文章目录

概述

JPA 介绍

JPA 官方文档：传送门

JPA（Java Persistence API）是 Java 标准中的一套 ORM 规范（提供了一些编程的 API 接口，具体实现由 ORM 厂商实现，如Hiernate、TopLink 、Eclipselink等都是 JPA 的具体实现），借助 JPA 技术可以通过注解或者 XML 描述【对象-关系表】之间的映射关系，并将实体对象持久化到数据库中（即Object Model与Data Model间的映射）。

JPA 是 Java 持久层API，由 Sun 公司开发，希望规范、简化 Java 对象的持久化工作，整合 ORM 技术，整合第三方 ORM 框架，建立一种标准的方式，目前也是在按照这个方向发展，但是还没能完全实现。在ORM框架中，Hibernate框架做了较好的 JPA 实现，已获得Sun 的兼容认证。

JPA 提供了以下规范：

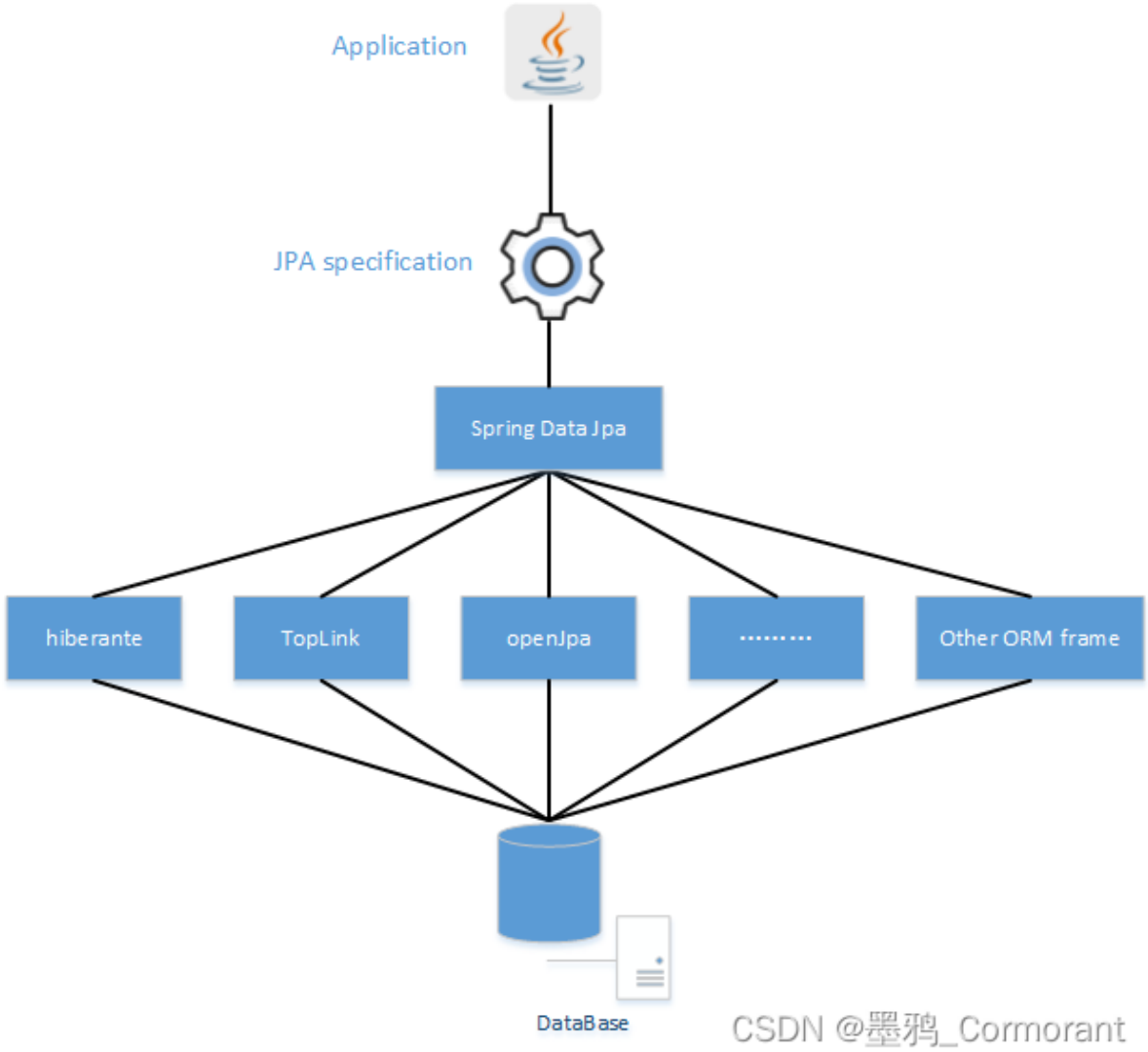
- ORM 映射元数据：JPA 支持 XML 和注解两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中
- JPA 的**Criteria API**：提供 API 来操作实体对象，执行 CRUD 操作，框架会自动将之转换为对应的 SQL，使开发者从繁琐的 JDBC、SQL 中解放出来。
- JPQL 查询语言：通过面向对象而非面向数据库的查询语言查询数据，避免程序的 SQL 语句紧密耦合。

JPA 的优势：

- 开发者面向 JPA 规范的接口，但底层的 JPA 实现可以任意切换：觉得Hibernate好的，可以选择 Hibernate JPA 实现；觉得TopLink 好的，可以选择 TopLink JPA 实现。
- 开发者可以避免为使用 Hibernate 学习一套 ORM 框架，为使用 TopLink 又要再学习一套 ORM 框架。

常用 JPA 实现的介绍

JPA 与 SpringDataJpa、Hibernate 之间的关系



Spring Data JPA 介绍

Spring Data JPA 是在实现了 JPA 规范的基础上封装的一套 JPA 应用框架（Criteria API还是有些复杂）。虽然 ORM 框架都实现了 JPA 规范，但是在不同的 ORM 框架之间切换仍然需要编写不同的代码，而使用 Spring Data JPA 能够方便的在不同的 ORM 框架之间进行切换而不需要更改代码。Spring Data JPA 旨在通过统一 ORM 框架的访问持久层的操作，来提高开发人的效率。

Spring Data JPA 是一个 JPA 数据访问抽象。也就是说 Spring Data JPA 不是一个实现或 JPA 提供的程序，它只是一个抽象层，主要用于减少为各种持久层存储实现数据访问层所需的样板代码量。但是它还是需要 JPA 提供实现程序，其实 Spring Data JPA 底层就是使用的 Hibernate 实现。

Spring Data JPA 其实并不依赖于 Spring 框架。

Spring Data JPA 通过 Repository 来支持上述功能，默认提供的几种 Repository 已经满足了绝大多数需求：

- JpaRepository（为Repository的子接口：JpaRepository -> PagingAndSortingRepository -> CrudRepository -> Repository）
- QueryByExampleExecutor
- JpaSpecificationExecutor
- QuerydslPredicateExecutor

后三者用于更复杂的查询，如动态查询、关联查询等；第一种用得最多，提供基于方法名（query method）的查询，用户可基于第一种继承创建自己的子接口（只要是Repository的子接口即可），并声明各种基于方法名的查询方法。

- Repository 的实现类：
  - SimpleJpaRepository
- QueryDslJpaRepository

Hibernate 介绍

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。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。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。

Hibernate框架（3.2及以上版本）对 JPA 接口规范做了较好的实现，主要是通过以下三个组件来实现的：

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

hibernate 对 JPA 的支持，不是另提供了一套专用于 JPA 的注解。一些重要的注解如 @Column, @OneToMany 等，hibernate 并没有提供，这说明 JPA 的注解已经是 hibernate 的核心，hibernate 只提供了一些补充，而不是两套注解。JPA 和 hibernate 都提供了的注解（例如 @Entity），若 JPA 的注解够用，就直接用，若 JPA 的注解不够用，直接使用 hibernate 的即可。

Querydsl-JPA 介绍

Springdata-JPA 是对 JPA 使用的封装，Querydsl-JPA 也是基于各种 ORM 之上的一个通用查询框架，使用它的 API 类库可以写出“Java代码的sql”，不用去手动接触 sql 语句，表达含义却如 sql 般准确。更重要的一点，它能够构建类型安全的查询，这比起 JPA 使用原生查询时有很大的不同，可以不必再对恶心的“Object[]”进行操作了。SpringDataJPA + Querydsl-JPA 联合使用方案是使用 JPA 操作数据库的最佳方案，它们之间有着完美的相互支持，以达到更高效的编码。

SpringDataJpa 框架使用

官方文档：传送门

基本使用

基本使用步骤及依赖

基本使用步骤

1. 将 spring-data-jpa 包，数据库驱动包等添加为项目依赖；
2. 配置文件定义相应的数据源；
3. 定义 Entity 实体类；
4. 定义自己业务相关的的 JPA repository 接口，继承自 JpaRepository 或者 JpaSpecificationExecutor；
5. 为应用添加注解@EntityScan、@EnableJpaRepositories，此步不是必须的；
6. 将自定义的 JPA repository 接口注入到服务层并使用它们进行相应的增删改查；

依赖

```
    <!-- Spring Boot JPA 依赖 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

     <!-- mysql 驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <!-- pgsql 驱动-->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
    </dependency>
```

Entity 类与 Respository 接口示例

Entity类

```
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
@Table(name="CUSTOMERS")
@Entity
@DynamicInsert
@DynamicUpdate
public class Customer {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "id",insertable = false, updatable = false, length = 32)
    private Integer id;

    @Column(name = "name", nullable = false, length = 10)
    private String name;

    @Column(name = "age")
    private Integer age;

    @Temporal(TemporalType.TIMESTAMP)
    @CreationTimestamp
    @Column(name = "create_date", columnDefinition = "timestamp(6)")
    private Date createDate;

    @Temporal(TemporalType.TIMESTAMP)
    @UpdateTimestamp
    @Column(name = "update_date", columnDefinition = "timestamp(6)")
    private Date updateTime;
}
```

Respository接口

```
// Customer 为该respository对应的实体类，Long为实体类的主键的类型
public interface CustomerRespository extends JpaRespository<Customer, Long>{ }
```

