

哈尔滨工业大学 计算学部

2024 年秋季学期《开源软件开发实践》

Lab 2: 开源软件开发协作流程

姓名	学号	联系方式
李佳欣	2023120244	2502923170@qq.com/13592560663

目 录

1	实验要求	1
2	实验内容 1 发送 pull request	1
2.1	fork 项目	1
2.2	git 操作命令	2
2.3	代码修改	5
2.4	测试类代码	7
2.5	测试通过截图	8
3	实验内容 2 接受 pull request	9
4	实验内容 3 github 辅助工具	12
4.1	熟悉 GoodFirstIssue 工具	12
4.2	安装并使用 Hypercrx	14
4.3	利用 OpenLeaderboard 工具	16
5	小结	17

1 实验要求

实验内容 1 发送 pull request

- 1.在 github 上 fork 题目仓库到个人账号中
- 2.Clone github 上 fork 后的仓库到本地
- 3.创建 fix 分支，在 fix 分支上修改所分配题目代码中的所有错误，编写测试类，测试通过无误后，提交到本地仓库并推送到 github 上
- 4.在 github 上向编程题目所在仓库提交 pull request

实验内容 2 接受 pull request

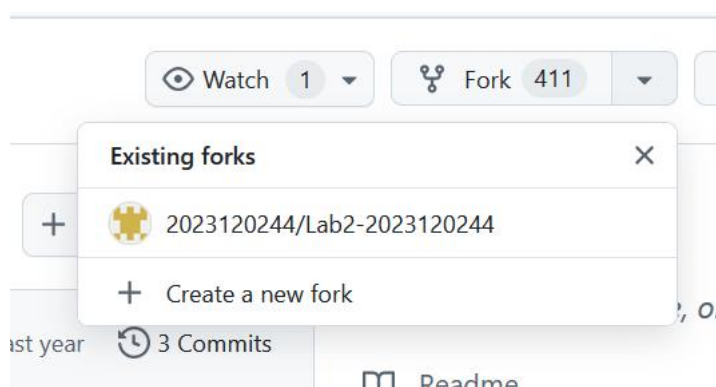
- 1.创建一个新的仓库，将实验内容 1 中建立的代码仓库中所有内容（包含代码修改成功后的文件）复制到新仓库中，任选一位其他同学，相互 fork 对方在实验内容 2 中建立的新仓库。
- 2.评审对方的程序代码和测试用例，对其中的文件进行修改（修改内容不强制要求：修改代码、添加注释、添加评论等均可），然后提交 PR
- 3.通过 PR 进行交流
- 4.检查无误后接受 PR

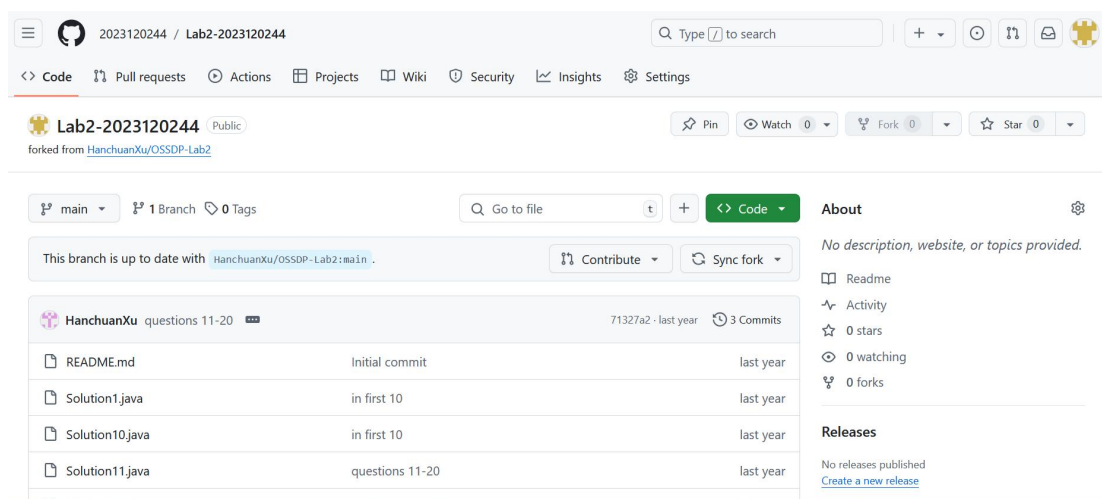
实验内容 3github 辅助工具

- 1.熟悉 <https://goodfirstissue.dev> 网站的使用
- 2.熟悉 <https://open-leaderboard.x-lab.info/>
阅读下面 3 篇文章
 - 活跃度指标: http://blog.frankzhao.cn/how_to_measure_open_source_1/
 - 影响力指标: http://blog.frankzhao.cn/how_to_measure_open_source_2/
 - 价值流网络: http://blog.frankzhao.cn/how_to_measure_open_source_3/
- 3.安装并使用 <https://github.com/hypertrons/hypertrons-crx>

2 实验内容 1 发送 pull request

2.1 fork 项目





2.2 git 操作命令

给出本地 clone、提交、远程推送等步骤的 git 命令和对应的截图

git clone https://github.com/2023120244/OOSDP-Lab2.git 克隆到本地。

```
DELL@jxx MINGW64 /e/Desktop/二学位/开源软件/Lab2
$ git clone https://github.com/2023120244/OOSDP-Lab2.git
Cloning into 'OOSDP-Lab2'...
remote: Enumerating objects: 45, done.
remote: Counting objects: 100% (40/40), done.
remote: Compressing objects: 100% (34/34), done.
remote: Total 45 (delta 6), reused 14 (delta 4), pack-reused 5 (from 1)
Receiving objects: 100% (45/45), 24.29 KiB | 12.15 MiB/s, done.
Resolving deltas: 100% (6/6), done.
```

git checkout -b fix 创建 fix 分支，在 fix 分支上进行修改所分配题目代码。

```
DELL@jxx MINGW64 /e/Desktop/二学位/开源软件/Lab2/OOSDP-Lab2 (main)
$ git checkout -b fix
Switched to a new branch 'fix'

DELL@jxx MINGW64 /e/Desktop/二学位/开源软件/Lab2/OOSDP-Lab2 (fix)
$
```

