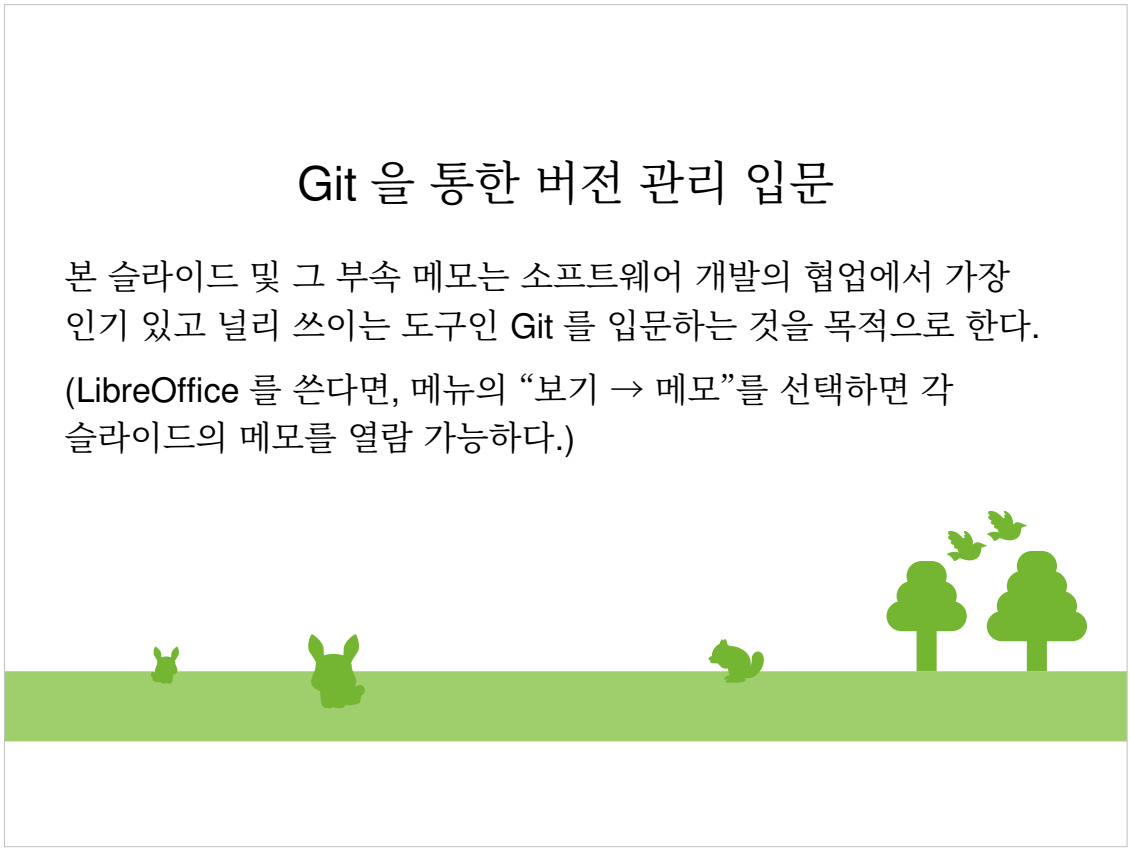


지금 보이는 이 영역이 메모 영역이다.

메모 영역은 본래 강연자가 슬라이드를 발표하기 위한 키 포인트를 요약해둔 부분이다.

발표나 강연을 위해 메모를 함께 인쇄하고자 한다면 인쇄 유형을 ‘메모’로 하여 프린트 하면 이 메모가 포함된 자료를 인쇄할 수 있다.

Git 를 전혀 써본 적이 없는 사람 및 그룹이 Git 를 사용하여 프로젝트를 공유하여 협업할 수 있도록 하는 것이 본 슬라이드의 목적이다.

본 슬라이드는 프로젝트의 구조나 특정 언어에 관한 설명은 되도록 하지 않는다. 어디까지나 Git 의 개념과 Git 의 기본적인 사용 방법을 익히는 것을 목표로 한다.

본 강의는 code monkey 함에 알맞게 디자인되어 있지 **않다**. 즉, (다른 많은 입문 강의와 같이) **따라하기 식으로 구성되어 있지 않다**. “Git (좀 더 정확히는 VCS)을 사용해야겠다”라는 생각을 일으키기 위한 것이지, 이미 Git 을 사용해야 하는 필요성이 충만한 청취자라면 이 강의가 목표로 하는 것을 이미 가지고 있는 것이다. 이 강의는 주로 “이메일이나 메신저로 파일 공유하면 되는데 왜 굳이 Git (혹은 다른 VCS)을 사용해야 하는지 모르겠다”고 생각하는 사람에게 호소하기 위한 방식으로 구성되어 있다.

