

# 进阶 3：merge：合并 commits

前面说到，pull 的内部操作其实是把远程仓库取到本地后（使用的是 fetch），再用一次 merge 来把远端仓库的新 commits 合并到本地。这一节就说一下，merge 到底是什么。

## 含义和用法

merge 的意思是「合并」，它做的事也是合并：指定一个 commit，把它合并到当前的 commit 来。具体来讲，merge 做的事是：

从目标 commit 和当前 commit（即 HEAD 所指向的 commit）分叉的位置起，把目标 commit 的路径上的所有 commit 的内容一并应用到当前 commit，然后自动生成一个新的 commit。

例如下面这个图中：

HEAD 指向了 master，所以如果这时执行：

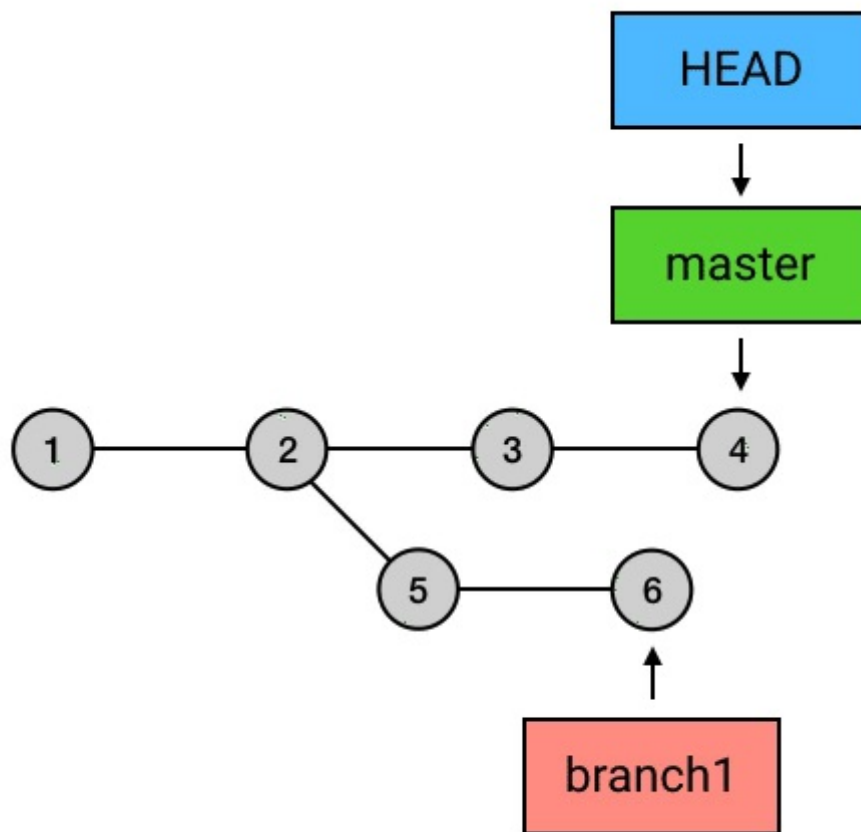
```
git merge branch1
```

