Git 简介

Git 是什么？

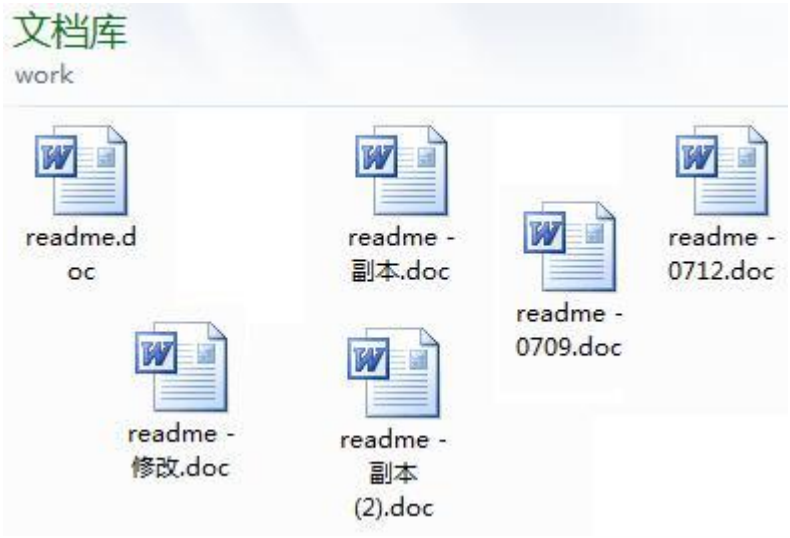
Git 是目前世界上最先进的分布式版本控制系统（没有之一）。

Git 有什么特点？简单来说就是：高端大气上档次！

那什么是版本控制系统？

如果你用 Microsoft Word 写过长篇大论，那你一定有这样的经历：

想删除一个段落，又怕将来想恢复找不回来怎么办？有办法，先把当前文件“另存为……”一个新的 Word 文件，再接着改，改到一定程度，再“另存为……”一个新文件，这样一直改下去，最后你的 Word 文档变成了这样：



过了一周，你想找回被删除的文字，但是已经记不清删除前保存在哪个文件里了，只好一个一个文件去找，真麻烦。

看着一堆乱七八糟的文件，想保留最新的一个，然后把其他的删掉，又怕哪天会用上，还不敢删，真郁闷。

更要命的是，有些部分需要你的财务同事帮助填写，于是你把文件 Copy 到 U 盘里给她（也可能通过 Email 发送一份给她），然后，你继续修改 Word 文件。一天后，同事再把 Word 文件传给你，此时，你必须想想，发给她之后到你收到她的文件期间，你作了哪些改动，得把你的改动和她的部分合并，真困难。

于是你想，如果有一个软件，不但能自动帮我记录每次文件的改动，还可以让同事协作编辑，这样就不用自己管理一堆类似的文件了，也不需要把文件传来传去。如果想查看某次改动，只需要在软件里瞄一眼就可以，岂不是很方便？

这个软件用起来就应该像这个样子，能记录每次文件的改动：

版本	文件名	用户	说明	日期
1	service.doc	张三	删除了软件服务条款 5	7/12 10

版本	文件名	用户	说明	日期
2	service.doc	张三	增加了 License 人数限制	7/12 18
3	service.doc	李四	财务部门调整了合同金额	7/13 9:
4	service.doc	张三	延长了免费升级周期	7/14 15

这样，你就结束了手动管理多个“版本”的史前时代，进入到版本控制的 20 世纪。

Git 的诞生

很多人都知道，Linus 在 1991 年创建了开源的 Linux，从此，Linux 系统不断发展，已经成为最大的服务器系统软件了。

Linus 虽然创建了 Linux，但 Linux 的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为 Linux 编写代码，那 Linux 的代码是如何管理的呢？

事实是，在 2002 年以前，世界各地的志愿者把源代码文件通过 diff 的方式发给 Linus，然后由 Linus 本人通过手工方式合并代码！

你也许会想，为什么 Linus 不把 Linux 代码放到版本控制系统里呢？不是有 CVS、SVN 这些免费的版本控制系统吗？因为 Linus 坚定地反对 CVS 和 SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比 CVS、SVN 好用，但那是付费的，和 Linux 的开源精神不符。

不过，到了 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，社区的弟兄们也对这种方式表达了强烈不满，于是 Linus 选择了一个商业的版本控制系统 BitKeeper，BitKeeper 的东家 BitMover 公司出于人道主义精神，授权 Linux 社区免费使用这个版本控制系统。

安定团结的大好局面在 2005 年就被打破了，原因是 Linux 社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发 Samba 的 Andrew 试图破解 BitKeeper 的协议（这么干的其实也不只他一个），被 BitMover 公司发现了（监控工作做得不错！），于是 BitMover 公司怒了，要收回 Linux 社区的免费使用权。

