算法解释文档

变更记录

修改人员	日期	变更原因	版本号
李蒋泽辰	2022.4.2	创建文档，添加基础推荐部分	v1.0
楼澜	2022.5.10	添加目标3 工人相关性	v1.1
徐琪	2022.5.11	添加目标1 工人能力 和目标2 工人活跃度	v1.2
李蒋泽辰	2022.5.11	添加了多目标优化算法及缓存和定期刷新	v1.3
蒲中正	2022.5.12	添加目标4 工人多样性	v1.4
蒲中正	2022.5.22	添加任务缺陷数量预测算法	v1.5

一、基础推荐与相似度计算

1 推荐算法

考虑到系统刚部署时用户数量少，新注册的用户也缺乏历史数据。此外，为了提高推荐算法的灵活性以及解决上述的冷启动问题，我们综合多种推荐算法，并让管理员来定义这些算法在应用过程中的权值。

我们定义了三种影响任务优先级的因素：任务的时间紧迫程度、任务与被推荐用户历史选择任务的相似度、已经选择该任务的用户和被推荐用户的相似程度，我们使用这三个因素为每一个待推荐的任务计算出一个因素值，并将这三个因素依据管理员的设定值加权平均得到一个待推荐任务最终的任务优先数，将待推荐任务按照优先数降序排列即得到推荐列表。

1.1 推荐因素的计算

1.1.1 基于任务时间紧迫程度的优先级

假设有一组待推荐任务T，其中第i个任务为t_i，它的剩余时间（结束时间减去当前时间的值）为r_i，它还需要招募的工人数量为x_i，它的时间紧迫度因素值f_i为$f_i = \frac{x_i}{r_i \times f{max}}$，其中$f{max}$代表所有$f_i$的最大值。

```
public class TaskUrgencyConcreteFactor extends RuleFactor {

    /**
     * 第i个任务没有归一化的紧迫程度(记为Ui)：(测试还需要招募的人数 / 测试剩余时间)
     * 所有任务中最大的未归一化紧迫程度Umax
     * @param targetUserId 目标用户
     * @param candidateTasks 候选任务列表
     * @return 优先数列表，优先数值为candidateTasks中任务的紧迫程度，既Ui/Umax
     */
    @Override
```

```
    protected List<Double> calculateRawFactorValues(Integer targetUserId,
List<TaskViewVO> candidateTasks) {
        List<Double> taskUrgencies=new ArrayList<>(); //任务紧急程度
        double taskMaxUrgency=0.0;
        for(TaskViewVO taskVO:candidateTasks){
            /*
            ...省略了计算任务相似度的代码
            */
        }
        if(taskMaxUrgency==0.0) //处理紧急程度为0的特殊情况
            taskMaxUrgency=1.0;
        for(int i=0;i<taskUrgencies.size();i++){
            taskUrgencies.set(i,taskUrgencies.get(i)/taskMaxUrgency);//归一化
        }
        return taskUrgencies;
    }
}
```

1.1.2 基于任务与被推荐用户历史选择任务的相似度的优先级

我们使用任务标签所构成的向量来表示一个任务，使用该向量的余弦相似度来计算被推荐任务与用户历史选择任务的相似度矩阵M，其中$M*{ij}*$代表第*i*个待推荐任务和第*j*个用户历史任务的相似度，则第*i*个待推荐任务的因素值为*$\frac1n \times \Sigma*{j=0}^{n}{M_{ij}}$，其中$n$为用户历史选择任务的个数

```
public class TaskSimilarityConcreteFactor extends RuleFactor {
    @Resource
    TaskUserMapper taskUserMapper;
    @Resource
    TaskTagMapper taskTagMapper;

    /**
     *
     * @param targetUserId 目标用户
     * @param candidateTasks 候选任务列表
     * @return 优先数列表，优先数值为candidateTasks中任务与targetUser用户历史任务的平均相
似度
     */
    @Override
    protected List<Double> calculateRawFactorValues(Integer targetUserId,
List<TaskViewVO> candidateTasks) {
        //求得候选任务矩阵candidateTasksMatrix
        int sz=candidateTasks.size();
        DenseMatrix64F candidateTasksMatrix = new DenseMatrix64F(sz,
Constant.TASK_TAG_COUNT);
        for(int i=0;i<sz;i++){
           /**具体计算任务矩阵的过程**/
        }
        //求得历史任务矩阵pickedTasksMatrix
        List<Integer>
userPickedTaskIds=taskUserMapper.selectByUserId(targetUserId);
        int userPickedTaskSz=userPickedTaskIds.size();
        int
pickedTasksMatrixRowNum=Math.min(userPickedTaskSz,Constant.PICKED_TASK_CAL_NUM);
        //历史选择任务个数为0，均构建全零列表并返回
        if(pickedTasksMatrixRowNum==0){
```

```
            /**构建全0列表**/
            return specificAllZeroAns;
        }
        int cnt=0;
        DenseMatrix64F pickedTasksMatrix = new
DenseMatrix64F(pickedTasksMatrixRowNum, Constant.TASK_TAG_COUNT);
            /**请求数据库查询数据库获取用户历史选择的任务矩阵**/

        //对候选任务矩阵做归一化
        DenseMatrix64F[] candidateTasksVectors=new DenseMatrix64F[sz];
        /**归一化具体过程**/
        //对历史任务矩阵做归一化
        DenseMatrix64F[] pickedTasksVectors=new
DenseMatrix64F[pickedTasksMatrixRowNum];
        /**归一化具体过程**/

        //候选任务矩阵与历史任务矩阵的转置做乘积，结果每一行是候选任务和所有历史任务的相似度
        //对上述结果矩阵每行求均值，得到单列向量，转置即为结果对应向量
        DenseMatrix64F taskSimilarityMatrix = new
DenseMatrix64F(sz,pickedTasksMatrixRowNum);
        DenseMatrix64F pickedTasksMatrixTrans = new
DenseMatrix64F(Constant.TASK_TAG_COUNT,pickedTasksMatrixRowNum);
        CommonOps.transpose(pickedTasksMatrix,pickedTasksMatrixTrans);

 CommonOps.mult(candidateTasksMatrix,pickedTasksMatrixTrans,taskSimilarityMatrix
);

        //计算矩阵每一行的均值得出一个待推荐任务和用户历史选择任务的相似度情况
        DenseMatrix64F taskSimilarityMeanMatrix = new DenseMatrix64F(sz,1);

        CommonOps.sumRows(taskSimilarityMatrix,taskSimilarityMeanMatrix);
        CommonOps.divide(pickedTasksMatrixRowNum,taskSimilarityMeanMatrix);

        List<Double> taskSimilarity=new ArrayList<>();
        /**将转换好的平均相似度向量转换成数组类型**/
        return taskSimilarity;
    }
}
```

