Gson v2.8.6 번역

작성자 : Charlezz, 추지철
작성일 : 2020.10.13
수정일 : 2020.10.13
버전 : v1.0

Gson 공식 문서를 번역한 내용입니다.
(https://github.com/google/gson/blob/master/UserGuide.md)
오역 및 오탈자가 다수 있을 수 있으니 양해 부탁드립니다.

오역 제보 및 건의(charlezz@charlezz.com)
이 문서의 내용을 블로그 및 웹사이트에 게시하는 경우 출처(https://charlezz.com)**를 꼭 남겨주시기 바랍니다.**

그 외 문의사항은 찰스의 안드로이드 오픈채팅방에 오셔서 자유롭게 질문해주세요.

개요

Gson은 자바 객체를 JSON 형식으로 변경시킬 때 사용할 수 있는 자바 라이브러리다. Gson은 또한 JSON 문자열을 동등한 자바 객체로 변환하는데 사용할 수 있다.

Gson은 소스 코드가 없는 기존 객체를 포함하여 임의의 자바 객체로 작업 할 수 있다.

Gson을 사용하는 목적

- toString() 및 생성자(팩토리 메서드)와 같은 방법을 제공하여 Java와 JSON간에 쉬운 변환 방법을 제공한다.
- 기존의 수정 불가능한 객체를 JSON으로 변환하거나 JSON으로 부터 변환할 수 있도록 한다.
- 객체에 대한 커스텀 표현을 허용한다.
- 임의의 복잡한 객체를 지원한다.
- 간결하고 읽기 쉬운 JSON을 생성한다.

Gson 퍼포먼스와 확장성

다음은 많은 것들을 실행하는 테스트와 함께 데스크탑에서 얻은 몇가지 측정결과를 보여준다. PerformanceTest 클래스를 사용하여 이러한 테스트를 재수행할 수 있다.

- 문자열: 오류없이 25MB 이상의 문자열을 역직렬화 (PerformaceTest의 disabled_testStringDeserializationPerformance 메서드를 확인)
- 대량의 컬렉션:
  - 140만개의 컬렉션 객체 직렬화 (PerformanceTest에서

disabled_testLargeCollectionSerialization를 확인)
- 87,000 개 컬렉션 객체 역직렬화 (PerformanceTest에서
disabled_testLargeCollectionDeserialization를 확인)

Note: 이러한 테스트를 실행하려면 disabled_접두어를 삭제하자. 이 접두어를 사용하여 JUnit 테스트를 실행할 때마다 이러한 테스트가 실행되는 것을 방지할 수 있다.

Gson 사용자

Gson은 원래 Google 내부에서 사용하기 위해 만들어졌으며, 현재 구글의 여러 프로젝트에서 사용중이다. 또한 많은 공공 프로젝트 및 회사에서 사용되고 있다.

Gson 사용하기

사용되는 기본 클래스는 Gson으로 new Gson()을 호출하여 생성 할 수 있다. 또한 GsonBuilder라는 클래스가 있으며, 버전 제어 등과 같은 다양한 설정과 함께 Gson 인스턴스를 생성할 때 사용한다.

Gson 인스턴스는 Json 연산자를 호출할 때 어떠한 상태도 유지하지 않는다. 그래서 같은 객체를 통해 여러 Json 직렬화 및 역직렬화를 할 때 자유롭게 재사용할 수 있다.

Gradle/Android에서 Gson 사용하기

```
dependencies {
    implementation 'com.google.code.gson:gson:2.8.6'
}
```

Maven에서 Gson 사용하기

Maven 2/3과 함께 Gson을 사용하려면, 다음에 오는 의존성을 추가함으로써 Maven 중심으로 Gson 버전을 사용할 수 있다.

```
<dependencies>
    <!--  Gson: Java to Json conversion -->
    <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.8.6</version>
      <scope>compile</scope>
    </dependency>
</dependencies>
```

즉, 이제 Maven 프로젝트에서 Gson을 사용할 수 있다.

