PRO GIT 정리

Day 2
3. Git의 브랜치

Day 2

3.1 Git 브랜치란

- **브랜치란** : Git의 브랜치는 어떤 한 커밋을 가리키는 40글자의 SHA-1 체크섬 파일 (로컬 저장소에서/서버 연결해서)

- 코드를 통째로 복사하고 나서 원래 코드와는 상관없이 독립적으로 개발할 수 있게 함.
- Git의 최고 장점. 매우 가벼워 만들기도 쉽고 지우기도 쉬움
- 브랜치 사이를 이동 가능하기 때문에 브랜치로 작업 후 나중에 Merge함을 권장 (이전 커밋 정보를 저장하기 때문에 합쳐야하는 곳인 merge base를 알기 때문)
- 브랜치가 필요할 때 프로젝트를 통째로 복사해야 하는 다른 버전 관리 도구와 차이를 보임.

- **git의 데이터 저장**

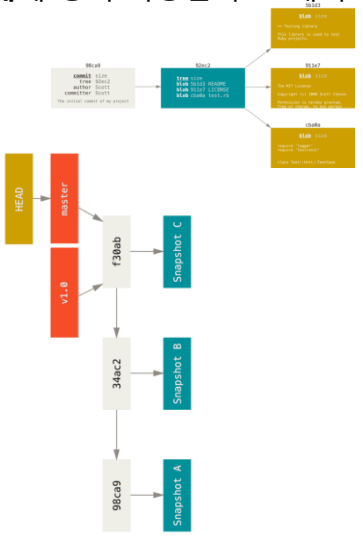
- git은 chage set이나 변경 사항 대신 **일련의 스냅샷**으로 기록

- **commit 할 때 일어나는 일**

- 현 staging area에 있는 1. 데이터의 스냅샷에 대한 포인터, 2. 저자나 커밋 메시지 같은 메타 데이터, 3. 이전 커밋에 대한 포인터 등을 포함하는 커밋 개체를 저장.
- 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.

→ 예) 파일이 3개 있는 디렉토리가 하나 있고 이 파일을 Staging Area에 저장하고 커밋할 때 수행 되는 동작

1. Stage 하면 Git저장소에 **파일**을 저장하고(Blob이라고 부름) Staging Area에 해당 파일의 **체크섬** 저장
2. commit 하면 루트 디렉토리와 각 하위 디렉토리의 **트리 개체**를 체크섬과 함께 저장소에 저장한다. 그 다음에 커밋 개체를 만들고 메타데이터와 루트 디렉토리 트리 개체를 가리키는 포인터 정보를 **커밋 개체**에 넣어 저장한다. 그래서 필요하면 언제든지 스냅샷을 다시 만들 수 있다.
3. 그 후 Git 저장소에는 다섯 개의 데이터 개체가 생김.
각 **파일**에 대한 Blob 세 개,
파일과 디렉토리 구조가 들어 있는 **트리 개체** 하나,
메타데이터와 루트 트리를 가리키는 포인터가 담긴 **커밋 개체** 하나.
4. 수정 후 커밋된 파일이면 이전 커밋이 무엇인지도 저장. (parent로)



**Git의 브랜치는 커밋 사이를 가볍게 이동할 수 있는 포인터 같은 것.

- 기본적으로 Git은 master브랜치를 만든다. 처음 커밋하면 master 브랜치가 생성된 커밋을 가리킨다. 이후 커밋을 만들면 master 브랜치는 자동으로 가장 마지막 커밋을 가리킨다. 지금 작업 중인 브랜치가 무엇인지 Git은 'HEAD'라는 특수한 포인터로 지금 작업하는 로컬 브랜치를 가리킨다.

3.1 Git 브랜치란

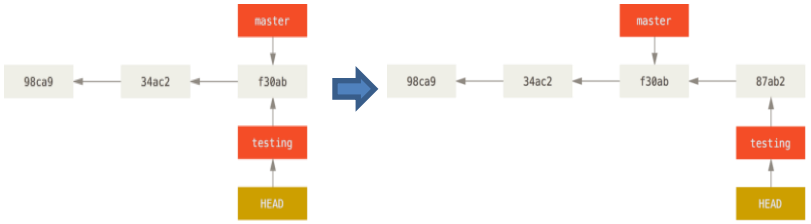
- **$ git branch <브랜치이름>** 명령

: 브랜치를 만듬 (head를 옮겨서 브랜치를 이동하지는 않음), ** 브랜치 이름 없이 쓰면 브랜치 목록을 보여줌

- **$ git log -- decorate** 옵션 명령

: 브랜치가 어떤 커밋을 가리키는지 확인

→ $ git log –online –decorate



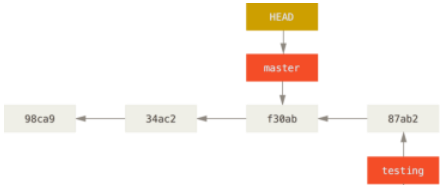
- $ **git checkout <브랜치 이름> 명령 - 이동**

: 다른 브랜치로 이동 (head가 다른 브랜치를 가리키게 함.)

- **두 브랜치가 이전 상태와 현재 상태를 가리키게 하고 새로 커밋하면**

-1 head가 가리키는 브랜치만 이동.

```
$ vim test.rb(파일이름?)
$ git commit -a -m 'made a change'
```



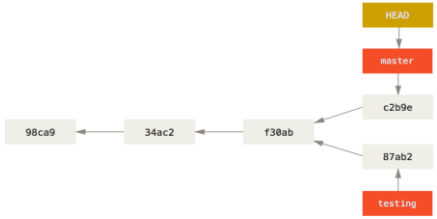
- **브랜치를 다시 master로 이동시키면**

```
$ git checkout master
```

- 브랜치가 가리키는 커밋을 HEAD가 가리키게 하고 워킹 디렉토리의 파일도 그 시점으로 되돌아감.
  - : 따라서 앞으로 커밋을 하면 다른 브랜치의 작업들과 별개로 진행되기 때문에 testing 브랜치에서 임시로 작업하고 원래 master 브랜치로 돌아와서 하던 일을 계속할 수 있다

- **파일을 수정하고 다시 커밋하면**

```
$ vim test.rb
$ git commit -a -m 'made other changes
```



- 프로젝트 히스토리가 분리돼어 진행 (갈라지는 브랜치)가 됨

: 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 Merge할 수 있다.

