实验2 Teambition、Git、软件协作开发工作流

2.2 Teambition团队协作

2.1.1 实验目的

了解如何使用Teambition进行团队协作、项目管理，在Teambition创建项目小组。

2.1.2 实验要求与内容

要求

- 在Teambition上组建项目团队。
- 掌握如何使用Teambition进行团队协作。

内容

一、Teambition账号注册

1. 前往 Teambition 官网：https://www.teambition.com/官网进行账号注册。

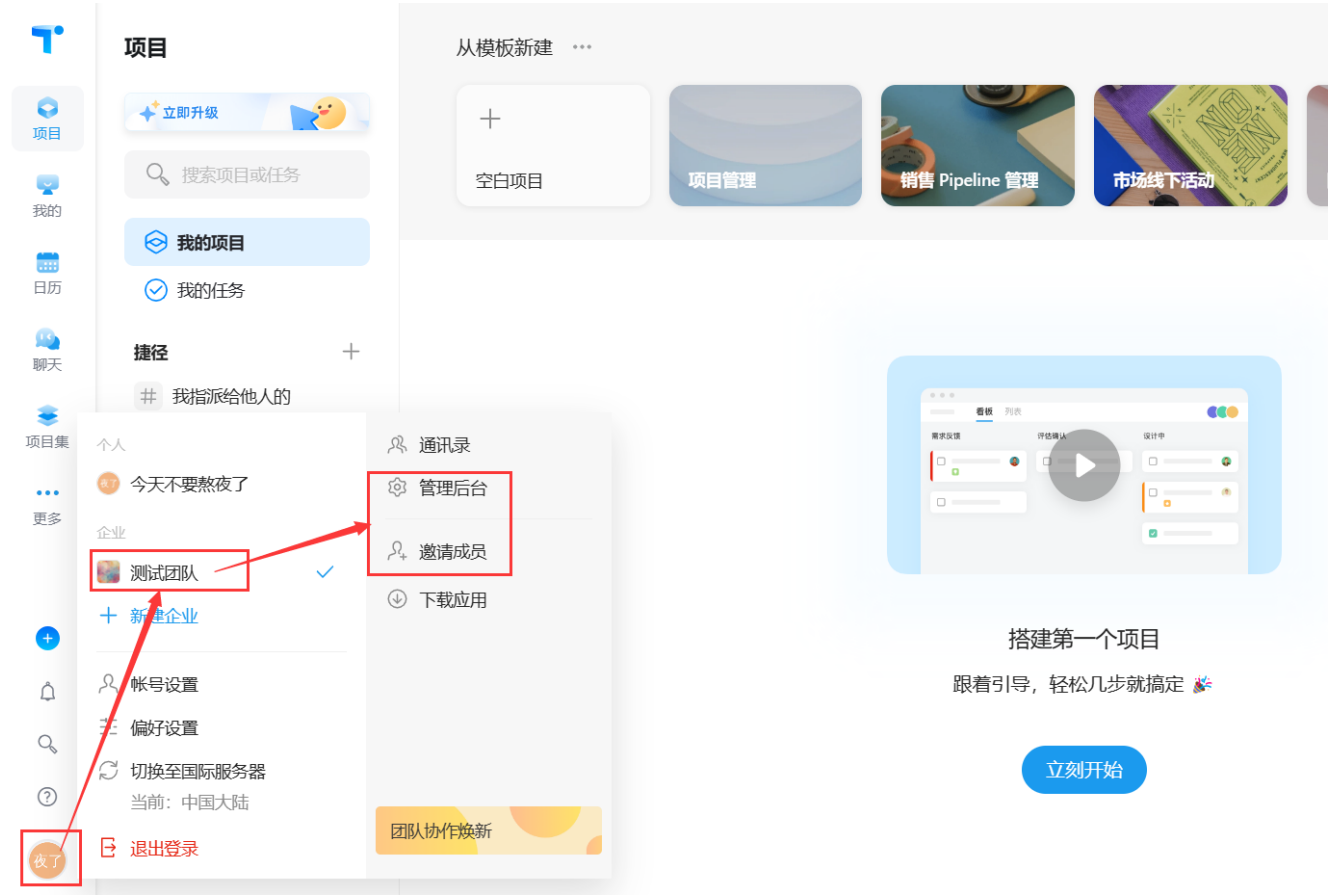
   遇到问题，可查看 Teambition 官方帮助手册 (https://support.teambition.com/help)了解Teambition的相关操作。

   推荐查看“快速了解 Teambition”(https://alidocs.dingtalk.com/i/p/nb9XJl7k8dxRPGyA/docs/OBldywvrKxo89xLzzLd3JQk2ngpNbLz4)的短视频迅速上手Teambition。

