4. 브랜치와 병합

브랜치 생성과 전환

✅ 1. 브랜치란?

> Git에서 브랜치(branch)는 커밋의 **가리키는 포인터**이자, **개발의 분기점**이다.

- 기본 브랜치: `main` (또는 예전엔 `master`)
- 새로운 기능, 버그 수정, 실험 등을 브랜치에서 따로 작업한 뒤 병합함
- Git 브랜치는 매우 **가볍고 빠르다** → 자유롭게 만들고 지우는 걸 권장

✅ 2. 브랜치 목록 확인

```
git branch
```

- 현재 존재하는 로컬 브랜치 목록 출력
- 현재 위치한 브랜치는 `*`로 표시됨

✅ 3. 브랜치 생성

```
git branch <브랜치명>
```

- 현재 브랜치를 기준으로 새 브랜치를 생성 (전환은 안 됨)

예시:

```
git branch feature/login
```

✅ 4. 브랜치 전환

```
git checkout <브랜치명>
```

- 현재 작업 중인 브랜치를 다른 브랜치로 전환
- **작업 중인 파일이 깨끗해야(safe) 전환 가능**

예시:

```
git checkout feature/login
```

✅ 5. 생성 + 전환 한 번에 (`-b`)

```
git checkout -b <브랜치명>
```

- 브랜치를 만들고 즉시 그 브랜치로 전환

예시:

```
git checkout -b feature/signup
```

✅ 6. Git 2.23 이상: `git switch` 명령

> `git checkout`은 기능이 너무 많아서 헷갈린다는 이유로
> Git 2.23부터는 **전환과 생성**을 명확히 분리해서 쓸 수 있게 되었어.

작업	명령어
브랜치 전환	`git switch 브랜치명`
새 브랜치 생성 + 전환	`git switch -c 브랜치명`

예시:

```
git switch -c feature/profile
```

✅ 7. 브랜치 삭제

```
git branch -d <브랜치명>     # 병합된 브랜치만 삭제 가능
git branch -D <브랜치명>     # 강제 삭제 (병합 여부 무시)
```

✅ 8. 원격 브랜치 전환

```
git checkout -b 브랜치명 origin/브랜치명
```

- 원격에 있는 브랜치를 로컬로 내려받고 추적하는 브랜치로 전환

예시:

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

✅ 9. 브랜치 생성 시점 커밋 지정

```
git branch 새브랜치 커밋해시
```

- 특정 커밋을 기준으로 브랜치를 생성 가능

```
git branch hotfix abc1234
```

✅ 10. 실무 브랜치 이름 컨벤션 (권장)

prefix	용도
`feature/`	새로운 기능 개발
`bugfix/` or `fix/`	버그 수정
`hotfix/`	긴급 수정
`release/`	배포 준비
`refactor/`	리팩토링
`test/`	테스트용 실험 브랜치

✅ 마무리 요약

목적	명령어	설명
브랜치 목록	`git branch`	현재 브랜치 보기
브랜치 생성	`git branch new`	새 브랜치 생성
브랜치 전환	`git checkout bname` / `git switch bname`	브랜치 이동
생성+전환	`git checkout -b` / `git switch -c`	새 브랜치 생성 후 이동
원격 브랜치 전환	`git checkout -b bname origin/bname`	원격 브랜치 추적 설정
브랜치 삭제	`git branch -d/-D`	병합 여부에 따라 삭제

병합(Merge)

✅ 1. 병합(Merge)이란?

