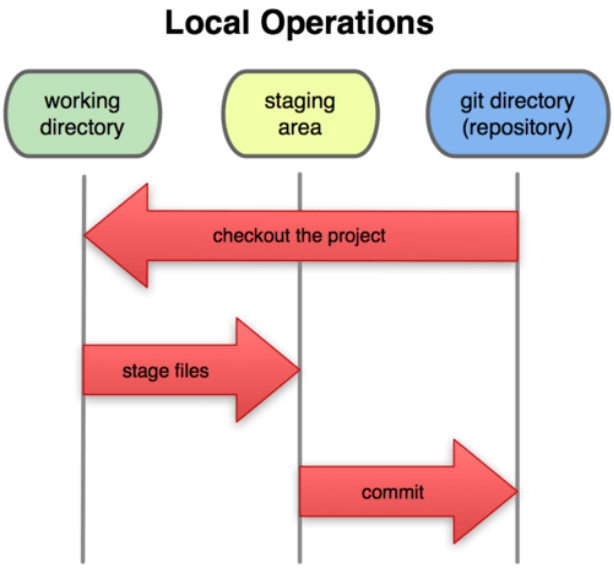
Git 기본 용어

***깃(Git)**

- Code(Project)에 대한 Version 변경을 관리할 수 있는 분산 버전 관리 소프트웨어

***깃허브(GitHub)**

- 프로젝트 버전관리(Git)를 지원하는 웹 호스팅 서비스
- 로컬이 아닌 리모트(웹)로 깃을 사용해 협업이 가능



***Working Area (Working Directory, 워킹디렉토리)**

- Unstaged 상태인 Modified file(수정된 파일)들이 있는 디렉토리. git status를 통해 확인할 수 있다.

***Staging Area**

- Working Directory(워킹디렉토리)에 있는 file을 git add 하게되면, 파일은 'Staged(커밋 준비 상태)'상태가 되고 이때 Staged file은 Staging Area에 위치하게 된다. 커밋을 준비(to be committed)하는 위치라고 생각하면 된다.

***커밋(Commit)**

- '커밋'은 업데이트를 확정한다는 의미. 스냅샷 처럼 확정된 순간의 코드 상태를 커밋 메세지와 함께 저장소(Repository, Git Directory)에 기록(저장)한다.

★ commit = hash + message + author + 코드 snapshot

(ref. 롤백(Rollback) = 업데이트 취소 처리)

***브랜치(Branch)**

- 작업을 할 때, 메인 프로젝트 브랜치(나뭇가지, 일반적으로 'master' branch)로 부터 추가 가지를 뻗어(Branch off) 작업이 마무리 되면 메인 브랜치(가지)에 병합(merge) 한다. ('Head' 는 현재 작업 중인 브랜치를 가리킨다)

깃 터미널 명령어 (Git Command Line)

