15. 고급 기능 및 자동화

`git stash`로 변경사항 임시 저장

1️⃣ 기본 개념

- `git stash`는 **작업 중인 변경사항(스테이지된 것 + 워킹 디렉토리 변경사항)을 임시로 저장소 밖으로 옮기는 기능**이다.
- "현재 작업 중인 내용은 잠깐 보관하고 다른 브랜치로 작업 전환 후 다시 복원"할 때 주로 사용.

핵심 원리

✅ 현재 변경사항 → 임시 영역(stash stack)에 저장
✅ 워킹 디렉토리 → clean 상태로 되돌림
✅ 필요 시 다시 적용 가능

2️⃣ 기본 사용법

변경사항 임시 저장

```
git stash
```

- 기본으로 **스테이지된 변경사항 + 워킹 디렉토리 변경사항 모두 stash** 됨.

```
git stash save "작업 설명"   # Git 2.15 이하 호환용
```

Git 2.15 이후는 그냥 `git stash push -m "작업 설명"` 사용 추천:

```
git stash push -m "작업 설명"
```

Stash 목록 확인

```
git stash list
```

출력 예시:

```
stash@{0}: On main: WIP on main: 작업 설명
stash@{1}: On main: WIP on main: 또 다른 작업 설명
```

Stash 복원 (pop)

```
git stash pop
```

- **가장 최근 stash를 적용하고 제거** (스택에서 제거됨)

Stash 복원 (apply)

```
git stash apply stash@{1}
```

- 특정 stash를 적용 **하지만 스택에서는 제거하지 않음**

Stash 삭제

```
git stash drop stash@{0}
```

Stash 전체 삭제

```
git stash clear
```

3️⃣ 고급 사용법

특정 파일만 stash 하기

```
git stash push -p
```

- **patch 모드** → 파일별, hunk별로 stash할지 선택 가능 (interactively 적용 여부 확인)

Staged 변경사항만 stash 하기

```
git stash push --staged
```

Untracked 파일도 stash 하기

기본으로는 untracked 파일(추적되지 않은 파일)은 stash 대상 아님.
원한다면 옵션 사용 가능:

```
git stash push -u
```

or

```
git stash push --include-untracked
```

Ignored 파일까지 stash 하기

```
git stash push -a
```

or

```
git stash push --all
```

4️⃣ 실전 활용 시나리오

1️⃣ 급하게 다른 브랜치로 전환 필요할 때

```
git stash
git checkout hotfix-branch
# 핫픽스 작업 후
git checkout main
git stash pop
```

2️⃣ 변경사항을 유지한 채 브랜치 변경

```
git stash
git checkout feature-x
git stash pop
```

3️⃣ 여러 변경사항 중 일부만 commit 하고 나머지는 stash

```
git add file1 file2
git commit -m "Partial commit"
git stash push -p   # 남은 변경사항 중 일부 stash
```

5️⃣ Stash 내부 구조 이해

- `git stash`는 실제로는 **커밋 2개(or 3개)** 를 만들어서 저장:

이름	내용
stash@{n} WIP commit	Index 상태(스테이지 상태)
stash@{n} WIP commit^	Working directory 상태(비스테이지 상태)
stash@{n} WIP commit^^ (옵션에 따라)	Untracked / Ignored 파일 상태

→ stash도 결국 Git 내부적으로는 숨겨진 **커밋으로 저장**됨 → 되돌릴 때도 안전하게 복원 가능.

6️⃣ 주의사항

✅ `git stash pop` 후 충돌 발생 가능 → conflict 해결 후 진행
✅ `git stash clear` → 복구 불가 → **stash 적용 여부 확인 후 삭제**
✅ 여러 stash 쌓아두면 관리 어려워짐 → 적절히 사용하고 빨리 적용/처리하는 것이 좋음

7️⃣ Stash vs Branch 차이

항목	git stash	git branch
목적	현재 변경사항 임시 보관	독립적인 작업 영역 분리
사용 대상	잠깐 작업 중단 시	기능 단위 작업 시
복원 방법	pop/apply	checkout + merge/rebase
내부 구조	숨겨진 커밋 스택	브랜치 커밋 트리

→ 브랜치에 commit 하기 애매할 때 → `stash` 사용
→ 기능 단위로 개발 → 브랜치 사용

정리

명령어	설명
`git stash push`	변경사항 임시 저장
`git stash list`	stash 목록 확인
`git stash pop`	최근 stash 복원 + 삭제
`git stash apply stash@{n}`	특정 stash 복원 (삭제 X)
`git stash drop stash@{n}`	특정 stash 삭제
`git stash clear`	모든 stash 삭제

활용 Tip:

✅ 빠른 context switch (작업 중 급한 브랜치 전환 등)
✅ 기능 개발 중 일부만 commit, 나머지 stash 활용
✅ 스테이지된 변경사항 / Untracked 파일까지 stash 활용 가능

`git cherry-pick`

1️⃣ 개념

- `git cherry-pick`은 **특정 커밋(하나 또는 여러 개)을 현재 브랜치 위로 적용(복사)하는 기능**이다.
- **전체 브랜치를 merge/rebase 하는 것이 아니라, "필요한 커밋만 골라서 적용"**하는 경우에 사용.