> 두 브랜치의 **공통 조상 이후의 변경 내용**을 현재 브랜치에 통합하는 작업
> 작업한 기능을 `main` 브랜치로 반영하거나, 최신 변경을 기능 브랜치로 가져올 때 사용

✅ 2. 병합 기본 구조

```
git checkout main
git merge feature/login
```

이 작업은 **main 브랜치에** `feature/login` **의 내용을 반영**하는 것
즉, 병합은 항상 `현재 브랜치`에 다른 브랜치의 내용을 **합쳐넣는 방향성**을 가짐

✅ 3. 병합의 종류

🔹 3.1 Fast-forward Merge (단순 병합)

- 브랜치가 분기된 후에 다른 브랜치에서만 작업했을 때, **그냥 포인터만 옮김**
- 커밋 히스토리가 직선으로 이어짐

```
# 브랜치 분기 없음 → 그냥 main이 feature/login을 따라감
main
  ↓
  A---B---C  ← feature/login
```

```
git checkout main
git merge feature/login
```

→ 결과: `main` 포인터가 `feature/login`으로 이동

🔹 3.2 3-way Merge (비 Fast-forward 병합)

- 두 브랜치 모두에서 커밋이 발생한 경우
- 공통 조상(commit base) + 두 브랜치 비교 → 병합 커밋 생성

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

→ 병합 시 **새로운 병합 커밋(M)** 이 생김

```
git checkout main
git merge feature
```

→ 결과:

```
     A---B---C-----M (main)
    /             /
D---E---F (feature)
```

✅ 4. 병합 명령어

```
git merge <브랜치명>
```

- 현재 브랜치에 `<브랜치명>`의 변경 내용을 병합

✅ 5. 병합 충돌(Merge Conflict)

> 두 브랜치에서 **같은 파일의 같은 줄을 수정**했을 때 충돌 발생

```
Auto-merging hello.js
CONFLICT (content): Merge conflict in hello.js
```

🔹 해결 방법

1. `<<<<<`, `=====`, `>>>>>` 로 표시된 충돌 내용을 **직접 수정**
2. 충돌 해결 후:

```
git add <충돌 해결된 파일>
git commit   # 또는 git merge --continue
```

> ⚠️ 충돌이 발생하면 Git은 자동으로 커밋을 하지 않음 → 사용자가 해결 후 직접 commit 해야 함

✅ 6. 병합 취소

병합 커밋이 아직 완료되지 않았다면:

```
git merge --abort
```

Fast-forward라면:

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

✅ 7. `--no-ff` 옵션: 병합 커밋 강제 생성

```
git merge --no-ff feature/login
```

- Fast-forward 상황에서도 **명시적인 병합 커밋**을 남김
- 실무에서는 변경 기록을 명확하게 하려고 이 옵션을 많이 사용

✅ 8. 병합 전략 요약

전략	설명	커밋
Fast-forward	포인터만 이동	병합 커밋 없음
3-way merge	서로 다른 조상에서 병합	병합 커밋 생김
`--no-ff`	강제로 병합 커밋 생성	항상 커밋 생김
Merge Conflict	충돌 발생 시 수동 해결 필요	직접 commit

✅ 실무 병합 흐름 예시

```
# 최신 main 브랜치로 이동
git checkout main
git pull origin main

# 병합
git merge feature/signup

# 충돌 발생 시 수정 → add → commit
```

> 병합 후 원격 푸시:

```
git push origin main
```

✅ 마무리 요약

명령어	설명
`git merge <branch>`	현재 브랜치에 대상 브랜치 병합
`--no-ff`	병합 커밋 강제 생성
`--abort`	병합 중단 및 이전 상태 복구
충돌 해결	충돌 파일 수정 → `git add` → `git commit`

충돌(Conflict) 해결

✅ 1. 충돌(Conflict)이란?

> **두 브랜치에서 동일한 파일의 동일한 라인**을 서로 다르게 수정한 경우,
> Git은 자동 병합을 하지 못하고 충돌을 발생시킴.

> 즉, **Git이 무엇을 선택해야 할지 판단할 수 없는 상황**이다.

