GIT进阶学习

GIT 进阶

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？

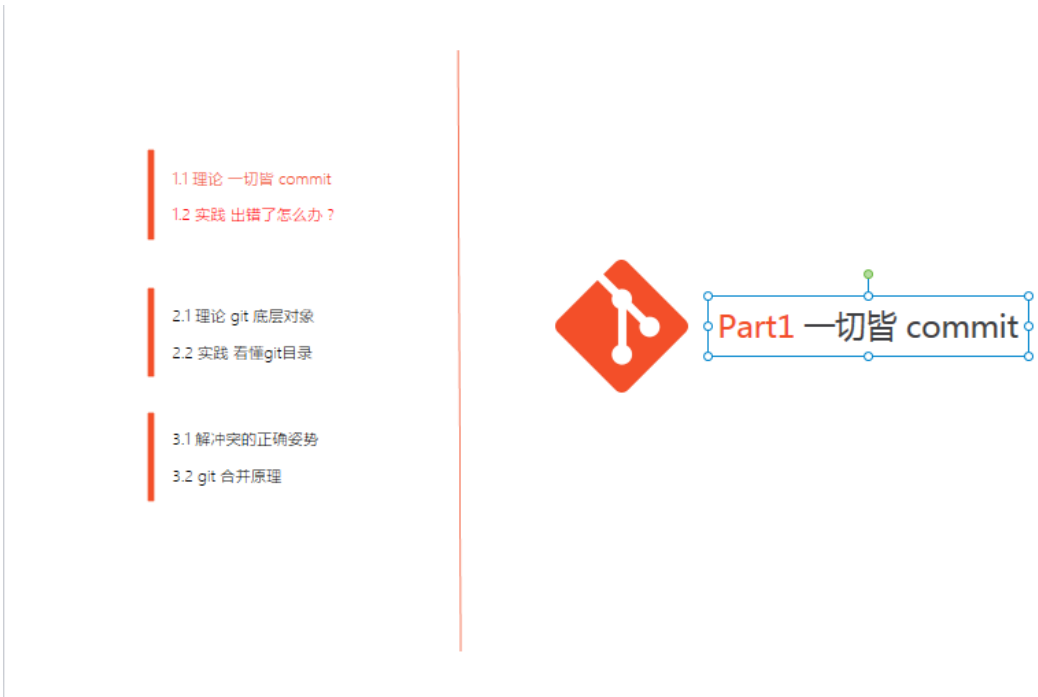
4 解冲突怎么样避免遗漏别人的修改，不留坑？

目标

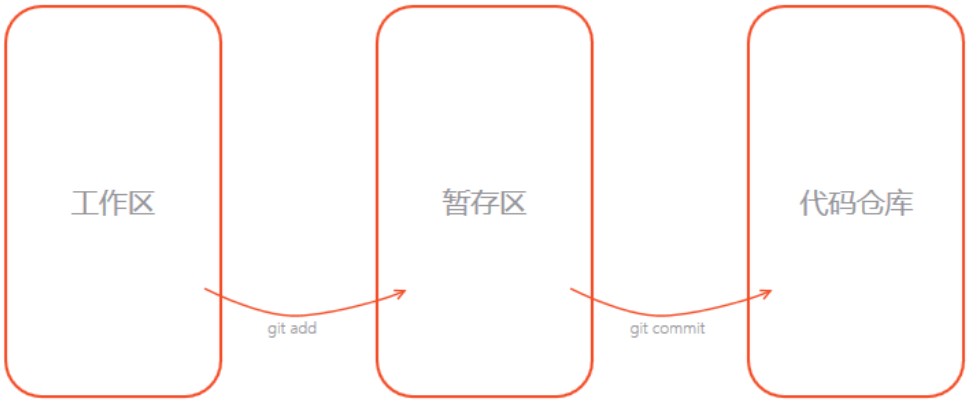
1 理解”一切皆commit”的含义，能够对常见误操作进行修复。

2 理解Git底层对象的含义，看懂部分Git目录中的内容。

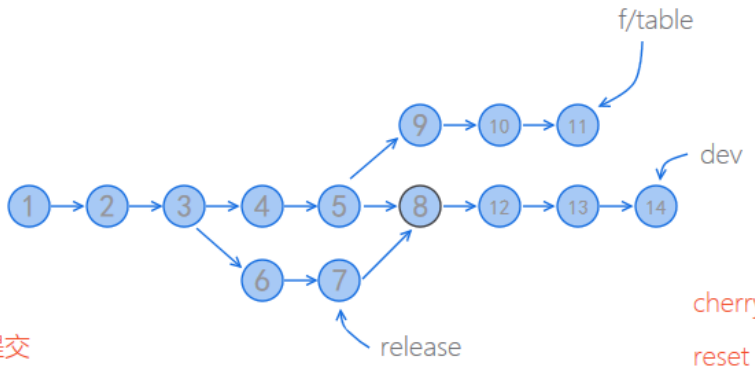
3 掌握解冲突的正确姿势和注意事项。



Part1 一切皆 commit



Part1 一切皆 commit



commit - 提交

branch - 分支

tag - 标签

HEAD - 当前头结点

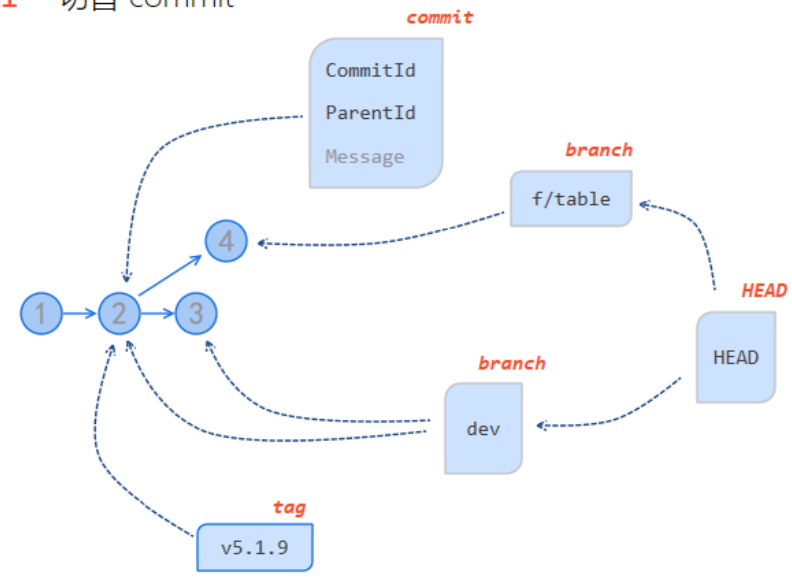
cherry-pick - 摘取

reset - 重置

rebase- 变基

revert - 撤销

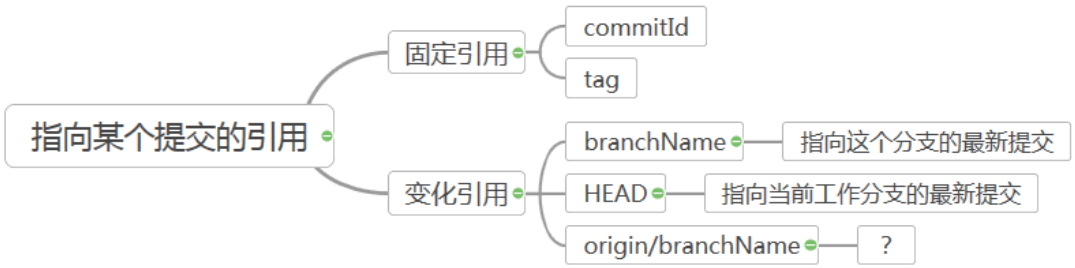
Part1 一切皆 commit

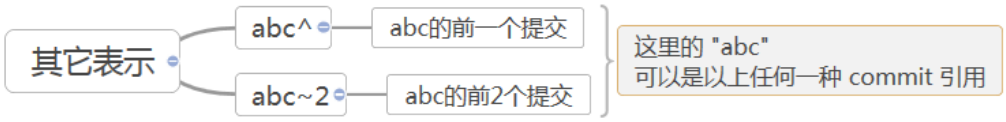


Part1 一切皆 commit


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

Part1 一切皆 commit





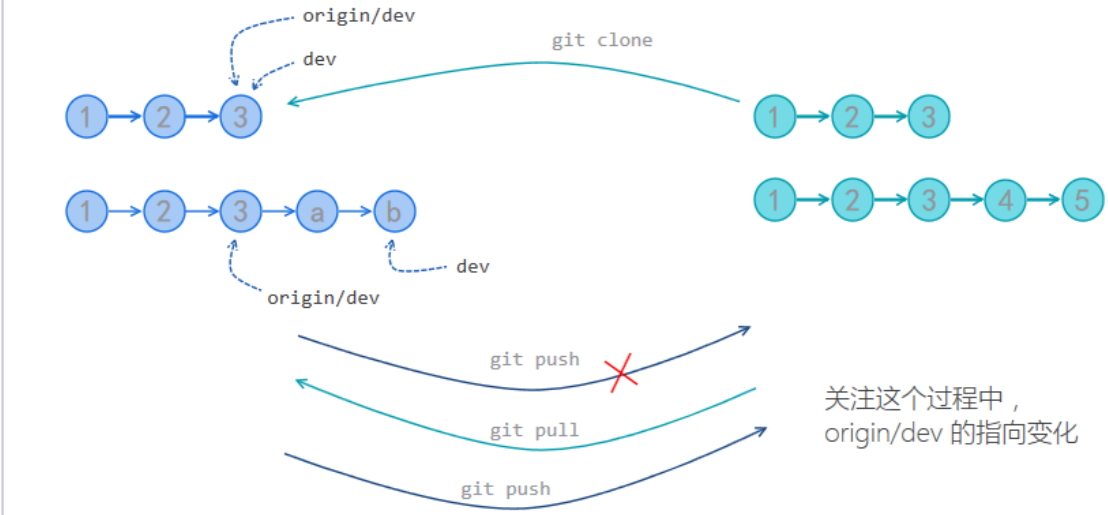
Part1 一切皆 commit

远程仓库

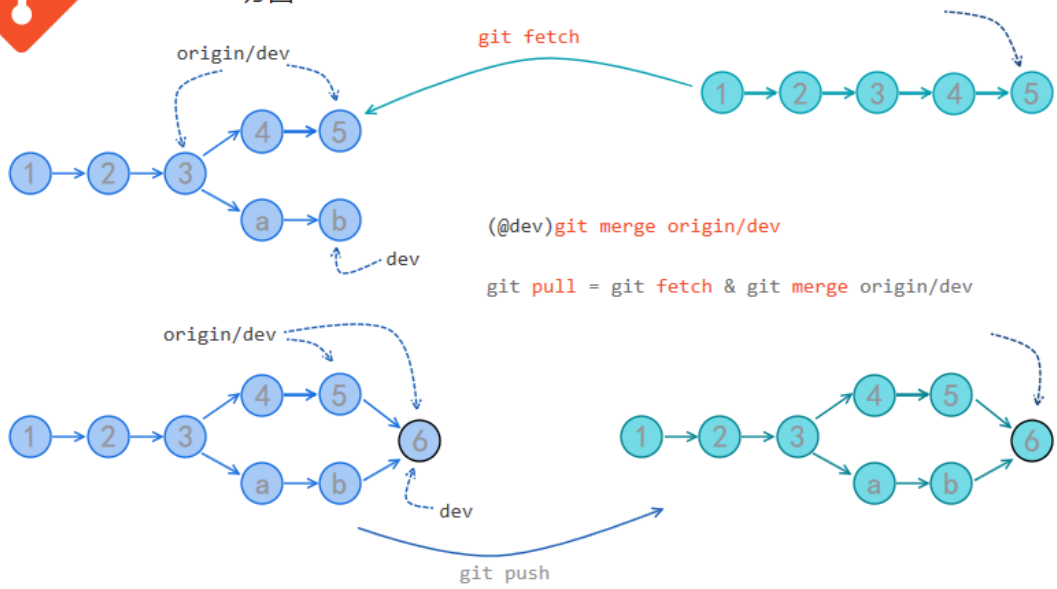
