2. 공통 코드 설계

#0.강의/2.데이터베이스로드맵/4.설계2

공통 코드가 필요한 이유

데이터베이스를 설계하다 보면 비슷한 종류의 데이터가 여러 테이블에 걸쳐 반복되는 상황을 자주 만난다. 예를 들어 주문 상태, 회원 등급, 결제 방법 같은 것들이다. 이런 데이터를 어떻게 관리해야 할까? 먼저 문제 상황부터 살펴보자.

실습 준비

```
-- 데이터베이스가 존재하지 않으면 생성
CREATE DATABASE IF NOT EXISTS my_shop4;
USE my_shop4;
```

- 실습에서는 `my_shop4` 데이터베이스를 사용하겠다.

문제 상황 - 하드코딩된 상태값

쇼핑몰을 운영한다고 가정하자. 주문 테이블에는 주문 상태가 필요하다. 가장 단순한 방법은 상태값을 문자열로 직접 저장하는 것이다.

```
DROP TABLE IF EXISTS orders;
```

```sql
CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    member_id BIGINT NOT NULL,
    order_status VARCHAR(20) NOT NULL,
    total_amount INT NOT NULL,
    created_at DATETIME NOT NULL
);
```

주문 데이터를 입력해보자.

```sql
INSERT INTO orders (member_id, order_status, total_amount, created_at) VALUES
(1, '주문완료', 50000, '2026-01-15 10:30:00'),
(2, '결제완료', 75000, '2026-01-15 11:00:00'),
(3, '배송중', 30000, '2026-01-15 12:00:00'),
(1, '배송완료', 120000, '2026-01-14 09:00:00'),
(4, '주문취소', 45000, '2026-01-13 15:00:00');
```

```sql
SELECT * FROM orders;
```

[실행 결과]

order_id	member_id	order_status	total_amount	created_at
1	1	주문완료	50000	2026-01-15 10:30:00
2	2	결제완료	75000	2026-01-15 11:00:00
3	3	배송중	30000	2026-01-15 12:00:00
4	1	배송완료	120000	2026-01-14 09:00:00
5	4	주문취소	45000	2026-01-13 15:00:00

언뜻 보기에는 문제가 없어 보인다. 하지만 실무에서 이 방식을 그대로 사용하면 여러 가지 심각한 문제가 발생한다.

문제1: 데이터 불일치

개발자 A는 주문 상태를 '주문완료'로 입력하고, 개발자 B는 '주문 완료'(띄어쓰기 포함)로 입력한다. 또 다른 개발자 C는 'ORDER_COMPLETE'로 입력한다.

```
INSERT INTO orders (member_id, order_status, total_amount, created_at) VALUES
(5, '주문 완료', 80000, '2026-01-16 10:00:00'),
(6, 'ORDER_COMPLETE', 65000, '2026-01-16 11:00:00');
```

```
SELECT order_id, order_status FROM orders;
```

[실행 결과]

order_id	order_status
1	주문완료
2	결제완료
3	배송중
4	배송완료
5	주문취소
6	주문 완료
7	ORDER_COMPLETE

같은 의미인데 서로 다른 값이 저장되었다. 이제 '주문완료' 상태인 주문을 조회하면 어떻게 될까?

```
SELECT * FROM orders WHERE order_status = '주문완료';
```

[실행 결과]

order_id	member_id	order_status	total_amount	created_at

1	1	주문완료	50000	2026-01-15 10:30:00

분명히 3건이 주문완료 상태인데, 1건만 조회된다. 데이터 불일치로 인해 비즈니스 로직에 오류가 발생한 것이다.

문제2: 오타로 인한 버그

상태값을 직접 문자열로 입력하면 오타가 발생할 수 있다.

```
INSERT INTO orders (member_id, order_status, total_amount, created_at) VALUES
(7, '배송증', 55000, '2026-01-17 09:00:00');  -- '배송중'을 '배송증'으로 오타
```

이 데이터는 정상적으로 저장된다. 데이터베이스는 '배송증'이 오타인지 알 수 없기 때문이다. 이런 오타는 나중에 발견하기 매우 어렵고, 발견하더라도 이미 많은 데이터가 잘못 저장된 후일 수 있다.

문제3: 상태 변경의 어려움

기획팀에서 '주문완료'라는 단어가 명확하지 않다고 앞으로는 '주문완료'라는 단어 대신에 '주문성공'이라는 단어로 상태를 변경하자고 요청한다. 이 경우 이미 저장된 모든 데이터를 UPDATE 해야 한다.

```
SET SQL_SAFE_UPDATES = 0; -- 안전 업데이트 모드 끄기

UPDATE orders SET order_status = '주문성공' WHERE order_status = '주문완료';

SET SQL_SAFE_UPDATES = 1; -- 안전 업데이트 모드 활성화
```

> **MySQL 워크벤치 안전 업데이트 모드**
> MySQL 워크벤치는 실수로 인한 대량의 데이터 변경을 예방하기 위해 안전모드를 제공한다.
> WHERE 문에 키를 제공하거나 또는 안전 업데이트 모드를 꺼야 UPDATE, DELETE 문을 실행할 수 있다. (입문편 참고)

데이터가 수백만 건이라면? 이 UPDATE 문은 매우 오래 걸리고, 그 동안 테이블에 락이 걸려 서비스에 영향을 줄 수 있다. 그리고 앞서 본 것처럼 '주문 완료(띄어쓰기 포함)', 'ORDER_COMPLETE' 같은 변형된 데이터는 UPDATE 되지 않는다.

문제4: 표시 이름과 코드값의 혼재

화면에 표시되는 이름을 데이터베이스에 직접 저장하면 또 다른 문제가 생긴다. 만약 영어 서비스를 추가해야 한다면? '주문완료'를 'Order Complete'로 표시해야 하는데, 데이터베이스에 한글로 저장되어 있으면 처리가 복잡해진다.

해결책 - 코드값 분리

이 문제들을 해결하려면 **코드값**과 **표시 이름**을 분리해야 한다. 데이터베이스에는 코드값만 저장하고, 화면에 표시할 이름은 별도로 관리하는 것이다.

먼저 주문 테이블을 다시 만들어보자.

```
DROP TABLE IF EXISTS orders;

CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    member_id BIGINT NOT NULL,
    order_status VARCHAR(20) NOT NULL,
    total_amount INT NOT NULL,
    created_at DATETIME NOT NULL
);
```

이번에는 상태값을 영문 코드로 저장한다.

```
INSERT INTO orders (member_id, order_status, total_amount, created_at) VALUES
(1, 'ORDER', 50000, '2026-01-15 10:30:00'),
(2, 'PAID', 75000, '2026-01-15 11:00:00'),
(3, 'SHIPPING', 30000, '2026-01-15 12:00:00'),
(1, 'DELIVERED', 120000, '2026-01-14 09:00:00'),
(4, 'CANCEL', 45000, '2026-01-13 15:00:00');
```

```
SELECT * FROM orders;
```

[실행 결과]

order_id	member_id	order_status	total_amount	created_at
1	1	ORDER	50000	2026-01-15 10:30:00
2	2	PAID	75000	2026-01-15 11:00:00
3	3	SHIPPING	30000	2026-01-15 12:00:00
4	1	DELIVERED	120000	2026-01-14 09:00:00
5	4	CANCEL	45000	2026-01-13 15:00:00

그리고 코드값과 표시 이름의 매핑 정보는 별도로 관리한다. 가장 단순한 형태는 다음과 같다.

코드값	한글 이름	영문 이름
ORDER	주문시작	Order Received
PAID	결제완료	Payment Complete
SHIPPING	배송중	Shipping
DELIVERED	배송완료	Delivered
CANCEL	주문취소	Cancelled

이 매핑 정보를 어디에 저장할까? 바로 **공통 코드 테이블**이다.

공통 코드란?

공통 코드(Common Code)는 시스템 전체에서 사용하는 코드값과 그에 대응하는 이름을 중앙에서 관리하는 방식이다. 코드값은 데이터베이스에 저장되고, 표시 이름은 공통 코드 테이블에서 조회한다.

공통 코드를 사용하면 다음과 같은 이점이 있다.

이점1: 데이터 일관성 보장

정해진 코드값만 사용하도록 강제할 수 있다. 개발자가 임의로 다른 값을 넣을 수 없다.

이점2: 표시 이름 변경 용이

'주문접수'를 '주문완료'로 변경하고 싶다면, 공통 코드 테이블의 이름만 수정하면 된다. 실제 데이터가 저장된 주문 테이블은 건드리지 않아도 된다.

이점3: 다국어 지원 용이

한글 이름, 영문 이름 등을 공통 코드 테이블에서 관리하면 다국어 지원이 쉬워진다.

이점4: 중앙 집중 관리

모든 코드값을 한 곳에서 관리하므로, 어떤 코드값들이 있는지 한눈에 파악할 수 있다. 새로운 개발자가 투입되어도 공통 코드 테이블만 보면 시스템에서 사용하는 코드값들을 쉽게 이해할 수 있다.

공통 코드가 필요한 대표적인 경우

실무에서 공통 코드로 관리하면 좋은 데이터들을 살펴보자.

상태값

- 주문 상태: ORDER, PAID, SHIPPING, DELIVERED, CANCEL
- 회원 상태: ACTIVE, DORMANT, WITHDRAWN
- 결제 상태: PENDING, COMPLETE, FAILED, REFUND

유형/종류

- 회원 유형: NORMAL, VIP, VVIP
- 상품 유형: PHYSICAL, DIGITAL, SUBSCRIPTION
- 결제 수단: CARD, BANK, VIRTUAL, MOBILE

카테고리성 데이터

- 배송 방법: QUICK, PICKUP
- 문의 유형: PRODUCT, DELIVERY, REFUND, ETC

기타 코드성 데이터

- 성별: MALE, FEMALE
- 요일: MON, TUE, WED, THU, FRI, SAT, SUN

정리

하드코딩된 문자열로 상태값이나 유형을 관리하면 데이터 불일치, 오타로 인한 버그, 변경의 어려움 등 다양한 문제가 발생한다. 공통 코드를 도입하면 이러한 문제들을 해결하고, 코드값을 중앙에서 일관되게 관리할 수 있다.

다음 시간에는 공통 코드 테이블을 실제로 어떻게 설계하고 적용하는지 알아보겠다.

공통 코드 테이블 설계

앞서 공통 코드가 왜 필요한지 알아보았다. 이번에는 실제로 공통 코드 테이블을 어떻게 설계하고 적용하는지 단계별로 살펴보겠다.

공통 코드의 단순 설계

가장 기본적인 형태의 공통 코드 테이블부터 시작해보자.

기본 구조

공통 코드 테이블에 필요한 최소한의 정보는 무엇일까? 코드값과 표시 이름이다.

```
DROP TABLE IF EXISTS common_code;

CREATE TABLE common_code (
    code VARCHAR(50) PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);
```

- `code`를 PK로 사용한다.

주문 상태 코드를 입력해보자.

```
INSERT INTO common_code (code, name) VALUES
('ORDER', '주문접수'),
('PAID', '결제완료'),
('SHIPPING', '배송중'),
('DELIVERED', '배송완료'),
```

```
('CANCEL', '주문취소');
```

```
SELECT * FROM common_code;
```

[실행 결과]

code	name
ORDER	주문접수
PAID	결제완료
SHIPPING	배송중
DELIVERED	배송완료
CANCEL	주문취소

이제 앞서 만든 주문 테이블과 조인해서 표시 이름을 조회할 수 있다.

```
SELECT
    o.order_id,
    o.order_status,
    c.name AS order_status_name,
    o.total_amount
FROM orders o
JOIN common_code c ON o.order_status = c.code;
```

[실행 결과]

order_id	order_status	order_status_name	total_amount
1	ORDER	주문접수	50000
2	PAID	결제완료	75000
3	SHIPPING	배송중	30000

4	DELIVERED	배송완료	120000
5	CANCEL	주문취소	45000

문제점 발견

그런데 여기서 문제가 발생한다. 회원 등급 코드도 공통 코드에 추가해보자.

```
INSERT INTO common_code (code, name) VALUES
('NORMAL', '일반회원'),
('VIP', 'VIP회원'),
('VVIP', 'VVIP회원');
```

```
SELECT * FROM common_code;
```

[실행 결과]

code	name
CANCEL	주문취소
DELIVERED	배송완료
NORMAL	일반회원
ORDER	주문접수
PAID	결제완료
SHIPPING	배송중
VIP	VIP회원
VVIP	VVIP회원

주문 상태 코드와 회원 등급 코드가 섞여버렸다. 어떤 코드가 주문 상태이고, 어떤 코드가 회원 등급인지 구분할 수 없다. 더 심각한 문제는 코드값이 충돌할 수 있다는 것이다.

예를 들어 주문 상태가 아니라 결제 상태에도 'CANCEL' 코드를 사용하고 싶다면?

```
INSERT INTO common_code (code, name) VALUES
('CANCEL', '결제취소');  -- 주문취소와 코드가 중복!
```

[실행 결과]

```
Error Code: 1062. Duplicate entry 'CANCEL' for key 'common_code.PRIMARY'
```

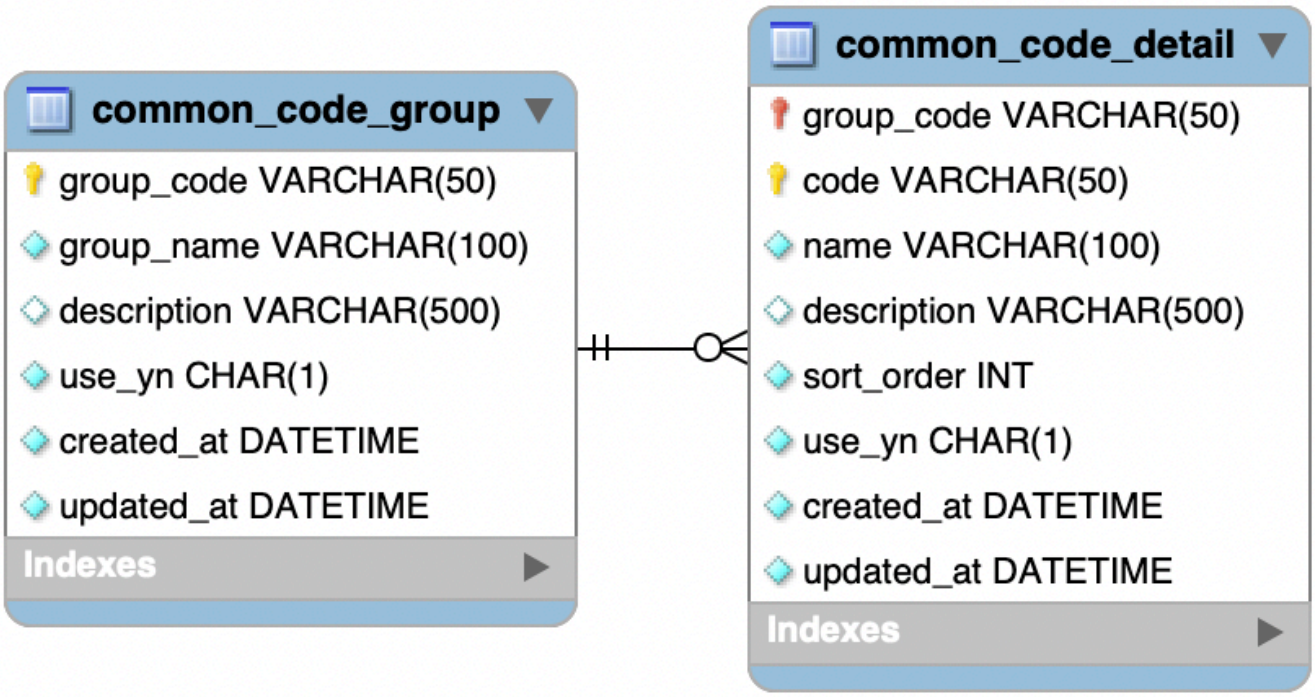
기본 키가 중복되어 입력이 실패한다. 이 문제를 해결하려면 코드를 그룹으로 분류해야 한다.

