1-4 Git 브랜치로 작업하기

3.1 Git 브랜치 - 브랜치란 무엇인가

모든 버전 관리 시스템은 브랜치를 지원한다. 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.

브랜치란 무엇인가

Git이 브랜치를 다루는 과정을 이해하려면 우선 Git이 데이터를 어떻게 저장하는지 알아야 한다.

Git은 데이터를 Change Set이나 변경사항(Diff)으로 기록하지 않고 일련의 스냅샷으로 기록한다는 것을 시작하기 에서 보여줬다.

커밋하면 Git은 현 Staging Area에 있는 데이터의 스냅샷에 대한 포인터, 저자나 커밋 메시지 같은 메타데이터, 이전 커밋에 대한 포인터 등을 포함하는 커밋 개체(커밋 Object)를 저장한다. 이전 커밋 포인터가 있어서 현재 커밋이 무엇을 기준으로 바뀌었는지를 알 수 있다. 최초 커밋을 제외한 나머지 커밋은 이전 커밋 포인터가 적어도 하나씩 있고 브랜치를 합친 Merge 커밋 같은 경우에는 이전 커밋 포인터가 여러 개 있다.

파일이 3개 있는 디렉토리가 하나 있고 이 파일을 Staging Area에 저장하고 커밋하는 예제를 살펴 보자. 파일을 Stage 하면 Git 저장소에 파일을 저장하고 (Git은 이것을 Blob이라고 부른다) Staging Area에 해당 파일의 체크섬을 저장한다(시작하기 에서 살펴본 SHA-1을 사용한다).

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

`git commit` 으로 커밋하면 먼저 루트 디렉토리와 각 하위 디렉토리의 트리 개체를 체크섬과 함께 저장소에 저장한다. 그다음에 커밋 개체를 만들고 메타데이터와 루트 디렉토리 트리 개체를 가리키는 포인터 정보를 커밋 개체에 넣어 저장한다. 그래서 필요하면 언제든지 스냅샷을 다시 만들 수 있다.

이 작업을 마치고 나면 Git 저장소에는 다섯 개의 데이터 개체가 생긴다. 각 파일에 대한 Blob 세 개, 파일과 디렉토리 구조가 들어 있는 트리 개체 하나, 메타데이터와 루트 트리를 가리키는 포인터가 담긴 커밋 개체 하나이다.

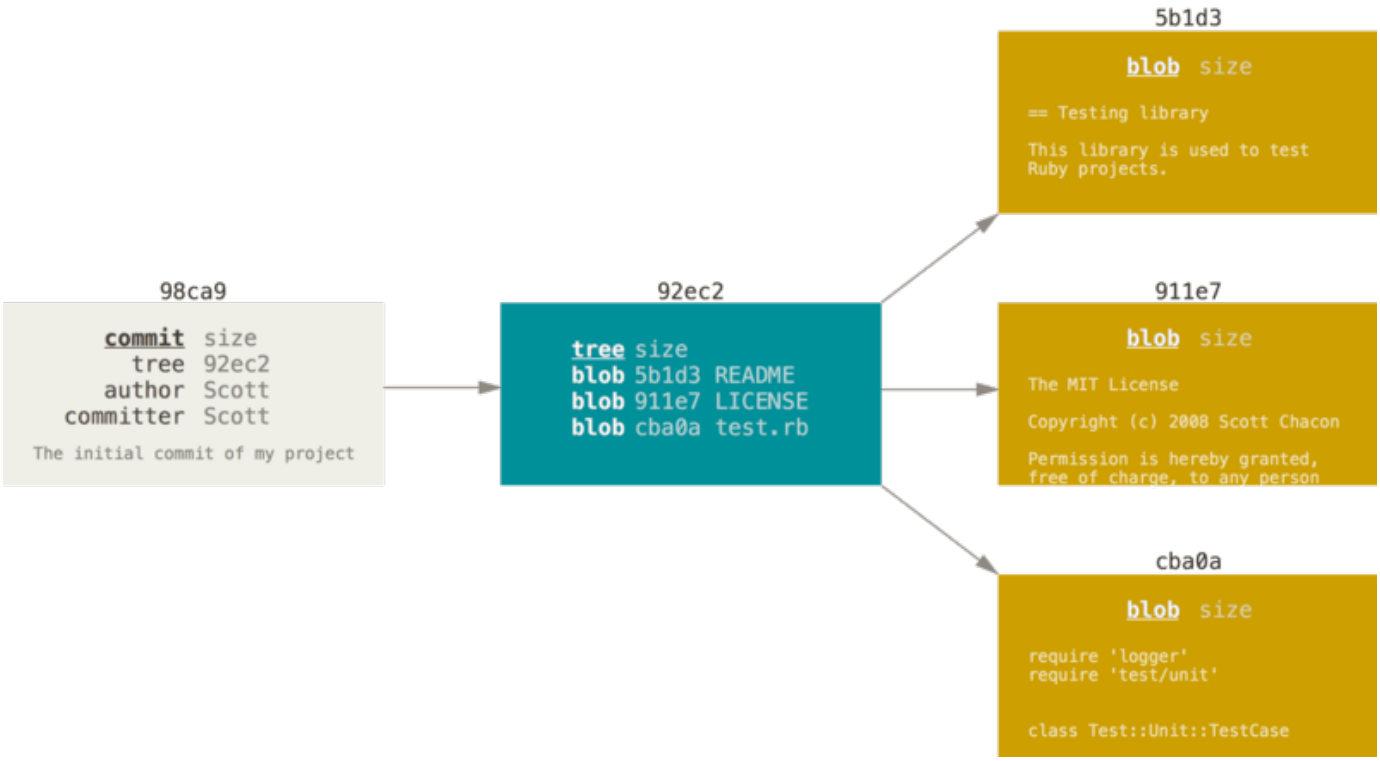


그림 9. 커밋과 트리 데이터

다시 파일을 수정하고 커밋하면 이전 커밋이 무엇인지도 저장한다.

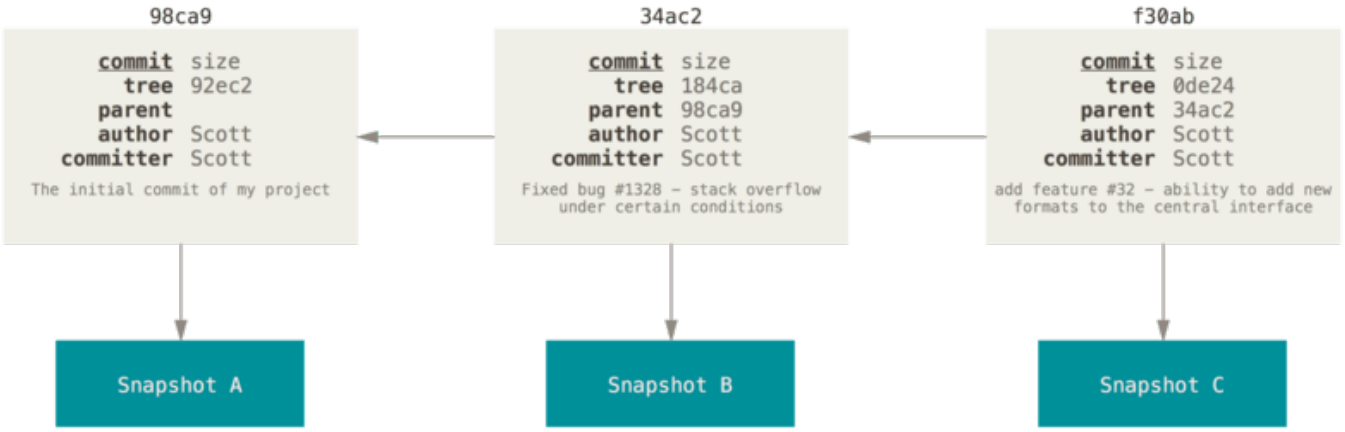


그림 10. 커밋과 이전 커밋

Git의 브랜치는 커밋 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것이다. 기본적으로 Git은 `main` 브랜치를 만든다. 처음 커밋하면 이 `main` 브랜치가 생성된 커밋을 가리킨다. 이후 커밋을 만들면 `main` 브랜치는 자동으로 가장 마지막 커밋을 가리킨다.

노트	Git 버전 관리 시스템에서 “`main`” 브랜치는 특별하지 않다. 다른 브랜치와 다른 것이 없다. 다만 모든 저장소에서 “`main`” 브랜치가 존재하는 이유는 `git init` 명령으로 초기화할 때 자동으로 만들어진 이 브랜치를 애써 다른 이름으로 변경하지 않기 때문이다.

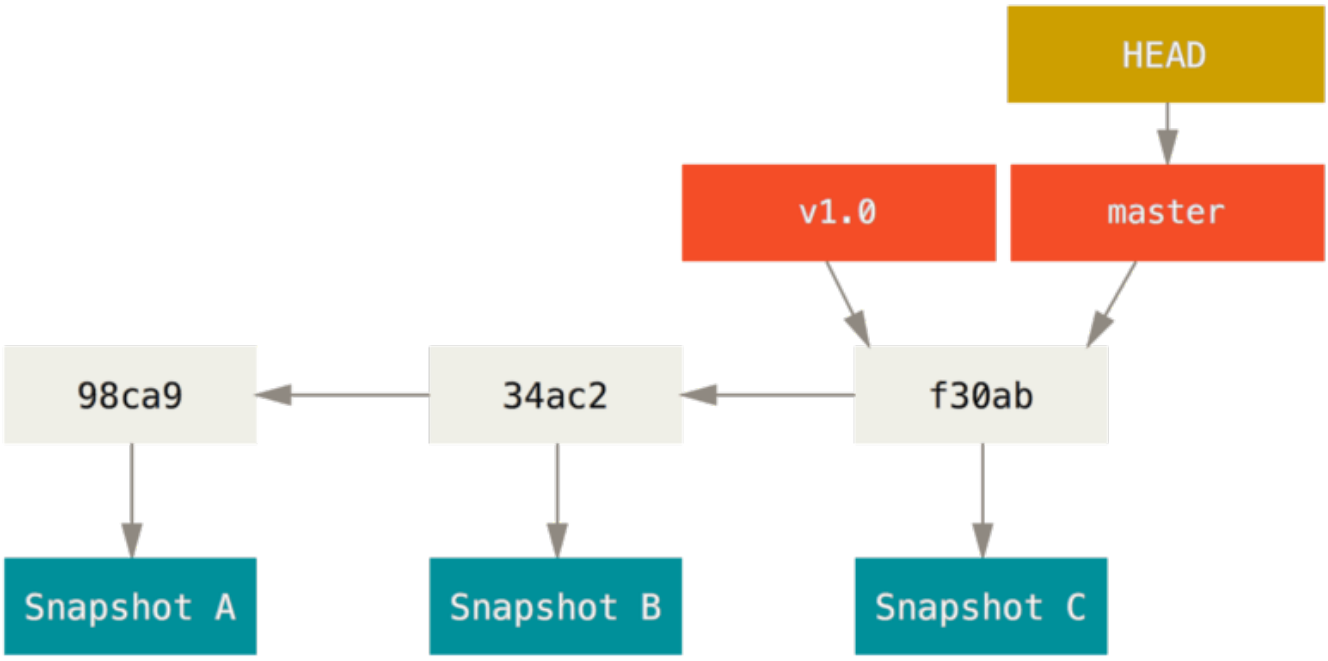


그림 11. 브랜치와 커밋 히스토리

새 브랜치 생성하기

브랜치를 하나 새로 만들면 어떨까. 브랜치를 하나 만들어 보자. 아래와 같이 `git branch` 명령으로 `testing` 브랜치를 만든다.

```
$ git branch testing
```

새로 만든 브랜치도 지금 작업하고 있던 마지막 커밋을 가리킨다.

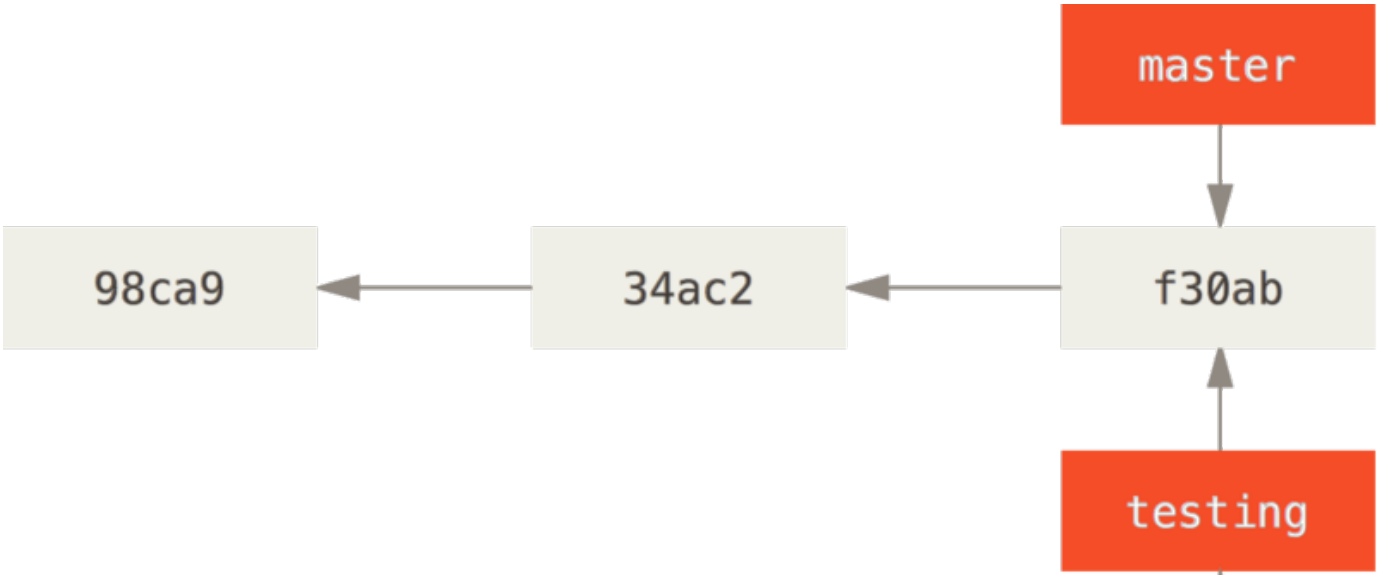


그림 12. 한 커밋 히스토리를 가리키는 두 브랜치

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.

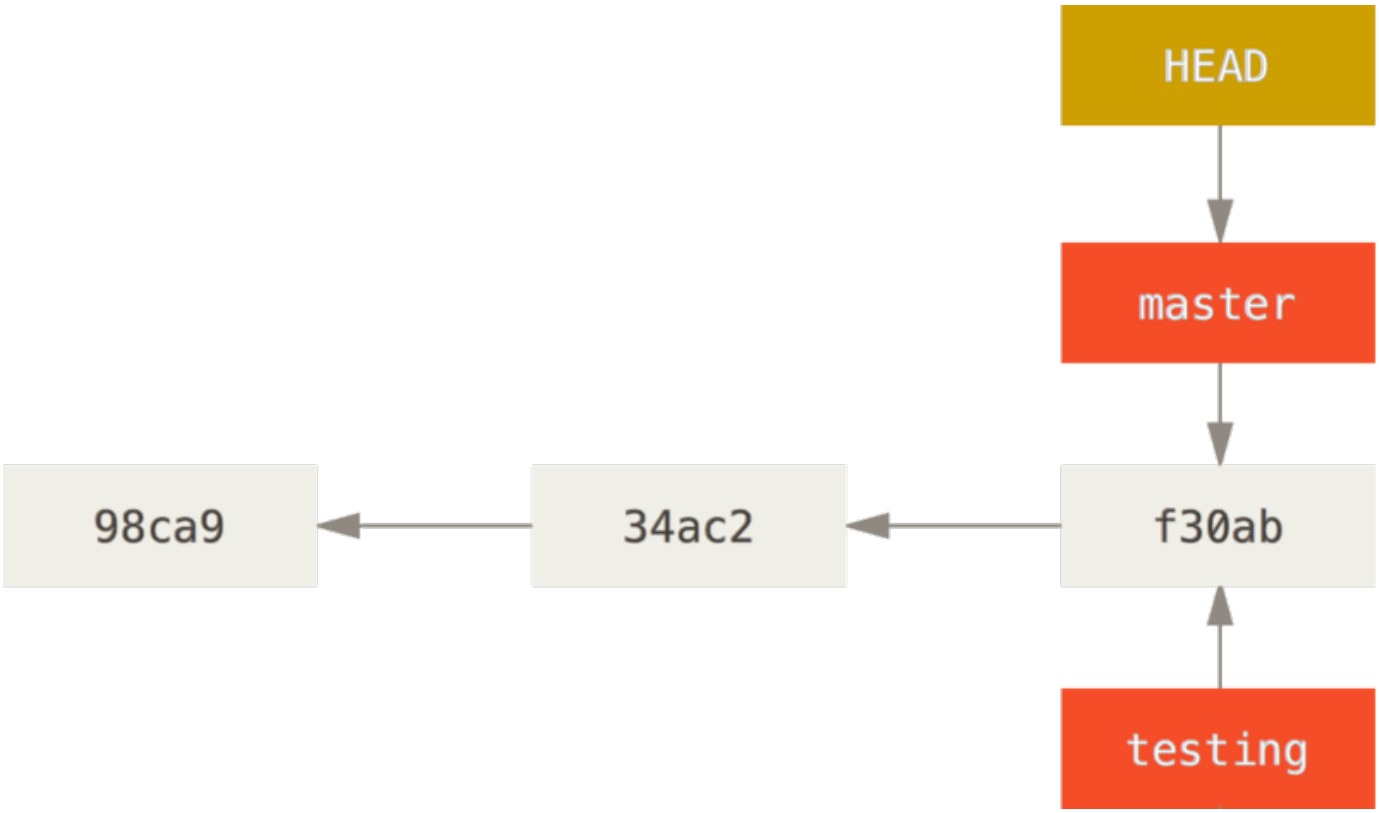


그림 13. 현재 작업 중인 브랜치를 가리키는 HEAD

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

```
$ git log --oneline --decorate
7f5ad25 (HEAD, tag: v1.0, origin/main, origin/HEAD, testing) 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
```

“main” 와 “testing” 이라는 브랜치가 `7f5ad25` 커밋 옆에 위치하여 이런식으로 브랜치가 가리키는 커밋을 확인할 수 있다.

