

写在最前面.....

本手册详细说明了 64 位 Windows 系统下 Git 工具的安装过程并粗略提及其与 GitHub 的入门级使用，面向主要是学生的一切初学者。

手册内容并非完全原创，内容也并不完善，只能说是一个对网上的一些教程做了一个较为完善的整理，或者说是记录了笔者自己学习的过程，加入了自己的理解。如果想进一步学习 Git 和 GitHub，请大家多多百度或者 Google 吧！

谨以此献给所有在秃头道路上知难而上的同胞们！

在此非常感谢廖雪峰老师在网站里详细的介绍说明，我正是在廖雪峰老师的肩膀上整理出这份“经验之谈”。

以下是廖雪峰老师的 Git 教程的网址

<https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000>

我也将本手册的 PDF 放到了我的 Github 下，欢迎大家下载！

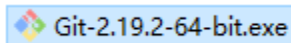
https://github.com/ShiningSYK/Git-GitHub_Document.git

git@github.com:ShiningSYK/Git-GitHub_Document.git

一、Git 的安装（Windows10）

首先在官网中获取最新版本。<https://git-scm.com/>

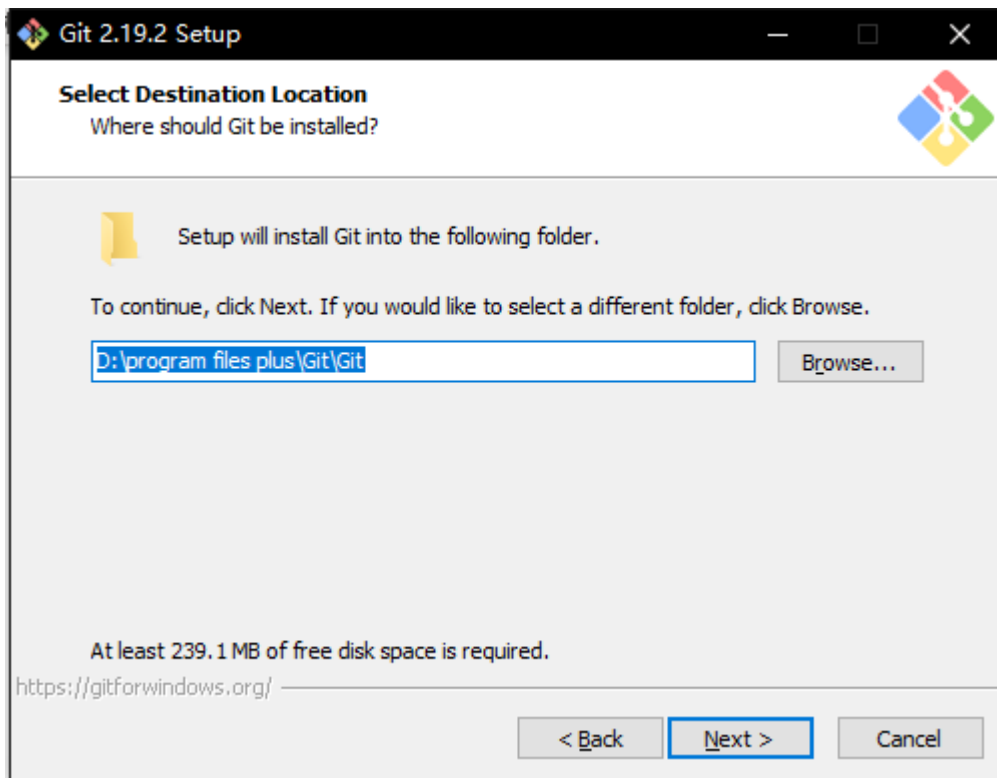
选择系统（系统版本以及位数）下载安装包。如图为 win10，64 位版本安装包。



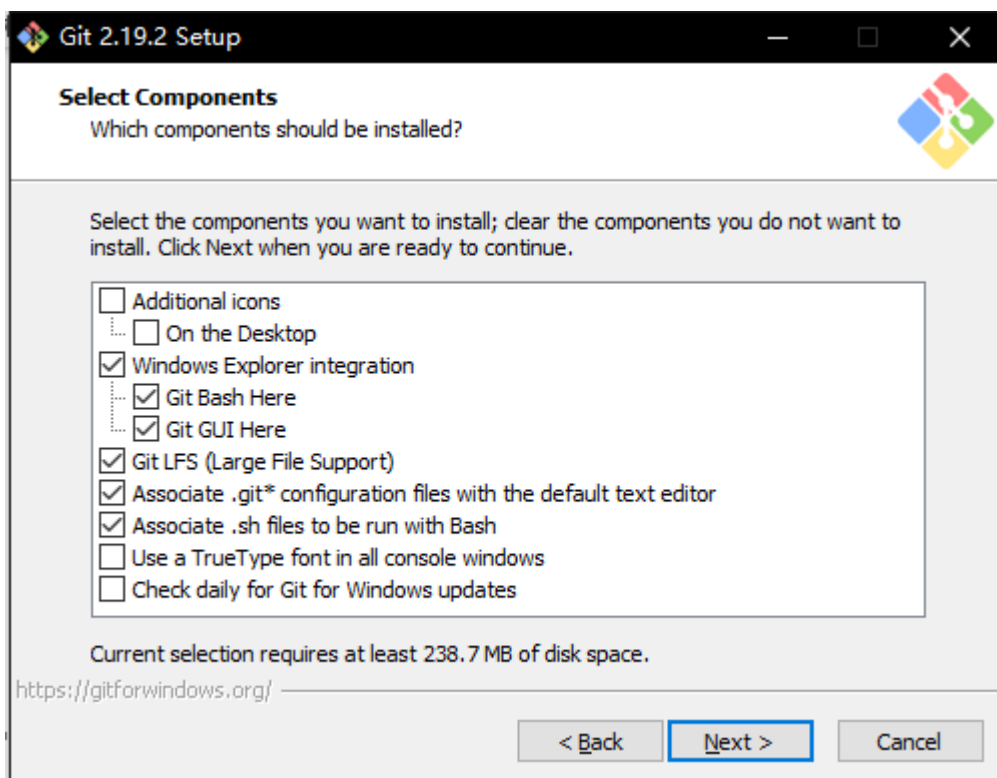
1.双击运行



2.选择安装路径



3.选择组件



说明:

Additional icons 附加图标 On the Desktop 在桌面上

Windows Explorer integration Windows 资源管理器集成鼠标右键菜单

Git Bash Here 命令行 Git GUI Here 可视化界面

Git LFS (Large File Support) 大文件支持

Associate .git* configuration files with the default text editor

将 .git 配置文件与默认文本编辑器相关联

Associate .sh files to be run with Bash

将.sh 文件关联到 Bash 运行

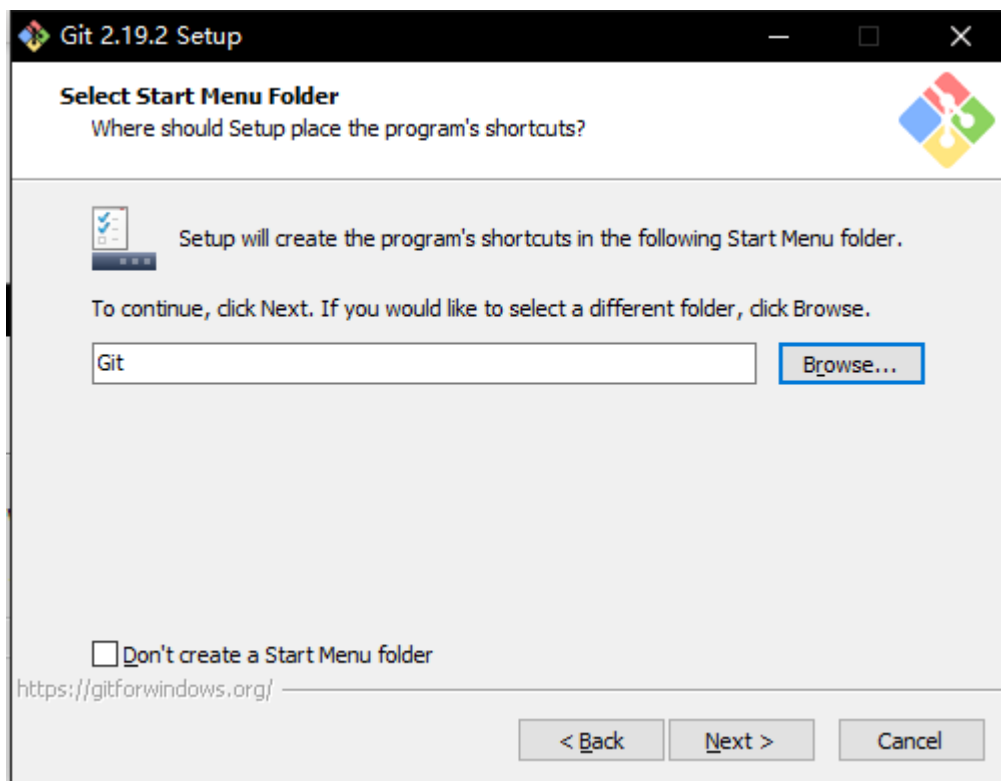
Use a TrueType font in all console windows