1.1.3 基于已经选择该任务的用户和被推荐用户的相似程度

用户相似度有两个组成部分，第一个部分是用户标签向量的杰卡德相似度，第二部分是用户任务偏好与专业能力向量的余弦相似度（下面简称为偏好相似度），这两部分的平均值作为用户相似度。但在实际应用过程中，为了减少性能开销，我们先使用其他用户与目标用户标签向量的杰卡德距离作为筛选标准，取杰卡德距离最短的前n个用户作为候选相似用户集，在计算目标用户与这些候选相似用户之间的偏好相似度。对于不在候选相似用户集中的用户，我们认为其与目标用户的相似度为0。

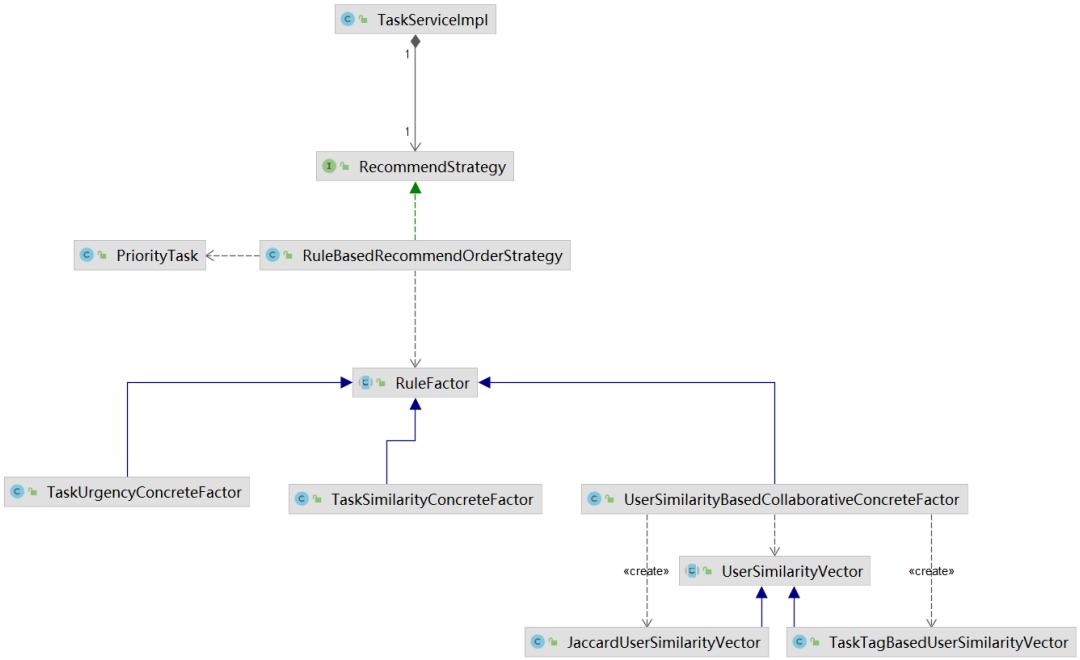
1.1.3.1 用户任务偏好与专业能力向量

该向量的维数等于任务标签数量+1，除了最后一维外，其余维度和任务标签一一对应，可以认为最后一维对应了任务的一个默认标签，每个任务都有此标签。初始时将这个向量置为0，随后用户每在一个任务上提交报告，那么对于这个任务的所有标签（包括默认标签）我们都在用户向量与这些标签对应的维度上增加这个报告中缺陷的评分总和。虽然评分是动态的，但是由于每次推荐都重新计算用户的向量，所以这个向量总能保持最新。

```
public class UserSimilarityBasedCollaborativeConcreteFactor extends RuleFactor {
    private final UserSimilarityMapper userSimilarityMapper;
```

```
    private final TaskUserMapper taskUserMapper;
    public UserSimilarityBasedCollaborativeConcreteFactor(UserSimilarityMapper
userSimilarityMapper, TaskUserMapper taskUserMapper){
        this.userSimilarityMapper = userSimilarityMapper;
        this.taskUserMapper = taskUserMapper;
    }


    /**
     *
     * @param targetUserId 目标用户Id
     * @param candidateTasks 候选任务列表
     * @return 优先数列表，优先数值为选择candidateTasks中任务的用户与targetUser用户的平均
相似度
     */
    @Override
    protected List<Double> calculateRawFactorValues(Integer targetUserId,
List<TaskViewVO> candidateTasks) {
        UserSimilarityVector jaccardSimilarityVector = /**计算用户标签的杰卡德相似度
向量**/
        UserSimilarityVector taskTagBasedSimilarityVector =/**计算用户的任务偏好与专
业能力相似度向量**/

        //将两个相似度向量求平均值形成聚合相似度向量
        UserSimilarityVector weightedSimilarityVector =
jaccardSimilarityVector.times(Constant.JACCARD_USER_SIMILARITY_WEIGHT)

.plus(taskTagBasedSimilarityVector.times(Constant.TASK_TAG_USER_SIMILARITY_WEIGH
T));

        //将获取选择每个待推荐任务的用户列表
        Map<Integer, List<TaskUser>> taskUserGroupByTaskId = taskUserMapper

.selectByTaskIds(candidateTasks.stream().map(TaskViewVO::getId).collect(Collecto
rs.toList()))
                .stream().collect(Collectors.groupingBy(TaskUser::getTaskid));

        return candidateTasks.stream().map(task ->{
                    /**计算选择每个待推荐任务的用户和目标用户的平均相似度**/
        ).collect(Collectors.toList());
    }


}
```

1.2 算法接口设计



1.2.1推荐策略接口

推荐算法使用策略模式：

```
public interface RecommendStrategy {
    List<TaskViewVO> recommend(Integer targetUserId, List<TaskViewVO>
candidateTasks);
}
```

可扩展性在于：可以通过实现这个推荐策略接口来添加不同的推荐策略类，事实上，上下文类可以持有一个推荐规则的列表，然后让一组任务依次经过多个推荐策略类的推荐（比如先筛选后排序）。可能的推荐策略类型有：

1. 对于候选任务进行排序的推荐策略
2. 对于候选任务进行筛选的推荐策略

目前只有一个实现类，就是上文提到的基于管理员设置的推荐规则，对推荐策略进行

1.2.2 规则因素抽象类

规则因素也使用策略模式：

```
public abstract class RuleFactor {
    protected  double weight;
    public RuleFactor setWeight(double weight) {/**...**/}
    public List<Double> calculateWaitedFactorValues(Integer targetUserId,
List<TaskViewVO> candidateTasks){/**进行一些统一的处理**/}
    protected abstract List<Double> calculateRawFactorValues(Integer
targetUserId, List<TaskViewVO> candidateTask);//模板方法
}
```

可扩展性在于：可以通过继承`RuleFactor`类来添加新的规则因素，为管理员提供更多选择

考虑：为什么`public List<Double> calculateWaitedFactorValues(Integer targetUserId, List<TaskViewVO> candidateTasks)`的方法参数中需要传入整个的待推荐任务列表？这样做的原因是：

