【Git】語法觀念

- 【Git】相關連結
- 【Git】還原相關
- 【Git】TorotoiseGit 占用資源解決方法
- 【Git】windows系統 warning: LF will be replaced by CRLF in
- 【Git】checkout 與 switch 比較
- 【Git】HEAD 是什麼
- 【Git】每個 commit 不都是從某個分支出來的嗎？
- 【Git】head, tag, commit, branch的關係
- 【Git】git revert <commit> 和 git checkout <commit> 差異
- 【Git】常用指令

【Git】相關連結

github 線上教學
GitHub Learning Lab

Git基本原理介□ | Escape (escapelife.site)

Learn Git Branching (gitee.io)

Git基本原理介□ | Escape (escapelife.site)

【Git】還原相關

以下是關於 Git 還原（恢復）功能的完整說明

Git 還原功能總表

指令	還原範圍	是否會刪除暫存區	是否會刪除未追蹤檔案	適用情境
`git checkout .`	工作目錄	否	否	還原已追蹤檔案
`git restore .`	工作目錄	否	否	推薦用法，等同於 checkout .
`git restore --staged`	暫存區	是	否	還原 `git add` 的檔案
`git reset`	暫存區	是	否	退回 staging 區
`git reset --hard HEAD`	工作目錄 + 暫存區	是	是（若搭配 clean）	徹底放棄所有變更（⚠危險）
`git clean -fd`	未追蹤檔案/資料夾	無關	是	清空未納入 Git 管控的檔案
`git revert <commit>`	版本歷史	無關	否	安全回復 commit（保留紀錄）

Git 還原功能完整整理與使用說明

在日常開發中，Git 提供多種指令用來還原工作目錄、暫存區，或是回復錯誤的 commit。根據不同的需求，選擇合適的還原方式可避免資料遺失或版本混亂。

---

1. `git checkout .`

還原所有「已追蹤檔案」回到最新 commit 的狀態。

- **作用範圍**：僅限工作目錄（Working Directory）
- **不會還原**：未追蹤（untracked）檔案與暫存區（Staging Area）

適用情境：

- 編輯了多個檔案，想快速放棄變更，但未進行 `git add`。

---

2. `git restore`（推薦用法）

`git restore` 是 Git 2.23+ 提供的新指令，取代過去 `checkout` 的混用狀況。

範例：

```
git restore .                # 還原所有檔案
git restore path/to/file.js # 還原單一檔案
```

- 與 `git checkout .` 行為一致，語意更清楚
- 可搭配 `--staged` 還原暫存區（如下）

---

3. `git restore --staged`

還原暫存區的內容，讓已 `git add` 的檔案退回至工作目錄狀態。

```
git restore --staged path/to/file.js
```

