

专业的软件版本控制系统-GIT

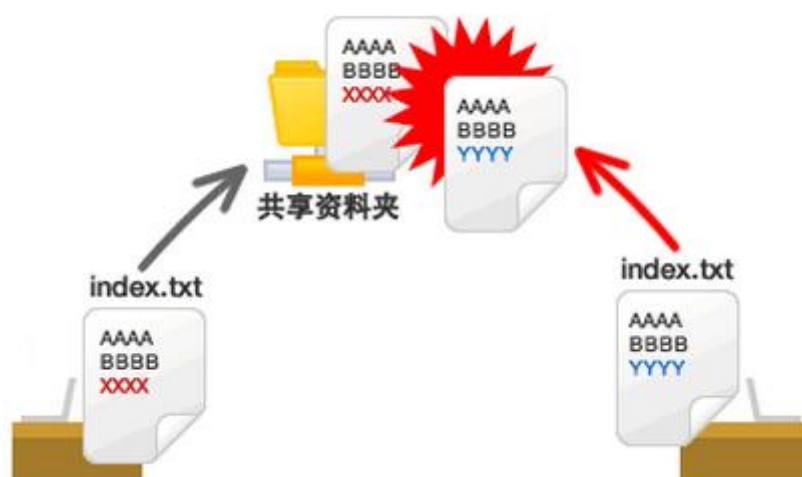
GIT 的诞生

在我们开发项目的时候，经常需要把代码，文档还原到之前的某个时间点，大家都是怎么做的呢？

最传统的办法，就是备份文档名或者在目录上添加编辑的日期，比如 Code1_NAME_20150428。但是，每次编辑文档都要先复制，这样非常麻烦，也容易出错，时间一长，自己都不知道想找回哪天的备份文件了。

Name
120525_文档_更新.txt
120604_文档.txt
120605_文档_monkey.txt
120605_文档_最新.txt
120605_文档_最新复制.txt
120605_文档_修改版.txt
120605_文档.txt
1200602_文档.txt
文档_会议用.txt

更加严重的是，如果备份的文件命名风格松散，就更无法区分哪个文档是最新的了，如果是合作项目，还需要加上编辑者的名字。就更麻烦了。另外，如果两个人同时编辑某个共享文件，先进行编辑的人所做的修改内容会被覆盖，造成文档内容冲突，相信大家都有这样惨痛的经历。



GIT 版本管理系统就是为了解决这些问题而应运而生的。

GIT 是一个分布式版本管理系统，是为了更好地管理 Linux 内核开发而创立的。

GIT 可以在任何时间点，把文档的状态作为一个时间点上的记录保存起来。因此可以把编辑过的文档复原到以前的状态，也可以显示编辑前后的内容差异。而且，当您编辑本地文件后，试图覆盖较新的文件的时候（即上传文件到服务器时），系统会发出警告，因此可以避免在无意中覆盖了他人的编辑内容。



用 GIT 管理文件，每一次更新都会保存在 GIT 中，所有的文档，代码可以在任意时刻切换回任意时间点，所以不需要备份文件，非常方便。

GIT 的诞生

以下内容摘自维基百科-GIT:

很多人都知道，Linux 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。

Linux 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 Linux，然后由 Linux 本人通过手工方式合并代码！

你也许会想，为什么 Linux 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 Linux 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 Linux 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 Linux 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。

Linux 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：

Linux 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 的核心 - 数据库

数据库是记录文件或者目录状态的地方，储存着内容修改的历史记录，在数据库的管理下，把文件和目录修改的历史记录放在对应目录下：



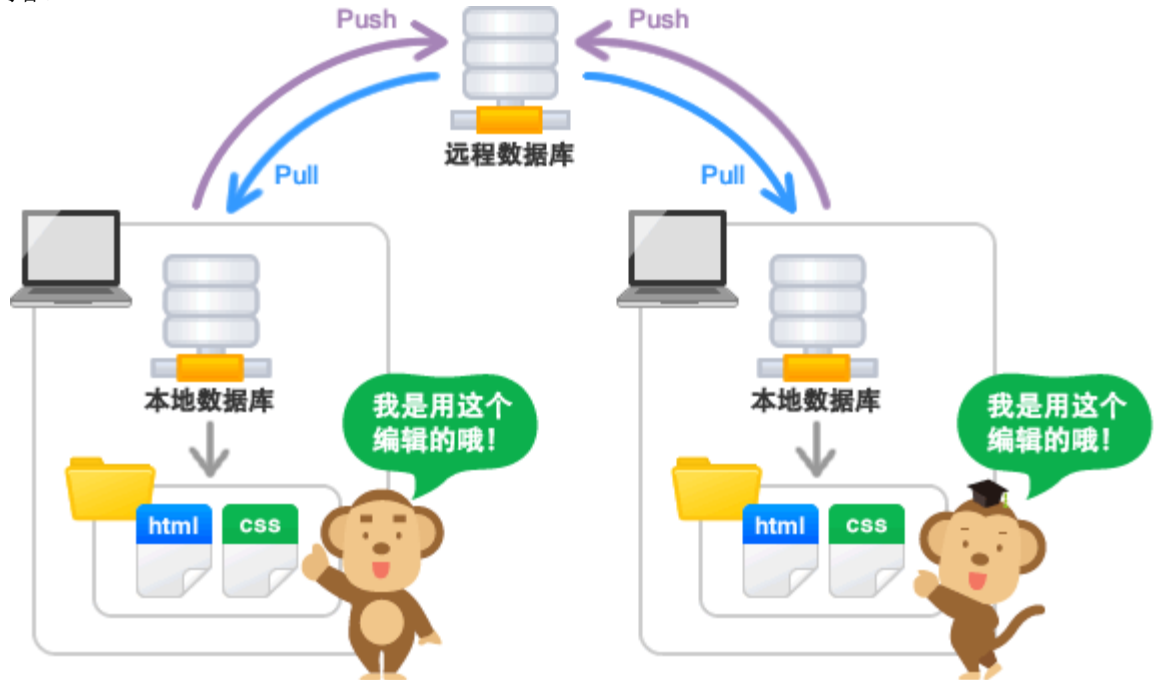
Tips: 放心，学习 GIT 并不需要你对数据库有多么深的了解，这里只不过强调，GIT 是依赖于数据库来管理你的所有文档的。

