Git 入门与基础

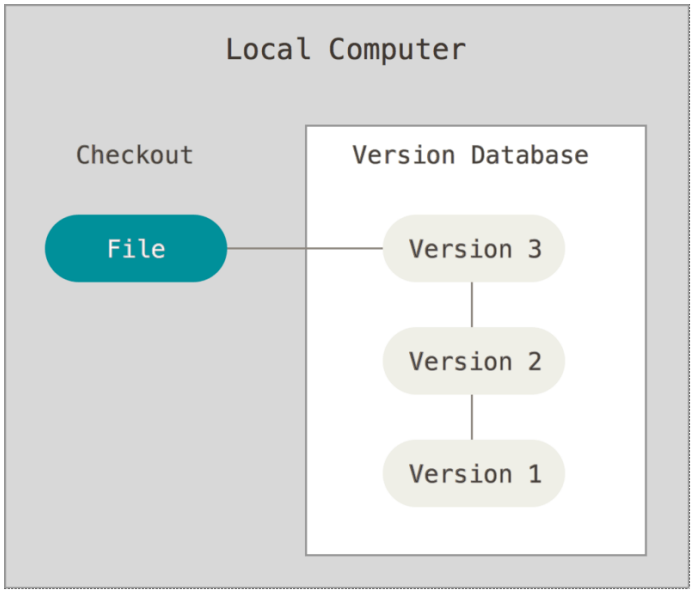
2017年9月25日 16:52

原载于 http://lzw429.site
转载请注明出处。

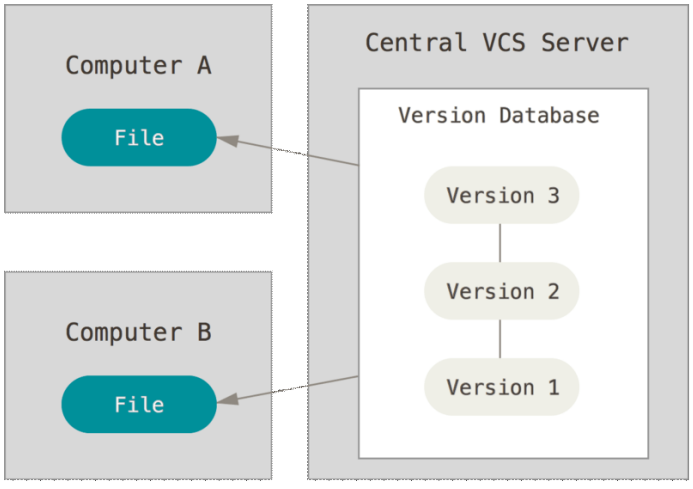
本文将介绍 Git 的基础知识与分支模型，作为 Git 初学者的指导；没有过多地涉及 Git 的原理及进阶知识。

1. 入门
    a. 关于版本控制
    版本控制是维护工程蓝图的标准做法，能追踪工程蓝图从诞生一直到定案的过程；是一种软件工程的技巧，能确保由不同人所编辑的同一代码文件都能得到同步。

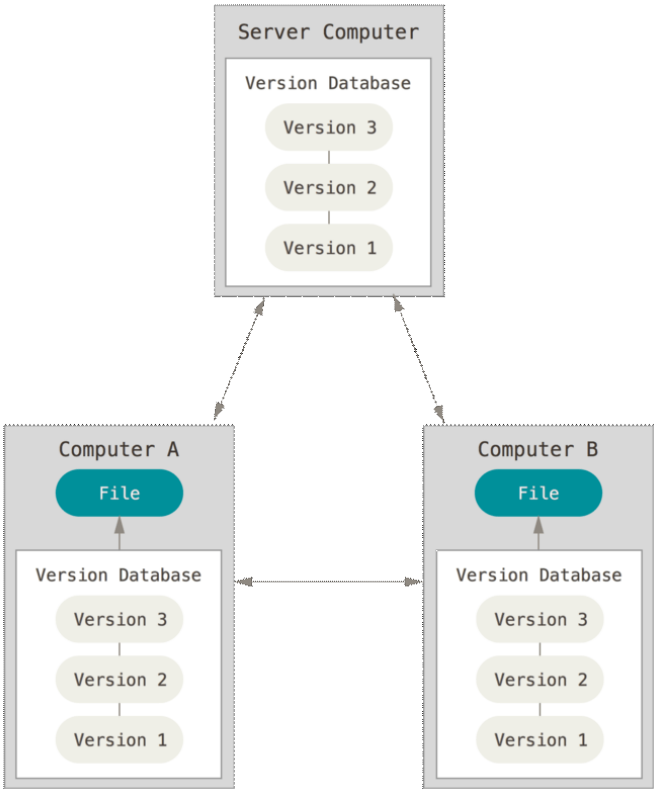
    本地版本控制系统：



    集中式版本控制系统：



分布式版本控制系统：



简单地说，SVN 和 Git 的主要区别是，前者是集中式，不能离线提交修改；后者是分布式，几乎所有操作都能离线完成。

b. Git 的特点

Git 最初是由 Linus Torvalds 创作，最初目的是为更好地管理 Linux 内核开发。

i. 速度
ii. 简单的设计
iii. 良好地支持数千个并行分支
iv. 完全分布式
v. 能够处理如 Linux 内核的大型项目

c. Git 的基础知识

i. 记录快照

Git 存储的不是每个文件与某一版本的差异。在 Git 提交更新或保存项目状态时，它主要把所有文件制作一个快照并保存其索引。如果文件没有修改，不再重新存储，仅保留一个链接指向之前存储的文件。

ii. 保证完整性

Git 中所有数据在存储前都计算哈希值，然后以哈希值来引用。Git 用以计算哈希值的机制是 SHA-1.

iii. 三种状态

1）已提交 committed

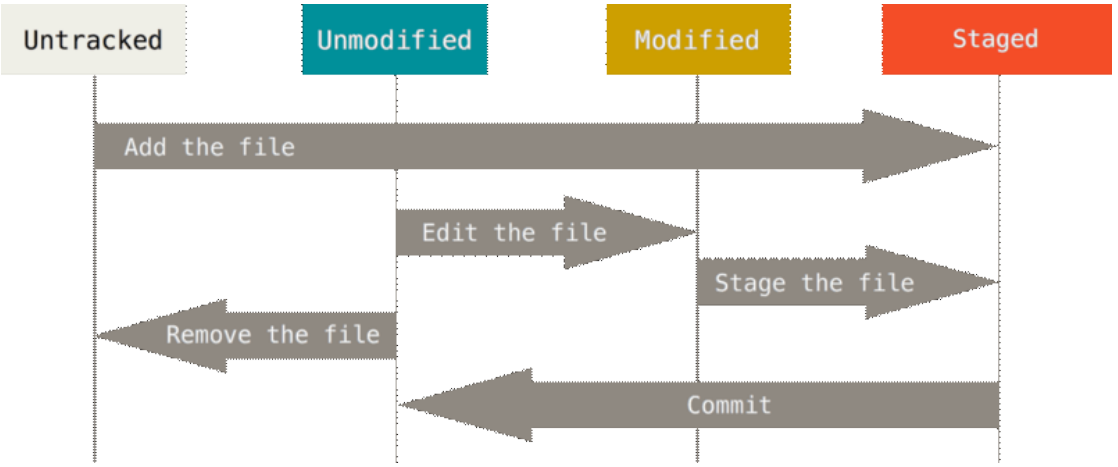
数据已保存在本地数据库中。

2）已修改 modified

修改了文件，尚未保存到数据库中。

3）已暂存 staged

对已修改文件的当前版本做了标记，使之包含在下次提交的快照中。

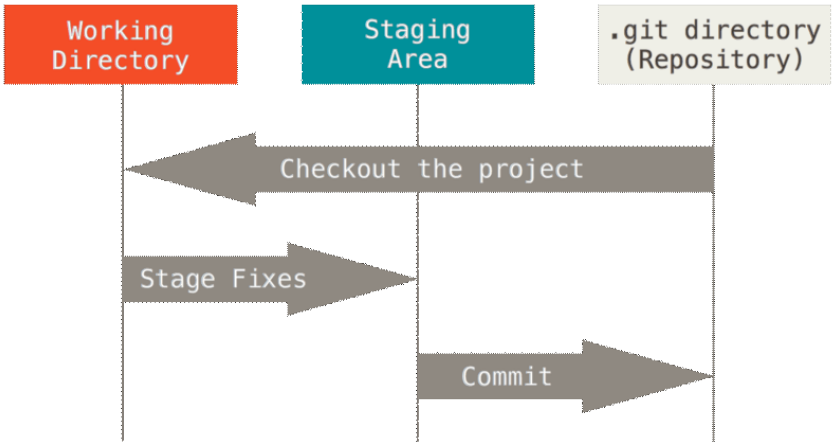


iv. 三个工作区域

1）版本库 .git directory

2）工作目录 Working Directory

3）暂存区域 Staging Area



d. 安装 Git

i. 在 Linux 安装 git，命令行中键入：

Red Hat/Fedora：`sudo yum install git`

Debian/Ubuntu：`sudo apt-get install git`

ii. 在 Windows 安装 git：

https://git-scm.com/download/win

iii. 在 macOS 安装 git：

https://git-scm.com/download/mac

e. Git 初始配置

安装完 Git，首先应该设置用户名与邮件地址。

```
git config --global user.name "name_example"
git config --global user.email example@example.com
```

