1-5 Git 협업 워크플로우

5.1 분산 환경에서의 Git - 분산 환경에서의 워크플로

앞 장에서 다른 개발자와 코드를 공유하는 리모트 저장소를 만드는 법을 배웠다. 로컬에서 작업하는 데 필요한 기본적인 명령어에는 어느 정도 익숙해졌다. 이제는 분산 환경에서 Git이 제공하는 기능을 어떻게 효율적으로 사용할지를 배운다.

이번 장에서는 분산 환경에서 Git을 어떻게 사용할 수 있을지 살펴본다. 프로젝트 기여자 입장과 여러 수정사항을 취합하는 관리자 입장에서 두루 살펴본다. 즉, 프로젝트 기여자 또는 관리자로서 작업물을 프로젝트에 어떻게 포함시킬지와 수 많은 개발자가 수행한 일을 취합하고 프로젝트를 운영하는 방법을 배운다.

분산 환경에서의 워크플로

중앙집중형 버전 관리 시스템과는 달리 Git은 분산형이다. Git은 구조가 매우 유연하기 때문에 여러 개발자가 함께 작업하는 방식을 더 다양하게 구성할 수 있다. 중앙집중형 버전 관리 시스템에서 각 개발자는 중앙 저장소를 중심으로 하는 한 노드일 뿐이다. 하지만, Git에서는 각 개발자의 저장소가 하나의 노드이기도 하고 중앙 저장소 같은 역할도 할 수 있다. 즉, 모든 개발자는 다른 개발자의 저장소에 일한 내용을 전송하거나, 다른 개발자들이 참여할 수 있도록 자신이 운영하는 저장소 위치를 공개할 수도 있다. 이런 특징은 프로젝트나 팀이 코드를 운영할 때 다양한 워크플로를 만들 수 있도록 해준다. 이런 유연성을 살려 저장소를 운영하는 몇 가지 방식을 소개한다. 각 방식의 장단점을 살펴보고 그 방식 중 하나를 고르거나 여러 가지를 적절히 섞어 쓰면 된다.

중앙집중식 워크플로

중앙집중식 시스템에서는 보통 중앙집중식 협업 모델이라는 한 가지 방식밖에 없다. 중앙 저장소는 딱 하나 있고 변경 사항은 모두 이 중앙 저장소에 집중된다. 개발자는 이 중앙 저장소를 중심으로 작업한다

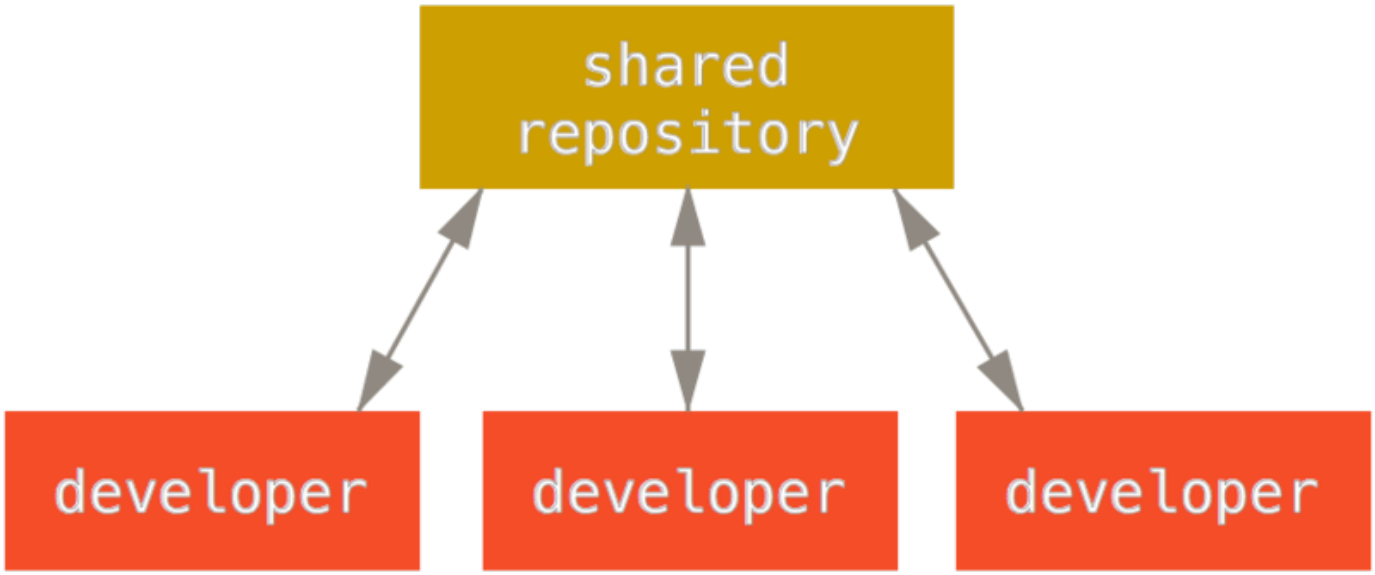


그림 54. 중앙집중식 워크플로.

