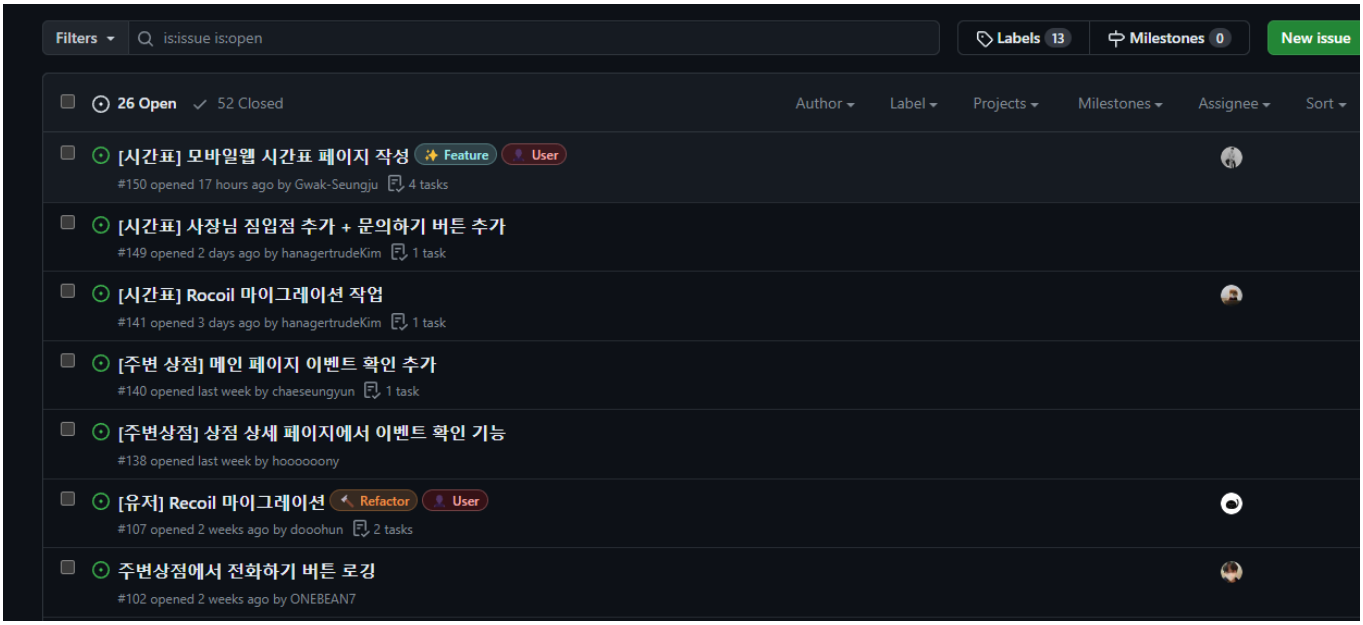
서론

최근 동아리 프로젝트와 관련해서 인원이 늘어나면서, develop에 대한 배포 이슈가 많아지고 있었습니다. 현재 동아리에서 서비스하고 있는 "코인-한기대 커뮤니티"(줄여서 코인)를 유지하고 있는데 최근 인원이 늘어나서 각 기능별로 담당 팀을 통해 업무를 분담하였습니다.
하지만 이러한 팀 변화에 있어서 git 관리에 문제가 발생하였습니다. 각 팀에서 작업을 하자마자 `develop` 브런치에 바로 머지하게 되어 `develop` 브런치에 다른 팀들의 작업까지 들어오게 되는 문제를 일으키고 있습니다.
그렇기에 이러한 문제를 피하고자 고민하게 되었습니다.

현 코인 프로젝트의 작업 방식

현재 코인 프로젝트에서는 github의 issue를 활용해서 기능과 작업 단위를 구별하고 있습니다.



이렇게 issue를 통해서 작업단위를 구성하고 작업자는 `develop` 브런치에서 `feat/#${이슈번호}` 를 넣어 작업하게 됩니다.

그리고 개발이 완료되었으면 Github에서 Pull Request를 develop 브런치에 넣음으로써 lint action과 test를 통과하고 다른 작업자들의 리뷰를 받으면 `develop` 에 올라오게됩니다.

문제점

위의 방식에서는 하나의 팀으로 행동하는 사람들에게는 작업을 알기 쉬운 것이 issue들에 대해서 모든 사람들이 필요한 것이기에 모든 이슈를 알 수 있습니다.
하지만 팀을 통해 작업을 관리하기 시작하면서 자신의 팀이 아닌 issue를 직관적으로 확인하기 어렵습니다. 또한 `develop` 브런치에는 머지를 통해서만 들어오기에 `Merged feat/#~` 으로만 표기되어있어 다시 issue가 어떤 것 에 관련되어있는지 찾기가 어려워집니다.

각 팀에서 기능을 배포하기 이전에 `develop` 브런치에 올려두는 것으로 여러 팀의 기능들이 올라와 수많은 conflict를 유발할 수 있습니다.