若使用 -- global 选项，该命令仅需运行一次。当需要针对特定项目使用不同的用户名与邮件地址，可以在那个目录下运行没有 --global 选项的命令来配置。

（可选）设置默认编辑器，例如设置 emacs

```
git config --global core.editor emacs
```

默认通常是vim.

检查配置信息：

```
git config --list
```

f. 获取帮助

三种方法可找到 Git 命令的使用手册：

```
git help <verb>
git <verb> --help
man git-<verb>
```

例如要获得 config 命令的手册，执行

```
git help config
```

⭐2. Git 基础

a. 获取 Git 仓库

i. 创建新的仓库

进入项目目录，命令行键入

```
git init
```

就可以使用 Git 对现有项目进行管理。

该命令将创建一个名为 .git 的隐藏的子目录，包含该仓库的所有必须文件，不要删除！

ii. 克隆现有的仓库

```
git clone https://github.com/lzw429/LeetCode
```

该命令会在当前目录创建一个名为 LeetCode 的目录，初始化 .git 文件夹，并从远程仓库拉取(pull)所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。

```
git clone https://github.com/lzw429/LeetCode <dir_name>
```

该命令除了上述的克隆操作，还会将本地目录的名称修改为<dir_name>.

b. 记录每次更新

所有被纳入版本控制的文件，应该被追踪(track)。

`git add README` 暂存 README 文件

`git add '*.txt'` 暂存所有 .txt 文件

`git add .` 暂存新文件与修改操作，不暂存删除操作

`git add -u` 暂存修改与删除操作，不暂存新文件
`git add -A` 暂存目录下所有文件，相当于 `git add .; git add -u`

注意，运行了 git add 之后又作了修订的文件，需要重新运行 git add 把最新版本重新暂存。

i. 检查当前文件状态
   `git status`

   状态简览
   `git status -s`
   或
   `git status --short`

ii. 忽略文件
   一些自动生成的文件，包括日志文件、编译过程中创建的临时文件是不需要版本管理的，它们也不需要出现在未跟踪文件的列表。
   在目录下创建名为 .gitignore 的文件，列出要忽略的文件即可。

iii. 查看修改
   git status 的信息可能显得比较模糊，有时需要查看文件被更改之处。
   `git diff` 查看未暂存文件的更改
   `git diff --staged` 查看已暂存的将要添加到下次commit里的内容
   `git diff --cached` 同上
   `git diff HEAD` 比较工作区与上次提交时的差异（HEAD 会在稍后提到）
   `git diff <branch_name>` 比较当前工作区与某分支的差异

iv. 提交更新
   暂存区准备妥当后就可以提交。提交前建议使用 `git status` 确认相关文件是否已被暂存，提交更新不会对未被暂存的文件记录快照。命令行键入：
   `git commit`
   这会启动文本编辑器以输入本次提交的说明。
   或者使用-m选项直接键入说明：
   `git commit -m"I'm showing you an example."`
   引号内的即提交说明。

   –a选项会自动将已跟踪过的文件暂存起来并一并提交，从而跳过 git add 步骤。
   `git commit -a -m"I'm showing you the second example."`

v. 移除文件
   从暂存区和工作区移除某个文件：

```
git rm <filename>
```
如果删除之前修改过并且已放入暂存区，需要使用强制删除：
```
git rm -f <filename>
```
这是用于防止误删还没有添加到快照的数据。

从暂存区清除，但保留在工作区：
```
git rm --cached <filename>
```

vi. 重命名文件
```
git mv <file_from> <file_to>
```
这条命令等价于
```
mv <file_from> <file_to>
git rm <file_from>
git add <file_to>
```

c. 查看提交历史
```
git log
```

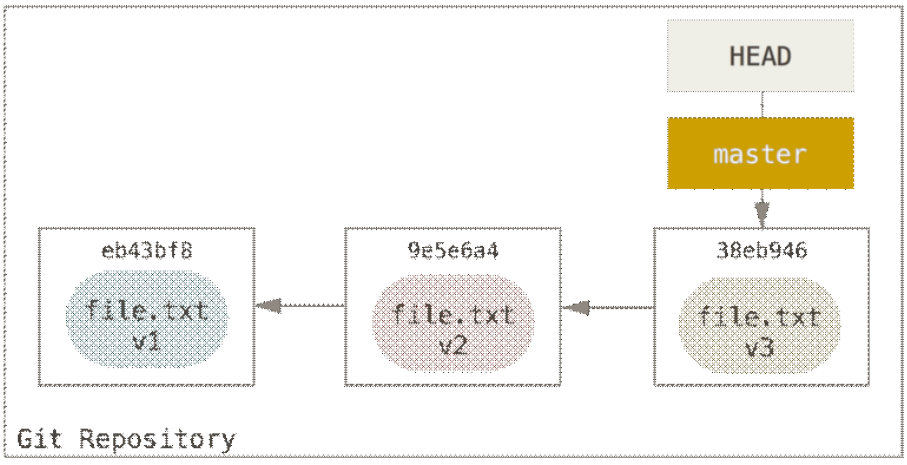
d. 撤销操作与版本回退

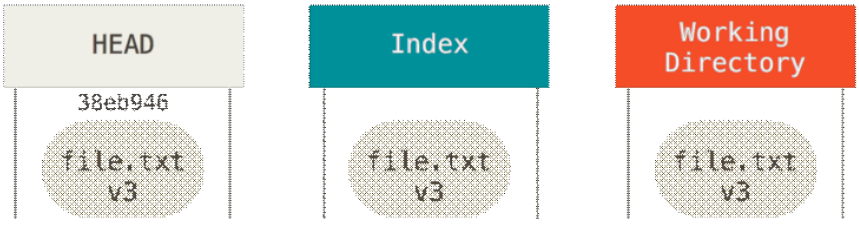
i. 重新提交以代替上一次提交
```
git commit --amend
```
这会覆盖上次的提交信息，而不是增加新的提交。