브랜치 이동하기

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

```
$ git checkout testing
```

이렇게 하면 HEAD는 testing 브랜치를 가리킨다.

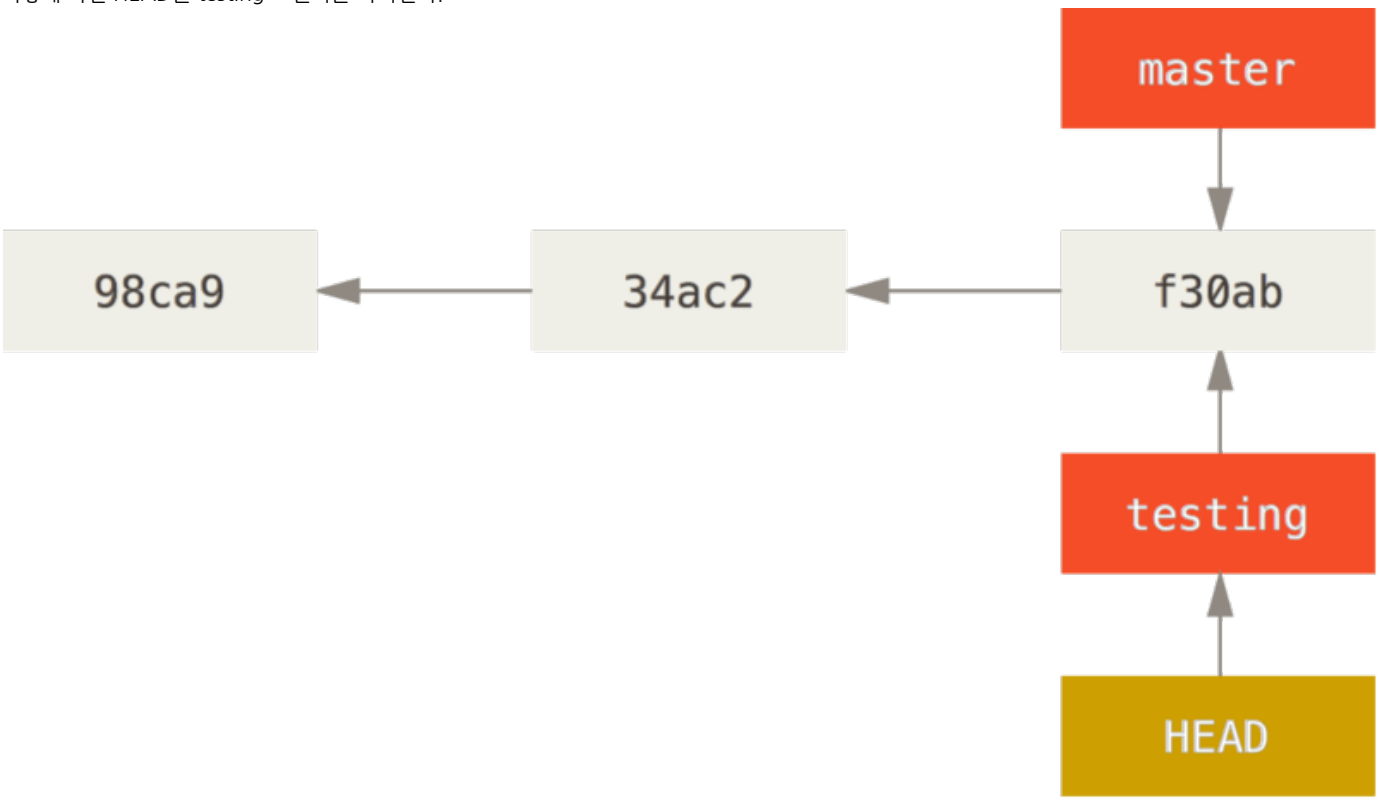


그림 14. HEAD는 testing 브랜치를 가리킴

이 상태에서 커밋을 새로 한 번 해보자.

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

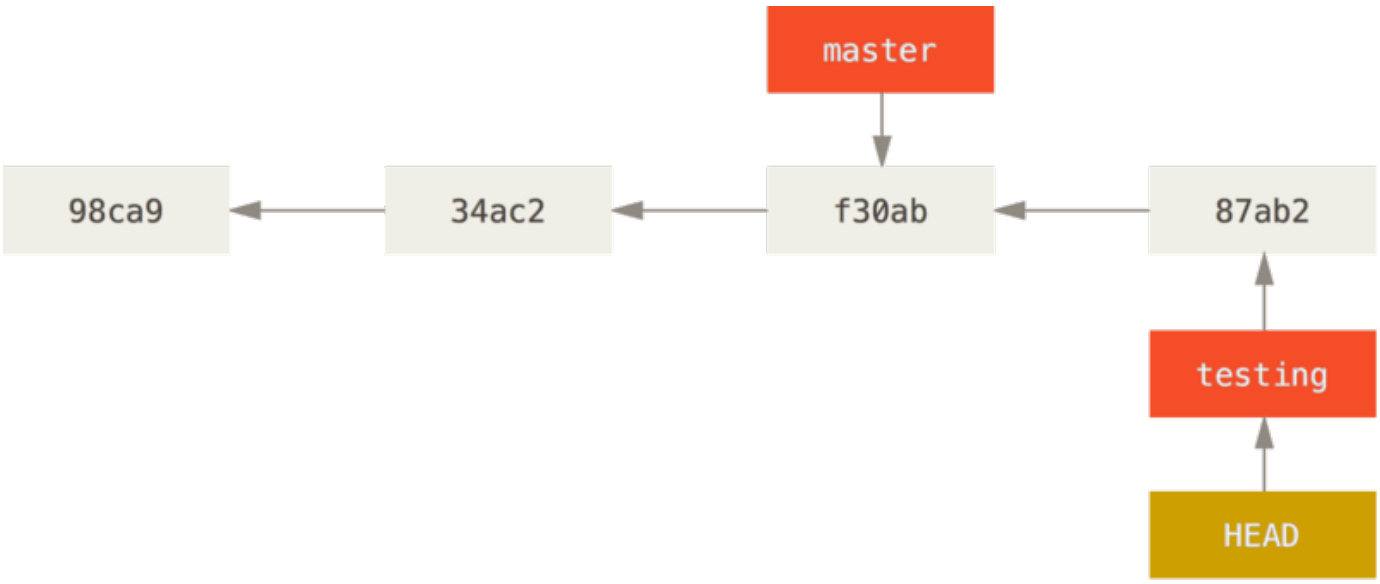


그림 15. HEAD가 가리키는 testing 브랜치가 새 커밋을 가리킴

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

```
$ git checkout main
```

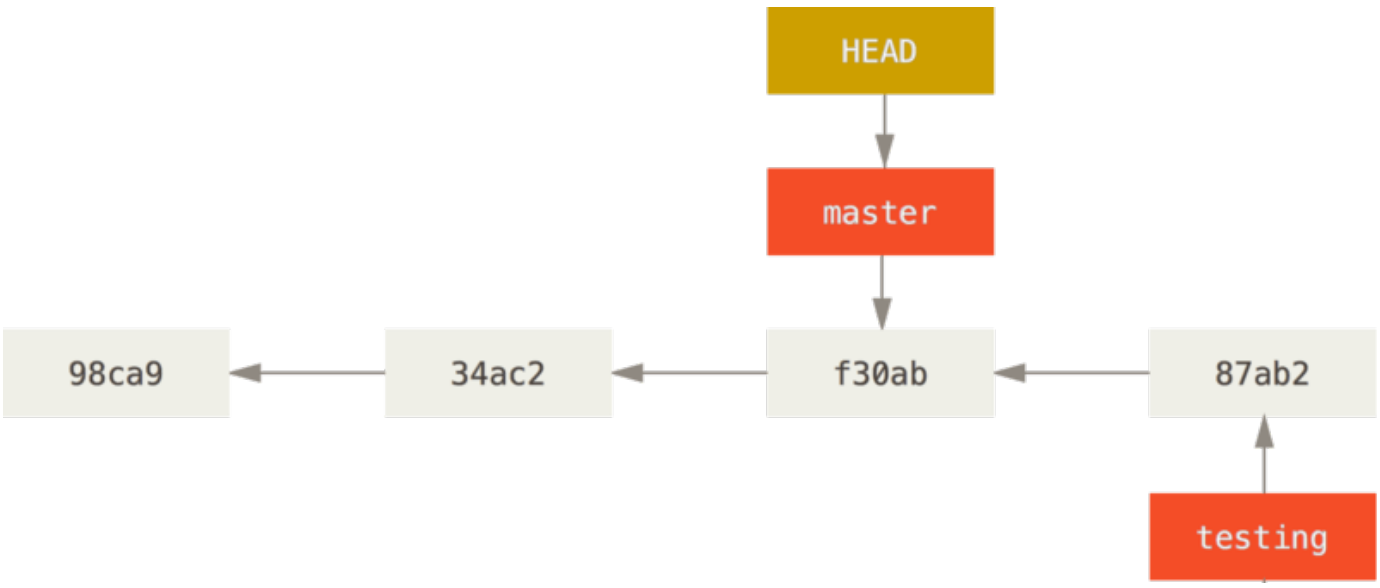


그림 16. HEAD가 Checkout 한 브랜치로 이동함

방금 실행한 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 불가능한 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 히스토리는 분리돼 진행한다(갈라지는 브랜치). 우리는 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 Merge 한다. 간단히 `branch`, `checkout`, `commit` 명령을 써서 말이다.

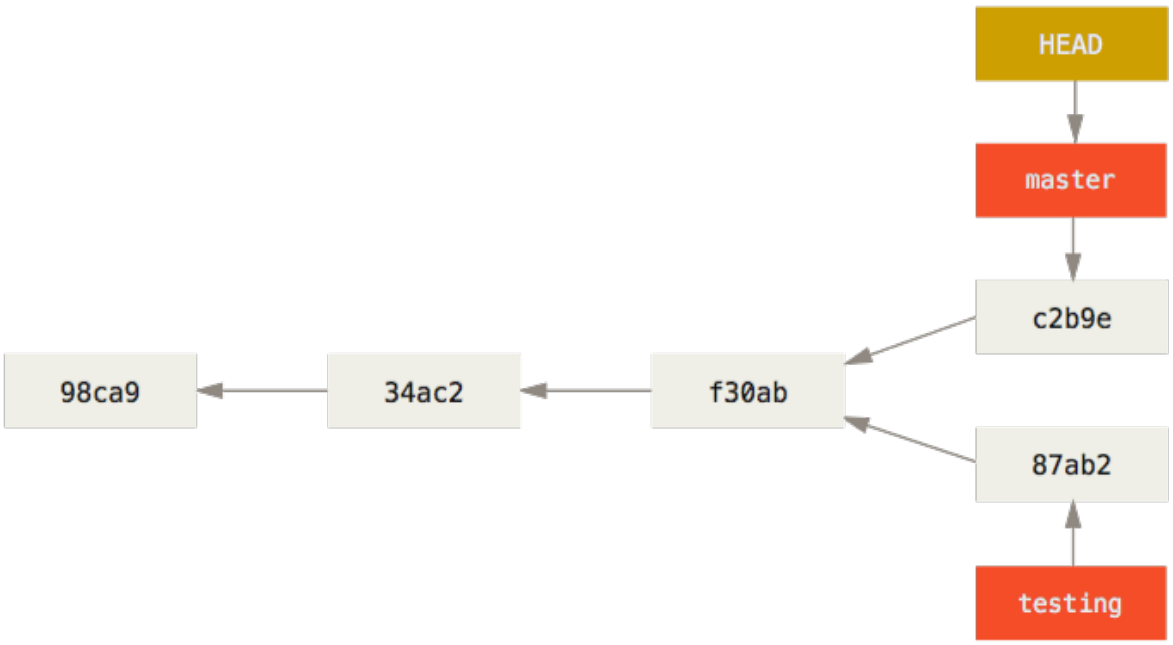


그림 17. 갈라지는 브랜치

`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.

3.2 Git 브랜치 - 브랜치와 Merge 의 기초

브랜치와 Merge 의 기초

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.

브랜치의 기초

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

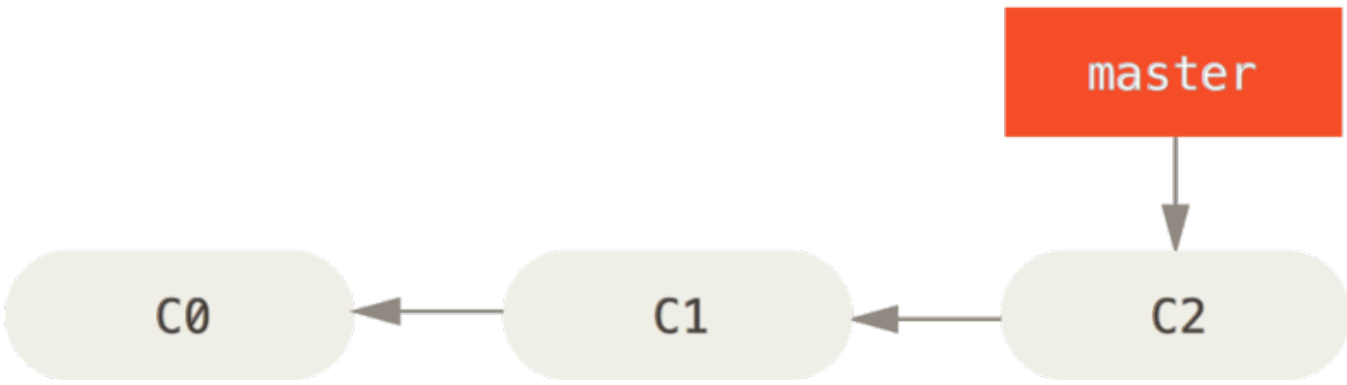


그림 18. 현재 커밋 히스토리

이슈 관리 시스템에 등록된 53번 이슈를 처리한다고 하면 이 이슈에 집중할 수 있는 브랜치를 새로 하나 만든다. 브랜치를 만들면서 Checkout까지 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
```

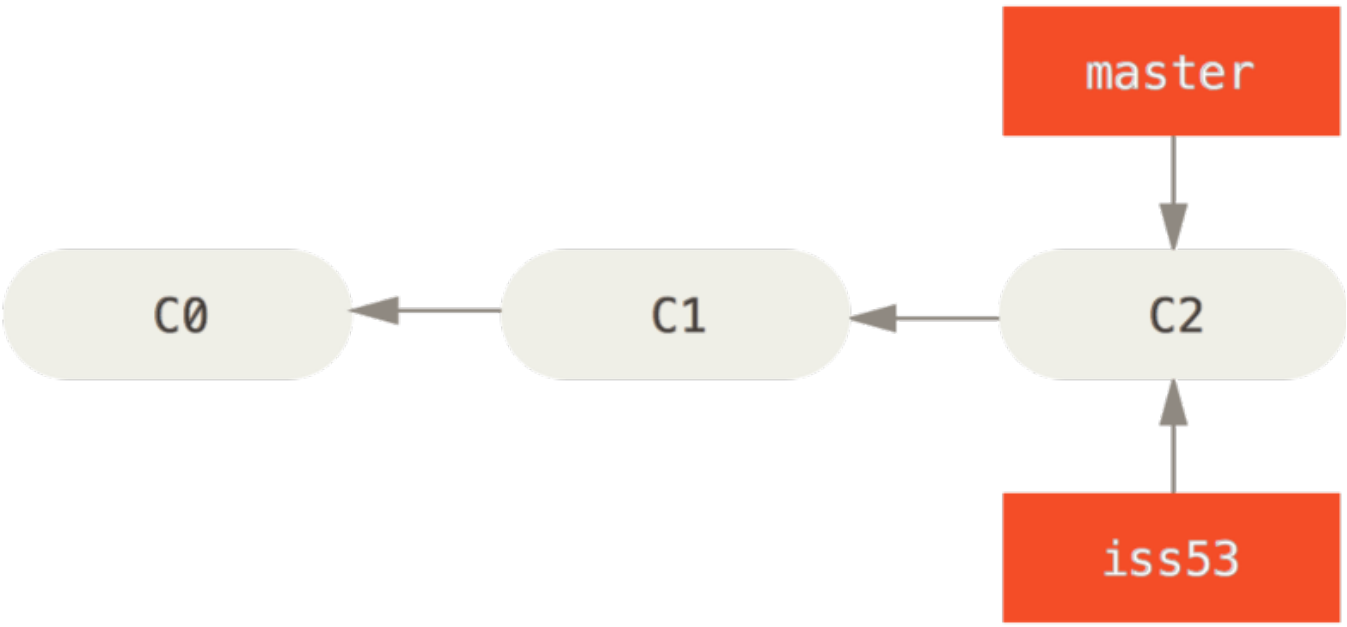


그림 19. 브랜치 포인터를 새로 만듦

`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
```

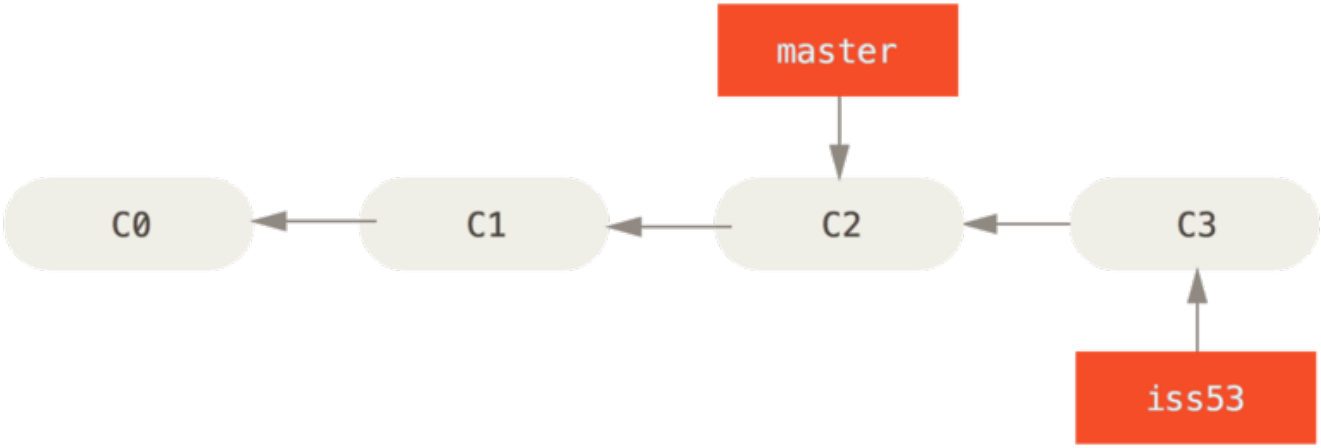


그림 20. 진행 중인 `iss53` 브랜치

다른 상황을 가정해보자. 만드는 사이트에 문제가 생겨서 즉시 고쳐야 한다. 버그를 해결한 Hotfix에 `iss53` 이 섞이는 것을 방지하기 위해 `iss53` 과 관련된 코드를 어딘가에 저장해두고 원래 운영 환경의 소스로 복구해야 한다. Git을 사용하면 이런 노력을 들일 필요 없이 그냥 `main` 브랜치로 돌아가면 된다.