3.1 Git 브랜치란

- **git log --oneline --decorate --graph –all** 명령

현재 브랜치가 가리키고 있는 히스토리가 무엇이고 어떻게 갈라져 나왔는지 볼 수 있다.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain
conditions
* 98ca9 initial commit of my project
```

3.2 브랜치와 Merge의 기초

- **실제 예제**

1. 웹사이트가 있고 뭔가 작업을 진행하고 있다.
2. 새로운 이슈를 처리할 새 Branch를 하나 생성해 작업한다.
3. 이때 중요한 문제가 생겨서 그것을 해결하는 Hotfix 브랜치를 먼저 만들어야 한다.

→ 해결

s1. 새로운 이슈를 처리하기 이전의 운영(Production) 브랜치로 이동한다.
s2. Hotfix 브랜치를 새로 하나 생성한다.
s3. 수정한 Hotfix 테스트를 마치고 운영 브랜치로 Merge 한다.
s4. 다시 작업하던 브랜치로 옮겨가서 하던 일 진행한다.
s5. 작업 후 Hotfix와 merge된 운영 브랜치와 merge

3.2 브랜치와 Merge의 기초

1. 웹사이트가 있고 뭔가 작업을 진행하고 있다.
2. 53번 이슈를 처리한다고 하면 이 이슈에 집중할 수 있는 브랜치를 새로 하나 만든다.
- **git checkout -b 옵션 명령**

: 브랜치를 만들면서 Checkout까지 한 번에 (= $ git branch iss53 $ git checkout iss53 수행과 같음)

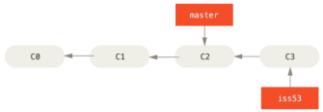
```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

- iss53 브랜치를 Checkout 했기 때문에 (즉, HEAD 는 iss53 브랜치를 가리킴) 커밋하면 iss53 브랜치가 앞으로 나아감.

3. 이때 중요한 문제가 생겨서 그것을 해결하는 Hotfix를 먼저 만들어야 한다.
- 원래의 경우 : 버그를 해결한 Hotfix에 iss53 이 섞이는 것을 방지하기 위해 iss53 과 관련된 코드를 어딘가에 저장해두고 원래 운영 환경의 소스로 복구해야 한다.
→ **git**에서는 : 그냥 master 브랜치로 돌아가 hotfix라는 브랜치를 만들고 새로운 이슈를 해결할 때까지 사용

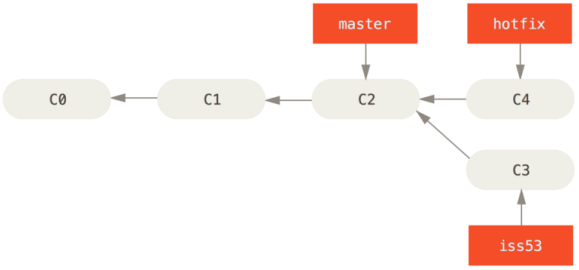
<u>s1) 작업하던 것을 모두 커밋해 워킹 디렉토리를 정리하고 master 브랜치로 옮긴다.</u>
: master 브랜치의 시점은 iss53 작업 적용 이전.

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



<u>s2) `hotfix`라는 브랜치를 만들어 사용.</u>
: 1 checkout –b 브랜치 , **2 vim** 파일 , **3 git commit –a -m** '메시지'

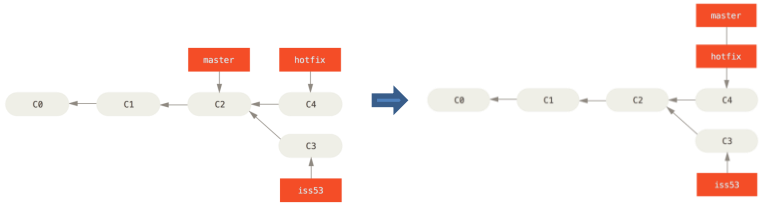
```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
1 file changed, 2 insertions(+)
```



3.2 브랜치와 Merge의 기초

s3) 급한 문제를 고친 후 운영 환경에 적용 – master 브랜치에 merge해야 한다.
- 먼저 master 브랜치로 옮겨가서 merge 시킴

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward index.html | 2 ++
1 file changed, 2 insertions(+)
```

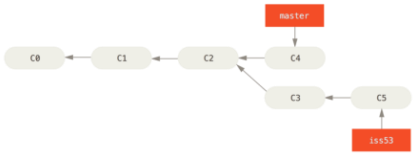


- **"fast-forward"** : 해 A 브랜치에서 다른 B 브랜치를 Merge 할 때 B 브랜치가 A 브랜치 이후의 커밋을 가리키고 있으면 그저 A 브랜치가 B 브랜치와 동일한 커밋을 가리키도록 이동시킬 뿐인 Merge 방식
위에서 작업한 hotfix 가 iss53 브랜치에 영향을 끼치지 않는다는 점을 이해하는 것이 중요

s4) 다시 일하던 iss53 브랜치로 돌아가서 하던 일 한다.

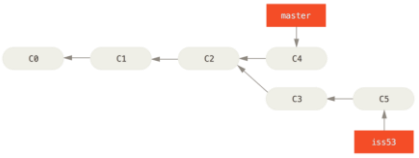
- 더 이상 필요없는 hotfix 브랜치는 삭제
  - **git branch - d <브랜치 이름>** 명령
  : 브랜치를 삭제

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



- 일하던 환경으로 되돌아가서 하던 일을 계속 (git checkout)

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```



3.2 브랜치와 Merge의 기초

- **4. 수정된 master 브랜치를 iss 브랜치에 적용하려면 → 둘을 merge시킴.**

방법1- git merge master 명령으로 master 브랜치를 iss53 브랜치에 Merge 하면 iss53 브랜치에 hotfix 가 적용된다.
방법2- iss53 브랜치가 master 에 Merge 할 수 있는 수준이 될 때까지 기다렸다가 Merge함.

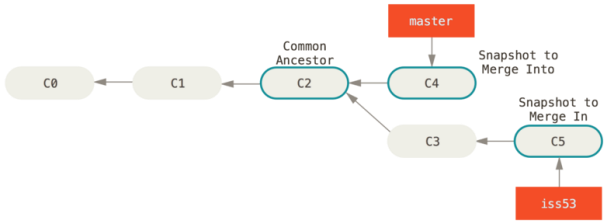
```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy. index.html | 1 +
1 file changed, 1 insertion(+)
```

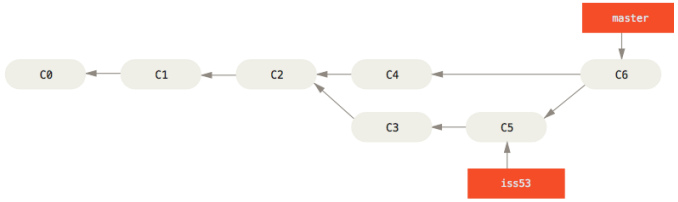
- 지금 merge하려는 두 브랜치가 조상관계가 아니므로 둘의 **공통 조상과 3-way merge**를 함

→ 3-way Merge의 결과를 별도의 커밋으로 만들고 나서 해당 브랜치가 그 커밋을 가리키도록 이동
→ **merge 커밋** : 부모를 여러 개 가진 커밋이 됨

5. iss53 브랜치 삭제

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





**** 브랜치를 이동할 때 주의!**

→ 아직 커밋하지 않은 파일이 Checkout 할 브랜치와 충돌 나면 브랜치를 변경할 수 없다.
브랜치를 변경할 때는 워킹 디렉토리를 정리하는 것이 좋다.
(나중에 Stashing과 Cleaning 에서 다룰 것)

3.2 브랜치와 Merge의 기초 – 충돌(conflict)

- **3-way merge가 실패하는 경우**

- Merge 하는 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고 Merge 하면 Git은 그 부분을 Merge 하지 못함.
- 충돌 메시지

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

- 1. Git이 어떤 파일을 Merge 할 수 없었는지 살펴보려면 **git status 명령**을 이용
    - git status 명령

충돌 난 파일 - unmerged 상태로 표시.
충돌 난 부분 – 표준 형식에 따라 표시.
→ 개발자는 해당 부분을 수동으로 해결해야함.

1. git status 명령으로 충돌난 파일과 부분 확인

