4장 부호화와 발전

학습목표

> 데이터 부호화를 위한 다양한 형식을 살펴본다.
> 어떻게 스키마를 변경하고 예전 버전과 새로운 버전의 데이터와 코드가 공존하는 시스템을 어떻게 지원하는지 설명한다.
> 다양한 데이터 부호화 형식이 데이터 저장과 통신에 어떻게 사용되는지 살펴본다.

서론 요약

- 애플리케이션의 유지 보수성 - 발전성에서 말했듯, 애플리케이션은 **기능 변경**이 쉬워야 함.
- **기능 변경**은 **데이터 변경**을 수반함: 데이터 타입, 스키마의 변화
- **데이터 변경**은 **애플리케이션 코드를 바꾸**게 될 수 있음
- 코드를 바꾸면 상위 호환성과 하위 호환성을 고려해야함.
    - 상위 호환 : 새로운 코드가 기록한 데이터를 기존 코드가 읽을 수 있어야 함
      (어려움. 예전 코드가 읽을 수 있게 만들어야하니까.)
    - 하위 호환 : 기존 코드가 기록한 데이터를 새로운 코드가 읽을 수 있어야 함

1. 데이터 부호화 형식

- 부호화 (직렬화, 마샬링) : 파일이나 데이터베이스에 쓰거나 전송할 때, 바이트열 형태로 변환
- 복호화 (파싱, 역직렬화, 언마샬링) : 메모리서 사용하는 데이터 형태(주로 포인터 사용)로 변환

1) 언어별 형식

> 프로그래밍 언어별 부호화 라이브러리

- 자바 java.io.Serializable, Kryo
- 루비 Marchal, 파이썬 pickle

장점	단점
최소한의 코드 추가	다른 언어에서 읽기 어려움
편리하게 사용 가능	복호화 과정에서 원격 임의 코드로 공격할 수 있어 보안 문제
	상-하위 호환성 고려
	효율성 (자원 고비용) 고려

2) 이진 변형 (JSON, XML)

> 표준화된 부호화

장점	단점
언어에 상관 없이 사용 가능 (표준화)	(XML, CSV) 수와 숫자 형식 문자열을 구분하기 어렵다
(JSON) 자바스크립트 사용으로 웹에서 편리하게 사용할 수 있다.	**! 큰 숫자의 경우 정확하게 파싱되지 않을 수 있다**
텍스트 형식이라 사람이 어느정도 읽을 수 있다	이진 문자열은 지원하지 않는다. 해결을 위해 Base64로 부호화한다면 데이터 용량이 증가한다.
데이터 교환 형식(조직간 데이터	(XML, JSON) 스키마를 지원하는 것은, 스키마를 사용하지 않는 어플리케이션의 경우 부호화, 복호화를 직접 하드코딩 해야

2) 이진 변형 - 이진 부호화

- 이진 문자열은 지원하지 않는다...
  - JSON, XML 용 이진 부호화가 개발되었다. (하지만 널리 채택되진 않음)
  - 예) JSON을 메시지팩으로 부호화해 얻은 바이트열 (66바이트) vs 텍스트 JSON 부호화 (81바이트)
    - 용량 (공간 절약, 파싱속도 향상) 우수 vs 사람의 가독성
- 조직 내 규모가 작은 경우
  - **! 조직 내에서만 사용하는 데이터라면 최소 공통 분모 부호화 형식을 사용해야 하는 부담감이 덜하다.**
  - 간편하고 파싱이 빠른 형식인 이진 변형(Json, XML) 사용. (하지만 공간을 더 많이 차지)

3) 스리프트와 프로토콜 버퍼 (이진 부호화 라이브러리)

> 동일 레코드를 32바이트로 부호화

- 스리프트 : 페이스북 개발 , 프로토콜 버퍼 : 구글 개발
- 과정
    i. 둘 다 스키마를 필요로 한다. 각각의 스키마 정의를 사용해 코드를 생성하는 도구를 활용한다.
    ii. 이 도구가 프로그래밍 언어로 구현한 클래스를 생성한다.
    iii. 애플리케이션 코드는 생성된 코드를 호출해 스키마의 레코드를 부호화하고 복호화할 수 있다.

