

Git 项目中常用操作及项目版本开发常规流程简介

- Git 项目中常用操作及项目版本开发常规流程简介

- Part1 Git简介

- 写在前面

- 1. 什么是版本控制
 - 2. 集中式版本控制
 - 3. 分布式版本控制

- Git是什么

- Git特点

- Git与Svn的区别

- Git工作流程

- Git的几个核心概念

- Part2 Git安装

- Part3 Git常规操作流程

- 创建版本库

- 文件操作

- 版本回退

- 删除文件

- 推送到远程仓库

- Part4 分支管理

- 1. 分支简介

- 2. 创建与合并分支

- 3.解决冲突

- 4.分支开发注意事项

- 附 项目中常用的Git操作命令

Part1 Git简介

写在前面

1. 什么是版本控制

- 简介:

版本控制系统(Version Control System,简称VCS)是一种记录一个或若干文件内容变化,以便将来查阅特定版本修改情况的系统.

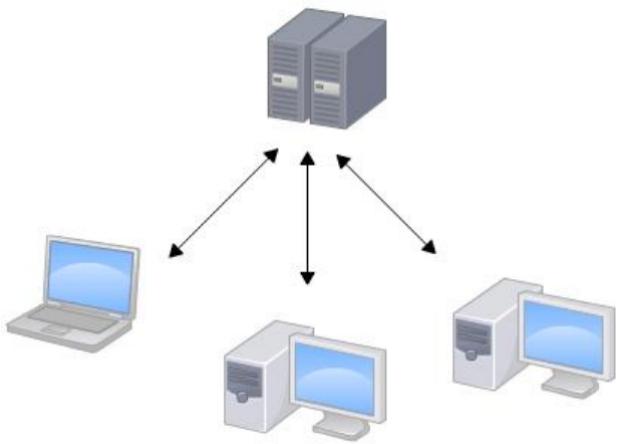
- 作用:

版本控制最主要的功能就是追踪文件的变更.它将什么时候、什么人更改了文件的什么内容等信息记录下来.每一次文件的改变,文件的版本号都将增加.

除了记录版本变更外,版本控制的另一个重要功能是并行开发.软件开发往往是多人协同作业,版本控制可以有效地解决版本的同步以及不同开发者之间的开发通信问题,提高协同开发的效率.并行开发中最常见的不同版本软件的错误(Bug)修正问题也可以通过版本控制中分支与合并的方法有效地解决.

2. 集中式版本控制

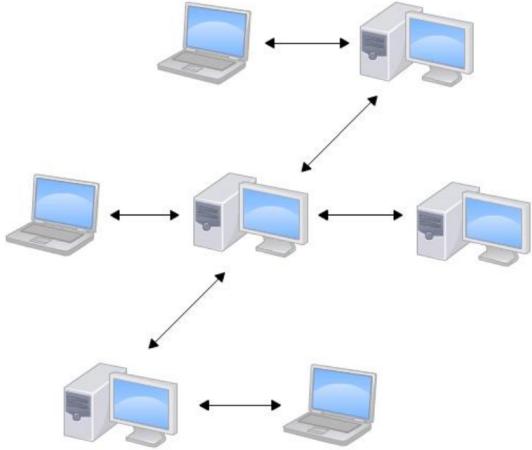
- 集中式版本控制的版本库集中存放在中央服务器,而实际工作的是自己电脑,所以要先从中央服务器取得最新的版本,然后才能工作.工作完成,再把作品内容推送到中央服务器.
- 集中式版本控制系统最大缺点就是必须联网才能工作,如果在局域网内还好,带宽够大,速度够快,可如果在互联网上,遇到网速慢的话,可能提交一个10M的文件就需要5分钟.低效.



3. 分布式版本控制

- 分布式版本控制系统没有“中央服务器”,每个人的电脑上都有一个完整的版本库,这样,工作的时候,就不需要联网,因为版本库就在自己电脑上.
- 多人协作: 你在自己电脑上改了文件A,你的同事也在他的电脑上改了文件A,这时,你们俩之间只需把各自的修改推送给对方,就可以互相看到对方的修改.

- 和集中式版本控制系统相比,分布式版本控制系统的安全性高:因为每个人电脑里都有完整的版本库,某一个人电脑故障,可从其它电脑复制一份.而集中式版本控制系统的中央服务器出问题,所有人都将无法获得版本更新.
- 实际使用分布式版本控制系统,很少在两人之间电脑上直接推送版本修改,因为可能相互间不在一个局域网内,互相访问不了,或电脑故障无法开机.因此,分布式版本控制系统通常也有一台充当“中央服务器”的电脑,但仅仅用来方便“交换”修改,没有中央服务器,仍可继续提交版本,只是相互间交换修改不方便而已.



Git是什么

Git是一款免费、开源的分布式版本控制系统,用于敏捷高效地处理任何或小或大的项目.可以有效、高速的处理从很小到非常大的项目版本管理.Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件.

Git特点

- 1.速度块、灵活、离线工作
- 2.适合分布式开发,强调个体
- 3.允许上千个并行开发的分支
4. 公共服务器压力和数据量都不会太大
5. 任意两个开发者之间可以很容易的解决冲突
6. 能高效管理类似 Linux 内核一样的超大规模项目

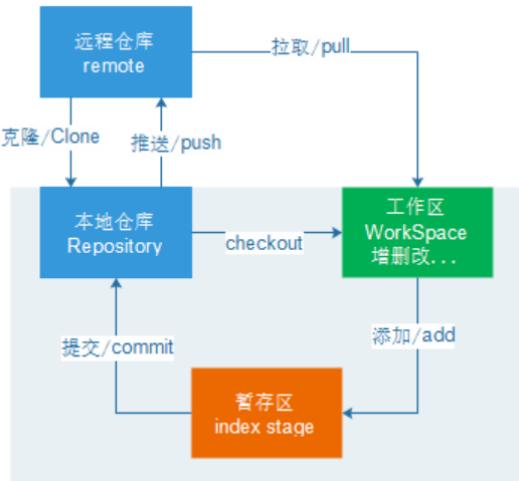
Git与Svn的区别

- SVN是集中式版本控制系统, Git是分布式版本控制系统
- Git保存的不是文件的变化或者差异,而是一系列不同时刻的文件快照
- 分支在SVN中是一个完整的目录,且目录拥有完整的实际文件.Git分支本质上仅仅是指向提交对象的可变指针
- Git的内容完整性要优于SVN:Git的内容存储使用的是SHA-1哈希算法.这能确保代码内容的完整性,确保在遇到磁盘故障和网络问题时降低对版本库的破坏
- 分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在,没有联网也可以正常工作,当有网络的时候,Git再把本地提交推送一下就完成同步!而SVN在没有联网的时候是无法操作.

