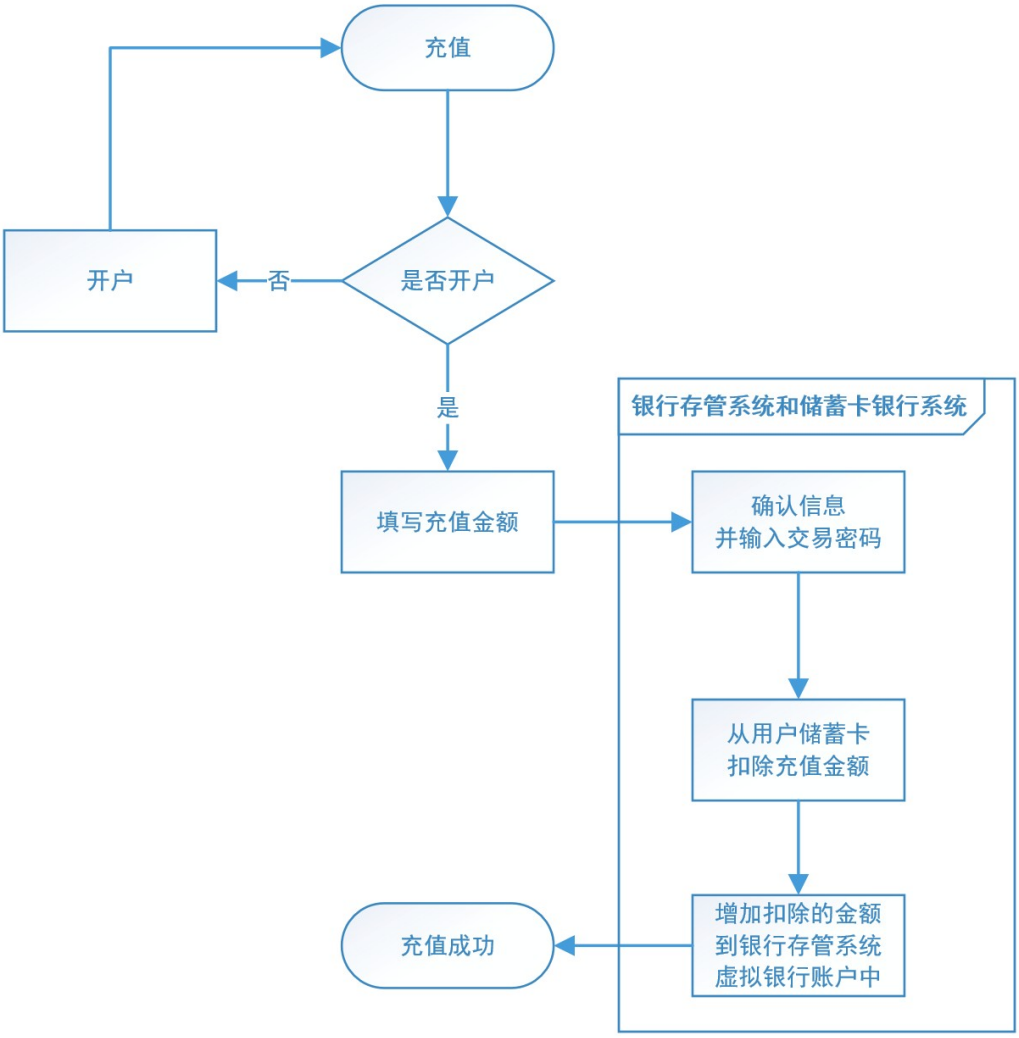
万信金融 第4章 讲义- 投标放款

1 业务概述

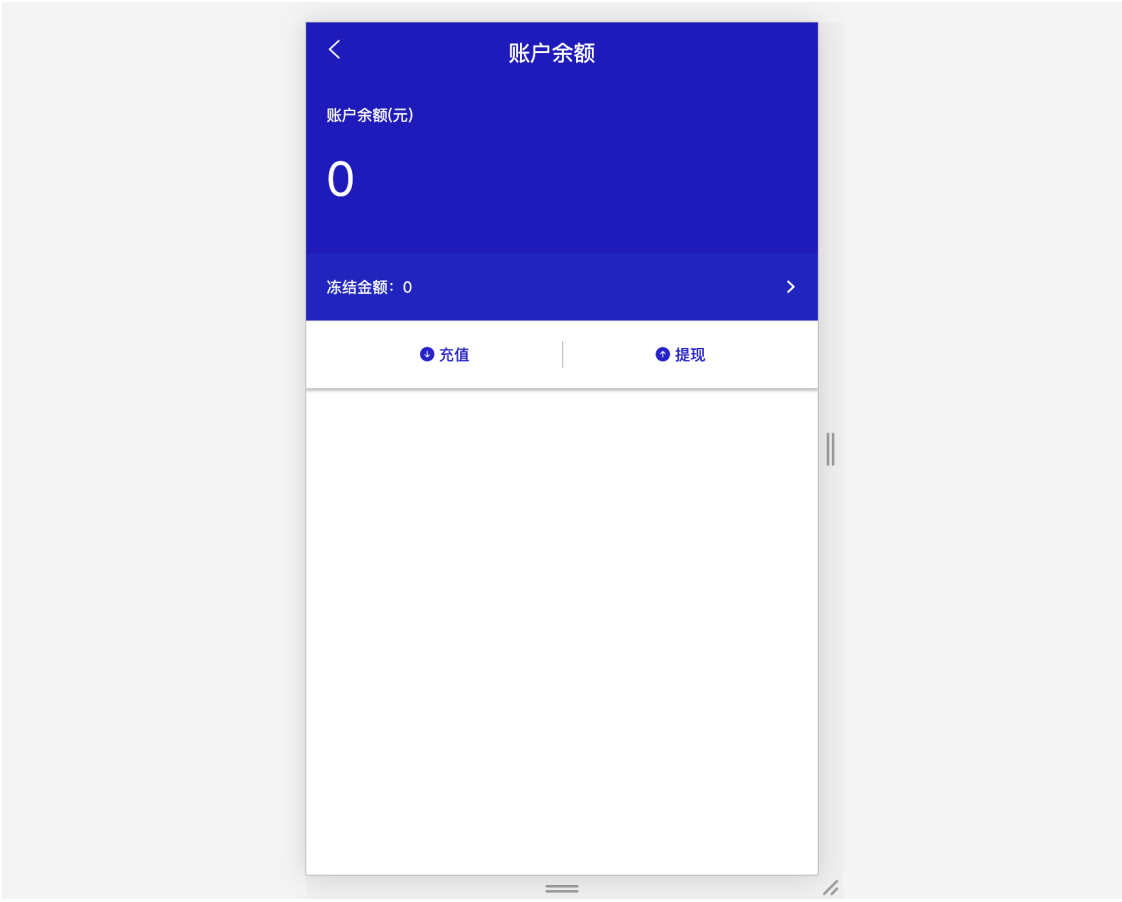
1.1 充值

用户在平台开户后会在银行存管系统中有一个虚拟银行账号，用户通过万信金融把储蓄卡中的金额转入到银行存管系统的虚拟银行账户中，即为充值。就好比你把银行卡中的金额转入到支付宝或微信中是一个道理。

