Git 面试题

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的 Git 面试题的大保健。

而题目的难度，尽量按照从容易到困难的顺序，逐步下去。

基础

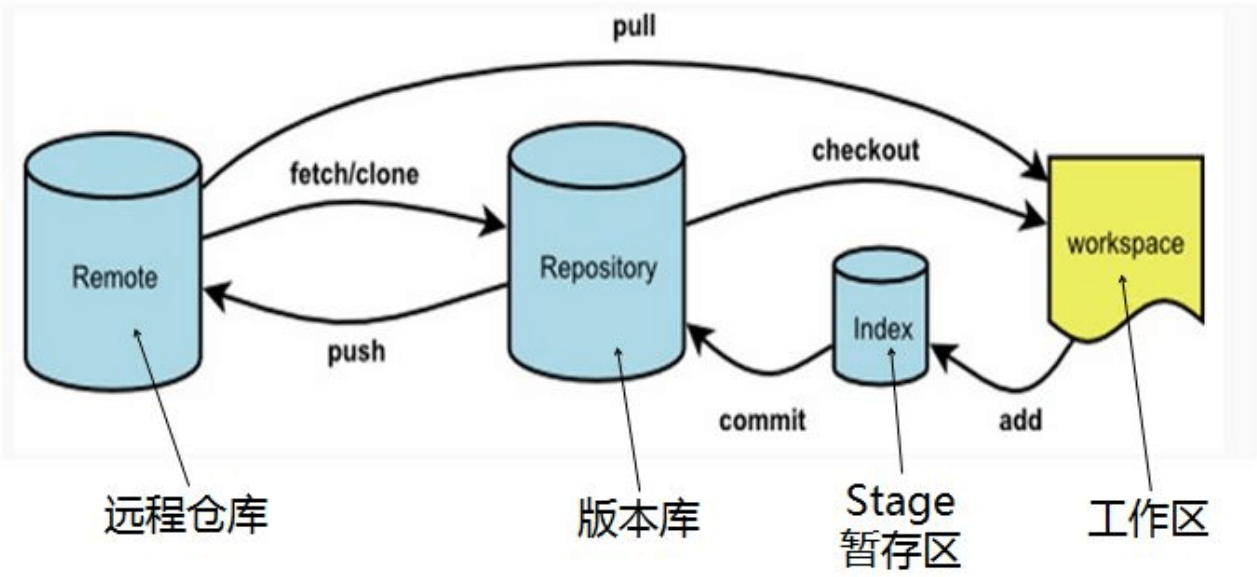
Git 的一些常用命令?

- `git init`：创建 Git 库。
- `git status`：查看当前仓库的状态。
- `git show`：# 显示某次提交的内容 git show $id
- `git diff`：查看本次修改与上次修改的内容的区别。
- `git add <file>`：把现在所要添加的文件放到暂存区中。
    - `git log -p <file>`：查看每次详细修改内容的 diff 。

    - `git rm <file>` ：从版本库中删除文件。
    - `git reset <file>` ：从暂存区恢复到工作文件。
    - `git reset HEAD^` ：恢复最近一次提交过的状态，即放弃上次提交后的所有本次修改`。

      > HEAD 本身是一個游标，它通常會指向某一个本地端分支或是其它 commit，所以你也可以把 HEAD 当做是目前所在的分支（current branch）。 可参见 《Git 中 HEAD 是什么东西》 。

- `git commit` ：把 Git add 到暂存区的内容提交到代码区中。
- `git clone` ：从远程仓库拷贝代码到本地。
- `git branch`：查看当前的分支名称。
    - `git branch -r` ：查看远程分支。
- `git checkout` ：切换分支。
- `git merge <branch>` ：将 branch 分支合并到当前分支。
- `git stash`：暂存。
    - `git stash pop` ：恢复最近一次的暂存。
- `git pull`：抓取远程仓库所有分支更新并合并到本地。
    - `git push origin master` ：将本地主分支推到远程主分支。



平时使用什么 Git 工具?

