**Git 브랜치**

개발을 하다 보면 기존 코드에서 코드를 통째로 복사해서 원래 코드와는 상관 없이 개별적으로 코드 개발을 하는 경우가 생기는데 이렇게 개별적으로 개발하는 것이 브랜치다.

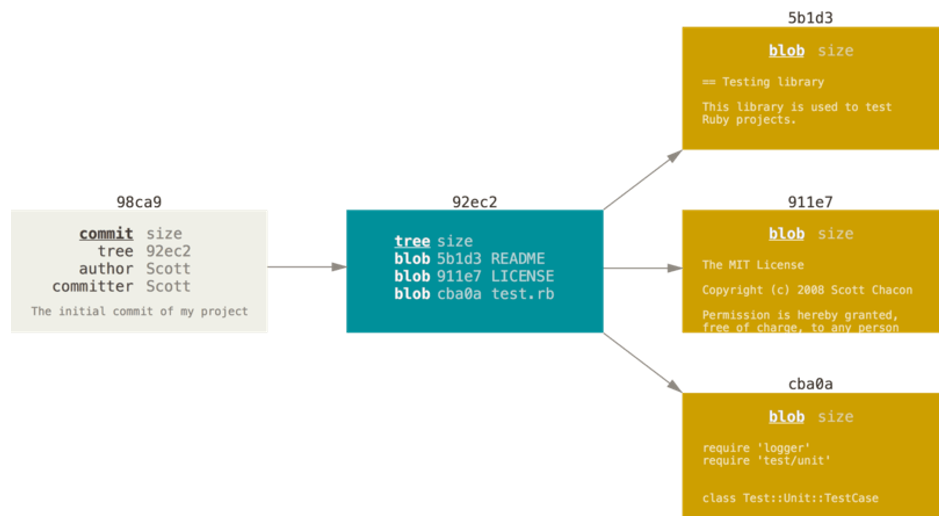
**Git 브랜치의 특징**

* 가볍다 (순식간에 만들고 브랜치 사이를 쉽게 이동 가능하다)
* 브랜치를 만들고 다시 merge하는 식으로 활용. 개발방식의 혁신.

**브랜치란 무엇인가(Git이 데이터를 저장하는 방식)**

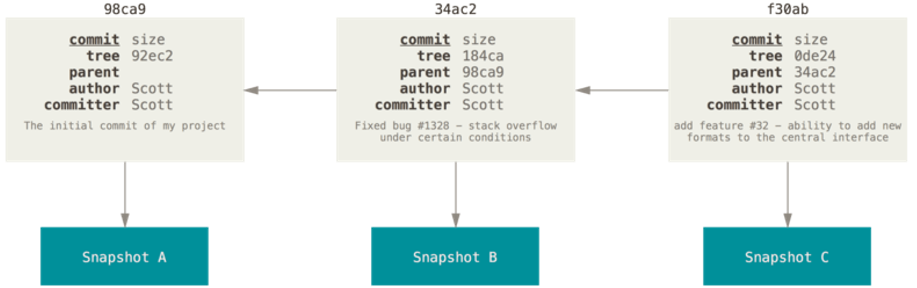
* 데이터를 스냅샷으로 저장
* 커밋을 하면 커밋 객체(Object)를 만든다.
* 커밋 객체는 현 Staging Area(현 저장 상태?)에 있는 데이터 스냅샷에 대한 포인터, 저자나 커밋 메시지 같은 메타 데이터, 이전 커밋에 대한 포인터(현재 커밋이 무엇을 기준으로 바뀌었는지 알 수 있음)등을 가지고 있다.
* 모든 커밋 객체가 이전 커밋 포인터가 적어도 하나씩 있고 Merge 커밋 같은 경우 이전 포인터가 여러 개 있다.

**예) 파일이 3개가 있는 디렉토리를 Staging Area에 저장하고 커밋하는 예제**

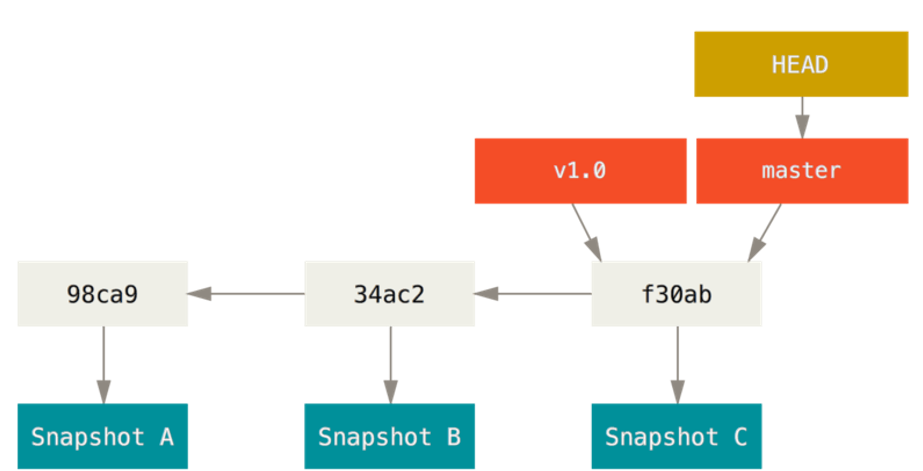
****

* 파일을 commit 하면 **각 파일에 대한 blob**(파일을 Stage 했을때 Git 저장소에 저장되는 것)와 **트리 개체 하나**, **커밋 개체 하나**가 생성된다. 트리 개체(초록색)에는 디렉토리에 대한 정보가 들어가고 커밋 개체(회색)에는 메타데이터와 트리 개체를 가리키는 포인터 정보가 들어간다.