- **不會影響工作區的內容**
- 常用於誤將檔案加入 staging 區的情況

---

4. git reset

還原 staging 區（暫存區），讓所有已 `git add` 的檔案回到修改中狀態。

```
git reset
```

- 與 `git restore --staged` 功能類似
- 不會更動工作目錄內容

---

5. git reset --hard HEAD

將 **工作目錄與暫存區** 都還原成最近一次提交（HEAD）的狀態。

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

- **非常危險**，會清除未提交的所有變更
- 包含 `git add` 過的內容也會消失

適用情境：

- 變更混亂，想完全清空所有未提交的東西

---

6. git clean -fd

清除所有「未被追蹤的檔案與資料夾」。

```
git clean -fd
```

- `-f`：強制執行
- `-d`：包括未追蹤的資料夾

適用情境：

- 清除 build 檔案或不小心留下的臨時檔案

---

7. git revert

對某個已提交的 commit 產生一個「反向操作的 commit」。

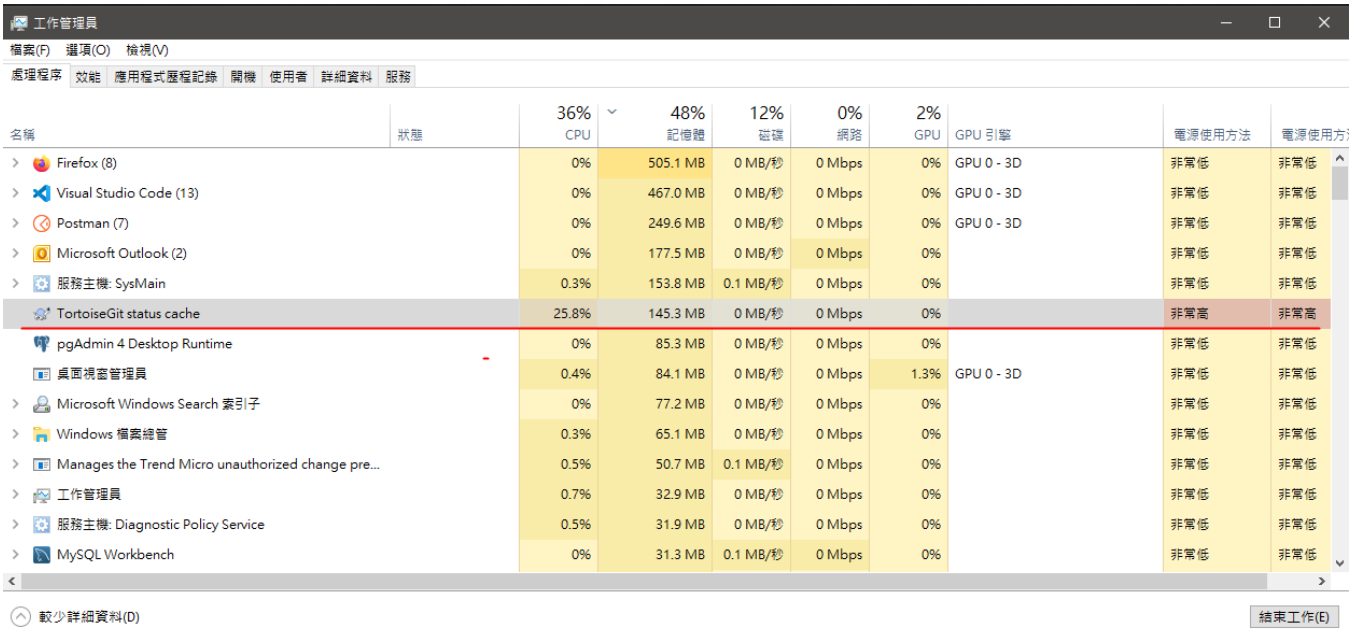
```
git revert <commit_hash>
```

- 保留歷史紀錄