注解扫描及 JPA、JDBC 常用配置

注解扫描

在 SpringBoot 中：

- 默认情况下，当 Entity 类、Repository 类与主类在同一个包下或在主类所在包的子类时，Entity 类、Repository 类会被自动扫描到并注册到 Spring 容器，此时使用者无需任何额外配置。
- 当不在同一包或不在子包下时，需要分别通过在主类上加注解
  - `@EntityScan( basePackages = {"xxx.xxx"})` 来指定 Entity 的位置

    可多处使用 @EntityScan，其 basePackages 可以有交集，但必须覆盖到所有被 Resository 使用到的 Entity，否则会报错。
  - `@EnableJpaRepositories( basePackages = {"xxx.xxx"})` 来指定Repository类的位置

    可多处使用 @EnableJpaRepositories，但它们的 basePackages 不能有交集否则会报重复定义的错（除非配置允许覆盖定义），必须覆盖到所有被使用到的 Resository

JPA 和 JDBC 常用配置

在利用 SpringBoot 框架进行开发时，大部分服务避不开用数据库进行数据存储和使用。SpringBoot 里面一般有两种方式进行数据表的创建和数据存储：

- Spring JDBC，需要在 application.yml 或者 application.properties 中配置 JDBC 相关属性，主要是 spring.datasource.xxx属性配置。当然，使用 jpa 也需要用到 spring.datasource.url/username/password 等属性配置进行数据库地址、用户名、密码等配置。
- Spring Boot JPA，需要在 application.yml 或者 application.properties 中配置 jpa 相关属性 spring.jpa.xxx 属性配置。

配置模板：

```
spring:
  datasource:
    # driver-class-name: com.mysql.jdbc.Driver
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=utf8&serverTimezone=UTC
    username: root
    password: root
    type: com.alibaba.druid.pool.DruidDataSource     # 数据源配置
    schema: classpath:db/schema.sql                  # 建表语句脚本的存放路径
    data: classpath:db/data.sql                      # 数据库初始化数据的存放路径
    sql-script-encoding: UTF-8                       # 设置脚本的编码
  jpa:
    database: mysql              # 配置数据库方言。使用JPA访问数据库时，必需配置。
    hibernate:
      ddl-auto: none             # 每次程序启动时的数据库初始化策略
```

```
    database-platform: org.hibernate.dialect.MySQL5InnoDBDialect    # 配置数据库引擎，不配置则默认为myisam引擎
    show-sql: true              # 日志打印执行的SQL
    properties:
      hibernate:
        # show_sql: true        # 日志打印执行的SQL。与spring.jpa.show-sql配置效果相同，两者使用其一即可。
        format_sql: true        # 格式化sql语句
```

配置说明：

- **spring.datasource.xxx**
  - spring.datasource.**driver-class-name**：配置driver的类名，默认是从 JDBC URL 中自动探测
  - spring.datasource.**url**：配置数据库 JDBC 连接串
  - spring.datasource.**username**：配置数据库连接用户名
  - spring.datasource.**password**：配置数据库连接用户名对应的密码
  - spring.datasource.**type**：连接池配置
  - spring.datasource.**schema**：使用脚本创建表的语句的存放路径

    classpath/db 表示在工程的 resource 层级下的 db 目录中存放
  - spring.datasource.**data**：使用脚本初始化数据库数据的语句的存放路径
  - spring.datasource.**sql-script-encoding**：设置脚本的编码，默认常用设置为UTF-8

  使用上述方式（db 脚本）建表时，spring.jpa.hibernet.ddl-auto 设置成 none，否则可能会被覆盖！
- **spring.jpa.xxx**
  - spring.jpa.hibernet.**ddl-auto** 值说明：
    - **create**： 服务程序重启后，加载 hibernate 时都会删除上一次服务生成的表，然后根据服务程序中的 model（entity）类再重新生成表。**慎用，会导致数据库中原表数据丢失。**
    - **create-drop** ：服务服务程序重启后，加载 hibernate 时根据 model（entity）类生成表，当sessionFactory关闭时，创建的表就自动删除。
    - **update**：默认常用属性，第一次加载 hibernate 时根据 model（entity）类会自动建立表结构，后面服务程序重启时，加载 hibernate 会根据 model（entity）类自动更新表结构，如果表结构改变了，但是表行仍然存在，不会删除以前的行（对于表结构行只增不减）。
    - validate ：服务程序重启后，每次加载 hibernate 时，验证创建数据库表结构，只会和数据库中的表进行比较，如果不同，就会报错。不会创建新表，但是会插入新值。
    - **none** ：什么也不做。
  - spring.jpa.**database**：配置数据库方言，使用JPA访问数据库时，必需配置。
  - spring.jpa.**database-platform**：配置数据库引擎。

    注意：SpringBoot 2.0 后使用 JPA、Hibernate 来操作 MySQL 时，Hibernate 默认使用 MyISM 存储引擎而非InnoDB，前者不支持外键故会忽略外键定义。

使用 JPA 访问数据库的注意事项：

- spring.jpa.database 和 spring.jpa.database-platform 这两项配置至少要配置一个来指明数据库方言
- 访问的是 MySQL 数据库时，spring.datasource.driver-class-name 需配置为 com.mysql.cj.jdbc.Driver

- MySQL jdbc 6.0 版本以上 spring.datasource.url 中地址必须要配置 “serverTimezone”参数

  若使用的时间是北京时区也就是东八区，领先 UTC（全球标准时间）八个小时。url 需指定时区为中国标准时间，即 serverTimezone=Asia/Shangha

Respository 接口核心方法

```
// 添加 or 修改数据
  // 底层逻辑为：当entity的id为null，则直接新增，不为null，则先select，如果数据库存在，则update。如果不存在，则insert
S save(S entity)
// 批量保存
  // 注意：当批量保存大量数据时，效率会很慢！因为 saveAll 本质是循环集合调用save方法。优化方案见 批量保存优化
List<T> saveAll(Iterable<S> list)

// 删除
void delete(T entity)

// 查询所有数据
List<T> findAll()
// 根据id查询
Optional<T> findOne()
// 根据实体类属性查询（需命名方法）
findByProperty (type Property)      // 例如：findByAge(int age)
// 分页查询
Page<S> findAll(Example<S> example, Pageable pageable)
// 计数 查询
long count()
// 根据某个属性的值查询总数
countByAge(int age)
// 是否存在
boolean existsById(ID primaryKey)
```

查询 API

自定义命名查询及查询关键字

介绍、基本使用、解析原理

- **通过方法名来指定查询逻辑，而不需要自己实现查询的 SQL 逻辑**

  示例：`List<Student> getByName(String name)`

- **JPA 集合类型查询参数**

  ```
  List<StudentEntity> getByIdInAndSchoolId(Collection<String> studentIdList, String schoolId);
  ```

  关键在于 In 关键字。参数用 Collection 类型，当然也可以用 List、Set 等，但用 Collection 更通用，因为此时实际调用可以传 List、Set 等实参。

解析原理

- **方法名 解析原理**：

  对方法名中除了保留字（findBy、top、within等）外的部分以 and 为分隔符提取出条件单词，然后解析条件获取各个单词并看是否和 Entity 中的属性对应（不区分大小写进行比较）。

  注意：get/find 与 by之间的会被忽略，所以 `getNameById` 与 `getById` 是等价的，会根据 id 查出整个 Entity 而不会只查 name 字段（指定部分字段的查询见后面条目）。

- **查询条件 解析原理**：

  假设 School 和 Student 是一对多关系，Student 中有个 `School school` 字段、School 有个 `String addressCode` 字段，以如下查询为例：

  ```
  // 查询student表
  Studetn getByNameAndSchoolAddressCode(String studentName, String
  addressCode)
  // JPA会自动生成条件studentName和关联条件student.school.addressCode进行查询

  // 查询student表，推荐写法
  Studetn getByNameAndSchool_AddressCode(String studentName, String
  addressCode)
  ```

  1. 由 And 分割得到 `studentName`、`SchoolAddressCode`；

     分别看 Student中 是否有上述两属性，显然前者有，后者没有，则后者需要进一步解析；

  2. JPA 按驼峰命名格式从后往前尝试分解 `SchoolAddressCode`：

     1. 先得到 `SchoolAdress`、`Code`，由于 Student 没有 `SchoolAddress` 属性，故继续尝试分解，得到 `School`、`AdressCode`
     2. 由于 `Student` 有 `School` 属性且 `School` 有 `addressCode` 属性，故满足，最终得到条件 `student.school.addressCode`

  注：若 `Student` 中有个 `SchoolAdress schoolAddress` 属性，但 `schoolAddress` 中没有 `code` 属性，则会因找不到 `student.schoolAdress.code` 而报错，所以可通过下划线显示指定分割关系，即写成：`getByNameAndSchool_AddressCode`

- **查询字段 解析原理**：

  默认会查出 Entity 的所有字段且返回类型为该 Entity 类型，有两种情况可查询部分字段（除此外都会查出所有字段）：

  1. 通过 @Query 写自定义查询逻辑中只查部分字段。这种不属于直接通过方法名指定查询（详见后面查询指定部分字段的条目）。
  2. 返回类型为自定义接口或该接口列表，接口中仅包含部分字段的get方法，此时会根据接口方法名查询部分字段。示例：

     ```
     /**
      * 此方法在 CourseRepository 中
      * 注：find和By间的部分在解析时会被忽略。但为了见名知意，最好加上字段信息，如
     findVersionByGroupId
      */
     List<MyCustomColumns> findCustomColumnsByGroupId(String groupId);

     /**
      * 封装查询结果的接口
      */
     public interface MyCustomColumns {
         //JPA生成查询语句时只会查下面get方法中指定的字段名。需要确保Entity中有该字段名
     否则会报错
     ```

```
    public String getId();
    public String getVersion();
    public String getGroupId();
}
```

查询关键字

在查询时，通常需要同时根据多个属性进行查询，且查询的条件也格式各样（大于某个值、在某个范围等等），SpringDataJPA 为此提供了一些表达条件查询的关键字，官方文档如下：

keyword	Sample	JPQL snippet
Distinct	findDistinctByLastnameAndFirstname	select distinct … where x.lastname = ?1 and x.firstname = ?2
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, Null	findByAge(Is)Null()	… 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 ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection ages)	… where x.age not in ?1

keyword	Sample	JPQL snippet
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstname) = UPPER(?1)

