Table of Contents

资源

> git

git bash 修改时区

先查看当前时间是否正确：

```
lxw@lx MINGW64 /e/src_git/demo (develop)
$ date
Fri Jan 17 06:41:27 GMT 2025
```

修改 `TZ` 变量，注意这里时区设置和 linux 中格式有区别 (linux 中国时区为 `Asia/Shanghai`)

```
export TZ="CST-8"
```

永久修改则在配置文件中修改，如当前用户修改可在 `~/.bashrc` 中设置
修改完后 `. ~/.bashrc` 时期生效，再用 date 命令查看，时区已修改成功：

```
lxw@lx MINGW64 /e/src_git/demo (develop)
$ date
Fri Jan 17 14:46:31 CST 2025
```

git bash 中为常用命令设置别名

希望全局配置，可以在 `/etc/profile.d/aliases.sh` 中添加。
针对当前用户配置，在 `$HOME/.bashrc` 中添加。

```
alias stashsuobin='git diff --name-only | grep -E "\.(suo|bin)$" | xargs git stash push -m ".suo and .bin files" '
alias restoresuobin='git status --porcelain | cut -d" " -f3- |  grep -E "\.(suo|bin)$" | xargs git restore -- '
alias checkSkipWorktree=' git ls-files -v | grep "^S"'

alias addUnchanged='git update-index --assume-unchanged '
alias cancelAllSkipWorktree='git ls-files -v | grep "^S" | cut -d" " -f2 | xargs git update-index --no-skip-worktree '
alias cancelAllUnchangedSkipWorktree='git ls-files -v | grep "^[S|s]" | cut -d" " -f2 | xargs git update-index --no-skip-worktree '
alias cancelSkipWorktree='git update-index --no-skip-worktree '
alias cancelUnchanged='git update-index --no-assume-unchanged '
alias checkDeletedLog='git fsck --unreachable | grep commit | cut -d" " -f3 | xargs git log --merges --no-walk --oneline '
alias checkSkipWorktree='git ls-files -v | grep "^S" '
alias checkUnchangedSkipWorktree=' git ls-files -v | grep "^[S|s]" '
alias restoreStashCommit='git update-ref --create-reflog refs/stash '
```

修改 log 时区

修改时区以便用 git log 查看日志时的时间和系统时间处于一个时区：

```
git config --global log.date=local
```

git remote

查看远程仓库名称列表

```
lx@lx MINGW64 /d/Documents/git_test04 (main2)
$ git remote
origin
```

查看远程仓库URL

```
lx@lx MINGW64 /d/Documents/git_test04 (main2)
$ git remote -vv
origin  https://github.com/lxwcd/git_test.git (fetch)
origin  https://github.com/lxwcd/git_test.git (push)
```

添加本地远程仓库

```
git remote add local-origin file:///d/Documents/git_test
```

- 将 `D:/Documents/git_test` 仓库添加为远程仓库，别名为 `local-origin`。

git status 查看文件状态

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git --version
git version 2.47.1.windows.1

lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git status
On branch fix_B
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   git.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   git.md
        modified:   test01.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        0001-commit-B.patch
        0001-fix-B.patch
        0001-update-fix_B.patch
        0002-commit-C.patch
        0002-update-fix_B.patch
        1.patch
```

-s, --short 显示简短的状态信息

不显示文件的具体更改内容。

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git status -s
A  git.md
 M test01.txt
?? 0001-commit-B.patch
?? 0001-fix-B.patch
?? 0001-update-fix_B.patch
?? 0002-commit-C.patch
?? 0002-update-fix_B.patch
?? 1.patch
```

-u, --untracked-files[=] 显示未跟踪的文件

`<mode>` 可以是 `no`, `normal`, 或 `all`
- `no`：不显示未跟踪的文件。
- `normal`：显示未跟踪的文件，但排除那些在 `.gitignore` 中指定的文件。
- `all`：显示所有未跟踪的文件，包括那些在 `.gitignore` 中指定的文件。

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git status --short --untracked-files=no
A  git.md
 M test01.txt
```

输出未被跟踪的文件名

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git status -s
AM git.md
 M test01.txt
?? .gitignore
?? 0001-commit-B.patch
?? 0001-fix-B.patch
?? 0001-update-fix_B.patch
?? 0002-commit-C.patch
?? 0002-update-fix_B.patch
?? 1.patch
?? 2.txt

lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git status -s |  grep "??" | cut -d" " -f2
.gitignore
0001-commit-B.patch
0001-fix-B.patch
0001-update-fix_B.patch
0002-commit-C.patch
0002-update-fix_B.patch
1.patch
2.txt
```

git log 查看日志

查看当前分支所有提交

```
git log
```

默认按照时间顺序，从最新的开始显示。

查看特定分支的提交

```
git log <branch-name>
```

这个命令显示指定分支的提交历史。

指定输出日志数目

```
git log -3
```

输出日志显示最新的 3 条

查看提交差异 --patch

使用-p或--patch参数，可以查看每个提交的具体差异：

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git log -p -1
commit 51da54a57cdc95263072173726d187a544725289 (HEAD -> fix_B)
Author: lxwcd <15521168075@163.com>
Date:   Sun Jan 12 21:22:45 2025 +0800

    update fix_B

diff --git a/test01.txt b/test01.txt
index cd7fb11..a821b44 100644
--- a/test01.txt
+++ b/test01.txt
@@ -4,3 +4,4 @@ local modify test01.txt
 A
 b
 C
+001
diff --git a/test02.txt b/test02.txt
index 8de02e1..98bbcac 100644
--- a/test02.txt
+++ b/test02.txt
@@ -1,2 +1,3 @@
 test02
-local git rebase
\ No newline at end of file
+local git rebase002
+002
diff --git a/test05.txt b/test05.txt
new file mode 100644
index 0000000..7ed6ff8
--- /dev/null
+++ b/test05.txt
@@ -0,0 +1 @@
+5
```

这将显示最新两个提交的详细差异，包括文件的增删改。
其中 a 表示该提交前的版本，b 表示该提交后的版本。
@@ 标记差异的开始，如 @@ -4,3 +4,4 @@ local modify test01.txt 表示对于 a 版本从第 4 行开始的 3 行内容，对于 b 版本从第 4 行开始的 4 行内容，有差异。
- 表示原始版本中存在但新版本被删除的行，+ 表示原始版本没有 ， 新版本添加的行，即状态表示从 a 版本到 b 版本需要进行的添加、删除等操作。

统计信息 --stat

```
$ git log -1 --stat
commit 740e65b2fd98f0b99f3bcfd8dc8e1b8ad8bb6a3f (HEAD -> feature)
Author: lxw <15521168075@163.com>
Date:   Thu Jan 9 13:09:24 2025

    modify test.md

 demo/test.md | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```

这将显示每个提交修改的文件列表、文件数量变化以及添加和删除的行数统计。

简短 stat 信息

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git log --shortstat -1
commit 51da54a57cdc952630721737726d187a544725289 (HEAD -> fix_B)
Author: lxwcd <15521168075@163.com>
Date:   Sun Jan 12 21:22:45 2025 +0800

    update fix_B

 3 files changed, 4 insertions(+), 1 deletion(-)
```

自定义格式 --pretty

使用--pretty参数，可以自定义日志的显示格式。

哈希值 - 作者，相对日期 : message

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git log --pretty=format:"%h - %an, %ar : %s" -1
51da54a - lxwcd, 7 days ago : update fix_B
```

