

UserInfo

基本字段：user_id, account昵称, password, 附基本构造函数与含参构造函数

一个用户可以有多个动态，可以关注多个用户。

```
@OneToMany(mappedBy = "publisher", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@ApiModelProperty(value="该用户的动态")
private List<TimelyNews> timelyNewsList = new ArrayList<>();

@OneToMany(mappedBy = "follower", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@ApiModelProperty(value="该用户关注的人")
private List<Follow> followeeList = new ArrayList<>();
```

@OneToMany

在关系一对多的“一”这一方进行配置

- mappedBy参数用于定义类之间的双向关系（如果是单向关系则不用提供定义），该属性的值为“多”方class里的“一”方的属性名称。比如学生-辅导员，在学生这一方的“辅导员姓名”。
- cascade参数配置级联(可以配置到多表的映射关系的注解上，关系递归调用)

```
CascadeType.all: 所有
    REFRESH : 级联刷新
    REMOVE  : 级联删除
    MERGE   : 级联更新
    PERSIST : 级联保存
```

- fetch参数配置关联对象的加载方式

```
EAGER: 立即加载（主类加载时加载）
LAZY  : 延迟加载（关系类被访问时才加载）
```

- 关系配置好之后，直接使用即可，比如user.timelyNewsList

```
private List<Follow> followeeList = new ArrayList<Follow>();
```

TimelyNews

多个动态对应一个用户

基本参数：动态编号id，发布时间Long time，内容String news

```
@ManyToOne()
@JoinColumn(name = "user_info_user_id")
@JsonBackReference
@ApiModelProperty(value = "发布者ID")
private UserInfo publisher;
```

@ManyToOne : 配置多对一关系

参照表的概念

拿dept部门表与iemp表举例：

这两张表有一种关系的，即父子关系，部门表是父表，员工表示子表，因为：在一个部门中可以有很多的员工，但是一个员工不可能在许多的部门吧！这是一种一对多的关系，比如说在部门表中deptno字段没有50这个部门编号，而在员工表的deptno字段中出现了50，这显然是不符合逻辑的，所有我们在建表的时候就要设定一种限制，让子表的deptno字段取值参照主表的deptno，形成一种参照关系，这样做出来的才有实际意义，懂了吗？在建立子表的时候要指定这种参照关系，也就是用约束来指定，请看下面的语法：

```
create table emp (empno number (4) primary key, ename varchar2 (10) not null, deptno number (2) ) constraint fk_deptno foreign key(deptno) references dept(deptno);
```

二级（关联）表

根据具体注解配置，不一定会生成DB的二级表，但多对多关系是必须要有关联表的。

@JoinColumn：配置外键

由于 @OneToOne（一对一）、@OneToMany（一对多）、@ManyToOne（多对一）、@ManyToMany（多对多）等注解只能确定实体之间几对几的关联关系，**它们并不能指定与实体相对应的数据库表中的关联字段，因此，需要与 @JoinColumn 注解来配合使用。**我认为，可以把 @JoinColumn当成对@Column的一种扩展。

name参数：外键字段名称，若不设置，默认取值为：实体名称 + "_" + 被引用的主键列的名称

referencedColumnName参数：参照的主表的主键字段名称。默认值为被引用的实体的主键的名称，因此通常不需要提供定义。若不想被引用的实体的主键作为外键，则需要设置。

以下是一些参考资料：

在此，我们以员工、地址、部门、角色四者之间的关联关系为例进行详细介绍，基于如下假设：

- 一个员工只能有一个地址，同样的，一个地址也只属于一个员工；
 - 一个员工只能属于一个部门，但是一个部门可以包含有多个员工；
 - 一个员工可以拥有多个角色，同样的，一个角色也可以属于多个员工。

@OneToOne (一对一)

@OneToOne 用来表示类似于以上员工与地址之间的一对一的关系，在员工表中会有一个指向地址表主键的字段address_id，所以主控方（指能够主动改变关联关系的一方）一定是员工，因为，只要改变员工表的address_id就改变了员工与地址之间的关联关系，所以@JoinColumn要写在员工实体类Employee上，自然而然地，地址就是被控方了。

```
1 @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
2 @JoinColumn(name = "address_id")
3 private Address address;
```

