深入理解Git - 一切皆commit

目录

在对 git 有了基本理解和知道常规操作之后，如何对 git 的使用有进一步的理解？
一切皆 commit 或许是个不错的理解思路。

本文将从『一切皆 commit 』的角度，通过 git 中常见的名词，如 commit, branch, tag, HEAD 和动词，如 cherry-pick, rebase, reset, revert, stash 来理解 git。通过这些理解，期望能够更好地处理使用 git 中遇到的问题。

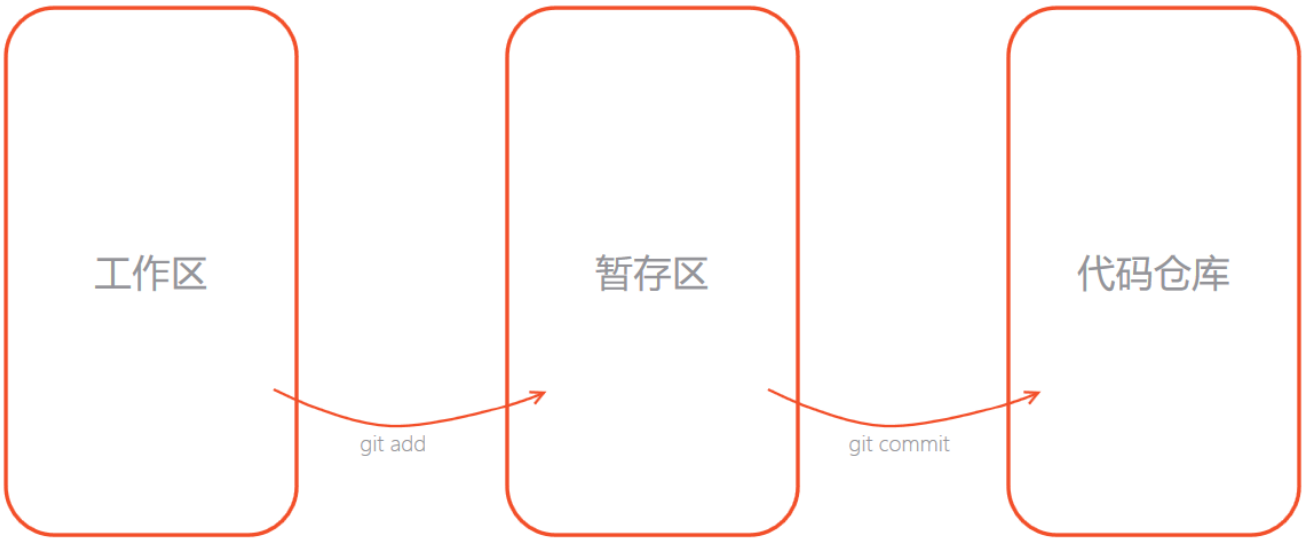
比如：

- 1 做了两个提交的修改，然后删掉分支了，过会发现刚才两个提交有价值，怎么找回来？
- 2 基于当前 release 分支开发功能，中途误合并了 dev 分支，
  然后又进行了几次提交，怎么取消合并dev的操作？
- 3 rebase(变基)究竟是什么意思？
  等等。

> 配合希沃白板课件食用，效果更佳：
> 【希沃白板5】课件分享：《Git 进阶 - 从使用角度深入理解Git》
> https://r302.cc/ke8XdO?platform=enpc&channel=copylink
> 点击链接直接预览课件

一切皆 commit

1 commit 的原子性



在 git 中有工作区，暂存区和代码仓库三个概念，那为什么要有暂存区呢？为了保证提交的原子性，在 git 的应用层面上，提交（commit，名词）是 git 主要命令的操作的最小单位了。