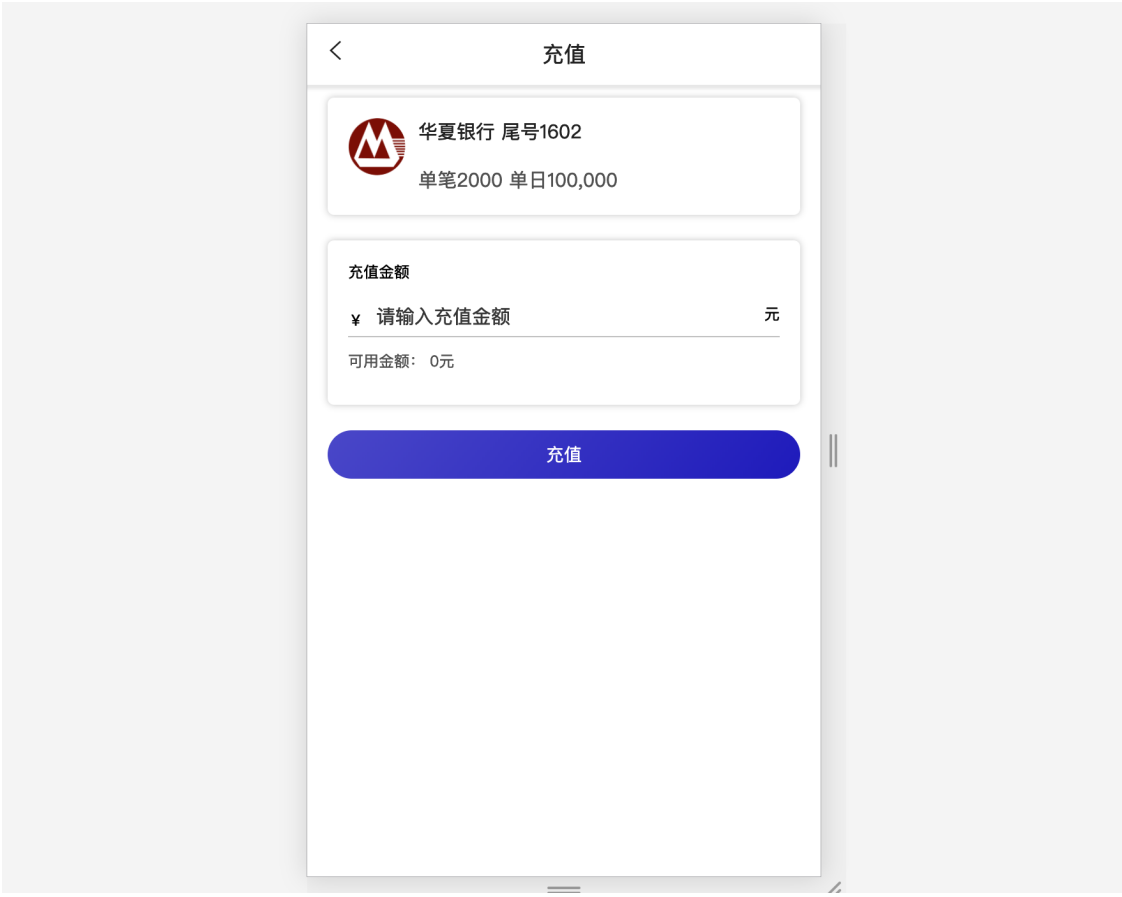
充值流程如下：



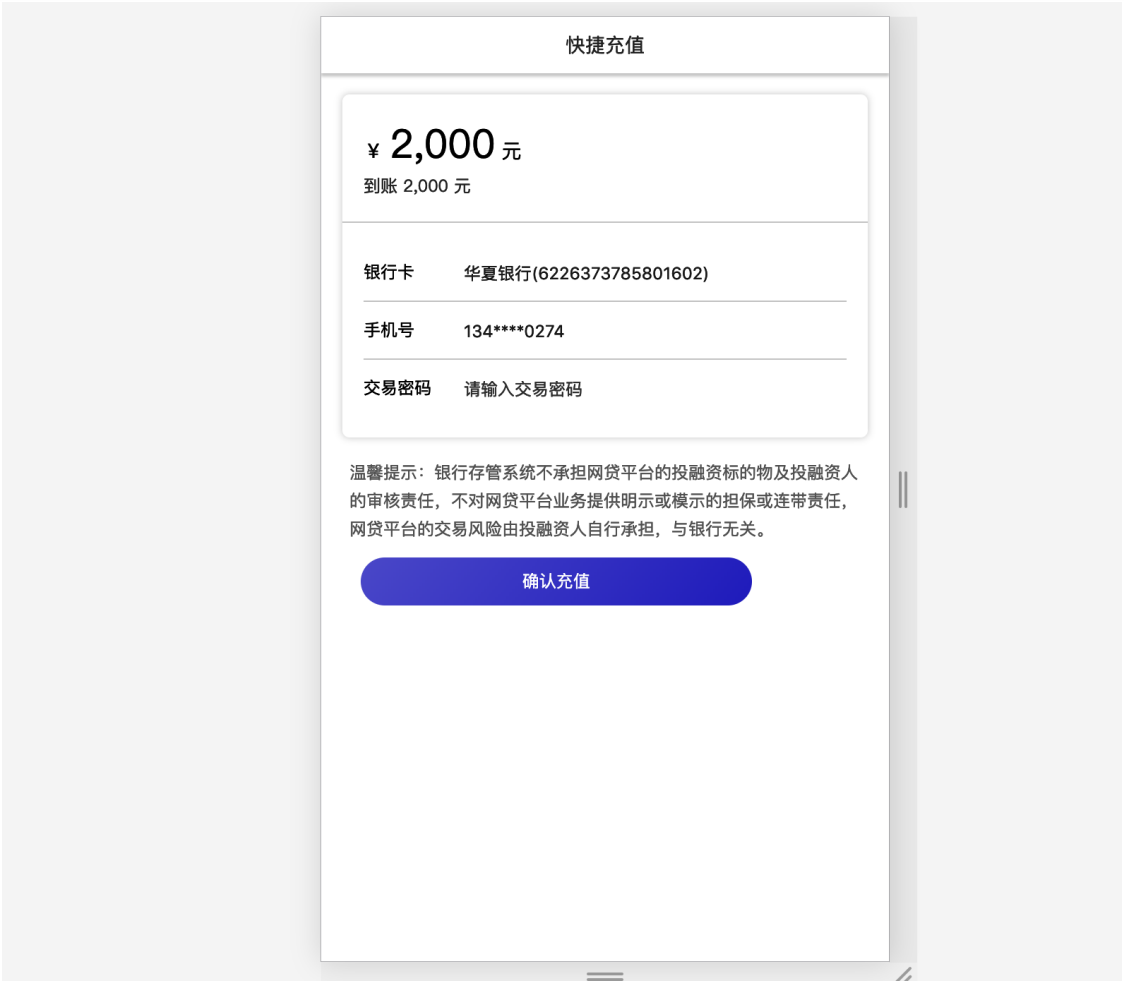
1. 用户在'我的'页面点击'账户余额'



2. 点击充值(如果用户尚未开户，会被自动跳转到开户界面)，填写充值金额

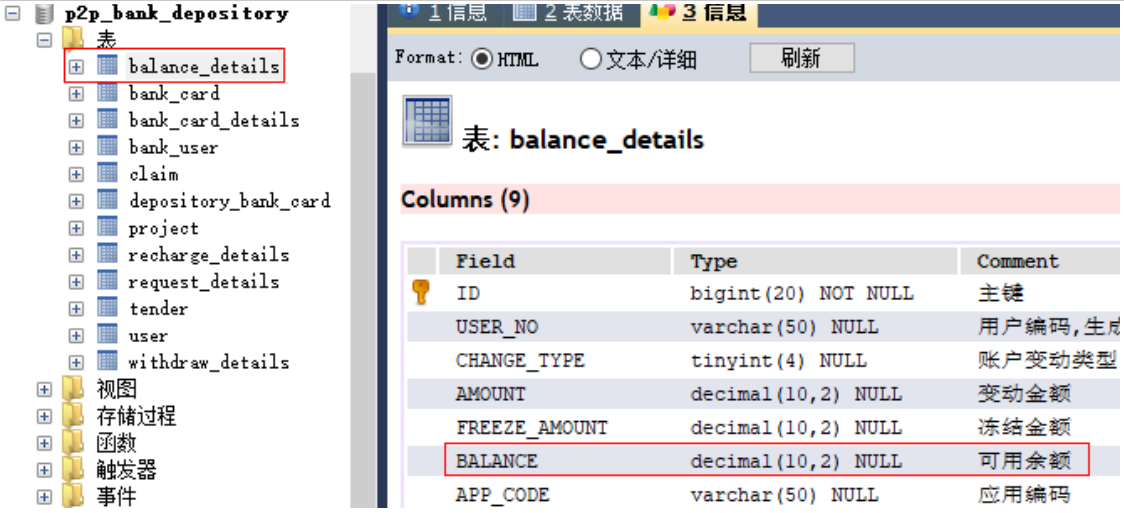


3. 确认信息并输入开户时设置的交易密码



4. 充值成功

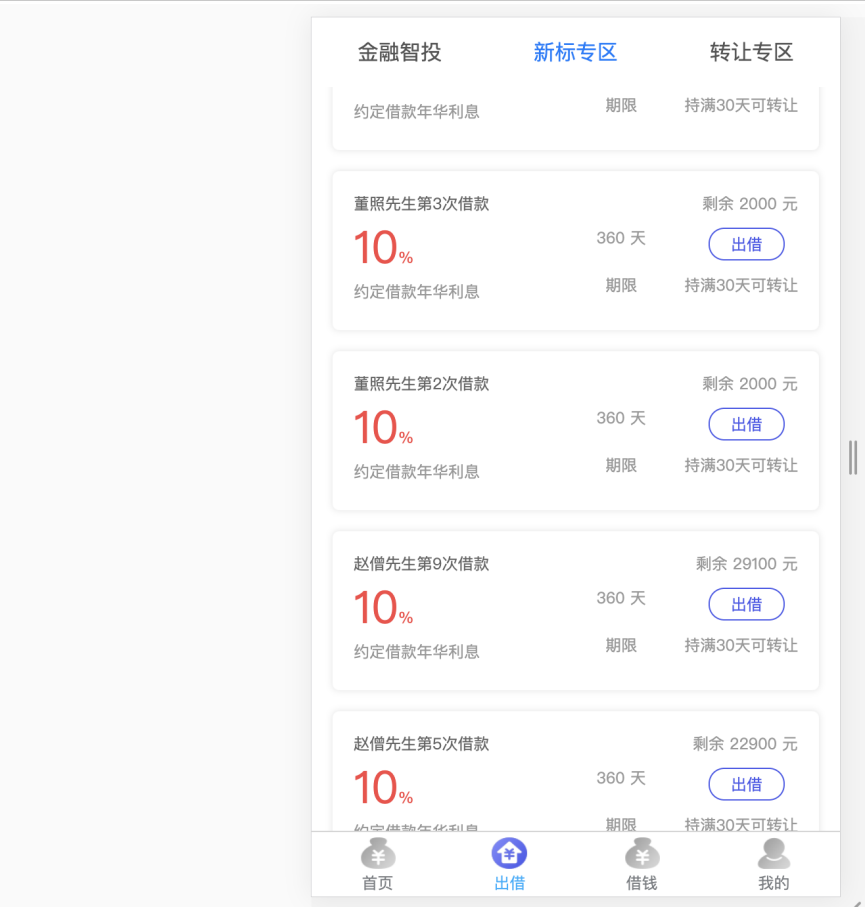
5. 目前暂时通过直接修改数据库的方式进行充值(后续会进行功能开发)



1.2 投标

借款人发标并通过审核后，投资人就可以在P2P平台看到这些标的信息(可投资项目)，投资人对这个项目进行投资(出借)就叫做投标。用户投标流程如下：

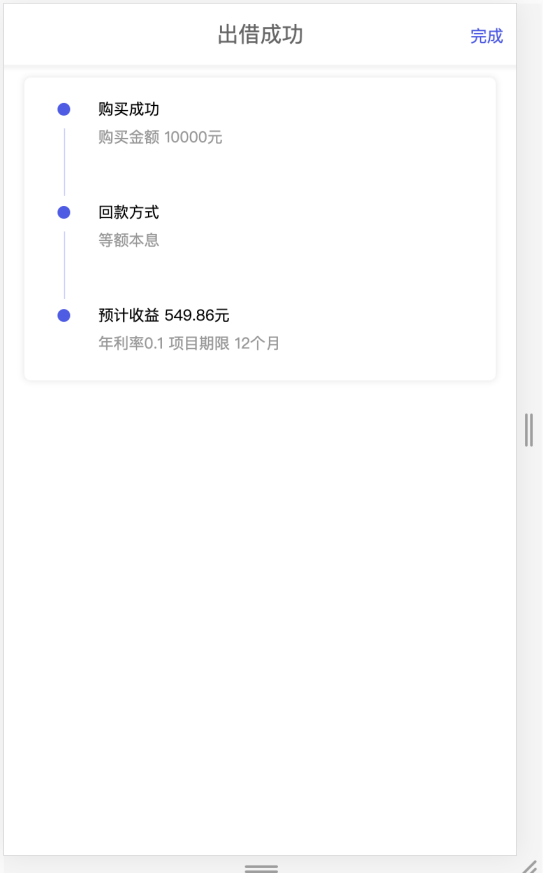
1. 用户浏览借款列表(标的)