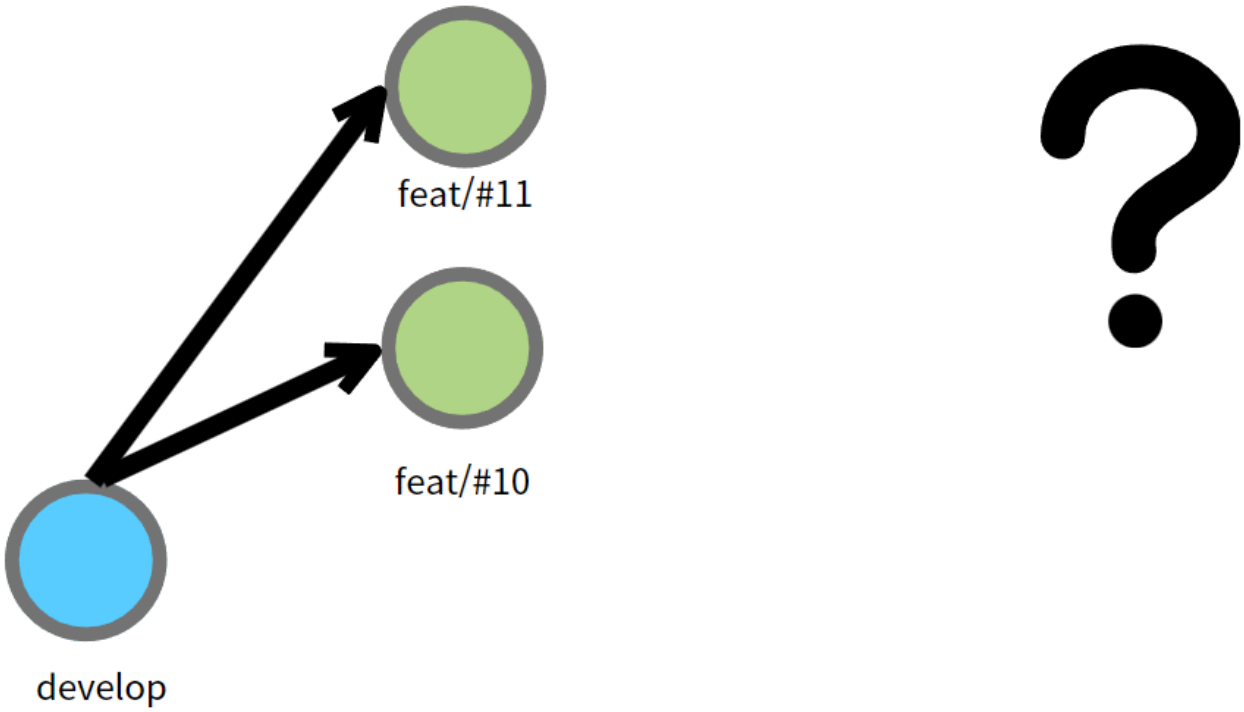
How to Solve

먼저 develop 자체를 통제할 필요가 있습니다.
현재 상황에서는 develop에 기능들로 머지하는 이유는 하나의 기능에 대해서 2번 이상 작업하게 되어 이를 합쳐져야하기 일어나는 현상입니다.
예를 들어 UI 작업, 기능 작업을 분할하는 등의 이유가 있을 것입니다.

그래서 해당 문제에서는 issue를 최대한으로 활용하는게 좋을 것 같다는 생각입니다. issue 하나를 작업 단위로 규모를 잘 선택해야할 것입니다.

예를 들어, A라는 팀에서 1번작업을 할당받았습니다. 그래서 A팀은 issue를 생성하여 issue 번호 `#10` 을 할당 받았습니다.

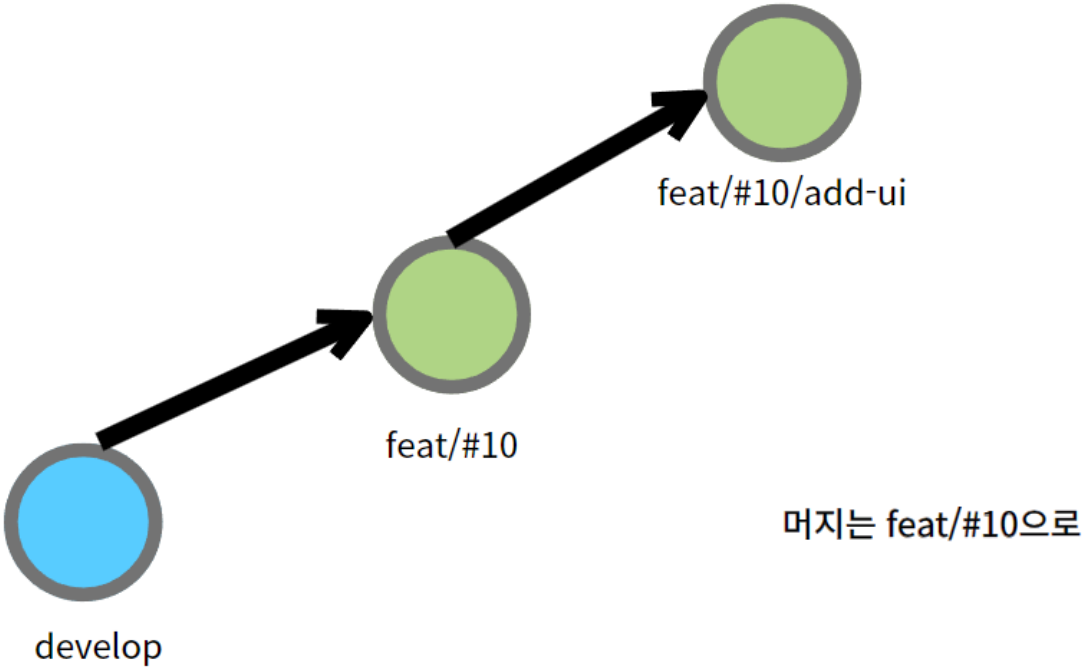
그렇다면 develop 브런치에서 feat/#10으로 1번 작업을 진행하였지만, 1번 작업을 2명 이상이서 작업해야하는 경우가 있을 경우에는 브런치를 하나 더 만들기 위해 issue를 하나 더 생성해야하는가?