기본 예제

```
// 직렬화
Gson gson = new Gson();
gson.toJson(1);            // ==> 1
gson.toJson("abcd");       // ==> "abcd"
gson.toJson(new Long(10)); // ==> 10
int[] values = { 1 };
gson.toJson(values);       // ==> [1]

// 역직렬화
int one = gson.fromJson("1", int.class);
Integer one = gson.fromJson("1", Integer.class);
Long one = gson.fromJson("1", Long.class);
Boolean false = gson.fromJson("false", Boolean.class);
String str = gson.fromJson("\"abc\"", String.class);
String[] anotherStr = gson.fromJson("[\"abc\"]", String[].class);
```

객체 예제

```
class BagOfPrimitives {
  private int value1 = 1;
  private String value2 = "abc";
  private transient int value3 = 3;
  BagOfPrimitives() {
    // 인자 없는 생성자
  }
}

// 직렬화
BagOfPrimitives obj = new BagOfPrimitives();
Gson gson = new Gson();
String json = gson.toJson(obj);

// ==> {"value1":1,"value2":"abc"}
```

순환 참조가 있는 객체는 무한 재귀함수를 초래할 수 있기 때문에 직렬화 할 수 없다.

```
// 역직렬화
BagOfPrimitives obj2 = gson.fromJson(json, BagOfPrimitives.class);
// ==> obj2는 마치 obj와 같다
```

객체에 대한 팁들

- private 필드를 사용해도 문제없고, 권장한다.
- 직렬화 및 역직렬화에 포함될 필드를 가르키는 어떠한 어노테이션도 필요가 없다. (상위 클래스를 포함한)현재 클래스의 모든 필드가 기본적으로 포함된다.
- 필드가 transient로 표시된 경우, (기본적으로) 이는 무시되고 JSON 직렬화 또는 역직렬화에 포함되지 않는다.
- 다음과 같이 null을 올바르게 다룬다.
  - 직렬화를 할 때 출력된 결과물에서는 null 필드가 생략된다.

- 역직렬화를 하는 동안 JSON에서 누락된 항목은 객체의 해당 필드를 기본값으로 설정한다: 객체 타입은 null, 숫자 타입은 0(zero), boolean 타입은 false

- synthetic 필드의 경우, 무시되고 JSON의 직렬화 또는 역직렬화에 포함되지 않는다.
- 내부 클래스(Inner class)안에 있는 외부 클래스(Outer class), 익명 클래스 그리고 로컬 클래스에 해당하는 필드들은 무시되고 직렬화 또는 역직렬화에 포함되지 않는다.

중첩 클래스(내부 클래스 포함)

Gson은 static 중첩 클래스를 꽤 쉽게 직렬화할 수 있다.

Gson은 또한 static 중첩 클래스를 역직렬화할 수 있다. 그러나, Gson은 인자가 없는 생성자 또한 역직렬화 시에 사용할 수 없는 포함된 객체에 대한 참조를 필요하기 때문에 순수 내부 클래스를 자동으로 역직렬화할 수 없다. 내부 클래스를 static으로 만들거나 커스텀 InstanceCreator을 제공함으로써 이 문제를 해결할 수 있다. 예제를 보자.

```
public class A {
  public String a;

  class B {

    public String b;

    public B() {
      // 인자 없는 B 생성자
    }
  }
}
```

Note : 위 클래스 B는 (기본적으로) Gson으로 직렬화될 수 없다.

Gson은 클래스 B는 내부 클래스이기 때문에 {"b" : "abc"}를 B 인스턴스로 역직렬화할 수 없다. 만약 static 클래스 B로 정의되어 있다면, Gson은 문자열을 역직렬화할 수 있었을 것이다. 또 다른 해결책은 B에 대한 커스텀 인스턴스 creator를 작성하는 것이다.

```
public class InstanceCreatorForB implements InstanceCreator<A.B> {
  private final A a;
  public InstanceCreatorForB(A a)  {
    this.a = a;
  }
  public A.B createInstance(Type type) {
    return a.new B();
  }
}
```

위 예제코드는 실행가능하지만, 추천하지는 않는다.

배열 예제

```
Gson gson = new Gson();
int[] ints = {1, 2, 3, 4, 5};
String[] strings = {"abc", "def", "ghi"};
```

```
// 직렬화
gson.toJson(ints);     // ==> [1,2,3,4,5]
gson.toJson(strings);  // ==> ["abc", "def", "ghi"]

// 역직렬화
int[] ints2 = gson.fromJson("[1,2,3,4,5]", int[].class);
// ints2는 ints와 같다.
```

또한 임의의 복잡한 요소 타입이 있는 다차원 배열을 지원한다.