공통 코드를 더 범용성 있게 - 그룹화 설계

코드를 주문, 회원등 종류별로 구분하기 위해 그룹 개념을 도입해보자. 실무에서는 이 방식을 가장 많이 사용한다.

그룹 코드와 상세 코드 분리

공통 코드를 두 개의 테이블로 분리한다. 그룹을 관리하는 테이블과 상세 코드를 관리하는 테이블이다.




```sql
DROP TABLE IF EXISTS common_code;
DROP TABLE IF EXISTS common_code_detail;
DROP TABLE IF EXISTS common_code_group;

-- 그룹 코드 테이블
CREATE TABLE common_code_group (
    group_code VARCHAR(50) PRIMARY KEY,
    group_name VARCHAR(100) NOT NULL,
    description VARCHAR(500),
    use_yn CHAR(1) NOT NULL DEFAULT 'Y',
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

-- 상세 코드 테이블
CREATE TABLE common_code_detail (
    group_code VARCHAR(50) NOT NULL,
    code VARCHAR(50) NOT NULL,
    name VARCHAR(100) NOT NULL,
    description VARCHAR(500),
    sort_order INT NOT NULL DEFAULT 0,
    use_yn CHAR(1) NOT NULL DEFAULT 'Y',
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
```

```
CURRENT_TIMESTAMP,
    PRIMARY KEY (group_code, code),
    FOREIGN KEY (group_code) REFERENCES common_code_group(group_code)
);
```

각 컬럼의 역할을 살펴보자.

그룹 코드 테이블 (common_code_group)

- `group_code`: 그룹을 식별하는 코드 (예: ORDER_STATUS, MEMBER_GRADE)
- `group_name`: 그룹의 표시 이름 (예: 주문상태, 회원등급)
- `description`: 그룹에 대한 설명
- `use_yn`: 사용 여부 (Y/N). 코드 그룹 전체를 비활성화할 때 사용
- `created_at`, `updated_at`: 생성일시, 수정일시

상세 코드 테이블 (common_code_detail)

- `group_code`: 이 코드가 속한 그룹
- `code`: 실제 코드값
- `name`: 코드의 표시 이름
- `description`: 코드에 대한 설명
- `sort_order`: 정렬 순서. 화면에 목록을 표시할 때 순서를 지정
- `use_yn`: 사용 여부. 특정 코드만 비활성화할 때 사용
- `created_at`, `updated_at`: 생성일시, 수정일시

기본 키는 `(group_code, code)` 복합 키다. 같은 그룹 내에서만 코드가 유일하면 되므로, 다른 그룹에서는 같은 코드값을 사용할 수 있다.

> ✨ **use_yn은 왜 CHAR(1)인가요?**
> "MySQL에서 `BOOLEAN`을 사용하면 내부적으로 `TINYINT(1)` (숫자 0, 1)로 저장된다. 둘다 1 Byte를 사용하므로 저장 효율은 비슷하지만, 실무에서는 **데이터의 의미를 더 명확하게 표현하기 위해('Y'는 예, 'N'은 아니오)** `CHAR(1)`을 사용하여 'Y', 'N' 문자로 저장하는 관습이 널리 퍼져 있다. 특히 대규모 프로젝트나 레거시 시스템과의 호환성을 고려할 때 많이 사용되는 방식이다. 참고로 과거에는 `BOOLEAN` 타입이 없는 데이터베이스도 있었다. 이 방식은 'Y', 'N' 이외에 다른 단어도 입력할 수 있기 때문에 실수로 인한 버그가 발생할 수 있다. 따라서 최신 프로젝트라면 `BOOLEAN`을 사용하여 `true/false`로 매핑하는 것이 더 나은 방법이다."
> 여기서는 레거시를 포함한 다양한 예시를 보여주기 위해 이 방식을 선택했다.

데이터 입력

이제 그룹 코드와 상세 코드를 입력해보자.

참고로 실무에서는 어드민 페이지를 만들어서 어드민 페이지를 통해 공통 코드를 등록하고 관리한다.

```
-- 그룹 코드 입력
INSERT INTO common_code_group (group_code, group_name, description) VALUES
('ORDER_STATUS', '주문상태', '주문의 진행 상태를 나타내는 코드'),
('MEMBER_GRADE', '회원등급', '회원의 등급을 나타내는 코드'),
('PAYMENT_STATUS', '결제상태', '결제의 진행 상태를 나타내는 코드'),
('PAYMENT_METHOD', '결제수단', '결제 방법을 나타내는 코드');
```

```
SELECT * FROM common_code_group;
```

[실행 결과]

group_code	group_name	description	use_yn
ORDER_STATUS	주문상태	주문의 진행 상태를 나타내는 코드	Y
MEMBER_GRADE	회원등급	회원의 등급을 나타내는 코드	Y
PAYMENT_STATUS	결제상태	결제의 진행 상태를 나타내는 코드	Y
PAYMENT_METHOD	결제수단	결제 방법을 나타내는 코드	Y

- 실행 결과에 중요하지 않은 일부 필드는 생략했다.

> **실행 결과 필드 생략**
> 설명하는 핵심 내용에 집중하기 위해 앞으로 실행 결과에서 중요하지 않은 일부 필드들은 생략하겠다.

이번에는 상세 코드를 입력해보자.

```
-- 상세 코드 입력
INSERT INTO common_code_detail (group_code, code, name, sort_order) VALUES
```

```sql
-- 주문 상태
('ORDER_STATUS', 'ORDER', '주문접수', 1),
('ORDER_STATUS', 'PAID', '결제완료', 2),
('ORDER_STATUS', 'SHIPPING', '배송중', 3),
('ORDER_STATUS', 'DELIVERED', '배송완료', 4),
('ORDER_STATUS', 'CANCEL', '주문취소', 5),

-- 회원 등급
('MEMBER_GRADE', 'NORMAL', '일반회원', 1),
('MEMBER_GRADE', 'VIP', 'VIP회원', 2),
('MEMBER_GRADE', 'VVIP', 'VVIP회원', 3),

-- 결제 상태
('PAYMENT_STATUS', 'PENDING', '결제대기', 1),
('PAYMENT_STATUS', 'COMPLETE', '결제완료', 2),
('PAYMENT_STATUS', 'FAILED', '결제실패', 3),
('PAYMENT_STATUS', 'CANCEL', '결제취소', 4),
('PAYMENT_STATUS', 'REFUND', '환불완료', 5),

-- 결제 수단
('PAYMENT_METHOD', 'CARD', '신용카드', 1),
('PAYMENT_METHOD', 'BANK', '계좌이체', 2),
('PAYMENT_METHOD', 'VIRTUAL', '가상계좌', 3),
('PAYMENT_METHOD', 'MOBILE', '휴대폰결제', 4);
```

```sql
SELECT * FROM common_code_detail ORDER BY group_code, sort_order;
```

[실행 결과]

group_code	code	name	description	sort_order	use_yn
MEMBER_GRADE	NORMAL	일반회원	NULL	1	Y
MEMBER_GRADE	VIP	VIP회원	NULL	2	Y
MEMBER_GRADE	VVIP	VVIP회원	NULL	3	Y
ORDER_STATUS	ORDER	주문접수	NULL	1	Y
ORDER_STATUS	PAID	결제완료	NULL	2	Y

ORDER_STATUS	SHIPPING	배송중	NULL	3	Y
ORDER_STATUS	DELIVERED	배송완료	NULL	4	Y
ORDER_STATUS	CANCEL	주문취소	NULL	5	Y
PAYMENT_METHOD	CARD	신용카드	NULL	1	Y
PAYMENT_METHOD	BANK	계좌이체	NULL	2	Y
PAYMENT_METHOD	VIRTUAL	가상계좌	NULL	3	Y
PAYMENT_METHOD	MOBILE	휴대폰결제	NULL	4	Y
PAYMENT_STATUS	PENDING	결제대기	NULL	1	Y
PAYMENT_STATUS	COMPLETE	결제완료	NULL	2	Y
PAYMENT_STATUS	FAILED	결제실패	NULL	3	Y
PAYMENT_STATUS	CANCEL	결제취소	NULL	4	Y
PAYMENT_STATUS	REFUND	환불완료	NULL	5	Y

주목할 점은 `ORDER_STATUS`의 'CANCEL'과 `PAYMENT_STATUS`의 'CANCEL'이 공존한다는 것이다. 그룹이 다르기 때문에 같은 코드값을 사용할 수 있다.

실제 테이블에 적용

이제 주문 테이블과 결제 테이블을 만들고 공통 코드를 적용해보자.

```
DROP TABLE IF EXISTS payments;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS members;

-- 회원 테이블
CREATE TABLE members (
    member_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    grade VARCHAR(20) NOT NULL DEFAULT 'NORMAL',
```

```
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);

-- 주문 테이블
CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    member_id BIGINT NOT NULL,
    order_status VARCHAR(20) NOT NULL DEFAULT 'ORDER',
    total_amount INT NOT NULL,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (member_id) REFERENCES members(member_id)
);

-- 결제 테이블
CREATE TABLE payments (
    payment_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    order_id BIGINT NOT NULL,
    payment_method VARCHAR(20) NOT NULL,
    payment_status VARCHAR(20) NOT NULL DEFAULT 'PENDING',
    amount INT NOT NULL,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

테스트 데이터를 입력하자.

```
-- 회원 데이터
INSERT INTO members (name, email, grade) VALUES
('션', 'seon@example.com', 'NORMAL'),
('네이트', 'nate@example.com', 'VIP'),
('이순신', 'lee@example.com', 'VVIP');

-- 주문 데이터
INSERT INTO orders (member_id, order_status, total_amount) VALUES
(1, 'ORDER', 50000),
(1, 'PAID', 75000),
(2, 'SHIPPING', 120000),
(2, 'DELIVERED', 85000),
(3, 'CANCEL', 45000);

-- 결제 데이터
```

```sql
INSERT INTO payments (order_id, payment_method, payment_status, amount) VALUES
(1, 'CARD', 'PENDING', 50000),
(2, 'CARD', 'COMPLETE', 75000),
(3, 'BANK', 'COMPLETE', 120000),
(4, 'MOBILE', 'COMPLETE', 85000),
(5, 'CARD', 'CANCEL', 45000);
```

공통 코드와 조인하여 조회

이제 공통 코드와 조인하여 표시 이름을 함께 조회해보자.

회원 목록 조회

```sql
SELECT
    m.member_id,
    m.name,
    m.email,
    m.grade,
    c.name AS grade_name
FROM members m
JOIN common_code_detail c
    ON c.group_code = 'MEMBER_GRADE' AND m.grade = c.code
ORDER BY m.member_id;
```

- 코드를 구분할 때 `group_code`, `code` 두 필드를 모두 구분해야 한다.

[실행 결과]

member_id	name	email	grade	grade_name
1	션	seon@example.com	NORMAL	일반회원
2	네이트	nate@example.com	VIP	VIP회원
3	이순신	lee@example.com	VVIP	VVIP회원

주문 목록 조회

```sql
SELECT
```

```
    o.order_id,
    m.name AS member_name,
    o.order_status,
    c.name AS status_name,
    o.total_amount
FROM orders o
JOIN members m ON o.member_id = m.member_id
JOIN common_code_detail c
    ON c.group_code = 'ORDER_STATUS' AND o.order_status = c.code
ORDER BY o.order_id;
```

[실행 결과]

order_id	member_name	order_status	status_name	total_amount
1	션	ORDER	주문접수	50000
2	션	PAID	결제완료	75000
3	네이트	SHIPPING	배송중	120000
4	네이트	DELIVERED	배송완료	85000
5	이순신	CANCEL	주문취소	45000

결제 목록 조회 (여러 공통 코드 조인)

```
SELECT
    p.payment_id,
    o.order_id,
    p.payment_method,
    pm.name AS method_name,
    p.payment_status,
    ps.name AS status_name,
    p.amount
FROM payments p
JOIN orders o ON p.order_id = o.order_id
JOIN common_code_detail pm
    ON pm.group_code = 'PAYMENT_METHOD' AND p.payment_method = pm.code
JOIN common_code_detail ps
    ON ps.group_code = 'PAYMENT_STATUS' AND p.payment_status = ps.code
```

```
ORDER BY p.payment_id;
```

[실행 결과]

payment_id	order_id	payment_method	method_name	payment_status	status_name	amount
1	1	CARD	신용카드	PENDING	결제대기	50000
2	2	CARD	신용카드	COMPLETE	결제완료	75000
3	3	BANK	계좌이체	COMPLETE	결제완료	120000
4	4	MOBILE	휴대폰결제	COMPLETE	결제완료	85000
5	5	CARD	신용카드	CANCEL	결제취소	45000

특정 그룹의 코드 목록 조회

화면에서 드롭다운 목록을 표시할 때는 특정 그룹의 코드 목록이 필요하다.

```
-- 주문 상태 목록 조회
SELECT code, name
FROM common_code_detail
WHERE group_code = 'ORDER_STATUS'
  AND use_yn = 'Y'
ORDER BY sort_order;
```

- `use_yn` 필드를 사용해서 화면에 출력할 필드만 필터링 할 수 있다.
- 만약 특정 상태를 노출하고 싶지 않다면 공통 코드 테이블에서 `use_yn`만 `N`으로 변경하면 된다.

[실행 결과]

code	name
ORDER	주문접수
PAID	결제완료

SHIPPING	배송중
DELIVERED	배송완료
CANCEL	주문취소

```sql
-- 결제 수단 목록 조회
SELECT code, name
FROM common_code_detail
WHERE group_code = 'PAYMENT_METHOD'
  AND use_yn = 'Y'
ORDER BY sort_order;
```

[실행 결과]

code	name
CARD	신용카드
BANK	계좌이체
VIRTUAL	가상계좌
MOBILE	휴대폰결제

코드 비활성화

특정 코드를 더 이상 사용하지 않으려면 `use_yn`을 'N'으로 변경한다. 예를 들어 가상계좌 결제를 중단한다면:

```sql
UPDATE common_code_detail
SET use_yn = 'N'
WHERE group_code = 'PAYMENT_METHOD' AND code = 'VIRTUAL';
```

```sql
-- 결제 수단 목록 다시 조회
```

```sql
SELECT code, name
FROM common_code_detail
WHERE group_code = 'PAYMENT_METHOD'
  AND use_yn = 'Y'
ORDER BY sort_order;
```

[실행 결과]

code	name
CARD	신용카드
BANK	계좌이체
MOBILE	휴대폰결제

가상계좌가 목록에서 제외되었다. 기존에 가상계좌로 결제된 데이터는 그대로 유지되면서, 새로운 결제에서는 가상계좌를 선택할 수 없게 된다.
보통 이런 모든 조작은 어드민 툴을 통해서 운영자가 실시간으로 편리하게 관리할 수 있도록 한다.

공통 코드와 추가 속성

실무에서는 코드에 추가 속성이 필요한 경우가 있다. 예를 들어 회원 등급별 할인율, 결제 수단별 수수료율 등이다.
이런 값을 별도의 테이블에서 관리해도 되지만, 간단한 경우 공통 코드 테이블을 사용하면 편리하다.

방법1: 컬럼 추가

가장 단순한 방법은 필요한 컬럼을 약간 무식하게 추가하는 것이다.