修改且编写好测试文件后。

git add . 添加到暂存区。

git commit -m “update and test”提交。

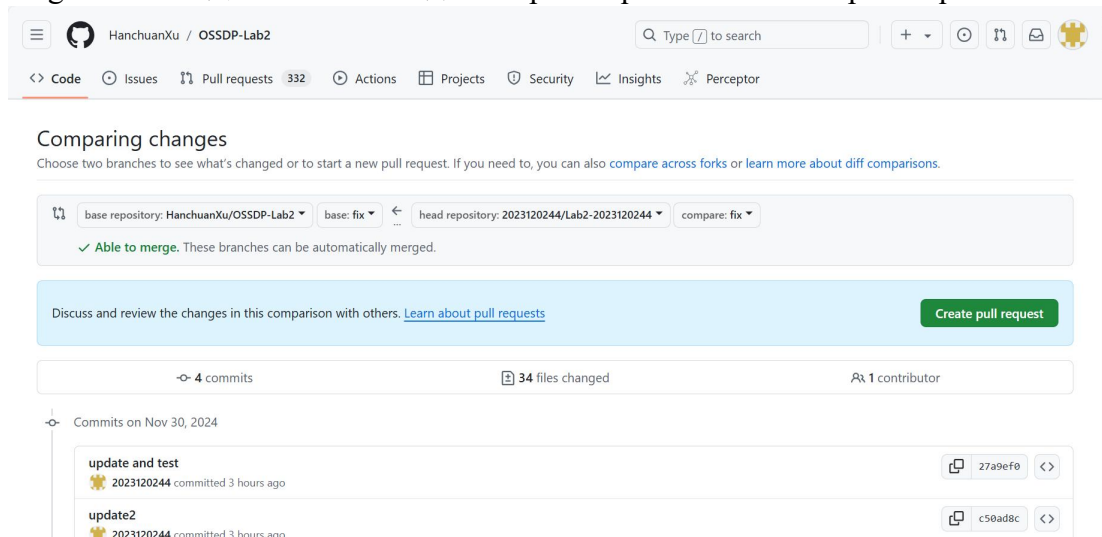
```
DELL@jxx MINGW64 /e/Desktop/二学位/开源软件/Lab2/Lab2-2023120244 (fix)
$ git add .

DELL@jxx MINGW64 /e/Desktop/二学位/开源软件/Lab2/Lab2-2023120244 (fix)
$ git commit -m "update and test"
[fix 27a9ef0] update and test
34 files changed, 145 insertions(+), 1322 deletions(-)
create mode 100644 .idea/.gitignore
create mode 100644 .idea/misc.xml
create mode 100644 .idea/modules.xml
create mode 100644 .idea/vcs.xml
create mode 100644 L2023120244_6_Test.java
create mode 100644 Lab2-2023120244.iml
delete mode 100644 Solution1.java
delete mode 100644 Solution10.java
delete mode 100644 Solution11.java
delete mode 100644 Solution12.java
delete mode 100644 Solution13.java
delete mode 100644 Solution14.java
delete mode 100644 Solution15.java
delete mode 100644 Solution16.java
delete mode 100644 Solution17.java
delete mode 100644 Solution18.java
delete mode 100644 Solution19.java
delete mode 100644 Solution2.java
delete mode 100644 Solution20.java
delete mode 100644 Solution3.java
delete mode 100644 Solution4.java
delete mode 100644 Solution5.java
delete mode 100644 Solution7.java
delete mode 100644 Solution8.java
delete mode 100644 Solution9.java
create mode 100644 out/production/Lab2-2023120244/.idea/.gitignore
create mode 100644 out/production/Lab2-2023120244/.idea/misc.xml
create mode 100644 out/production/Lab2-2023120244/.idea/modules.xml
create mode 100644 out/production/Lab2-2023120244/.idea/vcs.xml
create mode 100644 out/production/Lab2-2023120244/L2023120244_6_Test.class
create mode 100644 out/production/Lab2-2023120244/Lab2-2023120244.iml
create mode 100644 out/production/Lab2-2023120244/README.md
create mode 100644 out/production/Lab2-2023120244/Solution6.class
```


git push -u origin fix 推送到 fix 分支上。

```
DELL@jxx MINGW64 /e/Desktop/二学位/开源软件/Lab2/Lab2-2023120244 (fix)
$ git push -u origin fix
Enumerating objects: 17, done.
Counting objects: 100% (17/17), done.
Delta compression using up to 8 threads
Compressing objects: 100% (13/13), done.
Writing objects: 100% (15/15), 4.04 KiB | 4.04 MiB/s, done.
Total 15 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
remote:
remote: Create a pull request for 'fix' on GitHub by visiting:
remote:   https://github.com/2023120244/Lab2-2023120244/pull/new/fix
remote:
To https://github.com/2023120244/Lab2-2023120244.git
 * [new branch]      fix -> fix
branch 'fix' set up to track 'origin/fix'.
```

在 github 上向编程题目所在仓库提交 pull request, 创建新的 pull request。



填写 pull request 相关内容。



Add a title



Solution6









Helpful resources

GitHub Community Guidelines

Add a description

WritePreview

H B I  < > 

学号: 2023120244, 姓名: 李佳欣

修改思路:

1.数组 s 初始化的循环条件修正:

原错误文件中, 循环条件是 for (inti = 1; i < 105; ++i), 这意味着数组 s 的第一个元素 s[0] 将不会被初始化。

修改后的文件中, 循环条件是 for (inti = 0; i < 105; ++i), 这样确保了数组 s 的所有元素都被正确初始化。

2.循环变量 n 的定义修正:

原错误文件中, n 被定义为 favoriteCompanies.size()-1, 这会导致最后一个元素被忽略。

修改后的文件中, n 被正确定义为 favoriteCompanies.size(), 确保了所有元素都被考虑。

3.遍历遍历 favoriteCompanies 的顺序修正:

原错误文件中, 填充 s 数组的循环和检查是否为子集的循环是嵌套的, 这是逻辑错误, 因为应该先填充所有的集合, 再进行子集的检查。

修改后的文件中, 两个循环是顺序执行的, 首先填充 s 数组, 然后进行子集的检查。


4.检查是否为子集的逻辑修正:


原错误文件中, ans 集合添加索引的条件是 if (isSub), 这意味着如果当前清单是其他清单的子集, 它的索引会被添加到答案中, 这是错误的。


修改后的文件中, ans 集合添加索引的条件是 if (!isSub), 这意味着只有当前清单不是其他任何清单的子集时, 它的索引才会被添加到答案中, 这是正确的逻辑。

5.check 方法的逻辑修正:


原错误文件中, check 方法的逻辑是正确的, 但是因为其他部分的错误, 导致整体结果不正确。

 Markdown is supported

 Paste, drop, or click to add files

 Allow edits by maintainers

Create pull request

 Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

提交后可在列表查看到提交记录。

☰

HanchuanXu / OSSDP-Lab2

🔍 Type to search

+ ▾

🔍

🔗

📧

🧩

<> Code 🔍 Issues 🔗 Pull requests 334 📄 Actions 📁 Projects 🔒 Security 📄 Insights 🚫 Perceptor

Filters ▾ 🔍 is:pr:open

🔗 Labels 9 📌 Milestones 0

New pull request

🔗 334 Open ✓ 31 Closed

Author ▾ Label ▾ Projects ▾ Milestones ▾ Reviews ▾ Assignee ▾ Sort ▾

🔗 Solution6
#366 opened 5 minutes ago by 2023120244

🔗 solution2
#365 opened 6 minutes ago by Mercury-circle

🔗 Fix Solution 18 and Add Unit Tests
#364 opened 3 hours ago by xxx11

🔗 修改的题目号为11
#363 opened 19 hours ago by 2023120252

🔗 15号修改及测试用例
#362 opened 3 days ago by HIT2023120256

🔗 17
#361 opened last week by hogaiz

可以查看提交记录详情。

Menu

HanchuanXu / OSSDP-Lab2

Search

Icons

Code

Issues

Pull requests 334

Actions

Projects

Security

Insights

Perceptor

Solution6 #366

Open 2023120244 wants to merge 4 commits into HanchuanXu:fix from 2023120244:fix

Conversation 0 Commits 4 Checks 0 Files changed 34 +172 -1,322

2023120244 commented now

...