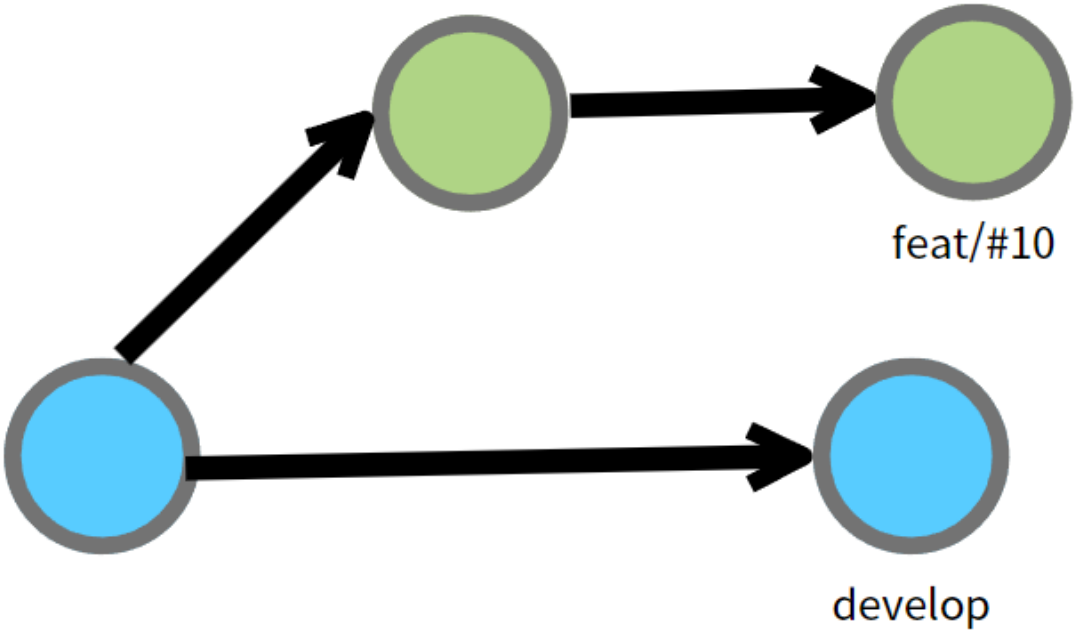
여기서 issue가 생성된다면 issue 2개를 develop에 머지하기까지의 과정은 어렵지 않지만 여러 팀이 이러한 상황에서 develop에 머지하게 된다면? 서로의 develop 환경이 변경이 많아지기 때문에 수많은 merge conflict를 맞이해야할 것입니다.

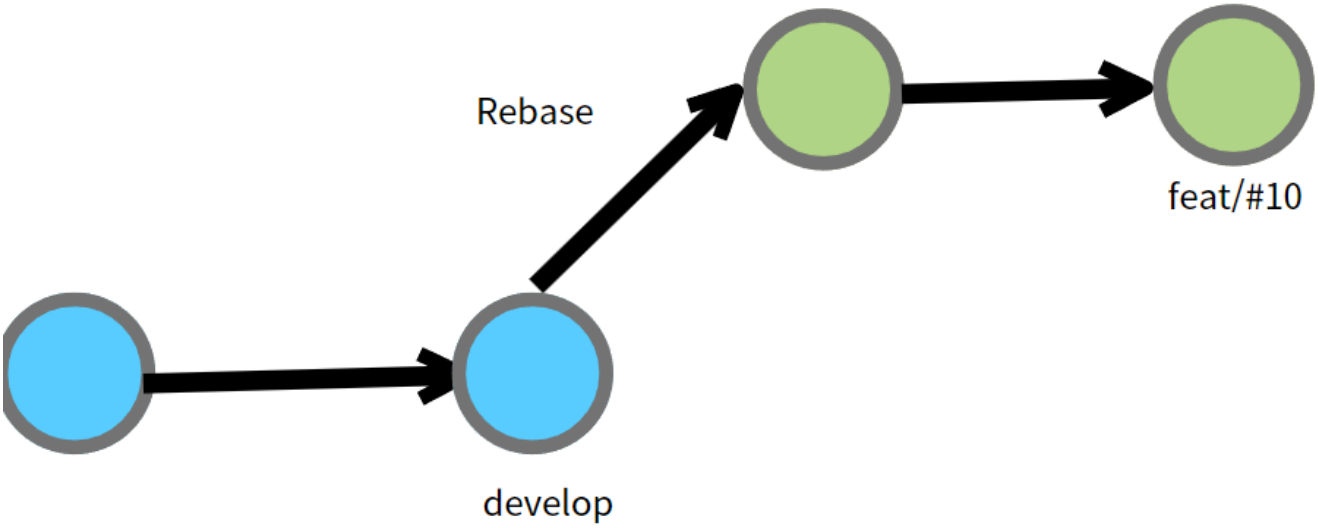
해당 문제를 막기 위해서는 issue의 작업 단위를 잘 설정해야하는 것입니다.
issue는 develop의 들어갈 기능 하나에 대해서만 할당하고 이후 분할되는 작업이나 여러번 붙여야하는 경우에는 feat/#10/add-ui 이런 식으로 해서 기능에 대한 붙이는 방식을 최소한으로 하는