Example（动态实例）查询

介绍、基本使用

Example 查询翻译过来叫 " 按例查询（QBE）"。是一种用户界面友好的查询技术。 它允许动态创建查询，并且不需要编写包含字段名称的查询。 而且按示例查询不需要使用特定的数据库的查询语言来编写查询语句。

简单来说，就是通过一个例子来查询。例如要查询的是 Customer 对象，查询条件也是一个 Customer 对象，通过一个现有的客户对象作为例子，查询和这个例子相匹配的对象。

```
// 示例：Example对象（使用默认的匹配器）
Example<Customer> ex = Example.of(customer);
// 示例：Example对象，由customer和matcher共同创建
Example<Customer> ex = Example.of(customer, matcher);
```

- **实体对象**：在持久化框架中与 Table 对应的域对象，一个对象代表数据库表中的一条记录。在构建查询条件时，一个实体对象代表的是查询条件中的“数值”部分。

  如：要查询名字是“Dave”的客户，实体对象只能存储条件值“Dave”

- **匹配器**：ExampleMatcher 对象，它是匹配“实体对象”的，表示了如何使用“实体对象”中的“值”进行查询，它代表的是“查询方式”，解释了如何去查的问题。

  例如：要查询 FirstName 是“Dave”的客户，即名以“Dave"开头的客户，该对象就表示了“以什么开头的”这个查询方式。withMatcher(“name”, GenericPropertyMatchers.startsWith())

- **实例**：即 Example 对象，代表的是完整的查询条件。由实体对象（查询条件值）和匹配器（查询方式）共同创建。

优缺点

优势：

- 可以使用动态或者静态的限制去查询
- 在重构实体的时候，不用担心影响到已有的查询
- 可以独立地工作在数据查询 API 之外

劣势：

- 不支持组合查询，比如：firstname = ?0 or (firstname = ?1 and lastname = ?2)
- 只支持字符串的 starts/contains/ends/regex 匹配，对于非字符串的属性，只支持精确匹配。

  换句话说，并不支持大于、小于、between 等匹配。

- 对一个要进行匹配的属性（如：姓名 name），只能传入一个过滤条件值

Example（动态实例）查询的原理

- 从生成的 SQL 语句可以看到，它的判断条件是根据实体的属性来生成查询语句的。

  如果实体的属性是 null，它就会忽略它；

  如果不是，就会取其值作为匹配条件。

- **注意：**如果一个字段是不是包装类型，而是基本类型，它也会参与 where 条件中，其值是默认值。所以在定义实体时，基本数据类型的字段应尽量使用包装类型。

- **使用示例**：

```
@Test
public void test01() {
    User user = User.builder().name("Bob").build();
    Example<User> example = Example.of(user);

    List<User> list = userRepository.findAll(example);
    list.foreach(System.out::println)

    Optional<User> userOptional = userRepository.findOne(Example.of(user));
    userOptional.ifPresent(x ->
System.out.println(x.getName()).isEqualTo("Bob"));
}
```

自定义匹配器规则

ExampleMatcher，不传时会使用默认的匹配器。

```
    @Test
    public void test02() {
        //创建查询条件数据对象
        User user = new User();
        user.setUsername("y");
        user.setAddress("sh");
        user.setPassword("admin");

        //创建匹配器，即如何使用查询条件
        ExampleMatcher matcher = ExampleMatcher.matching()
                //模糊查询匹配开头，即{username}%
//                .withMatcher("username",
ExampleMatcher.GenericPropertyMatchers.startsWith())
                .withMatcher("username", match -> match.startsWith())
                //全部模糊查询，即%{address}%
                .withMatcher("address"
,ExampleMatcher.GenericPropertyMatchers.contains())
                //忽略字段，即不管password是什么值都不加入查询条件
                .withIgnorePaths("password");

        //创建实例
        Example<User> example = Example.of(user ,matcher);

        //查询
        List<User> list = userRepository.findAll(example);
        System.out.println(list);
    }
```

```
/*
打印的sql语句如下：
select
    user0_.id as id1_0_,
    user0_.address as address2_0_,
    user0_.email as email3_0_,
    user0_.password as password4_0_,
    user0_.phone as phone5_0_,
    user0_.username as username6_0_
from
    t_user user0_
where
    (
        user0_.username like ?
    )
    and (
        user0_.address like ?
    )

参数如下：
2018-03-24 13:26:57.425 TRACE 5880 --- [           main]
o.h.type.descriptor.sql.BasicBinder      : binding parameter [1] as [VARCHAR] -
[y%]
2018-03-24 13:26:57.425 TRACE 5880 --- [           main]
o.h.type.descriptor.sql.BasicBinder      : binding parameter [2] as [VARCHAR] -
[%sh%]
*/

    @Test
    public void test03() {
        //创建查询条件数据对象
        Customer customer = new Customer();
        customer.setName("zhang");
        customer.setAddress("河南省");
        customer.setRemark("BB");
        //创建匹配器，即如何使用查询条件
        ExampleMatcher matcher = ExampleMatcher.matching() //构建对象
                .withStringMatcher(StringMatcher.CONTAINING) //改变默认字符串匹配方
式：模糊查询
                .withIgnoreCase(true) //改变默认大小写忽略方式：忽略大小写
                .withMatcher("address", GenericPropertyMatchers.startsWith()) //
地址采用“开始匹配”的方式查询
                .withIgnorePaths("focus");  //忽略属性：是否关注。因为是基本类型，需要忽
略掉
        //创建实例与查询
        List<Customer> ls = dao.findAll(Example.of(customer, matcher));
    }

    @Test
    public void test04() {
        List<Customer> ls = dao.findAll(Example.of(
            new Customer(),
            ExampleMatcher.matching() //构建对象
                .withIncludeNullValues() //改变“Null值处理方式”：包括
                //忽略其他属性

.withIgnorePaths("id","name","sex","age","focus","addTime","remark","customerTyp
e")
```

```
        ));
    }
```

StringMatcher 参数

Matching	生成的语句	说明
DEFAULT (case-sensitive)	firstname = ?0	默认（大小写敏感）
DEFAULT (case-insensitive)	LOWER(firstname) = LOWER(?0)	默认（忽略大小写）
EXACT (case-sensitive)	firstname = ?0	精确匹配（大小写敏感）
EXACT (case-insensitive)	LOWER(firstname) = LOWER(?0)	精确匹配（忽略大小写）
STARTING (case-sensitive)	firstname like ?0 + ‘%’	前缀匹配（大小写敏感）
STARTING (case-insensitive)	LOWER(firstname) like LOWER(?0) + ‘%’	前缀匹配（忽略大小写）
ENDING (case-sensitive)	firstname like ‘%’ + ?0	后缀匹配（大小写敏感）
ENDING (case-insensitive)	LOWER(firstname) like ‘%’ + LOWER(?0)	后缀匹配（忽略大小写）
CONTAINING (case-sensitive)	firstname like ‘%’ + ?0 + ‘%’	模糊查询（大小写敏感）
CONTAINING (case-insensitive)	LOWER(firstname) like ‘%’ + LOWER(?0) + ‘%’	模糊查询（忽略大小写）

说明：

- 在默认情况下（没有调用 withIgnoreCase()）都是大小写敏感的。

总结

- 在使用 springdata jpa 时可以通过 Example 来快速的实现动态查询，同时配合 Pageable 可以实现快速的分页查询功能
- 对于非字符串属性的只能精确匹配，比如想查询在某个时间段内注册的用户信息，就不能通过 Example 来查询

ExampleMatcher 的使用：

- **基本类型的处理**

  如客户 Customer 对象中的年龄 age 是 int 型的，当页面不传入条件值时，它默认是0，是有值的，那是否参与查询呢?

  - 实体对象中，基本数据类型无论是否传值，都会参与查询（因为有默认值），故应避免使用基本数据类型，采用包装器类型（默认值是 null）。

    - 如果已经采用了基本类型，而这个属性查询时若不需要进行过滤，则需把它添加到忽略列表（ignoredPaths）中。
- **Null值的处理**

  当某个条件值为 Null，是应当忽略这个过滤条件呢，还是应当去匹配数据库表中该字段值是 Null 的记录?

    - 当条件值为 null 时，默认是忽略此过滤条件，一般业务也是采用这种方式就可满足。
    - 当需要查询数据库表中属性为 null 的记录时，可将值设为 include，这时，对于不需要参与查询的属性，都必须添加到忽略列表（ignoredPaths）中，否则会出现查不到数据的情况。
- **忽略某些属性值**

  一个实体对象，有许多个属性，是否每个属性都参与过滤？是否可以忽略某些属性?

    - 若属性值为 null，默认忽略该过滤条件;
    - 若属性值为基本数据类型，默认参与查询，若需忽略，则需添加至则需把它添加到忽略列表（ignoredPaths）中。
- **不同的过滤方式**

  同样是作为 String 值，可能"姓名"希望精确匹配，"地址"希望模糊匹配，如何做到?

    - 默认创建匹配器时，字符串采用的是精确匹配、不忽略大小写，可以通过操作方法改变这种默认匹配，以满足大多数查询条件的需要，如将"字符串匹配方式"改为 CONTAINING（包含，模糊匹配），这是比较常用的情况。
    - 对于个别属性需要特定的查询方式，可以通过配置"属性特定查询方式"来满足要求。
- **大小写匹配**

  字符串匹配时，有时可能希望忽略大小写，有时则不忽略，如何做到?