关于此，可以查看这篇知乎贴：为什么要先 git add 才能 git commit ？ - Ivony的回答 - 知乎

本文中的内容很少涉及工作区和暂存区的操作，有了 commit 是 git 操作的基本单位这个概念，接下来将从『一切皆 commit』来理解 git。

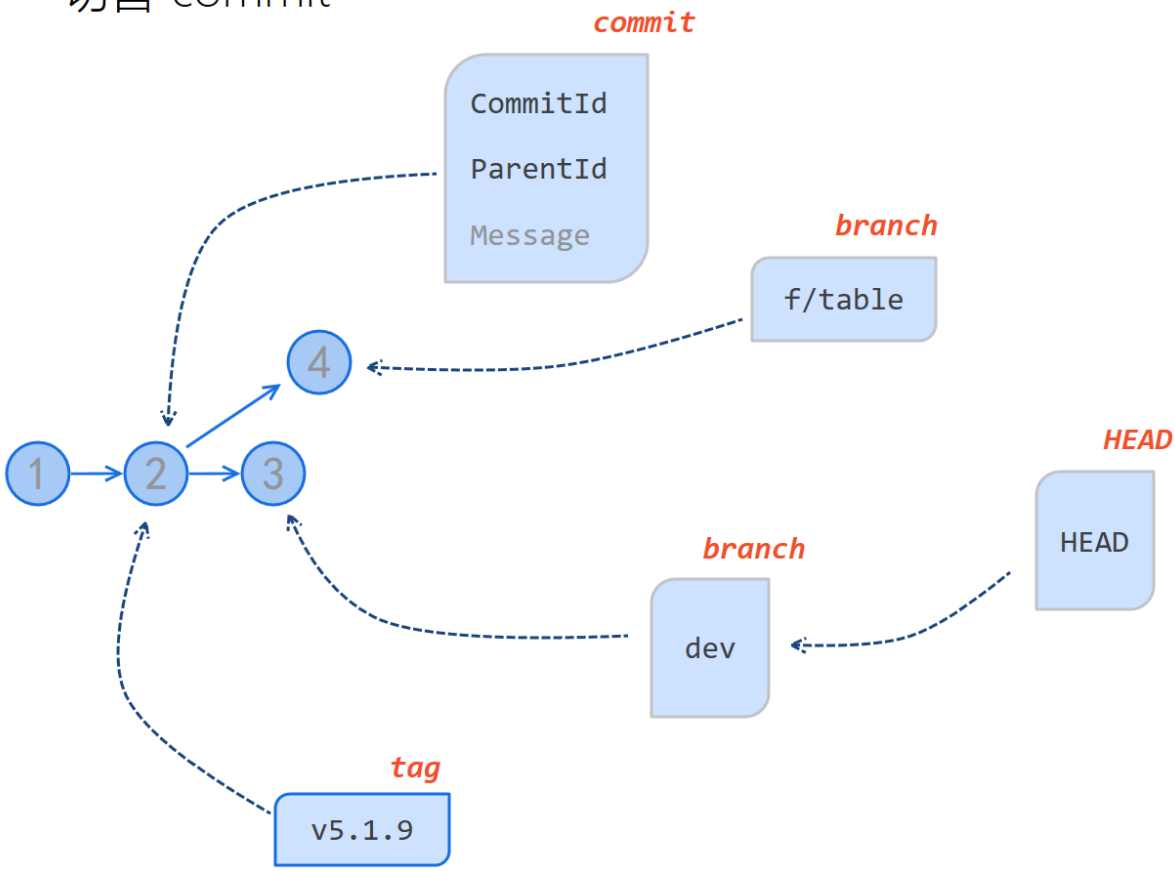
---

2 一切皆 commit ：名词部分

2.1 本地仓库

关注

Part1 一切皆 commit

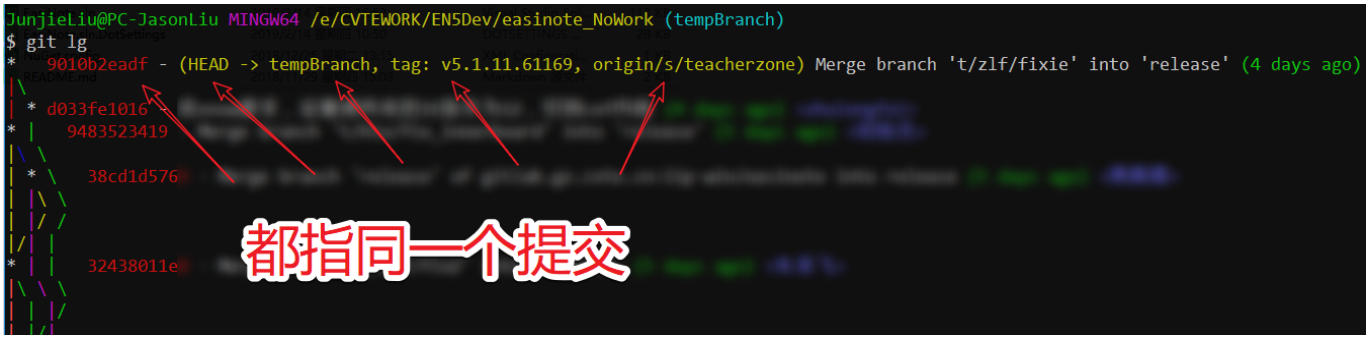


目录

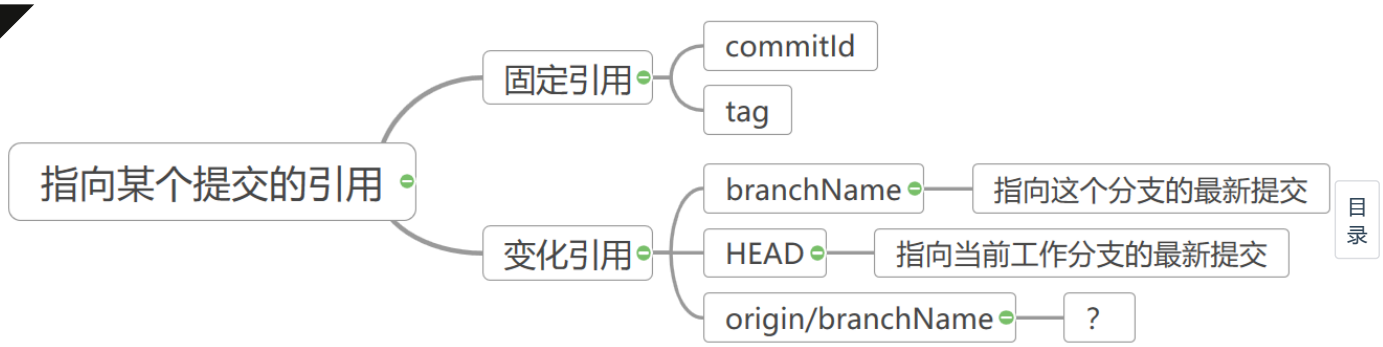
如上图，其实比较好理解，我们知道 commit 有一个 commit id，另外还是 branch（分支），tag（标签），HEAD（当前分支头结点）这些概念。他们都是指向某个提交的引用（或者理解为指针）。