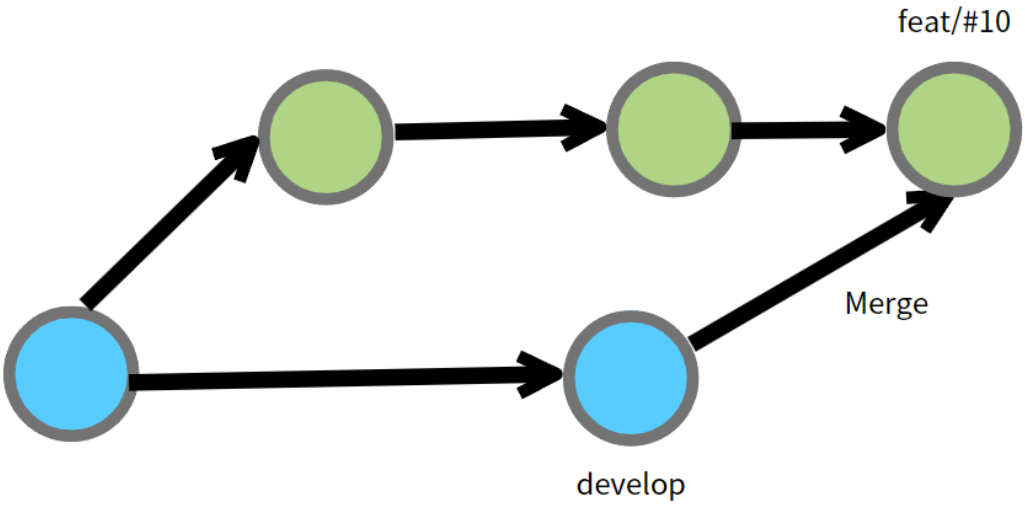
것이 서로의 작업 범위를 침범하지 않게 될 것입니다.



만약 develop이 업데이트 되었을 경우에는? develop에 feat/#10을 rebase하거나 develop을 머지해서 작업내용을 가져와서 최신화를 하면됩니다.







이를 통해 각 팀들은 여전히 issue를 통해 작업을 통제 받을 수 있지만 실제 develop에 들어가야 하는 작업의 경우에는 바로 배포될 내용들만 develop에 올리고 전부 올라간 이후 QA를 거쳐서 배포가 승인이 되면 그때 Main으로 머지하는 것으로 작업의 딜레이 시간을 거쳐야만 합니다.

> 전체적인 작업의 흐름 정리
> 작업 할당 -> issue 생성 -> feat/#issue 브런치 생성 -> 기능 개발 -> PR을 통한 리뷰 및 Lint 검사 -> 리뷰 완료 -> (배포할 기능 정리 및 일정 관리) -> develop 머지 -> QA -> main 머지

웹에서 유연하게 배포할 수 있다는 장점 부분에 대해서는 main 부분에서 hotfix 브런치를 생성해서 머지하는 방식으로 대응하면 좋을 것 같다.