중앙집중식에서 개발자 두 명이 중앙저장소를 Clone 하고 각자 수정하는 상황을 생각해보자. 한 개발자가 자신이 한 일을 커밋하고 나서 아무 문제 없이 서버에 Push 한다. 그러면 다른 개발자는 자신의 일을 커밋하고 Push 하기 전에 첫 번째 개발자가 한 일을 먼저 Merge 해야 한다. Merge를 해야 첫 번째 개발자가 작업한 내용을 덮어쓰지 않는다. 이런 개념은 Subversion과 같은 중앙집중식 버전 관리 시스템에서 사용하는 방식이고 Git에서도 당연히 이런 워크플로를 사용할 수 있다.

팀이 작거나 이미 중앙집중식에 적응한 상황이라면 이 워크플로에 따라 Git을 도입하여 사용할 수 있다. 중앙 저장소를 하나 만들고 개발자 모두에게 Push 권한을 부여한다. 모두에게 Push 권한을 부여해도 Git은 한 개발자가 다른 개발자의 작업 내용을 덮어쓰도록 허용하지 않는다. John과 Jessica가 동시에 같은 부분을 수정하는 상황을 생각해보자. John이 먼저 작업을 끝내고 수정한 내용을 서버로 Push 한다. Jessica도 마찬가지로 작업을 끝내고 수정한 내용을 서버로 Push 하려 하지만 서버가 바로 받아주지 않는다. 서버에는 John이 수정한 내용이 추가되었기 때문에 Push 하기 전에 Fetch로 받아서 Merge 한 후 Push 할 수 있다. 이런 개념은 개발자에게 익숙해서 거부감 없이 도입할 수 있다.

작은 팀만 이렇게 일할 수 있는 것이 아니다. Git이 제공하는 브랜치 관리 모델을 사용하면 수백명의 개발자가 한 프로젝트 안에서 다양한 브랜치를 만들어서 함께 작업하는 것도 쉽다.

Integration-Manager 워크플로

Git을 사용하면 리모트 저장소를 여러 개 운영할 수 있다. 다른 개발자는 읽기만 가능하고 자신은 쓰기도 가능한 공개 저장소를 만드는 워크플로도 된다. 이 Workflow에는 보통 프로젝트를 대표하는 공식 저장소가 있다. 기여자는 우선 공식 저장소를 하나 Clone 하고 수정하고 나서 자신의 저장소에 Push 한

다. 그 다음에 프로젝트 Integration-Manager에게 새 저장소에서 Pull 하라고 요청한다. 그러면 그 Integration-Manager는 기여자의 저장소를 리모트 저장소로 등록하고, 로컬에서 기여물을 테스트하고, 프로젝트 메인 브랜치에 Merge 하고, 그 내용을 다시 프로젝트 메인 저장소에 Push 한다. 이런 과정은 아래와 같다(Integration-manager workflow.).

1. 프로젝트 Integration-Manager는 프로젝트 메인 저장소에 Push를 한다.
2. 프로젝트 기여자는 메인 저장소를 Clone 하고 수정한다.
3. 기여자는 자신의 저장소에 Push 하고 Integration-Manager가 접근할 수 있도록 공개해 놓는다.
4. 기여자는 Integration-Manager에게 변경사항을 적용해 줄 것을 이메일로 요청한다.
5. Integration-Manager는 기여자의 저장소를 리모트 저장소로 등록하고 수정사항을 Merge 하여 테스트한다.
6. Integration-Manager는 Merge 한 사항을 메인 저장소에 Push 한다.