**예) 파일을 수정하고 다시 커밋하는 예제**

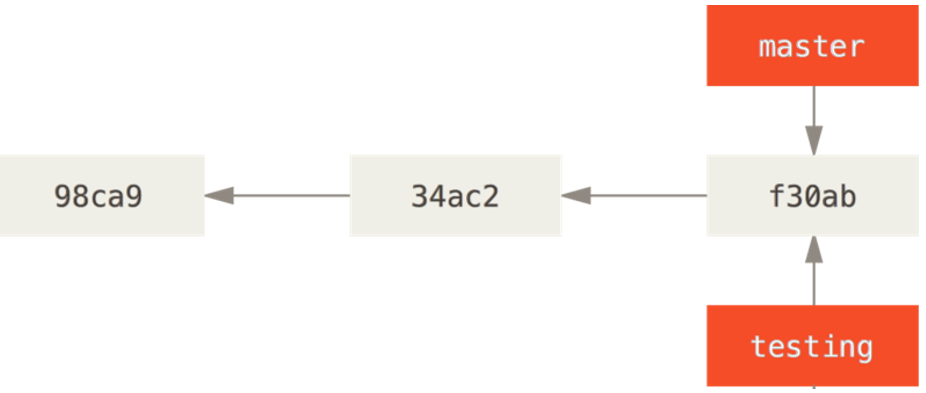
****

* 파일을 수정하고 다시 커밋하면 이전 커밋이 무엇인지도 저장한다.
* 브랜치는 커밋 사이를 가볍게 이동할 수 있는 포인터와 같은 개념
* 기본적으로 Git은 master 브랜치를 만든다. 처음 커밋할 때 이 master 브랜치가 생성된 커밋을 가리킨다. 이후 커밋을 보면 브랜치는 자동으로 마지막 커밋을 가리킨다. Master라는 이름의 브랜치가 특별한건 아니지만 git init으로 초기화할 때 이름(다른이름으로 변경 x)이다.

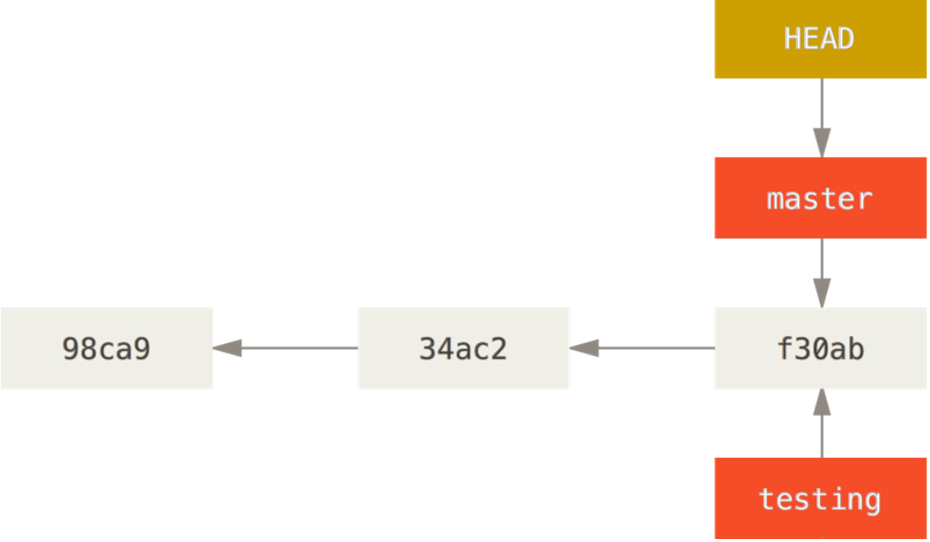
****

**새 브랜치 생성**

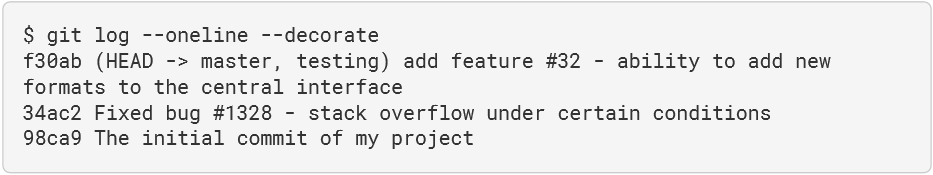
* $ git branch testing -> 지금 작업하고 있던 마지막 커밋을 가리킨다.

****

* 현재 작업하고 있는 브랜치는 ‘Head’라는 특수한 포인터로 알 수 있다.

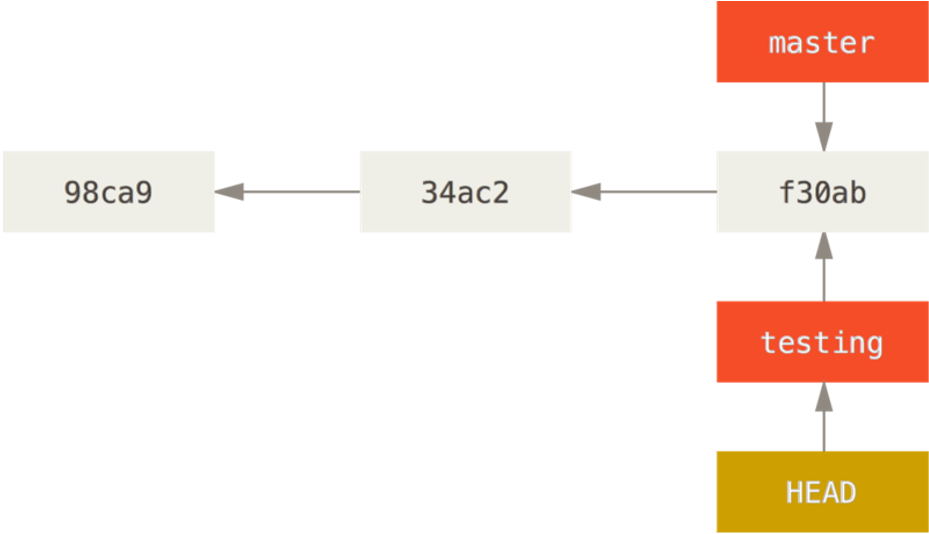
****

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

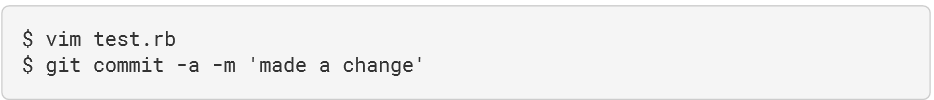
****

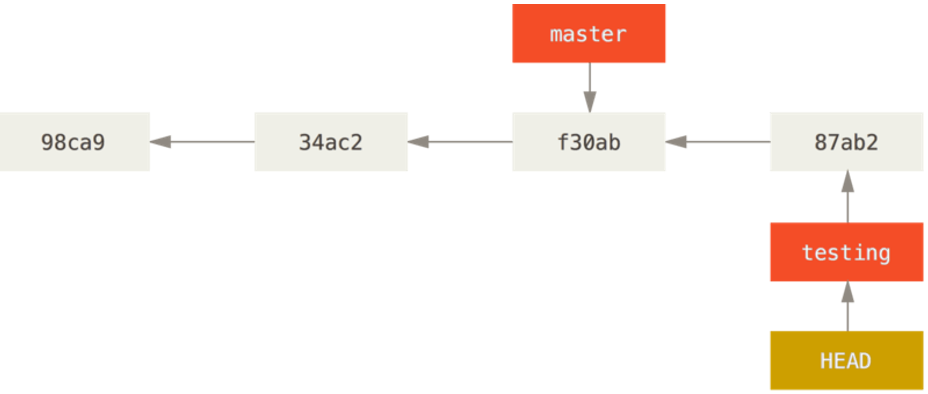
**브랜치 이동하기**

* $ git checkout testing -> Head가 testing을 가리킨다.