1）命令行

只能说十个里面九个菜，还有一个是大神，虽然命令行提供了全部的功能，但是很多用 GUI 工具可以很便捷解决的问题，命令行做起来都比较麻烦。

当然并不是让大家不要去命令行，通过命令行可以对 git 的功能和原理有一个更深入的了解。

2）IDEA Git 插件

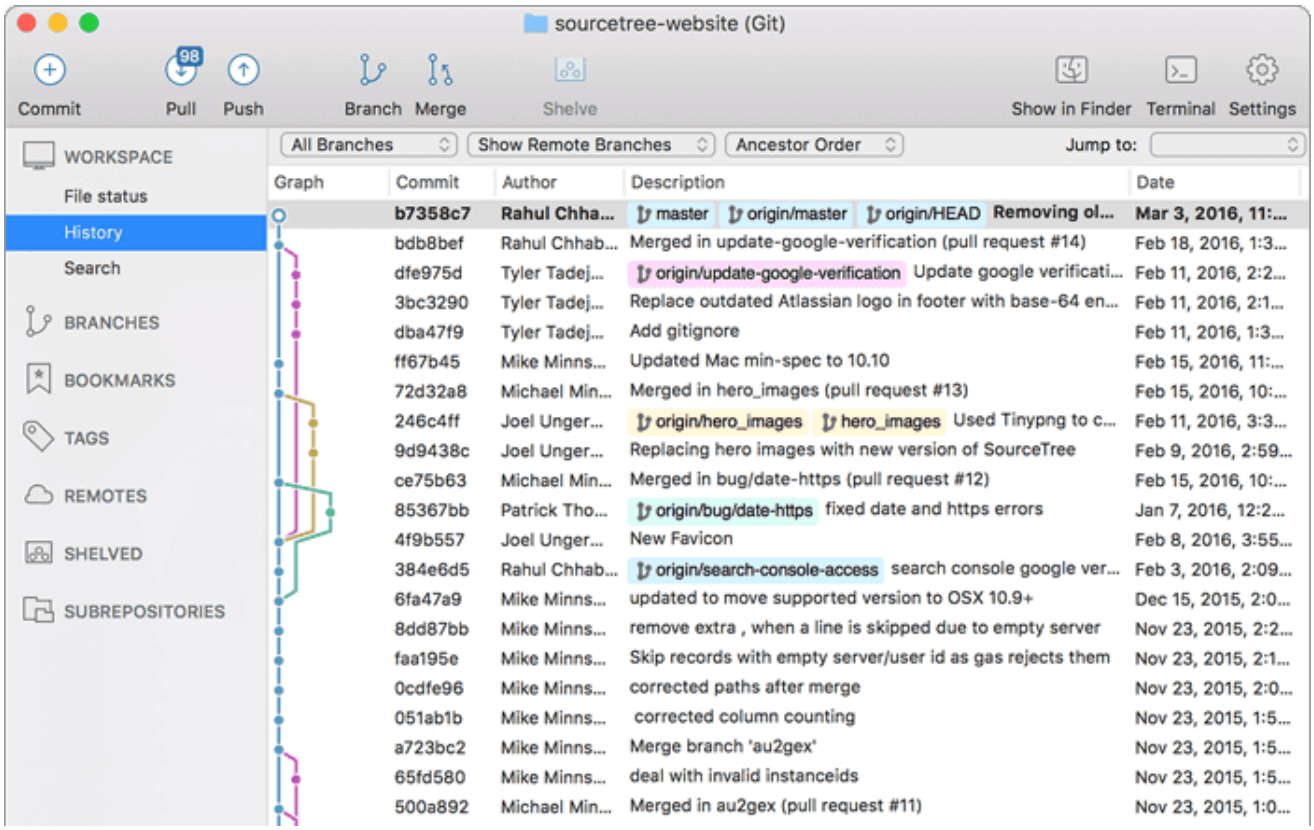
IDEA Git 插件越来越强大，很多时候，我们日常使用 Git ，更多使用它。具体的教程，可以看看 《IntelliJ IDEA 下的使用 git》 。

3）SourceTree

> ：可以说是最好用的 Git 工具，没有之一。

日常使用的一个图形化的 Git 增强工具，而最好用的功能就在于它集成了 GitFlow，让开发者可以更简单、更规范的去做一些 Git 操作；

另外它还提供了更友好的 merge 界面，但是操作起来不是很顺手，因为它只支持整行删除。



4）其它

- SmartGit
- Tower
- Atom

Git 和 SVN 的优缺点?

Git 是分布式版本控制系统，SVN 是集中式版本控制系统。

1）SVN 的优缺点

- 优点
  - 1、管理方便，逻辑明确，符合一般人思维习惯。
  - 2、易于管理，集中式服务器更能保证安全性。
  - 3、代码一致性非常高。
  - 4、适合开发人数不多的项目开发。
- 缺点
  - 1、服务器压力太大，数据库容量暴增。
  - 2、如果不能连接到服务器上，基本上不可以工作，因为 SVN 是集中式服务器，如果服务器不能连接上，就不能提交，还原，对比等等。

    - 3、不适合开源开发（开发人数非常非常多，但是 Google App Engine 就是用 SVN 的）。但是一般集中式管理的有非常明确的权限管理机制（例如分支访问限制），可以实现分层管理，从而很好的解决开发人数众多的问题。

2）Git 优缺点

- 优点
    - 1、适合分布式开发，强调个体。
    - 2、公共服务器压力和数据量都不会太大。
    - 3、速度快、灵活。
    - 4、任意两个开发者之间可以很容易的解决冲突。
    - 5、离线工作。
- 缺点
    - 1、学习周期相对而言比较长。
    - 2、不符合常规思维。
    - 3、代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。

所以，很多公司的开发团队使用 Git 作为版本管理，而产品团队使用 SVN 。

说说创建分支的步骤?

- 1、`git branch xxx_dev`：创建名字为 `xxx_dev` 的分支。
- 2、`git checkout xxx_dev`：切换到名字为 `xxx_dev` 的分支。
- 3、`git push origin xxx_dev`：执行推送的操作，完成本地分支向远程分支的同步。

更详细的，可以看看 《Github 创建新分支》 文章。

📣 **

📣 tag 是什么?

tag ，指向一次 commit 的 id ，通常用来给分支做一个标记。

> 大多数情况下，我们会将每个 Release 版本打一个分支。例如 SkyWalking 的 Tag 是 https://github.com/apache/incubator-skywalking/tags 。

- 打标签：`git tag -a v1.01 -m "Release version 1.01"` 。
- 提交标签到远程仓库：`git push origin --tags` 。
- 查看标签：`git tag` 。
- 查看某两次 tag 之间的 commit：`git log --pretty=oneline tagA..tagB` 。
- 查看某次 tag 之后的 commit：`git log --pretty=oneline tagA..` 。

📣 Git 提交代码时候写错 commit 信息后，如何重新设置 commit 信息?

可以通过 `git commit --amend` 来对本次 commit 进行修改。

📣 删除已经合并过的分支会发生什么事?