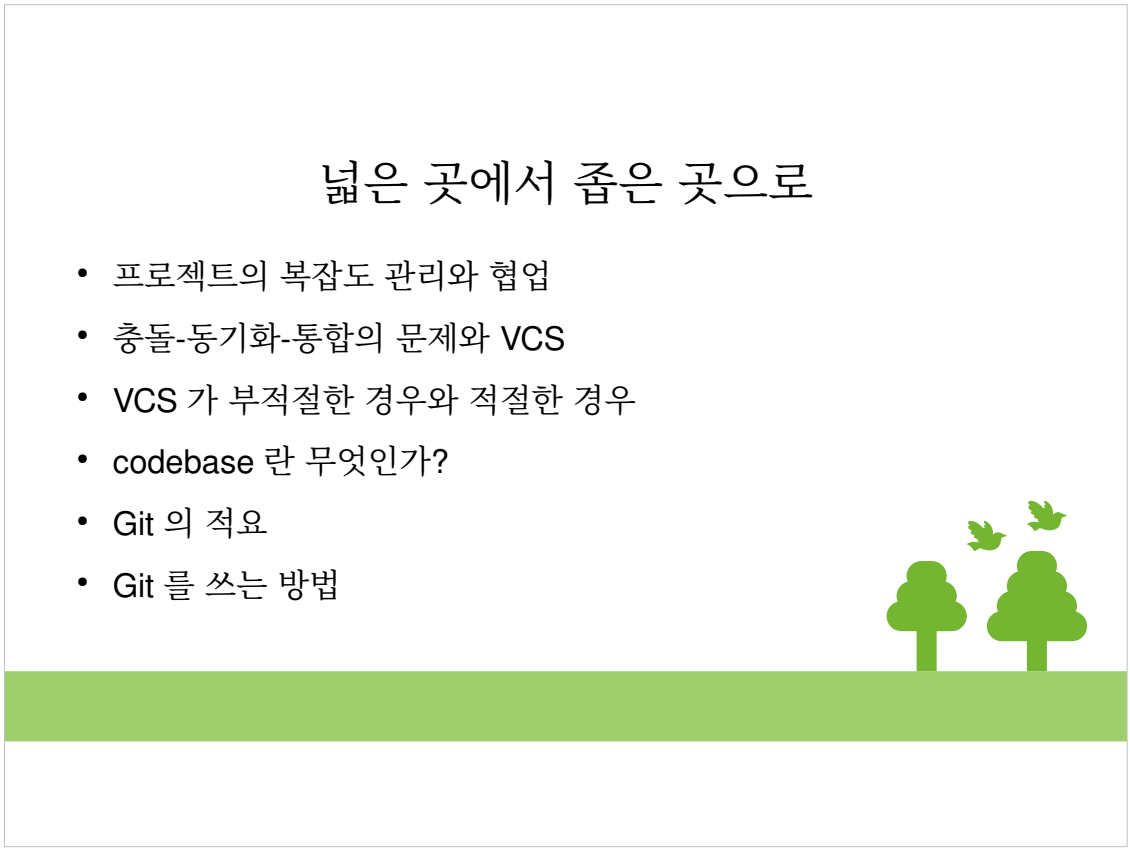
저작권

본 슬라이드의 저자인 김태엽은 본 슬라이드를 퍼블릭 라이선스로 공개함을 명시한다.

사용자는 본 슬라이드를 임의로 가공, 배포, 활용 가능하다.

본 슬라이드의 활용에 따른 어떠한 유형/무형의 불이익 및 금전적 손해에 대해서 저자는 책임지지 않는다.

저작물이 활용되기 위해서는 저작권이 반드시 명시되어 있어야 하며, 저작권이 명시되지 않은 저작물은 기본적으로 사유 재산으로 취급되어 사용 방법이 제약된다. 이는 이러한 간단한 강의 자료에도 적용되는 사실이며, 본 슬라이드의 자유로운 활용을 할 수 있는 형태의 저작권을 명기한다.



Git 를 설명한다고는 하지만, Git 을 무작정 보는 것보다는 그 주위의 넓은 범위에서 들어가는 것이 이미 알고 있는 지식에서 새로운 지식을 도입하는 데 도움이 된다고 믿는다. 넓은 곳에서 좁은 곳으로 들어가자.

