Git

什么是Git

> Git是一个开源的分布式版本控制系统，可以有效、高速地处理从很小到非常大的项目版本管理。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下载

```
https://git-scm.com/download/
```

Git安装

windows系统安装Git按照默认选项一直下一步即可.

右键单击桌面, 看到Git GUI Here 和 Git Bash Here就说明安装成功.

Git GUI 是Git 图形界面, 不太好用... ; Git Bash 是Git 命令行, **初学Git建议熟悉常用命令.**

初次运行 Git 前的配置

当安装完 Git 应该做的第一件事就是设置你的用户名称与邮件地址。 这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到你的每一次提交中，不可更改：

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

再次强调，如果使用了 `--global` 选项，那么该命令只需要运行一次，因为之后无论你在该系统上做任何事情， Git 都会使用那些信息。 当你想针对特定项目使用不同的用户名称与邮件地址时，可以在那个项目目录下运行没有 `--global` 选项的命令来配置。

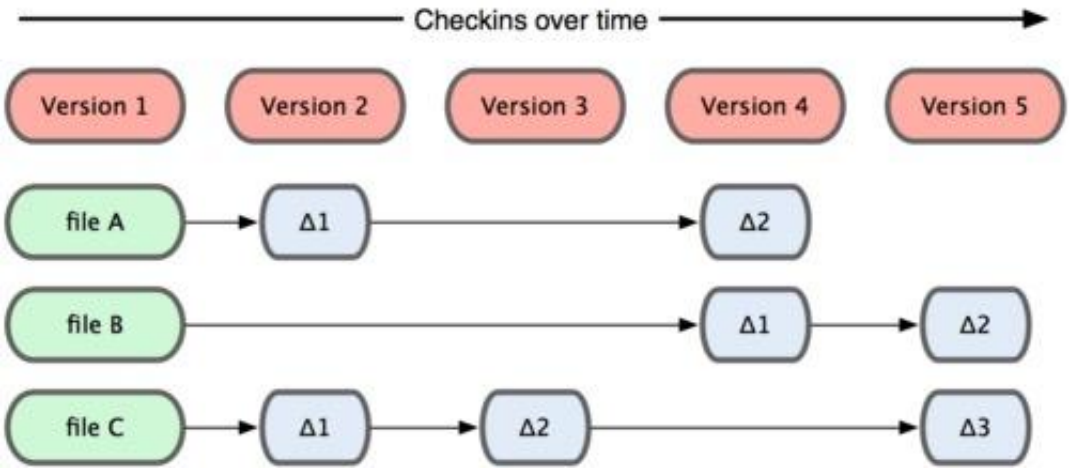
很多 GUI 工具都会在第一次运行时帮助你配置这些信息。

了解Git的工作原理

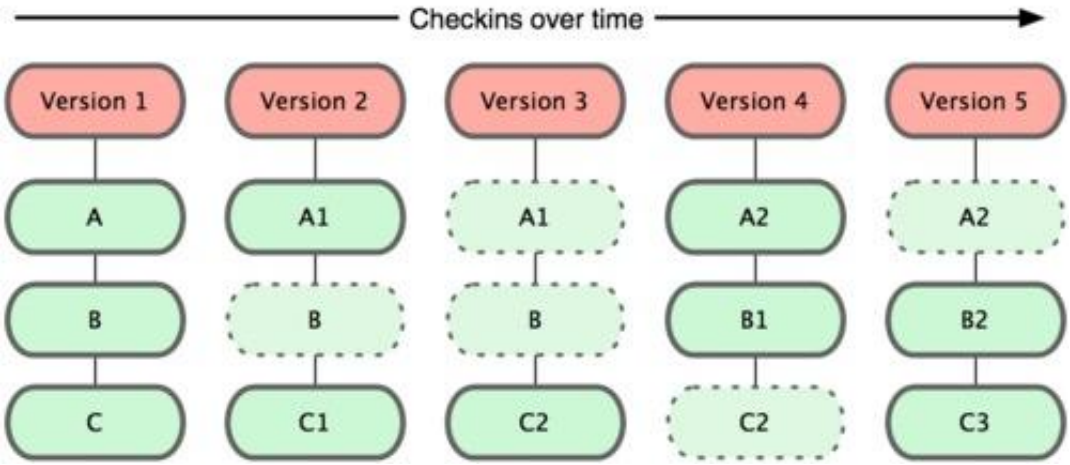
记录文件整体快照

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。

SVN在每个版本中，以单一文件为单位，记录各个文件的差异：



Git在每个版本中，以当时的全部文件为单位，记录一个快照：



Git的绝大多数操作都只需要访问本地文件和资源，不用连网。因为你的本机上，就已经是完整的代码库了。

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。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。该字串由40个十六进制字符组成，看起来就像是：