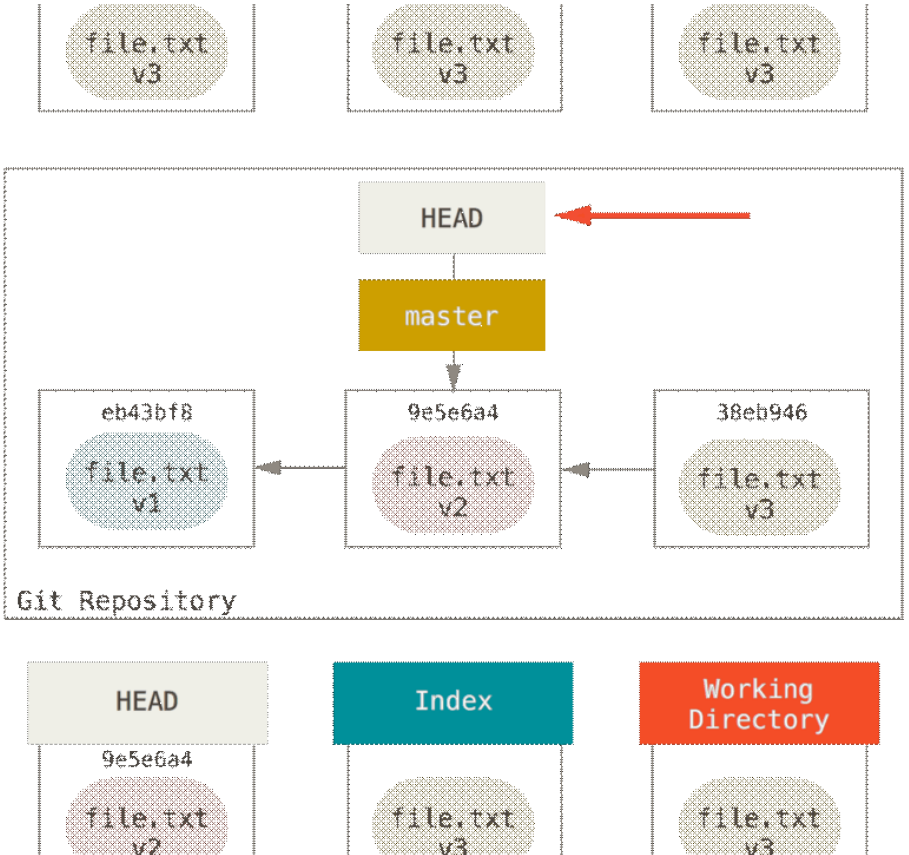
ii. 撤销对文件的修改
```
git checkout -- <file>
```
这会使工作区的文件恢复上次提交时的状态，或刚克隆完，亦或是刚放入工作区时的状态，而不改变 HEAD.

iii. 重置的作用

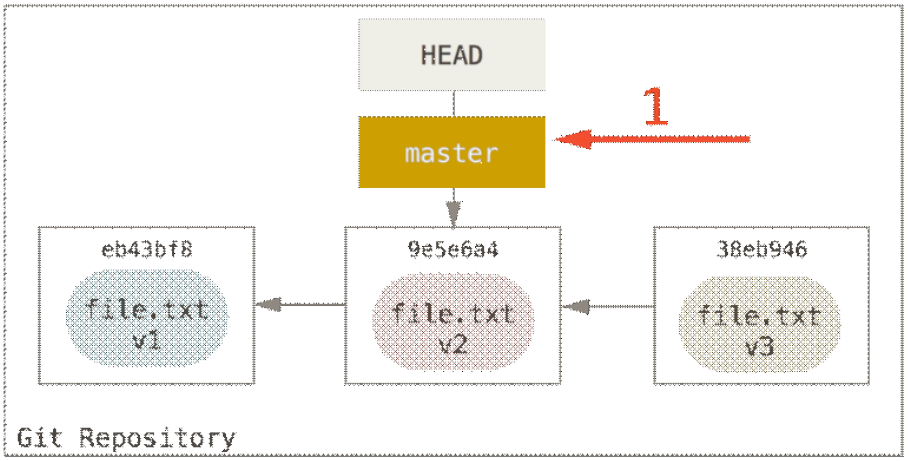
重置的三步：

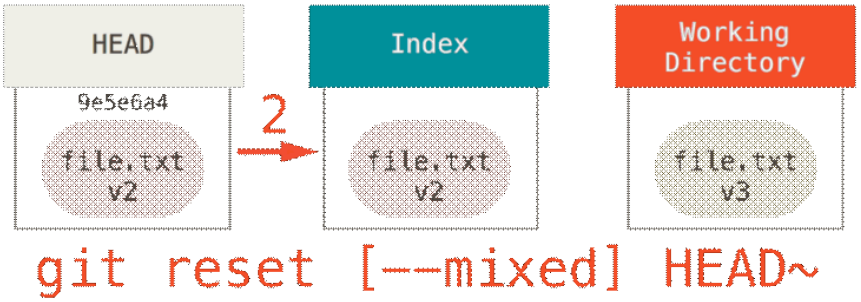




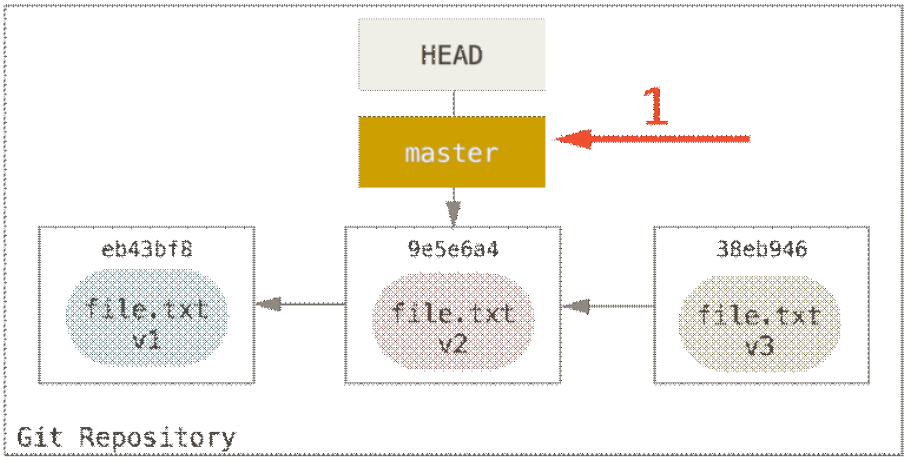


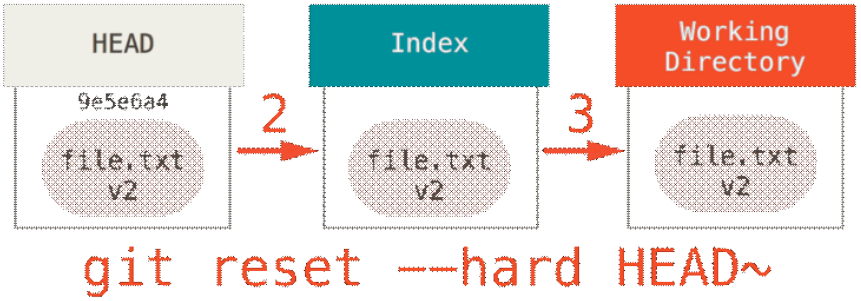
git reset --soft HEAD~





git reset [--mixed] HEAD~





执行到第一步：`git reset --soft HEAD~` 先使 HEAD 指针前移

执行到第二步：`git reset [--mixed] HEAD~` 再用 HEAD 中的提交替换 Index 中的提交

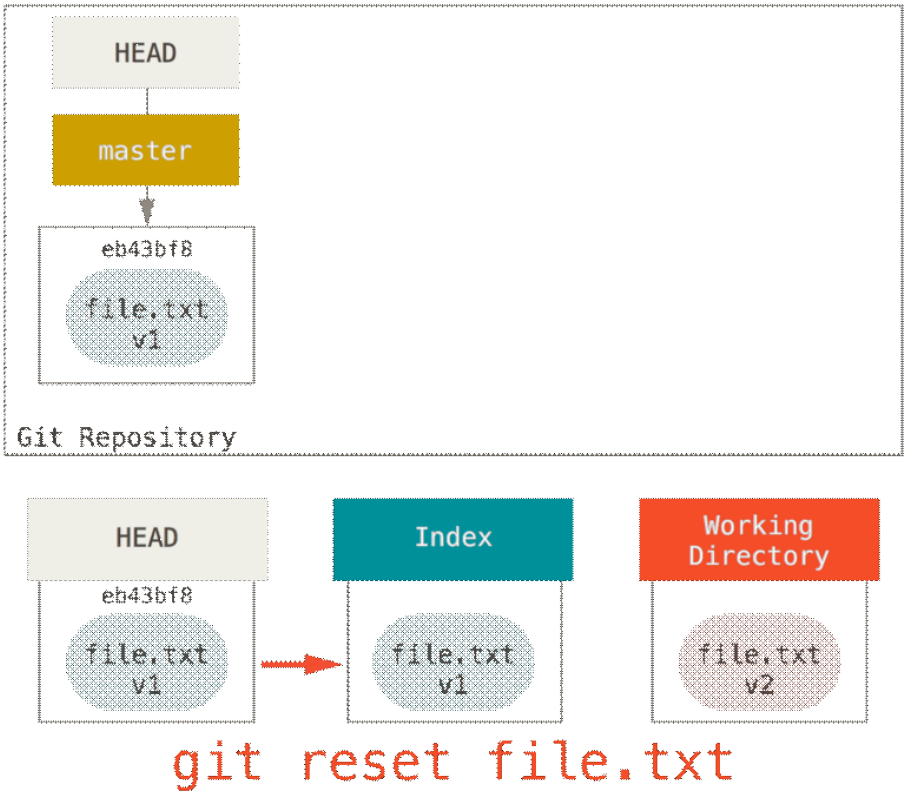
执行三步：`git reset --hard HEAD~` 再用 Index 替换工作区

iv. 通过路径重置

`git reset [--mixed] [HEAD] <file>`

`--mixed` 和 `HEAD` 是默认选项。

该命令的本质只是将 file 从 HEAD 复制到 Index 中。



`git reset <commit_id> --<file>` 将某次提交复制到 Index.