```
$ git status
On branch master
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")
```

3.2 브랜치와 Merge의 기초 – 충돌(conflict)

- 충돌난 부분 표시

```
<<<<<<< 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 master 브랜치)
아래쪽은 iss53 브랜치의 내용

2. 충돌을 해결
: 위쪽이나 아래쪽 내용 중에서 고르거나 새로 작성하여 Merge.

3. 수정해서 작성한 후
1. <<<<<<<, ========, >>>>>>>`가 포함된 행을 **삭제**
2. **git add** 명령으로 다시 git에 저장
3. **git status**로 충돌이 해결된 상태인지 다시 확인

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

Changes to be committed:
modified: index.html
```

4. **git commit** 명령으로 merge한 것을 커밋한다.
- 아래 커밋 메시지 확인.

```
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 '#'
```

3.2 브랜치와 Merge의 기초 – 충돌의 기초

+ 2. 다른 merge 도구로 해결 git mergetool 명령

```
$ 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
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.
+ 고급 merge

3.3 브랜치 관리

- 브랜치 관리 명령
- **git branch 명령** : 옵션 없이 실행 시 브랜치 목록 출력, * 표시로 현재 작업 중인 브랜치 표시

```
$ git branch
  iss53
* master
```

- **- d <branch이름>** : 브랜치 삭제
  - v : 각 브랜치마다 마지막 커밋 메시지도 포함한 브랜치 목록 출력
  --merged <branch이름> : 이름 입력 안하면 현재 checkout한 브랜치 기준, merge한 브랜치 목록 출력, *로 merge해 삭제가능한 브랜치 표시
  --no-merged <브랜치> : 이름 입력 안하면 현재 checkout한 브랜치 기준 merge하지 않은 브랜치 목록 출력.
  - merge안된 커밋을 담고 있어서 git branch –d 가 에러가 뜸. **강제 삭제는 git banch** –D 이용.

3.4 branch 워크플로

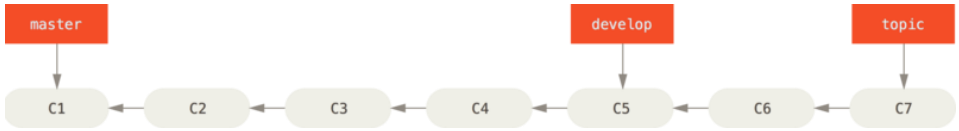
- Git 브랜치가 유용한 몇 가지 워크플로

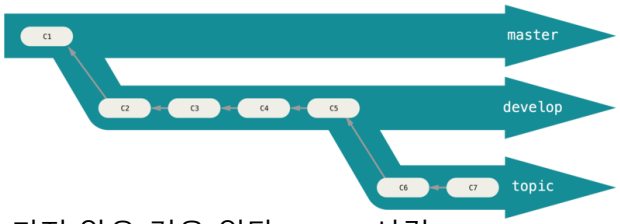
- **Long-Running 브랜치**

: 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 사용**하다 정기적으로 브랜치를 다른 브랜치로 Merge 할 수 있다.

→ 배포할 코드만 **master 브랜치에 merge해서 안정 버전의 코드만으로 구성**. 개발이나 안정화를 위한 브랜치(develop, next)를 추가로 만들어 사용하다가 안정적이라고 판단되면 master 브랜치에 merge함.

- 토픽 브랜치는 토픽 해결을 위한 브랜치(예. 앞의 iss53)
- 개발 브랜치는 공격적으로 히스토리를 만들어 나아가고 안정 브랜치는 이미 만든 히스토리를 뒤따르며 나아감.





- 코드를 여러 단계로 나누어 안정성을 높여가며 운영 가능.
- proposed 브랜치는 프로젝트 규모가 클 때 아직 develop에 merge할 준비가 되지 않은 것을 일단 merge시킴.