学号: 2023120244, 姓名: 李佳欣

修改思路:

1.数组 s 初始化的循环条件修正:

原错误文件中, 循环条件是 for (int i = 1; i < 105; ++i), 这意味着数组 s 的第一个元素 s[0] 将不会被初始化。修改后的文件中, 循环条件是 for (int i = 0; i < 105; ++i), 这样确保了数组 s 的所有元素都被正确初始化。

2.循环变量 n 的定义修正:

原错误文件中, n 被定义为 favoriteCompanies.size()-1, 这会导致最后一个元素被忽略。修改后的文件中, n 被正确定义为 favoriteCompanies.size(), 确保了所有元素都被考虑。

3.循环遍历 favoriteCompanies 的顺序修正:

原错误文件中, 填充 s 数组的循环和检查是否为子集的循环是嵌套的, 这是逻辑错误, 因为应该先填充完所有的集合, 再进行子集的检查。

修改后的文件中, 两个循环是顺序执行的, 首先填充 s 数组, 然后进行子集的检查。

4.检查是否为子集的逻辑修正:

原错误文件中, ans 集合添加索引的条件是 if (isSub), 这意味着如果当前清单是其他清单的子集, 它的索引会被添加到答案中, 这是错误的。

修改后的文件中, ans 集合添加索引的条件是 if (!isSub), 这意味着只有当当前清单不是其他任何清单的子集时, 它的索引才会被添加到答案中, 这是正确的逻辑。

5.check 方法的逻辑修正:

原错误文件中, check 方法的逻辑是正确的, 但是因为其他部分的错误, 导致整体结果不正确。

修改后的文件中, check 方法的逻辑保持不变, 因为它本身是正确的。

Reviewers

No reviews

Still in progress? [Convert to draft](#)

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

2.3 代码修改

将修改完善后的原文件粘贴在此处

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

/**
 * @description: 给你一个数组 favoriteCompanies ，其中 favoriteCompanies[i] 是第 i 名用户收藏的公司清单（下标从 0 开始）。
 * <p>
 * 请找出不是其他任何人收藏的公司清单的子集的收藏清单，并返回该清单下标。下标需要按升序排列。
 * <p>
 * <p>
 * 示例 1:
 * <p>
 * 输入:
favoriteCompanies =
[["leetcode","google","facebook"],["google","microsoft"],["google","facebook"],["google"],["amazon"]]
 * 输出: [0,1,4]
 * 解释:
 *
favoriteCompanies[2]=["google","facebook"] 是
favoriteCompanies[0]=["leetcode","google","facebook"] 的子集。
 * favoriteCompanies[3]=["google"] 是 favoriteCompanies[0]=["leetcode","google","facebook"]
和 favoriteCompanies[1]=["google","microsoft"] 的子集。
 * 其余的收藏清单均不是其他任何人收藏的公司清单的子集，因此，答案为 [0,1,4] 。
 * 示例 2:
 * <p>
 * 输入:
favoriteCompanies =
[["leetcode","google","facebook"],["leetcode","amazon"],["facebook","google"]]
 * 输出: [0,1]
 * 解释:
favoriteCompanies[2]=["facebook","google"] 是
favoriteCompanies[0]=["leetcode","google","facebook"] 的子集，因此，答案为 [0,1] 。
 * 示例 3:
 * <p>
 * 输入: favoriteCompanies = [["leetcode"],["google"],["facebook"],["amazon"]]
 * 输出: [0,1,2,3]
```

```
* <p>
* <p>
* 提示:
* <p>
* 1 <= favoriteCompanies.length <= 100
* 1 <= favoriteCompanies[i].length <= 500
* 1 <= favoriteCompanies[i][j].length <= 20
* favoriteCompanies[i] 中的所有字符串 各不相同 。
* 用户收藏的公司清单也 各不相同 ， 也就是说，即便我们按字母顺序排序每个清单，
favoriteCompanies[i] != favoriteCompanies[j] 仍然成立。
* 所有字符串仅包含小写英文字母。
*/
public class Solution6 {
    Set<String>[] s = new Set[105];

    public List<Integer> peopleIndexes(List<List<String>> favoriteCompanies) {
        for (int i = 0; i < 105; ++i) {
            s[i] = new HashSet<String>();
        }
        int n = favoriteCompanies.size();
        List<Integer> ans = new ArrayList<Integer>();

        for (int i = 0; i < n; ++i) {
            for (String com : favoriteCompanies.get(i)) {
                s[i].add(com);
            }
        }

        for (int i = 0; i < n; ++i) {
            boolean isSub = false;
            for (int j = 0; j < n; ++j) {
                if (i == j) {
                    continue;
                }
                isSub |= check(favoriteCompanies, i, j);
            }
            if (!isSub) {
                ans.add(i);
            }
        }

        return ans;
    }
}
```



```
public boolean check(List<List<String>> favoriteCompanies, int x, int y) {  
    for (String com : favoriteCompanies.get(x)) {  
        if (!s[y].contains(com)) {  
            return false;  
        }  
    }  
    return true;  
}  
}
```

2.4 测试类代码

[将测试类代码的原文件粘贴在此处](#)

```
/**  
 * 测试类 L2023120244_6_Test 用于验证 Solution6 类中的 peopleIndexes 方法。  
 *  
 * === 测试用例设计总体原则 ===  
 * - 等价类划分：根据输入参数的特性将可能的输入值划分为几个逻辑等价类，然后为每一类选择一个或多个具有代表性的测试用例。  
 * - 边界值分析：针对输入范围的边界值进行测试，例如数组长度的最大最小值、列表为空等。  
 * - 错误猜测法：基于经验和直觉预测程序中可能出现的问题，构造针对性的测试用例。  
 * - 覆盖率：确保测试覆盖所有功能分支，包括正常路径和异常处理路径。  
 *  
 * === 测试方法 testPeopleIndexesWithSubsets ===  
 * 测试目的：验证当存在收藏清单是其他清单子集的情况时，能够正确找出不是任何其他  
人收藏清单子集的收藏清单。  
 * 用到的测试用例：  
 * - 示例 1：包含多种不同长度的公司清单，其中一些是另一些的子集。  
 * - 示例 2：包含部分相同公司的两个清单，其中一个明显是另一个的子集。  
 * 测试用例设计说明：  
 * - 每个测试用例均涵盖了不同的等价类，如不同的清单长度、清单间的关系（子集关系）  
等。  
 * - 边界情况也被考虑在内，比如最短的清单只有一个元素。  
 *  
 * === 测试方法 testPeopleIndexesWithoutSubsets ===  
 * 测试目的：检查当所有收藏清单均不为其他清单子集时，能否返回全部索引。  
 * 用到的测试用例：  
 * - 示例 3：每个用户的收藏清单都是唯一的，没有一个清单是另一个的子集。  
 * 测试用例设计说明：  
 * - 此测试用例旨在检验算法是否能正确处理完全独立的收藏清单。  
 * - 它同样属于等价类划分的一部分，即所有清单互不为子集这一类别。  
 */
```

```
import java.util.*;

public class L2023120244_6_Test {
    public static void main(String[] args) {
        Solution6 solution = new Solution6();

        // 测试用例 1 包含多种不同长度的公司清单，其中一些是另一些的子集。
        List<List<String>> favoriteCompanies1 = Arrays.asList(
            Arrays.asList("leetcode", "google", "facebook"),
            Arrays.asList("google", "microsoft"),
            Arrays.asList("google", "facebook"),
            Arrays.asList("google"),
            Arrays.asList("amazon")
        );
        System.out.println(solution.peopleIndexes(favoriteCompanies1)); // 预期输出: [0, 1, 4]

        // 测试用例 2 包含部分相同公司的两个清单，其中一个明显是另一个的子集。
        List<List<String>> favoriteCompanies2 = Arrays.asList(
            Arrays.asList("leetcode", "google", "facebook"),
            Arrays.asList("leetcode", "amazon"),
            Arrays.asList("facebook", "google")
        );
        System.out.println(solution.peopleIndexes(favoriteCompanies2)); // 预期输出: [0, 1]

        // 测试用例 3 每个用户的收藏清单都是唯一的，没有一个清单是另一个的子集。
        List<List<String>> favoriteCompanies3 = Arrays.asList(
            Arrays.asList("leetcode"),
            Arrays.asList("google"),
            Arrays.asList("facebook"),
            Arrays.asList("amazon")
        );
        System.out.println(solution.peopleIndexes(favoriteCompanies3)); // 预期输出: [0, 1, 2,
3]
    }
}
```

2.5 测试通过截图

单元测试要求：

