5. Git 브랜치로 작업하기 II

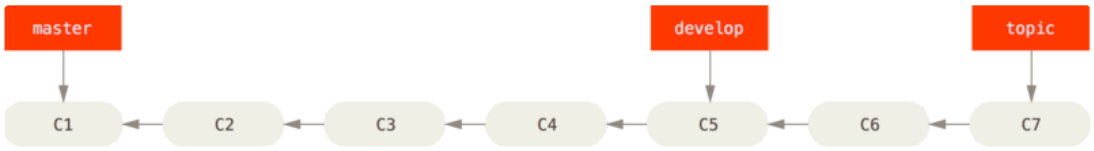
1. Git 브랜치 워크플로우

Long-Running 브랜치

Git 은 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.

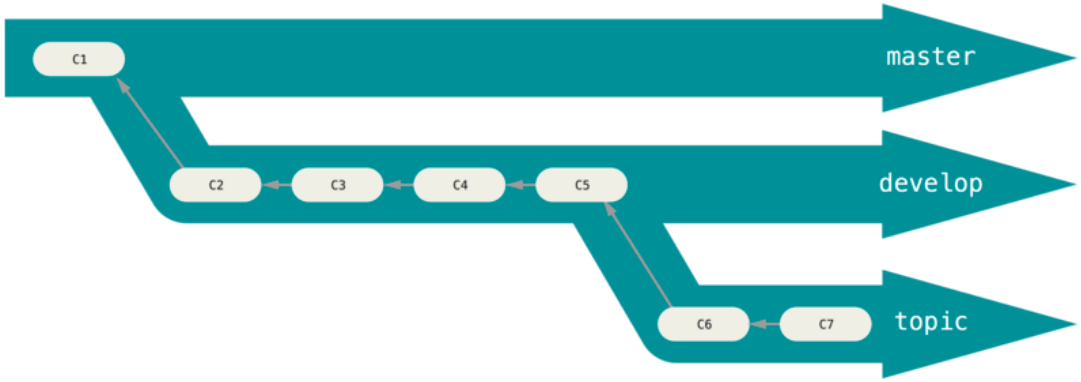
개발 브랜치는 공격적으로 히스토리를 만들어 나아가고 안정 브랜치는 이미 만든 히스토리를 뒤따르며 나아간다.

[안정적인 브랜치일수록 커밋 히스토리가 뒤쳐짐]



실험실에서 충분히 테스트하고 실전에 배치하는 과정으로 보면 이해하기 쉽다

[각 브랜치를 하나의 “실험실” 로 생각]

