Git入门教程

# 课程大纲

1. Git简介
2. Git安装
3. Git的基本使用
4. 远程仓库
5. 分支管理
6. 标签管理
7. 案例：将Java工程加入到版本库

# 课堂笔记

# Git简介

## Git是什么？

Git是目前世界上最先进的分布式版本控制系统（没有之一）。

Git有什么特点？简单来说就是：高端大气上档次！

## Git的历史

同生活中的许多伟大事件一样，Git 诞生于一个极富纷争大举创新的年代。Linux 内核开源项目有着为数众广的参与者。绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上（1991－2002年间）。到 2002 年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，于是整个项目组开始启用分布式版本控制系统 BitKeeper 来管理和维护代码。

到 2005 年的时候，开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，他们收回了免费使用 BitKeeper 的权力。这就迫使 Linux 开源社区（特别是 Linux的缔造者 Linus Torvalds ）不得不吸取教训，只有开发一套属于自己的版本控制系统才不至于重蹈覆辙。他们对新的系统订了若干目标：

• 速度

• 简单的设计

• 对非线性开发模式的强力支持（允许上千个并行开发的分支）

• 完全分布式

• 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

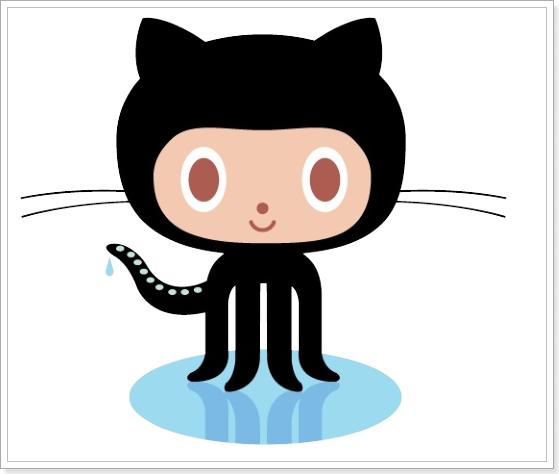
历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。



## Git和GitHub和GitLab

什么是GitHub？

确切的说 GitHub 是一家公司，位于旧金山，由 Chris Wanstrath, PJ Hyett 与 Tom Preston-Werner 三位开发者在2008年4月创办。这是它的 Logo：



2008年4月10日，GitHub正式成立，地址：How people build software · GitHub ，主要提供基于git的版本托管服务。一经上线，它的发展速度惊为天人，截止目前，GitHub 已经发展成全球最大的开源社区。

所以 Git 只是 GitHub 上用来管理项目的一个工具而已，GitHub 的功能可远不止于此！

什么是GitLab？

**GitLab**是由GitLabInc.开发，使用[MIT许可证](https://baike.baidu.com/item/MIT%E8%AE%B8%E5%8F%AF%E8%AF%81)的基于[网络](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C)的[Git](https://baike.baidu.com/item/Git)[仓库](https://baike.baidu.com/item/%E4%BB%93%E5%BA%93)管理工具，且具有[wiki](https://baike.baidu.com/item/wiki)和issue跟踪功能。使用[Git](https://baike.baidu.com/item/Git)作为代码管理工具，并在此基础上搭建起来的web服务。

GitLab由乌克兰程序员DmitriyZaporozhets和ValerySizov开发，它由[Ruby](https://baike.baidu.com/item/Ruby)写成。后来，一些部分用[Go语言](https://baike.baidu.com/item/Go%E8%AF%AD%E8%A8%80)重写。截止2018年5月，该公司约有290名团队成员，以及2000多名开源贡献者。GitLab被IBM，Sony，JülichResearchCenter，NASA，Alibaba，Invincea，O’ReillyMedia，Leibniz-Rechenzentrum(LRZ)，CERN，SpaceX等组织使用。

**GitHub 和  GitLab 都是基于 web 的 Git 仓库，使用起来二者差不多，它们都提供了分享开源项目的平台。**

**GitLab 让开发团队对他们的代码仓库拥有更多的控制，相比较 GitHub , 它有不少特色：**

**(1) 允许免费设置仓库权限；**

**(2) 允许用户选择分享一个 project 的部分代码；**

**(3) 允许用户设置 project 的获取权限，进一步提升安全性；**

**(4) 可以设置获取到团队整体的改进进度；**

**(5) 通过 innersourcing 让不在权限范围内的人访问不到该资源；**

**所以，从代码的私有性上来看，GitLab 是一个更好的选择。但是对于开源项目而言，GitHub 依然是代码托管的首选。**

## 集中式和分布式

### 集中式版本控制（Svn）

SVN是集中式版本控制系统，版本库是集中放在中央服务器的，而干活的时候，用的都是自己的电脑，所以首先要从中央服务器哪里得到最新的版本，然后干活，干完后，需要把自己做完的活推送到中央服务器。集中式版本控制系统是必须联网才能工作，如果在局域网还可以，带宽够大，速度够快，如果在互联网下，如果网速慢的话，就郁闷了。

下图就是标准的集中式版本控制工具管理方式：



集中管理方式在一定程度上看到其他开发人员在干什么，而管理员也可以很轻松掌握每个人的开发权限。

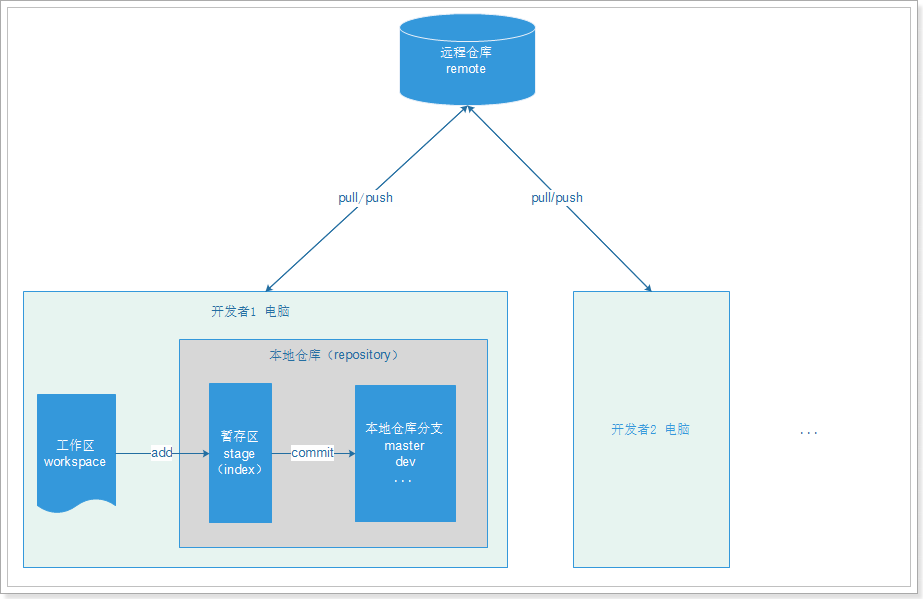
但是相较于其优点而言，集中式版本控制工具缺点很明显：

* 服务器单点故障
* 容错性差

## 分布式版本控制

Git是分布式版本控制系统，那么它可以没有中央服务器的，每个人的电脑就是一个完整的版本库，这样，工作的时候就不需要联网了，因为版本都是在自己的电脑上。既然每个人的电脑都有一个完整的版本库，那多个人如何协作呢？比如说自己在电脑上改了文件A，其他人也在电脑上改了文件A，这时，你们两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。