远程数据库和本地数据库：

首先，GIT 的数据库分为远程数据库和本地数据库的两种。

- 远程数据库：配有专用的服务器，为了多人共享而建立的数据库。
- 本地数据库，为了方便个人使用，在自己机器上配置的数据库。

数据库分为远程和本地两种。平时用手头上的机器在本地数据库上操作就可以了。如果想要公开在本地数据库中修改的内容，把内容上传到远程数据库就可以了。另外，通过远程数据库还可以取得其他人修改的内容。



Tips: GIT 并不存在所谓“主数据库”的概念，每一个数据库，不管是远程数据库还是本地数据库，都是平等的

Tips: GIT 这种本地远程两个版本的数据架构有诸多好处，比如，一些操作在本地数据库就可以完成，完全可以在断网的时候执行，再需要和远程数据库同步的时候再一次行的提交。这样就节省了很多的网络带宽。

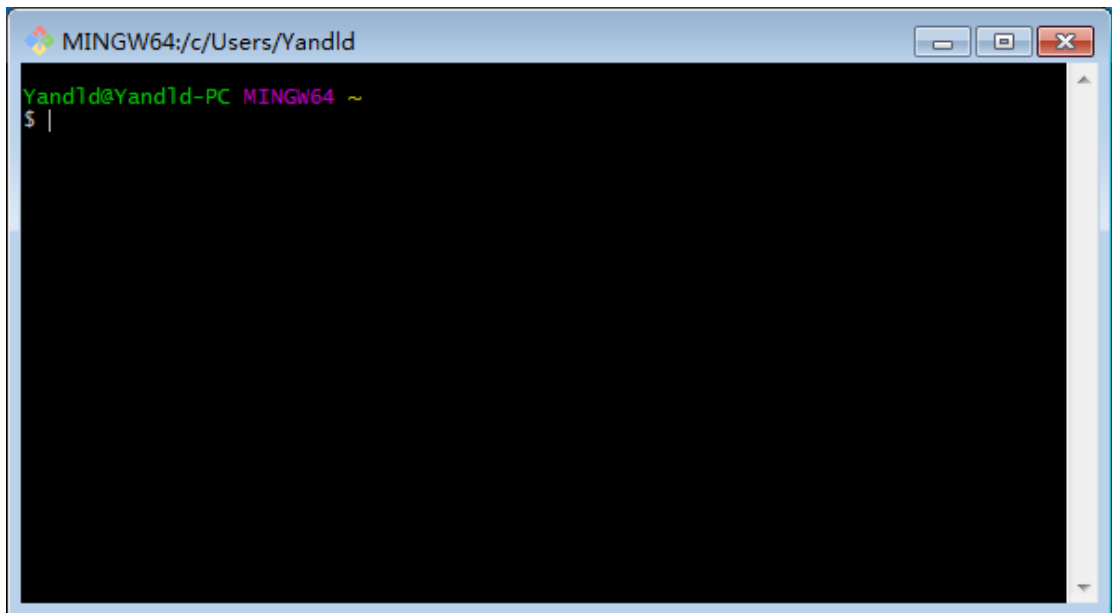
GIT 入门笔记

下载安装 GIT

本节介绍如何在 Windows 平台上下载，安装 GIT 软件

首先进入 GIT 的 下载页面：<http://GIT-scm.com/download/>

下载适合您计算机操作系统的版本。GIT 是开源软件，没有版权之类的困扰，安装过程就像普通的软件一样，没有什么可值得可圈可点的，安装完成后，在开始菜单里找到“GIT”->“GIT BASH”，弹出一个类似命令行窗口的东西，就说明 GIT 软件已经安装成功。



