4. Git 브랜치로 작업하기 I

1. Git 브랜치 시작하기

브랜치 이해하기 - Commit 자료 구조

개발을 하다 보면 코드를 여러 개로 복사해야 하는 일이 자주 생긴다. 코드를 통째로 복사하고 나서 원래 코드와는 상관없이 독립적으로 개발을 진행할 수 있는데, 이렇게 독립적으로 개발하는 것이 브랜치다.

- 브랜치 모델
  - Git의 장점, Git이 다른 것들과 구분되는 특징.
  - Git의 브랜치는 매우 가볍다.
  - Git은 브랜치를 만들어 작업하고 나중에 Merge 하는 방법을 권장한다.

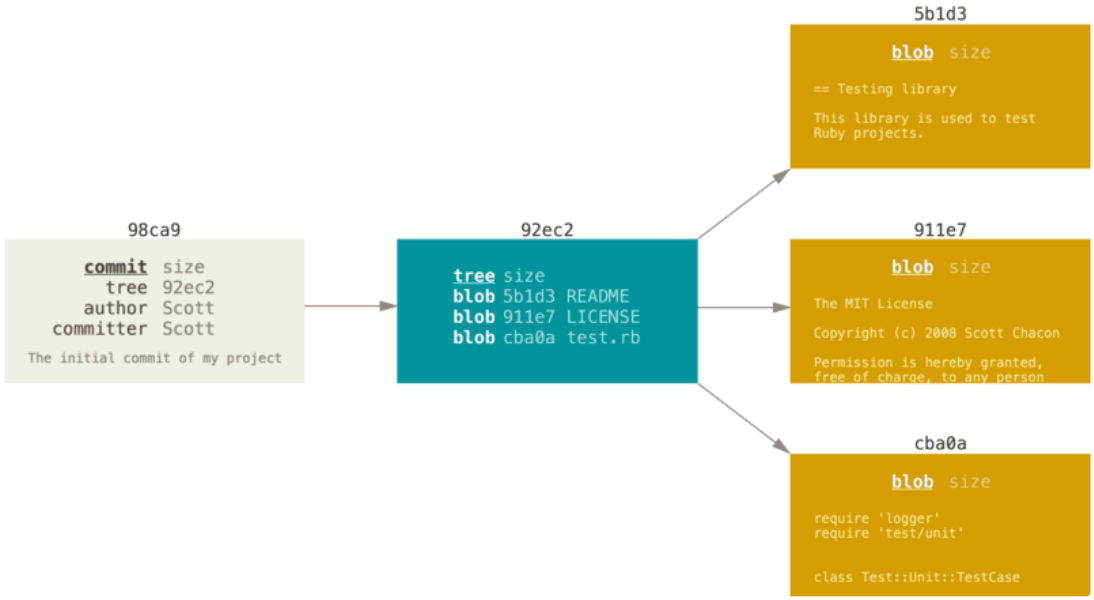
Git은 데이터를 Change Set 이나 변경사항(Diff)으로 기록하지 않고 일련의 스냅샷으로 기록한다.

다음과 같이 3개의 파일 변경사항이 커밋 되었다고 가정해보자.

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

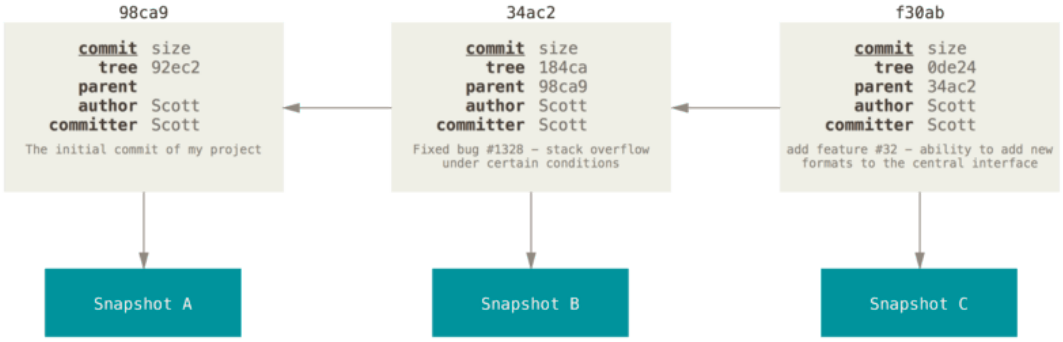
`git commit` 으로 커밋하면 Git 저장소에는 다섯 개의 데이터 개체가 생긴다.

- Blob: 각각의 파일에 대한 스냅샷
- 트리 개체: 파일과 디렉토리 구조가 들어 있다.
- 커밋 개체: 메타데이터와 루트 트리를 가리키는 포인터가 들어있다.



브랜치 이해하기 - HEAD 포인트와 브랜치

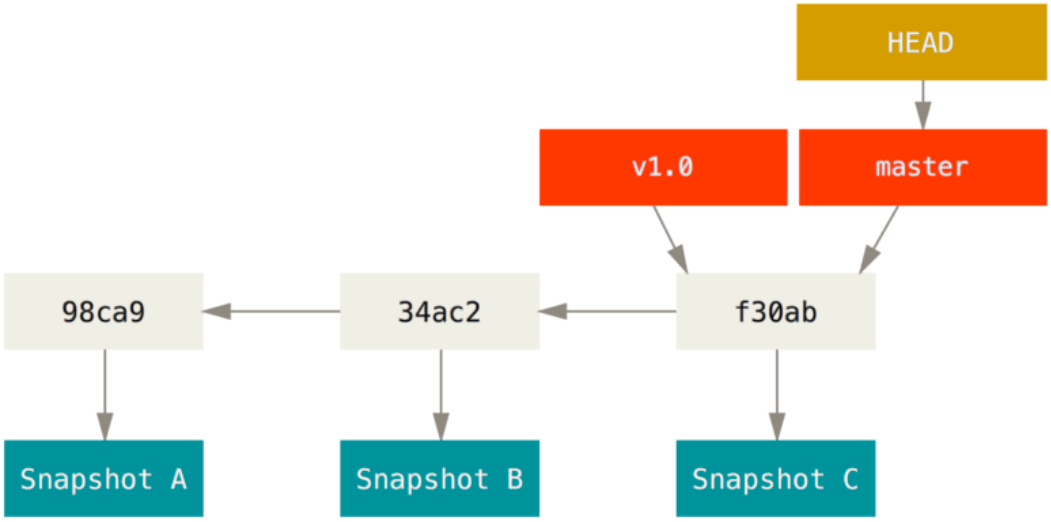
커밋 개체는 parent 항목에 이전 커밋이 무엇인지 저장한다.



