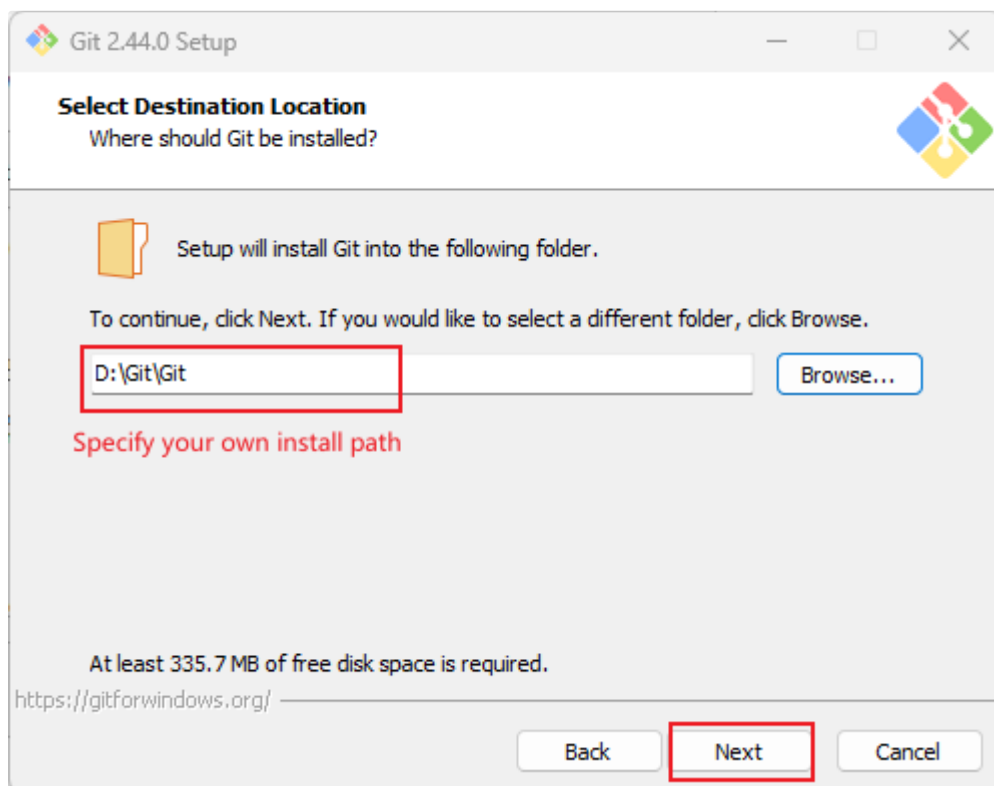
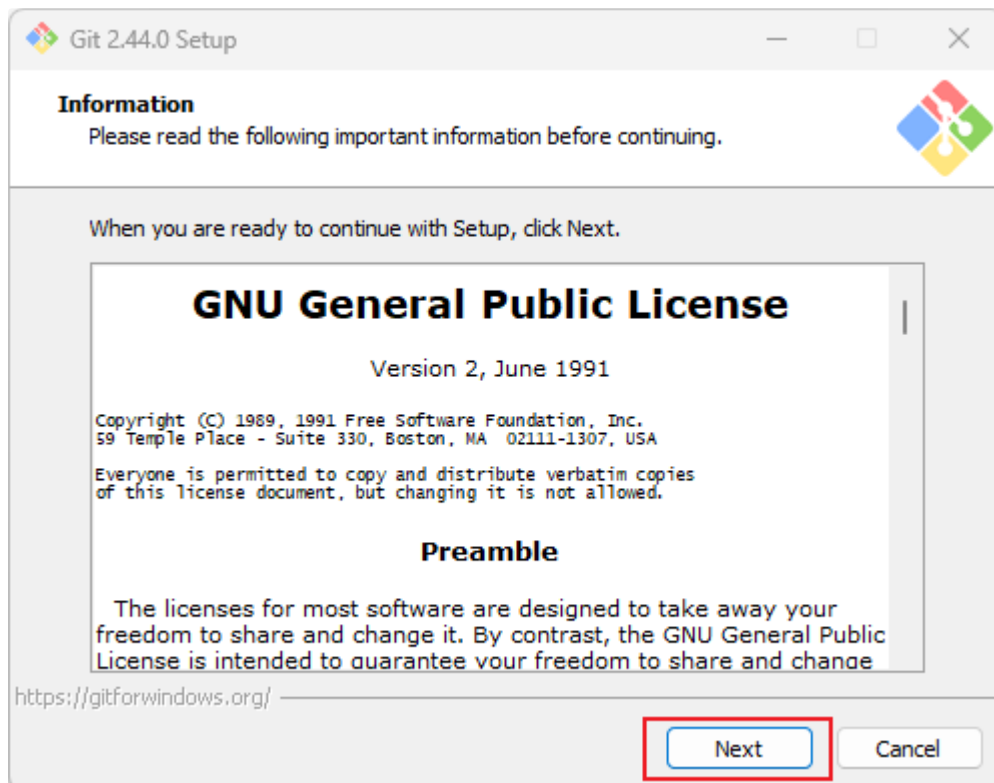
