



GIT （分布式版本控制系统）

Git的诞生

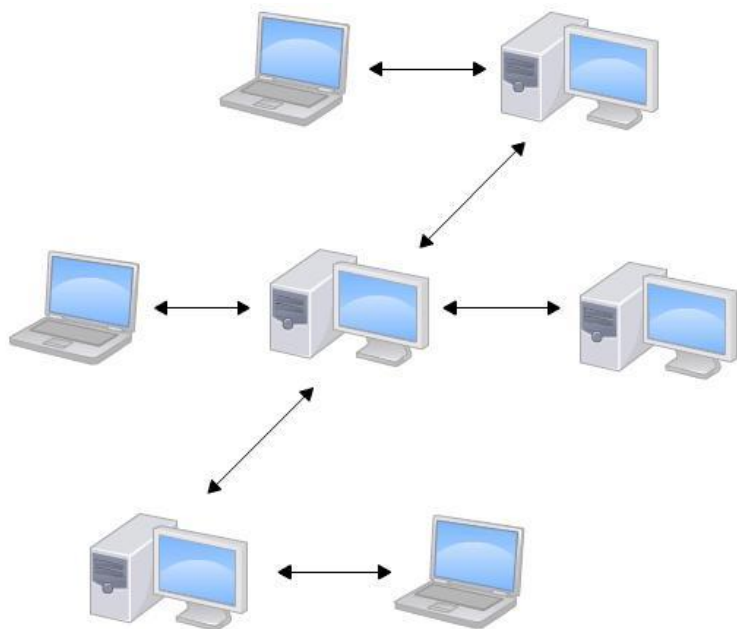
- Git是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。Git的读音为/git/。
- Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。
- Git 与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持。

Git 与 SVN 区别

- GIT不仅仅是个版本控制系统，它也是个内容管理系统(CMS),工作管理系统等。
- 如果你是一个具有使用SVN背景的人，你需要做一定的思想转换，来适应GIT提供的一些概念和特征。
- Git 与 SVN 区别点：
 - 1、GIT是分布式的，SVN不是：这是GIT和其它非分布式的版本控制系统，例如SVN，CVS等，最核心的区别。
 - 2、GIT把内容按元数据方式存储，而SVN是按文件：所有的资源控制系统都是把文件的元信息隐藏在一个类似.svn,.cvs等的文件夹里。
 - 3、GIT分支和SVN的分支不同：分支在SVN中一点不特别，就是版本库中的另外的一个目录。
 - 4、GIT没有一个全局的版本号，而SVN有：目前为止这是跟SVN相比GIT缺少的最大的一个特征。
 - 5、GIT的内容完整性要优于SVN：GIT的内容存储使用的是SHA-1哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏。

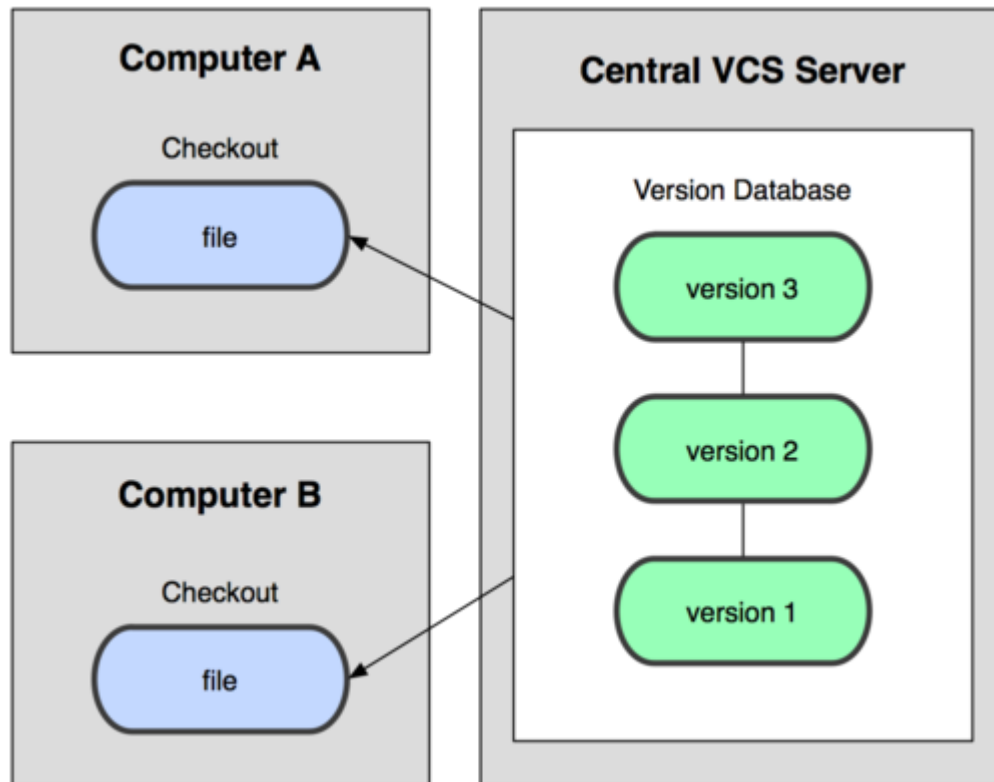
集中式vs分布式

- 集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。
- 分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。



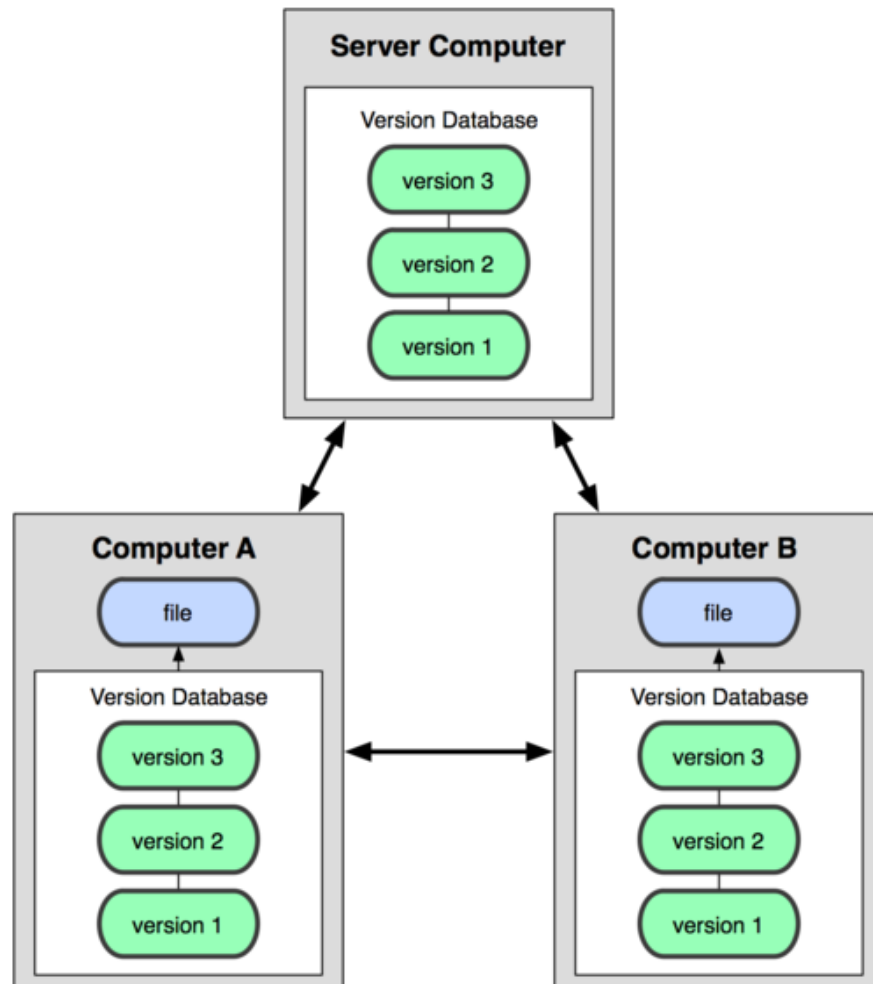
集中化的版本控制系统

- 集中化的版本控制系统 (Centralized Version Control Systems , 简称 CVCS)