下图就是分布式版本控制工具管理方式：



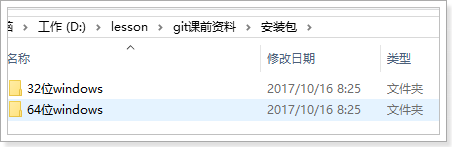
# Git安装

## 下载

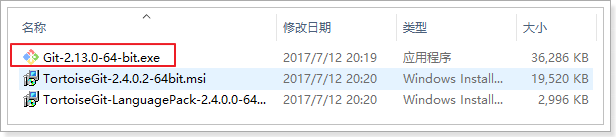
下载地址：<https://git-scm.com/download>



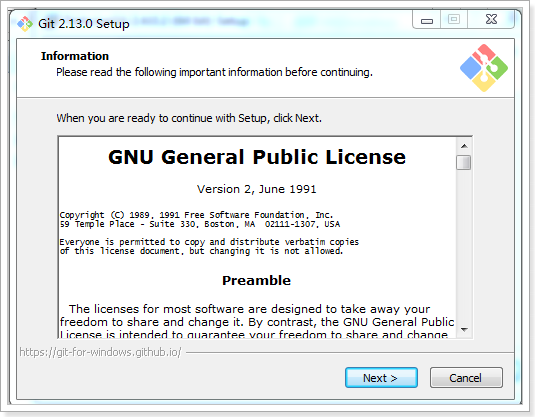
或者我们也可以使用课前资料提供的安装包：



## 安装git for windows



双击安装：

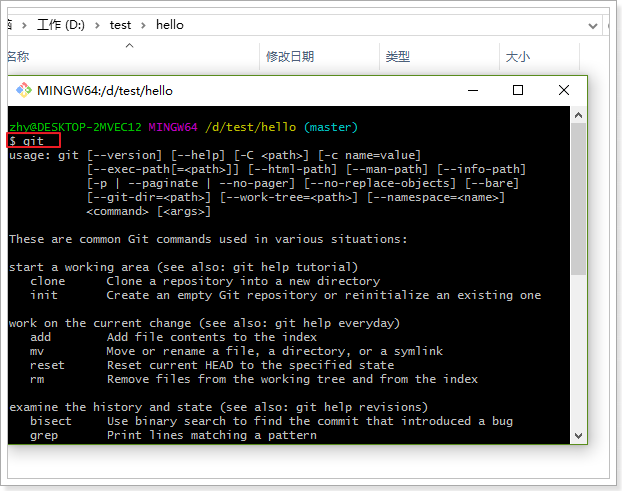


一路“Next”使用默认选项即可。

安装完成后，可以在任意文件夹点右键，看到如下菜单：



点击后，出现Git的控制台，在控制台输入git，可以看到相关的帮助信息：



# Git的基本使用

## 创建版本库

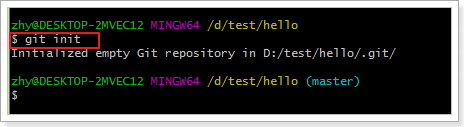
什么是版本库呢？版本库又名仓库，英文名repository，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。由于git是分布式版本管理工具，所以git在不需要联网的情况下也具有完整的版本管理能力。

创建一个版本库非常简单，

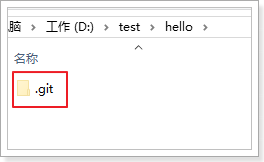
1）首先，选择一个合适的地方，创建一个空目录。我在本机的D:\\test目录下，创建了一个Hello目录：



2）使用git init命令把这个目录变成Git可以管理的仓库：



命令输入后，会提示你，已经创建了一个空的Git仓库。此时你会在hello目录下发现一个隐藏目录.git



这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。如果你没有看到.git目录，那是因为这个目录默认是隐藏的，用ls -ah命令就可以看见。

此处的hello目录就是我们的：**工作区**，存放所有当前文档。此目录下的文件才会被Git管理

hello中的.git目录就是我们的：**本地仓库**，管理并保存所有的文档变化及历史状态。

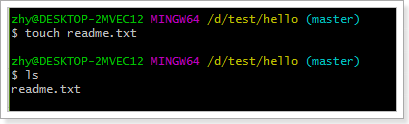
总结：创建版本库的步骤：

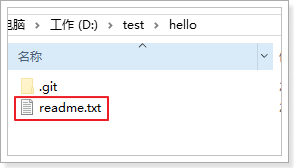
* + 1. 进入需要管理的目录
    2. 执行 git init 命令

## 添加文件并提交

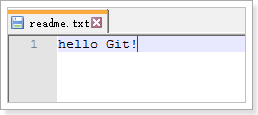
版本控制系统，其目的就是跟踪文本文件的改动，例如我们开发时编写的.java、.xml、.properties本质都是文本文件。文件中每一个字符的变化都会被跟踪并且管理。

1）我们在当前的hello目录下创建一个新的文本文件：readme.txt

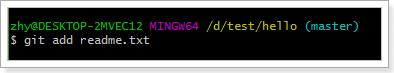




编写一段文字（注意，一定不要用windows的记事本）：hello Git!

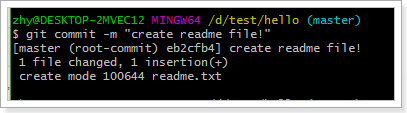


2）接下来，我们使用 git add 命令，将文件添加到暂存区



没有任何的反应，证明没有问题

3）使用 git commit 命令，将暂存区文件提交到本地仓库



命令解释：

git commit 命令可以将暂存区的文件提交到版本库。

-m 参数，是本次提交的说明信息，用来注释本次提交做了些说明事情。

**总结**，将一个文件添加到本地仓库，分两步：

* + 1. 使用 git add <file> 命令，添加文件。可以一次添加多个文件。
    2. 使用 git commit 命令，提交，一次即可。

可能大家会有疑问，为什么这里不是直接commit提交，而是要经过add和commit两个步骤呢？

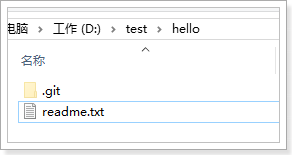
这就关系到Git的版本库中的 工作区 暂存区概念了。

## 工作区、暂存区、版本库

我们先来理解下Git 工作区、暂存区和版本库概念

### 工作区：

工作区就是你在电脑里能看到的目录。比如我们刚刚创建的hello目录：

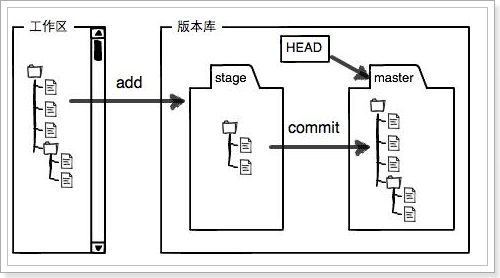


其中包含了一个隐藏目录 .git ，其它就是我们需要被管理的文件。

### 版本库及暂存区

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。



前面讲了我们把文件往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。

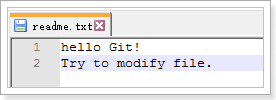
因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

## 管理文件修改

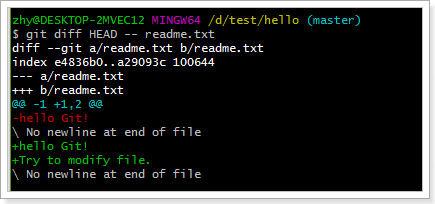
被版本库管理的文件不可避免的要发生修改，此时只需要直接对文件修改即可。修改完毕后需要将文件的修改提交到版本库。

我们对readme.txt文件进行修改，添加一行数据：



### 差异比较

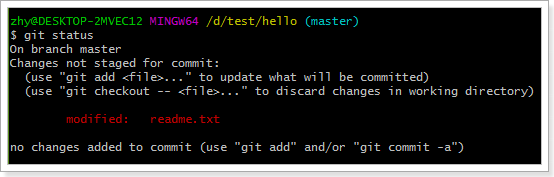
用git diff -- readme.txt命令可以查看工作区和版本库里面最新版本的区别：



可以发现，与版本库中的 readme.txt相比，我们多了一行文本！

### 查看状态，提交修改

我们如果不确定自己的哪些文件被修改了，可以使用git status 命令，查看当前工作区的状态：



可以清楚的看到：changes not staged for commit（修改没有被缓存，需要使用git add来进行添加操作）

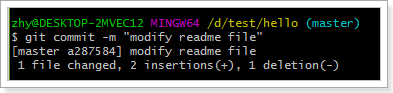
我们使用git add 命令，添加到暂存区：



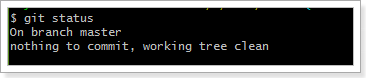
再次查看状态：



这次提示： changes to be commited （修改需要被提交），我们使用git commit 进行提交



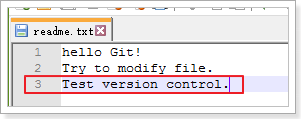
再次查看状态：



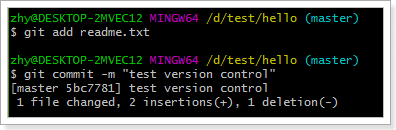
提示说：工作区很干净，没有任何需要提交，搞定！

### 版本回退

现在，我们再次修改readme.txt，添加一行内容：

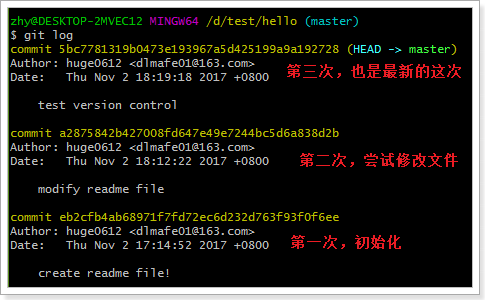


然后提交到版本库：

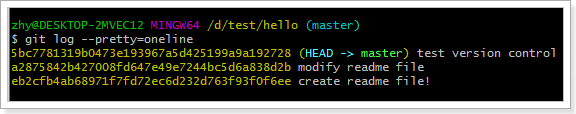


#### 日志查看

我们通过 git log 命令，可以查看历史的每次提交信息：



如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：



可以发现，目前为止，我们已经在本地仓库中提交了3次，也就是说有3个不同版本。其中，最近的这个版本有一个标示：HEAD ，这就是标记当前分支的当前版本所在位置。本例当中，当前版本即 test version control这次提交。