추가적인 내용

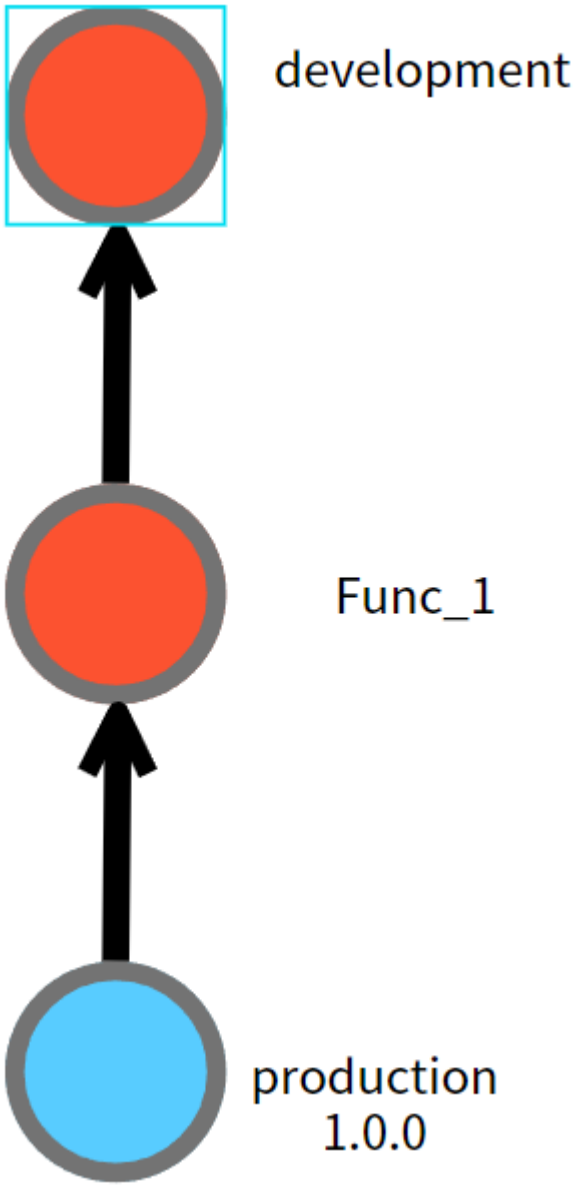
기존에 내가 생각했던 main 브런치에 develop 브런치를 rebase한 형태를 제안하려했지만 많은 부분에서 허점이 있어 위의 방식을 제안하게 되었습니다.
하지만 제가 말했던 부분에 궁금하신 분들이 있을 수도 있으니 git 한 줄 관리에 대한 내용을 좀 더 적어보겠습니다.

먼저, git을 한 줄로 본다는 것은 브런치 하나에서 버저닝을 한다는 것입니다.
먼저 브런치 종류를 설명드리겠습니다.

> production: 실제 배포 브런치
> development: 개발 환경 브런치
> feature: 기능 개발 브런치

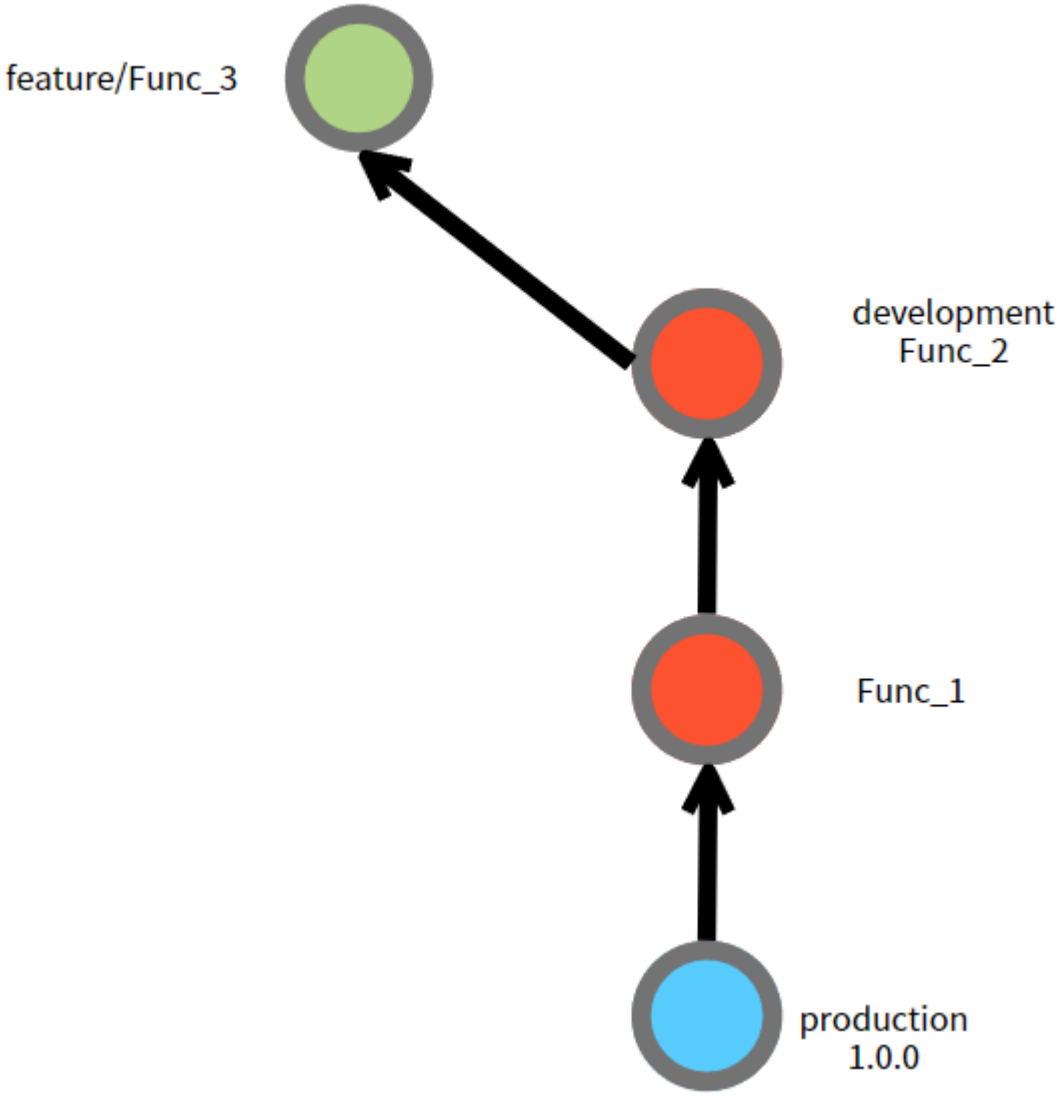
1. production에 develop을 rebase한 상태를 유지합니다.