2. 输入团队名称，行业可以设置为无行业，选择1～10人团队规模，完成团队创建。

二、团队管理

1. 在初始界面点击左下角用户头像,可在账号设置—>创建的团队（如图中：测试团队）邀请成员进入团队。

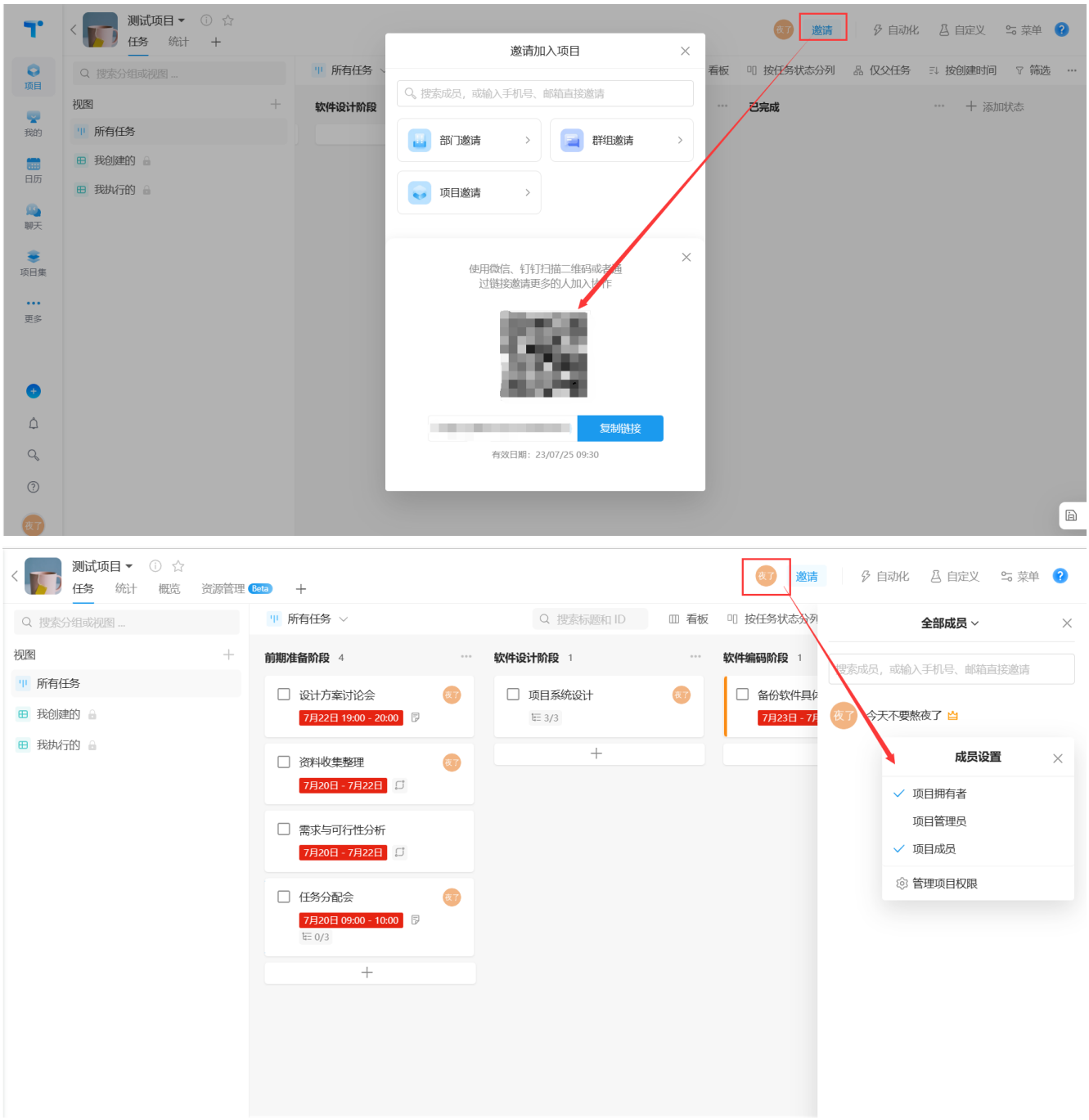


2. 在"管理后台"中，可以对团队(或企业)按需修改团队名称，图片，以及变更团队的成员。

三、项目管理

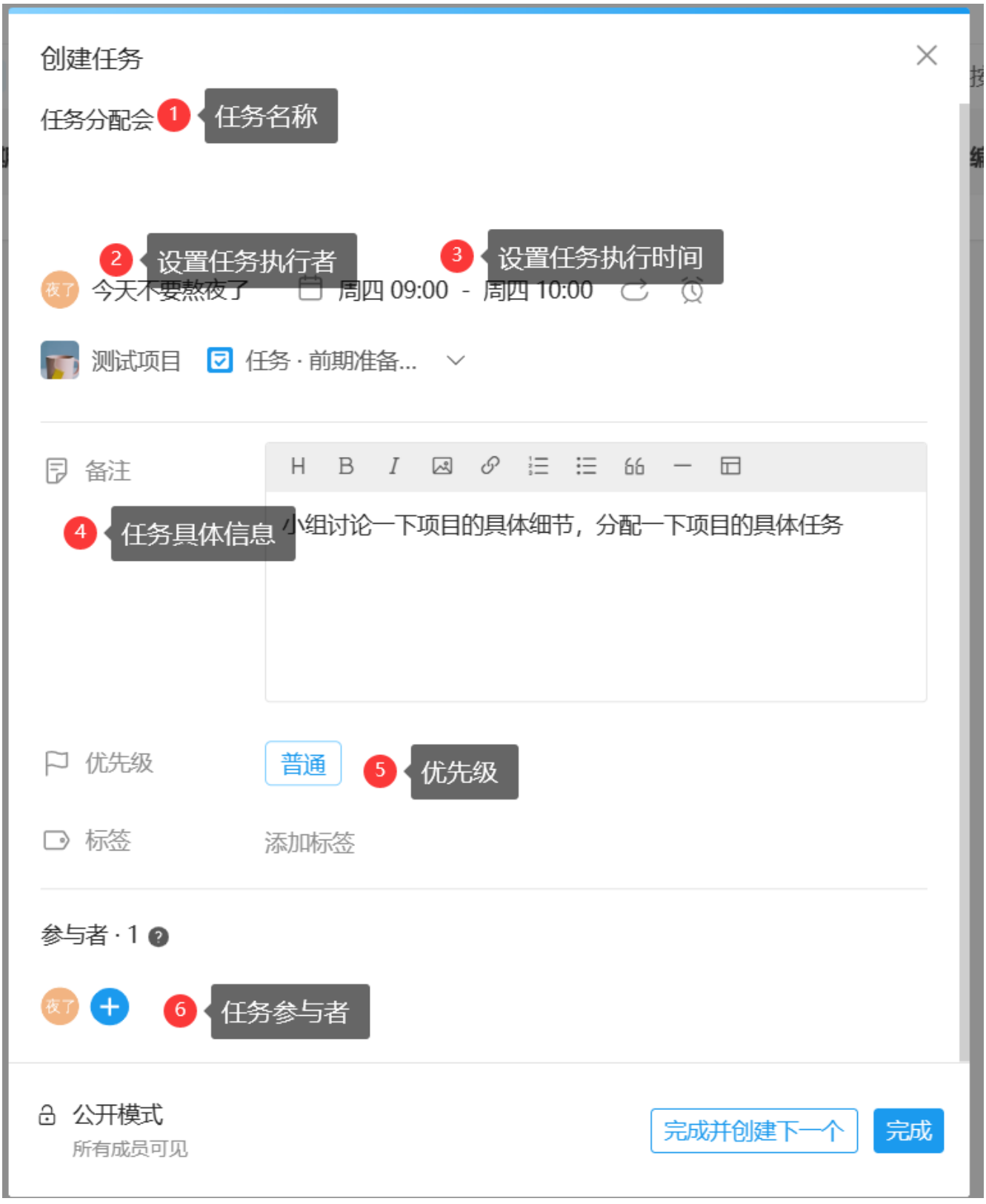
1. 在初始界面点击新建空白项目，完善项目信息，也可选用相关项目模板创建。

2. 邀请团队成员加入项目，可在成员界面管理成员身份。

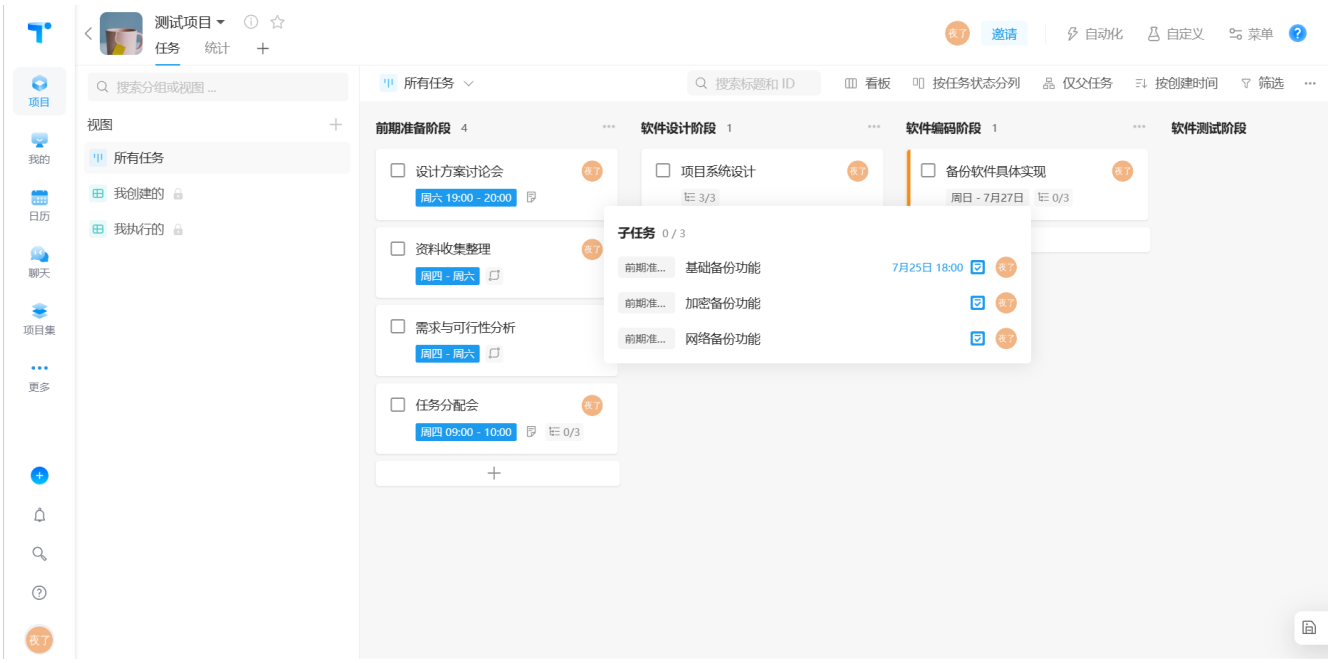


3. 设置项目对应阶段，阶段可以自行添加/删除，阶段顺序可以拖动改变。如本例中，在项目阶段中依次添加“前期准备阶段”、“软件设计阶段”、“软件编码阶段”。实际可跟需要自行设置。

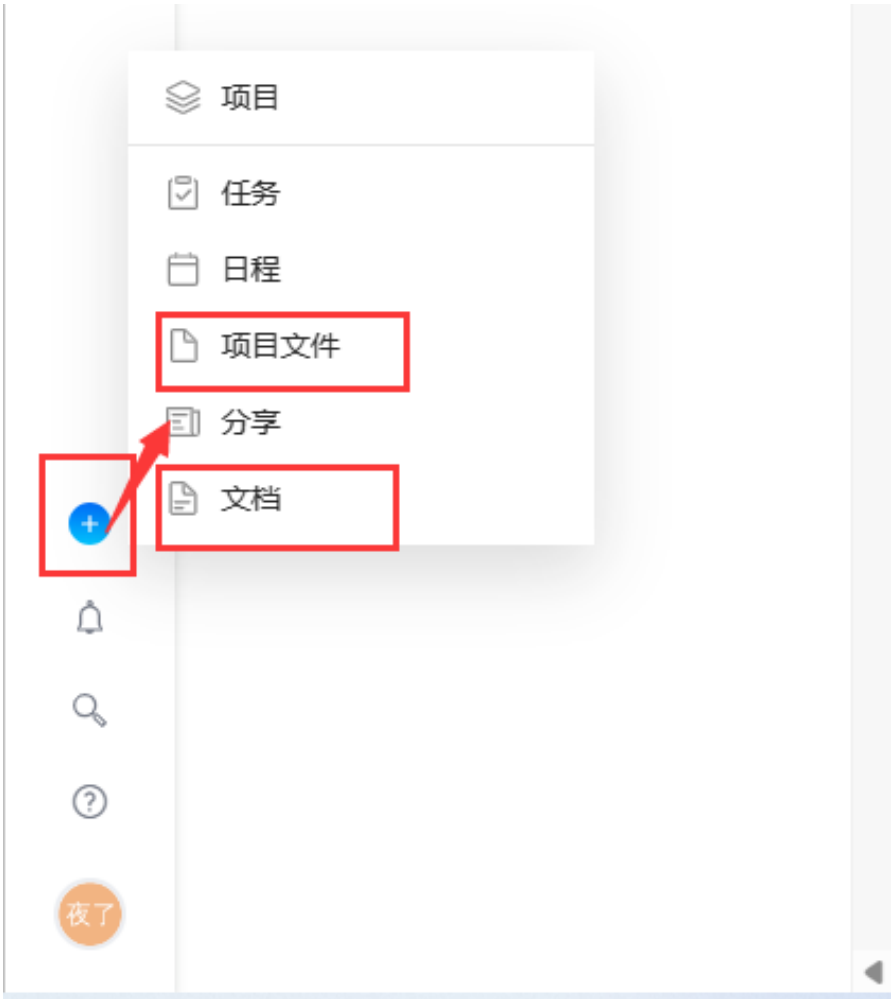
4. 在对应阶段根据需要添加相应任务。


创建任务
任务分配会
1
任务名称
2
设置任务执行者
3
设置任务执行时间
今天不要熬夜了
周四 09:00 - 周四 10:00
测试项目
任务 · 前期准备...
备注
4
任务具体信息
小组讨论一下项目的具体细节，分配一下项目的具体任务
优先级
普通
5
优先级
标签
添加标签
参与者 · 1
6
任务参与者
公开模式
所有成员可见
完成并创建下一个
完成

