CHAPTER 06

브랜치

6장 브랜치

6장 브랜치

6.1 새로운 작업

1. 새로운 작업

➢ 새로운 작업

- 브랜치(branch):

  나뭇가지, 지사, 분점 등 줄기 하나에서 뻗어 나온 갈림길을 의미함

- 큰 나무 줄기에서 작은 줄기가 뻗어 나오는 것처럼 저장 공간 하나에서 가상의 또 다른 저장 공간을 만드는 것이라고 생각하면 됨

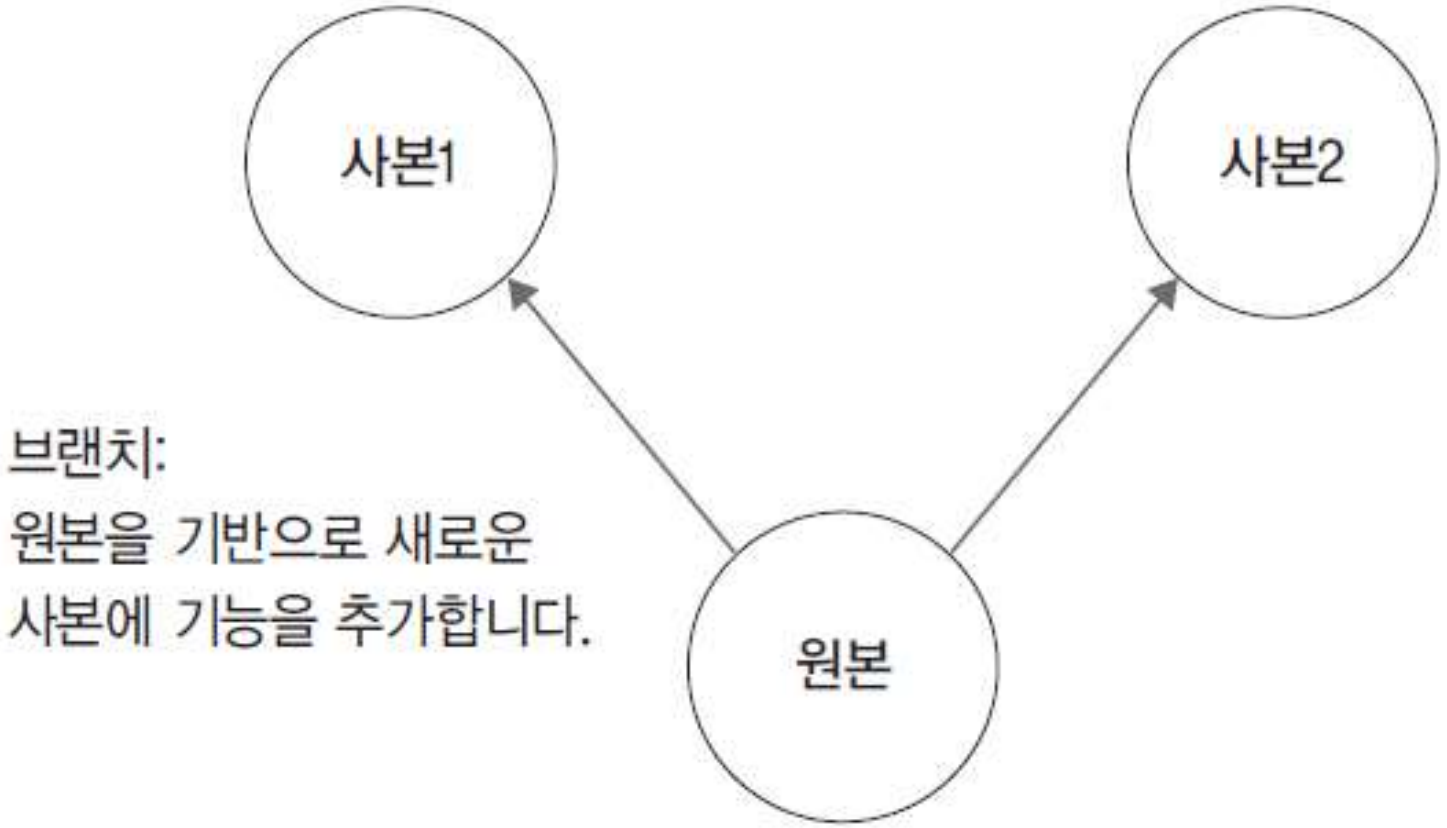
1. 새로운 작업

➢ 브랜치 작업

- 커밋은 파일의 수정 이력을 관리하는 데 사용한다면, 브랜치는 프로젝트를 독립적으로 관리하는데 사용함
- 개발자는 항상 안정된 코드 상태를 유지하고, 개발 중인 작업과 구분하여 관리해야 함
- 잦은 버그 수정과 새로운 기능을 구현할 때마다 작업 폴더를 복사하는 것은 프로젝트를 유지 관리하는 측면에서 좋지 않음
- 많은 프로젝트 폴더를 복제하면 향후 코드를 통합하기 어려움

1. 새로운 작업

▼ 그림 6-1 브랜치 분기



1. 새로운 작업

➢ 깃 브랜치 특징

- 깃 브랜치는 기존 폴더를 복제하는 것과 다르게 가상 폴더를 사용하여 개발 작업을 구분함
- 브랜치는 다음 특징들이 있음

가상 폴더

- 깃의 브랜치는 작업 폴더를 실제로 복사하지 않고, **가상 폴더**로 생성함
- 외부적으로는 물리적인 파일 하나만 있는 것으로 보임
- 생성된 작업 폴더는 물리적으로 복제된 구조보다 유연하게 처리할 수 있음
- 브랜치로 생성된 가상 폴더는 빠르게 공간 이동이 가능함
- 개발자는 쉽게 가상 폴더인 브랜치를 이동하면서 프로젝트를 수행할 수 있음

1. 새로운 작업

➢ 깃 브랜치 특징

독립적인 동작

- 브랜치를 이용하면 원본 폴더와 분리하여 독립적으로 개발 작업을 수행할 수 있음
- 기존에는 소스 코드의 작업 폴더를 별도로 생성함
- 물리적으로 복사된 각자의 폴더에서 코드를 작업한 후 소스 코드 2개를 다시 하나로 합쳐야 했음
- 코드를 하나로 합치려면 작업 내역들을 일일이 찾아 정리해야 함
- 소스 코드를 하나로 통합하는 것은 매우 힘든 작업임
- 깃과 같은 버전 관리 시스템을 이용하면 분리된 코드를 좀 더 쉽게 병합할 수 있음
- 분리된 브랜치에서 소스 코드를 각자 수정한 후 원본 코드에 병합하는 명령만 실행하면 됨
- 깃의 브랜치는 규모가 큰 코드 수정이나 병합을 처리할 때 매우 유용함

1. 새로운 작업

➢ 깃 브랜치 특징

빠른 동작

- 다양한 버전 관리 도구도 브랜치 기능을 지원함
- 보통 다른 VCS들은 브랜치를 생성할 때 내부 파일 전체를 복사함
- 파일 크기가 매우 크다면 브랜치를 생성하는 데 시간이 오래 걸림
- 깃의 브랜치 기능은 다른 버전 관리 도구보다 가볍고, 브랜치 전환이 빠른 것이 특징임
- **깃은 Blob 개념을 도입하여 내부를 구조화**함
- Blob은 포인트와 유사한 객체임
- 깃은 브랜치를 변경할 때 포인터를 이동하여 빠르게 전환함
- 브랜치 명령을 사용하면 내부적으로 **커밋을 하나 생성하여 브랜치로 할당**함
- 다른 버전 관리 시스템은 폴더의 파일 전체를 복사하는 반면, 깃은 41바이트를 가지는 해시(SHA1) 파일 하나만 만들면 됨
- 브랜치를 더 빠르게 생성할 수 있음

6.2 실습 준비

2. 실습 준비

➢ 실습 준비

- 깃은 기본적으로 **master** 브랜치를 하나 가지고 있음
- 브랜치는 **HEAD 포인터**를 가지고 있음
- 이 장의 실습 예제는 깃허브 저장소에 공개되어 있음
- 실습 흐름을 확인할 때 참고함

• https://github.com/jinygit/gitstudy06

2. 실습 준비

➢ 저장소 생성 및 초기화

- 브랜치 실습을 위한 환경을 구축함

```
$ cd 메인폴더
$ mkdir gitstudy06 -------- 새 폴더 만들기
$ cd gitstudy06

infoh@DESKTOP MINGW64 /e/gitstudy06
$ git init -------- 저장소 초기화
Initialized empty Git repository in E:/gitstudy06/.git/
```

2. 실습 준비

➢ 저장소 생성 및 초기화

- 깃 배시에서 초기화 명령어를 실행함
- 저장소가 초기화되면 다음과 같이 터미널 프롬프트 창에 현재 브랜치 이름이 같이 출력됨

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
```

- **현재 브랜치가 master라는 것을 확인**할 수 있음
- 깃 배시는 리눅스 명령을 사용할 수 있고, 현재 **브랜치의 작업 위치**도 쉽게 알 수 있음

2. 실습 준비

➢ 기본 브랜치

- 모든 커밋과 이력은 브랜치에 기록됨
- 깃은 최소한 **브랜치가 1개 이상** 필요함
- 저장소를 처음 초기화하면 **master 브랜치 하나가 자동으로 생성**됨
- 첫 번째 커밋은 master 브랜치에서 시작함
- 초기화한 후에 status 명령어를 실행해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git status
On branch master -------- 브랜치 작업 위치
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

2. 실습 준비

➢ 기본 브랜치

- **branch 명령어**로 현재 브랜치를 확인할 수 있음

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git branch -------- 브랜치 목록
* master
```

- branch 명령어는 생성된 모든 브랜치를 출력함
- 깃에서 기본적으로 선택되는 브랜치는 master임
- 꼭 기본값인 master 이름을 그대로 사용할 필요는 없음
- 통상적으로 깃이 master 브랜치를 자동으로 생성하기 때문에 이를 그대로 많이 사용할 뿐임

6.3 브랜치 생성

3. 브랜치 생성

➢ **브랜치 생성**

- 브랜치는 가상의 작업 폴더임
- 처음 깃을 초기화할 때 워킹 디렉터리는 master 브랜치를 생성함
- 브랜치를 생성하려면 기준이 되는 브랜치 또는 커밋이 하나 있어야 함
- 깃은 master 브랜치를 기준으로 새로운 브랜치를 생성함

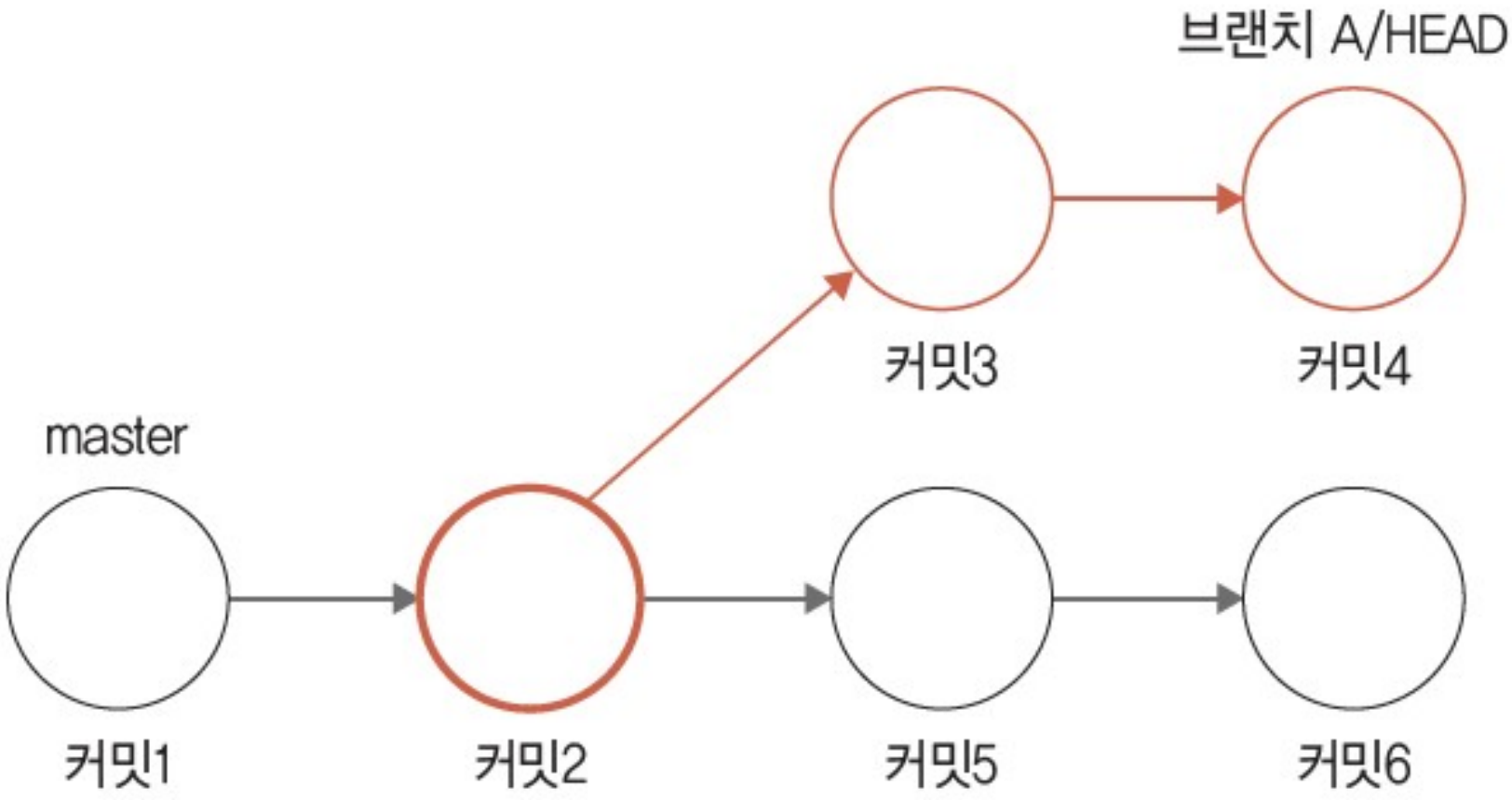
3. 브랜치 생성

➢ **브랜치 생성**

- 브랜치는 **공통된 커밋을 가리키는 지점**
- 브랜치는 **커밋처럼 SHA1 해시키**를 가리킴
- 커밋의 SHA1 해시키는 기억하기가 어렵기 때문에 특정 커밋을 가리키는 **별칭**을 만드는 것임
- 이렇게 만든 별칭이 브랜치임
- 브랜치를 생성한다는 의미는 기존 브랜치 또는 커밋에 새로운 연결 고리를 하나 더 만드는 것과 같음

3. 브랜치 생성

▼ 그림 6-2 브랜치는 공통된 커밋을 가리키는 지점



3. 브랜치 생성

➢ **브랜치 생성**

- **새 브랜치를 생성하면 포인터만 있는 브랜치가 생성**됨
- 일반적으로 브랜치 생성 명령을 실행하면 현재 커밋을 가리키는 HEAD를 기준으로 생성됨
- HEAD는 현재 마지막 커밋을 가리킴
- 새롭게 브랜치가 생성되면 독립된 공간을 할당함
- 기존 작업 영역에는 영향을 주지 않는 새로운 가상 공간임
- 기존 브랜치의 소스 코드에 영향을 주지 않고 새로운 작업을 할 수 있음