在所有控制台窗口中使用 TrueType 字体

Check daily for Git for Windows updates

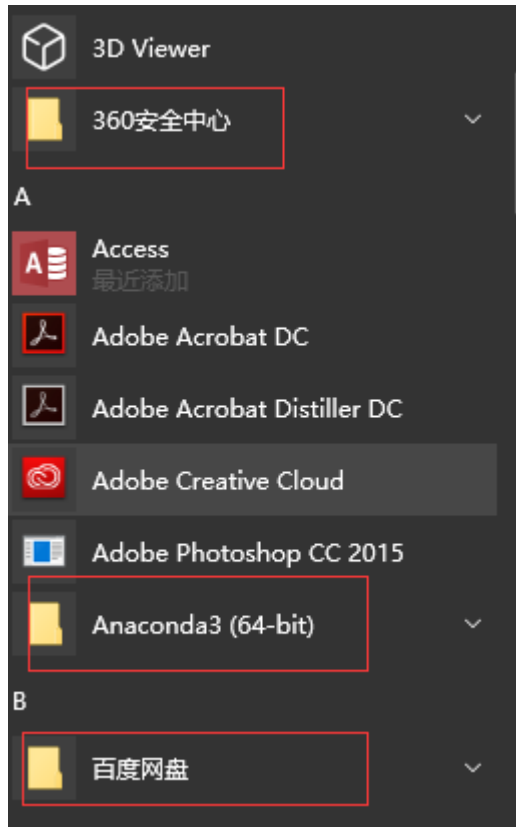
每天检查 Git 是否有 Windows 更新

5.选择开始菜单文件夹



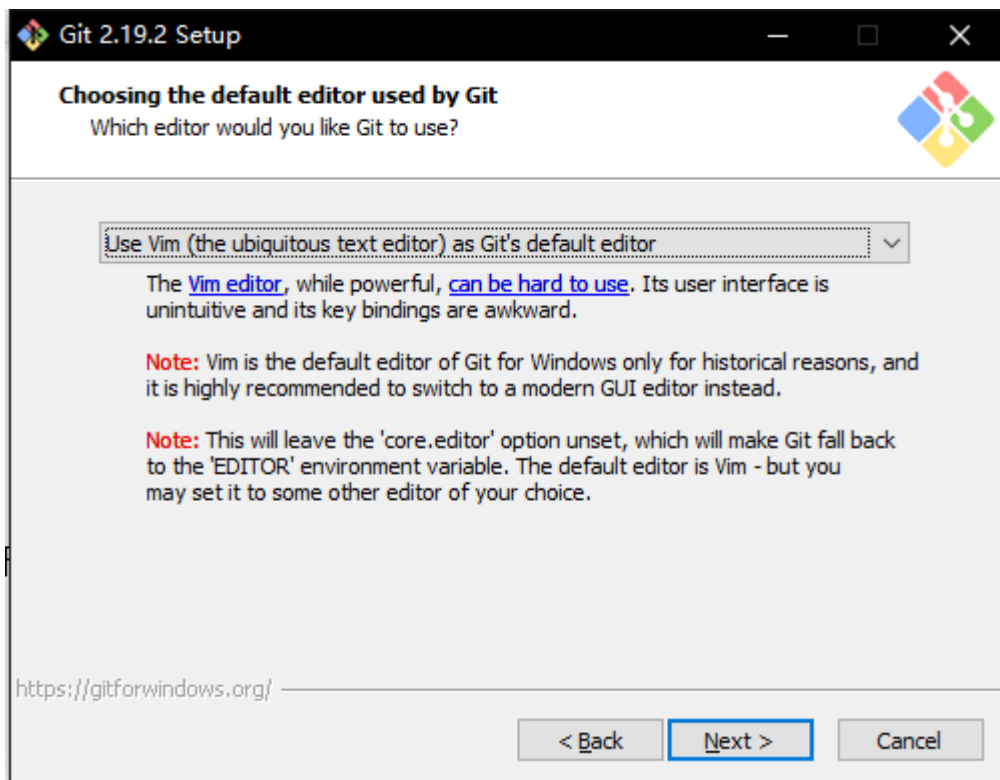
选择开始菜单文件夹

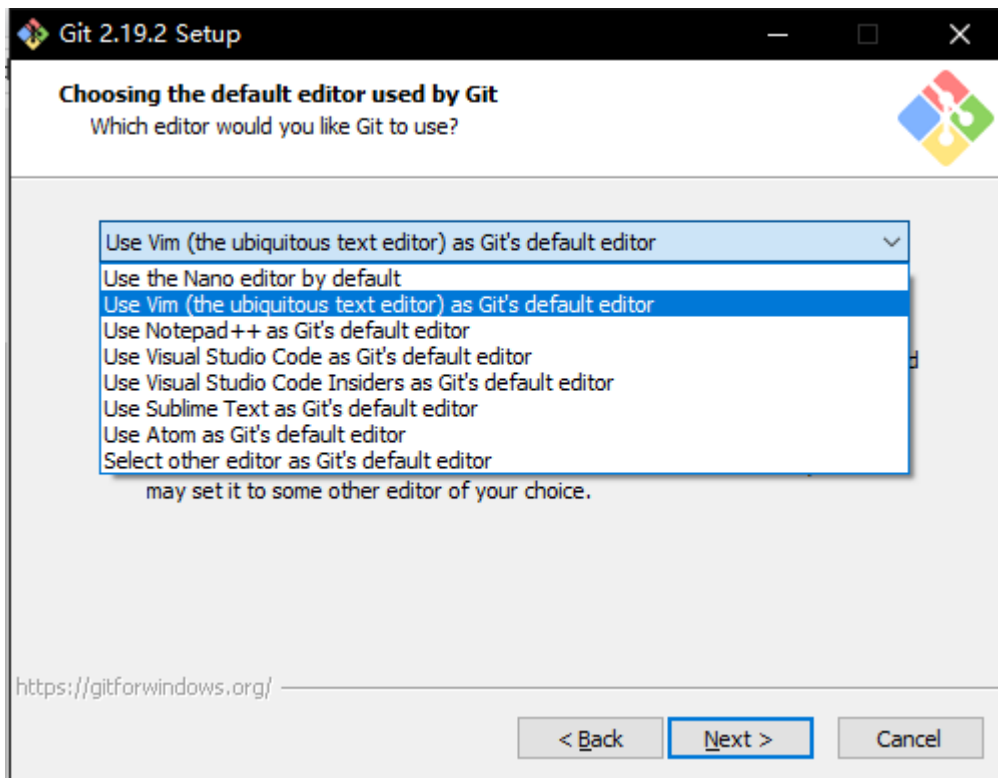
Ps: 开始菜单文件夹是什么呢, 顾名思义, 按一下 win 或者点击打开开始菜单



