

Pro Git

Scott Chacon*

2011-08-31

*This is the PDF file for the Pro Git book contents. It is licensed under the Creative Commons Attribution-Non Commercial-Share Alike 3.0 license. I hope you enjoy it, I hope it helps you learn Git, and I hope you'll support Apress and me by purchasing a print copy of the book at Amazon: <http://tinyurl.com/amazonprogit>

목차

1	시작하기	1
1.1	버전 관리란?	1
1.1.1	로컬 버전 관리 시스템	1
1.1.2	중앙집중식 버전 관리 시스템 (Centralized VCS)	2
1.1.3	분산형 버전 관리 시스템(Distributed VCS)	3
1.2	간단히 살펴보는 Git의 역사	3
1.3	Git 기초	4
1.3.1	차이점이 아니라 Snapshot	4
1.3.2	로컬에서 거의 모든 명령을 실행	4
1.3.3	Git의 무결성	5
1.3.4	Git은 보통 데이터를 추가할 뿐이다	5
1.3.5	세 가지 상태	6
1.4	Git 설치	7
1.4.1	Source로 설치하기	7
1.4.2	리눅스에 설치	8
1.4.3	Mac에 설치하기	8
1.4.4	윈도우에 설치	8
1.5	Git 최초 설정	9
1.5.1	사용자 정보	9
1.5.2	편집기	9
1.5.3	Diff 도구	10
1.5.4	설정 확인	10
1.6	도움말 보기	10
1.7	정리	11

1장

시작하기

이 챕터는 Git을 처음 사용하는 법에 대해 다룰 것이다. 나는 버전 관리 도구들에 대한 지식들을 먼저 설명한 후에 Git을 설치하는 방법을 다루고 마지막으로 설정해서 사용하는 방법에 대해 다룰 것이다. 이 장을 다 읽고 나면 Git의 탄생배경, Git을 사용하는 이유, Git을 설정하고 사용하는 방법을 알게 될 것이다.

1.1 버전 관리란?

버전 관리는 뭐고 왜 알아야 할까? 버전 관리는 파일들의 변화를 시간에 따라 기록하는 시스템을 말한다. 이 책에 있는 모든 예제들은 모두 버전 관리 시스템을 사용한다. 실제로 컴퓨터에서 사용하는 거의 모든 종류의 파일들은 버전 관리를 할 수 있다.

당신이 그래픽 디자이너이거나 웹 디자이너라면 이미지나 레이아웃의 모든 버전(이력)을 관리하려 한다면 버전 관리 시스템 (VCS:Version Control System)을 사용하는 것이 현명하다. VCS를 사용하면 파일을 이전 상태로 되돌릴 수도 있고, 프로젝트 전체를 이전 상태로 되돌릴 수도 있고, 수정 내역을 계속 비교해 볼 수도 있고, 누가 문제를 만들었는지도 추적할 수 있고, 이슈를 누가 언제 제기했는지도 알 수 있다. VCS를 사용하면 파일을 잃거나 잘못 고쳤을 때 쉽게 복구할 수 있다. 이 모든 것을 하는데 드는 노력도 크지 않다.

1.1.1 로컬 버전 관리 시스템

많은 사람들이 버전을 관리하는 방법으로 다른 디렉토리로 파일을 복사하는 방법이 있다(현명한 사람이라면 디렉토리 이름에 시간을 활용했을 것이다). 이 방법은 간단하기 때문에 자주 사용한다. 그렇지만 정말 예러나기 쉽다. 작업하고 있는 디렉토리를 잃어버리거나 실수로 파일을 잘못 고칠 수도 있고 잘못 복사할 수도 있다.

이런 이유로 프로그래머들은 오래 전에 로컬 VCS를 만들었다. 그 VCS는 관리 중인 파일의 변경 정보를 저장하기 위해 아주 간단한 데이터베이스를 사용했다.