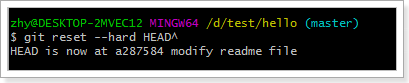
另外，在log中，每一个版本的前面，都有一长串随即数字：5bc7781319b…192728 ，这是每次提交的commit id ，这是通过SHA1算法得到的值，Git通过这个唯一的id来区分每次提交。

#### 版本回退

现在，假设我们要回到上一级版本，该如何操作呢？

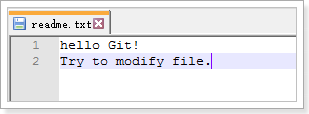
首先，Git通过HEAD来判断当前所在的版本位置。那么上一个版本，就用HEAD^标示，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

如果要从 “test version control” 回退到 “modify readme file” ，我们可以使用 git reset 命令



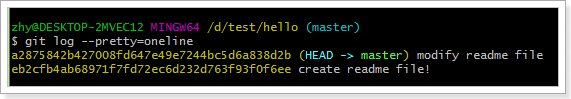
提示说：HEAD 现在已经被设置到 a287584 的版本，即 modify readme file。

我们查看readme.txt：



果然，版本已经回退了，最新添加的数据“Test version control”已经没了。

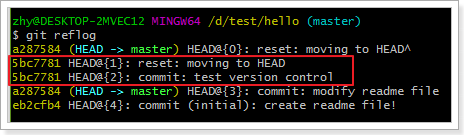
此时再次查看日志，发现只剩下2次提交信息了，第三次提交的信息已经没了：



假如此时我后悔了，还想回到第3次提交的版本，怎么办？

#### 查看所有关联日志

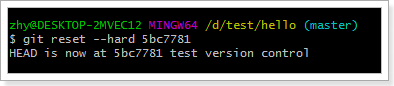
我们可以通过git reflog命令，看到以前的每次执行动作：



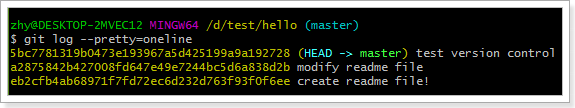
其中红框内的部分，就是我们第三次提交的日志信息。前面的5bc7781 就是第三次提交的 commit id 的前几位数字。

我们可以通过指定commit id 的方式，来指定HEAD的位置：

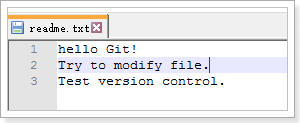
指令：git reset --hard {commit id}



查看日志：



查看文件：



数据又回来了！

#### 总结

如果要进行版本回退或前进，一般分两步：

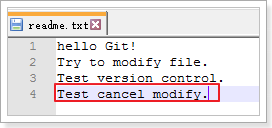
1. 通过git log 或 git reflog 查看操作日志吗，查找版本的commit id
2. 通过 git reset --hard <commit id> 设置HEAD到指定版本

其实版本的回退，仅仅是修改HEAD指针的位置而已，因此Git进行版本的切换，比svn要快的多！

### 撤销修改

#### 撤销工作区修改

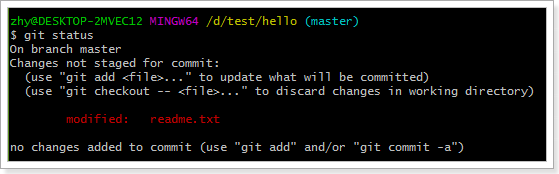
现在我们在readme.txt中添加一行数据：



在你提交前，你突然发现这个修改是有问题的，你打算恢复到原来的样子。怎么办？

如果修改的并不多，我们完全可以手动恢复到原始状态。但是如果改动比较大，手动处理就很容易有遗漏，而且很麻烦。怎么办？

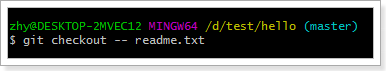
查看状态：



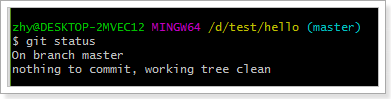
Git提示我们，现在文件已经修改，等待被staged（暂存）。我们有两个选择：

1. 可以使用git add 来添加到暂存区，接着去提交文件
2. 可以使git checkout -- <file> 来撤销修改

所以，这里我们选择第二种方案：

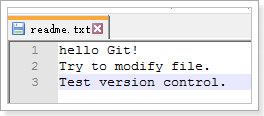


再次查看状态：



工作区是干净的！

查看文件：



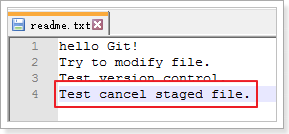
修改已经被撤销了！

#### 撤销staged（暂存区）修改

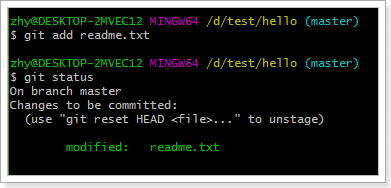
刚才的案例中，我们修改了数据，并没有add带暂存区，处理起来还算简单。

如果我们已经吧数据add 到了暂存区，又该如何处理呢？

我们首先添加一行数据到readme.txt



并且添加到staged（暂存区），然后查看状态

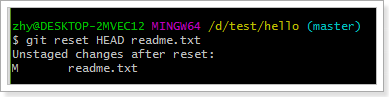


有一个修改等待被提交，并且有一行提示：

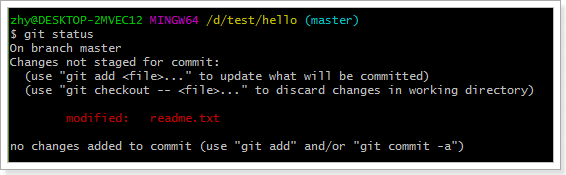


可以使用 git reset HEAD <file> 来撤销缓存修改。

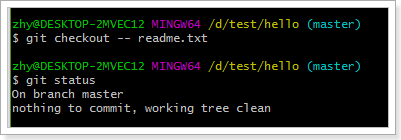
我们前面说过，git reset 命令可以进行版本回退，此处reset 指定的是HEAD ，而不是其他版本，因此就有撤销缓存修改的作用：



查看状态：

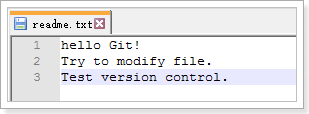


发现文件的修改被撤回到了工作区，尚未添加到staged（暂存区），我们再次执行git checkout -- <file> 即可撤销工作区修改



工作区干净了！

查看文件：



文件也恢复了原来的状态，整个世界都清净了！

#### 总结

撤销修改分两种情况：

1. 撤销工作区修改，使用git checkout -- <file>
2. 撤销暂存区修改，分两步：
   1. 使用git reset HEAD <file>来撤销暂存区修改。
   2. 使用git checkout -- <file>来撤销工作区修改

git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

**4.远程仓库**

到目前为止，我们已经学会了如何在本机利用git进行文件版本管理，但是如果要想进行多人协作，我们就必须使用远程仓库。将本地仓库的数据同步到远程仓库，实现多人协作开发。

