git – 생활코딩

|  |
| --- |
| git1  깃의 3대 목적.  버전관리, 백업, 협업  (순서) 뒤가 앞에 의존. 어려워.  폴더에 깃 저장소 연결.  커밋하면 로그가 기록된다.  변경한 상태에서 이전과 비교 가능.  커밋하면 플래그 생성.  변경하고 커밋 안하고 도 변경하면 플래그는 없겠지.  내 컴퓨터에만 보관하면 언젠가는 그 파일이 유실될 수 있다.  그게 아니더라도 나는 노트북이 2대니까 이곳저곳 저장이 분산되게 된다.  원격 저장소. 로컬 저장소.  깃허브 사이트 – 내 컴퓨터.  깃허브에 저장한다. push  깃허브에서 가져온다. pull.  [커밋와 푸시의 차이는?]  커밋은 버전이 생성된다.  커밋하면 버전이 변경되었다며 push가 뜬다.  push를 누르면 버전이 저장되는 구조인가보다.  원격 저장소를 통해 push와 pull을 하며 협업이 이루어진다.  깃 허브 데스크탑. 프로그램.  git bash. 사용해보기.  git log [버전 로그 보여준다.]  git status [변경사항 알려준다.]  git commit -am "message" [커밋.]  [-m은 아는데 a는 뭐지?]  git push [업로드] |
| 파일 A,B를 수정했다.  파일 A만 따로 버전 관리 가능.  버전 관리하고 싶은 파일을 선택 – 스테이지에 올린다.  스테이지에만 올린 상태 – stage area  워킹 카피/디렉터리. 물리적 개념 – 폴더와 파일들.  stage area는 논리적인 개념인가보다.  정보 시스템에 따져봐야 할 것.  CRUD.    reset 버전 돌리기.  클릭한 상태로 되돌아가겠다는 뜻.  mixed나 soft를 선택하면 변경 안된다고 한다.  hard를 선택하자.  위험성.  인터넷-원격에 올라가 배포한 버전은  reset하면 안된다고 한다.  reset은 삭제하는 것.  삭제하지 못하는 경우가 있다.  기록을 실수해서 지우고자 한다.  그런데 지우면 안된다. 실수조차 기록..  어떻게 이 문제를 해결할 수 있을까?  실수했다고 기록하는 모양이다. 과정은 복잡하겠지만  이것이 revert.  revert는 reset보다 어렵다.  되돌리기.  함부로 reset하는 것 좋지 않다.  실수조차도 기록하기 revert.  오른쪽 마우스.  reset은 되돌아가고 싶은 버전을 클릭한다.  revert == reverse는 실수=기록을 지우고 싶은 버전을 클릭한다.  자동으로 커밋이 생성된다.    revert7 되어 있는 상태에서 M4로 revert하고 싶다?  바로 M4 revert 하면 안된다.  충돌이 일어난단다.  그럼 어떻게 할까?  revert 6과 5를 하면 M4가 된다.  이렇게 해야 한단다.  revert는 역순으로 하나씩 해나가는 거라는데..  최초에 할 때,도 겠지?  M1-7이 있는 상태에서?  M4를 revert하겠다는 의미는,  M4 이후의 모든 변경사항을 버리겠다는 뜻이 아니라고 한다.  M4에서 일어난 변경사항만!을! 되돌리겠다는 의미라고 한다.  그런데 M4만 지우면 그 이후의 사항에 영향을 주기 때문에  깃이 conflict 충돌을 발생시킨다고 한다.  단순히 과거의 상태를 보고싶은 거라면  더블클릭하면 된다고 한다.  그러면서도 버전을 삭제되지 않고 남아있다.  가장 최신 상태로 돌아가고 싶으면  왼쪽 네이게이션의 master를 더블클릭하면 된다.  왜 그런지는,  나중에 브런치, 헤드를 배우면 알게된다.  각각의 버전은 커밋이라고도 부른다.  각 커밋은 고유한 식별자가 있다. commit id, sha1  버전id, 리비전id  버전관리의 장점은.  변경 이전과 이후의 차이를 쉽게 비교할 수 있다는 점.  텍스트 파일은 아주 쉽게 확인할 수 있다.  그런데 한컴, 이미지 등은 비교가 어렵다.  비교할 수 있는 방법을 살펴보자.  스테이지에 올리기 전.  