Git&Github

1장 실습 환경 구축

Git 명령어 종류

usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]

[--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]

[-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]

[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]

[--config-env=<name>=<envvar>] <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)

clone Clone a repository into a new directory

init Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)

add Add file contents to the index

mv Move or rename a file, a directory, or a symlink

restore Restore working tree files

rm Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)

bisect Use binary search to find the commit that introduced a bug

diff Show changes between commits, commit and working tree, etc

grep Print lines matching a pattern

log Show commit logs

show Show various types of objects

status Show the working tree status

grow, mark and tweak your common history

branch List, create, or delete branches

commit Record changes to the repository

merge Join two or more development histories together

rebase Reapply commits on top of another base tip

reset Reset current HEAD to the specified state

switch Switch branches

tag Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)

fetch Download objects and refs from another repository

pull Fetch from and integrate with another repository or a local branch

push Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some

concept guides. See 'git help <command>' or 'git help <concept>'

to read about a specific subcommand or concept.

See 'git help git' for an overview of the system.

Sourcetreeapp

Python

Pip 명령어 종류

Usage:

pip <command> [options]

Commands:

install	Install packages.
download	Download packages.
uninstall	Uninstall packages.
freeze	Output installed packages in requirements format.
inspect	Inspect the python environment.
list	List installed packages.
show	Show information about installed packages.
check	Verify installed packages have compatible dependencies.
config	Manage local and global configuration.
search	Search PyPI for packages.
cache	Inspect and manage pip's wheel cache.
index	Inspect information available from package indexes.
wheel	Build wheels from your requirements.
hash	Compute hashes of package archives.
completion	A helper command used for command completion.
debug	Show information useful for debugging.
help	Show help for commands.

General Options:

-h, --help	Show help.
--debug	Let unhandled exceptions propagate outside the main subroutine,

instead of logging them

to stderr.

--isolated Run pip in an isolated mode, ignoring environment variables and user configuration.

--require-virtualenv Allow pip to only run in a virtual environment; exit with an error otherwise.

--python <python> Run pip with the specified Python interpreter.

-v, --verbose Give more output. Option is additive, and can be used up to 3 times.

-V, --version Show version and exit.

-q, --quiet Give less output. Option is additive, and can be used up to 3 times (corresponding to

WARNING, ERROR, and CRITICAL logging levels).

--log <path> Path to a verbose appending log.

--no-input Disable prompting for input.

--keyring-provider <keyring_provider>

Enable the credential lookup via the keyring library if user input is allowed. Specify

which mechanism to use [disabled, import, subprocess]. (default: disabled)

--proxy <proxy> Specify a proxy in the form scheme://[user:passwd@]proxy.server:port.

--retries <retries> Maximum number of retries each connection should attempt (default 5 times).

--timeout <sec> Set the socket timeout (default 15 seconds).

--exists-action <action> Default action when a path already exists: (s)witch, (i)gnore, (w)ipe, (b)ackup,

(a)bort.

--trusted-host <hostname> Mark this host or host:port pair as trusted, even though it does

not have valid or any

HTTPS.

--cert <path> Path to PEM-encoded CA certificate bundle. If provided, overrides the default. See 'SSL

Certificate Verification' in pip documentation for more information.

--client-cert <path> Path to SSL client certificate, a single file containing the private key and the

certificate in PEM format.

--cache-dir <dir> Store the cache data in <dir>.

--no-cache-dir Disable the cache.

--disable-pip-version-check

Don't periodically check PyPI to determine whether a new version of pip is available for

download. Implied with --no-index.

--no-color Suppress colored output.

--no-python-version-warning

Silence deprecation warnings for upcoming unsupported Pythons.

--use-feature <feature> Enable new functionality, that may be backward incompatible.

--use-deprecated <feature> Enable deprecated functionality, that will be removed in the future.

PyQt5

VSCODE

GITHUB 가입하기

2장 소스트리로 GIT 체험하기

1. 원격 저장소 생성->GITHUB
2. 로컬 저장소 생성->sourceTree
    A. Clone : 원격저장소의 내용을 복제해서 생성하기
    B. Create : 개인 PC에만 로컬저장소 생성하기
    C. Add : 개인 PC에 존재하는 로컬저장소를 소스트리 인터페이스에 추가하기

    이력 저장 순서 : 기존 파일 수정 또는 새 파일 추가->스테이지에 올리기->메시지를 입력하고 커밋
3. 로컬저장소와 원격저장소의 내용 일치시키기
4. 원격저장소의 내용을 로컬저장소로 다운로드 : Pull

3장 Git의 동작 개념

Working Directory : 이력 관리 대상 파일들이 위치하는 영역/저장소 디렉터리에서 .git 폴더를 제외한 공간/작업 중인 파일이나 코드가 저장되는 공간

Staging Area : 이력을 기록할, 즉 커밋을 진행할 대상 파일들이 위치하는 영역/.git 폴더 하위에 파일형태로 존재(index)

Repository : 이력이 기록된 파일들이 위치하는 영역/.git 폴더에 이력 관리를 위한 모든 정보가 저장, 관리됨

Working Directory->

git add->Stating area(Staging)

->git commit->Repository(Commit)

Git이 관리하는 3가지 파일 상태

Modified : 관리 중인 파일에 수정 사항이 감지되었지만 커밋이 되지 않은 상태

Staged : 감지된 파일의 수정 사항이 Staging area로 이동한 상태

Committed : 파일의 수정 사항에 대한 이력 저장이 완료된 상태

Tracked vs untracked : Tracked -> Git 저장소의 관리 대상인 파일들

4장 Git 기본 명령어

저장소 생성하기

빈 저장소 생성하기

사용자 정보 설정하기

Add와 commit : 개발 이력 기록하기

프로그램 작성하기

첫 번째 이력 저장하기

두 번째 이력 저장하기

세 번째 이력 저장하기

Status, log, 그리고 show : 저장소의 상태와 커밋 내역 확인하기

Git status

Git log

Git show

Diff : 파일의 수정 내용 비교하기

프로그램 수정하기

Git diff로 파일의 수정된 내용 확인하기

Git diff로 커밋 간의 내용 비교하기

Reset1 : 스테이징 되돌리기

파일 추가하고 스테이징하기

Git reset으로 파일 언스테이징하기

Amend : 최근에 작성한 커밋 수정하기

현재 상태 커밋하기

Git commit –amend로 최근 커밋 수정하기

Checkout : 커밋 되돌리기

Git checkout는 HEAD의 참조를 변경하는 명령이다. 최신 커밋을 참조할 때 HEAD는 master를 간접 참고하고 있다. HEAD가 특정 커밋을 참조하기 위해 체크아웃되면 HEAD는 master 참조에서 떨어진 Deatached 상태 또는 Detached HEAD가 됨.

현재 저장소의 상태 확인하기

Git checkout으로 커밋 되돌리기

최신 커밋으로 돌아가기

*git checkout으로 modified 파일 되돌리기

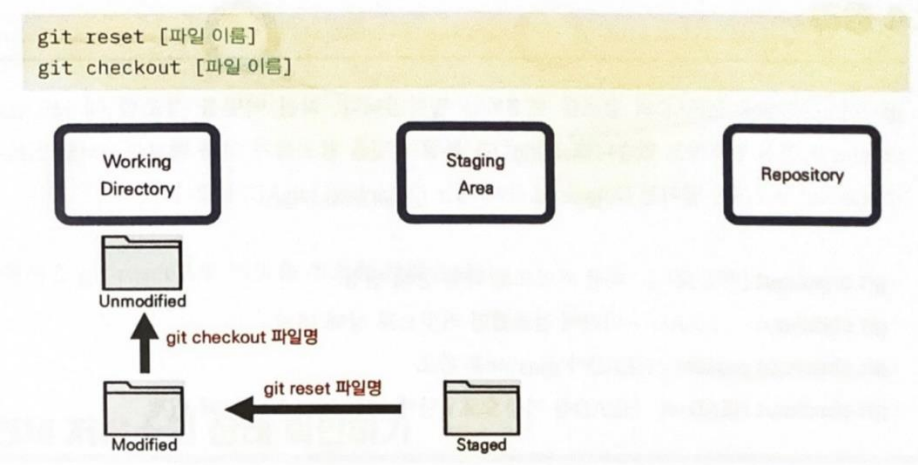


그림 4.50 git checkout으로 Modified 파일 되돌리기

Reset2 : 커밋 취소하기

	Working Directory	Stating area	Repository
--hard	변경	변경	변경
--mixed (default)	유지	변경	변경
--soft	유지	유지	변경

***** Git checkout은 HEAD의 참조를 변경하는 반면, git reset은 master의 참조를 변경함.

현재 저장소의 상태 확인하기

Git reset으로 커밋 취소하기

리셋으로 커밋이 삭제된 것일까?

--soft 옵션으로 git reset하기

Git reset은 브랜치의 참조를 변경하는 명령어이다. Git 저장소는 브랜치 이름으로 참조하는 커밋을 최신 커밋으로 인식함. 이러한 특징을 인해 git reset으로 최신 커밋을 재설정할 수 있음.

Reflog : HEAD의 참조 이력 확인하기

Reflog=reference(참조)+log(로그)의 합성어로 HEAD의 참조 이력을 로그 형태로 출력함

name@{quantifier}-> HEAD@{2} : HEAD가 2번 움직임 전에 참조한 해시가 저장됨. 특정 커밋을 참조하고 있기 때문에, 해시 대신 사용할 수 있음.

HEAD와 master

브랜치(branch)

: 저장소(repository) 내에 존재하는 독립적인 작업 관리 영역

master branch : 저장소를 git init으로 생성하면 자동으로 생기는 브랜치

master : 해당 브랜치의 끝(최신 커밋)을 참조하는 개체(Refs)