    - 忽略大小的生效与否，是依赖于数据库的。

      例如 MySql 数据库中，默认创建表结构时，字段是已经忽略大小写的，所以这个配置与否，都是忽略的。
    - 如果业务需要严格区分大小写，可以改变数据库表结构属性来实现，具体可百度。

JPQL 与 nativeQuery (原生SQL)查询

介绍、基本使用

JPQL 是专门为 Java 应用程序访问和导航实体实例设计的。Java Presistence Query Language(JPQL)，java 持久性查询语言。它是JPA 规范的重要组成部分，其实它就是一种查询语言，语法类似于 SQL 语法，但是有着本质的区别。

- **JPQL基本语法**

```
select 实体别名.属性名，实体别名.属性名……
from 实体名 [as] 实体别名
where 实体别名.实体属性 op 比较值
```

- 使用 @Query 注解创建查询，将该注解标注在 Repository 的方法上，然后提供一个需要的 JPQL 语句即可，如:

```
@Query("select p from Person p where name like %?1%")
Person findByName(String name);
```

- JPQL 查询时，可以使用 SpEL 表达式：#{#entityName} （取数据库实体名称 ）

  好处是当修改类名后，不需要再单独去修改 JPQL 中的实体类名称

```
@Query("select p from #{#entityName} p where name like %?1%")
Person findByName(String name;
```

SpEL表达式了解：

SpEL（Spring Expression Language），即 Spring 表达式语言。它是一种类似 JSP 的 EL 表达式，但又比后者更为强大有用的表达式语言。SpEL 表达式可以在 spring 容器内实时查询和操作数据，尤其是操作 List 列表型、Array 数组型数据。所以使用SpEL 可以有效缩减代码量，优化代码结构。

@Query注解查询时候，条件查询如何使用占位符：

- 方式1：**？+数字**

  若使用这种方式，则参数列表中参数的入参顺序必须与 @Query 注解当中标注的顺序相同

```
@Query("SELECT s from Student s where s.email=?1 and s.age=?2")
Student findStudentByEmailAndAge(String email , Integer age);
```

- 方式2：**+参数名称**

  这种方式可以自定义参数的名称。需要在参数列表当中用 @Param 注解标注参数名称。

  不用考虑顺序，是根据参数名称进行绑定。

```
@Query("SELECT s from Student s where s.email=:email and s.age=:age")
Student findStudentByEmailAndAge2(@Param("age") Integer age, @Param("email")
String email);
```

nativeQuery（原生SQL查询）

- 应尽可能避免使用 nativeQuery，使得与数据库字段的耦合限制在 Entity 内而不扩散到 Repository 内，更易于维护
- 尽可能避免在 JPQL、nativeQuery 中进行联表查询，而是在 Service 层通过 JPA Specification 进行动态关联查询
- **nativeQuery 返回 Entity**

  使用 nativeQuery 时 SQL 语句查询的字段名若没有取别名，则默认是数据库中的字段名（例如`school_id`），而 API 返回值通常是`schoolId`，可以在SQL里通过`school_id as schoolId`取别名返回。

  然而若查询很多个字段值则得一个个通过 as 取别名，很麻烦，可以直接将返回值指定为数据库表对应的 Entity，不过此法要求查询的是所有字段名，如：

```
// nativeQuery返回类型可以声明为Entity，会自动进行匹配，要求查回与Entitydb中字段对应的所有db中的字段
@Query(value = " select t.* from teacher t where t.school_id=?1 "// 以下为搜索字段
        + "and (?4 is NULL or name like %?4% ) "
        + "order by job_number limit ?2,  x?3 ", nativeQuery = true)
List<TeacherEntity> myGetBySchoolIdOrderByJobNumber(String schoolId, int
startIndex, Integer size,
                                                    String searchName);
```

JPQL 与 SQL 的区别

- JPQL 是面向对象的查询语言，因此它可以完全理解继承、多态和关联等特征。而且 JPQL 内置了大量函数，极大地方便了 JPQL 查询的功能。当然 JPQL 底层依然是基于 SQL 的，但 JPQL 到 SQL 的转换无须开发者关心，JPQL解析器会负责完成这种转换，并负责执行这种转换的SQL语句来更新数据库。
- SQL 是面向关系数据库的查询语言，因此 SQL 操作的对象是数据表、数据列；

  而 JQPL 操作的对象是实体对象，对象属性。
- 代码对比

```
// 原生的SQL语句。对t_user table表执行查询，查询name、age、user_id三个数据列
select name,age,user_id from t_user

// 面向对象的JPQL语句。对User实体执行查询，查询的是User实体的name、age、userId 属性
select name,age,userId from User
```

- 其他比较项

比较项	SQL	JPQL
面向	处理关系数据	处理JPA实体
关联实体的方式	内连接、外连接、左连接、右连接	内连接和左外连接

| 支持的操作 | 增（Insert）、删（Delete）
改（Update）、查（Select） | Delete（remove）
Update（merge）、Select（find） |

排序查询、分页查询

排序查询

- 静态方式：直接在方法体现（如 getByNameOrderByIdDesc），也可以在 JPQL 的 @Query 的逻辑中使用 `order by` 进行排序
- 动态方式：可以在 Repository 的方法的最后加一个 Sort 或者 Pageable 类型的参数，便可动态生成排序或分页语句（编译后会自动在语句后加 `order by` 或 `limit` 语句）

```
// Repository 中定义方式
List<User> findByAndSort(String name, Sort sort);

// 调用
UserRepository.findByAndSort("bolton", Sort.by(Direction.Desc, "age"));
```

了解：通过 JpaSort.unsafe 实现待 function（函数计算） 的 sort（排序）：

```
// Repository 中定义方式
public interface UserRepository extends JpaRepository<User, Long> {

   @Query("select u from User u where u.lastname like ?1%")
  List<User> findByAndSort(String lastname, Sort sort);
```

```
   @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where
u.lastname like ?1%")
  List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);
}


// 调用
userRepository.findByAndSort("lannister", new Sort("firstname"));

//userRepository.findByAndSort("stark", new Sort("LENGTH(firstname)"));   //
报错：invalid 无效
userRepository.findByAndSort("targaryen",
JpaSort.unsafe("LENGTH(firstname)"));
```

分页查询

- 动态方式：在 Repository 的方法的最后加一个 Pageable 类型的参数，便可动态生成分页语句（编译后会自动在语句后加 limit 语句）

```
// 不写@Query语句也可以加Pageable。另外若这里声明为List则不会分页，总是返回所有数据
@Query("select se from StudentExperimentEntity se "
       + "where se.studentId= ?2 and se.experimentId in "
       + "( select e.id from ExperimentEntity e where e.courseId= ?1 ) ")
List<StudentExperimentEntity> myGetByCourseIdAndStudentId(String courseId,
String studentId,
                                                          Pageable
pageable);


/**
 * 调用
 * 编译后会在myGetByCourseIdAndStudentId所写SQL后自动加上 order by
studentexp0_.lastopertime desc limit ?
 */
repository.myGetByCourseIdAndStudentId(courseId, studentId,
                                       PageRequest.of(0, 10,
newSort(Sort.Direction.DESC, "lastopertime")));
```

注：上述用法也支持 **nativeQuery**，示例：

```
@Query(value = "select d.*, u.username from developer d inner join user u on
d.id=u.id "
       + " where (?1 is null or d.nick_name like %?1% ) ", nativeQuery =
true)
List<DeveloperEntity> myGetByNicknameOrPhoneOrEmailOrBz(String
searchNicknameOrPhoneOrEmailOrBz,
                                                        Pageable pageable);

// 如果要同时返回分页对象，则可用Page<XX>返回类型
Page<DeveloperEntity> myGetByNicknameOrPhoneOrEmailOrBz(String
searchNicknameOrPhoneOrEmailOrBz,
                                                        Pageable pageable);
```

需要注意的是，**只有元素是 Entity 类型时才支持直接将返回值声明为Page对象**，否则会报错：Convert Exception。

自定义封装查询的结果集（投影）

介绍

查询一个表的部分字段，称为投影（Projection）

对于只返回一个字段的查询，方法返回类型直接声明为该字段的类型或类型列表

```
@Query(value = "select p.name from PserSon p where p.id=?1")
String findNameById(String id);

@Query(value = "select name from PserSon where age=?1")
Set<String> findNameByAge(String id);
```

对于返回多个字段的查询：

- 如果是查询**所有字段**，则使用 Entity 实体类接收查询的结果集，支持 JPA，JPQL，原生 sql 查询。
- 如果是查询**部分字段**，自定义查询的结果集有3种方法：
  - 使用自定义接口来映射结果集，支持 JPA，JPQL，原生sql查询。
  - 使用自定义对象来接收结果集，支持 JPQL查询。
  - 使用 `List<Object[]>` 或 `Map<String, Object>` 来接收结果集，只支持原生 sql 查询。

使用自定义接口来映射结果集

直接通过方法名命名指定返回对象为包含部分字段 getter 方法的自定义接口（interface），只需要定义属性的 getter 方法，jdk 动态代理封装数据。

注意：**如果不用 @Query 则需要确保接口中 getter 方法名中的字段与 Entity 中的一致，而如果使用 @Query 则不需要，因为可以通过 as 取别名**

示例：

- Repository自定义查询方法

```
List<IdAndLanguageType> getLanguagesTypeByCourseIdIn(Collection<String>
courseIdCollection);
```

- 自定义接口来映射结果

```
// 示例1
public interface IdAndLanguageType {
    String getIdx();
    String getLanguageType();

    default String getAll() {
        return getIdx() + ", " + getLanguageType();
    }
}
```

```
// 示例2
public interface PersonSummary {
    String getFirstname();
    String getLastname();
    List<AddressSummary> getAddress();

    public interface AddressSummary {
        String getCity();
    }
}
```

```
/**
 * 投影接口中的 getter 可以使用可为空的包装器来提高空安全性。
 * 如果基础投影值不是null，则使用包装器类型的当前表示返回值。如果支持值是null，则 getter 方法返回所用包装器类型的空表示。
 * 当前支持的包装器类型有：
 * java.util.Optional、com.google.common.base.Optional、scala.Option、io.vavr.control.Option
 */
public interface NamesOnly {
  Optional<String> getFirstname();
}
```

内部原理：

1. 根据自定义接口中的 getter 方法解析出要查询的字段：idx、languageType
2. JPA 内部转成了用 @Query 注解的查询：

```
// 注意：这里第一个字段名为 idx 而不是 id ，因为是根据接口getter方法产生的。
@Query("select new map(idx as idx, languageType as languageType) from CourseEntity where id in ?1 ")
```

使用自定义对象来接收结果集

返回对象为自定义的 class，可定义 toString 方法，打印日志方便。

自定义的 class 须包含查询的字段的属性，且要封装的字段由公开的构造方法确定，对象属性不要求名称一致，只需要构造方法参数位置正确。

JPQL 语法须为：select new + 对象全限定类名

```
/**
 * Repository自定义查询方法
 */
@Query(select new com.xx.yy.PersonResult(p.id,p.name,p.age) from Person p)
List<PersonResult> findPersonResult();


/**
 * 自定义class类来映射结果
 * 自定义class类中属性若全为JPQL中的查询字段且顺序一致，使用@AllArgsConstructor全参构造方法即可，否则需手写相应构造方法。
 */
@Data
```

```
@AllArgsConstructor
@NoArgsConstructor
public class IdAndLanguageType {
    String id;
    String name;
    String age;
}
```

