보관소의 규율: 하루도 잃지 않는 버전 관리의 기술

git을 백업 도구 수준에서 쓰는 1인 개발자를 위한 실전 가이드

ThakiCloud (Hyojung Han)

보관소의 규율: 하루도 잃지 않는 버전 관리의 기술

git을 백업 도구 수준에서 쓰는 1인 개발자를 위한 실전 가이드

ThakiCloud (Hyojung Han)

Contents

1 되돌릴 수 있는 시간선: 백업 도구로 쓰는 git의 한계

대부분의 사람은 git을 한 가지 목적을 위해 시작한다. "만약 뭐가 깨지면"이라는 목적이다. 레포지토리를 만들고 GitHub에 push를 하고 끝. 이후로 "깨지면 클라우드에서 가져오면 되지"라는 정신적 모델로 살아간다.

이 정신적 모델에는 초반에 아무 문제가 없다. 하지만 문제는 git 난 실제로 할 수 있는 일의 극히 일부만 쓰고 있다는 것이다. 그리고 못 쓰고 있는 것을 모른 채로, 그 대가를 매일 조금씩 낸다. 이 장에서는 git을 백업 도구로 쓸 때 무엇을 잃는지부터 짚어본다.

1.1 혼자일수록 git이 필요한 이유

팀에서는 git의 가치를 협업으로 설명하는 경우가 많다. 하지만 1인 개발자에게 협업은 주된 이유가 아니다. 오히려 반대다. 혼자일수록 "나"라는 단점이 커진다.

코드를 고쳐보다가 어젯밤 버전이 좋았다는 걸 깨달을 때, 팀원에게 "어제 그건 어땠지?"라고 물을 수 있는 상대가 없다. 대신 물어야 할 상대는 "어제의 나". git이 없으면 어제의 나에게는 아무것도 물어볼 수 없다. 파일이 그대로일 뿐, 왜 그렇게 썼는지, 그때 어떤 상태였는지, 다음에 뭐부터 고칠지 아무것도 모른다.

즉, 1인 개발자에게 git은 "시간을 되돌릴 수 있는 기록장치"다. 이 장의 나머지는 그 기록을 실제로 쓰게 만드는 방법이다.

1.2 백업 정신은 세 가지 한계가 있다

백업은 “어제의 파일”을 준다. git은 “어제의 상태, 그 이유, 그 이후의 모든 상태”를 준다. 이 차이가 세 가지 실천적 한계로 드러난다.

1.2.1 한계 1: 되돌리는 것만 가능하고, 옆으로 움직이는 것은 불가능하다

수요일에 뭔가 깨졌다고 하자. 백업은 “수요일 복사본”을 돌려줄 뿐이다. 그런데 내가 실제로 원하는 답은 “이 함수가 마지막으로 정상일 시점이 언제였지?”인 경우가 많다.

구체적인 예를 들자. 결제 함수를 리팩토링한다. 커밋하고 테스트하고 깨졌다. 나는 이 리팩토링만 되돌리고 싶고, 그 이후에 한 다른 작업은 유지하고 싶다. 백업 도구로는 이 질문에 답할 수 없다. 파일 전체의 특정 시점 상태만 알 뿐, “부분을 되돌린다”는 개념이 없다. git에는 답이 있다. 그 함수만 마지막 정상 상태로 되돌리고, 유지하고 싶은 작업을 그대로 둔 채 실패한 리팩토링만 버릴 수 있다.

1.2.2 한계 2: 변경의 이유가 사라진다

백업 파일은 “파일의 복사본”이다. git 커밋은 “이유가 붙은 변경 기록”이다. 커밋에는 메시지가 있고 그 메시지에 왜 바꿨는지, 무엇을 바꿨는지 쓴다.

여섯 달 뒤, 코드 한 줄을 보며 “왜 이렇게 썼지?”라고 생각할 때, 나는 기억에 의존할 수밖에 없다. 커밋 메시지는 기억에 의존하지 않는 증인. 백업 도구는 증인이 없다. 그냥 그때의 파일 그대로를 돌려줄 뿐, 왜 그랬는지는 모른다.

1.2.3 한계 3: 실험을 믿을 수 없다

백업은 "저장"이므로, 큰 변경을 하면 긴장한다. 좋은 상태를 망가뜨리면 돌아올 수 없다고 생각하기 때문이다. 이 두려움은 "지금 안 저장하면 잃는다"는 정신에서 온다.

git은 구조가 다르다. 변경은 커밋되기 전까지는 "사실"이 아니다. 언제든 버릴 수 있는 작업 중인 상태다. 이걸 알면 실험 방식이 달라진다. 대담하게 리팩토링을 시도하고 안 되면 시작 전으로 돌아가면 된다. 다음 장에서는 이 부분을 더 깊이 다룬다.

1.3 git이 실제로 무엇인가: 커밋의 사슬

백업 도구의 안경을 벗고 git 레포를 잠시 들여다보자.

git 레포의 중심은 커밋의 사슬. 각 커밋에는 세 가지가 있다.

- 직전 커밋을 가리키는 포인터
- 그때의 프로젝트 전체 스냅샷, 즉 트리
- 메타데이터: 누가, 언제, 어떤 메시지로 만들었는지

커밋은 "스냅샷"이다. 이게 되돌리기를 안정적으로 만드는 이유다. 옛날 커밋으로 체크아웃하면 "역방향 패치를 적용하는" 게 아니라 "그 스냅샷으로 이동하는" 것이다. 파일 내용이 그때 그대로가 된다. 그래서 과거와 현재를 오가기가 자유롭다.

1.3.1 디테이치드 HEAD, 길을 잃은 게 아니다

"옛날 커밋으로 체크아웃하면"이라는 말에 한 가지를 덧붙이자. 그때 레포는 "디테이치드 HEAD" 상태가 된다. 그 상태에서는 서 있는 커밋은 보이지만, 브랜치 이름이 없다. 여기에서 커밋을 만들면, 그 커밋은 잠시 어떤 브랜치에도 속하지 않게 된다.

위험한 일이 아니다. 그 자리에 브랜치를 만들면(예: git branch rescue) 그 커밋은 rescue 브랜치의 것이 된다. 디테이치드 HEAD 상태는 "관찰 모드"에 가깝다. 과거를 보는 동안 서 있는 자리일 뿐이다. 중요한 것은 "지금 디테이치드 HEAD에 있다"는 사실을 아는 것이지, "길을 잃었다"는 공포가 아니다. git status가 디테이치드 HEAD라고 알려줄 때, 그 문장을 읽을 수 있으면 된다.

브랜치는 커밋을 가리키는 이름이다. 태그도 커밋을 가리키는 이름이다. HEAD는 지금 있는 브랜치의 이름이다. 이게 전부 "이름"이라는 걸 알면 git에 대한 두려움이 크게 줄어든다. 이름은 파일을 이동시키지 않는다. 이름은 사슬의 한 지점을 가리킬 뿐이다. 이름이 사라져도 커밋은 남는다. 그 이야기는 3장의 reflog에서 다룬다.

git 레포는 "작업의 시간선과, 그 시간선의 각 지점을 가리키는 이름들".