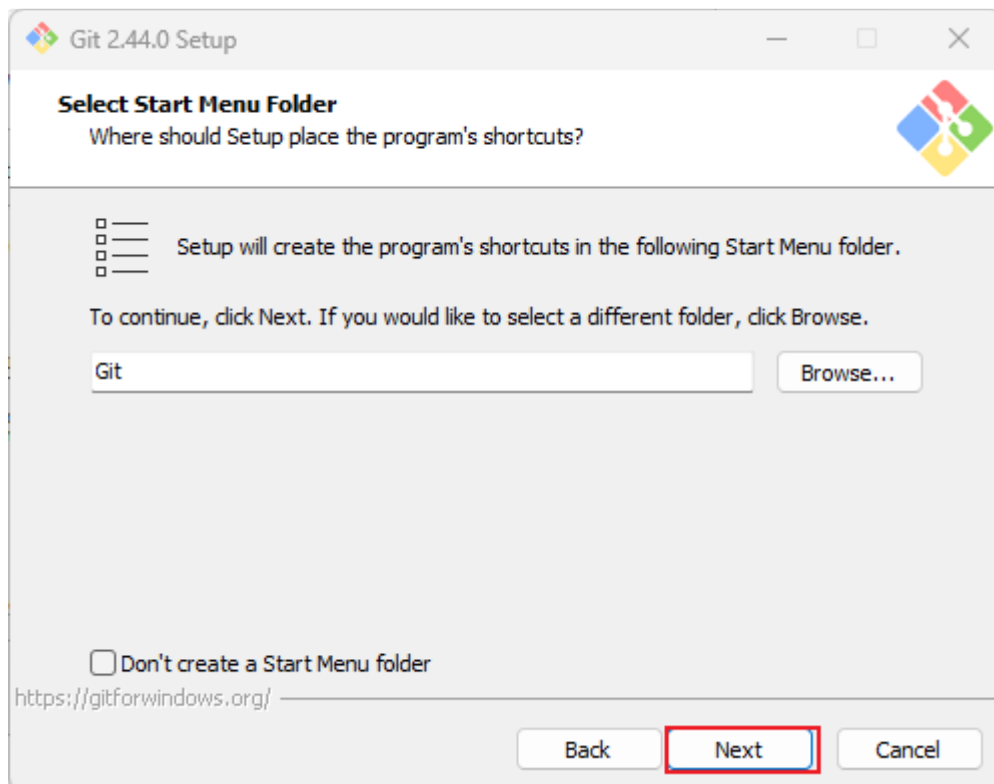
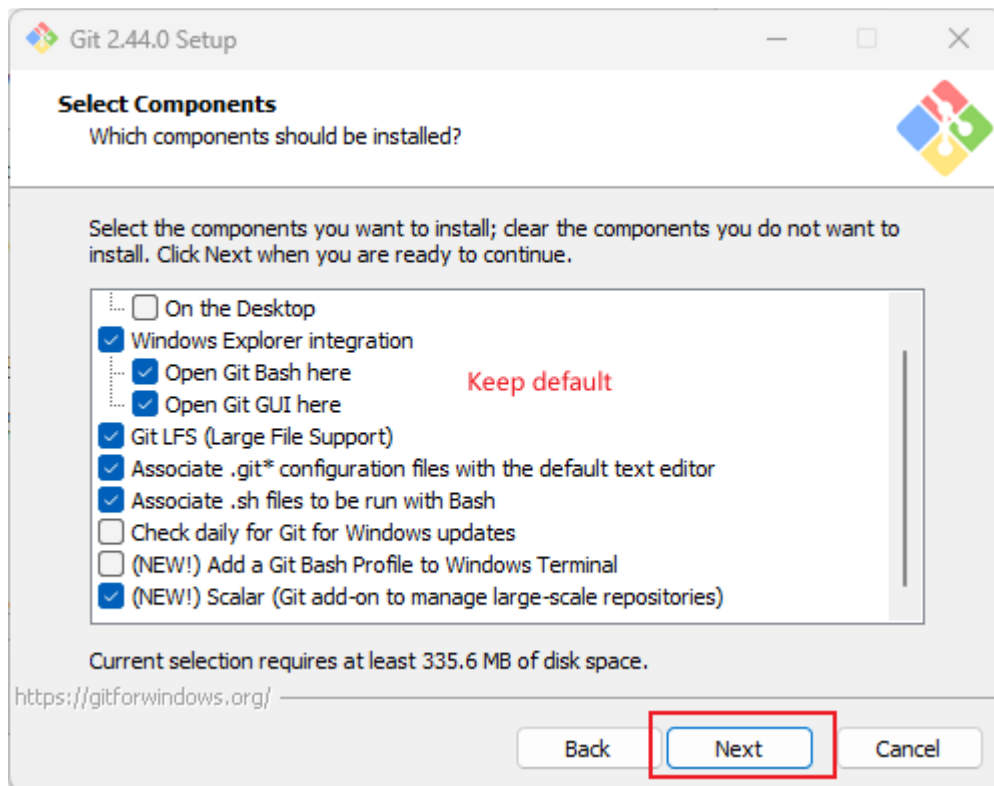


