EJ

이펙티브 자바 CP.11

작성 일시	@2023년 4월 2일 오후 9:48
최종 편집 일시	@2023년 4월 4일 오전 12:00
유형	이펙티브 자바
작성자	종현 박
참석자	
언어	

11. 직렬화
85. 자바 직렬화의 대안을 찾으라
86. Serializable을 구현할지는 신중히 결정하라
87. 커스텀 직렬화 형태를 고려해라
88. readObject 메서드는 방어적으로 작성하라
89. 인스턴스 수를 통제해야 한다면 readResolve 보다는 열거 타입을 사용해
90. 직렬화된 인스턴스 대신 직렬화 프록시 사용을 검토하라

11. 직렬화

▼ 85. 자바 직렬화의 대안을 찾으라

- 직렬화
  - 장점
    - 프로그래머가 어렵지 않게 분산 객체를 만들 수 있음
  - 단점
    - 보이지 않는 생성자, API와 구현 사이의 모호해진 경계, 잠재적인 정확성 문제, 성능, 보안, 유지보수성 등 대가가 큼.
    - 공격 범위가 너무 넓고 지속적으로 더 넓어져 방어하기 어렵다
      - `ObjectInputStream` 의 `readObject` 메서드를 호출하면서 객체 그래프가 역직렬화되기 때문이다.

        - `readObject` 메서드는 클래스 패스 안의 거의 모든 타입의 객체를 만들어 낼 수 있는 생성자이다.
        - 바이트 스트림을 역직렬화하는 과정에서 이 메서드는 그 타입들 안의 모든 코드를 수행 할 수 있다. (타입들의 코드 전체가 공격 범위)
    - CERT 조정 센터의 기술 관리자 로버트 시커드

> 자바의 역질렬화는 명백하고 현존하는 위험이다. 이 기술은 지금도 애플리케이션에서 직접 혹은 자바 하부 시스템(RMI - Remote Method Invocation), (JMX - Java Management Extensio), (JMS - Java Messaging System) 을 통해 간접적으로 쓰이고 있기 때문이다. 신뢰할 수 없는 스트림을 역직렬화하면 원격 코드 실행, 서비스 거부 등의 공격으로 이어질 수 있다. 아무 잘 못도 없는 애플리케이션이라도 이런 공격에 취약해질 수 있다.

- 가젯
    - 자바 라이브러리와 널리 쓰이는 서드파트 라이브러리에서 직렬화 가능 타입 중 역직렬화 과정에서 호출되 잠재적으로 위험한 동작을 수행하는 메서드를 뜻함.
- 역직렬화 폭탄
    - 역직렬화에 시간이 오래 걸리는 짧은 스트림을 역직렬화하는 것만으로도 서비스 거부 공격에 쉽게 노출될 수 있는 스트림
    - ex) 역직렬화 폭탄 - 이 스트림의 역직렬화는 영원히 계속됨

```
static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Obejct> s2 = new HashSet<>();
    for (int i = 0; i < 100; i++) {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo"); // t1 과 t2 를 다르게 만든다.
        s1.add(t1); s1.add(t2);
        s2.add(t1); s2.add(t2);
        s1 = t1;
        s2 = t2;
    }
    return serialize(root); // 간결하게 하기 위해 이 메서드의 코드는 생략
}
```

        - `root` 객체 그래프는 201개의 `HashSet` 인스턴스로 구성되어, 그 각각은 3개 이하의 객체 참조를 갖는다.
        - 역직렬화시 끝나지 않는다.
            - 문제는 `HashSet` 인스턴스를 역직렬화하려면 그 원소들의 해시코드를 계산하는 데 있다. 반복문에 의해 구조의 깊이가 100단계까지 만들어진다. 따라서 이 `HashSet` 을 역직렬화하려면 `hashCode` 메서드를 2^100 번 넘게 호출 될 것이다.
            - 역직렬화가 영원히 지속되는 것도 문제지만, 무언가 잘 못되었다는 신호조차 주지 않는 것도 문제이다.
- 역직렬화의 대처법
    - **아무것도 역직렬화하지 않는 것이 가장 좋은 직렬화 위험을 회피하는 방법이다.**
    - **크로스-플랫폼 구조화된 데이터 표현**
        - 객체와 바이트 시퀀스를 변환해주는 다른 매커니즘이 있다.
        - 이 방식들은 자바 직렬화의 위험을 회피하면서 다양한 플랫폼 지원, 우수한 성능, 풍부한 지원 도구, 활발한 커뮤니티 등 수 많은 이점을 제공한다.
        - 이러한 매커니즘들도 직렬화 시스템이라 부르지만, 자바 직렬화와 구분을 해 **크로스-플랫폼 구조화된 데이터 표현**이라고 한다.
- **크로스-플랫폼 구조화된 데이터 표현**
    - 이 표현들의 공통점은 자바 직렬화보다 훨씬 간단하다.
    - 임의 객체 그래프를 자동으로 직렬화/역직렬화 하지 않는다.
    - 대신 속성-값 쌍의 집합으로 구성된 간단하고 구조화된 데이터 객체 사용.
    - 그리고 기본 타입 몇 개와 배열 타입만 지원할 뿐이다.
    - 이런 간단한 추상화만으로도 아주 강력한 분산 시스템을 구축하기에 충분하고, 자바 직렬화의 문제들을 회피 할 수 있다.
- **크로스-플랫폼 구조화된 데이터 표현**의 선두주자
  JSON, 프로토콜 버퍼 (Protocol Buffers, protobuf)
    - JSON
        - 브라우저와 서버의 통신용 설계
        - 자바스크립트용
        - 텍스트 기반 사람이 읽을 수 있음