3. 브랜치 생성

➢ 브랜치 생성

- 처음 생성되는 기본 master 브랜치 외의 브랜치는 사용자가 직접 branch 명령어를 입력하여 생성해야 함
- 브랜치를 생성한다는 것은 기본적으로 제공되는 master 브랜치 이외에 사용자가 직접 정의한 **사용자 브랜치**를 이야기하는 것임
- 브랜치는 깃에서 또 하나의 **개발 분기점**을 의미함
- 새로운 개발 분기점이 필요할 때는 브랜치를 추가로 생성할 수 있음
- 브랜치 생성 개수에는 제한이 없음
- 필요한 만큼 여러 브랜치를 생성할 수 있으며, 각 브랜치를 구분하려면 브랜치별로 이름을 지정해야 함

3. 브랜치 생성

➢ 브랜치 생성

- 브랜치를 생성할 때는 **branch** 명령어를 사용함

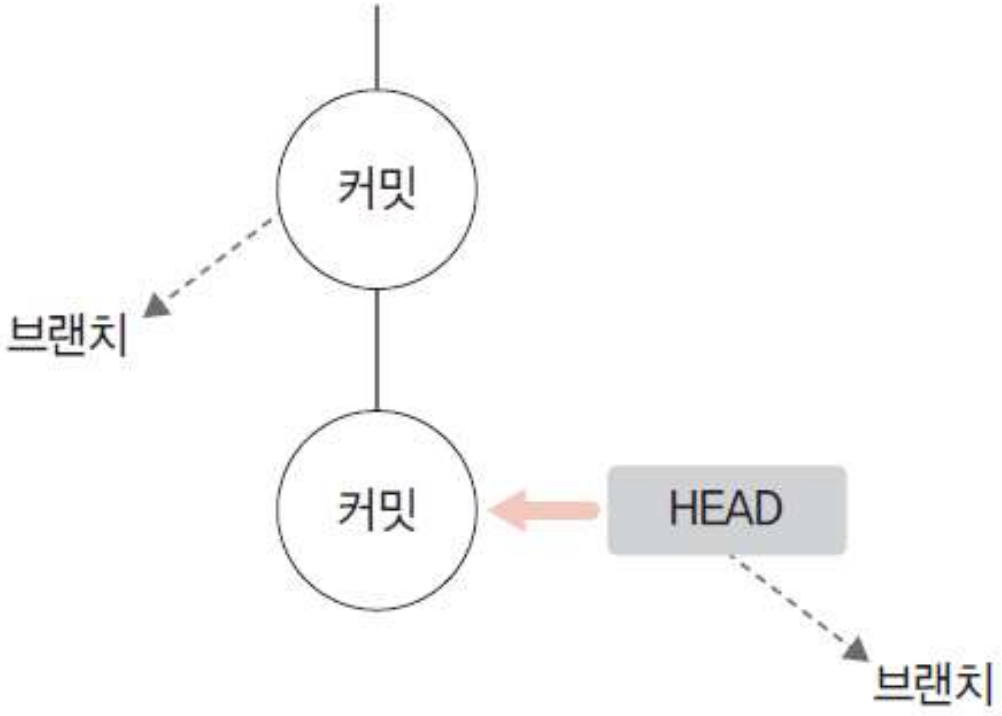
```
$ git branch 브랜치이름 커밋ID
```

3. 브랜치 생성

➢ **브랜치 생성**

- branch 명령어 뒤에 브랜치 이름을 **인자 값**으로 추가함
- 브랜치 이름만 입력하면 현재 **HEAD포인터**를 기준으로 새로운 브랜치를 생성함
- 직접 **커밋 ID** 인자 값을 지정하면, 지정한 커밋 ID를 기준으로 브랜치를 생성함

▼ **그림 6-3 브랜치 생성**



3. 브랜치 생성

➢ 브랜치 생성

- 저장소에 새로운 파일을 하나 생성한 후 저장함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ code branch.htm -------- VS Code로 파일 작성
```

branch.htm

```
<h1>브랜치 실습을 합니다.</h1>
```

3. 브랜치 생성

➢ **브랜치 생성**

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git add branch.htm -------- 추적 등록

infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git commit -m "first" -------- 커밋 작성
[master (root-commit) cc66812] first

 1 file changed, 1 insertion(+)
 create mode 100644 branch.htm
```

3. 브랜치 생성

➢ **브랜치 생성**

- 코드를 저장하고 커밋을 하나 추가함
- 커밋이 있는 상태에서 새로운 브랜치를 생성함
- 마지막 커밋ID(**100644**)를 기준으로 브랜치를 생성 및 추가함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git branch footer -------- 브랜치 생성

infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git branch
  footer -------- 생성된 브랜치
* master -------- 현재의 브랜치
```

- 브랜치를 생성할 때는 결과 메시지를 별도로 표시하지 않음
- 메시지가 없어도 정상적으로 브랜치가 생성된 것임

3. 브랜치 생성

➢ **브랜치 이름**

- 브랜치 이름을 지을 때 사전 예약된 명칭은 따로 없음
- 다만 해당 브랜치 작업은 알기 쉬운 이름으로 짓는 것이 좋음
- 깃 플로(git flow)에서 정의한 브랜치 이름을 적용하는 것도 좋은 방법

3. 브랜치 생성

➢ **브랜치 이름**

- 브랜치 이름은 슬래시(/)를 사용하여 계층적인 구조로 만들어서 사용할 수 있음
- 작성 규칙은 다음과 같음

  - 기호(-)로 시작할 수 없음
  - 마침표(.)로 시작할 수 없음
  - 연속적인 마침표(..)를 포함할 수 없음
  - 빈칸, 공백 문자, 물결(~), 캐럿(^), 물음표(?), 별표(*), 대괄호([ ]) 등은 포함할 수 없음
  - 아스키 제어 문자는 포함할 수 없음

3. 브랜치 생성

➢ **브랜치 이름**

- 주의할 점은 **브랜치 이름은 중복해서 사용하지 않아야 한다는 것**임
- 이미 생성된 브랜치 이름과 동일한 이름으로 생성한다면 오류가 발생함

예

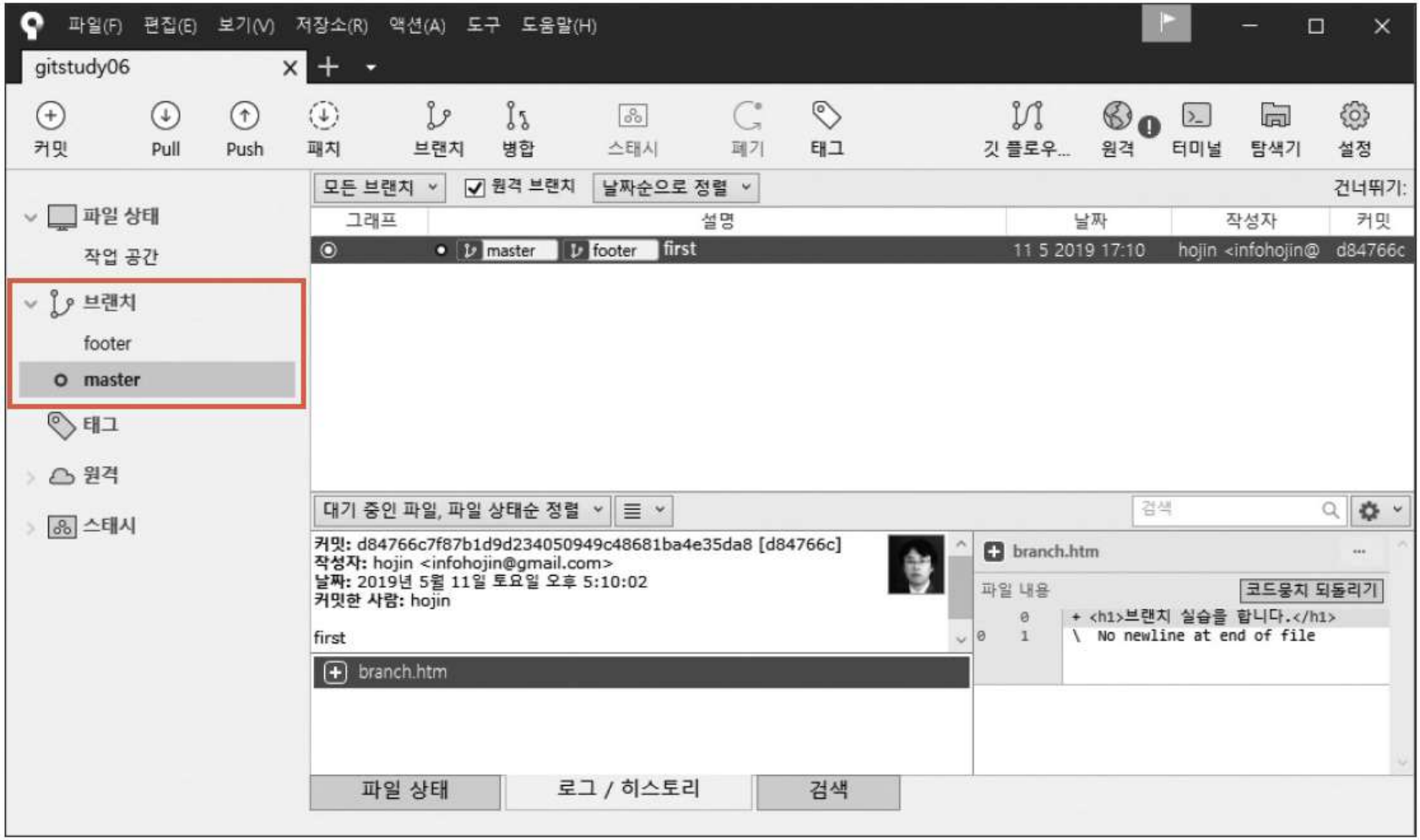
```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git branch footer
fatal: A branch named 'footer' already exists.
```

3. 브랜치 생성

- **소스트리 브랜치**
  - 소스트리의 새 탭에서 **Add** 버튼을 클릭함
  - **탐색**을 눌러 앞에서 만든 gitstudy06 폴더를 찾아 선택한 후 **추가**를 누름
  - gitstudy06 저장소와 연결됨

3. 브랜치 생성

▼ 그림 6-4 소스트리에서 브랜치 확인



3. 브랜치 생성

▼ 그림 6-5 브랜치 메뉴 선택

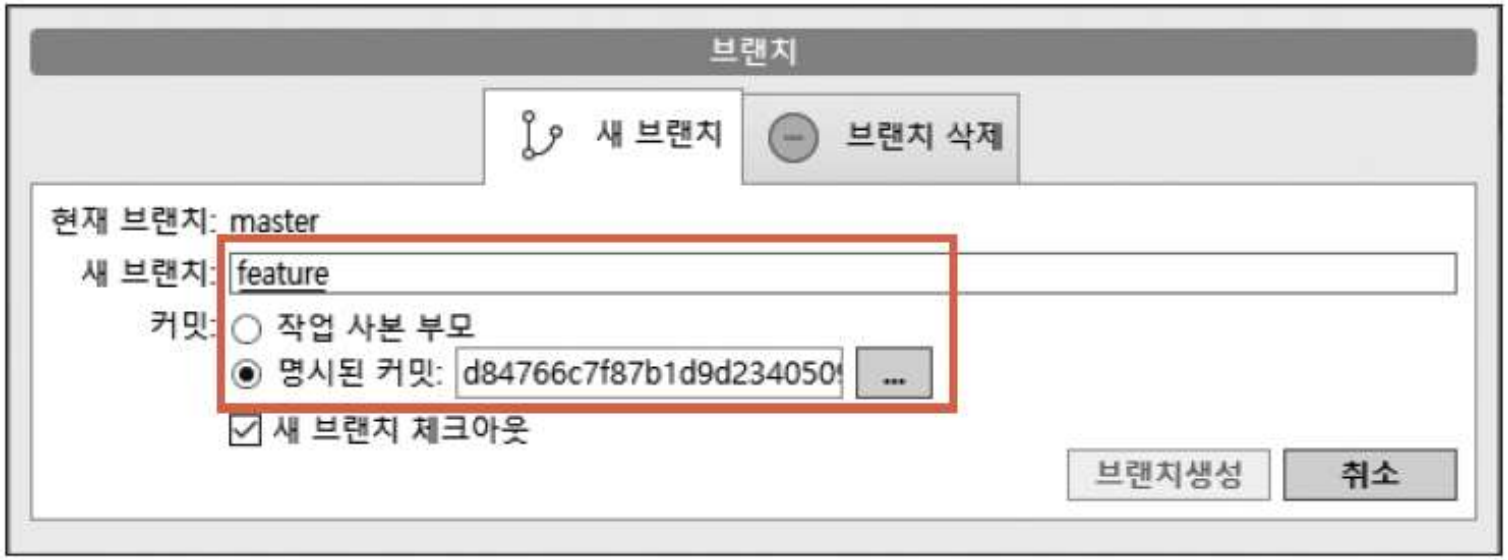
3. 브랜치 생성

➢ **소스트리 브랜치**

- 브랜치 생성 창이 열리면 소스트리에서 **커밋을 선택**하여 브랜치를 생성하는 것은 **지정한 커밋을 기준으로 브랜치를 생성**한다는 의미
- master의 최종 커밋(HEAD)을 기준으로 브랜치를 생성할 때는 **작업 사본 부모** 항목을 선택함

3. 브랜치 생성

▼ 그림 6-6 feature 브랜치 생성



3. 브랜치 생성

▼ 그림 6-7 추가한 브랜치 확인

6.4 브랜치 확인

4. 브랜치 확인

➢ **간단 브랜치 목록**

- branch 명령어만 입력하면 됨
- **branch 명령어는 단독으로도 실행할 수 있음**

```
$ git branch
```

4. 브랜치 확인

➢ 간단 브랜치 목록

- git branch 명령어를 실행하면 현재 모든 브랜치가 나열됨
- * 표시가 된 브랜치로 자동 이동함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ git branch -------- 브랜치 목록
* feature -------- 현재의 브랜치(소스트리를 실습할 때 변경된 브랜치 위치)
  footer
  master
```

4. 브랜치 확인

➢ **간단 브랜치 목록**

- 생성된 전체 브랜치 목록을 출력함
- 브랜치 이름 앞에는 별표(*)가 붙은 것을 확인할 수 있음
- **별표(*)는 현재 선택된 브랜치를 의미함**
- 소스트리에서 feature 브랜치를 생성함
- 소스트리 목록에서 ○ **마크가 이동된 것을 확인**함
- 깃 배시에서는 ○ 마크 대신 별표(*)로 표시됨

4. 브랜치 확인

➢ **브랜치 해시**

- 브랜치는 특정한 **커밋의 해시 값(SHA1)**을 가리킴
- 깃의 저수준 명령어인 rev-parse를 사용하면 현재 브랜치가 어떤 커밋 해시 값(SHA1)을 가리키는지 확인할 수 있음

```
$ git rev-parse 브랜치이름
```

4. 브랜치 확인

➢ 브랜치 해시

- 실습으로 브랜치와 커밋 간 관계를 확인해보자
- 먼저 커밋 로그를 확인함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ git log -------- 로그 확인
commit d84766c7f87b1d9d234050949c48681ba4e35da8 (HEAD -> feature, master, footer)
Author: hojin <infohojin@gmail.com>
Date:   Sat May 11 17:10:02 2019 +0900
    first
```

4. 브랜치 확인

➢ **브랜치 해시**

- footer 브랜치의 커밋 해시(SHA1)를 확인함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ git rev-parse footer
d84766c7f87b1d9d234050949c48681ba4e35da8
```