Git 会把 5 和 6 这两个 commit 的内容一并应用到 4 上，然后生成一个新的提交，并跳转到提交信息填写的界面：

```
Merge branch 'branch1'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

merge 操作会帮你自动地填写简要的提交信息。在提交信息修改完成后（或者你打算不修改默认的提交信息），就可以退出这个界面，然后这次 merge 就算完成了。



## 适用场景

merge 有什么用？最常用的场景有两处：

### 1. 合并分支

当一个 branch 的开发已经完成，需要把内容合并回去时，用 merge 来进行合并。

那 branch 又应该怎么用呢？

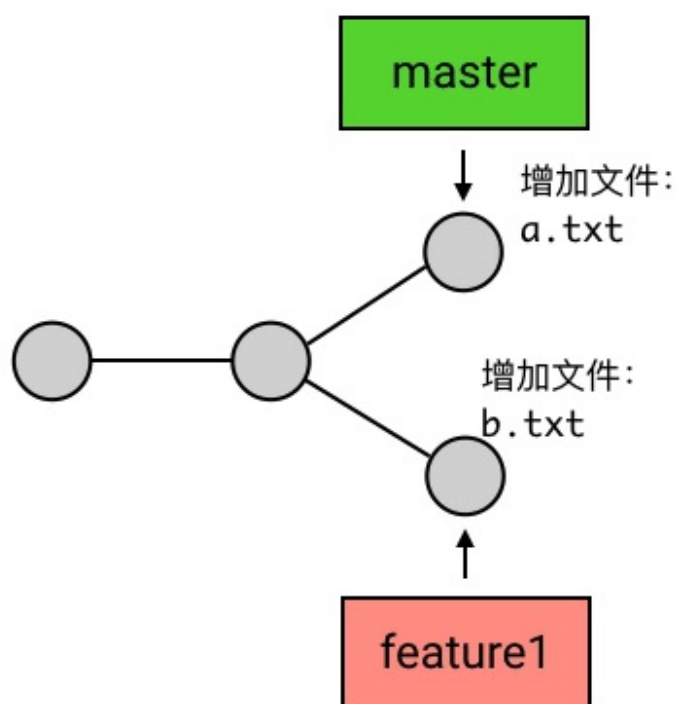
下节就说。

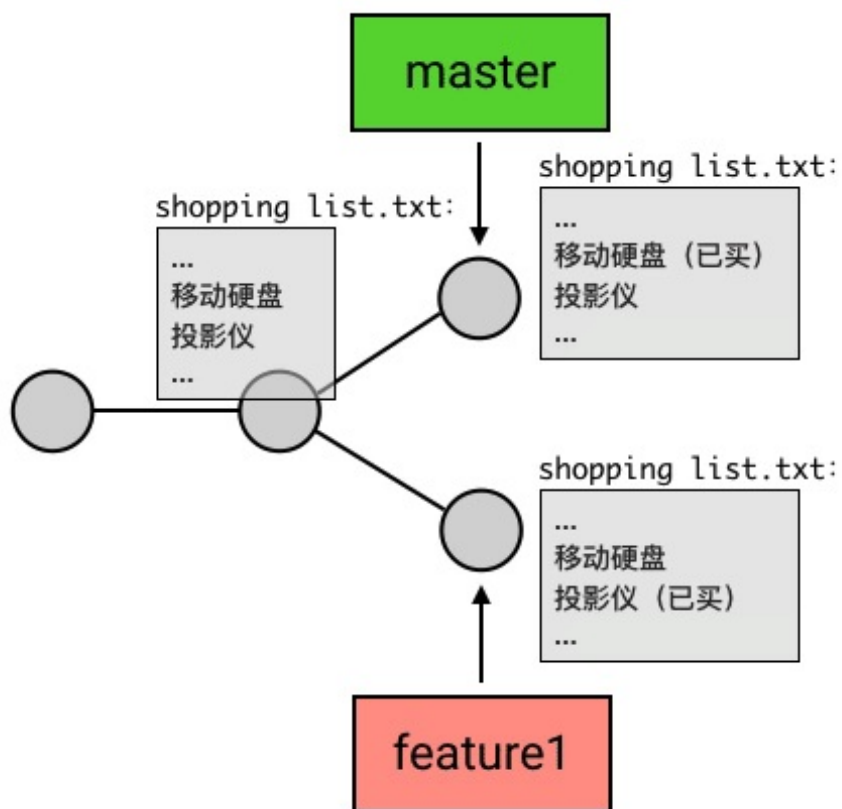