1.3.2 커밋 사슬의 생김새

건강한 레포의 git log -oneline는 이런 생김새를 가진다.

a3f9c2d fix(payment): prevent double charge on retry 8b21e47 feat(payment): add idempotency key c4d5e6a refactor(api): extract retry logic into helper 5f6a7b8 chore(ci): add test step 9c8d7e6 feat(api): add webhooks endpoint

여기서 해시는 예시로 지어둔 값이다. 중요한 것은 구조다. 다섯 줄을 읽으면 프로젝트의 이야기가 보인다. 웹훅을 추가했고 CI에 테스트 단계를 넣었고 리트리 로직을 뽑았고 결제에 멱등성을 붙였고 이중 과금 버그를 고쳤다. 만약 다섯 줄이 전부 "asdf", "asdfasdf", "asdfasdfasdf"였으면, 이 이야기는 사라진다. 파일은 그대로인데, 왜 그렇게 되었는지를 설명할 증인이 없는 셈이다.

커밋 해시는 40자리의 16진수다. -oneline은 그 앞의 일부만 표시한다. 몇

자리 앞자리를 공유하는 커밋이 생길 확률은, 한 프로젝트 안에서 매우 낮다. 그래서 앞의 몇 자리만 봐도 특정 커밋을 가리킬 수 있다.

1.4 1인 개발자의 세 가지 큰 손실

팀에서는 git의 가치를 협업으로 설명한다. 하지만 1인 개발자의 손실은 더 직접적이다. 세 가지로 정리할 수 있다.

첫째, 실험하지 못해서 잃는 손실. 망가뜨리면 안 되니까 새로운 방식을 시도하지 않는다. 아는 것만 반복한다. 1인 개발자의 생산성 성장은 "안 해본 것 시도하기"에서 크게 오는데, 상태를 잃을 두려움이 그 시도를 막는다.

둘째, 되돌리지 못해서 잃는 손실. 뭔가 깨졌을 때, 손으로 되돌리거나 백업에서 가져와서 다시 시작한다. 둘 다 시간이 걸리고 둘 다 "본래 안 잃을 다른 작업"까지 잃을 위험이 있다. 10분 걸릴 문제가 오후를 통째로 삼킨다.

두 번째 손실의 흔한 모양을 하나 그려보자. 설정 파일을 고치다가 다른 것을 깨뜨렸다. 그런데 어떤 줄이 문제인지 모른다. 백업과 지금 파일 사이를 왔다 갔다 하며 한 줄씩 비교한다. 5분 걸릴 변경이 3시간이 된다. 만약 그 설정 파일이 작게 작게 커밋되어 있었다면, 문제의 범위는 "이번 커밋 하나"로 좁혀진다. git show로 그 커밋만 보면 되는 일이다.

셋째, 설명하지 못해서 잃는 손실. 석 달 전에 쓴 코드를 보며 왜 그렇게 썼는지 모를 때, 선택지는 두 개다. 추측해서 리팩토링한다(깨질 위험이 있다), 아니면 그대로 둔다(채무가 쌓인다). 커밋 히스토리가 없으면 "확인한다"는 세 번째 선택지가 없다.

이 세 손실은 일회성 비용이 아니다. 매일 내는 비용. 그리고 매번 금액이 작아서 모른다. "되돌리느라 오후를 잃었다", "새 방식을 포기했다", "왜 이렇게 썼는지 기억하느라 두 시간을 썼다". 이렇게 쌓인다.

백업 도구 정신이 간과하는 비용이 하나 더 있다. 문제가 생겼을 때, 어디서부터 보아야 할지 모르는 비용이다. 버전 관리가 있으면 탐색은 "마지막으로 알았던 좋은 상태"에서 시작한다. 백업이면 탐색은 "지금 있는 파일과 기억"에서 시작한다. 시작점의 차이가 조사 시간의 차이를 정한다. 작은 프로젝트라면 그 차이가 몇 시간 정도로 보일 수 있다. 하지만 한 해를 쌓으면 상당한 양이 된다.

더 미세한 차이가 하나 있다. 역사가 있으면 과거의 결정에서 배울 수 있다. 이전에 같은 벽에 부딪쳤는지, 커밋 기록이 알려준다. 백업은 기억하지 않는다.

1.5 체크포인트의 규율: 매 정해진 지점에 커밋하기

이 손실을 없애는 규율은 단순하다. 저장할 가치가 있는 작업의 각 단계를 커밋으로 만든다.

"정해진 지점에 도달할 때마다" 커밋한다. 정해진 지점이 뭐냐면, 다음 세 조건을 만족하는 자리다.

- 동작한다: 테스트가 통과하거나, 최소한 문법 오류가 없다
- 작업의 의미가 온전히 담긴다: 무엇을 했는지 한 문장으로 설명할 수 있다
- 여기서 되돌려도 괜찮다: 이 지점 이전으로 돌아가도 잃을 게 없다

제안하는 일상 루틴은 이렇다. 아침에 git status와 git log -oneline -10을 실행해서 내가 서 있는 자리를 확인한다. 하루 중 정해진 지점에 도달할 때마다 커밋한다. 메시지에 매달리지 않는다. 짧게 쓰고 커밋한다. 퇴근 전에 git log -oneline로 하루의 커밋을 돌아본다. 거기서 "오늘 내가 뭘 했는지"가 보이지 않으면, 커밋이 너무 굵다. 각 커밋이 무엇을 했는지 모르겠으면, 메시지가 너무 빈약하다.

이 루틴은 "그냥 저장하기"처럼 보인다. 하지만 백업과 다르다. 각 커밋에

이유와 지점이 있으므로, 하루의 작업은 "이동할 수 있는 시간선"이 된다.

"정해진 지점"에 도달하지 못한 채로 다음 작업으로 넘어가야 할 때는 stash로 작업 중 상태를 잠시 옆에 두고 진행한다. 다만 stash한 걸 당일 내로 다시 꺼내는 습관을 들인다. 하루를 넘기면, 그 stash는 또 다른 "이게 뭐였지" 뭉치가 된다.

실제 하루를 그려보자. 1인 개발자가 작은 서비스 일을 하는 화요일이다. 아침에 git status와 git log -oneline -10으로 서 있는 자리를 확인한다. 먼저 어제 보고받은 버그를 고친다. 테스트가 통과하는 지점에 커밋한다. 다음으로 새 기능을 시작한다. 뼈대가 서서 동작하는 지점에 또 커밋한다. 오후에는 새로운 방식을 시도하는데, 방향이 틀렸다. 이때 당황하지 않는다. 한 명령으로 직전 커밋으로 돌아와 틀린 방식을 버린다. 틀린 방식은 reflog에 남아 있다. 나중에 다시 보면 된다는 뜻이다. 저녁에는 하루의 커밋을 다시 읽고, 오늘 무엇을 했는지 읽어 본다. 읽힌다면 성공이다.

다만 한 가지만 분명히 하자. 목적이 "커밋을 많이 하는 것"은 아니다. 오타 한 자 고칠 때마다 커밋하라는 말이 아니다. 의미 없는 상태를 계속 커밋하면, 시간선이 시끄러워진다. 이야기가 읽히지 않고 결국 다시 "뭉치"로 돌아간다. 기준은 위 세 조건이다. 동작하고 의미가 온전하고 여기서 되돌려도 괜찮으면 커밋할 가치가 있다. 그중 하나라도 안 되면, 조금 더 기다리거나 stash를 쓰는 것이 맞다.