```sql
ALTER TABLE common_code_detail
ADD COLUMN attr1 VARCHAR(100),
ADD COLUMN attr2 VARCHAR(100),
ADD COLUMN attr3 VARCHAR(100);
```

```sql
-- 회원 등급별 할인율 설정
UPDATE common_code_detail SET attr1 = '0' WHERE group_code = 'MEMBER_GRADE'
AND code = 'NORMAL';
UPDATE common_code_detail SET attr1 = '5' WHERE group_code = 'MEMBER_GRADE'
AND code = 'VIP';
UPDATE common_code_detail SET attr1 = '10' WHERE group_code = 'MEMBER_GRADE'
AND code = 'VVIP';

-- 결제 수단별 수수료율 설정
UPDATE common_code_detail SET attr1 = '2.5' WHERE group_code =
'PAYMENT_METHOD' AND code = 'CARD';
UPDATE common_code_detail SET attr1 = '0' WHERE group_code = 'PAYMENT_METHOD'
AND code = 'BANK';
UPDATE common_code_detail SET attr1 = '0' WHERE group_code = 'PAYMENT_METHOD'
AND code = 'VIRTUAL';
UPDATE common_code_detail SET attr1 = '3.0' WHERE group_code =
'PAYMENT_METHOD' AND code = 'MOBILE';
```

```sql
SELECT group_code, code, name, attr1
FROM common_code_detail
WHERE group_code IN ('MEMBER_GRADE', 'PAYMENT_METHOD')
ORDER BY group_code, sort_order;
```

[실행 결과]

group_code	code	name	attr1
MEMBER_GRADE	NORMAL	일반회원	0
MEMBER_GRADE	VIP	VIP회원	5
MEMBER_GRADE	VVIP	VVIP회원	10
PAYMENT_METHOD	CARD	신용카드	2.5
PAYMENT_METHOD	BANK	계좌이체	0
PAYMENT_METHOD	VIRTUAL	가상계좌	0
PAYMENT_METHOD	MOBILE	휴대폰결제	3.0

이 방법은 간단하지만 단점이 있다. `attr1`, `attr2` 같은 범용적인 이름을 사용하면 각 그룹에서 이 컬럼이 무엇을 의미하는지 알기 어렵다. 회원은 할인율이고, 결제 수단은 수수료율이다. 그룹 테이블에 각 속성의 의미를 설명하는 컬럼을 추가하면 이 문제를 완화할 수 있다.

```
ALTER TABLE common_code_group
ADD COLUMN attr1_name VARCHAR(50),
ADD COLUMN attr2_name VARCHAR(50),
ADD COLUMN attr3_name VARCHAR(50);
```

```
UPDATE common_code_group SET attr1_name = '할인율(%)' WHERE group_code =
'MEMBER_GRADE';
UPDATE common_code_group SET attr1_name = '수수료율(%)' WHERE group_code =
'PAYMENT_METHOD';
```

```
SELECT group_code, group_name, attr1_name
FROM common_code_group
WHERE group_code IN ('MEMBER_GRADE', 'PAYMENT_METHOD');
```

[실행 결과]

group_code	group_name	attr1_name
MEMBER_GRADE	회원등급	할인율(%)
PAYMENT_METHOD	결제수단	수수료율(%)

group_code	group_name	attr1_name	attr2_name	attr3_name
MEMBER_GRADE	회원등급	할인율(%)	적립	무료배송
PAYMENT_METHOD	결제수단	수수료율(%)		

방법2: EAV(Entity-Attribute-Value) 방식

먼저 질문을 던져보자. **회원 등급**에 필요한 속성이 정말 '할인율' 하나뿐일까?
쇼핑몰이 성장하면 다음과 같은 다양한 혜택이 추가될 수 있다.

- **할인율(`discount_rate`)**: 등급별 할인율 (예: 5%)
- **적립률(`point_rate`)**: 등급별 포인트 적립률 (예: 1%, 2%)
- **무료배송 여부(`free_shipping`)**: 무료배송 혜택 제공 여부 (Y/N)
- **우선 상담 여부(`priority_support`)**: 고객센터 우선 상담 혜택 (Y/N)

앞서 본 **방법1(컬럼 추가)** 방식을 사용한다면 `attr1`, `attr2`만으로는 부족하다. `attr3`, `attr4`를 계속 추가해야 한다. 게다가 어떤 그룹은 속성이 1개만 필요한데, 회원 등급 그룹 때문에 테이블에 의미 없는 컬럼이 계속 늘어나는 문제가 발생한다.

이처럼 속성의 개수가 가변적이고, 그룹마다 필요한 속성이 제각각일 때 **EAV(Entity-Attribute-Value)** 방식을 사용한다.

> **EAV 방식은 뒤에서 자세히 설명한다. 지금은 이런 문제에 대한 대안책이 있다는 점만 알아두자**

하지만 EAV 방식은 **조회 성능**이 떨어질 수 있고, 데이터를 가져와서 애플리케이션에서 가공하거나 복잡한 조인 쿼리(Pivot)를 작성해야 한다는 단점이 있다. 따라서 속성이 2~3개 정도로 고정되어 있다면 **방법1(컬럼 추가)**이 좋고, 속성이 매우 다양하고 자주 변경된다면 **방법2(EAV)**가 적합하다.

프로젝트가 크지 않다면 **실용적인 관점에서 방법1을 우선 고려**하자.

정리

공통 코드 테이블 설계의 핵심 포인트를 정리해보자.

그룹화 설계의 장점

- 코드를 종류별로 분류하여 관리할 수 있다
- 같은 코드값을 다른 그룹에서 사용할 수 있다
- 특정 그룹의 코드 목록을 쉽게 조회할 수 있다

주요 컬럼

- `group_code` : 코드 그룹 식별자
- `code` : 실제 코드값
- `name` : 표시 이름
- `sort_order` : 정렬 순서
- `use_yn` : 사용 여부

추가 속성 처리

- 속성이 적으면: 컬럼 추가 방식 (실용적 선택)
- 속성이 많고 다양하면: EAV 방식

다음 시간에는 공통 코드를 사용할 때 발생하는 성능 문제와 그 해결 방법에 대해 알아보겠다.

공통 코드의 단점

공통 코드를 도입하면 코드 관리가 편리해지지만, 또 다른 복잡성과 성능과 관련된 고민이 생긴다. 이번 시간에는 공통 코드 사용 시 발생하는 복잡성과 성능 문제의 트레이드오프에 대해 알아보겠다.

공통 코드 사용 시 단점

앞서 공통 코드와 조인하여 표시 이름을 조회하는 방법을 배웠다. 그런데 실무에서 이 방식을 사용하면 몇 가지 불편한 점이 있다.

문제1: SQL이 복잡해진다

먼저 기존 테이블과 데이터를 준비하자.

```
DROP TABLE IF EXISTS payments;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS members;
DROP TABLE IF EXISTS common_code_attribute; -- 복습시 문제가 되는 테이블 제거
DROP TABLE IF EXISTS common_code_detail;
DROP TABLE IF EXISTS common_code_group;

-- 그룹 코드 테이블
CREATE TABLE common_code_group (
```

```sql
    group_code VARCHAR(50) PRIMARY KEY,
    group_name VARCHAR(100) NOT NULL
);

-- 상세 코드 테이블
CREATE TABLE common_code_detail (
    group_code VARCHAR(50) NOT NULL,
    code VARCHAR(50) NOT NULL,
    name VARCHAR(100) NOT NULL,
    sort_order INT NOT NULL DEFAULT 0,
    use_yn CHAR(1) NOT NULL DEFAULT 'Y',
    PRIMARY KEY (group_code, code),
    FOREIGN KEY (group_code) REFERENCES common_code_group(group_code)
);

-- 회원 테이블
CREATE TABLE members (
    member_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    grade VARCHAR(20) NOT NULL DEFAULT 'NORMAL'
);

-- 주문 테이블
CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    member_id BIGINT NOT NULL,
    order_status VARCHAR(20) NOT NULL DEFAULT 'ORDER',
    total_amount INT NOT NULL,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (member_id) REFERENCES members(member_id)
);

-- 결제 테이블
CREATE TABLE payments (
    payment_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    order_id BIGINT NOT NULL,
    payment_method VARCHAR(20) NOT NULL,
    payment_status VARCHAR(20) NOT NULL DEFAULT 'PENDING',
    amount INT NOT NULL,
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

```sql
-- 공통 코드 데이터
INSERT INTO common_code_group (group_code, group_name) VALUES
('ORDER_STATUS', '주문상태'),
('MEMBER_GRADE', '회원등급'),
('PAYMENT_STATUS', '결제상태'),
('PAYMENT_METHOD', '결제수단');

INSERT INTO common_code_detail (group_code, code, name, sort_order) VALUES
('ORDER_STATUS', 'ORDER', '주문접수', 1),
('ORDER_STATUS', 'PAID', '결제완료', 2),
('ORDER_STATUS', 'SHIPPING', '배송중', 3),
('ORDER_STATUS', 'DELIVERED', '배송완료', 4),
('ORDER_STATUS', 'CANCEL', '주문취소', 5),
('MEMBER_GRADE', 'NORMAL', '일반회원', 1),
('MEMBER_GRADE', 'VIP', 'VIP회원', 2),
('MEMBER_GRADE', 'VVIP', 'VVIP회원', 3),
('PAYMENT_STATUS', 'PENDING', '결제대기', 1),
('PAYMENT_STATUS', 'COMPLETE', '결제완료', 2),
('PAYMENT_STATUS', 'FAILED', '결제실패', 3),
('PAYMENT_STATUS', 'CANCEL', '결제취소', 4),
('PAYMENT_METHOD', 'CARD', '신용카드', 1),
('PAYMENT_METHOD', 'BANK', '계좌이체', 2),
('PAYMENT_METHOD', 'MOBILE', '휴대폰결제', 3);

-- 테스트 데이터
INSERT INTO members (name, email, grade) VALUES
('션', 'seon@example.com', 'NORMAL'),
('네이트', 'nate@example.com', 'VIP'),
('이순신', 'lee@example.com', 'VVIP');

INSERT INTO orders (member_id, order_status, total_amount, created_at) VALUES
(1, 'ORDER', 50000, '2026-01-15 10:00:00'),
(1, 'PAID', 75000, '2026-01-15 11:00:00'),
(2, 'SHIPPING', 120000, '2026-01-16 09:00:00'),
(2, 'DELIVERED', 85000, '2026-01-14 15:00:00'),
(3, 'CANCEL', 45000, '2026-01-13 14:00:00');

INSERT INTO payments (order_id, payment_method, payment_status, amount) VALUES
(1, 'CARD', 'PENDING', 50000),
(2, 'CARD', 'COMPLETE', 75000),
(3, 'BANK', 'COMPLETE', 120000),
(4, 'MOBILE', 'COMPLETE', 85000),
(5, 'CARD', 'CANCEL', 45000);
```

공통 코드 없이 단순하게 주문 목록을 조회하면 다음과 같다.

```sql
-- 공통 코드 없이 단순 조회
SELECT
    o.order_id,
    m.name AS member_name,
    o.order_status,
    o.total_amount
FROM orders o
JOIN members m ON o.member_id = m.member_id
ORDER BY o.order_id;
```

[실행 결과]

order_id	member_name	order_status	total_amount
1	션	ORDER	50000
2	션	PAID	75000
3	네이트	SHIPPING	120000
4	네이트	DELIVERED	85000
5	이순신	CANCEL	45000

SQL이 간단하다. 하지만 `order_status`에 'ORDER', 'PAID' 같은 코드값이 표시된다.
그런데 화면에는 '주문접수', '결제완료' 같은 한글 이름을 표시해야 한다. 코드의 이름을 찾기 위해 공통 코드와 조인해보자.

```sql
-- 공통 코드와 조인하여 조회
SELECT
    o.order_id,
    m.name AS member_name,
    o.order_status,
    os.name AS order_status_name,
```

```
    o.total_amount
FROM orders o
JOIN members m
    ON o.member_id = m.member_id
JOIN common_code_detail os
    ON os.group_code = 'ORDER_STATUS' AND o.order_status = os.code
ORDER BY o.order_id;
```

[실행 결과]

order_id	member_name	order_status	order_status_name	total_amount
1	션	ORDER	주문접수	50000
2	션	PAID	결제완료	75000
3	네이트	SHIPPING	배송중	120000
4	네이트	DELIVERED	배송완료	85000
5	이순신	CANCEL	주문취소	45000

조인이 하나 더 추가되었다. 아직은 괜찮아 보인다. 그런데 회원 등급 이름도 함께 조회해야 한다면?

```
-- 회원 등급까지 조인
SELECT
    o.order_id,
    m.name AS member_name,
    m.grade,
    mg.name AS grade_name,
    o.order_status,
    os.name AS order_status_name,
    o.total_amount
FROM orders o
JOIN members m
    ON o.member_id = m.member_id
JOIN common_code_detail os
    ON os.group_code = 'ORDER_STATUS' AND o.order_status = os.code
JOIN common_code_detail mg
    ON mg.group_code = 'MEMBER_GRADE' AND m.grade = mg.code
```

```
ORDER BY o.order_id;
```

[실행 결과]

order_id	member_name	grade	grade_name	order_status	order_status_name	total_amount
1	션	NORMAL	일반회원	ORDER	주문접수	50000
2	션	NORMAL	일반회원	PAID	결제완료	75000
3	네이트	VIP	VIP회원	SHIPPING	배송중	120000
4	네이트	VIP	VIP회원	DELIVERED	배송완료	85000
5	이순신	VVIP	VVIP회원	CANCEL	주문취소	45000

조인이 또 추가되었다. 이제 결제 정보까지 함께 조회해야 한다면 어떻게 될까?

```
-- 결제 정보까지 모두 조인
SELECT
    o.order_id,
    m.name AS member_name,
    m.grade,
    mg.name AS grade_name,
    o.order_status,
    os.name AS order_status_name,
    o.total_amount,
    p.payment_method,
    pm.name AS payment_method_name,
    p.payment_status,
    ps.name AS payment_status_name,
    p.amount AS payment_amount
FROM orders o
JOIN members m ON o.member_id = m.member_id
JOIN payments p ON o.order_id = p.order_id
JOIN common_code_detail os
    ON os.group_code = 'ORDER_STATUS' AND o.order_status = os.code
JOIN common_code_detail mg
    ON mg.group_code = 'MEMBER_GRADE' AND m.grade = mg.code
```

```
JOIN common_code_detail pm
    ON pm.group_code = 'PAYMENT_METHOD' AND p.payment_method = pm.code
JOIN common_code_detail ps
    ON ps.group_code = 'PAYMENT_STATUS' AND p.payment_status = ps.code
ORDER BY o.order_id;
```

[실행 결과]

order_id	member_name	...	payment_method	payment_method_name	payment_status	payment_status_name
1	션	...	CARD	신용카드	PENDING	결제대기
2	션	...	CARD	신용카드	COMPLETE	결제완료
3	네이트	...	BANK	계좌이체	COMPLETE	결제완료
4	네이트	...	MOBILE	휴대폰결제	COMPLETE	결제완료
5	이순신	...	CARD	신용카드	CANCEL	결제취소

SQL이 매우 복잡해졌다. 비즈니스 테이블 조인은 3개(orders, members, payments)인데, 공통 코드 조인이 4개나 된다. 공통 코드 때문에 SQL이 2배 이상 복잡해진 것이다.

문제2: 코드 이름 조회 로직이 중복된다

주문 목록을 조회하는 곳이 여러 곳이라면 어떻게 될까? 관리자 페이지, 사용자 마이페이지, API 등 여러 곳에서 주문 목록을 조회한다. 모든 곳에서 공통 코드 조인을 반복해야 한다.

```
-- 관리자용 주문 목록 조회
SELECT
    o.order_id,
    m.name AS member_name,
    o.order_status,
    os.name AS order_status_name,  -- 중복!
    o.total_amount
FROM orders o
JOIN members m ON o.member_id = m.member_id
JOIN common_code_detail os
```

```
    ON os.group_code = 'ORDER_STATUS' AND o.order_status = os.code
ORDER BY o.created_at DESC;
```

[실행 결과]

order_id	member_name	order_status	order_status_name	total_amount
3	네이트	SHIPPING	배송중	120000
2	션	PAID	결제완료	75000
1	션	ORDER	주문접수	50000
4	네이트	DELIVERED	배송완료	85000
5	이순신	CANCEL	주문취소	45000

```
-- 사용자 마이페이지용 주문 목록 조회
SELECT
    o.order_id,
    o.order_status,
    os.name AS order_status_name,  -- 중복!
    o.total_amount
FROM orders o
JOIN common_code_detail os
    ON os.group_code = 'ORDER_STATUS' AND o.order_status = os.code
WHERE o.member_id = 1
ORDER BY o.created_at DESC;
```

[실행 결과]

order_id	order_status	order_status_name	total_amount
2	PAID	결제완료	75000
1	ORDER	주문접수	50000

```
-- 배송중인 주문 목록 조회
```

```sql
SELECT
    o.order_id,
    m.name AS member_name,
    o.order_status,
    os.name AS order_status_name,  -- 중복!
    o.total_amount
FROM orders o
JOIN members m ON o.member_id = m.member_id
JOIN common_code_detail os
    ON os.group_code = 'ORDER_STATUS' AND o.order_status = os.code
WHERE o.order_status = 'SHIPPING';
```

[실행 결과]

order_id	member_name	order_status	order_status_name	total_amount
3	네이트	SHIPPING	배송중	120000

세 개의 SQL 모두 `order_status_name`을 조회하기 위해 동일한 조인을 반복한다. 만약 공통 코드 테이블 구조가 변경되면? 모든 SQL을 수정해야 한다.

