

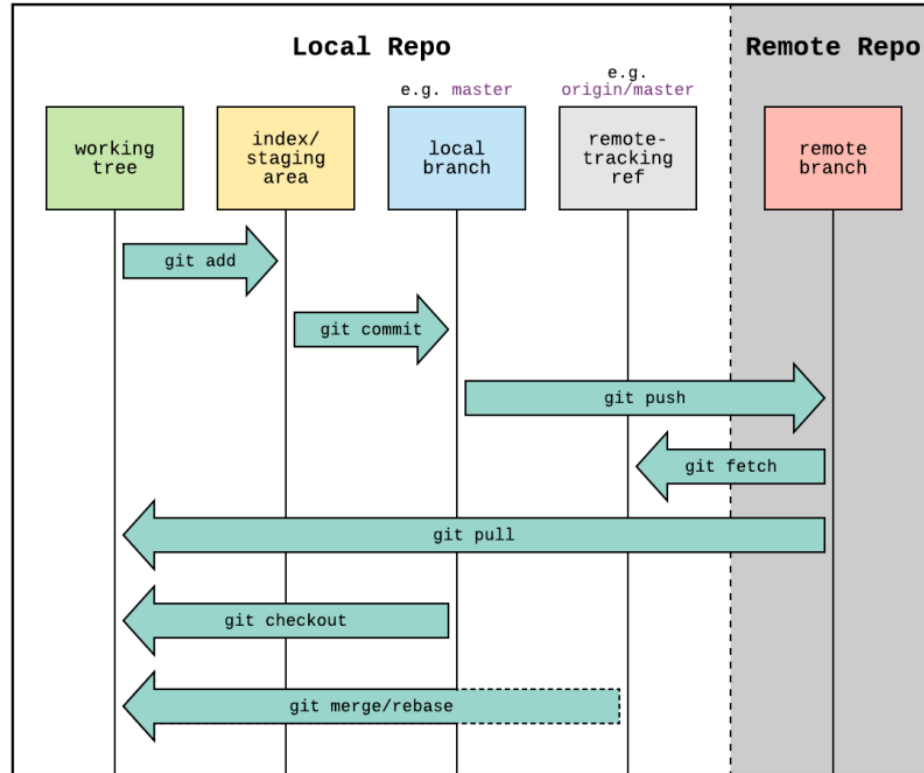
# WEB DEVELOPMENT STUDY

## 웹개발 스터디 3회차

신동민

College of Art & Technology  
Chung-Ang University





git clone

초기화



원격 저장소를 로컬로 복제

```
$ git clone https://github.com/CISLAB-git-id/cislab-web-study.git
```

원격 저장소를 로컬에 복제합니다.

- ✓ 모든 파일, 브랜치, 커밋 히스토리를 가져옴
- ✓ .git 폴더도 함께 복제되어 Git 저장소로 인식
- ✓ 자동으로 'origin'이라는 이름으로 원격 저장소 연결

💡 개념

원격 저장소의 완전한 복사본을 만드는 작업으로, 최초 1회만 실행하면 됩니다.

git checkout

이동



브랜치 간 전환

```
$ git checkout main
```

다른 브랜치로 전환합니다.

- ✓ 작업 디렉토리의 파일들을 해당 브랜치 상태로 변경
- ✓ HEAD 포인터를 해당 브랜치로 이동

💡 개념

HEAD는 현재 작업 중인 브랜치를 가리키는 포인터입니다.

```
git checkout -b
```

생성+이동



새 브랜치 생성 후 이동

```
$ git checkout -b fe/choi-seonmi/week1-react
```

새 브랜치를 생성하고 동시에 전환합니다.

- ✓ git branch + git checkout의 단축 명령
- ✓ 현재 브랜치를 기준으로 새 브랜치 생성

git switch

브랜치 전환



브랜치 전환 (Git 2.23+)

```
$ git switch -c feature/new-feature
```

브랜치를 전환하는 새로운 명령어입니다.