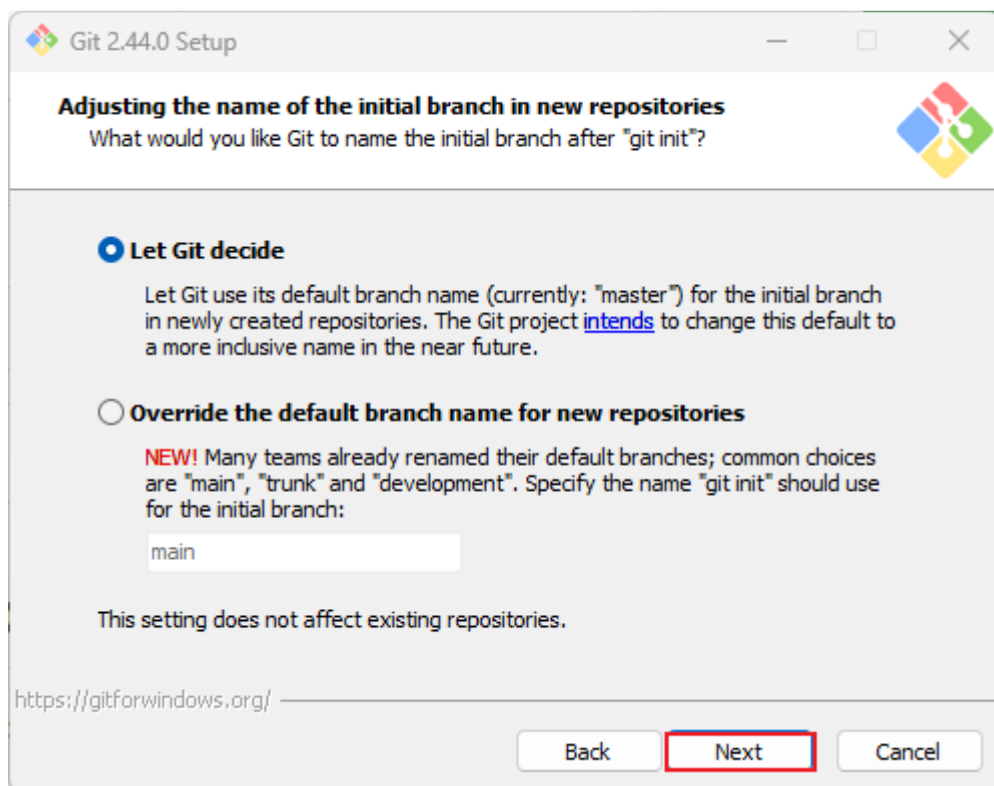
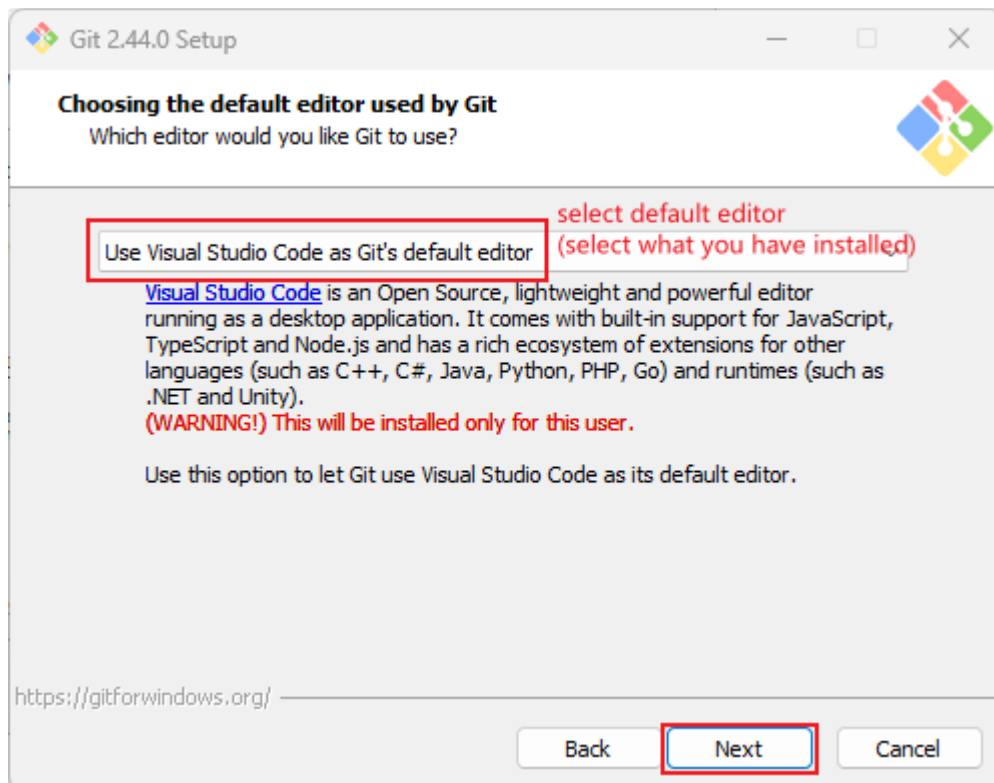
Git使用教程（Windows）


