SVN vs GIT

svn으로 작업할 때에는 소스를 중앙 저장소에 commit 하기 전에 대부분의 기능을 완성해놓고 commit 하는 경우가 많습니다. 그도 그럴 것이, commit을 한다는 자체가 중앙 저장소에 내가 만든 기능을 공개한다는 뜻이기 때문입니다. 그래서 개발자가 자신만의 version history를 가질 수 없고, commit한 내용에 실수가 있을 시에 다른 개발자에게 바로 영향을 미치게 됩니다.
반면, **git**은 개발자가 자신만의 commit history를 가질 수 있고, 개발자와 서버의 저장소는 독립적으로 관리 가능합니다. 여기서 독립적으로 관리한다는 말은 개발자의 commit이 바로 서버에 영향을 미치지 않는다는 것입니다. 개발자는 마음대로 commit하다가 자신이 원하는 순간에 서버에 변경 내역 commit history을 보낼 수 있으며, 서버의 통합 관리자는 관리자가 원하는 순간에 각 개발자의 commit history를 가져올 수 있습니다.

통합 관리자가 각 개발자의 commit history를 가져온다는 것은, 각 개발자가 완성한 commit histroy에 대해서 통합 관리자가 차후에 아무때나 가져와 적용할 수 있다는 뜻입니다. 통합 관리자가 각 개발자의 commit history를 가져오기 전에 이미 각 개발자는 자신의 저장소와 서버의 저장소간 통합을 마친 상태이므로 통합 관리자는 별다른 어려움 없이 commit history를 가져올 수 있습니다.

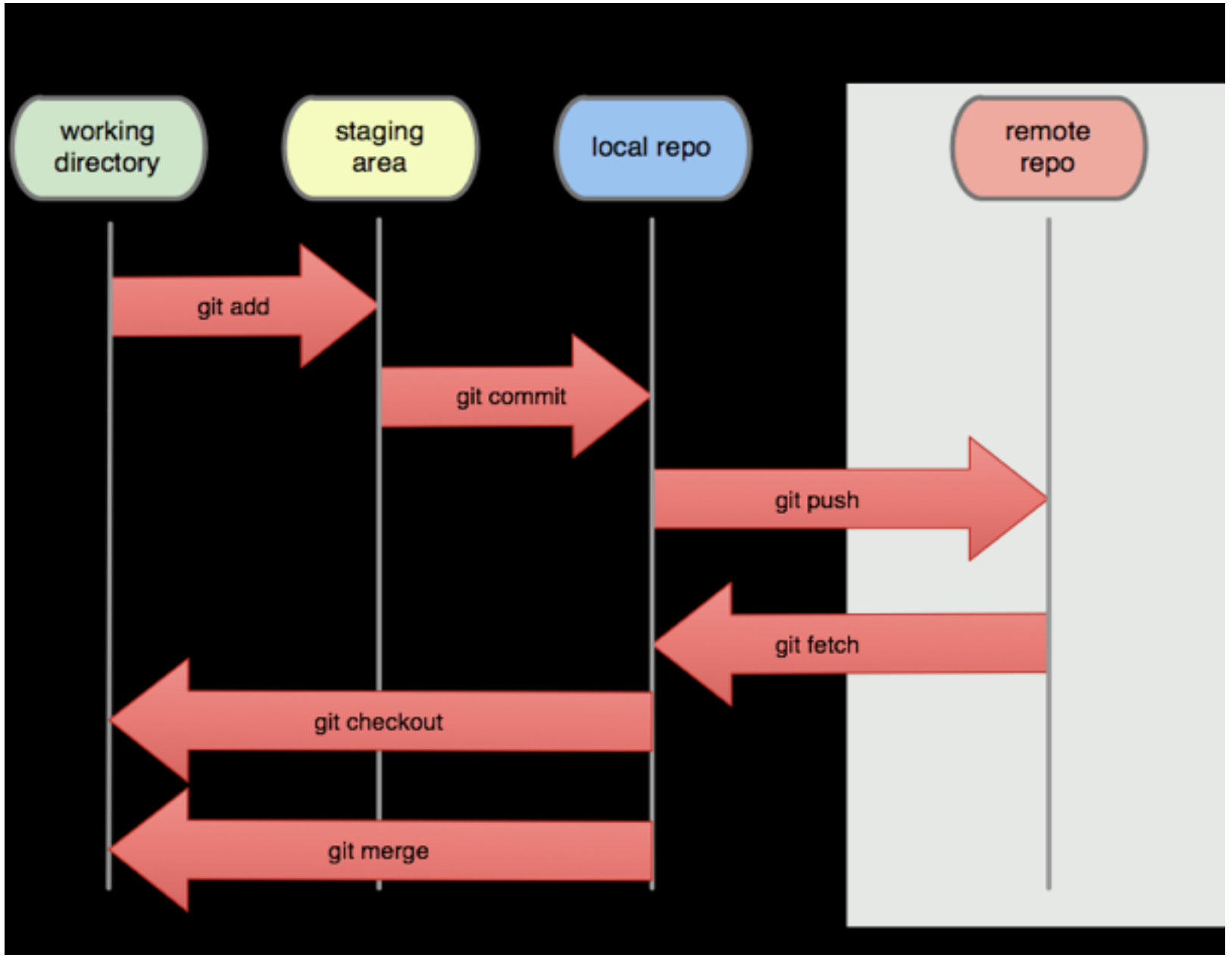
Git을 사용하면 가능한 것들

- 소스코드 주고받기가 필요 없고, 같은 파일을 여려 명이 동시에 작업하는 등 병렬 개발이 가능해지며, 버전 관리가 용이해져 생산성이 증가합니다.
- 소스코드의 수정 내용이 커밋 단위로 관리되고, 패치 형식으로 배포할 수 있기 때문에 프로그램의 변동 과정을 체계적으로 관리할 수 있고, 언제든지 지난 시점의 소스코드로 점프(Checkout)할 수 있습니다.

- 새로운 기능을 추가하는 Experimental version을 개발하는 경우, 브랜치를 통해 충분히 실험을 한 뒤 본 프로그램에 합치는 방식(Merge)으로 개발을 진행할 수 있습니다.

- '분산' 버전관리이기 때문에, 인터넷이 연결되지 않은 곳에서도 개발을 진행할 수 있으며, 중앙 저장소가 폭파되어도 다시 원상복구할 수 있습니다.

- 팀 프로젝트가 아닌, 개인 프로젝트일지라도 GIT을 통해 버전 관리를 하면 체계적인 개발이 가능해지고, 프로그램이나 패치를 배포하는 과정도 간단해집니다. (Pull을 통한 업데이트, Patch 파일 배포)



"작업한 내용을 **스테이지**에 올려서 **로컬 저장소**에 커밋하고,

이를 **푸시**해서 **원격 저장소**로 보낸다."

저장소(repository)

- 소스코드가 저장되어 있는 여러 개의 브랜치(Branch)들이 모여 있는 디스크상의 물리적 공간을 의미합니다.
- 원격 저장소만 있는 SVN과 달리, GIT에서는 저장소가 **로컬 저장소(Local Repository)**와 **원격 저장소(Remote Repository)**로 나뉩니다.
- 작업을 시작할 때 원격 저장소에서 로컬 저장소로 소스코드를 복사해서 가져오고(Clone), 이후 소스코드를 변경한 다음 커밋(Commit)을 합니다. 이 때, 커밋한 소스는 로컬 저장소에 저장되며, 푸시를 하기 전에는 원격 저장소에 반영되지 않습니다.

Git 최초설정

1. 사용자 정보

Git을 설치하고 나서 가장 먼저 해야 하는 것은 사용자 이름과 이메일 주소를 설정하는 것입니다. Git은 커밋할 때마다 이 정보를 사용합니다. 한 번 커밋한 후에는 정보를 변경할 수 없습니다.

