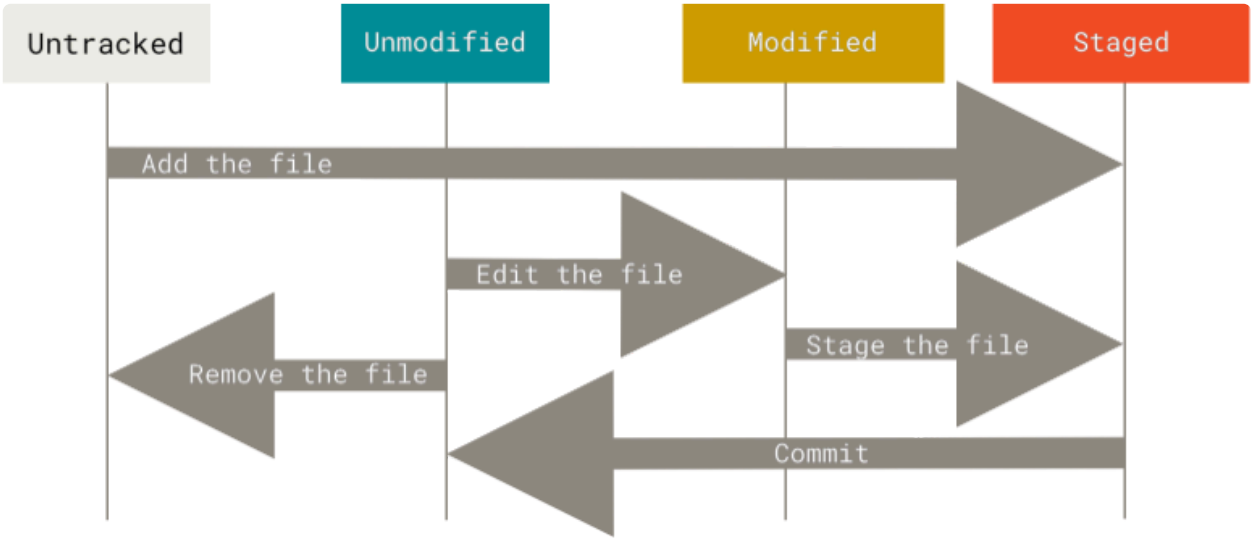
Session 1

> `$ git <command> -h`로 간략한 도움말을, `$ git <command> --help`로 상세한 도움말을 볼 수 있다. 명령어가 궁금하면 꼭 찾아보자.

깃은 왜 사용하나요?

- **Version Control System**(VCS)로서 각 버전을 관리할 수 있다.
- **git** 서버를 사용해 더 편하게 협업할 수 있다.

`commit`, `add`, `rm`의 파일 상태 변화



`add`

- **Untracked**인 파일을 **git에서 Track**할 수 있게 해준다. `add`로 추가된 파일들은 바로 **Staged** 상태가 된다. **Glob pattern**을 사용해 파일을 **Track**할 파일을 지정할 수 있다.
- `git add <path>`와 같이 사용한다.

`commit`

> 1. 각 커밋은 하나의 버전이다.
> 2. 모든 커밋은 부모 커밋에 대한 포인터를 가지고 있다.

- **Tracked** 중인 파일들의 변경 사항 중 **Staged** 된 것들을 저장하는 커밋을 생성한다.
- **Tracked**인 파일이 커밋될 경우 **Unmodified**가 되며, 이후에 수정되면 **Modified** 상태가 된다.

`rm`

- **git**에서 **Tracked** 상태인 파일을 **Untracked**로 바꾼다. 실제로 저장되어 있는 파일 또한 삭제된다.
- 만약 파일은 삭제하지 않고 **git**이 추적하는 것만 멈추고 싶다면 `.gitignore`에 파일을 등록한 후 `$ git rm --cached <file>`를 사용하면 된다.

`status`

- **git** 레포지토리의 변경 사항들을 보여준다.

