

Chapter 0. The history of version control systems

Chapter 1. Basic concepts

1.1 Some concepts of the Git

1.2 **Working states [IMPORTANT]**

1.3 Git command line settings (Linux)

Chapter 2. Git basic commands

2.1 Manipulate a git repository

2.2 Viewing changing history

2.3 Undoing things

2.4 Working with Remotes

2.5 Tagging (版本标签)

2.6 Git Aliases (别名设置)

Chapter 3. Git Branching: "killer feature"

3.1 Branches in a nutshell

3.2 Common Branching Workflows

3.3 Remote Branches

3.4 Rebasing

Chapter 0. The history of version control systems

- Local Version Control Systems

downsides: It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

- Centralized Version Control Systems

downsides:

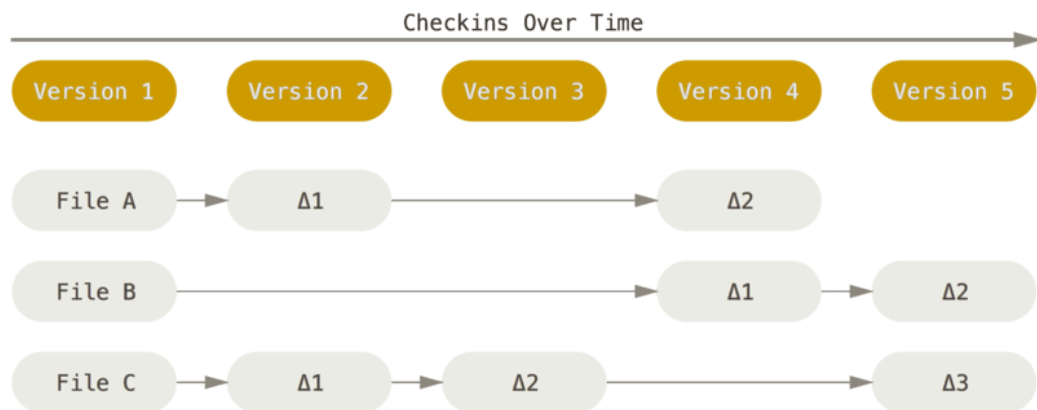
1. the single point of failure that the centralized server represents
2. if the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything (Local VCSs suffer from this same problem)

- Distributed Version Control Systems (such as Git, Mercurial, Bazaar or Darcs)

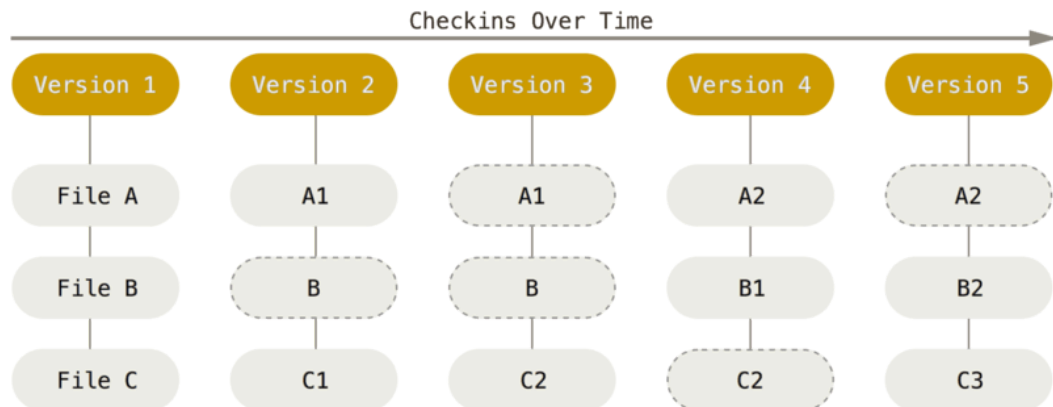
Chapter 1. Basic concepts

1.1 Some concepts of the Git

- How do snapshots record differences?
 - Other VCS: **delta-based version control** (store information as a list of file-based changes);



- Git: takes a picture of what all your files look like at that moment and stores a reference to that snapshot. (if files have not changed, Git doesn't store the file again, just a **link** to the previous identical file it has already stored)



- **Local operation**

Nearly Every Operation Is Local !

1. to browse the history of the project (in local database);
2. see the changes introduced between the current version of a file and the file a month ago (do a local difference calculation);
3. do a little work and **commit offline or off VPN** (upload when online);

- **Integrity (For using SHA-1 hash)**

Everything in Git is checksummed before it is stored.