1.粗粒度的接口有利于高效的实现

2.任务紧迫程度不能依靠单个候选任务计算得出

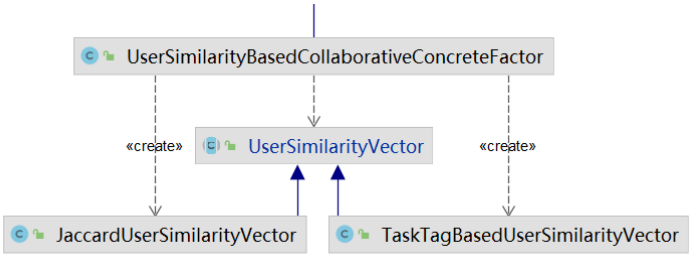
1.2.3 用户相似度向量

根据单一职责原则将计算相似度的职责抽象成一个类

```
public abstract class UserSimilarityVector {
    protected final int targetUserId;
    /**
     *
     * @param targetUserId 目标用户id，这个向量代表目标用户与其他所有用户的相似度
     */
    public UserSimilarityVector(int targetUserId){
        this.targetUserId = targetUserId;
    }
    /**
     *
     * @param otherUserId 与目标用户进行相似度比较的用户
     * @return targetUserId 和 otherUserId 所代表用户的相似度
     */
    public abstract double getSimilarityWithTargetUser(int otherUserId);

    public UserSimilarityVector plus(UserSimilarityVector otherVector){
        /**提供公共的向量加法实现（可能并不高效）**/
    }
    public UserSimilarityVector times(double factor){
        /**提供公共的向量数乘实现（可能并不高效）**/
    }
}
```

可扩展性在于：可以通过继承`UserSimilarityVector`类来添加新的用户相似度计算方法

考虑：`UserSimilarityBasedCollaborativeConcreteFactor`类直接依赖于`UserSimilarityVector`的具体实现，这样的设计是否合理?



纵观整个系统，`UserSimilarityVector`的子类只在`UserSimilarityBasedCollaborativeConcreteFactor`中进行初始化，而`UserSimilarityBasedCollaborativeConcreteFactor`本身已经是一个具体实现类，所以这种耦合已经被封装在其中了，所以就目前的系统来说，这样的设计是合理的，如果在这种情况下依然选择使用静态工厂方法等模式进行优化反而是一种过度设计。

如果要对管理员开放自定义用户相似度计算规则的界面，那么可以考虑工厂模式。

2 相似度算法

在报告相似度的计算中，我们使用了SIF算法来计算报告的文本相似度，并使用为本相似度代表报告相似度。我们选择该算法的原因是因为它不需要依赖于特定的语料库所训练出的词向量，因为我们没有大量于软件测试报告相关的句子来构建这个语料库。

在实践过程中我们遇到了一些问题并解决了它们，最终的相似度算法效果良好。

2.1 SIF算法的探索与优化

2.1.1 概述

SIF算法在计算句子向量时是以组维单位的，它首先计算一个句子中所有词向量的加权平均值（类似于TF-IDF加权，但有所差异），然后求出这些句子向量的第一主成分，并将每个句子向量减去其在该主成分上的投影后获得该句子最终的向量，最后我们使用这个向量来计算句子之间的余弦相似度。

2.1.2 困难

在实践过程中，我们在数据库中存储了缺陷之间的相似度值，并在每一次插入新缺陷的时候计算并存储该缺陷和其余所有缺陷之间的相似度，如果直接使用SIF算法可能会产生一些问题：由于每一次传入的一组句子都不同，算法对每一组句子计算出的主成分也不同，这就使得SIF算法对每一组句子计算出的相似度仅仅在这一组句子之间适用，例如：**我们用句子a, b, c计算出A和b，c间的相似度（$Sim*{ab},$和$Sim*{ac}$），再使用句子x, a, b, c计算出x和a, b, c之间的相似度（$Sim*{xa},$和$Sim*{xb}$和$Sim*{xc}$），那么当我们用$Sim*{ac}$和Sim_{xc}来作比较，期望以此来判断c和x，a之中的哪一个句子更为相近时，由于这两个相似度是在不同的上下文中计算得出的，所以这种比较完全可能带来错误的结果。**而这种错误在句子很少的时候可能尤其容易发生，因为在句子很少的情况下，每增加一个新的句子，句子组的主成分可能会发生很大的变化。此外，当句子组中只有两个句子时，由于主成分被去除，这两个句子向量的余弦值会接近于-1，而这并不是我们希望看到的。

我们尝试过定制SIF算法，使其在计算句子向量的时候不去除句子在主成分上的投影，直接对词向量进行加权平均，但这样所得出的相似度不尽人意，所有的句子之间的相似度都很高。我们推测这可能是因为我们选取的词向量模型过于通用。

2.1.3 解决方案

仔细思考过后，我们决定先在网络上寻找一组与测试报告描述相近的句子，使用这一组句子计算出一个主成分pc，然后在之后的使用过程中不再根据传入的句子计算主成分，而是直接将这些句子向量在主成分pc上的投影去除。以此来保证每一次的相似度计算都是在相同的上下文中进行的。同时，由于我们寻找的句子时与测试报告描述相近的，所以相似度的准确性也可以被保证。