使用 Map<String, Object> 或 List<Object[]>来接收结果集

注意：指定为 Map 时，实际类型是 `org.springframework.data.jpa.repository.query.AbstractJpaQuery$TupleConverter$TupleBackedMap`，该类型只能读不能改或写

- **nativeQuery 查询**，即 原生 SQL 查询

  直接 select 部分字段即可，**结果集默认会自动包装为 Map。**

  缺点是 sql 里用的直接是数据库字段名，导致耦合大，数据库字段名一变，所有相关 sql 都得相应改变。

  ```
  /**
   * Repository自定义查询方法
   */
  @Query(value = "select g.id, g.school_id as schoolId, g.name from grade g "
         + "left join student s on g.name=s.grade "
         + " where g.school_id=(select a.school_id from admin a where a.id=?
  1)"
         + " and (?4 is null or g.name like %?4% or g.bz like %?4% ) "
         + " group by g.id limit ?2,?3", nativeQuery = true)
  List<Map<String, Object>> myGetGradeList(String adminId, Integer page,
  Integer size,
                                           String searchGradeNameOrGradeBz);
  ```

- **JPQL 查询**

  可以手动指定包装为 map，此时 map 的 key 为字段序号，故最通过 as 指定 key 为字段名。

  默认会将结果包装为 List 而不是 Map，可以手动指定包装为 map，此时 map 的 key 为字段序号（0、1、2...），也可以通过as 指定 key 为字段名。

  注意：由于声明为 Map 时并不知道数据的返回类型是什么，故默认会用最大的类型（例如对于数据库中的整型列，查出时 Map中该字段的类型为 BigInteger）

  ```
  /**
   * Repository自定义查询方法
   * 注意是 'map'，不是 jdk 中的 'Map' !
   */
  @Query("select new map(g.name as name, count(s.id) as stuCount) from
  GradeEntity g, StudentEntity s "
         + "where g.name=s.grade and g.schoolId=?1 group by g.id")
  List<Map<String, Object>> myGetBySchoolId(String schoolId);

  @Query("select new map(g.name as name, count(s.id) as stuCount) from
  GradeEntity g, StudentEntity s "
         + "where g.name=s.grade and g.schoolId=?1 group by g.id")
  List<Object[]> myGetBySchoolId(String schoolId);
  ```

动态投影

```
/**
 * Repository自定义查询方法
 * 动态投影方式。泛型根据需要传入Entity实体类或封装部分字段的自定义接口
 */
public interface PersonRepository extends Repository<Person, UUID> {
  <T> Collection<T> findByLastname(String lastname, Class<T> type);
}
```

count 查询、In 查询

count查询

```
Integer countByName(String name);
```

In查询

不管是否是 @Query 都可以用 in 查询，如：

```
@Query( "select * from student where id in ?1", nativeQuery=true)
//@Query( "select s from StudentEntity s where s.id in ?1")
List<StudentEntity> myGetByIdIn(Collection<String> studentIds );    //复杂查询，自定义查询逻辑

List<StudentEntity> getByIdIn( Collection<String> studentIds );     //简单查询，声明语句即可
```

- 不管是否自己写查询语句、是否是 nativeQuery，都要求调用该方法时所传的 id 列表必须**至少有一个元素，否则执行时会报错。**
- 原因：运行时动态生成 sql 语句，如果 id 列表为 null 或空列表，则最终生成的 sql 语句中"where id in null"不符合sql语法。

联表查询

- Entity 内未定义关联实体时的联表查询，示例：

  ```
  @Query("select cd from CourseDeveloperEntity cd join Developer d where d.nickName='stdeveloper'")
  ```

- Entity 内定义的关联实体的关联查询，示例：

```
@Query("select cd, d from CourseDeveloperEntity cd join cd.developer d where d.nickName='stdeveloper'")

 || (等价于)

@Query("select cd, cd.developer from CourseDeveloperEntity cd where cd.developer.nickName='stdeveloper'")
```

- 若将一个对象的关联对象指定为延迟加载 LAZY，则每次通过该对象访问关联对象时（如 courseDeveloper.developer）都会执行一次SQL 来查出被关联对象，显然如果被关联对象访问频繁则此时性能差。

  解决：

  - 法1：改为 EAGER 加载；
  - 法2：使用 `join fetch` 查询，其会立即查出被关联对象。示例：

    ```
    @Query("select cd from CourseDeveloperEntity cd join fetch cd.developer where cd.id='80'")
    ```

    `join Fetch` 其实就是使用 `inner join`，可以显示指定用其他关联方式，例如 `left join fetch`

    `join fetch` 的缺点之一在于有可能导致“Duplicate Data and Huge Joins”，例如多个实验关联同一课程，则查询两个实验时都关联查出所属课程，后者重复查询。

延迟加载与立即加载（FetchType）

默认情况下，@OneToOne、@ManyToOne 是 LAZY，@OneToMany、@ManyToMany 是 EAGER。但不绝对，看具体需要。

- **FetchType.LAZY**：延迟加载。在查询实体 A 时，不查询出关联实体 B，在调用 getxxx 方法时，才加载关联实体。

  但是注意，查询实体 A 时和 getxxx 必须在同一个 Transaction 中，不然会报错：“no session”，即会表现为两次单独的 SQL 查询（非联表查询）

- **FetchType.EAGER**：立即加载。在查询实体 A 时，也查询出关联的实体 B。即会表现为一次查询且是联表查询

有两个地方用到延迟加载：relationship（@OneToMany等）、attribute（@Basic）。后者一般少用，除非非常确定字段很少访问到。

增删改API

保存 与 更新

- **添加 or 修改数据**

  Repository 方法核心方法

  ```
  S save(S entity)
  ```

  底层逻辑为：当 entity 的 id 为 null，则直接新增，不为 null，则先 select，如果数据库存在，则 update。如果不存在，则 insert

注意：

- **若 JPA 启用了逻辑删除（软删除）功能，使用 save 方法则可能会出现 主键冲突 或 唯一索引冲突 等问题**

  原因：若数据库启用了逻辑删除功能，记录逻辑删除后，该条记录实际仍存在于数据库中，但是 JPA 根据 Entity 的主键查询数据库判断该执行 insert 还是 update 时，查询语句会自动加上逻辑删除的判断条件，从而查不到数据而最终执行 insert，进而可能会导致报主键冲突或唯一索引冲突。

- **update 不支持直接通过方法名声明**

  进行 update 操作方式：

  - 方式1：通过 Repository 的 save 方法
  - 方式2：通过 Repository 中注解 @Query、@Modifying 组合自定义方法

删除 与 逻辑删除

删除记录

- **Repository 接口核心方法**

```
void delete(T entity)
```

- **Repository 自定义删除方法**

  需要加 @Modefying、@Transactional 注解

```
@Transactional  //也可以只标注在上层调用者方法上
@Modifying
@Query("delete from EngineerServices es where es.engineerId = ?1")//update与此类似
int deleteByEgId(String engineerId);

// 直接通过方法名声明（注：update不支持这样写）
@Transactional
@Modifying
int deleteByEgId(String engineerId);
```

注：

- JPA 中非 nativeQuery 的删除操作（如 deleteByName）内部实际上是先分析出方法名中的条件，接着按该条件查询出所有 Entity，然后根据这些 Entity 的 id 执行 SQL 删除操作。

  也正因为这样，软删除功能中指定 @SQLDelete("update student set is_delete='Y' where id=? "), 即可对所有非 nativeQuery 起作用。

- 方法名包含条件的删除操作（例如 `Integer deleteByNameAndSId(String name, String uuid);` ），其执行时与 save 类似，也是先根据条件查出目标 Entity 再执行删除操作。
- `void delete(T entity)` **直接根据 Entity 的主键操作而不用先查。**

逻辑删除

使用**org.hibernate.annotations（不是JPA的标准）**的 **@Where、@SQLDelete、@SQLDeleteALL** 三个注解来实现。

```
// 对非nativeQuery 的delete起作用，包括形如deleteByName等，下同。
@SQLDelete(sql = "update " + StudentEntity.tableName
           + " set " + constant.ISDELETE_COLUMN_NAME + " =true where sid=?")
@SQLDeleteAll(sql = "update " + StudentEntity.tableName +
              " set " + constant.ISDELETE_COLUMN_NAME + " =true where sid=?")
// 对非nativeQuery的select起作用(如count、非nativeQuery的String myGetNameByName等，前者本质上也是select)
@Where(clause = constant.ISDELETE_COLUMN_NAME + " = false")
@Data
@Entity
@Table(name = StudentEntity.tableName)
public class StudentEntity extends BaseEntity {

    public static final String tableName = "student";
    ...

    @Column(name = constant.ISDELETE_COLUMN_NAME, nullable = false)
    private Boolean isDelete = false;

}
```

需要注意的是：

- @Where 会自动在查询语句后拼接 @Where 中指定的条件；该注解对所有的非 nativeQuery 的查询起作用，如 count、自己写的非 nativeQuery 的查询语句（例如：myGetByName）等。
- @SQLDelete 会自动将删除语句替换为 @SQLDelete 中指定的 sql 操作；

  该注解对所有非 nativeQuery 的删除操作起作用，如delete(StudenEntity entity)、deleteBySId、deleteByName 等，但由于指定的 sql 操作中条件不得不写死，所以要按期望起作用的话，**@SQLDelete 中的 sql 操作应以 Entity 的主键为条件，且自定义的删除方法必须按 delete(StudenEntity entity)、deleteBySId 两种写法写**，而不能用 deleteByName（会将 name 参数值传给 sid
- 通过 JPQL 的方法名指定删除操作（如 Integer deleteByName(String name)）时背后是先根据条件查出 Entity ，然后根据Entity 的主键删除该 Entity。所以通过 @SQLDelete、@SQLDeleteALL 实现逻辑删除时，由于其语句是写死的，故：
  - @SQLDelete、@SQLDeleteALL 同时存在时会按后者来执行软删除逻辑
  - @SQLDeleteALL 并不会批量执行软删除逻辑（因为一来不知具体有几个数据，二来 in 中只指定了一个元素），而是一个个删，即有多条待删除时也会一条条执行软删除逻辑，每条语句中 in 中只有一个元素。故其效果与 @SQLDelete 的一样，然而 “in” 操作效率比 “=” 低，故**推荐使用@SQLDelete**
- 关于软删除：