용도

✅ 기능은 main에 merge 안 하지만 특정 버그 fix만 release 브랜치에 적용
✅ 여러 feature 브랜치에서 유용한 커밋만 main에 골라 적용
✅ Hotfix 커밋을 과거 버전 브랜치에도 적용

2️⃣ 기본 사용법

```
git cherry-pick <커밋 해시>
```

- 해당 커밋 하나가 현재 브랜치 위에 새로운 커밋으로 적용됨

예시

```
git checkout release-1.0
git cherry-pick a1b2c3d4
```

→ `release-1.0` 브랜치에 `a1b2c3d4` 커밋 내용이 **새로운 커밋**으로 생성됨
→ 커밋 해시가 유지되는 것이 아니라 **새 커밋이 생김** → rebase처럼 rewrite 아님

3️⃣ 여러 커밋 cherry-pick

```
git cherry-pick <해시1> <해시2> <해시3>
```

or **연속된 커밋 범위 적용**:

```
git cherry-pick <시작 커밋>^..<끝 커밋>
```

주의: `시작 커밋`은 **제외**, `끝 커밋`까지 적용됨 (range 문법 주의 필요)

4️⃣ 충돌 발생 시 처리

- cherry-pick 도중 Conflict 발생 가능

처리 흐름:

```
git cherry-pick <해시>
# 충돌 발생 시 → 파일 수정 후:
git add <수정된 파일>
git cherry-pick --continue
```

중단하고 원상복구:

```
git cherry-pick --abort
```

→ cherry-pick 중단, 원래 상태로 복귀

---

5 고급 옵션

옵션	설명
`--no-commit`	적용하되 commit 하지 않고 staging 상태로 유지
`--edit`	커밋 메시지 수정 가능
`--signoff`	커밋에 sign-off 정보 추가 (`Signed-off-by`)
`--strategy-option theirs`	Conflict 발생 시 자동으로 `theirs` 쪽 선택

예시

```
git cherry-pick --no-commit a1b2c3d4
```

→ 수정된 내용만 staging에 올라오고 → 필요 시 추가 수정 후 직접 commit

---

6 실전 활용 시나리오

1 Bug fix cherry-pick

```
main → 최신 개발 브랜치
release-1.0 → 배포 유지보수 브랜치

main 에서 버그 fix 커밋 발생 → release-1.0에도 적용 필요
bash코드 복사git checkout release-1.0
git cherry-pick <버그 fix 커밋 해시>
```

2 Feature 브랜치에서 부분 커밋만 main에 적용

```
git checkout main
git cherry-pick <feature 브랜치 유용 커밋 해시>
```

3 긴급 Hotfix 커밋 이전 버전에도 적용

→ release-1.0 / release-2.0 / main → 동일 버그 fix 커밋 cherry-pick → 모두 적용 가능

---

7 cherry-pick vs merge vs rebase 차이

기능	적용 범위	새 커밋 생성 여부	사용 목적
merge	전체 브랜치	병합 커밋 생성 가능	전체 변경사항 병합
rebase	전체 브랜치	커밋 rewrite	깔끔한 linear history 유지
cherry-pick	**특정 커밋**	새 커밋 생성	필요한 커밋만 골라 적용

→ cherry-pick은 **정확히 필요한 커밋만 골라서 가져오고 싶을 때** 사용

---

8 주의사항

✅ cherry-pick은 새 커밋을 생성하므로 → **중복 커밋 주의** (같은 변경사항 여러 번 merge 되는 경우 발생 가능)
✅ cherry-pick한 커밋은 원 커밋과 해시가 다름 → Rebase 시나 merge 시 Conflict 발생 가능성 존재
✅ 팀에서 cherry-pick 사용 시 → **"이 커밋 cherry-pick 적용됨" 주석 남기면 추후 혼동 방지**
✅ Pull Request에 cherry-pick 적용 여부 명시하는 것도 추천

정리

명령어	설명
`git cherry-pick <해시>`	특정 커밋 적용
`git cherry-pick <해시1> <해시2> ...`	여러 커밋 적용
`git cherry-pick --no-commit`	적용만 하고 commit 보류
`git cherry-pick --continue`	Conflict 해결 후 이어서 적용
`git cherry-pick --abort`	cherry-pick 작업 중단

활용 포인트:

✅ Hotfix 적용
✅ 부분적인 기능 cherry-pick
✅ 긴급 패치 다중 브랜치 반영
✅ 전체 브랜치 merge가 부담스러운 경우 활용

`git filter-branch`, `git rebase --root`

`git filter-branch`

---

1 개념

- `git filter-branch`는 **Git history(커밋 기록 전체)를 재작성(rewrite)하는 고급 명령어**이다.
- 주로 다음과 같은 작업에 사용:

✅ 실수로 민감 정보(commit에 password, key 등) 넣은 경우 **과거 커밋까지 모두 제거**
✅ 전체 author name/email 변경 (회사 이메일 바뀌었을 때 등)
✅ 특정 디렉토리/파일을 **과거 전체 커밋에서 삭제/변경/이동**
✅ Subdirectory만 남기고 리포를 슬림하게 만들기

