

Git:分布式版本控制系统

<https://www.liaoxuefeng.com/wiki/896043488029600/896067008724000>

[git学习（廖雪峰教程）.note](#)

• Git简介

Git属于分布式版本控制系统,分布式版本控制系统没有“中央服务器”，每个人的电脑上都是一个完整的版本库,开发者通过克隆（git clone），在本地机器上拷贝一个完整的Git仓库。在本机修改文件后把修改推送出去,就可以看到修改内容;

Git的功能特性：

- 1、从服务器上克隆完整的Git仓库（包括代码和版本信息）到单机上。
- 2、在自己的机器上根据不同的开发目的，创建分支，修改代码。
- 3、在单机上自己创建的分支上提交代码。
- 4、在单机上合并分支。
- 5、把服务器上最新版的代码fetch下来，然后跟自己的主分支合并。
- 6、生成补丁（patch），把补丁发送给主开发者。
- 7、看主开发者的反馈，如果主开发者发现两个一般开发者之间有冲突（他们之间可以合作解决的冲突），就会要求他们先解决冲突，然后再由其中一个人提交。如果主开发者可以自己解决，或者没有冲突，就通过。
- 8、一般开发者之间解决冲突的方法，开发者之间可以使用pull 命令解决冲突，解决完冲突之后再向主开发者提交补丁。

<https://baike.baidu.com/item/GIT/12647237?fr=aladdin#7>

• 创建本地仓库

Git 的安装

1. 从官网下载安装程序进行安装
2. 安装后在开始菜单找到Git Bash命令窗口
3. 使用命令指定一个用户名与邮箱

```
1 #设置用户名与邮箱
2 $ git config --global user.name "用户名"
3 $ git config --global user.email "邮箱@xx.com"
4 #查看用户名或邮箱
5 $ git config user.name
6 $ git config user.email
7 //git config 命令:允许获得和设置配置变量,这些变量可以控制Git的外观和操作的各个方面
8 //--global 参数:表示这台机器上所有的Git仓库都会使用这个配置
9
```

创建版本库

版本库又名仓库，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，任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

1. 选择电脑上一个位置创建一个空目录,确保目录路径不要包含中文

- 1 #例如,我在E盘目录中创建了文件夹repository作为版本库
- 2 #电脑路径为 E:\ConfigFile\repository

2.使用命令将当前git路径转移至目标路径

- 1 \$ cd /e/ConfigFile/repository
- 2 #cd 命令:切换目录

3.通过git init命令将目录变成可管理的仓库

- 1 \$git init
- 2 //创建完成后将会提示此句
- 3 Initialized empty Git repository in E:/ConfigFile/repository/.git/
- 4
- 5 再次输入 ls -ah命令查看是否有.git目录
- 6 \$ ls -ah
- 7 ../../.git/
- 8 #git init 命令:初始化本地仓库，在当前目录下生成 .git 文件夹

添加版本

1.将文件放在版本库目录或子目录下(必须在版本库中)

2.使用 git add 把文件或文件夹添加到仓库(添加时,当前git命令行位置需要在仓库目录下)

- 1 1.根目录位置添加文件或文件夹
- 2 \$ git add 文件名称或文件夹名称(例如添加一个readme.txt文件)
- 3 2.在版本库内部文件夹中的文件需要先进入目录再进行添加
- 4 \$ cd 文件夹
- 5 \$ git add 文件名称

3.使用git commit 命令将文件提交到仓库

- 1 \$ git commit -m "提交说明"
- 2 提交成功后将会提示
- 3 [master (root-commit) eaadf4e] wrote a readme file
- 4 1 file changed, 2 insertions(+)
- 5 create mode 100644 readme.txt
- 6 //1 file changed: 1个文件被改动(我们新添加的readme.txt文件);
- 7 //2 insertions: 插入了两行内容(readme.txt有两行内容)。

提交时不加说明进去VIM编译器后使用Esc键退出输入状态，然后按Shift+";",再输入q!或wq!退出

Git 添加文件一共需要 **add** 与 **commit** 两步,add多次添加不同的文件,commit可以一次提交很多文件

```
1 $ git add file1.txt
2 $ git add file2.txt file3.txt
3 $ git commit -m "add 3 files."
```

- **文件操作**

修改文件

1.文件添加至版本库后,对文件进行修改后,需要重新提交版本(版本库图标将会发生改变)

运行**git status**命令查看仓库当前的状态

```
1 $ git status
2 //提示信息
3 On branch master
4 Changes not staged for commit:
5   (use "git add <file>..." to update what will be committed)
6   (use "git checkout -- <file>..." to discard changes in working
   directory)
7
8   modified:   文件名称
9 no changes added to commit (use "git add" and/or "git commit -a")
10 #上面的命令输出告诉我们,某文件被修改过了,但还没有准备提交的修改
```

2.运行**git diff**命令可以查看具体修改的内容

```
1 $ git diff readme.txt
2 //提示内容
3 diff --git a/readme.txt b/readme.txt
4 index 013b5bc..0c44e87 100644
5 --- a/readme.txt
6 +++ b/readme.txt
7 @@ -1,2 +1,2 @@
8 -Git is a distributed version control system.//修改前内容
9 +Git is a distributed admin version control system.//修改后内容
10  Git is free software.
11 \ No newline at end of file
12
```

