6. Git Flow 협업하기

1. Git Flow 란?

분산 환경에서의 워크플로우

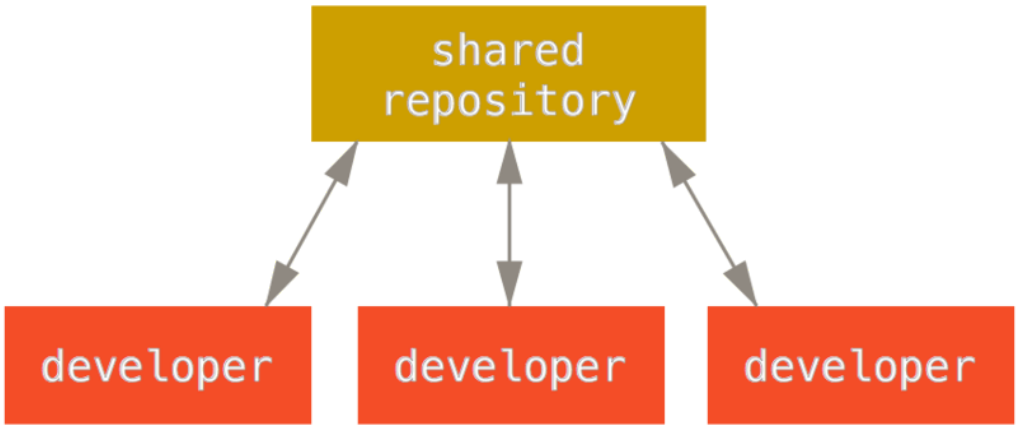
중앙집중형 버전 관리 시스템과는 달리 Git 은 구조가 매우 유연하기 때문에 여러 개발자가 함께 작업하는 방식을 더 다양하게 구성할 수 있다.

- 중앙집중형 버전 관리 시스템
  - 각 개발자는 중앙 저장소를 중심으로 하는 한 노드
- Git
  - 각 개발자의 저장소가 하나의 노드 이기도 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 워크플로우 를 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 쓰는 것이 가능하다.

중앙집중식 워크플로우

중앙집중식 시스템에서는 보통 중앙집중식 협업 모델이라는 한 가지 방식 밖에 없다. 중앙 저장소는 하나만 있고 변경 사항은 모두 이 중앙 저장소에 집중된다. 개발자는 이 중앙 저장소를 중심으로 작업한다.



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 Jessica 가 동시에 같은 부분을 수정하는 상황을 생각해보자. John이 먼저 작업을 끝내고 수정한 내용을 서버로 Push 한다. Jessica 도 마찬가지로 작업을 끝내고 수정한 내용을 서버로 Push 하려 하지만 서버가 바로 받아주지 않는다. 서버에는 John 이 수정한 내용이 추가되었기 때문에 Push 하기 전에 Fetch 로 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.

Git Flow 란?

