Git Essentials

우리는 왜 그리고 어떻게 Git 을 사용해야 하는가?

목차

그림 목록

그림 목록

1. 왜 Git을 '공부'하지 않는가?

이 장에서는 Git을 공부해야 하는 대상으로 보아야 하는 이유에 대해 이야기한다.

1.1 공부할 필요성

이 주제를 다루려면 먼 옛날 이야기부터 시작해야겠다. 그렇다고 아더왕이 살던 시절이나 민주화 시절과 같은 정말 아주 먼 옛날이 아니라, 한국이란 나라에서 개발 문화가 동트기 시작할 그 무렵으로 말이다.

2000년대 후반의 개발 환경에 Git은 없었다.(다시 한번 말하지만 한국 이야기다.) 이 때만 해도 Git은 커녕 Subversion을 제대로 쓸 줄 아는 사람도 극히 드물었다.

버전 관리라는 것은 하나의 **백업 도구**일 뿐이었다. 다른 사람에게 코드를 공유하려고 할 때, 점심먹으러 갈 때, 혹은 퇴근해서 집에 갈 때, 아니면 그냥 생각나서 하는 것이 커밋이었다. 어떤 다른 의도는 전혀 없었고 단지 중앙 저장소에 저장해두려는 목적일 뿐이다. 지금의 버전 관리 문화에서도 공유의 목적이 있긴 하지만 그 때의 공유와는 확연히 다르다.

정말 웃지 못할 풍경은 지금부터다. Subversion을 통해 버전 관리를 하는 사람들에게 '너 SVN 잘 알아?'라고 물어보면 열에 아홉은 '음... 글쎄... 그냥 이클립스에서 커밋하고 체크아웃하고... 그런 거 아냐?'라고 대답했었다. 조금 더 구체적으로 'SVN 서버 설치해본 적 있어?'라고 물어보면 거의 다 '아니오'를 외쳤다. 물론 현재 약간 규모가 있는 회사의 몇몇 개발자들 중에는 혼란의 시기(개발자도 하고, 서버 관리자도 하고)에 자신이 직접 회사의 버전 관리 시스템을 담당했었던 사람도 있다.

대부분의 사람들에게 Subversion이란 그저 이클립스(혹은 다른 IDE의) 플러그인일 뿐이었다.(혹은 TortoiesSVN) 공부할 필요없었던 이유가 그러한 도구들이 버전 관리의 전부였기 때문이다. 아는 사람은 알겠지만 Subversion 또한 많은 노력을 요구하는 버전 관리 시스템이다. 그러나 도구가 그러한 노력의 비용을 많이 감쇄시켜주었다.

하지만, Subversion 등의 버전 관리 시스템들이 공부의 대상이 되지 못했던 것은 더 설득력 있는 다른 이유가 있었다. 바로 **문화**다. 현재의 개발 문화는 (모두 다 그런 것은 아니겠지만) 창조와 협력의 문화로 이동하고 있다. '어떻게 하면 데드라인을 맞출까?'를 고민하는 것이 아니라 '어떻게 하면 쓸모있는 앱을 만들까?'에 집중하고 있는 것이다. Subversion은 그러한 문화 속에서 그리 탄력적인 기능을 발휘하지 못하는 구조를 가지고 있다. 반면 Git은 그러한 문화 속에서 매우 유연하게 대처할 수 있도록 (의도한 것이든 아니든) 설계되었다. 그것을 반증하는 것이 바로 Github다.

Github를 통한 커뮤니케이션 성장세는 무서울 정도다. 거의 대부분의 오픈 소스 진영들은 이 Github를 통해 자신들의 플로우를 이어나가고 있다.

그런데 Git을 공부하지 않을 이유가 있을까? 그 저변에는 여전히 'Git은 버전 관리를 위한 도구일뿐'이라는 생각 때문이다. 사실 도구라는 말은 틀린 말이 아니다. 정확하다. 도구일 뿐이다. 하지만 도구를 정확히 알지 못하면 생산성은 떨어지게 된다.

또, 그 도구로 인해 꽃핀 문화를 보자. Github가 저토록 성공할 수 있었던 이유는 무료 저장소여서가 아니다. *fork*와 *pull request*라는 플로우 때문에 성장할 수 있었다. 지리적으로 떨어져 있는 다대다 관계에 놓인 개발자들이 자신들의 의견을 코드로 말할 수 있게끔 했다. 그리고, 서로가 서로의 아이디어를 훔쳐오는 것이 아니라 채택하게 된다. 정말 놀라운 문화 아닌가?

자신이 개발자라면 이러한 개발 문화에 발담그지 않을 이유가 없을 것이다. 그러므로 Git을 공부해야 한다.마크업 개발은 프런트-엔드 페이지의 기본 골격을 형성하기 때문에 디자인, 브라우저, 스크립트, 성능, 접근성 등과 긴밀한 관계가 있다. 즉, 마크업 개발을 잘 해야 모든 브라우저에서 콘텐츠를 손실 없이, 빠르고

쉽게 사용자에게 전달할 수 있다. 코딩 컨벤션은 이러한 조건을 만족시키기 위해 마크업 개발자가 지켜야 할 표준을 제시한다.

또한, 유지보수에 투자되는 비용을 최소화하기 위해 통일된 코드 작성법을 제시한다. 코드를 최초로 작성한 사람이 끝까지 유지보수할 확률은 매우 낮다. 따라서, 최초 개발자가 아닌 사람도 코드를 빠르고 정확하게 이해할 수 있도록 작성하는 것은 코드의 유지보수 비용을 절감하고 업무 효율을 높이는 데 결정적인 역할을 한다.

적어도 한 프로젝트의 마크업 코드는 같은 코딩 컨벤션에 따라 작성해야 한다. 코딩 컨벤션을 준수하면 프로젝트 멤버 간 코드 공유도 쉬워지고, 일관성 있게 코드를 작성할 수 있다. 어떤 코딩 컨벤션을 선택하느냐가 중요한 것이 아니라, 통일된 기준으로 소스 코드를 작성하는 것이 중요하다.

1.2 도구로 접근하는 Git

Subversion을 사용하는 사람들이 많이 사용하는 도구는 아마도 IDE 플러그인 혹은 TortoiseSVN 일 것이다. 도구가 주는 편안함은 달콤하다. 피곤함에 찌들어 있을 때 커맨드라인과의 싸움을 사람을 더욱 더 지치게 만든다. 그래서 도구들은 아주 단순한 인터페이스로 개발자들을 현혹시킨다.

타이핑해야 하던 것을 버튼 하나만 누르면 끝낸다. 그리고 자신이 무엇을 하고 있는지 눈에 보이는 것 같아 왠지 편안한 기분이 든다. 하지만 아쉽게도 진짜 무슨 일이 벌어지고 있는지 알지 못한다. 그리고 의외의 상황에 놓이게 되면 더욱 더 어려움을 겪게 된다. 대부분의 도구들이 문제가 없는 상황에서는 단순함을 제공해주지만 예외적인 상황이 발생하게 되면 알 수 없는 메시지들을 내뱉으며 모든 것을 사용자의 몫으로 돌린다.

그렇게 되는 이유는 '미지의 세계'이기 때문이다. 커밋에도 여러 가지 종류의 커밋이 있으며, 그에 따라 옵션을 주는 방법 또한 달라진다. 하물며 커밋을 수정해야 하는 경우나, 추가 브랜치에서 작업해야 하는 경우라면 점점 더 알 수 없는 '미지의 세계'로 빠져든다. 도구는 우리에게 '이걸 누르면 이렇게 돼'라고 말하지만 Git을 공부하지 않은 사람은 정작 *이렇게 된다*라는 의미를 전혀 알지 못한다.

그러므로 Git은 공부해야 하는 대상이며 우리에게 화려한 시각적 정보를 주는 도구들은 아쉽게도 큰 도움이 되지 않는다. Git을 사용하는데 있어서 최고의 도구는 CLI다.

참고

최근에는 많은 개발 도구들이 CLI 기반으로 나오고 있다. 물론 예전부터 그렇긴 했지만, 자바스크립트와 node.js의 영향으로 인해 그것이 더 심해지고 있다.

2. 시작하기 전에

이 장에서는 Git을 설치하는 방법, 사용 환경, 명령어의 구조 그리고 환경 설정하는 법에 대해 알아본다.

2.1 설치

Git은 거의 모든 플랫폼을 지원한다. Git 설치 전에 설치 확인을 참고하여 이미 설치되어 있는지 확인하자.

2.1.1 데비안/우분투 계열의 리눅스

```
$ sudo apt-get install git
```

2.1.2 페도라 계열의 리눅스

```
$ sudo yum install git
```

2.1.3 OS X

```
$ brew install git
```

참고

homebrew를 설치해야 한다.

혹은 git-scm.com에서 설치 파일(dmg)을 다운로드받아 설치하면 된다.

2.1.4 Windows

윈도우즈에 Git을 설치하는 방법에는 여러 가지 방법이 있는데 여기서는 두 가지만 소개하겠다.

- git-scm.com에서 다운로드하고 설치한다.
  - 정식 버전을 이용한다.(가장 최근의 버전을 사용하려면 Git 저장소을 이용해 컴파일하여 사용해야 한다.)
  - 별도의 CLI를 사용하거나 설정을 해줘야 편리하게 Git을 사용할 수 있다.
- Github for Windows 설치를 추천한다.
  - 포터블 버전을 이용하게 된다.
  - Git shell 이라는 CLI가 제공되어 별도의 CLI를 사용하지 않아도 Git을 편리하게 이용할 수 있다.
  - Github 계정이 필요하다.

2.1.5 설치 확인

```
1  $ git --version
2  git version 2.0.1
```

2.2 Git 사용 환경

Git을 좀 더 편하게 사용하기 위한 도구들은 많다. 그러나, 그 어떤 도구도 CLI 환경에서 커맨드를 이용하는 방법을 뛰어넘지는 못하는 것 같다.

GUI 환경의 도구들은 편한 점이 분명히 있으나 몇 가지 문제점이 있다.

- 대부분의 도구들이 편의성을 위해 만들어져 Git의 많은 커맨드 옵션들을 무시하게 된다. Git은 상황에 맞는 적절한 대응을 위해 여러 가지 옵션을 제공하는데 이러한 편리한 옵션들을 사용할 수 없게 된다.
- Git에 대한 경험을 방해한다. Git은 도구일 뿐이기에 공부할 필요가 없다는 사람들도 있을 수 있으나 경험으로써의 공부가 필요하다. 그런데 많은 그래픽 기반의 도구들은 경험의 폭을 줄어들게 한다.
- 어떤 명령이 실행되는지 혹은 실행됐는지 알 수가 없다. 그러므로 어떤 상황에 놓였을 때의 대처 능력이 현저히 떨어지게 된다.

물론 GUI 환경의 도구들을 사용하면 타이핑을 하지 않고도 많은 것들이 가능해진다. 그러나 잃는 것이 너무 많다. 그러한 도구들만 이용해서 Git을 사용한 사람들은 Git을 경험한게 아니다. 도구를 익혔을 뿐이다.

이것은 SVN도 마찬가지인 거 같다. SVN이 주류인 시절(불과 몇 년전)에 대부분의 사람들이 Eclipse의 SVN 플러그인 혹은 TortoiseSVN으로 버전 관리를 사용했다. SVN은 Git에 비해 use-flow가 꽤 간단했기 때문에 그래픽 기반 도구들만으로도 충분했는지 모르겠다. 그리고 한편으로는 use-flow 자체를 도구가 강제했는지도 모른다. 사람들이 매일 매일 사용할 만한 몇 가지를 응축하여 제공함으로써 효율성을 높인다는 취지였기에 그러한 환경이 됐는지도 모르겠다.

CLI에 익숙해지면 또 좋은 점은 많은 도구들이 여전히 그리고 더욱 더 CLI 기반에서 제공된다는 점이다. 그것도 리눅스의 스타일로 말이다. node.js 기반의 Grunt나 Gulp 등의 빌드 도구, 자바 빌드 도구인 Maven이나 Gradle도 기본적으로는 CLI를 기반으로 한다. CLI에 대한 적응 및 활용 필요성이 점점 증대되고 있다.

2.3 명령어 구조

각자의 터미널에서 git을 입력하면 다음과 같이 출력될 것이다.

```
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value] [--exec-
path[=<path>]] [--html-path] [--man-path] [--info-path] [-p|--paginate|--no-
pager] [--no-replace-objects] [--bare] [--git-dir=<path>] [--work-tree=<path>]
[--namespace=<name>] <command> [<args>]

The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find by binary search the change that introduced a bug
   branch     List, create, or delete branches
   checkout   Checkout a branch or paths to the working tree
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   fetch      Download objects and refs from another repository
   grep       Print lines matching a pattern
   init       Create an empty Git repository or reinitialize an existing one
   log        Show commit logs
   merge      Join two or more development histories together
   mv         Move or rename a file, a directory, or a symlink
   pull       Fetch from and integrate with another repository or a local
branch
   push       Update remote refs along with associated objects
   rebase     Forward-port local commits to the updated upstream head
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
   show       Show various types of objects
   status     Show the working tree status
   tag        Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' lists available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

2.3.1 Subcommand

```
$ git add foo.bar
$ git commit -m "foo~ bar~"
```

Git은 하위 명령어를 가지고 있다. 하위 명령어 별로 별도의 옵션을 가지고 있으며, 이에 따라 같은 하위 명령어일지라도 (비슷하지만) 다른 효과를 발휘한다.

2.3.2 옵션

옵션은 단일 하이픈(-) 그리고 더블 하이픈(--)을 사용하기도 하는데 단일 하이픈은 짧은 형식, 더블 하이픈은 긴 형식의 옵션이다. 그래서 위에서 예로 든 git commmit -m은 다음과 같이 바꿔 사용할 수 있다.

```
$ git commit --message "foo~ bar~"
```

2.3.3 순수 이중 대시

간혹 더블 하이픈(--)을 옵션이 아닌 형태로 사용하기도 한다.

```
$ git reset HEAD -- foo.bar
```

이것을 **순수 이중 대시**라고 하는데, 명령어의 옵션과 인수를 구분하기 위해 사용된다.

이 이중 대시에 따라 명령어의 의미가 완전히 달라지므로 유의해야 한다.

```
# "foo.bar"라는 이름의 태그를 체크아웃한다.
$ git checkout foo.bar
# "foo.bar"라는 이름의 파일을 체크아웃한다.
$ git checkout -- foo.bar
```

2.4 환경 설정

Git은 세 가지 계층에 걸쳐 설정 파일을 가지고 있고, 그 설정들이 모두 모여서 최종 설정을 이룬다.

설정 파일의 계층는 다음과 같다.

- ***system***
  - git이 설치되어 있는 위치에 있는 설정 파일로 시스템의 모든 사용자에게 적용되는 설정들을 정의한다.
  - 설정 시, *--system* 플래그를 사용하도록 한다.
  - system 환경 설정의 경우, 플랫폼 별로 모두 위치가 다른데, 리눅스 계열의 경우 /etc/gitconfig이고 OSX나 윈도우즈의 경우, 설치 위치 내 etc/gitconfig 내에 있다.
- ***global***
  - 시스템 계정 사용자 별로 따로 환경 설정을 따로 보관할 수 있다.
  - global 환경 설정 파일은 계정 사용자의 홈 디렉토리에 있는 .gitconfig 파일이다.
  - 설정 시, *--global* 플래그를 사용하도록 한다.
- ***repository***
  - 저장소 별로 정의되는 환경 설정을 말한다.
  - .git 디렉토리의 하위에 config라는 파일로 존재한다.
  - 설정 시, *--local* 플래그를 사용하도록 한다.

저장소 내 환경 설정 파일을 제외하고는 모두 없을 수도 있는데 동작 자체에는 크게 문제가 없다.

일반적으로 Git을 설치하고 나면 하는 최초의 환경 설정은 다음과 같다.

```
$ git config --global user.name '<자신의 이름이나 별칭>'
$ git config --global user.email '<유요한 이메일 주소>'
```

두 가지 설정을 *--global* 플래그로 지정한 이유는 위에서 설명한 계층 별 환경 설정을 봤으면 이해가 갈 것이다.

현재 시스템에 설정되어 있는 환경 설정들을 보려면 다음과 같은 명령어로 확인이 가능하다.

```
1  $ git config --list
2  credential.helper=osxkeychain
3  user.name=Choi Leejun
4  user.email=devcken@gmail.com
```

위 명령은 저장소를 갖지 않은 일반 디렉토리에서 실행된 것인데 저장소가 포함된 프로젝트 디렉토리에서 실행하면 저장소 설정까지 함께 포함되어 나온다. 그리고, 각 계층 별로 설정을 보고 싶으면 계층 플래그를 함께 전달하면 된다.

Git의 환경 설정 파일 포맷은 *ini* 파일의 포맷을 따른다. 그래서 환경 설정 파일을 확인해보면 다음과 같은 형태를 보인다.

```
# system gitconfig
$ cat /usr/local/git/etc/gitconfig
[credential]
    helper = osxkeychain
```

앞서 말한바와 같이 *ini* 파일의 포맷이므로 그룹, 키, 값의 계층 구조를 따른다. dot(.)으로 구분하여 세분화된 키도 존재한다.

실제로 저장되는 파일의 포맷은 *ini* 형식인데, 명령어를 이용할 때, 그룹과 키는 dot(.)으로 연결한다. 키와 값은 출력 시에만 equation mark(=)으로 연결하므로 주의해야 한다.

설정되어 있는 내용을 환경 설정에서 빼려면 다음과 같이 해야 한다.

```
$ git config --unset --global user.email
```

이 때 조심해야 하는 것은 해당 설정이 어느 계층에 속하는지 정확히 확인한 후에 실행해야 한다는 것이다. 빼서는 안되는 환경 설정을 잘못된 곳에서 뺄 수도 있다.

3. 파일 핸들링

이 장에서는 Git이 파일을 어떻게 다루는지에 대해서 설명한다.

3.1 Git 내부 살짝 들여다보기

Git의 내부는 git-scm book, 10 git internals에 잘 나와 있다. 하지만 git 내부를 초급자가 알아보기에는 너무 많고 어려운 내용으로 구성되어 있다. 하지만, 또 너무 모르고 넘어가기에는 설명이 부족한 면이 있어서 최대한 간략하게 간추리도록 한다.

3.1.1 저장소(repository)

git의 저장소는 **객체(object)**와 **인덱스(index)**라는 데이터 구조를 관리한다. 저장소 내의 객체들은 객체 저장소에서 보관, 관리된다.

3.1.2 객체 저장소

원본 데이터 파일과 커밋 로그 메시지, 작성자 정보 등 많은 정보들이 객체 저장소에 보관된다. 객체 저장소는 네 가지 유형의 객체를 관리하는데 이를 **Git 객체**라고 한다.

다음은 Git 객체의 종류이다.

- *Blob*
  - 파일 그 자체를 나타내는 데이터로 파일에 대한 메타정보(이름, 생성일시, 수정일시 등의 부수정보)는 보관하지 않는다.
  - Blob은 blob ID로 구분된다.
- *Tree*
  - 파일 시스템의 디렉토리를 한 레벨만 표현한다.
  - 한 개의 디렉토리 내 파일들에 대한 blob ID, 경로 이름과 약간의 메타데이터를 기록한다.
  - 다른 트리 객체에 대한 재귀적 참조가 가능하므로 파일 시스템에 대한 계층적 구조를 표현할 수 있다.
- *Commit*
  - 커밋은 변경 사항에 대한 메타데이터를 가지는데, 이 메타데이터는 작성자, 커밋 실행자, 커밋 날짜, 로그 메시지 등으로 구성된다.
  - 하나의 커밋은 하나의 트리를 가르키고, 트리는 커밋한 순간의 저장소 상태를 표현하므로 저장소에 대한 스냅샷을 만들어낸다.
  - 거의 모든 커밋은 상위 커밋을 가지는데 최상위 커밋만 상위 커밋을 가지지 않는다.
  - 두 개의 상위 커밋을 가지는 특수한 커밋도 존재한다.
- *Tag*
  - 태그는 다른 객체에 인간 친화적인 임시 이름을 할당하는데 사용된다.

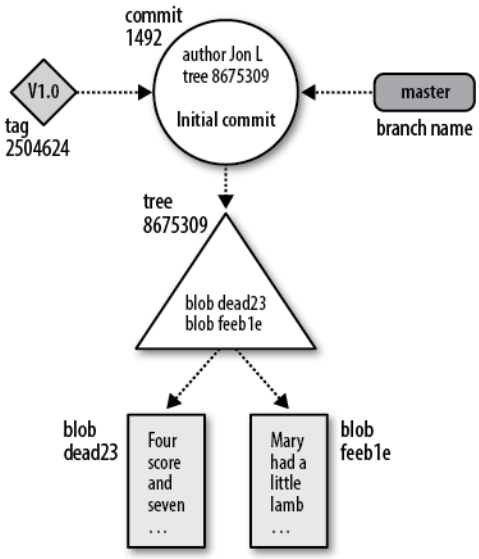

그림 3-1 Git 객체

위 그림에서 원은 커밋(commit) 객체 한 개를 나타낸다. 작성자 정보와 커밋 로그 메시지를 가지고 있으며 자신의 트리 객체 ID 또한 가지고 있다.

트리 객체는 blob을 가리켜 한 개의 커밋에 연결되어 있는 blob 목록을 표현한다. 또한 다른 트리를 가리키기도 한다.

blob은 트리 객체에 의해서만 참조되며 파일의 컨텐츠를 나타낸다.

커밋은 브랜치에 속하고 커밋의 이름(해시)를 결정하는데 중요한 역할을 한다.(Git 객체는 아니다.)

한 개의 태그는 커밋 하나만을 가리킬 수 있으나 한 개의 커밋 여러 커밋이 가리킬 수는 있다.

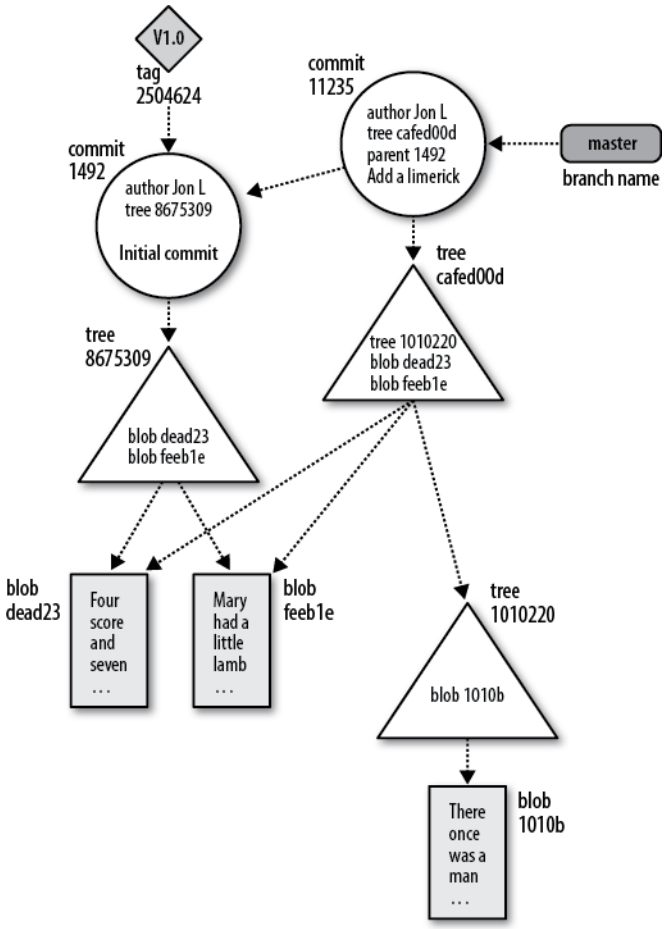

그림 3-2 Git 객체와 커밋 간의 관계

새로운 커밋이 추가되었다. 새로운 커밋에는 하위 디렉터리에 한 개의 파일을 추가한 상태에서 이루어졌다.

새로운 커밋은 자신의 상위 커밋을 가리키며, 동시에 한 개의 트리를 가리킨다. 이전 커밋에 포함된 두 blob은 변화가 없었기에 새로운 커밋의 트리 객체인 *cafed00d*는 이전 커밋의 두 blob을 가리킨다.

새로운 커밋에서 하위 디렉터리가 포함됐으므로 *cafed00d* 트리 객체의 하위 트리 객체인 *1010220*이 생겼으며 *cafed00d* 트리 객체가 이를 가리킨다. 또, 새로이 추가된 파일의 blob이 하위 트리인 *1010220*에 의해 참조된다.

새로운 커밋이 추가됐으므로 *master* 브랜치는 새로운 커밋을 가리키게 되었다.

3.1.3 인덱스

Git에서 인덱스(index)라고 하는 것은 전체 저장소 디렉토리 구조를 표현하는 바이너리 파일을 말한다. `.git/index` 파일이 바로 인덱스 파일이다.

인덱스 파일은 바이너리 파일이기 때문에 사람은 바로 열람이 불가능하며 명령어를 사용해 열람해야 한다.

```
$ git ls-files
```

```
index.html
```

`git ls-files` 명령은 인덱스 내 컨텐츠 목록을 보여주는데 조금 더 자세한 내용을 보려면 *--stage* 플래그를 사용해야 한다.

```
$ git ls-files --stage
100644 21dbefb04ea8d5cef18871f0db5d61c1bc78378f 0    index.html
```

--stage 플래그를 전달하면 컨텐츠의 이름과 함께 파일의 mode 비트, 스테이지 번호를 함께 출력한다.

즉, 인덱스라는 것은 커밋하기 전에 커밋 후보 컨텐츠들을 모아두는 곳으로 파일에 대한 추가, 변경, 삭제 등의 변경 사항을 기록, 유지하게 된다.

3.1.4 객체 이름

Git 객체에는 주소화 이름이라는 것이 있는데, 객체별로 고유한 이름을 위해 각 객체 콘텐츠에 SHA1을 적용하여 얻어지는 SHA1 해쉬 값을 부여한다.

```
21dbefb04ea8d5cef18871f0db5d61c1bc78378f
```

위와 같이 40자리 16진수 문자열로 표현되며 160비트 해쉬 값이다. Git에서는 이를 좀 더 짧은 (보통) 7자리로 표현하여 간결하게 사용한다.

3.1.5 컨텐츠 추적

Git은 파일의 이름이나 경로를 추적하지 않는다. 이 말은 이름과 경로로써 파일을 판단하지 않는다는 의미이다.

대신 파일의 내용을 추적하는데 바로 SHA1을 이용해 해쉬 값을 가지고 변경 사항을 감지하게 된다. 그러므로 파일에 변화가 생기면 새로운 SHA1가 생기게 된다.

3.1.6 팩 파일(pack file)

Git은 저장소의 전체 스냅샷을 기록한다. 그래서, 파일, 파일의 용량, 커밋 횟수가 늘어날 수록 비효율적일 수 있다. 그래서 Git은 변경된 콘텐츠와 가장 유사한 콘텐츠를 찾고 두 콘텐츠 간의 델타(Δ, 차이점)를 따로 저장한다. 그런 후에, 팩 파일에 이 델타를 함께 보관한다.

패키징된 파일은 zlib으로 압축된 내용을 보관하는데, 다른 객체들과 마찬가지로 객체 저장소에 저장하며 네트워크를 통한 데이터 전송 시에도 그 효율성을 발휘한다.

3.2 네 가지 상태

Git의 영향권 내에서 파일은 네 가지 상태에 놓이게 된다.

먼저, 상태에 대해 이야기하기 전에 세 가지 공간에 대해서 먼저 알아보자.

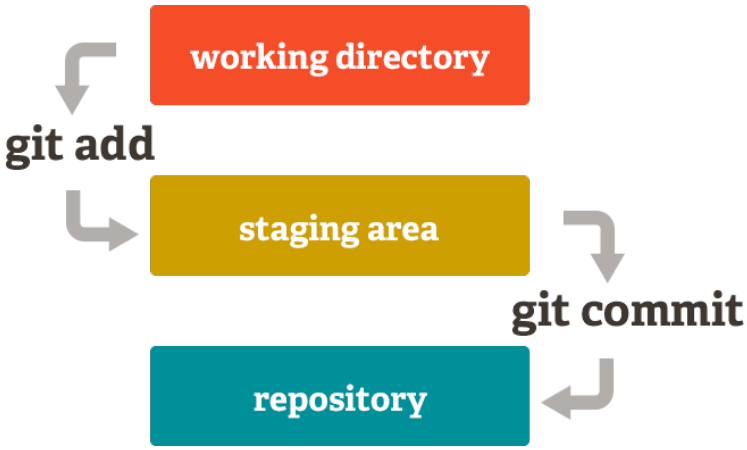

그림 3-3 Git의 3가지 공간

working directory(작업 디렉토리)

파일을 추가/수정/삭제하는 공간이라고 생각하면 된다. 쉽게 말해 Git 저장소를 제외한 프로젝트 디렉토리 내의 모든 공간이 'working directory'다.

staging area(인덱스)

보통 *index*라고 하며 저장소와 작업 디렉토리 중간에 있는 공간이다. 파일들이 커밋되기 전에 모여있는 임시 저장 공간으로 모든 파일은 이 공간을 거쳐 저장소로 옮겨지게 된다.

repository(저장소)

커밋을 비롯해 관련된 모든 객체들이 저장되는 공간이다. 저장소로 초기화되고 나면
생성되는 `.git` 디렉토리를 저장소로 봐도 크게 무리는 없으나 *index*도 함께 자리하고 있음을 잊지 말자.

이제 네 가지 상태에 대해 알아보자.

참고

git-scm.com의 책 Pro Git에서는 three states를 이야기하는데, 세 가지 상태만을 이야기하는 것이 오히려 혼동을 줄 수 있다고 생각한다. 그래서 나는 four states에 대해서 이야기할까 한다.

3.2.1 추적되지 않음(untracked)

추적되지 않았다는 것은 git이 해당 파일을 처리하지 않을 것임을 의미한다. 이 섹션의 제목에서 +1이 나타내는 무시됨(ignored) 상태를 빼고 나머지는 모두 untracked 상태가 아니다. 그러니까 모두 tracked, 추적되고 있다는 뜻이다.

추적되지 않은 파일은 모두 작업 디렉토리에 있는 상태이며, 인덱스나 저장소에는 한번도 들어간적이 없는 파일이다.

3.2.2 수정되지 않음(unmodified)

저장소에 커밋된 파일이 수정되지 않은 경우, 작업 디렉토리, 인덱스 그리고 저장소에서의 해당 파일은 모두 동일하다. 즉, 저장소에 커밋된 파일을 뜻한다.(committed)

3.2.3 수정됨(modified)

이미 커밋되었던 파일이 수정됐음을 뜻한다. 아직 추적되지 않은 파일도 수정된 파일로 볼 수 있지 않냐고 할 수 있지만 둘은 내부적으로 엄연히 다르다.

3.2.4 인덱싱됨(staged/indexed)

수정된 파일이 인덱스에 포함됐다면 *staged* 상태라고 볼 수 있다. *tracked*라는 것은 수정됨 그리고 수정되지 않음 상태에 놓인 파일 모두를 뜻하는데 *staged*와는 조금 다르다. 왜냐하면 *tracked*라는 것은 이미 커밋이 이루어진 상태이기 때문이다.

실제로 새로운 파일을 만들고 `git status` 명령으로 상태를 확인하면 *Untracked files*라는 항목에 포함되어 출력되지만, 이미 커밋됐던 파일을 수정한 뒤 상태를 확인하면 *Changes not staged for commit*이라고 나온다.

3.2.5 무시됨(ignored)

마지막으로 무시된 상태가 있는데, .gitignore 혹은 저장소 내 exclude에 설정되어 있는 패턴에 의해 파일이 무시될 수 있다. 파일이 무시되면 해당 파일은 작업 디렉토리 내에 있다고 해도 git의 영향권에서 벗어나게 된다.

3.3 스테이징

파일을 새로 만들거나 혹은 편집한 경우, 커밋하기 위해서는 반드시 staging 단계를 거쳐야 한다. staging은 파일을 인덱스에 추가시키는 단계로 커밋의 준비 단계로 생각하면 된다.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    index.html

nothing added to commit but untracked files present (use "git add" to track)
$ git add index.html
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   index.html
```

첫번째 `git status` 명령을 통해 index.html이라는 추적되지 않은 파일이 있으면 알았다. git이 파일을 관리하도록 하려면 *tracked* 상태로 만들어야 하는데 `git add` 명령으로 가능하다.