3) 스리프트와 프로토콜 버퍼 (이진 부호화 라이브러리)

- 스리프트의 이진 부호화 형식
  - 바이너리프로토콜
    - 구성 : 타입, 필드태그, (길이), 값
      - 필드 태그 : 필드 이름 대신, 어떤 필드를 다루는지 알려주는 항목
  - 컴팩트프로토콜
    - 구성 : 타입+필드태그, (길이), 값
      - 타입과 필드태그를 단일 바이트로 줄이고 가변 길이 정수를 사용해서 부호화한다.
      - 각 바이트의 상위 비트는 앞으로 더 많은 바이트가 있는지 나타내는데 사용
- 프로토콜 버퍼의 이진 부호화 형식
  - (스리프트의 컴팩트프로토콜 방식과 유사)
- 스키마의 각 필드에 필수, 옵션 표시가 있지만 이진화시에는 나타나지 않음.

3) 스리프트와 프로토콜 버퍼 - 필드 태그와 스키마 발전

- 스키마 발전 : 스키마는 시간이 지나면 변한다.
- 부호화된 레코드에는 타입 / 필드태그 / (길이) / (값) 밖에 없다. 필드 이름을 전혀 참조하고 있지 않다.
    - 필드 이름은 바꿀 수 있음
    - 필드 태그는 바꾸면 모든 부호화된 값을 인식하지 못하게 만들수도 있음.
        - 그래서 필드 태그에 번호를 추가하는 방식으로 스키마에 새로운 필드를 추가할 수 있다.
- 그럼 새로운 코드로 기록한 데이터를 기존 코드가 읽는데 문제가 없음 (데이터 타입 주석을 기준으로 없으면 건너 뜀) : 상위 호환성을 유지
- 기존 코드가 기록한 데이터도 새로운 코드가 읽는데 문제 없음 : 하위 호환성 유지 (대신 새로운 필드가 필수값이 되면 기존 데이터 없어서 못읽음. 그러니 옵션 혹은 기본값을 가져야함)
- 필드 삭제 하는 법 : 옵션 필드만 삭제할 수 있고, 태그 번호를 중복해서 사용할 수 없음.

3) 스리프트와 프로토콜 버퍼 - 데이터타입과 스키마 발전

- 데이터 타입을 변경하는 방법 ?
  - 값이 잘리거나 변형될 수 있음.
  - 예시) 32비트 정수 -> 64비트 정수.
    - 새로운 코드는 예전 코드가 기록한 데이터를 쉽게 읽을 수 있다
    - 새로운 코드가 기록한 데이터를 예전 코드가 읽기 어렵다. 64비트 -> 32비트와 맞지 않아 잘릴 수 있다.
- 이진부호화 라이브러리 별 "목록" 기능
  - 프로토콜 버퍼 : 스키마를 기술하는 필드에 repeated 표시자가 있다. (옵션이었을때의) 단일값을 (반복으로)다중값으로 바꿀 수도 있다. 데이터 타입에 목록은 없지만 목록 역할을 해준다.
  - 스리프트 : 전용 목록 데이터 타입이 있다. 단일값에서 다중값으로 바꿀 수 없지만, 중첩된 목록을 지원한다.

4) 아브로 (이진 부호화 라이브러리)

- 하둡의 하위 프로젝트로 시작
- 스키마 사용.
- 스키마 언어의 종류
  - 아브로 IDL (사람이 편집할 수 있음)
  - JSON 기반 언어 (기계가 더 쉽게 읽음)
- 특징
  - 태그 번호가 없다.
  - 그래서 부호화 길이가 (앞에 말한것 보다) 짧다.
  - 데이터 타입 식별 정보가 없다.
    - 그래서 스키마의 필드를 보고 데이터 타입을 미리 파악해야 한다. 정확하게 같은 스키마를 사용해야 이진 데이터를 복호화 할 수 있다.

4) 아브로 - 쓰기 스키마와 읽기 스키마

- 쓰기 스키마 : 부호화, 읽기 스키마 : 복호화
- 스키마가 동일할 필요가 없다. 아브로 라이브러리는 쓰기 스키마와 읽기 스키마를 함께 살핀 다음 쓰기 스키마에서 읽기 스키마로 데이터를 변환해 그 차이를 해소한다.
    - 순서가 달라도 상관 없음. 이름으로 필드를 일치시킴.
    - 쓰기에만 있는 필드는 무시. 읽기에만 있는 필드는 기본값으로 채운다.