1.6 "깨지면 가져오면 되지"에서 "언제든 되돌린다"로

백업 정신	버전 관리 정신
파일을 되찾는다	상태로 이동한다
이유가 없다	이유가 붙는다
변경이 두렵다	실험이 자유롭다

"깨지면 가져오면 되지"는 미래에 대한 말법이다. "언제든 되돌린다"는 현재에 대한 말법이다. git은 현재를 준다. 거쳐온 모든 상태의 기록, 각 상태의 이유, 어떤 상태로든 이동할 수 있는 이름.

체크포인트의 규율을 유지하면, 하루의 작업은 "시간선"이 된다. 그리고 시간선은 파일과 다르게, 되돌리고 갈라지고 이름 붙일 수 있다. 다음 장에서는 이걸 가능하게 만드는 규율을 쌓기 시작한다. 먼저 레포의 형체부터. 무엇을 넣고 무엇을 내보내야 하는지.

2 보관소의 설계: 무얻을 넣을지, 커밋은 어떻게 쓸지

레포지토리는 장소다. 커밋과 브랜치의 규율보다 앞서, 더 기초적인 질문이 있다. 이 장소는 무얻을 위한 것인가. 이 장에서는 두 가지를 다룬다. 먼저 레포의 형체, 무얻을 넣고 무얻을 내보낼지. 그리고 커밋의 언어, 메시지를 어떻게 쓸지.

2.1 레포의 범위 결정

팀에서는 한 레포냐 여러 레포냐의 논쟁이 복잡하다. 1인 개발자에게는 실용적인 답이 있다.

기본값: 한 제품, 한 레포. 서비스, 웹 앱, 공유 코드, 스크립트. 이들이 함께 빌드되고 함께 배포된다면, 한 곳에 두는 것이 마찰을 줄인다. “라이브러리를 따로 릴리스하고 따로 버전 관리하고 싶다”는 욕구가 생겼을 때, 그때 나누는 것을 고려하면 된다.

나눌 때의 신호: - 릴리스 주기가 다른 부분. 여러 프로젝트에서 쓰는 라이브러리 같은 것 - 접근 권한이 다른 부분. 특정 고객만 보는 비공개 설정 같은 것 - 언어나 스택이 다르고 빌드가 독립적인 부분

합칠 때의 신호: - 레포 사이를 계속 복사 붙여넣기 한다 - 함수 시그니처를 바꾸면 두 레포에 영향을 주는 교차 리팩토링이 있다 - 레포 간 의존성을 수동으로 관리하고 있다

결정이 안 되면 한 곳에 둔다. 레포를 나누는 것은 일방통행이다. 나누기는 해도, 다시 합치기는 아프다. 합추는 되돌릴 수 있다. 폴더를 레포 사이로 옮기는 일이다.

레포에 전혀 들어갈 것이 아닌 것도 있다. 내 머신 설정, 개인 메모, 실험용 스크래치 파일. "misc" 폴더에 이것들을 모으는 사람이 있다. 석 달 뒤, 그 폴더 속 파일이 작업 산출물인지 개인 메모인지 구분하지 못한다. 스크래치 작업은 레포 밖에서 한다. 다른 디렉터리, 또는 다른 비공개 레포. 보관소는 "새 기기에 이 프로젝트를 다시 세우려면 필요한 것"만 들어야 한다.

2.1.1 건강한 레포 구조의 예

1인 개발자의 서비스 레포는 이런 모양을 가질 수 있다.

app/ src/ tests/ web/ src/ scripts/ deploy.sh backup.sh docs/ runbook.md .env.example .gitignore README.md package.json

이것은 예시 구조다. 언어나 스택에 따라 모양은 달라진다. 중요한 것은 기준이다.

- 이 프로젝트가 필요한 것이 모두 여기 있다
- 나만 필요한 것은 여기에 없다
- README가 처음 시작하는 법을 알려준다

scripts는 자주 놓친다. 배포 스크립트, 유지보수 스크립트, 일회용 스크립트. 레포에 없으면 역사에도 없다. 석 달 뒤 배포가 뭘 하는지 기억나지 않을 때, 스크립트를 보관소에 두고 있었음을 아쉬워하게 된다.

##.gitignore는 보관소의 앞문

.gitignore는 "무얻이 레포에 들어오는지"를 결정한다. 두 원칙이 있다.

첫째, 다시 만들 수 있는 것은 넣지 않는다. node_modules, dist 같은 빌드 산출물은 소스에서 항상 다시 만들 수 있다. 넣으면 레포가 비대해지고 클론이 느려질 뿐이다.

둘째, 나만 해당하는 것은 넣지 않는다. 로컬 설정은 "레포의 진실"로서 의미가 없다. 넣으면 레포는 "누구나 가질 것"과 "나만 가질 것"이 뒤섞인

곳이 된다.

범주로 보면: - 빌드 산출물: node_modules, dist, build, target - 로컬 설정:.env, local.json, IDE 설정 - 시크릿: API 키, 인증서, 개인 키 - OS나 편집기 잔여물:.DS_Store, 스왑 파일

흔한 실수: 먼저 커밋하고 나중에.gitignore로 지우려 하는 것..gitignore는 "새로운 것"만 막는다. 이미 추적 중인 파일은 git rm -cached로 명시적으로 제거해야 한다. 그리고 그 파일은 역사에 남는다. 시크릿처럼 정말 들어와서는 안 될 것을 커밋했다면, 파일을 지우는 것만으로 끝내지 말라. 4장의 역사 재작성 섹션을 읽어 보자.

참고: git rm -cached는 "추적은 끊되, 파일은 디스크에 남긴다"는 명령이다. 파일은 제자리에 있다. 다음 커밋부터 git이 그 파일을 보지 않는다. -cached 없이 git rm을 쓰면 파일 자체가 디스크에서 지워진다. "이 파일은 더 이상 추적하지 말고, 파일은 남겨둬" 싶을 때 -cached 형태를 쓴다.

2.2 시크릿은 절대 보관소에 넣지 않는다

"API 키, 데이터베이스 비밀번호, 인증서. 넣으면 안 된다는 건 알지만 나중에 교체할게요." 그 "나중에"가 시크릿이 새는 순간이다. 규칙:

- 로컬 설정은.env에 넣고,.env는.gitignore에 든다
- 템플릿은.env.example에 두고 진짜 값 대신 플레이스홀더만 쓴다
- 커밋해 버렸다면, 키를 먼저 교체한다. 역사 정리는 그 다음이다

많은 사람이 이 순서를 놓친다. 키가 여전히 유효한 동안 역사에서 키를 지우느라 몇 시간을 쓴다. 키를 교체하면 역사 정리 문제가 사라진다. 역사를 지우되 키를 교체하지 않으면, 문제는 미루어질 뿐이다.