- branch（分支）：指向你当前工作分支的最新的那个提交，当在当前分支有了新的提交，则 git 自动更新这个分支指针，以指向最新的提交。
- tag（标签）：对某个提交或者分支打 tag 之后，将固定指向那个提交，后续即使分支有更新甚至删除，tag 所指向的提交不变，且一直存在。
- HEAD（头结点）：指向当前工作的分支，即 HEAD 是当前分支的一个引用，如果切换了分支，HEAD 随之更新。

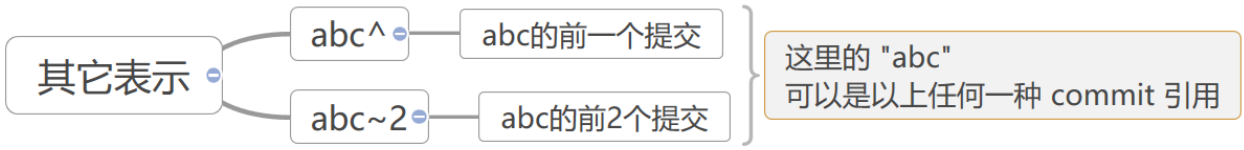
如此，便理解了，branch，tag，HEAD 这些，本质上都是指向某个提交的引用，即：一切都是 commit 。



关注



目录



2.2 远端仓库

有一个引用，需要单独说明，就是 origin/branch ，通常称之为远程分支，那这个远程分支指向哪里呢？
如何在 『一切皆commit』 这句咒语下理解远程仓库？

以 master 分支为例，origin/master 指向的，就是当前远端 master 分支最新的那个提交。等等，其实这句话有点小问题，应该是最后一次更新本地仓库时，远端 master 分支最新的那个提交。那什么时候会更新远程仓库？在执行 pull push fetch 时更新。

你或许听说过 git pull = git fetch + git merge 的说法。
当执行 `git fetch` 命令时，只更新 origin/master 分支（包括所有其它的 origin 远端分支），但并不会影响本地的任何分支。

那要更新本地的 master 分支怎么办？ `git merge origin/master` ，将远端的分支合并到本地分支，即完成了对本地 master 分支的更新。所以，实际上，git pull = git fetch + git merge 。
`(@master)git pull = git fetch & git merge origin/master`

案例

你在 f/table 分支开发功能，现在需要合并最新dev，可以怎么做？

刚学 git 时，可能会这么做：

Copy

```
(@f/table) git checkout dev
(@dev) git pull
(@dev) git checkout f/table
(@f/table) git merge dev
```

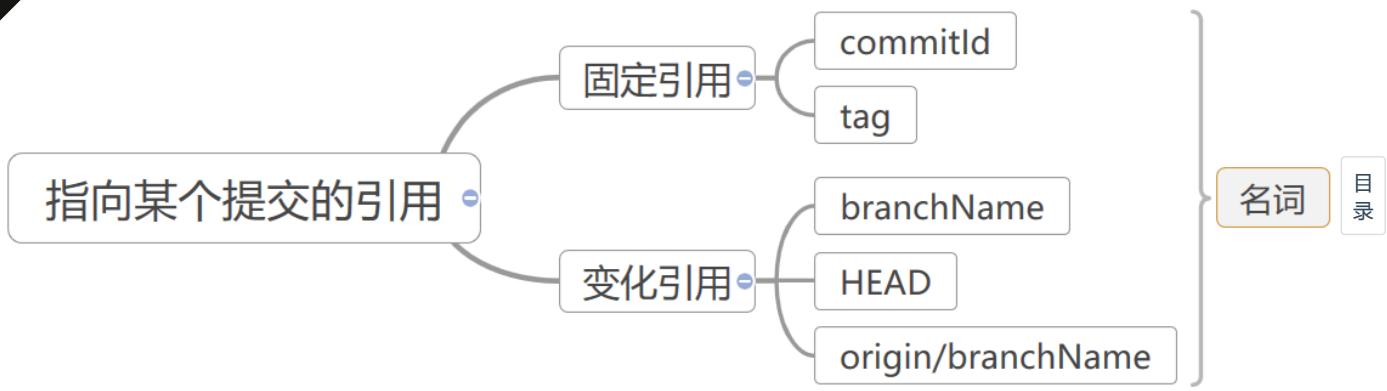
实际上，不需要切到 dev 分支，先更新 dev，则合并。以下命令即可：

Copy

```
(@f/table) git fetch
(@f/table) git merge origin/dev
```

小结：origin/branch 是指向此分支云端最新提交的引用（最新=最后一次更新），在执行 fetch pull push 指令时自动更新。

关注



可以使用 git show 命令查看一个提交的详细信息，
因为 commitId/HEAD/branch/tag/origin-branch 这些都是指向一个提交，所以 show 命令后面写任意一个都可以。
另外，还可以使用其他参数控制显示内容，这里不展开。

```
git show commitId/HEAD/branch/tag/origin-branch --format=short
```