2️⃣ 기본 구조

```
git filter-branch [옵션] HEAD
```

or 원하는 범위:

```
git filter-branch [옵션] <start-commit>..<end-commit>
```

3️⃣ 주요 옵션

옵션	기능
`--tree-filter`	워킹 디렉토리 상태 기준으로 변경 적용
`--index-filter`	인덱스(스테이지) 기준으로 변경 적용 → 빠름
`--commit-filter`	커밋 메타데이터(메시지, author 등) 변경
`--env-filter`	author name/email 등 환경 변수 변경

4️⃣ 대표 예시

민감 파일 제거 (tree-filter 사용)

```
git filter-branch --tree-filter 'rm -f secret.txt' HEAD
```

- `secret.txt` 파일이 **모든 커밋에서 제거**됨 → 해당 파일 흔적이 history에서 사라짐

Author 정보 변경 (env-filter 사용)

```
git filter-branch --env-filter '
if [ "$GIT_AUTHOR_EMAIL" = "old@email.com" ]; then
    export GIT_AUTHOR_EMAIL="new@email.com"
    export GIT_COMMITTER_EMAIL="new@email.com"
fi
' --tag-name-filter cat -- --branches --tags
```

→ old@email.com 으로 작성된 커밋들이 모두 new@email.com 으로 업데이트됨.

5️⃣ 주의사항

✅ `filter-branch`는 **히스토리 rewrite 작업** → **기존 커밋 해시가 전부 변경됨** → 강력한 주의 필요
✅ 작업 전 반드시 **백업(branch로 따로 복사)** 해두는 것이 좋음:

```
git branch backup-before-filter
```

✅ Push 시 강제 push 필요 (force push):

```
git push --force --all
git push --force --tags
```

✅ **협업 중인 리포에서는 filter-branch 사용 후 반드시 팀원과 조율 필요** → clone한 사람들은 pull 불가 상태가 됨.

6️⃣ 최신 대체 도구

- `git filter-repo` (Git 공식 추천 modern replacement) → 훨씬 빠르고 안전함
- GitHub에서 민감 정보 제거 시도 할 때도 `git filter-repo` 많이 사용

설치:

```
pip install git-filter-repo
```

`git rebase --root`

1️⃣ 개념

- `git rebase --root`는 **리포지토리 최초 커밋(root commit)부터 전체 커밋을 rebase 하는 기능**이야.
- `git rebase -i HEAD~n` → 마지막 n개 커밋만 대상으로 하는 반면
  → `git rebase --root` → **최초 커밋부터 현재까지 전부 대상으로 interactive rebase 가능**

2️⃣ 사용 용도

✅ Repository 전체 history **깔끔하게 정리**
✅ 과거 commit squash → 적은 수의 의미있는 commit으로 통합
✅ 최초 커밋 message 변경 (보통 최초 커밋은 직접 수정이 어려운데 --root로 가능)
✅ PoC 단계에서 급하게 개발 후 → 최종 정제된 history 만들 때 활용
✅ Subtree split 이후 새 repo 깔끔하게 history 정리

3️⃣ 사용법

```
git rebase -i --root
```

→ interactive rebase editor 열림

```
pick a1b2c3 first commit
pick b2c3d4 second commit
pick c3d4e5 third commit
...
```

- pick → squash, reword, edit 등으로 변경 가능

예:

```
pick a1b2c3 first commit
squash b2c3d4 second commit
squash c3d4e5 third commit
```

→ first commit 한 줄로 squash, 메시지는 편집 가능

4️⃣ 주의사항

✅ `git rebase --root`는 **새로운 root commit 생성 X** → 기존 root commit부터 rewrite 가능
✅ 기존 커밋 해시 전부 변경됨 → `filter-branch`처럼 force push 필요:

```
git push --force
```

✅ 협업 중인 경우 반드시 팀원과 조율 후 사용 (Pull conflict 발생 가능)

5️⃣ `filter-branch` vs `rebase --root` 비교

항목	`git filter-branch`	`git rebase --root`
주요 목적	Commit 내용/메타데이터/파일 구조 전체 수정	Commit 순서 변경, squash, 메시지 수정
속도	느림 (큰 리포는 오래 걸림)	빠름
유연성	매우 높음 (모든 변경 가능)	주로 Commit 메시지/순서/내용 정리
복잡도	높음 (명령어 복잡)	중간 (interactive rebase 익숙하면 쉬움)
대체 도구	git filter-repo	기본 Git 지원

정리

명령어	주요 용도	주의사항
`git filter-branch`	History에서 파일 삭제, author 수정, 디렉토리 제거 등	전체 커밋 해시 변경, force push 필요
`git rebase --root`	전체 history 깔끔하게 squash/reword/reorder	커밋 해시 변경, force push 필요

추천 활용 패턴:

- 민감 정보 실수 commit → `filter-repo` or `git filter-branch`로 완전 제거
- PoC 개발 후 깔끔한 history → `git rebase --root`로 squash/reword 후 main branch merge
- Fork한 리포에서 subdirectory만 따서 새 리포 만들고 history 정리 → `git filter-repo` + `git rebase --root` 조합 사용

`.gitattributes` 설정

1 `.gitattributes`란?

