Git使用指北

内容参考：https://www.liaoxuefeng.com/wiki/896043488029600/896067074338496

什么是Git？Git是目前世界上最先进的分布式版本控制系统（没有之一）（C语言编写）

什么是版本控制系统？自动记录每次文件的改动，方便文件（程序）的协作编辑

沉下心往下看，你会理解Git的强大功能

以linux系统为例进行演示，windows系统供参考（实际上是一样的）

请确保你的系统已经安装并配置好git环境。如果没有安装，这里我贴一个链接供你参考

```
# 确保已正确安装并配置好git
git

usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-
path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [-
-bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

```
# 在合适的位置建立文件夹
mkdir learngit
cd learngit
```

```
# 初始化git版本库
git init
warning: templates not found in /Users/bytedance/.gittemplates
Initialized empty Git repository in
/Users/bytedance/PycharmProjects/learngit/.git/
```

当前文件夹中还是没有文件的（除了隐藏的.git版本库），我们新建一个文件用于测试

```
vim readme.txt
# 写入数据如下:
Git is a version control system.
Git is free software.
# 保存退出
```

我们当前新建了一个文件，但版本库并没有感知到，我们需要进行一些操作让版本库也添加这个文件

添加文件到Git仓库，分两步：

1. 使用命令 `git add <file>`，可反复多次使用，添加多个文件
2. 使用命令 `git commit -m <message>`，提交所有添加，完成

```
# 我们把新建的readme.txt文件提交至Git仓库
git add readme.txt
git commit -m "my first commit"
```

我们怎么知道上面的操作成功了呢？可以使用 `git status` 命令查看当前仓库与本地的差异

我们先对本地文件进行一些修改

```
vim readme.txt

# 修改数据如下:
Git is a distributed version control system.
Git is free software.
# 保存退出
```

```
# 我们使用git status查看一下当前状态
git status

On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

上面的代码表示：文件 `readme.txt` 被修改了但还没有添加到准备提交阶段（也就是没有add）

我们add一下

```
git add readme.txt
git status

On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   readme.txt
```

我们将修改的文件添加（add）到待提交队列，此时用 `git status` 查看状态：提示 `readme.txt` 文件待提交

我们提交一下

```
git commit -m "first revise"
[master 3e4e2c1] first revise
 1 file changed, 1 insertion(+), 1 deletion(-)
git status
On branch master
nothing to commit, working tree clean
```

commit完毕，会提示修改的内容，查看状态，提示：没有需要提交的内容了，工作目录与Git仓库完全一致

小结

1. 要随时掌握工作区的状态，使用`git status`指令
2. 如果`git status`告诉你有文件被修改过，还可以使用`git diff`查看修改内容

版本回退

现在，你已经学会了修改文件，并把修改提交到Git版本库，你可以再联系一次

在实际操作过程中，我们不可能记住每一次修改的内容，版本控制系统有指令会告诉我们历史记录，在Git中，可以使用`git log`命令查看

```
git log

commit 3e4e2c1886ce79c51518212ac6eb25f302092e7e (HEAD -> master)
Author: wangzhenghui.bupt <wangzhenghui.bupt@bytedance.com>
Date:   Sun Apr 25 15:46:50 2021 +0800

    first revise

commit 53a1b745b20463803fa1b47a28e8dcafd4afda29
Author: wangzhenghui.bupt <wangzhenghui.bupt@bytedance.com>
Date:   Sun Apr 25 11:23:42 2021 +0800
```

```
    wrote a readme file
```

`git log` 命令可以看到从最近到最远的三次提交，前面一大串类似 `3e4e2c1886` 的是 `commit id` 版本号

现在，我们希望回退一个版本，该怎么做呢？比如，我现在要退回一开始的 `readme.txt` 文件，应该怎么做？我们可以根据 `commit id` 找到对应版本，然后用 `git reset --hard <commit id>` 的方式把本地代码刷新回之前的版本

```
git reset --hard 53a1b745b20463803fa1b47a28e8dcafd4afda29
HEAD is now at 53a1b74 wrote a readme file
```

我们再来查看一下 `readme.txt` 文件

```
vim readme.txt

Git is a version control system.
Git is free software.
```

可以看到， `readme.txt` 文件已经回到了最初的版本

我们现在想回到之前最新的版本怎么办？先使用 `git log` 指令看一下

```
git log

commit 53a1b745b20463803fa1b47a28e8dcafd4afda29 (HEAD -> master)
Author: wangzhenghui.bupt <wangzhenghui.bupt@bytedance.com>
Date:   Sun Apr 25 11:23:42 2021 +0800

    wrote a readme file
```