如何在 “一切皆commit” 这句咒语下理解远程仓库

origin/branch 是只读的 commit 引用（内部自动更新）

Part1 一切皆 commit



Part1 一切皆 commit



Part1 一切皆 commit

案例

你在 f/table 分支开发功能，现在需要合并最新dev？

```
git checkout dev
git pull
git checkout f/table
git merge dev
```

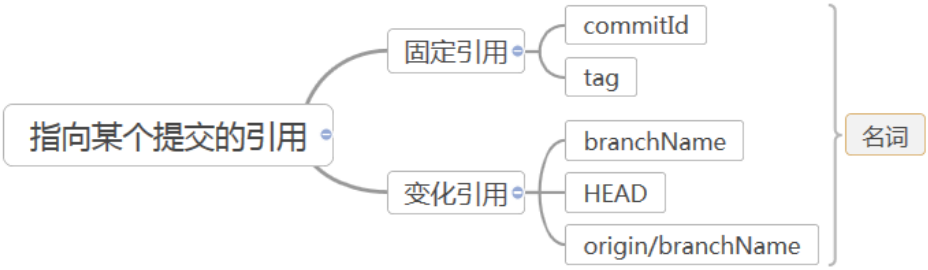
```
git fetch
git merge origin/dev
```

Part1 一切皆 commit

如何在 “一切皆commit” 这句咒语下理解远程仓库

`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



动词部分

Part1 一切皆 commit

动词

cherry-pick

rebase

reset

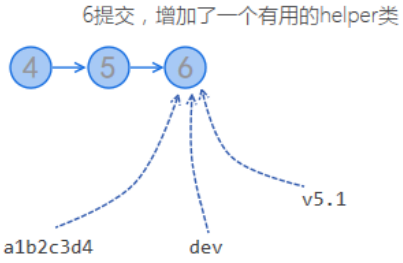
revert

如何在 commit 形成的提交链上实现各种移花接木

Part1 一切皆 commit

cherry-pick

(@f/table)git cherry-pick a1b2c3d4/dev/v5.1



Part1 一切皆 commit

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？

```
git reflog

git cherry-pick 03620f1

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