콜렉션 예제

```
Gson gson = new Gson();
Collection<Integer> ints = Lists.immutableList(1,2,3,4,5);

// 직렬화
String json = gson.toJson(ints);  // ==> [1,2,3,4,5]

// 역직렬화
Type collectionType = new TypeToken<Collection<Integer>>(){}.getType();
Collection<Integer> ints2 = gson.fromJson(json, collectionType);
// ==> ints2는 ints와 같다
```

상당히 찝찝한 부분: 어떻게 컬렉션 유형을 정의하는지 주목하자. 불행하게도, Java에서는 이 문제를 해결할 수있는 방법이 없다.

콜렉션 제한

Gson은 임의의 객체의 컬렉션을 직렬화할 수 있지만, 사용자가 결과 객체의 유형을 표시하기 위한 방법이 없기 때문에 이를 역직렬화 할 수 없다. 대신에, 역직렬화하는 동안, 컬렉션은 특정 제네릭 타입이어야 한다. 이는 의미가 있고, 좋은 자바 코딩 관습을 따를 때 거의 문제가 되지 않는다.

제네릭 타입 직렬화 및 역직렬화 하기

toJson(obj)을 호출 할 때, Gson은 직렬화하기 위한 필드정보를 가져오기 위해 obj.getClass()를 호출한다. 그와 유사하게, MyClass.class 객체를 fromJson(json, MyClass.class) 메서드에 일반적으로 전달 할 수 있다. 객체가 제네릭 타입이 아니라면 이는 잘 동작한다. 그러나 객체가 제네릭 타입이라면 자바의 타입 소거(Type Erasure) 때문에 해당 제네릭 타입 정보를 잃게 된다. 다음 예제를 통해 요점을 살펴보자.

```
class Foo<T> {
  T value;
}
Gson gson = new Gson();
Foo<Bar> foo = new Foo<Bar>();
gson.toJson(foo); // foo.value는 직렬화 할 수 없다.

gson.fromJson(json, foo.getClass()); // Bar타입의 foo.value를 역직렬화 하는데 실패한다.
```

Gson은 해당 클래스 정보를 얻기 위해 foo.getClass()를 호출하므로 위의 코드는 Bar타입으로 값을 해석하지 못한다. 하지만 이 메서드는 raw 클래스인 Foo.class를 반환한다. 이것은 Gson이 단순한

Foo가 아닌 Foo<Bar> 객체 타입을 알 방법이 없다는 것을 의미한다.

제네릭 타입에 대해 올바른 매개변수화 된 타입을 지정하여 이 문제를 해결 할 수 있다. TypeToken 클래스를 사용하여 이를 수행할 수 있다.

```
Type fooType = new TypeToken<Foo<Bar>>() {}.getType();
gson.toJson(foo, fooType);

gson.fromJson(json, fooType);
```

fooType을 가져 오는데 사용되는 관용구는 실제로 완전히 매개변수화 된 타입을 반환하는 getType() 메서드를 포함하는 익명 로컬 내부 클래스를 정의한다.

임의의 타입의 객체들을 가지고 컬렉션 직렬화 및 역직렬화 하기

때때로 섞인 유형들을 포함한 JSON 배열을 다룬다. 예를 들어: ['hello',5, {name:'GREETINGS',source:'guest'}]

를 포함하는 Collection은

```
Collection collection = new ArrayList();
collection.add("hello");
collection.add(5);
collection.add(new Event("GREETINGS", "guest"));
```

이며, Event 클래스는 아래처럼 정의된다.

```
class Event {
  private String name;
  private String source;
  private Event(String name, String source) {
    this.name = name;
    this.source = source;
  }
}
```

어떤 특정한 작업 - toJson(컬렉션)은 원하는 출력을 작성한다. - 없이 Gson을 컬렉션을 직렬화 할 수 있다.

그러나, fromJson(json, Collection.class)를 가진 역직렬화는 Gson이 입력 유형에 매핑하는 방법을 알 방법이 없기 때문에 동작하지 않을 것이다. Gson은 fromJson()에 컬렉션 유형의 일반화된 버전을 제공하기를 요청한다. 그래서, 세개의 옵션을 가진다.

1. 배열 요소들을 파싱하기 위해 Gson의 파서(parser) API (저수준 스트리밍 파서 또는 DOM 파서 JsonParser) 를 사용한 다음 각 배열 요소에서 Gson.fromJson()을 사용한다. 이건 선호화된 접근방법이다. 해당 페이지 는 이를 수행하는 방법을 보여주는 예제다.
2. 각 배열 멤버들을 확인하고 적절한 객체들에 매핑하는 Collection.class에 대한 유형 어댑터를 등록하자. 이 접근 방법의 단점은 Gson에 다른 컬렉션 유형의 역직렬화를 망칠 것이라는 점이다.