哈希值 - 作者，绝对日期 : message

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git log --pretty=format:"%h - %an, %ad : %s" -1
51da54a - lxwcd, Sun Jan 12 21:22:45 2025 +0800 : update fix_B
```

ASCII 图形显示历史 --graph

官方示例：

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 Ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of https://github.com/dustin/grit.git
|\
| * 420eac9 Add method for getting the current branch
* | 30e367c Timeout code and tests
* | 5a09431 Add timeout protection to grit
* | e1193f8 Support for heads with slashes in them
|/
* d6016bc Require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

时间限制

> Git - Viewing the Commit History

`--since`和`--until`参数可以用来限制显示特定时间范围内的提交：

```
$ git log --since="2 weeks ago"
```

绝对时间

```
git log --since="2024-12-01" --until="2024-12-31"
git log --since="2024-12-01 00:00:00" --until="2024-12-31 23:59:59"
```

相对时间

```
git log --since="1 week ago"
git log --until="yesterday"
git log --since="2 days ago"
git log --since="1 hour ago"
git log --since="1 minute ago"
git log --since="2 weeks ago" --until="1 week ago"
git log --since="yesterday" --until="today"
```

作者和关键词搜索

`--author`和`--grep`参数可以用来根据作者或提交信息中的关键词来过滤提交：

```
$ git log --author="Scott Chacon" --grep="version"
```

这将显示所有作者为"Scott Chacon"且提交信息中包含"version"的提交。

文件路径过滤

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git log --oneline  -- test02.txt
51da54a (HEAD -> fix_B) update fix_B
b3852e1 (origin/branch01) local git rebase  test test02.txt
e67a0f3 add test02.txt and test03.txt
332de10 update file
```

查看特定内容的日志 -S

```
$ git log --oneline -S "function_name"
8ef1913 local modify test01.txt
e67a0f3 add test02.txt and test03.txt
332de10 update file
470dcf0 add files
```

这个命令会列出所有添加或删除了 function_name 这个字符串的提交。

正则表达式筛选特定内容的日志

```
git log -G"frotz\(nitfol"
```

显示第一个父提交 --first-parent

对于有多个分支的项目，如果其他分支用 git merge 合并到主分支，后，合并到主分支上最终会产生一个合并的提交记录，该合并的提交有两个父提交，当前分支所在的合并前的最新提交为 first parent，而另一个分支的最新提交为 second parent。

```
lx@lx MINGW64 /d/Documents/git_test (main)
$ git log --oneline --graph -15
*   a00fc7a (HEAD -> main) Merge branch 'fix_B'
|\
| * 099b5e1 (fix_B) update test files' '
| * 51da54a update fix_B
| * 9aaa1c6 (branch01) fix B
| * 09035ad commit C
| * e8867c1 commit B
| * 0b4aed3 commit A
| * b3852e1 (origin/branch01) local git rebase  test test02.txt
| * 8ef1913 local modify test01.txt
| * c8d5a84 Update test01.txt  git pull
* | 03d14ae (origin/main) update main test01.txt
* | 737c5b7 commit B
|/
* e67a0f3 add test02.txt and test03.txt
* 332de10 update file
* 470dcf0 add files
```

如上面 a00fc7a 合并提交，其 first parent 为 03d14ae，其 second parent 为 099b5e。

如果只显示 first parent，则只会显示 main 分支上的提提交和最终合并到 main 上的提交：

```
lx@lx MINGW64 /d/Documents/git_test (main)
$ git log --oneline --graph --first-parent -15
* a00fc7a (HEAD -> main) Merge branch 'fix_B'
* 03d14ae (origin/main) update main test01.txt
* 737c5b7 commit B
* e67a0f3 add test02.txt and test03.txt
* 332de10 update file
* 470dcf0 add files
```

查看合并提交的多个父提交

```
lx@lx MINGW64 /d/Documents/git_test (main)
$ git show a00fc7a --shortstat
commit a00fc7a17f1e55dee84a79f4d16b4e88edb0ba00 (HEAD -> main)
Merge: 03d14ae 099b5e1
Author: lxwcd <15521168075@163.com>
Date:   Sun Jan 19 18:34:59 2025 +0800

    Merge branch 'fix_B'

 12 files changed, 322 insertions(+)
```

从 `Merge: 03d14ae 099b5e1` 可以看出，03d14ae 为 first parent, 099b5e1 为 second parent。

排除合并信息 --no-merges

`--no-merges` 选项用于排除合并信息。当使用 `--no-merges` 选项时，`git log` 只显示那些没有合并操作的提交。

所有提交：

```
lx@lx MINGW64 /d/Documents/git_test (main)
$ git log --oneline --graph --no-merges -15
* 099b5e1 (fix_B) update test files' '
* 51da54a update fix_B
* 9aaa1c6 (branch01) fix B
* 09035ad commit C
* e8867c1 commit B
* 0b4aed3 commit A
* b3852e1 (origin/branch01) local git rebase  test test02.txt
* 8ef1913 local modify test01.txt
* c8d5a84 Update test01.txt  git pull
| * 03d14ae (origin/main) update main test01.txt
| * 737c5b7 commit B
|/
* e67a0f3 add test02.txt and test03.txt
* 332de10 update file
* 470dcf0 add files
```

排除合并后的提交：

```
lx@lx MINGW64 /d/Documents/git_test (main)
$ git log --oneline --graph -15
*   a00fc7a (HEAD -> main) Merge branch 'fix_B'
|\
| * 099b5e1 (fix_B) update test files' '
| * 51da54a update fix_B
| * 9aaa1c6 (branch01) fix B
| * 09035ad commit C
| * e8867c1 commit B
| * 0b4aed3 commit A
| * b3852e1 (origin/branch01) local git rebase  test test02.txt
| * 8ef1913 local modify test01.txt
| * c8d5a84 Update test01.txt  git pull
* | 03d14ae (origin/main) update main test01.txt
* | 737c5b7 commit B
|/
* e67a0f3 add test02.txt and test03.txt
* 332de10 update file
* 470dcf0 add files
```

查看不同分支差异

```
$ git log foo bar ^baz
```

上面命令查看那些可达于 foo 或 bar 分支，但不可达于 baz 的提交。即列出那些在 foo 或 bar 分支上存在，但在 baz 分支上不存在的提交。

查看一个分支相对于另一个分支的提交差异

git log branch1..branch2

```
$ git log origin..HEAD --oneline
```

HEAD 当前分支最新提交相对于 origin 分支最新提交的提交记录，即 HEAD 有但 origin 没有的提交记录

和下面命令功能相同：

```
$ git log HEAD ^origin --oneline
```

git show

`git show` 用于展示各种 Git 对象的详细内容，包括提交（commit）、标签（tag）和分支（branch）。

查看当前分支最新提交的详细信息

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git show main --stat
commit a00fc7a17f1e55dee84a79f4d16b4e88edb0ba00 (main)
Merge: 03d14ae 099b5e1
Author: lxwcd <15521168075@163.com>
Date:   Sun Jan 19 18:34:59 2025 +0800

