시간을 다루는 기술

타임존, 스케줄, 만료가 무너지는 자리

ThakiCloud (Hyojung Han)

시간을 다루는 기술

타임존, 스케줄, 만료가 무너지는 자리

ThakiCloud (Hyojung Han)

Contents

1 시간은 하나의 타입이 아니다

시간 버그는 대체로 조용하다. 널 참조처럼 그 자리에서 터지지 않고, 몇 달 뒤 정산서의 숫자가 하루치 어긋난 모습으로 나타난다. 로그를 뒤져도 예외 하나 없다. 코드는 시키는 대로 정확히 계산했고, 다만 우리가 시킨 것이 우리가 원한 것이 아니었을 뿐이다.

이런 사고의 뿌리는 거의 언제나 같은 자리에 있다. 우리는 시간을 하나의 타입으로 다룬다. 데이터베이스에 `timestamp` 컬럼 하나를 만들고, 코드에서 `Date` 하나를 돌려쓰고, API로는 문자열 하나를 주고받는다. 그런데 현실의 시간은 최소 세 종류이고, 이 셋은 연산 규칙이 서로 다르다. 다른 것을 같은 그릇에 담으면 언젠가 반드시 잘못된 연산이 통과한다.

1.1 세 가지 타입

첫째는 순간이다. 우주 어디에서 보든 같은 하나의 점, 흔히 에포크 기준 초나 밀리초로 표현하는 값이다. 서버 로그의 기록 시각, 결제가 승인된 시각, 파일이 수정된 시각이 여기 속한다. 순간끼리는 뺄 수 있고, 그 결과는 흘러간 시간의 길이다. 순간은 지역 개념이 없다. 서울에서 보든 두바이에서 보든 그 결제는 같은 한 점에서 일어났다.

둘째는 벽시계 시각이다. 달력과 시계판이 가리키는 표현, 이를테면 2026년 3월 8일 오전 2시 30분이다. 이 값은 그 자체로는 순간이 아니다. 어느 지역의 시계판인지 모르면 지구상의 어느 점인지 특정할 수 없고, 심지어 어떤 지역에서는 그런 시각이 아예 존재하지 않기도 한다. 알람, 회의 예약, 마감 기한, 근무 교대표가 여기 속한다.

셋째는 기간이다. 여기서 다시 두 갈래로 나뉜다. 90분처럼 물리적 길이가 고정된 양이 있고, 한 달이나 하루처럼 달력에 의존해 길이가 달라지는 양이

있다. 앞의 것은 순간에 그대로 더할 수 있다. 뒤의 것은 벽시계 시각에만 더할 수 있고, 그 결과를 다시 순간으로 바꾸려면 지역 규칙을 한 번 더 거쳐야 한다. 하루는 24시간이 아니다. 일광절약이 시작되는 날의 하루는 23시간이고, 끝나는 날은 25시간이다.

1.2 섞이면 어디가 무너지나

세 타입을 하나로 뭉개면 다음 같은 코드가 자연스러워 보인다. 구독 갱신일을 계산한다며 현재 시각에 30일을 더하는 코드다. 사용자에게는 매달 같은 날 청구된다고 안내해 놓고, 실제로는 30일 주기로 청구된다. 2월이 낀 해에는 청구일이 앞당겨지고, 1년이 지나면 날짜가 닷새쯤 밀린다. 고객센터로 문의가 들어오기 전까지 아무도 모른다.

반대 방향의 사고도 흔하다. 응답 지연을 재겠다며 요청 시작과 끝의 시스템 시각을 뺀다. 그 사이에 시각 동기화가 시계를 뒤로 돌리면 지연이 음수가 된다. 음수를 걸러내는 방어 코드를 넣으면 이번에는 지연 통계에서 특정 구간이 통째로 사라진다. 측정에 쓸 도구를 잘못 고른 것인데, 증상은 지표 이상으로 나타나 원인을 엉뚱한 곳에서 찾게 만든다.

가장 비싼 실수는 미래의 약속을 순간으로 저장하는 것이다. 반년 뒤 오전 9시에 열리는 회의를 예약받아 그 자리에서 순간으로 변환해 저장한다고 하자. 그 사이에 해당 지역의 일광절약 규칙이 바뀌면 회의는 오전 8시나 10시에 열린다. 저장된 값은 그대로인데 세상의 규칙이 바뀐 것이다. 이건 버그를 심어 놓고 시한을 걸어 둔 것에 가깝다.

1.3 저장 규칙 한 장

실무에서 판단이 갈릴 때 쓸 기준은 짧다. 이미 일어난 일인가, 앞으로 일어날 일인가를 먼저 묻는다.

성격	저장 형태
이미 일어난 사실	순간 (UTC)
미래의 약속	벽시계 + 지역 이름
물리적 길이	초 단위 정수
달력 주기	단위와 수량

이미 일어난 일은 다시 바뀌지 않는다. 규칙이 어떻게 개정되든 그 결제는 그 점에서 일어났으므로 순간으로 못 박는 편이 안전하다. 반면 미래의 약속은 사람의 의도이고, 사람은 시계판을 기준으로 약속한다. 오전 9시에 만나자는 말은 그때의 오전 9시를 뜻하지 지금 계산해 둔 어떤 점을 뜻하지 않는다. 그러니 벽시계 값과 지역 이름을 나란히 저장하고, 실제로 알림을 보낼 때 순간으로 환산한다.

지역 이름이라고 쓴 자리에는 반드시 시간대 데이터베이스의 식별자를 넣는다. 아시아 서울, 아메리카 뉴욕 같은 이름이다. 여기에 오프셋 값을 넣으면 안 되는데, 그 이유는 다음 장에서 자세히 본다.

1.4 경계에서 한 번만 변환한다

타입을 나눴다면 그 다음 규칙은 변환 지점을 좁히는 것이다. 시스템 안쪽에서는 순간만 흐르게 하고, 사람이 읽는 표면과 사람이 입력하는 표면에서만 지역 시각으로 오간다. 입력 경계에서 한 번 순간으로 바꾸고, 출력 경계에서 한 번 지역 시각으로 바꾼다. 그 사이 어떤 계층도 시간대를 알 필요가 없다.

이 규칙이 깨지는 전형적인 자리가 있다. 로그 포맷터, 리포트 생성기, 이메일 템플릿, 데이터 적재 배치처럼 나중에 붙는 주변부다. 이런 코드는 대개 급하게 작성되고 서버의 기본 시간대를 그냥 쓴다. 그러면 같은 데이터가 서비스 화면에서는 맞고 리포트에서는 하루 어긋나는 상황이

생긴다. 두 값이 다르다는 사실 자체를 아무도 눈치채지 못한 채 몇 분기가 그냥 흘러간다.