```
* 示例 1:
* <p>
* 输入: favoriteCompanies = [["leetcode", "google", "facebook"], ["google", "microsoft", "amazon"], ["google", "facebook"], ["amazon", "apple", "facebook", "microsoft", "twitter"], ["leetcode", "google", "facebook"]]
* 输出: [0,1,4]
* 解释:
* favoriteCompanies[2]=["google", "facebook"] 是 favoriteCompanies[0]=["leetcode", "google", "facebook"] 的子集。
* favoriteCompanies[3]=["google", "facebook"] 是 favoriteCompanies[0]=["leetcode", "google", "facebook"] 的子集。
* 其余的收藏清单均不是其他任何人收藏的公司清单的子集，因此，答案为 [0,1,4] 。
* 示例 2:
* <p>
* 输入: favoriteCompanies = [["leetcode", "google", "facebook"], ["leetcode", "amazon", "facebook", "microsoft"], ["facebook", "google"]]
* 输出: [0,1]
* 解释: favoriteCompanies[2]=["facebook", "google"] 是 favoriteCompanies[0]=["leetcode", "google", "facebook"] 的子集。
* 示例 3:
* <p>
* 输入: favoriteCompanies = [["leetcode"], ["google"], ["facebook"], ["amazon"]]
* 输出: [0,1,2,3]
```

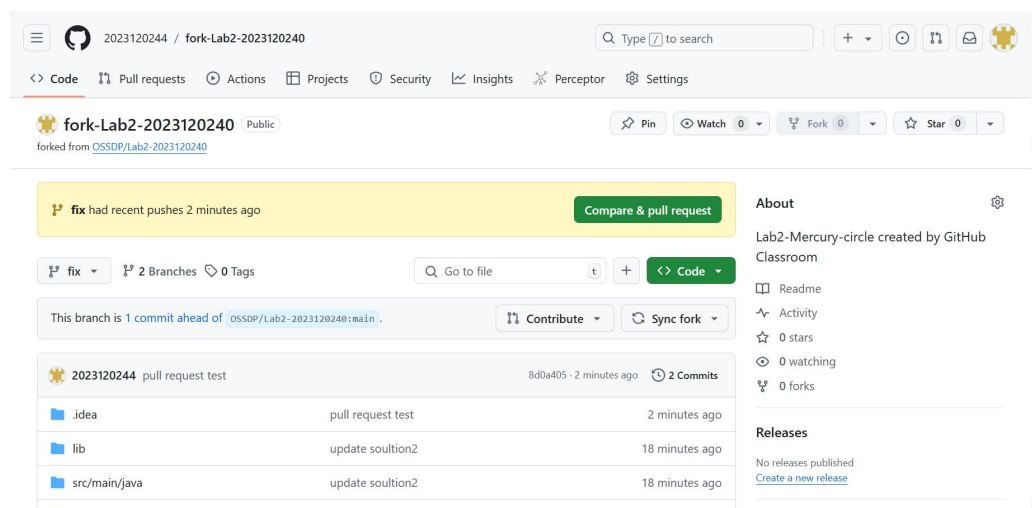
单元测试结果:

