**前言**

近几年技术发展十分迅猛，让部分同学养成了一种学习知识停留在表面，只会调用一些指令的习惯。我们时常有一种“我会用这个技术、这个框架”的错觉，等到真正遇到问题，才发现事情没有那么简单。

而Git也是一个大部分人都知道如何去使用它，知道有哪些命令，却只有少部分人知道具体原理的东西。了解一些底层的东西，可以更好的帮你理清思路，知道你真正在操作什么，不会迷失在Git大量的指令和参数上面。

**Git是怎么储存信息的**

这里会用一个简单的例子让大家直观感受一下git是怎么储存信息的。

首先我们先创建两个文件

$ git init

$ echo '111' > a.txt

$ echo '222' > b.txt

$ git add \*.txt

Git会将整个数据库储存在.git/目录下，如果你此时去查看.git/objects目录，你会发现仓库里面多了两个object。

$ tree .git/objects

.git/objects

├── 58

│ └── c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c

├── c2

│ └── 00906efd24ec5e783bee7f23b5d7c941b0c12c

├── info

└── pack

好奇的我们来看一下里面存的是什么东西

$ cat .git/objects/58/c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c

xKOR0a044K%

怎么是一串乱码？这是因为Git将信息压缩成二进制文件。但是不用担心，因为Git也提供了一个能够帮助你探索它的api git cat-file [-t] [-p]， -t可以查看object的类型，-p可以查看object储存的具体内容。

$ git cat-file -t 58c9

blob

$ git cat-file -p 58c9

111

可以发现这个object是一个blob类型的节点，他的内容是111，也就是说这个object储存着a.txt文件的内容。

这里我们遇到第一种Git object，blob类型，它只储存的是一个文件的内容，不包括文件名等其他信息。然后将这些信息经过SHA1哈希算法得到对应的哈希值  
58c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c，作为这个object在Git仓库中的唯一身份证。

也就是说，我们此时的Git仓库是这样子的：



我们继续探索，我们创建一个commit。

$ git commit -am '[+] init'

$ tree .git/objects

.git/objects

├── 0c

│ └── 96bfc59d0f02317d002ebbf8318f46c7e47ab2

├── 4c

│ └── aaa1a9ae0b274fba9e3675f9ef071616e5b209

...

我们会发现当我们commit完成之后，Git仓库里面多出来两个object。同样使用cat-file命令，我们看看它们分别是什么类型以及具体的内容是什么。

$ git cat-file -t 4caaa1

tree

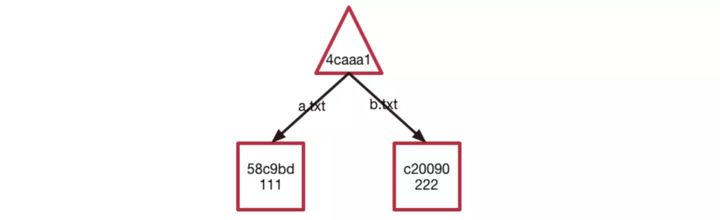
$ git cat-file -p 4caaa1

100644 blob 58c9bdf9d017fcd178dc8c0... a.txt

100644 blob c200906efd24ec5e783bee7... b.txt

这里我们遇到了第二种Git object类型——tree，它将当前的目录结构打了一个快照。从它储存的内容来看可以发现它储存了一个目录结构（类似于文件夹），以及每一个文件（或者子文件夹）的权限、类型、对应的身份证（SHA1值）、以及文件名。

此时的Git仓库是这样的：



$ git cat-file -t 0c96bf

commit

$ git cat-file -p 0c96bf

tree 4caaa1a9ae0b274fba9e3675f9ef071616e5b209

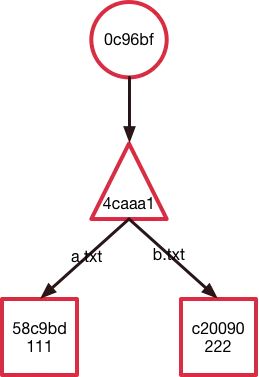
author lzane 李泽帆 1573302343 +0800

committer lzane 李泽帆 1573302343 +0800

[+] init

接着我们发现了第三种Git object类型——commit，它储存的是一个提交的信息，包括对应目录结构的快照tree的哈希值，上一个提交的哈希值（这里由于是第一个提交，所以没有父节点。在一个merge提交中还会出现多个父节点），提交的作者以及提交的具体时间，最后是该提交的信息。

此时我们去看Git仓库是这样的：



到这里我们就知道Git是怎么储存一个提交的信息的了，那有同学就会问，我们平常接触的分支信息储存在哪里呢？

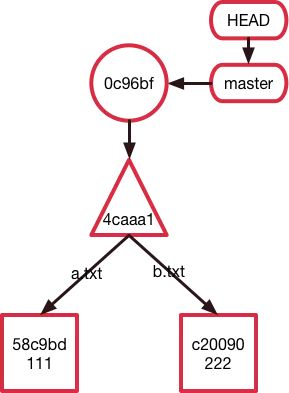
$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

