Git 사용자 설명서

목차

Git 사용자 설명서 (버전 1.5.3 이상)

원문 : http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

번역 : 2009-04-13 start!

서문

git는 빠른 분산형 버전 관리 시스템이다.

이 문서는 기본적인 유닉스 명령행 도구에 익숙한 사람들을 위해 작성되었으며, git에 대한 사전 지식이 필요하지는 않다.

1장 "저장소와 브랜치" 및 2장 "Git 변경 내역 조사하기"에서는 git를 이용하여 프로젝트를 가져오고 검토하는 방법을 설명한다. 특정 버전의 소프트웨어를 빌드하고 테스트하거나, 퇴보한 부분을 검색하는 등의 일을 하고 싶다면 이 부분을 읽어보기 바란다.

실제로 개발에 참여하고 싶은 사람들은 3장 "Git를 이용하여 개발하기" 및 4장 "개발 과정 공유하기" 부분도 읽어야 할 것이다.

이후의 장들은 좀 더 특수한 주제들을 다룬다.

자세한 참조 문서는 man 페이지 혹은 **git-help(1)** 명령을 통해 볼 수 있다. 예를 들어 "git clone <저장소>" 명령에 대한 문서를 보고 싶다면 다음의 두 명령 중 하나를 이용할 수 있다.

```
CODE $ man git-clone
```

혹은

```
CODE $ git help clone
```

두 번째 경우라면 여러분이 원하는 설명서 보기 프로그램을 선택할 수 있다. 자세한 내용은 **git-help(1)**을 보기 바란다.

자세한 설명 없이 git 명령 사용법을 간략히 보고 싶다면 부록 A "Git 빠른 참조" 부분을 살펴보도록 하자.

마지막으로 부록 B "이 설명서에 대한 노트 및 할 일 목록" 부분을 살펴보고 이 설명서가 보다 완벽해지도록 도와줄 수 있다.

저장소와 브랜치#

git 저장소 가져오기#

이 설명서를 읽으면서 각 명령들을 실험해 볼 git 저장소가 있다면 유용할 것이다.

git 저장소를 가지기 위한 가장 좋은 방법은 **git-clone(1)** 명령을 이용하여 기존의 저장소의 복사본을 다운로드하는 것이다. 여러분이 딱히 정해둔 프로젝트가 없다면 아래의 흥미로운 예제들을 이용해 보기 바란다.

```
CODE          # git 프로그램 자체 (대략 10MB 정도 다운로드):
    $ git clone git://git.kernel.org/pub/scm/git/git.git
          # 리눅스 커널 (대략 150MB 정도 다운로드):
    $ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

규모가 큰 프로젝트의 경우 최초 복사(clone) 작업에는 꽤 많은 시간이 소요될 수 있다. 하지만 이 과정은 오직 한 번만 필요하다.

clone 명령은 프로젝트의 이름에 해당하는 디렉토리를 새로 만든다. (위의 예제에서는 "git"와 "linux-2.6"에 해당한다.) 생성된 디렉토리 안으로 이동해 보면 프로젝트 파일의 복사본들과 (이를 작업 트리라고 부른다) 프로젝트의 변경 내역에 대한 모든 정보를 포함하고 있는 ".git" 라는 특수한 최상위 디렉토리가 존재하는 것을 볼 수 있을 것이다.

다른 버전의 프로젝트 체크아웃 하기#

git는 여러 파일들에 대한 변경 내역을 저장하기 위한 도구로 최선의 선택이다. git는 변경 내역을 프로젝트 내용물의 상호 연관된 스냅샷들을 압축하여 모아두는 방식으로 관리한다. git에서는 이러한 각각의 버전들을 커밋이라고 부른다.

이러한 스냅샷들은 꼭 시간 순으로만 일렬로 배열될 필요는 없다. 작업 내용은 개발 과정에 따라 동시에 병렬로 이루어 질 수 있으며 (이를 브랜치라고 부른다) 이는 하나로 합쳐지거나 여러 개로 나누어질 수 있다.

하나의 git 저장소는 여러 브랜치의 개발 과정을 관리할 수 있다. 이는 각 브랜치의 마지막 커밋에 대한 참조를 나타내는 헤드의 목록을 관리하는 방식으로 수행된다. **git-branch(1)** 명령은 브랜치 헤드의 목록을 보여준다.

```
CODE $ git branch
     * master
```

새로 복사된 저장소는 기본적으로 "master"라는 이름을 가지는 하나의 브랜치와 이에 대한 헤드가 참조하는 프로젝트의 상태로 초기화된 작업 디렉토리를 포함한다.

대부분의 프로젝트에서는 태그도 함께 사용한다. 태그는 헤드와 마찬가지로 프로젝트의 변경 내역에 대한 참조이며, **git-tag(1)** 명령으로 태그의 목록을 볼 수 있다.

```
CODE $ git tag -l
     v2.6.11
     v2.6.11-tree
     v2.6.12
     v2.6.12-rc2
     v2.6.12-rc3
     v2.6.12-rc4
     v2.6.12-rc5
     v2.6.12-rc6
     v2.6.13
     ...
```

태그는 항상 프로젝트의 동일한 버전을 가리키도록 되어 있지만, 헤드는 개발 과정에 따라 계속 변경되도록 되어 있다.

git-checkout(1) 명령을 이용하여 이러한 버전 중의 하나에 대한 브랜치 헤드를 만들고 코드를 체크아웃 할 수 있다.

```
CODE $ git checkout -b new v2.6.13
```

이제 작업 디렉토리는 v2.6.13으로 태그를 붙인 시점의 프로젝트의 내용을 반영하게 되며, **git-branch(1)** 명령은 두 개의 브랜치를 보여 준다. 이 중 별표(*) 표시된 것은 현재 체크아웃된 브랜치임을 나타낸다.

```
CODE $ git branch
       master
     * new
```

만약 (2.6.13 버전 대신) 2.6.17 버전의 코드를 보기로 결정했다면, 다음과 같은 방법으로 현재 브랜치가 v2.6.17을 가리키도록 수정할 수 있다.

```
CODE $ git reset --hard v2.6.17
```

기억해 두어야 할 것은 위의 경우에서 만약 현재 브랜치 헤드가 변경 내역 중의 특정 지점을 가리키는 유일한 참조였다면, 브랜치를 초기화(reset)하는 과정에서 이전의 위치로 돌아갈 수 있는 방법이 사라지게 된다는 점이다. 따라서 이 명령은 주의해서 사용해야 한다.

변경 내역 이해하기: 커밋#

프로젝트의 변경 내역에 포함된 모든 변경 사항은 커밋으로 표현된다. **git-show(1)** 명령은 현재 브랜치 상의 가장 최신 커밋 정보를

보여준다:

```
CODE $ git show
commit 17cf781661e6d38f737f15f53ab552f1e95960d7
Author: Linus Torvalds <torvalds@ppc970.osdl.org.(none)>
Date:   Tue Apr 19 14:11:06 2005 -0700

    Remove duplicate getenv(DB_ENVIRONMENT) call

    Noted by Tony Luck.

diff --git a/init-db.c b/init-db.c
index 65898fa..b002dc6 100644
--- a/init-db.c
+++ b/init-db.c
@@ -7,7 +7,7 @@

 int main(int argc, char **argv)
 {
-       char *sha1_dir = getenv(DB_ENVIRONMENT), *path;
+       char *sha1_dir, *path;
        int len, i;

        if (mkdir(".git", 0755) < 0) {
```

위에서 볼 수 있듯이, 커밋은 최신 변경 사항을 누가, 무엇을, 왜 수정했는지에 대한 정보를 보여준다.

모든 커밋은 40자의 16진수 ID를 가지며 (이는 때때로 "객체 이름" 혹은 "SHA-1 id"라고 부른다) 이는 "git-show" 명령의 출력에서 첫 번째 줄에 나타난다. 보통 각각의 커밋은 태그 혹은 브랜치와 같은 더 짧은 이름으로 참조하지만, 이러한 긴 이름이 유용할 때도 있다. 가장 중요한 점은 이것은 해당 커밋에 대한 유일한 (globally unique) 이름이라는 것이다. 따라서 여러분이 다른 사람에게 객체 이름을 이야기 하게되면 (예를 들어 이메일 등을 통해) 여러분의 저장소 내의 커밋과 동일한 (다른 사람의 저장소 내에 있는) 커밋을 가리킨다고 보장할 수 있는 것이다. (그 저장소에는 이미 여러분의 모든 커밋 정보가 다 들어있다고 가정한다.) 객체 이름은 커밋 내용들에 대한 해시값으로 계산되기 때문에 객체 이름이 함께 바뀌지 않은 한 커밋 내용도 바뀌지 않는다고 보장받을 수 잇다.

사실 7장 "Git 개념 정리" 부분에서 우리는 파일 데이터 및 디렉토리 내용을 포함하여 git 변경 내역 안에 저장되는 모든 것은 객체 안에 저장되고, 그 객체의 이름은 객체의 내용에 대한 해시값으로 만들어 진다는 것을 알게 될 것이다.

변경 내역 이해하기: 커밋, 부모, 도달 가능성#

(프로젝트의 최초 커밋을 제외한) 모든 커밋은 해당 커밋 바로 전에 변경된 내용을 포함하는 부모 커밋을 가진다. 이러한 부모 커밋의 연결을 따라가면 결국에는 프로젝트의 시작점에 다다르게 될 것이다.

하지만 커밋들은 단순한 경로로 연결되지 않는다. git는 개발의 진행 단계가 (여러 경로로) 나눠지고 다시 합쳐지는 것을 허용하며, 이러한 두 개발 경로가 다시 합쳐지는 지점을 "머지"라고 부른다. 따라서 머지를 가리키는 커밋은 둘 이상의 부모 커밋을 가지며, 각 부모 커

밋은 각 개발 경로에서 현재 지점까지 이르는 가장 최근의 커밋을 나타낸다.

이 작업이 어떻게 이루어지는지 보기 위한 가장 좋은 방법은 **gitk(1)** 명령을 이용하는 것이다. 지금 git 저장소에서 gitk 명령을 실행하여 머지 커밋을 찾아본다면 git가 변경 내역을 저장하는 방식을 이해하는 데 도움이 될 것이다.

이후부터는, 만약 커밋 X가 커밋 Y의 조상인 경우에, 커밋 X는 커밋 Y로부터 "도달 가능"하다고 할 것이다. 마찬가지로 커밋 Y는 커밋 X의 후손이다라거나 커밋 Y로부터 커밋 X에 이르는 부모의 연결이 존재한다고 말할 수 있다.

변경 내역 이해하기: 변경 내역 다이어그램#

우리는 때때로 아래와 같은 다이어그램을 이용하여 git 변경 내역을 표시할 것이다. 커밋은 소문자 "o"로 표시하며, 커밋 간의 경로는 -,/,\ 기호를 이용한 선으로 표시한다. 시간은 왼쪽에서 오른쪽으로 흐른다.

```
         o--o--o <-- 브랜치 A
        /
o--o--o <-- master
       \
         o--o--o <-- 브랜치 B
```

만약 특정 커밋에 대해 말할 필요가 있을 때는, 소문자 "o" 대신 다른 기호나 숫자를 사용할 수도 있다.

변경 내역 이해하기: 브랜치란 무엇인가?#

정확한 용어를 사용해야 할 경우에는, "브랜치"란 용어는 개발 경로를 뜻하고, "브랜치 헤드" (혹은 그냥 "헤드")가 브랜치 상의 가장 최근 커밋을 의미하는 용어로 사용된다. 위의 예제에서 A라는 이름의 "브랜치 헤드"는 특정한 커밋을 가리키는 포인터이고, 이 지점에 도달하기 위한 세 커밋이 속한 경로는 "브랜치 A"의 일부라고 말한다.

하지만 혼동의 여지가 없는 경우에는 "브랜치"라는 용어를 브랜치 및 브랜치 헤드를 나타낼 때 모두 사용한다.

브랜치 다루기#

브랜치를 만들거나, 지우거나, 고치는 일은 쉽고 빠르게 처리된다. 다음은 이러한 명령들을 요약해 둔 것이다:

- **git branch**

모든 브랜치의 목록을 보여준다

- **git branch <브랜치>**

<브랜치>라는 이름으로, 변경 내역 상에서 현재 브랜치와 동일한 지점을 참조하는 브랜치를 새로 만든다

- **git branch <브랜치> <시작지점>**

<브랜치>라는 이름으로, <시작지점>을 참조하는 브랜치를 새로 만든다. <시작지점>은 다른 브랜치 이름이나 태그 이름 등을 포함한 어떠한 방식으로도 지정 가능하다.

- **git branch -d <브랜치>**

<브랜치>라는 이름의 브랜치를 삭제한다. 만약 삭제하려는 브랜치가 현재 브랜치에서 도달할 수 없는 커밋을 가리키는 경우에는, 경고를 보여주며 이 명령을 실패한다.

- **git branch -D <브랜치>**

삭제하려는 브랜치가 현재 브랜치에서 도달할 수 없는 커밋을 가리키는 경우에도, 해당 커밋이 다른 브랜치 혹은 태그를 통해 접근할 수 있다는 것을 알고 있을 수 있다. 이러한 경우 이 명령을 사용하여 git가 강제로 브랜치를 삭제하도록 할 수 있다.

- **git checkout <브랜치>**

<브랜치>를 현재 브랜치로 만든다. <브랜치>가 가리키는 버전을 반영하도록 작업 디렉토리를 내용을 갱신한다.

- **git checkout -b <새브랜치> <시작지점>**

<새브랜치>라는 이름으로, <시작지점>을 참조하는 브랜치를 새로 만들고, 체크아웃을 수행한다.

"HEAD"라는 이름의 특수 심볼은 항상 현재 브랜치의 최신 커밋을 가리키는 데 사용된다. 사실 git는 현재 브랜치를 기억하기 위해 .git 디렉토리 내에 "HEAD"라는 이름의 파일을 사용한다.