2.2 最终效果与分析

根据我们的实践，我们使用了17个句子来计算pc，并使用它去判断不位于这17个句子中的两个句子的相似度：

下面的句子用来计算主成分pc

```
{"strings":[
"3个数据集合其中dev或者train数据量和batch_size能整除的时候就回出现问题，DatasetIterater函数这块一个bug，会导致爆出问题",
    "DatasetIterater函数这块有个问题，batch_size能整除dev或者train数据量的话就会报错",
    ""Elevator 电梯楼层"组件，有这样一种情况，在城市列表中，有热门城市一项；一个城市的id固定，但是它又是热门城市，造成点击后会出现两个高亮色。",
    "使用双色BarCharts，但是柱状图的位置有3成左右的几率是乱掉了，这个时候如果拖动鼠标调整一下浏览器窗口大小，柱状图就会排列正常了",
    "在“第一个数”中输入10 在第二个数中输入0 点击除法按钮 在错误提示框中点击确定按钮 预期结果为：“错误提示框”关闭，程序继续运行 实际结果：程序关闭",
```

```
    "新增和编辑行时无法成功， 报错了。给操作的小图标一个title提示，以便鼠标浮动上去时能够显示该图标是干嘛的",
    "假设文件夹里含有文件，比如pdf文件，同时这个pdf文件未被导入到索引中。",
    "当这个文件夹被剪切走时，会出现这样的现象：这个文件夹并不能被剪切走，文件夹依然存在，只是未被加入到索引中，而被粘贴出现的新文件夹一切正常。",
    "如果那个pdf文件已经被导入了索引中，剪切粘贴功能似乎就一切正常。解决办法：在剪切前，把文件夹底下的内容自动导入到索引中？我进一步测试了，如果文件夹里含有未被导入索引的文件，删除文件夹功能也失效。",
    "但是柱状图的位置有一部分是乱掉的，这个时候只有用拖动鼠标的方式调整一下浏览器窗口大小，柱状图才会排列正常了",
    "例如系统已存在可正常登录的用户名称为Chang，如果使用chang作为用户名登录，系统提示登录成功，但是仍然会返回到登录界面，实际上是没有登录成功的。",
    "主界面中连续快速点击“更改设置”按钮，程序会卡死首先，进入应用主界面；其次，连续点击右上角的“更改设置”按钮，注意一定要点的飞快。十几次后程序会无法响应，强制退出。",
    "主界面中快速点“更改设置”按钮就会卡死首先，进入应用主界面然后连续点击右上角的“更改设置”按钮，要点的飞快。十几次后会无法响应",
    "内置跳转小程序打不开在“发现页面”点击左下角“开心一下”，页面发生跳转，不过一直是白屏，无法打开",
    "去掉保存和取消的文字， 取消图标用减号-显示；对于新增未保存（区别于默认传进来的数据和已编辑过的数据）的数据点击取消时应该将该列删除掉",
    "[集成管理]模块的[项目注册管理]页面， 查看项目立项信息时，出现“未能加载类型“ProjectGeneralView”错误",
    "在[项目管理]-> [项目基本信息管理]页面下；2）选择[已审核]TAB页； 3）在GRID列表中，选择任意一条项目立项信息，点击[查看]按钮；4）系统页面弹出：按钮报错信息“"
]}
```

下面的三个句子用来测试算法的效果： （实验代码在附录结尾处）

```
{"strings":[
"无法发布测试;进入首页，以发包方身份登陆，点击右上角菜单中的发布测试，填写测试信息，点击发布，预期系统提示发布成功，但实际上系统没有任何提示，测试页并没有发布", //句子1
"发布测试出错；发包方登录后从导航栏菜单进入发布测试页面后，输入测试描述，所需人数，起止时间，并上传测试文档以后点击发布，系统无响应", //句子2
"无法提交报告，以众包工人身份登录，在主页中选择正在进行一栏，选择需要提交报告的测试，填写报告信息，点击提交报告，预期系统提示提交成功，但实际上系统没有任何提示，报告并未提交" //句子3
]}
```

其中，句子1描述了无法发布测试的缺陷，句子2和1描述相同的缺陷，而句子3虽然使用了和句子1相近的形式，但描述了不同的缺陷

计算相似度的结果为：

相似度	句子2	句子3
句子1	0.9140273837372119	0.8601595233279353
句子2	-	0.8349675055440855

这个结果差强人意，虽然相似度接近，但是通过大小依然可以判断句子1和句子2描述相同缺陷。

2.3 实际应用

在实际应用的过程中，我们可以在系统运行的过程中定期的更新这个主成分，并在每一次更新的时候都重新计算所有缺陷之间的相似度矩阵，以此进一步提升相似度计算的准确性。

2.4 算法相似度类设计

略

2.5 算法相似度接口设计

基于http请求的接口，具有天然的设备无关性，可以很好地将算法和上下文进行解耦

```
@app.route('/docsim',methods=['POST','GET'])
def getSimilarities():
    #### 调用文本相似度类来计算文本相似度
        return {"res":sims}
```

二、多目标评分与推荐

1. 整体概述

在迭代三中，我们对工人进行了基于任务与基于工人自身的多目标评分，包括工人能力、工人活跃度、工人相关性、工人多样性五个维度。在此基础上，我们用遗传算法NSGA-II进行搜索优化，来进行任务推荐；除此之外，我们可视化了五个维度的信息，呈现给发包方，让发包方及时调整自己的任务的推荐维度权重占比以及手动停止招募；最后，我们还给工人可视化了其自身能力、活跃度、报告关键词词云，方便其自我提升。

总体过程是将一组候选工人的某个子集推荐到一个测试任务中，以达到以下目标：

1. 最大化工人能力
2. 最大化工人活跃度
3. 最大化工人和任务的相关性
4. 最大化工人的多样性
5. 最小化招募工人开销

2. 目标1——工人能力

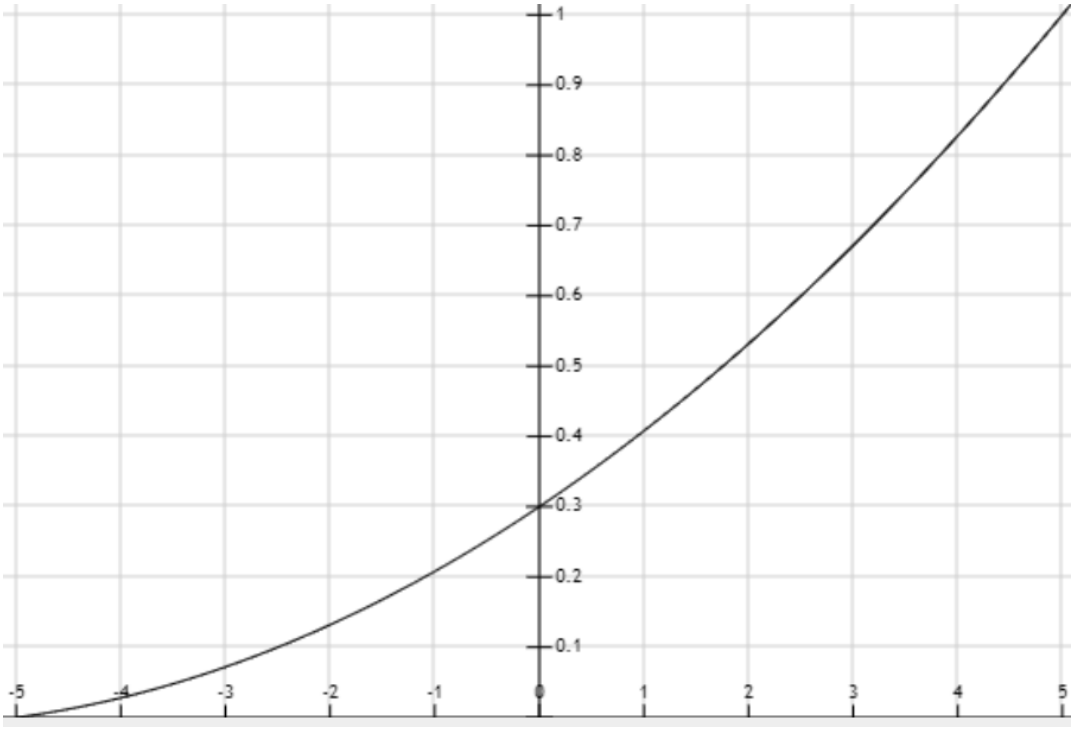
工人能力分为以下五个部分：报告协作能力、报告审查能力、创新能力、寻找bug的能力和语言表达能力。