그렇지만, 브랜치를 이동하려면 해야 할 일이 있다. 아직 커밋하지 않은 파일이 Checkout 할 브랜치와 충돌 나면 브랜치를 변경할 수 없다. 브랜치를 변경할 때는 워킹 디렉토리를 정리하는 것이 좋다. 이런 문제를 다루는 방법은(주로, Stash이나 커밋 Amend에 대해) 나중에 Stashing과 Cleaning 에서 다룰 것이다. 지금은 작업하던 것을 모두 커밋하고 `main` 브랜치로 옮긴다:

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

이때 워킹 디렉토리는 53번 이슈를 시작하기 이전 모습으로 되돌려지기 때문에 새로운 문제에 집중할 수 있는 환경이 만들어진다. Git은 자동으로 워킹 디렉토리에 파일들을 추가하고, 지우고, 수정해서 Checkout 한 브랜치의 마지막 스냅샷으로 되돌려 놓는다는 것을 기억해야 한다.

이젠 해결해야 할 핫픽스가 생겼을 때를 살펴보자. `hotfix` 라는 브랜치를 만들고 새로운 이슈를 해결할 때까지 사용한다.

```
$ 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
```

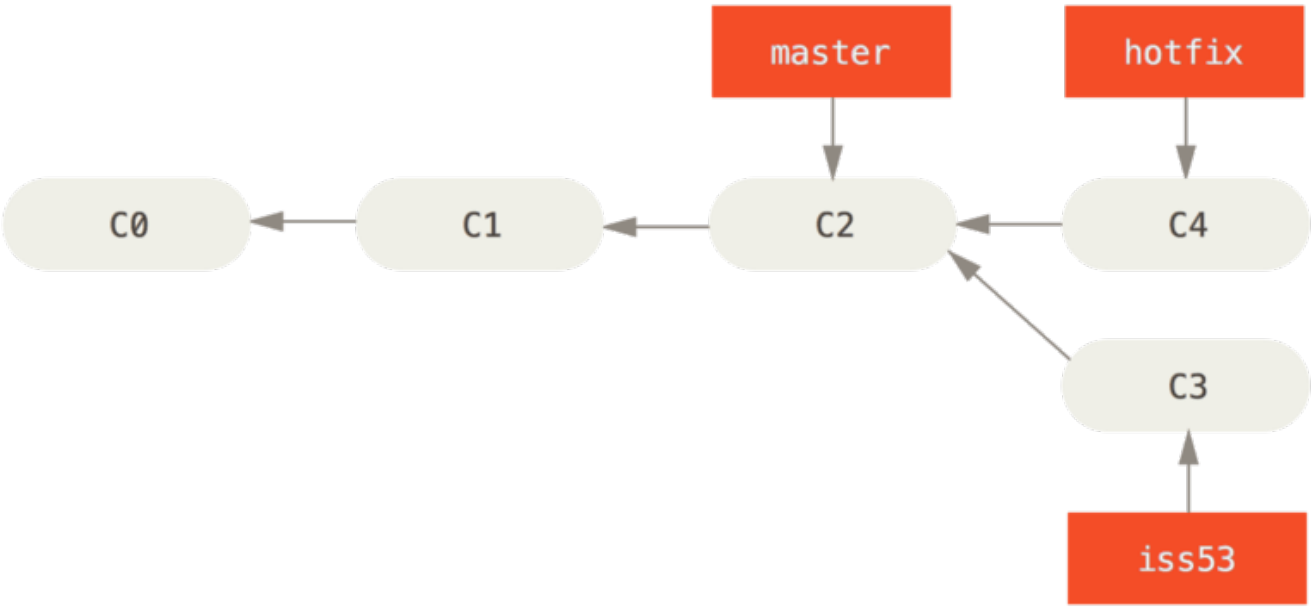


그림 21. `main` 브랜치에서 갈라져 나온 hotfix 브랜치

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. 다시 말해 A 브랜치에서 다른 B 브랜치를 Merge 할 때 B 브랜치가 A 브랜치 이후의 커밋을 가리키고 있으면 그저 A 브랜치가 B 브랜치와 동일한 커밋을 가리키도록 이동시킬 뿐이다.

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

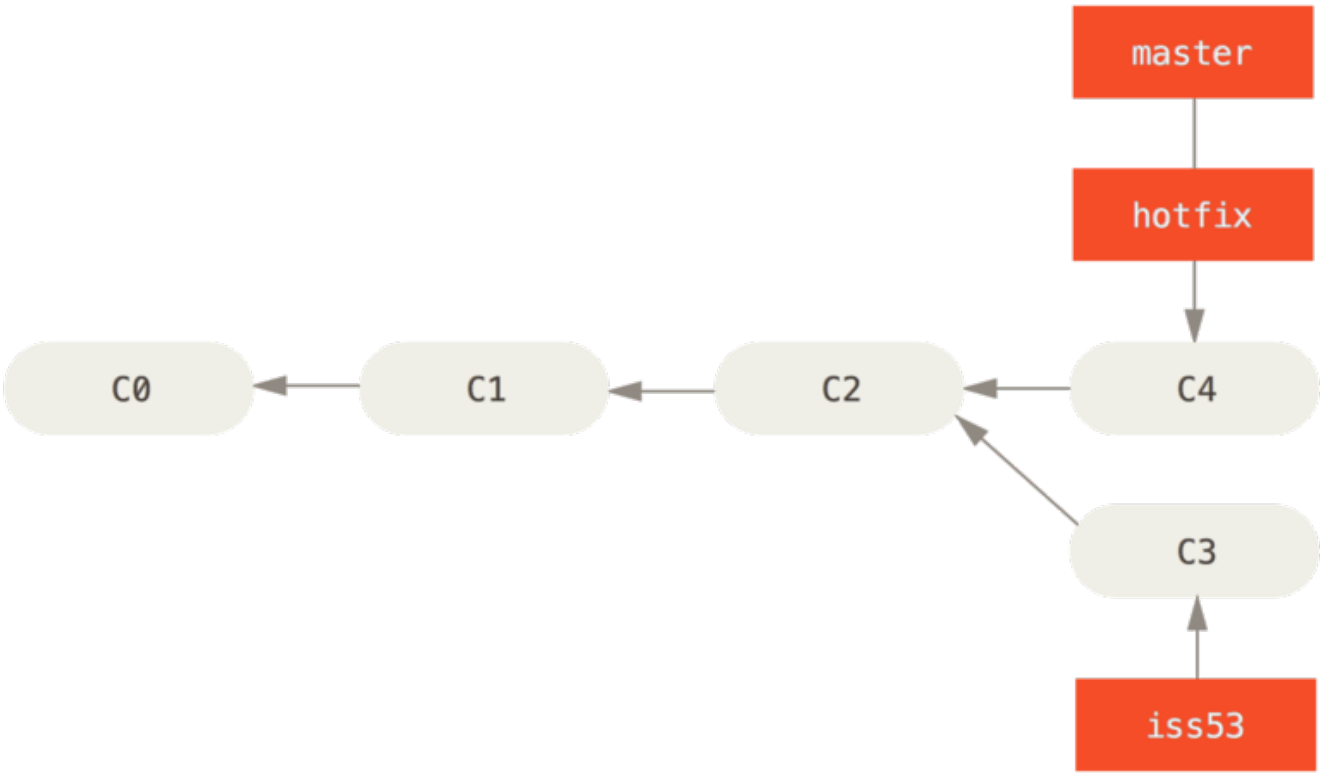


그림 22. Merge 후 `hotfix` 같은 것을 가리키는 `main` 브랜치

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).
```

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(-)
```

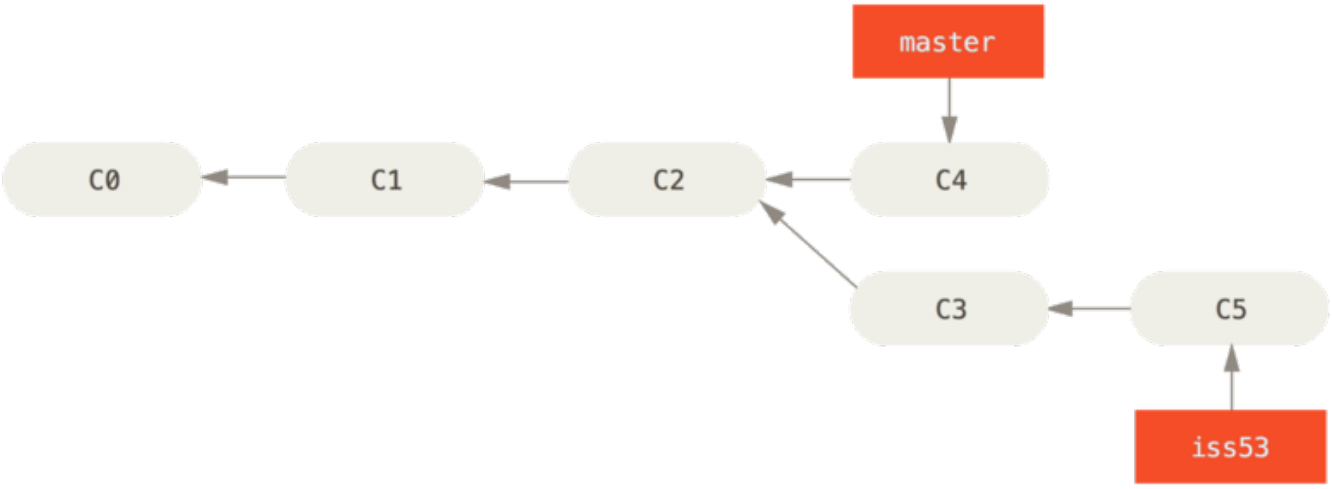


그림 23. main와 별개로 진행하는 iss53 브랜치

위에서 작업한 `hotfix` 가 `iss53` 브랜치에 영향을 끼치지 않는다는 점을 이해하는 것이 중요하다. `git merge main` 명령으로 `main` 브랜치를 `iss53` 브랜치에 Merge 하면 `iss53` 브랜치에 `hotfix` 가 적용된다. 아니면 `iss53` 브랜치가 `main` 에 Merge 할 수 있는 수준이 될 때까지 기다렸다가 Merge 하면 `hotfix` 와 `iss53` 브랜치가 합쳐진다.

Merge 의 기초

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.

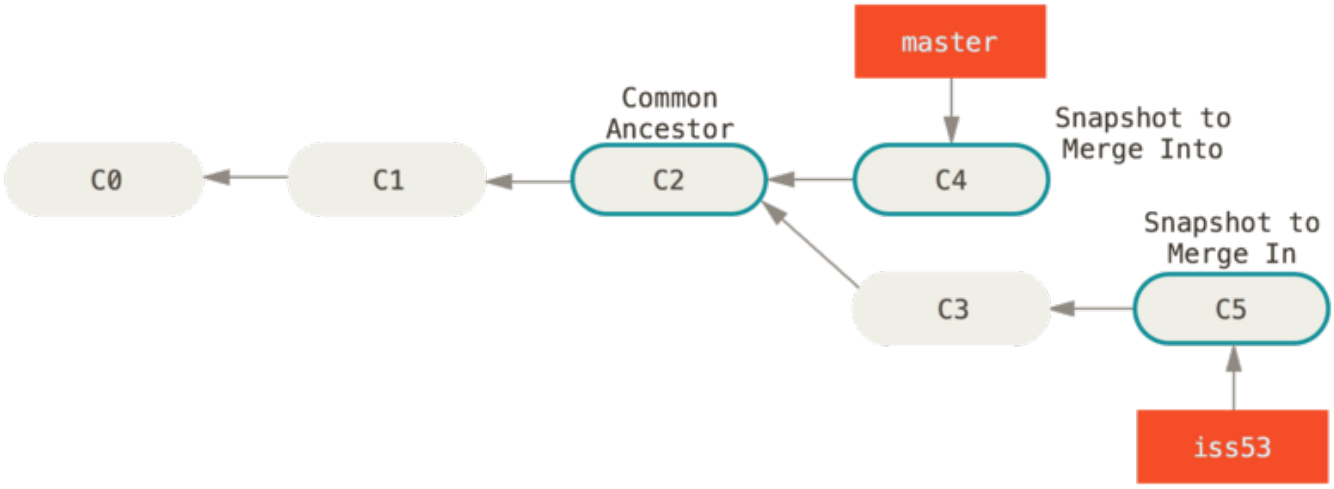


그림 24. 커밋 3개를 Merge

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.