        - 오직 데이터 표현
    - 프로토콜 버퍼 (Protocol Buffers, protobuf)
        - (구글) 서버 사이에 데이터를 교환하고 저장하기 위해 설계 (gRPC)
        - C++용
        - 이진 표현이라 효율이 높음
        - 문서를 위한 스키마를 제공하고 올바로 쓰도록 강요
- 어쩔 수 없이 자바 직렬화를 사용시
    - 신뢰할 수 없는 데이터는 절대 역직렬화하지 않는 것이다.
- 직렬화를 피할 수 없고 역직렬화한 데이터가 안전한지 완전히 확신할 수 없다면
    - 객체 역직렬화 필터링을 사용하자
        - 데이터 스트림이 역직렬화되기 전에 필터를 설치하는 기능이다.
        - 클래스 단위로, 특정 클래스를 받아들이거나 거부할 수 있다.
        - 블랙 리스트 (기본 수용 모드)
            - 기록된 잠재적으로 위험한 클래스들은 거부
        - 화이트 리스트 방식 (기본 거부 모드) 추천
            - 기록된 안전하다고 알려진 클래스만 승인
- 86 아이템 부터 직렬화 가능 클래스를 올바르고 안전하고 효율적으로 작성하려는 방법을 제시한다.
- 정리
    - 직렬화는 위험하니 피해야한다.
    - 시스템의 밑바닥 부터 설계 한다면 JSON 이나 프로토콜 버퍼 같은 대안을 사용하자
    - 신뢰할 수 없는 데이터는 역직렬화 하지 말자.
    - 꼭 한다면 객체 역직렬화 필터링을 사용하되, 모든 공격을 막아줄 수 없음을 기억하자.
    - 클래스가 직렬화를 지원하도록 만들지 말고, 꼭 만들어야겠으면 정말 신경써야 한다.(아이템 86 ~ 90)

▼ 86. Serializable을 구현할지는 신중히 결정하라

- 85 에서 말했듯이, 직렬화의 대안을 찾지 못한 경우에만 적용되는 내용이다.
- 개요
  - 직렬화는 클래스 선언에서 `implements Serializable`만 덧 붙이면 된다.
  - 그래서 굉장히 쉬워 보이지만, 길게 보면 아주 값 비싼 일이다.
  - 구현시 고려할 사항을 소개 한다.
- **Serializable 을 구현하면 릴리스 한 뒤에는 수정하기 어렵다.**
  - 구현하게 되면, 직렬화된 바이트 스트림 인코딩도 하나의 공개 API가 된다.
  - 커스텀 직렬화 형태를 설계하지 않고 자바의 기본 방식을 사용하면 직렬화 형태는 최소 적용 당시 클래스의 내부 구현 방식에 영원히 묶여버린다.
  - 달리 말하면, 기본 직렬화 형태에서는 클래스의 `private` 과 `package-private` 인스턴스 필드들 마저 API로 공개되는 꼴이 된다. (캡슐화 깨짐)
  - 릴리스 후 클래스 내부 구현을 손을 보면 원래 직렬화 형태와 달라지게 된다.
    - 한 쪽은 구버전 인스턴스를 직렬화하고 다른 쪽은 신버전 클래스로 역직렬화한다면, 실패한다.
    - 원래 직렬화 형태를 유지하면서 내부 표현을 바꿀 수도 있지만, 어렵기도 하고 소스코드에 지저분한 형태가 남겨지게 된다.
    - 그러니, 직렬화 가능 클래스를 만들고자 한다면, 길게 보고 감당할 수 있을 만큼 고품질의 직렬화 형태도 주의해서 함께 설계해야 한다. (아이템 87, 90)
  - 직렬화는 클래스 개선을 방해한다.
    - 대표적으로 스트림 고유 식별자 (직렬 버전 UID)를 들 수 있다.
    - 모든 직렬화된 클래스는 고유 식별 번호를 부여받는다.
    - `serialVersionUID` 라는 이름의 `static final long` 필드로, 이 번호를 명시하지 않으면 시스템이 런타임에 암호 해시 함수(SHA-1)를 적용해 자동으로 클래스 안에 생성해 넣는다.
    - 이 값을 생성하는 데는 클래스 이름, 구현한 인터페이스들, 컴파일러가 자동으로 생성해 넣은 것을 포함한 대부분의 클래스 멤버들이 고려된다.
    - 그래서 추후에 편의 메서드를 추가하는 식으로 이들 중 하나라도 수정한다면 직렬 버전 UID 값도 변한다.
    - 즉, 자동 생성되는 값에 의존하면 쉽게 호환성이 깨져버려 런타임에 `InvalidClassException` 이 발생한다.