$ git config –global user.name " John Doe "

$ git config –global user.email johndoe@example.com

2. 편집기

다시 말하자면 --global 옵션으로 설정한 것은 딱 한 번만 하면 됩니다. 해당 시스템에서 해당 사용자가 사용할 때에는 이 정보를 사용합니다. 만약 프로젝트마다 다른 이름과 이메일 주소를 사용하고 싶으면 --global 옵션을 빼고 명령을 실행합니다. 사용자 정보를 설정하고 나면 Git에서 사용할 텍스트 편집기를 고른다. 기본적으로 Git은 시스템의 기본 편집기를 사용하고 보통 Vi나 Vim입니다. 하지만, Emacs 같은 다른 텍스트 편집기를 사용할 수 있고 아래와 같이 실행하면 됩니다.

$ git config --global core.editor emacs

Git 사용하기 1. 저장소 만들기

1. 기존 디렉토리를 git 저장소로 만들기

$ git init

기존 프로젝트를 Git으로 관리하고 싶을 때, 프로젝트의 디렉토리로 이동해서 위와 같은 명령을 실행합니다.이 명령은 .git 이라는 하위 디렉토리를 만듭니다. .git 디렉토리에는 저장소에 필요한 뼈대 파일이 들어 있습니다. 이 명령만으로는 아직 프로젝트의 어떤 파일도 관리하지 않습니다.

2. 기존 저장소를 clone하기

다른 프로젝트에 참여하려거나 Git 저장소를 복사하고 싶을 때 git clone 명령을 사용합니다. 이미 Subversion 같은 VCS에 익숙한 사용자에게는 “checkout” 이 아니라 “clone” 이라는 점이 도드라져 보일 것입니다. Git이 Subversion과 다른 가장 큰 차이점은 서버에 있는 거의 모든 데이터를 복사한다는 것입니다. git clone을 실행하면 프로젝트 히스토리를 전부 받아옵니다. git clone [url] 명령으로 저장소를 Clone 합니다. libgit2 라이브러리 소스코드를 Clone 하려면 아래과 같이 실행합니다.

```
$ git clone https://github.com/libgit2/libgit2
```

이 명령은 “libgit2”이라는 디렉토리를 만들고 그 안에 .git 디렉토리를 만듭니다. 그리고 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전을 Checkout 해 놓습니다. libgit2 디렉토리로 이동하면 Checkout으로 생성한 파일을 볼 수 있고 당장 하고자 하는 일을 시작할 수 있습니다.

아래와 같은 명령을 사용하여 저장소를 Clone 하면 “libgit2”이 아니라 다른 디렉토리 이름으로 Clone 할 수 있습니다.

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

디렉토리 이름이 mylibgit 이라는 것만 빼면 이 명령의 결과와 앞선 명령의 결과는 같습니다.

2. 관리할 파일 추가 제거

디렉토리에 관리할 파일을 추가합니다.

$ touch README.md

$ git add README.md

README.md라는 파일을 추가해주었습니다. 그리고 그 파일을 git에서 관리하도록 add시켜주었습니다.

$ git status

이 명령어를 치면 현재 상태를 볼 수 있습니다.

만일 add되지 않은 파일이 있다면 붉은색으로 Untracked files 또는 Changes not staged for commit 목록이 나옵니다.

README.md는 add되었기 때문에 초록색으로 Changes to be committed 목록에 나옵니다.

** 만일 add할 파일이 많은 경우 그냥 git add *나 git add .이라고 적으면 모든 파일, 디렉토리가 add됩니다.

git add 디렉토리명을 적으면 그 디렉토리와 디렉토리의 내용 모두 add됩니다.

반대로 파일을 지울 때는 git rm을 사용합니다. git으로 파일을 지우면 git에서 그 파일이 없어졌다는 사실을 알고 더이상 추적하지 않습니다.

git을 쓰지 않고 파일을 지우거나 이름을 바꾸면 git은 추적중이던 파일이 사라졌다고 생각하기 때문에 git rm으로 사라진 파일들을 추적하지 않도록 한 번 더 지우는 작업을 해주어야 합니다.

$ git rm <지울 파일 명>
$ git rm -r <지울 디렉토리 명>

git으로 파일 이름을 바꿀 때는 git mv를 씁니다.
$ git mv <기존 파일/디렉토리 경로> <바꿀 파일/디렉토리 경로>

몇 가지 쓸모없는 파일들은 그냥 무시하고 싶다면 .gitignore라는 이름의 파일을 만들어야 합니다.

```
# 확장자가 .a인 파일 무시
*.a
# 윗 라인에서 확장자가 .a인 파일은 무시하게 했지만 lib.a는 무시하지 않음
!lib.a
# 현재 디렉토리에 있는 TODO파일은 무시하고 subdir/TODO처럼 하위디렉토리에 있는 파일은 무시하지 않음
/TODO
# build/ 디렉토리에 있는 모든 파일은 무시
build/
# doc/notes.txt 파일은 무시하고 doc/server/arch.txt 파일은 무시하지 않음
doc/*.txt
# doc 디렉토리 아래의 모든 .txt 파일을 무시
doc/**/*.txt
```

**많이 쓰이는 종류의 .gitignore은 인터넷에 찾으면 많이 나오고, gitignore.io같은 사이트도 있습니다.

```
# Compiled source #
###################
*.com
*.class
*.dll
*.exe
*.o
*.so

# Packages #
############
# it's better to unpack these files and commit the raw source
# git has its own built in compression methods
*.7z
*.dmg
*.gz
*.iso
*.jar
*.rar
*.tar
*.zip
```

.git 이 있는 디렉토리 (최상위 폴더)에 가셔서
.gitignore 파일을 만듭니다.
그리고 그 파일에 아래 예시처럼 커밋을 제외하는 룰을 넣어 주면됩니다

$ git add .
$ git commit -m "add ignore file config"
하면 동작합니다.

3. 커밋 commit

관리할 파일들을 모두 add시켜놓았으면, 이제 커밋을 해봅시다.

커밋은 간단히 말해 현재 상태를 저장하는 것입니다.

$ git commit -m <커밋 메시지>

ex) git commit -m "First commit" 커밋 메시지는 상세할수록 좋습니다.

커밋한 이후에 개발하면서 파일이 변경되면 또 git add를 해주어야 합니다.

Staged는 되지 않은 상태이기 때문입니다. 커밋은 Staged된 파일만 저장하기 때문에

git add를 하지 않으면 변경된 내용이 없다고 판단해버립니다.

매번 git add를 하는 것은 번거롭기 때문에 좀 더 편하게 커밋할 수 있는 방법을 제공합니다.

$ git commit –a

이렇게 하면 추적중인 파일들을 stage한 뒤 commit합니다.

git log로 커밋 히스토리를 조회할 수 있습니다.

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

저장소의 커밋 히스토리를 시간순으로 보여준다. 즉, 가장 최근의 커밋이 가장 먼저 나온다. 그리고 이어서 각 커밋의 SHA-1 체크섬, 저자 이름, 저자 이메일, 커밋한 날짜, 커밋 메시지를 보여준다.

4. 원격저장소

git remote 명령으로 현재 프로젝트에 등록된 **1. 원격 저장소를 확인**할 수 있습니다. 이 명령은 원격 저장소의 단축 이름을 보여줍니다. 저장소를 Clone 하면 origin이라는 원격 저장소가 자동으로 등록되기 때문에 origin이라는 이름을 볼 수 있습니다.

```
$ git remote
origin
```

2. 새 원격 저장소를 쉽게 추가할 수 있는데 git remote add [단축이름] [원격서버주소] 명령을 실행합니다.

```
$ git remote add pb https://github.com/paulboone/ticgit
```

-v옵션을 주어 단축이름과 원격서버주소를 함께 볼 수 있습니다.

```
$ git remote -v
origin	https://github.com/schacon/ticgit (fetch)
origin	https://github.com/schacon/ticgit (push)
pb	https://github.com/paulboone/ticgit (fetch)
pb	https://github.com/paulboone/ticgit (push)
```

3. 원격 저장소 pull, fetch 하기

$ git pull

pull 을 실행하면 원격 저장소의 변경된 데이터를 가져올 수 있습니다. pull 을 실행하면, 원격 저장소의 내용을 가져와 자동으로 로컬 데이터와 병합 작업을 실행하게 됩니다. 그러나 단순히 원격 저장소의 내용을 확인만 하고 로컬 데이터와 병합은 하고 싶지 않은 경우에는 fetch 명령어를 사용할 수 있습니다.