- 安全的還原方式，尤其在多人開發時

適用情境：

- 上線後發現某次 commit 有錯誤，需在版本歷史中保留修正記錄

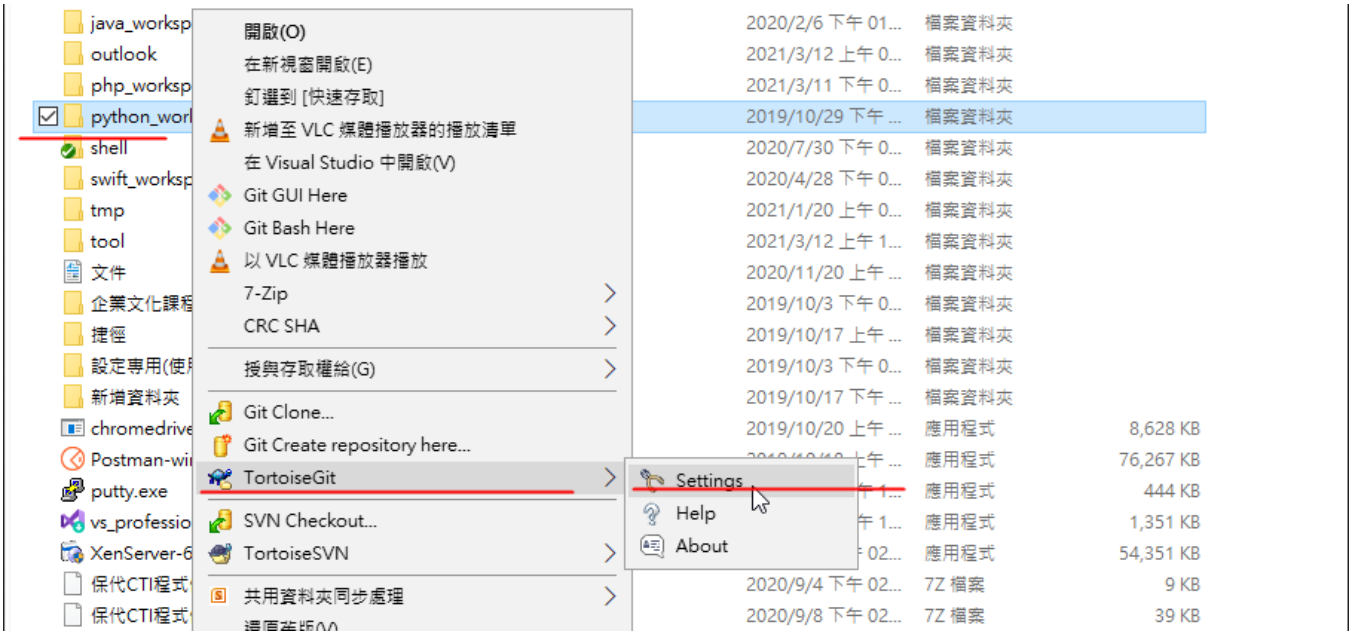
【Git】TorotoiseGit 占用資源解決方法

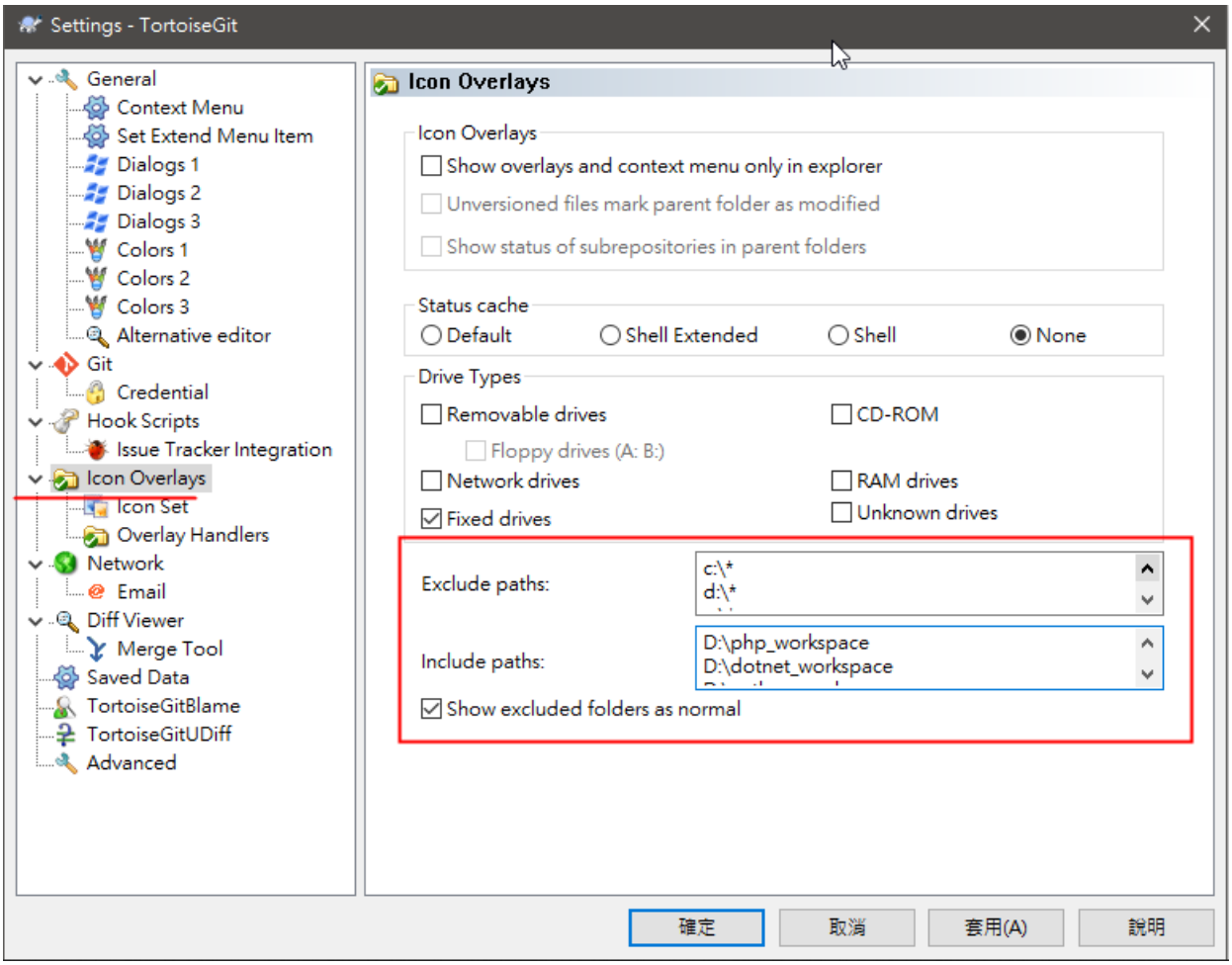


【原因】 TorotoiseGit 會掃描硬碟內有git 版控資料夾的的異動，並產生相對應的圖示，有時會造成主機磁碟讀寫過高

【解決】限縮掃描範圍

任一個資料夾按右鍵->【TortoiseGit】->【setting】





Exclude paths: 將本機的每一個磁碟加入

Include paths: 將有用到git的目錄加入

【Git】windows系統 warning: LF will be replaced by CRLF in

windows系統下使用git出現：warning: LF will be replaced by CRLF in

```
$ git add readme.txt readme.txt
warning: LF will be replaced by CRLF in learngit/readme.txt.
The file will have its original line endings in your working directory
```