1. Git安装

- 访问[Git官网](#)获取Git最新版本安装程序;
- 双击安装程序进行安装:







 Git 2.44.0 Setup

Adjusting your PATH environment
How would you like to use Git from the command line?

☐

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

☒

Git from the command line and also from 3rd-party software
(Recommended) This option adds only some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from Git Bash, the Command Prompt and the Windows PowerShell as well as any third-party software looking for Git in PATH.

☐


Use Git and optional Unix tools from the Command Prompt
Both Git and the optional Unix tools will be added to your PATH.
Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.

<https://gitforwindows.org/>

Back

Next

Cancel

 Git 2.44.0 Setup

Choosing the SSH executable
Which Secure Shell client program would you like Git to use?

☒

Use bundled OpenSSH
This uses ssh.exe that comes with Git.

☐

Use external OpenSSH
NEW! This uses an external ssh.exe. Git will not install its own OpenSSH (and related) binaries but use them as found on the PATH.

<https://gitforwindows.org/>

Back

Next

Cancel

Git 2.44.0 Setup

Choosing HTTPS transport backend

Which SSL/TLS library would you like Git to use for HTTPS connections?

☒ **Use the OpenSSL library**

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

☐ **Use the native Windows Secure Channel library**

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.

<https://gitforwindows.org/>

Back Next Cancel

Git 2.44.0 Setup

Configuring the line ending conversions

How should Git treat line endings in text files?

☒ **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").