방어는 코드 규칙보다 환경 규칙이 효과적이다. 모든 실행 환경의 기본 시간대를 UTC로 고정하고, 기본 시간대에 의존하는 함수 호출을 금지 목록에 올린다. 지역 시각이 필요한 함수는 지역 이름을 인자로 반드시 받게 만든다. 인자를 생략할 수 있게 두면 언젠가 누군가는 생략한다. 그리고 그 코드는 정작 개발자의 노트북에서는 잘 돌아간다.

1.5 이름이 절반이다

마지막으로 값싸면서 효과가 큰 습관 하나. 시간을 담는 필드 이름에 타입과 기준을 박아 넣는다. `created_at` 대신 `created_at_utc`, `due` 대신 `due_local_wall`, `timeout` 대신 `timeout_ms`처럼 쓴다. 컬럼명 하나에 정보를 실어 두면 여섯 달 뒤에 그 필드를 처음 보는 사람이 잘못된 연산을 시작하기 전에 멈춘다.

이름 규칙은 컴파일러가 강제해 주지 않는다. 그래서 더 일찍 정해야 한다. 프로젝트 초기에 세 타입의 이름과 저장 형태를 문서 한 쪽으로 못 박고, 코드 리뷰에서 그것만 확인해도 시간 버그의 상당 부분은 애초에 들어오지 못한다.

1.6 직렬화에서 잃어버리는 정보

타입을 잘 나눠 놓고도 경계를 넘는 순간 정보가 사라지는 경우가 있다. 직렬화 형식 때문이다. 문자열로 바꾸는 순간 순간인지 벽시계인지 구분이 흐려지고, 받는 쪽은 자기 편한 대로 해석한다.

가장 흔한 사고는 오프셋 없는 문자열을 주고받는 것이다. `2026-03-08 09:00:00` 이라고만 적힌 값은 아무 뜻도 없다. 보내는 쪽은 서울 기준으로 적었고 받는 쪽은 UTC로 읽는다. 두 시스템 모두 잘 동작하며, 매일 아홉

시간씩 어긋난 데이터를 쌓는다. 그러니 순간을 실어 보낼 때는 오프셋이나 Z 표기를 반드시 붙이고, 붙지 않은 값은 파싱 단계에서 거절한다. 관대하게 받아 주는 파서는 친절한 게 아니라 사고를 유예할 뿐이다.

숫자로 보낼 때도 함정이 있다. 에포크 값은 단위가 형식에 드러나지 않는다. 초와 밀리초를 섞어 쓰는 두 시스템을 연결하면 어떤 값은 1970년대로, 어떤 값은 수만 년 뒤로 해석된다. 다행히 이 오류는 티가 크게 나서 대개 개발 중에 잡힌다. 문제는 마이크로초와 밀리초를 섞을 때다. 천 배 차이는 눈에 띄지만 조용히 정밀도만 잘리는 변환은 눈에 안 띈다. 필드 이름에 단위를 박아 두라는 앞의 조언이 여기서 값을 한다.

미래의 약속을 실어 보낼 때는 형식 자체가 달라야 한다. 순간을 담는 문자열 하나로는 지역 이름을 전달할 수 없으니, 벽시계 값과 지역 이름을 각각의 필드로 나눠 보낸다. 하나의 문자열에 억지로 욱여넣으려는 시도는 결국 자체 규격을 만들게 되고, 그 규격을 아는 사람이 팀을 떠나면 아무도 손대지 못하는 코드가 된다.

1.7 반복 일정은 값이 아니라 규칙이다

세 타입에 더해 실무에서 자주 등장하는 네 번째 형태가 있다. 반복이다. 매주 화요일 오전 10시, 매월 마지막 영업일, 분기 첫날 같은 것들이다.

반복 일정을 만나면 미리 계산해 목록으로 저장하고 싶은 유혹이 든다. 앞으로 2년치를 펼쳐 저장해 두면 조회가 간단해지니까. 그런데 이 선택은 규칙이 바뀌는 순간 비용을 청구한다. 사용자가 시간을 30분 옮기면 저장된 수백 건을 모두 손봐야 하고, 이미 지나간 항목과 앞으로 올 항목을 갈라내야 하며, 그 사이에 개별적으로 수정된 예외 건은 덮어쓰지 말아야 한다.

그래서 반복은 규칙으로 저장하고 필요할 때 펼치는 편이 낫다. 저장하는 것은 시작 시각과 반복 규칙, 그리고 예외 목록이다. 예외는 두 종류다. 특정

회차를 건너뛴 기록과 특정 회차만 다른 시각으로 옮긴 기록이다. 이 구조를 잡아 두면 규칙 변경이 한 행 수정으로 끝난다.

물론 조회 성능이 필요하면 펼친 결과를 캐시로 둘 수 있다. 중요한 건 무엇이 정본인가다. 정본은 규칙이고 펼친 목록은 파생물이다. 이 방향이 뒤집히는 순간, 다시 말해 펼친 목록을 사람이 직접 고치기 시작하는 순간부터 두 값은 서로 다른 이야기를 하기 시작한다.

1.8 이 장에서 가져갈 것

시간을 다루는 코드를 처음 설계할 때 결정해야 할 것은 라이브러리가 아니다. 어떤 값이 순간이고 어떤 값이 벽시계이며 어떤 값이 규칙인지를 먼저 정하는 일이다. 그 구분이 서 있으면 좋은 라이브러리는 도와주고, 구분이 없으면 어떤 라이브러리를 써도 잘못된 연산을 막아 주지 못한다.

다음 장은 이 구분이 가장 험하게 시험받는 자리로 들어간다. 지역과 규칙이 개입하는 순간, 즉 타임존과 일광절약이다.

2 타임존과 일광절약이 무너뜨리는 것들

한국에서만 서비스한다면 이 장을 건너뛰어도 될까. 아쉽게도 아니다. 우리가 쓰는 클라우드 리전은 여러 나라에 걸쳐 있고, 결제 대행사는 다른 나라 기준으로 정산 파일을 내려주며, 데이터 파이프라인의 스케줄러는 UTC로 돈다. 한 지역만 쓰는 서비스에서도 시간대는 이미 시스템 안에 여러 개 들어와 있다. 다만 눈에 보이지 않을 뿐이다.

2.1 오프셋은 타임존이 아니다

가장 먼저 정리할 오해는 이것이다. 플러스 9시간은 타임존이 아니라 어느 한 순간의 오프셋이다. 타임존은 지역의 이름이고, 그 이름 뒤에는 오프셋이 시간에 따라 어떻게 변해 왔고 앞으로 어떻게 변할지 정하는 규칙 묶음이 붙어 있다.

이 차이는 과거를 다룰 때 곧바로 드러난다. 한국은 지금 플러스 9시간을 쓰지만 늘 그랬던 것은 아니다. 과거 몇 차례 표준시가 바뀌었고 일광절약을 시행한 시기도 있었다. 오래된 기록을 다루면서 오프셋을 상수로 박아 두면, 그 상수가 적용되지 않던 시기의 데이터가 조용히 어긋난다. 통계 그래프에서 특정 연도만 이상한 모양이 나오는데 원인을 찾기 어려운 경우, 상당수가 이런 종류다.