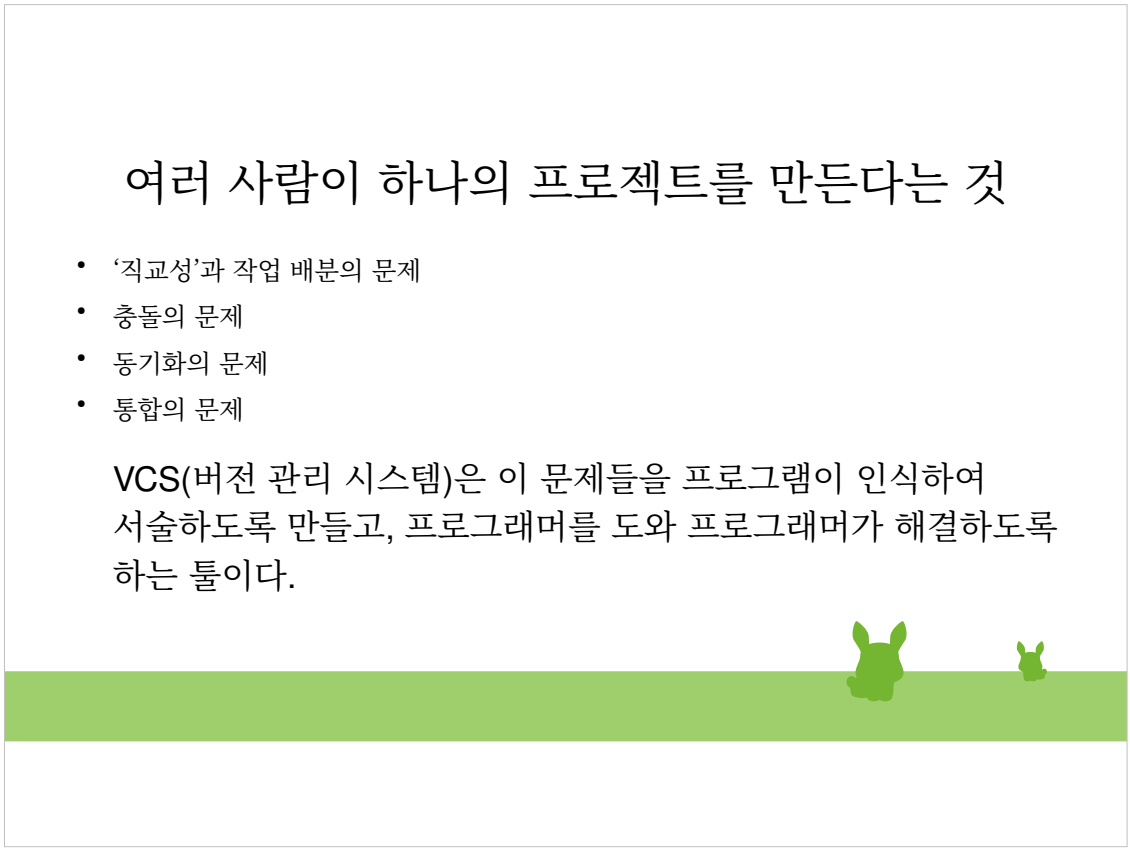
이곳에서는 청취자가 기본적으로 프로그래밍 프로젝트를 어렴풋이나마 알고 있다고 가정한다. 우리는 그곳에서 처음 프로젝트 복잡도 관리와 협업에 대해서 논한다.

그리고 협업에서 자주 발생하는 충돌-동기화-통합의 문제에 대해서 도입한다. 이 문제들을 해결하는 기존의 방법을 소개하고, VCS 라는 일반적인 개념에 대해서 논한다. Git 은 VCS 의 일종이다.

VCS 는 엘릭서가 아니며 만능의 도구가 아니다. 사용이 적절한 곳과 그렇지 않은 곳이 있다. 무딘 도끼로도 연필을 깎을 수 있지만 잘 깎을 수는 없듯, VCS 는 사용이 적절한 대상과 그렇지 않은 대상이 있다.

그리고 그 다음에는 VCS 가 관리하는 대상인 codebase 에 대해서 논한다. Git 로 관리하는 대상은 codebase 의 history 이며, 무엇이 codebase 인지 정하는 것은 궁극적으로 프로그래머의 결정에 달려있다.

이것들을 다 도입하고 나면 드디어 Git 으로 들어간다. Git 은 수많은 VCS 중 하나일 뿐이다.