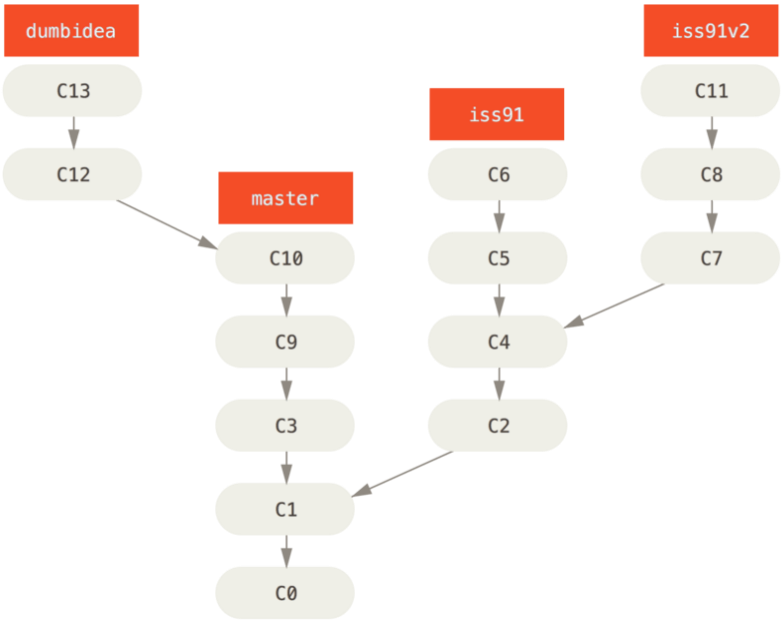
- **토픽 브랜치**

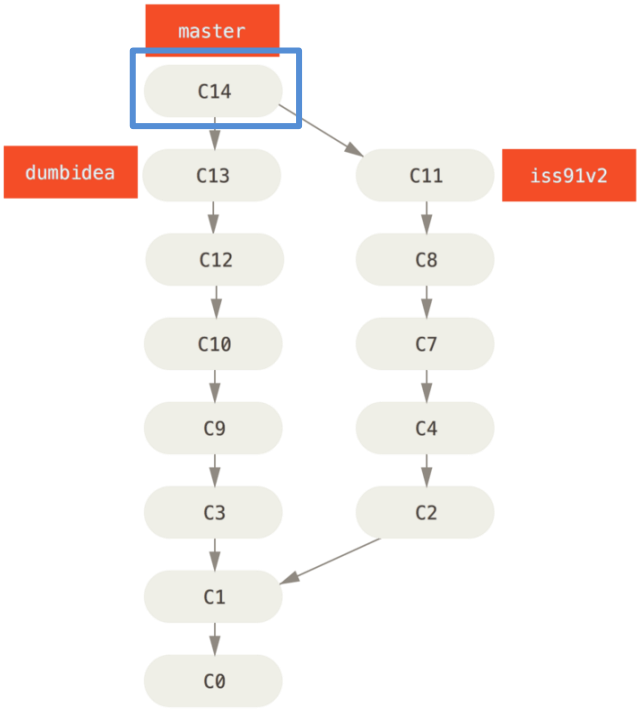
: 어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치.
- 주제별로 브랜치를 만들고 각각은 독립돼 있기 때문에 매우 쉽게 컨텍스트 사이를 옮겨 다닐 수 있다.

- 같은 이슈의 v2 브랜치 : 같은 이슈를 다른 방법으로 해결해보고 싶을 때 다른 방법을 시도해볼 브랜치
- dumbidea 브랜치 : 확신할 수 없는 아이디어를 적용해보기 위해 master 브랜치에 만든 브랜치

+ 분산 환경에서의 Git에서 다양한 워크플로 다룸

3.4 branch 워크플로





- master 브랜치에 iss91을 해결하기 위해 iss91 브랜치를 만들고 이 이슈를 다른 방법으로 시도해볼 iss91v2 브랜치를 만들었다. 또 확신할 수 없는 아이디어를 적용해보기 위해 dumbidea 브랜치를 만듬.

→ iss91v2와 dumbidea 방법이 채택되어 dumbidea 와 iss91v2 브랜치를 Merge함.(c5,c6는 버림) : 새로운 커밋 C14

3.5 리모트 브랜치 -리모트 Ref , 리모트 트래킹 브랜치

- **리모트 Ref**

: 리모트 저장소에 있는 포인터인 레퍼런스다. 리모트 저장소에 있는 브랜치, 태그, 등등을 의미한다.

  - git ls-remote [remote이름] 명령
    : 모든 리모트 Refs를 조회
  - git remote show [remote] 명령
    : 모든 리모트 브랜치와 그 정보 출력

리모트 Refs가 있지만 보통은 리모트 트래킹 브랜치를 사용

- **리모트 트래킹 브랜치**

: 리모트 브랜치를 추적하는 레퍼런스이며 브랜치로 **리모트 저장소에 마지막으로 연결했던 순간에 이 브랜치가 무슨 커밋을 가리키고 있었는지** 나타냄. 일종의 북마크 역할

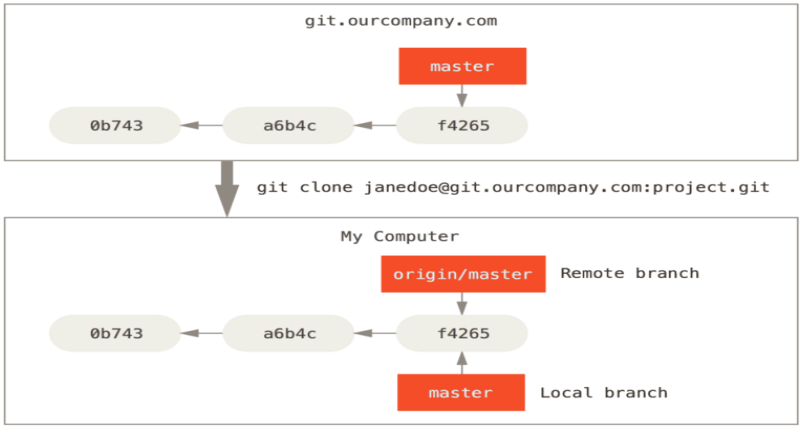
- 로컬에 있지만 임의로 움직일 수 없고 리모트 서버에 연결할 때마다 리모트의 브랜치 업데이트 내용에 따라서 자동으로 갱신.
- 이름 : <remote>/<branch> 형식 (예 리모트 저장소 origin 의 master 브랜치 : origin/master)

- 다른 팀원과 함께 어떤 이슈를 구현할 때 그 팀원이 **iss53 브랜치를 서버로 Push** 했고 당신도 **로컬에 iss53브랜치가** 있다고 가정하면, **서버의 iss53 브랜치가 가리키는 커밋은 로컬에서 `origin/iss53`이 가리키는 커밋**이다.

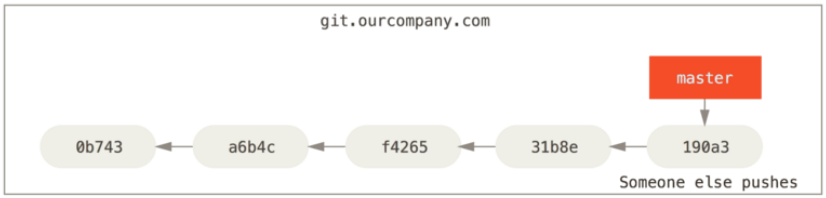
3.5 리모트 브랜치

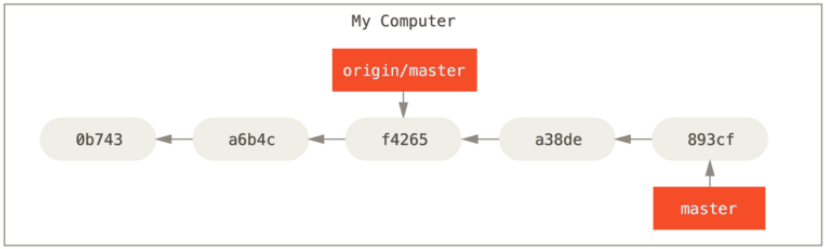
- **예시**

1. git.ourcompany.com 이라는 **Git 서버**가 있고 **이 서버의 저장소를 하나 Clone**
→ 1) Git은 자동으로 **origin** 이라는 이름을 붙이고 origin 으로부터 저장소 데이터를 모두 내려받고 2) **master 브랜치를 가리키는 포인터**를 만들어 **origin/master** 라고 부르고 이는 멋대로 조종할 수 없다. 또 3)**로컬의 master 브랜치가 origin/master 를 가리키게** 한다.