`git add` 명령으로 index.html을 인덱스에 추가한 뒤 다시 상태를 확인하니 **Changes to be committed** 항목에 포함된 것을 알 수 있다. 이제 index.html 파일은 *tracked* 상태이며 곧 *staged*된 것을 알 수 있다.

아직 커밋되지 않은 파일들이 인덱스에 포함됐는지 명시적으로 확인할 수 있는 방법이 있다.

```
$ git ls-files --stage
100644 1da9fabcc8ef195b1b8ceeccf3466fd6d8131476 0    index.html
```

'1da9fabcc8ef195b1b8ceeccf3466fd6d8131476'은 index.html의 SHA1 해시이다. 이제 우리는 index.html 파일이 인덱스 내에 있음을 확신할 수 있다. 이는 다른 말로 '파일이 준비되었다.'라고도 한다.

이제 index.html 파일을 커밋하지 않은 상태에서 다시 수정하고 `git add` 명령으로 인덱스에 추가해보자.

```
# index.html 파일을 수정했다. 즉, 콘텐츠가 변경되었다.
$ vim index.html
$ git add index.html
```

index.html 파일 해시가 변경되었음을 알 수 있다. 이렇게 된 이유는 index.html 파일의 콘텐츠가 변경되었기 때문이다.(blob의 해시는 파일의 콘텐츠로 만들어진다.)

그러므로 `git add` 명령은 '파일의 콘텐츠를 추가한다.'라고 이해할 수 있다.

참고

명령의 인수로 파일의 이름을 사용했지만 git은 내부적으로 파일의 이름을 이용하지 않는다. 단지, 저장해놓을 뿐이다. 실제로 이용되는 것은 파일의 내용이다.

3.4 커밋

Git에서 커밋하는 것은 생각보다 간단하다.

```
# 파일은 이미 인덱스에 추가된 상태다.
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   index.html

$ git commit -m "Initial commit"
[master (root-commit) 77f8e97] Initial commit
 1 file changed, 8 insertions(+)
 create mode 100644 index.html
```

파일이 이미 인덱스에서 준비된 상태이므로 git commit 명령을 실행하고 커밋 로그 메시지를 저장하기만 하면 된다. 예제에서처럼 -m 플래그(혹은 --message)로 인라인 커밋 로그 메시지를 전달해도 된다.

3.4.1 파일의 상태와 git commit 간의 상관 관계

커밋은 그 자체가 아닌 파일의 상태와 관련해서 혼동될 수 있다. 이번 섹션에서는 수정되지 않음(unmodified) 상태를 제외한 나머지 3가지 상태의 파일에 대해서 git commit 명령이 어떻게 동작하는지 알아보도록 하겠다.

```
# modified라는 파일을 만든다.
$ echo "I am 'modified'." >> modified
$ git add modified
# 실험을 위해 modified 파일을 인덱스에 추가한 뒤 커밋한다.
$ git commit -m "Initial commit"
[master (root-commit) 386f0c8] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 modified
# staged 파일을 만든다. 이 파일은 인덱스에 추가할 것이다. 하지만 커밋하지는 않는다.
$ echo "I am 'staged'" >> staged
# modified 파일은 수정한다.
$ vim modified
# untracked 파일을 만든다. 이 파일은 인덱스에 추가하지 않을 것이다.
$ echo "I am 'untracked'." >> untracked
# staged 파일을 인덱스에 추가한다.
$ git add staged
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   staged

Changes not staged for commit:
```

```
25    (use "git add <file>..." to update what will be committed)
26    (use "git checkout -- <file>..." to discard changes in working directory)
27
28      modified:   modified
29
30  Untracked files:
31    (use "git add <file>..." to include in what will be committed)
32
33      untracked
```

주석문을 읽으면 어떤 내용인지 알겠지만 간단히 요약하자면 이렇다.

modified 파일을 처음에 만들고 인덱스에 추가한 후 커밋까지 완료한 상태이다. 그런 뒤에, staged 파일을 만들고 인덱스에 추가한다. 하지만 커밋은 하지 않는다. modified 파일은 수정하지만 인덱스에 추가하지는 않는다. untracked 파일은 만든 상태에서 더 이상 진행하지 않는다.

결과적으로 세 개 파일은 이름 그대로 staged, modified(but not staged), untracked 상태이다.

```
$ git commit
```

위와 같이 옵션이나 인수 없이 그냥 커밋을 실행하면 커밋 메시지 에디터에 다음과 같이 출력될 것이다.

```
1   # Please enter the commit message for your changes. Lines starting
2   # with '#' will be ignored, and an empty message aborts the commit.
3   # On branch master
4   # Changes to be committed:
5   #       new file:   staged
6   #
7   # Changes not staged for commit:
8   #       modified:   modified
9   #
10  # Untracked files:
11  #       untracked
12  #
```

커밋의 대상이 되는 파일은 *staged* 상태인 staged 파일 뿐임을 알 수 있다. modified 파일은 *tracked* 상태이지만 *staged* 상태가 아니므로 커밋의 대상이 되지 못한다.

다시 `--all` 혹은 `-a` 플래그를 전달하여 커밋을 실행해보자.

```
1   # Please enter the commit message for your changes. Lines starting
2   # with '#' will be ignored, and an empty message aborts the commit.
3   # On branch master
4   # Changes to be committed:
5   #       modified:   modified
6   #       new file:   staged
7   #
```

```
8   # Untracked files:
9   #       untracked
10  #
```

앞에서는 커밋의 대상이 아니었던 modified 파일이 이제 커밋의 대상이 되었다. 그리고 untracked 파일은 여전히 커밋의 대상이 아니다.

커밋은 파일의 상태에 따라 조금 다르게 동작하는데 이는 Git이 파일의 상태를 어떻게 다루느냐와 관련되어 있으므로 중요하다.

`--all` 플래그(`-a`)는 수정되었지만(*unmodified* 그리고 *tracked*) 아직 *not staged*인 파일도 커밋 대상에 포함시키는 옵션을 가능하게 한다. 이것이 가능한 이유는 이미 저장소에 포함된 파일을 Git이 계속해서 추적(tracked)하기 때문이다.

3.4.2 원자적 커밋

참고

다음은 위키피디아의 Atomic commit을 번역한 것이다. 직접 번역한 내용이라 다소 어색한 표현이나 의역, 오역이 있을 수도 있다.

컴퓨터 과학의 영역에서, **원자적 커밋**이란 단일 동작에 있어 구별되는 변경사항의 세트가 적용되는 하나의 동작을 말한다. 변경사항들이 적용되고 나면, 원자적 커밋이 성공했다고 한다. 만약 실패한 경우, 원자적 커밋 내에서 완료된 모든 변경 사항들은 다시 되돌려진다. 이것이 가능케 하는 점은 시스템이 항상 안정적인 상태에 있다는 것을 확신시켜준다는 것이다. 분리에 대한 또 다른 중요한 점은 원자적 작업의 특징에서 비롯된다. 이러한 분리성은 한번에 단 한 개의 원자적 커밋만이 처리된다는 것을 보장한다. 원자적 커밋이 일반적으로 사용되는 곳은 데이터베이스 시스템 그리고 리비전 컨트롤 시스템들이다.

... 중간 생략 ...

리비전 컨트롤

... 중간 생략 ... 변경사항 세트의 데이터로 기존 데이터를 덮어쓰는(overwrite) 데이터베이스 시스템과는 다르게, 리비전 컨트롤 시스템들은 기존 데이터 내부로 변경사항 세트의 수정 사항들을 합쳐버린다(merge). 시스템이 merge를 완료할 수 없는 경우, 커밋은 거절된다. 실패된 merge가 리비전 컨트롤 소프트웨어에 의해 해결되지 못하는 상태라면 시스템은 변경사항들을 합치기 위해 사용자에게 요청할 것이다. 원자적 커밋을 지원하는 리비전 컨트롤 시스템들의 경우, 합치기에 있어서 실패라는 것은 실패된 커밋의 결과라고 볼 수 있다.

... 중간 생략 ...

원자적 커밋 관례

리비전 컨트롤 시스템을 사용할 때의 공통적인 관례는 작은 커밋을 사용하라는 것이다. 이러한 작은 커밋들은 때때로 원자적 커밋으로도 불리우는데, 그들이 (이상적으로) 시스템의 단일 양상에만 영향을 미치기 때문이다.

이러한 원자적 커밋들은 우리에게 좀 더 큰 이해도를 제공하고 적은 노력으로 변경사항을 되돌릴 수 있게 하며, 그리고 좀 더 쉬운 버그 발견을 가능케 한다.

좀 더 큰 이해도는 작은 크기와 커밋의 주요 특성(어떤 변경 사항인지를 나타내는)에서 온다. 한 가지 종류의 변경 사항에 대해서 알아볼 수 있다고 한다면 변경사항들을 통해 무엇이 변경되었고 왜 변경했는지를 좀 더 쉽게 이해할 수 있을 것이다. 이러한 점은 소스 코드의 형태적 변화가 이루어질 때 특히 중요해진다. 형태와 기능적 변경사항이 혼합되어버리면 유용한 변경 사항을 확인하는 것이 매우 어려울 것이다. 예를 들어, 파일 내에서 들여쓰기 규칙이 탭에서 3개의 스페이스로 변경된다면 해당 파일 내의 모든 탭들은 변경된 것으로 보일 것이다. 이로 인해 코드 리뷰어들이 기능적 변경사항을 간단하게 살펴보는 것이 쉽지 않다면 이것은 매우 치명적이다.

오직 원자적 커밋만이 있다고 한다면 오류가 발생한 커밋을 확인하는 것이 매우 쉬워질 것이다. 오류가 발생했다면 모든 커밋을 확인할 필요없이 관련된 커밋만을 확인하면 된다. 오류를 되돌려야 할 경우에도 원자적 커밋은 일을 수월하게 만들어준다. 이후의 변경사항들과 통합하기 이전에 문제가 되는 리비전을 수동으로 되돌리고 삭제하는 대신에 커밋 내에서 변경사항을 쉽게 되돌릴 수 있다. 또한 이것은 같은 커밋 내에서 문제와 관련없는 변경사항들이 뜻하지 않게 삭제되는 위험 부담을 줄여줄 것이다.

원자적 커밋은 또한, 커밋 한번에 단일 버그 픽스 한 개만 들어 있다면 버그 픽스에 대한 리뷰를 좀 더 쉽게 만들어 줄 것이다. 리뷰어가 관련없는 파일들을 희미한 가능성만 보고 확인하는 대신에 버그가 수정된 내용과 직접 맞닿아 있는 변경사항들과 파일만을 확인하도록 해줄 것이다. 또한 이것이 의미하는 바는, 버그 픽스가 버그를 수정한 변경사항이 포함되어 있는 커밋만을 테스트를 위해 쉽게 패키징할 수 있다는 것이다.

... 이후 생략 ...

위 내용을 아주 짧게 요약하자면 다음과 같다.

우리가 프로그래밍 등의 수단을 이용해 만들어내려는 모든 것을 애플리케이션이라고 지칭한다면, 이 애플리케이션들은 어떤 feature들의 집합이라고 할 수 있을 것이다. 각 feature는 독립적일 수도 있고 서로 유기적일 수도 있다. 어쨌거나 이러한 feature들은 하위 feature가 모아져서 이루어질 가능성이 크다. 또 하위 feature들은 그것의 하위 feature들로 이루어져 있을 수 있다. 이렇게 하위 수준까지 도달할 경우 코드 수준으로 이야기할 수 있는 하위 feature 수준까지 도달할 수 있을 것이다.

이렇게 더 이상 쪼갤 수 없는 수준까지 feature를 쪼개서 구현하고 커밋하는 것을 원자적 커밋(혹은 원자적 변경 사항, atomic changeset)이라고 한다. 이러한 원자적 커밋을 이용할 경우, 커밋만 재배치 혹은 수정/삭제해도 상위 feature가 변하게 되므로 feature에 대한 기능 변경이나 삭제가 매우 간편해질 수 있다. 또 어떤 부분에서 문제가 있었는지 알아내기 쉬워진다.

원자적 커밋은 강제되는 것이 아니라 어디까지나 개발자 혹은 구성된 팀의 의지와 관련되어 있다. 조직 내에서의 규범이며 약속일 뿐이다. 그것을 이용함으로써 얻는 이득이나 혹은 거부감 모두 그들의 몫이다. 그러나 현대의 개발 트렌드가 그렇게 흘러가고 있음은 부정할 수 없다.

3.5 파일 삭제하기

Git이 추적하는 파일을 삭제하려면 `git rm` 명령을 사용해야 한다. `git rm` 명령은 작업 디렉토리와 인덱스로부터 파일을 제거하는 것이 목표다.

3.5.1 추적하지 않은 파일은 지우지 못한다.

untracked 상태인 untracked 라는 파일이 있다고 가정하자.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    untracked

nothing added to commit but untracked files present (use "git add" to track)
$ git rm untracked
fatal: pathspec 'untracked' did not match any files
```

untracked 상태인 파일을 지우려고 하자, untracked라는 pathspec과 일치하는 파일이 없다는 메시지를 출력한다. 이 파일은 작업 디렉토리에는 존재하는 파일이다. 하지만 Git의 입장에서는 아직 추적전이기 때문에 없는 파일로 간주한다.

만약 정말 쓸모없는 파일이어서 지워야 한다면, 일반적인 rm 명령으로 지우면 된다.

3.5.2 인덱스에 추가된 파일을 인덱스에서 지우기

untracked 파일을 인덱스에 추가한 뒤 잘못 추가했다고 판단이 들었다. 어떻게 해야할까?

```
$ git add untracked
# untracked 파일이 잘못 추가되었음을 알았다!
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   untracked
# untracked 파일을 --cached 플래그를 이용해 인덱스에서 제거한다.
$ git rm --cached untracked
rm 'untracked'
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    untracked

```

```
19    nothing added to commit but untracked files present (use "git add" to track)
```

--cached 플래그를 전달하면 인수로 전달된 파일을 인덱스에서 제거한다.

3.5.3 인덱스와 작업 디렉토리에서 지우기

만약 인덱스에 추가한 뒤에 --cached 플래그를 전달하지 않고 그냥 git rm 명령을 실행하면 어떻게 될까?

```
1    $ git rm untracked
2    error: the following file has changes staged in the index:
3        untracked
4    (use --cached to keep the file, or -f to force removal)
```

파일이 이미 staged 상태임을 알려준다. 그리고 파일을 작업 디렉토리에 유지한 상태로 지우려면 --cached 플래그를, 무조건 지우려면 -f 플래그(--force)를 사용하라고 한다. -f 플래그는 인덱스에서 파일을 지우면서 작업 디렉토리에서도 제거한다.

3.5.4 커밋한 파일 지우기

이미 커밋된 파일을 지우려고 한다.

```
1    $ git rm modified
2    rm 'modified'
3    $ git status
4    On branch master
5    Changes to be committed:
6      (use "git reset HEAD <file>..." to unstage)
7
8        deleted:    modified
9
10   $ ls -al
11   total 0
12   drwxr-xr-x   3 devcken  staff  102  3 20 02:50 .
13   drwxr-xr-x  18 devcken  staff  612  3 19 11:47 ..
14   drwxr-xr-x  13 devcken  staff  442  3 20 02:50 .git
```

플래그 없이 git rm 명령을 실행하면 커밋된 파일, 즉 이미 저장소로 들어간 파일을 삭제할 수 있다.

하지만, 주의해야 할 것이 있다. 이는 인덱스의 내용을 변경하는 것이지 커밋을 변경하는 것이 아니다.
인덱스의 내용 중 해당 파일을 삭제하는 변경을 가하는 것이므로 커밋해야 최종적으로 저장소에 반영이 된다.

ls 명령을 실행해보면 커밋 전에 작업 디렉토리에서는 이미 파일이 삭제된 것을 알 수 있다. 만약, 실수로 삭제한 것이라면 되돌릴 수 있어야 한다. 이 때는 파일을 HEAD로부터 체크아웃해야 한다.

```
$ git checkout -- modified
```

3.6 파일 이름 변경하기

mv 명령을 사용하면(Windows의 경우 move), 파일을 다른 디렉토리로 이동시키거나 이름을 변경할 수 있다. git에는 이러한 이동 혹은 이름 변경을 위한 비슷한 명령이 있다.

```
$ echo "I love winnie the Pooh." >> beer
$ git add .
$ git commit -m "poooooooooooh"
[master (root-commit) 1985681] poooooooooooh
 1 file changed, 1 insertion(+)
 create mode 100644 beer
```

파일 이름을 bear로 했어야 했는데 beer로 짓는 비극이 벌어지고 말았다. 이런 상황에서는 여러 가지 대처 방안이 있지만 직관적인 방법을 사용해보자.

```
$ git mv beer bear
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    beer -> bear

$ git commit -m "rename."
[master 18f01a7] rename.
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename beer => bear (100%)
$ git ls-files --stage
100644 4f4440388897492fe6744aa614f7e4c311e59064 0    bear
```

`git mv` 명령으로 파일의 이름을 변경했고 커밋하여 변경 사항을 저장했다. 확인 결과, 잘 변경된 것을 확인했다.

`git log` 명령에 파일 이름을 전달하면 해당 파일에 대한 이력만을 간추려 볼 수 있다.

```
$ git log bear
commit 18f01a708fcfaeae11b35c02ddc418ee900ecdcb
Author: Choi Leejun <devcken@gmail.com>
Date:   Fri Mar 20 23:12:16 2015 +0900

    rename.
```

그런데 아쉽게도 이름이 변경되기 전 이력은 출력되지 않는다. 이 때는 `--follow` 플래그를 전달하여 실행하면 이름이 변경되기 전 이력까지 함께 보여준다.

```
$ git log --follow bear
commit 18f01a708fcfaeae11b35c02ddc418ee900ecdcb
Author: Choi Leejun <devcken@gmail.com>
Date:   Fri Mar 20 23:12:16 2015 +0900

    rename.

commit 19856816a6e074bcc02e5bf8ffc672944c9077e7
Author: Choi Leejun <devcken@gmail.com>
Date:   Fri Mar 20 23:09:37 2015 +0900

    pooooooooooh
```

참고

만약 파일의 경로만 수정된다면 어떻게 될까? 파일의 경로 또한 파일 이름의 한 부분이므로 해당 경로 내에 있는 파일 모두에 영향을 준다. 하지만 파일들의 컨텐츠가 변경된 것은 아니다. 경로를 수정하기 전에 blob ID를 확인하고 경로 수정 후 blob ID를 확인해보자.

3.6.1 이름 변경 추적

SVN의 경우, 이름 변경 추적을 좀 더 쉽게 처리하기 위해 svn mv 명령을 강제한다.

Git은 이 문제를 굉장히 간단하게 처리한다. 파일의 이름이라는 것은 우리가 경로라고 부르는 파일까지 도달하기 위한 계층 구조의 이름과 실제 파일을 특정하는 이름의 조합이다. Git은 이 두 조합을 트리객체에 저장하는데 파일 이름이 변경되게 되면(혹은 이동하게 되면) 파일에 대한 트리 객체만 변경한다.

그러므로 파일의 컨텐츠를 보관하는 blob에는 변화가 생기지 않아서 자원과 성능의 측면에서 매우 효율적이다.

3.7 파일 무시하기

프로젝트를 진행하다보면 같은 파일이지만 사람마다 약간씩 다른 내용을 가진 파일들이 존재한다. 예를 들어, 존은 WebStorm으로 개발하고 피터는 Sublime Text로 개발한다고 가정하자. WebStorm의 경우 .idea라는 프로젝트용 디렉토리를 생성하고 유지하는데 이는 피터에게 필요없는 것이다. 더구나 WebStorm을 사용하는 다른 개발자가 있다고 가정한다면 서로의 디렉토리 내 파일들은 충돌을 일으킬 수도 있다.

이러한 문제는 git을 사용하든 SVN을 사용하든 부딪힐 수 있다. 그래서 이를 위한 해결책이 필요한데 그 해결책을 git에서는 **gitignore**라고 표현한다.

gitignore는 *gitconfig*와 마찬가지로 여러 계층에 걸쳐 설정을 가지며 그에 따른 우선 순위가 존재한다.

1. CLI 상에서 명령어와 함께 입력된 파일 패턴

사용자가 입력한 명령어에 전달되는 파일의 패턴이 가장 우선시 된다. 어떤 파일 패턴을 제외하는 목적일 경우 가장 최상위로 처리된다.

2. 파일이 존재하는 디렉토리 내 .gitignore에 설정된 패턴

어떤 파일이 존재하는 디렉토리 내에 .gitignore 파일이 있다면 거기에 설정된 패턴이 적용된다.

3. 상위 디렉토리의 .gitignore에 설정된 패턴

만약 어떤 파일이 존재하는 디렉토리 내에 .gitignore 파일이 없다면 상위 디렉토리를 재귀적으로 탐색하여 .gitignore 파일을 찾는다. 존재한다면 가장 가까운 경로의 .gitignore 파일의 패턴이 적용된다.

4. .git/info/exclude 파일의 패턴

.gitignore의 경우 일반 파일로 취급되어 다른 사람들과 원격 저장소를 통해 공유될 수 있으므로 공통으로 제외되어야 하는 패턴만 설정되어야 한다. 만약 자신의 저장소에서만 제외되어야 하는 패턴이 있다면 exclude 파일에 패턴을 적용하면 된다. 이것은 다른 사람들에게 공유되지 않는다.

5. 환경 설정 변수인 core.excludefile에 지정된 파일에 들어 있는 패턴

이 환경 설정에는 파일들의 경로가 설정되는데 해당 파일들에는 gitignore의 패턴 규칙이 설정된다. 이 파일들을 어느 위치에 놓느냐에 따라서 2~3번처럼 다른 사람들과 공유할 수 있거나 4번처럼 공유하지 않을 수 있다.

참고

gitignore를 추가하거나 삭제하는 명령어가 있을 거 같지만 없다. 써드파티에서 제작된 관련 쉘이 있긴 하지만 굳이 명령어 기반으로 작성할 필요가 없다.

gitignore가 제공하는 패턴에는 몇 가지 규칙이 있다.

- 빈줄은 무시된다.
- 파운드 기호(#) 이후부터는 주석으로 간주된다. 파운드 기호 앞은 그렇지 않다.
- 일반적인 문자열은 파일 이름에 적용된다.
- 디렉토리 이름 뒤에는 슬래쉬(/)를 붙인다. 이러한 패턴은 파일이나 심볼릭 링크에는 적용되지 않는다.

Unix shell glob 패턴으로 광범위하게 디렉토리와 파일들을 적용시킬 수 있다.

참고

Unix shell glob이 어렵다고 느껴지는가? 걱정마라. 기껏해야 *, **를 가장 많이 쓰고, ?, !을 조금 응용하는 정도다.

.gitignore 파일을 만들어 주는 플러그인이나 서비스들도 존재한다. 인기있는 IDE(이클립스나 JetBrains계열, 그리고 VS 등등)에는 이미 built-in되어 있거나 플러그인들이 써드파티로 제공되고 있으며, 온라인 상에서 만들어주는 서비스들도 많다. 그 중 가장 유명한 것은 gitignore.io다.

이들 플러그인이나 서비스들은 플랫폼별, 환경별, 언어별로 거의 모든 환경을 지원하고 있어 gitignore 패턴을 직접 다룰 일이 없을 정도다. 하지만 프로젝트가 커지고 구성원이 늘어남에 따라 예외도 늘어나므로 꼭 알아두는 것이 좋다.

4. Git 객체 모델

이 장에서는 Git 객체가 커밋 내에서 어떤 역할을 하는지 그리고 커밋을 어떻게 다루는지 설명한다.

4.1 객체 모델

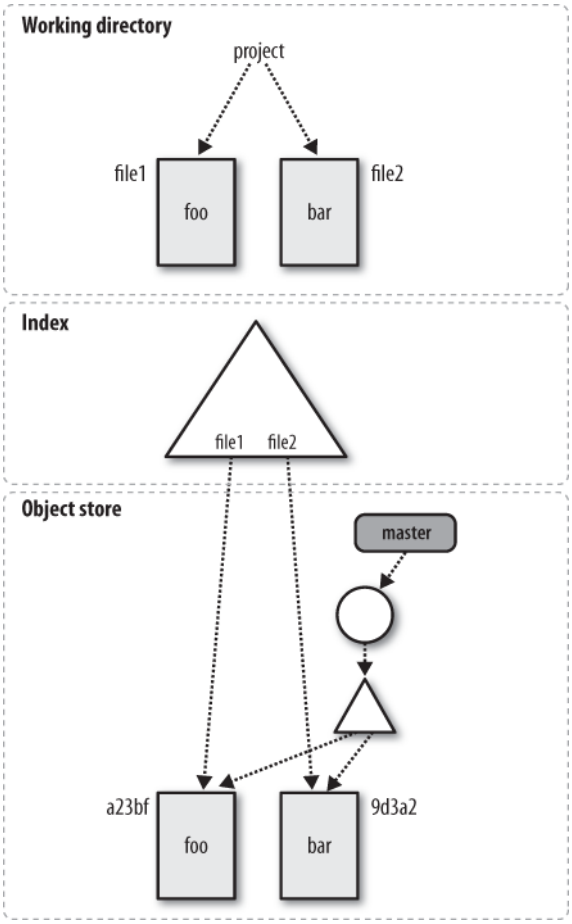

그림 4-1 Git의 3가지 공간과 Git 객체

위 그림을 보면 최초의 커밋이 이루어진 직후의 상태임을 알 수 있다. 'foo'와 'bar'라는 컨텐츠를 가진 두 개의 파일이 작업 디렉토리에 있고, 인덱스에도 두 개 파일에 대한 정보가 존재한다.

객체 저장소에는 한 개의 커밋 그리고 그 커밋이 가리키는 트리 객체, 그리고 트리 객체가 가리키고 있는 두 개의 blob 객체가 존재한다. 각각의 blob은 인덱스에서도 참조하고 있다.

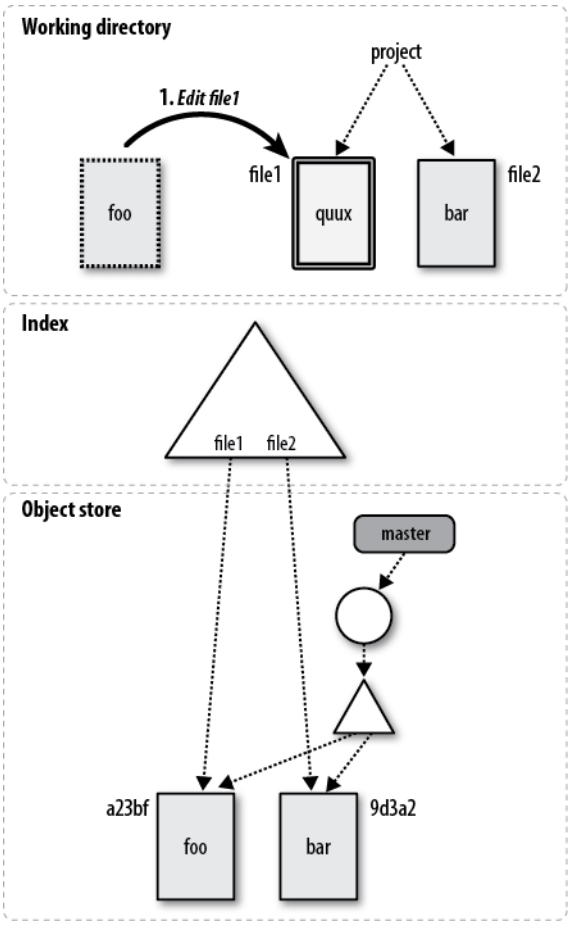

그림 4-2 파일의 수정

file1의 내용이 foo에서 quux로 변경되었다. 아직은 작업 디렉토리에서만 변경된 상태이기 때문에 인덱스나 객체 저장소에는 변화가 일어나지 않았다.

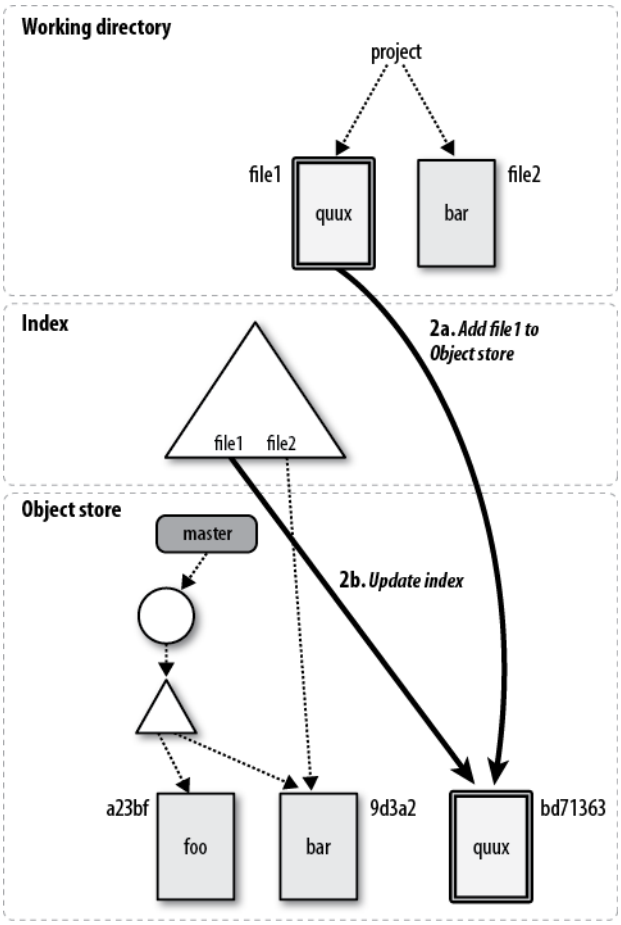

그림 4-3 스테이징

`git add`를 통해 file1의 변경 사항을 준비 상태로 만든 직후의 모습이다.

변경된 file1에 대한 blob 'bd71363'이 추가된 것을 알 수 있다. 커밋 트리 객체는 여전히 이전 blob을 가리키고 있지만 인덱스의 file1은 'bd71363'을 가리키고 있다.

file2의 변경사항은 없었기 때문에 여전히 file2에 대한 blob(콘텐츠)에는 변화가 없다.

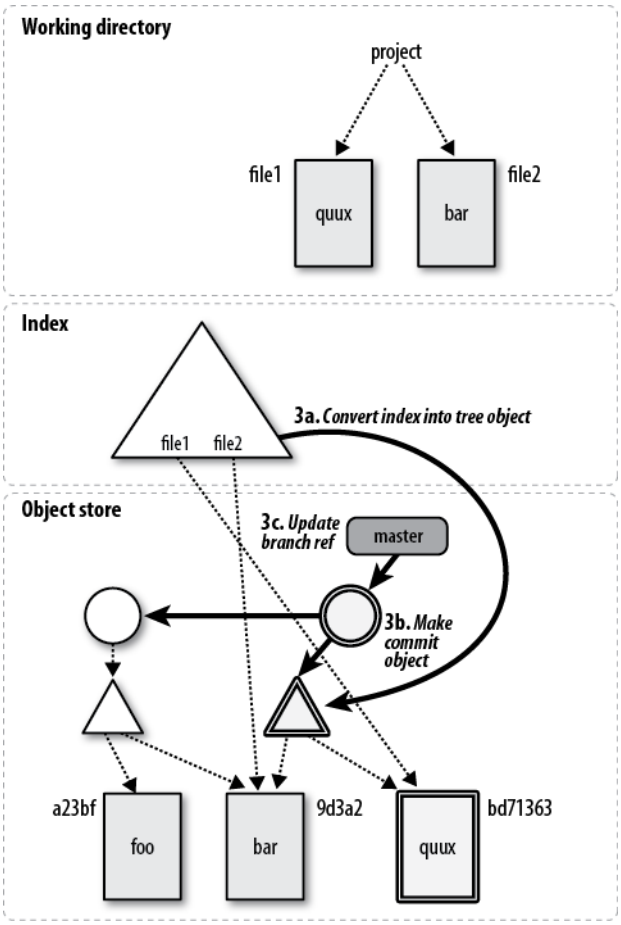

그림 4-4 커밋

`git commit`을 통해 준비된 변경 사항을 객체 저장소에 저장한 직후다.

file1에 대한 새로운 SHA1 'bd71363'을 가리키기 위해 새로운 커밋과 트리 객체가 생성되었다. 이 과정에서 인덱스를 이용해 트리객체가 유사하게 만들어지며 이 트리 객체를 가리키는 커밋 객체는 동시에 이전 커밋을 가리키게 된다. 이 이전 커밋을 상위 커밋이라고 한다.

새로 만들어진 트리 객체는 file2가 변경 사항이 없으므로 file2에 대한 blob을 가리키게 된다.

이렇게 변경 사항이 반영된 인덱스를 다음 커밋의 트리에 반영하는 것을 '**스냅샷(snapshot)**을 기록한다.'라고 한다.

'master' 브랜치는 이전 커밋에서 옮겨와 새로운 커밋을 가리키게 된다.

4.2 커밋 식별하기

Git을 통해 무슨 작업을 하든 커밋을 식별하는 것은 필히 거쳐야 하는 단계이다. 브랜치를 만들기 위해서는 커밋을 분기점으로 해야 하고, 변경 사항을 비교하기 위해서는 커밋 구간을 지정하기도 하는 등 커밋은 주요 작업들의 기준이 된다.

4.2.1 절대적 커밋 이름을 통한 식별

커밋이 만들어질 때, Git은 커밋에 40자리의 **SHA1 해시값**을 만들어 저장한다. 이것을 우리는 **커밋 해시** 혹은 **커밋 ID**라고 부른다. 이 커밋 해시는 항상 고유한 값을 나타내는데 그래서 하나의 커밋만을 참조한다.