오른쪽 마우스. Open Current Version.  밑에 Open Selected Version 클릭.  각 버전에서 창을 열고 비교 가능하다.  외부 도구를 이용해서 비교하는 방법도 있다.  KDiff3 프로그램.    두 파일의 차이를 보여준다. 편리.  파일 이름 봐봐.  이런 프로그램을 소스트리와 연결할 수 있다.  소스트리 위쪽 네비게이션 [옵션]    프로그램 설치 후 선택하면 된다.  항목에 없는 것도 가능하다고 한다.    이전 상태와 비교하고 싶은 파일을 선택하고 (위) 클릭하면  연동된 프로그램으로 비교 가능하다고 한다.  음.. 한 번 해보면 알거같다.  버전 관리의 핵심은 비교.  비교를 통해서 과거를 되돌아볼 수 있다는 것이  버전 관리의 핵심적인 효용.  diff tool 검색하면  여러 비교 도구들이 있다고 한다.  차이점을 정교하게 비교하고  현재 상황을 신속하게 파악하는 데 정말 큰 도움이 된다고 한다.  버전 관리하지 않을 파일이 있을 수 있다.  .gitignore 파일을 만들고 거기에 파일 이름을 적으면 된다.  또 깃의 정말 중요한 특징,  깃의 혁신적인 면 중 하나는 branch.  마치 평행우주처럼  우리 저장소를 여러 가지 상태로 공존할 수 있게 해줍니다.  브랜치를 이용하면  저장소의 이름을 더럽히지 않고  하나의 저장소에서 다양한 작업을 진행할 수 있다.  사람이 보기 힘든 commit id에 대응하는 것이 tag  백업도 중요하다.  깃은 자체적으로 매우 편리하고 안전한 백업 장치를 가지고 있다.  나는 노트북이 두 대니까  깃으로 백업 연습을 해보면 좋을것 같다.  비공개 계정으로 해보자!  Sisi55는 공개 계정.  siCsds3은 비공개 계정.  백업에 익숙해지면  자연스레 협업이라는 기능이 보인다. |
| 깃 호스팅이란 작업하고 있는 로컬 저장소에 버전을 업로드할 원격 저장소를 임대해주는 비즈니스.  잘 이용하면 백업 서버인 원격 저장소를 운영할 수 있다.  복제. 클론.  원격 저장소와 지역 저장소가 같은 상태가 된다.  GitLab. 오픈소스 프로젝트에 대해서는 무료.  오픈소스라서 우리의 서버에 직접 설치할 수 있느냐.  GitLab은 private 저장소를 무제한으로 생성할 수 있다.  그래서 GitHub와 GitLab을 같이 사용하자.  GitLab에서도 저장소를 만들었다.  앞으로 어떻게 공부할 것언지 전략을 세워보자.  어떻게 지역 저장소와 원격 저장소를 연결할 수 있지, 하는 궁금증.  통신 방법 2가지.  http, ssh  우리는 http를 배운다.  http는 보안적으로 조금 부족하다. 조금 불편할 수도 있다.  근데 배울 필요가 없다. 그냥 할 수 있다.  ssh는 보안쪽으로 편리하고 강력하다.  하지만 배워야 할 것이 꽤 많다.  그리고 혼란스럽다.  원격 저장소와 지역 저장소를 연결하는 방법을 살펴봅시다.    [리파지토리]-[add remote] 선택하고 저장소 url 복붙.  기본적인 원격 저장소는 origin으로 하기로 약속되어 있다.  (그래서 origin이었구만)  지역 저장소에서 작업한 내용을 여러 저장소에 분산해서 필요에 따라 업로드 할 수 있다.  별명을 통해 저장소를 구분한다. ? (잘 모르겠군)  (내가 add remote를 해놓았던 것 같기도 하다.. 이거저거 하다가)  원격 저장소로 업로드 하는 push. |