- `.gitattributes` 파일은 **Git에서 특정 파일/디렉토리에 대해 특별한 속성(attribute)을 설정**하는 메타데이터 파일이다.
- 주로 사용되는 기능:

✅ Line ending(eol) 자동 변환 → cross-platform 호환성 확보 (Windows ↔ Linux ↔ macOS)
✅ 특정 파일에 대해 diff/merge 전략 변경 → 바이너리 파일, Word, PDF 등 처리
✅ 특정 파일 export 제외 → `git archive` 사용 시
✅ Linguist override → GitHub language detection 수정
✅ Text normalization (자동으로 텍스트 파일 인코딩 관리)

→ `.gitattributes`는 **리포지토리 루트에 위치**(필수 아님, 하위 디렉토리에도 가능)
→ Git clone 시 자동으로 반영됨

---

2 기본 문법

```
<패턴> <속성 설정>
```

예시:

```
*.txt text
*.sh eol=lf
*.bat eol=crlf
*.jpg binary
*.zip binary
```

---

3 주요 속성

1 eol(Line ending 처리)

설정	의미
text	텍스트 파일 → CRLF/LF 변환 가능 (OS 별)
text=auto	Git이 자동으로 텍스트 여부 판단
eol=lf	강제로 LF 사용 (Linux/macOS 스타일)
eol=crlf	강제로 CRLF 사용 (Windows 스타일)

예시

```
*.sh text eol=lf
*.bat text eol=crlf
```

→ 플랫폼에 관계없이 지정된 Line Ending 유지 → **윈도우에서 LF 필요할 때 필수**

2️⃣ binary

```
*.png binary
*.jpg binary
*.zip binary
```

- 해당 파일은 **텍스트 diff 불가**, 자동 변환 안 함
- Git diff/merge 시 "binary files differ" 로만 표시

→ 바이너리 파일은 반드시 `binary` 속성 지정 추천

3️⃣ diff

```
*.md diff=markdown
*.py diff=python
*.conf diff
```

- diff driver 를 지정해서 해당 파일 diff 처리 방식을 Git에게 알려줌
- `.git/config` 또는 `~/.gitconfig` 에 driver 정의 필요 가능

```
[diff "markdown"]
    textconv = pandoc -t plain
```

→ markdown diff 를 plain text로 보기 쉽게 변환

4️⃣ merge

```
*.lock merge=ours
```

- merge 시 어떤 전략을 쓸지 설정 가능

전략	의미
ours	**현재 브랜치** 내용 우선
theirs	상대방 브랜치 내용 우선
custom driver	사용자 정의 merge driver 사용

→ lock 파일 충돌 방지 등에서 많이 사용

---

5 export-ignore

```
secret.txt export-ignore
```

- `git archive` 명령으로 tar/zip 생성 시 해당 파일 제외
- 패키징용으로 사용 → `.gitignore` 와는 용도가 다름

---

6 linguist 관련

```
*.json linguist-language=JavaScript
docs/ linguist-vendored
tests/ linguist-documentation
```

- GitHub 에서 리포지토리의 Language 표시 제어 가능

---

4 실전 예시 `.gitattributes`

```
# Line ending 관리
*.c text eol=lf
*.h text eol=lf
*.cpp text eol=lf
*.cs text eol=crlf

# Binary files
*.png binary
*.jpg binary
*.zip binary
*.pdf binary

# Diff 설정
*.md diff=markdown
*.html diff=html

# Merge 전략
package-lock.json merge=ours

# Export ignore
secret.txt export-ignore
config/local.yaml export-ignore

# Linguist 설정
vendor/ linguist-vendored
docs/ linguist-documentation
*.yml linguist-language=YAML
```

---

5 주요 사용 사례

✅ cross-platform 프로젝트에서 **line ending 문제 방지 (CRLF ↔ LF)**
✅ 바이너리 파일 diff/merge 처리 방지
✅ 특정 파일의 diff/merge 커스터마이징
✅ 릴리즈용 archive export 시 불필요한 파일 제거
✅ GitHub language 표시 제어

---

6 관리 주의사항

✅ `.gitattributes`는 **팀 전체가 동일하게 사용**해야 의미 있음 → 반드시 리포에 커밋해서 공유
✅ Line ending 관련 문제 발생 시 → `.gitattributes` + `.git/config` core.autocrlf 설정 반드시 일치하도록 관리
✅ 새로 추가된 파일도 `.gitattributes` 갱신 필요 (바이너리 파일 추가 시 binary 지정 등)

정리

속성	용도
`text` / `eol`	Line ending 변환 제어
`binary`	Binary 파일 diff/merge 방지
`diff`	diff driver 지정
`merge`	merge driver 지정
`export-ignore`	`git archive` 제외 대상 지정
`linguist-*`	GitHub language detection 제어