在我们回退了一次以后，之前的最新提交已经看不到了，怎么办？要想恢复到某次提交，必须找到对应的 `commit id` ， Git提供了一个命令 `git reflog` 用来记录每一次命令

```
git reflog

53a1b74 (HEAD -> master) HEAD@{0}: reset: moving to
53a1b745b20463803fa1b47a28e8dcafd4afda29
3e4e2c1 HEAD@{1}: commit: first revise
53a1b74 (HEAD -> master) HEAD@{2}: reset: moving to
53a1b745b20463803fa1b47a28e8dcafd4afda29
52d8828 HEAD@{3}: reset: moving to
52d8828903e0d6e629c1f1774dfc0ba611e5b669
53a1b74 (HEAD -> master) HEAD@{4}: reset: moving to
53a1b745b20463803fa1b47a28e8dcafd4afda29
52d8828 HEAD@{5}: commit: update
53a1b74 (HEAD -> master) HEAD@{6}: commit (initial): wrote a readme file
```

我们应该是想回退到 `first revise` 这个版本，因此我们故技重施：

```
git reset --hard 3e4e2c1
```

再次打开 `readme.txt` 文件，你会发现又回到了最新的版本（带distributed）

小结

1. 需要注意一下， `HEAD` 指向的版本就是当前版本
2. 如果我们希望回到某个版本，我们可以是用命令： `git reset --hard <commit id>`
3. 我们可以利用 `git log` 查看提交历史，确定要回到过去的哪个版本
4. 也可以使用 `git reflog` 查看命令历史，以确定要回到“未来”哪个版本

工作区和暂存区

工作区（Working Directory）：就是在电脑里能看到的目录，比如我创建的 `learngit` 文件夹，就是一个工作区

版本库（Repository）：工作区有一个隐藏目录 `.git`，这个不算工作区，而是Git的版本库

Git版本库里存了很多东西，其中比较重要的有称为stage的暂存区，还有Git为我们创建的第一个分支 `master` ，以及指向 `master` 的一个指针叫 `HEAD`

分支和 `HEAD` 的概念以后再讲，前面讲的把文件往版本库添加的时候分两步执行

第一步是用 `git add` 把文件添加进去，实际上这就是把文件修改添加到暂存区

第二步是用 `git commit` 提交更改，实际上就是把暂存区的所有内容提交到当前分支

因为我们创建Git版本库时，Git自动为我们创建了唯一一个 `master` 分支，所以，现在 `git commit` 就是在往 `master` 分支上提交更改

简单理解为：需要提交的文件修改通通先放入暂存区，然后，一次性提交暂存区的所有修改

实践出真知，我们对 `readme.txt` 文件再次进行修改，同时新建一个文件 `index.txt`

```
vim readme.txt

Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
```

```
vim index.txt

Hello World!
```

复习一下，用 `git status` 查看一下当前版本库状态

```
git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   readme.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    index.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

提示你：`readme.txt`被修改了，`index.txt`为`untracked`状态（即新建文件的状态）

我们使用命令`git add .`一次性提交所有更新

```
git add .
```

学习了暂存区的概念以后，你应该理解，`add`命令实际上就是把修改的文件放入了暂存区

当你使用`commit`后，就意味着将暂存区的所有内容提交到当前分支上了

一旦提交以后，只要没有修改，你的工作区就是“干净”的

```
On branch master
nothing to commit, working tree clean
```

撤销修改

假设现在是凌晨两点，你被领导要求赶进度，你在`readme.txt`中加了一行：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
My stupid boss still prefers SVN.
```

在你准备提交之前，你发现了这个问题，你意识到`stupid boss`可能使你丢掉这份工作！

错误发现的很及时，你可以很容易地纠正它。你可以直接删除最后一行，手动把文件恢复到上一个版本的状态。如果用`git status`查看一下：

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Git提示你可以使用`git restore <file>`来丢弃工作区的修改

尝试一下

```
git restore readme.txt
# 实际上，你还可以这样做
git checkout -- readme.txt # 效果是一样的
```

查看一下`readme.txt`

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
```

果然恢复了

这个命令就是让文件恢复到最近一次 `commit` 的状态

假设现在是凌晨三点，你不但写了一些胡话，还 `git add` 到暂存区了。庆幸的是，在 `commit` 之前你发现了这个问题

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   readme.txt
```