Copy

---

3 一切皆 commit ：动词部分

3.1 cherry-pick

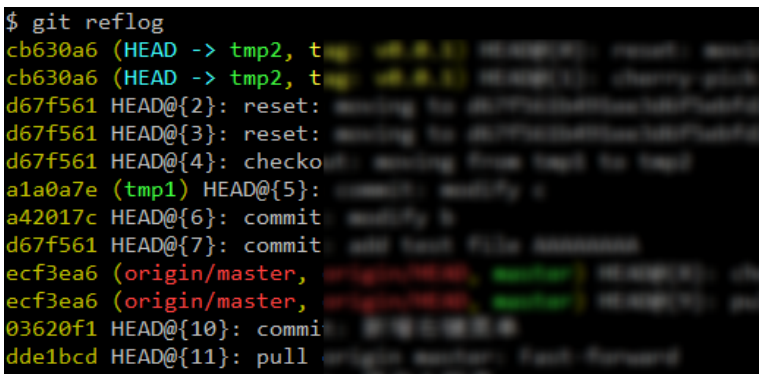
cherry-pick 比较好理解，就是将一个指定提交的修改摘取过来，举例：

如图，6 提交是增加一个有用的 helper 类（间接说明，一个 commit 最好功能独立），但你不想将整个分支合并过来，就可以使用 cherry-pick 命令。使用任何一个指向 6 提交的引用都可以。
需要说明的是，cherry-pick 过来的提交，只是内容与之前的提交一样，他们是两个不同的提交。

案例

做了两个提交的修改，然后删掉分支了，过会发现刚才两个提交有价值，怎么找回来？

Step1 使用 `git reflog` 查看之前的提交历史，找到需要找回的提交ID。



关注

...p2 使用 cherry-pick 命令将需要的提交摘取出来即可。

如何丢失的提交比较多，除了可以批量 cherry-pick 之外，根据实际情况，可以直接在那些提交的最新提交上，新建一个分支，那些提交在此之前的所有提交，都在新的分支上了。

新建分支(03620f1 指提交号/commit id)：

目录

Copy

```
git branch newbranch 03620f1
git checkout -b newbranch 03620f1
```

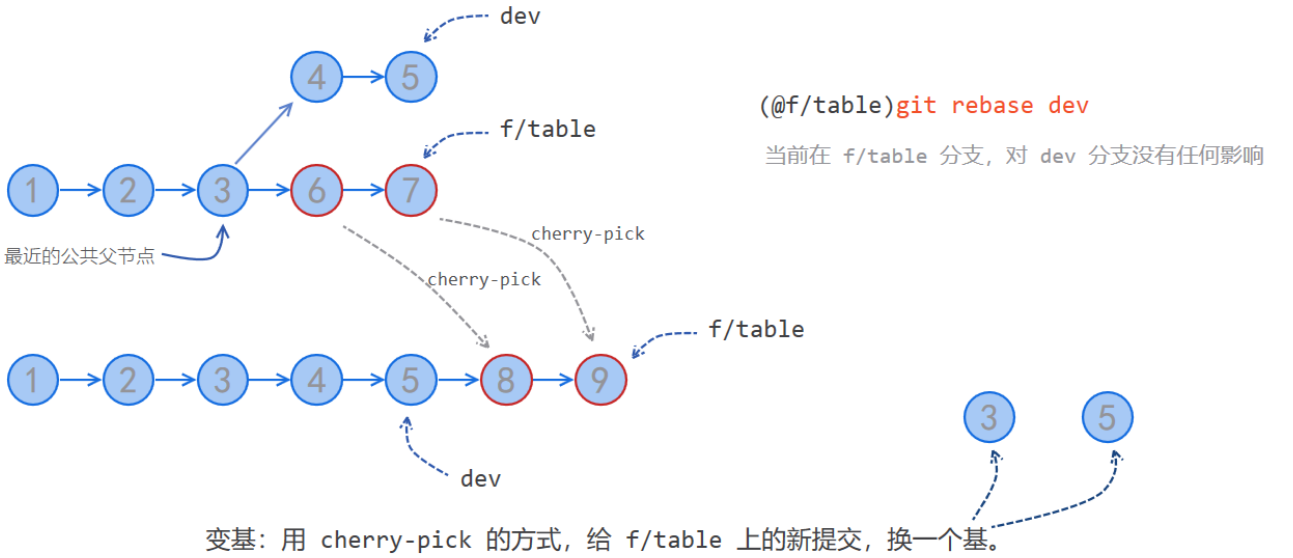
3.2 rebase

如果用一句话理解 rebase 的话，就是：rebase = 一连串自动的 cherry-pick 。

关于 rebase ，需要回答三个问题：

- 为什么推荐使用 rebase 而不是 merge ?
- 为什么听说过使用 rebase 会被打 ?
- 使用 rebase 有什么问题(什么情况不用 rebase) ?

rebase 究竟是什么意思 ?



