创建分支前后的内容都属于新增内容

即master分支后的所有改动在develop后都属于develop

“作者”是修改者（参与者），“提交者”是最后将成果提交到仓库的人（项目核心成员）

git的本质就是操作工作流，在工作流中加入文件修改，一边修改文件一边操作工作流

基础操作：

增

删

改（忽略）

查（回溯）

【增加项目master】

git add abc.txt

git commit -m "新增点赞功能"

【增加分支】（是什么样的存在？存在的意义是什么）

在项目本体上增加内容

git checkout -b develop【增加develop分支】

git add .

git commit -m "新增关注功能"

【合并分支】

git merge develop【合并所有分支】

-------------------------------------------------------------------------------

【检查当前文件状态】

git status

｛

A //新上膛的文件（暂存区）

M //已修改，已上膛（暂存区）

M //已修改，未上膛（暂存区）

?? //新添加未跟踪文件

｝

【查看未上膛文件修改的部分】

git diff

【查看已上膛文件修改部分】

git diff --cached

或git diff --staged

（--staged 和 --cached 是同义词）

【查看日志】

git log

git log -p -2//-p每次提交内容差异 -2只查看近两次提交

git log --stat//查看每次简略的提交信息

git log --pretty=oneline（oneline长、short短、full、fuller、format可定制格式）//--pretty=指定展示格式

git log --pretty=format:"%h - %an, %ar : %s"（...请查询相关文档）

git log --pretty=format:"%h %s" --graph（--graph，当 oneline 或 format 与另一个 log 选项 --graph 结合使用时尤其有用）

（git log 的常用选项...查看相关文档）

（ 限制 git log 输出的选项...查看相关文档）

https://bingohuang.gitbooks.io/progit2/content/02-git-basics/sections/viewing-history.html

【忽略文件】（这是一个比较复杂的排除系统，有严格的符号规则）

cat .gitignore//创建.gitignore文件

.[oa] //忽略.o .a后缀文件

.~ //忽略以~结尾的文件

文件 .gitignore 的格式规范如下：

1.所有空行或者以 ＃ 开头的行都会被 Git 忽略。

2.可以使用标准的 glob 模式匹配。

3.匹配模式可以以（/）开头防止递归。

4.匹配模式可以以（/）结尾指定目录。

5.要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（!）取反。

注：

要养成一开始就设置好 .gitignore 文件的习惯，以免将来误提交这类无用的文件

【提交】

git commit - m "..."

git commit -a -m//直接将上过膛的文件直接提交，不用再经过再次上膛git add的操作

【取消追踪】

rm xxx.txt

git rm xxx.txt//记录此次移除操作

【移动文件（改名字）】

mv xxx\_form xxx\_to//从...移动到...

git mv README.md README（git mv file\_from file\_to）

//相当于

mv README.md README//移动

git rm README.md//移除README.md

git add README//上膛

注：

不过有时候用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

【覆盖提交操作】

git commit --amend

（提交完发现没改好需要重新提交）

git commit -m 'initial commit'//初次错误的提交

git add forgotten\_file//修改后的提交

git commit --amend//覆盖上次提交

操作暂存区和工作目录中已修改的文件

【撤销暂存区某一文件】

git reset HEAD CONTRIBUTING.md//git reset HEAD <file>…

【撤销对文件的修改】（撤销为上次提交的样子）

git checkout -- CONTRIBUTING.md//git checkout -- <file>...

【转到Master分支】

Git checkout master

【回溯】

git checkout [代码号]

// git checkout这是一个很危险的命令，因为他会覆盖你已修改但未提交的代码。（需要派和保护进度与分支会更好）

【克隆别人的仓库】

git clone

\*\*打标签\*\*

列出已有的标签、如何创建新标签、以及不同类型的标签分别是什么

【列出标签】

git tag

git tag -l 'v1.8.5\*'//只对 1.8.5 系列感兴趣

【创建标签】

轻量标签（lightweight）与附注标签（annotated）

轻量标签，特定提交的引用

附注标签是存储在 Git 数据库中的一个完整对象，有很多的信息

【附注标签】

git tag -a v1.4 -m 'my version 1.4'//图片1.1

//git tag -a

git show v1.4//可以看到标签信息与对应的提交信息//输出显示了打标签者的信息、打标签的日期时间、附注信息，然后显示具体的提交信息

【轻量标签】

git tag v1.4-lw//不需要使用 -a、-s 或 -m 选项，只需要提供标签名字

【后期打标签】

git tag -a v1.2 9fceb02//需要在命令的末尾指定提交的校验和

【共享标签】

git push origin v1.5//在创建完标签后你必须显式地推送标签到共享服务器上。

//在创建完标签后你必须显式地推送标签到共享服务器上。

//git push origin [tagname]