下面是一个简单的例子：

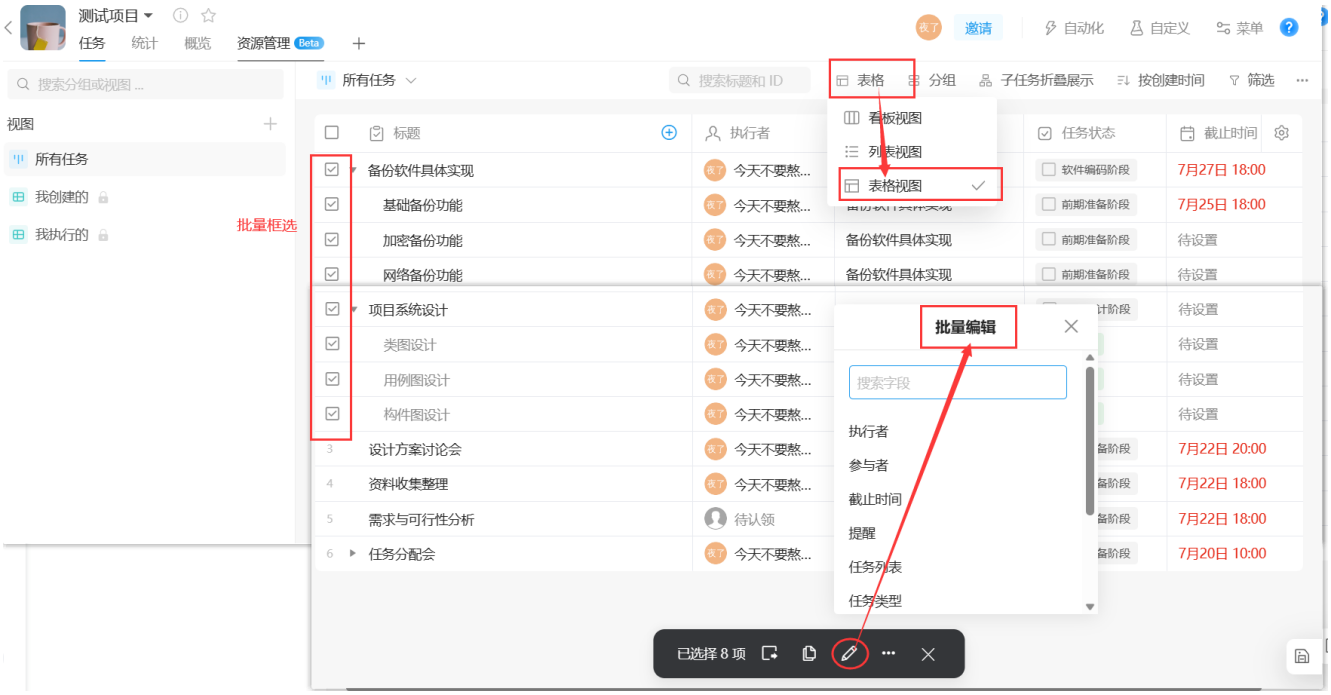


5. 可在左下角点击"+"号添加，任务所需的文档和项目文件资源。

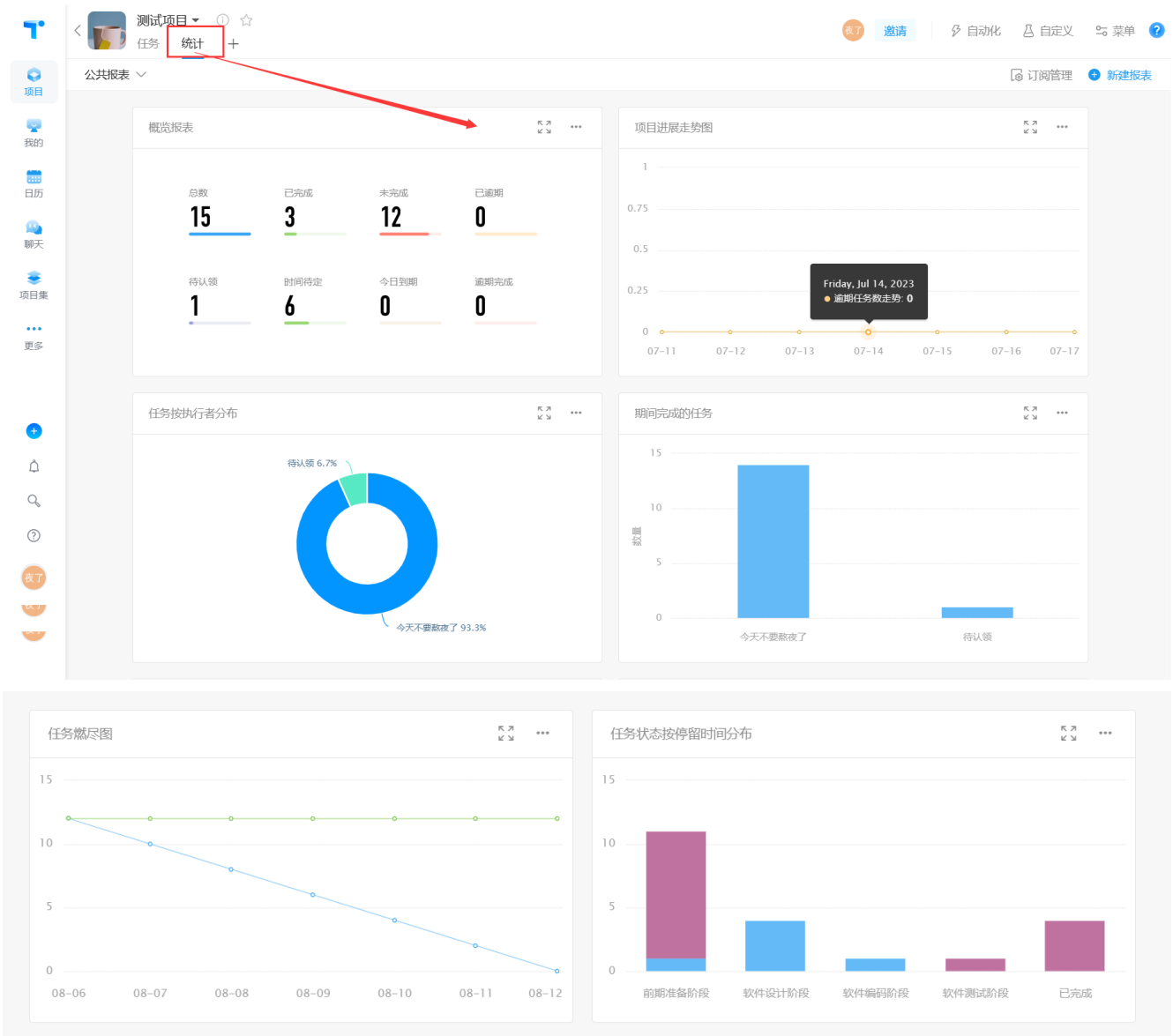


6. 创建好任务后点击任务进一步进行子任务拆分、添加任务资源、还可以将任务相关参与者邀请进此任务，进行信息同步。

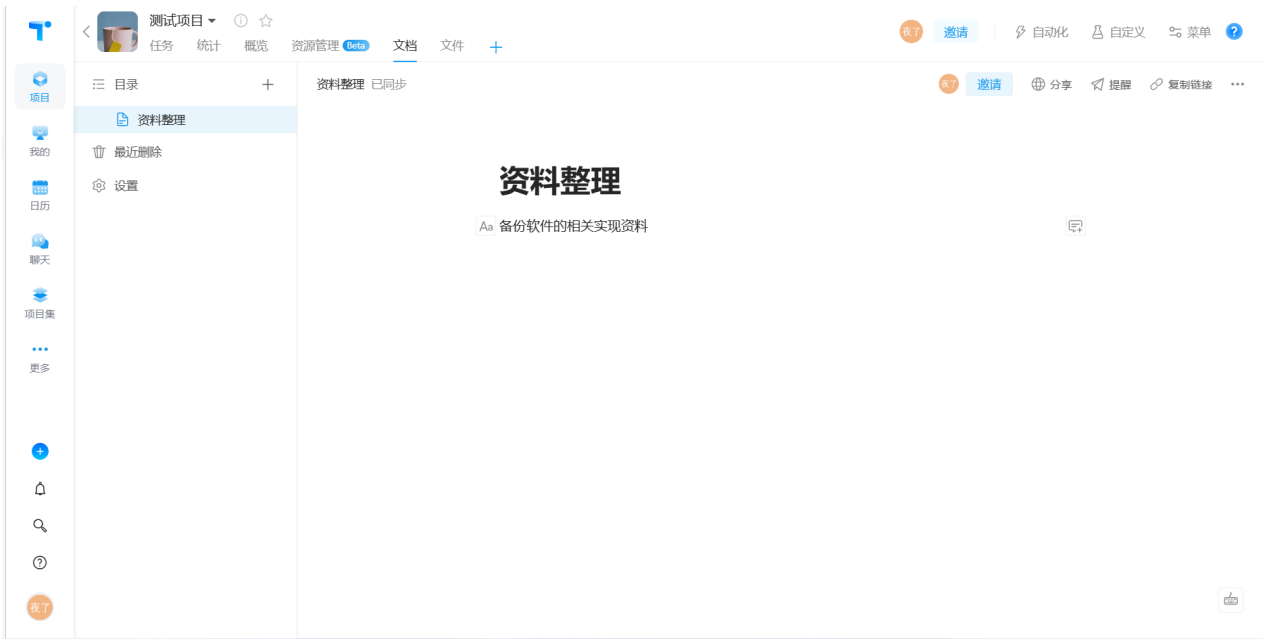
7. 可以在表格视图方便的批量查看、清点、编辑任务

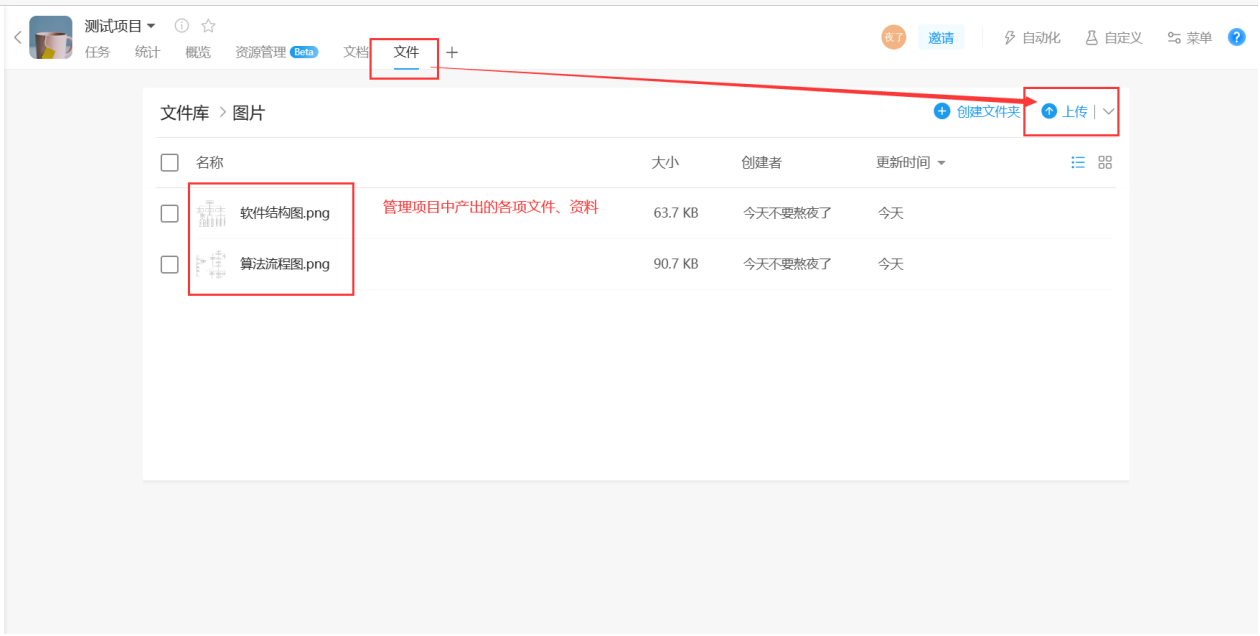


8. 在统计栏查看项目的完成进度，在实际开发中可以实时查看，根据燃尽图等即使发现任务问题和机会点。

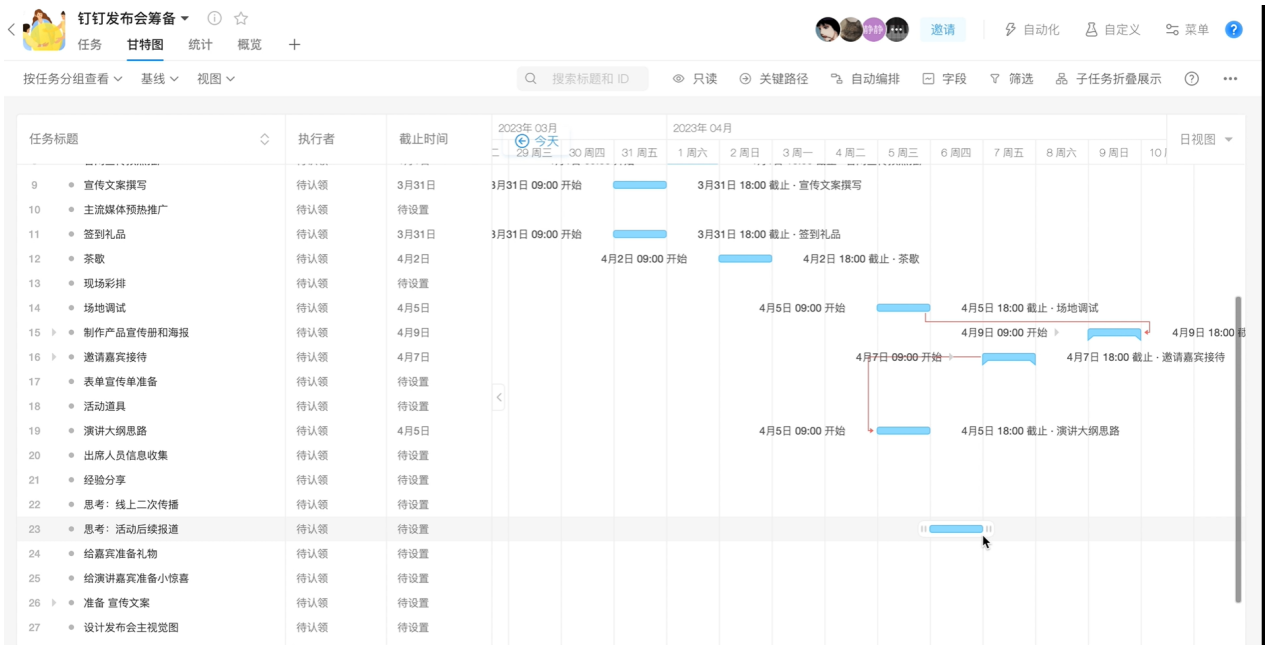


9. 在工具栏添加其他实用的应用，如文档、文件帮助管理任务中产生的文档图片，也可添加甘特图（免费试用14天）来帮助协调任务关系，更好的布置任务。
    - 添加文档、文件，管理项目中的文档、资料



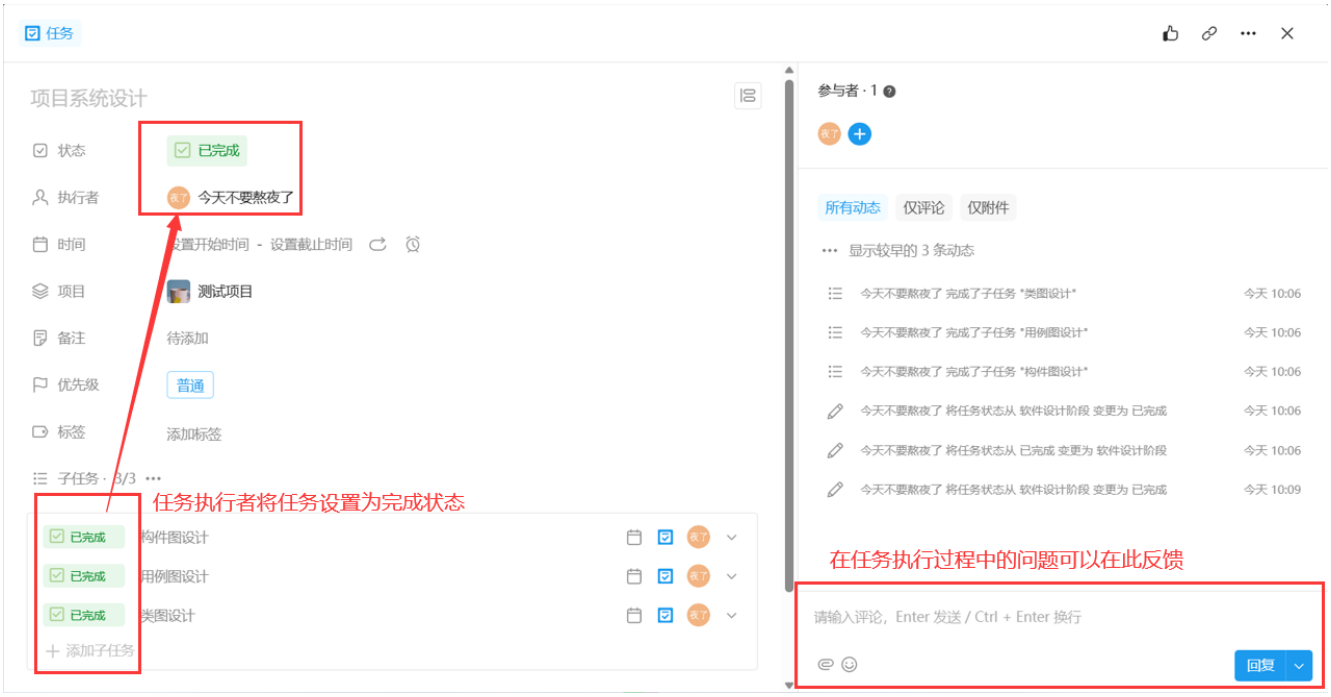


- 添加甘特图，查看任务与时间的顺序，通过拖拉操作对任务关系进行协调



四、成员任务完成与反馈

1. 任务执行者直接点击项目栏中的任务，在任务评论区反馈执行任务时的情况。
2. 在任务完成，将对应任务设置为完成状态。



至此，你已经了解了如何使用Teambition进行基本的团队项目管理。

2.2 Git代码管理

2.2.1 实验目的

了解并掌握git的基本操作，学会如何使用git对代码进行版本控制和分支管理。

2.2.2 实验要求与内容

要求

- 在Ubuntu上配置git。
- 掌握如何使用git进行版本控制。
- 学会如何使用git进行分支管理。

内容

一、在Ubuntu上安装git

1. 使用`git --version`查看Ubuntu上是否已经安装git，若正确输出git版本信息，则跳过此部分。
2. 安装Git。

```
sudo apt update      # 更新软件包列表，检查可用的软件包更新
sudo apt install git # 安装git软件包

git --version        # 验证是否安装成功
```

