12. 충돌 해결 및 디버깅

`git mergetool` 사용

✅ 1. `git mergetool`이란?

> Git merge 시 충돌이 발생했을 때,
> `git mergetool` 명령어로 **GUI 기반 또는 CLI 기반 머지 툴을 호출**해서
> 편하게 충돌을 해결할 수 있게 도와주는 명령어.

👉 일일이 텍스트 에디터로 `<<<<<<< HEAD` 같은 마크업을 수정하지 않고
3-way merge 화면에서 선택적으로 병합 가능.

✅ 2. 기본 흐름

```
git merge branchX
 → 충돌 발생 (CONFLICT)

git mergetool
 → 머지 툴 실행 → 충돌 해결 → 저장 후 종료

git commit
```

✅ 3. 기본 명령어

```
git mergetool
```

- 현재 충돌 상태인 파일 목록을 자동으로 탐색
- 설정된 머지 툴(Git config의 `merge.tool`)을 실행
- 머지 후 저장하면 자동으로 Git이 해당 파일을 `resolved` 상태로 처리

✅ 4. 사용할 수 있는 머지 도구 예시

툴	설명
`vimdiff`	CLI 기반 Vim merge 도구
`meld`	GUI 기반 머지 도구 (리눅스, 윈도우 가능)
`kdiff3`	GUI 기반, 윈도우/Linux 지원
`Beyond Compare`	강력한 유료 GUI 툴
`p4merge`	무료 GUI 툴

툴	설명
`Araxis Merge`	고급 유료 GUI 툴

✅ 5. 머지 툴 설정 방법

```
git config --global merge.tool meld
```

👉 이후 `git mergetool` 입력 시 meld가 실행됨.

기본 툴 설정 확인:

```
git config --global merge.tool
```

✅ 6. 설치 예시 (Ubuntu + Meld)

```
sudo apt-get install meld
git config --global merge.tool meld
```

✅ 7. 사용 예시 절차

① merge 시도

```
git merge featureX
# CONFLICT 발생
```

② mergetool 실행

```
git mergetool
```

③ GUI 화면에서 머지

- 보통 왼쪽: 내 커밋(LOCAL)
- 오른쪽: 병합 대상(REMOTE)
- 중간/아래: BASE (공통 조상)
- 최종 결과 편집 후 저장

④ 저장 후 Git이 자동으로 `resolved` 표시

```
git status
# 충돌 해결됨
```

⑤ commit

```
git commit
```

✅ 8. 옵션

명령어	설명
`git mergetool --tool=<툴명>`	특정 머지 툴 사용 (1회용)
`git mergetool --no-prompt`	자동으로 툴 실행 (프롬프트 생략)
`git mergetool --tool-help`	지원하는 머지 툴 목록 출력

✅ 9. `.gitconfig` 설정 예시 (GUI 툴 고정)

```
[merge]
    tool = meld
[mergetool "meld"]
    trustExitCode = false
```

- `trustExitCode = false`: 일부 툴은 종료 코드를 Git이 믿지 않음 (false 권장)

✅ 10. 사용 팁

팁	내용
설정 후 테스트	`git mergetool --tool-help` 로 툴 목록 확인
여러 파일 충돌 시	한 번 실행 후 파일별로 순차 처리
CLI vs GUI	GUI 기반 툴 추천 (`meld`, `kdiff3`, `Beyond Compare`)
자동 머지 후 확인	`git diff`, `git status` 로 머지 결과 점검 필수

🧠 요약

단계	명령
툴 설정	`git config --global merge.tool meld`
머지 시도	`git merge branch`
충돌 발생	Git이 알림 표시
머지 툴 실행	`git mergetool`

단계	명령
머지 저장	GUI/CLI에서 해결 후 저장
commit	`git commit` 으로 마무리

충돌 해결 전략 및 커뮤니케이션

1 충돌(Conflict)이란?

- Git에서 충돌(conflict)은 **동일한 파일의 동일한 부분**을 **여러 브랜치에서 다르게 수정한 경우** 발생
- 보통 **Merge** 또는 **Rebase** 시 발생
- Git은 단순한 자동 병합은 해주지만, 의미적인 충돌은 사람이 수동으로 해결해야 함

예시 상황:

```
브랜치 A에서는 main.c 5번째 줄을 "Hello World"로 수정
브랜치 B에서는 같은 줄을 "Hello Git"으로 수정
→ 두 브랜치를 merge할 때 충돌 발생
```

---

2 충돌 발생 시 Git의 동작

- Git은 해당 파일을 다음과 같이 구분해 표시:

```
<<<<<<< HEAD
Hello World
=======
Hello Git
>>>>>>> feature-branch
```

- `HEAD` 쪽은 현재 브랜치
- `feature-branch` 쪽은 병합하려는 브랜치
- **이 사이에서 올바른 최종 코드를 직접 작성**해야 함