Git工作流程

常规工作流程:

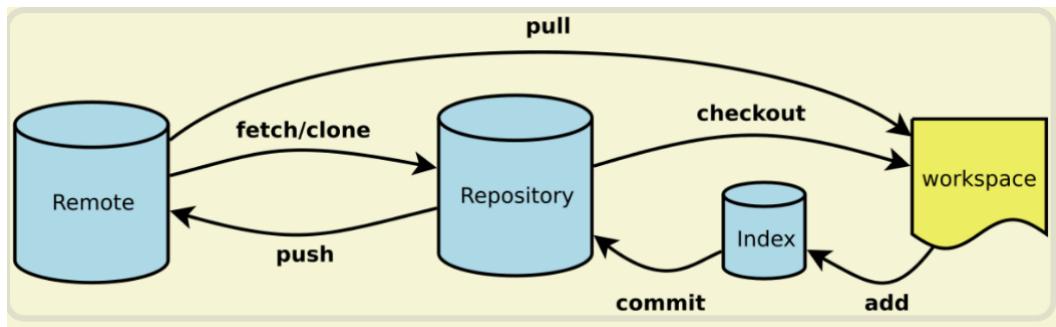
1. 从远程仓库中克隆 Git 资源作为本地仓库. `git clone <url>`
2. 从本地仓库中创建新分支或切换到指定分支,然后进行代码修改. `git switch -c <新分知名>` 或 `git switch <已有分支名>`
3. 将工作区的代码提交(add)到暂存区. `git add <file>...`
4. 将修改提交(commit)到本地仓库的当前分支;本地仓库中保存各个历史版本的修改记录. `git commit -m 'comment' [file]...`
5. 在需要和团队成员共享代码时,将提交到本地仓库的代码push到远程仓库. `git push origin <要推送的本地分支名>`



Git的几个核心概念

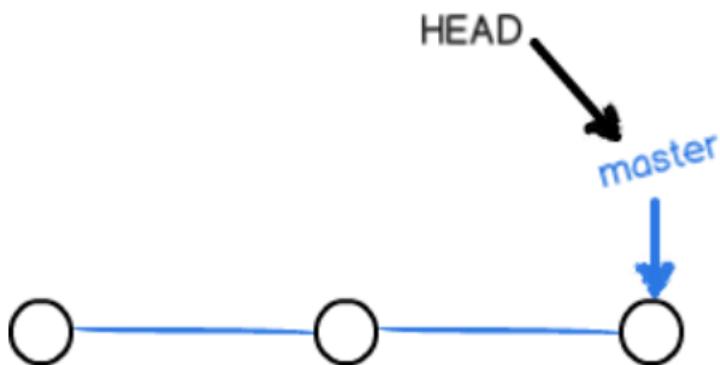
工作区、暂存区、版本库、远程仓库

- **Workspace:** 工作区,实际操作/存放项目代码的地方
- **Remote:** 远程仓库,托管代码的服务器,用作项目组成员间数据交换
- **Repository:** 本地仓库区(或本地版本库),本地存放数据的位置,保存提交的所有历史版本数据
- **Index/Stage:** 暂存区,用于临时存放文件的改动.事实上它只是一个文件,保存改动的文件、即将提交的文件列表信息

