<https://www.cnblogs.com/jiangxiaoyaoblog/p/5635152.html>

# [Spring Hibernate JPA 联表查询 复杂查询](http://www.cnblogs.com/jiangxiaoyaoblog/p/5635152.html)

今天刷网，才发现：

1）如果想用hibernate注解，是不是一定会用到jpa的？

是。如果hibernate认为jpa的注解够用，就直接用。否则会弄一个自己的出来作为补充。

2）jpa和hibernate都提供了Entity，我们应该用哪个，还是说可以两个一起用？

Hibernate的Entity是继承了jpa的，所以如果觉得jpa的不够用，直接使用hibernate的即可

 正文：

### 一、Hibernate VS Mybatis

#### 1、简介

    Hibernate对数据库结构提供了较为完整的封装，Hibernate的O/R Mapping实现了POJO 和数据库表之间的映射，以及SQL 的自动生成和执行。程序员往往只需定义好了POJO 到数据库表的映射关系，即可通过Hibernate 提供的方法完成持久层操作。程序员甚至不需要对SQL 的熟练掌握， Hibernate/OJB 会根据制定的存储逻辑，自动生成对应的SQL 并调用JDBC 接口加以执行。

    iBATIS 的着力点，则在于POJO 与SQL之间的映射关系。然后通过映射配置文件，将SQL所需的参数，以及返回的结果字段映射到指定POJO。 相对Hibernate“O/R”而言，iBATIS 是一种“Sql Mapping”的ORM实现

#### 2、开发对比

    Hibernate的真正掌握要比Mybatis来得难些。Mybatis框架相对简单很容易上手，但也相对简陋些。个人觉得要用好Mybatis还是首先要先理解好Hibernate。针对高级查询，Mybatis需要手动编写SQL语句，以及ResultMap。而Hibernate有良好的映射机制，开发者无需关心SQL的生成与结果映射，可以更专注于业务流程

#### 3、系统调优对比

Hibernate调优方案：

1. 制定合理的缓存策略；
2. 尽量使用延迟加载特性；
3. 采用合理的Session管理机制；
4. 使用批量抓取，设定合理的批处理参数（batch\_size）;
5. 进行合理的O/R映射设计

Mybatis调优方案：

    MyBatis在Session方面和Hibernate的Session生命周期是一致的，同样需要合理的Session管理机制。MyBatis同样具有二级缓存机制。 MyBatis可以进行详细的SQL优化设计。

#### SQL优化方面：

    Hibernate的查询会将表中的所有字段查询出来，这一点会有性能消耗。Hibernate也可以自己写SQL来指定需要查询的字段，但这样就破坏了Hibernate开发的简洁性。而Mybatis的SQL是手动编写的，所以可以按需求指定查询的字段。

    Hibernate HQL语句的调优需要将SQL打印出来，而Hibernate的SQL被很多人嫌弃因为太丑了。MyBatis的SQL是自己手动写的所以调整方便。但Hibernate具有自己的日志统计。Mybatis本身不带日志统计，使用Log4j进行日志记录。

#### 4、缓存机制对比

Hibernate缓存：

Hibernate一级缓存是Session缓存，利用好一级缓存就需要对Session的生命周期进行管理好。建议在一个Action操作中使用一个Session。一级缓存需要对Session进行严格管理。Hibernate二级缓存是SessionFactory级的缓存。 SessionFactory的缓存分为内置缓存和外置缓存。内置缓存中存放的是SessionFactory对象的一些集合属性包含的数据(映射元素据及预定SQL语句等),对于应用程序来说,它是只读的。外置缓存中存放的是数据库数据的副本,其作用和一级缓存类似.二级缓存除了以内存作为存储介质外,还可以选用硬盘等外部存储设备。二级缓存称为进程级缓存或SessionFactory级缓存，它可以被所有session共享，它的生命周期伴随着SessionFactory的生命周期存在和消亡。