3. MyCollectionMemberType에 대한 유형 어댑터를 등록하고 Collection<MyCollectionMemberType>를 가진 fromJson()을 사용하자.

이 접근법은 배열이 최상위 요소로 표시되거나 컬렉션을 보유하는 필드 유형을 Collection<MyCollectionMemberType> 유형으로 변경할 수 있는 경우에만 유용하다.

내장된 직렬화 및 역직렬화 변환기

Gson에서는 일반적으로 사용되는 클래스에 대해 기본적인 표현이 부적절할 수도 있는 내장된 직렬화 및 역직렬화 변환기를 가지고 있다. 예를 들면 다음과 같다.

- "https://github.com/google/gosn/"과 같은 문자열은 java.net.URL과 일치한다.
- "/google/gson/"과 같은 문자열은 java.net.URI와 일치한다.

자세한 내용은 내부 클래스 TypeAdapters를 참조하자.

또한 JodaTime과 같은 몇가지 일반적으로 사용되는 클래스에 대한 소스코드는 이 페이지에서 찾을 수 있다.

커스텀 직렬화와 역직렬화

때로는 기본적인 표현이 원하는 내용이 아닐 수 있다. 라이브러리 클래스 (DateTime, 등등) 를 다룰 때 이런 경우가 종종 있다. Gson은 개인의 커스텀 직렬화 변환기와 역직렬화 변환기 등록할 수 있다. 이건 다음 두 부분을 정의함으로써 끝난다.

- Json 직렬화 변환기 : 객체에 대한 커스텀 직렬화 정의가 필요하다.
- Json 역직렬화 변환기 : 유형에 대한 커스텀 역직렬화 정의가 필요하다.
- Instance Creators : 인자가 없는 생성자가 사용되거나 역직렬화 변환기가 등록된다면 필요하지 않는다.

```
GsonBuilder gson = new GsonBuilder();
gson.registerTypeAdapter(MyType2.class, new MyTypeAdapter());
gson.registerTypeAdapter(MyType.class, new MySerializer());
gson.registerTypeAdapter(MyType.class, new MyDeserializer());
gson.registerTypeAdapter(MyType.class, new MyInstanceCreator());
```

registerTypeAdapter 호출은 유형 어댑터가 이러한 인터페이스들 중 하나 이상을 구현하는지 확인하고 모든 인터페이스에 대해 등록한다.

직렬화 변환기 작성

JodaTime DateTime 클래스에 대한 커스텀 직렬화 변환기를 작성하는 방법에 대한 예제이다.

```
private class DateTimeSerializer implements JsonSerializer<DateTime> {
  public JsonElement serialize(DateTime src, Type typeOfSrc, JsonSerializationContext context) {
    return new JsonPrimitive(src.toString());
  }
}
```

Gson은 직렬화하는 동안 DateTime 객체 내에서 실행될 때 serialize()를 호출한다.

역직렬화 변환기 작성

JodaTime DateTime 클래스에 대한 커스텀 역직렬화 변환기를 작성하는 방법에 대한 예제이다.

```
private class DateTimeDeserializer implements JsonDeserializer<DateTime> {
  public DateTime deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context)
      throws JsonParseException {
    return new DateTime(json.getAsJsonPrimitive().getAsString());
  }
}
```

Gson은 DateTime 객체 내에서 JSON 문자열 조각이 역직렬화를 필요로 할 때 deserialize()를 호출한다.

직렬화 변환기와 역직렬화 변환기에 미세한 점

종종 raw 유형에 일치하는 모든 제네릭 유형에 대해 단일 핸들러를 등록하길 원한다.

- 예를 들어, id 표현/번역에 대해 Id 클래스를 가진다 가정하자.(즉, 내부 표현과 외부 표현)
- 모든 제네릭 유형에 대한 같은 직렬화를 있는 Id<T> 유형
  - 기본적으로 id값을 작성해야 한다.
- 역직렬화는 매우 유사하지만 정확하게 동일하지는 않다.
  - Id<T>의 인스턴스를 반환하는 새로운 Id(Class<T>, String)을 호출해야만 한다.