- **브랜치의 해시 값과 브랜치를 생성한 기준 커밋의 해시 값이 동일**함
- 브랜치가 커밋의 해시를 기준으로 생성된다는 것을 다시 한 번 알 수 있음

4. 브랜치 확인

➢ 브랜치 세부 사항 확인

- 기본적인 branch 명령어는 간단한 브랜치 이름만 출력함
- 옵션을 사용하면 좀 더 상세한 브랜치 정보를 얻을 수 있음
- branch 명령어 뒤에 **-v 또는 -verbose 옵션을 함께 사용하면 브랜치 이름, 커밋 ID, 커밋 메시지**를 같이 볼 수 있음

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ git branch -v -------- 브랜치 세부 사항 확인
* feature  d84766c first
  footer   d84766c first
  master   d84766c first
```

6.5 브랜치 이동

5. 브랜치 이동

➢ **체크아웃**

- **현재 브랜치를 떠나** 새로운 브랜치로 들어간다는 의미임
- 깃에서 브랜치 간 이동할때는 **checkout 명령어**를 사용함

```
$ git checkout 브랜치이름
```

- checkout 명령어로 브랜치 간 이동하면서 실습해보자
- 주의할 점은 깃은 **하나의 워킹 디렉터리만 가지고 있다는 것**
- 워킹 디렉터리는 선택한 브랜치 하나만 연결되어 있음
- 한 브랜치에서만 작업과 커밋을 할 수 있음
- 다른 브랜치에서 작업하려면 반드시 **브랜치를 변경하여 워킹 디렉터리를 재설정**해야 함

5. 브랜치 이동

➢ 체크아웃

- checkout 명령어를 사용하여 footer 브랜치로 변경해보자

```
infoh@DESKTOP-MINGW64 /e/gitstudy06 (feature)
$ git checkout footer -------- 브랜치 이동
Switched to branch 'footer'

infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
```

변경됨

- 깃의 체크아웃은 거의 순간적으로 실행됨
- 깃은 빠르게 포인터를 이용하여 빠르게 브랜치를 이동할 수 있는 것이 장점

5. 브랜치 이동

➢ **체크아웃**

- 다시 브랜치 목록을 확인함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
$ git branch -v -------- 브랜치 목록
  feature  d84766c first
* footer   d84766c first
  master   d84766c first
```

- 별표(*)가 footer 브랜치 이름 앞으로 변경된 것을 확인할 수 있음

5. 브랜치 이동

➢ 브랜치 동작 원리

- checkout 명령어로 브랜치가 변경되면 깃은 내부적으로 몇 가지 동작을 수행함

  - HEAD 정보는 항상 변경된 브랜치의 마지막 커밋을 가리킴
    이처럼 HEAD가 브랜치의 마지막 커밋을 의미하기 때문에 **브랜치가 이동하면 HEAD**
  - **포인터도 함께 이동**함
    변경된 브랜치로 새로운 작업을 할 수 있도록 워킹 디렉터리를 변경함
    브랜치를 변경하려면 기존 브랜치의 **워킹 디렉터리를 정리해야 함**
    기존 브랜치의 워킹 디렉터리를 정리하지 않고서는 브랜치를 변경할 수 없음

5. 브랜치 이동

➢ 소스트리

- 소스트리에서 브랜치를 변경하려면 소스트리의 왼쪽에서 변경하고 싶은 브랜치를 선택함

▼ 그림 6-8 master 브랜치로 체크아웃

5. 브랜치 이동

▼ 그림 6-9 master 브랜치 선택

5. 브랜치 이동

➢ 이전 브랜치

- 브랜치를 이동하려면 브랜치 이름을 적어 주어야 함
- 새로운 브랜치가 아닌 이전 브랜치로 좀 더 편하게 이동할 수 있는 방법이 있음
- 대시(-)를 사용하는 방법임
- 리눅스에서 대시(-) 기호는 이전 디렉터리를 의미함
- 이 대시를 사용하여 쉽게 이전 브랜치로 이동할 수 있음

예

```
$ git checkout -
```

5. 브랜치 이동

➢ 이전 브랜치

- 깃 배시에서 git branch -v 명령어를 다시 실행하면 master 브랜치로 변경된 것을 확인할 수 있음

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git branch -v -------- 브랜치 목록
  feature  d84766c first
  footer   d84766c first
* master   d84766c first -------- 현재의 브랜치
```

5. 브랜치 이동

➢ 이전 브랜치

- master 브랜치에서 이전 footer 브랜치로 돌아감

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git checkout - -------- 이전 브랜치로 이동
Switched to branch 'footer'

infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
```

이전 브랜치로 되돌아감

- 이전 브랜치인 footer 브랜치로 되돌아간 것을 확인할 수 있음

5. 브랜치 이동

➢ **워킹 디렉터리 정리**

- 체크아웃을 사용하여 브랜치를 이동할 때는 주의 사항이 있음
- 현재 작업하고 있는 워킹 디렉터리를 **정리**하고 넘어가야 함
- 브랜치 동작 원리에서 브랜치가 변경되면 워킹 디렉터리도 같이 변환된다고 했음
- 워킹 디렉터리 안에서 작성하던 내용이 있고, 커밋을 하지 않았다면 체크아웃할 때 경고가 발생함

5. 브랜치 이동

➢ 워킹 디렉터리 정리

- master 브랜치에서 코드를 수정해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
$ git checkout master -------- 브랜치 이동
Switched to branch 'master'

infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ code branch.htm -------- VS Code 실행, 코드 수정
```

branch.htm

```
<h1>브랜치 실습을 합니다.</h1>
<h2>마스터 워킹 디렉터리 작업 중</h2>
```

5. 브랜치 이동

➢ 워킹 디렉터리 정리

- 저장은 하지만 **커밋은 하지 않음**
- status 명령어로 확인해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git status -------- 상태 확인
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   branch.htm -------- 워킹 디렉터리 수정 상태
no changes added to commit (use "git add" and/or "git commit -a")
```

5. 브랜치 이동

➢ 워킹 디렉터리 정리

- 현재의 상태에서 브랜치를 변경해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git checkout footer -------- 브랜치 이동
Switched to branch 'footer'
M       branch.htm -------- 워킹 디렉터리의 수정 상태
```

5. 브랜치 이동

➢ **워킹 디렉터리 정리**

- 워킹 디렉터리에서 작업하다 커밋하지 않고 남겨 둔 상태에서 다른 브랜치로 체크아웃하면 이처럼 **브랜치 이동이 제한**됨
- 깃은 향후 **충돌을 방지하려고 워킹 디렉터리에 작업이 남아 있다면 경고 메시지를 보여 주고 브랜치를 변경할 수 없게 제한**함
- footer 브랜치는 master 브랜치를 기준으로 생성한 후 별도로 추가 작업을 하지 않았기 때문에 브랜치 생성 당시 master의 dcdb1c1 커밋을 가리킴
- 아직 브랜치 2개가 가리키는 커밋 위치는 같음

5. 브랜치 이동

➢ 워킹 디렉터리 정리

- 브랜치 간에 정상적으로 이동하려면 남아 있는 작업들을 정리해 주어야 함
- 이전으로 돌아가 수정된 내용을 커밋함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
$ git checkout - -------- 브랜치 이동
Switched to branch 'master'
M       branch.htm

infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git commit -am "master working..." -------- 커밋, 워킹 디렉터리 정리
[master 9ca05fb] master working...
 1 file changed, 2 insertions(+), 1 deletion(-)

infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git checkout footer -------- 브랜치 이동
Switched to branch 'footer'
```

6.6 브랜치 공간

6. 브랜치 공간

➢ 브랜치 로그

- 로그를 출력할 때 브랜치 흐름도 같이 보려면 --graph 옵션을 함께 사용함
- **--graph --all 옵션**을 사용하면 모든 로그를 출력함

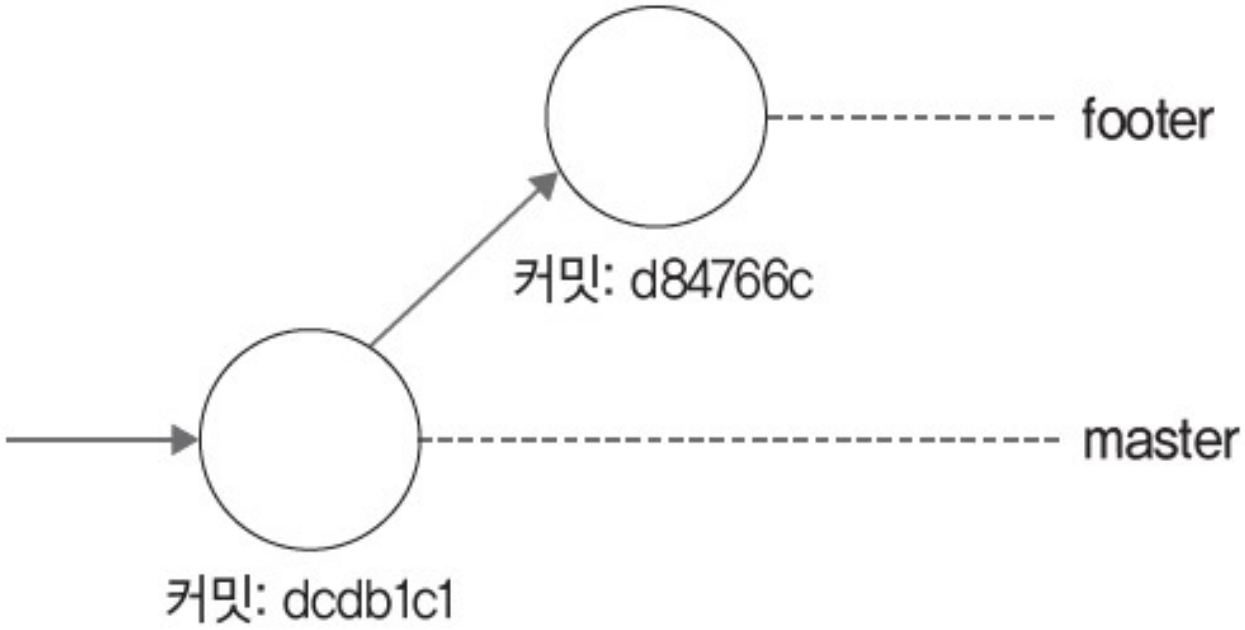
```
infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
$ git log --graph --all
* commit dcdb1c1fa4ef78bedd8dc13bc267e99391cc9782 (master)
| Author: hojin <infohojin@gmail.com>
| Date:   Sat May 11 18:45:35 2019 +0900
|     master working...
|
* commit d84766c7f87b1d9d234050949c48681ba4e35da8 (HEAD -> footer, feature)
  Author: hojin <infohojin@gmail.com>
  Date:   Sat May 11 17:10:02 2019 +0900
      first
```

6. 브랜치 공간

➢ **브랜치 로그**

- 지금까지 로그 내역을 참고하여 커밋 과정을 나타내면 다음과 같음

▼ 그림 6-10 지금까지의 브랜치 작업



- 로그 출력 왼쪽 부분에 브랜치 경로와 작업들이 **텍스트 그래프로 같이 출력**됨

6. 브랜치 공간

➢ 브랜치 소스 확인

- master 브랜치는 dcdb1c1 커밋을 가리킴
- footer 브랜치의 소스 코드 내용을 확인해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
$ cat branch.htm -------- footer의 내용
<h1>브랜치 실습을 합니다.</h1>
```

6. 브랜치 공간

➢ 브랜치 소스 확인

- 이번에는 master 브랜치로 체크아웃하여 소스 코드를 확인해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
$ git checkout master -------- 브랜치 이동
Switched to branch 'master'

infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ cat branch.htm -------- master의 내용
<h1>브랜치 실습을 합니다.</h1>
<h2>마스터 워킹 디렉터리 작업 중</h2>
```

6. 브랜치 공간

- **브랜치 소스 확인**
  - **두 소스 코드에 차이가 있음**
  - 브랜치를 이동하면 변경된 각자 브랜치의 마지막 워킹 디렉터리 상태로 빠르게 변경됨
  - footer는 아직 한 줄짜리 정보를 워킹 디렉터리에 가지고 있는 상태임
  - master는 두 줄짜리 정보를 워킹 디렉터리에 가지고 있는 상태임

6.7 HEAD 포인터

7. HEAD 포인터

- **HEAD 포인터**
  - 깃은 객체의 포인터 개념을 사용함
  - 대표적인 객체 포인터는 HEAD임

7. HEAD 포인터

➢ **마지막 커밋**

- 깃은 마지막 커밋 정보를 기반으로 새로운 커밋을 생성함
- 마지막 커밋은 새로운 커밋의 부모 커밋임
- 시스템이 매번 커밋할 때마다 마지막 커밋 정보를 찾으면 부하가 발생함
- 깃은 마지막 커밋을 쉽게 확인할 수 있도록 특수한 포인터를 제공함
- HEAD는 작업 중인 브랜치의 마지막 커밋 ID를 가리키는 참조 포인터임
- 깃은 마지막 커밋을 가리키는 HEAD 포인터를 **부모** 커밋으로 대체하여 사용함
- HEAD 포인터를 사용하여 빠르게 스냅샷을 생성할 수 있음

7. HEAD 포인터

➢ 마지막 커밋

- 커밋 로그를 이용하여 HEAD를 확인해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git checkout footer -------- 브랜치 이동
Switched to branch 'footer'

infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
$ git log --graph --all -------- 로그 확인
* commit dcdb1c1fa4ef78bedd8dc13bc267e99391cc9782 (master)
| Author: hojin <infohojin@gmail.com>
| Date:   Sat May 11 18:45:35 2019 +0900
|     master working...
|
* commit d84766c7f87b1d9d234050949c48681ba4e35da8 (HEAD -> footer, feature) -------- HEAD 위치
  Author: hojin <infohojin@gmail.com>
  Date:   Sat May 11 17:10:02 2019 +0900
      first
```

7. HEAD 포인터

➢ **마지막 커밋**

- master 브랜치의 마지막 커밋은 dcdb1c1이고, footer 브랜치의 마지막 커밋은 d84766c임
- master 브랜치에서 새로운 커밋을 생성할 때 부모 커밋으로 dcdb1c1을 가리키는 HEAD 포인터를 사용함
- footer 브랜치에서 새로운 커밋을 생성할 때는 d84766c를 가리키는 HEAD 포인터를 사용함
- 각 브랜치의 마지막 HEAD 포인터를 사용하여 커밋함
- 현재 HEAD는 footer 브랜치의 d84766c를 가리킴

7. HEAD 포인터

- **브랜치 HEAD**
  - **브랜치를 이동하면 HEAD 포인트도 이동됨**
  - 브랜치가 여러 개면 HEAD 포인트도 여러 개임
  - 각각의 브랜치마다 마지막 커밋이 다르기 때문임
  - 브랜치마다 마지막 커밋 ID를 가리키는 HEAD 포인터가 하나씩 있음

7. HEAD 포인터

➢ 브랜치 HEAD

- master 브랜치로 이동하여 HEAD 위치를 확인해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (footer)
$ git checkout master -------- 브랜치 이동
Switched to branch 'master'

infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git log --graph --all -------- 로그 출력
```

7. HEAD 포인터

➢ 브랜치 HEAD

```
* commit dcdb1c1fa4ef78bedd8dc13bc267e99391cc9782 (HEAD -> master) -------- HEAD 위치
| Author: hojin <infohojin@gmail.com>
| Date:   Sat May 11 18:45:35 2019 +0900
|     master working...
|
* commit d84766c7f87b1d9d234050949c48681ba4e35da8 (footer, feature)
  Author: hojin <infohojin@gmail.com>
  Date:   Sat May 11 17:10:02 2019 +0900
      first
```

