Specifications && Querydsl

先了解下什么是 `criteria 查询` ,标准查询就是，将到数据库的映射类 entity，生成元模型，通过操作这些元数据就可以操作数据库，通过写 `Java` 代码就可以实现很多定制化的操作，且更加的灵活。且能够保证类型安全。

所谓元模型，是一个模型，这个模型中管理者具体我们需要持久化的单元(column)，在类中增加`java.persitence.Entity` 注解就可以生成这个元模型了，

生成的如下：

```
import javax.annotation.Generated;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.ListAttribute;
import javax.persistence.metamodel.StaticMetamodel;
@Generated("org.hibernate.jpamodelgen.JPAMetaModelEntityProcesso")
@StaticMetamodel(Employee.class)
public class Employee_ {
    public static volatile SingularAttribute<Employee, Integer> id;
    public static volatile SingularAttribute<Employee, Integer> age;
    public static volatile SingularAttribute<Employee, String> name;
    public static volatile ListAttribute<Employee, Address> addresses;
}
```

其属性都是static和public 的。

于是就可以使用 `CriteriaQuery` 接口查询了，比如：`select` 、`from`、`where`、`group by`、`order by`等。

例如，要完成一个查询代码：`SELECT * FROM employee WHERE age > 24`

```
# 获取建立查询工厂
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();

# 建立查询定义
CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);

# 查询初始
Root<Employee> employee = criteriaQuery.from(Employee.class);

# 建立查询过滤
Predicate condition = criteriaBuilder.gt(employee.get(Employee_.age), 24);

# 将查询过滤塞入到查询定义，也就是在查询初始之后
criteriaQuery.where(condition);
```

```
# 建立可执行查询
TypedQuery<Employee> typedQuery = em.createQuery(criteriaQuery);
# 执行查询
List<Employee> result = typedQuery.getResultList();
```

其它几个例子：

```
// IN
CriteriaQuery<Employee> criteriaQuery = criteriaBuilder
.createQuery(Employee.class);
Root<Employee> employee = criteriaQuery.from(Employee.class);
criteriaQuery.where(employee.get(Employee_.age).in(20, 24));
em.createQuery(criteriaQuery).getResultList();

// 排序
CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
Root<Employee> employee = criteriaQuery.from(Employee.class);
criteriaQuery.orderBy(criteriaBuilder.asc(employee.get(Employee_.age)));
em.createQuery(criteriaQuery).getResultList();
```

上面的代码确实会比较难受，因为有很多的模板代码，完成一个简单的 sql 查询，却要写那么多的模板。

Specification 接口

```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery query, CriteriaBuilder
cb);
}
```

当将对象类型 T 传进去之后，你就定义好了，`Root`，`query`，`cb` 等，然后就可以返回谓语语句，然后将这个谓语语句在 继承了 `JpaSpecificationExecutor` 接口的 `Repository` 里面使用。

如，写两个函数，函数返回的是 `Specification` 类型

```
public CustomerSpecifications {
  public static Specification<Customer> customerHasBirthday() {
        return (root, query, cb) ->{
            return cb.equal(root.get(Customer_.birthday), today);
        };
   }
 public static Specification<Customer> isLongTermCustomer() {
        return (root, query, cb) ->{
            return cb.lessThan(root.get(Customer_.createdAt), new
LocalDate.minusYears(2));
       };
   }
}
```

然后使用：

```
customerRepository.findAll(hasBirthday());
customerRepository.findAll(isLongTermCustomer());
// customerRepository 接口 repository 是继承了 JpaSpecificationExecutor 接口
```

`JpaSpecificationExecutor` 的方法有：

```
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);
}
```

但是上面的代码还是有模板代码的，每次都需要写一个 `Specification` 于是还需要继续减少代码量

目前 `QueryDsl` 支持的有 JPA, JDO, JDBC, Lucene, Hibernate Search, MongoDB, Collections and RDFBean as backends.。

同样使用 `@Entity` 注解的类，当项目启动的时候就会生成代码，生成的代码的存放位置在Maven 的pom中当引入QueryDsl 的时候就会定义。

但是如果你使用的不是JPA或者JDO，那么需要使用QueryDsl 的注解去生成代码 `com.querydsl.core.annotations.QueryEntity` 不过在项目中会直接使用这个注解

假如现在有一个类entity

```
@QueryEntity
public class Customer {
  private String firstName;
  private String lastName;
  public String getFirstName() {
```

```
    return firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public void setFirstName(String fn) {
    firstName = fn;
  }
  public void setLastName(String ln) {
    lastName = ln;
  }
}
```

会生成名为 `QCustomer` 的 `query type`， 各个类属性生成代码 `StringPath`, `NumberPath`, `EnumPath` 类型，且这些类型都是继承 `Path`, 当然 `Path` 又继承自 `Expression` ，类属性中还会有其它类作为属性(如在构建一对多的表机构关系)，而所有的这些Q开头的都是继承自 `EntityPath`， 然后 `EntityPath` 又是继承自 `Path`.

