa

Spring Data JPA是Spring Data家族的一部分，可以轻松实现基于JPA的存储库。 此模块处理对基于JPA的数据访问层的增强支持。 它使构建使用数据访问技术的Spring驱动应用程序变得更加容易。

在相当长的一段时间内，实现应用程序的数据访问层一直很麻烦。 必须编写太多样板代码来执行简单查询以及执行分页和审计。 Spring Data JPA旨在通过减少实际需要的工作量来显著改善数据访问层的实现。 作为开发人员，您编写repository接口，包括自定义查找器方法，Spring将自动提供实现。

Jpa、Hibernate、Spring Data Jpa三者之间的关系

总的来说JPA是ORM规范，Hibernate、TopLink等是JPA规范的具体实现，这样的好处是开发者可以面向JPA规范进行持久层的开发，而底层的实现则是可以切换的。Spring Data Jpa则是在JPA之上添加另一层抽象（Repository层的实现），极大地简化持久层开发及ORM框架切换的成本。

配置

```yaml
spring:
  jpa:
    database: MySQL
    database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
    show-sql: true
    hibernate:
      ddl-auto: update
```

接口说明

Repository接口

Repository 接口是 Spring Data JPA 中为我我们提供的所有接口中的顶层接口 Repository 提供了两种查询方式的支持 1)基于方法名称命名规则查询 2)基于@Query 注解查询

规则: findBy(关键字)+属性名称(属性名称的首字母大写)+查询条件(首字母大写)

关键字	方法命名	sql where 字句
And	findByNameAndPwd	where name= ? and pwd =?
Or	findByNameOrSex	where name= ? or sex=?
Is,Equal	findById	findByIdEquals
Between	findByIdBetween	where id between ? and ?
LessThan	findByIdLessThan	where id < ?
LessThanEqual	findByIdLessThanEquals	where id <= ?
GreaterThan	findByIdGreaterThan	where id > ?
GreaterThanEqual	findByIdGreaterThanEquals	where id > = ?
After	findByIdAfter	where id > ?
Before	findByIdBefore	where id < ?
IsNull	findByNameIsNull	where name is null
isNotNull,NotNull	findByNameNotNull	where name is not null
Like	findByNameLike	where name like ?
NotLike	findByNameNotLike	where name not like ?
StartingWith	findByNameStartingWith	where name like '?%'
EndingWith	findByNameEndingWith	where name like '%?'
Containing	findByNameContaining	where name like '%?%'
OrderBy	findByIdOrderByXDesc	where id=? order by x desc
Not	findByNameNot	where name <> ?
In	findByIdIn(Collection<?> c)	where id in (?)
NotIn	findByIdNotIn(Collection<?> c)	where id not in (?)
True	findByAaaTue	where aaa = true
False	findByAaaFalse	where aaa = false
IgnoreCase	findByNameIgnoreCase	where UPPER(name)=UPPER(?)

```java
public interface UsersDao extends Repository<Users, Integer> {
    //使用@Query注解查询
    @Query(value="from Users where username = ?")
    List<Users> queryUserByNameUseJPQL(String name);

    //nativeQuery:默认的是false.表示不开启sql查询。是否对value中的语句做转义。
    @Query(value="select * from t_users where username = ?",nativeQuery=true)
    List<Users> queryUserByNameUseSQL(String name);

    @Query("update Users set userage = ? where userid = ?")
    @Modifying //@Modifying当前语句是一个更新语句
    void updateUserAgeById(Integer age,Integer id);
```

CrudRepository接口

PagingAndSortingRepository 接口

```
int page = 2; //page:当前页的索引。注意索引都是从0开始的。
int size = 3;// size:每页显示3条数据
Pageable pageable= new PageRequest(page, size);
Page<Users> p = this.usersDao.findAll(pageable);
System.out.println("数据的总条数："+p.getTotalElements());
System.out.println("总页数："+p.getTotalPages());
List<Users> list = p.getContent();
```

