

# Git

Git是分布式的代码管理工具，远程的代码管理是基于SSH的，所以要使用远程的Git则需要SSH的配置。

查看是否已经有了ssh密钥：`cd ~/.ssh`

## git常用指令

`git status` 获得git的状态

`git add -A` 添加改动到暂存区

`git commit -m "the place you changed:what you have done"` 提交commit

`git branch` 查看当前的branch

`git branch "name"` 新建一个名为"name"的branch

`git checkout -b "name"` 新建名为name 的分支

`git branch -d "name"` 删除名为"name"的branch

`git merge "name"`

`git checkout "name"`

`git remote` 查看远程仓库

`git log` 查看log

`git push origin Branch1` 将新分支发布在github上：

---

## Merge

master 分支合并到 feature 分支最简单的办法就是用下面这些命令：

`git checkout feature`

`git merge master`

Merge 好在它是一个安全的操作。现有的分支不会被更改，避免了 rebase 潜在的缺点（后面会说）。

另一方面，这同样意味着每次合并上游更改时 feature 分支都会引入一个外来的合并提交。如果 master 非常活跃的话，这或多或少会污染你的分支历史。虽然高级的 `git log` 选项可以减轻这个问题，但对于开发者来说，还是会增加理解项目历史的难度。

## 取代pull merge的方法：rebase

`git rebase -i [startpoint][endpoint]`

-i means `interactive`，弹出交互式的界面让用户编辑完成合并操作，`[startpoint] [endpoint]` 则指定了一个编辑区间，如果不指定 `[endpoint]`，则该区间的终点默认是当前分支HEAD所指向的 commit（注：该区间指定的是一个前开后闭的区间）。

它会把整个 feature 分支移动到 master 分支的后面，有效地把所有 master 分支上新的提交并入过来。但是，rebase 为原分支上每一个提交创建一个新的提交，重写了项目历史，并且不会带来合并提交。

rebase最大的好处是你的项目历史会非常整洁。首先，它不像 git merge 那样引入不必要的合并提交。其次，如上图所示，rebase 导致最后的项目历史呈现出完美的线性——你可以从项目终点到起点浏览而不需要任何的 fork。这让你更容易使用 `git log`、`git bisect` 和 `gitk` 来查看项目历史。

不过，这种简单的提交历史会带来两个后果：安全性和可跟踪性。如果你违反了 rebase 黄金法则，重写项目历史可能会给你的协作工作流带来灾难性的影响。此外，rebase 不会有合并提交中附带的信息——你看不到 feature 分支中并入了上游的哪些更改。

## 交互式的 rebase

交互式的 rebase 允许你更改并入新分支的提交。这比自动的 rebase 更加强大，因为它提供了对分支上提交历史完整的

控制。一般来说，这被用于将 feature 分支并入 master 分支之前，清理混乱的历史。

把 -i 传入 git rebase 选项来开始一个交互式的rebase过程：

```
git checkout feature  
git rebase -i master
```

```
git rebase -i master  
git rebase -i origin/master
```

区别

## Rebase instructions

Graph	Description	Commit
	master commit 8,9	1a222c3
	commit 6,7	02501fb
	commit temp	ce81811
	branch2 commit aaa	6382c7d
	branch1 commit 5.5	1ae50cd
	commit 5	0a0cdf6
	commit 4	0cf92f0
	commit 3	9fc9896
	commit 2, 2.5	cc59b55
	commit 1	d1abf43
	add test file	cde6c09

图24

cherry-pick其实在工作中还挺常用的，一种常见的场景就是，比如我在A分支做了几次commit以后，发现其实我并不应该在A分支上工作，应该在B分支上工作，这时就需要将这些commit从A分支复制到B分支去了，这时候就需要cherry-pick命令了，B分支指着这些commit说：妈妈，我也要！比如说，我们在master分支上继续做两次提交，第一次添加一行“test 10”，git commit -am “commit 10”，第二次添加“test 11”，到达如下图的状态：

Graph	Description	Commit
	master commit 11	1a04d5f
	commit 10	0bda20e

图25

这个时候我们发现，哦NO，我们不应该直接更改master分支，我们应该在自己的分支上做提交。这个时候先新建一个分支git checkout -b branch3 1a222c3，注意这里的最后一个参数是新分支的起点，也就是说，新的分支branch3是从“commit 8,9”开始的，现在我们需要把刚才的两次提交移动到新的分支上。运行git cherry-pick 0bda20e 1a04d5f，命令行会给出提示两个commit被复制到了当前分支上，此时SourceTree的状态如下图：

Graph	Description	Commit
	branch3 commit 11	ddaf7d
	commit 10	6e841da
	master commit 11	1a04d5f
	commit 10	0bda20e
	commit 8,9	1a222c3
	commit 6,7	02501fb
	commit temp	ce81811

图26

确定这两个commit被复制到指定分支以后，在master分支上将这两个commit删除。先切回master分支：git checkout master，运行git reset –hard 1a222c3，此时SourceTree的状态图为：

Graph	Description	Commit
	branch3 commit 11	ddaf7d
	commit 10	6e841da
	master commit 8,9	1a222c3
	commit 6,7	02501fb
	commit temp	ce81811
	branch2 commit aaa	6382c7d
	branch1 commit 5.5	1ae50cd
	commit 5	0a0cdf6
	commit 4	0cf92f0
	commit 3	9fc9896
	commit 2, 2.5	cc59b55
	commit 1	d1abf43
	add test file	cde6c09

图27

两个commit被成功的从master分支移动到了branch3分支。

```
git cherry-pick <Commit1> <Commit2> <...>
```

## Rebase 的黄金法则

当你理解 rebase 是什么的时候，最重要的就是什么时候不能用 rebase。git rebase 的黄金法则便是，绝不要在公共的分支上使用它。

比如说，如果你把 master 分支 rebase 到你的 feature 分支上会发生什么：

这次 rebase 将 master 分支上的所有提交都移到了 feature 分支后面。问题是它只发生在你的代码仓库中，其他所有的开发者还在原来的 master 上工作。因为 rebase 引起了新的提交，Git 会认为你的 master 分支和其他人的 master 已经分叉了。

同步两个 master 分支的唯一办法是把它们 merge 到一起，导致一个额外的合并提交和两堆包含同样更改的提交。不用说，这会让人非常困惑。

所以，在你运行 **git rebase** 之前，一定要问问你自己「有没有别人正在这个分支上工作？」。如果答案是肯定的，那么把你的爪子放回去，重新找到一个无害的方式（如 **git revert**）来提交你的更改。不然的话，你可以随心所欲地重写历史。

---

## 修改**git**默认的编辑器

**git**默认的编辑器为**nano**，不常用，需要修改为**vim**，方法如下：

打开**.git/config**文件，在**core**中添加 **editor=vim**即可。

或者运行命令 **git config --global core.editor vim** 修改更加方便。

---