一、**Git** 和 **SVN** 的区别

	类型	描述
Git	分布式	本地有镜像,无网络时也可以提交到本地镜像,待到有网络时再push到服务器
SVN	集中式	无网络不可以提交, 和 Git 的主要区别是历史版本维护的位置

二、安装：

1. Git 下载地址 (Linux/Unix, Mac, Windows 等相关平台)

> 注意: 以下所有命令都在 *Git Bash* 中运行,不是 cmd, 拷贝命令的时候不用复制前面的 $ 符号

```
### 配置所有 Git 仓库的 用户名 和 email
$ git config --global user.name "Your Name"
$ git config --global user.email "youremail@example.com"

### 配置当前 Git 仓库的 用户名 和 email
$ git config user.name "Your Name"
$ git config user.email "youremail@example.com"

### 查看全局配置的 用户名 和 email
$ git config --global user.name      查看用户名
$ git config --global user.email      查看邮箱地址

### 查看当前仓库配置的 用户名 和 email
$ git config user.name      查看用户名
$ git config user.email      查看邮箱地址

# 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
# 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(不加 --global)。

$ git config --list --show-origin 查看所有的配置以及它们所在的文件
```

三、相关命令

```
(所有命令都在 Git Bash 中运行)
$ git                              查看 git 的相关命令 (git --help)
$ git --version                    查看 git 的版本
$ git config                       查看 git config 的相关命令
$ git pull origin develop          从远程(origin) 的 develop 分支拉取代码
```

1. 初始化本地仓库**:** 在 ***Git Bash*** 中输入对应的命令

```
注: 下面所有的命令使用的时候不用拷贝最前面的 $ 符号

$ cd d:
$ mkdir learngit
$ cd learngit
$ pwd

# cd: change directory 改变目录
# mkdir  创建目录
# pwd    用于显示当前目录
注意: 为避免遇到各种奇怪的问题,确保目录名 (包括父目录) 不含中文
```

```
# 不想要 git 管理跟踪的文件,可以在仓库根目录添加 .gitignore 文件,在里面写对应的规则
$ git init                 把当前目录初始化为 git 仓库
$ ls -ah                   查看当前目录下的文件,包含隐藏文件 (不带 -ah 看不了隐藏文件)
```

2. 添加文件到仓库

```
$ git add <file>               如: git add readme.txt
$ git commit -m "description"      如: git commit -m "add readme.txt"

# 添加文件到仓库分两步:
# 1. add 添加该文件到仓库,
# 添加许多同种类型的文件,可以使用通配符 * (记得加引号)  如: git add "*.txt"  命令就是添加所有 .txt 文件
# 2. commit 提交该文件到仓库, description 为你对该次提交的描述说明,
注意: 可以多次 add 不同的文件,commit 可以一次提交多个文件
```

3. 查看仓库目前状态 **(**项目是否有修改、添加、未追踪的文件等**)**

```
$ git status
```

4. 查看修改内容,查看文件不同 **(difference)**

```
$ git diff
$ git diff <file>
$ git diff --cached
$ git diff HEAD -- <file>
# git diff 查看工作区(work dict)和暂存区(stage)的区别
# git diff --cached 查看暂存区(stage)和分支(master)的区别
# git diff HEAD -- <file> 查看工作区和版本库里面最新版本的区别
如: git diff readme.txt  表示查看 readme.txt 修改了什么,有什么不同
```

5. 查看提交日志

```
$ git log
$ git log --oneline     #美化输出信息,每个记录显示为一行,显示 commit_id 前几位数
$ git log --pretty=oneline     #美化输出信息,每个记录显示为一行,显示完整的commit_id
$ git log --graph --pretty=format:'%h -%d %s (%cr)' --abbrev-commit --
$ git log --graph --pretty=oneline --abbrev-commit

# 显示从最近到最远的提交日志
# 日志输出一大串类似 3628164...882e1e0 的是commit_id (版本号),和 SVN 不一样,Git 的commit_id 不是 1，2，3…… 递增的数字，而是一个 SHA1 计算出来的一个非常大的数字，用十六进制表示，因为 Git 是分布式的版本控制系统，当多人在同一个版本库里工作，如果大家都用 1，2，3……作为版本号，那肯定就冲突了
# 最后一个会打印出提交的时间等, (HEAD -> master)指向的是当前的版本
# 退出查看 log 日志,输入字母 q (英文状态)
```