문제3: SELECT 결과에 항상 이름 필드가 필요하다

공통 코드와 조인하면 조회 결과에 코드값과 이름이 모두 포함된다. 코드값만 필요한 경우에도 이름까지 함께 조회하게 되어 불필요한 데이터가 전송된다.

예를 들어서 애플리케이션에서 코드값만 필요한 경우도 있고, 코드값과 코드의 이름이 모두 필요한 경우도 있다고 가정해보자.

1. 코드 값만 구하는 SQL

```sql
-- 코드값만 구하는 SQL
SELECT
    o.order_id,
    o.order_status
FROM orders o
WHERE o.order_status IN ('ORDER', 'PAID');
```

order_id	order_status
1	ORDER
2	PAID

2. 코드 값과 코드 이름을 모두 구하는 SQL

```sql
-- 코드값과 코드 이름을 모두 구하는 SQL
SELECT
    o.order_id,
    o.order_status,
    os.name AS order_status_name  -- 상황에 따라 필요하지 않을 수 있음
FROM orders o
JOIN common_code_detail os
    ON os.group_code = 'ORDER_STATUS' AND o.order_status = os.code
WHERE o.order_status IN ('ORDER', 'PAID');
```

order_id	order_status	order_status_name
1	ORDER	주문접수
2	PAID	결제완료

이 경우 애플리케이션 개발자의 선택지는 다음과 같다.

1. 코드값과 코드 이름을 모두 구하는 SQL을 하나만 만들어서 중복을 줄이고 범용적으로 함께 사용한다.
2. 코드값만 구하는 SQL과 코드값과 코드 이름을 모두 구하는 SQL을 각각 따로 만들어서 사용한다.

1번의 경우 코드값만 필요한 경우에도 코드 이름까지 함께 조회하게 되어 불필요한 데이터가 전송되는 단점이 있다.
2번의 경우 성능은 최적화 되지만, 거의 비슷한 SQL을 두 번 중복으로 개발해야 하는 단점이 있다.

문제4: 서브쿼리 방식도 복잡하다

공통 코드의 값을 구할 때 조인 대신 서브쿼리를 사용할 수도 있다.

```sql
SELECT
```

```
    o.order_id,
    m.name AS member_name,
    o.order_status,
    (SELECT name FROM common_code_detail
     WHERE group_code = 'ORDER_STATUS' AND code = o.order_status) AS
order_status_name,
    m.grade,
    (SELECT name FROM common_code_detail
     WHERE group_code = 'MEMBER_GRADE' AND code = m.grade) AS grade_name,
    o.total_amount
FROM orders o
JOIN members m ON o.member_id = m.member_id
ORDER BY o.order_id;
```

[실행 결과]

order_id	member_name	order_status	order_status_name	grade	grade_name	total_amount
1	션	ORDER	주문접수	NORMAL	일반회원	50000
2	션	PAID	결제완료	NORMAL	일반회원	75000
3	네이트	SHIPPING	배송중	VIP	VIP회원	120000
4	네이트	DELIVERED	배송완료	VIP	VIP회원	85000
5	이순신	CANCEL	주문취소	VVIP	VVIP회원	45000

서브쿼리 방식은 조인 개수를 줄여주지만, 코드 이름이 필요한 컬럼마다 서브쿼리를 작성해야 하므로 SQL이 여전히 복잡하다. 또한 대체로 동일 기능의 조인보다 성능 튜닝 여지가 적고, 가독성도 떨어진다.

실무 개발자의 괴로움 - 조인 지옥(Join Hell)

이러한 문제들로 인해 실무에서는 소위 **"조인 지옥(Join Hell)"**이라는 상황이 벌어진다.

단순히 주문 목록 하나를 조회하려고 했을 뿐인데, 화면에 표시할 이름들을 가져오기 위해 공통 코드 테이블을 4번, 5번씩 조인해야 한다. SQL은 점점 길어지고, 가독성은 떨어지며, 쿼리 성능은 저하된다.

신입 개발자가 들어와서 "선배님, 주문 테이블에 왜 이렇게 조인이 많아요?"라고 물어보면, "응, 그거 코드 이름 가져오느라 그래. 그냥 복사해서 써."라고 대답하게 되는 슬픈 현실이 펼쳐진다.

데이터베이스 설계 관점에서는 정규화를 통해 중복을 제거하고 데이터를 일관성 있게 관리하는 것이 맞다. 하지만 애플리케이션 개발 생산성과 성능 관점에서는 이것이 오히려 독이 될 수 있다.

그렇다면 이 딜레마를 어떻게 해결해야 할까?

공통 코드의 단점 해결 방안 1

애플리케이션에서 공통 코드 조회

결과적으로 실무에서는 공통 코드 때문에 SQL이 매우 복잡해지고, 개발자는 고통 받는다.
이런 단점을 극복할 수 있는 방안이 있는데, 공통 코드 이름을 SQL에서 조회하지 않고, 애플리케이션에서 별도로 조회하는 것이다.

기본 아이디어

1. SQL은 코드값만 조회한다 (공통 코드 조인 없음)
2. 애플리케이션에서 공통 코드 테이블을 별도로 조회한다
3. 애플리케이션에서 코드값을 이름으로 변환한다

```sql
-- 1단계: 주문 목록 조회 (공통 코드 조인 없음)
SELECT
    o.order_id,
    m.name AS member_name,
    m.grade,
    o.order_status,
    o.total_amount
FROM orders o
JOIN members m ON o.member_id = m.member_id
ORDER BY o.order_id;
```

[실행 결과]

order_id	member_name	grade	order_status	total_amount
1	션	NORMAL	ORDER	50000
2	션	NORMAL	PAID	75000
3	네이트	VIP	SHIPPING	120000
4	네이트	VIP	DELIVERED	85000
5	이순신	VVIP	CANCEL	45000

SQL이 훨씬 단순해졌다. 이제 애플리케이션에서 공통 코드를 별도로 조회한다.

```sql
-- 2단계: 주문 상태 코드 목록 조회
SELECT code, name
FROM common_code_detail
WHERE group_code = 'ORDER_STATUS';
```

[실행 결과]

code	name
ORDER	주문접수
PAID	결제완료
SHIPPING	배송중
DELIVERED	배송완료
CANCEL	주문취소

```sql
-- 2단계: 회원 등급 코드 목록 조회
SELECT code, name
FROM common_code_detail
WHERE group_code = 'MEMBER_GRADE';
```

[실행 결과]

code	name
NORMAL	일반회원
VIP	VIP회원
VVIP	VVIP회원

애플리케이션에서는 이 코드 목록을 Map(딕셔너리, Key-Value), JSON 등의 형태로 저장해두고, 주문 목록의 코드값을 이름으로 변환한다.

```
// 의사 코드 (Pseudo Code)
orderStatusMap = {ORDER: '주문접수', PAID: '결제완료', ...} // 주문 상태 공통 코드 맵에 저장
memberGradeMap = {NORMAL: '일반회원', VIP: 'VIP회원', ...} // 회원 등급 공통 코드 맵에 저장

order.orderStatusName = orderStatusMap['order.orderStatus'] // 공통 코드 -> 공통 코드 이름 찾기
order.gradeName = memberGradeMap[order.grade]
```

실무에서는 더 편리하게 재사용할 수 있도록 공통 코드를 넣으면 코드값이 나오는 클래스나 함수를 만들어서 사용하면 된다.

```
//함수: 코드 그룹과 해당 코드 그룹에 속하는 코드를 넣으면 코드 이름을 반환
String getCodeName(String codeGroup, String code);

// order
order.orderStatusName = code.getCodeName("ORDER_STATUS", order.orderStatus)
order.gradeName = code.getCodeName("MEMBER_GRADE", order.grade)
```

장점

- SQL이 단순해진다
- 공통 코드 조회 로직을 한 곳에서 관리할 수 있다
- 필요한 곳에서 언제든지 코드 이름을 조회할 수 있다

단점: 다량의 SQL 실행 문제

그런데 이 방식에는 심각한 단점이 있다. 주문 목록을 조회할 때마다 공통 코드를 조회하는 쿼리가 추가로 실행된다.

```
1. 주문 목록 조회: 1번 쿼리
2. 주문 상태 공통 코드 조회: 1번 쿼리
3. 회원 등급 공통 코드 조회: 1번 쿼리
총 3번의 쿼리, 3번의 네트워크 통신
```

만약 결제 정보까지 조회한다면?

```
1. 주문 목록 조회: 1번 쿼리
2. 주문 상태 공통 코드 조회: 1번 쿼리
3. 회원 등급 공통 코드 조회: 1번 쿼리
4. 결제 수단 공통 코드 조회: 1번 쿼리
5. 결제 상태 공통 코드 조회: 1번 쿼리
총 5번의 쿼리, 5번의 네트워크 통신
```

이번에는 N+1 문제로 불리는 더 심각한 경우를 생각해보자. 만약 각 주문마다 공통 코드를 개별적으로 조회한다면?

```
1. 주문 목록 조회: 1번 쿼리 (5건 조회)
2. 주문1의 상태 코드 조회: 1번 쿼리
3. 주문2의 상태 코드 조회: 1번 쿼리
4. 주문3의 상태 코드 조회: 1번 쿼리
5. 주문4의 상태 코드 조회: 1번 쿼리
6. 주문5의 상태 코드 조회: 1번 쿼리
총 6번의 쿼리 (1 + 5)
```

이것이 바로 **N+1 문제**다. 목록을 조회하는 1번의 쿼리와, 목록의 각 항목마다 추가로 N번의 쿼리가 실행된다. 주문이 100건이면 101번, 1000건이면 1001번의 쿼리가 실행된다.

IN 절로 N+1 문제 완화

N+1 문제를 완화하려면 IN 절을 사용해서 한 번에 조회할 수 있다.

```sql
-- 주문 목록에서 사용된 상태 코드들을 한 번에 조회
SELECT code, name
FROM common_code_detail
WHERE group_code = 'ORDER_STATUS'
  AND code IN ('ORDER', 'PAID', 'SHIPPING', 'DELIVERED', 'CANCEL');
```

이렇게 조회한 내용을 앞서 설명한 것 처럼 Map에 잘 저장해서 사용하는 것이다.

하지만 이 방식도 몇 가지 문제가 있다.

1. 애플리케이션 로직이 복잡해진다. 먼저 주문 목록에서 사용된 코드값들을 수집하고, IN 절로 조회한 후, 다시 매핑해야 한다.
2. 여전히 추가 쿼리가 필요하다. 네트워크 통신이 1번 더 발생하는 것은 피할 수 없다.

트레이드오프 정리

지금까지 살펴본 두 가지 방식의 트레이드오프를 정리해보자.

방식1: SQL에서 공통 코드 조인

장점	단점
한 번의 쿼리로 모든 데이터 조회	SQL이 복잡해짐
네트워크 통신 1회	공통 코드 조인 로직이 여러 SQL에 중복
데이터베이스에서 최적화 가능	공통 코드 테이블 구조 변경 시 모든 SQL 수정 필요

방식2: 애플리케이션에서 공통 코드 조회

장점	단점
SQL이 단순해짐	추가 쿼리 필요 (네트워크 통신 증가)
공통 코드 조회 로직 중앙화	N+1 문제 발생 가능
유지보수 용이	애플리케이션 로직 복잡도 증가

어떤 방식이 더 좋을까? 정답은 없다. 상황에 따라 적절한 방식을 선택해야 한다.

- **성능이 중요하고 SQL 복잡도를 감수할 수 있다면**: SQL에서 공통 코드 조인
- **유지보수성이 중요하고 약간의 성능 저하를 감수할 수 있다면**: 애플리케이션에서 공통 코드 조회

이론적으로는 상황에 따라 적절한 방식을 선택해야 한다고 말하겠지만, 실무에서 SQL에서 직접 조인이나 서브쿼리를 통해 공통 코드를 조회하는 방식은 SQL이 복잡해지는 것은 물론이고, 유지보수를 어렵게 하는 문제가 있다.
안그래도 복잡한 SQL이 코드 이름이라는 부가 정보 때문에 수 많이 조인이 발생하고 그 결과 SQL을 분석할 때 이 SQL의 핵심 로직이 무엇인지 집중하기가 어려워진다. SQL을 작성할 때 공통 코드 조인을 무수히 많이 해본 개발자라면 아마도 그 괴로움을 잘 알 것이다.

그런데 두 가지 장점을 모두 가져갈 수 있는 방법이 있다. 바로 **캐싱**이다. 다음 시간에는 캐싱을 통해 애플리케이션 공통 코드 조회 방식의 단점을 보완하는 방법을 알아보겠다.

공통 코드의 단점 해결 방안 2

캐싱을 통한 애플리케이션 공통 코드 조회 단점 극복

앞서 애플리케이션에서 공통 코드를 별도로 조회하면 SQL이 단순해지지만, 추가 쿼리로 인한 네트워크 통신이 발생한다는 단점이 있었다. 이번 시간에는 캐싱을 통해 이 단점을 극복하는 방법을 알아보겠다.

공통 코드의 특성

먼저 공통 코드 데이터의 특성을 생각해보자.

특성1: 데이터가 매우 적다

공통 코드 테이블에는 데이터가 얼마나 있을까? 일반적인 쇼핑몰 시스템을 예로 들어보자.

그룹	코드 수
주문 상태	5개
회원 등급	3개
결제 상태	5개

결제 수단	4개
배송 방법	3개
상품 유형	4개
문의 유형	5개
쿠폰 유형	3개
…	…

그룹이 50개이고 각 그룹에 평균 5개의 코드가 있다고 해도 총 250개 정도다. 대규모 시스템이라도 공통 코드는 보통 수백 개를 넘지 않는다.

특성2: 변경이 거의 없다
공통 코드는 한번 정의되면 거의 변경되지 않는다. 새로운 코드가 추가되거나 표시 이름이 변경되는 경우는 있지만, 그 빈도는 매우 낮다. 일주일에 한 번 변경되면 많이 변경되는 편이다.

특성3: 조회가 매우 빈번하다
반면 조회는 매우 빈번하게 발생한다. 모든 화면에서 코드의 표시 이름이 필요하기 때문이다. 주문 목록, 회원 목록, 결제 내역 등 거의 모든 화면에서 공통 코드를 조회한다.

이러한 특성을 정리하면:

- 데이터 크기: 매우 작음 (수백 건)
- 변경 빈도: 매우 낮음 (일주일에 몇 번)
- 조회 빈도: 매우 높음 (초당 수백~수천 번)

이런 특성의 데이터는 **캐싱**에 완벽하게 적합하다.

메모리 캐싱의 위력

캐싱의 효과를 이해하려면 먼저 각 저장소의 접근 속도를 알아야 한다.