3. 配置git的用户名和邮箱。

```
git config --global user.name "zhangsan"              # 设置用户名，根据自己实际替换姓名
git config --global user.email "zhangsan@163.com"     #设置邮箱，根据自己实际替换邮箱
git config -l                                         #设置完成后,检查信息
```

二、使用git进行版本控制

1. 创建一个git仓库。

```
mkdir ~/git_learn    # 在~下创建一个git_learn文件夹
cd ~/git_learn       # 进入code文件夹
git init             # 初始化git仓库
```

2. 创建文件并提交第一个代码版本。

```
echo "code" > code.cpp              # 创建code.cpp 并写入
git add .                           # 将当前目录下所有更改存入暂存区
git commit -m "first version"       # 将暂存区的修改提交至本地仓库，提交信息为first
version
```

3. 修改文件并提交第二个版本。

```
echo "modified code" > code.cpp               # 修改code.cpp内容
git add .; git commit -m "modified version"    # 提交第二版代码
git log                                        # 可以使用此命令查看历史提交信息，按“q”键退出
```

4. 查看上一版内容。

```
cat code.cpp          # 查看文件内容，为modified code
git checkout HEAD^    # 返回到当前版本的上一版本。HEAD 是一个特殊的指针，指向当前所在的分支或具
体的提交。
cat code.cpp          # 发现文件内容恢复到上一版本，为code
git checkout master   # 回到最后提交的状态
cat code.cpp          # 发现文件返回最后提交的状态，为modified code
```

在恢复文件内容上，注意`git checkout`与随后会讲到的`git reset`的区别：

- `git checkout HEAD^`用于返回到当前版本的上一版本，但并不会改变我们的提交历史。即我们使用`git checkout HEAD^`回到第一版，但是我们的第二版提交记录依然保存着，我们可以使用`git checkout master`返回到第二版状态。

- `git reset` 也可以返回到之前的节点，但是我们的提交历史不会保留。如果我们将此处的checkout替换未reset，我们就不能使用`git checkout master`返回到第二次提交的状态，因为master也被重置到第一版。这意味着我们将第二版的提交信息完全丢弃了。但是第二版与第一版之间的差异（reset之前第二版提交的内容）是否被重置，和reset的参数选项有关，下面会详细介绍。

5. 丢弃暂存区代码

```
echo "some mistake" > code.cpp      # 我们写了一些不正确的代码
git add .         # 将错误的代码提交到了暂存区
git reset --hard     # 丢弃暂存的内容，将工作区内文件恢复到最近一次commit的状态

cat code.cpp     # 发现文件返回到第二版的状态，为modified code。第二版以后未提交的修改（包括add的修改）均被重置。
```

> 注: 此处如果我们使用的是`git reset HEAD^ --hard`，则会将code.cpp内容恢复到第一版（即“code”），此小节的更改（即“some mistake”）和第二版的提交记录（即“modified code”）都将被丢弃，且无法恢复。

6. 处理错误的commit

```
# 我们继续进行第三次提交,这次我们要写另一个文件
echo "third code" > code_2.cpp
git add .; git commit -m "third version"

# 准备继续写第四版
echo "four code" > code_2.cpp

# 此时我们发现提交的"modified version"有错误，需要修改
# 做之前我们可以先查看当前的状态
git status       # 查看一下当前工作区状态

# 信息
On branch master                 # 在master分支
Changes not staged for commit:  # 存在没有暂存的更改
    (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
            modified:   code_2.cpp # 被修改的文件
no changes added to commit (use "git add" and/or "git commit -a")
# 上面为git status的信息

git log          # 查看一下我们的提交历史，应该有三次提交
```

查看完状态，我们需要进行修改。由于"modified version"有错误，我们需要回退到错误之前，即第一次提交完成的时候。

但是，目前我们已经准备要做第四版提交了，使用`git reset --hard`代价过高。怎么办呢？使用“--soft”选项！

```
git reset HEAD~2 --soft # 将提交历史恢复到第一版的时候
```

```
git status                # 查看工作区状态

# 可以看到我们第二次第三次做的更改还存在，只是处于暂存状态，第四次的更改没有使用add添加到暂存区，也依然存在
# 如果我们再使用ls以及cat命令查看文件，会发现文件内容没有变化

git log                   # 查看提交信息，发现只有第一版的提交信息了

# 修改第二版的错误，将code.cpp 内容"modified code"改为"second code"
echo "second code" > code.cpp
# 提交修改，我们可能只想把之前的更改都提交，而保留第四次的更改不提交，因为第四版可能还处于未完成状态
git add code.cpp; git commit -m "second version"

git status  # 此时"four code"依然未提交，其余更改都提交了
git log     # 可以看到只有"first version"和"second version"两次提交记录
```

补充：`git reset`默认为`--mixed`模式，此模式与`--soft`的区别:

- `git reset --soft`保留工作区代码不变，会将之前的更改自动加入暂存区，我们使用`git commit`就可以将代码状态恢复到reset之前。
- `git reset --mixed`同样保留工作区代码不变，会将之前的更改重置为未暂存的状态，此时直接使用`git commit`并不能将之前的更改提交。我们需要手动使用`git add`将需要的文件添加到暂存区，随后再进行提交。

```
git reset --hard # 丢弃 four code 的修改
git status       # 查看工作区状态
git log          # 可以看到只有"first version"和"second version"两次提交记录
```

7. 使用.gitignore管理不需要进行版本控制的文件。

```
mkdir trash               # 创建文件夹，假定里面放入的是无关的文件
touch cache               # 创建文件，也是无关的文件

# .gitignore 里记录的文件名不会被git提交和保存

echo trash > .gitignore                 # 将trash写入.gitignore
echo cache >> .gitignore                # 将cache写入.gitignore
git add .;git commit -m "update .gitignore" # 完成一次提交
```

三、使用git进行分支管理

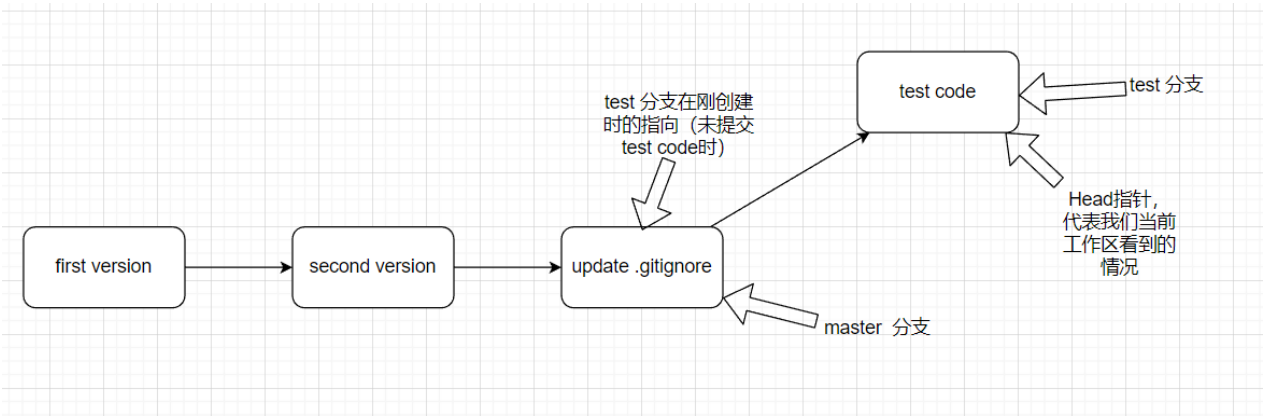
> 说明：如果对一个软件进行多版本开发（如：stable分支、dev分支），此时需要用到分支管理功能，此部分的操作建立在 **第二节（使用git进行版本控制）** 的基础上

1. 创建并切换至test分支，在test分支完成一次提交。

```
git branch test                       # 在当前节点创建branch分支
git switch test                       # 切换至test分支
echo "test code" > code.cpp           # 在test分支向code.cpp写入内容
git add .;git commit -m "test code" # 提交
```

在创建并提交这个分支后，我们需要详细了解一下git的HEAD指针这个概念：

- git的提交历史节点组成了一个树形的结构，不同的分支则是树的不同分支。
- HEAD指针则指向了我们当前所处在的节点，我们通过`git switch test`，将HEAD指针从master切换到了test，并提交了一个节点。
- 具体说明看下图：

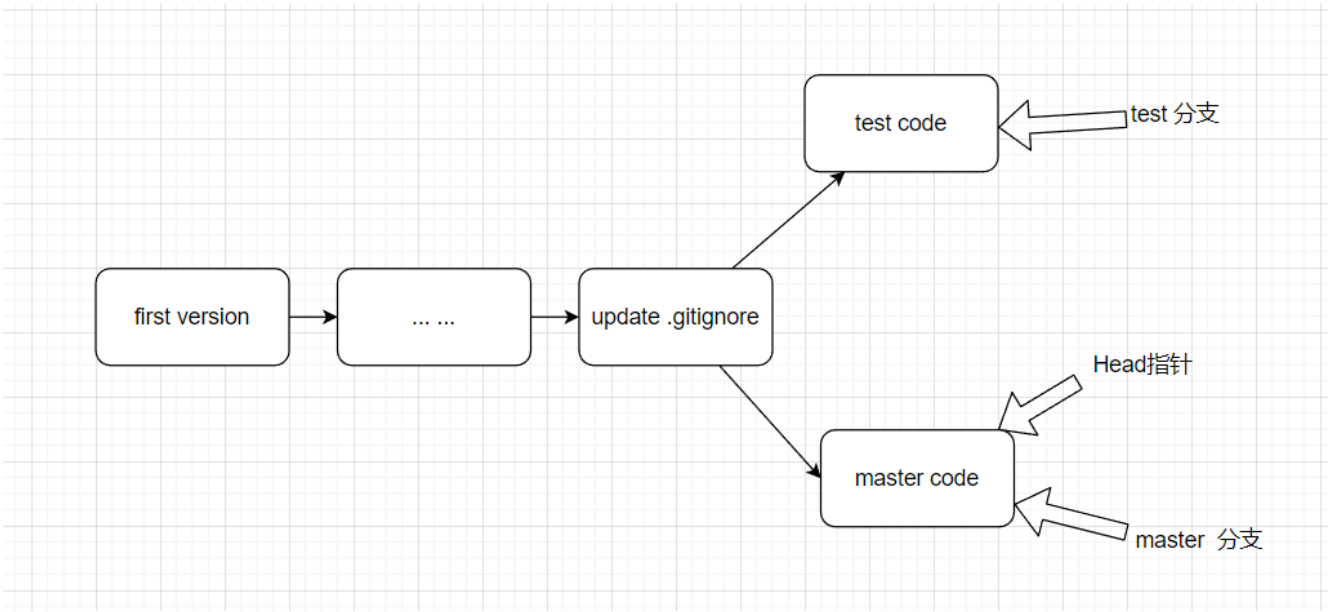


2. 切换回master分支，并完成一次提交。

   需要明确的是，master和test两个分支提交的内容互不干扰。

```
git switch master     # 将head指针切换至master分支
echo "master code" > code.cpp
git add .;git commit -m "master code" # 提交
```