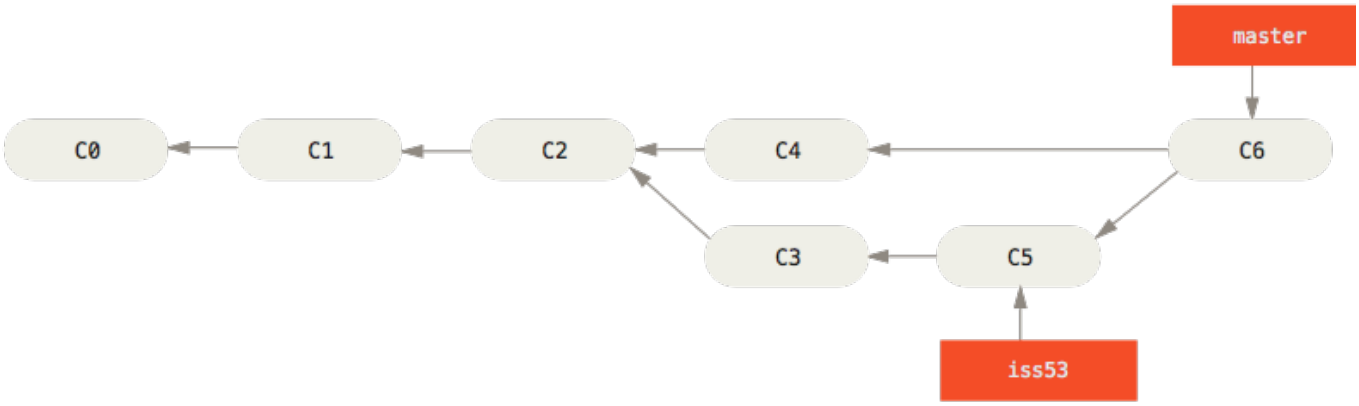


그림 25. Merge 커밋

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

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

충돌의 기초

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.
```

Git은 자동으로 Merge 하지 못해서 새 커밋이 생기지 않는다. 변경사항의 충돌을 개발자가 해결하지 않는 한 Merge 과정을 진행할 수 없다. 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.

다른 Merge 도구도 충돌을 해결할 수 있다. `git mergetool` 명령으로 실행한다.

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse
diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

기본 도구 말고 사용할 수 있는 다른 Merge 도구도 있는데(Mac에서는 `opendiff` 가 실행된다), “one of the following tools.” 부분에 보여준다. 여기에 표시된 도구 중 하나를 고를 수 있다.

노트	Merge 시에 발생한 충돌을 다루는 더 어렵고 요상한 내용은 뒤에 고급 Merge 에서 다루기로 한다.

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:

    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 기록한다. 자세한 기록은 나중에 이 Merge 커밋을 이해하는데 도움을 준다.

3.3 Git 브랜치 - 브랜치 관리

브랜치 관리

지금까지 브랜치를 만들고, Merge 하고, 삭제하는 방법에 대해서 살펴봤다. 브랜치를 관리하는 데 필요한 다른 명령도 살펴보자.

`git branch` 명령은 단순히 브랜치를 만들고 삭제하는 것이 아니다. 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`

3.4 Git 브랜치 - 브랜치 워크플로

브랜치 워크플로

브랜치를 만들고 Merge 하는 것을 어디에 써먹어야 할까. 이 절에서는 Git 브랜치가 유용한 몇 가지 워크플로를 살펴본다. 여기서 설명하는 워크플로를 개발에 적용하면 도움이 될 것이다.

Long-Running 브랜치

Git은 꼼꼼하게 3-way Merge를 사용하기 때문에 장기간에 걸쳐서 한 브랜치를 다른 브랜치와 여러 번 Merge 하는 것이 쉬운 편이다. 그래서 개발 과정에서 필요한 용도에 따라 브랜치를 만들어 두고 계속 사용할 수 있다. 그리고 정기적으로 브랜치를 다른 브랜치로 Merge 한다.

이런 접근법에 따라서 Git 개발자가 많이 선호하는 워크플로가 하나 있다. 배포했거나 배포할 코드만 `main` 브랜치에 Merge 해서 안정 버전의 코드만 `main` 브랜치에 둔다. 개발을 진행하고 안정화하는 브랜치는 `develop` 이나 `next` 라는 이름으로 추가로 만들어 사용한다. 이 브랜치는 언젠가 안정 상태가 되겠지만, 항상 안정 상태를 유지해야 하는 것이 아니다. 테스트를 거쳐서 안정적이라고 판단되면 `main` 브랜치에 Merge 한다.

토픽 브랜치(앞서 살펴본 `iss53` 브랜치 같은 짧은 호흡 브랜치)에도 적용할 수 있는데, 해당 토픽을 처리하고 테스트해서 버그도 없고 안정적이면 그때 Merge 한다.

사실 우리가 계속적으로 이야기 하는 것은 커밋을 가리키는 포인터에 대한 얘기다. 커밋 포인터를 만들고 수정하고 분리하고 합치는지에 대한 것이다. 개발 브랜치는 공격적으로 히스토리를 만들어 나아가고 안정 브랜치는 이미 만든 히스토리를 뒤따르며 나아간다.

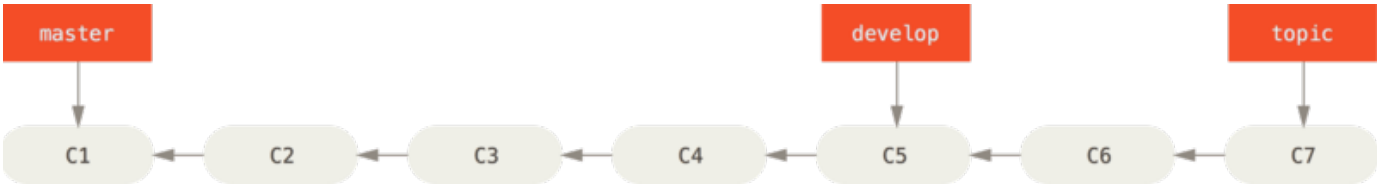


그림 26. 안정적인 브랜치일수록 커밋 히스토리가 뒤쳐짐

실험실에서 충분히 테스트하고 실전에 배치하는 과정으로 보면 이해하기 쉽다

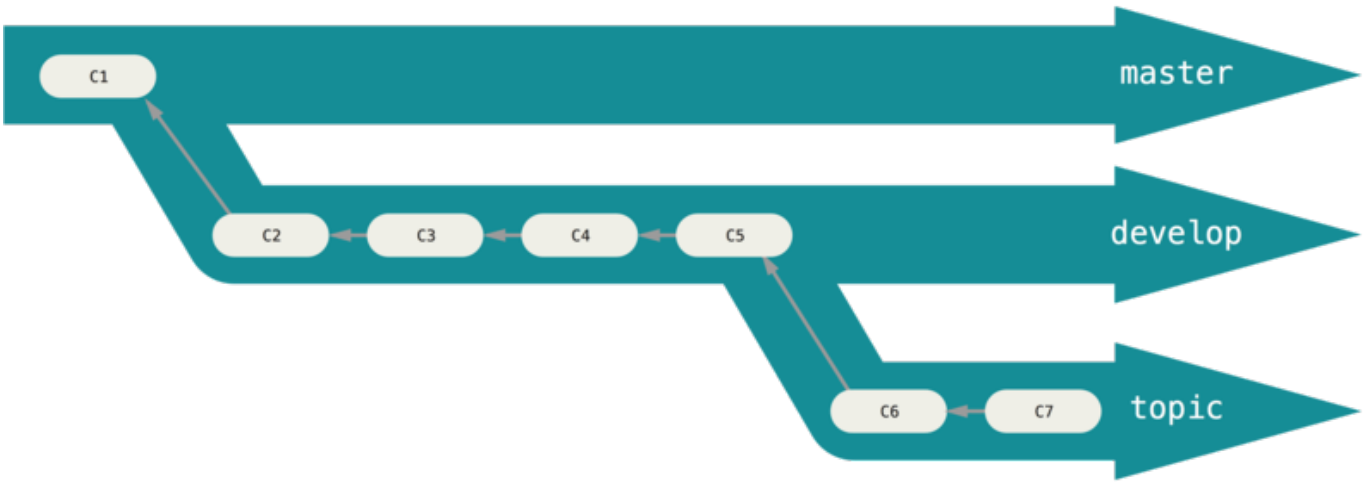


그림 27. 각 브랜치를 하나의 “실험실” 로 생각

코드를 여러 단계로 나누어 안정성을 높여가며 운영할 수 있다. 프로젝트 규모가 크면 `proposed` 혹은 `pu` (proposed updates)라는 이름의 브랜치를 만들고 `next` 나 `main` 브랜치에 아직 Merge 할 준비가 되지 않은 것을 일단 Merge 시킨다. 중요한 개념은 브랜치를 이용해 여러 단계에 걸쳐서 안정화해 나아가면서 충분히 안정화가 됐을 때 안정 브랜치로 Merge 한다는 점이다. 다시 말해서 Long-Running의 브랜치가 여러 개일 필요는 없지만 정말 유용하다는 점이다. 특히 규모가 크고 복잡한 프로젝트일수록 그 유용성이 좋다.

토픽 브랜치

토픽 브랜치는 프로젝트 크기에 상관없이 유용하다. 토픽 브랜치는 어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치다. 다른 버전 관리 시스템에서는 이런 브랜치를 본 적이 없을 것이다. Git이 아닌 다른 버전 관리 도구에서는 브랜치를 하나 만드는 데 큰 비용이 든다. Git에서는 매우 일상적으로 브랜치를 만들고 Merge 하고 삭제한다.

앞서 사용한 `iss53` 이나 `hotfix` 브랜치가 토픽 브랜치다. 우리는 브랜치를 새로 만들고 어느 정도 커밋하고 나서 다시 `main` 브랜치에 Merge 하고 브랜치 삭제도 해 보았다. 보통 주제별로 브랜치를 만들고 각각은 독립돼 있기 때문에 매우 쉽게 컨텍스트 사이를 옮겨 다닐 수 있다. 묶음별로 나눠서 일하면 내용별로 검토하기에도, 테스트하기에도 더 편하다. 각 작업을 하루든 한 달이든 유지하다가 `main` 브랜치에 Merge 할 시점이 되면 순서에 관계없이 그때 Merge 하면 된다.

`main` 브랜치를 checkout 한 상태에서 어떤 작업을 한다고 해보자. 한 이슈를 처리하기 위해서 `iss91` 브랜치를 만들고 해당 작업을 한다. 같은 이슈를 다른 방법으로 해결해보고 싶을 때도 있다. `iss91v2` 브랜치를 만들고 다른 방법을 시도해 본다. 확신할 수 없는 아이디어를 적용해보기 위해 다시 `main` 브랜치로 되돌아가서 `dumbidea` 브랜치를 하나 더 만든다. 지금까지 말했던 커밋 히스토리는 아래 그림 같다.

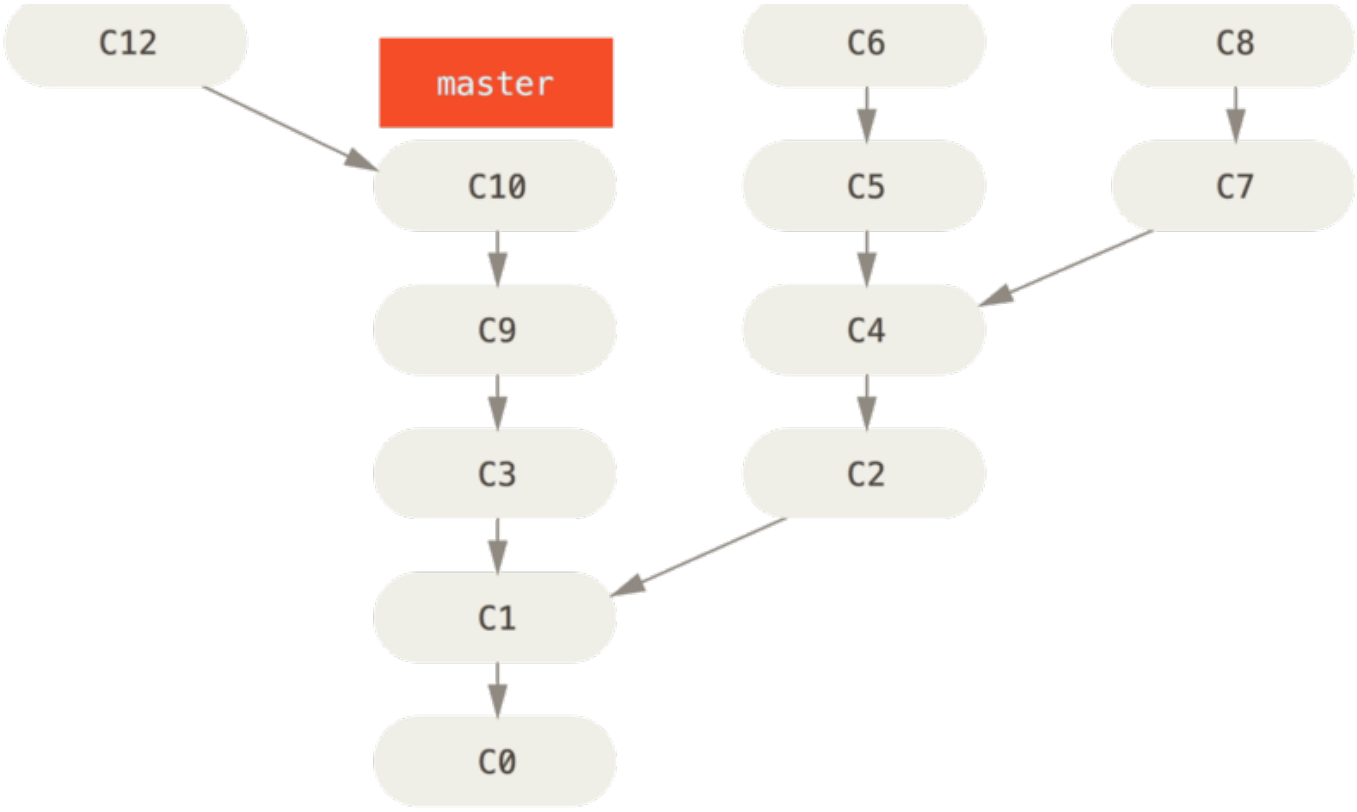
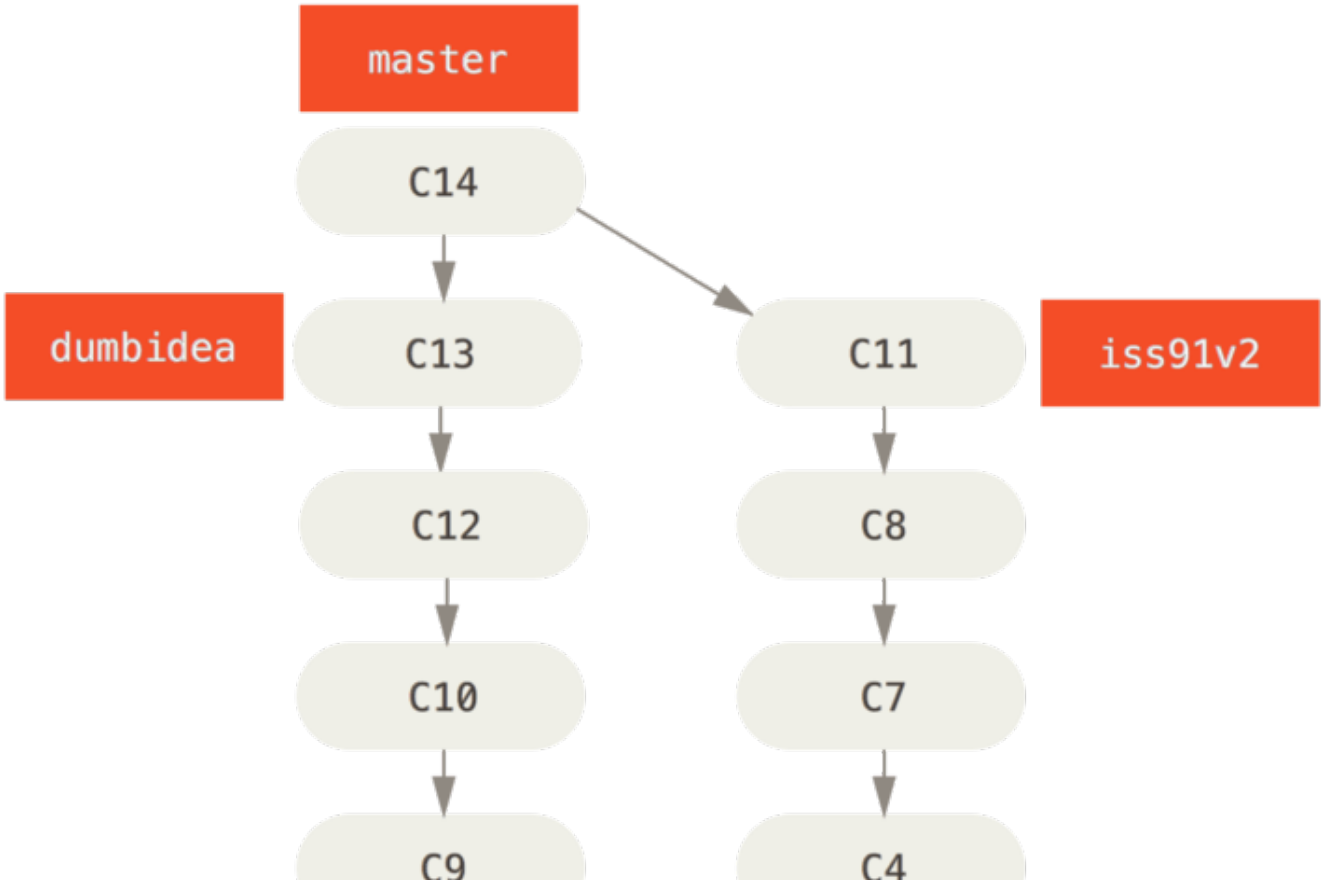


그림 28. 토픽 브랜치가 많음

이슈를 처리했던 방법 중 두 번째 방법인 `iss91v2` 브랜치가 괜찮아서 적용하기로 결정했다. 그리고 아이디어를 확신할 수 없었던 `dumbidea` 브랜치를 같이 일하는 다른 개발자에게 보여줬더니 썩 괜찮다는 반응을 얻었다. `iss91` 브랜치는 (C5, C6 커밋도 함께) 버리고 다른 두 브랜치를 Merge 하면 아래 그림과 같이 된다.



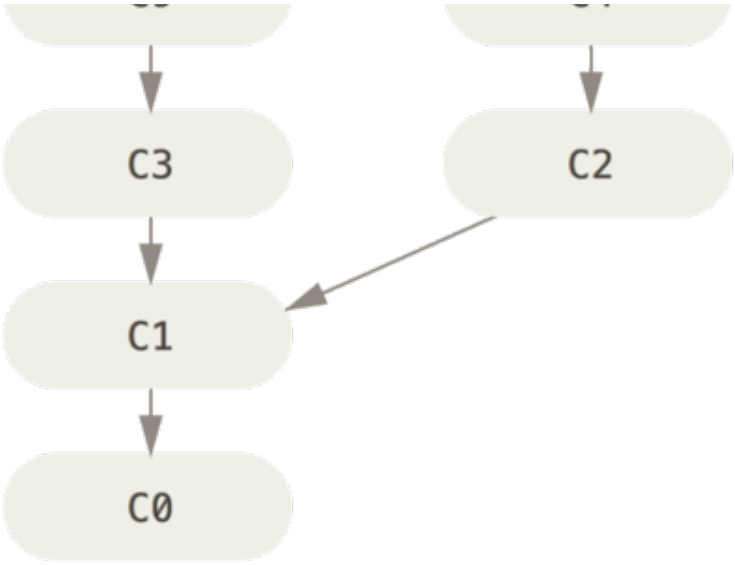


그림 29. `dumbidea` 와 `iss91v2` 브랜치를 Merge 하고 난 후의 모습

분산 환경에서의 Git에서 프로젝트를 Git으로 관리할 때 브랜치를 이용하여 만들 수 있는 여러 워크플로에 대해 살펴본다. 관련 부분을 살펴보면 프로젝트에 어떤 형태로 응용할수 있을 지 감이 올 것이다.

지금까지 한 작업은 전부 로컬에서만 처리한다는 것을 꼭 기억하자. 로컬 저장소에서만 브랜치를 만들고 Merge 했으며 서버와 통신을 주고받는 일은 없었다.

3.5 Git 브랜치 - 리모트 브랜치

리모트 브랜치

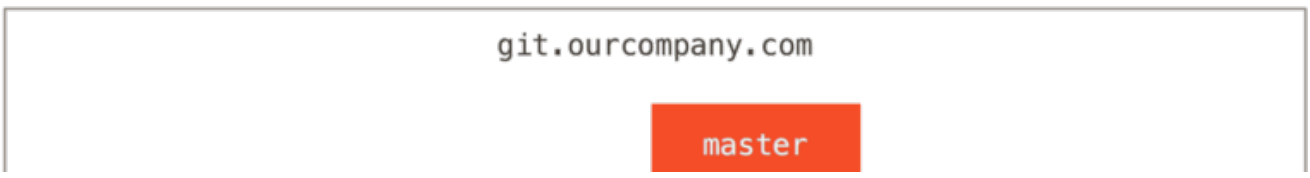
리모트 Refs는 리모트 저장소에 있는 포인터인 레퍼런스다. 리모트 저장소에 있는 브랜치, 태그, 등등을 의미한다. `git ls-remote [remote]` 명령으로 모든 리모트 Refs를 조회할 수 있다. `git remote show [remote]` 명령은 모든 리모트 브랜치와 그 정보를 보여준다. 리모트 Refs가 있지만 보통은 리모트 트래킹 브랜치를 사용한다.