分布式版本控制系统

- 分布式版本控制系统 (Distributed Version Control System , 简称 DVCS)



优缺点

- **优点：**

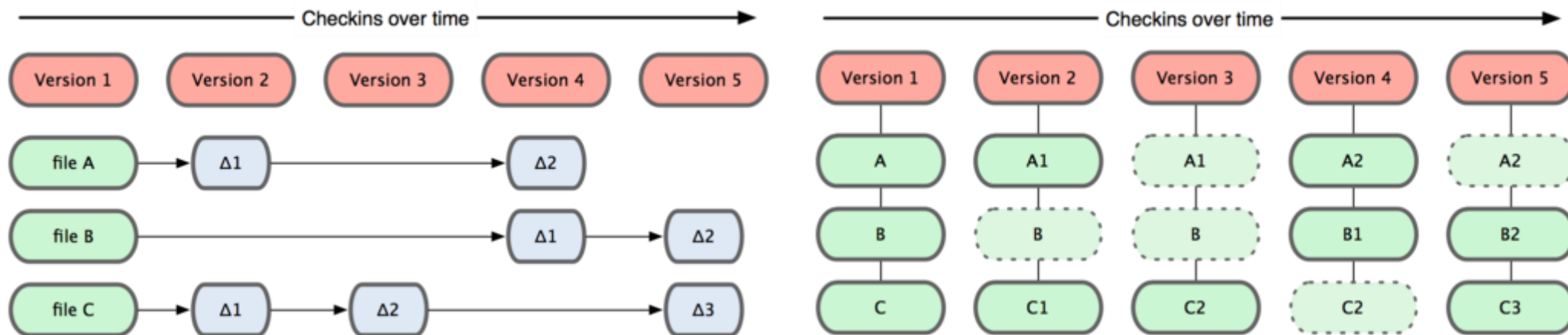
- 适合分布式开发，强调个体。
- 公共服务器压力和数据量都不会太大，速度快、灵活。
- 任意两个开发者之间可以很容易的解决冲突。
- 离线工作。

- **缺点：**

- 学习周期相对而言比较长。
- 不符合常规思维。
- 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。

Git原理

- Git 和其他版本控制系统的主要差别在于，Git 只关心文件数据的整体是否发生变化，而大多数其他系统则只关心文件内容的具体差异。这类系统（CVS，Subversion，Perforce，Bazaar 等等）每次记录有哪些文件作了更新，以及都更新了哪些行的什么内容，如左图。



- 如右图
- Git 并不保存这些前后变化的差异数据。实际上，Git 更像是把变化的文件作快照后，记录在一个微型的文件系统中。每次提交更新时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照 的索引。为提高性能，若文件没有变化，Git 不会再次保存，而只对上次保存的快照作一链接。

Git原理

- **近乎所有操作都是本地执行**
- 在 Git 中的绝大多数操作都只需要访问本地文件和资源，不用连网。但如果用 CVCS 的话，差不多所有操作都需要连接网络。因为 Git 在本地磁盘上就保存着所有当前项目的历史更新，所以处理起来速度飞快。
- 举个例子，如果要浏览项目的历史更新摘要，Git 不用跑到外面的服务器上去取数据回来，而直接从本地数据库读取后展示给你看。所以任何时候你都可以马上翻阅，无需等待。如果想要看当前版本的文件和一个月前的版本之间有何差异，Git 会取出一个月前的快照和当前文件作一次差异运算，而不用请求远程服务器来做这件事，或是把老版本的文件拉到本地来作比较。
- 用 CVCS 的话，没有网络或者断开 VPN 你就无法做任何事情。但用 Git 的话，就算你在飞机或者火车上，都可以非常愉快地频繁提交更新，等到了有网络的时候再上传到远程仓库。同样，在回家的路上，不用连接 VPN 你也可以继续工作。换作其他版本控制系统，这么做几乎不可能，抑或非常麻烦。比如 Perforce，如果不连到服务器，几乎什么都做不了（译注：默认无法发出命令 `p4 edit file` 开始编辑文件，因为 Perforce 需要联网通知系统声明该文件正在被谁修订。但实际上手工修改文件权限可以绕过这个限制，只是完成后还是无法提交更新。）；如果是 Subversion 或 CVS，虽然可以编辑文件，但无法提交更新，因为数据库在网络上。看上去好像这些都不是什么大问题，但实际体验过之后，你就会惊喜地发现，这其实是会带来很大不同的。

Git原理

- **时刻保持数据完整性**
- 在保存到 Git 之前，所有数据都要进行内容的校验和（checksum）计算，并将此结果作为数据的唯一标识和索引。换句话说，不可能在你修改了文件或目录之后，Git 一无所知。这项特性作为 Git 的设计哲学，建在整体架构的最底层。所以如果文件在传输时变得不完整，或者磁盘损坏导致文件数据缺失，Git 都能立即察觉。
- Git 使用 SHA-1 算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个 SHA-1 哈希值，作为指纹字符串。该字符串由 40 个十六进制字符（0-9 及 a-f）组成，看起来就像是：
- 24b9da6552252987aa493b52f8696cd6d3b00373
- Git 的工作完全依赖于这类指纹字符串，所以你会经常看到这样的哈希值。实际上，所有保存在 Git 数据库中的东西都是用此哈希值来作索引的，而不是靠文件名。

Git原理

- **多数操作仅添加数据**

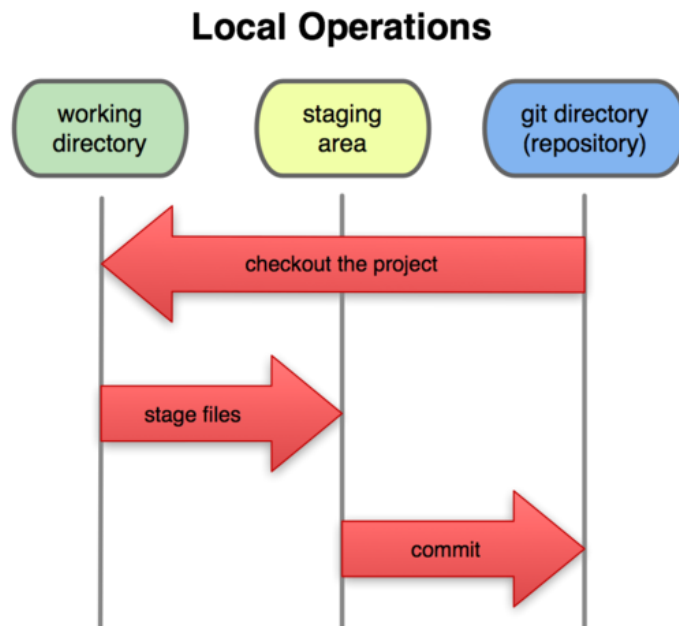
- 常用的 Git 操作大多仅仅是把数据添加到数据库。因为任何一种不可逆的操作，比如删除数据，都会使回退或重现历史版本变得困难重重。在别的 VCS 中，若还未提交更新，就有可能丢失或者混淆一些修改的内容，但在 Git 里，一旦提交快照之后就完全不用担心丢失数据，特别是养成定期推送到其他仓库的习惯的话。
- 这种高可靠性令我们的开发工作安心不少，尽管去做各种试验性的尝试好了，再怎样也不会弄丢数据。