| push를 누르면 나오는 창.  local branch가 master, remote branch가 master.  연결시켜서 올리겠다. 이게 track.  branch를 알면.. 일단 체크하고 넘어가자.    최초에 누구인지 물어보는 창이 나왔다고 한다.  내가 잘못 입력했던 듯.  github 등의 계정 id, password를 입력하면 된다.  백업을 할 수 있으면  복원도 할 수 있어야 한다.  깃의 아주 놀라운 점.  이동하면서 작업하는데 굉장히 큰 편리함을 얻게 된다.  백업 받은 것을 복원하는 방법을 살펴봅시다.    new tab을 클릭.  create는 완전히 새로운 저장소를 만들고 그 저장소를 소스트리로 관리하는 것.  add는 이미 존재하는 저장소를 소스트리로 관리하는 것. (로컬->소스트리)  clone은 원격 저장소를 복제해서 지역 저장소를 만들고 그것을 소스트리로 관리하는 것.  (원격->소스트리/로컬)  클론할 때, 없는 폴더 이름 입력하면 자동 생성해주나봐.  지역 저장소에서 작업한 내용을 원격 저장소로 push하고  다른 지역 저장소로 이동해서 작업을 한다면,  원격 저장소에 있는 내용을 지역 저장소로 가져와야 할 것이다. pull  다른 컴퓨터로 작업을 하려면,  원격 저장소에서 가져오고 작업을 시작해야겠다. pull 해야 한다.    pull을 클릭, ok 클릭.  깃을 통해 어마어마하게 많은 소스코드들을 사용할 수 있는 첫걸음.  깃도 깃허브 위에 올라가 있다.  깃도 깃을 통해 버전 관리해서 만들어지고 있는 프로그램.    clone or download 버튼.  일반인은 download zip 클릭.  우리는 배웠으니 url을 복사해서 복제할 수 있다. clone  깃은 모든 저장소가 소스코드 뿐만 아니라 버전에 대한 정보까지 모두 가지고 있다.  <더 살펴보기>  인증 관련해서 원격 저장소와 통신을 할 때마다 인증을 물어보는 불편함.  ssh를 통해 저장소간 통신하는 방법을 살펴봅니다.  ssh를 이용하면 자동으로 로그인 할 수 있습니다.  원격 저장소를 제공하는 깃 호스팅의 기능을 꼼곰히 살펴보는 것.  깃헙 깃랩 등 서비스는 프로젝트를 진행하는 데 도움이 될 만한  매우 편리한 기능을 제공하고 있다.  그 중 하나가 이슈 트래커.  프로젝트를 진행하면서 생기는 여러가지 이슈들  그런 이슈들을 공유할 수 있는 일종의 게시판. 또는 투두 리스트.  이슈 트래커를 이용하면,  프로젝트를 하면서 발생한 문제들을 기록할 수 있고,  각각의 문제에 대한 처리 상태를 마킹해서 처리해야 될 문제를 빠짐없이  체계적으로 관리할 수 있습니다.  또 협업 기능과 이슈 tracker를 결합하게 되면  업무를 분담해서 처리하는 강력한 도구로도 활용될 수 있다.  깃을 통해 동시에 작업을 했을 때 생기는 지옥들을 해결할 수 있다.  같은 부분을 동시에 수정하는 상황을 버전 관리 시스템에서는 충돌 conflict라고 한다.  충돌이 발생했을 때 정보가 유실되는 최악의 상황에서 벗어나도록 하는 도구를 버전 관리 시스템이 가지고 있다. |
| 깃 소스트리 위에서 브랜치와 충돌 기능을 다루는 방법을 살펴보자.  일을 하다보면 서로 다른 작업이 추가된 복제본이 필요한 경우가 있다.  최신 상태 코드 == 워킹 카피 ??  브랜치를 시작하려면 충돌을 극복해야 한다.  브랜치와 브랜치를 병합할 때.  같은 파일의 다른 부분을 수정한 파일 A,B는 깃이 알아서 병합해준다. 대박.  그런데 같은 파일의 같은 부분을 수정했다면 깃은 병합을 중지하고,  개발자에게 충돌이 났다며 수동으로 수정하라 한다. conflick. 충돌.  충돌이 발생했을 때 어떻게 해결할 수 있는가를 살펴봅시다.  충돌은 사고가 아니라 깃이 제공하는 가장 멋진 기능. |