    Merge branch 'fix_B'

 .gitignore                 |   1 +
 0001-commit-B.patch        |  33 ++++++++++++
 0001-fix-B.patch           |  23 +++++++++
 0001-update-fix_B.patch    |  41 +++++++++++++++
 0002-commit-C.patch        |  20 ++++++++
 0002-update-fix_B.patch    |  41 +++++++++++++++
 1.patch                    |   1 +
 2.txt                      |   1 +
 git.md                     | 130 +++++++++++++++++++++++++++++++++++++++++++++++
 test01.txt                 |  28 +++++++++++
 test02.txt                 |   2 +
 test05.txt                 |   1 +
 12 files changed, 322 insertions(+)
```

查看特定提交的信息

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git log --oneline -2
099b5e1 (HEAD -> fix_B) update test files' '
51da54a update fix_B
```

查看上面第二个提交的文件名的修改情况:

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git log --oneline -2 | cut -d" " -f1 | tail -n1 | xargs git show --name-status
commit 51da54a57cdc95263072173726d187a544725289
Author: lxwcd <15521168075@163.com>
Date:   Sun Jan 12 21:22:45 2025 +0800

    update fix_B

M       test01.txt
M       test02.txt
A       test05.txt

lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git log --oneline -2
099b5e1 (HEAD -> fix_B) update test files' '
51da54a update fix_B
```

--name-only

仅显示提交中涉及的文件名列表。

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git show --name-only
commit 099b5e1194fcb305b239e1a04d1a8ddac66bdb3d (HEAD -> fix_B)
Author: lxwcd <15521168075@163.com>
Date:   Sun Jan 19 18:33:57 2025 +0800

    update test files'
    '

.gitignore
0001-commit-B.patch
0001-fix-B.patch
0001-update-fix_B.patch
0002-commit-C.patch
0002-update-fix_B.patch
1.patch
2.txt
git.md
test01.txt
```

上面显示最新提价涉及的文件名。

--name-status

显示提交中涉及的文件名以及它们的状态（新增、修改、删除）。

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git show --name-status
commit 099b5e1194fcb305b239e1a04d1a8ddac66bdb3d (HEAD -> fix_B)
Author: lxwcd <15521168075@163.com>
Date:   Sun Jan 19 18:33:57 2025 +0800

    update test files'
    '

A       .gitignore
A       0001-commit-B.patch
A       0001-fix-B.patch
A       0001-update-fix_B.patch
A       0002-commit-C.patch
A       0002-update-fix_B.patch
A       1.patch
A       2.txt
A       git.md
M       test01.txt
```

--stat

显示提交的统计信息，包括每个文件的增删行数和文件状态。

--shortstat

显示提交的简要统计信息，只包括每个文件的增删行数。

--summary

显示提交的统计信息摘要，类似于 `--stat`，但不包括每个文件的详细信息。

--patch

显示提交的差异（默认选项），展示具体的代码变化。

git diff 查看文件差异

`git diff` 的输出格式通常包括：

- **差异标记**：`+` 表示新增的行，`-` 表示删除的行。
- **文件名**：显示发生差异的文件名。
- **行号**：显示差异行的行号。
- **差异内容**：显示具体的差异内容。

差异内容显示从 a 版本到 b 版本需要做的修改。

比较工作区和暂存区的差异

```
git diff
```

这个命令显示自上次提交以来**未暂存**的更改。
不包括没有被跟踪的文件。
如果文件已暂存，不会查看到。

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff
warning: in the working copy of 'test01.txt', LF will be replaced by CRLF the next time Git t
ouches it
diff --git a/test01.txt b/test01.txt
index cd7fb11..a821b44 100644
--- a/test01.txt
+++ b/test01.txt
@@ -4,3 +4,4 @@ local modify test01.txt
 A
 b
 C
+001
```

- a 最新提交版本
- b 表示当前工作目录未暂存的版本
- `100644` 中 `100` 表示文件类型为普通文件，`644` 表示文件权限，可以通过 `ll` 查看：

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ ll test01.txt
-rw-r--r-- 1 lx 197121 56 12月 21 22:10 test01.txt
```

- `@@ -4,3 +4,4 @@ local modify test01.txt`

a 版本的修改从第 4 行开始，共 3 行
b 版本的修改为第 4 行开始，共 4 行

- 001 表示 a 版本需要增加改行才能和 b 版本一致

比较已暂存的文件和最新提交的差异

```
git diff --cached
```

或者

```
git diff --staged
```

这些命令显示已暂存的更改与上次提交的差异。
不会查看到没有暂存的文件差异。

将工作目录的修改 add 到暂存区后，查看：

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git status
On branch fix_B
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   test01.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test05.txt


lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff

lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff --cached
diff --git a/test01.txt b/test01.txt
index cd7fb11..a821b44 100644
--- a/test01.txt
+++ b/test01.txt
@@ -4,3 +4,4 @@ local modify test01.txt
 A
 b
 C
+001
```

a 表示最新的提交版本
b 表示暂存区的版本

比较工作区和最新提交的差异

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff HEAD
warning: in the working copy of 'test02.txt', LF will be replaced by CRLF the next time Git t
ouches it
diff --git a/test01.txt b/test01.txt
index cd7fb11..a821b44 100644
--- a/test01.txt
+++ b/test01.txt
@@ -4,3 +4,4 @@ local modify test01.txt
 A
 b
 C
+001
diff --git a/test02.txt b/test02.txt
index 8de02e1..98bbcac 100644
--- a/test02.txt
+++ b/test02.txt
@@ -1,2 +1,3 @@
 test02
-local git rebase
\ No newline at end of file
+local git rebase002
+002
```

工作区中跟踪的文件，已暂存和未暂存的文件和最新提交的差异都能看到。
a 表示最新的提交版本
b 表示工作目录的版本

比较当前工作目录中特定文件和最新提交的差异

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff HEAD -- test01.txt
diff --git a/test01.txt b/test01.txt
index cd7fb11..a821b44 100644
--- a/test01.txt
+++ b/test01.txt
@@ -4,3 +4,4 @@ local modify test01.txt
 A
 b
 C
+001
```

a 表示最新的提交版本
b 表示工作目录的版本
指定查看 test01.txt 文件和 HEAD 的差异。

比较当前工作目录和任意提交的差异

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff 332de10
diff --git a/test01.txt b/test01.txt
index 4c19859..a821b44 100644
--- a/test01.txt
+++ b/test01.txt
@@ -1 +1,7 @@
 test01
+git pull
+local modify test01.txt
+A
+b
+C
+001
```

a 为指定的提交版本
b 为当前工作目录，包括为暂存的修改，不包括未跟踪的文件

比较当前已暂存和任意提交的差异

```
$ git diff 332de10 --cached
diff --git a/test01.txt b/test01.txt
index 4c19859..a821b44 100644
--- a/test01.txt
+++ b/test01.txt
@@ -1 +1,7 @@
 test01
+git pull
+local modify test01.txt
+A
+b
+C
+001
```

a 为指定的提交版本
b 为当前工作目录已暂存的文件修改

比较两个提交