原因是windows系統的換行符和Unix下的換行符不同，但是git會幫我們自動進行換行符的轉換，所以出現了這個warning

```
# 解決的方法就是，禁止自動轉換：
$git config --global core.autocrlf false
```

https://noelhsiao.medium.com/%E9%81%87%E5%88%B0-warning-lf-will-be-replaced-by-crlf-in-%E8%A9%B2%E6%80%8E%E9%BA%BC%E8%A7%A3-ab9882ee07be

【Git】checkout 與 switch 比較

我來幫你整理 `git switch` 和 `git checkout` 的差異，並用表格讓你快速比較。

1⃣ 歷史背景

- **`git checkout`** 是 Git 早期就有的指令，功能很多，但語意太複雜，既可以切換分支、也可以檢出檔案，容易讓初學者混淆。
- **`git switch`**（Git 2.23 之後加入）是為了讓「切換分支」這個動作更直觀而出現的，專注於分支切換，不負責檔案檢出。

2⃣ 功能比較表

功能	`git checkout`	`git switch`
切換到已存在分支	✅ `git checkout branch_name`	✅ `git switch branch_name`
建立並切換到新分支	✅ `git checkout -b new_branch`	✅ `git switch -c new_branch`
檢出特定 commit / tag	✅ `git checkout commit_hash`	❌ 不支援
檢出特定檔案版本	✅ `git checkout commit_hash -- file.txt`	❌ 不支援
語意單純性	❌ 功能太多易混淆	✅ 專注於切換分支
推薦使用情境	老專案 / 習慣舊指令	新專案 / 想簡化操作

3⃣ 使用範例

切換到現有分支

```
# 傳統方式
git checkout dev

# 新方式（推薦）
git switch dev
```

建立並切換到新分支

```
# 傳統方式
git checkout -b feature/login
# 新方式（推薦）
git switch -c feature/login
```

4⃣ 為什麼官方推薦 `git switch`

- 讓指令語意更明確，不再把「檔案檢出」和「分支切換」混在一起。
- 搭配 `git restore`（處理檔案版本還原）更符合直覺：
  - `git switch` → 專注切換分支
  - `git restore` → 還原檔案內容

✅ **簡單記：**

- **切換分支** → 用 `git switch`
- **還原檔案** → 用 `git restore`

- **需要舊版本的行為** → 用 `git checkout`

太好了！以下是 `git switch` 和 `git restore` 如何取代舊指令 `git checkout` 的**完整對照表**，讓你更容易理解並開始使用新指令：

🔁 `git checkout` 對照表：對應的新指令用法

原本 `git checkout` 用法	對應的新指令	功能說明
`git checkout <branch>`	`git switch <branch>`	切換到已存在的分支
`git checkout -b <new-branch>`	`git switch -c <new-branch>`	建立並切換到新分支
`git checkout <commit>`	❌（建議保留 `checkout`）	切換到某個 commit 狀態（暫不取代）
`git checkout <file>`	`git restore <file>`	還原工作區中的單一檔案
`git checkout -- <file>`	`git restore <file>`	同上
`git checkout <commit> -- <file>`	`git restore --source <commit> <file>`	還原某個 commit 中的檔案
`git checkout .`	`git restore .`	還原所有工作區的檔案
`git checkout HEAD <file>`	`git restore --source=HEAD <file>`	還原最新 commit 的某個檔案
`git checkout HEAD^`	❌（建議保留 `checkout`）	切換到上一個 commit（建議保留舊指令）

🧠 建議做法

任務	建議用法
切換分支	`git switch dev`
建新分支並切過去	`git switch -c feature/new-ui`
還原單一檔案	`git restore src/App.tsx`
還原某 commit 的檔案	`git restore --source=abc123 file.txt`
還原整個工作目錄	`git restore .`

🧪 Git 版本要求

- `git switch` 和 `git restore` 需 Git **2.23 或以上**
- 使用 `git --version` 檢查目前版本

【Git】HEAD 是什麼

當然可以！`HEAD` 是 Git 中最核心也是最常見的概念之一，很多 Git 指令（如 `reset`、`checkout`、`rebase`）都與它有關。下面我會從簡單到深入，幫你完整說明：

一句話理解 HEAD 是什麼：

> **`HEAD` 是你目前所在的分支的「指標」，也就是「你現在工作的 commit」位置。**

基本概念圖：

假設你有三個 commit：

```
A → B → C
      ↑
    HEAD
```

- 這表示目前 `HEAD` 指向 commit C
- 如果你新增一個 commit，會變成：

```
A → B → C → D
          ↑
        HEAD
```

HEAD 的 3 種狀態：

狀態類型	說明
`HEAD` → 分支名稱（正常狀態）	你在某個分支上（例如 `main`、`dev`）
`HEAD` → commit（detached）	你處於「分離 HEAD 狀態」（Detached HEAD）
`HEAD` → 其他指標	比如在 Rebase、Merge 中會指向暫存區域