JpaRepository接口

说明

1. 定义了接口继承JpaRepository<T,ID>后，只需在配置类上注解EnableJpaRespositories(basePackes="")便会自动扫描指定包下的Repository接口，为其生成相应的代理类。@EnableJapRespositories注解的作用类似于Mybatis中的MapperScannerConfigurer Bean
2. JpaRepository中的getOne方法，类似于Hibernate中的load于Iterator方法，会产生懒加载的问题，在查询时会返回一个代理对象，对象在第一次使用前不能关闭事务，连接。
3. 当实现类的方法上注解了@Transactional后，不能通过该实现类来获得Bean，只能通过该实现类的接口类来获得Bean。
   原理：开启事务需要使用到AOP的功能，而AOP功能又要通过代理来实现，默认使 用的JDK代理只能通过接口来生成代理类。
4. Repository接口中支持自定义查询。
   @Query("select s from student s where s.id = ?1 ")
   public Student find (int id)
5. 懒加载导致问题的解决：过滤器 OpensessionInViewFilter (页面打开时开启Session,页面关闭时关闭Session)

Spring Data REST

Spring Data REST 作为 Spring Data 项目的子集，开发者只需使用注解 `@RepositoryRestResource` 标记，就可以把整个Repository转换为HAL风格的REST资源，目前已支持Spring Data JPA、Spring Data MongoDB、Spring Data Neo4j等

HAL（Hypertxt Application Language）是一个被广泛采用的超文本表达的规范。应用可以考虑遵循该规范

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

Spring Data REST把需要手工编写的大量REST模版接口做了自动化实现

Spring Data Rest设计的目的是消除curd的模板代码，减少程序员的刻板的重复劳动，尽管拥有强大的功能和精妙的设计，但它作为Spring Data系列产品，终究不能完全代替传统的SpringMVC，其特点也如Spring Data JPA之与Spring Data JDBC等低封装度的产品，高度封装了许多细节，但在用法上有它自己的一套规则。

```
spring:
  data:
    rest:
      # Restful API 路径前缀
      base-path: api
      max-page-size: 10
      default-page-size: 5
  datasource:
    url: jdbc:mysql://localhost:3306/test?
useUnicode=true&zeroDateTimeBehavior=convertToNull&characterEncoding=utf-
8&useSSL=false&serverTimezone=GMT%2B8&tinyInt1isBit=false
    username: root
    password: root
  mvc:
    servlet:
      load-on-startup: 1
    throw-exception-if-no-handler-found: true
  jpa:
    database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
    hibernate:
      ddl-auto: update
    show-sql: true
    open-in-view: false
  jackson:
    time-zone: GMT+8
logging:
  level:
    web: debug
```

定义实体类

```java
@Data
@MappedSuperclass
@NoArgsConstructor
@AllArgsConstructor
public class BaseEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @UpdateTimestamp
    @Column(nullable = false)
    private Date updateTime;

    @CreationTimestamp
    @Column(nullable = false, updatable = false)
    private Date createTime;

    @Version
    private Long version;

    @NotNull
    private Boolean deleted = false;

}
```

```java
@Data
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(callSuper = false)
public class User extends BaseEntity {
    @NotBlank
    private String name;
     @JsonIgnore
    private String password;
    @NotNull
    private Boolean sex;
}
```

添加 Repository

```java
//Spring Data REST默认规则是在实体类之后加上"s"来形成路径，我们可以通过@RepositoryRestResource注解的path属性进行修改
@RepositoryRestResource(path = "user")
public interface UserRepository extends JpaRepository<User, Integer> {
    //请求路径为  /api/user/search/findByName
    List<User> findByName(@Param("name") String name);

   //请求路径为：/api/user/search/updateDeletedById{?id}
```

```
    @Transactional
    @Modifying
    @Query("UPDATE User u SET u.deleted = true WHERE u.id = ?1")
    int updateDeletedById(Integer id);
//在实际生产环境中，不会轻易的删除用户数据，此时不希望DELETE的提交方式生效，可以添加@RestResource注解，并设置exported=false，即可屏蔽Spring Data REST的自动化方法
    @RestResource(exported = false)
    public void delete(Integer id);
}
```

请求方式	请求路径	接口说明
GET	http://ip:port/api/user{?page,size,sort}	分页查询
GET	http://ip:port/api/user/1	查询id为1的用户
GET	http://ip:port/api/user/search/findByName?name=xxx	查询name为xxx的用户
POST	http://ip:port/api/user	新增用户，注意测试时提交数据应该为Content-Type:application/json，数据为raw类型json格式的
PUT	http://ip:port/api/user/1	更新id为1的用户
DELETE	http://ip:port/api/user/1	删除id为1的用户

HAL Browser 使用

HAL-browser 是基于hal+json的media type的API浏览器，Spring Data Rest 提供了集成。启动后打开浏览器http://127.0.0.1:8080/api/browser/index.html#/api