  对于关联表（一对一、一对多、多对多），若要启用软删除，则须为多对多关联表定义额外的主键字段而不能使用联合外键作为主键，否则软删除场景下删除关联关系再重新关联时会主键冲突。

  另外，特殊情况下多对多关联表可以不启用软删除（被关联表、一对多或多对一关联表则需要，因为它们侧重的信息往往不在于关联关系而是重要的业务信息）

批量保存优化

原生的 saveAll() 方法可以保证程序的正确性，但是如果数据量比较大时效率低。

源码逻辑原理是：for 循环集合调用 save 方法；save 方法逻辑为，当 entity 的 id 为 null，则直接新增，不为 null，则先 select，如果数据库存在，则 update。如果不存在，则 insert。

解决方案：

- **批量插入**

  优化方案为：当保存大量数据时，直接使用 `em` 进行持久化插入，省了一步查询操作。

  并且考虑到如果最后才提交所有数据，数据库的负载可能会比较大，故每 1000 条记录就提交（flush）一次。

```
    @Autowired
    private EntityManager entityManager;

    private final int BATCH_SIZE = 1000;

    @Transactional(rollbackFor = Exception.class)
    public void addBatch(List<S> list) {
        int num = 0;

        for (S s : list) {
            entityManager.persist(s);    // insert插入操作（变成托管状态）
            int num += 1;

            if (i % BATCH_SIZE == 0) {
                entityManager.flush(); // 变成持久化状态
                entityManager.clear(); // 变成游离状态
            }
        }
    }
```

- **批量更新**

  在确保数据已经存在的情况下，如果是批量更新可以如下代码代替上面的 `entityManager.persist(projectApplyDO)` 语句：

```
entityManager.merge(projectApplyDO);     //update更新操作
```

自动提交问题

JPA 事务内 Entity 变更会自动更新到数据库

若启用了事务，则对于 managed 状态的 entity，若在事务内该 entity 有字段的值发生了变化，则即使未调 save 方法，该 entity 的变化最后也会被自动同步到数据库，即 sql update 操作。即相当于在 Persist Context flush 时自动对各 engity 执行 save 方法。
（org.hibernate.event.internal.AbstractFlushingEventListener中）

详情可参阅：Spring Data JPA的自动更新，为什么会自动更新？如何避免自动更新？

实现自定义的 Repository 实现类

1. 写一个与 Repository 接口同名的类，加上后缀 Impl，标注 @Repository 注解；这个类不需要实现任何接口，可以自动被扫描到。
2. 在 Repository 接口中加入自定义的方法，比如：

```
public interface MyRepository extends JpaRespository<UserEntity, String>{
    // 自定义的方法
    public Page<Object[]> getByCondition(UserQueryModel u);
}
```

3. 在实现类中，去实现在Repository接口中加入的自定义方法，会被自动找到

```
@Repository
public class MyRepositoryImpl{
    @Autowired
    private EntityManager em;

    // 实现在Repository接口中加入的自定义方法
    public Page<Object[]> getByCondition(UserQueryModel u){
        String hql = "select o.uuid,o.name from UserEntity o where 1=1 and
o.uuid=:uuid";
        Query q = em.createQuery(hql);
        q.setParameter("uuid", u.getUuid());
        q.setFirstResult(0);
        q.setMaxResults(1);
        Page<Object[]> page = new PageImpl<Object[]>(q.getResultList(), new
PageRequest(0,1), 3);
        return page;
    }
}
```

Repository 方式调用存储过程

Repository 方式调用存储过程需要基于 Entity 实体类，在实体类上使用 @NamedStoredProcedureQuery 注解（**需要数据库中有对应的表**，可自动映射结果集）。详见 `Hibernate EntityManger专题-JPA调用存储过程` 条目。

```
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.query.Procedure;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;
import com.labofjet.entity.A;
import com.labofjet.entity.APK;

@Repository
public interface ARepository extends JpaRepository<A, APK>{

    // 方式1。若用这种方式，方法名要与存储过程名一样。【推荐】
    @Procedure
    Integer plus1inout(Integer arg);

    @Procedure
    Object[] mytest();

    // 方式2。Procedure的name为实体类上@NamedStoredProcedureQuery注解中name的值
    @Procedure(name="User.plus1")
```

```
    Integer alias2(@Param("arg")Integer argAlias);  // @Param必须匹配
@StoredProcedureParameter注释的name参数

    // 方式3。Procedure的procedureName参数必须匹配实体类上@NamedStoredProcedureQuery的
procedureName的值
    @Procedure(procedureName="plus1inout")
    Integer alias3(Integer arg);
}
```

注意：返回类型必须匹配。in_only 类型的存储过程返回是 void，in_and_out 类型的存储过程返回相应数据类型

JpaSpecificationExecutor 接口

spring data jpa 提供了 JpaSpecificationExecutor 接口，只要简单实现 toPredicate 方法就可以实现复杂的动态查询。

Specification 是 Spring 对 Criteria 的封装。

JpaSpecificationExecutor 提供了以下接口

```
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);
}


//其中Specification就是需要传入查询方法的参数，它是一个接口
public interface Specification<T> {

    Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder
cb);
    // root：根参数，代表了可以查询和操作的实体对象的根，如果将实体对象比喻成表名，那root里面
就是这张表里面的字段，是JPQL的实体字段，通过Path<Y>get(String var0)来获得操作的字段
    // criteriaQuery：代表一个specific的顶层查询对象，它包含着查询的各个部分，如：select、
form、where、group by、order by 等，它提供了查询的的方法，常用的有 where、select、having
    // criteriaBuilder：用来构建CriteriaQuery的构建器对象，其实就相当于条件或条件组合
}
```

提供唯一的一个方法 toPredicate，只要按照 JPA 2.0 criteria api 写好查询条件就可以了。

关于 JPA 2.0 criteria api 的介绍和使用，可以参考：

- JPA criteria 查询:类型安全与面向对象
- 详解JPA 2.0动态查询机制:Criteria API

Repository 继承 JpaSpecificationExecutor接口

```
public interface TaskResposity extends JpaRespository<Task, Long>,
JpaSpecificationExecutor<Task>{}
```

调用

```
@Service
public class TaskService {

    @Autowired
    private TaskRepository taskRepository ;

    /**
     * 多条件 + 分页排序 查询
     */
    public Page<Task> findBySepc(Task task, int page, int size){
        // 分页排序请求
        PageRequest pageReq = new PageRequest(page, size, new
Sort(Direction.DESC,"createTime"));

        Page<Task> tasks = taskRepository.findAll(new Specification<Task>(){
            // 匿名内部类
            @Override
            public Predicate toPredicate(Root<Task> root, CriteriaQuery<?>
query, CriteriaBuilder builder) {
                //1.混合条件查询
                Path<String> exp1 = root.get("taskName");
                Path<Date>  exp2 = root.get("createTime");
                Path<String> exp3 = root.get("taskDetail");
                Predicate pre = builder.and(
                    builder.like(exp1, "%" + task.getTaskName + "%"),
                    builder.lessThan(exp2, new Date()));
                return builder.or(pre, builder.equal(exp3, task.getTaskDetail));

                /* 生成的sql语句为:
                Hibernate:
                    select
                        count(task0_.id) as col_0_0_
                    from
                        tb_task task0_
                    where
                        (
                            task0_.task_name like ?
                        )
                        and task0_.create_time<?
                        or task0_.task_detail=?
                */


                //2.多表查询
                Join<Task,Project> join = root.join("project", JoinType.INNER);
                Path<String> exp4 = join.get("projectName");
                return cb.like(exp4, "%"+ task.getProjectName +"%");

                /* 生成的sql语句为:
                Hibernate:
```

```
                    select
                        count(task0_.id) as col_0_0_
                    from
                        tb_task task0_
                    inner join
                        tb_project project1_
                            on task0_.project_id=project1_.id
                    where
                        project1_.project_name like ?
                */
        }, pageReq);
        return tasks;
    }

    // 多条件 + 排序 查询
    // 将多个的条件封装成数组的形式传递给接收多个参数的方法完成多条件查询
    public List<Task> findBySepc(Task task){
        /*Specification<UserEntity> spc = (root, query, builder)->{
            ArrayList<Predicate> list = new ArrayList<>();
            list.add(builder.equal(root.get("username"), task.getUsername));
            list.add(builder.equal(xrootget("password"), task.getPassword));
            return builder.and(list.toArray(new Predicate[list.size()]));
        };*/

        Specification<UserEntity> spc = (root, query, builder) -> builder.and(
            builder.equal(root.get("username"), task.getUsername),
            builder.equal(xrootget("password"), task.getPassword)
        );

        List<UserEntity> list = userRepository.findAll(spc,
Sort.by(Sort.Direction.DESC,"createTime"));

        list.forEach(System.out::println);

        return list;
    }
```

每次都要写一个类来实现 Specification 比较麻烦，可以将查询条件封装在专门的一个类中，使用时调用静态方法

```
public class TaskSpec {
 // 封装查询条件的静态方法
    public static Specification<Task> method1(Task task){
        return new Specification<Task>(){
            @Override
            public Predicate toPredicate(Root<Task> root, CriteriaQuery<?>
query, CriteriaBuilder cb) {
                // 示例，未写具体的查询条件，入参从 task 中获取
                return null;
            }
        };
    }
}

// 使用
Page<Task> tasks = this.taskDao.findAll(TaskSpec.method1(), pageReq);
```

拓展了解

获取数据库的类型

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

@Component
@Slf4j
public class DataSourceUtil {

    @Autowired
    private DataSource dataSource;

    // 数据库类型名称
    private static String databaseProductName;