分支本身就像是指标或贴纸一样的东西，它指着或贴在某个 commit 上面，分支并不是目录或档桉的复制品（但在有些版控系统的确是）。

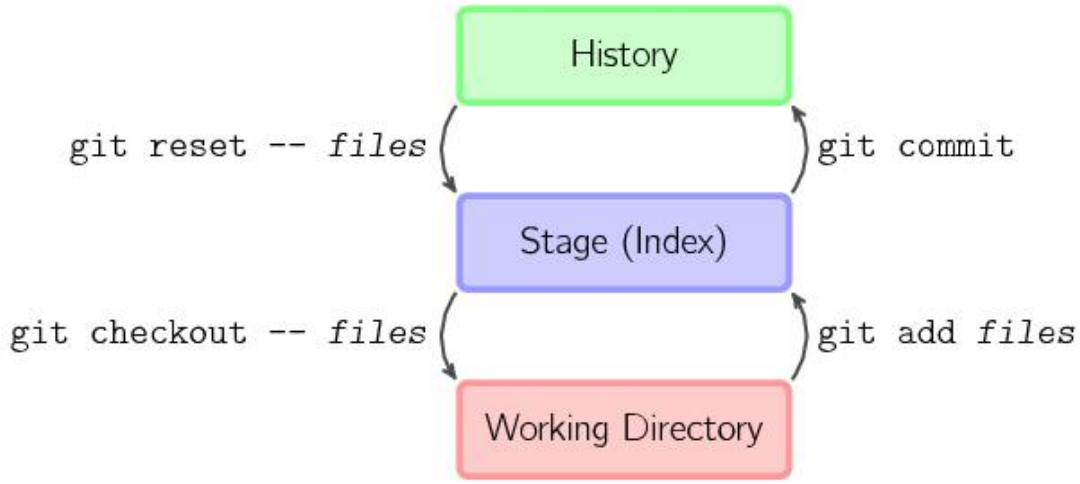
在 Git 裡，删除分支就像是你把包装盒上的贴纸撕下来，贴纸撕掉了，盒子并不会就这样跟着消失。所以，当你删除合并过的分支不会发生什么事，也不会造成档桉或目录跟着被删除的状况。

add 和 stage 有什么区别?

在回答这个问题之前需要先了解 Git 仓库的三个组成部分：

- 工作区(Working Directory)：在 Git 管理下的正常目录都算是工作区，我们平时的编辑工作都是在工作区完成。
- 暂存区(Stage)：临时区域。里面存放将要提交文件的快照。
- 历史记录区(History)：`git commit` 后的记录区。

然后，是这三个区的转换关系以及转换所使用的命令：



再然后，我们就可以来说一下 `git add` 和 `git stage` 了。

- **其实，他们两是同义的**，所以，惊不惊喜，意不意外？这个问题竟然是个陷阱...引入 `git stage` 的原因其实比较有趣：是因为要跟 `svn add` 区分，两者的功能是完全不一样的，svn add 是将某个文件加入版本控制，而 git add 则是把某个文件加入暂存区。
- 因为在 Git 出来之前大家用 SVN 比较多，所以为了避免误导，Git 引入了 `git stage` ，然后把 `git diff --staged` 做为 `git diff --cached` 的相同命令。基于这个原因，我们建议使用 `git stage` 以及 `git diff --staged` 。

如何从 Git 中删除文件，而不将其从文件系统中删除?

如果你在 `git add` 过程中误操作，你最终会添加不想提交的文件。但是，`git rm` 则会把你的文件从你暂存区（索引）和文件系统（工作树）中删除，这可能不是你想要的。

换成 `git reset` 操作：

```
git reset filename          # or
echo filename >> .gitingore # add it to .gitignore to avoid re-adding it
```

- 上面意思是，`git reset <file>` 是 `git add <file>` 的逆操作。

merge 和 rebase 的有什么区别?

Git 合并的两种方法。

- `git merge` ，把本地代码和已经取得的远程仓库代码合并。

- `git rebase`，是复位基底的意思。

`git merge` 会生成一个新的节点，之前的提交会分开显示；而 `git rebase` 操作不会生成新的操作，将两个分支融合成一个线性的提交。

推荐看看 《git rebase 和 git merge 的区别》 和 《git merge 和 git rebase 的区别》。

📣 什么时候使用 rebase 代替 merge？

这两个命令都是把修改从一个分支集成到另一个分支上，它们只是以非常不同的方式进行。

- 考虑一下场景，在合并和变基前：

```
A <- B <- C    [master]
^
 \
  D <- E       [branch]
```

- 在 `git merge master` 之后：

```
A <- B <- C
^         ^
 \         \
  D <- E <- F
```

- 在 `git rebase master` 之后：

```
A <- B <- C <- D <- E
```

使用变基时，意味着使用另一个分支作为集成修改的新基础。

- 何时使用：
  - 如果你对修改不够果断，请使用合并操作。
  - 根据你希望的历史记录的样子，而选择使用变基或合并操作。
- 更多需要考虑的因素：
  - 分支是否与团队外部的开发人员共享修改（如开源、公开项目）？如果是这样，请不要使用变基操作。变基会破坏分支，除非他们使用 `git pull --rebase`，否则这些开发人员将会得到损坏的或不一致的仓库。
  - 你的开发团队技术是否足够娴熟？变基是一种破坏性操作。这意味着，如果你没有正确使用它，你可能会丢失提交，并且/或者会破坏其他开发者仓库的一致性。
  - 分支本身是否代表有用的信息？一些团队使用功能分支（branch-per-feature）模式，每个分支代表一个功能（或错误修复，或子功能等）。在此模式中，分支有助于识别相关提交的集合。在每个开发人员分支（branch-per-developer）模式中，分支本身不会传达任何其他信息（提交信息已有作者）。则在这种模式下，变基不会有任何破坏。
  - 是否无论如何都要还原合并？恢复（如在撤销中）变基，是相当困难的，并且/或者在变基中存在冲突时，是不可能完成的。如果你考虑到日后可能需要恢复，请使用合并操作。

📣 reset 与 rebase 有什么区别?

- reset 操作，不修改 commit 相关的东西，只会去修改 `.git` 目录下的东西。

- rebase 操作，会试图修改你已经 commit 的东西，比如覆盖 commit 的历史等，但是不能使用 rebase 来修改已经 push 过的内容，容易出现兼容性问题。rebase 还可以来解决内容的冲突，解决两个人修改了同一份内容，然后失败的问题。

推荐看看 《“git reset”和“git rebase”有什么区别？》 。

📣 reset 与 revert 与 checkout 有什么区别?

