

Git协作规范

creeper

UniGPT是大学学生的**课程作业**，在代码复杂度和开发者经验等方面有别于企业的大型商业项目，因此，利用Git进行多人协作和分支管理的一般工作流程与企业存在较大的差异。

本文档对UniGPT的Git多人协作进行了一定的规范。

分支和分支的命名

无论是前端还是后端，代码仓库主要包含以下几类分支：

- **主分支(master)**
 - 存储**稳定、可发布**的代码。
 - 所有其他的分支都从主分支**派生**。
 - 不接受直接提交，只允许其他分支合并。
- **功能分支(feature-XXX)**
 - 从主分支创建，用于实现某个特定的功能（特性）。
 - 命名以feature-开头，后接功能的简短描述，例如 `feature-login-page`。
- **修复分支(bug-XXX)**
 - 从主分支创建，用于修复特定的bug。
 - 命名以bug开头，后接bug对应的issue，例如 `bug-issue-19`。
- **热修复分支(hotfix-XXX)**
 - 从主分支创建，用于修复某个影响较为严重，需要快速修复的bug。
 - 命名以hotfix开头，后接修复的问题的简短描述，例如 `hotfix-security-issue`。

注：分支的名称全部使用小写，不同单词之间使用连字号 - (hyphen)连接。比

如 `feature-bot-chat-page` 是规范的，`feature-BotChatPage`，`feature_bot_chat_page` 等都是不规范的。

开发新的功能

feature分支是最常见的分支，用于实现新的功能。

开发前的准备工作

在开发新特性前，应该遵循以下的工作流。

在本地切换到主分支：

```
git checkout master
```

获取远程仓库的更新：

```
git fetch origin
```

更新本地的主分支与远程仓库同步：

```
git pull origin master
```

到目前为止，你本地的master分支已经与远端同步。现在，你可以创建一个feature分支并进入，开始新功能的开发：

```
git checkout -b feature-foo-bar-baz
```

总之，你需要保证三件事：

- 本地主分支与远端主分支同步。
- 新建一个feature分支进行开发，不要在本地主分支**直接开发**。
- 新的feature分支应该从**主分支**派生出来，不能从**别的分支**派生出来。

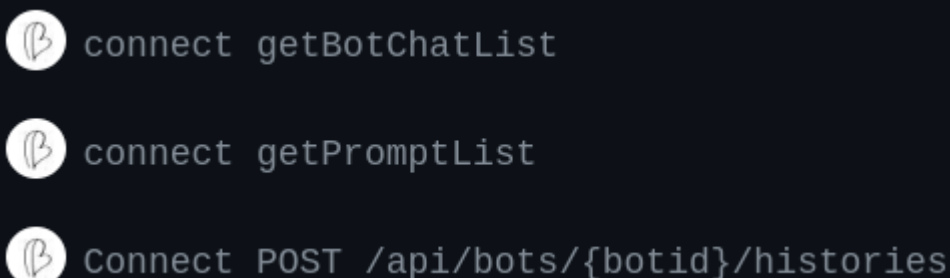
开发过程中的提交

对于一个功能分支来说，其实现的功能通常可以分成多个子功能。那么，开发者可以分步实现每个子功能，并将每个子功能对应到一个提交（commit），每个提交的信息**简单明了**地描述该提交的内容。

比如， feature-bot-chat-page 分支开发的功能是：将前端的**BotChatPage**页面与后端的api连接，后端api包括：

- GET /api/histories/{historyid}/chats
- GET /api/histories/{historyid}/promptlist
- POST /api/bots/{botid}/histories

那么，可以将每个api的连接作为一个子功能，对应到一个提交记录，像这样：



```
connect getBotChatList  
connect getPromptList  
connect POST /api/bots/{botid}/histories
```

与当前功能分支无关的修改

这是一个**常见**的问题，为方便描述，先看下面这个例子：

你正在开发**后端图片上传**的功能，位于分支 `feature-image-upload`，在打开前端调试时，你突然发现：

返回给前端的 `CommentDTO` 中的 `name` 字段，与前端识别的 `userName` 不一致，导致前端的评论区无法正常地显示评论者的名字。

于是，你直接在 `feature-image-upload` 分支上对后端的 `CommentDTO` 进行了改动：

```
@@ -19,7 +19,7 @@ public CommentDTO(Comment comment) {
    this.content = comment.getContent();
    this.time = comment.getTime();
    this.avatar = comment.getUser().getAvatar();
-   this.name = comment.getUser().getName();
+   this.userName = comment.getUser().getName();
    this.botId = comment.getBot().getId();
    this.userId = comment.getUser().getId();
}
```

接着，你继续进行图片存储功能的开发。

这个例子中，对 `CommentDTO` 字段的**改动**，和这个分支要完成的功能 `image-upload` 没有任何关系，这样做会导致严重的合作者之间仓库不同步的问题。

原因是，从你**创建** `image-upload` 分支，开发所有关于图片上传的分支，发起pull request，再到仓库管理者将你的feature分支合并到主分支，这段时间周期是**相对较长**的，那么这就意味着，在这段时间内，**你的组员**在测试时，评论区的评论者名字都**无法正常显示**。