.env.example에 하나만 더 하자. 이 파일은 "새로 온 사람이 진짜 값에

손대지 않고 프로젝트를 돌릴 수 있게 하는" 파일이다. 각 줄에 그 값이 무얻용인지 주석으로 적자. "데이터베이스 주소, 로컬에서는 쓰지 말 것" 같은 것을 적는 것이다. 예제 파일은 문서의 일부다.

2.3 커밋의 언어: Conventional Commits

Conventional Commits는 커밋 메시지의 형태를 표준화하는 규약이다. 오픈 소스 세계에서 태어나, 지금은 넓게 쓰인다. 형태는 한 줄이다.

type(scope): subject

type은 어떤 변경인지. 흔한 것들: - feat: 새 기능 - fix: 버그 수정 - docs: 문서 - refactor: 동작을 바꾸지 않는 재구성 - test: 테스트 추가나 수정 - chore: 빌드나 도구 작업

scope은 어디에서였는지, 생략 가능. subject는 무얻을 했는지, 한 문장.

규칙 몇 가지: - 지시형으로 쓴다. "로그인을 추가했다"가 아니라 "로그인 추가" - subject는 짧게. 한 줄에 안 들면, 커밋이 너무 크다 - 이유는 본문(body)에 넣고 subject에는 넣지 않는다

1인 개발자는 "규약은 팀용이고 나는 필요 없다"고 자주 생각한다. 그것은 아니다. 규약의 가치는 "남이 읽는 것"이 아니라 "미래의 내가 찾는 것"이다. type이 있으면 git log -grep=fix로 버그 수정만, git log -grep=feat로 기능 작업만 볼 수 있다. 메시지는 내가 나를 위해 만드는 검색 인덱스다.

본문을 쓸 때는 언제인가. 이유가 스스로 명백하지 않을 때다. 예를 들어 원인이 잘 안 보이는 버그를 고쳤다면, 본문에 "버그가 어떻게 일어났는지, 무얻으로 찾았는지"를 적자. 여섯 달 뒤 같은 종류의 버그가 다시 나오면, 그 본문이 조사 시작점이 된다.

메시지 예: - feat(auth): add refresh token rotation - fix(web): prevent double submit on slow networks - docs(readme): explain local

setup steps - chore(ci): update build scripts

메시지 본문은 여러 줄이 될 수 있다. 예:

fix(payment): prevent double charge on retry

The retry path re-sent the request after a timeout. The idempotency key was generated per request. Derive the key from the order id instead.

본문에 왜를 길게 쓰고 subject에는 무얻을 짧게 쓴다. 원인이 복잡한 버그라면, 본문은 조사 노트가 된다. 다음 사람이 아니라, 다음 날의 내가 그 노트를 읽는다.

2.4 하나의 커밋에는 논리적 변경 하나

커밋에는 하나의 변경 이유만 들어야 한다. "논리적 변경 하나" 규칙이다.

예: - 함수 시그니처를 바꾸고 모든 호출부를 함께 고치는 것: 하나의 커밋 - 리팩토링 중 변수를 바꾸면서 새 기능을 덧붙이는 것: 두 개의 커밋 - 코드를 바꾸고 그 코드의 테스트를 추가하는 것: 보통 하나의 커밋 - 기존 동작을 문서화하는 회귀 테스트만 추가하는 것: 따로 하나를 만들 수 있다

분할하는 법: 1. 변경에 "준비 작업"(이름 변경, 파일 이동)과 "변경 그 자체"가 함께 있으면, 준비 작업을 먼저 하고 커밋한 뒤 변경을 한다 2. 깔끔하게 나누지지 않는다면, 변경이 너무 크다는 신호다. 작업 자체를 나누는 것을 고려한다

2.4.1 커밋을 나누는 예

에러 처리 방식을 바꾸는 작업을 하자. 중간에 변수 이름도 바꾸고 싶다. 리트라이 기능도 추가하고 싶다. 순서가 중요하다.

1. 변수 이름만 바꾼다. 커밋: refactor(err): rename error variables

2. 에러 처리를 바꾼다. 커밋: fix(err): return structured error instead of string
3. 리트라이를 추가한다. 커밋: feat(err): add retry with backoff

각 커밋은 하나의 이유를 가진다. 나중에 리트라이를 없애려면, 3번 커밋만 되돌리면 된다. 1번과 2번은 그대로 남는다. 세 가지를 한 커밋에 넣으면, 되돌릴 때 세 가지가 같이 사라진다.

실용적인 기준: 하나의 커밋 diff가 300줄을 넘는다면, 나눌 수 있을지스스로에게 물어 본다.

구체적인 예를 하나. API 응답의 날짜 형식을 ISO에서 로컬 형식으로 바꾸고 싶다. 변경은 세 이유를 가진다. 직렬화 코드를 고치고 테스트를 고치고 README에 변화를 문서화한다. 세 이유, 세 커밋이 맞다. 세 이유가 한 커밋에 섞이면, 석 달 뒤 날짜 버그가 나면 “어떤 커밋에서 형식이 바뀌었는지”를 바로 찾지 못한다. diff를 쿠게 들여다보며 추측하게 된다. 세 커밋이면, git log -grep에서 형식 변경 커밋을 몇 초 만에 찾는다.

세분된 커밋의 가치: - git revert가 정확해진다. “그 변경”만 되돌릴 수 있다 - git bisect가 빨라진다. 작은 커밋 50개 사이에서 “깨진 지점”을 빠르게 찾는다 - 역사가 읽힌다. git log가 프로젝트의 이야기가 되고, “큰 변경” 뭉치가 되지 않는다

2.5 5분 커밋 루틴

제안하는 하루의 흐름:

1. git status(30초): 무얻이 바뀌었나
2. git diff(2분): 한 번 읽어 본다. 의도한 것만 포함되어 있는지 확인
3. git add를 의도적으로: 전부 아니라, 이 변경에 속하는 파일만
4. git commit: Conventional 형태의 메시지로 쓴다
5. git log -oneline -5: 최근 역사에 어떻게 들어갔는지 확인

가장 중요한 것은 2단계다. 빨리 커밋하는 사람은 diff를 건너뛰고, “커밋할 의도가 아니었던 것”이 레포에 들어간다. 한 번 역사에 들어가면, 지우기가 어렵다. 2분 diff 검토는 싼 보험이다.

습관 하나 더: 큰 변경 전에 커밋한다. 리팩토링 전에, 의존성 업그레이드 전에, 새로운 방식 시도 전에, 지금의 안정된 상태를 먼저 커밋한다. 깨지면 그 자리로 한 명령 만에 돌아온다. 이게 안전한 실험의 가장 기본적인 형태다.

2.6 push는 언제 하는가

remote(GitHub, GitLab 등)가 진정한 백업이다. 로컬 레포는 한 디스크 위의 사본이다. 디스크가 죽으면 로컬 사본은 사라질 수 있다. remote는 기계를 건너간 사본이다.

규칙: - 커밋한 뒤, 안정적이라면 push한다 - 하루가 끝나면 push한다 - 큰 변경 전에 push한다

자주 push하는 비용은 거의 없다. 커밋 몇 개면 git push는 빠르다. 얻는 것은 하나다. 그날의 작업은 안전하다고 말할 수 있는 것.

push 빈도를 높이면, 3장의 규칙이 더 중요해진다. push한 커밋은 다시 쓰지 않는다.