HEAD의 주요 특징

Working Directory의 내용은 HEAD가 참조하는 커밋의 내용

Master, origin 등과 달리 변경 불가능한 공식 명칭

커밋을 직접 참조할 수 있을 뿐만 아니라 Refs(다른 참조 개체)도 참조 가능

HEAD와 브랜치의 관계

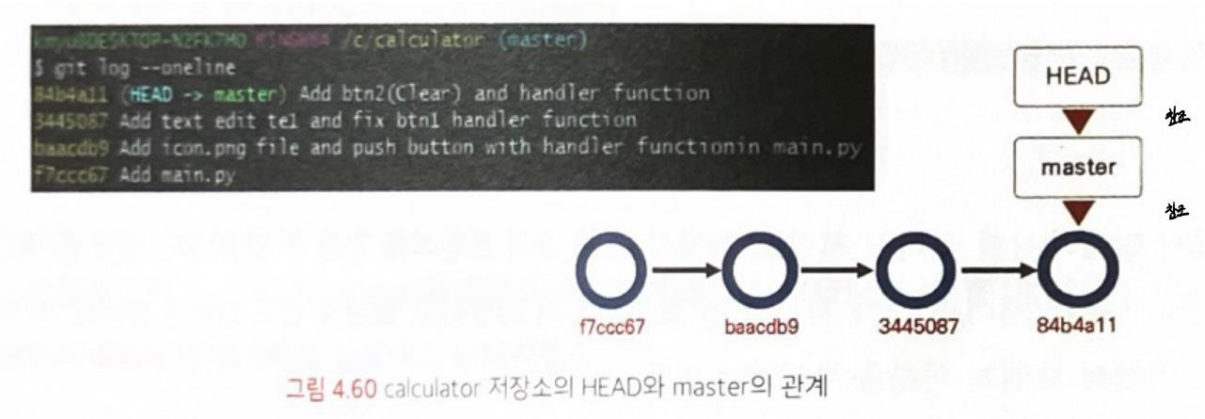


그림 4.60 calculator 저장소의 HEAD와 master의 관계

Checkout, reset 명령에 따른 HEAD 이동

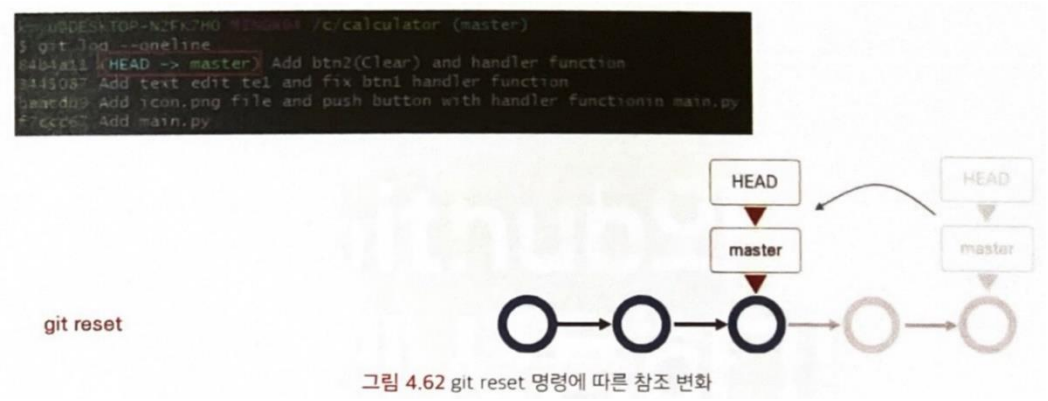


그림 4.62 git reset 명령에 따른 참조 변화

Git checkout = Detached HEAD

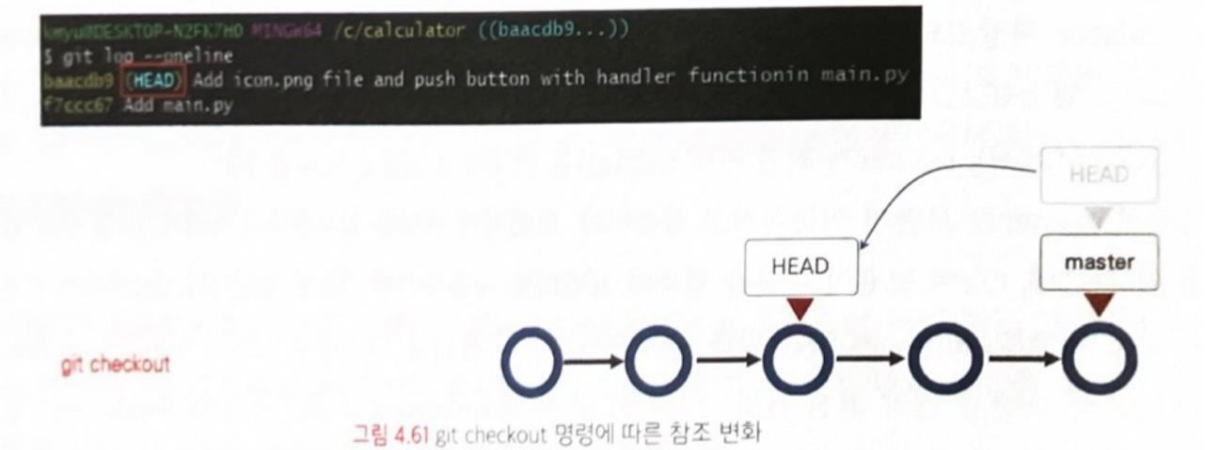


그림 4.61 git checkout 명령에 따른 참조 변화

5장 github와 함께 사용하기

안전성 : 로컬저장소가 손상되었을 때 백업 저장소 역할

협업 : 원격저장소를 중심으로 다수의 개발자가 파일 시스템을 공유하고 협업 가능

원격저장소 생성과 연동

원격저장소 생성하기

원격저장소 등록하기

다운받는 경로 : fetch, 업로드하는 경로(push)

*origin : 원격저장소의 기본이름으로 자주 사용되는 이름, 로컬저장소의 master브랜치와 마찬가지로 일반적으로 사용되는 이름

업스트림 설정하기

*업스트림 설정후 git push 사용법 : 한번 업스트림 설정을 하면 각 저장소 브랜치를 계속 추적합니다. 따라서, 다음부터 파일로 업로드를 할 때는 git push origin master 대신 git push만 입력해도 됨.

Push와pull1:저장소로 업로드, 저장소에서 다운로드하기

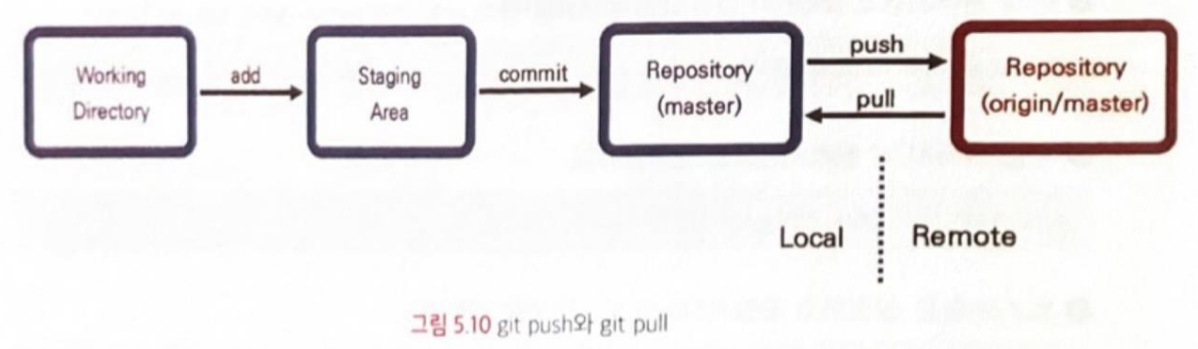


그림 5.10 git push와 git pull

로컬저장소의 파일 수정하기

수정 내용 커밋하기

Git push로 로컬저장소의 내용 업로드하기

원격저장소의 파일 수정하기

Git pull로 원격저장소의 내용 가져오기

Tag : 부가정보 추가하기

git에서는 태그 이름을 참조개체로 사용할 수 있음.

git에서 관리하는 태그: Lightweight 태그(태그이름만 기록), Annotated 태그(태그 이름과 함께 설명, 서명, 작성자 정보, 날짜 등의 정보 포함)

저장소 확인하기

Lightweight 태그 작성하기

Annotated 태그 작성하기

태그 확인하기

태그로 체크아웃하기

태그 삭제하기

원격저장소에 태그 푸시하기

Revert:푸시한 커밋 되돌리기

그림 5.31 git revert 명령 개념

로컬저장소의 파일 수정하기

수정한 내용 커밋, 푸시하기

Git revert로 커밋 되돌리기

*git revert [커밋 해시1]..[커밋 해시2] : 한번의 명령으로 여러 구간을 되돌리면 git revert 뒤에 구간을 지정해주면 된다. 이때 커밋 해시1은 되돌리기 대상에 포함되지 않음.

*git revert -n [커밋 해시1]/git revert –no-commit [커밋 해시1] : 커밋하지 않은 채 파일 내용만 되돌리고 싶을 경우

되돌린 내용 푸시하기

*git reset과 git revert의 차이점

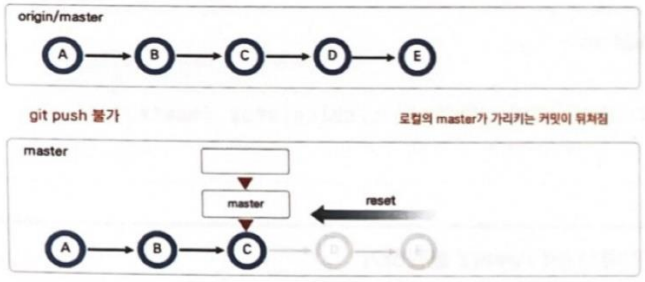


그림 5.41 git reset 이후 저장소 상태

따라서 push까지 완료한 상태에서 작업 내용을 되돌리려면 git revert를 사용해야 합니다. 물론 push까지 마치지 않은 상태라면 git reset을 사용해서 해결 가능합니다.

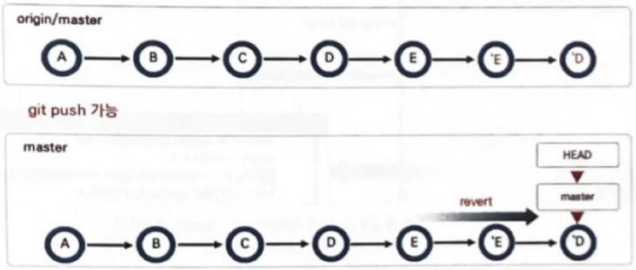


그림 5.42 git revert 이후 저장소 상태

CH06 Github로 협업하기

Clone : 원격저장소 복제하기

기존 로컬저장소 이름 수정하기

Git clone으로 원격저장소 내용 복제하기

calculator_B 저장소 사용자 설정하기

*서로 다른 계정으로 원격저장소를 공유하는 방법

Push와 pull 협업 환경에서 작업 내용 업데이트하기

개발자 A : ui.py 수정, 커밋, 푸시하기

개발자 B : ctrl.py 수정, 커밋, 푸시하기

개발자 A : 원격저장소의 내용을 가져와서 병합하기

충돌해결하기

Git pull = git fetch + git merge

개발자 B : ui.py 수정, 커밋, 푸시하기

개발자 A : ui.py 수정, 커밋, 푸시하기

개발자 A : git pull 그리고 충돌 해결하기

개발자 B : git pull로 원격저장소의 내용 병합하기

Fetch와 merge

개발자 B : ctrl.py 수정, 커밋, 푸시하기

개발자 A: git fetch로 원격저장소의 정보 가져오기

개발자 A : git merge로 원격저장소의 내용 병합하기

개발자 A : ctrl.py 파일의 sum 함수 수정 후 커밋, 푸시

*git pull과 git fetch&merge 중 어떤 명령을 써야 할까?->상황에 따라 결정

원격저장소 브랜치의 작업내용을 검토할 필요가 있을 경우->git pull(시간 효율이 높음)

원격저장소의 작업내용을 사전에 면밀히 검토할 필요가 있을 경우->git fetch(안정성 측면에서 유리함)

Blame : 코드의 수정 내역 확인하기

Git log : 저장소에 기록된 커밋 히스토리를 확인하는 명령

Git show : 특정 커밋의 상세정보를 확인하는데 도움이 되는 명령

Git diff : 커밋, 또는 브랜치 간의 파일의 변경 사항을 확인하는데 유용하게 사용되는 명령

Git blame: 특정파일의 수정내역을 라인 단위로 설명

Git blame으로 소스 코드 수정내역 확인하기

Stash : 작업 내용 임시 저장하기

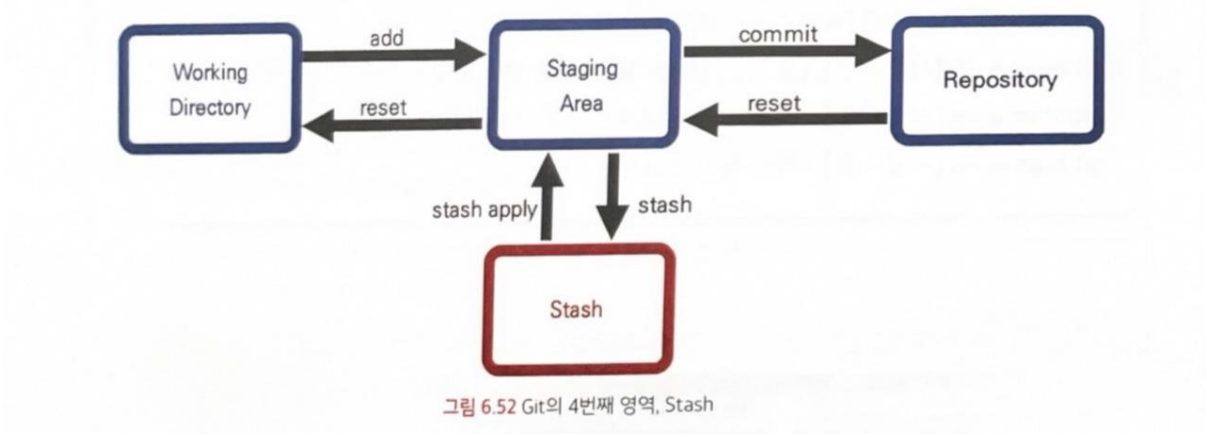


그림 6.52 Git의 4번째 영역, Stash

Git이 관리하는 3가지 영역(Working Directory, Staging Area, Repository)+Stash((임시 저장 공간)

Stash : (무언가를) 안전하게 저장한다

개발자 A : ui.py 수정, 커밋, 푸시하기

개발자 B : ctrl.py 수정하기

개발자 B : git stash로 작업 내용 임시 저장하기

Git stash 살펴보기

개발자 B : 원격저장소의 내용 가져와서 확인하기

개발자 B : 임시 저장한 내용 가져오기

Stash 저장 공간은 스택 구조를 따른다.