Part1 一切皆 commit

rebase

为什么推荐使用 rebase 而不是 merge

为什么听说过使用 rebase 会被打

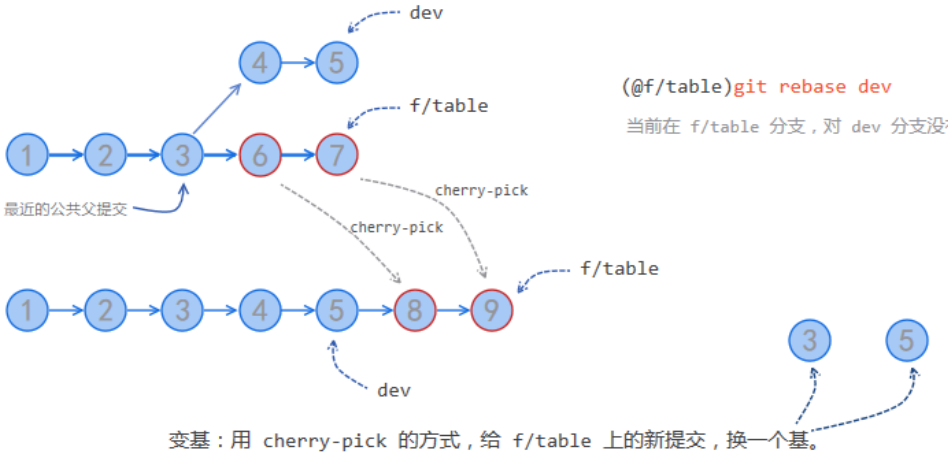
使用 rebase 有什么问题(什么情况不用 rebase)

rebase 是什么，变基？

一连串的 cherry-pick

Part1 一切皆 commit

rebase



(@f/table)git rebase dev

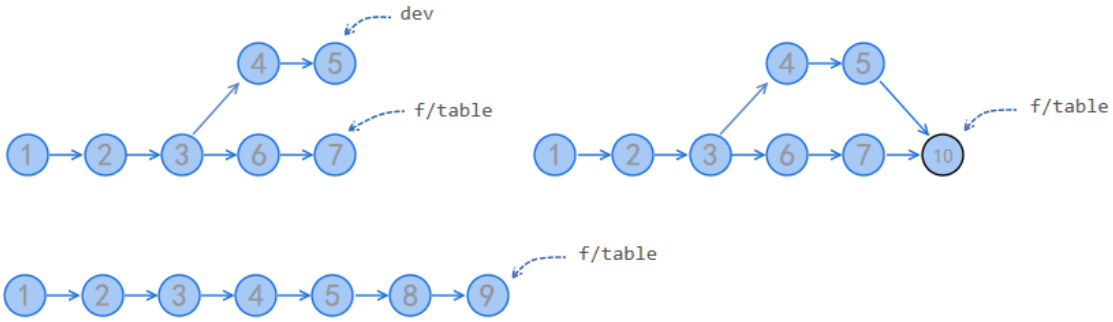
当前在 f/table 分支，对 dev 分支没有任何影响

变基：用 cherry-pick 的方式，给 f/table 上的新提交，换一个基。

Part1 一切皆 commit

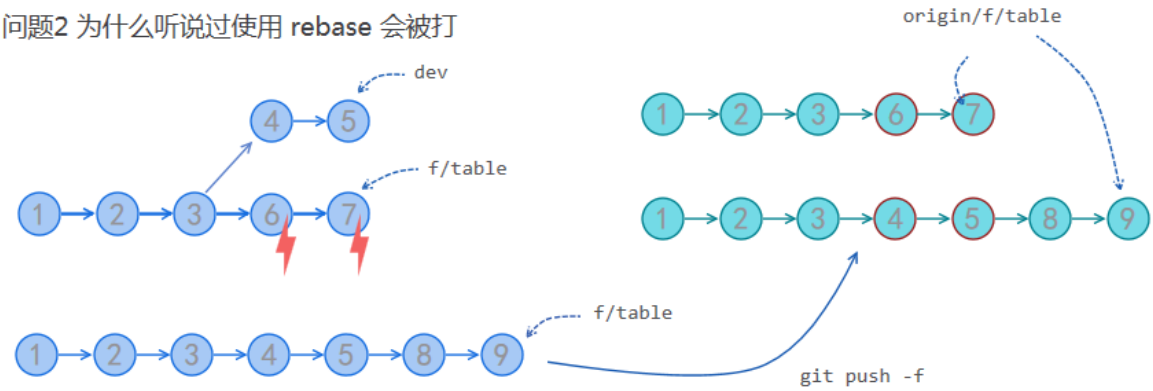
rebase

问题1 为什么推荐使用 rebase 而不是 merge



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



rebase 会删除原来的提交，新增新的提交

任何情况下，都要避免在协作分支上强推

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 ，逐个解冲突，使合并变复杂。

以合并 dev 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）
则建议使用 merge 的方式合并 dev。

案例 什么情况下用 rebase 比较多？

多人在同一个分支上进行开发，如 `f/table` 。