Linus 可以向 BitMover 公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

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。

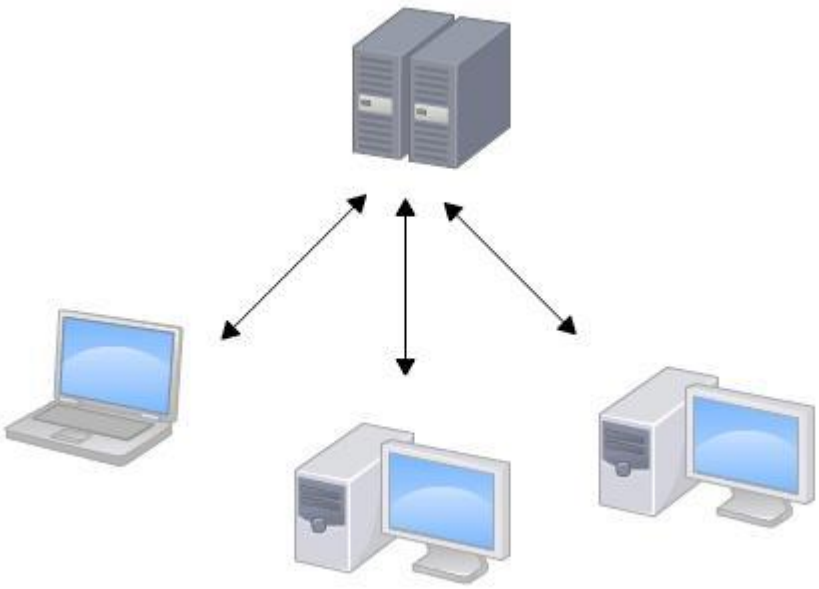
Git 迅速成为最流行的分布式版本控制系统，尤其是 2008 年，GitHub 网站上线了，它为开源项目免费提供 Git 存储，无数开源项目开始迁移至 GitHub，包括 jQuery，PHP，Ruby 等等。

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。

集中式 vs 分布式

Linus 一直痛恨的 CVS 及 SVN 都是集中式的版本控制系统，而 Git 是分布式版本控制系统，集中式和分布式版本控制系统有什么区别呢？

先说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。

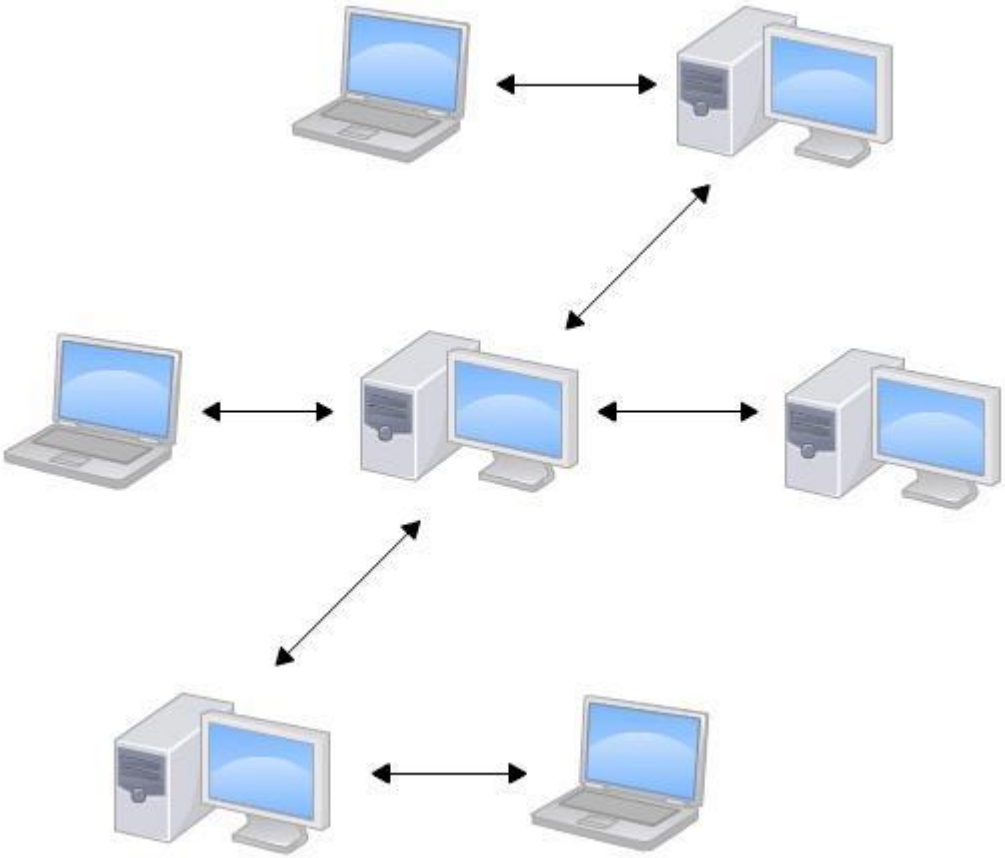


集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个 10M 的文件就需要 5 分钟，这还不得把人给憋死啊。

那分布式版本控制系统与集中式版本控制系统有何不同呢？首先，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。



当然，Git 的优势不单是不必联网这么简单，后面我们还会看到 Git 极其强大的分支管理，把 SVN 等远远抛在了后面。

CVS 作为最早的开源而且免费的集中式版本控制系统，直到现在还有不少人在用。由于 CVS 自身设计的问题，会造成提交文件不完整，版本库莫名其妙损坏的情况。同样是开源而且免费的 SVN 修正了 CVS 的一些稳定性问题，是目前用得最多的集中式版本库控制系统。

除了免费的外，还有收费的集中式版本控制系统，比如 IBM 的 ClearCase（以前是 Rational 公司的，被 IBM 收购了），特点是安装比 Windows 还大，运行比蜗牛还慢，能用 ClearCase 的一般是世界 500 强，他们有个共同的特点是财大气粗，或者人傻钱多。

微软自己也有一个集中式版本控制系统叫 VSS，集成在 Visual Studio 中。由于其反人类的设计，连微软自己都不好意思用了。

分布式版本控制系统除了 Git 以及促使 Git 诞生的 BitKeeper 外，还有类似 Git 的 Mercurial 和 Bazaar 等。这些分布式版本控制系统各有特点，但最快、最简单也最流行的依然是 Git！