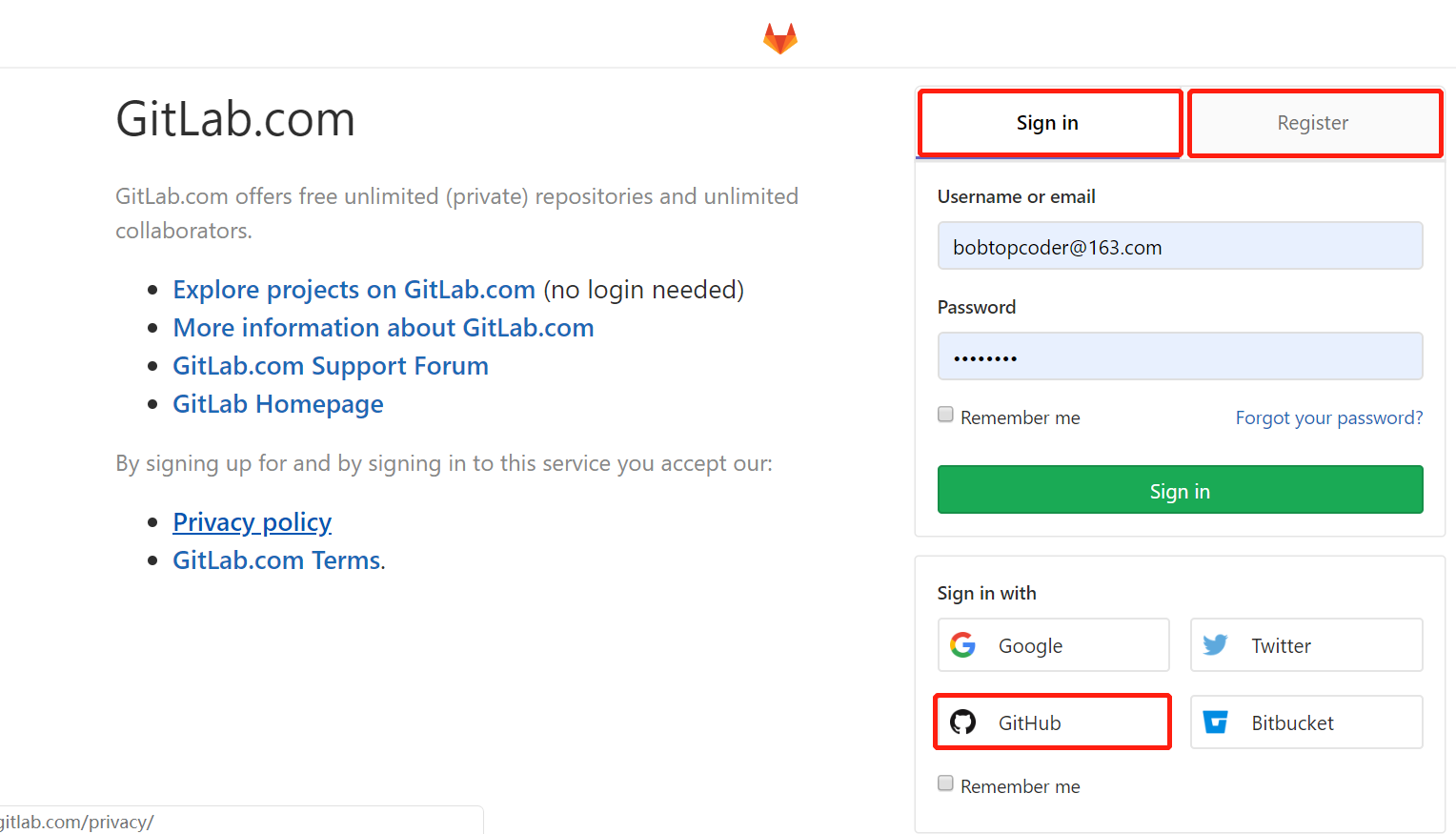
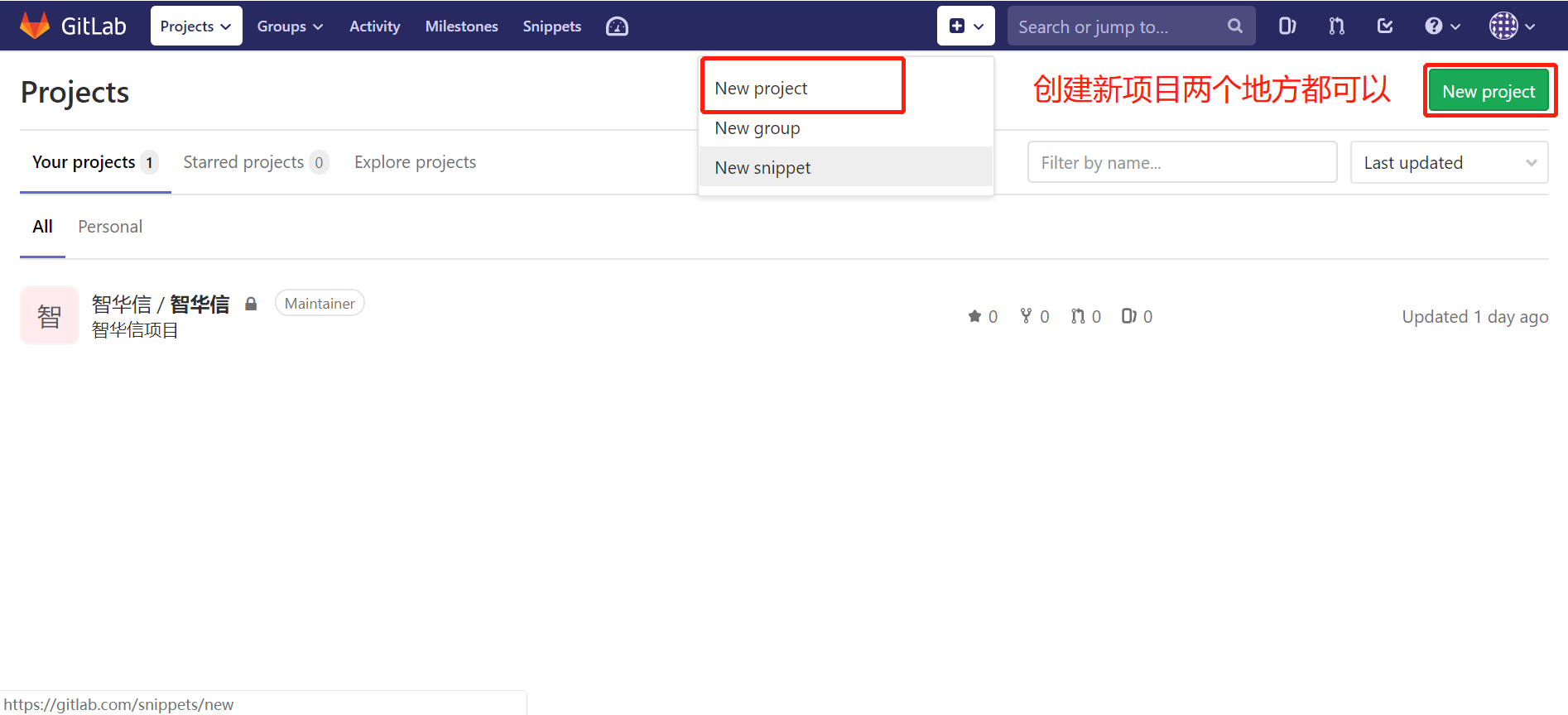
目前比较热门的代码托管社区：GitLab 网址： <https://gitlab.com> 提供了免费的远程git仓库功能。阿里云一体机里面也是集成了gitlab