总结撤销操作与版本回退：

	HEAD	Index	Workdir	WD Safe?
Commit Level				
reset --soft [commit]	REF	NO	NO	YES
reset [commit]	REF	YES	NO	YES
reset --hard [commit]	REF	YES	YES	NO
checkout [commit]	HEAD	YES	YES	YES
File Level				
reset (commit) [file]	NO	YES	NO	YES
checkout (commit) [file]	NO	YES	YES	NO

HEAD一列中：
REF表示移动了HEAD指向的分支引用
HEAD表示只移动了HEAD本身

WD Safe表示工作区的安全性

v. 撤销版本回退

版本回退后的提交会覆盖回退前的版本，要恢复回退前的版本可用 `git reflog` 查询哈希值。

该命令将显示所有提交历史，无论版本回退与否。

e. 远程仓库的使用

任何装有 Git 的服务器均可作为远程仓库，而不限于 GitHub.

i. 查看远程仓库

列出远程服务器：

```
git remote
```

origin 是 Git 给该远程仓库的默认名称。

加上 -v 选项可查看远程仓库简写及其 URL.

```
git remote -v
```

ii. 添加远程仓库

```
git remote add <shortname> <url>
```

其中的 shortname 可作为代号表示该仓库。

iii. 从远程仓库抓取与拉取

```
git fetch <remote-name>
```

该命令会抓取 克隆或上一次抓取后 新推送的所有工作，它并不会自动合并或修改当前的工作，而需要你手动合并。

```
pull = fetch + merge
```

合并(merge) 将在分支一章中提到。

iv. 推送到远程仓库

```
git push <remote-name> <branch-name>
```

例如，`git push origin master.`

v. 查看远程仓库

```
git remote show <remote-name>
```

可查看远程仓库的 URL 与跟踪分支的信息。

vi. 远程仓库的移除与重命名

```
git remote rename <old_name> <new_name>
git remote rm <name>
```

f. 打标签(tags)

i. 列出已有标签

```
git tag
```

将以字母顺序列出标签，而不是创建标签的时间顺序。

ii. 创建标签

Git 主要有两种标签，**轻量标签**（lightweight）与**附注标签**（annotated）。

轻量标签像一个不会被改变的分支，只是一个特定提交的引用。

附注标签是存储在 Git 数据库中的完整对象，是可被校验的；其中包括打标

签者的名字、电子邮件地址、日期时间等信息。

iii. 附注标签

创建附注标签的命令：

```
git tag -a v0.99 -m 'my version 0.99'
```

-a 即 annotated.

这时执行 `git show v0.99`

会显示该标签对应的提交信息、打标签者的信息、日期时间、附注时间等。

iv. 轻量标签

创建轻量标签不需要选项，例如

```
git tag v0.99
```

同样地，`git show v0.99`

可查看该标签的提交信息。

v. 后期打标签

在通过 `git log` 找到过去的提交对应的哈希值后，在前述的打标签命令的末尾加上该哈希值即可为过去的提交打标签，例如：

```
git tag v0.99 9fceb02
```

其中 9fceb02 是某次过去的提交的哈希值。

vi. 共享标签

一般地，git push 并不会传送标签到远程仓库上。在创建完标签后必须手动推送标签到共享服务器上。

```
git push <remote-name> <tag-name>
```

传送所有不在远程仓库上的标签：

```
git push <remote-name> --tags
```

vii. 查看标签

```
git checkout <tag-name>
```

如果想要在某标签上做改动，则需要在此标签上创建一个新分支。

```
git checkout -b <branch-name> <tag-name>
```

注意，如果之后又进行了改动，那么这个新的分支与标签又有所不同了。

viii. 删除标签

```
git tag -d <tag-name>
```

如果标签已被推送到远程，还需要执行以下命令删除远程仓库中的标签：

```
git push <remote-name> :refs/tags/<tag-name>
```

g. 给命令设置别名

例如，

```
git config --global alias.co checkout
git config --global alias.br branch
```

可将命令中的 checkout 替换为 co，branch 替换为 br……

3. Git 分支(branch)

真正地了解 Git 的精髓，就在此处。

a. 分支简介

前面提到，Git 保存的不是文件的差异，而是一系列不同时刻的文件快照。

在进行提交时，Git 会保存一个**提交对象**(commit object)，包含一个指向暂存内容快照的指针，作者的姓名、邮箱，提交时的信息以及指向它的父对象的指针。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，由多个分支合并产生的提交对象有多个父对象。