Mybatis缓存：

MyBatis 包含一个非常强大的查询缓存特性,它可以非常方便地配置和定制。MyBatis 3 中的缓存实现的很多改进都已经实现了,使得它更加强大而且易于配置。

默认情况下是没有开启缓存的,除了局部的 session 缓存,可以增强变现而且处理循环 依赖也是必须的。要开启二级缓存,你需要在你的 SQL 映射文件中添加一行:  <cache/>

字面上看就是这样。这个简单语句的效果如下:

1. 映射语句文件中的所有 select 语句将会被缓存。
2. 映射语句文件中的所有 insert,update 和 delete 语句会刷新缓存。
3. 缓存会使用 Least Recently Used(LRU,最近最少使用的)算法来收回。
4. 根据时间表(比如 no Flush Interval,没有刷新间隔), 缓存不会以任何时间顺序 来刷新。
5. 缓存会存储列表集合或对象(无论查询方法返回什么)的 1024 个引用。
6. 缓存会被视为是 read/write(可读/可写)的缓存,意味着对象检索不是共享的,而 且可以安全地被调用者修改,而不干扰其他调用者或线程所做的潜在修改。

所有的这些属性都可以通过缓存元素的属性来修改。

#### 5、总结

Mybatis：小巧、方便、高效、简单、直接、半自动

Hibernate：强大、方便、高效、复杂、绕弯子、全自动

### 二、Hibernate & JPA

#### 1、JPA

    全称Java Persistence API，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的出现有两个原因：  
 其一，简化现有Java EE和Java SE应用的对象持久化的开发工作；  
 其二，Sun希望整合对ORM技术，实现持久化领域的统一。

     JPA提供的技术：

 1）ORM映射元数据：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；

 2）JPA 的API：用来操作实体对象，执行CRUD操作，框架在后台替我们完成所有的事情，开发者从繁琐的JDBC和SQL代码中解脱出来。

 3）查询语言：通过面向对象而非面向数据库的查询语言查询数据，避免程序的SQL语句紧密耦合。

#### 2、JPA & Hibernate 关系

    JPA是需要Provider来实现其功能的，Hibernate就是JPA Provider中很强的一个。从功能上来说，JPA现在就是Hibernate功能的一个子集。可以简单的理解为JPA是标准接口，Hibernate是实现。Hibernate主要是通过三个组件来实现的，及hibernate-annotation、hibernate-entitymanager 和hibernate-core。

1）hibernate-annotation是Hibernate支持annotation方式配置的基础，它包括了标准的JPA annotation以及  Hibernate自身特殊功能的annotation。

2）hibernate-core是Hibernate的核心实现，提供了Hibernate所有的核心功能。

3）hibernate-entitymanager实现了标准的JPA，可以把它看成hibernate-core和JPA之间的适配器，它并不直接提供ORM的功能，而是对hibernate-core进行封装，使得Hibernate符合JPA的规范。

    总的来说，JPA是规范，Hibernate是框架，JPA是持久化规范，而Hibernate实现了JPA。

### 三、JPA 概要

#### 1、概述

JPA在应用中的位置如下图所示：

JPA维护一个Persistence Context（持久化上下文），在持久化上下文中维护实体的生命周期。主要包含三个方面的内容：

1. ORM元数据。JPA支持annotion或xml两种形式描述对象-关系映射。
2. 实体操作API。实现对实体对象的CRUD操作。
3. 查询语言。约定了面向对象的查询语言JPQL（Java Persistence Query Language。

JPA的主要API都定义在javax.persistence包中。如果你熟悉Hibernate，可以很容易做出对应：

| **org.hibernate** | **javax.persistence** | **说明** |
| --- | --- | --- |
| cfg.Configuration | Persistence | 读取配置信息 |
| SessionFactory | EntityManagerFactory | 用于创建会话/实体管理器的工厂类 |
| Session | EntityManager | 提供实体操作API，管理事务，创建查询 |
| Transaction | EntityTransaction | 管理事务 |
| Query | Query | 执行查询 |