红框中的就是开始菜单文件夹，可有可无，只不过在开始菜单中显示，看起来程序集中一点。

6. 选择 Git 使用的默认编辑器





说明：Use the Nano editor by default 默认使用 Nano 编辑器

Use Vim (The ubiquitous text editor) as Git's default editor

使用 Vim 作为 Git 的默认编辑器

Use Notepad++ as Git's default editor

使用 Notepad++ 作为 Git 的默认编辑器

Use Visual Studio Code as Git's default editor

使用 Visual Studio Code 作为 Git 的默认编辑器

Use Visual Studio Code Insiders as Git's default editor

使用 Visual Studio Code Insiders 作为 Git 的默认编辑器

Use Subline Text as Git's default editor

使用 Subline 作为 Git 的默认编辑器

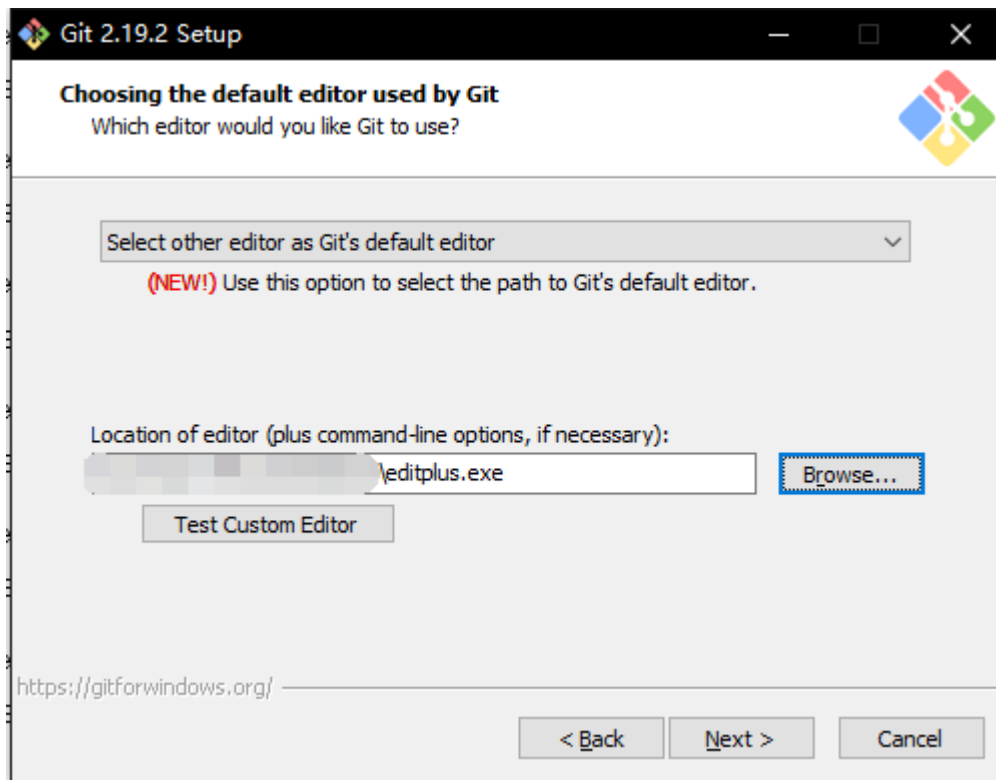
Use Atom as Git's default editor

使用 Atom 作为 Git 的默认编辑器

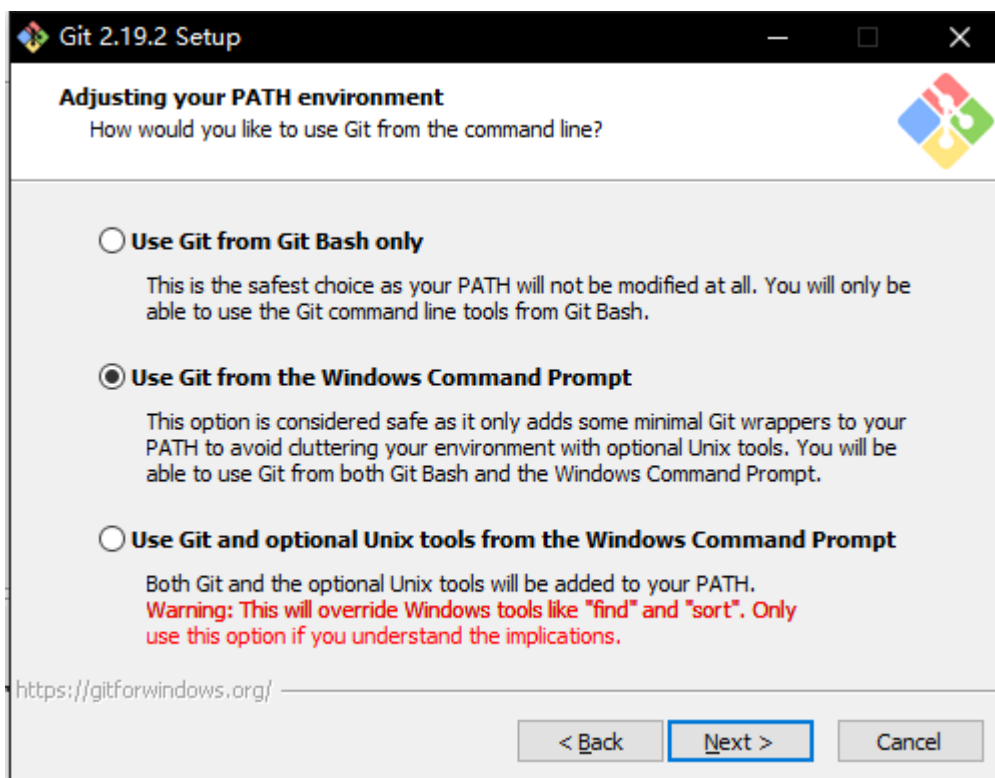
Select other editor as Git's default editor

选择其他的编辑器作为 Git 的默认编辑器

PS：在这里我选择了 **Editor Plus** 作为默认编辑器，找到应用程序即可



7.配置环境变量



说明： Use Git from Git Bash only

This is the safest choice as your PATH will not be modified at all. You will only be able to use the Git command line tools form Git Bash.

这是最安全的选择，因为您的 PATH 根本不会被修改。您只能使用 Git Bash 的 Git 命令行工具。

Use Git from the Windows Command Prompt

This option is considered safe as it only adds some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools . You will be able to use Git from both Git Bash and the Windows Command Prompt.

这个选项被认为是安全的，因为它只向 PATH 添加一些最小的 Git 包，以避免使用可选的 Unix 工具混淆环境。 您将能够从 Git Bash 和 Windows 命令提示符中使用 Git。

Use Git and optional Unix tools from the Windows Command Prompt

从 Windows 命令提示符使用 Git 和可选的 Unix 工具

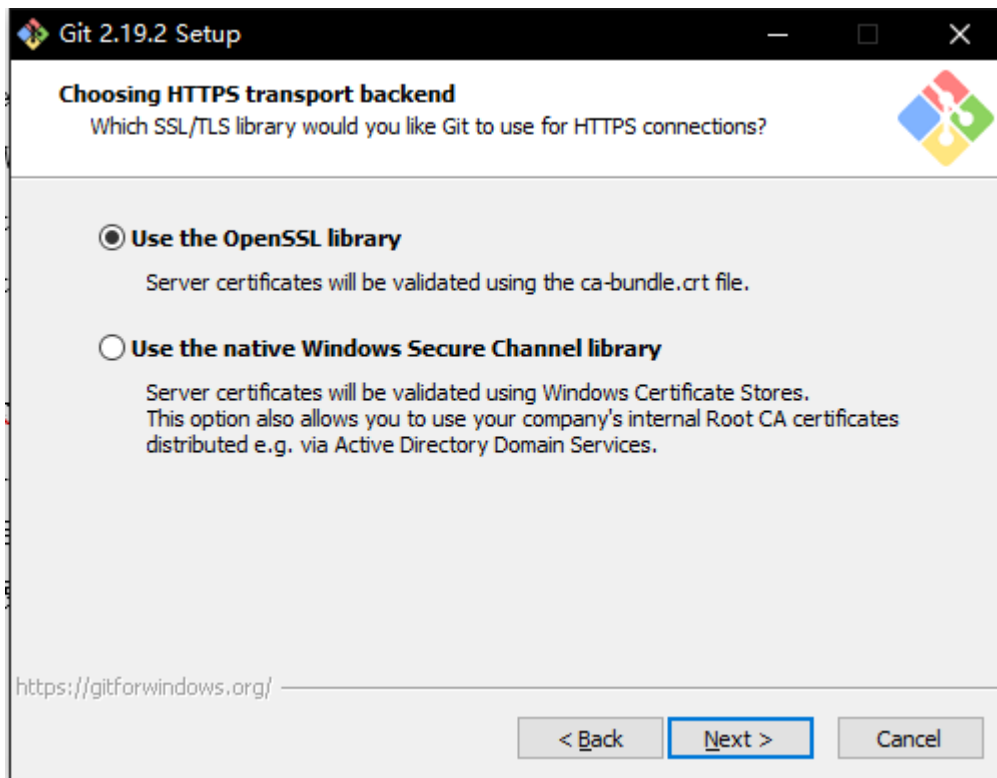
Both Git and the optional Unix tools will be added to you PATH

Git 和可选的 Unix 工具都将添加到您计算机的 PATH 中

Warning: This will override Windows tools like "find and sort". Only use this option if you understand the implications.

警告：这将覆盖 Windows 工具，如 “ find 和 sort ”。只有在了解其含义后才使用此选项。

8.选择 Https 传输后端



说明： Use the OpenSSL library

使用 OpenSSL 库

Server certificates will be validated using the ca-bundle.crt file.

服务器证书将使用 ca-bundle.crt 文件进行验证。

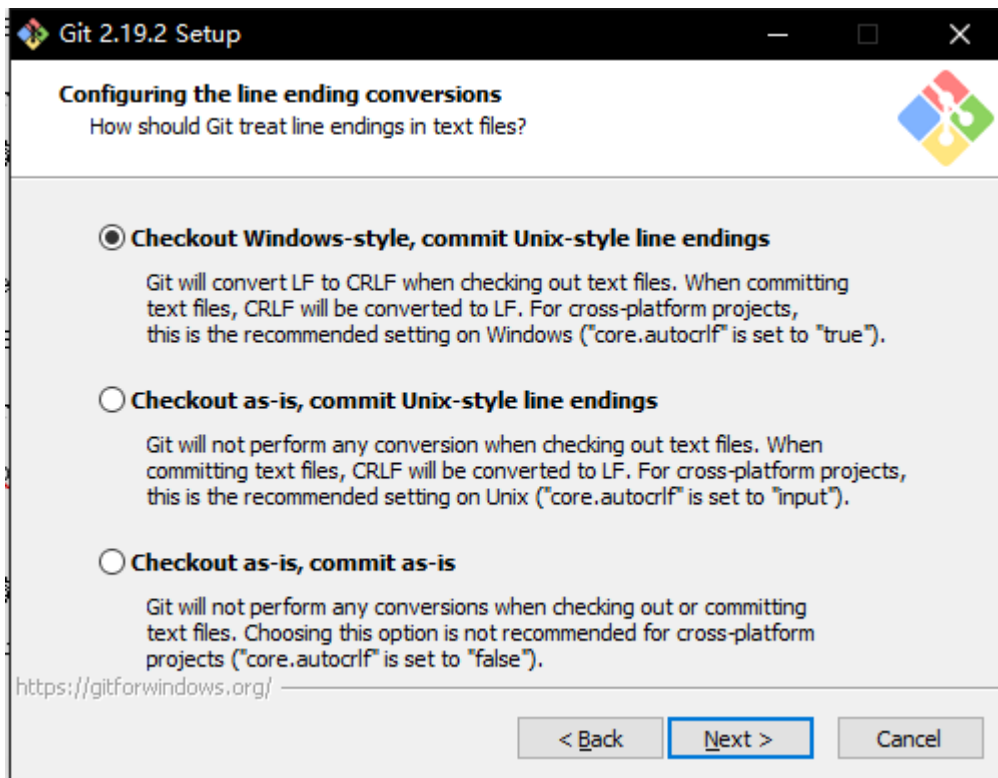
Use the native Windows Secure Channel library

使用本地 Windows 安全通道库

Server certificates will be validated using Windows Certificate Stores. This option also allows you to use your company's internal Root CA certificates distributed e.g. via Active Directory Domain Services.