6. 版本回退

```
$ git reset --hard HEAD^
$ git reset --hard <commit_id>

# HEAD     表示当前版本，也就是最新的提交
# HEAD^    上一个版本
# HEAD^^   上上一个版本
# HEAD~100    往上100个版本

# 回退到 commit_id 对应的那个版本,commit_id 为版本号,只需要前几位就行
```

7. 查看命令历史 **(**用于版本切换**)**

```
$ git reflog
# 假如我们依次提交了三个版本 a->b->c,然后昨天我们从版本 c 回退到了版本 b,今天我们又想要回到版本 c,此时就可以使用 reflog 命令来查找 c 版本的 commit_id,然后使用 reset 命令来进行版本回退
```

8. 撤销修改

######- 丢弃工作区 (Working Directory) 的修改

```
$ git restore <file>  (建议使用) (如: git restore readme.txt)
$ git checkout -- <file>
# 命令中 -- 很重要，没有就变成“切换到另一个分支”的命令
```

######- 丢弃暂存区 (stage/index) 的修改

```
# 第一步: 把暂存区的修改撤销掉(unstage)，重新放回工作区
$ git restore --staged <file>

# 第二步: 撤销工作区的修改
$ git restore <file>
```

######- 小结

> - 当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令`git restore`。
> - 当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令`git restore --staged`，就回到了场景1，第二步按场

> 景1操作。
> - 已经提交了不合适的修改到版本库时，想要撤销本次提交，参考 版本回退 一节，不过前提是没有推送到远程库。

9. 删除文件

```
$ git rm <file>

# git rm <file> 相当于执行
- rm <file>
- git add <file
```

其他命令

```
$ cat <file>     显示文件内容,如: cat readme.txt 就是在 git bash 中显示该文件内容
$ cd ~           进入用户主目录
$ open ~/.ssh    Mac 打开存放 ssh 文件夹
```

四、相关名词理解：

1. 工作区 **(Working Directory):** 自己电脑里能看到的目录

2. 版本库 **(Repository):** 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

五、远程仓库：

1. 创建 **SSH Key**

```
$ ssh-keygen -t rsa -C "youremail@example.com"
# 邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。
```

> 在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id_rsa和id_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key

> 如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有 id_rsa 和 id_rsa.pub 两个文件，这两个就是 SSH Key 的秘钥对，id_rsa 是私钥，不能泄露出去，id_rsa.pub 是公钥，可以放心地告诉任何人。

2. 登录 **GitHub ,**在 **Settings** 中找到 **SSH** 设置项中添加新的 **SSH Key,**设置任意 **title,**在 **Key** 文本框里粘贴 **id_rsa.pub** 文件的内容

```
# 复制Key用这种方式复制
$ cd ~/.ssh
$ cat id_rsa.pub

$ open ~/.ssh   (Mac 下打开存放 Github 生成的 ssh Key 文件夹)

$ pbcopy < ~/.ssh/id_rsa.pub  Mac 下拷贝生成的公钥内容
```

3. 关联远程仓库 **(**先有本地仓库**)**

```
$ git remote add origin git@github.com:renyuns/learngit.git
# 后面的地址换成自己的 GitHub 仓库地址
```

4. 推送到远程仓库

```
$ git remote        查看远程库信息
$ git remote -v     查看远程库详细信息
$ git remote rm origin  删除已关联的远程库 origin
$ git push -u origin master    #第一次推送
$ git push origin master       推送本地 master 分支到远程库
$ git push origin dev          推送本地 dev 分支到远程库
#  除了第一次推送,不需要添加 -u 参数

# 一个本地库关联多个远程库,例如同时关联 GitHub 和 Gitee:
# 1. 先关联GitHub的远程库：(注意:远程库的名称叫 github，不叫 origin)
$ git remote add github git@github.com:renyun/learngit1.git
# 2. 再关联Gitee的远程库：(注意:远程库的名称叫 gitee，不叫 origin)
$ git remote add gitee git@gitee.com:renyun/learngit1.git
# 3. 推送到远程库
$ git push github master
$ git push gitee master
```