인기있는 VCS 도구 중에 rcs라고 부르는 것이 있는데 오늘날까지도 아직 많은 회사들이 사용하고 있다. 심지어 인기있는 Mac OS X 운영체제에서 개발자 도구를 설치하면 rcs는 함께 설치된다. 이 툴은 기본적으로 patch set(파일에서 변경되는 부분)을 관리한다. 이 patch set을 위한 특수한 형식이 있고 디스크에 저장한다. 일련의 이 patch set을 적용하므로써 모든 파일을 특정 시점으로 되돌릴 수 있다.

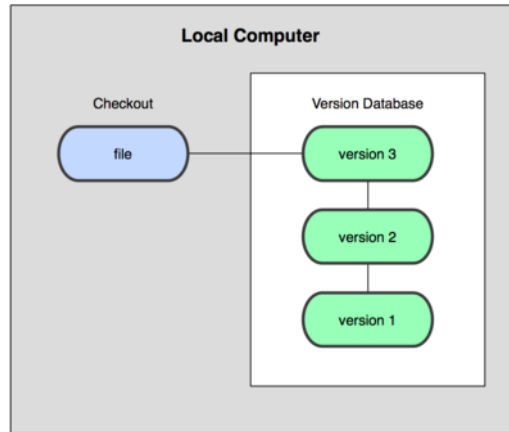


그림 1.1: 로컬 버전 관리 다이어그램.

1.1.2 중앙집중식 버전 관리 시스템 (Centralized VCS)

프로젝트를 진행하다보면 다른 개발자와 함께해야하는 하는 경우도 많다. 이런 경우에 생기는 문제를 해결하기 위해 CVCS(Centralized Version Control System)가 개발됐다. CVS, Subversion, Perforce같은 시스템들은 모든 파일을 관리하는 서버가 별도로 있고 다수의 클라이언트들이 중앙 서버에서 파일을 체크아웃한다. 몇 년 동안 이러한 시스템들이 많은 사랑을 받았다.

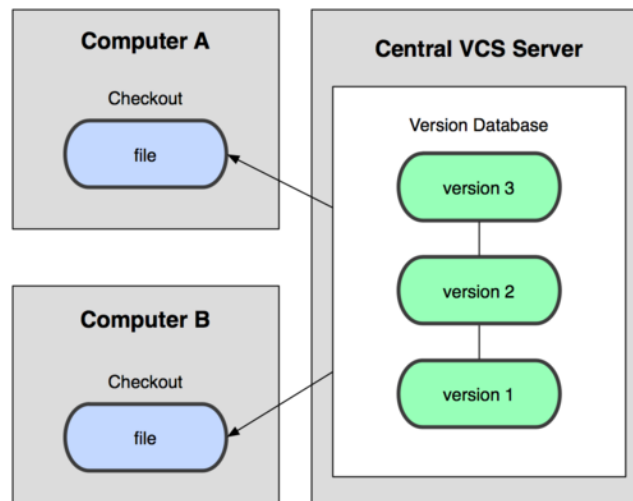


그림 1.2: 중앙집중식 버전 관리 (CVCS) 다이어그램.

이 환경은 로컬 VCS비해 많은 장점이 있다. 예를 들어, 누가 무엇을 하고 있는지 프로젝트에 참여하고 있는 사람 모두 알 수 있다. 관리자는 누가 무엇을 할 수 있는지 세세히 관리할 수 있다. 모든 클라이언트의 로컬 데이터베이스를 관리하는 것보다 CVS 하나를 관리하는 것이 훨씬 쉽다.

그러나 이런 환경은 몇 가지 치명적인 결점이 있다. 가장 대표되는 것은 중앙 서버에서 생기는 문제다. 만약 서버가 한 시간 동안 다운된다면 그동안 아무도 다른 사람과 협업을 할 수 없고 사람들이 하고 있는 일을 백업할 수 있는 방법도 마땅히 찾을 수 없다. 그리고 중앙 데이터베이스가 있는 하드디스크에 문제가 생긴다면 프로젝트의 모든 히스토리를 잃게 된다. 물론 사람들마다 하나씩 가지고 있는 snapshot은 괜찮다. 로컬 VCS 시스템도 비슷하다. 만약 같은 문제가 발생하면 모든 것을 잃게 된다.