```
$ git log 19856816a6e074bcc02e5bf8ffc672944c9077e7
commit 19856816a6e074bcc02e5bf8ffc672944c9077e7
Author: Choi Leejun <devcken@gmail.com>
Date:   Fri Mar 20 23:09:37 2015 +0900

    poooooooooooh
$ git log 1985681
commit 19856816a6e074bcc02e5bf8ffc672944c9077e7
Author: Choi Leejun <devcken@gmail.com>
Date:   Fri Mar 20 23:09:37 2015 +0900

    poooooooooooh
```

커밋 해시는 40자리의 SHA1 해시값이므로 사용하는데 불편함이 뒤따르는 것이 사실이다. 그래서 Git은 SHA1 해시값의 접두사를 이용해 커밋을 지정하는 것을 지원한다.

보통 SHA1 해시값 앞 7자리(혹은 6자리)를 접두사로 사용한다.

4.2.2 심볼릭 참조

심볼릭 참조는 저장소 내 .git/refs 디렉토리에 계층적으로 저장되는 Git 객체에 대한 간접 참조이다.

refs/로 시작되는 명시적인 전체 이름을 가지고 있는데, 로컬 브랜치를 나타내는 refs/heads/ref, 원격 추적 브랜치용을 나타내는 refs/remotes/ref 그리고 태그용으로 사용되는 refs/tags/ref라는 세 가지 네임스페이스가 존재한다.

예를 들어, 우리가 흔히 아는 master 브랜치는 실제로 refs/heads/master라는 이름을 줄여 사용하는 것이다. 원격 저장소에 있는 origin/master 브랜치(원격 저장소 이름이 origin인 경우)는 refs/remotes/origin/master라는 실제 이름을 갖고 있다.

4.2.3 상대적 커밋 이름

Git에는 커밋 해쉬나 참조를 기준으로 상대적인 위치에 있는 커밋을 식별하는 매커니즘이 있다.

A. 캐럿(caret, ^)

캐럿은 동일 차수 내에서 각기 다른 상위 커밋을 선택할 때 사용된다. 커밋 중에는 위 그림처럼 1개 이상의 상위 커밋을 가지는 커밋도 존재하는데 이런 경우 상위 커밋을 선택할 때 캐럿을 이용한다.

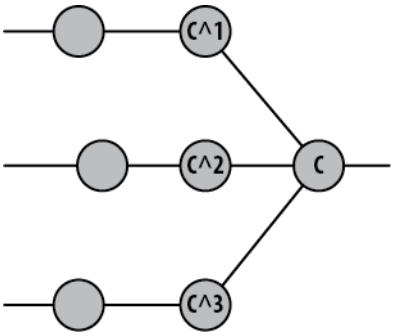

그림 4-5 캐럿을 이용한 커밋 참조

B. 틸데(tilde, ~)

틸데는 기준 커밋으로부터 단계적인 상위 차수의 커밋 참조를 가능하게 해준다. C~1은 C로부터 한 단계 위 커밋 그리고 C~2는 C로부터 두 단계 위 커밋을 참조한다.

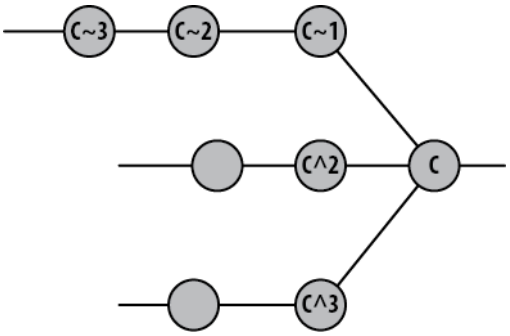

그림 4-6 틸데를 이용한 커밋 참조

캐럿과 틸데를 혼합하여 다소 복잡한 참조도 가능하다.

참고

한 가지 기억해야 할 것은 상대적 참조는 어디까지나 상대적이라는 것이다. 상대적 참조의 기준이 되는 커밋이 어떤 커밋인지에 따라 참조되는 커밋도 달라진다는 것이다. 혹은 같은 기준 커밋일지라도 상위 커밋에 변화가 생기면 상대적인 참조도 변할 수 있다.

4.2.4 특수 심볼릭 참조

Git에는 특별한 목적을 위해 정의되어 있는 특수 심볼릭 참조가 있다.

- *HEAD*
  - HEAD 참조는 항상 현재 브랜치의 끝(tip)을 참조한다. 즉 해당 브랜치 내에서 최신의 커밋을 참조한다.
  - 브랜치를 옮겨가면 이 HEAD 참조가 업데이트되면서 해당 브랜치의 최신 커밋을 참조하게 된다.
- *ORIG_HEAD*
  - 병합이나 커밋을 조정하는 작업에서 HEAD를 새 값으로 조정하기 이전에 ORIG_HEAD에 조정해둔다.
  - 이 참조는 이전 상태로 되돌리거나 이전 상태를 비교할 때 사용할 수 있다.
- *FETCH_HEAD*
  - `git fetch` 명령이 실행되면 원격 저장소로부터 가져온 모든 브랜치의 헤드를 `.git/FETCH_HEAD` 파일에 기록하게 된다. 즉, FETCH_HEAD는 원격 저장소로부터 가져온 브랜치의 HEAD를 의미한다.
  - `git fetch` 이후에만 유효한데, 페치 작업 직후에 가져온 브랜치의 최신 커밋을 참조할 때 유용하다.
- *MERGE_HEAD*
  - 병합 작업이 이루어지면 병합되는 브랜치의 헤더가 MERGE_HEAD에 기록된다.
  - 심볼릭 참조의 측면에서 봤을 때, 병합이란 MERGE_HEAD가 HEAD로 병합되는 것이다.

4.3 커밋 히스토리를 통해 되돌아보기

커밋 히스토리는 저장소 내에서 무슨 일이 있었는지 알려주는 중요한 단서다. 이 히스토리를 이용해 무슨 일이 있었는지를 알아내 앞으로의 작업들을 좀 더 현명하게 해낼 수 있다는 것이다.

참고

여기서 알려주는 조회 방법은 주로 사용되는 대표적인 방법 몇 가지일뿐이다. 이외에도 엄청나게 많은 방법들이 있고 그들의 조합도 가능하므로 필요한 방법이 있다면 git-log reference를 살펴보자.

커밋 히스토리를 출력하기 위한 명령은 `git log`다.

```
$ git log
commit 2380028ec4a6a77401b867a51de26a3cb8e8d311
Author: Oleg Gaidarenko <markelog@gmail.com>
Date:   Tue Feb 17 10:09:54 2015 +0300

    Core: change jQuery.each and jQuery#each signatures

    Fixes gh-2090
    Closes gh-2097

commit a4715f4216ace92fba6991106053415e66289686
Author: Oleg Gaidarenko <markelog@gmail.com>
Date:   Tue Feb 17 10:05:25 2015 +0300

    Ajax: remove use of jQuery#each second argument

    Ref gh-2090

commit 8356281bed643bb3d56ad02f52580a0e20dc0237
Author: Oleg Gaidarenko <markelog@gmail.com>
:
```

위 커밋 히스토리는 jquery의 커밋 히스토리다. `git log` 명령을 실행하고 나면 Git은 커밋 히스토리를 화면에 출력할 수 있을 만큼 출력하고 나서, `:`을 출력한 상태에서 사용자 입력을 기다린다.

커밋 히스토리는 저장소에서 작업이 진행됨에 따라 무제한적으로 늘어날 수 있기 때문에 사용자의 입력을 통해서 중단할지 아니면 계속해서 보여줄지를 결정한다.

q(uit)를 누르면 `git log` 명령은 중지된다. 아래 방향의 화살표키를 누르면 과거 히스토리를 계속해서 열람할 수 있고 위 방향의 화살표키를 누르면 좀 더 최근의 히스토리를 다시 볼 수 있다.

쉼표로 구분되는 선택자 간 공백은 제거한다.

4.3.1 브랜치의 커밋 히스토리 조회하기

```
$ git log origin/standard-then
commit b677541bc4a6d48f96fc7d1351ce6d72c5055bdc
Author: jaubourg <j@ubourg.net>
Date:   Wed Sep 10 16:59:45 2014 +0200

    Deferred: produce and consume standard thenables

    Deferred.then is now aligned with the emerging standard. Objects that implem
    the thenable interface will also be properly consumed.

    We use Q to test interoperability.

    Fixes #14510

commit b807aedb7fee321fb3aa5d156a5a256ab0634e2d
Author: Daniel Herman <daniel.c.herman@gmail.com>
Date:   Thu May 15 12:26:20 2014 -0400

    Event: Restore the `constructor` property on jQuery.Event prototype

:
```

`git log` 명령에 브랜치 이름을 전달하면 해당 브랜치에 대한 정보만 볼 수 있다. 예제에서는 jquery 저장소에서 원격 저장소 브랜치인 origin/standard-then의 커밋 히스토리를 조회한 것이다.

4.3.2 커밋 해시와 참조를 이용해 히스토리 조회를 제한하기

`git log` 명령은 광범위한 커밋 히스토리를 다루는 명령이기 때문에 범위를 두어 조회하는 것이 좋다. 그 중에서도 커밋 해시와 참조를 그 범위를 제한할 수 있다.

```
$ git log HEAD~4..HEAD~2
commit 8356281bed643bb3d56ad02f52580a0e20dc0237
Author: Oleg Gaidarenko <markelog@gmail.com>
Date:   Tue Feb 17 10:45:30 2015 +0300

    Tests: make top of the HTML suite compliant with style guide

    See http://contribute.jquery.org/style-guide/html/

    Closes gh-2098

commit 9d1b989f20b550af3590691723b0620f6914626e
Author: Oleg Gaidarenko <markelog@gmail.com>
Date:   Sun Feb 15 05:41:38 2015 +0300

    Ajax: remove deprecated extensions from ajax promise

    Fixes gh-2084
    Closes gh-2092
```

HEAD~4가 HEAD~2보다 먼저 일어났으므로 from..to의 형태로 조회했다. 물론 그 역순으로 조회하는 것도 가능하다.

참고

커밋 해시와 참조 등에 대한 내용은 4.2 커밋 식별하기를 참고하기 바란다.

4.3.3 커밋 히스토리의 개수를 제한하여 보기

`git log` 명령은 범위 뿐만 아니라 개수를 제한할 수도 있는데 약간 특이한 플래그 옵션을 사용한다.

`-n` 플래그인데 여기서 n은 영문자 'n'이 아니라 1 이상의 정수를 말한다.

```
$ git log -2
commit 2380028ec4a6a77401b867a51de26a3cb8e8d311
Author: Oleg Gaidarenko <markelog@gmail.com>
Date:   Tue Feb 17 10:09:54 2015 +0300

    Core: change jQuery.each and jQuery#each signatures

    Fixes gh-2090
    Closes gh-2097

commit a4715f4216ace92fba6991106053415e66289686
Author: Oleg Gaidarenko <markelog@gmail.com>
Date:   Tue Feb 17 10:05:25 2015 +0300

    Ajax: remove use of jQuery#each second argument

    Ref gh-2090
```

위 예제에서처럼 하이픈(-)과 함께 숫자를 전달하면 해당 숫자를 최대 커밋 수로 받아들여 제한한다.

4.3.4 날짜 범위로 제한하기

커밋을 날짜 범위로 제한하여 보고싶은 경우도 있을 것이다. 언제부터 언제까지의 범위로 제한하는 것이 가능하다.

```
$ git log --since="3 months ago" --until="2 months ago"
commit e905dcd8f33e99275e29b0333e7b255559197c81
Author: Timmy Willison <timmywillisn@gmail.com>
Date:   Mon Jan 19 12:02:13 2015 -0500

    Release: update AUTHORS.txt

commit 4116914dcac32868c2c822366785e3dd2ccc69d4
Author: Timo Tijhof <krinklemail@gmail.com>
```

```
Date:   Wed Jan 7 20:12:49 2015 +0000

    Core: Return empty array instead of null for parseHTML("")

    Fixes gh-1997
    Close gh-1998

commit d7e5fcee519e5f3e840beef9e67a536e75133df9
Author: Dave Methvin <dave.methvin@gmail.com>
Date:   Wed Jan 14 14:34:09 2015 -0500

    Event: HTML5 drop events inherit from MouseEvent

    Fixes gh-2009
    Ref gh-1925
:
```

`--since` 플래그는 `--after` 플래그와 같은 의미로 ..부터를 의미한다. 뒤에 오는 값은 yyyy-MM-dd hh:mm:ss와 같은 날짜 형식(시간은 빼도 된다.)과 상대적 시간이 가능하다. 가령 현재부터 1시간 이전은 1 hours ago 같은 식이다.

반대로 ..까지의 의미를 가진 플래그는 --until과 --before 플래그이다.

4.4 커밋 그래프

Git 객체 모델을 통해 상위 커밋과 하위 커밋의 관계를 그림으로 나타내곤 한다. (4.1 Git 객체 모델을 참고하자.)

하지만, 커밋 그래프를 나타내기 위해 매번 모든 커밋 객체와 연결되어 있는 트리 객체 및 blob 객체를 그려내는 것은 거의 불가능하며 효율적이지도 않다.

그래서 Git에서는 커밋 그래프(커밋 히스토리)를 나타내기 위해서 DAG(directed acyclic graph, 방향성 비순화 그래프)를 사용한다.

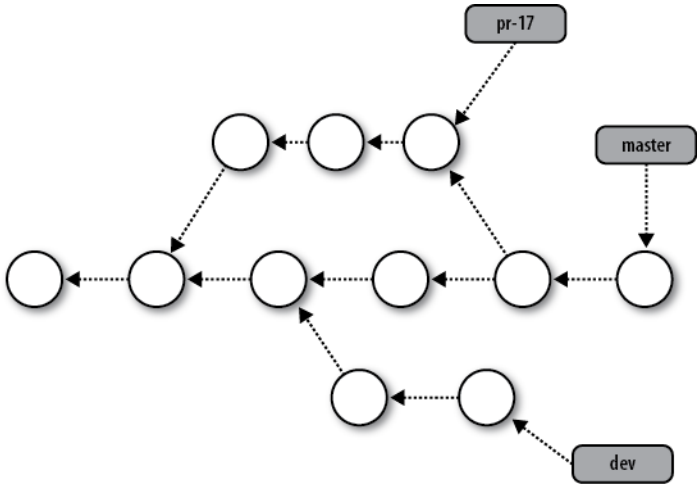

그림 4-7 방향성 비순환 그래프

DAG는 모든 선이 한 노드에서 다른 한 노드로 방향성을 가지고 연결되며(directed) 처음 한 노드에서 시작해 선을 따라 다른 노드로 이동해도 처음의 노드로 돌아갈 수 없는 그래프를 말한다.(그래프라는 것은 노드와 선(에지)의 연결로 이루어진다.)

이러한 그래프에서 시간의 흐름은 왼쪽에서 오른쪽으로 흐른다. 그러나 Git에서 커밋의 시간은 중요한 역할을 하지 않는다. 다음 그래프를 보자.

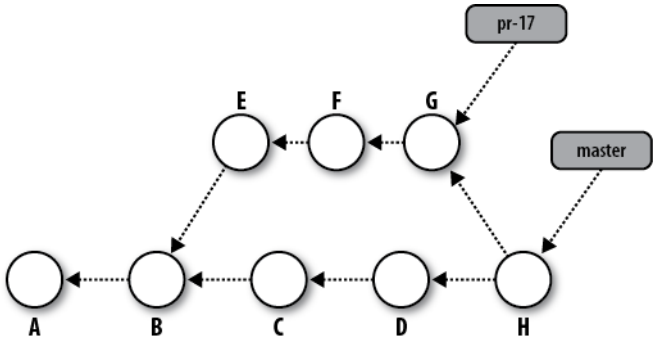

그림 4-8 브랜치와 병합

커밋 B로부터 두 개의 커밋 E와 C가 생겨났다. 두 커밋은 시간의 흐름 상 어떤 커밋이 먼저 생겨났는지 알 수가 없다. B, E, C 커밋 간의 관계 상 알 수 있는 것은 커밋 B가 생긴 뒤에 커밋 E와 C가 생겼다는 것이다.

커밋 C와 커밋 E는 커밋 B로부터 생겨났는데 이런 경우 커밋 B를 **base commit(기원 커밋)**라고 한다.

커밋 H는 커밋 G와 D를 상위 커밋으로 가지고 있는데 pr-17 브랜치가 master 브랜치로 병합된 것이다. 즉 커밋 H는 **merge commit(병합 커밋)**이다.

병합 이전에 master 브랜치는 커밋 D를 참조하고 있었을텐데 병합이 발생하면서 병합 커밋인 커밋 H를 참조하게 된다. 그리고 pr-17 브랜치는 병합 이후에도 커밋 G를 참조한다.

4.4.1 커밋 그래프를 확인하는 방법

커밋 히스토리를 커밋 그래프로 봐야 커밋 히스토리에 대한 이해가 빠를 것이다. 그런 목적을 위한 도구들이 꽤 많이 나와 있다.

Github나 BitBucket의 경우, 커밋 히스토리를 커밋 그래프로 보여주는 기능이 있으나 이는 어디까지나 원격 저장소의 커밋 히스토리를 보여주는 것이다. 즉, 원격 저장소로 push한 내용에 대해서만 볼 수 있는 것이다.

로컬에서 커밋 히스토리를 확인하기 위한 많은 도구들이 제공되고 있지만 Git이 제공하는 `git log` 명령의 `--graph` 플래그로 사용하는 것을 가장 추천한다.

```
$ git log --graph --all
*   commit 5573b48fe313ee0b9e45895958d373fcdba7f18a
|\  Merge: 6ef53d5 03f15a7
| | Author: Junio C Hamano <gitster@pobox.com>
| | Date:   Fri Mar 20 15:14:44 2015 -0700
| |
| |     Merge branch 'tg/fix-check-order-with-split-index' into pu
| |
| |     * tg/fix-check-order-with-split-index:
| |       read-cache: fix reading of split index
| |
| * commit 03f15a79a9006d819a25fe04199f9302ad4aaea0
| | Author: Thomas Gummerer <t.gummerer@gmail.com>
| | Date:   Fri Mar 20 22:43:14 2015 +0100
| |
| |     read-cache: fix reading of split index
| |
| |     The split index extension uses ewah bitmaps to mark index entries as
| |     deleted, instead of removing them from the index directly.  This can
| |     result in an on-disk index, in which entries of stage #0 and higher
| |     stages appear, which are removed later when the index bases are merged.
| |
| |     15999d0 read_index_from(): catch out of order entries when reading an
| |     index file introduces a check which checks if the entries are in order
| |     after each index entry is read in do_read_index.  This check may however
| |     fail when a split index is read.
```

```
| |
| |     Fix this by moving checking the index after we know there is no split
| |     index or after the split index bases are successfully merged instead.
| |
| |     Signed-off-by: Thomas Gummerer <t.gummerer@gmail.com>
| |     Signed-off-by: Junio C Hamano <gitster@pobox.com>
| |
* |   commit 6ef53d58ca003a6b5ca5433da8202b5977efddef
|\ \  Merge: 4061cc6 6c036d4
| | | Author: Junio C Hamano <gitster@pobox.com>
| | | Date:   Fri Mar 20 15:14:20 2015 -0700
:
```

위 커밋 그래프는 Git의 커밋 그래프다. 좌측에 각 브랜치를 나타내는 엣지를 노드(커밋)와 함께 표현했다.

4.5 커밋 범위

4.5.1 그래프에서의 도달 가능성

그래프에서 임의의 노드 A로부터 시작해 규칙에 맞게 그래프를 따라 다른 임의의 노드 B에 도달할 수 있을 경우 노드 B를 노드 A로부터 도달할 수 있는 노드라고 한다. 즉, 노드 B는 노드 A로부터 도달할 수 있는 노드 집합 중 한 요소이다.

Git 커밋 그래프에서 임의의 커밋으로부터 도달 가능한 커밋 집합이라는 것은 방향성이 있는 링크를 따라 이동하여 도달할 수 있는 상위 커밋들의 집합을 의미한다. 이 집합은 시작점이 되는 커밋에 영향을 미치는 상위 커밋의 집합이라는 점에서 중요하다.

4.5.2 커밋 범위를 이중 마침표로 지정하기

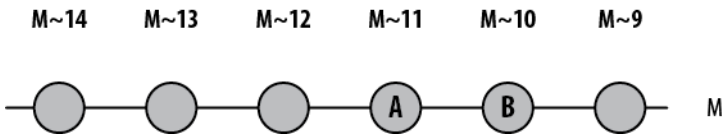

그림 4-9 커밋의 범위

좀 더 간단한 설명을 위해 선형 커밋 히스토리를 예로 보자. 커밋 A와 커밋 B를 범위로 지정하려면 Git에서는 다음과 같은 이중 마침표를 이용해 표현한다.

```
1 M~12..M10
```

Git은 범위 지정 시 from의 대상이 되는 커밋은 실제로 포함되지 않는다.

참고

이중 마침표를 사용할 때 전인자 혹은 후인자가 생략된 경우 그 자리를 HEAD가 대신한다.

4.5.3 커밋으로 범위 지정하기

임의의 커밋을 지정하여 `git log` 명령을 실행하는 것은 해당 커밋으로부터 **도달할 수 있는** 모든 커밋에 대한 커밋 로그를 출력하려는 것이다. 역(inverse)를 나타내는 캐럿(^)을 사용할 경우 이에 대한 역집합을 구할 수 있다.

예를 들어 앞서 본 선형 그래프 상에서 커밋 A로부터 도달할 수 있는 모든 커밋들의 로그를 조회하려면 다음과 같은 명령을 실행하면 된다.

```
1 $ git log M~11
```

만약 커밋 A 이후의 커밋들에 대한 로그를 조회하려면 다음과 같은 명령을 실행하면 된다.