`.gitignore` 파일

- 앞서 얘기했듯이 파일 중 **git**이 **Track**하기를 원치 않는 파일들을 지정하기 위해 사용된다. 빌드된 실행 파일, 로그, 환경변수 등을 일반적으로 등록해 사용한다.
- 레포지토리 내부의 하위 디렉토리에 추가적인 `.gitignore` 파일을 생성해 사용할 수 있다.

tags 사용법

- 이름을 붙인 커밋 포인터이며 추가적인 정보를 넣을 수 있다. 커밋을 가르키기 때문에 **tag**를 이용해 이동할 수 있다.
- **lightweight**와 **annotated** 두 종류가 있으며 **annotated**의 경우 태그에 추가적인 정보를 더 넣을 수 있다.
- `$ git tag -a`로 **annotated** 태그를 생성, `$git tag`로 조회할 수 있다.

HEAD와 Branch

Branch

- 커밋을 가르키는 포인터이다. 커밋으로 구성된 *linked list*의 가장 최신 커밋을 가르킨다.

  > `$ git branch <branch name>` 명령어로 생성할 수 있지만 `$ git checkout -b <branch name>`으로 생성하며 이동하는 게 더 편하다.

- **git**을 사용해 여러가지 작업을 할 수 있게 도와준다.

HEAD

- 일반적으로 **Branch**를 가르키는 포인터이다. 현재 작업중인 **Branch**를 가르킨다.

> *Dangling HEAD* 상태일 경우에는 커밋을 가르킨다.

커밋 사이를 이동하는 방법 `reset` & `checkout`

> 당장은 이 두 명령어의 공통점이 너무 없어 묶여있는 게 이상할 수 있지만 나중에 비교를 위해 묶어놓았다.

`checkout`

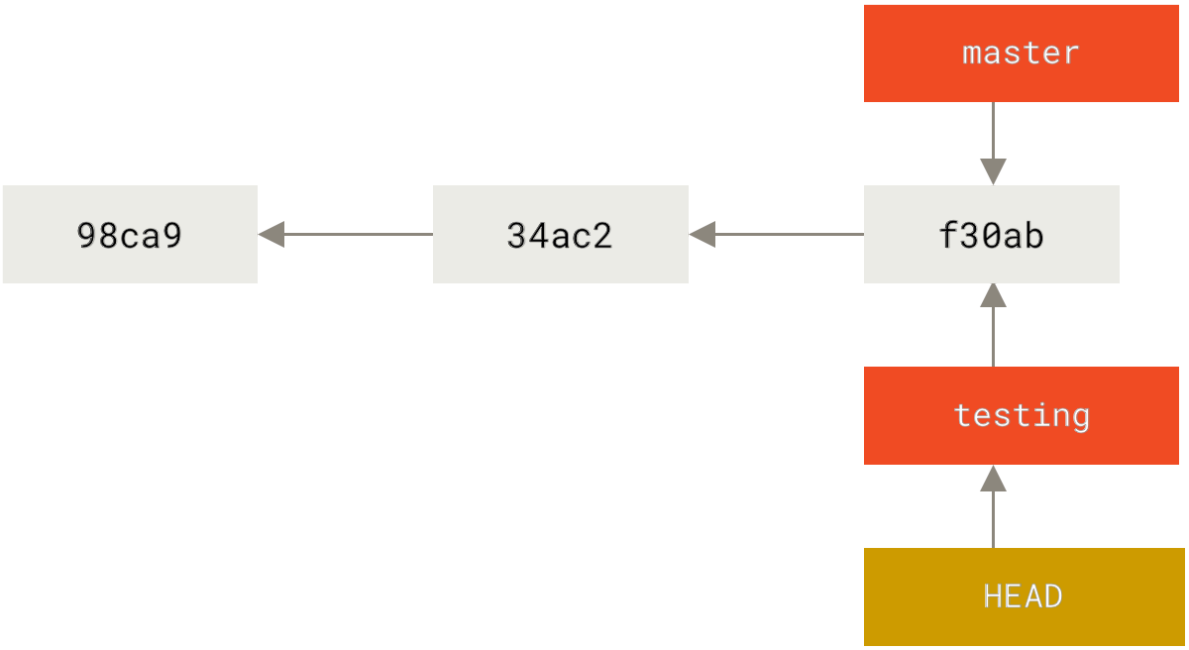
- `HEAD`를 다른 브랜치로 변경한다.
- `$ git checkout <target branch>`와 같이 사용할 수 있다.