미래를 다룰 때는 더 직접적이다. 각국은 정치적 결정으로 표준시와 일광절약 시행 여부를 바꾼다. 발표에서 시행까지 몇 주밖에 남지 않는 경우도 있다. 시간대 이름으로 저장한 예약은 규칙이 갱신되면 자동으로 올바른 순간을 가리키지만, 오프셋으로 저장한 예약은 옛 규칙에 붙들려 남는다.

그래서 규칙은 짧다. 저장하는 것은 아시아 서울 같은 지역 식별자이고,

오프셋은 계산 결과일 뿐 입력이 아니다. 사용자에게 보여 줄 때 오프셋을 함께 표시하는 것은 좋지만, 그 표시값을 다시 읽어 들여 계산의 근거로 삼으면 안 된다.

2.2 존재하지 않는 시각과 두 번 오는 시각

일광절약이 시작되는 날, 시계는 특정 시각에서 한 시간을 건너뛴다. 그 사이의 벽시계 시각은 그날 그 지역에 존재하지 않는다. 끝나는 날에는 반대로 같은 벽시계 시각이 두 번 온다. 한 번은 여름 규칙으로, 한 번은 겨울 규칙으로.

문제는 라이브러리들이 이 상황에서 각자 다르게 행동한다는 점이다. 어떤 것은 예외를 던지고, 어떤 것은 조용히 한 시간 뒤로 밀어 주며, 어떤 것은 두 후보 중 앞의 것을 고른다. 문서를 읽지 않으면 어느 쪽인지 알 수 없고, 대부분의 코드는 이 경로를 한 번도 실행해 보지 않은 채 배포된다.

여기서 필요한 것은 정책이다. 라이브러리의 기본 동작에 맡기지 말고, 우리 도메인에서 무엇이 맞는지 정해서 코드에 적는다.

상황	권장 처리
없는 시각	뒤로 밀기
두 번 오는 시각	앞의 것 선택
사용자 입력	즉시 되물어 확정

알람이라면 없는 시각을 뒤로 미는 편이 자연스럽다. 사용자가 두 시 반에 깨워 달라고 했는데 그 시각이 사라졌다면 세 시 반에 깨우는 쪽이 아예 건너뛰는 쪽보다 낫다. 반대로 정산 마감처럼 놓치면 안 되는 일은 앞으로 당기는 편이 맞을 수도 있다. 정답은 도메인이 정한다. 다만 정하지 않은 채로 두면 라이브러리가 우리 대신 아무 쪽이나 정해 버린다.

두 번 오는 시각은 더 성가시다. 그 한 시간 동안 생성된 로그를 벽시계 기준으로 정렬하면 순서가 뒤엉킨다. 이 구간을 다루는 코드는 반드시 순간 기준으로 정렬해야 하고, 화면에 보여 줄 때만 벽시계로 바꾼다. 1장에서 말한 경계 규칙이 여기서 값을 한다.

2.3 하루의 경계

시간대가 흔드는 것은 시각만이 아니다. 날짜도 흔든다. 오늘이 언제 시작해서 언제 끝나는가는 지역마다 다르고, 그 차이는 집계 결과를 통째로 바꾼다.

일간 활성 사용자 수를 센다고 하자. UTC 기준 자정으로 자르면 한국 사용자에게는 오전 아홉 시가 하루의 경계가 된다. 아침 출근길 사용이 전날에 포함되고, 늦은 밤 사용이 다음 날로 넘어간다. 숫자가 틀린 건 아니지만 사람들이 생각하는 하루와 다르다. 마케팅 팀이 캠페인 효과를 보려고 이 지표를 보면, 캠페인 시작일의 성과가 이상하게 낮게 나온다.

해법은 집계 기준 시간대를 명시적 설정으로 올리는 것이다. 지표 정의 문서에 어느 시간대의 자정을 경계로 삼는지 적고, 쿼리에서 그 값을 파라미터로 받는다. 여러 나라 데이터를 한 표에 올릴 거라면, 각 행이 어느 기준으로 잘렸는지 열을 하나 더 두는 편이 나중에 논쟁을 줄인다.

여기 얽힌 함정이 하나 더 있다. 일광절약이 있는 지역에서는 하루가 23시간이거나 25시간이다. 하루를 86400초로 가정하고 구간을 만드는 코드는 그날 한 시간을 빠뜨리거나 겹쳐 센다. 구간은 자정에서 다음 자정까지로 정의해야지, 자정에 86400초를 더해 정의하면 안 된다.

2.4 시간대 규칙도 배포 대상이다

시간대 규칙 묶음은 운영체제나 언어 런타임 안에 파일로 들어 있다. 각국이 규칙을 바꾸면 이 파일이 갱신되고, 갱신본이 우리 컨테이너 이미지에

들어와야 우리 서비스가 올바른 계산을 한다.

여기서 자주 나오는 사고가 있다. 컨테이너 베이스 이미지를 오래 고정해 두면 그 안의 규칙 파일도 함께 늙는다. 어느 나라가 일광절약을 폐지했는데 우리 서비스만 여전히 시행하는 것처럼 계산한다. 코드도 바뀐 게 없고 배포도 안 했으니 아무도 의심하지 않는다. 사용자 문의가 들어오고 나서야 원인에 도달하는데, 그때쯤이면 잘못된 알림이 이미 여러 차례 나간 뒤다.

이 문제는 관리 항목으로 올려야 해결된다. 규칙 파일 버전을 애플리케이션 기동 로그에 남기고, 모니터링 대시보드에 노출한다. 언어 런타임과 운영체제가 서로 다른 버전을 들고 있을 수도 있으니 둘 다 찍는다. 그리고 규칙이 갱신되면 예약 데이터 중 영향받는 것이 있는지 훑어보는 점검 절차를 문서로 남긴다. 자주 쓸 일은 없지만 필요할 때 없으면 손이 굳는다.

2.5 이 장의 점검 목록

배포 전에 짚어 볼 것들을 짧게 모아 둔다.

- 예약 데이터에 지역 식별자가 들어 있는가, 아니면 오프셋만 있는가
- 없는 시각과 겹치는 시각에 대한 처리 정책이 코드에 적혀 있는가
- 일간 집계의 기준 시간대가 설정으로 노출되어 있는가
- 하루를 86400초로 가정하는 계산이 남아 있는가
- 시간대 규칙 파일의 버전을 기동 시점에 확인할 수 있는가

다섯 줄뿐이지만, 이 다섯 줄을 통과하지 못하는 시스템이 생각보다 많다. 다음 장에서는 시간대와 무관하게 발생하는 또 다른 층위의 문제, 시계 자체를 믿을 수 없는 상황을 다룬다.

2.6 사용자에게 무엇을 보여 줄 것인가

기술적 처리를 다 맞춰 놓고도 화면에서 신뢰를 잃는 경우가 있다. 시각을 어느 기준으로 보여 줄지 일관성이 없을 때다.