3.重新使用add 与 commit 重新提交文件

```
1 添加文件之后使用git status命令将会提示要提交的文件
```

版本回退

git log 命令:查看版本详细修改信息,显示从最近到最远的提交日志

```
1 $ git log
2 commit da8211e54b581b133987b21950af56970367c03f //版本号
3 Author: sakura <1191178938@qq.com> //用户
4 Date: Wed Sep 4 11:37:49 2019 +0800 //时间
5
6  readme 0.3 //版本说明
7
8 commit ed9e90ccb5b2e463f309d5a8f38d5d176404140e
9 Author: sakura <1191178938@qq.com>
10 Date: Wed Sep 4 11:17:26 2019 +0800
11
12  readme 0.2
13 ...
14
```

git log 命令使用--pretty=oneline命令可以查看简化后的输出信息

```
1 $ git log --pretty=oneline
2 da8211e54b581b133987b21950af56970367c03f  readme 0.3
3 ed9e90ccb5b2e463f309d5a8f38d5d176404140e  readme 0.2
4 34fc6ef350a2ef1e59b587a144943e8d64415c37  readme3
5 32c70ed8e2f23f9449d3c0e4649e0be5b616926a  2019-9-4 10:58:04 0.1
6 691a87509c2da902e4f52f8d2b5a7f28569bb3bf  2019-9-4 10:47:23 0.1
7 #版本号 版本说明
```

git reset命令实现版本跳转

```
1 #1. 前往上一个版本
2 //在Git中HEAD代表当前版本(最新的提交),上个版本为HEAD^(当前版本^),上上个为HEAD^^或HEAD~2
3 $ git reset --head HEAD^
4 //提示信息 HEAD is now at 版本号 版本说明
5 #2. 前往指定版本
6 //版本号不必全部填写,只需填填写前几位即可
7 //例如da8211e54b581b133987b21950af56970367c03f > --hard da8211e
8 $ git reset --hard 版本号
9 //提示信息 HEAD is now at 版本号 版本说明
```

git reflog命令可以用来记录每一次操作命令

```
1 $ git reflog
2 da8211e HEAD@{0}: reset: moving to da8211e5 //版本修改了四次
3 34fc6ef HEAD@{1}: reset: moving to HEAD~2
4 da8211e HEAD@{2}: reset: moving to da8211e5
5 ed9e90c HEAD@{3}: reset: moving to HEAD^
```

```

6 da8211e HEAD@{4}: commit: readme 0.3 //提交了版本0.3,提交了5次
7 ed9e90c HEAD@{5}: commit: readme 0.2 //提交了版本0.2
8 34fc6ef HEAD@{6}: commit: readme3 //...
9 32c70ed HEAD@{7}: commit: 2019-9-4 10:58:04 0.1
10 691a875 HEAD@{8}: commit (initial): 2019-9-4 10:47:23 0.1
11 #版本号 操作 说明

```

工作区与暂存区

工作区:电脑中的仓库文件夹,工作区的隐藏目录.git属于Git版本库,不算是工作区,版本库存储了许

多东西,其中包括:1.称为**stage**(或**index**)的**暂存区**,

2.Git自动创建的第一个分支**master**,

3.执行master的指针**HEAD**

用**git add**把文件添加进去,实际上就是把文件修改添加到暂存区;

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。

暂存区:内存缓冲区, add就类似于先在这个缓冲区里填入一些数据,然后commit就类似于一次性

缓冲区里的数据写入硬盘

```

1 $ git diff #是工作区(work dict)和暂存区(stage)的比较
2 $ git diff --cached #是暂存区(stage)和分支(master)的比较

```

每次修改,如果不用**git add**到暂存区,那就不会加入到commit中,Git跟踪并管理的是修改,而非文件,第一次修改文件后进行**git add**添加至暂存区后,再次进行第二次修改,如果第二次修改没有进行**git add**添加操作,commit提交后仅会保存第一次修改的内容

撤销修改

git checkout -- file命令可以丢弃工作区的更改,把文件在工作区做的修改全部撤销

```

1 $git checkout -- 文件名称
2 #git checkout -- file命令中的--很重要,没有--,就变成了“切换到另一个分支”的命令

```

根据文件状态的不同存在以下几种结果

1. 文件在工作区被修改,但未**git add** 添加

执行后文件回到未修改状态,与版本库一致

2. 假设文件已经**git add**添加到暂存区,但未提交,在此期间又做了修改

执行后文件回到刚添加至暂存区的状态

3. 文件修改之后**git add**添加到暂存区,但未进行提交

a. 此时需要先通过**git reset HEAD <file>**可以把暂存区的

修改撤销掉 (unstage) , 重新放回 工作区:

```
1 $ git reset HEAD 文件名称 //将文件重新放回工作区
2 #git reset命令可以回退版本, 或把暂存区的修改回退到工作区。用HEAD时, 表示最新的版本
```

b. 此时再使用命令将工作区修改丢弃

```
1 $git checkout -- 文件名称
2 //操作完毕后可以使用git status查看仓库状态确定
```