如上图，假设 dev 上的提交是 1-2-3-4-5，f/table 分支上的提交是 1-2-3-6-7。现在我们需要合并 dev，通常，会使用 `(@f/table)git merge dev` 的方式合并。这里，我们使用 rebase 来合并 dev 。

首先，rebase 会找到 dev 和 f/table 共同的父提交，即 3 提交。然后以 dev 最新的提交为基础，把 f/table 分支上新的提交（这里就是 6 和 7），逐个 cherry-pick 过来。形成新的 f/table 分支。

注意，整个过程中，对 dev 分支不会有任何影响，因为你是在 f/table 上进行的操作。所有，rebase 的中文翻译，变基，就可以理解为：变基：用 cherry-pick 的方式，给 f/table 上的新提交，换一个基，将基从之前的 3 换到了 dev 所指的提交 5 上。

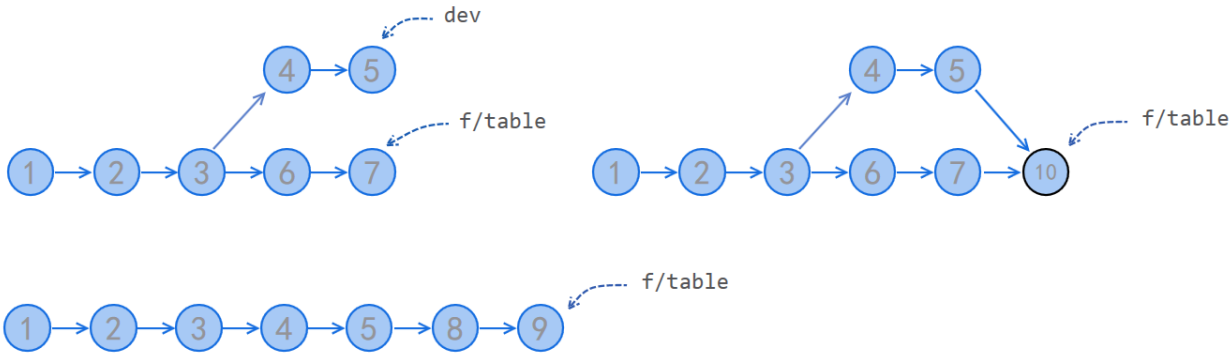
问题1 为什么推荐使用 rebase 而不是 merge ?

Part1 一切皆 commit

rebase

问题1 为什么推荐使用 rebase 而不是 merge

目录

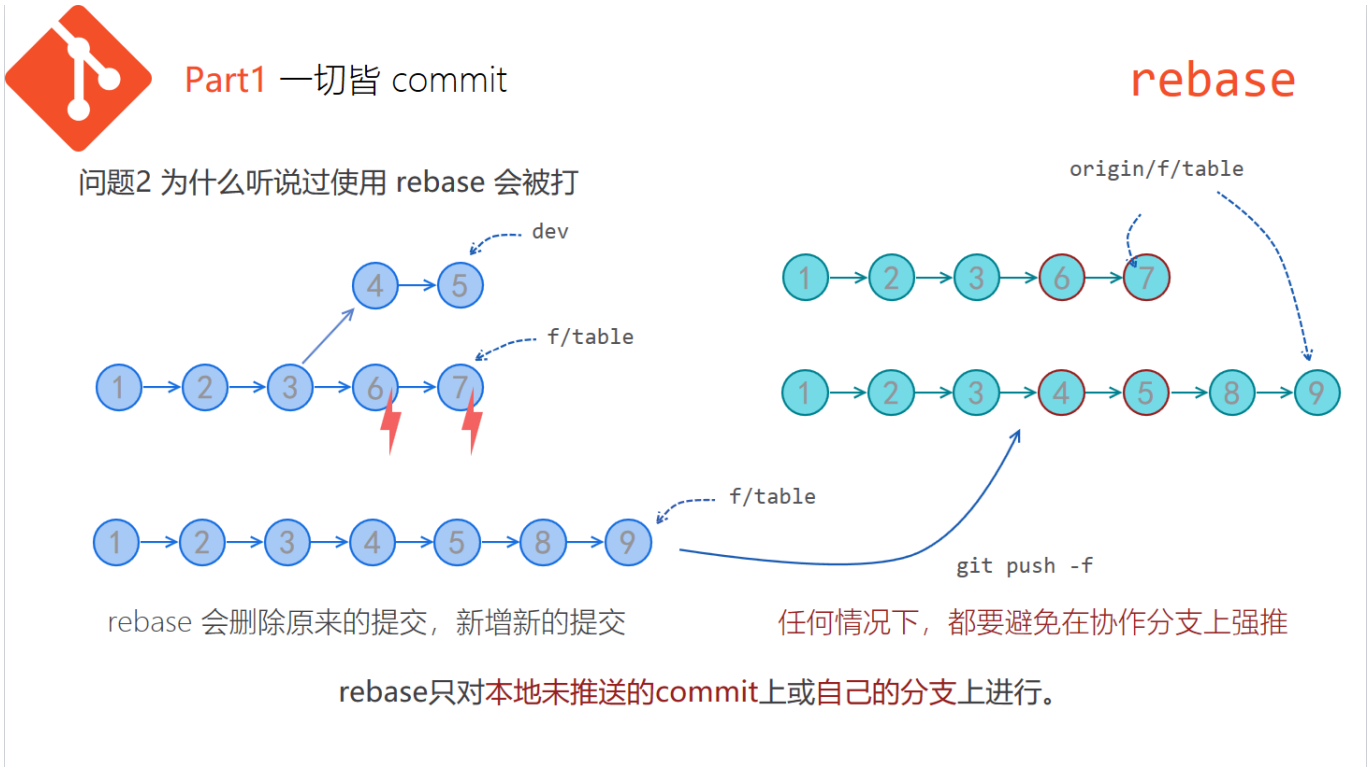


提交历史更清晰，当分支非常多时，回溯提交与查找问题更容易。

当使用 merge 时，提交历史如右侧所示，使用 rebase 的提交历史如下侧所示。
提交历史更清晰，当分支非常多时，回溯提交与查找问题更容易。

问题2 为什么听说过使用 rebase 会被打

使用 rebase 会修改提交历史，上面的例子中，6和7提交将不在 f/table 分支上存在，取而代之的是8和9分支，在协作分支上，如果6和7已经存在于远端仓库（即别人可能已经基于此有了新的修改），再将6和7移除，将带来诸多冲突与合并的麻烦。（这是，你 push 时，也需要强推，在协作分支上强推，是很危险的行为。）
所以：rebase只对本地未推送的commit上或自己的分支上进行。