2. 选择标的，输入投标金额

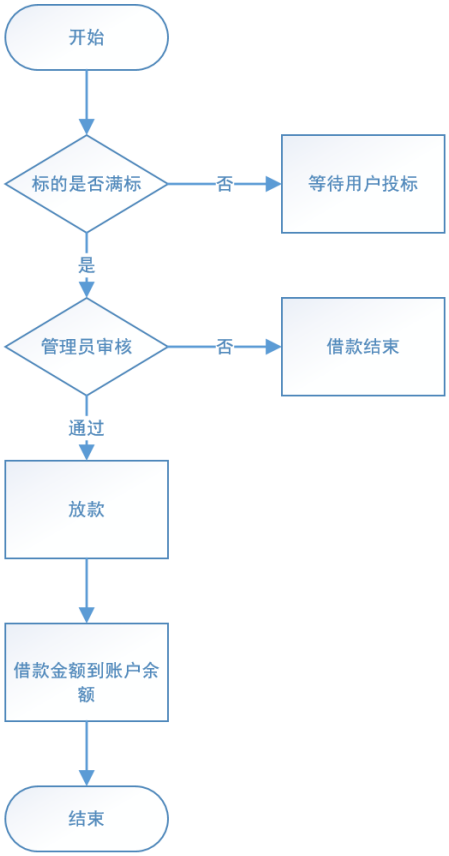
3. 确认出借，支付金额

4. 投标成功，显示预期收益



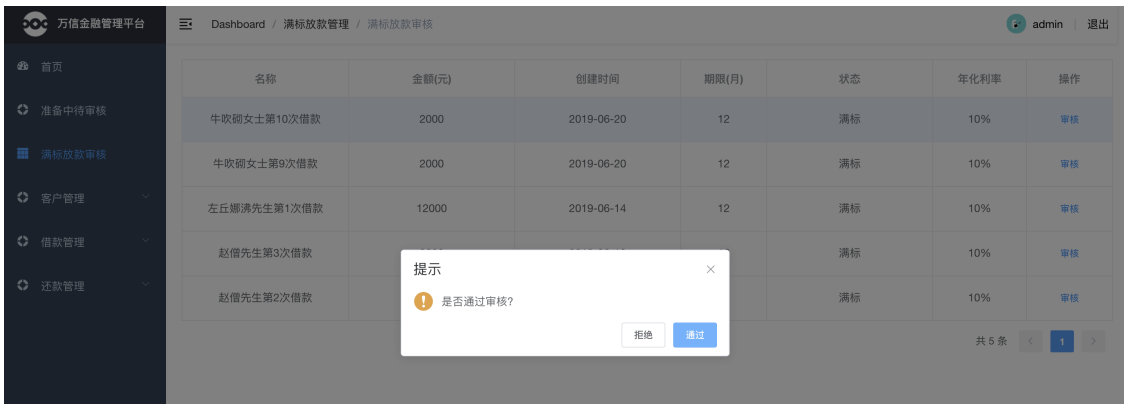
1.3 放款

当一个标的已经筹集到了所借的全部资金，即为“满标”。此时P2P平台管理员会进行审核，审核通过后，P2P平台会把投资人的出借资金打入借款人在平台的账户中，这就叫“放款”，此时借款人贷款成功。平台放款流程如下：



1. 管理员审核满标标的

2. 确认审核结果

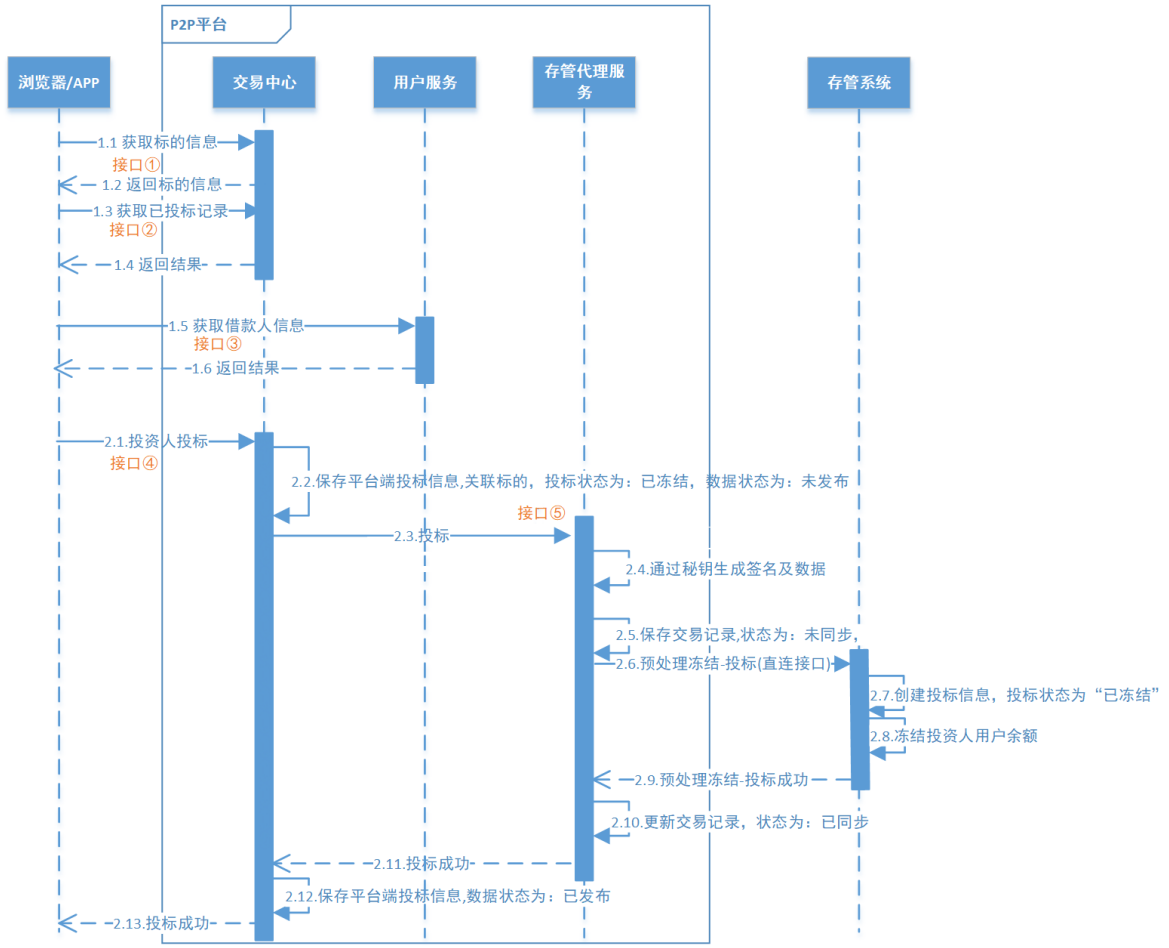


3. 审核通过，放款成功

2 用户投标

2.1 需求分析

借款人发标并通过审核后，投资人就可以在P2P平台看到这些标的信息(可投资项目)，投资人对这个项目进行投资(出借)就叫做投标。投资人在投标前需要先开户并充值。



第一阶段：投标预览(图中1.1-1.6)

1. 用户在前端选择要投资的标的
2. 请求交易中心获取标的基本信息和已投标记录
3. 交易中心请求用户服务获取借款人基本信息
4. 交易中心返回投标预览信息给前端
5. 前端显示投标预览信息，用户填写出借金额

第二阶段：用户投标(图中2.1-2.13)

1. 用户在前端确认投标信息，并请求交易中心保存投标信息
2. 交易中心保存用户投标信息(未发布)
3. 交易中心请求存管代理服务对投标数据进行签名，并生成交易记录(未同步)
4. 存管代理服务携带签名后的投标数据请求银行存管系统
5. 银行存管系统保存投标信息，并冻结投资人用户余额
6. 银行存管系统返回处理结果给存管代理服务
7. 存管代理服务更新交易记录(已同步)，并返回投标成功结果给交易中心
8. 交易中心更新投标结果后返回给前端
9. 前端展示投标结果给用户

2.2 投标预览

参考前面的流程图，投资人浏览标的列表时，可以点击某个标的进去预览相关信息，例如：标的信息，已投标信息，借款人信息等，这些都是投标前必须给投资人预览的信息。

2.2.1 接口定义

2.2.1.1 交易中心查询标的信息接口(接口①)

1、接口描述

1）根据标的id查询标的信息

2）获取标的剩余可投额度

3）获取标的已投记录数

2、接口定义

在TransactionApi接口中新增queryProjectsIds方法：

```
/**
 * 通过ids获取多个标的
 * @param ids
 * @return
*/
RestResponse<List<ProjectDTO>> queryProjectsIds(String ids);
```

在TransactionController类中实现该方法：

```
@Override
@ApiOperation("通过ids获取多个标的")
@GetMapping("/projects/{ids}")
public RestResponse<List<ProjectDTO>> queryProjectsIds(@PathVariable String ids)
{
    return null;
}
```

2.2.1.2 交易中心查询投标记录接口(接口②)

1、接口描述

1）根据标的id查询所有投标记录

2）封装投标记录列表返回

2、接口定义

在TransactionApi接口中新增queryTendersByProjectId方法：

```
/**
 * 根据标的id查询投标记录
 * @param id
 * @return
*/
RestResponse<List<TenderOverviewDTO>> queryTendersByProjectId(Long id);
```

在TransactionController类中实现该方法：

```
@Override
@ApiOperation("根据标的id查询投标记录")
@GetMapping("/tenders/projects/{id}")
public RestResponse<List<TenderOverviewDTO>> queryTendersByProjectId(
                                                    @PathVariable Long id){
    return null;
}
```

2.2.1.3 用户中心获取借款人信息接口(接口③)

1、接口描述

1）根据借款人id获取个人信息

2）返回借款人详细信息

2、接口定义

在ConsumerApi接口中新增getBorrower方法：

```
/**
 * 获取借款人用户信息
 * @param id
 * @return
*/
RestResponse<BorrowerDTO> getBorrower(Long id);
```

在ConsumerController类中实现该方法：

```
@Override
@ApiOperation("获取借款人用户信息")
@ApiImplicitParam(name = "id", value = "用户标识", required = true,
                            dataType = "Long", paramType = "path")
@GetMapping("/my/borrowers/{id}")
public RestResponse<BorrowerDTO> getBorrower(@PathVariable Long id){
    return null;
}
```

2.2.2 交易中心查询标的信息接口(接口①)

2.2.2.1 功能实现

1. 数据访问层

在mapper包中新增一个TenderMapper接口，用来操作投标数据：

```
/**
 * <p>
 * 投标信息表 Mapper 接口
 * </p>
 */
public interface TenderMapper extends BaseMapper<Tender> {
    /**
    * 根据标的id，获取标的已投金额，如果未投返回0.0
    * @param id
    * @return
    */
    @Select("SELECT IFNULL(SUM(AMOUNT), 0.0) FROM tender
        WHERE PROJECT_ID = #{id} AND STATUS = 1")
    List<BigDecimal> selectAmountInvestedByProjectId(Long id);
}
```

新建一个映射配置文件TenderMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.itcast.wanxinp2p.transaction.mapper.TenderMapper">
</mapper>
```

2. 业务层

在ProjectService接口中新增一个queryProjectsIds方法：

```
/**
 * 通过ids获取多个标的
 * @param ids
 * @return
*/
List<ProjectDTO> queryProjectsIds(String ids);
```

在ProjectServiceImpl类中实现该方法：