常見用途說明

1. `HEAD` 是你目前的開發位置

當你執行：

```
git commit
```

其實 Git 是「把你的 commit 接在 `HEAD` 指向的 commit 後面」。

---

2. `HEAD` 是 Git 重設的基準點

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

這表示：**把你的暫存區與工作目錄都還原成目前這個 commit 的狀態。**

---

3. `HEAD^` 、`HEAD~1`：往上移動一個 commit

表示法	意思
`HEAD`	現在這個 commit
`HEAD^`	上一個 commit（parent）
`HEAD~1`	同 `HEAD^`
`HEAD~2`	往上兩個 commit

Detached HEAD（分離 HEAD 狀態）

當你執行：

```
git checkout <某個 commit 哈希>
```

你會進入「Detached HEAD」狀態：

```
A → B → C
      ↑
   (HEAD)
```

你不再是在某個分支上，而是「直接指向某個 commit」。這時如果你做出 commit，它們會變成「懸空」，容易被 Git 回收（垃圾回收）。

所以如果你看到這樣的訊息：

```
You are in 'detached HEAD' state.
```

請小心：最好先建立分支來保存變更：

```
git checkout -b temp-branch
```

總結：HEAD 是 Git 的當前位置指標

功能	用途說明
表示目前 commit	所有操作（commit、reset）都以它為基礎
指向當前分支	預設情況 HEAD 是指向你當前分支
變更分支 = 移動 HEAD	切分支就是改變 HEAD 的位置
可用於版本回朔、還原	`HEAD^` 、`HEAD~2` 常見於 reset、diff 等
被分離 = Detached HEAD	指向 commit 而非分支，要小心變更遺失

如果你想要我畫一張圖示 `HEAD` 與分支移動的流程圖，也可以說一聲，我能幫你生成視覺化版本！是否需要？

為什麼我們要「移動 HEAD」？

簡單來說：

> **移動 HEAD = 改變我們正在「看的版本」、「工作的版本」、或「建立新歷史的起點」。**

Git 是一個版本控制系統，HEAD 就像「目前播放的時間軸指標」，移動它可以讓你做版本切換、回朔、修正、實驗開發等動作。

移動 HEAD 的常見目的（整理成表）

用途/目的	說明
切換分支或版本	改變目前正在編輯/開發的版本，比如 `git switch dev`（切換 HEAD 到 dev）
回朔 commit	想把 HEAD 拉回到前一版：`git reset --hard HEAD^`
版本測試或除錯某一版	用 `git checkout <commit>` 移動到某個版本試驗、測試（會進入 detached HEAD）
清除或重建 commit 歷史	用 `git reset` 或 `rebase` 改變 HEAD 來整理版本歷史
建立新的分支開發點	移動 HEAD 到 commit 後再用 `git switch -c new-branch` 開分支
放棄變更並回到穩定版本	移動 HEAD 到穩定 commit 位置，還原所有內容

舉幾個具體例子

1 切換分支（背後其實是移動 HEAD）

```
git switch dev
```

> HEAD 從 main → dev，代表你現在工作的是 dev 的最新 commit。

---

2 回到前一版（移動 HEAD 到上一個 commit）

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

> HEAD 往回一格，捨棄最近一次的提交內容（危險但有用）。

---

3 想測試舊版功能（Detached HEAD 狀態）

```
git checkout abc1234
```

> HEAD 不再指向分支，而是指向一個 commit，你進入「Detached HEAD」狀態，可以安全看舊版本。

---

4 切 HEAD + 建立新分支（從中間版本開新分支）

```
git checkout abc1234
git switch -c fix-login
```

> □ 移動 HEAD 到舊版，再基於此版本開新分支，進行 hotfix。

把 HEAD 當「播放頭」的比喻

想像 Git 是個影片編輯器：

- commit 是一個個時間點的「場景」
- HEAD 是播放頭
- 你移動 HEAD，就等於跳到那一幕開始剪接

這樣你就能理解：

- reset 是「退回到前一段」
- rebase 是「換剪接順序」
- checkout 是「跳到某個片段」
- commit 是「加入新片段」

□ 小結：為什麼要移動 HEAD？

目的	解釋
□ 切換你正在工作的版本	因為 HEAD 代表你「現在的位置」
□ 回到過去版本	移動 HEAD 讓你回到某個穩定 commit
□ 實驗新東西 / 修 bug	從舊 commit 移動 HEAD，開新分支來做測試與修復
□ 重構 Git 歷史	通過 reset 或 rebase 移動 HEAD 重寫 commit 歷史