저장소별 접근 속도 비교

저장소	실제 속도	상대 비용(배수)

L1 캐시	1ns	1
메모리 (RAM)	100ns	100
SSD	100μs (100,000ns)	100,000
네트워크 (같은 데이터센터)	500μs (500,000ns)	500,000
네트워크 (해외)	150ms (150,000,000ns)	150,000,000

- 참고로 이 수치는 매우 대략적인 수치이다. 실제 수치는 시스템의 상황에 따라서 다르다. 여기서는 단순히 그 차이가 매우 크다는 것만 이해하면 충분하다.

메모리 접근과 네트워크 접근의 차이는 약 5,000배다. 실제로는 네트워크 지연, 데이터베이스 처리 시간 등이 추가되어 그 차이는 훨씬 더 커진다. 데이터베이스는 보통 같은 데이터센터에 별도의 서버로 구성되어 있으므로 애플리케이션에서 네트워크 통신이 필요하다.

간단히 말해서:

- **메모리 호출**: 거의 공짜
- **네트워크 호출 (DB 조회)**: 비용이 큼

공통 코드를 메모리에 캐싱해두면, 네트워크 호출 없이 즉시 코드 이름을 조회할 수 있다.

로컬 메모리 캐싱

가장 단순한 캐싱 방법은 애플리케이션을 시작할 때 애플리케이션의 메모리에 공통 코드를 모두 불러서 올려두는 것이다.

기본 구조

```
// 의사 코드 (Pseudo Code)
class CommonCodeCache {
    // 캐시 저장소: Map<그룹코드, Map<코드, 이름>>
    private Map<String, Map<String, String>> cache;

    // 애플리케이션 시작 시 캐시 로딩
    void init() {
        // DB에서 모든 공통 코드 조회
        List<CommonCode> codes = db.query(
```

```
            "SELECT group_code, code, name FROM common_code_detail WHERE
use_yn = 'Y'"
        );

        // 캐시에 저장
        for (code in codes) {
            cache[code.groupCode][code.code] = code.name;
        }
    }

    // 코드 이름 조회 (메모리에서 즉시 반환)
    String getName(String groupCode, String code) {
        return cache[groupCode][code];
    }
}
```

사용 예시

```
// 주문 목록 조회
orders = db.query("SELECT * FROM orders");

// 각 주문의 상태 이름 변환 (메모리 조회, 네트워크 호출 없음!)
for (order in orders) {
    order.statusName = CommonCodeCache.getName("ORDER_STATUS", order.status);
}
```

이제 공통 코드 이름을 조회할 때 데이터베이스에 쿼리를 보내지 않는다. 메모리에서 즉시 조회하므로 성능이 매우 빠르다.

메모리 사용량

공통 코드 250개를 캐싱한다고 가정하자. 각 코드당 평균 100바이트(그룹코드 20자 + 코드 20자 + 이름 50자 + 기타)라고 하면:

```
250개 × 100바이트 = 25,000바이트 = 약 25KB
```

25KB는 현대 서버의 메모리(보통 8GB~64GB)에서 무시해도 될 정도로 작다. 캐싱에 대한 메모리 부담은 전혀 없다.

캐시 동기화 문제

로컬 메모리 캐싱은 간단하고 빠르지만, 심각한 문제가 하나 있다. 바로 **동기화 문제**다.

문제 상황

1. 애플리케이션 시작 시 공통 코드를 캐시에 로딩
2. 운영 중 관리자가 데이터베이스의 공통 코드 테이블에 '주문접수'를 '주문완료'로 이름 변경
3. 데이터베이스는 변경되었지만, 캐시는 여전히 '주문접수'

```
[데이터베이스]
ORDER_STATUS / ORDER / 주문완료  (변경됨!)

[애플리케이션 캐시]
ORDER_STATUS / ORDER / 주문접수  (예전 값 그대로)
```

사용자는 여전히 '주문접수'를 보게 된다. 데이터베이스와 캐시가 불일치하는 것이다.

해결책1: 관리자 페이지에서 캐시 갱신

가장 직관적인 해결책은 관리자가 공통 코드를 수정할 때 캐시도 함께 갱신하는 것이다.

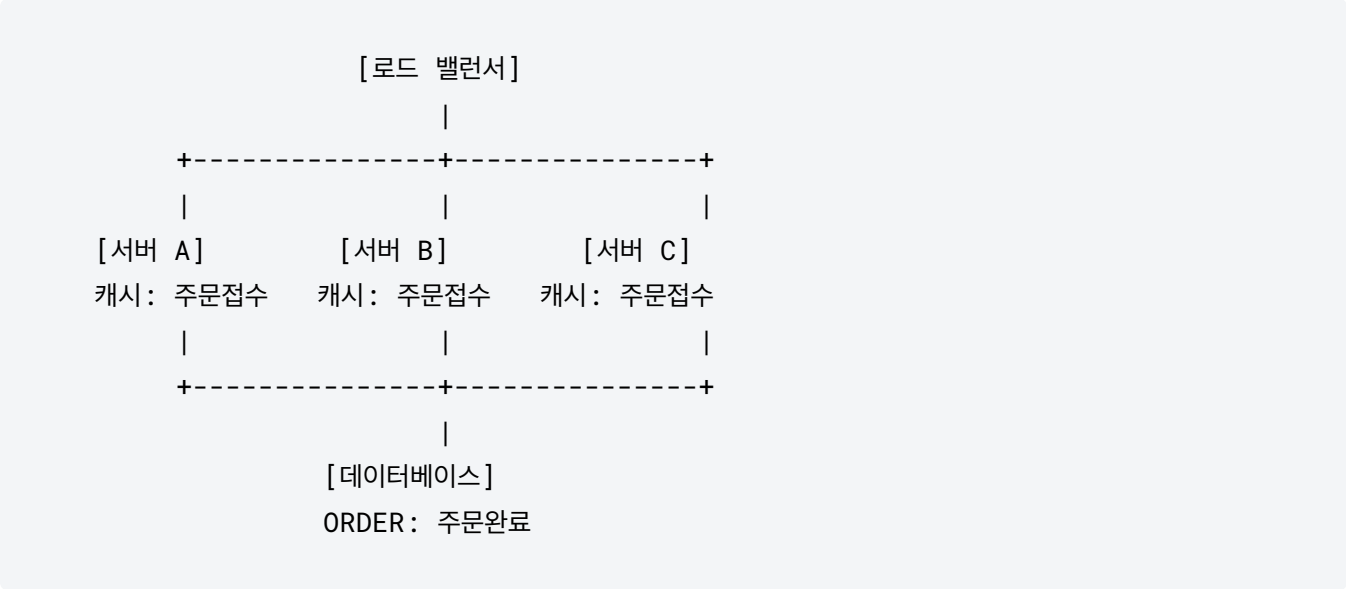
```
// 관리자 페이지: 공통 코드 수정
void updateCommonCode(String groupCode, String code, String newName) {
    // 1. DB 업데이트
    db.execute("UPDATE common_code_detail SET name = ? WHERE group_code = ?
AND code = ?",
               newName, groupCode, code);

    // 2. 캐시 갱신
    CommonCodeCache.refresh();
}
```

하지만 이 방법에는 문제가 있다.

다중 서버 환경의 문제

실제 운영 환경에서는 애플리케이션 서버가 여러 대 운영된다. 트래픽이 많은 서비스는 수십 대, 수백 대의 서버가 운영되기도 한다.



관리자가 공통 코드를 수정하면:

1. 데이터베이스 업데이트
2. 관리자 요청을 처리한 서버(예: 서버 A)의 캐시만 갱신
3. 서버 B, C의 캐시는 여전히 예전 값

사용자 요청이 서버 A로 가면 '주문완료'가 표시되고, 서버 B나 C로 가면 '주문접수'가 표시된다. 같은 화면인데 새로고침할 때마다 다른 값이 보이는 황당한 상황이 발생한다.

해결책2: 모든 서버에 캐시 갱신 요청

모든 서버에 캐시 갱신 요청을 보내는 방법이 있다.

```
// 관리자 페이지: 공통 코드 수정
void updateCommonCode(String groupCode, String code, String newName) {
    // 1. DB 업데이트
    db.execute("UPDATE ...");

    // 2. 모든 서버에 캐시 갱신 요청
    for (server in getAllServers()) {
        http.post(server + "/cache/refresh");
    }
}
```

이 방법은 가능하지만 복잡하다.

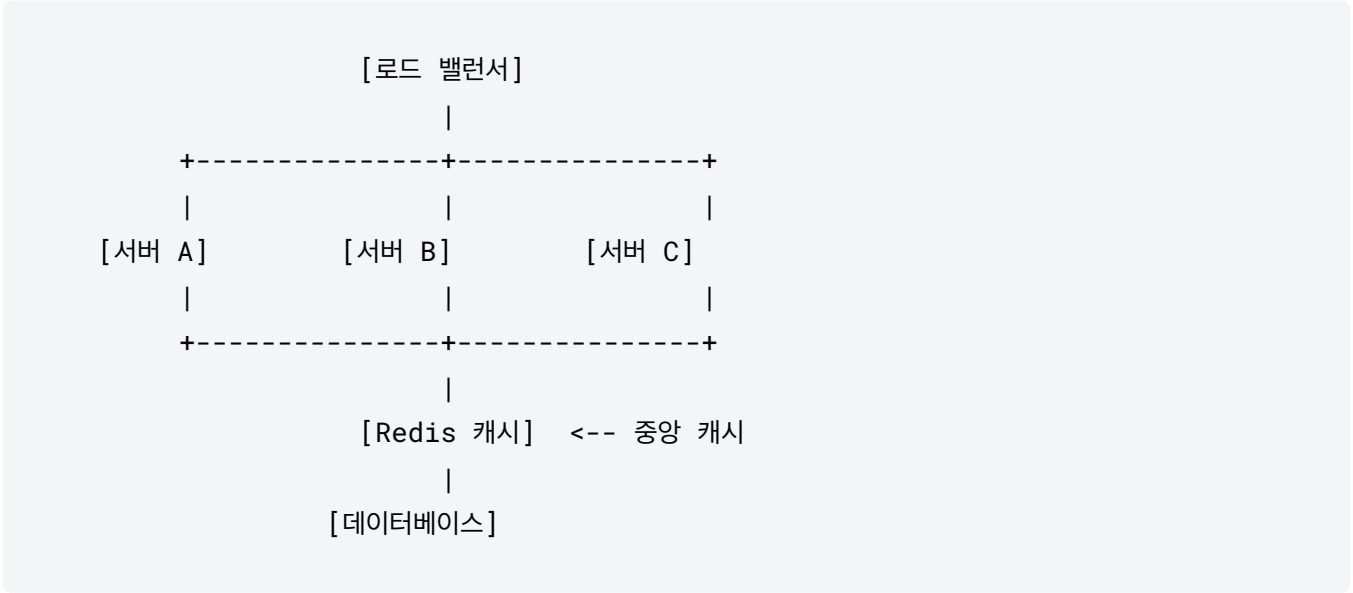
- 모든 서버 목록을 관리해야 한다
- 서버가 추가/제거될 때마다 목록을 업데이트해야 한다
- 일부 서버에 요청이 실패하면 어떻게 처리할 것인가?
- 서버가 수백 대라면 수백 번의 HTTP 요청이 필요하다

이렇게 직접 모든 서버에 캐시 갱신 요청을 보내는 방법은 완벽하게, 그리고 안전하게 처리하기 쉽지 않다.

해결책3: Redis 같은 중앙 캐시 사용

중앙에 Redis 같은 캐시 서버를 두고, 모든 애플리케이션 서버가 이 캐시를 공유하는 방법이 있다.

참고로 Redis를 단순하게 말하면 데이터를 메모리에 올려서 사용하는 캐시 용도의 응답이 매우 빠른 서버로 이해하면 된다.



관리자가 공통 코드를 수정하면:

1. 데이터베이스 업데이트
2. Redis 캐시 갱신 (한 번만)
3. 모든 서버가 Redis에서 새 값을 조회

동기화 문제가 깔끔하게 해결된다. 하지만...

Redis 캐시의 한계

Redis를 사용하면 동기화 문제는 해결되지만, 원래 문제가 다시 발생한다. Redis 조회도 **네트워크 호출**이기 때문이다.

```
[애플리케이션] ---(네트워크)---> [Redis] : 약 0.5ms
[애플리케이션] ---(네트워크)---> [MySQL] : 약 1~2ms
```

Redis가 MySQL보다 빠르긴 하지만, 둘 다 네트워크 호출이다. 공통 코드 조회가 빈번하게 발생하면 여전히 네트워크 비용이 누적된다.

결국 **트레이드오프**가 발생한다.

방식	장점	단점
로컬 메모리 캐시	매우 빠름 (네트워크 없음)	다중 서버 동기화 어려움
Redis 중앙 캐시	동기화 쉬움	네트워크 호출 필요

TTL을 활용한 최적의 전략

두 가지 장점을 모두 가져가는 방법이 있다. 바로 **로컬 캐시 + TTL(Time To Live)** 전략이다.

핵심 아이디어

1. 공통 코드를 로컬 메모리에 캐싱한다 (빠른 조회)
2. 캐시에 TTL(유효 기간)을 설정한다 (예: 1분)
3. TTL이 만료되면 데이터베이스에서 다시 로딩한다 (자동 동기화)

```
// 의사 코드 (Pseudo Code)
class CommonCodeCache {
    private Map<String, Map<String, String>> cache;
    private DateTime lastLoadTime;
    private int TTL_SECONDS = 60;  // 1분

    String getName(String groupCode, String code) {
        // TTL 만료 확인
        if (isExpired()) {
            refresh();  // 캐시 갱신
        }
        return cache[groupCode][code];
```

```
    }

    boolean isExpired() {
        return (now() - lastLoadTime) > TTL_SECONDS;
    }

    void refresh() {
        // DB에서 다시 로딩
        cache = loadFromDatabase();
        lastLoadTime = now();
    }
}
```

- 이 코드는 1분마다 자동으로 DB에서 공통 코드를 조회한다. 그리고 메모리의 공통 코드 데이터를 DB의 최신 데이터에 맞추어 갱신한다.

TTL 전략의 동작 방식

```
시간 00:00 - 서버 A 시작, 캐시 로딩 (ORDER: 주문접수)
시간 00:30 - 관리자가 DB 수정 (ORDER: 주문완료)
시간 00:30 - 서버 A 캐시: 여전히 '주문접수' (TTL 만료 전)
시간 01:00 - 서버 A TTL 만료, 캐시 갱신 (ORDER: 주문완료)
```

관리자가 수정하고 최대 1분 후에는 모든 서버에 변경 사항이 반영된다. 결과적으로 모든 서버가 1분에 1번씩만 공통 코드 값을 갱신하기 위해 데이터베이스를 조회하게 된다.

1분이면 충분한가?

운영 관점에서 1분은 충분히 수용 가능한 지연 시간이다.

생각해보자. 관리자가 공통 코드를 수정하는 상황은 언제인가?

- 새로운 결제 수단 추가
- 표시 이름 오타 수정
- 사용하지 않는 코드 비활성화

이런 작업은 급하게 처리해야 하는 경우가 거의 없다. 관리자가 1분 정도 기다리는 것은 전혀 문제가 되지 않는다.
이런 TTL 캐싱 전략은 공통 코드를 관리할 때 딱 맞는 전략이다.

만약 비즈니스적으로 1분도 기다릴 수 없다면? 그런 경우는 매우 드물지만, TTL을 10초나 30초로 줄일 수 있다. 또는

앞서 설명한 Redis 방식을 선택할 수 있다.

성능 효과 계산

TTL 캐싱의 성능 효과를 구체적인 숫자로 계산해보자.

시나리오

- 1초에 사용자 10명이 동시 접속
- 각 사용자가 1초에 평균 20개의 목록 항목 조회
- 각 항목마다 공통 코드 이름 조회 필요
- TTL: 60초 (1분)

캐싱 없이 매번 DB 조회하는 경우 (공통 코드 조회만)

```
1초당 공통 코드 조회: 10명 × 20개 = 200회
60초(1분)당 조회: 200회 × 60초 = 12,000회
```

1분에 12,000번의 네트워크 호출과 데이터베이스 쿼리가 실행된다.