git 특정한 커밋으로 포인터(HEAD) 를 맞출 때, git 은 포인터에 해당하는 커밋 개체의 트리 개체 들과 Blob 정보를 이용하여 스냅샷을 워킹 디렉토리에 복원하는 것이다.

그리고 branch 는 이 특정한 포인트(HEAD) 가 가볍게 이동할 수 있는 어떤 이정표 같은 것이다.

즉, branch 를 생성하고 관리한다는 것은 포인트(HEAD) 를 이동 시킴으로써 원하는 시점의 스냅샷을 대상으로 개발 작업을 용이하게 한다는 뜻이다.



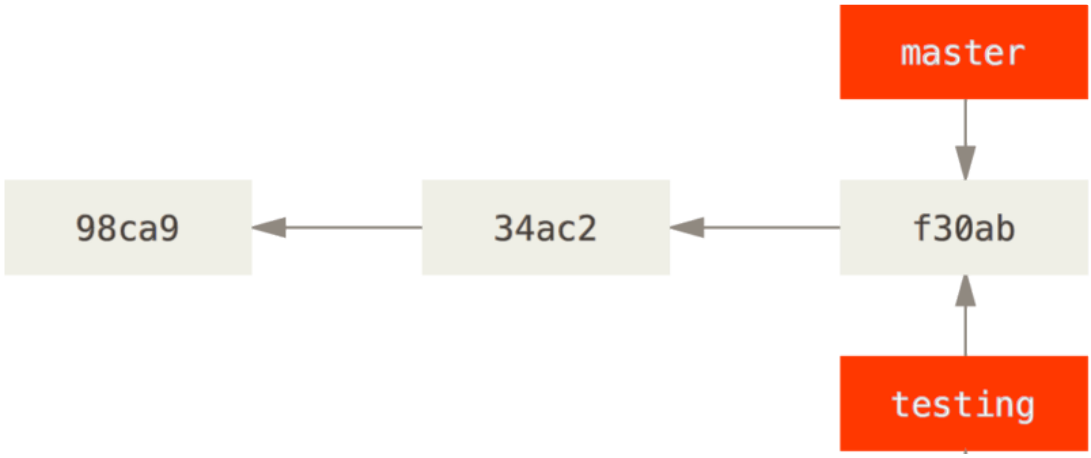
기본적으로 Git은 `main` 브랜치를 만든다. 처음 커밋하면 이 `main` 브랜치가 생성된 커밋을 가리킨다. 이후 커밋을 만들면 `main` 브랜치는 자동으로 가장 마지막 커밋을 가리킨다.

새 브랜치 생성하기

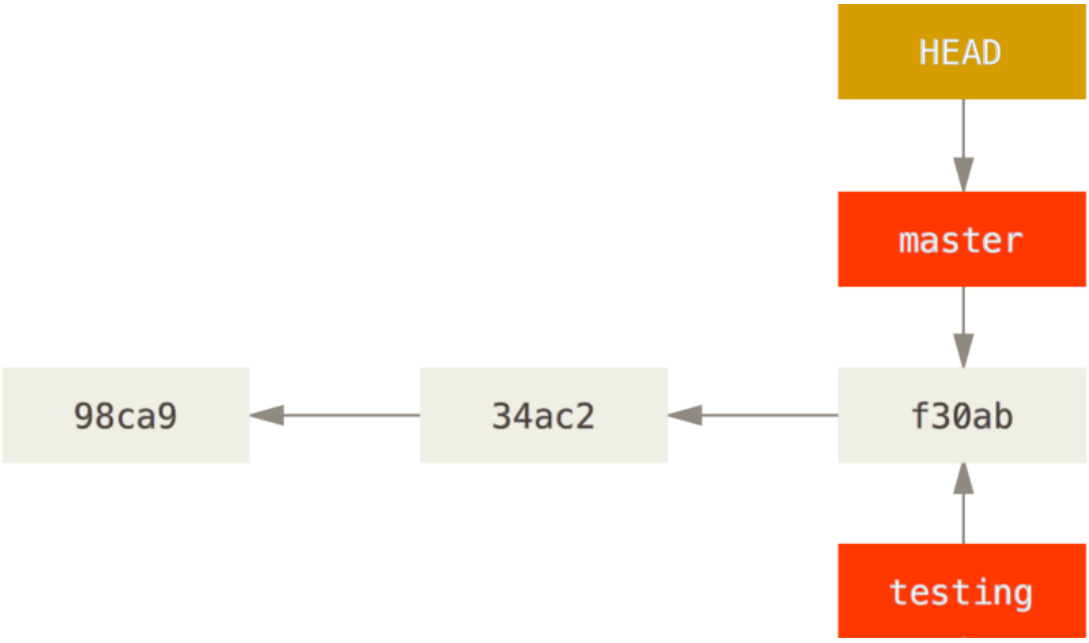
아래와 같이 `git branch` 명령으로 `testing` 브랜치를 만든다.

```
$ git branch testing
```

새로 만든 브랜치도 지금 작업하고 있던 마지막 커밋을 가리킨다.



