Git Magic

Ben Lynn

2007 년 8 월

서문

Git 은버전관리계의스위스아미나이프정도로보면됩니다. 아주유연하고믿을수있지만그만큼배우기는어려울수도있는이 Version Control System 을마스터해봅시다!

Arthur C. Clarke 는충분히발전한기술은마술과같다고말하였습니다. Git 도마찬가지입니다. 초보자들은 Git 이어떻게돌아가는지알필요가없으며 Git 이라는간단한장치가어떻게친구들과적들을놀라게하는지만알면됩니다 ^_^.

세부사항들을설명하는대신에, 우리는몇몇기능들의대략적인설명을하려합니다. 여기에설명된기능들을자주사용하다보면각각의명령어들이어떻게작동하는지알게될것입니다. 그리고그명령어들을적용하여새로운일들을해낼수있겠지요.

- Simplified Chinese: by JunJie, Meng and JiangWei. Converted to Traditional Chinese via `cconv -f UTF8-CN -t UTF8-TW`.
- French: by Alexandre Garel, Paul Gaborit, and Nicolas Deram.
- German: by Benjamin Bellee and Armin Stebich; also hosted on Armin's website.
- Italian: by Mattia Rigotti.
- link:/~blynn/gitmagic/intl/ko/ [Korean]: by John J. Han; also hosted on John's website.
- Polish: by Damian Michna.
- Brazilian Portuguese: by José Inácio Serafini and Leonardo Siqueira Rodrigues.
- Russian: by Tikhon Tarnavsky, Mikhail Dymskov, and others.
- Spanish: by Rodrigo Toledo and Ariset Llerena Tapia.
- Ukrainian: by Volodymyr Bodenchuk.
- Vietnamese: by Trần Ngọc Quân; also hosted on his website.
- Single webpage: CSS 없는 HTML 버전.

- PDF file: 프린팅버전.
- Debian package, Ubuntu package: 이웹사이트의사본. Handy when this server is offline.
- Physical book [Amazon.com]: 64 pages, 15.24cm x 22.86cm, black and white. 전기가들어오지않을때유용.

고맙습니다!

많은분들께서번역에힘써주셔서저는어떻게몸둘바를모르겠습니다. 이분들을통해더많은독자들을만날수있어서정말기쁘고감사드립니다.

Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar, Frode Aannevik, Keith Rarick, Andy Somerville, Ralf Recker, Øyvind A. Holm, Miklos Vajna, Sébastien Hinderer, Thomas Miedema, Joe Malin, Tyler Breisacher, Sonia Hamilton, Julian Haagsma, Romain Lespinasse, Sergey Litvinov, Oliver Ferrigni, David Toca, , Joël Thieffry, and Baiju Muthukadan 수정및편집에힘써주셧습니다..

François Marier 는 Daniel Baumann 이개발한 Debian 패키지를관리합니다.

고마워해야할사람들이많지만은여기에다쓸수는없는노릇입니다.

그래도만약에제가이웹사이트에실수로이름을개제하지않았다면연락을주시거나패치를만들어주세요!

라이센스

도입부

Github 에대해설명하기쉽게비유법을사용하여버전관리시스템 (Version Control System; 이하 VCS) 를설명해보려합니다. 제가하려는설명에비해덜정신나간버전의설명을원하시면

http://en.wikipedia.org/wiki/Revision_control 를방문하시길권장합니다.

” 일하는것” 은곧” 노는것”

저는거의평생을컴퓨터게임을하며지냈습니다. 그에반해, 어른이되서야 VCS 를사용하기시작했지요. 이런건저혼자가아닐것이라생각하니, Git 을게임에비유하며설명하는것이 Git 을이해하는데도움이될것이라생각합니다.

자, 이제코드나문서를편집하는작업이게임을하는것과같다고생각해보세요. 편집을마친후에는세이브하고싶겠지요? 그렇게하기위해서는당신의든든한코딩에디터에서 세이브버튼을누르면될것입니다.

그러나단순히’ 세이브’ 를누르는것은예전세이브를덮어쓰는결과를초래하죠. 세이브슬롯이한개밖에없는옛날구형게임을생각하면됩니다. 다시말하자면, 세이브를할수는있지만, 예전세이브포인트로돌아갈수없는것입니다. 마치, 게임진행하다가정말재미있는파트에적재적소에맞게세이브를해놓았는데그포인트로다신돌아갈수없다는것이죠. 좀더심하게말하자면, 절대로깰수없는보스앞에서세이브를한당신은그상태에평생머무르게될수도있다는것입니다. 그럴경우에는아주처음부터다시시작해야된다는말이되겠지요.

버전관리

코드등을편집시, 다른이름으로저장아니면사본을다른디렉토리에카피떠놓는방법등을이용해옛버전의코드들을보존할수는있습니다. 컴퓨터용량을더욱효율적으로사용하기위해서압축을할수도있죠. 이것은참원시적인버전컨트롤방법입니다. 컴퓨터게임은이런과정에서이미발전해나간지오래되었지요. 요즘게임들은여러개의세이브슬롯에시간을기록해가며세이브를영리하게해냅니다.

이문제를좀더꼬아서바라봅시다. 당신이어떤프로젝트나웹사이트를구성하는소스코드와같이여러개의파일을보유한다고가정합시다. 현버전의프로젝트/웹사이트를세이브하고싶다면모든디렉토리를기록해야한다는번거로움이있겠지요. 일일이그수많은버전들을수동으로관리한다는것은그리효율적이지않을겁니다. 컴퓨터용량도더불어많이사용하게될꺼고요.

어떤컴퓨터게임들은정말로모든디렉토리를각개관리하는형식으로게임을세이브하기도합니다. 이런게임들은이런불필요하게세부적인사항들을게이머들이보지못하게하고간편한인터페이스를통해게이머들이세이브파일들을관리할수있게해둡니다.

VCS 는이런컨셉과그리다르지않습니다. VCS 들은파일디렉토리들을관리하기에아주편한인터페이스로구성되어있습니다. 원하는횟수만큼세이브를할수있고, 원하는세이브포인트를특정지어불러오기를실행할수도있습니다. 그리고컴퓨터게임들과는다르게용량을효율적으로사용하는데에는탁월한성능을보여주죠. 대부분의어떤코드의버전을바꿀때에는소수의파일들만살짝바꾸게되죠. 코드자체가아주많이바뀌는경우는드뭅니다. 디렉토리전체를세이브하는것보다는버전과버전사이의차이를세이브하는것이용량을효율적으로쓰는 VCS 의비밀입니다.

분산제어

여러분이어려운컴퓨터게임을한다고생각해보세요. 너무어렵기때문에전세계의프로게이머들이팀을구성해이게임을끝내보겠다고합니다. 게임을빨리끝내는것에초점을두는스피드런방식의게임스타일이현실적인예시이지요: 각기다른특기를가지고있는게이머들이한게임안에서각자자신있는부분을담당함으로써성공적인결과를만들어내는것을예로들어봅니다.

어떻게시스템을구축해두어야게이머들이서로의세이브파일들을쉽게업로드하거나, 바통을이어받을수있을까요?

프로그래밍프로젝트들은예전에중앙집중식 VCS 를사용하였습니다. 한개의서버가모든세이브파일을저장했었지요. 그서버외에는아무것도그세이브파일들을관리할수없었습니다. 게임으로말하자면, 게이머들은각자의게임기에몇개의세이브파일들을가지고있었고, 게임을다른사람으로부터이어받아진행하고싶을때에는, 모든세이브파일들이저장되어있는중앙서버에서파일들을다운로드받은후, 게임을좀하다가, 다시다른게이머들이진행할수있게그서버에업로드해놓아야합니다.

만약에어떤한게이머가예전에세이브해두었던오래된파일을불러오고싶다면어떻게될까요? 현재최신의세이브시점은누군가게임의전단계에서다음단계에필요한아이템을주워오지않아서아무리잘해도게임을진행할수없는상태로저장이되어있을지도모르고, 그런게아니라면그들은아마도세이브파일두개를비교하여한특정게이머가얼마나진행을하였는지알고싶어할지도모릅니다.

예전세이브파일을불러오고싶은이유는여러가지일수있습니다, 그러나방법은한가지일수밖에없지요. 중앙서버에서불러오는방법말입니다. 더많은세이브파일을원할수록서버와의통신이더잦아질수밖에없지요.

(Git 을포함하여) 새로운세대의 VCS 들은분산제어를기본으로합니다. 예전의중앙관리방식의보편화된방식이라고생각하면되지요. 한게이머가서버로부터 (가장최신) 세이브파일을받는다해도그하나만받게되는것이아니라모든예전버전의세이브파일까지도같이받게되는겁니다. 마치중앙서버를각자의컴퓨터에미러링한다고보시면됩니다.

그렇기에처음에는 Git 을셋업할때시간이많이걸릴수있습니다. 특히, 그세이브파일이오래되었고, 아주긴역사를가지고있다면말이지요. 그러나이것은길게보면아주효율적인방법입니다. 이방법을통해즉시이득을볼수있는점을따진다면, 예전세이브파일을원할때중앙서버와교신을하지않아도된다는점이지요.

멍청한미신

사람들이분산제어시스템에대해일반적으로오해하는게, 분산제어시스템은공식적인중앙저장소가필요한프로젝트에는적합하지않다고생각하는것입니다. 이것은말도안되는오해이지요. 이오해는누군가의사진을찍는다는것은그피사체의영혼을같이담아버린다는말도안되는논리와같습니다. 다시말하면, 중앙저장소의파일을카피하여분산제어하는것이중앙저장소의중요성을훼손한다는것이아닙니다.

첫번째이해해야하는부분이, 중앙버전관리시스템이할수있는모든일들은잘짜여진분산관리시스템이더잘할수있다는것을인식해야한다는것입니다. 네트워크상의자원들은기본적으로로컬상의자원들보다시간적으로나물질적으로비경제적일수밖에없습니다. 물론, 나중에도말씀드리겠지만분산제어시스템도문제점이없는시스템은아닙니다. 그러나주먹구구식의생각으로중앙관리시스템과분산관리시스템을비교하는일은없어야할것입니다. 다음인용문이이것을대변해줍니다.

” 규모가작은프로젝트들은시스템의부분적인특성만으로도실행할수있겠지만, 이런프로젝트들을스케일업할수없는확장성이낮은시스템으로계속실행하는것은마치옛로마숫자를이용해계산기를두드리는것과같다.”

더욱이당신의프로젝트는당신이처음생각했던것보다더거대한일이될지도모르는겁니다. 처음부터 Git 을사용한다는것은단순히병뚜껑을여는데스위스아미나이프를들고다니는것과같은것입니

다. 그러나, 어느날, 드라이버가필요할경우, 당신은병따개만들고다니지않았다는사실에안도의 한숨을쉬게될것입니다.

병합충돌

이주제를설명하기위해서는컴퓨터게임에비유하는것은더이상적합하지않을수있습니다. 대신에 여기서는문서편집에비유해서설명드리도록하죠.

다음예시를생각해봅시다. 앨리스 (Alice) 는파일을편집하던도중에새로운줄을첫줄로추가하고, 밥 (Bob) 은그같은파일의마지막에코드한줄을더한다고가정합시다. 그리고그들은편집된파일을각자중앙서버에업로드합니다. 대부분의시스템은자동으로두사람이각자한편집을받아들이고병합할것입니다. 결과적으로는앨리스와밥두사람의편집이모두한파일에적용되겠죠.

자이제앨리스와밥이어떤파일의정확히같은부분에서서로다른편집을한다고가정해봅시다. 이럴경우에는인간의직접적인개입없이는온순한편집이불가능하겠지요? 누가편집을하던두번째로편집하는사람은작업을결합하는과정에서오류메세지 ("*merge conflict*") 를볼수밖에없겠지요. 이런오류메세지를피하기위해선한사람만의작업을선택하거나두사람의작업을아우를수있는새로운코딩작업을추가로해줘야하는번거로움이발생하지요.

예시보다더복잡한상황이일어날수도있습니다. VCS 는간단한상황들을알아서해결해주고, 어려운상황은인간의손에맡기지요. 이런 VCS 의행동은대체적으로조정가능합니다.

== 기본적인요령 ==

Git 의수많은명령어속으로곧바로다이빙하는것보단, 다음간단한예시들을통해서천천히배우는 방법이좋을것같습니다. 제가소개하는예시들은, 표면적으로는간단하게보이지만, 앞으로여러방면으로많은도움이될것입니다. 저역시도처음 Git 을사용할때에는아래에있는예시외에는건들여보지도않았습니다.

상태 (state) 저장하는방법 ===

파일에무엇인가큰변화를주고싶으시다고요? 그러시기전에, 현디렉토리에들어있는모든파일의 스냅샷을찍어봅시다:

```
$ git init
$ git add .
$ git commit -m "My first backup"
```

위명령어들을입력후, 만약에편집을하다가잘못됐다면, 편집되기전의깨끗한버전으로되돌리면됩니다:

```
$ git reset --hard
```

또어떤작업후 state 를저장하고싶다면:

```
$ git commit -a -m "Another backup"
```

파일더하기 (add), 지우기 (delete), 이름바꾸기 (rename)

위의간단한요령들은처음 **git add** 명령어를실행했을때이미존재하던파일들만저장하게됩니다. 존재하던파일의편집이외의새로운파일들이나하위디렉토리들을추가했다면, Git 에게알려줘야

합니다:

```
$ git add readme.txt Documentation
```

그리고만약에원하지않는파일을 Git 에서없애려면그것역시 Git 에게알려줘야합니다:

```
$ git rm kludge.h obsolete.c
$ git rm -r incriminating/evidence/
```

이렇게함으로써 Git 은지정한파일들을지워주게됩니다.