2.7 요약: 레포는 두 번째 뇌

잘 설계된 레포는 세 질문에 답한다. - 이 장소는 무얻을 위한 것인가 (범위) - 무얻이 들어 있고 무얻이 안 들어 있는가 (.gitignore) - 왜 바뀐 것인가 (커밋 메시지)

이 세 답이 분명하면, 다음 장의 브랜치 규율이 가능해진다.

3 실험자를 위한 브랜치: 안전하게 시도하고 깨끗이 돌아오기

1장에서 "변경은 커밋되기 전까지는 사실이 아니다"라고 했다. 이 장은 그 사실을 어떻게 이용하는지 이야기한다. 그게 브랜치. 1인 개발자에게 브랜치는 "팀 협업 기능"이 아니라 "무료로 쓰는 샌드박스"다.

3.1 브랜치는 포인터일 뿐

데이터 모델을 다시 살펴보자. 커밋은 사슬 위의 노드. 브랜치는 그 노드를 가리키는 이름. git branch new-feature는 파일을 복사하지 않는다. 지금의 커밋을 가리키는 새 이름 하나를 만든다.

이게 브랜치 만들기의 비용을 사실상 0으로 만든다.

- 복사 비용이 없다
- 동기화 문제가 없다
- 브랜치를 지우는 것은 이름을 지우는 것일 뿐

그래서 정신 모델이 달라진다. "브랜치 만들기가 무섭다"에서 "브랜치는 싸니까 많이 만든다"로. 브랜치에 대한 두려움은, 1인 개발자의 실험 습관을 막는 가장 큰 장애물이다. 브랜치는 git이 주는 가장 싼 보험.

한 가지를 분명히 하자. 브랜치를 지워도 커밋은 사라지지 않는다. 그 커밋이 다른 브랜치에서 도달 가능하다면(병합했다면) 그대로 남는다. 도달 불가능하다면, reflog가 잠시 붙들고 있다. "브랜치를 잘못 만들었다"는 실수의 비용이 아주 작다는 뜻.

3.2 main에서 바로 하는 작업, 언제 괜찮은가

브랜치를 항상 만들라는 규칙은 없다. main에서 바로 커밋해도 되는 경우: - 작고 결과가 이미 정해진 변경. 오타 수정, 문서 한 줄, 설정 값 한 곳 - 실패해도 되돌리기 쉬운 변경 - 실험이 아니라, 할 일이 분명한 작업

브랜치가 필요한 경우: - 결과가 불확실한 작업 - 여러 파일을 건드리는 작업 - 이렇게 해도 될까를 묻고 있는 작업

구분이 애매하면, 브랜치를 만든다. 브랜치는 싸다. 만들다 보니 필요가 없으면, 지우면 된다. 지울 때의 비용은 거의 없다.

3.3 브랜치를 잘라야 하는 세 가지 상황

모든 일에 브랜치가 필요하지는 않다. 하지만 세 가지 상황에서는, 브랜치를 자르는 것이 정확한 판단.

상황 1: 큰 변경 앞에서. 리팩토링, 의존성 업그레이드, 구조 조정. 이런 작업은 많은 파일을 건드리고 결과가 불확실하다. main을 먼저 안정 상태로 커밋하고 브랜치를 잘라서 작업한다. 실패하면 브랜치를 지우면 된다. main은 그대로.

상황 2: 여러 방식을 동시에 시도하는 것. 두 가지 방식이 있고 어느 쪽이 맞는지 모를 때. A 방식은 A 브랜치에서, B 방식은 B 브랜치에서 시도한다. 둘 다 해보고 비교한다. 브랜치가 싸니까, "둘 다 해보는 것"을 감당할 수 있다.

이때는 "시도"라는 것이 보일 수 있는 이름 규칙을 쓴다. try/redis-cache, try/in-memory-cache. try/ 접두사는, 나와 미래의 나에게 "이는 확인된 작업이 아니라 실험이다"라고 말한다.

상황 3: 독립적인 주기를 가진 작업. 여러 날에 걸쳐 하는 작업, main 흐름과

분리해 두고 싶은 버그 수정. main에서 하면, "작업 중"과 "안정 상태"가 섞인다. main은 "언제나 배포 가능한 것"이라는 의미를 잃는다.

간단한 규칙 하나: main은 언제나 배포 가능한 상태여야 한다. 그렇게 보장할 수 없다면, main은 main이 아니라 작업대. 작업대의 일은 브랜치가 한다.

3.3.1 stash와 브랜치의 경계

stash와 브랜치는 비슷한 문제를 푼다. "지금의 작업 중 상태를 다음으로 넘겨야 하는데, 잃고 싶지 않다"는 문제. 다만 쓰임이 다르다. stash는 상태가 잠깐일 때. 브랜치는 상태가 며칠을 산다.

기준을 하나. 내일 다시 이 작업으로 돌아올 것 같으면, 브랜치. 30분간 책상을 비워야 할 뿐이면, stash. stash는 주차장이고 브랜치는 작업실이다.

3.4 브랜치 이름 짓기: 규약은 나에게 남기는 메모

권장하는 패턴: - fix/짧은설명: 버그 수정 - feat/짧은설명: 새 기능 - try/짧은설명: 실험 - chore/짧은설명: 유지보수

짧게 쓴다. 브랜치 이름은 라벨이지, 보고서가 아니다. 문장이 필요하다면, 그건 커밋 메시지나 issue에 쓰는 것이지, 브랜치 이름이 아니다.

습관 하나: 브랜치의 첫 커밋에 주석을 남긴다. git branch는 이름을 보여줄 뿐, "왜 이 브랜치를 만들었는지"는 이름에 없다. 브랜치의 첫 커밋 본문에, 브랜치가 존재하는 이유를 한 줄 쓴다. 석 달 뒤, try/redis-cache라는 이름만 남아 있는 브랜치를 봤을 때, 그 한 줄이 기억을 대신한다.

3.5 merge와 rebase: 기준은 하나

브랜치의 작업을 main으로 가져오는 방식은 두 가지다.

git merge는 브랜치의 커밋을 그대로 두고, 두 사슬을 잇는 "병합 커밋"을 하나 만든다. 역사는 "갈라졌다가 다시 만난다"는 모양이 된다.

git rebase는 브랜치의 커밋을 들고 main 위에 하나씩 다시 재생한다. 역사는 "한 줄의 직선"이 된다.

기준은 하나다: 이 역사는 다른 이와 공유되고 있는가.

- 나만 아는 브랜치(push 전): rebase. 직선 역사가 깔끔하다
- 다른 이가 아는 브랜치(push 후, 사용 중): merge. 남의 역사를 다시 쓰는 것은 분쟁의 씨앗이다

1인 개발자의 브랜치는 대부분 개인적이다. 그래서 대부분의 경우 rebase가 맞다. 다만 예외는 분명하다. 한 번 remote에 push한 브랜치는, 다시 쓰지 않는다. push한 브랜치를 정리하고 싶다면, merge로, 또는 새 커밋으로 한다.

rebase의 절차: 1. git checkout main 2. git pull로 main을 최신으로 3. git rebase main으로 브랜치의 커밋을 main 위에 재생 4. 충돌이 나면: 해결하고 git add, git rebase -continue 5. 포기하고 싶으면: git rebase -abort

5단계가 rebase의 안전장치. rebase가 엉망이 되어도, 시작 전 상태로 돌아갈 수 있다. 이게 있다는 걸 알면, rebase에 대한 두려움이 크게 줄어든다.

