CP4. 부호화와 발전

Part 1. 데이터 시스템의 기초

4. 부호화와 발전

▼ 개요

- 대부분의 경우 애플리케이션 기능을 변경하려면 저장하는 데이터도 변경해야 한다. 아마도 새로운 필드나 레코드 유형을 저장해야 하거나 기존 데이터를 새로운 방법으로 제공해야 할지 모른다.
    - 관계형 DB는 일반적으로 DB의 모든 데이터가 하나의 스키마를 따른다.
      스키마가 변경될 수 있지만 특정 시점에는 정확하게 하나의 스키마가 적용된다.
      반면 읽기 스키마 DB는 스키마를 강요하지 않으므로 다른 시점에 쓰여진 이전 데이터 타입과 새로운 데이터 타입이 섞여 포함될 수 있다.
    - 데이터 타입이나 스키마가 변경될 때 애플리케이션 코드에 대한 변경이 종종 발생한다. (ex. 레코드에 새로운 필드가 추가되면 애플리케이션 코드는 해당 필드의 읽고 쓰기를 시작한다.)
      하지만 대규모 애플리케이션에서 코드 변경은 대개 즉시 반영할 수 없다.
        - 서버 측 애플리케이션에서는 한 번에 몇 개의 노드에 새 버전을 배포하고 새로운 버전이 원할하게 실행되는지 확인한 다음 서서히 모든 노드에 실행되게 하는 **순회식 업그레이드(단계적 롤아웃)** 방식이 있다. 순회식 업그레이드는 서비스 정지 시간 없이 새로운 버전을 배포 할 수 있기 때문에 더욱 자주 출시할 수 있다. (좋은 발전성)
        - 클라이언트 측 애플리케이션은 사용자에 전적으로 좌우된다. (업데이트를 안 할 수도 있음)
    - 위 두 가지 내용은 새로운 버전과 이전 버전의 시스템이 동시에 공존할 수 있다는 의미이다.
      시스템이 원활하게 실행되려면 양방향 호환성을 유지해야 한다.
        - **하위 호환성** (새로운 코드는 예전 코드가 기록한 데이터를 읽을 수 있어야 한다.)
        - **상위 호환성** (예전 코드는 새로운 코드가 기록한 데이터를 읽을 수 있어야 한다.)
          하위 호환성 보다 다루기 어려움
        - **호환성**은 데이터를 부호화하는 하나의 프로세스와 그것을 복호화하는 다른 프로세스 간의 관계이다.
    - JSON, XML, 프로토콜 버퍼(Protocol Buffers), 스리프트, 아브로 등 데이터 부호화를 위한 다양한 형식을 확인한다.
        - 어떻게 스키마를 변경하고, old, new 버전의 데이터와 코드가 공존하는 시스템을 어떻게 지원하는지 체크한다.

        - rest, remote procedure call 뿐 아니라 actor 와 메시지 큐 같은 메시지 전달 시스템에서 다양한 데이터 부호화 형식이 데이터 저장과 통신에 어떻게 사용되는지 체크한다.

▼ 데이터 부호화 형식

- 프로그램은 (최소) 두 가지 형태로 표현된 데이터를 사용해 동작한다.
    - 메모리에 객체, 구조체, 목록, 해쉬 테이블 등 데이터가 유지 된다. (이런 데이터 구조는 CPU 에서 효율적으로 접근 조작할 수 있게 최적화(보통 포인터(다른 프로세스가 이해 할 수 없음)) 된다.)
    - 데이터를 파일에 쓰거나 네트워크를 전송하려면 스스로를 포함한 일련 바이트열(ex. JSON) 의 형태로 부호화 해야한다. (다른 프로세스가 이해할 수 있어야 함)
    - 부호화 (직렬화, 마샬링) - 인메모리 표현에서 바이트열로 전환
    - 복호화 (파싱, 역직렬화, 언마샬링) - 바이트열에서 인메모리 표현으로 전환
- 언어별 형식
    - 프로그래밍 내장된 부호화 라이브러리는 최소한의 추가 코드로 인메모리 객체를 저장하고 복원할 수 있어 편리하지만, 심각한 문제점 또한 많다. (이팩티브 자바에서는 새로 개발 한다면 JSON, gRPC protobuf 사용하라고 권장)
        - 부호화는 보통 특정 프로그래밍 언어와 묶여 있어 다른 언어에서 데이터를 읽기는 매우 어렵다.
          이런 부호화로 데이터를 저장하고 전송하는 경우 매우 오랜 시간이 될지도 모를 기간 동안 현재 프로그래밍 언어로만 코드를 작성해야 할 뿐 아니라 다른 시스템과 통합하는데 방해된다.
        - 동일한 객체 유형의 데이터를 복원하려면 복호화 과정이 임의의 클래스를 인스턴스화할 수 있어야 한다.
          이것은 종종 보안 문제의 원인이 된다. 공격자가 임의의 바이트열을 복호화할 수 있는 애플리케이션을 얻을 수 있으면 임의의 클래스를 인스턴스화 할 수 있고 공격자가 원격으로 임의 코드를 실행하는 것과 같은 끔직한 일이 발생할 수 있다.
        - 데이터 버전 관리는 보통 부호화 라이브러리에서는 나중에 생각하게 됨.
          데이터를 빠르고 쉽게 부호화하기 위해 상위, 하위 호환성의 불편한 문제가 등한시되곤 한다.
        - 효율성 (부호화, 복호화 시간, 부호화된 크기)도 나중에 생각하게 됨
- JSON, XML, CSV 이진 변형
    - JSON, XML, CSV 부호화
        - 결점
            - 수의 부호화, XML, CSV 에서는 수와 숫자로 구성된 문자열을 구분할 수 없다. JSON 은 문자열과 수를 구분하지만 정수와 부동소수점 수를 구별하지 않고 정밀도도 지정하지 않음. (큰 수를 다룰 때 문제가 일어난다.)
            - JSON 과 XML은 유니코드 문자열(사람이 읽을 수 있는)을 잘 지원한다. 그러나 이진 문자열(문자열에 비해 성능이 훨 뛰어남) 을 지원하지 않는다.
            - 필수는 아니지만 XML, JSON 모두 스키마를 지원한다. 두 부호화 스키마 언어는 상당히 강력하지만 구현하기 난해하다.
              CSV는 스키마가 없으므로 각 로우와 컬럼의 의미를 정의하는 작업이 필요하다.
        - 이러한 결점에도 JSON, XML, CSV는 다양한 용도에 사용하기 충분하다. (특히 데이터 교환 형식)
- 이진 부호화
    - 큰 데이터 셋인 경우 JSON 같은 부호화 형식일 경우 좋지 못하다. (이진 형식과 비교하면 JSON 같은 부호화는 더 많은 공간을 사용함)