通过 `git status` 命令，你得到了提示，使用 `git restore --staged <file>` 来放弃 `unstaged` 暂存区的内容

```
git restore --staged readme.txt
git reset HEAD readme.txt # 这可以实现相同的功能
```

不过通过上面的命令，你还没有撤销对文件的修改，你只是撤销了自己 `git add readme.txt` 的命令。所以你还需要：

```
git restore readme.txt
```

这样，你的修改就被移除了

现在，假设你不但改错了东西，还从暂存区提交到了版本库，应该咋办？相信你是有解决方案的（没有的话，请往上翻，找找思路哈～）

小结

1. 当你改乱了工作区的内容，想直接丢弃工作区，你可以使用命令：`git restore <file>` 来丢弃工作区的修改
2. 当你改乱了工作区并且保存至暂存区了，你可以使用命令：`git restore --staged <file>` 来清空暂存区（注意，你的工作区是没有改变的，你需要使用1. 的命令丢弃工作区
3. 当你改乱了工作区且已经提交了，请参考版本回退一节

删除文件

在Git中，删除也是一种修改操作。实战一下，我们先添加一个文件 `test.txt` 到Git并且提交

```
vim test.txt

It's just a file for test!
```

```
# 我们把新建的文件提交到版本库
git add test.txt
git commit -m "summit the test file"
```

Ok，当前文件已经提交到版本库中，现在我们用 `rm` 指令把刚刚添加的文件删除

```
rm test.txt
```

查看一下工作区的状态

```
git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:    test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Git提示你：你的修改还没有 `staged` ，现在你有两个选择

1. 你确实要从版本库中移除这个文件

    ```
    git rm test.txt
    git commit -m "remove test.txt"
    ```

这样，文件就从版本库中被删除了

2. 你误删了这个文件，现在需要还原它

```
git checkout -- test.txt
cat test.txt

It's just a file for test!
```

文件又恢复了！

小结

1. 命令 `git rm` 用于删除一个文件。如果一个文件已经被提交到版本库，那么你根本不需要担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容
2. `git checkout` 其实使用版本库里的版本替换工作区的版本，无论工作区中发生了修改还是删除，都可以“一键还原”

远程仓库

到目前为止，我们已经可以比较熟练地在本地操控Git帮助我们进行版本控制。如果只是在一个仓库里管理文件历史，Git和SVN确实没啥区别，下面开始介绍Git的杀手级功能之一：远程仓库

Git是分布式管理控制系统，同一个Git仓库可以分布到不同的机器上。最早，肯定只有一个原始版本库，此后，别的机器可以“克隆”这个原始版本库，而且每台机器的版本库其实都是一样的，并没有主次之分

在继续阅读后续内容前，请自行注册GitHub账号。由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置，详细的过程请参考：GitHub SSH设置

确保你拥有一个GitHub账号后，我们就即将开始远程仓库的学习

添加远程库

现在的情景是：你已经在本地创建了一个Git仓库，你希望把它部署到GitHub服务器上，并且让两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作。首先登陆GitHub，在右上角找到 `create a new repo` 按钮，创建一个新的仓库

在Repository name填入 `learngit` ，其他保持默认设置，点击 `Create repository` 按钮，就成功地创建了一个新的Git仓库

目前，在GitHub上的这个 `learngit` 仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。现在，我们根据提示，在本地的 `learngit` 仓库下运行如下命令：

```
git remote add origin git@github.com:bupt-zhenghui/learngit.git
```

注意：请务必使用自己的路径！否则你在本地关联的就是我的远程库，关联没有问题，但是你的push行为是无效的，因为你的SSH Keys公钥不在我的账户列表中

添加后，远程库的名字就是 `origin` ，这是Git默认的叫法，也可以改成别的，但是 `origin` 这个名字一看就知道是远程库，所以无必要别乱改！

下一步，我么就可以把本地库的所有内容推送到远程库上：

```
git push -u origin master

Enumerating objects: 16, done.
Counting objects: 100% (16/16), done.
Delta compression using up to 12 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (16/16), 1.39 KiB | 1.39 MiB/s, done.
Total 16 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), done.
To github.com:bupt-zhenghui/learngit.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