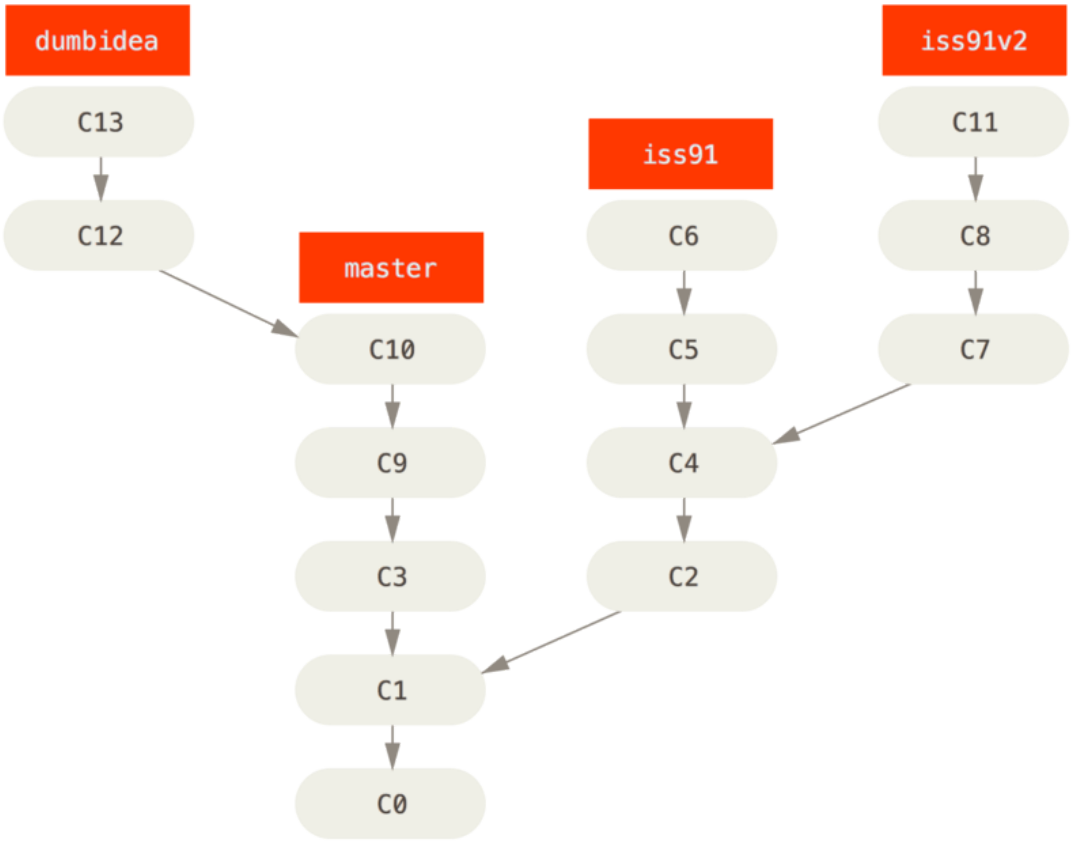


코드를 여러 단계로 나누어 안정성을 높여가며 운영할 수 있다. 특히 규모가 크고 복잡한 프로젝트일수록 그 유용성이 좋다.

토픽 브랜치

토픽 브랜치는 어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치다.

앞서 사용한 `iss53` 이나 `hotfix` 브랜치가 토픽 브랜치다. 각 작업을 하루든 한 달이든 유지하다가 `main` 브랜치에 Merge 할 시점이 되면 순서에 관계없이 그때 Merge 하면 된다.



---

2. Git 리모트 브랜치

리모트 브랜치란

리모트 Refs

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 브랜치

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.

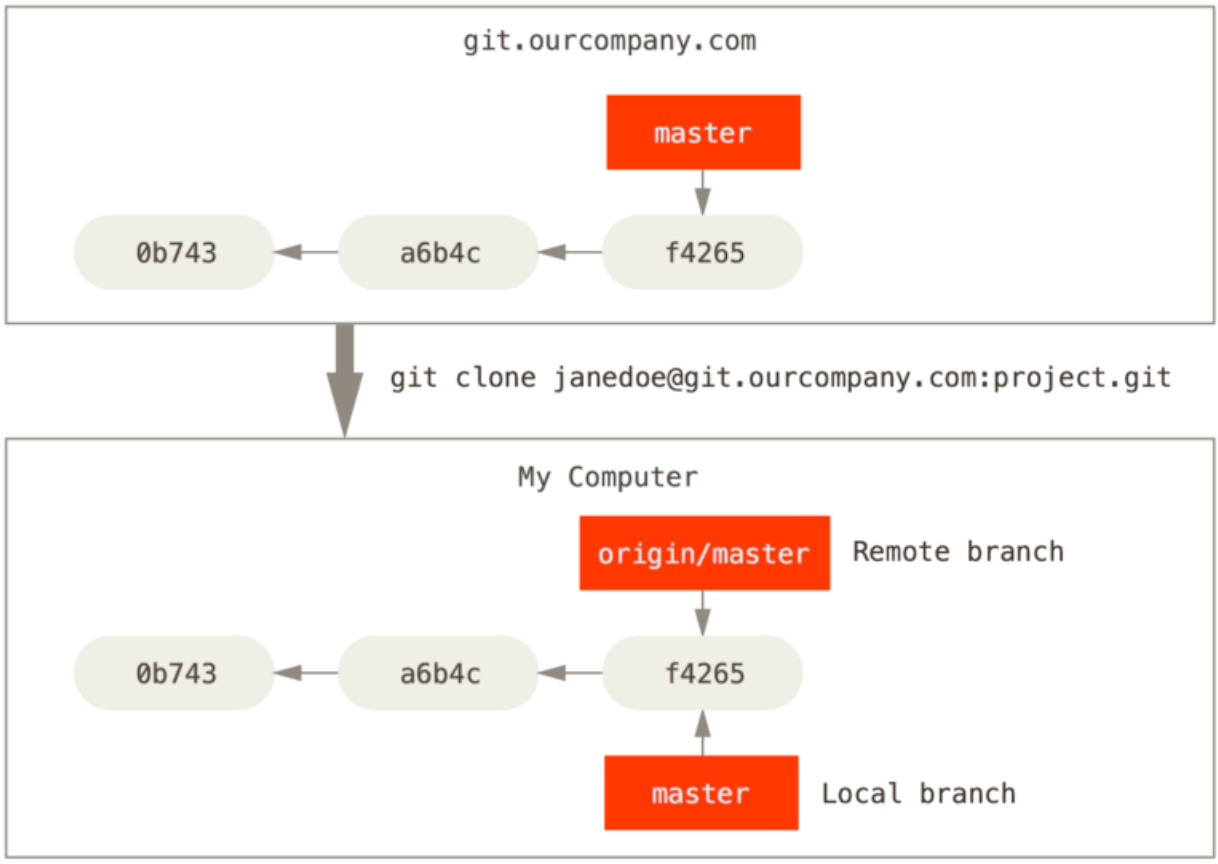
리모트 트래킹 브랜치의 이름은 `<remote>/<branch>` 형식으로 되어 있다. 예를 들어 리모트 저장소 `origin` 의 `main` 브랜치를 보고 싶다면 `origin/main` 라는 이름으로 브랜치를 확인하면 된다. 다른 팀원과 함께 어떤 이슈를 구현할 때 그 팀원이 `iss53` 브랜치를 서버로 Push 했고 당신도 로컬에 `iss53` 브랜치가 있다고 가정하자. 이때 서버의 `iss53` 브랜치가 가리키는 커밋은 로컬에서 `origin/iss53` 이 가리키는 커밋이다.