7. HEAD 포인터

➢ **브랜치 HEAD**

- master 브랜치로 변경한 후 HEAD 포인터 위치는 이동된 브랜치의 마지막 커밋 dcdb1c1을 가리킴
- HEAD 포인터는 브랜치에 따라서 위치가 달라짐
- HEAD는 현재 작업하는 브랜치를 가리키기 때문임

7. HEAD 포인터

➢ 소스트리 HEAD

- 소스트리에서도 HEAD 포인트 상태를 쉽게 확인할 수 있음
- 각 브랜치의 마지막 위치를 **브랜치 아이콘**으로 표시함
- 예를 들어 각 브랜치마다 마지막 커밋을 HEAD 포인트로 표시함

▼ 그림 6-11 소스트리에서의 HEAD

- HEAD 위치는 각 브랜치의 커밋 개수에 따라 서로 다르게 표시함

7. HEAD 포인터

➢ **상대적 위치**

- 깃의 HEAD 포인터는 내부적으로 커밋을 생성하고 브랜치를 관리하는 데 매우 유용함
- 깃의 다양한 명령어를 입력할 때도 **기준점**으로 사용함
- 마지막 커밋 위치인 HEAD를 기준으로 상대적 커밋 위치도 지정할 수 있음
- 상태적 커밋 위치를 지정할 때는 **캐럿(^)과 물결(~) 기호**를 같이 사용함
- ^과 ~은 HEAD를 기준으로 몇 번째인지 상대적인 위치를 지정함

7. HEAD 포인터

➢ **AHEAD, BHEAD**

- HEAD 앞에 A 또는 B가 붙은 AHEAD와 BHEAD 포인터도 있음
- 원격 저장소와 연동하여 깃을 관리한다면 브랜치마다 HEAD가 2개 있음
- 로컬 저장소 브랜치의 HEAD 포인터와 원격 저장소 브랜치의 HEAD 포인터임
- 원격 저장소와 로컬 저장소는 물리적으로 서로 다른 저장소임
- 두 저장소의 마지막 커밋 위치가 일치하지 않을 수 있음
- 이는 서로 다른 커밋을 가리키는 HEAD 포인터를 가진다는 의미임
- AHEAD와 BHEAD는 서로 다른 저장소 간 HEAD 포인터의 위치 차이를 의미함
- 깃은 항상 원격 저장소의 HEAD와 로컬 저장소의 HEAD를 비교함
- HEAD는 브랜치마다 다름
- 브랜치를 여러 개 운영한다면 다수의 AHEAD와 BHEAD가 생길 수 있음

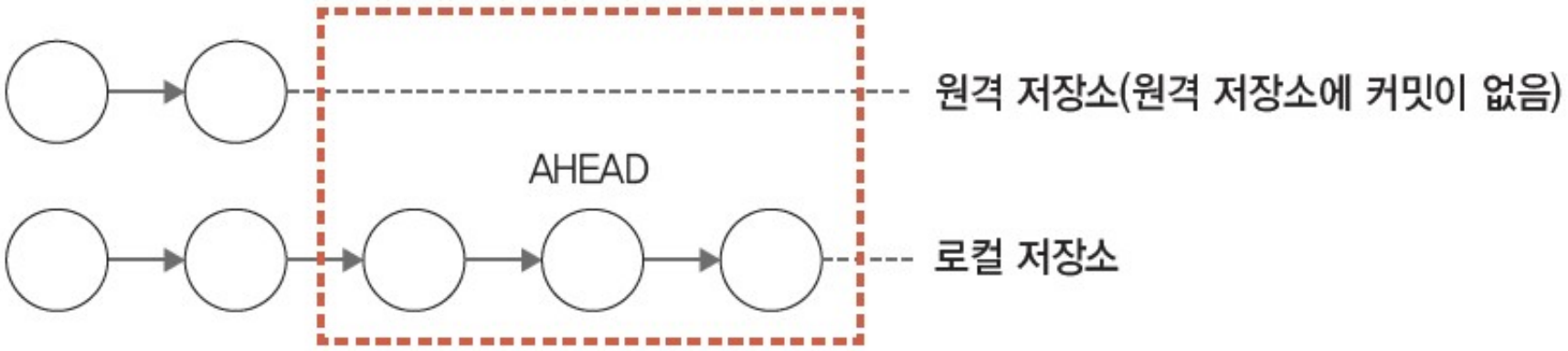
7. HEAD 포인터

➢ AHEAD, BHEAD

AHEAD

- AHEAD는 **서버로 전송되지 않은 로컬 커밋**이 있는 것임
- 예를 들어 로컬 저장소에 새로운 커밋을 생성하고, 새로운 커밋 정보를 서버로 전송하지 않는 상황임
- 이런 경우 AHEAD가 발생함

▼ 그림 6-12 AHEAD



- 로컬 저장소의 HEAD 포인터를 기준으로 **로컬 브랜치에 있는 커밋이 서버의 커밋 개수보다 많은 경우**임

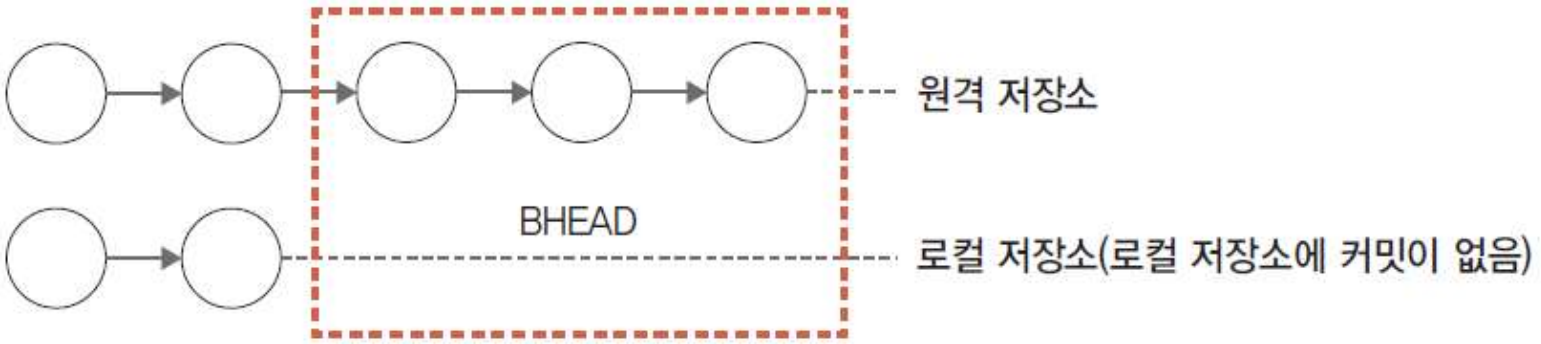
7. HEAD 포인터

➢ **AHEAD, BHEAD**

BHEAD

- BHEAD는 **로컬 저장소로 내려받지 않은 커밋이 있는 것**임
- 예를 들어 누군가 서버에 새로운 커밋을 함
- 아직 로컬 저장소는 서버의 새로운 커밋을 내려받지 않음
- 이런 경우 BHEAD가 발생함

▼ 그림 6-13 BHEAD



- 다른 개발자가 코드를 수정하여 **원격 저장소의 커밋이 자신의 로컬 저장소보다 더 최신 상태**인 것을 의미함

6.8 생성과 이동

8. 생성과 이동

➢ 생성과 이동

- 깃의 브랜치를 생성하는 동작과 이동하는 동작은 별개임
- 브랜치 생성은 branch 명령어를 사용하고, 브랜치 이동은 checkout 명령어를 사용함
- 브랜치를 생성하면서 동시에 생성된 브랜치로 이동하려면 별도의 명령을 실행해야 함

8. 생성과 이동

➢ 자동 이동 옵션

- 브랜치 생성과 이동 명령을 따로 두 번씩 입력하는 것은 불편함
- 깃은 브랜치 생성과 이동 명령을 한 번에 처리하는 **옵션**을 제공함
- 다음과 같이 체크아웃할 때 **-b 옵션**을 같이 사용하면 브랜치 생성과 이동을 한 번에 할 수 있음

```
$ git checkout -b 브랜치이름
```

8. 생성과 이동

➢ 자동 이동 옵션

- 브랜치 생성과 체크아웃을 동시에 하는 실습을 하겠음
- -b 옵션으로 브랜치를 생성하면서 동시에 체크아웃도 함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git checkout -b hotfix -------- 체크아웃
Switched to a new branch 'hotfix' -------- ① 브랜치 생성

infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix) -------- ② 체크아웃
```

8. 생성과 이동

➢ 자동 이동 옵션

- 새로운 브랜치가 생성된 메시지와 함께 hotfix 브랜치로 자동 전환됨
- 브랜치 목록을 살펴보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git branch -v -------- 브랜치 목록
  feature d84766c first
  footer  d84766c first
* hotfix  dcdb1c1 master working... -------- 현재의 브랜치
  master  dcdb1c1 master working... -------- 브랜치 생성만 했기 때문에 같은 커밋 ID를 가리킴
```

- 새로운 hotfix 브랜치가 보이고 앞에 별표(*)가 표시된 것을 확인할 수 있음

8. 생성과 이동

➢ 커밋 이동

- 브랜치는 특정한 커밋에 별명을 부여한 것과 같음
- 일반적으로 브랜치를 생성할 때는 마지막 커밋을 기준으로 함
- 커밋 해시 값을 지정한 별칭으로 브랜치 목록에 등록함
- 이러한 동작 원리로 볼 때 **브랜치 이름은 커밋 해시키와 동일**함
- 브랜치로 이동할 때 꼭 브랜치 이름만 사용할 필요는 없음
- 브랜치 이름 대신 **커밋 해시키**를 사용하여 체크아웃할 수 도 있음

```
$ git checkout 커밋해시키
```

8. 생성과 이동

➢ 커밋 이동

- 커밋 해시키를 사용하여 체크아웃하려면 해시키를 알고 있어야 함
- 커밋 해시 값은 40자리로 매우 길어 입력할 때 오류가 많이 생김
- 해시키를 전체 사용하지 않고, 앞의 7자리 정도만 사용해도 무리가 없음
- 이는 유일한 해시 값이 가지는 특징임
- 7자리만 사용해도 중복될 확률이 적음

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git branch -v -------- 브랜치 목록
  feature d84766c first
  footer  d84766c first
* hotfix  dcdb1c1 master working...
  master  dcdb1c1 master working...
          ------- 이 자리만 표시됨
```

8. 생성과 이동

➢ 커밋 이동

- 브랜치 이름 대신에 직접 master 브랜치의 dcdb1c1 커밋 값을 사용하여 체크아웃해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git checkout dcdb1c1 -------- 커밋으로 브랜치 이동
Note: checking out 'dcdb1c1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:
```

8. 생성과 이동

➢ 커밋 이동

```
git checkout -b <new-branch-name>

HEAD is now at dcdb1c1 master working...

infoh@DESKTOP MINGW64 /e/gitstudy06 ((dcdb1c1...))
```

브랜치 이름이 커밋 ID로 표시됨

8. 생성과 이동

➢ **커밋 이동**

- 지정한 커밋 위치로 체크아웃됨
- 깃 배시의 브랜치 이름에 입력한 커밋 해시 값이 표시됨
- 이처럼 브랜치 중간에 작업한 특정 커밋으로 직접 이동하여 상태를 확인할 수 있음

8. 생성과 이동

➢ HEAD를 활용한 이동

- 커밋의 해시키를 사용하여 체크아웃하려면 복잡한 해시키를 알고 있어야 함
- 복잡한 영어와 숫자로 표현하므로 입력 오류도 많이 생김
- 좀 더 간편하게 HEAD 포인터를 사용하여 체크아웃할 수도 있음
- 예를 들어 바로 이전 커밋으로 체크아웃하고 싶을 때는 다음 명령을 실행함

예

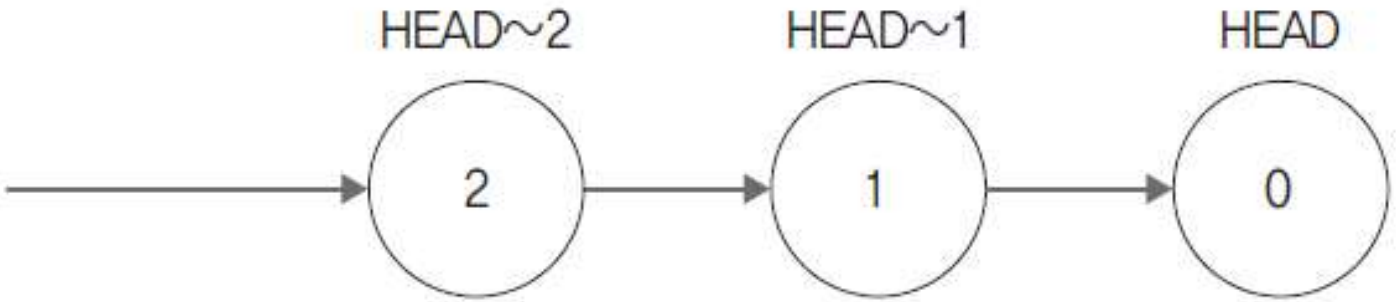
```
$ git checkout HEAD~1 -------- 현재의 한 단계 전
```

8. 생성과 이동

➢ HEAD를 활용한 이동

- 마지막 커밋인 HEAD를 기준으로 1단계의 커밋 상태로 이동함

▼ 그림 6-14 HEAD 기준 이동



- 여러 단계 이전으로 이동하고자 한다면 뒤에 있는 숫자만 바꿈

예

```
$ git checkout HEAD~5
```

8. 생성과 이동

➢ 돌아오기

- 커밋 해시키 또는 HEAD를 사용하여 과거의 커밋으로 체크아웃했다면 다시 현재 시점으로 돌아와야 함
- 간단하게 다시 돌아오는 방법은 대시(-)를 사용하는 것임

예

```
$ git checkout -
```

- 대시(-)를 사용하면 바로 이전의 브랜치로 복귀함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 ((dcdb1c1...))
$ git checkout - -------- 이전 브랜치로 이동
Switched to branch 'hotfix'

infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
```

브랜치 복귀

8. 생성과 이동

➢ 돌아오기

- 커밋 단계를 여러 번 이동한 후 원래 브랜치(현재)로 되돌아오려면 대시(-) 명령어를 여러 번 실행해야 함
- 이때는 그냥 직접 브랜치 이름을 입력하는 것이 편리함

예