리모트 트래킹 브랜치는 리모트 브랜치를 추적하는 레퍼런스이며 브랜치다. 리모트 트래킹 브랜치는 로컬에 있지만 임의로 움직일 수 없다. 리모트 서버에 연결할 때마다 리모트의 브랜치 업데이트 내용에 따라서 자동으로 갱신될 뿐이다. 리모트 트래킹 브랜치는 일종의 북마크라고 할 수 있다. 리모트 저장소에 마지막으로 연결했던 순간에 브랜치가 무슨 커밋을 가리키고 있었는지를 나타낸다.

리모트 트래킹 브랜치의 이름은 `<remote>/<branch>` 형식으로 되어 있다. 예를 들어 리모트 저장소 `origin` 의 `main` 브랜치를 보고 싶다면 `origin/main` 라는 이름으로 브랜치를 확인하면 된다. 다른 팀원과 함께 어떤 이슈를 구현할 때 그 팀원이 `iss53` 브랜치를 서버로 Push 했고 당신도 로컬에 `iss53` 브랜치가 있다고 가정하자. 이때 서버의 `iss53` 브랜치가 가리키는 커밋은 로컬에서 `origin/iss53` 이 가리키는 커밋이다.

다소 헷갈릴 수 있으니 예제를 좀 더 살펴보자. `git.ourcompany.com` 이라는 Git 서버가 있고 이 서버의 저장소를 하나 Clone 하면 Git은 자동으로 `origin` 이라는 이름을 붙인다. `origin` 으로부터 저장소 데이터를 모두 내려받고 `main` 브랜치를 가리키는 포인터를 만든다. 이 포인터는 `origin/main` 라고 부르고 멋대로 조종할 수 없다. 그리고 Git은 로컬의 `main` 브랜치가 `origin/main` 를 가리키게 한다. 이제 이 `main` 브랜치에서 작업을 시작할 수 있다.

노트	"origin" 의 의미 브랜치 이름으로 많이 사용하는 "main" 라는 이름이 괜히 특별한 의미를 가지는 게 아닌 것처럼 "origin" 도 특별한 의미가 있는 것은 아니다. `git init` 명령이 자동으로 만들기 때문에 사용하는 이름인 "main" 와 마찬가지로 "origin" 도 `git clone` 명령이 자동으로 만들어주는 리모트 이름이다. `git clone -o booyah` 라고 옵션을 주고 명령을 실행하면 `booyah/main` 라고 사용자가 정한 대로 리모트 이름을 생성해준다.



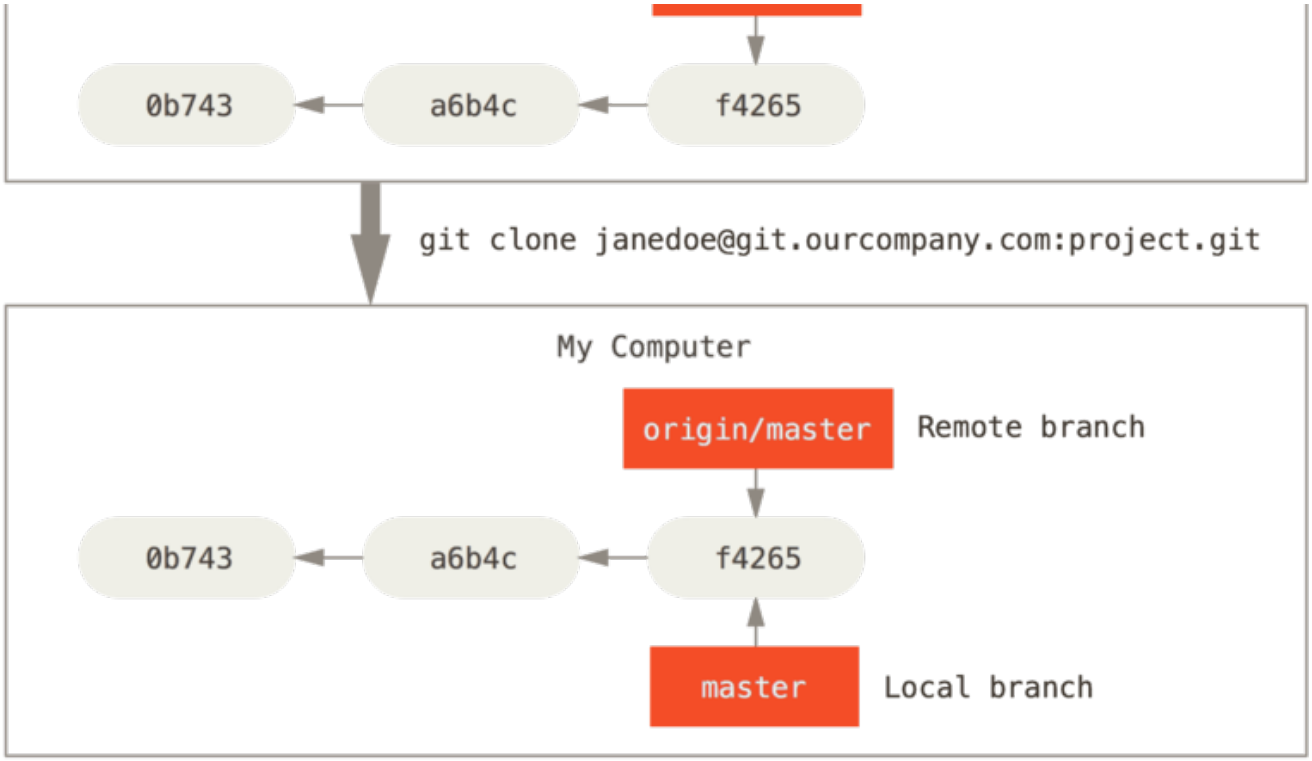


그림 30. Clone 이후 서버와 로컬의 main 브랜치

로컬 저장소에서 어떤 작업을 하고 있는데 동시에 다른 팀원이 `git.ourcompany.com` 서버에 Push 하고 `main` 브랜치를 업데이트한다. 그러면 이제 팀원 간의 히스토리는 서로 달라진다. 서버 저장소로부터 어떤 데이터도 주고받지 않아서 `origin/main` 포인터는 그대로다.

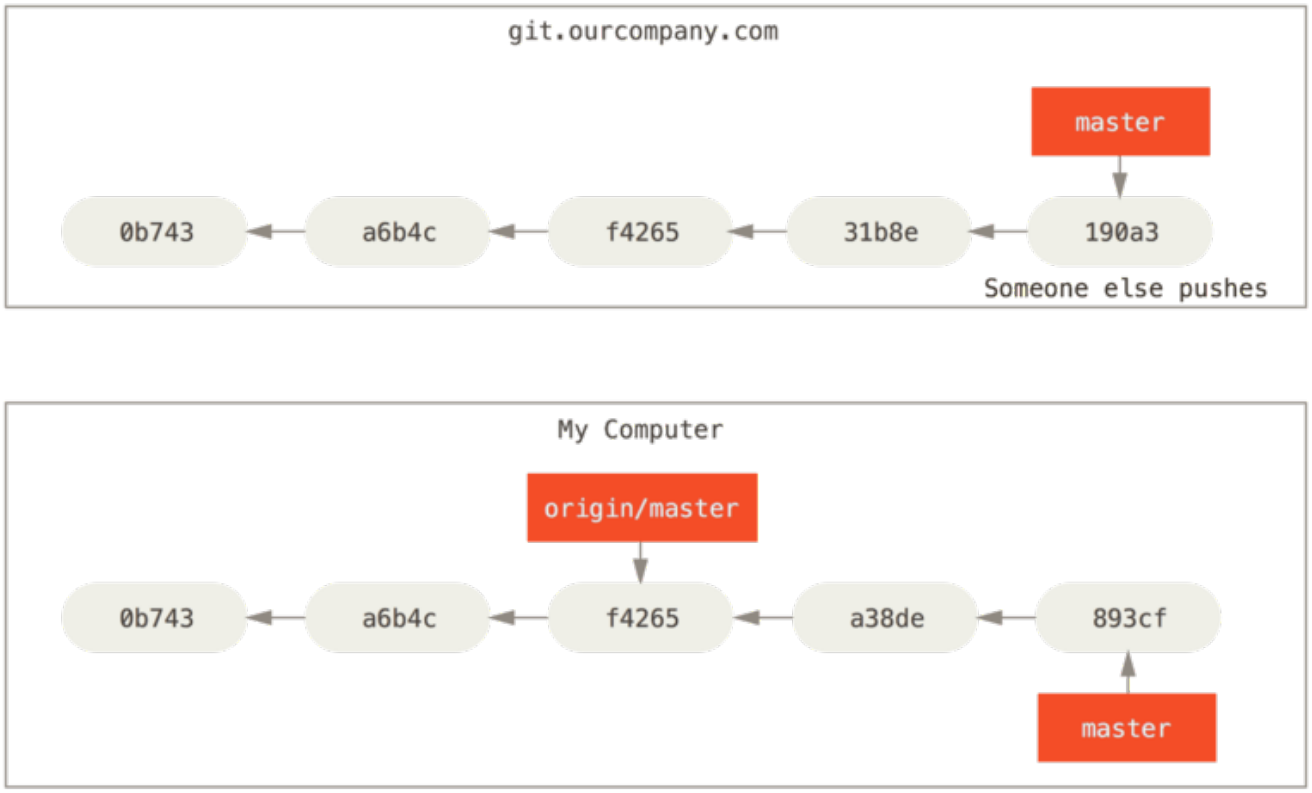


그림 31. 로컬과 서버의 커밋 히스토리는 독립적임

리모트 서버로부터 저장소 정보를 동기화하려면 `git fetch origin` 명령을 사용한다. 명령을 실행하면 우선 “origin” 서버의 주소 정보(이 예에서는 `git.ourcompany.com`)를 찾아서, 현재 로컬의 저장소가 갖고 있지 않은 새로운 정보가 있으면 모두 내려받고, 받은 데이터를 로컬 저장소에 업데이트하고 나서, origin/main 포인터의 위치를 최신 커밋으로 이동시킨다.

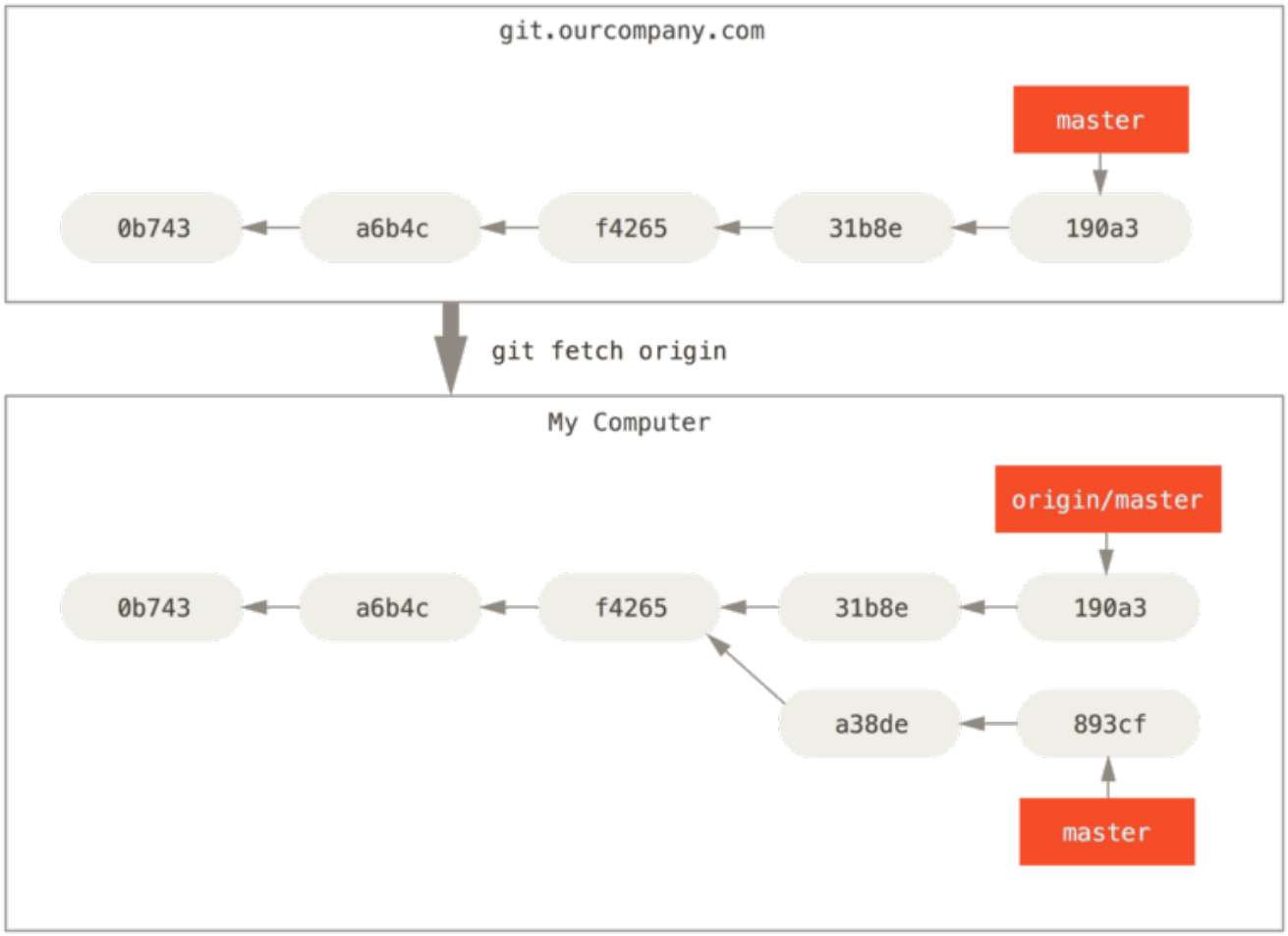


그림 32. `git fetch` 명령은 리모트 브랜치 정보를 업데이트

Push 하기

로컬의 브랜치를 서버로 전송하려면 쓰기 권한이 있는 리모트 저장소에 Push 해야 한다. 로컬 저장소의 브랜치는 자동으로 리모트 저장소로 전송되지 않는다. 명시적으로 브랜치를 Push 해야 정보가 전송된다. 따라서 리모트 저장소에 전송하지 않고 로컬 브랜치에만 두는 비공개 브랜치를 만들 수 있다. 또 다른 사람과 협업하기 위해 토픽 브랜치만 전송할 수도 있다.

`serverfix` 라는 브랜치를 다른 사람과 공유할 때도 브랜치를 처음 Push 하는 것과 같은 방법으로 Push 한다. 아래와 같이 `git push <remote> <branch>` 명령을 사용한다.

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Git은 serverfix 라는 브랜치 이름을 `refs/heads/serverfix:refs/heads/serverfix` 로 확장한다. 이것은 `serverfix` 라는 로컬 브랜치를 서버로 Push 하는데 리모트의 `serverfix` 브랜치로 업데이트한다는 것을 의미한다.

`git push origin serverfix:serverfix` 라고 Push 하는 것도 같은 의미인데 이것은 "로컬의 serverfix 브랜치를 리모트 저장소의 serverfix 브랜치로 Push 하라" 라는 뜻이다.

로컬 브랜치의 이름과 리모트 서버의 브랜치 이름이 다를 때 필요하다. 리모트 저장소에 `serverfix` 라는 이름 대신 다른 이름을 사용하려면 `git push origin serverfix:awesomebranch` 처럼 사용한다.

노트	암호를 매번 입력하지 않아도 된다 HTTPS URL로 시작하는 리모트 저장소를 사용한다면 아마도 Push 나 Pull 을 할 때 인증을 위한 사용자이름이나 암호를 묻는 것을 볼 수 있다. 보통 터미널에서 작업하는 경우 Git이 이 정보를 사용자로부터 받기 위해 사용자이름이나 암호를 입력받아 서버로 전달해서 권한을 확인한다. 이 리모트에 접근할 때마다 매번 사용자이름나 암호를 입력하지 않도록 "credential cache" 기능을 이용할 수 있다. 이 기능을 활성화하면 Git은 몇 분 동안 입력한 사용자이름이나 암호를 저장해둔다. 이 기능을 활성화하려면 `git config --global credential.helper cache` 명령을 실행하여 환경설정을 추가한다. 이 기능이 제공하는 다른 옵션에 대한 자세한 설명은 Credential 저장소를 참고한다.

나중에 누군가 저장소를 Fetch 하고 나서 서버에 있는 `serverfix` 브랜치에 접근할 때 `origin/serverfix` 라는 이름으로 접근할 수 있다.

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Fetch 명령으로 리모트 트래킹 브랜치를 내려받는다고 해서 로컬 저장소에 수정할 수 있는 브랜치가 새로 생기는 것이 아니다. 다시 말해서 `serverfix` 라는 브랜치가 생기는 것이 아니라 그저 수정 못 하는 `origin/serverfix` 브랜치 포인터가 생기는 것이다.

새로 받은 브랜치의 내용을 Merge 하려면 `git merge origin/serverfix` 명령을 사용한다. Merge 하지 않고 리모트 트래킹 브랜치에서 시작하는 새 브랜치를 만들려면 아래와 같은 명령을 사용한다.

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

그러면 `origin/serverfix` 에서 시작하고 수정할 수 있는 `serverfix` 라는 로컬 브랜치가 만들어진다.

브랜치 추적

리모트 트래킹 브랜치를 로컬 브랜치로 Checkout 하면 자동으로 "트래킹(Tracking) 브랜치" 가 만들어진다 (트래킹 하는 대상 브랜치를 "Upstream 브랜치" 라고 부른다). 트래킹 브랜치는 리모트 브랜치와 직접적인 연결고리가 있는 로컬 브랜치이다. 트래킹 브랜치에서 `git pull` 명령을 내리면 리모트 저장소로부터 데이터를 내려받아 연결된 리모트 브랜치와 자동으로 Merge 한다.

서버로부터 저장소를 Clone을 하면 Git은 자동으로 `main` 브랜치를 `origin/main` 브랜치의 트래킹 브랜치로 만든다. 트래킹 브랜치를 직접 만들 수 있는데 리모트를 `origin` 이 아닌 다른 리모트로 할 수도 있고, 브랜치도 `main` 가 아닌 다른 브랜치로 추적하게 할 수 있다. `git checkout -b <branch> <remote>/<branch>` 명령으로 간단히 트래킹 브랜치를 만들 수 있다. `--track` 옵션을 사용하여 로컬 브랜치 이름을 자동으로 생성할 수 있다.