```
1 $ git log ^M~11
```

이와 같은 정의를 이용하면 이중 마침표를 이용한 범위 지정 또한 커밋을 이용한 범위 지정으로 표현 가능하다.

`git log X..Y`는 X는 제외하고 Y로부터 X까지 도달 가능한 모든 커밋들의 집합인데 `git log ^X Y`가 그러한 의미이기 때문이다. 결국 X..Y와 ^X Y는 의미적으로 동일하기 때문에 수학적으로 봤을 때 차집합으로 볼 수 있다.

4.5.4 차집합으로 본 X..Y

앞에서 X..Y에서 X는 제외된다고 했다. 그 이유를 차집합적인 측면에서 증명해낼 수 있다.

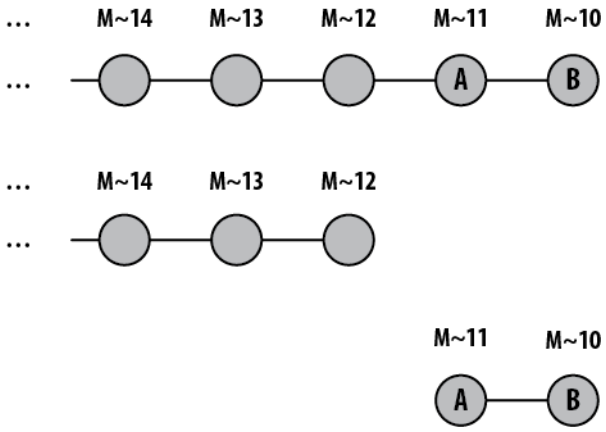

그림 4-10 두 그래프 간의 차집합

위 세 개의 그래프 중 첫 그래프는 M~10으로부터 도달 가능한 모든 커밋들의 집합을 나타낸다. 두번째는 M~12로부터 도달 가능한 모든 커밋들의 집합을 나타낸다. 세번째 그래프는 앞에 두 그래프의 차집합이 된다. 결국 X 자체는 제외된다.

4.5.5 브랜치 상에서의 차집합

아래 그래프에서 master 브랜치가 topic 브랜치로 V 커밋에서 병합된 것을 볼 수 있다. 브랜치 간의 커밋 관계를 커밋 비교에서와 마찬가지로 이중 마침표로 표현할 수 있는데 master 브랜치에는 속하지만 topic에는 속하지 않은 커밋들을 *topic..master*로 표현할 수 있다.

즉, master 브랜치에 속하는 커밋들 중 topic 브랜치에서 도달하지 못하는 {W, X, Y, Z} 커밋 집합을 말한다. 그리고 master 브랜치에 속하면서 topic 브랜치에서 도달 가능한 {..., T, U, V} 커밋 집합은 topic 브랜치에 **영향을 준다**고 말한다.

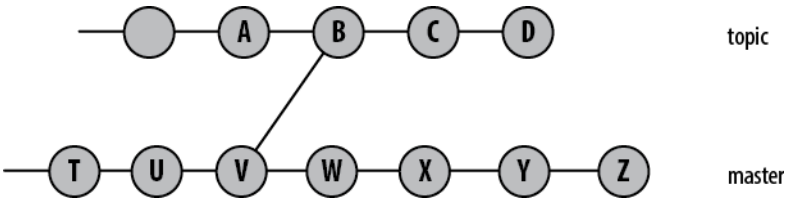

그림 4-11 두 개의 브랜치와 도달 가능한 커밋 집합

이제 topic 브랜치가 master 브랜치로 병합된 예를 살펴보자. 위 그래프에서 topic 브랜치는 자신의 커밋 B에서 master 브랜치의 커밋 X로 병합되었다.

앞에서의 예와 똑같이 master 브랜치에는 속하지만 topic 브랜치에는 속하지 않는 *topic..master*는 {V, W, X, Y, Z} 집합을 포함하여 이들로부터 도달할 수 있는 커밋들의 집합이다.

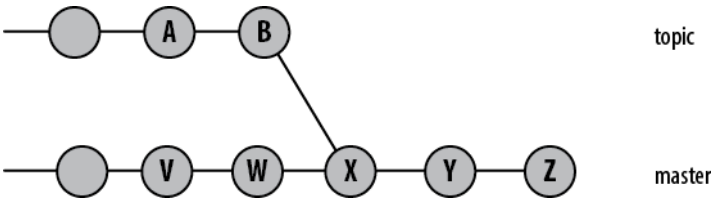

그림 4-12 이중 마침표로 표현한 도달 가능한 커밋 집합

master 브랜치가 topic 브랜치로 병합되었다가 다시 topic 브랜치가 master 브랜치로 병합된 예를 보자.

*topic..master*는 {W, X, Y, Z}가 된다. 커밋 B에서 master 브랜치가 병합되었으므로 커밋 V에서 도달 가능한 모든 커밋 또한 topic 브랜치에 영향을 주기 때문이다.

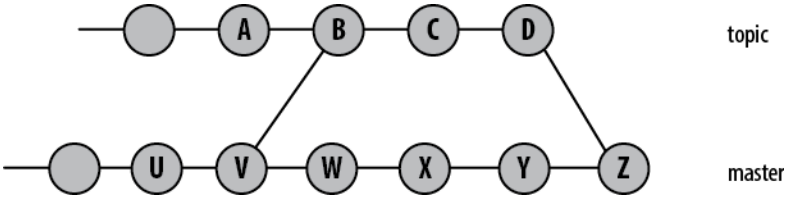

그림 4-13 병합과 도달 가능한 커밋 집합

4.5.6 삼중 마침표

앞에서 이중 마침표는 전인자와 후인자 간의 차집합을 나타냄을 알아봤다. Git에서는 삼중 마침표를 범위 표현을 위해 사용하는데 이는 **대칭 차집합**을 위해 사용한다.

A...B라는 표현이 있을 때, A나 B로부터 도달 가능하지만, A, B 모두로부터 도달할 수 없는 커밋의 집합이기 때문에 대칭이라는 표현을 사용한다.

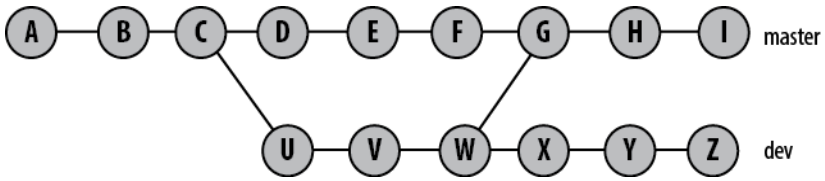

그림 4-14 두 브랜치 간의 대칭 차집합

위 그래프에서 대칭 차집합을 찾아보자. master 브랜치로부터 도달 가능하면서 dev 브랜치로부터는 도달하지 못하는 커밋 집합과, 반대로 master 브랜치로부터 도달하지 못하면서 dev 브랜치로부터는 도달 가능한 커밋 집합의 합집합을 구하면 된다.

전자의 경우 {I, H, G, F, E, D}이고 후자의 경우 {Z, Y, X}가 된다. 그러므로 두 브랜치 간의 대칭 차집합은 {I, H, G, F, E, D, Z, Y, X}가 된다.

이것은 두 브랜치 간의 병합 기반이 중요한데, 여기서 병합 기반은 커밋 W가 된다. 병합 기반에서 도달할 수 있는 커밋 집합을 제외하면 대칭 차집합이 된다.

결국 대칭 차집합이라는 것은 A와 B에서 도달할 수 있는 모든 커밋 중 두 그룹의 분기가 발생한 시점 이후 병합 대상 측의 모든 커밋과 두 그룹의 병합이 발생한 시점 이후 병합된 측의 모든 커밋의 합집합이 된다.

5. 브랜치

이 장에서는 브랜치에 대해서 설명하고 브랜치를 다루는 방법을 알아본다.

5.1 브랜치

Branch는 커밋 그래프 상에서 하나의 경로를 나타낸다.

Git은 각 저장소에 최소 1개 이상의 브랜치를 가지고 있는데 기본이 되는 브랜치가 바로 master 브랜치다. master 브랜치도 다른 브랜치와 다를 바가 없는 일반적인 브랜치이고 단지 각 저장소가 만들어지면 built-in되는 브랜치일 뿐이다.(그래서 master 브랜치의 이름을 바꿀 수도 있다.)

(master 브랜치를 제외하고) 브랜치들은 다른 브랜치로부터 분기되어 자신의 경로를 만들게 되며 또 다른 브랜치로 병합되기도 한다.

5.1.1 분기의 목적

만약, 개발 과정을 분기하지 않는다고 생각해보자. 한 개의 커밋 그래프가 그려질 것이고 분기가 없으므로 병합도 없다. 복잡하게 생각할 것이 별로 없다. 물론, 커밋의 순서를 바꾼다던가, 커밋을 덮어씌운다던가, 혹은 커밋을 무효 처리한다던가 하는 처리는 여전히 있어야 하겠지만 분기가 없으므로 그 과정 조차 더 쉽게 생각될 수 있다.

그 대신 단일화된 그래프로 인해서 유연성을 잃는다. 모든 코드는 릴리즈 코드를 대상으로 하여 테스트되어야 하거나 프로토타입을 진행해야 한다. 또, 릴리즈에 대한 버전도 따로 관리하지 못하게 된다.

만약, 분기가 가능하다면 앞에서 이야기한 것들에 대한 대처가 더욱 유연해질 것이다. 버전 컨트롤이 조금 더 어려워질 수 있지만 그 대신 얻는 것이 더 많다.

5.1.2 이름 짓기

브랜치의 이름은 자유롭게 결정할 수 있다. 대부분 어떤 용도를 가르키도록 짓는 경우가 대부분이며, 결과물에 대한 버전을 표시하기도 한다.

- 슬래쉬(/)를 사용하여 용도를 알아볼 수 있도록 네임스페이스를 구성하는 경우가 많다. 예를 들어, 핫픽스와 관련된 브랜치들은 `bug/dpc-2041` 등과 같이 접두어 *bug*를 붙이고 슬래쉬로 구분하는 것이다. 하지만 이름이 슬래쉬로 끝나서는 안된다. 그리고 슬래쉬로 구분된 네임스페이스에서 슬래쉬 바로 뒤에 마침표(`.`)를 붙여서는 안된다.
- 어떤 단어와 단어를 서로 확실히 구분하기 위해서 하이픈(-)을 사용하는 경우가 종종 있는데, 이름이 하이픈으로 시작해서는 안된다.
- 다중 마침표(예를 들어 `..`, `...`)는 사용하지 못한다.
- 스페이스나 탭과 같은 공백 문자는 사용하지 못한다.
- ~, ^, :, ?, *, [등 Git에서 사용하는 특수 문자는 사용하지 못한다.
- 아스키 제어 문자(₩040보다 작은 값을 가진 바이트 혹은 DEL 문자 ₩177)

브랜치 이름이 나타내는 의미와 목적은 명확한 것이 좋다. 너무 포괄적이거나 추상적인 이름은 사용의 이유를 쉽게 알지 못하게 하므로 혼동을 줄 수 있다.

5.1.3 활성 브랜치

저장소 내에 브랜치는 1개 이상일 수 있지만 활성 브랜치는 딱 1개뿐이다.(활성 브랜치는 현재 브랜치라고도 표현한다.) 이 활성 브랜치가 중요한 이유는 활성 브랜치의 내용이 작업 디렉토리의 내용이 되기 때문이다. 또한 작업 디렉토리의 변경 사항을 커밋하게 되면 그 커밋은 활성 브랜치의 커밋이 된다.

5.1.4 브랜치의 시작과 끝

브랜치의 시작을 ***merge base***라고 한다. Git은 이 merge base를 별도로 저장해두지 않는다. 새로운 커밋이 발생하게 되면 브랜치 이름을 해당 커밋으로 이동시키는데 특정 커밋에 대한 포인터 같은 역할을 한다.

이렇게 새로운 커밋(즉, 최근의 커밋)이 발생하면 브랜치 이름이 새로운 커밋을 향하게 되는데, 이를 브랜치의 끝(tip of the branch)이라고 한다. 브랜치 이름이 참조하고 있는 커밋은 이 브랜치의 끝을 말한다.

5.2 브랜치의 활용

5.2.1 브랜치 만들기

모든 브랜치는 저장소 내에 존재하는 어떤 커밋을 기반으로 한다. 브랜치를 만들기 위해서는 어떤 '시점'에 분기할 것인가를 정해야 하는데 그 기준은 커밋이다.

```
$ git branch dev
```

위와 같이 `git branch` 명령을 통해 브랜치와 관련된 작업이 이루어지는데(모든 브랜치 관련 작업이 다 그런 것은 아니다.) 뒤에 인자로 전달된 문자열이 브랜치의 이름이 된다.

브랜치를 만드는 명령어의 형식은 다음과 같다.

```
$ git branch <branchname> [<commit>]
```

브랜치 이름 뒤에는 커밋 해시를 전달하게 되어 있는 생략하게 되면 현재 브랜치의 HEAD로부터 브랜치를 만들게 된다. 커밋 해시대신 브랜치 이름을 전달하면 해당 브랜치의 끝에서 브랜치를 만들게 된다.

`git branch` 명령으로 브랜치를 만든다고 해서 새 브랜치를 체크아웃하진 않는다.('브랜치 체크아웃'이란 작업 디렉토리를 특정 브랜치의 끝에 해당하는 리비전으로 교체하는 것을 말한다.)

5.2.2 브랜치 나열하기

앞에서 *dev* 브랜치를 만들었으니 브랜치 목록에서 확인할 수 있을 것이다.

```
1  $ git branch
2    dev
3  * master
```

활성(현재) 브랜치가 master임을 애스터리스크(*)가 앞에 붙어있으므로 알 수 있다.

`-r` 플래그를 함께 전달하면 원격 저장소의 브랜치를 추적하는 브랜치를 조회할 수 있고, `-a` 플래그를 전달하면 모든 브랜치들을 조회할 수 있다.

5.2.3 브랜치 체크아웃

활성(현재) 브랜치라는 것은 작업 디렉토리가 어느 브랜치에 있는가를 말한다. 만약, 특정 브랜치의 리비전을 자신의 작업 디렉토리에 반영하려면 브랜치를 체크아웃해야 한다.

```
1  $ git checkout dev
2  Switched to branch 'dev'
```

앞서 만든 *dev* 브랜치로 체크아웃했다.

```
$ echo "This is a new file." >> newfile
$ git add newfile
$ git commit -m "add 'newfile"
[dev 93a6058] add 'newfile'
 1 file changed, 1 insertion(+)
 create mode 100644 newfile
$ git checkout master
Switched to branch 'master'
```

체크아웃한 브랜치에서 새로운 파일을 만들고 커밋한 후 다시 master 브랜치로 체크아웃해보자.

아마도 'newfile'이라는 파일은 작업 디렉토리 상에 존재하지 않을 것이다. 왜냐하면, 'newfile'이라는 파일은 master 브랜치 상에서는 존재하지 않았다. 'dev' 브랜치를 만든 뒤에 체크아웃하고 파일을 작성했기 때문에 그런 것이다.

즉, 체크아웃하게 되면 체크아웃한 브랜치의 리비전을 작업 디렉토리에 반영한다는 것을 알 수 있다.

5.2.4 커밋하지 않은 상태에서의 체크아웃

앞선 예제에서는 dev 브랜체이서 새로 만든 파일을 인덱스에 추가함은 물론 커밋까지 한 뒤에 다시 master 브랜치로 체크아웃했다.

만약, 인덱스에 추가도, 그리고 커밋도 하지 않았다면 어떻게 될까? 현재 브랜치에서 어떤 일을 하던 도중에 갑작스럽게 다른 수정 사항이 들어왔다. 엄청난 버그여서 지금 당장 수정해야 하는 사항이고 그것을 처리할 사람은 당신밖에 없다고 가정하자. 방금하던 작업은 아직 미완성 상태이고 억지로 커밋하고 싶지도 않다.

```
# dev 브랜치로 체크아웃했고 newfile을 수정했다.
$ vim newfile
$ git checkout master
error: Your local changes to the following files would be overwritten by checko
    newfile
Please, commit your changes or stash them before you can switch branches.
Aborting
```

아쉽지만 체크아웃은 실패한다. 이유는 작업 디렉토리에 변경사항이 발생했는데 그것이 리비전에 반영되지 않았기 때문이다. Git은 우리에게 변경사항에 대한 커밋을 하거나 혹은 안전한 곳에 넣어둘 것(stash)을 당부했다.

앞서 말했지만 커밋을 할 시점은 아니라고 가정했다. 안전한 곳에 넣어둔다는 것은 무슨 말인지 모르겠다.

이럴 때 우리에게는 두 가지 옵션이 있다.

`--force` 플래그는 그 단어에서도 느껴지듯이 변경사항은 모두 무시하고 무조건 체크아웃한다는 것이다. 새로운 브랜치에서 하던 작업이 사실 크게 성과가 없었다던가 필요가 없는 상황에서 사용할 수도 있을 것

같다. 하지만, 매우 조심해야 한다. Git에서 사용되는 모든 명령어의 `--force` 플래그는 무자비하다.(되돌릴 수 없다는 말이다.)

`--merge` 플래그도 상당히 직관적이다. 체크아웃과 동시에 병합해버린다는 의미다. 일단 한번 해보자.

```
$ git checkout --merge master
A    newfile
Switched to branch 'master' # 우리는 이제 master 브랜치로 체크아웃되었다.
$ git log -2
* 93a6058 | (dev) 2015-03-23 00:04:25 +0900 (27 minutes ago)
|           add 'newfile' - Choi Leejun
* c24a417 | (HEAD, master) 2015-03-20 23:28:43 +0900 (2 days ago)
             some stuff - Choi Leejun
```

뭔가 이상하다. `--merge` 플래그를 전달하자 체크아웃이 이루어졌다. 출력 메시지 중 `A newfile`이라는 것이 보인다. 여기서 A라는 것은 *Add*의 의미로 master 브랜치에는 없었던 파일이 추가되었다는 의미를 말한다. 만약 master 브랜치에도 있었던 파일이고 dev 브랜치가 분기될 때 함께 가져갔는데 수정된 파일이었다면 `M`으로 표현됐을 것이다.(삭제되었다면 당연히 `D`) 이러한 표현은 브랜치를 병합할 때 결과로 표현되는 것이다. 정말 병합이 된걸까?

그래서 로그를 확인해봤는데 우리가 생각하는 그런 병합은 이루어지지 않았다. 파일을 `ls -al` 명령으로 확인해보면 master 브랜치임에도 불구하고 newfile 이라는 파일이 존재할 것이다. 심지어 파일의 내용을 보면 dev 브랜치 상에서 수정했던 내용 그대로일 것이다.

```
$ git status
On branch master
Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add/rm <file>..." as appropriate to mark resolution)

    deleted by us:   newfile

no changes added to commit (use "git add" and/or "git commit -a")
```

`git status` 명령을 실행하면 굉장히 엉뚱해보이는 결과를 보여준다. *deleted by us* 항목에 newfile이 나열된다. 위쪽을 보니 *Unmerged paths* 항목이 나온다. 파일이 삭제되었다고 나오긴 하는데 아직 병합되지 않은 경로라고도 나오는 것으로 보아 병합이 아직 안끝난 것 같다.

결론적으로 병합하려면 newfile을 인덱스에 추가하고 커밋을 해야한다. 이런 과정까지 Git이 해주지는 않는다. 사실 이 대목에서 커밋을 하고 나면 어떻게 될지 직접 해보지 않는 이상은 상상이 잘 안 간다.

```
$ git add newfile
$ git commit -m "merge?"
[master f6eb99f] merge?
 1 file changed, 2 insertions(+)
 create mode 100644 newfile
$ git lg -3
* f6eb99f | (HEAD, master) 2015-03-23 00:33:06 +0900 (2 seconds ago)
|          merge? - Choi Leejun
| * 93a6058 | (dev) 2015-03-23 00:04:25 +0900 (29 minutes ago)
|/            add 'newfile' - Choi Leejun
* c24a417 | 2015-03-20 23:28:43 +0900 (2 days ago)
             some stuff - Choi Leejun
```

이러한 결과는 예상해본 적이 없다. 아마 그나마 예상했던 결과들은 대부분 dev 브랜치의 끝인 93a6058 커밋에서 로그 상 보이는 master 브랜치의 끝인 f6eb99f 커밋으로 병합되는 결과였을 것이다.

그러나 실제 결과에서는 아무 것도 병합된 것이 없다. dev 브랜치로 체크아웃한 뒤에 newfile을 확인해보면 수정사항은 반영되어 있다. 그리고 브랜치 상에서 변경된 내용은 전혀 없다. 변경 사항은 모두 master 브랜치로 이관됐고 심지어 두 브랜치 간에 변경 사항이 오고간 이력도 전혀 없다. 단지 master 브랜치의 커밋이 한 개 더 진행됐을 뿐이다.

굉장히 긴 설명이 이어졌는데 결론은 다음과 같다.

`git checkout --merge` 명령은 실제로는 병합이 아니라 파일 복사가 일어나고 그것을 커밋하도록 유도한다. *merge base* 상에는 없는 파일이 생겨나므로 인덱스 후 커밋을 해야만 한다.

그런데, 이상한 점은 실제로 병합은 일어나지 않는다는 것이다. 두 브랜치가 병합되지도 그렇다고 파일 복사가 왜 일어났는지에 대한 이력은 없다는 것이다. 그래서 엉뚱하다고 하고 결과에 대한 의문을 제기한 것이다. `--merge`플래그는 사용하지 않는 것이 좋다는 것이 최종 결론이다. 다소 황당한 결과물을 만들어냄은 물론 원래의 목적에도 맞지 않는다.

사실 이 섹션의 제목은 'Stash를 써야 하는 이유'가 더 어울릴지도 모른다. 앞서 이 섹션 처음에 체크아웃을 시도한 후 나온 메시지에서 'stash'에 대한 언급이 있었다. `git stash` 명령은 이 문제에 대해서 고급스러운 해결책을 제시해준다.

5.2.5 브랜치를 만들면서 체크아웃하기

`git branch <branchname>` 명령은 브랜치를 만들어주지만 체크아웃하지는 않는다. 이로 인해 브랜치를 체크아웃하지 않고 작업을 하는 실수를 유발할 수도 있다.(이를 위한 해결책도 `stash`다.)

그래서 브랜치를 만듦과 동시에 체크아웃을 하는 방법이 있다.

```
$ git checkout -b bug/pr-0112
Switched to a new branch 'bug/pr-0112'
```

이와 같이 `-b` 플래그를 함께 전달하면 해당 브랜치가 없을 경우 새로 만들고 체크아웃한다. 하지만 동일한 이름의 브랜치가 이미 존재할 경우에는 오류 메시지와 함께 명령이 중지된다. 만약 동일한 이름의 브랜치가 존재할 경우에도 체크아웃이 되길 원한다면 `-B` 플래그를 사용해야 한다.

```
$ git checkout -b master
fatal: A branch named 'master' already exists.
$ git checkout -B master
Switched to and reset branch 'master'
```

5.2.6 Detached HEAD

체크아웃은 브랜치와 브랜치를 옮겨다니는 용도로 주로 사용된다. 하지만, HEAD를 커밋으로 옮기는 용도로 사용되기도 한다.

```
$ git checkout HEAD~2
Note: checking out 'HEAD~2'.

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:

  git checkout -b new_branch_name

HEAD is now at 8f01aab... test
```

상대적인 커밋 이름을 지정해 체크아웃하면 위와 같은 메시지를 볼 수 있다. 혹은 커밋 해시를 직접 지정해도 되고, 태그를 지정해도 된다.(커밋 해시는 절대적인 커밋 이름이고 태그는 특정 태그를 참조하므로 모두 커밋을 참조한다는 면에서 동일하다.)

실제로 체크아웃이 어디로 이루어졌는지 보려면 브랜치 목록을 확인해보면 된다.

```
$ git branch
* (detached from 8f01aab)
  bug/pr-0112
  dev
  master
$ git lg
* f6eb99f | (master, bug/pr-0112) 2015-03-23 00:33:06 +0900 (2 hours ago)
|           merge? - Choi Leejun
| * 93a6058 | (dev) 2015-03-23 00:04:25 +0900 (2 hours ago)
|/            add 'newfile' - Choi Leejun
* c24a417 | 2015-03-20 23:28:43 +0900 (2 days ago)
|           some stuff --stage - Choi Leejun
* 8f01aab | (HEAD) 2015-03-20 23:24:31 +0900 (2 days ago)
|           test - Choi Leejun
* 18f01a7 | 2015-03-20 23:12:16 +0900 (2 days ago)
|           rename. - Choi Leejun
* 1985681 | 2015-03-20 23:09:37 +0900 (2 days ago)
             poooooooooooh - Choi Leejun
```

내친김에 로그까지 확인해봤는데 실제로 HEAD는 8f01aab 커밋에 위치해 있다. 즉, 작업 디렉토리가 해당 리비전으로 반영되어 있다는 것이다.

이렇게 어떤 브랜치의 끝이 아닌 커밋으로 HEAD가 이동해 있는 것을 ***detached HEAD***(분리된 HEAD)라고 한다. 보통 HEAD는 브랜치의 끝을 향하는 특성을 가지고 있는데 이와 달리 분리되어 있음을 표현한 말이다.

HEAD가 특정 커밋으로 분리되고 난 뒤에는 다른 때와 동일하게 작업이 가능하다. 작업 후에 커밋해보면 새로운 **경로**가 생기는데, 브랜치의 이름은 없다. 경로라고 표현한 것에는 이유가 있다. 이렇게 분리된 HEAD에서 새로운 브랜치 없이 커밋이 이루어지면 이 커밋으로 접근할 수 있는 방법은 체크아웃이나 리셋 밖에 없다.

대신 분리된 HEAD 상에서 새로운 브랜치를 만들게 되면 마치 처음부터 브랜치가 있었던 것처럼 작업을 이어나갈 수 있다.

5.2.7 브랜치 삭제

브랜치를 만들고 옮겨다닐 수 있지만 삭제 또한 가능하다.

```
1  # 이전에 dev는 병합되었다.
2  $ git branch -d dev
3  Deleted branch dev
```

`-d` 플래그(혹은 `--delete`)는 브랜치를 삭제한다.

브랜치를 삭제한다는 말이 많은 사람들에게 혼동을 주는 것 같다. 이것은 아마도 브랜치와 커밋 그래프에 대한 이해도가 낮아서 그런 것 같다. 브랜치라는 것은 브랜치 이름을 통한 커밋 참조 포인터이다. 물론 우리가 흔히 어떤 브랜치라고 부르는 것은 해당 브랜치로부터 merge base까지 도달할 수 있는 모든 커밋 집합을 이야기한다.(그래서 브랜치 이름을 통해 상대적 커밋 이름을 사용한다.) 하지만, 실제 내부적으로 브랜치는 단지 포인터일 뿐이다. 그러므로 브랜치를 따라 남겨진 커밋 히스토리들은 바로 지워지지 않는다.(git 가비지 콜렉션에 의해 얼마가지 않아 삭제되기는 한다.)

브랜치를 삭제할 때 고려해야 할 상황이 있다.

5.2.8 아직 병합되지 않은 브랜치를 삭제하기

```
1  $ git branch -d dev
2  error: The branch 'dev' is not fully merged.
3  If you are sure you want to delete it, run 'git branch -D dev'.
```

dev 브랜치가 아직 병합되지 않았다고 가정하자. `-d` 플래그를 전달하여 브랜치를 삭제하려고 하면 위와 같은 메시지를 보게 된다.

dev 브랜치가 아직 완전히 병합되지 않았고 만약 그것이 삭제되어도 상관없다면 `git branch -D dev` 명령을 실행하라고 한다.

여기서 **완전히 병합되지 않았다**라는 말은 병합에 대한 이해가 있는 사람이라면 무슨 말인지 알 것이다.
병합은 두 가지 측면에서 안 됐을 수 있다는 것이다. 아예 시도 조차하지 않았거나, 혹은 시도했으나 충돌이 발생했는데 아직 해결하지 않았거나.

`-D` 플래그를 전달하여 삭제해보자.

```
$ git branch -D dev
Deleted branch dev (was 93a6058).
$ git log -3
* f6eb99f | (HEAD, master, bug/pr-0112) 2015-03-23 00:33:06 +0900 (2 hours ago)
|           merge? - Choi Leejun
* c24a417 | 2015-03-20 23:28:43 +0900 (2 days ago)
|           git ls-files --stagegit ls-files --stage - Choi Leejun
* 8f01aab | 2015-03-20 23:24:31 +0900 (2 days ago)
|           test - Choi Leejun
```

`-D`플래그는 브랜치의 삭제를 병합 여부와는 상관없이 강제한다. 로그를 확인해보면 dev 브랜치를 통해 진행됐던 커밋이 나오지 않는다.

```
$ git show 93a6058
commit 93a6058e46ec513ee5e0234870ce2b59c1452338
Author: Choi Leejun <devcken@gmail.com>
Date:   Mon Mar 23 00:04:25 2015 +0900

    add 'newfile'

diff --git a/newfile b/newfile
new file mode 100644
index 0000000..6434b13
--- /dev/null
+++ b/newfile
@@ -0,0 +1 @@
+This is a new file.
```

하지만 `git show` 명령을 통해 해당 커밋을 열람해보면 아직 커밋이 살아있다는 것을 알 수 있다.(이 커밋은 git 가비지 컬렉션에 의해 삭제될 가능성이 있다.)

만약 브랜치의 이력이 원격 저장소에 올라간 상태라면 이 커밋은 로그에서 볼 수 있다. 왜냐하면 공개된 커밋이기 때문에 이력을 유지해야 하기 때문이다.

여기서 궁금한 점은 삭제된 브랜치의 이력을 과연 병합할 수 있느냐이다.

5.2.9 삭제된 브랜치의 이력을 병합하기

앞의 질문에 대한 답은 '그렇다'이다. 커밋 이름 식별에 대해 잘 알고 있는 사람이라면 당연히 가능하다고 답변했을 것이다.

먼저 병합 대상이 될 브랜치로 체크아웃해야 한다.

```
$ git merge 93a6058e46ec513ee5e0234870ce2b59c1452338
Auto-merging newfile
CONFLICT (add/add): Merge conflict in newfile
Automatic merge failed; fix conflicts and then commit the result.
$ vim newfile
$ git add newfile
$ git commit -m "Merge"
[master 6947a3f] Merge
$ git log -4
*   6947a3f | (HEAD, master) 2015-03-23 02:55:23 +0900 (4 seconds ago)
|\            Merge - Choi Leejun
| * 93a6058 | 2015-03-23 00:04:25 +0900 (3 hours ago)
| |           add 'newfile' - Choi Leejun
* | f6eb99f | (bug/pr-0112) 2015-03-23 00:33:06 +0900 (2 hours ago)
|/            merge? - Choi Leejun
* c24a417 | 2015-03-20 23:28:43 +0900 (2 days ago)
|           git ls-files --stagegit ls-files --stage - Choi Leejun
```

앞선 다른 예제들로 인해 두 브랜치 간의 차이점 발생으로 충돌(newfile에서)이 발생했지만 충돌을 해결한 뒤에 최종적으로 병합이 완료되었다.

삭제되었던 브랜치의 이력 또한 로그 상에서 잘 나온다. 그리고 이제 병합된 커밋 이력들은 가비지 컬렉션의 대상도 아니다.

6. 커밋 간의 차이점

이 장에서는 커밋 간의 차이점을 알아보는 방법에 대해서 설명한다.

UNIX 계열의 운영체제에는 diff라는 유틸리티가 있다. 간단히 설명하자면 두 파일 간의 차이를 출력해주는 프로그램이다.

이렇게 차이점을 출력해주는 기능은 버전 컨트롤에 있어서도 매우 중요하다. 어떠한 상황에서 diff가 필요한 걸까?

1. 어떤 파일(혹은 파일들)을 인덱스에 추가해놓고 계속해서 다른 파일을 열어 작업 중이었다. 그런데 갑자기 핫픽스가 들어와 모든 일을 중단해야 했고 핫픽스를 해결했다. 돌아와서 보니 어떤 일을 했었는지 구체적으로 생각이 나질 않는다.
2. 현재 작업 중인 파일(아직 인덱스에는 추가하지 않은 파일)이 저장소의 HEAD 리비전 blob가 어떤 차이가 있는지 궁금할 때가 있다.
3. 어떤 파일(혹은 파일들)을 인덱스에 추가해놓고 동일한 작업 디렉토리 내에서 다른 작업을 했다. 커밋을 해야 하는데 이전에 했던 작업이 어떤 내용인지 잘 안 떠오른다.
4. 모든 변경사항이 저장소에 커밋된 상태에서 HEAD와 어떤 시점의 커밋 간의 리비전 차이가 궁금할 때가 있다.(릴리즈할 때 변경 사항만 필요한 경우가 있다.)

참고

두 커밋 간의 diff가 필요한 때가 과연 있을까 싶기도 했다. 버그 픽스를 위해서 diff를 이용한 히스토리 추적이 어느 정도는 유용하겠지만 그렇게 빈도수가 높진 않을거라는 생각이었다. 하지만, 이 diff가 빈번히 필요한 경우가 내 근처에 있었다.

6.1.2 상황별 diff

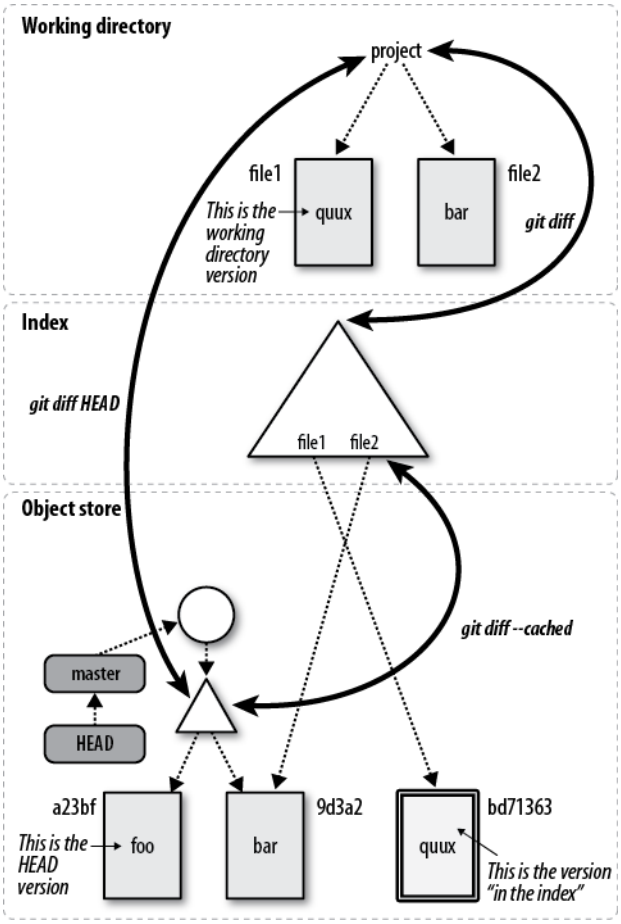

그림 6-1 상황별 diff

위 그림으로 앞에서 말했던 4가지 상황에 대한 해결 방법이 모두 설명된다. 굵은 선 3개에 대해 알아보자.

A. 인덱스 vs 작업 디렉토리

첫번째 경우는, 인덱스와 작업 디렉토리 간의 차이점을 알아내야 할 경우다. 그림에서 보면 작업 디렉토리와 인덱스 간을 양방향으로 연결한 화살표 선이 보일 것이다. 두 지점을 서로 diff하는 것은 다음과 같다.