```
$ git checkout master -------- 특정 브랜치로 복귀
```

- 이 명령을 실행하면 master 브랜치로 이동함
- 브랜치 이름을 입력하면 브랜치의 마지막 커밋 위치인 HEAD 포인트로 복귀함

6.9 원격 브랜치

9. 원격 브랜치

➢ **원격 브랜치**

- 깃은 다수의 개발자와 협업으로 코드를 유지할 수 있음
- 주요 개발 작업들은 로컬 저장소에서 하지만 협업은 원격 저장소도 공유함
- 로컬 저장소도 하나의 저장소고, 원격 저장소도 하나의 저장소임
- 깃은 분산형 버전 관리로서 다수의 저장소를 만들어 연결할 수 있기 때문임
- 이번에는 브랜치를 이용하여 협업하는 방법을 알아보자

9. 원격 브랜치

➢ 리모트 브랜치

- 저장소는 각자의 고유한 브랜치를 생성하고 관리함
- **리모트 브랜치:**

  원격 저장소에 생성한 브랜치
- 로컬 저장소에 생성한 브랜치는 서버로 공유할 수 있음
- 원격 저장소와 연결된 로컬 저장소에서 새로운 브랜치를 생성한다고 해서 자동으로 원격 저장소에도 브랜치가 생성되는 것은 아님
- 원격 저장소에 등록된 브랜치가 자동으로 로컬 저장소를 만들지도 않음
- 별도 명령을 실행하여 **저장소를 동기화**해야 함

9. 원격 브랜치

➢ **리모트 브랜치**

- 원격 저장소와 로컬 저장소의 브랜치 이름은 보통 같지만, 반드시 일치하지 않아도 괜찮음
- 서로 다른 이름으로 브랜치를 연결할 수도 있음
- 두 저장소는 서로 다른 브랜치로 운영·관리 할 수 있음
- 리모트 브랜치는 보통 **별칭/브랜치 이름** 형태임

9. 원격 브랜치

Git 교과서

➢ 실습 준비

- 실습을 위해 깃허브에 새로운 실습 저장소를 하나 생성함
- 반드시 자신의 계정으로 로그인함

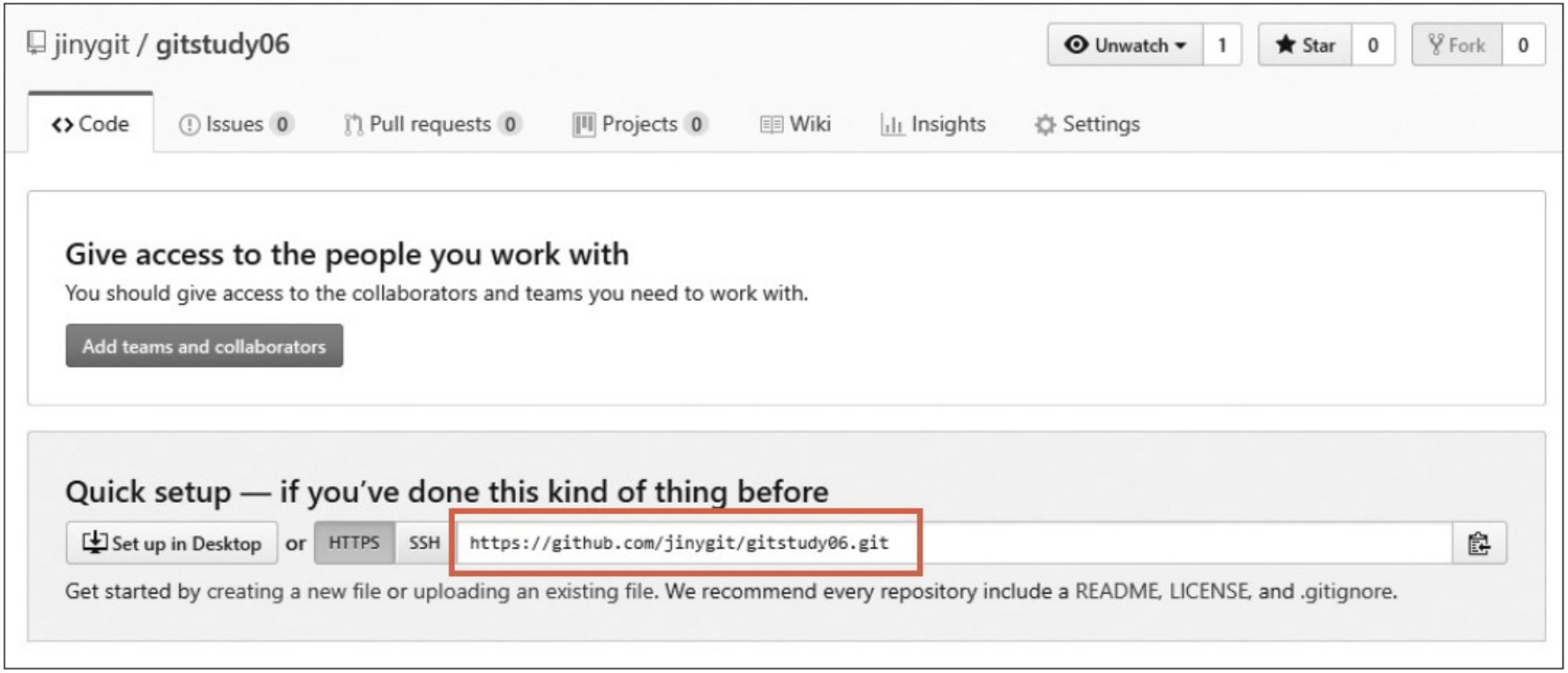
▼ 그림 6-15 깃허브에서 새 저장소 생성

9. 원격 브랜치

➢ 실습 준비

- 모두 입력한 후 Create repository를 누르면 다음 화면이 나오고 저장소 주소를 확인할 수 있음

▼ 그림 6-16 새 저장소의 주소



9. 원격 브랜치

➢ 실습 준비

- 새 저장소를 생성했다면 다음과 같이 실행해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git remote add origin https://github.com/jinygit/gitstudy06.git -------- 원격 저장소 등록
                                                  -------- 자신의 URL 주소

infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git remote -v -------- 원격 저장소 목록
origin  https://github.com/jinygit/gitstudy06.git (fetch)
origin  https://github.com/jinygit/gitstudy06.git (push)
```

9. 원격 브랜치

➢ **브랜치 추적**

- 깃의 브랜치는 특정 커밋 해시 값을 가리키는 포인터임
- 리모트 브랜치 또한 원격 저장소의 브랜치를 가리키는 포인터임
- 브랜치 추적:

  원격 저장소의 브랜치를 가리키는 것

9. 원격 브랜치

➢ 브랜치 추적

- 다른 용어로 추적 브랜치를 트래킹 브랜치라고 함
- 트래킹 브랜치는 원격 브랜치를 가리키는 북마크와 같음
- 추적 브랜치는 원격 브랜치의 마지막 커밋 해시 값을 가리킴
- 이 포인터 정보는 .git/refs 폴더 안에 저장되어 있음

예

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ ls .git/refs/ -------- 원격 브랜치 정보
heads  tags
```

- 로컬 저장소가 원격 저장소와 연결될 때 원격 브랜치의 트래킹 정보는 자동으로 갱신됨
- 로컬 저장소는 마지막으로 연결된 리모트 브랜치의 커밋 해시 값을 항상 가지고 있음

9. 원격 브랜치

➢ 브랜치 업로드

- 저장소의 브랜치 정보는 원격 저장소에 자동으로 등록되지 않음
- 등록된 원격 저장소의 리모트 브랜치는 **remote show 명령어**로 확인할 수 있음

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git remote show origin -------- 원격 브랜치
* remote origin
  Fetch URL: https://github.com/jinygit/gitstudy06.git
  Push  URL: https://github.com/jinygit/gitstudy06.git
  HEAD branch: (unknown)
```

- 현재는 원격 저장소를 등록만 했기 때문에 아직 리모트 브랜치는 없음
- **리모트 브랜치는 서버간에 통신을 하고 나서 생성**됨

9. 원격 브랜치

➢ 브랜치 업로드

- 로컬 저장소의 브랜치를 원격 저장소에 동기화하려면 **푸시** 작업을 해야 함

```
$ git push 원격저장소별칭 브랜치이름
```

9. 원격 브랜치

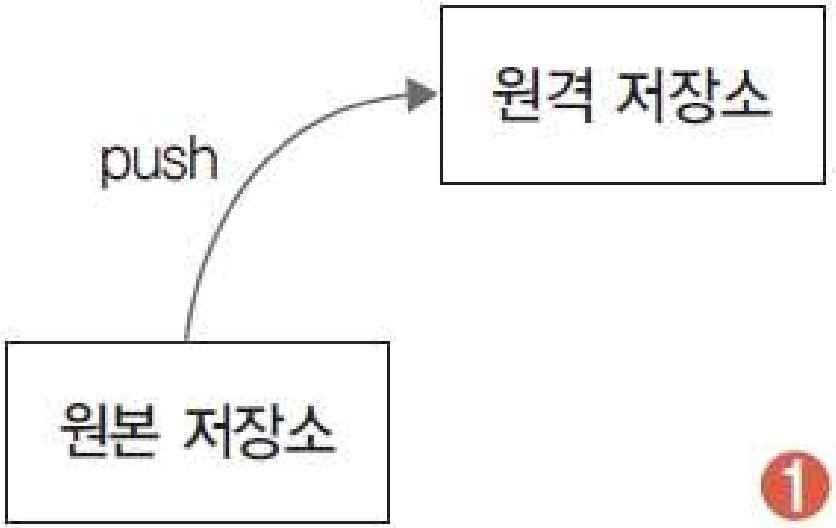
➢ 브랜치 업로드

- 로컬 브랜치를 푸시하면 원격 저장소는 로컬 저장소와 동일한 브랜치를 생성함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git push -u origin master -------- master 브랜치 전송
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 546 bytes | 109.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/jinygit/gitstudy06.git
 * [new branch]      master -> master -------- 리모트 브랜치 생성
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

9. 원격 브랜치

▼ 그림 6-17 원격 저장소 전송

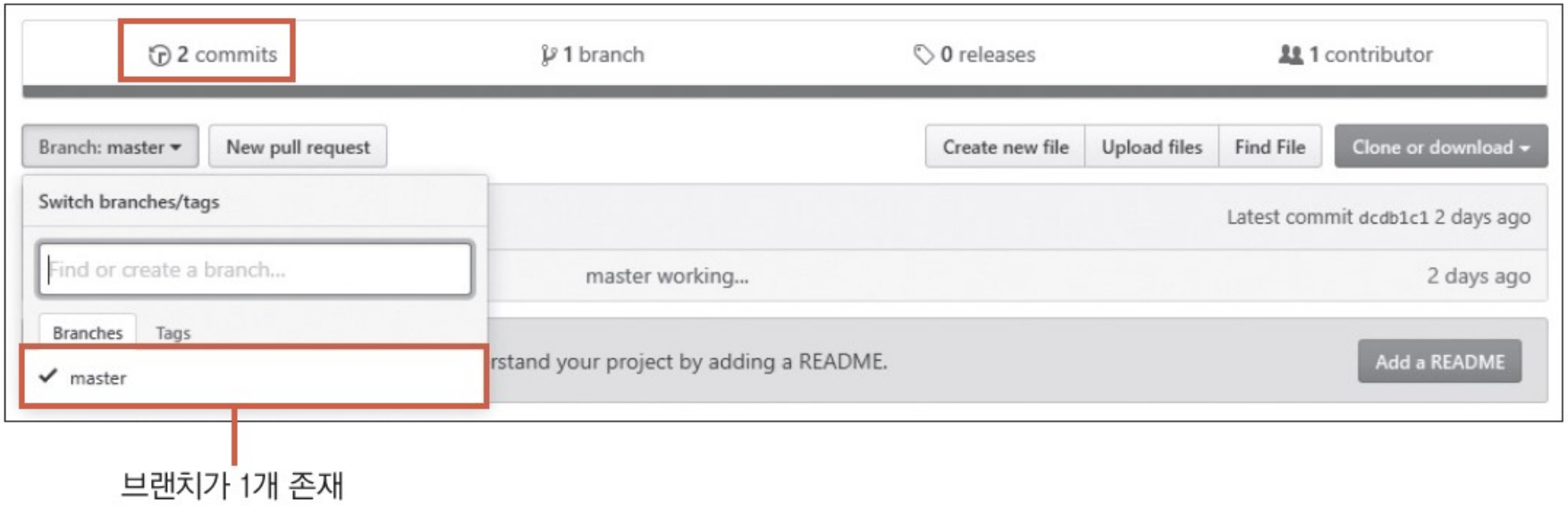


9. 원격 브랜치

➢ 브랜치 업로드

- 깃허브 저장소로 이동해서 확인해보자
- 이전과 달리 새로운 master 브랜치가 생성되고, 관련된 커밋이 등록됨

▼ 그림 6-18 깃허브에서 브랜치 확인



9. 원격 브랜치

➢ 브랜치 업로드

- 그럼 다시 현재 **로컬 저장소의 브랜치**를 확인함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git branch -v -------- 브랜치 목록
  feature d84766c first
  footer  d84766c first
* hotfix  dcdb1c1 master working... -------- 현재의 브랜치
  master  dcdb1c1 master working...
```

- 로컬 저장소에는 브랜치가 4개 있음
- 깃허브에는 master 브랜치만 하나 있음
- 원격 저장소에 리모트 브랜치를 생성하려면 로컬 저장소에서 브랜치 전송 명령을 실행해 주어야 함

9. 원격 브랜치

➢ 브랜치 업로드

- 그럼 hotfix 브랜치도 원격 저장소로 등록해보자

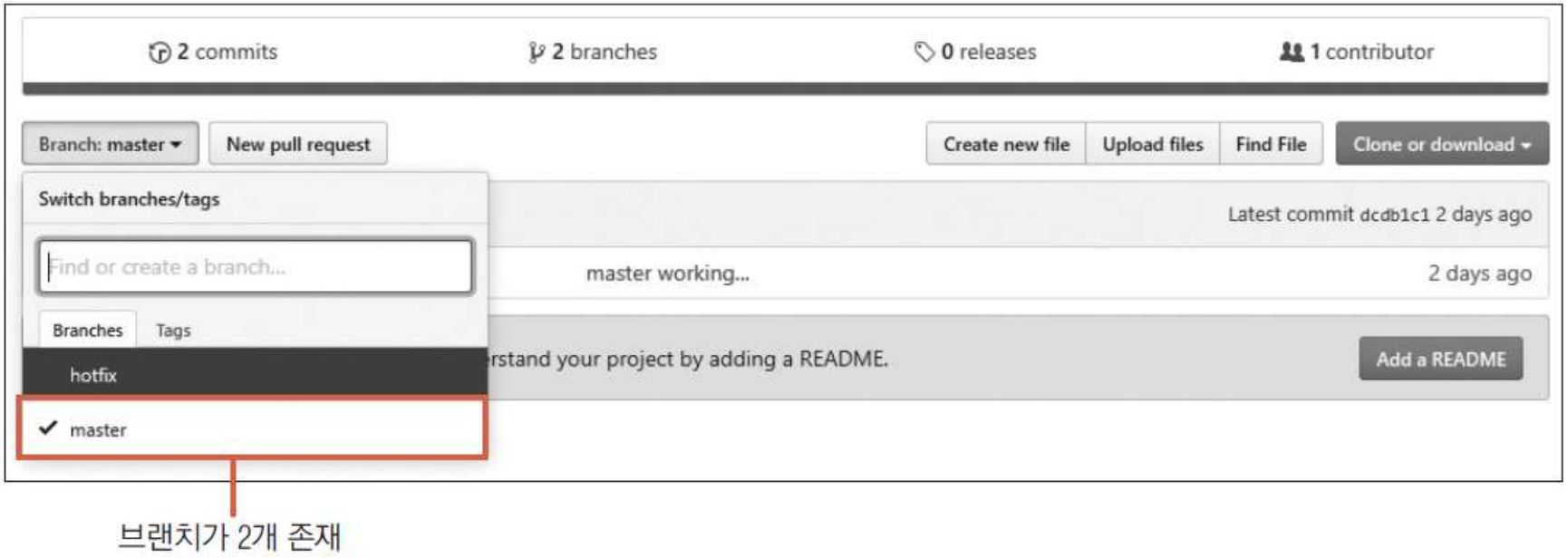
```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git push -u origin hotfix -------- hotfix 브랜치 전송
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'hotfix' on GitHub by visiting:
remote:      https://github.com/jinygit/gitstudy06/pull/new/hotfix
remote:
To https://github.com/jinygit/gitstudy06.git
 * [new branch]      hotfix -> hotfix -------- 리모트 브랜치 생성
Branch 'hotfix' set up to track remote branch 'hotfix' from 'origin'.
```

9. 원격 브랜치

➢ 브랜치 업로드

- 정상적으로 hotfix 브랜치가 푸시됨
- 깃허브를 보면 새로운 hotfix 리모트 브랜치가 추가된 것을 확인할 수 있음

▼ 그림 6-19 푸시된 hotfix 브랜치 확인



9. 원격 브랜치

➢ 이름이 다른 브랜치

- 일반적으로 로컬 저장소의 브랜치 이름과 원격 저장소의 브랜치 이름은 동일하게 사용함
- 반드시 이름이 동일할 필요는 없음
- 가끔은 동일한 브랜치 이름을 사용하기 어려울 때가 있음
- 예를 들어 다른 개발자가 만든 원격 서버의 브랜치를 테스트하려고 할 때 자신의 브랜치 이름과 동일하면 충돌이 생김
- 깃은 서로 다른 로컬 브랜치와 리모트 브랜치를 수동으로 지정하여 연결할 수 있음