安装 Git

最早 Git 是在 Linux 上开发的，很长一段时间内，Git 也只能在 Linux 和 Unix 系统上跑。不过，慢慢地有人把它移植到了 Windows 上。现在，Git 可以在 Linux、Unix、Mac 和 Windows 这几大平台上正常运行了。

要使用 Git，第一步当然是安装 Git 了。根据你当前使用的平台来阅读下面的文字：

在 Linux 上安装 Git

首先，你可以试着输入 `git`，看看系统有没有安装 Git：

```
$ git

The program 'git' is currently not installed. You can install it by typing:

sudo apt-get install git
```

像上面的命令，有很多 Linux 会友好地告诉你 Git 没有安装，还会告诉你如何安装 Git。

如果你碰巧用 Debian 或 Ubuntu Linux，通过一条 `sudo apt-get install git` 就可以直接完成 Git 的安装，非常简单。

老一点的 Debian 或 Ubuntu Linux，要把命令改为 `sudo apt-get install git-core`，因为以前有个软件也叫 GIT（GNU Interactive Tools），结果 Git 就只能叫 `git-core` 了。由于 Git 名气实在太大，后来就把 GNU Interactive Tools 改成 `gnuit`，`git-core` 正式改为 `git`。

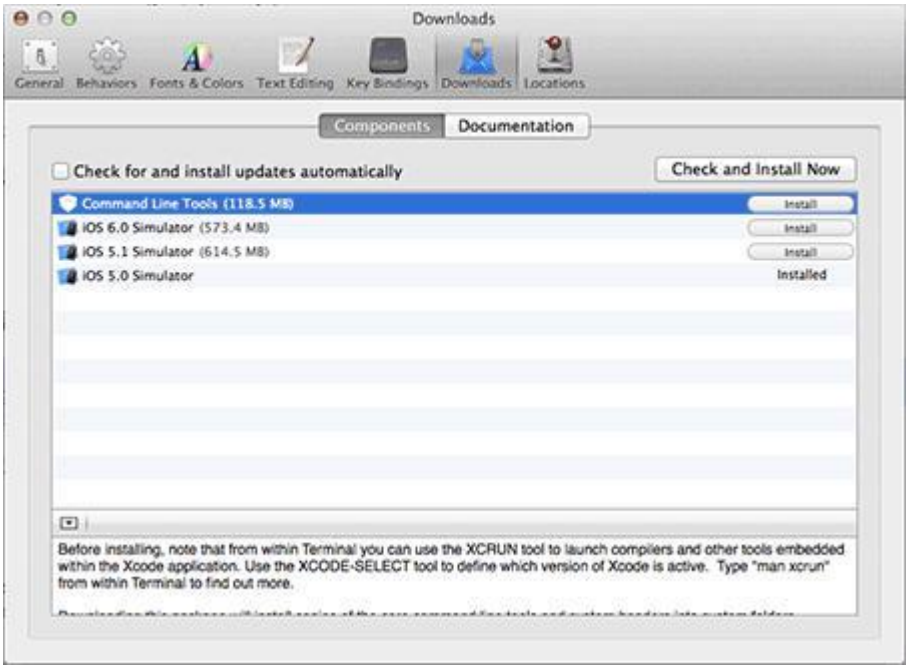
如果是其他 Linux 版本，可以直接通过源码安装。先从 Git 官网下载源码，然后解压，依次输入：`./config`，`make`，`sudo make install` 这几个命令安装就好了。

在 Mac OS X 上安装 Git

如果你正在使用 Mac 做开发，有两种安装 Git 的方法。

一是安装 homebrew，然后通过 homebrew 安装 Git，具体方法请参考 homebrew 的文档：http://brew.sh/。

第二种方法更简单，也是推荐的方法，就是直接从 AppStore 安装 Xcode，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。

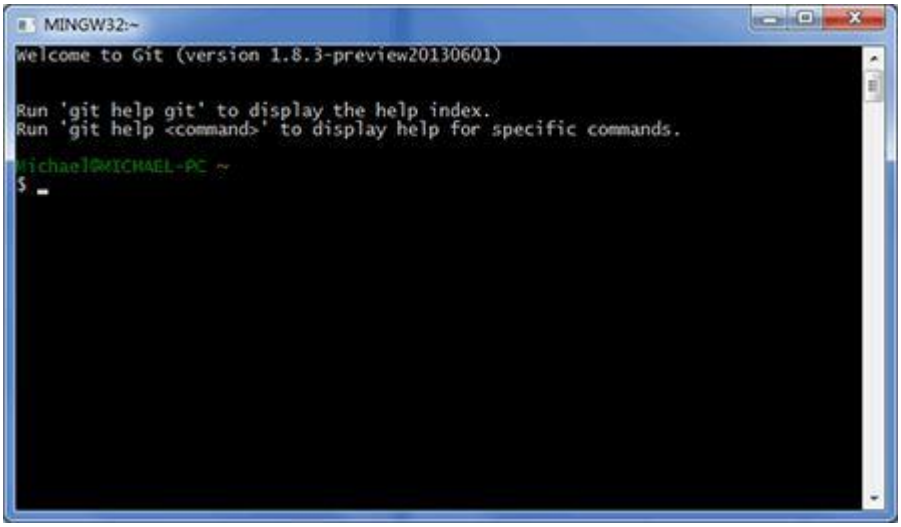


Xcode 是 Apple 官方 IDE，功能非常强大，是开发 Mac 和 iOS App 的必选装备，而且是免费的！

在 Windows 上安装 Git

在 Windows 上使用 Git，可以从 Git 官网直接下载安装程序，（网速慢的同学请移步国内镜像），然后按默认选项安装即可。

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明 Git 安装成功！



安装完成后，还需要最后一步设置，在命令行输入：

```
$ 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。

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”。

所以，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