服务器证书将使用 Windows 证书存储验证。此选项还允许您使用公司的内部根 CA 证书，例如，通过 Active Directory Domain Services 。

9. 配置行结束转换



说明：Checkout Windows-style,commit Unix-style line endings

Git will convert LF to CRLF when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Windows ("core.autocrlf" is set to "true")

在检出文本文件时，Git 会将 LF 转换为 CRLF。当提交文本文件时，CRLF 将转换为 LF。对于跨平台项目，这是 Windows 上推荐的设置（“core.autocrlf”设置为“true”）

Checkout as-is , commit Unix-style line endings

Git will not perform any conversion when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Unix ("core.autocrlf" is set to "input")

在检出文本文件时，Git 不会执行任何转换。提交文本文件时，CRLF 将转换为 LF。对于跨平台项目，这是 Unix 上的推荐设置（“core.autocrlf”设置为“input”）

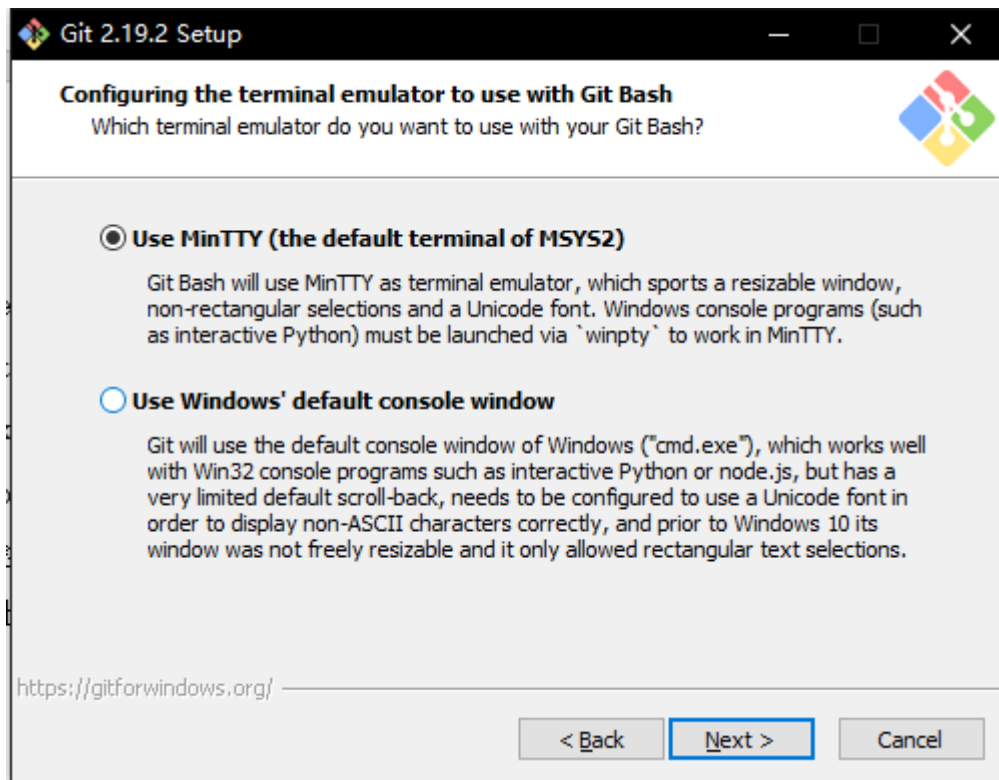
Checkout as-is, commit as-is

Git will not perform any conversions when checking out or committing text files. Choosing this option is not recommended for cross-platform projects

("core.autocrlf" is set to "false")

在检出或提交文本文件时，Git 不会执行任何转换。对于跨平台项目，不推荐使用此选项（“core.autocrlf”设置为“false”）

10. 配置终端模拟器以与 Git Bash 一起使用



说明：Use MinTTY (the default terminal of MSYS2)

Git Bash will use MinTTY as terminal emulator, which sports a resizable window, non-rectangular selections and a Unicode font. Windows console programs (such as interactive Python) must be launched via 'winpty' to work in MinTTY.

Git Bash 将使用 MinTTY 作为终端模拟器，该模拟器具有可调整大小的窗口，非矩形选区和 Unicode 字体。Windows 控制台程序（如交互式 Python）必须通过 'winpty' 启动才能在 MinTTY 中运行。

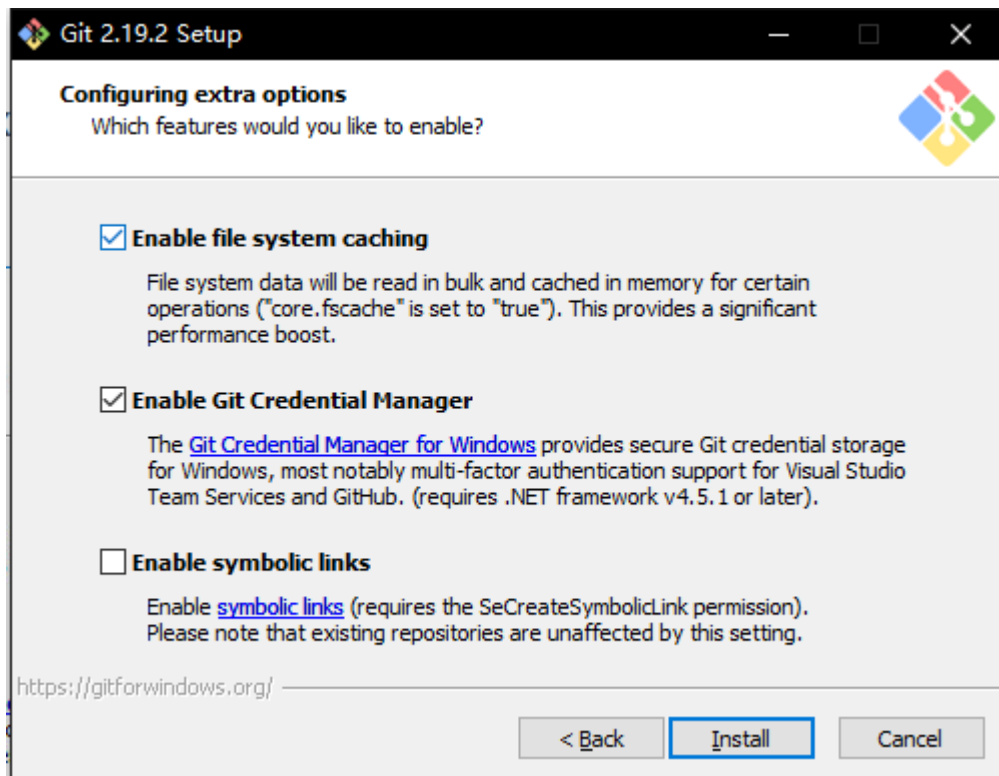
Use Windows' default console window

Git will use the default console window of Windows ("cmd.exe"), which works well with Win32 console programs such as interactive Python or node.js, but has a very limited default scroll-back, needs to be configured to use a Unicode font in order to display non-ASCII characters correctly, and prior to Windows 10 its windows was not

freely resizable and it only allowed rectangular text selections.

Git 将使用 Windows 的默认控制台窗口 (“cmd.exe”), 该窗口可以与 Win32 控制台程序 (如交互式 Python 或 node.js) 一起使用, 但默认的回滚非常有限, 需要配置为使用 unicode 字体以正确显示非 ASCII 字符, 并且在 Windows 10 之前, 其窗口不能自由调整大小, 并且只允许矩形文本选择。

11. 配置额外的选项



说明: Enable file system caching 启用文件系统缓存

File system data will be read in bulk and cached in memory for certain operations ("core.fscache" is set to "true"). This provides a significant performance boost.

文件系统数据将被批量读取并缓存在内存中用于某些操作 (“core.fscache”设置为 “true”)。 这提供了显著的性能提升。

Enable Git Credential Manager 启用 Git 凭证管理器

The Git Credential Manager for Windows provides secure Git credential storage for Windows, most notably multi-factor authentication support for Visual Studio Team Services and GitHub. (requires .NET framework v4.5.1 or or later).