TTL 캐싱을 적용한 경우 (공통 코드 조회만)

```
60초(1분)당 조회: 1회 (TTL 만료 시 1번만 DB 조회)
나머지 11,999회: 메모리에서 조회 (네트워크 호출 없음)
```

구분	DB 조회 횟수 (1분)	감소율
캐싱 전	12,000회	-
캐싱 후	1회	99.99% 감소

데이터베이스 부하가 12,000분의 1로 줄어든다. 네트워크 통신도 마찬가지로 12,000분의 1로 줄어든다.
물론 애플리케이션 서버가 10대라면 1분마다 각 서버별로 공통 코드의 갱신을 위해 총 10번 정도의 SQL이 실행될 수 있다.

트래픽이 더 많은 경우

대규모 서비스에서는 효과가 더욱 극적이다.

```
1초당 사용자: 1,000명
각 사용자당 조회: 20개
1초당 조회: 1,000 × 20 = 20,000회
60초당 조회: 20,000 × 60 = 1,200,000회 (120만 회)

캐싱 적용 후: 1회
```

1분에 120만 번의 DB 조회가 1번으로 줄어든다.

참고로 애플리케이션 시작 시 캐시를 미리 로딩해두면 첫 번째 요청부터 빠르게 응답할 수 있다.

> **실무이야기 - "바퀴를 다시 발명하지 마라 (Don't reinvent the wheel)"**
> "그럼 Map이랑 Timer 등을 써서 캐시를 직접 구현하면 되겠네요?" 라고 질문할 수 있다.
> 절대 안 된다. 캐시를 직접 구현하면 동시성 문제(Concurrency), 메모리 누수(Memory Leak) 등 골치 아픈 버그를 만날 수 있다.
> 실무에서는 각 언어나 프레임워크 별로 주로 사용하는 캐시 라이브러리가 있다.
> 검증된 프레임워크와 라이브러리를 사용해라. 코드 몇 줄이면 강력한 캐시 기능을 사용할 수 있다.

정리

공통 코드 캐싱 전략을 정리해보자.

공통 코드의 특성

- 데이터가 적다 (수백 건)
- 변경이 드물다 (일주일에 몇 번)
- 조회가 빈번하다 (초당 수백~수천 번)

캐싱 전략 비교

전략	속도	동기화	복잡도	추천
캐싱 없음	느림	항상 최신	낮음	X

로컬 캐시 (TTL 없음)	빠름	어려움	중간	X
Redis 중앙 캐시	중간	쉬움	높음	△
로컬 캐시 + TTL	빠름	자동	중간	**O**

로컬 캐시 + TTL의 장점

- 네트워크 호출 없이 메모리에서 즉시 조회
- TTL 만료 시 자동으로 데이터베이스와 동기화
- 다중 서버 환경에서도 복잡한 동기화 로직 불필요
- 구현이 상대적으로 간단

권장 TTL 값

- 일반적인 경우: 60초 (1분)
- 빠른 반영이 필요한 경우: 10~30초
- 변경이 거의 없는 경우: 300초 (5분) 이상

다음 시간에는 공통 코드와 애플리케이션 ENUM을 함께 사용하는 방법에 대해 알아보겠다.

공통 코드 vs 애플리케이션 ENUM 1

실무의 많은 개발자들이 공통 코드를 사용할지 아니면 애플리케이션 ENUM을 사용할지 고민한다. 이번 시간에는 이 고민을 해결해보자.

지금까지 공통 코드 테이블을 설계하고 캐싱을 적용하는 방법을 배웠다. 그런데 실무에서는 공통 코드만으로는 해결되지 않는 문제가 있다. 이번 시간에는 애플리케이션 ENUM과 공통 코드를 언제, 어떻게 사용해야 하는지 알아보겠다.

애플리케이션에서 공통 코드를 직접 사용하지 않는 경우

먼저 공통 코드가 단순히 표시 목적으로만 사용되는 경우를 살펴보자.

예시: 결제 수단

여기서 결제 수단 코드는 주로 화면에 표시하거나 목록을 보여줄 때 사용된다고 가정하자.

이미 이전에 데이터를 입력했으므로, 여기서는 데이터베이스에 실제로 INSERT 하지는 않겠다.

```
-- 공통 코드 데이터
INSERT INTO common_code_detail (group_code, code, name, sort_order) VALUES
('PAYMENT_METHOD', 'CARD', '신용카드', 1),
('PAYMENT_METHOD', 'BANK', '계좌이체', 2),
('PAYMENT_METHOD', 'MOBILE', '휴대폰결제', 3),
('PAYMENT_METHOD', 'VIRTUAL', '가상계좌', 4);
```

애플리케이션에서는 이 코드를 어떻게 사용할까?

```
// 의사 코드 (Pseudo Code)

// 1. 결제 수단 목록 화면에 표시
paymentMethods = CommonCodeCache.getList("PAYMENT_METHOD");
// 결과: [{code: 'CARD', name: '신용카드'}, {code: 'BANK', name: '계좌이체'}, ...]

// 2. 결제 내역에서 결제 수단 이름 표시
payment = db.query("SELECT * FROM payments WHERE payment_id = 1");
payment.methodName = CommonCodeCache.getName("PAYMENT_METHOD",
payment.paymentMethod);
// 결과: '신용카드'
```

이 경우 애플리케이션 코드에서 'CARD', 'BANK' 같은 코드값을 직접 다루지 않는다. 단순히 데이터베이스에 저장하고, 화면에 표시할 때 이름으로 변환할 뿐이다. **코드값에 따라 분기 처리를 하거나 특별한 로직을 수행하지 않는다.**

이런 경우에는 공통 코드 테이블만으로 충분하다.

애플리케이션에서 공통 코드를 직접 사용하는 경우

이번에는 코드값에 따라 애플리케이션 로직이 달라지는 경우를 살펴보자.

예시: 주문 상태

주문 상태는 단순한 표시 목적을 넘어서, 애플리케이션의 비즈니스 로직에 깊이 관여한다.

이미 이전에 데이터를 입력했으므로, 여기서는 데이터베이스에 실제로 INSERT 하지는 않겠다.

```
-- 주문 상태 코드
INSERT INTO common_code_detail (group_code, code, name, sort_order) VALUES
('ORDER_STATUS', 'ORDER', '주문접수', 1),
('ORDER_STATUS', 'PAID', '결제완료', 2),
('ORDER_STATUS', 'SHIPPING', '배송중', 3),
('ORDER_STATUS', 'DELIVERED', '배송완료', 4),
('ORDER_STATUS', 'CANCEL', '주문취소', 5);
```

애플리케이션에서 공통 코드 테이블로 관리되는 주문 상태를 어떻게 사용하는지 보자.

```
// 의사 코드 (Pseudo Code)

// 1. 주문 취소 가능 여부 확인
boolean canCancel(Order order) {
    // 주문접수, 결제완료 상태에서만 취소 가능
    if (order.status == "ORDER" || order.status == "PAID") {
        return true;
    }
    return false;
}

// 2. 배송 시작 처리
void startShipping(Order order) {
    // 결제완료 상태에서만 배송 시작 가능
    if (order.status != "PAID") {
        throw new Exception("결제완료 상태에서만 배송을 시작할 수 있습니다.");
    }
    order.status = "SHIPPING";
    db.update(order);
}

// 3. 환불 처리
void processRefund(Order order) {
    // 취소된 주문만 환불 가능
    if (order.status != "CANCEL") {
        throw new Exception("취소된 주문만 환불할 수 있습니다.");
    }
    // 환불 로직...
```

```
}

// 4. 주문 상태에 따른 버튼 표시
List<String> getAvailableActions(Order order) {
    List<String> actions = new ArrayList();

    if (order.status == "ORDER") {
        actions.add("결제하기");
        actions.add("주문취소");
    } else if (order.status == "PAID") {
        actions.add("배송조회");
        actions.add("주문취소");
    } else if (order.status == "SHIPPING") {
        actions.add("배송조회");
    } else if (order.status == "DELIVERED") {
        actions.add("리뷰작성");
        actions.add("재구매");
    } else if (order.status == "CANCEL") {
        actions.add("환불조회");
    }

    return actions;
}
```

코드 곳곳에 "ORDER", "PAID", "SHIPPING" 같은 문자열이 하드코딩되어 있다. 이것이 바로 **공통 코드를 애플리케이션에서 직접 사용하는 경우**다. **각각의 공통 코드에 따라 비즈니스 로직의 흐름이 달라진다.**

공통 코드만 적용한 방식의 단점

위 코드에는 심각한 문제가 있다. 하나씩 살펴보자.

문제1: 오타로 인한 버그

문자열을 직접 사용하면 오타가 발생할 수 있다.

```
// 오타 발생!
if (order.status == "SHIPING") {  // SHIPPING을 SHIPING으로 오타
    // 이 조건은 절대 true가 되지 않는다!
}
```

프로그래밍 컴파일러는 "SHIPING"이 오타인지 알 수 없다. 문법적으로는 올바른 문자열이기 때문이다. 이런 버그는 테스트에서도 발견하기 어렵고, 운영 환경에서 문제가 발생한 후에야 알게 되는 경우가 많다.

문제2: 코드 변경 시 추적 어려움

'ORDER' 코드를 'RECEIVED'로 변경하고 싶다면? 애플리케이션 전체에서 "ORDER" 문자열을 찾아서 변경해야 한다.

```
// 파일 A
if (order.status == "ORDER") { ... }

// 파일 B
order.status = "ORDER";

// 파일 C
return status.equals("ORDER");

// 파일 D
case "ORDER": ...
```

수십 개의 파일에 흩어진 "ORDER" 문자열을 모두 찾아서 변경해야 한다. 하나라도 빠뜨리면 버그가 발생한다. 더 심각한 문제는 단순 문자열 검색으로 "ORDER"를 찾으면 관계없는 코드까지 검색될 수 있다는 것이다.

```
// 이것도 검색 결과에 포함됨 (관계없는 코드)
String message = "Please ORDER your items";
int ORDER_COLUMN_INDEX = 3;
```

문제3: 자동 완성 지원 없음

IDE에서 코드를 작성할 때 문자열은 자동 완성이 지원되지 않는다.

```
// 어떤 상태값이 있는지 알 수 없다
if (order.status == "???") {
    // ORDER? ORDERED? ORDER_RECEIVED? ORDER_COMPLETE?
```

```
}
```

개발자는 공통 코드 테이블을 직접 조회하거나 문서를 찾아봐야 한다. 결과적으로 생산성이 떨어진다.

문제4: 타입 안전성 없음

문자열 타입은 아무 값이나 들어갈 수 있다.

```
// 잘못된 값도 컴파일 에러 없이 저장됨
order.status = "HELLO";  // 완전히 잘못된 값
order.status = "order";  // 대소문자 실수
order.status = "ORDER ";  // 공백 포함
```

이런 잘못된 값이 데이터베이스에 저장되면 나중에 심각한 문제를 일으킨다.

그럼 이런 문제를 어떻게 해결할 수 있을까?

공통 코드 vs 애플리케이션 ENUM 2

애플리케이션 ENUM 도입을 통한 문제 해결

이런 문제들을 해결하려면 애플리케이션에서 ENUM(열거형)을 사용해야 한다.

ENUM 정의

Java, Kotlin, TypeScript 등 대부분의 프로그래밍 언어는 ENUM을 지원한다.

```
// Java 예시
public enum OrderStatus {
    ORDER("주문접수"),
    PAID("결제완료"),
    SHIPPING("배송중"),
    DELIVERED("배송완료"),
    CANCEL("주문취소");
```

```
    private final String displayName;

    OrderStatus(String displayName) {
        this.displayName = displayName;
    }

    public String getDisplayName() {
        return displayName;
    }
}
```

- ENUM의 핵심은 "ORDER", "PAID" 같은 문자열이 아니라 `ORDER`, `PAID`와 같은 상수가 정의된다는 점이다. 그리고 앞으로 `OrderStatus`를 사용할 때 반드시 이곳에 정의해둔 상수 목록만 사용할 수 있다.
- 이곳에 정의되지 않은 `ORD`, `PA`, `REQUEST` 와 같은 잘못된 단어를 사용하면 컴파일 오류가 발생한다.

ENUM 적용 후 코드

앞서 작성한 코드를 ENUM을 사용하도록 변경해보자.

```
// 의사 코드 (Pseudo Code) - ENUM 적용

// 1. 주문 취소 가능 여부 확인
boolean canCancel(Order order) {
    if (order.status == OrderStatus.ORDER || order.status == OrderStatus.PAID)
{
        return true;
    }
    return false;
}

// 2. 배송 시작 처리
void startShipping(Order order) {
    if (order.status != OrderStatus.PAID) {
        throw new Exception("결제완료 상태에서만 배송을 시작할 수 있습니다.");
    }
    order.status = OrderStatus.SHIPPING;
    db.update(order);
}

// 3. 환불 처리
```

```
void processRefund(Order order) {
    if (order.status != OrderStatus.CANCEL) {
        throw new Exception("취소된 주문만 환불할 수 있습니다.");
    }
    // 환불 로직...
}

// 4. 주문 상태에 따른 버튼 표시
List<String> getAvailableActions(Order order) {
    switch (order.status) {
        case ORDER: // OrderStatus.ORDER
            return Arrays.asList("결제하기", "주문취소");
        case PAID: // OrderStatus.PAID
            return Arrays.asList("배송조회", "주문취소");
        case SHIPPING:
            return Arrays.asList("배송조회");
        case DELIVERED:
            return Arrays.asList("리뷰작성", "재구매");
        case CANCEL:
            return Arrays.asList("환불조회");
        default:
            return Collections.emptyList();
    }
}
```

ENUM의 장점

장점1: 오타 방지 (컴파일 타임 검증)

```
// 오타가 발생하면 컴파일 오류로 컴파일 자체가 안 됨 (개발 단계에서 발견 가능)
if (order.status == OrderStatus.SHIPING) {
    // 컴파일 에러: SHIPING은 OrderStatus에 정의되지 않음
}
```

장점2: 코드 변경 시 추적 용이

```
// 'ORDER'를 'RECEIVED'로 변경하려면?
// ENUM 정의만 변경하면 모든 사용처에서 컴파일 에러 발생
// -> 변경이 필요한 곳을 정확히 파악 가능
```

장점3: IDE 자동 완성 지원

```
// IDE에서 OrderStatus. 을 입력하면 자동 완성 목록이 표시됨
if (order.status == OrderStatus.[자동완성목록])
// 자동 완성: ORDER, PAID, SHIPPING, DELIVERED, CANCEL
```

장점4: 타입 안전성

```
// 잘못된 값을 넣으면 컴파일 에러
order.status = "HELLO";  // 컴파일 에러! String을 OrderStatus에 할당할 수 없음
```

장점5: 표시 이름도 함께 관리 가능

```
// ENUM에서 바로 표시 이름 조회
String name = OrderStatus.PAID.getDisplayName();  // "결제완료"

// 목록 조회도 가능
List<OrderStatus> allStatuses = Arrays.asList(OrderStatus.values());
```

- `getDisplayName()` 은 앞서 정의한 메서드(함수)
- `OrderStatus.values()` 는 자바가 기본 제공하는 메서드(함수) - 모든 ENUM을 리스트(목록, 컬렉션)으로 제공한다.

ENUM과 데이터베이스 연동

ENUM 값을 데이터베이스에 저장할 때는 보통 코드값 문자열로 저장한다.

```
// 저장 시: ENUM -> 문자열
String statusCode = order.status.name();  // "PAID" 문자열 반환
db.execute("INSERT INTO orders (status) VALUES (?)", statusCode);

// 조회 시: 문자열 -> ENUM
String statusCode = rs.getString("status");  // "PAID" 문자열 조회
OrderStatus status = OrderStatus.valueOf(statusCode);  // OrderStatus.PAID
```

- `name()` 은 자바가 기본 제공하는 메서드(함수) - ENUM을 문자열로 변환한다.
  - PAID ➜ "PAID"

- `valueOf()`는 자바가 기본 제공하는 메서드(함수) - 문자열을 ENUM으로 변환한다.