## 2. pull 的内部操作

之前说过，pull 的实际操作其实是把远端仓库的内容用 fetch 取下来之后，用 merge 来合并。

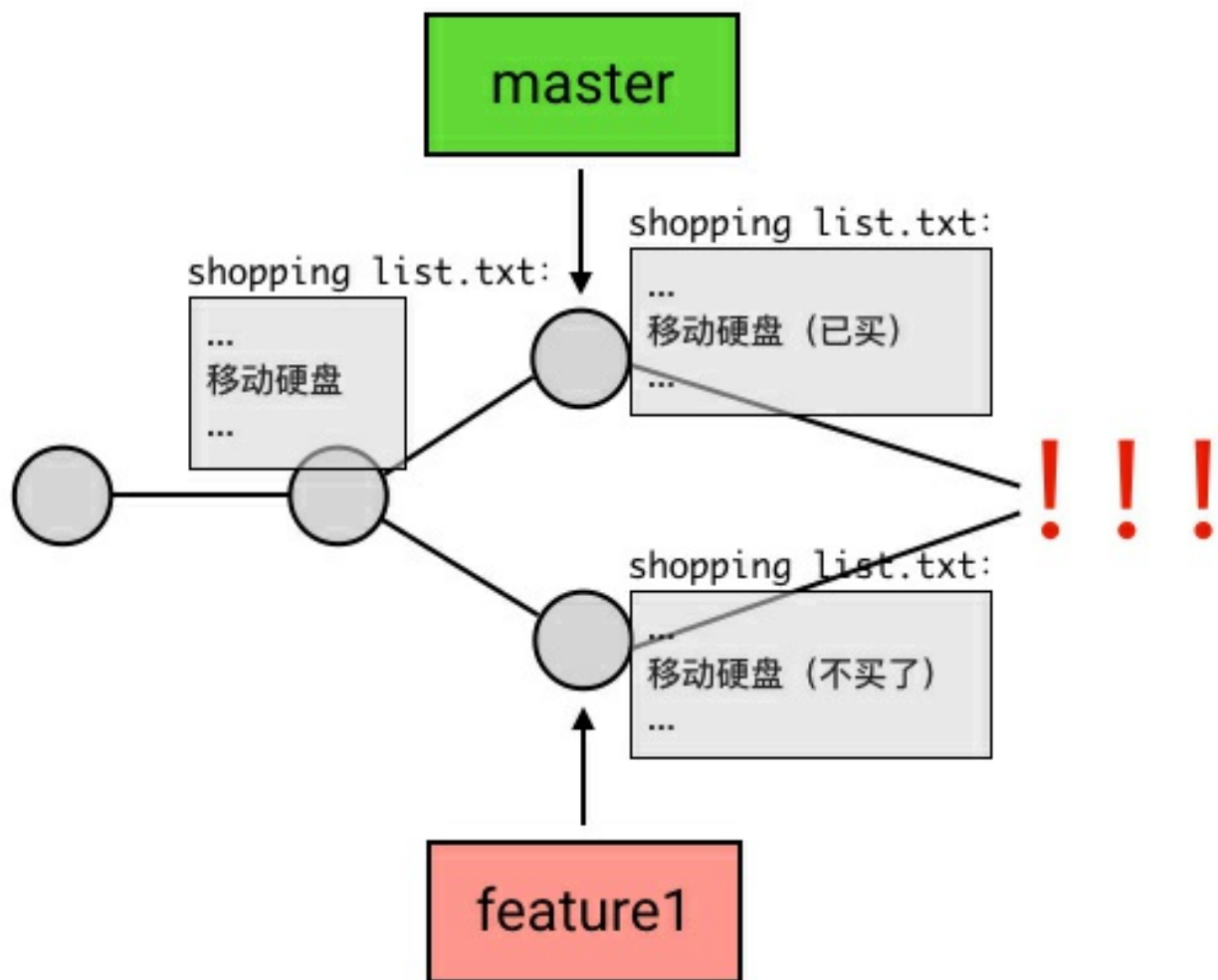
### 特殊情况 1：冲突

merge 在做合并的时候，是有一定的自动合并能力的：如果一个分支改了 A 文件，另一个分支改了 B 文件，那么合并后就是既改 A 也改 B，这个动作会自动完成；如果两个分支都改了同一个文件，但一个改的是第 1 行，另一个改的是第 2 行，那么合并后就是第 1 行和第 2 行都改，也是自动完成。





但，如果两个分支修改了同一部分内容，merge 的自动算法就搞不定了。这种情况 Git 称之为：冲突（Conflict）。



直白点说就是，你的两个分支改了相同的内容，Git 不知道应该以哪个为准。如果在 merge 的时候发生了这种情况，Git 就会把问题交给你来决定。具体地，它会告诉你 merge 失败，以及失败的原因：