4. 修改文之后并且提交到了版本库

!使用版本回退进行回退到上一个版本

场景1: 当你改乱了工作区某个文件的内容, 想直接丢弃工作区的修改时, 用命令 `git checkout -- file`。

场景2: 当你不但改乱了工作区某个文件的内容, 还添加到了暂存区时, 想丢弃修改, 分两步, 第一步用命令 `git reset HEAD <file>`, 就回到了场景1, 第二步按场景1操作。

场景3: 已经提交了不合适的修改到版本库时, 想要撤销本次提交, 使用[版本回退](#), 不过前提是还没有推送到远程库。

删除文件

rm:命令可以对文件在工作区进行删除,但是版本库还存在文件记录,使用 `git status` 命令,会提

示有哪些文件被删除了

```
1 $ rm 文件名
2 $ rm -rf 文件名
3 #参数 -rf 删除文件夹
```

git rm命令 与git commit命令可以从版本库对文件进行删除

```
1 $ git rm 文件名
2 $ git commit
```

git checkout -- 文件名 命令可以用版本库里的版本替换工作区的版本,无论工作区是修

改还是删除,都可以一键还原(**从来没有被添加到版本库就被删除的文件,是无法恢复的!**)

```
1 git checkout -- 文件名
2 #!使用git rm 删除的无法进行还原
```

1.如果用的**rm**删除文件, 那就相当于只删除了工作区的文件, 如果想要恢复, 直接用**git checkout --**

2.如果用的是**git rm**删除文件, 那就相当于不仅删除了文件, 而且还添加到了暂存区, 需要先**git reset HEAD <file>**, 然后再**git checkout -- <file>**

3.如果想彻底把版本库的删除掉, 先**git rm**, 再**git commit**

- 远程仓库

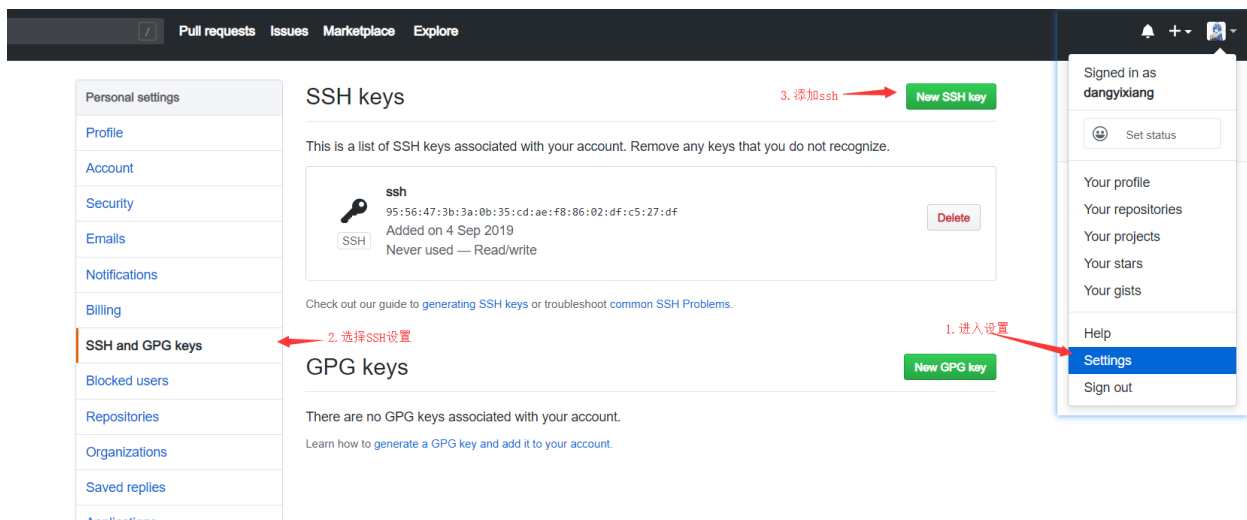
github创建仓库

准备工作

GitHub需要SSH Key, GitHub需要识别出你推送的提交确实是你推送的, 而不是别人冒充的, 而Git支持SSH协议, 所以, GitHub只要知道了你的公钥, 就可以确认只有你自己才能推送。

先检查有没有sshKey, 如果没有则使用命令创建

```
1 $ ssh-keygen -t rsa -C "邮箱@qq.com"
2 一路回车, 创建完成后可以通过命令查看Key值
3 $cat ~/.ssh/id_rsa.pub
4 ssh-rsa...
5 或者打开文件进行查看
6 默认文件地址:
7 C:\Users\用户名\.ssh\id_rsa.pub
```



Personal settings

Profile

Account

Security

Emails

Notifications

Billing

SSH and GPG keys

Blocked users

Repositories

Organizations

Saved replies

Applications

SSH keys / Add new

Title

名称

Key

Begin with 'ssh-rsa', 'ssh-dss', 'ssh-ed25519', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521'
刚刚创建的key

Add SSH key

GitHub允许你添加多个Key,
码云的sshKey创建同理

1.现在github中创建一个仓库

RepositoriesIssuesMarketplaceExplore

Discover interesting projects and people to populate your personal news feed.