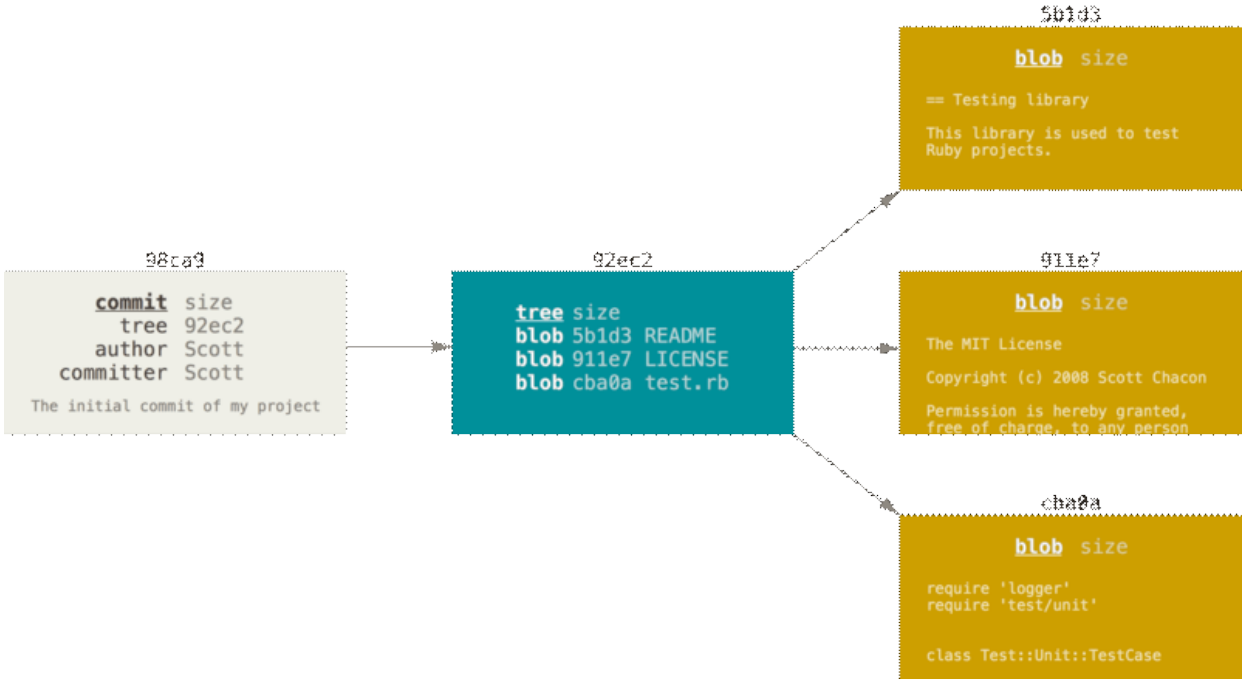
一个提交三种对象：

i. blob(binary large object)：保存文件快照

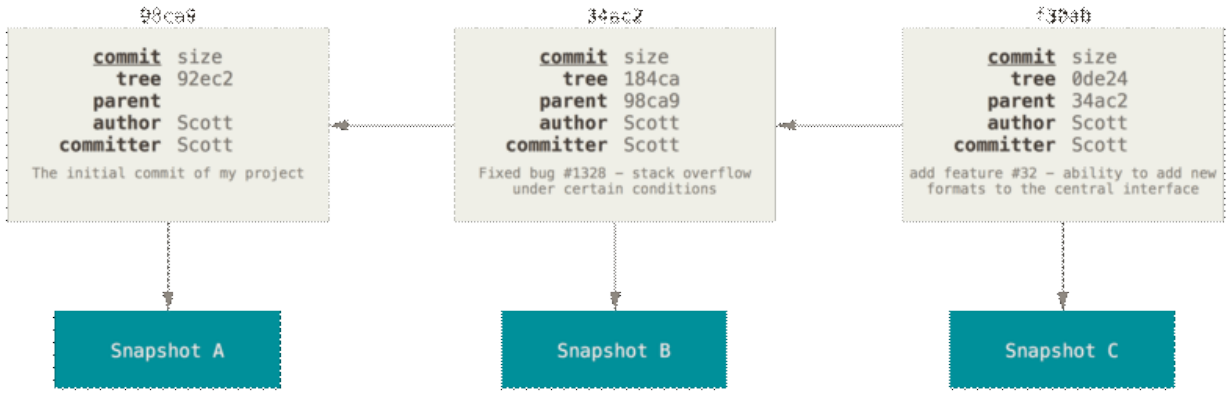
ii. 树对象：记录目录结构和 blob 对象索引

iii. 提交对象：包含指向树对象的指针和所有提交信息（作者、日期等）

例如，一个有三个文件的代码库的五个对象（三个 blob 对象，一个树对象和一个提交对象）如图：



修改后再次提交，提交对象会指向上次的提交对象，即父对象。



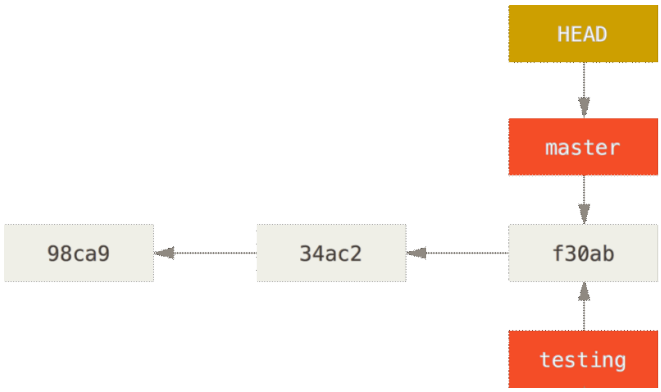
Git 的分支，本质上是指向提交对象的可变指针。

⭐ 默认的分支名称是 master，在多次提交后，master 分支仍然指向最后的提交对象，是因为每次的提交操作会使它自动向前移动。

b. 创建分支

`git branch testing`

该操作在当前所在的提交对象上创建了一个指针。



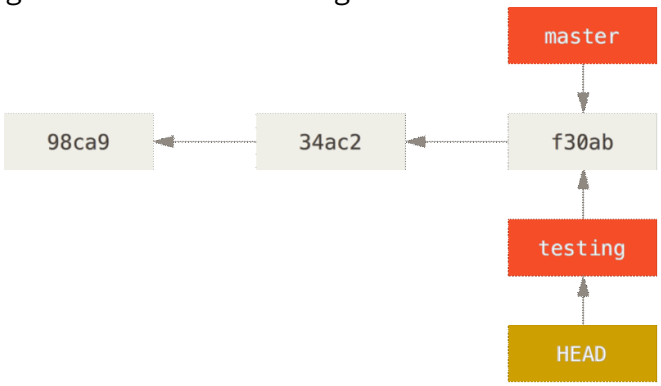
⭐ 创建分支不意味着切换到新的分支，当前工作区的内容是由 HEAD 这个特殊的指针决定的。

使用 `git checkout -b testing` 可以同时创建并切换到 testing 分支。

使用 `git log --oneline --decorate` 可查看各个分支与 HEAD 指针当前所指的提交对象。

c. 分支切换

`git checkout testing`



与查看标签的操作类似。这使得 HEAD 指针指向了 testing.

d. 删除分支

```
git branch -d <branch-name>
```

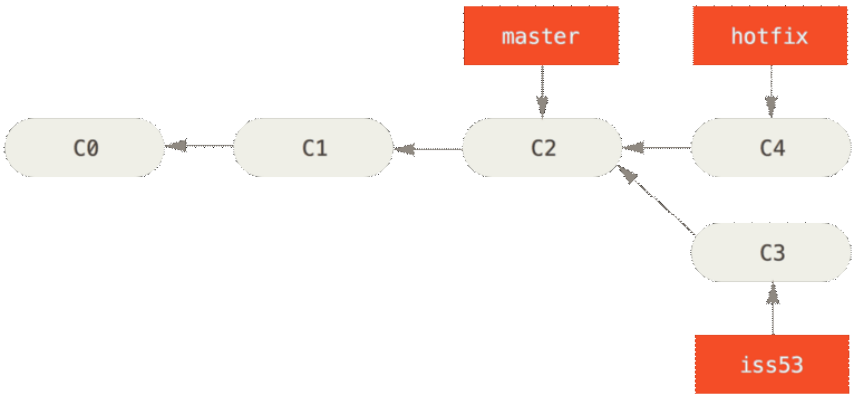
删除的是指针，并不会删除提交对象。

该操作并不能删除没有被合并的分支，如果要放弃分支、强制删除，请将 –d 选项替换为 –D.

⭐ 删除分支时，HEAD 不能指向该分支。

e. 合并分支

两个被合并的分支如果是上下游关系，合并操作会使得上游分支指向下游，这种合并不会有任何冲突。

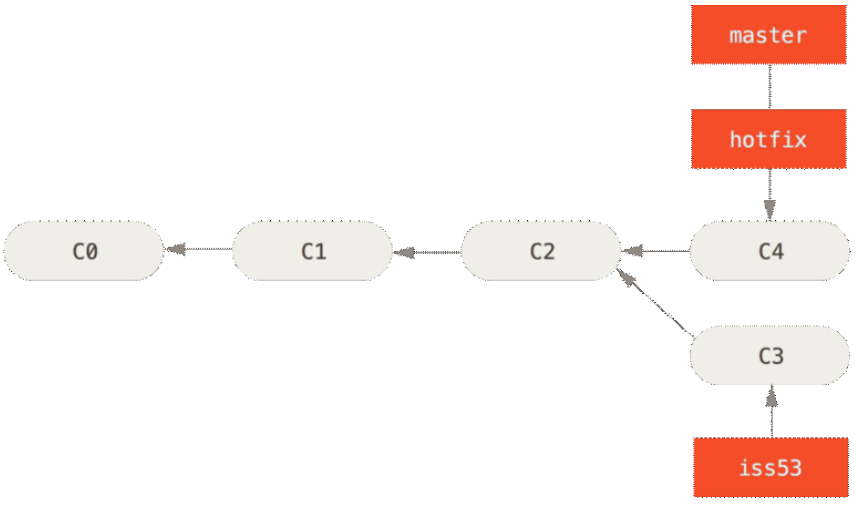


如图，将 hotfix 合并到 master 只需执行：

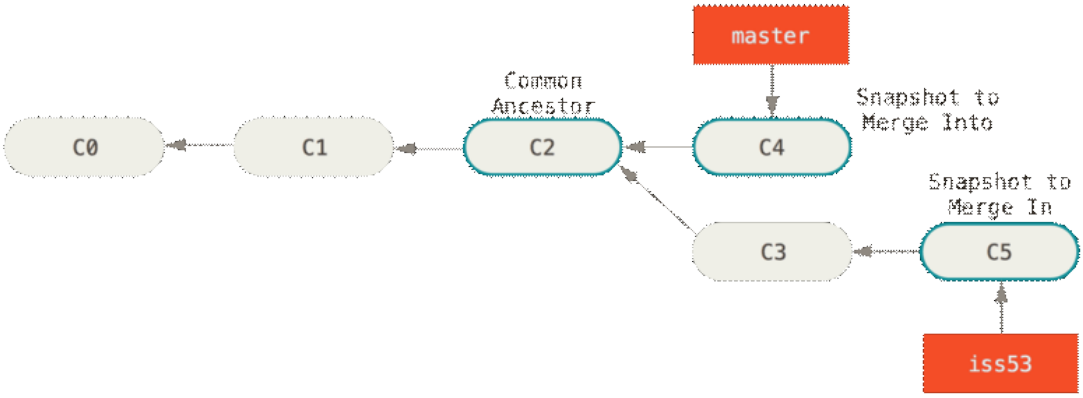
```
git checkout master
git merge hotfix
```

