想把word所有图片导出来可以把docx改成zip之后解压即可

在写md的时候，要新建一个文件夹用于存放md文档中的图片，之后在md文档里面通过相对路径去引用这个图片，如<img src=".\Images\test.jpg">

<root> (tree)

|

+- foo (tree)

| |

| + bar.txt (blob, contents = "hello world")

|

+- baz.txt (blob, contents = "git is wonderful")

在Git的术语里，文件被称作**Blob对象**（数据对象），也就是一组数据。目录则被称之为“**tree**（树）”，它将名字与 Blob 对象或树对象进行映射（使得目录中可以包含其他目录）。

图示

描述已自动生成

在 Git 中，历史记录是一个由快照组成的**有向无环图**（DAG）。其中 o 表示一次提交（快照，就是上面那一整棵文件夹树），箭头指向了当前提交的父辈。以上图为例，1是原始版本，然后做了一些工作令其变为2，此时我想在2的基础上加一些feature，但别人又告诉我2有一些bug，那么可以不用在同一个文件上完成add feature与bugfix的工作（暂且称为fbfix，下面会说），可以分别完成之后合并起来成为5。（git会尝试自动合并，如果合并异常会报错让使用者去选择如何去合并）

以伪代码形式表示上述数据模型，如下：

// 文件就是一组数据

type blob = array<byte>

// 一个包含文件和目录的目录

type tree = map<string, tree | blob>

// 每个提交都包含一个父辈，元数据（比如说作者和备注）和顶层树

type commit = struct {

parent: array<commit>

author: string

message: string

snapshot: tree

}

Git 中的**对象可以是 blob、tree或commit：**

type object = blob | tree | commit

Git 在储存数据时，所有的对象都会基于它们的 **SHA-1哈希** 进行寻址。

objects = map<string, object>

string是对象内容的hash值，object是对象的指针

def store(object):

id = sha1(object)

objects[id] = object

def load(id):

return objects[id]

Blobs、tree和commit都一样，它们都是对象。当它们引用其他对象时，它们并没有真正的在硬盘上保存这些对象，而是仅仅保存了它们的哈希值作为引用。（可以通过 git cat-file -p hash\_num 来进行可视化）

SHA-1这个是哈希函数，转换后id有40char那么长，那么可以通过引用（也是一个映射）去解决这个问题：references = map<string, string>，其中一个是id，另一个是可读性文字比如说上面的bugfix引用是指向提交的指针。与对象不同的是，它是可变的（引用可以被更新，指向新的提交，比如说在5的基础上又增加了6，那么6的引用就是fbfix）。例如，master 引用通常会指向主分支的最新一次提交，只要输入git init，默认就会有一个master分支，它通常表示代码中的主开发分支，所以它通常代表项目的最新版本。

Git 仓库的定义：对象 和 引用。

在硬盘上，Git 仅存储对象和引用：因为其数据模型仅包含这些东西。所有的 git 命令都对应着对提交树的操作，例如增加对象，增加或删除引用。

以上就是git的数据结构。以下是git的命令。（不用输入<>）

**Git command-line interface**

**Basics**

* git help <command>: 获取 git 命令的帮助信息
* git init: 创建一个新的 git 仓库，其数据会存放在一个名为 .git 的目录下
* git status: 显示当前的仓库状态
* git add <filename>: 添加文件到暂存区
* git commit(输入后，显示出来的一些值的意思可见课程第30分钟)
* : 创建一个新的提交
  + 如何编写 [良好的提交信息](https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html)!
  + 为何要 [编写良好的提交信息](https://chris.beams.io/posts/git-commit/)
* git log: 显示历史日志
* git log -x <filename>:查看关于filename最新的x次提交
* git log --all --graph --decorate: 可视化历史记录（有向无环图，在电脑上是竖着的图，不是像上面那张图那样是横着的），在后面再加个--oneline就会只出现一行（内容会简短很多）
* git diff <filename>: 显示与暂存区文件的差异（默认为head与filename的差异）直接用git diff也行
* git diff <revision> <filename>: 显示如今的filename相对于revision这个版本时的差异
* git diff <revision1> <revision2> <filename>:显示某个文件两个版本之间的差异（与上面的区别一下，这个可以指定版本，上面是现版本）
* git checkout <revision>: 更新 HEAD 和目前的分支，如果直接输入了git checkout <filename>的话，它会丢弃未提交的修改并更新head指向的快照的状态。