결론:

`.gitattributes`는 **Git을 고급스럽게 쓰기 위한 핵심 도구**

특히 **cross-platform 개발 / 바이너리 파일 관리 / diff/merge 커스터마이징**에서 반드시 필요

`.gitignore`와 다른 기능임 → **배포 / merge / diff 처리에 영향 주는 설정**

- # eol 처리

1 기본 개념

  - 텍스트 파일에는 줄 끝(Line Ending)이 존재
  - 운영체제 별로 Line Ending이 다름:

OS	Line Ending
Linux / macOS	LF (`\n`)

OS	Line Ending
Windows	CRLF (`\r\n`)

→ OS 별로 달라서 → 협업 시 Git이 자동으로 변환해주는 기능 필요

2️⃣ Git 기본 Line Ending 처리

core.autocrlf

설정	동작
true	Checkout 시 CRLF, commit 시 LF 로 변환 (Windows 추천 기본)
input	Checkout 시 그대로, commit 시 LF (Linux/macOS 추천 기본)
false	변환 안 함 (Line Ending 유지)

```
git config --global core.autocrlf true    # Windows
git config --global core.autocrlf input   # Linux/macOS
```

문제: 이 설정만으로는 **충분하지 않음** → **.gitattributes에서 파일별로 확실히 지정해줘야 함** → 협업 환경 통일

3️⃣ `.gitattributes` 에서 eol 지정

기본 문법

```
<패턴> text eol=<lf|crlf>
```

- `text` → 해당 파일은 텍스트 파일로 간주 → Git이 자동으로 Line Ending 처리
- `eol=lf` → Git 내부에 **LF로 저장**하고 Checkout 시에도 **LF 유지**
- `eol=crlf` → Git 내부에 **LF로 저장**하지만 Checkout 시에는 **CRLF로 변환**
- `binary` → Line Ending 변환 **금지**

4️⃣ 실전 예시

```
# 유닉스 스타일 스크립트 → LF 고정
*.sh text eol=lf
*.py text eol=lf
*.yaml text eol=lf

# Windows 전용 스크립트 → CRLF 고정
*.bat text eol=crlf
*.cmd text eol=crlf

# 일반 텍스트 → 자동 감지
```

```
*.txt text

# Binary 파일 → Line Ending 변환 금지
*.png binary
*.jpg binary
*.pdf binary
```

5️⃣ 왜 필요한가?

Line Ending 문제 예시

- Windows → CRLF로 commit → Linux에서 Checkout → 빌드/테스트 실패
- Linux → LF로 commit → Windows에서 Checkout 시 core.autocrlf 미설정 → 혼합된 Line Ending 발생 → PR diff 이상하게 보임

해결 전략

✅ `.gitattributes`로 **파일 패턴에 대해 명확히 eol 정책 고정**
✅ core.autocrlf 는 개인 설정, `.gitattributes`는 **리포지토리 전체 통일 정책** → 둘 모두 병행 사용 권장

6️⃣ 관리 포인트

✅ **스크립트 파일(.sh, .py 등)은 반드시 LF로 고정** → Windows에서도 Bash 사용 시 필요
✅ **Windows 전용 실행 파일(.bat, .cmd 등)은 CRLF로 고정** → Windows 표준 준수
✅ 일반 텍스트 파일은 text 옵션으로 **자동 관리 가능**
✅ 바이너리 파일은 반드시 binary 지정 → Line Ending 변환 금지

7️⃣ 기존 리포지토리에 적용 시 주의사항

- 기존에 Line Ending이 **혼합되어 커밋된 경우** → 적용 후 diff 폭발 가능
- 적용 전 반드시 `.gitattributes` 설정 후:

```
git add --renormalize .
git commit -m "Normalize line endings"
```

→ 기존 파일들의 Line Ending을 강제로 정상화한 후 커밋 → 이후 깨끗하게 관리 가능

정리

설정 항목	용도
`*.ext text`	텍스트 파일 자동 변환
`*.ext text eol=lf`	LF 고정 (Linux/macOS 호환)
`*.ext text eol=crlf`	CRLF 고정 (Windows 호환)

설정 항목	용도
`*.ext binary`	바이너리 파일 → 변환 금지

정책 추천:

✅ `.sh`, `.py`, `.yaml`, `.json` → LF 고정
✅ `.bat`, `.cmd` → CRLF 고정
✅ 일반 `.txt`, `.md` → text (자동)
✅ 바이너리 → binary 명시

결론:

- **Line Ending 문제는 팀 협업 시 아주 빈번하게 발생하는 진짜 고질병**
- `.gitattributes` 설정으로 **협업 환경 Line Ending을 확실하게 통일**
- 개인 설정(core.autocrlf) + `.gitattributes` 병행 사용 시 최적 효과
- 기존 레거시 프로젝트에도 적용 적극 추천 → 혼란 예방 가능

• diff/merge 전략

1️⃣ 기본 개념

diff/merge driver

- Git은 파일 diff/merge 시 **기본 텍스트 처리(diff3 기반)** 를 사용
- `.gitattributes`에서 **파일별로 diff/merge 전략을 변경**할 수 있음
- 커스텀 driver 를 설정하거나 **기본 처리 방식을 변경** 가능