```
```

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

이 명령은 매우 자주 쓰여서 더 생략할 수 있다. 입력한 브랜치가 있는 (a) 리모트가 딱 하나 있고 (b) 로컬에는 없으면 Git은 자동으로 트래킹 브랜치를 만들어 준다.

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

리모트 브랜치와 다른 이름으로 브랜치를 만들려면 로컬 브랜치의 이름을 아래와 같이 다르게 지정한다.

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

이제 `sf` 브랜치에서 Push 나 Pull 하면 자동으로 `origin/serverfix` 로 데이터를 보내거나 가져온다.

이미 로컬에 존재하는 브랜치가 리모트의 특정 브랜치를 추적하게 하려면 `git branch` 명령에 `-u` 나 `--set-upstream-to` 옵션을 붙여서 아래와 같이 설정한다.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

노트	Upstream 별명 추적 브랜치를 설정했다면 추적 브랜치 이름을 `@{upstream}` 이나 `@{u}` 로 짧게 대체하여 사용할 수 있다. `main` 브랜치가 `origin/main` 브랜치를 추적하는 경우라면 `git merge origin/main` 명령과 `git merge @{u}` 명령을 똑같이 사용할 수 있다.

추적 브랜치가 현재 어떻게 설정되어 있는지 확인하려면 `git branch` 명령에 `-vv` 옵션을 더한다. 이 명령을 실행하면 로컬 브랜치 목록과 로컬 브랜치가 추적하고 있는 리모트 브랜치도 함께 보여준다.

그리고 로컬 브랜치가 앞서가는지 뒤쳐지는지에 대한 내용도 보여준다.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  main    1ae2a45 [origin/main] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this
should do it
  testing   5ea463a trying something new
```

위의 결과를 보면 `iss53` 브랜치는 `origin/iss53` 리모트 브랜치를 추적하고 있다는 것을 알 수 있고 “ahead” 표시를 통해 로컬 브랜치가 커밋 2개 앞서 있다 (리모트 브랜치에는 없는 커밋이 로컬에는 존재) 는 것을 알 수 있다.

`main` 브랜치는 `origin/main` 브랜치를 추적하고 있으며 두 브랜치가 가리키는 커밋 내용이 같은 상태이다.

로컬 브랜치 중 `serverfix` 브랜치는 `server-fix-good` 이라는 `teamone` 리모트 서버의 브랜치를 추적하고 있으며 커밋 3개 앞서 있으며 동시에 커밋 1개로 뒤쳐져 있다.

이 말은 `serverfix` 브랜치에 서버로 보내지 않은 커밋이 3개, 서버의 브랜치에서 아직 로컬 브랜치로 머지하지 않은 커밋이 1개 있다는 말이다. 마지막 `testing` 브랜치는 추적하는 브랜치가 없는 상태이다.

여기서 중요한 점은 명령을 실행했을 때 나타나는 결과는 모두 마지막으로 서버에서 데이터를 가져온(fetch) 시점을 바탕으로 계산한다는 점이다. 단순히 이 명령만으로는 서버의 최신 데이터를 반영하지는 않으며 로컬에 저장된 서버의 캐시 데이터를 사용한다.

현재 시점에서 진짜 최신 데이터로 추적 상황을 알아보려면 먼저 서버로부터 최신 데이터를 받아온 후에 추적 상황을 확인해야 한다. 아래처럼 두 명령을 이어서 사용하는 것이 적당하다 하겠다.

```
$ git fetch --all; git branch -vv
```

Pull 하기

`git fetch` 명령을 실행하면 서버에는 존재하지만, 로컬에는 아직 없는 데이터를 받아와서 저장한다. 이 때 워킹 디렉토리의 파일 내용은 변경되지 않고 그대로 남는다. 서버로부터 데이터를 가져와서 저장해두고 사용자가 Merge 하도록 준비만 해둔다. 간단히 말하면 `git pull` 명령은 대부분 `git fetch` 명령을 실행하고 나서 자동으로 `git merge` 명령을 수행하는 것 뿐이다. 바로 앞 절에서 살펴본 대로 `clone` 이나 `checkout` 명령을 실행하여 추적 브랜치가 설정되면 `git pull` 명령은 서버로부터 데이터를 가져와서 현재 로컬 브랜치와 서버의 추적 브랜치를 Merge 한다.

일반적으로 `fetch` 와 `merge` 명령을 명시적으로 사용하는 것이 `pull` 명령으로 한번에 두 작업을 하는 것보다 낫다.

리모트 브랜치 삭제

동료와 협업하기 위해 리모트 브랜치를 만들었다가 작업을 마치고 `main` 브랜치로 Merge 했다. 협업하는 데 사용했던 그 리모트 브랜치는 이제 더 이상 필요하지 않기에 삭제할 수 있다. `git push` 명령에 `--delete` 옵션을 사용하여 리모트 브랜치를 삭제할 수 있다. `serverfix` 라는 리모트 브랜치를 삭제하려면 아래와 같이 실행한다.

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

위 명령을 실행하면 서버에서 브랜치(즉 커밋을 가리키는 포인터) 하나가 사라진다. 서버에서 가비지 컬렉터가 동작하지 않는 한 데이터는 사라지지 않기 때문에 종종 의도치 않게 삭제한 경우에도 커밋한 데이터를 살릴 수 있다.

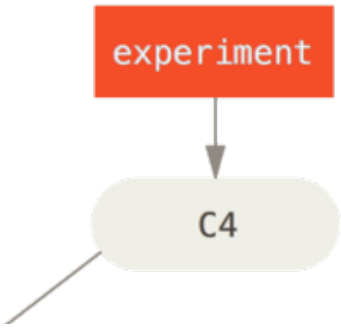
3.6 Git 브랜치 - Rebase 하기

Rebase 하기

Git에서 한 브랜치에서 다른 브랜치로 합치는 방법으로는 두 가지가 있다. 하나는 Merge 이고 다른 하나는 Rebase 다. 이 절에서는 Rebase가 무엇인지, 어떻게 사용하는지, 좋은 점은 뭐고, 어떤 상황에서 사용하고 어떤 상황에서 사용하지 말아야 하는지 알아 본다.

Rebase 의 기초

앞의 Merge 의 기초 절에서 살펴본 예제로 다시 돌아가 보자. 두 개의 나누어진 브랜치의 모습을 볼 수 있다.



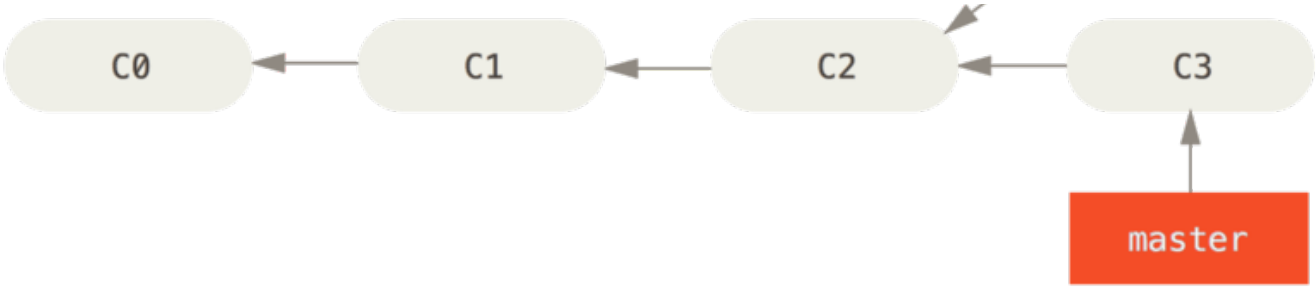


그림 35. 두 개의 브랜치로 나누어진 커밋 히스토리

이 두 브랜치를 합치는 가장 쉬운 방법은 앞에서 살펴본 대로 `merge` 명령을 사용하는 것이다. 두 브랜치의 마지막 커밋 두 개(C3, C4)와 공통 조상(C2)을 사용하는 3-way Merge로 새로운 커밋을 만들어 낸다.

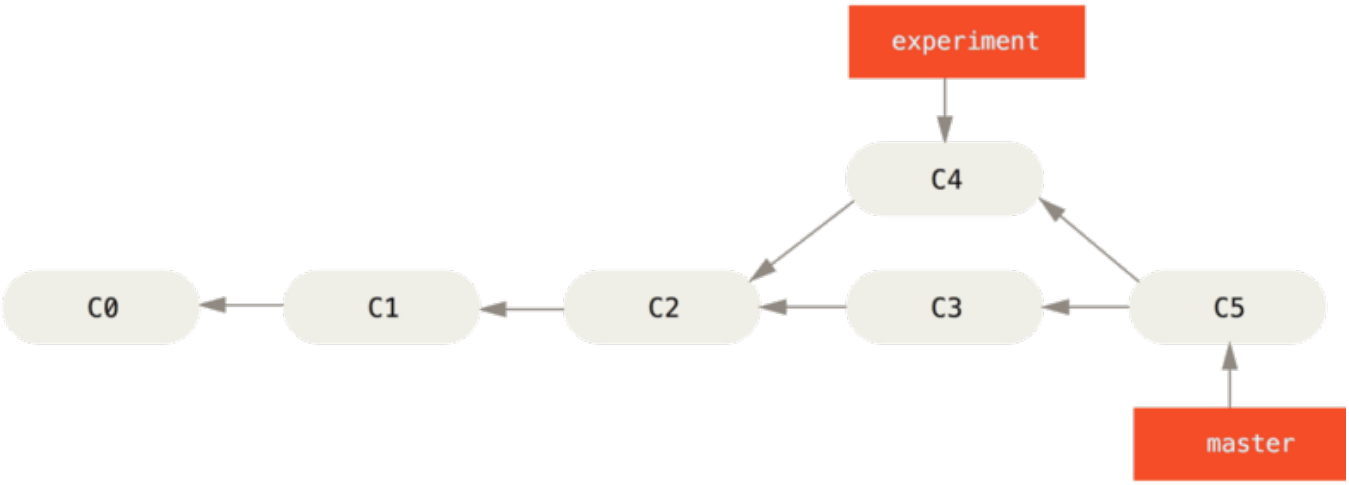


그림 36. 나뉜 브랜치를 Merge 하기

비슷한 결과를 만드는 다른 방식으로, C3 에서 변경된 사항을 Patch로 만들고 이를 다시 C4 에 적용시키는 방법이 있다. Git에서는 이런 방식을 *Rebase* 라고 한다. `rebase` 명령으로 한 브랜치에서 변경된 사항을 다른 브랜치에 적용할 수 있다.

위의 예제는 아래와 같은 명령으로 Rebase 한다.

```
$ git checkout experiment
$ git rebase main
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

실제로 일어나는 일을 설명하자면 일단 두 브랜치가 나뉘기 전인 공통 커밋으로 이동하고 나서 그 커밋부터 지금 Checkout 한 브랜치가 가리키는 커밋까지 diff를 차례로 만들어 어딘가에 임시로 저장해 놓는다. Rebase 할 브랜치(experiment)가 합칠 브랜치(main)가 가리키는 커밋을 가리키게 하고 아까 저장해 놓았던 변경사항을 차례대로 적용한다.

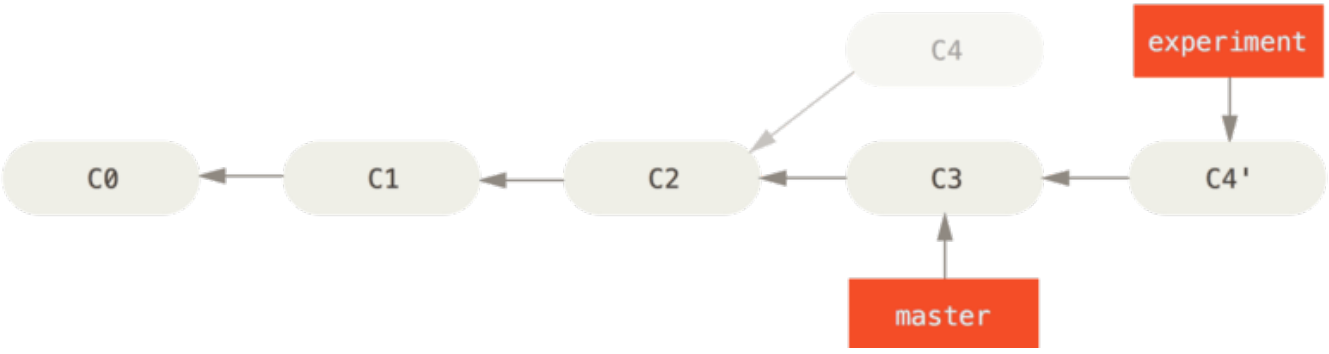


그림 37. C4 의 변경사항을 C3 에 적용하는 Rebase 과정

그리고 나서 main 브랜치를 Fast-forward 시킨다.