Git 파일이름을바꿀때에는원치않는일반파일들의이름을지우고새로운이름을새롭게지정하는간단한절차와같습니다. 좀더손쉬운방법으로는 **git mv** 명령어가있습니다. 예를들어:

```
$ git mv bug.c feature.c
```

고급 undo 와 redo

가끔씩은작업을하다가하던일을멈추고전버전으로돌아가고싶다거나, 어느시점이후의모든편집을지우고싶을때가있을것입니다. 그렇다면:

```
$ git log
```

이명령어는최근의 commit 들을정리한리스트와그의 SHA1 hashes 를보여줍니다:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date:   Tue Mar 14 01:59:26 2000 -0800

    Replace printf() with write().

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

    Initial commit.
```

Hash 앞의알파벳몇개만으로도 commit 을세분화설정하실수있습니다; 다른방법으로는, 아래의명령어와같이 hash 전문을복사/붙여넣기하는방법도있지요:

```
$ git reset --hard 766f
```

위명령어를입력하시면설정된 commit 으로돌아갈수있으며그후의새로운 commit 들은영구적으로삭제됩니다.

가끔씩은또아주예전의 state 로잠시만돌아가길원하실수있습니다. 그럴경우에는:

```
$ git checkout 82f5
```

이명령어는 82f5 이후의 commit 들을보존함과동시에과거의시간으로잠시돌아가게해줍니다. 그러나, SF 영화에서처럼, 과거에돌아간상태에서편집을하고 commit 을한다면또다른시간대의현실을만들어가게되는것이죠. 왜냐하면당신의편집이과거의편집과는다르게입력이되었기때문입니다.

이렇게새롭게만들어진대체현실을'branch (나뭇가지)' 라고부릅니다에관해선추후에자세히설명합니다. 지금알고계셔야할것은

```
$ git checkout master
```

이명령어는과거에서현재의 state 로돌아오게해줄것입니다. 그리고 Git 이유저에게푸념을놓기전에과거에서편집했던사항들이있다면 master branch 로돌아오기전 commit 을하거나 reset 을한번실행하시길바랍니다.

게임과또다시비교해본다하면:

- **`git reset --hard`**: 예전에세이브해뒀던게임으로돌아가며, 돌아간시점이후의세이브들을모두삭제합니다.
- **`git checkout`**: 예전에세이브해뒀던게임으로돌아가며, 돌아간시점이후의게임들은처음세이브와다른길을가게됩니다. 추후의모든세이브들은다른 branch 로써새로운현실세계를만들게됩니다에관해선추후에자세히설명합니다.

예전의파일/하위디렉토리들을되돌리고싶을때다음명령어를이용함으로써필요한파일/하위디렉토리만을되돌릴수있습니다:

```
$ git checkout 82f5 some.file another.file
```

그러나이 **checkout** 명령어가다른파일들을조용히덮어씌우기할수있다는점을알아두세요! 이러한사고를방지하고싶다면 checkout 명령어를쓰기전에 commit 을이용하세요. Git 을처음이용하는분들은특히더조심하시기바랍니다. 대체적으로파일이삭제될까두려우시다면 *git commit -a* 를우선해놓고생각하세요.

긴 hash 전체를복붙하기싫으시다고요? 그렇다면:

```
$ git checkout :/"My first b"
```

이명령어를사용함으로써이 commit message 를사용해서 commit 했었던 state 로돌아갈수있습니다. 그리고이다음명령어로 5 번스텝전의 state 로돌아갈수도있습니다:

```
$ git checkout master~5
```

되돌리기 (Reverting)

법정에서는어떠한일에관해서는기록에서지울수있습니다. 이런식으로, Git 에서는원하는 commit 을정해서없던일로할수있습니다.

```
$ git commit -a
$ git revert 1b6d
```

이렇게하는것으로특정 hash 에대한 commit 을 undo 할수있습니다. 이렇게되돌린 state 는새로운 commit 으로인식되어 *git log* 에기록됩니다.

변경기록만들기

어떤프로젝트들은 changelog. 필요로합니다. 다음명령어를이용해변경기록을만들어봅시다.:

```
$ git log > ChangeLog
```

파일다운로드하기

Git 으로관리되는프로젝트사본을얻기위해서는:

```
$ git clone git://server/path/to/files
```

예를들어, 본웹사이트를만들기위해사용한파일들을얻기위해서는:

```
$ git clone git://git.or.cz/gitmagic.git
```

곧 **clone** 명령어에관해많은것을소개하도록하겠습니다.

최첨단기술

git clone 명령어를이용해어떤프로젝트의사본을다운로드해뒀다면, 다음명령어를이용해그프로젝트의최신버전으로업데이트할수있습니다:

```
$ git pull
```

즉석발행

당신이다른사람들과공유하고싶은스크립트를작성했다고가정합니다. 당신은그들에게당신의컴퓨터에서다운로드를받으라고할수있지만, 당신친구들이만약당신이해당스크립트를편집하는도중에받게된다면, 그들은예상치못한트러블에걸릴수있습니다. 이러한이유때문에릴리스사이클이란것이존재하는것입니다. 개발자들은개발중인프로젝트디렉토리에자주들락날락거릴것이고, 그들은그들이한작업이다른사람들앞에내놓을만한상태로만들어지기전까지남들에게보여주지않을겁니다.

Git 으로릴리스사이클을맞추려면, 당신의스크립트가들어있는디렉토리에서:

```
$ git init
$ git add .
$ git commit -m "First release"
```

그리고당신들친구들에게다음명령어를사용하도록하십시오:

```
$ git clone your.computer:/path/to/script
```

그들이이렇게하면당신의스크립트를다운로드할수있을것입니다. 이작업은다른유저들이 ssh 접근을할수있다고가정합니다. 그렇지않다면, 소유주인당신이 **git daemon** 명령어를쓴후친구들에게다음명령어를쓰라고하십시오:

```
$ git clone git://your.computer/path/to/script
```

이렇게하고난다음부터당신의스크립트가준비되었을때마다다음명령어를실행하면됩니다:

```
$ git commit -a -m "Next release"
```

당신의친구들은다음명령어를사용함으로써가장최근버전으로당신의스크립트를보유하고있을수있게되죠:

```
$ git pull
```

그들은절대로당신이보여주고싶지않은버전의스크립트를보는일이없을것입니다.

제가도대체뭘한거죠?

마지막으로한 commit 으로부터어떤변화가있었는지확인하기위해서는:

```
$ git diff
```

어제부터어떤변화가있었는지확인하기위해서는:

```
$ git diff "@{yesterday}"
```

어떤특정버전에서부터 2 번째전버전사이의변화를확인하기위해서는:

```
$ git diff 1b6d "master~2"
```

각각의결과는 *git apply* 와함께적용할수있는패치가될것입니다. 다음명령어도사용해보세요:

```
$ git whatchanged --since="2 weeks ago"
```

저는윗방법대신 qgit 를따로다운받아서 commit 히스토리를체크하곤합니다. 이프로그램은깨끗한그래픽인터페이스로구성되어있어보기쉽지요. 아니면, tig, 텍스트형식인터페이스역시느린인터넷속도를가지고있는분들에겐도움이될것입니다. 또다른방법으로는웹서버를설치한후 *git instaweb* 명령어를사용하는방법도있겠지요.

연습

우선 A, B, C, D 를각각연속된한파일에대한 commit 이라고가정합니다. 그리고 B 는 A 에서몇개의파일들이삭제된버전으로가정합니다. 문제는여기서그삭제된파일들을 D 에더하고싶을때어떻게하는것인가입니다.

적어도세가지의방법이있습니다. 우선우리가현재 D 에있다고생각합시다:

1. A 와 B 의차이점은몇개의지워진파일들뿐입니다. 우리는이차이점을패치로따로작성하여본래의디렉토리에적용할수있습니다:

   ```
   $ git diff B A | git apply
   ```

2. 우리는 A 에파일을저장해두었기에, 그곳에서다시받아올수있겠지요:

   ```
   $ git checkout A foo.c bar.h
   ```

3. 또는 A 에서 B 까지로갈때의변화를 undo 한다고생각하셔도됩니다:

   ```
   $ git revert B
   ```

어떤방법이가장좋은해답일까요? 답은본인이원하는것이곧해답입니다. Git 을이용한다면당신이원하는것은쉽게해낼수있고, 그것을해내는방법은한가지만있는것이아닐겁니다.

클론만들기

구형 VCS 에서는체크아웃명령어가어딘가에저장되어있는파일들을가져오는보편적인방법이었습니다.

Git 을포함한다른분산제어식 VCS 에서는클론만들기를체크아웃을대체하는보편적인방법으로채택하고있습니다. 어떤저장된파일을얻기위해서는, 원하는파일원본들이저장되어있는저장소에서내컴퓨터로끌고와' 클론' 을만들어야합니다. 즉, 중앙관리서버를미러링해오는것과같은이치라고설명할수있습니다. 클론을본떠온다면중앙관리서버가할수있는모든것들을당신이이제할수있는것이죠.

컴퓨터동기화

기본적인동기화및백업을할때 tarball 을만드는것과 *rsync* 명령어를사용하는것은이해할수있는행동입니다. 그러나저는가끔씩노트북에서편집을할때도있고, 데스크탑에서할때도있는데, 이두개의컴퓨터는 *rsync* 같은명령어를사용하면서작업할때잦은동기화를하지않을지도모릅니다.

한컴퓨터에서 Git Repository 를초기화하고파일들을 commit 함으로써이문제를해결할수있습니다. 그후다른컴퓨터에서:

```
$ git clone other.computer:/path/to/files
```

위명령어를이용해서두번째 Git repository 사본을만들수있습니다. 그다음부터는,

```
$ git commit -a
$ git pull other.computer:/path/to/files HEAD
```

을이용하여현재작업중인컴퓨터로다른컴퓨터에서작업하던파일들을 당겨올 *(pull)* 수있습니다. 만약에같은파일에대해서전후가맞지않는작업을했을경우, Git 은당신에게에러메세지로먼저이모순을해결후 commit 을할것을알려줄것입니다.

고전적인소스관리

우선 Git 저장소를초기화해줍니다:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

그리고중앙서버에서, 아무디렉토리에서나태초의 (bare) repository 를초기화해줍니다:

```
$ mkdir proj.git
$ cd proj.git
$ git --bare init
$ touch proj.git/git-daemon-export-ok
```

필요하다면 Git daemon 을실행합니다:

```
$ git daemon --detach  # 아마 이미 daemon이 실행하고 있을지도 모릅니다.
```

Git 호스팅서비스를한다면우선빈 Git repository 를만들어야합니다. 대부분웹페이지에서어떠한문서를작성하곤하죠.

다음 명령어를 사용해 당신의 프로젝트를 중앙서버로 '밀어넣기 (push)' 할 수 있습니다:

```
$ git push central.server/path/to/proj.git HEAD
```

소스를확인하고싶을때에개발자는다음명령어를사용합니다:

```
$ git clone central.server/path/to/proj.git
```

편집작업이끝난후에개발자는다음명령어를사용해로컬드라이브에각종바뀐사항들을저장을합니다:

```
$ git commit -a
```

가장최신버전으로로컬파일들을갱신하려면:

```
$ git pull
```

merge 할때일어날수있는오류들은수동으로해결후, commit 명령어를사용하여작업을 commit 해주셔야합니다:

```
$ git commit -a
```

로컬에서바뀐사항들을중앙저장소로저장하기위해서는:

```
$ git push
```

중앙서버가다른개발자들로인하여새로운변경사항이생겼을경우에는, 당신의 밀어넣기 *(Push)'* 는실패할것입니다. 그렇다면당신은 ' 밀어넣기 *(Push)* 하기전에최신버전을다시당겨서 (pull) 오류를수동으로해결후다시밀어넣기를시도해야하겠지요.

모든개발자들은특정 Git repository 에대한 SSH 접근권한이있어야 push 와 pull 를할수있습니다. 그러나개발소스는대부분모든이들에게개방된것으로써다음명령어를이용하면조회및클로닝이가능합니다:

```
$ git clone git://central.server/path/to/proj.git
```

Git 프로토콜은 HTTP 와비슷합니다: 증명서가존재하지않죠. 그래서아무나프로젝트를조회할수있는겁니다. 그런이유에서' 밀어넣기 (push)' 는 Git 프로토콜로는할수없게디폴트설정이되어있지요.

숨겨진소스

개방되어있지않은소스의프로젝트를진행할때에는 Git 의터치 (Touch) 명령어를생략합니다. 그리고'git-daemong-export-ok' 라는이름의파일을절대만들지않도록주의합니다. 이렇게하면 git 프로토콜을사용해서원치않는사람들이당신의저장소를조회하거나클로닝할수있는일은없을것입니다; 이제는 SSH 접근권이있는사람들만조회할수있게될겁니다. 당신의모든 repository 가개방되지않은경우에는 git daemon 명령어는필요없겠지요. 모든 repository 는 SSH 를통해서만허락된개발자들에게만공개될테니까요.

태초의저장소

이괴상한이름의저장소 (bare repository) 는현재작업중인디렉토리가아니기에이렇게이름이붙여졌습니다; 이저장소는하위 *.git* 디렉토리에서숨겨진파일들만을저장하는저장소입니다. 풀어설명하면, 이저장소는현프로젝트의과거를관리하기만하고, 아무런버전도저장하지않는저장소입니다.

헐벗은저장소는중앙서버관리식 VCS 와비슷한기능을담당하고있고당신의프로젝트가저장되어있는집과같은기능을담당하고있습니다. 개발자들은이곳에서부터클론을만들수있고, 작업한내용을 밀어넣기 *(Push)* 할수있습니다. 보편적으로이 bare repository 는서버에서상주하고있다가데이터를퍼트리는역할을맡고있습니다. 개발은개발자각자가만들어놓은로컬컴퓨터에서의클론에서이루어짐으로써, 워킹디렉토리없이서버내에서보호받는저장소역할을할수있습니다.

많은 Git 명령어들은 *GIT_DIR* 환경변수가 repository 로경로설정이되어있지않다면이 bare repository 에인식되지않을것입니다. *--bare* 옵션을이용한다면모를까.

밀어넣기 (push) vs. 당기기 (pull)

당기기 *(pull)* 커맨드에만의존하는대신에왜제가' 밀어넣기 (push)' 를소개했을까요? 먼저, 당기기 (pull) 는아까소개드린 bare repository 에서는실행이되지않는명령어입니다: 물론나중에소개할' 물어오기 (fetch)' 라는명령어로같은일을할수있지만요. 그러나중앙서버에저장되어있는보통일반적인 repository 에서도, 당기기 (pull) 는번거로울수밖에없습니다. 서버에로그인을해야될것이고그런후에야당기기 (pull) 을사용해야하다는말이지요. 파이어월이이런절차를방해할수도있습니다. 그리고쉘접근권한이없다면중앙서버에접속이나가능할런지요?

그러나이러한특수상황들이아니라면밀어넣기 (push) 를사용하시는것을비추합니다. 목적지가현재작업중인디렉토리가있을경우에는굉장히햇갈릴수있기때문입니다.

줄여서, Git 을배울때에는, bare repository 일경우에는' 밀어넣기 (push)' 를진행하시고아니면' 당기기 (pull)' 을사용합시다.

프로젝트포크질 (forking) 하기

현재프로젝트가진행되고있는방식이마음에안드신다고요? 당신이좀더잘할수있다고생각하세요? 그렇다면당신서버에서:

```
$ git clone git://main.server/path/to/files
```

이명령어를쓴후에, 다른사람들에게당신이포크질 (fork) 을한프로젝트에대해알리세요.

이후작업이끝난후아무때나원래프로젝트파일에서다음명령어를씀으로써포크질해놓은프로젝트로부터오리지널프로젝트파일로병합을실행할수있습니다:

```
$ git pull
```

궁극의백업

아무도건들수없고지리적으로다양한곳에저장해놓고싶은기록보관소를소유하고싶다고요? 만약당신의프로젝트에많은개발자들이참여한다면아무것도하지마십시오. 그수많은개발자들이각자클론을만들었다면그클론자체가아주효율적인프로젝트백업이될것입니다. 현상태의프로젝트뿐만이아니라, 그프로젝트의모든과거버전까지말이죠. 만약이라도어떤개발자분의클론이훼손된다면암호화된 hashing 덕에다른모든개발자들이프로젝트훼손여부에관해알수있게될것입니다.

만약당신의프로젝트에그리많은개발자들이참여하지않는다면, 최대한많은서버를확보해서클론을만들어놓으십시오.

편집증이걸린개발자들은언제나프로젝트 HEAD 의 20-바이트 SHA1 hash 를어딘가에는안전하게모셔놓죠. 안전하다는말이꼭사적인공간에저장해놓는다는말은아닙니다. 예를들면, 어떤

신문에기사를개제하는것도안전한기록보관의한방법이될수있지요. 그정보를훼손하고자하는작자들이세상에발간된모든신문카피를바꿀수는없기때문입니다.

광속의멀티테스킹

만약에어떠한프로젝트의여러부분을동시에작업하고싶으실때에는우선현재상태의프로젝트를한번 commit 한후다음명령어를사용합니다:

```
$ git clone . /some/new/directory
```

hardlinking 이라는기능덕분에로컬시스템에생성된클론들은일반백업에비해비교적적은시간과공간만필요로합니다.

이렇게하면두개의독립적인작업을동시진행할수있습니다. 예로, 한클론이컴파일중일때다른클론에서또다른작업을진행하고있을수있습니다. 그리고다른클론으로부터아무때나 commit 과당기기 (pull) 도사용할수있습니다.

```
$ git pull /the/other/clone HEAD
```

게릴라버전관리

당신은현재다른 VCS 를사용하고있지만, Git 을그리워하고있진않습니까? 그렇다면현재작업중인디렉토리에서 Git 을초기화시켜주십시오:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

그리고클론을만들고:

```
$ git clone . /some/new/directory
```

이제방금클론된디렉토리에서작업을진행하시면됩니다. 가끔은다른개발자분들과동기화하고싶으시겠죠. 그개발자분들은아직 Git 을사용하고있지않아도우리 Git 에서는:

```
$ git add .
$ git commit -m "Sync with everyone else"
```

그리고는새로운디렉토리에서:

```
$ git commit -a -m "Description of my changes"
$ git pull
```

다른분들에게당신의작업을공유하는일은그쪽분들이쓰시는 VCS 에따라다릅니다. 새로운디렉토리는이제당신이실행한작업들이포함되어있겠죠. 위의명령어를쓰신후에다른 VCS 에서쓰는명령어를통해서그들의중앙서버에업로드하실수있습니다.

Subversion 은가장좋은중앙관리식 VCS 로써개발자들사이에서애용되고있습니다. Git 에서 *git svn* 을사용해서위에서언급한일들은 Subversion 저장소를대상으로행할수있습니다.Git 프로젝트를 Subversion 저장소로보내기.

Mercurial

Mercurial 역시비슷한 VCS 으로써 Git 과쉽게연동될수있습니다. 'hg-git' 플러그인을통해서 Mercurial 유저들은 Git 저장소에쉽게' 밀어넣기 (push)' 와' 당기기 (pull)' 을사용할수있죠.

Git 으로 *hg-git* 플러그인을구하는방법:

```
$ git clone git://github.com/schacon/hg-git.git
```

Mercurial 로'hg-git' 플러그인을구하는방법:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

유감스럽지만다른 VCS 에선 Git 과비슷한플러그인이있는지는모르겠습니다. 그렇기때문에 Mercurial 보다는 Git 을주저장소를쓰길선호합니다. Mercurial 로프로젝트를진행할경우에는대부분한개발자가 Git 저장소를같이병행관리하는업무를떠맡곤합니다. 그러나 Git 으로 Mercurial 프로젝트를진행할경우에는'hg-git' 플러그인의도움으로그러한번거로움이필요없겠죠.

Mercurial 에있는 repository 를 Git repository 로' 밀어넣기 (push)' 를사용하여쉽게바꿀수있으나, 'hg-fast-export.sh' 스크립트를사용해이작업을더쉽게끝낼수있습니다. 다음저장소에서이스크립트를구할수있습니다:

```
$ git clone git://repo.or.cz/fast-export.git
```

빈저장소에서이작업을한번해봅시다:

```
$ git init
$ hg-fast-export.sh -r /hg/repo
```

위명령어는스크립트를'$PATH' 에넣은후에실행합니다.

Bazaar

Bazaar 는 Git 과 Mercurial 다음으로많이알려진 VCS 입니다.

Bazaar 는만들어진지별로되지않은시스템이기에엄청난가능성이잠재하고있지요; Bazaar 개발자들은다른 VCS 의단점을배우고고쳐나가는중입니다. 그리고 Bazaar 개발자들은 Bazaar VCS 가다른 VCS 들과연동하는문제에많은노력을기울이고있습니다.

bzr-git' 플러그인은 *Bazaar* 이용자들이 *Git* 과함께연동해작업할수있도록해줍니다. *'tailor* 프로그램은 Bazaar repository 를 Git repository 로바꿔줍니다. 'bzr-fast-export' 도한번검색해보세요.

내가 Git 을사용하는이유

제가 Git 을처음에사용했던이유는제가듣기에 Git 은 Linux kernel source 관리에용이하다고들었기때문입니다. Git 을사용한이후로는다른 VCS 로바꿔야겠다는생각은들지도않았지요. Git 은저에게매우유용했고저는아직 Git 으로인한어떠한심각한오류를겪어보지도않았습니다. 저는 Linux 를주로이용하기때문에다른플랫폼에서발생할수있는문제는생략하겠습니다.

그리고저는 C, bash scripts, Python 을이용하는사람이고프로그램런타임에목숨을거는사람중하나입니다.

Git 이어떻게좀더발전할수있을지, 또 Git 과비슷한프로그램도직접짜보기도했지만학교프로젝트정도로만썻었을뿐입니다. 그러나제가직접저만의 VCS 를만들었더라도저는 Git 을계속이용했을겁니다. 제프로그램을써도별로투자한것에비해얻을것이적어보였기때문이지요.

자연스레여러분들이필요로하고원하는프로그램은계속해서바뀝니다. 그러나 Git 과는그럴가능성이매우적지요. == 브랜칭마법 ==

Git 의끝내주는기능들중에는즉석으로브랜칭 (brancing) 및병합 (merging) 이가능하다는것입니다.

예시: 외부적인요소들은가끔불가피하게당신이하던일을그만두게합니다. 예를들어, 치명적인버그가경고없이퍼져나가게생겼습니다. 프로그램에새로넣어야할기능이있는데데드라인은가까워져옵니다. 당신이도움을요청하고자했던개발자는퇴사할려고하니도움을요청할수도없고요. 시간이촉박한만큼하던일을멈추고버그를고치는데에올인을해야겠지요.

위의예시와같이하던일을갑자기멈추는것은일의생산성을치명적으로떨어트립니다. 특히나지금까지하던일과정상관없는부분의프로그램을건들어야할때말이죠. 이럴때, 중앙관리식 VCS 를사용하는경우엔버그없는버전의프로그램을새로다시받아야할껍니다. 분산관리식 VCS 일경우에는원하는버전만로컬컴퓨터로받아내면되죠.

하지만클로닝은작업중인디렉토리포함그디렉토리의히스토리를어느선까지는같이다운로드받게합니다. Git 은최대한효율성있게시스템이디자인되어있긴하지만, 클로닝명령어를쓴다면프로젝트파일들이 (비효율적으로) 현재작업중인디렉토리에전부다시생성될것입니다.

해답: Git 은이런상황에서좀더빠르고공간적으로효율성있게클로닝을할수있는명령어를가지고있습니다: **git branch**

이런환상적인명령어를이용하여디렉토리에있는파일들은탈바꿈을감행해이버전과저버전을넘나들수있습니다. 이변형기법은버전사이를넘나드는것외에도더많은것을할수있습니다. 당신의프로젝트는브랜칭을통해구버전에서임시버전, 개발버전, 친구들이보유하고있는버전등으로아무렇게나변형할수있습니다.

일하는척하기버튼

버튼하나만누르면 (" 일하는척하기버튼") 게임화면이최소화되고엑셀파일이화면상에나타나는기능을보신적이있을겁니다. 이기능을활용하면직장상사의눈을속이고일하던척할수있지요?

어떤디렉토리에서:

```
$ echo "I'm smarter than my boss" > myfile.txt # 난 내 상사보다 똑똑하다
$ git init
$ git add .
$ git commit -m "Initial commit"
```

우리는" 난내상사보다똑똑하다" 라는내용을가진텍스트파일을 Git repository 에만들었습니다. 그리고:

```
$ git checkout -b boss  # 이 명령어를 사용한 후엔 아무것도 바뀌지 않은 것처럼 보일겁니다.
$ echo "My boss is smarter than me" > myfile.txt
```

```
$ git commit -a -m "Another commit"
```

겉으로보기에는그텍스트파일을새로운문장으로덮어씌우고 commit 을한것처럼보일겁니다. 그러나그건착각입니다. 다음명령어를입력해보세요:

```
$ git checkout master  # 처음 버전으로 돌아가기
```

자! 그럼처음생성했던텍스트파일이돌아왔을겁니다. 만약에그상사가이사실을알아채고당신의 디렉토리를살펴본다고할때는:

```
$ git checkout boss  # 아까 두 번째로 만들어놓은 "상사는 나보다 똑똑합니다"라는 메세지를 담
```

이런식으로두가지다른버전의파일사이를오갈수있습니다. 그리고각각따로 commit 을할수있지요.

힘든작업

당신이어떤작업을하고있다고가정합니다. 작업도중에세버전전으로돌아가서새로운 print 라인을넣고테스팅해보고싶다는생각이들었습니다. 그럴때엔:

```
$ git commit -a
$ git checkout HEAD~3
```

이제테스팅하고싶었던파일에더하고싶은것을걱정없이마구넣어도됩니다. 이미친짓 (?) 을 commit 해놓을수도있습니다. 작업이다끝났다면,

```
$ git checkout master
```

를사용해아까미친짓을하기전의작업상태로돌아올수있습니다. Commit 하지않았던작업들이 같이딸려왔다는것을확인 (조심!) 할수있을겁니다.

아까그임시작업 (미친짓) 을세이브하고싶다면어떻게해야할까요? 쉽습니다:

```
$ git checkout -b dirty
```

를실행하여그나뭇가지 (branch) 에서마스터나뭇가지로돌아오기전에 commit 을하면됩니다. 그런후다시미친짓을할때의상태로돌아가고싶다면:

```
$ git checkout dirty
```

우리는이체크아웃이라는명령어를전에도설명했었죠. 여기서는이명령어가어떻게예전버전들을 불러오는지살펴볼수있었습니다: 파일을원하는버전으로돌아가게할수있으나, master 나뭇가지를우선벗어나야하지요. 벗어난후의 commit 은 master 나뭇가지와는다른길을걷게될것입니다. 그길을나중에이름도지어줄수있지요.

다시말하면, 예전상태 (state) 에서벗어나면 Git 은자동으로이름이없는새로운나뭇가지로이동시켜줍니다. 이나뭇가지는 *git checkout -b* 로이름을바꿔저장해줄수있죠.

빠른코드수정

작업중에갑자기하던일을멈추고'1b6d...'commit 에있는버그를고치라고부탁을받았다고생각해봅시다:

```
$ git commit -a
$ git checkout -b fixes 1b6d
```

버그를다고친후에:

```
$ git commit -a -m "Bug fixed"
$ git checkout master
```

이제아까잠시중단했던작업으로돌아갈수있습니다. 버그가고쳐진파일도병합해올수있죠:

```
$ git merge fixes
```

병합 (Merging)

Git 을제외한다른 VCS 들을이용할땐나뭇가지 (branch) 들을만드는것은쉽지만나뭇가지들을병합하기는어려웠습니다. Git 에서는병합작업이정말쉽고병합이진행되고있는중인지우리도모르는사이에끝날것입니다.

병합은전에도소개했었습니다. 당겨오기 *(pull)* 명령어는 commit 들을가져와지금머물고있는 branch 에병합하여줍니다. 로컬에서아무런작업을진행하지않았더라면 *pull* 은현 branch 를 빨리감기하여중앙서버에서가장최근의정보를가져와병합합니다. 로컬에서작업을한기록이있었다면, Git 은자동으로병합을시도할것이고, 병합중에버전간의차질이있다면당신에게친절히알려줄것입니다.

Commit 은보통하나의 부모 *commit'* 이 *(*과거의 *commit)* 있습니다. 그러나병합을하게된다면하나의 *commit* 이적어도 *'* 아버지 *commit'* 와 *'* 어머니 *commit* 이있다고생각할수있는것이죠. 그럼'HEAD~10' 은어떤 commit 을가르키는걸까요? 부모 commit 두개이상이라면어떤 commit 을받아순조롭게작업을계속진행할수있을까요?

Git 은먼저시간적으로제일먼저 commit 되었던부모를따르게설정되어있습니다. 현재작업중인 branch 가병합이실행될경우이 branch 자체가첫번째부모가되기때문에당연한겁니다: 당신은언제나현나뭇가지에서가장최근에한작업에만관심이있을가능성이크기때문이지요. 다른나뭇가지에서한작업은다음일입니다.

탈자기호 (^) 를이용하서부모를수동으로정해줄수도있습니다. 예를들어두번째부모의기록을조회하고싶다면:

```
$ git log HEAD^2
```

첫번째부모의기록을조회할때는탈자기호이후의번호는생략해도됩니다. 굳이보여드리자면:

```
$ git diff HEAD^
```

이표기법은다른형식의표기법과도병행해서사용할수있습니다:

```
$ git checkout 1b6d^^2~10 -b ancient
```

(집중하십시오) 새로운나뭇가지인"ancient" 를시작하고두번째부모의첫번째부모나뭇가지에서 1b6d 로시작하는 commit 과그 commit 전 10 개의 commit 을불러와줄것입니다.

방해받지않는작업진행

하드웨어관련작업을하다보면현재작업중인단계가완료되어야만다음단계진행이가능할것입니다. 자동차를예로들면외부로부터오기로했던부품들이도착해야비로소수리에들어갈수있겠지요. 프로토타입들은칩들이가공되어야건축이가능해지겠죠.

소프트웨어관련작업도비슷합니다. 다음작업이진행될려면현재작업이이미발표및테스트가되어있어야할겁니다. 어떤작업들은당신의코드가받아들여지기전검토부터되어야겠지요. 그래서당신은그검토가끌날때까지는다음작업으로진행하지못할것입니다.

하지만나뭇가지와병합기능덕분에이규칙을깨고파트 1 이완료되기도전에파트 2 에서미리작업을진행하고있을수있습니다. 파트 1 을 commit 하고검토를위해어디론가보냈다고생각해봅시다. 만약당신이 Master 나뭇가지에서작업하고있었다면, 그 branch 에서다른 branch 로갈아타야합니다:

```
$ git checkout -b part2
```

그리곤파트 2 에서 commit 을하며작업을계속진행하세요. 인간은실수를많이하는동물이기에파트 1 으로다시돌아가서무엇인가고치고싶을지도모릅니다. 만약에당신이천재적인프로그래머라면다음명령어를사용할일은없겠지요.

```
$ git checkout master  # 파트 1로 돌아갑니다.
$ fix_problem # 수정 작업
$ git commit -a        # 수정 작업을 commit합니다.
$ git checkout part2   # 파트 2로 다시 갑니다.
$ git merge master     # 아까 파트 1의 수정을 파트 2로 병합합니다.
```

이때즈음이면이미파트 1 은허가받았겠지요.

```
$ git checkout master  # 파트 1로 돌아갑니다.
$ submit files         # 파일 배포!
$ git merge part2      # 파트 2도 파트 1으로 병합.
$ git branch -d part2  # 파트 2 나뭇가지 삭제.
```

이제파트 2 의모든것과함께업데이트된 master 나뭇가지로돌아왔습니다.

branch는 갯수의 제한 없이 원하는 만큼 생성할 수 있습니다. 역순으로 branch를 만들 수도 있죠: 만약에 7번의 commit전에 나뭇가지를 하나 만들어 놓았어야 함을 늦게 깨닫았을 때, 다음 명령어를 이용해 보세요:

```
$ git branch -m master part2  # master 나뭇가지의 이름을 part2로 바꿉니다.
$ git branch master HEAD~7    # 7 commit 전의 상황에서 master 나뭇가지를 새로 만듭니다.
```

Master 나뭇가지는이제 part 1 만들어있고, 나머지는모두 part 2 에들어가게되었습니다. 그리고우리는지금 part 2 에서작업을하고있는중이겠지요; master 를만들면서 master 로는현재옮겨간상태가아닙니다. 처음보시죠? 여태까지설명한예제들에서는나뭇가지를만들면서곧바로작업공간도같이옮겨갔었는데말이죠. 이런식으로요:

```
$ git checkout HEAD~7 -b master  # 나뭇가지를 만들고 바로 작업공간도 그 나뭇가지로 옮긴다.
```

메들리의재정리

하나의 branch 에서모든작업을끝내고싶을수도있습니다. 작업중인일들은혼자만알고중요한 commit 들만다른사람들에게보여주고싶을수있습니다. 그럴경우엔두개의 branch 를우선만드세요:

```
$ git branch sanitized    # 정돈된 commit을 보여주기 위한 나뭇가지를 만듭니다.
$ git checkout -b medley  # 작업을 하게 될 "메들리" 나뭇가지를 만들어 이동합니다.
```

버그를고치던, 어떤기능을더하던, 임시코드를더하던작업을진행합니다. 물론 commit 을해가면서말이죠. 그리고:

```
$ git checkout sanitized
$ git cherry-pick medley^^
```

위의명령어들을차례로사용한다면" 메들리" 나뭇가지의 commit 들을"sanitzed" 나뭇가지에붙입니다. "cherry-pick" 명령어를잘사용한다면마지막결과물에만첨부된코드들이들어있는 branch 를만들수있습니다. 잡다한 commit 들도잘정리되어있을겁니다.

Branch 관리하기

여태까지프로젝트에서생성한나뭇가지들을보려면:

```
$ git branch
```

기본적으로"master" 나뭇가지에서작업을시작하는것이디폴트로지정되어있습니다. 그러나어떤개발자들은"master" 나뭇가지는그대로냅두고새로운나뭇가지를만들어서그곳에서작업하는것을선호합니다.

-d* 와 *-m 옵션들은각각 branch 들을지우거나 branch 이름을바꿔줄수있는파라메터들입니다. *git help branch* 를보시면더욱자세히설명되어있을겁니다 (번역주: 어차피영어입니다)

"master" 나뭇가지는관례적인이름의 branch 일뿐입니다. 다른개발자들은당신의저장소에당연히"master" 라는이름을가진 branch 가있을것이라고생각하겠지요. 그리고그 branch 는모든공식적인자료들이들어있다고넘겨짚을것입니다. 물론당신은"master" 를없에거나새로운이름을지정해줄수있으나, "master" 나뭇가지를쓰는관례를따르는것을추천합니다.

임시 branch

Git 을사용하다보면당신은쓸모없는하루살이같은쓸때없는 branch 들을많이만들고있다는사실을깨달을것입니다. 이유는다음과같겠지요: 그많은 branch 들은작업의경과를저장하기위해임시로만들어놓고무엇인가고칠것이있을때빨리돌가가기위해서쌓아두기만하고있는거겠죠.

다른 TV 채널에서무얼하나확인할때잠시채널을바꾸는것과같은아이디어입니다. 그러나리모콘버튼몇개누르면되는것과는달리, 많은나뭇가지를만들고, 설정하고, 병합하고, 나중에다쓰면지워야합니다. 다행히도 Git 에서는 TV 리모트와비슷하게지름길이있습니다:

```
$ git stash
```

이명령어는현버전을임시저장소 (stash) 에저장해주고작업하기전의상태로돌아갑니다. 작업중인디렉토리는작업 (편집, 버그고침등) 하기전의상태로돌아가겠지요. 그리고임시 (stash) 로돌아가고싶다면:

```
$ git stash apply  # 에러 (version conflict)가 날지도 몰라요. 수동적으로 해결하세요.
```

물론여러개의임시저장소 (stash) 를만들수도있습니다. *git help stash* 에설명이되어있으니읽어보세요. 눈치챘을지모르겠지만, Git 은올바른임시저장소 (stash) 기능을쓰게해주기위해서자체적으로임의생성된 branch 들을몰래이용한답니다.

내방식대로작업하기

Branch 를이용하는것이꼭필요한지생각할지도모르겠습니다. 파일들을클로닝하는게제일빠르고 *cd* 를이용해디렉토리를바꿈으로써 branch 사용을대체하고싶을지도모릅니다.

웹브라우저의예를들어보겠습니다. 여러개의창아니면여러개의탭을지원하는이유는무엇일까요? 여러이용자들의작업방식을존중하여주기위해서랍니다. 어떤이용자들은웹브라우저창하나만열고여러탭을열어서작업하는방식을추구합니다. 다른이용자들은반대의형식으로작업하는것을추구할지도모르죠: 여러개의창을만들고탭이없이작업하는것을말이죠. 또어떤이용자들은이두방법들을섞어서작업하는걸선호할지도모릅니다.

Branch 들은마치작업중인디렉토리의탭과같습니다. 클로닝은새로운브라우저창을여는것과같은것이죠. 이두가지방법은모두빠르고로컬에서진행됩니다. 그러니당신에게맞는방법을찾아보는건어떨까요? Git 은당신이원하는대로일하게도와줄것입니다.

히스토리레슨

분산관리식시스템을택한 Git 은개발자들이 history 관리를용이하게할수있게해줍니다. 그러나프로그램의과거를들춰내려면조심하세요: 당신이소유하고있는파일들만다시쓰기하세요. 세계각국의나라들이누군가어떤잘못을하나하면누가했는지끝임없이따지는것처럼만약한개발자가당신이가지고있는파일과기록 (history) 이다른파일들을클론하여갔을때추후병합시문제가생길지도모릅니다.

어떤개발자들은파일의수정기록들이절대로조작되면안되는것이라고믿고있습니다. 반면에어떤개발자들은수정기록들이깨끗하게정리되어보여져야할것만선택하여보여져야한다고합니다. Git 은이렇게다른성향의개발자들을모두포용할수있습니다. 클로닝, branch, 병합과같은기능들이당신이어떤개발자타입이던당신의일처리를도와줄것입니다. 어떻게영리하게사용하는지는당신에게달려있죠.

오류수정합니다

방금 commit 을했는데, 그 commit 에달린메세지를바꾸고싶다고요? 그렇다면:

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

위명령어를사용하면마지막으로한 commit 의메세지를바꿀수있습니다. 파일을더하는것을잊어버리고 commit 을했다고요? *git add* 를사용하고서위의명령어를사용하세요.

마지막으로했던 commit 에편집을더하고싶으신가요? 그렇다면작업후에다음명령어를쓰세요.

```
$ git commit --amend -a
```

... 더있습니다

이제전보다더꼬인상황을마주했다고생각합시다. 우선당신이긴시간동안작업해서많은 commit 을하였다고가정해봅시다. 그러나당신은그 commit 들의난잡한구성이마음에들지않습니다. 그리고몇몇 commit 메세지들을다시쓰고싶다면:

```
$ git rebase -i HEAD~10
```

위명령어를사용한다면당신의작업용에디터에지난열개의 commit 이출력될것입니다. 샘플을보자면:

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

여기서는'log' 와는달리가장오래된 commit 부터가장최근의 commit 의순서로나열되어출력되었습니다. 여기서는 5c6eb73 가가장오래된 commit 이고 100834f 이가장최근 commit 이죠. 그리고:

- 한줄을지움으로써 commit 을삭제합니다. *revert* 명령어와비슷하지만기록에는남지않게지웁니다: 이방법은마치 commit 이처음부터존재하지않던것처럼보여지게해줍니다.
- Commit list 를재정렬하며 commit 의순서를바꾸어줍니다.
- 위의 *pick* 명령어대신에
  - `edit` 을사용하여개정시킬 commit 을마킹합니다.
  - 'reword'를사용하여로그메세지를바꿉니다.
  - `squash` 를사용하여전에했던 commit 과합병합니다.
  - `fixup` 를사용하여전에했던 commit 과합병후 log 메세지를삭제합니다.

예를들어, 두번째행의 *pick'* 을*'squash* 명령어로바꾸어봅니다:

```
pick 5c6eb73 Added repo.or.cz link
squash a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

저장후프로그램을종료하면, Git 은 a311a64 를 5c6eb73 로병합시킵니다. **squash** (짓누르기) 는지정된 commit 을바로다음 commit 으로밀어붙어버린다고생각하시면됩니다.