每种能力由多个评价指标构成，根据经验和认知对各个指标赋予权重，加权求和得到相应的能力值。

所有能力的取值范围均为0-1，对五种能力加权求和，可以得到工人的综合能力，范围同样为0-1。

2.1 报告协作能力

1. 【权重：0.35】fork其他缺陷的概率
    - 某众包工人提交的缺陷中，fork别人的缺陷数量 / 该众包工人提交的所有缺陷数量
    - 算出的值范围为 0 - 1，值越大表示fork缺陷的概率越大，即协作能力越高
2. 【权重：0.65】评分差距（当前评分和fork的节点的评分的差距）
    - 找到某众包工人fork其他缺陷的所有节点
    - 针对某个缺陷节点，当前节点的评分为 curScore，fork的节点的评分为 preScore（如果两者任一个没有评分，则放弃对该节点的评估）
    - 计算 diff = curScore - preScore，diff 范围为 -5 到 5
    - 将上述 diff 代入 $f(x) = 0.008x^2+0.1x+0.3$ 中的x，计算$f(x)$ 为 gapRate



- 对所有的 gapRate 求均值
- 算出的值范围为 0 - 1，值越高表示比fork节点的效果越好，即协作能力越高

2.2 报告审查能力

1. 【权重：0.4】评分差距（给他人评分的分值和该报告平均分值的差距）
   - 找到某众包工人对于其他缺陷的所有评分
   - 对于每条评分数据，计算 diff = abs(该工人对他的评分 - 该缺陷的均分)
   - 对于每条评分数据，计算 gapRate = diff / 5（评分范围为0-5）
   - 对所有 gapRate 求均值
   - 1 - 上述值
   - 算出的值在 0 - 1 之间，值越大表示评分差距越小，即审查能力越高
2. 【权重：0.6】评论的点赞数、点踩数
   - 采用威尔逊算法，计算威尔逊得分
   - $$n = u + v$$

     $$p = v/n$$

     $$S = (p + \frac{z_\alpha^2}{2n} - \frac{z_\alpha^2}{2n}\sqrt{4n(1-p)p + z_\alpha^2})/(1 + \frac{z_\alpha^2}{n})$$
   - 其中，u 表示点赞数，v 表示点踩数，n 表示评价总数，p 表示好评率，z 表示正态分布的分位数（参数）。
   - 由于我们数据量普遍比较小，所以 z 取 0.5，约等于 70% 的置信度。
   - 算出的值在 0 - 1 之间，值越大表示评论越受认可，即审查能力越高

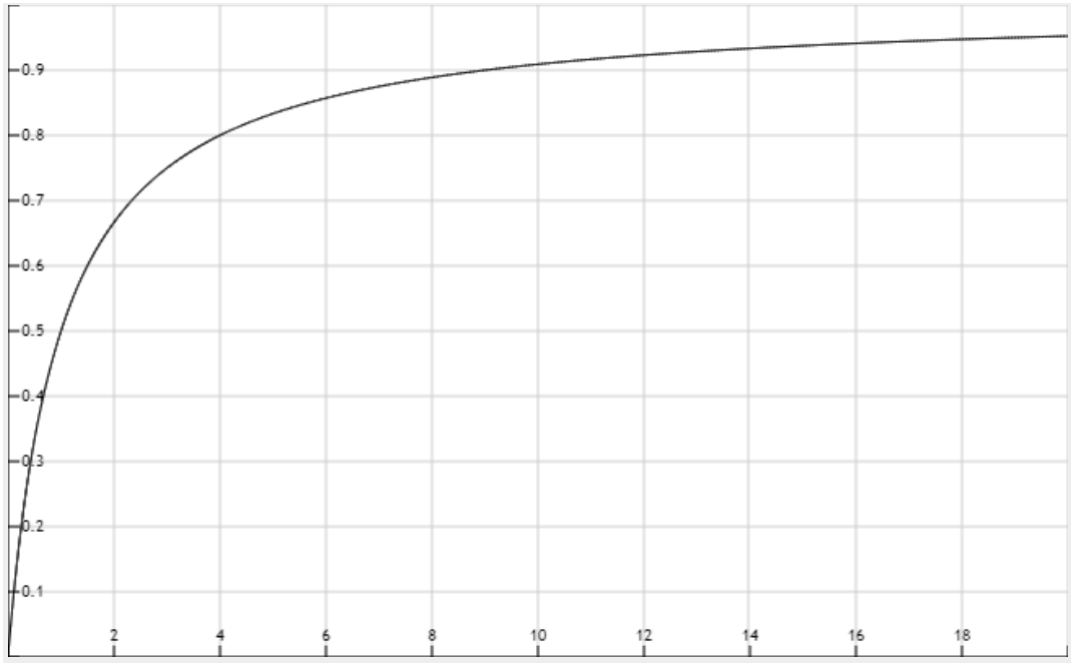
2.3 创新能力

1. 【权重：0.2】flaw节点作为根节点的概率
   - 对于某众包工人提交的所有缺陷，作为根节点的缺陷数量 / 提交的所有缺陷
   - 算出的值在0-1之间，值越大表示概率越大，即创新能力越高
2. 【权重：0.1】缺陷被fork的次数（概率）
   - 计算某众包工人提交的所有缺陷，缺陷直接或间接被fork的次数，求和

- 为了简单归一化，将上述结果除以所有flaw的数量
- 算出的值在0-1之间，值越大表示被fork的次数越多，即创新能力越高

3. 【权重：0.35】提交报告的重复程度（表征重复数和重复程度）
   - 对于某个缺陷，duplicates为该缺陷所属的缺陷树的节点个数（重复个数）
   - 定义该缺陷的 duplicate index = 1 / duplicates
   - 对于某众包工人，计算它提交的所有缺陷的duplicate index之和，并除以他提交的所有缺陷的个数
   - 算出的值在0-1之间，值越大表示重复程度越小，即创新能力越高
4. 【权重：0.25】缺陷相似度
   - 对于某个缺陷，计算它和所属任务下的所有他人提交的缺陷的相似度均值
   - 对于某众包工人，将其所有提交的缺陷计算上述均值，求出最终均值
   - 1 - 上述值
   - 算出的值在0-1之间，值越大表示相似度越低，即创新能力越高