- ✓ checkout보다 명확한 브랜치 전환 전용 명령
- ✓ -c: 새 브랜치 생성 후 전환 (create)
- ✓ -: 이전 브랜치로 전환



Git 2.23부터 도입된 브랜치 전환 전용 명령어로, checkout의 브랜치 기능만 분리한 것입니다.

git restore

파일 복구



파일 변경사항 되돌리기

```
$ git restore --staged file.txt
```

```
$ git restore --source HEAD~1 file.txt
```

작업 디렉토리의 파일을 복구합니다.

- ✓ 작업 디렉토리의 변경사항 취소
- ✓ --staged: 스테이징 취소 (unstage)
- ✓ --source: 특정 커밋에서 파일 복구



개념

checkout의 파일 복구 기능을 분리한 명령어입니다. 실수로 수정한 파일을 원래대로 되돌릴 때 사용합니다.

`git diff`

비교



변경사항 비교

```
$ git diff --staged
```

변경사항을 상세하게 확인합니다.

- ✓ 추가된 줄은 +, 삭제된 줄은 -
- ✓ --staged: 스테이징된 변경사항



git add

스테이징



변경사항을 스테이징 영역으로

```
$ git add .
```

변경사항을 스테이징 영역에 추가합니다.

- ✓ 커밋할 파일들을 준비
- ✓ `..`: 현재 디렉토리의 모든 변경사항



작업 디렉토리 → 스테이징 영역 → 저장소

git status

상태 확인



현재 상태 확인

```
$ git status
```

작업 디렉토리와 스테이징 영역 상태를 확인합니다.

- ✓ 수정된 파일, 스테이징된 파일 표시
- ✓ 현재 브랜치 정보

git commit

저장



스테이징된 변경사항 저장

```
$ git commit -m "feat: 새로운 기능 추가"
```

스테이징된 변경사항을 저장소에 기록합니다.

- ✓ -m: 메시지 직접 입력
- ✓ --amend: 마지막 커밋 수정

git reset

되돌리기



커밋이나 스테이징 되돌리기

```
$ git reset --soft HEAD^
```

커밋이나 스테이징을 되돌립니다.

✓ --soft: 커밋만 취소

⚠ --hard: 모든 변경사항 삭제 (주의!)

git push

업로드



로컬 변경사항을 원격으로

```
$ git push -u origin feature-branch
```

로컬 커밋을 원격 저장소에 업로드합니다.

- ✓ -u: 업스트림 브랜치 설정
- ✓ --delete: 원격 브랜치 삭제

git pull

가져오기+병합



원격 변경사항을 가져와 병합

```
$ git pull origin main
```

원격 저장소의 변경사항을 가져와서 병합합니다.

✓ git fetch + git merge의 단축 명령

⚠ 충돌이 발생할 수 있으므로 주의 필요

`git fetch`

가져오기만



원격 변경사항만 가져오기

```
$ git fetch upstream
```

원격 저장소의 변경사항을 가져오기만 합니다.

- ✓ 로컬 브랜치는 변경하지 않음
- ✓ pull과 달리 안전한 명령

git merge

병합



브랜치 병합

```
$ git merge upstream/main
```

다른 브랜치의 변경사항을 현재 브랜치에 병합합니다.