1.1.3 분산형 버전 관리 시스템(Distributed VCS)

이제 DVCS 차례다. Git, Mercurial, Bazaar, Darcs같은 DVCS에서는 클라이언트가 파일의 마지막 snapshot을 체크아웃하지 않는다. 단지 repository를 전부 복제할 뿐이다. 서버가 죽으면 그 복제물로 다시 작업을 시작할 수 있다. 클라이언트 중에서 아무거나 골라도 서버를 복원할 수 있다. 모든 체크아웃은 모든 데이터를 가지고 있는 진정한 백업이다.

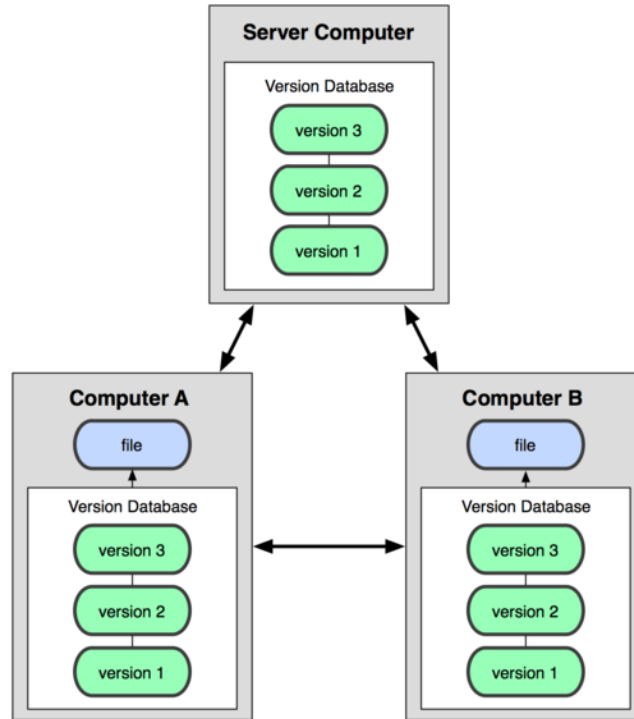


그림 1.3: 분산형 버전 관리 (DVCS) 다이어그램.

게다가 대부분의 DVCS들은 다수의 원격 저장소(Repository)가 존재할 수 있다. 원격 저장소가 여러개라고 해도 아무 문제 없다. 그래서 사람들은 동시에 다양한 그룹과 다양한 방법으로 협업할 수 있다. 계층 모델같은 중앙집중식 시스템으로는 할 수 없는 몇 가지 워크플로우를 사용할 수 있다.

1.2 간단히 살펴보는 Git의 역사

인생을 살다보면 여러가지 일들이 벌어지듯, Git의 삶 또한 창조적인 파괴와 모순속에서 시작되었다. 리눅스 커널은 굉장히 규모가 큰 오픈소스 프로젝트다. 리눅스 커널의 일생에서 대부분의 시절은 패치와 단순 파일(Archived file)로만 관리했다. 2002년에 드디어 리눅스 커널은 BitKeeper라고 불리는 상용 DVCS를 사용하기 시작했다.

2005년 커뮤니티가 만드는 리눅스 커널과 사익을 추구하는 회사가 개발한 BitKeeper의 관계는 틀어졌다. BitKeeper의 무료 사용이 제고된 것이다. 이것은 리눅스 개발 커뮤니티(특히 리눅스 창시자 리누스 토발즈)가 자체 도구를 만들도록 만들었다. Git은 BitKeeper를 사용하면서 배운 교훈을 기초로 다음과 같은 목표를 표방했다:

- 속도
- 단순한 설계
- 비선형적인 개발(수 천개의 동시 다발적인 브랜치)

- 완벽한 분산
- 리눅스 커널같이 대형 프로젝트에도 유용할 것(속도나 데이터 크기 면에서)

Git은 2005년 탄생한 후에 아직 초기 지향점을 그대로 유지하고 있으면서도 사용하기 쉽게 진화하고 성숙했다. Git은 미친듯이 빨라서 대형 프로젝트에도 굉장히 유용하다. 이것은 동시다발적인 브랜치에도 끄떡없는 슈퍼 울트라 브랜칭 시스템이다 (3장 참고).