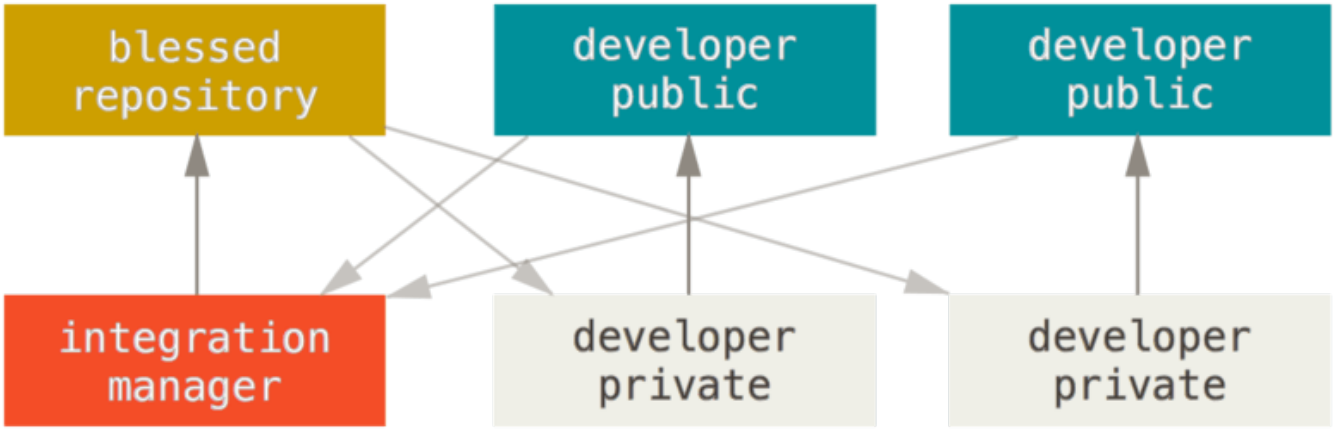


그림 55. Integration-manager workflow.

이 방식은 GitHub나 GitLab 같은 Hub 사이트를 통해 주로 사용하는 방식이다. 프로젝트를 Fork 하고 수정사항을 반영하여 다시 모두에게 공개하기 좋은 구조로 돼 있다. 이 방식의 장점은 기여자와 Integration-Manager가 각자의 사정에 맞춰 프로젝트를 유지할 수 있다는 점이다. 기여자는 자신의 저장소와 브랜치에서 수정 작업을 계속해 나갈 수 있고 수정사항이 프로젝트에 반영되도록 기다릴 필요가 없다. 관리자는 여유를 가지고 기여자가 Push 해 놓은 커밋을 적절한 시점에 Merge 한다.

Dictator and Lieutenants 워크플로

이 방식은 저장소를 여러개 운영하는 방식을 변형한 구조이다. 보통 수백 명의 개발자가 참여하는 아주 큰 프로젝트를 운영할 때 이 방식을 사용한다. Linux 커널 프로젝트가 대표적이다. 여러 명의 Integration-Manager가 저장소에서 자신이 맡은 부분만을 담당하는데 이들을 *Lieutenants* 라고 부른다. 모든 Lieutenant는 최종 관리자 아래에 있으며 이 최종 관리자를 *Benevolent Dictator* 라고 부른다. Benevolent Dictator는 Lieutenant의 저장소를 가져와 공식 저장소에 Push 하고 모든 프로젝트 참여자는 이 공식 저장소에서 반드시 Pull 해야 한다. 이러한 워크플로는 아래와 같다(Benevolent dictator workflow.).

1. 개발자는 코드를 수정하고 `main` 브랜치를 기준으로 자신의 토픽 브랜치를 Rebase 한다. 여기서 `main` 브랜치란 공식 저장소의 브랜치를 말한다.
2. Lieutenant들은 개발자들의 수정사항을 자신이 관리하는 `main` 브랜치에 Merge 한다.
3. Dictator는 Lieutenant의 `main` 브랜치를 자신의 `main` 브랜치로 Merge 한다.
4. Dictator는 자신의 `main` 브랜치를 Push 하며 다른 모든 개발자는 Dictator의 `main` 브랜치를 기준으로 Rebase 한다.