把本地库的内容推送到远程，用 `git push` 命令，实际上是把当前分支 `master` 推送到远程。由于远程库是空的，我们第一次推送 `master` 分支时，加上了 `-u` 参数，Git不但会把本地的 `master` 分支内容推送到远程新的 `master` 分支，还会把本地 `master` 分支和远程 `master` 分支关联起来，在以后的推送或者拉取时就可以简化命令

推送成功后，可以立刻在GitHub页面看到远程库的内容已经和本地一模一样。从现在起，只要本地做了提交，就可以通过 `git push origin master` 把本地 `master` 分支的最新修改推送至GitHub，现在，真正的分布式版本库已然建立

当前，你的本地版本库与远程库的内容应该是一致的，如果你不确定，可以做一个验证

```
git push origin master
Everything up-to-date
```

Git提示你 `Everything up-to-date` ，表示当前本地库与远程库完全一致

你可以对本地的 `index.txt` 文件做一些修改，然后提交，再推送远程分支

```
git add .
git commit -m "update index.txt"
git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

`commit` 以后，查看status，Git会提示你本地库已经领先远程库一次提交，我们把提交同步到远程库

```
git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 335 bytes | 335.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:bupt-zhenghui/learngit.git
   20f5037..e493a21  main -> main
```

删除远程库

如果添加的时候地址写错了，或者就是想删除远程库，可以用 `git remote rm <name>` 命令。使用前，建议先用 `git remote -v` 查看远程库信息

此处的删除指的是接触了本地和远程的绑定关系，并不是物理上删除了远程库。远程库本身并没有任何改动，要真正删除远程库，需要登录到GitHub，在后台页面找到删除按钮

小结

1. 关联一个远程库，使用 `git remote add origin git@github.com:bupt-zhenghui/learngit.git` 类似命令
2. 关联一个远程库时必须给远程库制定一个名字， `origin` 是默认习惯命名
3. 关联后，使用 `git push -u origin master` 第一次推送 `master` 分支的所有内容
4. 此后，每次本地提交后，只要有必要，就可以使用命令 `git push origin master` 推送最新修改

分布式版本系统最大好处之一是在本地工作完全不需要考虑远程仓库的存在，有没有联网都可以正常工作

从远程库克隆

上面我们讲了把本地库上传到远程库的流程，现在，假设我们需要copy别人的成型项目并做一些修改（很多人都干过吧），我们需要从远程拉取项目

```
$ git clone git@github.com:michaelliao/gitskills.git
Cloning into 'gitskills'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 3
Receiving objects: 100% (3/3), done.
```

注意把Git库的地址换成你自己的，这一部分内容相对简单，就不细说了

分支管理

分支在实际中有什么用呢？假设你是抖音团队的后端研发人员，你现在准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码没有写完，不完整的代码库会导致之前版本出现问题，进而导致抖音某个部分发生崩溃；如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，知道开发完毕后，再一次行合并到原来的分支上，这样，既安全，又不影响别人工作。Git分支是非常高效的，无论创建、切换还是删除分支，Git在1秒钟之内就能完成，无论你的版本库是1个文件还是1万个文件

创建与合并分支

之前的内容里，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即 `master` 分支。 `HEAD` 严格来说不是指向提交，而是指向 `master` ， `master` 才是指向提交的，所以， `HEAD` 指向的就是当前分支

一开始的时候， `master` 分支是一条线，Git用 `master` 指向最新的提交，再用 `HEAD` 指向 `master` ，就能确定当前分支，以及当前分支的提交点；每次提交， `master` 分支都会向前移动一步，这样，随着你不断提交， `master` 分支的线也越来越长

当我们创建新的分支时，例如 dev 时，Git新建了一个指针叫 dev ，指向 master 相同的提交，再把 HEAD 指向 dev ，就表示当前分支在 dev 上；因此，Git创建一个分支很快，仅仅是增加一个 dev 指针，改改 HEAD 的指向，工作区的文件没有发生任何变化。从现在开始，对工作区的修改和提交就是针对 dev 分支了，比如新提交一次后， dev 指针往前移动一步，而 master 指针不变

假如我们在 dev 上的工作完成了，就可以把 dev 合并到 master 上。Git怎么合并呢？最简单的方法，就是直接把 master 指向 dev 的当前提交，直接完成合并；所以Git合并分支也很快，就改改指针，工作区内容也不变

在实际开发中，我们应该按照几个基本原则进行分支管理

1. 首先， master 分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活
2. 那在哪干活呢？干活都在 dev 分支上，也就是说， dev 分支是不稳定的，到某个时候，比如1.0版本发布时，再把 dev 分支合并到 master 上，在 master 分支发布1.0版本
3. 你和你的伙伴每个人都在 dev 分支上干活，每个人都有自己的分支，时不时往 dev 分支上合并就可以了

实践一下

我们先创建一个 dev 分支，然后切换到 dev 分支