0c96bfc59d0f02317d002ebbf8318f46c7e47ab2

在Git仓库里面，HEAD、分支、普通的Tag可以简单的理解成是一个指针，指向对应commit的SHA1值。



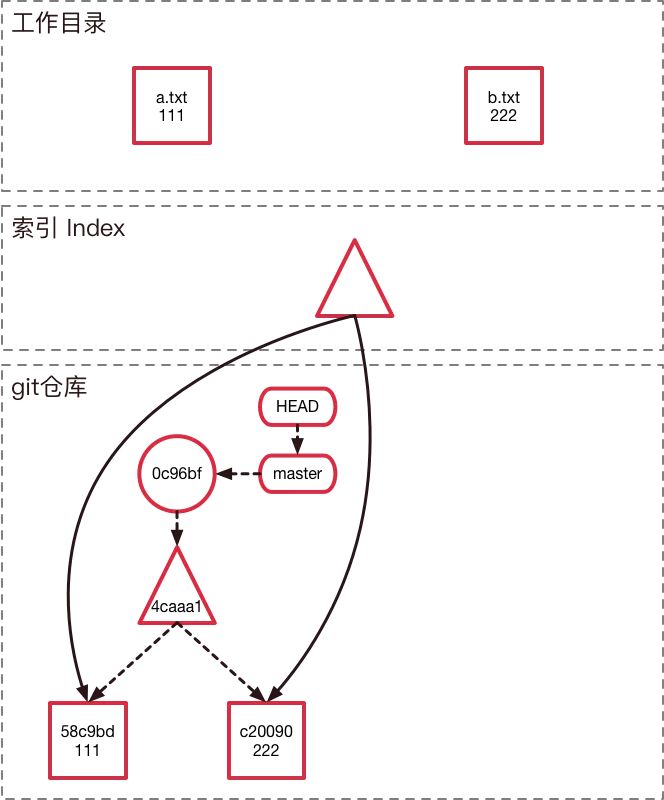
其实还有第四种Git object，类型是tag，在添加含附注的tag（git tag -a）的时候会新建，这里不详细介绍，有兴趣的朋友按照上文中的方法可以深入探究。

至此我们知道了Git是什么储存一个文件的内容、目录结构、commit信息和分支的。**其本质上是一个key-value的数据库加上默克尔树形成的有向无环图（DAG）**。这里可以蹭一下区块链的热度，区块链的数据结构也使用了默克尔树。

**Git的三个分区**

接下来我们来看一下Git的三个分区（工作目录、Index 索引区域、Git仓库），以及Git变更记录是怎么形成的。了解这三个分区和Git链的内部原理之后可以对Git的众多指令有一个“可视化”的理解，不会再经常搞混。

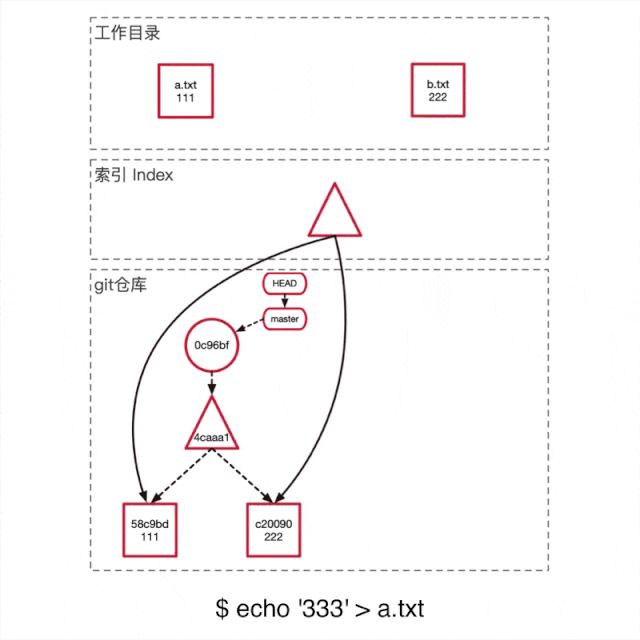
接着上面的例子，目前的仓库状态如下：



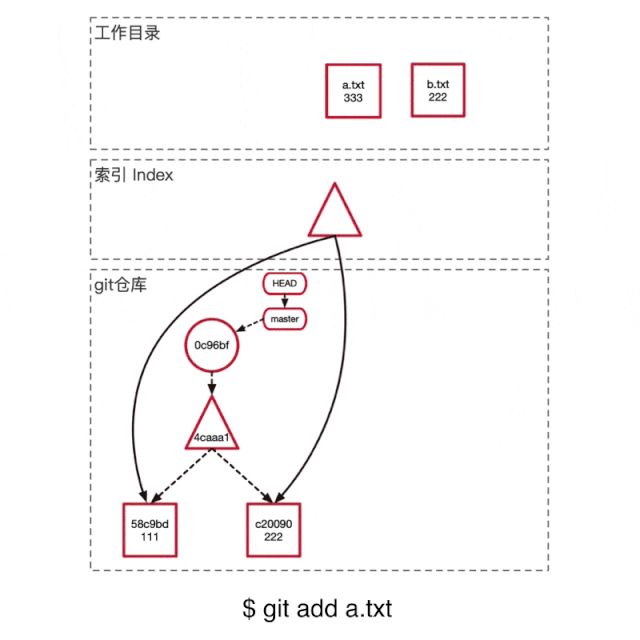
这里有三个区域，他们所储存的信息分别是：

* 工作目录 （ working directory ）：操作系统上的文件，所有代码开发编辑都在这上面完成。
* 索引（ index or staging area ）：可以理解为一个暂存区域，这里面的代码会在下一次commit被提交到Git仓库。
* Git仓库（ git repository ）：由Git object记录着每一次提交的快照，以及链式结构记录的提交变更历史。