---

3 충돌 해결 흐름

```
1. git merge 또는 git rebase 수행
2. 충돌 발생 → Git이 알려줌 (git status 확인 가능)
3. 충돌난 파일을 편집기에서 열어 수동으로 수정
4. 수정 후 파일을 git add
5. git commit (merge의 경우 자동 메시지 생성됨)
```

도구 사용:

- `git mergetool` → GUI 기반 머지 도구 지원 (예: kdiff3, meld, p4merge 등)
- VSCode 사용 시 자동으로 conflict 표시 지원

4️⃣ 충돌 예방 전략

🚥 브랜치 전략

- **작은 단위 브랜치 관리** → 변경사항이 적으면 충돌 가능성 낮음
- **브랜치 주기적 업데이트** → 자주 `git pull --rebase` 또는 `git merge`로 최신 상태 반영
- **짧은 생명주기 브랜치 사용** → 오래된 feature branch는 conflict 가능성이 높아짐

👫 팀 협업 커뮤니케이션

- 팀원 간 **작업 영역 공유** → 어느 파일, 어느 영역을 수정 중인지 가시화
- 중요한 리팩토링, 구조 변경 시 **사전 공지**
- 코드 리뷰 시 **conflict 예측 지점 논의**

🛠️ 자동화 및 정책

- PR 병합 전 자동 리베이스 → Github Actions + `auto-rebase` 설정
- PR에 conflict 발생 시 머지 제한 → Protected branch + required up-to-date branch 설정

5️⃣ 커뮤니케이션 Best Practice

- 충돌이 발생했을 때 **서로 탓하지 않는다**
- Git 충돌은 자연스러운 협업 현상 → "누구 실수 때문"이 아님
- 충돌 발생 시 Slack, Issue 등으로 **수정 방향 공유** 후 해결
- **역사(log)를 깨끗하게 유지하는 방향**으로 머지 / 리베이스 전략 결정

6️⃣ 결론

충돌 해결 = 기술 + 커뮤니케이션의 균형

기술적 측면:

- 충돌난 파일의 정확한 이해
- 필요한 경우 `git log -p`, `git blame`으로 변경 이력 분석

커뮤니케이션 측면:

- 서로의 변경 내용을 존중
- 적시에 팀과 논의하여 최적의 해결 방향 모색

`git bisect`로 문제 커밋 추적

1 개요

- `git bisect`는 **바이너리 탐색(binary search)** 알고리즘을 활용해서
- **어떤 커밋에서 버그가 발생했는지 자동으로 찾아주는 도구**이다.

2 동작 원리

```
1. 정상 동작하는 "좋은(good)" 커밋과
2. 문제가 발생하는 "나쁜(bad)" 커밋을 알려주면
3. Git이 해당 구간을 이진 탐색하며 중간 커밋으로 checkout
4. 사용자가 해당 커밋에서 문제가 있는지 없는지 판정 → good 또는 bad로 입력
5. 반복 → 결국 최초로 문제가 발생한 커밋 pinpoint
```

효율

- O(log N) → 커밋 1000개 있어도 10번 정도면 찾음

---

3 사용 절차 (실습 중심 예시)

💻 상황

```
main 브랜치에 100개 커밋이 있음.
v1.0 에서는 정상 동작 → GOOD
HEAD(v1.5)에서는 버그 발생 → BAD
```

🚀 Step by Step

```
# 1. bisect 시작
git bisect start

# 2. 현재 커밋은 문제가 있음
git bisect bad

# 3. 정상 동작했던 커밋(예: v1.0 커밋 해시)
git bisect good <커밋 해시>

# → Git이 자동으로 중간 커밋 checkout

# 4. 테스트 실행 → 버그가 있으면
git bisect bad

# 5. 테스트 실행 → 버그 없으면
git bisect good

# → Git이 다음 중간 커밋으로 이동

# 6. 반복
```

```
# → 최종적으로 문제가 처음 발생한 커밋을 출력

# 7. bisect 종료 (원래 브랜치로 복귀)
git bisect reset
```

---

4 자동화 사용

🖥️ 자동 스크립트 적용 (test.sh 예시)

```
git bisect start
git bisect bad
git bisect good <good commit hash>

git bisect run ./test.sh
```

- `test.sh`는 실행 후 `0`이면 good, `1`이면 bad로 판정됨
- → **테스트 자동화** 시 매우 유용

---

5 실전 Tips

언제 사용하나?