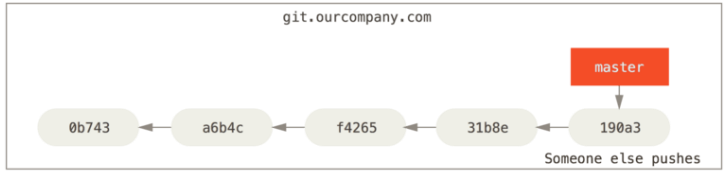
2. master 브랜치에서 작업을 하고 있었는데 동시에 다른 팀원이 **서버에 Push 하고 master 브랜치를 업데이트**.
→ 1) 팀원 간의 **히스토리는** 서로 달라진다. 2**) origin/master 포인터**는 그대로, 서버 저장소로부터 어떤 데이터도 주고받지 않았기 때문.

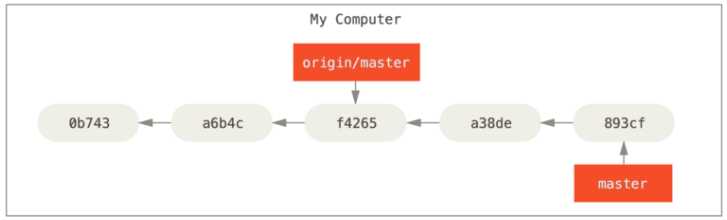




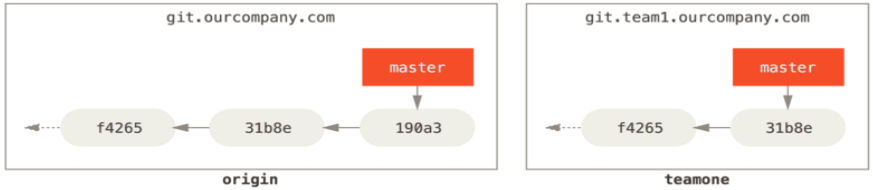
3.5 리모트 브랜치

3. **git fetch origin 명령** : 리모트 서버로부터 저장소 정보를 동기화하기 위해 사용.
→ 명령을 실행하면 우선 "origin" 서버의 주소 정보(git.ourcompany.com)에서 현재 로컬의 저장소가 갖고 있지 않은 **새로운 정보가 있으면 모두 내려받고**, 받은 데이터를 로컬 저장소에 업데이트하고 나서, **origin/master 포인터의 위치를 최신 커밋으로 이동**.

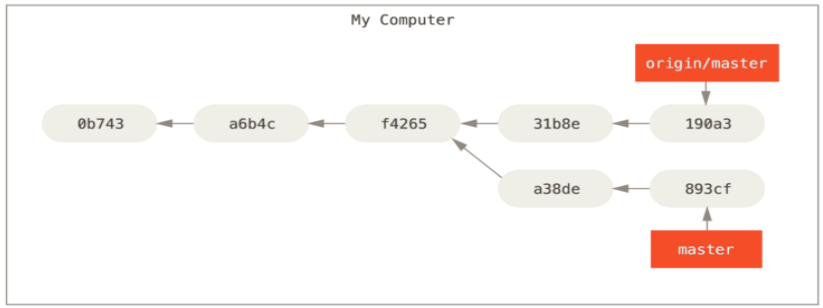




4. **git remote add** 명령으로 현재 작업 중인 프로젝트에 개발용으로 사용할 팀의 Git 저장소를 하나 추가.
(git.team1.ourcompany.com, -단축 이름을 teamone로 지정) : 겹치는 내용이 있고 origin보다 덜 업데이트된 내용을 가진 저장소임.

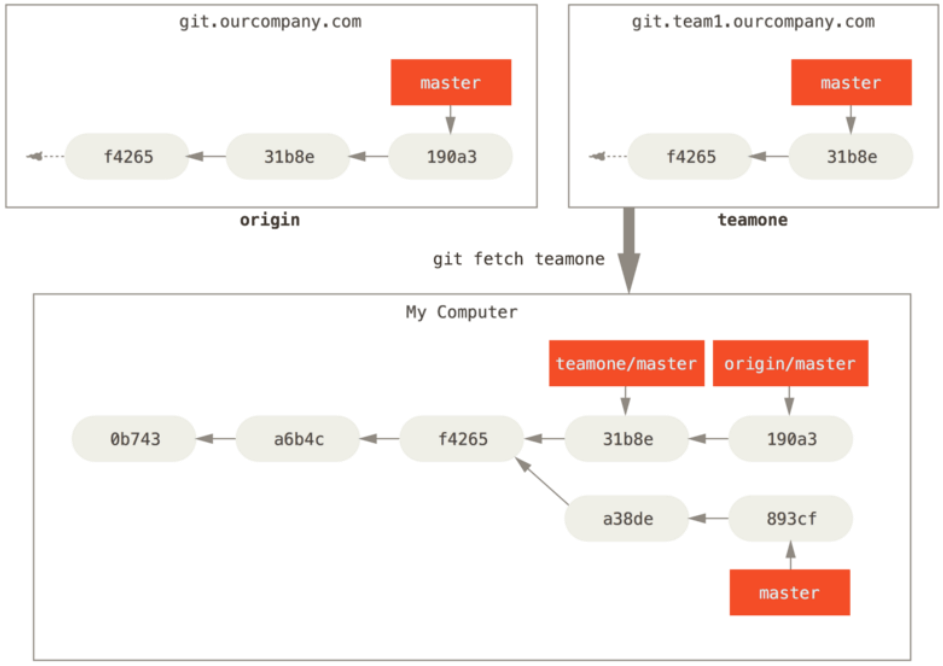


git remote add teamone git://git.team1.ourcompany.com



3.5 리모트 브랜치

5. **git fetch teamone 명령으로** teamone 서버의 데이터를 내려받는다.
→ 1)teamone의 데이터 모두 origin 서버와 겹치는 것들이라서 아무것도 내려받지 않는다. 하지만 2) **리모트 트래킹 브랜치 teamone/master 가 teamone 서버의 master 브랜치가 가리키는 커밋을 가리키게 한다.**



→ 다른 팀원과 함께 어떤 이슈를 구현할 때 그 팀원이 **iss53 브랜치를 서버로 Push** 했고 당신도 **로컬에 iss53브랜치가** 있다고 가정하면, **서버의 iss53 브랜치가 가리키는 커밋은 로컬에서 `origin/iss53`이 가리키는 커밋**이다.????

3.5 리모트 브랜치 – push 하기

- **로컬의 브랜치를 서버로 전송하려면 쓰기 권한이 있는 리모트 저장소에 Push 해야 한다.**

로컬 저장소의 브랜치는 자동으로 리모트 저장소로 전송되지 않고 명시적으로 브랜치를 Push 해야 정보가 전송된다. 다른 사람과 협업하기 위해 토픽 브랜치만 전송할 수도 있다.
따라서 리모트 저장소에 전송하지 않고 로컬 브랜치에만 두는 **비공개 브랜치**를 만들 수 있다.