☐ **Checkout as-is, commit Unix-style line endings**

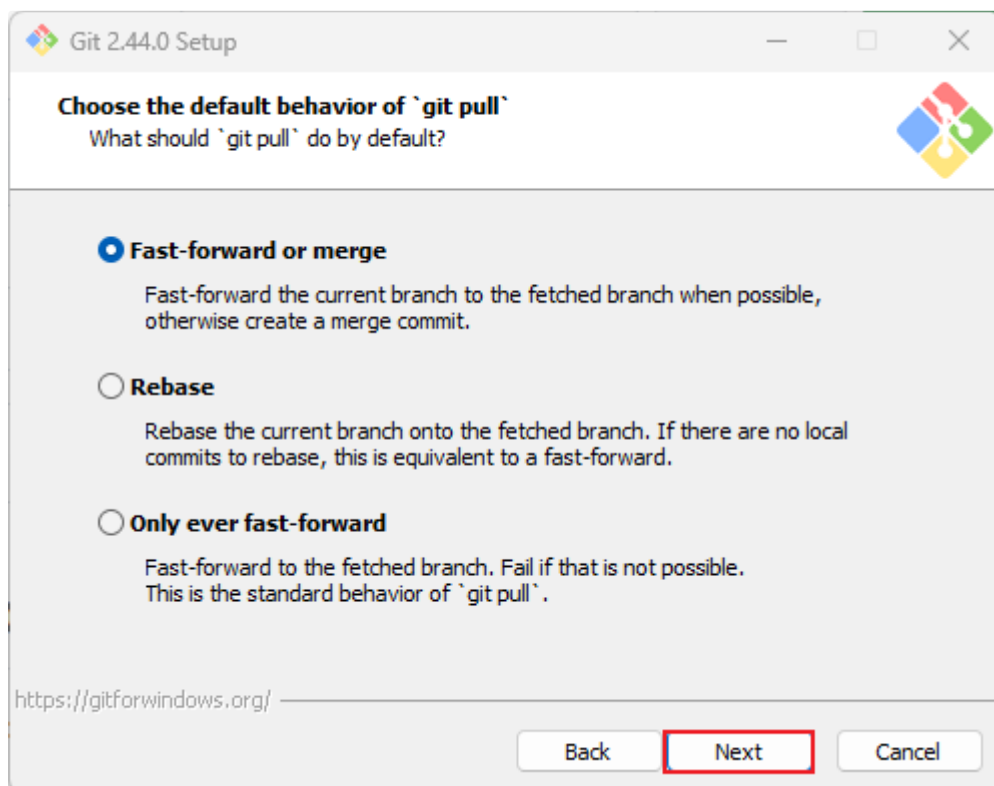
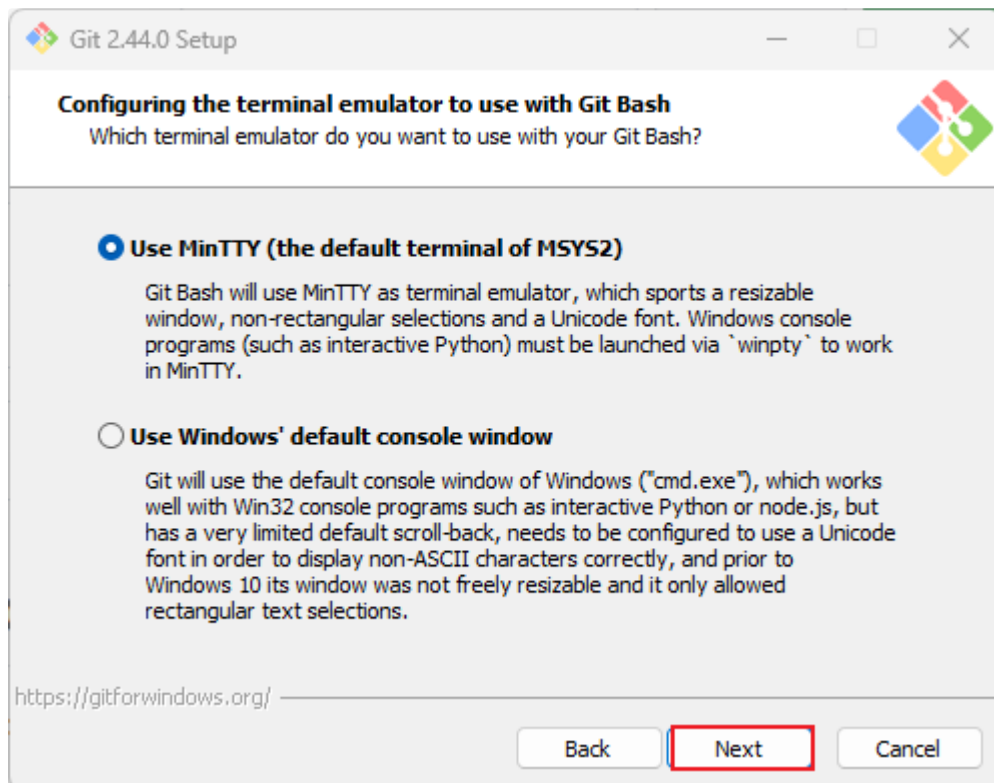
Git will not perform any conversion when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Unix ("core.autocrlf" is set to "input").

☐ **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").

<https://gitforwindows.org/>

Back Next Cancel



Git 2.44.0 Setup

Choose a credential helper
Which credential helper should be configured?

☒ **Git Credential Manager**
Use the [cross-platform Git Credential Manager](#).
See more information about the future of Git Credential Manager [here](#).

☐ **None**
Do not use a credential helper.