Git原理

- **文件的三种状态**

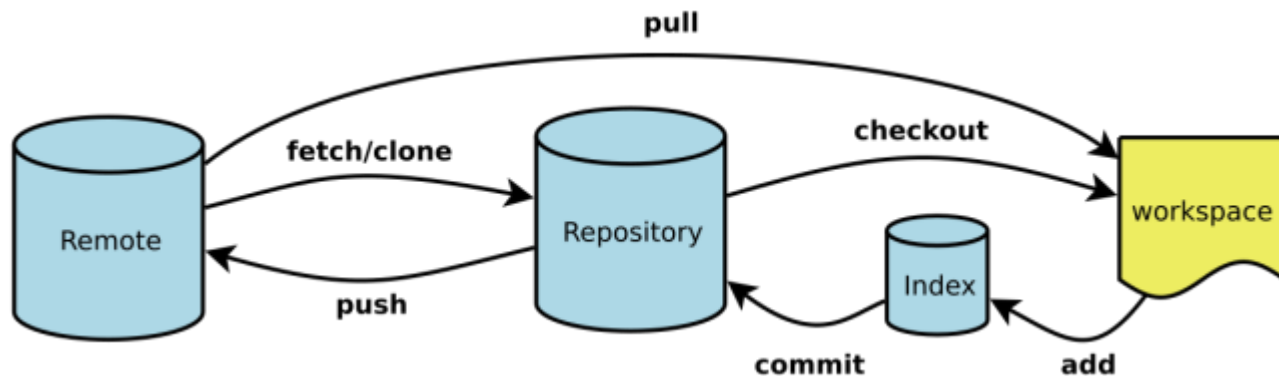
- 对于任何一个文件，在 Git 内都只有三种状态：已提交（committed），已修改（modified）和已暂存（staged）。
- 已提交表示该文件已经被安全地保存在本地数据库中了；
- 已修改表示修改了某个文件，但还没有提交保存；
- 已暂存表示把已修改的文件放在下次提交时要保存的清单中。

由此我们看到 Git 管理项目时，文件流转的三个工作区域：Git 的工作目录，暂存区域，以及版本库（Repository）本地仓库。



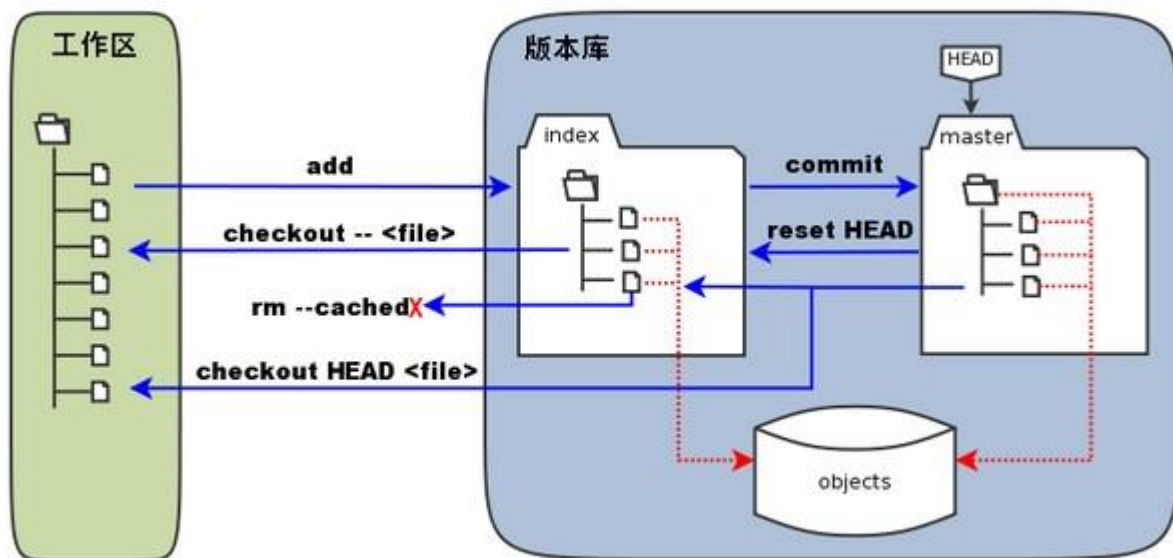
Git 工作区、暂存区和版本库

- **Workspace** : 工作区
Index/Stage : 暂存区, 也叫索引
Repository : 版本库 (仓库区、或本地仓库), 也存储库
Remote : 远程仓库



Git 工作区、暂存区和版本库

- **工作区**：就是你在电脑里能看到的目录。
- **暂存区**：英文叫stage, 或index。一般存放在 ".git目录下" 下的index文件 (.git/index) 中，所以我们把暂存区有时也叫作索引 (index)。
- **版本库**：工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

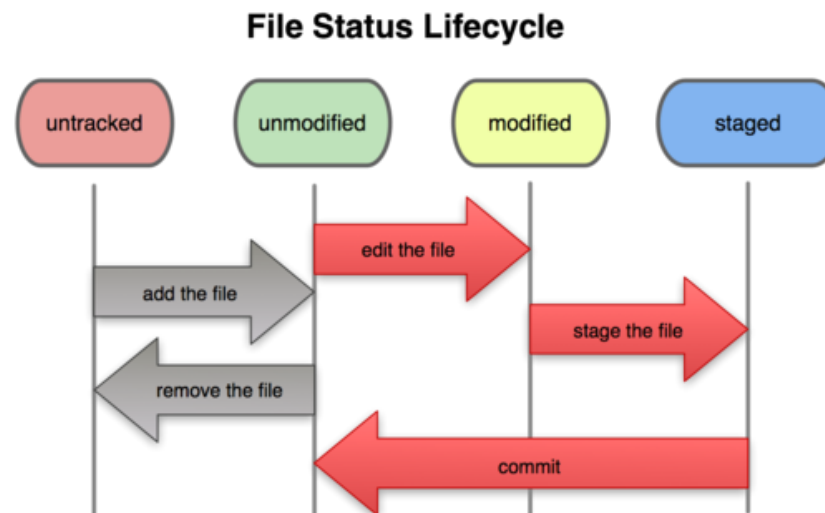


Git 工作区、暂存区和版本库

- 图中左侧为工作区，右侧为版本库。在版本库中标记为 "index" 的区域是暂存区（stage, index），标记为 "master" 的是 master 分支所代表的目录树。
- 图中我们可以看出此时 "HEAD" 实际是指向 master 分支的一个"游标"。所以图示的命令中出现 HEAD 的地方可以用 master 来替换。
- 图中的 objects 标识的区域为 Git 的对象库，实际位于 ".git/objects" 目录下，里面包含了创建的各种对象及内容。
- 当对工作区修改（或新增）的文件执行 "git add" 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID被记录在暂存区的文件索引中。
- 当执行提交操作（git commit）时，暂存区的目录树写到版本库（对象库）中，master 分支会做相应的更新。即 master 指向的目录树就是提交时暂存区的目录树。
- 当执行 "git reset HEAD" 命令时，暂存区的目录树会被重写，被 master 分支指向的目录树所替换，但是工作区不受影响。
- 当执行 "git rm --cached <file>" 命令时，会直接从暂存区删除文件，工作区则不做出改变。
- 当执行 "git checkout ." 或者 "git checkout -- <file>" 命令时，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。
- 当执行 "git checkout HEAD ." 或者 "git checkout HEAD <file>" 命令时，会用 HEAD 指向的 master 分支中的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

文件的状态周期

- 工作目录下面的所有文件都不外乎这两种状态：已跟踪或未跟踪。已跟踪的文件是指本来就被纳入版本控制管理的文件，在上次快照中有它们的记录，工作一段时间后，它们的状态可能是未更新，已修改或者已放入暂存区。而所有其他文件都属于未跟踪文件。它们既没有上次更新时的快照，也不在当前的暂存区域。初次克隆某个仓库时，工作目录中的所有文件都属于已跟踪文件，且状态为未修改。
- 在编辑过某些文件之后，Git 将这些文件标为已修改。我们逐步把这些修改过的文件放到暂存区域，直到最后一次性提交所有这些暂存起来的文件，如此重复。
- 注意：对于untracked、modified提交到staged都是“git add”命令



Git原理-管理修改

- Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。
- 把文件往Git版本库里添加的时候，是分两步执行的：
- 第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；
- 第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。
- 因为我们创建Git版本库时，Git自动为我们创建了一个master分支，所以，现在，git commit就是往master分支上提交更改。
- 一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的，暂存区就没有任何内容了。
- Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。
- 提交后，用git diff HEAD -- readme.txt命令可以查看工作区和版本库里面最新版本的差别