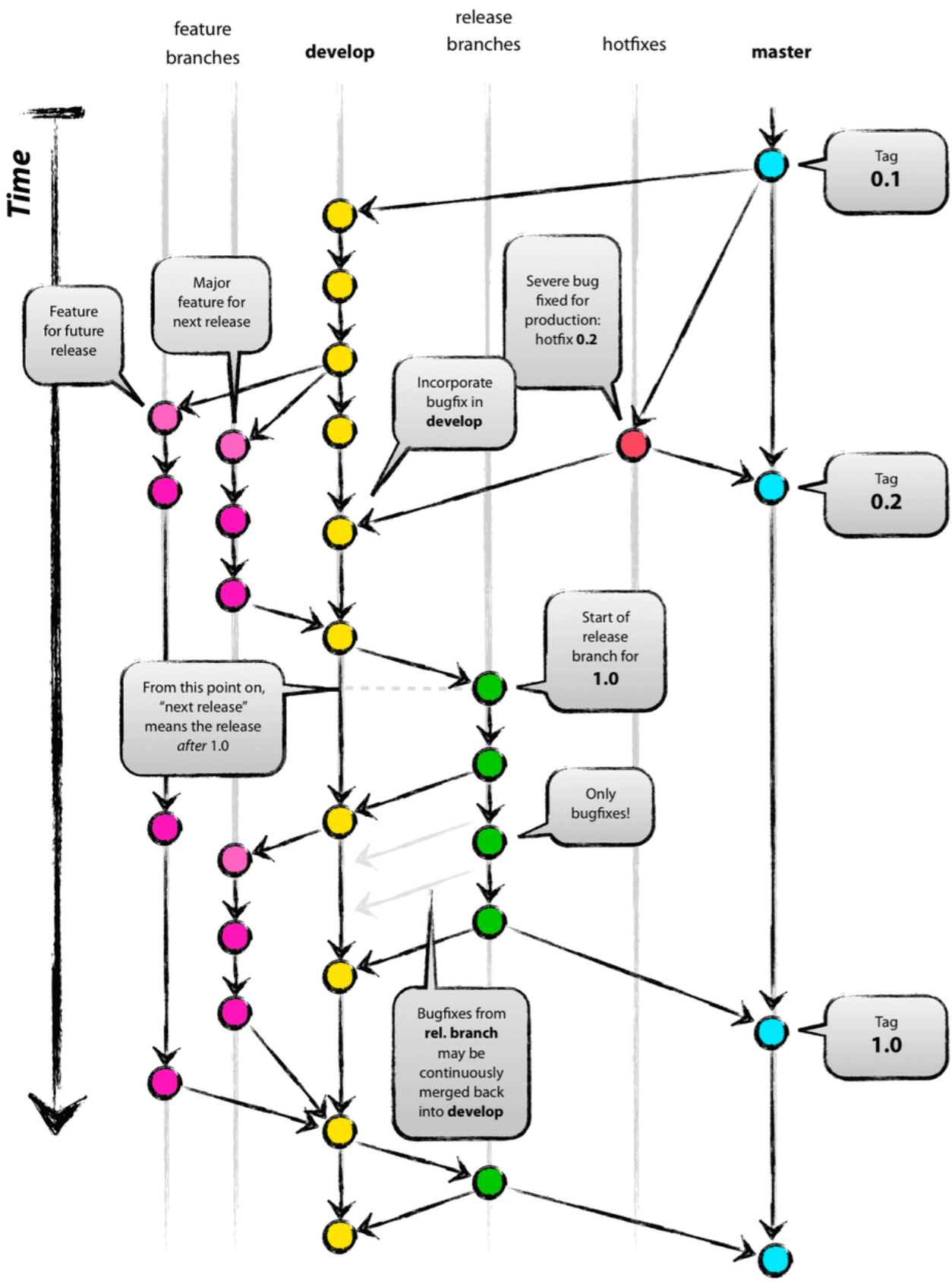
Git-flow는 Git이 새롭게 활성화되기 시작하는 10년전 쯤에 **Vincent Driessen** 이라는 사람의 블로그 글에 의해 널리 퍼지기 시작했고 현재는 Git으로 개발할 때 거의 표준과 같이 사용되는 방법론입니다.

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 떄 긴급 수정하는 브랜치 입니다.

> **master**와 **develop**가 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

2. Git Flow 실습

실습 진행전에 day1 repository 를 초기화 합니다.

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

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

실습을 진행합니다.

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

```
# main 으로부터 develop 개발 브랜치를 생성합니다.
$ git checkout develop

# C1, C2 커밋을 합니다.
$ 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.

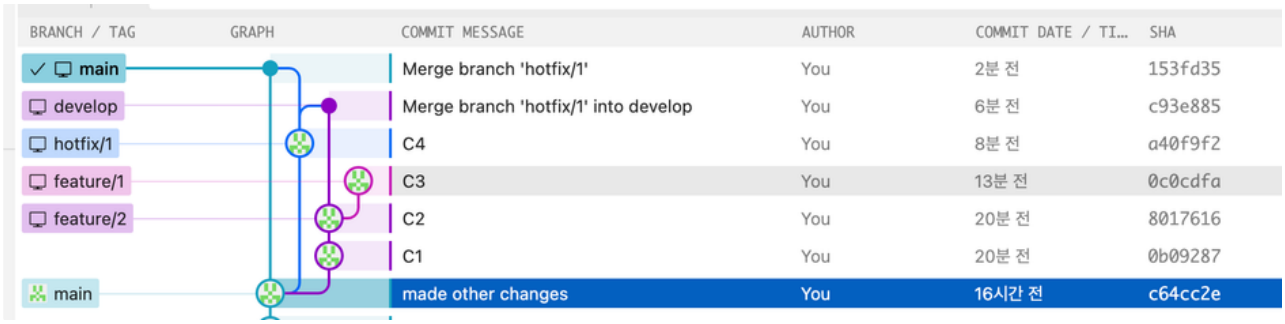
```
# develop 으로부터 feature/1, feature/2 를 생성합니다.
$ git checkout develop
$ git branch feature/1
$ git branch feature/2

# feature1 개발자가 C3 commit 을 합니다.
$ 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 으로부터 hotfix/1 을 생성합니다.
$ git checkout main
$ git checkout -b hotfix/1 (-b 옵션은 브랜치 생성과 체크아웃을 동시에 합니다.)