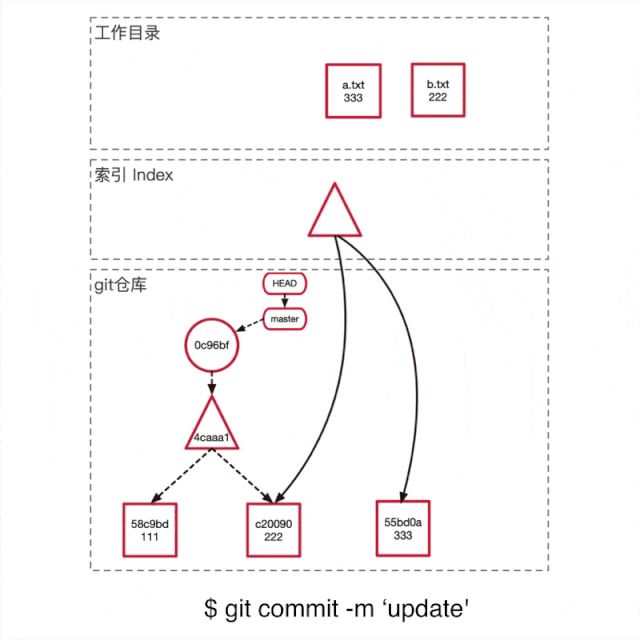
我们来看一下更新一个文件的内容这个过程会发生什么事。



运行echo "333" > a.txt将a.txt的内容从111修改成333，此时如上图可以看到，此时索引区域和git仓库没有任何变化。



运行git add a.txt将a.txt加入到索引区域，此时如上图所示，git在仓库里面新建了一个blob object，储存了新的文件内容。并且更新了索引将a.txt指向了新建的blob object。



运行git commit -m 'update'提交这次修改。如上图所示

1. Git首先根据当前的索引生产一个tree object，充当新提交的一个快照。
2. 创建一个新的commit object，将这次commit的信息储存起来，并且parent指向上一个commit，组成一条链记录变更历史。
3. 将master分支的指针移到新的commit结点。

至此我们知道了Git的三个分区分别是什么以及他们的作用，以及历史链是怎么被建立起来的。**基本上Git的大部分指令就是在操作这三个分区以及这条链。**可以尝试的思考一下git的各种命令，试一下你能不能够在上图将它们**“可视化”**出来，这个很重要，建议尝试一下。

如果不能很好的将日常使用的指令“可视化”出来，推荐阅读 [图解Git](https://link.zhihu.com/?target=https%3A//marklodato.github.io/visual-git-guide/index-zh-cn.html)

**一些有趣的问题**

有兴趣的同学可以继续阅读，这部分不是文章的主要内容

**问题1：为什么要把文件的权限和文件名储存在tree object里面而不是blob object呢？**

想象一下修改一个文件的命名。

如果将文件名保存在blob里面，那么Git只能多复制一份原始内容形成一个新的blob object。而Git的实现方法只需要创建一个新的tree object将对应的文件名更改成新的即可，原本的blob object可以复用，节约了空间。

**问题2：每次commit，Git储存的是全新的文件快照还是储存文件的变更部分？**

由上面的例子我们可以看到，Git储存的是全新的文件快照，而不是文件的变更记录。也就是说，就算你只是在文件中添加一行，Git也会新建一个全新的blob object。那这样子是不是很浪费空间呢?

这其实是Git在空间和时间上的一个取舍，思考一下你要checkout一个commit，或对比两个commit之间的差异。如果Git储存的是问卷的变更部分，那么为了拿到一个commit的内容，Git都只能从第一个commit开始，然后一直计算变更，直到目标commit，这会花费很长时间。而相反，Git采用的储存全新文件快照的方法能使这个操作变得很快，直接从快照里面拿取内容就行了。

当然，在涉及网络传输或者Git仓库真的体积很大的时候，Git会有垃圾回收机制gc，不仅会清除无用的object，还会把已有的相似object打包压缩。

**问题3：Git怎么保证历史记录不可篡改？**

通过SHA1哈希算法和哈系树来保证。假设你偷偷修改了历史变更记录上一个文件的内容，那么这个问卷的blob object的SHA1哈希值就变了，与之相关的tree object的SHA1也需要改变，commit的SHA1也要变，这个commit之后的所有commit SHA1值也要跟着改变。又由于Git是分布式系统，即所有人都有一份完整历史的Git仓库，所以所有人都能很轻松的发现存在问题。

希望大家读完有所收获，下一篇文章会写一些我日常工作中觉得比较实用的Git技巧、经常被问到的问题、以及发生一些事故时的处理方法。

作者：lzaneli，腾讯 TEG 前端开发工程师

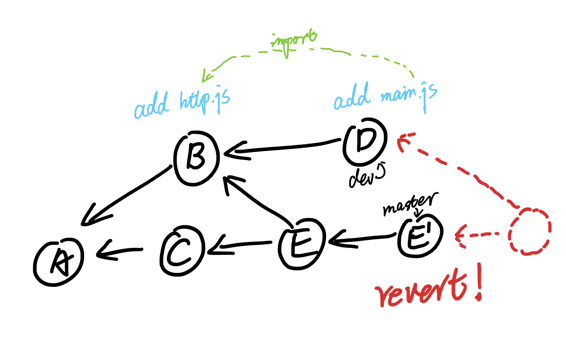
**参考**

* [Scott Chacon, Ben Straub - Pro Git-Apress (2014)](https://link.zhihu.com/?target=https%3A//git-scm.com/book/en/v1) 免费，有兴趣继续深入的同学推荐阅读这本书
* [Jon Loeliger, Matthew McCullough - Version Control with Git, 2nd Edition - O'Reilly Media (2012)](https://link.zhihu.com/?target=https%3A//www.amazon.com/Version-Control-Git-collaborative-development/dp/1449316387/ref%3Dsr_1_1%3Fkeywords%3DVersion%2BControl%2Bwith%2BGit%26qid%3D1573794832%26sr%3D8-1) 作为上面那本书的补充

**故事时间**

在开始正文之前，先来听一下这个故事。

如下图，小明从节点 A 拉了一条 dev 分支出来，在节点 B 中新增了一个文件 http.js，并且合并到 master 分支，合并节点为 E。这个时候发现会引起线上 bug，赶紧撤回这个合并，新增一个 revert 节点 E'。过了几天小明继续在 dev 分支上面开发新增了一个文件 main.js，并在这个文件中 import 了 http.js 里面的逻辑，在 dev 分支上面一切运行正常。可当他将此时的 dev 分支合并到 master 时候却发现，**http.js 文件不见了**，导致 main.js 里面的逻辑运行报错了。但这次合并并没有任何冲突。他又得重新做了一下 revert，并且迷茫的怀疑是 Git 的 bug。