```
$ git push origin 브랜치이름:새로운브랜치
```

9. 원격 브랜치

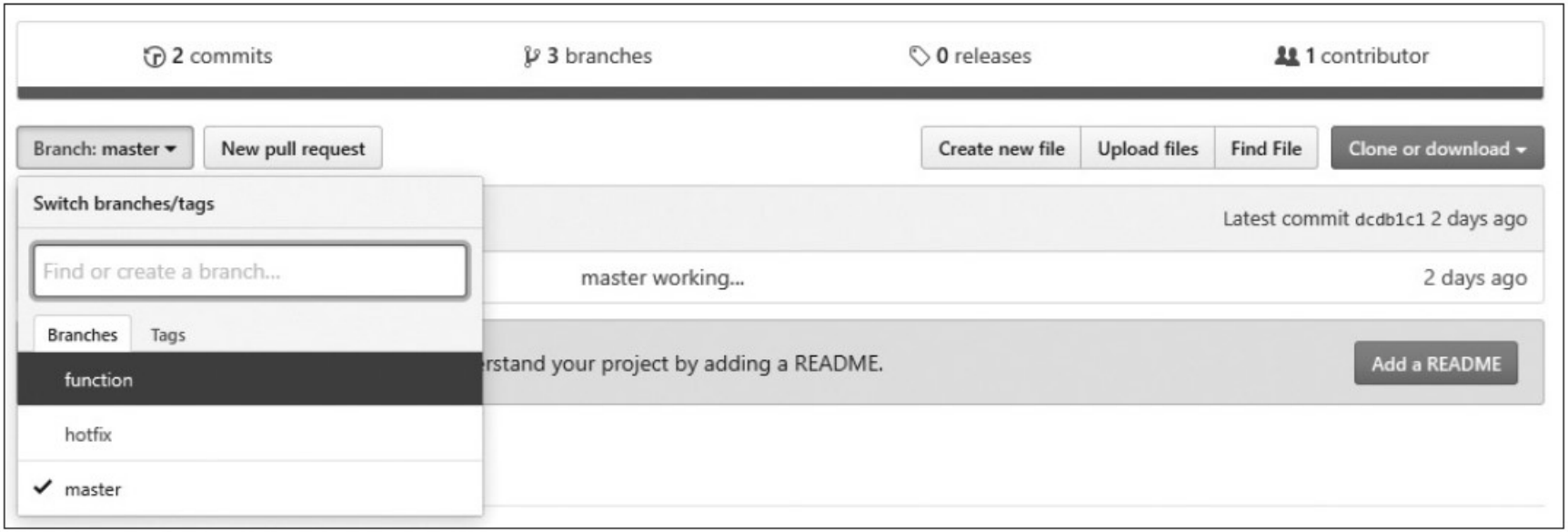
➢ 이름이 다른 브랜치

- 현재 브랜치를 서버(origin)의 새로운 브랜치 이름으로 전송하라는 의미
- 이번에는 로컬 저장소의 feature 브랜치를 원격 저장소의 function 브랜치로 푸시해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git push -u origin feature:function -------- 다른 이름으로 브랜치 전송
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'function' on GitHub by visiting:
remote:      https://github.com/jinygit/gitstudy06/pull/new/function
remote:
To https://github.com/jinygit/gitstudy06.git
 * [new branch]      feature -> function -------- 리모트 브랜치 생성
Branch 'feature' set up to track remote branch 'function' from 'origin'.
```

9. 원격 브랜치

▼ 그림 6-20 깃허브에서 생성한 브랜치 확인



9. 원격 브랜치

➢ 업스트림 트래킹

- 업스트림(upstream)은 브랜치 추적을 다르게 표현한 것
- 리모트 브랜치는 브랜치 이름을 동일하게 생성할 수도 있고, 다른 이름으로 생성할 수도 있음
- **업스트림 트래킹:**

  로컬 저장소의 브랜치와 원격 저장소의 브랜치는 업로드할 수 있도록 매칭

▼ 그림 6-21 업스트림 트래킹

9. 원격 브랜치

➢ 업스트림 트래킹

- 트래킹 브랜치(업스트림)는 **리모트 브랜치와 로컬 브랜치를 연결해 주는 중간 다리 역할**을 함
- clone 명령어로 저장소를 복제할 때 원격 저장소에 등록된 트래킹 브랜치들을 자동으로 함께 설정함

9. 원격 브랜치

➢ 업스트림 트래킹

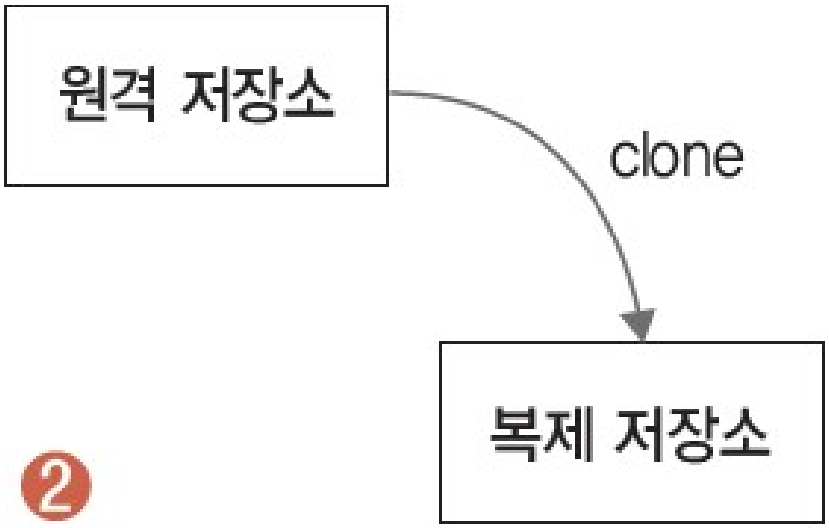
- 새롭게 저장소를 복제해보자
- 상위 메인 폴더로 이동하여 새로운 깃 저장소를 복제함

```
$ cd 메인폴더
$ git clone https://github.com/jinygit/gitstudy06.git gitstudy06_clone -------- 원격 저장소를 복제. 복제한 폴더 이름은 gitstudy06_clone
Cloning into 'gitstudy06_clone'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 6 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.

$ cd gitstudy06_clone -------- 복제된 폴더로 이동
```

9. 원격 브랜치

▼ 그림 6-22 복제 저장소



9. 원격 브랜치

➢ 업스트림 트래킹

- clone 명령어는 원격 저장소의 모든 브랜치 정보를 한 번에 다 가지고 오지 않음
- 다음과 같이 복제된 저장소의 브랜치를 확인해 보면 master 브랜치 하나만 표시됨

```
infoh@DESKTOP MINGW64 /e/gitstudy06_clone (master)
$ git branch -v -------- 브랜치 목록
* master dcdb1c1 master working...
```

- 원격 저장소에는 다수의 리모트 브랜치가 있음
- **-r 옵션**을 사용하면 원격 저장소의 리모트 브랜치 목록을 확인할 수 있음

```
$ git branch -r
```

9. 원격 브랜치

➢ 업스트림 트래킹

- 예를 들어 gitstudy06_clone 저장소와 연결된 원격 저장소의 리모트 브랜치 목록은 다음과 같이 확인할 수 있음

```
infoh@DESKTOP MINGW64 /e/gitstudy06_clone (master)
$ git branch -r -------- 리모트 브랜치 목록
  origin/HEAD -> origin/master
  origin/function
  origin/hotfix
  origin/master
```

9. 원격 브랜치

➢ 업스트림 트래킹

- 모든 브랜치 정보를 확인하고 싶다면 **-a 옵션**을 사용함

```
$ git branch -a
```

- 저장소를 복제하면 원본과 동일한 트래킹 브랜치가 자동으로 설정됨
- 이번에는 복제 저장소의 트래킹 브랜치를 확인해보자
- 트래킹 브랜치는 **-vv 옵션**을 사용함

```
$ git branch -vv
```

9. 원격 브랜치

➢ 업스트림 트래킹

- 다음 명령을 실행하면 복제한 저장소의 트래킹 브랜치 목록을 확인할 수 있음

```
infoh@DESKTOP MINGW64 /e/gitstudy06_clone (master)
$ git branch -vv -------- 트래킹 브랜치 목록
* master dcdb1c1 [origin/master] master working...
```

- 현재는 트래킹 브랜치만 하나 표시됨
- 현재 master 브랜치가 원격 저장소의 origin/master로 업스트림 트래킹된 것을 확인할 수 있음
- 트래킹 브랜치가 하나만 표시된 것은 clone 명령어로 복제할 때 모든 브랜치를 한 번에 복제하지 않았기 때문임
- 불필요한 브랜치를 한 번에 다 가져오는 것은 현명하지 않음
- 이는 깃의 효율성과 연관됨

9. 원격 브랜치

➢ 업스트림 트래킹

- 다른 브랜치를 풀 작업으로 받아 트래킹 브랜치를 활성화하거나 직접 트래킹 브랜치를 지정할 수 있음
- 업스트림 동작을 위한 트래킹 브랜치는 직접 명령어를 실행하여 생성할 수 있음
- 우리는 원본 로컬 저장소에서 feature 브랜치를 function 리모트 브랜치로 등록함
- 다음 명령어를 사용하면 새로운 업스트림을 만들 수 있음

```
$ git checkout --track origin/브랜치이름
```

9. 원격 브랜치

➢ 업스트림 트래킹

- 복제된 저장소에서 새로운 업스트림을 만들어보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06_clone (master)
$ git checkout --track origin/function -------- 업스트림 브랜치 생성
Switched to a new branch 'function'
Branch 'function' set up to track remote branch 'function' from 'origin'.

infoh@DESKTOP MINGW64 /e/gitstudy06_clone (function)
$ git branch -vv -------- 트래킹 브랜치 목록
* function d84766c [origin/function] first -------- 트래킹 브랜치
  master   dcdb1c1 [origin/master] master working...
```

- 트래킹을 위해 새로운 function 브랜치를 생성한 후 원격 저장소는 origin/function으로 업스트림을 설정함
- 이 방식은 복제된 로컬 저장소와 원격 저장소의 브랜치 이름을 동일하게 생성한 예임

9. 원격 브랜치

➢ 업스트림 트래킹

- function 브랜치에서 코드를 수정하여 좀 더 실습해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06_clone (function)
$ code branch.htm -------- VS Code 실행
```

branch.htm

```
<h1>브랜치 실습을 합니다.</h1>
복제된 gitstudy06_clone의 function 브랜치를 수정합니다.
```

9. 원격 브랜치

➢ 업스트림 트래킹

- 커밋한 후 브랜치 정보를 다시 확인해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06_clone (function)
$ git commit -am "function working" -------- 등록 및 커밋
[function 85f1dfa] functionmaster2 working
 1 file changed, 2 insertions(+), 1 deletion(-)

infoh@DESKTOP MINGW64 /e/gitstudy06_clone (function)
$ git branch -vv -------- 트래킹 브랜치 목록
* function 85f1dfa [origin/function: ahead 1] functionmaster2 working -------- AHEAD 표시
  master   dcdb1c1 [origin/master] master working...
```

9. 원격 브랜치

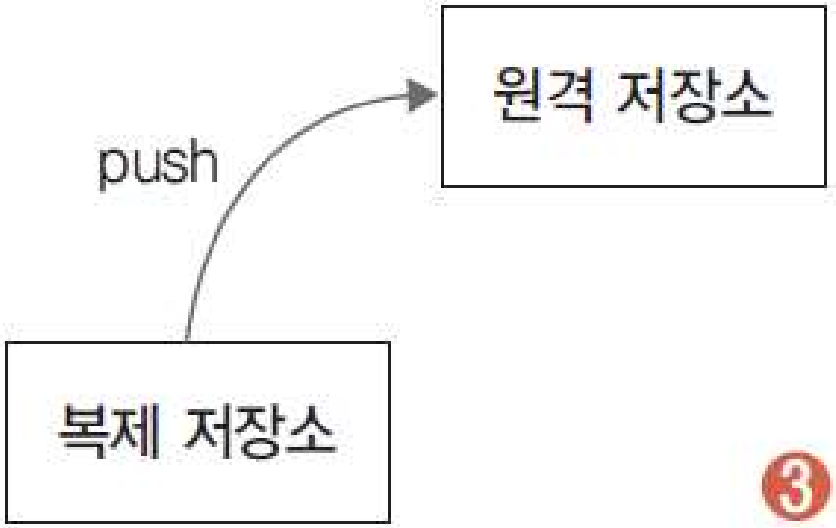
➢ 업스트림 트래킹

- function 브랜치 정보에 AHEAD 1이 표시됨
- 원격 저장소로 전송되지 않은 커밋이 하나 있다는 의미임
- push 명령어를 사용하여 원격 저장소로 새롭게 추가된 커밋을 전송함

```
infoh@DESKTOP MINGW64 /e/gitstudy06_clone (function)
$ git push -------- 커밋 전송
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 346 bytes | 115.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/jinygit/gitstudy06.git
   d84766c..85f1dfa  function -> function -------- 원격 저장소로 전송
```

9. 원격 브랜치

▼ 그림 6-23 변경 내용 전송



9. 원격 브랜치

➢ 업스트림 트래킹

- 다시 원본 저장소(gitstudy06)에서 git pull 명령어를 실행하면 feature 브랜치 정보로 자동 병합됨
- 먼저 원본 로컬 저장소로 이동한 후 feature 브랜치로 체크아웃함