4) 아브로 - 스키마 발전 규칙

- 아브로에서의 상위 호환성 : 새로운 쓰기 스키마와 기존 버전의 읽기 스키마를 가질 수 있음
- 아브로에서의 하위 호환성 : 새로운 읽기 스키마와 기존 버전의 쓰기 스키마를 가질 수 있음
- 호환성 유지를 위해 필요한 것
  - 기본값이 있는 필드만 추가하거나 삭제할 수 있다. (기본값으로 채워진다.)
- 아브로에서 널을 기본값으로 사용하는 방법
  - 유니온 타입 사용.
  - 첫번째로 와야 기본값으로 사용. union {null, long, string} field
- 데이터 변환
  - 타입은 변환할 수 있음
  - 필드 이름 바꾸는 경우, 하위 호환성은 있지만 상위 호환성 없음
  - 유니온 타입에 엘리먼트를 추가하는 것은 하위 호환성 있지만 상위 호환성 없음

4) 아브로 - 그러면 쓰기 스키마는 무엇인가?

- 상황별 쓰기 스키마 사용
  - 많은 레코드가 있는 대용량 파일 : 동일한 스키마로 부호화된 수백만 개 레코드를 포함한 파일을 저장하는 용도로 사용. 이 경우에 파일 시작부분에 한 번만 쓰기 스키마를 포함하면 된다.
  - 개별적으로 기록된 레코드를 가진 데이터베이스 : 각각 다른 쓰기 스키마를 가질 수 있는 경우, 부호화된 레코드의 시작 부분에 버전 번호를 포함하고 데이터베이스에는 스키마 버전 목록을 유지한다. 읽기 레코드에서 버전 정보를 뽑고 데이터베이스에서 버전으로 쓰기 스키마를 찾아 가져온다.
  - 네트워크 연결을 통한 레코드 보내기 : 두 프로세스가 합의된 스키마를 사용한다.

4) 아브로 - 동적 생성 스키마

- = 프로토콜 버퍼와 스리프트보다 아브로 방식이 나은점
    - 태그 번호가 없어 동적 생성 스키마에 더 친숙하다.
    - 태그 번호 있는 경우에는 스키마가 바뀔 때마다 태그 번호를 수동으로 갱신,할당 해줘야 한다.

4) 아브로 - 동적 생성 스키마

- 동적 생성 예시
  - 파일로 덤프할 관계형 데이터베이스가 있다.
  - 관계형 스키마로부터 아브로 스키마를 생성 한다
  - 아브로 스키마를 이용해 데이터베이스를 부호화하고 아브로 객체 컨테이너 파일로 모두 덤프한다
  - 데이터 베이스 테이블에 맞게 레코드 스키마를 생성하고 각 칼럼은 레코드의 필드가 된다.
  - 데이터베이스의 칼럼 이름은 아브로의 필드 이름에 매핑된다.
  - 데이터베이스 스키마가 변경되면 갱신된 데이터베이스 스키마로부터 새로운 아브로 스키마를 생성한다.
  - 새로운 아브로 스키마로 내보낸다.
  - 새로운 데이터 파일을 읽는 사람은 레코드 필드가 변경된 사실을 알게 되지만 필드는 이름으로 식별되기 때문에 갱신된 쓰기 스키마는 여전히 이전 읽기 스키마와 매치 가능하다.

4) 아브로 - 코드 생성과 동적 타입 언어

- 정적 타입 언어 : 자바, C++, C#
  - 복호화된 데이터를 위해 효율적인 인메모리 구조를 사용하고 데이터 구조에 접근하는 프로그램을 작성할 때, IDE에서 타입검사 및 자동완성됨. 스키마 구현 코드를 생성할 수 있다.
  - 스리프트와 프로토콜 버퍼
- 동적 타입 언어 : 자바스크립트, 루비, 파이썬
  - 아브로 (스키마 구현 코드 생성 가능하나 없이도 사용할 수 있다.)

5) 스키마의 장점

- 간단하게 구현하고 사용할 수 있으며, 자세한 유효성 검사 규칙을 지원한다.
- ASN.1 : 네트워크 프로토콜 정의하는데 사용.
    - 태그번호 사용. 복잡하고 문서 부족 등
- 데이터 시스템에서 독자적 구현한 이진 부호화도 있음. 예. 관계형 데이터베이스의 네트워크 프로토콜 ODBC, JDBC API 등의 드라이버
- 장점
    - 필드 이름 생략 가능하여, 다양한 이진 JSON 변형보다 크기가 작을 수 있다.
    - 유용한 문서화 형식이다. 복호화 할 때 스키마가 필요해, 최신 상태인지 확신할 수 있다.
    - 스키마 데이터베이스를 유지하면 스키마 변경이 적용되기 전에 상-하위 호환성을 확인할 수 있다.
    - 정적 타입에서 컴파일 시점에 타입 체크가 가능해, 스키마 코드 생성이 유용하다.