Tips: GIT BASH 是一个在 windows 上模拟 Linux bash 的命令行界面，已经安装了最常用的一些基本命令，诸如 ls , cd rm cp mv 等等。可以提供文件的增删改查等常用功能。如果您对这些基本的命令不是很了解，可以从这里获取帮助：

<http://blog.csdn.net/ljianhui/article/details/11100625>

Tips: GIT 有两种交互方式，一种是 GUI 图形方式，一种是命令行方式，笔者建议使用命令行方式。GUI 方式虽然略显直观，操作简单，但是并不利于学习，而命令行方式更加直接，对于学习效果也更显著。软件安装成功年后，需要进行几个简单的配置：

```
GIT config --global user.name "your name"
```

```
GIT config --global user.email "email@example.com"
```

Tips: 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。

初始化代码仓库

什么是代码仓库呢？代码仓库的英文叫做 repository，你可以简单理解成一个普通的文件夹，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，以便任何时候都可以追踪历史，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：

第一步：首先，选择一个合适的地方，创建一个空目录：

```
MINGW64:/c/Users/Yandld/Desktop/MyFirstResp
Yandld@Yandld-PC MINGW64 ~/Desktop
$
Yandld@Yandld-PC MINGW64 ~/Desktop
$
Yandld@Yandld-PC MINGW64 ~/Desktop
$
Yandld@Yandld-PC MINGW64 ~/Desktop
$
Yandld@Yandld-PC MINGW64 ~/Desktop
$ pwd
/c/Users/Yandld/Desktop
Yandld@Yandld-PC MINGW64 ~/Desktop
$ mkdir MyFirstResp
Yandld@Yandld-PC MINGW64 ~/Desktop
$ cd MyFirstResp/
Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp
$
```

第二步:

进入刚才的目录, 输入 `GIT init` 挖成命令把这个目录变成 GIT 可以管理的仓库:

```
MINGW64:/c/Users/Yandld/Desktop/MyFirstResp
$
Yandld@Yandld-PC MINGW64 ~/Desktop
$
Yandld@Yandld-PC MINGW64 ~/Desktop
$
Yandld@Yandld-PC MINGW64 ~/Desktop
$ pwd
/c/Users/Yandld/Desktop
Yandld@Yandld-PC MINGW64 ~/Desktop
$ mkdir MyFirstResp
Yandld@Yandld-PC MINGW64 ~/Desktop
$ cd MyFirstResp/
Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp
$ git init
Initialized empty Git repository in C:/Users/Yandld/Desktop/MyFirstResp/.git/
Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$
```

分分钟的功夫 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。

Tips: 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 -al` 命令就可以看见。

Tips: 目录创建好后, 在目录的右边, 会出现 (master) 表示目前这个目录已经是 GIT 管理的仓库, 并且当前分支是 master。

添加文件并提交修改

所有的版本控制系统, 只能跟踪, 记录文本文件的改动, 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。

下面我们编写一个非常简单的文本文件



Tips: 所有需要 GIT 管理的文本文件都要放到刚刚的代码仓库目录下。否则 GIT 无法识别这些文件。

下面我们来介绍一个最重要也最常用的命令: GIT status 他可以用来列出当前 GIT 所检测到任何在版本仓库中的改动, 包括改动的提示, 改动了哪些文件。等等

The image shows a Windows command prompt window titled 'MINGW64:/c/Users/Yandld/Desktop/MyFirstResp'. The prompt shows the user 'Yandld@Yandld-PC' in the 'MINGW64' environment, located at '~/Desktop/MyFirstResp' on the 'master' branch. The user has executed the following commands and received the following output:

```
Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ vim test.txt

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ cat test.txt
Git is a version control system.
Git is free software.

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test.txt

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

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$
```

这里显示的 untracked files 就是指 GIT 探测到, 但是并没有提交到 GIT 数据库暂存的的文件, 这类文件用红色标出, 如果我们这时候把这个文件改动或者删除, GIT 也没有办法恢复他们。