```
git diff <commit1> <commit2>
```

这个命令比较两个提交之间的差异。顺序不同则结果不同。

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff 332de10 HEAD
diff --git a/test01.txt b/test01.txt
index 4c19859..cd7fb11 100644
--- a/test01.txt
+++ b/test01.txt
@@ -1 +1,6 @@
 test01
+git pull
+local modify test01.txt
+A
+b
+C
```

a 为 332de10 提交版本
b 为当前分支最新提交。

如果调换顺序：

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff HEAD 332de10
diff --git a/test01.txt b/test01.txt
index cd7fb11..4c19859 100644
--- a/test01.txt
+++ b/test01.txt
@@ -1,6 +1 @@
 test01
-git pull
-local modify test01.txt
-A
-b
-C
```

a 为当前分支最新提交。
b 为 332de10 提交版本
相当于 332de10 相对于 HEAD 的变化，因此 HEAD 中增加的内容前面为 -，表示需要减去这些内容才能和 a 的版本一致。

比较当前最新提交和上一次提交的差异

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff HEAD^ HEAD
diff --git a/test01.txt b/test01.txt
index 5c232c3..cd7fb11 100644
--- a/test01.txt
+++ b/test01.txt
@@ -2,5 +2,5 @@ test01
 git pull
 local modify test01.txt
 A
-B
+b
 C
```

a 为 HEAD^ 上一次提交版本
b 为 HEAD 最新提交版本

比较两个分支最新提交的差异

```
git diff <branch1> <branch2>
```

或者等价于：

```
git diff <branch1>..<branch2>
```

这个顺序则 a 为 branch1 版本，b 为 branch2。
查看的是两个分支的最新提交的差异。

如果调换顺序，则 a 和 b 的版本也调换：

```
git diff <branch2> <branch1>
```

这个顺序则 a 为 branch2 版本，b 为 branch1。

比较一个分支相对于另一个分支的差异

```
git diff <branch1>...<branch2>
```

这个命令显示从 `branch1` 和 `branch2` 的共同祖先到 `branch2` 的所有差异。
即从两个分支开始分叉后，branch2 上所有的提交内容相对共同祖先的差异。
查看差异中 a 为两个分支共同的祖先，b 为 branch2 最新提交。

注意和 `git diff <branch1>..<branch2>` 的区别，两个点号表示两个分支最新提交的差异。

查看差异的文件名

```
$ git diff --name-only
```

比较工作目录和 **stash** 中特定文件差别

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (main)
$ echo "000 modify after stash test01" >> test01.txt

lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (main)
$ git diff stash@{0} -- test01.txt
warning: in the working copy of 'test01.txt', LF will be replaced by CRLF the next time Git t
ouches it
diff --git a/test01.txt b/test01.txt
index d494af0..86e607d 100644
--- a/test01.txt
+++ b/test01.txt
@@ -4,4 +4,4 @@ local modify test01.txt
 A
 B
 add main test01.txt
-stash test01.txt
+000 modify after stash test01
```

a 为 stash@{0} 的版本
b 为当前工作目录

比较暂存区和 **stash** 中特定文件差别

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (main)
$ git diff stash@{0} --cached -- test01.txt
diff --git a/test01.txt b/test01.txt
index d494af0..9d86808 100644
--- a/test01.txt
+++ b/test01.txt
@@ -4,4 +4,3 @@ local modify test01.txt
 A
 B
 add main test01.txt
-stash test01.txt
```

a 为 stash@{0}
b 为暂存区

查看当前最新提交和 **stash** 的差异

```
git diff stash@{0} HEAD
```

a 为 stash@{0}
b 为HEAD

查看两个分支某个文件的差异

```
git diff <branch1> <branch2> -- <file-path>
```

要查看两个分支中某个文件夹的差异：

```
git diff <branch1> <branch2> -- <folder-path>
```

比较标签

```
git diff <tag1> <tag2>
```

这个命令比较两个标签之间的差异。

git diff 导出补丁文件

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff HEAD^ HEAD -- test01.txt
diff --git a/test01.txt b/test01.txt
index cd7fb11..a821b44 100644
--- a/test01.txt
+++ b/test01.txt
@@ -4,3 +4,4 @@ local modify test01.txt
 A
 b
 C
+001

lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git diff HEAD^ HEAD -- test01.txt > ../test01.patch
```

将一个仓库中的某个文件的最新修改生产补丁文件。

在另一个仓库应用该补丁文件：

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test_02 (fix_B)
$ cat test01.txt
test01
git pull
local modify test01.txt
A
b
C

lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test_02 (fix_B)
$ git apply ../test01.patch
```

git format-patch 生成补丁文件

生成最近一次提交的补丁文件

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git format-patch HEAD^
0001-update-fix_B.patch
```

生成最近两次提交的补丁文件

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git log --oneline -2
51da54a (HEAD -> fix_B) update fix_B
9aaa1c6 (branch01) fix B
```

```
lx@LAPTOP-VB238NKA MINGW64 /d/Documents/git_test (fix_B)
$ git format-patch HEAD^^
0001-fix-B.patch
0002-update-fix_B.patch
```

生成指定提交范围的补丁文件

```
git format-patch <start-commit>..<end-commit>
```

这条命令会生成从 `<start-commit>` 到 `<end-commit>` 之间的所有提交的补丁文件。
例如，生成从 `abc123` 到 `def456` 之间的所有提交的补丁文件：
但不包括 start-commit 那个提交。

如果希望包括起点和终点两个提交，则如下：

```
lx@LAPTOP-VB238NKA MINGW64 /e/src_git/demo (develop)
$ git format-patch --output-directory=../patch c433384cd^..910b59afe
../patch/0001-modify-test01.md.patch
../patch/0002-modify-test02.patch
../patch/0003-modify-test03.patch
```

生成某个提交以来的所有补丁文件

```
git format-patch <commit>
```

这条命令会生成从指定提交以来的所有提交的补丁文件，但不包括指定的提交。

生成从根到某个提交的所有补丁文件

```
git format-patch --root <commit>
```

这条命令会生成从仓库的根到指定提交的所有补丁文件。

输出格式选项

输出到标准输出

```
git format-patch --stdout <commit> > output.patch
```

这条命令会将补丁文件输出到标准输出，并重定向到 `output.patch` 文件中。

以原始格式输出

```
git format-patch --raw <commit>
```

这条命令会以原始格式输出补丁文件，适合向非 Git 存储库应用补丁。

按顺序编号补丁

```
git format-patch --numbered <commit>
```

使用 --subject-prefix 自定义补丁文件前缀

`--subject-prefix` 选项可以自定义补丁文件名的前缀。默认前缀是 `[PATCH]`，但你可以通过这个选项更改它。

命令：

```
git format-patch --subject-prefix="MY_PATCH" <commit>
```

这条命令会生成补丁文件，文件名前缀为 `MY_PATCH`。例如，生成的文件名可能是 `0001-MY_PATCH-commit-message.patch`。

使用 --output-directory 自定补丁文件目录

`--output-directory` 选项可以指定生成的补丁文件的保存目录。

```
git format-patch --subject-prefix="MY_PATCH" --output-directory=/path/to/patches --suffix=.txt
HEAD^
```