| 브랜치를 만들고 만들어진 브랜치들을 열람하고 그리고 브랜치를 사용하는 법을 살펴봅시다.  브랜치를 이해하는 과정에서 브랜치만큼 중요한 것은  히스토리를 읽는 능력=독해력을 갖게되는 것.  어떤 일을 할 때마다 히스토리가 어떻게 바뀌는가를 잘 살펴봅시다.    브랜치의 목록은 이 부분에 보인다.  저장소를 만드는 순간부터 마스터라는 기본 브랜치 위에서 작업을 하게 된다.  브랜치 없이는 아무것도 할 수가 없다.    [브랜치] 클릭.    히스토리가 바뀐다.  master 옆에 새로 만든 브랜치 이름.    브랜치 계속 만들면 추가된다.    점이 있는 브랜치에 있다는 뜻. ms.  클릭하면 다른 브랜치로 이동한다.  master 브랜치를 클릭한 상태.  commit을 하면 마스터에 브랜치 된다.  다른 브랜치로 전환하고 – 클릭  master에 수정한 파일을 열어 보면  master에 적용된 결과가 나오지 않게 된다.  그니까 commit전의 상태.  버전관리 시스템에서 굉장히 중요한 것은 헤드.  헤드는 현재 속해 있는 상태를 보여주는 것.    ms에 있는 동그라미가 헤드.  즉 현재 ms 브랜치 상태에 있다는 의미.    동그라미 왼/오 같이 이동한다.  내가 현재 어떤 브랜치에 있는지는 헤드를 통해 알 수 있어야 한다.  헤드를 다른 곳으로 옮기는 행위를 체크아웃이라 한다.    오른쪽 마우스.  checked~ 체크아웃. 헤더를 옮기는 행위.  헤드를 통해 현재 우리가 가리키고 있는 브랜치를 찾을 수 있고  브랜치가 가리키고 있는 최신 커밋을 통해서  브랜치의 현재 상태를 알 수 있다. |
| 이번 시간은 merge에 대해 살펴본다.  커밋을 하면 버전이 만들어진다.  버전들은 기본적으로 마스터 브랜치에 속해 있다.  브랜치 a,m,g가 있다.  브랜치 a에서 작업하던 내용이  m에서도 유용할 것 같다.  그래서 합치는 작업을 하고 싶을 것이다.  a와 m의 공통의 조상을 base라고 한다.  합치는 것의 공통의 조상 base.  베이스를 바탕으로 새로운 버전을 만든다.  결과 만들어진 커밋을, 병합된 커밋이라고 한다.  merge commit  깃은 대부분의 작업을 자동화해주고  자동화가 논리적으로 불가능한 작업을  작성자에게 수동 요청한다.  merge 전에 브랜치를 지우는 법을 먼저 살펴보자.  마스터 브랜치만 살릴 것이므로  마스터 브랜치에 체크아웃하자.    그리고 오른쪽 마우스.  delete [브랜치] 클릭.    그리고 force delete해야 삭제된다.  이유는 아직 병합하지 않았는데 삭제하려니까,  유실될까봐 걱정하는 거다.  브랜치가 다른 파일을 생성하고  마스터가 그와 다른 파일을 수정하고  둘을 merge하면 어떤 일이 일어나는지 살펴보자.  merge는 순서가 중요하다.  누구를 누구에게 병합할 것인가를 정해야 한다.  기준을 정하라는 말이겠지.  브랜치를 마스터로 병합하려면  마스터로 체크아웃해야 한다.    현재 헤더를 기준으로  병합할 브랜치 오른쪽 마우스.  merge ~ into current branch. 클릭.  결과 마스터 브랜치는  다른 브랜치만 갖고 있던 파일을 갖게 된다.  이와 같이  서로 다른 브랜치가  서로 다른 이름의 파일을 갖고 merge했을 때  어떤 일이 일어나는지 살펴보았다.  서로 다른 브랜치가 서로 같은 파일을 수정했다.  같은 파일의 서로 다른 부분은 수정했다ㅣ  같은 부분을 수정했다. 경우.  깃의 반응을 살펴보자. 전자.  마스터 기준으로 깃이 알아서 병합해줬다.  병합 과정의 꽃이라고 할 수 있는 부분.  서로 다른 브랜치가 같은 파일의 같은 부분을 수정했을 때.  깃이 어떻게 반응하는지 살펴봅시다.  헤드는 체크아웃한 브랜치의  현재 가리키고 있는 버전.    충돌이 일어났다.  가운데 ===구분자를 기준으로    내가 작업한게 맞다면  스테이지 올리기 전 파일 오른쪽 마우스.  resolve conflict.    그중 내가 맞으면 mine  아니면 theirs    이것도 저것도 아니라면  <<< === >>> 기호를 모두 지우고  파일 수정 후  마우스 오른쪽, mark resolved. 해결되었음.  클릭하면 자동으로 스테이지에 올라간다.  커밋하면 자동으로 충돌이 있었음을  파일에 ?? 아닌듯 기록해준다.  남들이 볼 수 있게.  충돌에 대해 더 알아보자.  