```
$ mkdir learngit

$ cd learngit

$ pwd

/Users/michael/learngit
```

`pwd` 命令用于显示当前目录。在我的 Mac 上，这个仓库位于 `/Users/michael/learngit`。

如果你使用 Windows 系统，为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。

第二步，通过 `git init` 命令把这个目录变成 Git 可以管理的仓库：

```
$ git init

Initialized empty Git repository in /Users/michael/learngit/.git/
```

瞬间 Git 就把仓库建好了，而且告诉你是一个空的仓库（empty Git repository），细心的读者可以发现当前目录下多了一个 `.git` 的目录，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。

也不一定必须在空目录下创建 Git 仓库，选择一个已经有东西的目录也是可以的。不过，不建议你使用自己正在开发的公司项目来学习 Git，否则造成的一切后果概不负责。

把文件添加到版本库

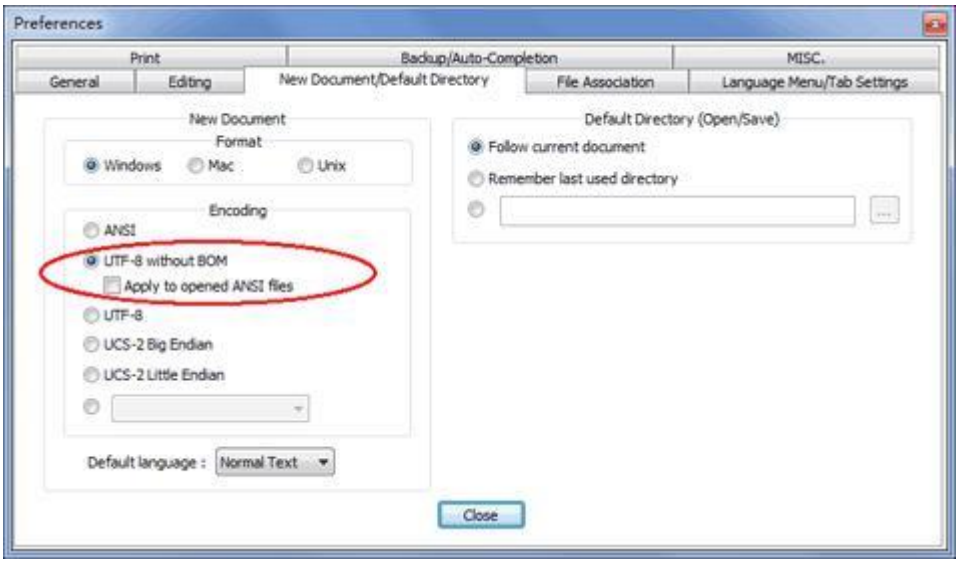
首先这里再明确一下，所有的版本控制系统，其实只能跟踪文本文件的改动，比如 TXT 文件，网页，所有的程序代码等等，Git 也不例外。版本控制系统可以告诉你每次的改动，比如在第 5 行加了一个单词“Linux”，在第 8 行删了一个单词“Windows”。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从 100KB 改成了 120KB，但到底改了啥，版本控制系统不知道，也没法知道。

不幸的是，Microsoft 的 Word 格式是二进制格式，因此，版本控制系统是没法跟踪 Word 文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，就要以纯文本方式编写文件。

因为文本是有编码的，比如中文有常用的 GBK 编码，日文有 Shift_JIS 编码，如果没有历史遗留问题，强烈建议使用标准的 UTF-8 编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

使用 Windows 的童鞋要特别注意：

千万不要使用 Windows 自带的**记事本**编辑任何文本文件。原因是 Microsoft 开发记事本的团队使用了一个非常弱智的行为来保存 UTF-8 编码的文件，他们自作聪明地在每个文件开头添加了 0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载 Notepad++代替记事本，不但功能强大，而且免费！记得把 Notepad++的默认编码设置为 UTF-8 without BOM 即可：



言归正传，现在我们编写一个 `readme.txt` 文件，内容如下：

```
Git is a version control system.

Git is free software.
```

一定要放到 `learngit` 目录下（子目录也行），因为这是一个 Git 仓库，放到其他地方 Git 再厉害也找不到这个文件。

和把大象放到冰箱需要 3 步相比，把一个文件放到 Git 仓库只需要两步。

第一步，用命令 `git add` 告诉 Git，把文件添加到仓库：

```
$ git add readme.txt
```

执行上面的命令，没有任何显示，这就对了，Unix 的哲学是“没有消息就是好消息”，说明添加成功。

第二步，用命令 `git commit` 告诉 Git，把文件提交到仓库：