지금 작업 중인 브랜치가 무엇인지 Git은 어떻게 파악할까. 다른 버전 관리 시스템과는 달리 Git은 'HEAD'라는 특수한 포인터가 있다. 이 포인터는 지금 작업하는 로컬 브랜치를 가리킨다. 브랜치를 새로 만들었지만, Git은 아직 `main` 브랜치를 가리키고 있다. `git branch` 명령은 브랜치를 만들기만 하고 브랜치를 옮기지 않는다.



`git log` 명령에 `--decorate` 옵션을 사용하면 쉽게 브랜치가 어떤 커밋을 가리키는지도 확인할 수 있다.

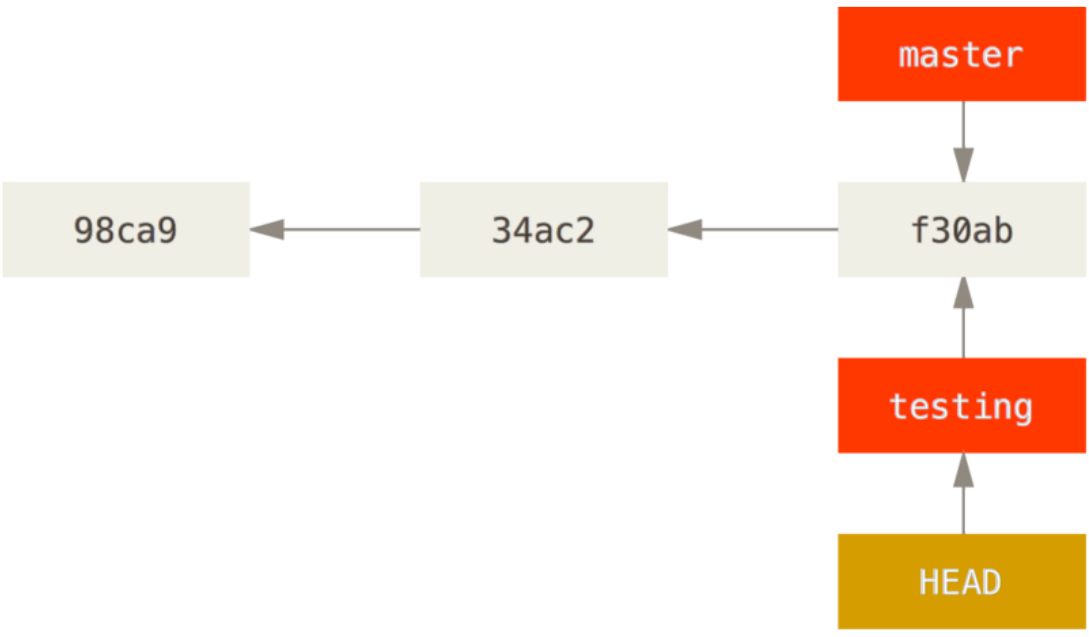
```
1 $ git log --oneline --decorate
```

브랜치 분리하기

`git checkout` 명령으로 다른 브랜치로 이동할 수 있다. 한번 `testing` 브랜치로 바꿔보자.

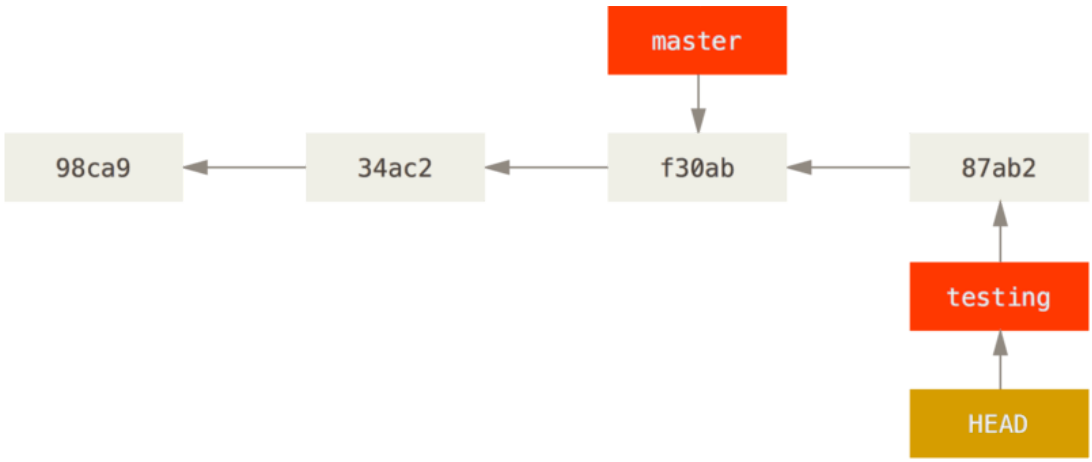
```
1 $ git checkout testing
```

이렇게 하면 HEAD는 testing 브랜치를 가리킨다.



이 상태에서 커밋을 새로 한 번 해보자.

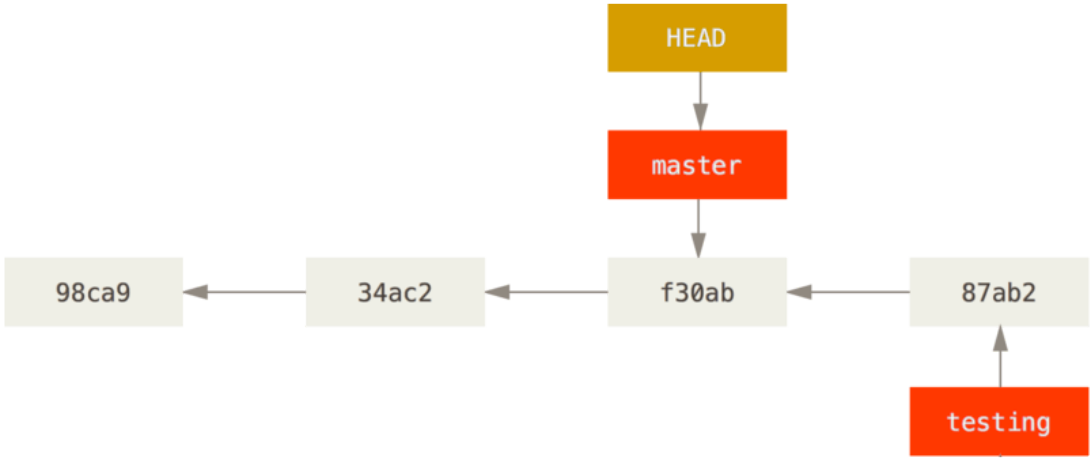
```
$ echo 'test' > test.rb
$ git add test.rb
$ git commit -m 'made a change'
```