这条命令会生成从 `HEAD^` 到 `HEAD` 之间的所有提交的补丁文件，文件名前缀为 `MY_PATCH`，后缀为 `.txt`，并保存到 `/path/to/patches` 目录中。生成的文件名可能是 `0001-MY_PATCH-commit-message.txt`。

使用 --numbered-files 生成仅包含数字的文件名

`--numbered-files` 选项可以生成仅包含数字的文件名，不包含提交信息。

```
git format-patch --numbered-files <commit>
```

这条命令会生成补丁文件，文件名仅为数字，例如 `0001.patch`、`0002.patch` 等。

使用 --suffix 指定补丁文件后缀

`--suffix` 选项可以自定义补丁文件的后缀名。默认后缀名是 `.patch`，但你可以通过这个选项更改它。

```
git format-patch --suffix=.txt <commit>
```

这条命令会生成补丁文件，文件名后缀为 `.txt`，例如 `0001-commit-message.txt`。

应用补丁文件

git apply 应用补丁文件

```
git apply /path/to/mypatch.patch
```

这条命令会将 `mypatch.patch` 文件中的更改应用到当前工作目录中。
如果应用成功，会看到提示信息。

如果补丁文件有多个，在一个目录中，应用时不能指定目录，需要遍历里面的文件来应用：

```
for patch in ../patch/*.patch; do
    git apply "$patch"
done
```

git am 应用补丁文件

```
git am /path/to/mypatch.patch
```

这条命令会将 `mypatch.patch` 文件作为新的提交应用到当前分支中。
如果补丁文件应用成功，Git 会自动创建一个新的提交，其中包含补丁中的更改。

git blame 查看文件每行的最新提交信息

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ cat test02.txt
test02
local git rebase002
002
2
```

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git blame test02.txt
e67a0f30 (lxwcd             2024-12-15 19:44:40 +0800 1) test02
51da54a5 (lxwcd             2025-01-12 21:22:45 +0800 2) local git rebase002
51da54a5 (lxwcd             2025-01-12 21:22:45 +0800 3) 002
00000000 (Not Committed Yet 2025-01-19 20:18:25 +0800 4) 2
```

git checkout

`git checkout` 是 Git 中用于切换分支或检出特定版本的文件到工作目录的命令。

切换分支

```
git checkout <branch-name>
```

如果分支已存在，则切换到该分支。
如果分支不存在，但分支名在远程仓库存在，则创建并切换到该分支，且设置本地该分支跟踪远程对应名字的分支，相当于执行 `git checkout -b <branch-name> origin/<branch-name>`。

创建新分支并切换

```
git checkout -b <new-branch-name>
```

基于远程分支创建新分支并切换

```
git checkout -b <new-branch-name> origin/<branch-name>
```

这个命令会创建一个新的分支，并将其设置为跟踪远程分支 `origin/<branch-name>`。

检出特定文件到工作目录

```
git checkout <branch-name> -- <file-path>
```

这个命令会从 `<branch-name>` 分支检出 `<file-path>` 文件到当前工作目录，替换本地的文件。

检出特定提交到工作目录

```
git checkout <commit-hash> -- <file-path>
```

这个命令会从 `<commit-hash>` 提交检出 `<file-path>` 文件到当前工作目录。

检出特定提交到新分支

```
git checkout <commit-hash> -b <new-branch-name>
```

这个命令会创建一个新的分支 `<new-branch-name>` 并检出 `<commit-hash>` 提交的内容到这个新分支。

恢复已修改但未暂存的文件

```
git checkout -- <file-path>
```

这个命令会将 <file-path> 文件恢复到最近一次提交的状态，放弃本地的修改。
检出最新提交的相应文件替换当前工作目录的文件。

git branch

创建分支

```
git branch <branch-name>
```

这个命令会创建一个新分支，但不会切换到该分支。

列出所有本地分支

```
git branch
```

列出所有远程分支

```
git branch -r
```

列出所有本地和远程分支

```
git branch -a
```

显示当前分支

```
git branch --show-current
```

删除分支

```
git branch -d <branch-name>
```

这个命令会删除一个已经完全合并到当前分支的本地分支。如果分支未完全合并，Git 会阻止删除以防止数据丢失。

强制删除分支

```
git branch -D <branch-name>
```

这个命令会强制删除一个分支，无论它是否已经合并。

重命名分支

```
git branch -m <old-name> <new-name>
```

强制重命名分支

```
git branch -M <old-name> <new-name>
```

设置上游分支

```
git branch -u <remote-branch>
```

查看当前分支与上游分支的对应关系

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git branch -vv
  branch01  9aaa1c6 [origin/branch01: ahead 4] fix B
  feature01 b387cb1 [origin/tb01: ahead 2] Merge branch 'tb01' of https://github.com/lxwcd/git_test into feature01
  feature02 f3f08ca add test04.txt
* fix_B     f54dd26 [origin/fix_B] update 2.txt 222
  main      a00fc7a [origin/main: ahead 10] Merge branch 'fix_B'
  tb01      f9e71d6 [origin/tb01] Update test01.txt
```

- 显示所有分支及其上游信息
- * 表示当前分支的对应关系
- 显示与远程分支的 ahead 和 behind 的信息

查看当前分支与上游分支的对应关系

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git branch -vv | grep "*"
* fix_B     f54dd26 [origin/fix_B] update 2.txt 222
```

包含已合并/未合并信息

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git branch
  branch01
  fix_B
* main
  tb01

lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git branch --merged
* main

lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git branch --no-merged
  branch01
  fix_B
  tb01
```

将 `branch01` 分支合并到 `main` 分支后查看：

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git branch --merged
  branch01
* main
```

删除远程跟踪分支

```
git branch -dr <remote/branch>
```

这条命令会删除本地的远程跟踪分支。但并不会直接删除远程仓库中的分支，只是删除了本地对远程分支的跟踪信息。

删除本地分支的上游分支设置

```
git branch --unset-upstream my-branch
```

git switch

如果工作目录中有未被跟踪的文件，可以切换，未被跟踪的文件也会出现在切换后的分支上。

切换到已存在的分支

```
git switch <branch-name>
```

创建并切换新分支

```
git switch -c <new-branch-name>
```

强制创建新分支

```
git switch -C <new-branch>
```

根据远程分支创建本地分支

使用 `git switch --track` 命令创建一个新的本地分支，并设置它跟踪远程分支。

```
git switch --track origin/feature
```

或者使用 `-t` 选项：

```
git switch -t origin/feature
```

创建一个新的本地 `feature` 分支，并立即设置它跟踪远程的 `origin/feature` 分支。

也可以使用下面方法：

```
git checkout -b feature origin/feature
```

快速切换回前一个分支

```
git switch -
```

git add

暂存所有更改

```
git add .
```

或者

```
git add --all
```

使用通配符

```
git add *.cpp *.h
```

交互式暂存

```
git add -i
```

git commit

指定提交信息 -m

```
git commit -m "Commit message"
```

使用 `-m` 选项可以直接在命令行中指定提交信息。

提交所有更改 -am

```
git commit -a -m
```

使用 `-a` 选项会自动将所有已跟踪文件的更改添加到暂存区并提交，但不包括新文件。

修改最后一次提交 --amend

提交后如果发现有内容想变更，但不想新建提交记录，则可以修改后更新最后一次提交。
如果修改提交内容后不更新提交日志，可以加 `--no-edit` 选项。

```
git commit --amend --no-edit
```