리모트 브랜치 예시

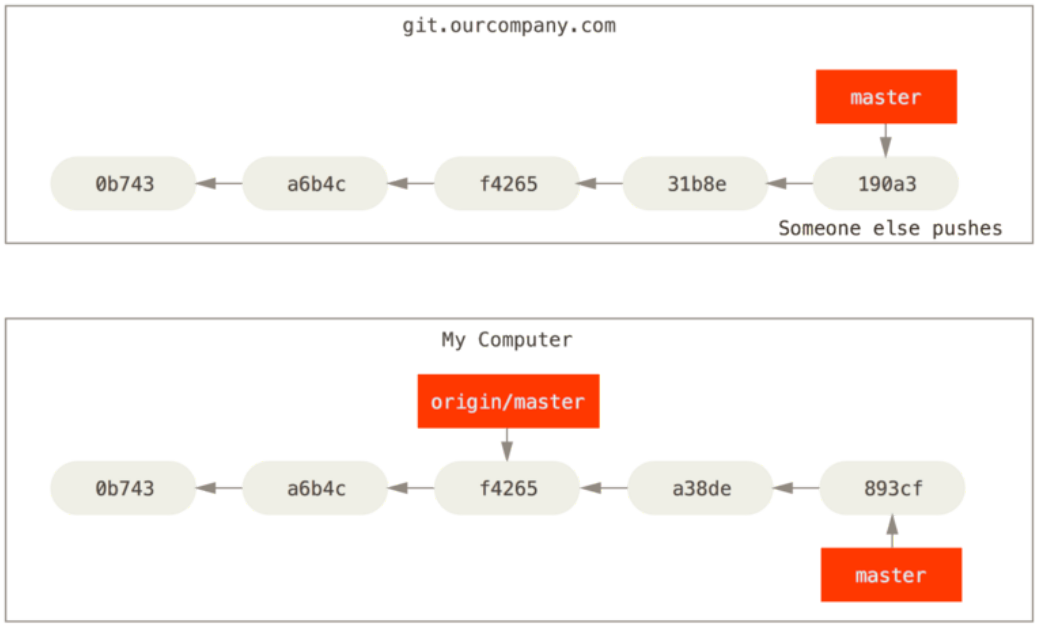
다음 예제를 살펴보자. `git.ourcompany.com` 이라는 Git 서버가 있고 이 서버의 저장소를 하나 Clone 하면 Git은 자동으로 `origin` 이라는 이름을 붙인다. `origin` 으로부터 저장소 데이터를 모두 내려받고 `main` 브랜치를 가리키는 포인터를 만든다. 이 포인터는 `origin/main` 라고

부르고 멋대로 조종할 수 없다. 그리고 Git은 로컬의 `main` 브랜치가 `origin/main` 를 가리키게 한다. 이제 이 `main` 브랜치에서 작업을 시작할 수 있다.