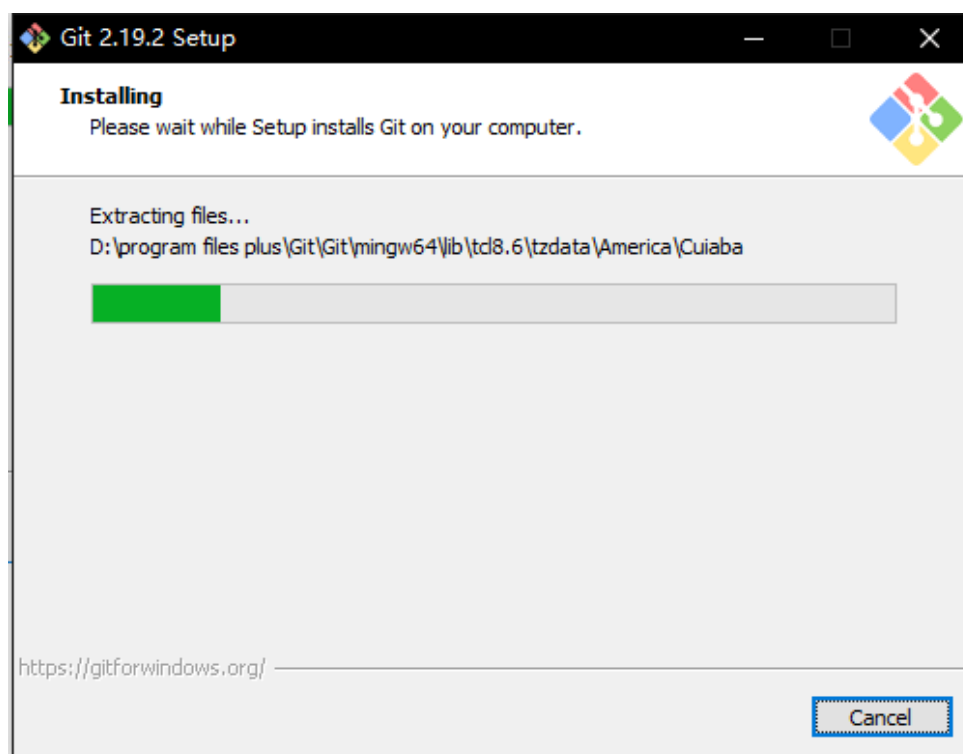
Windows 的 Git 凭证管理器为 Windows 提供安全的 Git 凭证存储，最显着的是对 Visual Studio Team Services 和 GitHub 的多因素身份验证支持。（需要 .NET Framework v4.5.1 或更高版本）。

Enable symbolic links 启用符号链接

Enable symbolic links (requires the SeCreateSymbolicLink permission). Please note that existing repositories are unaffected by this setting.

启用符号链接（需要 SeCreateSymbolicLink 权限）。请注意，现有存储库不受此设置的影响。

12.开始安装



二、Git 的配置（不使用 GitHub 的同学请直接从该目录下的 3. 开始学习，但我强烈推荐使用 GitHub）

题外话：

首先，强烈建议先注册一个 GitHub 账号，<https://github.com/>

通过官方指南了解 GitHub 是个什么东西，并且如何使用
<https://guides.github.com/activities/hello-world/>

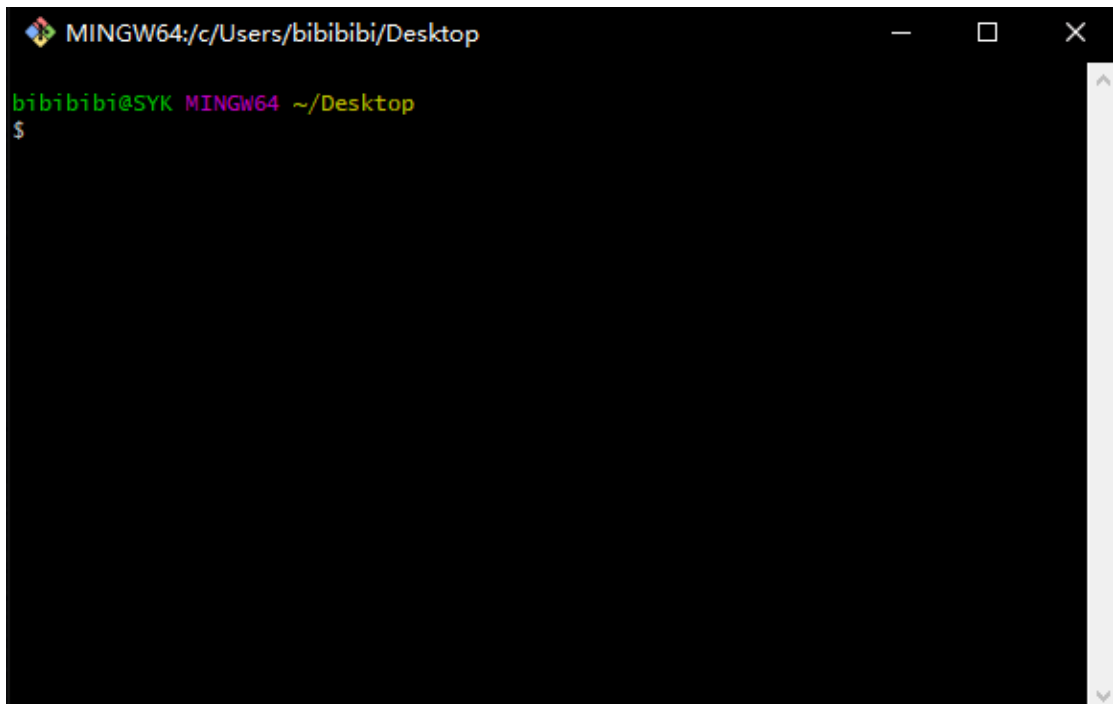
在这里，我们先来说一下什么是 GitHub 和为什么要使用 GitHub

回想一下我们为什么要使用 Git？为了实现版本控制。一般来说实现版本控制肯定是要将项目放到一台机器中，对项目的修改进行追踪，这样，我们至少需要两台电脑（一台电脑当然也可以，但不会有人无聊到在一台电脑中创建多个版本库导来导去玩），那么在实际使用的时候，我们会将版本库布置到远端的服务器上，但对于我们学生来说，无论是日常使用还是做小项目，这么做绝对是小题大做。于是，我们就爱上了 GitHub 这个东西。

首先从名字上我们就可以看出，GitHub 就是提供 Git 仓库托管服务的一个网站，实际上我们也可以把它当作一个布置在远端的服务器，里面保存了你的版本库。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。

为了使用 GitHub，首先我们要**获取并设置一个 SSH 密钥**，为什么要使用这个东西呢，用 GitHub 官方的话来说，“使用 ssh 协议，您可以连接远程服务器和服务并对其进行身份验证。使用 ssh 密钥，您可以连接到 github，而无需在每次访问时提供您的用户名或密码。”接下来展示步骤。

1.生成密钥



输入 `ssh-keygen -t rsa -b 4096 -C "your_email@example.com"`，使用你注册 GitHub 的邮箱替换 `your_email@example.com`。

点击回车后提示输入，输入要在其中保存密钥的文件

```
Enter file in which to save the key (/c/Users/bibibibi/.ssh/id_rsa):
```

此时我们按 `Enter`，这将接受默认文件位置，即保存在 `C/Users/username/.ssh/id_rsa` 中。

然后又有提示输入，输入一个密码，该密码为 SSH 密钥安全密码。

```
Enter passphrase (empty for no passphrase):
```

为什么我们需要这个密码呢，因为“使用 `ssh` 密钥，如果有人访问您的计算机，他们还可以访问使用该密钥的每个系统。若要添加额外的安全层，可以向 `ssh` 密钥添加密码。您可以使用安全地保存密码，这样就不必重新输入密码。”详情请参看官方说明 <https://help.github.com/articles/working-with-ssh-key-passphrases/>。

对于我们个人使用来说，完全没必要添加安全密码，因此只要按 `Enter` 即可。

至此，我们已经成功生成一个 SSH 密钥，结果如图：

```
MINGW64:/c/Users/bibibibi/Desktop
bibibibi@SYK MINGW64 ~/Desktop
$ ssh-keygen -t rsa -b 4096 -C ""
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/bibibibi/.ssh/id_rsa):
Created directory '/c/Users/bibibibi/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/bibibibi/.ssh/id_rsa.
Your public key has been saved in /c/Users/bibibibi/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:
The key's randomart image is:
+---[RSA 4096]---+
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
|                 |
+---[SHA256]-----+
```

2.添加密钥

我们需要先获取密钥内容，有两种方式，其一为找到该文件，用文本编辑器打开全选复制内容到剪切板，另一种方法为在 Git Bash 中输入

```
clip < ~/.ssh/id_rsa.pub
```