最好在未将最后一次提交推送到远程仓库前使用这个命令修改。
如果已经推送到远程，则修改后需要用 `git push --force` 来更新远程仓库。如果此时远程仓库有其他新的提交，将不会存在，变成和本地一样的提交记录。

输入多行提交日志

利用 here 字符串输入多行日志：

```
$ git commit -F - <<EOF
> modify test.md
> EOF
[feature c433384cd] modify test.md
 1 file changed, 1 insertion(+), 1 deletion(-)
```

git rm 删除文件

删除文件

```
git rm <file1> <file2> ...
```

删除目录

```
git rm -r <directory>
```

git mv 移动文件

重命名文件

```
git mv <old-name> <new-name>
```

移动文件

```
git mv <file> <directory>
```

将 `<file>` 移动到 `<directory>` 目录中。

移动目录

```
git mv <old-directory> <new-directory>
```

git restore

```
git restore [<options>] [--source=<tree>] [--staged] [--worktree] [--] <pathspec>…
```

指定恢复位置

默认不指定则恢复工作目录。
指定 `--staged` 则仅恢复暂存区。
同时指定 `--staged --worktree` 则恢复工作目录和暂存区。

恢复暂存区的特定文件

已暂存的文件，希望取消暂存，但保持文件在工作目录中不变：

```
git restore --staged hello.c
```

恢复暂存区的全部文件

```
git restore --staged .
```

恢复暂存区的部分文件

```
git restore --staged *.cpp
```

恢复暂存区和工作目录的全部已跟踪的文件

```
$ git restore --source=HEAD --staged --worktree .
```

不会修改未跟踪的文件

从其他提交中恢复文件

```
$ git restore --source=origin/main~2 test01.txt
```

将指定文件恢复到 `origin/main` 分支的当前提交的前 2 个提交的版本，且恢复的是工作目录，不影响暂存区。

git revert

`git revert` 用于撤销之前的提交。
它创建一个新的提交，这个提交的内容是前一个提交的逆操作，即"反做"之前的提交。
这个操作是安全的，因为它不会改变项目的历史记录，而是在历史记录的基础上新增一个提交来表示撤销操作。

撤销最新的提交

```
git revert HEAD
```

撤销特定的提交

```
lx@lx MINGW64 /d/Documents/git_test04 (main2)
$ git log --oneline -5
dc76ad7 (HEAD -> main2) Revert "update test01.txt: add main"
16ac277 update test01.txt: add main
03d14ae (origin/main, origin/HEAD, main) update main test01.txt
737c5b7 commit B
e67a0f3 add test02.txt and test03.txt
```

如撤销上面最后一个提交，执行以下命令：

```
lx@lx MINGW64 /d/Documents/git_test04 (main2)
$ git log --oneline -5 | tail -n1 | cut -d" " -f1
e67a0f3

lx@lx MINGW64 /d/Documents/git_test04 (main2)
$ git log --oneline -5 | tail -n1 | cut -d" " -f1 | xargs git revert
[main2 b93b033] Revert "add test02.txt and test03.txt"
 Date: Mon Jan 20 20:50:14 2025 +0800
 2 files changed, 1 insertion(+), 2 deletions(-)
 delete mode 100644 test03.txt
```

撤销后查看日志，多了一个撤销的记录：

```
lx@lx MINGW64 /d/Documents/git_test04 (main2)
$ git log --oneline -7
b93b033 (HEAD -> main2) Revert "add test02.txt and test03.txt"
dc76ad7 Revert "update test01.txt: add main"
16ac277 update test01.txt: add main
03d14ae (origin/main, origin/HEAD, main) update main test01.txt
737c5b7 commit B
e67a0f3 add test02.txt and test03.txt
332de10 update file
```

撤销一系列提交

- `git revert <commit>^..<commit>`：撤销从第一个提交到第二个提交之间的所有提交。
- 不包括起始的提交

撤销操作但不创建提交

```
git revert --no-commit abc123
```

撤销操作并编辑提交信息：

```
git revert --edit abc123
```

git stash

git stash 存放位置

> The latest stash you created is stored in refs/stash; older stashes are found in the reflog of this reference and can be named using the usual reflog syntax (e.g. stash@{0} is the most recently created stash, stash@{1} is the one before it, stash@{2.hours.ago} is also possible). Stashes may also be referenced by specifying just the stash index (e.g. the integer n is equivalent to stash@{n}).

```
$ cat .git/refs/stash
07929627e841f63c9c306a647df4e84c32ae6de2

$ git stash list
stash@{0}: modify test.md
stash@{1}: modify 1.md
stash@{2}: modify 2.md
stash@{3}: modify 3.md
```

git stash push 保存工作状态

git stash 和 git stash push 效果相同。

> Save your local modifications to a new stash entry and roll them back to HEAD (in the working tree and in the index). The part is optional and gives the description along with the stashed state.

不会保存没有被跟踪的文件。

-m 或 --message 添加描述

```
git stash push -m "WIP: Implement login feature"
```

-p 或 --patch 交互存储

```
git stash push -p
```

-k 或 --keep-index 保留暂存区的更改

只会将工作区的更改保存到 stash 中，而暂存区的更改将被保留。

```
git stash push -k
```

-u 或 --include-untracked 包含未跟踪的文件

默认情况下，`git stash push` 只会保存已被追踪的文件的更改。

-a 或 --all 保存全部文件

这个选项会保存所有文件的更改，包括未跟踪的文件和被 `.gitignore` 忽略的文件。

```
git stash push -a
```

-q 或 --quiet 静默执行

```
git stash push -q
```

使用这个选项后，命令执行时不会输出任何信息。

`--pathspec-from-file=<file>` 从文件读取

这个选项允许从文件中读取路径规范，而不是从命令行参数中读取。如果文件内容是 `-`，则从标准输入读取。

```
git stash push --pathspec-from-file=pathspecs.txt
```

`--`

这个选项用于消除歧义，将路径规范与选项分开。

```
git stash push -- path/to/file
```

使用这个选项后，`path/to/file` 会被视为路径规范，而不是选项。

git stash save

```
git stash save "optional message"
```

这个命令会保存当前的工作状态到一个 stash 中，并清理工作目录。如果省略 `"optional message"`，Git 会自动生成一个消息。不会保存未被跟踪的文件。

git stash list 列出所有 stash

```
git stash list
```

这个命令会列出所有的 stash，每个 stash 前面都有一个标识符，如 `stash@{0}`。

git stash apply 应用 stash

```
git stash apply
```

这个命令会应用最近的 stash 到当前工作目录。

也可以指定一个 stash 来应用：

```
git stash apply stash@{n}
```

其中 `n` 是 stash 的索引号，最新的 stash 编号为 0，编号最大的为最先 stash 的内容。

git stash drop 删除 stash

```
git stash drop stash@{n}
```

这个命令会删除指定的 stash。

git stash pop 应用 stash 并删除

```
git stash pop
```

这个命令会应用最近的 stash 并从 stash 列表中删除它。

git stash show 预览 stash 内容

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git stash show stash@{0}
 2.txt      |  1 -
 test01.txt | 31 +++----------------------------
 test03.txt |  1 +
 3 files changed, 4 insertions(+), 29 deletions(-)
```

仅查看 stash 中文件名

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git stash show stash@{0} --name-only
2.txt
test01.txt
test03.txt
```