****

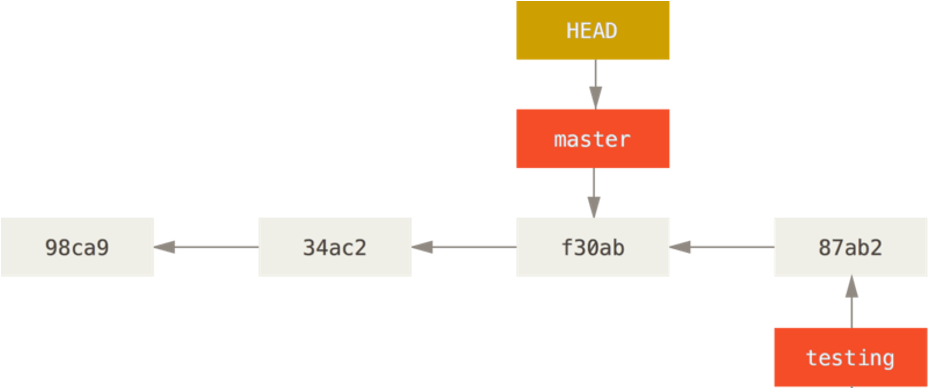
* 이 상태에서 새로 커밋을 하게 된다면?

****

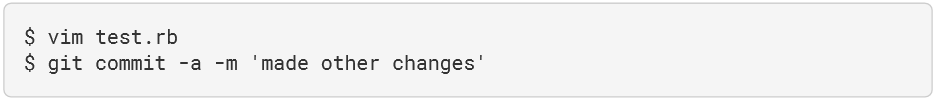
****

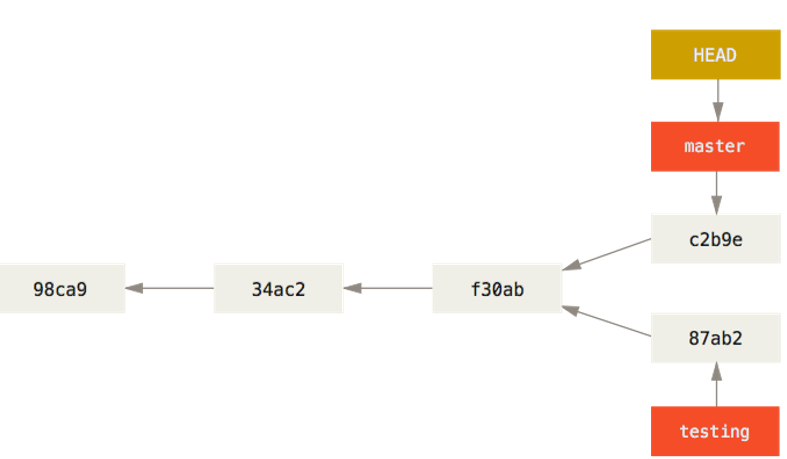
* Head가 testing을 가리키고 있기 때문에 testing은 새로운 커밋을 가리키지만 master은 기존 커밋을 가리킨다.
* 이 상태에서 master 브랜치로 이동한다면?

****

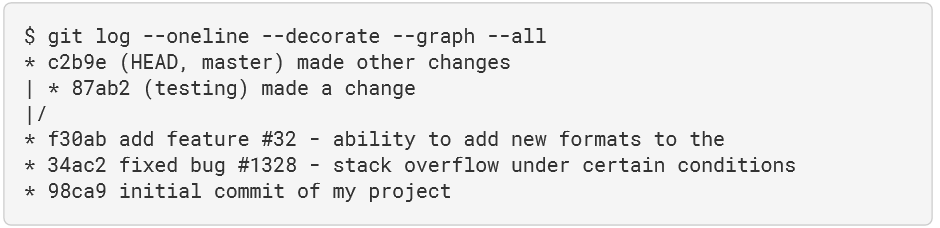
****

* Head가 master을 가리킴과 동시의 파일의 버전을 이전 시점으로 되돌려 놓았다. 이런 방식으로 인해 앞으로 모든 ‘테스트’는 testing에서 임시로 작업할 수 있게 되었다. 또 master로 돌아와서 언제든지 기존에 하던 일을 계속할 수 있게 되었다.
* 이 상태에서 파일을 수정하고 다시 커밋하게 된다면?



****

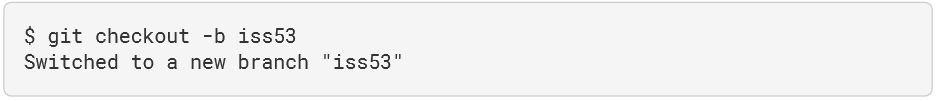
* 브랜치가 갈라진다.
* 커밋 사이를 자유롭게 이동하다가 때가 되면 두 브랜치를 Merge한다. 이때 쓰이는 명령어는 branch, checkout, commit이다.
* git log --oneline --decorate --graph --all 명령으로 히스토리를 확인할 수 있다.

****

* 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. Cf) 다른 버전 관리 도구는 프로젝트를 통째로 복사한다.

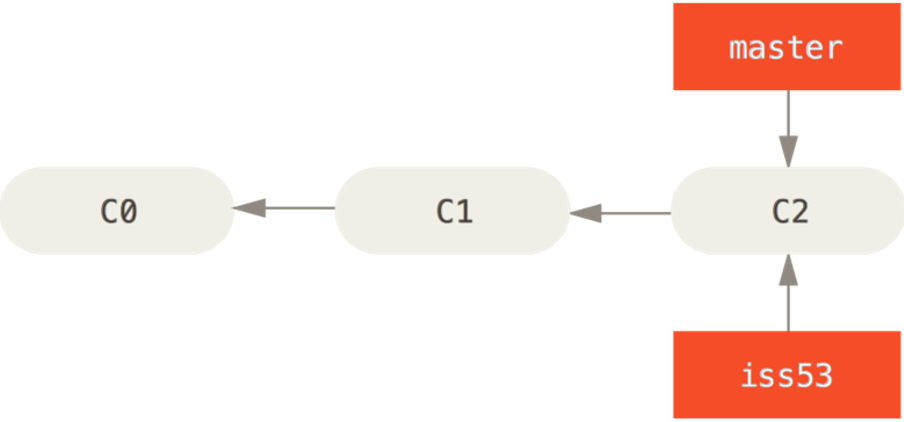
**브랜치와 Merge의 기초**

* 웹사이트를 만들 때 새로운 이슈가 생길 때 마다 새 브랜치를 만들고 이전의 운영 브랜치로 merge. 그 사이에 다른 문제가 생기면 원래 운영 브랜치에서 새로운 브랜치 만들어서 문제 해결하고 merge.
* 브랜치를 만들면서 Checkout까지 한 번에 하려면 git checkout 명령에 -b 라는 옵션을 추가하면 된다.