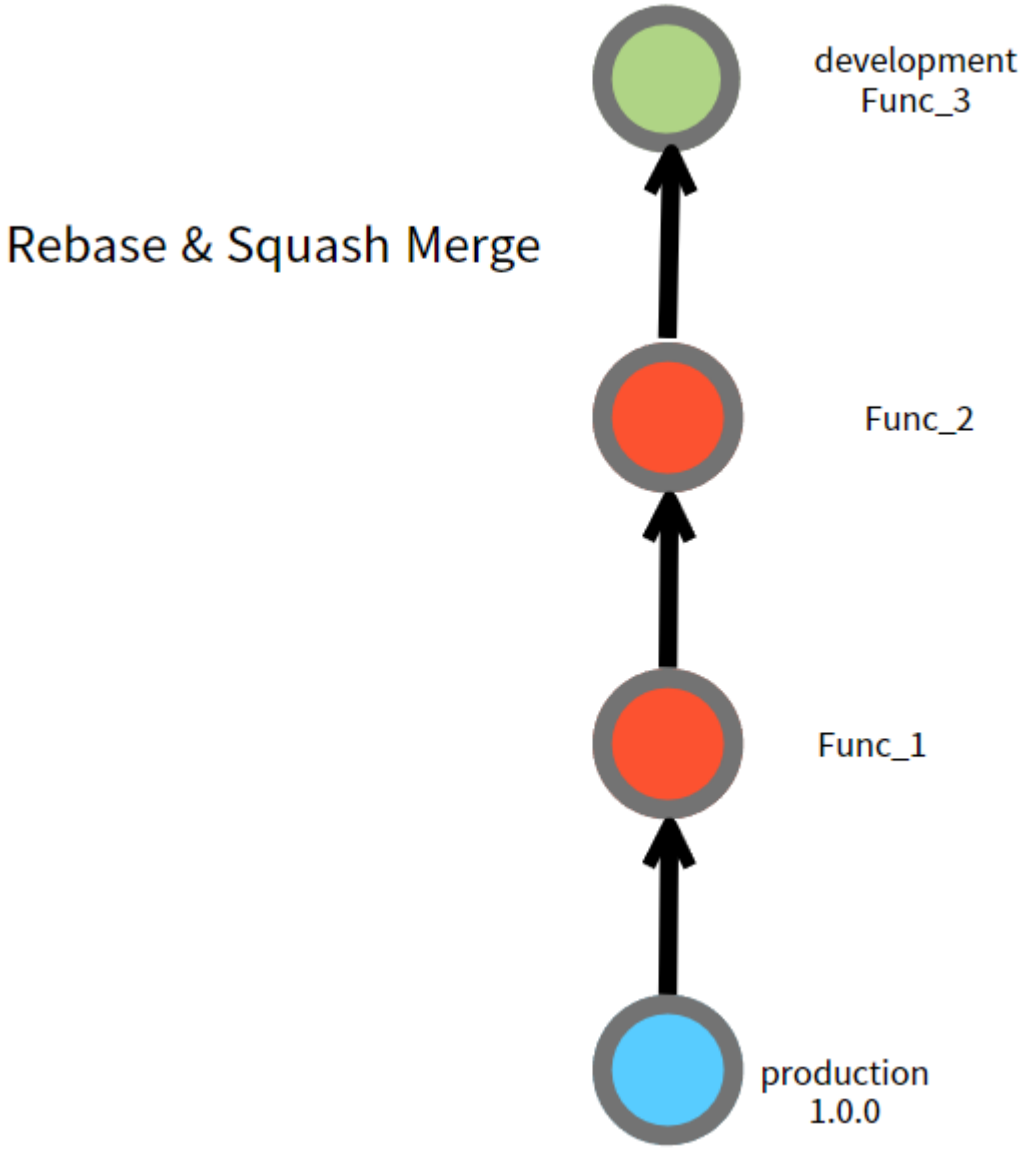
해당 내용은 항상 develop의 작업 환경은 production 환경을 갖고 있다는 것을 의미합니다.



2. feature는 development에 Rebase 이후 Squash Merge를 진행한다.