→ 적용 대상:

✅ 바이너리 파일 → diff 금지
✅ 특정 형식 파일 → custom diff converter 사용
✅ config 파일 → merge 시 "ours"/"theirs" 고정 전략 사용
✅ lock 파일 → merge 시 Conflict 회피

2️⃣ diff 전략 설정

기본 문법

```
<패턴> diff[=driver-name]
```

예시

```
*.md diff=markdown
*.json diff=json
*.png diff    # no driver → binary로 처리됨
```

- diff driver-name은 **Git config에 별도로 정의 가능**
- driver 설정 없으면 기본 텍스트 diff 사용

커스텀 diff driver 설정

1️⃣ `.gitattributes` 설정

```
*.md diff=markdown
```

2️⃣ `.git/config` 에 driver 등록

```
[diff "markdown"]
    textconv = pandoc -t plain
```

→ `.md` 파일 diff 시 → `pandoc -t plain` 으로 변환한 후 diff 수행
→ 예를 들어 markdown → plain text로 변환 후 보기 좋은 diff 생성

`textconv` 사용 예시

파일 유형	textconv 사용 시 효과
Markdown	렌더링 후 보기 좋은 diff
JSON	`jq` 등을 이용해 pretty-print 후 diff
HTML	`tidy` 등을 이용해 formatted HTML로 diff
Word 문서(.docx)	`pandoc` 으로 plain text 변환 후 diff
PDF	`pdftotext` 이용해서 text 변환 후 diff (PDF diff는 default로 안 됨)

---

textconv 예시: JSON

```
*.json diff=json
```

```
[diff "json"]
    textconv = jq .    # pretty-print 후 diff
```

→ JSON diff 시 raw가 아닌 **정렬된 pretty-print 형태로 diff 출력** → 매우 유용함

3️⃣ merge 전략 설정

기본 문법

```
<패턴> merge[=driver-name]
```

기본 내장 merge 전략

전략명	효과
merge	기본 merge 처리 (diff3 기반)
ours	**항상 현재 브랜치 내용 유지**
theirs	**항상 merge 대상 브랜치 내용 사용**

예시

```
*.lock merge=ours
*.json merge
*.conf merge=theirs
```

→ `package-lock.json`, `yarn.lock`, `.lock` 파일은 팀 충돌 발생 시 → 우리 브랜치 내용 우선 사용 → Conflict 방지
→ config 파일은 → 상대방 변경 우선 적용 가능

커스텀 merge driver

1️⃣ `.gitattributes` 설정

```
*.myconf merge=myconfdriver
```

2️⃣ `.git/config` 에 driver 등록

```
[merge "myconfdriver"]
    name = Custom merge for myconf files
    driver = my-merge-driver %O %A %B %L
```

driver 스크립트는:

```
my-merge-driver base our theirs marker-size
```

→ 완전한 사용자 정의 merge 프로세스 작성 가능
→ 복잡한 config/DSL 파일 등에 유용 (예: XML config, Protobuf schema 등)

4️⃣ Binary 파일 처리

```
*.png binary
*.jpg binary
*.pdf binary
```

→ Git diff/merge 시 text 처리 금지 → "Binary files differ" 만 표시
→ 매우 중요 (그렇지 않으면 Git이 바이너리 파일에 diff 적용하려고 하다가 이상한 결과가 발생할 수 있음)

5️⃣ 실전 `.gitattributes` 예시 (diff + merge)

```
# Line Ending 관리
*.sh text eol=lf
*.bat text eol=crlf

# Binary 처리
*.png binary
*.jpg binary
*.pdf binary
*.zip binary

# Diff driver 적용
*.md diff=markdown
*.json diff=json
*.html diff=html

# Merge 전략 적용
*.lock merge=ours
*.conf merge=theirs

# 기타 설정
secret.txt export-ignore
```

6️⃣ 적용 시 주의사항

✅ diff driver → textconv 사용 시 → 로컬 환경 tool(pandoc, jq 등) 필요 → CI/CD 환경에서는 주의 필요
✅ merge driver → custom driver 사용 시 → Git Hooks와 같이 관리 추천 (Team 환경에서 driver 스크립트 공유 필요)
✅ ours/theirs merge 사용 시 → 신중하게 적용 (실수로 상대방 커밋 무시하게 될 수 있음)
✅ binary 설정은 필수 (이미지, PDF, Word 등)

정리

속성	용도	주요 예시
diff	diff 처리 전략 설정	`diff=markdown`, `diff=json`
textconv	diff 변환 커맨드 설정	`pandoc`, `jq`

속성	용도	주요 예시
merge	merge 처리 전략 설정	`merge=ours`, `merge=theirs`
merge driver	custom merge driver 설정	`merge=myconfdriver`
binary	binary diff/merge 금지	`*.png binary`

결론:

- `.gitattributes`의 diff/merge 전략 설정은 Git을 정말 **프로답게 활용하는 핵심 기술**
- **diff 개선** → Markdown/JSON 등 → 보기 좋은 diff 제공 가능
- **merge 전략 적용** → lock 파일 / config 파일에서 Conflict를 줄이고 자동 merge 효과
- **binary 파일 관리** → 반드시 binary 설정 필수

Git Hooks (`pre-commit`, `commit-msg`, `post-merge`)

Git Hooks란?