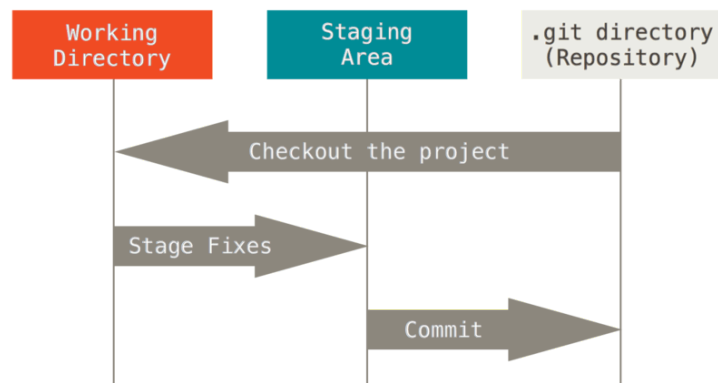
– it's impossible to change the contents of any file or directory without Git knowing about it.

- *Add* but never *erase/modify* in repository

1.2 Working states [IMPORTANT]

Git has three main states that your files can reside in: **modified**, **staged**, and **committed**

- **Modified** means that you have changed the file but have not committed it to your database yet.
- **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.
- **Committed** means that the data is safely stored in your local database.



总结

工作区：放置从数据库中被取出的、在磁盘上供用户修改和使用的文件；

中转地（staging area）：储存在Git目录中，即将进入下一个commit的选中的部分文件；

Git储存库（repository）：储存元数据、objects数据库的区域，通常 `clone` 操作也放置在这里；

正确使用Git的流程

1. 在工作区编辑项目文件；（modified）
2. 选出某些需要的features/需要进入next commit中的文件，加入中转地；（staged）
3. commit（执行此操作时提交中转地中的文件作为一个commit（或者说一个snapshot），永久存于Git储存库中）；（committed）

1.3 Git command line settings (Linux)

1. Git 程序配置文件位置：`/etc/gitconfig`（系统全局设置）、`~/.gitconfig` 或 `~/.config/git/config`（用户个性化设置，针对某个用户的全局设置，使用`--global`参数指定）、`[current repository PATH]/.git/config`（项目个性化设置，针对某个用户项目的局部设置，使用`--local`指定）
2. 级别最接近用户级的，最有效（局部设置与全局设置冲突时，服从局部设置）
3. 检查Git当前所有设置及其定义的位置：`git config --list --show-origin`
4. 检查Git当前所有设置：`git config --list`
5. 一些建议
 - 安装git后，立即设置“用户全局设定”的用户名、邮箱，因为这是git commit 所有项目的目标地址（关联github）`git config --global user.name "xxx" && git config --global user.email xxx@xxx.com`
 - 设置Git默认打开的系统编辑器 `git config --global core.editor [PATH/NAME, e.g., vim]`
 - 设置默认的新的储存库（主分支）的名字 `git config --global init.defaultBranch main`

同样的，如果某个项目需要特殊指定，那么就在项目局部设置中执行设置操作

Chapter 2. Git basic commands

2.1 Manipulate a git repository

1. 创建Git存储库

1. You can take a local directory that is currently **not under version control**, and turn it into a Git repository, or
2. You can *clone* an existing Git repository from elsewhere.

- 从存在的非版本控制目录创建：命令行切换到该目录下，输入 `git init`

当前目录下会产生一个子目录 `.git/`，其中记录这个存储库的框架，不包含任何被追踪的项目；

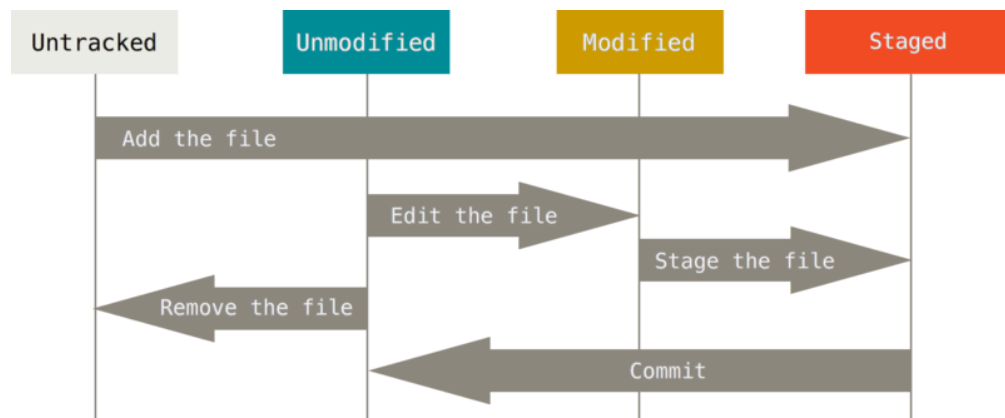
可以使用 `git add [FILENAME]` 将文件（建议放在这个目录下再进行）放入staging area（跳过追踪，下面详述），并使用 `git commit -m ["DESCRIPTION"]` 提交生成一个主分支下的commit节点

- 从存在的存储库中创建（一般是别人的，位于服务器上的，也可以是自己的），输入 `git clone <url> [new dirName]`

以URL=<https://github.com/libgit2/libgit2> 为例，在当前目录下会生成一个子目录 `.git/` 和子目录 `libgit2/`，后者包含原来存储库中所有内容，并且内容处于工作区，并且属于你的master分支。

2. 更改Git存储库、更改项目状态

Each file in your working directory can be in one of two states: **tracked** or **untracked**.