  - git push <remote> <branch> 명령 상용

- serverfix 브랜치.
  - $ git push origin serverfix 명령

: serverfix라는 브랜치가 refs/heads/serverfix:refs/heads/serverfix 로 확장됨.
serverfix 라는 로컬 브랜치를 서버로 Push 하는데 리모트의 serverfix 브랜치로 업데이트한다는 것을 의미.

```
$ 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
```

  - git push origin serverfix:<리모트 저장소의 serverfix 브랜치 이름>

: 로컬의 그 이름의 브랜치를 리모트 저장소의 serverfix 브랜치로 Push, 로컬 브랜치의 이름과 리모트 서버의 브랜치 이름이 다를 때 필요. 예(git push origin serverfix:awesomebranch)

+ 나중에 Git의 내부에서 refs/heads/ 의 뜻 나옴

** git config --global credential.helper cache 명령을 실행
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.

3.5 리모트 브랜치 – push 하기

- **저장소를 Fetch 하고 나서 서버에 있는 serverfix 브랜치에 접근할 때 origin/serverfix라는 이름으로 접근.**

```
$ 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.**
→ serverfix 라는 브랜치 대신 수정 못 하는 **origin/serverfix 브랜치 포인터가** 생기는 것. ?

- git merge origin/serverfix

: 새로 받은 브랜치의 내용을 Merge 시켜 수정가능하게?

- $ git checkout -b [브랜치 이름] [리모트 트래킹 브랜치]

: 리모트 트래킹 브랜치에서 시작하는 새 브랜치를 만듬

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

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

3.5 리모트 브랜치 – 브랜치 추적

- **리모트 트래킹 브랜치(리모트 저장소에 마지막으로 연결했을 때 이 브랜치가 가리키는 커밋** 나타냄**)를 로컬 브랜치로 Checkout 하면 :** 자동으로 **"트래킹(Tracking) 브랜치"** 가 만들어짐.
- **서버로부터 저장소를 Clone**을 하면 : 자동으로 master 브랜치를 **origin/master 브랜치의 트래킹 브랜치로** 만든다.

- tracking 브랜치 : 리모트 브랜치와 직접적인 연결고리가 있는 로컬 브랜치
: 트래킹 브랜치에서 git pull 명령을 내리면 리모트 저장소로부터 데이터를 내려받아 **연결된 리모트 브랜치**와 자동으로 Merge.
- Upstream 브랜치 : 트래킹 하는 대상 브랜치

- git checkout -b <branch> <remote>/<branch> 명령
: remote/branch를 트래킹하는 트래킹 브랜치인 branch를 만들기.
--track 옵션 : 로컬 브랜치 이름을 자동으로 생성.

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

- 자주 쓰이기 때문에
1. 입력한 브랜치가 있는 **리모트가 딱 하나** 있고 2. **로컬에는 없으면 브랜치를** checkout하면 **트래킹 브랜치**를 만들어줌.

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

- **git checkout –b [트래킹 브랜치 이름지정] [리모트/브랜치] 명령**
:리모트 브랜치와 다른 이름으로 브랜치를 만들려면 로컬 브랜치의 이름을 아래와 같이 다르게 지정

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

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

3.5 리모트 브랜치 – 브랜치 추적

Upstream 별명
추적 브랜치를 설정했다면 추적 브랜치 이름을 `@{upstream}` 이나 `@{u}` 로 짧게 대체하여 사용. `master` 브랜치가 `origin/master` 브랜치를 추적하는 경우라면 `git merge origin/master` 명령과 `= git merge @{u}` 명령을 똑같이 사용.

- **이미 로컬에 존재하는 브랜치가 리모트의 특정 브랜치를 추적하게 하려면**
  - **git [존재하는 branch] –u 나 –set—upstream-to [리모트/브랜치]**

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

- git branch -vv 옵션

: 로컬 브랜치 목록과 로컬 브랜치가 추적하고 있는 리모트 브랜치도 함께 보여줌. 로컬 브랜치가 앞서가는지 뒤쳐지는지도 출력

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

= 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. master 브랜치는 origin/master 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 확인.

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

3.5 리모트 브랜치 - Pull 하기

- git fetch 명령

: 워킹 디렉토리의 파일 내용은 변경되지 않고 그대로 남기고, 서버에는 존재하지만, 로컬에는 아직 없는 데이터를 받아와서 저장, 사용자가 Merge 하도록 준비만.

- git pull 명령

: (대부분) git fetch 명령을 실행하고 나서 자동으로 git merge 명령을 수행

: clone 이나 checkout 명령을 실행하여 추적 브랜치가 설정되면 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.

협업 후 리모트 브랜치 삭제

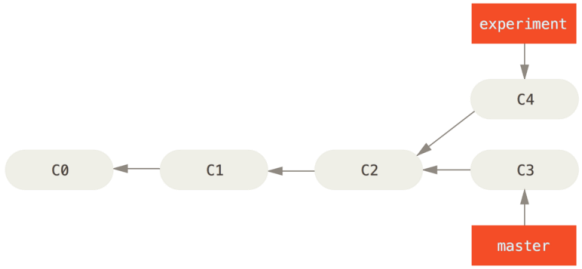
- git push 명령에 --delete 옵션

: 리모트 브랜치를 삭제

```
$ 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.

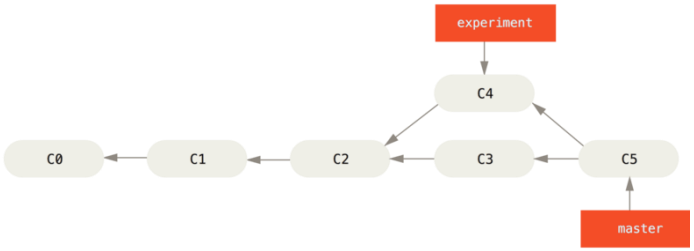
3.6 Rebase 하기



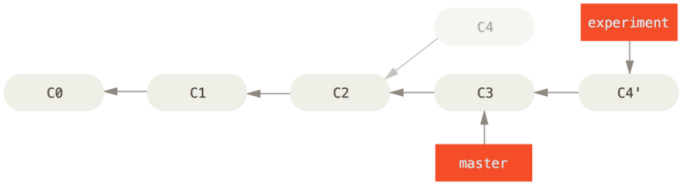
- 한 브랜치를 다른 브랜치로 합치는 방법
  = 1. merge 2. rebase

- **Rebase 의 기초**

merge 명령을 사용
: 두 브랜치의 마지막 커밋 두 개(C3, C4)와 공통 조상(C2)을 사용하는 3-way Merge로 새로운 커밋을 만들어 낸다.