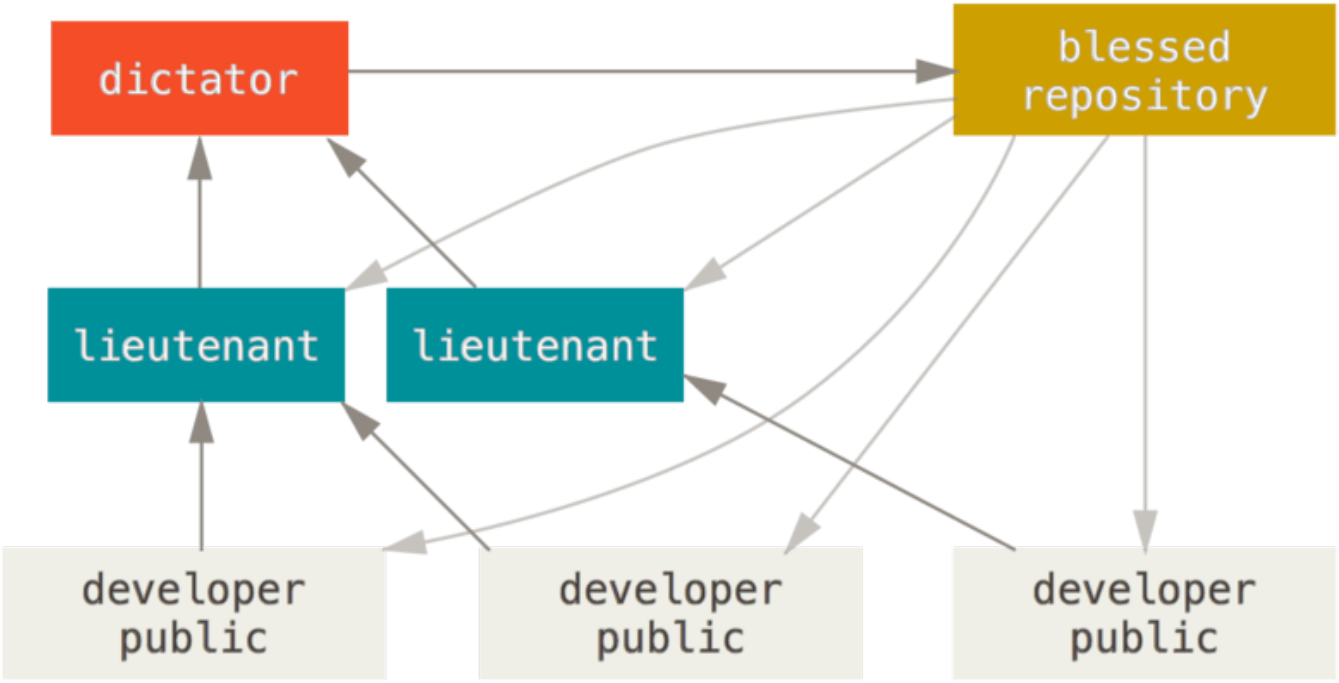


그림 56. Benevolent dictator workflow.

이 방식이 일반적이지 않지만 깊은 계층 구조를 가지는 환경이나 규모가 큰 프로젝트에서는 매우 쓸모 있다. 프로젝트 리더가 모든 코드를 통합하기 전에 코드를 부분부분 통합하도록 여러 명의 Lieutenant에게 위임한다.

워크플로 요약

이 세 가지 워크플로가 Git 같은 분산 버전 관리 시스템에서 주로 사용하는 것들이다. 사실 이런 워크플로뿐만 아니라 다양한 변종 워크플로가 실제로 사용된다. 어떤 방식을 선택하고 혹은 조합해야 하는 지 살짝 감이 잡힐 것이다. 앞으로 몇 가지 구체적 사례를 들고 우리가 다양한 환경에서 각 역할을 어떻게 수행하는지 살펴본다. 이어지는 내용에서 프로젝트에 참여하고 기여할 때 작업 패턴이 어떠한지 몇 가지 살펴보기로 한다.