此时，分支情况如下图所示：



3. 保存临时代码。

此操作主要用于：当你在一个分支（master），写了一些未完善（未提交的代码），但需要切换到另一个分支（test）进行另外的编写时，暂存你的工作。如下例子所示：

```
echo "new code (uncompleted)" > code.cpp      # 这代表你未完成的工作

# 此时来了紧急任务，需要我们在test提交代码
git checkout test   # 尝试直接切换到test分支

# 下面是git的错误报告，告诉我们修改的文件和test分支冲突，我们需要将修改完成一次提交或存入stash区
error: Your local changes to the following files would be overwritten by checkout:
    code.cpp
Please commit your changes or stash them before you switch branches.
Aborting
# 上面是错误报告

# 由于新代码未完成，不宜进行提交，所以我们将其存入stash
git add .           # 将未完成的修改加入暂存区
git stash -m "new code (uncompleted)"          # 将暂存区的代码存入stash，可以额外添加 -m 选项用于写一些此暂存内容的描述
git stash list      # 此命令用于查看暂存区的内容

git checkout test   # 切换至test分支
echo "complete urgent Tasks" > code.cpp
git add . ; git commit -m "complete urgent Tasks"  # 完成分派的紧急任务

# 完成任务后，我们需要继续切换回master分支，继续完成我们的代码
git checkout master
git stash pop   # 将stash里最近一次保存复原到工作区
git stash list  # 复原后可再次查看stash区，发现已经空了

cat code.cpp    # 将显示之前暂存的内容，为“new code (uncompleted)”
echo "new code (completed)" > code.cpp
git add .; git commit -m "new code"     # 模拟完成了功能
```

注：此部分，我们使用`git checkout`完成了分支切换，此前使用的是`git switch`，两者的区别有以下几点：

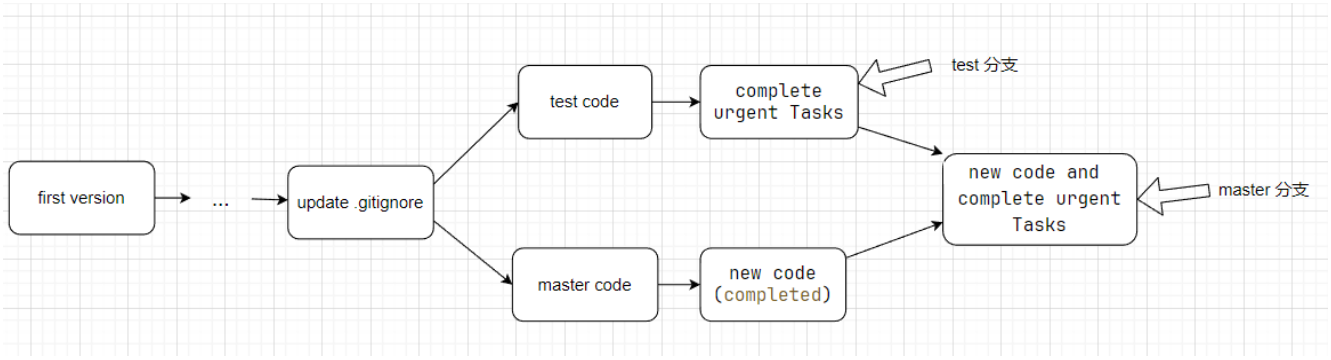
- `git checkout`可以用于分支间切换，也可以用于切换head指针到git提交历史中的任意一个节点，如`git checkout <节点哈希值(可有git log查看)>`；而`git switch`只允许分支间的切换。
- `git switch <分支>`与`git checkout <分支>`没有区别，均为切换分支；`git swich -c <分支>`与`git checkout -b <分支>`没有区别，均为创建并切换至新分支。
- `git checkout`功能比`git swich`更多，是因为`git switch`本身就是为了解决`git checkout`功能繁杂的问题；可以将`git switch`看作`git checkout`切换分支时的特化版本，在切换分支时，应尽量使用`git switch`。

4. 分支合并。

   假定我们在test和master分支完成了两个功能，现在需要将两个分支汇总为一个分支，合并我们代码里的功能。

```
git switch master
git merge test   # 将test合并到我们当前所在分支(master)

# 此时由于我们两个分支的内容有冲突的部分，git会提示我们处理冲突

cat code.cpp     # 查看冲突文件，git帮我们把冲突部分列出来了
<<<<<<< HEAD
new code (completed)
=======
complete urgent Tasks
>>>>>>> test

# 打开code.cpp，修改文件到我们期望的样子，如"new code and complete urgent Tasks"或其他形式，应根据实际需要处理冲突代码

echo "new code and complete urgent Tasks" > code.cpp   # 此语句相当于打开文件处理了冲突代码

git add . ; git commit -m "new code and complete urgent Tasks"  # 提交解决冲突后的文件，自动完成分支合并
```

此时，git的节点树形图如下图所示:



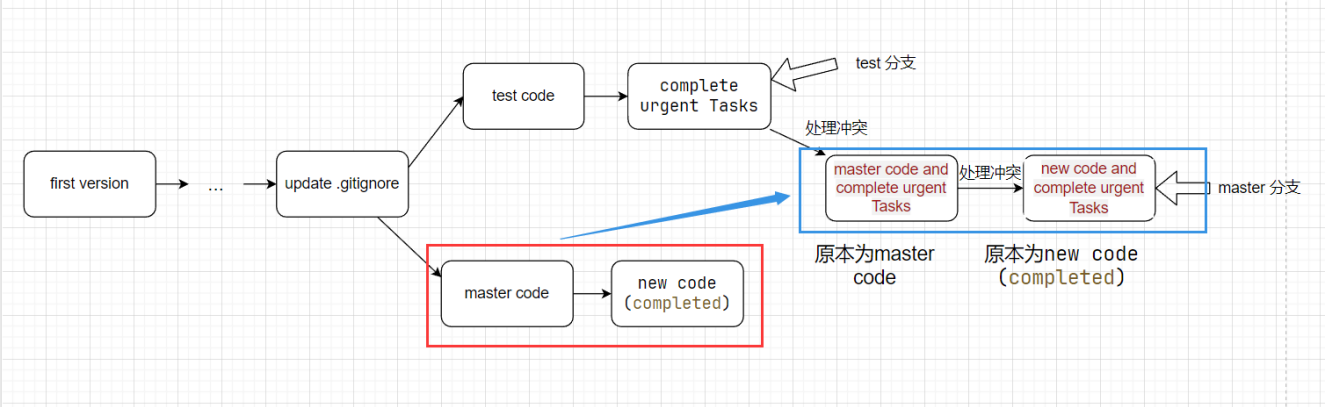
5. 分支变基。

   合并两个分支，也可以使用`git rebase <要合并的分支>`，下面假定你没有做上一步**4. 分支合并**:

```
git switch master
git rebase test          # 使用rebase合并分支
# git rebase会将master分支上的每个提交都在test分支上提交一次，再将test分支变为master分支，即“变基”。
# 此时由于我们两个分支上的各个提交有冲突的部分，git会提示我们处理冲突。
Auto-merging code.cpp
CONFLICT (content): Merge conflict in code.cpp
error: could not apply 761e3a7... master code
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
```

```
Could not apply 761e3a7... master code

# git会依次逐个提示我们变基过程中的每个冲突（本例中，理论上最多的冲突次数为master分支上的提交次数）。

cat code.cpp            # 查看冲突
<<<<<<< HEAD
complete urgent Tasks
=======
master code
>>>>>>> ae4db29 (master code)

echo "master code and complete urgent Tasks" > code.cpp                # 解决第一次冲突
git add .
git rebase --continue   # 继续rebase
# 第二次冲突。（master分支上的每个提交都会在test分支上提交一次。）
[detached HEAD c86da6d] master code and complete urgent Tasks
 1 file changed, 1 insertion(+), 1 deletion(-)
Auto-merging code.cpp
CONFLICT (content): Merge conflict in code.cpp
error: could not apply 99ac134... new code
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 99ac134... new code

cat code.cpp
<<<<<<< HEAD
master code and complete urgent Tasks
=======
new code (completed)
>>>>>>> 99ac134 (new code)

echo "new code and complete urgent Tasks" > code.cpp                # 解决第二次冲突
git add .
git rebase --continue   # 继续rebase
# rebase成功。
[detached HEAD 136ecb6] new code and complete urgent Tasks
 1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/master.

git log                 # 查看提交历史
```

变基将红框的部分搬运到test分支之后，依次进行提交，我们需要依次处理冲突，最终得到变基后的结果。具体原理见下图：



至此，你已经了解了如何使用Git进行代码管理。附:

- Git 官网：https://git-scm.com/
- Git - Book Git官方文档：https://git-scm.com/book/zh/v2
- Oh My Git! 一个可视化Git学习游戏：https://ohmygit.org/

2.3 Github工作流

2.3.1 实验目的

学会如何使用Github进行团队协作开发。

2.3.2 实验要求与内容

要求

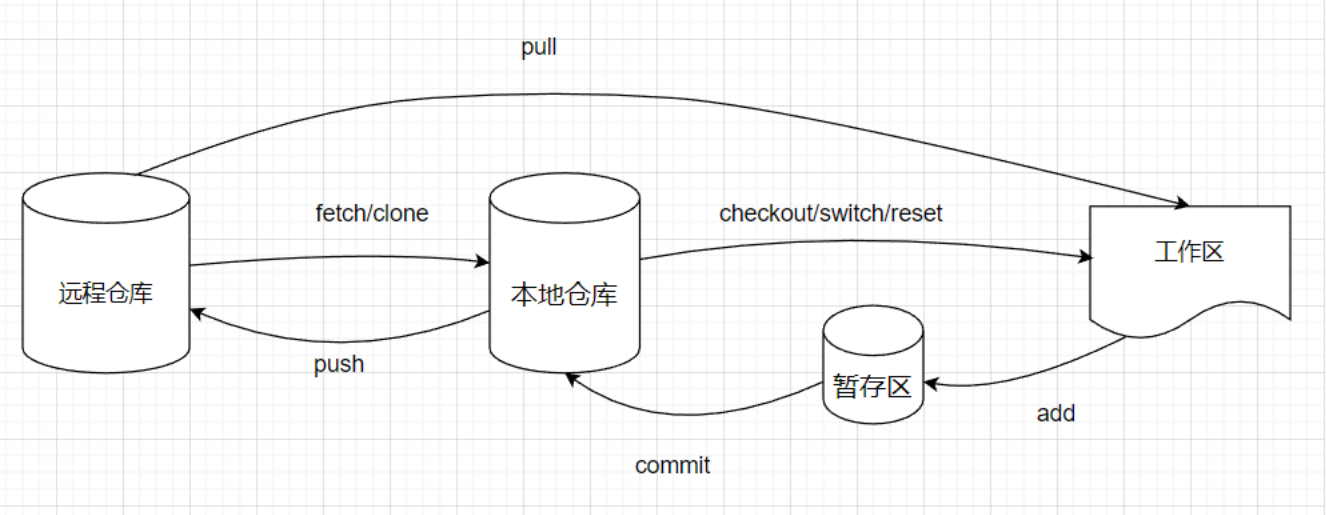
- 了解git和github的工作原理。
- 在Ubuntu下配置好ssh。
- 使用github仓库存储项目代码。
- 学会使用github进行协作开发。

内容

一、git与github的工作原理

> 此部分为了解性知识，无需手动操作，但是此部分的说明对理解git以及github的工作原理有很大帮助。

1. git的原理图如下：



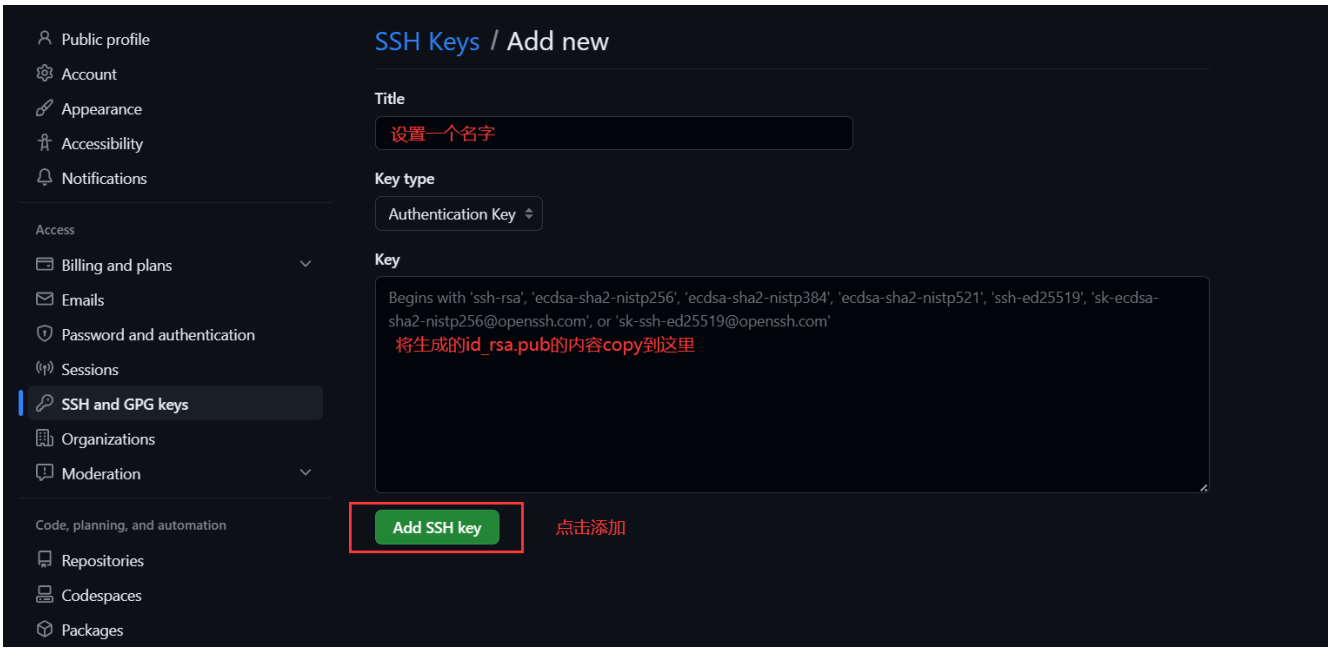
2. 工作区即我们可以在项目文件夹看到的内容，暂存区和本地仓库保存至工作区的.git文件夹内，github等代码托管平台则是远程仓库。
3. 我们可以使用`git push`、`git pull`、`git fetch`等命令将本地仓库的内容推送到github或gitlab这样的远程仓库，或将内容从远程仓库拉取到本地，以便进行团队开发。