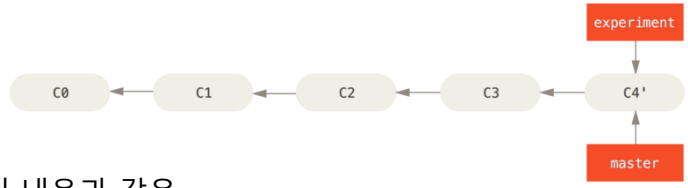


rebase 명령 사용
: C3 에서 변경된 사항을 Patch로 만들고 이를 다시 C4 에 적용시킴.
1. 일단 두 브랜치가 나뉘기 전인 공통 커밋으로 이동
2. 그 커밋부터 지금 Checkout 한 브랜치가 가리키는 커밋까지 diff를 차례로 만들어 어딘가에 임시로 저장 c4'



3. master 브랜치를 Fast-forward 시킨다.

```
$ git checkout master
$ git merge experiment
```



→ C4' 로 표시된 커밋에서의 내용은 Merge 예제에서 살펴본 C5 커밋에서의 내용과 같음
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 선형

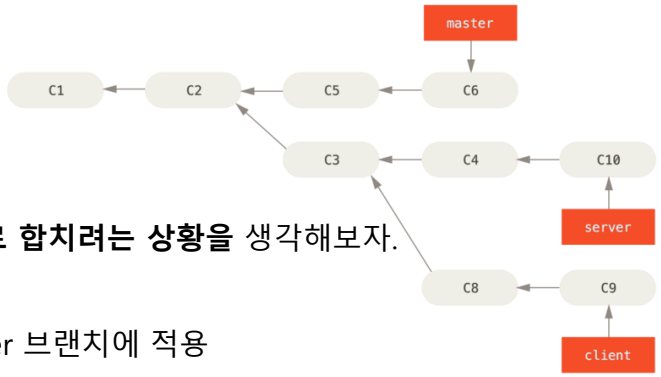
3.6 Rebase 하기

• Rebase는 보통 리모트 브랜치에 커밋을 깔끔하게 적용하고 싶을 때 사용
아마 참여하는 브랜치메인 프로젝트에 Patch를 보낼 준비가 되면 하는 것
= 브랜치에서 하던 일을 완전히 마치고 origin/master 로 Rebase 한다. 이렇게 Rebase 하고 나면 프로젝트 관리자는 어떠한 통합작업도 필요 없다. 그냥 master 브랜치를 Fast-forward 시키면 된다.

• merge/ rebase 최종 결과물은 같고 **커밋 히스토리만 다르다**
- Rebase 의 경우는 브랜치의 변경사항을 순서대로 다른 브랜치에 적용하면서 합친다.
- Merge 의 경우는 두 브랜치의 최종결과만을 가지고 합친다.

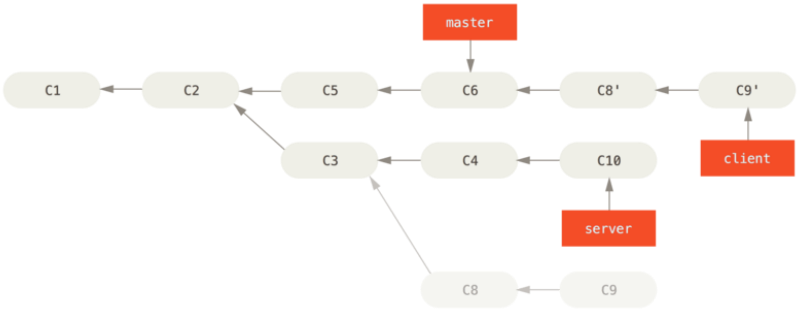
Rebase 활용
예) 다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치 같은 히스토리.
- server 브랜치를 만들어서 **서버 기능을 추가**하고
- 그 브랜치에서 다시 client 브랜치를 만들어 **클라이언트 기능을 추가**
- 마지막으로 server 브랜치로 돌아가서 **몇 가지 기능을 더 추가**

이때 테스트가 덜 된 server 브랜치는 그대로 두고 **client 브랜치만 master 로 합치려는 상황을** 생각해보자.



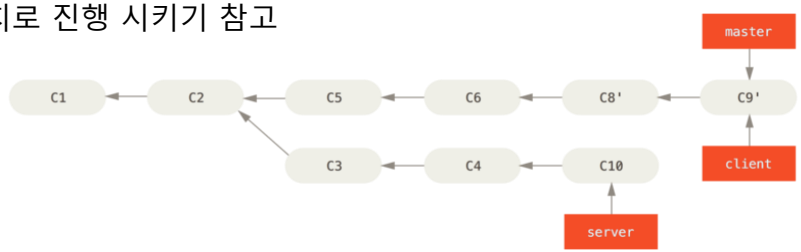
→
server 와는 아무 관련이 없는 client 커밋은 C8, C9 이다. 이 두 커밋을 master 브랜치에 적용
* git rebase –onto[master][server][client]
: master 브랜치부터 server 브랜치와 client 브랜치의 공통 조상까지의 커밋을 client 브랜치에서 없애고 싶을 때 사용
client 브랜치에서만 변경된 패치를 만들어 master 브랜치에서 **client 브랜치를 기반으로 새로 만들어 적용**한다 c8', c9'

```
$ git rebase --onto master server client
```



3.6 Rebase 하기

이제 master 브랜치로 돌아가서 Fast-forward 시킴
+ master 브랜치를 client 브랜치 위치로 진행 시키기 참고



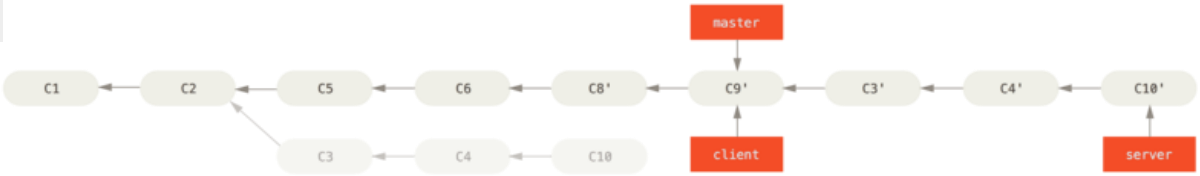
2. server 브랜치의 일이 다 끝나면

- git rebase <basebranch> <topicbranch> 명령

: Checkout 하지 않고 바로 server 브랜치를 master 브랜치로 Rebase

이 명령은 토픽(server) 브랜치를 Checkout 하고 베이스(master) 브랜치에 Rebase 한다.
server 브랜치의 수정사항을 master 브랜치에 적용 c3′c4′c10′
그 결과는 'master 브랜치에 server 브랜치의 수정 사항을 적용'과 같다.

```
$ git rebase master server
```