새로 커밋해서 `testing` 브랜치는 앞으로 이동했다. 하지만, `main` 브랜치는 여전히 이전 커밋을 가리킨다.

`main` 브랜치로 되돌아가보자.

```
$ git checkout main
```



방금 `checkout` 명령이 한 일은 두 가지다. `main` 브랜치가 가리키는 커밋을 HEAD가 가리키게 하고 워킹 디렉토리의 파일도 그 시점으로 되돌려 놓았다. 앞으로 커밋을 하면 다른 브랜치의 작업들과 별개로 진행되기 때문에 `testing` 브랜치에서 임시로 작업하고 원래 `main` 브랜치로 돌아와서 하던 일을 계속할 수 있다.

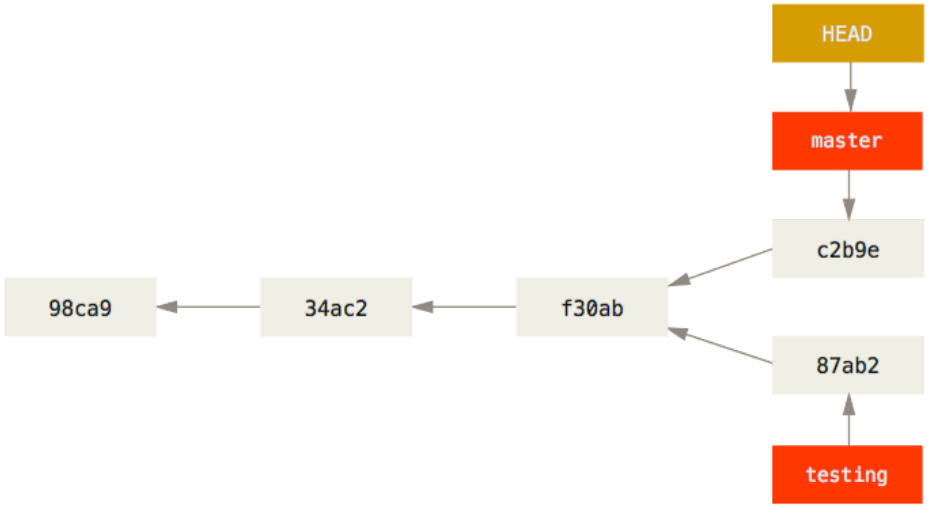
노트	브랜치를 이동하면 워킹 디렉토리의 파일이 변경된다 브랜치를 이동하면 워킹 디렉토리의 파일이 변경된다는 점을 기억해두어야 한다. 이전에 작업했던 브랜치로 이동하면 워킹 디렉토리의 파일은 그 브랜치에서 가장 마지막으로 했던 작업 내용으로 변경된다. 파일 변경시 문제가 있어 브랜치를 이동시키는게 불

	가능한 경우 Git은 브랜치 이동 명령을 수행하지 않는다.

파일을 수정하고 다시 커밋을 해보자.

```
$ echo 'test' > test.rb
$ git add test.rb
$ git commit -m 'made other changes'
```

프로젝트 히스토리가 분리되었다. 방금 실습에서 브랜치를 하나 만들어 그 브랜치에서 일을 좀 하고, 다시 원래 브랜치로 되돌아와서 다른 일을 했다. 두 작업 내용은 서로 독립적으로 각 브랜치에 존재한다. 커밋 사이를 자유롭게 이동하다가 때가 되면 두 브랜치를 병합(merge) 한다.



이러한 브랜치의 갈라짐과 병합 동작은, 개발팀으로 하여금 동시에 여러건의 개발 작업이 가능하게 한다.

`git log` 명령으로 쉽게 확인할 수 있다. 현재 브랜치가 가리키고 있는 히스토리가 무엇이고 어떻게 갈라져 나왔는지 보여준다. `git log --oneline --decorate --graph --all` 이라고 실행하면 히스토리를 출력한다.

```
$ git log --oneline --decorate --graph --all
* c64cc2e (HEAD -> main) made other changes
* fe07710 (tag: v1.1) add new file
| * 3d5212b (testing) made a change
|/
* 7f5ad25 (tag: v1.0, origin/main, origin/HEAD) Create new_file
* 2a082e9 reset
* a399df2 initial commit
* 359e6e8 rm README
* a39442e add PROJECTS.md
* 7f73ba0 added new benchmarks
* 73e66cf Story 182: Fix benchmarks for speed
* 928924c (tag: v0.9, rm) Initial commit
```

실제로 Git의 브랜치는 어떤 한 커밋을 가리키는 40글자의 SHA-1 체크섬 파일에 불과하기 때문에 만들기도 쉽고 지우기도 쉽다. 새로 브랜치를 하나 만드는 것은 41바이트 크기의 파일을(40자와 줄 바꿈 문자) 하나 만드는 것에 불과하다.