```
git branch dev
git checkout dev
Switched to a new branch 'dev'
```

实际上你可以使用一条命令完成上面创建+切换的功能：

```
# 创建dev分支并直接切换到该分支上
git checkout -b dev
```

可以使用 git branch 查看当前分支状态

```
git branch
* dev
  main
```

`git branch` 会列出所有分支，当前分支前面会标一个 `*` 号；然后，我们就可以在 `dev` 分支上正常提交；你可以对readme.txt做个修改，加上一行：

```
Creating a new branch is quick.
```

然后提交

```
git add .
git commit -m "new branch commit"
```

我们切回 `master` 分支看一眼文件

```
git checkout master
cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
My stupid boss still prefers SVM.
```

增加的一行不见了，这是因为那个提交是在 `dev` 分支上，而 `master` 分支此刻的提交点并没有改变；现在，我们把 `dev` 分支的工作成果合并到 `master` 分支上：

```
git merge dev
Updating e493a21..9ccefe3
Fast-forward
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

你可以再看一眼 `readme.txt` 文件，肯定是发生了变化的

`git merge` 命令用于合并指定分支到当前分支；注意到上面的 `fast-forward` 信息，Git告诉我们，这次合并时“快进模式”，也就是直接把 `master` 指向 `dev` 的当前提交，所以合并速度非常快；当然，不是每次合并都能 `Fast-forward` ，我们后面会讲到其他方式

合并完成后，就可以放心地删除 `dev` 分支了

```
git branch -d dev
Deleted branch dev (was 9ccefe3).
```

删除后，查看 `branch` ，就只剩下 `master` 分支了。因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删除分支，这和直接在 `master` 分支上工作效果是一样的，但过程更安全

小结

1. 查看分支状态： `git branch`
2. 创建分支： `git branch <branch-name>`
3. 切换分支： `git checkout <branch-name>` 或 `git switch <branch-name>`
4. 创建+切换分支： `git checkout -b <branch-name>` 或 `git switch -c <branch-name>`
5. 合并某分支到当前分支： `git merge <branch-name>`
6. 删除分支： `git branch -d <branch-name>`

解决冲突

什么情况下会产生冲突？这很好理解。你和你的团队成员在共同推进一个项目，你们都在往同一个分支提交内容；你的伙伴先把他更新的内容推到了 `master` 分支，紧接着你也把你的分支推到 `master` 上，但你和你的伙伴修改了同一个地方，这就会产生问题——到底用谁的代码？冲突由此产生

还有一种情况：你提交了本地的修改，然后拉取远程分支的最新代码，如果最新的代码中更新的内容与你本地更新的内容在同一位置，就会产生相同的问题

解决冲突并不复杂，但一般需要产生冲突的同学合作解决问题（简单理解就是：到底用谁的代码。用A的代码，就要确保：撤销B的修改不影响运行的逻辑）

为了尽量避免这个问题，请每天开始修改代码之前先拉一版远程分支的最新提交，每次 `git push` 之前也作一次同样的操作，将冲突在本地解决后再提交

我们模拟一下产生冲突、解决冲突的过程

先准备一个新的分支 `feature1`，继续我们的新分支开发，我们修改 `readme.txt` 最后一行，改为：

```
Create a new branch is quick AND simple
```

```
git add .
git status
git checkout -b master
```

切回 `master` 分支以后，我们对 `readme.txt` 作不同的修改：

```
Create a new branch is quick & simple
```

提交以后，我们尝试把 `feature1` 上的更新同步到 `master` 上

```
git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

在这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突。当前Git已经提示我们存在冲突

我们可以使用 `git status` 查看当前状态：

```
git status
On branch main
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)
```

```
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

可以直接进入冲突的文件查看具体的冲突内容：

```
vim readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
My stupid boss still prefers SVM.
<<<<<<< HEAD
Creating a new branch is quick & simple.
=======
Creating a new branch is quick And simple.
>>>>>>> feature1
```