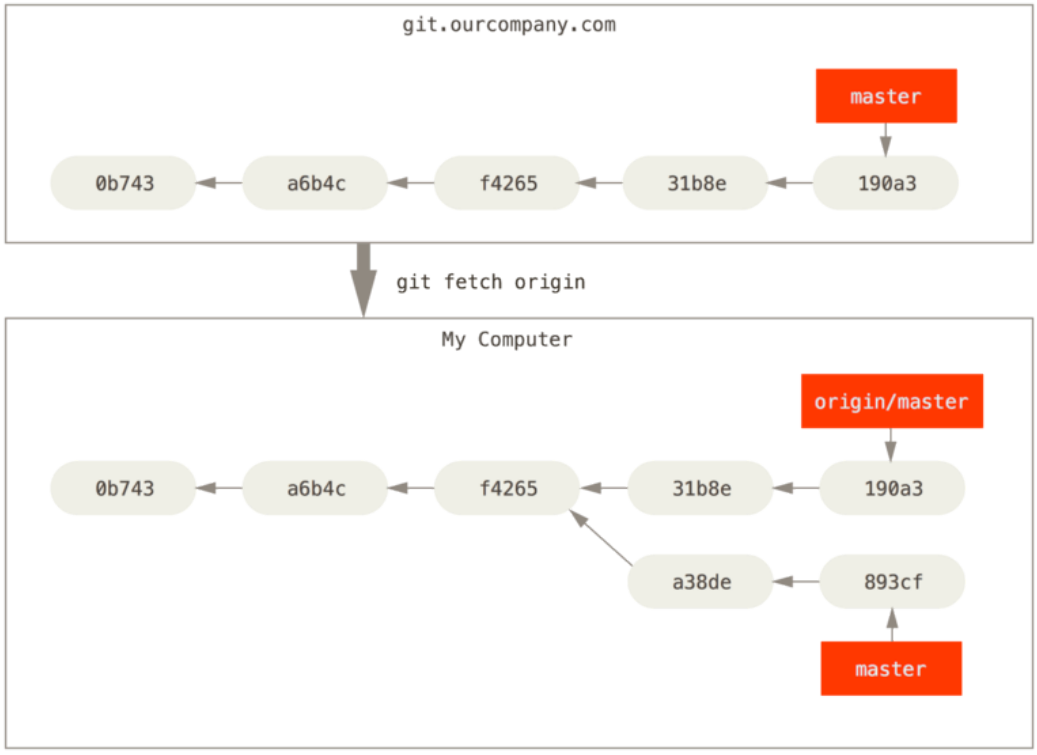
[Clone 이후 서버와 로컬의 main 브랜치]



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.



이때 다른 팀원이 추가 작업 한 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.



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 작업하는 브랜치는 비공개 브랜치가 된다. 또 다른 사람과 협업하기 위해 일부 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.

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.

`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 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.

