CP4. 부호화와 발전

Part 1. 데이터 시스템의 기초
4. 부호화와 발전
개요
데이터 부호화 형식
데이터플로 모드
정리

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 원할하게 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
    - 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 그냥 JSON 쓰라고 못 박음)
        - 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

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

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

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"
        {"name": "interests",       "type": {"type
    ]
}
```

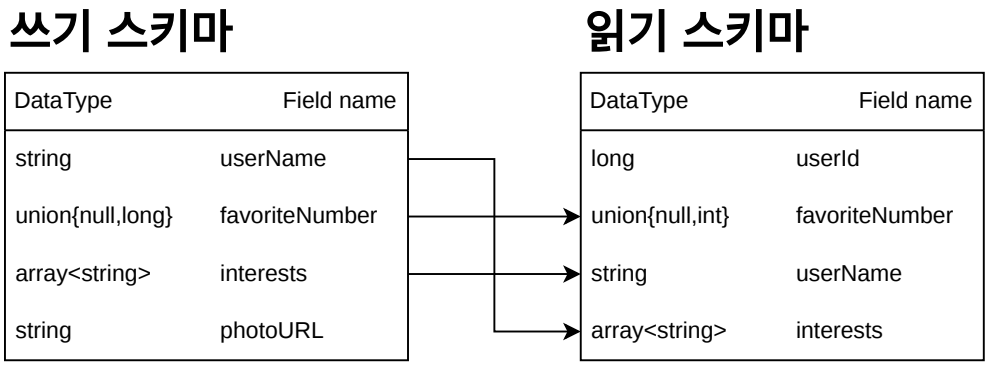
  - 스키마에 태그 번호가 없다.

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

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.

▼ 데이터플로 모드

▼ 정리