#### 2、实体生命周期

实体生命周期是JPA中非常重要的概念，描述了实体对象从创建到受控、从删除到游离的状态变换。对实体的操作主要就是改变实体的状态。

JPA中实体的生命周期如下图：

1. New，新创建的实体对象，没有主键(identity)值
2. Managed，对象处于Persistence Context(持久化上下文）中，被EntityManager管理
3. Detached，对象已经游离到Persistence Context之外，进入Application Domain
4. Removed, 实体对象被删除

#### 3、实体关系映射(ORM)

1)基本映射

|  |  |  |  |
| --- | --- | --- | --- |
| Class | Table | @Entity | @Table(name="tablename") |
| property | column | – | @Column(name = "columnname") |
| property | primary key | @Id | @GeneratedValue 详见ID生成策略 |
| property | NONE | @Transient |  |
| **对象端** | **数据库端** | **annotion** | **可选annotion** |

2)映射关系

JPA定义了one-to-one、one-to-many、many-to-one、many-to-many 4种关系。

对于数据库来说，通常在一个表中记录对另一个表的外键关联；对应到实体对象，持有关联数据的一方称为owning-side，另一方称为inverse-side。

为了编程的方便，我们经常会希望在inverse-side也能引用到owning-side的对象，此时就构建了双向关联关系。 在双向关联中，需要在inverse-side定义mappedBy属性，以指明在owning-side是哪一个属性持有的关联数据。

对关联关系映射的要点如下：

| **关系类型** | **Owning-Side** | **Inverse-Side** |
| --- | --- | --- |
| one-to-one | @OneToOne | @OneToOne(mappedBy="othersideName") |
| one-to-many / many-to-one | @ManyToOne | @OneToMany(mappedBy="xxx") |
| many-to-many | @ManyToMany | @ManyToMany(mappedBy ="xxx") |

其中 many-to-many关系的owning-side可以使用@JoinTable声明自定义关联表，比如Book和Author之间的关联表：

@JoinTable(name = "BOOKAUTHOR", joinColumns = { @JoinColumn(name = "BOOKID", referencedColumnName = "id") }, inverseJoinColumns = { @JoinColumn(name = "AUTHORID", referencedColumnName = "id") })

关联关系还可以定制延迟加载和级联操作的行为（owning-side和inverse-side可以分别设置）：

通过设置fetch=FetchType.LAZY 或 fetch=FetchType.EAGER来决定关联对象是延迟加载或立即加载。

通过设置cascade={options}可以设置级联操作的行为，其中options可以是以下组合：

* CascadeType.MERGE 级联更新
* CascadeType.PERSIST 级联保存
* CascadeType.REFRESH 级联刷新
* CascadeType.REMOVE 级联删除
* CascadeType.ALL 级联上述4种操作

#### 4、事件及监听

通过在实体的方法上标注@PrePersist，@PostPersist等声明即可在事件发生时触发这些方法。

### 四、JPA应用

#### 1、Dependencies

<dependencies>

  <dependency>

    <groupId>org.springframework.data</groupId>

    <artifactId>spring-data-jpa</artifactId>

  </dependency>

<dependencies>

#### 2、JPA提供的接口

主要来看看Spring Data JPA提供的接口，也是Spring Data JPA的核心概念：

1)：Repository：最顶层的接口，是一个空的接口，目的是为了统一所有Repository的类型，且能让组件扫描的时候自动识别。

2)：CrudRepository ：是Repository的子接口，提供CRUD的功能

public interface CrudRepository<T, ID extends Serializable>extends Repository<T, ID> {

<S extends T> S save(S entity);

T findOne(ID primaryKey);

Iterable<T> findAll();

Long count();

void delete(T entity);

 boolean exists(ID primaryKey);