```
$ git checkout main
$ git merge experiment
```

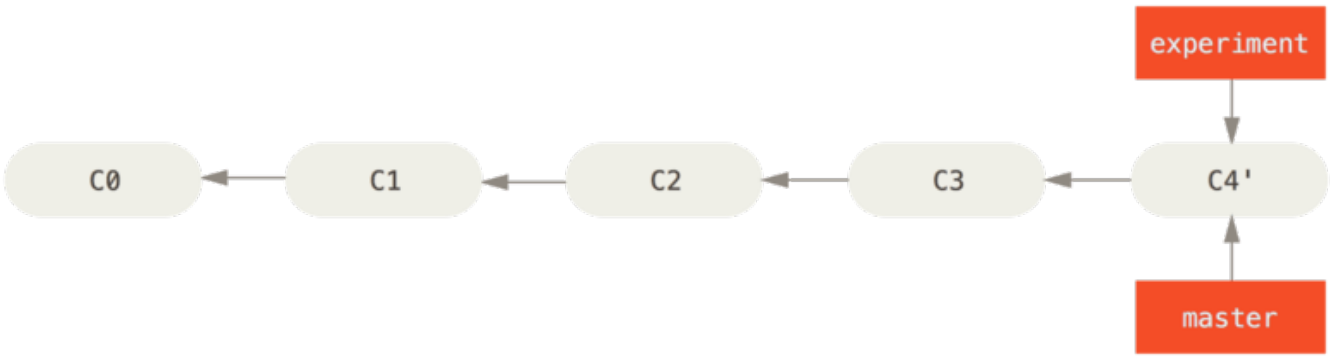


그림 38. main 브랜치를 Fast-forward시키기

C4' 로 표시된 커밋에서의 내용은 Merge 예제에서 살펴본 C5 커밋에서의 내용과 같을 것이다. Merge 이든 Rebase 든 둘 다 합치는 관점에서는 서로 다를 게 없다. 하지만, Rebase가 좀 더 깨끗한 히스토리를 만든다. Rebase 한 브랜치의 Log를 살펴보면 히스토리가 선형이다. 일을 병렬로 동시에 진행해도 Rebase 하고 나면 모든 작업이 차례대로 수행된 것처럼 보인다.

Rebase는 보통 리모트 브랜치에 커밋을 깔끔하게 적용하고 싶을 때 사용한다. 아마 이렇게 Rebase 하는 리모트 브랜치는 직접 관리하는 것이 아니라 그냥 참여하는 브랜치일 것이다. 메인 프로젝트에 Patch를 보낼 준비가 되면 하는 것이 Rebase 니까 브랜치에서 하던 일을 완전히 마치고 origin/main 로 Rebase 한다. 이렇게 Rebase 하고 나면 프로젝트 관리자는 어떠한 통합작업도 필요 없다. 그냥 main 브랜치를 Fast-forward 시키면 된다.

Rebase를 하든지, Merge를 하든지 최종 결과물은 같고 커밋 히스토리만 다르다는 것이 중요하다. Rebase 의 경우는 브랜치의 변경사항을 순서대로 다른 브랜치에 적용하면서 합치고 Merge 의 경우는 두 브랜치의 최종결과만을 가지고 합친다.

Rebase 활용

Rebase는 단순히 브랜치를 합치는 것만 아니라 다른 용도로도 사용할 수 있다. 다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치 같은 히스토리가 있다고 하자. server 브랜치를 만들어서 서버 기능을 추가하고 그 브랜치에서 다시 client 브랜치를 만들어 클라이언트 기능을 추가한다. 마지막으로 server 브랜치로 돌아가서 몇 가지 기능을 더 추가한다.

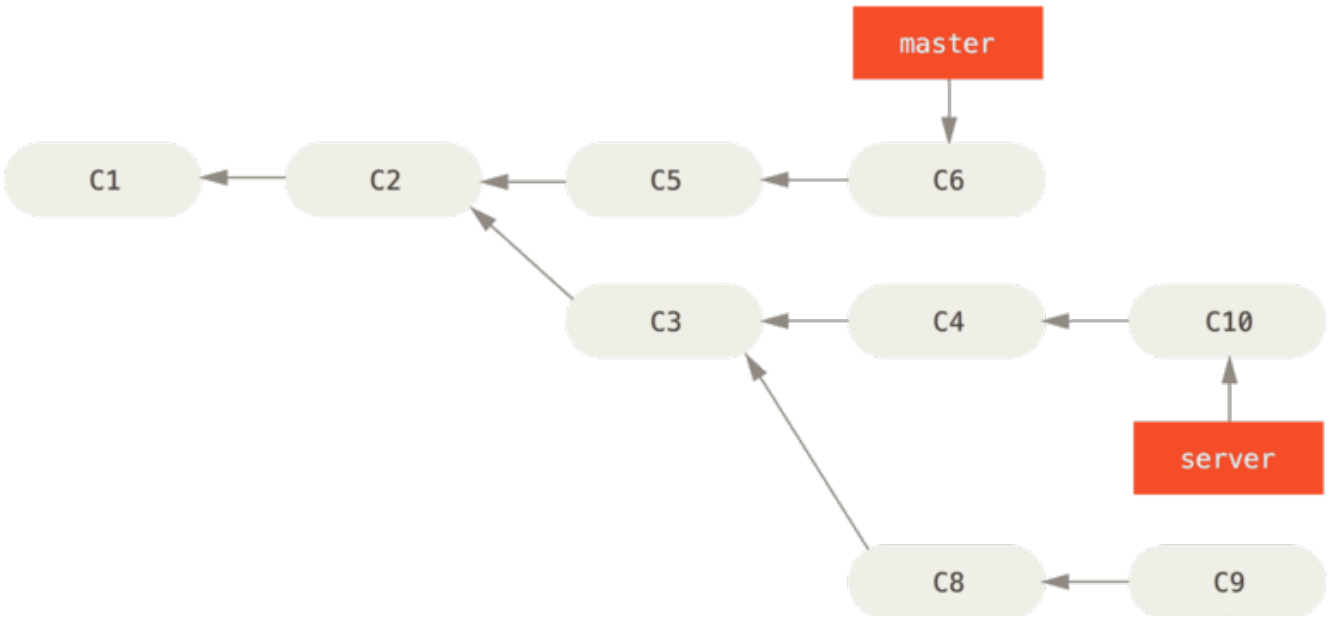


그림 39. 다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치

이때 테스트가 덜 된 `server` 브랜치는 그대로 두고 `client` 브랜치만 `main` 로 합치려는 상황을 생각해보자. `server` 와는 아무 관련이 없는 `client` 커밋은 `C8`, `C9` 이다. 이 두 커밋을 `main` 브랜치에 적용하기 위해서 `--onto` 옵션을 사용하여 아래와 같은 명령을 실행한다:

```
$ git checkout client
$ git rebase --onto main server client
```

이 명령은 `main` 브랜치부터 `server` 브랜치와 `client` 브랜치의 공통 조상까지의 커밋을 `client` 브랜치에서 없애고 싶을 때 사용한다. `client` 브랜치에서만 변경된 패치를 만들어 `main` 브랜치에서 `client` 브랜치를 기반으로 새로 만들어 적용한다. 조금 복잡하긴 해도 꽤 쓸모 있다.

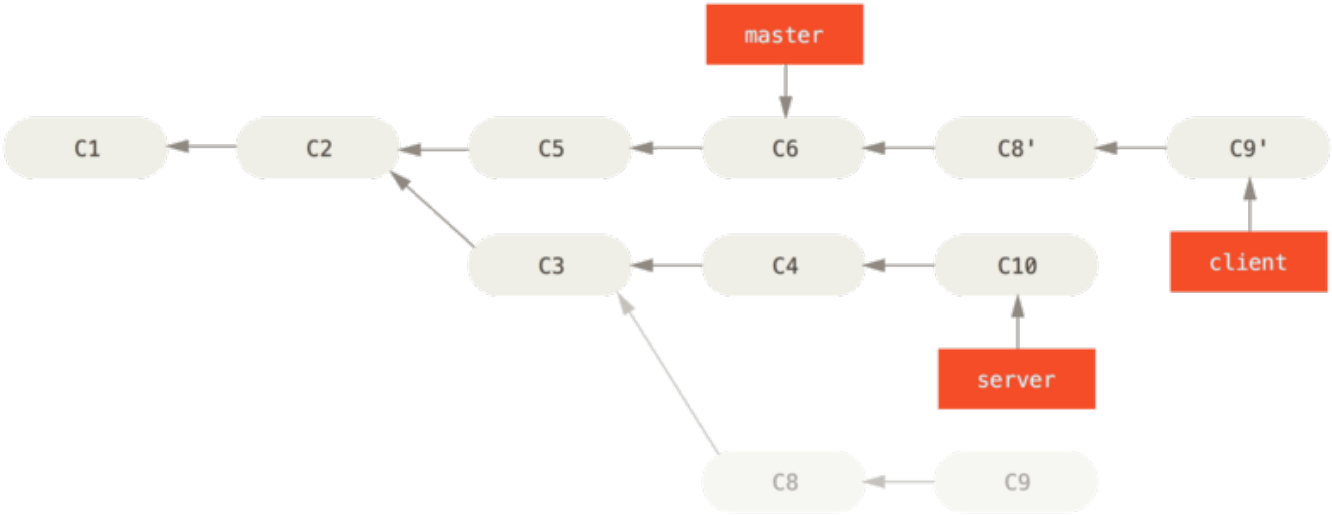


그림 40. 다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치를 Rebase 하기

이제 main 브랜치로 돌아가서 Fast-forward 시킬 수 있다(main 브랜치를 client 브랜치 위치로 진행 시키기 참고).

```
$ git checkout main
$ git merge client
```

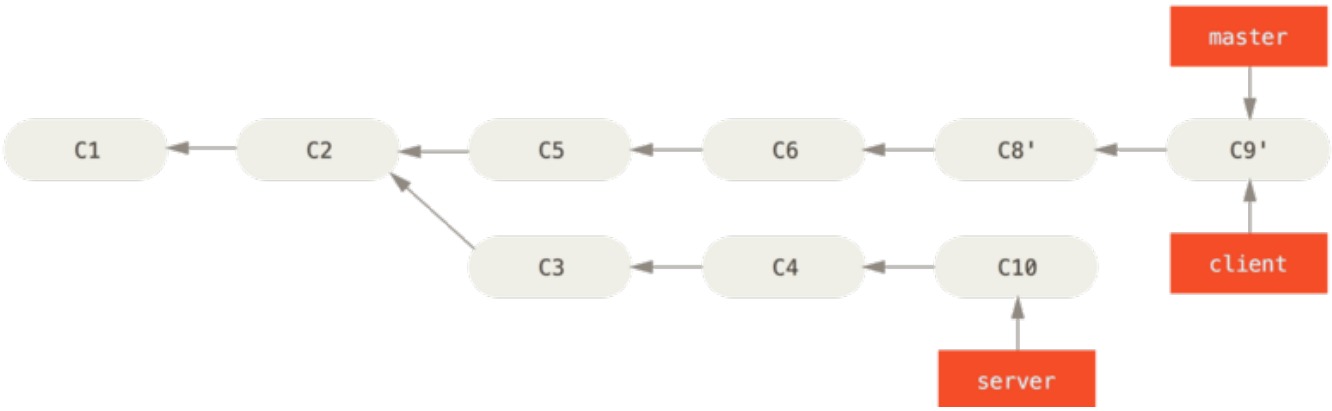


그림 41. main 브랜치를 client 브랜치 위치로 진행 시키기

`server` 브랜치의 일이 다 끝나면 `git rebase <basebranch> <topicbranch>` 라는 명령으로 Checkout 하지 않고 바로 `server` 브랜치를 `main` 브랜치로 Rebase 할 수 있다. 이 명령은 토픽(server) 브랜치를 Checkout 하고 베이스(main) 브랜치에 Rebase 한다.

```
$ git rebase main server
```

server 브랜치의 수정사항을 main 브랜치에 적용했다. 그 결과는 main 브랜치에 server 브랜치의 수정 사항을 적용 같다.

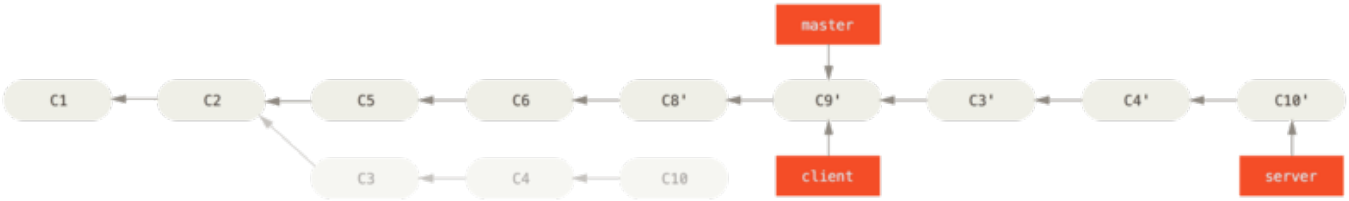


그림 42. main 브랜치에 server 브랜치의 수정 사항을 적용

그리고 나서 `main` 브랜치를 Fast-forward 시킨다.

```
$ git checkout main
$ git merge server
```

모든 것이 `main` 브랜치에 통합됐기 때문에 더 필요하지 않다면 `client` 나 `server` 브랜치는 삭제해도 된다. 브랜치를 삭제해도 커밋 히스토리는 최종 커밋 히스토리 같이 여전히 남아 있다.

```
$ git branch -d client
$ git branch -d server
```

그림 43. 최종 커밋 히스토리

Rebase 의 위험성

Rebase가 장점이 많은 기능이지만 단점이 없는 것은 아니니 조심해야 한다. 그 주의사항은 아래 한 문장으로 표현할 수 있다.

이미 공개 저장소에 Push 한 커밋을 Rebase 하지 마라

이 지침만 지키면 Rebase를 하는 데 문제 될 게 없다. 하지만, 이 주의사항을 지키지 않으면 사람들에게 욕을 먹을 것이다.

Rebase는 기존의 커밋을 그대로 사용하는 것이 아니라 내용은 같지만 다른 커밋을 새로 만든다. 새 커밋을 서버에 Push 하고 동료 중 누군가가 그 커밋을 Pull 해서 작업을 한다고 하자. 그런데 그 커밋을 `git rebase` 로 바꿔서 Push 해버리면 동료가 다시 Push 했을 때 동료는 다시 Merge 해야 한다. 그리고 동료가 다시 Merge 한 내용을 Pull 하면 내 코드는 정말 엉망이 된다.

이미 공개 저장소에 Push 한 커밋을 Rebase 하면 어떤 결과가 초래되는지 예제를 통해 알아보자. 중앙 저장소에서 Clone 하고 일부 수정을 하면 커밋 히스토리는 아래와 같아 진다.