Gson은 이에 대해 단일 핸들러 등록하는 것을 지원한다. 특정 제네릭 유형(예, Id<RequiresSpecialHandling> 특별한 핸들링이 필요하다)에 대해 특정 핸들러를 등록할 수 있다. toJson()과 fromJson()에 대한 Type 파라미터는 같은 raw 유형에 일치하는 모든 제네릭 유형에 대한 단일 핸들러를 작성하는 것을 돕기 위한 제네릭 유형 정보를 포함한다.

Instance Creator 작성하기

객체를 역직렬화할 때, Gson은 클래스의 기본 인스턴스를 만들어야 한다. 직렬화 및 역직렬화가 제대로 작동하려면 클래스에는 인자가 없는 생성자가 있어야 한다.

- private 이건 public이건 상관없다.

일반적으로 InstanceCreator는 인자가 없는 생성자가 정의되지 않은 라이브러리 클래스를 다룰 때 필요하다.

Instance Creator 예제

```
private class MoneyInstanceCreator implements InstanceCreator<Money> {
  public Money createInstance(Type type) {
    return new Money("1000000", CurrencyCode.USD);
  }
}
```

타입은 제네릭 타입일 수도 있다.

- 특정 제네릭 타입 정보가 필요한 생성자를 호출하는 데 매우 유용하다.
- 예를 들면, Id 클래스가 Id가 생성되는 클래스를 저장하는 경우

매개변수화 된 타입을 위한 InstanceCreator

인스턴스화 하려는 타입이 매개변수화 된 타입인 경우가 있다. 일반적으로 실제 인스턴스는 raw 타입이므로 문제가 되지 않는다. 다음 예제를 보자.

```
class MyList<T> extends ArrayList<T> {
}

class MyListInstanceCreator implements InstanceCreator<MyList<?>> {
    @SuppressWarnings("unchecked")
  public MyList<?> createInstance(Type type) {
    // 실제 인스턴스는 어쨌든 raw 타입을 가지므로 매개변수가 있는 List를 사용할 필요가 없다.
    return new MyList();
  }
}
```

그러나 실제 매개변수화 된 타입을 기반으로 인스턴스를 만들어야하는 경우가 있다. 이 경우 createInstance 메서드에 전달되는 type 매개변수를 사용할 수 있다. 다음 예제를 보자.

```
public class Id<T> {
  private final Class<T> classOfId;
  private final long value;
  public Id(Class<T> classOfId, long value) {
    this.classOfId = classOfId;
    this.value = value;
  }
}

class IdInstanceCreator implements InstanceCreator<Id<?>> {
  public Id<?> createInstance(Type type) {
    Type[] typeParameters = ((ParameterizedType)type).getActualTypeArguments();
    Type idType = typeParameters[0]; // Id 단지 하나의 매개변수화된 타입 T를 갖는다.
    return Id.get((Class)idType, 0L);
  }
}
```

위의 예제에서, Id 클래스의 인스턴스는 매개변수화 된 타입에 대한 실제 타입을 실제로 전달하지 않고는 생성될 수 없다. 이 문제를 해결하기 위해 전달된 메소드 매개변수 type을 사용한다. 이 경우 type 객체는 실제 인스턴스가 Id<Foo>에 바인딩 되어야 하는 Id<Foo>의 자바 매개변수 타입 표현이다. Id 클래스에는 매개변수화 된 타입 매개 변수 T가 하나만 있으므로 이 경우 Foo.class를 보유할 getActualTypeArgument()가 반환한 타입 배열의 0번째 요소를 사용한다.

JSON 출력 포맷, 함축 vs 예쁜 출력

Gson에서 제공하는 기본 JSON 출력 포맷은 함축적인 JSON 형식이다. 이는 출력되는 JSON 구조 내에 공백이 없음을 의미한다. 그러므로 JSON 출력에서 필드 이름과 값, 객체 필드, 그리고 배열 내 객체 사이에서 공백은 없을 것이다. 뿐만 아니라, "null" 필드는 출력에서 무시된다.(NOTE : null 값은 객체의 컬렉션/배열 내 여전히 포함되어 있다). 모든 null 값을 출력하도록 Gson을 구성하는 방법에 대해선 Null 객체 지원을 살펴보자.