명령어 정리를 찾다가 찾은 Git Cheat Sheet PDF 파일

첨부파일

git terminal command line ㄱ ㅣㅅ ㅌ ㅓㅁ ㅣㄴ ㅓㄹ ㅁ ㅕㅇㄹ ㅕㅇ ㅇ ㅓ.pdf

파일 다운로드

git terminal command line ㄱㅣㅅ ㅌㅓㅁㅣㄴㅓㄹ ㅁㅕㅇㄹㅕㅇㅇㅓ pdf 파일

> git help

가장 많이 사용하는 깃 명령어 21개가 나온다.

```
$ git help
usage: git [--version] [--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>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   git clone     Clone a repository into a new directory
   git init      Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   git add       Add file contents to the index
   git mv        Move or rename a file, a directory, or a symlink
   git restore   Restore working tree files
   git rm        Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   git bisect    Use binary search to find the commit that introduced a bug
   git diff      Show changes between commits, commit and working tree, etc
   git grep      Print lines matching a pattern
   git log       Show commit logs
   git show      Show various types of objects
   git status    Show the working tree status

grow, mark and tweak your common history
   git branch    List, create, or delete branches
   git commit    Record changes to the repository
   git merge     Join two or more development histories together
   git rebase    Reapply commits on top of another base tip
   git reset     Reset current HEAD to the specified state
   git switch    Switch branches
   git tag       Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   git fetch     Download objects and refs from another repository
   git pull      Fetch from and integrate with another repository or a local branch
   git 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.

> git help -a (이때 a는 all)

사용할 수 있는 **Git Command 명령어들의 전체 리스트**가 나온다.
'q'를 누르면 해당 화면(vim editor)에서 빠져나올 수 있다.

(+) 특정 명령어가 자세하게 궁금할 땐!

git help {명령어} 를 실행하면, 해당 명령어의 매뉴얼 페이지가 떠서 Name / Synopsis / Description(설명) / Option(세부 옵션) 을 자세하게 볼 수 있다.

Main Porcelain Commands

git add	Add file contents to the index
git am	Apply a series of patches from a mailbox
git archive	Create an archive of files from a named tree
git bisect	Use binary search to find the commit that introduced a bug
git branch	List, create, or delete branches
git bundle	Move objects and refs by archive
git checkout	Switch branches or restore working tree files
git cherry-pick	Apply the changes introduced by some existing commits
git citool	Graphical alternative to git-commit
git clean	Remove untracked files from the working tree
git clone	Clone a repository into a new directory
git commit	Record changes to the repository
git describe	Give an object a human readable name based on an available ref
git diff	Show changes between commits, commit and working tree, etc
git fetch	Download objects and refs from another repository
git format-patch	Prepare patches for e-mail submission
git gc	Cleanup unnecessary files and optimize the local repository
git gitk	The Git repository browser
git grep	Print lines matching a pattern
git gui	A portable graphical interface to Git
git init	Create an empty Git repository or reinitialize an existing one
git log	Show commit logs
git merge	Join two or more development histories together
git mv	Move or rename a file, a directory, or a symlink
git notes	Add or inspect object notes
git pull	Fetch from and integrate with another repository or a local branch
git push	Update remote refs along with associated objects
git range-diff	Compare two commit ranges (e.g. two versions of a branch)
git rebase	Reapply commits on top of another base tip
git reset	Reset current HEAD to the specified state
git restore	Restore working tree files
git revert	Revert some existing commits

git rm Remove files from the working tree and from the index
git shortlog Summarize 'git log' output
git show Show various types of objects
git stash Stash the changes in a dirty working directory away
git status Show the working tree status
git submodule Initialize, update or inspect submodules
git switch Switch branches
git tag Create, list, delete or verify a tag object signed with GPG
git worktree Manage multiple working trees

Ancillary Commands / Manipulators
git fast-export Git data exporter
git fast-import Backend for fast Git data importers
git filter-branch Rewrite branches
git mergetool Run merge conflict resolution tools to resolve merge conflicts
git pack-refs Pack heads and tags for efficient repository access
git prune Prune all unreachable objects from the object database

Ancgit illary Commands / Interrogators
git annotate Annotate file lines with commit information
git blame Show what revision and author last modified each line of a file
git count-objects Count unpacked number of objects and their disk consumption
git difftool Show changes using common diff tools
git fsck Verifies the connectivity and validity of the objects in the database
git gitweb Git web interface (web frontend to Git repositories)
git help Display help information about Git
git instaweb Instantly browse your working repository in gitweb
git merge-tree Show three-way merge without touching index
git rerere Reuse recorded resolution of conflicted merges
git show-branch Show branches and their commits
git verify-commit Check the GPG signature of commits
git verify-tag Check the GPG signature of tags

> git config

- git의 환경 설정을 위한 명령어.

(+) git config --list

: 설정된 내용들을 확인할 수 있다. ex. 유저 이름, 유저 이메일, 리모트 저장소 위치 등

: 역시 'q'를 눌러 빠져나올 수 있다.

> git pull

- Fetch from and integrate with another repository or a local branch.

(+) git pull origin master

: origin은 remote를 의미하고, Remote Repogitory의 master 브랜치에서 로컬로 Fetch(복사)해서 Merge하는 것을

의미한다.

> git status

- 현재 로컬 파일들의 상태를 확인할 수 있다. (modified, untracked, staged, committed 등)

> git add {파일명}

- Workint Directory(워킹디렉토리)의 파일들을 Staging Area로 보내 Staged (커밋 준비 상태)로 만든다.

(+) git add -i (= git add --interactive)

: Working Directory의 파일들을 Optional하게 Cammand 할 수 있다.

: Add modified contents in the working tree interactively to the index. Optional path arguments may be supplied to limit operation to

a subset of the working tree. See "Interactive mode" for details