<https://gitforwindows.org/>

Back Next Cancel

Git 2.44.0 Setup

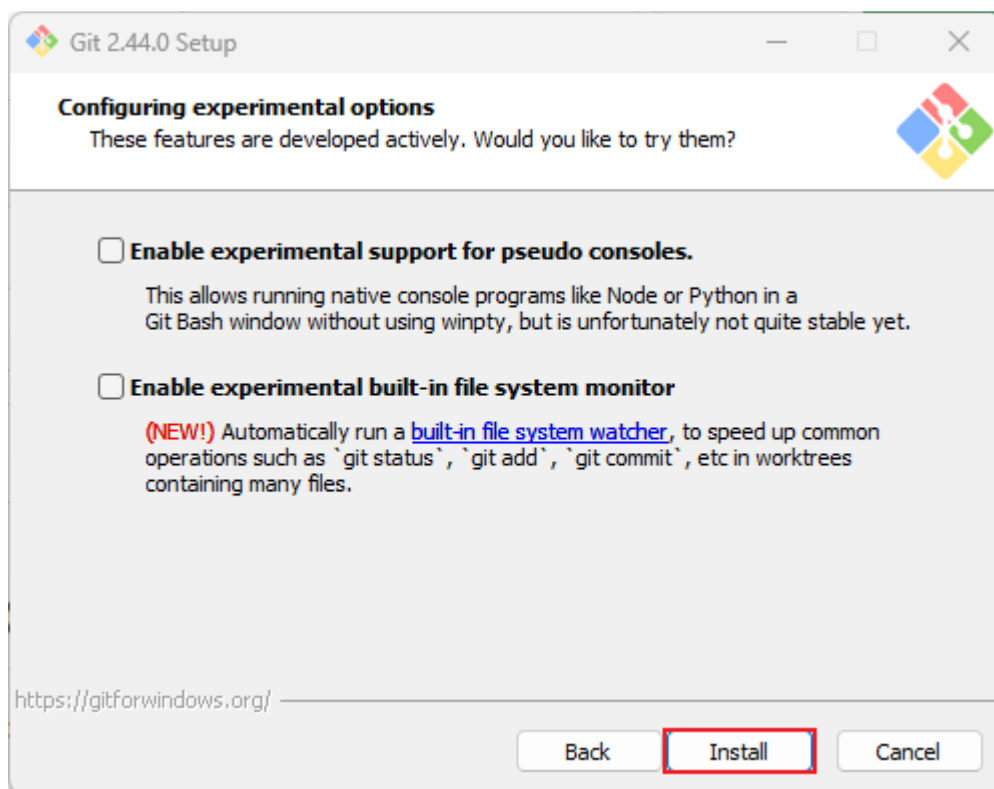
Configuring extra options
Which features would you like to enable?

☒ **Enable file system caching**
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.

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

<https://gitforwindows.org/>

Back Next Cancel



- 安装完毕即可启动Git，开始菜单（或任意文件夹下右键）一般包含了若干各相关的子程序：
 - Git Bash：Unix与Linux风格的命令行，使用最多（推荐）
 - Git CMD：Windows风格的命令行
 - Git GUI：图形界面的Git，不推荐初学者使用

2. 常用Linux命令

- `cd`：改变/进入目录，不加参数则回到根目录
- `cd ..`：回退到上一层目录
- `pwd`：显示当前所在目录路径
- `ls`，`ll`：列出当前目录下的所有文件，`ll`列出的内容更详细
- `touch`：新建一个文件
- `rm`：删除一个文件
- `mkdir`：新建一个文件夹
- `rm -r`：删除一个文件夹
- `mv`：移动文件，例如 `mv afile.txt adir`，将afile.txt移动到adir文件夹中
- `reset`：重新初始化终端/清屏

- `clear` : 清屏
- `history` : 查看历史命令
- `help` : 查看帮助
- `exit` : 退出
- `#` 表示注释

3. Git配置

- 查看所有配置: `git config -l`
- 查看不同级别的配置文件:
 - 查看系统配置: `git config --system --list`
 - 查看当前用户 (global) 配置: `git config --global --list`

所有Git配置文件都保存在本地: 系统配置文件保存于Git安装目录下 (`./etc/gitconfig`) ; 用户配置文件保存于用户文件夹下 (`C:\Users\username\.gitconfig`) , 不同Git版本路径可能不同。