$ git fetch [원격 저장소 이름]

Fetch 를 실행하면, 원격 저장소의 최신 이력을 확인할 수 있습니다. 이 때 가져온 최신 커밋 이력은 이름 없는 브랜치로 로컬에 가져오게 됩니다.

4. 원격 저장소로 push 하기

프로젝트를 공유하고 싶을 때 원격 저장소에 Push 할 수 있습니다. 이 명령은 git push [원격 저장소 이름] [브랜치 이름]으로 단순합니다. master 브랜치를 origin 서버에 Push 하려면(다시 말하지만 Clone 하면 보통 자동으로 origin 이름이 생성된다) 아래와 같이 서버에 Push 합니다.

```
$ git push origin master
```

이 명령은 Clone 한 원격 저장소에 쓰기 권한이 있고, Clone 하고 난 이후 아무도 원격 저장소에 Push 하지 않았을 때만 사용할 수 있습니다. 다시 말해서 Clone 한 사람이 여러 명 있을 때, 다른 사람이 Push 한 후에 Push 하려고 하면 Push 할 수 없습니다. 먼저 다른 사람이 작업한 것을 가져와서 병합한 후에 Push 할 수 있습니다

5. 원격저장소 살펴보기

git remote show [원격 저장소 이름] 명령으로 원격 저장소의 구체적인 정보를 확인할 수 있습니다.

리모트 저장소의 URL과 추적하는 브랜치를 출력합니다. 이 명령은 `git pull` 명령을 실행할 때

master 브랜치와 Merge 할 브랜치가 무엇인지 보여 줍니다.

$ git remote show origin

6. 원격 저장소 이름을 바꾸거나 원격 저장소를 삭제하기

git remote rename 명령으로 원격 저장소의 이름을 변경할 수 있습니다. 예를 들어 pb를 paul로

변경하려면 git remote rename 명령을 사용합니다.

$ git remote rename pb paul

원격 저장소를 삭제해야 한다면 git remote rm 명령을 사용합니다.

$ git remote rm paul

5. Branch와 merge

- 개발을 하다보면 기존에 개발하던 부분과 다른 부분의 개발을 해야하는 상황이 발생합니다.
  예를 들어 기존 코드를 리팩토링한다고 생각해봅시다. 리팩토링이 끝나기 전까지는 프로그램에
  기능을 추가하는 것이 어렵습니다. 만일 여러 명이서 작업하고 있었다면, 이것은 전체가
  올스톱되는 상황을 만듭니다.

- 이럴 때, 서로 독립적으로 평행하게 개발을 진행하기 위해서 브랜치를 만듭니다.
  브랜치는 분리된 작업 영역이라고 생각하면 됩니다.
  한 브랜치는 기능을 추가하는 개발을 하고,
  한 브랜치는 기존 코드를 리팩토링하는 개발을 진행하는 것입니다.

- 브랜치가 나뉘어있으면, 둘 사이의 충돌이 일어나지 않습니다. 일종의 평행세계입니다.
  물론 나중에 하나로 묶는 과정에서 충돌이 일어나기도 하지만 그 전까지는 평화롭습니다.

- 기본적으로 git을 초기화하면 master 브랜치가 생깁니다. 그리고 commit을 하면
  이 master 브랜치에서 커밋이 일어납니다.

실제 개발과정에서 겪을 만한 예제를 하나 살펴봅시다.

브랜치와 Merge는 보통 이런 식으로 진행합니다:

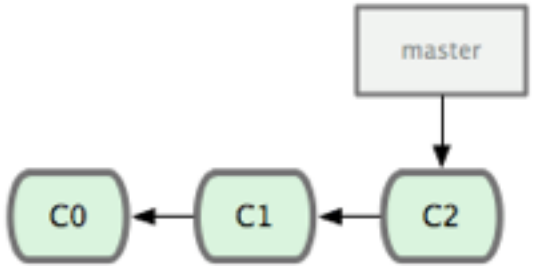
1. 작업 중인 웹사이트가 있다.
2. 새로운 이슈를 처리할 새 Branch를 하나 생성.
3. 새로 만든 Branch에서 작업 중.

이때 중요한 문제가 생겨서 그것을 해결하는 Hotfix를 먼저 만

들어야 합니다. 그러면 다음과 같이 할 수 있습니다:

1. 새로운 이슈를 처리하기 이전의 운영(Production) 브랜치로 이동.
2. Hotfix 브랜치를 새로 하나 생성.
3. 수정한 Hotfix 테스트를 마치고 운영 브랜치로 Merge.
4. 다시 작업하던 브랜치로 옮겨가서 하던 일 진행.

먼저 커밋을 몇 번 했다고 가정합니다.



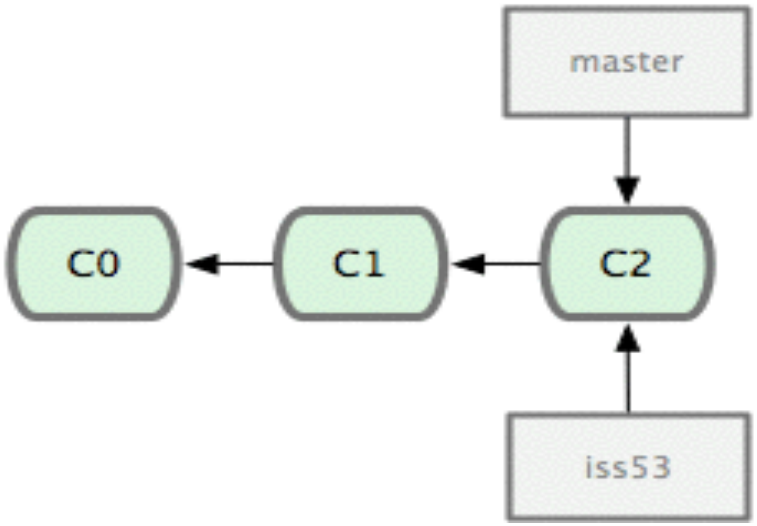
현재 커밋 히스토리

그런데 이슈 관리 시스템에 등록된 53번 이슈를 처리한다고 하면
이 이슈에 집중할 수 있는 브랜치를 새로 하나 만듭니다.
Git은 어떤 이슈 관리 시스템에도 종속돼 있지 않습니다.
브랜치를 만들면서 Checkout까지 한 번에 하려면 git checkout 명령에
–b라는 옵션을 줍니다.