    /**
     * 初始化静态成员变量
     */
    @PostConstruct
    public void init(){
        try(Connection connection = dataSource.getConnection()) {
            databaseProductName =
connection.getMetaData().getDatabaseProductName();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    /**
     * 获取数据库类型名称
     */
    public static String getDatabaseProductName(){
        return databaseProductName;
    }
}
```

Jpa 表名大小写转换、字段名规避数据库关键字

在 linux 下，mysql 的表名是区分大小写的，如果不能通过修改 mysql 配置取消表名区分大小写，则可以通过在 Hibernate 将转化的 SQL 语句发送给数据库执行之前转换大小写。

如果数据库字段名与数据库关键字（保留字）同名导致 sql 语句执行失败，也可通过该自定义类处理

```
import com.duran.ssmtest.utils.DataSourceUtil;
```

```
import org.apache.commons.lang3.StringUtils;
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl;
import org.hibernate.cfg.ImprovedNamingStrategy;
import org.hibernate.engine.jdbc.env.spi.JdbcEnvironment;
import org.springframework.stereotype.Component;

import java.util.Arrays;

@Component
public class MyPhysicalNamingStrategyStandardImpl extends
PhysicalNamingStrategyStandardImpl {
    private static final long serialVersionUID = 1L;

    // mysql关键字列表
    private static final String mysqlKey =
"ADD,ALL,ALTER,ANALYZE,AND,AS,ASC,ASENSITIVE" +

 ",BEFORE,BETWEEN,BIGINT,BINARY,BLOB,BOTH,BY,CALL,CASCADE,CASE,CHANGE,CHAR,CHARA
CTER" +

 ",CHECK,COLLATE,COLUMN,CONDITION,CONNECTION,CONSTRAINT,CONTINUE,CONVERT,CREATE,
CROSS" +

 ",CURRENT_DATE,CURRENT_TIME,CURRENT_TIMESTAMP,CURRENT_USER,CURSOR,DATABASE,DATA
BASES" +

 ",DAY_HOUR,DAY_MICROSECOND,DAY_MINUTE,DAY_SECOND,DEC,DECIMAL,DECLARE,DEFAULT,DE
LAYED" +

 ",DELETE,DESC,DESCRIBE,DETERMINISTIC,DISTINCT,DISTINCTROW,DIV,DOUBLE,DROP,DUAL,
EACH,ELSE" +

 ",ELSEIF,ENCLOSED,ESCAPED,EXISTS,EXIT,EXPLAIN,FALSE,FETCH,FLOAT,FLOAT4,FLOAT8,F
OR,FORCE" +

 ",FOREIGN,FROM,FULLTEXT,GOTO,GRANT,GROUP,HAVING,HIGH_PRIORITY,HOUR_MICROSECOND,
HOUR_MINUTE" +

 ",HOUR_SECOND,IF,IGNORE,IN,INDEX,INFILE,INNER,INOUT,INSENSITIVE,INSERT,INT,INT1
,INT2,INT3" +

 ",INT4,INT8,INTEGER,INTERVAL,INTO,IS,ITERATE,JOIN,KEY,KEYS,KILL,LABEL,LEADING,L
EAVE,LEFT" +

 ",LIKE,LIMIT,LINEAR,LINES,LOAD,LOCALTIME,LOCALTIMESTAMP,LOCK,LONG,LONGBLOB,LONG
TEXT,LOOP" +

 ",LOW_PRIORITY,MATCH,MEDIUMBLOB,MEDIUMINT,MEDIUMTEXT,MIDDLEINT,MINUTE_MICROSECO
ND,MINUTE_SECOND" +

 ",MOD,MODIFIES,NATURAL,NOT,NO_WRITE_TO_BINLOG,NULL,NUMERIC,ON,OPTIMIZE,OPTION,O
PTIONALLY,OR" +

 ",ORDER,OUT,OUTER,OUTFILE,PRECISION,PRIMARY,PROCEDURE,PURGE,RAID0,RANGE,READ,RE
ADS,REAL" +
```

```
 ",REFERENCES,REGEXP,RELEASE,RENAME,REPEAT,REPLACE,REQUIRE,RESTRICT,RETURN,REVOK
E,RIGHT,RLIKE" +

 ",SCHEMA,SCHEMAS,SECOND_MICROSECOND,SELECT,SENSITIVE,SEPARATOR,SET,SHOW,SMALLIN
T,SPATIAL" +

 ",SPECIFIC,SQL,SQLEXCEPTION,SQLSTATE,SQLWARNING,SQL_BIG_RESULT,SQL_CALC_FOUND_R
OWS" +

 ",SQL_SMALL_RESULT,SSL,STARTING,STRAIGHT_JOIN,TABLE,TERMINATED,THEN,TINYBLOB,TI
NYINT" +

 ",TINYTEXT,TO,TRAILING,TRIGGER,TRUE,UNDO,UNION,UNIQUE,UNLOCK,UNSIGNED,UPDATE,US
AGE,USE" +

 ",USING,UTC_DATE,UTC_TIME,UTC_TIMESTAMP,VALUES,VARBINARY,VARCHAR,VARCHARACTER,V
ARYING,WHEN" +
        ",WHERE,WHILE,WITH,WRITE,X509,XOR,YEAR_MONTH,ZEROFILL";

    @Override
    public Identifier toPhysicalTableName(Identifier name, JdbcEnvironment
context) {
        // 驼峰命名策略转换表名
        String tableName =
ImprovedNamingStrategy.INSTANCE.tableName(name.getText());
        // 将entity中的表名全部转换成大写
        tableName = tableName.toUpperCase();
        return Identifier.toIdentifier(tableName);
    }

    @Override
    public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment
context) {
        // 驼峰命名策略转换字段名
        String colnumName =
ImprovedNamingStrategy.INSTANCE.columnName(name.getText());
        // 将entity中的字段名全部转换成大写
        colnumName = colnumName.toUpperCase();
        // 如果entity字段名与mysql关键字同名，则在entity字段名加上``
        if ("mysql".equalsIgnoreCase(DataSourceUtil.getDatabaseProductName())){
            if (Arrays.asList(StringUtils.split(mysqlKey,
",")).contains(colnumName.toUpperCase())){
                colnumName = "`" + colnumName + "`";
            }
        }
        return Identifier.toIdentifier(colnumName);
    }
}
```

如果自定义的 MyPhysicalNamingStrategyStandardImpl 类未加 @Component 注解（将类交给 spring 容器管理），则需要在 application.properties 里面添加如下配置（hibernate版本 >= 5.0）：

```
# 值为自定义的MyPhysicalNamingStrategyStandardImpl类的全限定类名
spring.jpa.hibernate.naming.physical-
strategy=com.test.config.MyPhysicalNamingStrategyStandardImpl
```

需要注意的是，如果 hibernate 版 < 5.0，则配置里的内容应是

```
spring.jpa.hibernate.naming-
strategy=com.test.config.strategy.MyImprovedNamingStrategy
```

同时自定义的 MyImprovedNamingStrategy 类继承 ImprovedNamingStrategy，并且重写相应的方法。

Spring JPA Junit 关闭自动回滚

使用 JPA 配合 Hibernate ，采用注解默认是开启了 LayzLoad 也就是懒加载，所以当操作为增删改时需在 Junit 的单元测试上加上`@Transactional`注解，这样 Spring 会自动为当前线程开启 Session，这样在单元测试里面懒加载才不会因为访问完 Repository 之后，出现 session not found.

但若在单元测试上面加了 @Transactional 会自动回滚事务，需要在单元测试上面加上`@Rollback(false)`，才能修改数据库。

```
@Test
@Transactional
@Rollback(false) //关闭自动回滚
public void saveTest() {
    ProductCategory category = new ProductCategory();
    category.setCategoryname("快乐");
    category.setCategorytype(6);
    ProductCategory save = categoryRepository.save(category);
    System.out.println(save.toString());
```

延迟加载与立即加载（FetchType）

通常可以在@OneToMany中用LAZY、在@ManyToOne/Many中用EAGER，但不绝对，看具体需要。

- FetchType.LAZY：延迟加载，在查询实体A时，不查询出关联实体B，在调用getxxx方法时，才加载关联实体，但是注意，查询实体A时和getxxx必须在同一个Transaction中，不然会报错:no session。即会表现为两次单独的SQL查询（非联表查询）
- FetchType.EAGER：立即加载，在查询实体A时，也查询出关联的实体B。即会表现为一次查询且是联表查询

默认情况下，@OneToOne、@ManyToOne是LAZY，@OneToMany、@ManyToMany是EAGER。

有两个地方用到延迟加载：relationship（@OneToMany等）、attribute（@Basic）。后者一般少用，除非非常确定字段很少访问到。

时间类型的精度问题

如 MySQL 的 DATETIME 类型，默认是精确到秒的，故存入的时间戳的毫秒会被舍弃并根据四舍五入加入到秒（如1s573ms变成2s、1s473ms变成1s），从而保存进去与查出来的也会不一致。

外键关联、关联删除

外键关联

相关注解：@ManyToOne/@OneToMany/@OneToOne 、@JoinColumn/@PrimaryKeyJoinColumn、@MapsId，用法及区别见：hibernate基于注解的维护权反转：@OneToMany(mappedBy=)

- @JoinColumn 用来指定外键，其 name 属性指定该注解所在 Entity 对应的表的一个列名
- @ManyToOne 等用来指定对应关系是多对一等数量对应关系

通过 `@ManyToOne` 等注解指定数量对应关系时，须在多的一方标注（@ManyToOne），一的一方注不注均可。（以下以 School、Student 为例，为一对多关系）

- 若只用 `@ManyToOne` 等注解没用 `@JoinColumn` 注解，则在生成表时会自动生成一张关联表来关联 School、Student，表中包含School、Studeng 的 id
- 若在用了 `@ManyToOne` 等注解的基础上用了 `@JoinColumn` 注解则不会自动生成第三张表，而是会在多的一方生成一个外键列。列名默认为 ${被引用的表名}_id （可以通过 `@JoinColumn` 的name 属性指定列名）。
- 上法的缺点是在 insert 多的一方后会再执行一次 update 操作来设置外键的值（即使 在insert 时已经指定了），避免额外update 的方法：在一的一方不使用 `@JoinColumn` ，而是改为指定 `@OneToMan` 等注解 的 mappedBy 属性。
- 注意：`@JoinColumn` 注解和 `@ManyToOne` 等注解的 mappedBy 属性不能同时存在，会报错。

关联删除

假设有 user、admin 两表，admin.user_id 与 user.id 对应。当要删除 userId 为"xx"一条 admin 表记录时：

- 若业务逻辑中未使用 JPA 软删除：
  - 若后者通过外键关联前者，则直接从 user 删除id为"xx"的记录即可，此时会级联删除admin 表的相应记录。

    当然要分别从两表删除记录也可，此时须保证先从admin表再从user表删除；
  - 若无外键关联，则需要分别从user、admin删除该记录，顺序先后无关紧要；
- 若使用了软删除，对于软删除操作外键将不起作用（因为物理上并未删除记录），因此此时也只能分别从两表软删除记录。但不同的是，此时须先从admin再从user表删除记录。
- 若顺序相反，会发现 user 表的记录不会被软删除。猜测原因为：内存中存在 userEntity、adminEntity 且adminEntity.userByUserId 引用了 userEntity，导致delete userEntity 时发现其被 adminEntity 引用了从而内部取消执行了 delete 操作。

在实际业务中一般都会启用软删除，所以物理删除的场景很少，综上，**在涉及到关联删除时，最好按拓扑排序的顺序（先引用者再被引用者）依次删除各 Entity 记录。**

示例：

进行如下设置后，JPA 会自动生成为 student 表生成两个外键约束：student 表 school_id 关联 school 表 id 自动、student 表 id 字段关联 user 表 id 字段。

```
//StudentEntity
//get set ...