```
$ git-studies git:(master) ✗ git add -i
           staged     unstaged path
  1:        +1/-0        +1/-1 test.txt
  2:        +1/-1     nothing user.txt

*** Commands ***
  1: status     2: update     3: revert     4: add untracked
  5: patch      6: diff   7: quit   8: help
```

> git restore --staged {파일명}

- git add와 반대로 staging area에 있는 파일을 working directory로 다시 보낸다. 다시 unstaged 상태로 만듦!

> git diff

-Working Directory에 위치한 파일들의 수정상태를 보여준다.

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

(+) git diff --cached (= git diff --staged)

: (커밋 Commit 직전에) Staged File의 어떤 부분이 변경되었는지 확인될 때 엄청 이용한다.

: This form is to view the changes you staged for the next commit relative to the named <commit>. Typically you would want comparison with the latest commit, so if you do not give <commit>, it defaults to HEAD. If HEAD does not exist (e.g. unborn branches) and <commit> is not given, it shows all staged changes. --staged is a synonym of --cached.

> git commit

- 커밋! 명령어를 치면 Commit Message 커밋 메세지 작성을 위해 Vim Editor가 뜨는데 'I'를 눌러 Insert 상태로 만들고 Commit Message 커밋 메세지 작성 후 → esc 클릭 → :wq 라고 작성하면 에디터에서 빠져나올 수 있다. 무사히 커밋이 되었다면, 로컬 저장소에 커밋 메세지와 함께 수정된 파일이 잘 저장 되었을 것이다.

(+) git commit -m {커밋 메세지}

: 귀찮다면, -m 옵션을 추가해서 한 줄로 커밋을 할 수도 있다. 하지만, 이런 경우 커밋 메세지가 title(Vim Editor에서 1번째 줄)과 message body(Vim Editor에서 3번째 줄)로 구성되지 않기 때문에 자세한 설명이 필요한 commit이라면 추천하지 않는다.