```
@Override
public List<ProjectDTO> queryProjectsIds(String ids) {
    //1.构造查询条件
    QueryWrapper<Project> queryWrapper = new QueryWrapper<>();
    List<Long> list = new ArrayList<>();
    Arrays.asList(ids.split(",")).forEach(str -> {
        list.add(Long.parseLong(str));
    });
    queryWrapper.lambda().in(Project::getId, list);
    //2.执行查询
    List<Project> projects = list(queryWrapper);
    List<ProjectDTO> dtos = new ArrayList<>();
    //3.实体转DTO并封装信息
    for (Project project : projects) {
        // 实体转换为DTO
```

```
            ProjectDTO projectDTO = convertProjectEntityToDTO(project);
            // 封装剩余额度
            projectDTO.setRemainingAmount(getProjectRemainingAmount(project));
            // 封装标的已投记录数
            projectDTO.setTenderCount(tenderMapper.selectCount(Wrappers
                                .<Tender>lambdaQuery().eq(Tender::getProjectId,
                                                        project.getId())));
            dtos.add(projectDTO);
        }
        return dtos;
    }
    /**
     * 获取标的剩余可投额度
     * @param project
     * @return
     */
    private BigDecimal getProjectRemainingAmount(Project project) {
        // 根据标的id在投标表查询已投金额
        List<BigDecimal> decimalList =
            tenderMapper.selectAmountInvestedByProjectId(project.getId());
        // 求和结果集
        BigDecimal amountInvested = new BigDecimal("0.0");
        for (BigDecimal d : decimalList) {
            amountInvested = amountInvested.add(d);
        }
        // 得到剩余额度
        return project.getAmount().subtract(amountInvested);
    }
```

3. 完善TransactionController类中的代码

```
@Override
@ApiOperation("通过ids获取多个标的")
@GetMapping("/projects/{ids}")
public RestResponse<List<ProjectDTO>> queryProjectsIds(@PathVariable String ids)
{
    List<ProjectDTO> projectDTOS = projectService.queryProjectsIds(ids);
    return RestResponse.success(projectDTOS);
}
```

2.2.2.2 功能测试

由于目前尚未实现投标功能，所以表中没有数据可用。暂时需要自己造点测试数据，这里需要注意：前面对p2p_transaction进行了分库分表，所以造测试数据时要依据分库分表策略进行，否则会影响测试效果。

分库策略：发标人ID % 2

分表策略：标的ID % 2

2.2.3 交易中心查询投标记录接口(接口②)

2.2.3.1 功能实现

1. 在ProjectService接口中新增queryTendersByProjectId方法：

```
/**
 * 根据标的id查询投标记录
 * @param id
 * @return
*/
List<TenderOverviewDTO> queryTendersByProjectId(Long id);
```

2. 在ProjectServiceImpl类中实现该方法：

```
@Override
public List<TenderOverviewDTO> queryTendersByProjectId(Long id) {
    List<Tender> tenderList = tenderMapper.selectList(Wrappers
                              .<Tender>lambdaQuery().eq(Tender::getProjectId,
id));
    List<TenderOverviewDTO> tenderOverviewDTOList = new ArrayList<>();
    tenderList.forEach(tender -> {
        TenderOverviewDTO tenderOverviewDTO = new TenderOverviewDTO();
        BeanUtils.copyProperties(tender, tenderOverviewDTO);
        tenderOverviewDTO.setConsumerUsername(CommonUtil

.hiddenMobile(tenderOverviewDTO.getConsumerUsername()));
        tenderOverviewDTOList.add(tenderOverviewDTO);
    });
    return tenderOverviewDTOList;
}
```

3. 完善TransactionController类中的代码：

```
@Override
@ApiOperation("根据标的id检索投标记录")
@GetMapping("/tenders/projects/{id}")
public RestResponse<List<TenderOverviewDTO>> queryTendersByProjectId(
                                                    @PathVariable Long id) {
    return RestResponse.success(projectService.queryTendersByProjectId(id));
}
```

2.2.3.2 功能测试

2.2.4 用户中心获取借款人信息接口(接口③)

2.2.4.1 功能实现

1. 在ConsumerService接口中新增getBorrower方法：

```
/**
 * 获取借款人基本信息
 * @param id
 * @return
*/
BorrowerDTO getBorrower(Long id);
```

2. 在ConsumerServiceImpl类中实现该方法：

```
@Override
public BorrowerDTO getBorrower(Long id) {
    ConsumerDTO consumerDTO = get(id);
    BorrowerDTO borrowerDTO = new BorrowerDTO();
    BeanUtils.copyProperties(consumerDTO, borrowerDTO);

    Map<String, String> cardInfo =
IDCardUtil.getInfo(borrowerDTO.getIdNumber());
    borrowerDTO.setAge(new Integer(cardInfo.get("age")));
    borrowerDTO.setBirthday(cardInfo.get("birthday"));
    borrowerDTO.setGender(cardInfo.get("gender"));

    return borrowerDTO;
}
private ConsumerDTO get(Long id) {
    Consumer entity = getById(id);
    if (entity == null) {
        log.info("id为{}的用户信息不存在", id);
        throw new BusinessException(ConsumerErrorCode.E_140101);
    }
    return convertConsumerEntityToDTO(entity);
}
```

3. 完善ConsumerController类中的代码：

```
@Override
@ApiOperation("获取借款人用户信息")
@ApiImplicitParam(name = "id", value = "用户标识", required = true,
                                dataType = "Long", paramType = "path")
@GetMapping("/my/borrowers/{id}")
public RestResponse<BorrowerDTO> getBorrower(@PathVariable Long id) {
    return RestResponse.success(consumerService.getBorrower(id));
}
```

2.2.4.2 功能测试

2.2.5 前后端集成测试

2.3 用户投标

参考前面的流程图，该业务涉及到交易中心，存管代理和银行存管系统。交易中心需要保存投标信息，存管代理需要签名数据并保存交易记录，银行存管系统需要保存标的信息并扣除投资人余额。

2.3.1 接口定义

2.3.1.1 交易中心保存投标信息接口(接口④)

在交易中心定义保存投标信息接口：

1、接口描述

1）接受用户填写的投标信息

2）交易中心校验投资金额是否符合平台允许最小投资金额

3）校验用户余额是否大于投资金额

4）校验投资金额是否小于等于标的可投金额

5）校验此次投标后的剩余金额是否满足最小投资金额

6）保存投标信息

7）请求存管代理服务进行投标预处理冻结

8）存管代理服务返回处理结果给交易中心，交易中心计算此次投标预期收益

9）返回预期收益给前端

2、接口定义

在TransactionApi接口中新增createTender方法：

```
/**
 * 用户投标
 * @param projectInvestDTO
 * @return
*/
RestResponse<TenderDTO> createTender(ProjectInvestDTO projectInvestDTO);
```

在TransactionController类中实现该方法：

```
@Override
@ApiOperation("用户投标")
@ApiImplicitParam(name = "projectInvestDTO", value = "投标信息",
                  required = true, dataType = "ProjectInvestDTO", paramType =
"body")
@PostMapping("/my/tenders")
public RestResponse<TenderDTO> createTender(@RequestBody ProjectInvestDTO
projectInvestDTO){
    return null;
}
```

2.3.1.2 存管代理服务预处理冻结接口(接口⑤)

在存管代理服务中创建预处理冻结接口：

1、接口描述

1）保存交易记录

2）请求银行存管系统进行预处理冻结

3）返回处理结果给交易中心

2、接口定义

在DepositoryAgentApi接口中新增userAutoPreTransaction方法：

```
/**
 * 预授权处理
 * @param userAutoPreTransactionRequest 预授权处理信息
 * @return
 */
RestResponse<String> userAutoPreTransaction(UserAutoPreTransactionRequest
userAutoPreTransactionRequest);
```

在DepositoryAgentController类中实现该方法：

```
@Override
@ApiOperation(value = "预授权处理")
@ApiImplicitParam(name = "userAutoPreTransactionRequest",
                  value = "平台向存管系统发送标的信息", required = true,
                  dataType = "UserAutoPreTransactionRequest", paramType =
"body")
@PostMapping("/l/user-auto-pre-transaction")
public RestResponse<String> userAutoPreTransaction(@RequestBody
UserAutoPreTransactionRequest userAutoPreTransactionRequest){
    return null;
}
```

2.3.2 交易中心用户投标接口(接口④)

2.3.2.1 功能实现

1.在ProjectService接口中新增createTender方法：

```
/**
 * 用户投标
 * @param projectInvestDTO
 * @return
*/
TenderDTO createTender(ProjectInvestDTO projectInvestDTO);
```

2.在ProjectServiceImpl类中实现该方法，由于业务非常复杂，我们拆分成若干部分逐一实现

2.1 前置条件判断准备工作(投标金额是否大于最小投标金额、账户余额是否足够)

A：检查Apollo上是否配置最小投标金额

已发布	mini.investment.amount	100.0	最小投资金额

B：实现查询账户余额功能，在ConsumerAPI接口中新增查询当前余额的方法：

```
/**
    获取当前登录用户余额信息
    @param userNo 用户编码
    @return
*/
RestResponse<BalanceDetailsDTO> getBalance(String userNo);
```

C：在ConsumerController类中实现该方法：

```
@Override
@ApiOperation("获取用户可用余额")
@ApiImplicitParam(name = "userNo", value = "用户编码", required = true,
                  dataType = "String")
@GetMapping("/l/balances/{userNo}")
public RestResponse<BalanceDetailsDTO> getBalance(@PathVariable String userNo) {
    RestResponse<BalanceDetailsDTO> balanceFromDepository =
                                    getBalanceFromDepository(userNo);
    return balanceFromDepository;
}

@Value("${depository.url}")
private String depositoryURL;

private OkHttpClient okHttpClient=new OkHttpClient().newBuilder().build();

/**
远程调用存管系统获取用户余额信息
@param userNo 用户编码
@return
*/
//不用大家编码实现，直接复制使用即可
private RestResponse<BalanceDetailsDTO> getBalanceFromDepository(String userNo)
{
    String url = depositoryURL + "/balance-details/" + userNo;
    BalanceDetailsDTO balanceDetailsDTO;
    Request request = new Request.Builder().url(url).build();
    try (Response response = okHttpClient.newCall(request).execute()) {
        if (response.isSuccessful()) {
            String responseBody = response.body().string();
            balanceDetailsDTO = JSON.parseObject(responseBody,
BalanceDetailsDTO.class);
            return RestResponse.success(balanceDetailsDTO);
        }
    } catch (IOException e) {
        log.warn("调用存管系统{}获取余额失败 ", url, e);
    }
    return RestResponse.validfail("获取失败");
}
```

D：在ConsumerApiAgent接口中新增getBalance方法，供交易中心远程查询当前余额