這是個很關鍵的 Git 概念問題！
當你的 `HEAD` **不在任何分支上，而是直接指向某個 commit**，這個狀態稱為：

⚠ **Detached HEAD**（分離 HEAD 狀態）

□ 什麼是 Detached HEAD？

正常情況：

```
HEAD → main → C
```

Detached HEAD 狀況：

```
HEAD → C （但不是任何分支）
```

你直接「跳到一個 commit」，例如透過：

```
git checkout abc1234
```

或：

```
git switch --detach abc1234
```

這時你就**脫離了任何分支的控制**，你在 **那個 commit 上自由工作**，但：

> □ 所做的一切（如新增 commit）都**不屬於任何分支**，如果不小心離開就會「失聯」。

Detached HEAD有什麼影響？

有什麼影響？

結果	說明
□ 可以修改檔案、做 commit	你仍然可以正常工作（只是沒有在分支上）
□ 新的 commit 沒有分支指向	所有新 commit 是「懸空狀態」，不儲存在任何分支上
□ 一旦切換到其他分支就可能遺失那些 commit	除非你把它們另存（例如建立新分支），否則 Git 垃圾回收後就找不回來
□ `git status` 會提示 Detached HEAD	告訴你現在是在 Detached 狀態

什麼情況會進入 Detached HEAD？

情境	指令
想要看舊版本	`git checkout <commit-hash>`
從某個 commit 測試 / 編譯	`git switch --detach <commit>`
在 CI/CD 工具中 checkout 特定 commit	例如 GitHub Actions 的 `actions/checkout@v2` 預設為 Detached
你用 `git reset --hard` 到某個 commit	HEAD 也會臨時分離

□ 正確處理方式：建立分支保存你做的事！

如果你在 Detached HEAD 狀態下做了 commit，請馬上建立一個分支把它們保存下來：

```
git switch -c fix-login-bug
```

這樣你的 commit 就有名字，不會被回收。

總結：Detached HEAD 是可用但有風險的模式

特性	狀態說明
可以編輯與 commit	□ 是
commit 是否屬於分支	□ 否
離開當前 commit 後是否會遺失	□ 會（除非建立分支）
建議	**要保留修改就馬上建立分支**

【Git】每個 commit 不都是從某個分支出來的嗎？

這是個非常棒的深入問題，顯示你對 Git 的理解已經進入「底層思維模式」，我們來一一說明：

一句話解釋你提的問題：

> **不是每個 commit 都屬於分支，但每個分支都指向某個 commit。**

這句話是 Git 的核心設計邏輯，反直覺但重要。

先理解 Git 的本質：**commit 是資料庫，分支是指標**

在 Git 裡：

- 每個 commit 是獨立、完整、不可變的「快照」
- 「分支」其實只是指向某個 commit 的**指標（pointer）**
- `HEAD` 是你當下的游標，會指向一個「分支」或「commit」

當你做 commit 時發生了什麼？

1. Git 會建立一個新 commit（有一個父 commit）
2. `HEAD` 所在的分支（如 `main`）會更新，指向這個新的 commit

所以我們通常**覺得 commit 是從分支「出來」的**，但其實：

> 分支並不綁定在 commit 上，是 commit 被分支「指到了」。

Detached HEAD：從 commit 建立 commit，但不屬於分支

來看個例子：

```
git checkout abc1234   # 進入 detached HEAD
echo "foo" > a.txt
git add a.txt
git commit -m "detached commit"
```

這時你新增的 commit 並不屬於任何分支！

```
A --- B --- C    ← main
         \
          D   ← HEAD (沒有分支名稱指向)
```

如果你切走：

```
git switch main
```

那個 `D` commit **就被遺棄了**，未來可能被 Git 垃圾回收（GC）清掉。

分支只是「指到某個 commit」的名稱

你可以用 `git branch` 動態為任何 commit 建立分支：

```
git branch recover-commit abc1234
```

這個 `abc1234` 本來不是任何分支，但現在你「取了個名字」給它，它就變成你認為的「分支」。

實際演示：證明 commit 不一定屬於分支

```
git init
echo "init" > a.txt
git add .
git commit -m "init commit"
git checkout HEAD^ --detach
echo "detached" > a.txt
git commit -am "detached commit"
git log --oneline --decorate
```

你會看到第二個 commit 沒有任何分支名稱指向它，只是個孤兒 commit。

所以回到你的問題：

「每個 commit 不都是從某個分支出來的嗎？」

表面上是，但實際上：

- commit 是獨立存在的節點
- 分支是後來「指」過去的
- commit 是內容，分支是參考名稱（reference）

就像硬碟上的檔案可以沒被任何資料夾引用一樣，但它依然存在（直到被清理）

類比理解

Git 概念	類比成電腦系統
commit	實際的檔案內容（在磁碟上）
分支（branch）	檔案捷徑（shortcut）
HEAD	你正在開啟的那個捷徑
detached HEAD	你直接打開某個檔案（沒從捷徑）