**Branching and merging**

* git branch: 显示分支
* git branch <name>: 创建分支
* git checkout -b <name>: 创建分支并切换到该分支
  + 相当于 git branch <name>; git checkout <name>
* git merge <revision>: 将revision自动合并到当前分支，想要回到merge之前的状态输入git merge --abort
* git mergetool: 使用工具来处理合并冲突(后续输入vimdiff)
* git rebase: 将一系列补丁变基（rebase）为新的基线

**Remotes**

* git remote: 列出远端
* git remote add <name> <url>: 添加一个远端（不仅可以github还可以本地另一个文件夹，如果只使用一个仓库的话name一般是origin）
* git push <remote> <local branch>:<remote branch>: 将对象传送至远端(remote这里填的是远程仓库的name)并更新远端引用，它会在远程仓库创建一个新分支或者更新上面的一个分支，并将其改为本地这个分支的内容，不想这么麻烦就见1:07:00左右的视频
* git branch --set-upstream-to=<remote name>/<remote branch>: 创建本地(head所在位置)和远端分支的关联关系(比如说origin/master)
* git fetch (<remote>): 从远端获取对象/索引(如果只有一个仓库origin就不用输入括号内的内容)
* git pull: 相当于 git fetch; git merge
* git clone <url> <directory>: 从远端下载仓库到directory这里

**Undo**

* git commit --amend: 编辑提交的内容或信息
* git reset HEAD <file>: 恢复暂存的文件
* git checkout -- <file>: 丢弃修改

**Advanced Git**

* git config: Git 是一个高度可定制的 工具
* git clone --depth=1: 浅克隆（shallow clone），不包括完整的版本历史信息，即只会有最新版本的快照
* git add -p: 交互式暂存，见1:16:03
* git rebase -i: 交互式变基
* git blame: 查看最后修改某行的人
* git stash: 暂时移除工作目录下的修改内容
* git bisect: 通过二分查找搜索历史记录
* .gitignore: 指定不追踪的文件(首先得vim ~/.gitignore,见课程1:21:00)

图形用户界面, 文本

描述已自动生成

所有仓库数据将存储在这两个目录的下面（上面有说:在硬盘上，Git 仅存储对象和引用）

文本

描述已自动生成

（第一次commit输入了helloworld，第二次输入了another line）

Master可以理解为指向这个提交的指针，随着提交的添加，这个指针将被改变来指向后面的提交。Head基本上用于指向你当前正在查看的提交。

当输入git checkout 42fb…（不用输入完），那么此时就会走到第一个commit（commit是一个结构体，上面有提到），见下图，可见head的位置改变了。因为回到了第一个commit，所以cat后只见到helloworld。（回想weblab做ws的时候，会输入git checkout w3-step1等等，这就是用来回到某个提交的）此时还想回到master分支，可以直接git checkout master

文本

描述已自动生成

文本

描述已自动生成

Git checkout换成master分支后，输入git merge cat，再输入git merge dog(中途处理冲突代码git mergetool)，之后git add加上处理过后的新内容，再git merge –continue即可合并分支为如下（合并后是切换到合并的那个分支）：

文本

描述已自动生成

文本

描述已自动生成

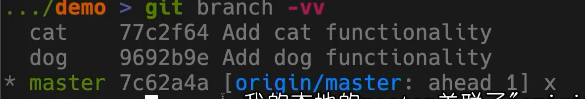
效果：在远程仓库(origin)上创建一个名为master（后者）的分支，他将于本地仓库上的master（前者）分支相同，效果如下所示