```
@GetMapping("/consumer/l/balances/{userNo}")
public RestResponse<BalanceDetailsDTO> getBalance(@PathVariable("userNo")
                                                   String userNo)
```

E：考虑到前端也需要获取账户余额，所以在ConsumerAPI接口中新增供前端使用的查询当前余额的方法：

```
/**
 * 获取当前登录用户余额信息
 * @return
 */
RestResponse<BalanceDetailsDTO> getMyBalance();
```

F：在ConsumerController类中实现该方法：

```
@Override
@GetMapping("/my/balances")
public RestResponse<BalanceDetailsDTO> getMyBalance() {
    ConsumerDTO consumerDTO = consumerService
            .getByMobile(SecurityUtil.getUser().getMobile());
    return getBalanceFromDepository(consumerDTO.getUserNo());
}
```

2.2 实现前置条件判断

```
//1.前置条件判断
//1.1 判断投标金额是否大于最小投标金额
//获得投标金额
BigDecimal amount=new BigDecimal(projectInvestDTO.getAmount());
//获得最小投标金额
BigDecimal miniInvestmentAmount=configService.getMiniInvestmentAmount();
if(amount.compareTo(miniInvestmentAmount)<0){
    throw  new BusinessException(TransactionErrorCode.E_150109);
}
//1.2 判断用户账户余额是否足够
//得到当前登录用户
LoginUser user=SecurityUtil.getUser();
//通过手机号查询用户信息
RestResponse<ConsumerDTO>
restResponse=consumerApiAgent.getCurrConsumer(user.getMobile());
//通过用户编号查询账户余额
RestResponse<BalanceDetailsDTO>
balanceDetailsDTORestResponse=consumerApiAgent.getBalance(restResponse.getResult
().getUserNo());
BigDecimal myBalance=balanceDetailsDTORestResponse.getResult().getBalance();
if(myBalance.compareTo(amount)<0){
    throw  new BusinessException(TransactionErrorCode.E_150112);
}
```

2.3 前置条件判断(投标金额是否超出所剩可投金额、是否满标等)

```
//1.3 判断标的是否满标，标的状态为FULLY就表示满标
Project project = getById(projectInvestDTO.getId());
```

```
if(project.getProjectStatus().equalsIgnoreCase("FULLY")){
    throw  new BusinessException(TransactionErrorCode.E_150114);
}

//1.4 判断投标金额是否超过剩余未投金额
BigDecimal remainingAmount = getProjectRemainingAmount(project);
if(amount.compareTo(remainingAmount)<1){
    //1.5 判断此次投标后的剩余未投金额是否满足最小投标金额
    //例如：借款人需要借1万    现在已经投标了8千    还剩2千    本次投标1950元
    //公式：此次投标后的剩余未投金额 = 目前剩余未投金额 - 本次投标金额
    BigDecimal subtract=remainingAmount.subtract(amount);
    int result=subtract.compareTo(configService.getMiniInvestmentAmount());
    if(result<0){
        throw new BusinessException(TransactionErrorCode.E_150111);
    }

    //2. 保存投标信息并发送给存管代理服务

    //3. 根据结果更新投标状态

}else{
    throw new BusinessException(TransactionErrorCode.E_150110);
}
```

2.4 保存投标信息并发送给存管代理服务

A : 在DepositoryAgentApiAgent接口中新增userAutoPreTransaction方法:

```
@PostMapping("/depository-agent/1/user-auto-pre-transaction")
RestResponse<String> userAutoPreTransaction(
            UserAutoPreTransactionRequest userAutoPreTransactionRequest);
```

B : 实现保存投标信息并发送给存管代理服务

```
//2.1 保存投标信息, 数据状态为: 未发布
// 封装投标信息
final Tender tender = new Tender();
// 投资人投标金额( 投标冻结金额 )
tender.setAmount(amount);
// 投标人用户标识
tender.setConsumerId(restResponse.getResult().getId());
tender.setConsumerUsername(restResponse.getResult().getUsername());
// 投标人用户编码
tender.setUserNo(restResponse.getResult().getUserNo());
// 标的标识
tender.setProjectId(projectInvestDTO.getId());
// 标的编码
tender.setProjectNo(project.getProjectNo());
// 投标状态
tender.setTenderStatus(TradingCode.FROZEN.getCode());
// 创建时间
tender.setCreateDate(LocalDateTime.now());
// 请求流水号
tender.setRequestNo(CodeNoUtil.getNo(CodePrefixCode.CODE_REQUEST_PREFIX));
// 可用状态
tender.setStatus(0);
tender.setProjectName(project.getName());
```

```
// 标的期限(单位:天)
tender.setProjectPeriod(project.getPeriod());
// 年化利率(投资人视图)
tender.setProjectAnnualRate(project.getAnnualRate());
// 保存到数据库
tenderMapper.insert(tender);

//2.2 发送数据给存管代理服务
// 构造请求数据
UserAutoPreTransactionRequest userAutoPreTransactionRequest = new
UserAutoPreTransactionRequest();
// 冻结金额
userAutoPreTransactionRequest.setAmount(amount);
// 预处理业务类型
userAutoPreTransactionRequest.setBizType(PreprocessBusinessTypeCode.TENDER.getCo
de());
// 标的号
userAutoPreTransactionRequest.setProjectNo(project.getProjectNo());
// 请求流水号
userAutoPreTransactionRequest.setRequestNo(tender.getRequestNo());
// 投资人用户编码
userAutoPreTransactionRequest.setUserNo(restResponse.getResult().getUserNo());
// 设置 关联业务实体标识
userAutoPreTransactionRequest.setId(tender.getId());

// 远程调用存管代理服务
RestResponse<String> response = depositoryAgentApiAgent
        .userAutoPreTransaction(userAutoPreTransactionRequest);
```

2.5 根据结果修改投标状态

```
// 3.1 判断结果
if (DepositoryReturnCode.RETURN_CODE_00000.getCode()
        .equals(response.getResult())) {
    // 3.2 修改状态为: 已发布
    tender.setStatus(1);
    tenderMapper.updateById(tender);
    // 3.3 投标成功后判断标的是否已投满，如果满标，更新标的状态
    BigDecimal remainAmount=getProjectRemainingAmount(project);
    if (remainAmount.compareTo(new BigDecimal(0)) == 0) {
        project.setProjectStatus(ProjectCode.FULLY.getCode());
        updateById(project);
    }

    // 3.4 转换为dto对象并封装数据
    TenderDTO tenderDTO = convertTenderEntityToDTO(tender);
    // 封装标的信息
    project.setRepaymentWay(RepaymentWayCode.FIXED_REPAYMENT.getDesc());
    tenderDTO.setProject(convertProjectEntityToDTO(project));
    // 封装预期收益
    // 根据标的期限计算还款月数
    final Double ceil = Math.ceil(project.getPeriod() / 30.0);
    Integer month = ceil.intValue();
    //计算预期收益
    tenderDTO.setExpectedIncome(IncomeCalcUtil
            .getIncomeTotalInterest(new
BigDecimal(projectInvestDTO.getAmount()),
```

```
                    configService.getAnnualRate(), month));
    return tenderDTO;
} else {
    // 抛出一个业务异常
    log.warn("投标失败 ！ 标的ID为: {}, 存管代理服务返回的状态为: {}",
            projectInvestDTO.getId(), restResponse.getResult());
    throw new BusinessException(TransactionErrorCode.E_150113);
}

private TenderDTO convertTenderEntityToDTO(Tender tender) {
    if (tender == null) {
        return null;
    }
    TenderDTO tenderDTO = new TenderDTO();
    BeanUtils.copyProperties(tender, tenderDTO);
    return tenderDTO;
}
```

3. 完善TransactionController类中的代码：

```
@Override
@ApiOperation("用户投标")
@ApiImplicitParam(name = "projectInvestDTO", value = "投标信息",
                  required = true, dataType = "ProjectInvestDTO", paramType =
"body")
@PostMapping("/my/tenders")
public RestResponse<TenderDTO> createTender(@RequestBody ProjectInvestDTO
projectInvestDTO) {
    TenderDTO tenderDTO = projectService.createTender(projectInvestDTO);
    return RestResponse.success(tenderDTO);
}
```

2.3.2.2 功能测试

可以通过Postman结合断点调试进行测试

2.3.3 存管代理服务预处理冻结接口(接口⑤)

2.3.3.1 银行存管系统接口说明

和前面开户一样，这里需要提供“存管系统接口”四大参数，详情请参考“银行存管系统接口说明.pdf”。

2.3.3.2 功能实现

1. 在DepositoryRecordService接口中新增userAutoPreTransaction方法：

```
/**
 * 投标预处理
 * @param userAutoPreTransactionRequest
 * @return
*/
DepositoryResponseDTO<DepositoryBaseResponse>
userAutoPreTransaction(UserAutoPreTransactionRequest
userAutoPreTransactionRequest);
```

2. 在DepositoryRecordServiceImpl类中实现该方法：

```
@Override
public DepositoryResponseDTO<DepositoryBaseResponse> userAutoPreTransaction(
        UserAutoPreTransactionRequest userAutoPreTransactionRequest) {
    DepositoryRecord depositoryRecord = new
            DepositoryRecord(userAutoPreTransactionRequest.getRequestNo(),
                             userAutoPreTransactionRequest.getBizType(),
                                     "UserAutoPreTransactionRequest",
                                 userAutoPreTransactionRequest.getId());

    // 幂等性实现
    DepositoryResponseDTO<DepositoryBaseResponse> responseDTO =
                                     handleIdempotent(depositoryRecord);
    if (responseDTO != null) {
        return responseDTO;

    // 根据requestNo获取交易记录
    depositoryRecord =
            getEntityByRequestNo(userAutoPreTransactionRequest.getRequestNo());

    // userAutoPreTransactionRequest 转为 json 用于数据签名
    final String jsonString = JSON.toJSONString(userAutoPreTransactionRequest);
    // 业务数据报文，对json数据进行base64编码处理方便传输
```

```
    String reqData = EncryptUtil.encodeUTF8StringBase64(jsonString);
    // 发送请求，获取结果
    // 拼接银行存管系统请求地址
    String url = configService.getDepositoryUrl() + "/service";
    // 向银行存管系统发送请求
    return sendHttpGet("USER_AUTO_PRE_TRANSACTION", url, reqData,
depositoryRecord);
}
```

3. 完善DepositoryAgentController类中的代码：