News feed helps you keep up with recent activity on repositories you watch and people you follow.

More GitHub

News feed shows you events from people you follow and repositories you watch.

to your news feed

New repository

Import repository

New gist

New organization

New project

GitHub Actions

Automate any workflow. See the latest updates in beta.

GitHub Sponsors Matching Fund

Ready to support open source? GitHub will match your contribution to developers during their first year in GitHub Sponsors.

Welcome to the new dashboard. Get closer to the stuff you care about most.

Explore repositories

reason-react-native/reason-react-native

Write your React Native apps with Reason


OCaml 612

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 *

 **dangyixiang** ▼


 /

仓库名称

Great repository names are short and memorable. Need inspiration? How about **glowing-guide**?

Description (optional)

☒  **Public** 公开
Anyone can see this repository. You choose who can commit.

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

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

☐ **Initialize this repository with a README** 不勾选
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▼

Add a license: **None** ▼





Create repository

创建

2.关联仓库

Quick setup — if you've done this kind of thing before

 Set up in Desktop or **HTTPS** **SSH** `https://github.com/dangyixiang/Test.git` 

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# Test" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/dangyixiang/Test.git
git push -u origin master
```

使用此命令将本地仓库与远程仓库进行关联

推送至远程仓库

...or push an existing repository from the command line

```
git remote add origin https://github.com/dangyixiang/Test.git
git push -u origin master
```

- 1 #用于将本地库与远程库关联起来
- 2 \$ git remote add origin 仓库地址

```
3 #关联成功后用如下命令把本地内容推送到远程库中:
4 $ git push -u origin master
5 //master为默认分支名称,首次推送可以添加 -u 参数把本地的master分支内容推送的远程
  新的master分支,
6 //还会把本地的master分支和远程的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。
7 #修改远程库
8 $ git remote set-url origin 仓库地址
```

当第一次使用Git的clone或者push命令连接GitHub时,会得到一个警告:

```
1 The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.
2 RSA key fingerprint is xx.xx.xx.xx.xx.
3 Are you sure you want to continue connecting (yes/no)?
```

SSH连接在第一次验证GitHub服务器的Key时,需要确认GitHub的Key的指纹信息是否真的来自

GitHub的服务器,输入yes回车即可。

码云仓库创建成功首次连接需要输入确认并添加主机到本机SSH可信列表