  > `$ git checkout -`를 입력하면 긴 브랜치 이름 대신 이전에 있었던 브랜치로 이동할 수 있다.

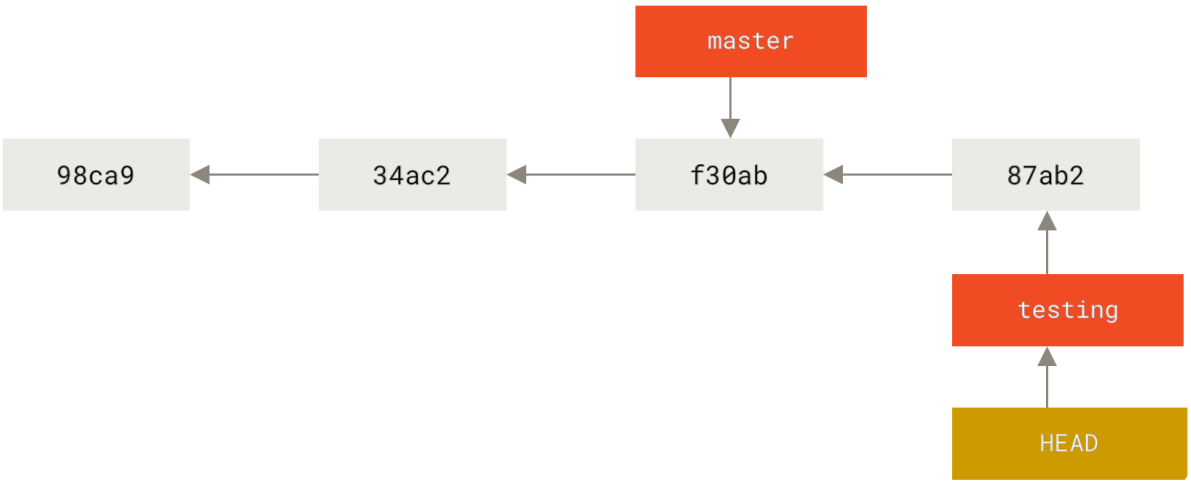
- git 2.23 버전부터 `switch` 명령어를 같은 역할을 위해 사용할 수 있다.

Example

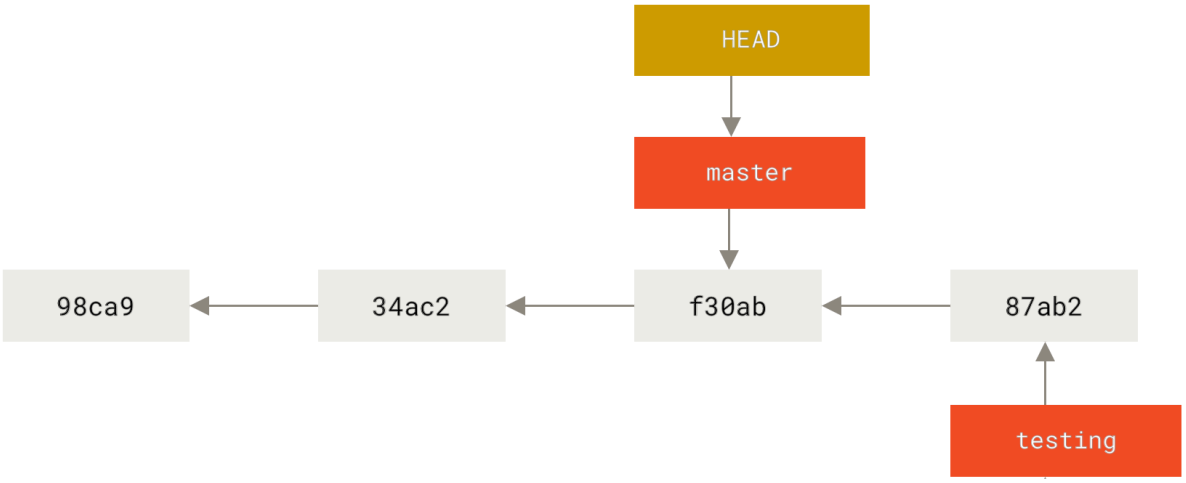
1. `master`에서 `testing` 브랜치로 `checkout`한 모습. 현재 두 브랜치는 모두 `f30ab`를 가르키고 있다.