```
git.team1.ourcompany.com
```

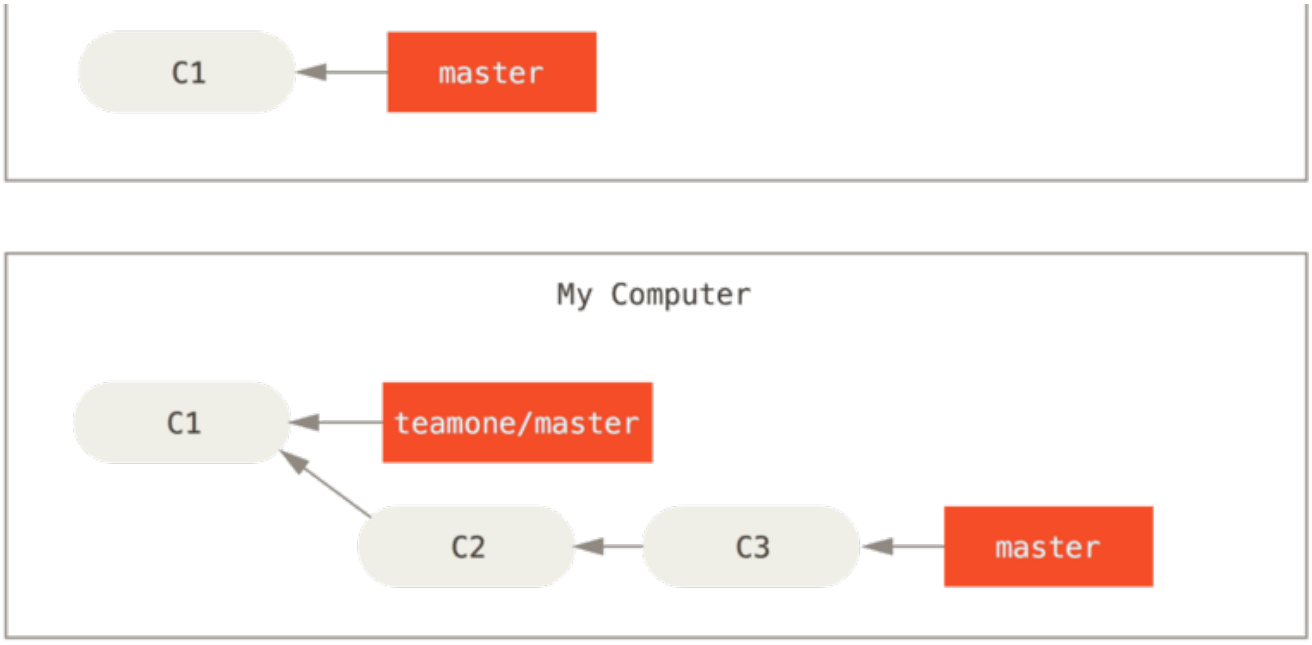


그림 44. 저장소를 Clone 하고 일부 수정함

이제 팀원 중 누군가 커밋, Merge 하고 나서 서버에 Push 한다. 이 리모트 브랜치를 Fetch, Merge 하면 히스토리는 아래와 같이 된다.

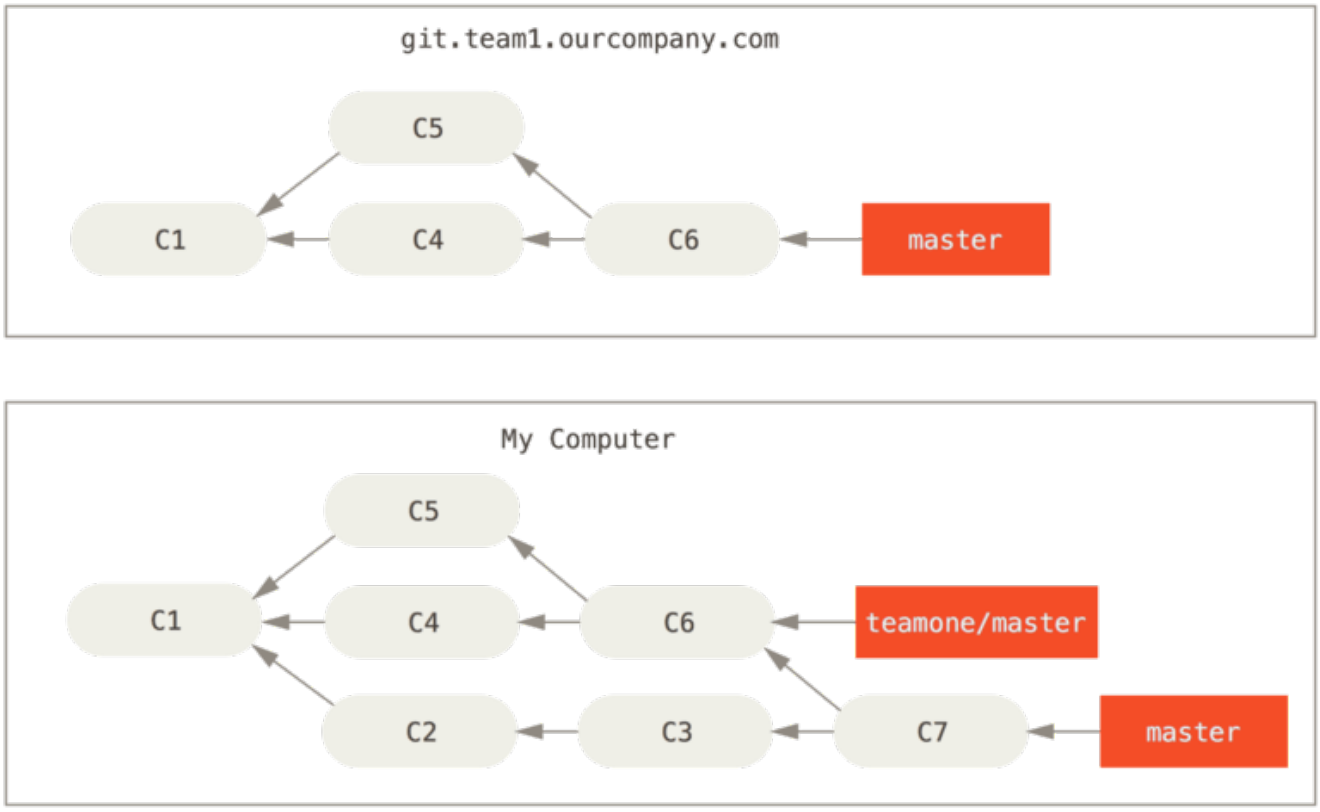


그림 45. Fetch 한 후 Merge 함

그런데 Push 했던 팀원은 Merge 한 일을 되돌리고 다시 Rebase 한다. 서버의 히스토리를 새로 덮어씌우려면 `git push --force` 명령을 사용해야 한다. 이후에 저장소에서 Fetch 하고 나면 아래 그림과 같은 상태가 된다.

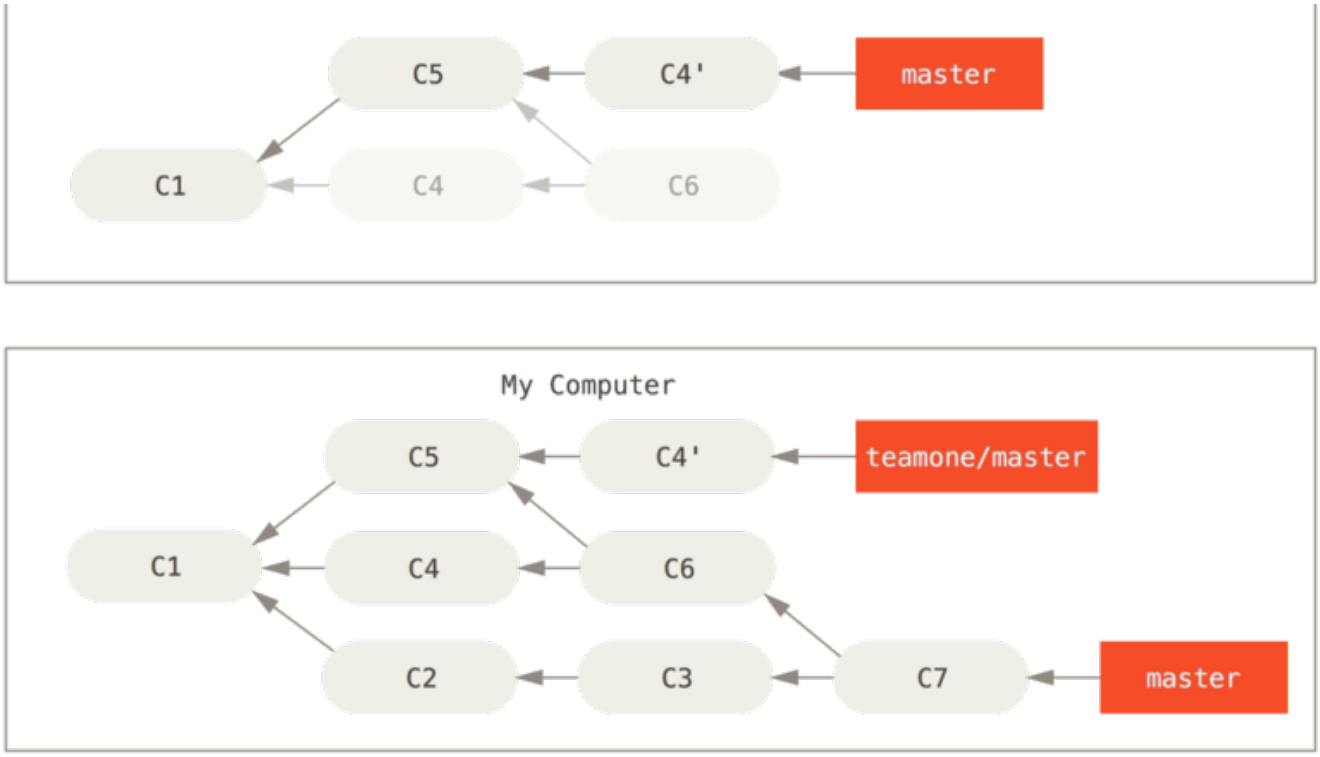


그림 46. 한 팀원이 다른 팀원이 의존하는 커밋을 없애고 Rebase 한 커밋을 다시 Push 함

이렇게 되면 커밋들이 중복이 된다. `git pull` 로 서버의 내용을 가져와서 Merge 하면 같은 내용의 수정사항을 포함한 Merge 커밋이 아래와 같이 만들어진다.

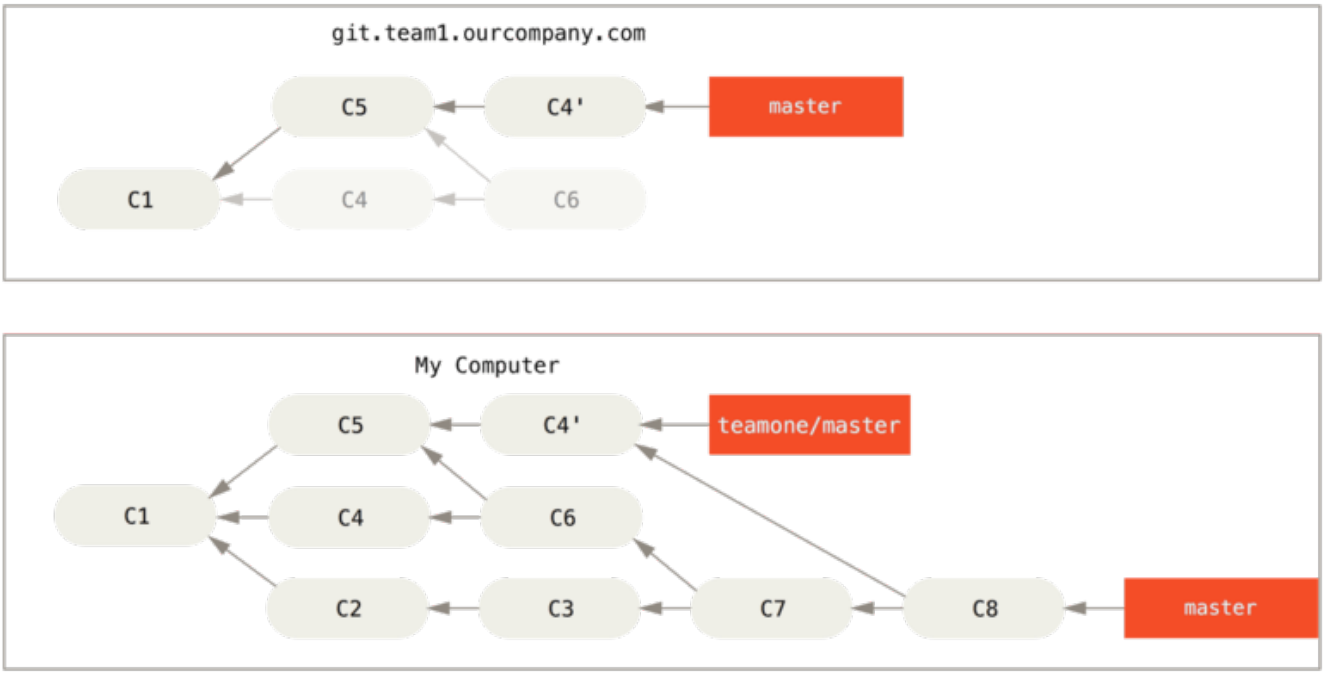


그림 47. 같은 Merge를 다시 한다

`git log` 로 히스토리를 확인해보면 저자, 커밋 날짜, 메시지가 같은 커밋이 두 개 있다(C4, C4'). 이렇게 되면 혼란스럽다. 게다가 이 히스토리를 서버에 Push 하면 같은 커밋이 두 개 있기 때문에 다른 사람들도 혼란스러워한다. `C4` 와 `C6` 는 포함되지 말았어야 할 커밋이다. 애초에 서버로 데이터를 보내기 전에 Rebase로 커밋을 정리했어야 했다.

Rebase 한 것을 다시 Rebase 하기

만약 이런 상황에 빠질 때 유용한 Git 기능이 하나 있다. 어떤 팀원이 강제로 내가 한일을 덮어썼다고 하자. 그러면 내가 했던 일이 무엇이고 덮어쓴 내용이 무엇인지 알아내야 한다.

커밋 SHA 체크섬 외에도 Git은 커밋에 Patch 할 내용으로 SHA-1 체크섬을 한번 더 구한다. 이 값은 "patch-id" 라고 한다.

덮어쓴 커밋을 받아서 그 커밋을 기준으로 Rebase 할 때 Git은 원래 누가 작성한 코드인지 잘 찾아 낸다. 그래서 Patch가 원래대로 잘 적용된다.

예를 들어 앞서 살펴본 예제를 보면 한 팀원이 다른 팀원이 의존하는 커밋을 없애고 Rebase 한 커밋을 다시 Push 함 상황에서 Merge 하는 대신 `git rebase teamone/main` 명령을 실행하면 Git은 아래와 같은 작업을 한다.

- 현재 브랜치에만 포함된 커밋을 결정한다. (C2, C3, C4, C6, C7)
- Merge 커밋이 아닌 것을 결정한다. (C2, C3, C4)
- 이 중 merge할 브랜치에 덮어쓰이지 않은 커밋을 결정한다. (C2, C3. C4는 C4'와 동일한 Patch다)
- 결정한 커밋을 `teamone/main` 브랜치에 적용한다.

결과를 확인해보면 같은 Merge를 다시 한다 같은 결과 대신 제대로 정리된 강제로 덮어쓴 브랜치에 Rebase 하기 같은 결과를 얻을 수 있다.

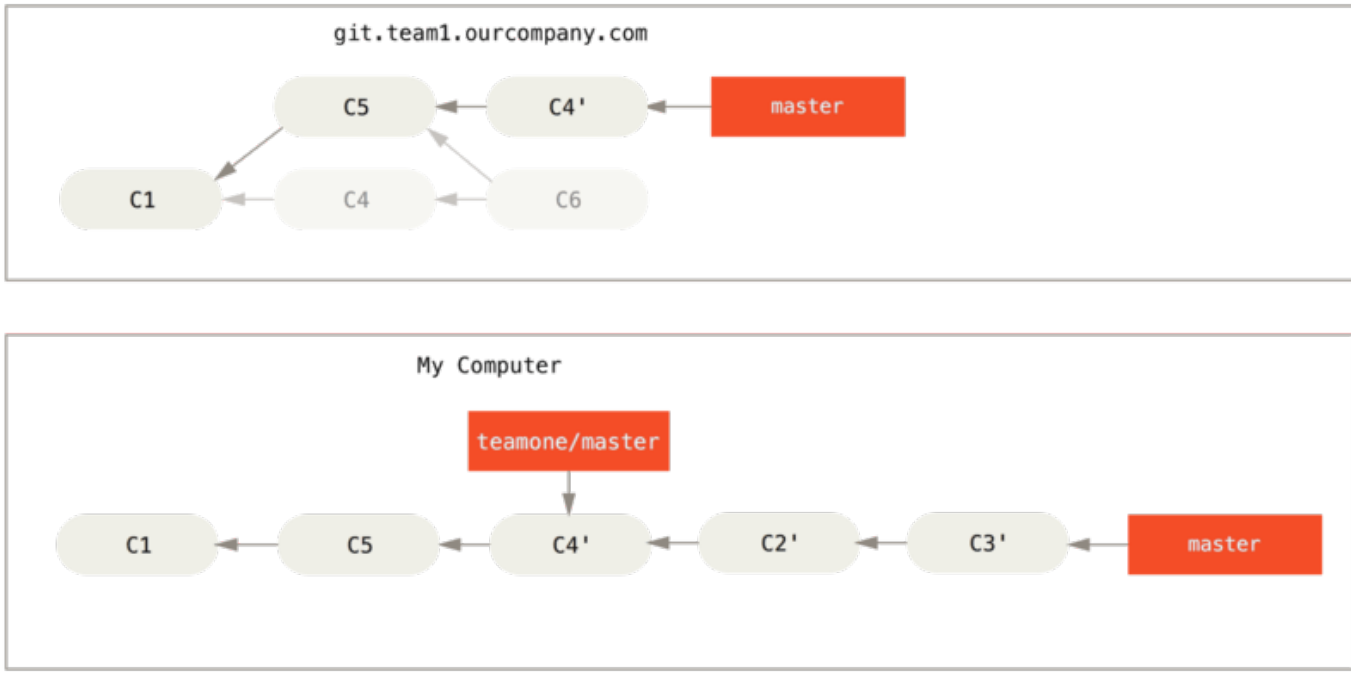


그림 48. 강제로 덮어쓴 브랜치에 Rebase 하기

동료가 생성했던 C4와 C4' 커밋 내용이 완전히 같을 때만 이렇게 동작된다. 커밋 내용이 아예 다르거나 비슷하다면 커밋이 두 개 생긴다(같은 내용이 두 번 커밋될 수 있기 때문에 깔끔하지 않다).

`git pull` 명령을 실행할 때 옵션을 붙여서 `git pull --rebase` 로 Rebase 할 수도 있다. 물론 `git fetch` 와 `git rebase teamone/main` 이 두 명령을 직접 순서대로 실행해도 된다.

`git pull` 명령을 실행할 때 기본적으로 `--rebase` 옵션이 적용되도록 `pull.rebase` 설정을 추가할 수 있다. `git config --global pull.rebase true` 명령으로 추가한다.

Push 하기 전에 정리하려고 Rebase 하는 것은 괜찮다. 또 절대 공개하지 않고 혼자 Rebase 하는 경우도 괜찮다. 하지만, 이미 공개하여 사람들이 사용하는 커밋을 Rebase 하면 틀림없이 문제가 생긴다.

나중에 고생하기 전에, `git pull --rebase` 로 문제를 미리 방지할 수 있다는 것을 같이 작업하는 동료와 모두 함께 공유하기 바란다.

Rebase vs. Merge

Merge가 뭔지, Rebase가 뭔지 여러 예제를 통해 간단히 살펴보았다. 지금쯤 이런 의문이 들 거로 생각한다. 둘 중 무엇을 쓰는 게 좋지? 이 질문에 대한 답을 찾기 전에 히스토리의 의미에 대해서 잠깐 다시 생각해보자.

히스토리를 보는 관점 중에 하나는 **작업한 내용의 기록**으로 보는 것이 있다. 작업 내용을 기록한 문서이고, 각 기록은 각각 의미를 가지며, 변경할 수 없다. 이런 관점에서 커밋 히스토리를 변경한다는 것은 역사를 부정하는 꼴이 된다. 언제 무슨 일이 있었는지 기록에 대해 *거짓말* 을 하게 되는 것이다. 이렇게 했을 때 지저분하게 수많은 Merge 커밋이 히스토리에 남게 되면 문제가 없을까? **역사**는 후세를 위해 기록하고 보존해야 한다.

히스토리를 **프로젝트가 어떻게 진행되었나에 대한 이야기**로도 볼 수 있다. 소프트웨어를 주의 깊게 편집하는 방법에 메뉴얼이나 세세한 작업내용을 초벌부터 공개하고 싶지 않을 수 있다. 나중에 다른 사람에게 들려주기 좋도록 Rebase 나 filter-branch 같은 도구로 프로젝트의 진행 이야기를 다듬으면 좋다.

Merge 나 Rebase 중 무엇이 나으냐는 질문은 다시 생각해봐도 답이 그리 간단치 않다. Git은 매우 강력한 도구고 기능이 많아서 히스토리를 잘 쌓을 수 있지만, 모든 팀과 모든 이가 처한 상황은 모두 다르다. 예제를 통해 Merge 나 Rebase가 무엇이고 어떤 의미인지 배웠다. 이 둘을 어떻게 쓸지는 각자의 상황과 각자의 판단에 달렸다.

다만 로컬 브랜치에서 작업할 때는 히스토리를 정리하기 위해서 Rebase 할 수도 있지만, 리모트 등 어딘가에 Push로 내보낸 커밋에 대해서는 절대 Rebase 하지 말아야 한다.