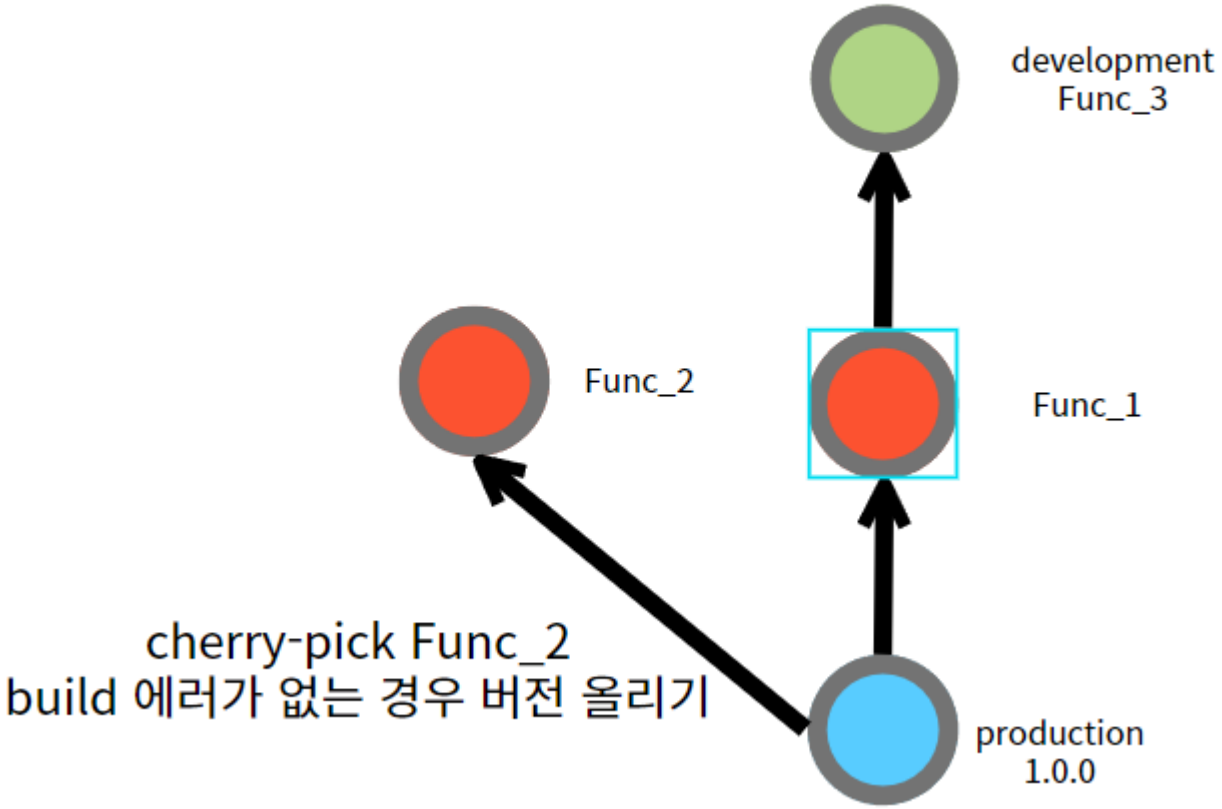
feature 부분은 development 부분에서 출발되어 항상 위와 같이 develop의 HEAD에 rebase를 한 상태로 Squash Merge를 하게 된다면 하나의 작업으로 축약되어 development 부분에는 작업단위가 한 개씩 올라가게 될 것입니다.