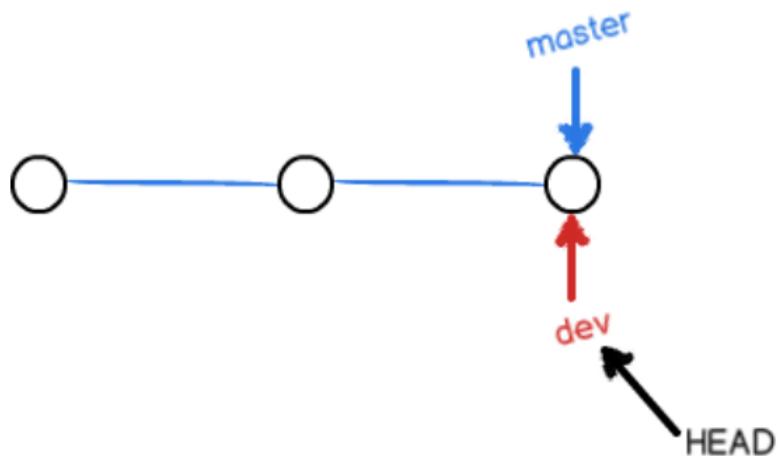


常规分支操作流程

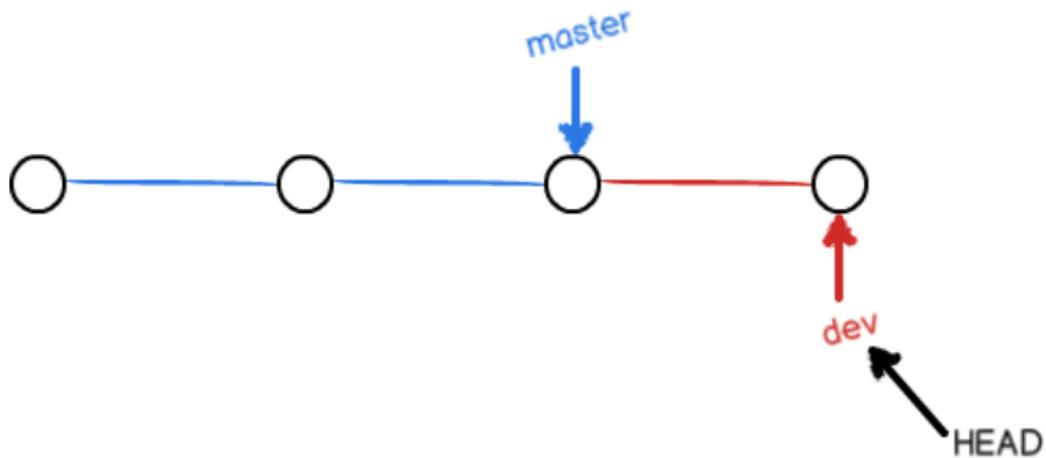
- 每次提交Git都把它们串成一条时间线,这条时间线就是一个分支.所有版本控制项目,都有一条分支,在Git里这个分支叫主分支,即master分支.
Git中,HEAD标识符指向分支名,如master,分支名指向某一次的提交
- 项目初始,master分支是一条线,Git用master指向最新的提交,用HEAD指向master,确定当前分支,以及当前分支的提交点
- 每次提交,master分支都会向前移动一步,多次提交,master分支的线也越来越长.



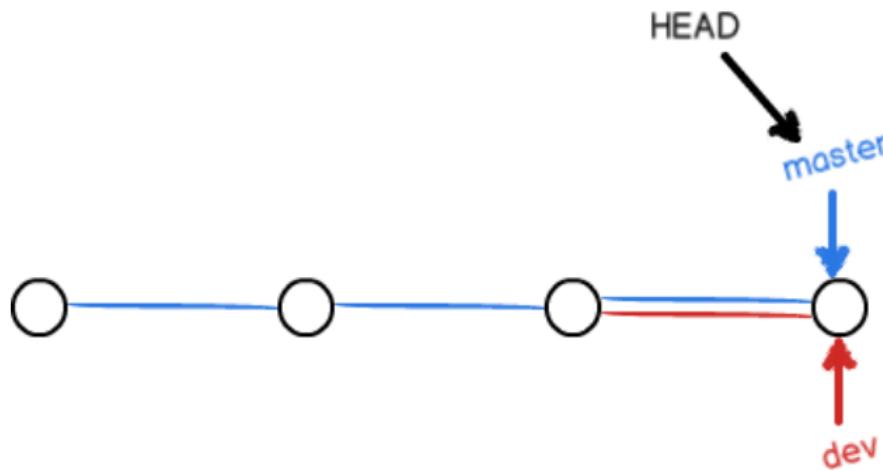
- 当创建新分支,如dev时,Git新建了一个指针叫dev,指向创建时刻master的提交,再把HEAD指向dev,表示当前分支在dev上:



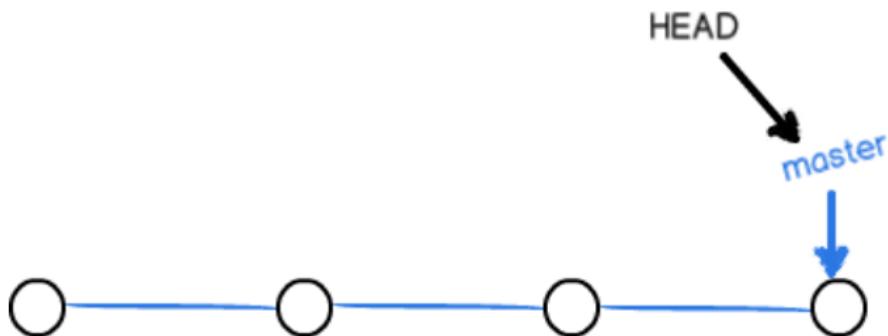
- Git创建一个分支很快: 只需增加一个dev指针,改变HEAD指向,无需操作工作区的文件!
- 不过切换到dev分支后,对工作区修改和提交就是针对dev分支了,如新提交一次,dev指针就往前移一步,而master指针指向不变



- 假如在dev上的工作完成,可把dev合并到master上:最简单的方法,就是直接把master指向dev的当前提交(即Fast-Forward合并策略),完成合并:



- Git合并分支很快! 只需改下指针指向位置,工作区内容不变.合并完分支后,可以删除dev分支:把dev指针给删掉,保留master分支:



Part2 Git安装

• Linux

- 输入git,查看系统是否已安装Git:

```
$ git  
The program 'git' is currently not installed. You can install it by typing:  
sudo apt-get install git
```

如上Git没有安装,提示如何安装Git:

```
sudo apt install git
```

• Mac

- 如果尚未安装Git,或者已安装的Git版本过低,可以去Git官网<https://git-scm.com>进行安装

The screenshot shows the main navigation menu of the git-scm.com website. The menu items include 'About', 'Documentation', 'Downloads', and 'Community'. A red arrow points to the 'Downloads' link, which is highlighted in red. Below the menu, there are four circular icons: a gear icon for 'About', an open book icon for 'Documentation', a download arrow icon for 'Downloads', and a speech bubble icon for 'Community'. The 'Downloads' section contains text about GUI clients and binary releases for major platforms.

The screenshot shows the 'Downloads' page of the git-scm.com website. On the left, there is a sidebar with links to 'Documentation', 'Downloads', and 'Community'. The main content area has a large title 'Downloads' and three download links: 'macOS' (with an Apple logo), 'Windows' (with a Windows logo), and 'Linux/Unix' (with a Linux logo). A red arrow points to the 'macOS' link. Below the download links, there is a note about older releases and the GitHub repository.

The entire [Pro Git book](#) written by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

Download for macOS

There are several options for installing Git on macOS. Note that any non-source distributions are provided by third parties, and may not be up to date with the latest source release.