- 被追踪的文件一定是：上一个提交的snapshot中的文件，或者是在中转地的文件；
- 未被追踪的文件是：上次snapshot中不含此文件，或者也没有staged过的文件，又或者被手动移除追踪属性（在工作区下自行新建一个文件，但没有任何操作时，也是未被追踪的文件）；

这样做的好处是，不会误添加不相关的文件进入项目

- 当刚clone完成一个存储库，其中所有文件都处于Unmodified阶段、工作区（working directory）；
- 一旦更改了Unmodified状态的文件，该文件立即变为Modified状态（for SHA-1）；

- 检查文件所属状态、所属分支（在存储库目录下）：`git status`

简短表述: `git status -s/--short`

Untracked: ??

Modified: M

Staged: A

Modified & stage & modified later (未将更改再次写入staging area): MM

- 追踪Untracked状态文件，并添加进staging area（别问为什么不能加到其他状态中去，因为如果不commit的话，就没有必要追踪了啊）：`git add [PATH / FILENAME]`

若参数是 PATH，那么会递归地将该路径下所有文件加入staging area

- 将Modified状态文件加入staging area（称为staging Modified files）：`git add [PATH / FILENAME]`

注意：如果修改了staged文件，那么git仍会保留之前的staged的版本（提交时也会默认这个版本），除非将修改后的文件再次加入staging area

综合上面的和这个例子可知：

1. git永远不会将用户不指定的文件加入到staging area中
2. git不会将unmodified文件加入staging area，但在储存库的下一个commit中unmodified的文件仍然存在，只不过是上个commit的内容的引用罢了。

- 添加需要git忽略的文件（比如程序日志、编译的中间文件，没必要上传到存储库中）：
写 `.gitignore` 文件

使用场景列举：比如使用 `git add .` 来添加当前目录下所有文件（包括untracked、unmodified、modified的文件）到staging area中，但又不希望某些文件/目录被添加

建议：在创建存储库后就写这个文件，以防提交不应该被提交的文件

针对范围：此目录及其子目录（递归生效）

`.gitignore` 文件的语法规则

- 注释 `#`
- glob patterns: 一种类似正则表达式的语句
 - `*`: 0~N个字符, `?`: 1个字符, `/**/` 用于匹配任意中间目录的结构
- 在每行的glob patterns前加 `/` 取消递归和递归查找生效
- 在每行的glob patterns后加 `/` 说明前面的是目录
- 否定忽略策略（和UNIX系统设置一样，优先级高于肯定）：在约束名前加上 `!`

示例：

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a
```

```
# only ignore the TODO file in the current directory, not
  subdir/TODO
  /TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its
  subdirectories
doc/**/*.pdf
```

- 查看文件（无论是staged还是unstaged，但未commit）相较于上次提交的更改情况

比较modified文件的不同: `git diff`

额外参数: `--staged`, 指定比较staged文件

图形化: `git difftool`

- 提交staging area中的文件: `git commit`

此举将打开默认编辑器来显示消息，并提示用户输入一些**description**，描述一下提交的是什么，这在团队开发中很必要

强烈建议: 加 `-m "[DESCRIPTION]"` 参数，提供有效的提交信息

最终输出信息中会出现本次提交的分支、**SHA-1 hash code**等信息

- 跳过Staging area并提交: `git commit -a -m "..."`

此举git将自动提交所有被追踪的文件（包括**unmodified**、**modified**），注意这样做前考虑一下，别提交了不想要的文件

- 将文件从追踪属性上移除（主要用于将文件从staging area中移出）

- 手动将文件移出工作区(变为deleted状态) + `git rm`: 此举只会使下下次的提交没有这个文件（不再追踪），但是下次提交仍会有。如果想这次就彻底删除，使用强制参数:

```
git rm -f
```

这样做是为了保证数据安全，防止误删将要提交的文件，并且不能恢复

这个`git rm`移除的是处于deleted状态的文件

- 如果只是将文件移出(某个状态)，但保留在硬盘上（比如不小心忘写`.gitignore`文件，结果一大串日志/编译中间文件都放到了staging area）