또, Git 은로그메세지들을합친후나중에편집할수있게해주기도합니다. **fixup** 명령어를사용하면이런절차를하지않아도됩니다; 짓눌려진 (squash 된) 로그메세지들은간단히삭제되기때문입니다.

edit 을이용하여어떤 commit 을마킹해두었다면, Git 은같은성향의 commit 들중에가장오래전에했던 commit 의작업상태로당신을되돌려보냅니다. 이상태에서아까전말했듯이편집작업을할수도있고, 그상태에맞는새로운 commit 을만들수도있습니다. 모든수정작업이만족스럽다면다음명령어를사용해앞으로감기를실행할수있습니다.:

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

Git 은다음 *edit* 까지아니면아무런 *edit* 이없다면현재작업상태까지 commit 을반복실행할것입니다.

새로운 rebase 를포기할수도있습니다:

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

그러니 commit 을부지런하게자주하십시오: 나중에 rebase 를사용하여정리할수있으니까요.

로컬에서의수정작업

어떤프로젝트를진행하고있습니다. 당신의컴퓨터에서 commit 을하며작업을하다가이제공식적인프로젝트파일들이있는 branch 와동기화해야합니다. 이런절차는메인 branch 에올리기전에거쳐야할과정이지요.

그러나당신의로컬 Git 클론은당신혼자만이해할수있는수많은기록이뒤죽박죽섞여있을것입니다. 아무래도개인적인기록이깨끗하게정리되어공식적인기록만볼수있게된다면좋겠지요:

위에서설명했듯이 **git rebase** 명령어가이작업을해줄것입니다. **--onto** 플래그를사용하여상호작용을피할수도있습니다.

git help rebase 를확인해서좀더자세한예를한번봐보세요. Commit 을나눌수있다는걸알게될것입니다. 여러 branch 들을재정리할수도있죠.

rebase 는유용한명령어입니다. 여러가지실험을하기전에 *git clone* 으로복사본을만들어두고놀아보세요.

기록다시쓰기

가끔은어떤그룹사진에서포토샵으로몇사람지우는기능과같은명령어가필요할지도모릅니다. 스탈린식스타일로사람을무자비하게지우는명령어말입니다. 예를들어이제어떤프로젝트를대중에게공개할시간이왔다고가정합니다. 그러나어떤파일들은일반유저들이보지못하도록하고싶습니다. 당신크레딧카드번호를실수로썻다던지했다면더욱더그러고싶겠지요. 이런경우, 파일을지우는것만으로는부족합니다. 예전 commit 으로파일을지워진파일을복구하는것이가능하기때문이죠. 당신은이파일을모든 commit 으로부터없에야할것입니다:

```
$ git filter-branch --tree-filter 'rm top/secret/file' HEAD
```

git help filter-branch 를보세요. 여기서는본예시에대해설명하고있고더빠른방법을제시하여줄것입니다. 대체적으로 *filter-branch* 은하나의명령어만으로도대량의파일기록을변화시킬수있을것입니다.

그리고 `+.git/refs/original+` 디렉토리는 이렇게 만든 변화가 오기 전의 기록을 보여줄 것입니다.

마지막으로, 당신의클론을새로운버전의클론으로바꾸시면됩니다.

기록만들기

어떤프로젝트를 Git 으로옮겨오고싶다고요? 다른 VCS 에서옮겨오는것이라면, 어떤개발자가이미프로젝트의기록을 Git 으로쉽게옮겨오는스크립트를써두었을지도모릅니다.

아니라면, 특정포맷으로텍스트를읽어 Git 기록을처음부터작성하여주는 **git fast-import** 명령어를확인해보세요. 대체적으로한번에간편하게프로젝트를 Git 에서사용할수있게해줄겁니다.

예를들어, */tmp/history* 같은임시파일에다음텍스트를붙여넣기해보세요:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Initial commit.
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
EOT


commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Replace printf() with write().
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

그리고이임시파일로 Git repository 를만들어보세요:

```
$ mkdir project; cd project; git init
$ git fast-import --date-format=rfc2822 < /tmp/history
```

가장최근버전을가져오고싶다면:

```
$ git checkout master .
```

git fast-export 명령어는아무 Git repository 를결과물이사람들이읽을수있는포맷으로

바꾸어주는 **git fast-import** 포맷으로바꾸어줍니다. 이명령어들은텍스트파일들을텍스트파일전용채널을통해서 repository 로밀어넣기해줍니다.

어디서부터잘못되었을까?

당신은몇달전에는잘작동되던프로그램이갑자기안된다는것을발견했습니다. 아놔! 이버그는어디서부터생긴것일까요? 개발을하면서테스팅을종종했더라면진작에알아챘을텐데요.

이미그러기엔너무늦었습니다. 그러나 commit 이라도자주했다는가정하에 Git 은이러한짐을덜어줄수있습니다:

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git 에서한프로젝트를자체적으로테스팅을실행합니다. 그리고버그가발견된다면:

```
$ git bisect bad
```

버그가더이상없다면위명령어에서"bad" 를"good" 으로바꾸세요. Git 은 good 버전과 bad 버전사이로당신을데려갈겁니다. 물론버그를찾을확률은높아지지요. 몇번이렇게반복하다보면결국엔버그를일으킨 commit 을찾아낼수있게도와줄것입니다. 버그찾기를완료했다면다시처음작업상태로 (이젠버그가없겠지요) 돌아가야겠지요:

```
$ git bisect reset
```

수동으로테스팅하는것보단, 다음명령어로테스팅을자동화할수있습니다:

```
$ git bisect run my_script
```

```
Git 은 기존 대비할 스크립트에 약간의 변화를 주어서 이것이 좋은 변화인지
안 좋은 변화인지 체크합니다: 좋은 변화는 0으로 무시해야할 변화는 125로
안 좋은 변화는 1과 127사이의 번호로 테스팅을 종료합니다. 마이너스 숫자는
이분화 (bisect)를 강제종료하지요.
```

```
당신은 이것보다 더 많은 일을 할 수 있습니다. 도움말은 이분화를 시각화해주는 방법과,
이분화 기록을 다시보는 방법, 이미 확인된 이상없는 변화들은 건너띄어 테스팅을 가속 시켜주는
```

누가망가뜨렸을까?

다른 VCS 들과같이 Git 은누군가를탓할수있는기능이있습니다:

```
$ git blame bug.c
```

이명령어를사용하면누가언제마지막으로어느부분을작업하였는지표시하여줍니다. 다른 VCS 들과는달리모든작업은오프라인에서진행됩니다.

나의개인경험담

중앙관리식 VCS 에서는파일들의기록변경은어려울뿐만아니라관리자만이변경할수있습니다. 클론, branch 만들기와병합하기는네트워크를통해서만할수있는작업들입니다. 브라우징기록

보기, commit 하기역시중앙관리식 VCS 에서는네트워크를통해야만합니다. 어떤시스템에서는네트워크연결이되어야지만자기자신이작업한기록을보거나편집할수있습니다.

중앙관리식 VCS 는개발자의수가늘어남에비례해서더많은네트워크통신을요구하기때문에오프라인에서작업하는것보다비경제적일수밖에없습니다. 그리고제일중요한것은모든개발자들이고급명령어들을적재적소에쓰지않는다면모든작업이어느정도는무조건느릴수밖에없다는것입니다. 극적인경우에는아주기본적인명령어역시잘못하면느려질수밖에없습니다. 무거운명령어를써야한다면일의효율성은나쁜영향을받을수밖에없습니다.

저는직접이런상황들을겪어보았습니다. Git 은제가사용한가장먼저사용한 VCS 였죠. 저는 Git 의여러기능들을당연하다생각하고금방적응하였습니다. 저는당연히다른 VCS 들역시 Git 이제공하는기능들을가지고있을것이라고생각하였습니다: VCS 를선택하는것은텍스트에디터나웹브라우저를선택하는것과같은맥락일것이라고생각하였습니다.

제가나중에강제로중앙관리식 VCS 를사용하게되었을땐완전쇼킹이였죠. 불안정한인터넷연결은 Git 을사용할때중요치않습니다. 그러나이러한인터넷연결은로컬디스크에서작업하는것만큼은효율적일수는없죠. 그리고저는어떤명령어는연결딜레이를고려함에따라습관적으로쓰지않고있는걸시간이지나며깨달았습니다. 이런습관은제가원하는방식대로작업을할수없게하는방해물들이었죠.

어쩔수없이느린명령어를사용할때는저의작업효율에치명타를입히기일쑤였죠. 네트워크로처리되야하는작업이완료되길기다리면서이메일확인및다른문서작업을하며시간을때웠습니다. 그러다가원래하던작업에돌아가면다시무엇을했었는지기억을해내는데시간이많이허비된경험이잦았습니다. 인간은환경변화에적응을할수는있으나그적응이결코빠르진않죠.

일을하면서발생하는아이러니한비극도존재했죠: 네트워크상황이원활하지않을것이라는걸아는개발자들은미래에딜레이를줄이기위해지금하는작업들이오히려현재트래픽을더잡아먹을수가있다는것입니다. 모든개발자들이네트워크를원활하게하는노력을할수록오히려서로에게해가될수있다는것입니다. 이게무슨아이러니한일입니까? == Git 은멀티플레이어 ==

제가과거에단독개발자였던시절부터 Git 을사용해오고있었습니다. 그당시엔여태까지소개했던많은명령어들중 *pull* 과 *clone* 정도만사용하여같은프로젝트를여러디렉토리에저장하는데사용하였습니다.

시간이지난후 Git 에제가만든코드를올리고싶었고다른개발자들이한작업도반영하고싶었습니다. 저는전세계의많은개발자들을관리하는방법을배워야했습니다. 다행히도이런일을도와주는것은 Git 의가장큰힘입니다. Git 이존재하는이유이기도하지요.

난누굴까?

각 commit 은작성자의이름과작성자의이메일주소를저장합니다. 이목록은 *git log* 를사용해조회할수있습니다. 기본설정상 Git 은시스템에이미기본으로세팅이되어있는정보를이용해작성자의이름과이메일주소를저장합니다. 그러나수동으로이름과이메일주소를설정하려면:

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

를사용하여지정해주십시오.

현재사용중인 repository 에만한정적으로사용할수있는이름이나이메일을설정하려면위명령어에서 *--global* 을사용하지마세요.

SSH, HTTP 를통한 Git 사용

웹서버에관한 SSH 접근권한을보유하고있다고합니다. 그러나 Git 은아직설치되어있지않다고 가정합니다. 기존프로토콜만큼효율적이진않겠지만, Git 은 HTTP 를통해데이터교환이가능 합니다.

우선기본적으로컴퓨터에 Git 을다운받아서설치합니다. 그리고웹디렉토리에저장소를만듭니 다:

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

Git 의예전버전에선복사를지시하는명령어가안들을수있습니다. 그렇다면:

```
$ chmod a+x hooks/post-update
```

이제아무클론에서 SSH 를통해당신의작업을업로드할수있습니다:

```
$ git push web.server:/path/to/proj.git master
```

다른사람들은당신의작업을다운받으려면다음명령어를쓰면될겁니다:

```
$ git clone http://web.server/proj.git
```

모든것은 Git 을통한다

서버나인터넷연결없이저장소를동기화시키고싶다고요? 긴급하게수정할것이발견되었다고요? **git fast-export** 그리고 **git fast-import** 명령어들은 repository 를하나의파일로묶어 주거나그하나의파일을 repository 로되돌려줄수있는것을배웠습니다. 다양한매개체를통해서 repository 의파일들을옮길수있지만, 정말효율적인방법은 **git bundle** 명령어를쓰는것입니 다.

보내는이가' 묶음 (bundle)' 을만듭니다:

```
$ git bundle create somefile HEAD
```

그리고 다른 장소로 그 묶음, +somefile+을 어떻게든 옮겨야한다고 가정합시다: 이메일로 보내야할

```
$ git pull somefile
```

파일을 받는 사람들은 빈 repository에서도 이 명령어를 사용할 수 있습니다. 파일의 사이즈가 작아 보임에도 불구하고 +somefile+ 은 저장소의 본 모습을 담고 있습니다.

큰프로젝트에서는묶음만들기를좀더효율적으로하기위해서버전차이만을묶어줍니다. 예를들어 서"1b6d" 의 hash 가달린 commit 이가장최근에보내는이와받는이사이에공유된 commit 이라고가정해봅니다:

```
$ git bundle create somefile HEAD ^1b6d
```

너무자주이렇게한다면, 어떤 commit 이가장최근것인지기억하지못할수있습니다. 도움말에서 는태그를이용해이런문제점들을피하라명시합니다. 풀어서말하자면어떤묶음을보낸후에는:

```
$ git tag -f lastbundle HEAD
```

그리고새로운묶음을만들어줍니다:

```
$ git bundle create newbundle HEAD ^lastbundle
```

패치: 세계적통화

패치는컴퓨터와인간이쉽게알아들을수있는언어로파일의변화를텍스트로표현할수있는방법입니다. 전세계어떤개발공간에서나이런식으로파일의변화를시도합니다. 어떠한 VCS 를쓰던간에개발자들에게패치를이메일로보낼수도있습니다. 그개발자들이그이메일을읽을수만있다면그들은당신이편집을한작업기록을손쉽게볼수있을겁니다. 즉, 온라인용 Git repository 를만들필요가없다는말이지요.

첫장에서본명령어를다시한번해봅시다:

```
$ git diff 1b6d > my.patch
```

위명령어는간단한패치를생성하여이메일로보낼수있게도와줍니다. Git 저장소에서다음을따라해보세요:

```
$ git apply < my.patch
```

위명령어를사용하여패치를적용시킵니다.