两句经常听到的话：

—— ”合并前文件还在的，合并后就不见了“

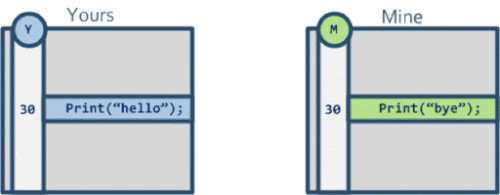
—— ”我遇到 Git 的 bug 了“

相信很多同学或多或少在不熟悉 Git 合并策略的时候都会发生过类似上面的事情，明明在合并前文件还在的，为什么合并后文件就不在了么？一度还怀疑是 Git 的 bug。这篇文章的目的就是想跟大家讲清楚 Git 是怎么去合并分支的，以及一些底层的基础概念，从而避免发生如故事中的问题，并对 Git 的合并结果有一个准确的预期。

**如何合并两个文件**

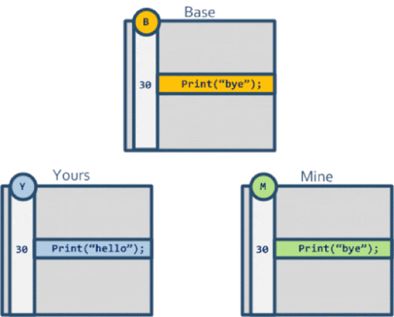
在看怎么合并两个分支之前，我们先来看一下怎么合并两个文件，因为两个文件的合并是两个分支合并的基础。

大家应该都听说过“三向合并”这个词，不知道大家有没有思考过为什么两个文件的合并需要三向合并，只有二向是否可以自动完成合并。如下图

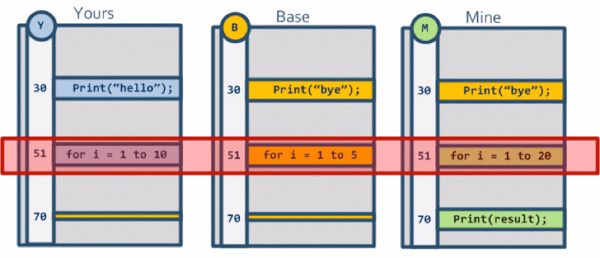


很明显答案是不能，如上图的例子，Git 没法确定这一行代码是我修改的，还是对方修改的，或者之前就没有这行代码，是我们俩同时新增的。此时 Git 没办法帮我们做自动合并。

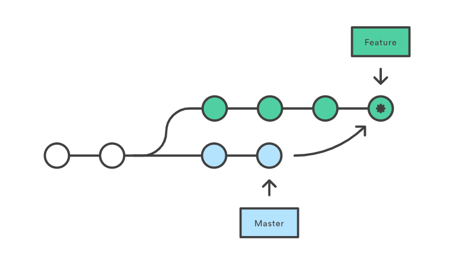
所以我们需要三向合并，所谓三向合并，就是找到两个文件的一个合并 base，如下图，这样子 Git 就可以很清楚的知道说，对方修改了这一行代码，而我们没有修改，自动帮我们合并这两个文件为 Print("hello")。



接下来我们了解一下什么是冲突？冲突简单的来说就是三向合并中的三方都互不相同，即参考合并 base，我们的分支和别人的分支都对同个地方做了修改。



**Git 的合并策略**

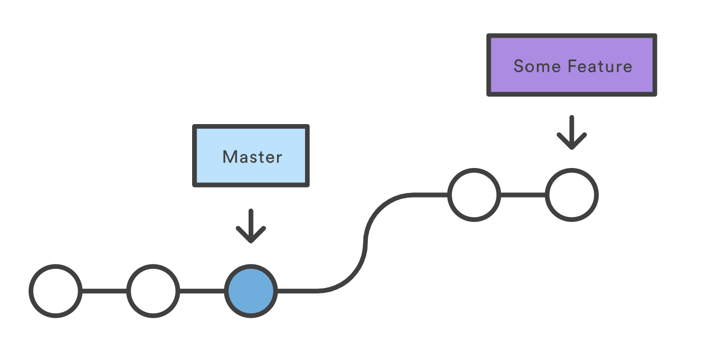


了解完怎么合并两个文件之后，我们来看一个使用 git merge 来做分支合并。如上图，将 master 分支合并到 feature 分支上，会新增一个 commit 节点来记录这次合并。

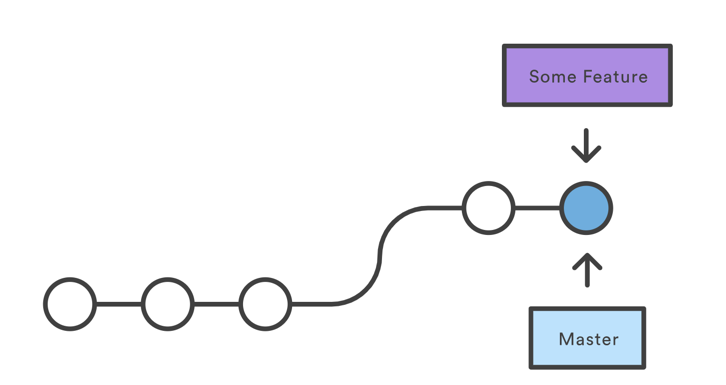
Git 会有很多合并策略，其中常见的是 Fast-forward、Recursive 、Ours、Theirs、Octopus。下面分别介绍不同合并策略的原理以及应用场景。默认 Git 会帮你自动挑选合适的合并策略，如果你需要强制指定，使用git merge -s <策略名字>

