**Git 사용자 설명서**<http://namhyung.springnote.com/pages/3132772>

**목차**

1. [저장소와 브랜치](http://namhyung.springnote.com/pages/3132772#toc_0)
   1. [git 저장소 가져오기](http://namhyung.springnote.com/pages/3132772#toc_1)
   2. [다른 버전의 프로젝트 체크아웃 하기](http://namhyung.springnote.com/pages/3132772#toc_2)
   3. [변경 이력 이해하기: 커밋](http://namhyung.springnote.com/pages/3132772#toc_3)
      1. [변경 이력 이해하기: 커밋, 부모, 도달 가능성](http://namhyung.springnote.com/pages/3132772#toc_4)
      2. [변경 이력 이해하기: 변경 이력 다이어그램](http://namhyung.springnote.com/pages/3132772#toc_5)
      3. [변경 이력 이해하기: 브랜치란 무엇인가?](http://namhyung.springnote.com/pages/3132772#toc_6)
   4. [브랜치 다루기](http://namhyung.springnote.com/pages/3132772#toc_7)
   5. [새 브랜치를 만들지 않고 이전 버전 살펴보기](http://namhyung.springnote.com/pages/3132772#toc_8)
   6. [원격 저장소의 브랜치 살펴보기](http://namhyung.springnote.com/pages/3132772#toc_9)
   7. [브랜치, 태그 및 다른 참조에 대한 이름 붙이기](http://namhyung.springnote.com/pages/3132772#toc_10)
   8. [git-fetch를 이용하여 저장소 업데이트하기](http://namhyung.springnote.com/pages/3132772#toc_11)
   9. [다른 저장소에서 브랜치 가져오기](http://namhyung.springnote.com/pages/3132772#toc_12)
2. [Git 변경 이력 조사하기](http://namhyung.springnote.com/pages/3132772#toc_13)
   1. [regression을 찾기 위해 bisect 이용하기](http://namhyung.springnote.com/pages/3132772#toc_14)
   2. [커밋 이름 붙이기](http://namhyung.springnote.com/pages/3132772#toc_15)
   3. [태그 만들기](http://namhyung.springnote.com/pages/3132772#toc_16)
   4. [버전 살펴보기](http://namhyung.springnote.com/pages/3132772#toc_17)
   5. [차이점 생성하기](http://namhyung.springnote.com/pages/3132772#toc_18)
   6. [예전 버전의 파일 보기](http://namhyung.springnote.com/pages/3132772#toc_19)
   7. [예제](http://namhyung.springnote.com/pages/3132772#toc_20)
      1. [특정 브랜치 상의 커밋 개수 세기](http://namhyung.springnote.com/pages/3132772#toc_21)
      2. [두 브랜치가 동일한 지점을 가리키는지 검사하기](http://namhyung.springnote.com/pages/3132772#toc_22)
      3. [특정 커밋을 포함하는 것 중에서 제일 먼저 태그가 붙은 버전 찾기](http://namhyung.springnote.com/pages/3132772#toc_23)
      4. [주어진 브랜치에만 존재하는 커밋 보기](http://namhyung.springnote.com/pages/3132772#toc_24)
      5. [소프트웨어 배포를 위한 변경 로그 및 tarball 만들기](http://namhyung.springnote.com/pages/3132772#toc_25)
      6. [파일의 특정 내용을 참조하는 커밋 찾기](http://namhyung.springnote.com/pages/3132772#toc_26)
3. [Git를 이용하여 개발하기](http://namhyung.springnote.com/pages/3132772#toc_27)
   1. [git에게 이름 말해주기](http://namhyung.springnote.com/pages/3132772#toc_28)
   2. [새 저장소 만들기](http://namhyung.springnote.com/pages/3132772#toc_29)
   3. [커밋을 만드는 방법](http://namhyung.springnote.com/pages/3132772#toc_30)
   4. [좋은 커밋 메시지 작성하기](http://namhyung.springnote.com/pages/3132772#toc_31)
   5. [파일 무시하기](http://namhyung.springnote.com/pages/3132772#toc_32)
   6. [머지하는 방법](http://namhyung.springnote.com/pages/3132772#toc_33)
   7. [머지 (충돌) 해결하기](http://namhyung.springnote.com/pages/3132772#toc_34)
      1. [머지 중에 충돌을 해결하기 위해 필요한 정보 얻기](http://namhyung.springnote.com/pages/3132772#toc_35)
   8. [머지 되돌리기](http://namhyung.springnote.com/pages/3132772#toc_36)
   9. [고속 이동 머지](http://namhyung.springnote.com/pages/3132772#toc_37)
   10. [실수 바로잡기](http://namhyung.springnote.com/pages/3132772#toc_38)
       1. [새로운 커밋을 만들어 실수 바로잡기](http://namhyung.springnote.com/pages/3132772#toc_39)
       2. [변경 이력을 수정하여 실수 바로잡기](http://namhyung.springnote.com/pages/3132772#toc_40)
       3. [이전 버전의 파일을 체크아웃하기](http://namhyung.springnote.com/pages/3132772#toc_41)
       4. [작업 중인 내용을 임시로 보관해 두기](http://namhyung.springnote.com/pages/3132772#toc_42)
   11. [좋은 성능 보장하기](http://namhyung.springnote.com/pages/3132772#toc_43)
   12. [신뢰성 보장하기](http://namhyung.springnote.com/pages/3132772#toc_44)
       1. [저장소가 망가졌는지 검사하기](http://namhyung.springnote.com/pages/3132772#toc_45)
       2. [잃어버린 작업 내용 복구하기](http://namhyung.springnote.com/pages/3132772#toc_46)
          1. [Reflogs](http://namhyung.springnote.com/pages/3132772#toc_47)
          2. [댕글링 객체 살펴보기](http://namhyung.springnote.com/pages/3132772#toc_48)
4. [개발 과정 공유하기](http://namhyung.springnote.com/pages/3132772#toc_49)
   1. [git pull 명령으로 업데이트하기](http://namhyung.springnote.com/pages/3132772#toc_50)
   2. [프로젝트에 패치 제출하기](http://namhyung.springnote.com/pages/3132772#toc_51)
   3. [패치를 프로젝트로 가져오기](http://namhyung.springnote.com/pages/3132772#toc_52)
   4. [공개 git 저장소](http://namhyung.springnote.com/pages/3132772#toc_53)
      1. [공개 저장소 설정하기](http://namhyung.springnote.com/pages/3132772#toc_54)
      2. [git 프로토콜을 이용해 git 저장소 공개하기](http://namhyung.springnote.com/pages/3132772#toc_55)
      3. [http를 이용해 git 저장소 공개하기](http://namhyung.springnote.com/pages/3132772#toc_56)
      4. [공개 저장소에 변경 이력 올리기](http://namhyung.springnote.com/pages/3132772#toc_57)
      5. [push 실패 시의 처리](http://namhyung.springnote.com/pages/3132772#toc_58)
      6. [공유 저장소 설정하기](http://namhyung.springnote.com/pages/3132772#toc_59)
      7. [저장소를 웹으로 살펴볼 수 있게 공개하기](http://namhyung.springnote.com/pages/3132772#toc_60)
   5. [예제](http://namhyung.springnote.com/pages/3132772#toc_61)
      1. [리눅스 하위시스템 관리자를 위한 주제별 브랜치 관리하기](http://namhyung.springnote.com/pages/3132772#toc_62)
5. [변경 이력 수정하기와 패치 묶음 관리하기](http://namhyung.springnote.com/pages/3132772#toc_63)
   1. [완벽한 패치 묶음 만들기](http://namhyung.springnote.com/pages/3132772#toc_64)
   2. [git rebase를 이용하여 패치 묶음을 최신 상태로 유지하기](http://namhyung.springnote.com/pages/3132772#toc_65)
   3. [하나의 커밋 수정하기](http://namhyung.springnote.com/pages/3132772#toc_66)
   4. [패치 모음에서 커밋 순서 변경 및 커밋 선택하기](http://namhyung.springnote.com/pages/3132772#toc_67)
   5. [다른 도구들](http://namhyung.springnote.com/pages/3132772#toc_68)
   6. [변경 이력을 수정하는 것에 따른 문제점](http://namhyung.springnote.com/pages/3132772#toc_69)
   7. [머지 커밋 상에서 bisect를 수행하는 것이 선형 변경 이력 상에서 bisect를 수행하는 것보다 어려운 이유](http://namhyung.springnote.com/pages/3132772#toc_70)
6. [고급 브랜치 관리](http://namhyung.springnote.com/pages/3132772#toc_71)
   1. [각각의 브랜치 가져오기](http://namhyung.springnote.com/pages/3132772#toc_72)
   2. [git fetch 명령과 고속 이동](http://namhyung.springnote.com/pages/3132772#toc_73)
   3. [git fetch 명령이 고속 이동이 아닌 업데이트를 하도록 강제 지정하기](http://namhyung.springnote.com/pages/3132772#toc_74)
   4. [원격 브랜치 설정하기](http://namhyung.springnote.com/pages/3132772#toc_75)
7. [Git 개념 정리](http://namhyung.springnote.com/pages/3132772#toc_76)
   1. [객체 데이터베이스](http://namhyung.springnote.com/pages/3132772#toc_77)
      1. [커밋 객체](http://namhyung.springnote.com/pages/3132772#toc_78)
      2. [트리 객체](http://namhyung.springnote.com/pages/3132772#toc_79)
      3. [블롭 객체](http://namhyung.springnote.com/pages/3132772#toc_80)
      4. [신뢰](http://namhyung.springnote.com/pages/3132772#toc_81)
      5. [태그 객체](http://namhyung.springnote.com/pages/3132772#toc_82)
      6. [git가 객체를 효율적으로 저장하는 방법: 팩 파일](http://namhyung.springnote.com/pages/3132772#toc_83)
      7. [댕글링 객체](http://namhyung.springnote.com/pages/3132772#toc_84)
      8. [저장소 손상 시 복구하기](http://namhyung.springnote.com/pages/3132772#toc_85)
   2. [인덱스](http://namhyung.springnote.com/pages/3132772#toc_86)
8. [서브 모듈](http://namhyung.springnote.com/pages/3132772#toc_87)
   1. [서브모듈에 관한 함정](http://namhyung.springnote.com/pages/3132772#toc_88)
9. [저수준 git 연산](http://namhyung.springnote.com/pages/3132772#toc_89)
   1. [객체 접근 및 조작](http://namhyung.springnote.com/pages/3132772#toc_90)
   2. [작업 순서](http://namhyung.springnote.com/pages/3132772#toc_91)
      1. [작업 디렉터리 -> 인덱스](http://namhyung.springnote.com/pages/3132772#toc_92)
      2. [인덱스 -> 객체 데이터베이스](http://namhyung.springnote.com/pages/3132772#toc_93)
      3. [객체 데이터베이스 -> 인덱스](http://namhyung.springnote.com/pages/3132772#toc_94)
      4. [인덱스 -> 작업 디렉토리](http://namhyung.springnote.com/pages/3132772#toc_95)
      5. [이들을 모두 합치기](http://namhyung.springnote.com/pages/3132772#toc_96)
   3. [데이터 살펴보기](http://namhyung.springnote.com/pages/3132772#toc_97)
   4. [여러 트리를 머지하기](http://namhyung.springnote.com/pages/3132772#toc_98)
   5. [여러 트리를 머지하기 (계속)](http://namhyung.springnote.com/pages/3132772#toc_99)
10. [Git 해킹하기](http://namhyung.springnote.com/pages/3132772#toc_100)
    1. [객체 저장 형식](http://namhyung.springnote.com/pages/3132772#toc_101)
    2. [git 소스 코드의 개략적인 설명](http://namhyung.springnote.com/pages/3132772#toc_102)
11. [Git 용어](http://namhyung.springnote.com/pages/3132772#toc_103)
12. [부록 A: Git 빠른 참조](http://namhyung.springnote.com/pages/3132772#toc_104)
    1. [새 저장소 만들기](http://namhyung.springnote.com/pages/3132772#toc_105)
    2. [브랜치 관리하기](http://namhyung.springnote.com/pages/3132772#toc_106)
    3. [변경 이력 살펴보기](http://namhyung.springnote.com/pages/3132772#toc_107)
    4. [변경 사항 만들기](http://namhyung.springnote.com/pages/3132772#toc_108)
    5. [머지](http://namhyung.springnote.com/pages/3132772#toc_109)
    6. [변경 사항을 공유하기](http://namhyung.springnote.com/pages/3132772#toc_110)
    7. [저장소 관리](http://namhyung.springnote.com/pages/3132772#toc_111)
13. [부록 B: 이 설명서에 대한 노트 및 할 일 목록](http://namhyung.springnote.com/pages/3132772#toc_112)
14. [번역 용어표](http://namhyung.springnote.com/pages/3132772#toc_113)

**Git 사용자 설명서 (버전 1.5.3 이상)**

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

번역 : 김남형 <namhyung\_at\_gmail\_dot\_com> 2009-04-13 ~ 2009-08-26

**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 이력 조사하기"에서는 git를 이용하여 프로젝트를 가져오고 검토하는 방법을 설명한다. 특정 버전의 소프트웨어를 빌드하고 테스트하거나, 퇴보한(regression) 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.

1. $ man git-clone

혹은

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_0)

**git 저장소 가져오기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_1)

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.

1. # 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 이력에 대한 모든 정보를 포함하고 있는 ".git" 라는 특수한 최상위 디렉토리가 존재하는 것을 볼 수 있을 것이다.

**다른 버전의 프로젝트 체크아웃 하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_2)

git는 여러 파일들에 대한 변경 이력을 저장하기 위한 도구로 최선의 선택이다. git는 변경 이력을 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.

1. $ 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 이력에 대한 참조이며, **git-tag(1)**명령으로 태그의 목록을 볼 수 있다.

1. $ 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.

1. $ 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.

1. $ 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.

1. $ git reset --hard v2.6.17

기억해 두어야 할 것은 위의 경우에서 만약 현재 브랜치 헤드가 변경 이력 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 이력 이해하기: 커밋**[**#**](http://namhyung.springnote.com/pages/3132772#toc_3)

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

1. $ git show
2. commit 17cf781661e6d38f737f15f53ab552f1e95960d7
3. Author: Linus Torvalds <torvalds@ppc970.osdl.org.(none)>
4. Date:   Tue Apr 19 14:11:06 2005 -0700
6. Remove duplicate getenv(DB\_ENVIRONMENT) call
8. Noted by Tony Luck.
10. diff --git a/init-db.c b/init-db.c
11. index 65898fa..b002dc6 100644
12. --- a/init-db.c
13. +++ b/init-db.c
14. @@ -7,7 +7,7 @@
16. int main(int argc, char \*\*argv)
17. {
18. -       char \*sha1\_dir = getenv(DB\_ENVIRONMENT), \*path;
19. +       char \*sha1\_dir, \*path;
20. int len, i;
22. 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 이력 안에 저장되는 모든 것은 객체 안에 저장되고, 그 객체의 이름은 객체의 내용에 대한 해시값으로 만들어 진다는 것을 알게 될 것이다.

**변경 이력 이해하기: 커밋, 부모, 도달 가능성**[**#**](http://namhyung.springnote.com/pages/3132772#toc_4)

(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 경로가 다시 합쳐지는 지점을 "머지"라고 부른다. 따라서 머지를 가리키는 커밋은 둘 이상의 부모 커밋을 가지며, 각 부모 커밋은 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 이력을 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 이력 이해하기: 변경 이력 다이어그램**[**#**](http://namhyung.springnote.com/pages/3132772#toc_5)

우리는 때때로 아래와 같은 다이어그램을 이용하여 git 변경 이력을 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 이력 이해하기: 브랜치란 무엇인가?**[**#**](http://namhyung.springnote.com/pages/3132772#toc_6)

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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_7)

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 이력 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.

1. $ cat .git/HEAD
2. ref: refs/heads/master

**새 브랜치를 만들지 않고 이전 버전 살펴보기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_8)

**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:

1. $ git checkout v2.6.17
2. Note: moving to "v2.6.17" which isn't a local branch
3. If you want to create a new branch from this checkout, you may do so
4. (now or later) by using -b with the checkout command again. Example:
5. git checkout -b <new\_branch\_name>
6. 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:

1. $ cat .git/HEAD
2. 427abfa28afedffadfca9dd8b067eb6d36bac53f
3. $ git branch
4. \* (no branch)
5. 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_9)

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.

1. $ 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.

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_10)

브랜치, 원격 추적 브랜치, 태그는 모두 커밋을 참조한다. 모든 참조의 이름은 "refs"로 시작하고 슬래시(/) 기호로 구분되는 경로명으로 구성된다. 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_11)

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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_12)

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

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

1. $ 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:

1. $ 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 이력 조사하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_13)

git는 여러 파일들에 대한 변경 이력을 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 이력을 조사하기 위한 매우 유연하고 빠른 도구를 제공한다.

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

**regression을 찾기 위해 bisect 이용하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_14)

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

1. $ 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:

1. $ 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.

1. $ 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

1. $ 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:

1. $ 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:

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_15)

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 태그가 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:

1. $ git show fb47ddb2 # 보통은 객체 이름 첫 부분의 몇 글자로도
2. # 해당 커밋을 지정하기에 충분하다
3. $ git show HEAD^    # HEAD의 부모 커밋
4. $ git show HEAD^^   # HEAD의 부모의 부모 커밋
5. $ 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:

1. $ git show HEAD^1   # HEAD의 첫 번째 부모 커밋
2. $ 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

1. $ 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.

1. $ git rev-parse origin  
   e05db0fd4f31dde7005f075a84f96b360d05984b

**태그 만들기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_16)

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

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_17)

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

1. $ 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:

1. $ git log v2.5.. Makefile fs/

또한 git log에게 패치 형식으로 보여달라고 요청할 수도 있다.

1. $ 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 이력은 다중 개발 경로를 포함할 수 있기 때문에 커밋의 목록이 출력되는 상세한 순서는 약간 달라질 수 있다.

**차이점 생성하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_18)

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

1. $ 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.

1. $ git diff master...test

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

1. $ git format-patch master..test

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

**예전 버전의 파일 보기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_19)

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:

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_20)

**특정 브랜치 상의 커밋 개수 세기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_21)

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

1. $ 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:

1. $ git rev-list origin..mybranch | wc -l

**두 브랜치가 동일한 지점을 가리키는지 검사하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_22)

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

1. $ git diff origin..master

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

1. $ 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

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_23)

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 이력 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:

1. $ 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:

1. $ 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:

1. $ 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:

1. $ 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

1. $ 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

1. $ 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  
    ! [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

1. + ++ [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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_24)

"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:

1. $ 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:

1. $ 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:

1. $ 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:

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_25)

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

1. $ 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:

1. $ release-script 2.6.12 2.6.13-rc6 2.6.13-rc7

release-script는 다음과 같은 쉘 스크립트이다:

1. #!/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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_26)

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:

1. $  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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_27)

**git에게 이름 말해주기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_28)

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

1. [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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_29)

아무 것도 없는 상태에서 저장소를 새로 만드는 것은 아주 간단하다:

1. $ mkdir project  
   $ cd project  
   $ git init

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

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

**커밋을 만드는 방법**[**#**](http://namhyung.springnote.com/pages/3132772#toc_30)

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.

1. $ git add path/to/file

새 파일의 내용을 인덱스에 추가하려면 다음을 실행한다.

1. $ git add path/to/file

인덱스와 작업 트리에서 파일을 삭제하려면 다음을 실행한다.

1. $ git rm path/to/file

각각의 단계에서 다음 명령을 실행하면

1. $ git diff --cached

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

1. $ 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.

1. $ 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.

1. $ git show

특수한 바로 가기로써

1. $ 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:

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_31)

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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_32)

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:

1. # '#'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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_33)

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

1. $ 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.

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_34)

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:

1. $ 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:

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

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

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_35)

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:

1. $ 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\_HEAD에 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:

1. $ 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 스테이지 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:

1. $ 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:

1. $ 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:

1. $ 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:

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_36)

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

1. $ git reset --hard HEAD

혹은 머지한 것을 이미 커밋했을 때 되돌리려면 다음 명령을 실행한다.

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_37)

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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_38)

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

1. $ 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 이력이 이미 공개되었다면 절대 이렇게 해서는 안 된다. git는 일반적으로 프로젝트의 "변경 이력"이 변경되는 것을 고려하지 않기 때문에, 변경 이력이 변경된 브랜치로의 반복된 머지 작업은 올바로 동작하지 않을 것이다.

**새로운 커밋을 만들어 실수 바로잡기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_39)

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:

1. $ 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:

1. $ 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 이력을 수정하여 실수 바로잡기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_40)

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

1. $ 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 이력 상의 더 이전 상태로 되돌리는 것도 가능하다. 하지만 이러한 복잡한 주제는 다른 장에서 설명하도록 남겨 둔다.

**이전 버전의 파일을 체크아웃하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_41)

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:

1. $ 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:

1. $ git show HEAD^:path/to/file

위 명령은 해당 파일의 특정 버전을 보여줄 것이다.

**작업 중인 내용을 임시로 보관해 두기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_42)

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.

1. $ 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.

1. ... 수정 및 테스트...
2. $ git commit -a -m "blorpl: typofix"

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

1. $ git stash apply

**좋은 성능 보장하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_43)

거대한 저장소에서 git는 변경 이력 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.

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_44)

**저장소가 망가졌는지 검사하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_45)

**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:

1. $ git fsck  
   dangling commit 7281251ddd2a61e38657c827739c57015671a6b3  
   dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63  
   dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5  
   dangling blob 218761f9d90712d37a9c5e36f406f92202db07eb  
   dangling commit bf093535a34a4d35731aa2bd90fe6b176302f14f  
   dangling commit 8e4bec7f2ddaa268bef999853c25755452100f8e  
   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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_46)

**Reflogs**[**#**](http://namhyung.springnote.com/pages/3132772#toc_47)

여러분이 'git reset --hard' 명령을 이용하여 브랜치를 수정했다고 하자. 그런데 알고보니 이 브랜치가 해당 변경 이력에 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 이력에 여전히 접근할 수 있다.

1. $ 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:

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

1. $ 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 이력은 일반 git 변경 이력과는 매우 다르다는 것을 알아두도록 하자. 일반 변경 이력은 같은 프로젝트를 수행하는 모든 저장소에서 공유하지만, reflog 변경 이력은 공유되지 않는다. 이것은 오직 로컬 저장소 내의 브랜치들이 시간에 따라 어떻게 변경되어 왔는지 만을 보여준다.

**댕글링 객체 살펴보기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_48)

어떤 상황에서는 reflog도 도움이 되지 않을 수가 있다. 예를 들어 여러분이 어떤 디렉토리를 지웠다고 가정해 보자. 나중에 여러분은 이 브랜치에 포함된 변경 이력이 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.

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

예를 들면 다음과 같이 댕글링 객체 중의 하나를 살펴볼 수 있다

1. $ gitk 7281251ddd --not --all

위 명령은 다음과 같은 주문대로 동작한다: 댕글링 객체(들)로 설명하는 커밋 변경 이력을 보여주되, 다른 모든 브랜치나 태그에서 설명하는 변경 이력들은 보이지 않도록 하고 싶다. 따라서 잃어버린 커밋에서 도달할 수 있는 변경 이력 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 이력 전체가 제거되었을 수도 있다.)

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

1. $ git branch recovered-branch 7281251ddd

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

**개발 과정 공유하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_49)

**git pull 명령으로 업데이트하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_50)

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:

1. $ git fetch  
   $ git merge origin/master

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

1. $ 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:

1. $ 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

1. $ git pull . branch  
   $ git merge branch

위의 명령들은 거의 동일하다. 실제로는 위쪽의 방식이 주로 사용된다.

**프로젝트에 패치 제출하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_51)

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

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_52)

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.

1. $ 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

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_53)

프로젝트에 변경 내용을 제출하는 또 다른 방법은 해당 프로젝트의 관리자에게 **git-pull(1)** 명령을 이용하여 여러분의 저장소 내의 변경 이력을 가져가라고 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 이력을 직접 가져올 수 있다. 저장소 URL을 인자로 받는 명령들은 로컬 디렉토리 이름도 인자로 받을 수 있다:

1. $ git clone /path/to/repository  
   $ git pull /path/to/other/repository

혹은 SSH URL도 가능하다.

1. $ 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:

자신이 push

자신의 개인 저장소 ------------------> 자신의 공개 저장소

^ |

| 자신이 pull | 다른 사람이 pull

| 다른 사람이 push v

다른 사람의 공개 저장소 <------------- 다른 사람의 저장소

아래에서는 이 과정에 대해 설명한다.

**공개 저장소 설정하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_54)

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

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

결과로 생성되는 proj.git 디렉토리는 "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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_55)

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 넘겨주면 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_56)

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 bare git 저장소를 웹 서버를 통해 공개된 디렉토리에 복사하고, 웹 클라이언트에게 필요한 부가 정보들을 제공할 수 있도록 약간의 작업을 수행하기만 하면 된다:

1. $ 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 이력을 받아갈(pull) 수 있다. 예를 들어 명령행에서는 다음과 같이 할 수 있다:

1. $ git clone <http://yourserver.com/~you/proj.git>

(WebDAV를 이용하여 http 상에서 변경 이력을 올릴(push) 수 있도록 하는 좀 더 복잡한 설정 방법은 "[http 상의 git 서버 설정](http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt)" 문서를 살펴보기 바란다.)

**공개 저장소에 변경 이력 올리기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_57)

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.

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

혹은 단순히 다음과 같이 실행할 수 있다.

1. $ 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 "bare" 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

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

위의 push 명령을 다음과 같이 수행할 수 있다.

1. $ 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_58)

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

1. error: remote 'refs/heads/master' is not an ancestor of
2. local  'refs/heads/master'.
3. Maybe you are not up-to-date and need to pull first?
4. 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 이력을 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:

1. $ 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 이력을 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_59)

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 이력에 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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_60)

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

**예제**[**#**](http://namhyung.springnote.com/pages/3132772#toc_61)

**리눅스 하위시스템 관리자를 위한 주제별 브랜치 관리하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_62)

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:

1. $ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git work
2. $ 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. 1장 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.

1. $ git branch --track test origin/master
2. $ git branch --track release origin/master

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

1. $ git checkout test && git pull
2. $ git checkout release && git pull

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

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

1. $ cat >> .git/config <<EOF
2. [remote "mytree"]
3. url =  master.kernel.org:/pub/scm/linux/kernel/git/aegl/linux-2.6.git
4. push = release
5. push = test
6. EOF

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

1. $ git push mytree

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

1. $ git push mytree test

혹은

1. $ 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:

1. $ 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.

1. $ ... 패치 ... 테스트  ... 커밋 [ ... 패치 ... 테스트 ... 커밋 ]\*

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

1. $ 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.

1. $ 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:

1. $ git log linux..branchname | git shortlog

해당 브랜치가 test 혹은 release 브랜치에 머지되었는지 알아보려면 다음을 실행한다:

1. $ git log test..branchname

혹은

1. $ 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:

1. $ git log origin..branchname

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

1. $ 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:

1. $ git diff --stat origin..release

그리고

1. $ git log -p origin..release | git shortlog

아래는 지금까지 설명한 것들을 실행하는 스크립트들이다.

1. ==== 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

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

1. ==== 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"  
                   ;;  
           l)  
                   echo "Waiting for linus"  
                   ;;  
           "")  
                   echo $rb All done $restore  
                   ;;  
           \*)  
                   echo $rb "<$status>" $restore  
                   ;;  
           esac  
           git log origin/master..$branch | git shortlog  
   done

**변경 이력 수정하기와 패치 묶음 관리하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_63)

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**[**#**](http://namhyung.springnote.com/pages/3132772#toc_64)

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 이력을 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 regression이 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 이력을 수정했을 때 발생할 수 있는 문제점들에 대해서 설명할 것이다.

**git rebase를 이용하여 패치 묶음을 최신 상태로 유지하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_65)

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

1. $ git checkout -b mywork origin
2. $ vi file.txt
3. $ git commit
4. $ vi otherfile.txt
5. $ git commit
6. ...

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 이력을 머지 커밋 없이 단순히 연결된 커밋 상태로 유지하고 싶다면, **git-rebase(1)** 명령을 이용해 볼 수 있다:

1. $ git checkout mywork
2. $ 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:

1. $ 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:

1. $ git rebase --abort

**하나의 커밋 수정하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_66)

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

1. $ 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:

1. $ 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.)

1. $ 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:

1. $ 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.

**패치 모음에서 커밋 순서 변경 및 커밋 선택하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_67)

기존의 커밋에 대해서,  **git-cherry-pick(1)** 명령은 해당 커밋의 변경 내용을 적용하여 새로운 커밋으로 만들어 준다. 예를 들어, 만약 "mywork"가 "origin" 브랜치 상에서 작업한 일련의 패치들을 가리킨다고 하면, 다음을 실행하고:

1. $ git checkout -b mywork-new origin  
   $ gitk origin..mywork &

 gitk를 이용하여 mywork 브랜치 상의 패치 목록을 살펴본 후, cherry-pick 명령을 이용하여 (아마도 다른 순서로) 적용할 수 있다. 혹은 'git commit --amend' 명령을 통해 커밋을 수정할 수도 있다. 또한 **git-gui(1)** 명령은 인덱스 내의 차이점들 중에서 특정한 차이점들을 개별적으로 선택하여 적용할 수 있도록 해 준다. (차이점 상에서 오른쪽 클릭 후 "Stage Hunk for Commit" 항목을 선택한다.)

또 다른 방법은 'git format-patch' 명령을 이용하여 패치 묶음을 만든 뒤, 트리의 상태를 패치가 만들어지기 전으로 되돌리는 것이다:

1. $ git format-patch origin  
   $ git reset --hard origin

 그 후 원하는 대로 패치를 수정하거나 순서를 바꾸거나 삭제한 후에, **git-am(1)** 명령을 이용하여 패치 묶음을 다시 적용한다.

**다른 도구들**[**#**](http://namhyung.springnote.com/pages/3132772#toc_68)

패치 모음을 관리하기 위한 StGIT 등의 다른 많은 도구들이 존재한다. 이들은 이 설명서의 범위를 벗어나므로 여기서는 더 이상 설명하지 않는다.

**변경 이력을 수정하는 것에 따른 문제점**[**#**](http://namhyung.springnote.com/pages/3132772#toc_69)

브랜치의 변경 이력을 수정하는 것에 따르는 가장 큰 문제점은 머지에 대한 것이다. 누군가가 여러분의 브랜치의 작업 내용을 가져가서(fetch) 자신의 브랜치에 머지했다고 가정해 보자. 그 결과는 다음과 같을 것이다:

 o--o--O--o--o--o <-- origin  
        \        \  
         t--t--t--m <-- 다른 사람의 브랜치:

그 후에 여러분이 마지막 3개의 커밋을 수정했다고 하자:

         o--o--o <-- 수정 후의 origin 헤드  
        /  
o--o--O--o--o--o <-- 수정전의 origin 헤드

만약 이러한 모든 사항을 하나의 저장소 내에서 살펴본다면 다음과 같을 것이다:

o--o--o <-- 수정 후의 origin 헤드

/

o--o--O--o--o--o <-- 수정 전의 origin 헤드

\ \

t--t--t--m <-- 다른 사람의 브랜치:

 git는 (수정 후의) 새로운 헤드가 이전 헤드에서 업데이트된 버전이라는 것을 알지 못한다. git는 이 상황을 두 명의 개발자가 서로 다른 두 버전에 대해서 따로 개발해 왔을 때와 완전히 똑같이 동일하게 처리한다. 이 시점에서, 다른 사람이 자신의 브랜치에 새로운 헤드를 머지하려고 하면, git는 수정 전의 헤드를 수정 후의 헤드로 바꾸려고 시도하는 대신, 두 개발 경로 (수정 전과 수정 후)를 머지하려고 시도할 것이다. 이것은 아마도 예상할 수 없는 결과를 가져올 것이다.

여러분은 변경 이력이 수정된 트리도 공개하기를 원할 수 있고, 이는 다른 사람들이 내려받아서 변경 사항을 살펴보거나 테스트 하는 데 유용하게 사용될 수 있다. 하지만 이를 자신이 작업한 브랜치에 머지하려고 해서는 안 된다.

적절한 머지를 지원하는 진정한 분산 개발에서는, 공개된 브랜치의 변경 이력을 수정해서는 절대 안 된다.

**머지 커밋 상에서 bisect를 수행하는 것이 선형 변경 이력 상에서 bisect를 수행하는 것보다 어려운 이유**[**#**](http://namhyung.springnote.com/pages/3132772#toc_70)

**git-bisect(1)** 명령은 머지 커밋을 포함한 변경 이력 상에서도 올바로 동작한다. 하지만 bisect가 찾아낸 커밋이 머지 커밋이라면, 사용자는 왜 해당 커밋이 문제를 일으켰는지 알아내기 위해 평소보다 더 노력해야 할 수도 있다.

다음과 같은 변경 이력을 상상해 보자:

---Z---o---X---...---o---A---C---D

\ /

o---o---Y---...---o---B

 위쪽의 개발 경로에서, Z에 존재하는 함수 중 하나의 의미가 X에서 변경되었다고 가정해 보자. Z와 A 사이의 커밋들은 함수의 구현과 Z 상에 존재하는 이 함수를 호출하는 부분을 변경된 의미에 맞게 모두 수정하였으며, 새로 이 함수를 호출하는 부분도 추가하였다. A 커밋 상에서 버그는 없다.

동시에 아래쪽 개발 경로에서는, 누군가가 (위에서 수정한) 해당 함수를 호출하는 부분을 새로 추가했다고 가정해 보자. Z와 B 사이의 커밋들은 해당 함수의 의미가 변경되지 않았다고 생각하며, 함수의 구현과 호출하는 부분은 일관성을 가진다. B 커밋에서도 버그는 없다.

또한 두 개발 경로가 C 커밋에서 정상적으로 머지되었다고 가정해 보자. 이 시점에서 해결해야 할 충돌은 발생하지 않았다.

하지만 C의 코드는 문제가 있다. 왜냐하면 아래쪽 개발 경로에서 해당 함수를 호출한 부분은, 위쪽 개발 경로에서 의미를 변경한 대로 수정되지 않았기 때문이다. 만약 여러분이 단지 D에서는 문제가 있고 Z에서는 문제가 없다는 사실 만을 알고 있고, **git-bisect(1)** 명령이 C가 문제라고 찾아냈다면, 이 문제가 함수의 의미가 변경되었기 때문에 발생한 것이라는 어떻게 알아낼 수 있을까?

만약 'git bisect' 명령이 찾아낸 것이 머지 커밋이 아니라면, 일반적으로 해당 커밋을 자세히 살펴보고 문제를 발견할 수 있을 것이다. 개발자들은 자신의 변경 내용을 작은 단위의 완전한 커밋으로 나누어서, 쉽게 문제를 발견하도록 만들 수 있다. 하지만 위의 경우에는 이러한 방법도 통하지 않는데, 그것은 이 문제가 각각의 커밋을 자세히 살펴본다고 해도 명확하게 드러나지 않기 때문이다. 대신 이 경우에는 전체적인 개발 과정에 대한 안목이 필요하다. 문제가 되었던 함수의 의미가 변경된 것이 위쪽 개발 경로 상의 변경 내용 중 극히 일부일 경우에는, 상황이 더욱 악화될 수 있다.

 다른 방법으로, C 시점에서 머지를 하는 대신에, A 상에서 Z부터 B 사이의 변경 이력을 수정(rebase)하는 방법이 있다. 이 경우 다음과 같은 선형 변경 이력을 갖게 될 것이다:

---Z---o---X--...---o---A---o---o---Y\*--...---o---B\*--D\*

Z와 D\*에 대해 bisect 명령을 수행하면, Y\* 커밋이 문제라는 것을 찾아낼 것이고, Y\* 커밋이 문제가 되는 이유를 찾아보는 것이 아마도 더 쉬울 것이다.

이런 이유에서라도, 많은 숙련된 git 사용자들은, 머지 작업이 많은 시간을 소모하는 프로젝트에서 작업하는 경우에도, 변경 내용을 공개하기 전에 최신 업스트림 버전에 대해 변경 이력을 정리(rebase)하여 변경 이력을 선형으로 유지한다.

**고급 브랜치 관리**[**#**](http://namhyung.springnote.com/pages/3132772#toc_71)

**각각의 브랜치 가져오기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_72)

**git-remote(1)** 명령을 이용하는 대신, 한 번에 하나의 브랜치를 선택하여 임의의 이름으로 로컬에 저장할 수 있다:

1. $ git fetch origin todo:my-todo-work

첫 번째 인자인 "origin"은 git가 원래 복사해 온 저장소에서 소스를 내려받도록 지정한다. 두 번째 인자는 git가 원격 저장소의 "todo" 브랜치를 가져와서 로컬의 refs/heads/my-todo-work에 저장하도록 지정한다.

또한 다른 저장소에서 브랜치를 내려받을 수도 있다.

1. $ git fetch git://example.com/proj.git master:example-master

위 명령은 주어진 저장소에서 "master"라는 이름의 브랜치를 내려받아서 "example-master"라는 새 브랜치에 저장할 것이다. 만약 example-master라는 브랜치가 이미 존재한다면, example.com의 master 브랜치의 커밋까지 고속 이동을 시도할 것이다. 더 자세한 사항은 다음에 설명한다:

**git fetch 명령과 고속 이동**[**#**](http://namhyung.springnote.com/pages/3132772#toc_73)

앞의 예제에서 기존의 브랜치를 업데이트했을 때, "git fetch" 명령은 새 커밋이 가리키는 곳으로 브랜치를 업데이트하기 전에 원격 브랜치 상의 가장 최신 커밋이 로컬의 브랜치 복사본 상의 최신 커밋의 자손인지 확인한다. 이러한 과정을 git에서는 고속 이동(fast-forward)이라고 한다.

고속 이동은 다음과 같이 볼 수 있다:

 o--o--o--o <-- 브랜치의 이전 헤드  
           \  
            o--o--o <-- 브랜치의 새로운 헤드

어떤 경우에는 새로운 헤드가 실제로 이전 헤드의 자손이 **아닌** 경우가 있을 수 있다. 예를 들어 개발자가 무언가를 수정한 후에 실수를 깨달았고, 변경 이력을 되돌아가기로 결정했다면 다음과 같은 상황이 될 것이다:

 o--o--o--o--a--b <-- 브랜치의 이전 헤드  
           \  
            o--o--o <-- 브랜치의 새로운 헤드

위의 경우, "git fetch" 명령은 경고를 보여주며 실패할 것이다.

이 경우 다음에서 설명하듯이 강제로 git가 새로운 헤드로 업데이트 하도록 지정할 수 있다. 하지만 이러한 상황에서 강제 업데이트를 수행한다면, (따로 해당 커밋에 대한 참조를 만들어두지 않았다면) "a"와 "b" 커밋에 대한 참조를 잃어버린다는 것을 뜻한다.

**git fetch 명령이 고속 이동이 아닌 업데이트를 하도록 강제 지정하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_74)

만약 새로운 헤드가 이전 헤드의 자손이 아니라는 이유로 'git fetch' 명령이 실패한다면, 다음과 같이 강제 업데이트를 수행하도록 지정할 수 있다:

1. $ git fetch git://example.com/proj.git +master:refs/remotes/example/master

"+" 기호가 추가된 것에 유의하자. 또는 "-f" 플래그를 이용하여 모든 내려받은(fetched) 브랜치들을 강제로 업데이트하도록 지정할 수 있다:

1. $ git fetch -f origin

앞에서 설명한대로 example/master가 가리키던 이전 버전의 커밋들을 잃어버릴 수도 있다는 것을 기억하기 바란다.

**원격 브랜치 설정하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_75)

앞서 "origin"은 단지 원래 내려받은 저장소를 가리키는 줄임말에 불과하다는 것을 살펴보았다. 이 정보는 **git-config(1)** 명령으로 확인할 수 있는 git 설정 변수에 저장된다:

1. $ git config -l  
   core.repositoryformatversion=0  
   core.filemode=true  
   core.logallrefupdates=true  
   remote.origin.url=git://git.kernel.org/pub/scm/git/git.git  
   remote.origin.fetch=+refs/heads/\*:refs/remotes/origin/\*  
   branch.master.remote=origin  
   branch.master.merge=refs/heads/master

만약 자주 이용하는 다른 저장소가 있다면, 타이핑 횟수를 줄이기 위해 비슷한 설정 옵션을 만들 수도 있다. 예를 들어

1. $ git config remote.example.url git://example.com/proj.git

위 명령을 실행한 후에는, 다음 두 명령이 같은 작업을 수행할 것이다:

1. $ git fetch git://example.com/proj.git master:refs/remotes/example/master  
   $ git fetch example master:refs/remotes/example/master

게다가, 다음과 같은 옵션도 추가했다고 하면:

1. $ git config remote.example.fetch master:refs/remotes/example/master

다음 명령들은 모두 같은 작업을 수행할 것이다:

1. $ git fetch git://example.com/proj.git master:refs/remotes/example/master  
   $ git fetch example master:refs/remotes/example/master  
   $ git fetch example

또한 매번 강제 업데이트를 하도록 지정하기 위해 "+" 기호를 추가할 수도 있다:

1. $ git config remote.example.fetch +master:ref/remotes/example/master

"git fetch" 명령이 *example/master* 상의 커밋들을 잃어버리게 만들어도 상관없다는 확신이 없다면 이 기능을 사용하지 않는 것이 좋다.

그리고 위의 모든 설정은 **git-config(1)** 명령을 이용하는 대신 직접 .git/config 파일을 편집해서도 설정이 가능하다.

위에서 설명한 설정 옵션들에 대한 자세한 정보는 **git-config(1)** man 페이지를 살펴보기 바란다.

**Git 개념 정리**[**#**](http://namhyung.springnote.com/pages/3132772#toc_76)

git는 단순하지만 강력한 여러 가지 아이디어들로 구성되어 있다. 비록 이러한 내용들을 이해하지 않고서도 git를 잘 사용할 수 있지만, 이를 이해한다면 git를 좀 더 직관적으로 사용할 수 있을 것이다.

먼저 가장 중요한 객체 데이터베이스와 인덱스에 대해서 살펴보기로 하자.

**객체 데이터베이스**[**#**](http://namhyung.springnote.com/pages/3132772#toc_77)

우리는 이미 "변경 이력 이해하기: 커밋"이라는 부분에서 모든 커밋들은 40자의 "객체 이름"으로 저장된다는 것을 보았다. 사실, 프로젝트의 변경 이력을 보여주는 데 필요한 모든 정보는 이 이름과 함께 객체에 저장되어 있다. 각각의 경우에 대해서 객체 이름은 객체의 내용에 SHA-1 해시값을 계산한 것이다. SHA-1는 암호화에 쓰이는 해시 함수이다. 이것은 두 개의 서로 다른 객체가 같은 이름을 가질 수 없다는 것을 의미한다. 이것은 다음과 같은 장점을 가진다:

* git는 두 객체의 이름 만을 비교하여 두 객체가 서로 같은지 다른지 빨리 판단할 수 있다.
* 객체 이름은 모든 저장소에서 동일한 방법으로 계산되기 때문에, 서로 다른 저장소에 저장된 동일한 객체는 항상 같은 이름으로 저장된다.
* 객체를 읽을 때 객체의 이름과 객체의 내용에 대한 해시값이 일치하는 검사하여 에러가 발생했는지 알 수 있다.

(객체 형식과 SHA-1 계산에 대한 자세한 내용은 "객체 저장 형식" 부분을 살펴보기 바란다.)

객체에는 다음과 같은 네 가지 종류가 있다: "블롭", "트리", "커밋", "태그"

"블롭" 객체는 파일 내용을 저장하기 위해 사용된다.

"트리" 객체는 하나 이상의 "블롭" 객체들을 디렉터리 구조로 묶는다. 또한 트리 객체는 다른 트리 객체를 참조할 수 있으며, 따라서 디렉터리 계층이 구성된다.

"커밋" 객체는 이러한 디렉터리 계층들을 리비전들의 방향있는 비순환 그래프로 묶는다. 각 커밋은 커밋이 만들어진 시점의 디렉터리 계층을 지정하는 오직 하나의 트리의 객체 이름을 포함한다. 그리고 부모 커밋 객체를 참조하는 커밋은 해당 디렉토리 계층에 도달하기까지의 변경 이력을 설명한다.

"태그" 객체는 다른 객체를 이름으로 참조하거나 서명하는 데 사용될 수 있다. 태그 객체는 다른 객체의 이름과 타입, (당연히!) 태그 이름을 포함하며, 선택적으로 서명을 추가할 수도 있다.

이들 객체 타입에 대해서는 아래에서 자세히 설명한다:

**커밋 객체**[**#**](http://namhyung.springnote.com/pages/3132772#toc_78)

"커밋" 객체는 트리의 물리적인 상태와 그 상태에 도달하기까지의 방법과 이유에 대한 설명을 연결한다. 커밋 내용을 살펴보려면**git-log(1)** 혹은 **git-show(1)** 명령에 --pretty=raw 옵션을 사용하기 바란다:

1. $ git show -s --pretty=raw 2be7fcb476  
   commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4  
   tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf  
   parent 257a84d9d02e90447b149af58b271c19405edb6a  
   author Dave Watson <dwatson@mimvista.com> 1187576872 -0400  
   committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700  
     
       Fix misspelling of 'suppress' in docs  
     
       Signed-off-by: Junio C Hamano <gitster@pobox.com>

위에서 볼 수 있듯이 커밋은 다음과 같은 사항들로 정의된다:

* 트리: 특정한 시간의 디렉터리의 내용을 나타내는 (아래에서 설명할) 트리 객체의 SHA1 이름
* 부모(들): 프로젝트의 변경 이력 상에서 바로 이전의 단계를 나타내는 커밋(들)의 SHA1 이름. 위의 예제에서는 하나의 부모를 가지고 있지만, 머지 커밋의 경우에는 둘 이상의 부모를 가질 수 있다. 부모를 가지지 않는 커밋은 "루트" 커밋이라고 하며, 프로젝트의 최초 버전을 나타낸다. 모든 프로젝트는 최소 하나의 루트 커밋을 가져야 한다. 프로젝트는 여러 개의 루트 커밋을 가질 수도 있지만, 이런 상황은 일반적이지 않다 (또한 그리 좋은 아이디어도 아니다).
* 작성자: 이 변경 사항을 작성한 사람의 이름과 작성한 날짜
* 커미터: 실제로 커밋을 만든 사람의 이름과 만든 날짜. 이것은 작성자와 달라질 수 있는데, 예를 들어 누군가가 패치를 만들어서 개발자에게 이메일로 보낸 경우 등이 있다.
* 이 커밋에서 변경한 내용에 대한 설명

커밋 그 자체로는 실제로 어떤 사항들이 변경되었는지에 대한 정보는 포함하지 않는다는 것에 주의하자. 모든 변경 사항은 이 커밋이 가리키는 트리의 내용과 부모가 가리키는 트리를 비교하여 계산된다. 특히, git는 파일 이름 변경을 명시적으로 기록하지 않는다. 하지만 동일한 파일 데이터가 다른 경로에 존재하는 것을 인식하여 이름이 변경되었음을 나타낸다. (예를 들어 **git-diff(1)** man 페이지의 -M 옵션 부분을 살펴보기 바란다).

커밋은 보통 **git-commit(1)** 명령을 통해 만들어지며, 이 커밋의 부모는 보통 현재 HEAD로, 트리는 현재 인덱스 상에 저장된 내용이 가리키는 값으로 설정된다.

**트리 객체**[**#**](http://namhyung.springnote.com/pages/3132772#toc_79)

트리 객체를 살펴보기 위해서는 다양한 기능을 가진 **git-show(1)** 명령을 이용할 수도 있지만, **git-ls-tree(1)** 명령이 더욱 자세한 정보를 보여준다:

1. $ git ls-tree fb3a8bdd0ce  
   100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c    .gitignore  
   100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d    .mailmap  
   100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING  
   040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745    Documentation  
   100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200    GIT-VERSION-GEN  
   100644 blob 289b046a443c0647624607d471289b2c7dcd470b    INSTALL  
   100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1    Makefile  
   100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52    README  
   ...

위에서 볼 수 있듯이, 트리 객체는 이름 순으로 정렬된 여러 항목들을 가지고 있으며 각 항목들은 모드, 객체 타입, SHA1 이름 및 (일반) 이름을 가진다.

객체 타입은 파일의 내용을 나타내는 블롭(blob)이나 하위 디렉터리의 내용을 나타내는 다른 트리가 될 것이다. 트리와 블롭은 다른 객체들과 마찬가지로 내용에 대한 SHA1 해시값에 해당하는 이름을 가지므로, 두 트리는 오직 (하위 디렉터리의 내용들을 재귀적으로 포함하여) 트리의 내용이 동일한 경우에만 같은 SHA1 이름을 가질 수 있다. 이러한 특성은 git가 동일한 객체 이름을 가지는 항목들을 무시할 수 있기 때문에, 연관된 두 트리 객체 간의 차이점을 빨리 판단할 수 있도록 해 준다.

(주의: 서브 모듈이 있는 상황에서, 트리는 커밋 객체도 항목에 포함할 수 있다. 자세한 내용은 8장 서브 모듈 부분을 살펴보기 바란다.)

모든 파일들의 권한이 644 혹은 755로 설정된 것에 주의하자. git는 실제로 실행 권한 비트에 대해서만 관심을 가진다.

**블롭 객체**[**#**](http://namhyung.springnote.com/pages/3132772#toc_80)

블롭의 내용을 살펴보기 위해서는 **git-show(1)** 명령을 이용할 수 있다. 예를 들어 위의 트리에서 COPYING 항목 내의 블롭을 살펴보기로 하자:

1. $ git show 6ff87c4664  
     
    Note that the only valid version of the GPL as far as this project  
    is concerned is \_this\_ particular version of the license (ie v2, not  
    v2.2 or v3.x or whatever), unless explicitly otherwise stated.  
   ...

"블롭" 객체는 데이터의 바이너리 표현일 뿐이며, 다른 객체를 가리키거나, 어떠한 속성도 가지지 않는다.

블롭은 오직 데이터에 의해서만 정의되기 때문에, 디렉토리 트리 내의 (혹은 저장소의 여러 버전 내의) 두 파일의 내용이 동일하다면 같은 블롭 객체를 공유하게 될 것이다. 블롭 객체는 디렉터리 트리 내의 위치와 전혀 무관하며, 파일의 이름을 바꾸는 것은 그 파일에 해당하는 블롭 객체를 변경하지 않는다.

어떠한 트리 객체 혹은 블롭 객체든지 <리비전>:<경로> 형식을 통해 git-show(1) 명령으로 살펴볼 수 있다는 것을 기억하자. 이것은 현재 체크아웃하지 않은 트리의 내용을 살펴볼 때 유용하게 사용될 수 있다.

**신뢰**[**#**](http://namhyung.springnote.com/pages/3132772#toc_81)

만약 여러분이 어떤 곳으로부터 블롭 객체의 SHA1 이름을 받고 다른 곳에서 블롭 객체의 내용을 받았다면, SHA1 이름이 일치하는 한 이 객체의 내용이 올바르다는 것을 보장할 수 있다. 이것은 SHA1가 서로 다른 내용에서 동일한 해시를 생성할 수 없도록 설계되었기 때문이다.

마찬가지로, 어떤 트리 객체가 가리키는 디렉터리의 전체 내용을 신뢰하기 위해서는 최상위 트리 객체의 SHA1 이름 만을 신뢰할 수 있으면 된다. 또한 신뢰할 수 있는 곳으로부터 커밋의 SHA1 이름을 받았다면, 해당 커밋의 부모를 통해 도달할 수 있는 커밋들의 전체 변경 이력과 이 커밋이 참조하는 트리의 모든 내용들을 쉽게 검증할 수 있다.

따라서 시스템에 진정한 신뢰성을 제공하려면, 최상위 커밋의 이름을 포함하는 단지 *하나의* 특별한 노트에 디지털 서명을 하기만 하면 된다. 여러분의 디지털 서명은 다른 사람들이 여러분이 이 커밋을 신뢰하고 있다는 것을 보여주며, 커밋 변경 이력의 불변성은 다른 사람들이 전체 변경 이력을 신뢰할 수 있다는 것을 보여준다.

다시 말하면, 최신 커밋의 이름(SHA1 해시값)을 (PGP/GPG와 같은 도구를 이용하여) 디지털 서명해서 다른 사람들에게 이메일로 보내기만 하면, 전체 내용이 올바르다는 것을 간단히 검증해 보일 수 있는 것이다.

이를 돕기 위해 git는 태그 객체를 제공한다..

**태그 객체**[**#**](http://namhyung.springnote.com/pages/3132772#toc_82)

태그 객체는 객체, 객체 타입, 태그 이름, 태그를 만든 사람("tagger")의 이름과 (디지털 서명을 포함할 수 있는) 메시지를 포함하며 이는 **git-cat-file(1)** 명령을 통해 볼 수 있다:

1. $ git cat-file tag v1.5.0  
   object 437b1b20df4b356c9342dac8d38849f24ef44f27  
   type commit  
   tag v1.5.0  
   tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000  
     
   GIT 1.5.0  
   -----BEGIN PGP SIGNATURE-----  
   Version: GnuPG v1.4.6 (GNU/Linux)  
     
   iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui  
   nLE/L9aUXdWeTFPron96DLA=  
   =2E+0  
   -----END PGP SIGNATURE-----

태그 객체를 만들거나 검증하는 방법은 **git-tag(1)** man 페이지를 살펴보기 바란다. (**git-tag(1)** 명령은, 실제 태그 객체는 아니지만 단지 이름이 "refs/tags/"로 시작하는 단순한 참조인 "가벼운" 태그를 만드는 데 사용할 수도 있다는 것을 기억하자.)

**git가 객체를 효율적으로 저장하는 방법: 팩 파일**[**#**](http://namhyung.springnote.com/pages/3132772#toc_83)

새로 만들어진 객체들은 (.git/objects 디렉터리에  저장되는) 객체의 SHA1 해시값을 이름으로 하는 파일 내에 생성된다.

불행히도 이 방식은 프로젝트가 많은 객체를 소유하게 되면 비효율적이 된다. 오래된 프로젝트에서 다음 명령을 실행해 보기 바란다:

1. $ git count-objects  
   6930 objects, 47620 kilobytes

첫 번째 숫자는 별도의 파일로 저장된 객체들의 개수이며, 두 번째 숫자는 이러한 "느슨한" (loose) 객체들이 차지하는 공간의 양이다.

이러한 느슨한 객체들을 효율적인 압축 방식으로 저장하는 "팩 파일"로 옮기면 공간을 절약하는 것은 물론 git의 속도도 빠르게 할 수 있다. 팩 파일의 형식에 대한 자세한 정보는 technical/pack-format.txt 부분을 살펴보기 바란다.

느슨한 객체들을 팩 파일로 옮기려면 단지 git repack 명령을 실행한다:

1. $ git repack  
   Generating pack...  
   Done counting 6020 objects.  
   Deltifying 6020 objects.  
    100% (6020/6020) done  
   Writing 6020 objects.  
    100% (6020/6020) done  
   Total 6020, written 6020 (delta 4070), reused 0 (delta 0)  
   Pack pack-3e54ad29d5b2e05838c75df582c65257b8d08e1c created.

그리고 다음 명령을 실행하면

1. $ git prune

이제 팩 파일에 저장된 느슨한 객체들(의 원본)을 삭제할 수 있다. 또한 이 명령은 참조되지 않는 모든 객체들도 삭제할 것이다. (이러한 객체들은 예를 들어 "git-reset" 명령을 통해 커밋을 삭제한 경우에 만들어진다.) .git/objects 디렉터리를 확인하거나 다음 명령을 실행하여 느슨한 객체들이 사라진 것을 확인할 수 있다.

1. $ git count-objects  
   0 objects, 0 kilobytes

비록 객체 파일을 사라졌지만, 이 객체들을 참조하는 모든 명령은 전과 동일하게 동작할 것이다.

**git-gc(1)** 명령은 이러한 packing, pruning 등의 작업을 대신해 주므로, 일반적으로는 gc 명령 하나 만을 이용하면 된다.

**댕글링 객체**[**#**](http://namhyung.springnote.com/pages/3132772#toc_84)

**git-fsck(1)** 명령은 때때로 댕글링 객체에 대한 경고를 보여줄 것이다. 이것은 문제가 아니다.

댕글링 객체가 발생하는 주된 원인은 rebase 명령을 이용하여 브랜치의 변경 이력을 수정하였거나, 그렇게 변경 이력을 수정한 사람으로부터 변경 이력을 내려 받은 (pull) 경우이다. - 5장 변경 이력 수정하기와 패치 묶음 관리하기 부분을 살펴보기 바란다. 이 경우 원래 브랜치의 이전 헤드와 그것이 가리키는 모든 것들은 여전히 존재하며, 브랜치 포인터 자체는 변경되었으므로 이전의 것이 존재하지 않는다.

물론 댕글링 객체를 만드는 다른 경우도 존재한다. 예를 들어 어떤 파일에 대해 "git add" 명령을 수행하고 나서 커밋을 실행하기 전에, 다른 변경도 함께 적용하기 위해 해당 파일의 다른 부분을 수정하고, **수정된** 파일을 커밋하는 경우에는 "댕글링 블롭"이 만들어 진다. 원래 add 명령을 수행한 이전의 상태는 더이상 어떠한 커밋이나 트리에서 참조하지 않으므로 댕글링 블롭 객체가 된다.

마찬가지로 "recursive" 머지 전략이 실행될 때, 서로 얽힌 (criss-cross) 머지들이 존재하고 따라서 둘 이상의 머지 베이스가 존재한다는 것을 (이런 경우는 매우 드물지만 분명히 존재한다) 찾아내면 임시 내부 머지 베이스로 사용할 하나의 (만약 많은 수의 서로 얽힌 머지들과 셋 이상의 머지 베이스가 존재한다면 더 많아질 수 있다) 임시 중간 트리(temporary midway tree)를 생성할 것이다. 여기서도 이들은 실제 객체이지만 최종 결과는 이들을 가리키지 않기 때문에 저장소 내에 "댕글링" 객체로 남게 된다.

일반적으로, 댕글링 객체는 전혀 신경쓸 만 한 것이 아니다. 오히려 어떤 경우에는 유용하게 사용할 수 있다. 만약 작업 내용이 엉망이 된 경우에는 댕글링 객체를 이용하여 이전 트리를 복구할 수도 있다. (예를 들어 rebase 명령을 수행한 후에 보니 이것이 원하던 것이 아니었음을 알게된다면, 댕글링 객체들을 살펴보고 헤드를 이전의 댕글링 객체의 상태로 되돌리도록 (reset) 할 수 있다.)

커밋 객체에 대해서는 다음 명령을 이용할 수 있다:

1. $ gitk <댕글링 커밋의 SHA1 해시값> --not --all

위 명령은 주어진 커밋으로부터는 도달할 수 있지만 다른 어떤 브랜치, 태그 및 다른 참조로부터는 도달할 수 없는 모든 변경 이력을 보여준다. 만약 이것이 원하던 것이라고 판단되면 언제든 이를 가리키는 새로운 참조를 만들 수 있다. 예를 들면 다음과 같다.

1. $ git branch recovered-branch <dangling-commit-sha-goes-here>

블롭이나 트리 객체에 대해서는 위와 동일한 작업을 수행할 수 없지만, 여전히 그 내용을 볼 수는 있다. 다음 명령을 이용하면

1. $ git show <댕글링 블롭/트리의 SHA1 해시값>

해당 블롭의 원래 내용 (트리에 대해서는 기본적으로 "ls" 명령의 결과)를 볼 수 있으며, 이를 통해 댕글링 객체를 생성한 작업 내용에 대한 아이디어를 얻을 수도 있을 것이다.

보통 댕글링 블롭과 댕글링 트리 객체는 별로 중요하지 않다. 이들은 거의 대부분 처리가 완료되지 않은 (half-way) 머지 베이스의 결과(때때로 머지 충돌이 발생한 경우 수동으로 이를 해결했다면 댕글링 블롭 객체는 머지 충돌 표식을 포함하고 있을 수도 있다)이거나 단순히 "git-fetch" 명령이 수행될 때 ^C 혹은 그와 비슷한 방식으로 중단된 경우 객체 데이터베이스 내에 일부 새로운 객체들이 남아있는 경우이며 이들은 쓸모가 없다.

어쨌든 어떠한 댕글링 상태도 필요하지 않다고 판단되면, 도달할 수 없는 모든 객체들을 제거(prune)해 버릴 수 있다:

1. $ git prune

이제 댕글링 객체들은 사라질 것이다. 하지만 "git prune" 명령은 저장소에 변경 사항이 없을 때에만 실행해야 한다. 이것은 파일 시스템에 fsck 명령을 실행하여 복구하는 것과 비슷하다. 여러분은 파일 시스템이 마운트된 상태에서 이러한 작업을 하고 싶지는 않을 것이다.

(이 내용은 "git-fsck" 명령 자체에도 적용된다. 하지만 git-fsck 명령은 실제로 아무 것도 **변경하지 않고**, 발견한 내용을 보고만 해 주기 때문에 git-fsck 명령 자체를 실행하는 것은 전혀 "위험하지 않다". 누군가가 저장소의 내용을 변경하고 있는 동안 git-fsck 명령을 실행한다면 혼란스러운 메시지를 보여줄 수도 있지만 실제로 문제가 될 만한 작업은 수행하지 않는다. 반대로 누군가가 저장소의 내용을 변경하고 있는 동안 git-prune 명령을 실행하는 것은 좋은 생각이 **아니다**.)

**저장소 손상 시 복구하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_85)

git는 자신이 신뢰하는 데이터를 조심스럽게 다루도록 설계되었다. 하지만 git 자체에는 버그가 없다 하더라도 하드웨어 혹은 운영체제의 에러로 인해 데이터가 손상될 수 있다.

이러한 문제에 대한 일차적인 대책은 백업이다. git 디렉토리는 clone 명령 혹은 단순한 cp, tar 등의 다른 백업 방법을 통해 백업할 수 있다.

마지막 방법으로는 손상된 객체들을 찾아서 일일이 변경해 줄 수도 있다. 이 방법을 사용하기 전에는 저장소가 더욱 손상되지 않도록 미리 백업을 해 두도록 하자.

우리는 문제가 하나의 손상되었거나 사라진 블롭으로인해 생긴 것이라고 가정할 것이다. 이러한 문제는 때때로 해결이 가능하다. (사라진 트리나 특히 커밋들을 복구하는 것은 **무척** 힘들다.)

시작하기 전에 **git-fsck(1)** 명령을 통해 저장소가 손상되었는지, 그렇다면 손상된 부분이 어디인지 확인한다. 이 작업은 시간이 꽤 걸릴 수도 있다.

여기서는 다음과 같이 출력되었다고 가정하자:

1. $ git fsck --full  
   broken link from    tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8  
                 to    blob 4b9458b3786228369c63936db65827de3cc06200  
   missing blob 4b9458b3786228369c63936db65827de3cc06200

(보통은 댕글링 객체에 대한 메시지들도 함께 출력될 것이지만, 여기에서는 고려하지 않는다.)

이제 2d9263c6 트리가 가리키는 4b9458b3 블롭 객체가 사라진 것을 알았다. 만약 (아마도 다른 저장소에서) 해당 블롭 객체의 복사본을 찾을 수 있다면, 이 객체를 .git/objects/4b/9458b3… 로 복사하여 문제를 해결할 수 있다. 하지만 이것이 불가능하다고 가정하자. **git-ls-tree(1)** 명령을 이용하여 해당 블롭을 가리키고 있는 트리를 살펴볼 수 있다. 이 명령은 다음과 같은 메시지를 출력할 것이다:

1. $ git ls-tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8  
   100644 blob 8d14531846b95bfa3564b58ccfb7913a034323b8    .gitignore  
   100644 blob ebf9bf84da0aab5ed944264a5db2a65fe3a3e883    .mailmap  
   100644 blob ca442d313d86dc67e0a2e5d584b465bd382cbf5c    COPYING  
   ...  
   100644 blob 4b9458b3786228369c63936db65827de3cc06200    myfile  
   ...

이제 사라진 블롭이 "myfile"이라는 이름의 파일이라는 것을 알 수 있다. 그리고 해당 파일이 있는 디렉터리를 알아낼 경우도 있다. 여기서는 "somedirectory"라는 디렉터리 내에 myfile이 존재한다고 해 보자. 만약 운이 좋다면 사라진 블롭이 현재 체크아웃한 작업 트리 내의 "somedirectory/myfile"의 복사본과 동일할 수 있다. 이를 확인하려면 **git-hash-object(1)** 명령을 이용할 수 있다:

1. $ git hash-object -w somedirectory/myfile

위 명령은 somedirectory/myfile의 내용으로 블롭 객체를 만들어서 저장하고, 객체의 SHA-1 해시값을 출력한다. 만약 아주 운이 좋다면 예상대로 이 값은 4b9458b3786228369c63936db65827de3cc06200이 될 것이고, 손상은 해결된다!

그렇지 않다면 더 많은 정보가 필요하다. 해당 파일의 어느 버전이 사라진 것인지 어떻게 알려줄 것인가?

가장 간단한 방법은 다음 명령을 실행하는 것이다:

1. $ git log --raw --all --full-history -- somedirectory/myfile

위 명령에서 raw 객체를 요청하였기 때문에 다음과 같이 출력될 것이다.

1. commit abc  
   Author:  
   Date:  
   ...  
   :100644 100644 4b9458b... newsha... M somedirectory/myfile  
     
     
   commit xyz  
   Author:  
   Date:  
     
   ...  
   :100644 100644 oldsha... 4b9458b... M somedirectory/myfile

이것은 해당 파일의 바로 이전 버전이 "newsha"이고, 바로 다음 버전이 "oldsha"라는 것을 보여준다.(?) 또한 oldsha에서 4b9458b까지의 변경 사항과 4b9458b에서 newsha까지의 변경 사항에 대한 커밋 메시지들도 알 수 있다.

만약 변경 사항을 작은 단위로 나누어서 커밋해 왔다면, 4b9458b 사이의 커밋들로부터 내용을 복구할 수 있는 지점(goot shot)을 찾을 수도 있을 것이다.

그렇다면 다음과 같이 사라진 객체를 다시 만들어 낼 수 있다:

1. $ git hash-object -w <다시 만든 파일>

이제 저장소는 복구되었다!

(그런데, fsck는 무시하고 다음을 실행하여 문제 해결을 시도해 볼 수 있다.

1. $ git log --raw --all

위 명령의 출력에서 사라진 객체의 SHA1 해시값(4b9458b..)을 찾아볼 수 있다. 이것은 선택하기 나름이다. git는 수 많은 정보를 **가지고 있고**, 이것은 단지 특정한 하나의 사라진 블롭 버전일 뿐이다.

**인덱스**[**#**](http://namhyung.springnote.com/pages/3132772#toc_86)

인덱스는 (일반적으로 .git/index 폴더에 저장되며) 블롭 객체의 이름(path name)으로 정렬된 목록을 포함하는 바이너리 파일이며, 각 블롭 객체의 접근 권한과 SHA-1 해시값도 함께 포함한다. **git-ls-files(1)** 명령을 통해 인덱스의 내용을 볼 수 있다:

1. $ git ls-files --stage  
   100644 63c918c667fa005ff12ad89437f2fdc80926e21c 0       .gitignore  
   100644 5529b198e8d14decbe4ad99db3f7fb632de0439d 0       .mailmap  
   100644 6ff87c4664981e4397625791c8ea3bbb5f2279a3 0       COPYING  
   100644 a37b2152bd26be2c2289e1f57a292534a51a93c7 0       Documentation/.gitignore  
   100644 fbefe9a45b00a54b58d94d06eca48b03d40a50e0 0       Documentation/Makefile  
   ...  
   100644 2511aef8d89ab52be5ec6a5e46236b4b6bcd07ea 0       xdiff/xtypes.h  
   100644 2ade97b2574a9f77e7ae4002a4e07a6a38e46d07 0       xdiff/xutils.c  
   100644 d5de8292e05e7c36c4b68857c1cf9855e3d2f70a 0       xdiff/xutils.h

이전 버전의 문서에서 인덱스는 "현재 디렉터리 캐시" 혹은 단지 "캐시"라고 했던 것을 보았을 지도 모른다. 인덱스는 다음과 같은 세 가지 중요한 특징을 가진다.

1. 인덱스는 하나의 (고유한) 트리 객체를 생성하기 위해 필요한 모든 정보를 포함한다.  
     
   예를 들어 **git-commit(1)** 명령을 실행하면 인덱스로부터 이 트리 객체를 만들고, 객체 데이터베이스에 저장한 뒤, 이를 새로운 커밋에 연관된 트리 객체로 사용한다.
2. 인덱스는 자신이 정의한 트리 객체와 작업 트리를 재빨리 비교할 수 있게 도와준다.  
     
   이것은 각 항목들에 대해 (최종 수정 시간과 같은) 추가적인 정보를 저장하기 때문에 가능하다. 이러한 정보들은 위에서 출력되지 않았고, 생성된 트리 객체 내에 저장되지도 않았지만, 작업 트리 내의 어떤 파일이 인덱스가 저장될 때와 다른지 빨리 판단하는 데 사용될 수 있다. 따라서 git는 변경 사항을 알아내기 위해 이러한 파일들의 모든 데이터를 읽지 않아도 된다.
3. 서로 다른 트리 객체 간의 머지 충돌에 대한 정보를 효율적으로 표현할 수 있다. 각 경로명은 자신이 속한 트리에 대한 충분한 정보를 가질 수 있게 되므로 이들 간에 3-방향 머지를 만들 수 있다.  
     
   "머지 중에 충돌을 해결하기 위해 필요한 정보 얻기" 부분에서, 머지 중에 인덱스는 한 파일의 여러 버전(스테이지라고 한다)을 저장할 수 있다는 것을 살펴보았다. **git-ls-files(1)** 명령의 출력에서 세 번째 열은 이러한 스테이지 번호이며, 머지 충돌이 발생한 파일의 경우에는 0 이외의 값을 가질 것이다.

따라서 인덱스는 작업 중인 트리의 내용을 저장하는 일종의 임시 저장 공간이라고 볼 수 있다.

만약 인덱스를 완전히 제거한다고 해도, 일반적으로 인덱스가 가리키는 트리의 이름이 남아있는 한 어떠한 정보도 사라지지 않을 것이다(?).

**서브 모듈**[**#**](http://namhyung.springnote.com/pages/3132772#toc_87)

대규모 프로젝트들은 때때로 상대적으로 작은 단위의, 완전한 기능을 갖춘 모듈들로 이루어져 있다. 예를 들어 임베디드 리눅스 배포판의 소스 트리는 각각의 수정된 소프트웨어들을 포함할 것이다. 동영상 재생기는 동작이 확인된 특정한 버전의 압축 라이브러리와 빌드될 것이며, 여러 가지 독립적인 프로그램들은 동일한 빌드 스크립트를 공유할 수도 있다.

중앙 집중적인 버전 관리 시스템을 이용한다면 이를 위해 모든 모듈을 하나의 저장소에 포함시킬 것이다. 개발자는 작업하려는 특정 모듈 혹은 전체 모듈을 체크아웃 할 수 있다. 심지어는 파일의 위치를 변경하고, API와 번역을 업데이트하는 동안, 여러 모듈에 걸친 파일들을 하나의 커밋으로 제출할 수도 있다.

git는 일부만 체크아웃 하는 것을 허용하지 않는다. 따라서 위와 같은 방식을 git에서 사용하려면 개발자가 수정하지 않는 모듈들의 로컬 버전도 저장하도록 해야 한다. 대규모의 체크아웃에 커밋하는 일은 예상보다 오래 걸릴 수 있는데, 이는 git가 변경 사항을 위해 모든 디렉토리를 검사해야 하기 때문이다. 만약 모듈이 아주 많은 변경 이력을 가지고 있다면 복사(clone) 작업은 굉장히 오랜 시간이 걸릴 것이다.

장점을 살펴보자면, 분산 버전 관리 시스템은 외부 소스를 보다 쉽게 통합할 수 있다. 중앙 집중적인 방식에서는 외부 프로젝트의 원본 버전 관리 시스템에서 한 스냅샷이 공개되면(export) 그것을 로컬 버전 관리 시스템의 브랜치로 가져온다(import). 이 때 모든 변경 이력은 사라진다. 분산 버전 관리 시스템을 사용하면 전체 변경 이력을 복사할(clone) 수 있으며 보다 쉽게 개발 과정을 따라가거나 로컬의 변경 사항을 다시 머지할 수 있다.

git의 서브모듈 지원은 저장소가 다른 프로젝트의 체크아웃을 하위 디렉터리 형태로 저장할 수 있도록 해 준다. 서브모듈은 자신의 정체성을 유지하며, 서브모듈 지원은 단지 서브모듈의 저장소와 커밋 ID를 저장하여, 서브모듈을 포함한 상위 프로젝트 ("superproject")를 복사한 다른 개발자가 동일한 버전의 모든 서브모듈을 쉽게 복사할 수 있도록 해 준다. 상위 프로젝트의 일부 만을 체크아웃 하는 것이 가능하다. git에게 서브모듈을 전혀 포함하지 않거나, 일부 만 포함하거나, 모두 포함하도록 지정할 수 있다.

git 버전 1.5.3 이후부터는 **git-submodule(1)** 명령을 이용할 수 있다. git 1.5.2 사용자는 저장소 내의 서브모듈 커밋들을 확인하여 직접 체크아웃 할 수 있다. 그 이전 버전들은 서브모듈을 전혀 인식하지 못할 것이다.

서브모듈 지원이 동작하는 방식을 보기 위해, 이후에 서브모듈로 사용될 4 개의 예제 저장소를 만들어 보자:

1. $ mkdir ~/git  
   $ cd ~/git  
   $ for i in a b c d  
   do  
           mkdir $i  
           cd $i  
           git init  
           echo "module $i" > $i.txt  
           git add $i.txt  
           git commit -m "Initial commit, submodule $i"  
           cd ..  
   done

이제 상위 프로젝트를 만들고 모든 서브모듈을 추가한다:

1. $ mkdir super  
   $ cd super  
   $ git init  
   $ for i in a b c d  
   do  
           git submodule add ~/git/$i $i  
   done

**주의**

만약 상위 프로젝트를 공개할 예정이라면, 여기서 서브모듈의 로컬 URL을 사용해서는 안된다.

'git-submodule' 명령이 생성한 파일들을 살펴보자:

1. $ ls -a  
   .  ..  .git  .gitmodules  a  b  c  d

'git-submodule add <저장소> <경로>' 명령은 다음과 같은 작업을 수행한다:

* <저장소>에서 서브모듈을 복사하여 현재 디렉토리 내의 <경로>에 저장한다. 기본적으로 master 브랜치를 체크아웃 한다.
* 서브모듈의 복사 경로를 **gitmodule(5)** 파일에 복사하고 이 파일을 인덱스에 추가하여 커밋할 준비를 한다.
* 서브모듈의 현재 커밋 ID를 인덱스에 추가하여 커밋할 준비를 한다.

상위 프로젝트를 커밋한다:

1. $ git commit -m "Add submodules a, b, c and d."

이제 상위 프로젝트를 복사한다:

1. $ cd ..  
   $ git clone super cloned  
   $ cd cloned

서브모듈 디렉터리는 존재하지만, 비어있다:

1. $ ls -a a  
   .  ..  
   $ git submodule status  
   -d266b9873ad50488163457f025db7cdd9683d88b a  
   -e81d457da15309b4fef4249aba9b50187999670d b  
   -c1536a972b9affea0f16e0680ba87332dc059146 c  
   -d96249ff5d57de5de093e6baff9e0aafa5276a74 d

**주의**

위에 보이는 커밋 객체 이름은 이름은 다를 수 있지만, 이들은 저장소의 HEAD 커밋 객체의 이름과 일치해야 한다. 이것은 'git ls-remote ../a' 명령을 실행하여 확인할 수 있다.

서브모듈을 내려받는 것은 두 단계를 거치는 작업이다. 먼저 'git submodule init' 명령을 실행하여 서브 모듈 저장소 URL을 .git/config 파일에 추가한다:

1. $ git submodule init

이제 'git submodule update' 명령을 실행하여 저장소를 복사하고 상위 프로젝트 내에 지정된 커밋들을 체크아웃 한다:

1. $ git submodule update  
   $ cd a  
   $ ls -a  
   .  ..  .git  a.txt

'git submodule update' 명령과 'git submodule add' 명령의 중요한 차이점 중 하나는, 'git submodule update'는 브랜치의 최신 커밋이 아닌, 지정한 커밋을 체크아웃 한다는 점이다. 이것은 태그를 체크아웃 하는 것과 비슷한데, 헤드가 분리되어 브랜치에서 작업하는 것이 아니다.

1. $ git branch  
   \* (no branch)  
     master

만약 서브모듈 내에서 수정을 하고 싶은데 헤드가 분리되어 있다면, 브랜치를 만들거나 체크아웃 한 후, 소스를 수정하고, 서브모듈 내에서 변경 사항을 공개하고, 상위 프로젝트가 새로운 커밋을 참조하도록 업데이트한다:

1. $ git checkout master

혹은

1. $ git checkout -b fix-up

그리고 다음을 수행한다:

1. $ echo "adding a line again" >> a.txt  
   $ git commit -a -m "Updated the submodule from within the superproject."  
   $ git push  
   $ cd ..  
   $ git diff  
   diff --git a/a b/a  
   index d266b98..261dfac 160000  
   --- a/a  
   +++ b/a  
   @@ -1 +1 @@  
   -Subproject commit d266b9873ad50488163457f025db7cdd9683d88b  
   +Subproject commit 261dfac35cb99d380eb966e102c1197139f7fa24  
   $ git add a  
   $ git commit -m "Updated submodule a."  
   $ git push

만약 'git pull' 명령을 실행한 후에 서브모듈도 업데이트 하고 싶다면 'git submodule update' 명령을 실행해야 한다.

**서브모듈에 관한 함정**[**#**](http://namhyung.springnote.com/pages/3132772#toc_88)

서브모듈을 포함하는 상위 프로젝트에 변경 사항을 공개하기(publish) 전에 항상 서브모듈의 변경 사항을 먼저 공개해야 한다. 만약 서브모듈의 변경 사항을 공개하는 것을 잊어버렸다면 다른 사람들이 저장소를 복사할 수 없게 될 것이다:

1. $ cd ~/git/super/a  
   $ echo i added another line to this file >> a.txt  
   $ git commit -a -m "doing it wrong this time"  
   $ cd ..  
   $ git add a  
   $ git commit -m "Updated submodule a again."  
   $ git push  
   $ cd ~/git/cloned  
   $ git pull  
   $ git submodule update  
   error: pathspec '261dfac35cb99d380eb966e102c1197139f7fa24' did not match any file(s) known to git.  
   Did you forget to 'git add'?  
   Unable to checkout '261dfac35cb99d380eb966e102c1197139f7fa24' in submodule path 'a'

또한 상위 프로젝트에 저장된 커밋 이후에 서브모듈 내의 브랜치를 되돌려서는(rewind) 안 된다.

브랜치를 체크아웃 하지 않은 채로 서브모듈 내의 변경 사항을 커밋했다면, 'git submodule update' 명령을 실행하는 것이 안전하지 않다. 이러한 사항들은 조용히 덮어씌워질 것이다.:

1. $ cat a.txt  
   module a  
   $ echo line added from private2 >> a.txt  
   $ git commit -a -m "line added inside private2"  
   $ cd ..  
   $ git submodule update  
   Submodule path 'a': checked out 'd266b9873ad50488163457f025db7cdd9683d88b'  
   $ cd a  
   $ cat a.txt  
   module a

**주의**

이러한 변경 사항들은 서브모듈의 reflog를 통해 여전히 볼 수 있다.

이것은 커밋하지 않은 경우에는 적용되지 않는다.

**저수준 git 연산**[**#**](http://namhyung.springnote.com/pages/3132772#toc_89)

 많은 수의 고수준 명령들은 원래 작고 핵심적인 저수준 git 명령들을 이용한 쉘 스크립트로 구현되었다. 이러한 저수준 명령들은 git로 일상적이지 않은 작업을 수행하거나, git의 내부 동작을 이해하는 데 유용하다.

**객체 접근 및 조작**[**#**](http://namhyung.springnote.com/pages/3132772#toc_90)

**git-cat-file(1)** 명령은 어떠한 객체의 내용도 표시할 수 있다. 하지만 고수준의 **git-show(1)** 명령이 보통 더 유용하게 사용된다.

**git-commit-tree(1)** 명령은 임의의 부모와 트리들로 커밋을 구성하도록 해 준다.

트리는 **git-write-tree(1)** 명령으로 만들 수 있으며, 트리 데이터는 **git-ls-tree(1)** 명령으로 볼 수 있다. 두 트리를 비교할 때는 **git-diff-tree(1)** 명령을 이용할 수 있다.

태그는 **git-mktag(1)** 명령으로 만들고, 서명은 **git-verify-tag(1)** 명령으로 확인할 수 있다. 하지만 일반적으로 두 경우 모두 **git-tag(1)** 명령을 사용하는 것이 더 간편하다.

**작업 순서**[**#**](http://namhyung.springnote.com/pages/3132772#toc_91)

**git-commit(1)**, **git-checkout(1)**, **git-reset(1)**과 같은 고수준 명령들은 데이터를 작업 트리, 인덱스, 객체 데이터베이스로 이동시키며 작업한다. git는 이러한 각 단계를 개별적으로 수행하는 저수준의 연산을 제공한다.

일반적으로 모든 "git" 연산들은 인덱스 파일 상에서 이루어진다. 몇몇 연산들은 **순전히** 인덱스 파일 상에서만 동작하지만 (인덱스의 현재 상태를 보여주는 연산), 많은 연산들은 데이터를 인덱스 파일과 객체 데이터베이스 혹은 작업 트리 사이로 이동시킨다. 따라서 다음과 같은 4가지 조합이 존재한다:

**작업 디렉터리 -> 인덱스**[**#**](http://namhyung.springnote.com/pages/3132772#toc_92)

**git-update-index(1)** 명령은 작업 디렉터리의 정보로부터 인덱스를 갱신한다. 일반적으로 다음과 같이 갱신하고자 하는 파일의 이름을 지정하기만 하면 인덱스 정보를 갱신한다:

1. $ git update-index filename

하지만 파일 이름 글로빙(globbing - 역주: \*, ? 기호 등을 이용한 여러 파일 선택) 등으로 인한 일반적인 실수를 방지하기 위해, 이 명령은 일반적으로 새로운 항목을 추가하거나 기존의 항목을 삭제하는 작업을 수행하지 않는다. 즉, 일반적으로 기존의 캐시 항목을 업데이트하는 작업 만을 수행할 것이다.

이를 수행하려면, 여러분이 어떤 파일들은 더 이상 존재하지 않거나 새로운 파일들을 추가해야 한다는 것을 제대로 인식하고 있다고 git에게 알려주기 위해, 각각 --remove 플래그와 --add 플래그를 사용해야 한다.

주의! --remove 플래그는 지정한 파일이 삭제되어야 한다는 것을 의미하지 않는다. 만약 해당 파일이 계속 디렉터리 구조 내에 남아있다면, 인덱스는 제거되지 않고 새로운 상태로 갱신될 것이다. --remove 플래그가 의미하는 유일한 내용은 update-index 명령이 삭제된 파일을 올바른 것으로 인식하도록 하며, 만약 해당 파일이 더 이상 존재하지 않으면 그에 따라 인덱스를 적절히 갱신하는 것이다.

특별한 경우로, 각 인덱스의 현재 "상태"(stat) 정보를 현재 상태 정보와 일치하도록 갱신하는 'git update-index --refresh' 명령을 이용할 수 있다. 이 명령은 객체 상태 자체를 갱신하는 것이 아니라, 객체가 이전에 저장된 객체와 일치하는지 빨리 알아보는 데 사용하는 필드만을 갱신할 것이다.

앞서 설명한 **git-add(1)** 명령은 **git-update-index(1)** 명령의 래퍼(wrapper)일 뿐이다.

**인덱스 -> 객체 데이터베이스**[**#**](http://namhyung.springnote.com/pages/3132772#toc_93)

 다음 명령을 이용하여 현재 인덱스 파일을 "트리" 객체에 기록한다.

1. $ git write-tree

위 명령은 아무런 인자도 받지 않으며, 현재 인덱스를 해당 상태가 지정하는 트리 객체 집합에 쓰고, 그 결과로 나온 최상위 트리 객체의 이름을 반환할 것이다. 이 객체는 언제라도 다른 방향으로 이동하여 인덱스를 새로 생성하는 데 사용할 수 있다. (아래 참고)

**객체 데이터베이스 -> 인덱스**[**#**](http://namhyung.springnote.com/pages/3132772#toc_94)

객체 데이터베이스에서 "트리" 파일을 읽어서, 현재 인덱스를 채운다. (그리고 덮어쓴다 - 만약 인덱스가 저장되지 않은 상태를 포함하고 있으며, 나중에 이 상태로 돌아가고 싶다면 이 작업을 수행해서는 안 된다.) 일반적인 작업 과정은 다음과 같다.

1. $ git read-tree <트리의 SHA-1 해시값>

위 명령은 인덱스 파일을 이전에 저장했던 트리와 동일하게 만들 것이다. 하지만 이것은 오직 **인덱스** 파일 만을 수정하는 것이며 작업 디렉토리의 내용은 변경되지 않는다.

**인덱스 -> 작업 디렉토리**[**#**](http://namhyung.springnote.com/pages/3132772#toc_95)

파일들을 "체크아웃" 하여 인덱스로부터 작업 디렉터리를 갱신할 수 있다. 이것은 아주 일상적인 연산은 아니다. 보통은 파일을 최신 상태로 유지하기를 원하며, 작업 디렉터리에 쓰기 보다는 작업 디렉터리의 변경 사항을 인덱스에 저장하기 때문이다. (git update-index)

하지만 새로운 버전으로 변경(jump)하거나, 다른 사람의 버전을 체크아웃하거나, 이전 트리를 복구하기로 결정했다면, read-tree 명령을 통해 인덱스 파일을 채우고(populate), 그 결과를 체크아웃해야 한다.

1. $ git checkout-index filename

혹은, 모든 인덱스를 체크아웃하려면 -a 플래그를 이용한다.

주의! 'git checkout-index' 명령은 일반적으로 이전 버전의 파일을 덮어쓰지 않을 것이다. 따라서 만약 이전 버전의 트리를 이미 체크아웃했다면, **강제로** 체크아웃을 수행하기 위해 ("-a" 플래그나 파일 이름 **앞에**) "-f" 플래그를 사용해야 한다.

마지막으로, 한 형태(representation)에서 다른 형태로 완전히(purely) 이동하지 않는 몇 가지 것(a few odds and ends)들이 존재한다.

**이들을 모두 합치기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_96)

"git write-tree" 명령을 통해 생성한 트리를 커밋하려면, 해당 트리와 그에 대한 변경 이력(대부분 변경 이력 상의 바로 이전에 위치하는 "부모" 커밋을 말한다)을 가리키는 "커밋" 객체를 만들어야 한다.

일반적으로 "커밋" 객체는 하나의 부모 커밋을 가진다. 이 부모 커밋은 트리에서 특정한 수정이 이루어지기 전의 상태를 가리킨다. 하지만, 우리가 "머지"라고 하는 경우에 있어서는 둘 이상의 부모 커밋을 가질 수 있다. 이는 이러한 커밋이 다른 커밋들이 가리키는 둘 이상의 이전 상태가 합쳐져서(머지되어) 이루어진 것이기 때문이다.

다시 말하면, "트리"는 작업 디렉터리의 특정한 디렉터리 상태를 말하지만, "커밋"은 특정한 "시간"에서의 상태와 그 상태에 이르기까지의 과정을 나타낸다.

커밋이 이루어지는 시각에서의 상태를 나타내는 트리와 부모 커밋의 목록을 이용하여 다음과 같이 커밋 객체를 만들 수 있다:

1. $ git commit-tree <tree> -p <parent> [-p <parent2> ..]

그리고 표준 입력을 통해 커밋에 대한 설명을 추가한다. (파이프나 파일 재지정(redirection)을 이용하거나 터미널에 직접 입력하면 된다.)

git commit-tree 명령은 생성된 커밋을 나타내는 객체의 이름을 반환하며, 나중을 위해 이를 저장해 두어야 한다. 일반적으로 새로운 HEAD 상태를 커밋하는 데, git는 이 상태에 대한 정보를 저장하는 위치를 고려하지 않기 때문에 실제로는 .git/HEAD가 가리키는 파일에 객체의 이름을 직접 저장해야 하며, 그렇게 되면 마지막 커밋 상태가 무엇인지 항상 확인할 수 있다.

아래는 여러 과정들이 함께 동작하는 과정을 설명하기 위해 Jon Loeliger가 그린 ASCII art이다.

1. commit-tree  
                         commit obj  
                          +----+  
                          |    |  
                          |    |  
                          V    V  
                       +-----------+  
                       | Object DB |  
                       |  Backing  |  
                       |   Store   |  
                       +-----------+  
                          ^  
              write-tree  |     |  
                tree obj  |     |  
                          |     |  read-tree  
                          |     |  tree obj  
                                V  
                       +-----------+  
                       |   Index   |  
                       |  "cache"  |  
                       +-----------+  
            update-index  ^  
                blob obj  |     |  
                          |     |  
       checkout-index -u  |     |  checkout-index  
                stat      |     |  blob obj  
                                V  
                       +-----------+  
                       |  Working  |  
                       | Directory |  
                       +-----------+

**데이터 살펴보기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_97)

여러 가지 보조 도구를 이용하여 객체 데이터베이스 내에 표현된 데이터나 인덱스를 살펴볼 수 있다. 모든 객체들은 **git-cat-file(1)** 명령을 이용하여 해당 객체에 대한 상세한 내용을 살펴볼 수 있다:

1. $ git cat-file -t <객체 이름>

위 명령은 객체의 타입 정보를 보여준다. 객체 타입을 알고나면 (일반적으로 객체 타입 정보는 객체를 찾을 때는 표시되지 않는다) 다음 명령을 이용할 수 있다.

1. $ git cat-file blob|tree|commit|tag <objectname>

위 명령은 객체의 내용을 보여준다. 주의! 트리 객체는 이진 데이터를 포함한다. 따라서 트리 객체의 내용을 표시하기 위한 git-ls-tree라는 특별한 명령이 별도로 존재하며, 이는 이진 데이터를 읽기 쉬운 형식으로 변환해 준다.

"커밋" 객체를 살펴보는 것은 특히 도움이 된다. 커밋 객체는 보통 크기가 작으며 자체적으로 충분한 정보를 제공하기 때문이다. 특히 최종 커밋의 이름을 .git/HEAD가 가리키는 파일에 저장하는 관례를 따르는 (일반적인) 경우라면, 다음 명령을 이용하여 최종 커밋의 내용을 확인할 수 있다.

1. $ git cat-file commit HEAD

**여러 트리를 머지하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_98)

git는 3방향 머지를 수행할 수 있도록 도와주며, 최종적으로 "커밋"할 상태에 이르기까지 이 작업을 반복함으로서 n방향 머지를 수행할 수도 있다. 일반적인 상황은 (2개의 부모가 있는) 3방향 머지를 한 번 수행하고 커밋하는 것이지만, 원한다면 여러 부모를 머지한 후 한 번에 커밋할 수 있다.

3방향 머지를 수행하려면, 머지할 두 개의 "커밋" 객체들이 있어야 하며, 이들을 이용하여 가장 가까운 공통 부모(세번째 "커밋" 객체)를 찾고, 이러한 커밋 객체들을 이용하여 각 지점에서의 디렉터리의 상태("트리" 객체)를 찾는다.

머지의 "베이스"를 얻으려면, 먼저 다음 명령을 이용하여 두 커밋의 공통 부모를 찾는다:

1. $ git merge-base <커밋1> <커밋2>

위 명령은 두 커밋들이 공통적으로 참조하는 커밋을 반환할 것이다. 이제 각 커밋들의 "트리" 객체를 살펴봐야 한다. 이는 (예를 들어) 다음 명령을 이용하여 간단히 수행할 수 있다.

1. $ git cat-file commit <커밋 이름> | head -1

트리 객체 정보는 항상 커밋 객체의 첫 번째 줄에 나타나기 때문이다.

머지를 수행할 3개의 트리를 얻고 나면 (하나의 "원본" 트리(또는 공통 트리)와 두 개의 "결과" 트리(머지를 수행할 브랜치)) 인덱스로 "머지" 읽기를 수행한다. 이 과정은 이전 인덱스의 내용을 덮어쓰게 되는 경우 경고를 보여줄 것이다. 따라서 인덱스의 내용을 모두 커밋했는지 확인해야 한다. 실제로는 거의 항상 최신 커밋에 대해 머지를 수행할 것이므로 현재 인덱스의 내용과 일치할 것이다.

머지를 수행하려면 다음 명령을 실행한다.

1. $ git read-tree -m -u <원본 트리> <여러분의 작업 트리> <대상 트리>

위 명령은 인덱스 파일 내에서 직접 여러 간단한 머지 작업들을 수행할 것이며, 그 결과를 git write-tree 명령을 통해 기록할 수 있다.

**여러 트리를 머지하기 (계속)**[**#**](http://namhyung.springnote.com/pages/3132772#toc_99)

슬프게도 많은 머지 작업은 단순하지가 않다. 만약 추가, 이동, 삭제된 파일이 있거나 두 브랜치에서 모두 같은 파일을 수정했다면 "머지 항목"들을 포함한 인덱스 트리를 가지게 될 것이다. 이러한 인덱스 트리는 트리 객체로 쓸 수 **없으며**, 결과를 기록하기 전에 다른 도구를 이용하여 머지 충돌 사항들을 해결해야 한다.

git ls-files --unmerged 명령을 이용하여 이러한 인덱스 상태를 살펴볼 수 있다. 예를 들면:

1. $ git read-tree -m $orig HEAD $target  
   $ git ls-files --unmerged  
   100644 263414f423d0e4d70dae8fe53fa34614ff3e2860 1       hello.c  
   100644 06fa6a24256dc7e560efa5687fa84b51f0263c3a 2       hello.c  
   100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello.c

git ls-files --unmerged 명령의 출력의 각 줄은 블롭 모드 비트, 블롭의 SHA1 해시값, 스테이지 번호, 파일 이름으로 이루어진다. 스테이지 번호는 git에서 해당 객체가 어느 트리에서 온 것인지 표현하기 위한 방법이다. 스테이지 1, 2, 3은 각각 $orig, HEAD, $target 트리에 해당한다.

앞에서 간단한 머지 작업은 git-read-tree -m 명령을 통해 수행된다고 하였다. 예를 들어 어떤 파일이 $orig에서부터 HEAD나 $target까지 변경되지 않았거나, 동일한 방식으로 변경되었다면 당연히 최종 결과는 HEAD의 내용이 된다. 위의 예제에서는 hello.c 파일이 $orig부터 HEAD까지와 $orig부터 $target까지 다른 방식으로 변경되었다는 것을 보여준다. 이것은 각 스테이지의 블롭 객체들에 대해 직접 여러분이 주로 사용하는 3방향 머지 프로그램을 (예를 들어, diff3, merge 혹은 git merge-file) 이용하여 다음과 같이 해결할 수 있다:

1. $ git cat-file blob 263414f... >hello.c~1  
   $ git cat-file blob 06fa6a2... >hello.c~2  
   $ git cat-file blob cc44c73... >hello.c~3  
   $ git merge-file hello.c~2 hello.c~1 hello.c~3

위 명령은 머지 결과를 hello.c~2 파일에 저장하며, 만약 충돌이 발생했다면 충돌 표시를 남긴다. 머지 결과가 올바른지 확인한 후에는, git에게 해당 파일의 최종 머지 결과를 알려줄 수 있다:

1. $ mv -f hello.c~2 hello.c  
   $ git update-index hello.c

만약 경로가 "unmerged" 상태에 있었다면, git-update-index 명령은 git에게 해당 경로의 충돌이 해결되었다고 표시하도록 알려준다.

지금까지 내부에서 개념적으로 어떠한 일들이 수행되는지 이해하기 위해 git의 머지 과정을 가장 낮은 수준에서 설명하였다. 실제로는 (git를 자체를 포함하여) 아무도 이를 위해 git cat-file 명령을 3번 수행하지 않는다. 임시 파일에서 각 스테이지를 추출하기 위한 git-merge-index 명령이 존재하며, 이를 이용하여 "merge" 스크립트를 호출한다:

1. $ git merge-index git-merge-one-file hello.c

이것은 상위 수준의 git-merge -s resolve 명령이 구현된 방식이다.

**Git 해킹하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_100)

이 부분에서는 아마도 git 개발자들 만이 이해할 필요가 있는 git 내부 구현의 세부 사항들을 설명한다.

**객체 저장 형식**[**#**](http://namhyung.springnote.com/pages/3132772#toc_101)

모든 객체는 해당 객체의 형식을 나타내는 고정된 "타입"을 가지고 있다. 이 타입은 해당 객체가 어떻게 사용되며, 어떻게 다른 객체를 참조하는 지를 결정한다. 현재는 다음과 같은 4가지 타입이 존재한다: "블롭", "트리", "커밋", "태그"

객체의 타입과는 별도로 모든 객체는 다음과 같은 공통적인 특징을 가진다. 모든 객체는 zlib으로 압축되며, 객체의 형식과 객체 내의 데이터의 크기 정보를 제공하는 헤더를 포함한다. 객체의 이름으로 사용되는 SHA-1 해시값은 원본 데이터와 이 헤더를 합친 데이터에 대한 해시값이므로, 해당 파일에 sha1sum 명령을 수행한 값과 객체 이름은 다르다는 것을 알아두기 바란다. (역사적인 정보: git 초창기의 객체 이름은 압축된 객체의 SHA1 해시값을 사용하였다.)

따라서, 일반적인 객체의 일관성은 객체의 종류 및 그 내용과는 무관하게 항상 테스트할 수 있다. 객체의 일관성을 검사하려면 (a) 해시값이 객체의 내용과 일치하는지 (b) 객체의 압축이 <타입을 나타내는 ASCII 문자열 (공백없음)> + <공백> + <크기를 나타내는 ASCII 문자열 (10진수)> + <바이트 \0> + <바이너리 객체 데이터> 형식으로 잘 해제되는지 확인하면 된다.

구조화된 객체는 추가로 자신의 구조를 가질 수 있으며, 다른 객체와의 연결도 확인해야 한다. 이는 일반적으로 git-fsck 프로그램을 이용하여 확인할 수 있다. git-fsck는 모든 객체의 완전한 의존 그래프를 생성하고 (해사값을 통한 외부 일관성을 확인하는 것과 별도로) 각각의 내부 일관성을 검사한다.

**git 소스 코드의 개략적인 설명**[**#**](http://namhyung.springnote.com/pages/3132772#toc_102)

새로운 개발자가 git 소스 코드에서 적절한 방향을 찾는 것이 항상 쉽지는 않다. 이 부분에서는 git 소스를 살펴보는데 도움이 되는 약간의 조언을 하려한다.

가장 좋은 출발점은 다음과 같은 명령을 통해 최초 커밋의 내용을 살펴보는 것이다:

1. $ git checkout e83c5163

최초의 버전은 현재 git가 가진 거의 대부분의 기능을 기본적으로 구현하고 있지만 한 번에 살펴볼 수 있을 만큼 적은 양이다.

용어는 계속 변경되어 왔다. 예를 들어 최초 버전의 README 파일에 포함된 "changeset"이라는 용어는 지금의 커밋에 해당한다.

또 한 현재는 "캐시"라는 용어는 사용하지 않고 "인덱스"라는 용어를 사용한다. 하지만 파일 이름은 그대로 cache.h로 남아있다. 참고: 현재 이를 변경할 큰 이유는 없다. 특히 이를 대체할 단일 용어가 마땅치 않으며, 이 파일은 git 내의 모든 C 소스 파일에서 include하고 있는 헤더 파일이기 때문이다.

만약 최초 버전내의 아이디어들을 이해했다면, 좀 더 최신 버전을 체크아웃하여 cache.h, object.h, commit.h 파일들을 살펴보기 바란다.

초 기의 git는 (유닉스의 전통에 따라) 스크립트 내에서 파이프를 이용하여 한 프로그램의 출력을 다른 프로그램으로 전달하는 아주 작은 프로그램들의 집합이었다. 이 방식은 새로운 기능을 테스트하기에 편하기 때문에 초기 개발 형태로 적당하다. 하지만 최근에는 이러한 많은 부분들이 내장되었고 (builtin), 핵심 기능 중 일부는 라이브러리 형태로 구성되었다. (libified) 즉 성능, 이식성, 코드 중복 방지 등을 위해 libgit.a 라이브러리 내에 포함되었다.

지금까지 인덱스(및 cache.h 내의 관련된 자료 구조)에 대해 살펴보았고, struct object에서 공통적인 구조를 상속받은 (즉 구조체의 첫 멤버가 struct object인) 여러 객체 타입들(블롭, 트리, 커밋, 태그)이 있다는 것을 알게되었다. (따라서 (struct object \*) commit와 같은 캐스트 연산은 &commit --> object와 동일하며 이를 통해 객체 이름과 플래그 등을 얻을 수 있다.)

이제 이러한 정보들을 머리 속에 담아두기 위해 잠시 휴식을 취하는 것이 좋겠다.

다음 단계: 객체 이름에 익숙해지기. 커밋 이름 붙이기 부분을 읽어보자. 객체(리비전 만이 아니다!)에 이름을 붙이는 몇 가지 방법이 존재한다. 이 모든 방법들은 sha1\_name.c 파일 내에서 처리한다. get\_sha1() 함수를 대강 살펴보자. 대부분의 특별한 처리는 get\_sha1\_basic() 및 이와 비슷한 함수들이 수행한다.

이제 git의 기능 중에서 가장 라이브러리화 된 부분인 리비전 탐색기(revision walker)를 사용하는 법을 살펴보기로 하자.

기본적으로 초기 버전의 git-log 명령은 다음과 같은 쉘 스크립트였다:

1. $ git-rev-list --pretty $(git-rev-parse --default HEAD "$@") | \  
           LESS=-S ${PAGER:-less}

이것은 어떤 의미일까?

git-rev-list 명령은 리비전 탐색기의 원래 버전으로, 항상 리비전 목록을 표준 출력으로 내보낸다. 이 명령은 아직도 동작하며, 필요하다. 왜냐하면 대부분의 새로운 git 프로그램들은 git-rev-list를 사용하는 스크립트로부터 시작하기 때문이다.

git-rev-parse 명령은 더 이상 그리 중요하지 않다. 이 명령은 여러 스크립트에서 사용하는 서로 다른 명령들과 관련된 옵션을 필터링 하기 위해서만 사용되었다.

git-rev-list 명령에서 수행하던 대부분의 작업은 revision.c와 revision.h 파일에 포함되었다. 여러 옵션들은 리비전 탐색 방법 및 어떤 리비전을 탐색할 지 등에 대한 정보를 포함하는 rev\_info 구조체 내에 포함된다.

원 래 git-rev-parse에서 수행하던 작업은 이제 리비전 정보와 리비전 탐색기에 관련된 공통 명령행 옵션들은 분석하는 setup\_revision() 함수에서 처리한다. 이 정보는 나중에 참조하기 위해 rev\_info 구조체에 저장한다. setup\_revision() 함수를 호출한 뒤에 자신만의 명령행 옵션 분석을 수행할 수도 있다. 그 후에는 초기화를 수행하는 prepare\_revision\_walk() 함수를 호출해야 하며, 그리고 get\_revision() 함수를 통해 각 커밋을 하나씩 얻을 수 있다.

만약 리비전 탐색 과정을 좀 더 자세하게 알아보고 싶다면 cmd\_log() 함수의 최초 구현을 살펴보는 것이 좋다. 이를 살펴보려면 git show v1.3.0~155^2~4 명령을 수행하고 해당 함수를 찾아본다. (더 이상 setup\_pager() 함수를 직접 호출할 필요가 없다는 것에 주의하라.)

최근에는 git-log 명령은 빌트인 명령으로 구현된다. 즉 git 명령 내에 포함되었다. 빌트인 명령은 소스에서 다음과 같이 존재한다:

* cmd\_<bla>라는 함수는 일반적으로 builtin-<bla>.c 파일 내에서 정의하며, builtin.h 파일에서 선언한다.
* git.c 파일 내의 commands[] 배열 내에 해당 항목을 포함한다.
* Makefile 내의 BUILTIN\_OBJECTS 내에 해당 항목을 포함한다.

때 때로, 둘 이상의 빌트인 명령이 한 소스 파일 내에 포함되기도 한다. 예를 들어 cmd\_whatchanged() 함수와 cmd\_log() 함수는 모두 builtin-log.c 파일 내에 구현되어 있는데, 이는 이들이 많은 코드를 공유하기 때문이다. 이 경우 c 파일과 이름이 다른 명령은 Makefile 내의 BUILT\_INS 내에 포함되어야 한다.

git-log 명령은 원래 스크립트로 구현되었을 때 보다 C로 구현한 버전이 더 복잡해 보이지만, 이는 보다 높은 유연성과 성능을 가져다 준다.

여기에서 다시 한 번 쉬어가는 기회를 가지는 것이 좋겠다.

세 번째 레슨은 코드 연구하기이다. 실제로 이것은 (기본 개념을 이해한 이후에) git가 어떻게 구성되어 있는지 파악할 수 있는 가장 좋은 방법이다.

따 라서, 여러분이 흥미를 가지고 있는 부분에 대해서 생각해 보자. 이를테면 "어떻게 객체 이름 만 가지고 블롭에 접근할 수 있을까?"와 같은 부분이다. 먼저 해야할 일은 해당 작업을 수행할 수 있는 git 명령을 찾는 것이다. 이 경우는 git-show 명령과 git-cat-file 명령이 모두 가능하다.

확실히 하기 위해 여기서는 git-cat-file의 경우를 살펴볼 것이다. 왜냐하면

git-cat-file은 plumbing하다(???)

git-cat-file은 최초 커밋 시에도 존재하였다. (cat-file.c의 형태로 약 20번 정도 수정되었으며, 빌트인이 도입된 후 builtin-cat-file.c 파일로 변경된 후에는 10번 이하로 수정되었다.)

따라서 builtin-cat-file.c 파일에서 cmd\_cat\_file() 함수를 찾아서 내용을 살펴보자.

1. git\_config(git\_default\_config);  
           if (argc != 3)  
                   usage("git-cat-file [-t|-s|-e|-p|<type>] <sha1>");  
           if (get\_sha1(argv[2], sha1))  
                   die("Not a valid object name %s", argv[2]);

뻔 한 내용들은 건너뛰기로 한다. 여기서 우리가 관심을 가지는 부분은 오직 get\_sha1() 함수를 호출하는 부분이다. 여기서는 argv[2]를 객체 이름으로 해석하려고 시도하며, 만약 이 이름이 현재 디렉터리 내에 존재하는 객체를 가리킨다면 sha1 변수에 객체의 SHA1 해시값을 저장한다.

다음과 같은 두 가지 사항을 눈여겨 보자.

* get\_sha1() 함수는 성공 시 0을 반환한다. 어쩌한 이것은 몇몇 새로운 git 해커들에게는 놀라울지도 모르지만, 유닉스에는 여러 에러 상황의 경우 서로 다른 음수를 반환하고 성공 시 0을 반환하는 오랜 전통이 있다.
* get\_sha1() 함수의 원형에서 sha1 변수의 타입은 unsinged char \*이지만, 실제로는 unsigned char[20]에 대한 포인터라고 가정한다. 이 변수는 주어진 커밋에 대한 160비트 크기의 SHA-1 해시값을 저장할 것이다. 주의할 점은 SHA-1 해시값이 unsigned char \* 타입으로 넘겨지면 이는 바이너리 표현이며, char \* 타입으로 넘겨질 때는 16진수 문자에 대한 ASCII 표현이라는 것이다.

코드를 살펴보다보면 이 두 가지 경우를 모두 보게될 것이다.

이제 핵심 부분이다:

1. case 0:  
                   buf = read\_object\_with\_reference(sha1, argv[1], &size, NULL);

위 의 코드가 블롭(실제로는 블롭 만이 아니라 모든 타입의 객체에 해당한다)을 읽는 부분이다. read\_object\_with\_reference() 함수가 실제로 어떻게 동작하는지 알아보고 싶다면, 소스 코드에서 그 부분을 찾아서 (git 저장소에서 git grep read\_object\_with | grep ":[a-z]"와 같이 실행해보자) 읽어보면 된다.

이 결과를 사용하는 방법은 cmd\_cat\_file() 함수를 계속 읽어보면 다음과 같이 나온다:

1. write\_or\_die(1, buf, size);

어떤 때는 특정한 기능을 어디서 찾아야 할 지 모를 때가 있다. 이러한 때 중 많은 경우는 git log 명령의 출력을 찾아본 뒤 해당 커밋에 대해 git show 명령을 수행해 보는 것이 도움이 된다.

예 제: 여러분은 git-bundle 명령의 테스트 케이스가 있다는 것을 알고 있지만 어디에 있는지는 기억하지 못한다고 가정해 보자. (그렇다. git grep bundle t/ 명령을 이용하여 이를 알아낼 수 있지만, 이는 지금 여기서 얘기하고자 하는 바가 아니다.):

1. $ git log --no-merges t/

페이저 (less) 상에서 "bundle"을 검색하고 몇 줄 뒤로 가보면 이것이 18449ab0... 커밋 내에 포함되어 있다는 것을 알 수 있다. 이제 객체 이름을 복사해서 다음과 같이 명령행에 붙여넣는다:

1. $ git show 18449ab0

완료.

또 다른 예제: 스크립트를 빌트인으로 만들기 위해 해야하는 일들을 알아 보자:

1. $ git log --no-merges --diff-filter=A builtin-\*.c

여기서 보았듯이 git는 git 소스 자체를 살펴보기 위한 최고의 도구이다.

**Git 용어**[**#**](http://namhyung.springnote.com/pages/3132772#toc_103)

*alternate object database (대리 객체 데이터베이스)*

대리 (alternate) 메커니즘을 통하여, 저장소는 "대리" 객체 데이터베이스라고 하는 다른 객체 데이터베이스에서 객체 데이터베이스의 일부를 상속받을 수 있다.

*bare repository (bare 저장소)*

bare 저장소는 일반적으로 .git라는 접미사가 붙은 디렉터리로 버전 컨트롤 중인 어떤 파일도 체크아웃하지 않은 상태를 말한다. 즉 git에서 사용하는 모든 관리용 및 제어용 파일들은 repository.git 디렉터리 바로 아래의 .git라는 숨겨진 디렉터리 내부에 존재하며, 다른 파일들은 존재하지 않는다. 보통 공개 저장소를 관리하는 사람들은 bare 저장소 형태로 공개한다.

*blob object (블롭 객체)*

타입이 없는(untyped) 객체로, 파일의 내용과 같은 것이다.

*branch (브랜치)*

"브랜치"는 개발이 활발하게 진행되는 노선(active line of development)이다. 브랜치의 가장 최근 커밋은 해당 브랜치의 팁(tip)이라고 부른다. 브랜치의 팁은 브랜치의 헤드가 참조한다. 브랜치의 헤드는 해당 브랜치에 추가적이 개발이 진행되면 앞으로 옮겨진다. 하나의 git 저장소는 여러 브랜치를 관리(track)할 수 있지만, 작업 트리는 그 중 하나에만 연관되며 (이를 :현재" 브랜치 혹은 "체크아웃한" 브랜치라고 한다) HEAD는 해당 브랜치를 가리킨다.

*cache (캐시)*

더 이상 사용하지 않는다. 대신 index라는 용어를 사용한다.

*chain (체인)*

객체의 목록(list)이다. 목록 내의 각 객체는 다음 객체(successor)에 대한 참조를 포함한다. (예를 들어 어떤 커밋의 다음 객체는 그부모 중 하나가 될 수 있다.)

*changeset (변경 세트)*

BitKeeper/cvsps에서 "커밋"을 지칭하는 말이다. git는 변경 사항이 아닌, 상태를 저장하기 때문에 "변경 세트"라는 표현은 git에서는 올바르지 않다.

*checkout (체크아웃)*

작업 트리 전체 혹은 일부를 객체 데이터베이스에서 가져온 트리 객체 또는 블롭으로 갱신(update)하는 것을 말한다. 또한 인덱스를 업데이트하며 만약 작업 트리가 새로운 브랜치를 가리키게 되는 경우에는 HEAD도 갱신한다.

*cherry-picking (체리피킹)*

SCM 용어에서 "체리피킹"이라는 말은 일련의 변경 사항 (일반적으로 커밋) 중에서 일부를 선택하여 별도의 코드 베이스 상에 새로운 일련의 변경 사항으로 기록한다는 것을 뜻한다. GIT에서 이것은 "git cherry-pick" 명령을 통해 수행하며 기존의 커밋에서 변경한 내용들 중 일부를 추출하여 현재 브랜치의 팁에 새로운 커밋으로 기록한다.

*clean (깨끗하다)*

작업 트리가 깨끗하다는 것은 현재 헤드가 참조하는 리비전과 작업 트리가 같다는 것을 뜻한다. dirty의 설명도 참고하기 바란다.

*commit (커밋)*

명사로 쓰일 때: git 변경 이력 내의 한 지점. 프로젝트의 전체 변경 이력은 관련된 커밋들의 집합으로 표현한다. git에서 "커밋"이라는 용어는 때때로 다른 버전 관리 시스템에서 사용하는 "리비전" 또는 "버전"이라는 용어와 같은 의미로 사용된다. 또한 커밋 객체의 줄임말로도 쓰인다.

동사로 쓰일 때: 프로젝트 상태의 새로운 스냅샷을 git 저장소 내에 저장하는 작업을 뜻한다. 이는 인덱스의 현재 상태를 나타내는 새로운 커밋을 만들고 HEAD가 이 커밋을 가리키도록 이동시키는 것이다.

*commit object (커밋 객체)*

특정 리비전을 나타내는 정보, 이를테면 부모, 커밋한 사람, 수정한 사람, 날짜 및 저장된 리비전의 최상위 디렉터리에 대응하는 트리 객체들을 포함하는 객체이다.

*core git (코어 git)*

git의 기본 자료 구조와 도구들. 제한된 소스 코드 관리 도구 만을 제공한다.

*DAG*

Directed Acyclic Graph. (비순환 방향 그래프) 커밋 객체들은 비순환 방향 그래프를 이룬다. 커밋은 부모를 가지며 (directed) 커밋 객체의 그래프는 순환하지 않기 (acyclic, 동일한 객체에서 시작하고 끝나는 체인이 존재하지 않는다) 때문이다.

*dangling object (댕글링 객체)*

도달할 수 없는 객체. 심지어는 다른 도달할 수 없는 객체로부터도 이 객체에 도달할 수 없다. 댕글링 객체는 저장소 내의 어떤 참조나 객체로부터 참조되지 않는다.

*detached HEAD (분리된 HEAD)*

일반적으로 HEAD는 브랜치의 이름을 저장한다. 하지만 git에서는 임의의 커밋을 체크아웃할 수 있는데, 이 커밋은 특정 브랜치의 팁이 아닌 경우도 가능하며 이 때 HEAD가 분리되었다고 한다.

*dircache*

아주 오래된 용어이다. index를 참고하기 바란다.

*directory (디렉터리)*

"ls" 명령을 통해 얻을 수 있는 목록. :-)

*dirty (변경되다)*

작업 트리가 현재 브랜치에 커밋되지 않은 변경 사항을 포함한 경우에 "dirty"하다고 한다.

*ent*

몇몇 geek들이 사용하는 "tree-ish"에 대한 동의어. 자세한 설명은 [http://en.wikipedia.org/wiki/Ent\_(Middle-earth)](http://en.wikipedia.org/wiki/Ent_%28Middle-earth%29) 문서를 참조하기 바란다. 혼란을 주지 않도록 이 용어를 사용하지 않는 것이 좋다.

*evil merge*

어떠한 부모에도 속하지 않는 변경 사항을 만들어내는 머지

*fast forward (고속 이동)*

고속 이동이란 머지의 특별한 형태로, 어떤 리비전을 가지고 있을 때 이를 다른 브랜치의 변경 사항과 머지하는 경우에, 다른 브랜치의 모든 변경 사항들이 현재 리비전의 자손인 경우이다. 이 때 새로운 머지 커밋을 만드는 대신 현재 리비전을 다른 브랜치의 최신 리비전으로 갱신한다. 이는 원격 저장소를 추적하는 브랜치에서 자주 발생할 것이다.

*fetch (내려받기)*

브랜치를 머지하는 것은 원격 저장소의 헤드에 대한 참조(head ref)를 가져오고, 로컬 객체 데이터베이스에 없는 객체를 찾아서 이러한 객체들도 함께 가져오는 것이다. [git-fetch(1)](http://www.kernel.org/pub/software/scm/git/docs/git-fetch.html) 문서를 살펴보기 바란다.

*file system (파일 시스템)*

리누스 토발즈는 원래 git가 사용자 공간 파일 시스템이 되도록 설계했다. 즉, 파일과 디렉토리를 포함할 수 있는 기반 구조인 것이다. 이 점이 git의 효율성과 속도를 보장해 준다.

*git archive (git 아카이브)*

(arch 사용자들을 위한) 저장소의 다른 이름이다.

*grafts*

grafts는 가상의(fake) 조상 커밋을 생성하여 두 개의 서로 다른 개발 경로를 하나로 합치도록 해 준다. 이러한 방식으로 git가 커밋을 생성할 때 저장된 것과 다른 부모 커밋들을 가장할 수 있게 해 준다. .git/info/grafts 파일을 통해 설정한다.

*hash (해시)*

git에 대해서는 객체 이름과 같은 의미이다.

*head (헤드)*

브랜치의 팁에 있는 커밋을 가리키는 이름있는 참조.

*HEAD*

현재 브랜치. 더 자세히는 일반적으로 HEAD를 통해 참조할 수 있는 트리의 상태에서 만들어내는 작업 트리. HEAD는 저장소 내의헤드 중의 하나에 대한 참조이다. 단 분리된 HEAD를 사용하는 경우는 예외이며, 이 때는 임의의 커밋에 대한 참조가 될 수 있다.

*head ref (헤드 참조)*

헤드와 동일하다.

*hook (훅)*

여러 git 명령이 일반적으로 수행되는 과정에서, 개발자가 추가적인 기능이나 검사를 수행하도록 외부 스크립트를 실행할 수 있다. 일반적으로 훅을 이용하여 명령을 미리 검증하여 필요한 경우 이를 중지하거나, 작업이 끝난 후에 이를 알려주는 기능을 수행할 수 있다. 훅 스크립트는 $GIR\_DIR/hooks/ 디렉터리에서 찾을 수 있으며, 이들을 실행 가능하게 만들면 바로 사용할 수 있다.

*index (인덱스)*

상태 정보를 포함한 파일의 모음. 각 파일의 내용은 객체로 저장된다. 인덱스는 작업 트리의 저장된 버전이다. 사실 인덱스는 머지시에 사용되는 두번째 버전을 (심지어는 세번째 버전도) 포함할 수도 있다.

*index entry (인덱스 항목)*

인덱스 내에 저장된 특정 파일에 대한 정보. 만약 머지가 시작된 후 아직 끝나지 않았다면 (인덱스가 해당 파일의 여러 버전을 포함한 경우) 인덱스 항목을 머지에서 제외할 수 있다.

*master*

기본 개발 브랜치. git 저장소를 생성할 때마다 "master"라는 이름의 브랜치가 만들어지고, 활성 브랜치가 된다. 대부분의 경우 이 브랜치는 로컬의 개발 내용을 포함하지만, 이는 순전히 관례적인 것이며 필수 사항은 아니다.

*merge (머지)*

동사로 쓰일 때: 다른 브랜치(외부 저장소에 있을 수도 있음)의 내용을 현재 브랜치로 가져오는 것을 말한다. 머지해 오는(merged-in) 브랜치가 다른 저장소에 속한 경우에는, 먼저 원격 브랜치의 내용을 내려받고(fetch) 그 결과를 현재 브랜치에 머지한다. 이러한 내려받기와 머지의 조합을 pull이라고 한다. 머지는 브랜치가 나뉜 이후의 변경 사항들을 찾아낸 후, 이러한 모든 변경 사항들을 함께 적용하는 과정을 자동으로 수행하는 것이다. 변경 사항이 충돌하는 경우에는, 머지를 완료하기 위해 개발자가 직접 개입해야 할 수도 있다.

명사로 쓰일 때: 고속 이동이 아니라면, 머지가 성공적으로 끝난 후에 이에 대한 결과를 나타내는 새 커밋이 만들어지며, 머지된 브랜치들의 팁을 부모로 가지게 된다. 일러한 커밋을 "머지 커밋"이라고 부르며, 때로는 단순히 "머지"라고도 한다.

*object (객체)*

git 내의 저장 단위. 객체는 그 내용에 대한 SHA1 해시값으로 구분된다. 따라서, 객체는 변경될 수 없다.

*object database (객체 데이터베이스)*

"객체"의 집합을 저장하며, 각각의 객체는 객체 이름으로 구분한다. 객체는 보통 $GIR\_DIR/objects/ 내에 존재한다.

*object identifier (객체 식별자)*

객체 이름과 동일하다.

*object name (객체 이름)*

객체에 대한 고유한 식별자. Secure Hash Algorithm 1을 이용한 객체의 내용에 대한 해시값이며, 보통 40글자의 16진수 인코딩으로 표현한다.

*object type (객체 형식)*

객체의 타입을 나타내는 "commit", "tree", "tag", "blob" 중의 하나

*octopus*

3개 이상의 브랜치를 머지하는 것을 말한다. 또한 영리한 포식 동물(?)을 말한다.

*origin*

기본 업스트림 저장소. 대부분의 프로젝트는 최소 하나 이상의 추적하는 업스트림 프로젝트를 가진다. origin은 기본적으로 이러한 목적으로 사용된다. 새 업스트림 업데이트는 "git branch -r" 명령을 통해 볼 수 있는 origin/<업스트림 브랜치 이름>의 원격 추적 브랜치로 내려받아지게 될 것이다.

*pack (팩)*

(공간을 절약하고 전송 효율을 높이기 위해) 하나의 파일로 압축된 여러 객체들의 집합

*pack index (팩 인덱스)*

팩의 내용을 효율적으로 접근할 수 있도록 해주는 팩 내의 객체에 대한 식별자 및 다른 정보의 목록.

*parent (부모)*

개발 경로 상에 논리적으로 바로 앞서 있는 것들을 포함하는 (비어있을 수도 있음) 커밋 객체.

*pickaxe*

pickaxe란 용어는 diffcore 루틴이 주어진 텍스트를 추가하거나 지운 변경 사항을 선택하도록 도와주는 옵션을 가리킨다. --pickaxe-all 옵션을 이용하면 특정한 텍스트 줄을 추가하거나 삭제한 전체 변경 세트를 볼 수 있다. **git-diff(1)** man 페이지를 살펴보기 바란다.

*plumbing*

코어 git에 대한 별칭

*porcelain*

코어 git에 의존하며 고수준의 접근 기능을 제공하는 프로그램 및 프로그램 스위트에 대한 별칭. porcelain은 plumbing에 비해 더 풍부한 SCM 인터페이스를 제공한다.

*pull*

브랜치를 pull한다는 것은 브랜치를 내려받아(fetch) 머지(merge)한다는 것을 의미한다. **git-pull(1)** man 페이지를 살펴보기 바란다.

*push*

브랜치를 push한다는 것은 원격 저장소에서 헤드 참조를 얻어와서, 해당 참조가 브랜치의 로컬 헤드 참조의 직접적인 조상인지를 검사하고 그렇다면 로컬 헤드 참조에서 도달 가능하고 원격 저장소에는 없는 모든 객체들을 원격 객체 데이터베이스로 보내고 원격 헤드 참조를 업데이트하는 것을 의미한다. 만약 원격 헤드가 로컬 헤드의 직접적인 조상이 아니라면 push는 실패한다.

*reachable (도달 가능)*

주어진 커밋의 모든 조상들은 해당 커밋에서 "도달 가능"하다고 말한다. 좀 더 일반적으로 표현하면, 어떤 객체가 다른 객체에서 도달 가능한 경우는 해당 객체가 만든 태그 혹은 부모나 트리에 대한 커밋 혹은 트리나 그를 포함한 블롭에 대한 트리 등의 체인을 통해 다른 객체에 닿을 수 있는 경우이다.

*rebase*

어떤 브랜치에서의 일련의 변경 사항들을 다른 베이스에 다시 적용하고 그 결과로 브랜치의 헤드를 리셋하는 것을 말한다.

*ref (참조)*

특정 객체를 나타내는 40바이트 16진수 표현의 SHA1 해시값 혹은 이름. 참조는 $GIT\_DIR/refs/ 내에 저장될 수 있다.

*reflog*

reflog는 참조의 로컬 "변경 이력"을 보여준다. 다시 말하면 이 저장소의 3번째 이전 리비전이 무엇인지, 이 저장소의 현재 상태가 무엇인지를 알려줄 수 있다. 자세한 정보는 **git-reflog(1)** man 페이지를 살펴보기 바란다.

*refspec*

"refspec"은 fetch나 push  명령 수행 시 원격 참조와 로컬 참조 간의 매핑을 나타내기 위해 사용된다. 이들은 <소스>:<목적지>의 형식으로 콜론(:)으로 구분되며 앞에 + 기호가 올 수도 있다. 예를 들어 git fetch $URL refs/heads/master:refs/heads/origin 명령은 "$URL에서 master 브랜치 헤드를 가져와서 로컬의 origin 브랜치 헤드에 저장"하라는 의미이다. git push $URL refs/heads/master:refs/heads/to-upstream 명령은 "로컬의 master 브랜치 헤드를 $URL의 to-upstream 브랜치로 공개"하라는 의미이다. **git-push(1)** man 페이지도 함께 살펴보기 바란다.

*repository (저장소)*

참조 및 해당 참조로부터 도달할 수 있는 모든 객체들을 포함하는 객체 데이터베이스의 모음이며 하나 이상의 porcelain에서 사용하는 메타데이터도 포함될 수 있다. 저장소는 대리 메커니즘을 통해 다른 저장소와 객체 데이터베이스를 공유할 수 있다.

*resolve (충돌 해결)*

자동 머지의 실패로 남아있는 것을 직접 수정하는 행위.

*revision (리비전, 버전)*

객체 데이터베이스에 저장된 파일과 디렉터리들의 특정 상태. 이는 커밋 객체를 통해 참조된다.

*rewind*

개발된 내용의 일부를 날려버리는 것. 즉, 헤드를 이전 리비전으로 할당하는 것을 뜻한다.

*SCM*

Source Code Management. 소스 코드 관리 도구

*SHA1*

객체 이름과 동일하다.

*shallow repository (shallow 저장소)*

shallow 저장소는 일부 커밋들의 부모가 사라진(cauterized away) 불완전한 변경 이력을 가진다. (다시 말해 git는 이러한 커밋들이커밋 객체 상에는 부모에 대한 정보가 기록되어 있지만, 마치 부모가 없는 것처럼 보여준다.) 이것은 업스트림에 저장된 실제 변경 이력이 매우 크지만, 오직 프로젝트의 최신 변경 이력에만 관심이 있는 경우에 유용하게 사용될 수 있다. shallow 저장소는 **git-clone(1)** 명령에 --depth 옵션을 주어 만들 수 있으며. 변경 이력은 나중에 **git-fetch(1)** 명령을 통해 확장(deepen)될 수 있다.

*symref*

심볼릭 참조. SHA1 id 자체를 포함하는 대신 ref: refs/some/thing 형태이며, 참조되는 경우 재귀적으로 이 참조로 역참조된다. HEAD가 대표적인 symref의 예이다. 심볼릭 참조는 **git-symbolic-ref(1)** 명령을 통해 제어할 수 있다.

*tag (태그)*

태그 혹은 커밋 객체를 가리키는 참조. 헤드와 달리 태그는 커밋에 의해 변경되지 않는다. (태그 객체가 아닌) 태그는 $GIT\_DIR/refs/tags 내에 저장된다. git 태그는 Lisp 태그(git에서는 객체 형식이라고 부를 수 있는)와는 아무런 관련이 없다. 태그는 거의 대부분 커밋 조상 체인 내의 특정 지점을 표시하기 위해 사용한다.

*tag object (태그 객체)*

다른 객체를 가리키는 참조를 포함하는 객체이며, 커밋 객체와 같이 메시지를 포함할 수 있다. 또한 (PGP) 서명을 포함할 수도 있으며, 이 경우에는 "서명된 태그 객체"라고 불린다.

*topic branch (주제별 브랜치)*

개발자가 개념적인 개발 경로를 구분하기 위해 사용하는 일반 git 브랜치. 브랜치는 매우 쉽고 가벼우므로, 잘 정의된 개념들을 포함하거나 서로 관련이 없는 변경 사항들을 모아둔 작은 브랜치를 여러 개 가지는 것이 때로는 더 좋은 경우가 있다.

*tracking branch (추적 브랜치)*

다른 저장소의 변경 사항들을 따라가기 위해 사용하는 일반 git 브랜치. 추적 브랜치는 직접적인 수정이나 그에 대한 로컬 커밋을 가져서는 안된다. 추적 브랜치는 보통 pull 명령 수행 시 오른쪽에 오는 참조로 구분할 수 있다. refspec 부분을 살펴보기 바란다.

*tree (트리)*

작업 트리거나, 의존적인 블롭 및 트리 객체(작업 트리의 저장된 표현)를 포함한 트리 객체 중의 하나이다.

*tree object (트리 객체)*

파일 이름과 모드 및 그에 대한 블롭 또는 트리 객체에 대한 참조들의 목록을 포함하는 객체. 트리는 디렉터리와 동일하다.

*tree-ish*

커밋 객체, 트리 객체 혹은 태그, 커밋, 트리 객체를 가리키는 태그 객체 중의 하나를 가리키는 참조.

*unmerged index (머지되지 않은 인덱스)*

머지되지 않은 인덱스 항목을 포함하는 인덱스.

*unreachable object (도달할 수 없는 객체)*

브랜치나 태그 혹은 다른 어떤 객체에서도 도달할 수 없는 객체.

*working tree (작업 트리)*

실제로 체크아웃된 파일들의 트리. 작업 트리는 일반적으로 HEAD와 로컬에서 변경하였지만 아직 커밋되지 않은 사항들을 합친 것과 같다.

**부록 A: Git 빠른 참조**[**#**](http://namhyung.springnote.com/pages/3132772#toc_104)

이 부분은 git의 주요 명령에 대한 간단한 요약이다. 앞 장에서 각 명령이 동작하는 방법에 대해 자세히 설명한다.

**새 저장소 만들기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_105)

tarball에서 만들기:

1. $ tar xzf project.tar.gz  
   $ cd project  
   $ git init  
   Initialized empty Git repository in .git/  
   $ git add .  
   $ git commit

원격 저장소에서 만들기:

1. $ git clone git://example.com/pub/project.git  
   $ cd project

**브랜치 관리하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_106)

1. $ git branch         # 이 저장소 내의 모든 로컬 브랜치의 목록을 보여줌  
   $ git checkout test  # 작업 디렉터리를 "test" 브랜치로 전환  
   $ git branch new     # 현재 HEAD에서 시작하는 "new" 브랜치 생성  
   $ git branch -d new  # "new" 브랜치 삭제

현재 HEAD (기본값)가 아닌 다른 위치에서 시작하는 브랜치를 만들기:

1. $ git branch new test    # "test"라는 이름의 브랜치에서 시작  
   $ git branch new v2.6.15 # tag named v2.6.15 태그에서 시작  
   $ git branch new HEAD^   # 가장 최근 커밋 바로 전의 커밋에서 시작  
   $ git branch new HEAD^^  # 위의 커밋 바로 전의 커밋에서 시작  
   $ git branch new test~10 # "test" 브랜치 팁의 10번째 앞의 커밋에서 시작

새 브랜치를 만듦과 동시에 해당 브랜치로 전환:

1. $ git checkout -b new v2.6.15

clone 명령을 수행했던 저장소로부터 브랜치를 업데이트하거나 살펴보기:

1. $ git fetch             # 업데이트  
   $ git branch -r         # 목록 보기  
     origin/master  
     origin/next  
     ...  
   $ git checkout -b masterwork origin/master

다른 저장소에서 브랜치를 내려받아서 로컬 저장소에 다른 이름으로 저장:

1. $ git fetch git://example.com/project.git theirbranch:mybranch  
   $ git fetch git://example.com/project.git v2.6.15:mybranch

정기적으로 작업하는 저장소들의 목록 관리:

1. $ git remote add example git://example.com/project.git  
   $ git remote                    # 원격 저장소의 목록  
   example  
   origin  
   $ git remote show example       # 자세한 정보  
   \* remote example  
     URL: git://example.com/project.git  
     Tracked remote branches  
       master next ...  
   $ git fetch example             # example로부터 브랜치 업데이트  
   $ git branch -r                 # 모든 원격 브랜치의 목록

**변경 이력 살펴보기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_107)

1. $ gitk                      # 변경 이력을 그래피컬하게 살펴보기  
   $ git log                   # 모든 커밋의 목록  
   $ git log src/              # src/ 를 변경한 모든 커밋의 목록  
   $ git log v2.6.15..v2.6.16  # v2.6.15과 v2.6.16 사이의 모든 커밋의 목록  
   $ git log master..test      # test 브랜치에는 있지만 master 브랜치에는 없는 모든 커밋의 목록  
   $ git log test..master      # master 브랜치에는 있지만 test 브랜치에는 없는 모든 커밋의 목록  
   $ git log test...master     # 둘 중 하나의 브랜치에만 속한 모든 커밋의 목록  
   $ git log -S'foo()'         # "foo()"를 변경한 모든 커밋의 목록  
   $ git log --since="2 weeks ago"  
   $ git log -p                # 패치도 함께 보여줌  
   $ git show                  # 가장 최근 커밋  
   $ git diff v2.6.15..v2.6.16 # 두 태그 버전 간의 차이  
   $ git diff v2.6.15..HEAD    # 현재 헤드와의 차이  
   $ git grep "foo()"          # "foo()"를 포함하는 작업 트리 검색  
   $ git grep v2.6.15 "foo()"  # "foo()"를 포함하는 예전 트리 검색  
   $ git show v2.6.15:a.txt    # 예전 버전의 a.txt 파일 보기

regression 찾기:

1. $ git bisect start  
   $ git bisect bad                # 현재 버전은 문제 있음  
   $ git bisect good v2.6.13-rc2   # 잘 동작하는 최근 버전  
   Bisecting: 675 revisions left to test after this  
                                   # 여기서 테스트를 수행:  
   $ git bisect good               # 현재 버전이 문제가 없다면 수행  
   $ git bisect bad                # 현재 버전이 문제가 있다면 수행  
                                   # 완료될 때까지 반복

**변경 사항 만들기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_108)

git에게 사용자 정보를 제공:

1. $ cat >>~/.gitconfig <<\EOF  
   [user]  
           name = 이름 입력  
           email = 이메일 주소 입력  
   EOF

다음 커밋에 포함할 파일 내용들을 선택하고 커밋:

1. $ git add a.txt    # 업데이트된 파일  
   $ git add b.txt    # 새 파일  
   $ git rm c.txt     # 예전 파일  
   $ git commit

혹은 커밋을 준비하고 수행하는 것을 한 번에 처리:

1. $ git commit d.txt # d.txt의 최신 내용 만을 사용  
   $ git commit -a    # 추적하는 모든 파일의 최신 내용을 사용

**머지**[**#**](http://namhyung.springnote.com/pages/3132772#toc_109)

1. $ git merge test   # "test"브랜치를 현재 브랜치로 머지  
   $ git pull git://example.com/project.git master  
                      # 원격 브랜치의 내용을 내려받아서 머지  
   $ git pull . test  #git merge test와 동일

**변경 사항을 공유하기**[**#**](http://namhyung.springnote.com/pages/3132772#toc_110)

패치 가져오기(import) 혹은 내보내기(export):

1. $ git format-patch origin..HEAD # HEAD에는 속하고 origin에는 속하지 않는  
                                   # 모든 커밋들에 대한 패치 생성  
   $ git am mbox # "mbox" 메일함에서 패치 가져오기

다른 git 저장소에서 브랜치를 내려받고 현재 브랜치에 머지:

1. $ git pull git://example.com/project.git theirbranch

내려받은 브랜치를 머지하기 전에 로컬 브랜치로 저장하기:

1. $ git pull git://example.com/project.git theirbranch:mybranch

로컬 브랜치에 커밋을 수행한 후에, 해당 커밋으로 원격 브랜치를 업데이트:

1. $ git push ssh://example.com/project.git mybranch:theirbranch

원격 브랜치와 로컬 브랜치의 이름이 모두 "test"인 경우:

1. $ git push ssh://example.com/project.git test

자주 사용하는 원격 저장소를 위한 별칭:

1. $ git remote add example ssh://example.com/project.git  
   $ git push example test

**저장소 관리**[**#**](http://namhyung.springnote.com/pages/3132772#toc_111)

손상 검사:

1. $ git fsck

압축 및 사용하지 않는 객체(cruft?) 제거:

1. $ git gc

**부록 B: 이 설명서에 대한 노트 및 할 일 목록**[**#**](http://namhyung.springnote.com/pages/3132772#toc_112)

이 설명서는 계속 작업 중이다.

기본적인 요구 사항은:

* 이 설명서는 기본적인 유닉스 명령행에 대한 내용을 이해하고 있지만 git에 대해서는 아무런 지식도 없는 (지적인) 사람이 처음부터 끝까지 차례대로 읽고 이해할 수 있어야 한다.
* 가능하다면 각 절의 제목은 불필요한 지식을 요구하지 않는 선에서 설명하고자 하는 작업을 분명히 나타내야 한다. 예를 들어 "git-am 명령"보다는 "프로젝트로 패치 가져오기"가 낫다.

사람들이 중간의 모든 내용을 읽지 않고도 중요한 주제를 바로 접할 수 있도록 확실한 chapter dependency graph를 만드는 방법을 생각해 보자.

Documentation/ 디렉터리에서 남겨진 다른 주제들이 남아있는지 찾아보자. 특히:

* HOWTO 문서
* 기술적인 문서
* 훅(hook)에 대한 설명
* git(1) 내의 명령의 목록

이메일 아키이브를 살펴보고 다른 주제들이 남아있는지 찾아보자.

man 페이지들을 살펴보고 이 설명서에서 제공하는 것보다 더 많은 배경 지식이 필요한 것이 있는지 찾아보자.

임시 브랜치를 만드는 대신 연결이 끊어진 헤드를 제안하여 시작하는 것(?)을 단순화시켜보자.

좋은 예제들을 더 추가하자. 전체가 cookbook 형식의 예제 만으로 이루어진 장(chapter)를 만드는 것도 좋을 것 같다. 각 장의 마지막에 "고급 예제" 부분을 추가할 수도 있을 것이다.

적절한 곳에 git 용어에 대한 상호 참조를 추가하자.

shallow clone에 대한 문서화를 추가하자? 자세한 내용은 1.5.0 릴리스 노트의 제안을 살펴보자.

CVS, SVN 등의 다른 버전 관리 시스템과의 연동을 설명하는 부분을 추가하자. 또한 일련의 릴리스 tarball들을 가져오는(import) 하는 방법도 추가하자.

gitweb에 대한 자세한 설명을 추가하자.

plumbing을 이용하고 스크립트를 작성하는 장을 추가하자.

대리 메커니즘, clone -reference 등..

저장소 손상으로부터 복구하는 방법에 대해서는 다음 문서들을 살펴보기 바란다:

<http://marc.theaimsgroup.com/?l=git&m=117263864820799&w=2>

<http://marc.theaimsgroup.com/?l=git&m=117147855503798&w=2>

**번역 용어표**[**#**](http://namhyung.springnote.com/pages/3132772#toc_113)

일반적인 용어들은 [번역 용어집](http://namhyung.springnote.com/pages/325239) 참조할 것!

* builtin : 빌트인
* conflict : 충돌
* diff : 차이점, (명령) diff
* fast-forward : 고속 이동
* linear : 선형(의)
* local : 로컬
* patch series : 패치 묶음. 일련의 패치
* prune : (댕글링 객체 등을) 정리(하다)
* shortcut : 줄임말

[번역](http://namhyung.springnote.com/pages?tags=%EB%B2%88%EC%97%AD),[git](http://namhyung.springnote.com/pages?tags=git),[user](http://namhyung.springnote.com/pages?tags=user),[manual](http://namhyung.springnote.com/pages?tags=manual),[사용법](http://namhyung.springnote.com/pages?tags=%EC%82%AC%EC%9A%A9%EB%B2%95)

2011-09-22 02:42 에 namhyung님이 마지막으로 수정

**댓글 (*14*)**

* prolinko

git 문서들을 한글화 하려 합니다. 이 문서 좀 가져다 써도 될까요?

 2009-08-17 22:40

* namhyung

CC BY-NC-ND license에 의해 사용하실 수 있습니다. 어디에 쓰실지 저에게도 알려주시면 더 좋겠구요.. ^^

 2009-08-17 23:49

* prolinko

감사합니다 ^^. git 공식 리포지터리 안의 문서들을 번역해서 git upstream 에도 반영될 수 있도록 해보려구요. 틈틈히 작업해서 github나 kldp.net 같은 싸이트에 프로젝트 개설하고 올리게 되면 알려드릴게요.

2009-08-18 18:06

* prolinko

근데 ND이면 git의 공식 document형식인 asciidoc 형식으로 변환할 수가 없겠네요. 원 저작물이 GPL이니 adaptation인 번역물도 GPL이 되어야 할 것 같네요.

 2009-08-18 18:26

* namhyung

일단 BY-NC-SA로 변경했습니다. 문서의 경우 license가 어찌되는지 정확히 모르겠군요..

 2009-08-19 01:23

* prolinko

네 감사합니다. 그 정도면 작업하는데 지장이 없을 것 같습니다. 원하신다면 완성된 번역문서에 번역자로 이름을 넣어드릴께요. (참고로 git 공식 문서들은 git 소스의 Document폴더에 포함되어 소스코드와 함께 배포되므로 소스코드와 같은 GPL 적용을 받는 것으로 알고 있습니다)

 2009-08-19 14:13

* namhyung

upstream에 들어가기 위해 필요하다면 GPL로 변경할 의향이 있습니다. 번역자로 이름 넣어주세요.. ^^

 2009-08-19 14:55

* chlrbgh0

정말 유용한 정보네요 잘 보고 갑니다 ^^

 2009-11-03 14:08

* sangheestyle

정말 잘 만드셨습니다. 감사합니다.

 2010-03-11 09:40

* 평강공주

정말 필요할 때마다 잘 보고 있습니다. 이것이 문서화되어 나오면 정말 좋겠습니다.^^..감사합니다~

 2010-06-25 10:05

* 회색늑대

감사드립니다. 행복하십시요.

 2010-08-11 19:29

* 대한민주짱

포스팅을 통해 얻은정보, 정말로 요긴하게 사용하고 있습니다. 감사합니다.

 2011-03-03 17:50

* 태후니

안녕하세요. 김태훈 입니다. 좋은 정보를 많이 올려주시네요. 감사합니다. :)

 2011-05-07 00:46

* bwikbs

헐!! 읽다보니 행님이 쓰신 글이군요!!! 역시 행님이십니다!! 무홍입니다!!

 2011-09-15 10:11

양식의 맨 위