git push origin --tags//可一次性推送很多标签。这将会把所有不在远程仓库服务器上的标签全部传送到那里。

【检出标签】

git checkout -b version2 v2.0.0

//如果你想要工作目录与仓库中特定的标签版本完全一样，可以使用：

//git checkout -b [branchname] [tagname]

//当然，如果在这之后又进行了一次提交，version2 分支会因为改动向前移动了，那么 version2 分支就会和 v2.0.0 标签稍微有些不同，这时就应该当心了。

【Git别名】

可以通过 git config 文件来轻松地为每一个命令设置一个别名

图1.2

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

【取消暂存文件的易用性，添加自己的取消暂存别名】

git config --global alias.unstage 'reset HEAD --'

这会使下面的两个命令等价：

$ git unstage fileA

$ git reset HEAD -- fileA

通常也会添加一个 last 命令，像这样：

$ git config --global alias.last 'log -1 HEAD'

这样，可以轻松地看到最后一次提交：

$ git last

（Git 只是简单地将别名替换为对应的命令）

图1.3

\*\*Git分支\*\*

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。

该提交对象会包含一个指向暂存内容快照的指针。

首次提交产生的提交对象没有父对象。普通提交操作产生的提交对象有一个父对象，而由多个分支合并产生的提交对象有多个父对象

https://bingohuang.gitbooks.io/progit2/content/03-git-branching/sections/nutshell.html

Figure 1. 首次提交对象及其树结构

做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。

Figure 2. 提交对象及其父对象

Git 的分支，其实本质上仅仅是指向提交对象的可变指针。 Git 的默认分支名字是 master。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。 它会在每次的提交操作中自动向前移动。

master它就跟其它分支完全没有区别。

Figure 3. 分支及其提交历史

【分支创建】

git branch testing//这会在当前所在的提交对象上创建一个指针testing。

Figure 4. 两个指向相同提交历史的分支

Git 又是怎么知道当前在哪一个分支上呢？通过HEAD指针

HEAD是一个指针，指向当前所在的本地分支

（刚刚创建的分支HEAD不会自动借还到新分支中）

Figure 5. HEAD 指向当前所在的分支

git log 命令查看各个分支当前所指的对象

git log --oneline --decorate//显示HEAD指针当前所在的分支

【分支切换】

git checkout testing//git checkout，使HEAD指针指向testing分支

Figure 6. HEAD 指向当前所在的分支

好处是什么？

$ vim test.rb//编辑test.rb文件

$ git commit -a -m 'made a change'//自动提交上过膛的文件，不用add

Figure 7. HEAD 分支随着提交操作自动向前移动

git checkout master//查看master

Figure 8. 检出时 HEAD 随之移动

测试：

$ vim test.rb

$ git commit -a -m 'made other changes'

产生分支

Figure 9. 项目分叉历史

git log --oneline --decorate --graph --all//它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况

为什么Git处理这种逻辑这么快？

由于 Git 的分支实质上仅是包含所指对象校验和（长度为 40 的 SHA-1 值字符串）的文件，所以它的创建和销毁都异常高效。 创建一个新分支就像是往一个文件中写入 41 个字节（40 个字符和 1 个换行符），如此的简单能不快吗？

【知识点】

Git数据库中有tree、blob和Commit 对象：

Blob 对象：存储文件内容的快照，每个文件都被存储为一个blob对象。这些对象包含了实际的文件数据。

Tree 对象：树对象存储了文件和目录的层级结构信息。它类似于一个文件夹，可以包含多个blob对象（文件）和其他树对象（子目录）。

Commit 对象：每次提交产生一个commit对象，记录了一个快照，包含了指向树对象的引用、作者、提交消息等信息。

----------

【分支的建立与合并】

https://bingohuang.gitbooks.io/progit2/content/03-git-branching/sections/basic-branching-and-merging.html