2. `testing` 브랜치에서 `87ab2`를 생성한 모습. 이제 두 브랜치가 가르키는 커밋이 다르다. 이때 `HEAD`는 `testing`을 가르키고 있기 때문에 사용자의 레포지토리는 `87ab2`의 내용과 같다.



3. 다시 `master` 브랜치로 돌아온 모습. 사용자 레포지토리의 내용은 `f30ab`가 된다. 원한다면 언제든지 `testing` 브랜치를 활용해 `87ab2`의 내용을 불러올 수 있다.



`reset`

- `checkout`과 다르게 브랜치 사이를 이동하는 것이 아닌, 브랜치를 다른 커밋으로 이동시킨다.
- `git reset HEAD <file>` 명령어를 사용해 **Staged** 된 파일을 **Unstaged**로 만들 수 있다.
- git 2.23 이후부터는 `restore` 명령어를 더 직관적인 옵션으로 사용해 같은 일을 할 수 있다.

> `$ git restore --staged <file>` or `$ git restore <file>`

`revert` vs `restore` vs `reset`

- `revert`는 커밋들을 골라 그 커밋들에 담긴 변경사항을 취소시키는 커밋을 생성한다.
- `revert` 명령어는 혼자서 작업할 경우 `restore`, `reset`에 비해 사용할 일이 없는 명령어이다.
- `restore`는 `reset`의 작업을 할 수 있지만 더 의미가 전달되는 명령어이다.

> `reset`에 파괴적인 옵션이 더 많다.

`rebase` & `merge`

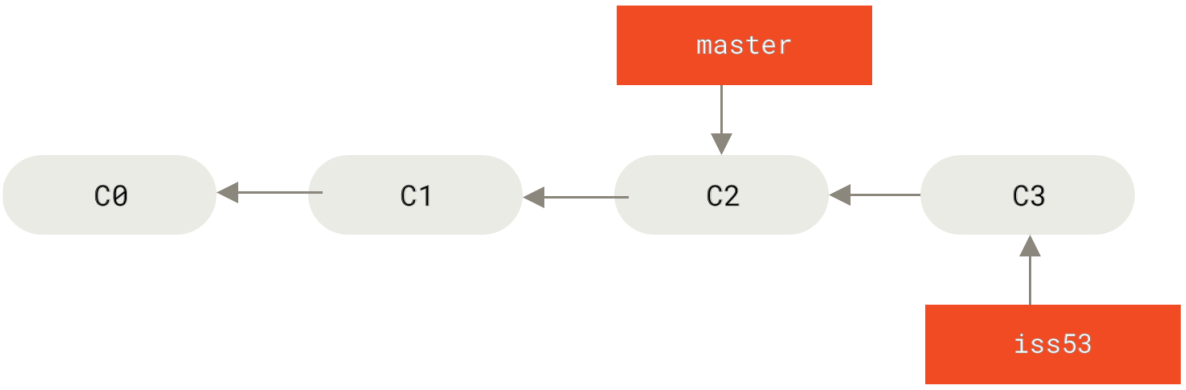
두 명령어는 모두 브랜치를 합칠 때 사용될 수 있다. `rebase`의 경우 유용하게 쓸 수 있는 경우가 있는데 이는 나중에 공부할 예정이다.