- 다른 버전 도구: 브랜치가 필요할 때 프로젝트를 통째로 복사, 수십 초에서 수십 분까지 걸린다.
- Git 브랜치는 순식간이며, 커밋을 할 때마다 이전 커밋의 정보를 저장하기 때문에 Merge 할 때 어디서부터(Merge Base) 합쳐야 하는지 안다. 이런 특징은 개발자들이 수시로 브랜치를 만들어 사용하게 한다.

---

2. Git 브랜치로 작업하기

브랜치를 활용한 작업 예시

실제 개발과정에서 겪을 만한 예제를 하나 살펴보자. 브랜치와 Merge는 보통 이런 식으로 진행한다.

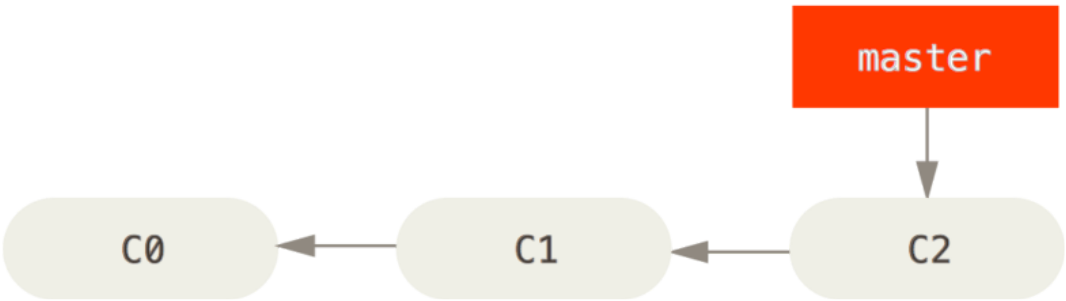
1. 웹사이트가 있고 뭔가 작업을 진행하고 있다.
2. 새로운 이슈를 처리할 새 Branch를 하나 생성한다.
3. 새로 만든 Branch에서 작업을 진행한다.

이때 중요한 문제가 생겨서 그것을 해결하는 Hotfix를 먼저 만들어야 한다. 그러면 아래와 같이 할 수 있다.

1. 새로운 이슈를 처리하기 이전의 운영(Production) 브랜치로 이동한다.
2. Hotfix 브랜치를 새로 하나 생성한다.
3. 수정한 Hotfix 테스트를 마치고 운영 브랜치로 Merge 한다.
4. 다시 작업하던 브랜치로 옮겨가서 하던 일 진행한다.

Fast-forward Merge

먼저 지금 작업하는 프로젝트에서 이전에 `main` 브랜치에 커밋을 몇 번 했다고 가정한다.

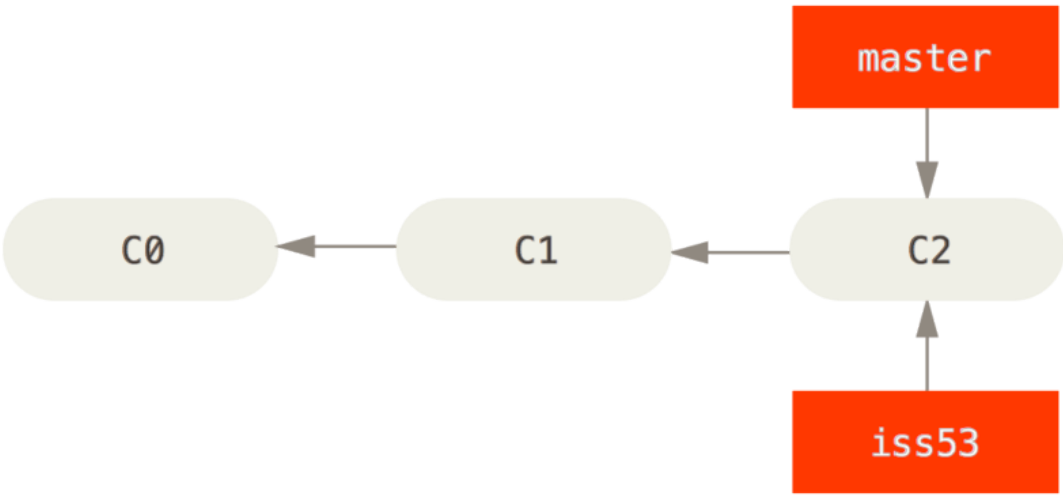


이슈 관리 시스템에 등록된 53번 이슈를 처리한다고 하면 이 이슈에 집중할 수 있는 브랜치를 새로 하나 만든다. 브랜치를 만들면서 Checkout 까지 한 번에 하려면 `git checkout` 명령에 `-b` 라는 옵션을 추가한다.

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

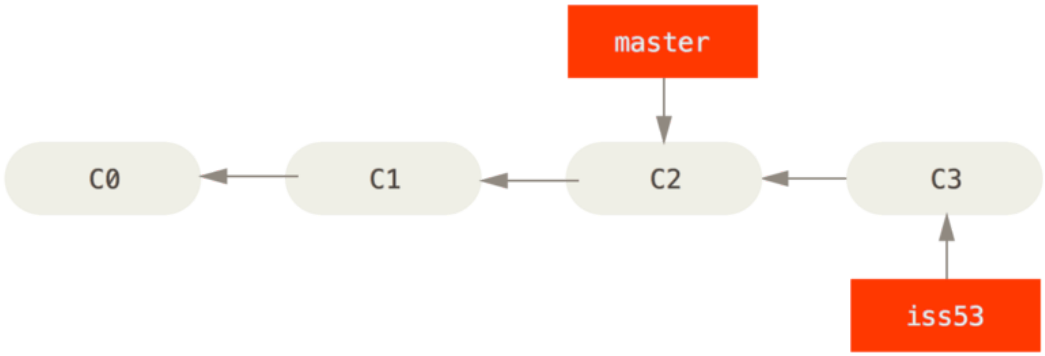
위 명령은 아래 명령을 줄여놓은 것이다.

```
$ git branch iss53
$ git checkout iss53
```