```
1 ssh -T git@gitee.com
```

3.克隆仓库

克隆之前可对git进行选择位置,克隆后会在当前git位置中创建远程仓库同名文件夹

```
1 # $ git clone命令用于将远程仓库克隆至本地文件中
2 $ git clone https://github.com/daoke0818/testGit3.git
```

● 分支管理

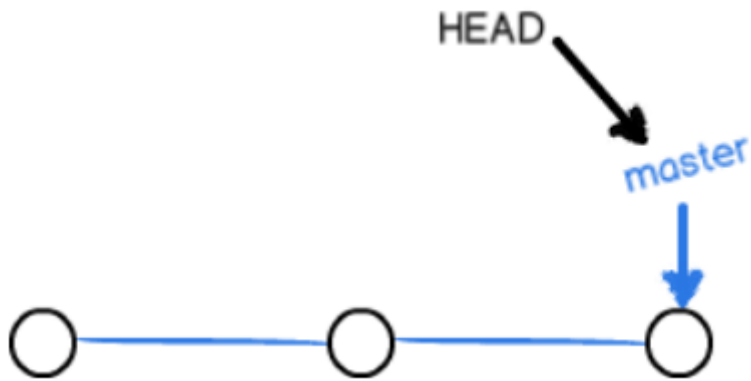
1. 概述

假设你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,由于代码还没写完,不完整的代码库会导致别人不能干活了。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。这种情况下需要**分支**来管理。自己在创建的新分支上进行开发,完成后一次性提交合并即可。

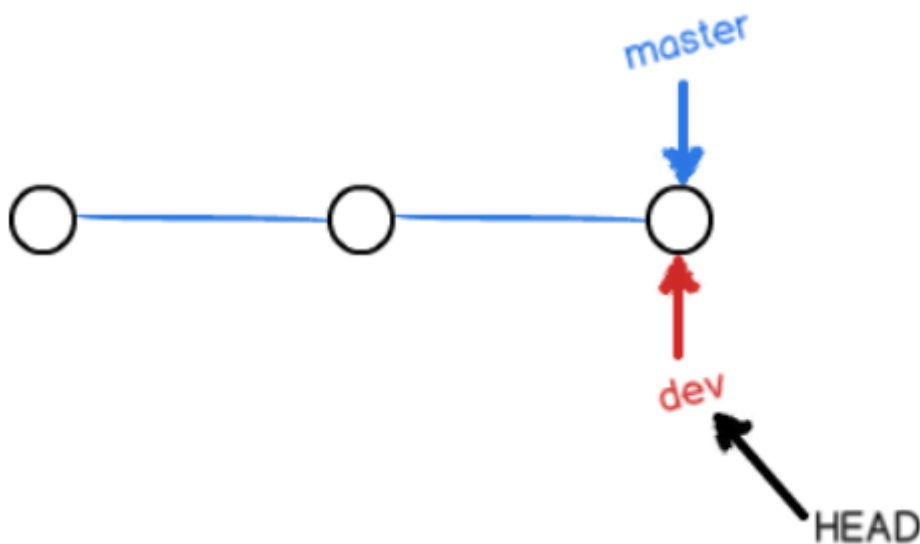
Git对于分支的创建、切换和删除都能非常快的实现,而SVN就很慢

2.创建分支与合并

创建分支



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

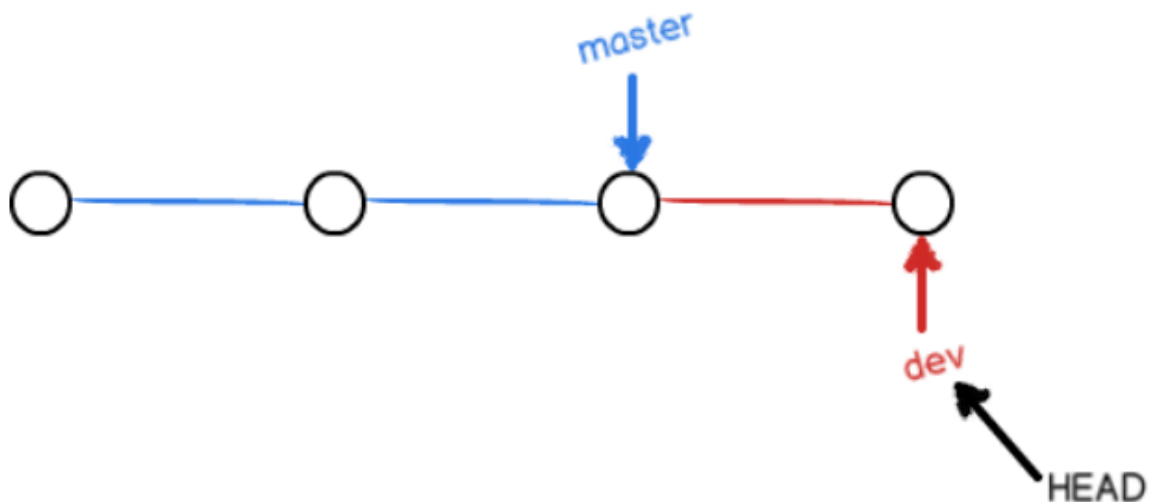


当我们创建新的分支，例如`dev`时，Git新建了一个指针叫`dev`，指向`master`相同的提交，再把`HEAD`指向`dev`，就表示当前分支在`dev`上：

```

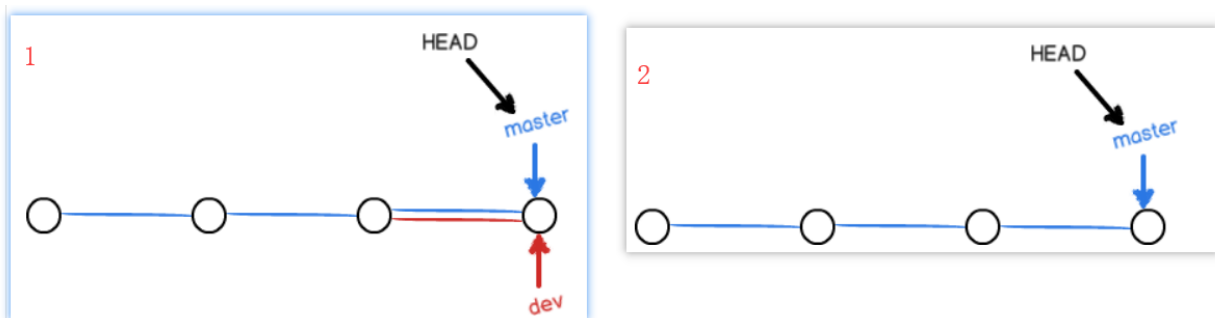
1  #创建分支方法1
2  $ git checkout -b dev
3  #创建分支方法2
4  $ git branch dev //先创建分支
5  $ git checkout dev //指向新分支
6
7  #查看分支
8  $ git branch
9  * dev
10 master
11  //* 号代表的是当前所在分支

```



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，分支切换后做的更改提交之后再次切换到`master`分支后，刚刚在`dev`分支所做的改动将会消失，

合并分支



直接把`master`指向`dev`的当前提交，就可以完成合并；合并完分支后，甚至可以删除`dev`分支。删除`dev`分支就是把`dev`指针给删掉，删掉后，就剩下了一条`master`分支；合并后`dev`分支更改的内容，也将会出现在`master`分支中；

```
1 #合并分支
2 $ git merge dev
3 #删除分支
4 $ git branch -d dev
```

switch命令

`git switch`命令也可以来创建于切换分支(新版本可以用)

```
1 #创建并切换到新的dev分支，可以使用：
2 $ git switch -c dev
3 #直接切换到已有的master分支，可以使用：
4 $ git switch master
```

3.分支冲突

例如当前分支A与分支B共同对一个文件进行了修改提交，当分支合并时将会出现：