작성자의이름과싸인이기록되어야하는좀더공식적인환경에서는그에상응하는패치 (1b6d 이후의작업) 를만들기위해다음명령어를사용합니다:

```
$ git format-patch 1b6d
```

이렇게만든파일묶음은 *git-send-email* 을사용하여보낼수있습니다. 보내고싶은 commit 묶음을수동으로지정해줄수도있습니다:

```
$ git format-patch 1b6d..HEAD^^
```

받는쪽에서는이메일을받을때는받은 txt 형태의패치파일을저장한후:

```
$ git am < email.txt
```

이 명령어는 새로받은 패치를 적용시키고 작성자의 정보가 포함된 새로운 commit을 만듭니다.

브라우저상으로이메일을수신한다면, 당신의이메일클라이언트에서첨부된패치의본코드의형태를확인해야할수도있습니다.

mbox-를기반으로하는이메일클라이언트는약간의문제점들이있습니다. 그러나이런방식의클라이언트를쓸만한사람이라면손쉽게튜토리얼을읽지않고도해결할수있을것입니다.

죄송합니다. 주소를옮겼습니다

Repository 를클로닝한후 *git push* 나 *git pull* 을사용하면원래의 URL 에서해당명령어를실행합니다. Git 은어떤원리로이렇게하는것일까요? 그비밀은클론을만들때생선된 config 옵션에서찾을수있습니다. 한번볼까요?:

```
$ git config --list
```

`remote.origin.url` 옵션은 URL 소스를통제합니다; "origin" 은원래 repository 에붙여진별명이라고보면됩니다. Branch 에"master" 라고이름이붙듯이말이죠. 그말은이이름을바꾸거나지울수있는데할필요는없다는것입니다.

제일처음사용하던 repository 가옮겨지면, URL 을수정해주어야합니다:

```
$ git config remote.origin.url git://new.url/proj.git
```

`brach.master.merge` 옵션은 *git pull* 로당겨올수있는 branch 를설정하여줍니다. 처음으로클론을생성하였을때, 그클론의 branch 는그클론을만들어온 repository 의현재사용중인 repository 와같게설정이되어있습니다. 그렇기때문에현재작업헤드가다른 branch 로옮겨갔었다고하더라도, 추후의당겨오기는본래의 branch 를따를수있게해줄것입니다.

본옵션은처음에 +branch.master.remote+ 옵션에기록되어있는클론의대상 repository 에만적용됩니다. 다른저장소에서당겨오기를실행한다면, 구체적으로어떤나뭇가지에서당겨오길원하는지설정해주어야합니다:

```
$ git pull git://example.com/other.git master
```

이것은왜전에보여드렸던 **push* 와 *pull** 예제에다른 argument 가붙지않았었는지설명하여줍니다.

원격 branch 들

어떠한 repository 를클론할때에는그클론의모든 branch 를클론하게됩니다. Git 은이사실을은폐하기에당신은클론을하면서몰랐을지도모릅니다: 그러니당신은직접 Git 에게물어보아야합니다. 이설정은원격 repository 에있는 branch 들은당신의 branch 들과꼬이게하는일을없게해줍니다. 그래서초보자들이 Git 을사용할수있는것이고요.

다음명령어를이용하여숨겨진 branch 들을포함해서모두나열합니다:

```
$ git branch -r
```

당신은다음과비슷한결과물들을보게될것입니다:

```
origin/HEAD
origin/master
origin/experimental
```

이결과는각행마다원격 repository 의 HEAD 와 branch 를보여주며, 다른 Git 명령어들과함께사용될수있습니다. 예를들면, 당신은지금많은 commit 을하였다고먼저가정합니다. 그러고는가장최근에가져온버젼과비교를하고싶다고생각해봅니다. SHA1 해쉬를찾아서확인할수도있지만다음명령어로더간단히비교할수있습니다:

```
$ git diff origin/HEAD
```

아니면"experimental" 나뭇가지가지금어떠한상태인지알아낼수도있습니다.

```
$ git log origin/experimental
```

다수의원격 repository

당신외의두명의개발자가프로젝트를공동으로진행하고있다고가정합니다. 그리고그둘의작업상황을주시하고싶습니다. 당신은다음명령어를사용함으로써하나이상의 repository 를추적할수있습니다:

```
$ git remote add other git://example.com/some_repo.git
$ git pull other some_branch
```

이제두번째 repository 의 branch 로병합을시도하였으며모든 repository 의모든 branch 에대한접근권한이생겼습니다.

```
$ git diff origin/experimental^ other/some_branch~5
```

그러나내작업과관련없이버전의변화를비교해내는방법은무엇일까요? 풀어말하자면그들의 branch 를보는동시에그들의작업이내작업에영향받지않게하고싶다는것입니다. 그렇다면간단하게당겨오기보다는:

```
$ git fetch        # 태초의 repository로부터 물어옵니다. 디폴트 명령어.
$ git fetch other  # 다른 개발자의 repository를 물어옵니다.
```

작업기록들만을가져오는명령어들입니다. 현재작업중인디렉토리는영향을받지않을것이지만, 로컬사본을가지고있기에우리는이제어느 repository 의어떤 branch 라도 Git 명령어를사용하여활용할수있습니다.

Pull 은간단히풀어서설명하면 **fetch(물어오기)** 후 *merge(병합하기)* 를합친하나의고급명령어라고말할수있습니다. 우리는마지막으로한 commit 을현재작업에병합하길원하기때문에주로 *pull(당겨오기)* 를사용하게될것입니다; 위에설명한상황은특수상황이지요.

git help remote 에는원격 repository 를삭제하는방법, 특정 branch 를무시하는방법외에많은것을볼수있습니다.

나만의취향

저는작업을할때다른개발자들이제가당겨오기를실행할수있게항시준비해두는것을선호합니다. 어떠한 Git 호스팅서비스는클릭한번만으로도쉽게이를행할수있게도와주는것도있습니다.

어떤파일꾸러미를물어온후에는 Git 명령어들을사용하여프로젝트가잘정리되어있길빌며변화기록을조회합니다. 그러고는저의작업을병합합니다. 그후제작업이맘에들경우메인저장소에밀어넣기합니다.

제가다른사람들로부터많은도움을받는스타일은아니지만, 이러한제작업방식을추천드리고싶습니다. 다음링크를한번보세요. Linus Torvalds 의블로그포스팅.

Git 의세상에거주하는것은패치파일들을만들어배포하는것보다더효율적입니다. Git 은단순한버전관리외에도작업을행한사람의이름, 이메일주소, 작업날짜를같이기록하여줍니다. == Git 마스터링 ==

지금까지배운것만으로도당신은 **git help** 페이지를자유롭게돌아다니며거의모든것을이해할수있을것입니다. 그러나어떠한문제를풀기위해어느한가지의알맞는명령어를찾는것은아직성가실수있습니다. 그런문제에대해제가도와줄수있을것같습니다: 아래는제가 Git 을사용하며유용하게썼던몇가지팁들입니다.

소스배포

제프로젝트에서 Git 은제가저장및공개하고싶은파일들을정확히추적하여줍니다.

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

바뀐것은꼭 commit

Git 에게무엇을추가, 삭제및파일이름을바꾸었는지일일히알려주는것은귀찮은짓일지도모릅니다. 대신에당신은다음명령어를쓸수있습니다:

```
$ git add .
$ git add -u
```

Git 은현재작업중인디렉토리에있는파일들을자동으로살피며자세한사항들을기록하여줍니다. 위의두번째명령어 (git add -u) 대신에'git commit -a' 를사용하여그명령어와 commit 을 동시에해낼수있습니다. *git help ignore* 를참고하여특별히지정된파일을무시하는방법을알아보십시오.

위의모든것을한줄의명령어로실행할수있습니다.

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

-z* 와 *-0 옵션은파일이름이이상한문자를포함하고있을때생길수있는여러가지문제점들을처리하여줍니다. 이옵션들은무시된파일들을포함하여줌으로써'-x' 아니면'-X' 을같이써주어야할 것입니다.

내 commit 이너무클경우?

Commit 을한지시간이좀많이지난상황입니까? 코딩을너무열심히한나머지버전컨트롤하는것을깜빡했나요? 프로젝트에여러가지연관성없는수정을한상태입니까?

걱정하지말고:

```
$ git add -p
```

당신이만든모든수정작업에대하여 Git 은어떠한것들이바뀌였는지코드로보여주며당신에게다음에실행할 commit 에부분적인코드가포함될사항인지물어볼것입니다. "y" 와"n" 를이용하여 대답할수있습니다. 물론이대답을당장하지않아도됩니다; "?" 로좀더알아보십시요.

모든수정작업이마음에든다면:

```
$ git commit
```

위의간단한명령어를사용하여직접선택한작업만을 commit 합니다. 이상황에선반드시 *-a* 옵션을생략하시길바랍니다. 그렇지않으면 Git 은모든수정작업을 commit 할것입니다.

만약에여러군데다른디렉토리에많은수정작업을해놓았다면어떻게할까요? 수정된사항을하나씩검토하는작업은정말귀찮은짓입니다. 이럴땐 *git add -i* 를사용합니다. 몇번의타이핑만으로도특정파일의수정작업을검토할수있게됩니다. 또는 *git commit --interactive* 를사용하여작업중자동으로작업후 commit 하는방법도있을수있습니다.

인덱스: Git 의스테이징구역

여태까지 Git 의중요한기능인'index' 를피해왔지만이제한번살펴본시간이온것같습니다. 인덱스는임시적인스테이징구역 (번역주: 책갈피처럼) 으로보면됩니다. Git 은당신의프로젝트와프로젝트의기록사이에데이터를직접옮기는경우는드뭅니다. 대신, Git 은인덱스에파일을쓰며그리고그파일들을마지막목표지점에카피하여줍니다.

예를들어 **commit -a* 는원래두단계과정을거치는하나의명령어입니다. 첫번째단계에서는현작업상황의스냅샷을찍어모든파일들을인덱스하는과정을거칩니다. 두번째단계에서는방금찍은스냅샷을영구적으로보관하게됩니다. *-a** 옵션을쓰지않고 commit 을하는것은이두번째과정만실행하는일입니다. 그렇기에 **git add** 같은명령어를쓴후에 commit 을하는것이당연한이야기가되겠지요.

대체적으로인덱스에관한컨셉트는무시하고파일기록에서직접적으로쓰기와읽기가실행된다는개념으로이해하면편합니다. 이런경우에는우린인덱스를제어하는것처럼좀더세부한제어를하기원할것입니다. 부분적인스냅샷을찍은후영구적으로이' 부분스냅샷' 을보존하는것이죠.

머리 (HEAD) 를잃어버리지않기

HEAD 태그는문서작업시보이는커서처럼마지막 commit 포인트를가르키는포인터역할을합니다. Commit 을실행할때마다물론 HEAD 도같이앞으로움직이겠지요. 어떤 Git 명령어들은수동으로 HEAD 를움직일수있게해줍니다. 예를들면:

```
$ git reset HEAD~3
```

위명령어를사용한다면 HEAD 를 commit 을 3 번하기전으로옮깁니다. 이후모든 Git 명령어는가지고있던파일은현재상태로그대로두되그 3 번의 commit 을하지않은것으로이해하죠.

그러나어떻게해야다시가장최근으로돌아갈수있을까요? 예전에했던 commit 들은미래에대해서아무것도모를텐데말이지요.

원래의 HEAD 의 SHA1 을가지고있다면:

```
$ git reset 1b6d
```

그러나이것을어디에도써두지않았었더라도걱정하지마십시오: Git 은이런경우를대비해서원래의 HEAD 를 ORIG_HEAD 로어딘가에는저장하여둡니다. 그러고는다음명령어를사용하여미래로안전하게돌아올수있지요:

```
$ git reset ORIG_HEAD
```

HEAD-헌팅

ORIG_HEAD 로돌아가는것만으로는충분하지않을지도모릅니다. 당신은방금엄청나게큰실수를발견하였고아주오래전에했던 commit 으로돌아가야할지모릅니다.

기본적으로 Git 은 branch 를수동으로삭제하였더라도 2 주의시간동안 commit 을무조건저장하여둡니다. 문제는돌아가고싶은 commit 의 hash 를찾는일입니다. '.git/objects' 의모든 hash 값을조회하여얻어걸릴때까지해보는방법이있긴합니다만, 여기좀더쉬운방법이있습니다.

Git 은모든 commit 의 hash 를'.git/logs' 에저장해둡니다. 하위디렉토리'refs' 은모든 branch 의활동기록을저장하여두고동시에'HEAD' 파일은모든 hash 값을저장하고있습니다. 후자는실수로마구건너뛴 commit 들의 hash 도찾을수있게해줍니다.

reflog 명령어는당신이사용하기쉬운유저인터페이스로 log 파일들을표현하여줍니다. 다음명령어를사용하여보십시오.

```
$ git reflog
```

hash 를 reflog 으로부터자르고붙여넣기보다는:

```
$ git checkout "@{10 minutes ago}"
```

아니면 5 번째전에방문했던 commit 으로돌아갈수도있습니다:

```
$ git checkout "@{5}"
```

좀더많은정보는 *git help rev-parse* 의" 재편집구체화하기" 섹션을참고하십시오.

아까언급한 commit 의 2 주살이생명을수동으로연장하여줄수있습니다. 예를들어:

```
$ git config gc.pruneexpire "30 days"
```

위명령어는 30 일이지난후에지워진 commit 들역시영구적으로삭제된다는의미입니다. 그러고는 *git gc* 가실행되지요.

git gc 가자동실행되는것을꺼줄수도있습니다:

```
$ git config gc.auto 0
```

이경우에는 *git gc* 를수동적으로실행시켜 commit 들을삭제할수있지요.