선택지는 셋이다. 보는 사람의 기기 시간대로 보여 주거나, 계정에 저장된 시간대로 보여 주거나, 그 사건이 일어난 지역의 시간대로 보여 주는 것이다. 어느 쪽도 항상 옳지 않다. 출장 중인 사람에게 기기 기준으로 회의 시각을 보여 주면 혼란스럽고, 반대로 항공권 시각을 계정 기준으로 환산해 보여 주면 더 혼란스럽다. 항공권은 출발지와 도착지의 현지 시각으로 적혀야 사람이 계획을 세울 수 있다.

그래서 화면 요소마다 어떤 기준을 쓰는지 정해 두고, 애매한 자리에는 기준을 함께 적는다. 오후 세 시라고만 쓰지 말고 어느 지역 기준인지 짧게 덧붙이는 것만으로 문의가 크게 준다. 특히 서로 다른 지역의 사람이 함께 보는 화면, 이를테면 협업 도구의 일정이나 운영 대시보드에서는 기준 표기가 사실상 필수다.

상대 시각 표기도 조심스럽게 쓴다. 3분 전 같은 표현은 읽기 편하지만 정밀도가 필요한 자리에서는 위험하다. 장애 대응 중에 로그를 보는 사람에게는 몇 분 전인지보다 정확히 몇 시 몇 분인지가 중요하다. 상대 표기는 보조로 두고, 정확한 값은 손이 닿는 곳에 남겨 둔다.

2.7 마이그레이션할 때 벌어지는 일

이미 오프셋으로 저장해 버린 시스템을 고치는 일도 만나게 된다. 이때 흔히 저지르는 실수는 전체 데이터를 한 번에 변환하려는 시도다.

과거 기록은 대체로 그대로 두어도 된다. 이미 일어난 사실이고, 오프셋이 붙어 있다면 순간으로 해석하는 데 문제가 없다. 손봐야 하는 것은 미래를 가리키는 데이터, 즉 예약과 반복 일정이다. 이 데이터는 사용자가 어떤

의도로 그 시각을 골랐는지 추정해야 하는데, 저장된 값만으로는 알 수 없는 경우가 많다.

그래서 마이그레이션은 두 단계로 나눈다. 먼저 새로 들어오는 데이터부터 지역 식별자를 받도록 입력 경로를 바꾼다. 그다음 기존 미래 데이터에 대해서는 계정의 기본 지역을 추정값으로 채우되, 그 값이 추정이라는 사실을 별도 열에 표시한다. 다음번에 사용자가 그 항목을 열어 볼 때 확인을 받고 확정으로 바꾼다. 조용히 추정값을 확정처럼 쓰기 시작하면, 나중에 어떤 값이 사실이고 어떤 값이 짐작인지 아무도 구분할 수 없게 된다.

3 순서와 드리프트, 시계를 믿을 수 없을 때

앞의 두 장은 사람의 달력이 만든 문제였다. 이 장의 문제는 기계가 만든다. 컴퓨터의 시계는 우리가 생각하는 것만큼 정확하지도, 단조롭게 흐르지도 않는다. 그리고 여러 대의 컴퓨터가 각자의 시계를 들고 있을 때, 그 시계들이 서로 맞다는 보장은 어디에도 없다.

3.1 시스템 시계와 단조 시계

운영체제는 보통 두 종류의 시계를 제공한다. 하나는 지금이 몇 시인지를 알려 주는 시스템 시계다. 이 값은 사람이 읽을 수 있고 기록에 적합하지만, 외부 시각 동기화 결과에 따라 앞뒤로 조정된다. 다른 하나는 어떤 기준점부터 얼마나 흘렀는지만 알려 주는 단조 시계다. 이 값은 절대 뒤로 가지 않지만 몇 시인지는 알려 주지 않는다.

용도는 명확히 갈린다. 두 시점 사이의 경과 시간을 재려면 단조 시계를 쓴다. 타임아웃, 지연 측정, 재시도 간격, 속도 제한 창처럼 흐른 양이 중요한 모든 계산이 여기 해당한다. 반대로 언제 일어난 일인지를 기록할 때는 시스템 시계를 쓴다. 로그, 감사 기록, 생성 시각이 그렇다.

이 구분을 지키지 않으면 증상이 요란하다. 시각 동기화가 시계를 1초 뒤로 돌린 순간, 시스템 시계로 재던 타임아웃은 1초를 더 기다리거나 즉시 만료된다. 서버 시각이 크게 틀어져 있다가 한 번에 교정되는 경우에는 모든 연결이 동시에 만료되거나, 반대로 만료되어야 할 것들이 몇 분간 살아남는다. 장애 보고서에는 갑자기 커넥션이 몰렸다고만 적히고, 진짜 원인은 몇 주 뒤에나 밝혀진다.

언어나 프레임워크가 제공하는 기본 함수가 어느 쪽인지 확인해 두는 것이 좋다. 이름만 봐서는 구분되지 않는 경우가 흔하고, 잘못 골라도 평소에는

아무 일도 일어나지 않는다.

3.2 타임스탬프로 정렬하지 마라

분산 시스템에서 가장 비싼 착각은 타임스탬프가 순서를 알려 준다는 믿음이다. 두 서버가 찍은 시각을 비교해 어느 쪽이 먼저인지 판단하는 코드는, 두 서버의 시계가 정확히 맞다는 전제 위에 서 있다. 그 전제는 성립하지 않는다.

일반적인 환경에서 서버 간 시계 오차는 수십 밀리초 수준이고, 동기화가 잘못되면 훨씬 커진다. 가상화된 환경에서는 순간적으로 더 벌어지기도 한다. 그러니 밀리초 단위로 승패가 갈리는 판단, 예를 들어 마지막 쓰기가 이긴다는 규칙을 시각 비교로 구현하면 데이터가 조용히 사라진다. 나중에 도착한 갱신이 더 이른 시각을 달고 있다는 이유로 버려지는 것이다.

같은 서버 안에서도 안심할 수 없다. 시계 해상도가 충분하지 않으면 서로 다른 두 사건이 정확히 같은 값을 받는다. 동률이 나오면 정렬 결과는 구현에 따라 달라지고, 같은 쿼리를 두 번 돌렸을 때 순서가 바뀐다. 페이지 단위로 조회하는 화면에서는 이 흔들림이 항목 누락이나 중복으로 나타난다.

해결의 방향은 시계를 더 정확하게 만드는 쪽이 아니다. 순서를 표현할 다른 수단을 두는 쪽이다.

- 단일 지점에서 발급하는 증가 시퀀스
- 시각 상위 비트와 난수 하위 비트를 결합한 식별자
- 갱신마다 올라가는 버전 번호
- 인과 관계를 담는 벡터 형태의 카운터

무엇을 고르든 공통 원칙은 같다. 정렬 키는 항상 유일해야 한다. 시각으로 정렬해야만 하는 상황이라면 시각 뒤에 식별자를 붙여 동률을 깨고, 그 조합을 페이지 커서로 쓴다. 그러면 적어도 같은 쿼리가 같은 순서를 돌려준다.