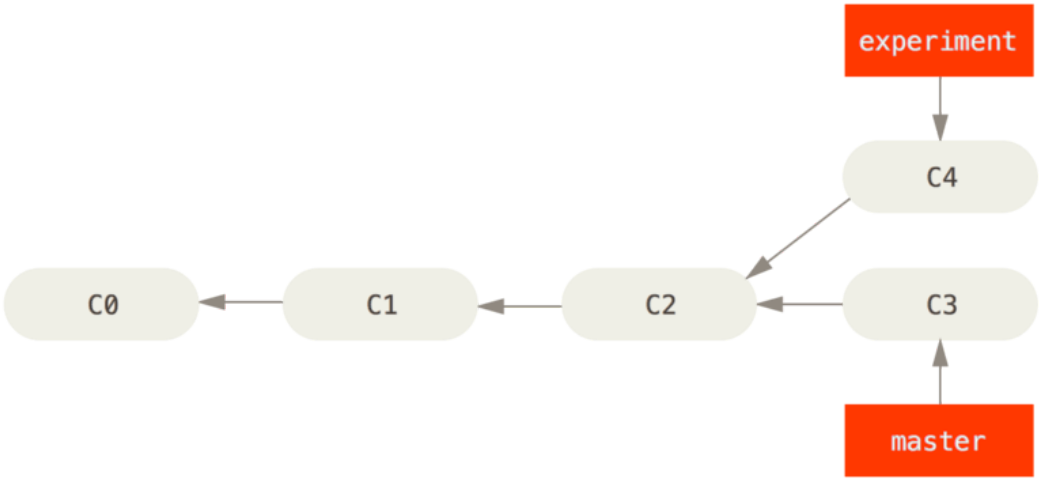
---

3. Git Rebase 활용하기

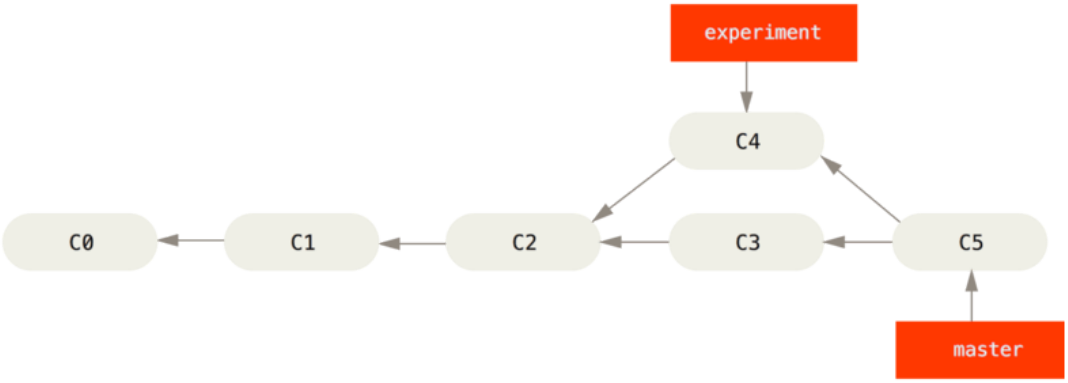
Rebase 기초

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.

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



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.

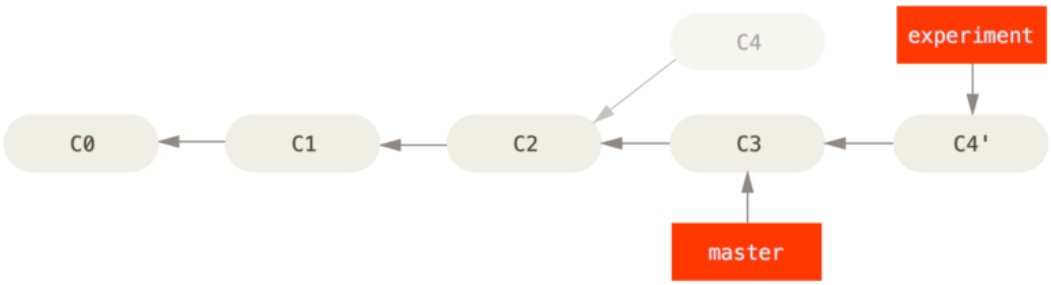


비슷한 결과를 만드는 다른 방식으로, `C3` 에서 변경된 사항을 Patch로 만들고 이를 다시 `C4` 에 적용시키는 방법이 있다. Git에서는 이런 방식을 *Rebase* 라고 한다. `rebase` 명령으로 한 브랜치에서 변경된 사항을 다른 브랜치에 적용할 수 있다.

위의 예제는 아래와 같은 명령으로 Rebase 한다.