개발자 B : 작업 마무리하고 커밋, 푸시하기

개발자 B : 저장 내용 삭제하기

*한꺼번에 저장 내용을 가져오고, stash 영역에서 삭제하기

Git stash apply와 git stash drop을 한꺼번에 실행하는 명령->Git stash pop [인덱스]->다른 stash 명령과 마찬가지로 stash 인덱스를 지정하지 않으면 최근에 저장된 내용이 내보내지고 stash 영역에서 삭제됨.

개발자 A : 원격저장소의 내용을 저장하기

*stash 영역을 활용할 수 있는 또 다른 예

현업에서는 한 저장소 안에 여러 개의 브랜치를 구성하여 작업을 진행하는 경우가 있는데, 그러다 보면 엉뚱한 브랜치에서 작업을 하는 상황도 종종 발생합니다. 이때 당황하지 않고 stash 영역을 활용하면 쉽게 활용할 수 있습니다. 이미 커밋을 해버렸을 경우에는 작업내용을 소프트 리셋하고, 커밋을 하지 않았다면 바로 작업한 내용을 stash영역으로 이동시키면 됩니다. 저장소 안의

모든 브랜치는 공통의 stash 영역을 사용하기 때문에 원래 작업하려고 했던 브랜치로 전환해서 작업 내용을 가져올 수 있습니다.

Ch07 브랜치

Git branch와 checkout : 브랜치 생성과 전환

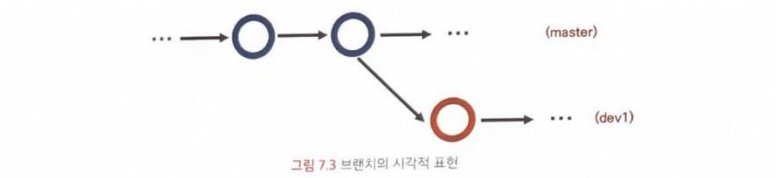


그림 7.3 브랜치의 시각적 표현

브랜치는 특정 지점의 커밋에서 분기해서 커밋을 이어 나가는 모습이 마치 나무의 가지가 뻗어 나가는 것과 비슷하다고 하여 붙여진 이름입니다. 이번 절에서는 새롭게 만든 브랜치에서 각종 연산 함수를 작성하고 커밋을 기록하는 실습을 해 봅니다.

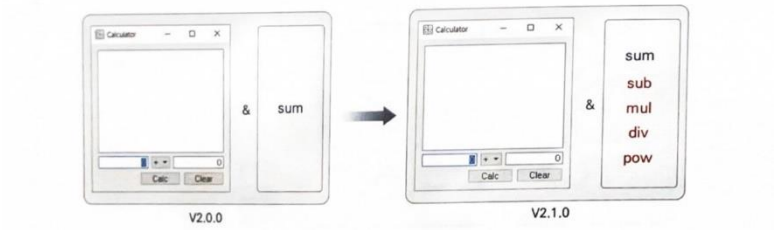


그림 7.4 새 브랜치에서 할 작업

Git branch로 브랜치 생성, 삭제하기

Git checkout으로 브랜치 전환하기

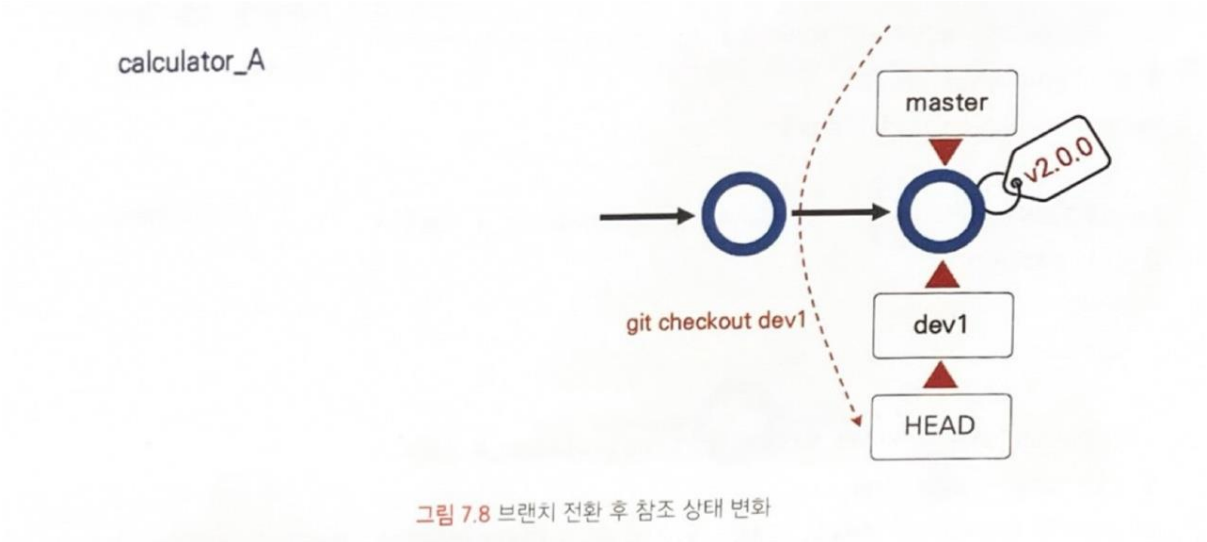


그림 7.8 브랜치 전환 후 참조 상태 변화

새 브랜치에서 작업하기

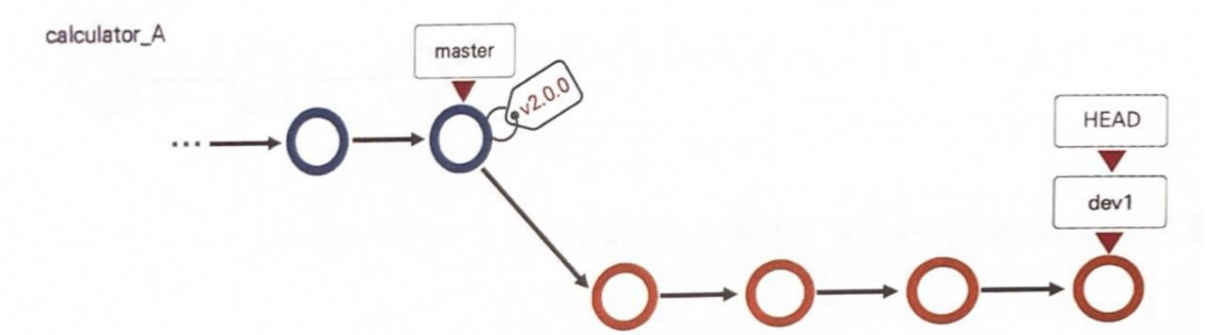


그림 7.14 네 번째 커밋 후 브랜치 참조 변화

Merge : 브랜치 병합하기

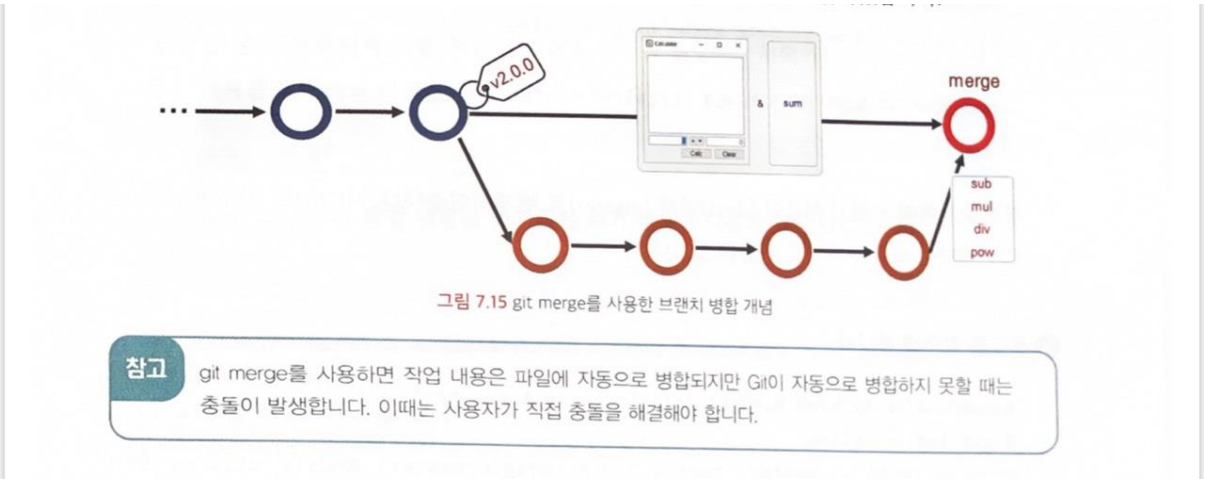


그림 7.15 git merge를 사용한 브랜치 병합 개념

참고 git merge를 사용하면 작업 내용은 파일에 자동으로 병합되지만 Git이 자동으로 병합하지 못할 때는 충돌이 발생합니다. 이때는 사용자가 직접 충돌을 해결해야 합니다.

작업 브랜치를 master로 전환하기

Master 브랜치에 dev1 브랜치의 작업 내용 병합하기

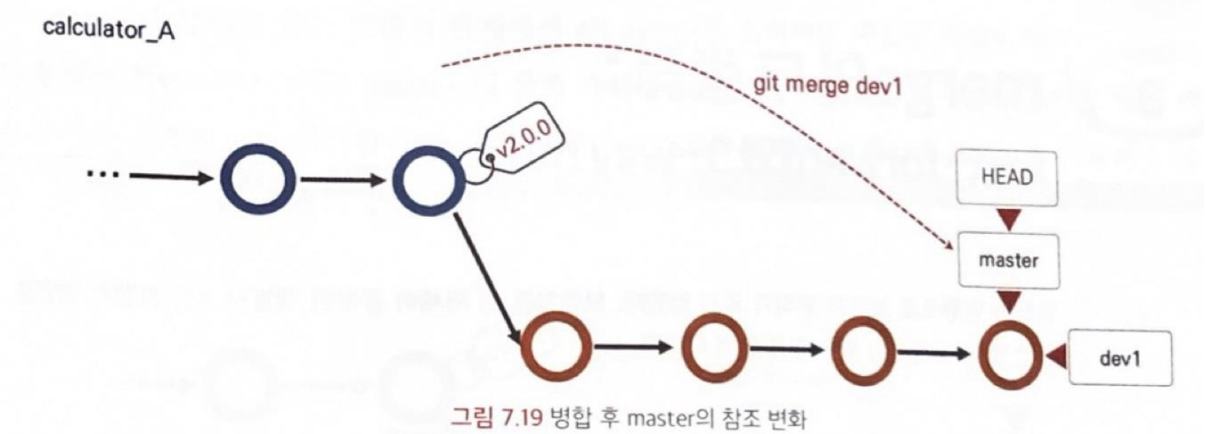


그림 7.19 병합 후 master의 참조 변화

Merge의 두 종류 : fast-fastforward, 3-way merge

Fast-forward merge : 현재 브랜치가 base 커밋에 위치해서 단순히 참조 커밋을 이동시켜 병합

Base : master 브랜치에서 dev1브랜치가 분기해 나가는 시점, 즉 두 브랜치가 공통적으로 가지고 있는 커밋

Fast-forward 상태에 있는 브랜치 관계에서 git merge를 입력할 경우에는 새로운 커밋이 생기지 않음. 마치 브랜치가 점프하듯 상대 브랜치의 커밋으로 이동하는 모습을 본 떠 fast-forward(빨리 감기)라는 이름이 붙었음.