3.3 이벤트 시간과 처리 시간

데이터 파이프라인으로 넘어오면 시간이 두 개로 갈라진다. 사건이 실제로 일어난 시각과 우리 시스템이 그것을 받은 시각이다. 앞의 것을 이벤트 시간, 뒤의 것을 처리 시간이라 부른다.

두 값은 대체로 다르다. 모바일 기기는 오프라인 상태에서 기록을 모아 두었다가 연결되면 한꺼번에 올린다. 중간 큐에 밀리면 몇 분씩 늦어진다. 재처리 배치를 돌리면 며칠 전 사건이 지금 도착한다. 처리 시간으로 집계하면 이 모든 지연이 오늘 숫자에 섞이고, 이벤트 시간으로 집계하면 이미 마감한 어제 숫자가 계속 움직인다.

그래서 필요한 것이 마감선이다. 어느 시점까지 도착한 데이터로 그 구간을 확정할지 정하고, 그 이후 도착분을 어떻게 처리할지도 미리 정한다. 선택지는 셋이다. 버리거나, 별도 보정 구간에 담거나, 확정을 미루고 계속 갱신하거나.

방식	대가
지각분 폐기	총량 과소 집계
보정 구간 적재	조회 로직 복잡
계속 갱신	숫자가 안 굳음

정답은 도메인이 정한다. 청구서는 어느 시점에 반드시 굳어야 하고, 운영 대시보드는 계속 갱신되는 편이 유용하다. 중요한 것은 선택을 명시하고 그 사실을 소비자에게 알리는 일이다. 지표 설명에 이 구간은 사흘간 갱신될 수 있다고 한 줄 적어 두면, 어제와 오늘 숫자가 다르다는 문의가 절반으로 준다.

3.4 지연의 꼬리를 지표로 만든다

이벤트 시간과 처리 시간의 차이는 그 자체로 훌륭한 관측 지표다. 이 값의 분포를 계속 기록해 두면 파이프라인이 막히기 시작하는 순간을 조기에 감지할 수 있다.

평균만 보면 안 된다. 지연은 대개 긴 꼬리를 갖고, 평균은 그 꼬리를 감춘다. 상위 백분위 값을 함께 본다. 그리고 마감선을 정할 때 이 실측 분포를 근거로 삼는다. 감으로 정한 마감선은 대개 너무 짧아서 데이터를 흘리거나, 너무 길어서 숫자가 굳지 않는다.

여기서 조심할 점 하나. 이벤트 시간을 클라이언트가 보내 준 값으로 쓴다면, 그 값은 사용자 기기의 시계다. 기기 시계는 임의로 설정될 수 있고, 실제로 몇 년씩 어긋난 기기가 존재한다. 그러니 클라이언트가 보낸 시각은 그대로 믿지 말고, 서버 수신 시각과 함께 저장한 뒤 둘의 차이가 비현실적으로 크면 이상값으로 표시한다. 아예 버리기보다는 표시해 두는 편이 낫다. 나중에 원인을 추적할 실마리가 남는다.

3.5 시각 동기화를 운영 항목으로 올린다

시계가 틀어지는 것은 소프트웨어 결함이 아니라 하드웨어의 성질이다. 수정 발진기는 온도와 전압에 따라 미세하게 빨라지거나 느려지고, 그 오차가 하루에 몇 초씩 쌓인다. 그래서 모든 서버는 주기적으로 외부 기준과 맞춘다.

문제는 이 동기화가 조용히 실패한다는 점이다. 방화벽 규칙이 바뀌어 동기화 트래픽이 막히거나, 기준 서버 주소가 응답하지 않거나, 컨테이너 안에서 동기화 데몬이 아예 돌지 않는 경우가 있다. 애플리케이션은 아무 오류도 보지 못한 채 점점 어긋난 시각을 찍는다.

그러니 시계 오차를 지표로 수집한다. 동기화 상태와 추정 오차를

주기적으로 읽어 대시보드에 올리고, 임계를 넘으면 알림을 보낸다. 이 지표가 없으면 시계 문제는 언제나 사후에 발견되고, 사후에는 이미 잘못된 데이터가 쌓인 뒤다.

가상화 환경에서는 한 가지가 더 있다. 가상 머신이 멈췄다 재개되면 그 사이의 시간이 통째로 사라진다. 단조 시계마저 정지 구간을 어떻게 다룰지 플랫폼마다 다르게 처리한다. 긴 정지 뒤에 깨어난 프로세스가 밀린 작업을 한꺼번에 실행하려 들면서 부하가 몰리는 사고가 여기서 나온다. 재개 직후에는 밀린 일을 한꺼번에 쏟지 말고 나눠서 흘려보내는 편이 안전하다.

3.6 윤초와 갑작스러운 조정

세계 표준시에는 지구 자전 속도를 맞추기 위해 가끔 1초가 삽입된다. 이 1초 때문에 큰 규모의 장애가 여러 차례 있었다. 존재하지 않아야 할 초 값이 등장하면서 시간 계산 코드가 예외를 던지거나 무한 반복에 빠진 사례들이다.

요즘 큰 사업자들은 이 1초를 하루에 걸쳐 아주 조금씩 나눠 흡수하는 방식을 쓴다. 시계가 잠깐 아주 느리게 흐를 뿐 건너뛰거나 되돌아가지 않으므로, 애플리케이션 입장에서는 아무 일도 일어나지 않는다. 다만 같은 조직 안에서 어떤 서버는 흡수 방식을 쓰고 어떤 서버는 삽입 방식을 쓰면 그날 하루 두 그룹의 시각이 최대 1초까지 벌어진다. 시각으로 순서를 판단하지 말라는 앞의 조언이 여기서 또 한 번 값을 한다.

3.7 이 장에서 가져갈 것

시계는 측정 도구이지 진실의 원천이 아니다. 흐른 양을 재는 데는 단조 시계를 쓰고, 순서를 정하는 데는 시계 대신 시퀀스나 버전을 쓰며, 사건 시각과 도착 시각은 따로 기록한다. 이 세 가지만 지켜도 분산 환경에서 시간이 일으키는 사고 대부분은 범위 밖으로 밀려난다.

다음 장은 이 시계 위에서 돌아가는 반복 작업으로 간다. 스케줄러와 재시도, 그리고 만료다.

4 스케줄, 재시도, 만료가 겹치는 자리

정기 배치는 대체로 조용히 잘 돈다. 그러다 어느 날 두 번 돌거나, 한 번도 안 돌거나, 앞의 실행이 끝나기 전에 다음 실행이 시작된다. 이 셋은 서로 다른 사고처럼 보이지만 원인은 하나로 모인다. 스케줄러를 실행 횟수로 이해했기 때문이다.