충돌은 브랜치와 브랜치를 병합할 때도 일어나고,  다른 사람과 협업을 할 때도 일어난다.  근데 협업이 브랜치를 쓰는 거다.  충돌의 상호아에서 자신감을 가지고  당황하지 않고 문제를 해결하려면  충돌의 속성을 잘 이해할 필요가 있다.  원리적인 이야기를 해보자.  깃을 어떻게 충돌을 파악하는가.  3 way merge  깃이 자동으로 병합해주는 부분과  우리가 수동으로 병합해줘야 하는 부분은 어디.    2 way merge  충돌이 일어난다.  하지만 좀 더 많은 것을 자동화하고 싶다.  그것을 보완하는 것이 3 way merge  here와 there의 공통 부분 base가 등장.  base는 공통의 조상.    base/here/there 모두 다르면 충돌.  base와 하나가 다르면 다른 하나가 수정되고  나머지는 그대로이기 때문에 수정된 것으로 바뀐다.  외부 도구를 이용하여 병합하는 방법을 살펴봅시다.  충돌이 일어난 부분은 base가 드러나지 않기 때문에  수정이 어려울 수 있다.  여러 가지 병합 도구들이 있다.  p4merge를 살펴보자.    충돌이 일어난 후  외부 도구로 merge를 클릭.  외부 병합 툴.  위의 왼쪽 local 가운데 base 오른쪽 remote  아래는 최종적으로 만들려는 결과.  저장하고 소스트리로 돌아오면 된다.  자동으로 백업 파일을 만들어주는데 지우면 된다.  이런 편리한 외부도구에 관심을 가지면  일하는게 즐거워질 것이다.  아주 가벼운 부분도 브랜치로 만들어 작업 완료한 후  필요없으면 버리고  필요하면 마스터로 병합한 후에 버린다.  브랜치는 버릴 수 있는 작업을 만들 수 있는 최선의 도구.  관심 가져볼 만한 주제.  브랜치의 기능을 배운 것만으로는 부족할 수 있다.  브랜치를 어떻게 나의 삶 속에 적용할 수 있을까.  그럴 때는 사례를 공부해 보면 좋다.  git workflow라고 검색하면  여러 사례가 나온다.  그 중 git flow를 찾아보면  개발자들에게 널리 채택된 모범 사례를 살펴볼 수 있다.  그리고 이 모범 사례를 우리가 쉽게 적용할 수 있도록  고안된 소프트웨어인 git flow라는 프로그램도 있다.  <https://boxfoxs.tistory.com/347>  참고하면  혼자서 어떤 규칙을 정하는 것보다  좀 더 많은 사람의 지혜를 참고할 수 있다.  깃 플로우는 소스트리에도 있으니  나중에 사용해보자.  cherry pick 기능도 살펴보자.  병합과 관련된 기능.    base 1에서  master a, other b.  other b2의 내용을 master에 병합하고 싶다.  cherry pick을 이용하면 된다.  cherry pick을 통해 부분적으로  병합할 수 있다.  rebase 기능도 좋다.  merge와 목표가 같다.  rebase를 이용하면 타임라인을 더 깔끔하게 할 수 있다.  b2,b3의 base를 a3로 하고싶은 것.  b2,b3를 따로 떼어 임시로.    그리고 b2를 a3와 병합하는 것. a4  그리고 a4와 b3를 병합하여 a5를 만든다.  결과적으로 병합한 효과를 내면서도  가지가 정리된다. 깔끔 – 이게 타임라인이 깔끔 |
| 이론적인 얘기를 해보자. 정리.  이 시간의 주인공은  헤드, 브랜치, 커밋.  간의 관계를 살펴본다.  위 모두 데이터인데  데이터를 조작하는 방법이 체크아웃.  과 리셋.  닮은듯 비슷한 체크아웃, 리셋.  저장소를 만들면 기본적으로 헤드가 만들어진다.  .git의 헤드 파일을 열어보면 master라고 적혀있고  우리가 작업을 한다는 것은  기본적으로 마스터 브랜치 위헤서 버전을 만들어가는 것.  마스터 브랜치 상태  ==마스터 브랜치에 체크아웃된 상태.    소스트리의 헤드. 점으로 표시된다.  브랜치를 만들 때  어떤 버전에서 시작할지 결정.  