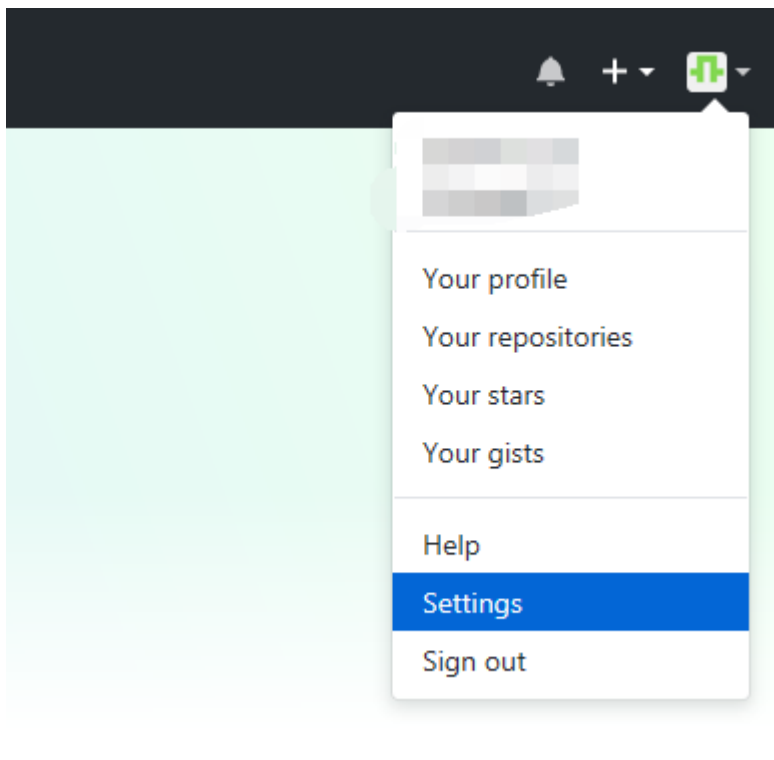
即可将文件内容复制到剪切板，当然如果你之前选择的不是该默认文件保存密钥，你需要修改上述代码。

结果如图：

```
bibibibi@SYK MINGW64 ~/Desktop
$ clip < ~/.ssh/id_rsa.pub

bibibibi@SYK MINGW64 ~/Desktop
$
```

现在，我们回到 GitHub，找到页面右上角的个人头像，选择设置：



Personal settings

Profile

Account

Emails

Notifications

Billing

SSH and GPG keys

Security

Sessions

Blocked users

Repositories

Organizations

Saved replies

Applications

Developer settings

SSH keys

There are no SSH keys associated with your account.

Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#).

GPG keys

There are no GPG keys associated with your account.

Learn how to [generate a GPG key](#) and add it to your account.

New SSH key

New GPG key

点击红框中的按钮

Personal settings

Profile

Account

Emails

Notifications

Billing

SSH and GPG keys

Security

Sessions

Blocked users

Repositories

Organizations

Saved replies

Applications

Developer settings

SSH keys / Add new

Title

Key

Begins with 'ssh-rsa', 'ssh-dss', 'ssh-ed25519', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521'

Add SSH key

其中，title 为描述性标签，“在 "标题" 字段中，为新键添加描述性标签。例如，如果您使用的是个人 *mac*，则可以将此密钥称为 "个人 *macbook air*"。”，key 内容则是刚才我们复制的密钥，输入完成后点击添加。

Personal settings

Profile

Account

Emails

Notifications

Billing

SSH and GPG keys

Security

Sessions

Blocked users

Repositories

Organizations

Saved replies


Applications

Developer settings

SSH keys

New SSH key

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

 SSH

Added on Dec 5, 2018
Never used — Read/write

Delete

Check out our guide to [generating SSH keys](#) or [troubleshoot common SSH Problems](#).

GPG keys

New GPG key

There are no GPG keys associated with your account.

Learn how to [generate a GPG key](#) and [add it to your account](#).

之后，我们也可以通过访问该页面来查看并修改密钥。

并且我们可以测试是否成功创建密钥。同样是在 `Git Bash` 中，输入

```
ssh -T git@github.com
```

如果输出的最后一句话为 `You've successfully authenticated, but GitHub does`

not provide shell access，就表明你已成功创建并连接到你的 GitHub。

3.用户信息配置

因为 Git 是分布式版本控制系统，Git 跟踪了是谁修改了项目，所以，每个机器都必须自报家门。Git 需要知道你的用户名和电子邮件。你必须提供用户名，但可以使用虚构的电子邮件地址。

在 Git Bash 中输入

```
git config --global user.name "Your Name"
```

```
git config --global user.email email@example.com
```

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。

结果如图：



```
MINGW64:/c/Users/bibibibi
bibibibi@SYK MINGW64 ~
$ git config --global user.name " "
bibibibi@SYK MINGW64 ~
$ git config --global user.email " "
bibibibi@SYK MINGW64 ~
$ |
```

4.创建版本库（Repository）

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”。

首先进入到 Git Bash 中，为你的版本库选择一个路径，并创建。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。操作如下：

```

bibibibi@SYK MINGW64 ~
$ cd d:

bibibibi@SYK MINGW64 /d
$ mkdir TestRepository

bibibibi@SYK MINGW64 /d
$ cd TestRepository

bibibibi@SYK MINGW64 /d/TestRepository
$ pwd
/d/TestRepository

bibibibi@SYK MINGW64 /d/TestRepository
$ git init
Initialized empty Git repository in D:/TestRepository/.git/

bibibibi@SYK MINGW64 /d/TestRepository (master)
$

```

cd d: #进入 d 盘

mkdir TestRepository #创建文件夹 TestRepository

cd TestRepository #进入 TestRepository

pwd #显示 TestRepository 的路径

git init #通过该命令把这个目录变成 Git 可以管理的仓库

提示信息的意思是：初始化了一个空的 Git 版本库。并且我们可以在库路径下找到一个隐藏文件夹 `.git` 这个文件夹是用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。**

并且我们可以看到最后有个青色的 (master)，这表示我们现在在该版本库的 master 分支中。（有关分支的概念自行百度）

5. 添加文件进库

下面我们就往我们的库里放点东西，最简单的当然是 TXT 文本啦。

说明（网上看来的）：

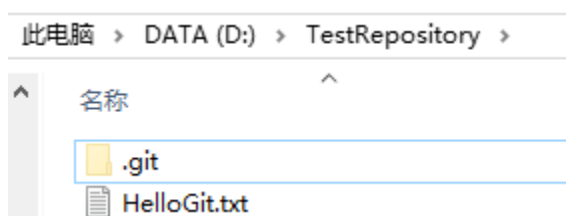
“使用 Windows 的童鞋要特别注意：

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。”

因此我建议使用 Notepad++（免费又好用）或者 EditPlus（收费的，但是有可以破解，B 格很高），并且将编码方式改为 UTF-8 或者 UTF-8 without BOM。

我们将创建的文件放到 TestRepository 中，如图：



文件内容为：

```
1 hello Git&Github
```

现在回到 Git Bash 中，让我们先来看一下版本库的状态：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git status
On branch master

No commits yet

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

        HelloGit.txt

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

我们发现在 master 分支中，没有东西被提交，但存在一个未被追踪的文件，就是我们刚才创建并放进去的那个 TXT，并且它提示我们可以使用

`git add <file>...`

操作来添加文件并使其被追踪。很明显，光是把文件放到库对应的文件夹里是远远不够的，因此，我们使用上述指令来将它放进去。结果如图：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git add HelloGit.txt

bibibibi@SYK MINGW64 /d/TestRepository (master)
$
```

我们发现没有提示信息。这就对了，Unix 的哲学是“没有消息就是好消息”，说明添加成功。

现在再来看一下版本库的状态：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git status
On branch master

No commits yet

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

        new file:   HelloGit.txt

bibibibi@SYK MINGW64 /d/TestRepository (master)
$
```

好了，这个文件已经被放进去啦！提示的命令

`git rm --cached <file>...`

的意思是，将 file 踢出库，使其不被追踪，我们可以通过测试来验证，如图：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git rm --cached HelloGit.txt
rm 'HelloGit.txt'

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git status
On branch master

No commits yet

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

        HelloGit.txt

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

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ |
```

我们现在再把它放回去.....

但是仅仅是 add，这个文件实际上还未真正放到库中。接下来，我们使用

`git commit -m "说明"`

指令来提交文件，结果如图：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git commit -m "第一次提交，版本一"
[master (root-commit) 9717723] 第一次提交，版本一
1 file changed, 1 insertion(+)
create mode 100644 HelloGit.txt
```