```
CODE $ cat .git/HEAD
     ref: refs/heads/master
```

새 브랜치를 만들지 않고 이전 버전 살펴보기#

git checkout 명령은 보통 브랜치 헤드를 인자로 받지만, 임의의 커밋을 인자로 넘길 수도 있다. 예를 들어 다음과 같이 특정 태그가 가리키는 커밋을 체크아웃할 수 있다:

```
CODE $ git checkout v2.6.17
     Note: moving to "v2.6.17" which isn't a local branch
     If you want to create a new branch from this checkout, 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 427abfa... Linux v2.6.17
```

이제 HEAD는 브랜치를 가리키는 대신 해당 커밋의 SHA-1 해시값을 가리키며, git branch 명령은 여러분이 이제 더 이상 브랜치 상에 있지 않다는 것을 보여준다:

```
CODE $ cat .git/HEAD
     427abfa28afedffadfca9dd8b067eb6d36bac53f
     $ git branch
     * (no branch)
       master
```

이러한 경우를 HEAD가 "분리"되었다고 (detached) 말한다.

이것은 새로운 브랜치를 만들지 않고 특정 버전을 체크아웃하기 위한 가장 손쉬운 방법이다. 원한다면 나중에 이 버전에 대한 브랜치 (혹은 태그)를 새로 만들 수 있다.

원격 저장소의 브랜치 살펴보기#

clone 명령을 수행할 때 생성되는 "master" 브랜치는 복사해온 저장소 내의 HEAD의 복사본이다. 하지만 이 저장소에는 다른 브랜치들도 존재할 수 있으며, 여러분의 로컬 저장소에도 이러한 원격 브랜치를 추적하기 위한 브랜치가 존재한다. 이러한 원격 브랜치들은 **git-branch(1)** 명령의 "-r" 옵션을 이용하여 볼 수 있다.

```
CODE $ git branch -r
       origin/HEAD
       origin/html
```

```
origin/maint
origin/man
origin/master
origin/next
origin/pu
origin/todo
```

이러한 원격 추적 브랜치를 직접 체크아웃할 수는 없다. 하지만 태그를 이용하는 것과 같이 브랜치를 직접 생성한 후 살펴볼 수 있다.

```
$ git checkout -b my-todo-copy origin/todo
```

"origin"이라는 이름은 단지 git가 복사해 온 저장소를 가리키기 위해 기본적으로 사용하는 이름이라는 것을 기억해 두기 바란다.

브랜치, 태그 및 다른 참조에 대한 이름 붙이기#

브랜치, 원격 추적 브랜치, 태그는 모두 커밋을 참조한다. 모든 참조들은 "refs"로 시작하고 슬래시(/) 기호로 구분되는 경로명으로 이름지어진다. 지금껏 우리가 사용해 온 이름들은 사실 상 줄임말에 해당한다:

- "test" 라는 브랜치는 "refs/heads/test"의 줄임말이다.
- "v2.6.18"이라는 태그는 "refs/tags/v2.6.18"의 줄임말이다.
- "origin/master"는 "refs/remotes/origin/master"의 줄임말이다.

완전한 이름은 (같은 이름의 브랜치와 태그가 동시에 존재하는 경우와 같이) 때때로 유용하게 사용된다.

(새로 생성된 참조들은 실제로 .git/refs 디렉토리 내의 주어진 이름의 경로에 해당하는 곳에 저장된다. 하지만 성능 상의 이유로 인해 이들은 하나의 파일로 합쳐질 수도 있다 - **`git-pack-refs(1)`** man 페이지를 살펴보기 바란다)

또 다른 유용한 줄임말로, 저장소의 "HEAD"는 단지 저장소의 이름으로 참조할 수 있다. 예를 들어 "origin"은 보통 "origin" 저장소 내의 HEAD 브랜치에 대한 줄임말로 사용된다.

git가 참조를 검사하는 경로의 완전한 목록과, 동일한 줄임말이 여러 경로 상에 존재할 때 어떤 것을 사용할 지 결정하는 순서는 **`git-rev-parse(1)`** man 페이지의 "리비전 지정하기" 부분을 살펴보기 바란다.

git-fetch를 이용하여 저장소 업데이트하기#

저장소를 복사해 온 개발자는 결국 자신의 저장소 내에서 추가적인 작업을 하고, 새로운 커밋을 생성하고, 새 커밋을 가리키도록 브랜치를 키워나갈 것이다.

"git fetch" 명령을 아무 인자 없이 실행하면, 모든 원격 추적 브랜치들은 해당 저장소 내의 가장 최신 버전으로 업데이트할 것이다. 이 명령은 여러분이 생성한 브랜치들은 전혀 손대지 않는다 - 여러분이 복사해 온 "master" 브랜치 자체도 변경되지 않는다.

다른 저장소에서 브랜치 가져오기#

`git-remote(1)` 명령을 이용하면 최초 복사해 온 저장소 이외의 저장소에서도 브랜치를 추적할 수 있다:

```
$ git remote add linux-nfs git://linux-nfs.org/pub/nfs-2.6.git
```

```
CODE $ git fetch linux-nfs
     * refs/remotes/linux-nfs/master: storing branch 'master' ...
       commit: bf81b46
```

새로 만들어진 원격 추적 브랜치는 "git remote add" 명령에서 인자로 넘긴 줄임말 이름 아래에 저장된다. 위의 예제에서는 linux-nfs에 해당한다:

```
CODE $ git branch -r
     linux-nfs/master
     origin/master
```

나중에 "git fetch <원격브랜치>" 명령을 실행하면 <원격브랜치>라는 이름의 원격 추적 브랜치가 업데이트 될 것이다.

.git/config 파일을 살펴본다면, git가 새로운 내용을 추가했음을 확인할 수 있다:

```
CODE $ cat .git/config
     ...
     [remote "linux-nfs"]
             url = git://linux-nfs.org/pub/nfs-2.6.git
             fetch = +refs/heads/*:refs/remotes/linux-nfs/*
     ...
```

이것은 git가 원격 저장소의 브랜치들을 추적할 수 있게 해 주는 것이다. 텍스트 편집기를 이용하여 .git/config 파일을 직접 수정하면 이러한 설정 옵션들을 변경하거나 삭제할 수 있다. (자세한 내용은 **`git-config(1)`** man 페이지의 "설정 파일" 부분을 살펴보기 바란다.)

Git 변경 내역 조사하기#

git는 여러 파일들에 대한 변경 내역을 저장하기 위한 도구로 최선의 선택이다. git는 파일 및 디렉토리의 내용들을 압축된 스냅샷으로 저장하고 각 스냅샷 간의 관계를 보여주는 "커밋"을 함께 저장하는 방식으로 이러한 작업을 수행한다.

git는 프로젝트의 변경 내역을 조사하기 위한 매우 유연하고 빠른 도구를 제공한다.

여기서는 프로젝트 내에 특정 버그를 만들어 낸 커밋을 찾아내는 데 유용한 도구를 먼저 살펴보도록 하자.

퇴보를 찾기 위해 bisect 이용하기#

여러분의 프로젝트가 2.6.18 버전에서는 동작하지만, "master" 버전에서는 문제가 발생한다고 가정해 보자. 때때로 이러한 퇴보(regression)의 원인을 찾기 위한 가장 좋은 방법은 문제를 발생시킨 특정 커밋을 찾을 때 까지 프로젝트의 변경 내역을 무식하게 검색하는 것이다. **`git-bisect(1)`** 명령은 이러한 작업을 도와줄 수 있다:

```
CODE $ git bisect start
     $ git bisect good v2.6.18
     $ git bisect bad master
     Bisecting: 3537 revisions left to test after this
     [65934a9a028b88e83e2b0f8b36618fe503349f8e] BLOCK: Make USB storage depend on SCSI rather than selecting it [try #6]
```

만약 이 상태에서 "git branch" 명령을 실행한다면, git는 임시로 "(no branch)" 상태로 전환되었다는 것을 확인할 수 있을 것이다. 이제 HEAD는 모든 브랜치에서 분리되었으며, "master"에서는 도달 가능하지만 v2.6.18에서는 도달할 수 없는 특정 커밋 (커밋 ID는 65934...)을 직접 가리키게된다. 소스를 컴파일해서 실행한 후 문제가 발생하는지 지켜보자. 여기에서는 여전히 문제가 발생한다고 가정한다. 그렇다면 다음과 같이 실행한다:

```
CODE $ git bisect bad
     Bisecting: 1769 revisions left to test after this
     [7eff82c8b1511017ae605f0c99ac275a7e21b867] i2c-core: Drop useless bitmaskings
```

이는 더 오래된 버전을 체크아웃한다. 이와 비슷하게 각 단계 별로 해당 버전이 잘 동작하는지 여부를 git에게 알려주고 이 과정을 반복한다. 눈여겨 볼 것은 테스트할 버전의 수는 각 단계 마다 대략 1/2로 감소한다는 것이다.

(위의 예제에서) 13번의 테스트 끝에, 문제가 되는 커밋의 커밋 ID를 출력하게 되었다. 이제 **`git-show(1)`** 명령을 통해 커밋 정보를 살펴보고, 누가 이 커밋을 작성했는지 알아냈다면 이메일로 커밋 ID를 포함한 버그 보고서를 보낼 수 있다. 마지막으로는 다음을 실행한다.

```
CODE $ git bisect reset
```

이는 이전에 작업하던 브랜치로 돌아가도록 해 준다.

'git bisect' 명령이 각 단계 별로 검사하는 버전은 단지 제안일 뿐이며, 그것이 올바르다고 판단되는 경우에는 다른 버전을 대신 검사해 볼 수 있다. 예를 들어, 때때로 관련 없는 부분에서 문제를 일으키는 커밋에 도달했을 수도 있다. 이 때 다음 명령을 실행한다

```
CODE $ git bisect visualize
```

이는 gitk 명령을 실행하고 선택한 커밋에 "bisect"라고 써진 마크와 레이블을 보여줄 것이다. 그 근처에 있는 안전해 보이는 커밋을 선택하고, 커밋 ID를 메모해 둔 후 다음과 같이 해당 커밋을 체크아웃 한다:

```
CODE $ git reset --hard fb47ddb2db...
```

다시 테스트를 수행하고 그 결과에 따라 "bisect good" 혹은 "bisect bad"를 실행하는 과정을 반복한다.

"git bisect visualize" 명령과 "git reset --hard fb47ddb2db..." 명령을 수행하는 대신, git가 현재 커밋을 건너뛰도록 하기를 원할 수도 있다:

```
CODE $ git bisect skip
```

하지만 이 경우에 git는 최초에 건너뛴 커밋과 이후의 문제 있는 커밋 사이에서 어느 것이 처음으로 문제가 발생된 커밋인지 판단하지 못할 수도 있다.

만약 해당 커밋에 문제가 있는지 여부를 판단할 수 있는 테스트 스크립트를 가지고 있다면 이진 탐색(bisect) 과정을 자동화할 수 있는 방법들도 존재한다. 이에 대한 자세한 내용 및 다른 "git bisect" 기능에 대한 내용은 **`git-bisect(1)`** man 페이지를 살펴보기 바란다.

커밋 이름 붙이기#

지금까지 특정 커밋에 이름을 붙이는 여러가지 방법을 살펴 보았다:

- 40자의 16진수 객체 이름
- 브랜치 이름 : 해당 브랜치의 헤드에 있는 커밋을 참조한다

- 태그 이름 : 해당 태그가 가리키는 커밋을 참조한다 (우리는 브랜치와 태그와 특별한 형태의 참조라는 것을 살펴 보았다)
- HEAD : 현재 브랜치의 헤드에 있는 커밋을 참조한다

이 외에도 여러가지 방법이 존재한다. 커밋에 이름을 붙이는 방법에 대한 완전한 목록은 **`git-rev-parse(1)`** man 페이지의 "리비전 지정하기" 부분을 살펴보기 바란다. 다음은 몇 가지 예제를 보여준다:

```
CODE $ git show fb47ddb2 # 보통은 객체 이름 첫 부분의 몇 글자로도
                         # 해당 커밋을 지정하기에 충분하다
     $ git show HEAD^    # HEAD의 부모 커밋
     $ git show HEAD^^   # HEAD의 부모의 부모 커밋
     $ git show HEAD~4   # HEAD의 부모의 부모의 부모의 부모 커밋
```

머지 커밋의 경우에는 둘 이상의 부모를 가질 수 있다는 것을 기억하자. 기본적으로 ^ 와 ~ 는 해당 커밋에 연결된 첫 번째 부모를 따라간다. 하지만 다음 명령을 사용하여 다른 부모를 선택할 수 있다:

```
CODE $ git show HEAD^1   # HEAD의 첫 번째 부모 커밋
     $ git show HEAD^2   # HEAD의 두 번째 부모 커밋
```

HEAD 이외에도 커밋을 가리키는 특수한 이름들이 더 존재한다.

(이후에 살펴 볼) 머지 및 'git reset'과 같이 현재 체크아웃된 커밋을 변경하는 명령들은 일반적으로 해당 명령이 수행되기 전의 HEAD의 값으로 ORIG_HEAD를 설정한다.

'git fetch' 명령은 항상 마지막으로 가져온 브랜치의 헤드를 FETCH_HEAD에 저장한다. 예를 들어 명령의 대상으로 로컬 브랜치를 지정하지 않고 'git fetch' 명령을 실행했다면

```
CODE $ git fetch git://example.com/proj.git theirbranch
```

가져온 커밋들은 FETCH_HEAD를 통해 접근할 수 있을 것이다.

머지에 대해 설명할 때 현재 브랜치와 통합하는 다른 브랜치를 가리키는 MERGE_HEAD라는 특수한 이름도 살펴볼 것이다.

`git-rev-parse(1)` 명령은 때때로 커밋에 대한 이름을 해당 커밋의 객체 이름으로 변환할 때 유용하게 사용되는 저수준 명령이다.