Merging

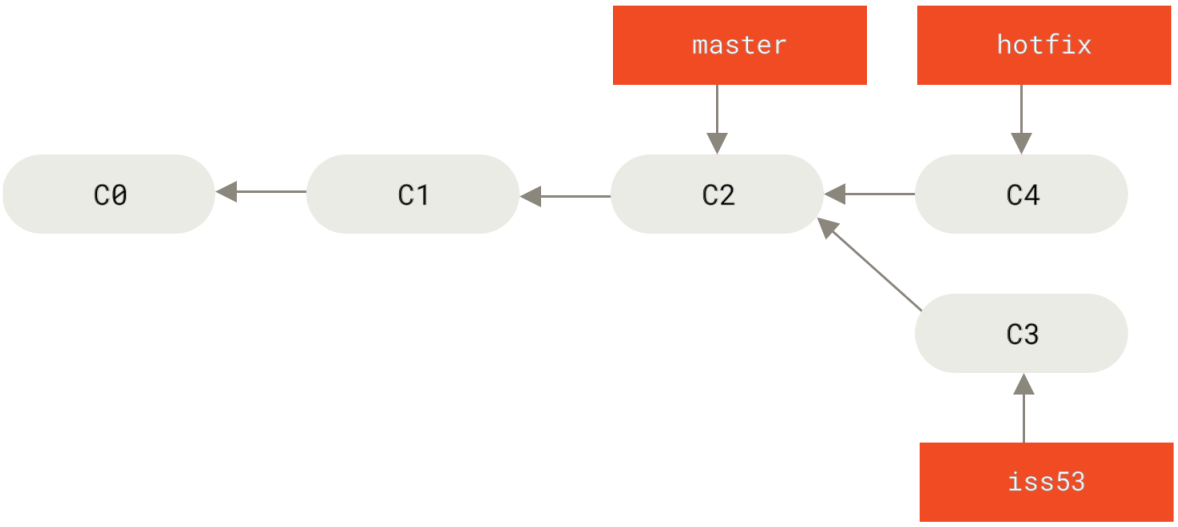
- *3-way merge*와 *fast-foward merge*로 구분할 수 있다. 둘 다 서로 다른 브랜치를 하나의 브랜치로 만들어준다.
- `$ git merge <target branch>`로 사용할 수 있으나 실제로 협업을 할 때는 대부분 **PR**을 이용해 브랜치를 병합한다.

Example

1. `master`에서 분기된 `iss53` 에서 `C3` 커밋을 생성했다.

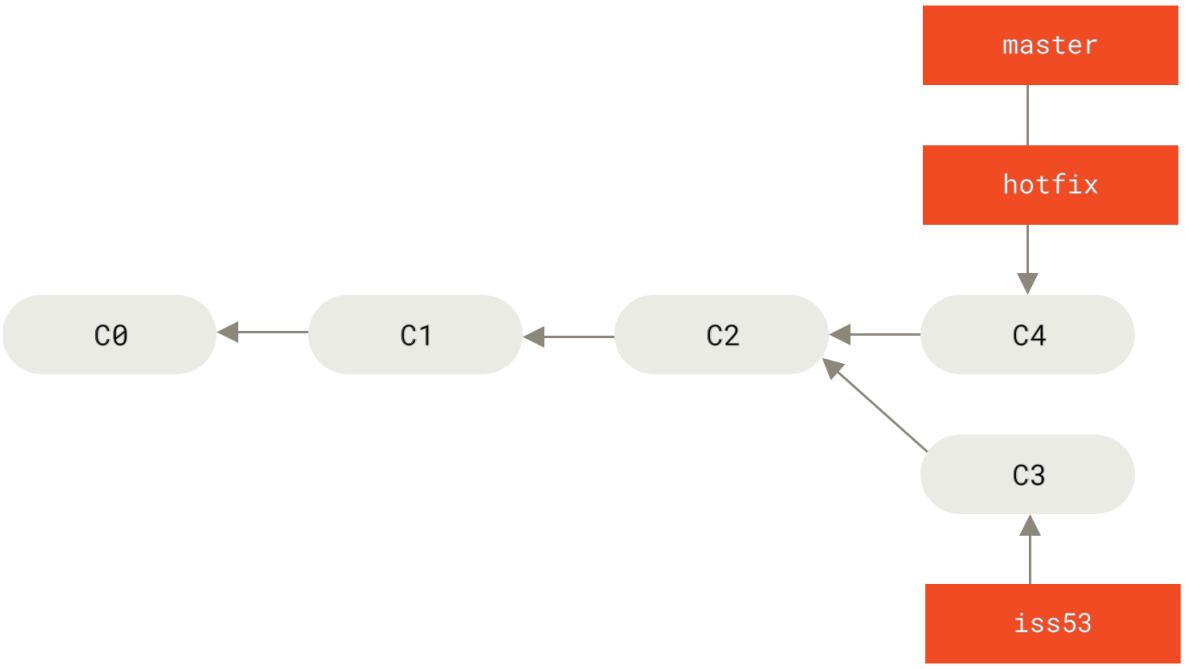


2. `master`에서 급하게 수정해야 할 사항들을 발견해 `hotfix` 브랜치를 추가로 만들고 `C4` 커밋을 생성했다.