了解 Git 合并策略的原理可以让你对 Git 的合并结果有一个准确的预期。

**Fast-forward**



Fast-forward 是最简单的一种合并策略，如上图中将 some feature 分支合并进 master 分支，Git 只需要将 master 分支的指向移动到最后一个 commit 节点上。

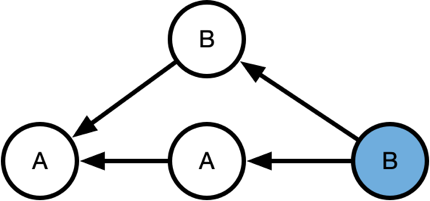


Fast-forward 是 Git 在合并两个没有分叉的分支时的默认行为，如果不想要这种表现，想明确记录下每次的合并，可以使用git merge --no-ff。

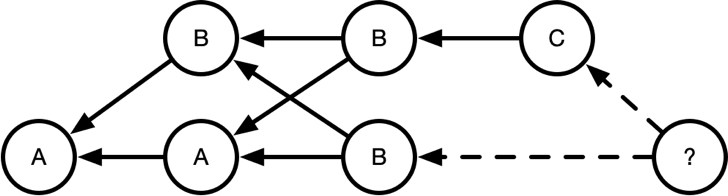
**Recursive**

Recursive 是 Git 分支合并策略中最重要也是最常用的策略，是 Git 在合并两个有分叉的分支时的默认行为。其算法可以简单描述为：递归寻找路径最短的唯一共同祖先节点，然后以其为 base 节点进行递归三向合并。说起来有点绕，下面通过例子来解释。

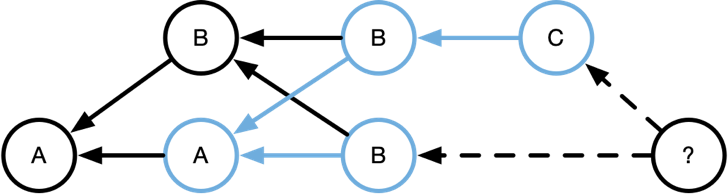
如下图这种简单的情况，圆圈里面的英文字母为当前 commit 的文件内容，当我们要合并中间两个节点的时候，找到他们的共同祖先节点（左边第一个），接着进行三向合并得到结果为 B。（因为合并的 base 是“A”，下图靠下的分支没有修改内容仍为“A”，下图靠上的分支修改成了“B”，所以合并结果为“B”）。



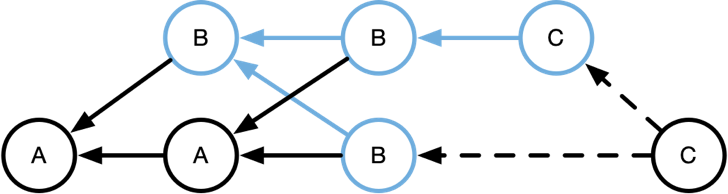
但现实情况总是复杂得多，会出现历史记录链互相交叉等情况，如下图：



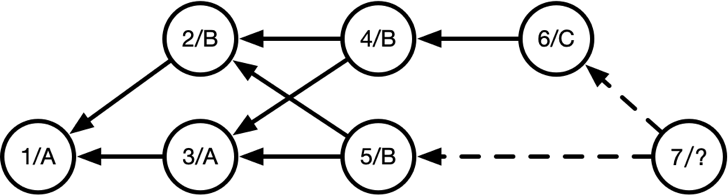
当 Git 在寻找路径最短的共同祖先节点的时候，可以找到两个节点的，如果 Git 选用下图这一个节点，那么 Git 将无法自动的合并。因为根据三向合并，这里是是有冲突的，需要手动解决。（base 为“A“，合并的两个分支内容为”C“和”B“）



而如果 Git 选用的是下图这个节点作为合并的 base 时，根据三向合并，Git 就可以直接自动合并得出结果“C”。（base 为“B“，合并的两个分支内容为”C“和”B“）

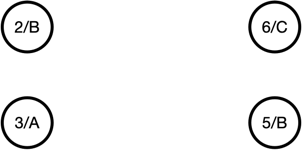


作为人类，在这个例子里面我们很自然的就可以看出来合并的结果应该是“C”（如下图，节点 4、5 都已经是“B”了，节点 6 修改成“C”，所以合并的预期为“C”）

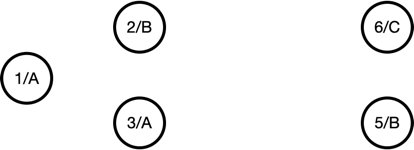


那怎么保证 Git 能够找到正确的合并 base 节点，尽可能的减少冲突呢？答案就是，Git 在寻找路径最短的共同祖先节点时，如果满足条件的祖先节点不唯一，那么 Git 会继续递归往下寻找直至唯一。还是以刚刚这个例子图解。

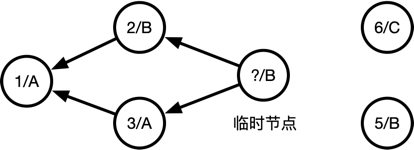
如下图所示，我们想要合并节点 5 和节点 6，Git 找到路径最短的祖先节点 2 和 3。



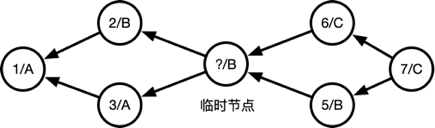
因为共同祖先节点不唯一，所以 Git 递归以节点 2 和节点 3 为我们要合并的节点，寻找他们的路径最短的共同祖先，找到唯一的节点 1。



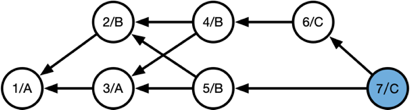
接着 Git 以节点 1 为 base，对节点 2 和节点 3 做三向合并，得到一个临时节点，根据三向合并的结果，这个节点的内容为“B”。