```
CODE $ git rev-parse origin
     e05db0fd4f31dde7005f075a84f96b360d05984b
```

태그 만들기#

특정 커밋을 참조하는 태그를 만드는 것이 가능하다. 다음 명령을 실행하고 나면

```
CODE $ git tag stable-1 1b2e1d63ff
```

커밋 1b2e1d63ff를 가리키기 위해 stable-1을 사용할 수 있다.

이것은 "가벼운" (lightweight) 태그를 만든다. 태그에 설명을 추가하거나 이에 암호화된 서명을 하려는 경우에는 가벼운 태그 대신 태그

객체를 만들어야 한다. 이에 대한 자세한 내용은 **git-tag(1)** man 페이지를 살펴보기 바란다.

버전 살펴보기#

git-log(1) 명령을 커밋의 목록을 보여줄 수 있다. 기본적으로 이 명령은 부모 커밋에서부터 도달할 수 있는 모든 커밋들을 보여주지만 특정한 범위를 지정할 수도 있다:

```
CODE $ git log v2.5..        # v2.5부터의 커밋 (v2.5에서는 도달할 수 없는 커밋)
     $ git log test..master  # master에서는 도달할 수 있지만 test에서는 도달할 수 없는 커밋
     $ git log master..test  # test에서는 도달할 수 있지만 master에서는 도달할 수 없는 커밋
     $ git log master...test # test 혹은 master 중 하나에서 도달할 수 있는 있는 커밋
                             #    하지만 둘 다 도달할 수 있는 커밋들은 제외
     $ git log --since="2 weeks ago" # 지난 2 주 간의 커밋
     $ git log Makefile      # Makefile 파일을 수정한 커밋
     $ git log fs/           # fs/ 디렉토리 내의 파일들을 수정한 커밋
     $ git log -S'foo()'     # 문자열 'foo()'에 매칭되는 어떠한 파일 데이터라도
                             # 추가 혹은 삭제한 커밋
```

물론 이러한 조건들을 함께 사용할 수 있다. 아래의 명령은 v2.5 이후로 Makefile이나 fs 디렉토리 내의 어떤 파일을 수정한 커밋들을 찾아준다:

```
CODE $ git log v2.5.. Makefile fs/
```

또한 git log에게 패치 형식으로 보여달라고 요청할 수도 있다.

```
CODE $ git log -p
```

다양한 표시 옵션에 대해서는 **git-log(1)** man 페이지의 "--pretty" 옵션 부분을 살펴보기 바란다.

git log는 가장 최신의 커밋에서부터 시작하여 부모의 경로를 따라 역방향으로 출력한다는 것을 알아두도록 하자. 하지만 git 변경 내역은 다중 개발 경로를 포함할 수 있기 때문에 커밋의 목록이 출력되는 상세한 순서는 약간 달라질 수 있다.

차이점 생성하기#

git-diff(1) 명령을 이용하여 임의의 두 버전 간의 차이점(diff)를 생성할 수 있다:

```
CODE $ git diff master..test
```

위 명령은 두 브랜치의 최신 내용(tip) 간의 차이점을 만들어 낼 것이다. 만약 두 브랜치의 공통 조상으로부터 test까지의 차이점을 찾아보고 싶다면 점(.)을 두개 대신 세 개 사용하면 된다.

```
CODE $ git diff master...test
```

때로는 여러 개의 패치 형태로 출력하고 싶을 수도 있다. 이를 위해서는 **git-format-patch(1)** 명령을 이용할 수 있다.

```
CODE $ git format-patch master..test
```

위 명령은 test에서는 도달할 수 있지만 master에서는 도달할 수 없는 각 커밋에 대한 패치에 해당하는 파일들을 생성할 것입니다.

예전 버전의 파일 보기#

예전 버전으로 체크아웃하고 나면 해당 버전의 파일을 언제나 볼 수 있다. 하지만 때때로 체크아웃없이 한 파일의 예전 버전을 볼 수 있다면 좀 더 편리할 것이다. 다음 명령은 이 같은 작업을 수행한다:

```
CODE $ git show v2.5:fs/locks.c
```

콜론(:) 앞에는 커밋을 가리키는 어떠한 이름도 올 수 있고, 콜론 뒤에는 git에서 관리하는 어떠한 파일에 대한 경로도 올 수 있다.

예제#

특정 브랜치 상의 커밋 개수 세기#

여러분이 "origin" 브랜치에서 "mybranch" 브랜치를 만든 이후에 얼마나 많은 커밋을 했는지 알고 싶다고 가정해 보자:

```
CODE $ git log --pretty=oneline origin..mybranch | wc -l
```

다른 방법으로, 이러한 작업을 주어진 모든 커밋에 대한 SHA-1 해시값의 목록을 보여주는, 보다 저수준 명령인 **git-rev-list(1)** 명령을 통해 수행하는 것을 볼 수 있을 것이다:

```
CODE $ git rev-list origin..mybranch | wc -l
```

두 브랜치가 동일한 지점을 가리키는지 검사하기#

두 브랜치가 변경 내역 상의 같은 지점을 가리키는지 알고 싶다고 가정해 보자.

```
CODE $ git diff origin..master
```

위 명령은 두 브랜치에서 프로젝트의 내용이 동일한지 여부를 보여줄 것이다. 하지만 이론적으로, 동일한 프로젝트의 내용이 서로 다른 변경 내역을 거쳐 만들어졌을 수도 있다. 객체 이름을 이용하여 이를 비교할 수 있다:

```
CODE $ git rev-list origin
    e05db0fd4f31dde7005f075a84f96b360d05984b
    $ git rev-list master
    e05db0fd4f31dde7005f075a84f96b360d05984b
```

혹은 ... 연산자가 두 참조 중의 하나에서 도달할 수 있지만 둘 다에서는 도달할 수 없는 모든 커밋들을 선택한다는 것을 기억해 볼 수 있다. 따라서

```
CODE $ git log origin...master
```

두 브랜치가 동일할 경우 위 명령은 아무 커밋도 보여주지 않을 것이다.

특정 커밋을 포함하는 것 중에서 제일 먼저 태그가 붙은 버전 찾기#

e05db0fd 커밋이 어떤 문제를 해결한 것이라고 가정해 보자. 여러분은 이 해결책을 포함한 것 중 가장 먼저 태그가 붙은 배포 버전을 찾고 싶다.

물론 이를 위한 방법은 여러가지 존재한다. 만약 변경 내역 상에서 e05db0fd 커밋 이후에 브랜치가 생성되었다면 여러 개의 "가장 먼저" 태그가 붙은 배포 버전이 존재할 수 있다.

다음과 같이 e05db0fd 이후의 커밋들을 그래프로 볼 수 있다:

```
$ gitk e05db0fd..
```

또는 해당 커밋의 자손 중의 하나를 가리키는 태그를 찾아 이를 바탕으로 해당 커밋에 이름을 붙이는 **git-name-rev(1)** 명령을 이용할 수 있다:

```
$ git name-rev --tags e05db0fd
e05db0fd tags/v1.5.0-rc1^0~23
```

git-describe(1) 명령은 반대의 작업을 수행하는 데, 이는 주어진 커밋에서 도달할 수 있는 태그를 사용하여 버전에 이름을 붙인다:

```
$ git describe e05db0fd
v1.5.0-rc0-260-ge05db0f
```

하지만 이는 때때로 해당 커밋 다음에 어떤 태그를 사용해야 할 지 결정할 때 도움을 줄 수 있다.

만약 어떤 태그가 가리키는 버전이 주어진 커밋을 포함하는지 만을 알아보고 싶다면 **git-merge-base(1)** 명령을 이용한다:

```
$ git merge-base e05db0fd v1.5.0-rc1
e05db0fd4f31dde7005f075a84f96b360d05984b
```

merge-base 명령은 주어진 커밋들의 공통 조상을 찾는데, 주어진 두 커밋 중의 하나가 다른 것의 후손이라면 항상 (조상에 해당하는) 이 둘 중의 하나를 반환한다. 즉 위의 결과에서 e05db0fd 커밋은 실제로 v1.5.0-rc1의 조상임을 보여준다.

다른 방법으로

```
$ git log v1.5.0-rc1..e05db0fd
```

오직 v1.5.0-rc1 커밋이 e05db0fd 커밋을 포함하고 있는 경우에만 위 명령은 아무런 결과도 출력하지 않을 것이다. 왜냐하면 위 명령은 v1.5.0-rc1에서 도달할 수 없는 커밋 만을 출력하기 때문이다.

또 다른 방법으로 **git-show-branch(1)** 명령은 주어진 인자에서 도달할 수 있는 커밋의 목록을 보여주는데, 왼편에 어떤 인자들이 해당 커밋에 도달할 수 있는지를 표시해준다. 따라서 다음과 같은 명령을 실행하고

```
$ git show-branch e05db0fd v1.5.0-rc0 v1.5.0-rc1 v1.5.0-rc2
! [e05db0fd] Fix warnings in sha1_file.c - use C99 printf format if
available
```

```
CODE  ! [v1.5.0-rc0] GIT v1.5.0 preview
       ! [v1.5.0-rc1] GIT v1.5.0-rc1
        ! [v1.5.0-rc2] GIT v1.5.0-rc2
      ...
```

다음과 같은 줄을 검색한다

```
CODE + ++ [e05db0fd] Fix warnings in sha1_file.c - use C99 printf format if
     available
```

이 결과는 e05db0fd 커밋이 자기 자신과 v1.5.0-rc1, v1.5.0-rc2에서는 도달 가능하지만 v1.5.0-rc0에서는 도달할 수 없다는 것을 보여준다.

주어진 브랜치에만 존재하는 커밋 보기#

"master"라는 이름의 브랜치 헤드에서는 도달할 수 있지만 저장소 내의 다른 어떤 헤드에서도 도달할 수 없는 모든 커밋들을 보고 싶다고 가정해 보자.

git-show-ref(1) 명령을 이용하여 저장소 내의 모든 헤드의 목록을 볼 수 있다:

```
CODE $ git show-ref --heads
     bf62196b5e363d73353a9dcf094c59595f3153b7 refs/heads/core-tutorial
     db768d5504c1bb46f63ee9d6e1772bd047e05bf9 refs/heads/maint
     a07157ac624b2524a059a3414e99f6f44bebc1e7 refs/heads/master
     24dbc180ea14dc1aebe09f14c8ecf32010690627 refs/heads/tutorial-2
     1e87486ae06626c2f31eaa63d26fc0fd646c8af2 refs/heads/tutorial-fixes
```

cut과 grep이라는 표준 도구를 사용하여 "master"를 제외한 모든 브랜치 헤드의 목록을 얻을 수 있다:

```
CODE $ git show-ref --heads | cut -d' ' -f2 | grep -v '^refs/heads/master'
     refs/heads/core-tutorial
     refs/heads/maint
     refs/heads/tutorial-2
     refs/heads/tutorial-fixes
```

그럼 이제 master에서는 도달할 수 있지만 위의 헤더에서는 도달할 수 없는 커밋들의 목록을 보여달라고 요청할 수 있다:

```
CODE $ gitk master --not $( git show-ref --heads | cut -d' ' -f2 |
                                  grep -v '^refs/heads/master' )
```

물론 이를 계속 변형하는 것이 가능하다. 예를 들어 어떤 헤드에서는 도달할 수 있지만 저장소 내의 어떤 태그에서도 도달할 수 없는 커밋들의 목록을 보려면 다음 명령을 실행한다:

```
CODE $ gitk $( git show-ref --heads ) --not  $( git show-ref --tags )
```

('--not'과 같은 커밋 선택 문법에 대한 설명은 **git-rev-parse(1)** man 페이지를 보기 바란다)

소프트웨어 배포를 위한 변경 로그 및 tarball 만들기#

git-archive(1) 명령은 프로젝트 내의 어떤 버전에서도 tar 혹은 zip 압축 파일을 만들 수 있다. 예를 들어:

```
$ git archive --format=tar --prefix=project/ HEAD | gzip >latest.tar.gz
```

위 명령은 HEAD 버전에 해당하는 tar 아카이브 파일을 만드는데, 모든 파일 이름의 앞에는 "project/"가 추가될 것이다.

만약 여러분이 소프트웨어 프로젝트의 새 버전을 배포하려 한다면 배포 공지 사항에 변경 로그도 추가하고 싶을 것이다.

예를 들면, 리누스 토발즈는 커널을 배포할 때, 태그를 붙인 다음 아래와 같은 명령을 실행한다:

```
$ release-script 2.6.12 2.6.13-rc6 2.6.13-rc7
```

release-script는 다음과 같은 쉘 스크립트이다:

```
#!/bin/sh
    stable="$1"
    last="$2"
    new="$3"
    echo "# git tag v$new"
    echo "git archive --prefix=linux-$new/ v$new | gzip -9 > ../linux-$new.tar.gz"
    echo "git diff v$stable v$new | gzip -9 > ../patch-$new.gz"
    echo "git log --no-merges v$new ^v$last > ../ChangeLog-$new"
    echo "git shortlog --no-merges v$new ^v$last > ../ShortLog"
    echo "git diff --stat --summary -M v$last v$new > ../diffstat-$new"
```

그리고 정상적으로 출력되는 지 확인한 후, 출력된 명령들을 잘라서 붙여넣기로 실행한다.

파일의 특정 내용을 참조하는 커밋 찾기#

누군가 여러분에게 어떤 파일의 복사본을 주면서, 어떤 커밋이 이 파일을 수정하였는지, 즉 해당 커밋의 이전 혹은 이후에 주어진 내용이 포함되어 있는지 물어본다. 다음 명령을 이용하여 이를 알아볼 수 있다:

```
$  git log --raw --abbrev=40 --pretty=oneline |
        grep -B 1 `git hash-object filename`
```

이것이 어떻게 가능한지는 (우수한) 학생들을 위한 숙제로 남겨두겠다. **git-log(1)**, **git-diff-tree(1)**, **git-hash-object(1)** man 페이지가 도움이 될 것이다.

Git를 이용하여 개발하기#

git에게 이름 말해주기#

커밋을 생성하기 전에, git에게 여러분을 소개해야 한다. 이를 위한 가장 간단한 방법은 홈 디렉토리 내의 .gitconfig 파일에 다음과 같은

내용을 기록하는 것이다:

```
CODE [user]
        name = 여기에 이름을 적는다
        email = you@yourdomain.example.com
```

(설정 파일에 대한 자세한 설명은 **`git-config(1)`** man 페이지의 "설정 파일" 부분을 살펴보기 바란다)

새 저장소 만들기#

아무 것도 없는 상태에서 저장소를 새로 만드는 것은 아주 간단하다:

```
CODE $ mkdir project
     $ cd project
     $ git init
```

초기에 사용할 어떤 내용물이 있다면 (말하자면 tarball 같은 것) 다음 명령을 실행한다:

```
CODE $ tar xzvf project.tar.gz
     $ cd project
     $ git init
     $ git add . # 최초 커밋 시에 ./ 아래의 모든 것을 추가한다
     $ git commit
```

커밋을 만드는 방법#

새로운 커밋을 만드려면 다음과 같은 세 단계를 거친다:

1. 주로 사용하는 편집기를 이용하여 작업 디렉토리 상에 어떤 변경 사항을 만든다
2. git에게 변경 사항을 알려준다
3. 위 단계에서 git에게 알려준 내용을 이용하여 커밋을 만든다

실제로는 원하는 만큼 과정 1과 2를 계속 (순서를 섞어서도) 반복할 수 있다. 3 단계에서 커밋할 내용들을 추적하기 위해 git는 "인덱스"라고 하는 특별한 준비 영역에 트리 내용들의 스냅샷을 유지한다.

처음에는, 인덱스의 내용은 HEAD의 내용과 같을 것이다. 따라서 HEAD와 인덱스 간의 차이점을 보여주는 "git diff --cached" 명령은, 이 시점에서는 아무 것도 출력하지 않을 것이다.

인덱스를 수정하는 것은 간단하다:

수정한 파일의 새 내용을 이용하여 인덱스를 갱신하려면 다음을 실행한다.

```
CODE $ git add path/to/file
```

새 파일의 내용을 인덱스에 추가하려면 다음을 실행한다.

```
CODE $ git add path/to/file
```

인덱스와 작업 트리에서 파일을 삭제하려면 다음을 실행한다.

```
CODE $ git rm path/to/file
```

각각의 단계에서 다음 명령을 실행하면

```
CODE $ git diff --cached
```

항상 HEAD와 인덱스 간의 차이점을 보여준다는 것을 확인할 수 있다. 이것은 지금 커밋을 만든다고 할 때 적용되는 내용이다. 그리고

```
CODE $ git diff
```

위 명령은 작업 트리와 인덱스 간의 차이점을 보여준다.

기억해야 할 것은 "git add" 명령은 단지 해당 파일의 현재 내용을 인덱스에 추가할 뿐이므로, 이 파일을 더 수정하는 경우 다시 'git add' 명령을 수행하지 않는다면 이후의 수정 사항들은 무시될 것이라는 점이다.

준비가 되었다면 단지 아래의 명령을 실행한다.

```
CODE $ git commit
```

그러면 git가 커밋 메시지를 입력하도록 프롬프트를 띄울 것이고, 그 후에 커밋을 생성한다. 커밋 메시지를 입력할 때는 아래의 명령을 실행했을 때 나타나는 것과 비슷한 형태로 작성한다.

```
CODE $ git show
```

특수한 바로 가기로써

```
CODE $ git commit -a
```

위 명령은 수정 혹은 삭제된 모든 파일을 이용하여 인덱스를 갱신하고, 커밋을 생성하는 일을 한 번에 다 수행할 것이다.

다음의 몇 가지 명령은 커밋 시에 어떤 것들이 반영되는 지 추적하기에 유용하다:

```
CODE $ git diff --cached # HEAD와 인덱스 간의 차이점. 지금 바로
                         # "commit" 명령을 수행할 경우 반영되는 내용들
     $ git diff          # 인덱스 파일과 작업 디렉토리 간의 차이점.
                         # 지금 바로 "commit" 명령을 수행할 경우
                         # 반영되지 않는 변경 사항들
     $ git diff HEAD     # HEAD와 작업 디렉토리 간의 차이점. 지금 바로
                         # "commit -a" 명령을 수행할 경우 반영되는 내용들
     $ git status        # 위 내용에 대한 파일 별 간략한 요약 정보
```

또는 **git-gui(1)** 명령을 이용하여 인덱스 혹은 작업 디렉토리 내의 변경 사항을 보거나, 커밋을 생성하는 것이 가능하다. 또한 인덱스

내의 차이점들 중에서 (마우스 오른쪽 버튼을 클릭하여 "Stage Hunk for Commit"을 선택하면) 커밋 시에 포함할 내용들을 별도로 선택하는 일도 할 수 있다.

좋은 커밋 메시지 작성하기#

이것이 반드시 지켜져야 하는 것은 아니지만, 커밋 메시지를 작성할 때는 변경 사항을 요약하는 (50자 이내의) 짧은 한 줄 짜리 메시지로 시작하고, 빈 줄을 넣은 뒤, 보다 상세한 설명을 기록하는 형식으로 작성하는 것이 좋다. 예를 들어, 커밋 내용을 이메일로 변환하는 도구는 첫 줄의 내용을 메일의 제목으로 사용하고 나머지 부분은 메일의 본문으로 사용한다.

파일 무시하기#

프로젝트는 때로 git가 관리하지 않아야 할 파일들을 생성하기도 한다. 보통 여기에는 빌드 과정에서 생기는 파일이나 여러분이 사용하는 편집기가 만드는 임시 백업 파일 등이 포함된다. 물론 이러한 파일들을 git에서 관리하지 않게 하려면 단지 해당 파일에 대해 'git add' 명령을 수행하지 않으면 된다. 하지만 이렇게 관리되지 않는 파일들이 생기면 성가신 문제들이 발생한다. 예를 들면, 이러한 파일들이 있을 때는 'git add .' 명령을 거의 이용할 수 없으며, 'git status' 명령의 출력에도 항상 포함된다.

작업 디렉토리의 최상위에 .gitignore라는 파일을 만들어두면 git에게 특정 파일을 무시하라고 알려줄 수 있다. 이 파일의 내용은 다음과 같은 내용을 포함할 수 있다:

```
CODE # '#'으로 시작하는 줄은 주석으로 처리된다
     # foo.txt라는 이름의 파일들은 모두 무시한다
     foo.txt
     # (자동 생성된) HTML 파일들을 무시한다
     *.html
     # 하지만 직접 작성하는 foo.html 파일은 예외로 한다
     !foo.html
     # 오브젝트 파일과 아카이브 파일들을 무시한다
     *.[oa]
```

문법에 대한 자세한 설명은 **`gitignore(5)`** man 페이지를 살펴보기 바란다. 또한 .gitignore 파일을 작업 디렉토리 상의 어떤 디렉토리에도 넣어둘 수 있으며, 이 경우 그 내용은 해당 디렉토리와 그 하위 디렉토리에 적용된다. '.gitignore' 파일은 다른 파일들과 마찬가지로 (평소와 같이 'git add .gitignore' 와 'git commit' 명령을 수행하여) 저장소에 추가될 수 있으며, 무시할 패턴이 (빌드 출력 파일과 매칭되는 패턴과 같이) 여러분의 저장소를 복사해 갈 다른 사용자들에게도 적용되는 경우에 편리하다.

만약 무시할 패턴이 (해당 프로젝트에 대한 모든 저장소가 아닌) 특정 저장소에서만 동작하도록 하고 싶다면, 이러한 내용을 해당 저장소 내의 .git/info/exclude 파일에 적어두거나 'core.excludefile' 설정 변수에 지정된 파일에 적어두면 된다. 몇몇 git 명령은 명령행에서 무시할 패턴을 직접 넘겨받을 수도 있다. 자세한 내용은 `gitignore(5)` man 페이지를 살펴보기 바란다.

머지하는 방법#

`git-merge(1)` 명령을 이용하여 개발 과정에서 나누어진 두 브랜치를 다시 합치는 것이 가능하다.

```
CODE $ git merge branchname
```

위 명령은 "branchname"이라는 브랜치의 개발 내용을 현재 브랜치로 통합(머지)한다. 만약 충돌이 발생한 경우 (예를 들면 원격 브랜치와 로컬 브랜치에서 동일한 파일이 서로 다른 방식으로 변경된 경우) 경고를 보여준다. 출력되는 내용은 아래와 같은 형태가 될 것이다.

```
CODE $ git merge next
      100% (4/4) done
     Auto-merged file.txt
     CONFLICT (content): Merge conflict in file.txt
     Automatic merge failed; fix conflicts and then commit the result.
```

문제가 생긴 파일에는 충돌 표시가 남아있게 된다. 이를 직접 해결하고나면 새 파일을 만들 때와 같이 변경된 내용으로 인덱스를 갱신하고 "git commit" 명령을 수행한다.

만약 이러한 커밋을 gitk 명령으로 살펴본다면, 해당 커밋이 (현재 브랜치의 끝을 가리키는 것과 다른 브랜치의 끝을 가리키는) 두 개의 부모를 가지고 있는 것을 보게 될 것이다.

머지 (충돌) 해결하기#

머지 시에 (충돌이) 자동으로 해결되지 않는 경우, git는 인덱스와 작업 디렉토리를 충돌을 해결하기에 필요한 모든 정보를 저장하는 특수한 상태로 남겨둔다.

충돌이 발생한 파일들은 인덱스 내에 특별하게 표시되므로, 문제점을 해결하고 인덱스를 갱신하기 전까지 **git-commit(1)** 명령은 실패할 것이다:

```
CODE $ git commit
     file.txt: needs merge
```

또한 **git-status(1)** 명령은 이 파일들을 "머지되지 않은" (unmerged) 상태로 출력하며, 충돌이 발생한 파일들은 아래와 같이 충돌 표시를 포함하게 될 것이다:

```
CODE <<<<<<< HEAD:file.txt
     Hello world
     =======
     Goodbye
     >>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

여러분이 해야할 일은 파일을 편집하여 충돌을 해결하고 다음 명령을 실행하는 것이다.

```
CODE $ git add file.txt
     $ git commit
```

커밋 메시지는 머지에 대한 정보를 포함하여 미리 작성된다는 것을 알아두기 바란다. 일반적으로 이러한 기본 메시지를 그대로 사용할 수 있지만 원한다면 부가적인 메시지를 더 입력할 수 있다.

위 사항은 단순한 머지 시의 문제를 해결하기 위해 반드시 알아두어야 할 것이다. 하지만 git는 충돌을 해결하기 위해 필요한 자세한 정보들을 제공해 준다.

머지 중에 충돌을 해결하기 위해 필요한 정보 얻기#

git가 자동으로 머지할 수 있는 모든 변경 사항들은 인덱스 파일에 이미 추가되었으므로, **git-diff(1)** 명령은 오직 충돌한 내용 만을 보여준다. 여기에는 보통과는 다른 문법이 사용된다:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD:file.txt
 +Hello world
++=======
+ Goodbye
++>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

이 충돌 사항을 해결한 후에 생성될 커밋은 보통과는 달리 두 개의 부모를 가진다는 것을 기억해 보자. 하나는 현재 브랜치의 끝인 HEAD이고, 다른 하나는 임시로 MERGE_H
EAD에 저장된, 다른 브랜치의 끝이 될 것이다.