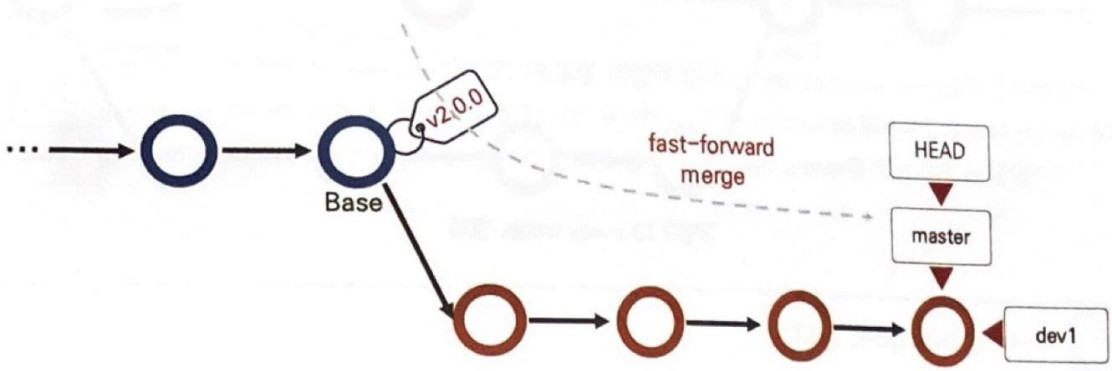


그림 7.21 fast-forward 관계에서 병합 결과

3-way merge : base를 기준으로 두 브랜치의 변경 사항이 생겨 새로운 커밋이 생기면서 병합

두 브랜치의 참조개체 중 어느 것도 base를 참조하지 않음.

병합과정에서 세 개의 커밋을 참고함.(base 커밋, 두 브랜치의 참조 커밋)

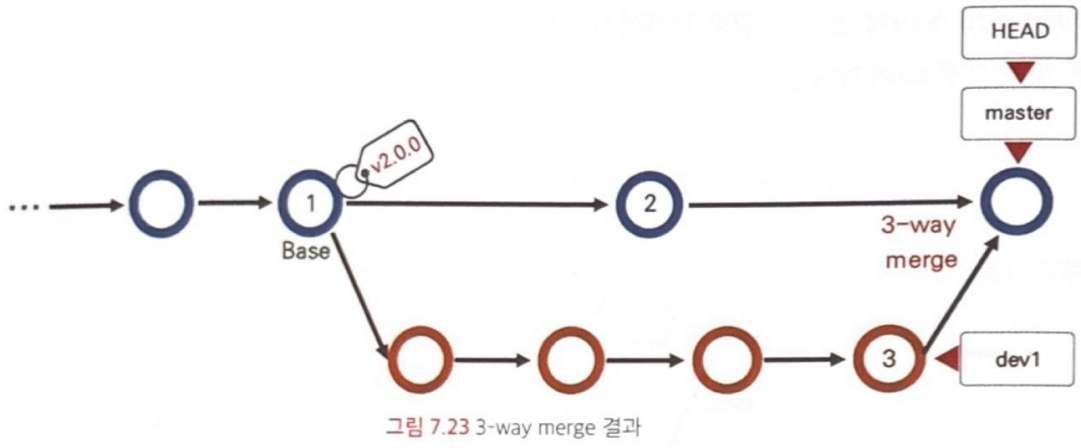


그림 7.23 3-way merge 결과

->충돌이 발생할 경우 직접 파일을 수정하여 해결해야 함.

Merge 옵션 : --ff, --no-ff, --squash

Fast-forward merge

➔ --ff옵션은 기본값으로, git merge[브랜치 이름]형식으로 입력하면 --ff옵션으로 동작함.

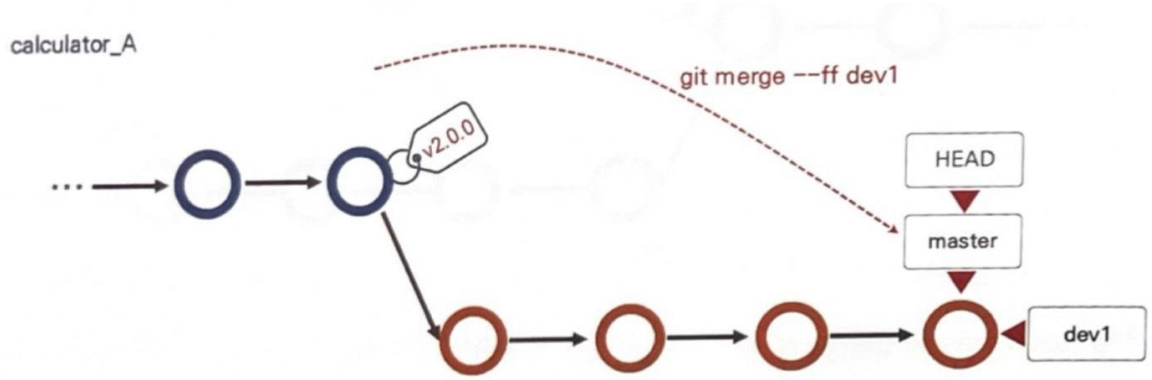


그림 7.28 fast-forward merge 결과

Non fast-forward merge

➔ 브랜치 관계에 상관없이 필요한 커밋만 가져올 수 있음.

➔ 어떤 브랜치에서 병합을 했는지 기록을 남길 수 있음.

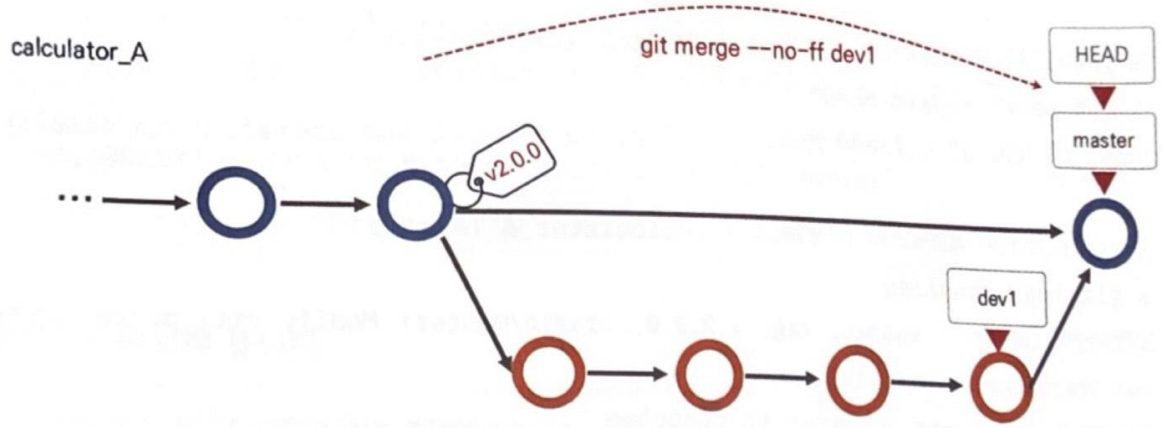


그림 7.31 non fast-forward merge 결과

Squash merge

➔ --squash 내용으로 병합되면 브랜치의 내용을 가져오기만 하고 커밋을 생성하지 않음. 상대 브랜치의 작업 내용이 추가되어 파일 상태가 변경되고, 변경된 파일은 스테이징 단계까지 이동하게 됨.

➔ --squash 옵션을 사용하여 병합하면 커밋이 어떤 브랜치에서 왔는지 확인할 수 없음.

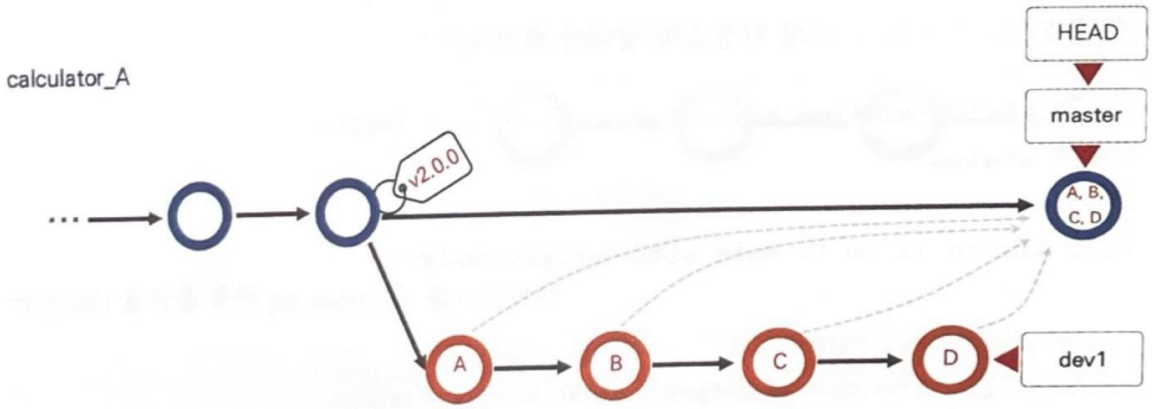


그림 7.33 --squash 옵션으로 병합, 커밋 결과

브랜치 정리하기

*--ff, --no-ff, --squash 중에 무엇을 써야 하나?

배포용 버전을 기록할 브랜치에서 다른 브랜치의 내용을 병합할 때 --ff보다는 –no-ff를 사용할 것을 권장함. –no-ff와 --squash중에서는 병합이력까지 상세하게 기록하고 싶으면 전자를, 브랜치를 깔끔하게 정리하고 싶으면 후자가 효과적임.

Rebase : 브랜치 재배치하기

저장소 안에서 생성한 각 브랜치에서 작업을 완료한 후 master 브랜치에 병합해야 하는 상황입니다. 첫 번째 방안은 4절에서 소개한 git merge를 사용해서 브랜치끼리 병합을 이어나가는 전략입니다.

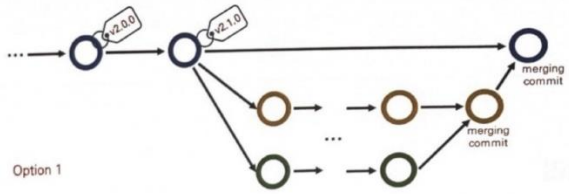


그림 7.37 첫 번째 방안 : 브랜치끼리 병합 이어나가기

두 번째는 git rebase 명령으로 브랜치의 base를 조정하여 정리하는 방법입니다. 브랜치의 수가 많아질 경우 깔끔하게 정리할 수 있습니다.

