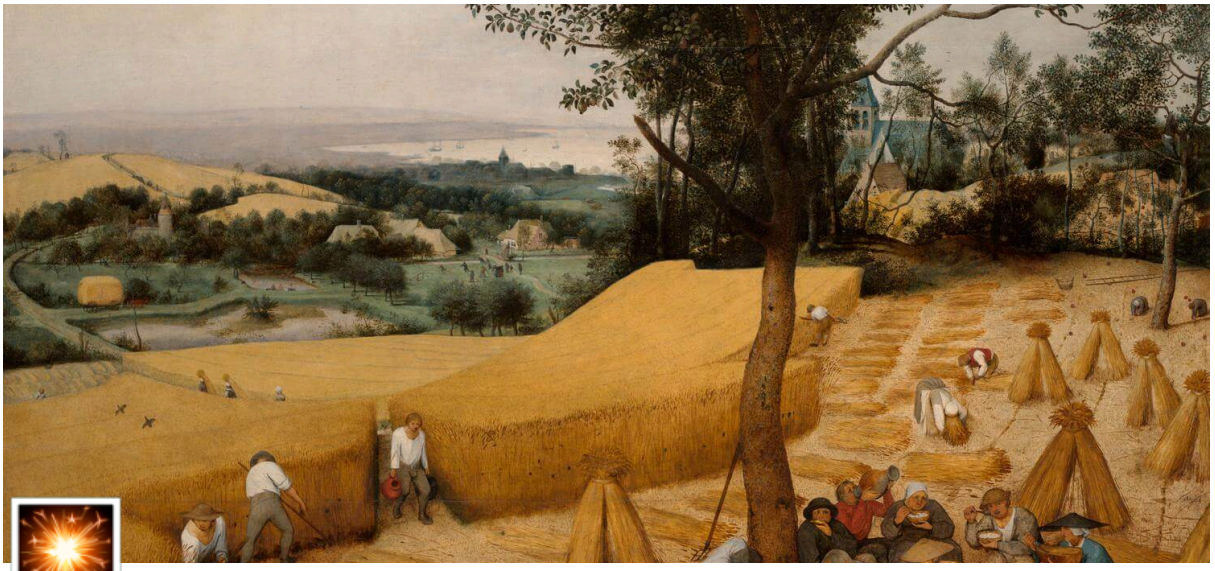

Git 에 대한 전략 및 고찰

📌 1. 머지가 꼬였을 때 어떻게 대처해야 하나?

1) 어디서 충돌이 발생했는지 파악

- 충돌이 난 파일은 `<<<<<<<`, `=======`, `>>>>>>>` 표시가 자동 생성됨

2) 충돌 파일 하나씩 해결

- 원하는 코드 선택 / 합쳐서 해결

3) 의도한 결과가 맞는지 반드시 테스트

- 빌드 / 유닛 테스트
- 로컬에서 기능 동작 확인

📌 2. 왜 충돌이 발생할까?

👥 등장인물

- **개발자 A**: 로그인 기능 담당
- **개발자 B**: 마이페이지 기능 담당
- 두 사람 모두 develop 브랜치에서 갈라진 feature 브랜치에서 작업 중

🎬 상황 1) 같은 파일, 같은 라인을 동시에 수정했을 때

📌 상황 설명

- A는 `User.java`의 `status` 필드를 수정
- B도 `User.java`의 `status`를 수정

🗣️ 상황극

개발자 A: “status를 enum으로 바꿔야지!”

개발자 B: “statusText 필드도 추가해야지.”

→ 서로 모르게 **한 파일, 한 라인**을 수정 → **충돌 발생**

🎬 상황 2) 오래된 브랜치를 유지(stale branch)

📌 상황 설명

- A는 5일 전에 브랜치 생성 후 작업
- 그 사이 develop에 20개 이상 커밋 누적

🗣️ 상황극

A: “이제 머지해야지!”

B: “UserService 구조 다 바꿨어!”

→ A가 develop을 merge하는 순간 충돌 폭발

🎬 상황 3) 공통 모듈을 동시에 수정할 때

📌 상황 설명

- A는 lastLoginAt 추가
- B는 createdAt 추가
- 둘 다 DTO, Domain 파일에 영향

→ 같은 파일 내 구조 변화 → 자동 병합 불가 → 충돌

📌 3. 충돌을 방지하는 방법

✔️ 1) 짧고 작은 단위로 브랜치 끊기

✔️ 2) 작업 시작 전 최신 develop pull

```
git pull origin develop
```

✔️ 3) 하루 1회 merge/rebase로 최신 유지

```
git pull --rebase origin develop
```

📌 Merge vs Rebase

🔀 Merge

```
A --- B --- C (develop)
      \
       D --- E (feature)
```

→ Merge 후

```
A --- B --- C ------ M
      \         /
       D --- E ---
```

✔️ 특징

- 히스토리 보존
- 충돌은 merge 시 한 번
- 커밋이 지저분해질 수 있음
- 롤백/추적 쉬움

✔️ 언제 쓰나?