问题3 使用 rebase 有什么问题(什么情况不用 rebase)

使用 rebase 的收益：更简洁清晰易回溯的提交历史。

使用 rebase 的代价：逐个 cherry-pick ，如果有冲突，需要逐个解冲突，使合并变复杂。

关注

合并 dev 分支为例，当工作分支已经做了大量修改（有很多提交，预期有许多冲突），或者之前 merge 过 dev。则建议使用 merge 的方式合并 dev。

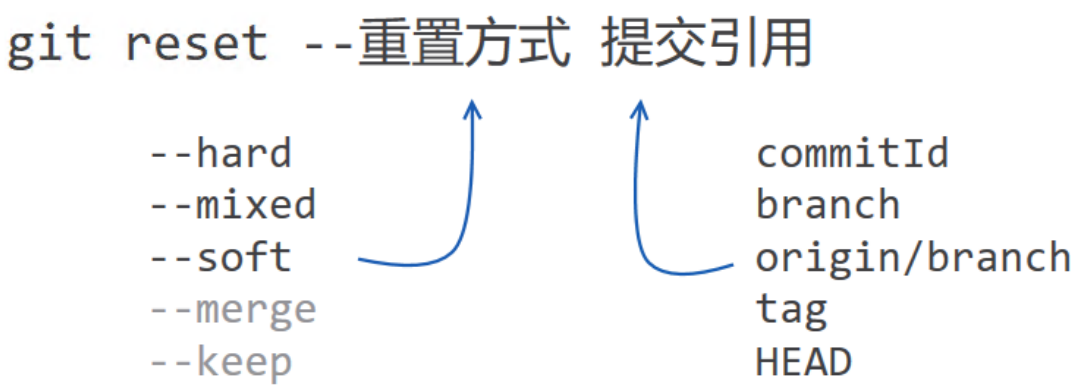
rebase 小结：
rebase：一连串的 cherry-pick。（移花接木）

3.3 reset

目录

reset，重置，将当前分支的状态（这里指工作区，暂存区，代码仓库）重置到指定的状态。reset 的语法如下图，第一个参数是重置方式，后面是一个指向提交的引用（可以是提交ID，分支，tag，HEAD~1等等）。

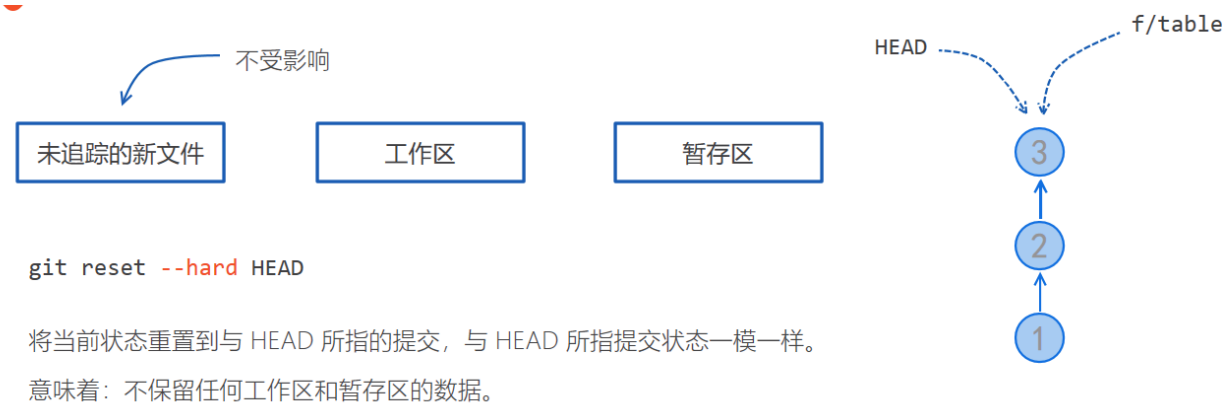
与 rebase 一样，reset 只对当前分支和工作区，暂存区的数据有影响，对参数中指定的引用没有影响。即 `(@f/table)git reset --hard dev` 这句命令，影响的是 f/table 分支，对 dev 没有任何影响。



具体来看：

git reset --hard

从参数名可以猜到，这个重置方式比较“强硬”，实际上就是，将当前分支，重置到与指定引用一样的状态，丢弃在这之后的提交，以及工作区和暂存区的提交。




```
git reset --hard HEAD
```

将当前状态重置到与 HEAD 所指的提交，与 HEAD 所指提交状态一模一样。
意味着：不保留任何工作区和暂存区的数据。

未追踪的文件是不受影响的，PS：git clean 命令会清除掉未追踪的文件。

案例1

(@f/table)git reset --hard f/table~2 的含义？

当前在 f/table 分支，将其重置到 f/table~2 ，结果就是：丢弃掉 f/table 最新的两个提交。

案例2

将当前分支重置到远端最新 dev 的状态，怎么做？

```
(@f/table)git fetch
(@f/table)git reset --hard origin/dev
```