2. 데이터플로 모드

- 데이터플로란?
  - 프로세스 간 데이터를 전달하는 가장 보편적인 방법
    i. 데이터베이스를 통해
    ii. 서비스 호출을 통해
    iii. 비동기 메시지 전달을 통해

1) 데이터베이스를 통한 데이터플로

> 데이터베이스에 기록하는 프로세스가 부호화하고 데이터베이스에서 읽는 프로세스가 복호화하는 데이터베이스

- 동시에 다양한 프로세스가 데이터베이스를 접근하는 일이 생김

1) 데이터베이스를 통한 데이터플로 - 다양한시점에 기록된 다양한 값

- 데이터는 코드보다 오래 산다
- 데이터를 새로운 스키마로 다시 기록 (마이그레이션)도 가능하지만 비싸다
- 그래서 대부분의 관계형 데이터베이스는 기존 데이터를 다시 기록하지 않고 널을 기본값으로 갖는 새로운 칼럼을 추가하는 간단한 스키마 변경을 허용한다.
- 스키마 발전은 기본 저장소가 여러가지 버전의 스키마로 부호화된 레코드를 포함해도 전체 데이터베이스가 단일 스키마로 부호화된 것 처럼 보이게 한다.

1) 데이데이터베이스를 통한 데이터플로터플로 - 보관 저장소

- 데이터 덤프시에 스키마 버전이 섞여 있다고 해도 데이터 복사본을 일관되게 부호화 하는 것이 낫다
    - 기록하고 변하지 않으므로 아브로 객체 컨테이너 파일과 같은 형식이 적합.
    - 파케이와 같은 분석 친화적인 칼럼 지향 형식으로 데이터를 부호화 할 기회

2) 서비스를 통한 데이터플로 : REST와 RPC

> 클라이언트가 요청을 부호화하고 서버는 요청을 복호화하고 응답을 부호화하고 최종적으로 클라이언트가 응답을 복호화하는 RPC와 REST API

- 클라이언트 <-> 서버
- 클라이언트는 서버에 GET, POST 요청을 보낼 수 있음
  - API는 표준화된 프로토콜과 데이터타입 등으로 구성된다.
  - 클라이언트 애플리케이션 코드가 처리를 편리하게 할 수 있게 부호화한 데이터를 서버가 응답할 수 있다.
  - API는 애플리케이션마다 특화되어 있고, 클라이언트와 서버가 해당 API의 세부사항에 동의해야 한다.
- 서비스와 데이터베이스는 유사하다
  - 데이터를 보내거나, 질의한다. 다만 서비스는 미리 정해진 입출력만 허용.
- 서버와 클라이언트가 사용하는 데이터 부호화는 서비스 API 버전간 호환이 가능해야 한다.

2) 서비스를 통한 데이터플로 - 웹 서비스

- HTTP를 통해 서비스에 요청하는 클라이언트 애플리케이션
- 서비스 지향/마이크로서비스 아키텍쳐의 일부로 같은 조직 다른 서비스에 요청하는 서비스
- 인터넷을 통해 다른 조직의 서비스에 요청하는 서비스
  = 모두 웹서비스
- 방법
  - REST : 설계 철학 (API 설계 방법을 정의)
    - 문서 기술하는데 Swagger를 사용할 수 있다.
  - SOAP : XML 기반 프로토콜.
    - WSDL 이라 부르는 XML 기반 언어를 사용한다. 정적 프로그래밍언어에 유용하나 동적으론 유용성이 떨어진다.

2) 서비스를 통한 데이터플로 - 원격 프로시저 호출(RPC) 문제

- RPC 모델은 네트워크 요청을 프로세스 안에서 특정 프로그래밍 언어의 함수나 메소드로 호출하는 것과 동일하게 사용할 수 있게 해준다.
- 로컬 함수 호출은 아니다. 네트워크 요청은 예측이 어렵다.
    - 타임아웃
    - 응답 유실
    - 지연시간
    - 데이터의 크기
    - 데이터의 타입
      를 고려해서 대책을 세워야 한다.