若要把文件或目录的添加和变更保存到数据库, 就需要进行提交。

执行提交后, 数据库中会生成上次提交的状态与当前状态的差异记录 (也被称为 revision)。

如下图, 提交是以时间顺序排列状态被保存到数据库中的。凭借该提交和最新的文件状态, 就可以知道过去的修改记录以及内容。



Tips:看其他人提交的修改内容或自己的历史记录的时候，提交信息是需要用到的重要资料。GIT 使用一下格式来统一标准注解：

第一行：提交修改内容的摘要

第二行：空行

第三行：修改的理由

下面我们用 `git add` 命令把更改/新添加 的文件提交到 GIT 仓库

```
MINGW64:/c/Users/Yandld/Desktop/MyFirstResp

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git add test.txt
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

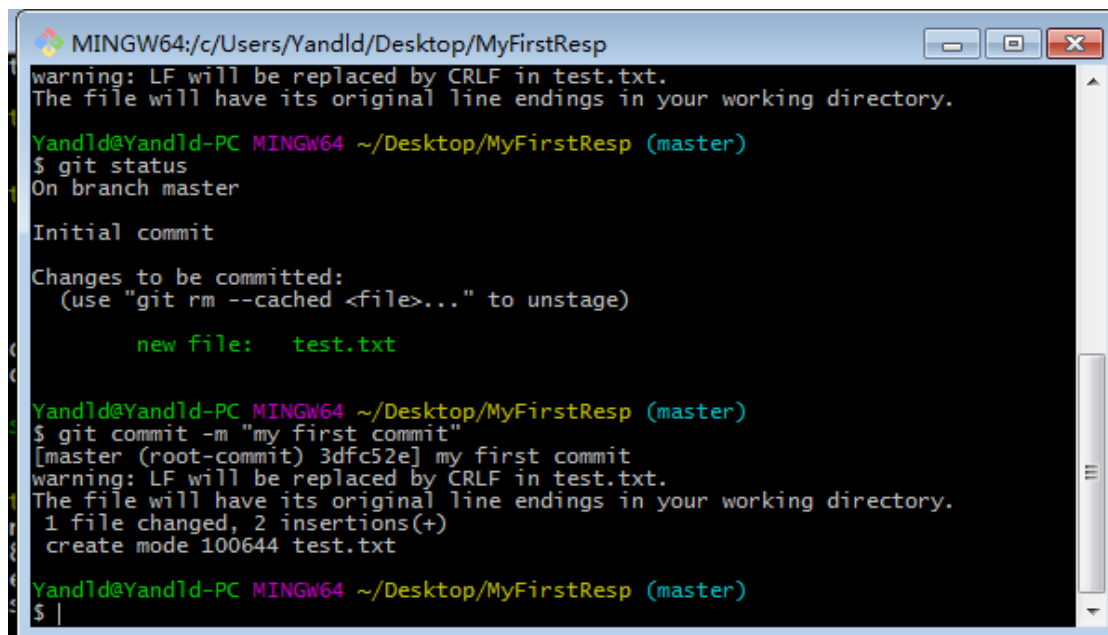
    new file:   test.txt

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ |
```

执行 `git add` 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。此时，再输入 `git status` 查看状态，可以看到 `test.txt` 已经从红色变为绿色，代表 GIT 已经成功的把新文件保存到 GIT 数据库了。

Tips: 可以使用 `git add .` 来提交所有更改的内容(空格+点 ‘.’ 代表当前目录下所有内容)

下一步，用命令 `GIT commit` 告诉 GIT，把文件提交到仓库：



```
MINGW64:/c:/Users/Yandld/Desktop/MyFirstResp
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   test.txt

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git commit -m "my first commit"
[master (root-commit) 3dfc52e] my first commit
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.
1 file changed, 2 insertions(+)
 create mode 100644 test.txt

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ |
```

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。

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 test.txt 文件），插入了两行

内容（test.txt 有两行内容）。

Tips:为什么 GIT 添加文件需要 git add, git 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"
```

并且，更重要的是，GIT 的还原是以 commit 为单位的，无论你 add 多少文件，只要不 commit, 都无法

创建一个还原点。

增删改查 GIT 仓库中的文件

我们已经成功地添加并提交了一个 test.txt 文件，现在，是时候继续工作了，于是，我们继续修改

test.txt 文件，改成如下内容：

```
GIT is a distributed version control system.
```

```
GIT is free software.
```

```
运行 GIT status:
```



```
MINGW64:/c/Users/YandId/Desktop/MyFirstResp
YandId@YandId-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git commit -m "my first commit"
[master (root-commit) 3dfc52e] my first commit
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.
1 file changed, 2 insertions(+)
create mode 100644 test.txt

YandId@YandId-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ vim test.txt

YandId@YandId-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ 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:   test.txt

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

YandId@YandId-PC MINGW64 ~/Desktop/MyFirstResp (master)
$
```

这里重申一下，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 test.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：

```
MINGW64:/c/Users/YandId/Desktop/MyFirstResp
YandId@YandId-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git diff test.txt
diff --git a/test.txt b/test.txt
index 46d49bf..9247db6 100644
--- a/test.txt
+++ b/test.txt
@@ -1,2 +1,2 @@
-Git is a version control system.
+Git is a distributed version control system.
 Git is free software.
warning: LF will be replaced by CRLF in test.txt.
The file will have its original line endings in your working directory.