这样做可能导致：

- 你的组员发现了这个bug，但是不知道你在 `feature-image-upload` 已经修复了，又修复了一遍bug，导致重复劳动。
- 你的图片上传功能存在问题，仓库管理员将你的pr打回了，同时将这个bug的修复一起**打回了**。

正确的做法应该是：

当你发现一个与当前开发功能无关的漏洞时，保存你当前开发分支的工作：

```
git commit -a --message='save all changes to fix commentDTO issue'
```

回到主分支：

```
git checkout master
```

创建修复漏洞的分支 hotfix-commentdto-name 并进入：

```
git checkout -b hotfix-commentdto-name
```

现在，你在原来主分支的基础上，对 CommentDTO 的字段问题进行解决。解决之后，**立即**将这个分支推送到远端，并发起pull request，这个pr的改动很小，所以在**短时间内**就能够被合并到主分支，让其他组员看到这个bug的修复。

在解决了这个漏洞后，你回到 feature-image-upload 分支，继续开发**图片上传**的功能。

功能开发完成之后

在新功能开发完成并测试没有问题后，向远端推送新功能的分支

```
git push origin feature-foo-bar-baz
```

在Github中发起**Pull Request**，等待仓库管理者的合并。

你已经完成这个功能的开发并推送到远端，所以你需要回到本地的主分支，并且**删除**本地的新功能分支。

发送Pull Request后在**微信通知群**发送一条消息，通知新的pull request。

```
git checkout master
git branch -D feature-foo-bar-baz
```

注意：开发新功能完成后，**严禁**不切换回主分支，在这个新功能分支上**派生出**其他分支进行开发，所有分支都应该从最新的主分支派生出来。

问题的发现和修复

确认新的问题

在开发和测试的过程中，难免会遇到一些问题和漏洞，在像组员报告这个问题之前，问自己三个问题：

- 当前分支在**主分支**吗？如果不在，切换到主分支。
- 本地的主分支和远端的主分支**同步**吗？如果没有，请同步。
- Github仓库页面的**issues**和**pull request**中，包含你遇到的这个bug吗？

在问完上面三个问题之后，如果你确认这是一个新的问题，可以告知你的组员。

判断问题的类型

问题分为**紧急**和**非紧急**两种。

- 紧急的问题包括编译无法通过、重大安全漏洞、即时的文档更新、文件误删 等需要快速修复并合并到**主分支**的问题。
- 非紧急的漏洞通常影响较小，大部分的bug都是非紧急问题。

解决紧急问题

解决紧急问题需要创建hotfix分支，在创建分支之前，你需要**保证**位于和**远端主分支**同步的**本地主分支**上。

确认之后，创建hotfix分支：

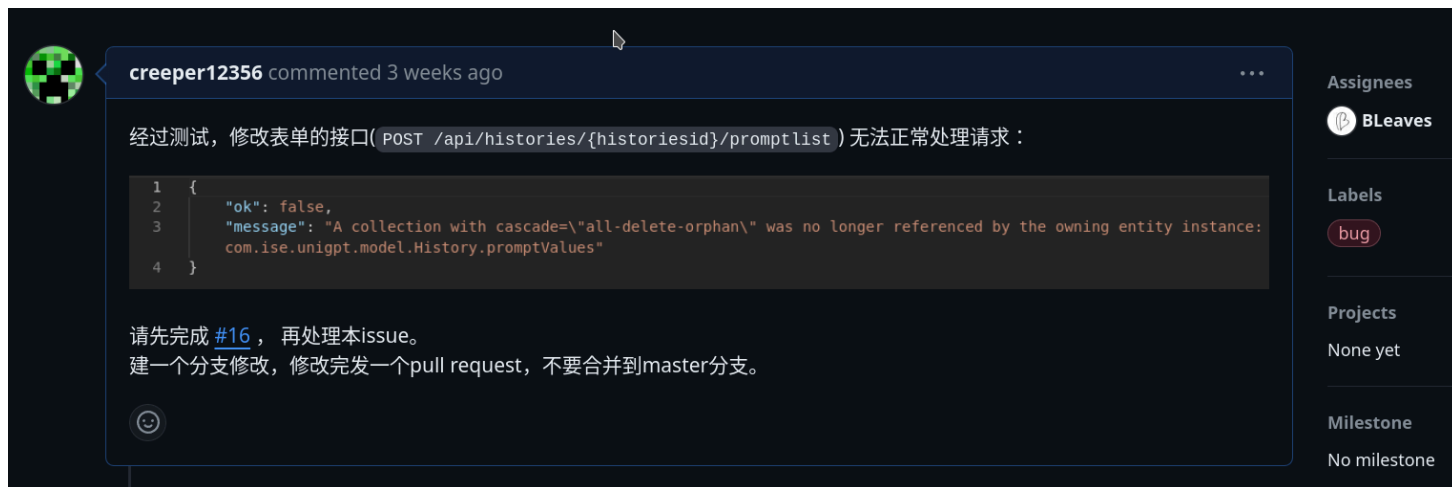
```
git checkout -b hotfix-foo-bar-baz
```