`iss53` 브랜치를 Checkout 했기 때문에(즉, `HEAD` 는 `iss53` 브랜치를 가리킨다) 뭔가 일을 하고 커밋하면 `iss53` 브랜치가 앞으로 나아간다.

```
$ echo '<footer>xxx</footer>' > footer.html
$ git add .
$ git commit -m 'added a new footer [issue 53]'
[iss53 19e60ed] added a new footer [issue 53]
 1 file changed, 1 insertion(+)
 create mode 100644 footer.html
```



다른 상황을 가정해보자. 만드는 사이트에 문제가 생겨서 즉시 고쳐야 한다. 버그를 해결한 Hotfix 에 `iss53` 이 섞이는 것을 방지하기 위해 `iss53` 과 관련된 코드를 어딘가에 저장해두고 원래 운영 환경의 소스로 복구해야 한다. 이것은 `main` 브랜치로 돌아가는 것을 의미한다.

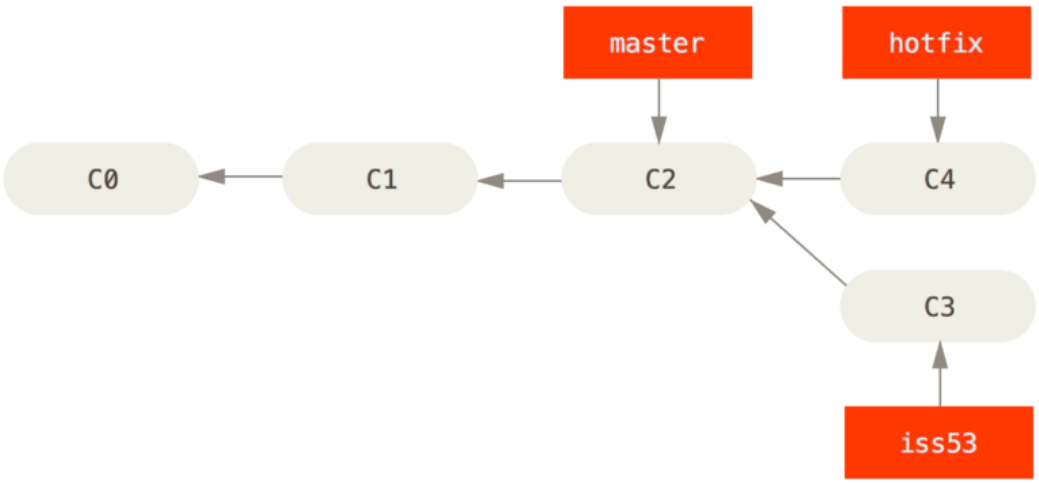
브랜치를 `main` 으로 이동할 때, 아직 커밋하지 않은 파일이 Checkout 할 브랜치와 충돌 나면 브랜치를 변경할 수 없다. 브랜치를 변경할 때는 워킹 디렉토리를 정리하는 것이 좋다. 작업하던 것을 모두 커밋하고 `main` 브랜치로 옮긴다:

```
$ git checkout main
Switched to branch 'main'
```

`hotfix` 라는 브랜치를 만들고 새로운 이슈를 해결한다.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ echo '<address>email</address>' > index.html
$ git add .
$ git commit -m 'fixed the broken email address'
[hotfix 907346f] fixed the broken email address
```