```
$ git commit -m "wrote a readme file"
[master (root-commit) cb926e7] wrote a readme file
 1 file changed, 2 insertions(+)
 create mode 100644 readme.txt
```

简单解释一下 `git commit` 命令，`-m` 后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

嫌麻烦不想输入 `-m "xxx"` 行不行？确实有办法可以这么干，但是强烈不建议你这么干，因为输入说明对自己对别人阅读都很重要。实在不想输入说明的童鞋请自行 Google，我不告诉你这个参数。

`git commit` 命令执行成功后会告诉你，1 个文件被改动（我们新添加的 readme.txt 文件），插入了两行内容（readme.txt 有两行内容）。

为什么 Git 添加文件需要 `add`，`commit` 一共两步呢？因为 `commit` 可以一次提交很多文件，所以你可以多次 `add` 不同的文件，比如：

```
$ git add file1.txt

$ git add file2.txt file3.txt

$ git commit -m "add 3 files."
```

小结

现在总结一下今天学的两点内容：

初始化一个 Git 仓库，使用 `git init` 命令。

添加文件到 Git 仓库，分两步：

- 第一步，使用命令 `git add <file>`，注意，可反复多次使用，添加多个文件；
- 第二步，使用命令 `git commit`，完成。

时光机穿梭

我们已经成功地添加并提交了一个 readme.txt 文件，现在，是时候继续工作了，于是，我们继续修改 readme.txt 文件，改成如下内容：

```
Git is a distributed version control system.

Git is free software.
```

现在，运行 `git status` 命令看看结果：

```
$ git status

# On branch master

# 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:   readme.txt

#

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

`git status` 命令可以让我们时刻掌握仓库当前的状态，上面的命令告诉我们，readme.txt 被修改过了，但还没有准备提交的修改。

虽然 Git 告诉我们 readme.txt 被修改了，但如果能看看具体修改了什么内容，自然是很好的。比如你休假两周从国外回来，第一天上班时，已经记不清上次怎么修改的 readme.txt，所以，需要用 `git diff` 这个命令看看：

```
$ git diff readme.txt

diff --git a/readme.txt b/readme.txt

index 46d49bf..9247db6 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,2 +1,2 @@

-Git is a version control system.

+Git is a distributed version control system.

 Git is free software.
```

`git diff` 顾名思义就是查看 difference，显示的格式正是 Unix 通用的 diff 格式，可以从上面的命令输出看到，我们在第一行添加了一个“distributed”单词。

知道了对 readme.txt 作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是 `git add`：

```
$ git add readme.txt
```

同样没有任何输出。在执行第二步 `git commit` 之前，我们再运行 `git status` 看看当前仓库的状态：

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       modified:   readme.txt

#
```

`git status` 告诉我们，将要被提交的修改包括 readme.txt，下一步，就可以放心地提交了：

```
$ git commit -m "add distributed"

[master ea34578] add distributed

 1 file changed, 1 insertion(+), 1 deletion(-)
```

提交后，我们再用`git status`命令看看仓库的当前状态：

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Git 告诉我们当前没有需要提交的修改，而且，工作目录是干净（working directory clean）的。

小结

- 要随时掌握工作区的状态，使用`git status`命令。
- 如果`git status`告诉你有文件被修改过，用`git diff`可以查看修改内容。

版本回退

现在，你已经学会了修改文件，然后把修改提交到 Git 版本库，现在，再练习一次，修改 readme.txt 文件如下：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
```

然后尝试提交：

```
$ git add readme.txt
$ git commit -m "append GPL"
[master 3628164] append GPL
 1 file changed, 1 insertion(+), 1 deletion(-)
```

像这样，你不断对文件进行修改，然后不断提交修改到版本库里，就好比玩 RPG 游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打 Boss 之前，你会手动存盘，以便万一打 Boss 失败了，可以从最近的地方重新开始。Git 也是一样，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在 Git 中被称为`commit`。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个`commit`恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

现在，我们回顾一下 readme.txt 文件一共有几个版本被提交到 Git 仓库里了：

版本 1：wrote a readme file

```
Git is a version control system.
Git is free software.
```

版本 2：add distributed

```
Git is a distributed version control system.

Git is free software.
```

版本 3：append GPL

```
Git is a distributed version control system.

Git is free software distributed under the GPL.
```

当然了，在实际工作中，我们脑子里怎么可能记得一个几千行的文件每次都改了什么内容，不然要版本控制系统干什么。版本控制系统肯定有某个命令可以告诉我们历史记录，在 Git 中，我们用 `git log` 命令查看：

```
$ git log

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Tue Aug 20 15:11:49 2013 +0800

    append GPL

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Tue Aug 20 14:53:12 2013 +0800

    add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Mon Aug 19 17:51:55 2013 +0800

    wrote a readme file
```

`git log` 命令显示从最近到最远的提交日志，我们可以看到 3 次提交，最近的一次是 `append GPL`，上一次是 `add distributed`，最早的一次是 `wrote a readme file`。 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：

```
$ git log --pretty=oneline

3628164fb26d48395383f8f31179f24e0882e1e0 append GPL

ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed

cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file
```

需要友情提示的是，你看到的一大串类似 `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，而且你看到的 `commit id` 和我的肯定不一样，以你自己的为准。为什么 `commit id` 需要用这么一大串数字表示呢？因为 Git 是分布式的版本控制系统，后面我们还要研究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。

每提交一个新版本，实际上 Git 就会把它们自动串成一条时间线。如果使用可视化工具查看 Git 历史，就可以更清楚地看到提交历史的时间线：

好了，现在我们启动时光穿梭机，准备把 readme.txt 回退到上一个版本，也就是“add distributed”的那个版本，怎么做呢？

首先，Git 必须知道当前版本是哪个版本，在 Git 中，用 `HEAD` 表示当前版本，也就是最新的提交 `3628164...882e1e0`（注意我的提交 ID 和你的肯定不一样），上一个版本就是 `HEAD^`，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`。