```
$ cd ../gitstudy06 -------- 원본 로컬 저장소로 이동

infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git checkout feature -------- 브랜치 이동
Switched to branch 'feature'
Your branch is behind 'origin/function' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)
```

9. 원격 브랜치

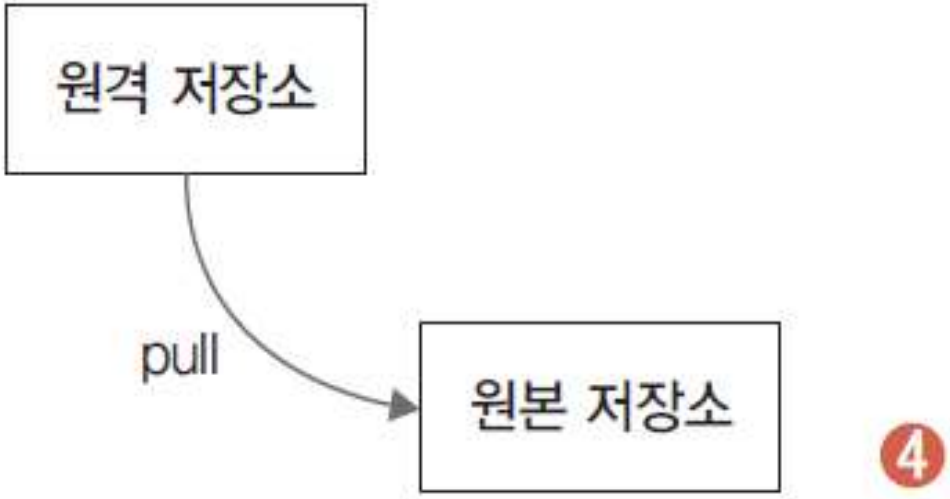
➢ 업스트림 트래킹

- 원격 저장소의 function 브랜치를 로컬 저장소의 feature 브랜치로 내려받음

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ git pull -------- 원격 저장소의 function 브랜치를 feature 브랜치로 내려받기
Updating d84766c..85f1dfa
Fast-forward
 branch.htm | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

9. 원격 브랜치

▼ 그림 6-24 변경 내용 받기



9. 원격 브랜치

➢ 업스트림 트래킹

- 로컬 저장소의 feature 브랜치에 수정한 내용이 반영되었는지 확인해보자
- cat 명령어를 사용하여 잘 반영된 것을 확인함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ cat branch.htm
<h1>브랜치 실습을 합니다.</h1>
복제된 gitstudy06_clone의 function 브랜치를 수정합니다. -------- 복제 저장소에서 변경된 내용
```

9. 원격 브랜치

➢ 원격 브랜치 복사

- 원격 저장소와 로컬 저장소의 브랜치 목록은 서로 다를 수 있음
- 다른 개발자가 원격 저장소에 새로운 리모트 브랜치를 생성할 수 있기 때문임
- 이렇게 생성된 원격 저장소의 리모트 브랜치를 이용해서 로컬 저장소에도 새로운 브랜치를 생성하여 동기화할 수 있음

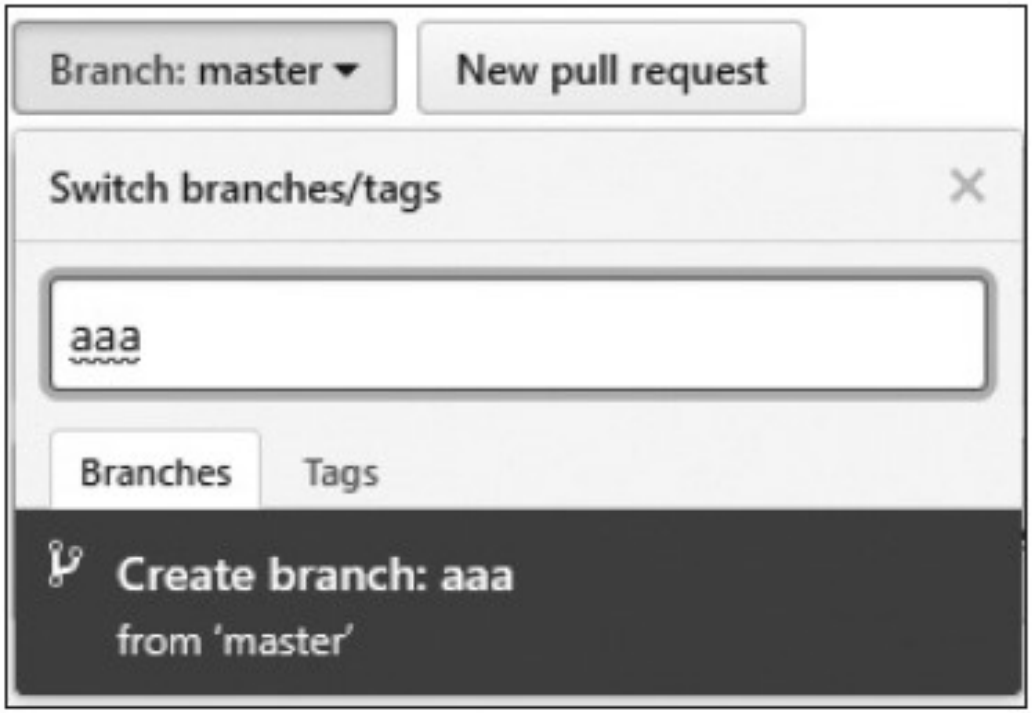
```
$ git checkout -b 새이름 origin/브랜치이름
```

9. 원격 브랜치

➢ 원격 브랜치 복사

- 실습을 위해 깃허브 원격 저장소에서 새로운 브랜치를 하나 생성함

▼ 그림 6-25 aaa 브랜치 생성

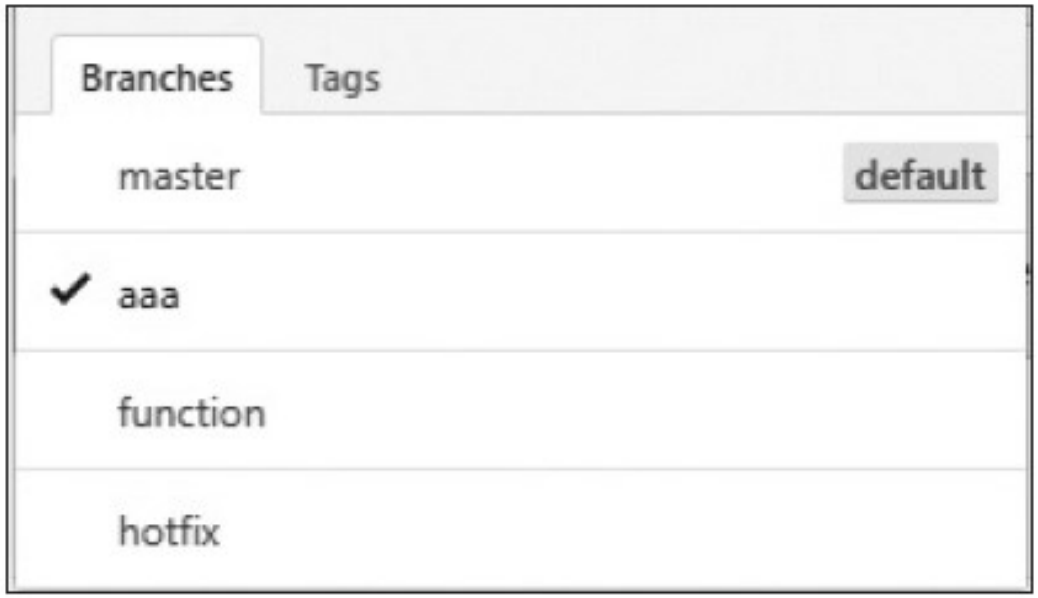


9. 원격 브랜치

➢ 원격 브랜치 복사

- 깃허브는 자체적으로 새로운 리모트 브랜치를 생성할 수 있음

▼ 그림 6-26 aaa 브랜치 생성



9. 원격 브랜치

➢ 원격 브랜치 복사

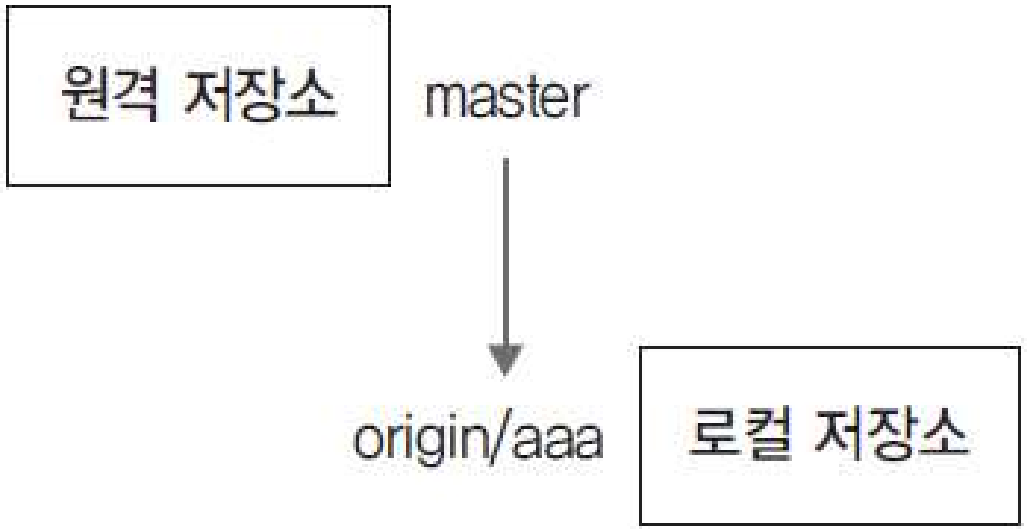
- 원격 저장소의 브랜치 정보를 로컬 저장소로 가져옴

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ git fetch -------- 브랜치 커밋 가져오기
From https://github.com/jinygit/gitstudy06
 * [new branch]        aaa          -> origin/aaa

infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ git branch -r -------- 원격 브랜치 확인
  origin/aaa -------- 깃허브에서 추가된 원격 브랜치
  origin/function
  origin/hotfix
  origin/master
```

9. 원격 브랜치

▼ 그림 6-27 원격 브랜치 가져오기



9. 원격 브랜치

➢ 원격 브랜치 복사

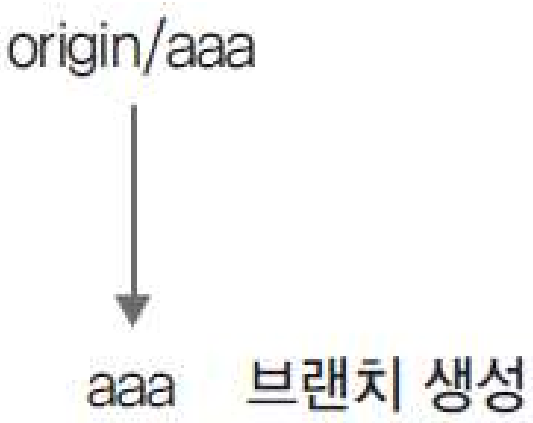
- 페치된 리모트 브랜치 목록을 이용하여 새로운 로컬 브랜치를 만들어보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (feature)
$ git checkout -b aaa origin/aaa -------- 브랜치 생성 및 이동
Switched to a new branch 'aaa'
Branch 'aaa' set up to track remote branch 'aaa' from 'origin'.

infoh@DESKTOP MINGW64 /e/gitstudy06 (aaa)
```

9. 원격 브랜치

▼ 그림 6-28 로컬 브랜치 생성



9. 원격 브랜치

➢ 원격 브랜치 복사

- 잘 만들어졌는지 트래킹 브랜치 목록을 확인함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (aaa)
$ git branch -vv -------- 트래킹 브랜치 목록
* aaa      dcdb1c1 [origin/aaa] master working... -------- 생성된 브랜치
  feature 85f1dfa [origin/function] functionmaster2 working
  footer  d84766c first
  hotfix  dcdb1c1 [origin/hotfix] master working...
  master  dcdb1c1 [origin/master] master working...
```

9. 원격 브랜치

➢ 원격 브랜치 복사

- 실습을 위해 aaa 브랜치에서 코드를 수정한 후 커밋함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (aaa)
$ code branch.htm -------- VS Code 실행
```

branch.htm

```
<h1>브랜치 실습을 합니다.</h1>
<h2>마스터 워킹 디렉터리 작업 중</h2>
새로운 원격 브랜치 aaa에서 수정 작업을 합니다.
```

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (aaa)
$ git commit -am "testing aaa" -------- 등록 및 커밋
[aaa c162b67] testing aaa
 1 file changed, 2 insertions(+), 1 deletion(-)
```

9. 원격 브랜치

➢ 원격 브랜치 복사

- 트래킹 브랜치 목록을 다시 한 번 확인함

```
infoh@DESKTOP-VAKLOFQ MINGW64 /e/gitstudy06 (aaa)
$ git branch -vv -------- 트래킹 브랜치 목록
* aaa      c162b67 [origin/aaa: ahead 1] testing aaa -------- AHEAD

feature 85f1dfa [origin/function] functionmaster2 working
footer  d84766c first
hotfix  dcdb1c1 [origin/hotfix] master working...
master  dcdb1c1 [origin/master] master working...
```

9. 원격 브랜치

➢ 원격 브랜치 복사

- aaa 브랜치에 서버로 전송하지 않은 커밋이 하나 있어 AHEAD 1로 표시됨
- push 명령어로 추가한 커밋을 다시 원격 저장소의 aaa 브랜치로 저장함

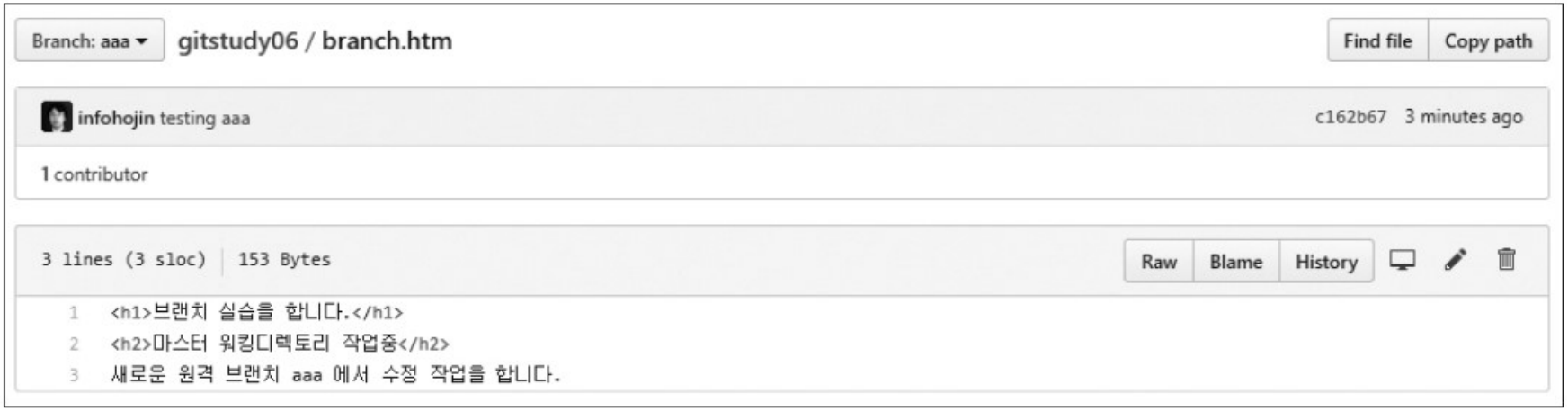
```
infoh@DESKTOP MINGW64 /e/gitstudy06 (aaa)
$ git push -------- 커밋 전송
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 366 bytes | 183.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/jinygit/gitstudy06.git
   dcdb1c1..c162b67  aaa -> aaa
```

9. 원격 브랜치

➢ 원격 브랜치 복사

- 깃허브의 원격 저장소 브랜치에서 수정된 내용을 확인함
- 깃허브에서 aaa 브랜치를 선택하고, 파일 목록에서 branch.htm 파일을 클릭함
- 수정된 내용과 커밋을 확인할 수 있음

▼ 그림 6-29 aaa 브랜치 수정 내용과 커밋 확인



9. 원격 브랜치

➢ 업스트림 연결

- 기존에 있는 브랜치를 업스트림으로 직접 설정할 수도 있음
- 브랜치를 생성한 후 직접 트래킹 브랜치를 지정함
- 업스트림을 직접 설정하면 원격 저장소로 트래킹 브랜치가 설정됨