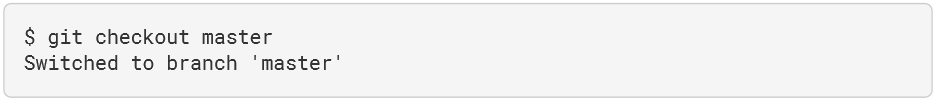




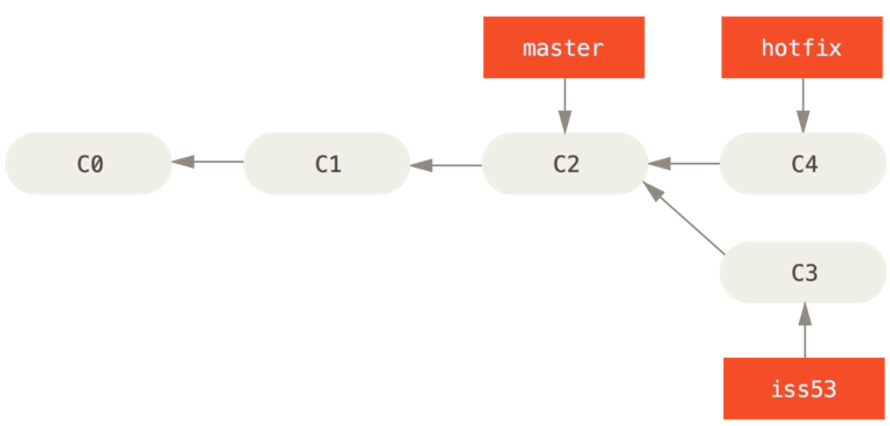
* 위의 두 명령어는 같은 명령어이다.



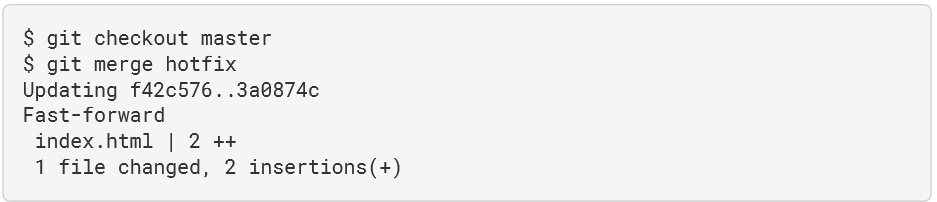
* 이 상태에서 커밋을 하게 되면 master은 그대로 C2을 가리키지만 iss53 브랜치는 앞으로 나아가게 된다.
* Hotfix라는 문제를 해결하기 위해 새로운 브랜치를 만들 때



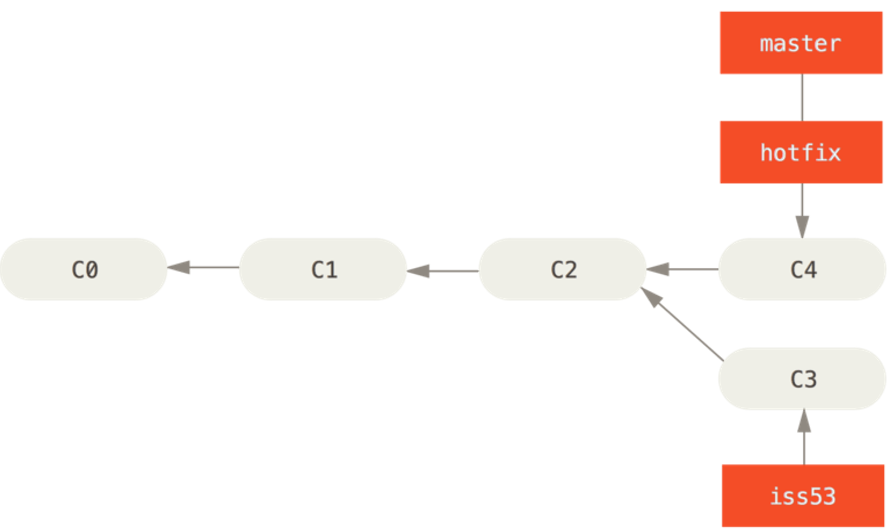




* 이런 그림이 된다.
* 문제가 해결되는지 확인하고 master 브랜치에가서 hotfix 브랜치를 merge한다.



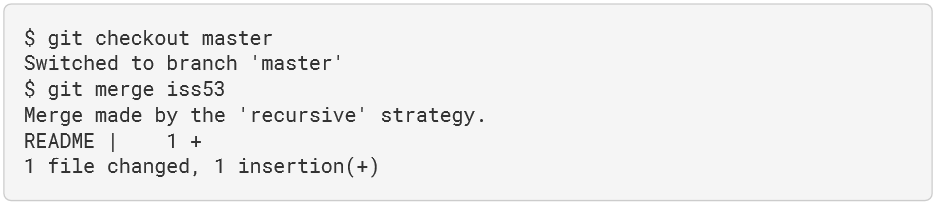
* 해야 하는데 이 경우에 Fast-forward라는 말이 보인다. 이 말은 hotfix 브랜치가 가리키는 C4 커밋이 C2 커밋에 기반한 브랜치이기 때문에 브랜치 포인터는 Merge 과정 없이 그저 최신 커밋으로 이동한다는 말이다.



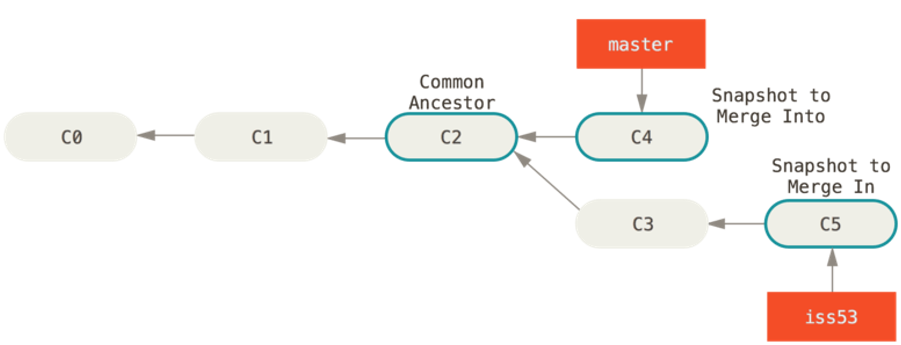
* 더 이상 hotfix 브랜치는 필요가 없기 때문에 git branch 명령에 -d 옵션을 주고 삭제한다.