$ git checkout –b iss53
Switched to a new branch ' iss53‘

위 명령은 아래 명령을 줄여놓은 것입니다:

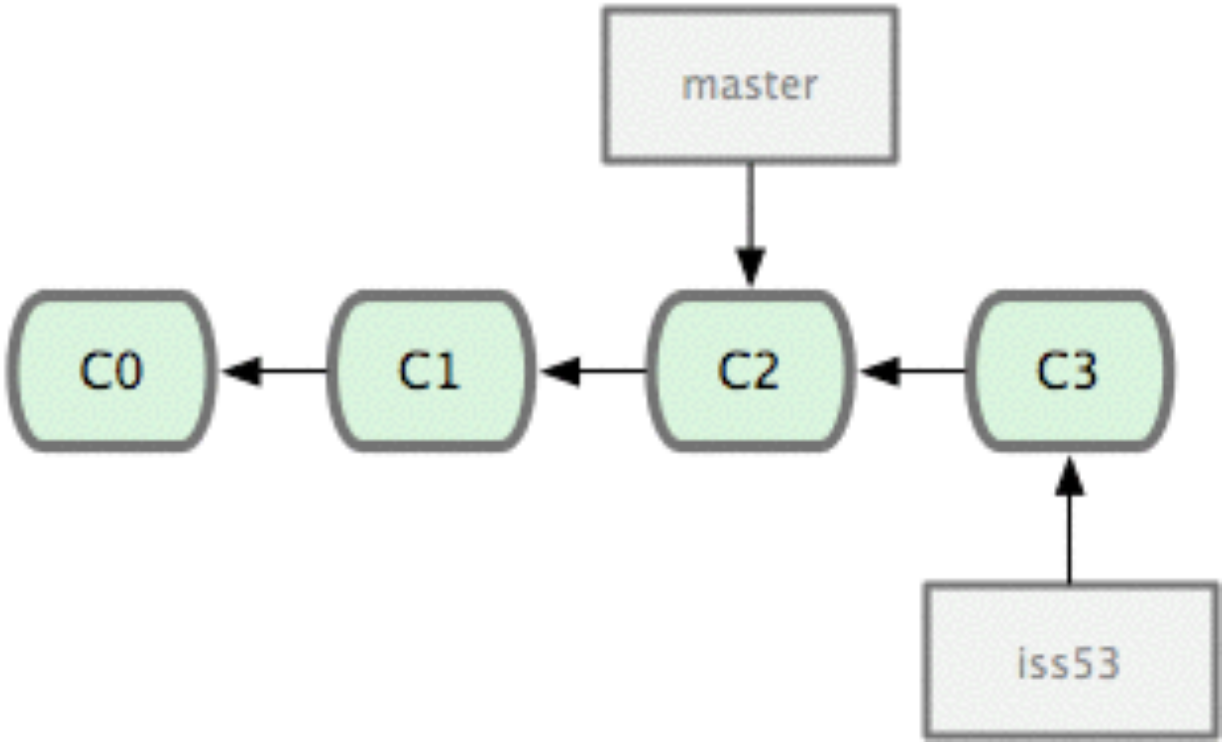
$ git branch iss53 =>만들고
$ git checkout iss53 =>옮겨가고



iss53 브랜치를 Checkout했기 때문에(즉, HEAD는 iss53 브랜치를 가리킨다) 뭔가 일을 하고 커밋하면 iss53 브랜치가 앞으로 진행합니다:

```
$ vim index.html
```

```
$ git commit -a -m 'added a new footer [issue 53]'
```



다른 상황을 가정해봅시다. 만드는 사이트에 문제가 생겨서 즉시 고쳐야 합니다. 버그를 해결한 Hotfix에 'iss53'이 섞이는 것을 방지하기 위해 'iss53'와 관련된 코드를 어딘가에 저장해두고 원래 운영 환경의 소스로 복구해야 합니다. Git을 사용하면 이런 노력을 들일 필요 없이 그냥 master 브랜치로 옮기면 됩니다.

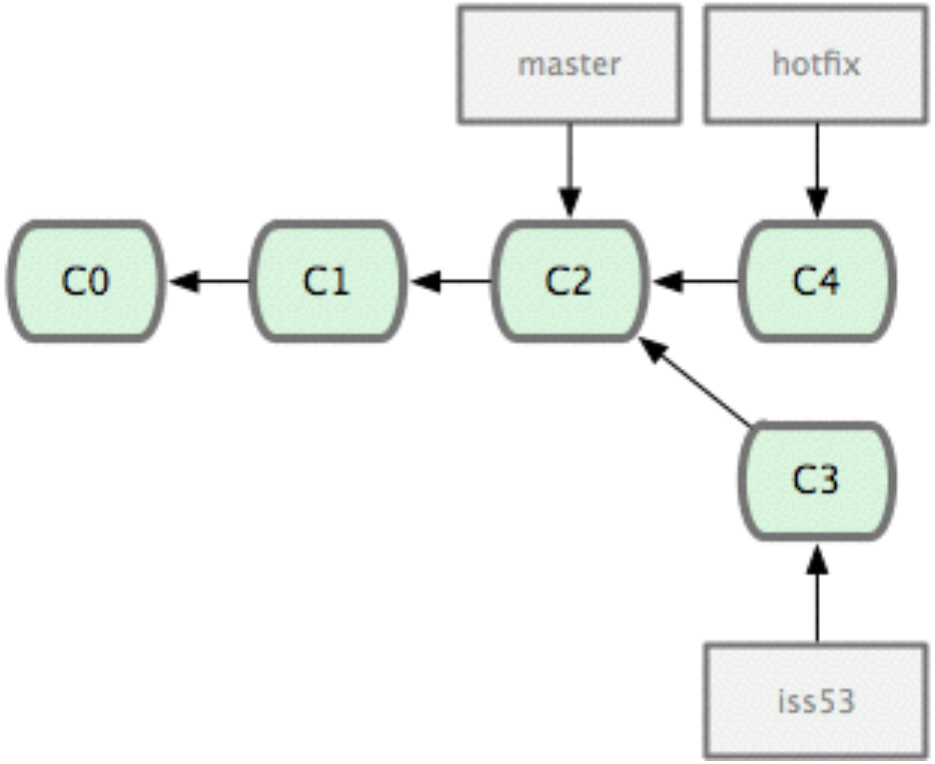
그렇지만, 브랜치를 이동하려면 해야 할 일이 있습니다. 아직 커밋하지 않은 파일이 Checkout할 브랜치와 충돌 나면 브랜치를 변경할 수 없습니다. 브랜치를 변경할 때에는 워킹 디렉토리를 정리하는 것이 좋습니다. 이런 문제를 다루는 방법은(주로, Stash이나 커밋 Amend에 대해) 나중에 다룰 것입니다. 지금은 작업하던 것을 모두 커밋하고 master 브랜치로 옮깁니다:

이때 워킹 디렉토리는 53번 이슈를 시작하기 이전 모습으로 되돌려지기 때문에 새로운 문제에 집중할 수 있는 환경이 만들어집니다.