现在，我们要把当前版本“append GPL”回退到上一个版本“add distributed”，就可以使用 `git reset` 命令：

```
$ git reset --hard HEAD^

HEAD is now at ea34578 add distributed
```

`--hard` 参数有啥意义？这个后面再讲，现在你先放心使用。

看看 readme.txt 的内容是不是版本 `add distributed`：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software.
```

果然。

还可以继续回退到上一个版本 `wrote a readme file`，不过且慢，然我们用 `git log` 再看看现在版本库的状态：

```
$ git log

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Tue Aug 20 14:53:12 2013 +0800


    add distributed


commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date:   Mon Aug 19 17:51:55 2013 +0800


    wrote a readme file
```

最新的那个版本 `append GPL` 已经看不到了！好比你从 21 世纪坐时光穿梭机来到了 19 世纪，想再回去已经回不去了，肿么办？

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到那个 `append GPL` 的 `commit id` 是 `3628164...`，于是就可以指定回到未来的某个版本：

```
$ git reset --hard 3628164

HEAD is now at 3628164 append GPL
```

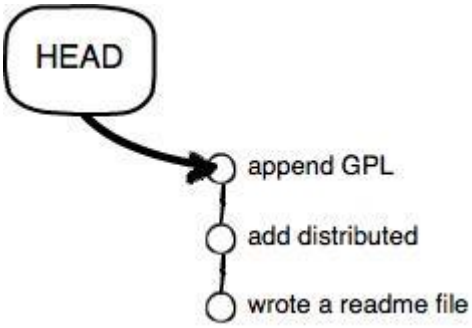
版本号没必要写全，前几位就可以了，Git 会自动去找。当然也不能只写前一两位，因为 Git 可能会找到多个版本号，就无法确定是哪一个了。

再小心翼翼地看看 readme.txt 的内容：

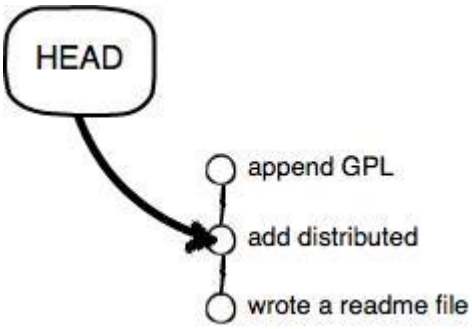
```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.
```

果然，我胡汉三又回来了。

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`：



然后顺便把工作区的文件更新了。所以你让 `HEAD` 指向哪个版本号，你就把当前版本定位在哪。

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的 `commit id` 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

ea34578 HEAD@{0}: reset: moving to HEAD^

3628164 HEAD@{1}: commit: append GPL

ea34578 HEAD@{2}: commit: add distributed

cb926e7 HEAD@{3}: commit (initial): wrote a readme file
```

终于舒了口气，第二行显示 `append GPL` 的 commit id 是 `3628164`，现在，你又可以乘坐时光机回到未来了。

小结

现在总结一下：

- `HEAD` 指向的版本就是当前版本，因此，Git 允许我们在版本的历史之间穿梭，使用命令 `git reset --hard commit_id`。
- 穿梭前，用 `git log` 可以查看提交历史，以便确定要回退到哪个版本。
- 要重返未来，用 `git reflog` 查看命令历史，以便确定要回到未来的哪个版本。