注意，合并的二者有主次关系。

合并后的效果如图：

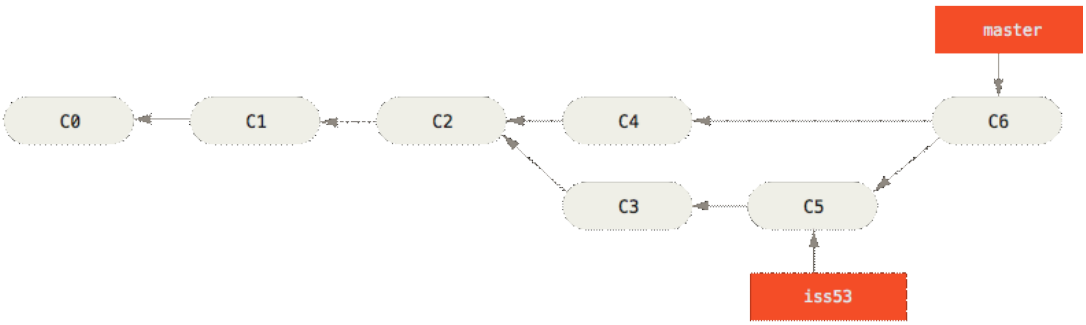


如果两个被合并的分支不是上下游关系，则会根据这两个分支以及它们的共同祖先进行三方合并。




```
git checkout master
git merge iss53
```

合并后的效果如图：



合并完成后，无用的分支 iss53 可被删除。

```
git branch -d iss53
```

但很显然，iss53 和 master 相对于它们的共同祖先可能在同一文件上有改动，Git 这时不会创建新的合并提交，会暂停下来等待用户**解决冲突**。

在合并冲突后的任意时刻可使用 `git status` 查看因包含冲突而处于**未被合并状态**(unmerged)的文件。

Git 还会在有冲突的文件中加入标准的冲突解决标记，你可打开这些文件手动解决冲突。

- 在解决了所有文件里的冲突后，对每个文件使用 `git add` 命令来将其标记为**冲突已解决**。

Git 默认使用 opendiff 作为默认合并工具，还可使用命令 `git mergetool` 可打开图形化合并工具。

在退出合并工具后，Git 会询问合并是否成功；如果完成冲突的解决，便可执行 `git commit` 提交合并.