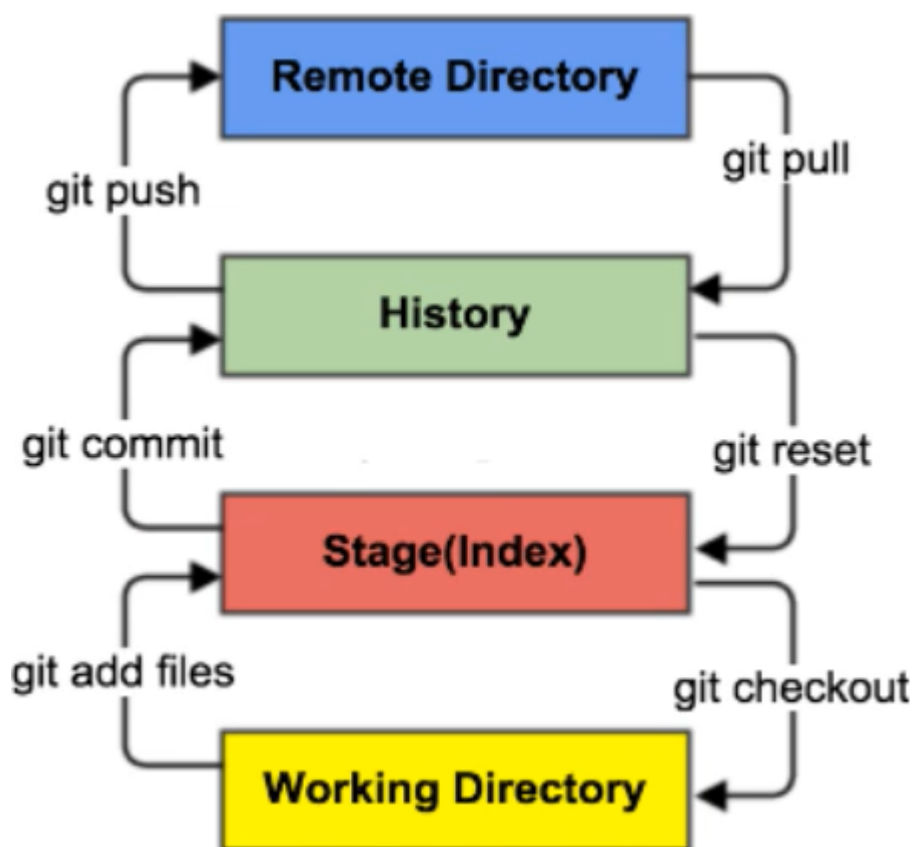
在使用Git前, 必须先配置用户名和用户邮箱, 每次Git提交都会使用该信息。

```
git config --global user.name "yourname" #配置用户名
git config --global user.email youremail@address.com #配置用户邮箱
```

使用 `--global` 选项的配置只需要进行一次, 后续Git进行的所有处理都将使用该配置信息。如果想在特定的项目中使用不同的用户名或邮箱, 可以不带 `--global` 选项运行上述命令, 为某个特定项目添加额外的配置。

4. Git工作原理

Git在本地有三个工作区域: 工作目录 (Working Directory) 、暂存区 (Stage/Index) 、资源库 (Repository/Git Directory) , 如果加上远程Git仓库 (Remote Directory) , 则包含四个区域。文件在四个工作区域的流转逻辑如下:



- 工作目录：本地存放项目代码/工程文件的位置；
- 暂存区：临时存放文件改动，本质上是一个保存了即将提交的文件列表信息的文件；
- 本地Git仓库：本地安全存放数据的位置，其中包含用户提交的所有版本的数据，其中HEAD指向最新提交进入仓库的版本；
- 远程Git仓库：远程托管代码/文件的服务器

5. Git文件的状态

Git中的文件有4种状态：

- Untracked：未跟踪，即此文件在工作目录中，但未添加到Git仓库中。通过 `git add` 命令可使其状态变为Staged。
- Staged:暂存状态，执行 `git commit` 会将其同步进Git仓库中，文件则变成Unmodify状态。
- Unmodify:文件已入库，未修改，即此文件在工作目录中与Git仓库中完全一致。
- Modified：文件已修改，未暂存未入库。

查看文件状态的命令：

```
# 查看指定文件的状态
git status <filename>

# 查看所有文件的状态
git status
```

6. Git操作命令

Git 常用命令速查表

```
master : 默认开发分支      Head : 默认开发分支
origin  : 默认远程版本库    Head^ : Head 的父提交
```

创建版本库

```
$ git clone <url>          #克隆远程版本库
$ git init                 #初始化本地版本库
```

修改和提交

```
$ git status              #查看状态
$ git diff                #查看变更内容
$ git add .               #跟踪所有改动过的文件
$ git add <file>          #跟踪指定的文件
$ git mv <old> <new>      #文件改名
$ git rm <file>           #删除文件
$ git rm --cached <file>  #停止跟踪文件但不删除
$ git commit -m "commit message"
#提交所有更新过的文件
$ git commit --amend      #修改最后一次提交
```

查看提交历史

```
$ git log                 #查看提交历史
$ git log -p <file>       #查看指定文件的提交历史
$ git blame <file>        #以列表方式查看指定文件的提交历史
```

撤销

```
$ git reset --hard HEAD  #撤销工作目录中所有未提交文件的修改内容
$ git checkout HEAD <file> #撤销指定的未提交文件的修改内容
$ git revert <commit>    #撤销指定的提交
```

分支与标签

```
$ git branch              #显示所有本地分支
$ git checkout <branch/tag> #切换到指定分支或标签
$ git branch <new-branch> #创建新分支
$ git branch -d <branch>  #删除本地分支
$ git tag                 #列出所有本地标签
$ git tag <tagname>       #基于最新提交创建标签
$ git tag -d <tagname>    #删除标签
```

合并与衍合

```
$ git merge <branch>      #合并指定分支到当前分支
$ git rebase <branch>     #衍合指定分支到当前分支
```