```
(@f/table)git pull
(@f/table)git push
```

```
(@f/table)git fetch
(@f/table)git rebase origin/f/table
(@f/table)git push
```

Part1 一切皆 commit rebase

rebase : 一连串的 cherry-pick

移花接木

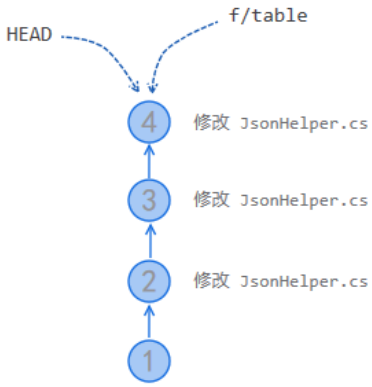
为所欲为的 reset

Part1 一切皆 commit

reset

案例

如何将 2 3 4 合并为一个提交？



Part1 一切皆 commit

reset

```
git reset --重置方式 提交引用
```

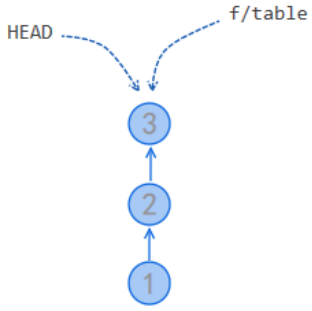
```
--hard                commitId
--mixed               branch
--soft                origin/branch
--merge               tag
--keep                HEAD
```

Part1 一切皆 commit

reset

不受影响

未追踪的新文件 | 工作区 | 暂存区




```
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。

练习

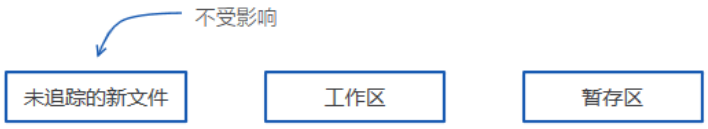
1 (@f/table)git reset --hard f/table~2 的含义？