YandId@YandId-PC MINGW64 ~/Desktop/MyFirstResp (master)
$
```

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 test.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，第二部是 git status，同样，初学是可以在每一步后都敲一次 git status 来查看当前的状态，这样对每个命令执行的结果都有一个清晰的了解，便于学习理解。

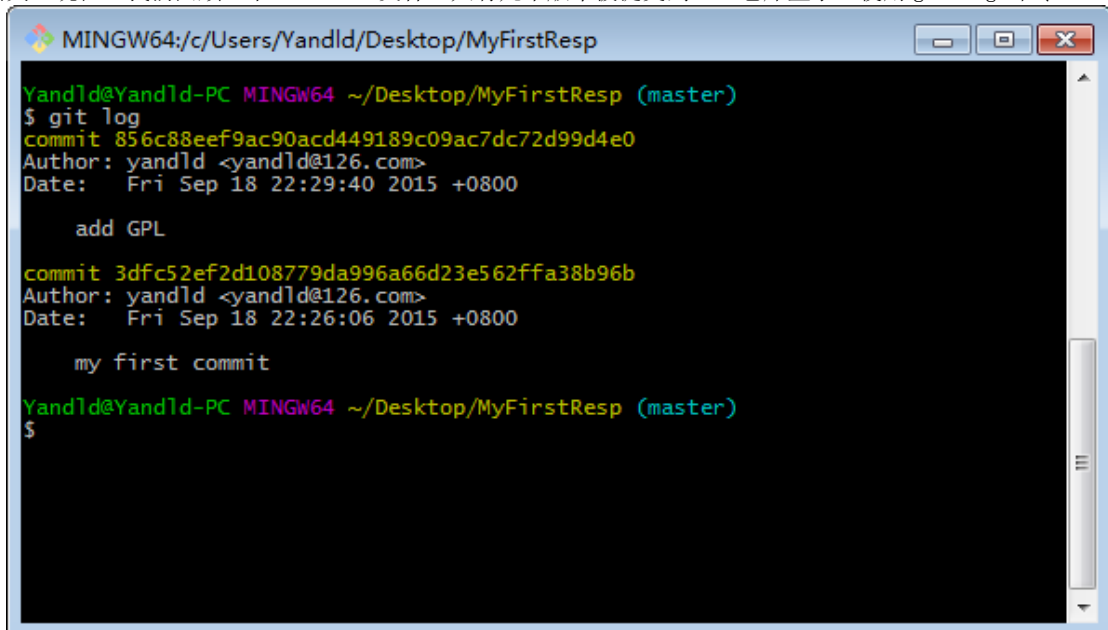
现在，你已经学会了修改文件，然后把修改提交到 GIT 版本库，现在，再练习一次，修改 test.txt 文件如下：

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

然后尝试提交，制作成 commit: git commit -m “add GPL”

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 test.txt 文件一共有几个版本被提交到 GIT 仓库里了，使用 git log 命令



```
MINGW64:/c/Users/Yandld/Desktop/MyFirstResp
Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git log
commit 856c88eef9ac90acd449189c09ac7dc72d99d4e0
Author: yandld <yandld@126.com>
Date:   Fri Sep 18 22:29:40 2015 +0800

    add GPL

commit 3dfc52ef2d108779da996a66d23e562ffa38b96b
Author: yandld <yandld@126.com>
Date:   Fri Sep 18 22:26:06 2015 +0800

    my first commit

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$
```

GIT log 命令显示从最近到最远的提交日志，我们可以看到 2 次提交，最近的一次是 my first commit，第二次是 add GPL。

Tips: 使用键盘上的 up 和 down 来翻页，q 来退出 git log 模式。

Tips:你看到的一大串类似 3628164...882e1e0 的是 commit id (版本号) 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。Commit 号，也叫哈希号，是 commit 的唯一 ID，可以认为是这个 commit 的名字。

当然，除了增加，修改文件，也可以删除文件，使用 git rm 命令，他与 git add 天生是相反的一对，当然，使用 git rm 后，也需要 git commit 来提交到 git 数据库。这里就不再详细介绍了，相信大家的自学能力都很强，可以自己结合上网的一些教程来实验 git rm 删除某个文件。

退回到某一个 commit

好了，现在我们启动时光穿梭机，准备把 test.txt 回退到上一个版本，也就是“my first commit”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。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“addGPL”回退到上一个版本“my first commit”，就可以使用 GIT reset 命令：

```
GIT reset --hard HEAD^
```

看看 test.txt 的内容是不是版本 my second commit:

```
Cat test.txt
```

```
MINGW64:/c/Users/Yandld/Desktop/MyFirstResp

add GPL

commit 3dfc52ef2d108779da996a66d23e562ffa38b96b
Author: yandld <yandld@126.com>
Date: Fri Sep 18 22:26:06 2015 +0800

my first commit

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git status
On branch master
nothing to commit, working directory clean

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git reset --hard HEAD^
HEAD is now at 3dfc52e my first commit

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ cat test.txt
Git is a version control system.
Git is free software.

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$
```

此时也可以使用 GIT log 来看下此时的 log 信息，确认当前的操作是否正确。

最新的那个版本 add 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 add GPL 的 commit id 是 9a92c9489d12c4e3d627d22c3f6e7b5b046845ab，于是就可以指定回到未来的某个版本：

`Git reset --hard 9a92c`

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。此时我们再去用 cat 查看下内容

```
MINGW32:/D/GitResp1

Yandld@YANDLD-PC /D/GitResp1 (master)
$ git reset --hard 9a92c
HEAD is now at 9a92c94 add GPL