Git用 `<<<<<<<` ， `=======` ， `>>>>>>>` 标记出不同分支的内容，我们修改如下后保存：

```
Creating a new branch is quick and simple.
```

修改完成后重新提交：

```
git add .
git status -m "conflict fixed"
[main 1a0ef4d] conflict fixed
```

用带参数的 `git log` 可以看到分支的合并情况

```
git log --graph --pretty=oneline --abbrev-commit

*   1a0ef4d (HEAD -> main) conflict fixed
|\
| * bfc0568 (feature1) commit to feature1
* | f346e88 main commit
|/
* 9ccefe3 new branch commit
* e493a21 (origin/main) index update
* 20f5037 2 commit
* a9d730d summit the test file
* 7ec044a add a new index.txt
* 3e4e2c1 first revise
* 53a1b74 wrote a readme file
```

最后，删除 `feature1` 分支，完成

```
git branch -d feature1
Deleted branch feature1 (was bfc0568).
```

小结

1. 当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成
2. 解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交
3. 用 `git log --graph` 命令可以看到分支合并图

Bug分支

软件开发中，bug就像家常便饭一样。有了bug就要修复，在Git中，由于分支的强大，每个bug都可以通过一个新的分支来修复，修复后，合并分支，然后将临时分支删除。当你接到一个修复代号为101的bug任务时，很自然地，你会创建一个分支 `issue-101` 来修复它，但是，你当前可能正在 `dev` 分支上开发一个新项目

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需要1天时间。但是，你必须要在两个小时内修复bug，怎么办?

幸好，Git还提供了一个 `stash` 功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作

我们实践一下，假设我们当前在 `dev` 分支上开发，要修复 `master` 分支的一个bug

我们先查看一下 `dev` 分支当前的状态

```
git status
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

我们在 `dev` 分支上的修改还没有提交，我们使用 `git stash` 尝试一下

```
git stash
Saved working directory and index state WIP on dev: 1a0ef4d conflict
fixed
```

现在，用 `git status` 查看工作区:

```
git status
On branch dev
nothing to commit, working tree clean
```

工作区已经还原了。现在我们需要先投入到修复bug的工作中；我们需要先切回`master`分支，在`master`分支上新建修复bug的分支`issue-101`

```bash
git checkout master
git checkout -b issue-101
Switched to a new branch 'issue-101'
```

现在修复bug，需要把 `readme.txt` 中的 `Git is free software` 修改为 `Git is a free software` ，修改完成后提交；修复完成后，切换到 `master` 分支，完成合并，最后删除 `issue-101` 分支

```
git add readme.txt
git commit -m "fix bug"
git checkout master
git merge issue-101
Updating 1a0ef4d..40622e4
Fast-forward
 readme.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

git branch -d issue-101
git checkout dev
```

现在，你已经修复了 `master` 分支上的bug，现在切回 `dev` 分支继续开发，怎么恢复之前的工作区内容呢？可以通过 `git stash list` 命令看看

```
git stash list
stash@{0}: WIP on dev: 1a0ef4d conflict fixed
```

工作区的内容还在，但是需要恢复一下，有两个办法

1. 用 `git stash apply` 恢复，但恢复后， `stash` 内容不删除，需要用 `git stash drop` 来删除
2. 用 `git stash pop` ，恢复的同时把 `stash` 也删了

```
git stash apply stash@{0}
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

在 `master` 分支上修复了bug后，我们要想一想，dev分支是早期从master分支分出来的，所以，这个bug其实在当前dev分支上也存在。那么问题来了：怎么在dev分支上修复同样的bug？重复操作一次？我们有更简单的办法——同样的bug，要在 `dev` 上修复，我们只需要把上一次合并操作“重演”一遍就可以。为了方便操作，Glt专门提供了一个 `git cherry-pick` 命令，让我们能复制一个特定的提交到当前分支

实现一下，我们先找到上一次主分支上的 `merge` id，用 `git reflog`

```
git reflog
1a0ef4d (HEAD -> dev) HEAD@{0}: checkout: moving from main to dev
40622e4 (main) HEAD@{1}: merge issue-101: Fast-forward
1a0ef4d (HEAD -> dev) HEAD@{2}: checkout: moving from issue-101 to main
40622e4 (main) HEAD@{3}: commit: fix bug
1a0ef4d (HEAD -> dev) HEAD@{4}: checkout: moving from main to issue-101
1a0ef4d (HEAD -> dev) HEAD@{5}: checkout: moving from dev to main
1a0ef4d (HEAD -> dev) HEAD@{6}: reset: moving to HEAD
......
```