@Column(name = "id")
private String sId;
```

```
@Column(name = "school_id")
private String schoolId;


@ManyToOne
@JoinColumn(name = "school_id", referencedColumnName = "id", insertable = false,
updatable = false)//school.school_id字段外键关联到school.id字段；多个字段对应数据库同一字段时会报错，通过添加insertable = false, updatable = false即可
private SchoolEntity schoolBySchoolId;

@OneToOne
@JoinColumn(name = "id", referencedColumnName = "id", insertable = false,
updatable = false) //student.id字段外键关联到user.id字段。也可用@PrimaryKeyJoinColumn
@MapsId(value = "id")
private UserEntity userByUserId;
```

对于外键属性（如上面 student 表的 school_id），当该属性不是当前表的主键时，通过 @OneToOne/@ManyToOne + @JoinColumn 定义即可成功地在数据库中自动生成产生外键约束。但当该属性也是当前表的主键时（如为 student.id 定义外键来依赖user.id字段），单靠 @OneToOne + @JoinColumn 并不能自动产生外键约束，此时可通过加 @MapIds 来解决。

总结：

通过 @ManyToOne/@OneToMany/@OneToOne + @JoinColumn/@PrimaryKeyJoinColumn 定义外键，是否需要 @MapsId 视情况而定。

外键场景有两种：

- 外键属性不是当前表的主键（如上面 student 表的 school_id 字段不是主键）
- 外键属性也是当前表的属性（如上面 student 表的 id 字段是主键）

基于这两种场景，各注解使用时的组合及效果如下：

外键属性 不是主键 的场景			
		@JoinCloumn	@PrimaryKeyJoinColumn
@OneToOne	无@MapsId	√	×
	有@MapsId		√，id关联user.id
@ManyToOne	无@MapsId	√	√，school_by_schoool_id关联school.id
	有@MapsId		

外键属性 也是主键 的场景			
		@JoinCloumn	@PrimaryKeyJoinColumn
@OneToOne	无@MapsId	×	×
	有@MapsId	√	√
@ManyToOne	无@MapsId	√	√，user_by_user_id关联user.id
	有@MapsId		

说明：

使用注解组合后是否会自动为表生成外键约束？打钩的表示会、打叉的表示不会、半勾半叉的表示会但是生成的不是预期的（如场景1中期望 school_id 关联了 school id 自动，但一种结果是 id 关联了 user id、另一种是自动产生了 school_by_school_id 字段并关联到了school_id，显然都不符合期望）。

结论：

- 外键属性不是主键的场景（第一种），用 **@OneToOne/@ManyToOne + @JoinColumn** 即可，为了简洁推荐不用 @MapIds，示例见上面的 school_id 关联 school id 设置。
- 外键属性是主键的场景（第二种），用 **@OneToOne + @JoinColumn + @MapsId**

示例见上面的 student id 关联 user id 设置。

虽从表中可见场景二有三种组合都可以达到目标，但为了符合业务语义（主键嘛，当然是唯一的，因此是一对一）且为了和场景一的尽可能统一，采用这个的组合。

实践发现，使用 @MapsId 时，要求外键字段、被关联的字段的数据库列名得相同且都得为"id"。why? 如何避免？TODO

通过 JPA 定义表结构的关联关系（如共用部分字段等）

这里以实际项目中课程、实验、步骤与其翻译数据的表结构关联方案设计为例：

多语言表（翻译表）与原表（主表）关联方案设计，需求：字段（列）复用以免重复代码定义、同一个列的定义如是否为空在不同表中可不一样（如有些字段主表中非空但翻译表中可空），有如下方案：

1. 无关联，重复定义。**pass**
2. 有关联
3.
    1. 通过@MappeSuperclass，不同子类可以完全继承父类列定义且分别对应不同表，表结构完全相同，但不能覆盖父类的定义。**pass**
    2. 通过@Inheritance，三种策略：
    3.
        1. SINGLE_TABLE：父、子类对应同一张表。源课程和翻译课程id一样，违背主键唯一约束。**pass**
        2. JOINED：父、子类对应不同表且子类自动加与父类主键一样的字段与父类主键关联，但父表中除主键之外的所有字段无法在子表中再出现。**pass**
        3. TABLE_PER_CLASS：父、子类对应不同表且表定义完全相同，无外键，但同一字段在不同表中字段定义无法不同。**pass**
    4. 定义个普通父类，子类继承父类并分别进行@Column定义：不同子类对应不同表，不同表含有的字段及定义可不一样。**selected**

Web 支持

参阅：https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#core.web

Basic Web Support（Domain class、Pageable、Sort）

domain 类（即被 Spring Data Crud Repository 管理的 domain类，如 Entity 类）及 Pageable、Sort 可以直接作为 handler 方法的形参，框架会自动解析请求参数组装成相应的实参，示例：

```
@Controller
@RequestMapping("/users")
class UserController {

    @RequestMapping("/{id}")
    String showUserForm(@PathVariable("id") User user, Model model) {
        model.addAttribute("user", user);
        return "userForm";
    }
}
```

```
@Controller
@RequestMapping("/users")
class UserController {
```

```
    private final UserRepository repository;

    UserController(UserRepository repository) {
        this.repository = repository;
    }

    @RequestMapping
    String showUsers(Model model, Pageable pageable) {

        model.addAttribute("users", repository.findAll(pageable));
        return "users";
    }
}
```

- 对于 domain 类，会自动根据 request 的"id"参数调用 repository 的 findById 查得对象

  request 示例：/user?id=2
- 对于 Pageable，会根据 request"page"、"size"参数组装对象

  request 示例：/users?page=0&size=2
- 对于 Sort，会根据 request 的"sort"参数组装对象，该参数值须遵循规则：
  property,property(,ASC|DESC)(,IgnoreCase)

  request示例：/users?sort=firstname&sort=lastname,asc&sort=city,ignorecase
- 内部原理：第一者是由 DomainClassConverter 类完成的，后两者是由 HandlerMethodArgumentResolver 完成的。

Querydsl Web Support

可以直接将 Querydsl 的 Predicate 作为 handler 方法的形参，框架会自动（默认只要 Querydsl 在 classpath 上就会生效）根据请求参数组装创建 Predicate 实例。示例：

```
@Controller
class UserController {

    @Autowired UserRepository repository;

    @RequestMapping(value = "/", method = RequestMethod.GET)
    String index(Model model, @QuerydslPredicate(root = User.class) Predicate
predicate,
          Pageable pageable, @RequestParam MultiValueMap<String, String>
parameters) {

        model.addAttribute("users", repository.findAll(predicate, pageable));
        return "index";
    }
}
```

SpringDataJpa 和 mybatis 的比较

- spring data jpa 实现了 jpa（java persistence api）功能，即可以实现 pojo 转换为关系型数据库记录的功能，通俗来讲就是可以不写任何的建表 sql 语句了。jpa 是 spring data jpa 功能的一个子集。

  而 mybatis 并没有 jpa 功能，建表语句还是要自己写的。

- spring data jpa 是全自动框架，不需要写任何 sql。

  而 mybatis 是半自动框架，需要自己写 sql，mybatis-plus 为 mybatis 赋能，使其也可以基本上不需要写任何模板 sql。

- debug 模式下看生成的 sql，mybatis 下的 sql 可读性很好，而 spring data jpa下 的查询 sql 可读性并不好。

- spring data jpa 的 insert 与 update 都调用同一个方法 save，如果带有主键 id（如果启用了乐观锁，那么还有 version 字段），那么就是更新，否则就是新增，所以 addOrUpdate 是一个接口；

  而 mybatis 中提供 insert 方法和 updateById 方法。

  由于 spring data jpa 调用同一个方法，所以其要执行两条 sql，先执行查询，再执行插入/更新。

  另外就是返回值，spring data jpa 的返回值是 Employee 对象，而 mybatis 的返回值是影响的行数，当然 mybatis 也可以得到新增后的 id，返回新增后的对象

- spring data jpa 的 dynamic sql 是使用 JpaSpecificationExecutor，而 mybatis 中是使用 xml 来构造 dynamic sql。

  当执行分页查询的时候，spring data jpa 实际上是调用了两个 sql 语句，通过 count 获得总记录数，即当用到 Pageable 的时候会执行一条 count 语句，这可能是很昂贵的操作，因为 count 操作在 innodb 中要扫描所有的叶子节点，通过 limit 来获得分页记录

  mybatis 获得总记录数好像并不是通过执行 count 语句来获得的，可能是通过游标 cursor 的方式来获得的，通过 druid 监控，其只执行一条sql语句

- spring data jpa 支持自己来写 sql 语句，有两种方式：

  1）@Query 或 @Modifying 配合 @Query

  2）通过 entityManager

  但要注意的是：如果自己写 sql 语句，那么有些拦截器可能并不能起作用，如 @PreUpdate

  相对来说，mybatis 就比较简单，直接在 mapper xml 中写 sql 就可以了

问题交流|源码获取|商务合作

微信名片