Git原理-撤销修改

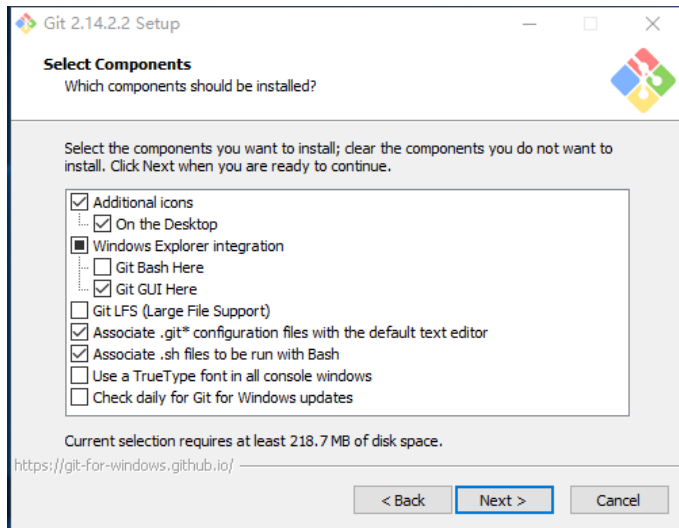
- 命令`git checkout -- readme.txt`意思就是，把`readme.txt`文件在工作区的修改全部撤销，这里有两种情况：
- 一种是`readme.txt`自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；
- 一种是`readme.txt`已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。
- 总之，就是让这个文件回到最近一次`git commit`或`git add`时的状态。

Git原理-删除文件

- 有两个选择，一是确实要从版本库中删除该文件，那就用命令`git rm`删掉，并且`git commit`。现在，文件就从版本库中被删除了。
- 另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本。
- 命令`git rm`用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

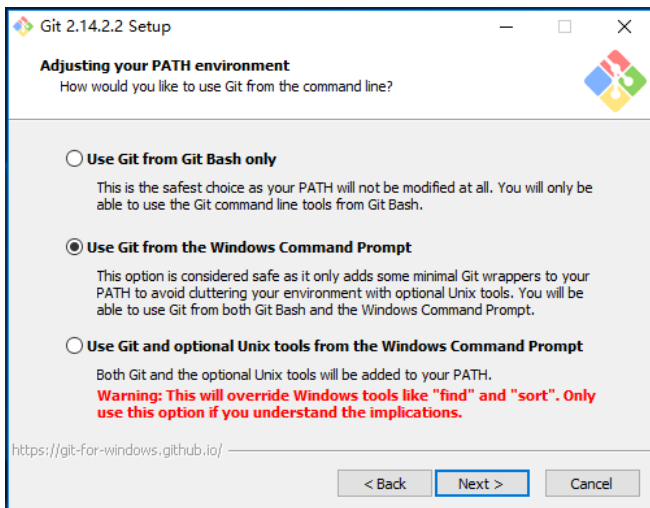
安装Git

- 1、下载git (<https://git-scm.com/download/win>)
- 2、安装git(Git-2.14.2.2-64-bit.exe)
- 3、next ,