- 팀 단위 협업
- 대규모 팀
- Git Flow 구조(develop → release → main)

🔧 Rebase

```
develop: A --- B --- C
feature:       D --- E
```

→ Rebase 후

```
develop: A --- B --- C
feature:               D' --- E'
```

✔️ 특징

- 히스토리가 매우 깔끔
- merge commit 없음
- 각 커밋마다 충돌 발생 가능
- **공유 브랜치에서 절대 금지**

❗ 왜 공유 브랜치에서 사용 금지인가?

히스토리가 달라지면 발생하는 문제

- pull/push가 거부됨
- fatal: Not possible to fast-forward
- 팀원 전원이 충돌
- PR merge 불가
- push -f로 타인의 커밋 삭제 위험
- 최악: 팀 전체 작업 중단

✔️ Rebase를 사용하는 경우

1) 혼자 작업하는 로컬 브랜치

→ 최신 develop 위에서 다시 쌓아 깔끔하게 유지

2) merge 전에 브랜치를 정리하고 싶을 때

→ 불필요한 커밋 묶기(squash) / 정돈

3) PR 제출 전 커밋 정리

→ 리뷰어가 보기 좋은 히스토리 유지

📌 4. 충돌이 필연적으로 발생하는 상황

- 파일 하나를 여러 명이 동시에 수정
- 대규모 리팩토링
- 패키지/폴더 전체 구조 변경
- 공통 도메인/DTO/Config 수정
- 대규모 팀에서 Slice 기반 개발 진행

🧭 5. Git Flow vs GitHub Flow

✔️ Git Flow

> main / develop / feature / release / hotfix
>
> → **대규모 팀, 안정성 중요할 때**

장점: 안정적 릴리즈 / QA 분리

단점: 브랜치 복잡, 충돌 증가

✔️ GitHub Flow

> main + feature

> → 소규모 팀, 빠른 배포 중심

장점: 단순 / 빠름

단점: 운영 안정성 관리 어려움

🔄 6. 푸시한 커밋 취소하는 법

✔️ remote까지 올라간 커밋 되돌리기

```
git revert <hash>
```

👉 협업에서는 `reset --hard + push -f` 금지

⬅️ 7. Local 커밋 취소

✔️ 최근 커밋 취소

```
git reset --soft HEAD~1

[커밋] A - B(HEAD)
git reset --soft HEAD~1
⇒ HEAD가 A로 이동
=> B의 모든 변경은 staging 상태로 유지됨
```

✔️ 여러 개 취소

```
git reset --soft <commit>
```

✔️ *완전 삭제 (절대 권장하지 않음!!! 그냥 staging으로 돌려두고 판단하에 discard하는것을 추천)*

```
git reset --hard <commit>
```

정리 표

명령어	HEAD 이동	Staging Area	Working Directory	실무 의미
`reset --soft`	이동함	유지됨	유지됨	커밋만 취소(코드는 남김)
`reset --mixed` *(기본값)*	이동함	초기화	유지됨	커밋 취소 + staging 제거
`reset --hard`	이동함	초기화	초기화	**커밋 + 코드까지 삭제 (위험)**

📦 8. Stash 활용

```
git stash
git stash list  //스태시 목록
git stash apply //스태시 적용 + 목록 유지
git stash pop   //스태시 적용 + 삭제(pop)
```

→ 급하게 브랜치를 바꿔야 하는 경우 유용

🍒 9. Cherry-pick

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

✔️ 언제 사용하는가

- 특정 커밋만 main/develop에 반영해야 할 때
  - Cherry-pick을 파워풀하게 사용하려면 rebase가 필수임
- hotfix 병행 반영
- 선택적 기능 반영

📌 시나리오 1 — 한 feature 브랜치에서만 만든 수정이 다른 브랜치에도 필요할 때

상황

- A 개발자가 `feature/payment` 브랜치에서 공통 util 버그 하나 발견
- 해당 util 은 다른 기능에도 영향 있음
- 하지만 `feature/payment` 전체를 머지하고 싶진 않음(아직 미완성)

해결

→ util 버그 수정한 특정 커밋만 따로 가져오기

```
git checkout feature/user
git cherry-pick <bugfix-commit>
```

요약

> 전체 기능은 가져오지 않고, 필요한 수정만 정확히 가져올 때 사용.