首先是它们的共同点：用来撤销代码仓库中的某些更改。

然后是不同点：

- 1）从 commit 层面来说：
  - `git reset` ，可以将一个分支的末端指向之前的一个 commit 。然后再下次 Git 执行垃圾回收的时候，会把这个 commit 之后的 commit 都扔掉。`git reset` 还支持三种标记，用来标记 reset 指令影响的范围：
    - `--mixed` ： 会影响到暂存区和历史记录区。也是默认选项；
    - `--soft` ： 只影响历史记录区；
    - `--hard` ：影响工作区、暂存区和历史记录区。

      > 注意：因为 `git reset` 是直接删除 commit 记录，从而会影响到其他开发人员的分支，所以不要在公共分支（比如 develop）做这个操作。
  - `git checkout` ，可以将 HEAD 移到一个新的分支，并更新工作目录。因为可能会覆盖本地的修改，所以执行这个指令之前，你需要 stash 或者 commit 暂存区和工作区的更改。
  - `git revert` ，和 `git reset` 的目的是一样的，但是做法不同，它会以创建新的 commit 的方式来撤销 commit ，这样能保留之前的 commit 历史，比较安全。另外，同样因为可能会覆盖本地的修改，所以执行这个指令之前，你需要 stash 或者 commit 暂存区和工作区的更改。
- 2）从文件层面来说
  - `git reset` ，只是把文件从历史记录区拿到暂存区，不影响工作区的内容，而且不支持 `--mixed` 、`--soft` 和 `--hard` 。
  - `git checkout` ，则是把文件从历史记录拿到工作区，不影响暂存区的内容。
  - `git revert` ，不支持文件层面的操作。

总的来说，回答关键点：

- 对于 commit 层面和文件层面，这三个指令本身功能差别很大。
- `git revert` 不支持文件层面的操作。
- 不要在公共分支做 `git reset` 操作。

📣 不小心用 git reset --hard 指令把提交理掉了，有机会救回來吗?

放心，基本上东西进了 Git 就不容易消失，它們只是以一种我们肉眼看不懂的格式存放在 Git 空间裡。我们可以透過 `git reflog` 指令去翻一下被 reset 的那個 Commit 的编号值，然後再做一次 `git reset --hard` 就可以把它救回來了。

Git 如何解决代码冲突?

```
git stash
git pull
git stash pop
```

> 最常用。

- 这个操作就是把自己修改的代码隐藏，然后把远程仓库的代码拉下来，然后把自己隐藏的修改的代码释放出来，让 Git 自动合并。

```
git reset –hard
git pull
```

📣 假如你现在的分支为 main_dev，并在这个分支上修复了一个Bug，但是在 main_zh_test 分支也发现了同样的一个 Bug，如果不用 copy 代码的方式，你如何把 main_dev 修复这个 Bug 提交的代码合并到 main_zh_test 分支上，请贴出你的 Git 操作指令和指令的含义?

> 假设合并时没有冲突

- 1、在 `main_dev` 分支上，通过 gitlog 命令，使用 bugid 搜索提交的 commit id 。
- 2、使用 `git checkout main_zh_test` 命令，切换到 `main_zh_test` 分支。
- 3、使用 `git cherry-pick commitid` 将对 Bug 的修改批量移植到该分支上。
- 4、`git commit`，提交到本地。
- 5、`git push`，推送到远程仓库。

所以，重心在于 cherry-pick 使用，参见 《git cherry-pick 使用指南》 文章。

📣 如果你正在某个分支进行开发，突然被老叫去修别的问题，这时候你会怎么处理手边的工作?

- 一种是直接先 `git commit`，等要处理的问题解決後再回來這個分支，再 `git reset` 把 Commit 拆开來继续接著做。
- 另一种做法，则是使用 `git stash` 指令，先把目前的进度存在 stash 上，等任务結束后可以再使用 `git stash pop` 或 `git stash apply` 把当时的及大怒再拿出來。

pull 与 fetch 有什么区别?

```
pull = fetch + merge
```

- 使用 `git fetch` 是取回远端更新，不会对本地执行 merge 操作，不会去动你的本地的内容。
- 而是用 ·git pull ` 会更新你本地代码到服务器上对应分支的最新版本。
- 如果要代码库的文件完全覆盖本地版本。

什么是 fork 操作?

fork，是对一个仓库的克隆。克隆一个仓库允许你自由试验各种改变，而不影响原始的项目。

一般来说，fork 被用于去更改别人的项目（贡献代码给已经开源的项目）或者使用别人的项目作为你自己想法的初始开发点。

> 使用 fork 提出改变的一个很好的例子是漏洞修复。与其记录一个你发现的问题，不如：
>
> - fork 这个仓库

> - 进行修复
> - 向这个项目的拥有者提交一个 pull requset
>
> 如果这个项目的拥有者认同你的成果，他们可能会将你的修复更新到原始的仓库中！

- 目前很多开源项目，采用 fork + pull request 的方式，实现新功能的开发，Code Review 等等。

📣 Fork 和 Clone 有什么区别?

Clone ，不是 Fork ，克隆是个对某个远程仓库的本地拷贝。克隆时，实际上是拷贝整个源存储仓库，包括所有历史记录和分支。

📣 Fork 和 Branch 有什么区别?

Branch ，是一种机制，用于处理单一存储仓库中的变更，并最终目的是用于与其他部分代码合并。

Git 服务器

Git 服务器的选择，实际上是比较多的。

- 公有服务方案
  - Github
  - Gitee
- 私有化部署方案
  - GitLab
  - Gogs
  - Bitbucket

    > 注意，Gitlab 和 Bitbucket 也提供公有服务的方案。

一般情况下，大多数公司使用 GitLab 作为 Git 服务器。

> GitLab是一个利用 Ruby on Rails 开发的开源应用程序，实现一个自托管的Git项目仓库，可通过Web界面进行访问公开的或者私人项目。
>
> 它拥有与Github类似的功能，能够浏览源代码，管理缺陷和注释。可以管理团队对仓库的访问，它非常易于浏览提交过的版本并提供一个文件历史库。它还提供一个代码片段收集功能可以轻松实现代码复用，便于日后有需要的时候进行查找。

- 不过因为 GitLb 使用 Ruby on Rails 实现，所以占用的系统资源会比较多。

【重要】Git 工作流

Git 因为其灵活性，所以提供了多种工作流的方式：

1. 集中式工作流。
2. 功能分支工作流。
3. Gitflow 工作流

Git 集中式工作流

参见 《Git 工作流指南：集中式工作流》 文章。

Git 功能分支工作流

参见 《Git 工作流指南：功能分支工作流》 文章。

Gitflow 工作流

参见 《Git 工作流指南：Gitflow 工作流》 文章。

- 两个长期维护分支
  - 主分支(master)
  - 开发分支(develop)
- 三种短期分支
  - 功能分支(feature branch)
  - 补丁分支(hotfix branch)
  - 预发分支(release branch)

GitFlow 的优势有哪些?