```
$ git branch -u origin/브랜치이름
```

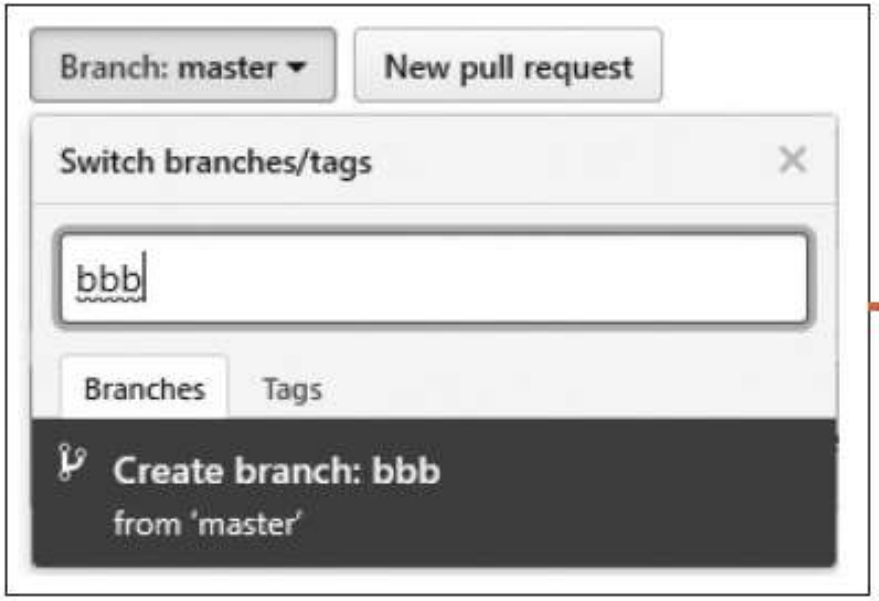
- -u 옵션은 --set-upstream-to의 약자임
- **기존 브랜치를 특정 원격 브랜치로 추적**함
- 한 번만 실행하면 이후로는 계속 업스트림으로 설정되어 작업할 수 있음

9. 원격 브랜치

➢ 업스트림 연결

- 실습을 위해 깃허브에 새로운 브랜치를 추가해보자

▼ 그림 6-30 bbb 브랜치 생성



9. 원격 브랜치

➢ 업스트림 연결

- 깃허브에 생성한 bbb 브랜치는 아직 로컬 저장소에 없음
- 깃 배시에서 깃허브 원격 저장소의 리모트 브랜치 목록을 확인함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (aaa)
$ git fetch -------- 서버의 브랜치 정보 갱신
From https://github.com/jinygit/gitstudy06
 * [new branch]        bbb            -> origin/bbb

infoh@DESKTOP MINGW64 /e/gitstudy06 (aaa)
$ git branch -r -------- 원격 브랜치 목록
  origin/aaa
  origin/bbb
  origin/function
  origin/hotfix
  origin/master
```

9. 원격 브랜치

➢ 업스트림 연결

- 원격 저장소에는 브랜치가 총 5개 있음
- 원격 저장소의 bbb 브랜치와 연결할 수 있는 bug라는 새 브랜치를 로컬 저장소에 하나 만듦

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (aaa)
$ git checkout -b bug -------- 브랜치 생성
Switched to a new branch 'bug'

infoh@DESKTOP MINGW64 /e/gitstudy06 (bug)
$ git branch -vv -------- 트래킹 브랜치 목록
  aaa     c162b67 [origin/aaa] testing aaa
```

9. 원격 브랜치

➢ **업스트림 연결**

```
* bug      c162b67 testing aaa -------- 새로운 브랜치
  feature 85f1dfa [origin/function] functionmaster2 working
  footer  d84766c first
  hotfix  dcdb1c1 [origin/hotfix] master working...
  master  dcdb1c1 [origin/master] master working...
```

9. 원격 브랜치

➢ 업스트림 연결

- bug 브랜치는 로컬 저장소에만 있는 브랜치임
- bug 브랜치를 원격 저장소의 bbb 리모트 브랜치로 **업스트림 설정**함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (bug)
$ git branch -u origin/bbb -------- 업스트림 연결
Branch 'bug' set up to track remote branch 'bbb' from 'origin'.

infoh@DESKTOP MINGW64 /e/gitstudy06 (bug)
$ git branch -vv -------- 트래킹 브랜치 목록
  aaa     c162b67 [origin/aaa] testing aaa
* bug     c162b67 [origin/bbb: ahead 1] testing aaa
  feature 85f1dfa [origin/function] functionmaster2 working
  footer  d84766c first
  hotfix  dcdb1c1 [origin/hotfix] master working...
  master  dcdb1c1 [origin/master] master working...
```

6.10 브랜치 전송

10. 브랜치 전송

➢ 브랜치 푸시

- 깃의 푸시 작업은 로컬 저장소의 파일들을 원격 저장소로 전송함
- 파일뿐만 아니라 브랜치 정보와 커밋까지 모두 전송함
- 처음으로 로컬 저장소에 새로운 원격 저장소가 등록되면 다음과 같이 push 명령어를 사용할 때 오류 메시지가 출력됨

예

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git push -------- 원격 서버 전송
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master -------- 오류 메시지
```

10. 브랜치 전송

➢ 브랜치 푸시

- 처음에는 커밋과 브랜치를 푸시하는 데 업스트림 설정이 필요함
- **원격 저장소 연결만으로 업스트림이 자동으로 설정되지는 않음**
- 이는 깃이 원격 저장소의 어느 브랜치에 어떻게 푸시해야할지 모르기 때문임
- 처음 푸시 작업을 하면 깃은 사용자에게 업스트림이 설정되지 않았다는 오류 메시지를 출력함
- 친절하게 설정하는 방법도 안내함
- 오류 메시지에서 표시된 명령어를 입력하면 됨

10. 브랜치 전송

➢ 브랜치 푸시

- 처음에는 다음과 같이 수동으로 트래킹 브랜치와 업스트림 설정을 해야 함

예

```
$ git push --set-upstream origin master
```

10. 브랜치 전송

➢ 브랜치 푸시

- 예를 들어 현재 master 브랜치를 origin 서버의 master로 업스트림 설정하려면 다음과 같이 함

예

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git push --set-upstream origin master -------- 업스트림 설정
git@211.110.1.195's password:
Counting objects: 36, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (24/24), done.
Writing objects: 100% (36/36), 2.84 KiB ¦ 0 bytes/s, done.
Total 36 (delta 12), reused 0 (delta 0)
To ssh://211.110.1.195/home/git/remote/
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

10. 브랜치 전송

➢ 브랜치 페치

- 리모트 브랜치 페치는 일반적인 커밋 페치와 동일함
- 리모트 브랜치를 페치한다고 해서 자동으로 로컬 저장소에 새로운 브랜치가 생성되지는 않음
- 페치 동작은 원격 저장소에서 리모트 브랜치 내용을 내려받기만 할 뿐이지 자동으로 병합하지 않기 때문임
- 리모트 브랜치가 페치되면 깃은 단순히 **원격저장소별칭/브랜치** 포인터만 생성함
- 원격 저장소에서 페치된 커밋들을 새로운 로컬 브랜치로 반영하려면 병합 명령을 실행해야 함

```
$ git merge 원격저장소별칭/브랜치이름
```

10. 브랜치 전송

➢ 브랜치 페치

- 가끔은 페치된 브랜치를 병합하지 않고 테스트만 하고 싶을 때도 있음
- 이때는 원격 브랜치의 포인터를 사용하여 임시 브랜치를 생성하거나 직접 체크아웃할 수 있음

```
$ git checkout -b 임시브랜치이름 origin/브랜치이름
```

6.11 브랜치 삭제

11. 브랜치 삭제

➢ 브랜치 삭제

- 생성된 브랜치를 삭제하는 것은 생각보다 간단함
- 브랜치를 삭제하는 것은 해당 브랜치 내용과 커밋을 모두 삭제하는 것임
- 삭제 명령을 실행할 때는 주의해야 함
- 브랜치 삭제는 크게 스테이지 상태에 따라 달라짐

11. 브랜치 삭제

➢ 브랜치 삭제

- 가장 먼저 주의할 점은 **현재 자신이 있는 브랜치는 삭제할 수 없다**는 것임
- 브랜치를 삭제하려면 master 브랜치나 다른 브랜치로 잠시 이동해야 함
- 예를 들어 앞에서 만든 로컬 저장소의 bug 브랜치를 삭제한다고 하자

예

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (bug)
$ git branch -d bug -------- 자신의 브랜치 삭제
error: Cannot delete branch 'aaa' checked out at 'E:/gitstudy06' -------- 오류 발생
```

- 현재 자신의 브랜치에서 삭제 명령어를 실행하면 오류 메시지가 출력됨
- 어느 브랜치로 체크아웃될지 모르기 때문임
- 삭제하고자 할 때는 다른 브랜치로 이동해서 삭제해야 함

11. 브랜치 삭제

➢ 일반적인 삭제 방법

- 일반적으로 브랜치를 삭제할 때는 -d 옵션을 사용함

```
$ git branch -d 브랜치이름
```

- **-d 옵션은 스테이지 상태가 깨끗할 때만 삭제를 허용**함
- 워킹 디렉터리에 작업한 기록이 있거나 add 명령어로 스테이지의 인덱스가 변경된 상태라면 삭제하지 않음
- 삭제하려면 반드시 최종 상태가 커밋되어 깨끗한 스테이지 상태여야 함
- 병합되지 않은 브랜치는 -d 옵션으로 삭제 할 수 없음

11. 브랜치 삭제

➢ **일반적인 삭제 방법**

- -d 옵션을 사용하여 footer 브랜치를 삭제해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (bug)
$ git branch -d footer -------- 브랜치 삭제
Deleted branch footer (was cc66812).
```

11. 브랜치 삭제

Git 교과서

➢ **강제로 삭제하는 방법**

- 워킹 디렉터리 또는 스테이지에 추가 커밋 작업이 남아 있다면 일반적인 방법으로는 브랜치를 삭제할 수 없음
- 이때는 강제로 삭제해야 함

11. 브랜치 삭제

➢ 강제로 삭제하는 방법

- 실습을 위해 hotfix 브랜치로 이동한 후 branch.htm 파일을 수정하고 저장함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (bug)
$ git checkout hotfix -------- 브랜치 이동
Switched to branch 'hotfix'

infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ code branch.htm -------- VS Code 실행
```

branch.htm

```
<h1>브랜치 실습을 합니다.</h1>
<h2>마스터 워킹 디렉터리 작업 중</h2>
<h3>hotfix 수정 작업입니다.</h3>
```

11. 브랜치 삭제

➢ **강제로 삭제하는 방법**

- 수정했다면 커밋함

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git commit -am "hotfix working" -------- 등록 및 커밋
[hotfix 0225e85] hotfix working
 1 file changed, 2 insertions(+), 1 deletion(-)
```

11. 브랜치 삭제

➢ 강제로 삭제하는 방법

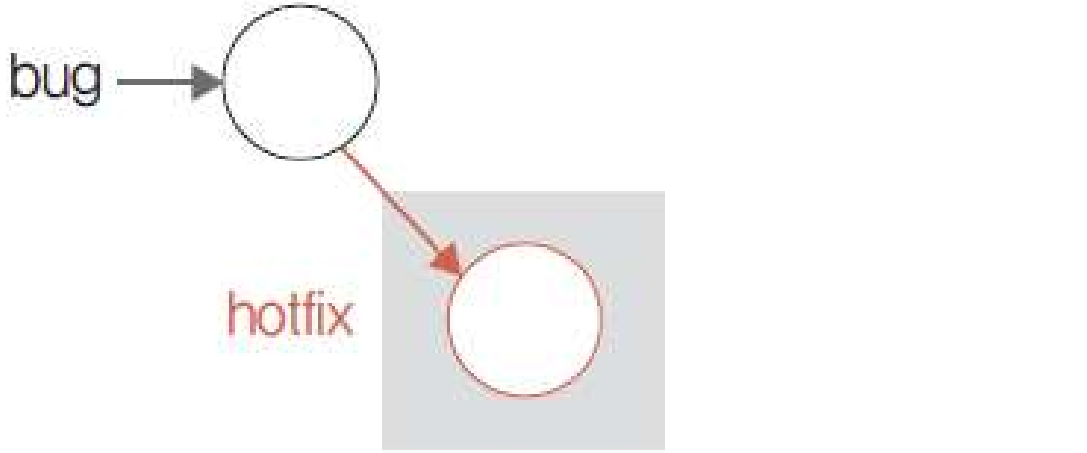
- master 브랜치로 체크아웃하여 hotfix 브랜치를 삭제해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (hotfix)
$ git checkout master -------- 브랜치 이동

infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git branch -d hotfix -------- 브랜치 삭제
error: The branch 'hotfix' is not fully merged.
If you are sure you want to delete it, run 'git branch -D hotfix'. -------- 오류 메시지
```

11. 브랜치 삭제

▼ 그림 6-31 커밋이 있는 브랜치 삭제



삭제하면 커밋 정보가 사라짐.
삭제를 방지

11. 브랜치 삭제

➢ **강제로 삭제하는 방법**

- 브랜치를 삭제하면 해당 브랜치에서 작업했던 커밋 기록들이 같이 삭제되기 때문에 오류가 발생함
- 이때는 브랜치를 강제로 삭제해야 함
- **대문자 -D 옵션을 사용하면 강제로 브랜치를 삭제**할 수 있음

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git branch -D hotfix -------- 브랜치 삭제
Deleted branch hotfix (was 8026ed6).
```

- -D 옵션으로 잘 삭제된 것을 확인할 수 있음

11. 브랜치 삭제

➢ **소스트리에서 삭제하는 방법**

- 소스트리에서도 브랜치를 삭제할 수 있음
- 먼저 소스트리의 왼쪽에서 브랜치 목록 중 삭제하고 싶은 것을 하나 선택함
- 브랜치를 선택한 상태에서 마우스 오른쪽 버튼을 누름
- 다음과 같이 브랜치를 삭제할 수 있는 메뉴가 나오면 **feature 삭제**를 선택함

11. 브랜치 삭제

▼ 그림 6-32 소스트리에서 브랜치 삭제

11. 브랜치 삭제

➢ 리모트 브랜치를 삭제하는 방법

- 원격 브랜치를 삭제하려면 먼저 **삭제 명령을 푸시**해야 함

```
$ git push origin --delete 리모트브랜치이름
```

- 깃허브의 aaa 브랜치를 삭제해보자

```
infoh@DESKTOP MINGW64 /e/gitstudy06 (master)
$ git push origin --delete aaa -------- 삭제 명령 전송
To https://github.com/jinygit/gitstudy06.git
 - [deleted]         aaa
```

- 깃허브 페이지에서 aaa 브랜치가 삭제된 것을 확인할 수 있음
- 원격 저장소의 브랜치를 삭제하면 리모트 브랜치에 기록된 커밋도 모두 삭제됨
- 함께 사용하는 브랜치라면 신중하게 삭제하는 것이 좋음

6.12 정리

12. 정리

➢ **정리**

- 브랜치는 기존 코드를 가상으로 분리함
- 분리해서 격리된 브랜치는 상호 간섭 없이 별개 작업을 수행할 수 있음
- 새로운 프로젝트로 발전시켜 나아갈 때 브랜치 기능은 매우 유용함
- 깃의 브랜치는 다른 개발자와 협업하여 프로젝트를 진행할 때도 매우 유용함
- 서로 간섭 없이 코드를 개선하고, 나아가 병합도 가능하기 때문임