Figure 1. 一个简单提交历史

你可以运行一个带有 -b 参数的 git checkout 命令：

git checkout -b iss53

它是下面两条命令的简写：

$ git branch iss53

$ git checkout iss53

Figure 2. 创建一个新分支指针

$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'

Figure 3. iss53 分支随着工作的进展向前推进

正在开发新的功能时候，接到紧急通知需要将上线分支进行漏洞修补，如何保证新功能保持干净

有一些方法可以绕过这个问题（即，保存进度（stashing） 和 修补提交（commit amending））

git checkout master//切回master分支

让我们建立一个针对该紧急问题的分支（hotfix branch），在该分支上工作直到问题解决：

git checkout -b hotfix//建立新的分支（紧急问题分支）

$ vim index.html//修改文件

$ git commit -a -m 'fixed the broken email address'//提交

请牢记！！：当你切换分支的时候，Git 会重置你的工作目录，使其看起来像回到了你在那个分支上最后一次提交的样子。Git 会自动添加、删除、修改文件以确保此时你的工作目录和这个分支最后一次提交时的样子一模一样。

Figure 4. 基于 master 分支的紧急问题分支 hotfix branch

你可以运行你的测试，确保你的修改是正确的，然后将其合并回你的 master 分支来部署到线上。 你可以使用 git merge 命令来达到上述目的：

$ git checkout master//转到master

$ git merge hotfix//合并hotfix分支

知识点：快进（fast-forward），没有分歧，HEAD直接向前移动，不产生分支问题

现在，最新的修改已经在 master 分支所指向的提交快照中，你可以着手发布该修复了。

Figure 5. master 被快进到 hotfix

关于这个紧急问题的解决方案发布之后，你准备回到被打断之前时的工作中。 然而，你应该先删除 hotfix 分支，因为你已经不再需要它了。master 分支已经指向了同一个位置

你可以使用带 -d 选项的 git branch 命令来删除分支：

$ git branch -d hotfix//删除hotfix分支

现在你可以切换回你正在工作的分支继续你的工作，也就是针对 #53 问题的那个分支（iss53 分支）。

$ git checkout iss53//转到iss53分支

$ vim index.html//编辑

$ git commit -a -m 'finished the new footer [issue 53]'//自动提交

Figure 6. 继续在 iss53 分支上的工作

如果你需要拉取 hotfix 所做的修改，你可以使用 git merge master 命令将 master 分支合并入 iss53 分支，或者你也可以等到 iss53 分支完成其使命，再将其合并回 master 分支。



图1.1

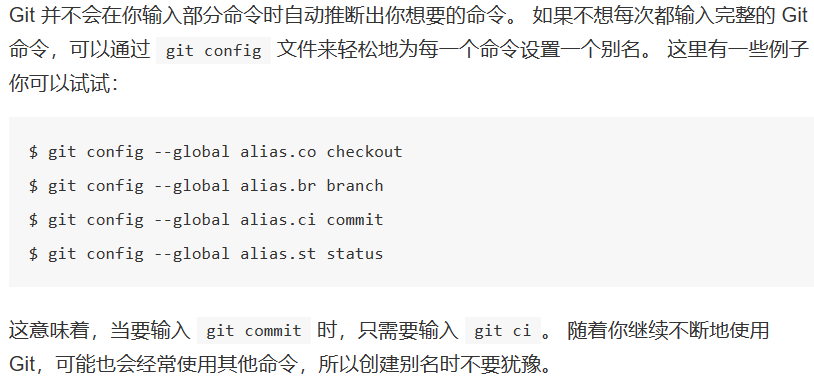


图1.2

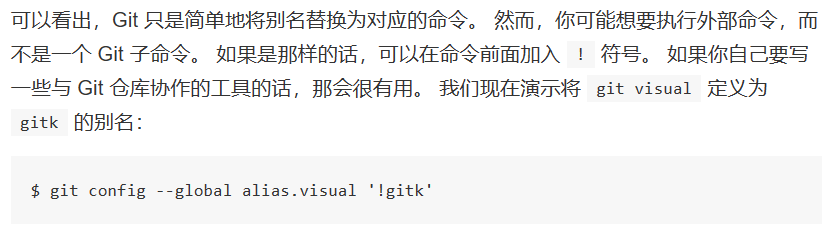


图1.3

https://bingohuang.gitbooks.io/progit2/content/03-git-branching/sections/basic-branching-and-merging.html#\_advanced\_merging