```
@Override
@ApiOperation(value = "预授权处理")
@ApiImplicitParam(name = "userAutoPreTransactionRequest",
                  value = "向银行存管系统发送投标信息", required = true,
                  dataType = "UserAutoPreTransactionRequest", paramType =
"body")
@PostMapping("/l/user-auto-pre-transaction")
public RestResponse<String> userAutoPreTransaction(@RequestBody
UserAutoPreTransactionRequest userAutoPreTransactionRequest) {
    DepositoryResponseDTO<DepositoryBaseResponse> depositoryResponse =


 depositoryRecordService.userAutoPreTransaction(userAutoPreTransactionRequest);
    return getRestResponse(depositoryResponse);
}

/**
 * 统一处理响应信息
 * @param depositoryResponse
 * @return
 */
private RestResponse<String>
getRestResponse(DepositoryResponseDTO<DepositoryBaseResponse>
depositoryResponse) {
    final RestResponse<String> restResponse = new RestResponse<>();
    restResponse.setResult(depositoryResponse.getRespData().getRespCode());
    restResponse.setMsg(depositoryResponse.getRespData().getRespMsg());
    return restResponse;
}
```

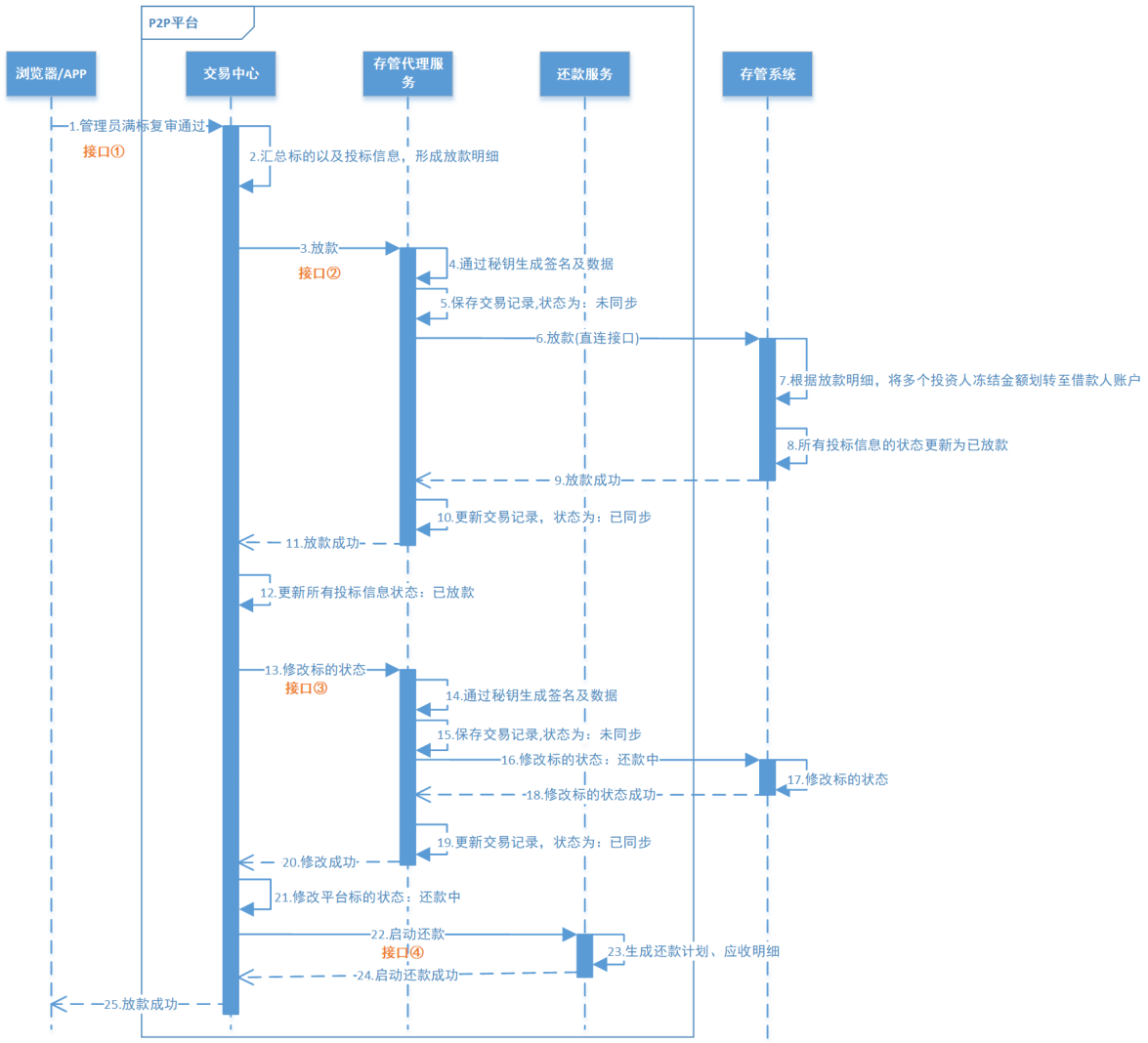
2.3.4 前后端集成测试

3 放款

3.1 需求分析

当一个标的已经筹集到了所借的全部资金，即为“满标”。此时P2P平台管理员会进行审核，审核通过后，P2P平台会把投资人的出借资金打入借款人在平台的账户中，这就叫“放款”，此时就表示借款人贷款成功。

系统交互流程如下图所示：



在该业务中会新增一个还款微服务：为平台提供还款计划的生成、执行、记录与归档等功能。

第一阶段：生成放款明细(图中1-2)

1. 前端向交易中心发起满标复审通过请求
2. 交易中心汇总标的以及投标信息，形成放款明细

第二阶段：放款(图中3-12)

1. 交易中心请求存管代理服务进行放款操作
2. 存管代理服务对放款明细进行签名，并保存交易记录为未同步
3. 存管代理服务请求存管系统进行放款
4. 存管系统根据放款明细，将多个投资人冻结余额划至借款人账户，并更新投标信息状态为已放款
5. 返回放款结果给存管代理服务
6. 存管代理服务更新交易记录为已同步，并返回放款结果给交易中心
7. 交易中心更新所有投标信息结果为：已放款

第三阶段：修改标的业务状态(图中13-21)

1. 交易中心请求存管代理服务修改标的状态
2. 存管代理服务对数据进行签名，并保存交易记录为：未同步
3. 存管代理服务请求存管系统修改标的状态
4. 存管系统修改标的为状态为：已放款，并返回结果给存管代理服务
5. 存管代理服务更新交易记录为：已同步，返回修改成功给交易中心
6. 交易中心收到返回结果，修改标的状态为：还款中

第四阶段：启动还款(图中22-25)

1. 交易中心请求还款服务启动还款
2. 还款服务收到请求后生成还款计划和应收明细，并返回启动还款成功给交易中心
3. 交易中心返回放款成功给前端

3.2 满标放款

3.2.1 接口定义

3.2.1.1 交易中心满标放款接口(接口①)

1、接口描述

1）接受前端满标放款信息

2）交易中心根据标的信息生成还款明细

3）交易中心请求存管代理服务进行满标放款

4）交易中心收到放款成功结果后，更新投标信心状态为：已放款

5）交易中心请求存管代理服务修改标的状态

6）交易中心请求还款服务启动放款

2、接口定义

在TransactionApi接口中新增loansApprovalStatus方法：

```
/**
 * 审核标的满标放款
 *
 * @param id  标的id
 * @param approveStatus  审核状态
 * @param commission  平台佣金
 * @return
 */
RestResponse<String> loansApprovalStatus(Long id, String approveStatus, String
commission);
```

在TransactionController类中实现该方法：

```
@Override
@ApiOperation("审核标的满标放款")
@ApiImplicitParams({
    @ApiImplicitParam(name = "id", value = "标的id", required = true,
                      dataType = "long", paramType = "path"),
    @ApiImplicitParam(name = "approveStatus", value = "标的审核状态", required =
true,
                      dataType = "string", paramType = "path"),
    @ApiImplicitParam(name = "commission", value = "平台佣金", required = true,
                      dataType = "string", paramType = "query")
})
@PutMapping("/m/loans/{id}/projectStatus/{approveStatus}")
```

```
public RestResponse<String> loansApprovalStatus(
                                @PathVariable("id") Long id,
                                @PathVariable("approveStatus") String
approveStatus,
                                String commission){
    return null;
}
```

3.2.1.2 存管代理服务确认放款接口(接口②)

1、接口描述

1）接受确认放款数据

2）保存请求记录

3）生成签名数据

4）请求存管系统进行确认放款

2、接口定义

在DepositoryAgentApi接口中新增confirmLoan方法：

```
/**
 * 审核标的满标放款
 * @param loanRequest
 * @return
 */
RestResponse<String> confirmLoan(LoanRequest loanRequest);
```

在DepositoryAgentController类中实现该方法：

```
@Override
@ApiOperation(value = "审核标的满标放款")
@ApiImplicitParam(name = "loanRequest", value = "标的满标放款信息", required =
true, dataType = "LoanRequest", paramType = "body")
@PostMapping("l/confirm-loan")
public RestResponse<String> confirmLoan(@RequestBody LoanRequest loanRequest){
    return null;
}
```

3.2.1.3 存管代理服务修改标的状态接口(接口③)

1、接口描述

1）接受修改标的状态数据

2）保存请求记录

3）生成签名数据

4）请求存管系统修改标的状态

2、接口定义

在DepositoryAgentApi接口中新增modifyProjectStatus方法：

```
/**
 * 修改标的状态
 * @param modifyProjectStatusDTO
 * @return
 */
RestResponse<String> modifyProjectStatus(ModifyProjectStatusDTO
modifyProjectStatusDTO);
```

在DepositoryAgentController类中实现该方法：

```
@Override
@ApiOperation(value = "修改标的状态")
@ApiImplicitParam(name = "modifyProjectStatusDTO", value = "修改标的状态DTO",
                  required = true, dataType = "ModifyProjectStatusDTO",
                  paramType = "body")
@PostMapping("l/modify-project-status")
public RestResponse<String> modifyProjectStatus(
    @RequestBody ModifyProjectStatusDTO modifyProjectStatusDTO){
    return null;
}
```

3.2.1.4 还款服务启动还款接口(接口④)

1、搭建还款微服务工程

1）导入工程

2）配置启动参数：-Denv=dev -Dapollo.cluster=DEFAULT -Dserver.port=53080

3）在Apollo上新建repayment-service项目并进行基础配置

2、接口描述

1）接受交易中心的还款信息

2）生成借款人还款计划，保存到数据库

3）生成投资人应收明细，保存到数据库

3、接口定义

在wanxinp2p-api工程中新建repayment包，在该包中新建RepaymentApi接口，并定义startRepayment方法：

```
/**
 * 启动还款
 * @param projectWithTendersDTO
 * @return
 */
public RestResponse<String> startRepayment(ProjectWithTendersDTO
projectWithTendersDTO);
```

在RepaymentController类中实现该方法：