4.1 실행이 아니라 구간을 처리한다

매시 정각에 도는 작업이 있다고 하자. 이 작업이 하는 일은 무엇인가. 대개는 지난 한 시간 동안 쌓인 데이터를 처리하는 것이다. 그렇다면 이 작업의 본질은 정각에 깨어나는 것이 아니라, 특정 한 시간 구간을 소비하는 것이다.

이 관점 전환이 설계를 바꾼다. 작업에 넘기는 인자를 실행 시각이 아니라 처리 구간의 시작과 끝으로 잡는다. 그러면 어제 오후 세 시 구간을 다시 처리하고 싶을 때 같은 코드를 인자만 바꿔 호출하면 된다. 실행 시각에 의존하는 코드는 이 재실행이 불가능하고, 결국 별도의 백필 스크립트가 하나 더 생긴다. 그 스크립트는 본체와 조금씩 달라지다가 언젠가 다른 결과를 낸다.

구간을 명시하면 누락도 드러난다. 처리한 구간을 기록해 두면 어느 구간이 비었는지 질의 한 번으로 알 수 있다. 스케줄러가 몇 시간 멈춰 있었다는 사실을, 사용자 문의가 아니라 우리 쪽 점검으로 먼저 발견하게 된다.

경계 규칙도 함께 못 박는다. 구간의 시작은 포함하고 끝은 제외한다. 이 관례를 어기면 인접한 두 구간이 한 시점을 두 번 세거나, 그 시점이 어느 쪽에도 안 들어간다. 사소해 보이지만 정산에서는 이 한 건이 대조 실패로 이어진다.

4.2 구간 식별자를 멱등성 키로

분산 스케줄러는 언젠가 같은 작업을 두 번 띄운다. 리더 선출이 흔들리거나, 배포 중에 새 인스턴스와 옛 인스턴스가 겹치거나, 네트워크가 갈라져 양쪽이 서로를 죽었다고 판단하는 경우다. 완벽하게 한 번만 실행되게 만드는 일은 대체로 비용이 크고, 그 비용을 치러도 완벽하지 않다.

방향을 바꾸는 편이 낫다. 두 번 실행되어도 결과가 같게 만든다. 이때 필요한 것이 멱등성 키인데, 앞서 정의한 구간 식별자가 그대로 좋은 키가 된다. 작업 이름과 구간 시작 시각을 합친 문자열을 유일 제약이 걸린 테이블에 먼저 삽입하고, 삽입에 실패하면 이미 누가 하고 있다는 뜻이니 조용히 물러난다.

키를 만들 때 실행 시각을 섞으면 안 된다. 그러면 두 실행이 서로 다른 키를 갖게 되어 중복이 걸러지지 않는다. 키는 무엇을 처리하는가로만 구성한다. 언제 처리하는가는 키의 일부가 아니다.

결과를 쓰는 쪽도 같은 원칙으로 만든다. 누적 증가 대신 구간별 값을 덮어쓰는 형태로 저장하면 재실행이 안전해진다. 증분만 더하는 구조는 한 번 더 돌 때마다 숫자가 부풀고, 어디까지가 정상인지 되짚기 어렵다.

4.3 겹침을 어떻게 처리할 것인가

앞선 실행이 예상보다 오래 걸려 다음 실행 시각이 되었을 때 무엇을 할지도 정책이다. 스케줄러의 기본값에 맡기지 말고 작업마다 정한다.

정책	적합한 작업
건너뛰기	최신 상태 동기화
큐에 쌓기	순서가 중요한 처리
병렬 허용	구간이 독립인 집계

최신 상태를 맞추는 작업이라면 밀린 실행을 건너뛰어도 손해가 없다. 다음 회차가 어차피 최신을 반영한다. 반대로 구간별 정산처럼 모든 구간이 정확히 한 번 처리되어야 하는 일은 밀려도 쌓아 두었다가 차례로 소화해야 한다. 이때 쌓인 양이 한계를 넘으면 알림을 보내야 한다. 조용히 쌓이다가 어느 순간 며칠치가 밀려 있는 상태를 발견하는 것이 최악이다.

일광절약을 쓰는 지역의 시간대로 스케줄을 걸었다면 봄과 가을에 각각 문제가 생긴다. 새벽 두 시 반에 도는 작업은 봄에 하루 걸러지고, 가을에는 두 번 돈다. 두 번 도는 쪽은 멱등성 키가 막아 준다. 걸러지는 쪽은 막아 줄 장치가 없으므로, 놓치면 안 되는 작업은 아예 UTC로 걸거나 새벽 두 시대를 피해서 건다.

4.4 재시도에는 마감이 필요하다

일시적 실패를 다시 시도하는 것은 당연한 습관이다. 문제는 재시도가 겹칠 때 생긴다. 어떤 요청이 서비스 세 계층을 통과하는데 각 계층이 세 번씩 재시도한다면, 맨 아래 서비스는 최대 스물일곱 번의 요청을 받는다. 부하가 걸려 느려진 서비스에 재시도가 쏟아지면 회복할 틈이 사라진다.

그래서 간격을 점점 늘리고, 그 간격에 무작위 흔들림을 섞는다. 흔들림이 없으면 동시에 실패한 클라이언트들이 동시에 다시 몰려온다. 여기에 더해 재시도 총량이 아니라 마감 시각을 기준으로 멈추는 규칙까지 세 가지를 함께 설계한다.

마감 시각을 기준으로 멈추는 규칙이 핵심인데 자주 빠진다. 요청이 들어올 때 이 요청은 언제까지 유효한가를 정하고, 그 마감 시각을 하위 호출로 계속 전달한다. 각 계층은 남은 시간을 보고 자기 타임아웃을 정하며, 남은 시간이 이미 없으면 시도조차 하지 않는다. 마감이 전파되지 않으면 상위에서는 이미 포기하고 오류를 돌려준 요청을 하위에서는 계속 붙들고 작업한다. 사용자는 실패를 봤는데 시스템 자원은 계속 타는 상태다.

재시도해도 되는 실패와 그렇지 않은 실패를 구분하는 일도 잊기 쉽다. 연결 거부나 일시적 과부하는 다시 시도할 만하다. 잘못된 입력은 몇 번을 보내도 같은 결과다. 후자를 재시도하는 코드는 부하만 만든다.

4.5 만료와 리스에는 여유가 필요하다

토큰 만료, 분산 잠금, 캐시 수명은 모두 시간에 기대는 장치다. 그리고 서로 다른 기계의 시계 위에서 판단된다는 공통점이 있다.

만료 시각을 발급자가 정하고 검증자가 확인하는 구조라면, 두 기계의 시계 차이만큼 오차가 생긴다. 검증 쪽 시계가 조금 빠르면 아직 유효한 토큰이 거부된다. 그래서 검증에는 짧은 허용 오차를 둔다. 몇 초 정도의 여유가 로그인 실패 문의를 크게 줄인다. 다만 이 여유는 짧아야 한다. 길게 잡으면 만료의 의미가 흐려진다.