```
L2023120244_6_Test x
| [0, 1, 4]
| [0, 1]
| [0, 1, 2, 3]
Process finished with exit code 0
```

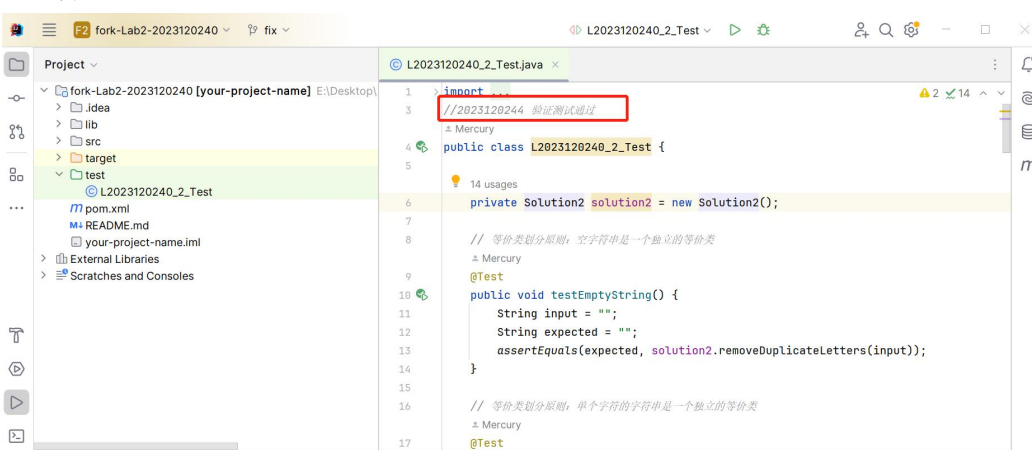
3 实验内容 2 接受 pull request

将双方通过 [github](#) 上 PR 交流的记录截图粘贴在此处

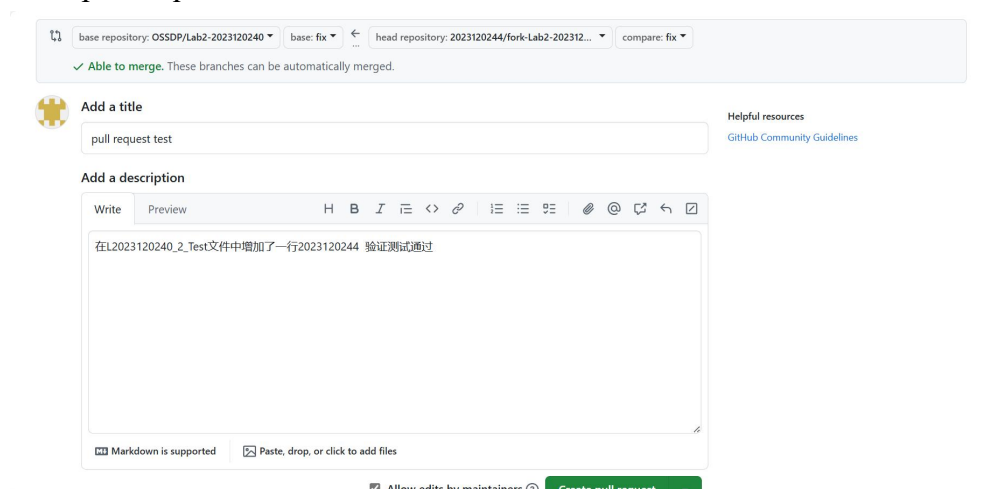
fork 对方的仓库，并克隆到本地。



创建 fix 分支，在 fix 分支上修改内容，在 L2023120240_2_Test 文件中增加了一行 2023120244 验证测试通过。将修改后的内容进行 git add、git commit、git push 操作。

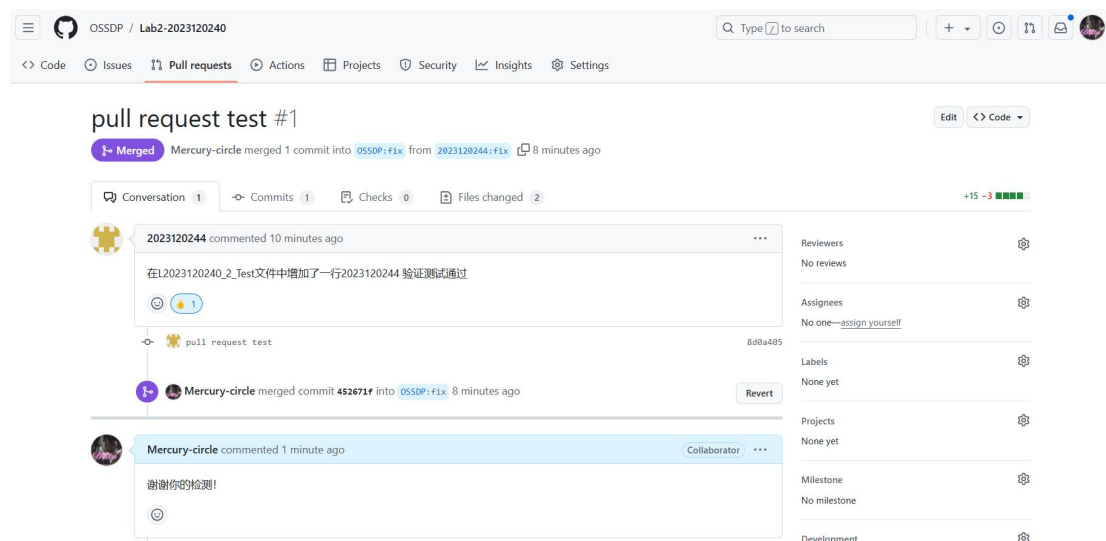


创建 pull request，填写相关内容。

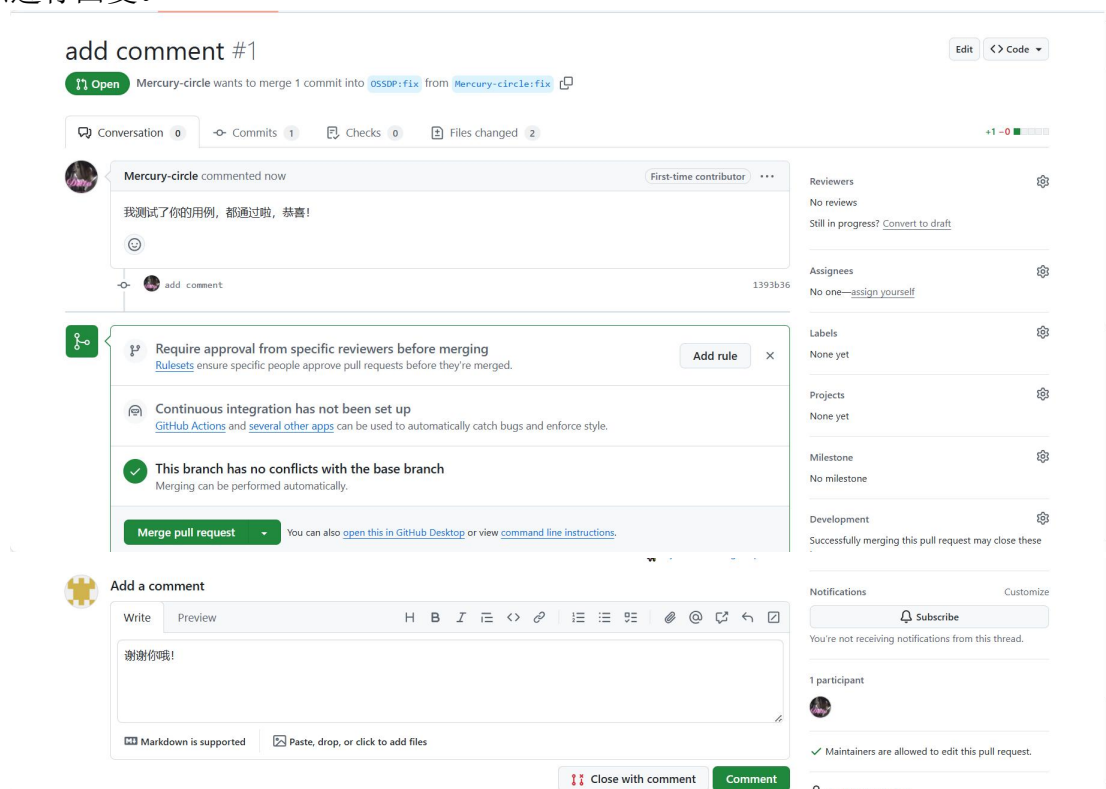


创建 pull request 成功后，对方进行 pull request 的审核，对方审核通过，回

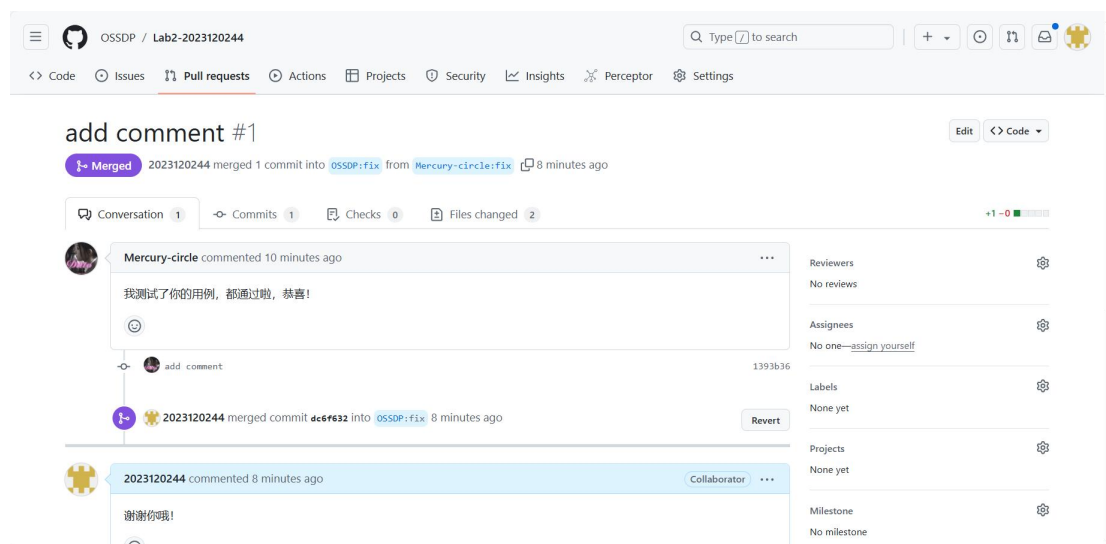
复。页面如下：



此处是自己处理对方的 pull request。审核通过可点击 Merge pull request，且可以进行回复。



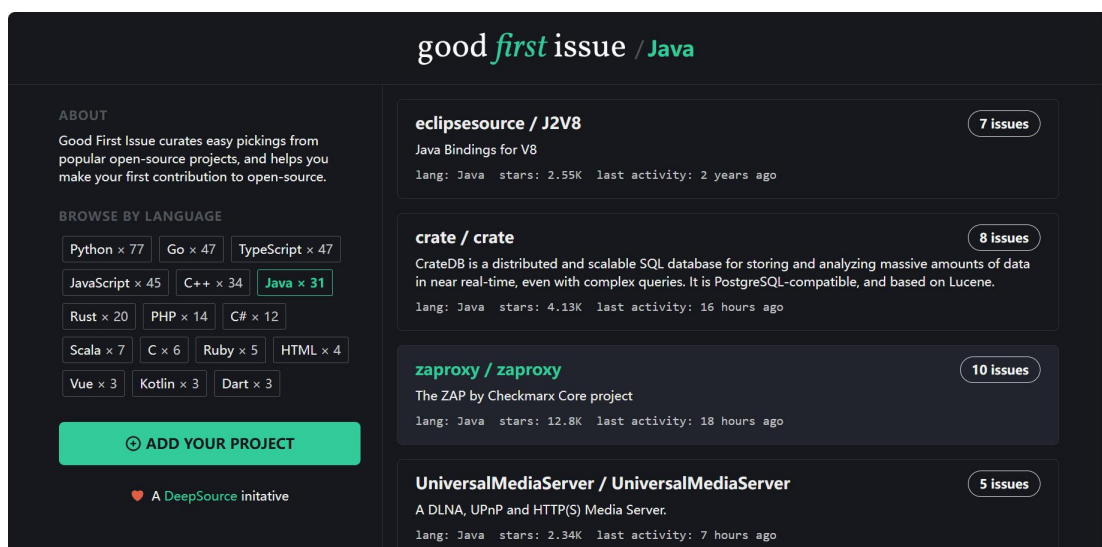
此处是自己仓库中 pull request 处理后的页面：



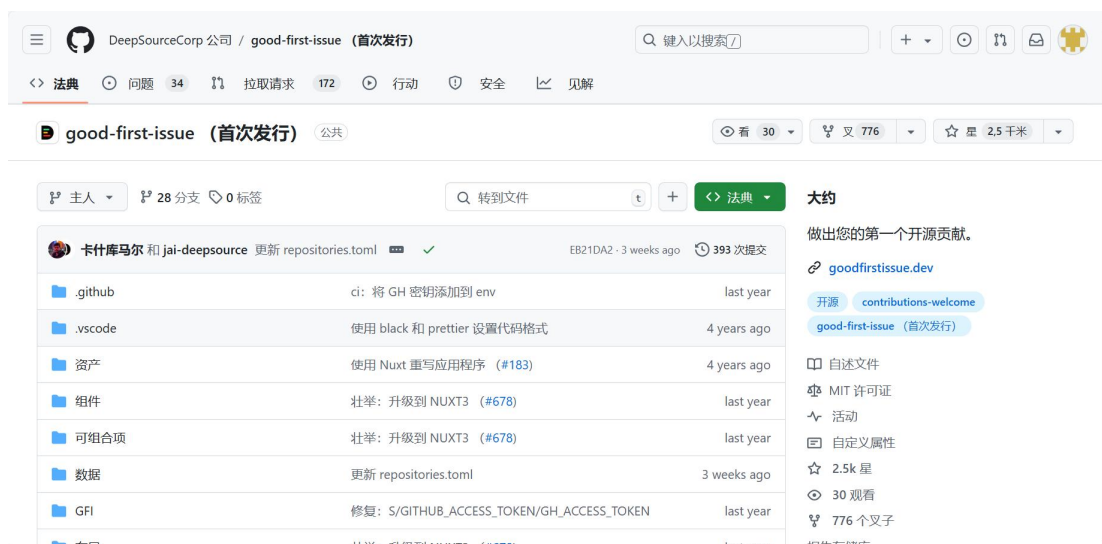
4 实验内容 3 github 辅助工具

4.1 熟悉 GoodFirstIssue 工具

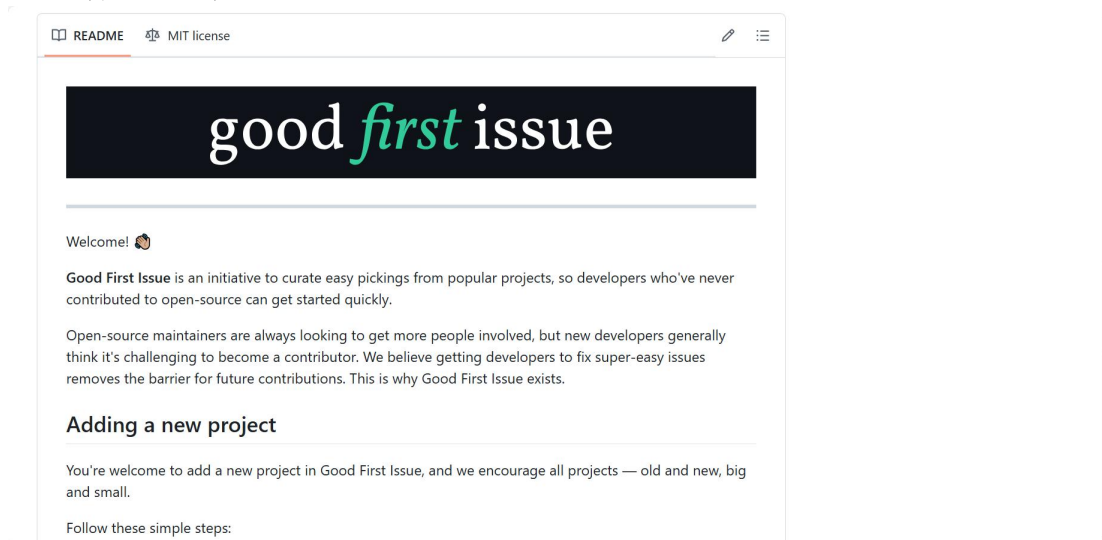
<https://goodfirstissue.dev> 网站可在左侧点击相关的语言挑选自己想要的项目。



点击 ADD YOUR PROJECT 可进入以下页面：



可查看项目文档：



查阅项目的文档，阐述如何使自己的开源项目被此网站收录

步骤 1：确保项目符合要求

为了保证项目质量，并帮助新手更容易地开始他们的第一个开源贡献，Good First Issue 对项目设定了几个基本的要求：