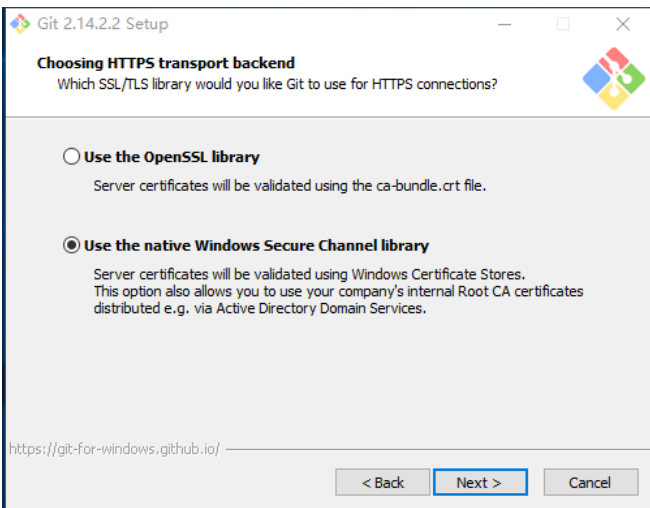


安装Git

- 4、next , 使用 Windos 命令行

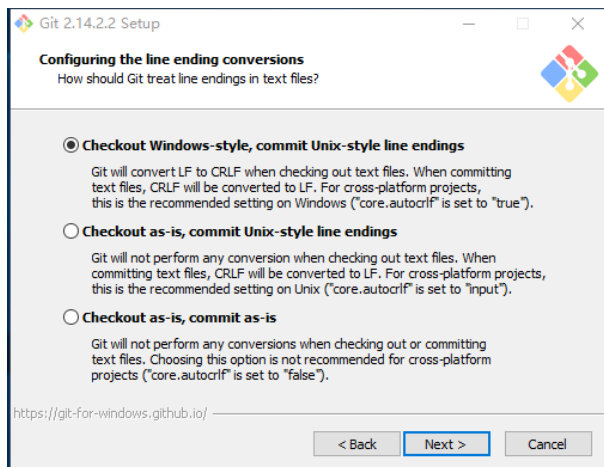


- 5、next , 使用 本地 Windos 加密通道

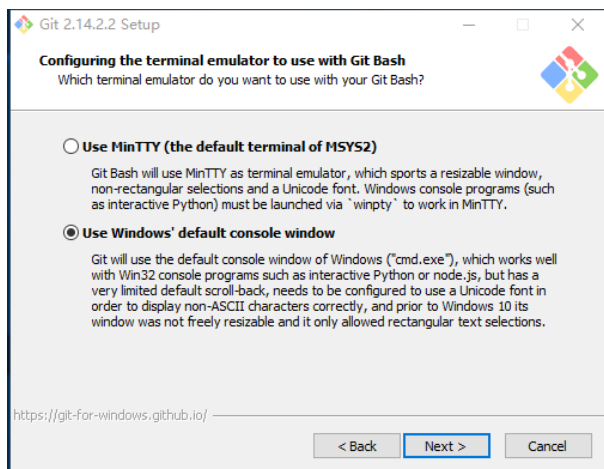


安装Git

- 6、next , checkout选择Windows风格

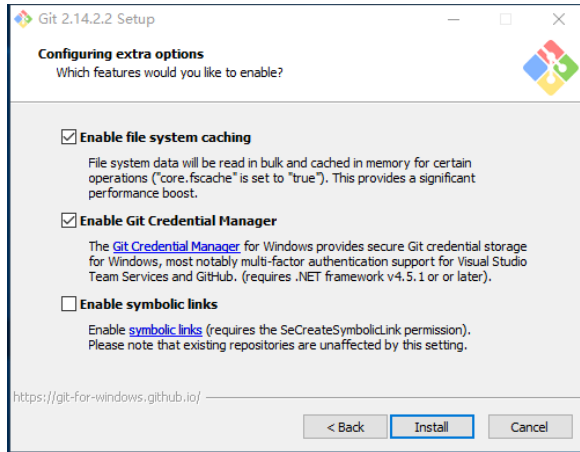


- 7、next , 使用Windows控制台



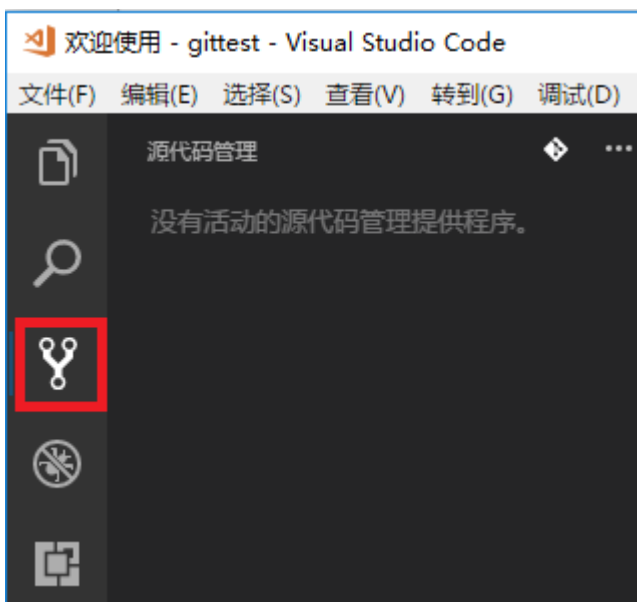
安装Git

- 8、next



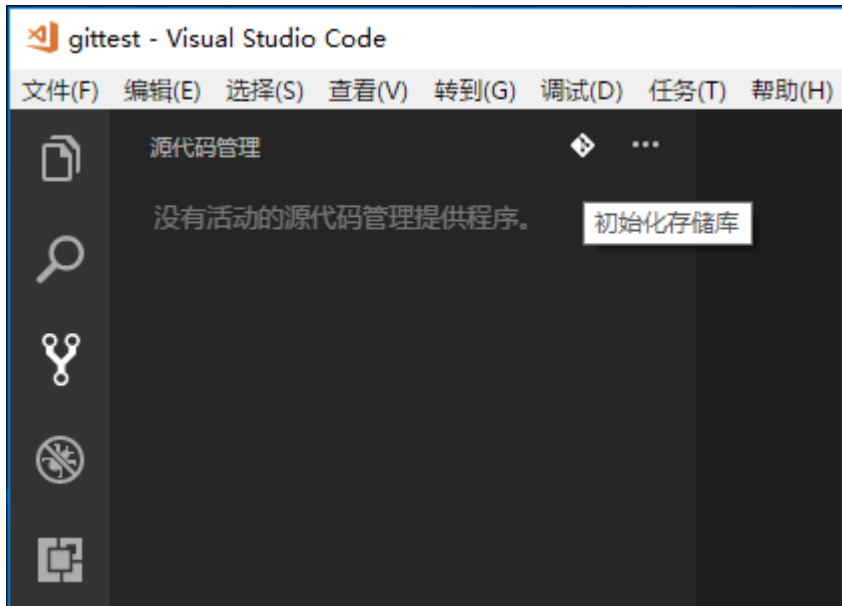
VS Code使用Git

- VS Code 集成了Git功能，并支持基本的git命令，这使得我们能够在开发过程方便的提交和获取代码。
- 1、创建一个名为gittest的文件夹。
- 2、用VS Code打开gittest的文件夹，单击左侧的git图标。



VS Code使用Git

- 3、单击“初始化存储库”的按钮，初始化存储库。



- 4、添加几个文件index.html、index1.html、html.css

VS Code使用Git

- 4、在GitHub上创建代码库 (<https://github.com/new>)

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner



junxian-chen ▾

Repository name

vscodegit



Great repository names are short and memorable. Need inspiration? How about **automatic-octo-barnacle**.

Description (optional)



Public

Anyone can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.



Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None ▾

Add a license: None ▾

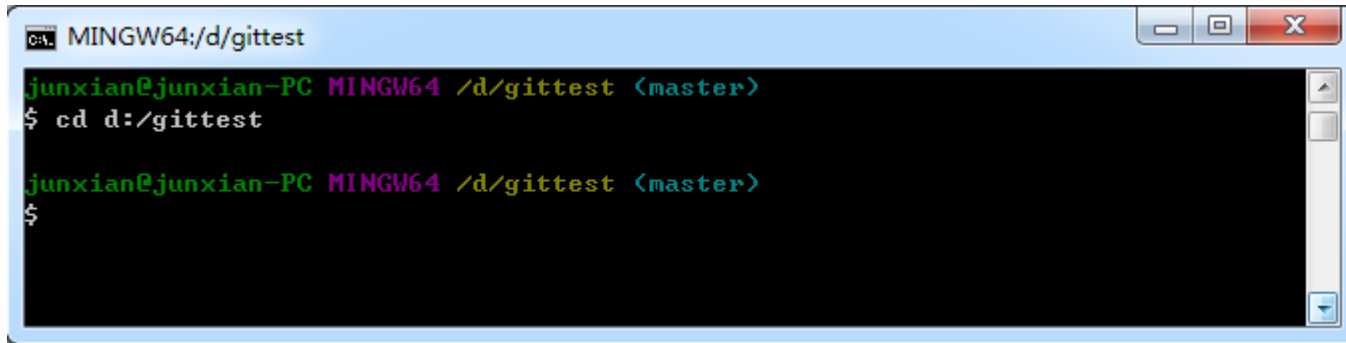


Create repository

记住代码库的网址

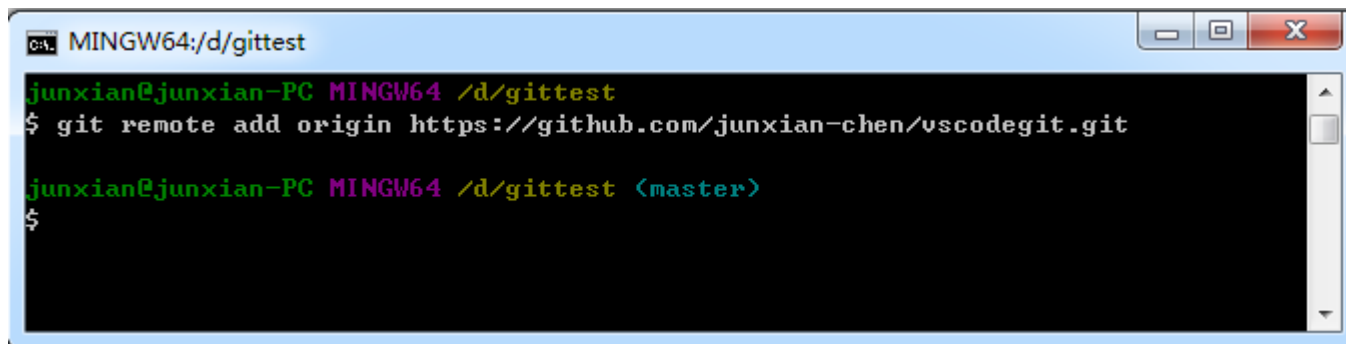
VS Code使用Git

- 5、启动git控制台，cd d:/gittest进入gittest的文件夹



```
MINGW64:/d/gittest
junxian@junxian-PC MINGW64 /d/gittest <master>
$ cd d:/gittest
junxian@junxian-PC MINGW64 /d/gittest <master>
$
```

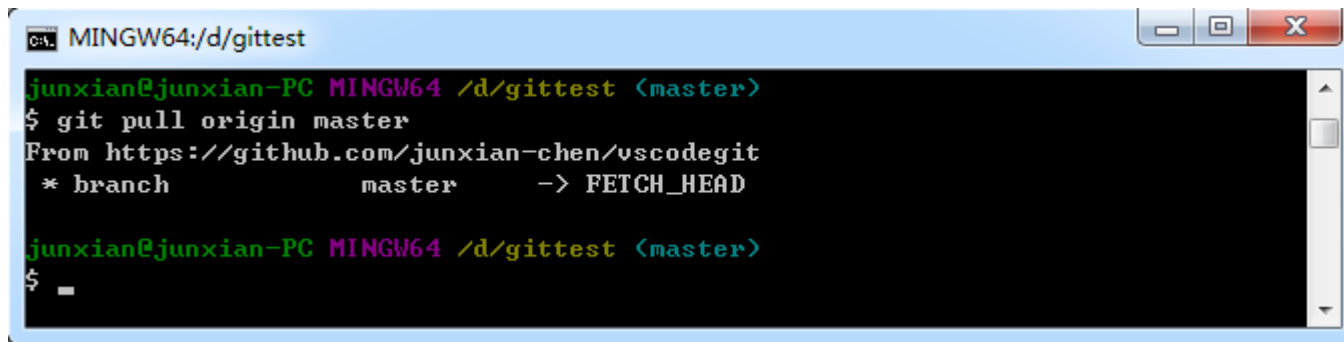
- 6、执行：git config --global user.email "you@example.com"
- git config --global user.name "Your Name"
- 7、执行：git remote add origin <https://github.com/junxian-chen/vscodegit.git>



```
MINGW64:/d/gittest
junxian@junxian-PC MINGW64 /d/gittest
$ git remote add origin https://github.com/junxian-chen/vscodegit.git
junxian@junxian-PC MINGW64 /d/gittest <master>
$
```