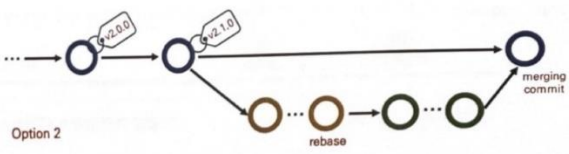


그림 7.38 두 번째 방안 : 브랜치의 base 조정하기(rebase)

Issue1 브랜치 작업하기

Issue2 브랜치 작업하기

Git rebase로 브랜치 재배치하기

*rebase 과정에서 충돌 해결하기

충돌 발생

Rebase를 취소하고 싶을 경우 : git rebase –abort

충돌을 직접 해결->코드를 정리한 후 후속 처리-> git add . -> git rebase –continue

Master 브랜치에서 병합하기

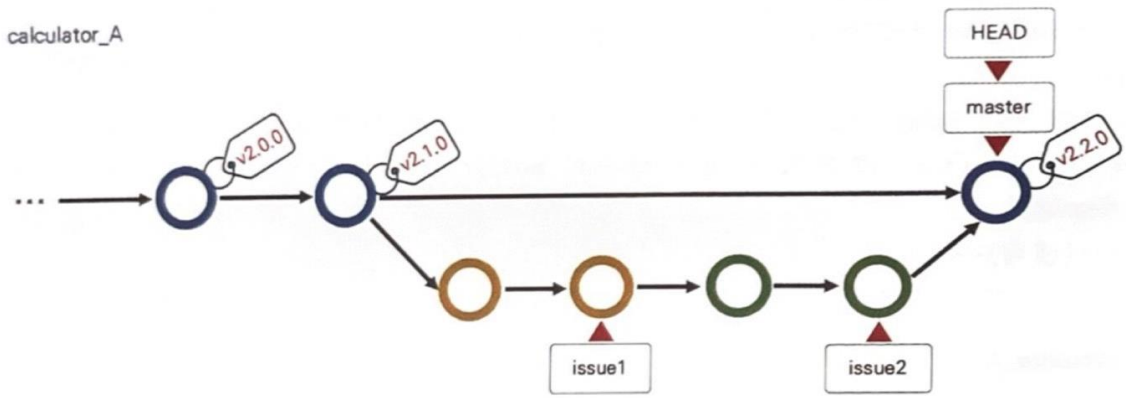


그림 7.53 태그 추가

*git merge와 rebase 중 어느 것을 사용해서 브랜치를 관리하는 것이 좋은가?

기록을 상세하게 남기고 싶을 경우->git merge

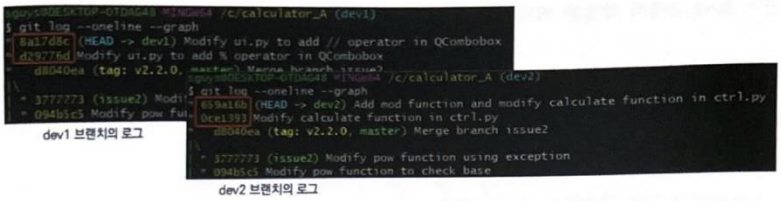
기록을 간결하게 정리하고 싶을 경우->git rebase

Cherry-pick : 다른 브랜치의 커밋 적용하기

Dev1 브랜치 작업하기

Dev2 브랜치 작업하기

Dev2 브랜치 : git cherry-pick으로 특정 커밋의 내용 가져오기



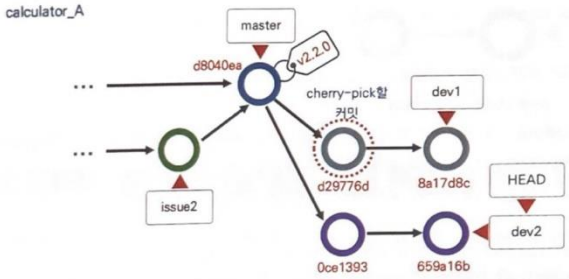


그림 7.68 각 브랜치의 커밋 확인

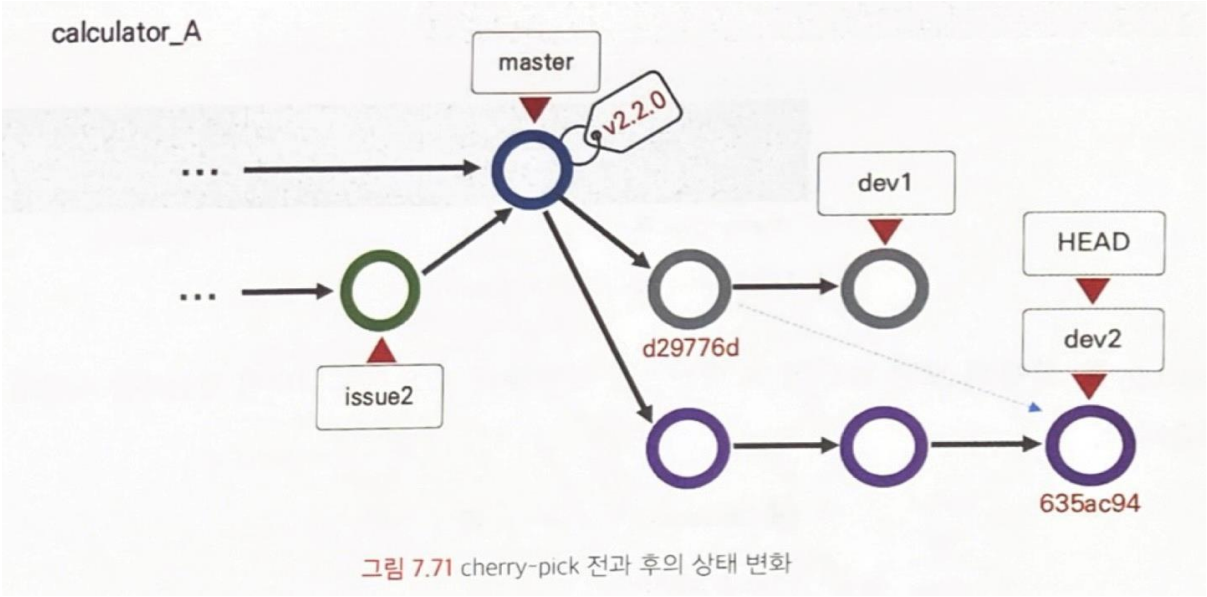


그림 7.71 cherry-pick 전과 후의 상태 변화

*cherry-pick 과정에서 충돌 해결하기

Git rebase와 마찬가지로 git cherry-pick을 사용하는 과정에서도 충돌이 발생할 수 있습니다.

충돌 해결과정은 git rebase와 비슷함->git cherry –abort -> git add . -> git cherry-pick --continue

Master 브랜치에서 dev2의 내용 병합하기

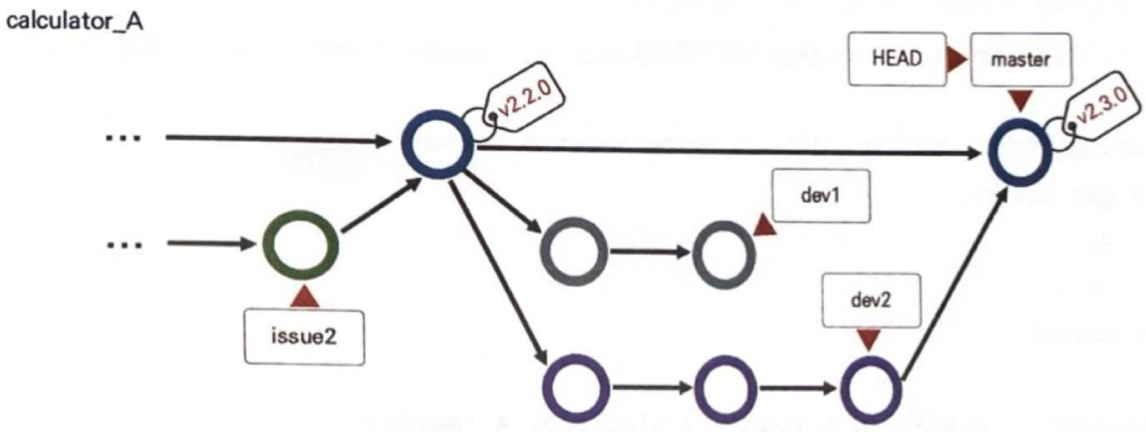


그림 7.75 태그 추가 후 저장소 상태 변화

사용하지 않는 브랜치 삭제하기

신규 브랜치 푸시하기

Master 브랜치의 내용 푸시하기

Light 브랜치 작업하기

Light 브랜치의 작업 내용 푸시하기

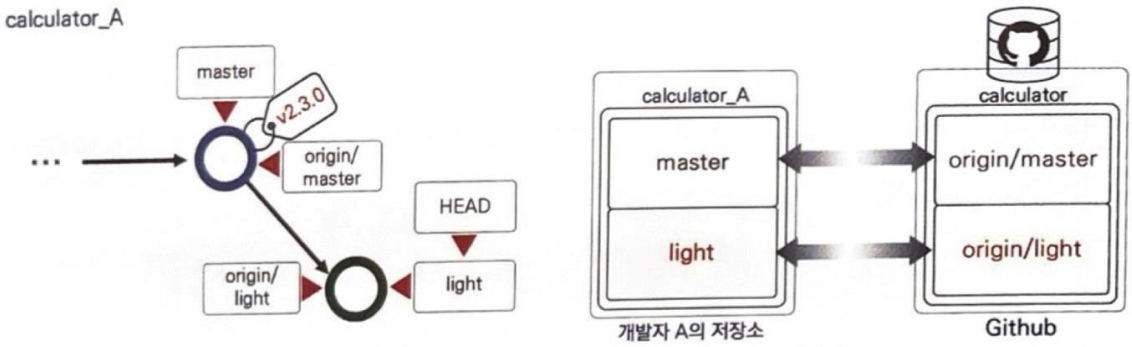


그림 7.82 푸시 후 저장소 상태

Ch08 브랜치 운영전략

Pull request

Github는 이와 같은 병합 검토를 돕는 풀 리퀘스트(pull request)라는 기능을 제공함.

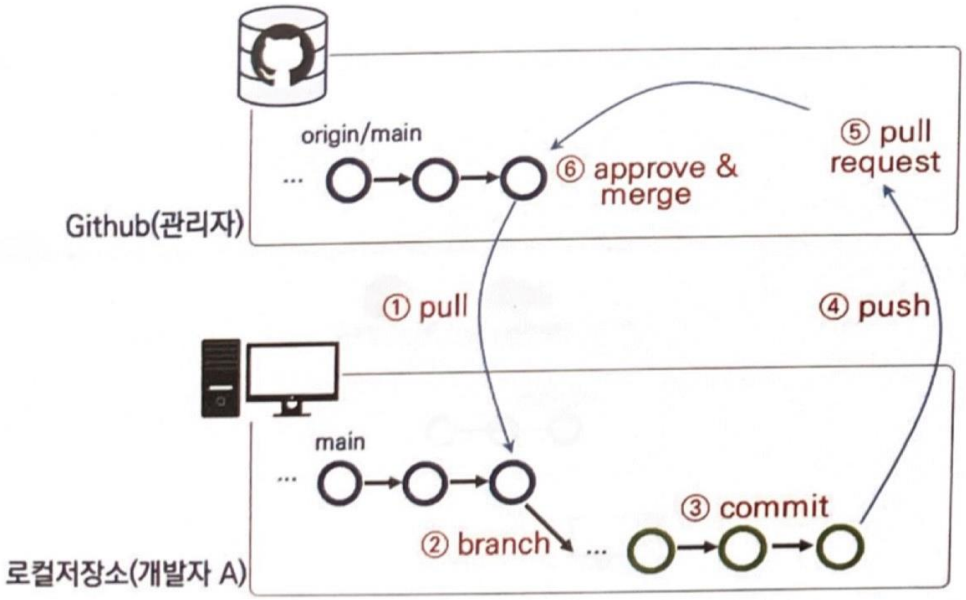


그림 8.2 풀 리퀘스트 개념

관리자용 원격저장소 생성하기

Branch name pattern : 설정을 반영할 브랜치 이름을 작성 ->main

Protect matching branches : 풀 리퀘스트 관련 옵션 지정

- Requset a pull request before merging : main 브랜치로 직접 푸시하지 않고. 풀 리퀘스트를 통합 병합을 사용하도록 설정합니다.
- Request approvals : 풀 리퀘스트 후에 협업자들의 승인을 얻어야 병합을 할 수 있도록 설정합니다. 병합을 위해 받아야 할 승인 숫자(Required number of approvals)도 설정할 수 있습니다. 여기서는 작업자는 관리자 포함 2명이므로 1명으로 설정합니다.
- Require review fro Code owners : 풀 리퀘스트 후, 저장소 관리자의 리뷰 승인이 필요하도록 설정합니다.