再以这个临时节点为 base，对节点 5 和节点 6 做三向合并，得到合并节点 7，根据三向合并的结果，节点 7 的内容为“C”



至此 Git 完成递归合并，自动合并节点 5 和节点 6，结果为“C”，没有冲突。

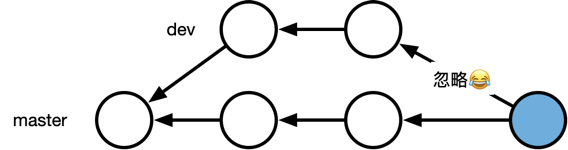


Recursive 策略已经被大量的场景证明它是一个尽量减少冲突的合并策略，我们可以看到有趣的一点是，对于两个合并分支的中间节点（如上图节点 4，5），只参与了 base 的计算，而最终真正被三向合并拿来做合并的节点，只包括末端以及 base 节点。

需要注意 Git 只是使用这些策略尽量的去帮你减少冲突，如果冲突不可避免，那 Git 就会提示冲突，需要手工解决。（也就是真正意义上的冲突）。

**Ours & Theirs**

Ours 和 Theirs 这两种合并策略也是比较简单的，简单来说就是保留双方的历史记录，但完全忽略掉这一方的文件变更。如下图在 master 分支里面执行git merge -s ours dev，会产生蓝色的这一个合并节点，其内容跟其上一个节点（master 分支方向上的）完全一样，即 master 分支合并前后项目文件没有任何变动。

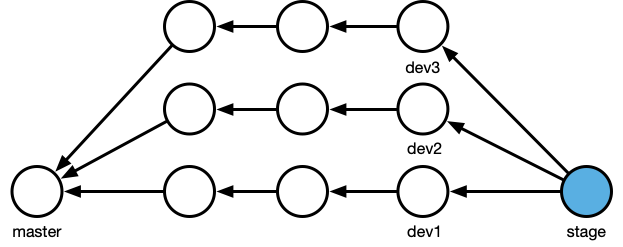


而如果使用 theirs 则完全相反，完全抛弃掉当前分支的文件内容，直接采用对方分支的文件内容。

这两种策略的一个使用场景是比如现在要实现同一功能，你同时尝试了两个方案，分别在分支是 dev1 和 dev2 上，最后经过测试你选用了 dev2 这个方案。但你不想丢弃 dev1 的这样一个尝试，希望把它合入主干方便后期查看，这个时候你就可以在 dev2 分支中执行git merge -s ours dev1。

**Octopus**

这种合并策略比较神奇，一般来说我们的合并节点都只有两个 parent（即合并两条分支），而这种合并策略可以做两个以上分支的合并，这也是 git merge 两个以上分支时的默认行为。比如在 dev1 分支上执行git merge dev2 dev3。

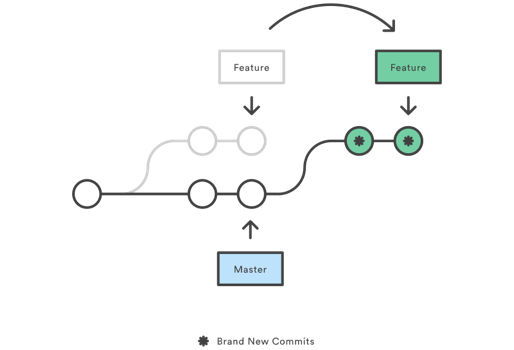


他的一个使用场景是在测试环境或预发布环境，你需要将多个开发分支修改的内容合并在一起，如果不用这个策略，你每次只能合并一个分支，这样就会导致大量的合并节点产生。而使用 Octopus 这种合并策略就可以用一个合并节点将他们全部合并进来。

**Git rebase**

git rebase 也是一种经常被用来做合并的方法，其与 git merge 的最大区别是，他会更改变更历史对应的 commit 节点。

如下图，当在 feature 分支中执行 rebase master 时，Git 会以 master 分支对应的 commit 节点为起点，新增两个**全新**的 commit 代替 feature 分支中的 commit 节点。其原因是新的 commit 指向的 parent 变了，所以对应的 SHA1 值也会改变，所以没办法复用原 feature 分支中的 commit。（这句话的理解需要[**这篇文章**](https://link.zhihu.com/?target=https%3A//www.lzane.com/tech/git-internal/)的基础知识）



对于合并时候要使用 git merge 还是 git rebase 的争论，我个人的看法是没有银弹，根据团队和项目习惯选择就可以。git rebase 可以给我们带来清晰的历史记录，git merge 可以保留真实的提交时间等信息，并且不容易出问题，处理冲突也比较方便。唯一有一点需要注意的是，不要对已经处于远端的多人共用分支做 rebase 操作。

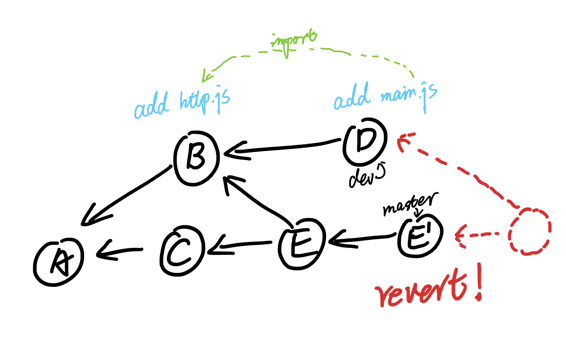
我个人的一个习惯是：对于本地的分支或者确定只有一个人使用的远端分支用 rebase，其余情况用 merge。

rebase 还有一个非常好用的东西叫 interactive 模式，使用方法是git rebase -i。可以实现压缩几个 commit，修改 commit 信息，抛弃某个 commit 等功能。比如说我要压缩下图 260a12a5、956e1d18，将他们与 9dae0027 合并为一个 commit，我只需将 260a12a5、956e1d18 前面的 pick 改成“s”，然后保存就可以了。