- 1、并行开发：GitFlow 可以很方便的实现并行开发：每个新功能都会建立一个新的 feature 分支，从而和已经完成的功能隔离开来，而且只有在新功能完成开发的情况下，其对应的 feature 分支才会合并到主开发分支上（也就是我们经常说的 develop 分支）。另外，如果你正在开发某个功能，同时又有一个新的功能需要开发，你只需要提交当前 feature 的代码，然后创建另外一个 feature 分支并完成新功能开发。然后再切回之前的 feature 分支即可继续完成之前功能的开发。
- 2、协作开发：GitFlow 还支持多人协同开发，因为每个 feature 分支上改动的代码都只是为了让某个新的 feature 可以独立运行。同时我们也很容易知道每个人都在干啥。
- 3、发布阶段：当一个新 feature 开发完成的时候，它会被合并到 develop 分支，这个分支主要用来暂时保存那些还没有发布的内容，所以如果需要再开发新的 feature，我们只需要从 develop 分支创建新分支，即可包含所有已经完成的 feature 。
- 4、支持紧急修复：GitFlow 还包含了 hotfix 分支。这种类型的分支是从某个已经发布的 tag 上创建出来并做一个紧急的修复，而且这个紧急修复只影响这个已经发布的 tag，而不会影响到你正在开发的新 feature 。

Forking 工作流程

Forking 工作流程，与其他流行的 Git 工作流程有着根本的区别。它不是用单个服务端仓库充当“中央”代码库，而是为每个开发者提供自己的服务端仓库。Forking 工作流程最常用于公共开源项目中。

- Forking 工作流程的**主要优点**是可以汇集提交贡献，又无需每个开发者提交到一个中央仓库中，从而实现干净的项目历史记录。开发者可以推送（push）代码到自己的服务端仓库，而只有项目维护人员才能直接推送（push）代码到官方仓库中。
- 当开发者准备发布本地提交时，他们的提交会推送到自己的公共仓库中，而不是官方仓库。然后他们向主仓库提交请求拉取（pull request），这会告知项目维护人员有可以集成的更新。

当然，这并不是说Forking 工作流程和上述的工作流是冲突的关系，而是可以相互结合。目前，很多公司都采用**Gitflow 工作流 + Forking 工作流程结合**的方式。为什么呢?

- 1、对于主仓库，读权限是所有开发人员都有，但是写权限是只有部分“管理”开发人员有，从而对主仓库的统一管理。

  > 如果不酱紫，主仓库岂不是可以被各种乱改。

- 2、所有开发人员，提交代码到自己的服务端仓库，通过 pull request 到主仓库。这样，“管理”开发人员就可以对代码进行 Code Review ，保证代码质量。

解释下 PR 和 MR 的区别?

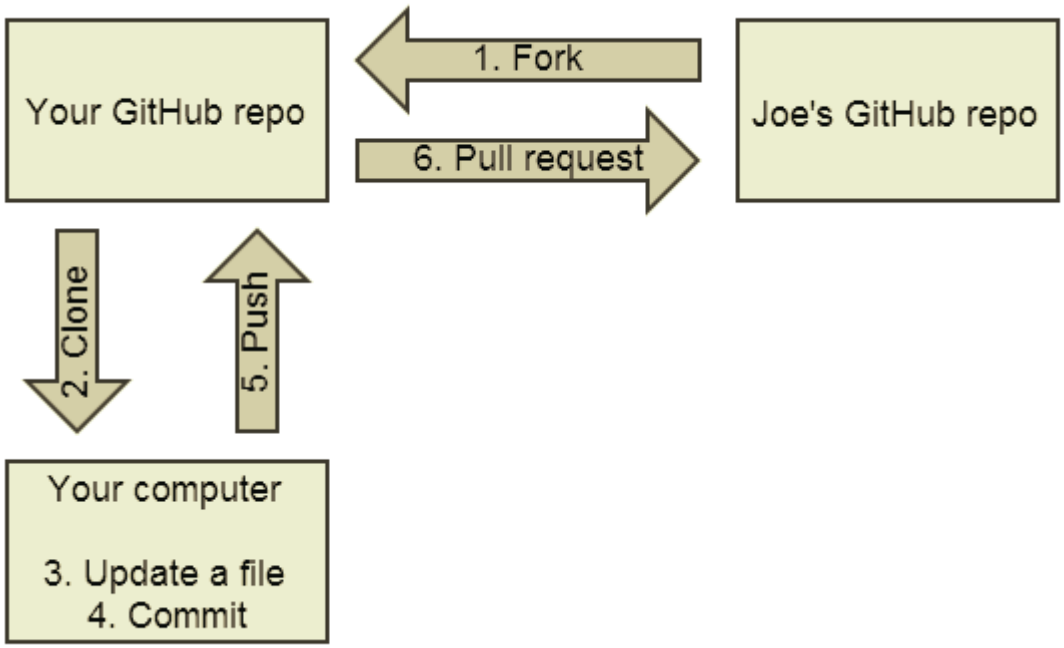
PR 和 MR 的全称分别是 pull request 和 merge request。

🚀 解释它们两者的区别之前，我们需要先了解一下 Code Review，因为 PR 和 MR 的引入正是为了进行 Code Review。

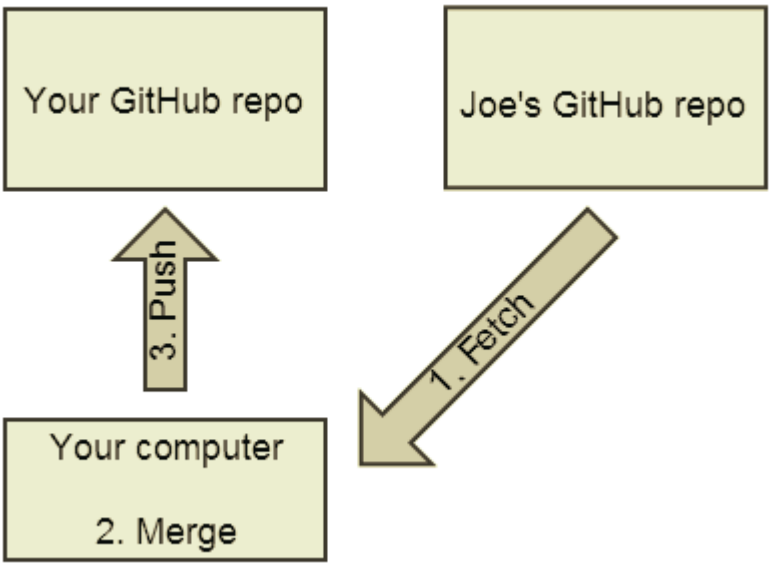
- Code Review 是指在开发过程中，对代码的系统性检查。通常的目的是查找系统缺陷，保证代码质量和提高开发者自身水平。 Code Review 是轻量级代码评审，相对于正式代码评审，轻量级代码评审所需要的各种成本要明显低的多，如果流程正确，它可以起到更加积极的效果。
- 进行 Code Review 的原因：
  - 提高代码质量
  - 及早发现潜在缺陷与 BUG ，降低事故成本。
  - 促进团队内部知识共享，提高团队整体水平
  - 评审过程对于评审人员来说，也是一种思路重构的过程，帮助更多的人理解系统。

🚀 然后我们需要了解下 fork 和 branch ，因为这是 PR 和 MR 各自所属的协作流程。

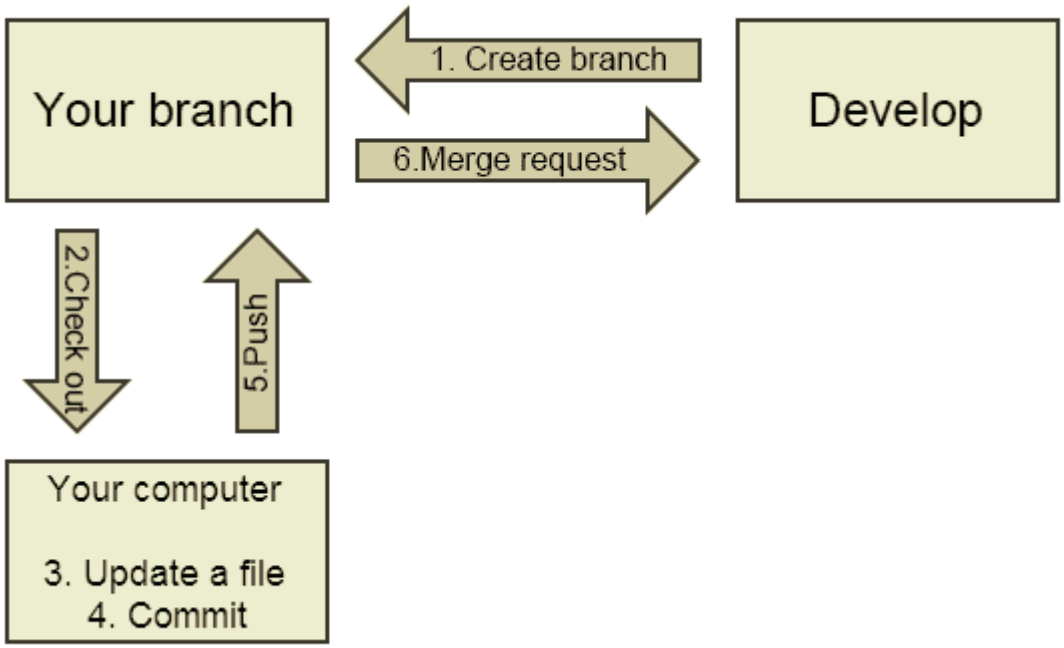
- **fork** 是 git 上的一个协作流程。通俗来说就是把别人的仓库备份到自己仓库，修修改改，然后再把修改的东西提交给对方审核，对方同意后，就可以实现帮别人改代码的小目标了。fork 包含了两个流程：
  - fork 并更新某个仓库



- 同步 fork



- 和 fork 不同，**branch** 并不涉及其他的仓库，操作都在当前仓库完成。



考察关键点：

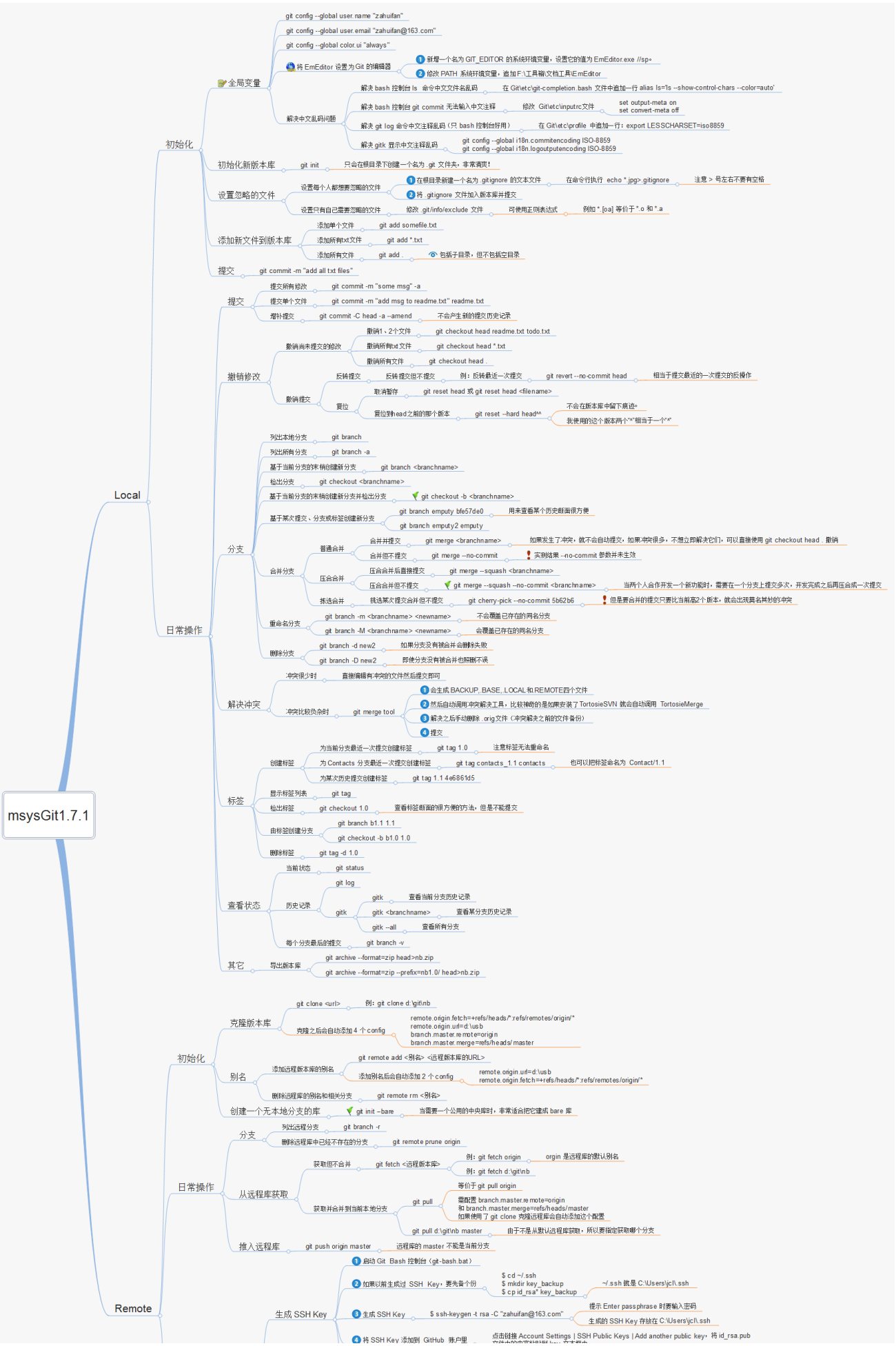
- Code review；
- PR 和 MR 所属流程的细节。

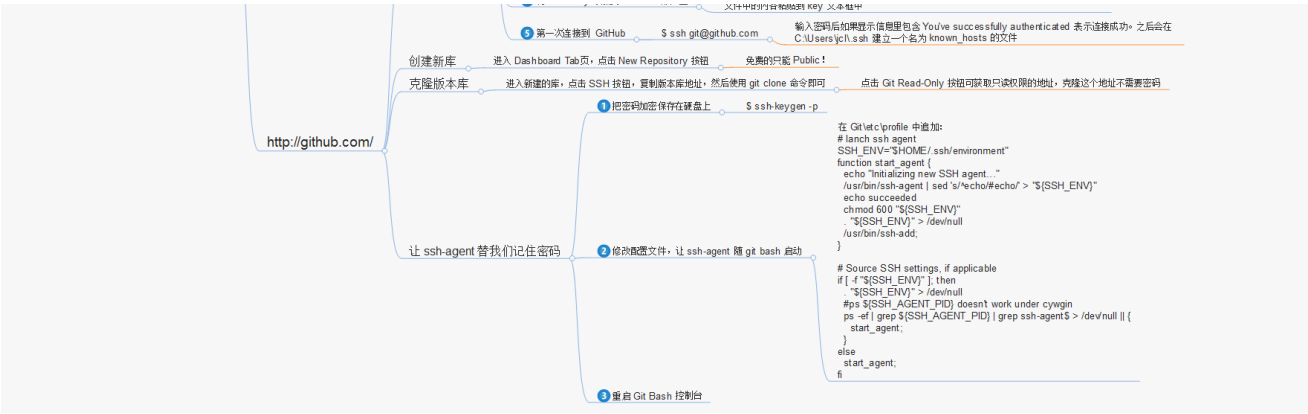
回答关键点：

- 回答这个问题的时候不要单单只说它们的区别。而是要从 PR 和 MR 产生的原因，分析它们所属的流程，然后再得出两者的区别。

666. 彩蛋

在网络上，找到一个牛逼的 Git 脑图：


msysGit1.7.1
Local
初始化
全局变量
git config --global user.name "zahuifan"
git config --global user.email "zahuifan@163.com"
git config --global color.ui "always"
将 EmEditor 设置为 Git 的编辑器
新增一个名为 GIT_EDITOR 的系统环境变量，设置它的值为 EmEditor.exe //sp-
修改 PATH 系统环境变量，追加 F:\工具箱\文档工具\EmEditor
解决中文乱码问题
解决 bash 控制台 ls 命令中文文件名乱码
在 Git\etc\git-completion.bash 文件中追加一行 alias ls='ls --show-control-chars --color=auto'
解决 bash 控制台 git commit 无法输入中文注释
修改 Git\etc\inputrc 文件
set output-meta on
set convert-meta off
解决 git log 命令中文注释乱码（只 bash 控制台好用）
在 Git\etc\profile 中追加一行：export LESSCHARSET=iso8859
解决 gitk 显示中文注释乱码
git config --global i18n.commitencoding ISO-8859
git config --global i18n.logoutputencoding ISO-8859
初始化新版本库
git init
只会在根目录下创建一个名为 .git 文件夹，非常满意！
设置忽略的文件
设置每个人都想要忽略的文件
在根目录新建一个名为 .gitignore 的文本文件
在命令行执行 echo *.jpg> .gitignore
注意 > 号左右不要有空格
将 .gitignore 文件加入版本库并提交
设置只有自己需要忽略的文件
修改 .git/info/exclude 文件
可使用正则表达式
例如 *.[oa] 等价于 *.o 和 *.a
添加新文件到版本库
添加单个文件
git add somefile.txt
添加所有txt文件
git add *.txt
添加所有文件
git add .
包括子目录，但不包括空目录
提交
git commit -m "add all txt files"
日常操作
提交
提交所有修改
git commit -m "some msg" -a
提交单个文件
git commit -m "add msg to readme.txt" readme.txt
增补提交
git commit -C head -a --amend
不会产生新的提交历史记录
撤销修改
撤销尚未提交的修改
撤销1、2个文件
git checkout head readme.txt todo.txt
撤销所有txt文件
git checkout head *.txt
撤销所有文件
git checkout head .
撤销提交
反转提交
反转提交但不提交
例：反转最近一次提交
git revert --no-commit head
相当于提交最近的一次提交的反操作
复位
取消暂存
git reset head 或 git reset head <filename>
复位到head之前的那个版本
git reset --hard head^^
不会在版本库中留下痕迹-
我使用的这个版本两个"^"相当于一个"~"
分支
列出本地分支
git branch
列出所有分支
git branch -a
基于当前分支的末梢创建新分支
git branch <branchname>
检出分支
git checkout <branchname>
基于当前分支的末梢创建新分支并检出分支
git checkout -b <branchname>
基于某次提交、分支或标签创建新分支
git branch emputy bfe57de0
用来查看某个历史截面很方便
git branch emputy2 emputy
合并分支
普通合并
合并并提交
git merge <branchname>
如果发生了冲突，就不会自动提交，如果冲突很多，不想立即解决它们，可以直接使用 git checkout head . 撤销
合并但不提交
git merge --no-commit
实测结果 --no-commit 参数并未生效
压合合并
压合合并后直接提交
git merge --squash <branchname>
压合合并但不提交
git merge --squash --no-commit <branchname>
当两个人合作开发一个新功能时，需要在一个分支上提交多次，开发完成之后再压合成一次提交
拣选合并
挑选某次提交合并但不提交
git cherry-pick --no-commit 5b62b6
但是要合并的提交只要比当前高2个版本，就会出现莫名其妙的冲突
重命名分支
git branch -m <branchname> <newname>
不会覆盖已存在的同名分支
git branch -M <branchname> <newname>
会覆盖已存在的同名分支
删除分支
git branch -d new2
如果分支没有被合并会删除失败
git branch -D new2
即使分支没有被合并也照删不误
解决冲突
冲突很少时
直接编辑有冲突的文件然后提交即可
冲突比较负杂时
git merge tool
会生成 BACKUP, BASE, LOCAL 和 REMOTE 四个文件
然后自动调用冲突解决工具，比较神奇的是如果安装了 TortosieSVN 就会自动调用 TortosieMerge
解决之后手动删除 .orig 文件（冲突解决之前的文件备份）
提交
标签
创建标签
为当前分支最近一次提交创建标签
git tag 1.0
注意标签无法重命名
为 Contacts 分支最近一次提交创建标签
git tag contacts_1.1 contacts
也可以把标签命名为 Contact/1.1
为某次历史提交创建标签
git tag 1.1 4e6861d5
显示标签列表
git tag
检出标签
git checkout 1.0
查看标签断面的很方便的方法，但是不能提交
由标签创建分支
git branch b1.1 1.1
git checkout -b b1.0 1.0
删除标签
git tag -d 1.0
查看状态
当前状态
git status
历史记录
git log
gitk
gitk
查看当前分支历史记录
gitk <branchname>
查看某分支历史记录
gitk --all
查看所有分支
每个分支最后的提交
git branch -v
其它
导出版本库
git archive --format=zip head>nb.zip
git archive --format=zip --prefix=nb1.0/ head>nb.zip
Remote
初始化
克隆版本库
git clone <url>
例：git clone d:\git\nb
克隆之后会自动添加 4 个 config
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
remote.origin.url=d:\usb
branch.master.remote=origin
branch.master.merge=refs/heads/master
别名
添加远程版本库的别名
git remote add <别名> <远程版本库的URL>
添加别名后会自动添加 2 个 config
remote.origin.url=d:\usb
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
删除远程库的别名和相关分支
git remote rm <别名>
创建一个无本地分支的库
git init --bare
当需要一个公用的中央库时，非常适合把它建成 bare 库
日常操作
分支
列出远程分支
git branch -r
删除远程库中已经不存在的分支
git remote prune origin
从远程库获取
获取但不合并
git fetch <远程版本库>
例：git fetch origin
origin 是远程库的默认别名
例：git fetch d:\git\nb
获取并合并到当前本地分支
git pull
等价于 git pull origin
需配置 branch.master.remote=origin
和 branch.master.merge=refs/heads/master
如果使用了 git clone 克隆远程库会自动添加这个配置
git pull d:\git\nb master
由于不是从默认远程库获取，所以要指定获取哪个分支
推入远程库
git push origin master
远程库的 master 不能是当前分支
生成 SSH Key
启动 Git Bash 控制台（git-bash.bat）
如果以前生成过 SSH Key，要先备个份
$ cd ~/.ssh
$ mkdir key_backup
$ cp id_rsa* key_backup
~/.ssh 就是 C:\Users\jcl\.ssh
生成 SSH Key
$ ssh-keygen -t rsa -C "zahuifan@163.com"
提示 Enter passphrase 时要输入密码
生成的 SSH Key 存放在 C:\Users\jcl\.ssh
将 SSH Key 添加到 GitHub 账户里
点击链接 Account Settings | SSH Public Keys | Add another public key，将 id_rsa.pub



参考与推荐如下文章：

- 《Git 面試題》

  > 读者写的非常棒，即使不准备面试，也可以看看，作为平时使用 Git 一些场景下的解决方案。

- 《面试当中的 Git 问题》
- 《Git 的常见问题以及面试题汇总》
- 《泪流满面的 11 个 Git 面试题》
- 《面试中的那些 Git 问题 - 基础部分》