初始化 `Query` 然后给进去参数就可以直接使用类，

```
public class QuerydslJpaSupport {
    private EntityManager entityManager; // Entity 的管理类，需要将这个参数，才可以使用jpaQueryFactory
    protected <T> JPAQuery<T> from(EntityPath<T> from) {
        JPAQueryFactory jpaQueryFactory = new JPAQueryFactory(entityManager);
        // 相当于 select * from XXX; XXX 是在后面根据需求定义
        // selectFrom 在源码里面的是这样的 select(from).from(from);
        return jpaQueryFactory.selectFrom(from);
    }
    protected <T> JPAQuery<T> select(EntityPath<T> what) {
        JPAQueryFactory jpaQueryFactory = new JPAQueryFactory(entityManager);
        // 相当于 select what..(from A);
        // 选一个column，至于说from 哪个entity，也是调用这个方法的代码中完成的
        return jpaQueryFactory.select(what);
    }
    protected <T> JPAQuery<T> select(Expression<T> what) {
        JPAQueryFactory jpaQueryFactory = new JPAQueryFactory(entityManager);
        return jpaQueryFactory.select(what);
    }
    protected void store(Object entity) {
        if (entityManager.contains(entity)) {
            entityManager.merge(entity);
            return;
        }
        entityManager.persist(entity);
    }
    protected JPADeleteClause delete(EntityPath<?> from) {
        return new JPADeleteClause(entityManager, from);
    }
```

```
    protected EntityManager entityManager() {
        return entityManager;
    }

    // 初始化持久化Entity
    @PersistenceContext
    protected void setEntityManager(final EntityManager entityManager) {
        this.entityManager = entityManager;
    }
}
```

除了使用上面的创建factory方法，还可以创建：

```
@Bean
@Autowired
public JPAQueryFactory jpaQuery(EntityManager entityManager) {
    return new JPAQueryFactory(entityManager);
}

@Autowired
JPAQueryFactory queryFactory;
```