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

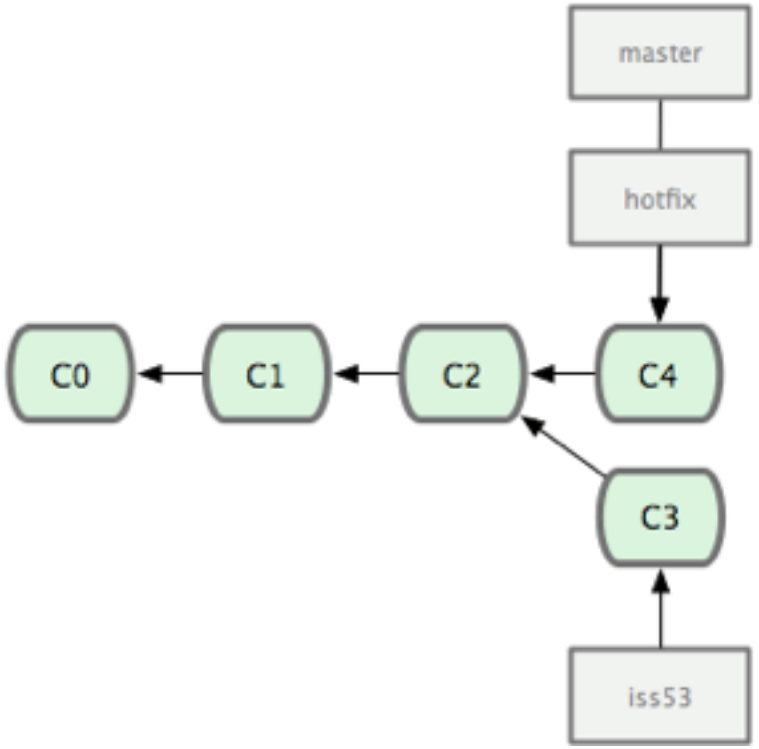
hotfix라는 브랜치를 만들고 새로운 이슈를 해결할 때까지 사용합니다:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 3a0874c] fixed the broken email address
 1 files changed, 1 deletion(-)
```



운영 환경에 적용하려면 문제를 제대로 고쳤는지 테스트하고 master 브랜치에 합쳐야 합니다. Git merge 명령으로 다음과 같이 합니다:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```



Merge할 브랜치가 가리키고 있던 커밋이 현 브랜치가 가리키는 것보다 '앞으로 진행한'

커밋이기 때문에 master 브랜치 포인터는 최신 커밋으로 이동합니다.

이런 Merge 방식을 **'Fast forward**'라고 부릅니다.

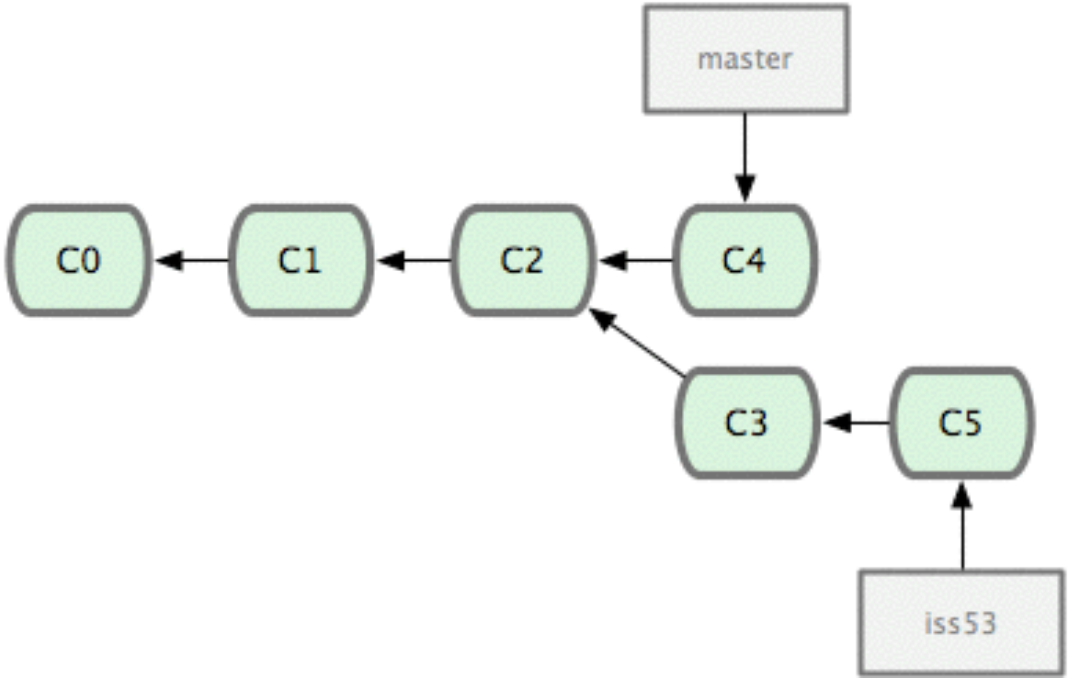
이제 hotfix는 master 브랜치에 포함됐고 운영환경에 적용할 수 있습니다.

문제를 급히 해결하고 master 브랜치에 적용하고 나면 다시 일하던 브랜치로 돌아가야 합니다. 하지만, 그전에 필요없는 hotfix 브랜치를 삭제합니다. git branch 명령에 -d 옵션을 주고 브랜치를 삭제합니다.

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

이제 이슈 53번을 처리하던 환경으로 되돌아가서 하던 일을 계속합니다:

```
$ 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(+)
```



53번 이슈를 다 구현하고 master 브랜치에 Merge하는 과정을 살펴보자. master 브랜치에 Merge하는 것은 앞서 살펴본 hotfix 브랜치를 Merge하는 것과 비슷합니다. Git merge 명령으로 합칠 브랜치에서 합쳐질 브랜치를 Merge하면 됩니다:

```
$ git checkout master
$ git merge iss53
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

hotfix를 Merge했을 때와 메시지가 다릅니다.

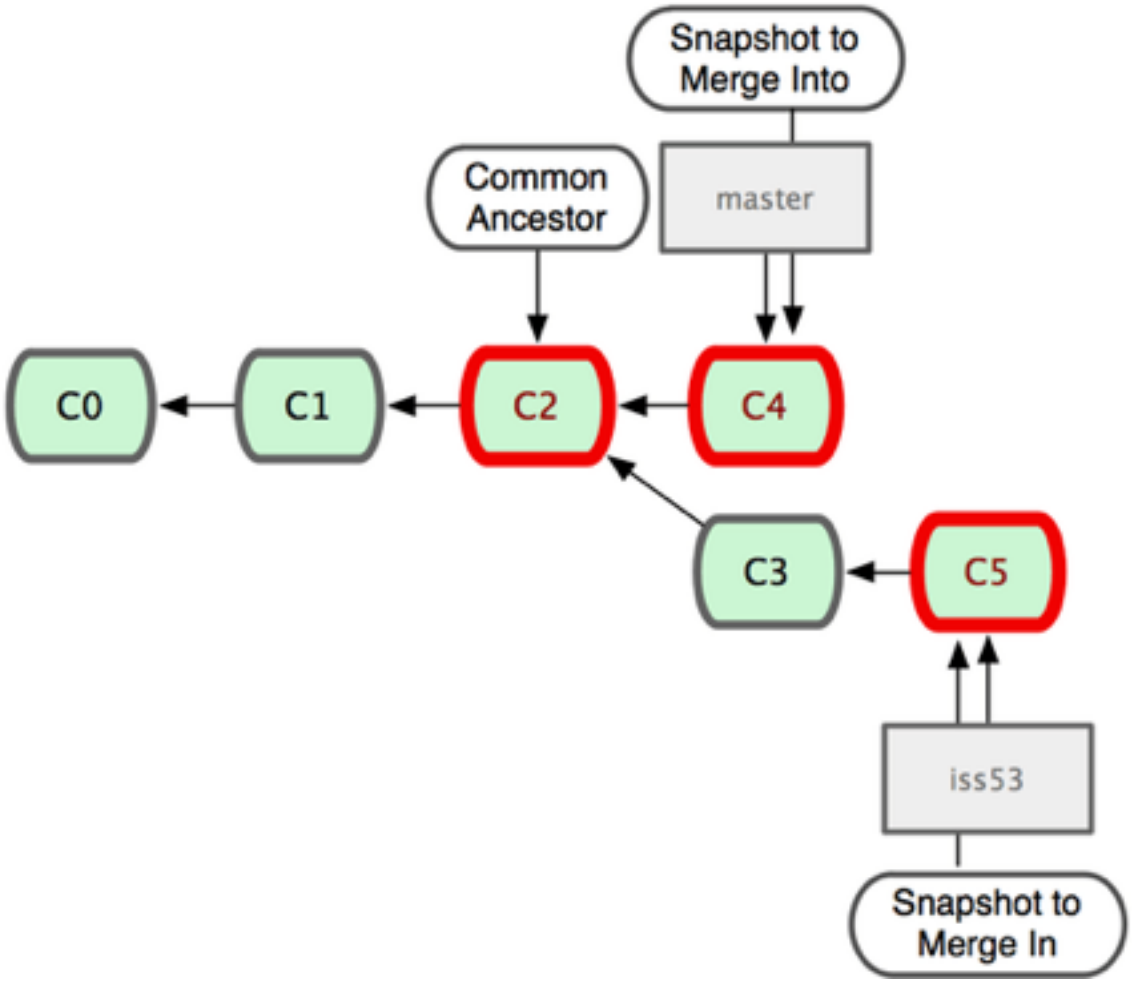
현 브랜치가 가리키는 커밋이 Merge할 브랜치의 조상이

아니므로 Git은 'Fast-forward'로 Merge하지 않습니다.