```
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
```



운영 환경에 적용하려면 문제를 제대로 고쳤는지 테스트하고 최종적으로 운영환경에 배포하기 위해 `hotfix` 브랜치를 `main` 브랜치에 합쳐야 한다. `git merge` 명령으로 아래와 같이 한다.

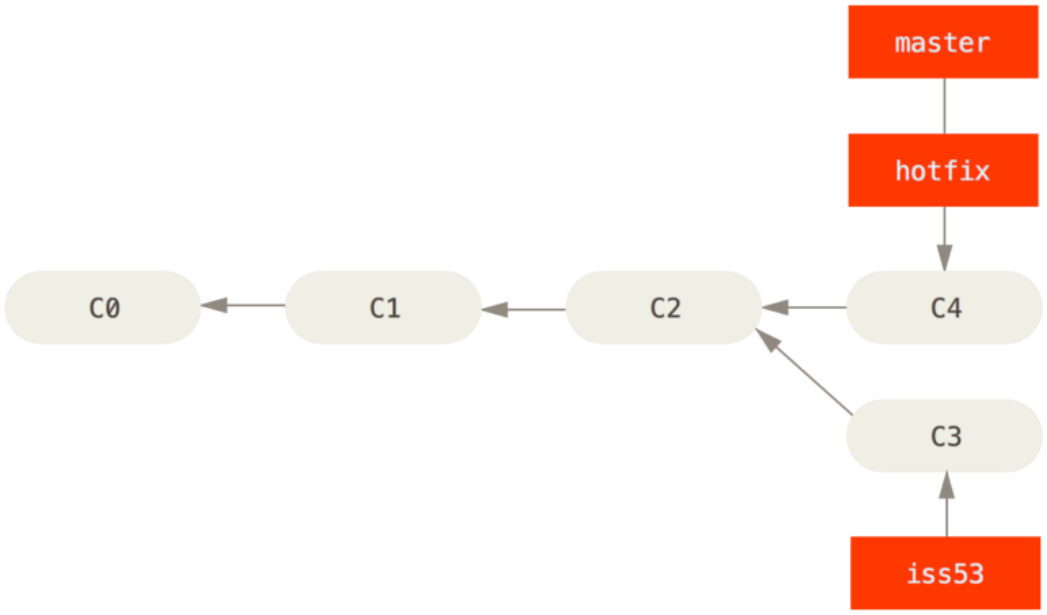
```
$ git checkout main
$ git merge hotfix
Updating c64cc2e..907346f
Fast-forward
 index.html | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
```

Merge 메시지에서 “Fast-forward” 가 보인다.

`hotfix` 브랜치가 가리키는 `C4` 커밋이 `C2` 커밋에 기반한 브랜치이기 때문에 브랜치 포인터는 Merge 과정 없이 그저 최신 커밋으로 이동한다. 이런 Merge 방식을 “Fast forward” 라고 부른다.

이제 `hotfix` 는 `main` 브랜치에 포함됐고 운영환경에 적용할 수 있는 상태가 되었다고 가정해보자.

[Merge 후 `hotfix` 같은 것을 가리키는 `main` 브랜치]



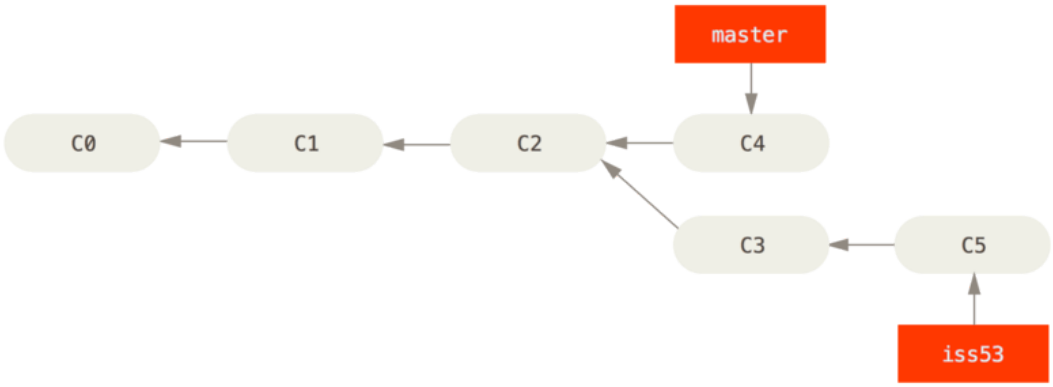
급한 문제를 해결하고 `main` 브랜치에 적용하고 나면 다시 일하던 브랜치로 돌아가야 한다. 이제 더 이상 필요없는 `hotfix` 브랜치는 삭제한다. `git branch` 명령에 `-d` 옵션을 주고 브랜치를 삭제한다.

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

3-way Merge

이제 이슈 53번을 처리하던 환경으로 되돌아가서 하던 일을 계속 하자.

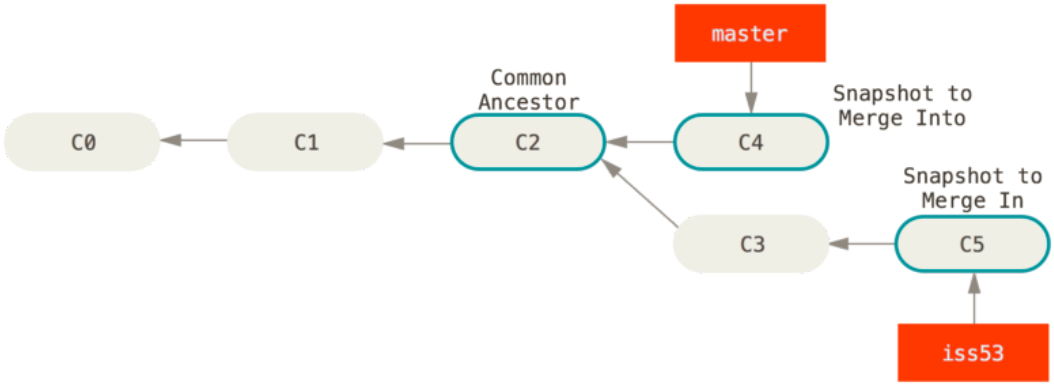
```
$ git checkout iss53
Switched to branch "iss53"
$ echo '<footer>finish</footer>' > footer.html
$ git add .
$ git commit -m 'finished the new footer [issue 53]'
[iss53 d9d5c22] finished the new footer [issue 53]
 1 file changed, 1 insertion(+), 1 deletion(-)
```



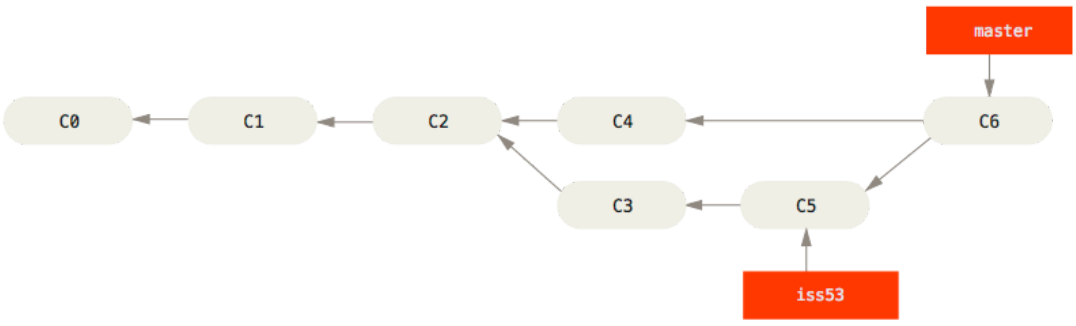
53번 이슈를 다 구현하고 `main` 브랜치에 Merge 하는 과정을 살펴보자. `iss53` 브랜치를 `main` 브랜치에 Merge 하는 것은 앞서 살펴본 `hotfix` 브랜치를 Merge 하는 것과 비슷하다. `git merge` 명령으로 합칠 브랜치에서 합쳐질 브랜치를 Merge 하면 된다.