참고로 대부분의 ORM(JPA, TypeORM 등)에서는 이 변환을 자동으로 처리해준다.

애플리케이션 ENUM만 사용할 때의 단점

공통 코드를 데이터베이스 공통 코드 테이블 대신에 ENUM으로 관리하면 장점이 많아 보이지만, ENUM만 사용하면 또 다른 문제가 발생한다.

문제1: 표시 이름 변경 시 재배포 필요

기획팀에서 '주문접수'를 '주문완료'로 변경해달라고 요청한다.

```
// ENUM 정의 수정
public enum OrderStatus {
    ORDER("주문완료"),   // "주문접수" -> "주문완료" 변경
    ...
}
```

애플리케이션 코드를 수정했으니 애플리케이션 소스 코드를 다시 빌드하고 배포해야 한다. 단순히 화면에 표시되는 이름 하나를 바꾸는데 전체 배포가 필요하다.
데이터베이스 테이블로 공통 코드를 관리할 때는 해당 컬럼의 값만 수정하면 끝난다.

만약 배포에 10분이 걸린다면? 단순한 텍스트 수정에 10분을 기다려야 한다. 운영 유연성이 떨어진다.

문제2: 추가 속성 관리의 어려움

회원 등급별로 할인율이 다르다고 가정하자.

```
public enum MemberGrade {
    NORMAL("일반회원", 0),
    VIP("VIP회원", 5),
    VVIP("VVIP회원", 10);

    private final String displayName;
    private final int discountRate;
```

```
    // 생성자, getter 생략
}
```

기획팀에서 VIP 할인율을 5%에서 7%로 변경해달라고 한다. 또 재배포가 필요하다.

실무에서는 이런 속성 변경 요청이 빈번하게 발생한다. 매번 재배포하는 것은 비효율적이다.
결과적으로 개발자도 운영자도 모두 힘들어진다.

공통 코드 vs 애플리케이션 ENUM 선택 방법

지금까지 공통 코드 테이블 방식과 애플리케이션 ENUM 방식의 장단점을 살펴보았다. 이 두 방식은 **'개발의 안전성'**과 **'운영의 유연성'** 사이의 트레이드오프(Trade-off) 관계에 있다.

장단점 비교

두 방식의 특징을 한눈에 비교해보자.

구분	공통 코드 테이블 (DB)	애플리케이션 ENUM (Java)
주요 용도	단순 목록 조회, 화면 표시	비즈니스 로직 분기 처리
코드 안전성	낮음 (문자열 사용, 오타 위험)	**높음 (컴파일 체크, 타입 안전)**
변경 유연성	**높음 (DB만 수정하면 반영)**	낮음 (코드 수정 및 재배포 필요)
자동 완성	지원 안 함	IDE 지원 강력함
관리 주체	데이터베이스 관리자 / 개발자 / 운영자	개발자

그래서 언제 무엇을 써야 할까?

실무에서 이 두 가지를 결정하는 명확한 기준을 제시하겠다. 이 기준만 따르면 대부분의 고민을 해결할 수 있다.

1. 애플리케이션 로직(if문)에 사용되는가? → ENUM 사용

`if (status == "PAYMENT")` 와 같이 코드 내에서 분기 처리가 필요하거나 비즈니스 로직의 기준이 되는 데이터라면 반드시 **ENUM**을 사용해라.

- **이유**: 오타 방지, 자동 완성, 리팩토링 용이성 등 코드의 품질과 유지보수성을 위해 필수적이다. 개발자가 실수를 줄일 수 있는 가장 확실한 방법이다.

- **예시**: 주문 상태(접수/결제/배송), 회원 등급(일반/VIP), 결제 수단(카드/현금) 등

2. 단순한 목록(Dropdown)으로만 보여주는가? ➜ 공통 코드 테이블 사용
로직에는 전혀 관여하지 않고, 단순히 사용자에게 선택지(Dropdown)를 제공하거나 화면에 텍스트를 보여주기 위한 용도라면 **공통 코드 테이블**을 사용해라.

- **이유**: 이런 데이터는 비즈니스 로직보다는 운영 설정에 가깝다. 항목이 추가되거나 이름이 변경되어도 코드를 다시 배포하지 않고 DB만 수정해서 즉시 반영할 수 있어야 한다.
- **예시**: 지역 코드(서울/경기/부산), 은행 코드(국민/신한/우리), 취미 목록, 가입 경로 등

3. 변경이 매우 빈번한가? ➜ 공통 코드 테이블 사용
로직에 일부 사용되더라도 하루에도 몇 번씩 바뀔 수 있는 데이터라면 ENUM으로 관리하기 어렵다. 이런 경우엔 코드의 안전성을 일부 포기하더라도 운영의 유연성을 위해 공통 코드 테이블을 선택해야 한다.

남은 문제점

여기까지 설명을 들었다면 한 가지 아쉬움이 남을 것이다.

> "주문 상태는 비즈니스 로직에서도 매우 중요하고(ENUM 필요), 화면에 보여지는 이름이나 속성도 자주 변경될 수 있는데(DB 필요), 둘 다 만족할 수는 없을까?"

맞다. 우리는 **ENUM의 타입 안전성**도 챙기면서, **DB의 변경 유연성**도 함께 가져가고 싶다. 실무에서는 이 두 마리 토끼를 잡기 위해 **두 방식을 혼합해서 사용**하기도 한다.

다음 시간에는 이 두 가지 방식의 장점을 결합한 하이브리드 전략에 대해 알아보겠다.

공통 코드 vs 애플리케이션 ENUM 3

공통 코드와 애플리케이션 ENUM을 함께 사용하는 하이브리드 전략

두 가지 방식의 장점을 모두 가져가려면 **하이브리드 전략**을 고려할 수 있다.
하이브리드 전략은 ENUM과 공통 코드 테이블 두 곳 모두에 데이터를 **중복으로 유지**하고 **각자 역할을 나누어 분담**하는 것이다.

하이브리드 전략의 핵심 아이디어

- **ENUM**: 코드값 정의, 타입 안전성, 비즈니스 로직
- **공통 코드**: 표시 이름, 추가 속성, 운영 유연성

```
[애플리케이션 ENUM]
- 코드값 정의 (ORDER, PAID, SHIPPING, ...)
- 비즈니스 로직에서 사용
- 컴파일 타임 검증

[공통 코드 테이블]
- 표시 이름 (주문접수, 결제완료, ...)
- 추가 속성 (할인율, 적립률, ...)
- 관리자가 직접 수정 가능
```

구현 예시

Step 1: ENUM 정의 (코드값만)

```java
// Java 예시
public enum OrderStatus {
    ORDER,
    PAID,
    SHIPPING,
    DELIVERED,
    CANCEL;

    // 표시 이름과 속성은 ENUM에 정의하지 않음! 대신에 공통 코드 테이블에서 조회
}
```

Step 2: 공통 코드 테이블 (표시 이름, 속성)

```sql
-- 표시 이름과 속성은 공통 코드 테이블에서 관리
INSERT INTO common_code_detail (group_code, code, name, name_en, sort_order)
VALUES
('ORDER_STATUS', 'ORDER', '주문접수', 'Order Received', 1),
('ORDER_STATUS', 'PAID', '결제완료', 'Payment Complete', 2),
('ORDER_STATUS', 'SHIPPING', '배송중', 'Shipping', 3),
('ORDER_STATUS', 'DELIVERED', '배송완료', 'Delivered', 4),
```

```
('ORDER_STATUS', 'CANCEL', '주문취소', 'Cancelled', 5);
```

- 여기서는 데이터베이스에 실제로 INSERT 하지는 않겠다.

Step 1 , Step 2를 보자.
하이브리드 전략은 애플리케이션 ENUM과 데이터베이스 공통 코드 테이블 두 곳 모두에 데이터를 중복으로 보관한다.

Step 3: 비즈니스 로직에서는 ENUM 사용

```
// 비즈니스 로직 - ENUM 사용 (타입 안전)
boolean canCancel(Order order) {
    return order.status == OrderStatus.ORDER ||
           order.status == OrderStatus.PAID;
}

void startShipping(Order order) {
    if (order.status != OrderStatus.PAID) {
        throw new Exception("결제완료 상태에서만 배송을 시작할 수 있습니다.");
    }
    order.status = OrderStatus.SHIPPING;
}
```

Step 4: 표시 이름은 DB나 캐시의 공통 코드에서 조회

```
// 화면 표시 - 공통 코드 사용 (운영 유연성)
String getStatusDisplayName(OrderStatus status) { // ENUM PAID 입력 가정
    return CommonCodeCache.getName("ORDER_STATUS", status.name()); // "결제완료"
문자 반환
}
```

- ENUM `PAID`를 입력했다고 가정하면 여기서는 데이터베이스의 공통 코드 테이블을 조회한 다음에 "결제완료" 문자를 반환한다.
- `status.name()`을 호출하면 ENUM을 문자열로 변환해서 반환한다.
  - `PAID` ➜ "PAID"

역할 분담

구분	ENUM	공통 코드

코드값 정의	O (동일하게 유지)	O (동일하게 유지)
비즈니스 로직	O	X
타입 안전성	O	X
표시 이름	X	O
추가 속성	X	O
운영 중 변경	X (재배포 필요)	O (즉시 반영)

하이브리드 전략은 ENUM과 공통 코드 두 곳 모두에 데이터를 중복으로 유지하고 각자 역할을 나누어 분담하는 것이다.

하이브리드 전략의 운영 규칙

하이브리드 전략을 효과적으로 운영하려면 몇 가지 규칙이 필요하다.

규칙1: 코드 추가는 개발자가 담당

새로운 상태 코드를 추가하는 것은 비즈니스 로직 변경을 의미한다. 따라서 개발자가 ENUM과 공통 코드를 함께 추가해야 한다.

```
// 1. ENUM에 새 코드 추가
public enum OrderStatus {
    ORDER,
    PAID,
    SHIPPING,
    DELIVERED,
    CANCEL,
    RETURN_REQUEST,  // 새로 추가: 반품요청
    RETURN_COMPLETE; // 새로 추가: 반품완료
}

// 2. DB의 공통 코드 테이블에도 추가, 여기에 이름과 속성도 추가
INSERT INTO common_code_detail (group_code, code, name, sort_order) VALUES
('ORDER_STATUS', 'RETURN_REQUEST', '반품요청', 6),
('ORDER_STATUS', 'RETURN_COMPLETE', '반품완료', 7);
```

```
// 3. 관련 비즈니스 로직 구현
void requestReturn(Order order) {
    if (order.status != OrderStatus.DELIVERED) {
        throw new Exception("배송완료 상태에서만 반품 요청할 수 있습니다.");
    }
    order.status = OrderStatus.RETURN_REQUEST;
}
```

규칙2: 속성 변경은 운영자가 담당

표시 이름이나 추가 속성의 변경은 코드 변경 없이 운영자가 직접 수행할 수 있다.

```
-- 표시 이름 변경: 개발자 개입 없이 즉시 반영
UPDATE common_code_detail
SET name = '주문완료'
WHERE group_code = 'ORDER_STATUS' AND code = 'ORDER';

-- 할인율 변경: 개발자 개입 없이 즉시 반영
UPDATE common_code_detail
SET attr1 = '7'
WHERE group_code = 'MEMBER_GRADE' AND code = 'VIP';
```

물론 실무에서는 SQL을 직접 다루는 것이 아니라, 어드민 운영툴을 통해서 처리한다.

규칙3: ENUM과 공통 코드의 동기화

ENUM에 정의된 코드와 공통 코드 테이블의 코드가 항상 일치해야 한다. 이를 보장하기 위해 검증용 테스트 케이스를 작성하는 것이 안전하다.

하이브리드 전략의 단점과 보완 방법

하이브리드 전략에도 단점이 있다.

단점1: 두 곳에서 관리해야 함

ENUM과 공통 코드 테이블 두 곳에서 코드를 관리해야 한다. 새로운 코드를 추가할 때 한 쪽만 추가하고 다른 쪽을 빠뜨리면 문제가 발생한다.

보완 방법: 테스트 케이스로 동기화 검증

```
// 의사 코드 (Pseudo Code)
@Test
void ENUM과_공통코드_동기화_검증() {
    // 1. ENUM에서 모든 코드값 조회
    Set<String> enumCodes = Arrays.stream(OrderStatus.values())
        .map(e -> e.name())
        .collect(toSet());

    // 2. 공통 코드 테이블에서 모든 코드값 조회
    Set<String> dbCodes = db.query(
        "SELECT code FROM common_code_detail WHERE group_code =
'ORDER_STATUS'"
    ).stream().map(r -> r.code).collect(toSet());

    // 3. 두 집합이 일치하는지 검증
    assertEquals(enumCodes, dbCodes,
        "ENUM과 공통 코드 테이블의 코드가 일치해야 합니다.");
}
```

이러한 테스트를 CI/CD 파이프라인에 포함시키면, 배포 전에 동기화 문제를 발견할 수 있다.

더 강력한 검증: 애플리케이션 시작 시 검증

하이브리드 전략을 더 확실하게 검증하고 싶다면 애플리케이션 시작 시점에 검증하는 방법도 고려할 수 있다.
만약 애플리케이션 시작시 검증에 예외가 발생하면 신규 배포를 중지하는 것이다.

정리

공통 코드와 애플리케이션 ENUM, 그리고 하이브리드 전략까지 긴 내용을 다루었다. 실무에서 어떤 방식을 선택해야 할지 다음 3가지 기준으로 깔끔하게 정리해보자. 이 기준만 따르면 복잡한 고민 없이 명확한 설계를 할 수 있다.

1. 화면 표시 목적만 있는 경우

- **상황**: 애플리케이션의 비즈니스 로직(`if`문 분기 처리)에는 전혀 관여하지 않고, 단순히 사용자에게 선택지(Dropdown)를 제공하거나 화면에 텍스트를 보여주기 위한 용도다.
- **예시**: 은행 코드(국민/신한/우리), 지역 코드(서울/경기/부산), 가입 경로, 취미 목록 등
    - 여러분의 서비스에서는 은행 코드, 지역 코드가 비즈니스 로직에 사용될 수 도 있다. 이 경우 다음 전략들을 선택해야 한다.

- **해결책**: **공통 코드 테이블만 사용해라.**
- **이유**: 이런 데이터는 비즈니스 로직보다는 운영 설정에 가깝다. 항목이 추가되거나 이름이 변경되어도 개발자가 코드를 수정하고 배포할 필요 없이, 운영자가 데이터베이스만 수정해서 즉시 반영할 수 있어야 한다. **운영의 유연성**이 최우선이다.

2. 비즈니스 로직에서만 사용하는 경우

- **상황**: 비즈니스 로직의 핵심 기준이 되어 코드에서 `if`문으로 제어해야 한다. 화면에 표시되는 이름이 중요하지 않거나 변경될 일이 거의 없다. 하지만 코드 내부에서는 엄격한 분기 처리가 필요하고 오타가 발생하면 치명적인 버그로 이어진다.
- **예시**: 내부 시스템 제어 플래그, 고정된 시스템 상수, 프로그래밍 언어 차원의 타입 구분 등
  - 주문 상태(접수/결제/배송), 회원 등급(일반/VIP/VVIP), 결제 수단 등
  - 속성을 변경할 일이 없거나, 아주 가끔 변경하는 경우
- **해결책**: **애플리케이션 ENUM만 사용해라.**
- **이유**: 데이터베이스를 거치지 않고 메모리 상에서 빠르게 처리할 수 있다. 컴파일 시점의 타입 체크와 IDE의 자동 완성 지원이 필수적이다. **코드의 안전성**이 최우선이다. 추가로 개발자 친화적이고, 코드를 깔끔하게 유지할 수 있다.