我们可以找到上次合并的操作ID： `40622e4`

把这次合并在 `dev` 分支上进行复现，因为刚才进行了 `stash` 操作，我们先回退一下

```
git restore .
git cherry-pick 40622e4
[dev d365280] fix bug
 Date: Fri Apr 30 17:31:57 2021 +0800
 1 file changed, 1 insertion(+), 1 deletion(-)
```

刚才修复bug的代码已经合到新的分支上了，然后再 `stash`

小结

1. 修复bug时，我们会创建新的bug分支
2. 当手头工作没有完成时，先把工作现场 `git stash` 一下，然后去修复bug，修复后，再 `git stash pop` 回到工作现场
3. 在 `master` 分支上修复的bug，想要合并到当前的 `dev` 分支，可以用 `git cherry-pick <commit-id>` 命令，把bug提交的修改“复制”到当前分支，避免重复劳动

Feature分支

软件开发中，总有无穷无尽的新的功能要不断添加进来。添加一个新功能时，肯定不能把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。

假设现在你接到一个任务：开发代号为Vulcan的新功能，该功能计划用于下一代星际飞船

于是你准备开发：你会新建一个 `feature-vulcan` 分支，在上面完成开发任务后合并到 `master` 分支上

```
git checkout -b feature-vulcan
vim vulcan.py
print("Vulcan project is done!")

git add .
git commit -m "commit vulcan project"
git checkout master
```

5分钟后，开发完成。我们切回 `master` 分支，准备合并。但是就在此时，接到上级命令，因为经费不足，新功能必须取消。虽然白干了，但是这个包含高度机密文件的分支必须被删除：

```
git branch -d feature-vulcan
error: The branch 'feature-vulcan' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature-
vulcan'.
```

但此时Git提示：无法删除这个分支，因为这个分支从没有被合并过。如果要强行删除，需要使用大写的 `-D` 参数

```
git branch -D feature-vulcan
Deleted branch feature-vulcan (was 63f6dc5).
```

成功删除该分支

小结

1. 开发一个新feature，最好新建一个分支
2. 如果要丢弃一个没有被合并过的分支，可以通过 `git branch -D <branch-name>` 强行删除

多人协作

当你从远程仓库克隆时，实际上Git自动把本地的 `master` 和远程的 `master` 分支对应起来了，并且，远程仓库的默认名称是 `origin` ，要查看远程库的信息，用 `git remote` ：

```
git remote
origin
```

或者用 `git remote -v` 查看更详细信息

```
git remote -v
origin  git@github.com:bupt-zhenghui/learngit.git (fetch)
origin  git@github.com:bupt-zhenghui/learngit.git (push)
```

上面显示了可以抓取和推送的 `origin` 的地址。如果没有推送权限，就看不到push的地址

推送分支

就是把该分支上所有本地提交推送到远程库。推送时，要制定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

```
git push origin master
```

如果要推送其他分支，比如 `dev` ，就改成：

```
git push origin dev
```

并不是一定要把本地分支往远程推送，哪些需要推送，哪些不要呢?

1. `master` 分支是主分支，因此要时刻与远程同步
2. `dev` 分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步
3. bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了多少bug～
4. feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发

抓取分支

多人协作时，大家都会往 `master` 和 `dev` 分支上推送各自的修改

现在，模拟一个你的小伙伴，可以在另一台电脑或者同一台电脑的另一个目录下克隆

```
git clone git@github.com:bupt-zhenghui/learngit.git
Cloning into 'learngit'...
warning: templates not found in /Users/bytedance/.gittemplates
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 19 (delta 3), reused 19 (delta 3), pack-reused 0
Receiving objects: 100% (19/19), done.
Resolving deltas: 100% (3/3), done.
```

现在，你的小伙伴拿到了远程的 `master` 分支，但他需要在 `dev` 分支上进行开发，默认情况下，他当前只能看到本地的 `master` 分支。不信可以用 `git branch` 命令看看：

```
git branch
* master
```

你的小伙伴要在 `dev` 分支上开发，就必须创建远程 `origin` 的 `dev` 分支到本地，于是他用这个命令创建本地 `dev` 分支

```
git checkout -b dev origin/dev
```

注意：如果此时你的远程库还没有 `dev` 分支，你需要把它在本地版本库中推送到远程：

```
git checkout dev
git push origin dev
```