(+) git commit --amend

: 커밋 메세지를 commit message 수정할 때 사용하는 명령어이다. 단, Push 작업 전에만 가능하다!

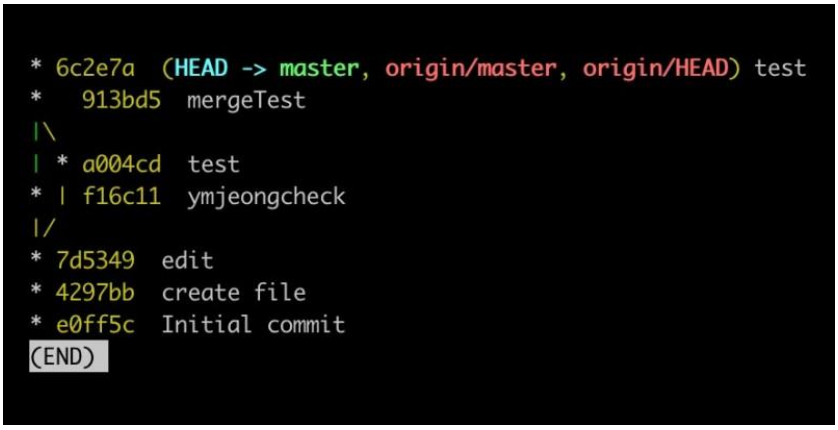
> git log

- 커밋 히스토리를 확인할 수 있다. (잘 커밋 되었나 확인할 때)
- git log의 옵션들을 잘 활용하면, log 파악이 정말 쉬워진다 꼬옥 알아두고 응용해보자.
- 항상 로그를 통해 현재 어떤 상태인지 체크하는 습관이 정말 중요!

★ 커밋 히스토리를 그래프로 보여주는 git log 옵션 정리!

(+) git log --oneline --graph

: 로그 명령어에 해당 옵션을 붙이면 이렇게 커밋 히스토리들을 브랜치 그래프로 볼 수 있다. (아주 기본적인 커밋 히스토리 ASCII 그래프)



git log --oneline --graph 출력결과

► git log 옵션 설명

--oneline

: (--pretty=oneline) + (--abbrev-commit) 두 옵션 기능이 합쳐진 옵션!

--pretty=oneline

: 해당옵션을 적용하면 date, author 정보는 볼 수 없지만 항목별 1줄로 표현해줘서 가독성이 좋아진다

--graph

: 브랜치와 머지 히스토리 정보를 아스키(ASCII) 그래프로 보여준다

--abbrev-commit

: 40자 SHA(Secure Hashed Algorithm) 해시 코드의 일부만 보여주는 옵션.

--name-status

: 수정된 파일 목록을 상태와 함께 알려준다. (graph랑 같이 사용하면 그래프가 무척 길어짐)

-(n)

: 최근 n개의 커밋만 조회할 수 있다

--since="YYYY-MM-DD", --after="YYYY-MM-DD"

: 해당 날짜 이후의 커밋 히스토리만 보여준다. (참고. "N years N day N week"와 같은 형식으로도 입력 가능)

--until="YYYY-MM-DD", --before="YYYY-MM-DD"

: 해당 날짜 이전의 커밋 히스토리만 보여준다.

--author="authorEmail or authorName"

: Remote Repository의 경우 여러명이 협업할 수 있기 때문에 author(작업 원작자)를 통해 필터해서 커밋 히스토리를 조회할 수 있다. (참고. git config --list를 조회하면 user.name과 user.email 확인 가능)

--committer="committerEmail or committerName"

: 특정 커미터(Committer, 커밋한 사람)을 필터해서 커밋 히스토리를 조회할 수 있다.

--grep="someText"