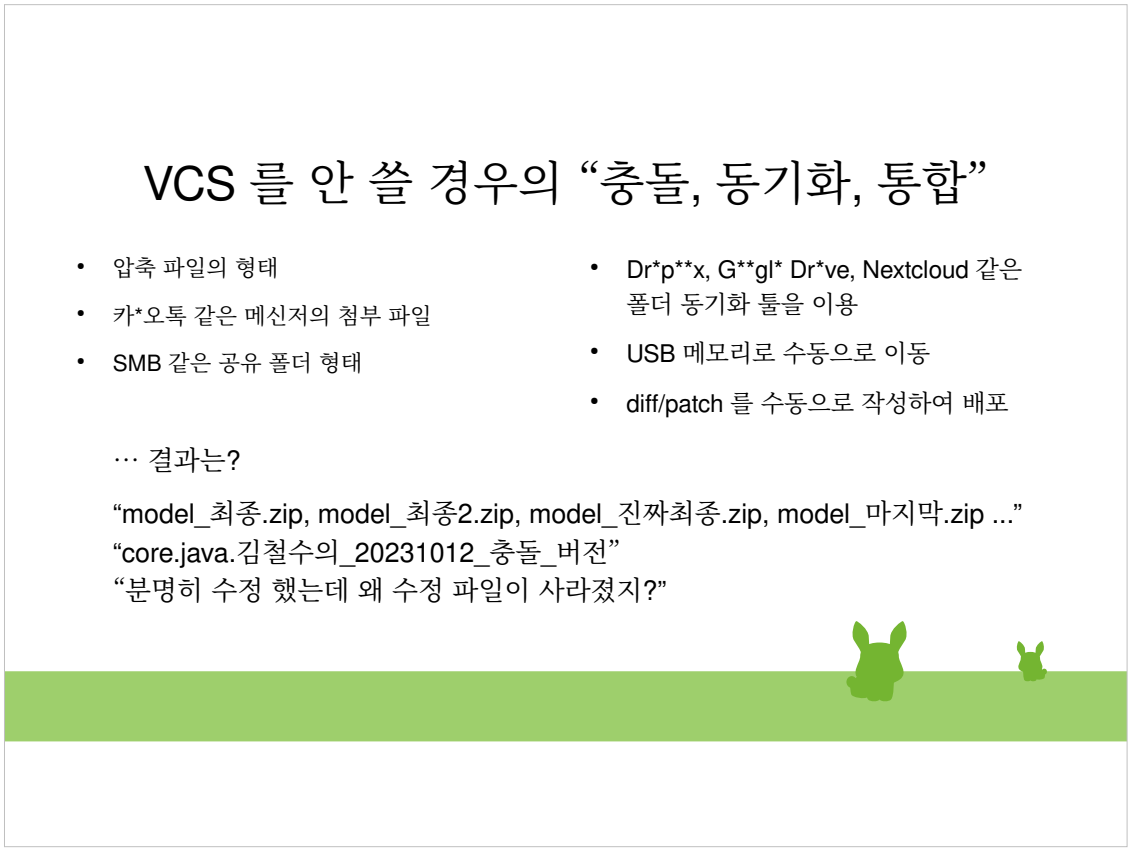
사공이 많으면 배가 산으로 가는 것처럼 프로젝트는 어떤 정확한 목표가 없으면 중구난방하게 되기 쉽다. 구체적 목표가 있다 하더라도 일을 나눠서 배분한다는 것은 쉽지 않은데 그 이유는 많은 일은 직교성이 떨어지는 형태로 존재하기 때문이다.

직교성이란, **문제를 서로 독립적이고 무관한 부분으로 나눌 수 있는 성질**을 이른다.

예를 들어서 서버-클라이언트 모델로 뭔가 구현한다고 할 때, 서버가 완성이 안되면 클라이언트가 일을 구현할 수 없거나 하고, 그 역의 경우도 자주 있다. 이걸 나누기 위해서 호출 방식을 미리 설계해서 쓰거나 하는 경우도 있긴 하다만 결국은 통합되어 빌드되는 것이 전제되는 것이기에 직교성이 떨어진다.

이것을 해결하기 위해서는 프로젝트 참여자가 프로젝트 전체를 통일된 방식으로 내려받아서 동기화 하고 빌드를 할 수 있고, 수정을 통한 기여를 통일된 방식으로 프로젝트에 반영할 수 있어야 한다. 그리고 Git 같은 버전 관리 시스템(VCS)는 그 방법을 제공한다.