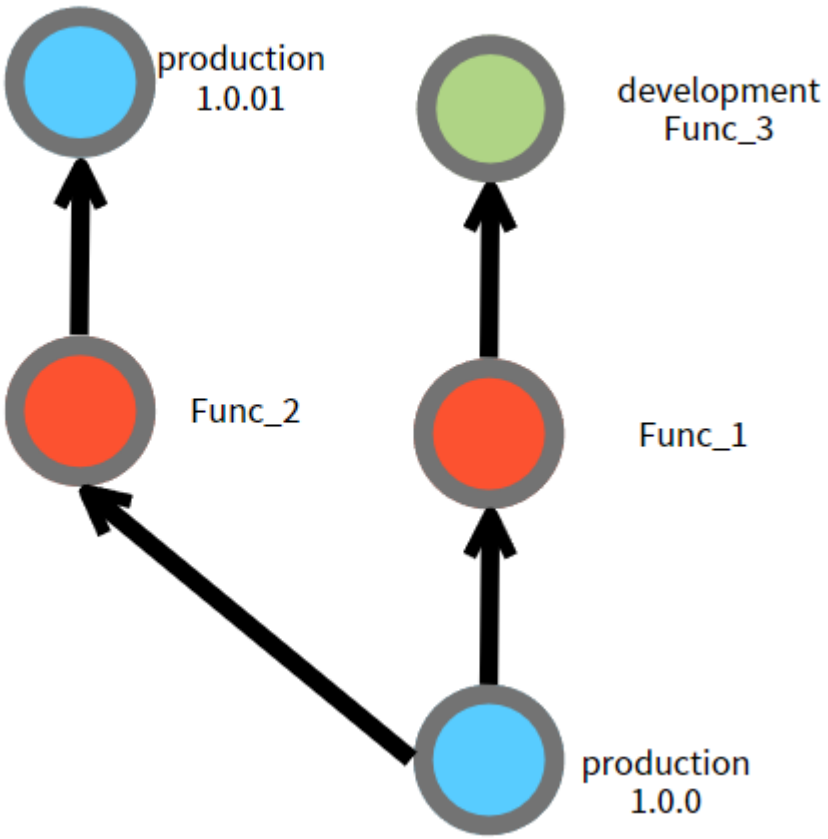

feature/Func_3
development
Func_2
Func_1
production
1.0.0



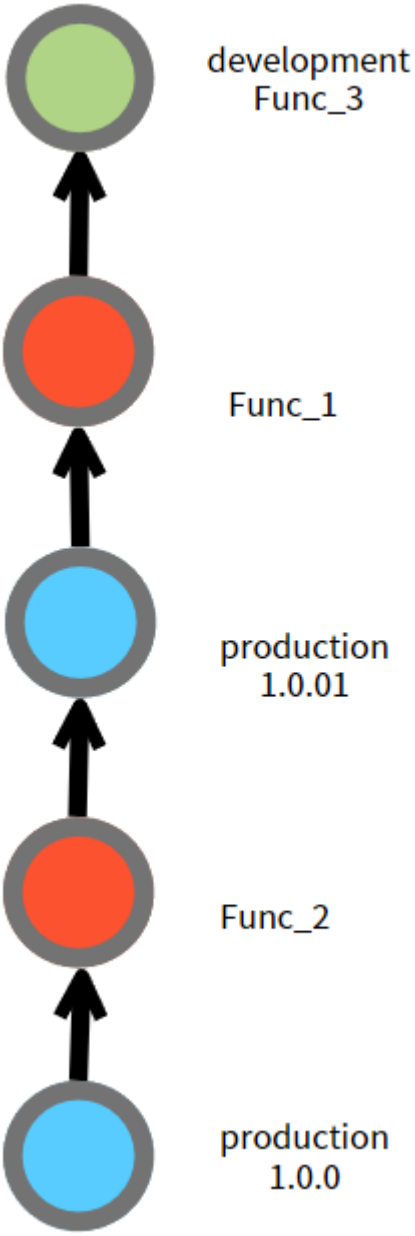
3. production을 cherry-pick을 통한 배포와 재정렬

이렇게 올라가게된 기능들에 대해서 Func_2에 대해서 production 1.0.01을 배포하고 싶다면은 production 브런치에서 cherry-pick을 이행하면 됩니다.


development
Func_3
Func_2
Func_1
cherry-pick Func_2
build 에러가 없는 경우 버전 올리기
production
1.0.0


production
1.0.01
development
Func_3
Func_2
Func_1
production
1.0.0

이후에 그렇다면 브런치가 2줄로 늘어났기 때문에 다시 재정렬이 필요합니다. development를 production에 rebase 하면 아래와 같이 브런치가 1줄로 정렬될 것입니다.



이렇게 브런치를 하나로만 사용한다면 어떤 이점이 있나요?

직관적으로 현재 devleopment 부분에 어떤 기능이 올라와있는지 해석이 쉽습니다. Func_2라는 제목 하나만 보고 손쉽게 기능을 확인할 수 있습니다.

또한 git log자체가 깔끔하게 남기 때문 commit id를 잘못 선택하는 경우를 최소화할 수 있습니다.

모든 기능은 development 위와 Production을 base로 두고 있기에 실제 환경과의 차이가 벌어지는 현상을 최소화할 수 있습니다.

배포 시 cherry-pick을 통해 작업들을 배포하기 때문에, develop에 머지 순서를 고려하는 걱정을 줄여줄 수 있습니다.

정리하면 아래와 같습니다.

> 로그 자체가 한줄로 남게 되어 commitID나 분기점자체가 거의 없어 기능을 직관적으로 확인할 수 있다.
> 코드 베이스를 development와 production으로 두고 있기에 conflict 날 확률이 적다.
> cherry-pick을 통한 배포이기에 development에 순서 상관없이 자유롭게 넣을 수 있다.

그렇다면 이런 방식을 하지 않고 위와 같이 issue로 해결하는 방식을 제안한 것은 허점이 어느정도 존재했기 때문입니다.
먼저 배포 시 cherry-pick을 하는 과정에서 휴먼에러로 배포되지 말아야할 것을 push 한 경우 Revert 명령어에 제한이 걸린다는 점입니다. 만약 실수로 Revert를 잘못하게 된 경우에는 최악의 경우 작업 자체가 날아가게되어 복구가 어려울 수 있습니다.

또한 production을 관리해줄 수 있는 관리자가 필요합니다. git cherry-pick을 통해 한 명이 작업에 대한 배포를 책임져야합니다. 이러한 과정에서 혹여나 발생하는 conflict를 해결하기 위해서는 관리자가 전반적인 작업에 대한 이해가 필요합니다.

git을 관리하고 작업을 일정부분 수정해줄 수 있는 책임자가 단일로 있어야하기에 앞으로의 git 교육이나 작업 교육 비용등이 생길 수 있습니다.

또한 cherry-pick 과 rebase의 과정에서는 force-push들이 강제되는 경우가 있기 때문에 현재 github에 세팅된 lint Action들이 main에 대해서는 무효화 될 수 있습니다.

> 정리
> 휴먼에러와 같은 cherry-pick 과정에서의 배포 문제 시 Revert 불가, 잘못될 경우 작업이 손실됨
> 기능 배포 cherry-pick 사용 시 conflict를 해결 및 기능 리딩이 가능한 총책임자의 존재가 필수적
> force-push를 통한 기존 lint Action이 의미를 상실할 수도 있음