예쁘게 출력(pretty print)하고 싶다면, GsonBuilder를 사용해 Gson 인스턴스를 구성해야 한다. JsonFormatter는 public API를 통해 노출되지 않으므로, 클라이언트는 JSON 출력에 대한 기본 출력 설정/여백을 구성을 할 수 없다. 현재는, 기본 줄 길이가 80자, 들여쓰기 2자, 오른쪽 여백이 4문자인 기본 JsonPrintFormatter 만을 제공한다.

다음은 JsonCompactFormatter 대신에 기본 JsonPrintFormatter를 사용해 Gson 인스턴스를 구성하는 방법에 대해 보여주는 예제이다.

```
Gson gson = new GsonBuilder().setPrettyPrinting().create();
String jsonOutput = gson.toJson(someObject);
```

Null 객체 지원

Gson에서 구현되는 기본 동작은 null객체 필드가 무시된다는 점이다. 이것은 보다 간결한 출력 형식을 허용한다. 그러나 클라이언트는 JSON 형식이 Java 형식으로 다시 변환되므로 이러한 필드에 대한 기본값을 정의해야 한다.

null을 출력하도록 Gson 인스턴스를 구성하는 방법은 다음과 같다.

```
Gson gson = new GsonBuilder().serializeNulls().create();
```

Note : Gson을 사용하여 null을 직렬화 할 때 JsonElement 구조에 JsonNull 요소가 추가된다. 따라서 이 객체는 커스텀 직렬화 또는 역직렬화에 사용할 수 있다.

다음 예제를 살펴보자

```
public class Foo {
  private final String s;
  private final int i;

  public Foo() {
    this(null, 5);
  }

  public Foo(String s, int i) {
    this.s = s;
    this.i = i;
  }
}

Gson gson = new GsonBuilder().serializeNulls().create();
Foo foo = new Foo();
String json = gson.toJson(foo);
System.out.println(json);

json = gson.toJson(null);
System.out.println(json);
```

출력은 다음과 같다:

```
{"s":null,"i":5}
null
```

버전 지원

같은 객체의 다양한 버전들은 @Since 어노테이션을 사용함으로써 유지할 수 있다. 이 어노테이션은 클래스, 필드 그리고 향후 릴리즈, 메서드에서 사용할 수 있다. 이 기능을 활용하기 위해, 일부 버전 번호보다 큰 필드/객체를 무시하는 Gson 인스턴스를 구성해야만 한다. 만약 버전이 Gson 인스턴스에 설정되어 있지 않다면, 버전에 관계없이 모든 필드와 클래스를 직렬화하고 역직렬화 할 것이다.

```
public class VersionedClass {
  @Since(1.1) private final String newerField;
  @Since(1.0) private final String newField;
  private final String field;

  public VersionedClass() {
    this.newerField = "newer";
    this.newField = "new";
    this.field = "old";
  }
}

VersionedClass versionedObject = new VersionedClass();
Gson gson = new GsonBuilder().setVersion(1.0).create();
String jsonOutput = gson.toJson(versionedObject);
System.out.println(jsonOutput);
System.out.println();

gson = new Gson();
jsonOutput = gson.toJson(versionedObject);
System.out.println(jsonOutput);
```

출력은 다음과 같다.

```
{"newField":"new","field":"old"}

{"newerField":"newer","newField":"new","field":"old"}
```

직렬화 및 역직렬화에서 필드 제외하기

Gson은 최상위 클래스, 필드 및 필드 타입을 제외하기 위한 다양한 방법을 지원한다. 다음은 필드 및 클래스 제외를 허용하는 유연한(Pluggable) 방법을 보여준다. 아래 방법 중 어느 것도 요구 사항을 충족하지 않는 경우 항상 커스텀 직렬화 및 역직렬화를 사용할 수 있다.

자바 제어자 제외

기본적으로 필드를 transient로 표시하면 제외된다. 또한 필드가 static으로 표시되면 기본적으로 제외된다. 일부 trasient 필드를 포함하려면 다음을 수행 할 수 있다.

```
import java.lang.reflect.Modifier;
Gson gson = new GsonBuilder()
    .excludeFieldsWithModifiers(Modifier.STATIC)
    .create();
```

Note : excludeFieldsWithModifiers 메서드에 Modifier 상수를 여러개 지정할 수 있다. 예를 들면 다음과 같다.

```
Gson gson = new GsonBuilder()
    .excludeFieldsWithModifiers(Modifier.STATIC, Modifier.TRANSIENT, Modifier.VOLATILE)
    .create();
```

Gson의 @Expose

이 기능은 JSON에 대한 직렬화 및 역직렬화시 제외할 객체의 특정 필드를 마킹 할 수 있는 방법을 제공한다. 이 어노테이션을 사용하려면 new GsonBuilder().excludeFieldsWithoutExposeAnnotation().create()를 사용하여 Gson을 만들어야 한다. 생성된 Gson 인스턴스는 @Expose 어노테이션으로 표시되지 않은 클래스의 모든 필드를 제외한다.