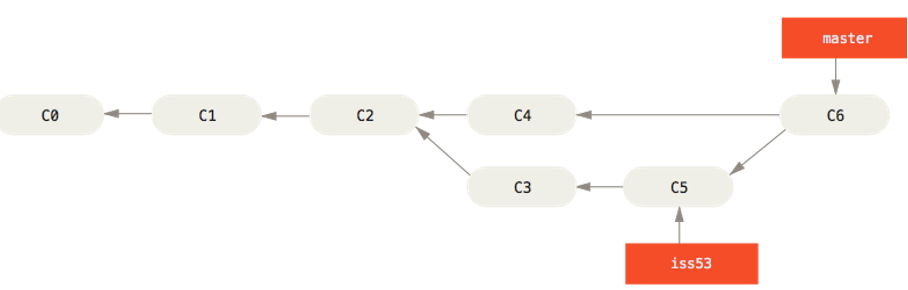
* 이 때 iss53과 별개로 작업이 진행된다.
* Iss53을 master에 merge한다.



* Hotfix 브랜치를 merge할 때와 다르게 ‘Fast-forward’로 Merge하지 않는 이유는 현재 브랜치가 가리키는 커밋이 Merge 할 브랜치의 조상이 아니기 때문이다. 이 경우에 Git은 각 브랜치가 가리키는 커밋 두 개와 공통 조상 하나를 사용하여 3-way Merge를 한다.



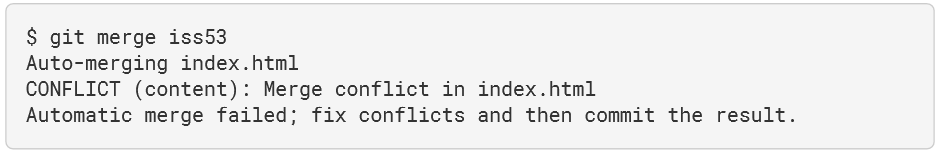
* 단순히 브랜치 포인터를 최신 커밋으로 옮기는 게 아니라 3-way Merge의 결과를 별도의 커밋으로 만들고 나서 해당 브랜치가 그 커밋을 가리키도록 이동시킨다. 그래서 이런 커밋은 부모가 여러 개고 Merge 커밋이라고 부른다.

****

* 이제 iss53이 필요가 없기 때문에 삭제해준다.

****

* 충돌이 일어나는 경우는 merge하는 두 brance에서 같은 파일을 수정했을 때 충돌이 발생한다.

****

* 어떤 파일이 충돌을 일으키는지 확인하려면 git status

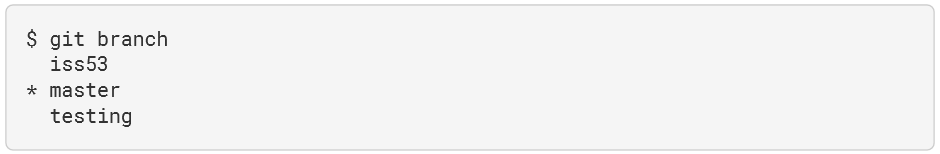
****

****

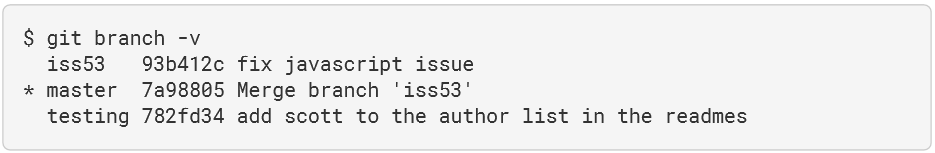
* 충돌을 해결하려면 두 내용 중에서 하나를 고르거나 직접 수정해야 한다.
* 다른 Merge 도구도 충돌을 해결할 수 있다. git mergetool 명령으로 실행한다.

**브랜치 관리**

* 아직까지 브랜치를 만들고, Merge 하고, 삭제하는 방법에 대해서 봤다면 브랜치를 관리하는 방법도 알아야 한다.
* git branch 명령은 단순히 브랜치를 만들고 삭제하는 것이 아니다. 아무런 옵션 없이 실행하면 브랜치의 목록을 보여준다

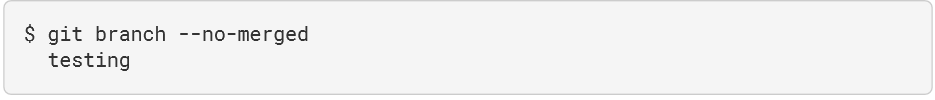
****

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

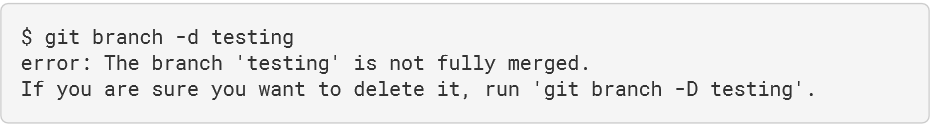
****

* 각 브랜치가 지금 어떤 상태인지 확인하려면 현재 Checkout 한 브랜치를 기준으로 --merged 와 --no-merged 옵션을 사용하여 Merge 된 브랜치인지 그렇지 않은지 필터링해 볼 수 있다.

****

****

* 위에는 \*가 없는 브랜치는 삭제할 수 있지만 밑에는 아직 Merge 하지 않은 커밋을 담고 있기 때문에 git branch -d 명령으로 삭제되지 않는다. 그래도 강제로 삭제하려면 -D 옵션을 사용한다.