【分支的合并】

合并 iss53 分支到 master 分支，你只需要检出到你想合并入的分支，然后运行 git merge 命令：

$ git checkout master//跳转分支

$ git merge iss53//合并分支

因为master不是iss53的root或父类，需要找到其二者共同的root或父类，进行三方操作

Figure 7. 一次典型合并中所用到的三个快照

三方共同合作提交，这个被称作一次合并提交，特别之处在于他有不止一个父提交。

Figure 8. 一个合并提交

Git 会自行决定选取哪一个提交作为最优的共同祖先，Git 的这个优势使其在合并操作上比其他系统要简单很多。

既然你的修改已经合并进来了，你已经不再需要 iss53 分支了。 现在你可以在任务追踪系统中关闭此项任务，并删除这个分支。

git branch -d iss53//删除iss53分支

【遇到冲突时的分支合并】

有时候合并操作不会如此顺利。

如果你对 #53 问题的修改和有关 hotfix 的修改都涉及到同一个文件的同一处，在合并它们的时候就会产生合并冲突

此时 Git 做了合并，Git 会暂停下来，等待你去解决合并产生的冲突。

你可以在合并冲突后的任意时刻使用 git status 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：

$ git status//

任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。

为了解决冲突，你必须选择使用由 ======= 分割的两部分中的一个，或者你也可以自行合并这些内容。

在你解决了所有文件里的冲突之后，对每个文件使用 git add 命令来将其标记为冲突已解决。

如果你想使用图形化工具来解决冲突，你可以运行 git mergetool，该命令会为你启动一个合适的可视化合并工具，并带领你一步一步解决这些冲突：

git mergetool

使用其他工具：如果你想使用除默认工具（在这里 Git 使用 opendiff 做为默认的合并工具，因为作者在 Mac 上运行该程序）外的其他合并工具，你可以在 ``下列工具中（one of the following tools）'' 这句后面看到所有支持的合并工具。 然后输入你喜欢的工具名字就可以了。

等你退出合并工具之后，Git 会询问刚才的合并是否成功。 如果你回答是，Git 会暂存那些文件以表明冲突已解决： 你可以再次运行 git status 来确认所有的合并冲突都已被解决：

git status

如果你对结果感到满意，并且确定之前有冲突的的文件都已经暂存了，这时你可以输入 git commit 来完成合并提交。

如果你觉得上述的信息不够充分，不能完全体现分支合并的过程，你可以修改上述信息，添加一些细节给未来检视这个合并的读者一些帮助，告诉他们你是如何解决合并冲突的，以及理由是什么。

【分支管理】

git branch 命令不只是可以创建与删除分支。 如果不加任何参数运行它，会得到当前所有分支的一个列表

$ git branch//检查所有分支

\* 字符：它代表现在检出的那一个分支（也就是说，当前 HEAD 指针所指向的分支）

如果需要查看每一个分支的最后一次提交，可以运行 git branch -v 命令：

$ git branch -v//查看所有分支的最后一次提交

--merged 与 --no-merged 这两个有用的选项可以过滤这个列表中已经合并或尚未合并到当前分支的分支。

如果要查看哪些分支已经合并到当前分支，可以运行 git branch --merged：

$ git branch --merged//查看当前分支的过往合并分支都有哪些

注：在这个列表中分支名字前没有 \* 号的分支通常可以使用 git branch -d 删除掉；你已经将它们的工作整合到了另一个分支，所以并不会失去任何东西。

查看所有包含未合并工作的分支：

$ git branch --no-merged

这里显示了其他分支。使用 git branch -d 命令删除它时会失败。

$ git branch -d testing

error: The branch 'testing' is not fully merged.

If you are sure you want to delete it, run 'git branch -D testing'.