你可能对合并过程中使用的 add、commit 命令的意义感到疑惑，此处可将 git merge 理解为对当前分支的一种修改，只是这种修改必须要解决冲突。

f. 分支管理

不加参数地运行

`git branch`

可查看包含所有分支的列表。列表中带 * 的是 HEAD 指针所指向的分支。

`git branch -v` 查看每个分支的最后一次提交

`git branch --merged` 查看已经合并到当前分支的分支

`git branch --no-merged` 查看尚未合并到当前分支的分支

g. 远程分支

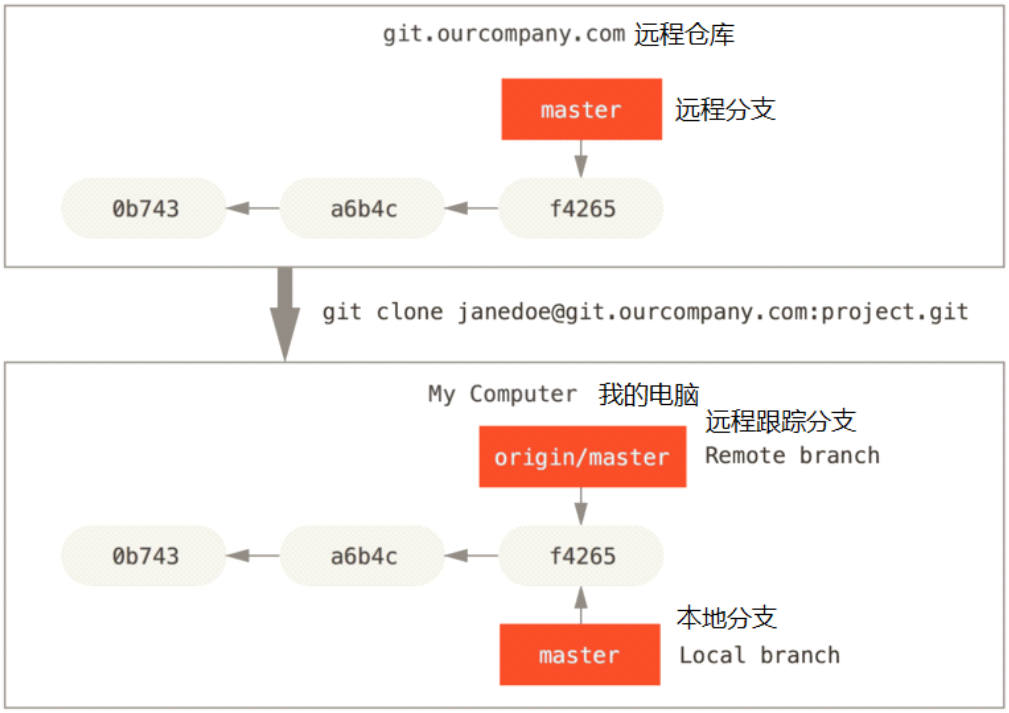
远程引用是对远程仓库的引用，包括分支、标签等。获得远程引用的完整列表的命令：

`git ls-remote <remote>`

远程跟踪分支是远程分支状态的引用，它是你不能移动的本地引用，在与远程通信时会自动移动。远程跟踪分支像是你上次连接到远程仓库时，分支所处状态的书签。

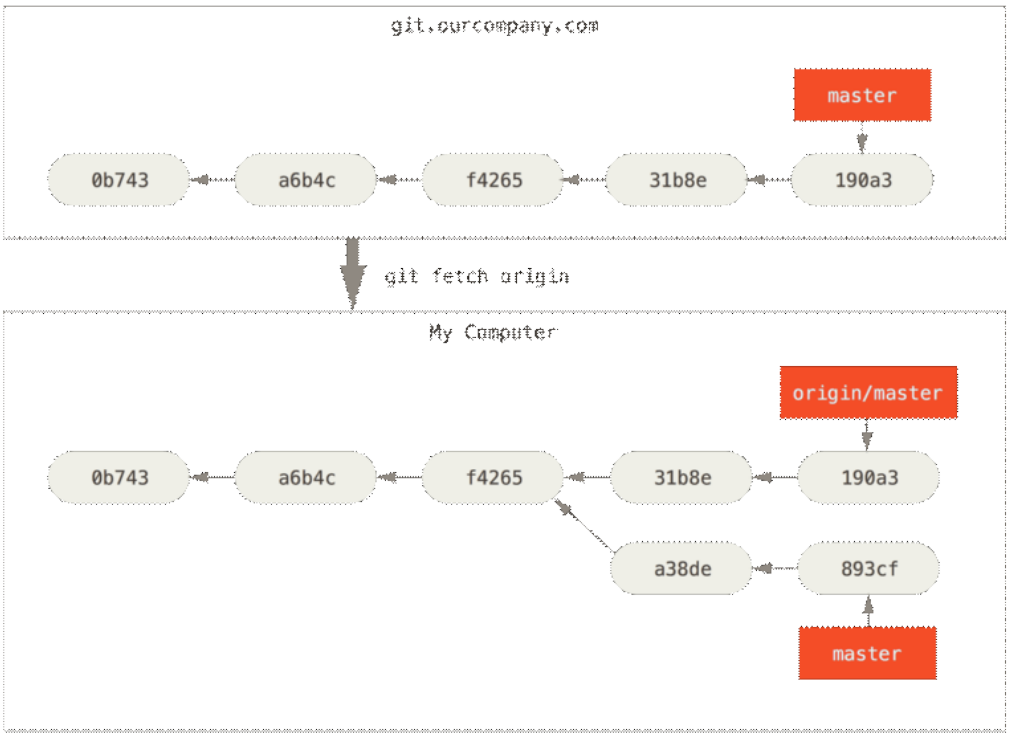
远程跟踪分支以 <remote>/<branch> 的形式命名。

克隆之后的服务器与本地仓库：



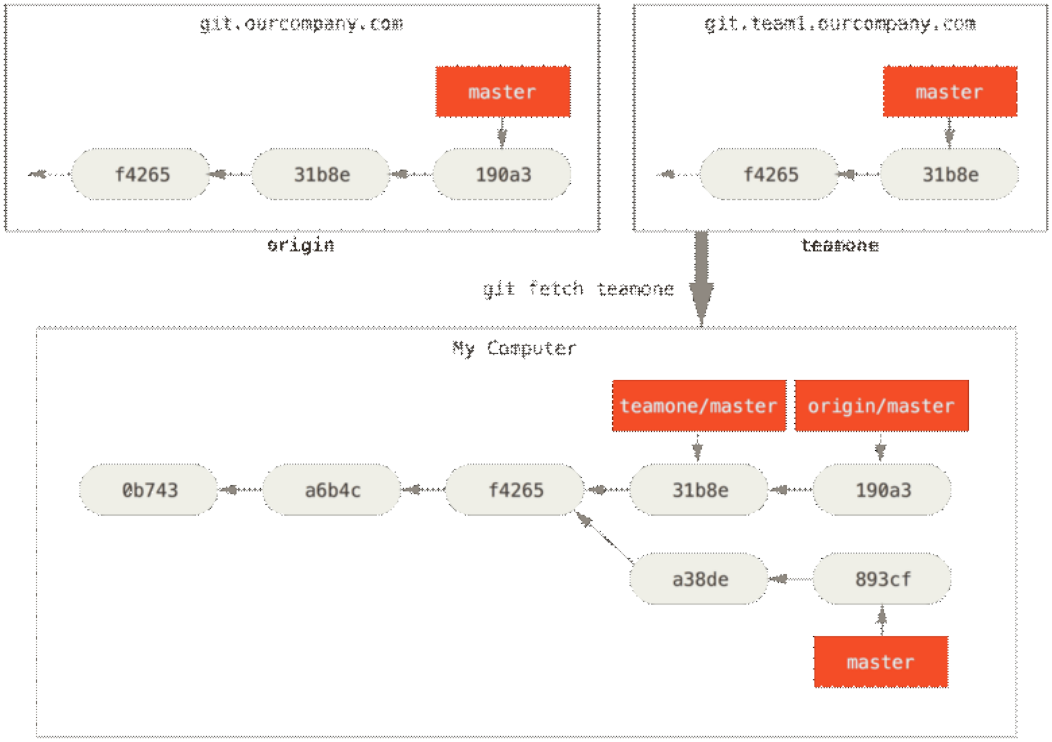
本地与远程的工作可以分叉：

本地与远程分叉后使用 git fetch 更新远程仓库引用：



可见 fetch 的特点是，只要不同就下载分支，不合并。

类似地，fetch 多个远程仓库的情况：



i. 推送

推送一个本地分支的命令：

`git push <remote> <local branch>:<remote branch>`

其中，local branch 是本地分支的名称，remote branch 是推送后的在服务器上的该分支的名称。

推送后，该仓库的其他协作者抓取到该远程跟踪分支时，只是获得了指针，不会获得该分支的内容。若想获得内容，需要在本地的某个分支上执行 `git merge <remote>/<branch>`.

ii. 跟踪分支

查看或修改一个远程跟踪分支的本地分支被称作**跟踪分支**(tracking branch)/**上游分支**(upstream)。 跟踪分支是与远程分支有直接关系的本地分支。

Q：设立“跟踪分支”概念的意义?

A：在跟踪分支里执行 git push 时，Git 会自行推断应该向哪个服务器的哪个分支推送数据，而不需要用户指定参数。同样，在这些分支里运行 git pull 会获取所有远程索引，并把它们的数据都合并到本地分支中来。

创建并切换到跟踪分支：

`git checkout -b <local_branch> <remote>/<branch>`

转到某跟踪分支：

`git checkout --track <remote>/<branch>`

设置已有的本地分支跟踪一个刚拉取的远程分支，或修改正在跟踪的上游分支，可在任意时间使用 -u 或 --set-upstream-to 选项运行 git branch.

```
git branch -u <remote>/<branch>
git branch --set-upstream-to <remote>/<branch>
```

查看设置的所有跟踪分支：

```
git fetch --all; git branch -vv
```

第一条命令从服务器获取信息，第二条命令会列举出每一个分支正在跟踪哪个远程分支，以及本地分支领先或落后的状况。

iii. 拉取

`git pull` 在大多数情况下的含义是 `git fetch; git merge`. 它会查找当前分支所跟踪的服务器与分支，从服务器上抓取数据然后尝试合并入那个远程分支。

iv. 删除远程分支

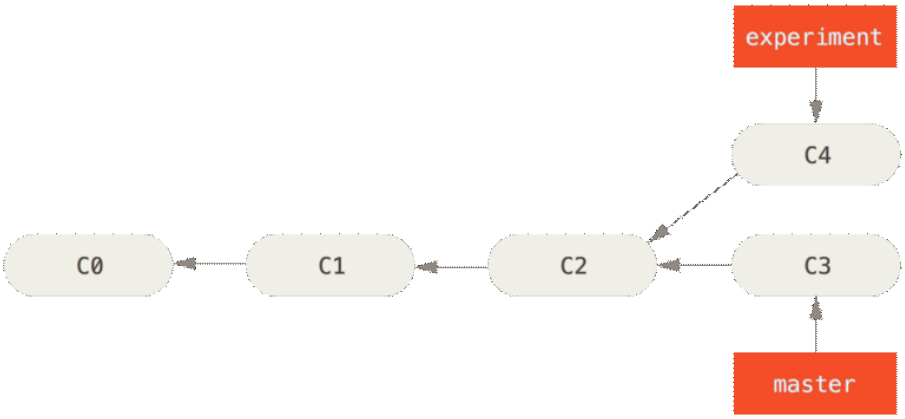
```
git push <remote> --delete <branch>
```

命令将从服务器上删除这个指针，Git 会保留数据一段时间，直至垃圾回收运行。

h. 变基(rebase)

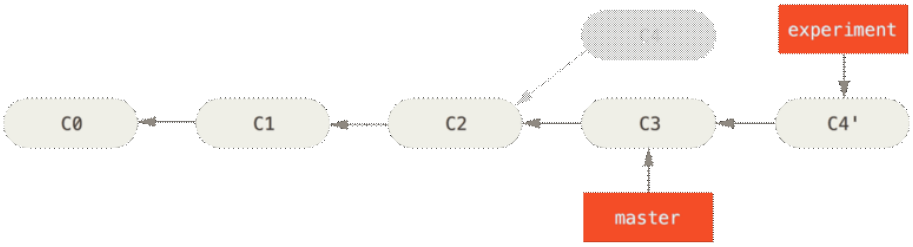
要将 experiment 合并到 master 中，除了前述的 merge，还可使用变基的方法。

```
git checkout experiment
git rebase master
```



在此例中，rebase 的具体原理是，找到二者的共同祖先 C2，将 experiment 相对于 C2 的每一步更改提取出来（这里只有一步），再对 master 按步逐次地进行同样的修改。

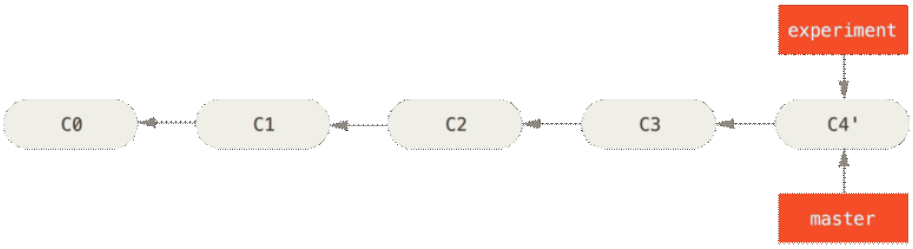
C4‘ 的 experiment 就是变基的结果，它的父对象变为 master，而不是之前的 C2.