限于篇幅，git rebase -i 还有很多实用的功能暂不展开，感兴趣的同学可以自己研究一下。

**总结**



现在我们再来看一下文章开头的例子，我们就可以理解为什么最后一次 merge 会导致 http.js 文件不见了。根据 Git 的合并策略，在合并两个有分叉的分支（上图中的 D、E‘）时，Git 默认会选择 Recursive 策略。找到 D 和 E’的最短路径共同祖先节点 B，以 B 为 base，对 D，E‘做三向合并。B 中有 http.js，D 中有 http.js 和 main.js，E’中什么都没有。根据三向合并，B、D 中都有 http.js 且没有变更，E‘删除了 http.js，所以合并结果就是没有 http.js，没有冲突，所以 http.js 文件不见了。

这个例子理解原理之后解决方法有很多，这里简单带过两个方法：1. revert 节点 E'之后，此时的 dev 分支要抛弃删除掉，重新从 E'节点拉出分支继续工作，而不是在原 dev 分支上继续开发节点 D；2. 在节点 D 合并回 E’节点时，先 revert 一下 E‘节点生成 E’‘（即 revert 的 revert），再将节点 D 合并进来。

Git 有很多种分支合并策略，本文介绍了 Fast-forward、Recursive、Ours/Theirs、Octopus 合并策略以及三向合并。掌握这些合并策略以及他们的使用场景可以让你避免发生一些合并问题，并对合并结果有一个准确的预期。

希望这篇文章对大家有用，感兴趣的同学可以逛一逛我的博客 [http://www.lzane.com](https://link.zhihu.com/?target=http%3A//www.lzane.com) 或看看我的其他文章。

**参考**

* 三向合并 [http://blog.plasticscm.com/2016/02/three-way-merging-look-under-hood.html](https://link.zhihu.com/?target=http%3A//blog.plasticscm.com/2016/02/three-way-merging-look-under-hood.html)
* Recursive 合并【视频】[https://www.youtube.com/watch?v=Lg1igaCtAck](https://link.zhihu.com/?target=https%3A//www.youtube.com/watch%3Fv%3DLg1igaCtAck)
* 书籍 [**Scott Chacon, Ben Straub - Pro Git-Apress (2014)**](https://link.zhihu.com/?target=https%3A//git-scm.com/book/en/v2)
* 书籍 [**Jon Loeliger, Matthew McCullough - Version Control with Git, 2nd Edition - O’Reilly Media (2012)**](https://link.zhihu.com/?target=https%3A//www.amazon.com/Version-Control-Git-collaborative-development/dp/1449316387/ref%3Dsr_1_1)

这是该系列的最后一篇文章，之前的文章主要讲的是基础原理部分，在理解原理的基础上，介绍一些实用的技巧给大家，希望能提高大家的开发效率。

这篇文章因为更多的是列举实际应用的技巧，所以文章结构会显得散乱一些，也不会像前两篇文章那样要求大家顺序阅读。每个点都是互相独立的，大家可以根据自己的需要学习。

在这篇文章里我会使用**操作录屏**的方式来介绍例子，希望这种方式可以让你更直观的了解命令的使用方法。

**将几个commit压缩成一个**



⚠️ 这里有一点要特别注意的是：**rebase会导致新的commit节点产生**，所以切记不要对多人共用的远端分支进行rebase。

rebase -i 是个很实用且应用广泛的工具，希望大家都学会它的使用。它还可以用来修改commit信息，抛弃某些commit，对commit进行排序等等。具体命令如下，操作方式跟动图一致，都是在vim里面进行编辑。这里不展开，感兴趣的同学可以自己操作一下。

# Commands:

# p, pick <commit> = use commit

# r, reword <commit> = use commit, but edit the commit message

# e, edit <commit> = use commit, but stop for amending

# s, squash <commit> = use commit, but meld into previous commit

# f, fixup <commit> = like "squash", but discard this commit's log message

# x, exec <command> = run command (the rest of the line) using shell

# d, drop <commit> = remove commit

# l, label <label> = label current HEAD with a name

# t, reset <label> = reset HEAD to a label

# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]

# . create a merge commit using the original merge commit's

# . message (or the oneline, if no original merge commit was

# . specified). Use -c <commit> to reword the commit message.

另外如果要合并的是最近的几个commit，我们还可以用git reset --soft HEAD~3 && git commit -m 'xxx'来实现。对这个有问题的同学可以参照[**Git内部原理**](https://link.zhihu.com/?target=https%3A//www.lzane.com/tech/git-internal/)强调的可视化方法思考一下。

**找回丢失的commit节点或分支**

像上一步rebase后发现不符合预期，如何恢复？不小心删除了一个分支，如何找回？

“学会这个技能，你的同事会请你喝奶茶的，而且说不定还能收获妹子。” —— 来自往期课程的某位同学



主要思路为：**找到要返回的commit object的哈希值，然后执行git reset恢复**。

我们知道Git的出现就是为了尽量保证我们的操作不被丢失，在[**Git内部原理**](https://link.zhihu.com/?target=https%3A//www.lzane.com/tech/git-internal/)中我们讲过，git object是旦被创建，就不可变更，所以只要找到它对应的哈希值，就能找回。但是ref呢？在[**Git内部原理**](https://link.zhihu.com/?target=https%3A//www.lzane.com/tech/git-internal/)中我们也讲过，它是一个可变的指针，比如说你在master中提交了一个commit，那当前的master这个ref就会指向新的commit object的哈希值。reflog 就是将这些可变指针的历史给记录下来，可以理解成 **ref的log**，也可以理解成 **版本控制的版本控制**。