```
@Override
@ApiOperation("启动还款")
@ApiImplicitParam(name = "projectWithTendersDTO", value = "通过id获取标的信息",
                  required = true, dataType = "ProjectWithTendersDTO",
                  paramType = "body")
@PostMapping("/l/start-repayment")
public RestResponse<String> startRepayment(@RequestBody ProjectWithTendersDTO
projectWithTendersDTO) {
    return null;
}
```

3.2.2 交易中心满标放款接口(接口①)

1. 在DepositoryAgentApiAgent接口中新增两个方法：

```
@PostMapping("/depository-agent/l/confirm-loan")
RestResponse<String> confirmLoan(LoanRequest loanRequest);

@PostMapping("/depository-agent/l/modify-project-status")
RestResponse<String> modifyProjectStatus(ModifyProjectStatusDTO
modifyProjectStatusDTO);
```

2. 在ProjectService接口中新增loansApprovalStatus方法：

```
/**
 * 审核标的满标放款
 * @param id
 * @param approveStatus
 * @param commission
 * @return String
 */
String loansApprovalStatus(Long id, String approveStatus, String commission);
```

3. 在ProjectServiceImpl类中实现该方法：

```
@Override
@Transactional(rollbackFor = BusinessException.class)
public String loansApprovalStatus(Long id, String approveStatus, String
commission) {
    // 第一阶段：生成放款明细
    // 获取标的信息
    final Project project = getById(id);
```

```
    // 构造查询参数，获取所有投标信息
    QueryWrapper<Tender> queryWrapper = new QueryWrapper<>();
    queryWrapper.lambda().eq(Tender::getProjectId, id);
    final List<Tender> tenderList = tenderMapper.selectList(queryWrapper);
    // 生成还款明细
    final LoanRequest loanRequest = generateLoanRequest(project, tenderList,
commission);

    // 第二阶段：放款
    // 请求存管代理服务
    final RestResponse<String> restResponse = depositoryAgentApiAgent
                                                    .confirmLoan(loanRequest);
    if(DepositoryReturnCode.RETURN_CODE_00000.getCode()
        .equals(restResponse.getResult())) {
        // 响应成功，更新投标信息：已放款
        updateTenderStatusAlreadyLoan(tenderList);

        // 第三阶段：修改标的业务状态
        // 调用存管代理服务，修改状态为还款中
        // 构造请求参数
        ModifyProjectStatusDTO modifyProjectStatusDTO = new
ModifyProjectStatusDTO();
        // 业务实体id
        modifyProjectStatusDTO.setId(project.getId());
        // 业务状态
        modifyProjectStatusDTO.setProjectStatus(ProjectCode.REPAYING.getCode());
        // 请求流水号
        modifyProjectStatusDTO.setRequestNo(loanRequest.getRequestNo());
        // 执行请求
        final RestResponse<String> modifyProjectProjectStatus =
depositoryAgentApiAgent
                .modifyProjectProjectStatus(modifyProjectStatusDTO);
        if (DepositoryReturnCode.RETURN_CODE_00000.getCode()
                            .equals(modifyProjectProjectStatus.getResult())) {
            //如果处理成功，就修改标的状态为还款中
            project.setProjectStatus(ProjectCode.REPAYING.getCode());
            updateById(project);
            // 第四阶段：启动还款
            // 封装调用还款服务请求对象的数据
            ProjectWithTendersDTO projectWithTendersDTO = new
ProjectWithTendersDTO();
            // 封装标的信息

 projectWithTendersDTO.setProject(convertProjectEntityToDTO(project));
            // 封装投标信息
            projectWithTendersDTO.setTenders(

convertTenderEntityListToDTOList(tenderList));
            // 封装投资人让利
            projectWithTendersDTO.setCommissionInvestorAnnualRate(configService

.getCommissionInvestorAnnualRate());
            // 封装借款人让利
            projectWithTendersDTO.setCommissionBorrowerAnnualRate(configService

.getCommissionBorrowerAnnualRate());
            // 调用还款服务，启动还款(生成还款计划、应收明细)
            // 由于涉及到分布式事务，后面单独讲解
```

```
            return "审核成功";
        } else {
            // 失败抛出一个业务异常
            log.warn("审核满标放款失败 ！ 标的ID为: {}, 存管代理服务返回的状态为: {}",
                    project.getId(), restResponse.getResult());
            throw new BusinessException(TransactionErrorCode.E_150113);
        }
    } else {
        // 失败抛出一个业务异常
        log.warn("审核满标放款失败 ！ 标的ID为: {}, 存管代理服务返回的状态为: {}",
                project.getId(), restResponse.getResult());
        throw new BusinessException(TransactionErrorCode.E_150113);
    }
}
/**
 * 根据标的及投标信息生成放款明细
 */
public LoanRequest generateLoanRequest(Project project, List<Tender> tenderList,
String commission) {
    LoanRequest loanRequest = new LoanRequest();
    // 设置标的id
    loanRequest.setId(project.getId());
    // 设置平台佣金
    if (StringUtils.isNotBlank(commission)) {
        loanRequest.setCommission(new BigDecimal(commission));
    }
    // 设置标的编码
    loanRequest.setProjectNo(project.getProjectNo());
    // 设置请求流水号( 标的不没有需要生成新的 )

 loanRequest.setRequestNo(CodeNoUtil.getNo(CodePrefixCode.CODE_REQUEST_PREFIX));

    // 处理放款明细
    List<LoanDetailRequest> details = new ArrayList<>();
    tenderList.forEach(tender -> {
        final LoanDetailRequest loanDetailRequest = new LoanDetailRequest();
        // 设置放款金额
        loanDetailRequest.setAmount(tender.getAmount());
        // 设置预处理业务流水号
        loanDetailRequest.setPreRequestNo(tender.getRequestNo());
        details.add(loanDetailRequest);
    });
    // 设置放款明细
    loanRequest.setDetails(details);
    // 返回封装好的数据
    return loanRequest;
}
/**
 * 更新投标信息: 已放款
 * @param tenderList
 */
private void updateTenderStatusAlreadyLoan(List<Tender> tenderList) {
    tenderList.forEach(tender -> {
        // 设置状态为已放款
        tender.setTenderStatus(TradingCode.LOAN.getCode());
        // 更新数据库
        tenderMapper.updateById(tender);
    });
```

```
}

private List<TenderDTO> convertTenderEntityListToDTOList(List<Tender> records) {
    if (records == null) {
        return null;
    }
    List<TenderDTO> dtoList = new ArrayList<>();
    records.forEach(tender -> {
        TenderDTO tenderDTO = new TenderDTO();
        BeanUtils.copyProperties(tender, tenderDTO);
        dtoList.add(tenderDTO);
    });
    return dtoList;
}
```

4. 完善TransactionController类中的代码

```
@Override
@ApiOperation("审核标的满标放款")
@ApiImplicitParams({
    @ApiImplicitParam(name = "id", value = "标的id", required = true,
                      dataType = "long", paramType = "path"),
    @ApiImplicitParam(name = "approveStatus", value = "标的状态", required =
true,
                      dataType = "string", paramType = "path"),
    @ApiImplicitParam(name = "commission", value = "平台佣金", required = true,
                      dataType = "string", paramType = "query")
})
@PutMapping("/m/loans/{id}/projectStatus/{approveStatus}")
public RestResponse<String> loansApprovalStatus(
                                @PathVariable("id") Long id,
                                @PathVariable("approveStatus") String
approveStatus,
                                String commission) {
    String result = projectService.loansApprovalStatus(id, approveStatus,
commission);
    return RestResponse.success(result);
}
```

3.2.3存管代理服务确认放款接口(接口②)

3.2.3.1 功能实现

1. 在DepositoryRecordService接口中新增confirmLoan方法：

```
/**
 * 审核满标放款
 * @param loanRequest
 * @return
 */
DepositoryResponseDTO<DepositoryBaseResponse> confirmLoan(LoanRequest
loanRequest);
```

2. 在DepositoryRecordServiceImpl类中实现该方法：

```
@Override
public DepositoryResponseDTO<DepositoryBaseResponse> confirmLoan(LoanRequest
loanRequest) {
    DepositoryRecord depositoryRecord = new
DepositoryRecord(loanRequest.getRequestNo(),
                                    DepositoryRequestTypeCode.FULL_LOAN.getCode(),
                                    "LoanRequest", loanRequest.getId());

    // 幂等性实现
    DepositoryResponseDTO<DepositoryBaseResponse> responseDTO =
                                                handleIdempotent(depositoryRecord);
    if (responseDTO != null) {
        return responseDTO;
    }

    // 根据requestNo获取交易记录
    depositoryRecord = getEntityByRequestNo(loanRequest.getRequestNo());
    // loanRequest 转为 json 用于数据签名
    final String jsonString = JSON.toJSONString(loanRequest);
    // 业务数据报文，json数据base64编码处理方便传输
    String reqData = EncryptUtil.encodeUTF8StringBase64(jsonString);
    // 拼接银行存管系统请求地址
    String url = configService.getDepositoryUrl() + "/service";
    // 封装通用方法，请求银行存管系统
    return sendHttpGet("CONFIRM_LOAN", url, reqData, depositoryRecord);
}
```

3. 完善DepositoryAgentController类中的代码：

```
@Override
@ApiOperation(value = "审核标的满标放款")
@ApiImplicitParam(name = "loanRequest", value = "标的满标放款信息", required =
true, dataType = "LoanRequest", paramType = "body")
@PostMapping("l/confirm-loan")
public RestResponse<String> confirmLoan(@RequestBody LoanRequest loanRequest) {
    DepositoryResponseDTO<DepositoryBaseResponse> depositoryResponse =

 depositoryRecordService.confirmLoan(loanRequest);
    return getRestResponse(depositoryResponse);
}
```

3.2.3.2 功能测试

3.2.4 存管代理服务修改标的状态接口(接口③)

3.2.4.1 功能实现

1. 在DepositoryRecordService接口中新增modifyProjectStatus方法：

```
/**
 * 修改标的状态
 * @param modifyProjectStatusDTO
 * @return
 */
DepositoryResponseDTO<DepositoryBaseResponse>
modifyProjectStatus(ModifyProjectStatusDTO modifyProjectStatusDTO);
```

2. 在DepositoryRecordServiceImpl类中实现该方法：

```
@Override
public DepositoryResponseDTO<DepositoryBaseResponse> modifyProjectStatus(
        ModifyProjectStatusDTO modifyProjectStatusDTO) {
    DepositoryRecord depositoryRecord = new
              DepositoryRecord(modifyProjectStatusDTO.getRequestNo(),
                   DepositoryRequestTypeCode.MODIFY_STATUS.getCode(),
                                                          "Project",
                                     modifyProjectStatusDTO.getId());