Git 을좀더발전시키는방법

진정한 UNIX 와같이 Git 의디자인은다른프로그램들의 GUI, 웹인터페이스와같은하위파트들과호환이됩니다. 어느 Git 명령어들은유명인사의어깨위에서있는것처럼 Git 그자체가스크립팅언어로사용될수도있습니다. 조금만시간을투자하면 Git 은당신의기호에더알맞게바꿀수있습니다.

한가지트릭을소개하자면자주사용할것같은명령어들을짧게만들어줄수있는방법이있습니다:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # 현재 설정한 명령어들의 '가명'을 표기해줍니다.
alias.co checkout
$ git co foo                              # 'git checkout foo'와 같은 것입니다.
```

또다른트릭은현재작업중인 branch 의이름을작업표시창에표시하여주는명령어도있습니다.

```
$ git symbolic-ref HEAD
```

위명령어는현재작업중인 branch 이름을표기하여줍니다. 실제로는"refs/heads/" 를없애고잠재적으로일어날수있는에러들을무시하는걸추천드립니다:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

`contrib` 하위디렉토리는유용한 Git 툴들이저장되어있는장소이기도합니다. 시간이지나면이곳에있는툴들은공식적으로인정받아고유명령어가생기기도하겠지요. Debian 과 Ubuntu 에서는이디렉토리는 +/usr/share/doc/git-core/contrib+ 에위치하고있습니다.

앞으로 +workdir/git-new-workdir+ 디렉토리에방문할일도많을것입니다. 시스템링크기술을통해서이스크립트는원래의 repository 와작업기록을공유하는새로운작업디렉토리를생성하여줍니다:

```
$ git-new-workdir an/existing/repo new/directory
```

새롭게생성된디렉토리는클론으로봐도무방하며일반클론들과의한가지차이점은어느한곳에서작업을하던두개의디렉토리는앞으로계속싱크를진행하며같은기록을가지게된다는것입니다. 즉, 병합, 밀어넣기, 당겨오기를해줄필요가없어지는것이지요.

용감한스턴트

Git 은요즘유저들이데이터를쉽게지우지못하도록하고있습니다. 그러나몇가지의상용적인명령어를통해서이런 Git 만의방화벽쯤은쉽게뚫어버릴수있지요.

Checkout: Commit 하지않은작업들은 checkout 을할수없습니다. 방금작업한모든것들을없던일로하고그래도굳이 commit 을진행하고싶다면:

```
$ git checkout -f HEAD^
```

반면에 checkout 을할위치를수동으로설정하여준다면 Git 의방화벽은처음부터작동하지않을것입니다. 설정해준위치는조용히덮어씌우게됩니다. 그러니, 이런방식으로 checkout 을할때에는주의하십시오.

Reset: 리셋은 commit 되지않은작업이있으면실행되지않을것입니다. 그래도강제로하고싶다면:

```
$ git reset --hard 1b6d
```

Branch: 방금한작업을잃어버릴것같으면 Git 은 branch 가지워지지않게합니다. 그래도하고싶다면:

```
$ git branch -D dead_branch  # -d 대신 -D
```

비슷한방식으로, commit 을안한작업이있어서 move 명령어를통해서덮어씌우기가안될경우에는:

```
$ git branch -M source target  # -m 대신 -M
```

체크아웃과리셋과는다르게위의두명령어는데이터를직접삭제하진않습니다. 모든변경기록은.git 하위디렉토리에남게되고필요한 hash 는'.git/logs' 에서찾을수있습니다 (위의"HEAD-헌팅" 섹션참고). 기본설정상, 이기록들은 2 주동안은삭제되지않습니다.

Clean: 몇 git 명령어들은추적되지않은파일들을망쳐놓을까봐실행이안되는경우가종종있습니다. 만약에그파일들이삭제되도된다는확신이선다면, 가차없이다음명령어를사용하여삭제하십시오:

```
$ git clean -f -d
```

이후에는위모든명령어들은다시잘실행되기시작할것입니다!

원치않는 commit 들을방지하기

바보같은실수들은내 repository 를망쳐놓곤합니다. 그중에서도제일무서운것은 *git add* 를쓰지않아서작업해놓은파일들을잃어버리는것이지요. 그나마코드뒤에빈공간을마구넣어놓는 다던지병합에서일어날수있는문제들을해결해놓지않는것은애교로보입니다: 별로문제가되는것 들은아니지만남들이볼수있는 repository 에서는보여주기싫습니다.

hook 을사용하는것과같이제가바보같은짓을할때마다경고를해주는기능이있다면얼마나좋을까 요:

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # 예전 Git 버젼에서는: chmod +x pre-commit
```

이제는아까설명했던애교스러운실수들이발견될때 Git 은 commit 을도중에그만둘것입니다.

이가이드에서는 **pre-commit** 앞에밑에써놓은코드를넣음으로써혹시있을지도모르는바보같 은짓을방지하였습니다.

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

많은 git 작업들은 hook 과상호작용합니다; **git help hooks* 를참조하십시오. 우리는"HTTP 를통한 Git" 을을설명할때 *post-update** hook 을활성화시켰습니다. HEAD 가옮겨질때마다같이실행되지요. Git over HTTP 예제에서는 post-update 스크립트가통신에필요한 Git 을업데이트했었습니다.

비밀을벗겨내다

Git 의안을들여다보고 Git 이어떻게작동하는지알려드리겠습니다. 너무디테일한것들은알아서빼놓고설명해드리겠습니다. 그래도자세히알고싶은분들은: the user manual로가시길바랍니다.

보이지않는능력

Git 은어떻게눈에띄지않게강력한툴일수있을까요? 습관적으로하게되는 commit 과병합을제외하고, VCS 자체가컴퓨터에설치되지않은것같아보일때가있습니다.

다른 VCS 들은사용할때쓸때없이많은절차와검열등으로고생할수있습니다. 파일의보안상태가' 읽기전용' 으로세팅되어작업을하려고할때마다중앙서버에승인을요청해야할경우도빈번합니다. 그리고 VCS 의유저들이많아질수록업무처리속도가현저히떨어질수도있습니다. 그리고중앙서버나네트워크가다운될경우에는아무런작업을할수없죠.

반면에, Git 은당신의로컬컴퓨터디렉토리에'.git' 디렉토리를형성하여그곳에작업기록을하게됩니다. 그기록은온전히당신만의것이죠. 그렇기에오프라인상태에서도작업을끊기지않고진행할 수있습니다. 그리고작업중인파일에대해모든권한을가지고있게해주죠.

진실성

대부분의사람들은크립토그래피를어떤정보를비밀스럽게숨기는정도로생각합니다. 그러나크립토그래피의진정한목적은정보를보안하는것이죠. 크립토그래피로보호되는 hash 는데이터가공격받거나지워질위험으로부터보호해줍니다.

SHA1 hash 는고유의 160-비트 ID 번호로생각하시면됩니다. 그리고이번호는당신이평생, 또는열번의삶을살정도의시간에, 쓸 byte 에부여되는번호이기에보안이철저합니다.

SHA1 hash 자체역시 byte 로구성되어있기에 SHA1 hash 의 hash 를만드는것도가능합니다. 이건생각보다유용한기능입니다: 'hash chains' 를한번살펴보세요. 우리는나중에 Git 이 이기능을어떻게사용해서효율적으로데이터를보호하는지살펴보겠습니다.

짧게말하자면, Git 은당신의모든데이터를 '.git/objects'섭디렉토리에저장합니다. 그리고보통의파일이름대신각파일에지정된 ID 를통해서이파일들을찾을수있습니다. 그렇기에 Git 은보통의파일시스템을뭔가굉장히효율적인데이터베이스로변화시켜줍니다.

똑똑함

어떻게 Git 이당신이파일의이름을변경할때 Git 에게이름을바꾼다말한적도없는데알수있을까요? *git mv* 를실행할수도있겠지만그것은 *git rm* 을사용하고 *git add* 를사용하는것과같습니다.

Git 은단순화된지침으로재설정된파일명을찾아내고버전사이의카피들을만들어냅니다. Git 은코드들이옮겨지고카피되고지워질경우를다알아낼수있죠. 모든경우의수를다알수는없겠지만, Git 은대부분을알아채고있고이자동화된기능은날이갈수록발전하고있습니다.

색인화

Git 은 Git 이트랙킹하는모든파일들의크기, 생성된시간, 마지막편집시간을색인 (index) 를 이용해서기록하여둡니다. 만약에어떤파일에작업하여변화가생겼다면 Git 은현파일상태와인덱스에저장되어있는상태를비교하여파일의변화를감지합니다. 만약에서로간의차이가없다면 Git 은그파일이가장최신버전으로업데이트되었다고생각하고더이상읽지않습니다.

인덱스정보를읽는작업은파일을읽는것보다는훨씬빠르게진행되니당신이한작업들은 Git 이아주빠르게업데이트해줄것입니다.

인덱스는마치중간대기구역과같다고말씀드린적이있습니다. 왜그렇게얘기했을까요? **Add** 명령어는파일들을순전히업데이트시켜데이터베이스에업데이트하지만 *commit* 은 (별도의옵션을사용하지않는다는전제하에) 자체적인로컬데이터베이스에있는파일들에대한 commit 만 진행하지요.

Git 의근원

Linux Kernel Mailing List post 에서 Git 의역사를나열하여설명해줍니다. Git 의역사학자들에게정말흥미있는웹사이트지요.

오브젝트데이터베이스

데이터의모든버전은' 오브젝트데이터베이스' 에보관되며, 이데이터베이스는 '.git/objects'의 섭디렉토리에상주하고있습니다. '.git/'에있는다른파일들은더적은정보를담고있지요 (인덱스, branch 이름, 태그, 설정정보, 로그, head commit 의위치등). 오브젝트데이터베이스는 Git 의기본이지만우아하고또 Git 의힘의원천입니다.

'.git/objects' 에있는파일들은각각오브젝트입니다. 그리고크게세가지오브젝트로나눌수있습니다: *blob* 오브젝트, *tree* 오브젝트, and *commit* 오브젝트.

Blob 오브젝트

첫째, 마술을보여드리겠습니다. 우선아무파일이름을선택하십시오. 빈디렉토리에서:

```
$ echo sweet > YOUR_FILENAME
$ git init
$ git add .
$ find .git/objects -type f
```

`.git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d` 을보게될것입니다.

저는파일이름을알지도못하는데이걸제가어떻게알까요? 왜냐하면

```
"blob" SP "6" NUL "sweet" LF
```

의 SHA1 hash 는 aa823728ea7d592acc69b36875a482cdf3fd5c8d 이고, SP 는공간이며, NUL 는 0 바이트이고, LF 는라인피드이기때문입니다. 직접확인해보시려면다음명령어를입력하세요:

```
$ printf "blob 6\000sweet\n" | sha1sum
```

Git 은콘텐츠주소를지정하는것이가능합니다: 파일은파일이름에따라저장되지않습니다. 대신에 hash 로저장이되며이런정보는 blob 오브젝트라고불리우는곳에저장되어있지요. 우리는 hash 를파일내용에대한고유 ID 로생각할수있습니다. 그래서어떤의미에서우리는파일의내용에따라주소를지정하는것으로생각이가능합니다. 초기 'blob 6'는오브젝트타입과크기를저장해놓은 header 나다름없습니다; 단지내부부기를단순화합니다.

따라서저는당신이보게될것을쉽게예측할수있었습니다. 파일이름은관련이없습니다. 내부데이터만을이용해 blob 오브젝트를구성하는게가능합니다.

그러면당신은동일한파일이어떻게되는지궁금할수있습니다. 우선아무런파일이름을사용해복사본을추가해보십시오. + .git / objects + 의내용은몇개의복사본을추가해도동일합니다. Git 은데이터를한번만저장합니다.

별개로, + .git / objects + 내의파일은 zlib 로압축되어있으므로그들을직접보지마십시오. zpipe -d 를통해필터링을해서보시던지, 아니면다음명령어를실행해보시면됩니다.:

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

주어진오브젝트를예쁘게인쇄해줍니다.

Tree 오브젝트

그러나파일이름은어디에간거죠? 그들은어떤단계에서어딘가에저장되어야합니다. Git 은커밋 중에파일이름을찾습니다.

```
$ git commit  # Type some message.
$ find .git/objects -type f
```

이제 3 개의개체를보게될것입니다. 이번에는당신이선택한파일이름에부분적으로의존하기때문에두개의새파일이무엇인지제가말씀드릴수없습니다. "rose" 를파일이름을지정한것으로가정하여진행하겠습니다. 그렇게설정하시지않은경우기록을다시작성하여그렇게지정한것처럼보이게할수있습니다.

You should now see 3 objects. This time I cannot tell you what the 2 new files are, as it partly depends on the filename you picked. We'll proceed assuming you chose "rose". If you didn't, you can rewrite history to make it look like you did:

```
$ git filter-branch --tree-filter 'mv YOUR_FILENAME rose'
$ find .git/objects -type f
```

그럼이제 +.git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9+ 를보실수 있을겁니다. 왜냐하면이것이 SHA1 hash 이기때문입니다:

```
"tree" SP "32" NUL "100644 rose" NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d
```

다음을입력하여이파일에실제로위내용이포함되어있는지확인하십시오.