- **버그와 보안 구멍이 생길 위험이 높아진다.** (아이템 85)
  - 객체는 생성자를 사용해 만드는 것이 기본이다.
    즉 직렬화는 언어의 기본 매커니즘을 우회하는 객체 생성 기법이다.
    역직렬화는 일반 생성자의 문제가 그대로 적용되는 '숨은 생성자' 이다.
  - 기본 역직렬화를 사용하면 불변식 깨짐과 허가되지 않은 접근에 쉽게 노출된다. (아이템 88)
- **직렬화 구현 클래스의 신버전을 릴리스할 때 테스트 할 것이 늘어난다.**
  - 직렬화 가능 클래스가 수정되면 신버전 인스턴스를 직렬화한 후 구 버전으로 역직렬화 할 수 있는지, 그 반대도 가능한지를 검사해야 한다.
    양방향 직렬화/역직렬화가 모두 성공하고, 원래의 객체를 충실히 복제하는지 확인 필요
  - 테스트 양 = 직렬화 가능 클래스 수 * 릴리스 횟수 가 된다.
  - 클래스를 처음 제작할 때 커스텀 직렬화 형태를 잘 설계했다면 이러한 테스트 부담을 줄 일 수 있다. (아이템 87, 90)
- **상속용으로 설계된 클래스(아이템 19)는 대부분 `Serializable`을 구현하면 안 되며, 인터페이스도 대부분 `Serializable`을 확장해서는 안 된다.**
  - 이 규칙을 따르지 않으면, Serializable 을 확장, 구현 한 클래스, 인터페이스에 커다란 부담을 준다.
  - 작성하는 클래스의 인스턴스 필드가 직렬화와 확장이 모두 가능하다면 주의할 점
    - 인스턴스 필드 값 중 불변식을 보장해야 할 게 있다면 반드시 하위 클래스에서 `finalize` 메서드를 재정의하지 못하게 해야한다.
      즉, `finalize` 메서드를 자신이 재정의하면서 `final` 로 선언하면 된다.
      이렇게 하지 않으면 `finalizer` 공격 (아이템 8) 을 받을 수 있다.
    - 인스턴스 필드 중 기본값 (정수형 `0`, `boolean`은 `false`, 객체 참조 타입은 `null`)으로 초기화되면 위배되는 불변식이 있는 클래스에 다음에 `readObjectNodata` 메서드를 반드시 추가해야 한다.
    - ex) 상태 있고, 확장 가능하고, 직렬화 가능한 클래스용 `readObjectNoData` 메서드

```
private void readObjectNoData() throws InvaildObjectException {
    throw new InvalidObjectException("스트림 데이터가 필요하다.");
}
```

- `Serialzable` 을 구현하지 않기로 할 때 주의점
  - 상속용 클래스인데 직렬화를 지원하지 않으면 그 하위 클래스에서 직렬화를 지원하려 할 때 부담이 늘어난다.
  - 보통은 이런 클래스를 역직렬화하려면 그 상위 클래스는 매개변수가 없는 생성자를 제공해야 한다는데, 이런 생성자를 제공하지 않으려면 하위 클래스에서는 어쩔 수 없이 직렬화 프록시 패턴 (아이템 90)을 사용해야 한다.
- 내부 클래스(아이템 24)는 직렬화를 구현하지 말아야 한다.
  - 내부 클래스에는 바깥 인스턴스의 참조와 유효 범위 안의 지역변수 값들을 저장하기 위해 컴파일러가 생성한 필드들이 자동으로 추가된다.
  - 내부 클래스에 대한 기본 직렬화 형태는 분명하지 않아 직렬화 구현을 하면 안 된다.
  - 단, 정적 멤버 클래스는 `Serializable` 을 구현해도 된다.
- 정리
  - Serializable 을 구현한다고 선언하기는 쉽지만, 굉장히 어려운 일이다.
  - 한 클래스의 여러 버전이 상호작용할 일 없고, 서버가 신뢰할 수 없는 데이터에 노출될 가능성이 없는 등, 보호된 환경에서만 쓰일 클래스가 아니라면 `Serializable` 구현은 아주 신중하게 이뤄져야 한다.
  - 상속할 수 있는 클래스라면 주의 사항이 더 많아진다.

▼ 87. 커스텀 직렬화 형태를 고려해라

▼ 88. readObject 메서드는 방어적으로 작성하라

▼ 89. 인스턴스 수를 통제해야 한다면 readResolve 보다는 열거 타입을 사용해

▼ 90. 직렬화된 인스턴스 대신 직렬화 프록시 사용을 검토하라