```
git merge feature1
```

```
→ git-practice git:(master) git merge feature1
Auto-merging shopping list.txt
CONFLICT (content): Merge conflict in shopping list.txt
Automatic merge failed; fix conflicts and then commit the result.
```

提示信息说，在 shopping list.txt 中出现了 "merge conflict"，自动合并失败，要求 "fix conflicts and then commit the result"（把冲突解决掉后提交）。那么你现在需要做两件事：

### 1. 解决掉冲突

## 2. 手动 commit 一下

### 1. 解决冲突

解决掉冲突的方式有多个，我现在说最直接的一个。你现在再打开 shopping list.txt 看一下，会发现它的内容变了：

```
Maltron 键盘
<<<<<< HEAD
移动硬盘（已买）
=====
移动硬盘（不买了）
>>>>>> feature1
投影仪
再买几个比特币
```

可以看到，Git 虽然没有帮你完成自动 merge，但它对文件还是做了一些工作：它把两个分支冲突的内容放在了一起，并用符号标记出了它们的边界以及它们的出处。上面图中表示，HEAD 中的内容是 移动硬盘（已买），而 feature1 中的内容则是 移动硬盘（不买了）。这两个改动 Git 不知道应该怎样合并，于是把它们放在一起，由你来决定。假设你决定保留 HEAD 的修改，那么只要删除掉 feature1 的修改，再把 Git 添加的那三行 <<< === >>> 辅助文字也删掉，保存文件退出，所谓的「解决掉冲突」就完成了。

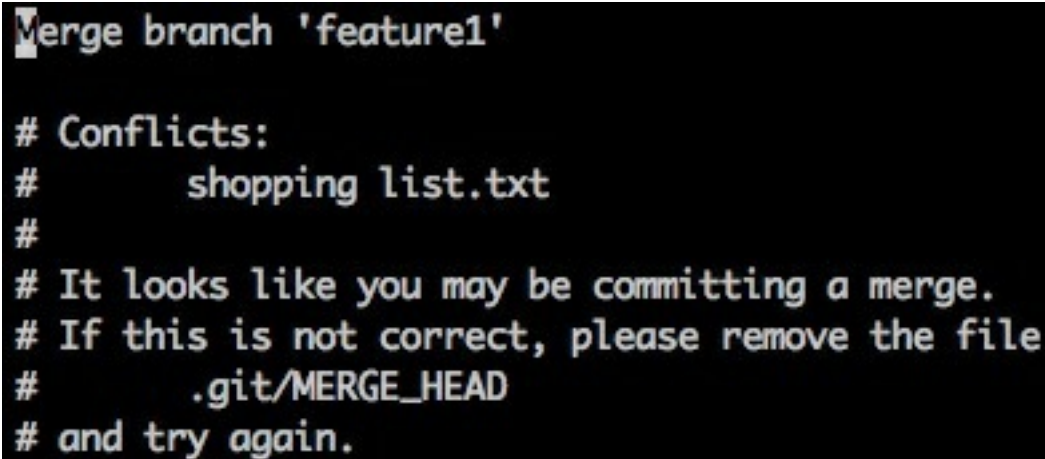
```
Maltron 键盘
移动硬盘（已买）
投影仪
再买几个比特币
```

你也可以选择使用更方便的 merge 工具来解决冲突，这个你可以自己搜索一下。

### 2. 手动提交

解决完冲突以后，就可以进行第二步—— commit 了。

```
git add shopping\ list.txt # 嗯是的，这里 commit 前  
也需要先 add 一下  
git commit
```

A terminal window with a black background and white text. The text shows the output of a Git merge command. It starts with 'Merge branch 'feature1'', followed by a section titled '# Conflicts:' which lists 'shopping list.txt'. Below this, it says '# It looks like you may be committing a merge. # If this is not correct, please remove the file # .git/MERGE\_HEAD # and try again.'

```
Merge branch 'feature1'  
  