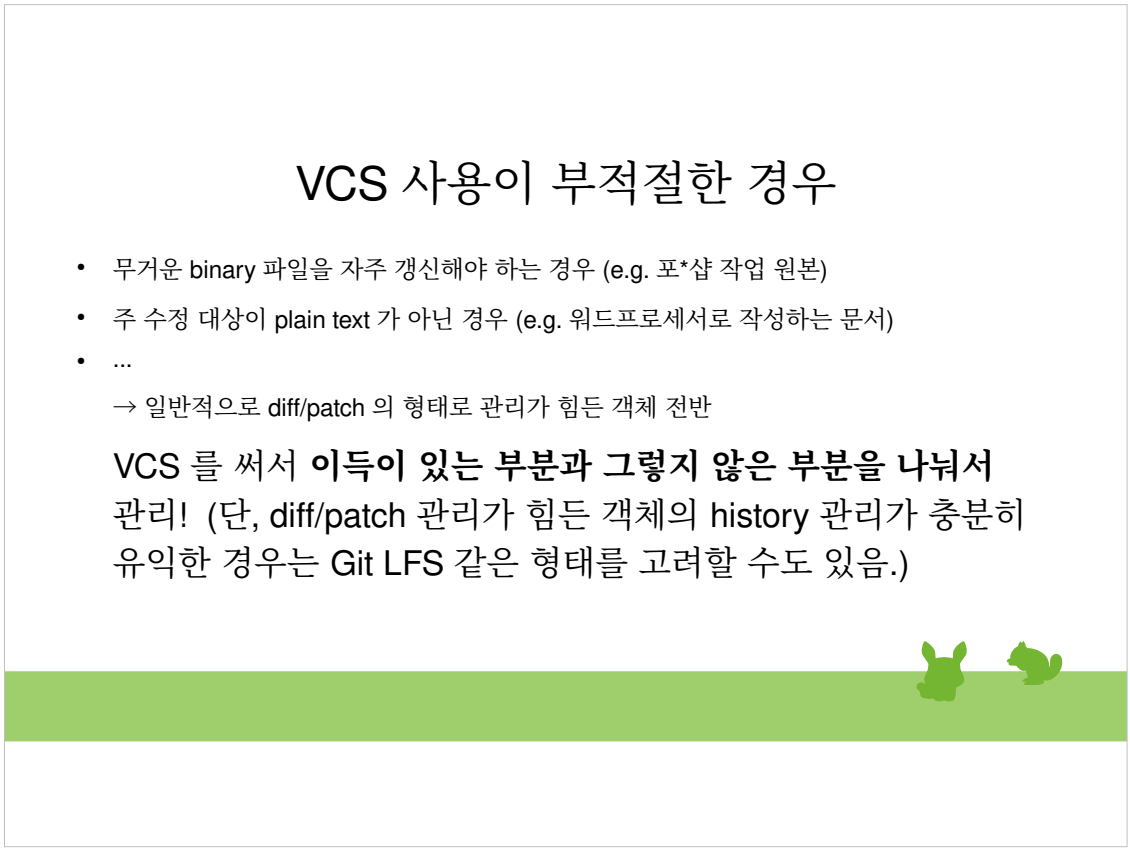
충돌의 문제, 동기화의 문제, 통합의 문제는 각각 VCS 에서 말하는 confliction, pull/push, merge 로 표현되는 것들이다. 대부분의 VCS 는 프로젝트에서 이 문제가 실제로 발생할 경우 그 형태를 규정하고 해결할 수 있도록 구성되어 있다.



VCS 를 쓰지 않는다는 것이 뜻하는 것은 프로젝트의 코드 및 내용 변화를 모두 스스로 관리하고 따라 가겠다는 것을 의미한다. 간단한 1인 프로젝트라면 이 형태로도 관리가 될 수 있긴 하지만, 협업에서는 다른 사람이 직접적으로 같은 코드를 수정해버리는 경우도 있을 수 있고, 다른 사람이 수정한 다른 부분의 코드에 의존하는 코드가 있어서 빌드가 망가지거나 하는 경우가 얼마든지 있을 수 있다.

여러 사람이 개발하는 프로그램에서 폴더 동기화 툴을 소스 코드 관리에 쓰는 건 재앙적인 실수일 가능성이 높다.

그나마 diff/patch 를 수동으로 작성하여 배포하는 방식은 작은 프로젝트에서는 그나마 작동하는 방법이다. Linux kernel 도 초창기에는 diff/patch 를 이메일로 주고 받는 형태로 개발되었고, 지금도 Git 는 diff/patch 사항을 이메일로 내보내서 패키 가능하도록 하는 기능을 제공한다. 어떤 의미로는 Git 는 이 diff/patch 를 수동으로 작성하는 방법의 발전판이라 할 수 있다.



할 수 있는 것보다 할 수 없는 것을 먼저 말하는 것이 연역적 논리의 기본이듯, 우선 VCS 사용이 부적절한 경우로 논의를 시작하자.

원칙적으로 VCS 로 관리하는 파일은 코드베이스(codebase)여야 하며, plain text 여야 한다. 그림이나 영상, 워드 프로세서 작업 파일 등은 직접 만드는 파일이긴 하지만 diff/patch 의 형태로 관리하기 힘든 파일이라 VCS 관리가 적절하지 않다. 그런 파일들은 다른 형태의 동기화가 더 적절하다.

예를 들어서 기계 학습이 프로젝트의 일부로 필요한 경우, 학습 결과 모델 파일은 무거운 binary 파일이거나 자동 생성된 파일이다. 이런 파일은 VCS 로 관리하기에는 부적절하며, 그런 파일의 관리는 일반적으로 VCS 와 분리하여 생각할 필요가 있다.

원칙적으로 생성물은 VCS 로 관리하지 않는다. 단, 생성이 오래 걸리거나 생성 비용이 큰 것들의 경우는 VCS 가 아닌 다른 방법으로 동기화를 하기도 한다. 그 동기화에는 Rsync, Unison 같은 툴이 이용될 수 있다. 참고로 이 문서 자체도 LibreOffice 로 작성하고 있는데, 이런 ODP 문서의 관리는 Git 이 적절하지 않기 때문에 Unison 으로 동기화를 하고 있다.

VCS 사용이 적절한 경우

주로 plain text 로 구성되어 있는 codebase

→ 프로그램 코드, 문서화!