✅ 2. 충돌이 발생하는 상황

상황	예시 명령어
브랜치 병합	`git merge`
리베이스	`git rebase`
체리픽	`git cherry-pick`
stash 복원	`git stash pop`
pull with rebase	`git pull --rebase`

✅ 3. 충돌 발생 예시

```
git merge feature/login
```

```
Auto-merging login.js
CONFLICT (content): Merge conflict in login.js
Automatic merge failed; fix conflicts and then commit the result.
```

✅ 4. 충돌 시 Git의 표시 형식

Git은 충돌한 파일의 충돌 부분을 아래와 같이 표시함:

```
<<<<<<< HEAD
const login = () => { console.log("로그인1"); }
=======
const login = () => { console.log("로그인2"); }
>>>>>>> feature/login
```

영역	의미
`<<<<<<< HEAD`	현재 브랜치에서의 변경 내용
`=======`	기준선
`>>>>>>> feature/login`	병합하려는 브랜치의 내용

✅ 5. 충돌 해결 순서 (핵심 로직)

① 충돌 파일 수동 수정

```
1. <<<<<<< ~ >>>>>>> 영역을 삭제
2. 어떤 버전을 유지할지 직접 선택하거나 병합해서 수정
```

예:

```
const login = () => { console.log("로그인2"); }  // 최신 코드만 유지
```

---

② 수정한 파일 스테이징

```
git add <충돌 수정한 파일>
```

---

③ 커밋 (병합 중이라면 Git이 메시지를 자동 생성함)

```
git commit
```

또는:

```
git merge --continue
```

💡 병합 중 충돌이 너무 많으면...

- 병합 취소:

  ```
  git merge --abort
  ```

- 변경사항 초기화:

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

✅ 6. GUI 툴로 충돌 해결 (추천)

툴	명령어 또는 환경
VSCode	충돌 시 자동 표시됨: “Current Change” / “Incoming Change”
SourceTree	충돌 파일 클릭 → Merge Tool
GitKraken	시각적 충돌 해결 지원
`git mergetool`	CLI 기반 충돌 편집기 실행

```
git mergetool
```

> CLI에서도 커서 기반 편집기로 충돌 해결 가능 (vimdiff 등)

✅ 7. 충돌 해결 실전 예시

```
git checkout main
git merge feature/login

# 충돌 발생!
nano login.js   # 또는 VSCode로 열기

# <<<<<<< ~ >>>>>>> 수정 후 저장

git add login.js
git commit      # 또는 git merge --continue
```

✅ 8. 충돌 없이 병합하려면?

- 가능한 서로 다른 파일에서 작업
- 같은 파일이라도 서로 다른 줄을 수정
- PR(review) 전에 최신 `main`을 먼저 병합/리베이스해서 테스트

✅ 9. 마무리 요약

단계	명령어/행동	설명
1️⃣ 충돌 확인	`git status`	충돌 난 파일 확인
2️⃣ 파일 열기	`code .`, `nano`, `vi` 등	`<<<<<<<` ~ `>>>>>>>` 영역 확인
3️⃣ 충돌 해결	직접 수정	한쪽 또는 양쪽 내용 선택
4️⃣ 스테이징	`git add`	해결한 파일만 추가
5️⃣ 커밋	`git commit`	병합 완료 처리

브랜치 삭제

✅ 1. 로컬 브랜치 삭제

🔹 기본 명령어

```
git branch -d 브랜치명
```

- 병합된 브랜치만 삭제 가능
- 병합되지 않은 브랜치를 삭제하려고 하면 Git이 경고함

🔹 병합 여부와 상관없이 강제 삭제

```
git branch -D 브랜치명
```

> `-D`는 `--delete --force`와 동일
> → 실험용 브랜치, 실수로 만든 브랜치 삭제 시 사용

✅ 예시

```
# 병합된 브랜치 삭제
git branch -d feature/login

# 강제로 삭제
git branch -D fix/tmp-crash
```

✅ 2. 원격 브랜치 삭제

🔹 명령어

```
git push origin --delete 브랜치명
```

- 로컬이 아닌 **원격(origin)** 저장소의 브랜치를 삭제함
- GitHub, GitLab, Bitbucket 등과 연동되는 원격 브랜치 제거 시 사용

✅ 예시

```
git push origin --delete feature/profile
```

> ❗ 삭제하면 다른 개발자가 `git pull`해도 해당 브랜치를 못 받음

✅ 3. GitHub 웹에서 삭제하는 방법

- Pull Request가 **merge**되면 GitHub UI에서 **"Delete branch"** 버튼이 활성화됨
- 해당 버튼 클릭 시 **원격 브랜치만 삭제**됨
  → 로컬 브랜치는 **직접 수동 삭제**해야 함