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

zpipe 으로이 hash 를확인하는것이아마쉬운방법일겁니다:

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum
```

Hash 확인은 cat-file 로는좀어려울수도있습니다. 왜냐하면압축되지않은원래의파일보다더많은파일을포함하고있을수있기때문입니다.

This file is a *tree* object: a list of tuples consisting of a file type, a filename, and a hash. In our example, the file type is 100644, which means 'rose' is a normal file, and the hash is the blob object that contains the contents of 'rose'. Other possible file types are executables, symlinks or directories. In the last case, the hash points to a tree object.

If you ran filter-branch, you'll have old objects you no longer need. Although they will be jettisoned automatically once the grace period expires, we'll delete them now to make our toy example easier to follow:

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

실제프로젝트의경우일반적으로이와같은명령을피해야합니다. 이명령어들은백업들을지울수있기때문이죠. 깨끗한 repository 를원한다면일반적으로새로운클론을새롭게만드는것이좋습니

다. 그리고 +.git+ 을조작할때는주의하십시오: 만약에여러가지커맨드들이동시에실행중이거나갑작스러운정전이발생하면안되잖아요. 일반적으로 refs 는 *git update-ref -d* 로삭제해야합니다. 그래도 +refs / original+ 를수동으로제거하는것이안전하긴하지만요.

Commit 오브젝트

우리는 3 개의오브젝트들중 2 개를설명했습니다. 세번째는'commit' 오브젝트입니다. 이오브젝트의내용은 commit 메시지와날짜및시간에따라다릅니다. 여기에있는것과일치시키려면약간의조정이필요합니다.

```
$ git commit --amend -m Shakespeare  # Commit 메시지를 바꿉니다.
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
    GIT_COMMITTER_EMAIL="bob@example.com"'  # 타임스탬프와 저자를 바꿉니다.
$ find .git/objects -type f
```

그럼이제 .git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187 를보게될것입니다.

```
"commit 158" NUL
"tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

이전과마찬가지로 zpipe 또는 cat-file 을실행하여직접확인할수있습니다. 이것은첫번째 commit 이므로부모 commit 이없지만이제나중에 commit 을할때마다항상부모 commit 을식별하는한줄이포함될것입니다.

마술과분간이안되는프로그램

Git 의비밀은너무단순해보입니다. 몇시간안에몇개의셸스크립트를혼합하고 C 코드를추가하여몇시간만에만들수있는것같아보입니다. 결국에 Git 은견고성을위해잠금파일과 fsync 로장식된기본파일시스템작업과 SHA1 해싱의혼합으로구성된프로그램입니다. 실제로이것은 Git 의초기버전을정확하게설명합니다. 그럼에도불구하고공간을절약하기위한독창적인패킹트릭과시간을절약하기위한독창적인인덱싱트릭을포함해이제우리는 Git 이버전컨트롤을위한완벽한데이터베이스로거듭나게되는지알게되었습니다.

예를들어, 오브젝트데이터베이스내의파일이디스크에의해손상된경우오류가발생하면 hash 가더이상일치하지않아문제를알려줍니다. 기존 hash 에다른 hash 를지정해주면서, 우리는모든수준에서무결성을유지합니다. Commit 은부분적으로파일을관리하지않고전체적으로프로젝트를관리하여줍니다. Commit 의 hash 를계산하고서모든 tree, blob 과부모 commit 들을

저장한후에데이터베이스에저장합니다. 오브젝트데이터베이스는정전과같은예상치못한방해에영향을받지않습니다.

우리는 Git 으로가장사악한적도물리칠수있습니다. 만약에누군가가아주예전버전의프로젝트에서파일내용을은밀하게수정한다고생각해봅시다. 오브젝트데이터베이스를정상상태처럼보이게유지하려면해당데이터베이스의해시도변경해야할겁니다. 이말은해당파일을참조하는 tree 오브젝트의 hash 도변경해야한다는의미입니다. 그리고차례로그러한 tree 를포함하는모든 commit 오브젝트의 hash 도변경해야한다는거지요. 이러면해당파일이후의 commit 모두변경을해야한다는말이됩니다. 이것은공식헤드의 hash 는나쁜 repository 의 hash 와달라질수밖에없다는것입니다. 그러니일치하지않는 hash 의흔적을따라나쁜의도로변경된파일을정확히찾아낼수있습니다. 처음에손상된 commit 도마찬가지로찾을수있습니다.

짧게말하자면, 마지막 commit 을나타내는 20 바이트가안전하다면 Git repository 를임의로아무에게도들키지않게조작하는것은불가능합니다.

Git 의유명한기능은또어떻습니까? branch 만들기? 병합하기? 태깅하기? 세부사항. 현재 head 는 + .git / HEAD + 파일에보관됩니다. commit 오브젝트의 hash 를포함합니다. Commit 중에 hash 가업데이트됩니다. Branch 는거의동일합니다: 그것들은 +.git/refs/heads+ 에저장된파일들입니다. 태그도 +.git/refs/tags + 에있지만다른명령어들로업데이트됩니다. == 부록 A: Git 의약점들 ==

Git 을소개하면서저는 Git 의약점들을몇개숨기긴했습니다. 몇가지약점들은 script 나 hook 을통해해결할수있고, 몇가지는프로젝트를수정하면서해결할수있고, 그외의약점들은현시점에선그냥앉아서기다리고있을수밖에없습니다. 그러기싫으시다면직접도와줘보십쇼!

SHA1 약점

시간이지나면해커들은 SHA1 의약점들을더많이발견하게될겁니다. 이미 hash 에서의충돌을찾아내는건가능한일이지요. 몇년안에는 Git repository 를위해할수있는연산능력을가진일반컴퓨터도있을수있습니다.

Git 이그런일이일어나기전에 hash 관련기능들을발전할수있었으면좋겠어요.

Microsoft Windows

Git 을 Microsoft Windows 에서사용하는건성가실수있습니다:

- Cygwin, 리눅스와비슷한윈도우체제에선 a Windows port of Git 가있습니다.
- Git for Windows 는아직몇몇허점이있지만 Windows 에서 Git 을효율적으로쓸수있게해줍니다.

Git 과연관없는파일들

만약에당신의프로젝트가굉장히크고, 쓸때없는파일들이많이들어있는상태이고, 상시로바뀌는상태라면, Git 은하나의파일을트랙킹하지않기에다른 VCS 보다유용하지않을수있습니다. Git 은프로젝트단위로트랙킹을하기때문입니다. 이건 Git 의장점입니다.

그래도만약하나의파일만을트랙킹하기원하다면프로젝트를여러개의파트로분리해두는겁니다. 여러개의파트로분리해도 **git submodule** 명령어를이용하면하나의 repository 를유지할

수있을겁니다.

누가어떤작업을하는거지?

몇몇의 VCS 는유저들로하여금작업하기전에파일들을강제로마킹시킵니다. 이러한강제성은중앙서버와연결하는데귀찮은절차이지만두개의장점이있습니다:

1. 버전의차이 (Diff) 를체크하는데매우빠릅니다. 마킹된파일만검사하면되니까요.
2. 유저는어떤사람이어떤작업을하는지중앙서버를조회하면간단히알아낼수있습니다.

Git 으로도이렇게하는게가능합니다. 그러나그렇게하기위해선코딩이좀필요하니프로그래머의도움이좀필요할수있겠군요.

파일히스토리

Git 은프로젝트전체를트랙킹하기때문에어떤한파일의히스토리를재건설하는데다른 (하나의파일만트랙킹하는) VCS 들보다느릴수있습니다.

그렇게심하게느려진다는것은아니고오히려 Git 의장점들이이하나의단점을상쇄하고도남습니다. 예를들어'git checkout' 은'cp -a' 보다빠르고프로젝트전체의변화를압축화하는것이파일하나하나씩압축하는것보다효율적입니다.

태초의클론

만약에어떠한프로젝트의히스토리가길경우, 클론을만드는것은다른 VCS 들의'checking out' 보다컴퓨터의용량을더차지할수있습니다.

그러나길게보면클론이 checking out 보다나을것입니다. 클로닝이후다른명령어들은매우빠르고오프라인으로도진행이가능하니까요. 그러나어떠한경우에는좀더히스토리가얕은클론을 *--depth* 명령어를통해만드는것이더나은선택일수있습니다. 이렇게만들어진클론은작업실행속도가빠르겠지만몇몇기능들이제외되어있을수있습니다.

불완전한프로젝트들

Git 은파일에작업을더많이할수록그작업량에대비해빠르게 Version Control 을할수있도록하기위해쓰여진프로그램입니다. 인간은하나의버전에서다음버전으로작업을할때소량의작업만진행할수있죠. 예를들어, 한줄짜리코드에있는버그를고친다던가, 새로운기능을넣는다던가, 코멘트를코드에단다거나말이죠. 그런데만약 commit 과 commit 사이에작업량이방대하게클경우그파일의히스토리는비례해서커질수밖에없겠죠.

VCS 는이것에대해아무것도할수없습니다. 일반 Git 유저들은그부풀어진파일들을곧대로받아들일수밖에없겠죠.

그러나왜방대한작업량이필요했는지에대해알아볼필요는있습니다. 파일포맷이바뀌어서그랬을수도있죠. 소량의작업은소량의변화를주기마련입니다.

아니면데이터베이스나백업자료실를구축해놓는것이이런방대한프로젝트들을진행하는데에있어 VCS 보다적합할수도있습니다. 예를들어 VCS 는어떤웹캠에서주기적으로찍은이미지를관리하는데에는적합하지않습니다.

만약에파일들이매번변화하고있고각각의변화에무조건버젼번호를매겨야겠다한다면 Git 을중앙서버처럼쓰는방법밖에없습니다. 개발자들은상대적으로가벼운클론을만들면되죠. 이렇게일을진해하면물론단점도있을겁니다. 픽스들을패치로배포해야하고 Git tool 들이들어먹지않을수도있어요. 근데이렇게라도일을진행해야하는게맞는방법일수있습니다. 아무도히스토리가매우긴프로젝트들을곧대로받긴싫어하거든요.

다른예시로는큰바이너리파일들을수행하는펌웨어들에기반한프로젝트를진행할경우입니다. 펌웨어의히스토리는유저들에게별로흥미로운소재는아니고, 업데이트들은압축률이매우좋지않습니다. 그래서펌웨어들을재구성할때는 repository 의크기가매우커지는경우가있죠.

이럴때에는모든소스코드들이 Git repository 에저장되어있는편이좋고, 바이너리파일들은따로보관되어야할것입니다. 이일을좀더쉽게진행하기위해서 Git 유저가어떤파일에대해클론을만들수있고 *rsync* 를할수있으며, 가벼운클론을만들수있는코드를배포하는것이좋을수있습니다.

글로벌카운터

몇몇중앙관리식 VCS 들은새로운 commit 이받아들여질때마다임의의양의정수를보존합니다. Git 은양의정수보다나은 hash 를써서 commit 을관리합니다.

그러나어느사람들은아직도양의정수로 commit 관리를하길추구합니다. 다행히도 Git 에추가프로그래밍을하여 Git repository 에서양의정수를 1 씩더하는방식으로 commit 을관리할수도있습니다.

어느클론파일이나양의정수를사용하여 commit 을관리할수있습니다. 그러나이건아무짝에도쓸모가없죠. 중앙 repository 만이숫자를쓸꺼니까요.

빈 (empty) 섭디렉토리

빈섭디렉토리는트랙킹되지않습니다. 그러나더미파일을만들어서트랙킹하게편법을쓸수있죠.

현버전의 Git 으로써이문제점은 Git 의약점입니다. Git 이다시수면위로올라가고더많은사람들이사용하게될수록이런약점도메꿔나가지겠죠.

태초의 commit

보통의컴퓨터공학자들은숫자를셀때 0 부터세지 1 부터세지않습니다. 하지만안타깝게도 commit 의횟수를셀때 git 은컴퓨터공학자들처럼숫자를세지않습니다. Git 의그많은명령어들은 commit 이태초적으로한번이루어지기전까지는실행되지않을겁니다. Branch 들을 rebasing 할때나이럴경우에는예외일수도있습니다.

애초에 Git 은태초의 commit 으로부터많은혜택을받습니다: repostiory 가생성되자마자 HEAD 는 20 zero bytes 의스트링으로자동설정됩니다. 이특별한 commit 은빈나무로표현합니다. 빈나무는부모님 commit 도없습니다. 한마디로근본이없는친구를태초의 commit 으로부릅니다.

그리고태초의 commit 후, git log 를로드했을때 Git 이오류를내지않고단순히 commit 이하나도안되었다고알려줄것입니다.

태초의 commit 은한마디로 zero commit 의양자같은컨셉트입니다.

그러나이런구성은가끔문제를야기합니다. 여러개의 branch 가모두태초의 commit 을하고이제 branch 를병합시켜야할때, rebasing 은아마유저본인이수동으로버전청소를하라고할수도있습니다.

별난인터페이스

A 와 B commit 이있을때, "A..B" 와"A...B" 표현의차이는명령어가두개의종점이나범위가입력되기를기다리고있느냐마느냐입니다. **git help diff** 와 *git help rev-parse* 를참조하십시오.

부록 B: 이가이드를번역하기

다음절차들을따라하셔야제프로그램이자동으로빠르게 HTML 과 PDF 버전의번역본을만들수있습니다.

우선원본이되는소스를클론하시고폴더를만드시는데, 폴더명을 IETF 에기재되어있는태그를쓰시기바랍니다: 다음링크에들어가셔서확인하세요, the W3C article on internationalization. 예를들자면, 영어의태그는"en" 이고, 일본어는"ja" 이며, 국어는"ko" 입니다. 그리고새로운디렉토리에서"en" 에들어있는원본 txt 를번역하시면됩니다. (번역주: 영어를못하시고국어를하실수있다면본가이드를쓰시면되겠지만그럴확률이거의없겠죠)

For instance, to translate the guide into Klingon, you might type:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" is the IETF language code for Klingon.
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # Translate the file.
```

and so on for each text file.

Edit the Makefile and add the language code to the `TRANSLATIONS` variable. You can now review your work incrementally:

```
$ make tlh
$ firefox book-tlh/index.html
```

Commit your changes often, then let me know when they're ready. GitHub has an interface that facilitates this: fork the "gitmagic" project, push your changes, then ask me to merge.