```
git rm --cached [FILENAME / PATH / glob-pattern(*需要使用转义字符)]
```

不在staging area的: `git rm [...]`

如果只想移出 **staging area**，还想 **track** 这个文件，那么就撤销加入 **staging area** 的命令，详见 “Undoing things”

- 移动文件: `git mv [PATH / FILENAME] [NEW_PATH / NEW_FILENAME]`

Git 不会知道你移动了/重命名了某个追踪的文件，会导致追踪丢失（deleted状态），如何找寻丢失的文件以后说，这次讨论如何告诉Git移动/重命名文件，防止发生追踪丢失

等价于：手动移除追踪属性，改完，再追踪回去

2.2 Viewing changing history

- 查看提交日志（提交时间从前向后的顺序）: `git log`
 - 额外参数: `-p/--patch [-{number}]`，表示展示每次提交的时候同时展示相对于上一次的修改的详细内容（指定展示commit的次数）
 - `--stat`，表示展示每次提交相对于上一次修改的简略内容（只有修改的行号等，但也有基本信息，所以在很多commit的总览时非常有用）
 - `--pretty=[oneline / short / full / fuller / format:"formatStr"]`，自定义提交日志展示的美化。
 - `oneline`: 在看很多commit时候有用，每个commit信息浓缩在一行
 - `short`: 比oneline长一点的简短信息
 - `full`、`fuller`: 更全的信息

谁是author、committer? ---> 最初自行创建这个存储库的人是author，后来加入这个项目开发的人是committer。

推荐使用的formatStr: `"%h - %an, %ar : %s"`

 - `--graph`: 显示commit的可视化DAG，可以结合浓缩在一行中的指令（`pretty=oneline/format`）
 - `--name-only`/`--name-status`: 仅显示提交的文件名/状态+文件名
 - 查找限制参数
 - `--author/committer='...'`: 指定作者/贡献者
 - `--grep='...'`: 同UNIX grep
 - `--since/after/until/before="yy-mm-dd"`: 按时间查找
 - `-- <PATH>`: 按文件/目录查找（中间有空格）

2.3 Undoing things

警告：这里的命令是git中可能丢失数据的命令。因为用户无法撤销某些“被撤销”的操作

- 补交commit: `git commit --amend`

此举将当前staging area中的所有文件提交到上一次的commit中（不创建新的commit）；如果当前staging area中没有文件，那么什么都不变，除非在此次提交中更新传入commit message，则会覆盖之前的commit message.

建议：只有当你修改幅度非常小的时候才建议使用这个命令，否则为何不直接提交一个新的commit呢？

- Unstaging a Staged File: `git reset HEAD FILENAME`

之后会进一步说明 reset 命令和 HEAD 的详细含义

- Unmodifying a Modified File (discard the changes you've made) : `git checkout -- FILENAME`

[Data loss Warning]: Git just replaced that file with the last staged or committed version. Don't ever use this command unless you absolutely know that you don't want those unsaved local changes.

Where: modified files (working area) (由于保护机制，对staging area内的文件无效)

tips. 1. 这个命令很危险，是因为：即便是`--amend`覆盖，也能恢复数据（因为已提交），但这个命令未提交，大概率无法找回；2. 可以用其他命令来避免这个命令的使用，例如分支来保留特性，见后面描述。

- Git 2.23.0更新: `git restore` (此命令比上面两个更好记)
 - Unstaging a Staged File: `git restore --staged FILENAME`
 - Unmodifying a Modified File: `git restore FILENAME`

这两条命令与上面的效果相同，**[Data loss Warning]**

2.4 Working with Remotes

这里的“远程”也可以是自己的主机！只不过这个目录不是被追踪的git目录，所以和远程的存储库的操作类似。

网络上的存放存储库的主机一般是 GitHub

1. 配置远程存储库

- 显示当前配置的远程主机: `git remote [-v]` (`--verbose` 更详细的信息，含缩略名对应的地址)

origin 缩略名表示上一次从远程主机clone的地址

- 添加存放存储库的远程主机: `git remote add <shortname> <url>`

2. 从远程存储库获取项目

- 从已记录的远程主机上获取项目: `git fetch <shortname>`

不会主动合并分支，具体分支的信息详见branching。

- 从已记录的远程主机上获取项目并将其合并到当前分支: `git pull <shortname>`
- 从远程主机上获取项目，并将自己的master引用置于获取到的项目的默认主分支上: `git clone <sn/url>`

以上命令涉及变基、分支和引用的更改，详见branching。

3. 向远程存储库提交、查看

- 向remote提交 (push, 只有有权限才可以这么做, 一般只有自己的GitHub才行, 因为别人一般不会允许其他人更改的): `git push <remote> <branch>`
 - 查看远程存储库的情况: `git remote show <remote>`
4. 重命名、移动、删除 本地对应远程存储库的缩略名 (记住: 缩略名只是联系远程存储库和本地的某些存储库的纽带)
- `git remote rename <old> <new>`
 - `git remote rm <shortname>`

2.5 Tagging (版本标签)

- 查看当前设置标签: `git tag [-l/--list "{wildcards}"]`
- 创建标签
 - Lightweight型标签 (固定不变, 指向特定commit的标签): `git tag tagName`
 - Annotated型标签 (相当于是一次完整的commit, 有tagging message等和commit一样多的信息): `git tag -a tagName -m "..."`
- 将tag向远程存储库提交: 由于git不主动提交标签, 需要显式指定 `git push <remote> <tagName>`, 或在普通push时, 加上参数 `--tags` (全部标签) / `--follow-tags` (annotated型标签)
- 删除标签: `git tag -d tagName`
- 删除远程存储库的标签: `git push <remote> :/refs/tags/tagName`
- 目前请勿轻易使用 `git checkout tagName` 来查询标签, 因为会导致HEAD引用分离, 进而在下一次提交时丢失commit! 详细请参考branching章节。

2.6 Git Aliases (别名设置)

类似于宏替换: `git config --global alias.ALIASNAME sourceName`

例如使用:

```
git config --global alias.unstage 'restore --staged',
```

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

这样就能用 `git unstage FILE` 来代替 `git restore --staged FILE`

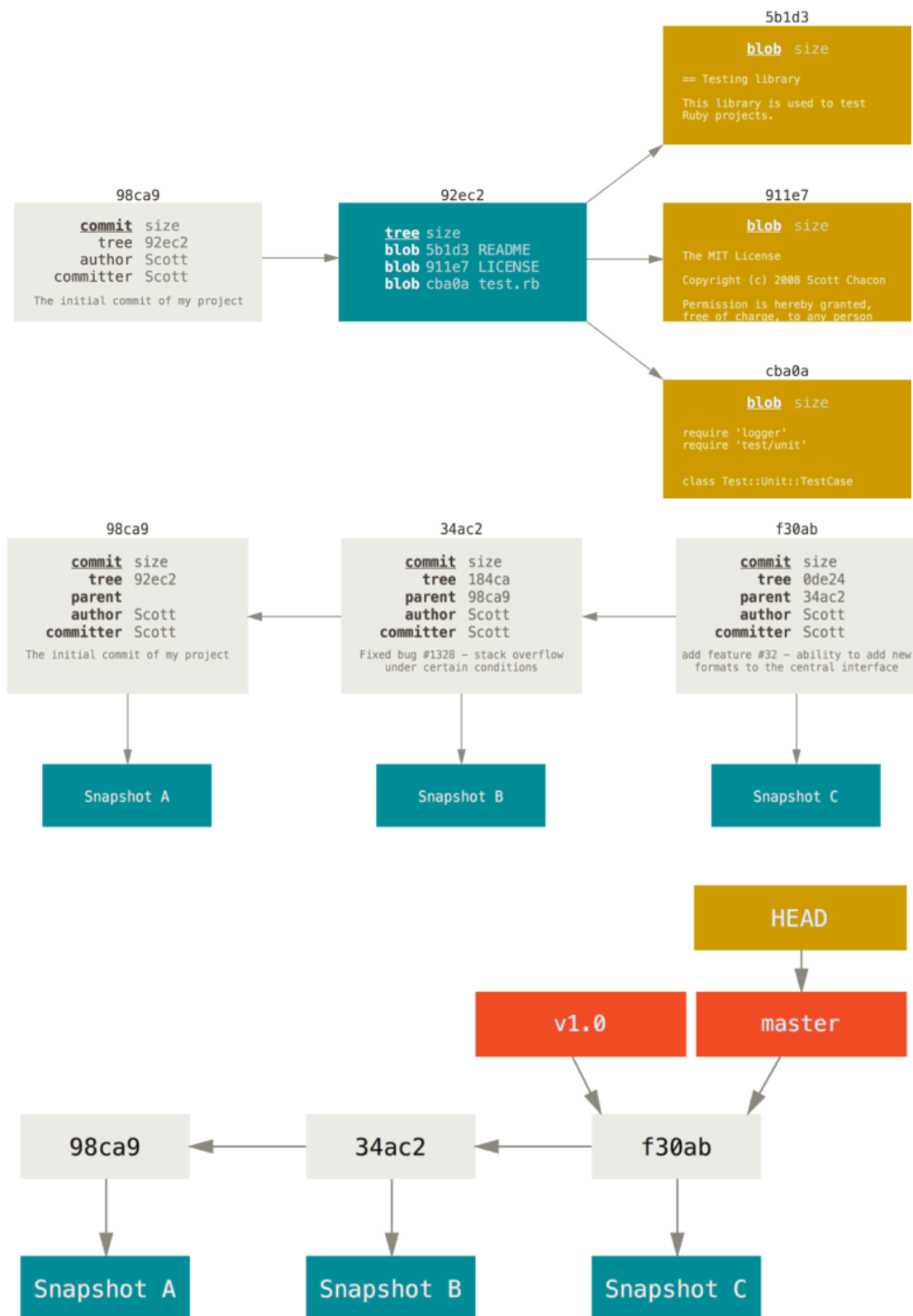
用 `git last` 来代替 `git log -1 HEAD`

注意: 外部命令需要用“!”, 例如: `git config --global alias.visual '!gitk'`

Chapter 3. Git Branching: "killer feature"

3.1 Branches in a nutshell

1. Look back on how Git stores its data (略, 详见[Git-book](#))



- A branch in Git is simply a **lightweight movable pointer** to one of these commits.

branch对象在Git中, 只是一个像C++的指针, 只存放指向对象的hash code (40位SHA-1)

- **Master**: default branch moves forward automatically (every time you commit).

master分支指针跟随当前**commit**移动的指针，没什么特别，只是git init一开始设置的而已。

master之所以可以跟随当前**commit**移动的指针，不是它特殊，是因为**HEAD**特殊！

HEAD一开始指向**master**！每次提交**commit**时默认移动**HEAD**指向的指针！一旦**HEAD**

离开**master**，**master**就是普通分支指针，不会随当前**commit**移动

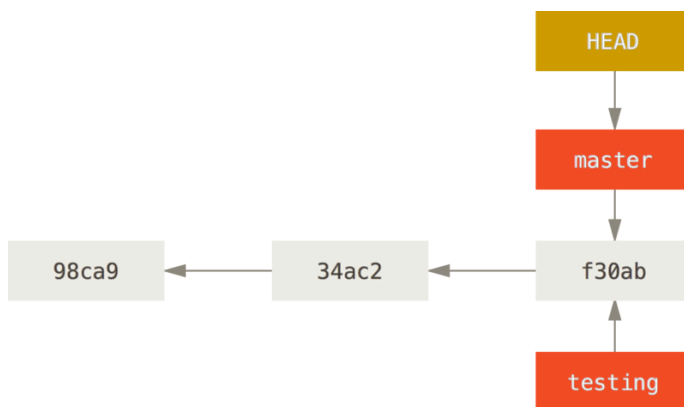
- **HEAD**: this is a pointer to the local branch you're currently on.

HEAD指针严格定义为当前所处的分支位置，可以把**HEAD**理解为指向分支指针的指针，其他分支操作默认依赖此指针，所以最好在改写它之前考虑一下后果！

2. Creating a New Branch: `git branch newBranchName [position=HEAD]`

从数据底层考虑这一步在做什么（以 `git branch testing` 为例）：

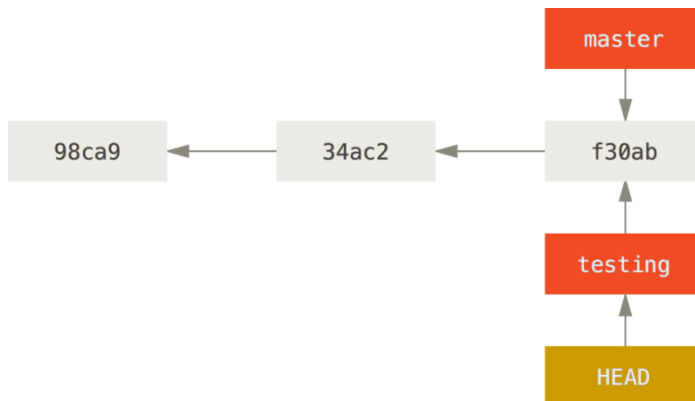
补充：**git log** 查看参数 `--decorate` 可以显示在当时**commit**下的分支指针情况



由图，执行创建分支操作后，只生成一个指向**HEAD**（当前**commit**）的指针**testing**，并不改变其他任何数据！

3. Switching Branches: `git checkout branchName`

