


jpa（增删改查，HQL查询，本地Sql查询，拼接查询）


1 配置数据源


```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://120.79.183.246:3306/db_springboot_jpa?useUnicode=true&characterEncoding=utf8
spring.datasource.username=root
spring.datasource.password=root
```


2 实体类

 **People.java**
1.19KB

3 dao

 **PeopleDao.java**
642B

 **PeopleDaoPlug.java**
132B

 **PeopleDaoHelper.java**
381B

继承了两个接口

JpaRepository<People, Integer> 泛型 第一个是实体Bean，第二个是实体Bean的主键类型

JpaSpecificationExecutor<People> 同样泛型指的实体Bean 用拼接查询需要继承的接口

PeopleDaoPlug 自定义一些查询，最原始的方式。

1 那么就需要定义一个接口 把查询方法定义在里面。（PeoplePlug）

2 JpaRepository类（PeopleDao） 集成 上面的类（PeoplePlug）

3 新建一个类（PeopleDaoHelper）实现 PeoplePlug。 这里是 后缀 Helper，是因为配置中写的 Helper，这个注解在 spring boot 中值应该是 Impl

```
@EnableJpaRepositories(basePackages="com.jx.dao", repositoryImplementationPostfix="Helper")
```

4 service

 **PeopleService.java**
1.63KB

5 controller

 **PeopleController.java**
2.11KB

完整项目

 **jpa.zip**
71.2KB

transactional（事务）

在jpa的基础上加上的事务，需要注意的地方在于service

代码中有两处修改数据库（两个save），很明显 `int i = 1/0;` 会抛出异常，如果不用事务注解，那么第一个保存会执行，第二个不会执行。

```
@Transactional
public void transferAccounts(Integer fromUserId , Integer toUserId , Double money) {
    User fromUser = userDao.findOne(fromUserId);
    fromUser.setMoney(fromUser.getMoney() - money);
    userDao.save(fromUser);
    User toUser = userDao.findOne(toUserId);
    toUser.setMoney(toUser.getMoney() + money);
    int i = 1/0;
    userDao.save(toUser);
}
```

完整项目



transactional.zip
67.46KB

Valid（表单校验）

要验证的字段注解

```
@Id
@GeneratedValue
private Integer id;

@Column(name = "t_name")
@Length(min=2 , max = 3 , message = "名称长度必须是2或者3")
private String name;

@Column(name = "t_age")
@Min(value = 19 , message = "年龄必须大于等于19")
private Integer age;
```

校验是否符合规则，不符合就把错误信息返回

```
@RequestMapping("valid")
private String valid(@Valid Student student , BindingResult bindingResult) {
    if(bindingResult.hasErrors()) {
        return bindingResult.getFieldError().getDefaultMessage();
    }
    return "success";
}
```

补充：

限制	说明
@Null	限制只能为null
@NotNull	限制必须不为null
@AssertFalse	限制必须为false
@AssertTrue	限制必须为true
@DecimalMax(value)	限制必须为一个不大于指定值的数字
@DecimalMin(value)	限制必须为一个不小于指定值的数字
@Digits(integer,fraction)	限制必须为一个小数，且整数部分的位数不能超过integer，小数部分的位数不能超过fraction
@Future	限制必须是一个将来的日期
@Max(value)	限制必须为一个不大于指定值的数字
@Min(value)	限制必须为一个不小于指定值的数字
@Past	限制必须是一个过去的日期
@Pattern(value)	限制必须符合指定的正则表达式
@Size(max,min)	限制字符长度必须在min到max之间
@Past	验证注解的元素值（日期类型）比当前时间早
@NotEmpty	验证注解的元素值不为null且不为空（字符串长度不为0、集合大小不为0）
@NotBlank	验证注解的元素值不为空（不为null、去除首位空格后长度为0），不同于@NotEmpty，@NotBlank只应用于字符串且在比较时会去除字符串的空格
@Email	验证注解的元素值是Email，也可以通过正则表达式和flag指定自定义的email格式

完整项目



Valid.zip
64.04KB

AOP（切面）

通过aop实现记录 controller层中每个方法的日志功能

@Aspect 定义切面

@Pointcut 切点定义 下面代码是定义的 com.jx.controller包下所有的public方法

@Before 前置执行 @Before("log()") 是建立前置执行与切点之间的连接，因为有些时候一个切面可能有多个切点，这里指定具体切点

@After 后置执行

@AfterReturning 方法执行返回之后执行，可以拿到返回值

@Around 环绕执行

@AfterThrowing 异常处理

1 执行顺序：@Around 先执行，当调用了jp.proceed() 之后， @Before 才会执行，当方法本身执行完毕 @After 才会执行，当方法本身么@AfterReturning 才能拿到值

2 @Around的特殊性：与其他执行不同的是，@Around可以干扰目标方法的执行，如果不调用 jp.proceed()，目标方法不会执行，或者回 b，那么最终返回的是b而不是a。

3 @AfterThrowing 异常处理，这个会在目标抛出异常的时候触发，但是也仅仅就是对异常进行获取，然后记录一些信息的或者做一些操作，同样会导致程序停止。

```
import javax.servlet.http.HttpServletRequest;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;
import org.springframework.web.context.request.RequestContextHolder;
import org.springframework.web.context.request.ServletRequestAttributes;

@Aspect
@Component
public class RequestAspect {

    private static Logger logger = Logger.getLogger(RequestAspect.class);

    @Pointcut("execution(public * com.jx.controller.**(..))")
    public void log() {}

    @Before("log()")
    public void doBefore(JoinPoint joinPoint) {
        logger.info("方法执行前...");
        ServletRequestAttributes sra = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
        HttpServletRequest request = sra.getRequest();
        logger.info("url:" + request.getRequestURI());
        logger.info("ip:" + request.getRemoteHost());
        logger.info("method:" + request.getMethod());
        logger.info("parameter_map:" + request.getParameterMap());
        logger.info("class_method:" + joinPoint.getSignature().getDeclaringTypeName() + "."
            + joinPoint.getSignature().getName());
        logger.info("args:" + joinPoint.getArgs());
        for (Object obj : joinPoint.getArgs()) {
            System.out.println(obj.toString());
        }
    }

    @After("log()")
    public void doAfter(JoinPoint joinPoint) {
        logger.info("方法执行后...");
    }

    @AfterReturning(returning = "result", pointcut = "log()")
    public void doAfterReturning(Object result) {
        logger.info("执行返回值: " + result);
    }
}
```

```
@Around("log()")
public Object doAround(ProceedingJoinPoint jp) throws Throwable {
    System.out.println("方法开始执行时间: " + System.currentTimeMillis());
    Object obj = jp.proceed();           //目标方法执行，获取目标方法的返回值
    System.out.println("方法结束执行时间:" + System.currentTimeMillis());
    return obj;
}
}
```