**’**

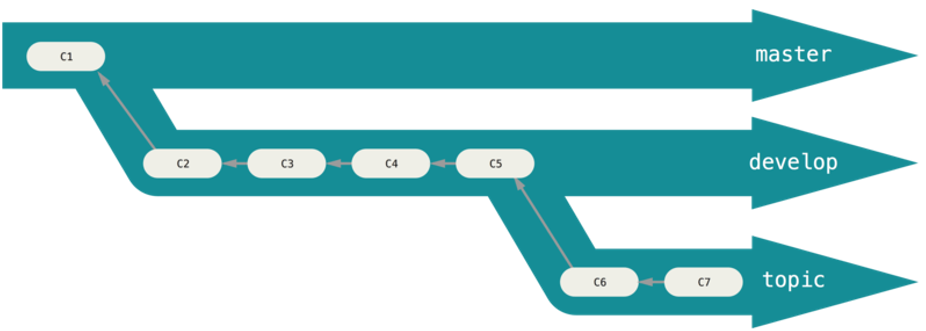
**브랜치 워크플로**

* Git 브랜치가 유용한 몇 가지 워크플로들

**Long-Running 브랜치**

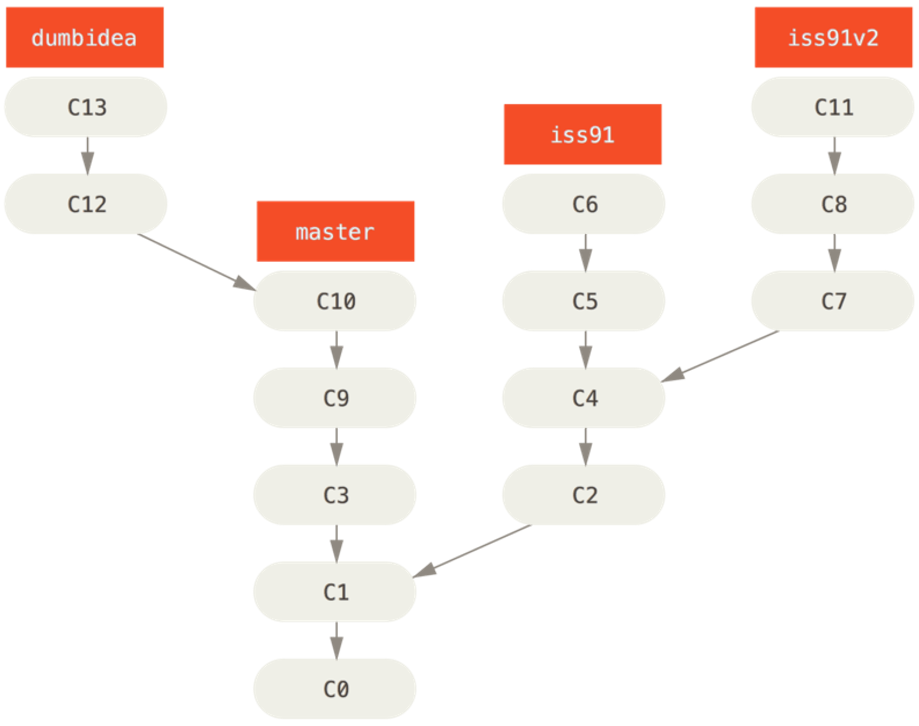
* 배포했거나 배포할 코드만 master에 둔다. 개발을 진행하고 안정화 단계에 있는 코드는 따로 브랜치를 만든다. 수많은 테스트를 거쳐 안전하다고 판단됬을 때 master에 merge 한다.
* 개발 브랜치는 공격적으로 히스토리를 만들어 나아가고 안정 브랜치는 이미 만든 히스토리를 뒤따르며 나아간다.
* 규모가 크고 복잡한 프로젝트일수록 그 유용성이 빛난다

****

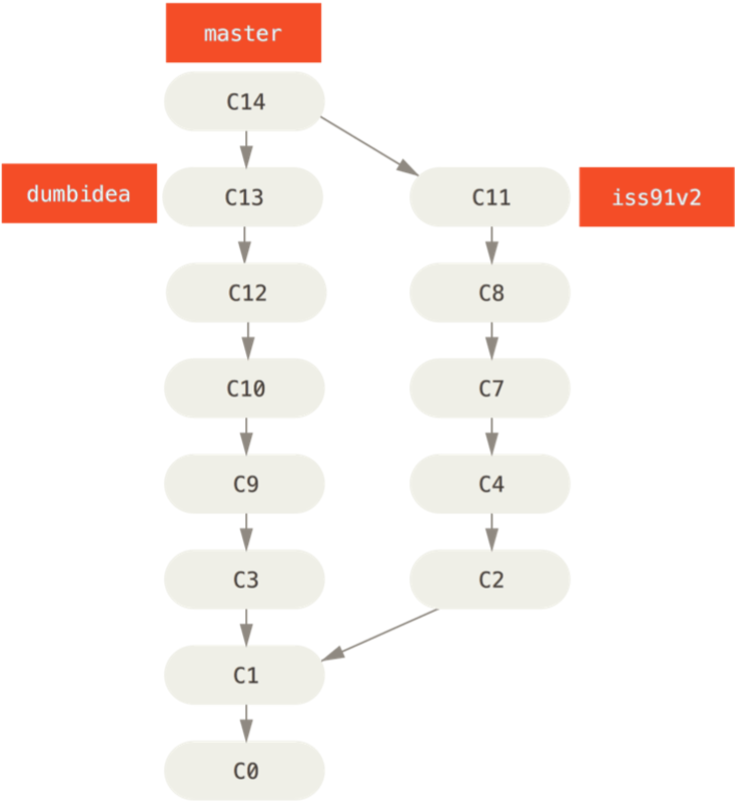
****

**Topic 브랜치**

* 프로젝트의 크기에 상관없이 유용하다. 토픽 브랜치는 어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치다. Git은 브랜치를 만들고 삭제하는게 간편하게 때문에 가능하다. 위의 iss53이나 hotfix가 예시.
* 묶음, 주제 별로 관리하기 때문에 내용을 검토하기에도, 테스트 하기에도 편하다. 각 작업을 하다가 master에 merge하게 되면 순서에 상관없이 merge하면 된다.
* master 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 master 브랜치로 되돌아가서 dumbidea 브랜치를 하나 더 만든다. 지금까지 말했던 커밋 히스토리는 아래 그림 같다.

****

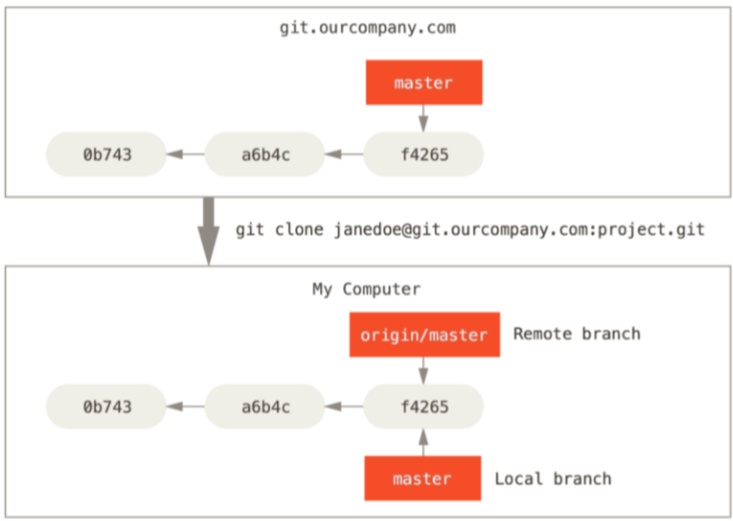
* iss91v2와 dumbidea를 master에 merge하게 되면?