从数据底层考虑这一步在做什么（以 `git checkout testing` 为例）：



tips1. 修改分支只是修改**HEAD**指向分支的指向！

一旦此时**commit**，那么**master**将停留在原处，只有**testing**和**HEAD**向前移动。

补充：**git log** 默认不展示所有分支，如果想要展示所有分支，添加 `--all` 参数，所以可以这样设置参数来查看完整的分支信息：`git log --oneline --decorate --graph --all`

tips2. 修改分支可能会改变**working directory**中被追踪的文件，因为**HEAD**可能回到一个老的版本分支上！而**HEAD**严格定义为当前所处的分支位置，意味着**working directory**中所有追踪文件必须与**HEAD**指向的**commit**相同（不然你怎么编辑其他分支的文件？）

tips3. 这样可以完成分支分离(Divergent history)的操作（回到以前的版本，进行全新的提交即可）

tips4. 可以同时创建、切换分支操作，但要记住在干什么：`git checkout -b newBranchName`

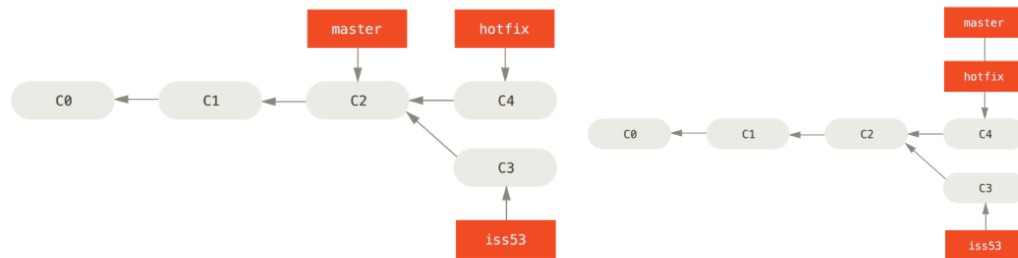
tips5. Git 2.23更新：可以使用`git switch [branchName / -c new / -(切换到上一次的分支指针)]`

tips6. 切换分支前，请确保working area、staging area中来自上一个分支未提交的更改及时清空，否则可能遇到冲突，git不会允许切换分支。（清空的方法可以是：`amend`补交等）

4. Merging Branches: `git merge targetBranch`（一般切换到master上再做操作，因为合并master，让master成为main主分支最新的版本是业界约定俗成的）

合并情况一：“Fast-forward”，即要master合并的分支是master的直达的子分支

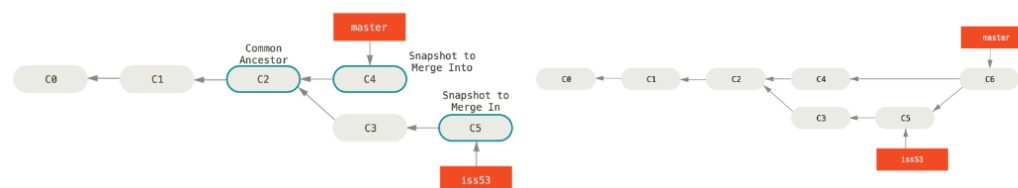
这种情况下一定不会出现合并冲突，只需将master向前移动若干commit即可，例如下面的例子：



上面的过程只需执行`git checkout master; git merge hotfix`即可，现在即可删除多余的hotfix分支了。

合并情况二：“Common Ancestor”，即要合并的两个分支有共同祖先结点

这种情况下，Git需要使用“三项合并”（three-way merge，即共同祖先、要合并的两个结点），生成并得到一个新commit（结点），它有两个父结点，就是刚刚合并的两个（为了防止结点丢失），现在就可以删除多余的分支指针了。



上面的过程只需执行`git checkout master; git merge iss53`即可

警告：情况二可能出现冲突（如果同一段代码不能协调合并），此时需要修改才能合并

那么情况二什么时候不冲突呢？----> 涉及 `three-way merge` 的原理：

通过“原件”（共同祖先结点）推断该采用谁的版本：如果文件的同一行中，三者都相同，则不变；如果文件的同一行中，一个分支与祖先结点相同，另一个更改了，那么合并后会保留更改的内容！如果三者都不相同，则会发生冲突。