注意，这里需要先 fetch 一下远程仓库，更新 origin/dev 分支。

reset --soft / --mixed

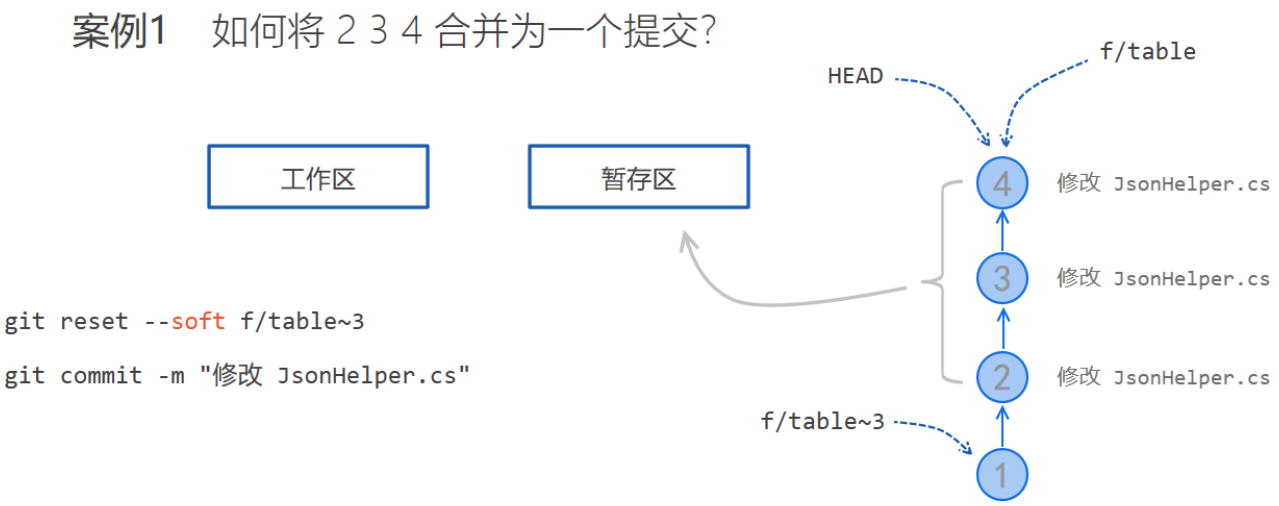
理解了 --hard 的含义，--soft 和 --mixed 就很好理解了，这两个参数，不会丢弃任何内容。

--soft 会将指定提交之后的提交内容，都放到 暂存区，同理，--mixed 会将指定提交之后的提交内容，以及暂存区中的内容，放到工作区。

所以，`git reset --mixed HEAD`（可以简写为 `git reset`），实现的效果就是：将暂存区中的内容，回退到工作区。
`git reset --hard HEAD`（可以简写为 `git reset --hard`），实现的效果就是：将工作区和暂存区中的全部内容。