```
$ echo "first" >> stuff
$ git add stuff
$ echo "second" >> stuff
$ git add stuff
# second라는 텍스트를 지운 뒤, 세번째 줄에 third라는 텍스트를 추가했다.
$ vim stuff
$ git diff -- stuff
diff --git a/stuff b/stuff
index 66a52ee..48858ad 100644
--- a/stuff
+++ b/stuff
@@ -1,2 +1,3 @@
 first
-second
+
+third
```

인덱스에 들어가 있는 stuff의 콘텐츠와 작업 디렉토리의 stuff를 비교한 결과다. 출력 결과를 보면 a/라는 접두사와 b/라는 접두사를 볼 수 있다. a/는 인덱스, /b는 작업 디렉토리를 나타낸다고 보면 이해가 쉽다.

그러므로 `--- a/stuff`는 인덱스에 있는 내용으로 밑에서 -가 붙는 줄이 이에 해당하고, `+++ b/stuff`는 작업 디렉토리에 있는 내용으로 밑에서 +가 붙는 줄이 이에 해당한다.

diff한 결과로 봤을 때, stuff에 있던 second라는 텍스트가 지워지고 세번째 줄에 third라는 텍스트가 추가된 것이다.

출력 결과 두번째 줄에서 index 66a52ee라고 되어 있는 부분은 인덱스 상에서의 stuff의 해시다. 그리고 48858ad는 작업 디렉토리에서의 stuff의 해시다. 이중 마침표로 연결되어 있는 것을 기억하자.

B. 작업 디렉토리 vs 저장소

두번째의 경우, 작업 디렉토리와 저장소 간의 비교가 필요하다. 예제가 조금 복잡할 수 있으니 주석문을 잘 살펴보는 것이 좋다.

```
# 인덱스에 추가된 stuff의 리비전을 커밋한다.
$ git commit -m "initial commit"
# 두번째로 변경된 내용을 인덱스에 추가한다.(second라는 텍스트를 지우고 third라는 텍스트를 세번째 줄
$ git add stuff
# 네번째 줄에 fourth라는 텍스트를 추가한다.
$ echo "fourth" >> stuff
# 작업 디렉토리와 HEAD를 비교한다.
$ git diff HEAD -- stuff
diff --git a/stuff b/stuff
index 66a52ee..2e364c9 100644
--- a/stuff
```

```
+++ b/stuff
@@ -1,2 +1,4 @@
 first
-second
+
+third
+fourth
```

HEAD, 즉 가장 최근 커밋의 stuff의 내용은 다음과 같다.

```
first
second
```

인덱스의 내용은 다음과 같다.

```
first

third
```

그리고 작업 디렉토리 상의 내용은 다음과 같다.

```
first

third
fourth
```

작업 디렉토리와 현재 브랜치의 끝인 HEAD와의 비교이므로 위와 같은 결과가 나온 것이다. 즉, `git diff` 명령에 커밋 해시 혹은 참조를 전달하면 작업 디렉토리와 커밋 간의 비교가 이루어진다.

C. 인덱스 vs 저장소

세번째의 경우는 인덱스와 저장소 간의 비교가 된다.

```
$ git diff --cached HEAD -- stuff
diff --git a/stuff b/stuff
index 66a52ee..48858ad 100644
--- a/stuff
+++ b/stuff
@@ -1,2 +1,3 @@
 first
-second
+
+third
```

`--cached` 플래그를 전달하면 작업 디렉토리가 아닌 인덱스를 저장소와 비교하게 된다. 앞에서 콘텐츠의 상태를 나열해봤었는데 인덱스와 저장소의 콘텐츠 상태를 잘 생각해보면 된다.

D. 커밋 vs 커밋

마지막 네 번째 경우는 저장소 내 비교를 해야한다.

```
# 두번째로 변경된 내용을 커밋한다.(second라는 텍스트를 지우고 third라는 텍스트를 세번째 줄에 추가한
$ commit -m "remove second & add third"
$ git log
commit b822f0e9f0f8f33ca1c098d734f16ccb3fe2e9a8
Author: Choi Leejun <devcken@gmail.com>
Date:   Mon Mar 23 15:35:23 2015 +0900

    remove second & add third

commit fa527cf7ce9b5972e46cb27d82d6e024c042a749
Author: Choi Leejun <devcken@gmail.com>
Date:   Mon Mar 23 14:35:05 2015 +0900

    Initial commit
```

인덱스에 들어있던 두번째 변경사항을 드디어 커밋했다. 확인해보니 저장소에는 두 개의 커밋이 존재한다. 두 커밋을 비교하게 될텐데 여기서 잠깐 아래 결과를 보기 전에 어떤 diff 결과가 나올지 예측해보자.

예측한 결과와 아래 결과가 어떠한가보자.

```
$ git diff HEAD~1..HEAD -- stuff
diff --git a/stuff b/stuff
index 66a52ee..48858ad 100644
--- a/stuff
+++ b/stuff
@@ -1,2 +1,3 @@
 first
-second
+
+third
```

만약 예측한 결과가 위와 동일하다면 앞서 본 모든 것을 이해했을 가능성이 높다. 인덱스에 들어 있던 내용이 커밋되면서 HEAD가 됐고 저장소의 이전 HEAD가 HEAD~1이 됐으므로 두 커밋을 비교한 결과는 커밋 이전에서 인덱스와 HEAD를 비교한 결과와 동일하다.

예에서는 상대적 커밋 이름을 사용했으나 커밋 해시를 직접적으로 사용하는 것도 가능하다.

참고

diff를 설명하면서 이중 하이픈(--)을 사용했는데 이중 하이픈은 인자의 영역을 구분하는 용도로 쓰일 뿐 어떤 의미도 가지고 있지 않다. 예를 들어, HEAD -- stuff라고 한 부분은 커밋과 파일을 구분하여 지정하기 위한 것이다.

7. 병합

이 장에서는 병합과 병합 시 발생할 수 있는 충돌을 극복하는 방법, 그리고 병합 전략에 대해서 설명한다.

7.1 병합

브랜치는 개발의 흐름을 나타낸다. 각 브랜치들은 저마다의 의미와 목적이 있고 쓰임새가 있다. 하지만 (master 브랜치를 제외한) 모든 브랜치는 부모가 브랜치가 있다. 브랜치로부터 파생된 브랜치는 상황에 따라 버려질 수도 있지만 대부분 다른 브랜치와 합쳐져 자신이 가지고 있던 의미와 목적을 다른 브랜치에 전달한다. 이것을 병합(merge)이라고 한다.

7.1.1 두 브랜치 결합하기

아주 기초적인 것부터 시작하기 위해서 아무런 저항(충돌)없는 상태를 만들어보자.

```
$ git log
* aad6aa6 | (HEAD, feature/1) 2015-03-24 23:14:23 +0900 (1 second ago)
|          add second stuff - Choi Leejun
| * 4a21cdc | (master) 2015-03-24 23:13:22 +0900 (62 seconds ago)
|/            edit stuff 'Where is the second stuff?' - Choi Leejun
* d103b6e | 2015-03-24 23:08:25 +0900 (6 minutes ago)
            Initial commit - Choi Leejun
```

저장소에는 master 브랜치와 feature/1 브랜치가 있다. 그리고 master 브랜치에서는 stuff라는 파일이 만들어져 aad6aa6 커밋까지 진행되었고 feature/1 브랜치에는 stuff2라는 파일을 만들어 4a21cdc 커밋이 진행되었다. stuff 파일은 수정되지 않은 상태다.

두 브랜치가 각각 커밋을 진행했고 서로 겹치게 수정한 파일은 없다. feature/1에서 진행된 사항들을 master 브랜치로 가져오고 싶어졌다.

```
# 먼저 master 브랜치로 체크아웃한다.
$ git checkout master
$ git merge feature/1
```

병합을 하려면 먼저 master 브랜치로 체크아웃한다. 그리고 `git merge` 명령으로 병합할 브랜치를 병합한다. 여기까지 제대로 했다면 아마도 *vim*이 실행됐을 것이다.(혹은 자신이 지정한 편집기가 실행됐을 것이다.)

```
Merge branch 'feature/1'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

이런 메시지가 나왔을텐데, 커밋할 때와 똑같은 메시지가 나왔다. 단지 주석 처리되지 않은 *Merge branch `feature/1`*이라는 메시지가 찍혀있는 것을 제외하고는 완전히 동일하다.

뭔지 모르지만 일단 **:**을 누른 뒤에 wq을 입력해 저장해보자.

```
$ git merge feature/1
Merge made by the 'recursive' strategy.
```

위와 같이 결과 메시지가 출력된다면 성공한 것이다. 로그를 확인해보자.

```
$ git log
*   65c8b16 | (HEAD, master) 2015-03-24 23:19:38 +0900 (7 minutes ago)
|\            Merge branch 'feature/1' - Choi Leejun
| * aad6aa6 | (feature/1) 2015-03-24 23:14:23 +0900 (12 minutes ago)
| |           add second stuff - Choi Leejun
* | 4a21cdc | 2015-03-24 23:13:22 +0900 (13 minutes ago)
|/            edit stuff 'Where is the second stuff?' - Choi Leejun
* d103b6e | 2015-03-24 23:08:25 +0900 (18 minutes ago)
             Initial commit - Choi Leejun
```

앞에서는 보지 못했던 새로운 커밋이 생겼다. 그리고 그 커밋의 로그 메시지는 vim이 실행됐을 때 봤던 그 메시지다. 이 커밋을 **병합 커밋(merge commit)**이라고 한다.

그런데 이 병합 커밋은 왜 생긴 걸까?

7.1.2 3-ways merge

master 브랜치로부터 분기된 feature/1 브랜치에서 어떤 일이 벌어지는 동안 master 브랜치에서도 어떤 일이 벌어졌다. 그 결과가 각각 aad6aa6 커밋과 4a21cdc 커밋이다. 두 커밋은 다른 내용을 가지고 있다.

서로 다른 두 커밋을 가진 브랜치가 하나로 병합되려고 한다. 이럴 때 이렇게 생각할 수도 있다. 시간에 따라 정렬해서 하나의 줄기로 만들면 되지 않나? 비슷하게 할 수 있는 방법이 실제로 있다. 하지만, 만약 feature/1 브랜치의 커밋 히스토리를 그대로 보존하고 싶다면 어떻게 해야할까?

커밋 d103b6e에서 두 브랜치는 분기되었다. 이것을 **병합 기점(merge base)**라고 부른다. feature/1 브랜치의 이력을 그대로 보존하면서 병합하려면 이와 반대로 병합되는 곳이 필요하지 않을까? 그래서 병합 커밋이 필요한 것이다.

그리고 이렇게 두 브랜치가 양쪽의 이력을 그대로 보존하고자 병합 커밋을 만들면서 병합되는 것을 ***3-ways merge***라고 한다.

3-ways merge의 특징은 병합된 브랜치는 여전히 제자리라는 것이다. 즉, 우리가 병합을 이야기할 때 브랜치를 병합한다고 하는데 이것은 사실 틀린 말이다. 브랜치의 진행 사항들을 병합한다라고 말하는 것이 맞다. 하지만 편의를 위한 것이므로, 진행 사항들을 병합한다는 사실만 알고 있으면 된다.

그리고 또 한 가지는 feature/1 브랜치에는 여전히 master 브랜치의 진행 사항(stuff 파일의 변경)들이 적용되어 있지 않다. 똑같이 병합해주면 되지 않을까?

7.1.3 Fast-forward merge

```
$ git checkout feature/1
$ git merge master
Updating aad6aa6..65c8b16
Fast-forward
 stuff | 1 +
 1 file changed, 1 insertion(+)
```

feature/1 브랜치를 체크아웃하여 병합을 시도했다.

결과 메시지를 보자. aad6aa6 커밋으로부터 65c8b16 커밋으로 업데이트되고 stuff 라는 파일의 변경 내역이 나온다. 1개 파일의 수정, 그리고 1줄이 삽입되었다는 뜻이다. 그런데 Fast-forward라는 메시지도 보인다. 이건 뭘까?

좀 더 명확한 실험을 위해 다음과 같은 조건을 만들자.

```
$ git log
* 6445bb5 | (HEAD, feature/1) 2015-03-25 00:05:42 +0900 (4 seconds ago)
|           edit stuff2 'The first stuff is here.' - Choi Leejun
* 094ec0e | 2015-03-25 00:04:23 +0900 (83 seconds ago)
|           edit stuff 'Ah! The second stuff is here.' - Choi Leejun
*   65c8b16 | (master) 2015-03-24 23:19:38 +0900 (46 minutes ago)
|\            Merge branch 'feature/1' - Choi Leejun
| * aad6aa6 | 2015-03-24 23:14:23 +0900 (51 minutes ago)
| |           add second stuff - Choi Leejun
* | 4a21cdc | 2015-03-24 23:13:22 +0900 (52 minutes ago)
|/            edit stuff 'Where is the second stuff?' - Choi Leejun
* d103b6e | 2015-03-24 23:08:25 +0900 (57 minutes ago)
             Initial commit - Choi Leejun
```

feature/1 브랜치에서 두 번의 커밋이 있었다. 그리고 master 브랜치에서는 커밋이 진행되지 않았다. 즉, 한쪽에서만 변경이 일어난 것이다. 아마도 눈치가 빠른 사람이라면 무엇을 말하려는지 이미 알고 있을 것이다.

```
$ git checkout master
$ git merge feature/1
Updating 65c8b16..6445bb5
Fast-forward
 stuff  | 1 +
 stuff2 | 1 +
 2 files changed, 2 insertions(+)
```

feature/1 브랜치의 변경 사항들이 master 브랜치로 병합되었다.

```
* 6445bb5 | (HEAD, master, feature/1) 2015-03-25 00:05:42 +0900 (5 minutes ago
|          edit stuff2 'The first stuff is here.' - Choi Leejun
* 094ec0e | 2015-03-25 00:04:23 +0900 (7 minutes ago)
|          edit stuff 'Ah! The second stuff is here.' - Choi Leejun
*   65c8b16 | 2015-03-24 23:19:38 +0900 (51 minutes ago)
|\           Merge branch 'feature/1' - Choi Leejun
| * aad6aa6 | 2015-03-24 23:14:23 +0900 (57 minutes ago)
| |          add second stuff - Choi Leejun
* | 4a21cdc | 2015-03-24 23:13:22 +0900 (58 minutes ago)
|/           edit stuff 'Where is the second stuff?' - Choi Leejun
* d103b6e | 2015-03-24 23:08:25 +0900 (63 minutes ago)
             Initial commit - Choi Leejun
```

그런데 로그를 살펴보니 변한 것은 master 브랜치의 끝이 65c8b16 커밋에서 6445bb5 커밋으로 이동한 것외에는 변한 것이 없다.

이것은 앞에서 3-ways merge 후 master 브랜치를 feature/1 브랜치에 병합한 것과 동일한 형태의 병합이다. master 브랜치으로만 병합이 일어났기 때문에 master 브랜치의 변경 사항이 feature/1 브랜치에는 적용되지 않았고 feature/1 브랜치에도 적용하기 위해 병합하니 Fast-forward되었다.

이번에는 feature/1 브랜치에서만 변경사항이 발생했고 이를 master 브랜치로 병합했을 뿐, 방법과 형태적으로는 완전히 동일하다. 이것을 ***fast-forward merge***라고 한다.

7.1.4 똑같은 변경

엉뚱한 상상을 많이 하는 사람이라면 이런 생각을 했을 수도 있다.

"만약 변경사항이 완전히 똑같은 커밋이 서로 다른 브랜치에서 일어난다면 어떻게 될까? 이걸 병합하면 하나로 합쳐질까?"

알 방법은 딱 하나. 직접 해보는 수밖에 없다. 물론 어느 정도 Git 객체에 대한 지식이 있는 사람이라면 예상할 수 있으나 한번 해보자. 병합 연습도 하면서 진짜 그렇게 되는지 확인할 겸.

참고

이 부분이 이해가 가지 않는 사람(혹은 궁금하지 않는 사람)은 건너뛰어도 좋다. 이름이 같은 파일을 서로 다른 브랜치에서 완전히 동일하게 변경할 가능성은 거의 0에 가까울 것이다.

```
$ git log
* 733ae07 | (HEAD, feature/1) 2015-03-25 00:23:48 +0900 (2 seconds ago)
|          edit stuff 'Hmm. Jump to Next.' - Choi Leejun
| * f364a7c | (master) 2015-03-25 00:22:08 +0900 (2 minutes ago)
|/            edit stuff 'Hmm. Jump to Next.' - Choi Leejun
* 6445bb5 | 2015-03-25 00:05:42 +0900 (18 minutes ago)
|          edit stuff2 'The first stuff is here.' - Choi Leejun
...
```

변경 사항이 완전히 똑같은 두 개의 커밋을 각각 양쪽 브랜치에서 진행했다. 심지어 커밋 로그 메시지도 똑같다. 다른 것은 커밋 해시와 날짜, 그리고 브랜치가 다르다는 것이다.

```
$ git merge feature/1
Merge made by the 'recursive' strategy.
$ git log
*   2606411 | (HEAD, master) 2015-03-25 00:29:01 +0900 (8 seconds ago)
|\            Merge branch 'feature/1' - Choi Leejun
| * 733ae07 | (feature/1) 2015-03-25 00:23:48 +0900 (5 minutes ago)
| |           edit stuff 'Hmm. Jump to Next.' - Choi Leejun
* | f364a7c | 2015-03-25 00:22:08 +0900 (7 minutes ago)
|/            edit stuff 'Hmm. Jump to Next.' - Choi Leejun
* 6445bb5 | 2015-03-25 00:05:42 +0900 (23 minutes ago)
|           edit stuff2 'The first stuff is here.' - Choi Leejun
...
```

일반적인 3-ways merge와 똑같다. 아니 3-ways merge다.

이 결과는 많은 사람들이 예상했을 결과다. 커밋 해시가 다르기 때문이다. 사실 정말 궁금했던 것은 따로 있다.

"blob이 만들어질 때 파일의 내용에 의해서 blob 해시를 만든다고 했는데, 그렇다면 서로 다른 브랜치에서 진행된 커밋이 (정확히는 트리 객체가) 동일한 blob 해시를 참조할까?"

쉽게 말해 변경된 파일이 하나로 저장될까 아니면 따로 저장될까를 말하는 것이다.

```
# feature/1 브랜치로 체크아웃했다.
$ git checkout feature/1
$ git ls-files --stage
100644 ed71c828ffe5de50062e95f380c1e573b43d8284 0    stuff
100644 13142c21151a0ff5e9ed77687723ffe1bb6841d2 0    stuff2
# master 브랜치로 체크아웃했다.
$ git checkout master
$ git ls-files --stage
100644 ed71c828ffe5de50062e95f380c1e573b43d8284 0    stuff
100644 13142c21151a0ff5e9ed77687723ffe1bb6841d2 0    stuff
```

방법은 아주 간단하다. 병합 전에 두 브랜치에서 인덱스를 살펴보았다. 변경사항이 발생해 인덱스에 추가하기 전까지는 한 브랜치 내에서의 인덱스와 최신 커밋의 트리 객체는 동일하다. 그러므로 양쪽 브랜치의 인덱스를 비교해보면 쉽게 알 수 있다.

서로 다른 브랜치에서의 (병합되기 이전) 인덱스를 살펴보았는데 완전히 똑같다. stuff2 파일이야 이전에 병합됐을 때 이미 양쪽이 똑같이 되었고 stuff 파일은 양쪽 브랜치에서 동일한 내용이지만 따로 수정, 커밋되었다. 그런데 인덱스 상의 해시는 동일하다.

즉, **Git은 파일 내용을 통해 blob 해시를 만들고 또 서로 다른 커밋일지라도 내용이 같으면 동일 blob을 참조한다**는 것이 증명되었다.

7.2 병합 취소

병합을 시도했거나 완료한 뒤에 병합을 취소하고 싶을 수도 있다. 병합을 취소하는 시점은 두 가지가 있을 것이다. 병합이 완료된 경우, 그리고 병합하려다가 충돌이 발생하여 잠시 중단된 경우이다.

다시 한번 병합 중 충돌 상황을 재현해보자. 이전 병합 이후부터 다시 시작할 것이다.(당연한 이야기지만 master 브랜치도 feature/1 브랜치로 병합되어 있어야 한다.)

```
$ git log
* f473009 | (HEAD, master) 2015-03-26 10:09:38 +0900 (3 seconds ago)
|          edit stuff 'Here is master branch.' - Choi Leejun
| * 176a7cd | (feature/1) 2015-03-26 10:08:55 +0900 (46 seconds ago)
|/           edit stuff 'Here is feature/1 branch' - Choi Leejun
*   8c302ea | 2015-03-25 15:20:59 +0900 (19 hours ago)
|\            Merge branch 'feature/1' - Choi Leejun
```

각각의 브랜치에서 stuff 파일을 다른 내용으로 수정했다. 이제 두 브랜치의 끝을 병합해보자.

```
$ git merge feature/1
Auto-merging stuff
CONFLICT (content): Merge conflict in stuff
Automatic merge failed; fix conflicts and then commit the result.
$ git branch
  feature/1
* master
$ 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:   stuff

no changes added to commit (use "git add" and/or "git commit -a")
$ git ls-files --unmerged
100644 ed71c828ffe5de50062e95f380c1e573b43d8284 1    stuff
100644 d2c7593de4a4d5a3883444c1be9886f1eaa9db66 2    stuff
100644 d3ab27bf9681e93d0adae8816bf9e690b62f0b03 3    stuff
```

예문이 좀 긴데, 앞서 본 내용과 전혀 다르지 않다. 병합 중 충돌이 발생한 상황을 몇 가지 명령어로 알아본 것이다.

먼저, stuff 파일에서 충돌이 발생한 것을 병합 명령 이후 바로 알 수 있고, HEAD는 master 브랜치에 있다. `git status` 명령으로 충돌 상황을 확신할 수 있고, 또 `git ls-files --unmerged` 명령으로 현재 인덱스에 stuff 파일의 blob 세개가 얽혀 있음을 알수 있다.

이와는 다르게 또 하나의 중요한 점이 있는데 바로 작업 디렉토리이다. 충돌이 발생하면 작업 디렉토리가 반드시 변하게 된다.

그러면 충돌 상황에서 병합을 취소하려면 어떻게 해야 할까?

잘 생각해보면 변한 것은 병합 대상인 master 브랜치 밖에 없다. 그중에서도 master 브랜치의 인덱스와 작업 디렉토리가 변한 것이다. 그것들을 되돌려주면 되는 것이다.

```
$ git reset --hard HEAD
HEAD is now at f473009 edit stuff 'Here is master branch.'
$ git status
On branch master
nothing to commit, working directory clean
$ git ls-files --unmerged
```

`git reset` 명령에서 인덱스와 작업 디렉토리를 모두 되돌려주는 방법은 `--hard` 플래그를 사용하는 것이다. 그리고 HEAD를 기점으로 되돌려주면 된다.

다른 한가지 경우인 병합 이후에 병합을 취소하려면 어떻게 해야할까?

현재 충돌 상태에서 다시 되돌린 상태이기 때문에 이번 섹션에서 본 커밋 히스토리와 동일한 상태일 것이다.(만약 아니라면 뭔가 잘못된거다.) 다시 병합하고 충돌을 해결하여 병합을 완료하자.

```
$ git log
*   8ce7700 | (HEAD, master) 2015-03-26 10:48:55 +0900 (4 seconds ago)
|\            Merge branch 'feature/1' - Choi Leejun
| * 176a7cd | (feature/1) 2015-03-26 10:08:55 +0900 (40 minutes ago)
| |           edit stuff 'Here is feature/1 branch' - Choi Leejun
* | f473009 | 2015-03-26 10:09:38 +0900 (39 minutes ago)
|/            edit stuff 'Here is master branch.' - Choi Leejun
*   8c302ea | 2015-03-25 15:20:59 +0900 (19 hours ago)
|\            Merge branch 'feature/1' - Choi Leejun
```

아마도 위와 비슷한 상태일 것이다. 이런 상태에서 병합을 취소하려면 어떻게 해야 할까? 현재 HEAD를 f473009 커밋으로 되돌리면 된다.

Git은 이러한 상황에 대비하여 ORIG_HEAD라는 참조 이름을 제공해준다. ORIG_HEAD에는 병합이나 커밋이 진행될 때 이전 HEAD를 저장해두는 이름이다.(병합이나 커밋이 진행되는 것이나 동일하다.) 실제로 그런지 확인해보자.

```
$ git show ORIG_HEAD
commit f473009a7d4900c264bf88d0356f64dda819a3f2
Author: Choi Leejun <devcken@gmail.com>
Date:   Thu Mar 26 10:09:38 2015 +0900

    edit stuff 'Here is master branch.'

diff --git a/stuff b/stuff
index ed71c82..d2c7593 100644
```

```
--- a/stuff
+++ b/stuff
@@ -2,3 +2,5 @@ This is a stuff.
 Where is the second stuff.
 Okay! The second stuff is here.
 Hmm. Jump to next.
+
+Here is master branch.
```

ORIG_HEAD 이름을 들여다보면 실제로 병합 이전의 커밋을 참조하고 있음을 알 수 있다. 병합 이후에 취소하는 것은 충돌 상태에서 취소하는 것과 크게 다르지 않다. 단지 되돌아가는 대상이 다를 뿐이다.

```
$ git reset --hard ORIG_HEAD
HEAD is now at f473009 edit stuff 'Here is master branch.'
$ git log
* f473009 | (HEAD, master) 2015-03-26 10:09:38 +0900 (49 minutes ago)
|          edit stuff 'Here is master branch.' - Choi Leejun
| * 176a7cd | (feature/1) 2015-03-26 10:08:55 +0900 (50 minutes ago)
|/            edit stuff 'Here is feature/1 branch' - Choi Leejun
*   8c302ea | 2015-03-25 15:20:59 +0900 (20 hours ago)
|\            Merge branch 'feature/1' - Choi Leejun
```

병합이 취소된 것을 알 수 있다. 이제 병합 커밋이었던 8ce7700은 쓸모없는 커밋(어떤 곳에서도 도달할 수 없는 커밋) 어떤 시점에 가비지 컬렉션에 의해 완전히 삭제될 것이다.

참고

만약 병합을 취소했다가 다시 되돌릴려면 어떻게 해야할까? 물론, git merge feature/1명령으로 다시 병합하는 방법도 있다. 하지만 기존 이력(8ce7700)을 그대로 살리지는 못한다. 병합에 대해서 잘 이해하고 있다면 해결할 수 있을 것이다.

7.3 충돌

병합이라는 것은 의미 자체에 불안함을 가지고 있다. 동일하다고 판단했던 어떤 파일이 서로 다른 브랜치 상에서 상이한 변경사항으로 인해 충돌할 가능성을 가지고 있다는 것이다.

충돌 상황을 알아보기 위해 아주 간단한 상황을 만들어보자.

```
$ git log
* b7c2575 | (HEAD, master) 2015-03-25 02:14:20 +0900 (3 seconds ago)
|           edit stuff2 'This branch is master.' - Choi Leejun
| * f5e0250 | (feature/1) 2015-03-25 02:13:28 +0900 (55 seconds ago)
|/            edit stuff2 'This branch is feature/1.' - Choi Leejun
*   2606411 | 2015-03-25 00:29:01 +0900 (2 hours ago)
|\            Merge branch 'feature/1' - Choi Leejun
| * 733ae07 | 2015-03-25 00:23:48 +0900 (2 hours ago)
| |           edit stuff 'Hmm. Jump to Next.' - Choi Leejun
...
```

앞에서 feature/1 브랜치로부터 만들어졌던 stuff2 파일을 두 브랜치에서 모두 수정한 뒤 커밋했다. 수정한 내용은 서로 다르다.

```
$ git checkout master
$ git merge feature/1
Auto-merging stuff2
CONFLICT (content): Merge conflict in stuff2
Automatic merge failed; fix conflicts and then commit the result.
```

master 브랜치로 feature/1 브랜치를 병합하려고 했으나 무언가 이상한 일이 발생했다.

앞서 충돌이 없는 상태에서 병합을 알아봤었는데 그러한 경우들과는 다른 상황이다. *Auto-merging*이라는 말이 나오는데 충돌이 없는 상태에서의 병합을 **Auto-merging**이라고 한다. 왜냐하면 충돌이 없으면 상황에 맞게 새로운 커밋이 만들어져 병합이 되든가 아니면, 두 브랜치가 원래 하나였던 것처럼 병합되기 때문이다. 'Automatic merge failed; fix conflicts and then commimt the result.'라는 것은 그러한 병합이 실패했고 충돌을 해결한 후 그 결과를 커밋해야 한다는 말이다.

Git은 고쳐야 할 충돌을 우리에게 친절하게도 알려준다. stuff2라는 파일에서 병합 충돌이 발생했음을 알려준다. 그런데 어떤 내용이 충돌인걸까?

7.3.1 충돌 지점 알아내기

일단 `git status` 명령을 통해 현재의 저장소 상태를 한번 알아보자.

```
$ 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:   stuff2

no changes added to commit (use "git add" and/or "git commit -a")
```

일단, 아직 병합되지 않은 경로(Git은 파일을 경로로 인식한다.)가 있다는 것을 알려준다. 그리고 아직 병합되지 않은 경로들을 나열해준다. 여기서 어떤 파일이 충돌했는지를 알 수가 있다.

다른 한가지 방법이 더 있다.

```
$ git ls-files --unmerged
100644 13142c21151a0ff5e9ed77687723ffe1bb6841d2 1    stuff2
100644 f438505deb93082a4eef633ec3fc915e2cf460b2 2    stuff2
100644 cfdfa07329490990d8e9c0e49ed726eefa28faa4 3    stuff2
```

`git ls-files` 명령은 인덱스 내의 상황(트리 객체 혹은 그 전 단계)을 출력해준다. `--unmerged` 플래그(혹은 `-u`)는 병합 도중 충돌 발생 시에 병합되지 않은 파일들만을 골라서 보여준다.

출력된 결과를 보니 동일한 파일(경로)에 대한 3가지 목록이 출력됐다. 출력 결과의 컬럼 중 3번째 컬럼은 충돌이 일어날 경우 1이상으로 출력된다.(충돌이 없는 경우에는 0)

```
$ git show 13142c21151a0ff5e9ed77687723ffe1bb6841d2
This is a second stuff.
The first stuff is here.
```

`git show` 명령에 blob 해시를 전달하면 해당 blob의 내용을 출력해준다. 내용을 보아하니 아마도 변경 사항 발생 이전의 blob인 것으로 추측된다.

```
$ git ls-tree 260641150ae3d8cf2592e0c9fc9a87c7c35ec40d
100644 blob ed71c828ffe5de50062e95f380c1e573b43d8284    stuff
100644 blob 13142c21151a0ff5e9ed77687723ffe1bb6841d2    stuff2
```

`git ls-tree` 명령은 특정 커밋에 대한 트리 객체(즉, 그 당시의 인덱스)를 열람할 수 있도록 해준다. stuff2라는 파일(경로)이 가지고 있는 해시가 앞에서 `git ls-files` 명령을 통해 출력했던 결과 중 첫번째 행의 해시와 동일하다는 것을 알 수 있다.