5. Deleting Branches: `git branch -d branchName`

6. Query Branches: `git branch [-v/--merged(已合并/存在于当前HEAD所处分支上的)/--no-merged(其他分支上的)/--all]`

“*”表示HEAD指针所指向的分支。

7. Rename Branches: `git branch --move <OLD_name> <NEW_name>`

Remote: `git push --set-upstream <remote> <NEW_name>` (不删除原名, 相当于复制一个不同名的分支, 需要手动删除旧名)

3.2 Common Branching Workflows

- 小项目的长期的分支建议: 维护 `master` (稳定版的代码)、`develop` (测试版的代码, 一旦可以稳定使用, 就将`master`合并到这里)、`topic` (开发中的代码, 确保没有bug、通过测试集, 短期存在)
- 大项目如果具备多种稳定版本, 可以设置多个长期稳定的分支, 根据需求来;
- 多目标项目的分支建议: 可以多设几个分支, 来回切换工作, 找合适的进行合并, 不合适的丢弃;

3.3 Remote Branches

tips1. 参数 `<remote>` 可以是url, 也可以是shortname

tips2. `origin` 作为shortname并不是特殊的, 和`master`一样, 只是创建时默认的名称而已

1. Query Branches: `git remote show <remote>`

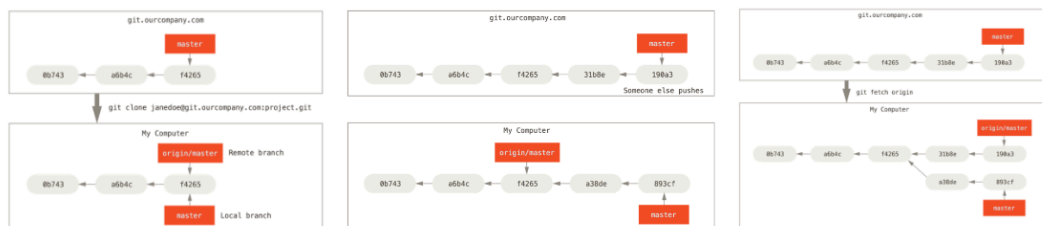
更加好的方法是查询 `Remote-tracking branches`, 终端无法更改这个分支, 由git自动维护, 精确指向上次对远程存储库进行操作的分支位置;

所谓 `Remote-tracking branches` 的形式是: `<remote>/<branchName>`, 表示在远程存储库中的 `<branchName>` 的位置;

注意: 除非手动同步, 否则 `Remote-tracking branches` 不会自动更新

2. Synchronize `Remote-tracking branches`: `git fetch <remote>`

注意: 如果远程存储库已经在原来的基础上提交了一些commit, 则在本地同步后会产生分叉, 例如下图



注意2: 如果是从多个server上fetch, 那么会有多种 `Remote-tracking branches`, 位置视情况而定;

注意3: `Remote-tracking branches` 不能作为本地的HEAD的指向对象! 如果不创建新的分支就将HEAD移动到这里, 会出现“detached HEAD”的情况, 可能造成commit的丢失!

3. Pushing Branches (向远程存储库commit分支): `git push <remote> <branch>`