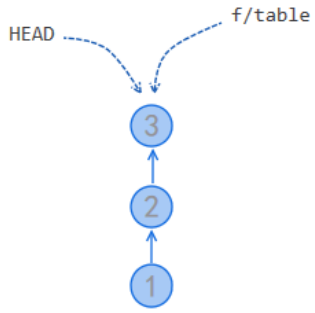
2 将当前分支重置到远端最新 dev 的状态

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

Part1 一切皆 commit

reset






```
git reset --soft f/table~1
```

工作区和暂存区不变，在 f/table~1 之后的提交，都进入暂存区。

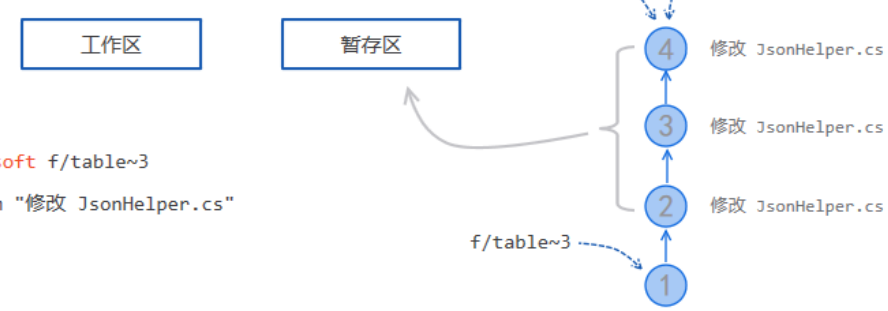
```
git reset --mixed f/table~1
```

工作区不变，在 f/table~1 之后的提交以及暂存区的内容，都进入工作区。

Part1 一切皆 commit

reset

案例1 如何将 2 3 4 合并为一个提交？



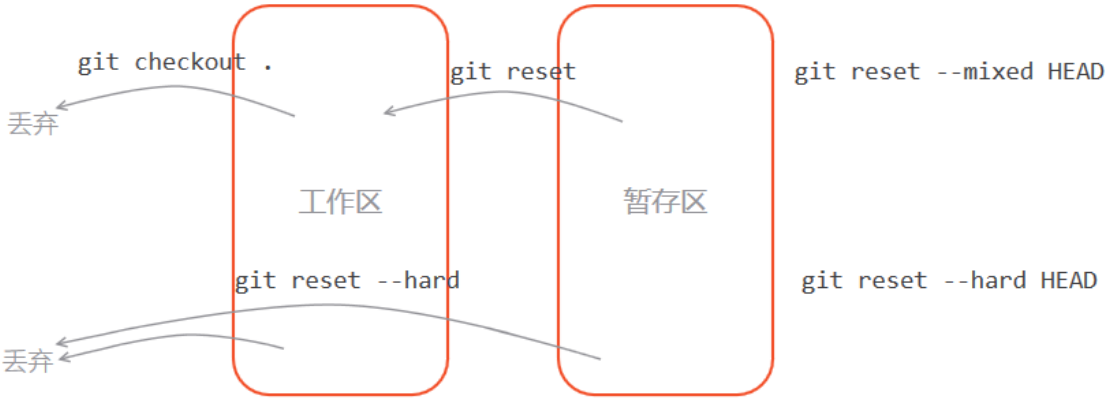

```
git reset --soft f/table~3
git commit -m "修改 JsonHelper.cs"
```

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。

Part1 一切皆 commit

reset

案例2



Part1 一切皆 commit

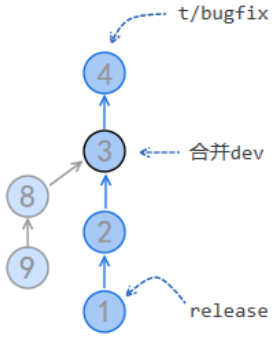
reset

案例3

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？

```
(@t/bugfix)git reflog
(@t/bugfix)git reset --hard 2
(@t/bugfix)git cherry-pick 4
(@t/bugfix)git push / git push -f
```