Yandld@YANDLD-PC /D/GitResp1 (master)
$ cat test.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Yandld@YANDLD-PC /D/GitResp1 (master)
$
```

果然，我胡汉山又回来了。这样，就解决了本章一开始提出的一个问题，如何切换到任意一个历史版本。到此为止，我们再也不需要“时间人物地点事件”式文件命名方法了。GIT 初探结束。大家可以再进入下一个小节之前多多练习。熟练掌握最近本的几个 GIT 命令

`Git init, git reset, git status, git add, git rm, git commit git log`

总结：

- commit 可以理解为 GIT 保存历史版本的一个最小单元，一个 commit。就是一个时间以及再这个时间点上的所有文件记录。是 GIT reset 命令切换的落脚点。

- `git commit` 是提交所有暂存(stage)的 文件修改信息，也就是说，在有效 `git add`, `git rm` 之后，才能进行 `git commit`，再换句话说 使用 `git status` 看到有绿色的部分的时候，代表目前 GIT 已经暂存了一些修改，才可以使用 `git commit`。否则，GIT 会认为，根本没有暂存的修改，无法保存成一个 `commit`。

- 一部分暂存的文件修改了之后，又想撤回，恢复到原来的状态，有什么简便的方法呢？那就是 GIT `checkout`，比如我们修改了 `test.txt`，但最后又决定放弃修改，使用 `GIT checkout test.txt` 就可以将 `test.txt` 恢复到最近的一个 `commit` 的状态。

使用远程仓库

到目前为止，我们已经掌握了如何在 GIT 仓库里对一个文件进行时光穿梭，你再也不用担心文件备份或者丢失的问题了。但是人有祸兮旦福，硬盘有阴晴圆缺，万一哪天自己的电脑挂掉了。那岂不是可悲可叹。。。但是 GIT 的远程仓库功能，轻松帮你搞定这种问题。有了远程仓库，再也不用担心自己的硬盘了。

GIT 是分布式版本控制系统，同一个 GIT 仓库，可以分布到不同的机器上。怎么分布呢？最早，肯定只有一台机器有一个原始版本库，此后，别的机器可以“克隆”这个原始版本库，而且每台机器的版本库其实都是一样的，并没有主次之分。

你肯定会想，至少需要两台机器才能玩远程库不是？但是我只有一台电脑，怎么玩？

其实一台电脑上也是可以克隆多个版本库的，只要不在同一个目录下。不过，现实生活中是不会有人这么傻的在一台电脑上搞几个远程库玩，因为一台电脑上搞几个远程库完全没有意义，而且硬盘挂了会导致所有库都挂掉，所以我也不告诉你在一台电脑上怎么克隆多个仓库。

实际情况往往是这样，找一台电脑充当服务器的角色，每天 24 小时开机，其他每个人都从这个“服务器”仓库克隆一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

完全可以自己搭建一台运行 GIT 的服务器，不过现阶段，为了学 GIT 先搭个服务器绝对是小题大作。好在这个世界上有个叫 github 的神奇网站，从名字就可以看出，这个网站就是提供 GIT 仓库托管服务的，所以，只要注册一个 github 账号，就可以免费获得 GIT 远程仓库。

在继续阅读后续内容前，请自行注册 github 账号。由于你的本地 GIT 仓库和 github 仓库之间的传输是通过 SSH 加密的，所以，需要一点设置：

第 1 步：创建 SSH Key。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：

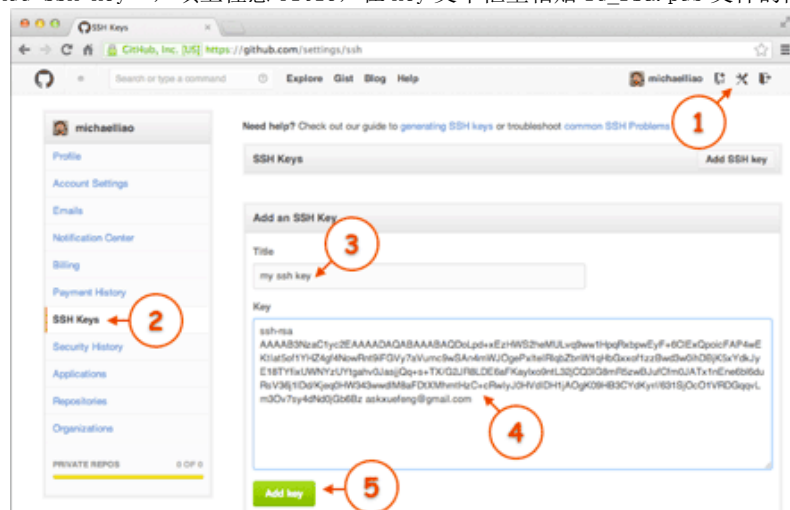
```
$ ssh-keygen -t rsa -C "youremail@example.com"
```