如果真的想要删除分支并丢掉那些工作，如同帮助信息里所指出的，可以使用 -D 选项强制删除它。

【分支开发工作流】

介绍一些常见的利用分支进行开发的工作流程

而正是由于分支管理的便捷，才衍生出这些典型的工作模式，你可以根据项目实际情况选择一种用用看。

【长期分支】

Figure 1. 渐进稳定分支的线性图

通常把他们想象成流水线（work silos）

Figure 2. 渐进稳定分支的流水线（“silo”）视图

再次强调一下，使用多个长期分支的方法并非必要，但是这么做通常很有帮助，尤其是当你在一个非常庞大或者复杂的项目中工作时。

【特性分支】

特性分支对任何规模的项目都适用。

特性分支是一种短期分支，它被用来实现单一特性或其相关工作。

Figure 3. 拥有多个特性分支的提交历史

......

Figure 4. 合并了 dumbidea 和 iss91v2 分支之后的提交历史

我们将会在 [\_distributed\_git] 中向你揭示更多有关分支工作流的细节，因此，请确保你阅读完那个章节之后，再来决定你的下个项目要使用什么样的分支策略（branching scheme）。

请牢记，当你做这么多操作的时候，这些分支全部都存于本地。 当你新建和合并分支的时候，所有这一切都只发生在你本地的 Git 版本库中 —— 没有与服务器发生交互。

https://bingohuang.gitbooks.io/progit2/content/03-git-branching/sections/remote-branches.html

【远程分支】就是远程服务器的分支（一种分支标签）

你可以通过 git ls-remote (remote) 来显式地获得远程引用的完整列表，或者通过 git remote show (remote) 获得远程分支的更多信息。

常用：远程跟踪分支，是远程分支状态的引用

以 (remote)/(branch) 形式命名

origin/master

Figure 1. 克隆之后的服务器与本地仓库

Figure 2. 本地与远程的工作可以分叉

如果要同步你的工作，运行 git fetch origin 命令

从远程服务器抓取本地没有的数据，并且更新本地数据库，移动 origin/master 指针指向新的、更新后的位置。

Figure 3. git fetch 更新你的远程仓库引用

你可以运行 git remote add 命令添加一个新的远程仓库（teamone）引用到当前的项目

将这个远程仓库命名为 teamone，将其作为整个 URL 的缩写。

Figure 4. 添加另一个远程仓库

运行 git fetch teamone 来抓取远程仓库（teamone）有而本地没有的数据。

teamone服务器上现有的数据是 origin 服务器上的一个子集

所以 Git 并不会抓取数据而是会设置远程跟踪分支 teamone/master 指向 teamone 的 master 分支。

（多了个指针/标签）

Figure 5. 远程跟踪分支 teamone/master

【推送】

当你想要公开分享一个分支时，需要将其推送到有写入权限的远程仓库上。

本地的分支并不会自动与远程仓库同步，你必须显式地推送想要分享的分支。

这样，你就可以把不愿意分享的内容放到私人分支上，而将需要和别人协作的内容推送到公开分支。

如果希望和别人一起在名为 serverfix 的分支上工作，你可以像推送第一个分支那样推送它。

运行 git push (remote) (branch):

$ git push origin serverfix//将serverfix分支推送到origin服务器

注：serverfix 分支名字展开为 refs/heads/serverfix:refs/heads/serverfix

如果并不想让远程仓库上的分支叫做 serverfix，可以运行 git push origin serverfix:awesomebranch 来将本地的 serverfix 分支推送到远程仓库上的 awesomebranch 分支。

下一次其他协作者从服务器上抓取数据时，他们会在本地生成一个远程分支 origin/serverfix，指向服务器的 serverfix 分支的引用

注：

如何避免每次输入密码

如果你正在使用 HTTPS URL 来推送，Git 服务器会询问用户名与密码。 默认情况下它会在终端中提示服务器是否允许你进行推送。

如果不想在每一次推送时都输入用户名与密码，你可以设置一个 `credential cache''。 最简单的方式就是将其保存在内存中几分钟，可以简单地运行 `git config --global credential.helper cache 来设置它。

要特别注意的一点是当抓取到新的远程跟踪分支时，本地不会自动生成一份可编辑的副本（拷贝）。 换一句话说，这种情况下，不会有一个新的 serverfix 分支 - 只有一个不可以修改的 origin/serverfix 指针。

