Git是什么？

Git最流行的分布式版本控制系统

趣味故事：

1991年-2002年

老李：使用文件比对方式合并来自世界各地的开发者提交的代码

老李想：坚决反对CVS和SVN这种集中是版本控制系统，效率低下还要联网，好用的还他妈收钱，果断不用。

2002年

世界各地的兄弟：老李啊，现在代码量这么庞大了，还人工比对合并，是不是太菜啦？

老李：也是，忙不过来啦，我还是找个收费的商业版吧，这个分布式管理系统BitKeeper怎么样？ok，就它了。

BitKeeper厂商BitMover：啊呀，是老李你要用啊，这样吧，给你免费使用。

2005年

社区一牛人Andrew：BitKeeper还收费，看我不破了它。…OK 搞定。

BitKeeper厂商BitMover：Shit，这群王八蛋，给他们免费使用了，还破我的协议。

BitKeeper厂商BitMover：老李，你给老子出来，那个软件我要收回了，你们别想用了，居然还找人破解老子的协议。

老李：Fuck，不用就不用，老子自己写一个，怕个球。

两星期后…

老李：要不是以前没空，早就自己写了，这么简单的东西。

一个月后

老李：终于不用手动管理代码了。

世界各地的兄弟：老大就是腻害，崇拜的仰望…

2008年

世界各地的开发者：握草，快来看，GitHub使用Git来管理代码啊，好东西，我要把代码提交上去管控，再也不用什么SVN之类的啦。

集中式与分布式对比



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。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。

集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，这还不得把人给憋死啊。



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。

和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。

Git安装

Windows安装

1. <https://git-scm.com/download/win> 下载安装
2. <https://desktop.github.com/> 下载GitHub的桌面版

Mac OS X安装

1. 使用HomeBrew安装Git
2. 使用Xcode安装，Xcode集成了Git，不过默认没有安装，你需要运行Xcode，选择菜单“Xcode”->“Preferences”，在弹出窗口中找到“Downloads”，选择“Command Line Tools”，点“Install”就可以完成安装了。

全局用户配置

打开Git命令行控制台

Windows

程序中打开：

Mac

控制台直接输入命令即可

用户配置

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

注意git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

Git入门操作

创建版本库

git init

新增文件并提交

README.md

git add .

git commit –m “新增README.md文件”

提交修改

修改前先查看状态与修改内容比对

git status

git diff README.md

git add README.md

git status

git commit –m “修改README.md增加一行内容”

查看提交log

git log [--pretty=oneline]

版本回退

回退到上一个版本：

git reset --hard HEAD^

git log

发现最新的版本找不到了，怎么办？

在上面输出未清除的情况下，直接reset版本号

git reset --hard 36780019

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD从指向append GPL：



改为指向add distributed：



但是，但是，我清空了输出，找不到编号了怎么办？

在Git中，总是有后悔药可以吃的。当你用$ git reset --hard HEAD^回退到add distributed版本时，再想恢复到append GPL，就必须找到append GPL的commit id。Git提供了一个命令git reflog用来记录你的每一次命令：

git reflog

工作区与暂存区

工作区：当前的工作目录

暂存区：使用git add 添加的内容，所存放的区域

版本库：.git文件夹，其中包含：暂存区 + 分支

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。



git add .



git commit



测试下不执行git add 直接git commit是什么情况？

管理的是修改

先修改内容

git add

git status

再次修改

git commit

git status

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区别：

git diff HEAD -- README.md

git add .

git commit

在未提交前，放弃修改

git checkout -- README.md

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 checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令

撤销已经add的修改

Git同样告诉我们，用命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区：

git reset HEAD README.md

git status

git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

然后再用

git checkout -- README.md

git status

分支

分支就像一个进程中的两个线程，都是为了到达最终的目的而工作的。在需要的时间点，由主线程开启一个用于处理子任务的线程，在子任务未完成前，两个线程互不干扰，在子任务完成后，不再需要时，将处理后的结果传递给主线程（合并），同时主线程可以在这段时间里做任何自己想做的事。