你需要把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。

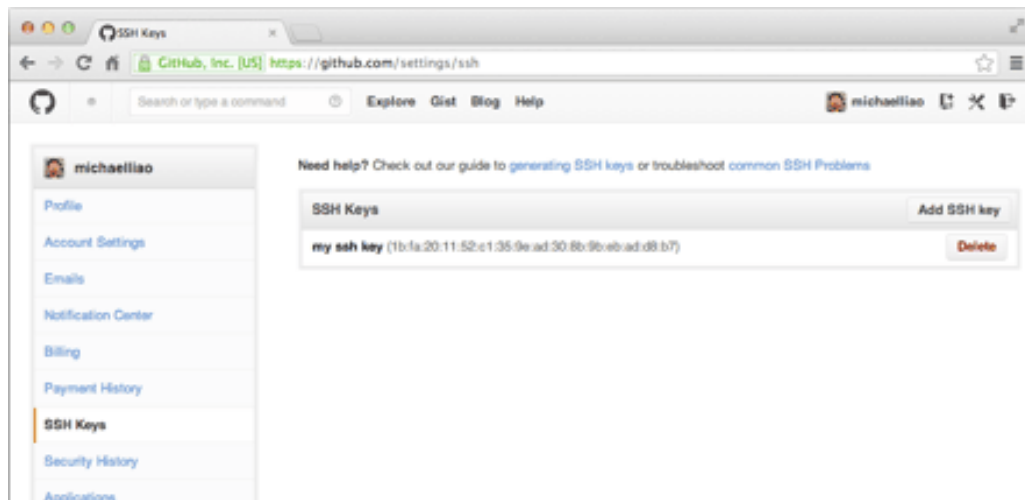
如果一切顺利的话，可以在用户主目录里找到 `.ssh` 目录，里面有 `id_rsa` 和 `id_rsa.pub` 两个文件，这两个就是 SSH Key 的密钥对，`id_rsa` 是私钥，不能泄露出去，`id_rsa.pub` 是公钥，可以放心地告诉任何人。

第 2 步：登陆 github，打开“Account settings”，“SSH Keys”页面：

然后，点“Add SSH Key”，填上任意 Title，在 Key 文本框里粘贴 `id_rsa.pub` 文件的内容：



点“Add Key”，你就应该看到已经添加的 Key：



为什么 github 需要 SSH Key 呢？因为 github 需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而 GIT 支持 SSH 协议，所以，github 只要知道了你的公钥，就可以确认只有你自己才能推送。

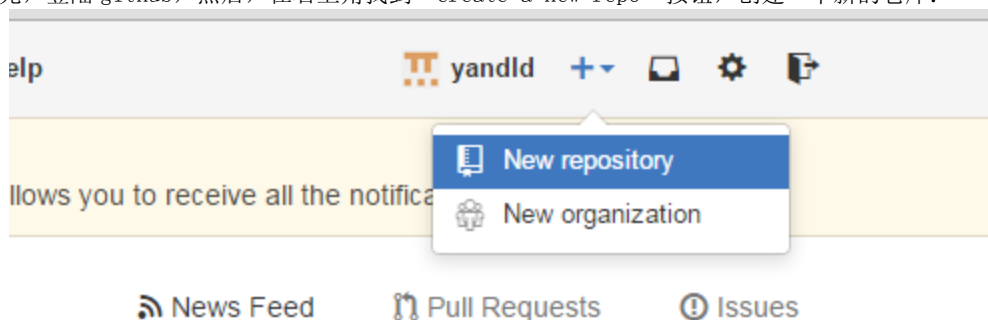
当然，github 允许你添加多个 Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的 Key 都添加到 github，就可以在每台电脑上往 github 推送了。

最后友情提示，在 github 上免费托管的 GIT 仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去。

如果你不想让别人看到 GIT 库，有两个办法，一个是交点保护费，让 github 把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个 GIT 服务器，因为是你自己的 GIT 服务器，所以别人也是看不见的。确保你拥有一个 github 账号后，我们就即将开始远程仓库的学习。


现在的情景是，你已经在本地创建了一个 GIT 仓库后，又想在 github 创建一个 GIT 仓库，并且让这两个仓库进行远程同步，这样，github 上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

首先，登陆 github，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：



在 Repository name 填入 GitRespl，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的 GIT 仓库：

Owner

 yandld

Repository name

GitResp1

Great repository names are short and memorable. Need inspiration? How about **flaming-octo-**

Description (optional)

My first Git Resp

☒ Public

Anyone can see this repository. You choose who can commit.

☐ Private

You choose who can see and commit to this repository.

☐ Initialize this repository with a README

目前，在 github 上的这个 GitResp1 仓库还是空的，github 告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到 github 仓库。

现在，我们根据 github 的提示，在本地的 GitResp1 仓库下运行命令：