머지 중에 인덱스는 각 파일들에 대한 세 가지 버전을 가지고 있다. 이러한 세 "파일 스테이지" 각각은 해당 파일의 서로 다른 버전을 나타낸다:

```
$ git show :1:file.txt  # 두 부모의 공통 조상에 속한 파일
$ git show :2:file.txt  # HEAD에 있는 버전
$ git show :3:file.txt  # MERGE_HEAD에 있는 버전
```

충돌 내용을 보기 위해 **git-diff(1)** 명령을 실행하면, 양쪽에서 함께 온 변경 사항들만을 같이 보여주기 위해 스테이지 2와 스테이즈 3의 작업 트리 내의 머지 충돌 결과 간의 3-방향 차이점 보기를 실행한다. (다시 말해, 머지 결과 내의 어떤 변경 사항이 스테이지 2에서만 왔다면 이 부분은 충돌이 발생하지 않으며 따라서 출려되지 않는다. 이는 스테이지 3에 대해서도 마찬가지이다.)

위의 diff 명령은 작업 트리 버전의 file.txt와 스테이지 2와 스테이지 3 버전 간의 차이점을 보여준다. 따라서 각 줄의 왼쪽에 하나의 "+" 혹은 "-" 기호를 사용하는 대신 두 개의 열(column)을 사용한다. 첫 번째 열은 첫 번째 부모와 작업 디렉토리 복사본 간의 차이점을 위해 사용되고, 두 번째 열은 두 번째 부모와 작업 디렉토리 복사본 간의 차이점을 위해 사용된다. (이 형식에 대한 자세한 설명은 **git-diff-files(1)** man 페이지의 "통합 차이점 형식" 부분을 살펴보기 바란다.)

명확한 방식으로 충돌을 해결한 후에 (하지만 아직 인덱스를 갱신하기 전에) diff 명령을 실행하면 다음과 같이 보일 것이다:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,1 @@@
- Hello world
 -Goodbye
++Goodbye world
```

이것은 우리가 해결한 버전이 첫 번째 부모에서 "Hello world"를 지우고, 두 번째 부모에서 "Goodbye"를 지우고, 양 쪽 부모에 모두 포함되지 않았던 "Goodbye world"를 추가했다는 것을 보여준다.

몇 가지 특별한 diff 옵션은 작업 디렉토리의 각각의 스테이지에 대한 차이점을 보여줄 수 있다:

```
CODE $ git diff -1 file.txt          # 스테이지 1에 대한 차이점
     $ git diff --base file.txt      # 위와 동일
     $ git diff -2 file.txt          # 스테이지 2에 대한 차이점
     $ git diff --ours file.txt      # 위와 동일
     $ git diff -3 file.txt          # 스테이지 3에 대한 차이점
     $ git diff --theirs file.txt    # 위와 동일
```

git-log(1) 명령과 **gitk(1)** 명령은 머지에 대한 특별한 도움도 제공한다:

```
CODE $ git log --merge
     $ gitk --merge
```

위 명령은 머지되지 않은 파일을 수정하는 모든 커밋 중 HEAD 혹은 MERGE_HEAD에만 존재하는 것들을 보여줄 것이다.

또는 이맥스나 kdiff3과 같은 외부 도구를 이용하여 머지되지 않은 파일을 머지할 수 있도록 해 주는 **git-mergetool(1)** 명령을 이용할 수도 있다.

파일 내의 충돌을 해결하고 인덱스를 갱신할 때 마다:

```
CODE $ git add file.txt
```

'git diff' 명령이 더 이상 (기본 상태대로) 해당 파일에 대한 아무런 출력도 보여주지 않게된 후에, 해당 파일의 서로 다른 스테이지들은 "사라질" (collapse) 것이다.

머지 되돌리기#

만약 여러분이 머지 과정에서 엉켜버려서 그냥 포기하고 원래 상태로 되돌리고 싶다면, 언제나 다음 명령을 이용하여 머지 이전의 상태로 돌아갈 수 있다.

```
CODE $ git reset --hard HEAD
```

혹은 머지한 것을 이미 커밋했을 때 되돌리려면 다음 명령을 실행한다.

```
CODE $ git reset --hard ORIG_HEAD
```

하지만, 위의 명령은 몇몇 경우에 위험해 질 수 있다. 기존에 생성한 어떤 커밋이 다른 브랜치에 머지된 경우 해당 커밋을 절대로 되돌리면 안된다. 이 경우 이후의 머지 작업에 혼란을 일으킬 수 있다.

고속 이동 머지#

위에서 언급하지 않은, 특별히 처리되는 한 가지 특수한 경우가 있다. 보통 머지의 결과는 두 개의 개발 경로를 하나로 합치는 머지 커밋으로 나타난다.

하지만 만약 현재 브랜치가 다른 브랜치의 자손 (역주: "조상"이 맞는 듯..) 인 경우 (따라서 한 쪽에 존재하는 모든 커밋들이 다른 쪽에 이미 포함되어 있다면) git는 단지 "고속 이동" (fast-forward)을 수행한다. 이는 아무런 커밋도 생성하지 않고 현재 브랜치의 헤드를 머지된 브랜치의 헤드가 가리키는 곳으로 이동시킨다.

실수 바로잡기#

만약 여러분이 작업 트리를 망가뜨렸지만 아직 이 실수에 대한 커밋을 생성하지 않았다면 다음 명령을 이용하여 작업 트리 전체를 이전의 커밋 상태로 되돌릴 수 있다

```
CODE $ git reset --hard HEAD
```

만약 커밋으로 만들지 말아야 할 사항을 만들어 버렸다면, 이 문제를 해결하기 위한 두 가지 다른 방법이 존재한다:

1. 이전 커밋에서 작업한 내용을 모두 되돌린 새로운 커밋을 만들 수 있다. 이것은 여러분의 실수가 이미 외부에 공개된 경우에 적당하다.
2. 뒤로 되돌아가서 이전 커밋을 수정한다. 만약 변경 내역이 이미 공개되었다면 절대 이렇게 해서는 안 된다. git는 일반적으로 프로젝트의 "변경 내역"이 변경되는 것을 고려하지 않기 때문에, 변경 내역이 변경된 브랜치로의 반복된 머지 작업은 올바로 동작하지 않을 것이다.

새로운 커밋을 만들어 실수 바로잡기#

이전의 변경 사항을 되돌리는 새로운 커밋을 만드는 일은 아주 간단하다. **`git-revert(1)`** 명령에게 실수한 커밋에 대한 참조를 넘겨주면 된다. 예를 들어 가장 최근의 커밋을 되돌리려면 다음을 실행한다:

```
CODE $ git revert HEAD
```

위 명령은 HEAD에서 변경한 내용들을 되돌리는 새 커밋을 만들고 그에 대한 커밋 메시지를 편집하도록 물어볼 것이다.

또한 더 이전의 변경 내용, 예를 들면 최근의 커밋 바로 전의 변경 내용을 되돌리는 것도 가능하다:

```
CODE $ git revert HEAD^
```

이 경우 git는 해당 변경 내용을 되돌리면서 그 이후에 변경된 내용들은 그대로 남겨두려고 시도한다. 만약 이후의 변경 내용이 되돌린 변경 내용 중 일부를 포함한다면, 머지 (충돌) 해결하기의 경우과 같이 직접 충돌을 해결하도록 요청할 것이다.

변경 내역을 수정하여 실수 바로잡기#

만약 문제가 있는 커밋이 가장 최신의 커밋이고, 이 커밋을 외부에 아직 공개하지 않았다면 'git reset' 명령을 이용하여 커밋을 제거할 수 있다.

아니면, 새 커밋을 만드는 것처럼 작업 디렉토리를 변경하고 인덱스를 갱신한 후에 다음을 실행한다

```
CODE $ git commit --amend
```

위 명령은 먼저 이전 커밋 메시지를 변경하도록 하고, 이전 커밋을 변경된 내용을 포함하는 새 커밋으로 바꿀 것이다.

다시 말하지만, 이미 다른 브랜치에 머지되었을 수도 있는 커밋에 이러한 작업을 수행하서는 안 된다. 이 경우에는 **git-revert(1)** 명령을 이용하자.

커밋들은 변경 내역 상의 더 이전 상태로 되돌리는 것도 가능하다. 하지만 이러한 복잡한 주제는 다른 장에서 설명하도록 남겨 둔다.

이전 버전의 파일을 체크아웃하기#

이전의 실수를 되돌리는 과정에서, **git-checkout(1)** 명령을 이용하여 특정 파일의 이전 버전을 체크아웃하는 것이 유용하다는 것을 알게 될 수 있다. 우리는 지금껏 'git checkout' 명령을 브랜치를 전환하기 위해 사용했었지만, 경로 이름이 주어진 경우에는 다른 동작을 수행한다:

```
CODE $ git checkout HEAD^ path/to/file
```

위 명령은 path/to/file의 내용을 HEAD^ 커밋에 있는 것으로 바꾸고, 그에 따라 인덱스도 갱신한다. 이 명령은 브랜치를 전환하지 않는다.

만약 작업 디렉토리의 변경없이, 단지 이전 버전의 파일을 보고 싶을 뿐이라면 **git-show(1)** 명령을 이용할 수 있다:

```
CODE $ git show HEAD^:path/to/file
```

위 명령은 해당 파일의 특정 버전을 보여줄 것이다.

작업 중인 내용을 임시로 보관해 두기#

복잡한 내용을 수정하고 있는 도중에, 지금 작업 중인 내용과 관련은 없지만 단순하고 명확한 버그를 발견했다고 하자. 여러분은 작업을 계속 진행하기 전에 이 버그를 먼저 수정하고 싶다. **git-stash(1)** 명령을 이용하여 현재 작업 상태를 저장하고, 버그를 수정한 후에 (혹은 다른 브랜치에서 이를 수정하고 다시 원래로 돌아온 후에) 작업 상태를 다시 복구할 수 있다.

```
CODE $ git stash save "work in progress for foo feature"
```

위 명령은 변경 내용을 'stash' (은닉처)에 저장하고, 작업 트리와 인덱스를 현재 브랜치의 끝과 동일하게 초기화할 것이다. 그러면 보통 때처럼 문제점을 수정할 수 있다.

```
CODE ... 수정 및 테스트...
     $ git commit -a -m "blorpl: typofix"
```

그 후에는 'git stash apply' 명령을 이용하여 이전 작업 내용으로 돌아갈 수 있다.

```
CODE $ git stash apply
```

좋은 성능 보장하기#

거대한 저장소에서 git는 변경 내역 정보가 디스크 혹은 메모리 상의 너무 많은 공간을 차지하지 않도록 하기 위해 압축을 수행한다.

이 압축 과정은 자동으로 수행되지는 않는다. 따라서 여러분은 때때로 **git-gc(1)** 명령을 실행해야 한다.

```
$ git gc
```

위 명령은 아카이브를 다시 압축한다. 이 작업은 아주 많은 시간이 걸릴 수도 있으므로 다른 작업을 하지 않을 때 'git gc' 명령을 실행하는 것이 좋을 것이다.

신뢰성 보장하기#

저장소가 망가졌는지 검사하기#

git-fsck(1) 명령은 저장소 상에서 몇 가지 일관성 검사를 수행한 후 문제가 발생하면 보고한다. 이 과정은 어느 정도 시간이 걸린다. 가장 많이 발생하는 경고는 "댕글링" (dangling) 객체에 대한 것이다:

```
$ git fsck
    dangling commit 7281251ddd2a61e38657c827739c57015671a6b3
    dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63
    dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5
    dangling blob 218761f9d90712d37a9c5e36f406f92202db07eb
    dangling commit bf093535a34a4d35731aa2bd90fe6b176302f14f
    dangling commit 8e4bec7f2ddaa268bef999853c2575545210 0f8e
    dangling tree d50bb86186bf27b681d25af89d3b5b68382e4085
    dangling tree b24c2473f1fd3d91352a624795be026d64c8841f
    ...
```

댕글링 객체들은 문제가 되지는 않는다. 가장 최악의 상황으로는 추가적으로 약간의 디스크 공간을 소비한다는 것 정도이다. 이러한 댕글링 객체들은 때로 잃어버린 작업 내용을 복구하기 위한 최후의 수단으로 사용되기도 한다. 자세한 내용은 "댕글링 객체" 부분을 살펴보기 바란다.

잃어버린 작업 내용 복구하기#

Reflogs#

여러분이 'git reset --hard' 명령을 이용하여 브랜치를 수정했다고 하자. 그런데 알고보니 이 브랜치가 해당 변경 내역에 대한 유일한 참조였다.

다행스럽게도 git는 각 브랜치에 대한 이전의 모든 값들을 기록하는 "reflog"라는 로그를 유지한다. 따라서 이 경우 다음과 같은 명령을 이용하여 이전의 변경 내역에 여전히 접근할 수 있다.