****

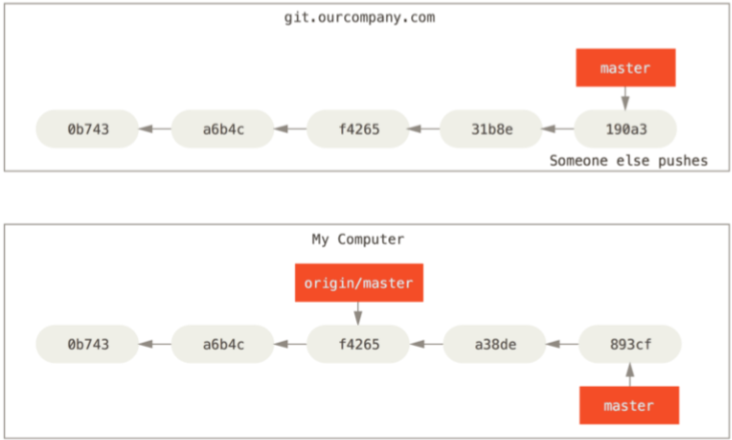
* 이렇게 된다.

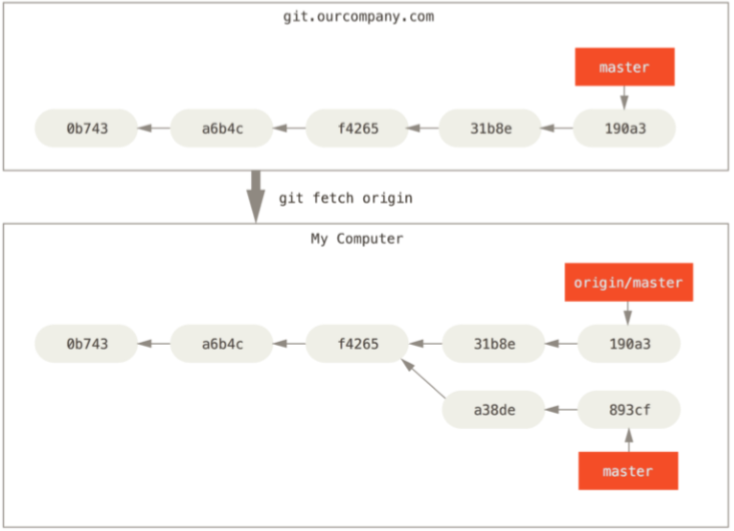
**리모트 브랜치**

* 리모트 Refs는 리모트 저장소에 있는 포인터인 레퍼런스다. 리모트 저장소에 있는 브랜치, 태그, 등등을 의미한다. **git ls-remote [remote] 명령**으로 모든 리모트 Refs를 조회할 수 있다. **git remote show [remote] 명령**은 모든 리모트 브랜치와 그 정보를 보여준다. 리모트 Refs가 있지만 보통은 리모트 트래킹 브랜치를 사용한다. 리모트 트래킹 브랜치는 리모트 저장소에 마지막으로 연결했던 순간에 브랜치가 무슨 커밋을 가리키고 있었는지를 나타낸다.
* 리모트 브랜치의 이름은 (remote)/(branch) 형식으로 되어 있다. 예를 들어 리모트 저장소 origin 의 master 브랜치를 보고 싶다면 origin/master 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**.
* 예를 들어 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 master 브랜치를 가리키는 포인터를 만든다. 이 포인터는 origin/master 라고 부르고 멋대로 조종할 수 없다. 그리고 Git은 로컬의 master 브랜치가 origin/master 를 가리키게 한다. 이제 이 master 브랜치에서 작업을 시작할 수 있다.
* master가 특별한 의미가 없이 초기 설정된 이름인 것 처럼 origin도 clone할 때 자동으로 주어지는 이름이고 사용자가 리모트 이름을 설정하고 싶으면 git clone -o 를 쓰면 된다. 예를 들어 git clone -o booyah 라고 하면 booyah/master가 된다.

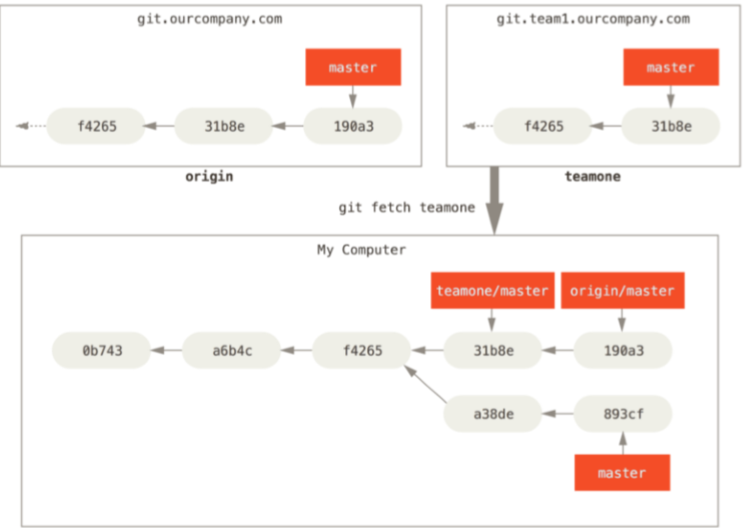
****

* 이 상태에서 내가 아닌 어떤 다른 사람이 push를 해서 master 브랜치를 업데이트 한다면? 팀원간의 히스토리는 달라지고 나의 origin/master의 포인터는 그대로다. 이때 리모트 서버로부터 저장소 정보를 동기화 하려면 **git fetch origin** 명령을 사용하면 된다. **그러면 git은 origin 서버의 주소로 가서 현재 로컬 저장소가 갖고 있지 않은 새로운 정보가 있으면 모두 내려받고 origin/master의 포인터의 위치를 최신 커밋으로 이동시킨다.**

****

****

* 리모트 저장소를 여러 개 운영할 때
* git remote add 명령어로 팀의 저장소를 추가한다. 이름은 teamone으로 짓고 주소는 git.team1.ourcoumpany.com이다.