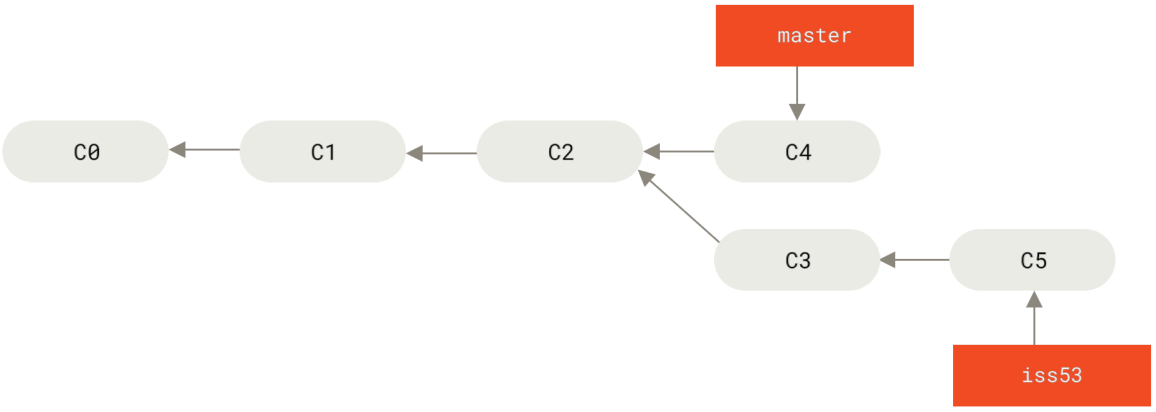


3. `master` 브랜치에서 `$ git merge hotfix` 명령어를 입력하면 `Fast-foward`라는 말이 포함된 출력이 나오면서 브랜치가 합쳐진다. `master`와 `hotfix`의 공통 조상 커밋은 `C2`였고 `master`는 `C2` 이후 추가된 커밋이 없기 때문에 *fast foward* 방식으로 `hotfix` 브랜치를 합

칠 수 있다.

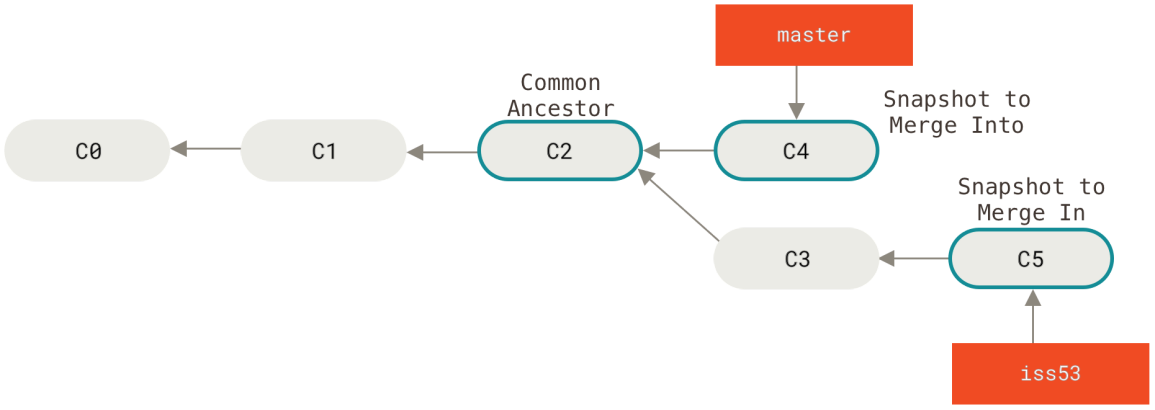


4. `iss53` 브랜치에서 `C5` 커밋을 추가적으로 생성했다.