比较 stash 与当前工作目录差异

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git stash show stash@{1} -p
diff --git a/1.patch b/1.patch
index e61c591..90d1dfa 100644
--- a/1.patch
+++ b/1.patch
@@ -1 +1,2 @@
 0001-update-fix_B.patch
+1
diff --git a/2.txt b/2.txt
index c200906..91bc947 100644
--- a/2.txt
+++ b/2.txt
@@ -1 +1,3 @@
 222
+22
+2
```

- a 版本为工作目录版本
- b 版本为 stash 中的版本

应用 stash 中的特定文件

当前工作目录的 2.txt 文件内容：

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ cat 2.txt
222
```

应用 stash@{1} 中的 2.txt 版本：

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git checkout stash@{1} -- 2.txt

lx@lx MINGW64 /d/Documents/git_test03 (test)
$ cat 2.txt
222
22
2
```

强制覆盖

```
git checkout --force stash@{0} -- <file-path>
```

或者：

```
git checkout -f stash@{0} -- <file-path>
```

找回被删除的 stash 记录

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git fsck --unreachable | grep commit | cut -d ' ' -f3 | xargs git log --merges --no-walk --oneline
Checking object directories: 100% (256/256), done.
Checking objects: 100% (33/33), done.
99641b1 On test: stash test
6fc3b2a WIP on test: 3030efb Merge branch 'fix_B' of https://github.com/lxwcd/git_test into fix_B
99c8421 WIP on test: 3030efb Merge branch 'fix_B' of https://github.com/lxwcd/git_test into fix_B
c66245f On fix_B: test
c78d607 WIP on fix_B: c188c3c Merge branch 'fix_B' of https://github.com/lxwcd/git_test into fix_B
d0bd729 WIP on fix_B: c188c3c Merge branch 'fix_B' of https://github.com/lxwcd/git_test into fix_B
```

根据日志的输出的 message 找到被删除的 stash，即 9964b1。

将要恢复的 stash 记录重新 stash 并添加 message：

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git update-ref --create-reflog refs/stash 99641b1 -m "restore stash : stash test"

lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git stash list
stash@{0}: restore stash : stash test
```

git fetch

获取远程仓库的所有分支的最新状态

```
git fetch origin
```

获取特定远程分支的最新状态

```
git fetch origin develop
```

这个命令只会获取 `origin` 远程仓库的 `develop` 分支的最新状态。

获取远程特定分支并映射到本地分支

```
git fetch origin src:dst
```

- src 为源端，即远程分支
- dst 为目的端，即本地分支
- 如果当前在 dst 分支，则无法 执行此命令

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git fetch origin fix_B:test
fatal: refusing to fetch into branch 'refs/heads/test' checked out at 'D:/Documents/git_test03
'
```

删除远程不存在的分支引用 **--prune**

```
git fetch --prune
```

从远程仓库获取最新信息的同时， 清除远程跟踪分支中不再存在于远程仓库的分支。

git push

设置本地分支跟踪远程分支

推送到远程仓库的特定分支，而不是本地分支的同名分支：

```
git push <remote> <local-branch>:<remote-branch>
```

- <remote>：远程仓库的名称

将本地 test 分支推送到远程的 test 分支，且设置跟踪状态，远程分支不存在则会在远程仓库中创建该分支：

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git push -u origin HEAD:test
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote:
remote: Create a pull request for 'test' on GitHub by visiting:
remote:      https://github.com/lxwcd/git_test/pull/new/test
remote:
To https://github.com/lxwcd/git_test.git
 * [new branch]      HEAD -> test
branch 'test' set up to track 'origin/test'.
```

- : 前为源分支，即本地分支
- : 后为目的分支，即远程分支
- -u 设置本地分支跟踪远程对应分支，以后可以直接 git push

推送所有本地分支

推送所有本地分支到远程仓库：

```
git push --all <remote>
```

强制推送

```
git push --force <remote> <branch>
```

或者：

```
git push -f <remote> <branch>
```

删除远程分支

```
git push <remote> --delete <branch>
```

或者：

```
git push <remote> :<branch>
```

推送特定提交

```
git push <remote> <commit>:<branch>
```

git pull

`git pull` 用于将远程仓库的更改拉取到本地仓库。
它实际上是一个组合命令，等同于 `git fetch` 后跟 `git merge`。

从远程仓库拉取最新代码并合并到当前分支

```
git pull origin master
```

这个命令会从远程仓库 `origin` 的 `master` 分支拉取最新的代码，并尝试与当前分支合并。

如果当前分支已经设置跟踪远程分支，可以省略远程分支名：

```
git pull origin
```

如果当前分支只跟踪一个远程分支，可以完全省略参数：

```
git pull
```

--rebase

使用 `rebase` 代替 `merge` 来合并更改：

```
git pull --rebase origin master
```

--ff-only

只允许快进式合并，不允许产生新合并提交：

```
git pull --ff-only origin master
```

如果无法进行快进式合并，命令会失败。

--no-commit

拉取后不自动提交合并的结果：

```
git pull --no-commit origin master
```

git rebase

> Git - Rebasing

git cherry-pick

> Apply the changes introduced by some existing commits.

默认会 pick 提交到当前分支且新增一个同样的提交记录，除了 hash 值不一样，其他都一样。

如果 Pick 多个提交，且顺序相关，最好按照原来的顺序从旧往前 pick：

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git log --oneline -5 fix_B
3030efb (fix_B) Merge branch 'fix_B' of https://github.com/lxwcd/git_test into fix_B
3c20c79 update git.md
171d25c update 2.txt 22
f54dd26 update 2.txt 222
15f80f2 (HEAD -> test, origin/test) update test02.txt
```

想 pick 上面 5 个提交，且按照原始的顺序：

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git log --oneline -5 fix_B  | tac | cut -d" " -f1
15f80f2
f54dd26
171d25c
3c20c79
3030efb

lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git log --oneline -5 fix_B  | tac | cut -d" " -f1 | xargs git cherry-pick --no-commit
```

不产生提交记录 --no-commit

```
lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git log --oneline -1 main
cbf4932 (main) update test03.txt

lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git log --oneline -1 main | cut -d" " -f1
cbf4932

lx@lx MINGW64 /d/Documents/git_test03 (test)
$ git log --oneline -1 main | cut -d" " -f1  | xargs git cherry-pick --no-commit
```

修改会在暂存区，不产生提交记录。

Undo things

修改本地的最新提交 git commit --amend

- 仅针对本地最新的提交
- 最新提交还未推送到远程

撤销本地多个提交 git reset

撤销最新的提交：