```
$ git log master@{1}
```

위 명령은 이전 버전의 "master" 브랜치 헤드에서 접근할 수 있는 커밋의 목록을 보여준다. 이 문법은 단지 git log 명령 뿐 아니라 커밋을 인자로 받는 모든 git 명령에서 사용될 수 있다. 다음은 이에 대한 예제이다:

```
$ git show master@{2}          # 2번의 변경 전의 브랜치가 가리키던 내용
$ git show master@{3}          # 3번의 변경 전의 브랜치가 가리키던 내용
```

```
CODE $ gitk master@{yesterday}       # 어제 브랜치가 가리키던 내용
     $ gitk master@{"1 week ago"}    # 지난 주에 브랜치가 가리키던 내용
     $ git log --walk-reflogs master # master 브랜치의 reflog 항목들을 표시
```

HEAD에 대해서는 별도의 reflog가 유지된다. 따라서

```
CODE $ git show HEAD@{"1 week ago"}
```

위 명령은 현재 브랜치가 일주일 전에 가리키던 내용이 아니라, HEAD가 일주일 전에 가리키던 내용을 보여줄 것이다. 이것은 여러분이 체크아웃했던 내역들을 볼 수 있게 해 준다.

reflog는, 해당 커밋들이 정리될 (prune) 수 있는 상태가 된 후에, 기본적으로 30일 간 저장된다. 이러한 정리 작업을 조정하는 방법은 **`git-reflog(1)`** 혹은 **`git-log(1)`** man 페이지를 살펴보고, 자세한 내용은 **`git-rev-parse(1)`** man 페이지의 "리비전 지정하기" 부분을 살펴보기 바란다.

reflog 변경 내역은 일반 git 변경 내역과는 매우 다르다는 것을 알아두도록 하자. 일반 변경 내역은 같은 프로젝트를 수행하는 모든 저장소에서 공유하지만, reflog 변경 내역은 공유되지 않는다. 이것은 오직 로컬 저장소 내의 브랜치들이 시간에 따라 어떻게 변경되어 왔는지만을 보여준다.

댕글링 객체 살펴보기#

어떤 상황에서는 reflog도 도움이 되지 않을 수가 있다. 예를 들어 여러분이 어떤 디렉토리를 지웠다고 가정해 보자. 나중에 여러분은 이 브랜치에 포함된 변경 내역이 필요하다는 것을 알았다. reflog도 이미 삭제되었다. 하지만 아직 저장소에서 이를 정리(prune)하지 않았다면, 아직은 'git-fsck' 명령이 보여주는 댕글링 객체 내의 잃어버린 커밋들을 찾을 수 있다. 자세한 내용은 "댕글링 객체" 부분을 살펴보기 바란다.

```
CODE $ git fsck
     dangling commit 7281251ddd2a61e38657c827739c57015671a6b3
     dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63
     dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5
     ...
```

예를 들면 다음과 같이 댕글링 객체 중의 하나를 살펴볼 수 있다

```
CODE $ gitk 7281251ddd --not --all
```

위 명령은 다음과 같은 주문대로 동작한다: 댕글링 객체(들)로 설명하는 커밋 변경 내역을 보여주되, 다른 모든 브랜치나 태그에서 설명하는 변경 내역들은 보이지 않도록 하고 싶다. 따라서 잃어버린 커밋에서 도달할 수 있는 변경 내역 만을 얻을 수 있다. (그리고 이것은 하나의 커밋에만 해당하는 것이 아니라는 것을 알아두자: 우리는 지금껏 댕글링 객체가 개발 경로의 끝인 경우 만을 살펴보았지만, 길고 복잡한 변경 내역 전체가 제거되었을 수도 있다.)

만약 이 변경 내역을 복구하기로 결정했다면, 이 커밋을 가리키는 새로운 참조를 (예를 들면, 새 브랜치를) 언제든 만들 수 있다:

```
CODE $ git branch recovered-branch 7281251ddd
```

다른 타입의 댕글링 객체 (블롭 혹은 트리)들도 가능하다. 그리고 이러한 댕글링 객체들은 다른 상황에서 발생한다.

개발 과정 공유하기#

git pull 명령으로 업데이트하기#

저장소를 복사해와서 약간의 작업을 한 후에는, 원본 저장소의 업데이트가 있었는지 살펴보고 이를 여러분의 작업 내용에 머지하고 싶을 경우가 있을 것이다.

이미 **git-fetch(1)** 명령을 이용하여 원격 추적 브랜치를 최신 상태로 유지하는 법과 이를 머지하는 법을 살펴보았다. 따라서 원본 저장소의 "master" 브랜치의 변경 사항들을 머지하려면 다음과 같이 할 수 있다:

```
CODE $ git fetch
     $ git merge origin/master
```

하지만 **git-pull(1)** 명령은 이 두 가지를 한 번에 수행한다:

```
CODE $ git pull origin master
```

사실, "master" 브랜치를 체크아웃한 상태이면, "git pull" 명령은 기본적으로 원본 저장소의 HEAD 브랜치를 머지해 온다. 따라서 위의 경우는 때로 다음과 같이 간단하게 수행될 수 있다:

```
CODE $ git pull
```

좀 더 일반적으로 말하면, 원격 브랜치로부터 생성된 브랜치는 기본적으로 해당 원격 브랜치의 내용을 가져올 (pull) 것이다. 이러한 기본값을 제어하는 방법은 **git-config(1)** man 페이지의 branch.<이름>.remote 및 branch.<이름>merge 옵션에 대한 설명과 **git-checkout(1)** 명령의 '--track' 옵션에 대한 논의를 살펴보기 바란다.

입력해야 할 글자를 줄이는 것 말고도, "git pull" 명령은 가져올 브랜치와 저장소를 기록한 기본 커밋 메시지를 작성해 준다.

(하지만 고속 이동 머지의 경우 이러한 커밋 메시지는 생성되지 않을 것이다. 대신 브랜치는 업스트림 브랜치의 최신 커밋을 가리키도록 갱신되어 있을 것이다.)

'git pull' 명령은 (현재 디렉토리를 나타내는) "."을 "원격" 저장소로 지정할 수 있다. 이 경우 현재 저장소의 브랜치와 머지를 수행한다. 따라서

```
CODE $ git pull . branch
     $ git merge branch
```

위의 명령들은 거의 동일하다. 실제로는 위쪽의 방식이 주로 사용된다.

프로젝트에 패치 제출하기#

만약 여러분이 소스를 몇 가지 수정했다면, 이를 제출하기 위한 가장 간단한 방법은 이메일에 패치를 넣어 보내는 것이다.

먼저 **git-format-patch(1)** 명령을 실행한다. 예를 들어

```
CODE $ git format-patch origin
```

위 명령은 현재 디렉토리에 연속된 숫자가 붙은 여러 파일을 만들어 낼 것이다. 각각의 파일은 현재 브랜치에는 존재하지만 origin/HEAD에는 존재하지 않는 패치에 해당한다.

그 후에 이 파일들을 이메일 클라이언트로 가져와서 직접 보낼 수 있다. 하지만 한 번에 보내기에는 너무 많은 작업을 수행했다면, 이 과정을 자동화하기 위해 **`git-send-email(1)`** 스크립트를 이용하는 것이 나을 것이다. 먼저 이러한 패치들을 어떻게 보내는 것이 좋은지, 해당 프로젝트의 메일링리스트에 문의하도록 하자.

패치를 프로젝트로 가져오기#

git는 위와 같은 이메일로 받은 일련의 패치들을 적용하기 위한 git-am(1) 도구도 제공해준다. (am은 "apply mailbox"의 줄임말이다.) 단지 패치를 포함한 모든 메일 메시지들을 차례대로 한 메일함 파일로 저장하고 (여기서는 "patches.mbox" 라고 가정한다) 다음 명령을 실행한다.

```
CODE $ git am -3 patches.mbox
```

git는 각각의 패치들을 순서대로 적용할 것이다. 만약 적용하는 도중 충돌이 발생한다면 작업은 중지될 것이다. "머지 (충돌) 해결하기" 부분에서 설명한 대로 이를 해결할 수 있다. ("-3" 옵션은 git에게 머지를 수행하라고 지정한다. 만약 작업을 중지하고 작업 트리와 인덱스를 원래 상태로 유지하기를 원한다면, 단순히 이 옵션을 빼 버리면 된다.)

충돌을 해결하는 결과로 새로운 커밋을 만드는 대신, 인덱스를 갱신했다면 다음 명령을 수행한다

```
CODE $ git am --resolved
```

그러면 git는 새로운 커밋을 만들고 메일함에 남아있는 나머지 패치들을 다시 적용해 갈 것이다.

최종 결과는 일련의 커밋으로 나타난다. 각각의 커밋은 원래 메일함에 들어있던 각 패치들에 해당하며, 작성자와 커밋 로그 메시지는 각 패치에 포함된 메시지로부터 추출된다.

공개 git 저장소#

프로젝트에 변경 내용을 제출하는 또 다른 방법은 해당 프로젝트의 관리자에게 **`git-pull(1)`** 명령을 이용하여 여러분의 저장소 내의 변경 내역을 가져오라고 말하는 것이다. "git pull 명령으로 업데이트하기" 부분에서 이 방법을 이용하여 "메인" 저장소의 변경 내용을 업데이트하는 방법을 설명하였지만, 이것은 반대의 경우에도 잘 동작한다.

만약 여러분과 관리자가 동일한 머신 상의 계정을 가지고 있다면, 다른 사람의 저장소로부터 변경 내역을 직접 가져올 수 있다. 저장소 URL을 인자로 받는 명령들은 로컬 디렉토리 이름도 인자로 받을 수 있다:

```
CODE $ git clone /path/to/repository
     $ git pull /path/to/other/repository
```

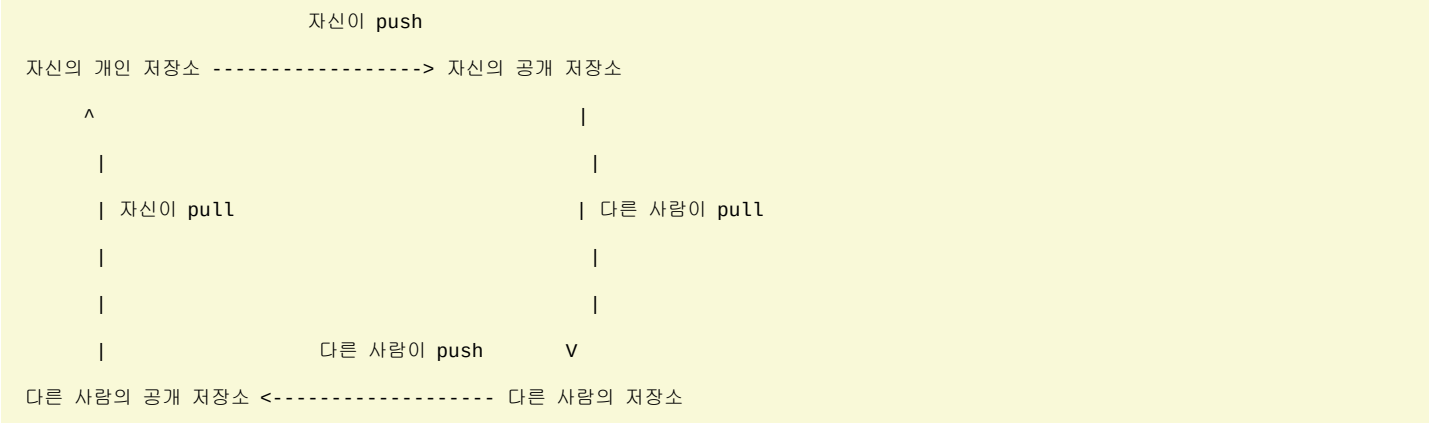
혹은 SSH URL도 가능하다.

```
CODE $ git clone ssh://yourhost/~you/repository
```

적은 개발자로 이루어진 프로젝트나 몇 개의 개인 저장소와 동기화하는 경우에는, 이것만으로도 충분할 것이다.

하지만 이런 일을 수행하는 좀 더 일반적인 방법은 (보통은 별도의 호스트 상에) 별도의 공개된 저장소를 두고 다른 사람들이 여기서 변경 사항을 내려받을 수 있도록 하는 것이다. 이 방법이 보통 더 편리하며, 개인적으로 작업 중인 내용과 외부에 공개된 것을 확실히 구분해 준다.

작업은 매일매일 개인 저장소에서 이루어지지만, 주기적으로 개인 저장소의 내용을 공개 저장소로 보내서 (push) 다른 개발자들이 공개 저장소로부터 작업 내용을 내려받을 수 있도록 해야 할 것이다. 따라서 다른 개발자 한 명이 공개 저장소를 가지고 있는 경우, 변경 사항들은 다음과 같이 이동할 것이다:



아래에서는 이 과정에 대해 설명한다.

공개 저장소 설정하기#

여러분의 개인 저장소가 ~/proj 디렉토리에 있다고 가정하자. 먼저 해당 저장소에 대한 복사본을 만들고, 'git 대몬'에게 이 저장소가 공개 저장소 임을 알려준다:

```
$ git clone --bare ~/proj proj.git
$ touch proj.git/git-daemon-export-ok
```

결과로 생성되는 proj.git 디렉토리는 "순수한" (bare) git 저장소를 포함한다. 이것은 단지 ".git" 디렉토리의 내용이며, 이를 체크아웃한 어떤 파일도 포함하지 않는다.

다음으로 proj.git 디렉토리를 공개 저장소를 운영할 서버로 복사한다. 이 때 scp, rsync 및 다른 익숙한 방식을 사용할 수 있다.

git 프로토콜을 이용해 git 저장소 공개하기#

이것이 권장하는 방법이다.

만약 다른 사람이 서버를 관리하고 있다면, 관리자는 git 저장소가 있어야 할 디렉토리와 그에 따른 git:// URL을 알려주어야 한다. 그렇다면 아래의 "공개 저장소에 변경 내용 올리기" 부분으로 건너뛸 수 있다.

그렇지 않다면 **git-daemon(1)**을 실행해야 한다. git 대몬은 9418 포트를 이용하며, 기본적으로 **git-daemon-export-ok**라는 특수 파일을 가지고 있으며 git 디렉토리로 보이는 모든 디렉토리에 대한 접근을 허용한다. 'git daemon' 명령에 인자로 디렉토리 목록을 넘겨

주면 해당 디렉토리 만을 공개하도록 제한할 것이다.

또한 'git 대몬'을 inetd 서비스로 실행할 수 있다. 자세한 내용은 **git-daemon(1)** man 페이지를 (특히 예제 부분을) 살펴보기 바란다.

http를 이용해 git 저장소 공개하기#

git 프로토콜이 더 나은 성능과 안정성을 제공하지만, 이미 웹 서버가 구동 중인 시스템이라면 http를 이용하여 저장소를 공개하는 것이 더 간단할 것이다.

이를 위해서는 새로 생성한 순수한 (bare) git 저장소를 웹 서버를 통해 공개된 디렉토리에 복사하고, 웹 클라이언트에게 필요한 부가 정보들을 제공할 수 있도록 약간의 작업을 수행하기만 하면 된다:

```
CODE $ mv proj.git /home/you/public_html/proj.git
     $ cd proj.git
     $ git --bare update-server-info
     $ mv hooks/post-update.sample hooks/post-update
```

(마지막 두 줄에 대한 설명은 **git-update-server-info(1)** 과 **githooks(5)** man 페이지를 살펴보기 바란다.)

proj.git의 URL을 알린다. 이제 다른 사람들이 해당 URL로부터 저장소를 복사(clone)하거나 변경 내역을 받아갈(pull) 수 있다. 예를 들어 명령행에서는 다음과 같이 할 수 있다:

```
CODE $ git clone http://yourserver.com/~you/proj.git
```

(WebDAV를 이용하여 http 상에서 변경 내역을 올릴(push) 수 있도록 하는 좀 더 복잡한 설정 방법은 "http 상의 git 서버 설정" 문서를 살펴보기 바란다.)

공개 저장소에 변경 내역 올리기#

위에서 설명한 두 가지 (git 및 http) 방법은 다른 관리자들이 여러분의 최신 작업 사항을 받아갈 수 있도록 해 주지만, 쓰기 접근은 허용하지 않는다는 것을 알아두길 바란다. 여러분은 개인 저장소의 최신 변경 내용을 공개 저장소에 업데이트해 주어야 한다.

이를 위한 가장 간단한 방법은 **git-push(1)** 명령과 ssh를 이용하는 것이다. "master"라는 원격 브랜치를 여러분이 작업한 "master" 브랜치의 최신 내용으로 업데이트하려면 다음을 실행한다.

```
CODE $ git push ssh://yourserver.com/~you/proj.git master:master
```

혹은 단순히 다음과 같이 실행할 수 있다.

```
CODE $ git push ssh://yourserver.com/~you/proj.git master
```

'git fetch' 명령과 같이, 'git push' 명령은 이 작업이 '고속 이동' 머지로 이루어지지 않으면 경고를 보여줄 것이다. 다음 부분에서 이 경우를 처리하는 법에 대해 자세히 설명한다.

일반적으로 "push" 명령의 대상은 순수 저장소이다. 체크아웃한 작업 트리를 가지는 저장소에도 'push' 명령을 수행할 수 있지만, 이것만으로는 작업 트리 자체가 변경되지는 않는다. 이것은 여러분이 'push' 명령을 수행한 브랜치가 현재 체크아웃한 브랜치라면 예상치 못한

결과가 발생할 수 있다!

'git fetch' 명령 때와 같이, 타이핑을 줄이기 위한 설정 옵션을 세팅할 수 있다. 예를 들어 다음과 같이 설정했다면

```
CODE $ cat >>.git/config <<EOF
     [remote "public-repo"]
             url = ssh://yourserver.com/~you/proj.git
     EOF
```

위의 push 명령을 다음과 같이 수행할 수 있다.

```
CODE $ git push public-repo master
```

자세한 내용은 **`git-config(1)`** man 페이지의 remote.<이름>.url, branch.<이름>.remote, remote.<이름>.push 옵션의 설명을 살펴보기 바란다.

push 실패 시의 처리#

만약 push 명령이 원격 브랜치의 고속 이동으로 이어지지 않았다면, 다음과 같은 에러를 내며 실패할 것이다:

```
CODE error: remote 'refs/heads/master' is not an ancestor of
      local  'refs/heads/master'.
      Maybe you are not up-to-date and need to pull first?
     error: failed to push to 'ssh://yourserver.com/~you/proj.git'
```

이것은 다음과 같은 경우에 발생할 수 있다:

- 이미 공개된 커밋을 'git reset --hard' 명령으로 제거한 경우
- 이미 공개된 커밋을 'git commit --amend' 명령으로 수정한 경우 ('변경 내역을 수정하여 실수 바로잡기' 부분에서 설명)
- 이미 공개된 커밋에 대해 'git rebase' 명령을 수행한 경우 ('git rebase 명령으로 여러 패치들을 최신 상태로 유지하기' 부분에서 설명)

브랜치 이름 앞에 '+' 기호를 붙이면 'git push' 명령이 강제로 업데이트를 수행하도록 지정할 수 있다:

```
CODE $ git push ssh://yourserver.com/~you/proj.git +master
```

일반적으로 공개 저장소 내의 브랜치 헤드가 변경될 때마다, 이전에 가리키던 커밋의 자손을 가리키도록 변경된다. 이 상황에서 강제로 push 명령을 수행하면, 이러한 관례를 깨뜨리는 것이다. ("변경 내역을 수정함에 따른 문제점" 부분을 살펴보기 바란다.)

그럼에도 불구하고, 이것은 작업 중인 패치들을 간단히 공개하기를 원하는 사람들이 주로 사용하는 방식이며, 여러분이 다른 개발자들에게 브랜치를 이런 식으로 관리할 것이라고 미리 얘기해 두었다면 용납될 만한 부분이다.

또한 다른 사람이 동일한 저장소에 push 명령을 수행할 수 있는 권한을 가지고 있는 경우, 이 방식은 실패할 수 있다. 이 경우 이를 해결하기 위한 올바른 방법은 먼저 pull 명령이나 fetch 및 rebase 명령을 수행하여 여러분의 작업 내용을 갱신하고나서 변경 내용을 다시 올리는(push) 것이다. 이후의 내용은 다음 절과 **`gitcvs-migration(7)`** man 페이지를 살펴보기 바란다.

공유 저장소 설정하기#

다른 개발자들과 작업을 공유하기 위한 또 다른 방법은, CVS 등에서 주로 사용되는 것과 비슷한 모델로, 특수한 권한을 가진 몇몇 개발자들이 모두 하나의 공유 저장소에 pull/push 명령을 수행하는 것이다. 이를 위한 설정 방법은 **`gitcvs-migration(7)`** man 페이지를 살펴보기 바란다.

하지만, git의 공유 저장소에 대한 지원이 아무 문제가 없다 할지라도, 이러한 방식의 운영은 일반적으로 추천하지 않는다. 왜냐하면 단지 git가 지원하는 (패치를 보내고 공개 저장소에서 내려받는) 협업 모델이 하나의 공유 저장소를 사용하는 것에 비해 매우 많은 장점을 가지기 때문이다.

- git가 제공하는 빠른 패치 관리 기능은, 매우 활발히 진행되는 프로젝트에서도 한 명의 관리자가 간단히 변경 사항들을 처리할 수 있도록 해 준다. 또한 이를 넘어서는 상황이 발생하는 경우라도, 'git pull' 명령은 다른 관리자에게 이러한 작업을 넘겨줄 수 있는 손쉬운 방법을 제공하며 그 상황에서도 새로운 변경 사항들에 대한 추가적인 검토를 수행할 수 있다.
- 모든 개발자들의 저장소가 프로젝트의 변경 내역에 대한 완전한 복사본을 가지므로 특별한 저장소가 존재하지 않으며, 협의에 의한 경우 혹은 기존 관리자가 활동이 없거나 더 이상 함께 작업하기 힘들어진 경우에 다른 개발자가 프로젝트의 관리 작업을 맡는 일이 아주 간단해진다.
- 중심적인 "커미터"(committer) 그룹이 없어지므로, 누가 이 그룹에 포함되어야 하는지 아닌지에 대한 공식적인 결정을 할 일이 적어진다.

저장소를 웹으로 살펴볼 수 있게 공개하기#

gitweb이라는 cgi 스크립트는 다른 사람들이 git를 설치하지 않고도 여러분의 저장소 내의 파일들과 변경 내역들을 살펴볼 수 있는 손쉬운 방법을 제공한다. gitweb을 설정하는 방법은 git 소스 트리 내의 gitweb/INSTALL 파일을 살펴보기 바란다.

예제#

리눅스 하위시스템 관리자를 위한 주제별 브랜치 관리하기#

이 부분은 Tony Luck이 리눅스 커널의 IA64 아키텍처 관리자로서 git를 사용하는 방법을 설명한다.

그는 2 개의 공개 브랜치를 사용한다:

- 패치가 최초 적용되는 "test" 트리. 이 트리는 다른 개발 과정에 통합되어 테스트를 받을 수 있도록 하기 위한 것이다. 이 트리는 Andrew가 원할 때마다 -mm 트리로 통합할 수 있도록 제공된다.
- 테스트를 통과한 패치들이 최종 검사를 위해 옮겨지는 "release" 트리. 이 트리는 Linus에게 보내서 업스트림에 반영되도록 하기 위한 것이다. (Linus에게 "please pull" 요청을 보낸다.)

그는 또한 패치들의 논리적인 그룹을 포함하는 몇 가지 임시 브랜치 ("주제별 브랜치")를 사용한다.

이를 설정하기 위해서, 먼저 Linus의 공개 트리를 복사하여 작업 트리를 만든다:

```
CODE $ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git work
     $ cd work
```

Linus의 트리는 origin/master라는 이름의 원격 브랜치로 저장되며, **`git-fetch(1)`** 명령을 통해 업데이트할 수 있다. 다른 공개 트리를 추적하려면 **`git-remote(1)`** 명령을 이용하여 "원격" 브랜치를 설정하고 **`git-fetch(1)`** 명령으로 이를 최신 상태로 유지하면 된다. 1

장 저장소와 브랜치 부분을 살펴보도록 하자.

이제 작업을 진행할 새 브랜치들을 만든다. 이 브랜치들은 origin/master 브랜치의 최신 상태에서 시작되며, 기본적으로 (**git-branch(1)** 명령의 --track 옵션을 이용하여) Linus의 변경 사항들을 머지하도록 설정되어야 한다.

```
CODE $ git branch --track test origin/master
     $ git branch --track release origin/master
```

이들은 **git-pull(1)** 명령을 이용하면 쉽게 최신 상태로 만들 수 있다.

```
CODE $ git checkout test && git pull
     $ git checkout release && git pull
```

중요한 사항! 만약 이 브랜치에 로컬 변경 사항들이 있는 경우에는, 머지 과정에서 변경 내역 상에 커밋 객체를 만들 것이다. (로컬 변경 사항이 없는 경우에는 단순히 "고속 이동" 머지를 수행할 것이다.) 많은 사람들은 리눅스 변경 내역 상에 이렇게 만들어진 "노이즈"가 섞이는 것을 싫어하기 때문에, "release" 브랜치에서 이러한 작업을 자주 수행하는 것은 피해야 한다. 여러분이 Linus에게 "release" 브랜치에서 변경 내역을 내려받도록(pull) 요청하면, 이러한 노이즈 커밋들이 영구적으로 변경 내역 상에 남아 있게 되기 때문이다.

몇 가지 설정 변수들은 (**git-config(1)** man 페이지를 보자) 두 개의 브랜치 모두에서 여러분의 공개 트리로 변경 내역을 올리는(push) 것을 간편하게 도와줄 수 있다. ("공개 저장소 설정하기" 부분을 살펴보기 바란다.)

```
CODE $ cat >> .git/config <<EOF
     [remote "mytree"]
             url =  master.kernel.org:/pub/scm/linux/kernel/git/aegl/linux-2.6.git
             push = release
             push = test
     EOF
```

이제 **git-push(1)** 명령을 이용하여 test 브랜치와 release 브랜치 모두를 올릴 수 있다:

```
CODE $ git push mytree
```

혹은 다음과 같이 test 나 release 브랜치 중의 하나 만 올릴 수 있다:

```
CODE $ git push mytree test
```

혹은

```
CODE $ git push mytree release
```

이제 커뮤니티에서 받은 몇 가지 패치들을 적용해 보자. 이 패치들 (혹은 관련 패치 모음들)을 담아둘 브랜치를 위한 짧은 이름을 하나 생각해서, Linus의 브랜치의 최신 상태로부터 새로운 브랜치를 만든다:

```
CODE $ git checkout -b speed-up-spinlocks origin
```

이제 패치들을 적용해서, 테스트를 수행하고, 변경 내용들을 커밋한다. 만약 패치가 여러 개로 나누어져 있다면 이들을 각각 별도의 커밋으로 나누어 적용해야 한다.