Git Flow 실습

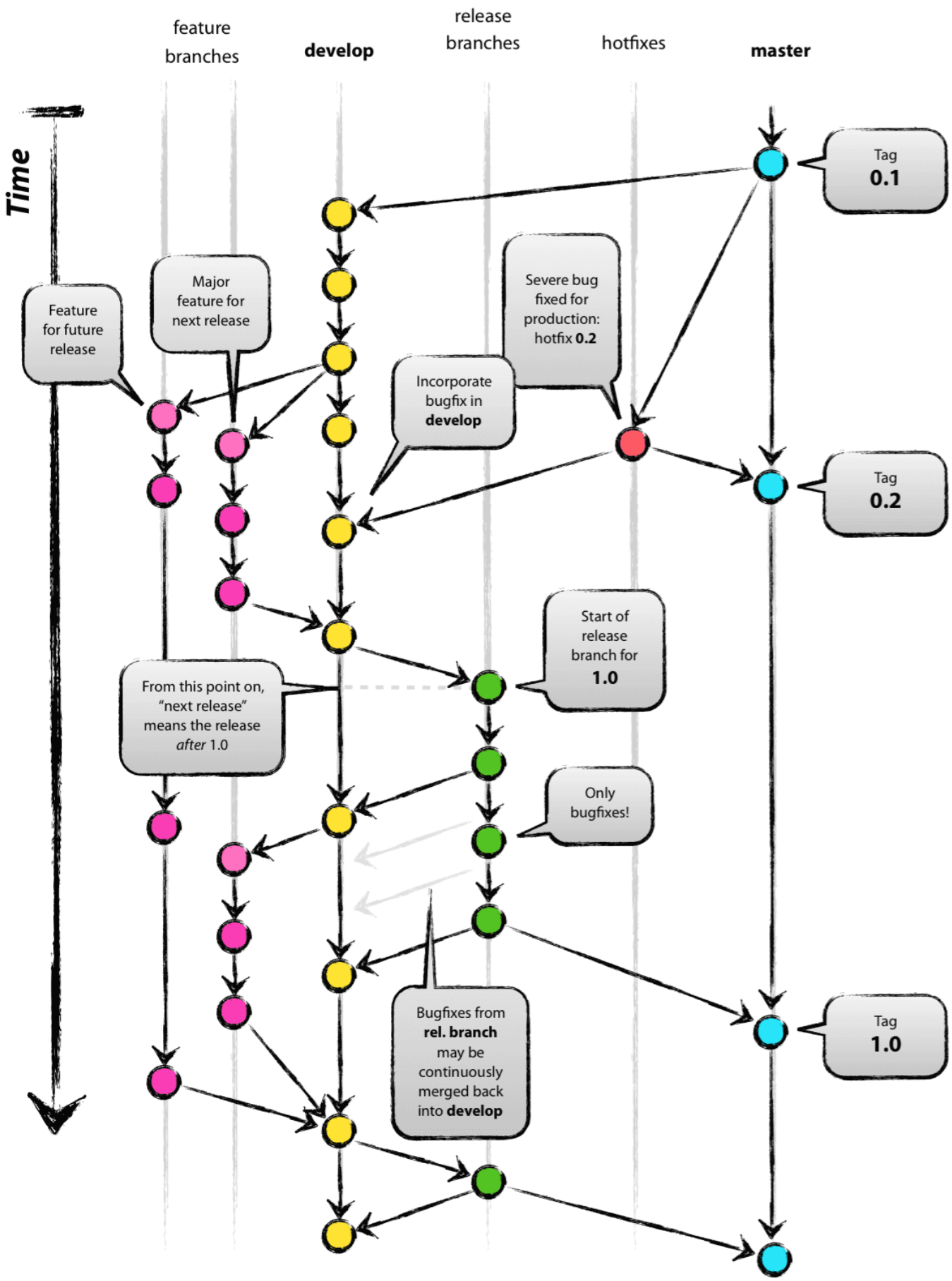
Git-flow는 Git이 새롭게 활성화되기 시작하는 10년전 쯤에 **Vincent Driessen** 이라는 사람의 블로그 글에 의해 널리 퍼지기 시작했고 현재는 Git으로 개발할 때 거의 표준과 같이 사용되는 방법론입니다.

말하자면 Git-flow는 기능이 아니고 서로간의 약속인 방법론이라는 점입니다. Vincent Driessen도 언급했듯이 Git-flow가 완벽한 방법론은 아니고 각자 개발 환경에 따라 수정하고 변형해서 사용하라고 언급했습니다.

Git-flow는 총 5가지의 브랜치를 사용해서 운영을 합니다.

- **master** : 기준이 되는 브랜치로 제품을 배포하는 브랜치 입니다.
- **develop** : 개발 브랜치로 개발자들이 이 브랜치를 기준으로 각자 작업한 기능들을 합(Merge)칩니다.
- **feature** : 단위 기능을 개발하는 브랜치로 기능 개발이 완료되면 develop 브랜치에 합칩니다.
- **release** : 배포를 위해 master 브랜치로 보내기 전에 먼저 QA(품질검사)를 하기위한 브랜치 입니다.
- **hotfix** : master 브랜치로 배포를 했는데 버그가 생겼을 때 긴급 수정하는 브랜치 입니다.

> *master와 develop가 중요한 **매인 브랜치**이고 나머지는 필요에 의해서 운영하는 브랜치라고 보시면 됩니다.*


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

이미지. https://nvie.com/posts/a-successful-git-branching-model/

실습

실습 진행전에 day1 repository 를 초기화 합니다.

```
#  tag
$ git tag -l | xargs -n 1 git push --delete origin
#  tag
$ git tag | xargs git tag -d

#  branch  (main )
$ git branch | grep -v "main" | xargs git push origin -d
#  branch  (main )
$ git branch | grep -v "main" | xargs git branch -D
```

실습을 진행합니다.