我们也可以不写@JoinColumn，Hibernate会自动在员工表生成关联字段，字段默认的命名规则：被控方类名_被控方主键，如：address_id。

如果两张表是以主键关联的，比如员工表主键是employee_id，地址表主键是address_id，可以使用如下注解：

```
1 @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
2 @PrimaryKeyJoinColumn(name = "employee_id", referencedColumnName = "address_id")
3 private Address address;
```

@OneToMany (一对多)

在分析员工与部门之间的关系时，一个员工只能属于一个部门，但是一个部门可以包含有多个员工，如果我们站在部门的角度来看，部门与员工之间就是一对多的关系，在部门实体类 Department 上添加如下注解：

```
1 @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
2 @JoinColumn(name = "department_id")
3 private List<Employee> employees;
```

译

我们也可以不写@JoinColumn，Hibernate会自动生成一张中间表来对员工和部门进行绑定，表名默认的命名规则：一的表名_一实体类中关联多的属性名，例如，部门表名为 tbl_department，部门实体中关联的员工集合属性名为 employees，则生成的中间表名为：tbl_department_employees。

通常并不推荐让Hibernate自动去自动生成中间表，而是使用@JoinTable注解来指定中间表：

```
1 @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
2 /**
3  * joinColumns 用来指定中间表中关联自己ID的字段
4  * inverseJoinColumns 用来指定中间表中关联对方ID的字段
5  */
6 @JoinTable(name = "tbl_employee_department", joinColumns = {
7   @JoinColumn(name = "department_id") }, inverseJoinColumns = { @JoinColumn(name = "employee_id") })
8 private List<Employee> employees;
```

@ManyToOne (多对一)

如果我们站在员工的角度来看员工与部门之间的关系时，二者之间就变成了多对一的关系，在员工实体类 Employee 上添加如下注解：

```
1 @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
2 @JoinColumn(name = "department_id")
3 private Department department;
```

@ManyToMany (多对多)

类似员工与角色之间的关系，一个员工可以拥有多个角色，一个角色也可以属于多个员工，员工与角色之间就是多对多的关系。通常这种多对多关系都是通过创建中间表来进行关联处理，并使用@JoinTable注解来指定。

一个员工可以拥有多个角色，在员工实体类中添加如下注解：

```
1 @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
2 @JoinTable(name = "tbl_employee_role", joinColumns = { @JoinColumn(name = "employee_id") }, inverseJoinColumns = {
3   @JoinColumn(name = "role_id") })
4 private List<Role> roles;
```

一个角色可以属于多个员工，在角色实体类中添加如下注解：

```
1 @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
2 @JoinTable(name = "tbl_employee_role", joinColumns = { @JoinColumn(name = "role_id") }, inverseJoinColumns = {
3   @JoinColumn(name = "employee_id") })
4 private List<Employee> employees;
```

完整代码[这篇文章](#)，对@JoinColumn介绍具有指导意义。个人认为，ManyToMany必须指定二级（中间）表，即@JoinTable，存放Aid-Bid这样的对，oneToMany和ManyToOne可以通过字段在“1”那一方的属性名进行标记。

@JsonBackReference, 循环引用问题

比如本代码中，返回给前端的UserInfo实体被实例化，由于双向引用，导致实例化的json数据无限循环) @JsonBackReference 在此处的作用相当于 @JsonIgnore，即在被注解处打断循环。

Follow

我感觉，Follow是UserInfo和UserInfo之间多对多关系的中间表。但具体的关系可能一时半会儿还研究不透，应该有更优雅的方法。

H2-database

内存模式不如本地模式，可以提供缓存

```
spring.datasource.url=jdbc:h2:file:./testDB;
spring.jpa.hibernate.ddl-auto=update
```