    // 幂等性实现
    DepositoryResponseDTO<DepositoryBaseResponse> responseDTO =
                                    handleIdempotent(depositoryRecord);
    if (responseDTO != null) {
        return responseDTO;
    }

    // 根据requestNo获取交易记录
    depositoryRecord =
getEntityByRequestNo(modifyProjectStatusDTO.getRequestNo());

    // loanRequest 转为 json 进行数据签名
    final String jsonString = JSON.toJSONString(modifyProjectStatusDTO);
    // 业务数据报文
    String reqData = EncryptUtil.encodeUTF8StringBase64(jsonString);
    // 拼接银行存管系统请求地址
    String url = configService.getDepositoryUrl() + "/service";
    // 封装通用方法，请求银行存管系统
    return sendHttpGet("MODIFY_PROJECT", url, reqData, depositoryRecord);
}
```

3. 完善DepositoryAgentController类中的代码：

```
@Override
@ApiOperation(value = "修改标的状态")
@ApiImplicitParam(name = "modifyProjectStatusDTO", value = "修改标的状态DTO",
          required = true, dataType = "ModifyProjectStatusDTO", paramType =
"body")
@PostMapping("l/modify-project-status")
public RestResponse<String> modifyProjectStatus(@RequestBody
ModifyProjectStatusDTO modifyProjectStatusDTO) {
    DepositoryResponseDTO<DepositoryBaseResponse> depositoryResponse =

 depositoryRecordService.modifyProjectStatus(modifyProjectStatusDTO);
    return getRestResponse(depositoryResponse);
}
```

3.2.4.2 功能测试

3.2.5 还款服务启动还款接口(接口④)

3.2.5.1 还款方式

目前万信金融平台对借款人默认采用了**等额本息**的还款方式。等额本息是指，将借款本金和利息总额之和等月拆分，借款人每月偿还相同数额的本息部分。等额本息法最重要的一个特点是每月的还款额相同，从本质上来说是本金所占比例逐月递增，利息所占比例逐月递减，而每月还款总额是不变的。其计算公式为：

```
月利率 = 借款年利率 / 12
每月还款额=借款本金×[月利率×(1+月利率) ^ 借款月数] / {[(1+月利率) ^ 借款月数]-1}
第n月利息 = 本金*月利率* [(1+月利率)^借款月数 - (1+月利率)^(n - 1)] / {[(1+月利率) ^ 借款月数]-1}
```

3.2.5.2 功能实现

数据访问层：

1. 在mapper包中定义一个PlanMapper接口：

```
/**
* <p>
* 借款人还款计划 Mapper 接口
* </p>
*/
public interface PlanMapper extends BaseMapper<RepaymentPlan> {
}
```

2. 在mapper包中新建一个映射配置文件PlanMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.itcast.wanxinp2p.repayment.mapper.PlanMapper">
</mapper>
```

3. 在mapper包中定义一个ReceivablePlanMapper接口：

```
/**
* <p>
* 投资人应收明细Mapper接口
* </p>
*/
public interface ReceivablePlanMapper extends BaseMapper<ReceivablePlan> {
}
```

4. 在mapper包中新建一个映射配置文件ReceivablePlanMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.itcast.wanxinp2p.repayment.mapper.ReceivablePlanMapper">
</mapper>
```

业务层：

1. 新建RepaymentService接口，并在该接口中定义startRepayment方法：

```
/**
 * 启动还款
 * @param projectWithTendersDTO
 * @return
 */
String startRepayment(ProjectWithTendersDTO projectWithTendersDTO);
```

2. 新建RepaymentServiceImpl类，并在该类中实现startRepayment方法：

```
@Service
@Slf4j
public class RepaymentServiceImpl implements RepaymentService {

    @Autowired
    private PlanMapper planMapper;

    @Autowired
    private ReceivablePlanMapper receivablePlanMapper;

    @Override
    public String startRepayment(ProjectWithTendersDTO projectWithTendersDTO) {

        //1.生成借款人还款计划
        //1.1 获取标的信息
        ProjectDTO projectDTO=projectWithTendersDTO.getProject();

        //1.2 获取投标信息
        List<TenderDTO> tenders = projectWithTendersDTO.getTenders();

        //1.3 计算还款的月数
       Double ceil = Math.ceil(projectDTO.getPeriod()/30.0);
       Integer month = ceil.intValue();

        //1.4 还款方式，只针对等额本息
        String repaymentWay = projectDTO.getRepaymentWay();

        if(repaymentWay.equals(RepaymentWayCode.FIXED_REPAYMENT.getCode())){
            //1.5 生成还款计划
            EqualInterestRepayment fixedRepayment =
RepaymentUtil.fixedRepayment(
                projectDTO.getAmount(),
                projectDTO.getBorrowerAnnualRate(),
                month,
                projectDTO.getCommissionAnnualRate());

            //1.6 保存还款计划
```

```
            List<RepaymentPlan> planList =
saveRepaymentPlan(projectDTO,fixedRepayment);

            //2.生成投资人应收明细
            //2.1 根据投标信息生成应收明细
            tenders.forEach(tender -> {
                // 当前投标人的收款明细
                final EqualInterestRepayment receipt =
                        RepaymentUtil.fixedRepayment(tender.getAmount(),
                                tender.getProjectAnnualRate(),
                                                month,
                                projectWithTendersDTO
                                        .getCommissionInvestorAnnualRate());

                /* 由于投标人的收款明细需要还款信息,所有遍历还款计划,
                把还款期数与投资人应收期数对应上*/
                planList.forEach(plan -> {
                    // 2.2 保存应收明细到数据库
                    saveRreceivablePlan(plan, tender, receipt);
                });
            });

        }else{
            return "-1";
        }

        return DepositoryReturnCode.RETURN_CODE_00000.getCode();
    }

    //保存还款计划到数据库
    public List<RepaymentPlan> saveRepaymentPlan(ProjectDTO projectDTO,
                                                 EqualInterestRepayment
                                                         fixedRepayment) {
        List<RepaymentPlan> repaymentPlanList = new ArrayList<>();
        // 获取每期利息
        final Map<Integer, BigDecimal> interestMap =
                fixedRepayment.getInterestMap();
        // 平台收取利息
        final Map<Integer, BigDecimal> commissionMap =
                fixedRepayment.getCommissionMap();
        // 获取每期本金
        fixedRepayment.getPrincipalMap().forEach((k, v) -> {
            // 还款计划封装数据
            final RepaymentPlan repaymentPlan = new RepaymentPlan();
            // 标的id
            repaymentPlan.setProjectId(projectDTO.getId());
            // 发标人用户标识
            repaymentPlan.setConsumerId(projectDTO.getConsumerId());
            // 发标人用户编码
            repaymentPlan.setUserNo(projectDTO.getUserNo());
            // 标的编码
            repaymentPlan.setProjectNo(projectDTO.getProjectNo());
            // 期数
            repaymentPlan.setNumberOfPeriods(k);
            // 当期还款利息
            repaymentPlan.setInterest(interestMap.get(k));
            // 还款本金
            repaymentPlan.setPrincipal(v);
```

```
            // 本息 = 本金 + 利息
            repaymentPlan.setAmount(repaymentPlan.getPrincipal()
                    .add(repaymentPlan.getInterest()));
            // 应还时间 = 当前时间 + 期数( 单位月 )
            repaymentPlan.setShouldRepaymentDate(DateUtil
                            .localDateTimeAddMonth(DateUtil.now(), k));
            // 应还状态, 当前业务为待还
            repaymentPlan.setRepaymentStatus("0");
            // 计划创建时间
            repaymentPlan.setCreateDate(DateUtil.now());
            // 设置平台佣金( 借款人让利 ) 注意这个地方是 具体佣金
            repaymentPlan.setCommission(commissionMap.get(k));
            // 保存到数据库
            planMapper.insert(repaymentPlan);
            repaymentPlanList.add(repaymentPlan);
        });
        return repaymentPlanList;
    }

    //保存应收明细到数据库
    private void saveRreceivablePlan(RepaymentPlan repaymentPlan,
                                     TenderDTO tender,
                                     EqualInterestRepayment receipt) {
        // 应收本金
        final Map<Integer, BigDecimal> principalMap = receipt.getPrincipalMap();
        // 应收利息
        final Map<Integer, BigDecimal> interestMap = receipt.getInterestMap();
        // 平台收取利息
        final Map<Integer, BigDecimal> commissionMap =
                receipt.getCommissionMap();
        // 封装投资人应收明细
        ReceivablePlan receivablePlan = new ReceivablePlan();
        // 投标信息标识
        receivablePlan.setTenderId(tender.getId());
        // 设置期数
        receivablePlan.setNumberOfPeriods(repaymentPlan.getNumberOfPeriods());
        // 投标人用户标识
        receivablePlan.setConsumerId(tender.getConsumerId());
        // 投标人用户编码
        receivablePlan.setUserNo(tender.getUserNo());
        // 还款计划项标识
        receivablePlan.setRepaymentId(repaymentPlan.getId());
        // 应收利息
        receivablePlan.setInterest(interestMap.get(repaymentPlan
                .getNumberOfPeriods()));
        // 应收本金
        receivablePlan.setPrincipal(principalMap.get(repaymentPlan
                .getNumberOfPeriods()));
        // 应收本息 = 应收本金 + 应收利息
        receivablePlan.setAmount(receivablePlan.getInterest()
                                .add(receivablePlan.getPrincipal()));
        // 应收时间
        receivablePlan.setShouldReceivableDate(repaymentPlan
                .getShouldRepaymentDate());
        // 应收状态, 当前业务为未收
        receivablePlan.setReceivableStatus(0);
        // 创建时间
        receivablePlan.setCreateDate(DateUtil.now());
```

```
            // 设置投资人让利，注意这个地方是具体：佣金
            receivablePlan.setCommission(commissionMap
                                    .get(repaymentPlan.getNumberOfPeriods()));
            // 保存到数据库
            receivablePlanMapper.insert(receivablePlan);
        }
    }
```

3. 完善RepaymentController类中的代码：

```
@Override
@ApiOperation("启动还款")
@ApiImplicitParam(name = "projectWithTendersDTO",
                  value = "通过id获取标的信息",
                  required = true,
                  dataType = "ProjectWithTendersDTO",
                  paramType = "body")
@PostMapping("/l/start-repayment")
public RestResponse<String> startRepayment(@RequestBody ProjectWithTendersDTO
projectWithTendersDTO) {
    String result = repaymentService.startRepayment(projectWithTendersDTO);
    return RestResponse.success(result);
}
```

3.2.6 前后端集成测试

3.3 分布式事务

满标放款功能涉及到分布式事务问题，需要使用**RocketMQ事务消息**实现最终一致性事务，请参考”分布式事务-讲义.pdf“。