- Git Hooks = **Git 이벤트(커밋, 머지 등)가 발생할 때 Git이 자동으로 실행하는 사용자 정의 스크립트**
- Git이 기본으로 `.git/hooks/` 디렉토리에 Hook 스크립트 샘플 제공
  (`pre-commit.sample`, `commit-msg.sample`, 등)

활용 목적:

✅ 자동 품질 검사 (테스트, 린트, 포맷 적용 등)
✅ 커밋 메시지 강제 포맷 적용
✅ 특정 파일/상황에 대해 후처리 자동화

Hook 파일 작성 원칙

- `.git/hooks/<hook-name>` 에 파일 생성
- 실행 권한 부여 (Linux/Mac 필수)

```
chmod +x .git/hooks/pre-commit
```

- Hook 스크립트는 쉘스크립트 형태로 작성 (bash/zsh 등 사용 가능)
- **exit 0 → 성공, exit ≠ 0 → Git 작업 중단**

주요 Hook 3종

1️⃣ pre-commit

언제 실행?

- `git commit` 실행 → **커밋 되기 직전** 실행됨
- 대상 = **Staging된 파일(Tracked + Staged)**

주요 활용 사례

✅ 코드 린트 검사 자동 실행
✅ 코드 포맷 자동 적용 (Prettier, Black 등)
✅ 테스트 실행 → 실패 시 커밋 중단
✅ 대용량 파일 커밋 방지
✅ 금지된 문자열 검사

예시: Prettier + ESLint 적용

```
#!/bin/sh
echo "Running Prettier..."
npx prettier --check .
if [ $? -ne 0 ]; then
  echo "Prettier failed."
  exit 1
fi

echo "Running ESLint..."
npx eslint .
if [ $? -ne 0 ]; then
  echo "ESLint failed."
  exit 1
fi

echo "Pre-commit checks passed."
exit 0
```

→ 린트/포맷 실패 시 커밋 중단

2️⃣ commit-msg

언제 실행?

- `git commit` → **커밋 메시지 입력 후** → **커밋 메시지를 검사**

주요 활용 사례

- ✅ 커밋 메시지 규칙 강제 적용 (Conventional Commits 등)
- ✅ Jira/이슈 번호 prefix 강제 적용
- ✅ 커밋 메시지 빈 메시지 방지
- ✅ 금지어 검사

예시: Conventional Commits 검사

```
#!/bin/sh
commit_msg_file=$1
commit_msg=$(cat "$commit_msg_file")

pattern="^(feat|fix|docs|style|refactor|perf|test|chore|build|ci|revert)(\(.+\))?: .+"

if ! echo "$commit_msg" | grep -qE "$pattern"; then
  echo "Error: Commit message must follow Conventional Commits."
  echo "Example: feat(parser): add new feature"
  exit 1
fi

exit 0
```

→ 규칙 미준수 시 커밋 중단

---

3 post-merge

언제 실행?

- `git merge` 성공적으로 완료 → **머지 직후 자동 실행**
- Conflict 없는 경우에만 실행됨

주요 활용 사례

- ✅ package-lock.json 등 변경 시 자동 `npm install`
- ✅ 빌드 자동 수행
- ✅ LFS pull 자동 수행
- ✅ 알림 자동 발송 (Slack 등)
- ✅ 기타 자동 후처리 작업

예시: package-lock.json 변경 시 npm install

```
#!/bin/sh

if git diff --name-only HEAD@{1} HEAD | grep -q 'package-lock.json\|yarn.lock'; then
  echo "package-lock.json or yarn.lock changed. Running npm install..."
  npm install
fi
```

→ dependency 파일 변경 시 자동 install

실전 적용 시 주의사항

✅ `.git/hooks/` 디렉토리는 **Git으로 Commit 안 됨 → 팀원과 공유 어려움**
✅ Husky 등 도구 활용 → Hook을 리포지토리 내에 관리 가능
✅ **에러 발생 시 반드시 exit 1 반환** → Git 작업 중단 가능
✅ Windows / Linux / macOS 모두 동작하도록 주의 (셸 호환성 고려 필요)

Team 관리 추천 도구 → Husky

- Husky → Git Hooks를 `.husky/` **디렉토리에 Git으로 Commit 가능하게 관리**
- 리포 clone 후 팀원도 동일하게 Hook 적용 가능

설치 예시

```
npx husky-init && npm install
npx husky add .husky/pre-commit "npm test"
npx husky add .husky/commit-msg "npx commitlint --edit \$1"
```

→ `.husky/` 디렉토리에 Hook 스크립트 생성됨 → Git으로 관리 가능

정리

Hook 종류	실행 시점	주요 활용
pre-commit	커밋 직전	린트 검사, 테스트, 포맷 적용, 대용량 파일 방지
commit-msg	커밋 메시지 입력 후	커밋 메시지 규칙 검사
post-merge	merge 성공 후	빌드 자동화, dependency install, 알림 발송