【Git】head, tag, commit, branch 的關係

這是一個非常關鍵的 Git 理解核心：
`HEAD`、**`tag`**、**`commit`**、**`branch`** **彼此的關係**。
我幫你用「定義 + 關聯圖 + 對照表 + 比喻」完整整理如下：

🧠 一張圖：理解 Git 中 HEAD、分支、Tag、Commit 的關係

🔍 核心定義

名稱	定義說明
commit	Git 的基本單位，是一個「版本快照」，有父子連結，不會變動
branch	指向某個 commit 的「可移動指標」，像捷徑，可以新增、移動
HEAD	代表你目前工作的「游標」，通常指向一個分支（或直接指向 commit）
tag	指向某個 commit 的「不可變名稱」（常用來標記版本，如 `v1.0.0`）

🔗 它們的關係總結

關係	說明
`branch` → `commit`	分支就是指向某個 commit 的「指標」
`HEAD` → `branch` or `commit`	HEAD 是你「目前所在的位置」，可以指向分支或直接指向 commit（detached HEAD）
`tag` → `commit`	Tag 永遠指向某個 commit，是不可變的
commit 之間	commit 是鏈狀連結（每個有一個 parent commit）

類比說明（像檔案系統）

Git 概念	類比	說明
commit	檔案版本	是實際的內容快照（不可變）
branch	資料夾捷徑	可以移動、可以新建、可以切換
HEAD	滑鼠游標	你目前的編輯焦點（在哪個捷徑/檔案上）
tag	唯一標籤貼紙	永久貼在某個版本上，表示這是一個重要的版本點

補充：變動性比較

元件	是否可變動	常見操作
commit	不可變	一旦建立就不會改變
branch	可變動	可以移動、刪除、重新建立
tag	可刪除（內容不可變）	可以刪除再重建，但本身是不可移動的指標
HEAD	一直在動	跟著你切換分支、commit 而變動

常見操作對應關係

操作說明	HEAD 變動	branch 變動	commit 變動	tag 變動
`git switch dev`				
`git commit`		（會跟著 HEAD 一起動）	（新增）	
`git reset --hard HEAD^`			（可能丟失）	
`git tag v1.0`				
`git checkout <commit>`	（分離）			
`git switch -c new-feature`		（新建）		

總結一句話：

> - **commit** 是內容
> - **branch/tag** 是「名字」，用來指向某個 commit
> - **HEAD** 是「你目前在哪裡」，通常跟著分支動，也可以離開分支進入 detached 狀態

【Git】git revert <commit> 和 git checkout <commit> 差異

`git revert <commit>` 和 `git checkout <commit>` 雖然都能「回到某個 commit」，但本質上用途完全不同。以下為詳細說明與比較。

一、`git revert <commit>`

功能說明：

- 用來**「反向操作」某個已提交的 commit**。
- Git 會建立一個新的 commit，內容是將指定的 commit 所做的變更**還原回去**。
- 原本的 commit 仍然會保留在歷史中。

特性：

- **安全**，適合多人協作情境。
- **保留歷史紀錄**。
- 可選擇 revert 多個 commit（可搭配 `-n` 進行批次處理）。

使用情境：

- 已經 push 到遠端，但發現某個 commit 有誤，需要「撤銷」那次變更，但不想刪除歷史。
- 緊急修復 bug，並希望讓其他人看見這次的「撤銷行為」。

範例：

```
git revert a1b2c3d
```

會產生一個新的 commit，把 `a1b2c3d` 的內容還原。

二、`git checkout <commit>`（或 `git switch --detach <commit>`）

功能說明：

- 將當前工作目錄切換到某個過去的 commit（**進入 detached HEAD 狀態**）。
- 不會建立新的 commit，純粹只是瀏覽某個歷史狀態。
- 如果在此狀態下修改檔案並 commit，會產生「無分支」的 commit（容易遺失）。

特性：

- 不會變動歷史，也不會建立新 commit。
- 適合測試、比較、build 老版本。

使用情境：

- 想**檢查過去某次版本的內容**、debug、測試、build。
- 需要比較不同版本的行為。

範例：

```
git checkout a1b2c3d
# 或推薦用法（Git 2.23+）
git switch --detach a1b2c3d
```

三、兩者比較表

指令	動作類型	是否建立新 commit	是否修改歷史	是否可安全推送遠端	常見用途
`git revert <commit>`	**建立反向 commit**	✅ 是	❌ 否	✅ 是	撤銷已推送的錯誤 commit
`git checkout <commit>`	**切換版本**	❌ 否	❌ 否	⚠️ 不適合推送 detached commit	查看歷史、測試、debug

四、補充說明：若想基於某個舊 commit 建立新分支

```
git checkout -b new-branch a1b2c3d
```

這樣可以保留該版本的狀態，並在其上繼續開發，而不會讓 HEAD 處於 detached 狀態。

如果你有實際情境（例如：某個 commit 上錯 production，或想回復某功能），可以提供情況，我能幫你判斷該用 revert 還是 checkout 或其他方案。