경고 하나: git pull은 기본값으로 merge를 한다. rebase를 원한다면, git config -global pull.rebase true로 한 번 설정해 둔다. 그 뒤 git pull은 "가져오기 전 재생"이 된다. 역사가 직선을 유지한다. 기본 merge를 선호하는 사람도 있다. 어느 쪽을 고르든, 한 가지로 고정하는 것이 중요하다.

rebase 충돌의 실제 모양을 하나. main에서 함수 시그니처가 바뀌었고 내 브랜치에서 그 함수를 불렀다면, git은 충돌하는 첫 커밋에서 멈춘다. 충돌

파일이 열리고 양쪽의 변경이 표시된다. 파일을 고치고 git add, 계속한다. 더 이상 고치기 싫다면, abort한다. abort하면 rebase 전의 상태로 완벽히 돌아간다. "반쪽이 rebase"라는 상태는 남지 않는다.

3.5.1 rebase -interactive: 커밋을 다시 다듬기

push 전 브랜치의 커밋 메시지가 엉망이라면, git rebase -interactive로 다시 다듬을 수 있다. 이 명령은 최근 N개의 커밋을 나열하고, 각 커밋 앞의 동사를 고르게 한다. pick, edit, squash, drop.

- pick: 그대로 둔다
- edit: 그 커밋에서 멈추고 메시지나 내용을 고친다
- squash: 이전 커밋에 합친다
- drop: 버린다

예를 하나. 브랜치에서 세 개의 커밋을 만들었다고 하자. 첫 번째는 wip, 두 번째는 fix typo, 세 번째는 기능 추가. 첫 커밋과 두 번째 커밋을 squash로 합치고, 메시지는 기능 추가의 메시지로 고친다. 결과는, 하나의 정리된 커밋. push 전에만, 이 작업은 자유롭게 한다.

3.5.2 버그가 main에 들어왔을 때: revert

main을 배포하고 버그가 나타났다고 하자. 여기서는 역사를 다시 쓰지 않는다. git revert 커밋을 쓴다. 이 명령은, "그 커밋의 변경의 반대"인 새 커밋을 하나 만든다. 역사는 바뀌지 않는다. 새 커밋이 위로 쌓인다.

reset은 왜 아닌가. reset은 main의 포인터를 이동시킨다. remote의 main과 로컬의 main이 갈라진다. 다른 클론도 갈라진다. revert는 누구와도 갈라지지 않는다. 공개된 일은 revert로, 공개 전의 일은 reset이나 rebase로. 3장의 기준과 같다. 공유되느냐, 안 되느냐.

3.6 엉망이 되었을 때: reflog

브랜치를 지웠다고 하자. rebase 중에 커밋을 잃었다고 하자. 잘못된 커밋으로 체크아웃했다고 하자. "사라졌다"는 생각은, 거의 항상 틀리다.

git은 HEAD의 모든 이동을 reflog에 기록한다. "HEAD가 어디를 거쳤는가"라는 로컬 로그. 기본값으로 90일 정도를 유지한다.

회복 절차: 1. git reflog: 잃은 커밋을 찾는다 2. git show 커밋: 그것이 그 커밋인지 확인 3. 회복: - main의 커밋: git reset -hard 커밋 - 지운 브랜치의 커밋: git branch rescue 커밋 - 작업 중 상태: git stash list로 stash를 확인

git fsck -lost-found도 떠돈 커밋을 찾는다. 하지만 reflog가 대부분의 경우를 덮는다.

주의: reflog는 로컬이다. remote와 동기화되지 않는다. remote에만 있는 브랜치의 커밋을 잃었다면, 로컬 reflog만으로 부족할 수 있다. 1인 개발자의 작업은 대부분 로컬에서 시작하므로, reflog는 믿을 수 있는 마지막 안전망.

3.6.1 reflog 읽는 법

git reflog의 출력은 이런 모양이다.

HEAD@{0}: checkout: moving from main to fix/login HEAD@{1}: commit: fix(login): keep session after refresh HEAD@{2}: rebase (finish): returning to refs/heads/fix/login HEAD@{3}: commit: feat: add logout button

한 줄 한 줄이, HEAD의 이동 기록. 최신순으로, 위부터 아래까지 읽는다. checkout: moving from A to B, commit: 메시지, reset: moving to 같은 표현이 뒤에 온다.

잃은 커밋을 찾을 때, 이 목록에서 멈춘 자리의 앞 줄을 본다. 브랜치를

지웠다면, 그 브랜치에서 마지막으로 만들었던 커밋의 줄을 찾는다. 그 커밋의 해시로 새 브랜치를 만든다. git branch rescue 해시. 끝.

3.7 주간 의식: 죽은 브랜치 정리

더 이상 안 쓰는 브랜치는 채무. git branch가 길어지면, "내가 어디에 있는가"를 찾기가 어려워진다. 주 1회:

1. git branch로 모든 브랜치를 나열
2. 각 브랜치에 질문: 끝났다(병합 됐다)인가, 아직 필요한가
3. 병합 된 것: git branch -d 이름
4. 정말 필요 없는, 병합 안 된 것: git branch -D 이름
5. remote 브랜치: git fetch -prune

-d와 -D의 차이는 중요하다. -d는 "병합 됐다"는 브랜치만 지운다. 경위다. 확신이 있으면 -D를 쓴다.

정리할 때, 하나만 더 물어본다. 이 브랜치의 작업 중, 살릴 만한 조각은 없는가. 방향이 틀렸어도, 쓸 수 있는 조각이 있다면, cherry-pick으로 뽑아낸다. git cherry-pick은, 그 하나의 커밋을 지금 브랜치에 넣는다. 죽은 브랜치는, 쓸 수 있는 조각의 창고가 되기도 한다.

3.7.1 하나의 실제 스토리

화요일 아침. main은 안정적이다. 새 캐시 계층을 시도하기로 하고 try/cache-v1 브랜치를 만든다. 오후에 방식 A가 느린 것을 확인한다. main으로 돌아가, try/cache-v2를 만든다. 수요일에 방식 B가 낫다는 것을 확인한다. v2의 작업을 main으로 가져온다. 브랜치가 나만 아는 것이었기 때문이다. v1 브랜치는 지운다. try/ 이름이, 지워도 되는 실험이었음을 말해준다. 목요일, git branch를 보면 main뿐. 청결하다.

이 스토리의 핵심은 하나. 브랜치를 만들고 비교하고 버리고 남기는 일주일

동안, main이 한 번도 불안한 상태가 되지 않았다는 것.

3.8 요약: 브랜치는 결정의 기록

브랜치는 "작업을 둘 자리"가 아니라 "결정의 기록"이다. "이것을 시도하기로 했다"는 브랜치를 자르는 것이고, "시도하지 않기로 했다"는 브랜치를 지우는 것이다. 결정이 레포에 남아 있으면, 작업을 포기해도 잃는 것이 없다. 그게 실험의 규율.