- history 를 효율적으로 관리할 수 있는 객체 → diff/patch
- diff/patch 를 유의미하게 적용할 수 있는 객체 → plain text
- 생성물이 아닌 프로그래머가 작성한 객체 → codebase

앞서 VCS 사용이 부적절한 경우의 논의를 고려한다면, 결국 VCS 사용이 적절한 경우는 대표적으로 plain text 로 구성되어 있는 codebase 가 된다.

diff/patch 라는 것은 차분(differential)의 개념을 말하며, 차분의 개념이 효과적으로 적용이 가능한 대상이 VCS 의 관리에 적절한 대상이다. diff/patch 의 효과적 적용이 가능한 것과 plain text 자체는 필요충분조건은 아니지만, 사람이 의식적으로 다룰 수 있는 diff/patch 의 가장 대표적인 대상은 plain text 인 것은 사실이다. 두 plain text 는 서로 겹치는 부분과 서로 다른 부분(즉, diff)을 효과적으로 계산해낼 수 있고, VCS 는 history 의 관리를 위해 이 계산을 이용한다. 구체적으로는 이전 상태와 현재 상태의 diff 를 계산하여 patch 형태로 역사를 보존하는 형태다.

부연하자면, diff/patch 는 비록 plain text 가 대표적인 것이긴 하지만, 꼭 plain text 여야 할 필요는 없다. 다만 이 주제는 좀 더 높은 단계의 추상화를 요구하기 때문에 현재 논의하기에는 부적절하다 생각되기에 생략한다.

사실 두 plain text 에 대한 diff 의 계산은 일의적이지 않다. 그 최적화는 실제로 꽤 까다로운 알고리듬 문제이기도 하다만, 실제 VCS 사용에서 크게 영향을 끼치는 부분이 아니기에 생략한다.

궁극적으로 어떤 파일을 VCS 로 관리하고 안 하고는 프로그래머의 결정에 달렸다. 어떤 파일은 VCS 로 관리하면 절대 안되고 어떤 파일은 꼭 VCS 로 관리해야 하거나 하는 절대적인 법칙은 없고, 어느 정도의 가이드라인 이 있을 뿐이다.

codebase 란 무엇인가?

- 일반적으로는 생성물이 아닌 원시코드
- VCS 의 history 로 기록될 필요성이 있는 객체
- 프로젝트에 필수적인 부품

  codebase 의 정의는 프로그래머나 프로젝트에 따라서 달라질 수 있으며 궁극적으로는 “무엇을 프로젝트의 일부로 다룰 것인가”의 결정에 따른다.

VCS 에서 codebase 라는 개념은 history 를 가지고 관리하고자 하는 대상 그 자체를 이른다. 흔히 source code 라는 이름으로 말해지곤 하는 것이 codebase 다만, 실제론 약간 다르다. source code 는 또 다른 source code 에서 생성될 수 있기에 (e.g. CoffeeScript → JavaScript) 그런 경우는 일반적으로 원시코드가 codebase 가 될 것이다.

그렇다고 원시코드가 항상 codebase 는 아니고, codebase 라고 해서 꼭 원시코드여야 하는 것도 아니다. **생성물도 codebase 가 될 수 있고, diff/patch 를 쓸 수 없는 객체도 codebase 의 일부가 될 수 있다**.

구체적으로 boilerplate code, code snippet 같은 계열은 일반적으로 **IDE 에서 생성하는 코드**인데, 이로부터 생성된 코드는 엄밀히는 그것을 생성하는 행위나 명령이 원시 행위인데도 불구하고, 그 생성을 위한 원시 행위를 표현하는 원시 코드가 아닌 일반적으로 생성된 결과물인 코드쪽이 codebase 의 일부로 취급된다. (이때, boilerplate code 라는 것은 코드를 작동시키기 위해서 기본적으로 필요하거나 하여 자주 사용하기에 주로 템플릿으로 생성하여 사용하는 코드를 말하며, Java 같은 언어가 특히 그런 코드가 자주 등장한다. code snippet 은 특정 기능을 구현할 때 자주 쓰는 패턴 같은 것을 조각(snippet)의 형태로 만들어 놓고 불러와서 바로 붙여넣기 해서 쓰는 코드를 말한다.)

boilerplatce code 는 결과적으로는 반복되는 논리이기 때문에 실제로 원시코드를 쓰도록 줄이는 게 불가능하진 않겠지만, 항상 타협적이다.

프로젝트의 로고 그림이나 빌드에 필수적인 blob 파일 등은 사실상 효과적으로는 diff/patch 를 쓸 수 없지만 편의상 codebase 의 일부로 포함하기도 한다.