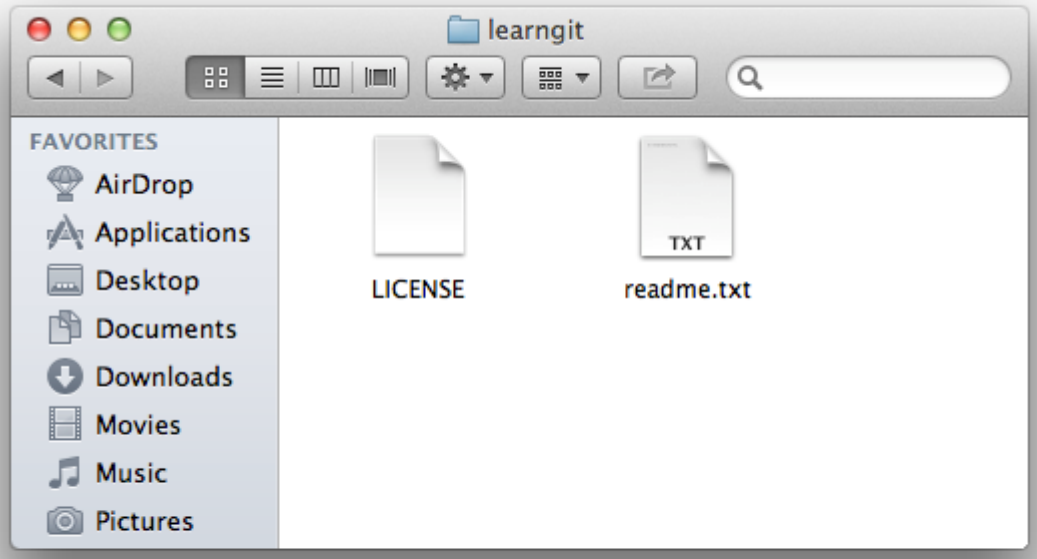
工作区和暂存区

Git 和其他版本控制系统如 SVN 的一个不同之处就是有暂存区的概念。

先来看名词解释。

工作区（Working Directory）

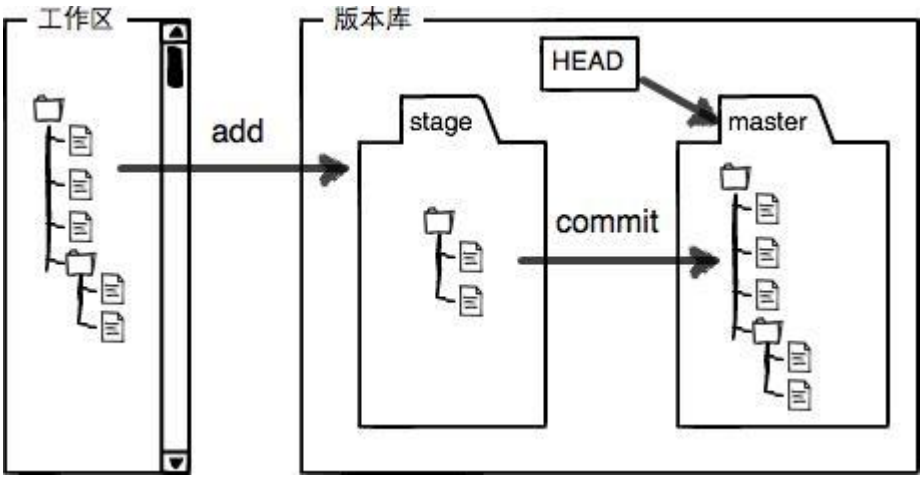
就是你在电脑里能看到的目录，比如我的 `learngit` 文件夹就是一个工作区：



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`。



分支和 `HEAD` 的概念我们以后再讲。

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。

俗话说，实践出真知。现在，我们再练习一遍，先对 `readme.txt` 做个修改，比如加上一行内容：

```
Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.
```

然后，在工作区新增一个 `LICENSE` 文本文件（内容随便写）。

先用 `git status` 查看一下状态：

```
$ git status
# On branch master
# 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:   readme.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       LICENSE
no changes added to commit (use "git add" and/or "git commit -a")
```

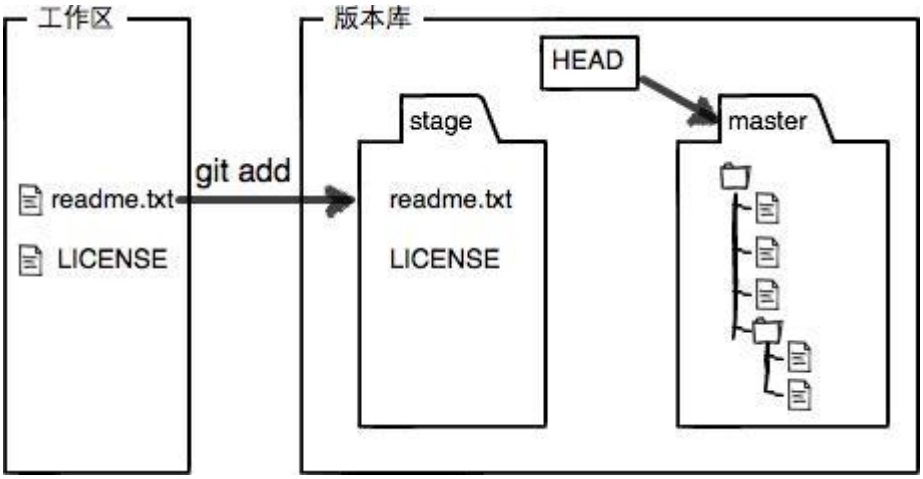
Git 非常清楚地告诉我们，`readme.txt` 被修改了，而 `LICENSE` 还从来没有被添加过，所以它的状态是 `Untracked`。

现在，使用两次命令 `git add`，把 `readme.txt` 和 `LICENSE` 都添加后，用 `git status` 再查看一下：

```
$ git status
# On branch master
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   LICENSE
#       modified:   readme.txt
#
```

现在，暂存区的状态就变成这样了：



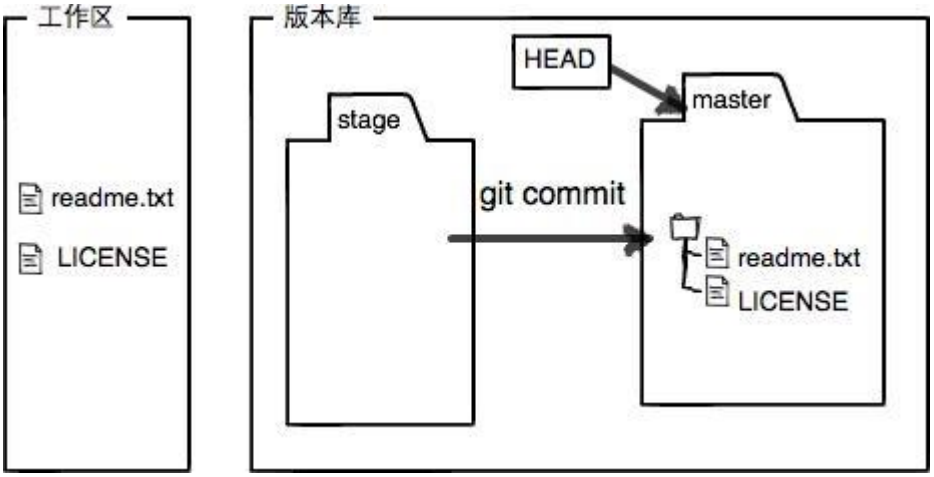
所以，`git add`命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行`git commit`就可以一次性把暂存区的所有修改提交到分支。

```
$ git commit -m "understand how stage works"
[master 27c9860] understand how stage works
 2 files changed, 675 insertions(+)
 create mode 100644 LICENSE
```