개발자 A : PC에 원격저장소의 내용 복제하기

개발자 A : feat1브랜치에서 ui.py 수정, 커밋, 푸시하기

github에서 풀 리쉐스트 생성하기

->origin/main에 병합하기 위해서는 github 원격저장소에 접속해서 풀 리퀘스트를 생성해야 함.

관리자 : github에서 풀 리쉐스트 승인하기

개발자 A :원격저장소에 병합된 내용 가져오기

*풀 리퀘스트를 위한 또다른 브랜치 운영 방법

협업대상이 명확하지 않은 경우->오픈소스 프로젝트의 저장소나 개인이 저장소의 코드를 공개해서 불특정 다수로부터 개선의 도움을 받고자 할 때는 협업자를 특정하기 어려움->원복저장소를 Fork(포크)하여 풀 리퀘스트를 생성할 수 있음.

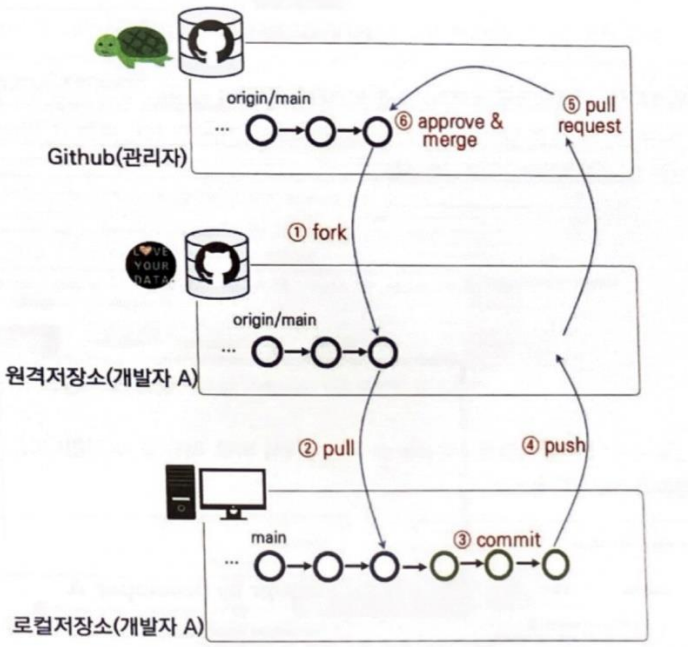


그림 8.28 저장소를 포크(fork)하여 풀 리퀘스트를 진행하는 절차

Gitflow

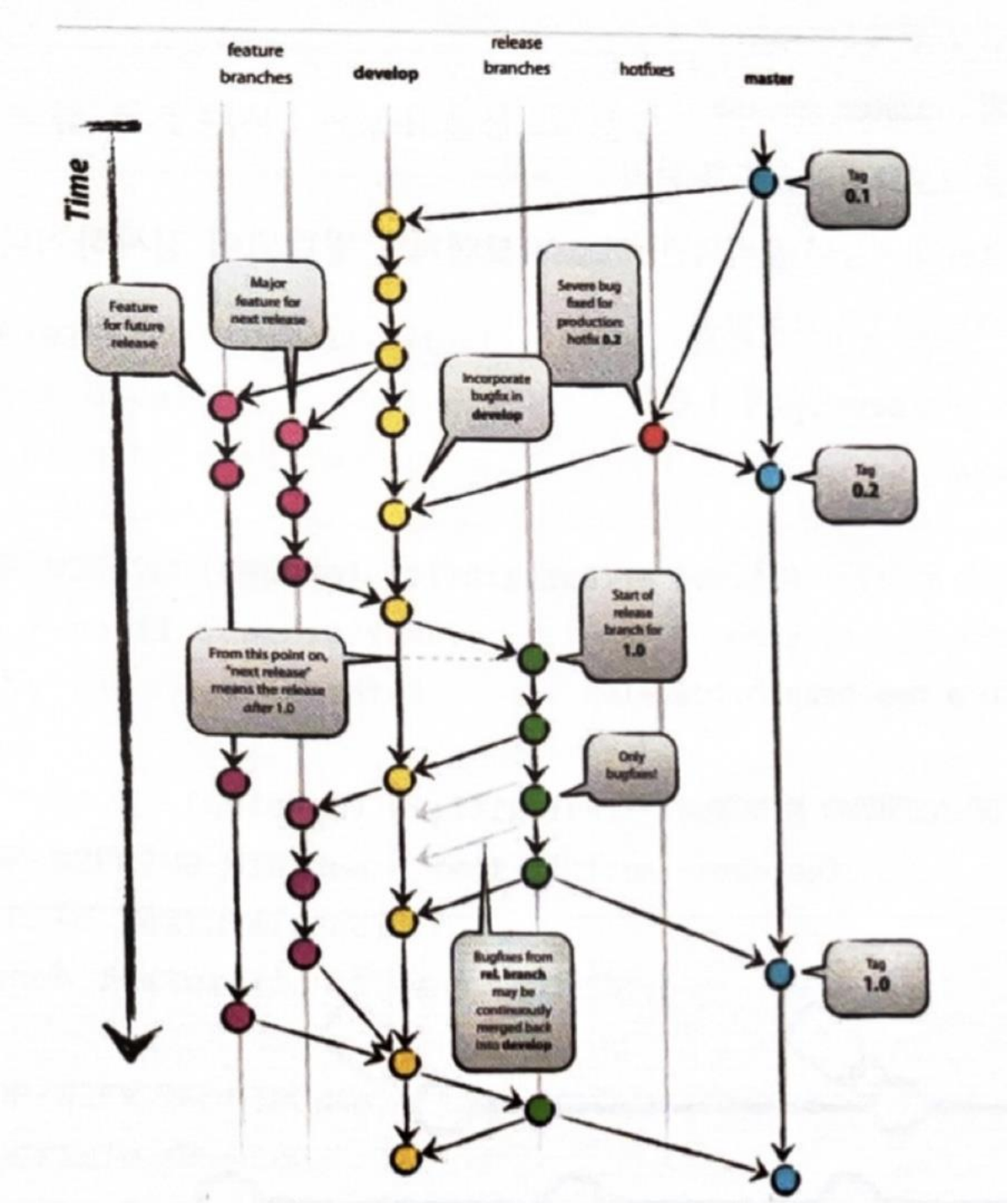


그림 8.38 Gitflow 브랜치 운영 개념
(출처: https://nvie.com/posts/a-successful-git-branching-model/)

Gitflow의 브랜치들

Master

- 과거에 배포된 코드 또는 앞으로 배포될 최종 단계의 코드가 관리되는 곳, 원격저장소에서 관리
- 태그와 함께 버전 정보가 기록됨
- 원격저장소(origin/master)에서 관리

Develop

- 분기되어 나온 브랜치 : master
- 병합할 브랜치 : master, release
- 다음에 배포할 프로그램의 코드를 관리, 원격저장소에서 관리
- 배포할 프로그램의 기능이 준비되면 release 브랜치로 병합되어 검사하거나, master로 병합되어 배포 버전으로 태그를 부여받음.
- 원격저장소(origin/develop)에서 관리

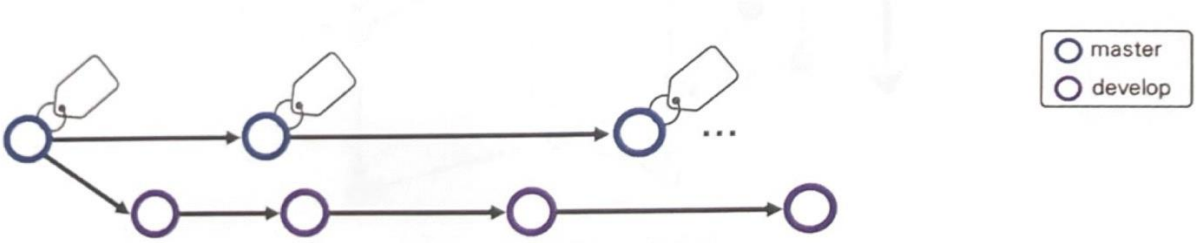


그림 8.39 master와 develop의 관계

Feature

- 분기 되어 나온 브랜치 : develop
- 병합할 브랜치 : develop
- 브랜치 이름 규칙 : feature/func1
- 토픽 브랜치라고도 부르며, 다음 배포에 추가될 기능을 개발하는 브랜치, 보통 개발자의 로컬 저장소에
- 기능이 완성되면 develop으로 병합됨
- 반드시 develop으로 병합할 필요는 없음. 예를 들어 기능이 필요하지 않거나 만족스럽지 않으면 병합하지 않고 삭제 가능

- 보통 개발자의 로컬저장소에 위치하며 원격저장소에는 푸시하지 않음.

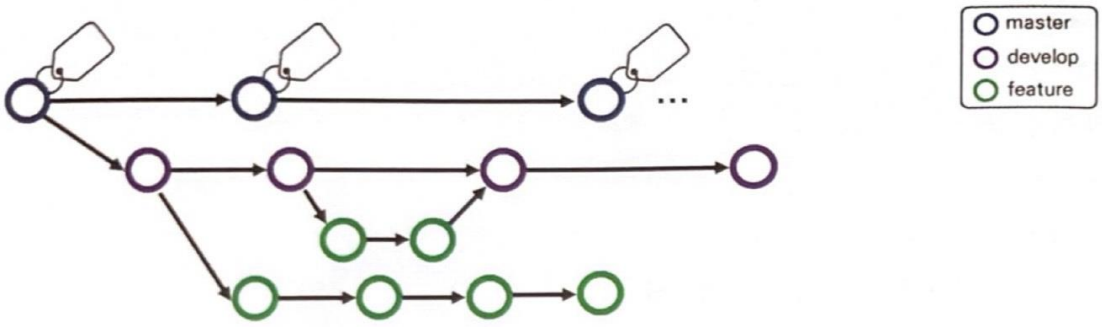


그림 8.40 develop과 feature의 관계

Release

- 분기되어 나온 브랜치 : develop
- 병합할 브랜치 : develop, master
- 브랜치 이름 규칙 : release/v0.1
- 기능이 완성된 develop 브랜치의 코드를 병합해서 배포를 준비하는 브랜치
- 배포에 필요한 준비, 품질 검사, 버그 수정 등을 진행
- Release 브랜치로 인해 develop 브랜치는 다음 배포에 필요할 기능에 집중할 수 있음
- Release 브랜치가 배포할 상태가 되면, master로 병합
- 이후 마스터에 만들어진 커밋에 태그를 추가함.
- 배포 후 release 브랜치가 필요 없으면 삭제(선택 사항)