✓ 두 브랜치의 히스토리를 합침

⚠ 충돌 발생 시 수동 해결 필요



`git stash`

임시 저장

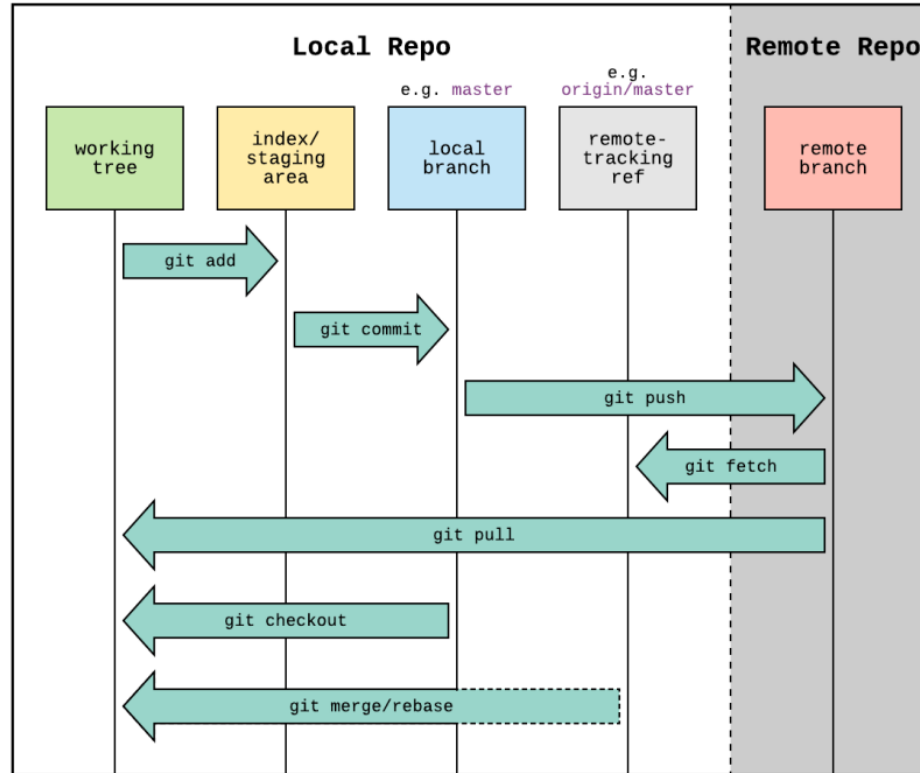


작업 내용 임시 보관

```
$ git stash pop
```

작업 중인 변경사항을 임시로 보관합니다.

- ✓ 브랜치 전환이 필요할 때 유용
- ✓ 스택(Stack) 구조로 저장





특정 파일만



모든 파일



실전 예시



특정 파일만 스테이징 취소

```
$ git restore --staged file.txt # Git 2.23+ 권장  
$ git reset HEAD file.txt # 기존 방법
```



특정 파일만



모든 파일




실전 예시



모든 파일 스테이징 취소

```
$ git restore --staged . # Git 2.23+ 권장  
$ git reset HEAD # 기존 방법
```

 특정 파일만

 모든 파일

 실전 예시

## 실전 시나리오


```
$ git status
Changes to be committed:
  modified:   src/App.js
  modified:   .env # 이걸 올리면 안됨!


$ git restore --staged .env

$ git status
Changes to be committed:
  modified:   src/App.js

Changes not staged for commit:
  modified:   .env
```

 메시지 수정

 커밋 취소

 Push 후 되돌리기

✓ 커밋 메시지만 수정

```
$ git commit --amend -m "새로운 커밋 메시지"
```

 메시지 수정

 커밋 취소

 Push 후 되돌리기

## 1 작업 내용 유지하며 커밋만 취소

```
$ git reset --soft HEAD~1
```

## 2 스테이징도 취소 (기본값)

```
$ git reset HEAD~1
```

## 3 완전히 없던 일로 (위험!)

⚠ 모든 변경사항이 사라집니다!

```
$ git reset --hard HEAD~1
```



1 충돌 확인

2 충돌 해결

3 예방 방법

```
$ git pull origin main
Auto-merging src/components/Header.js
CONFLICT (content): Merge conflict in src/components/Header.js
Automatic merge failed; fix conflicts and then commit the result.
```

## 충돌 파일 내용

```
function Header() {
  return (
    <header>
<<<<<<< HEAD
<h1>My Website</h1>
=====
<h1>우리 웹사이트</h1>
>>>>>> origin/main
</header>
);
}
```

1 충돌 확인

2 충돌 해결

3 예방 방법

## 1 수동으로 편집

충돌 마커를 제거하고 원하는 코드만 남기기

```
function Header() {  
  return (  
    <header>  
    <h1>우리 웹사이트</h1>  
    </header>  
  );  
}
```

## 2 한쪽 버전만 선택

```
$ git checkout --ours src/components/Header.js # 내 버전 선택  
$ git checkout --theirs src/components/Header.js # 원격 버전 선택
```