然后再使用的时候继承 `QuerydslJpaSupport`

```
@Repository
public class AuditSampleRepository extends QuerydslJpaSupport {
    public long countByLabelTaskId(String labelTaskId) {
        return from(auditSample)
                .where(auditSample.labelTask.id.eq(labelTaskId))
                .fetchCount();
    }
    public void save(AuditSample auditSample) {
        store(auditSample);
    }
    public Optional<AuditSample> findByTaskIdAndImageId(String taskId, long
imageId) {
        return Optional.ofNullable(from(auditSample)
                .where(auditSample.labelTask.id.eq(taskId)
                        .and(auditSample.imageId.eq(imageId)))
                .fetchFirst());
    }
    public Long findPreviousAuditImageId(String taskId, Long imageId) {
        return select(auditSample.imageId)
                .from(auditSample)
                .where(auditSample.labelTask.id.eq(taskId)
                        .and(auditSample.imageId.lt(imageId)))
                .orderBy(auditSample.imageId.desc())
                .fetchFirst();
```

```
    }
    public Long findNextAuditImageId(String taskId, Long imageId) {
        return select(auditSample.imageId)
                .from(auditSample)
                .where(auditSample.labelTask.id.eq(taskId)
                        .and(auditSample.imageId.gt(imageId)))
                .orderBy(auditSample.imageId.asc())
                .fetchFirst();
    }
    public long findNumberOfAudited(String taskId) {
        return select(auditSample.id)
                .from(auditSample)
                .where(auditSample.labelTask.id.eq(taskId)
                        .and(auditSample.auditState.ne(UNAUDITED)))
                .fetchCount();
    }
    public Long findFirstUnauditedImageId(String taskId) {
        return select(auditSample.imageId)
                .from(auditSample)
                .where(auditSample.labelTask.id.eq(taskId)
                        .and(auditSample.auditState.eq(UNAUDITED)))
                .orderBy(auditSample.imageId.asc())
                .fetchFirst();
    }
    public Long findLastImageIdByTaskId(String taskId) {
        return select(auditSample.imageId)
            .from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId))
            .orderBy(auditSample.imageId.desc())
            .fetchFirst();
    }
    public long countByTaskAndAuditState(String taskId, AuditState auditState)
{
        return select(auditSample.id)
                .from(auditSample)
                .where(auditSample.labelTask.id.eq(taskId)
                        .and(auditSample.auditState.eq(auditState)))
                .fetchCount();

    }
    public String findFirstUnauditedSubtaskId(String taskId) {
        return select(auditSample.labelSubtaskId)
            .from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId)
                .and(auditSample.auditState.eq(UNAUDITED)))
            .orderBy(auditSample.labelSubtaskName.asc())
            .fetchFirst();
    }
    public AuditSample findLastSubtaskIdByTaskId(String taskId) {
```

```
        return select(auditSample)
            .from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId))
            .orderBy(auditSample.labelSubtaskName.desc())
            .fetchFirst();
    }
    public Optional<AuditSample> findByTaskIdAndLabelSubtaskId(String taskId,
String labelSubtaskId) {
        return Optional.ofNullable(from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId)
                .and(auditSample.labelSubtaskId.eq(labelSubtaskId)))
            .fetchFirst());
    }
    public String findPreviousAuditLabelSubtaskId(String taskId, String
labelSubtaskId) {
        return select(auditSample.labelSubtaskId)
            .from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId)
.and(auditSample.labelSubtaskName.lt(select(auditSample.labelSubtaskName).from
(auditSample).where(auditSample.labelSubtaskId.eq(labelSubtaskId)))))
            .orderBy(auditSample.labelSubtaskName.desc())
            .fetchFirst();
    }
    public String findNextAuditLabelSubtaskId(String taskId, String
labelSubtaskId) {
        return select(auditSample.labelSubtaskId)
            .from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId)
.and(auditSample.labelSubtaskName.gt(select(auditSample.labelSubtaskName).from
(auditSample).where(auditSample.labelSubtaskId.eq(labelSubtaskId)))))
            .orderBy(auditSample.labelSubtaskName.asc())
            .fetchFirst();
    }
    public AuditSample findFirstUnauditedLabelSubtaskId(String taskId) {
        return select(auditSample)
            .from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId)
                .and(auditSample.auditState.eq(UNAUDITED)))
            .orderBy(auditSample.labelSubtaskName.asc())
            .fetchFirst();
    }
    public long countByTaskId(String taskId) {
        return select(auditSample.id)
            .from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId))
            .fetchCount();
    }
    public List<AuditSample> findByTaskIdAndAuditState(String taskId,
AuditState auditState) {
```

```
        return from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId)
                .and(auditSample.auditState.eq(auditState)))
            .fetch();
    }
    public long countRejectdByLabelTaskId(String labelTaskId) {
        return from(auditSample)
            .where(auditSample.labelTask.id.eq(labelTaskId)
                .and(auditSample.auditState.eq(AuditState.REJECTED)))
            .fetchCount();
    }
    public Optional<AuditSample> findByTaskIdAndRelativeImageId(String taskId,
long relativeImageId) {
          List<AuditSample> auditSamples =   from(auditSample)
            .where(auditSample.labelTask.id.eq(taskId)
                .and(auditSample.auditState.eq(AuditState.REJECTED)))
                .orderBy(auditSample.imageId.asc()).fetch();
          return  Optional.ofNullable(auditSamples.get((int) (relativeImageId-
1)));
    }
}


    default void queryByCreatedDateRange(
        QuerydslBindings bindings,
        DateTimePath<Date> startDate,
        DateTimePath<Date> endDate,
        DateTimePath<Date> createDate
    ) {
        bindings.bind(startDate).first((path, value) -> {
            Date from = DateUtil.atStartOfDay(value);
            return createDate.after(from);
        });

        bindings.bind(endDate).first((path, value) -> {
            Date to = DateUtil.atEndOfDay(value);
            return createDate.before(to);
        });
    }s
```

`selectFrom` 定义了查询的源头，`where` 定义了对源头查询的过滤操作，`fetchOne` 是真正执行查询的动作，返回一个数据。

下面这个例子是等价的，也就是上面代码中`select` 和 `selectFrom`

```
Customer bob = queryFactory.selectFrom(customer)
                    .where(customer.firstName.eq("Bob"))
                    .fetchOne();

Customer bob = queryFactory.select(customer).from(customer)
                    .where(customer.firstName.eq("Bob"))
                    .fetchOne();
```

常见的使用方法

- *select* : 设置查询的映射
- *from*: 添加查询数据源，
- *where*: 添加查询过滤条件
- *groupby*：groupby操作
- *having*： 也是过滤的时候时候，不过是在 `groupby` 之后使用
- *orderBy* ： 查询结果中增加排序，在数字、String 中都可以使用， `asc()` 或者 `desc()`
- *limit, offset, restrict* 等

排序

```
QCustomer customer = QCustomer.customer;
queryFactory.selectFrom(customer)
    .orderBy(customer.lastName.asc(), customer.firstName.desc())
    .fetch();
```