Homebrew

Install [homebrew](#) if you don't already have it, then:

```
$ brew install git
```

MacPorts

Install [MacPorts](#) if you don't already have it, then:

```
$ sudo port install git
```

Xcode

Apple ships a binary package of Git with [Xcode](#).

Binary installer

Tim Harper provides an [installer](#) for Git. The latest version is [2.33.0](#), which was released 6 months ago, on [2021-08-30](#).

Building from Source

If you prefer to build from source, you can find tarballs [on kernel.org](#). The latest version is [2.35.1](#).

Homebrew安装完成后,执行brew install git即可安装最新版本Git

```
# 安装git  
brew install git
```

安装完成后,需配置用户信息

```
git config --global user.name "rocky"  
git config --global user.email "3xxxxx5@qq.com"
```

Notice:

1. --global参数,表示这台机器上的所有的git仓库都会使用这个配置,当然也可对某个仓库指定不同的用户名和邮箱,更多参数我们也可以通过git config提示查看,还可以使用git config --list或git config -l来查看已经配置的信息

Part3 Git常规操作流程

创建版本库

- 什么是版本库

版本库(repository),可以简单理解成一个目录,这个目录里面的所有文件都可以被Git管理起来,每个文件的修改、删除,Git都能跟踪,以便任何时刻都可以追踪历史,或者在将来某个时刻可还原到某一历史版本.

- 创建版本库

- 1. 创建一个空目录

```
$ mkdir learngit // 创建learning目录  
$ cd learngit // 进入learngit目录  
$ pwd // 打印当前所处目录  
/Users/hy/learngit
```

- 2. 通过git init初始化一个本地仓库:把当前目录变成Git可以管理的仓库

```
$ git init  
Initialized empty Git repository in /Users/hy/learngit/.git/
```

这样Git仓库就建好,且命令输出提示:所建的是一个空仓库(empty Git repository).

通过 ls -a 命令,可发现当前目录下多一个.git的目录,这个目录是Git来跟踪管理版本库的,不可手动修改这个目录里面的文件,否则容易把Git仓库破坏.

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ ls -a  
. .. .git readme.txt
```

- 3. 文件添加到版本库

在刚创建的learngit目录下,编写一个readme.txt文件,内容如下:

```
Test: Git is a version control system.
```

- 把文件添加到暂存区

```
git add readme.txt
```

- 用命令git commit把文件提交到本地仓库:

```
$ git commit -m "commit a readme file"
[master (root-commit) eaadf4e] commit a readme file
 1 file changed, 1 insertions(+)
 create mode 100644 readme.txt
```

-m 后面输入的是本次提交的说明,可以输入任意内容,方便从历史记录里快速定位需要的历史版本

git commit命令执行成功后提示:

- 1 insertions:插入了一行内容
- 1 file changed:1个文件被改动(新添加的readme.txt文件)

文件操作

- 1. 修改文件内容

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

运行 git status 命令查看结果:

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

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 status命令可以掌握仓库当前的状态,上面的命令输出可知:

- readme.txt被修改过,但还没提交修改
 - 可用git add ...命令,将修改的文件提交到暂存区
 - 可用git restore ... 丢弃工作区已做的文件修改
- 2. 提交到暂存区

```
$ git add readme.txt
```

运行 git status 查看当前仓库的状态

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git add readme.txt
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ 
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ 
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ 
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

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

git status命令可知:

readme.txt可被提交

可用git restore --staged ...命令,撤销本次文件的暂存操作

- 3. 提交到本地仓库

```
git commit -m 'commit to local repository'
```

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git commit -m 'commit to local repository'
[master 42c8835] commit to local repository
 1 file changed, 1 insertion(+)
```

运行 git status 查看当前仓库的状态

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

git status命令可知:

1. 当前没有需要提交的修改,而且工作目录干净(working tree clean)

版本回退

- 1. 查看版本历史记录: git log

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git log
commit 42c8835af4e6c920c624129f4f111bd9a72ab214 (HEAD -> master)
Author: hzl <huangzhiliang@chanmama.com>
Date:   Sat May 14 17:50:40 2022 +0800

  commit to local repository

commit e6e140ac04611e4b320db4aca217380a927a068a
Merge: 8c78f2d 8d0a673
Author: hzl <huangzhiliang@chanmama.com>
Date:   Fri May 13 17:30:35 2022 +0800

  conflict fixed
```

git log命令可知:

1. 显示从最近到最远的提交日志,可以看到2次提交,最近的一次是commit to local repository
2. 一大串类似42c8835a...的是commit id(版本号),和SVN不一样,Git的commit id不是1,2,3.....递增的数字,而是一个SHA1计算出来的一个非常大的数字,用十六进制表示.因为Git是分布式的版本控制系统,多人在同一个版本库里工作,如用1,2,3.....作为版本号,则产生冲突

- 2. 版本回退

Git中,用HEAD指向当前所在版本,即最新提交42c8835a.....,上一个版本就是HEAD^,上上一个版本就是HEAD^^,往上100个版本可写HEAD~100

回退到上一个版本:

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git reset --hard HEAD^
HEAD is now at e6e140a conflict fixed
```

readme.txt的内容回退到 e6e140ac046...版本

```
≡ readme.txt
1 Creating a new branch is quick AND simple.
2
```

再查看版本历史记录: git log

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git log
commit e6e140ac04611e4b320db4aca217380a927a068a (HEAD -> master)
Merge: 8c78f2d 8d0a673
Author: hzl <huangzhiliang@chanmama.com>
Date:   Fri May 13 17:30:35 2022 +0800

conflict fixed
```

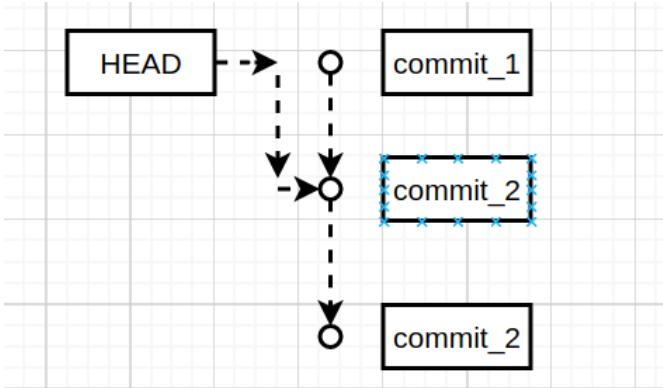
42c8835a...版本不再显示.可通过 git reflog 查看操作过的历史命令

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git reflog
e6e140a (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
42c8835 HEAD@{1}: commit: commit to local repository
```