2) 서비스를 통한 데이터플로 - RPC의 현재 방향

- 대책을 고려하여 만듦
- 병렬로 서비스 요청시에 간소화
- 서비스 찾기 제공 (포트번호, IP주소 제공)
- Rest API와의 차이
    - Rest API의 경우 실험과 디버깅 적합. 다양한 프로그래밍 언어, 플랫폼을 지원하며 도구 생태계를 가지고 있음
    - RPC는 같은 데이터 센터 내의 같은 조직이 소유한 서비스 간 요청에 있다

2) 서비스를 통한 데이터플로 - 데이터의 부호화와 RPC의 발전

- RPC 발전성 있으려면 클라이언트와 서버가 독립적으로 바뀔 수 있고 배포될 수 있어야 함.
- 모든 서버를 갱신한 후 모든 클라이언트를 갱신.
    - 요청은 새코드가 기존결과를 읽는 것만 필요하고, 응답은 기존코드가 새코드가 만든 결과를 읽는 것만 필요.
- RPC 서비스 호환성 유지 ? 어렵다
    - 서비스 제공자는 클라이언트를 제어하기 힘들고, 강제 업데이트도 힘들어서 호환성이 무한정 유지되어야 한다.
- API 버전 관리가 어떤 방식으로 동작하여야 한다는 합의는 없으나, 헤더에 버전 번호를 사용하는 것이 일반적
    - 아니면 API키로 특정 클라이언트 식별.

3) 메시지 전달 데이터플로

> 송신자가 부호화하고 수신자가 복호화하는 메시지를 서로 전송해서 노드간 통신하는 비동기 메시지 전달

- 비동기 메시지 전달 시스템
  - 메시지 큐
  - 메시지 브로커
  - 메시지 지향 미들웨어 와 같은 임시 메시지 저장을 이용한다.
- 장점 : 시스템 안정적, 죽었던 프로세스를 메시지에 다시 전달해 유실 방지, 송신자를 알 필요 없다, 하나의 메시지를 여러 수신자로 전송할 수 있다. 송신자와 수신자가 분리된다.
  = 응답을 기다리지 않는다. (비동기)

3) 메시지 전달 데이터플로 - 메시지 브로커

- 사용법
  - 프로세스 하나가 메시지를 이름이 지정된 큐나 토픽으로 전송
  - 브로커가 큐나 토픽에 하나 이상의 소비자 혹은 구독자에게 메시지 전달
  - 동일 토픽에 여러 생산자가 있을 수 있음.
  - (단방향으로 제공)
- 장점
  - 특정 데이터 모델에 한정할 필요 없음
  - 모든 부호화 형식 사용 가능
  - 부호화가 상하위 호환성 모두 가지는 경우 메시지 브로커에서 게시자와 소비자를 변경해 임의 순서 배포 가능 (유연성)
- 주의할 점
  - 소비자가 다른 토픽으로 다시 게시하는 경우 알지 못하는 필드 유실 가능성 있음.

3) 메시지 전달 데이터플로 - 분산 액터 프레임워크

1. 액터 프로그래밍 모델
    - 단일 프로세스 안 동시성을 위한 프로그래밍 모델
    - 스레드 (잠금)를 직접 처리하는 대신 로직이 액터에 캡슐화
    - 보통 액터는 하나의 클라이언트나 엔티티를 나타냄
    - 액터는 로컬 상태를 가질 수 있고 비동기 메시지 송수신으로 다른 액터와 통신한다.
    - 한번에 한 메시지만 처리. 프레임 워크와 독립적으로 실행
    - 여러 노드 간 애플리케이션 확장에 사용된다.
    - 송신자와 수신자가 동일한 메시지 전달 구조를 가지고 있고, 메시지는 바이트로 부호화 되며 네트워크를 통해 전송하고 복호화 된다.
    - 네트워크를 통한 지연시간이 높을 수도 있음
    - 로컬과 원격 통신간의 불일치는 적음.

3) 메시지 전달 데이터플로 - 분산 액터 프레임워크

2. 분산 액터 프레임워크
    - 메시지 브로커와 액터 모델을 단일 프레임워크에 통합
    - 순회식 업그레이드가 필요한 경우 상하위 호환성에 주의
    - 인기 프레임워크 세가지
        - 아카
        - 올리언스
        - 얼랭

정리

- 한번에 모든 노드 업데이트 ? X 순회식 업그레이드 필요
- 작은 출시를 반복하여 새로운 버전의 서비스 출시하도록 하고, 이래야 배포가 덜 위험함. (빠른 롤백 가능)
- 발전성에 도움이 많이 됨