- JSON 형태는 데이터 안에 속성 값도 포함해야 한다.

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

- 메시지 팩 형태 (JSON 전용 이진 부호화 형식)

```
83 a8 75 73 65 72 4e 61 6d 65 a6 4d 61 72 74 69 6e ae ... cd 05 39 .

83 (객체 항목 3)
a8 (문자열 길이 8)
75 73 65 72 4e 61 6d 65 (u s e r N a m e)
a6 (문자열 길이 6)
4d 61 72 74 69 6e (M a r t i n)
ae (문자열 길이 14)
...
cd (부호 없는 16비트 정수)
05 39 (1337)
...
```

  - 첫 번째 바이트 0x83 은 세 개 필드를 가진 객체를 뜻
    두 번째 바이트 0xa8은 이어지는 내용이 8btye 길이의 문자열 뜻
    다음 8바이트는 userName 의 ASCII
    다음 7바이트는 0xa6 문자열 길이 6개 나머지 6개 Martin ASCII

- 스리프트(apache thrift)와 프로토콜 버퍼(protocol buffers) (protobuf)
  - 둘 다 이진화 라이브러리, 모두 부호화할 데이터를 위한 스키마 정의가 필요함
  - 스리프트 스키마 정의

```
struct Person {
    1: required string       userName,
    2: optional i64          favoriteNumber,
    3: optional list<string> interests
}
```

  - 프로토콜 버퍼 스키마 정의

```
message Person {
    required string user_name      = 1;
    optional int64 favorite_number = 2;
    repeated string interests      = 3;
}
```

  - 스키마 정의 도구 (파일)로 구현한 클래스를 생성한다.
    애플리케이션 코드는 생성된 코드를 호출해 스키마의 레코드를 부호화하고 복호화할 수 있다.

- required, optional 에 존재는 required 를 사용하면, 필드가 설정되지 않은 경우를 실행 시에 확인할 수 있다. (버그 잡을 때 유용)
- 이진 부호화 형식
  - 스리프트 **바이너리 프로토콜**

```
0b 00 01 00 00 00 06 4d 61 72 74 69 6e 0a 00 02 00 00 00 00 00 00

0b 타입 11 (문자열)
00 01 (필드 태그 = 1)
00 00 00 06 (길이 6)
4d 61 72 74 69 6e (Martin)
0a 타입 10 (int 64)
00 02 (필드 태그 = 2)
00 00 00 00 00 00 05 39 (1337)
0f 타입 15 (리스트)
00 03 (필드 태그 = 3)
0b 타입 11 (문자열)
...
00 구성의 끝
```

    - 메시지 팩과 다르게 필드 이름이 없는 대신, 부호화된 숫자인 필드 태그를 포함한다.
  - 스리프트 **컴팩트 프로토콜**

```
18 06 4d 61 72 74 69 6e 16 f2 14 19 28 0b ... 00

18 (0001 1000 필드 태그 1, 타입 8(문자열))
06 (길이 6)
4d 61 72 74 69 6e (Martin)
16 (0001 0110 필드 태그 +=1, 타입 6(Int64))
f2 14 (1111 0010 0001 0100) - 0010100 111001 (1337)
19 (0001 1011 필드 태그 +=1, 타입 9(리스트))
28 (0010 1000 목록 항목 2, 타입 8 (문자열))
ob (길이 11)
...
00 구성의 끝
```

    - 필드 타입과 태그 숫자를 단일 바이트로 줄이고 가변 길이 정수를 사용해 부호화한다.
  - **프로토콜 버퍼**

```
0a 06 4d 61 72 74 69 6e 10 b9 0a 1a 0b 64 61 79 64 72 65 61 ...

0a (00001 010 필드 태그 1, 타입 2(문자열))
06 (길이 6)
4d 61 72 74 69 6e (Martin)
10 (00010 000 필드 태그 2, 타입 0(가변길이 정수)
b9 0a (10111001 00001010) - 00010100 111001
```

```
1a 0b (00011 010 필드 태그 3, 타입 2 (문자열))
64 61 79 64 72 65 61 (daydrea...)
...
```