**获得一个干净的工作空间**

当我们实验一种思路，或者跟朋友讲代码时，我们可能会随意的修改代码。而当我们回到正常的开发时，我们需要一个干净的工作目录，即保证目前工作目录跟Git最后一次commit的文件是一致的。我们可以怎么做？



**尽量少用会丢失文件的操作，除非你能够确定不再需要这些文件。**

**修改最近一个commit**

commit完发现有一些临时的log忘记去掉？有一些文件忘记添加？commit信息出现错别字？



也可以使用 git reset HEAD~，然后执行你需要的修改，再commit即可，同上面介绍的命令效果是相同的。

**提交一个文件中的部分修改**



Git interactive add 还有很多功能，也推荐大家有时间可以尝试一下。

**禁止修改多人共用的远端分支**

如果一条远端分支有多人共用，那么不要在上面执行reset、rebase等会修改这条分支已经存在的commit object的命令。

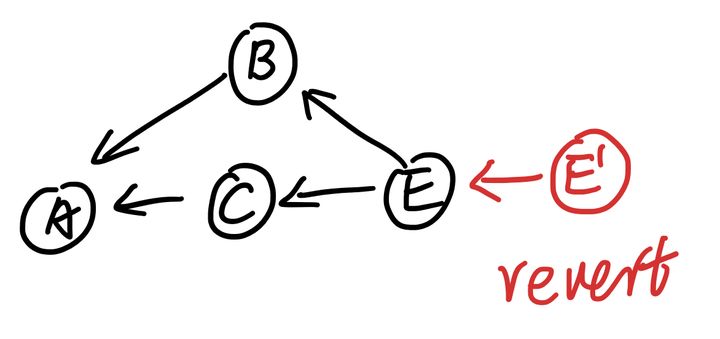


具体的解释参照这篇文章 [**Rebase and the golden rule explained**](https://link.zhihu.com/?target=https%3A//www.daolf.com/posts/git-series-part-2/) 。

**撤销一个合并**

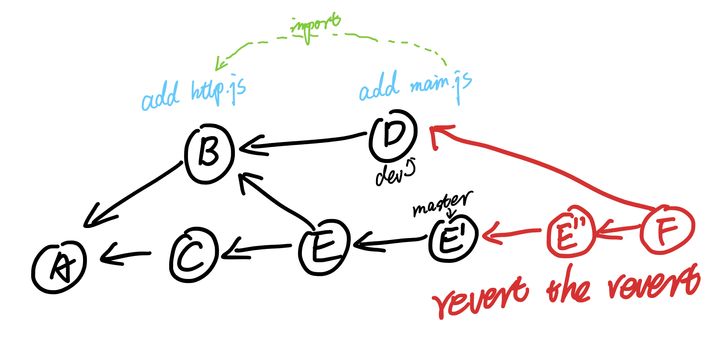
如果是一个本地分支，仅需git reset --hard <合并前的SHA1>即可。

如果这个分支已经被推送到远端，比如说合并进master，发到线上才发现有bug需要回滚。这时分支有可能已经被其他人所使用，根据“禁止修改多人共用的远端分支”，你需要执行git revert -m 1 <合并的SHA1>，新增一个revert节点，如下图中的E'。



但要注意不要在原特性分支继续开发，而应该删除原来的分支，从E'节点拉出新分支做bug修复等。

如果在原特性分支上继续开发，则在合并回master的时候需要做一次revert操作revert掉E'节点，变成E‘’（如下图），不然很容易出现丢失文件等问题。具体原因分析参照[**分支合并中的总结**](https://link.zhihu.com/?target=https%3A//www.lzane.com/tech/git-merge/%3F%23%25E6%2580%25BB%25E7%25BB%2593)



**从整个历史中删除一个文件**

代码要开源了，但发现其中包括密钥文件或内网ip怎么办？

git filter-branch --tree-filter 'rm -f passwords.txt' HEAD

可以使用filter-branch命令，它的实现原理是将每个commit checkout出来，然后执行你给它的命令，像上面的rm -f passwords.txt，然后重新commit回去。

⚠️ 这个操作属于高危操作，会修改历史变更记录链，产生全新的commit object。所以执行前请通知仓库的所有开发者，执行后所有开发者从新的分支继续开发，弃用以前的所有分支。

**其他好用的命令**

下面这些命令也是比较实用的命令，感兴趣的同学可以自己学习一下。

* git bisect 二分查找出现问题的变更节点，比如你发现当前提前下测试是不通过的，但HEAD～10（10个提交前）的测试是可以通过的，就可以用git bisect 来帮你定位到出现问题的变更点。
* git blame 查看某行代码最后是谁修改的。
* git show-branch 直观的展示多条分支间的关系。
* git subtree 拆分或合并仓库。

希望大家读完有所收获。感兴趣的同学可以阅读同系列的其他文章：

* [**这才是真正的Git——Git内部原理**](https://link.zhihu.com/?target=https%3A//www.lzane.com/tech/git-internal/)
* [**这才是真正的Git——分支合并**](https://link.zhihu.com/?target=https%3A//www.lzane.com/tech/git-merge/)

**参考**

* 书籍 [**Scott Chacon, Ben Straub - Pro Git-Apress (2014)**](https://link.zhihu.com/?target=https%3A//git-scm.com/book/en/v2)
* 书籍 [**Jon Loeliger, Matthew McCullough - Version Control with Git, 2nd Edition - O’Reilly Media (2012)**](https://link.zhihu.com/?target=https%3A//www.amazon.com/Version-Control-Git-collaborative-development/dp/1449316387/ref%3Dsr_1_1)

[Rebase and the golden rule explained](https://link.zhihu.com/?target=https%3A//www.daolf.com/posts/git-series-part-2/)