## 3 충돌 해결 완료

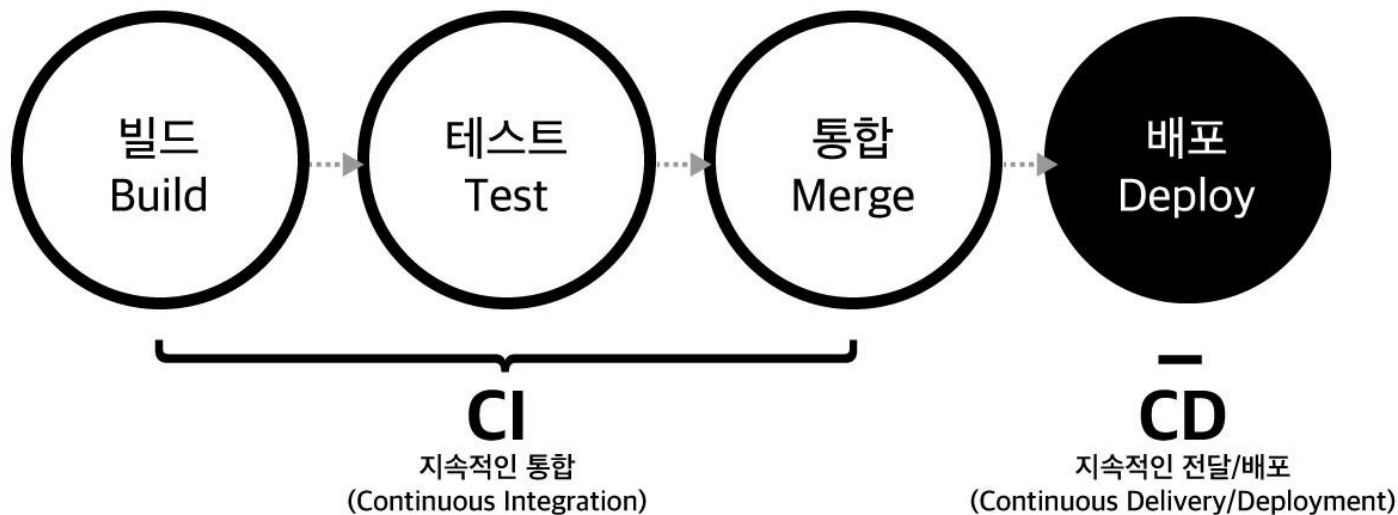
```
$ git add src/components/Header.js  
$ git commit -m "Merge: 충돌 해결"
```