文本

描述已自动生成  
如上图字幕所示，此时课程上模仿了两台机器操作github的样子



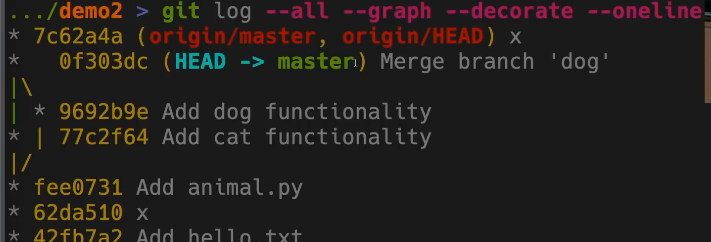
在machine2上git clone了远端仓库到demo2这个目录(注意是到origin/master这一块，注意没有7c62a4a这一块)，此时machine1 用了git push origin master:master，就是把7c62a4a这一块更新到远端仓库中去(这个方法比较麻烦，可以使用上面说的git branch --set-upstream-to=origin/master，此时



可见有三个分支，并且本地的master分支与远端仓库origin的master的分支关联起来了，此时直接输入git push即可），此时回到machine2，

文本

描述已自动生成它还不知道远端仓库已经更新了，要与这个远端仓库起联系，要先git fetch，此时就会有



可见远端仓库已经更新，但自身（head）还在原处，所以输入git merge即可合并了，可以将这两步合成一个命令叫做**git pull**

文本

描述已自动生成

此时machine1与2都更新到最新状态了

* [Pro Git](https://git-scm.com/book/en/v2) ，**强烈推荐**！学习前五章的内容可以教会您流畅使用 Git 的绝大多数技巧，因为您已经理解了 Git 的数据模型。后面的章节提供了很多有趣的高级主题。（[Pro Git 中文版](https://git-scm.com/book/zh/v2)）；
* [Oh Shit, Git!?!](https://ohshitgit.com/) ，简短的介绍了如何从 Git 错误中恢复；
* [Git for Computer Scientists](https://eagain.net/articles/git-for-computer-scientists/) ，简短的介绍了 Git 的数据模型，与本文相比包含较少量的伪代码以及大量的精美图片；
* [Git from the Bottom Up](https://jwiegley.github.io/git-from-the-bottom-up/)详细的介绍了 Git 的实现细节，而不仅仅局限于数据模型。好奇的同学可以看看；
* [How to explain git in simple words](https://smusamashah.github.io/blog/2017/10/14/explain-git-in-simple-words)；
* [Learn Git Branching](https://learngitbranching.js.org/) 通过基于浏览器的游戏来学习 Git ；

图形用户界面, 文本, 应用程序

描述已自动生成

git filter-branch --force --index-filter 'git rm --cached --ignore-unmatch ./my\_password' --prune-empty --tag-name-filter cat -- --all

上面这个命令的./my\_password是可以修改成别的，这个命令的作用是移除掉一些敏感文件（比如说这个文件夹里有个密码.txt，还不小心git add到缓存区，就可以用这个命令去移除），还可以将提交历史删除掉

git stash用于暂时保存当前工作目录的改动并将其清理干净，以便你可以在干净的工作目录上进行其他操作（例如切换分支、拉取更新等）。git stash 的主要作用是让你可以在不提交当前改动的情况下，暂时存储这些改动，并在稍后恢复它们。例子如下，详细可以见课程lab示例，运用stash和stash pop，可以自由选择 *stash存储的改动* 即将提交到的分支（lab中，将原本应该在main分支下提交的记录移动到了dog分支下进行提交）

图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

Git 也提供了一个名为 ~/.gitconfig 配置文件 (或 dotfile)。可以在 ~/.gitconfig 中配置自己的git；可以通过执行 git config –global core.excludesfile ~/.gitignore\_global 在 ~/.gitignore\_global 中创建全局忽略规则。配置您的全局 gitignore 文件来自动忽略系统或编辑器的临时文件，例如 .DS\_Store；