2.4 寻找bug的能力

1. 【权重：0.1】参与的任务数
   - 设参与的任务数为x，代入 1 - 1/x+1 求值
   - 

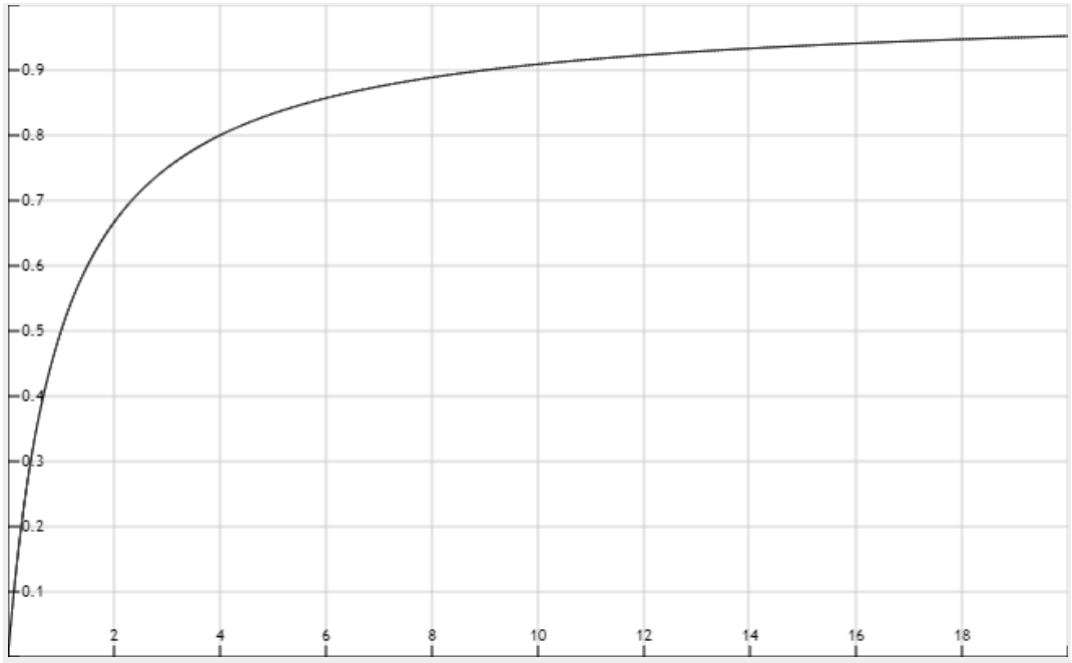
   - 算出的值在0-1之间，值越大表示参与的任务数越多，即找bug能力越强
2. 【权重：0.1】提交的报告数
   - 设提交的报告数为x，代入 1 - 1/x+1 求值
   - 图像同上
   - 算出的值在0-1之间，值越大表示提交的报告数越多，即找bug能力越强
3. 【权重：0.4】提交的缺陷数
   - 设提交的缺陷数为x，代入 1 - 1/x+1 求值
   - 图像同上
   - 算出的值在0-1之间，值越大表示提交的缺陷数越多，即找bug能力越强
4. 【权重：0.4】发现的缺陷占比
   - 对于某个任务，rate = 众包工人发现的flaw数量 / 按照缺陷图得到的该任务下的总flaw数
   - 对于众包工人参与的所有任务（需要是已经提交完善缺陷的任务），求出 rate 均值
   - 算出的值在 0 - 1 之间，值越大表示发现的缺陷占比越高，即寻找bug能力越强

2.5 语言表达能力

1. 图文一致性
    - 敬请期待

3. 目标2——工人活跃度

考虑以下几个指标。指标的取值范围均在0-1之间，加权求和，得到活跃度的范围也在0-1之间。

1. 【暂不考虑】最后一次提交时间
    - 计算当前时间和最后一次提交时间的差（以天数为单位）
2. 【权重：0.4】近三天提交的报告数
    - 按照 1-1/(x+1) 映射
    - 

    - 提交的报告数越多，活跃度越高。取值范围在0-1之间，非线性增长。
3. 【权重：0.3】近两周提交的报告数
    - 同上
4. 【权重：0.2】近一个月提交的报告数
    - 同上
5. 【权重：0.1】近半年提交的报告数
    - 同上

4. 目标3——工人相关性

工人相关度分成了五个小方面分别打分，再加权求和。

先引入一个概念，即一个工人/任务的领域知识。一个工人的领域知识指，将ta做过的所有报告文本信息汇总到一起，并用TFIDF和我们自创的众测停词表求出频率最高的十个关键字，这就是ta的领域知识。一个任务的领域知识类似，不过把文本信息换成了这个任务所有能获取到的领域信息。此外，值得注意的是，领域知识相似度算法采用余弦相似度，TAG相似度算法采用Jaccard算法。

```
    //余弦相似度代码
    public Double cosineSimilarity (List<String> vector1, List<String> vector2 )
{
        if ( vector1.size() == 0 || vector2.size() == 0 ) {
            return 0.0;
        }
```

```
        HashSet<String> totalTermList = new HashSet<String>();

        for ( int i =0; i < vector1.size(); i++ ) {
            totalTermList.add( vector1.get(i));
        }
        for ( int i =0; i < vector2.size(); i++ ) {
            totalTermList.add( vector2.get( i ));
        }

        int v1sum = 0, v2sum = 0, multiply = 0;
        for ( String term: totalTermList ) {
            int v1 = 0, v2 = 0;
            if ( vector1.contains( term ))
                v1 = 1;
            if ( vector2.contains( term ))
                v2 = 1;

            v1sum += v1*v1;
            v2sum += v2*v2;
            multiply += v1*v2;
        }

        double sim = (1.0*multiply) / (Math.sqrt( 1.0*v1sum ) * Math.sqrt(
1.0*v2sum ));
        return sim;
    }
```

4.1 领域知识相似性

任务与工人：任务的领域知识和工人的领域知识计算相似度。

任务与任务：工人承接过的所有任务的领域知识和当前任务的领域知识计算相似度。

工人与工人：工人承接过的所有任务和已经选择当前任务的所有工人这二者的领域知识相似度。

如果工人没有领域知识，或者任务没有人选，会给一个不算高的默认值。因为相关的活跃度信息已经在目标2衡量过了。

4.2 TAG相似性

任务与任务：工人承接过的所有任务的TAG和当前任务的TAG计算相似度。

工人与工人：工人承接过的所有任务和已经选择当前任务的所有工人这二者的TAG相似度。

详见算法解释文档——基于规则的推荐。

5. 目标4——工人多样性

先对工人领域知识多样性和工人测试环境多样性分别计算，再以1:1权重求和并归一化。

5.1 领域知识多样性

领域知识的定义详见目标3。

每个工人有一组领域知识。

依次对候选工人集的领域知识进行比较记录，去除候选工人集中所有领域知识的重复值，将剩余值的数量作为该候选工人集的领域知识多样性的表征，将该值与候选工人集中所有领域知识值的数量的比作为归一化结果。

5.2 测试环境多样性

每个测试环境以四个特征表述：设备类型、操作系统、内存大小、网络环境。

工人可对各特征选定一个值，四个特征值构成一组测试环境，每个工人对应一组测试环境。

依次对候选工人集的测试环境进行比较记录，去除候选工人集中所有测试环境特征重复的值，将剩余值的数量作为该候选工人集的测试环境多样性的表征，将该值与候选工人集中所有测试环境值的数量的比作为归一化结果。

6. 目标5——开销

由于我们的系统中暂时没有引入交易过程，所以我们以推荐集中的工人数量/候选工人数量代表开销。这并不影响后续扩展。

7. 多目标优化算法——NSGAII

我们参考了以下论文的设计和实现，并且根据我们自身系统的特点进行了适配：

> J. Wang *et al*., "Characterizing Crowds to Better Optimize Worker Recommendation in Crowdsourced Testing," in *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1259-1276, 1 June 2021, doi: 10.1109/TSE.2019.2918520.