// … more functionality omitted

}

3)：PagingAndSortingRepository：是CrudRepository的子接口，添加分页和排序的功能

public interface PagingAndSortingRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {

 Iterable<T> findAll(Sort sort);

Page<T> findAll(Pageable pageable);

}

4)：JpaRepository：是PagingAndSortingRepository的子接口，增加了一些实用的功能，比如：批量操作等。

5)：JpaSpecificationExecutor：用来做负责查询的接口

public interface JpaSpecificationExecutor<T>{

 T findOne(Specification<T> spec);

List<T> findAll(Specification<T> spec);

Page<T> findAll(Specification<T> spec, Pageable pageable);

List<T> findAll(Specification<T> spec, Sort sort);

long count(Specification<T> spec);

}

6)：Specification：是Spring Data JPA提供的一个查询规范，要做复杂的查询，只需围绕这个规范来设置查询条件即可

#### 3、查询语言

##### 3.1 根据名称判别

|  |  |  |
| --- | --- | --- |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstname,findByFirstnameIs,findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age ⇐ ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| Like | findByFirstnameLike | … where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1(parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1(parameter bound with prepended %) |
| Containing | findByFirstnameContaining | … where x.firstname like ?1(parameter bound wrapped in%) |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | … where x.lastname <> ?1 |
| In | findByAgeIn(Collection<Age> ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection<Age> age) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |
| **Keyword** | **Sample** | **JPQL snippet** |

##### 3.2 @Query

public interface UserRepository extends JpaRepository<User, Long> {

 //Declare query at the query method using @Query

  @Query("select u from User u where u.emailAddress = ?1")

  User findByEmailAddress(String emailAddress);

//Advanced like-expressions in @Query

@Query("select u from User u where u.firstname like %?1")

  List<User> findByFirstnameEndsWith(String firstname);

//Declare a native query at the query method using @Query

@Query(value = "SELECT \* FROM USERS WHERE EMAIL\_ADDRESS = ?1", nativeQuery = true)

  User findByEmailAddress(String emailAddress);

//Declare native count queries for pagination at the query method using @Query

@Query(value = "SELECT \* FROM USERS WHERE LASTNAME = ?1",countQuery = "SELECT count(\*) FROM USERS WHERE LASTNAME = ?1",nativeQuery = true)

  Page<User> findByLastname(String lastname, Pageable pageable);

//Declaring manipulating queries

@Modifying

@Query("update User u set u.firstname = ?1 where u.lastname = ?2")

int setFixedFirstnameFor(String firstname, String lastname);

}

##### 3.3 复杂查询 JpaSpecificationExecutor

Criteria 查询：是一种类型安全和更面向对象的查询

这个接口基本是围绕着Specification接口来定义的， Specification接口中只定义了如下一个方法：

Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder cb);

**Criteria查询**

基本对象的构建

1：通过EntityManager的getCriteriaBuilder或EntityManagerFactory的getCriteriaBuilder方法可以得到CriteriaBuilder对象

2：通过调用CriteriaBuilder的createQuery或createTupleQuery方法可以获得CriteriaQuery的实例

3：通过调用CriteriaQuery的from方法可以获得Root实例

过滤条件

1：过滤条件会被应用到SQL语句的FROM子句中。在criteria 查询中，查询条件通过Predicate或Expression实例应用到CriteriaQuery对象上。

2：这些条件使用 CriteriaQuery .where 方法应用到CriteriaQuery 对象上

3：CriteriaBuilder也作为Predicate实例的工厂，通过调用CriteriaBuilder 的条件方法（ equal，notEqual， gt， ge，lt， le，between，like等）创建Predicate对象。

4：复合的Predicate 语句可以使用CriteriaBuilder的and, or andnot 方法构建

实例：

ImTeacher.java

@Entity

@Table(name = "im\_teacher")

public class ImTeacher implements Serializable{

 private static final long serialVersionUID = 1L;

  @Id

    @GeneratedValue

    @Column(name = "id")

 private int id;

  @Column(name = "teacher\_id")

    private int teacherId;

  @Column(name = "name")

    private String name = "";

 @Column(name = "age")

     private int age;

 @Column(name = "sex")

    private String sex = "";

...