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git的工作完全依赖于这类指纹字串，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（modified） 已修改表示修改了某个文件，但还没有提交保存；
- 已暂存（staged） 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。

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，以及本地仓库。

Git常用操作(命令行)

Git克隆

假设已有Git工作目录, 我们需要从远程克隆代码：git clone [url]

```
$ git clone mobgit@134.32.51.60:learngit.git  #使用SSH传输协议
$ git clone git://134.32.51.60/learngit.git  #使用Git传输协议
$ git clone https://134.32.51.60/learngit.git  #使用HTTPS传输协议
```

Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。

检查当前文件状态

工作目录下的每一个文件都不外乎这两种状态：**已跟踪**或**未跟踪**。

已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。

工作目录中除已跟踪文件以外的所有其它文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。 **初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。**

编辑过某些文件之后，由于自上次提交后你对它们做了修改，Git 将它们标记为**已修改**文件。 我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存了的修改，如此反复。

要查看哪些文件处于什么状态，可以用 `git status` 命令。 如果在克隆仓库后立即使用此命令，会看到类似这样的输出：

```
$ git status
On branch master
nothing to commit, working directory clean
```

说明所有已跟踪文件在上次提交后都未被更改过, 当前目录下没有出现任何处于未跟踪状态的新文件.

假如我们创建了一个新文件README.md, 如果之前并不存在这个文件，使用 `git status` 命令，你将看到一个新的未跟踪文件：

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README.md

nothing added to commit but untracked files present (use "git add" to
track)
```

在状态报告中可以看到新建的 README.md 文件出现在 `Untracked files` 下面。 未跟踪的文件意味着 Git 在之前的快照（提交）中没有这些文件；Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”， 这样的处理让你不必担心将生成的二进制文件或其它不想被跟踪的文件包含进来。 不过现在的例子中，我们确实想要跟踪管理 README.md 这个文件。

跟踪新文件

使用命令 `git add` 开始跟踪一个文件。 所以，要跟踪 README.md 文件，运行：

```
$ git add README.md
```

此时再运行 `git status` 命令，会看到 README.md 文件已被跟踪，并处于暂存状态：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.md
```

只要在 `Changes to be committed` 这行下面的，就说明是已暂存状态。

暂存已修改文件

现在我们来修改一个已被跟踪的文件。 如果你修改了一个名为 `CONTRIBUTING.md` 的已被跟踪的文件，然后运行 `git status` 命令，会看到下面内容：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   CONTRIBUTING.md
```

文件 `CONTRIBUTING.md` 出现在 `Changes not staged for commit` 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。 要暂存这次更新，需要运行 `git add` 命令。 这是个多功能命令：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等。

文件对比

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 `git diff`：

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your
PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your
change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your
patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a
PR
 that highlights your work in progress (and note in the PR title that
it's
```

若要查看已暂存的将要添加到下次提交里的内容，可以用 `git diff --cached` 命令。（Git 1.6.1 及更高版本还允许使用 `git diff --staged`，效果是相同的，但更好记些。）

提交更新

每次准备提交前，先用 `git status` 看下，是不是都已暂存起来了， 然后再运行提交命令 `git commit`：

```
$ git commit -m "你的提交注释"
```

查看提交历史

查看提交历史 命令: git log

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

撤消操作

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 `--amend` 选项的提交命令尝试重新提交：

```
$ git commit --amend
```

这个命令会将暂存区中的文件提交。

例如，你提交后发现忘记了暂存某些需要的修改，可以像下面这样操作：

```
$ git commit -m 'initial commit'
$ git add 忘记提交的文件
$ git commit --amend
```