Sin1. develop 개발 브랜치가 최초로 생성됩니다. 초기에 개발자들이 개발하여 commit 을 합니다.

```
# main  develop   .
$ git checkout develop

# C1, C2  .
$ echo 'text' > C1
$ git add .
$ git commit -m 'C1'

$ echo 'text' > C2
$ git add .
$ git commit -m 'C2'
```

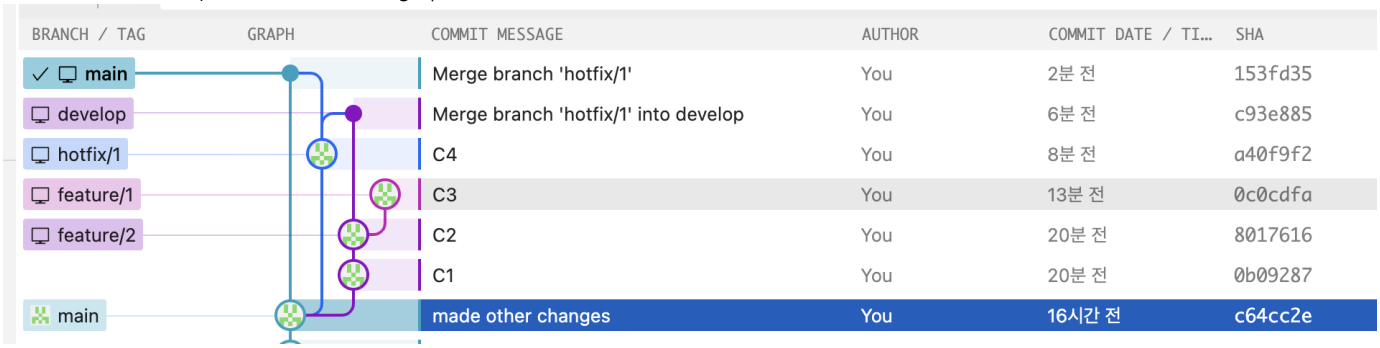
Sin2. 동시에 2가지 기능이 개발되어야 합니다. feature/1 은 작은 기능이고, feature/2 는 큰 기능입니다. develop 으로부터 2개의 브랜치가 파생됩니다. 우선적으로 feature/1 을 담당한 개발자가 commit 합니다.

```
# develop  feature/1, feature/2  .
$ git checkout develop
$ git branch feature/1
$ git branch feature/2

# feature1  C3 commit  .
$ git checkout feature/1
$ echo 'text' > C3
$ git add .
$ git commit -m 'C3'
```

Sin3. feature1 이 아직 마무리 되기 전에, 운영 어플리케이션에서 긴급 수정 사항이 발생했습니다. main 브랜치로부터 hotfix 브랜치를 생성하여 긴급 수정을 반영한 다음, main, develop 브랜치 양쪽에 병합합니다.

```
# main  hotfix/1  .
$ git checkout main
$ git checkout -b hotfix/1 (-b      .)

#   C4  commit .
$ echo 'text' > C4
$ git add .
$ git commit -m 'C4'

# develop, main  .
#  --no-ff  , Fast-Foward   3way merge
   .
$ git checkout develop
$ git merge hotfix/1 --no-ff
$ git checkout main
$ git merge hotfix/1 --no-ff
```

여기까지 진행 했을 때, GitLens 의 commit graph 는 아래 모습처럼 나옵니다.



Sin4. feature/1 개발이 마무리 되고 develop 으로 무사히 병합됩니다.