在协作分支上，如果必须使用 git push -f ；
需要先确认其他人的修改状态。



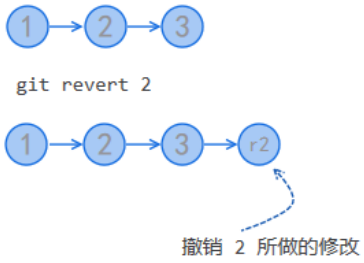
Part1 一切皆 commit

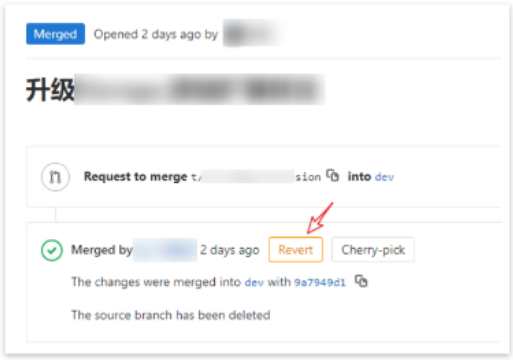
reset 用于修改错误，通常会修改提交历史

这在团队协作分支上是危险且不允许的（如许多仓库的 master 分支）

revert

新建一个提交，用于撤销之前的修改。





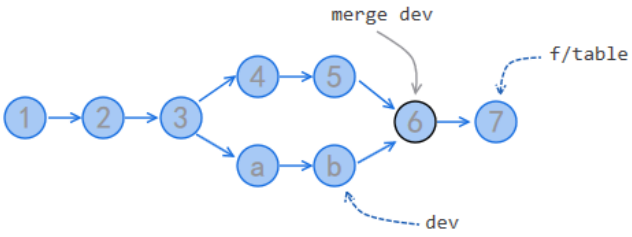
gitlab MR 的 revert 功能

Part1 一切皆 commit

revert

问题

revert 一个 merge 提交会怎么样？

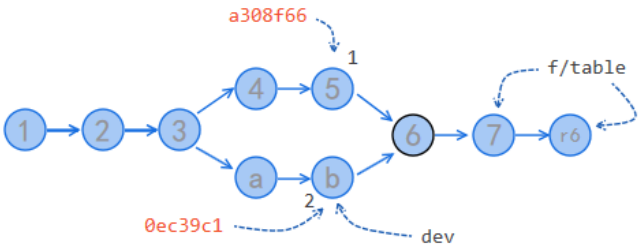


(@f/table)git revert 6

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

Part1 一切皆 commit

revert



(@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'
```

Part1 一切皆 commit

名词

commit - 提交

branch - 分支

origin/branch - 远程跟踪分支

tag - 标签

HEAD - 当前头结点

动词

cherry-pick - 摘取

reset - 重置

rebase- 变基

revert - 撤销

Part1 一切皆 commit

作业

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

Part 2 理解 git 底层对象

Part 2 理解 git 底层对象

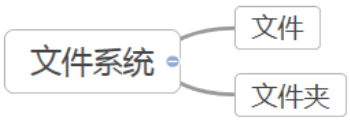
从底层实现理解一切皆commit

看懂部分 .git 目录中的内容

git : 基于键值对的文件系统

Part 2 理解 git 底层对象

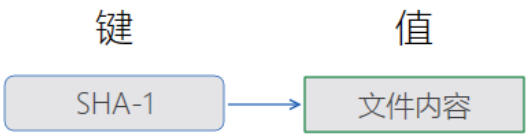
文件系统



Part 2 理解 git 底层对象

blob object（数据对象）

文件

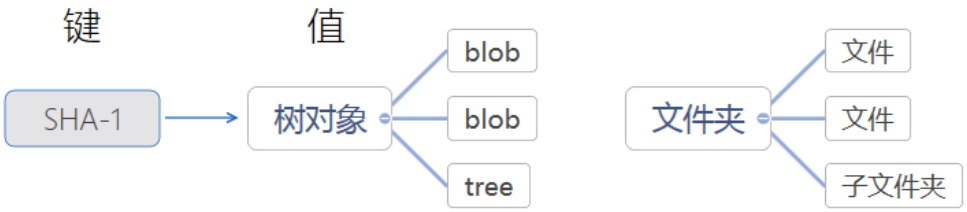


只存内容，不存文件名

Part 2 理解 git 底层对象

tree object（树对象）

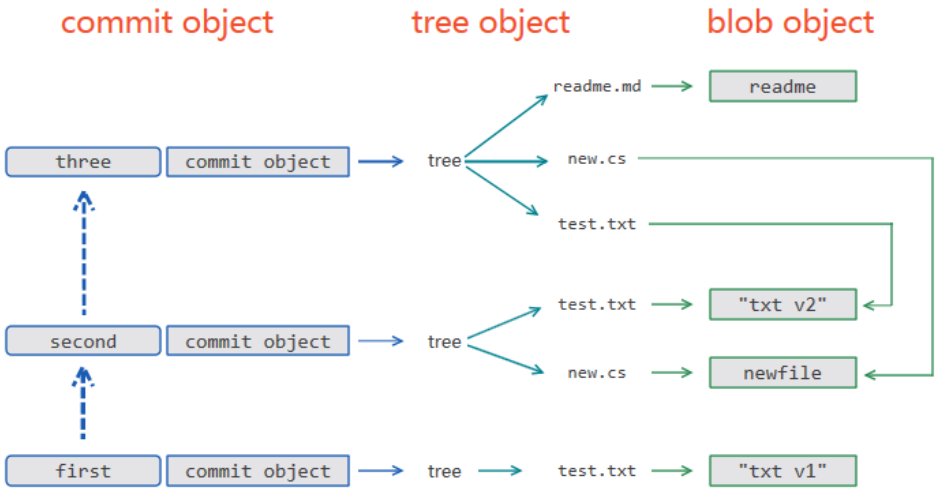
文件夹



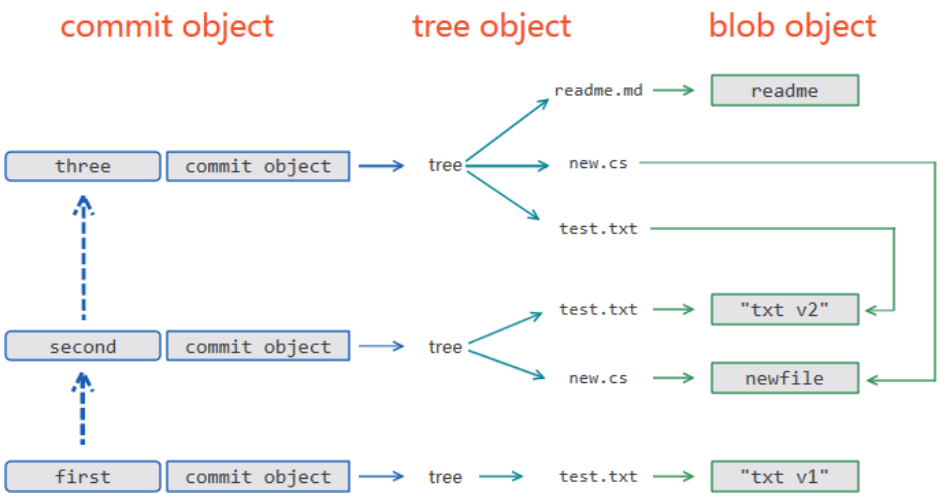
Part 2 理解 git 底层对象

commit object（提交对象）

Part 2 理解 git 底层对象

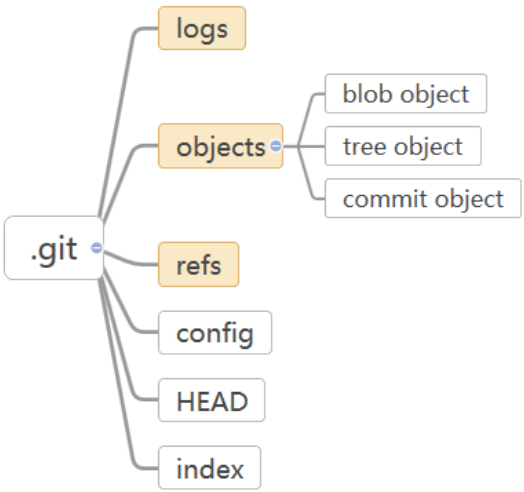


Part 2 理解 git 底层对象



Part 2 理解 git 底层对象

各个对象所在的 git 目录



Part 2 理解 git 底层对象

一切皆 commit

branch　　origin/branch　　HEAD　　HEAD~2

都是指向某个 commit 的引用

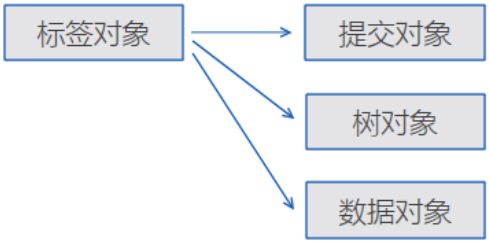
案例

将 t/bugfix 分支重置到 a1b2c3 提交