最终你只会有一个提交 - 第二次提交将代替第一次提交的结果。

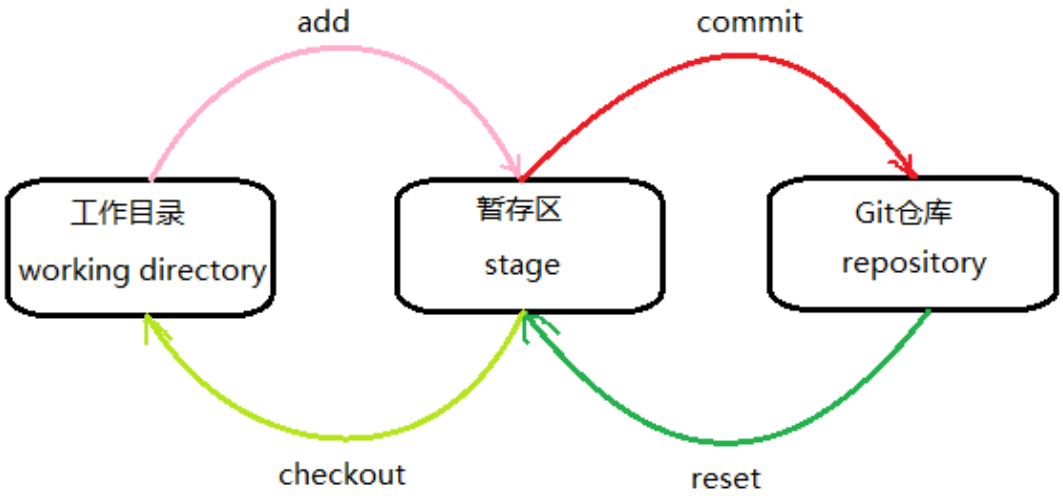
取消暂存的文件

```
$ git reset HEAD 暂存的文件
```

撤消对文件的修改

```
$ git checkout -- 已修改的文件
```

下图描述了文件三个状态以及互相转换的命令



以上操作都是在本地玩的。

Git分支

分支创建

例如创建一个 dev分支， 你需要使用 `git branch` 命令：

```
$ git branch dev
```

一般是在Git默认的master分支基础上创建新的分支。一般我们约定master分支是生产代码。

此时新创建好的分支还在本地。

分支切换

切换到刚才创建的dev分支

```
$ git checkout dev
```

删除分支

```
$ git branch -d 需要删除的分支
Deleted branch 需要删除的分支 (3a0874c).
```

分支的合并

例如我要合并dev分支到master上

```
$ git checkout master              #先切换到master分支
Switched to branch 'master'
$ git merge dev                    #合并dev到master
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

遇到冲突时的分支合并

合并冲突是这个样子的

```
$ git merge dev
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

此时 Git 做了合并，但是没有自动地创建一个新的合并提交。 Git 会暂停下来，等待你去解决合并产生的冲突。 你可以在合并冲突后的任意时刻使用 `git status` 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。 Git 会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。 出现冲突的文件会包含一些特殊区段，看起来像下面这个样子：

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> dev:index.html
```

这表示 `HEAD` 所指示的版本（也就是你的 `master` 分支所在的位置，因为你在运行 merge 命令的时候已经检出到了这个分支）在这个区段的上半部分（`=======` 的上半部分），而 `dev` 分支所指示的版本在 `=======` 的下半部分。 为了解决冲突，你必须选择使用由 `=======` 分割的两部分中的一个，或者你也可以自行合并这些内容。 例如，你可以通过把这段内容换成下面的样子来解决冲突：

```
<div id="footer">
    please contact us at email.support@github.com
</div>
```

在你解决了所有文件里的冲突之后，对每个文件使用 `git add` 命令来将其标记为冲突已解决。 一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决。

分支管理

`git branch` 命令不只是可以创建与删除分支。 如果不加任何参数运行它，会得到当前所有分支的一个列表：

```
$ git branch
  dev
* master  # *代表现在所处分支
  release
```

`--merged` 与 `--no-merged` 这两个有用的选项可以过滤这个列表中已经合并或尚未合并到当前分支的分支。 如果要查看哪些分支已经合并到当前分支，可以运行 `git branch --merged`：

```
$ git branch --merged
  dev
* master
```

查看所有包含未合并工作的分支，可以运行 `git branch --no-merged`

推送

如果希望和别人一起在名为 `dev` 的分支上工作，你可以像推送第一个分支那样推送它。运行 `git push (remote) (branch)`：

```
$ git push origin dev  #origin是Git上默认远程仓库名字
```