解决之后推送到远端，发送pull request，切回主分支，删除hotfix分支，与feature分支开发完成后的操作**相同**。

发送Pull Request后在**微信通知群**发送一条消息，通知新的pull request。

通知非紧急问题

在发现并确认一个新的非紧急问题后，应该在Github仓库页面增加一条新的issue，清晰地描述这个问题，比如：



在这条issue中，可以指定：

- 问题的内容
- 解决问题的人(Assignees)

发完这条issue后，在**微信通知群**发送一条消息，通知组员这条issue。

解决非紧急问题

解决非紧急问题需要创建bug分支，在创建分支之前，你需要**保证**位于和**远端主分支**同步的**本地主分支**上。

确认之后，创建bug分支，分支命名需要带有issue的编号

```
git checkout -b bug-issue-19
```

解决之后推送到远端，发送pull request，切回主分支，删除hotfix分支，与feature分支开发完成后的操作**相同**。

发送Pull Request后在**微信通知群**发送一条消息，通知新的pull request。

代码管理者

代码管理者主要负责处理其他开发者的pull request，对仓库代码风格和质量进行维护。

处理pull request

处理pull request主要分为以下几步：

- 切换分支

本地切换到该pull request对应的分支。

```
git fetch origin
git checkout branch-pull-request-at
```

- 黑盒测试

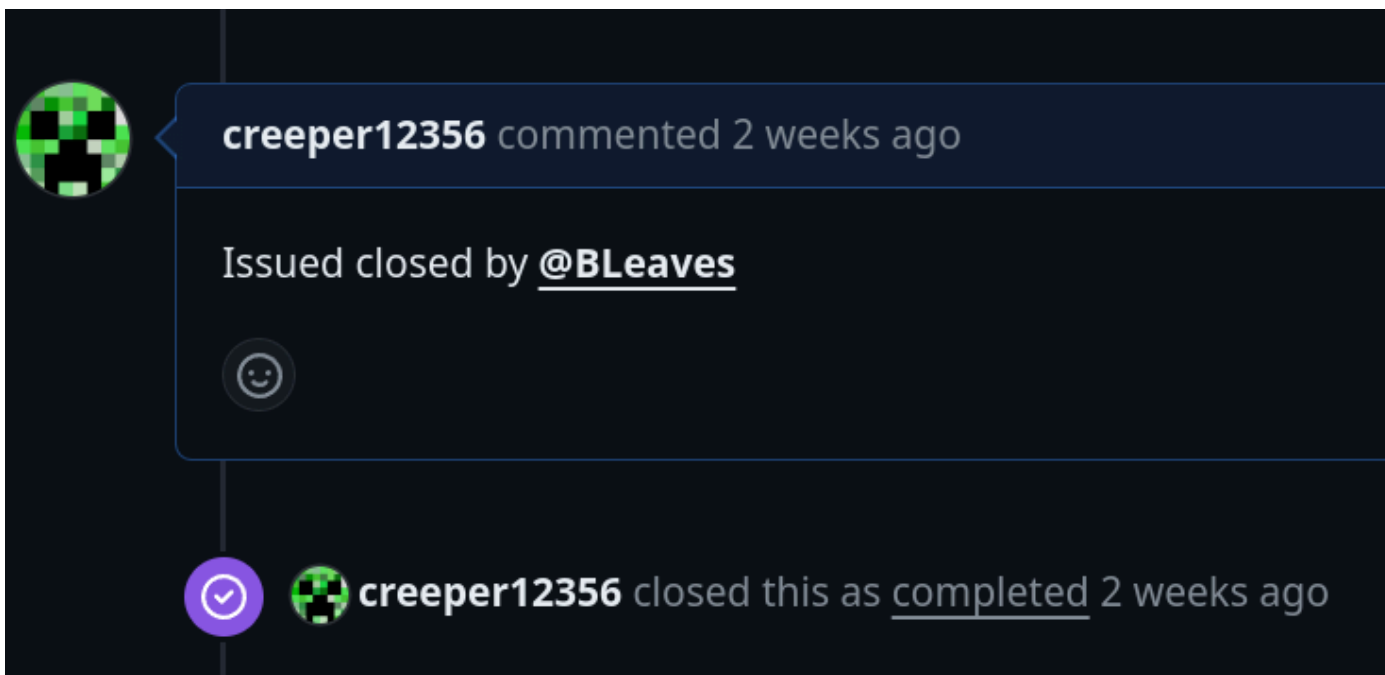
切换到该分支后，运行代码，检查是否实现了新的功能或者正确解决了问题。如果没有，通知pr发起者进行修改、或自己修改。

- 检查代码的修改

在Github Pull Request页面中，查看pull request的commit, files changed等信息。如果有问题，通知pr发起者进行修改、或自己修改。

- 关闭issue

如果pull request对应于某个issue的修复，关闭这个issue。像这样：



- 微信群通知

在**微信通知群**发送一条消息，通知pull request已经合并到主分支。