二、在Ubuntu下配置ssh

1. 安装ssh，Ubuntu默认安装好了openssh，输入`ssh`检查是否安装好，若未安装，请使用`sudo apt install openssh-server`安装ssh服务。
2. 使用`ssh-kengen -C "zhangsan@163.com"`生成ssh key，将其中的邮箱替换为自己的邮箱，执行命令直接全部回车确认即可。执行此命令将在`~/.ssh`目录下生成`id_rsa`、`id_rsa.pub`，请保护好自己的`id_rsa`私钥。
3. 使用`chmod 400 id_rsa id_rsa.pub`设置ssh key文件的权限为仅自己可读。

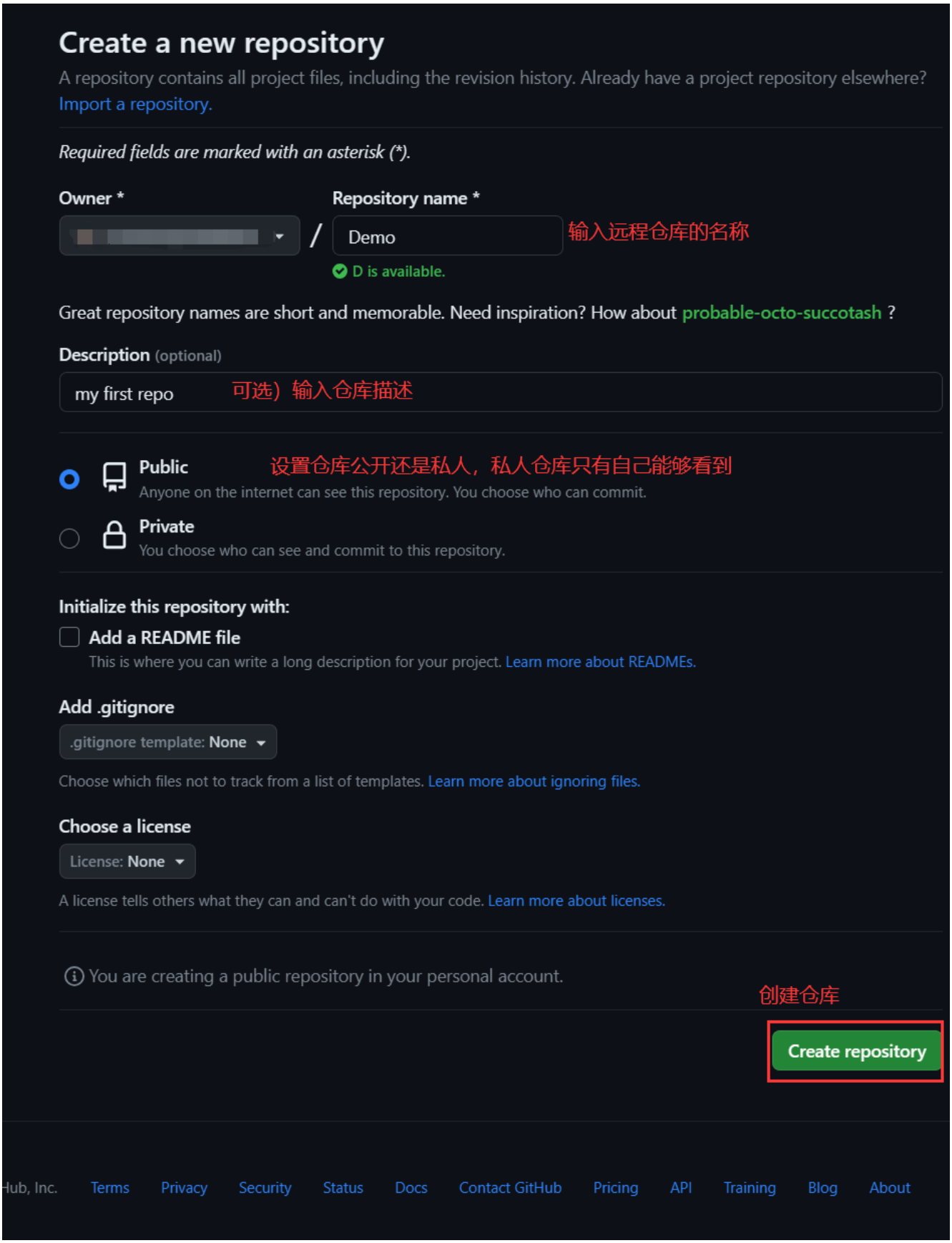
三、创建github仓库

1. 在github 官网创建账号：https://github.com
2. 在已登录的github界面，点击用户头像->"settings"->"SSH and GPG keys"
3. 点击"New SSH key"按钮，设置Title，然后将`~/.ssh/id_rsa.pub`的内容复制到key文本框中,保存设置

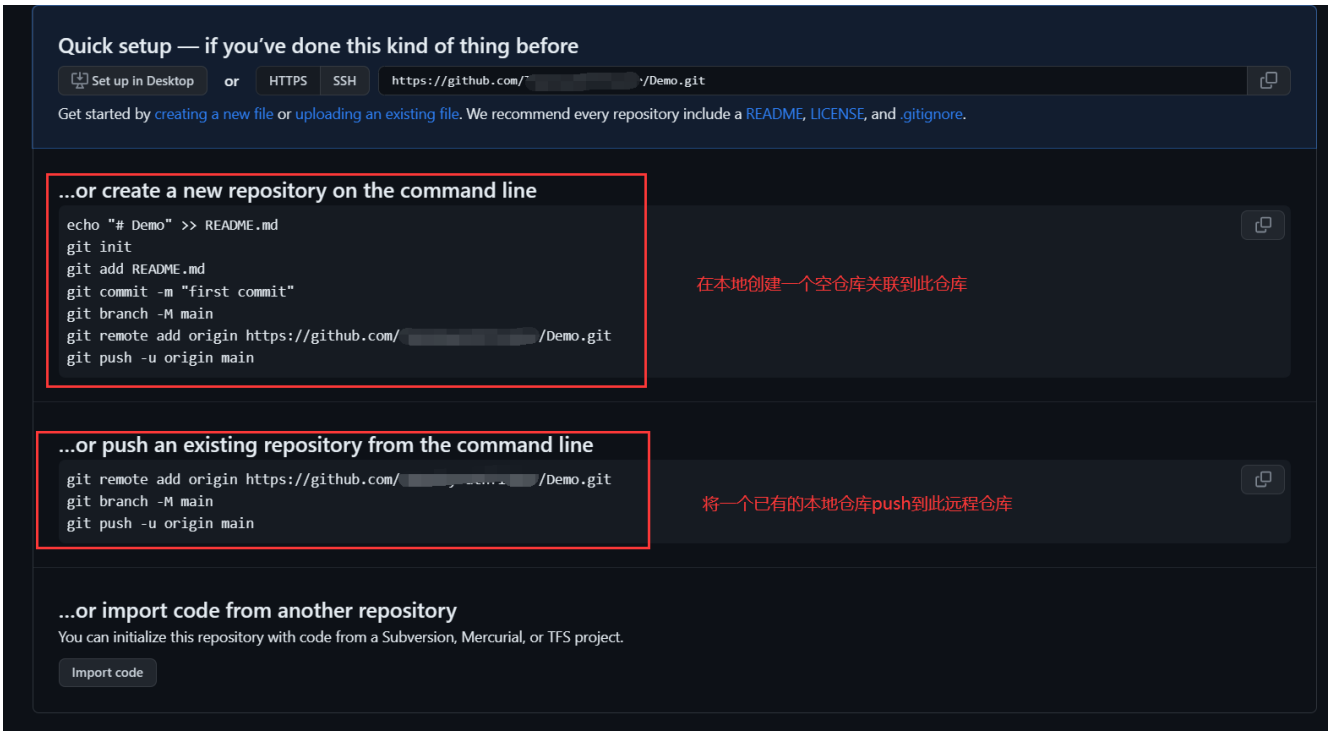


四、创建第一个仓库

1. 在已登录的github界面，点击用户头像->"Your repositories"，随后点击New，新建一个仓库



2. 将本地仓库与线上仓库关联，选择第二种方式将第二部分中Git创建的项目"git_learn"推送到远程仓库。



至此，git_learn成功上传至github，可以利用github进行多人项目协作。

五、trunk工作流

> trunk工作流是github下最简单的多人协作模式，此模式只存在一个主干分支，其他人都基于主干分支进行代码提交、修改。此部分需要我们使用两个文件夹，来模拟两个人的提交，亦可与朋友进行互动实验。

1. 创建一个空仓库，参考 **第四节（创建一个仓库）** 。
2. 在本地合适的路径下创建两个文件夹模拟两个团队成员

```
mkdir ~/github_test
cd ~/github_test          # 切换到合适的文件路径下

mkdir A_demo B_demo       # 创建A，B两个成员你的工作区
cd ~/github_test/A_demo     # 进入A的工作区
git clone <远程仓库链接> .    # clone 远程仓库

cd ~/github_test/B_demo     # 进入B的工作区
git clone <远程仓库链接> .    # clone 远程仓库
```

3. 模拟A进行一些代码的提交

```
cd ~/github_test/A_demo    # 进入A的工作区

# A编写自己的任务代码
echo A_fun_1 > code_1.cpp
echo A_fun_2 > code_2.cpp

git add .; git commit -m "A f_1 f_2" # 提交到本地仓库
```

```
# 提交到本地仓库后，A需要把自己的代码推送到远程仓库，以便其他人看到A的代码、在A提交的代码基础上继续改进

# 在push之前使用pull及时与远程仓库同步是一个好习惯
git pull origin main      # 由于A最先完成他的工作，此时远程仓库没有别人的提交，因此不存在远程仓库和本地仓库的冲突

git push    # 将本地的代码推送到远程仓库
```

4. 模拟B进行一些代码的提交

```
cd ~/github_test/B_demo # 进入B的工作区
# B编写自己的任务代码
echo B_fun_2 > code_2.cpp
echo B_fun_3 > code_3.cpp

git add .; git commit -m "B f_2 f_3" # 提交到本地仓库

# B也需要将自己的代码提交到远程。为确保自己是的修改是基于最新版本的，B在push之前最好先做pull操作，将自己的代码版本更新到最新。
git config pull.rebase true   # 首先设置pull时处理的方式，trunk工作流建议使用rebase模式

git pull origin main        # B提交的时候，远程仓库已经被A更改了，因此pull时会提示冲突
cat code_2.cpp             # 查看冲突的文件

<<<<<<< HEAD
A_fun_2
=======
B_fun_2
>>>>>>> 01f9d76 (B f_2 f_3)

echo A_B_fun_2 > code_2.cpp          # 处理冲突
git add . ; git rebase --continue   # 完成rebase
git log     # 可以看到A和B分别进行了两次提交
git push    # 将本地的代码推送到远程仓库
```