删除

```
QCustomer customer = QCustomer.customer;
// delete all customers
queryFactory.delete(customer).execute();
// delete all customers with a level less than 3
queryFactory.delete(customer).where(customer.level.lt(3)).execute();
```

查询部分字段

```
query = new JDOSQLQuery<Void>(pm, templates);
List<Tuple> rows = query.select(cat.id, cat.name).from(cat).fetch();

// 查询所有字段
List<Tuple> rows = query.select(cat.all()).from(cat).fetch();
```

join

```
query = new JDOSQLQuery<Void>(pm, templates);
cats = query.select(catEntity).from(cat)
    .innerJoin(mate).on(cat.mateId.eq(mate.id))
    .where(cat.dtype.eq("Cat"), mate.dtype.eq("Cat"))
    .fetch();
```

插入

```
QSurvey survey = QSurvey.survey;
queryFactory.insert(survey)
    .columns(survey.id, survey.name)
    .values(3, "Hello").execute();

queryFactory.insert(survey)
    .values(4, "Hello").execute();
```

更新

```
QSurvey survey = QSurvey.survey;
queryFactory.update(survey)
    .where(survey.name.eq("XXX"))
    .set(survey.name, "S")
    .execute();

queryFactory.update(survey)
    .set(survey.name, "S")
    .execute();
```

上面的使用方法，直接就在语句中查询数据，执行了fetch语句，这样就不能满足分页这些需求，满足分页：

1. 还是使用生成的Qxxx代码构建Predicate

`predicate = QTask.Task.name.containsIgnoreCase(query).and(predicate);`

2. 然后 `QuerydslPredicateExecutor` 会提供一些方法，可以使用predicate，pageable等参数:
    1. `Optional<T> findOne(Predicate predicate);`
    2. `Iterable<T> findAll(Predicate predicate);`
    3. `Iterable<T> findAll(Predicate predicate, Sort sort);`
    4. `Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);`
    5. `Page<T> findAll(Predicate predicate, Pageable pageable);`
    6. `long count(Predicate predicate);`
    7. `boolean exists(Predicate predicate);`

如果使用这种方法，那么应该是在一个接口中，而不是class中，上面的直接查询数据可以直接在class 中并且直接调用。注意在interface 中也可以添加方法，此时需要使用到 `default` 关键字

如在某 `interface` 中

```
default Page<LabelTask> findAll(String query, String groupId, Predicate
predicate,
                                      Pageable pageable) {
        BooleanExpression expression = .....;
        return findAll(expression, pageable, joinDescriptors);
}
// 根据名称也还是可以建立查询语句
List<LabelTask> findAllByStatus(Status status);
```

可以使用更优雅的BooleanBuilder 来进行条件分支管理

```
QMemberDomain qm = QMemberDomain.memberDomain;
Iterable<MemberDomain> iterable =
memberRepo.findAll(qm.status.eq("0013"));

BooleanBuilder builder = new BooleanBuilder();
builder.and(qm.status.eq("0013"));
Iterable<MemberDomain> iterable2 = memberRepo.findAll(builder);
```

支持自定义查询

如有一个 `url=/posts?title=title01` 那么它应该返回的是文章 `title` 等于 `title01` 的文字，但是这里你可以改变它的查询方法，如返回的文字的 `title` 名称中包含 `title01` 。

此时你的 `interface` 中不仅 `extends` `QueryDslPredicateExecutor`，还需要 `QuerydslBinderCustomizer<QT>`，然后再重写 (QT 是QueryDsl生成的Q类)

```
@Override
void customize(QuerydslBindings bindings, QT root);
```

一个例子：

```
interface UserReposotory extends CurdRepository<User, String>,
  QueryDslPredicateExecutor<User>,
  QuerydslBinderCustomizer<QUser> {
        @Override
        default public void customize(QuerydslBindings bindings, QUser user) {
              bindings.bind(user.username)
                      .first((path, value) -> path.contains(value));

              bindings.bind(String.class)
                      .all((StringPath path, String value) ->
 path.containsIgnoreCase(value));
              bindings.excluding(user.password);
    }
  }
```

第一个就定义了查询路径中的 `user.name` 的值需要包含`value`

第二个就定义了查询路径中的 `user.name` 的值需要忽略大小写包含`value`

```
first 和 all 的区别,
first:
    /posts?title=title01 title 这里只有一个值，在一次请求 url(predicate) 只有这么一个
值
all:
    /posts?title=[title01, title02...]title 这里会有多个值，在一次请求
url(predicate) 会对于多个值，而在源码中 all 中的value是一个 Collection 类型的
```