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

现在版本库变成了这样，暂存区就没有任何内容了：



小结

暂存区是 Git 非常重要的概念，弄明白了暂存区，就弄明白了 Git 的很多操作到底干了什么。

管理修改

现在，假定你已经完全掌握了暂存区的概念。下面，我们要讨论的就是，为什么 Git 比其他版本控制系统设计得优秀，因为 Git 跟踪并管理的是修改，而非文件。

你会问，什么是修改？比如你新增了一行，这就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

为什么说 Git 管理的是修改，而不是文件呢？我们还是做实验。第一步，对 readme.txt 做一个修改，比如加一行内容：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes.
```

然后，添加：

```
$ git add readme.txt

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       modified:   readme.txt
```

```
#
```

然后，再修改 readme.txt：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
```

提交：

```
$ git commit -m "git tracks changes"
[master d4f25b6] git tracks changes
 1 file changed, 1 insertion(+)
```

提交后，再看看状态：

```
$ git status
# On branch master
# 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:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

咦，怎么第二次的修改没有被提交？

别激动，我们回顾一下操作过程：

第一次修改 -> `git add` -> 第二次修改 -> `git commit`

你看，我们前面讲了，Git 管理的是修改，当你用 `git add` 命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，`git commit` 只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

提交后，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.txt

diff --git a/readme.txt b/readme.txt

index 76d770f..a9c5755 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,4 +1,4 @@

 Git is a distributed version control system.

 Git is free software distributed under the GPL.

 Git has a mutable index called stage.

-Git tracks changes.

+Git tracks changes of files.
```

可见，第二次修改确实没有被提交。

那怎么提交第二次修改呢？你可以继续 `git add` 再 `git commit`，也可以别着急提交第一次修改，先 `git add` 第二次修改，再 `git commit`，就相当于把两次修改合并后一块提交了：

第一次修改 -> `git add` -> 第二次修改 -> `git add` -> `git commit`

好，现在，把第二次修改提交了，然后开始小结。

小结

现在，你又理解了 Git 是如何跟踪修改的，每次修改，如果不 `add` 到暂存区，那就不会加入到 `commit` 中。

撤销修改

自然，你是不会犯错的。不过现在是凌晨两点，你正在赶一份工作报告，你在 `readme.txt` 中添加了一行：

```
$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.
```

在你准备提交前，一杯咖啡起了作用，你猛然发现了“stupid boss”可能会让你丢掉这个月的奖金！

既然错误发现得很及时，就可以很容易地纠正它。你可以删掉最后一行，手动把文件恢复到上一个版本的状态。如果用 `git status` 查看一下：

```
$ git status
# On branch master
# 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:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

你可以发现，Git 会告诉你，`git checkout -- file` 可以丢弃工作区的修改：

```
$ git checkout -- readme.txt
```

命令 `git checkout -- readme.txt` 意思就是，把 `readme.txt` 文件在工作区的修改全部撤销，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。

现在，看看 `readme.txt` 的文件内容：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
```

文件内容果然复原了。

`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，我们在后面的分支管理中会再次遇到 `git checkout` 命令。

现在假定是凌晨 3 点，你不但写了一些胡话，还 `git add` 到暂存区了：

```
$ cat readme.txt
```

```
Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.


$ git add readme.txt
```

庆幸的是，在 `commit` 之前，你发现了这个问题。用 `git status` 查看一下，修改只是添加到了暂存区，还没有提交：

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

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.txt
Unstaged changes after reset:
M       readme.txt
```

`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。

再用 `git status` 查看一下，现在暂存区是干净的，工作区有修改：

```
$ git status
# On branch master
# 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:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

还记得如何丢弃工作区的修改吗？

```
$ git checkout -- readme.txt


$ git status
# On branch master
nothing to commit (working directory clean)
```

整个世界终于清静了！

现在，假设你不但改错了东西，还从暂存区提交到了版本库，怎么办呢？还记得版本回退一节吗？可以回退到上一个版本。不过，这是有条件的，就是你还没有把自己的本地版本库推送到远程。还记得 Git 是分布式版本控制系统吗？我们后面会讲到远程版本库，一旦你把“stupid boss”提交推送到远程版本库，你就真的惨了……

小结

又到了小结时间。

场景 1：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 checkout -- file`。

场景 2：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 reset HEAD file`，就回到了场景 1，第二步按场景 1 操作。

场景 3：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。

删除文件

在 Git 中，删除也是一个修改操作，我们实战一下，先添加一个新文件 test.txt 到 Git 并且提交：

```
$ git add test.txt
$ git commit -m "add test.txt"
[master 94cdc44] add test.txt
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
```

一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用 `rm` 命令删了：

```
$ rm test.txt
```

这个时候，Git 知道你删除了文件，因此，工作区和版本库就不一致了，`git status`命令会立刻告诉你哪些文件被删除了：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    test.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令`git rm`删掉，并且`git commit`：

```
$ git rm test.txt
rm 'test.txt'
$ git commit -m "remove test.txt"
[master d17efd8] remove test.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 test.txt
```

现在，文件就从版本库中被删除了。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

```
$ git checkout -- test.txt
```

`git checkout`其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

小结

命令`git rm`用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失**最近一次提交后你修改的内容**。