## 4.1访问远程仓库

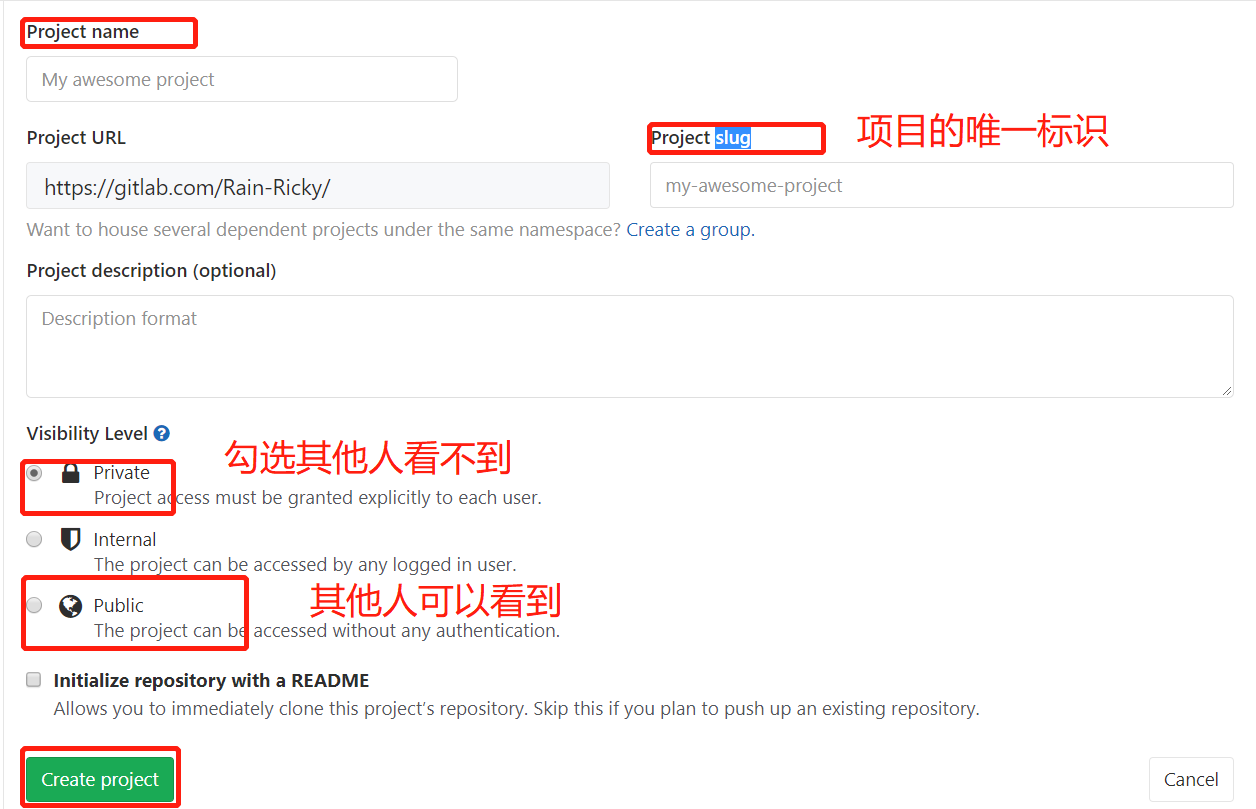
### 注册并登录

访问：<https://gitlab.com/users/sign_in>

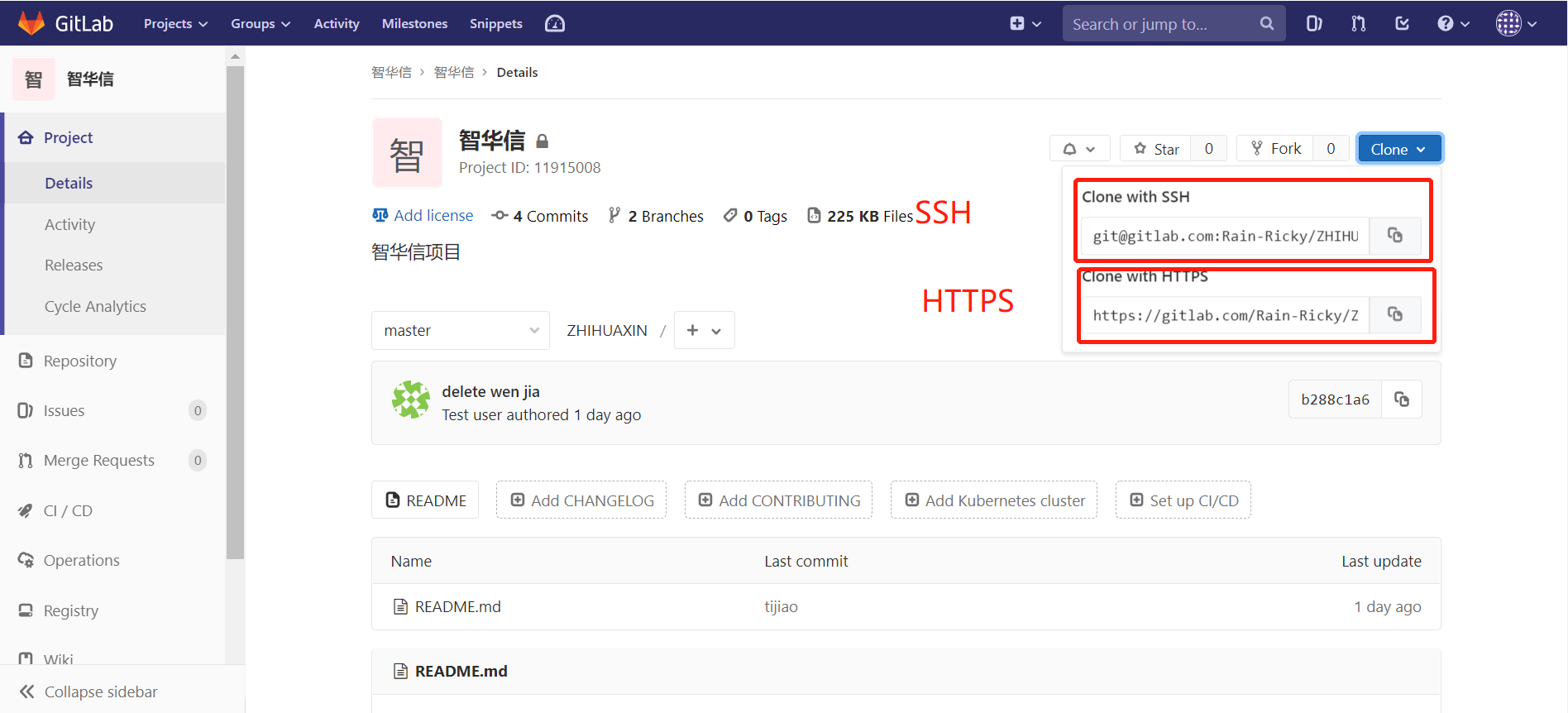
大家自己注册并登录：



### 4.1.2填写仓库信息：



仓库创建完毕，可以看到，如果我们要与远程仓库同步，这里支持两种不同的通信协议，ssh和http。当我们选中一种协议后，后面会出现对应的远程仓库地址。



这里推荐使用ssh协议。因为http速度较慢。

但是要使用SSH协议进行同步，你就必须生成SSH密钥，并添加到当前仓库的许可列表中！

### 4.1.3 SSH协议

#### 什么是ssh?

SSH是英文Secure Shell的简写形式。通过使用SSH，你可以把所有传输的数据进行加密，这样"中间人"这种攻击方式就不可能实现了，而且也能够防止DNS欺骗和IP欺骗。

使用SSH，还有一个额外的好处就是传输的数据是经过压缩的，所以可以加快传输的速度。SSH有很多功能，它既可以代替Telnet，又可以为FTP、Pop、甚至为PPP提供一个安全的"通道"。

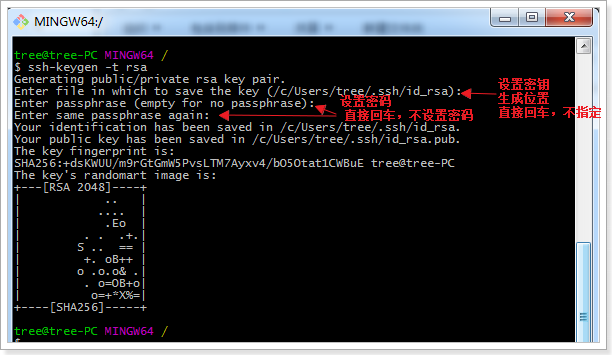
如果一个用户从本地计算机，使用SSH协议登录另一台远程计算机，我们就可以认为，这种登录是安全的，即使被中途截获，密码也不会泄露。

最早的时候，互联网通信都是明文通信，一旦被截获，内容就暴露无疑。1995年，芬兰学者Tatu Ylonen设计了SSH协议，将登录信息全部加密，成为互联网安全的一个基本解决方案，迅速在全世界获得推广，目前已经成为Linux系统的标准配置。