사용자 정의 제외 전략

필드 및 클래스 타입을 제외하는 위의 방법이 제대로 동작하지 않는 경우 언제든지 고유한 제외 전략을 작성하여 Gson에 연결할 수 있다. 자세한 내용은 ExclusionStrategy JavaDoc을 참조하자.

다음 예제는 특정 @Foo 어노테이션으로 마킹된 필드를 제외하고 String 클래스의 최상위 타입 (또는 선언 된 필드 타입)을 제외하는 방법을 보여준다.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface Foo {
  // Field tag only annotation
}

public class SampleObjectForTest {
  @Foo private final int annotatedField;
  private final String stringField;
  private final long longField;
  private final Class<?> clazzField;

  public SampleObjectForTest() {
    annotatedField = 5;
    stringField = "someDefaultValue";
    longField = 1234;
  }
}

public class MyExclusionStrategy implements ExclusionStrategy {
  private final Class<?> typeToSkip;

  private MyExclusionStrategy(Class<?> typeToSkip) {
    this.typeToSkip = typeToSkip;
  }

  public boolean shouldSkipClass(Class<?> clazz) {
    return (clazz == typeToSkip);
  }

  public boolean shouldSkipField(FieldAttributes f) {
    return f.getAnnotation(Foo.class) != null;
  }
}
```

```
public static void main(String[] args) {
  Gson gson = new GsonBuilder()
      .setExclusionStrategies(new MyExclusionStrategy(String.class))
      .serializeNulls()
      .create();
  SampleObjectForTest src = new SampleObjectForTest();
  String json = gson.toJson(src);
  System.out.println(json);
}
```

출력 결과는 다음과 같다.

```
{"longField":1234}
```

JSON 필드 네이밍 지원

Gson은 표준 자바 필드 이름(즉, 소문자-sampleFieldNameInJava로 시작하는 카멜 케이스 이름)을 Json 필드 이름(즉, sample_field_name_in_java 또는 SampleFieldNameInJava)으로 변환하기 위해 기존에 일부 정의된 필드 이름 정책을 지원한다. 기존 정의된 이름 정책의 정보를 위해 FieldNamingPolicy 클래스를 보자.

또한 클라이언트가 필드 기반 마다 커스텀 이름을 정의할 수 있도록 하는 어노테이션 기반 전략을 가진다. 어노테이션 기반 전략은 유효하지 않은 이름이 어노테이션 값으로 제공된다면 "Runtime" 예외가 발생할 수 있는 필드 이름의 유효성 검증을 가진다.

다음은 두 가지 Gson 이름 정책 특성들을 사용하는 방법에 대한 예제이다.

```
private class SomeObject {
  @SerializedName("custom_naming") private final String someField;
  private final String someOtherField;

  public SomeObject(String a, String b) {
    this.someField = a;
    this.someOtherField = b;
  }
}

SomeObject someObject = new SomeObject("first", "second");
Gson gson = new GsonBuilder().setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE).create();
String jsonRepresentation = gson.toJson(someObject);
System.out.println(jsonRepresentation);
```

출력은