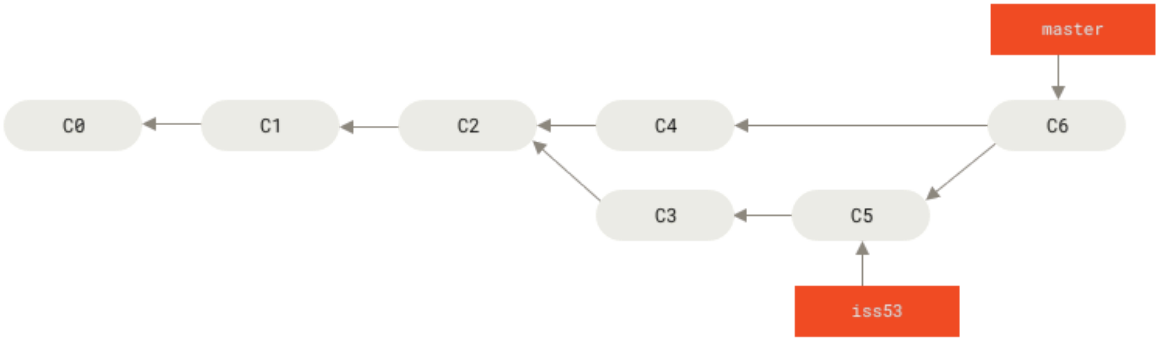


5. 이제 `master` 브랜치로 돌아와 `$ git merge iss53` 명령어를 입력하면 git은 공통 조상 커밋인 `C2`, 현재 있는 브랜치인 `master`의 `C4` 그리고 병합할 브랜치인 `iss53`의 `C5`를 이용

해 브랜치 병합을 진행한다.



6. 이 경우 두 개의 부모 커밋을 가지는 **merge commit** `C6`가 생성된다. 만약 이 과정에서 *merge conflict*이 발생할 경우 브랜치 병합이 중단되며 *conflict*를 해결해 줘야 한다.



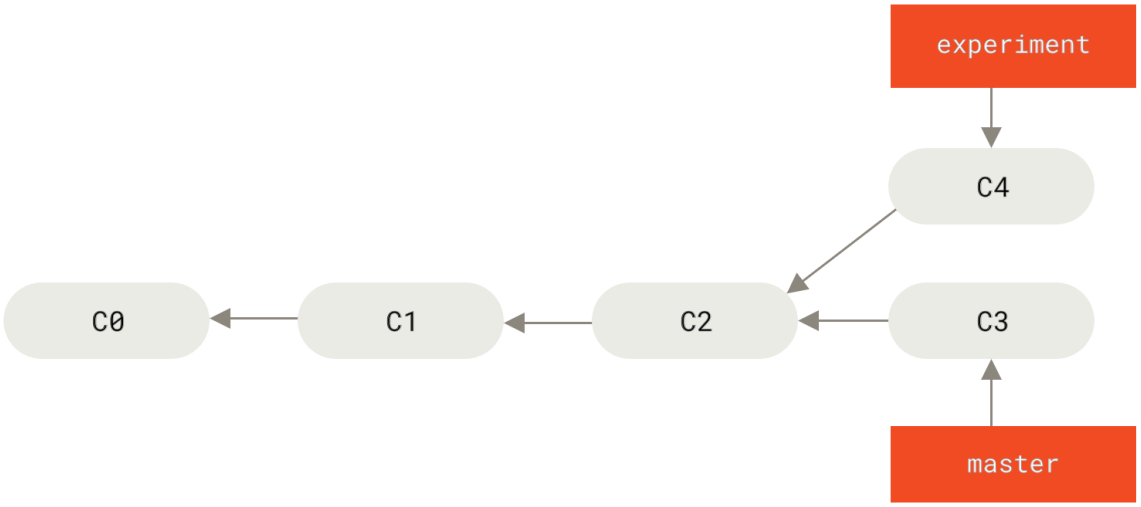
Rebasing

- 말 그대로 브랜치의 *base*를 변경하는 작업을 한다.