- 필드 태그와 스키마 발전
  - **스키마 발전** - 스키마는 필연적으로 시간이 지남에 따라 변함.
  - 스리프트, 프로토콜 버퍼는 하위 호환성과 상위 호환성을 유지하면서 스키마를 변경하는 법
    - 부호화된 레코드는 부호화된 필드의 연결일 뿐이다.
      각 필드는 태그 숫자로 식별하고 데이터 타입을 주석으로 단다.
      필드 값을 설정하지 않은 경우는 부호화 레코드에서 생략한다.
      부호화된 데이터는 필드 이름을 전혀 참조하지 않기 때문에
      스키마에서 필드 이름은 변경할 수 있다
    - 그러나, **필드 태그는 기존의 모든 부호화된 데이터를 인식 불가능하게 만들 수 있기 때문에 변경할 수 없다.**
      필드에 새로운 태그 번호를 부여하는 방식으로 스키마에 새로운 필드를 추가할 수 있다.
      예전 코드에서 새로운 코드로 기록한 데이터를 읽으려는 경우에는 해당 필드를 간단히 무시할 수 있다.
      데이터타입 주석은 파서가 몇 바이트를 건너뛸 수 있는지 알려준다.
      이는 상위 호환성을 유지하게 한다.
    - 각 필드에 고유한 태그 번호가 있는 동안에는 태그 번호가 계속 같은 의미를 가지고 있기 때문에 새로운 코드가 예전 데이터를 항상 읽을 수 있다.
      사소한 문제 하나로 새로운 필드를 추가한 경우 이 필드는 required 로 할 수 없다.
      새로운 필드를 required 로 추가한 경우 예전 코드는 추가한 새로운 필드를 기록하지 않기 때문에 새로운 코드가 예전 코드로 기록한 데이터를 읽는 작업은 실패한다.
      그러므로 하위 호환성을 갖으려면, 스키마 초기 배포 후에 추가되는 모든 필드는 optional 로 하거나 기본값이 있어야 한다.
    - **필드 삭제**는 optional 필드만 삭제 할 수 있다.
  - 데이터 타입과 스키마 발전
    - **필드의 데이터 타입을 변경하는 것**은 값이 정확하지 않거나 잘릴 위험이 있다.
    - 프로토콜 버퍼는 단일 값을 다중 값으로 변경해도 된다.
      - 프로토콜 버퍼에는 목록이나 배열 데이터타입이 없지만 대신 필드에 repeated 표시자가 있다.
      - repeated 필드의 부호화는 레코드에 단순히 동일한 필드 태그가 여러 번 나탄다.
      - optional 필드를 repeated 필드로 변경해도 문제가 없다는 것이다.
        - 이전 데이터를 읽는 새로운 코드는 0 이나 1개의 엘리먼트가 있는 목록으로 보게 되고, 새로운 데이터를 읽는 예전 코드는 목록의 마지막 엘리먼트만 보게된다.
    - 스리프트는 전용 목록 데이터 타입이 있다.
      - 목록 엘리먼트의 데이터타입을 매개변수로 받는다. 중첩된 목록을 지원한다.
- 아브로
  - 정의
    - 아브로도 부호화할 데이터 구조를 지정하기 위해 스키마를 사용한다.

- 사람이 편집 할 수 있는 아브로 IDL, 기계가 더 쉽게 읽을 수 있는 JSON 기반 두 개 스키마 언어가 있다.

```
# 아브로 IDL
record Person {
    string               userName;
    union {null, long } favoriteNumber = null;
    array<string>       interests;
}

# JSON 기반
{
    "type": "record",
    "name": "Person",
    "fields": [
        {"name": "userName",       "type": "string"},
        {"name": "favoriteNumber", "type": ["null", "long"], "defa
        {"name": "interests",       "type": {"type": "array", "ite
    ]
}
```

  - 스키마에 태그 번호가 없다.

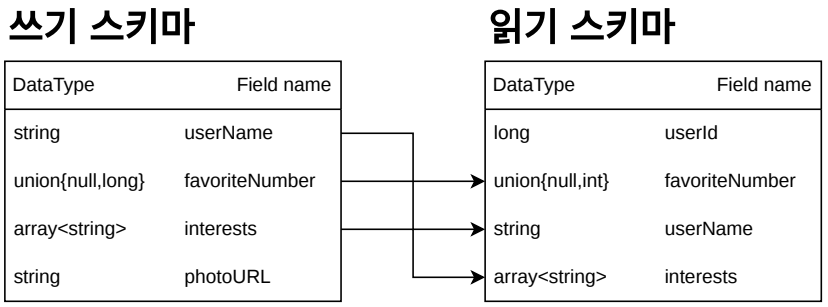
```
0c 4d 61 72 74 69 6e 02 f2 14 04 16 64 61 79 64 72 65...

0c (00001100) 길이 6
4d 61 72 74 69 6e (Martin)
02 (00000010) 유니온 분기 1 (long, null 아님)
f2 14 (1111 0010 0001 0100) - 0010100 111001 (1337)
04 (00000100) 2개 배열 항목 이어짐
16 (00010110) 길이 11
64 61 79 64 72 65 (daydream...)
```

  - 아브로 이진 부호화 길이가 스리프트, 프로토 버퍼에 비해서 더 짧다.
  - 데이터 타입을 식별하기 위한 정보가 없다.
  - 부호화는 단순히 연결된 값으로 구성

- 아브로를 이용해 이진 데이터를 파싱하려면 스키마에 나타난 순서대로 필드를 살펴보고 스키마를 이용해 각 필드의 데이터 타입을 미리 파악해야 한다.
  - 데이터를 읽는 코드가 데이터를 기록한 코드와 정확히 같은 스키마 일때만 이진 데이터를 올바르게 복호화할 수 있다.
- 아브로 읽기 스키마, 쓰기 스키마
  - 쓰기 스키마
    - 애플리케이션이 파일이나 DB에 쓰기 위해 또는 네트워크를 통해 전송 등의 목적으로 어떤 데이터를 아브로로 부호화하길 원한다면 알고 있는 스키마 버전을 사용해 데이터를 부호화 한다.
  - 읽기 스키마

- 애플리케이션이 파일이나 DB에서 또는 네트워크로 부터 수신 등으로 읽은 어떤 데이터를 복호화하길 원한다면 특정 스키마로 복호화하길 기대한다.

- 핵심은 쓰기 스키마와 읽기 스키마가 동일하지 않아도 되며 단지 호환 가능하면 된다는 뜻이다. 데이터를 복호화할 때 쓰기 스키마와 읽기 스키마를 함께 살펴본 다음 다음 쓰기 스키마에서 읽기 스키마로 데이터를 변환해 그 차이를 해소한다.