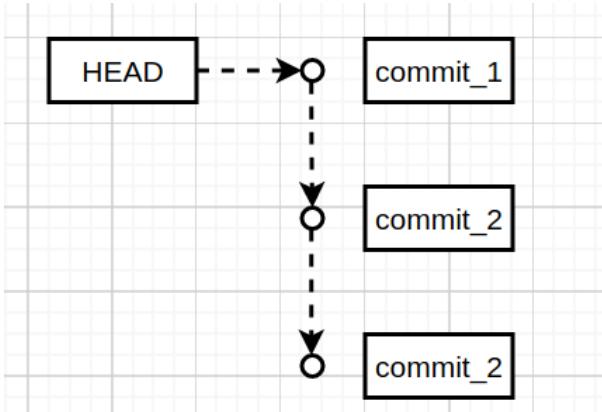
可找回回退前的版本的版本号42c8835a...,输入 git reset --hard 42c8835a ,可回到目标版本.这里相当于撤销刚刚回退操作.

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git reset --hard 42c8835a
HEAD is now at 42c8835 commit to local repository
```

Git的版本回退速度非常快:HEAD指针重新指向目标版本即可,无需涉及文件内容相关的操作



回退到commit_1



删除文件

一般情况下,通常直接在文件管理器中把没用的文件删除,或者用rm命令删:

```
$ rm test.txt
```

这时,Git知道文件被删除,工作区和版本库内容不一致,git status命令会输出哪些文件被删除,及撤销删除操作的命令:

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

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:    readme.txt

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

1.从版本库中删除改文件

```
$ git rm readme.txt  
rm 'readme.txt'  
  
$ git add . && git commit -m "remove readme.txt"  
[master d46f35e] remove readme.txt  
 1 file changed, 1 deletion(-)  
 delete mode 100644 test.txt
```

现在,文件就从版本库中被删除

2.另一种情况是误删,因为版本库里存有被删的文件,所以可以把误删的文件恢复到指定版本里的文件:

```
$ git restore readme.txt  
或  
$ git checkout -- readme.txt // 最新一次保存到暂存区的readme.txt
```

推送到远程仓库

- 代码提交到本地仓库后,可以推送到远程仓库备份,防止因本地主机硬盘损坏,代码丢失.同时,也便于他人从远程仓库克隆/更新代码到本地主机,查看/修改代码,也允许其它开发者推送各自的代码到远程仓库,通过远程仓库协作开发,分享开发经验

1.初始代码从远程仓库克隆

从已有的代码仓库克隆代码到本地,更新/修改后,暂存(`git add`)、提交(`git commit`)到本地仓库,最后需要推送到远程仓库,可用 `git push [origin] [当前分支]` 推送当前分支的修改到远程仓库

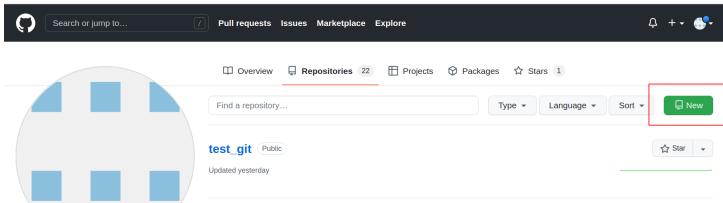
```
$ git clone git@github.com:xxxxxx/git_test.git  
。。。。。// 代码新增/修改  
git add . // 暂存  
git commit -m 'comment' // 提交本地仓库  
git push origin <待提交的分支名>
```

git支持多种协议,默认的git://使用ssh,也可以使用https等其他协议.使用https除了速度慢以外,还有个最大的麻烦是每次推送都必须输入口令.一般使用ssh协议同步代码

2.新建项目代码仓库

创建本地主机的代码仓库后,需要和远程仓库创建关联,才能把本地代码推送到远程仓库备份保存,与其它人协作开发

- 在Gitlab/Github/阿里云等代码托管网站建立一个空仓库



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * Repository name *

Great repository names are short and memorable. Need inspiration? How about improved-octo-lamp?

Description (optional)

* Public Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file

通过 `git remote add origin <远程仓库地址>` 把一个已有的本地仓库与之关联,然后通过 `git push origin master`,把本地仓库的内容推送到远程(Eg. GitHub)仓库.Eg.:

```
$ git remote add origin git@github.com:XXXXXX/git_test.git // 把远程仓库地址换成自有的仓库地
```

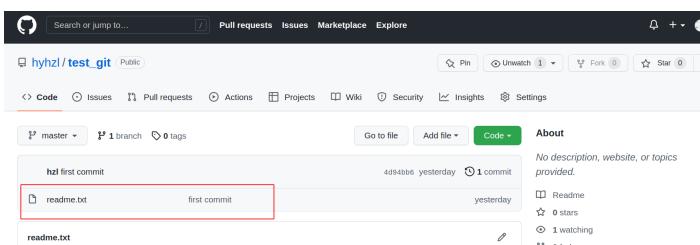
远程库的名字是origin,是Git默认的叫法,也可以改成别的(一般不做修改)