可以运行 git merge origin/serverfix 将这些工作合并到当前所在的分支

如果想要在自己的 serverfix 分支上工作，可以将其建立在远程跟踪分支之上：

$ git checkout -b serverfix origin/serverfix//这会给你一个用于工作的本地分支，并且起点位于 origin/serverfix。

【跟踪分支】

跟踪分支是与远程分支有直接关系的本地分支。

如果在一个跟踪分支上输入 git pull，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

如果你愿意的话可以设置其他的跟踪分支 - 其他远程仓库上的跟踪分支，或者不跟踪 master 分支。

运行 git checkout -b [branch] [remotename]/[branch]。 这是一个十分常用的操作所以 Git 提供了 --track 快捷方式：

$ git checkout --track origin/serverfix//跟踪origin的serverfix分支

如果想要将本地分支与远程分支设置为不同名字，你可以轻松地增加一个不同名字的本地分支的上一个命令：

$ git checkout -b sf origin/serverfix//本地分支 sf 会自动从 origin/serverfix 拉取信息

$ git branch -u origin/serverfix//-u 就是 --set-upstream-to，将origin/serverfix拉取在本分支上

当你正在本地的master跟踪origin/master时，如果愿意的话可以使用 git merge @{u} 来取代 git merge origin/master。

$ git branch -vv//查看设置的所有跟踪分支

这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。

需要重点注意的一点是这些数字的值来自于你从每个服务器上最后一次抓取的数据。 这个命令并没有连接服务器，它只会告诉你关于本地缓存的服务器数据。

如果想要统计最新的领先与落后数字，需要在运行此命令前抓取所有的远程仓库。 可以像这样做：

$ git fetch --all; git branch -vv

【拉取】

当 git fetch 命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。 它只会获取数据然后让你自己合并。

然而 git pull 在大多数情况下相当于：一个 git fetch 紧接着一个 git merge 命令。

由于 git pull 的魔法经常令人困惑所以通常单独显式地使用 fetch 与 merge 命令会更好一些。

git pull还是危险了些，轻易地将本地的东西并入了所追踪的服务器分支

【删除远程分支】

假设你和你的协作者已经完成了一个特性并且将其合并到了远程仓库的 master 分支（或任何其他稳定代码分支）。

可以运行带有 --delete 选项的 git push 命令来删除一个远程分支。

如果想要从服务器上删除 serverfix 分支，运行下面的命令：

$ git push origin --delete serverfix

基本上这个命令做的只是从服务器上移除这个指针。

Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。

https://bingohuang.gitbooks.io/progit2/content/03-git-branching/sections/rebasing.html

【变基】

在 Git 中整合来自不同分支的修改主要有两种方法：merge 以及 rebase。

【变基的基本操作】

Figure 1. 分叉的提交历史

merge 命令，它会把两个分支的最新快照（C3 和 C4）以及二者最近的共同祖先（C2）进行三方合并合并的结果是生成一个新的快照（并提交）。

Figure 2. 通过合并操作来整合分叉了的历史

你可以提取在 C4 中引入的补丁和修改，然后在 C3 的基础上再应用一次。 在 Git 中，这种操作就叫做 变基。

使用 rebase 命令将提交到某一分支上的所有修改都移至另一分支上，就好像“重新播放”一样。

$ git checkout experiment//补丁

$ git rebase master//地基

//C4作为补丁，一步一步从共同的父类C2添加补丁到C3，生成新的C4`，原补丁指针跟随指向新的C4`上

Figure 3. 将 C4 中的修改变基到 C3 上

回到 master 分支，进行一次快进合并。

$ git checkout master

$ git merge experiment

//直接合成C4`，两个指针都指向C4`

Figure 4. master 分支的快进合并

这两种整合方法的最终结果没有任何区别，但是变基使得提交历史更加整洁。

【更有趣的变基例子】

从C2到出C3到C8，引出新的分支最后重新将其上传到server指针的分支C10

Figure 5. 从一个特性分支里再分出一个特性分支的提交历史

$ git rebase --onto master server client

以上命令的意思是：“取出 client 分支，找出处于 client 分支和 server 分支的共同祖先之后的修改，然后把它们在 master 分支上重演一遍”。 这理解起来有一点复杂，不过效果非常酷。