7.1 简介

Collect使用NSGA-II算法（即非主导的分类遗传算法-II）以优化上述四个目标。NSGA-II是一种在软件工程领域广泛使用的多目标优化器。软件分析中超过65%的优化技术基于遗传算法（对于问题具有单一目标）或NSGA-II（对于多PLE目标问题）。详细的算法介绍请参考下面的论文：

> Deb, K., Agrawal, S., Pratap, A., Meyarivan, T. (2000). A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In: , *et al.* Parallel Problem Solving from Nature PPSN VI. PPSN 2000. Lecture Notes in Computer Science, vol 1917. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45356-3_83

7.2 算法流程

我们算法包括四个主要步骤：

7.2.1 解决方案编码

我们将解决方案编码为二进制变量，每个二进制位代表一个候选工人，其中0代表该候选工人不在推荐集合（解决方案）当中，1代表候选工人在其中。

7.2.2 初始化种群

考虑到我们的系统数据量较小，且服务器算力有限，我们将初始种群大小设置为100，一共执行50次迭代。

7.2.3 基因算子

我们使用单点交叉，BITFLIP突变产生下一代。我们使用二进制锦标赛作为选择操作员哪种解决方案是随机选择的，两者的更优者将在下一个人群中生存。

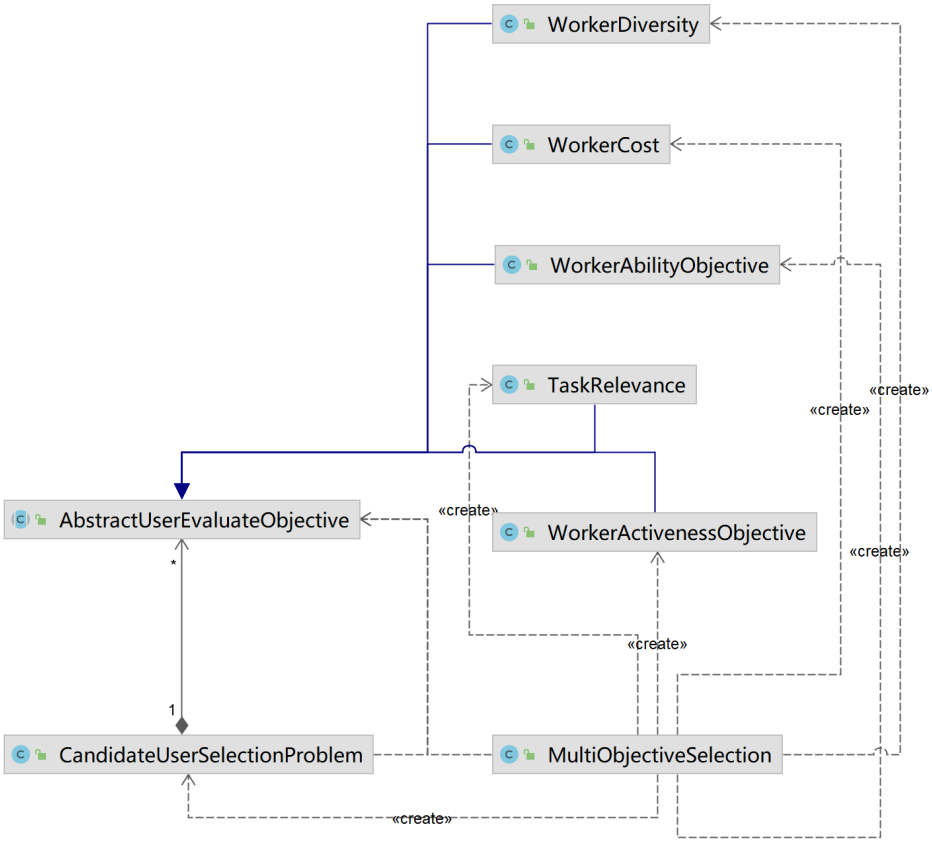
7.2.4 适应函数

我们需要同时优化5个目标，对于推荐集合（解）A和B来说，只有当集合A在所有目标上都不劣于B并且在某个目标上优于B的时候，才认为A是优于B的，否则A和B不可比。这意味着不同的推荐集合之间存在着偏序关系。给定一个解集，我们可以从中选出"极大值"的集合，其中所有的解互相之间都不可比，在每一轮迭代中我们都从集合中选出这些极大值，并将他们从原来的集合中剔除，经过多轮迭代，所有的解都被从中剔除。假设一个解在第n轮迭代中被剔除，那么它的适应函数值就是-n，适应函数值越大，适应度越高。

由此可见，适应函数的值主要有目标决定，而目标已经在之前部分介绍过。

7.3 实现结构

我们的任务分配形式是团队中的每个人负责一个目标或者负责多目标优化算法框架，因此在算法框架的结构上需要在将目标和算法框架解耦。因此我们使用了策略模式进行设计：算法框架作为上下文，多个目标作为策略。它们之间存在最为重要的一个接口，目标接收一个工人推荐集合，为算法提供一个目标值。



在实际的设计当中，MultiObjectiveSelection既承担了工厂的职责，用来产生目标对象也负责封装整个算法的实现，对外提供抽象接口，未将这两种职责分离主要是考虑到项目结构的复杂度。CandidateUserSelectionProblem是目标的上下文，由于项目本身依赖于算法框架，所以该类本身也是一个更大的上下文中的一个策略。

使用该中结构可以便于对目标的扩展。

8. 缓存与定期刷新

8.1 原因/动机

原因：使用缓存和定期刷新的原因是，NSGA-II是一种遗传算法，其中涉及到很多次对于工人推荐集合的评估，这就决定了该算法的效率不会很高，另外，通过之前对于目标的介绍可以得知，除了众包工人开销以外的每个目标都是比较复杂的，可能涉及到多次对于数据库的复杂查询和远程的接口调用。使用缓存可以提高算法整体的运行效率。使用批量处理的重要原因是，推荐的结果对于实时性要求并不高。

动机：在算法框架和目标基本开发好之后，我们进行了简单的集成测试，发现在当时的情况下即使只优化极少量的工人和极少量的任务，并且使用很少的迭代次数（种群数量10，迭代次数10）算法的运行速度依然超乎想象的慢。