- 1.至少有三个带有 **good first issue** 标签的问题：确保你的 GitHub 仓库中有一些问题已经被标记为适合新手解决。如果默认没有这些标签，你可以手动添加。
- 2.至少有 10 位贡献者：这表明项目有一定的活跃度和社区支持。
- 3.包含详细的 README.md 文件：其中应包含项目的安装指南和使用说明。
- 4.提供 CONTRIBUTING.md 文件：为新贡献者提供指导方针，如如何提交代码、遵循的编码规范等。
- 5.项目处于积极维护状态：这意味着项目定期更新，对 pull request 和 issue 有及时响应。

步骤 2: 提交项目信息

一旦确认项目满足了上述条件，接下来就可以将项目提交给 Good First Issue:

1. 添加仓库路径：你需要在 `data/repositories.toml` 文件中按字典顺序添加你的仓库路径。
2. 创建一个 Pull Request (PR)：在 PR 描述中加入指向你仓库 issue 页面的链接。当 PR 被合并后，你的项目就会出现在 `goodfirstissue.dev` 上了。

步骤 3: 设置本地环境（可选）

如果还打算为 Good First Issue 的网站本身做贡献或进行测试，那么你需要设置本地开发环境：

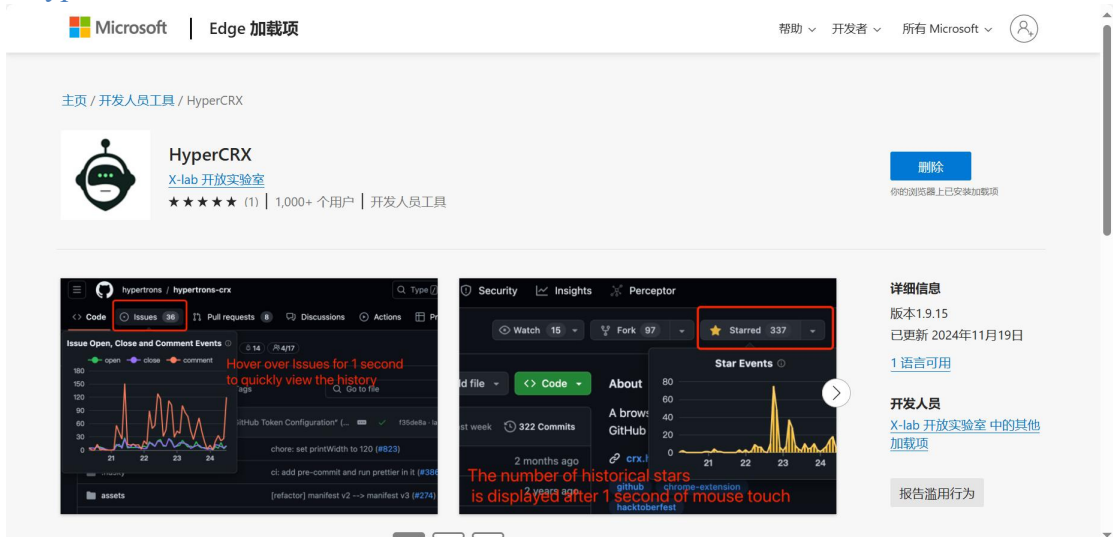
1. 克隆项目：首先需要从 GitHub 上克隆 Good First Issue 的仓库到本地计算机，并确保安装了 Python 3 和最新版本的 Node.js。
2. 复制示例数据文件：执行命令 `$ cp data/generated.sample.json data/generated.json` 和 `$ cp data/tags.sample.json data/tags.json`，以确保前端应用可以正常工作。
3. 构建和启动前端应用：通过运行 `$ bun install` 安装依赖项，然后使用 `$ bun dev` 启动开发服务器。
4. 访问应用：此时，你应该能够在浏览器中打开应用程序并查看其运行情况。

通过以上步骤，可以使自己的开源项目被 Good First Issue 网站收录，还可以参与到该平台本身的改进工作中来，从而更好地服务于开源社区的新成员。