4 역사의 규율: 태그, 정리, 그리고 일상 루틴

1장에서 3장까지는 “기록하기”와 “실험하기”를 다뤘다. 이 장은 남은 두 가지를 다룬다. “표시하기”(태그)와 “정리하기”(역사 관리). 그리고 모든 것을 일상 습관으로 묶는 루틴.

4.1 태그: 릴리스를 가리키는 계약

브랜치는 “작업 선의 최신 커밋”을 가리킨다. 태그는 “이름이 붙은 특정 커밋”을 가리킨다. 차이가 있다. 브랜치는 움직인다. 커밋할 때마다 진전한다. 태그는 움직이지 않는다. 붙인 순간의 커밋에 남는다.

이것이 태그를 “릴리스 표식”으로 만드는 이유다.

git tag v1.2.0은 “가벼운 태그”를 만든다. 이름뿐이다. git tag -a v1.2.0 -m “메시지”는 “아노테이티드 태그”를 만든다. 메시지, 서명한 사람, 날짜가 있다. 릴리스에는 항상 아노테이티드 태그를 쓴다.

왜인가. 가벼운 태그는, 브랜치와 “릴리스로서 구별”이 되지 않는다. 아노테이티드 태그는, “이것은 릴리스다”라고 말하고, “왜”를 함께 운반한다. 여섯 달 뒤 git tag -l을 볼 때, 아노테이티드 태그의 메시지가, 그 릴리스가 무엇일 위한 것이었는지 알려준다.

1인 개발자의 릴리스 흐름: 1. main이 안정적임을 확인한다. 테스트가 통과한다 2. 버전을 올린다. package.json 같은 곳 3. 올린 것을 커밋한다. chore(release): bump version to 1.2.0 4. git tag -a v1.2.0 -m “이 릴리스에 들어간 것의 요약” 5. 태그를 push한다. git push origin v1.2.0 6. 배포한다

태그는 보통 push로 가지 않는다. 명시적으로 push해야 한다. 또는 git

push -tags를 쓴다.

“태그 먼저, 배포 뒤”의 순서가 중요하다. 배포를 먼저 하고 태그를 뒤로 하면, “배포한 게 어느 커밋이었지?”라는 혼동이 생길 수 있다. 먼저 태그로 커밋을 고정하고 그 다음에 배포한다. 배포가 실패하면, 롤백할 버전이 정확히 알린다.

Semver, major.minor.patch는, 널리 쓰이는 버전 규칙이다. 엄격히 지킬 필요는 없다. 하지만 원칙은 유익하다. - patch: API를 바꾸지 않는 버그 수정 - minor: 뒤로 호환되는 새 기능 - major: 깨지는 변경, 이주가 필요한 것

major를 올릴 때, 태그 메시지에 “이 변경이 누구를 위한 것인가”를 적는 것은, 1인 프로젝트에서도 좋은 습관이다.

4.1.1 git describe: 나는 어디에 있는가

git describe -tags는, “가장 가까운 태그에서 얼마나 떨어졌는지를 알려준다. 예: v1.2.0-14-ga3f9c2d는,”v1.2.0에서 14개 커밋 뒤, 현재 a3f9c2d”라는 뜻이다.

배포해 둔 빌드의 버전이 어떤 것인지 모를 때 유익하다. 사용자가 버그를 보고했을 때, “어떤 버전이세요?”라고 물으면, 모호한 “요즘 것” 대신, 정확한 답을 얻을 수 있다.

4.1.2 태그와 버전 번호

태그와 버전 번호는 다른 것이다. 버전 번호는 코드 안에 쓴다. package.json의 version 필드 같은 곳. 태그는 커밋에 붙은 이름이다. 둘은 따로 갈 수 있다.

따로 가는 경우는 두 가지다. 버전 번호를 바꾸고 태그를 잊는 것. 태그를 만들고 버전 번호를 바꾸는 것을 잊는 것. 릴리스 흐름에서 “커밋 먼저, 태그

뒤"를 두는 이유가 바로 이 따로 가는 것을 막기 위한 것이다. 점검 하나: main에서 git describe --tags를 하면, manifest의 버전과 같은 값이 나와야 한다.

4.1.3 릴리스를 되돌리는 일

배포한 태그에서 문제가 나왔다고 하자. 여기서도 역사를 다시 쓰지 않는다. git revert 커밋을 쓴다. revert는, "그 커밋의 반대"인 새 커밋을 만든다. 태그는 그대로다. 릴리스 v1.2.0이 있었고 그 뒤에 v1.2.1이 나온 것이다.

reset으로 push된 버전의 main을 되돌리면, remote와 로컬이 갈라진다. 배포 기록도 혼란스럽다. revert는 아무도 갈라놓지 않는다. 공개된 일에는 revert, 공개 전의 일에는 reset이나 rebase. 3장의 기준과 같다.

4.1.4 git bisect: 깨진 지점을 찾다

"예전에는 돌아갔다"는 것을 알고, "이제는 돌아가지 않는다"는 것을 안다. 어느 커밋에서 깨졌는지는 모른다. git bisect가 찾는다.

1. git bisect start
2. git bisect bad: 지금의 상태가 bad
3. git bisect good 커밋: 그 커밋은 good
4. git이 중간 커밋으로 체크아웃한다
5. 테스트하고 good이냐 bad이냐를 답한다
6. git이 반복한다. 50개 커밋이면, 약 6단계로 찾는다

이분 탐색이다. git이 계산하고 나는 테스트만 한다. "어딘가에 있을 텐데"를 "이 커밋이다"로 바꾸는 도구다.

4.2 역사 재작성의 철칙

"정리"는, 커밋을 이동하거나 다시 쓰는 일이다. 도구: - git reset: 브랜치 포인터를 이동시킨다 - git rebase: 커밋을 재생한다 - git commit -amend: 가장 최근 커밋을 고친다 - git filter-repo: 전체 역사를 다시 쓴다

철칙: 내가 유일한 사용자인 역사만 다시 쓴다. - push 전의 로컬 커밋: 자유롭게 다시 쓴다 - push 되었지만 이 레포를 쓰는 것은 나뿐인 경우: 다시 쓸 수 있다. 다만 조심한다. remote는 git push -force-with-lease로 갱신한다 - 다른 이가 쓰는 레포: 다시 쓰지 않는다. 새 커밋으로 추가한다

-force-with-lease와 -force. -force는 remote를 무조건 덮어쓴다. -force-with-lease는, "내가 마지막으로 본 이후에 remote가 움직이지 않았다면"에만 덮어쓴다. 남의 작업을 덮어쓰는 것을 막는 경위다. 항상 -force-with-lease를 쓴다.