```
$ git remote add origin git@github.com:yandld/Resp1.git
```

```
MINGW64:/c/Users/Yandld/Desktop/MyFirstResp
On branch master
nothing to commit, working directory clean

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git reset --hard HEAD^
HEAD is now at 3dfc52e my first commit

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ cat test.txt
Git is a version control system.
Git is free software.

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git log
commit 3dfc52ef2d108779da996a66d23e562ffa38b96b
Author: yandld <yandld@126.com>
Date:   Fri Sep 18 22:26:06 2015 +0800

    my first commit

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git remote add origin git@github.com:yandld/gitResp1.git

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$
```

请千万注意，把上面的 yandld 替换成你自己的 github 账户名，否则，你在本地关联的就是我的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的 SSH Key 公钥不在我的账户列表中。

添加后，远程库的名字就是 origin，这是 GIT 默认的叫法，也可以改成别的，但是 origin 这个名字一看就知道是远程库。

下一步，就可以把本地库的所有内容推送到远程库上：

```
$ git push -u origin master
```



```
MINGW64:/c/Users/Yandld/Desktop/MyFirstResp
modified: test.txt

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master|MERGING)
$ git commit
[master 1f354b3] Merge branch 'master' of github.com:yandld/gitResp1

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git status
On branch master
nothing to commit, working directory clean

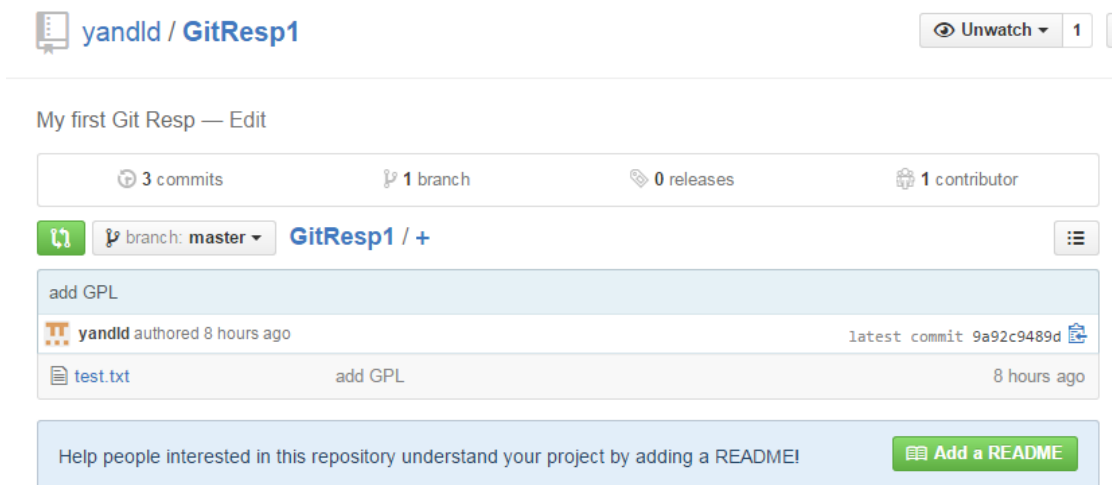
Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$ git push -u origin master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 647 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To git@github.com:yandld/gitResp1.git
 9a92c94..1f354b3 master -> master
Branch master set up to track remote branch master from origin.

Yandld@Yandld-PC MINGW64 ~/Desktop/MyFirstResp (master)
$
```

把本地库的内容推送到远程，用 GIT push 命令，实际上是把当前分支 master 推送到远程。

由于远程库是空的，我们第一次推送 master 分支时，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，在以后的推送或者拉取时就可以简化命令。

推送成功后，可以立刻在 github 页面中看到远程库的内容已经和本地一模一样：



看到网页自己刚刚同步成功的内容，心中自豪感是否油然而生？ 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 github，现在，你就拥有了真正的分布式版本库！

克隆远程库

有了远程库，我们就可以在任意一台本地计算机上克隆你远程服务器的代码

```
$ git clone git@github.com:yandld/gitResp1.git
```

注意把 GIT 库的地址换成你自己的，然后进入 GitResp1 目录看看，是不是你直接提交的内容呢？ 如果有多个人协作开发，那么每个人各自从远程克隆一份就可以了。然后就可以在本地计算上进行开发，之后暂存，提交，并且 push 到数据库了。

自此，为大家介绍了 GIT 最简单的入门知识，当然，GIT 的功能还远不止于此，这里面还有很多细节没有介绍，比如分支的概念，多人开发中冲突解决的问题等等。笔者自认为如果是初学 GIT, 这些概念最在已经熟练掌握了 GIT 最基本的命令如，status log add commit clone push pull 等的基础上再进一步学习。到此，本章的内容到此也就结束了。大家希望对 GIT 有更多的了解，还需要多多借助网上的资料。不过最重要的，还是要多多练习。尽量尝试使用 GIT 来管理你的软件库，只有在实践中，才能学习提高，才能进步的快。

本章小结

初始化一个 GIT 仓库，使用 `git init` 命令。

1. 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。

2. 经常使用 `git log` 和 `git status` 来查看当前的修改历史和 目前的 GIT 状态

3. 要关联一个远程库，使用命令 `git remote add origin git@server-name:path/repo-name.git`；

关联后，使用命令 `git push -u origin master` 第一次推送 master 分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令 `git push origin master` 推送最新修改；

分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作。当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

4. 要克隆一个仓库，首先必须知道仓库的地址，然后使用 `git clone` 命令克隆。

GIT 支持多种协议，包括 https，但通过 ssh 支持的原生 GIT 协议速度最快。