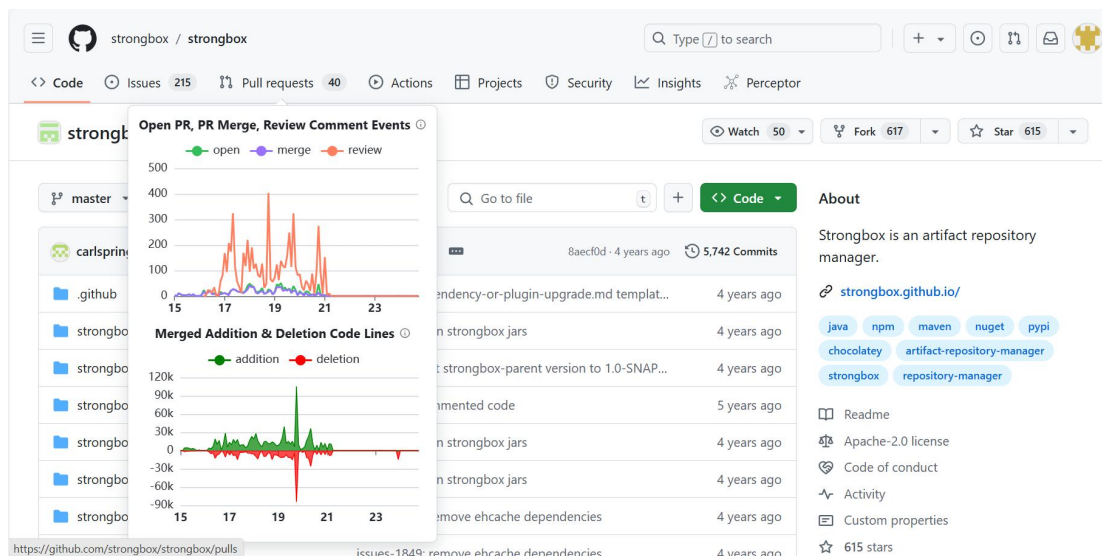
4.2 安装并使用 Hypercrx

访问此网站 <https://github.com/hypertrons/hypertrons-crx>

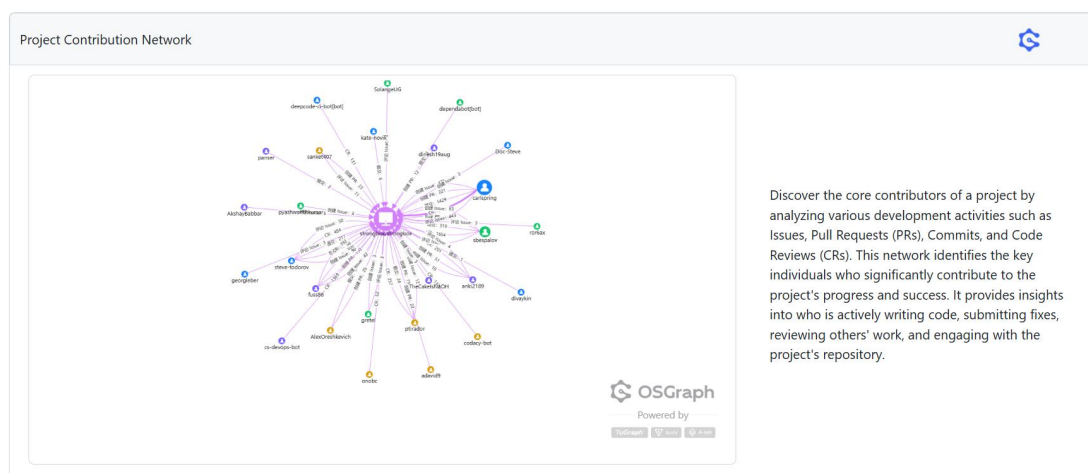
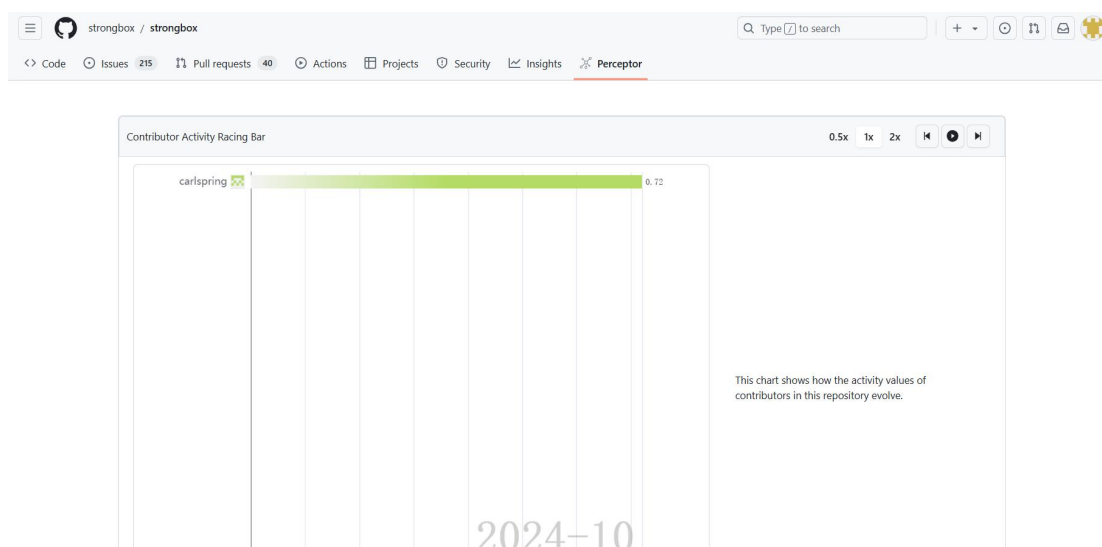
安装 Hypercrx 插件

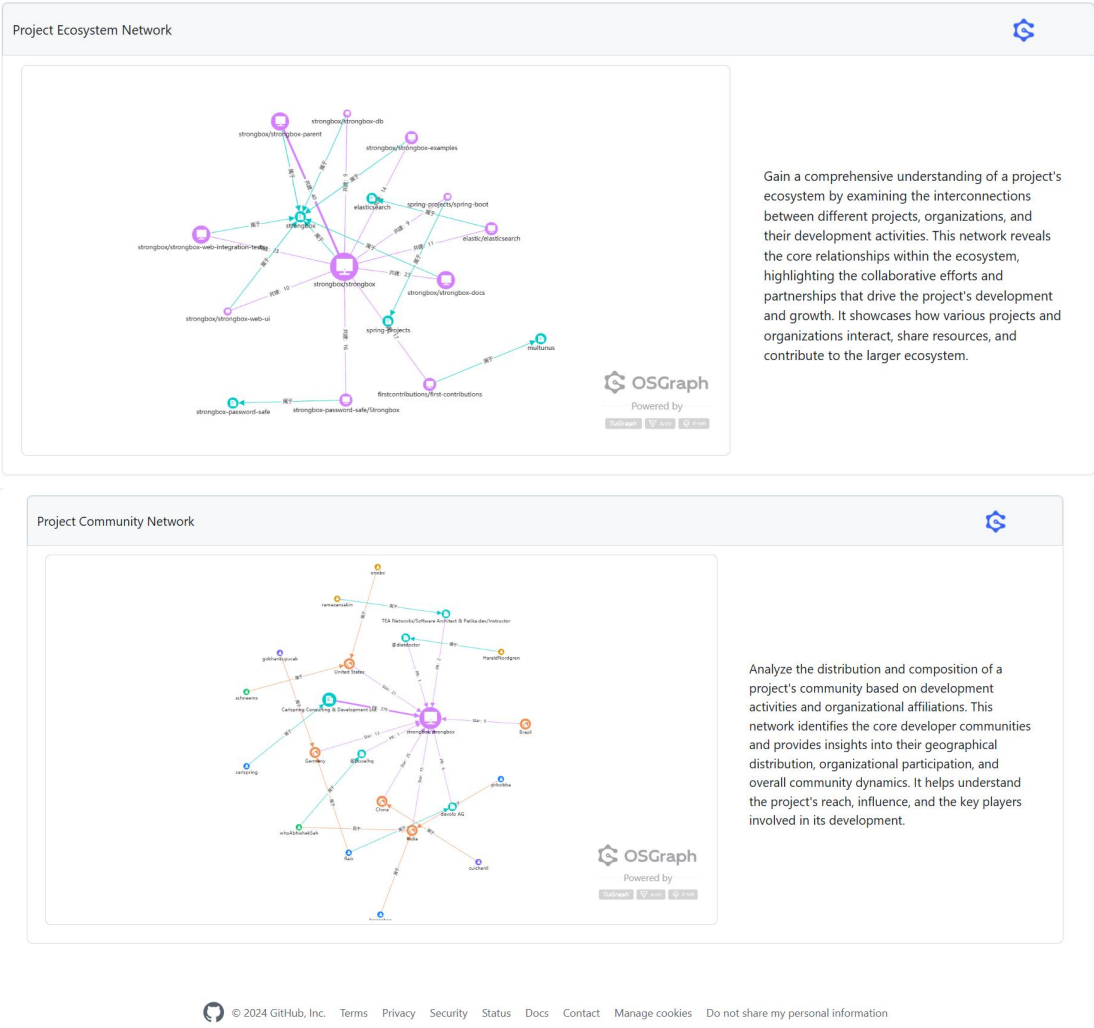


找到某个 star 在 100+ 的项目，给出利用插件查询项目状态的截图。例如下图：



对比下使用此插件前后的效果。使用插件前不能看到可视化的数据，使用后可看到。





通过 Hypercrx 插件可以可视化地看到项目关系网络图、项目活跃开发者协作网络图、项目活跃度&OpenRank 趋势图、仓库详情、开发者协作网络图、活跃仓库网络图、开发者活跃度&OpenRank 趋势图。

4.3 利用 OpenLeaderboard 工具

利用 OpenLeaderboard 工具了解 github 上开源项目的统计情况
阅读实验手册中指定的三篇文章，概括开源项目活跃度、影响力指标、价值流网络的含义，OpenRank 的计算原理。

开源项目活跃度

开源项目的活跃度是通过加权统计 GitHub 上的行为数据来衡量的。开发者在项目中的各种行为（如评论 Issue、发起 Issue、发起 Pull Request(PR)、PR 上的代码 review 评论、PR 合入）被赋予不同的权重分值（1-5 分），以此来量化开发者的活跃度。随后，所有开发者的活跃度会通过一种加权的方式汇总为项目的活跃度，其中采用了开方操作以降低核心开发者过高活跃度的影响，使得更多贡献者参与的社区能够获得更高的评分。

影响力指标

影响力指标是基于全域开发者协作网络构建的一种评估方法，它不仅考虑了

单个项目内部的行为，还分析了不同项目之间的协作关联。该指标利用了 PageRank 算法的思想，认为一个影响力较大的项目会与更多的项目有协作关系，并且与其关联度高的项目也会具有较大的影响力。这种方法避免了一些活跃度指标中存在的问题，例如自动化行为造成的异常高活跃度以及不完全使用 GitHub 功能导致的活跃度偏低的问题。

价值流网络