즉, `git ls-files --unmerged`의 출력 결과 중 3번째 열이 1인 것은 병합 기반의 blob인 것이다. 이 말은 병합 원본임을 뜻한다.

병합 기반에 대한 내용은 매우 중요하다. 보통 병합이라고 하면 두 브랜치 간의 변경 사항을 단순 비교해서 서로 다른 내용을 병합하기 위해 충돌을 일으킨다고 생각한다. 엄연히 이야기하자면 이것은 틀린 얘기다. 병합은 병합 기반을 통해 양쪽 브랜치에서 변경된 내용을 알아내고 충돌이 있으면 그 변경된 내용을 각각 충돌의 대상으로 표시한다. 즉, 병합 기반을 토대로 충돌이 있는지 없는지를 알아내는 것이다.

계속해서 `git ls-files`의 내용을 알아보자. 세번째 열이 2와 3으로 출력된 blob 해시를 들여다보자.

```
$ git show f438505deb93082a4eef633ec3fc915e2cf460b2
This is a second stuff.
The first stuff is here.
This branch is master.
$ git show cfdfa07329490990d8e9c0e49ed726eefa28faa4
This is a second stuff.
The first stuff is here.
This branch is feature/1.
```

2와 3에 대한 blob 해시는 양쪽 브랜치에서 변경된 blob의 해시인 것이다. 즉, 1을 기반으로 병합하려고 했는데 2에 3을 병합하려는 도중 충돌이 발생했다는 것으로 이해하면 된다.

7.3.2 충돌 내용

충돌이 발생한 내용 또한 파일(내부적으로는 blob)에 대한 것이다. 그러므로 diff를 통해 어떤 내용이 충돌했는지 알아낼 수 있다.

```
$ git diff
diff --cc stuff2
index f438505,cfdfa07..0000000
--- a/stuff2
+++ b/stuff2
@@@ -1,3 -1,3 +1,7 @@@
  This is a second stuff.
  The first stuff is here.
++<<<<<<< HEAD
 +This branch is master.
++=======
+ This branch is feature/1.
++>>>>>>> feature/1
```

`git diff` 명령은 병합 충돌 상황에서 현재 어떤 충돌이 발생했는지를 보여주는 특별한 대응을 한다. `git diff` 명령은 본래 인덱스와 작업 디렉토리를 비교하게 되는데 충돌 상황에서는 충돌을 해결하기 위한 지원을 한다.

f438505 해시와 cfdfa07 해시는 앞 절에서 본 병합 과정에 있는 blob 해시다. 즉, 충돌을 일으킨 장본인들이다. 충돌을 겪고 있는 과정에서 `git diff` 명령은 이 장본인들을 합친 내용과 인덱스를 비교한다.

그렇기 때문에 `git diff` 명령에서 두 가지 변경사항이 모두 합쳐져서 나오는 것을 볼 수 있다.

앞서 이야기한데로 `git diff`는 충돌 상황에서 약간 다른 대응을 하게 되는데 비교 자체도 다르게 동작하지만 출력에서도 조금 다른 모습을 보여준다.

플러스(+)는 보통 추가된 라인을 나타내는데 이중 플러스(++)가 표시된 라인이 있다. 이 표시가 된 줄은 양쪽 브랜치로부터 온 변경 사항과는 별도로 생긴 라인을 표시한다. 이러한 라인에 표시된 내용은 조금 독특한데 이를 3-ways **병합 표시줄(conflict marker, merge marker 혹은 텍스트 병합 드라이버)**이라고 한다.

'<<<<<<<'는 충돌 지점의 시작을 나타내며, '>>>>>>>'은 충돌 지점의 끝을 나타낸다. 그리고 '======='은 충돌 지점에 대한 대상 브랜치와 병합 브랜치의 변경 사항을 나누는 라인이다.

위와 같이 그냥 `git diff` 명령을 실행한 경우 출력 결과 중 충돌 해결이 필요한 부분을 모두 녹색으로 표시하는 것이 많은 도움이 되지 못한다. 물론, 병합 표시줄과 함께 표시되는 충돌 브랜치들의 이름 혹은 커밋 해시가 도움이 될 수는 있지만 결코 직관적이진 않다.

7.3.3 병합 헤드를 통해 충돌 내용 보기

Git에는 MERGE_HEAD라고 하는 특수한 커밋 이름이 있다. 이 이름이 가리키는 커밋은 병합되는 브랜치의 끝이다. 이 MERGE_HEAD를 이용하면 충돌 내용을 좀 더 쉽게 구분하여 볼 수 있다.

```
$ git diff MERGE_HEAD
diff --git a/stuff2 b/stuff2
index cfdfa07..b6c2d7a 100644
--- a/stuff2
+++ b/stuff2
@@ -1,3 +1,7 @@
 This is a second stuff.
 The first stuff is here.
+<<<<<<< HEAD
+This branch is master.
+=======
 This branch is feature/1.
+>>>>>>> feature/1
```

MERGE_HEAD를 기준으로 변경 사항을 비교한 것이므로 feature/1 브랜치의 변경 사항을 제외하고 master 브랜치의 변경 사항에는 플러스(+) 기호가 붙는다. 이것으로도 물론 충분할 수 있겠지만 좀 더 명확하게 보여주는 방법이 없을까?

```
$ git diff MERGE_HEAD HEAD
diff --git a/stuff2 b/stuff2
```

```
index cfdfa07..f438505 100644
--- a/stuff2
+++ b/stuff2
@@ -1,3 +1,3 @@
 This is a second stuff.
 The first stuff is here.
-This branch is feature/1.
+This branch is master.
```

위와 같이 MERGE_HEAD와 HEAD를 직접 비교하면 된다. 이렇게 하면 충돌을 해결하기 전에 어떤 부분에서 서로 충돌을 일으켰는지 한눈에 알 수 있다.

이외에도 충돌 지점을 관찰하는 방법은 많이 있다. `git log`를 이용하는 방법도 있을 것이고, 별도의 도구를 사용해도 된다. 어떤 것을 사용할지는 개인적인 주관에 의해 결정하면 된다. 하지만 명확하게 충돌을 관찰할 수 있어야 한다.

7.3.4 충돌을 해결하여 병합 끝내기

vim이든 아니면 주로 사용하는 편집기가 됐든 텍스트를 편집할 수 있는 도구만 있으면 충돌을 해결할 수 있다. 도구 중에는 텍스트 편집 기능과 더불어 몇 번의 클릭으로 충돌을 짜집기할 수 있는 도구도 있다. 매우 편리한데 그만큼 위험성도 뒤따른다.

여기서는 vim 혹은 일반적인 텍스트 편집기를 이용해 충돌을 해결하는 방법을 알아보도록 하겠다.

```
1 This is a second stuff.
2 The first stuff is here.
3 <<<<<<< HEAD
4 This branch is master.
5 =======
6 This branch is feature/1.
7 >>>>>>> feature/1
```

vim에서 줄번호를 출력하는 방법은 `:set number`이다.

충돌이 발생하면 충돌이 발생한 지점의 전체 라인에는 위와 같이 병합 표시줄을 통해 충돌이 표시된다. 이런 병합 표시줄를 충돌이 발생한 파일들의 모든 지점에서 제거해야 충돌이 해결되고 병합이 이루어진다.

병합 표시줄은 Git 내부적으로는 어떠한 영향력도 없다. 물론 병합 표시줄이 한 개라도 존재한다면 병합은 이루어지지 않지만 그저 텍스트에 지나지 않는다.

충돌을 해결할 때 결정해야 하는 사항은 병합 표시줄을 얼마나 빨리 없애느냐 혹은 어떻게 없애느냐가 아니라 실제로 충돌된 지점에서 필요한 것은 무엇인가이다.

예제에서는 편이를 위해 아무런 의미도 없는 텍스트를 사용했지만, 실제로는 코드 라인이거나 어떤 의미 있는 글의 일부분일수도 있다. 그러한 경우 그 한줄은 매우 중요한 역할을 하는 경우가 대부분이므로 함부로 삭제하거나 첨가하는 것이 부담스러울 수 있다. 충돌의 원인이 되는 코드 작성자들이 모두 함께 의논해야만 할 것이다.

병합 표시줄을 모든 충돌 지점에서 제거했다면 이제 병합할 준비가 끝났을 것이다. 저장한 뒤에 커밋을 진행하면 된다.

```
$ git add stuff2
$ git commit
```

커밋 명령을 실행하면 다음과 같이 커밋 로그 메시지를 위한 편집기가 실행될 것이다.

```
 Merge branch 'feature/1'

Conflicts:
        stuff2
#
# 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 '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified:   stuff2
#
```

일반적인 병합 커밋 로그 메시지와는 조금 다른데, 충돌이 발생한 파일에 대한 로그를 함께 보여준다.

병합 커밋의 로그 메시지는 경우에 따라 굉장히 중요할 수 있다. 충돌 지점이 많으면 많을수록, 수정이 많이 이루어질수록 더욱 더 중요해진다. 왜냐하면 양쪽 브랜치에서 수정된 내용 이외의 내용이 더 추가되거나 삭제될 수도 있기 때문이다.

```
$ git commit
[master 8c302ea] Merge branch 'feature/1'
*   8c302ea | (HEAD, master) 2015-03-25 15:20:59 +0900 (4 minutes ago)
|\            Merge branch 'feature/1' - Choi Leejun
| * f5e0250 | (feature/1) 2015-03-25 02:13:28 +0900 (13 hours ago)
| |           edit stuff2 'This branch is feature/1.' - Choi Leejun
* | b7c2575 | 2015-03-25 02:14:20 +0900 (13 hours ago)
|/            edit stuff2 'This branch is master.' - Choi Leejun
```

커밋 로그 메시지를 저장하고 나면 병합이 완료된다. 만약 충돌이 한개라도 해결되지 않았다면 병합되지 않는다.

7.4 재귀 병합 전략

참고

이 장은 plasticSCM의 블로그 포스트인 'Merge recursive strategy'를 토대로 작성되었다. 번역자의 짧은 영어 실력으로 인해 오역이 있을 수도 있으니 이상한 부분은 알려주기 바란다.

Git 사용자라면 아마도 *recursive strategy*라는 것에 대해 들어봤을 것이다. 이것은 Git이 두 개의 브랜치를 병합할 때 사용하는 기본 알고리즘이다.

그런데, 이 알고리즘은 어떻게 동작하고 또 왜 좋다고들 할까?

7.4.1 해결(resolve) 전략

병합하고 싶은 두 개의 브랜치가 있다고 하자. 병합 과정에 필요한 요소들에는 다음과 같은 요소들이 있다.

- 병합 브랜치(source)
  - 목적 브랜치를 병합하려는(from) 변경사항이다.
  - 아래 그림에서 커밋 4가 병합 브랜치이다.
- 병합 대상 브랜치(destination)
  - 병합 브랜치로 병합되는(to) 변경사항이다.
  - 아래 그림에서 커밋 7이 병합 대상 브랜치이다.
- 공통 조상(common ancestor)
  - 병합 브랜치와 대상 브랜치으로부터 가장 가까이에 있는 공통 조상 커밋을 말한다.
  - 병합 기반(merge base)라고도 한다.
  - 아래 그림에서 커밋 1이 병합 기반이다.

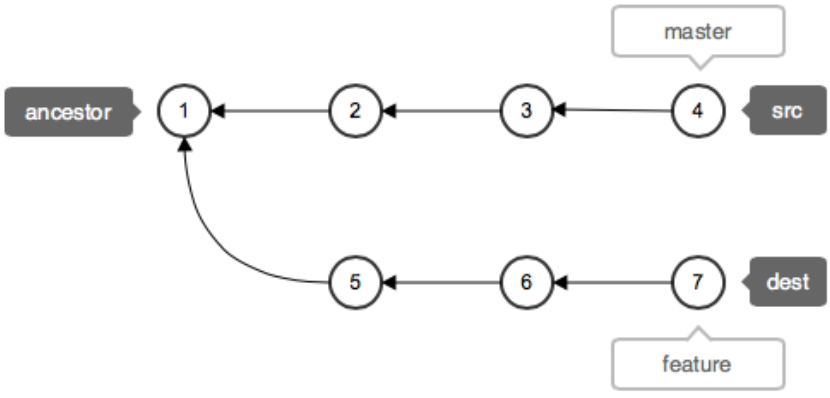

그림 7-1 서로 다른 두 브랜치

위 그림에서 두 브랜치 간의 병합은 두 브랜치의 병합 기반인 커밋 1을 토대로 이루어진다.

이렇게 두 브랜치 간의 병합 기반이 한 개인 3-ways 병합 전략을 **해결(resolve) 전략**이라고 한다.

7.4.2 언제 재귀 병합이 필요한가?

병합 기반이 두 개 이상일때도 있다.

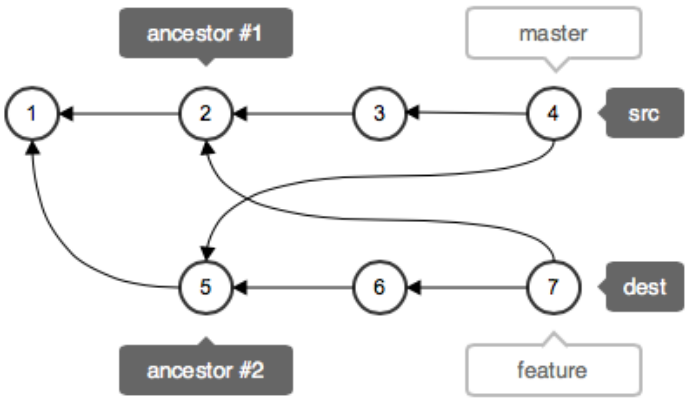

그림 7-2 두 브랜치의 병합

위 그림처럼 이미 서로가 병합을 한 이력이 있는 경우가 그렇다. 이렇게 될 경우 커밋 2와 커밋 5는 양측 브랜치의 병합 기반이 된다. 이런 경우라면, 해결 전략으로는 병합이 어려워진다. 좀 더 공통 조상에 가까운 기반을 찾아야 하는데 이럴 경우 득보다는 실이 더 크다.

위와 같은 경우가 만들어지는 경우가 흔한지 흔하지 않은지는 중요하지 않다. 저런 이력은 만들어질 가능성이 있다. 예를 들어, 커밋이 상당히 진행된 상태에서 서로를 완전히 병합하지 않고도 어떤 시점의 변경 사항들을 가져오고 싶어 병합을 하는 경우이다. 이렇게 병합이 양쪽으로 이루어져 있는 경우를 criss-cross 병합이라고 부른다.

7.4.3 재귀 병합은 어떻게 이루어지나?

결국 문제는 기반이 두 군데라는 것에 문제가 있는 것이다. 이를 하나로 만드는 것에 해결법이 있다.

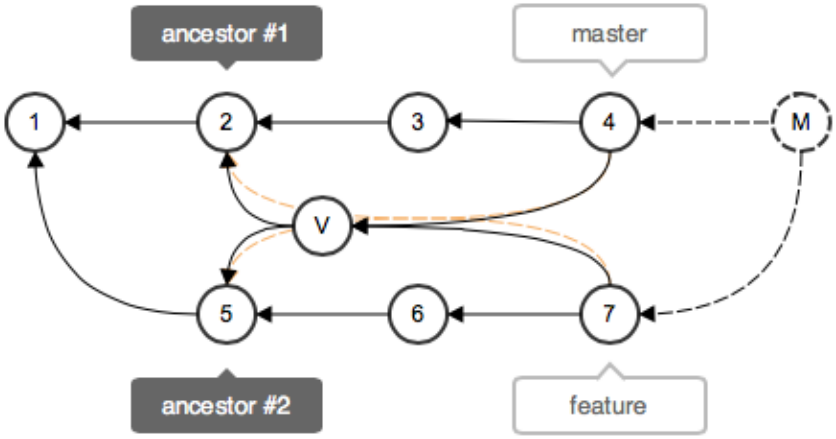

그림 7-3 재귀 병합

먼저, 병합 기반인 커밋 2와 커밋 5을 서로 병합한다. 둘 간의 병합이 이루어지면 커밋 V처럼 가상의 커밋이 생기게 된다. 바로 이 가상의 커밋을 실제 병합의 병합 기반으로 사용하는 것이 해결법의 핵심이다.

병합 브랜치와 대상 브랜치는 이 가상 커밋을 기반으로 병합을 하게 되는 것이다.

7.4.4 예를 들자면 이렇게 동작한다.

다음 예제는 `foo.c`라는 파일을 사용할 것이다. 이 파일은 다음과 같은 세 줄의 내용으로부터 출발할 것이다.

```
b
c
d
```

이와 같은 내용을 앞으로 다음과 같이 표시할 것이다. 이것은 예시의 간편함을 위한 방법이다.

```
변경유형: bcd
```

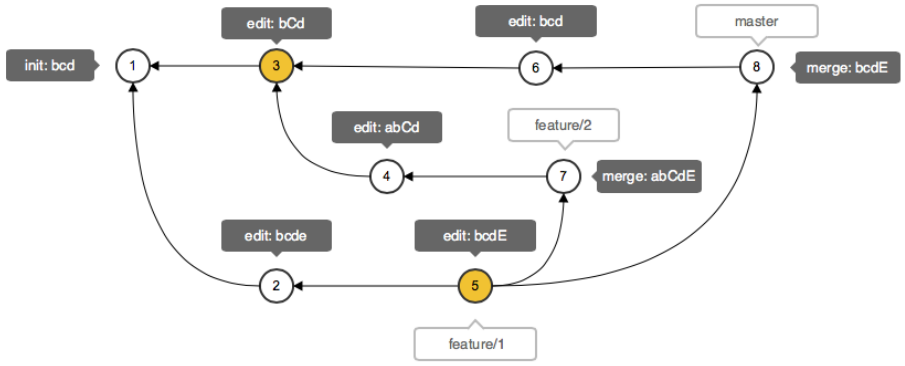

그림 7-4 복잡한 브랜치

위 다이어그램에서 보다시피 4번과 5번이 병합되어 7번 커밋이 이루어졌고, 5번과 6번이 병합되어 7번 8번커밋이 이루어진 상태이다.

이 상태에서 7번과 8번을 병합해야 하는 경우 둘의 공통 조상은 3번과 5번이 된다.(다이어그램에서 노란색 커밋들인데 도달할 수 있는 경로를 잘 생각해보면 된다.)

3-ways 자동 병합이 된다면 각 라인의 문자는 무엇이 되어야 하는가?

첫번째 줄은 a가 될 것이다. 이것은 4번 커밋에서 시작되어 7번 커밋으로 병합되었으므로 당연히 a가 첫번째 줄에 올 것이다.

두번째 줄은 모든 브랜치로부터 온 것으로 이견이 없을 것이다. b가 된다.

세번째 줄은 좀 애매해 보인다. 양쪽 공통 조상으로부터 참조된 값을 보면 서로가 다르다.(즉, 충돌이다.) 일단 넘어가자.

네번째 줄도 두번째 줄과 마찬가지. d가 된다.

다섯번째 줄은 E가 될텐데 이는 3번과 4번의 병합 시 6번 커밋으로 병합되었다. master 브랜치로 새로 추가되는 줄이므로 문제가 없다.

또 다른 버전 관리 시스템인 경우 Mercurial의 경우 세번째 줄은 C가 된다. 이는 한 개 이상의 공통 조상이 존재할 경우 가장 깊은 것을 선택하는 알고리즘 때문이라고 한다.

Git이 기본 알고리즘을 선택한 재귀 알고리즘의 경우 어떻게 처리하는지 알아보자.

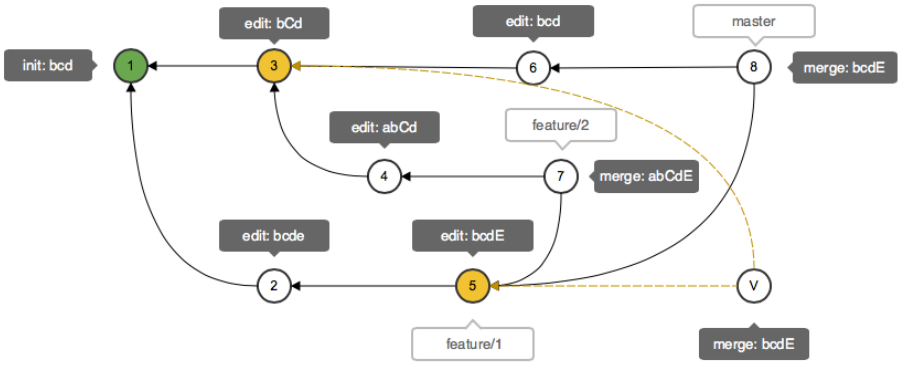

그림 7-5 재귀 병합의 예

앞에서 설명했듯이 재귀 알고리즘의 경우 공통 조상의 병합 커밋을 가상으로 만들어 이를 병합 기반으로 사용한다고 했다. 다이어그램에서 7번 커밋과 8번 커밋의 공통 조상 3번 커밋과 5번 커밋을 병합하면 bcdE가 된다. 이는 둘의 병합 기반이 커밋 1번이기 때문이다.

이제 문제는 간단해졌다. 커밋 7번과 8번을 병합할 때 문제가 되었던 c와 C 중 어떤 것을 선택할지는 그들의 병합 기반인 커밋 V에 의해 결정되므로 c가 최종 선택된다.

7.4.5 그래서 왜 좋은건가?

브랜칭과 병합을 사용해야 하는데 마땅히 좋은 방법(알고리즘)이 없다면, 어쩌면 어떤 경고도 없이 깨진 파일을 결과로 받을 수도 있다. Git은 정확하게 그러한 경우를 처리해내지만, Mercurial은 결과를 망가뜨릴 수 있고, SVN이나 다른 버전 관리 도구들은 모든 결정 사항들을 사용자에게 맡겨 버린다.

참고

Git이 기본적으로 재귀 병합 알고리즘을 사용하지만 꼭 강제하는 것은 아니다. 병합 시에 사용하고자 하는 알고리즘을 선택할 수 있는 플래그를 제공한다.

8. 커밋 히스토리 수정하기

이 장에서는 커밋의 히스토리를 변경하여 궁극적으로 변경 사항을 변경하는 방법에 대해 설명한다.

커밋 자체를 수정하는 방법은 `--amend` 플래그를 이용해 커밋하는 것이다. 커밋을 수정하는 것과 히스토리를 수정하는 것은 다르다. 커밋 한 개를 취소하는 것은 한 개 이상을 취소하는 것과 크게 다르지 않다. 그리고 커밋 시퀀스를 재배치하는 것도 결국은 히스토리가 변하게 되는 것이다.

커밋 히스토리를 수정하는 방법에는 여러 가지가 있다.

8.1 Git reset 으로 커밋 취소하기

`git reset` 명령은 저장소와, 작업 디렉토리 혹은 인덱스를 저장소에 저장되어 있는 임의의 상태로 변경한다.

그림 8-1 git reset

`git reset` 명령이 실행되면 위와 같이 진행된 커밋 이전의 커밋으로 HEAD가 이동된다. 즉, 위 다이어그램에서 점선으로 표시된 커밋들은 모두 무의미한 커밋이 되어버린다.

여기서 `git reset`을 처음 알아보는 사람이라면 한 가지 의문이 들어야 한다. HEAD가 바뀌므로 저장소의 상태도 HEAD가 가르키고 있는 커밋의 저장소 상태로 변경되어야 하는데, 그렇다면 인덱스와 작업 디렉토리는 어떻게 되는 것일까?

위 다이어그램에서 커밋 4번일 때 만약 어떤 파일이 수정되었거나 수정된 뒤 인덱스에 추가된 상태에서 `git reset` 명령이 실행된다면 어떻게 되는걸까?

```
$ git ls-files --stage
100644 38c98dce1e3e6f01e78bd038d290978389731a0a 0    stuff
100644 ff42a0c9738b3bb14669e6a4a79736682dc46892 0    stuff2
# stuff 파일을 수정하고 커밋했다.
$ vim stuff
$ git add stuff
$ git commit -m "stuff"
$ git log
* c71b7ff | (HEAD, master) 2015-03-29 17:12:18 +0900 (22 seconds ago)
|          stuff - Choi Leejun
*   8ce7700 | (feature/1) 2015-03-26 10:48:55 +0900 (3 days ago)
|\           Merge branch 'feature/1' - Choi Leejun
| * 176a7cd | 2015-03-26 10:08:55 +0900 (3 days ago)
| |          edit stuff 'Here is feature/1 branch' - Choi Leejun
...
$ git ls-files --stage
100644 a934046a85edc16dd196181f8326ed35deb3f903 0    stuff
100644 ff42a0c9738b3bb14669e6a4a79736682dc46892 0    stuff2
```

위와 같은 커밋 히스토리와 인덱스 상태에서 HEAD 커밋을 취소해보자.

```
$ vim stuff
# 수정된 stuff 파일을 인덱스에 추가한다.
$ git add stuff
# stuff2 파일은 수정하지만 인덱스에 추가하지는 않는다.
$ vim stuff2
$ git ls-files --stage
100644 e3e8dc2db25c86b737f7e4986e0854cc3f6b498b 0    stuff
100644 ff42a0c9738b3bb14669e6a4a79736682dc46892 0    stuff2
```

우선, 두 개의 파일 중 stuff 파일은 수정한 뒤에 인덱스에 추가하고 stuff2 파일은 수정하고 그대로 유지하자. 즉, stuff 파일은 커밋 준비 단계이고 stuff2 파일은 스테이징이 필요한 상태다.

```
$ git reset HEAD~1
Unstaged changes after reset:
M    stuff
M    stuff2
$ git ls-files --stage
100644 38c98dce1e3e6f01e78bd038d290978389731a0a 0    stuff
100644 ff42a0c9738b3bb14669e6a4a79736682dc46892 0    stuff2
```

커밋을 이전 커밋 상태로 되돌리기 위해서 위와 같이 `git reset` 명령에 `HEAD~1`이라는 상대적인 커밋 이름을 전달했다.

`git ls-files` 명령을 통해 stuff 파일과 stuff2 파일에 대한 blob 해시를 확인해보자. 두 파일 모두 최초의 상태로 돌아갔다. 그런데 파일의 내용을 확인해보면 두 파일이 마지막으로 수정했던 내용 그대로를 유지하고 있음을 알 수 있다.

특히 stuff2 파일에 대한 blob 해시는 한번도 변한적이 없다.

어떻게 된 일일까?

8.1.1 git reset 옵션

`git reset`은 필수 옵션이 있는데, 바로 리셋 모드(reset mode)이다. 리셋 모두 중 가장 중요한 3 가지에 대해서 알아보자.

- *--mixed*
  - 3 개의 모드 중 기본이 되는 모드이다. 앞에 예제에서 `git reset` 명령에 옵션을 주지 않았는데 이 모드가 기본적으로 적용되기 때문이다.
  - **mixed** 모드는 작업 디렉토리는 유지하면서 인덱스를 HEAD와 함께 되돌린다.
- *--hard*
  - **hard** 모드는 조심해서 사용해야 한다. 작업 디렉토리와 인덱스를 모두 유지하지 않고 이전 커밋으로 HEAD를 되돌리기 때문이다.
  - 작업하던 내용 모두를 정리하고자 할 때만 의도적으로 사용해야 한다.
- *--soft*
  - 현재의 인덱스 상태와 작업 디렉토리 내용을 그대로 보존한채 커밋만 취소할 경우 사용한다.

- 사용 방식에 따라 `git commit --amend`와 비슷하게 사용할 수 있다.

참고

git reset 명령으로 취소된 커밋 집합들은 저장소에 어느 정도의 변경 사항이 진행되면 가비지 컬렉션에 의해 완전히 삭제되게 된다. 완전히 삭제되기 전에는 git reflog 명령과 다른 명령을 통해 커밋을 복구할 수 있다.

8.2 git revert 로 커밋 취소하기

앞에서 알아본 `git reset`의 경우, 취소된 커밋이 이력에 남지 않는다. 즉, HEAD를 이전 상태로 되돌리고 과거부터 다시 시작한다고 보면 된다.

하지만, `git revert` 명령의 경우, 취소에 대한 이력을 남긴다. 즉, 취소를 커밋으로 처리한다.

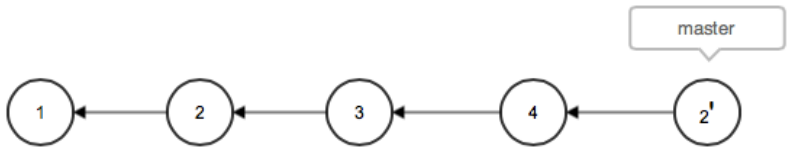

그림 8-2 git revert

커밋 2를 취소하려고 하면 커밋 2에 대한 내용을 되돌려야 하는데 `git reset`을 이용하는 경우 3번과 4번 커밋에서 진행된 변경 사항들을 그대로 유지하는 것이 쉽지 않다. 또, 이미 커밋들이 공유된 상태라면 더더욱 `git reset`을 이용해서는 안 된다.

```
$ git log
* c82b8a4 | (HEAD, master) 2015-03-29 22:37:47 +0900 (22 seconds ago)
|          edit stuff 'some stuff.' - Choi Leejun
* 26c8229 | 2015-03-29 22:27:15 +0900 (11 minutes ago)
|          edit stuff 'We may need to revert here.' - Choi Leejun
* 7c8815d | 2015-03-29 22:26:02 +0900 (12 minutes ago)
|          Next step on stuff - Choi Leejun
* e8f81f2 | 2015-03-29 22:24:18 +0900 (14 minutes ago)
|          New start - Choi Leejun
```

위와 같이 커밋 히스토리가 있다고 하자.

커밋 26c8229가 잘못된 내용을 가지고 있어서 이를 수정해야 할 것 같다. 그러나 이미 다음 커밋이 진행된 상태이고 이미 다른 사람들에게 공유가 된 상태라고 가정하자. 그러므로 `git revert`를 이용해 커밋 26c8229의 변경 사항을 히스토리 상에서 되돌려볼 것이다.

```
$ git revert 26c8229
error: could not revert 26c8229... edit stuff 'We may need to revert here.'
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

`git revert` 명령으로 커밋 26c8229를 되돌려고 하자 충돌이 발생했다는 내용을 보여준다. 충돌이 발생한 이유는 커밋 c82b8a4에서 동일한 파일에 변경 사항이 발생했기 때문이다.

커밋 c82b8a4에서 발생한 변경 사항까지 **자동으로** 되돌릴수는 없기 때문에 이를 사용자에게 어떻게 처리할지를 묻는 것이다.

```
$ vim stuff
$ git add stuff
$ git commit
Revert "edit stuff 'We may need to revert here.'"

This reverts commit 26c82296686065adf94ea2c13615d2670ab11608.