이러면 Git은 각 브랜치가 가리키는 커밋 두 개와

공통 조상 하나를 사용하여 3-way Merge를 합니다.

그림에 이 Merge에서 사용하는 커밋 세 개가 표시됩니다.

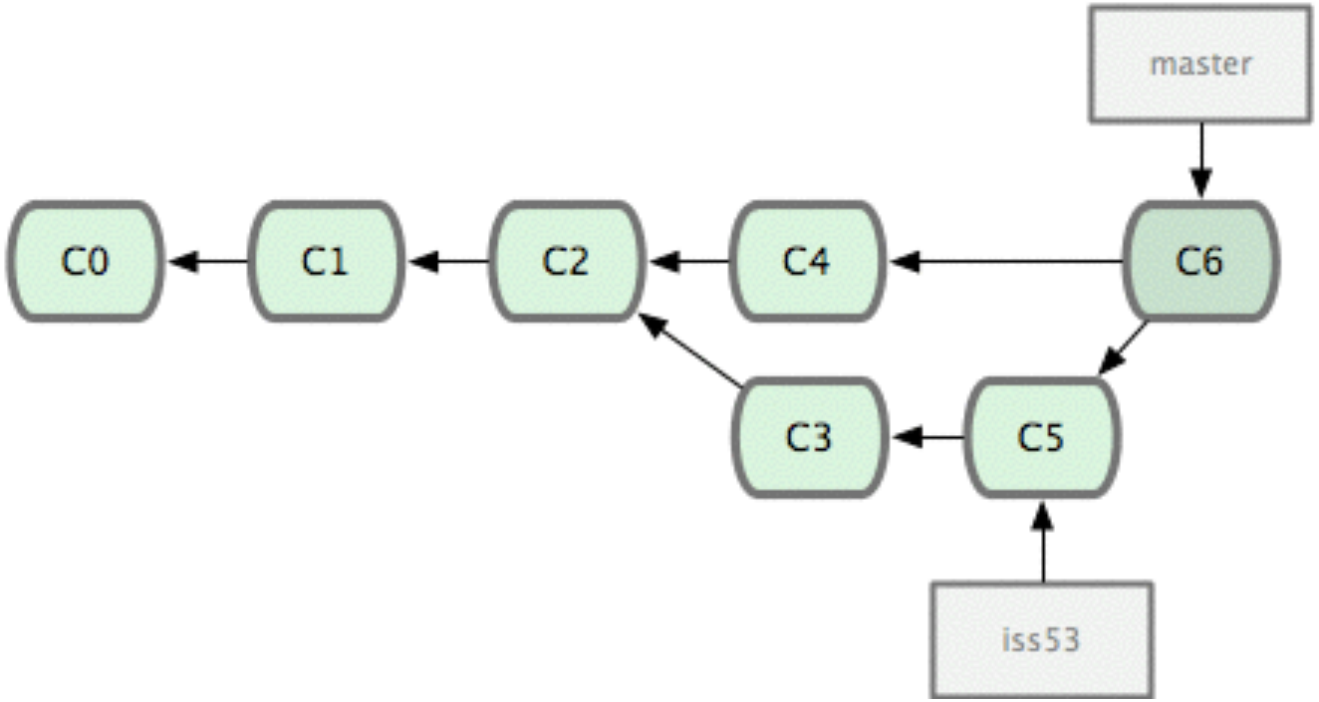


단순히 브랜치 포인터를 최신 커밋으로 옮기는 게 아니라 3-way Merge의 결과를 별도의 커밋으로 만들고 나서 해당 브랜치가 그 커밋을 가리키도록 이동시킵니다. 그래서 이런 커밋은 부모가 여러 개고 Merge 커밋이라고 부릅니다.

Git은 Merge하는데 필요한 최적의 공통 조상을 자동으로 찾습니다. 이런 기능도 Git이 다른 버전 관리 시스템보다 나은 점입니다. CVS나 Subversion 같은 버전 관리 시스템은 개발자가 직접 공통 조상을 찾아서 Merge해야 합니다. Git은 다른 시스템보다 Merge가 대단히 쉽습니다.

iss53 브랜치를 master에 Merge하고 나면 더는 iss53 브랜치가 필요 없습니다. 다음 명령으로 브랜치를 삭제하고 이슈의 상태를 처리 완료로 표시합니다:

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



6. 충돌

가끔씩 3-way Merge가 실패할 때도 있습니다. Merge하는 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고

Merge하면 Git은 해당 부분을 Merge하지 못합니다. 예를 들어, 53번 이슈와 hotfix가 같은 부분을 수정했다면

Git은 Merge하지 못하고 다음과 같은 충돌(Conflict) 메시지를 출력합니다:

```
$ 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 않습니다. 변경사항의 충돌을 개발자가 해결하지 않는 한 Merge 과정을 진행할 수 없습니다. Merge 충돌이 일어났을 때 Git이 어떤 파일을 Merge 할 수 없었는지 살펴보려면 git status 명령을 이용합니다.

```
$ 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")
```

충돌이 일어난 파일은 unmerged 상태로 표시됩니다. Git은 충돌이 난 부분을 표준 형식에 따라 표시해줍니다.

그러면 개발자는 해당 부분을 수동으로 해결합니다. 충돌 난 부분은 아래와 같이 표시됩니다.