# Conflicts:  
#       shopping list.txt  
#  
# It looks like you may be committing a merge.  
# If this is not correct, please remove the file  
#       .git/MERGE_HEAD  
# and try again.
```

可以看到，被冲突中断的 merge，在手动 commit 的时候依然会自动填写提交信息。这是因为在发生冲突后，Git 仓库处于一个「merge 冲突待解决」的中间状态，在这种状态下 commit，Git 就会自动地帮你添加「这是一个 merge commit」的提交信息。

## 放弃解决冲突，取消 merge?

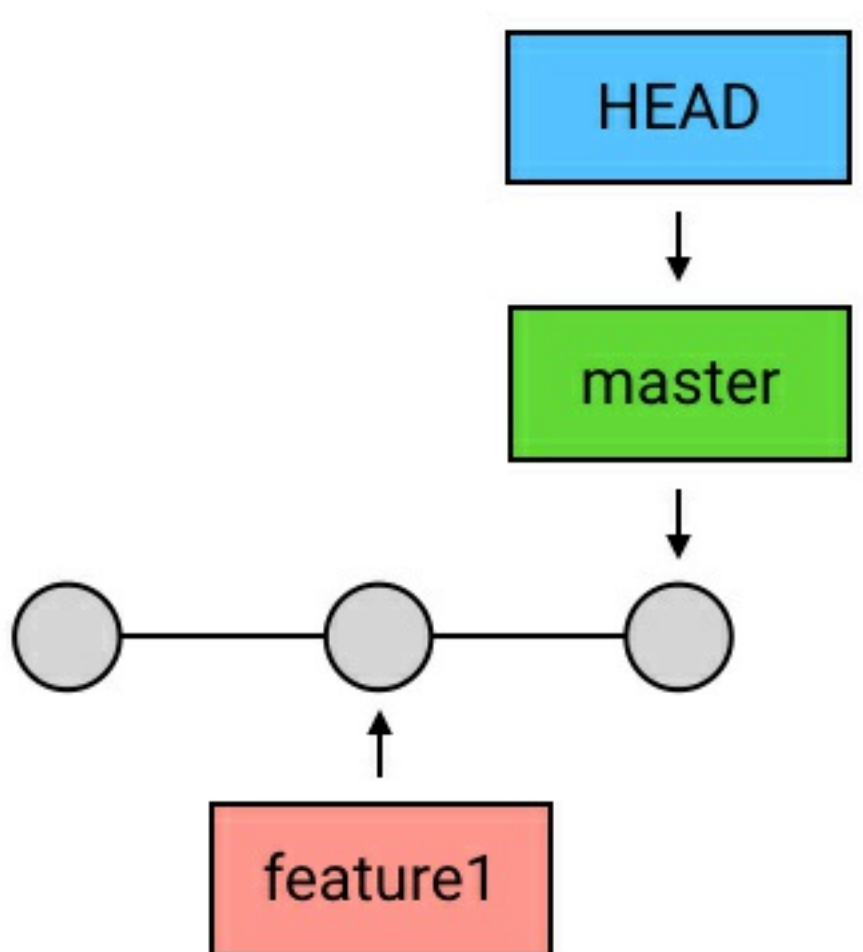
同理，由于现在 Git 仓库处于冲突待解决的中间状态，所以如果你最终决定放弃这次 merge，也需要执行一次 `merge --abort` 来手动取消它：

```
git merge --abort
```

输入这行代码，你的 Git 仓库就会回到 merge 前的状态。

## 特殊情况 2：HEAD 领先于目标 commit

如果 merge 时的目标 commit 和 HEAD 处的 commit 并不存在分叉，而是 HEAD 领先于目标 commit：



那么 merge 就没必要再创建一个新的 commit 来进行合并操作，因为并没有什么需要合并的。在这种情况下，Git 什么也不会做，merge 是一个空操作。

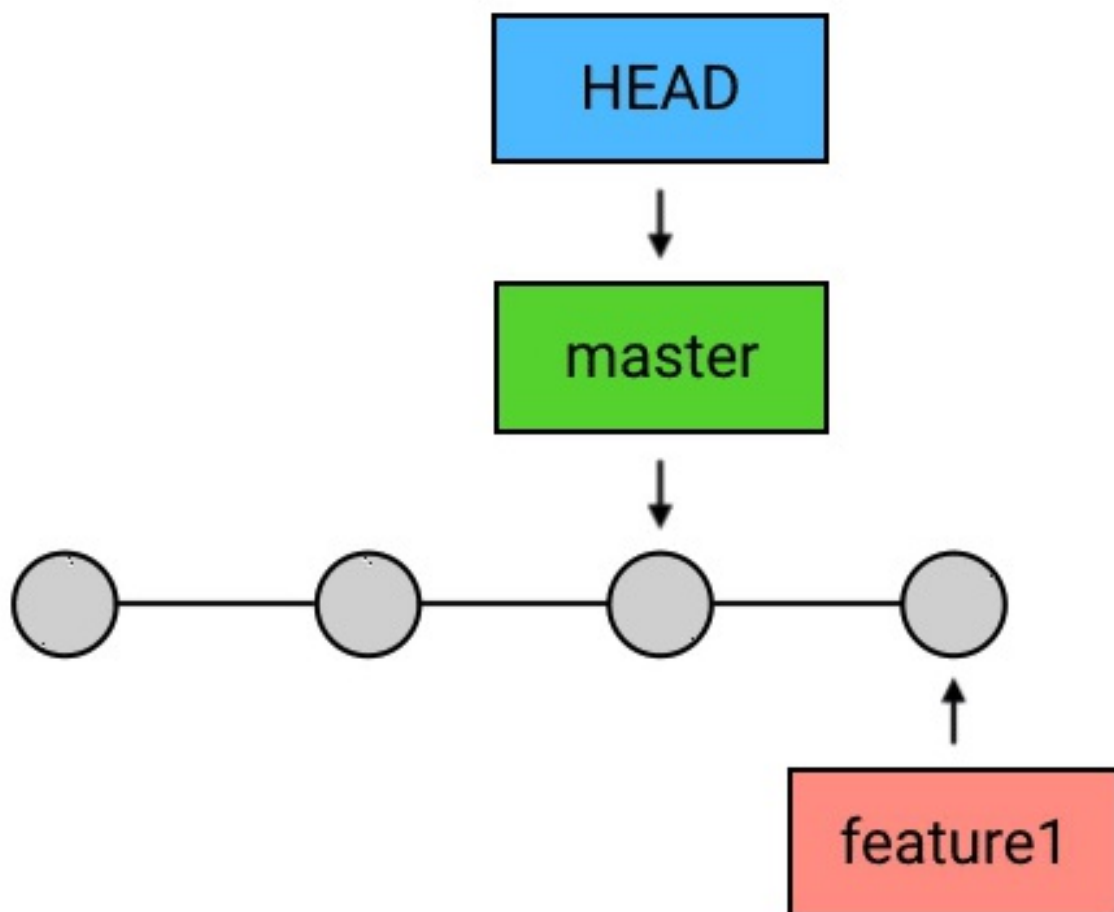
### 特殊情况 3：HEAD 落后于 目标 commit ——fast-forward

而另一种情况：如果 HEAD 和目标 commit 依然是不存在分叉，但 HEAD 不是领先于目标 commit，而是落后于目标 commit：



那么 Git 会直接把 HEAD（以及它所指向的 branch，如果有的话）移动到目标 commit：