远程操作

```
$ git remote -v           #查看远程版本库信息
$ git remote show <remote> #查看指定远程版本库信息
$ git remote add <remote> <url>
#添加远程版本库
$ git fetch <remote>      #从远程库获取代码
$ git pull <remote> <branch> #下载代码及快速合并
$ git push <remote> <branch> #上传代码及快速合并
$ git push <remote> :<branch/tag-name>
#删除远程分支或标签
$ git push --tags         #上传所有标签
```

```
# Git Cheat Sheet <CN> (Version 0.1)   # 2012/10/26 -- by @riku < riku@gitcafe.com / http://riku.wowubuntu.com >
```

7. 忽略文件

有时我们并不想将所有文件纳入版本控制中，可以在根目录下新建".gitignore"文件，Git将根据此文件中书写的规则忽略特定文件。

- 井号（#）在此文件中表示注释

- 可以使用Linux通配符, `*` 表示任意多个字符、`?` 表示一个字符, 方括号 `[abc]` 代表可选字符范围, 大括号 `{string1,string2,...}` 代表可选字符串等
- 名称前带一个感叹号 `!` 表示例外规则, 此文件即使满足其他忽略规则也不会被忽略
- 名称前带路径分隔符 `/`, 表示要忽略的文件在此目录下
- 名称最后带路径分隔符 `/`, 表示忽略此文件夹

```
#          表示此为注释,将被Git忽略
*.a        表示忽略所有 .a 结尾的文件
!lib.a     表示但lib.a除外
/TODO      表示仅仅忽略项目根目录下的 TODO 文件, 不包括
subdir/TOD 表示
build/     表示忽略 build/目录下的所有文件, 过滤整个build文件夹;
doc/*.txt  表示会忽略doc/notes.txt但不包括 doc/server/arch.txt

bin/       表示忽略当前路径下的bin文件夹, 该文件夹下的所有内容都会被
忽略, 不忽略 bin 文件
/bin       表示忽略根目录下的bin文件
/*.c       表示忽略cat.c, 不忽略 build/cat.c
debug/*.obj 表示忽略debug/io.obj, 不忽略 debug/common/io.obj和
tools/debug/io.obj
**/foo     表示忽略/foo,a/foo,a/b/foo等
a/**/b     表示忽略a/b, a/x/b,a/x/y/b等
!/bin/run.sh 表示不忽略bin目录下的run.sh文件
*.log      表示忽略所有 .log 文件
config.php 表示忽略当前路径的 config.php 文件

/mtk/      表示过滤整个文件夹
*.zip      表示过滤所有.zip文件
/mtk/do.c  表示过滤某个具体文件

# 还有一些规则如下:
fd1/*      忽略目录 fd1 下的全部内容; 注意, 不管是根目录下的 /fd1/
目录, 还是某个子目录 /child/fd1/ 目录, 都会被忽略
/fd1/*     忽略根目录下的 /fd1/ 目录的全部内容
```

8. 远程仓库配置SSH公钥

在向远程仓库Push项目时，如果使用http的方式，则每次都需要输入账号和密码。而采取SSH公钥的方式，则可以实现远程服务器和本地主机的绑定，免去了每次都输入账号和密码的麻烦。

1. 检查本地主机是否已经存在ssh key

```
cd ~/.ssh
ls
# 如果.ssh文件夹不存在，或者.ssh文件夹没有ssh key文件，则须生成ssh key，否则跳转第3步
```

2. 生成ssh key

```
ssh-keygen -t rsa -C "xxx@xxx.com"
# 邮箱填写远程服务器账号邮箱
# 执行后一直回车即可
```

3. 获取ssh key公钥内容

```
cd ~/.ssh
cat id_rsa.pub
# 终端将打印出公钥内容，复制下来
# 带.pub后缀的为公钥，不带此后缀的为私钥（不能向外提供）
```

4. 在远程服务器中添加ssh key（如Github、Gitee等）

5. 验证是否设置成功（以Github为例）

```
ssh -T git@github.com
# 提示 "You've successfully authenticated" 表示配置成功
```

6. 设置成功后，即可不需要账号密码从远程仓库 clone 和 push 代码/文件，注意链接远程仓库时使用SSH协议地址而非HTTPS协议地址

7. 直接从远程仓库克隆文件时，本地仓库自动连接远程仓库，使用 git remote -v 查看远程仓库信息。在本机新建仓库并连接远程仓库时，步骤如下：

远程服务器上新建仓库

使用Git命令添加远程仓库

`git remote <remote> <ssh url>` # <remote>为远程仓库别名, 通常可设置为“origin”

本地仓库修改commit后, push到远程仓库

`git push <remote> <branch>` # 若未指定远程仓库分支, 可能会报错“The current branch master has no upstream branch.”