```
(@t/bugfix)git reset --hard a1b2c3
git update-ref refs/head/t/bugfix a1b2c3
```

Part 2 理解 git 底层对象

tag object（标签对象）



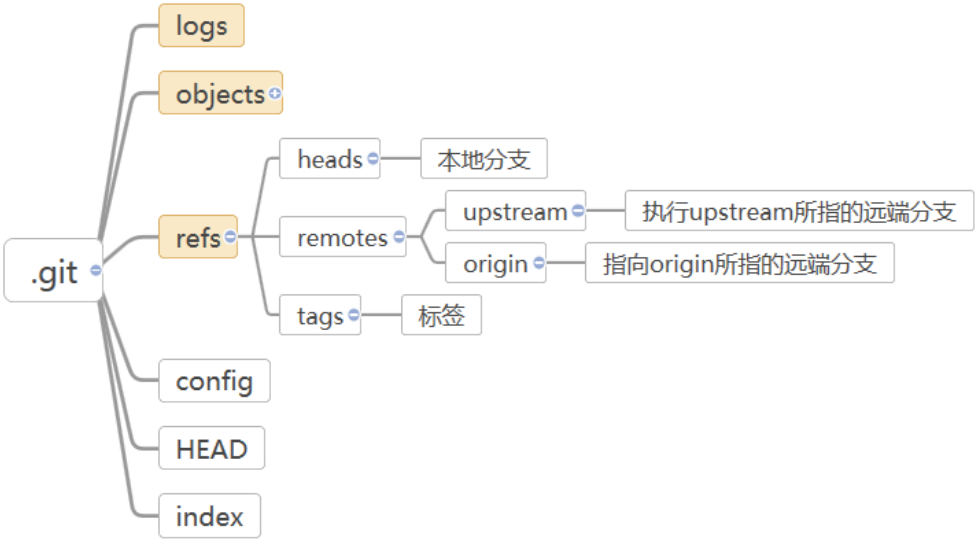
标签对象：指向一个特定对象的固定引用对象

可以给 git 中的任意对象打标签

标签对象不是引用（与分支名不同），是一种独立的git对象。
但在使用上(针对提交的tag)，体验一致。

Part 2 理解 git 底层对象

各种引用所在的 git 目录



Part 2 理解 git 底层对象

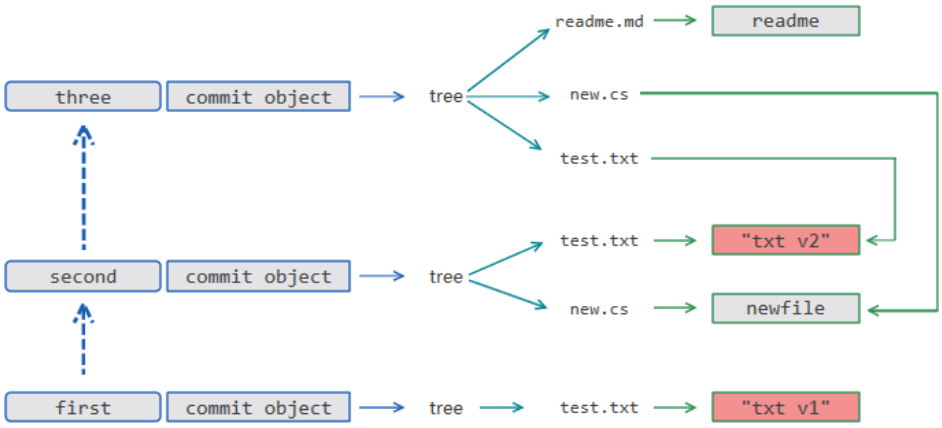
提交历史所在的 git 目录

Part 2 理解 git 底层对象

不是说 git 是增量存储的，怎么实现的？

为什么 git 不建议存大文件，彻底删除大文件要修改整个历史？

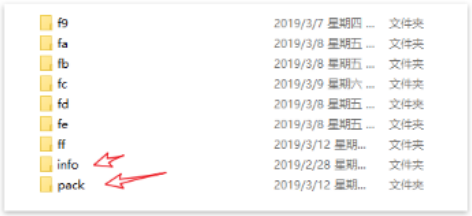
Part 2 理解 git 底层对象



Part 2 理解 git 底层对象

松散的对象模式 —— git gc ——> 包文件

在执行此命令时，使用增量存储的方式进行对象压缩。



Part 2 理解 git 底层对象

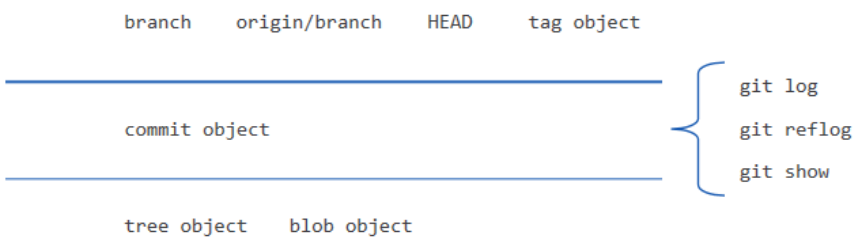
为什么 git 不建议存大文件。

- 大文件（通常是二进制文件），不能得到有效压缩。
过度使用仓库容量将不可控。
- 二进制文件缺少版本控制的使用场景。
- Git LFS (Large File Storage) 可以提供大文件支持。

Part 2 理解 git 底层对象

为什么 git 彻底删除大文件要修改整个历史？

Part 2 理解 git 底层对象



Part 3 冲突

Part 3 冲突

Ⅰ 什么情况下会产生冲突

Ⅱ 冲突的解法

Ⅲ 合并原理

Ⅳ 复杂冲突的解法

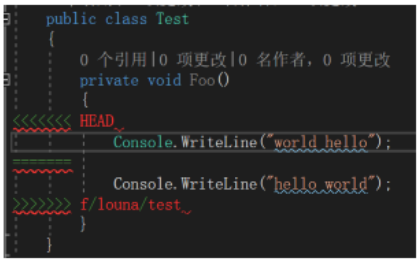
Ⅴ 如何避免冲突

Part 3 冲突

I 什么情况下会产生冲突

双方对同一文件都有修改，且此修改 git 无法"智能"合并。

cherry-pick merge rebase 等所有与内容相关的操作，都有可能产生冲突。



Part 3 冲突

II 冲突的解法

继续，放弃与结束

终止合并