역사에 시크릿이 들어간 경우, 한 가지를 더. 시크릿을 커밋하고 git rm -cached와 amend로 지웠다고 하자. 시크릿은 역사에 여전히 있다. 진짜로 없애려면, 역사를 다시 써야 한다. 그리고 remote에 사본이 남아 있을 수 있다. 가장 중요한 것은, 키를 교체하는 것. 역사 정리는 그 다음이다. 이 순서를 자주 놓친다. 역사 정리에 시간을 쓰고 키 교체를 잊는다. 먼저 교체하는 것이 안전하다.

filter-repo를 쓸 때는 언제인가. 시크릿이 역사 깊은 곳에 있고 전체를 다시 써야 할 때. 그때의 일은, 로컬 레포만 정리하는 것이 아니다. remote 레포를 대체하고, 모든 클론을 다시 받아야 한다. git에서 가장 비싼 작업이다. 피할 수 없을 때만 한다.

4.2.1 커밋을 되돌리는 실제 예

커밋을 했는데, 다시 하고 싶다. 흔한 경우: 커밋에 쓸데없는 파일이 들어갔다, 메시지가 틀렸다.

1. git reset -soft HEAD~1: 커밋은 취소된다. 변경은 남아 있다
2. git add: 넣을 것을 고른다
3. git commit: 다시 커밋한다

-soft이면 아무것도 잃지 않는다. “커밋된 상태”만 취소된다. 위험한 것은 -hard다. -hard는 변경도 지운다. 아직 저장되지 않은 작업 위에 -hard를 쓰는 것은, 스스로를 지우는 일이다.

4.3 아침과 퇴근, 그리고 주간 루틴

모든 것을 합쳐, 제안하는 루틴.

아침(2분): 1. git status: 나는 깨끗한 상태인가 2. git log -oneline -5: 나는 어디에 서 있는가 3. 커밋되지 않은 작업이 있으면: 결정한다. 커밋, stash, 아니면 계속

이 아침 점검은, “나는 오늘 어떤 상태인가”에 답한다. git에 대한 두려움의 상당 부분이, 내가 어디에 있는지 몰라서 생긴다. 2분의 status 확인이 그 두려움을 없앤다.

4.3.1 아침 점검의 실제 모양

git status를 돌렸더니, 이렇게 나왔다:

On branch main Your branch is up to date with ‘origin/main’. nothing to commit, working tree clean

깨끗하다. git log -oneline -5를 돌리면, 어제 커밋이 순서대로 보인다.

반대로, status가 커밋되지 않은 변경을 여러 개 보인다면, 신호다. “어제, 일을 반쯤 남겨둔 채로 끝냈다”는 신호. 오늘 결정한다. 어제의 작업을 끝내서 커밋할지, stash에 둘지. 둘 중 하나로 고르면, 하루는 알려진 상태에서 시작한다.

퇴근(5분): 1. 저장할 가치가 있는 것은 git add와 git commit으로 2. remote가 있다면 git push 3. git log -oneline -10으로 이번 주를 돌아본다

퇴근 커밋은, 하루의 마지막 체크포인트다. 작업이 끝나지 않아도, 안정된 상태라면 커밋한다. 안정되지 않았다면, WIP라고 쓴 커밋, 또는 stash. 다음 아침, 나는 미스터리가 아니라, 알려진 상태에서 시작한다.

주간(10분): 1. git branch로 죽은 브랜치를 정리한다. 3장의 의식 2. git stash list: 오래된 stash를 확인한다. 꺼내거나 지운다 3. git fetch -prune: remote 브랜치를 정리한다 4. git log -oneline -since="1 week ago": 이번 주의 이야기를 한 번 읽는다

주간 정리는, "레포를 잘라내는" 일이다. 자르지 않는 레포는, 이름이 많은 곳이 된다. 브랜치, 태그, stash, 역사. 모두 쌓인다. 그리고 어느 날, 나는 내가 어디에 있는지 모르게 된다.

4.3.2 30일 시작 계획

이 규율을 처음부터 다 쓰기는 어렵다. 30일 계획을 제안한다.

- 1주: 5분 커밋 루틴만. 퇴근 push만. "하루가 깨끗한 상태로 끝나는" 느낌을 익힌다
- 2주: 큰 변경 전에 브랜치를 만든다. try/ 접두사를 쓴다
- 3주: 주간 정리를 한 번 돌린다. stash와 죽은 브랜치를 확인한다
- 4주: 태그를 하나 만든다. 릴리스 흐름을 한 번 끝까지 돈다

30일이면, "안정 지점마다 커밋"은 더 이상 생각해야 하는 규칙이 아니다. 반사가 된다. 규율은 규칙으로 시작해서, 반사로 끝난다.

4.4 1인 개발자의 20개 명령어

실제로 필요한 명령어. 그룹별로.

상태 확인(5): - git status: 무엇이 바뀌었나 - git diff: 변경은 무엇인가 - git log -oneline -10: 최근 역사 - git branch: 브랜치 목록 - git stash list: stash 목록

커밋(4): - git add 파일: 커밋 대상으로 표시 - git commit -m "메시지": 커밋 - git commit -amend: 최근 커밋을 고친다 - git reset -soft HEAD~1: 마지막 커밋을 취소한다. 변경은 남는다

이동(5): - git checkout 브랜치: 브랜치 전환 - git checkout 커밋: 특정 커밋으로 이동 - git merge 브랜치: 브랜치를 병합 - git rebase 브랜치: 브랜치를 재생 - git reset -hard 커밋: 브랜치를 커밋으로 이동. 변경을 버린다

회복과 정리(6): - git reflog: HEAD가 어디를 거쳤나 - git revert 커밋: 변경을 없애는 새 커밋을 추가 - git stash: 작업 중 상태를 옆에 둔다 - git stash pop: 옆에 둔 것을 다시 꺼낸다 - git branch -d 이름: 병합된 브랜치를 지운다 - git tag -a 이름 -m "메시지": 아노테이티드 태그를 만든다

20개다. 전부 외우는 것이 목표가 아니다. "상황이 오면, 어떤 명령어를 뻗치는지" 아는 것이 목표다. 처음에는 이 목록을 보면 된다. 몇 주 지나면, 그룹 자체가 머리에 들어 있다.

한 가지만 더. 최신 git은, checkout의 일을 나누었다. 브랜치 전환은 git switch, 파일 복원은 git restore. git checkout는 둘 다 하므로, 혼동할 수 있다. checkout를 계속 써도 된다. switch와 restore로 넘어가도 된다. 넘어간다면, 같이 넘어가는 것이 낫다. 섞어 쓰는 것은 더 혼란스럽다.

4.5 마지막 말: 규율은 습관이다

버전 관리의 규율은, 지식의 문제가 아니다. 반복의 문제다. 5분의 커밋 루틴, 2분의 아침 점검, 10분의 주간 정리. 어느 것도 어렵지 않다. 어려운

것은, 매일 하는 것.

결과는 처음에 은은하다. “하루도 잃지 않았다”, “깨졌을 때 바로 돌아왔다”, “석 달 전에 왜 그렇게 썼는지 기억났다”. 이런 작은 승리가 쌓인다. 어느 날, 나는 git이 두렵지 않다는 것을 알아차린다. 두려움은 미지의 것에서 오고 미지는, 레포가 깨끗하고 역사가 읽힐 때 사라진다.

보관소는, 작업이 기록되는 장소다. 그 장소를 돌보면, 작업이 스스로 가벼워진다.