1.3 Git 기초

그래서 Git의 핵심은 뭘까? 이 질문은 Git을 이해하는데 굉장히 중요하다. Git이 무엇이고 어떻게 동작하는지 이해한다면 쉽게 Git을 효과적으로 사용할 수 있다. Git을 배우고 있다면 Subversion이나 Perforce같은 다른 VCS를 사용하던 경험을 지워버려야 한다. Git은 미묘하게 달라서 기존의 경험을 조금 혼란스럽게 만들어 버릴 것이다. 사용자 인터페이스는 매우 비슷하지만 Git은 다른 시스템과는 정보를 다르게 취급한다. 이 차이들을 이해한다면 Git을 사용하는 것이 어렵지 않을 것이다.

1.3.1 차이점이 아니라 Snapshot

Subversion, Subversion의 친구들과 Git의 가장 큰 차이점은 데이터를 다루는 방법에 있다. 개념적으로 대부분의 다른 시스템들이 관리하는 정보는 파일들을 목록이다. CVS, Subversion, Perforce, BaZaar등의 시스템들은 파일의 집합으로 정보를 관리한다. 각 파일들의 변화를 그림1.4처럼 시간순으로 관리한다.

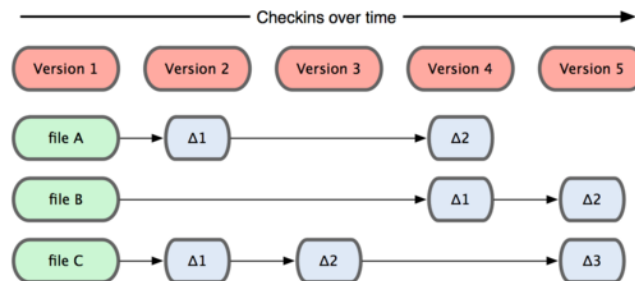


그림 1.4: 각 파일에 대한 변화(차이점)를 저장하는 시스템들.

Git은 이런 식으로 데이터를 저장하지도 취급하지도 않는다. 대신 Git의 데이터는 아주 작은 파일 시스템의 Snapshot이라고 할 수 있다. Git은 Commit하거나 프로젝트의 상태를 저장할 때마다 파일이 존재하고 있는 그 순간을 중요하게 여긴다. 성능을 위해서 파일이 달라지지 않았으면 Git은 저장하지 않는다. 단지 이전 상태의 파일에 대한 링크만 저장한다. Git은 그림1.5처럼 동작한다.

이것은 Git과 다른 VCS를 구분하는 중요한 점이다. Git은 이전 버전을 복사하는 다른 버전 관리 시스템을 바보로 만든다. Git은 마치 작은 독자적인 파일시스템처럼 보이게 만든다. 물론 강력한 도구도 있다. Git은 단순한 VCS가 아니다. 이제3장에서 설명할 Git 브랜칭을 사용할 때 얻을 수 있는 이득에 대해 설명할 것이다.

1.3.2 로컬에서 거의 모든 명령을 실행

거의 모든 명령은 로컬 파일과 자원 만을 사용한다. 보통 네트워크에 있는 다른 컴퓨터의 정보는 필요하지 않다. 대부분의 명령어가 네트워크의 속도에 영향을 받는 CVCS에 익숙하다면

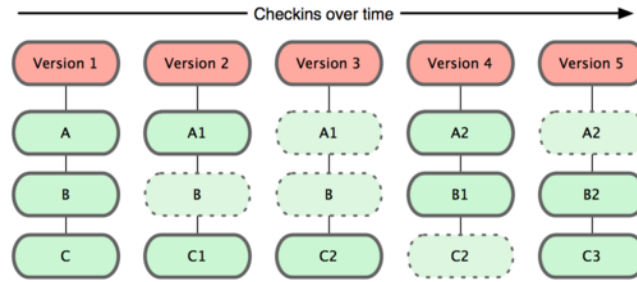


그림 1.5: Git은 시간순으로 프로젝트의 snapshot을 저장한다.

Git이 매우 놀라울 것이다. Git의 이런 특징은 이런 빛의 속도는 Git신만이 구사할 수 있는 초인적인 능력이라고 생각하게 만들 것이다. 프로젝트의 모든 히스토리를 로컬 디스크에 가지고 있기 때문에 모든 명령을 순식간에 실행한다.

예를 들어 프로젝트의 히스토리를 조회하려 할때 Git은 서버가 필요없다. 그냥 로컬 데이터베이스에서 히스토리를 읽어서 보여 준다. 그렇기에 눈깜짝할 사이에 히스토리를 조회할 수 있다. 어떤 파일의 현재버전과 한달전의 상태를 비교해보고 싶다면 Git은 그냥 한 달전의 파일과 지금의 파일을 찾는다. 로컬에서 비교하기 위해 원격에 있는 서버에 접근한 후 예전 버전을 가져올 필요가 없다.

즉 오프라인 상태에서도 비교할 수 있다. 비행기나 기차등에서 작업하고 네트워크에 접속하고 있지 않아도 Commit할 수 있다. 다른 시스템에서는 불가능한 일이다. 예를 들어 Perforce에서는 서버에 연결할 수 없을 때 할 수 있는 일이 별로 없다. Subversion이나 CVS에서도 마찬가지다. 데이터베이스에 접근할 수 없기 때문에 파일을 편집할 수 있지만 Commit할 수는 없다. 이것이 매우 사소해 보일지라도 당해보면 매우 큰 차이를 느낄 수 있다.

1.3.3 Git의 무결성

Git은 모든 것을 저장하기 전에 체크섬을 구한 후 그 체크섬으로 관리한다. 때문에 체크섬없이 모든 파일의 내용과 디렉토리를 바꾸는 것이 불가능하다. 체크섬은 Git에서 사용하는 원자적인 데이터 단위이고 Git의 기본 철학도 이에 따른다. Git없이 작업 내용을 잃어버릴 수도 파일의 상태를 알 수도 없다. 모든 것에는 Git이 필요하다.

Git은 SHA-1 해시를 사용하여 체크섬을 만든다. 만들어진 체크섬은 40자 길이의 16진수 문자열이다. 파일의 내용이나 디렉토리 구조를 이용하여 체크섬을 구한다. SHA-1은 다음과 같이 생겼다.

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git은 모든 것을 해시값으로 사용하기 때문에 눈으로 확인할 수 있다. 실제로 Git은 파일 이름으로 저장하지 않기 때문에 해당 파일의 해시값으로만 접근할 수 있다.

1.3.4 Git은 보통 데이터를 추가할 뿐이다

Git으로 무엇인가를 할때 Git은 데이터를 추가할 뿐이다. 어떤 방법으로든 되돌리거나 데이터를 삭제할 수 없다. 다른 VCS에서 처럼 Git에서도 Commit하지 않았기 때문에 변경사항을 잃어버리거나 망쳐버릴 수 있다. 하지만 Git으로 Snapshot을 Commit한 후에 데이터를 동일한 프로젝트의 다른 저장소로 Push할 수 있다.

심각하게 망칠 걱정없이 실험할 수 있기 때문에 Git을 사용하는 것은 매우 신나다. Git이 데이터를 어떻게 저장하고 손실을 복구할 수 있는지 좀 더 알아보려면 9장의 'Under the Covers'를 보아라.

1.3.5 세 가지 상태

이제 집중해야 한다. Git을 큰 어려움없이 계속 공부하기 위해 반드시 기억해야 할 부분이다. Git에서 파일은 세 가지 상태를 가질 수 있다. **Committed**, **Modified**, **Staged**가 그 세 가지 상태이다. **Committed**는 데이터가 로컬 데이터베이스에 안전하게 저장됐다는 것을 의미한다. **Modified**는 수정한 파일이 아직 로컬 데이터베이스에 **Commit**되지 않은 것을 말한다. **Staged**는 현재 수정한 파일을 곧 **Commit** 할 것이라고 표시한 상태를 의미한다.

그러니까 당연히 Git 프로젝트의 세 가지 단계 Git 디렉토리, 작업 디렉토리, Staging Area를 알아야 한다.

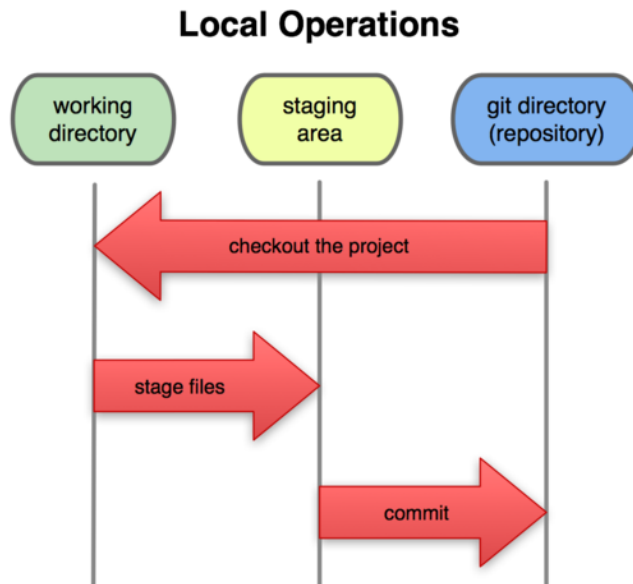


그림 1.6: 작업 디렉토리, Staging Area, Git 디렉토리

Git 디렉토리는 Git이 해당 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳을 말한다. 이것은 Git의 가장 중요한 부분이다. 다른 컴퓨터에서 저장소를 Clone할 때 생성된다.

작업 디렉토리는 프로젝트의 특정 버전을 체크아웃한 곳을 말한다. 이 파일들은 Git 디렉토리에 있는 압축된 데이터베이스에서 가져온다. 이것은 지금 작업하려는 디스크에 있다.

Staging Area는 보통 Git 디렉토리에 있는 단순한 파일이다. 여기에 곧 Commit할 파일에 대한 정보를 저장한다. 하나의 인덱스와 비슷하지만 단순히 Staging Area에 파일을 참조시키는 것에 불과하다.

기본적인 Git의 워크플로우는 다음과 같다:

1. 작업 디렉토리에서 파일을 수정한다.
2. 파일을 Stage해서 Staging area에 있는 snapshot에 그 파일을 추가한다.
3. Staging Area에 있는 파일들을 Commit해서 Git 디렉토리에 영구적인 snapshot으로 저장한다.

Git 디렉토리에서 파일을 수정했는데 그 것이 그 자체로 의미를 갖는다면 **Commit**하라. 파일을 수정하고 **Staging Area**에 추가했다면 단지 **Stage**했을 뿐이다. 체크아웃한 후에 파일을 수정하고 **Stage**하지 않았다면 그냥 **Modified** 상태가 된다. 2장에서 이 상태들에 대해서 좀 더 배울 수 있을 것이다. 거기서에서는 이 상태들의 활용법과 **staged** 단계를 사용하지 않는 방법도 배운다.

1.4 Git 설치

Git을 실제로 한 번 사용해 보려면 우선 설치해야 한다. 다양한 방법으로 Git을 설치할 수 있다. 그렇지만 가장 일반적인 방법은 두 가지가 있는데 하나는 소스로 설치하는 것이고 다른 하나는 각 플랫폼의 패키지로 설치하는 것이다.

1.4.1 Source로 설치하기

가장 최신 버전을 설치할 수 있기 때문에 여력이 되면 소스로 Git을 설치하는 것이 유용하다. Git은 계속 UI를 개선 하고 있다. 소스를 가지고 Git을 컴파일할 수 있다면 최신 버전을 사용할 수 있다. 많은 리눅스 배포판의 패키지들은 어느정도 예전의 버전이다. 그래서 **Backport**를 사용하거나 최근 버전의 배포판을 사용하고 있지 않다면 소스로 설치하는 것이 최선일 수 있다.

Git을 설치하려면 다음과 같은 라이브러리 들이 필요하다. Git은 **curl**, **zlib**, **openssl**, **expat**, **libiconv**에 의존한다. 예를 들어 Fedora처럼 **yum**을 가지고 있는 시스템을 사용하고 있거나 **apt-get**이 있는 데비안 기반 시스템을 사용하고 있다면 다음의 명령어를 실행하여 의존 패키지들을 설치할 수 있다:

```
$ yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
libz-dev libssl-dev
```

필요한 의존성을 다 해결하고 다음 단계를 진행한다. Git 웹 사이트에서 최신 **snapshot**을 가져온다:

<http://git-scm.com/download>

그리고 컴파일하고 설치한다:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

그 다음부터는 Git을 사용하여 Git 자체를 업데이트할 수 있다:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

1.4.2 리눅스에 설치

리눅스에서 인스톨러로 Git을 설치할 때에는 보통 각 배포판에서 사용하는 패키지 관리 도구를 사용하여 설치한다. Fedora에서는 다음과 같이 한다:

```
$ yum install git-core
```

Ubuntu같은 데비안 기반 배포판이라면 apt-get을 사용한다:

```
$ apt-get install git-core
```

1.4.3 Mac에 설치하기

Mac에 Git을 설치하는 방법은 두 가지다. 가장 쉬운 방법은 GUI 인스톨러를 사용하는 방법이다. 이 것은 Google Code 페이지에서 다운받을 수 있다:

<http://code.google.com/p/git-osx-installer>

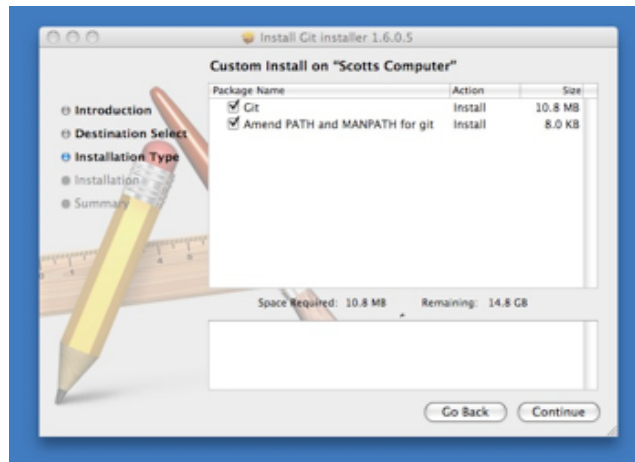


그림 1.7: OS X Git 인스톨러.

다른 방법은 MacPorts(<http://www.macports.org>)를 사용한 방법이다. MacPorts가 설치 돼 있으면 다음과 같이 Git을 설치한다:

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

이제 설치에 대한 모든 것은 알아봤다. 그렇지만 만약 Subversion 저장소를 Git과 함께 사용해야 하는 경우라면 svn도 필요할 것이다.

1.4.4 윈도우에 설치

윈도우에 Git을 설치하는 것은 매우 쉽다. msysGit 프로젝트가 가장 쉬운 방법이다. 그냥 구글 코드 페이지에서 인스톨러를 다운받고 실행하면 된다:

<http://code.google.com/p/msysgit>

설치가 완료되면 CLI와 GUI 둘 다 사용할 수 있다. CLI에는 유용하게 사용할 수 있는 SSH 클라이언트가 포함돼 있다.

1.5 Git 최초 설정

Git을 설치했다면 Git의 사용 환경을 원하는 대로 설정하고 싶은 것이다. 단 한번만 설정하면 된다. 업데이트를 해도 유지된다. 그리고 그냥 명령어를 다시 실행해서 언제든지 바꿀 수 있다.

Git은 'git config'라는 쓸만한 도구를 가지고 있다. 이 도구로 설정 변수를 확인하거나 변경할 수 있다. 이 변수를 통해 Git이 어떻게 동작해야 할지를 제어할 수 있다. 이 변수들은 세 가지 위치에 저장된다.

- /etc/gitconfig 파일: 시스템의 모든 사용자와 모든 저장소에 적용되는 설정이다. git config -system 옵션으로 이 파일을 읽고 쓸 수 있다.
- ~/.gitconfig 파일: 특정 사용자에게만 적용되는 설정이다. git config -global 옵션으로 이 파일을 읽고 쓸 수 있다.
- Git 디렉토리에 있는 config 파일(.git/config): 특정 저장소에만 적용되는 설정이다. 각각의 설정은 역순으로 오버라이드된다. 그래서 .git/config가 /etc/gitconfig보다 우선한다.

윈도우에서 Git은 \$HOME 디렉토리(C:\Documents and Settings\%USER)에 있는 .gitconfig 파일을 찾는다. 물론 msysGit은 /etc/gitconfig도 사용한다. 경로는 MSys 루트에 따른 상대 경로다. 인스톨러로 msysGit을 설치할 때 설치 경로를 선택할 수 있다.

1.5.1 사용자 정보

Git을 설치한 후에 가장 처음으로 해야 하는 것은 이름과 이메일 주소를 설정하는 것이다. 이것은 매우 중요하다. Git은 Commit할 때마다 이 정보를 사용한다. Commit한 후에는 정보를 변경할 수 없다.

```
$ git config -global user.name "John Doe"
$ git config -global user.email johndoe@example.com
```

다시 말하지만 -global 옵션으로 설정한 것은 단 한번만 하면 된다. Git은 해당 시스템에서 항상 이 정보를 사용한다. 만약 프로젝트마다 다른 이름과 이메일 주소를 사용하고 싶으면 -global 옵션을 빼고 명령을 실행하면 된다.

1.5.2 편집기

내가 누구인지를 설정했다. 이제는 Git에서 사용할 텍스트 편집기를 고를 차례다. Git은 시스템의 기본 편집기를 사용한다. 일반적으로는 Vi나 Vim이다. 하지만 Emacs같은 다른 텍스트 편집기를 사용하고 싶으면 다음과 같이 실행한다:

```
$ git config --global core.editor emacs
```

1.5.3 Diff 도구

병합(Merge) 충돌(Conflict)을 해결하기 위해 사용하는 Diff 도구를 설정할 수 있다. vimdiff를 사용하고 싶으면 다음과 같이 실행한다:

```
$ git config --global merge.tool vimdiff
```

kdifff3, tkdiff, meld, xxdif, emerge, vimdiff, gvimdiff, ecmerge, opendiff를 사용할 수 있다. 물론 다른 도구도 사용할 수 있다. 7장에서 좀 더 자세하게 다룬다.

1.5.4 설정 확인

내 설정을 확인하려면 git config --list 명령을 실행한다. 그러면 모든 설정 내역을 확인할 수 있다:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Git은 같은 키를 여러 파일(/etc/gitconfig and ~/.gitconfig같은)에서 읽기 때문에 같은 키가 하나 이상 나올 수도 있다. 이 경우에 Git은 나중 값을 사용한다.

git config key 명령을 실행시켜 Git이 특정키에 대해 어떤 값을 사용하는지 확인할 수 있다:

```
$ git config user.name
Scott Chacon
```

1.6 도움말 보기

명령어에 대한 도움말이 필요할 때 도움말을 볼 수 있는 방법은 세 가지이다:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

예를 들어 다음과 같이 실행하면 `config` 명령에 대한 도움말을 볼 수 있다:

```
$ git help config
```

이 도움말은 언제 어디서나 볼 수 있다. 오프라인 상태에서도 가능하다. 도움말과 이 책으로도 충분하지 않으면 다른 사람의 도움이 필요할 것이다. Freenode IRC 서버 (irc.freenode.net)에 있는 `#git`이나 `#github` 채널로 찾아가라. 이 채널은 보통 수 백명의 사람들이 접속해 있다. 이 사람들은 모두 Git에 대해 잘 알고 있다. 기꺼히 도와줄 것이다.

1.7 정리

당신은 Git이 무엇이고 당신이 지금까지 사용해 왔던 다른 CVCS와 어떻게 다른지 배웠다. 또 당신의 시스템에 Git을 성공적으로 설치하고 아이덴티티도 설정했다. 이제는 Git의 기초에 대해 배울 차례이다.