: Commit Message 속 텍스트를 필터해서 조회할 수 있다. someText를 포함한 모든 커밋들을 보여줌! sql에서 like '%someText%'와 같은 기능.

--no-merges

: Create a Merge Commit시 발생하는 커밋은 제외하고 볼 수 있다.

★ --pretty=format:

: format을 직접 입력해 로그 포맷을 설정할 수 있다 (개인적으로 로그 출력 커스터마이징... 넘 재미있었음)

► --pretty=format: 옵션 설명

%C(auto or colorName) ~ %Creset : ~ 부분의 색을 지정

%H : 커밋 해시

%h : 짧은 길이 커밋 해시

%T : 트리 해시

%t : 짧은 길이 트리 해시

%P : 부모 해시

%p : 짧은 길이 부모 해시

%an : 저자 이름

%ae : 저자 메일

%ad : 저자 시각 (--date 옵션을 통해 시각도 포맷팅 가능)

%ar : 저자 상대적 시각

%cn : 커미터 이름

%ce : 커미터 메일

%cd : 커미터 시각 (--date 옵션을 통해 시각도 포맷팅 가능)

%cr : 커미터 상대적 시각

%s : 요약 (커밋메세지 title, 첫 번째 줄)

▼ 로그 출력 포맷 커스터마이징 과정

step1. 위의 옵션들을 참고해 원하는 포맷을 작성해 명령어 실행해 본다.

$ git log --oneline --graph

--pretty=format:'%C(auto)%h%d%Creset %C(cyan)(%cd, %cr)%Creset %C(green)%cn%Creset %s'

--date=short

step2. 출력된 로그를 보며 짜잔!이라고 외쳐본다

하지만... 매번 로그 출력 할 때마다 저렇게...? 치면...? 얼마나 귀찮...?

팁! 특정 명령어를 Alias(별칭) 설정을 해두면 넘 편리합니다!

▼ 특정 명령어 별칭 Alias 설정 방법

$ git config --global alias.{별칭} "{명령어}"

--global은 별칭을 사용할 수 있는 옵션 값인데, 전역적으로 사용하고 싶다면 --global을, 지역적으로 사용하고 싶다면 옵션을 빼고 설정하면 된다. 나는 아래 예시 처럼 'lg'라는 별칭으로 설정해뒀고, 추후엔 git lg 단축 명령어로 실행할 수 있게 되었다.

참고로, git config --list를 통해 현재 어떤 단축어들이 설정되어 있는지 확인할 수 있다.

$ git config --global alias.lg "log --oneline --graph --pretty=format:'%C(auto)%h%d%Creset %C(cyan)(%cd, %cr)%Creset %C(green)%cn%Creset %s'--date=short"