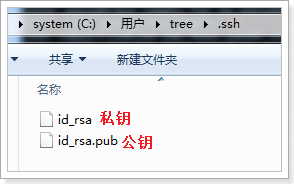
#### 4.1.2.2创建SSH密钥：

Git-bash中已经集成了ssh功能，所以我们只需要简单的命令，即可生成密钥：ssh-keygen -t rsa

一路回车向下走，不要输入任何内容即可！

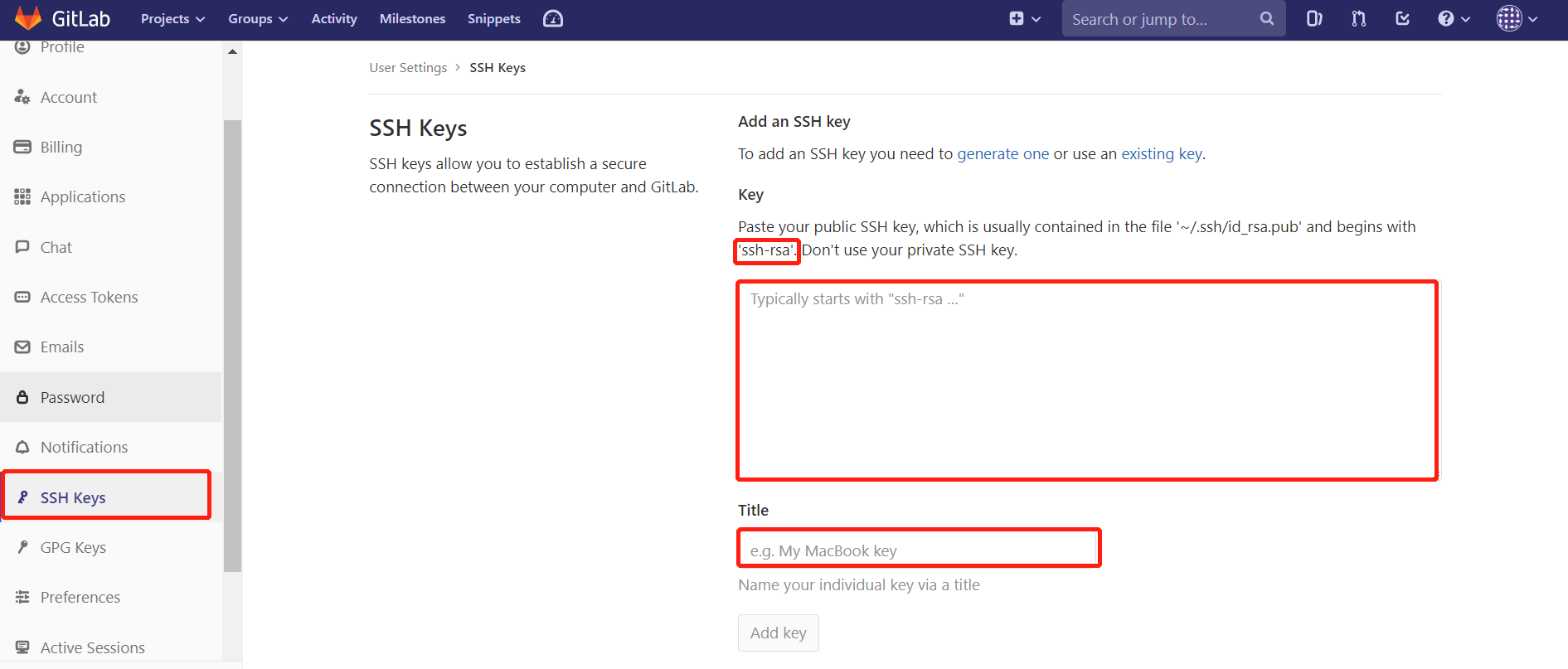


执行命令完成后,在window本地用户.ssh目录C:\Users\用户名\.ssh下面生成如下名称的公钥和私钥:

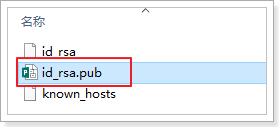


#### 4.1.2.3添加SSH密钥到git私服

点击SSH密钥设置，并添加密钥：



密钥的内容，就是刚刚生成的两个密钥中的公钥：id\_rsa.pub



用文本编辑工具（Notepad++）打开，并且复制内容即可。

将这段公钥添加到用户，这就说明，这台公钥的机器具备了访问这个git账号中的远程仓库的权限！因此一定要慎重添加。不能随意让别人操作你的仓库！

### 4.1.4推送远程仓库

在本地创建文件夹然后创建仓库使用命令创建

Touch README.md

Git init

Git add README.md

Git commit -m “first commit”

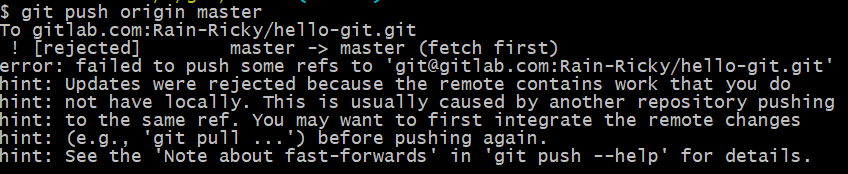
关联远程仓库：

Git remote add origin [git@gitlab.com:Rain-Ricky/hello-git.git](mailto:git@gitlab.com:Rain-Ricky/hello-git.git)

推送本地仓库到远程仓库的master分支：

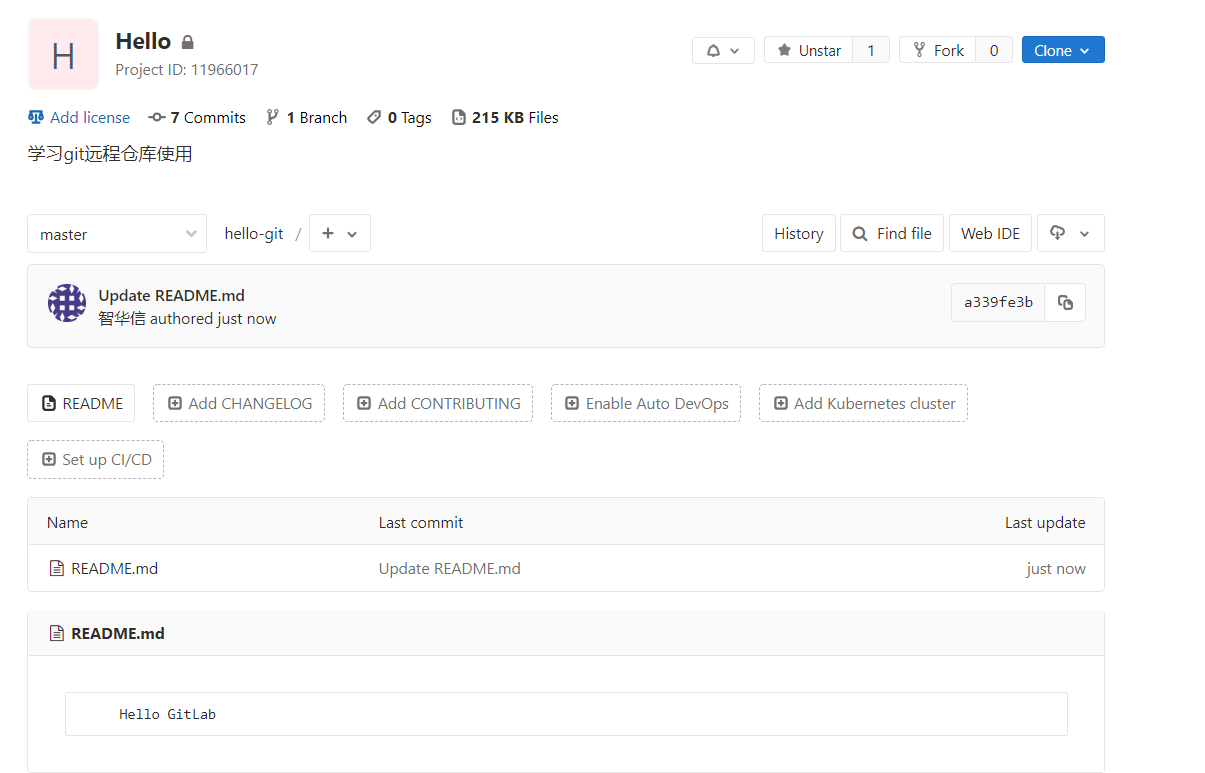
Git push -u origin master

【注这里有一问题如遇到就执行下面的步骤】



Git pull –rebase origin master

推送成功！此时登录远程仓库，刷新页面，可以看到数据已经推送，并且与本地仓库完全一致：



从现在起，只要本地作了提交，就可以通过命令：

git push origin master

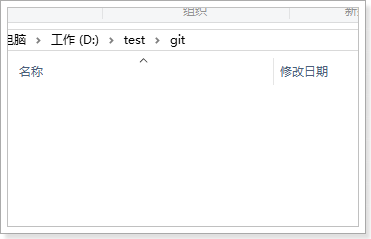
把本地master分支的最新修改推送至GitLab，现在，你就拥有了真正的分布式版本库！

### 4.1.5克隆远程仓库

现在，假设一个新的成员要加入我们的开发队伍，那他首先要做的第一件事情，一定是从远程仓库获取所有代码。此时就可以使用克隆动作。

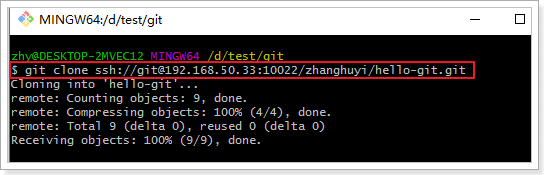
克隆远程仓库：就是从远程把仓库复制一份到本地，克隆后会创建一个新的本地仓库。

我们随意找到一个目录:

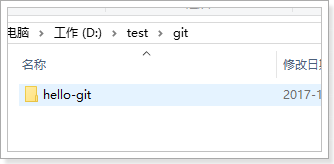


打开控制台，输入命令：git clone 远程仓库地址

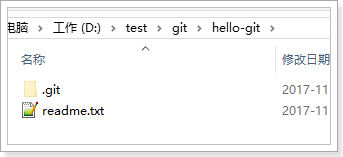
输入命令：



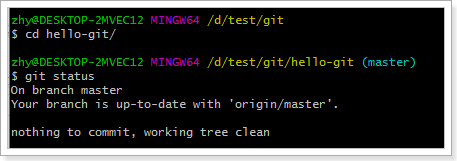
查看目录：



内容：



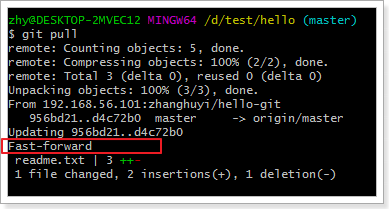
查看状态：



OK，你已经与远程仓库完全一致了，开始你的工作吧！

### 4.1.6拉取远程仓库数据

在本地拉取最新代码，使用git pull 命令



## 4.2总结

注意：

1. 如果要使用SSH协议与远程仓库同步，就必须先在本地生成公钥和私钥，然后将公钥添加到远程的SSH列表
2. 如果已经有本地仓库，并且想要与远程仓库关联，一般需要两步：
   1. 通过git remote add origin + 远程仓库地址 进行关联
   2. 通过git push -u origin master推送本地仓库修改到远程仓库
3. 如果没有本地仓库，先创建远程仓库，然后通过git clone + 远程仓库地址 进行克隆并创建本地仓库
4. git pull 拉取远程仓库数据

# 分支管理

## 5.1分支有什么用呢？

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。

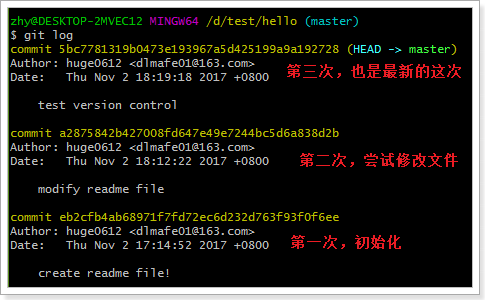
其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。

但Git的分支是与众不同的，无论创建、切换和删除分支，Git在1秒钟之内就能完成！无论你的版本库是1个文件还是1万个文件。

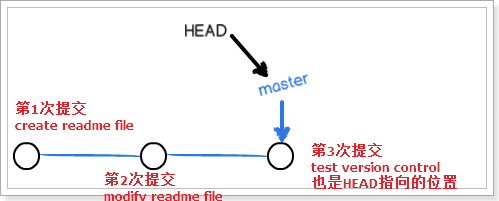
## 5.2 Git的分支管理原理

我们的每次提交，都对应一个具体的时间点，git会把这许多的时间点串起来，就形成了一条时间线，这条时间线就是一个分支。Git中默认的分支就是主分支，叫master。

我们查看当前的提交日志：



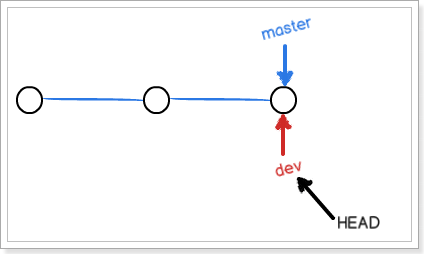
发现总共有3次提交，这3次提交可以串起来成一条时间线，就是master分支：



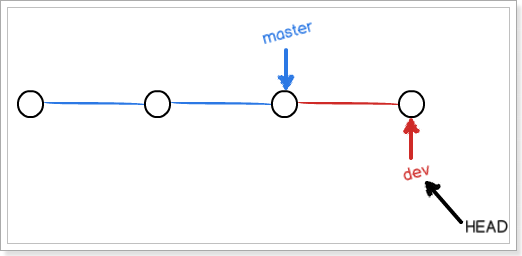
每次提交，master分支都会新增一个时间点，分支线也不断变长。

当我们创建新的分支，例如dev分支。Git会创建一个新的指针，叫做dev，指向跟master相同的时间点（提交点），这样分支就创建好了，你的工作区无需任何改变，创建分支的速度非常的快。

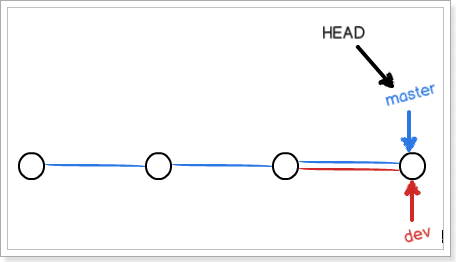
而要切换分支，只需要把HEAD指向dev即可，所以你的分支实现了光速切换！



不过，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：

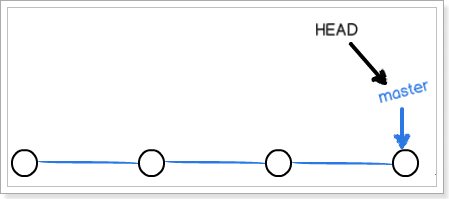


假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



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：



你会发现Git的分支管理，基本就是创建新的指针，改变HEAD指向，删除指针等操作，几乎没有文件的增删。所以速度非常快！

## 5.2分支的创建和合并

### 5.2.1创建分支

我们可以使用 git checkout -b 分支名 来创建并切换到新的分支：

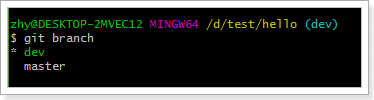


你会注意到我们已经切换到了dev分支。 git checkout 加上 -b 参数，就等同于创建分支，并切换分支。相当于以下两条命令：

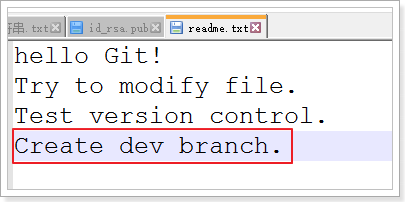
git branch dev # 创建分支

git checkout dev # 切换到具体分支

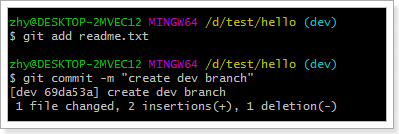
使用git branch 查看所有分支，当前分支前面会有一个\*表示：



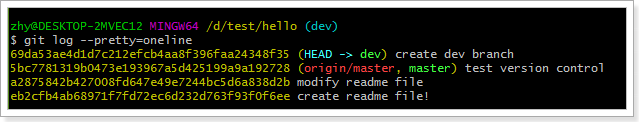
然后我们可以在dev分支上进行修改和提交。例如我们在readme.txt上添加一行文字：



提交修改：

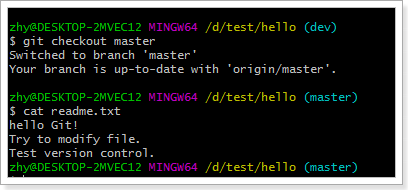


此时，dev分支已经比master领先了一次提交，并且HEAD指向的是dev

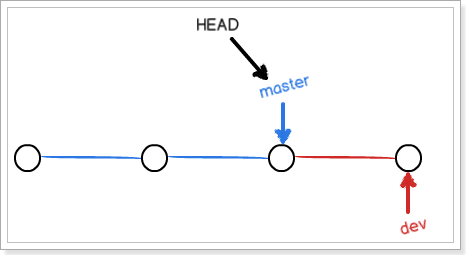


### 5.2.2合并分支

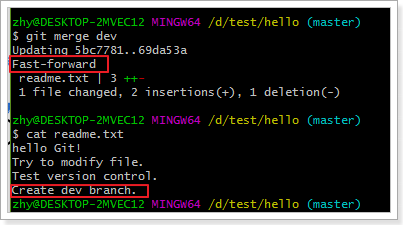
我们使用git checkout master切换回master分支，查看内容：



发现readme并没有改变，因为刚才修改的是dev分支。此时的HEAD已经指向了master了：



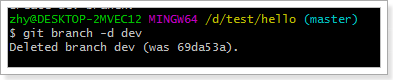
我们使用git merge dev命令将 dev分支的修改合并到master分支：