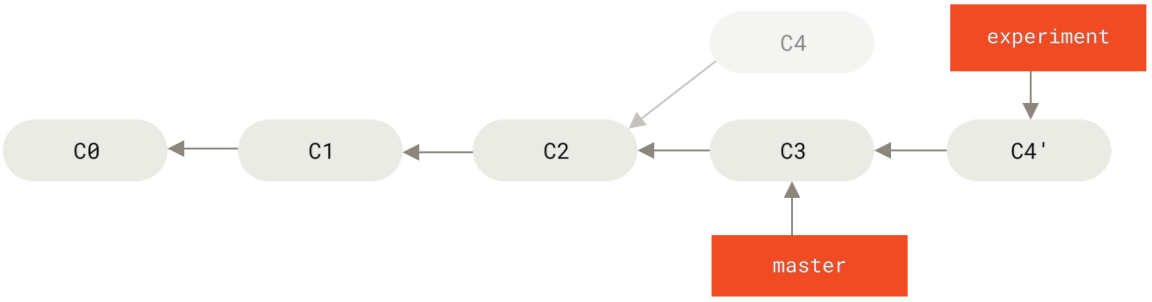
Example

1. `master`와 분기된 `experiment` 브랜치가 있다. `experiment` 브랜치에서 `C3`의 내용을 사용하고 싶을 때 `experiemt`에서 `$ git merge master`를 할 수도 있지만 `$ git rebase`

`master` 명령어를 사용할 수도 있다.



2. *rebase*를 할 경우 이렇게 기존의 `C4` 커밋과 동일한 변경사항의 커밋인 `C4'`이 `C3` 뒤에 생성된다. 커밋이 여러개일 경우 동일한 변경사항의 커밋 여러개가 생성된다.



Local Branch & Remote Branch

- 협업을 위해서 우리는 **원격 레포지토리**(ex: github, gitlab, ...)를 사용한다.
- **로컬 레포지토리**와 **원격 레포지토리**를 관리하기 위해 `$ git remote` 명령어를 사용한다.
- 로컬 레포지토리에는 **Remote tracking branch**와 **Tracking branch** 두 종류의 브랜치가 존재한다.

> 어떤 브랜치와 연결된 원격 브랜치가 존재할 경우 그 로컬 브랜치를 **Tracking branch**라 하고 해당 브랜치와 연결되어 있는 **Remote tracking branch**를 **Upstream branch**라고 한다.

`fetch`

- **Remote-tracking branch**들의 상태를 원격 브랜치들과 동기화시킨다.
- 내 로컬 레포지토리에 없는 변경사항들을 전부 가져온다.

`pull`

- 내 로컬 레포지토리에 있는 **Tracking branch**의 내용을 원격 브랜치와 동기화시킨다.
- `$ git fetch`를 한 후 `$ git merge`를 한 것과 동일한 결과를 보여준다.

`push`

- 원격 브랜치의 내용을 내 로컬 레포지토리에 있는 **Tracking branch**의 내용과 동기화시킨다.

> `git config --global push.autoSetupRemote true`를 설정하면 로컬 브랜치를 만든 후 **Upstream branch**가 없을 경우에 알아서 생성해준다.

`.gitconfig`

- git 환경설정을 해주는 파일이다.
- *system*, *global*, *local* 등 영역별로 다른 설정값을 설정할 수 있다.

`alias`

- 기존에 사용하던 명령어들을 간결하게 사용할 수 있게 해준다.

> 명령어를 치기 귀찮으면 모조리 마음대로 바꿔버리는 걸 추천한다

Example

- `$ git config --global alias.co checkout`
- `$ git config --global alias.logg "log --graph --decorate --oneline"`
- `$ git config --global alias.can "commit --amend --no-edit"`

Conditional Config

- `[includeIf]`를 `.gitconfig`에서 사용해 조건부로 설정값을 작동하게 해준다. 레포지토리별로 다른 설정값을 사용하고 싶을 때 사용하자.

`autoCorrect`

- 명령어를 잘못 입력했을 때 알아서 고쳐준다.
- `$ git config --global help.autoCorrect 20`를 등록하면 오타가 났을 때 2초 후에 예상되는 명령어를 실행시켜준다.