Controller

部分业务：

```

@ApiOperation(value = "添加动态消息", notes = "添加成功, 返回状态码201")
@PostMapping(path =("/{id}/display-add")
public Result<?> DisplayAdd(@PathVariable Integer id, @RequestBody String news) {
    TimelyNews timelyNews = new TimelyNews(System.currentTimeMillis(), news);
    Optional<UserInfo> result = userRepository.findById(id);

    if (result.isPresent()) {
        UserInfo user = result.get();

        timelyNews.setPublisher(user);
        timelyNewsRepository.save(timelyNews);









        return Result.ok(timelyNews);
    } else {
        return Result.error( code: 404, msg: "该用户不存在");
    }
}

```

```

@ManyToOne()
@JoinColumn(name = "user_info_user_id")
@JsonBackReference
@ApiModelProperty(value = "发布者ID")
private UserInfo publisher;

```

 jdbc:h2:file:./testDB
 FOLLOW
 TIMELY_NEWS
 USER_INFO
 INFORMATION_SCHEMA
 Sequences
 Users
 H2 1.4.200 (2019-10-14)

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM TIMELY_NEWS

SELECT * FROM TIMELY_NEWS;

ID	NEWS	TIME	USER_INFO_USER_ID
1	fuck you	0	null
2	fuckk you	0	null
3	fuckkk you	0	null
4	"LZC NB"	1586071320281	1

(4 rows, 3 ms)

译

```

@ApiOperation(value = "获取动态消息", notes = "返回值为动态消息, 为空则返回状态码404")
@GetMapping(path =("/{id}/display-show")
public Result<?> DisplayShow(@PathVariable Integer id) {
    Optional<UserInfo> user = userRepository.findById(id);

    if (user.isPresent()) { //用户存在
        List<Follow> followeeList = user.get().getFolloweeList();

        if (!followeeList.isEmpty()) { //被关注者存在
            List<TimelyNews> timelyNewsList = new ArrayList<TimelyNews>();

```

```

        for (int i = 0; i < followeeList.size(); i++) { //添加被关注用户动态消息
            Optional<UserInfo> result =
userRepository.findById(followeeList.get(i).getFollowee().getUserId());

            if(result.isPresent()){ //被关注用户存在
                UserInfo followee = result.get();

                if (followee.getTimelyNewsList().size() > 0) {
                    timelyNewsList.addAll(followee.getTimelyNewsList());
                }
            }

            if(timelyNewsList.size()>0){
                //动态未排序
                return Result.ok(timelyNewsList);
            }else{
                return Result.error(404,"该用户的好友没有动态!");
            }
        } else {
            return Result.error(404, "该用户没有好友!");
        }
    } else {
        return Result.error(404, "该用户不存在!");
    }
}

```

```

@ApiOperation(value = "添加关注用户", notes = "创建成功, 返回关注信息")
@PostMapping(path =("/{id1}/friends/add/{id2}")
public Result<?> Add(@PathVariable Integer id1, @PathVariable Integer id2) {
    Optional<UserInfo> result = userRepository.findById(id1);

    if (result.isPresent()) { //该用户存在
        Optional<UserInfo> followee = userRepository.findById(id2);
        UserInfo user = result.get();

        if (followee.isPresent()) { //被关注用户存在
            List<Follow> followList =
followRepository.findByFollowerUserIdAndFolloweeUserId(id1, id2);

            if (followList.isEmpty()) { //该关注关系并未建立过
                Follow follow = new Follow(followee.get().getAccount());
                follow.setFollower(user);
                follow.setFollowee(followee.get());

                //user.getFolloweeList().add(follow);
                follow = followRepository.save(follow);
                //userRepository.save(user);

                return Result.ok(follow);
            } else {
                return Result.error("该关注关系已存在!");
            }
        } else {
            return Result.error(500, "被关注用户不存在!");
        }
    }
}

```

```
    }else{  
        return Result.error("该用户不存在!");  
    }  
}
```

一些补充

可以调整一些接口，更加user-friendly,比如注册时采用Long (id) 和String (password) .