```
$ git checkout main
Switched to branch 'main'
$ git merge iss53
Merge made by the 'ort' strategy.
 footer.html | 1 +
 1 file changed, 1 insertion(+)
```

현재 브랜치가 가리키는 커밋이 Merge 할 브랜치의 조상이 아니므로 Git은 'Fast-forward'로 Merge 하지 않는다. 이 경우에는 Git은 각 브랜치가 가리키는 커밋 두 개와 공통 조상 하나를 사용하여 3-way Merge를 한다.



단순히 브랜치 포인터를 최신 커밋으로 옮기는 게 아니라 3-way Merge 의 결과를 별도의 커밋으로 만들고 나서 해당 브랜치가 그 커밋을 가리키도록 이동시킨다. 그래서 이런 커밋은 Merge 커밋이라고 부른다.



iss53 브랜치를 main에 Merge 하고 나면 더는 iss53 브랜치가 필요 없다. 다음 명령으로 브랜치를 삭제하고 이슈의 상태를 처리 완료로 표시한다.

```
$ git branch -d iss53
```

충돌 (Conflict) 해결

가끔씩 3-way Merge가 실패할 때도 있다. Merge 하는 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고 Merge 하면 Git은 해당 부분을 Merge 하지 못한다. 예를 들어, 53번 이슈와 `hotfix` 가 같은 부분을 수정했다면 Git은 Merge 하지 못하고 아래와 같은 충돌(Conflict) 메시지를 출력한다.

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Merge 충돌이 일어났을 때 Git이 어떤 파일을 Merge 할 수 없었는지 살펴보려면 `git status` 명령을 이용한다.

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

충돌이 일어난 파일은 unmerged 상태로 표시된다. Git은 충돌이 난 부분을 표준 형식에 따라 표시해준다. 그러면 개발자는 해당 부분을 수동으로 해결한다. 충돌 난 부분은 아래와 같이 표시된다.

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

`=======` 위쪽의 내용은 `HEAD` 버전(merge 명령을 실행할 때 작업하던 `main` 브랜치)의 내용이고 아래쪽은 `iss53` 브랜치의 내용이다. 충돌을 해결하려면 위쪽이나 아래쪽 내용 중에서 고르거나 새로 작성하여 Merge 한다. 아래는 아예 새로 작성하여 충돌을 해결하는 예제다.

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

충돌한 양쪽에서 조금씩 가져와서 새로 수정했다. 그리고 `<<<<<<<`, `=======`, `>>>>>>>` 가 포함된 행을 삭제했다. 이렇게 충돌한 부분을 해결하고 `git add` 명령으로 다시 Git에 저장한다.

Merge 도구를 종료하면 Git은 잘 Merge 했는지 물어본다. 잘 마쳤다고 입력하면 자동으로 `git add` 가 수행되고 해당 파일이 Staging Area에 저장된다. `git status` 명령으로 충돌이 해결된 상태인지 다시 한번 확인해볼 수 있다.

```
$ git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
```

```
7
8     modified:   index.html
```

충돌을 해결하고 나서 해당 파일이 Staging Area에 저장됐는지 확인했으면 `git commit` 명령으로 Merge 한 것을 커밋한다. 충돌을 해결하고 Merge 할 때는 커밋 메시지가 아래와 같다.

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch main
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

어떻게 충돌을 해결했고 좀 더 확인해야 하는 부분은 무엇이고 왜 그렇게 해결했는지에 대해서 자세하게 기록 해 준다.

브랜치 관리

`git branch` 명령을 아무런 옵션 없이 실행하면 브랜치의 목록을 보여준다.

```
$ git branch
  iss53
* main
  testing
```

`*` 기호가 붙어 있는 `main` 브랜치는 현재 Checkout 해서 작업하는 브랜치를 나타낸다. 즉, 지금 수정한 내용을 커밋하면 `main` 브랜치에 커밋되고 포인터가 앞으로 한 단계 나아간다. `git branch -v` 명령을 실행하면 브랜치마다 마지막 커밋 메시지도 함께 보여준다.

```
$ git branch -v
  iss53   93b412c fix javascript issue
* main  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

각 브랜치가 지금 어떤 상태인지 확인하기에 좋은 옵션도 있다. 현재 Checkout 한 브랜치를 기준으로 `--merged` 와 `--no-merged` 옵션을 사용하여 Merge 된 브랜치인지 그렇지 않은지 필터링해 볼 수 있다. `git branch --merged` 명령으로 이미 Merge 한 브랜치 목록을 확인한다.

```
$ git branch --merged
  iss53
* main
```

`iss53` 브랜치는 앞에서 이미 Merge 했기 때문에 목록에 나타난다. `*` 기호가 붙어 있지 않은 브랜치는 `git branch -d` 명령으로 삭제해도 되는 브랜치다. 이미 다른 브랜치와 Merge 했기 때문에 삭제해도 정보를 잃지 않는다.

반대로 현재 Checkout 한 브랜치에 Merge 하지 않은 브랜치를 살펴보려면 `git branch --no-merged` 명령을 사용한다.

```
$ git branch --no-merged
  testing
```

위에는 없었던 다른 브랜치가 보인다. 아직 Merge 하지 않은 커밋을 담고 있기 때문에 `git branch -d` 명령으로 삭제되지 않는다.

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Merge 하지 않은 브랜치를 강제로 삭제하려면 `-D` 옵션으로 삭제한다.

힌트	위에서 설명한 `--merged`, `--no-merged` 옵션을 사용할 때 커밋이나 브랜치 이름을 지정해주지 않으면 *현재* 브랜치를 기준으로 Merge 되거나 Merge 되지 않은 내용을 출력한다.
	
	위 명령을 사용할 때 특정 브랜치를 기준으로 Merge 되거나 혹은 Merge 되지 않은 브랜치 정보를 살펴보려면 명령에 브랜치 이름을 지정해주면 된다. 예를 들어 `main` 브랜치에 아직 Merge되지 않은 브랜치를 살펴보려면 다음과 같은 명령을 실행한다.
	
	`$ git checkout testing`
	`$ git branch --no-merged main`
	`  topicA`
	`  featureB`