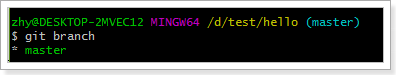
git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

### 5.2.3删除分支

合并完成后，就可以放心地删除dev分支了，可以使用git branch -d dev 命令删除dev分支，dev就是具体的分支名



再次查看分支列表：



### 5.2.4总结：

1. 使用git branch 分支名 创建分支
2. 使用git checkout 分支名 来切换分支
3. 也可以使用 git checkout -b 分支名 来完成 创建并切换分支的操作
4. 使用git merge 分支名 来合并分支到当前分支
5. 使用git branch -d 分支名 来删除指定分支，注意：要删除一个未合并的分支。需要使用-D参数进行强制删除

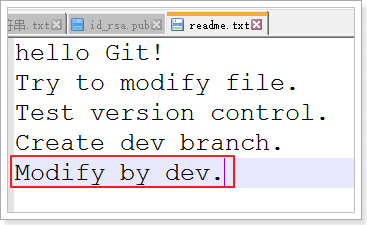
## 5.3解决冲突

### 5.3.1制造冲突

现在我们新建一个分支：dev



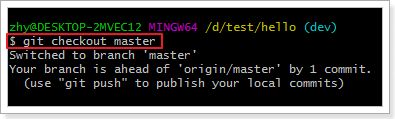
然后修改readme.txt：



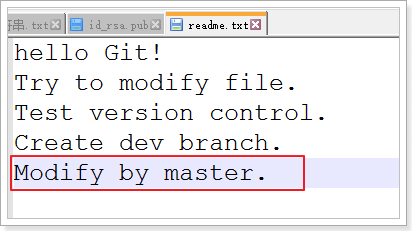
将dev的修改提交：



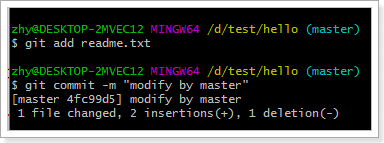
切换到master分支：



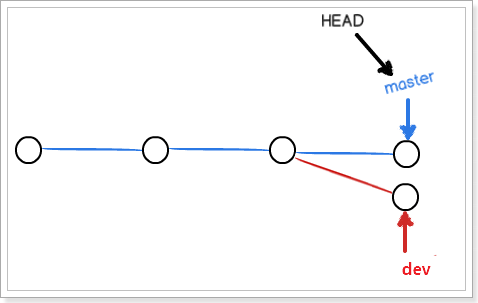
并且在readme.txt最后添加内容：



提交数据：

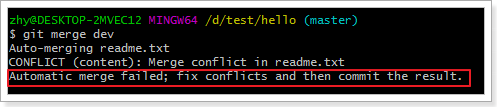


现在，master和dev都有了各自新的提交，变成了这样：



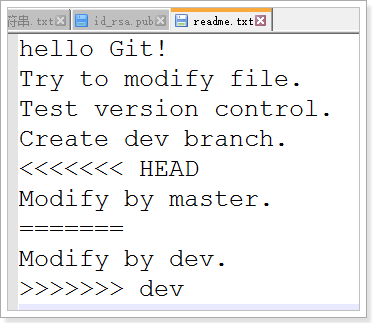
### 5.3.2解决冲突

这种情况下，是无法进行快速合并的。我们试一下：

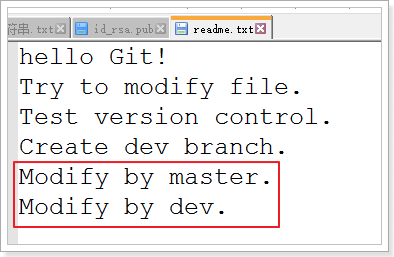


自动合并失败，必须先解决文件冲突，才能提交。

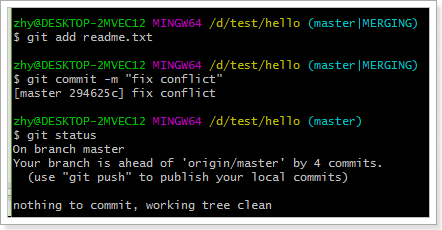
此时查看readme.txt文件：



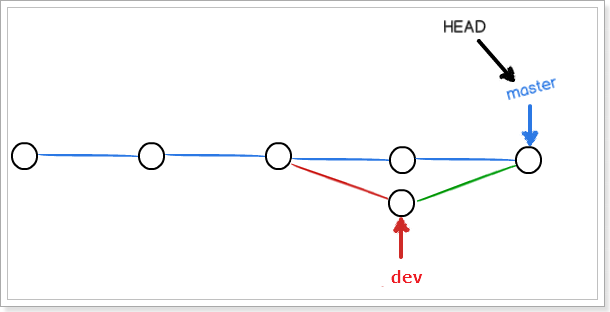
我们可以根据实际情况进行冲突解决，比如两者都保留：



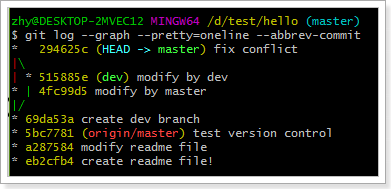
然后再次提交：



工作区就干净了。此时master和dev分支线就变成了这样：



可以用git log --graph --pretty=oneline --abbrev-commit命令来查看：



接下来就可以删除dev分支了。



## 5.4保存工作区状态(了解)

现在假设这样一个场景：

我们正在dev上进行开发，代码进行了一半，突然线上出现了一个紧急BUG需要你进行修复。我们可以创建一个新的hotFix分支来进行修复，但是等等。我当前的dev分支还有未提交的修改，怎么办？如果直接切换到新的分支，那么当前dev 的工作就白费了！有同学可能会说，我们先把代码提交啊，回头再接着写。但是这样的提交是不合理的，也没有任何实际意义的，因为你的代码并没有完成，你提交的是不完整的功能！

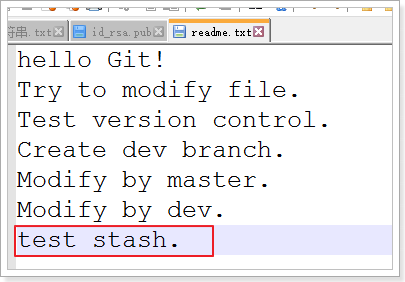
还好，git给我们提供了stash功能，可以把当前工作区的内容进行“快照”，等待以后恢复使用。

我们实际操作看看。

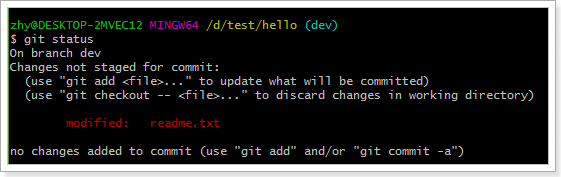
首先，我们创建并切换到dev分支：



修改readme.txt

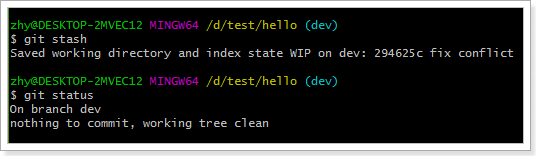


查看状态：



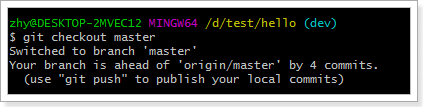
显然，readme有修改待提交。但是现在我们需要去进行紧急的BUG修复，没办法继续开发dev。

我们使用stash命令：



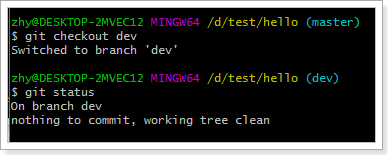
发现工作区已经被保存起来了，并且工作区再次变成了clean，可以放心的去新的分支修复BUG了。

我们切换回master：



现在假设你修复完成了BUG。。

bug修复完成，然后我们再次回到dev分支去查看：



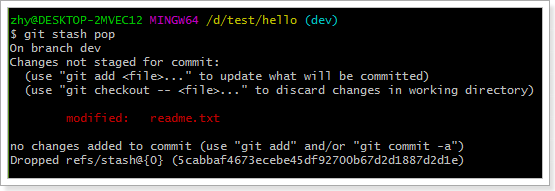
工作区是干净的，我们之前的修改去哪里了呢？通过git stash list命令来查看：



工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

1. 方式1：用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；
2. 方式2：用git stash pop，恢复的同时把stash内容也删了：
3. 注意，以上指令都可以在后面紧跟 stash的序号，来操作指定stash，例如： git stash apply stash@{0}

我们采用方式2：



发现工作区的内容又回来了：