现在，你的小伙伴的本地代码库已经有 `dev` 分支了，他将会对这个分支进行修改，然后提交到远程：

```
vim hacker.py
print("I'm a hacker hahahaha")

git add .
git commit -m "hacker commit"
git push origin dev
```

你的小伙伴已经向 `origin/dev` 分支推送了他的提交，现在你可以在本地拉取最新的 `dev` 分支：

```
git pull origin dev
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:bupt-zhenghui/learngit
 * branch            dev        -> FETCH_HEAD
   0f99df5..e364bda  dev        -> origin/dev
Updating 0f99df5..e364bda
Fast-forward
 hacker.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 hacker.py
```

现在，你成功的获取到了你的伙伴更新过的 `dev` 分支

很多时候并不是一帆风顺的，你可能和你的小伙伴修改了相同的代码，这时候就会产生冲突；解决办法也很简单，先用 `git pull` 把最新的提交从 `origin/dev` 上抓下来，然后，在本地合并，解决冲突，再推送

综上所述，多人协作的工作模式通常是这样：

1. 试图用 `git push origin <branch-name>` 推送自己的修改
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并

3. 如果合并有冲突，则解决冲突，并在本地提交
4. 没有冲突或者解决掉冲突后，再用 `git push origin <brnach-name>` 推送就能成功

如果 `git pull` 提示 `no tracking information`，则说明本地分支和远程分支的连接关系没有创建，用命令 `git branch --set-upstream-to <branch-name> origin/<branch-name>`

小结

1. 查看远程库信息，使用 `git remote -v`
2. 本地新建的分支如果不推送到远程，对其他人就是不可见的
3. 从本地推送分支，使用 `git push origin <branch-name>`，如果推送失败，先用 `git pull` 抓取远程的最新提交
4. 在本地创建和远程分支对应的分支，使用 `git checkout -b <branch-name> origin/<branch-name>`
5. 建立本地分支和远程分支的关联，使用 `git branch --set-upstream <branch-name> origin/<branch-name>`
6. 从远程抓取分支，使用 `git pull`，如果有冲突，要先处理冲突

Rebase（未完）

多人在同一个分支上协作时，很容易出现冲突。即使没有冲突，后push的同学不得不先pull，在本地合并，然后才能push成功。每次合并再push后，分支变成了这样：

```
git log --graph --pretty=oneline --abbrev-commit
*   d91dd7e (HEAD -> main, origin/main) Merge branch 'dev' into main
0501 first commit
|\
| * 0f99df5 0501 first commit
| * d365280 fix bug
* | 40622e4 fix bug
|/
*   1a0ef4d conflict fixed
|\
| * bfc0568 commit to feature1
```

```
* | f346e88 main commit
|/
* 9ccefe3 new branch commit
* e493a21 index update
```

总之看上去很乱，有没有可能让Git的提交历史变成一条干净的直线？其实是可以做到的！Git有一种称为rebase的操作，有人把它翻译成“变基”🐶

实践一下，我们先在先前 `git clone` 下来的版本库中对 `dev` 分支做一些修改，然后提交到远程 `dev` 分支上

```
vim hacker.py

print("Now I change the file first")

git add .
git commit -m "hacker change file first"
git push origin dev
```

你可以多提交几次，这样看起来比较明显一些

现在，我们切回最初的本地版本库，此时这个本地版本的 `master` 比远程分支要落后几个版本了。我们对这个版本做一下修改然后提交（我就不演示产生冲突了）

```
git push origin dev
To github.com:bupt-zhenghui/learngit.git
 ! [rejected]        dev -> dev (fetch first)
error: failed to push some refs to 'git@github.com:bupt-
zhenghui/learngit.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository
pushing
hint: to the same ref. You may want to first integrate the remote
changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

很不幸，失败了，这说明有人先于我们推送了分支。按照经验，先pull一下

```
git pull origin dev
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 1), reused 5 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From github.com:bupt-zhenghui/learngit
 * branch            dev        -> FETCH_HEAD
   e364bda..5731d40  dev        -> origin/dev
Merge made by the 'recursive' strategy.
 hacker.py | 2 ++
 1 file changed, 2 insertions(+)
```

很遗憾，水平有限，中间一段没看明白，想要继续了解这里我给一个链接

小结

1. rebase操作可以把本地未push的分叉提交历史整理成直线
2. rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方

对比