分支实际使用场景？

我手上的项目：翼销售

福建

1

2

3

4

宁夏

1

2

3

海南

1

2

3

4

5

基本上都需要服务端接口的配合，需求设计文档完成后，单独客户端部分可以先行开发，但是服务端接口会延迟，这时，我应该去处理其他任务能处理的部分。

假设使用SVN管理：

我只能在一个工程中不断的开发，每个任务可能都没有完成100%，或者部分任务完成，需要合并，需要发布。

这时怎么办？？？

选择性提交？？？要是同一个文件中涉及到两个任务呢？要是忘记的哪些文件对应哪个任务了呢？？？

git分支，大显身手的机会来了

你可以建任意多个的版本分支，对，任意多个。

主分支：master分支

自建分支：开发任务分支、feature分支、developer分支…

上述的问题我们这样来处理：

福建项目：我按日期 + 任务编号的方式建立四个分支

20170808-task1

20170808-task2

20170808-task3

20170808-task4

20170808-task5

宁夏海南亦如此…

现在：

假设任务2文档齐全，我先开发任务2，

git checkout 20170808-task1

刷新IDE项目目录，这样就在这里开发任务2。如果开发到一半不能进行下去 或者其他任务优先，这时先将分支的代码保存到该分支上。

git add .

git commit -m “提交日志”

假设要开发任务1，则切换到任务1

git checkout 20170808-task1

同样刷新IDE的项目目录，开始开发。

任务2的代码在任务1的分支上可以看到吗？？？

分支图解

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。



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：



创建分支

当我们创建新的分支，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，改改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：



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：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

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：





实战

新建分支

创建dev分支

git branch dev

切换分支

git checkout dev

简化命令

git checkout –b dev

查看分支情况

git branch

在dev分支下，修改README.md文件，然后提交

然后切换到主分支，查看README.md情况

git checkout master

这时的HEAD指向的变化情况







分支合并

dev开发完成，并且代码提交后，将dev分支合并到master上

切换到master，然后

git merge dev

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

合并完成后，就可以放心地删除dev分支了：

git branch –d dev

查看下分支情况

git branch

因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

分支管理相关命令

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

创建+切换分支：git checkout -b <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

解决冲突

新建两个分支feature1

feature1中添加一行，然后提交

切换到 master主分支，添加一行内容，然后提交



这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突

合并feature1分支，这时产生冲突

git merge feature1

产生冲突会中断分支的合并，只有解决了冲突后，然后提交，会继续完成分支合并。

解决冲突，然后提交

git add README.md

git commit –m “解决冲突”



用带参数的git log也可以看到分支的合并情况

git log –graph –pretty=oneline –abbrev-commit

现在可以删除feature1分支

git branch –d feature1

分支使用策略

合并分支的高级用法

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

git merge –no-ff –m “合并说明” dev

实战 新建dev分支

然后查看log

git log --graph --pretty=oneline --abbrev-commit

不使用Fast forward模式，merge后就像这样：



使用FF模式



如何正确使用分支？？？

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

其次基于master分支创建一个开发dev分支，用于管理所有的开发任务，

然后再dev分支上建立不同任务/不同功能的feature分支。

所有代码都合并再dev上，然后按照版本规划，使用master分支进行版本发布。



远程仓库与多人协作

测试仓库

<https://git.oschina.net/ymqq/gittest.git>

clone

git clone <https://git.oschina.net/ymqq/gittest.git>

查看分支情况

关联远程dev分支

方式一：git checkout –b dev origin/dev

方式二：

本地创建dev分支

git checkout –b dev

git pull origin dev // 关联

方式三：

本地创建dev分支，直接push

git checkout –b dev

git push origin dev // 这样在远程仓库中若没有分支，则直接创建分支，若有的话就push

同步远程提交

git pull

如果本地分支没有关联远程分支，同步失败，需要设置关联

方式一：

git pull origin dev

方式二：

git branch *--set-upstream dev origin/dev*

因此，多人协作的工作模式通常是这样：

1. 首先，可以试图用git push origin branch-name推送自己的修改；
2. 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；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 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。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

标签管理

标签基本上由于发布时的别名，方便后期的查看与定位。

切换到master分支

git tag v1.0

查看标签

git tag

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

方法是找到历史提交的commit id，然后打上就可以了

git log --pretty=oneline --abbrev-commit

git tag v0.9 6224937

再次查看

git tag

注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签信息：

$ git **show** v0.9

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

$ git tag -a v0.1 -m "version 0.1 released" 3628164

用命令git show <tagname>可以看到说明文字：

删除标签

$ git tag -d v0.1

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

如果要推送某个标签到远程，使用命令git push origin <tagname>：

推送标签

$ git push origin v1.0

一次性推送全部尚未推送到远程的本地标签：

$ git push origin --tags

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

$ git tag -d v0.9

然后，从远程删除。删除命令也是push，但是格式如下：

$ git push origin :refs/tags/v0.9

忽略文件配置，.gitignore文件

[https://github.com/github/gitignore](https://github.com/github/gitignore" \t "_blank)