Conflicts:
        stuff

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# You are currently reverting commit 26c8229.
#
# Changes to be committed:
#       modified:   stuff
#
```

위와 같이 충돌이 발생한 stuff의 내용을 정리해주고(병합 시의 충돌과 처리 방식은 동일하다.) 커밋까지 진행하면 위와 같이 (병합할 때와 비슷하게) revert에 대한 커밋 메시지를 자동으로 출력해준다. 커밋 메시지를 저장하면 *revert*는 완료된다.

8.3 체리 피킹

*cherry picking*이라는 말은 여러 분야에서 사용되는데, 체리나무 열매를 따는 행위가 어원인 것 같긴 하지만 Git에서 사용하는 cherry-pick과는 좀 거리가 있어 보인다. 오히려 과학계나 의학계에서 사용되는 의미가 더 와닿는다.

과학계나 의학계에서는 아직 검증되지 않은 이론이나 불확실한 근거에 대해서 자신에게 유리한 검사 결과나 사례만을 취사 선택하는 것을 *cherry picking*이라고 한다.

Git에서는 이와 유사한 의미로써 cherry-pick이라는 용어를 사용하는 것 같다. 물론 위 내용에서는 부정적인 의미이지만 Git의 cherry-pick은 좋은 용도로 사용될 수 있다.

```
$ git log
* a1505d7 | (HEAD, master) 2015-03-30 00:53:23 +0900 (2 seconds ago)
|           4 - master - Choi Leejun
* e0387ff | 2015-03-30 00:53:05 +0900 (20 seconds ago)
|           3 - master - Choi Leejun
| * a7892d7 | (feature) 2015-03-30 00:52:40 +0900 (45 seconds ago)
| |           3 - Choi Leejun
| * be001b6 | 2015-03-30 00:52:19 +0900 (66 seconds ago)
|/            2 - Choi Leejun
* 2eff9df | 2015-03-30 00:50:05 +0900 (3 minutes ago)
|           1 - Choi Leejun
* 297050d | 2015-03-30 00:48:50 +0900 (5 minutes ago)
            initial commit - Choi Leejun
```

위와 같이 두 브랜치가 있고 어느 정도의 변경사항이 각각 진행된 상태라고 가정하자. master 브랜치에서 버그가 발견되었는데 feature 브랜치에서는 이미 그 내용에 대한 대비책을 마련해놓은 상태라고 가정하자.(커밋 be001b6) 그런데 feature 브랜치를 아직 병합하기에는 문제가 좀 있는 상황이라고 하자. 그렇다고 해서 이미 feature 브랜치에서 해결된 사항을 또 다시 master에 구현하기는 마음에 들지 않는다.

이 때 필요한 것이 바로 `git cherry-pick` 명령이다. 원하는 변경 사항만 취사 선택하는 것이다.

```
# cherry pick하려는 브랜치로 체크아웃해야 한다.
$ git checkout master
# 충돌은 발생할 수도 있고 안할수도 있다.
$ git cherry-pick feature~1
error: could not apply be001b6... 2
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
$ vim stuff
$ git add stuff
$ git commit
[master 9140f36] 2
 1 file changed, 1 insertion(+)
$ git log
* 9140f36 | (HEAD, master) 2015-03-30 00:52:19 +0900 (11 minutes ago)
|           2 - Choi Leejun
```

```
* a1505d7 | 2015-03-30 00:53:23 +0900 (9 minutes ago)
|          4 - master - Choi Leejun
* e0387ff | 2015-03-30 00:53:05 +0900 (10 minutes ago)
|          3 - master - Choi Leejun
| * a7892d7 | (feature) 2015-03-30 00:52:40 +0900 (10 minutes ago)
| |          3 - Choi Leejun
| * be001b6 | 2015-03-30 00:52:19 +0900 (11 minutes ago)
|/           2 - Choi Leejun
* 2eff9df | 2015-03-30 00:50:05 +0900 (13 minutes ago)
|          1 - Choi Leejun
* 297050d | 2015-03-30 00:48:50 +0900 (14 minutes ago)
             initial commit - Choi Leejun
```

master 브랜치는 '2'가 빠졌는데 이를 feature 브랜치로부터 가져오기 위해 커밋 be001b6를 *cherry picking*했다. 이로 인해 복잡한 고민을 할 것 없이 변경 사항을 다른 브랜치로부터 가져와 적용했다.

물론, cherry picking된 변경 사항들은 완전히 새로운 커밋으로 적용된다. 위 예제에서 커밋 be001b6의 변경 사항을 master 브랜치에 적용하고 나니 커밋 9140f36이 새로 만들어졌다.

```
$ git log
* a2c6ac3 | (HEAD, feature) 2015-03-30 01:09:15 +0900 (2 seconds ago)
|          8 - Choi Leejun
* 027c3cf | 2015-03-30 01:08:25 +0900 (52 seconds ago)
|          7 - Choi Leejun
* 4f37450 | 2015-03-30 01:08:00 +0900 (77 seconds ago)
|          5 - Choi Leejun
* c5a00c0 | 2015-03-30 01:07:22 +0900 (2 minutes ago)
|          4 - Choi Leejun
* a7892d7 | 2015-03-30 00:52:40 +0900 (17 minutes ago)
|          3 - Choi Leejun
* be001b6 | 2015-03-30 00:52:19 +0900 (17 minutes ago)
|          2 - Choi Leejun
| * 9140f36 | (master) 2015-03-30 00:52:19 +0900 (17 minutes ago)
| |          2 - Choi Leejun
| * a1505d7 | 2015-03-30 00:53:23 +0900 (16 minutes ago)
| |          4 - master - Choi Leejun
| * e0387ff | 2015-03-30 00:53:05 +0900 (16 minutes ago)
|/           3 - master - Choi Leejun
* 2eff9df | 2015-03-30 00:50:05 +0900 (19 minutes ago)
|          1 - Choi Leejun
* 297050d | 2015-03-30 00:48:50 +0900 (20 minutes ago)
             initial commit - Choi Leejun
```

이전 상태를 이어서 feature 브랜치에서 좀 더 많은 내용이 진행되었고 이 중 몇 가지를 master 브랜치에 싣고 싶어졌다. 전체가 아닌 일부인데 이번에는 그 변경 사항이 하나의 커밋이 아니라 두 개의 커밋, 4f37450와 027c3cf에 걸쳐 있다. 어떻게 해야할까?

```
$ git checkout master
$ git cherry-pick feature~2..feature~1
```

```
error: could not apply 027c3cf... 7
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
$ git diff HEAD CHERRY_PICK_HEAD
diff --git a/stuff b/stuff
index 94ebaf9..119d9e8 100644
--- a/stuff
+++ b/stuff
@@ -2,3 +2,5 @@
 2
 3
 4
+5
+7
```

`git cherry-pick` 명령은 커밋의 범위를 지원한다. 그래서 원하는 범위의 커밋 집합을 가져와 적용할 수 있다. 이 때에도 모든 커밋들은 새로운 커밋들로 만들어진다.

위 예제에서 `git diff` 명령과 함께 CHERRY_*PICK*_HEAD라는 새로운 참조 이름을 사용했는데 이 참조 이름은 `git cherry-pick` 명령 직후에만 사용 가능하며 cherry picking된 제일 마지막 커밋을 참조하게 된다. 그러므로 `git diff` 명령을 통해 어떤 부분에서 변경 사항이 생길 것인지를 예측할 수 있고 충돌을 푸는데 도움을 받을 수 있다.

앞의 예제와 마찬가지로 충돌을 푼 뒤에 커밋하면 cherry picking은 완료된다.

8.4 리베이스

8.4.1 리베이스는 병합과 무엇이 다른가?

참고

rebase에 대해서 설명하는 많은 매체들이 어떻게 rebase해야 되는지와 왜 rebase해야 되는지에 대해서를 먼저 혹은 제일 많이 다룬다. 물론 병합과 rebase가 어떻게 다른지에 대해서 다루는 곳도 많다. 하지만 대부분 병합을 해야할지 아니면 rebase 해야할지의 대결 구도로 설명한다.(물론 이 주제도 중요하다.) 나는 조금 다른 각도로 봤다.

먼저, Git에서 브랜치를 사용한다면 필수적인 것이 바로 병합이다. 그 중에서도 대표 전략 두 가지, fast-forward 병합과 3-ways 병합이 있다. 이 두 가지를 모른다면 Merging을 읽고 오기 바란다.

아주 간단하게 두 병합 전략에 대해서 설명해보도록 하겠다.

fast-forward 병합은 두 브랜치를 병합해야 할 때 분기 이후 한 브랜치의 변경 사항이 없을 경우 가능한 병합이다.

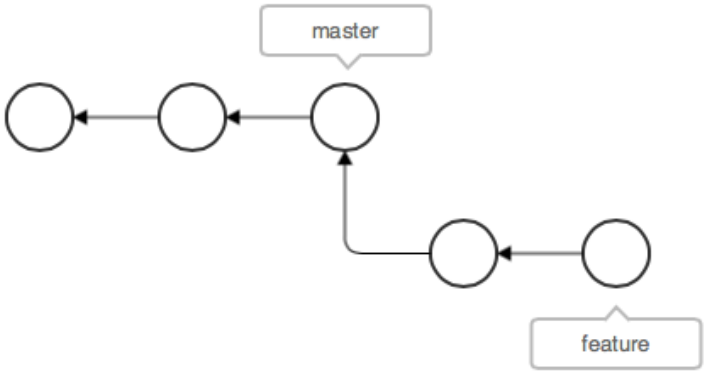

그림 8-3 fast-forward 병합이 가능한 상태

위 그래프에서 master 브랜치로 feature 브랜치를 병합해야 하는데 병합할 경우 master 브랜치가 feature 브랜치의 HEAD로 이동하게 된다.

3-ways 병합은 두 브랜치를 병합해야 할 때 분기 이후 두 브랜치 모두 변경 사항이 있을 경우에 선택되는 기본 병합 전략이다.

fast-forward 병합은 `--no-ff` 플래그를 전달해 3-ways 병합처럼 만들 수가 있다. 이는 병합하는 브랜치의 이력을 의도적으로 남기려고 사용한다.

그렇다면 그 반대의 경우는 없을까?

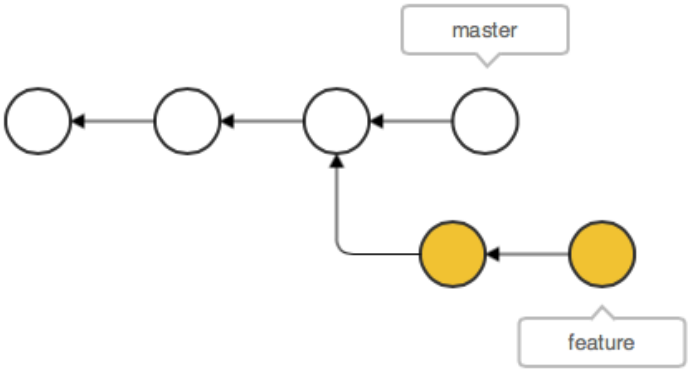

그림 8-4 fast-forward 병합이 불가능한 상태

위 그래프에서 master 브랜치의 변경사항과 feature 브랜치의 변경사항 모두이 존재함으로써 두 브랜치는 병합하고자 한다면 3-ways 병합을 해야 하는 상황이다.

그런데 feature 브랜치의 이력을 커밋 히스토리 상에 남기고 싶지 않아졌다. 만약 master 브랜치의 변경사항이 없었다면 *fast forwarding*이 가능했겠지만 불가능한 상황이다.

이럴 때 필요한 것이 바로 rebase다.

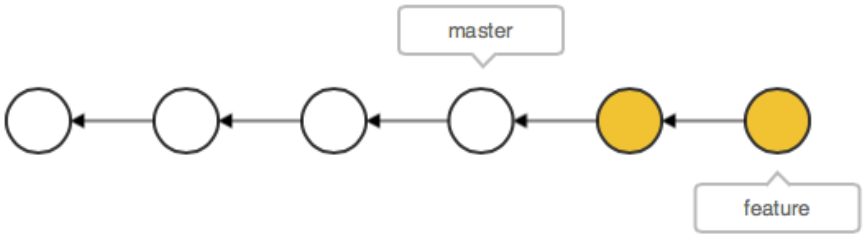

그림 8-5 리베이스 결과

feature 브랜치의 커밋들이 모두 master 브랜치의 끝으로 이동했다. 물론 이 과정에서 충돌이 발생할 수는 있다. 그리고 rebase된 뒤에 master 브랜치를 *fast forwarding*해야 한다.

8.4.2 아주 간단한 예제

rebase를 위한 예제는 아주 간단할 수도 있고, 아주 복잡할 수도 있다. 여기서는 아주 간단한 예제, 즉 앞에서 보았던 커밋 히스토리와 거의 유사한 상태에서 rebase 예제를 살펴보도록 하자.

```
$ git log
* 83df48b | (HEAD, master) 2015-03-30 12:09:11 +0900 (1 second ago)
|           3 - Choi Leejun
* 91017ee | 2015-03-30 12:08:53 +0900 (19 seconds ago)
|           1 - Choi Leejun
| * 5778949 | (feature) 2015-03-30 12:08:28 +0900 (44 seconds ago)
| |           4 - Choi Leejun
| * 3751dfb | 2015-03-30 12:08:14 +0900 (58 seconds ago)
|/            2 - Choi Leejun
* cce20a4 | 2015-03-30 12:07:25 +0900 (2 minutes ago)
            initial commit - Choi Leejun
```

위와 같이 첫번째 커밋 이후에 두 브랜치에서 각각 2개의 커밋이 진행된 상태를 가정하자. 파일은 한 개만 있으며 각 커밋에서의 로그 메시지는 파일에 추가된 변경사항이다.

feature 브랜치의 커밋들으 master 브랜치로 병합하고 싶은데 feature 브랜치의 이력을 남기기는 싫다. 그런데 이미 master 브랜치에서 변경 사항이 진행됐으므로 fast-forward 병합도 불가능한 상태이다. rebase 해보자.

rebase를 진행하기 전에 프로젝트 디렉토리를 복사해놓을 것을 추천한다.

```
$ git checkout feature
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: 2
Using index info to reconstruct a base tree...
M    stuff
Falling back to patching base and 3-way merge...
Auto-merging stuff
CONFLICT (content): Merge conflict in stuff
Failed to merge in the changes.
Patch failed at 0001 2
The copy of the patch that failed is found in:
   /Users/devcken/Projects/my-project/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

rebase는 어떤 브랜치에서 실행되느냐에 따라 그 대상이 달라진다. 앞선 예제에서 feature 브랜치를 master 브랜치로 rebase 했으므로 먼저 feature 브랜치로 체크아웃한 뒤에 rebase를 실행했다.

예제와 동일하게 커밋을 진행했다면 위와 같이 충돌이 발생했을 것이다. 동일한 파일을 수정한 뒤에 rebase했기 때문인데 자동 병합이 실패했고 그래서 사용자에게 충돌에 대한 해결을 묻는 것이다.

여기서 세 가지 갈래가 나오는데 첫 번째는 충돌을 해결하고 `git rebase --continue`를 실행하는 것이다. 다음은 `git rebase --skip`인데, skip할 경우 진행되던 rebase를 멈춘 상태에서 중지한다. 이 명령은 아주 특별한 경우가 아니면 실행하지 않는 것이 좋다. 결정을 미루는 것인데 언젠가는 문제가 될 소지가 있으므로 되도록이면 충돌을 해결하자.

만약 rebase를 취소하려면 `git rebase --abort` 명령을 실행해야 한다. 모든 rebase 작업을 취소하고 이전 상태로 돌아가게 된다.

```
$ vim stuff
$ git add stuff
```

stuff 파일에서 발생한 충돌을 해결하다보면 무언가 이상한 점을 느꼈을 것이다. master 브랜치에서 추가했던 '4'이 빠져있는 것을 알 수 있다. 일단 현재의 충돌만 해결을 하고 `git rebase --continue` 명령을 실행해보자.(충돌이 해결된 파일은 인덱스에 추가해줘야 한다.)

```
$ git rebase --continue
Applying: 2
Applying: 4
Using index info to reconstruct a base tree...
1
M    stuff
Falling back to patching base and 3-way merge...
Auto-merging stuff
CONFLICT (content): Merge conflict in stuff
Failed to merge in the changes.
Patch failed at 0002 4
The copy of the patch that failed is found in:
   /Users/devcken/Projects/my-project/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

그렇다면 다시 한번 위와 같이 충돌이 발생하는 것을 알 수 있다. 충돌한 부분을 확인해보니 이전 상황에서 빠져있던 '4'이 포함되어 있는 것을 알 수 있다.

충돌을 해결하고 마저 rebase해보자.

```
$ git rebase --continue
Applying: 4
$ git log
* 05ff9b1 | (HEAD, feature) 2015-03-30 12:08:28 +0900 (76 minutes ago)
|           4 - Choi Leejun
* 38d5967 | 2015-03-30 12:08:14 +0900 (76 minutes ago)
|           2 - Choi Leejun
* 83df48b | (master) 2015-03-30 12:09:11 +0900 (75 minutes ago)
|           3 - Choi Leejun
```

```
* 91017ee | 2015-03-30 12:08:53 +0900 (76 minutes ago)
|          1 - Choi Leejun
* cce20a4 | 2015-03-30 12:07:25 +0900 (77 minutes ago)
            initial commit - Choi Leejun
```

모든 rebase가 완료된 것을 알 수 있다.

앞에서 충돌이 두 번 발생했고 '4'이 포함되지 않는 경우도 있었다. 이것은 rebase가 커밋의 개수만큼 단계적으로 실행된다는 증거이다.

rebase 과정 중에 나오는 메시지를 보아도 알 수 있는데 `Applying:`으로 시작하는 라인을 보면 현재 적용되고 있는 커밋의 로그 메시지가 출력된다. 그리고 만약 충돌이 발생한다면 Git은 rebase를 즉시 중지하고 충돌 해결을 사용자에게 요청하는 것이다.

위 예제에서 한 파일에 대해 master 브랜치에서는 각각 1과 3을 추가한 커밋을, 그리고 feature 브랜치에서는 각각 2와 4를 추가한 커밋을 진행하고 rebase를 진행했으므로 두 번의 충돌이 발생할 수 밖에 없는 것이다.

rebase의 결과를 보면 feature 브랜치가 진행했던 커밋 히스토리는 모두 사라졌고 master 브랜치로 마치 fast-forward 병합된 것처럼 합쳐졌다.(물론, master 브랜치를 fast-forward 병합해줘야 한다.)

한 가지 알아둬야 할 것은 커밋도 완전히 새로운 커밋이라는 것이다. 왜냐하면 rebase를 하면서 충돌을 해결하거나 상위 커밋이 바뀌므로 아예 다른 커밋을 생성하여 진행하기 때문이다. 그러므로 feature 브랜치의 이전 커밋 집합들은 이후에 모두 가비지 컬렉션된다.

8.4.3 만약, master 브랜치에서 rebase 한다면?

앞에서 프로젝트 디렉토리를 복사해두길 당부한 적이 있다.(Git을 잘 쓰기 위해서 공부하려면 실험 정신이 투철해야 한다.)

예제에서 rebase 직전에 feature 브랜치로 체크아웃했는데 이는 feature 브랜치의 커밋 히스토리를 master 브랜치 다음으로 붙이기 위해서이다.

이를 반대로 해본다면 어떨까? 복사해둔 디렉토리로 이동해서 실험해보자.

```
$ git checkout master
$ git rebase feature
First, rewinding head to replay your work on top of it...
Applying: 1
Using index info to reconstruct a base tree...
M    stuff
Falling back to patching base and 3-way merge...
Auto-merging stuff
CONFLICT (content): Merge conflict in stuff
Failed to merge in the changes.
Patch failed at 0001 1
The copy of the patch that failed is found in:
   /Users/devcken/Projects/my-project2/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

충돌이 발생한 부분을 살펴보면 이번에는 '3'이 빠져있는 것을 알 수 있다.

```
$ vim stuff
$ git add stuff
$ git rebase --continue
Applying: 1
Applying: 3
Using index info to reconstruct a base tree...
M    stuff
Falling back to patching base and 3-way merge...
Auto-merging stuff
CONFLICT (content): Merge conflict in stuff
Failed to merge in the changes.
Patch failed at 0002 3
The copy of the patch that failed is found in:
   /Users/devcken/Projects/my-project2/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

또 다시 충돌이 발생한다. 이번에는 '3'이 충돌지점에서 보일 것이다.

```
$ vim stuff
$ git add stuff
$ git rebase --continue
```

충돌을 해결하자.

```
$ git log
* 303c010 | (HEAD, master) 2015-03-30 12:09:11 +0900 (78 minutes ago)
|           3 - Choi Leejun
* 2a67c68 | 2015-03-30 12:08:53 +0900 (78 minutes ago)
|           1 - Choi Leejun
* 5778949 | (feature) 2015-03-30 12:08:28 +0900 (78 minutes ago)
|           4 - Choi Leejun
* 3751dfb | 2015-03-30 12:08:14 +0900 (79 minutes ago)
|           2 - Choi Leejun
* cce20a4 | 2015-03-30 12:07:25 +0900 (79 minutes ago)
             initial commit - Choi Leejun
```

커밋 히스토리를 확인해보면 앞의 경우와 반대가 되어 있는 것을 알 수 있다. 즉, master 브랜치의 내용이 feature 브랜치로 rebase되었다.

이 예제를 진행한 것은 rebase 명령에 대해서 좀 더 알아보기 위한 것인데 어쨌든 결과는 똑같이 만들 수 있다. 충돌을 해결하는 과정에서 동일하게 수정이 이루어진다면 말이다.

8.4.4 rebase 하면 안되는 경우

`git reset`의 경우 다른 사용자에게 커밋이 공유되어버렸다면 해서는 안된다. `git rebase`의 경우에도 마찬가지이다.

앞선 예에서 만약 feature 브랜치가 원격 저장소에 올라가 공유된 상태라고 가정해보자. 누군가 feature 브랜치를 이용해서 변경 사항을 진행 중에 있을 수도 있다.

그런데 이를 rebase해버릴 경우, feature 브랜치가 원래 가지고 있던 커밋이 rebase한 측의 저장소에서는 사라지지만 원격 저장소와 이를 clone한 사용자의 저장소에는 남아 있을 것이다. 특히 clone하여 변경 사항을 진행 중인 사용자의 저장소가 문제다.

둘 중 누가 먼저 원격 저장소에 변경 사항을 적용하든간에 한쪽은 문제가 생긴다. 그 문제를 해결하여 원격 저장소에 또 적용하게 되면 다른 한쪽은 다시 문제에 부딪힌다.

공개된 커밋을 rebase하지 말아야 하는 이유에 대해서 예를 들지 않는 이유는 굳이 예를 들지 않아도 충분히 이해할 수 있는 부분이기 때문이다. Git을 사용하면서 하지말아야 하는 행동 중 하나가 이미 공개되어버린 커밋을 수정하지 말아야 한다는 것이다. rebase와 reset은 커밋을 수정 혹은 삭제해버린다.(checkout도 경우에 따라 마찬가지)

8.4.5 대화형 rebase

사실, `rebase`를 이용하는 이유가 대화형(`interactive`) `rebase`를 이용하기 위해서라고 말해도 과언이 아니다. 그만큼 대화형 `rebase`는 막강하다.

대화형 rebase를 사용하는 방법은 다음과 같다.

```
$ git rebase --interactive [<branch>|<commit-ish>]
```

대화형 rebase를 사용하기 위해 --interactive 옵션을 추가해 실행하게 되면 Git은 대화형 인터페이스를 제공한다. CLI 기반의 도구답게 GUI 기반의 인터페이스가 아니라 vim 등의 편집기를 통해 텍스트 기반의 인터페이스를 제공한다.

예를 위해, 다음은 같은 커밋 히스토리를 가정해보자.

```
$ git lg
* 4e2ea94 | (HEAD, feature) 2015-05-14 22:59:00 +0900 (2 seconds ago)
|           9 - Choi Leejun
* f911b4c | 2015-05-14 22:58:54 +0900 (8 seconds ago)
|           7 - Choi Leejun
* c5b50a5 | 2015-05-14 22:58:47 +0900 (15 seconds ago)
|           8 - Choi Leejun
* 04406ea | 2015-05-14 22:58:39 +0900 (23 seconds ago)
|           6? - Choi Leejun
* 7188c83 | 2015-05-14 22:58:33 +0900 (29 seconds ago)
|           5 - Choi Leejun
* 49e15f9 | 2015-05-14 22:56:37 +0900 (2 minutes ago)
|           3 - Choi Leejun
* ca440d4 | 2015-05-14 22:56:31 +0900 (3 minutes ago)
|           2 - Choi Leejun
```

```
| * db4358f | (master) 2015-05-14 22:56:19 +0900 (3 minutes ago)
|/            1 - Choi Leejun
* aa1698d | 2015-05-14 22:55:36 +0900 (3 minutes ago)
             Initial commit - Choi Leejun
```

초기 커밋을 제외하고 모두 커밋 메시지와 동일한 변경사항이 파일에 적용되었다. 로그 메시지와 순서를 유의해서 보자.

커밋 04406ea의 경우, 커밋 메시지에 물음표(?)가 있는데 이는 커밋 메시지 입력 시 잘못 입력된 문자다. 그러므로 제거해주고 싶은데 이미 다음 커밋이 진행된 상태라서 git commit --amend가 불가능하다. 그리고, 7과 8의 순서가 맞지 않고, 4에 대한 내용도 빠진 상태다.

rebase를 통해 이 모든 것을 한번에 수정할 수 있는 방법을 알아보자.

```
$ git rebase --interactive master
```

feature 브랜치의 변경사항들을 master 브랜치로 rebase해야 하므로 feature 브랜치로 체크아웃된 상태에서 실행해야 한다.

```
pick ca440d4 2
pick 49e15f9 3
pick 7188c83 5
pick 04406ea 6?
pick c5b50a5 8
pick f911b4c 7
pick 4e2ea94 9
# Rebase 45bebd1..3cf25fa onto 45bebd1
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

rebase를 실행하고 나면, 위와 같이 편집기를 통해 인터페이스가 출력되는데 커밋 메시지를 작성할 때와 별반 다르지 않다.

먼저, 주석 부분을 보면 인터페이스에서 사용할 수 있는 몇 가지 명령어들을 설명하고 있다.

- **pick**
  기본적으로 선택되는 옵션으로 변화를 주지 않고, 커밋을 그대로 사용한다.
- **reword**
  커밋 내용은 바뀌지 않고 커밋 메시지를 변경할 기회를 준다.
- **edit**
  pick과 비슷하지만 일시 정지한 후 커밋을 추가 혹은 삭제할 수 있다.
- **squash**
  두 개 이상의 커밋을 단일 커밋으로 합칠 수 있다. 적용된 커밋은 이전 커밋으로 합쳐지게 된다.
- **fixup**
  squash와 비슷하지만 커밋 메시지는 생략(삭제)된다. 앞선 커밋의 메시지가 사용된다.
- **exec**
  커밋에 대해 원하는 쉘 명령어를 실행할 수 있다.

이 명령어들은 각 커밋들 별로 적용할 수 있다. 적용하는 방법은 적용하고자 하는 커밋 번호 앞에 명령어를 바꿔주면 된다.(기본적으로 적용되어 있는 명령어는 pick이다.)

만약 커밋을 삭제하고 싶다면 커밋 번호 라인을 제거하면 된다. 또, 커밋 간의 순서를 바꾸고자 한다면 커밋 번호 라인의 순서를 변경해주면 된다.

앞서 가정한 바와 같이 변경사항을 rebase 인터페이스에 적용해보자.

```
1  pick ca440d4 2
2  edit 49e15f9 3
3  squash 7188c83 5
4  reword 04406ea 6?
5  pick f911b4c 7
6  pick c5b50a5 8
7  fixup 4e2ea94 9
8  # Rebase 45bebd1..3cf25fa onto 45bebd1
9  #
10 # Commands:
11 #  p, pick = use commit
12 #  r, reword = use commit, but edit the commit message
13 #  e, edit = use commit, but stop for amending
14 #  s, squash = use commit, but meld into previous commit
15 #  f, fixup = like "squash", but discard this commit's log message
16 #  x, exec = run command (the rest of the line) using shell
17 #
18 # These lines can be re-ordered; they are executed from top to bottom.
19 #
20 # If you remove a line here THAT COMMIT WILL BE LOST.
21 #
22 # However, if you remove everything, the rebase will be aborted.
23 #
```

첫번째 커밋(2)은 '2'를 추가한 변경사항인데, 초기 커밋으로부터 추가되었음을 잊지 말자. 이 커밋은 있는 그대로 master에 적용하고자 pick을 사용한다.

두번째 커밋(3)은 파일에 '4'를 추가한 커밋을 위해 edit를 적용했다.

세번째 커밋(5)은 이전 커밋에 병합시키기 위해 squash 명령을 사용한다.

네번째 커밋(6)은 커밋 메시지만 수정하면 되므로 reword 명령을 사용한다.

다섯번째 커밋(8)과 여섯번째 커밋(7)은 순서를 옮겼다.

마지막 커밋(9)는 이전 커밋에 병합시키기 위해 fixup을 적용한다.