```
CODE $ ... 패치 ... 테스트  ... 커밋 [ ... 패치 ... 테스트 ... 커밋 ]*
```

이 변경 내용에 대해 만족한 상태가 되면, 이것을 "test" 브랜치로 가져와서(pull) 공개할 준비를 한다.

```
CODE $ git checkout test && git pull . speed-up-spinlocks
```

이 과정에서는 거의 충돌이 발생하지 않는다. 하지만 이 과정에서 많은 시간을 소비했고, 업스트림에서 최신 버전을 내려받은 경우에는 충돌이 발생할 수도 있다.

충분한 시간이 지나고 테스트가 완료된 어느 시점에는, 업스트림에 올릴 수 있도록 동일한 브랜치를 "release" 트리로 가져올 수 있다. 이 부분이 바로 각 패치들을 (혹은 일련의 패치 모음을) 별도의 브랜치로 유지하는 것에 대한 가치를 느낄 수 있는 부분이다. 이것은 패치가 "release" 트리에 어떤 순서로도 이동될 수 있다는 것을 뜻한다.

```
CODE $ git checkout release && git pull . speed-up-spinlocks
```

조금 지나면 수 많은 브랜치들이 생겨나게 될 것이다. 비록 각각에 대해 적당한 이름을 붙였다고해도, 이들 각각이 무엇을 위한 것인지 혹은 현재 상태가 어느 정도인지 잊어버릴 수 있다. 특정 브랜치에 어떤 변경 사항이 있었는지 기억하려면 다음을 실행한다:

```
CODE $ git log linux..branchname | git shortlog
```

해당 브랜치가 test 혹은 release 브랜치에 머지되었는지 알아보려면 다음을 실행한다:

```
CODE $ git log test..branchname
```
혹은
```
CODE $ git log release..branchname
```

(만약 해당 브랜치가 아직 머지되지 않았다면, 로그 메시지들이 출력될 것이다. 만약 머지되었다면 아무런 메시지도 출력되지 않을 것이다.)

패치가 거대한 순환 (test 브랜치에서 release 브랜치로 이동되고, Linus가 내려받은 후 최종적으로 로컬의 "origin/master" 브랜치로 돌아오는 것)을 마치고 나면, 해당 변경 사항을 위한 브랜치는 더 이상 필요치 않다. 이를 알아보려면 다음을 실행한다:

```
CODE $ git log origin..branchname
```

위 명령이 아무런 메시지를 출력하지 않으면 해당 브랜치를 삭제할 수 있다:

```
CODE $ git branch -d branchname
```

어떤 변경 사항들은 너무 사소해서, 별도의 브랜치를 만든 후 test와 release 브랜치로 머지할 필요가 없을 수도 있다. 이런 사항들은 직접 "release" 브랜치에 적용하고, 그 후에 "test" 브랜치로 머지한다.

Linus에게 보낼 "please pull" 요청에 포함될 차이점 통계(diffstat)와 짧은 로그 요약을 만들려면 다음을 실행한다:

```
CODE $ git diff --stat origin..release
```
그리고
```
CODE $ git log -p origin..release | git shortlog
```

아래는 지금까지 설명한 것들을 실행하는 스크립트들이다.

CODE ==== 업데이트 스크립트 ====

```
# 로컬 GIT 트리 내의 브랜치들을 업데이트한다. 만약 업데이트할
# 브랜치가 origin이면, kernel.org를 내려받는다. 그렇지않으면
# origin/master 브랜치를 test|release 브랜치로 머지한다.

case "$1" in
test|release)
        git checkout $1 && git pull . origin
        ;;
origin)
        before=$(git rev-parse refs/remotes/origin/master)
        git fetch origin
        after=$(git rev-parse refs/remotes/origin/master)
        if [ $before != $after ]
        then
                git log $before..$after | git shortlog
        fi
        ;;
*)
        echo "Usage: $0 origin|test|release" 1>&2
        exit 1
        ;;
esac
```

CODE ==== 머지 스크립트 ====

```
# 주어진 브랜치를 test 혹은 release 브랜치로 머지한다.

pname=$0

usage()
{
        echo "Usage: $pname branch test|release" 1>&2
        exit 1
}

git show-ref -q --verify -- refs/heads/"$1" || {
        echo "Can't see branch <$1>" 1>&2
        usage
}

case "$2" in
test|release)
        if [ $(git log $2..$1 | wc -c) -eq 0 ]
```

```
CODE          then
                      echo $1 already merged into $2 1>&2
                      exit 1
              fi
              git checkout $2 && git pull . $1
              ;;
      *)
              usage
              ;;
      esac
```

```
CODE ==== 상태 스크립트 ====
      # 로컬의 ia64 GIT 트리의 상태를 보여준다.

      gb=$(tput setab 2)
      rb=$(tput setab 1)
      restore=$(tput setab 9)

      if [ `git rev-list test..release | wc -c` -gt 0 ]
      then
              echo $rb Warning: commits in release that are not in test $restore
              git log test..release
      fi

      for branch in `git show-ref --heads | sed 's|^.*/||'`
      do
              if [ $branch = test -o $branch = release ]
              then
                      continue
              fi

              echo -n $gb ======= $branch ====== $restore " "
              status=
              for ref in test release origin/master
              do
                      if [ `git rev-list $ref..$branch | wc -c` -gt 0 ]
                      then
                              status=$status${ref:0:1}
                      fi
              done
              case $status in
              trl)
                      echo $rb Need to pull into test $restore
                      ;;
              rl)
                      echo "In test"
```

```
CODE                    ;;
            |)
                    echo "Waiting for linus"
                    ;;
            "")
                    echo $rb All done $restore
                    ;;
            *)
                    echo $rb "<$status>" $restore
                    ;;
            esac
            git log origin/master..$branch | git shortlog
    done
```

변경 내역 수정하기와 패치 묶음 관리하기#

일반적으로 커밋은 프로젝트에 추가될 뿐이고, 사라지거나 수정되지 않는다. git는 이러한 가정을 두고 설계되었으므로, 이를 위반하게되면 (예를 들어) git의 머지 동작을 오동작하게 만들 수 있다.

하지만, 이러한 가정을 위반하는 것이 유용한 상황이 존재한다.

완벽한 패치 묶음 만들기#

여러분이 거대한 프로젝트의 공헌자 중의 하나라고 가정해 보자. 여러분은 복잡한 기능을 추가하고 싶고, 다른 개발자들이 여러분이 작업한 변경 내용을 보고, 올바로 동작하는 지 확인하고, 왜 이러한 작업을 하게 됐는지 이해하기 쉽도록 해 주고 싶다.

만약 모든 변경 사항을 하나의 패치(혹은 커밋)로 만들어 준다면, 다른 사람들이 한 번에 살펴보기가 매우 힘들것이다.

만약 여러분의 작업에 대한 전체 변경 내역을 준다면, 실수했던 부분과 이를 수정한 부분은 물론 잘못된 방향으로 가서 막힌 부분들에 대한 정보까지도 모두 공개되어, 다른 개발자들을 질리게 만들 것이다.

따라서 이상적인 방법은 보통 다음과 같은 패치 묶음의 형태로 만드는 것이다:

1. 각 패치들은 순서대로 적용될 수 있다.
2. 각 패치들은 하나의 논리적인 변경 사항과, 이 변경 사항을 설명하는 메시지를 포함한다.
3. 어떤 패치에도 퇴보가 없어야 한다. 패치 묶음 중 어느 단계까지를 적용하더라도, 프로젝트는 정상적으로 컴파일되어 동작해야 하며, 이전에 없었던 버그를 포함해서는 안된다.
4. 패치 묶음을 모두 적용한 최종 결과는, 여러분의 개발 과정(아마도 더 지저분할 것이다!)에서 얻은 결과와 동일해야 한다.

이제 이러한 작업을 도와주는 몇 가지 도구들에 대해 알아보면서, 이들을 사용하는 방법과 변경 내역을 수정했을 때 발생할 수 있는 문제점들에 대해서 설명할 것이다.

git rebase를 이용하여 패치 묶음을 최신 상태로 유지하기#

"origin"이라는 원격 추적 브랜치에서 "mywork"라는 브랜치를 만들고, 몇 가지 커밋을 생성했다고 가정해 보자:

```
CODE $ git checkout -b mywork origin
     $ vi file.txt
     $ git commit
     $ vi otherfile.txt
     $ git commit
     ...
```

mywork 브랜치에는 아무런 머지도 이루어지지 않았기 때문에, 이것은 단지 "origin" 브랜치로부터의 단순 선형 연결이다:

```
o--o--o <-- origin
        \
         o--o--o <-- mywork
```

이제 업스트림 프로젝트에서 다른 작업이 이루어졌고, "origin" 브랜치도 변경되었다:

```
o--o--O--o--o--o <-- origin
        \
         a--b--c <-- mywork
```

이 시점에서, 여러분의 변경 사항을 다시 머지하기 위해 "pull" 명령을 수행할 수 있다. 이 결과로 아래와 같이 새로운 머지 커밋이 생길 것이다:

```
o--o--O--o--o--o <-- origin
        \        \
         a--b--c--m <-- mywork
```

하지만, 만약 mywork 브랜치 내의 변경 내역을 머지 커밋 없이 단순히 연결된 커밋 상태로 유지하고 싶다면, **`git-rebase(1)`** 명령을 이용해 볼 수 있다:

```
CODE $ git checkout mywork
     $ git rebase origin
```

위 명령은 mywork 브랜치의 커밋들을 삭제하여, 패치의 형태로 (".git/rebase-apply"라는 디렉토리 내에) 임시 저장해두고, mywork 브랜치를 origin 브랜치의 최신 버전을 가리키도록 갱신한 후, 저장된 패치들을 새 mywork 브랜치에 적용할 것이다. 결과는 다음과 같은 형태가 될 것이다:

```
o--o--O--o--o--o <-- origin
                \
                 a'--b'--c' <-- mywork
```

이 과정에서 충돌이 발생할 수도 있다. 충돌이 발생한 경우에는, 동작을 멈추고 충돌을 수정할 수 있게 해 준다. 충돌이 수정되고 나면, 'git add' 명령을 이용하여 해당 내용에 대한 인덱스를 갱신한 다음, 'git commit' 명령을 실행하는 대신 다음을 실행한다:

```
CODE $ git rebase --continue
```

그러면 git는 나머지 패치들을 계속해서 적용할 것이다.

어떤 단계에서도 '--abort' 옵션을 이용하면, 이 과정을 중지하고 rebase를 실행하기 전의 상태로 mywork 브랜치를 되돌릴 수 있다:

```
CODE $ git rebase --abort
```

하나의 커밋 수정하기#

"변경 내역을 수정하여 실수 바로잡기" 부분에서 가장 최신 커밋을 수정하는 방법을 살펴보았다.

```
CODE $ git commit --amend
```

위 명령은 이전 커밋 메시지를 변경할 수 있는 기회를 준 후에, 이전 커밋을 변경 사항을 포함하는 새 커밋으로 바꿀 것이다.

이 명령과 **`git-rebase(1)`** 명령을 함께 사용하면 변경 이력 상의 이전 커밋을 수정하고, 그 이후의 변경 사항들을 다시 만들 수 있다. 먼저 문제가 있는 커밋에 다음과 같이 태그를 붙인다:

```
CODE $ git tag bad mywork~5
```

(gitk 혹은 'git log' 명령을 이용하면 해당 커밋을 찾기가 편리할 것이다.)

그리고 해당 커밋을 체크아웃하고, 문제점을 수정한 뒤, 그 위에 나머지 커밋들에 대해 'rebase' 명령을 수행한다. (알아두어야 할 점은, 해당 커밋을 임시 브랜치 상에 체크아웃할 수도 있었지만, 대신 "분리된 헤드"를 사용하였다는 것이다.)

```
CODE $ git checkout bad
     $ # 문제를 수정하고 인덱스를 갱신한다
     $ git commit --amend
     $ git rebase --onto HEAD bad mywork
```

이 과정이 완료되면, mywork 브랜치가 체크아웃된 상태로 남아 있고, mywork 브랜치의 최근 패치들은 수정된 커밋 상에 다시 적용되어 있을 것이다. 이제 불필요한 것을 정리한다:

```
CODE $ git tag -d bad
```

git 변경 이력의 불변성은, 실제로 기존 커밋을 "수정"하지 않는다는 것을 뜻한다. 대신 이전의 커밋을 새로운 객체 이름을 가지는 새 커밋으로 바꾼 것이다.

번역 용어표#

일반적인 용어들은 번역 용어집 참조할 것!

- blob : 블롭
- check out : 체크아웃(하다)
- clone : (저장소를) 복사하다
- commit : 커밋(하다)
- conflict : 충돌
- dangling object : 댕글링 객체

- diff : 차이점, (명령) diff
- head : 헤드
- history : 변경 내역
- local : 로컬
- merge : 머지(하다), 통합하다
- patch series : 패치 묶음. 일련의 패치
- prune : (댕글링 객체 등을) 정리(하다)
- pull : (저장소에서 변경 사항을) 내려받다, 가져오다
- push : (저장소로 변경 사항을) 올리다
- reachability : 도달 가능성
- regression : 퇴보?
- remote-tracking branch : 원격 추적 브랜치
- topic branch : 주제별 브랜치
- working tree : 작업 트리