VS Code使用Git

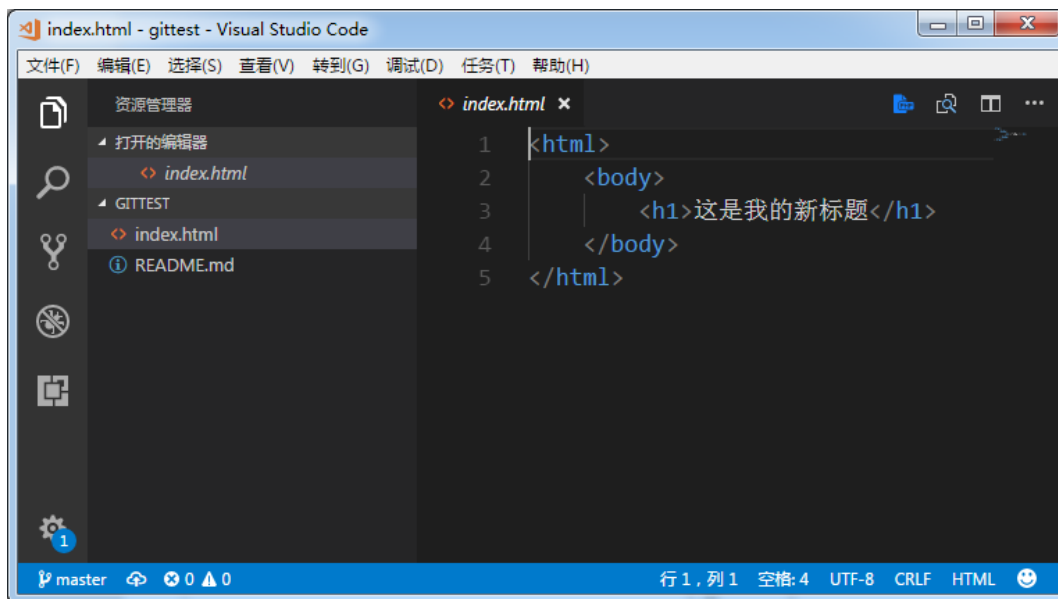
8、在git控制台输入命令：git pull origin master，把github同步到本地。



```
MINGW64:/d/gittest
junxian@junxian-PC MINGW64 /d/gittest <master>
$ git pull origin master
From https://github.com/junxian-chen/vscodegit
* branch          master      -> FETCH_HEAD

junxian@junxian-PC MINGW64 /d/gittest <master>
$
```

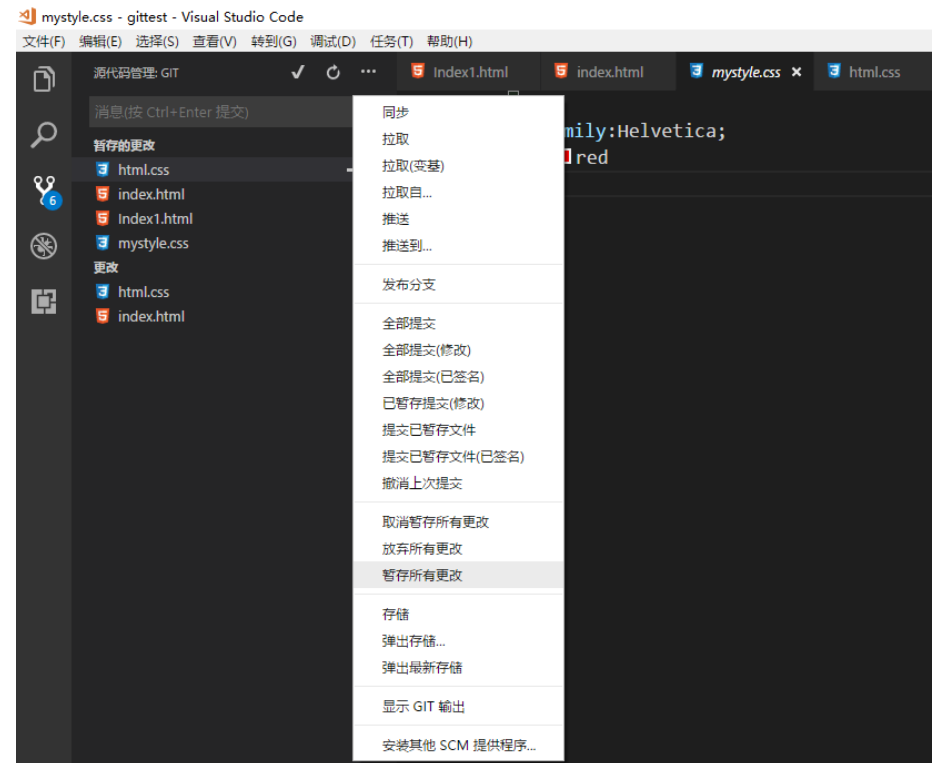
9、修改 README.md文件，并添加index.html文件



同步
拉取
拉取(变基)
拉取且...
推送
推送到...
发布分支
全部提交
全部提交(修改)
全部提交(已签名)
已暂存提交(修改)
提交已暂存文件
提交已暂存文件(已签名)
撤消上次提交
取消暂存所有更改
放弃所有更改
暂存所有更改
存储
弹出存储...
弹出最新存储
显示 GIT 输出
安装其他 SCM 提供程序...

VS Code使用Git

- 9、提交保存（点击+号，把所有文件提交到暂存区，然后打开菜单选择--提交已暂存文件，然后按提示随便在消息框里输入一个消息，再按ctrl+enter提交）
 - 提交保存的第一步是暂存文件。
 - 第二步是输入提交信息。
 - 第三步然后使用状态栏的提交按钮提交全部更改。



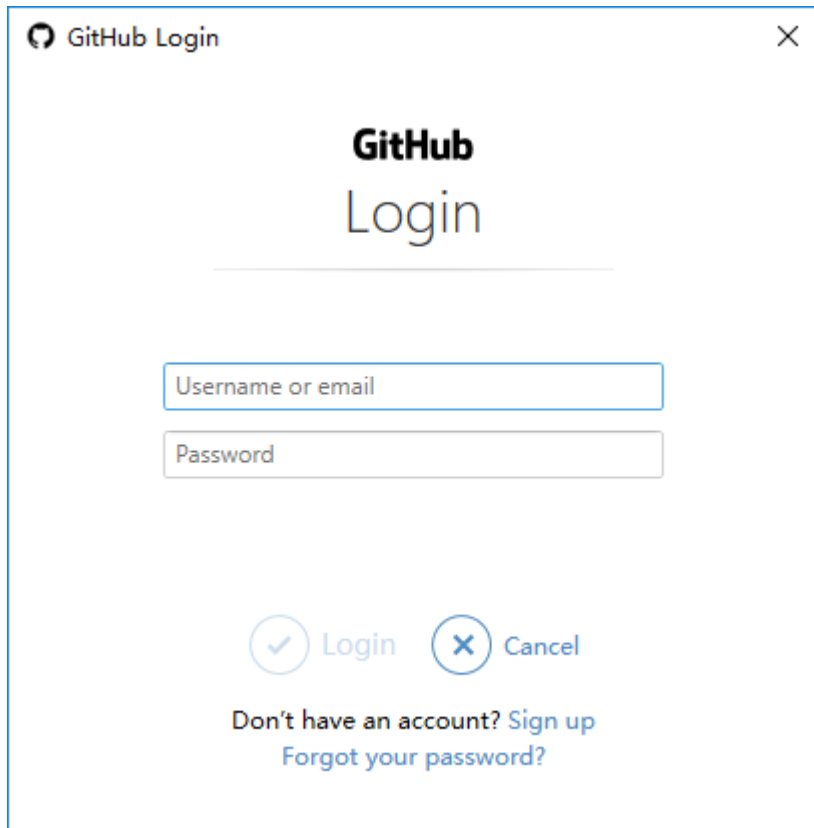
VS Code使用Git

- 10、接下来从下拉菜单中执行推送命令。



VS Code使用Git

- 9、输入用户名、密码后登录



The image shows a GitHub Login dialog box. At the top left is the GitHub logo and the text "GitHub Login". At the top right is a close button (X). In the center, the text "GitHub Login" is displayed. Below this, there are two input fields: "Username or email" and "Password". At the bottom, there are two buttons: "Login" (with a checkmark icon) and "Cancel" (with an X icon). Below the buttons, there is a link that says "Don't have an account? Sign up" and another link that says "Forgot your password?".