분산 잠금은 더 조심스럽다. 잠금을 쥔 프로세스가 멈춰 있는 사이 리스가 만료되면 다른 프로세스가 잠금을 얻는다. 그런데 첫 프로세스가 다시 깨어나 자기가 아직 잠금을 쥐고 있다고 믿고 쓰기를 시도할 수 있다. 시간만으로는 이 상황을 막지 못한다. 잠금을 얻을 때마다 증가하는 번호를 함께 발급하고, 저장소가 더 낮은 번호의 쓰기를 거절하게 만들어야 한다. 시간은 잠금을 회수하는 데 쓰고, 안전은 번호가 지킨다.

캐시 수명에는 다른 함정이 있다. 같은 시각에 대량으로 채워진 항목은 같은 시각에 한꺼번에 만료된다. 그 순간 뒷단으로 요청이 몰린다. 수명에 무작위 폭을 섞어 만료를 흩뜨리고, 만료 직전에 미리 갱신하는 경로를 두면 이 봉우리가 사라진다.

4.6 첫 실행과 마지막 실행

주기 작업에는 시작과 끝이 있는데 설계에서 자주 빠진다.

새 작업을 배포한 직후를 생각해 보자. 처리 기록이 비어 있으니 어느 구간부터 시작할지 정해야 한다. 아무 규칙이 없으면 코드가 데이터의 맨 처음부터 훑기 시작하고, 운이 나쁘면 몇 년치를 한 번에 밀어 넣는다. 배포 직후 새벽에 데이터베이스가 비명을 지르는 사고의 흔한 원인이다. 그러니 시작 구간을 설정으로 명시하고, 한 번에 처리할 구간 수에 상한을 둔다.

작업을 내릴 때도 마찬가지다. 스케줄만 지우고 처리 기록을 남겨 두면, 나중에 누군가 되살렸을 때 그 사이의 빈 구간을 몰아서 처리하려 든다. 작업을 중단할 때는 재개 시 어디부터 시작할지도 함께 정해 기록에 남긴다.

4.7 시간 기반 정리 작업

오래된 데이터를 지우는 작업은 시간 조건 위에서 돌기 때문에 이 장의 함정을 모두 물려받는다. 게다가 되돌릴 수 없다.

기준을 상대 표현으로만 두면 위험하다. 90일보다 오래된 것을 지운다는 조건은, 어떤 이유로 기준 시각이 미래로 튀는 순간 전부를 지운다. 시계가 잘못 설정된 컨테이너 하나가 이런 사고를 낸다. 방어는 단순하다. 한 번의 실행에서 지울 수 있는 최대 건수를 정하고, 그 상한에 걸리면 지우지 말고 멈춰서 알린다. 정상 상황에서 상한에 걸릴 일은 없으므로, 걸렸다는 사실 자체가 이상 신호다.

지우기 전에 표시만 하는 단계를 하나 두는 것도 값싼 보험이다. 대상을 먼저 표시하고 하루 뒤에 실제로 지우면, 잘못된 조건으로 표시된 경우를 되돌릴 시간이 생긴다.

4.8 이 장에서 가져갈 것

스케줄러는 시계를 읽는 장치가 아니라 구간을 배분하는 장치다. 구간으로 생각하면 재실행과 백필이 자연스러워지고, 구간 식별자를 키로 삼으면 중복 실행이 사고에서 무해한 일로 내려온다. 재시도에는 마감을 전파하고,

만료에는 여유와 번호를 함께 둔다. 다음 장은 이 모든 것이 이미 지나간 뒤, 과거를 다시 계산해야 할 때의 이야기다.

5 과거를 다시 계산하기와 시간을 테스트하는 법

시간을 다루는 시스템은 결국 같은 요청을 받는다. 지난달 정산을 다시 뽑아 달라, 이 고객의 요금을 그때 기준으로 계산해 달라, 지표 정의가 바뀌었으니 반년치를 다시 채워 달라. 이 요청을 처리할 수 있는가 없는가가 시스템의 성숙도를 가른다.

5.1 두 개의 시간축

과거를 다시 계산하려면 먼저 무엇을 기록해 두었는지가 중요하다. 여기서 필요한 개념이 두 개의 시간축이다.

하나는 사실이 성립한 시각이다. 계약이 발효된 날, 요금제가 적용되기 시작한 날, 직원이 부서를 옮긴 날이 여기 속한다. 다른 하나는 시스템이 그 사실을 알게 된 시각이다. 담당자가 화면에서 입력한 시각, 외부 시스템에서 데이터가 들어온 시각이다.

이 둘이 다를 수 있다는 사실이 핵심이다. 3월 1일자 부서 이동이 3월 15일에 입력되는 일은 흔하다. 만약 우리가 시스템 입력 시각만 기록했다면, 3월 10일 시점의 조직도를 다시 그릴 방법이 없다. 반대로 사실 시각만 기록했다면, 3월 10일에 우리가 무엇을 알고 있었는지를 재현할 수 없다. 감사 대응이나 분쟁 상황에서 필요한 것은 대개 후자다. 그때 우리가 가진 정보로 그 판단이 타당했는가를 묻기 때문이다.

두 축을 모두 기록하면 질문이 네 갈래로 나뉜다. 지금 아는 대로의 현재, 지금 아는 대로의 과거, 그때 알던 대로의 그때, 그때 알던 대로의 미래 예측이다. 모든 시스템이 네 가지를 다 지원할 필요는 없다. 다만 어느

것이 필요한지 결정하지 않은 채 설계하면, 나중에 필요해졌을 때 데이터가 이미 없다.

구현은 생각보다 단순하다. 이력이 중요한 테이블에서 갱신을 덮어쓰기가 아니라 새 행 추가로 바꾸고, 각 행에 유효 시작과 유효 종료, 기록 시각을 붙인다. 조회는 시점을 인자로 받는 함수 하나로 감싼다. 이 감싸는 층이 없으면 여기저기서 직접 조건을 적게 되고, 그중 하나는 반드시 틀린다.

5.2 그때의 규칙으로 계산하기

데이터를 시점별로 조회할 수 있게 되어도 계산이 남는다. 요율, 환율, 세율, 할인 정책, 등급 기준은 모두 시간에 따라 바뀐다. 지난달 청구서를 다시 뽑는데 이번 달 요율을 쓰면 숫자가 달라진다.

그래서 정책값도 데이터로 다룬다. 코드에 상수로 박거나 설정 파일 한 줄로 두면 과거 값이 남지 않는다. 대신 정책 테이블에 유효 구간을 붙여 저장하고, 계산 함수는 반드시 기준 시점을 인자로 받아 그 시점에 유효한 값을 찾아 쓰게 한다.

정책이 바뀌는 순간의 처리도 미리 정한다. 월 중간에 요율이 바뀌면 그 달을 어떻게 나눌지, 경계에 걸친 건은 어느 쪽 규칙을 적용할지 같은 것들이다. 이런 규칙은 대개 계약서나 약관에 적혀 있고, 코드에는 안 적혀 있다. 옮겨 적는 일을 미루면 몇 달 뒤에 개발자가 추측으로 구현하게 된다.