# 긴급 수정 C4 를 commit 합니다.
$ echo 'text' > C4
```

```
$ git add .
$ git commit -m 'C4'

# develop, main 양쪽에 반영합니다.
# 이때 --no-ff 옵션을 주어서, Fast-Foward 를 방지하고 3way merge 를 강제하시면
커밋 히스토리를 관리하기 용이합니다.
$ 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 개발을 마무리 합니다.
$ git checkout feature/1
$ echo 'text' > C5
$ git add .
$ git commit -m 'C5'

# feature/1 을 develop 으로 병합합니다.
$ git checkout develop
$ git merge feature/1 --no-ff

```

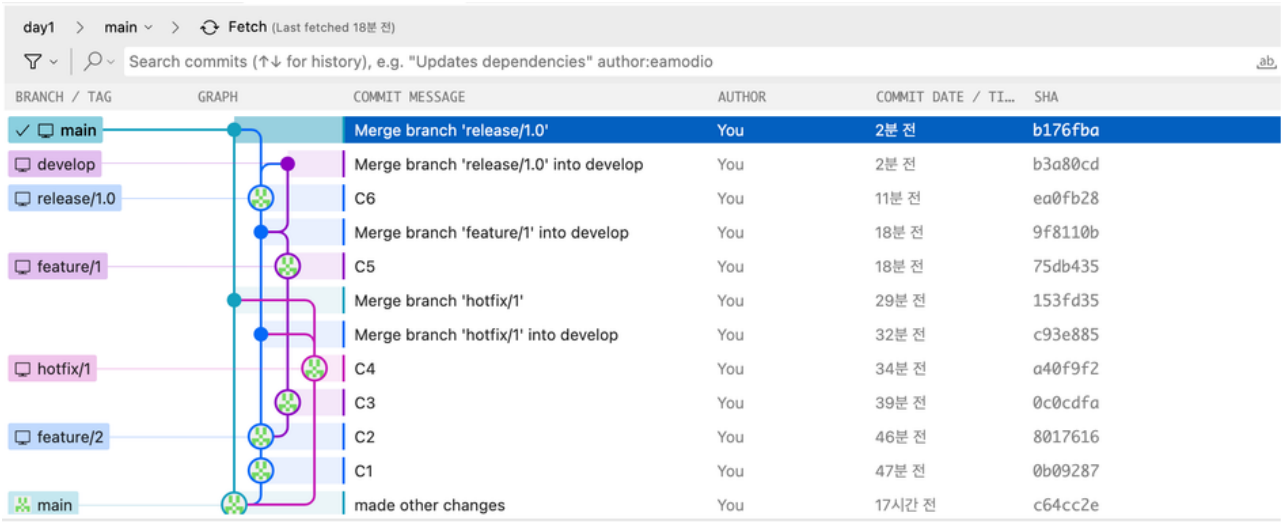
Sin5. 제품 담당자는 feature/1 기능까지 이번 운영 배포로 내보내기로 합니다. QA 를 위해서 릴리즈 1.0 을 구성합니다. QA 를 하는 도중 버그가 발견되었습니다!! QA 연락을 받은 개발자가 릴리즈 브랜치에서 버그를 수정합니다..

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

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

# release/1.0 로 QA 수행중...

# QA 중 버그가 발견되어 수정합니다.
$ echo 'text' > C6
$ git add .
$ git commit -m 'C6'

```

```
# QA 가 통과되었습니다. develop, main 양쪽에 반영합니다.
$ git checkout develop
$ git merge release/1.0 --no-ff
$ git checkout main
$ git merge release/1.0 --no-ff

# main 은 운영 배포되었습니다.
```

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

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

# QA 를 수행하기 위해 release/1.1 을 구성합니다.
$ git checkout develop
$ git checkout -b release/1.1

# release/1.0 로 QA 수행중...

# QA 중 버그가 발견되어 수정합니다.
$ echo 'text' > C9
$ git add .
$ git commit -m 'C9'

```

```
# QA 수행중...

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

# main 은 운영 배포되었습니다.
```

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