VCS 바깥에서 관리되는 codebase

- diff/patch 적용이 곤란하지만 codebase 인 것들 (주로 binary 파일들)
  - 프로젝트에서 사용되는 그림, 소리, 영상 등
  - 기계 학습의 모델 결과물 파일
- 비록 VCS 로 관리되지는 않지만 “충돌-동기화-통합”을 처리해야 하는 대상들

Rsync, Unison, Git LFS 같은 동기화 툴을 사용하여 통합!

VCS 가 주로 관리하는 파일은 주로 plain text 로 구성된 codebase 라고 하였다. 이것은 diff/patch 가 plain text 에서 가장 효과적으로 작동하기 때문이다. Git 를 포함한 대부분의 VCS 는 binary 파일을 다루는 것이 능숙하지 않다.

예를 들어서 프로그램을 빌드하여 나오는 프로그램 실행 바이너리는 codebase 에 보통 포함시키지 않는다. 그것들은 보통 release 라고 해서 따로 관리한다.

현실적으로, plain text 가 적절하지 않은 매체이면서, 그런 매체를 프로젝트의 일부로 다뤄야 하는 경우는 많다.

예를 들어서 프로그램을 만들면서 GUI 를 구성하는 일부로 프로그램 아이콘이나 버튼 같은 것을 그림 파일로 써야 할 수 있다. 그 정도 작은 binary 라면 작은 비효율을 감내하고 VCS 를 써도 될 것이다.

그런데 기계 학습 같은 경우에 모델의 결과물 파일 같은 큰 바이너리 파일은 VCS 를 쓰면 너무 비효율적이게 된다. 하지만 그 파일은 그대로 있는 것이 아니고, 변화하며, 업데이트 되어야 하는 파일이기에 부분적으로 역사 관리가 필요할 수 있다. 즉, 충돌-동기화-통합을 처리해야 하는 것이다.

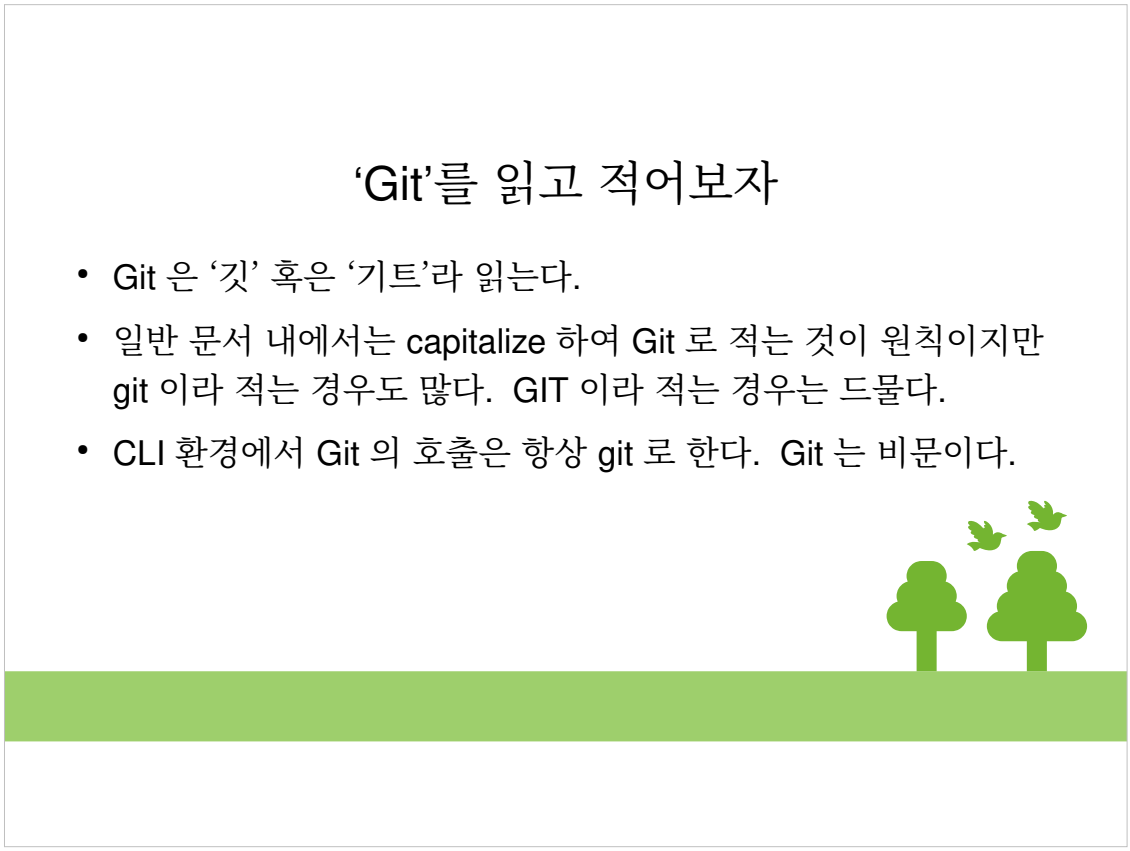
그런 파일들은 VCS 의 관리 범위에서 분리하여, Rsync, Unison 같은 동기화 툴을 사용해서 동기화를 하면 편리하다. 이곳에서는 그 툴을 다루진 않는다.