로직 자체가 바뀔 때는 더 조심스럽다. 계산식이 개선되었다고 과거 전부를 새 식으로 다시 계산하면, 이미 고객에게 나간 청구서와 숫자가 달라진다. 이럴 때는 계산 로직에 버전을 붙이고, 각 결과에 어느 버전으로 계산했는지 기록한다. 재계산이 필요하면 어느 버전으로 돌릴지 명시적으로 고른다.

5.3 백필을 안전하게 만드는 조건

반년치를 다시 채우는 작업은 위험하다. 오래 걸리고, 중간에 실패하며, 실패한 지점을 알기 어렵고, 돌아가는 동안 서비스 부하를 만든다. 안전한 백필에는 몇 가지 조건이 필요하다.

- 구간 단위로 쪼개고 각 구간이 독립적으로 성공하거나 실패한다
- 몇 번을 돌려도 결과가 같다
- 진행 상황을 조회할 수 있고 중단 후 재개할 수 있다
- 속도를 조절할 수 있어 운영 부하를 눌러 놓을 수 있다

앞 장에서 스케줄러를 구간 단위로 설계하라고 한 이유가 여기서 드러난다. 그 설계를 해 두었다면 백필은 별도 코드가 아니라 같은 코드에 다른 구간 목록을 넣는 일이 된다. 유지보수 대상이 하나로 줄고, 정기 실행에서 검증된 로직이 백필에도 그대로 적용된다.

결과를 어디에 쓸지도 결정해야 한다. 기존 값을 바로 덮어쓰면 도중에 문제가 생겼을 때 원래 값이 사라진다. 별도 영역에 채운 뒤 검증하고 교체하는 방식이 안전하다. 교체 전에 두 결과의 차이를 요약해 보는 단계를 두면 예상과 다른 변화를 배포 전에 발견할 수 있다. 차이가 0인 것이 정상인 경우도 많은데, 그럴 때 0이 아니면 로직이 의도치 않게 바뀐 것이다.

5.4 시계를 주입 가능한 것으로

이제 테스트다. 시간 관련 코드가 테스트되지 않는 가장 큰 이유는 시계를 직접 호출하기 때문이다. 함수 안에서 현재 시각을 읽으면 그 함수는 실행하는 시점에 따라 다르게 동작하고, 테스트는 특정 시점을 재현할 수 없다.

해법은 오래된 것이다. 현재 시각을 반환하는 일을 하나의 의존성으로 뽑아

밖에서 넣어 준다. 운영에서는 진짜 시계를 넣고, 테스트에서는 원하는 값을 돌려주는 가짜 시계를 넣는다. 이렇게 하면 일광절약 전환의 그 한 시간도, 윤년의 2월 29일도, 연말의 마지막 초도 테스트에서 재현할 수 있다.

여기서 규칙을 하나 세운다. 시계는 도메인 로직 안쪽으로 들어가지 않는다. 계산 함수는 기준 시각을 인자로 받고, 그 인자를 채우는 일은 바깥 계층이 한다. 그러면 도메인 로직 전체가 순수한 함수가 되어 어떤 시점이든 자유롭게 시험할 수 있다.

5.5 무엇을 테스트할 것인가

시간 테스트는 무한히 늘어날 수 있으므로 목록을 정해 두는 편이 낫다. 실제로 사고를 냈던 날짜들을 모아 고정 목록으로 만들고 회귀 테스트로 돌린다.

구분	대표 사례
달력 경계	윤년 2월 29일
월말	31일과 30일
전환일	일광절약 시작과 끝
연말	연도가 넘어가는 자정

여기에 지역을 곱한다. 일광절약이 있는 지역, 없는 지역, 오프셋이 30분 단위인 지역을 각각 하나씩 골라 두면 대부분의 경우가 걸린다.

무작위 입력을 던져 성질을 검증하는 방법도 잘 맞는다. 임의의 시각을 만들어 지역 시각으로 바꿨다가 되돌렸을 때 원래 값이 나오는지, 구간을 쪼갠 뒤 합쳤을 때 원래 구간과 같은지, 정렬 결과가 두 번 실행에서 동일한지 같은 성질을 확인한다. 사람이 떠올리지 못한 날짜를 기계가 찾아 준다.

5.6 운영 중에 확인하는 방법

테스트를 통과했다고 끝은 아니다. 시간 관련 결함은 특정 날짜가 되어야 나타나므로, 배포 시점에는 아무 신호도 없다가 몇 달 뒤에 터진다. 그래서 운영 중에 확인할 장치를 몇 개 심어 둔다.

가장 값싼 것은 미래 점검이다. 정기적으로 시계를 앞으로 옮긴 상태로 핵심 계산을 실행해 보는 작업을 하나 만든다. 실제 시계를 건드리는 것이 아니라, 앞서 주입 가능하게 만든 시계에 미래 값을 넣고 결과가 정상 범위인지 확인하는 것이다. 다음 일광절약 전환일, 다음 윤년, 다음 연말을 미리 통과시켜 보면 그날이 오기 전에 문제를 안다.

다음은 대조다. 서로 다른 경로로 계산된 같은 숫자를 주기적으로 맞춰 본다. 실시간 집계와 배치 집계, 서비스 화면과 정산 리포트처럼 짝이 있는 값들이 대상이다. 두 값이 벌어지기 시작하면 대개 시간 경계 처리가 어긋난 것이고, 이 신호는 사용자 문의보다 훨씬 빨리 온다.

마지막은 기록이다. 계산 결과에 어떤 기준 시각과 어떤 시간대, 어떤 정책 버전을 썼는지 함께 남긴다. 이 세 줄이 있으면 나중에 숫자가 왜 그랬는지 되짚는 데 몇 분이면 충분하다. 없으면 며칠이 걸리고, 그마저도 추측으로 끝나는 경우가 많다.

5.7 마지막으로

시간을 다루는 일에 우아한 해법은 없다. 규칙이 정치와 관습에서 나오고 예외가 계속 추가되기 때문이다. 우리가 할 수 있는 것은 그 복잡함을 코드 곳곳으로 흩뿌리지 않고 몇 개의 좁은 자리에 모아 두는 것뿐이다.

세 타입을 구분하고, 지역 이름으로 저장하고, 측정에는 단조 시계를 쓰고, 스케줄을 구간으로 생각하고, 정책에 유효 기간을 붙이고, 시계를 주입 가능하게 만든다. 여섯 개의 습관이고 어느 것도 어렵지 않다. 어려운 것은

이 습관들이 없는 코드베이스에서 사고가 난 뒤에 되돌리는 일이다.

시간 버그는 대체로 조용하다는 말로 이 책을 시작했다. 조용한 버그를 막는 유일한 방법은 시끄러워지기 전에 규칙을 정해 두는 것뿐이다.