```
$ git push -u origin master
```

Notice:

首次推送,由于远程库是空的,第一次推送master分支时,需加-u参数,Git不但会把本地的master分支内容推送到远程的master分支,还会把本地的master分支和远程的master分支进行关联,以后的推送或者拉取时就可以简化命令,无需-u参数.

推送成功,GitHub页面可看到远程库内容和本地仓库一样



Part4 分支管理

1. 分支简介

- 分支指从主线上分离出来进行另外操作,解决临时需求,而又不影响主线,主线可以继续开发,类比父子线程.最后分支合并到主线上,而分支的任务完成后可以删除.
- 几乎所有的版本控制系统都以某种形式支持分支.在很多版本控制系统中,这是一个略微低效的过程,常常需要完全创建一个源代码目录的副本.对于大项目来说,这样的过程会耗费很多时间.
- git的分支功能强大,不需要将所有数据进行复制,创建目录副本,只要重新创建一个分支的指针,指向你需要的某一提交commit点,,那么新分支的指针就会指向你最新提交的这个commit对象.原先分支的指针则指向原先开发位置.
- 在哪个分支开发,HEAD就指向那个分支的最新提交对象commit.

2. 创建与合并分支

1. 创建分支——整体流程

前文叙述可知,每次提交,git都串成一条时间线,这条时间线就是一个分支.截止目前,只有一条时间线,在git里,这个分支叫主分支,即master分支.

标识符HEAD严格来说不是指向提交,而是指向master,master指向提交.

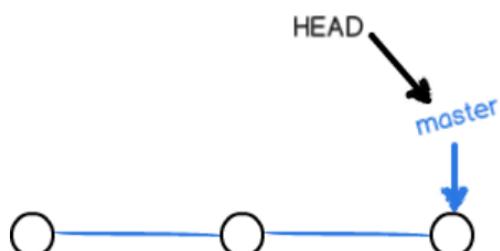
一开始,master分支是一条线,git用master指向最新的提交,再用HEAD指向master,确定当前分支,以及当前分支的提交点.

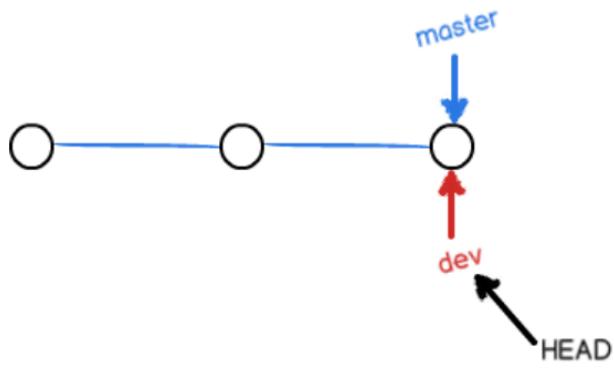
每次提交,master分支都会向前移动一步.随着持续提交,master分支的线越来越长.

当创建新分支,如dev时,git新建一个指针叫dev,指向master相同的提交,再把HEAD指向dev,表示当前分支在dev上:

git创建一个分支很快:除了增加一个dev指针,改变HEAD指向,工作区的文件没有任何变化

创建并切换分支后,对工作区的修改和提交就是针对dev分支,新提交一次,dev指针就往前移一步,而master指针指向位置不变:

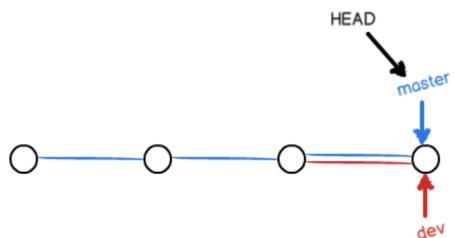
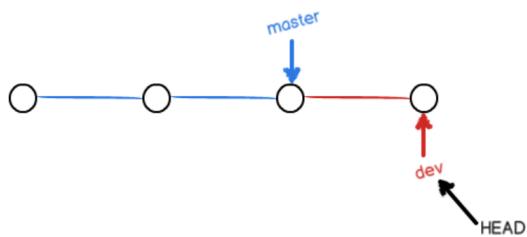




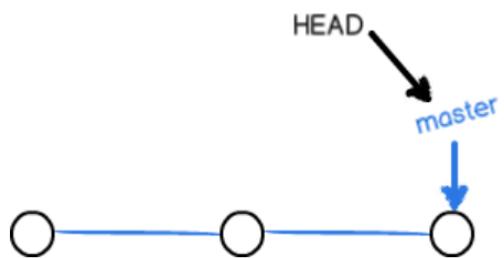
合并分支

假如在dev上的工作完成,可以把dev合并到master上.最简单的情形,就是直接把master指向dev的当前提交,就完成了合并(在master没有新提交的情况下,直接移动指针).

可见Git合并分支也很快:改变指针指向,工作区内容不变 !



合并完分支,dev分支可以删除.删除dev分支就是把dev指针给删掉,剩下一条master分支:



2. 创建分支——实际操作

创建dev分支,同时切换到dev分支.

```
$ git switch -c dev
Switched to a new branch 'dev'
```

`git switch` 命令加上-c参数表示创建并切换,相当于以下两条命令:

```
$ git branch dev
$ git switch dev
Switched to branch 'dev'
```

用 `git branch` 命令查看当前分支.命令会列出本地所有分支,当前分支前面会标一个*号.

```
$ git branch
* dev
  master
```

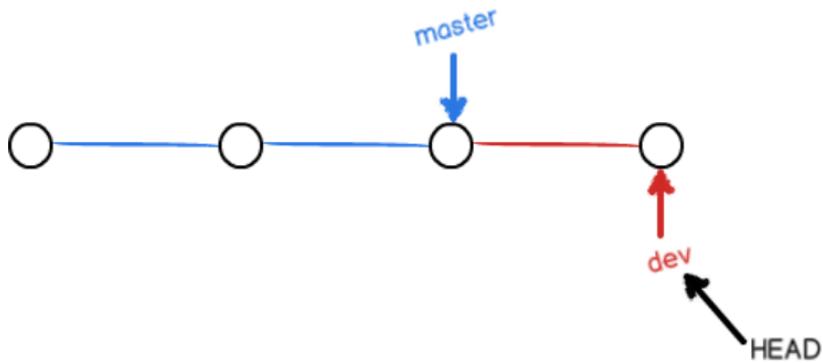
可以在dev分支上正常提交,比如对readme.txt做个修改,加上一行:'Creating a new branch is quick.'然后提交:

```
$ git add readme.txt
$ git commit -m "branch test"
[dev b17d20e] branch test
 1 file changed, 1 insertion(+)
```

dev分支完成提交,可以切换回master分支:

```
$ git switch master
Switched to branch 'master'
```

切换回master分支后,再查看readme.txt文件,刚才添加的内容不见!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变:



把dev分支的代码合并到master分支上:

```
$ git merge dev
Updating d46f35e..b17d20e
Fast-forward
 README.txt | 1 +
 1 file changed, 1 insertion(+)
```

git merge命令用于合并指定分支到当前分支.合并后查看readme.txt内容,和dev分支的最新提交是完全一样的.

合并完成后,可以删除dev分支:

```
$ git branch -d dev
Deleted branch dev (was b17d20e).
```

删除后,查看branch,只剩下master分支:

```
$ git branch
* master
```

3. 推送分支

查看远程仓库信息:

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git remote -v
origin  git@github.com:hyhzl/test_git.git (fetch)
origin  git@github.com:hyhzl/test_git.git (push)
```

输出可以抓取和推送的origin的地址.如果没有推送权限,就看不到push的地址.

推送分支,就是把该分支上的所有本地提交推送到远程库.推送时,指定本地分支,Git就把该分支推送到远程库对应的远程分支上:

```
$ git push origin <待推送分支名>
```

3.解决冲突

合并冲突,简单来说,就是项目中同时有多人在协作开发,针对其中一部分文件同时有多人进行了修改,此时git不能执行快速合并,就会发生合并冲突,这时需要手动解决有冲突的文件

新建并切换到dev分支

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git switch -c dev
Switched to a new branch 'dev'
```

修改readme.txt最后一行,改为: "Creating a new branch is quick AND simple.".在dev分支上提交

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git add readme.txt
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git commit -m 'modidy readme.txt on dev branch'
[dev 8d0a673] modidy readme.txt on dev branch
 1 file changed, 1 insertion(+), 1 deletion(-)
```

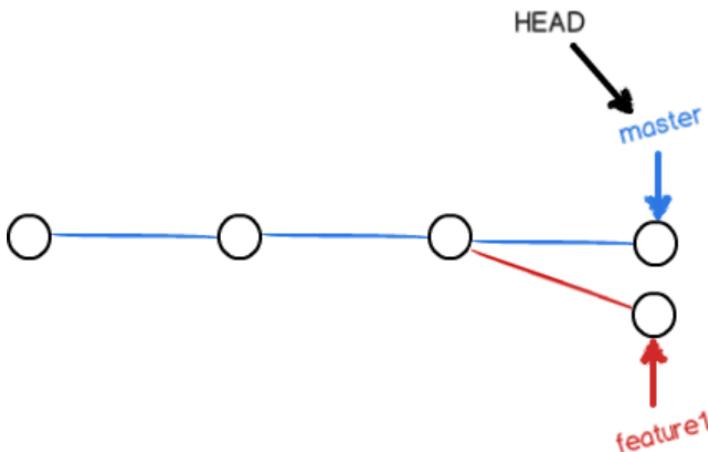
切换到master分支:

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git switch master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

在master分支上把readme.txt文件最后一行改为:"Creating a new branch is quick & simple.".提交:

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git add readme.txt
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git commit -m '& simple'
[master 8c78f2d] & simple
 1 file changed, 1 insertion(+), 1 deletion(-)
```

现在,master分支和dev分支各自都分别有新的提交:



修改到同一行的情况下,Git无法执行“快速合并”,只能试图把各自的修改合并起来,产生合并冲突。Git提示,readme.txt文件存在冲突,必须手动解决冲突后再提交.git status也可以告诉我们冲突的文件

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git merge dev
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)

You have unmerged paths.
(use "git commit" to 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")
```

查看readme.txt文件内容:

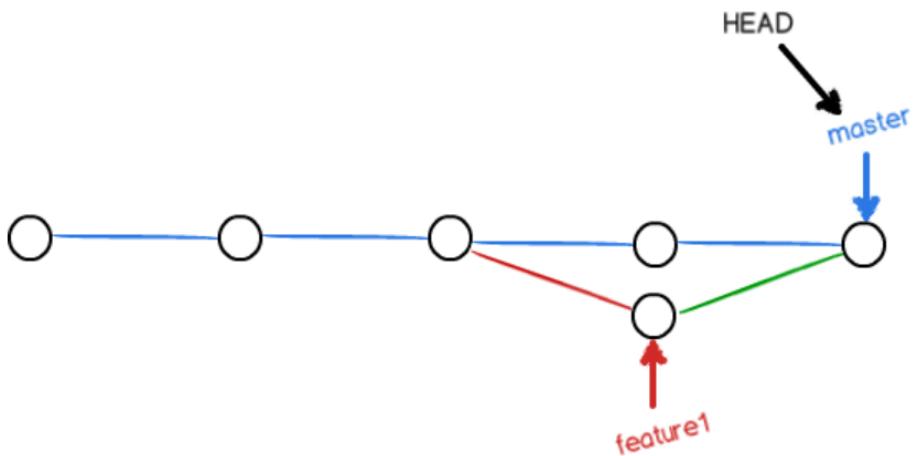
```
readme.txt
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
1 | <<<<< HEAD (Current Change)
2 | Creating a new branch is quick & simple.
3 | =====
4 | Creating a new branch is quick AND simple.
5 | >>>>> dev (Incoming Change)
6 |
```

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

Creating a new branch is quick and simple.

```
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git add readme.txt
hy@hy-21-04:~/work/jxProject/src/For_newColleague/Git/test_git$ git commit -m 'conflict fixed'
[master e6e140a] conflict fixed
```