헤드를 보고,  연결된 마스터 브랜치를 보고  새로운 브랜치가 어디에서 시작하는지 알 수 있다.    새로운 브랜치로의 체크아웃을 통해서  현재 저장소의 브랜치를 바꾼 다는 것은,  헤드가 가리키고 있는 것을  마스터에서 다른 브랜치로 변경하는 것.  따라서 우리는 헤드를 보고,  헤드에 연결된 브랜치를 보고  현재 저장소가 해당 브랜치에 체크아웃 되어 있는 것을 알게 된다.  결국 체크아웃이란,  헤드의 값을 바꾸는 것이다.  checkout [버전이름]  헤드가 브랜치를 가리키지 않고  버전을 직접 가리키게 된다.  그리고 저장소는 해당 버전의 상태가 될 것이다.  이처럼 헤드가 브랜치를 가리키지 않고  커밋을 가리키는 상태를,  detached 상태에 있다고 한다. |
| 체크아웃과 리셋은 비슷한 느낌.  체크아웃은 뭔가를 바꾸는 느낌. 체인지.  지우는 느낌은 없고.  근데 리셋에는 지우는 느낌이 있다.  checkout master  reset master  를 비교해보면서,  reset과 master의 차이점을 살펴보자.  결과적으로,  체크아웃은 헤드를 제어한다. 가리키는 커밋 변경.  리셋은 헤드가 브랜치를 가리키고 있는 동안은  브랜치를 제어한다.  (브랜치를 변경?)  헤드가 브랜치로 체크아웃하면  우리 워킹카피는 해당 브랜치의 최신 상태의  버전으로 바뀐다.  즉, 체인지.  그럼 reset master는 어떤 의미를 갖게될까?  리셋은 브랜치를 바꾸는 작업.  헤드가 브랜치를 가리키고 있는 상태에서  reset master를 하면  해당 브랜치를 마스터가 가리키고 있는  버전으로 바꾼다는 뜻.  리셋은 브랜치를 변경한다.  브랜치가 가리키는 커밋을 바꾸면  그 커밋 이후의 링크는 끊겨버리게 된다.  끊긴 링크는 더이상 브랜치 소속이 아니게 되기 때문에  삭제의 느낌을 갖게된다.  보통 reset [버전이름]  과 같은 형태롤 리셋을 이용한다.  그러면 현재 체크아웃된 브랜치가  지정한 버전의 상태가 되게 된다.  (브랜치가 지정한 버전을 가리킴)  checkout [브랜치이름] |
| 깃은 여러 개의 저장소를 서로 연결시켜  상호 간의 동기화를 시킬수 있다.  깃의 정수. 협업  소스트리를 이용해 협업하는 방법을 살펴보자.  로컬 디렉토리를 만들고  소스트리의 create를 선택했다.  그럼 원격 저장소 생기는거?  나는 그동한 clone해서  내 로컬에 새로 폴더 만들었는데.  저장소를 create하고  [저장소]-[설정] 탭에 들어가서  원격 저장소 설정하면  내가 생성한 로컬 폴더-저장소와 연결된다.  이제 협업을 하는 단계로 넘어가보자.  내 저장소에  다른 사람이 push, pull할 수 있게 권한을 줘야 한다.  깃헙의 settings로 가자.      초대하는 사람의 id를 적으면 된다.  이메일이 전송된다.  그 사람이 링크클릭하면    초대장 사진. 초록버튼 누른다.  작업을 하기 전에 바람직한 것은  pull을 해서 원격 저장소의 버전들을 내려받는 것.  버전을 최소한의 단위로 만들도록  깃이 부탁한다..  충돌이 일어나면 늦게 커밋한 사람이 해결해야 한다.  충돌이 일어나면  수기로 파일 작성하고  해결했다. 클릭하면 된다.  full과 fetch.  비슷한 아이콘.  같은듯 다른듯한 기능.  패치가 무엇인가.  원격 브랜치가 무엇인가 살펴보자.    로컬/원격  오리진 마스터는  원격 저장소에서 마지막으로 가져온 버전.  오리진 마스터를 로컬 마스터와 같게 하려면  push를 하면 된다.  pull 대신 fetch를 하면  원격 저장소에 있는 내용을  지역 저장소와 병합하지 않고  가져만 온다.  지역 저장소가 원격 저장소보다 뒤쳐진 상태에서  원격 브랜치 오른쪽 마우스. merge  pull = fatch + merge |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |