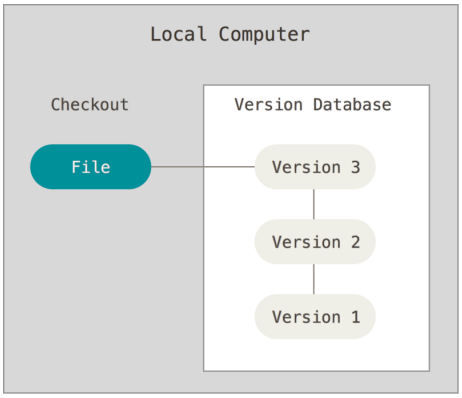
*버전관리란 무엇이고, 버전관리 방법에는 어떤 것들이 있는가?

버전 관리 시스템은 파일 변화를 시간에 따라 기록했다가 나중에 특정 시점의 버전을 다시 꺼내올 수 있는 시스템이다. 그래픽 디자이너나 웹 디자이너도 버전 관리 시스템(VCS - Version Control System)을 사용할 수 있다. VCS 로 이미지나 레이아웃의 버전(변경 이력 혹은 수정 내용)을 관리하는 것은 매우 현명하다. VCS 를 사용하면 각 파일을 이전 상태로 되돌릴 수 있고, 프로젝트를 통째로 이전 상태로 되돌릴 수 있고, 시간에 따라 수정 내용을 비교해 볼 수 있고, 누가 문제를 일으켰는지도 추적할 수 있고, 누가 언제 만들어낸 이슈인지도 알 수 있다. VCS 를 사용하면 파일을 잃어버리거나 잘못 고쳤을 때도 쉽게 복구할 수 있다. 이런 모든 장점을 큰 노력 없이 이용할 수 있다.

1. 로컬 버전 관리

많은 사람은 버전을 관리하기 위해 디렉토리로 파일을 복사하는 방법을 쓴다(똑똑한 사람이라면 디렉토리 이름에 시간을 넣을 거다). 이 방법은 간단하므로 자주 사용한다. 그렇지만, 정말 뭔가 잘못되기 쉽다. 작업하던 디렉토리를 지워버리거나, 실수로 파일을 잘못 고칠 수도 있고, 잘못 복사할 수도 있다.

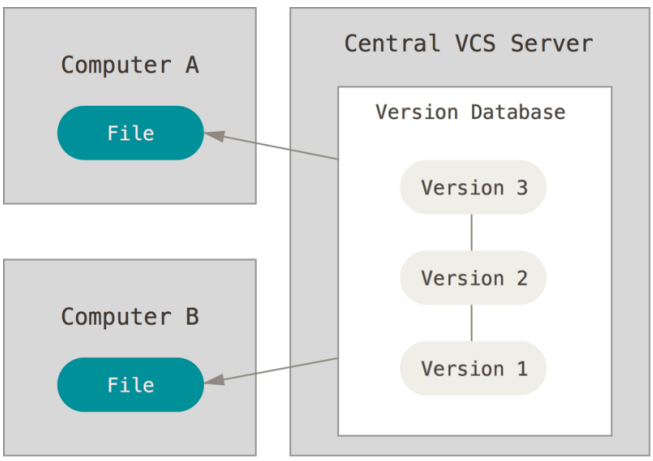


이런 이유로 프로그래머들은 오래전에 로컬 VCS 라는 걸 만들었다. 이 VCS 는 아주 간단한 데이터베이스를 사용해서 파일의 변경 정보를 관리했다.

많이 쓰는 VCS 도구 중에 RCS 라고 부르는 것이 있는데 오늘날까지도 아직 많은 회사가 사용하고 있다. Mac OS X 운영체제에서도 개발 도구를 설치하면 RCS 가 함께 설치된다. RCS 는 기본적으로 Patch Set(파일에서 변경되는 부분)을 관리한다. 이 Patch Set 은 특별한 형식의 파일로 저장한다. 그리고 일련의 Patch Set 을 적용해서 모든 파일을 특정 시점으로 되돌릴 수 있다.

2. 중앙집중식 버전 관리(CVCS)

프로젝트를 진행하다 보면 다른 개발자와 함께 작업해야 하는 경우가 많다. 이럴 때 생기는 문제를 해결하기 위해 CVCS(중앙집중식 VCS)가 개발됐다. CVS, Subversion, Perforce 같은 시스템은 파일을 관리하는 서버가 별도로 있고 클라이언트가 중앙 서버에서 파일을 받아서 사용(Checkout)한다. 수년 동안 이러한 시스템들이 많은 사랑을 받았다.

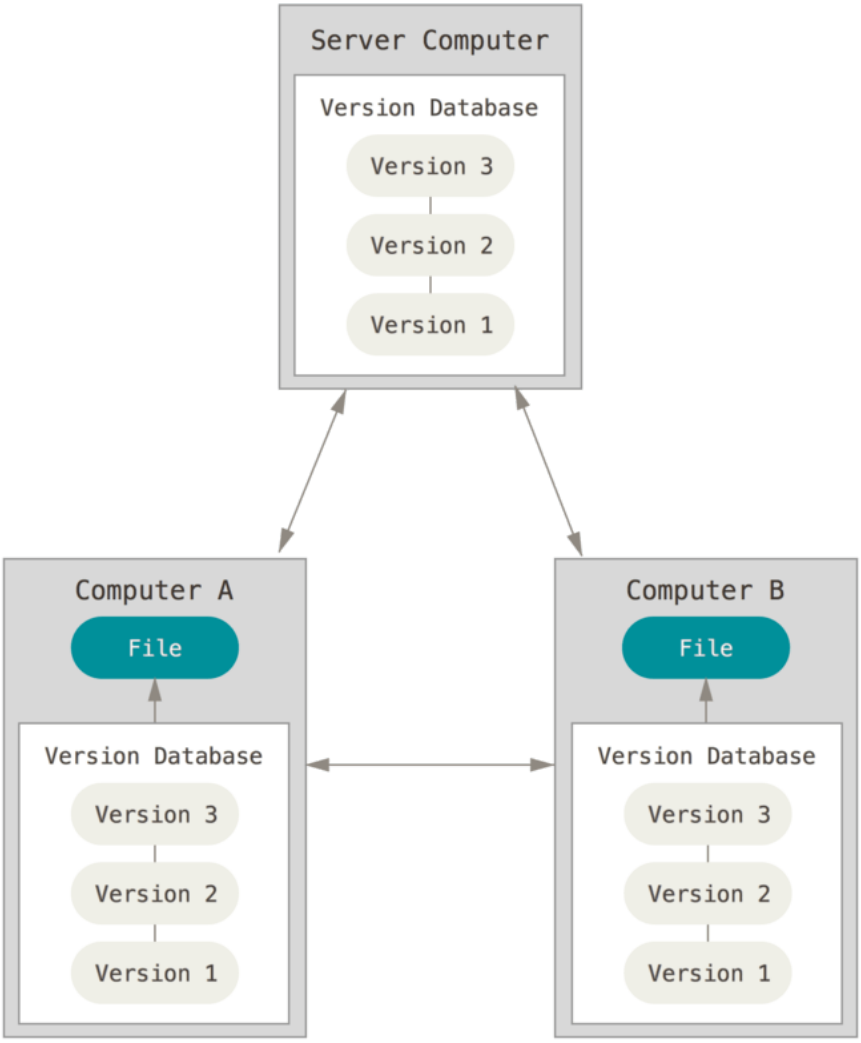


CVCS 환경은 로컬 VCS 에 비해 장점이 많다. 모두 누가 무엇을 하고 있는지 알 수 있다. 관리자는 누가 무엇을 할지 꼼꼼하게 관리할 수 있다. 모든 클라이언트의 로컬 데이터베이스를 관리하는 것보다 VCS 하나를 관리하기가 훨씬 쉽다.

그러나 이 CVCS 환경은 몇 가지 치명적인 결점이 있다. 가장 대표적인 것이 중앙 서버에 발생한 문제다. 만약 서버가 한 시간 동안 다운되면 그동안 아무도 다른 사람과 협업할 수 없고 사람들이 하는 일을 백업할 방법도 없다. 그리고 중앙 데이터베이스가 있는 하드디스크에 문제가 생기면 프로젝트의 모든 히스토리를 잃는다. 물론 사람마다 하나씩 가진 스냅샷은 괜찮다. 로컬 VCS 시스템도 이와 비슷한 결점이 있고 이런 문제가 발생하면 모든 것을 잃는다.

3. 분산 버전 관리 시스템

DVCS(분산 버전 관리 시스템)을 설명할 차례다. Git, Mecurial, Bazaar, Darcs 같은 DVCS 에서의 클라이언트는 단순히 파일의 마지막 스냅샷을 Checkout 하지 않는다. 그냥 저장소를 전부 복제한다. 서버에 문제가 생기면 이 복제물로 다시 작업을 시작할 수 있다. 클라이언트 중에서 아무거나 골라도 서버를 복원할 수 있다. 모든 Checkout 은 모든 데이터를 가진 진정한 백업이다.



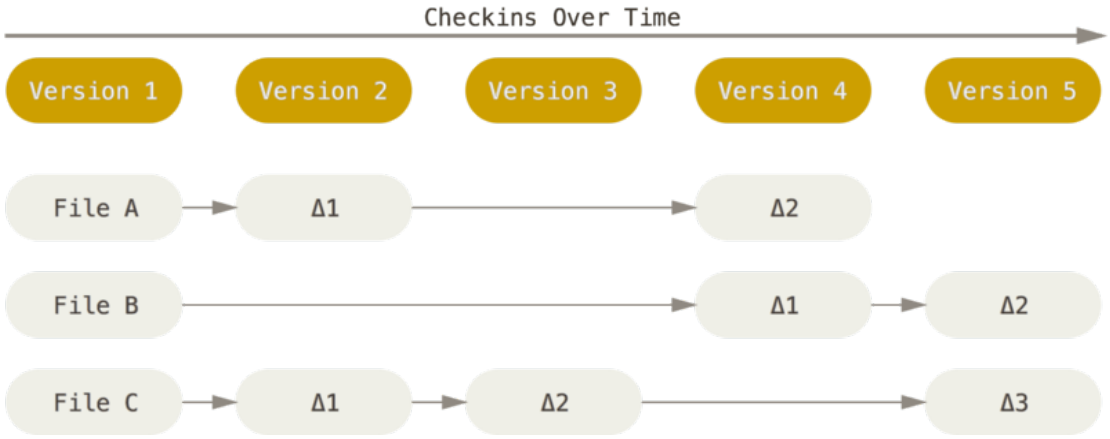
게다가 대부분의 DVCS 환경에서는 리모트 저장소가 존재한다. 리모트 저장소가 많을 수도 있다. 그래서 사람들은 동시에 다양한 그룹과 다양한 방법으로 협업할 수 있다. 계층 모델 같은 중앙집중식 시스템으로는 할 수 없는 Workflow 를 다양하게 사용할 수 있다.

*Git 의 기초

Git 을 배우려면 Subversion 이나 Perforce 같은 다른 VCS 를 사용하던 경험을 버려야 한다. Git 은 미묘하게 달라서 다른 VCS 에서 쓰던 개념으로는 헷갈린다. 사용자 인터페이스는 매우 비슷하지만, 정보를 취급하는 방식이 다르다.

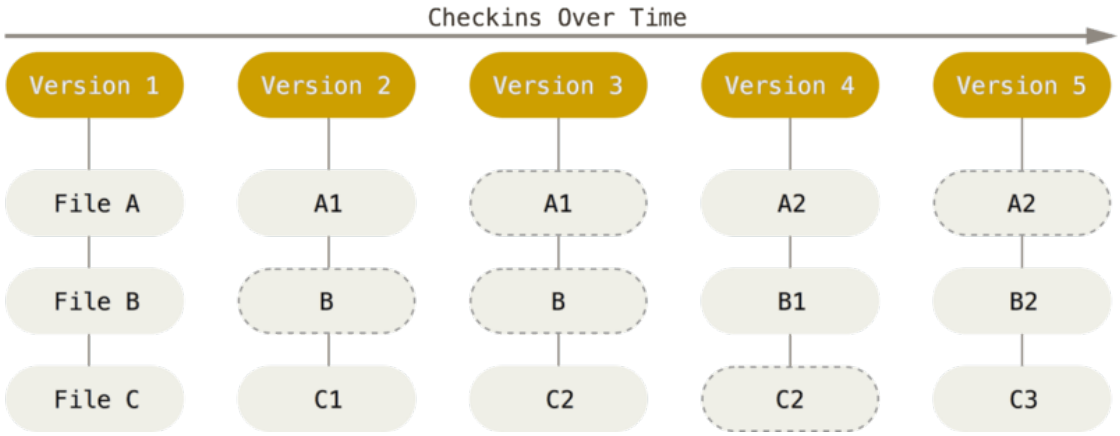
1. 차이가 아니라 스냅샷

Subversion 과 Git 의 가장 큰 차이점은 데이터를 다루는 방법에 있다. 큰 틀에서 봤을 때 VCS 시스템 대부분은 관리하는 정보가 파일들의 목록이다. CVS, Subversion, Perforce, Bazaar 등의 시스템은 각 파일의 변화를 시간순으로 관리하면서 파일들의 집합을 관리한다.



각 파일에 대한 변화를 저장하는 시스템들.

Git 은 이런 식으로 데이터를 저장하지도 취급하지도 않는다. 대신 Git 은 데이터를 파일 시스템 스냅샷으로 취급하고 크기가 아주 작다. Git 은 커밋하거나 프로젝트의 상태를 저장할 때마다 파일이 존재하는 그 순간을 중요하게 여긴다. 파일이 달라지지 않았으면 Git 은 성능을 위해서 파일을 새로 저장하지 않는다. 단지 이전 상태의 파일에 대한 링크만 저장한다. Git 은 데이터를 스냅샷의 스트림처럼 취급한다.



시간순으로 프로젝트의 스냅샷을 저장한다.

2. 거의 모든 명령을 로컬에서 실행

거의 모든 명령이 로컬 파일과 데이터만 사용하기 때문에 네트워크에 있는 다른 컴퓨터는 필요 없다. 대부분의 명령어가 네트워크의 속도에 영향을 받는 CVCS 에 익숙하다면 Git 이 매우 놀라울 것이다. 프로젝트의 모든 히스토리가 로컬 디스크에 있기 때문에 모든 명령을 순식간에 실행된다.

예를 들어 Git 은 프로젝트의 히스토리를 조회할 때 서버 없이 조회한다. 그냥 로컬 데이터베이스에서 히스토리를 읽어서 보여준다. 그래서 눈 깜짝할 사이에 히스토리를 조회할 수 있다. 어떤 파일의 현재 버전과 한 달 전의 상태를 비교해보고 싶을 때도 Git 은 그냥 한 달 전의 파일과 지금의 파일을 로컬에서 찾는다. 파일을 비교하기 위해 리모트에 있는 서버에 접근하고 나서 예전 버전을 가져올 필요가 없다.

비행기나 기차 등에서 작업하고 네트워크에 접속하고 있지 않아도 커밋할 수 있다. 다른 VCS 시스템에서는 불가능한 일이다. Perforce 를 예로 들자면 서버에 연결할 수 없을 때 할 수 있는 일이 별로 없다. Subversion 이나 CVS 에서도 마찬가지다. 오프라인이기 때문에 데이터베이스에 접근할 수 없어서 파일을 편집할 수는 있지만, 커밋할 수 없다. 매우 사소해 보이지만 실제로 이 상황에 부닥쳐보면 느껴지는 차이가 매우 크다.

3. Git 의 무결성

Git 은 데이터를 저장하기 전에 항상 체크섬을 구하고 그 체크섬으로 데이터를 관리한다. 그래서 체크섬 이해하는 Git 없이는 어떠한 파일이나 디렉토리도 변경할 수 없다. 체크섬은 Git 에서 사용하는 가장 기본적인(Atomic) 데이터 단위이자 Git 의 기본 철학이다. Git 없이는 체크섬을 다룰 수 없어서 파일의 상태도 알 수 없고 심지어 데이터를 잃어버릴 수도 없다.

Git 은 SHA-1 해시를 사용하여 체크섬을 만든다. 만든 체크섬은 40 자 길이의 16 진수 문자열이다. 파일의 내용이나 디렉토리 구조를 이용하여 체크섬을 구한다. SHA-1 은 아래처럼 생겼다.

24b9da6552252987aa493b52f8696cd6d3b00373

Git 은 모든 것을 해시로 식별하기 때문에 이런 값은 여기저기서 보인다. 실제로 Git 은 파일을 이름으로 저장하지 않고 해당 파일의 해시로 저장한다.

4. Git 은 데이터를 추가할 뿐

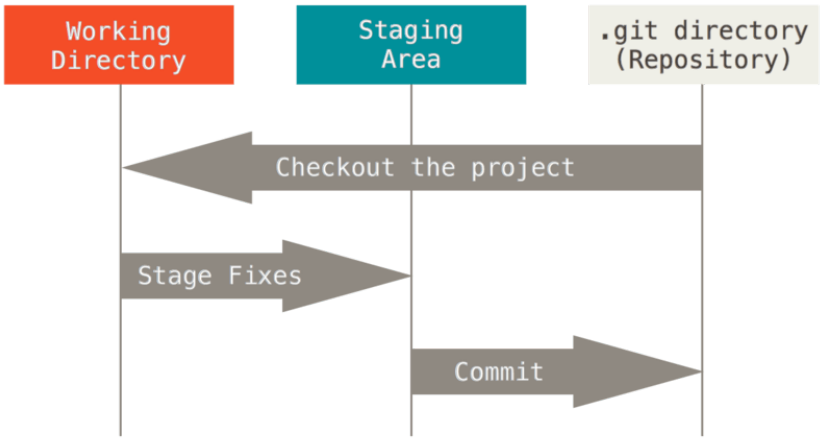
Git 으로 무얼 하든 Git 데이터베이스에 데이터가 추가된다. 되돌리거나 데이터를 삭제할 방법이 없다. 다른 VCS 처럼 Git 도 커밋하지 않으면 변경사항을 잃어버릴 수 있다. 하지만, 일단 스냅샷을 커밋하고 나면 데이터를 잃어버리기 어렵다.

Git 을 사용하면 프로젝트가 심각하게 망가질 걱정 없이 매우 즐겁게 여러 가지 실험을 해 볼 수 있다. “되돌리기”을 보면 Git 이 데이터를 어떻게 저장하고 손실을 어떻게 복구해야 할지 알 수 있다.

5. 세 가지 상태

Git 은 파일을 Committed, Modified, Staged 이렇게 세 가지 상태로 관리한다. Committed 란 데이터가 로컬 데이터베이스에 안전하게 저장됐다는 것을 의미한다. Modified 는 수정한 파일을 아직 로컬 데이터베이스에 커밋하지 않은 것을 말한다. Staged 란 현재 수정한 파일을 곧 커밋할 것이라고 표시한 상태를 의미한다.

이 세 가지 상태는 Git 프로젝트의 세 가지 단계와 연결돼 있다. Git 디렉토리, 워킹 디렉토리, Staging Area 이렇게 세 가지 단계를 이해하고 넘어가자.



워킹 디렉토리, Staging Area, Git 디렉토리.

Git 디렉토리는 Git 이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳을 말한다. 이 Git 디렉토리가 Git 의 핵심이다. 다른 컴퓨터에 있는 저장소를 Clone 할 때 Git 디렉토리가 만들어진다.

워킹 디렉토리는 프로젝트의 특정 버전을 Checkout 한 것이다. Git 디렉토리는 지금 작업하는 디스크에 있고 그 디렉토리 안에 압축된 데이터베이스에서 파일을 가져와서 워킹 디렉토리를 만든다.

Staging Area 는 Git 디렉토리에 있다. 단순한 파일이고 곧 커밋할 파일에 대한 정보를 저장한다. 종종 ``Index``라고 불리기도 하지만, Staging Area 라는 명칭이 표준이 되어가고 있다.

Git 으로 하는 일은 기본적으로 아래와 같다.

1. 워킹 디렉토리에서 파일을 수정한다.
2. Staging Area 에 파일을 Stage 해서 커밋할 스냅샷을 만든다.
3. Staging Area 에 있는 파일들을 커밋해서 Git 디렉토리에 영구적인 스냅샷으로 저장한다.

Git 디렉토리에 있는 파일들은 Committed 상태이다. 파일을 수정하고 Staging Area 에 추가했다면 Staged 이다. 그리고 Checkout 하고 나서 수정했지만, 아직 Staging Area 에 추가하지 않았으면 Modified 이다.