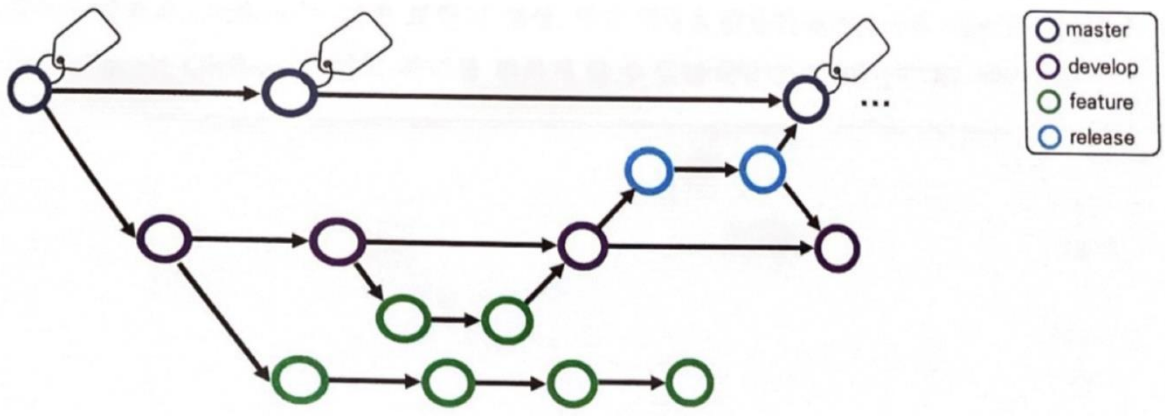


그림 8.41 master, develop와 release의 관계

Hotfix

- 분기되어 나온 브랜치 : master
- 병합할 브랜치 : develop, master
- 브랜치 이름 규칙 : hotfix/버전 정보
- 기존 배포한 버전에 문제를 해결하기 위한(버그 수정 등) 브랜치
- 브랜치는 문제가 발생한 master의 버전으로부터 생성
- 문제 해결 후에는 master와 develop에 각각 병합을 진행
- 만약 release 브랜치가 삭제되지 않았다면 release 브랜치에도 병합 진행
- 병합 후 브랜치는 삭제

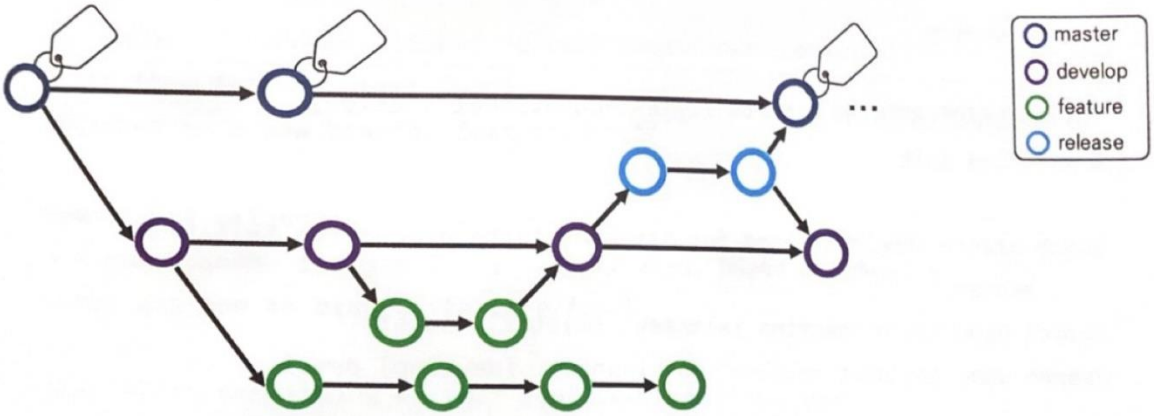


그림 8.42 master, develop와 hotfix의 관계

Git flow cheatsheet

1 초기화하기

```
git flow init
```

```
kmyu@DESKTOP-N2FK7H0 MINGW64 /c/test-cheatsheet (master)
$ git flow init

Which branch should be used for bringing forth production releases?
   - master
Branch name for production releases: [master] master
Branch name for"next release" development: [develop] develop

How to name your supporting branch prefixes?
Feature branches? [feature/] feature/
Bugfix branches? [bugfix/] bugfix/
Release branches? [release/] release/
Hotfix branches? [hotfix/] hotfix/
Support branches? [support/] support/
Version tag prefix? []
Hooks and filters directory? [C:/test-cheatsheet/.git/hooks]

kmyu@DESKTOP-N2FK7H0 MINGW64 /c/test-cheatsheet (develop)
$ git branch
* develop
  master

kmyu@DESKTOP-N2FK7H0 MINGW64 /c/test-cheatsheet (develop)
$
```

기존 Git 저장소에서 git flow init을 입력하면 사용할 브랜치의 이름과 접두어(prefix)를 설정하는 과정을 거치게 됩니다. 초기에 생성된 브랜치는 master와 develop이고 develop 브랜치로 전환된 상태가 됩니다.

❷ 브랜치(feature, release 등) 시작하기

```
git flow [브랜치 종류] start [브랜치 이름]
```

만약 func1이라는 feature 브랜치를 생성하고 싶다면 다음과 같이 입력합니다.

```
git flow feature start func1
```

```
kmyu@DESKTOP-N2FK7H0 MINGW64 /c/test-cheatsheet (develop)
$ git flow feature start func1
Switched to a new branch 'feature/func1'

Summary of actions:
- A new branch 'feature/func1' was created, based on 'develop'
- You are now on branch 'feature/func1'

Now, start committing on your feature. When done, use:

     git flow feature finish func1


kmyu@DESKTOP-N2FK7H0 MINGW64 /c/test-cheatsheet (feature/func1)
$
```

❸ 브랜치 완료하기

```
git flow [브랜치 종류] finish [브랜치 이름]
```

예를 들어 feature/func1 브랜치에서 작업을 마친 후, develop 브랜치로 병합, 사용을 다 한 feature/func1를 일괄 삭제하고 싶다면 다음과 같이 입력합니다.

```
git flow feature finish func1
```

```
kmyu@DESKTOP-N2FK7H0 MINGW64 /c/test-cheatsheet (feature/func1)
$ git flow feature finish func1
Switched to branch 'develop'
...

Summary of actions:
- The feature branch 'feature/func1' was merged into 'develop'
- Feature branch 'feature/func1' has been locally deleted
- You are now on branch 'develop'
```

명령어 정리

Git init : 저장소(Repository) 생성

Git config user.name "[작성자 이름]" : 사용자 이름 설정

Git config user.email “[이메일 계정]” : 사용자 이메일 설정

Git config –list : 저장소 설정 전체 출력

Git config [설정 항목] : 해당 항목 설정 값 출력(ex : git config user.name)

Git help [명령어] : 도움말

Git status : 저장소의 상태 정보 출력

Git add [파일 이름] : 해당 파일을 Staging Area에 올리기

Git add[디렉터리 이름] : 해당 디렉터리 안에 수정된 모든 파일을 Staging Area에 올리기

Git add . : Working Directory 안에 추가, 수정된 모든 파일을 Staging Area에 올리기

Git commit : 이력 저장, 커밋->i->commit->esc->:wq

Git commit -m “[메시지]” : vim을 사용하지 않고 인라인으로 메시지를 추가하여 커밋

Git commit -am “[메시지]” : git add와 git commit을 한꺼번에 명령

Git status : 저장소 파일의 상태 출력

Git status -s : 저장소 파일 상태를 간략하게 표시

Git log : 저장소의 커밋 히스토리(로그, 이력)를 출력

Git log –pretty=oneline : 로그 출력 시 커밋을 한 줄로 간략하게 표시(--pretty 옵션 사용)

Git log –onelinf : 로그 출력 시 커밋을 한 줄로 표시(해시도 앞자리 7글자만 출력)

Git log –graph : 로그를 그래프 형태로 출력

Git show : 가장 최근 커밋의 상세 정보 출력

Git show [커밋 해시] : 해당 커밋의 상세 정보 출력

Git show HEAD : HEAD가 참고하는 커밋의 상세 정보 출력

Git show HEAD^^ : HEAD를 기준으로 3단계 이전의 커밋 정보 출력

Git show HEAd ~n : HEAD를 기준으로 n단계 이전의 커밋 정보 출력

Git diff : 최근 커밋과 변경 사항이 발생한 파일들의 내용 비교

Git diff –staged : 최근 커밋과 스테이징된 파일 간의 변경 사항 출력

Git diff [커밋 해시1] [커밋 해시2] : 두 커밋 사이의 파일 간 변경 사항 출력

Git reset : Staging Area의 파일 전체를 언스테이징 상태로 되돌리기

Git reset [파일 이름(또는 경로) : 해당 파일(또는 경로)을 언스테이징 상태로 되돌리기

Git commit –amend : 최근 커밋 수정하기

Git commit –amend -m "[메시지]" : 해당 메시지로 커밋 수정하기

Git checkout[커밋 해시] : 해당 커밋으로 파일 상태 변경

Git checkout - : HEAD가 이전에 참조했던 커밋으로 상태 변경

Git checkout master : HEAD가 master를 참조

Git checkout HEAD~n: HEAD를 기준으로 n단계 이전 커밋으로 상태 변경

Git reset[커밋 해시] : 해당 커밋으로 브랜치의 참조를 변경

Git reset –hard[커밋 해시] : Working Directory, Staging Area, 커밋 모두 리셋

Git reset –mixed[커밋 해시] : Working Directory 유지, Staging Area와 커밋은 리셋, default option

Git reset –soft[커밋 해시] : Working Directory와 Staging Area 유지, 커밋 리셋

Git reset HEAD^ : HEAD를 기준으로 직전의 커밋으로 리셋

Git reset HEAD~n : HEAD를 기준으로 n단계 전 커밋으로 리셋

Git reflog : HEAD가 참조한 커밋 이력을 출력

Git remote : 현재 브랜치에 추가된 원격저장소 리스트 출력

Git remote -v(--verbose) : 현재 브랜치에 추가된 원격저장소 리스트 출력(주소 포함)

Git remote add[원격저장소 이름][원격저장소 주소]:해당 이름으로 원격저장소의 주소 등록

Git remote rm[원격저장소 이름] : 해당 원격저장소를 등록리스트에서 삭제

Git push -u(--set-upstream-to)[원격 저장소의 이름][로컬저장소의 브랜치]:로컬저장소의 브랜치가 원격저장소를 추적하도록 설정하고, 파일들을 원격저장소로 저장

Git push[원격저장소 이름][로컬저장소 브랜치] : 로컬 브랜치의 변경사항을 원격 브랜치로 업로드

Git push : upstream(-u) 설정 후 인자 생략 가능

Git pull[원격 저장소 이름][로컬저장소 브랜치]:원격저장소의 정보를 현재 로컬 브랜치에 가져와서 병합(fetch+merge) '예를 들어 원격저장소의 브랜치가 origin/master이면, git pull origin master'

Gti pull : upstream(-u) 설정 후 인자 생략 가능

Git tag : 로컬저장소의 모든 태그를 조회

Git tag[태그 이름] : 현재 커밋에 태그를 생성(Ligthweight 태그)

Git tag[태그 이름][커밋 해시] : 해당 커밋에 태그를 생성(Lightweight 태그)

Git tag -a [태그 이름] -m"[메시지]"[커밋 해시]:메시지를 추가하여 태그 생성(Annotataed tag)

Git tag -am[태그 이름]"[메시지]" : 현재 커밋에 메시지를 추가하여 태그 생성(Annotated tag)

Git show[태그 이름]:해당 태그가 부착된 커밋의 상세정보 확인

Git push –tags : 생성된 전체 태그를 원격저장소에 푸시(=git push origin –tags)

Git push[태그 이름]: 해당 태그를 원격저장소에 푸시(= git push origin "[태그 이름]")

Git tag -d[태그 이름]:해당 태그 삭제

Git push -d[태그 이름]:원격저장소에서 해당 태그 삭제

Git revert[커밋 해시]:해당 커밋을 되돌리기

Git revert –no-edit[커밋 해시] : 커밋 메시지 수정 없이 기본 메시지로 되돌리기

Git revert -n[커밋 해시] : 커밋하지 않고 스테이징 상태로 되돌리기

Git revert[커밋 해시1]..[커밋 해시2] : 해당 구간만큼 커밋 되돌리기, 커밋 해시1은 되돌려 지지 않

음.