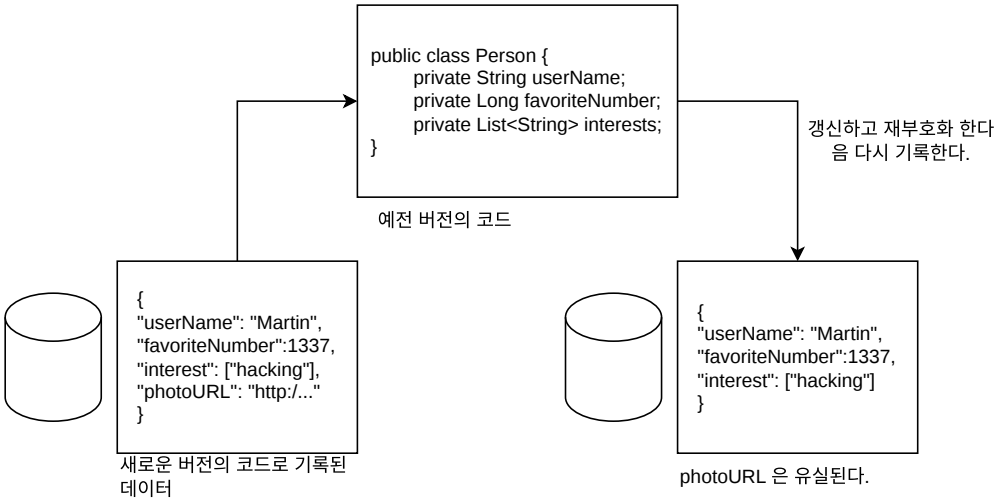
- 읽기 스키마와 쓰기 스키마는 필드 순서가 달라도 문제 없다.
  스키마 해석에서는 이름으로 필드를 일치시키기 때문이다.
  읽기 스키마에는 없고 쓰기 스키마에 존재하는 필드는 무시된다.
  쓰기 스키마에는 없고 읽기 스키마에 존재하는 필드는 default 값으로 채워진다.

- 스키마 발전 규칙
  - 상위 호환성 새로운 버전의 쓰기 스키마와 예전 버전의 읽기 스키마를 갖을 수 있다.
  - 하위 호환성 새로운 버전의 읽기 스키마와 예전 버전의 쓰기 스키마를 갖을 수 있다.
  - 호환성을 유지하기 위해서는 default 값이 있는 필드만 추가하거나 삭제할 수 있다.
  - 프로토콜 버퍼, 스리프트와 동일한 방식의 optional 과 required 표시자가 없다.
    대신, 필드의 null 을 허용하려면
    유니온 타입(union type)을 사용해야 한다.
  - 타입이 정의되지 않아 필드의 데이터 타입 변경이 가능하다.
  - 필드 이름도 변경가능 하지만, 하위 호환성만 만족하고, 상위 호환성은 만족 못한다.
- 읽기는 특정 데이터를 부호화한 쓰기 스키마를 알 수 있는 방법 (해결법)
  - 많은 레코드가 있는 대용량 파일
    - 아브로의 일반적인 용도는 모두 동일한 스키마로 부호화된 수백만 개 레코드를 포함한 큰 파일을 저장하는 용도다. 이 경우 파일의 쓰기는 파일의 시작 부분에 한 번만 쓰기 스키마를 포함하면 된다. (파일 형식, 객체 컨테이너 파일)
  - 개별적으로 기록된 레코드를 가진 DB
    - DB의 다양한 레코드들은 다양한 쓰기 스키마를 사용해 서로 다른 시점에 쓰여질 수 있다. 즉 모든 레코드가 동일한 스키마를 가진다고 가정할 수 없다.
      간단한 해결책으로 모든 부호화된 레코드의 시작 부분에 버전 번호를 포함하고 DB에는 스키마 버전 목록을 유지한다. 읽기는 레코드를 가져와 버전 번호를 추출한 다음 DB에서 비전 번호에 해당하는 쓰기 스키마를 가져온다. 가져온 쓰기 스키마를 사용해 남은 레코드를 복호화할 수 있다.
    - 스키마 버전들이 설명서 처럼 동작해 호환성 체크를 직접 할 수 있다.
  - 네트워크 연결을 통해 레코드 내보내기