```

1 #1. 分支A, 修改文件readme并提交
2 #2. 分支B, 修改文件readme并提交
3 #3. 合并分支
4 $ git merge 分支B
5 //提示:合并失败,解决冲突后再提交
6 Auto-merging readme.txt
7 CONFLICT (content): Merge conflict in readme.txt
8 Automatic merge failed; fix conflicts and then commit the result.

```

因为分支A与分支B都有了各自新的提交,Git无法执行快速合并,只能使徒把各自的修改合并起来,但是这种合并存在冲突,需要手动修改;合并时文件就已被修改,冲突提示是类似于一种提醒而不算是一种未执行成功的错误

```

SAKURA@DESKTOP-EPKEB13 MINGW32 /e/ConfigFile/GitlearnGit (master)
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.

SAKURA@DESKTOP-EPKEB13 MINGW32 /e/ConfigFile/GitlearnGit (master|MERGING)
$ git status 检查仓库状态
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
(use "git push" to publish your local commits)
You have unmerged paths.
(fix conflicts and run "git commit") 提示修复合并并提交
Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   readme.txt 冲突文件
no changes added to commit (use "git add" and/or "git commit -a")

```

文件中冲突部分,其中

<<<<<< HEAD 和 ===== 之间是当前分支的最新版,

===== 和 >>>>>> 分支B 之间是目标分支内容

需要做的是手动修改后删掉这些符号并重新提交

```

1 Git is a distributed version control system.
2 Git is free software distributed under the GPL.
3 Git has a mutable index called stage.
4 Git tracks changes of files.
5
6 <<<<<< HEAD
7 Creating a new branch is quick & simple.
8 =====
9 Creating a new branch is quick AND simple.
10 >>>>>> 分支B

```

可以使用带参数的git log 查看分支的合并情况

```

1 $ git log --graph --pretty=oneline --abbrev-commit

```

* 代表当前提交的分支目标

分支2位置

分支1(主分支位置)

```
SAKURA@DESKTOP-EPKEB13 MINGW32 /e/ConfigFile/Gitlearngit (master)
$ git log --graph --pretty=oneline --abbrev-commit
* b385072 合成测试2
* 47c7465 featrue1.1
* 70d0fe5 master0.1
* c5820a4 conflict fixed
* 0b49769 AND simple
* abfbf84 &simple
* f0748a4 test.06
* 5ac6905 test0.3
* 6c02047 test0.2
* 8777d64 test0.1
* 5f64333 test
* 12794ce delete license.txt
* 285914a readme0.5
* 844af5c readme0.4
```

再次进行了不同的提交后进行了合并

两组分支合并提交了 conflict fixed

出现了新的分支,并且两组分支都进行了提交

普通分支提交日志

4.分支管理策略

通常，合并分支时，如果可能，Git会用Fast forward模式(默认没产生分支冲突的亲狂笑)，但这种模式下，删除分支后，会丢掉分支信息。如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

合并分支时使用--no-f参数表示禁用Fast forward模式,因为本次合并要创建一个新的comit,所以要加上-m参数,声明提交说明

```
1 $ git merge --no-ff -m "提交说明" 分支
2 Merge made by the 'recursive' strategy.
3  readme.txt | 1 +
4  1 file changed, 1 insertion(+)
```

在实际开发中，我们应该按照几个基本原则进行分支管理：

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；

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；

你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

所以，团队合作的分支看起来就像这样：

小结

Git分支十分强大，在团队开发中应该充分应用。

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出曾经做过合并。

5.BUG分支

在当前分支做完修改之后需要保存提交才可以切换其他分支,但是其他分支有BUG需要去修复的情况下,并且当前工作进行了一般无法提交时,可以使用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：

```
1 #使用之后当前为提交的修改将会被暂时隐藏,此时可以切换至其他分支修复问题
2 $ git stash
```

```
3 Saved working directory and index state WIP on dev: f52c633 add merge
```

首先确定要在哪个分支上修复bug, 假定需要在`master`分支上修复, 就从`master`创建临时分支:

```
1 #切换修复BUG的分支
2 $ git checkout master
3 #创建新分支用于修复BUG
4 $ git checkout -b issue-101
5 #修复BUG并提交
6 $ git add readme.txt
7 $ git commit -m "fix bug 101"
8 [issue-101 4c805e2] fix bug 101
9 1 file changed, 1 insertion(+), 1 deletion(-)
```

修复完成后, 切换到`master`分支, 并完成合并, 最后删除`issue-101`分支:

```
1 #切换回分支
2 $ git checkout master
3 #合并bug分支
4 $ git merge --no-ff -m "修复了BUG" issue-101
5 #删除临时的BUG分支
6 $ git branch -d issue-101
```

在`master`分支上修复了bug后, `dev`分支是早期从`master`分支分出来的, 因此这个bug其实在当前`dev`分支上也存在。同样的bug, 要在`dev`上修复, 我们只需要把`fix bug 101`这个提交所做的修改“复制”到`dev`分支