Git clone[원격저장소 주소] : 원격저장소의 내용을 복제. 폴더 이름은 원격저장소 이름과 동일

Git clone[원격저장소 주소][폴더 이름] : 해당 폴더 이름으로 원격저장소의 내용을 복제

Git diff origin/master master : origin/master와 로컬저장소의 master 차이 비교

Git merge –abort : 병합(merging) 작업 종료

Git log –oneline –graph : 저장소의 커밋 이력을 한 줄로, 그래프를 추가하여 출력

Git fetch [브랜치 이름] : 해당 브랜치의 기록된 커밋(작업 내용)을 가져오기

Git merge [브랜치 이름]: 해당 브랜치의 작업 내용을 현재 브랜치에 병합하기

Git blame [파일 이름] : 파일의 작성자 정보 확인

Git blame[커밋 해시][파일 이름] : 해당 커밋에서 파일의 작성자 정보 확인

Git blame -L [시작라인], [끝 라인][파일 이름] : 파일 안의 특정 구간의 작성자 정보만 출력

Git blame -e [파일 이름] : 작성자 이름 대신 이메일 정보 표시

Git blame -s [파일 이름]: 커밋 해시만 표시

Git stash : 인덱스 영역에 트래킹되는 파일을 임시 영역에 저장하고, modified 부분을 Working Directory에서 제거, 기본명칭 WIP로 저장됨.

Git stash -u : 새롭게 추가된 파일(untracked)도 함께 임시 영역에 저장

Git stash save [저장 이름]: 저장 이름을 부여해서 저장

Git stash -m "[메시지]" : 메시지를 기록하여 저장

Git stash list : stash 기록 확인

Git stash apply : 가장 최근의 작업 내용 불러오기

Git stash apply [stash 인덱스]: 해당 인덱스에 해당되는 저장 내용 반영

Git stash drop : 가장 최근의 stash 제거

Git stash drop [stash 인덱스] : 해당 인덱스의 stash 삭제

Git stash pop : 가장 최근의 작업 내용을 불러오고, stash 영역에서 삭제

Git stash pop[stash 인덱스]: 해당 인덱스의 작업 내용을 불러오고, stash영역에서 삭제

Git stash clear : stash 영역의 모든 내용 삭제

Git branch : 브랜치 목록 표시

Git branch [브랜치 이름] : 해당 브랜치 이름으로 브랜치 생성

Git checkout[브랜치 이름] : 해당 브랜치로 전환

Git chechout -b [브랜치 이름] : 브랜치 생성과 동시에 전환

Git branch -m [브랜치 이름][새로운 브랜치 이름] : 브랜치 이름 변경

Git branch -d [브랜치 이름] : 해당 브랜치 삭제

Git merge [브랜치 이름] : 해당 브랜치의 작업 내용을 현재 브랜치에 병합

Git merge –ff[브랜치 이름]:fast-forward 관계에 있으면 커밋을 생성하지 않고 현재 브랜치의 참조만 변경(default)

Git merge –no-ff[브랜치 이름]:fast-forward 관계에 있어도 머지 커밋 생성

Git merge –squash[브랜치 이름]:병합할 브랜치의 내용을 하나의 커밋에 합침. 병합할 브랜치 정보는 생략

Git rebase[브랜치 이름]: 현재 브랜치가 해당 브랜치(브랜치 이름)부터 분기하도록 재배치

Git rebase –continue : 충돌 수정 후 재배치 진행

Git rebase –abort : rebase 취소

Git cherry-pick[커밋 해시]:해당 커밋의 내용을 현재 브랜치에 추가. 뒤에 커밋 해시를 연속해서 입력하면 복수 추가 가능

Git cherry-pick[시작 지점의 커밋 해시]...[끝 지점의 커밋 해시]:해당구간의 commit을 한번에 추가

Git cherry-pick—abort : 충돌과 같은 상황이 발생했을 때 cherry-pick 취소

Git cherry-pick—continue : 충돌 상황 해결 후 cherry-pick 진행

Git push -u(--set-upstream-to) [원격저장소 이름][로컬 브랜치 이름] : 로컬저장소의 브랜치가 원격저장소를 추적하도록 설정하고 파일들을 원격저장소의 브랜치로 푸시

Git push [원격저장소 이름][로컬 브랜치 이름]:로컬 저장소의 변경 사항을 원격저장소로 업로드

Git push : 업스트림(-u)설정 후 인자 생략 가능

#명령어 용도별 정리

Setup

- git init : 저장소 생성

- git clone [원격저장소 url] : 해당 원격저장소의 내용을 복제하여 로컬저장소 생성

- git config user.name "[작성자 이름]" : 작성자 이름 설정

- git config user.email "[이메일 계정]" : 작성자 이메일 설정

- git config --list : 저장소 설정 전체 출력

- git config --get [설정 창목] : 일부 설정된 항목만 출력(git config --set user.name 등)

- git help [커맨드 이름] : 도움말

Stage & commit

- git add [파일 이름] : 수정된 파일을 Staging Area 올리기

- git add [디렉토리 이름] : 해당 디렉토리 내에 수정된 모든 파일을 스테이징

- git add . : Working Directory 내에 수정된 모든 파일을 스테이징(untracked 파일 제외)

- git commit : 이력 저장, 커밋(commit)

- git commit -m "[메시지]" : 메시지 편집기를 사용하지 않고 인라인으로 메시지를 추가하여 커밋

- git commit am "[메시지]" : 스테이징(add)과 커밋을 일괄 진행

Inspect

git status

- git status : 저장소 과일의 상태 정보 출력

- git status -s : 파일 상태 정보를 간략하게 표시

git log

- git log : 저장소의 커밋 이력을 출력

- git log –pretty=oneline : 커밋을 한 줄로 출력(--pretty 옵션 사용)

- git log --oneline : 각 commit을 한 줄로 출력

- git log d --decorate=full : 브랜치나 태그 정보를 상세히 출력

- git log –graph : 그래프 형태로 출력

git show

- git show : 가장 최근 커밋 정보를 출력

- git show [커밋 해시] : 해당 커밋의 정보 출력

- git show HEAD : HEAD가 참조하는 커밋의 정보 출력

- git show HEAD^^^ : HEAD를 기준으로 3단계 이전의 커잇 정보 출력

- git Show HEAD-[n] : HEAD를 기준으로 n단계 이전의 커밋 정보 출력

git diff

- git diff : 최근 커밋과 변경 사항이 발생한(Unstaged) 파일들의 내용 비교

- git diff --staged : 최근 커밋과 Staging Area의 파일 간의 변경 사항 출력

- git diff [커밋 해시1] [커밋 해시2] : 두 커밋 간 파일의 변경 사항 출력

checkout

- git checkout [커밋 해시] : 해당 커밋으로 HEAD의 참조를 번경(Working Directory의 내용을 변경)

- git checkout - : HEAD의 참조를 직전에 참조했던 커밋으로 변경

- git checkout master : HEAD가 master를 참조

- git checkout HEAD~n : 현재 HEAD의 참조를 기준으로 n단계 이전 커밋으로 참조를 변경

Undoing changes

- git reset : Staging Area의 파일 전체를 unstaged 상태로 되돌리기

- git reset [파일 이름] : 해당 파일을 unstaged 상태로 되돌리기

- git commit --amend : 최근 커밋을 수정하기

- git commit --amend -m "[커밋 메시지]" : 해당 메시지로 커밋 수정하기

- git reset [커밋 해시) : 해당 커밋으로 브랜치의 참조를 변경

- git reset --hard [커밋 해시] : Working Directory, Staaing Area, 커밋 모두 되돌리기

- git reset --mixed [커밋 해시] : Working Directory 유지, Staing Area와 커밋은 되돌리기(default)

- git reset --soft [커밋 해시) : Working Directory와 Staging Area는 유지. 커밋은 되돌리기

- git reset HEAD^ : HEAD를 기준으로 직전의 커밋으로 브랜치의 참조 변경

- git reset HEAD~[정수] : HEAD를 기준으로 정수값 단계 전 커밋으로 브렌치의 참조 반경

Setup

- git branch : 브랜치 목물 표시

- git branch [브랜치 이름] : 해당 브랜치 이름으로 브랜치 생성

- git checkout [브랜치 이름) : 해당 브랜치로 전환

- git checkout -b [브랜치 이름) : 브랜치 생성과 동시에 전환

- git branch -m [브랜치 이름] [새로운 브랜치 아름] : 브랜치 이름 변경

- git branch -d [브랜치 이름] : 해당 브랜치 삭제

Merge. rewrite

merge

- git merge [브랜치 이름] : 현 브랜치에 해당 브랜치의 내용 병합

- git merge -ff [브랜치 이름] : fast-forward 관계에 있으면 commit을 생성하지 않고 현재 브랜치의

참 조 값만 변경(default)

- git merge --no-ff [브랜치 이름] : fast-forward 관계에 있어도 머지 커밋(merge commit) 생성

- git merge --squash [브랜치 이름] : fast-forward 관계에 있어도 머지 커밋 생성. 브랜치 정보 생략

- git rebase –abort : rebase 취소

rebase

- git rebase [브랜치 이름] : 현재 브랜치가 해당 브랜치(브랜치 이름)에부터 분기하도록 재배치

- git rebase --continue : 충돌 수정 후 재배치 진행

- git rebase –abort : rebase 취소

cherry-pick

- git cherry-pick [커밋 해시] : 해당 커밋의 내용을 현재 브랜치에 추가. 뒤에 커밋 해시를 연속 입력하면 복수로 추가 가능

- git cherry-pick [시작 지정의 커밋 해시].. [끝 지점의 커밋 해시] : 해당 구간의 커밋을 한번에 추가

- git cherry-pick --abort : 충돌이 발생했을 때 cherry-pick 취소

- git cherry-pick --continue : 충돌 해결 후 cherry-pick 진행

gitignore 파일 작성 규칙

소스코드 관리와 관련이 적은 파일들을 .gitignore 파일에 정의해 두면 Git의 추적 대상에서 제외되고 원격저장소의 푸시 대상에서도 제외함.

표현	예시	의미

#		주석, 제외 파일 지정에 영향을 미치지 않음
[파일 이름]	file.txt	해당 파일 이름으로 된 저장소의 모든 파일 무시
/[파일 이름]	/file.txt	현재 경로에 있는 해당 파일만 무시
*.[확장자]	*.txt	해당 확장자로 된 모든 파일을 무시
[폴더 이름]/	folder/	해당 폴더 아래의 모든 파일을 무시
[폴더 이름]/[파일 이름]	folder/file.txt	해당 경로의 파일 무시
[폴더 이름]/*.[확장자]	foleder/*.txt	해당 폴더 아래에 해당 확장자를 가진 모든 파일 무시
![파일 이름]	!file.txt	해당 파일 이름으로 된 것은 예외(무시 안함)

➔ https://www.toptal.com/developers/gitignore