대화형 인터페이스를 저장하고나면 rebase가 실행된다.

```
error: could not apply b8ce597... 2
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
Could not apply b8ce597dd64a31ff080fcce94cc708b7be33c258... 2
```

첫번째 커밋(2)을 리베이스하는 과정에서 충돌이 발생했으므로 해결해줘야 한다.

```
# 충돌을 해결한다.
$ vim stuff
$ git add stuff
$ git rebase --continue
[detached HEAD f5a8dc9] 2
 1 file changed, 1 insertion(+)
```

충돌을 해결한 뒤에 리베이스를 계속하면 커밋 메시지를 수정할 수 있는 기회를 준다. 커밋 메시지를 저장하면 해당 단계가 완료된다.

```
Stopped at 49e15f9dee38ccd2a23071100935357c6c19dd5f... 3
You can amend the commit now, with
    git commit --amend
Once you are satisfied with your changes, run
    git rebase --continue
```

두번째 커밋(3)까지 진행하고 나면 edit 명령의 영향으로 리베이스가 일시정지된다.

```
# 4를 추가한다.
$ vim stuff
$ git add stuff
$ git commit -m "4"
```

파일에 4를 추가하고 커밋을 만들어서 빼먹었던 변경사항을 추가할 수 있다.

```
$ git rebase --continue
error: could not apply 7188c83... 5
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
Could not apply 7188c83ea038175c168fc747cbec190077d849f2... 5
$ vim stuff
$ git add stuff
$ git rebase --continue
```

다음 단계를 계속하면 세번째 커밋(5)에서 충돌이 발생한다. 충돌을 해결한 뒤 다음 단계를 계속한다.

```
# This is a combination of 2 commits.
# The first commit's message is:
4
# This is the 2nd commit message:
5
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# rebase in progress; onto db4358f
# You are currently rebasing branch 'feature' on 'db4358f'.
#
# Changes to be committed:
#       modified:   stuff
#
```

세번째 커밋(5)의 경우 squash를 적용했으므로 리베이스를 계속할 경우 커밋 메시지 입력 시 위와 같이 출력된다. 왜냐하면 두 커밋을 병합해야 하기 때문에 커밋 메시지도 함께 출력한 것이다. 커밋 메시지를 적절히 편집한 뒤에 저장한다.

```
6?
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# rebase in progress; onto db4358f
# You are currently editing a commit while rebasing branch 'feature' on 'db4358
#
# Changes to be committed:
#       modified:   stuff
#
```

네번째 커밋(6)의 경우, 커밋 메시지 수정을 위해 reword를 적용했으므로 커밋 내용은 수정하지 못한다. 커밋 메시지를 수정한 뒤 저장한다.

```
error: could not apply f911b4c... 7
```

```
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
Could not apply f911b4c28a7546dfe79bab9118f96acfdea99a22... 7
$ vim stuff
$ git add stuff
$ git rebase --continue
[detached HEAD 6df6e26] 7
 1 file changed, 1 insertion(+)
```

다음 차례는 다섯번째 커밋(8)과 여섯번째 커밋(7)인데 두 커밋의 위치를 서로 바꾸었으므로 여섯번째 커밋이 먼저 처리된다. 이 과정에서 순서가 뒤틀리면서 충돌이 발생하게 된다. 이를 해결해주자.

```
error: could not apply c5b50a5... 8
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
Could not apply c5b50a51d7170d983c53b4516e037720407c4e74... 8
$ vim stuff
$ git add stuff
$ git rebase --continue
[detached HEAD a60432f] 8
 1 file changed, 1 insertion(+)
```

다섯번째 커밋(8)을 처리하는데, 다섯번째 커밋이 먼저 처리되면서 해결된 충돌 덕분에 다시 한번 충돌이 발생할 수 있다. 이 또한 해결해줘야 한다.

```
error: could not apply 4e2ea94... 9
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
Could not apply 4e2ea944043929ef36640cc85b81df4ab9ce5cd2... 9
$ vim stuff
$ git add stuff
$ git rebase --continue
```

일곱번째 커밋(9)을 처리하면 또 충돌이 발생된다. 이는 다섯번째 커밋과 여섯번째 커밋의 순서를 바꾸었기 때문에 일어나는 현상이다.

```
# This is a combination of 2 commits.
# The first commit's message is:
8
# The 2nd commit message will be skipped:
#       9
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# rebase in progress; onto db4358f
# You are currently rebasing branch 'feature' on 'db4358f'.
#
# Changes to be committed:
#       modified:   stuff
#
```

충돌을 해결한 뒤에 리베이스를 계속 한다. 그러면 일곱번째 커밋에 적용한 fixup의 작용으로 위와 같은 커밋 메시지를 볼 수 있다. 다섯번째 커밋과 일곱번째 커밋을 병합하게 되는데(다섯번째 커밋과 여섯번째 커밋의 순서가 바뀌었다는 것을 기억하라.) 마지막 커밋 메시지는 생략하게 된다. 물론, 커밋 메시지는 원하는데로 수정할 수 있다.

```
[detached HEAD 605a659] 8
 1 file changed, 2 insertions(+)
Successfully rebased and updated refs/heads/feature.
$ git lg
* 605a659 | (HEAD, feature) 2015-05-14 22:58:47 +0900 (32 minutes ago)
|           8 - Choi Leejun
* 6df6e26 | 2015-05-14 22:58:54 +0900 (32 minutes ago)
|           7 - Choi Leejun
* 7993363 | 2015-05-14 22:58:39 +0900 (32 minutes ago)
|           6 - Choi Leejun
* e3ea28b | 2015-05-14 23:11:30 +0900 (20 minutes ago)
|           4 5 - Choi Leejun
* 4e198c2 | 2015-05-14 22:56:37 +0900 (34 minutes ago)
|           3 - Choi Leejun
* f5a8dc9 | 2015-05-14 22:56:31 +0900 (35 minutes ago)
|           2 - Choi Leejun
* db4358f | (master) 2015-05-14 22:56:19 +0900 (35 minutes ago)
|           1 - Choi Leejun
* aa1698d | 2015-05-14 22:55:36 +0900 (35 minutes ago)
             Initial commit - Choi Leejun
```

리베이스는 완료됐고, 결과적으로 feature 브랜치는 master 브랜치와 일렬로 놓이게 된다.

예제에서 봤듯이 대화형 리베이스는 커밋 히스토리를 자유롭게 수정할 수 있는 기능이다. 물론 수정 중에 발생하는 충돌은 감수해야 하며, 해결해야 한다.

9. 스테이시

이 장에서는 현재 상태를 안전하게 저장하는 방법에 대해 설명한다.

9.1 현재 상태를 안전하게 저장한다.

어떤 브랜치에서 작업을 하고 있는 도중에 갑작스러운 핫픽스 요청이 들어왔다. 이 작업을 그냥 날리기에는 너무 많은 내용들이 진행되었고 또 커밋하기에는 아직 부족한 면이 있다.

이런 상황들은 프로젝트를 진행하다보면 생각보다 많이 발생할 수 있다. 또 해결 방법 또한 생각보다 많다.

아직 인덱스에 추가되지 않은 내용들이라면 그냥 파일을 다른 디렉토리에 임시로 복사해놓은 뒤에 다시 돌아왔을 때 복사해도 되지만 촌티가 나는 방법이다.

`git commit --amend`를 이용할 수도 있지만 이것도 커밋을 발생시킨다는 점에서 뭔가 꺼림직한 방법이다.

`git stash`는 위와 같은 상황에서 작업 디렉토리와 인덱스의 현재 상태를 **안전하게** 저장해둘 수 있는 명령이다.(stash의 뜻이 '안전하게 보관하다.'이다.)

현재 진행 중이던 내용들을 언제든지 `git stash`로 저장해두고 다른 브랜치로 이동하여 작업한 뒤에 다시 돌아와 복구하여 작업을 계속할 수 있다.

또, 저장한 내용을 다른 브랜치로 옮기는 것도 충분히 가능하다.(아마 잘못된 브랜치에서 어떤 변경 사항을 진행 중에 잘못된 브랜치임을 뒤늦게 알게 된 경우가 다들 있을거라고 생각된다.)

9.2 안전하게 보관한 뒤 복구해보기

고민할 필요도 없고 변경 사항을 잃을 걱정도 할 필요없다. 임시 저장처럼 느껴지지만 영구적으로도 가능하다. 오히려 사람의 기억이 문제다.

```
$ git log
* 68ccf72 | (HEAD, master) 2015-03-30 16:37:58 +0900 (18 seconds ago)
            initial commit - Choi Leejun
```

위와 같이 파일 하나가 포함된 초기 커밋이 있다고 가정하자.

```
$ vim stuff2
$ vim stuff
$ git add .
$ git commit
[master 8ada4ab] second commit
 2 files changed, 2 insertions(+)
 create mode 100644 stuff2
$ git lg
* 8ada4ab | (HEAD, master) 2015-03-30 16:44:07 +0900 (2 seconds ago)
|           second commit - Choi Leejun
* 68ccf72 | 2015-03-30 16:37:58 +0900 (6 minutes ago)
             initial commit - Choi Leejun
```

stuff2라는 파일을 만들고 stuff 파일은 수정한 뒤 둘 다 커밋한다. 즉 두 개 파일이 Git에 의해 추적되고 있는 상태다.

```
$ vim stuff
$ vim stuff2
$ vim stuff3
$ git add stuff
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   stuff

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:   stuff2

Untracked files:
  (use "git add <file>..." to include in what will be committed)

```

```
21      stuff3
```

다시 stuffd와 stuff2 파일을 수정한 뒤 이번에는 stuff 파일만 인덱스에 추가한다. 그리고 stuff3라는 파일을 새로 만든다.

`git status` 명령을 실행해보면 위와 같이 나와야 한다.

```
1  $ git stash
2  Saved working directory and index state WIP on master: 8ada4ab second commit
3  HEAD is now at 8ada4ab second commit
```

`git stash` 명령을 실행하자, 작업 디렉토리와 인덱스 상태가 저장되었다고 나온다. 그리고 HEAD는 작업 커밋 상에 있음을 알려준다.

참고

WIP는 working on progress의 줄임말이다.

```
1  $ git status
2  On branch master
3  Untracked files:
4    (use "git add <file>..." to include in what will be committed)
5
6      stuff3
7
8  nothing added to commit but untracked files present (use "git add" to track)
```

`git status` 명령을 실행해보면 stuff와 stuff2 두 파일의 변경 사항이 없어진 것을 알 수 있다.

한 가지 주목해야 할 점은 stuff3 파일은 그대로라는 것이다. stuff3 파일은 새로 만들어진 파일이고 한번도 인덱스에 추가되거나 커밋되지 않은 파일이다. 그러므로 Git은 이 파일을 자신이 추적하고 있지 않으므로 stash 대상으로도 여기지 않는다.

이제 문제될 것이 없으므로 다른 브랜치로 체크아웃하는 것이 가능하다.

```
1  $ git checkout -b feature
2  Switched to a new branch 'feature'
```

여기서 stuff3 파일은 추적되지 않고 있는 파일에 대한 예제를 위해 사용된 것일뿐 실제로 새 파일일지라도 인덱스에 추가한 뒤에 stash하는 것이 좋다.

그런데, 잠시 저장해둔 변경 사항들은 어디에 있는 것일까?

```
$ git stash list
stash@{0}: WIP on master: 8ada4ab second commit
```

위와 같이 `git stash` 명령의 하위 명령으로 `list`을 실행하게 되면 현재 stash area에 저장되어 있는 변경사항들을 모두 조회 가능하다.

목록 앞에 보이는 `stash@{0}`는 stash ID로 각각의 저장 내용을 구별짓는 번호이다.

```
$ git add stuff3
$ git stash
Saved working directory and index state WIP on feature: 8ada4ab second commit
HEAD is now at 8ada4ab second commit
$ git stash list
stash@{0}: WIP on feature: 8ada4ab second commit
stash@{1}: WIP on master: 8ada4ab second commit
```

앞서 만들어두었던 stuff3 파일을 인덱스에 추가한 뒤 `git stash` 명령을 실행해 저장했다. 그리고 stash 목록을 확인해보자.

위와 같이 stash는 스택 방식으로 동작한다. 즉, stash ID는 변한다. 가장 최근에 저장한 것이 가장 먼저 나올 수 있으므로 주의해야 한다.

```
$ git stash pop
On branch feature
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   stuff3

Dropped refs/stash@{0} (c8a6a5211f01f57e378e45960feecc31cddddd75)
$ git stash list
stash@{0}: WIP on master: 8ada4ab second commit
```

`git stash pop` 명령을 통해 저장내용을 현재 브랜치에 적용할 수 있는데 이 때 Git은 친절하게도 `git status` 결과를 함께 보여준다.

그리고, 스택에 대해서 *pop*했으므로 당연히 목록에서도 제거된다. `git stash list` 명령을 통해 다시 확인해보면 목록에서 제거됐음을 알 수 있다.

```
# 예제 진행을 위해 일단 feature 브랜치에서 커밋을 하자.
$ git commit -m "stuff3"
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
```

이번에는 hotfix 브랜치로 체크아웃해보자.

그리고 hotfix 브랜치에 stash에 저장된 내용을 적용해볼 건데 앞에서 본 것과는 다른 방식을 사용할 것이다.

```
$ git stash apply
On branch hotfix
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:   stuff
    modified:   stuff2

no changes added to commit (use "git add" and/or "git commit -a")
$ git stash list
stash@{0}: WIP on master: 8ada4ab second commit
```

`git stash apply` 명령은 `git stash pop`와 비슷하면서도 다르다. 일단 현재 브랜치에서 저장된 내용을 적용하는 것은 동일하다. 하지만 적용된 저장된 내용을 stash 목록에서 drop하지는 않는다. 그러므로 이를 이용하면 여러 브랜치에 저장된 내용을 적용하는 것이 가능하다.

아무튼 이제 stuff와 stuff2의 변경사항은 hotfix 브랜치의 작업 디렉토리와 인덱스에 적용되었다.

```
# 예제 진행을 위해 커밋하자.
$ git commit -a -m "git stash apply"
[hotfix e564ff4] git stash apply
 2 files changed, 2 insertions(+)
```

앞서 `git stash apply`를 실행했기 때문에 stash area에 `stash@{0}`가 남아 있는 상황이다. hotfix 브랜치에 이미 적용했기 때문에 필요없다고 판단이 들어 삭제하고 싶다.

```
$ git stash drop stash@{0}
Dropped stash@{0} (84374b142a4f5d9b76566506a5d12535ff3f555f)
```

`git stash drop` 명령은 특정 stash를 삭제(drop)해준다. 이 예제에서는 stash ID를 명시했으나 명시하지 않은 *pop*할 때와 마찬가지고 가장 나중에 저장된 내용이 먼저 삭제된다.

참고

stash는 Git의 고급 기술 중 가장 쉬우면서도 굉장히 유용한 기술이다. 잘 알아두도록 하자.

10. 참조 로그

이 장에서는 작업 히스토리를 사용하는 방법을 설명한다.

Git에서 커밋을 하면 커밋 히스토리가 남는다. reset을 하면 커밋 히스토리에서 커밋이 사라지기도 한다. 그런데 사용자가 작업한 히스토리는 볼 수 없는걸까?

다행히도 Git은 작업 히스토리를 기록해두는데 이것을 ***reflog***라고 한다. 이 reflog는 참조에 업데이트가 발생할 때마다 함께 업데이트된다. reflog가 업데이트되는 내역은 다음과 같다.

- clone
- push
- commit
- branch
- rebase
- reset, revert, checkout

reflog는 그 용도를 특정하기 어려울 정도로 유용한데 여기서는 아주 간단한 예제를 통해 사용 방법을 알아보도록 하겠다.

```
$ git log
* 739d124 | (HEAD, master) 2015-03-30 18:32:26 +0900 (5 seconds ago)
|           2 - Choi Leejun
* ed175a4 | 2015-03-30 18:32:03 +0900 (28 seconds ago)
             1 - Choi Leejun
```

현재 위와 같은 커밋 히스토리에 있다고 가정하자.

```
$ git reset --hard HEAD~1
HEAD is now at ed175a4 1
$ git log
* ed175a4 | (HEAD, master) 2015-03-30 18:32:03 +0900 (3 minutes ago)
             1 - Choi Leejun
```

`git reset --hard` 명령으로 HEAD 바로 앞에 있는 커밋으로 reset했다. 그런데 알고보니 reset을 한 것이 실수였다. 그리고 reset 이전에 HEAD 참조 커밋의 해시를 모른다고 가정하자.

당연한 이야기지만, `git reset` 명령 이후에 커밋 히스토리를 봐도 삭제된 커밋은 나오지 않는다. 커밋 해시를 알 수 있는 방법이 없을까? 이럴 때 사용하는 것이 바로 `git reflog`다.

```
$ git reflog
ed175a4 HEAD@{0}: reset: moving to HEAD~1
739d124 HEAD@{1}: commit: 2
ed175a4 HEAD@{2}: commit (initial): 1
```

`git reflog` 명령은 위와 같이 변경사항들을 기록해놓고 있다. 관련 커밋의 해시는 물론, 어떤 명령이 실행됐고 그 결과 어떻게 됐는지를 기록해놓는다.

```
$ git reset 739d124
$ git log
```

```
* 739d124 | (HEAD, master) 2015-03-30 18:32:26 +0900 (2 minutes ago)
|          2 - Choi Leejun
* ed175a4 | 2015-03-30 18:32:03 +0900 (3 minutes ago)
            1 - Choi Leejun
```

커밋 해시를 알아냈으므로 다시 reset하면 된다.

참고

쓸모없어진 커밋은 어느 순간 가비지 컬렉션에 의해 삭제된다. 하지만 가비지 컬렉션 전에는 참조가 가능하다.

A. reflog 아이디를 이용한 참조

reflog 아이디는 커밋을 참조한다. 그래서 reflog 아이디를 이용하는 것이 가능하다.

```
$ git show HEAD@{0}
commit ed175a470f906f1b3d2c252eec2840ac27c63c70
Author: Choi Leejun <devcken@gmail.com>
Date:   Mon Mar 30 18:32:03 2015 +0900

    1

diff --git a/stuff b/stuff
new file mode 100644
index 0000000..d00491f
--- /dev/null
+++ b/stuff
@@ -0,0 +1 @@
+1
```

`git show` 명령에 커밋 해시 대신 reflog 아이디를 전달하면 해당 아이디가 참조하는 커밋 해시의 내용을 보여준다.

11. 원격 저장소

이 장에서는 로컬 저장소와 원격 저장소를 함께 사용하는 방법에 대해 설명한다.

11.1 저장소란?

11.1.1 bare vs non-bare

Git 저장소는 크게 봤을 때 **bare 저장소**와 **non-bare 저장소**로 나뉘어 진다.

non-bare 저장소는 보통의 경우 로컬 저장소를 말하며 개발이 이루어지는 브랜치들의 개발 사본을 제공하여 개발이 직접적으로 이루어지는 작업 디렉토리를 제공한다. 반면, bare 저장소는 작업 디렉토리가 없는 저장소로 커밋을 할 수 없으며, 개발 변경사항들의 공유만을 위해 사용된다.

bare 저장소는 non-bare 저장소와 대조적으로, 일반적인 경우 원격 저장소를 말하는데 반드시 그런 것은 아니다.

bare 저장소를 자신의 로컬에 초기화한 뒤에 이를 복제해보라.

이 포스트 내내 원격 저장소라는 용어를 최대한 자제할 것이다. 대신 non-bare 저장소라는 용어를 사용할 것이다.

non-bare 저장소, 즉 로컬(개발) 저장소를 만드는 방법은 다음과 같다.

```
1  # 저장소로 만들기 원하는 디렉토리로 이동한 뒤
2  $ git init
```

bare 저장소를 만들기 위한 방법은 다음과 같다.

```
1  # 저장소로 만들기 원하는 디렉토리로 이동한 뒤
2  $ git init –bare
```

11.1.2 저장소 복제

bare 저장소는 공유와 협력을 위해서 사용되는데, 이를 복제하여 non-bare 저장소를 만들어야 해당 저장소에 대한 작업 디렉토리를 사용할 수 있다. 일반적으로 저장소를 복제하는 방법은 다음과 같다.

```
$ git clone <path>
```

복제하게 되면 bare 저장소의 브랜치들은 복제된 저장소(non-bare)에 **원격 추적 브랜치**로 저장된다. 이 원격 추정 브랜치는 .git 디렉토리 밑에 refs/remotes 디렉토리에 저장된다.(bare 저장소에 있는 원격 추적 브랜치는 복제되지 않는다.)

저장소를 복제하게 되면 원본 저장소에 대한 정보가 origin이라는 이름의 리모트로 저장되게 된다. 원격 추적 브랜치의 이름은 이 리모트 이름을 이용해 표기된다. origin 리모트에 있는 master 브랜치는 origin/master라는 이름으로 복제된 저장소에 저장된다.

11.1.3 리모트 구성

저장소를 복제하는 방법 외에 자신의 non-bare 저장소를 bare 저장소와 연결하는 방법이 있다.

앞서 '저장소 복제'에서 복제 시 origin 이라는 이름의 리모트로 bare 저장소의 정보가 저장된다고 했는데, 사실 이 origin이라는 이름은 특별한 이름이 아니다. Git이 bare 저장소를 복제할 때 기본적으로 사용하는 이름일 뿐이다. 우리가 원하는 리모트 이름으로 non-bare 저장소와 bare 저장소를 연결할 수 있다.

A. 리모트 추가

리모트를 추가하는 방법은 생각보다 간단하다.

```
$ git remote add <remote-name> <path>
```

리모트 이름에는 원하는 이름을 설정 가능한데, 당연한 이야기지만 중복은 불가능하다. 즉, 리모트 이름은 하나의 저장소 내에서 유일해야 한다.

B. 리모트 조회

어떤 리모트가 현재 저장소에 설정되어 있는지 조회하고 싶을 수 있다.

```
$ git remote
origin
```

git remote 명령으로 리모트 조회가 가능하며 조회된 리모트 목록이 나열된다.

그런데 해당 리모트에 대한 상세 내용은 어떻게 볼 수 있을까?

```
$ git remote show <remote-name>
$ git remote show origin
* remote origin
  Fetch URL: http://gitlab2.uit.nhncorp.com/nid/member.git
  Push  URL: http://gitlab2.uit.nhncorp.com/nid/member.git
  HEAD branch: master
  Remote branches:
    master              tracked
    sus                 tracked
    sus-membersus-16604 tracked
    sus-membersus-16758 tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local refs configured for 'git push':
    master              pushes to master              (up to date)
    sus                 pushes to sus                 (local out of date)
    sus-membersus-16604 pushes to sus-membersus-16604 (up to date)
    sus-membersus-16758 pushes to sus-membersus-16758 (up to date)
```

git remote show 명령으로 리모트에 대한 상세 정보를 열람할 수 있는데, Fetch URL과 Push URL을 알 수 있으며 브랜치 정보들도 한눈에 알 수 있다.

C. 리모트 변경 및 제거

리모트 이름을 변경하는 방법은 다음과 같다.

```
$ git remote rename <source-name> <destination-name>
```

리모트 이름이 변경되면 원격 추적 브랜치의 이름 또한 모두 변경된 리모트 이름으로 변경된다.

한 가지 아쉬운 점은 리모트에 대한 경로는 변경하지 못한다는 것인데, 간단하게 리모트 삭제 후 다시 추가하면 된다. 리모트를 삭제하는 방법은 다음과 같다.

```
$ git remote rm <remote-name>
```

11.2 원격 저장소와의 동기화

11.2.1 프로토콜

Git은 원격 저장소를 참조하기 위한 여러 가지 프로토콜을 제공한다. 이 프로토콜들은 원격 저장소를 참조하기 위한 용도로 사용되지만 서버 관리자 입장에서는 어떤 프로토콜을 개방할 것인지를 결정하는 대상이 된다.

A. 로컬 파일 시스템

로컬 파일 시스템은 실제 파일 시스템 혹은 네트워크 파일 시스템을 나타내는데, 다음과 같은 형식으로 나타낼 수 있다. 만약 로컬 파일 시스템의 경로를 네트워크 파일 시스템 경로로 이용해야 할 경우 네트워크에서 서로 다른 장비 간의 네트워크가 가능하도록 설정이 필요할 수도 있다.

```
/path/to/repo.git
file://path/to/repo.git
```

B. Git Native 프로토콜

Git Native 프로토콜은 Git이 직접 지원하는 프로토콜로 내부적인 통신에도 이용된다. Git 프로토콜은 Git에서 사용할 수 있는 프로토콜 중 가장 효율적이며 가장 빠르다고 알려져 있다.(주장하고 있다.) 다만, Git Native 프로토콜의 경우 9418 포트를 방화벽에서 열어야 하므로 사용되지 않는 곳도 있다.

```
git://example.com/path/to/repo.git
```

C. SSH

보안이 강화된 네트워크를 사용하고 싶다면 SSH를 사용할 수 있다. 다만, 네트워크 연결을 위한 사전 작업이 필요하며 접속해야 하는 모든 장비를 인증해줘야 하는 번거로움이 따른다.

```
ssh://[user@]example.com[:port]/path/to/repo.git
```

D. HTTP 및 HTTPS

많은 사람들에게 익숙한 웹 URL을 사용하는 방법이다. 대부분 80포트는 개방되어 있으며 HTTPS 포트(443)도 개방되어 있는 경우가 많아 사용 빈도가 가장 높다.

```
http://example.com/path/to/repo.git
https://example.com/path/to/repo.git
```

11.2.2 원격 저장소 추적

원격 저장소로부터 복제하거나 혹은 로컬 저장소로 원격 저장소에 연결하게 되면 원격 저장소의 변경 사항을 추적할 수 있게 된다. 이렇게 변경 사항을 추적 가능하도록 하는 것이 바로 **원격 추적 브랜치**다. 그래서 브랜치를 로컬과 원격의 개념에서 바라보면 다음과 같이 구분될 수 있다.

- **원격 추적 브랜치(remote tracking branch)**는 원격 저장소에 존재하는 브랜치의 변경 사항을 추적한다. 즉, 원격 저장소의 변경 사항은 로컬 상에서 원격 추적 브랜치를 통해 공유받게 된다.
- **로컬 추적 브랜치(local tracking branch)**는 원격 추적 브랜치와 쌍을 이루어 원격 저장소의 변경 사항을 추적하는데 사용된다. 쉽게 말해, 원격 저장소에 공유된 로컬 브랜치라고 생각하면 된다.

- 로컬 토픽 브랜치(local topic branch)는 원격 저장소에 공유되지 않은 비추적(non-tracking) 브랜치를 말한다.

A. 원격 추적 브랜치 조회

원격 추적 브랜치는 우리가 일반적으로 사용하는 git branch 명령으로는 조회가 되지 않는다.

```
$ git branch -r
refs/remotes/origin/master
```

-r 옵션과 함께 git branch 명령을 실행하게 되면 (보통의 경우)CLI 상에서 붉은 색으로 표시되며 이는 로컬 브랜치와 쌍을 이루는 원격 추적 브랜치를 나타낸다.

만약 로컬 브랜치와 원격 추적 브랜치를 함께 보고싶다면 --all(혹은 -a) 옵션을 사용한다.

```
$ git branch --all
* master
refs/remotes/origin/master
```

11.2.3 변경 사항 공유하기

로컬 저장소에 커밋한 변경 사항은 원격 저장소에는 존재하지 않는다. 원격 저장소에 커밋을 전송해야 하는 이유는 **다른 사람과의 변경 사항 공유**다. 물론, 백업의 이유도 있을 수 있으나 원격 저장소를 사용하는 가장 중요한 이유는 공유다. 공유를 위해 사용되는 가장 쉬운 방법은 다음과 같다.

```
$ git push <remote-name> <branch-name>
$ git push origin master
Counting objects: 4, done
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 400 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /tmp/Depot/public_html
    0d4ce8a..6f16880 master -> master
```

로컬 저장소의 변경 사항이 공유(푸쉬)되면 원격 저장소의 원격 브랜치에도 변경 사항이 반영된다. 또한 로컬 저장소의 원격 추적 브랜치 또한 로컬 브랜치의 변경 사항을 병합하게 되는데 fast-forwarding 방식으로 병합된다.

만약 1개 이상의 브랜치에서 발생한 모든 변경 사항을 한 번에 공유하고 싶다면 다음과 같이 --all 옵션을 사용하면 된다.

```
$ git push <remote-name> --all
```

원격 저장소에 로컬 저장소의 변경 사항이 공유될 때, 해당 변경 사항이 속하는 브랜치가 원격 저장소에 존재하지 않을 경우, 해당 브랜치에 대한 원격 브랜치가 원격 저장소에 만들어지고 로컬에서는 해당 원격 브랜치에 대한 원격 추적 브랜치가 만들어진다. 그래서 로컬 브랜치는 로컬 추적 브랜치가 되고 원격 추적 브랜치와 쌍을 이루어 원격 저장소의 원격 브랜치를 추적하게 된다.

A. 원격 브랜치를 삭제하기

앞서 원격 브랜치는 해당 브랜치가 원격 저장소에 없을 경우 만들어진다고 했다. 예를 들어 다음과 같은 명령어는 해당 브랜치가 원격 저장소에 없을 경우, 원격 브랜치를 만드는 동작으로 함께 수행하게 된다.

```
$ git push origin feature
```

feature 라는 브랜치가 origin이라는 리모트의 원격 저장소에 없을 경우, 변경 사항 공유 이전에 원격 브랜치를 원격 저장소에 만든다. 그런 뒤에 해당 브랜치의 변경 사항을 원격 저장소의 해당 브랜치에 적용하게 된다.

사실, 해당 명령어는 다음 표현과 동일한 표현이다.

```
$ git push origin feature:feature
# 혹은
$ git push origin feature:refs/heads/feature
```

원격 브랜치의 삭제는 이와 유사한 표현을 사용하게 된다.

```
$ git push <remote-name> :<branch-name>
혹은
$ git push <remote-name> --delete <branch-name>
```

:<branch-name> 표현이 어색하다면 --delete 옵션을 사용하면 된다.

11.2.4 변경 사항 공유받기

git push를 이용해 변경 사항을 원격 저장소로 공유하기도 하지만 다른 개발자의 변경 사항을 로컬 저장소로 공유 받기도 해야 한다.

변경 사항을 원격 저장소로부터 공유받기 위해서 보통 다음과 같은 명령을 사용한다.

```
$ git pull <remote-name> <branch-name>
$ git pull origin master
```

git pull 명령은 fetch, 병합(혹은 리베이스) 두 가지 단계에 걸쳐 동작한다.

A. fetch 단계

Git은 fetch 단계에서 원격 저장소의 변경 사항을 로컬 저장소로 가져오는데 이 때 원격 추적 브랜치를 사용하게 된다. 원격 브랜치에 대한 원격 추적 브랜치가 존재하지 않을 경우 원격 추적 브랜치를 만들고 이 원격 추적 브랜치에 변경 사항을 공유 받게 된다.

fetch는 원래 다음과 같이 fetch 명령을 통해 실행되는데, git pull 명령은 이를 포함한다.

```
$ git fetch <remote-name>
$ git fetch origin
```

B. 병합 혹은 리베이스 단계

두번째 단계에서는 공유 받은 원격 추적 브랜치를 로컬 추적 브랜치에 병합하거나 리베이스한다. 병합할 경우, 로컬 추적 브랜치로 원격 추적 브랜치의 내용이 병합된다.

C. 병합할 것인가, 리베이스할 것인가?

fetch 이후에 병합 시 충돌이 발생할 수도 있는데 이러한 경우 충돌을 해결해야 하므로 3-ways 병합이 발생할 수도 있다. 그러므로 히스토리가 지저분해지는 단점이 있다. 이러한 단점을 보완하려면 리베이스해야 하는데 물론 이때도 충돌을 해결할 수도 있긴 하지만 결과적으로 히스토리를 선형적으로 만들 수 있는 장점이 있다.

충돌이 없는 경우에는 fast-forward 병합이 발생하므로 히스토리는 (리베이스하지 않더라도) 선형적으로 만들어진다.

변경 사항을 공유받을 때, 사람들이 고민하는 것이 병합과 리베이스의 선택인데, 어떤 것을 선호하는지에 따라 결정하면 된다. 히스토리를 군더더기없이 최대한 선형적으로 관리하고 싶다면 (충돌이 예상될 경우) 리베이스해야 한다. 만약, 충돌이 발생하여 히스토리가 분기되어도 괜찮다면 굳이 리베이스하지 않아도 된다.

결국 본인의 취향적인 선택 사항이며 어느 것도 더 좋다라고 말하기에는 모호한 부분이 있다. 리베이스가 히스토리를 선형적으로 만든다는 장점 이면에는 병합 히스토리가 없어진다는 단점도 함께 있다. 취향에 따라 병합 히스토리를 남기고 싶어 하는 사람도 있다.