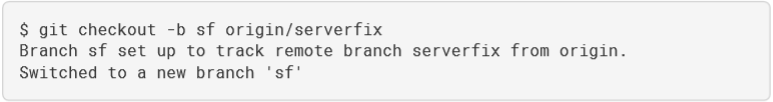
****

* git fetch teamone을 입력해도 이미 서버의 데이터를 로컬 저장소에서 다 가지고 있기 때문에 데이터는 내려받지 않지만 리모트 트래킹 브랜치 teamone/master 가 teamone 서버의 master 브랜치가 가리키는 커밋을 가리키게 한다.
* 로컬의 브랜치를 서버로 전송하려면 쓰기 권한이 있는 리모트 저장소에 Push 해야 한다. Push 하지 않으면 로컬 브랜치는 비공개 브랜치처럼 활용할 수 있다.
* 예) Serverfix를 다른 사람과 공유할 때 git push <remote> <branch>

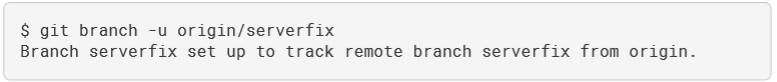
****

**브랜치 추적**

* 리모트 트래킹 브랜치를 로컬 브랜치로 Checkout 하면 자동으로 “트래킹(Tracking) 브랜치''가 만들어진다. 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 master 브랜치를 origin/master 브랜치의 트래킹 브랜치로 만든다.** 트래킹 브랜치를 직접 만들 수 있는데 리모트를 origin 이 아닌 다른 리모트로 할 수도 있고, 브랜치도 master 가 아닌 다른 브랜치로 추적하게 할 수 있다.
* git checkout -b [branch] [remotename]/[branch]

****

* 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.



* 추적 브랜치가 현재 어떻게 설정되어 있는지 확인하려면 git branch 명령에 -vv 옵션을 더하면 된다.

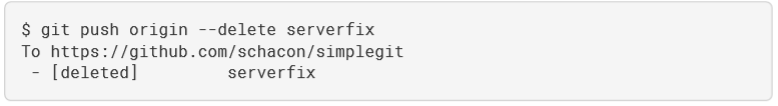
****

* 주의할 점은 이 정보는 로컬에서 서버에서 데이터를 가저온(fetch) 시점의 정보이기 때문에 부정확할 수 있다. 따라서 최신 추적 상황을 알아보려면 최신 데이터를 가져오고 나서 확인해야 한다.

****

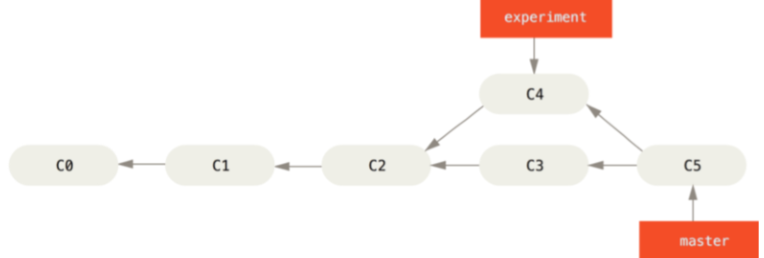
**Pull 명령**

* git pull 명령은 대부분 git fetch 명령을 실행하고 나서 자동으로 git merge 명령을 수행하는 것 뿐이다
* clone 이나 checkout 명령을 통해 추적 브랜치가 설정되면 git pull 명령은 서버로부터 데이터를 가져와서 현재 로컬 브랜치와 서버의 추적 브랜치를 Merge 한다. 일반적으로 fetch와 merge를 명시적으로 사용하는 것이 pull로 한번에 하는것보다 낫다.
* 동료와 협업하기 위해 리모트 브랜치를 만들었다가 작업을 마치고 master 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 하면 된다.

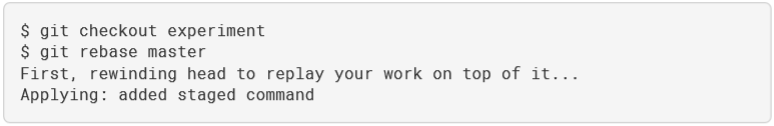
****

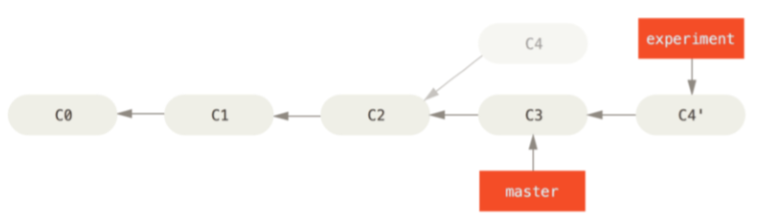
**Rebase 하기**

* Merge처럼 한 브랜치에서 다른 브랜치로 합치는 방법

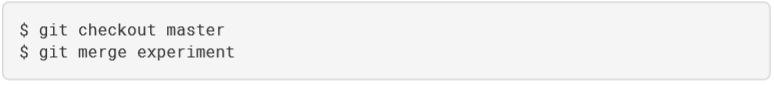
****

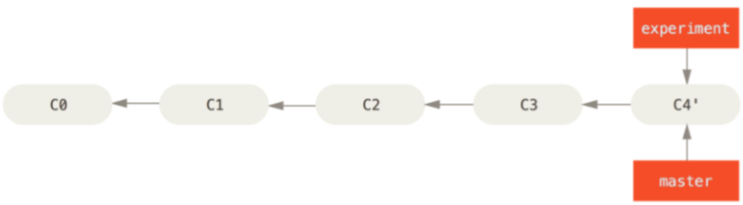
* 기존에 Merge는 C2, C3, C4로 3-Way Merge를 했다면 Rebase 방식은 C3 에서 변경된 사항을 Patch로 만들고 이를 다시 C4 에 적용시키는 방법이다.

****

****

* C4의 다른 부분을 C3에 적용하는 Rebase 과정 (C4’는 C3와 C4의 다른 부분)
* 이 상태에서 master 브랜치를 Fast-forward 시킨다.

****

****

* Merge된 것과 같은 결과이지만 커밋 히스토리가 더 깔끔하다. 마치 모든 작업이 차레대로 수행된 것처럼 보인다.
* Rebase 의 경우는 브랜치의 변경사항을 순서대로 다른 브랜치에 적용하면서 합치고 Merge 의 경우는 두 브랜치의 최종결과만을 가지고 합친다.
* 로컬 브랜치에서 작업할 때는 히스토리를 정리하기 위해서 Rebase 할 수도 있지만, 리모트 등 어딘가에 Push로 내보낸 커밋에 대해서는 Rebase를 절대 하지 말아야 한다.