```
<<<<<<< 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
```

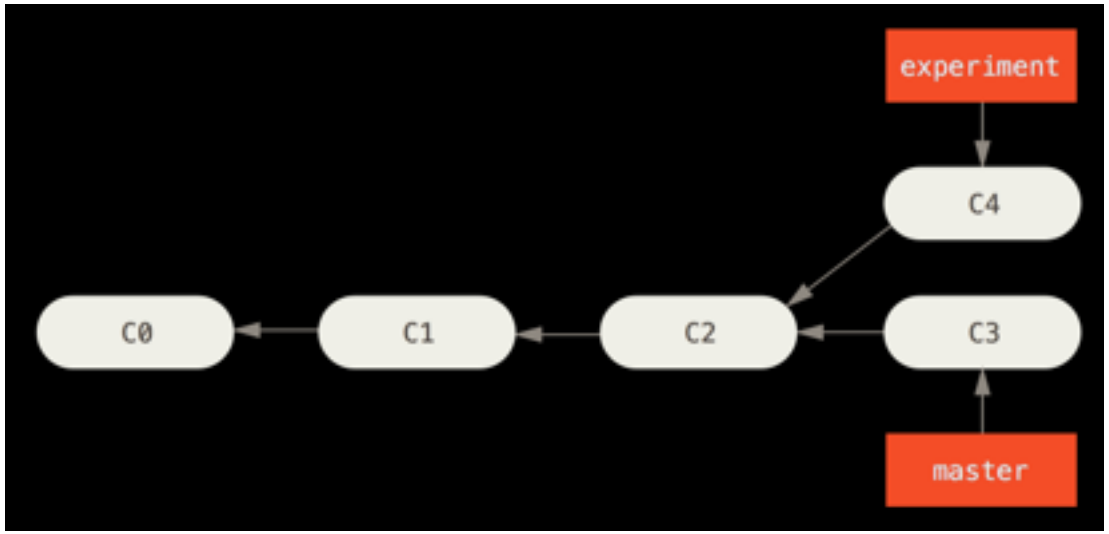
======= 위쪽의 내용은 HEAD 버전(merge 명령을 실행할 때 작업하던 master 브랜치)의 내용이고 아래쪽은 iss53 브랜치의 내용입니다. 충돌을 해결하려면 위쪽이나 아래쪽 내용 중에서 고르거나 새로 작성하여 Merge 합니다. 아래는 아예 새로 작성하여 충돌을 해결하는 예제입니다.

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

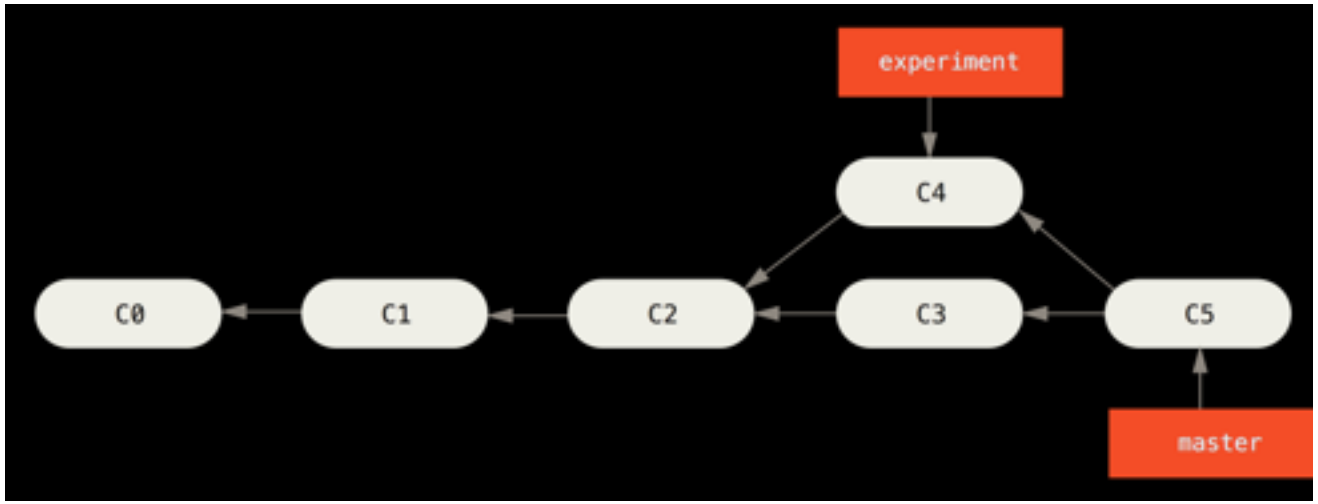
충돌한 양쪽에서 조금씩 가져와서 새로 수정했다. 그리고 <<<<<<<, =======, >>>>>>>가 포함된 행을 삭제하였습니다. 이렇게 충돌한 부분을 해결하고 git add 명령으로 다시 Git에 저장합니다.

7. 리베이스 rebase

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



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



이 두 브랜치를비슷한 결과를 만드는 다른 방식으로, C3에서 변경된 사항을 Patch(Patch)로 만들고 이를 다시 C4에 적용시키는 방법이 있습니다. Git에서는 이런 방식을 Rebase라고 한다. rebase 명령으로 한 브랜치에서 변경된 사항을 다른 브랜치에 적용할 수 있습니다.

위의 예제는 아래와 같은 명령으로 Rebase 합니다.

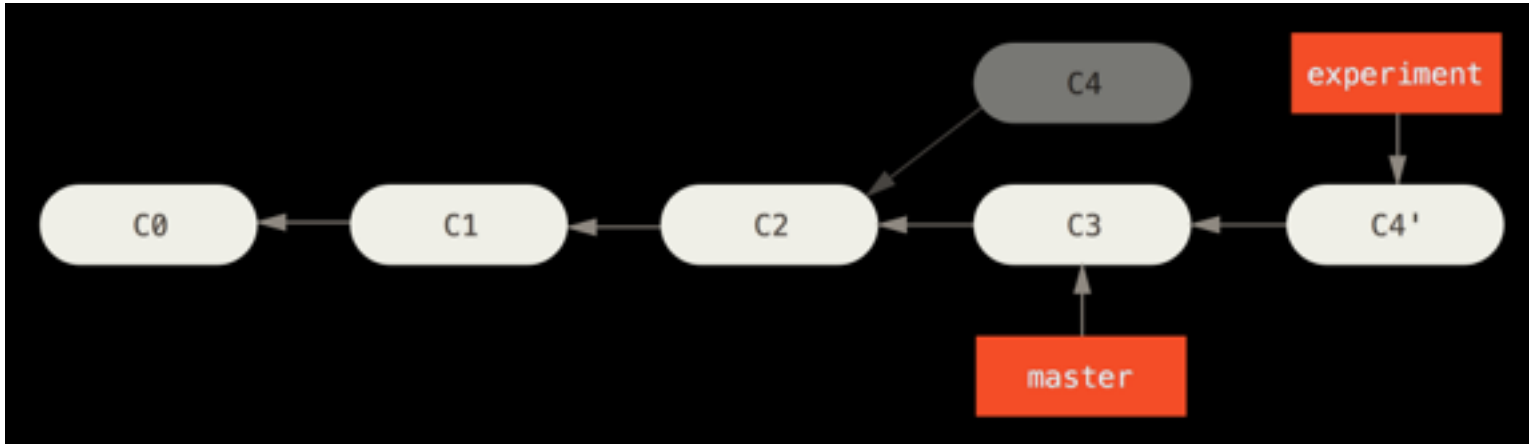
$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

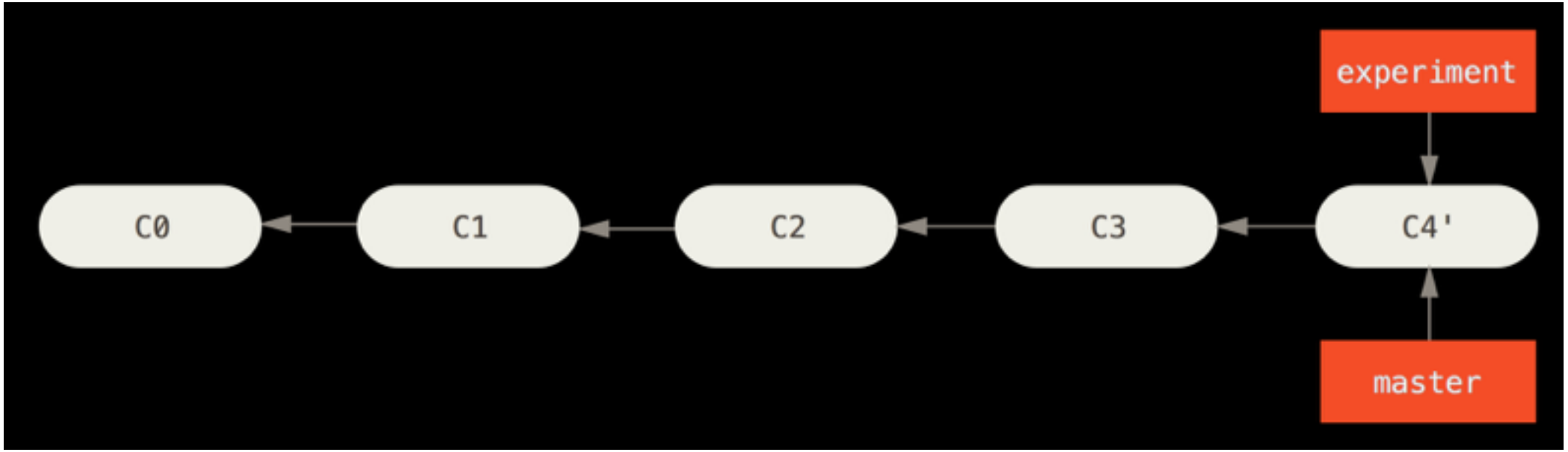
Applying: added staged command

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 놓습니다. Rebase 할 브랜치(experiment)가 합칠 브랜치(master)가 가리키는 커밋을 가리키게 하고 아까 저장해 놓았던 변경사항을 차례대로 적용한다.



그리고 나서 master 브랜치를 Fast-forward 시킵니다.

$ git checkout master
$ git merge experiment



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

Rebase의 위험성

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

이미 공개 저장소에 Push 한 커밋을 Rebase 하지 마라

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

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

8. 복구

개발 도중에는 코드가 엉켜서 도저히 나아가기 힘든 상황에 놓이기도 합니다.

이럴 때 git을 사용해 과거의 코드를 다시 가져올 수 있습니다.

$ git checkout [<커밋 이름>] -- <파일 명>

[]는 생략 가능한 부분입니다. 커밋 이름을 생략하면, 현재 가리키는중인 커밋(HEAD)을 사용합니다.

커밋 이름은 git log를 하면 나오는 커밋의 해시값입니다.

전부 다 적을 필요는 없고 앞에 네 글자만 적어도 대부분 됩니다.

해시값 말고 branch 이름도 가능합니다.

그 외에 HEAD라는 상수도 사용할 수 있는데, 이는 현재 가리키고 있는 커밋을 말합니다.

커밋 이름 뒤에 ^를 붙이면 그 커밋의 부모(바로 전) 커밋을 의미하고,

~<숫자>를 붙이면 그 커밋의 <숫자>번째 부모(전) 커밋을 의미합니다.

예를 들어 최근 커밋한 a.txt로 복구하고 싶다면 git checkout -- a.txt라고 적으면 됩니다.

또는 f5d4 커밋때의 모든 파일을 가져오고 싶다면 git checkout f5d4 -- *라고 적으면 됩니다.

커밋 자체를 취소하고 싶을 경우도 있습니다.

- 방금 한 커밋을 수정하고 싶을 경우

$ git commit –amend

- 메시지를 잘못 적었거나, 깜빡하고 add하지 못한 파일이 있거나 할 때,

git add를 하고나서 git commit --amend -m <새 메시지>를 적으면 방금 했던 커밋이 수정됩니다.

- 커밋 자체를 취소하고 싶은 경우

$ git reset <돌아갈 커밋 이름>

$ git revert <취소할 커밋>

reset과 revert의 작동 방식은 조금 다릅니다. reset은 돌아갈 커밋 이름으로 적은 커밋으로 롤백합니다.

바로 전으로 돌아가려면 git reset HEAD^나 git reset HEAD~1을 적으면 됩니다.

revert는 취소할 커밋의 변경 내용을 정확히 거꾸로 수행한 새로운 커밋을 만듭니다.

결과적으로는 둘 다 커밋하기 전 상태로 돌아갑니다.

대신 reset은 커밋했던 히스토리가 남지 않을 수도 있습니다.

revert는 커밋했던 히스토리가 남습니다.

9. stash

작업 도중에 커밋하지 않고 다른 브랜치로 이동하고 싶다거나 새로 pull하고 싶다거나 merge해야하는 경우
커밋하지 않은 파일들이 남아있으면 에러를 내면서 커밋하거나 Stash하라는 말이 나옵니다.
아직 커밋을 할 상황이 아닌 경우 Stash를 해서 지금까지의 작업을 잠시 백업하고 마지막 커밋 상황으로
돌려놓을 수 있습니다.