```
git merge feature1
```



这种操作有一个专有称谓，叫做 "fast-forward"（快速前移）。

一般情况下，创建新的 branch 都是会和原 branch（例如上图中的 master）并行开发的，不然没必要开 branch，直接在原 branch 上开发就好。但事实上，上图中的情形其实很常见，因为这其实是 pull 操作的一种经典情形：本地的 master 没有新提交，而远端仓库中有同事提交了新内容到 master：

那么这时如果在本地执行一次 pull 操作，就会由于 HEAD 落后于目标 commit（也就是远端的 master）而造成 "fast-forward"：

```
git pull
```

简单解释一下上图中的 `origin/master` 和 `origin/HEAD` 是什么鬼：它们是对远端仓库的 `master` 和 `HEAD` 的本地镜像，在 `git pull` 的「两步走」中的第一步——`git fetch` 下载远端仓库内容时，这两个镜像引用得到了更新，也就是上面这个动图中的第一步：`origin/master` 和 `origin/HEAD` 移动到了最新的 `commit`。

为什么前面的图里面从来都没有这两个「镜像引用」？  
因为我没有画呀！其实它们是一直存在的。

而 `git pull` 的第二步操作 `merge` 的目标 `commit`，是远端仓库的 `HEAD`，也就是 `origin/HEAD`，所以 `git pull` 的第二步的完整内容是：

```
git merge origin/HEAD
```

因此 `HEAD` 就会带着 `master` 一起，也指向图中绿色的最新 `commit` 了。

## 小结

本节对 `merge` 进行了介绍，内容大概有这么几点：

1. `merge` 的含义：从两个 `commit` 「分叉」的位置起，把目标 `commit` 的内容应用到当前 `commit` (`HEAD` 所指向的 `commit`)，并生成一个新的 `commit`；
2. `merge` 的适用场景：
  1. 单独开发的 `branch` 用完了以后，合并回原先的 `branch`；

2. `git pull` 的内部自动操作。

3. `merge` 的三种特殊情况：

1. 冲突

1. 原因：当前分支和目标分支修改了同一部分内容，Git 无法确定应该怎样合并；

2. 应对方法：解决冲突后手动 `commit`。

2. HEAD 领先于目标 `commit`：Git 什么也不做，空操作；

3. HEAD 落后于目标 `commit`：fast-forward。