- 최근 merge 후 **버그 발생 시 어떤 커밋 때문인지 모를 때**
- 특정 기능/속도 저하가 **어디서부터 발생했는지 추적할 때**
- **협업 중에 다른 사람 커밋 포함하여 문제 발생 원인을 찾을 때**

주의사항

- **재현 가능한 테스트**가 반드시 필요함 (안 그러면 bisect 과정에서 판정이 애매해짐)
- 중간 커밋에서 빌드 깨지는 경우 → 건너뛰기 가능:

```
git bisect skip
```

---

6 커맨드 정리

명령어	설명
`git bisect start`	바이섹트 시작
`git bisect bad`	현재 커밋 bad 판정
`git bisect good <hash>`	특정 커밋 good 판정
`git bisect skip`	현재 커밋 스킵

명령어	설명
`git bisect run <script>`	자동 테스트
`git bisect reset`	원래 브랜치로 복구

---

7 요약

- **git bisect = 디버깅 탐색기**
- 사용법 매우 단순하지만 효과는 매우 큼
- 수작업으로 "git log → checkout → 테스트" 반복하는 것보다 훨씬 빠름
- **CI/CD 자동화에 넣으면 고성능 회귀 테스트 가능**

`git blame`, `git annotate`로 이력 추적

1 목적

- 소스코드 라인별로 **최종 수정한 커밋, 작성자, 시간**을 확인할 수 있음
- 코드 리뷰, 디버깅, 리팩토링 시 매우 자주 사용
- **"이 코드 왜 이렇게 되어 있지?" → 해당 커밋과 작성자 추적 가능**

---

2 기본 명령어

```
git blame <파일명>
```

출력 예시:

```
^3f2a89d3 (Alice   2024-05-01 12:30:45 +0900  1) public class Main {
47fa2349b (Bob     2024-06-01 09:10:22 +0900  2)     public static void main(String[] args) {
47fa2349b (Bob     2024-06-01 09:10:22 +0900  3)         System.out.println("Hello Git");
^3f2a89d3 (Alice   2024-05-01 12:30:45 +0900  4)     }
^3f2a89d3 (Alice   2024-05-01 12:30:45 +0900  5) }
```

구조:

```
[커밋 해시] (작성자  날짜  라인번호) 코드라인
```

→ 어떤 **커밋에서 어느 라인이 마지막으로 수정됐는지 확인** 가능

---

3 상세 사용법

기본 옵션

```
git blame -L <시작라인>,<끝라인> <파일명>
```

- 특정 라인 범위만 blame 보기

```
git blame -p <파일명>
```

- **porcelain 포맷** → 스크립트로 파싱하기 좋음

```
git blame --since="3 months ago"
```

- 특정 기간 이후 변경된 것만 표시

```
git blame --ignore-rev <커밋해시>
```

- **포맷팅만 변경한 커밋(예: 코드 스타일 통일)을 무시**하고 blame
  → GitHub에서도 `.git-blame-ignore-revs` 설정 지원

---

4 `git annotate`와의 관계

- `git annotate`는 **구버전 Git에서 사용하던 명령어**
- 최신 Git에서는 사실상 `git blame`과 동일 (alias임)
- 대부분의 Git UI 툴이나 GitHub Blame 기능도 내부적으로 `git blame` 사용

---

5 활용 사례

디버깅 시

```
"이 if문이 왜 들어갔지?" → blame으로 커밋 확인 → 커밋 메시지 or PR 내용 확인 → 의도 이해
```

코드 리뷰 시

```
"이 부분 리팩토링해도 되나?" → blame 확인 → 오래된 코드 or 최근에 의도적으로 수정된 코드 여부 판단
```

코드 관리 시

```
팀에서 어떤 부분이 주로 어떤 사람이 유지보수하는 코드인지 파악
```

---

6 실전 Tip

Blame → Git log 연결

```
git show <커밋 해시>
```

- blame으로 확인한 커밋 해시를 `git show`로 확인하면 **변경 전체 diff + 커밋 메시지 + 작성자 의도**까지 파악 가능

GitHub Blame

- GitHub UI → **Blame 탭** 제공 → GUI로 blame 확인 가능
- 각 라인 클릭 → 해당 커밋으로 이동 가능 → PR/리뷰 내용도 확인 가능 → **최종 책임자 추적에 매우 유용**

---

7 주의사항

- **Blame 결과 = "마지막으로 수정한 사람"**이지 최초 작성자는 아님
- 리팩토링, 포맷팅 커밋으로 인해 blame이 바뀌는 경우 많음 → `--ignore-rev` 적극 활용 추천
- 자동화된 대규모 코드 포맷팅 후 blame이 쓸모없어지는 경우 발생 → **팀 내** `.git-blame-ignore-revs` **도입** 권장

---

8 정리

목적	도구
라인별 최종 수정자, 커밋 파악	`git blame`
오래된 코드 추적	`git blame --since`
포맷팅 커밋 무시	`git blame --ignore-rev`
커밋 전체 보기	`git show <해시>`
GitHub에서 blame 시각적 확인	Blame 탭 사용

---

9 추천 워크플로우

```
코드에서 이상한 동작 발견 → blame → show → PR/커밋 메시지 확인 → 팀원과 논의 후 수정 or 유지
```