Git LFS 는 Git 저장소 내부에서 큰 파일을 다루기 쉽게 하는 확장인데, 작은 binary 를 잔뜩 다루는 프로젝트라면 Git LFS 의 사용도 생각해볼 수 있다.

Git 는 수많은 VCS 중 하나로, Linux kernel 의 창시자인 Linus Torvalds 가 만든 VCS 다. 본래 Linux kernel 의 개발에는 BitKeeper 라는 VCS 가 사용되고 있었는데, 라이선스 문제로 인한 문제 때문에 Linus Torvalds 가 결국 직접 새로운 VCS 를 만들자고 결정하게 되어 만든 것이 Git 다. Linux kernel 도 걸작이지만, Git 는 개발자를 대상으로 본다면 Linux kernel 보다도 더 널리 쓰이는 킬러 프로그램으로 이름이 높다. 한 개발자가 이 정도로 유명한 프로그램을 두 개 이상을 만드는 경우는 매우 드문데, Linus Torvalds 는 그것을 해낸 사람이다.

VCS 의 선택은 고민할 필요가 없다. 기존의 프로젝트가 이미 VCS 를 쓰고 있다면 그걸 쓰는 게 최고다. 물론 프로젝트 규모가 크고 새로운 방식의 관리가 필요하다고 판단된다면 Git 로 이행하거나 하는 걸 고려할 수도 있겠지만 그게 아니라면 그냥 기존의 것을 배워서 쓰는 게 최고다. 어차피 각 프로젝트마다 정해진 관습적인 push/pull 방법이 존재할 것이다. 그것을 배우고 실행하라.

새로 시작하는 프로젝트라면 고민할 것 없이 Git 를 쓰면 된다. 물론 앞서 말했듯 VCS 로 관리할 부분과 그렇지 않은 부분을 나눌 필요는 있다.



Git 는 깃이라 읽거나 기트로 읽는다.

적을 때는 Git 이나 git 이라 적는다. GIT 이라고는 잘 안 적는다. 가끔 C 프로그래밍 언어나 쉘 스크립트 같은 것에서 매크로 혹은 환경 변수를 적거나 해야 할 때 GIT 이라 적는 경우가 있긴 한데, 그건 일반적으로 관습이 통용 표기보다 우선되기 때문이다.

Git 의 CLI 명령 호출은 git 로 한다. Git 이나 GIT 은 비문이다.

여기까지 오느라 매우 길었다. 서론이 긴 이야기는 싫어하는데 내가 그런 이야기를 적는 걸 보니 내가 뭔가 잘못하고 있는 게 아닌가 싶다.

Git 는 좋다. 쓰자.

프로그래밍은 거의 대부분 문서화와 씨름이다. 그리고 좋은 프로그램은 좋은 문서화가 따라 온다. Git 은 좋은 문서화가 잔뜩 있다.

Windows 에서 Git 을 쓸 경우는 좀 불편할 수 있다. Unix 환경에서는 Git 의 문서화를 찾고자 할 때는 바로 터미널에서 man git-commit 처럼 모르는 명령어를 바로 찾을 수 있는데, Windows 에서는 이게 기본적으로는 안된다. 결과적으로 Windows 에서 Git 을 쓸 때는 검색 엔진을 찾아가며 문서화를 찾아야 할 가능성이 높다. Unix 에서는 로컬에서 다 해결되는 것이 Windows 에서는 인터넷이 필요한 사안이 된다는 것은 골치 아픈 일이다.

Git CLI 와 다른 Git frontends

- Git 의 CLI 명령은 Git 의 기본이면서 최소한의 조작 방법이다.
- IDE 를 쓸 경우 IDE 가 제공하는 Git frontend 를 통해 사용한다.

Git 의 CLI 명령은 어떤 경우라도 적용 가능하며, 다른 Git frontend 들은 내부적으로는 사실상 Git CLI 명령을 실행하고 있다.

Git 는 GUI frontend 도 있긴 한데, 기본은 CLI 로 배우고 그걸 기반으로 편안한 다른 frontend 를 찾는 것을 추천한다. 대부분의 IDE (Eclipse, VSCode 등)는 Git 를 위한 확장을 제공하며, 에디터 종류도 내부 확장으로 자주 지원한다(e.g. Emacs 의 Magit).

Git 을 CLI 로 사용하는 것을 선호하는 사람도 많다만, 어느 정도 Git 에 익숙해진 이후에는 보통은 즐겨 사용하는 IDE 나 에디터의 확장을 이용해서 Git 을 사용한다.

Git 의 CLI 명령은 쉘 스크립트를 다룰 줄 알면 자동 동기화 시스템 같은 형태로도 만드는 게 가능하다.