```
$ git stash
```

stash를 하게 되면 지금까지 작업한 내용은 모두 사라집니다.
그 다음 브랜치를 옮기거나 pull, merge 등을 합니다.
할 일을 끝내고 다시 작업하던 내용을 복구하고자 할 경우에는 stash pop을 하면 됩니다.

```
$ git stash pop
```

그러면 stash했던 파일들이 복구됩니다.

10. submodule

한 프로젝트에서 다른 프로젝트의 파일들을 사용한다면 submodule 기능을 사용할 수 있습니다.

$ git submodule add <프로젝트 경로> <지정할 이름>

프로젝트 경로는 원격 저장소의 경로도 가능합니다.

성공적으로 submodule이 생성되면 지정한 이름으로 된 디렉토리와 .gitmodules파일이 생깁니다.

디렉토리에는 경로로 지정한 프로젝트와 .git파일이 들어갑니다.

프로젝트를 pull 해서 .gitmodule이 변경되면 submodule 을 update해야합니다.

프로젝트를 pull 해도 submodule은 pull 되지 않기 때문입니다.

$ git submodule update

프로젝트를 처음 pull받은 쪽의 경우 submodule 폴더는 텅 비어있습니다.

submodule을 내려 받으려면 다음 명령어를 치면 됩니다.

$ git submodule init

$ git submodule update

명령어 정리

git init: 깃 저장소를 초기화한다. 저장소나 디렉토리 안에서 이 명령을 실행하기 전까지는 그냥 일반 폴더이다.
이것을 입력한 후에야 추가적인 깃 명령어들을 줄 수 있다.
git config: “configure”의 준말, 처음에 깃을 설정할 때 가장 유용하다.
git help: 명령어를 잊어버렸다? 커맨드 라인에 이걸 타이핑하면 21개의 가장 많이 사용하는 깃 명령어들이 나타난다.
좀 더 자세하게 “git help init”이나 다른 용어를 타이핑하여 특정 깃 명령어를 사용하고 설정하는 법을 이해할 수도 있다.
git status: 저장소 상태를 체크. 어떤 화일이 저장소 안에 있는지, 커밋이 필요한 변경사항이 있는지,
현재 저장소의 어떤 브랜치에서 작업하고 있는지 등을 볼 수 있다.
git add: 이 명령이 저장소에 새 화일들을 추가하진 않는다. 대신, 깃이 새 화일들을 지켜보게 한다. 화일을 추가하면,
깃의 저장소 “스냅샷”에 포함된다.
git commit: 깃의 가장 중요한 명령어. 어떤 변경사항이라도 만든 후, 저장소의 “스냅샷”을 찍기 위해 이것을 입력한다.
보통 “git commit -m “Message hear.” 형식으로 사용한다. -m은 명령어의 그 다음 부분을 메시지로 읽어야 한다는 것을 말한다.
git branch: 여러 협업자와 작업하고 자신만의 변경을 원한다? 이 명령어는 새로운 브랜치를 만들고,
자신만의 변경사항과 화일 추가 등의 커밋 타임라인을 만든다. 당신의 제목이 명령어 다음에 온다. 새 브랜치를 “cats”로 부르고 싶으면,
git branch cats를 타이핑한다.
git checkout: 글자 그대로, 현재 위치하고 있지 않은 저장소를 “체크아웃”할 수 있다.
이것은 체크하길 원하는 저장소로 옮겨가게 해주는 탐색 명령이다. master 브랜치를 들여다 보고 싶으면, git checkout master를 사용할 수 있고,
git checkout cats로 또 다른 브랜치를 들여다 볼 수 있다.
git merge: 브랜치에서 작업을 끝내고, 모든 협업자가 볼 수 있는 master 브랜치로 병합할 수 있다.
git merge cats는 “cats” 브랜치에서 만든 모든 변경사항을 master로 추가한다.
git push: 로컬 컴퓨터에서 작업하고 당신의 커밋을 깃허브에서 온라인으로도 볼 수 있기를 원한다면, 이 명령어로 깃허브에 변경사항을 “push”한다.
git pull: 로컬 컴퓨터에서 작업할 때, 작업하고 있는 저장소의 최신 버전을 원하면, 이 명령어로 깃허브로부터 변경사항을 다운로드한다(“pull”).

참고할만한 사이트

- https://git-scm.com/book/ko/v1/%EC%8B%9C%EC%9E%91%ED%95%98%EA%B8%B0
- https://backlogtool.com/git-guide/kr/intro/intro1_1.html
- http://egloos.zum.com/riniblog/v/1024993
- http://tuwlab.com/ece/22202