

# GitHub PR 操作简明指南

## 1. Git 基础概念

### 1.1 Commit（提交）

**简单理解：**Commit 就是保存你的代码变更，即项目变更的快照。

**基本操作：**

```
# 添加文件到暂存区  
git add 文件名  
git add . # 添加所有文件  
# 提交变更  
git commit -m "描述你做了什么修改"  
# 示例  
git add README.md  
git commit -m "更新项目说明文档"
```

### 1.2 Branch（分支）

**简单理解：**Branch 就像一棵树的主干（main）的枝叶，你可以在不影响主代码的情况下开发新功能，之所以这么做是为了保证主代码始终保持能够正常运行的状态，且便于管理，因此，当你要进行 PR 操作时，首先要做的就是创建一个独立的 branch。

**基本操作：**

```
# 查看分支  
git branch  
# 创建并切换到新分支  
git checkout -b 分支名  
# 示例：创建一个修复 bug 的分支
```

```
git checkout -b fix/login-error
```

分支命名建议：

- feature/新功能名称
- fix/问题描述
- hotfix/紧急修复

## 1.3 Merge（合并）

**简单理解：**Merge 就是将你在分支中开发的功能合并回主代码，大多数贡献者本身并没有 merge 权限，需要将你提交的代码进行审核，自动化测试等一系列流程，最终由项目的所有者或者 merge 权限的所有者将你的代码进行合并。

**基本操作：**

```
# 切换到主分支
git checkout main
# 合并分支
git merge 你的分支名
# 示例
git merge feature/user-registration
```

## 1.4 Rebase（变基）

**简单理解：**Rebase 可以让你的分支基于最新的主代码继续开发。

**基本操作：**

```
# 获取主分支最新代码
git fetch origin main
# 变基到主分支
git rebase origin/main
```

## 1.5 Merge vs Rebase（合并 vs 变基）

详细对比：

方面	Merge（合并）	Rebase（变基）
历史记录	保留完整的分支历史，形成分支树状结构	重写提交历史，形成线性的提交序列
操作特点	简单直接，一次性完成	可能需要多次解决冲突
适用场景	功能开发完成后合并到主分支	开发过程中同步主分支最新代码
冲突处理	所有冲突一次性解决	按提交顺序逐个解决冲突
团队协作	适合多人协作的公共分支	适合个人开发的私有分支
撤销难度	容易撤销（git reset）	较难撤销，可能影响他人

具体使用场景：

```
# 场景 1：功能开发完成，需要合并到主分支
# 使用 merge，保留完整的开发历史
git checkout main
git merge feature/user-authentication
git push origin main

# 场景 2：正在开发新功能，主分支有了重要更新
# 使用 rebase，将你的提交重新应用到最新主分支上
git checkout feature/payment-integration
git fetch origin main
git rebase origin/main
git push origin feature/payment-integration

# 场景 3：修复 bug 后需要快速合并
# 使用 merge，简单快速
```

```
git checkout main
git merge hotfix/critical-error
```

操作结果对比：

**Merge 前：**

```
main  A ← B ← C
      \
feature    D ← E
```

**Merge 后：**

```
main  A ← B ← C ← F (merge commit)
      \  ↑
feature    D ← E
```

**Rebase 前：**

```
main  A ← B ← C ← G
      \
feature    D ← E ← F
```

**Rebase 后：**

```
main  A ← B ← C ← G
      \
feature    D' ← E' ← F'
```

重要原则：

### 1. 公共分支用 **Merge**

- main、develop 等多人协作的分支
- 保持历史记录完整性
- 避免影响其他开发者

## 1. 私有分支用 **Rebase**

- 个人开发的 **feature** 分支
- 保持提交历史的整洁
- 方便同步主分支更新

## 1. 避免在公共分支使用 **Rebase**

- 会重写提交历史
- 可能导致其他开发者的代码冲突
- 难以追踪问题

操作建议：

```
# 日常开发流程
```

1. 从 main 创建 **feature** 分支开始开发
2. 开发过程中定期用 **rebase** 同步 main 的更新
3. 功能完成后用 **merge** 合并回 main
4. 删除 **feature** 分支

---

## 2. 环境准备

### 2.1 配置 Git

```
# 设置用户名和邮箱
```

```
git config --global user.name "你的名字"
```

```
git config --global user.email "你的邮箱"
```

```
# 示例
```

```
git config --global user.name "张三"
```

```
git config --global user.email "zhangsan@example.com"
```

## 2.2 设置 SSH Key（可选但推荐）

```
# 生成 SSH Key  
ssh-keygen -t ed25519 -C "你的邮箱"  
  
# 复制公钥  
cat ~/.ssh/id_ed25519.pub
```

然后在 GitHub 上添加 SSH Key:

1. 登录 GitHub
2. 点击头像 → Settings → SSH and GPG keys
3. 点击 New SSH key, 粘贴公钥内容

---

## 3. 使用 VSCode 进行 PR 操作

### 3.1 克隆仓库

1. 打开 VSCode
2. 点击 "Clone Repository"
3. 输入仓库地址: `git@github.com:用户名/仓库名.git`
4. 选择保存位置

---

## 6. PR 最佳实践

### 6.1 PR 描述模板

## 变更描述

简要说明你做了什么修改

## 变更类型

- [ ] Bug fix

- [ ] New feature

- [ ] Documentation update

## 测试

- [ ] 测试通过

## 相关 Issue

Closes #123

### 6.2 代码提交规范

好的提交信息：

feat: 添加用户登录功能

fix: 修复购物车计算错误

docs: 更新 API 文档

不好的提交信息：

更新

修复 bug

修改

## 6.3 合并策略选择

**Squash and merge**: 将多个小提交合并为一个

- 适用于简单的功能或 bug 修复

**Create merge commit**: 保留完整的提交历史

- 适用于复杂的功能开发

---

## 7. 常见问题解决

### 7.1 合并冲突处理

冲突产生:

当两个人修改了同一文件的同一行时会产生冲突。

解决方法:

1. 打开冲突文件
2. 查找冲突标记:

```
<<<<<<< HEAD
当前分支的代码
=====
合并分支的代码
>>>>>>> 分支名
```

1. 编辑文件，保留正确的代码
2. 删除冲突标记
3. 提交解决结果:

```
git add .
git commit -m "解决合并冲突"
```



## 7.2 撤销错误提交

```
# 撤销最后一次提交，但保留变更
git reset --soft HEAD~1
# 修改最后一次提交信息
git commit --amend -m "新的提交信息"
```

## 7.3 更新分支

```
# 获取最新代码
git fetch origin
# 切换到主分支
git checkout main
# 拉取最新代码
git pull origin main
# 切换回你的分支
git checkout 你的分支名
# 变基到主分支
git rebase origin/main
```

---

## 8. 总结

### 8.1 PR 工作流程总结

1. **Fork** 项目（如果是开源贡献）
2. **Clone** 到本地
3. 创建分支
4. 开发功能

5. 提交变更
6. 推送到远程
7. 创建 **PR**
8. 代码审查
9. 合并到主分支