        - 두 프로세스가 양방향 네트워크 연결을 통해 통신할 때 연결 설정에서 스키마 버전 합의를 할 수있다. 이 후 연결을 유지하는 동안 합의된 스키마를 사용한다. 아브로 RPC 프로토콜이 이처럼 동작한다.
    - 동적 생성 스키마
        - 아브로 스키마는 태그 번호가 포함돼 있지 않아, 동적 생성 스키마에 유리하다.
        - 관계형 스키마로부터 아브로 스키마를 상당히 쉽게 생성할 수 있다.
          이 스키마를 이용해 DB 내용을 부호화하고 아브로 객체 컨테이너 파일로 모두 덤프 할 수 있다. 각 DB 테이블에 맞게 레코드 스키마를 생성하고 각 칼럼은 해당 레코드의 필드가 된다. DB 의 컬럼 이름은 아브로의 필드 이름에 매핑된다.
        - 이에 반해 스리프트나 프로토콜 버퍼를 이런 용도로 사용한다면 필드 태그를 수동으로 할당해야만 한다. 즉 DB 스키마를 변경할 때마다 관리자는 DB 칼럼 이름과 필드 태그의 매핑을 수동으로 갱신해야 한다.
    - 코드 생성과 동적 타입 언어
        - 스리프트와 프로토콜 버퍼는 코드 생성에 의존한다.
            - 스키마를 정의한 후 선택한 프로그래밍 언어로 스키마를 구현한 코드를 생성할 수 있다.
                - 정적 타입 언어에 유용하다, 복호화된 데이터를 위해 효율적인 인메모리 구조를 사용하고 데이터 구조에 접근하는 프로그램 작성 시 IDE 타입 확인, 자동 완성이 가능해짐)
                - 동적 타입 프로그래밍 언어에서는 만족시킬 컴파일 시점의 타입 검사기가 없기 때문에 코드를 생성하는 것이 중요하지 않다.
        - 아브로는 정적 타입 프로그래밍 언어를 위해 코드 생성을 선택적으로 제공한다.
          하지만 코드 생성 없이도 사용할 수 있다.
            - (쓰기 스키마를 포함한) 객체 컨테이너 파일이 있다면 아브로 라이브러리를 사용해 간단히 열어 JSON 파일을 보는 것과 같이 데이터를 볼 수 있다. 이 파일은 필요한 메타데이터 를 모두 포함하기 때문에 **자기 기술(self-descibing) 적**이다.
    - 스키마 장점
        - 프로토콜 버퍼와 스리프트, 아브로는 스키마를 사용해 이진 부호화 형식을 기술한다.
          이 스키마 언어는 XML, JSON 스키마 보다 간단하며 더 자세한 유효성 검사 규칙을 지원한다.
        - 많은 데이터 시스템이 이진 부호화를 독자적으로 구현하기도 한다.
          ex) 대부분의 관계형 DB는 질의를 DB로 보내고 응답을 받을 수 있는 네트워크 프로토콜이 있다. 이 프로토콜은 일반적으로 특정 DB에 특화되고 DB 벤더는 DB 네트워크 프로토콜로부터 응답을 인메모리 데이터 구조로 복호화하는드라이버(OJBC, JDBC)를 제공한다.
        - 이진 부호화 장점
            - 부호화된 데이터에서 필드 이름을 생략할 수 있기 때문에 이진 JSON 변형 보다 크기가 훨씬 작아질 수 있다.
            - 스키마는 유용한 문서화 형식이다. 복호화할 때 스키마가 필요하기 때문에 스키마가 최신 상태인지를 확신할 수 잇다.
            - 스키마 DB를 유지하면 스키마 변경이 적용되기 전에 상위 호환성과 하위 호환성을 확인할 수 있다.
            - 정적 타입 프로그래밍 언어 사용자에게 스키마로부터 코드를 생성하는 기능이 유용하다. (컴파일 체크 가능 해짐)

▼ 데이터플로 모드

- 데이터플로는 매우 추상적인 개념으로, 하나의 프로세스에서 다른 프로세스로 데이터를 전달하는 방법이다.
  데이터베이스를 통한, 서비스 호출을 통한, 비동기 메시지 전달을 통한 방식을 알아본다.
- **DB를 통한 데이터플로**
  - DB에 기록하는 프로세스는 데이터를 부호화하고 DB에서 읽는 프로세스는 데이터를 복호화한다.
  - 하위 호환성이 필요하다.
    이전 기록된 내용을 미래에 보려면 복호화를 할 수 있어야 하기 때문
  - 상위 호환성도 대게 필요하다.
    새로운 버전의 코드를 기록된 다음 현재 수행 중인 예전 버전의 코드로 새로운 버전으로 만들어진 데이터를 볼 수도 있기 때문
  - 레코드 스키마에 필드에 추가하고 새로운 코드는 새로운 필드를 위한 값을 DB에 기록하고 예전 버전의 코드가 레코드를 읽고 갱신한 후 갱신한 값을 다시 기록한다고 하면, 예전 코드가 레코드를 해석할 수 없더라도 새로운 필드를 그대로 유지하는게 바람직하다.
    이러한 방식의 알지 못하는 (unknown) 필드 보존은 지원하지만, 아래 그림처럼 애플리케이션 차원에서 신경 써야하는 경우도 있다.
    즉, 애플리케이션에서 DB 값을 모델 객체로 복호화하고 나중에 이 모델 객체를 다시 재부호화 한다면 변환 과정에서 알지 못하는 필드가 유실 될 수도 있다.



  - 다양한 시점에 기록된 다양한 값
    - DB 데이터는 5년 전 기록된 레코드일지, 5초 전 기록된 레코드 일지 모른다.
    - 마이그레이션 (데이터를 새로운 스키마로 다시 기록 하는 작업)은 가능하다.
      하지만 대용량 데이터 셋 대상으로는 값비싼 작업임으로 대부분의 DB에서 이런 상황을 피한다.
    - 대부분의 RDB는 기존 데이터를 다시 기록하지 않고 널을 기본값으로 갖는 새로운 칼럼을 추가하는 간단한 스키마 변경을 허용한다.
    - 스키마 발전은 기본 저장소가 여러 가지 버전의 스키마로 부호화된 레코드를 포함해도 전체 DB가 단일 스키마로 부호화된 것처럼 보이게 한다.
  - 보관 저장소
    - 데이터 덤프는 일관되게 부호화 하는 것이 좋다. (여러 시점의 스키마 버전이 섞여 있다고 하더라도)

        - 데이터 덤프는 한 번에 기록하고 이후에는 변하지 않으므로 아브로 객체 컨테이너 파일과 같은 형식에도 적합하다.