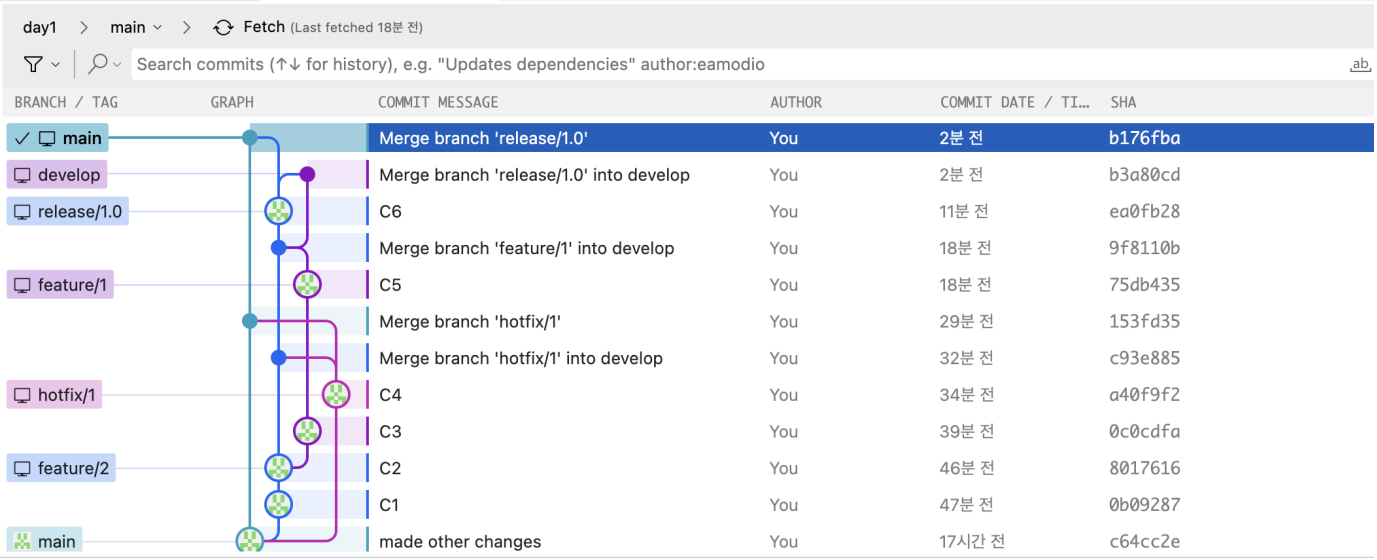
```
# feature/1   .
$ git checkout feature/1
$ echo 'text' > C5
$ git add .
$ git commit -m 'C5'

# feature/1  develop  .
$ git checkout develop
$ git merge feature/1 --no-ff
```

Sin5. 제품 담당자는 feature/1 기능까지 이번 운영 배포로 내보내기로 합니다. QA 를 위해서 릴리즈 1.0 을 구성합니다. QA 를 하는 도중 버그가 발견되었습니다!! QA 연락을 받은 개발자가 릴리즈 브랜치에서 버그를 수정합니다..

QA 가 통과된 이후에는 main, develop 브랜치 양쪽에 병합합니다. (그리고 main 브랜치는 운영으로 나갔습니다.)

```
# develop  release/1.0  .
$ git checkout develop
$ git checkout -b release/1.0

# release/1.0  QA ...

# QA    .
$ echo 'text' > C6
$ git add .
$ git commit -m 'C6'

# QA  . develop, main  .
$ git checkout develop
$ git merge release/1.0 --no-ff
$ git checkout main
$ git merge release/1.0 --no-ff

# main   .
```