价值流网络旨在从社会价值的角度对开源软件进行更全面的评价，包括生产和消费两个方面。生产侧关注的是开发者贡献的质量及其产生的实际价值；消费侧则强调软件是否真正被用户所使用并解决了现实需求。此模型引入了更多元的数据关系，比如开发者之间的关注关系、项目间的依赖关系等，并尝试衡量每个实体（如开发者、项目、公司等）的价值流动。最终目标是构建一个反映真实社会效用的价值评估体系。

OpenRank 的计算原理

OpenRank 结合了上述三种概念，形成了一种综合性的评价框架。它首先通过活跃度指标识别出活跃的参与者和项目，然后借助影响力指标捕捉到项目间的关系网，最后利用价值流网络进一步细化各参与方的价值传递路径。具体来说，OpenRank 可能包含以下几个步骤：

1. 活跃度计算：为每个开发者在各个项目中的活动打分，并汇总成项目级别的活跃度得分。
2. 协作网络构建：基于活跃度得分建立项目间的协作关联图谱。
3. PageRank 应用：应用 PageRank 算法计算各节点（项目或开发者）在整个网络中的相对重要性。
4. 价值流动模拟：定义价值流动规则，模拟生产侧（如代码贡献）到消费侧（如软件使用）的价值转移过程。
5. 综合评分生成：整合以上信息得出最终的 OpenRank 分数，这反映了项目在整个开源生态系统中的整体表现和社会影响。

综上所述，OpenLeaderboard 工具通过对活跃度、影响力及价值流网络三个维度的数据进行处理，提供了一个更加全面、准确的方式来理解和评估 GitHub 上的开源项目。

5 小结

通过 Lab 2 实验，我深入掌握了基于 GitHub 的开源协作开发技能，包括 Fork、Clone 仓库、创建分支进行代码修改、编写测试类确保代码质量，以及提交 Pull Request。这些实践不仅提升了我的技术能力，如 Git 操作和单元测试的应用，还强化了我的团队合作意识。在与同学互审代码的过程中，我体验到了交流不同观点和技术方案的重要性，这有助于共同进步并完善。

此外，通过查看 <https://goodfirstissue.dev> 网站中项目文档，了解了如何使自己的开源项目被此网站收录。使用 Hypercrx 插件，让我体会到了可视化数据对于开发和了解项目的益处。阅读关于衡量开源项目的三篇文章（活跃度指标、影响力指标、价值流网络），让我了解到评估一个开源项目健康与否的不同维度。

这次实验为我未来参与真实的开源项目打下了坚实的基础,增强了我在开放环境中高效工作的能力。