```
1 #切换分支到dev
2 $ git checkout dev
3 #在临时的BUG分支中所做的提交复制到dev分支中
4 $ git cherry-pick 4c805e2
```

BUG修复之后就可以恢复刚开始的工作空间继续工作, 用`git stash list`命令可以查看工作空间列表, 恢复的方法分为两种:

1: 用`git stash apply`恢复, 但是恢复后, `stash`内容并不删除, 你需要用`git stash drop`来删除;

2: 用`git stash pop`, 恢复的同时把`stash`内容也删了:

```
1 #从stash恢复工作控件
2 $git stash apply
3 #删除stash内的缓存
4 git stash drop
5
6 #恢复并删除
7 $git stash pop
```

```
8
9 #查看stash列表
10 $ git stash list
11 stash@{0}: WIP on dev: f52c633 add merge
12
13 #指定恢复某stash
14 $ git stash apply stash@{0}
```

6.Feature分支——强行删除分支

如果在master分支上删除一个已经提交但没有合并的其它分支，则会报错：

```
1 $ git branch -d f5
2 error: The branch 'f5' is not fully merged.
3 If you are sure you want to delete it, run 'git branch -D f5'.
```

这时可以用参数 -D 强制删除：

```
1 $ git branch -D f5
```

需要注意的是，由于分支未合并，删除之后就没有任何记录了，分支上所有的修改也会丢失。

7.多人协作

从远程仓库克隆时,Git会自动把本地master分支和远程master分支对应起来,远程仓库默认名为 origin

```
1 #git remote命令查看远程库名称
2 $git remote
3 #git remote -v 查看远程库详细信息
4 origin git@github.com:michaelliao/learngit.git (fetch)
5 origin git@github.com:michaelliao/learngit.git (push)
6 //名称 地址 (权限/抓取fetch或推送push)
```

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，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

```
1 $ git push origin master(可指定其他分支)
```

抓取分支:在另一台电脑从远程库clone时(注意要把SSH key添加到远程库),默认情况下只能看到本地的master分支,如果需要在dev分支上开发，就必须创建远程origin的dev分支到本地才能在dev上继续修改,和继续push到远程

```
1 $ git checkout -b dev origin/dev
```

其他人向origin/dev分支推送了他的提交，而自己也对同样的文件作了修改，并试图推送,将会因为他人最新提交与自己试图推送的提交产生冲突而产生异常：

```
1 $ git push origin dev
```



```

2 To github.com:michaelliao/learngit.git
3 ! [rejected] dev -> dev (non-fast-forward)
4 error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'
5 hint: Updates were rejected because the tip of your current branch is behind
6 hint: its remote counterpart. Integrate the remote changes (e.g.
7 hint: 'git pull ...') before pushing again.
8 hint: See the 'Note about fast-forwards' in 'git push --help' for details.

```

产生异常后需要先用`git pull`把最新的提交从`origin/dev`抓下来，然后，在本地合并，解决冲突，再推送：

```

1 #直接使用git pull
2 $ git pull
3 There is no tracking information for the current branch.
4 Please specify which branch you want to merge with.
5 See git-pull(1) for details.
6
7 git pull <remote> <branch>
8
9 If you wish to set tracking information for this branch you can do so with:
10
11 git branch --set-upstream-to=origin/<branch> dev
12 //git pull失败，原因是没有指定本地dev分支与远程origin/dev分支的链接
13 #设置dev与origin/dev的连接
14 $ git branch --set-upstream-to=origin/dev dev
15 Branch 'dev' set up to track remote branch 'dev' from 'origin'.
16 #再次抓取
17 $ git pull
18 Auto-merging env.txt
19 CONFLICT (add/add): Merge conflict in env.txt
20 Automatic merge failed; fix conflicts and then commit the result.

```

`git pull`成功后,将会合并两次文件内容,但是依旧存在冲突,需要手动解决冲突,解决方法与分支管理中的一致,解决后重新提交文件并且push至远程库

多人协作的工作模式:

1. 首先，可以试图用`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 <branch-name>`推送就能成功！

小结

- 查看远程库信息，使用`git remote -v`;
- 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;
- 从本地推送分支，使用`git push origin branch-name`，如果推送失败，先用`git pull`抓取远程的新提交;
- 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`，本地和远程分支的名称最好一致;
- 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`;
- 从远程抓取分支，使用`git pull`，如果有冲突，要先处理冲突。

8.Rebase(待深入)

git合并代码方式主要有两种方式，分别为：

- 1、merge处理，这是大家比较能理解的方式。
- 2、rebase处理，中文此处翻译为衍合过程。

总结为：

git rebase过程相比较git merge合并整合得到的结果没有任何区别，但是通过git rebase衍合能产生一个更为整洁的提交历史。

如果观察一个衍合过的分支的历史提交记录，看起来会更清楚：仿佛所有修改都是在一根线上先后完成的，尽管实际上它们原来是同时并行发生的。

<https://www.cnblogs.com/pinefantasy/articles/6287147.html>

• 标签管理

发布一个版本时，先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的此刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

创建标签：