VS Code使用Git

- index.html已经更新到<https://github.com/junxian-chen>了

3 commits

1 branch

0 releases

1 contributor

Branch: master

New pull request

Find file

Clone or download

junxian-chen c2

Latest commit d118e0a an hour ago

README.md	c1	an hour ago
index.html	c2	an hour ago

README.md

vscodegit

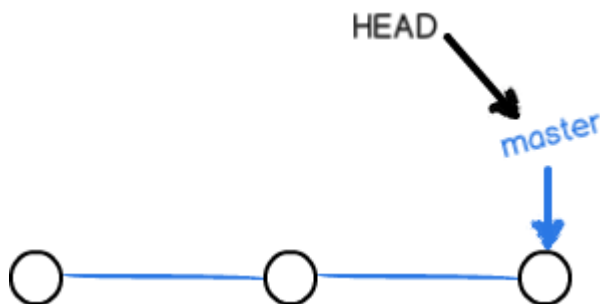
我的新文件

Git 分支

- 何谓分支
- 假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。
- 现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

Git 分支

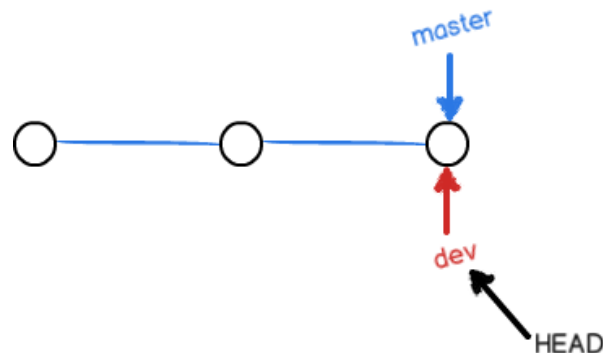
- 每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。
- 一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



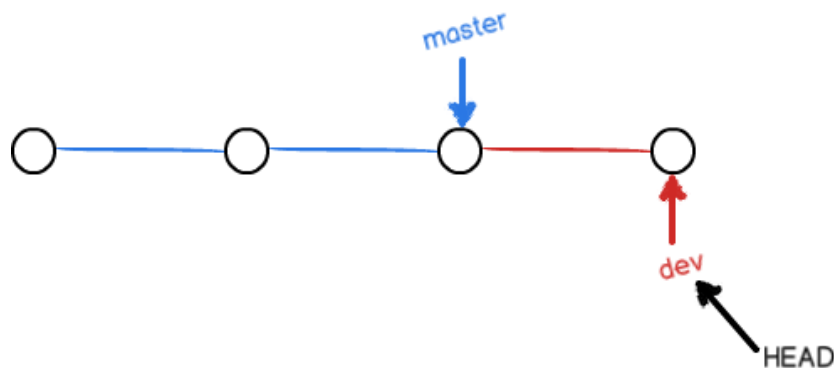
- 每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

Git 分支

- 当创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：

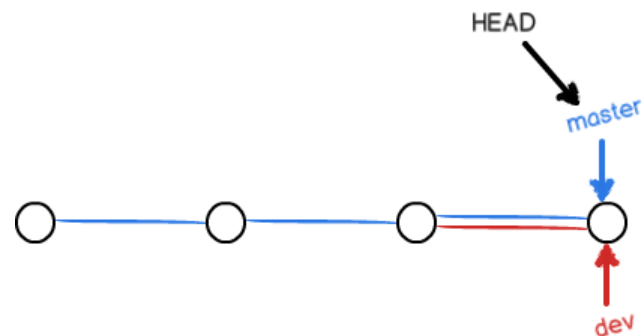


- Git创建一个分支很快，因为除了增加一个dev指针，改变HEAD的指向，工作区的文件都没有任何变化！
- 从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：

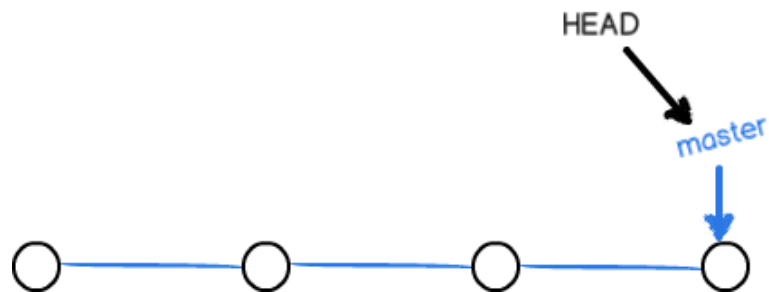


Git 分支

- 假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：

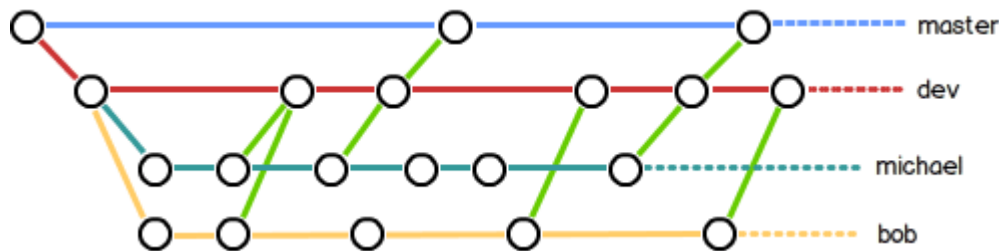


- 合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支：



分支策略

- 实际开发中，我们应该按照几个基本原则进行分支管理：
- 首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；
- 那在哪干活呢？干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；
- 你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。
- 所以，团队合作的分支看起来就像这样：



- 合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出曾经做过合并。

Feature分支

- 软件开发中，总有无穷无尽的新的功能要不断添加进来。
- 添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。
- 接到了一个新任务：开发代号为Vulcan的新功能
- `git checkout -b feature-vulcan`
- 修改。。。
- 切回dev，准备合并：
- `git checkout dev`
- 就在此时，接到上级命令，因经费不足，新功能必须取消！
- 虽然白干了，但是这个分支还是必须就地销毁：
- `git branch -d feature-vulcan`
- `error: The branch 'feature-vulcan' is not fully merged.`
- `If you are sure you want to delete it, run 'git branch -D feature-vulcan'.`

- 销毁失败。Git友情提醒，feature-vulcan分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用命令`git branch -D feature-vulcan`。

- 现在我们强行删除：
- `git branch -D feature-vulcan`

推送分支

- 推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：
- `git push origin master`
- 如果要推送其他分支，比如dev，就改成：
- `git push origin dev`
- 但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？
- master分支是主分支，因此要时刻与远程同步；
- dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；
- bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；
- feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