✅ 4. 로컬과 원격 동기화 (prune)

> 삭제된 원격 브랜치를 로컬에서도 목록에서 지우고 싶다면:

```
git fetch --prune
```

또는 항상 자동으로:

```
git config --global fetch.prune true
```

✅ 5. 삭제 시 주의사항

항목	설명
현재 브랜치는 삭제할 수 없음	삭제 전 `main` 또는 다른 브랜치로 이동
병합되지 않은 브랜치 삭제	`-D` 옵션 주의해서 사용
공유된 브랜치	삭제 전에 팀원과 상의 필수
GitHub 삭제 후 로컬 존재	`git branch -d` 로 직접 삭제해야 함

✅ 마무리 요약

목적	명령어	설명
로컬 브랜치 삭제	`git branch -d 브랜치명`	병합된 브랜치만
로컬 강제 삭제	`git branch -D 브랜치명`	병합 여부 상관없이 삭제
원격 브랜치 삭제	`git push origin --delete 브랜치명`	GitHub/GitLab 원격 삭제
원격 삭제 반영	`git fetch --prune`	로컬 목록 정리

`git log --oneline --graph`로 브랜치 시각화

✅ 1. 목적

> Git 커밋 로그를 **간결하게**, 그리고 **그래프 형태로 브랜치 흐름을 시각화**
> → 브랜치의 생성, 병합, 커밋 흐름 등을 한눈에 파악 가능

✅ 2. 기본 명령어

```
git log --oneline --graph
```

구성:

- `--oneline` : 커밋 로그를 한 줄 요약 (해시 + 커밋 메시지)
- `--graph` : ASCII 그래프로 브랜치 구조 표현

✅ 3. 확장 예시

```
git log --oneline --graph --all --decorate
```

옵션	의미
`--all`	모든 브랜치 포함
`--decorate`	HEAD, 브랜치 이름, 태그를 커밋 옆에 표시

✅ 4. 실행 예시 출력

```
* 8c9b1a1 (HEAD -> main) Merge branch 'feature/login'
|\
| * f1a2b3c (feature/login) feat: implement login form
| * d4e5f6g fix: login button style
|/
* a1b2c3d chore: initial project setup
```

- `*` : 커밋
- `|`, `/`, `\` : 병합/분기 표현
- `(HEAD -> main)` : 현재 브랜치가 `main`을 가리킴

✅ 5. 예제 시나리오

```
# 1. 초기 커밋
echo init > file.txt
git add .
git commit -m "chore: init"

# 2. 브랜치 생성
git checkout -b feature/test
echo test > test.txt
git add .
git commit -m "feat: add test file"

# 3. main으로 돌아와서 병합
git checkout main
git merge feature/test
```

```
git log --oneline --graph --all --decorate
```

결과:

```
*   9e8fabc (HEAD -> main) Merge branch 'feature/test'
|\
| * 3a4bcde (feature/test) feat: add test file
|/
* 12d3f45 chore: init
```

✅ 6. 출력 예쁘게 설정 (alias 추천)

`.gitconfig`에 등록:

```
[alias]
    lg = log --oneline --graph --all --decorate
```

사용법:

```
git lg
```

✅ 7. 유용한 조합

명령어	기능
`git log --graph --oneline --decorate --all`	전체 브랜치 시각화
`git log --graph --oneline --branches`	현재 로컬 브랜치들만
`git log --graph --oneline --remotes`	원격 브랜치들만
`git log --graph --oneline --since="1 week ago"`	최근 일주일 그래프

✅ 마무리 요약

옵션	기능
`--oneline`	커밋을 요약된 한 줄로 표시
`--graph`	브랜치 분기/병합 구조 시각화
`--decorate`	브랜치 이름, HEAD 표시
`--all`	모든 브랜치 로그 출력