含义是: 提取本地的branch分支上所有文件, 复制到远程存储库上, 并作为远程存储库的同名分支, 同时, 远程存储库的HEAD指针移动到当前的分支上;

知识补充: `git`的HEAD指针、分支信息存储于 `.git/refs/heads` 中, 所以这条指令可以这么写: `git push <remote> <[refs/heads/]localBranchName:[...]remoteBranchName`

补充2: 如果基于 `Remote-tracking branches` 创建了一个分支 (不是用hash/HEAD创建), 则这个分支会自动跟踪远程的对应分支, 并且这个本地的分支可以修改, 称为 `Tracking branch`。其主要作用是: 告诉 `git pull` 具体的合并位置在哪。

补充3: 不建议使用 `git pull`, 因为这是 `git fetch + git merge`, 后者更易于理解, 不易出错

4. Checking tracking branches: `git branch -vv`

5. Setting a normal branch as a tracking branch: `git branch -u <remote/branch>` (针对当前分支)

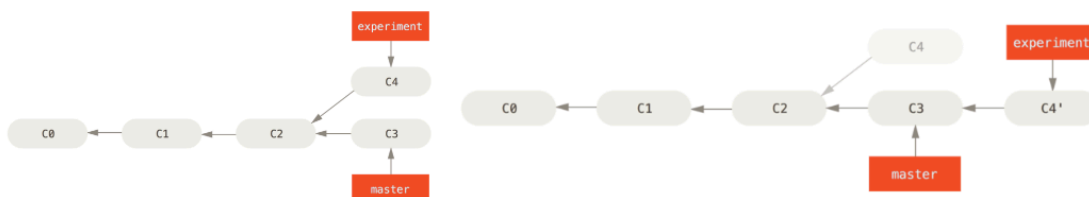
6. Deleting Remote Branches: `git push <remote> --delete <branch>`

3.4 Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`.

Usage: `git rebase targetBranch`

1. In a nutshell: In the picture shown below, we take the patch of the change that was introduced in `C4` and reapply it on top of `C3`. In Git, this is called *rebasing*.



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

算法原理: 找到二者的公共祖先结点、找到祖先结点和这两个commit共三者的不同, 将这些不同保存在临时文件中 (这里的合并方式和merge是一样的), 生成一个新的commit, 作为targetBranch的子结点, 同时将当前分支移动到新结点上, 删除原分支。

2. rebase和merge的区别: 仅仅是rebase的历史记录呈现线性形状, 比merge清晰一点而已;

通常应用**rebase**的场景：可能是正在为一个不是自己维护的项目提交代码，为了防止维护者阅读和合并的麻烦，在提交前使用**rebase**合并。

rebase严禁使用的场景：在远程公共存储库使用此命令，修改他人正在编辑的分支（否则可能造成一些重复的**commit**的问题）；

从可读性、项目思路上说，应该多多使用**rebase**；

从存储库保留历史、无论多乱也不篡改历史的角度说，应该多多使用**merge**。

总结：建议每次提交某个小分支前使用**rebase**提供良好阅读体验；但对已经提交过的部分不应该使用**rebase**.