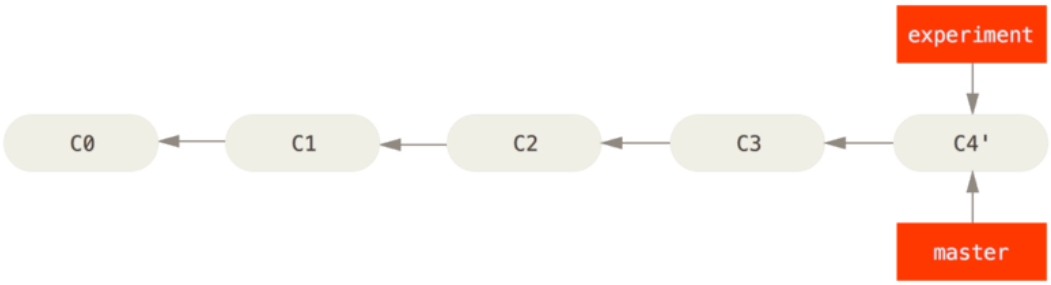
```
$ git checkout experiment
$ git rebase main
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

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.



그리고 나서 `main` 브랜치를 Fast-forward 시킨다.

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



Rebase 를 하든지, Merge 를 하든지 최종 결과물은 같고 커밋 히스토리만 다르다. Rebase 의 경우는 브랜치의 변경사항을 순서대로 다른 브랜치에 적용하면서 합치고 Merge 의 경우는 두 브랜치의 최종결과만을 가지고 합친다.

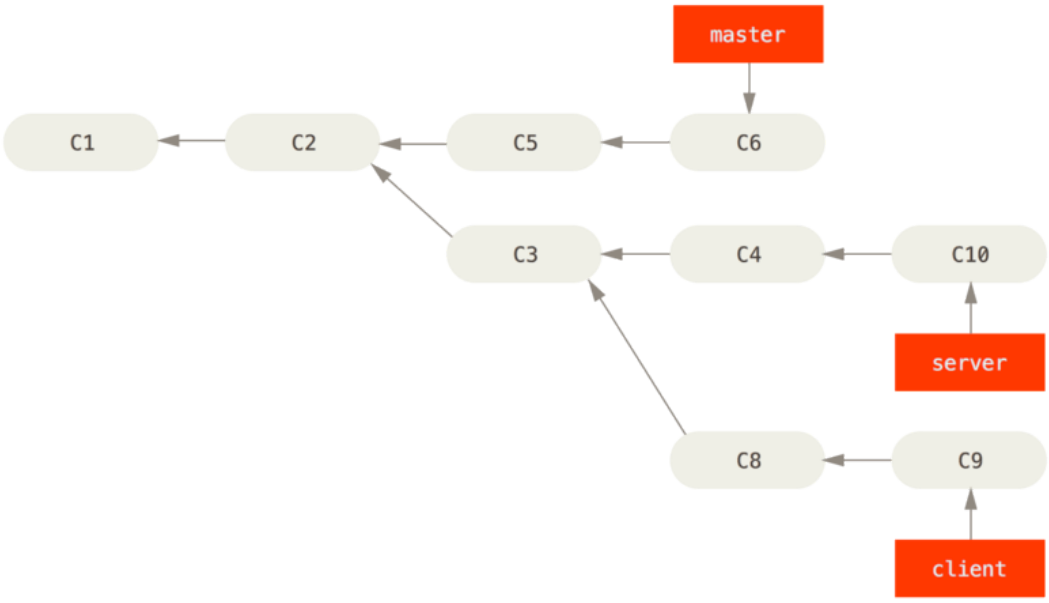
Rebase 활용

다음과 같이 실습 진행 세팅을 한다.