【Git】常用指令

Git 進階指令速查表（實戰版）

類別	指令	說明	常見情境
Merge	`git merge branch`	合併 branch	feature 合併到 main
Rebase	`git rebase branch`	重排 commit	整理 commit 歷史
Rebase	`git rebase -i HEAD~3`	互動式 rebase	squash commit
Cherry-pick	`git cherry-pick <commit>`	套用指定 commit	hotfix
Revert	`git revert <commit>`	回復某 commit	線上 rollback
Reflog	`git reflog`	查看 HEAD 歷史	找回被 reset 的 commit

Rebase vs Merge

操作	指令	結果
merge	`git merge main`	保留 merge commit
rebase	`git rebase main`	重新排列 commit

Merge

```
A---B---C main
    \
     D---E feature
         \
          F (merge)
```

Rebase

```
A---B---C main
         \
          D'---E' feature
```

結論

操作	使用情境
merge	保留歷史
rebase	整理 commit

Interactive Rebase（整理 commit）

```
git rebase -i HEAD~3
```

會出現

```
pick a1b2 commit1
pick c3d4 commit2
pick e5f6 commit3
```

可修改為

```
pick a1b2 commit1
squash c3d4 commit2
squash e5f6 commit3
```

結果：

```
commit1 + commit2 + commit3
變成一個 commit
```

常用於：

```
PR 整理 commit
```

Cherry-pick（套用某 commit）

```
git cherry-pick <commit-id>
```

用途：

情境	範例
hotfix	從 dev 複製到 main
patch	套用某 bug fix

例如：

```
dev
A--B--C--D

main
A--B
```

```
git cherry-pick D
```

結果

```
main
A--B--D
```

Revert（安全 rollback）

```
git revert <commit>
```

Git **會產生一個 反向 commit**

例如：

```
A--B--C
```

revert B

```
A--B--C--D
      ↑
```

```
undo B
```

 適合

```
production rollback
```

Reflog（救命指令）

查看 HEAD 歷史

```
git reflog
```

例如

```
a1b2 HEAD@{0}: reset: moving to HEAD~1
c3d4 HEAD@{1}: commit: fix bug
```

恢復：

```
git reset --hard c3d4
```

很多工程師不知道：

```
Git 幾乎不會真的丟資料
reflog 都能救回
```

Detached HEAD

發生情況

```
git checkout <commit-id>
```

Git 會顯示

```
You are in 'detached HEAD' state
```

代表：

```
HEAD 沒有 branch
```

解法

建立 branch：

```
git checkout -b new-branch
```

Clean（清除未追蹤檔案）

查看將刪除

```
git clean -n
```

刪除

```
git clean -fd
```

常用於：

```
CI pipeline
重新 build
```

Force Push

```
git push -f
```

⚠ 危險

可能覆蓋他人 commit

安全方式：

```
git push --force-with-lease
```

這會檢查遠端是否被更新。

Reset / Restore / Checkout 差異

指令	影響範圍
reset	commit + staging
restore	working tree
checkout	branch 或檔案

Git 三層結構（核心概念）

Git 有三個區域：

```
Working Directory
↓
Staging Area (Index)
↓
Repository (Commit)
```

對應指令：

操作	指令
working → staging	`git add`
staging → repo	`git commit`
repo → working	`git checkout`
repo → staging	`git reset`

DevOps 常用 Git 重置流程

很多 CI pipeline 會用：

```
git fetch origin
git reset --hard origin/main
git clean -fd
```

作用：

```
確保 workspace 乾淨
```

Git Flow（常見分支策略）

常見 branch：

```
main
develop
feature/*
hotfix/*
release/*
```

流程：

```
feature -> develop
develop -> release
release -> main
hotfix -> main
```

Git Debug 指令

指令	用途
`git blame file`	查看誰修改
`git bisect`	找 bug commit
`git show`	查看 commit
`git diff commit1 commit2`	比較 commit

工程師最強 Git 指令 TOP 10

指令	用途
`git log --oneline --graph`	查看歷史
`git reflog`	救 commit
`git rebase -i`	整理 commit
`git cherry-pick`	套 commit
`git revert`	rollback
`git clean -fd`	清 workspace
`git reset --hard`	重置
`git stash`	暫存
`git blame`	找修改人
`git bisect`	找 bug

給你一個小建議

如果你常用 Git（你目前 DevOps / CI / Jenkins workflow 看起來是這樣），
最推薦記住 **這 6 個核心指令**：

```
git log --oneline --graph
git reflog
git rebase -i
```

```
git cherry-pick
git reset --hard
git clean -fd
```

-

這份會非常接近 **資深工程師 Git 操作指南**。