(+) Git Alias 명령어 별칭 삭제 방법

$ git config --global --unset alias.{별칭}

> git push origin master

- Remote repository의 'master' branch로 push해서 branch를 업데이트 시킨다.

> git branch {새로운 브랜치 이름}

- {브랜치 이름} 브랜치를 새로 생성한다.

(+) git branch -d {삭제할 브런치 이름}

- 해당 브랜치를 삭제할 수 있다.

> git checkout {옮겨갈 브랜치 이름}

- 현재 위치한 브랜치에서 다른 브랜치로 위치를 옮길 때 사용한다. 이때, 옮겨갈 브랜치는 이미 존재해야 한다.

(+) git checkout -b {새로운 브랜치 이름}

: git branch + git checkout 의 수행을 합친 명령어. 옮겨갈 브랜치를 생성함과 동시에 이동한다.

> git stash

- 진행중인 작업을 임시로 저장해두고 싶을 때 사용하는 명령어.
- 아직 완료하지 않은 일을 잠시 다른 공간에 저장해두고, 나중에 다시 (git stash pop) 꺼내와 작업을 마무리 할 수 있다.

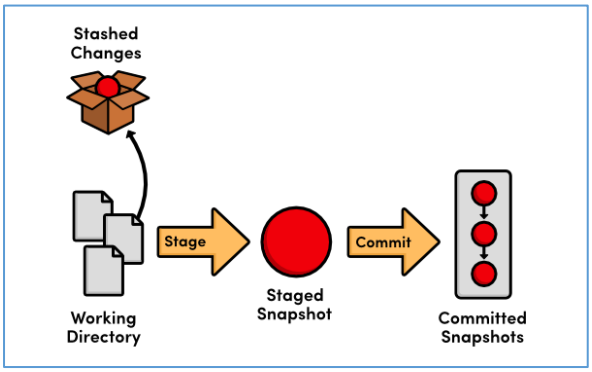
```
$ git-study git:(master) ✗ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   test.txt
        modified:   user.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
```

modified: test.txt

$ git-study git:(master) ✗ git stash
Saved working directory and index state WIP on master: 7d5349f edit

$ git-study git:(master) git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean



(+) git stash pop
: stash에 담겨진 작업들을 다시 불러온다.

(+) git stash list
: stash에 담겨진 작업들을 볼 수 있다.

(+) git stash drop
: '휴지통 비우기' 같은 느낌으로 stash에 저장된 작업들을 비우는 명령어이다.

> git cherry-pick {커밋 해시 넘버}
- A 브랜치에서 원하는 커밋을 복사해 B 브랜치에 새로운 커밋으로 생성할 수 있다. (잘라낸 게 아니라, 복사한 커밋이기 때문에 복사 후 옮겨진 커밋의 해시 넘버는 기존 커밋과 다르다.)