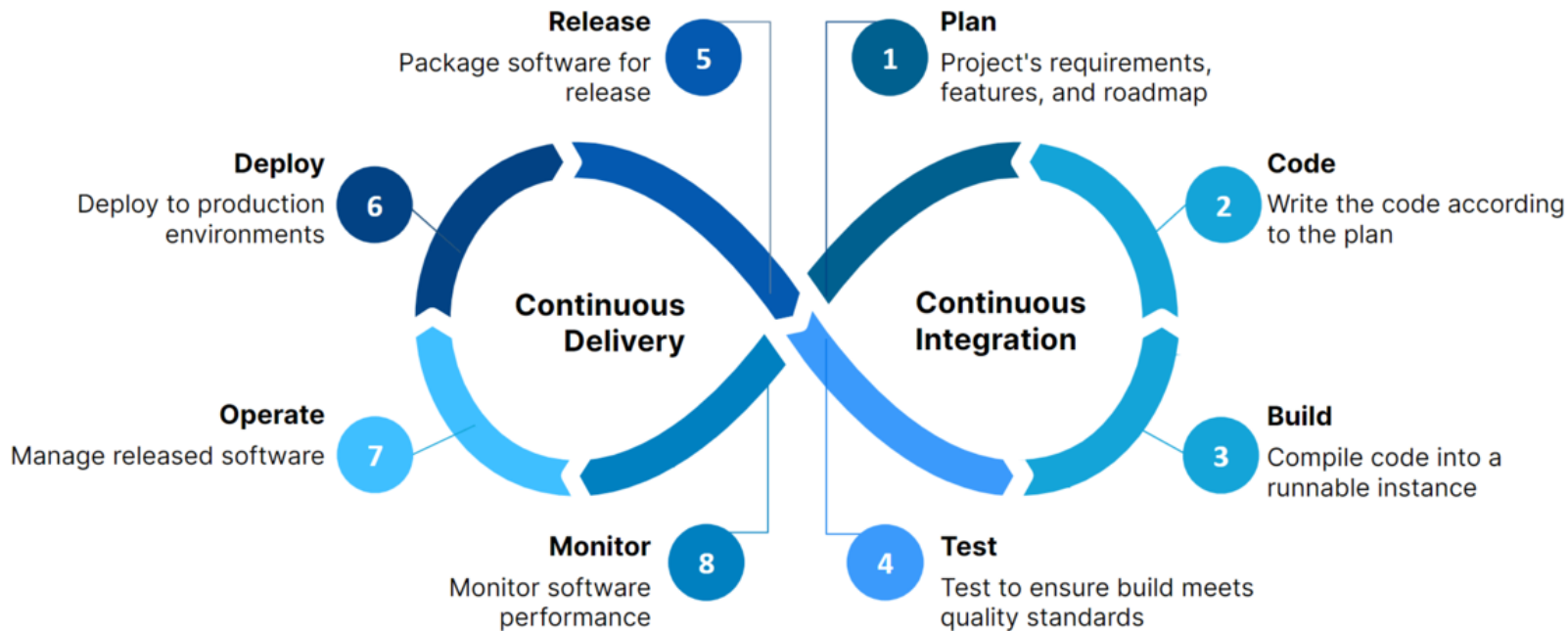
## Stash 활용

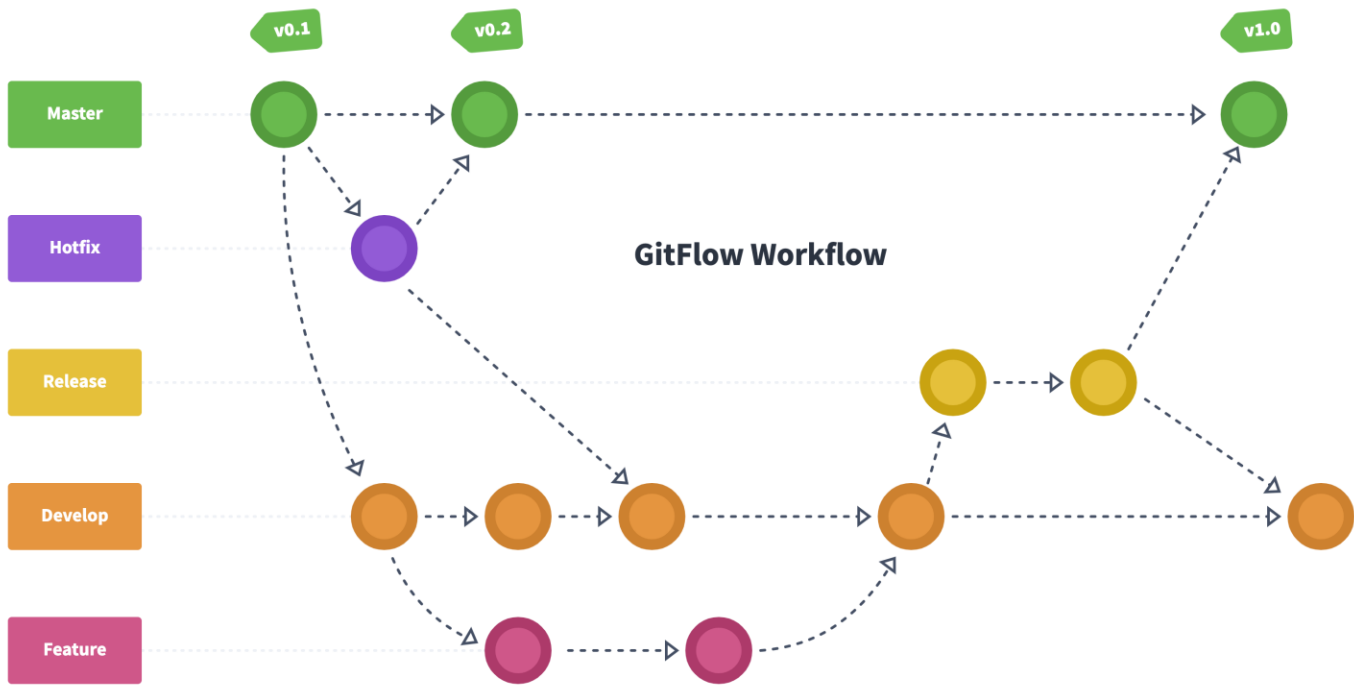
```
$ git stash # 작업 내용 임시 저장  
$ git pull origin main  
$ git stash pop # 저장한 작업 다시 적용
```