이제 master 브랜치를 fast-forward 시킴

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

3.6 Rebase 하기 - Rebase 의 위험성

Rebase 의 위험성

Rebase가 장점이 많은 기능이지만 단점이 없는 것은 아니니 조심해야함
주의 ! :

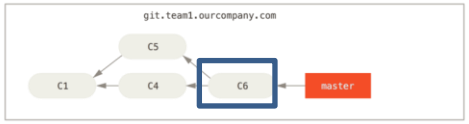
이미 공개 저장소에 Push 한 커밋을 Rebase 하지 마라

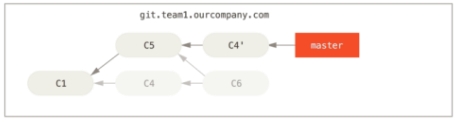
Rebase는 기존의 커밋을 그대로 사용하는 것이 아니라 내용은 같지만 다른 커밋을 새로 만든다.
새 커밋을 서버에 Push 하고 동료 중 누군가가 그 커밋을 Pull 해서 작업을 한다고 하자. 그런데 그 커밋을 git rebase 로 바꿔서 Push 해버리면 동료가 다시 Push 했을 때 동료는 다시 Merge 해야 한다. → 어제 말씀하신 base를 기반으로 HEAD가 정해져?

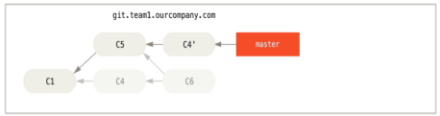
예) 공개 저장소에 Push 한 커밋을 Rebase 하면 어떤 결과가 초래되는지
1. 중앙 저장소에서 Clone 하고 일부 수정한 커밋 히스토리
2. 팀원 중 누군가 커밋, merge하고 나서 서버에 Push하면
2.1 이 리모트 브랜치를 Fetch, Merge 하면 히스토리
3. Push 했던 팀원은 **Merge 한 일을 되돌리고** 다시 Rebase하고 git push --force 명령을 사용해서 push.
3.1 fetch
4. git pull 로 서버의 내용을 가져와서 Merge 하면 같은 내용의 수정사항을 포함한 Merge 커밋이 아래와 같이 만들어진다.

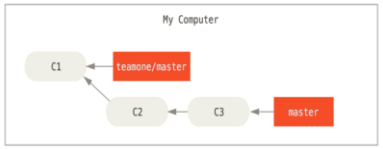
히스토리 : 저자, 커밋 날짜, 메시지가 같은 커밋이 두 개 있다(C4, C4').
이 히스토리를 서버에 Push 하면 같은 커밋이 두 개 있기 때문에 다른 사람들도 혼란스러워한다.
`C4`와 `C6`는 포함되지 말았어야 할 커밋이다.

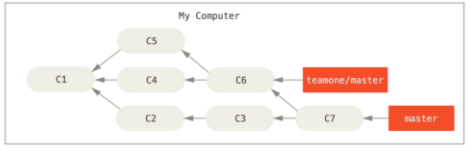
→ 애초에 서버로 데이터를 보내기 전에 Rebase로 커밋을 정리했어야 했다.

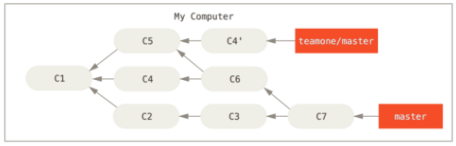


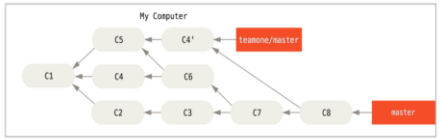




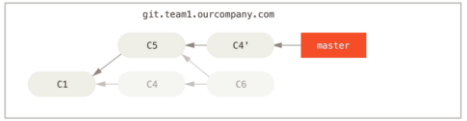




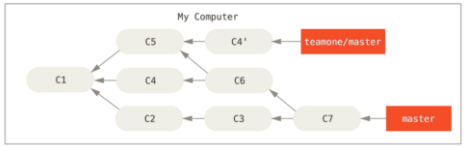




3.6 Rebase 하기 – Rebase를 rebase 하기



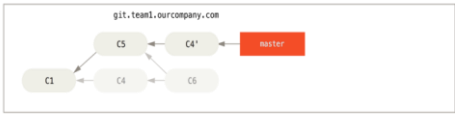
Rebase 한 것을 다시 Rebase 하기



예) 어떤 팀원이 강제로 내가 한일을 덮어썼을 때
→ 내가 했던 일이 무엇이고 덮어쓴 내용이 무엇인지 알아내야 한다.

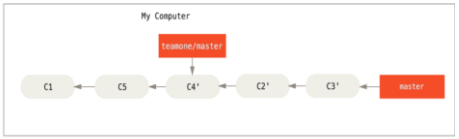
- patch-id : 커밋 SHA 체크섬 외에도 Git은 커밋에 Patch 할 내용으로 SHA-1 체크섬을 한번 더 구한다.

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



- 예) 앞 3번 한 팀원이 다른 팀원이 의존하는 커밋을 없애고 Rebase 한 커밋을 다시 Push 함 상황
 - git rebase teamone/master 명령을 실행하면 Git은 아래와 같은 작업을 한다.

1 현재 브랜치에만 포함된 커밋을 찾는다. (C2, C3, C4, C6, C7)
2 Merge 커밋을 가려낸다. (C2, C3, C4)
3 이 중 덮어쓰지 않은 커밋들만 골라낸다. (C2, C3. C4는 C4'와 동일한 Patch다)
4 **남은 커밋들** C6, C7 **만 다시 teamone/master 바탕으로 커밋을 쌓는다.**



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

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 --rebase git fetch와 git rebase teamone/master 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 후회하지 말고 git pull --rebase 로 문제를 미리 방지할 수 있다는 것을 같이 작업하는 동료와 모두 함께 공유하기 바란다.

3.6 Rebase 하기 – Rebase를 rebase 하기

Rebase vs. Merge

히스토리의 의미
1. 작업한 내용의 기록
: 작업 내용을 기록한 문서이고, 각 기록은 각각 의미를 가지며, 변경할 수 없다.
수많은 Merge 커밋이 히스토리에 남김
2. 프로젝트가 어떻게 진행되었나에 대한 이야기
: 진행 과정을 보기 좋도록 Rebase 나 filter-branch 같은 도구로 다듬으면 좋다.

주의할 점은 로컬 브랜치에서 작업할 때는 히스토리를 정리하기 위해서 Rebase 할 수도 있지만, 리모트 등 어딘가에 Push로 내보낸 커밋에 대해서는 절대 Rebase 하지 말아야 한다는 것