- B 브랜치의 위치에서 git cherry-pick {A브랜치에 속한 커밋 해시 넘버}을 실행해야한다!
- cherry-pick을 사용한 경우, 기존 A 브랜치에서 복사된 커밋들은 (더 이상 필요가 없어졌으니) git reset --hard 해주는 것 잊지 말기...!
- 마케팅 용어 중에 '체리 피커 (Cherry Picker), 체리가 장식된 케이크에서 하나뿐인 체리를 쏙 빼먹는 사람, 본인에게 이득이 되는 부분만 쏙쏙 챙기는 사람'을 의미하는 용어가 있는데 git의 cherry-pick도 같은 의미로 사용된다. 필요한 커밋만 쏙 복사해오는 췌리~픽! 아이라잌잇ㅋ

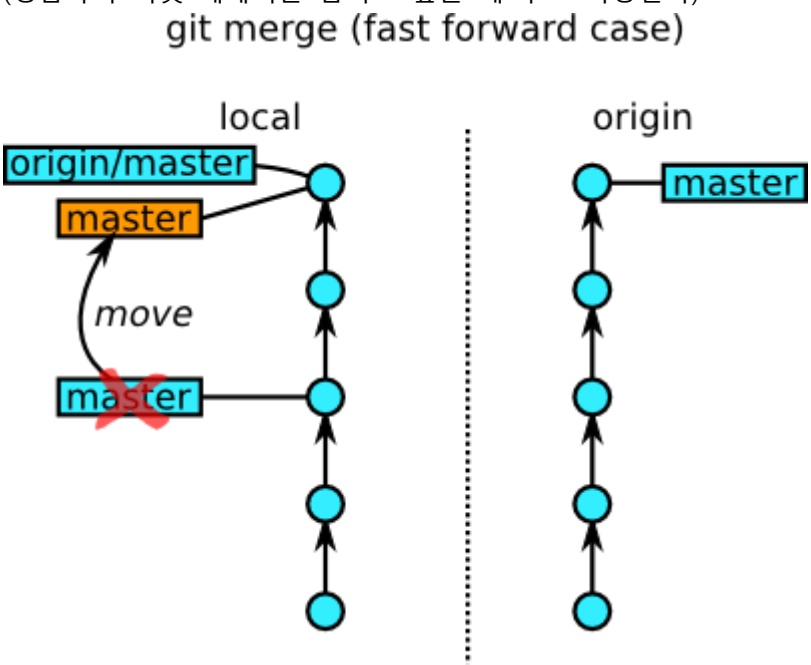
> git merge {병합할 브랜치}
- 현재 브랜치에 병합할 브랜치를 병합한다.

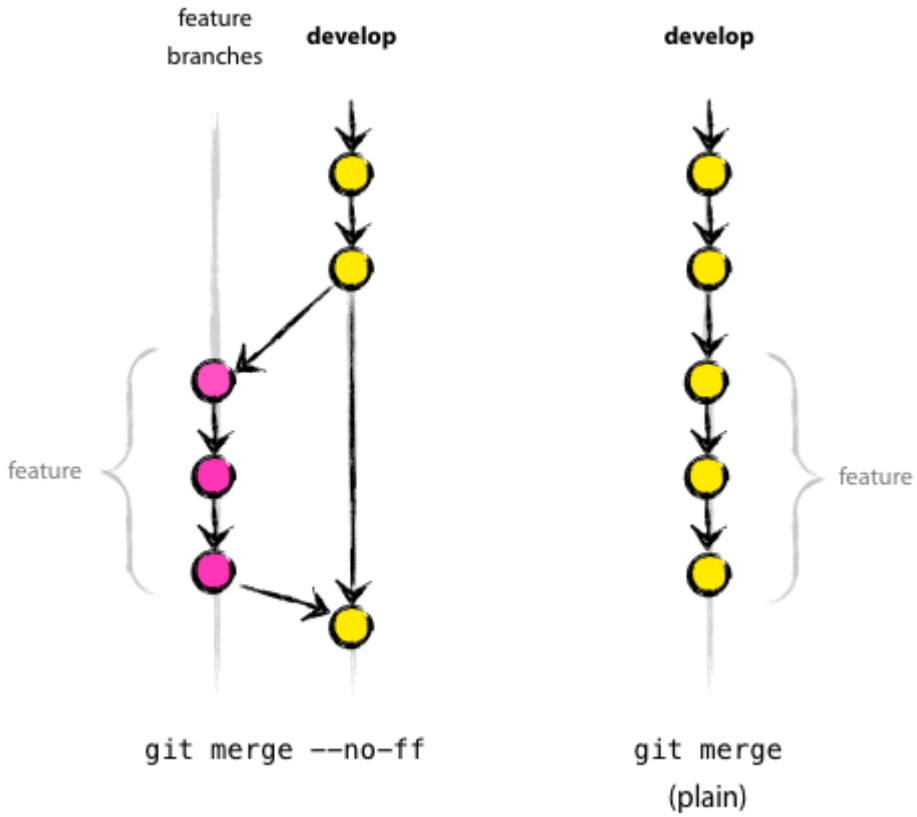
***Fast Forward?**
- 병합할 브랜치의 commit이 현재 브랜치 보다 앞서 있는 상황에서 현재 브랜치의 커밋을 병합할 브랜치의 커밋으로 이동하겠다는 의미

(+) 현재 브랜치가 타겟 브랜치와 동일 log tree → Fastforward merge : conflict 가 절대!! 발생하지 않음

(+) 현재 브랜치가 타겟 브랜치와 서로 갈라진 log tree → Create merge commit : conflict 발생할 수 있음

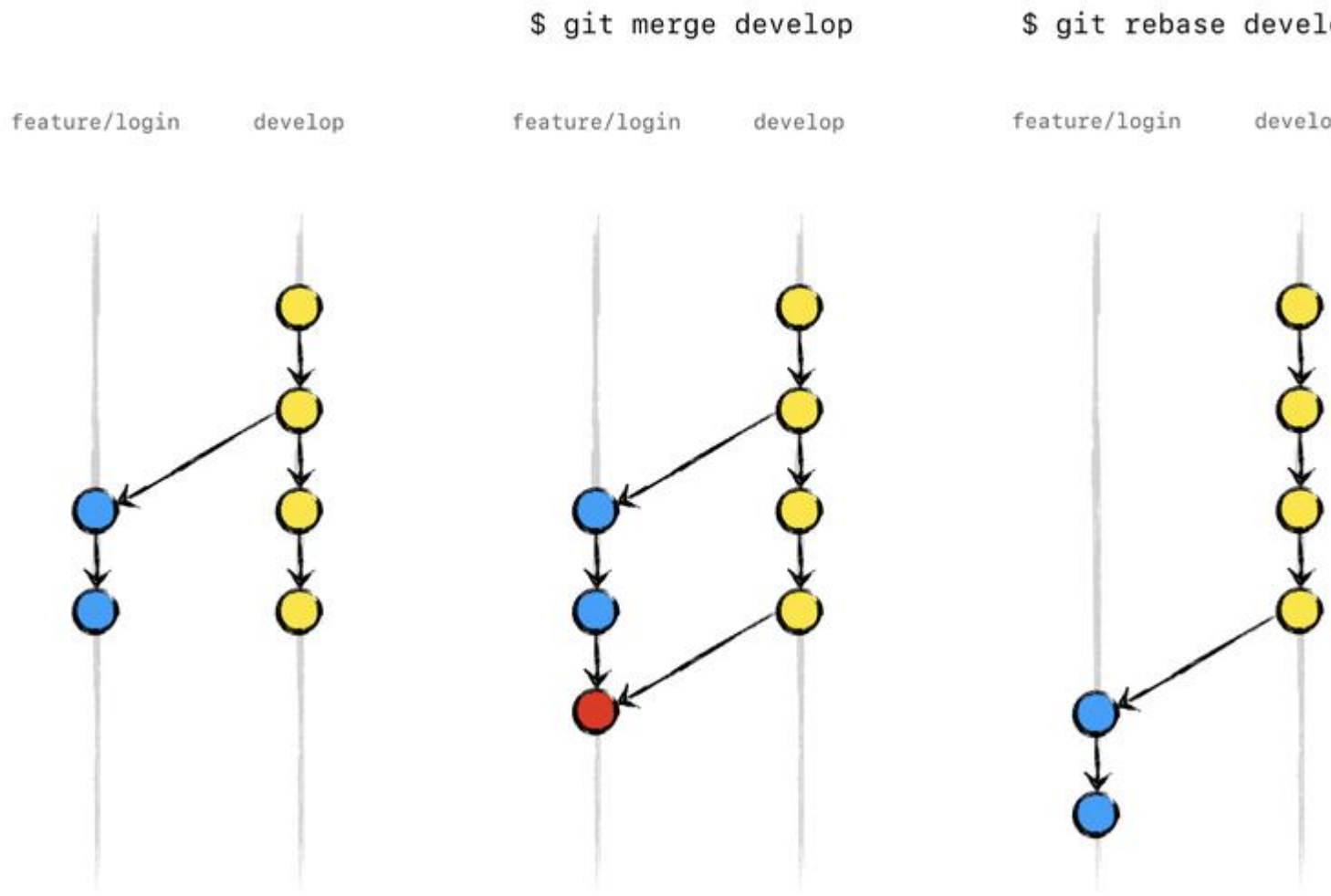
(+) git merge --no-ff : fast-forward가 가능한 상황에서 fast-forward 하지 않고 create merge commit 하고 싶을 때 (병합하며 커밋 메세지를 남기고 싶을 때 주로 사용한다)





> git rebase

- 'Base' 브랜치(일반적으로 'master' 브랜치)를 'Re' 다시 설정한다는 뜻.
- 현재 브랜치와 타겟 브랜치 사이의 공통 부모에서 부터 갈라져 나온 모든 커밋들을 타겟 브랜치 앞에 다시 쌓기
- 기존 커밋과는 SHA(Secure Hash Algorithms) 값이 다른, 새로운 커밋이 만들어지는 것!



> git reset

- Reset current HEAD to the specified state

(+) git reset --soft / --mixed / --hard {커밋 해시 넘버 or HEAD~(커밋 역순서)}

ex. HEAD~1 = 마지막 커밋, HEAD~2 = 마지막 커밋의 이전 커밋

- --hard 옵션이 쏵다 날려주는 시원한 옵션이라 위험하지만 짜릿하다. 생코 영상에 정말 잘 설명이 되어 있어서 강의도 함께 첨부해둔다!

	working directory working tree working copy	index staging area cache	repository history tree