-m 之后的字符串是本次提交的说明，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。

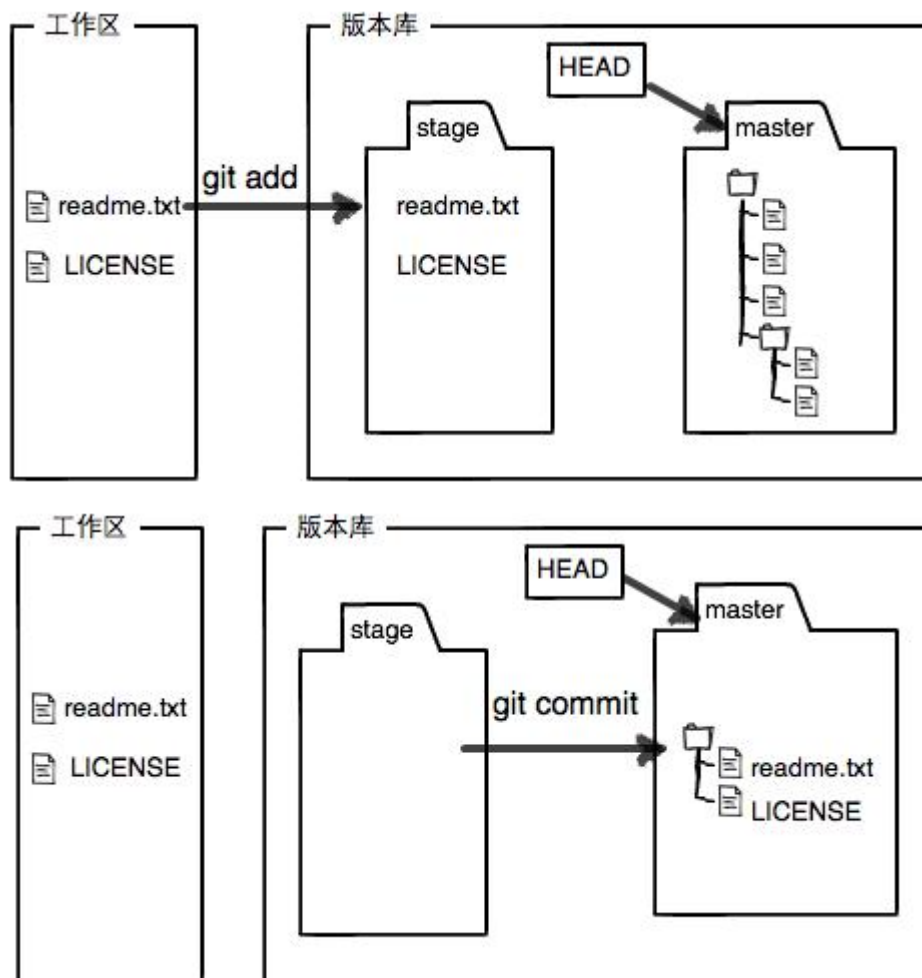
说明：

1 file changed: 1 个文件被改动(我们新添加的 HelloGit.txt 文件); 1 insertions:
插入了两行内容 (HelloGit.txt 有一行内容)

为什么 Git 添加文件需要 **add, commit** 一共两步呢? 一方面因为 **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."
```

另一方面在这两步中实际上涉及到了版本控制的机制, **add** 是将文件放到了一个暂存区 (**stage**) 中, **commit** 是将文件从暂存区移入分支, 下面展示两张图辅助理解, 详情请百度。



6.版本控制（直接相关到 7.）

为了体现 Git 的强大, 我们先修改一下 HelloGit.txt, 如图:

1 hello Git&Github

2 我加了一句话

再用

`git status`

命令来看一下库的状态，如图：

```
bibibibi@SYK MINGW64 /d/TestRepository (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:   HelloGit.txt

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

        HelloGit.txt.bak

no changes added to commit (use "git add" and/or "git commit -a")
bibibibi@SYK MINGW64 /d/TestRepository (master)
$
```

我们发现提示说 HelloGit.txt 文件被修改了，但修改还未被提交，而且又多了一个后缀为 .bak 文件（这个文件是 EditPlus 在保存文件时自动生成的备份文件，但是我们做的就是版本控制啊，我们需要这个么？出现这个就是对我们的侮辱！）。

对于被修改的文件，我们可以先查看被修改的内容，使用命令

`git diff <file>...`

效果如图：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git diff HelloGit.txt
diff --git a/HelloGit.txt b/HelloGit.txt
index bfd9533..a0f7e1a 100644
--- a/HelloGit.txt
+++ b/HelloGit.txt
@@ -1,2 @@
-hello Git&Github
\ No newline at end of file
+hello Git&Github
+我加了一句话
\ No newline at end of file
```

提示信息的意思很明显，与之前相比，我们多加了一行内容，现在我们还是通过 `add` 和 `commit` 来提交文件到版本库。结果如图（期间我加了一行内容为 1，然后又删掉了，都被 Git 记录下来）：


```

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git diff HelloGit.txt
diff --git a/HelloGit.txt b/HelloGit.txt
index bfd9533..c3d8676 100644
--- a/HelloGit.txt
+++ b/HelloGit.txt
@@ -1,3 @@
-hello Git&Github
\ No newline at end of file
+hello Git&Github
+我加了一句话
+1
\ No newline at end of file

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git add HelloGit.txt

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git commit -m "第二次提交, 版本二"
[master b990970] 第二次提交, 版本二
1 file changed, 2 insertions(+), 1 deletion(-)

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ |

```

好了，我们再来看版本库的状态：

```

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        HelloGit.txt.bak

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

```

又是这个东西，看到它我就来气！

##忽略特殊文件

实在受不了了，把这个“小三”去掉我才能继续工作！

我们打开文本编辑器，在库目录下新建一个名为 `.gitignore` 的特殊文件，在这个文件中放入我们想让 Git 屏蔽的“小三”，例如：

```

1 *.bak

```

这样我们就能让 Git 忽视后缀名为 `.bak` 的文件，再通过

```
git status
```

查看库状态，结果如图：

```

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
bibibibi@SYK MINGW64 /d/TestRepository (master)
$

```

现在好了，“小三”不见了，接下来只要再将 `.gitignore` 也提交到库中就可以了！结果如图：

```

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git add .gitignore

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git commit -m "第三次提交，版本三"
[master 4b8f244] 第三次提交，版本三
1 file changed, 1 insertion(+)
create mode 100644 .gitignore

```

我们发现 Git 还会对它有特殊对待。

7.版本控制（上接 6.）

现在让我们来回想一下，我们做了几次提交了。你能很快地说出来，三次，因为最后一次提交就在刚刚，截图还在上面呢！但我们总会有情况，需要看看历史纪录，因此，我们可以通过

`git log`

命令来查看历史提交记录，效果如图：

```

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git log
commit 4b8f244aafde74bc5c159f697540db7b7e5aab48 (HEAD -> master)
Author: ShiningSYK <1052303323@qq.com>
Date:   Wed Dec 5 19:05:01 2018 +0800

    第三次提交，版本三

commit b990970bbfa6f0d1d78491cb03ffc7719993b65c
Author: ShiningSYK <1052303323@qq.com>
Date:   Wed Dec 5 18:52:21 2018 +0800

    第二次提交，版本二

commit 9717723e9799bf9016228afdfb6aef6481d1c603
Author: ShiningSYK <1052303323@qq.com>
Date:   Wed Dec 5 18:19:31 2018 +0800

    第一次提交，版本一

```

如果嫌东西太多，还可以输出简略版，使用

```
git log --pretty=oneline
```

效果如图：

```

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git log --pretty=oneline
4b8f244aafde74bc5c159f697540db7b7e5aab48 (HEAD -> master) 第三次提交，版本三
b990970bbfa6f0d1d78491cb03ffc7719993b65c 第二次提交，版本二
9717723e9799bf9016228afdfb6aef6481d1c603 第一次提交，版本一

```

我们发现每一句输出的开头是一大堆类似乱码的东西，实际上这是 Git 使用的版本号，这是根据 SHA1 计算出来的一个非常大的数字，用十六进制表示。

好了，现在我们启动时光穿梭机，准备把 HelloGit.txt 回退到之前的版本，我们介绍一些相关的命令：

```
git reset --hard commit_id #命令模板，commit_id 表示版本号
```

```
git reset --hard HEAD^ #回退到上一个版本，HEAD 表示当前版本
```

```
git reset --hard HEAD^^ #回退到上上个版本
```

```
git reset --hard HEAD~num #回退到上 num 个版本
```

```
cat <file>... #查看文件内容
```

下面我们演示一下比较难理解的

```
git reset --hard HEAD~num
```

效果如图：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git reset --hard HEAD~2
HEAD is now at 9717723 第一次提交，版本一

bibibibi@SYK MINGW64 /d/TestRepository (master)
$ cat HelloGit.txt
hello git&Github
```

可以很清晰地看见，文件已经回退到版本一了！

但是我们又后悔了怎么办？我只是玩玩的哇！

作为一款强大的版本控制应用，在 Git 中当然可以吃后悔药，我们还可以通过上述的 `reset` 命令坐时光机飞来飞去，但是我们需要找到对应“时空”的编号（即版本号），非常注意，这个时候不能用 `git log` 指令了，因为你已经不能通过它看到所有的历史信息了，如图：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git log
commit 9717723e9799bf9016228afdfb6aef6481d1c603 (HEAD -> master)
Author: ShiningSYK <1052303323@qq.com>
Date:   Wed Dec 5 18:19:31 2018 +0800

    第一次提交，版本一
```