此时再执行

```
git checkout master
git merge experiment
```

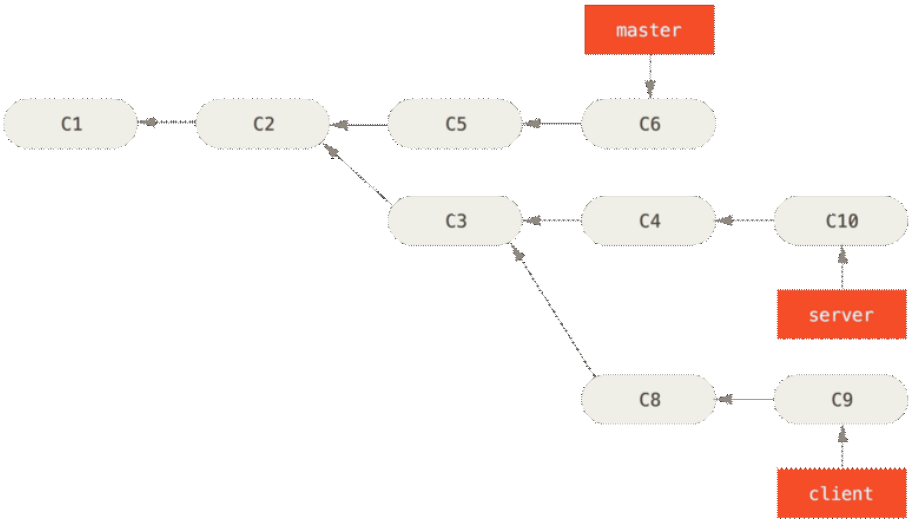
上下游合并后，master 也指向 C4’.



变基的效果与合并没有差别，但前者的提交历史更加整洁。尽管实际的开发工作是并行的，但看上去如同串行。你可能认为提交历史应该真实地记录文件发生的变动，而变基是从项目开发的角度整理了提交历史。

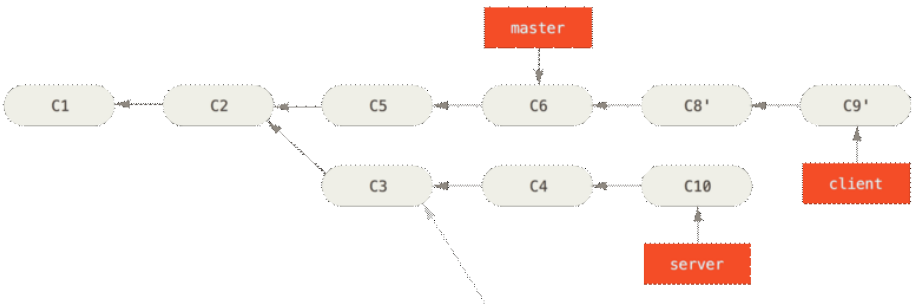
⭐ 与合并类似，变基过程中也可能遇到冲突，冲突解决的方法是相同的。请记得在修改冲突后使用 `git add`.

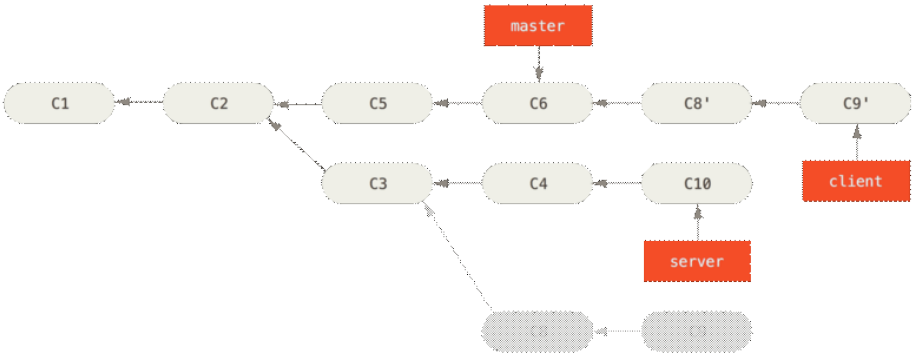
在特性分支中分出特性分支的变基的例子：



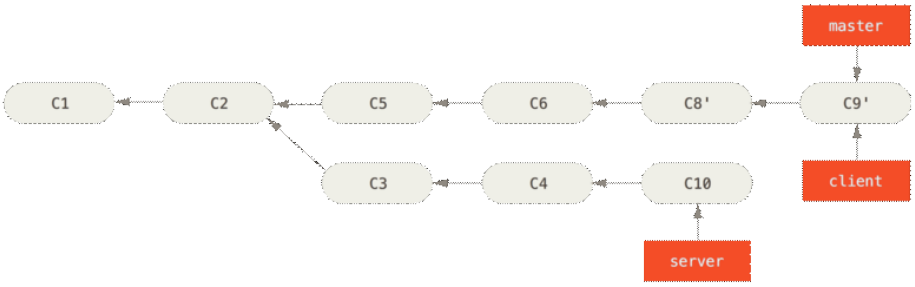
server 是一个特性分支，client 是 server 这个特性分支上的特性分支。

执行命令 `git rebase --onto master server client` 后效果如图：

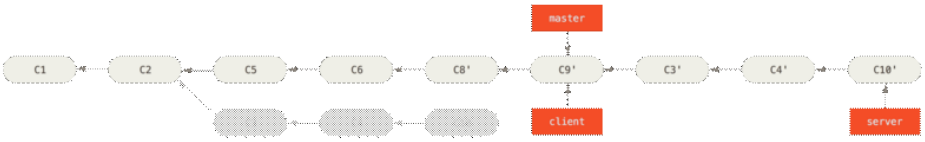




快进合并 master，`git checkout master; git merge client`



再将 server 中的修改变基到 master 上，`git rebase master server`



⭐ 变基的风险：

变基可作为推送前清理提交使之整洁的工具，只在从未推送至共用仓库的提交上执行变基命令，并不会有风险。否则，可能会使远程仓库的代码变得混乱。

4. GitHub

Git 是版本控制软件，而 GitHub 是提供代码托管服务的社区。

a. Student Developer Pack

https://education.github.com/

其中最大的福利是在学生期间拥有无限个私有代码库的使用权。

b. Fork

在操作系统中，fork是创建自身进程副本的操作。非自己所有的代码是无法直接修改的，想要修改他人的代码首先需要通过 fork 创建副本。

c. Pull Request

请求源仓库的所有者拉取自己的 fork.

d. Issue

可使用 GitHub 风格的 Markdown.

主要功能：

i. 报告项目的bug

ii. 向作者询问或探讨问题

iii. 列出之后的开发计划

e. Explore

在 GitHub 上有难以计数的优秀开源项目等待开发者的探索。

https://github.com/explore

5. Git 工具

a. 储藏与清理

当你在某个分支进行一些工作时，需要切换到另一个分支，而不想将当前的工作提交，就可以使用**储藏**(stash)功能。回想前述的暂存(stage)，它们是完全不同的概念。

储藏的命令是 `git stash` 或 `git stash save "description"`.

储藏可以多次进行，使用 `git stash list` 可查看历次的储藏。这是一个堆栈。

`git stash apply [--index]` 默认应用最新的储藏，其中加上 --index 选项可恢复工作区与暂存区，不加 --index 选项只恢复工作区。

`git stash apply [--index] stash@{0}` 可应用指定的储藏，其中0是储藏列表中的序号。

使用 apply 使用储藏后， 堆栈中的储藏并不会消失，需要使用 `git stash drop stash@{0}` 删除指定的储藏。 `git stash pop [--index]`可应用最近的储藏并删除它。

`git stash clear` 清除所有储藏。

b. 搜索

`git grep <keyword>` 查找仓库中所有含关键字的代码段

`git grep -n <keyword>` 查找仓库中关键字的行号

`git grep --count <keyword>` 统计关键字在各文件中的出现次数

`git grep -p <keyword> <file>` 查看关键字在某文件中的所在行属于哪个方法或函数

`git log -S <keyword>` 显示新增或删除关键字的提交

`git log -L:<keyword>:<file>` 查看指定文件中某行或某个函数的变更

6. 相关软件

a. GitHub Desktop

在 macOS 与 Windows 下必备的 GitHub Desktop. 该软件包含 Git Shell 命令行与图形界面客户端。

https://desktop.github.com/

b. IDE 中的 Git

Visual Studio、Eclipse、Jetbrains 等 IDE 中均有内置的 Git.

参考资料：

https://git-scm.com/book/zh/v2

https://learngitbranching.js.org/