			git reset --soft
		git reset --mixed	
	git reset --hard		

> git reflog

- 브랜치를 삭제하거나, reset하거나, rebase로 병합하는 과정에서 기존 커밋이 사라진 것 같지만 사실 git은 모든 이력을 보관하고 있다. 모든 이력을 볼 수 있는 명령어가 바로 git reflog!

(+) git reset --hard {삭제된 커밋 해시 넘버}
: 삭제한 커밋을 다시 복구하고 싶다면! 이렇게!

(+) git checkout -b {삭제한 브랜치 이름}{커밋 해시 넘버}
: 삭제한 브랜치를 다시 살려내고 싶다면! 이렇게!

***Command Line 실행 중 멈추고 싶을 때!**
: Ctrl + c (작동그만~!) (+) quote 문에서도 빠져서 나오지 못 할때... 작동그만(ctrl + c)을 외쳐보자!

GitHub PR(Pull Request) Merge 3가지 방법

1. Create a merge commit (default option)
: feature 브랜치의 모든 커밋을 master 브랜치에 --no-ff 옵션이 적용된 채로 merge 된다.
: All commits from the feature branch will be added to the base branch via a merge commit.

2. Squash and merge
: Pull Request의 커밋들이 하나의 커밋으로 스쿼시(Squash, 압축)되고 --ff 옵션이 적용된 채로 base branch로 merge 된다. (새로운 merge 커밋 만들지 X)
: 여러개의 커밋을 1개로 줄이기 때문에 Repository의 Git History가 덜 어지럽다.
:The commits from the feature branch will be combined into one commit in the base branch.

3. Rebase and merge
: Pull Request의 커밋들을 개별적으로 Base Branch에 --ff 옵션이 적용되어 merge 된다.
: The commits from the feature branch will be released and added to the base branch.

개발팀 팀장님 주도로 사내 Git 스터디가 진행 중인데, Git 고수 개발자 분들의 꿀팁 대방출로 신입 개발자는 열심히 줍줍하며 써먹을 준비 중이다. github에서 Pull Request(PR)를 통해 merge를 하려고 할 때 merge 종류가 3가지가 있는데 오늘 단번에 이해되고 혼자 소름 돋음...
하지만 브랜칭 연습은 많이 해봐야 할 것 같다.