```
git merge/rebase/cherry-pick --abort
```

解完冲突，结束合并

```
git add . & git commit
```

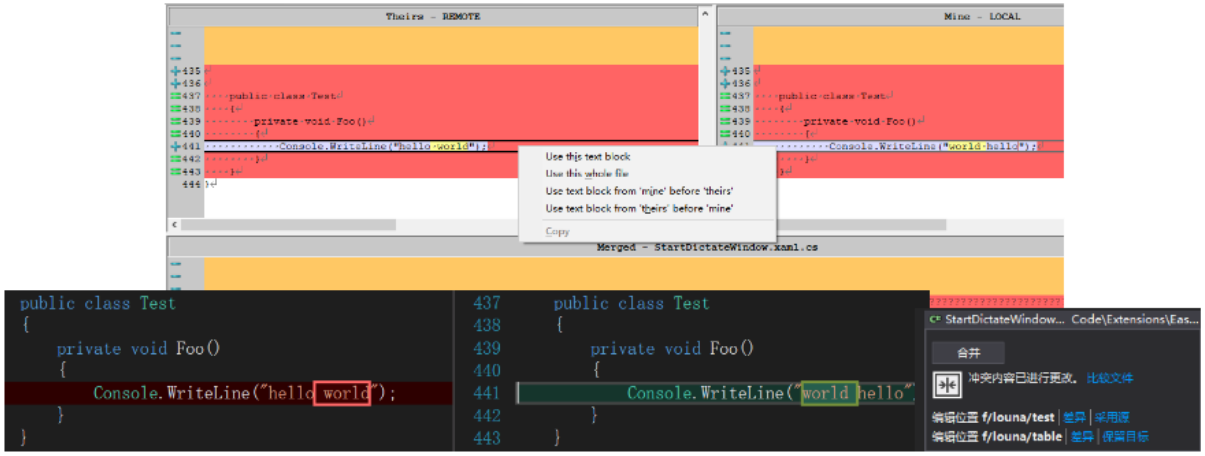
解冲突之后的修改代码在工作区，
合并的代码在暂存区。
需要先 git add .
再 git commit / git merge --continue

继续合并（也可以用于结束合并），一般用于 rebase 等可能多次解冲突的地方

```
git merge/rebase/cherry-pick --continue
```

Part 3 冲突

II 冲突的解法



Part 3 冲突

III 合并原理

对比文件：sha-1算法

- 由文件内容计算出一个40位长度的hash值
- hash值相同，文件内容相同

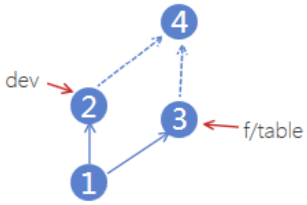
对比内容：

- 二路合并算法

  逐个对比，行内容不同则报冲突

- 三路合并算法

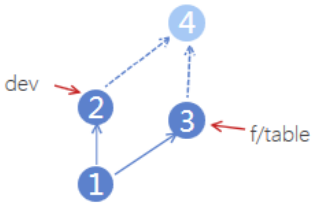
  以基准作对比



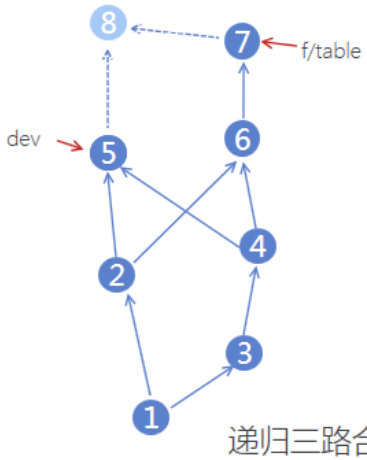
Part 3 冲突

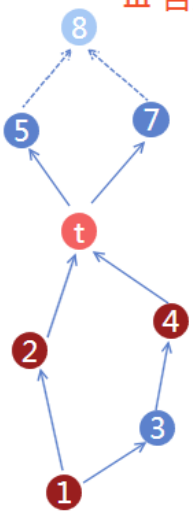
Ⅲ 合并原理

三路合并算法



以基准作对比





递归三路合并算法

Part 3 冲突

Ⅲ 合并策略

```
$ git  merge  branchname  -s(strategy)  合并策略
```

resolve

```
$ (@f/table)  git  merge  dev  -s  resolve
```

三路合并算法，找公共祖先为基准进行合并，如果存在多个公共祖先emmmmm.....
就选择其中一个可能的合并基准点并期望这是合并最好的结果；

recursive -Xours -Xtheirs -Xpatience

```
$ (@f/table)  git  merge  dev  (-s  recursive)  -Xours
```

递归三路合并算法，git默认的合并策略

Part 3 冲突

Ⅲ 合并策略

`$ git merge branchname -s(strategy)` 合并策略

octopus

合并多个分支

`$ (@f/table) git merge branch1 branch2 -s octopus`

ours

使用当前分支作为最终合并结果

subtree

同样使用递归三路合并算法，不同于recursive的是，它将合并的两个分支中的一个视为另一个分支的子树

参考：https://walterlv.com/post/git-merge-strategy.html#subtree

Part 3 冲突

Ⅳ 复杂冲突的解法

略

Part 3 冲突

V 如何避免冲突

想什么呢冲突是避免不了的 

如何减少冲突

- 勤合并
- 最好不要多人同时对一个模块进行操作

比较两边的修改，应用最新的；OR 参照两边的修改，兼顾两边的修改。