```
1 #查看标签 显示存在的标签列表
2 $git tag
3 t0.1
4 t0.2
5 t0.3
6
```

```

7 #创建标签,默认创建的标签将会打在最近的一次提交上
8 $ git tag v0.1
9
10 #指定标签,创建标签时追加版本号将会为此版本打上该标签
11 $git tag v0.2 f52c633
12
13 #创建标签的同时添加说明,用-a指定标签名, -m指定说明文字:
14 $ git tag -a v0.1 -m "version 0.1 released" 1094adb
15
16 #查看指定的标签信息
17 $ git show v0.9
18 commit f52c63349bc3c1593499807e5c8e972b82c8f286 (tag: v0.9)
19 Author: Michael Liao <askxuefeng@gmail.com>
20 Date: Fri May 18 21:56:54 2018 +0800
21
22 add merge
23
24 diff --git a/readme.txt b/readme.txt
25 ...
26

```

标签是指向commit的死指针, 分支是指向commit的活指针,任何分支都可找到标签,并且使当前分支

回到某一标签状态

操作标签:

```

1 #删除标签
2 $git tag -d v0.1
3 Deleted tag 'v0.1' (was f15b0dd)
4 创建的标签都只存储在本地, 不会自动推送到远程。打错的标签可以在本地安全删除。
5 #推送某标签到远程库
6 $git push origin <tagname>
7 #推送全部标签至远程库
8 $git push origin --tags
9 #删除远程标签
10 #1.先从本地删除
11 #2.再从远程删除
12 $git push origin :refs/tags/<tagname>
13

```

- **使用GitHub与码云**

- 在GitHub上, 可以任意Fork开源仓库;
- 自己拥有Fork后的仓库的读写权限;
- 可以推送pull request给官方仓库来贡献代码。
- 码云也同样提供了Pull request功能

本地库以及管理了一个名叫origin的远程库时,再使用git remote add添加相同名称时将会报错,可以删除已有的远程库,或者添加时使用另一个名称

```

1  #删除已有的远程库连接
2  $git remote rm 名称
3  #添加时使用其他名称
4  git remote add 自定义名称 远程地址
5
6  #提交时可根据不同名称推送至不同远程库
7  git push github master //推送至github
8  git push gitee master //推送至gitee
9
10 #把分支推到远程分支
11 git push origin test:master // 提交本地test分支 作为 远程的master分支
12 git push origin test:test // 提交本地test分支作为远程的test分支
13 #抓取远程库信息
14 git pull [remote] [branch] 取回远程仓库的变化, 并与本地分支合并
15

```

• 自定义Git

1. 忽略特殊文件

有时候有些文件必须放在Git工作目录中,但是不能提交他们,此时就需要提交时候忽略这些文件

- 1.创建一个.gitignore文件放在Git工作区的根目录下
- 2.把需要忽略的文件名填进去,Git就会自动忽略这些文件,例如

```

1  # Windows:
2  Thumbs.db
3  ehthumbs.db
4  Desktop.ini
5
6  # Python:
7  *.py[cod]
8  *.so

```

```
9 *.egg
10
11 # My configurations:
12 db.ini
13 deploy_key_rsa
```

3. 提交 .gitignore 文件, 可以使用 `git status` 检验文件后提交

设置完成忽略的文件将无法在git中添加, 因为他们都将被忽略, 但是可以使用 `-f` 强制添加到Git:

```
1 $ git add -f 文件
```

`git check-ignore` 命令检查 .gitignore 文件的问题

```
1 $ git check-ignore -v App.class
2 .gitignore:3:*.class App.class //第三行规则忽略了该文件
```

2. 配置别名

可以为命令或者操作设置别名对操作进行简化

例如:

```
1 #将git status 简化为 git st
2 $ git config --global alias.st status
3 输入git st时, 执行效果与git status 一致
4 #设置一个撤销快捷命令
5 $ git config --global alias.unstage 'reset HEAD'
6 输入
7 $ git unstage test.py
8 执行
9 $ git reset HEAD test.py
10 #配置一个git last, 显示最后一次提交信息
11 $ git config --global alias.last 'log -1'
12 //单个设置不需要加引号, 多个关键字设置需要加引号
```

配置Git的时候, 加上 `--global` 是针对当前用户起作用的, 如果不加, 那只针对当前的仓库起作用。

每个仓库的Git配置文件都放在 `.git/config` 文件中:

```
1 $ cat .git/config
2 [core]
3 repositoryformatversion = 0
4 filemode = true
5 bare = false
6 logallrefupdates = true
7 ignorecase = true
```

```
8  precomposeunicode = true
9  [remote "origin"]
10  url = git@github.com:michaelliao/learngit.git
11  fetch = +refs/heads/*:refs/remotes/origin/*
12  [branch "master"]
13  remote = origin
14  merge = refs/heads/master
15  [alias]
16  last = log -1
```

别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

当前用户的Git配置文件放在用户主目录下的一个隐藏文件.gitconfig中：

```
1  $ cat .gitconfig
2  [alias]
3  co = checkout
4  ci = commit
5  br = branch
6  st = status
7  [user]
8  name = Your Name
9  email = your@email.com
```

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置