现在,master分支和dev分支版本线变成了下图所示:



用带参数的git log也可以看到分支的合并情况:

```
hy@hy-21-04:~/work/jxProject/src/For_newColleage/Git/test_git$ git log --graph --pretty=oneline --abbrev-commit
*   e6e140a (HEAD -> master) conflict fixed
|\ \
| * 8d0a673 (dev) modidy readme.txt on dev branch
* | 8c78f2d & simple
|/
* 4d94bb6 (origin/master) first commit
```

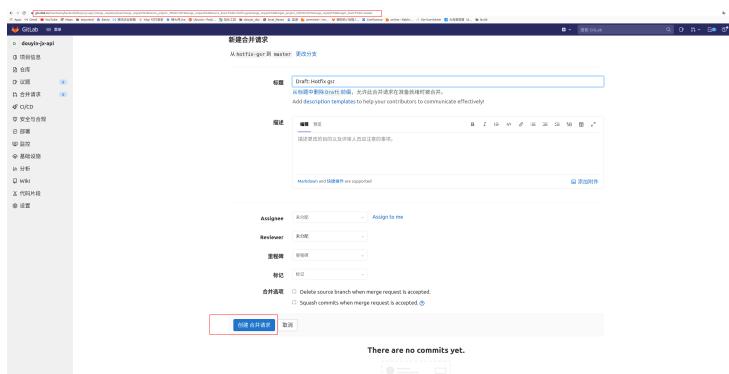
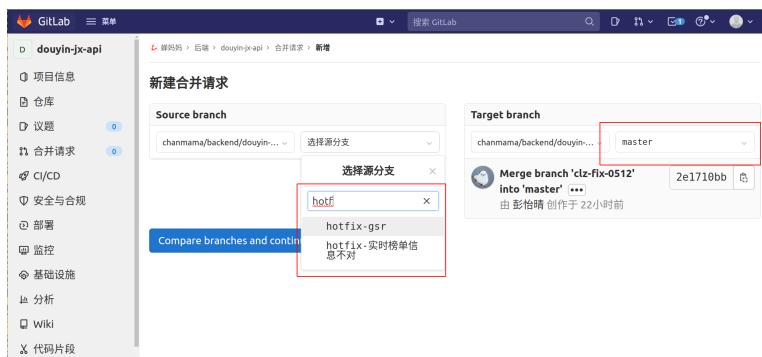
Notice:

- Git无法自动合并分支时,就必须手动解决冲突.解决冲突后,再提交,完成合并.
- 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.
- 用git log --graph命令可以看到分支合并图,--pretty 可选择输出格式,--abbrev-commit 简化commit-id输出

4. 分支开发注意事项

1. 紧急修复生产环境(线上的bug)

1. 切换到master分支(git switch master)
2. 拉取master分支最新的代码(git pull). 这步一定要记得执行, 否则将基于本地仓库的旧master代码修复Bug, 可能错上加错.
3. 创建临时分支, 在临时分支上修复生产Bug. 注意分支命名规范: 分支名需以hotfix或fix为前缀. 项目中统一标准.
4. 修复Bug, 提交Bug分支代码到远程仓库
5. 公司项目代码托管在GitLab, 分支代码合并到master需要有相关权限(一般项目组长有合并权限), 需到项目代码的GitLab上提交合并请求(即合并到master的请求), 然后将产生的'合并请求链接'发送给权限相关人员, 进行代码review、合并操作.



2. 功能分支名

开发新需求的分支称做功能分支, 分支名需以dev或feature为前缀. 项目中统一标准.

3. Bug修复分支名

新创建的Bug修复分支名, 一般以fix为前缀. 项目中统一标准.

4. 其它分支名

临时需求, 需开辟新分支开发的, 分支名要做到简明扼要, 见名知义. 如导出功能需求: 一般以export为前缀.

5. 前端对接/测试测试代码

在功能开发分支开发好后, 需将代码合并到测试分支(Eg.: dev. 以项目统一为准), 以便前端人员对接, 测试同事进行测试.

1. 提交功能分支代码到远程仓库
2. 切换到dev分支, 更新dev代码到最新(git pull). 这步一定要记得执行, 否则将基于本地仓库的旧dev代码进行合并, 导致异常
3. 合并新功能分支到测试分支, 并将合并代码推送到远程代码仓库.

6. 开发分支合并最新master代码

在功能分支中开发新需求功能时, 有时需要将master代码重新合并回开发分支:

- 在有新需求时, 往往需要拉取最新的master线上生产代码, 然后以最新的master代码为基础, 新开一条功能需求分支, 进行新需求的开发.
- 在开发过程中, 如碰到线上紧急Bug, 需紧急修复, 并推送到生产环境(即合并到master)后, 往往可以将修复后最新的master代码重新合并回正在开发的功能分支, 修复功能分支代码中相同Bug.

附 项目中常用的Git操作命令

git log 查看提交历史

git reflog 查看历史操作命令

git branch 查看当前所处分支

git clone <url> 克隆远程仓库

git pull 拉取当前分支的最新代码

git branch -a 查看本地仓库所有分支

git push 推送修改/新增的代码到远程仓库

git branch -D <分支名> 强制删除指定分支

git branch -d <分支名> 删除已合并过代码的分支

git add . 提交所有修改/新增的代码到本地暂存区

git switch branch_name 切换到branch_name分支

git restore <file>... 撤销本地工作区文件的修改内容

git reset --soft <commit_id> 回退版本,保留版本差异文件

git reset --hard <commit_id> 回退版本,丢弃版本差异文件

git restore --staged <file>... 撤销提交到本地暂存区的文件

git commit -m '说明字符串' 提交修改/新增的代码到本地仓库

git switch -c branch_name 以当前分支为基础,新开一条分支,并切换到新的分支上

git push -u origin <new_branch> 新开的分支, '第一次' 推送到远端仓库(origin: 字符串常量,指向远程仓库).后续推送可缩简成 git push ,即推送当前分支修改到远端.