5. `git fetch`与`git pull`

> 在**4.模拟B进行一些代码的提交**部分，我们已经了解了在trunk工作流如何进行简单的多人协作开发，以及遇到冲突该如何处理。此部分为了解性知识，介绍fetch与pull的区别。

- 参照**二、git与github的工作原理**中的原理图，可以了解到`git fetch`只是将远程仓库的代码同步到本地仓库，它并不会影响工作区的代码。
- 只看图或许难以理解`git pull`，`git pull`做了两件事：首先，它将远程仓库与本地仓库进行同步，这点与`git fetch`相同；其次，它在同步完成后立即将远程仓库的最新节点和本地工作区的节点进行了合并提交。合并的方式与设置相关，本例中选择的时rebase。

- `git fetch`一种可能的应用场景是，我们在开发期间能要进行分支的切换：需要先`git fetch`将远程的最新数据同步到本地；再使用`git stash`保存本地已经完成的代码；随后使用`git switch`切换到另一个分支查看或修改
- `git pull`则主要用于某些镜像仓库保持与远程一致，或在提交功能前pull远程代码进行冲突合并

6. 标签的使用

> 在项目开发过程中，我们可能有某些重要版本的代码，或者其余有标志意义的commit需要标记，以便我们能随时切换到这个版本查看相关信息。

```
# 假设B完成他的提交后，整个项目就完成到了1.0版本，可以打上tag标记
cd ~/github_test/B_demo # 进入B的工作区
git tag v1.0     # 在B最近一次的提交上标记
git show v1.0    # 查看指定tag信息

git tag -l       # 列出所有tag
git checkout <tag name> # 查看指定tag所在commit的内容，如果不标记tag，我们只能使用commit的hash值来指代要切换的节点，很不方便
# git tag -d <tag name>    # 删除指定的tag
# git tag -f v1.0 <指定节点hash或分支名称>  # 将v1.0这个tag移动到指定的节点
git push --tags # 推送所有tag到远程
```

六、功能分支工作流

1. 创建一个空仓库，参考 **第四节（创建一个仓库）** 。
2. 在本地空仓库进行一些提交，本例使用如下命令：

```
echo "1" >> demo.txt
git add demo.txt
git commit -m "1"
git push
```

3. 创建一个功能分支"feature_1"，在本地使用以下命令：

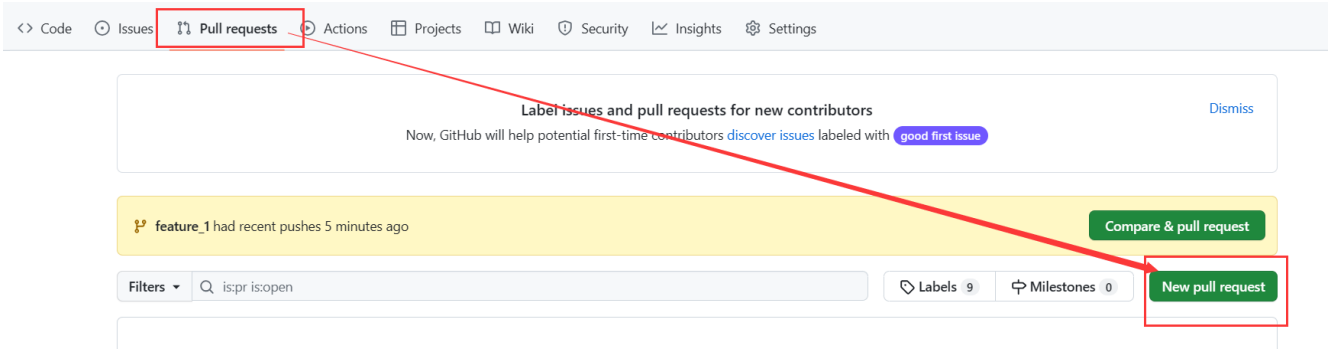
```
git pull # 在本地添加分支前，使用git pull保持你的本地master分支与远端同步
git checkout -b feature_1 main # 在main分支创建一个名为feature_1的分支
echo "feature_1 function">> demo.txt # 做一些更改，这就好比你在开发feature_1的功能
git add .;git commit -m "feature_1 function" # 进行修改的提交
git push # 你将这个分支也推送到远端仓库存储，以便和其他成员共同开发feature_1分支的功能
# 下面是使用git push后的报错，git提醒我们远端仓库不包含我们现在的分支
# 如果我们需要将此分支存放到远端仓库需要使用git push --set-upstream origin feature_1 命令
fatal: The current branch feature_1 has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature_1

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

```
# 上面是报错信息

# 将本地内容推送到远端，如果远端没有此分支则自动创建
git push --set-upstream origin feature_1
```

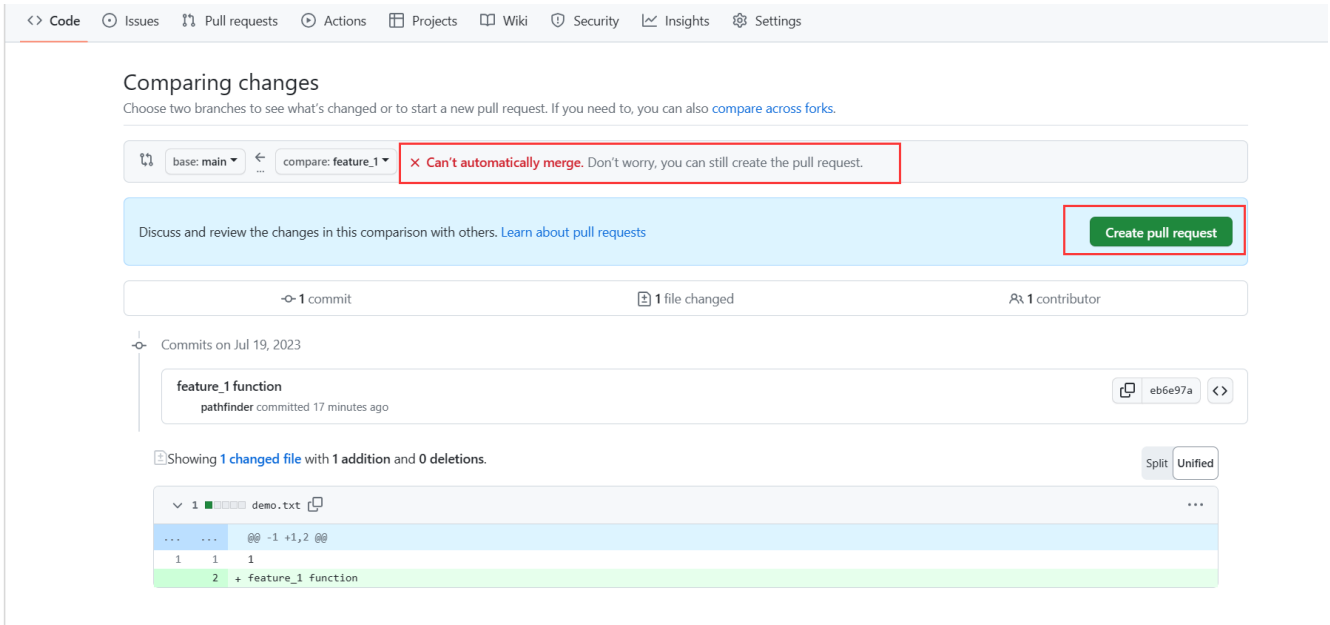
4. 模拟其他成员在我们完善feature_1时向主分支提交了新功能:

```
# 此代码模拟有人向main添加了修改的功能
git checkout main
echo "3" >> demo.txt
git add .; git commit -m "feature_3"
git push
```

5. 现在，需要将feature_1分支合并到main分支中。发起pull request：

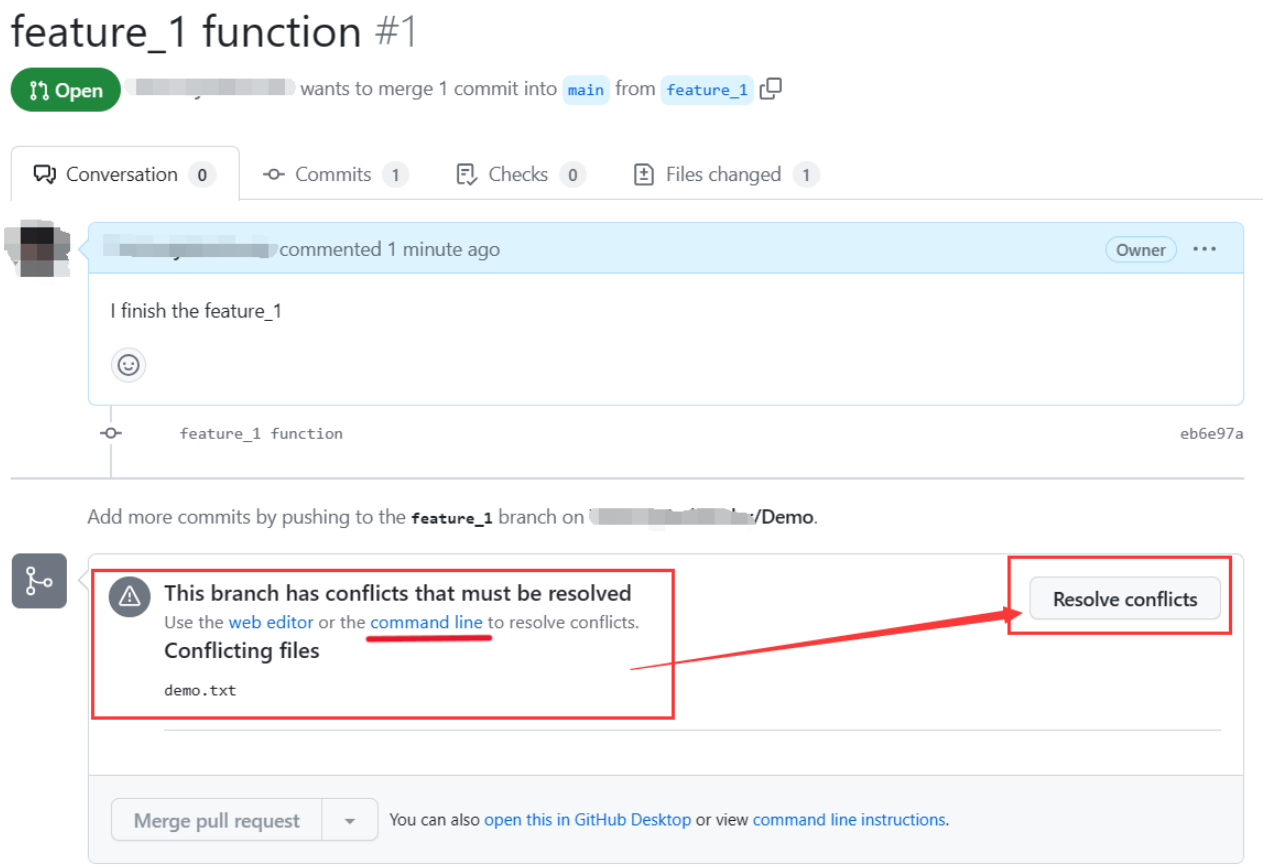


选择将feature_1分支合并到main分支中，此时，github提示我们要合并的代码版本有冲突，但并不影响pull request的创建。



创建pull request时，需要用户输入一些comment。

创建pull request后，github提示我们需要手动解决冲突，才能进行合并。



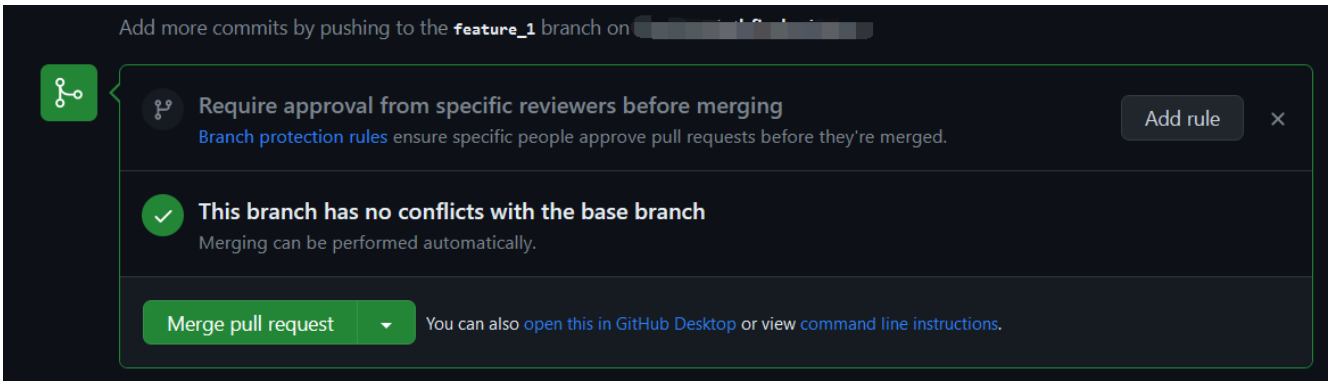
> 注意：对于带有冲突的pr，如需接受，git不可能自己解决这个冲突。此时，解决该冲突的人选有两个：**pr的提出者（如负责feature_1的项目成员）**或者**代码审查者（如项目组长）**。这两个角色都可以解决此类冲突。以下分别演示：

6. pr的提出者（如负责feature_1的项目成员）主动解决该冲突（这是常见方式，可以方便代码审查者直接合并该pr，减轻代码审查者负担）。对于简单的冲突，可以直接使用github提供的在线编辑器解决冲突；而对于复杂的冲突，则最好在本地命令行中进行解决，再推送到github中。参考如下命令：