这样的设计是合理的，因为我们期望它总能正确反映我们需要的历史纪录（回退到从前之后，我们并不能看到自己的未来）。

我们可以使用命令

`git reflog`

来查看历史版本信息，效果如图：

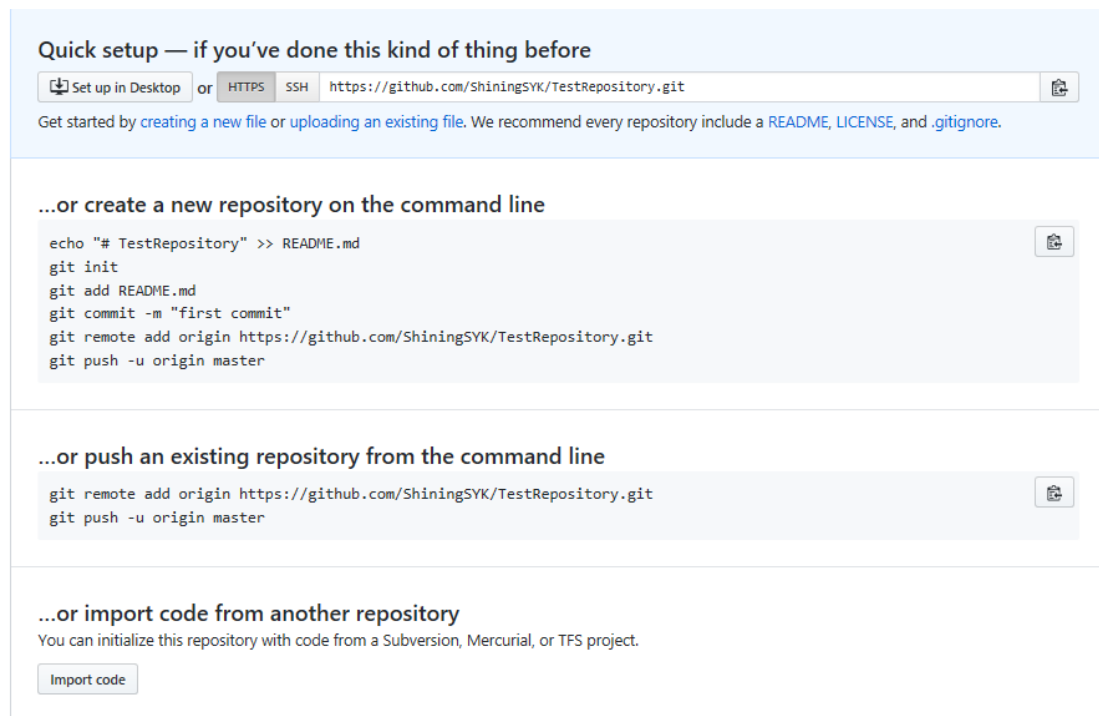
```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git reflog
9717723 (HEAD -> master) HEAD@{0}: reset: moving to HEAD~2
4b8f244 HEAD@{1}: commit: 第三次提交，版本三
b990970 HEAD@{2}: commit: 第二次提交，版本二
9717723 (HEAD -> master) HEAD@{3}: commit (initial): 第一次提交，版本一
```

现在我们又找到了其他版本对应的版本号，就可以使用 `reset` 命令来回到指定的版本了。

至此，我们已经基本实现了本地的版本库管理。

8.推送库到 Github（保证已经学过了 1. 2. 两部分内容，并且已经会使用 GitHub 创建库，如果不会请到 <https://guides.github.com/activities/hello-world/> 再学习一下）

首先我们登录 GitHub 并创建新同名库 TestRepository，结果如图：



ProTip! Use the URL for this page when adding GitHub as a remote.

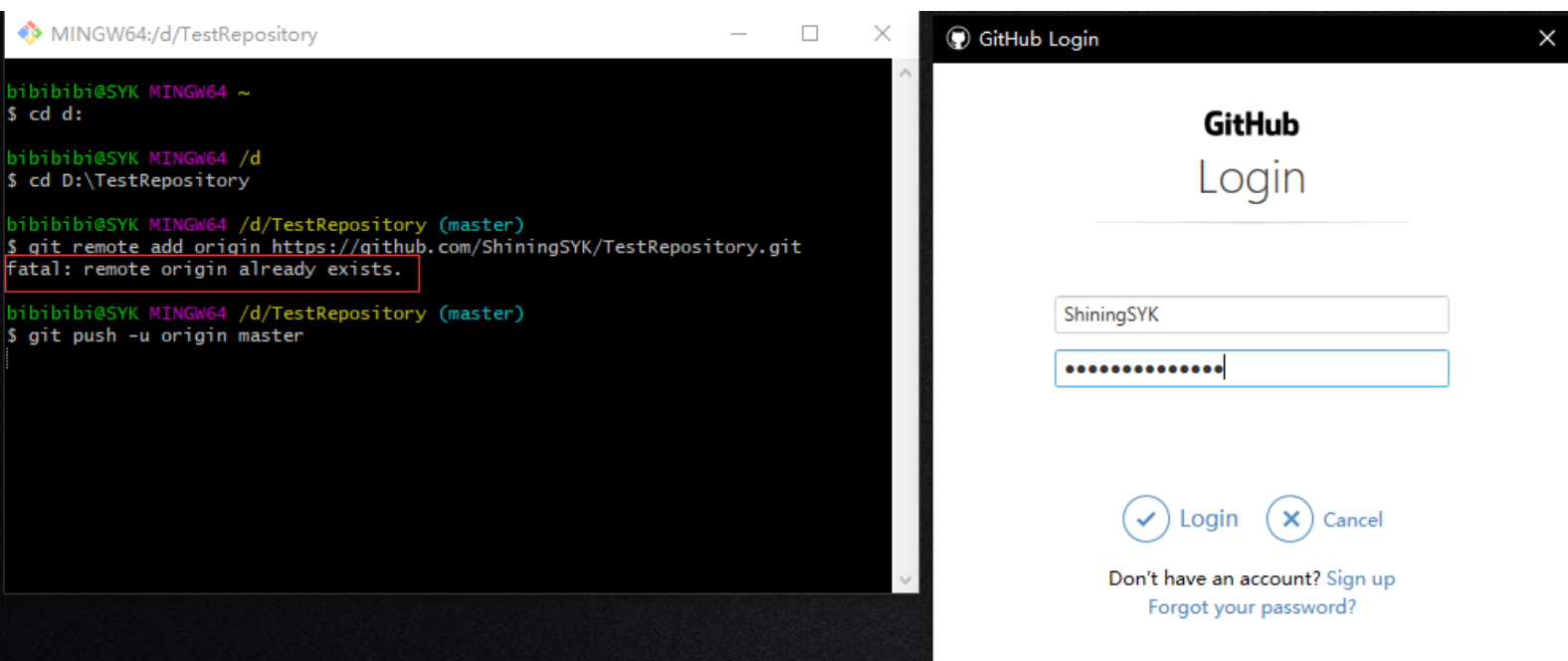
目前，在 GitHub 上的这个 `learngit` 仓库还是空的，GitHub 告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到 GitHub 仓库。

接下来我们根据它的提示来把本地库推送（push）到 GitHub 上。

在 Git Bush 中（保证当前在库对应的目录下）输入：

```
git remote add origin https://github.com/ShiningSYK/TestRepository.git  
git push -u origin master
```

得到结果如下：



说明：出现红色框内容是因为我之前已经执行过第一条命令。

指令正确执行，会弹出登录框（可能反应有点慢），输入用户名及密码后通过验证，即可成功推送到 GitHub，此时账号绑定的邮箱也会收到通知，同时刷新 GitHub 网页即可发现库已成功同步，并且在 Git Bush 中也会有如下输出信息：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 255 bytes | 127.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:   https://github.com/ShiningSYK/TestRepository/pull/new/master
remote:
To https://github.com/ShiningSYK/TestRepository.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

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。

以后再推送时，只要使用

`git push origin master`

命令即可

9.克隆库到本地

现在我们已经会了怎么推送，现在来看一下怎么克隆（clone）。

首先我们得保证 GitHub 已有一个非空库，我在这里使用了根据 GitHub guide 建造的库 hello-world（如果你学了 <https://guides.github.com/activities/hello-world/>，应该知道我在说什么）。

接下来使用指令

```
git clone git@github.com:ShiningSYK/hello-world.git
```

即可从 GitHub 中克隆库到本地，记得把 url 换成你自己的。结果如图：

```
bibibibi@SYK MINGW64 /d/TestRepository (master)
$ git clone git@github.com:ShiningSYK/hello-world.git
Cloning into 'hello-world'...
Warning: Permanently added the RSA host key for IP address '52.74.223.119' to the list of known hosts.
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
```

此时我们再到 TestRepository 的目录下，就可以看到 hello-world 库了（实际上我们应该回到 d 盘再 clone，因为库应该是并行的关系，但是本次演示并不涉及到项目的逻辑，因此这样的操作只是为了演示功能的实现过程而已）。

事实上我们还有一种方法 clone，那就是使用 Https 协议，指令更改为

```
git clone https://github.com/ShiningSYK/hello-world.git
```

两种地址可以从如下图中的 clone or download 按钮中获得：

The screenshot shows a GitHub repository page for 'ShiningSYK/hello-world'. The repository is a test repository with 3 commits, 1 branch, 0 releases, and 1 contributor. The 'Clone or download' button is highlighted in green. A dropdown menu is open, showing the 'Clone with HTTPS' option, which provides the URL 'https://github.com/shiningSYK/hello-world'. The dropdown also includes options to 'Open in Desktop' and 'Download ZIP'. The repository content shows a README.md file with the text 'hello-world' and '这是一个测试库'.

实际上，Git 支持多种协议，默认的 git:// 使用 ssh，但也可以使用 https 等其他协议。使用 https 除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放 http 端口的公司内部就无法使用 ssh 协议而只能用 https。

下图给出使用 Https 协议克隆到 d 盘下的结果：

```
bibibibi@SYK MINGW64 /d
$ git clone https://github.com/ShiningSYK/hello-world.git
Cloning into 'hello-world'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (7/7), done.
```