결론:

- Git Hooks → **Git 작업 흐름을 품질 관점에서 자동화 가능하게 해주는 필수 고급 기능**
- `pre-commit`, `commit-msg`, `post-merge`는 가장 많이 쓰이는 핵심 3종
- 팀 협업 시 Husky 등 도구와 함께 사용하면 관리 편리
- "품질 게이트 자동화 + 반복 작업 자동화" → Git 활용 수준이 확 올라감

husky, lint-staged와의 연동

핵심 개념

도구	역할
Husky	Git Hooks 를 Git 리포지토리 내에서 관리 가능하게 해주는 도구 (`.husky/` 디렉토리 생성)

도구	역할
lint-staged	pre-commit 시 **변경된 파일만 lint/format 등 처리** → 전체 repo 전체가 아닌 staged file만 빠르게 검사/처리

→ 둘을 **연동**하면:

✅ Git Hook을 repo 내부에서 Git 관리 → 팀원 전체 적용 일관성
✅ pre-commit 단계에서 → Staged 파일만 lint/format → 속도 빠름
✅ 품질 자동화 → 린트 실패 시 커밋 자동 중단

설치 및 기본 구성

1️⃣ 설치

프로젝트 초기화 후:

```
npm install --save-dev husky lint-staged
```

2️⃣ Husky 초기화

```
npx husky install
```

→ `.husky/` 디렉토리 생성됨
→ `npm run prepare` 스크립트 추가하면 → clone 후 자동 설치 가능:

```
"scripts": {
  "prepare": "husky install"
}
```

3️⃣ pre-commit Hook 추가

```
npx husky add .husky/pre-commit "npx lint-staged"
```

→ `.husky/pre-commit` 파일 생성됨:

```
#!/bin/sh
. "$(dirname "$0")/_/husky.sh"

npx lint-staged
```

4️⃣ lint-staged 설정

`package.json` 에 추가:

```
"lint-staged": {
  "**/*.js": [
    "eslint --fix",
    "prettier --write"
  ],
  "**/*.css": [
    "stylelint --fix"
  ]
}
```

→ 의 의미:

✅ staged 된 js 파일 → eslint --fix 실행 → prettier 적용 → commit
✅ staged 된 css 파일 → stylelint --fix → commit

실전 구성 예시

```
"lint-staged": {
  "**/*.{js,jsx,ts,tsx}": [
    "eslint --fix",
    "prettier --write"
  ],
  "**/*.{json,md,css,scss}": [
    "prettier --write"
  ]
}
```

동작 흐름

```
git add . → git commit 실행 시:

1️⃣ .husky/pre-commit Hook 실행
2️⃣ lint-staged 실행 → 현재 staged 상태 파일만 대상
3️⃣ eslint → 문제 발생 시 수정 또는 커밋 중단
4️⃣ prettier → 자동 포맷 적용
5️⃣ 모든 작업 성공 → 커밋 진행
```

장점

✅ pre-commit 시 린트 오류/스타일 오류 **자동 방지** → 깨끗한 커밋 유지
✅ Staged 파일만 처리 → 속도 빠름 (전체 프로젝트 린트보다 훨씬 빠름)
✅ 팀원 Git Hooks 통일 관리 → `.husky/` 디렉토리 Git commit 가능

Best Practice 구성

```
npm install --save-dev husky lint-staged prettier eslint stylelint
```

package.json:

```
"lint-staged": {
  "**/*.{js,jsx,ts,tsx}": [
    "eslint --fix",
    "prettier --write"
  ],
  "**/*.{css,scss,json,md}": [
    "prettier --write"
  ]
},
"scripts": {
  "prepare": "husky install"
}
```

husky:

```
npx husky add .husky/pre-commit "npx lint-staged"
```

주의사항

✅ lint-staged는 **Staged 파일만 처리** → 속도 최적화
✅ prettier, eslint 설정은 프로젝트에 맞게 사전 구성 필요
✅ pre-commit Hook 강제 skip 방법:

```
git commit --no-verify
```

→ CI 테스트 실패 대응 등으로 가끔 사용 가능 (단, 남용 금지)

정리

구성요소	역할
Husky	Git Hook 관리 (.husky/)
lint-staged	pre-commit 시 Staged 파일만 처리
eslint/prettier/stylelint	린트/포맷 도구

Workflow 흐름:

```
git add → git commit →
.husky/pre-commit 실행 →
lint-staged 실행 (Staged 파일만 대상) →
eslint/prettier/stylelint 적용 →
성공 → 커밋 진행
실패 → 커밋 중단
```

결론:

- **Husky + lint-staged 조합 → 요즘 개발팀 pre-commit 품질 자동화의 거의 표준 구성**
- 린트 적용 → 코드 품질 자동 유지
- 포맷 적용 → 코드 스타일 통일
- 커밋 단계에서 걸러내기 → CI 단계보다 훨씬 빠르게 피드백 가능