## Fetch 후 확인

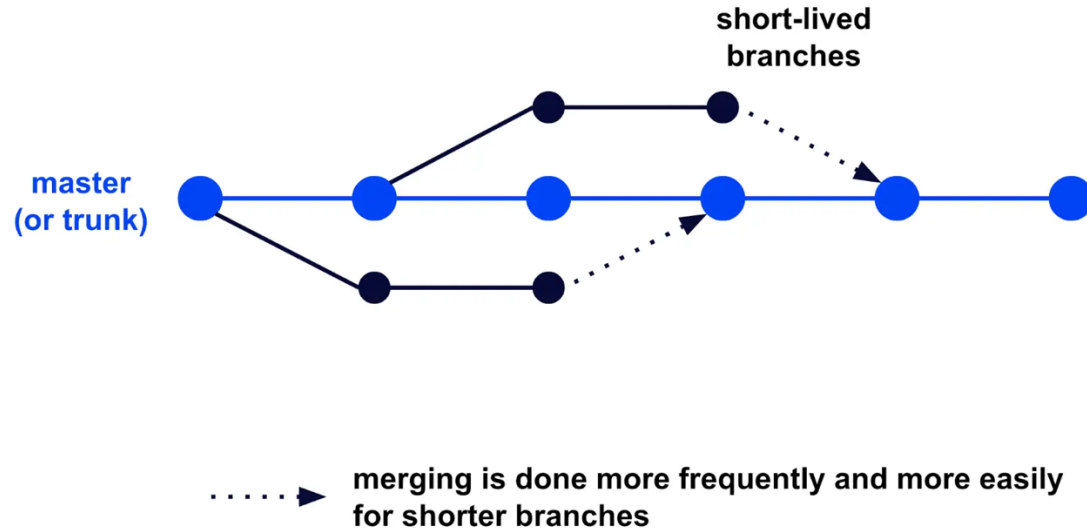
```
$ git fetch origin  
$ git diff HEAD origin/main # 차이점 미리 확인  
$ git merge origin/main
```







## Trunk-based development





구분	Git Flow	Trunk-based Development
브랜치 구조	복잡한 다중 브랜치 구조 master (운영) develop (통합) feature (기능) release (릴리스) hotfix (긴급수정)	단순한 구조 main/trunk (메인) 짧은 수 명의 feature (기능) release 브랜치 (선택적)
메인 브랜치 수	2개 (master + develop)	1개 (main/trunk)
기능 개발 방식	장기간 feature 브랜치 유지	단기간 feature 브랜치
병합 빈도	낮음 (기능 완성 후)	매우 높음 (하루 여러 번)
코드 리뷰	PR/MR 통한 전체 기능 리뷰	작은 단위의 빈번한 리뷰
배포 주기	주/월 단위의 정기 배포	일/시간 단위의 수시 배포
충돌 발생 가능성	높음 (장기 브랜치)	낮음 (빈번한 통합)
롤백 방식	브랜치 기반 롤백	커밋 기반 롤백
CI/CD 적합성	제한적	매우 높음

구분	Git Flow	Trunk-based Development
Feature Toggle	거의 사용 안 함	필수적으로 사용
테스트 전략	브랜치별 테스트	메인 브랜치 집중 테스트
품질 관리	브랜치 격리로 품질 보장	자동화된 테스트로 품질 보장
버전 관리	명시적 버전 태깅	커밋 해시 기반
핫픽스 처리	별도 hotfix 브랜치	메인 브랜치에 직접 수정