- **서비스를 통한 데이터플로 (REST 와 RPC)**
    - 개요
        - 서비스를 통한 데이터의 경우 미리 정해진 입력과 출력만 허용하는 API 를 공개해서 서버쪽 캡슐화를 가능하게 한다.
        - 네트워크를 통신해야 하는 프로세스가 있는 경우에 일반적인 방법은 클라이언트와 서버 두 역할로 배치한다.
            - 웹의 경우 (자바 스크립트 애플리케이션) XMLHttpRequest 를 사용해 HTTP 클라이언트가 될 수 있다. (AJAX 라고도 함)
            - 하나의 서비스가 다른 서비스의 일부 기능이나 데이터 필요하면 해당 서비스에 요청하는 것을 (서비스 지향 설계 (SOA), 최근에는 마이크로서비스 설계 (MSA)) 라고 도 부른다.
                - MSA 는 예전 버전과 새로운 버전의 서버와 클라이언트가 동시에 실행되기를 기대함으로, **서버와 클라이언트가 사용하는 데이터 부호화는 서비스 API의 버전 간 호환이 가능해야 한다.**
    - 웹 서비스
        - 서비스와 통신시 기본 프로토콜로 HTTP를 사용하면 웹 서비스라 한다.
          하지만 이는 웹뿐만 아니라 다른 상황에서 적용됨으로 약간 잘못된 표현이다.
            - 사용자 디바이스에서 실행하며 HTTP를 통해 서비스에 요청하는 클라이언트 애플리케이션 (javascript 웹 앱)
            - 서비스 지향/마이크로서비스 아키텍처로 데이터 센터에 위치한 다른 서비스에 요청하는 서비스 (미들웨어)
            - 보통 인터넷을 통해 다른 조직의 서비스에 요청하는 서비스, 다른 조직간에 백엔드 시스템 간 데이터 교환을 위해 사용함 (OAauth)
        - 통신하는 대중적인 방법으로 REST, SOAP 가 있다.
            - REST
                - 프로토콜이 아닌 HTTP의 원칙을 토대로 한 설계 철학
                - 간단한 데이터 타입을 강조하며 URL 을 통해 리소스 식별, 캐시 제어, 인증, 콘텐츠 유형에 HTTP 기능을 사용한다.
                - 조직 간 서비스 통합과 관련해서는 SOAP 보다 인기가 있다.
                - 마이크로서비스와 연관되기도 한다.
                    - REST API 는 stateless 를 설계해 jwt 토큰을 이용함
                - 공개 API 의 주요한 방식
            - SOAP
                - 네트워크 API 요청을 위한 XML 기반 프로토콜
                - HTTP 상에서 가장 일반적으로 사용되지만 HTTP와 독립적이며 대부분의 HTTP 기능을 사용하지 않는다.

        - 그 대신, 다양한 기능을 추가하고 광범위하고 복잡한 여러 관련 표준(웹 서비스 프레임워크(ws))을 제공한다.
        - SOAP 웹 서비스의 API는 웹 서비스 기술 언어 또는 WSDL 이라고 부르는 XML 기반 언어를 사용해 기술한다.
            - WSDL은 클라이언트가 (XML 메시지로 부호화하고 프레임워크가 다시 복호화하는) 로컬 클래스와 메서드 호출을 사용해 원격 서비스에 접근하는 코드 생성이 가능하다.
            - WSLD은 사람이 읽을 수 있게 설계하지 않았고 대게 SOAP 메시지를 수동으로 구성하기에는 복잡하기 때문에 SOAP 사용자는 도구 지원과 코드 생성과 IDE 에 의존한다. SOAP 벤더가 지원하지 않는 프로그래밍 언어 사용자의 경우 SOAP 서비스 통합은 어렵다.
    - 원격 프로시저 호출(RPC) 문제
        - 웹 서비스는 네트워크 상에서 API 요청하기 위한 여러 기술 중에 가장 최신 형상일 뿐이다. 서비스에 많은 부분이 과장됐고 여러 문제가 있다.
            - 엔터프라이즈 자바빈(EJB) 과 자바 원격 메서드 호출(RMI)은 자바 언어로 제한된다.
            - 분산 컴포넌트 객체 모델(DCOM)은 마이크로 소프트 플랫폼으로 제한된다.
        - 이러한 문제들은 원격 프로시저 호출 (RPC)의 기반으로 생성되었다.
        - RPC 모델은 원격 네트워크 서비스 요청을 같은 프로세스 안에서 특정 프로그래밍 언어의 함수나 메서드를 호출하는 것과 동일하게 사용 가능하게 해준다. (**위치 투명성**) 하지만, RPC 접근 방식은 근본적으로 결함이 있다. (네트워크 요청은 로컬 함수 호출과 매우 다름)
            - 로컬 함수 호출은 예측이 가능하다. 제어 가능한 매개변수에 따라 성공하거나 실패한다.
            - 네트워크 요청은 예측이 어렵다. 네트워크 문제로 요청과 응답이 유실되거나 원격 장비가 느려지거나 요청에 응답하지 않을 수 있다. (제어 할 수 없다)
            - 네트워크 문제
                - 로컬 함수 호출은 결과를 반환하거나 예외를 내거나 반환하지 않을 수 있다. 네트워크 요청은 또 다른 결과가 가능하다. 네트워크 요청은 타임아웃(timeout)으로 결과 없이 반환될 수 있다. (원격 서비스로부터 응답을 받지 못한다면 요청을 제대로 보냈는지 아닌지를 알 수 있는 방법이 없다.)
                - 실패한 네트워크 요청을 다시 시도할 때 요청이 실제로는 처리되고 응답만 유실될 수 있다. 이 경우 프로토콜에 중복 제거 기법(**멱등성**)을 적용하지 않으면 재시도는 작업이 여러 번 수행되는 원인이 된다. 로컬 함수는 문제 없다.
                - 로컬 함수를 호출할 때마다 보통 거의 같은 시간이 소요된다. 네트워크 요청은 함수 호출보다 훨씬 느리고 지연 시간은 매우 다양하다. 빠른 경우 1ms 도 가능하지만 네트워크가 혼잡하면 같은 작업하는데도 수 초가 걸릴 수 있다.
                - 로컬 함수를 호출하는 경우 참조를 로컬 메모리의 객체에 효율적으로 전달할 수 있다. 네트워크로 요청하는 경우에는 모든 매개변수는 네트워크를 통해 전송할 수 있게 끔 바이트 열로 부호화해야 한다. (매개 변수가 원시형 타입이면 괜찮지만, 큰 객체라면 문제될 수 있다.)
                - 클라이언트와 서비스는 다른 프로그래밍 언어로 구현할 수 있다. 따라서 RPC 프레임워크는 핮나의 언어에서 다른 언어로 데이터 타입 변환해야 한다. 모든 언어가 같은 타입을 가지는 것은 아니기 때문에 깔끔하지 않은 모습이 될 수 있다. (java long 2^63, js 2^53)
    - RPC 의 현재 방향