```
git pull origin main     # 更新本地仓库与线上仓库的main分支一致
git checkout feature_1   # 切换到我们负责的分支
git merge main           # 在本地与main分支进行合并

# 提示冲突
Auto-merging demo.txt
CONFLICT (content): Merge conflict in demo.txt
Automatic merge failed; fix conflicts and then commit the result.
# 上面为冲突信息

# 查看冲突
cat demo.txt
1
<<<<<<< HEAD
feature_1 function
=======
3
>>>>>>> main
```

```
echo "3 and feature_1 function" > demo.txt  # 处理冲突
git add .; git commit                       # 本地提交
git push -u origin feature_1                # 推送到远端
```

此时，我们再查看github的pr状态，提示已经可以自动合并：



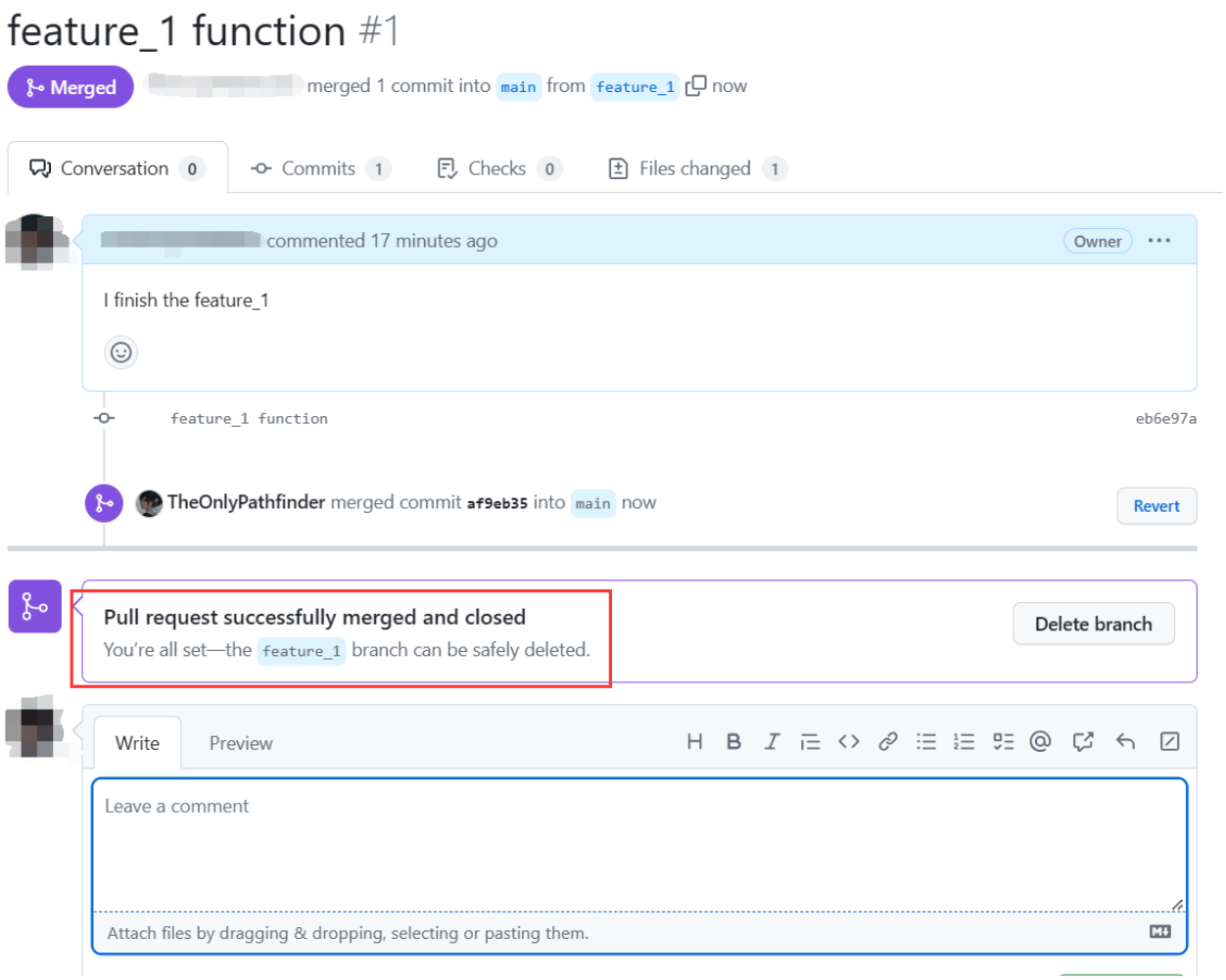
之后，代码审查者（如项目组长）点击"Merge pull request"即可完成合并。

7. 代码审查者（如项目组长）主动解决该冲突（该方式不常见，会增加代码审查者负担）。如果代码审查者（比如你的组长）同意了你的pr，且他想亲自解决该冲突并进行代码合并，则他可以采取以下处理方式：

```
git checkout main
git pull
git pull origin feature_1

# 这里git提示我们要选择遇见冲突时的解决方式
From github.com:xxxx/Demo
* branch            feature_1  -> FETCH_HEAD
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.

nothing to commit, working tree clean
# 上面是提示信息

# 根据自己需求选择使用rebase或者merge，本例使用merge方式
git config --global pull.rebase false

# 重新拉取并合并分支
git pull origin feature_1
```

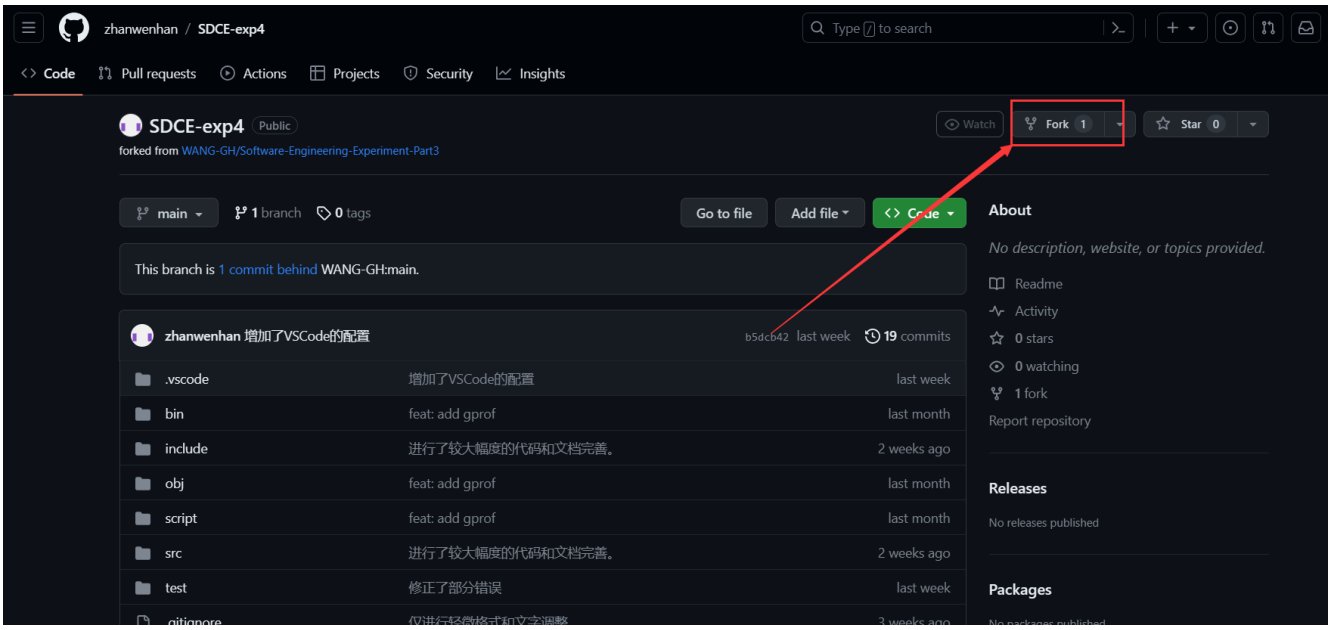
```
# 查看冲突的文件
git status
# 解决完冲突后提交
git add .; git commit

git push
```

git push成功之后，pr会自动关闭。

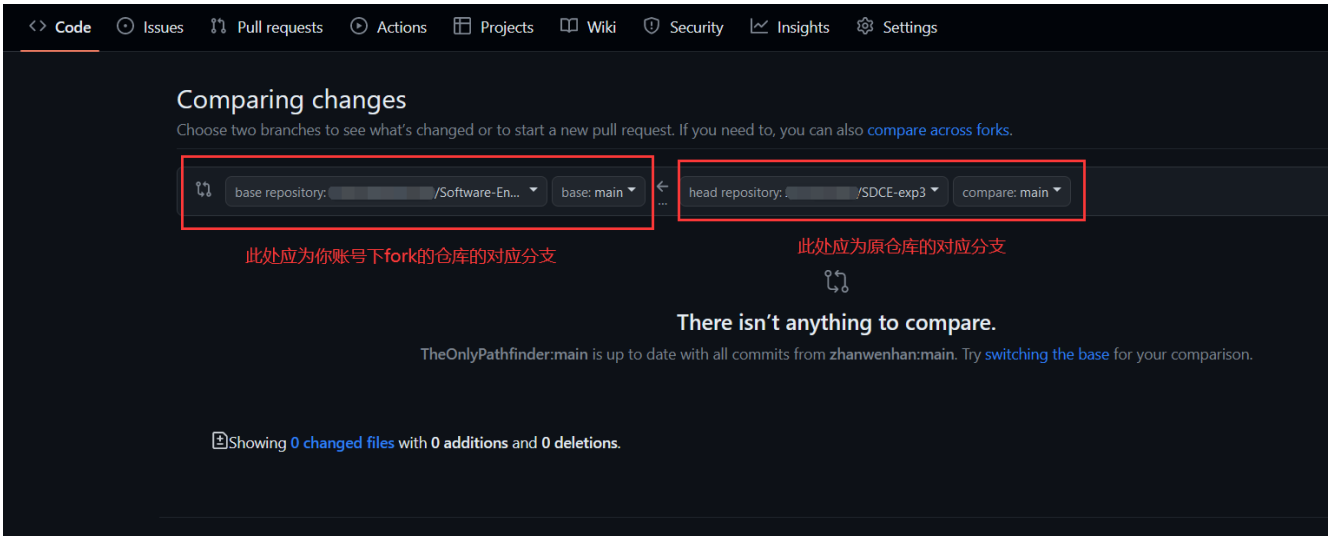


七、forking工作流

1. 进入你想要参与开发的项目仓库页面，点击fork按钮，将仓库fork到你自己的远程仓库。



2. 当需要同步远程仓库时，在自己fork的仓库发起pull request，需要注意将来源设置为原仓库，目的仓库为你账号下fork的仓库。



3. 在自己fork的仓库上完成相关的功能开发。
4. 发起pull request。
    - 此操作和 **六、功能分支工作流 小5**，以及**本部分小2**的内容基本相同，只是需要设置好pull request的来源仓库与目的仓库。
    - 之前**六、功能分支工作流 小5**是将仓库的`feature_1`合并到同一仓库下的`main`分支。
    - 在此处，则是需要设置你账号下fork的仓库为pull request的来源，主仓库为pull request目的仓库。
    - 发起pull request后需要按需解决冲突，参照**六、功能分支工作流 小6、小7**部分

至此，你已经了解如何使用Github进行协作开发。