3. 하이브리드 상황이 필요한 경우 (로직 + 표시)

- **상황**: 비즈니스 로직의 핵심 기준이 되어 코드에서 `if`문으로 제어해야 하고(안전성 필요), 동시에 화면에 보여지는 이름이나 할인율 같은 속성은 운영 상황에 따라 유연하게 변경되어야 한다(유연성 필요). 실무에서 가장 중요한 데이터들이 여기에 해당한다.
- **예시**: 주문 상태(접수/결제/배송), 회원 등급(일반/VIP/VVIP), 결제 수단 등
- **해결책**: **하이브리드 전략(ENUM + 공통 코드)을 사용해라.**
- **이유**: ENUM으로 **개발의 안전성**을 챙기고, 공통 코드 테이블로 **운영의 유연성**을 동시에 확보할 수 있다. 두 마리 토끼를 다 잡는 방법이다. 단, ENUM과 DB 데이터의 정합성을 맞추기 위해 동기화 검증 테스트를 반드시 작성해야 한다는 점을 명심하자.

3가지 패턴 비교

먼저 우리가 배운 3가지 방식의 장단점을 한눈에 비교해보자.

구분	1. 공통 코드만 사용	2. ENUM만 사용	3. 하이브리드 전략
주요 용도	단순 목록 표시, 정적 데이터	내부 로직 제어, 타입 안전성	로직 제어 + 유연한 운영
타입 안전성	X (문자열 직접 사용)	O (컴파일 타임 검증)	O (컴파일 타임 검증)

운영 유연성	O (DB 수정 시 즉시 반영)	X (코드 수정 및 배포 필요)	O (표시 이름/속성 즉시 반영)
구현 복잡도	낮음	중간	높음 (동기화 관리 필요)
추천 대상	은행 코드, 국가 코드	주문 상태, 결제 수단, 회원 등급	주문 상태, 결제 수단, 회원 등급

실무 선택 가이드 (Decision Tree)

개발을 하다가 "이걸 ENUM으로 해야 하나, 공통 코드로 해야 하나?" 고민이 든다면 다음 질문을 따라가자.

질문 1. 해당 코드가 비즈니스 로직(if문)에 사용되는가?

- **아니요 (NO)**
  - 단순히 화면에 셀렉트 박스(Select Box)나 리스트로 보여주는 것이 전부라면 **[1. 공통 코드 테이블]**만 사용해라.
  - 예: 은행 목록, 직업 목록, 가입 경로 등
- **예 (YES)**
  - 코드 값에 따라 비즈니스 로직이 분기되거나, 특정 상태를 검증해야 한다면 **[질문 2]**로 넘어간다.

질문 2. 표시 이름이나 추가 속성이 빈번하게 변경되는가?

- **아니요 (NO)**
  - 로직에는 사용되지만, 이름이 바뀌거나 속성이 변할 일이 거의 없다면 **[2. 애플리케이션 ENUM]**만 사용해라. 가장 깔끔하고 개발 생산성이 높다.
  - 예: 성별(남/여), 배송비 정책(무료/유료 - 단순 구분 시)
- **예 (YES)**
  - 로직에도 깊이 관여하고, 기획팀이나 운영팀에서 이름이나 할인율 같은 속성을 자주 변경 요청할 것 같다면 **[3. 하이브리드 전략]**을 선택해라.
  - 예: 주문 상태(이름 변경 잦음), 회원 등급(할인율 변경 잦음), 쿠폰 종류

실무 선택 팁

실무에서는 결국 ENUM, 공통 코드 테이블 두 가지를 함께 사용하게 된다. 어떤 코드들은 ENUM으로, 어떤 코드들은 공통 코드 테이블로 사용될 것이다. 그리고 어떤 것은 하이브리드 전략이 필요할 것이다.

여기서 처음부터 모든 것을 하이브리드 전략으로 가져갈 필요는 없다. **오버 엔지니어링(Over Engineering)**이 될 수

있기 때문이다.

1. **시작은 단순하게:** 애플리케이션 코드에서 비즈니스 로직에 필요하면 우선 **ENUM**으로 시작해라. 개발자에게는 이 방법이 코드 관리 관점에서 가장 실용적이고 편하다.
2. **필요할 때 확장:** 운영을 하다 보니 텍스트 변경 요청이 너무 잦거나, 관리자 페이지에서 속성을 제어해야 할 필요가 생기면 그때 **하이브리드 전략**으로 리팩토링해라.
3. **동기화는 필수:** 하이브리드 전략을 쓸 때는 반드시 **테스트 케이스**를 통해 ENUM과 DB의 코드가 일치하는지 검증하는 안전장치를 마련해라.

이 기준을 가지고 설계를 시작한다면, 유지보수하기 좋고 운영하기도 편한 단단한 시스템을 만들 수 있을 것이다.

이것으로 공통 코드 설계에 대한 내용을 마친다. 공통 코드는 단순해 보이지만, 잘 설계하면 시스템 전체의 유지보수성과 운영 효율성을 크게 높일 수 있다.

공통 코드 설계와 비즈니스 설계의 차이

공통 코드 테이블을 설계할 때와 일반적인 비즈니스 테이블을 설계할 때의 차이에 대해서 알아보자.

공통 코드에 대리키를 사용하지 않는 이유

우리는 앞서 `member`, `orders` 테이블을 설계할 때 `member_id`, `order_id` 같은 대리키(Surrogate Key, `BIGINT AUTO_INCREMENT`)를 기본 키로 사용했다. 시스템 내부에서 관리하는 의미 없는 숫자를 키로 사용하는 것이 비즈니스 키 변경 등 여러 변수에 대처하기 좋기 때문이다.

그런데 왜 공통 코드 테이블은 `code`라는 문자열(자연키, Natural Key)을 복합 키로 구성해서 기본 키로 사용했을까? 여기에는 실무적인 이유가 있다.

1. 데이터의 가독성

만약 공통 코드 테이블에도 대리키를 적용한다면 구조가 다음과 같이 변한다.

- `common_code_detail` 테이블 PK: `code_id` (1, 2, 3...)
- `orders` 테이블 컬럼: `order_status_id` (1, 2, 3...)

이 상태에서 주문 테이블을 조회하면 어떻게 될지 예상해보자.

```sql
-- 대리키를 사용했을 때의 조회 예시 (가상)
SELECT order_id, order_status_id FROM orders;
```

[실행 결과]

order_id	order_status_id
1	10
2	15
3	20

`10`, `15`라는 숫자만 봐서는 이것이 '주문완료'인지 '배송중'인지 직관적으로 알 수 없다. 반드시 공통 코드 테이블과 조인을 해야만 의미를 파악할 수 있다. 개발 과정에서 DB를 직접 조회해서 데이터를 검증하거나 디버깅할 때 매우 불편해진다.

반면, 문자열 코드를 사용하면 조인 없이도 데이터의 의미를 바로 파악할 수 있다.

```sql
SELECT order_id, order_status FROM orders;
```

[실행 결과]

order_id	order_status
1	ORDER
2	SHIPPING
3	CANCEL

이처럼 코드성 데이터는 그 자체로 의미를 전달하는 경우가 많기 때문에, 식별 가능한 짧은 문자열 코드를 사용하는 것이 유지보수에 유리하다.

2. 애플리케이션 매핑의 용이성

애플리케이션(Java, Python 등) 코드에서도 이 값을 상수로 관리한다.

```
// Java Enum 예시
public enum OrderStatus {
    ORDER, SHIPPING, CANCEL
}
```

DB에 문자열로 저장되어 있으면 애플리케이션의 Enum 이름과 DB의 값이 일치하므로 매핑이 직관적이다. 하지만 DB에 숫자로 저장되어 있다면 `10 = ORDER`라는 별도의 매핑 로직이 필요하고, 실수로 매핑이 잘못되면 엉뚱한 상태로 처리될 위험이 있다.

정리

일반적인 비즈니스 테이블(회원, 주문 등)은 **대리키(`BIGINT AUTO_INCREMENT`)** 사용을 적극 권장한다. 하지만 변경이 거의 없고, 참조용으로 주로 사용되며, 그 자체로 의미가 명확한 코드성 테이블의 경우 **자연키(문자열 코드)**를 사용하는 것이 실무적으로 더 효율적이다.

공통 코드 외래 키를 사용하지 않는 이유

보통의 경우 데이터의 무결성을 위해 외래 키(Foreign Key) 사용을 권장한다. 앞서 설계한 `orders` 테이블의 `member_id`가 대표적이다. 하지만 공통 코드를 사용할 때는 실무에서 외래 키를 거의 걸지 않는다. 여기에는 구조적인 이유와 실용적인 이유가 있다.

1. 복합 키(Composite Key) 구조의 문제

우리가 설계한 `common_code_detail` 테이블의 기본 키(PK)는 `group_code`와 `code`를 합친 **복합 키**다.

```
PRIMARY KEY (group_code, code)
```

관계형 데이터베이스에서 외래 키를 걸려면 참조하는 테이블의 기본 키 구성을 그대로 따라야 한다. 즉, `orders` 테이블에서 주문 상태 코드에 외래 키를 걸려면 다음과 같이 `group_code` 컬럼을 추가해야 한다.

```
-- 외래 키를 걸기 위해 불필요한 group_code 컬럼이 필요해진다
CREATE TABLE orders (
    ...
    order_status_group VARCHAR(50) DEFAULT 'ORDER_STATUS', -- 불필요한 컬럼
```

```
    order_status_code  VARCHAR(50),
    ...
    FOREIGN KEY (order_status_group, order_status_code)
        REFERENCES common_code_detail (group_code, code)
);
```

모든 주문 데이터마다 `ORDER_STATUS`라는 똑같은 그룹 코드를 저장하는 것은 저장 공간 낭비이며, 설계가 지저분해진다.

2. 성능과 유연성

주문 테이블처럼 데이터가 대량으로 쌓이는 트랜잭션 테이블에 외래 키를 걸면, 데이터를 입력(INSERT)할 때마다 공통 코드 테이블을 조회해서 값이 맞는지 확인하는 과정이 발생한다. 이는 미세하지만 성능 저하의 원인이 된다.

또한, 공통 코드는 애플리케이션 레벨(Enum 등)에서 이미 검증된 데이터만 들어오도록 통제하는 것이 일반적이다. 따라서 데이터베이스 레벨에서 외래 키로 한 번 더 강제하지 않아도 데이터 정합성이 크게 훼손되지 않는다.

결론

공통 코드 테이블은 참조용 메타 데이터의 성격이 강하다. 따라서 구조적인 복잡함과 성능 비용을 감수하면서까지 외래 키를 강제하기보다는, 애플리케이션 로직으로 정합성을 관리하고 DB에서는 느슨한 관계를 유지하는 것이 실무적으로 더 유리하다.

정리

공통 코드가 필요한 이유

- **데이터 불일치 방지**: 상태값을 문자열로 직접 저장하면 오타나 표기법 차이로 데이터 정합성이 깨진다.
- **변경 용이성**: 표시 이름이 변경될 때 데이터를 일일이 수정하지 않고 코드 매핑 정보만 수정하면 된다.
- **다국어 및 중앙 관리**: 코드값과 표시 이름을 분리하여 다국어 지원이 쉽고 시스템 전체 코드를 한눈에 파악할 수 있다.

공통 코드 테이블 설계

- **단순 설계**: 코드(PK)와 이름으로 구성되지만, 서로 다른 도메인(예: 주문, 결제)에서 같은 코드값을 사용할 때 충돌이 발생한다.
- **해결책**: 코드를 그룹으로 관리할 필요가 있다.

공통 코드를 더 범용성 있게 - 그룹화 설계

- **그룹 코드와 상세 코드 분리**: 가장 널리 사용되는 방식이다.
- **그룹 코드 테이블**: `group_code`(PK), `group_name` 등을 관리하여 코드의 분류를 정의한다.
- **상세 코드 테이블**: `group_code` + `code`를 복합 키(PK)로 사용하여 그룹 내에서 유일성을 보장한다.
- **장점**: 서로 다른 그룹에서 동일한 코드값(예: CANCEL)을 사용할 수 있다.

공통 코드와 추가 속성

- **컬럼 추가 방식**: `attr1`, `attr2` 같은 범용 컬럼을 추가하고 그룹 테이블에 해당 컬럼의 의미를 명시한다. 실무적으로 가장 간단하고 효율적이다.
- **EAV 방식**: 속성이 매우 가변적일 때 사용하지만 조회 성능 저하와 쿼리 복잡도가 증가하므로 신중해야 한다. EAV 방식은 뒤에서 설명한다.

공통 코드의 단점

- **SQL 복잡도 증가**: 코드 이름을 가져오기 위해 공통 코드 테이블을 반복적으로 조인해야 한다(조인 지옥).
- **중복 로직**: 코드 이름을 조회하는 조인 쿼리가 애플리케이션 전반에 중복된다.
- **불필요한 데이터 전송**: 코드값만 필요한 경우에도 이름까지 함께 조회되는 비효율이 발생한다.

공통 코드의 단점 해결 방안1

- **애플리케이션에서 매핑**: SQL에서는 코드값만 조회하고, 애플리케이션 메모리에 코드 정보를 로딩해두고 이름을 변환한다.
- **장점**: SQL이 단순해지고 매핑 로직을 중앙화할 수 있다.
- **단점**: 매 요청마다 코드 정보를 DB에서 조회하면 네트워크 비용이 발생하고 N+1 문제가 생길 수 있다.

공통 코드의 단점 해결 방안2

- **캐싱 활용**: 공통 코드는 데이터 양이 적고, 변경이 드물며, 조회는 빈번하므로 캐싱에 최적화된 데이터다.
- **로컬 캐시 + TTL 전략**: 애플리케이션 메모리에 코드를 캐싱하고, TTL(예: 60초)을 설정하여 주기적으로 DB와 동기화한다.
- **효과**: 네트워크 비용을 제거하고 DB 부하를 획기적으로 줄이면서 데이터 정합성도 어느 정도 유지할 수 있다.

공통 코드 vs 애플리케이션 ENUM 1

- **단순 표시용**: 로직 관여 없이 목록 표시가 목적이라면 공통 코드 테이블만 사용하는 것이 좋다.
- **로직 제어용**: `if` 문 등 비즈니스 로직에서 분기 처리가 필요하다면 문자열(공통 코드) 직접 사용은 오타 및 타입 안전성 문제가 있어 위험하다.

공통 코드 vs 애플리케이션 ENUM 2

- **ENUM 도입**: 컴파일 타임에 오타를 방지하고 타입 안전성을 보장하며 IDE의 지원을 받을 수 있다.

- **ENUM 단점**: 표시 이름이나 속성이 변경될 때마다 애플리케이션을 재배포해야 해서 운영 유연성이 떨어진다.

공통 코드 vs 애플리케이션 ENUM 3

- **하이브리드 전략**: 개발의 안전성과 운영의 유연성을 모두 확보하는 방법이다.
  - **ENUM**: 코드값 정의 및 비즈니스 로직 제어 담당 (타입 안전성).
  - **공통 코드(DB)**: 표시 이름 및 추가 속성 관리 담당 (운영 유연성).
- **운영 규칙**: 코드는 개발자가 추가하고, 속성은 운영자가 변경한다. ENUM과 DB 간의 데이터 동기화를 검증하는 테스트가 필수적이다.
- **선택 기준**
  1. 단순 표시 ➔ **공통 코드**
  2. 로직 사용 + 변경 없음 ➔ **ENUM**
  3. 로직 사용 + 잦은 변경 ➔ **하이브리드**

공통 코드 설계와 비즈니스 설계의 차이

- **자연키(String) 사용**: 공통 코드는 그 자체로 의미가 명확하고 애플리케이션 상수와 매핑하기 좋으므로 대리키(Auto Increment) 대신 문자열 코드를 PK로 사용한다.
- **외래 키 미사용**: 복합 키 구조로 인해 참조 테이블에 불필요한 컬럼(`group_code`)이 추가되어야 하고, 성능 저하 우려가 있어 실무에서는 외래 키 제약조건을 이 경우에는 잘 걸지 않는다.