抓取分支

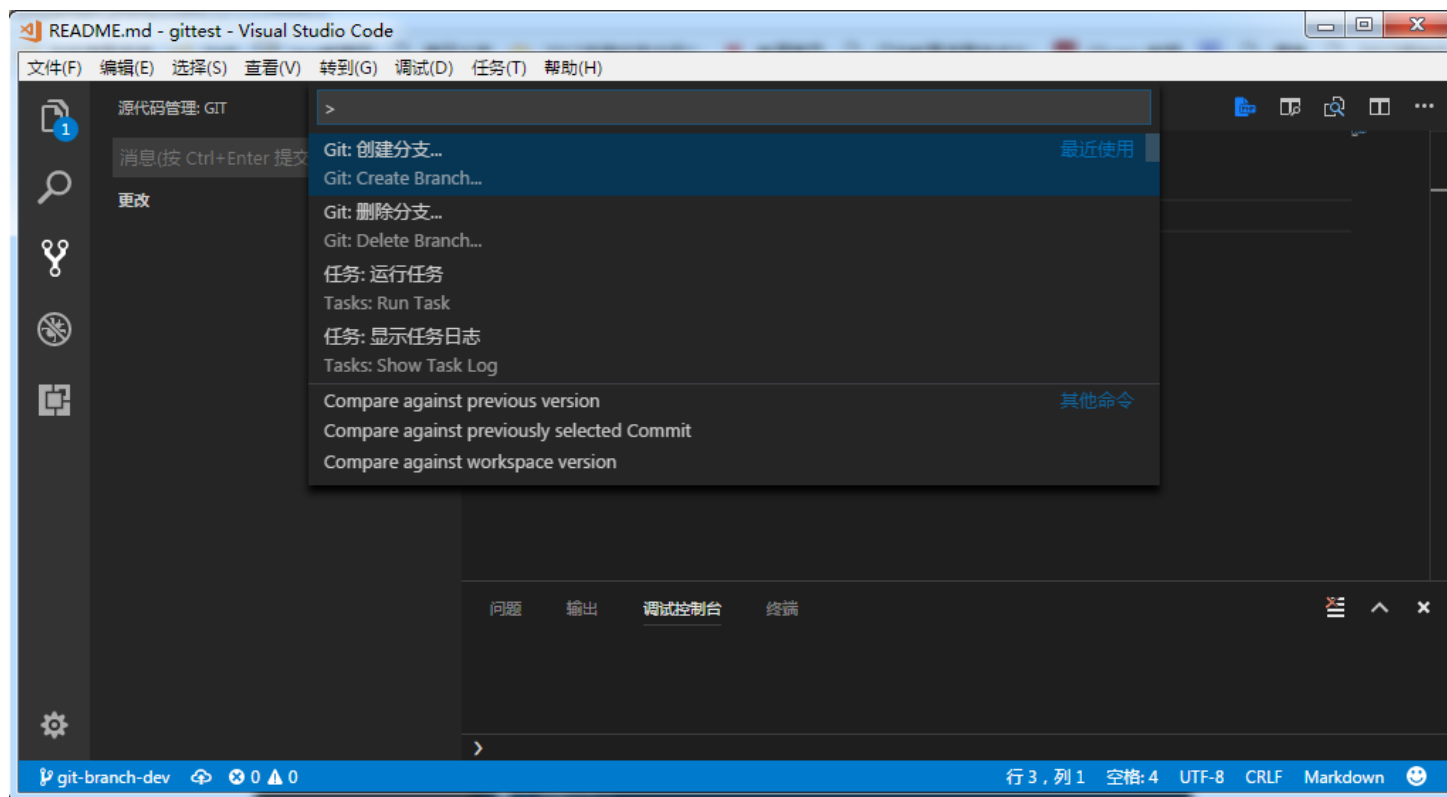
- 多人协作时，大家都会往master和dev分支上推送各自的修改。
- 当你的合作者从远程库clone时，默认情况下，你的合作者只能看到本地的master分支。
- 现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：
- `git checkout -b dev origin/dev`
- 现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程。
- 你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送。
- 推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用 `git pull` 把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送。
- `git pull`也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：
- `git branch --set-upstream dev origin/dev`
- 再pull：`git pull`
- 这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。

多人协作

- 多人协作的工作模式通常是这样：
- 首先，可以试图用`git push origin branch-name`推送自己的修改；
- 如果推送失败，则因为远程分支比你的本地更新，需要先用`git pull`试图合并；
- 如果合并有冲突，则解决冲突，并在本地提交；
- 没有冲突或者解决掉冲突后，再用`git push origin branch-name`推送就能成功！
- 如果`git pull`提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令`git branch --set-upstream branch-name origin/branch-name`。

VS Code使用Git分支

- 1、创建分支Dev (git branch dev) ,并切换到Dev (git checkout dev)
- ctrl+shift+p , 输入名字 : dev



VS Code使用Git分支

- 2、添加index.html文件，index1.html文件
- 3、提交后，推送到https://github.com/junxian-chen，选择Dev分支，如下图：

Branch: dev ▾

New pull request

Find file

Clone or download ▾

This branch is 3 commits behind master.

Pull request Compare

junxian-chen c12

Latest commit 699b959 2 hours ago

README.md	c1	5 hours ago
a.xt	c12	2 hours ago
index.html	c11	2 hours ago
index1.html	c10	2 hours ago

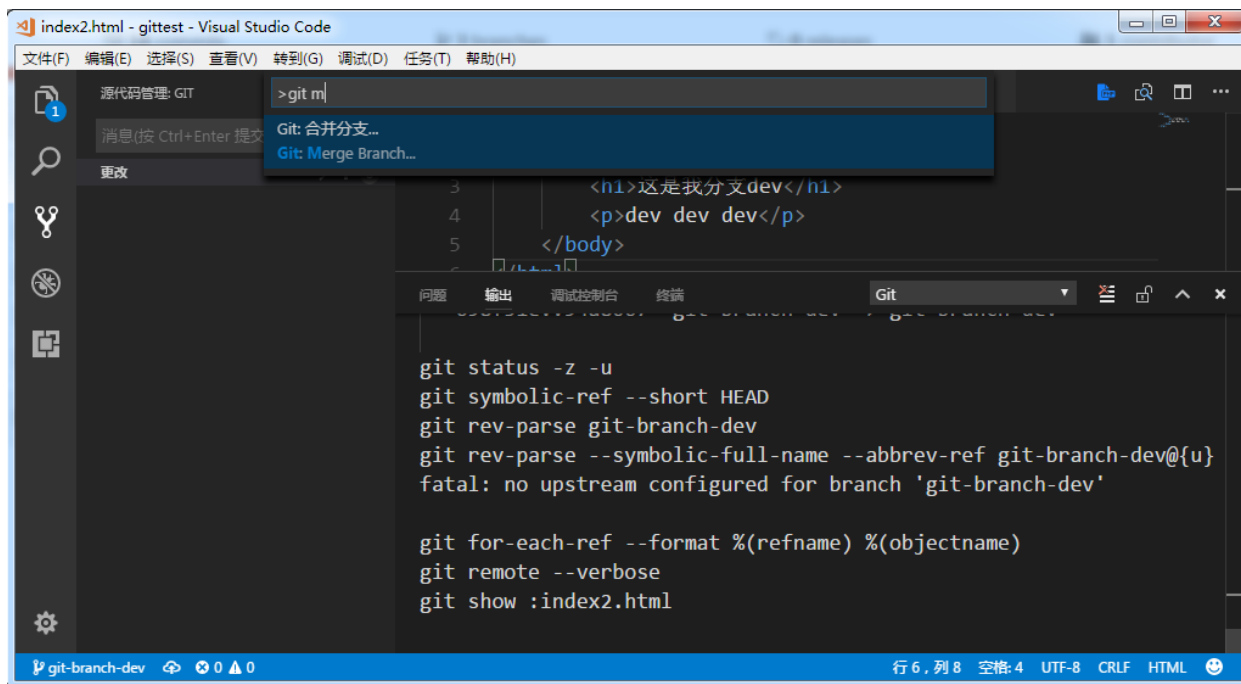
README.md

vscodegit

我的新文件

VS Code使用Git分支

- 4、合并分支，把dev分支合并到master上（`git merge dev`），并删除分支dev（`git branch -d dev`）



VS Code使用Git分支


- 5、查看分支master

Branch: master ▾





New pull request


Find file

Clone or download ▾

 junxian-chen c25

Latest commit 3277a4d 27 seconds ago

 README.md	merger	11 minutes ago
 a.xt	c12	2 hours ago
 index.html	c11	2 hours ago
 index1.html	c10	2 hours ago

 README.md

vscodegit

我的新文件 分支合并完成

VS Code使用Git同步

- 添加index2.html后，选择菜单同步，github上如下图：

Branch: masterNew pull requestFind fileClone or download

junxian-chen c29Latest commit 606f874 16 minutes ago

README.md	merger	41 minutes ago
a.xt	c12	3 hours ago
index.html	c11	3 hours ago
index1.html	c30	17 minutes ago
inex2.html	c29	16 minutes ago

README.md

vscodegit

我的新文件 分支合并完成

同步

拉取

拉取(变基)

拉取自...

推送

推送到...

发布分支

全部提交

全部提交(修改)

全部提交(已签名)

已暂存提交(修改)

提交已暂存文件

提交已暂存文件(已签名)

撤销上次提交

取消暂存所有更改

放弃所有更改

暂存所有更改

存储

弹出存储...

弹出最新存储

显示 GIT 输出

安装其他 SCM 提供程序...



Thanks