```
git reset --soft HEAD^
```

撤销最近的三次提交：

```
git reset --soft HEAD~3
```

--soft

- 重置 HEAD 到指定状态，但保留工作目录和暂存区的状态。

--mixed（默认）

- 重置 HEAD 和暂存区到指定状态，但保留工作目录的状态。

--hard

- 重置 HEAD、暂存区和工作目录到指定状态。

显示被撤销的提交

执行 `git reset --soft HEAD^` 后查看被撤销的上次提交，直接用 `git log` 无法查看之前被撤销的提交。

使用 ORIG_HEAD

当执行 `git reset` 时，Git 会自动创建一个名为 `ORIG_HEAD` 的引用，指向原始的 `HEAD` 位置。

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git log ORIG_HEAD --oneline -2
fc003e8 rename 1.txt to 11.txt:wq
e43f274 (HEAD -> main) Merge branch 'branch01'

lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git log --oneline -2
e43f274 (HEAD -> main) Merge branch 'branch01'
03d14ae (origin/main, origin/HEAD) update main test01.txt
```

使用 git reflog

`git reflog` 显示 `HEAD` 的更新历史，你可以通过它找到被撤销的提交：

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git log --oneline -3
36c7d37 (HEAD -> main) update new
e43f274 Merge branch 'branch01'
03d14ae (origin/main, origin/HEAD) update main test01.txt
```

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git reflog -4
36c7d37 (HEAD -> main) HEAD@{0}: commit: update new
e43f274 HEAD@{1}: reset: moving to HEAD^
5e51d74 HEAD@{2}: commit: update
e43f274 HEAD@{3}: reset: moving to HEAD^
```

可以看见在 5e51d74 提交后，进行 reset，因此 HEAD 又变成 e43f274，然后又继续有新的提交。

撤销暂存区的添加 git restore --staged

撤销后修改仍会存在工作目录，但处于未暂存的状态。

如撤销暂存区的全部修改：

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git restore --staged .
```

撤销未暂存的修改 git restore

撤销未暂存的全部修改，不影响已暂存的文件：

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git restore .
```

撤销工作目录全部修改 git reset

让工作目录和 HEAD 一致

```
git reset --hard HEAD
```

撤销远程的提交 git revert

本地提交后 Push 到远程，本地用 git reset --soft HEAD^ 撤销了最近一次提交，如果远程该分支只有自己用，且远程没有比本地更新的提交，则可以 git push --force 覆盖远程提交记录。

如果远程有更新的提交记录，则可以用 git revert HEAD，会新建一个撤销上次提交的提交记录，因此不会和远程冲突。

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git log --oneline -3
36c7d37 (HEAD -> main) update new
e43f274 Merge branch 'branch01'
03d14ae (origin/main, origin/HEAD) update main test01.txt
```

撤销最新一次提交

```
git revert HEAD
```

查看历史记录，发现过去的历史没有改变，只是新增了撤销的记录：

```
lx@lx MINGW64 /d/Documents/git_test03 (main)
$ git log --oneline -3
fabbb86 (HEAD -> main) Revert "update new"
36c7d37 update new
e43f274 Merge branch 'branch01'
```

撤销后可以用 `git push` 推送到远程仓库。

常用案例

git add 筛选特定文件

```
git add *.cpp *.h
```

显示已跟踪被修改的文件名

```
git diff HEAD --name-only
```

筛选已跟踪被修改的文件

```
git diff HEAD --name-only | grep -E "*/demo/*"
```

或：

```
git status --porcelain | cut -d" " -f3- | grep -E "*/demo/*"
```

筛选已跟踪被修改的特定后缀文件

```
git diff HEAD --name-only |  grep -E "\.(cpp|h)$"
```

或：

```
git status --porcelain | cut -d" " -f3- |  grep -E "\.(cpp|h)$"
```

git stash 过滤文件保存

```
git diff HEAD --name-only | grep -E "*/demo/*" | xargs git stash push -m "stash demo files" --
```

或：

```
git status --porcelain | cut -d" " -f3- | grep -E "*/demo/*" | xargs git stash push -m "stash demo files" --
```

输出未被跟踪的文件名

```
lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git status -s
AM git.md
 M test01.txt
?? .gitignore
?? 0001-commit-B.patch
?? 0001-fix-B.patch
?? 0001-update-fix_B.patch
?? 0002-commit-C.patch
?? 0002-update-fix_B.patch
?? 1.patch
?? 2.txt

lx@lx MINGW64 /d/Documents/git_test (fix_B)
$ git status -s |  grep "??" | cut -d" " -f2
.gitignore
0001-commit-B.patch
0001-fix-B.patch
0001-update-fix_B.patch
0002-commit-C.patch
0002-update-fix_B.patch
1.patch
2.txt
```

恢复工作目录到最新提交

git stash

```
git stash push -m "message"
```

如果有没有被跟踪的文件，希望一起存起来：

```
git stash push --all -m "message"
```

git checkout

恢复已跟踪的文件：

```
git checkout HEAD -- .
```

再删除没有被跟踪的文件：

```
git clean -fd
```

git reset

恢复已跟踪的文件：

```
git reset --hard HEAD
```

再删除没有被跟踪的文件：

```
git clean -fd
```

或者：

```
$ git status --short | grep "??" | cut -d" " -f2 | xargs rm -rf
```

git restore

恢复已跟踪的文件：

```
$ git restore --source=HEAD --staged --worktree .
```

再删除没有被跟踪的文件：

```
git clean -fd
```

有冲突时指定使用版本

两个仓库中的一个分支都修改了文件 2.txt 的相同一行，但修改内容内容不同，一个仓库已经将修改推送到远程仓库，另一个仓库修改后提交到本地，然后 git pull 时提示有冲突：

```
lx@lx MINGW64 /d/Documents/git_test03 (fix_B)
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 239 bytes | 9.00 KiB/s, done.
From https://github.com/lxwcd/git_test
   15f80f2..f54dd26  fix_B      -> origin/fix_B
Auto-merging 2.txt
CONFLICT (content): Merge conflict in 2.txt
Automatic merge failed; fix conflicts and then commit the result.
```

查看当前工作目录的状态，可以看见有个文件处于冲突中，待解决：

```
lx@lx MINGW64 /d/Documents/git_test03 (fix_B|MERGING)
$ git status
On branch fix_B
Your branch and 'origin/fix_B' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   2.txt

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

```

查看冲突文件的内容：

```
lx@lx MINGW64 /d/Documents/git_test03 (fix_B|MERGING)
$ cat 2.txt
<<<<<<< HEAD
22
=======
222
>>>>>>> f54dd265ddebde6a06e2ea619a0588d2b1555945
```

<<<<<<< HEAD 表示下方表示当前版本
======= 表示分隔符，下方内容为冲突版本的内容
>>>>>>> f54dd265ddebde6a06e2ea619a0588d2b1555945 表示冲突结束位置

使用对方版本

```
lx@lx MINGW64 /d/Documents/git_test03 (fix_B|MERGING)
$ git checkout --theirs 2.txt
Updated 1 path from the index

lx@lx MINGW64 /d/Documents/git_test03 (fix_B|MERGING)
$ cat 2.txt
222
```

使用本地版本

```
lx@lx MINGW64 /d/Documents/git_test03 (fix_B|MERGING)
$ git checkout --ours 2.txt
Updated 1 path from the index

lx@lx MINGW64 /d/Documents/git_test03 (fix_B|MERGING)
$ cat 2.txt
22
```

添加到暂存区

选择版本后，将这些文件添加到暂存区：

```
git add <file-path>
```

继续合并

如果你已经解决了所有文件的冲突，你可以继续完成合并操作：

```
git commit -m "message"
```

或者

```
git merge --continue
```