📌 시나리오 2 — Release 브랜치를 만들었는데 특정 기능만 포함시키고 싶을 때

상황

- 이번 릴리즈 스프린트에서 10개의 기능이 개발됨
- 하지만 QA 결과, 3개 기능만 먼저 출시하기로 결정

해결

→ feature 브랜치에서 릴리즈 브랜치로 **특정 기능 3개 커밋만 cherry-pick**

```
git checkout release-1.3.2
git cherry-pick <commit-A> <commit-B> <commit-C>
```

요약

> “선택적 배포”가 가능하도록 해주는 전략.

📌 시나리오 3 — 공동 작업 중인 브랜치에서 다른 사람의 특정 커밋만 재사용할 때

상황

- B 개발자가 결제 validation 로직을 먼저 구현
- A 개발자는 병원비 API에서도 똑같은 validation 로직이 필요
- 그런데 B의 로직이 아직 feature 브랜치 전체가 merge되기 전

해결

→ B의 validation 커밋만 A 브랜치로 가져옴

```
git checkout feature/hospital-bill
git cherry-pick <validation-commit>
```

▼ ⚠️ 의문

체리픽으로 특정 기능만 뽑아 쓴 경우, “나중에 merge할 때 충돌이 발생할 수 있지만, 대부분 예측 가능한 충돌이며 정상적인 충돌”입니다.

그리고 실무에서는 **이 충돌을 감수하더라도 cherry-pick을 쓰는 것이 더 효율적**인 경우가 많습니다.

아래에서 왜 그런지 현실적으로 설명드릴게요.

✅ 왜 충돌이 생길 수 있나?

시나리오 상황 복습:

- B가 `feature/payment` 브랜치에서 validation 기능을 구현함
- 하지만 전체 기능은 완성되지 않음 → 브랜치 자체는 merge하지 않음
- A는 그 validation 커밋만 미리 쓰고 싶음 → cherry-pick 실행

이후 B 브랜치가 develop으로 병합될 때:

- A가 이미 cherry-pick 해서 넣은 commit (B' 형태)과
- B의 원본 commit(B)가 같은 코드를 수정하는 커밋임
- Git은 “이거 내가 본 것 같은데... hash가 다르네?”라며 충돌 내 가능성이 있음

즉:

> 같은 코드 변경이 두 번 들어오니까 충돌 가능성은 있음.

하지만 이건 *정상적이고 예상 가능한 충돌*임.

❗ 그런데도 왜 실무에서는 이걸 자주 쓰는가?

이유 1) 시간 절약

validation 로직을 A가 직접 다시 만들면 중복 개발.

B의 기능 전체는 아직 미완성이기 때문에 기다릴 수도 없음.

➡️ 빠르게 필요한 기능만 가져와서 생산성 ↑

요약

> 공통 코드 또는 유틸 로직을 팀원 간 빠르게 공유하는 용도.

📌 시나리오 4 — 실수로 잘못된 브랜치에서 commit 한 경우

상황

- A 개발자가 `develop` 브랜치에서 실수로 커밋을 해버림
- 하지만 실제로는 `feature/proxy-v2`에 들어가야 하는 코드

해결

1. 커밋을 다른 브랜치에 cherry-pick
2. 원래 브랜치에서 revert(또는 reset)

```
git checkout feature/proxy-v2
git cherry-pick <mistaken-commit>

git checkout develop
git revert <mistaken-commit>
```

요약

> 잘못 커밋한 실수를 유연하게 복구하는 방식.

🛠️ 10. 머지 도구 활용

✔️ IntelliJ

- 충돌 파일 → `Merge...`

✔️ VSCode

- 실시간 diff + 선택 적용

✔️ Git mergetool

```
git mergetool
```

🏁 최종 정리

항목	요약
충돌 이유	같은 파일/라인 수정, 오래된 브랜치, 공통 모듈 수정
방지 방법	짧은 브랜치, rebase 활용, 사전 공유
필연적 상황	대규모 refactor, 패키지 구조 변경
Git Flow	대규모 팀 / 안정성
GitHub Flow	속도 중심 / 단순
reset	커밋 취소
revert	push된 커밋 되돌리기

항목	요약
stash	임시 저장
cherry-pick	특정 커밋만 옮기기