        - 이런 문제에도 RPC 사라지지 않는다.
          ex) 스리프트, 아브로 RPC 지원 기능을 내장하고 gRPC 는 프로토콜 버퍼를 이용한 RPC 구현이다. Rest.li 는 HTTP 위에 JSON을 사용한다.
        - 차세대 RPC 프레임워크는 원격 요청이 로컬 함수 호출과 다르다는 사실을 더욱 분명히 한다.
          gRPC는 HTTP 처럼 하나의 요청과 하나의 응답뿐만 아니라 시간에 따른 일련의 요청과 응답으로 구성된 스트림을 지원한다.
        - 이런 프레임워크 중 일부는 **서비스 찾기(service discovery)**를 제공한다. (특정 서비스를 찾을 수 있는 IP 주소와 포트 번호 제공)
        - REST 상에서 JSON과 같은 부류의 프로토콜보다 이진 부호화 형식을 사용하는 사용자 정의 RPC 프로토콜이 우수한 성능을 제공할지 모른다. 하지만 RESTful API 는 다른 중요한 이점이 있다. 실험과 디버깅에 적합(curl 사용해 간단히 요청), 그리고 모든 주요 프로그래밍 언어와 플랫폼이 지원하고 사용 가능한 다양한 도구 생태계 (서버, 캐시 로드 밸런서, 프락시, 방화벽, 모니터링, 디버깅 도구, 테스팅 도구 등)가 있다.
        - RPC 프레임워크의 주요 초점은 보통 같은 데이터센터 내의 같은 조직이 소유한 서비스 간 요청이 있다.
    - 데이터 부호화와 RPC의 발전
        - 발전성이 있으려면 RPC 클라이언트와 서버를 독립적으로 변경하고 배포할 수 있어야 한다. DB를 통한 데이터플로에 비해 서비스를 통한 데이터플로의 발전성은 가정을 단순화할 수 있다. 모든 서버를 먼저 갱신하고 나서 모든 클라이언트를 갱신해도 문제가 없다 가정한다. 그러면 요청은 하위 호환성만 필요하고 응답은 상위 호환성만 필요하다.
        - RPC 스키마의 상하위 호환 속성은 사용된 모든 부호화로부터 상속된다.
            - 스리프트, gRPC(프로토콜 버퍼), 아브로 RPC는 각 부호화 형식의 호환성 규칙에 따라 발전할 수 있다.
            - SOAP에서 요청과 응답은 XML 스키마로 지정한다. 이 방식은 발전 가능하지만 일부 미묘한 함정이 있다.
            - RESTful API는 응답에 JSON 을 가장 일반적으로 사용한다. 그리고 요청에는 JSON 이나 URI 부호화/폼 부호화 요청 매개변수를 사용하곤 한다. 선택적 요청 매개변수 추가나 응답 객체의 새로운 필드 추가는 대개 호환성을 유지하는 변경으로 간주한다.
        - RPC가 종종 조직 경계를 넘나드는 통신에 사용된다는 사실은 서비스 호환성 유지를 더욱 어렵게 한다. 서비스 제공자는 보통 클라이언트를 제어할 수 없고 강제로 업그레이드도 할 수 없기 때문에 호환성은 오랜 시간 동안 유지돼야 한다. 호환성을 깨는 변경이 필요하면 서비스 제공자는 보통 여러 버전의 서비스 API를 함께 유지한다.
        - API 버전 관리가 반드시 어떤 방식으로 동작해야 한다는 합의는 없다.
          RESTful API는 URL 이나 HTTP accept 헤더에 버전 번호를 사용하는 방식이 일반적이다. 특정 클라이언트를 식별하는 데 API 키를 사용하는 서비스는 클라이언트의 요청 API 버전을 서버에 저장한 뒤 버전 선택을 별도 관리 인터페이스를 통해 갱신할 수 있게 하는 것이 한 가지 방식이다.

- **메시지 전달 데이터플로**
    - RPC와 DB 간에 **비동기 메시지 전달 시스템**을 간단히 살펴본다.
      이 시스템은 클라이어트 요청(메시지)을 낮은 지연 시간으로 다른 프로세스에 전달하는 점에서는 RPC와 비슷하다.
      메시지를 직접 네트워크 연결로 전송하지 않고 임시 메시지를 저장하는

메시지 브로커(메시지 큐)나 **메시지 지향 미들웨어**라는 중간 단계를 거쳐 전송한다는 점은 DB와 비슷하다.