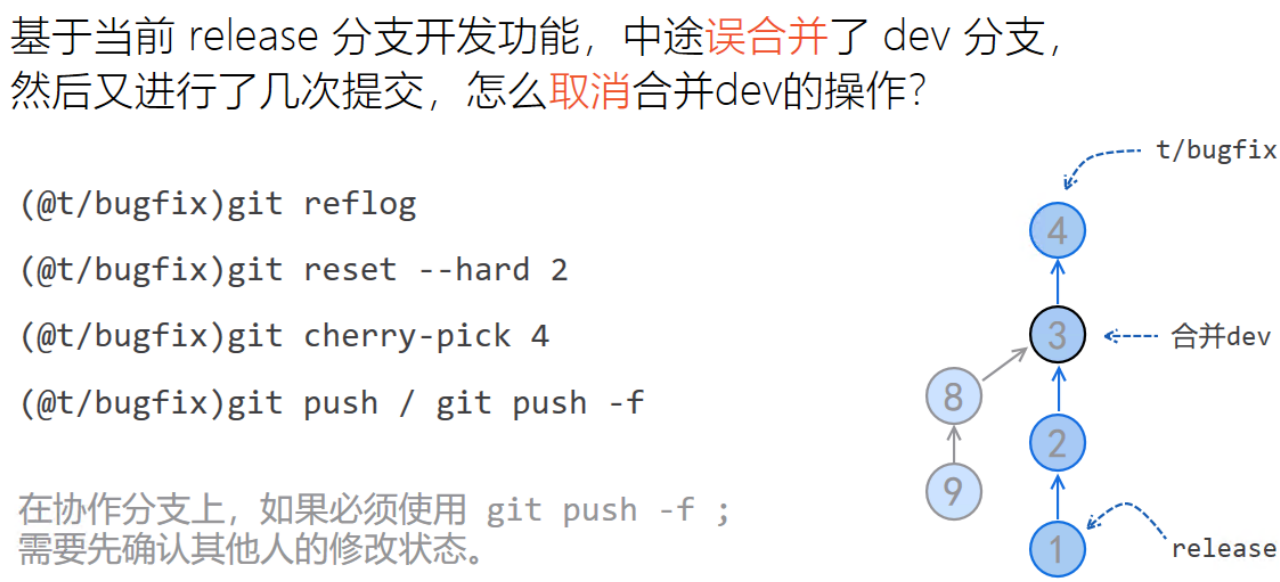
目录

案例1 将图中的 2 3 4合并为一个提交



注意:
这仍然是一种修改提交历史的行为，不要在协作分支上做这样的事情。

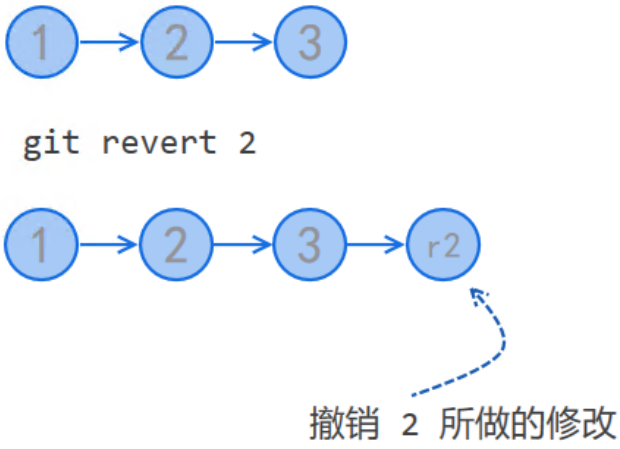
案例2 移除误合并



3.4 revert

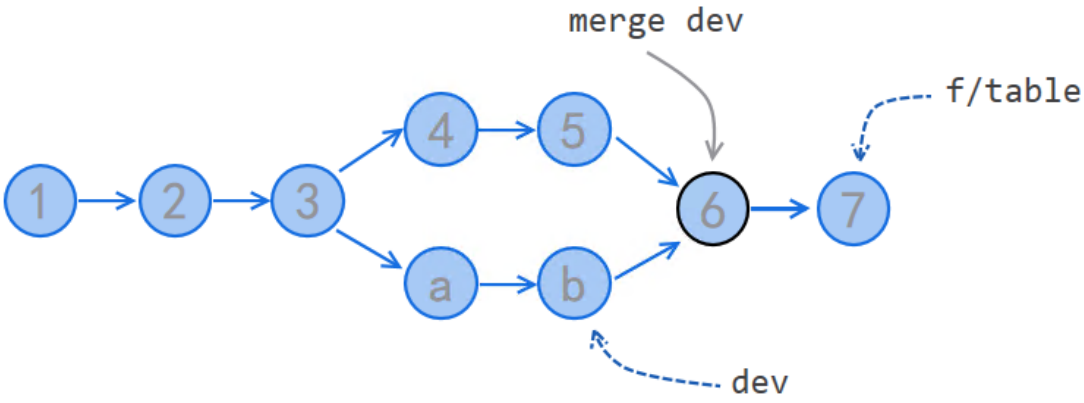
reset 用于修改错误，通常会修改提交历史，
这在团队协作分支上是危险且不允许的（如很多仓库的 master 分支）。
这时可以使用 revert 命令。

revert 很好理解，就是新建一个提交，用于撤销之前的修改。



目录

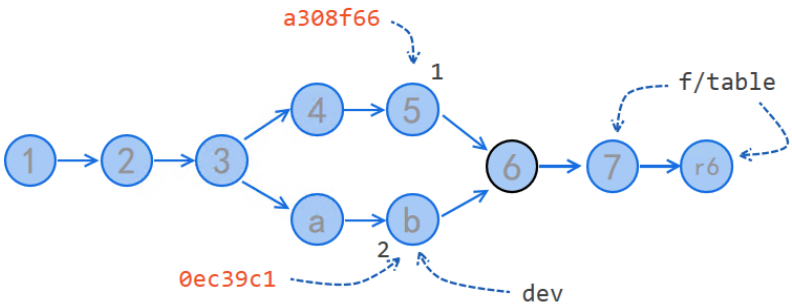
有个问题，revert 一个 merge 提交会怎么样？



如图，如果执行 `(@f/table)git revert 6`
会得到类似这样的提示：

```
$ git revert 8c7e
error: commit 8c7eb8ec9b13452e654d34544f76776c954a4e3e is a merge but no -m option was given.
fatal: revert failed
```

这时，使用 -m 参数可以指定保留那边的提交，可选内容只有 1 和 2（对于通常的两两合并的情况而言），
1 指代当前分支的那些提交，如果不是很确定，可以使用 git show 命令查看那个合并提交，在前的那个父节点为 1 。



(@f/table)git revert -m 1 6

使用 -m 参数，指定想要保留的父分支

使用 git show 命令，查看一个merge结点的父结点，前者为1，后者为2

```
$ git show 8c7e
commit 8c7eb8ec9b13452e654d34544f76776c954a4e3e
Merge: a308f66 0ec39c1
Author: liujunjie <liujunjie@cvte.com>
Date:   Mon Mar 11 15:02:22 2019 +0800

    Merge branch 'tmp'
```

关注

两个思考题：

1 如何在一切皆 commit 的语境下理解 git commit --amend

2 如何在一切皆 commit 的语境下理解 git stash

> 作者：JasonGrass
>
> 出处：https://www.cnblogs.com/jasongrass/p/10582449.html
>
> 本站使用「署名 4.0 国际」创作共享协议，转载请在文章明显位置注明作者及出处。

目录

标签: git

推荐 1 收藏 反对 0

« 上一篇：C# Zip解压缩，规避 [content_types].xml 文件
» 下一篇：深入理解Git - Git底层对象

posted @ 2019-03-23 08:36 J.晒太阳的猫 阅读(133) 评论(0) 编辑 收藏