> 加上了-u参数，Git 不但会把本地的 master 分支内容推送的远程新的 master 分支，还会把本地的 master 分支和远程的master分支关联起来

5. 从远程仓库克隆 **(**先有远程库**)**

```
$ git clone git@github.com:renyuns/gitskills.git
# GitHub 支持多种协议,上面是 ssh 协议,还有 https 协议
```

六、分支

```
$ git branch        查看分支列表及当前分支
$ git branch dev    创建 dev 分支
$ git switch dev    切换到 dev 分支  (git checkout dev)
$ git switch -c dev    创建并切换到新的 dev 分支  (git checkout -b dev)
$ git switch -c dev origin/dev  创建远程 origin 的 dev 分支到本地并切换到该分支
$ git branch -d dev    删除 dev 分支
$ git branch -D dev    强制删除 dev 分支
$ git merge dev        合并 dev 分支到当前分支 (当有冲突的时候,需要先解决冲突)
$ git merge --no-ff -m "merge with no-ff" dev  合并 dev 分支到当前分支(禁用 Fast forward 合并策略)

$ git pull  拉取远程分支最新的内容
$ git branch --set-upstream-to=origin/dev dev  指定本地 dev 分支与远程 origin/dev 分支的链接

# 为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去
# 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 合并就看不出来曾经做过合并

$ git log --graph  查看分支合并图
$ git log --graph --pretty=oneline --abbrev-commit

$ git stash  保存当前工作区和暂存区的修改状态,切换到其他分支修复 bug 等工作,然后在回来继续工作
$ git stash list  查看保存现场的列表
$ git stash pop    恢复的同时把 stash 内容也删除
$ git stash apply  恢复现场，stash内容并不删除
$ git stash drop    删除 stash 内容
$ git stash apply stash@{0}  多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash
# 通常在 dev 分支开发时,需要有紧急 bug 需要马上处理,保存现在修改的文件等,先修复 bug 后再回来继续工作的情况

$ git cherry-pick <commit> 复制一个特定的提交到当前分支(当前分支的内容需要先commit,然后冲突的文件需要解决冲突,然后 commit)

$ git rebase  把本地未push的分叉提交历史整理成直线(使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比)
```

七、标签

```
# 切换到对应的分支 branch 上,查看或者操作对应的标签 tag
$ git tag  查看所有的标签
$ git tag <tagname>  打标签(默认标签是打在最新提交的commit上) 如: git tag
v1.0
$ git tag <tagname> <commit_id>  给对应的 commit_id 打标签
$ git tag -a <tagname> -m "标签说明信息" <commit_id> 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 -d <tagname> 删除一个本地标签
$ git push origin :refs/tags/<tagname>可以删除一个远程标签
$ git show <tagname>  查看标签信息

$ git push origin <tagname>  推送一个本地标签到远程
$ git push origin --tags     一次性推送全部尚未推送到远程的本地标签

# 删除远程标签,需要先删除本地标签,然后在删除远程标签,如:删除标签 v0.9
$ git tag -d v0.9
$ git push origin :refs/tags/v0.9
```

八、相关工具及网站

1. Git 官网
2. GitHub-开源协作社区
3. Gitee(码云)-国内开源协作社区
4. 廖雪峰的 Git 教程-新手必看
5. 15 分钟学会 Git
6. Git Book
7. .gitignore 文件常用配置

九、参与开源项目

基本步骤：

1. 将他人的开源仓库 fork 到自己的 Github 上；
2. 将该开源仓库从自己的 Github 上克隆到本地；`git clone 项目地址`
3. 修改该项目；
4. 推送一个 pull reaquest 到他人开源仓库。（当然他人可选择接受或不接受）