除了推荐的结果以外，我们还对于一些计算过程非常复杂的目标值使用定期刷新的策略，因为这些目标值对于实时性的要求并不高，例如工人能力。

8.2 实现

缓存实现

在仔细审查代码之后，我们发现算法执行的过程中存在很多重复的接口调用。

事实上，在算法的开始之前，我们就可以确定一个候选工人全集，之后所有被评估的集合都是该全集的某个子集，而其中很多的目标所采用的策略都是将对于单独的用户计算一个值，然后将推荐集合中所有用户的值进行加权平均或者求和等等，所以目标值只与单独的用户有关，而与用户之间的相互组合无关。对于这类目标，可以在初始化的时候就提供用户全集，一次性计算并且存储所有用户的值（或者在计算单个用户数值的接口上进行加缓存处理），后续调用只需要读取存储好的值并进行求和或者平均。

对于于用户组合密切相关的目标，我们寻找其反复调用的底层接口，并对接口进行缓存处理。

我们使用spring boot中集成ehCache的方式进行自动缓存，或者使用hashmap实现缓存。

举例1：自动缓存

```
package
com.seiii.collect.serviceimpl.recommend.multiobjectiveoptmization.objective.conc
reteobjective;


@Component
public class Util {
    ...

    @Cacheable(cacheNames = "getDomainKnowledge")
    public ResponseVO<List<String>> getDomainKnowledge(Boolean isWorker, Integer
id) {
        //该方法的实现中设计多次数据库查询和网络请求，且被多次以重复的参数调用，因此我们为它配
置缓存。
    }
    ...
}
```

举例2：使用hashmap

```
public class WorkerAbilityObjective extends AbstractUserEvaluateObjective {
    private Map<Integer, Double> workerAbilityMap = new HashMap<>();
    ...

    //不要使用spring自动注入mapper，请将mapper放在从构造方法传入
```

```
    @Override
    protected double calculateValue(List<Integer> userIds) {
        return userIds.stream().mapToDouble(userId ->
this.workerAbilityMap.get(userId)).sum();
    }

    ...
     /**
     * 这个方法会在目标初始化之后被调用，用来预先计算一些中间结果，保证calculateValue方法的告
诉执行。
     * @param candidateUserIds 所有候选用户的id，之后调用calculateValue方法所传入的工人
Id将是该集合的子集。
     */

    @Override
    public void prepareForAllCandidateUsers(List<Integer> candidateUserIds) {
        ComprehensiveAbility comprehensiveAbility = new ComprehensiveAbility();
        candidateUserIds.forEach(
                userId ->
                this.workerAbilityMap.put(
                        userId,
                        comprehensiveAbility.getAbilityValue(userId)));
    }
}
```

定期刷新

我们使用spring boot中的基于注解的方式来定义定时任务。

```
@Component
public class RecommendResultScheduledTask {
    private final static Logger LOGGER =
LoggerFactory.getLogger(RecommendResultScheduledTask.class);

    private final RecommendService recommendService;
    private final UserService userService;

    public RecommendResultScheduledTask(RecommendService recommendService,
UserService userService){
        this.recommendService = recommendService;
        this.userService = userService;
    }

    @Scheduled(cron = "${scheduledTaskTime.refreshRecommendationResult}")
    private void refreshRecommendationResult(){
        LOGGER.info("schedule task begin");
        recommendService.refreshRecommendationResult();
        LOGGER.info("schedule task finished!");
    }

    @Scheduled(cron = "${scheduledTaskTime.refreshWorkerAbility}")
    private void refreshWorkerAbility(){
        LOGGER.info("calculating worker ability......");
        userService.refreshWorkerAbility();
        LOGGER.info("worker ability completed!");
    }
}
```

三、任务缺陷数量预测算法

1. 基本思路

以每份报告为一次Sample，以每一个缺陷树的根节点缺陷（元缺陷）作为到达bug，构建缺陷到达查看表

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	...
Sample #1	1	1	1	0	0	0	0	0	0	0	0	0	
Sample #2	0	0	1	1	0	0	0	0	0	0	0	0	
Sample #3	0	0	1	0	1	0	0	0	0	0	0	0	
Sample #4	0	0	0	1	1	1	1	1	0	0	0	0	
Sample #5	0	0	1	1	0	0	0	1	1	1	1	0	
Sample #6	1	0	1	0	1	0	0	0	0	0	0	1	
Sample #7	...												

使用CRC预测任务的总缺陷数量

公式为

$$N = \frac{D}{C} + \frac{f_1}{C}\gamma^2,\ C = 1 - \frac{f_1}{\sum_{k=1}^{t} k f_k} \tag{1}$$

$$\gamma^2 = max\{\frac{\frac{D}{C}\sum_k k(k-1)f_k}{2\sum\sum_{j<k} n_j n_k} - 1, 0\} \tag{2}$$

各变量含义为

Var.	Meaning	Computation based on bug arrival lookup table	Example value
N	Predicted total number of bugs		predicted value: ***24***
D	Actual number of bugs captured so far	Number of columns	12
t	Number of captures	Number of rows	6
n_j	Number of bugs detected in each capture	Number of cells with *1* in row *j*	3, 2, 2, 5, 6, 4
f_k	Number of bugs captured exactly k times in all captures, i.e., $\sum f_i = D$	Count the number of cells with *1* in each column, and denote as r_i; f_k is the number of r_i with value k	1=7, 2=2, 3=2, 5=1

在本项目中含义为

变量	含义
N	任务待预测的总缺陷数
D	目前已有的元缺陷数
t	任务已有的报告数
nj	报告j中发现的元缺陷的数量
fk	被有且仅有k份报告发现的元缺陷的数量

2.算法固有问题

当所有到达bug都有且仅有一个Sample有发现记录时，C变为0，失去预测效果

$$N = \frac{D}{C} + \frac{f_1}{C}\gamma^2, C = 1 - \frac{f_1}{\sum_{k=1}^{t} k f_k} \quad (1)$$

$$\gamma^2 = max\{\frac{\frac{D}{C}\sum_k k(k-1)f_k}{2\sum\sum_{j<k} n_j n_k} - 1, 0\} \quad (2)$$

3.问题影响分析

考虑到众包测试场景中各测试工人的独立性和多样性，实际环境中出现各Sample之间均无相似bug交叉的情况概率极低，故此问题对算法适用性影响较小

4.对应在本项目中问题

当所有缺陷均以缺陷树根节点存在即没有任何工人提交报告时fork已有缺陷，则失去预测效果

5.解决方案

经过调研和考虑，没有找到可以有效解决此问题的理论支撑，考虑到1.出现几率的极小性；2.出现此问题的场景通常可能为客观上待测制品缺陷数量极少、客观上待测制品缺陷极深以致工人只能偶然发现等，这些情况下已发现的缺陷数都应当接近于正常范围下可以测试出的缺陷数。故综上，对这个问题采用“将预测值设为已发现缺陷数与2的和”作为解决方案。