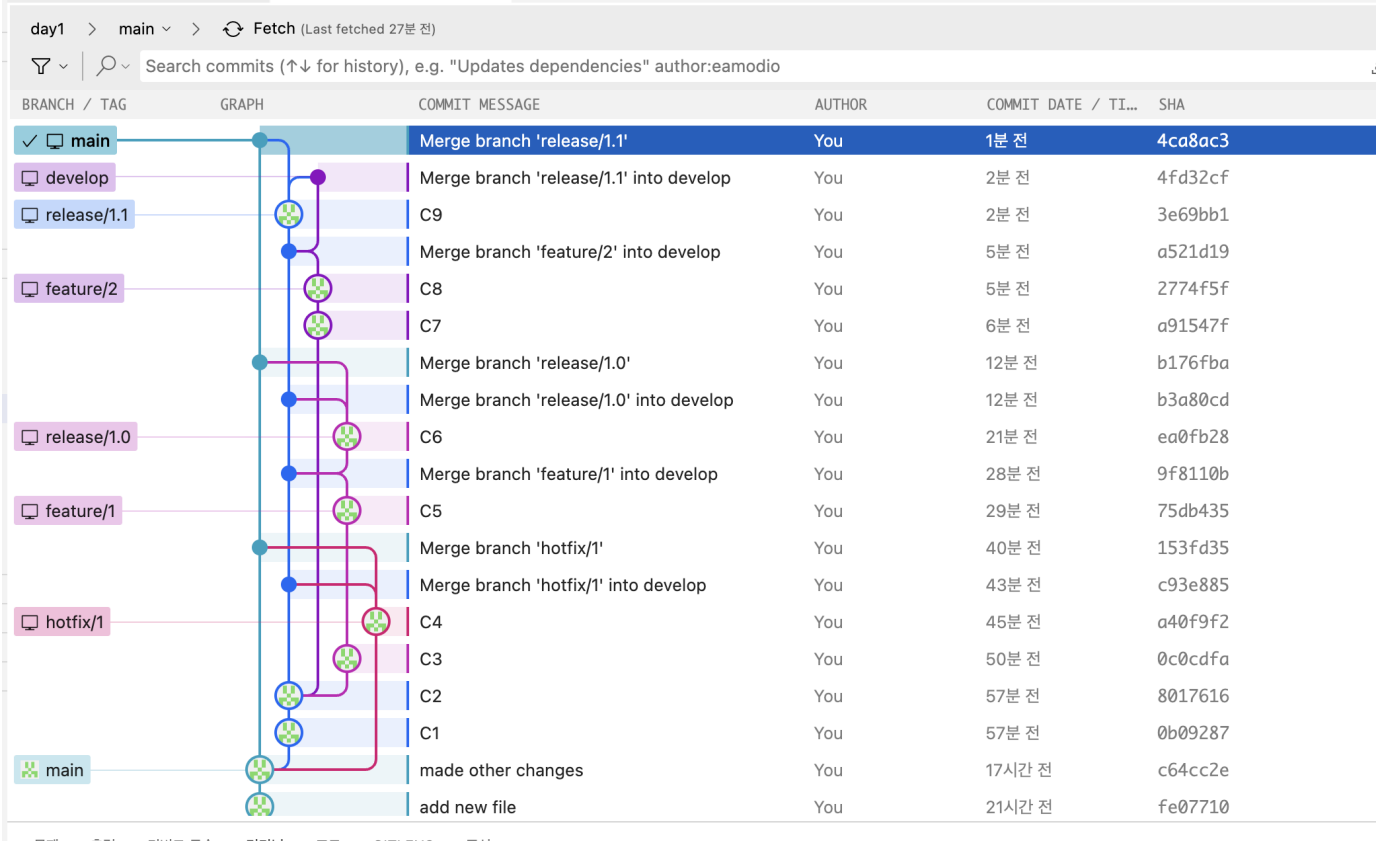
여기까지 진행 했을 때, GitLens 의 commit graph 는 아래 모습처럼 나옵니다.



Sin6. 오래 걸렸던 feature/2 개발도 마무리 되었습니다. QA 를 위해서 릴리즈 1.1 을 구성했고, 마찬가지로 버그 수정 후 운영으로 배포 되었습니다.

```
# feature/2  .
$ git checkout feature/2
$ echo 'text' > C7
$ git add .
$ git commit -m 'C7'
$ echo 'text' > C8
$ git add .
$ git commit -m 'C8'

# develop  .
$ git checkout develop
$ git merge feature/2 --no-ff

# QA    release/1.1  .
$ git checkout develop
$ git checkout -b release/1.1

# release/1.0  QA ...

# QA    .
$ echo 'text' > C9
$ git add .
$ git commit -m 'C9'

# QA ...

# QA  . develop, main  .
$ git checkout develop
$ git merge release/1.1 --no-ff
$ git checkout main
$ git merge release/1.1 --no-ff

# main   .
```

GitLens 의 commit graph 의 최종 모습입니다. 이로써 Git Flow 의 development cycle 을 모두 수행하여 보았습니다.


day1
main
Fetch (Last fetched 27분 전)
Search commits (↑↓ for history), e.g. "Updates dependencies" author:eamodio
BRANCH / TAG
GRAPH
COMMIT MESSAGE
AUTHOR
COMMIT DATE / TI…
SHA
main
Merge branch 'release/1.1'
You
1분 전
4ca8ac3
develop
Merge branch 'release/1.1' into develop
You
2분 전
4fd32cf
release/1.1
C9
You
2분 전
3e69bb1
Merge branch 'feature/2' into develop
You
5분 전
a521d19
feature/2
C8
You
5분 전
2774f5f
C7
You
6분 전
a91547f
Merge branch 'release/1.0'
You
12분 전
b176fba
Merge branch 'release/1.0' into develop
You
12분 전
b3a80cd
release/1.0
C6
You
21분 전
ea0fb28
Merge branch 'feature/1' into develop
You
28분 전
9f8110b
feature/1
C5
You
29분 전
75db435
Merge branch 'hotfix/1'
You
40분 전
153fd35
Merge branch 'hotfix/1' into develop
You
43분 전
c93e885
hotfix/1
C4
You
45분 전
a40f9f2
C3
You
50분 전
0c0cdfa
C2
You
57분 전
8017616
C1
You
57분 전
0b09287
main
made other changes
You
17시간 전
c64cc2e
add new file
You
21시간 전
fe07710