```
$ git checkout main
$ echo 'C1' > C1 && git add C1 && git commit -m 'C1'
$ echo 'C2' > C2 && git add C2 && git commit -m 'C2'
$ git checkout -b server
$ echo 'C3' > C3 && git add C3 && git commit -m 'C3'
$ git checkout -b client
$ echo 'C8' > C8 && git add C8 && git commit -m 'C8'
$ echo 'C9' > C9 && git add C9 && git commit -m 'C9'
$ git checkout server
$ echo 'C4' > C4 && git add C4 && git commit -m 'C4'
$ echo 'C10' > C10 && git add C10 && git commit -m 'C10'
$ git checkout main
$ echo 'C5' > C5 && git add C5 && git commit -m 'C5'
$ echo 'C6' > C6 && git add C6 && git commit -m 'C6'

```

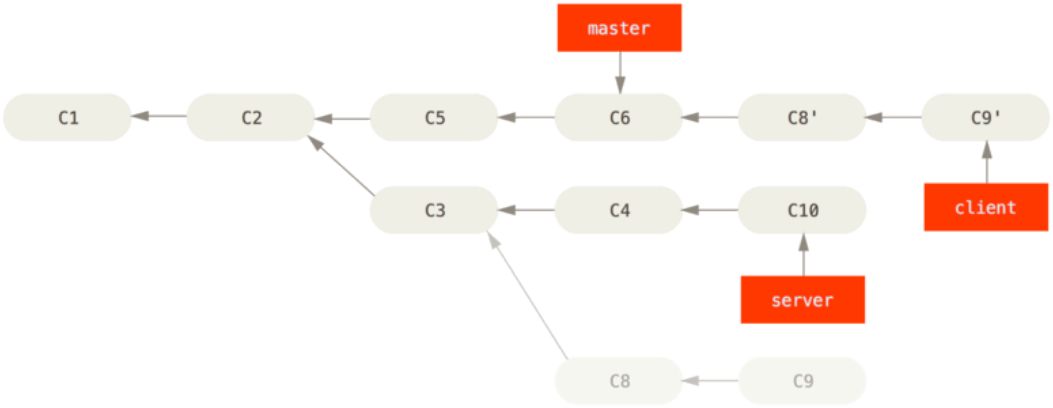
Rebase는 단순히 브랜치를 합치는 것만 아니라 다른 용도로도 사용할 수 있다. 사전 실습 스크립트를 진행했으면 다음과 같은 히스토리가 생긴다. `server` 브랜치를 만들어서 서버 기능을 추가하고 그 브랜치에서 다시 `client` 브랜치를 만들어 클라이언트 기능을 추가했다. 마지막으로 `server` 브랜치로 돌아가서 몇 가지 기능을 더 추가한 모습니다.



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:

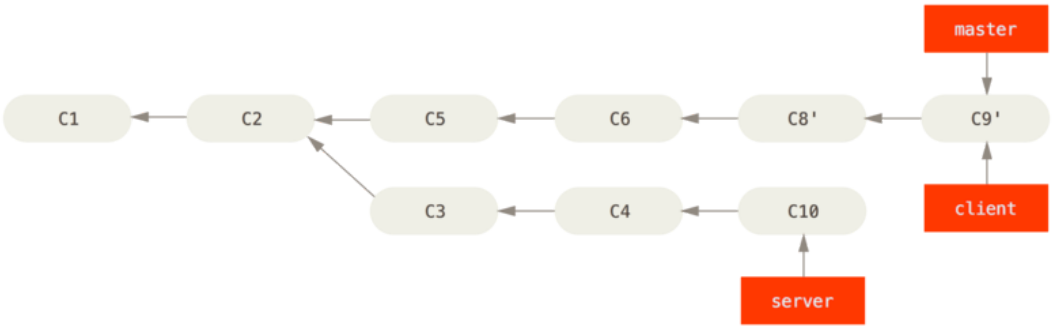
```
$ git checkout client
$ git rebase --onto main server client
```

이 명령은 `main` 브랜치부터 `server` 브랜치와 `client` 브랜치의 공통 조상까지의 커밋을 `client` 브랜치에서 없애고 싶을 때 사용한다.



이제 main 브랜치로 돌아가서 Fast-forward 시킬 수 있다.

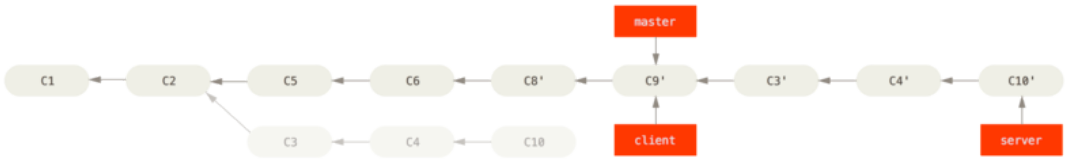
```
$ git checkout main
$ git merge client
```



`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.



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
```

Rebase 의 위험성

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

이미 공개 저장소에 Push 한 커밋을 Rebase 하면 안됨.

이 지침만 지키면 Rebase를 하는 데 문제 될 게 없다. 하지만, 이 주의사항을 지키지 않으면 동료를 곤란하게 만들 수 있다.

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 코드가 갈수록 엉망이 된다.