}

ImTeacherDao.java

public interface ImTeacherDao extends PagingAndSortingRepository<ImTeacher, Integer>,JpaSpecificationExecutor{

...

}

@Service

public class ImTeacherDaoService {

 @Autowired

 ImTeacherDao imTeacherDao;

 /\*\*

     \* 复杂查询测试

     \* @param page

     \*/

    public Page<ImTeacher> findBySepc(int page, int size){

        PageRequest pageReq = this.buildPageRequest(page, size);

        Page<ImTeacher> imTeachers = this.imTeacherDao.findAll(new MySpec(), pageReq);

         return imTeachers;

    }

     /\*\*

      \* 建立分页排序请求

      \*/

     private PageRequest buildPageRequest(int page, int size) {

           Sort sort = new Sort(Direction.DESC,"age");

           return new PageRequest(page,size, sort);

     }

     private class MySpec implements Specification<ImTeacher>{

        @Override

        public Predicate toPredicate(Root<ImTeacher> root, CriteriaQuery<?> query, CriteriaBuilder cb) {

        //1.混合条件查询

          Path<String> exp1 = root.get("name");

            Path<String> exp2 = root.get("age");

            query.where(cb.like(exp1, "%王%"),cb.equal(exp2, "45"));

         //2.多表查询

        /\*Join<ImTeacher,ImStudent> join = root.join("imStudent", JoinType.INNER);

            Path<String> exp3 = join.get("name");

            return cb.like(exp3, "%jy%");\*/

       return null;

    }

 }

}

##### 3.4 分页

上个实例的发杂查询已经带有分页，若实例的DAO接口有继承PagingAndSortingRepository接口，则可以直接调用

Page<ImTeacher> impeacher = imTeacherDao.findAll(new PageRequest(1,20));

##### 3.5 联表查询

方法：

法一：直接用Query语句或者上节复杂的连接查询，查出两张或多张表的数据。

法二：映射，接下来将详细介绍。

1）ImStudent.java

@Entity

@Table(name = "im\_student")

public class ImStudent {

 @Id

    @GeneratedValue

    @Column(name = "id")

 private int id;

 @Column(name = "student\_id")

    private int studentId;

 @Column(name = "name")

    private String name = "";

 @Column(name = "age")

    private int age;

 @Column(name = "sex")

    private String sex = "";

 @Column(name = "teacher\_id")

    private int  teacherId;

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

@JoinColumn(name="teacher\_id", referencedColumnName="id", insertable=false, updatable=false)

private ImTeacher imTeacher;

...

}

2）在ImTeacher.java中添加

 @OneToMany(mappedBy="imTeacher",cascade=CascadeType.ALL,fetch=FetchType.LAZY)

    private Set<ImStudent> imStudent = new HashSet<ImStudent>();

...

3）根据学生名字查出其老师信息

@Query("SELECT teacher FROM ImTeacher teacher JOIN teacher.imStudent student WHERE student.name=:name")

 ImTeacher findByStuName(@Param("name") String name);

根据老师名字查出其学生列表

@Query("SELECT student FROM ImStudent student JOIN student.imTeacher teacher WHERE teacher.name = :name")

 Set<ImStudent> findByStudByTeaName(@Param("name") String name);

四、总结

1、Hibernate的DAO层开发比较简单，对于刚接触ORM的人来说，能够简化开发工程，提高开发速度。

2、Hibernate对对象的维护和缓存做的很好，对增删改查的对象的维护要方便。

3、Hibernate数据库移植性比较好。

4、Hibernate功能强大，如果对其熟悉，对其进行一定的封装，那么项目的整个持久层代码会比较简单。

要么有深度，要么有趣，要么安静。