//当你不想将新写的东西上传server时的做法

Figure 6. 截取特性分支上的另一个特性分支，然后变基到其他分支

$ git checkout master

$ git merge client

//master合并client分支

Figure 7. 快进合并 master 分支，使之包含来自 client 分支的修改

使用 git rebase [basebranch] [topicbranch] 命令可以直接将特性分支

$ git rebase master server//将server基变到master分支

//相比于正常基变步骤，省略一个checkout master步骤

//server将指向最新的版本

Figure 8. 将 server 中的修改变基到 master 上

$ git checkout master

$ git merge server

//这个是重复学习了

至此，client 和 server 分支中的修改都已经整合到主分支里去了，你可以删除这两个分支：

$ git branch -d client

$ git branch -d server

Figure 9. 最终的提交历史

【变基的风险】

变基也并非完美无缺，要用它得遵守一条准则：

不要对在你的仓库外有副本的分支执行变基。

如果你遵循这条金科玉律，就不会出差错。 否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你。

会将主要开发者不要东西又重新找回来，并且你会发现被他扔掉的东西开发者竟然是你

【用变基解决变基】

你需要做的就是检查你做了哪些修改，以及他们覆盖了哪些修改。

Git 除了对整个提交计算 SHA-1 校验和以外，也对本次提交所引入的修改计算了校验和—— 即 ``patch-id''。

如果我们不是执行合并，而是执行 git rebase teamone/master, Git 将会：

1.检查哪些提交是我们的分支上独有的（C2，C3，C4，C6，C7）

2.检查其中哪些提交不是合并操作的结果（C2，C3，C4）

3.检查哪些提交在对方覆盖更新时并没有被纳入目标分支（只有 C2 和 C3，因为 C4 其实就是 C4'）

4.把查到的这些提交应用在 teamone/master 上面

从而得到一个新的结果

要想上述方案有效，还需要对方在变基时确保 C4' 和 C4 是几乎一样的。

如果你习惯使用 git pull ，同时又希望默认使用选项 --rebase，你可以执行这条语句 git config --global pull.rebase true 来更改 pull.rebase 的默认配置。

只要你把变基命令当作是在推送前清理提交使之整洁的工具，并且只在从未推送至共用仓库的提交上执行变基命令，你就不会有事。

假如你在那些已经被推送至共用仓库的提交上执行变基命令，并因此丢弃了一些别人的开发所基于的提交，那你就有大麻烦了，你的同事也会因此鄙视你。

如果你或你的同事在某些情形下决意要这么做，请一定要通知每个人执行 git pull --rebase 命令，这样尽管不能避免伤痛，但能有所缓解。

【变基 vs. 合并】

有一种观点认为，仓库的提交历史即是 记录实际发生过什么。 它是针对历史的文档，本身就有价值，不能乱改。 从这个角度看来，改变提交历史是一种亵渎，你使用谎言掩盖了实际发生过的事情。

如果由合并产生的提交历史是一团糟怎么办？

另一种观点则正好相反，他们认为提交历史是 项目过程中发生的故事。持这一观点的人会使用 rebase 及 filter-branch 等工具来编写故事，怎么方便后来的读者就怎么写。

总的原则是，只对尚未推送或分享给别人的本地修改执行变基操作清理历史，从不对已推送至别处的提交执行变基操作，这样，你才能享受到两种方式带来的便利。

\*\*远程仓库的使用\*\*

【查看远程仓库】

git remote//（remote远程）

git remote -v//显示需要读写远程仓库简写和对应的URL//轻松拉取其中任何一个用户的贡献//推送权限

【添加远程仓库】

git remote add pb https://github.com/paulboone/ticgit //可以在命令行中使用字符串 pb 来代替整个 URL

//git remote add <shortname> <url>

【从远程仓库中抓取与拉取】

git fetch [remote-name]

//此命令会将数据拉取到你的本地仓库 - 它并不会自动合并或修改你当前的工作

//运行 git pull 通常会从最初克隆的服务器上抓取数据并自动尝试合并到当前所在的分支。

【推送到远程仓库】

git push origin master//将此分支推送到远程仓库（服务器）上

//git push [remote-name] [branch-name]