- 메시지 전달 통신은 일반적으로 단방향이라는 점이 RPC 와 다르다.
  송신 프로세스는 대게 메시지에 대한 응답을 기대하지 않는다.
  프로세스가 응답을 전송하는 것은 가능하지만 이것은 보통 별도 채널에서 수행한다. (비동기 패턴 방식)
  송신 프로세스는 메시지가 전달될 때까지 기다리지 않고 단순히 메시지를 보낸 다음 잊는다.
- **메시지 브로커**
  - 래빗MQ, 아파치 카프카
  - 장점
    - 수신자가 사용 불가능하거나 과부하 상태라면 메시지 브로커가 버퍼처럼 동작할 수있기 때문에 시스템 안정성이 향상된다.
    - 죽었던 프로세스에 메시지를 다시 전달할 수 있기 때문에 메시지 유실을 방지할 수 있다.
    - 송신자가 수신자의 IP 주소나 포트 번호를 알 필요가 없다.
    - 하나의 메시지를 여러 수신자로 전송할 수 있다.
    - 논리적으로 송신자는 수신자와 분리된다. (publish, consume)
  - 일반적인 사용법
    - 프로세스 하나가 메시지를 이름이 지정된 **큐**나 **토픽**으로 전송하고 브로커는 해당 큐나 토픽 하나 이상 소비자(구독자)에게 메시지를 전달한다.
    - 동일한 토픽에 여러 생산자와 소비자가 있을 수 있다.
  - 토픽
    - 단방향 데이터플로만 제공한다.
    - 소비자 스스로 메시지를 다른 토픽으로 게시하거나 원본 메시지의 송신자가 소비하는 응답 큐로 게시할 수 있다.
  - 특징
    - 특정 데이터 모델을 강요하지 않는다. 메시지는 일부 메타데이터를 가진 바이트열이므로 모든 부호화 형식을 사용할 수 있다.
    - 부호화가 상하위 호환성을 모두 가진다면 메시지 브로커에게 게시자와 소비자를 독립적으로 변경해 임의 순서로 배포할 수 있는 유연성을 얻게 된다.
    - 소비자가 다른 토픽으로 메시지를 다시 게시한다면 DB 맥락에서 데이터 유실를 방지할 목적으로 알지 못하는 필드 보존에는 주의가 필요하다.
- **분산 엑터 프레임워크**
  - 액터 모델
    - 단일 프로세스 안에서 동시성을 위한 프로그래밍 모델
    - 스레드 (경쟁 조건, 잠금(lock), 교착 상태(dead lock) 등)를 직접 처리하는 대신 로직이 액터에 캡슐화된다.
    - 보통 액터는 하나의 클라이언트나 엔티티를 표현한다.
    - 액터는 (다른 액터와 공유되지 않는) 로컬 상태를 가질 수 있고 비동기 메시지의 송수신으로 다른 액터와 통신한다.

        - 액터는 메시지 전달을 보장하지 않는다. (어떤 에러 상황에서 메시지는 유실될 수 있다.)
        - 각 액터 프로세스는 한 번에 하나의 메시지만 처리하기 때문에 스레드에 대해 걱정할 필요가 없고 각 액터는 프레임워크와 독립적으로 실행할 수 있다.
        - 해당 모델은 여러 노드 간의 애플리케이션 확장에 사용된다. 송신자와 수신자가 같은 노드에 있는지 다른 노드에 있는지 관계없이 동일한 메시지 전달 구조를 사용한다. 다른 노드에 있는 경우 메시지는 명백하게 바이트열로 부호화되고 네트워크를 통해 전송되며 다른 쪽에서 복호화한다.
        - 해당 모델은 단일 프로세스 안에서도 메시지가 유실될 수 있다고 가정하기 때문에 위치 투명성은 RPC 보다 액터 모델에서 더 잘 동작한다.
          비록 네트워크를 통한 지연 시간이 동일한 프로세스 안에서 보다 더 높을 수 있지만, 액터 모델을 사용한 경우 로컬과 원격 통신 간 근본적 불일치가 적다.
    - 분산 엑터 프레임워크는 기본적으로 메시지 브로커와 액터 프로그래밍 모델을 단일 프레임워크에 통합한다. 하지만 액터 기반 애플리케이션의 순회식 업그레이드 수행을 원한다면 메시지가 새로운 버전을 수행하는 노드에서 예전 버전을 수행하는 노드로 전송하거나 그 반대의 경우도 있을 수 있으므로 여전히 상하위 호환성에 주의해야 한다.

▼ 정리

- 이번 장에서 데이터 구조를 네트워크나 디스크 상의 바이트열로 변환하는 다양한 방법을 확인하고, 이런 부호화의 세부 사항은 효율성뿐만 아니라 애플리케이션 아키텍처와 배포의 선택 사항에도 영향을 미친다. (하위, 상위 호환성)
- 특히 서비스가 새로운 버전의 서비스를 동시에 모든 노드에 배포하는 방식보다 한 번에 일부 노드에만 서서히 배포하는 순회식 업그레이드가 필요하다.
  순회식 업그레이드는 정지 시간 없이 새로운 버전의 서비스를 출시하게 하고 배포를 덜 위험하게 한다. (큰 출시 보다 잦은 작은 출시 권장)
  이런 속성은 애플리케이션 변경을 쉽게 할 수 있는 발전성에 도움이 된다.
- 순회식 업그레이드 중이거나 여러 가지 다른 이유로 다양한 노드에서 다른 버전의 여러 애플리케이션 코드가 수행된다. 따라서 시스템을 흐르는 모든 데이터는 하위 호환성과 상위 호환성을 제공하는 방식으로 부호화 해야 한다.
- 부호화 형식과 호환성 속성
    - 프로그래밍 언어에 특화된 부호화는 단일 프로그래밍 언어로 제한되며, 상위, 하위 호환성을 제공하지 못하는 경우가 종종 있다.
    - JSON, XML, CSV 같은 텍스트 형식은 널리 사용된다. 이들 간 호환성은 이 형식들을 사용하는 방법에 달렸다. 선택적 스키마 언어가 있으면 때로는 유용하고 때로는 방해된다. 이 형식들은 데이터타입에 대해 다소 모호한 점이 있기 때문에 숫자나 이진 문자열과 같은 항목에 주의해야한다.
    - 스리프트, 프로토콜 버퍼, 아브로 같은 이진 스키마 기반 형식은 짧은 길이로 부호화되며 명확하게 정의된 상위, 하위 호환성의 맥락에서 효율적인 부호화를 지원한다. 이러한 스키마는 정적 타입 언어에서 문서와 코드 생성에 유용하지만 사람이 읽으려면 복호화해야 한다.
- 데이터플로
    - DB에 기록되는 프로세스가 부호화되고 DB 에서 읽는 프로세스가 복호화하는 DB
    - 클라이언트가 요청을 부호화하고 서버는 요청을 복호화하고 응답을 부호화하고 최종적으로 클라이언트가 응답을 복호화하는 RPC 와 REST API
    - 송신자가 부호화하고 수신자가 복호화하는 메시지를 서로 전송해서 노드 간 통신하는 비동기 메시지 전달 (메시지 브로커나 액터)