//只有当你有所克隆服务器的写入权限，并且之前没有人推送过时，这条命令才能生效。 当你和其他人在同一时间克隆，他们先推送到上游然后你再推送到上游，你的推送就会毫无疑问地被拒绝。 你必须先将他们的工作拉取下来并将其合并进你的工作后才能推送。

【查看远程仓库】

git remote show origin

//git remote show [remote-name]

// 如果你是 Git 的重度使用者，那么还可以通过 git remote show 看到更多的信息

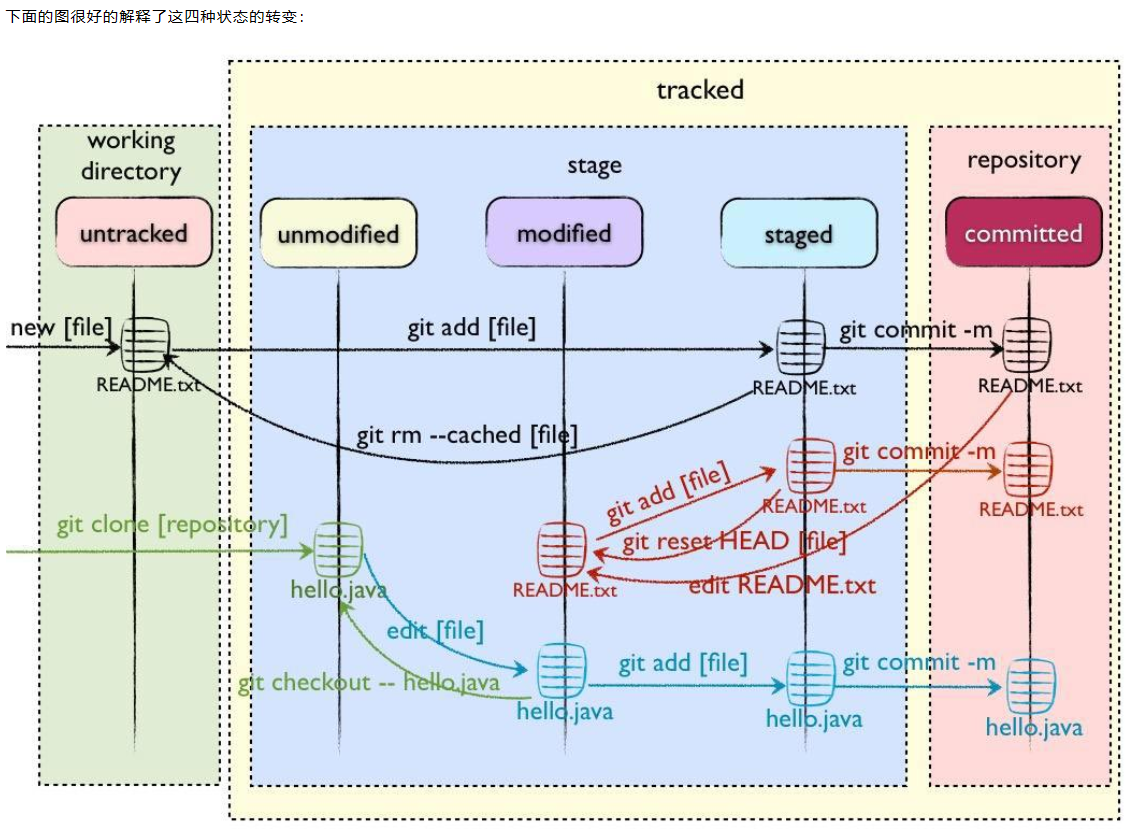
【远程仓库的移除与重命名】

git remote rename [old\_name] [new\_name]//重命名

//git remote rename pb paul

git remote rm paul//移除

//git remote rm [name]



[Git中文件的4种状态Untracked、Unmodify、Modified、Staged - 嘿十三 - 博客园 (cnblogs.com)](https://www.cnblogs.com/thirteen-yang/p/13878118.html)

服务器上的Git：

分布式Git：

GitHub的Git：

Git工具tool：

自定义Git：

Git与其他系统：

Git内部原理：

附录A：

附录B：

附录C：