```
{"custom_naming":"first","SomeOtherField":"second"}
```

만약 커스텀 이름 정책 (해당 논의를 보자) 에 대한 필요성이 있다면, @SerializedName 어노테이션을 사용할 수 있다.

커스텀 직렬화 및 역직렬화 간 상태 공유하기

때로는 커스텀 직렬화/역직렬화 변환기 간에 상태를 공유해야 한다(이 토론을 참조하자). 이를 위해 다음 세 가지 전략을 사용할 수 있다.

1. 정적 필드 내에서 공유된 상태를 저장하기
2. 직렬화/역직렬화 를 부모 타입의 내부 클래스로 선언하고 부모 타입의 인스턴스 필드를 사용하여 공유 상태를 저장하기
3. 자바의 ThreadLocal 사용하기

1과 2는 스레드에 안전하지 않지만 3은 안전하다.

스트리밍

Gson의 객체 모델과 데이터 바인딩을 추가하여 stream을 작성하는데, stream으로부터 읽는데 Gson을 사용할 수 있다. 스트리밍과 객체 모델 접근을 결합하여 두 가지 접근 방법을 모두 활용할 수 있다.

Gson 설계의 문제

Gson을 설계하는 동안 직면한 문제에 대한 논의는 Gson 설계 문서를 참조하자. 또한 Json 변환에 사용할 수 있는 다른 Java 라이브러리와 Gson의 비교도 포함된다.

Gson의 향후 개선 사항

제안된 개선 사항의 최신 목록을 확인하거나 새로운 개선 사항을 제안하려면 프로젝트 웹 사이트의 Issue 섹션을 참조하자.

Gson 설계 문서

이 문서는 Gson을 설계하는 동안 직면한 문제를 제시한다. Gson에서 작업하는 고급 사용자 또는 개발자를 위한 것이다. Gson 사용법을 배우고 싶다면 해당 사용자 가이드를 참조하자.

역직렬화하는 동안 Json 트리 또는 대상 타입 트리 탐색

Json 문자열을 원하는 타입의 객체로 역직렬화 할 때 입력 트리 또는 원하는 타입의 타입 트리를 탐색할 수 있다. Gson은 후자의 접근 방식을 사용하여 대상 객체의 타입을 탐색한다. 이렇게 하면 예상하는 객체 타입만 인스턴스화하는 것을 엄격하게 제어 할 수 있다 (본질적으로 예상되는 "스키마"에 대한 입력 유효성 검사). 이렇게 하면 Json 입력에 있지만 예상하지 못한 추가 필드도 무시된다

Gson의 일부로, 우리는 모든 객체를 가져와 원하는 방문자를 호출하여 해당 필드를 탐색 할 수있는 범용 ObjectNavigator를 작성했다.

역직렬화 의미 체계보다 풍부한 직렬화 의미 체계 지원

Gson은 임의 컬렉션의 직렬화를 지원하지만 제네릭화 된 컬렉션만 역 직렬화 할 수 있다. 이는 Gson이 작성한 Json을 일부 경우 역 직렬화하지 못할 수 있음을 의미한다. 임의 타입의 Json 배열을 발견하면 개별 요소의 타입을 감지 할 방법이 없기 때문에 이는 주로 Java 타입 시스템의 제약사항이다. 제네릭 컬렉션만 지원하도록 직렬화를 제한하는 선택을 할 수 있었지만 그렇게 하지 않도록 선택할 수 있었다. 이는 라이브러리 사용자가 직렬화 또는 역직렬화에 관심이 있는 경우가 많지만 둘 다는 아니기 때문이다. 이러한 경우 직렬화 기능을 인위적으로 제한 할 필요가 없다.

제어 할 수 없고, 수정할 수 없는 클래스의 직렬화 및 역직렬화 지원

일부 Json 라이브러리는 필드 또는 메서드에 어노테이션을 사용하여 Json 직렬화에 사용해야 하는 필드를 나타낸다. 이 접근 방식은 본질적으로 JDK 또는 타사 라이브러리의 클래스 사용을 배제한다. 우리는 커스텀 직렬화 및 역직렬화의 개념을 정의하여 이 문제를 해결했다. 이 접근 방식은 새로운 것이 아니며, JAX-RPC 기술에서 기본적으로 동일한 문제를 해결할 때 사용되었다.

확인된 예외와 확인되지 않은 예외를 사용하여 구문 분석 오류 표시

파싱 실패를 나타내기 위해 확인되지 않은 예외를 사용하기로 했다. 이것은 주로 클라이언트의 잘못된 입력으로 복구 할 수 없기 때문에 수행되며, 따라서 확인된 예외를 포착하도록 강제하면 catch() 블록에서 코드가 엉성해진다.

역직렬화를 위한 클래스 인스턴스 만들기

Gson은 Json 데이터를 필드로 역직렬화하기 전에 더미 클래스 인스턴스를 만들어야 한다. 그러한 인스턴스를 얻기 위해 Guice를 사용할 수 있었지만 Guice에 대한 의존성이 발생하게 된다. 게다가 Guice가 유효한 인스턴스를 반환 할 것으로 예상되기 때문에 잘못된 일을 했을 가능성이 있고, 더미 인스턴스를 만들어야 한다. 더 나쁜 것은 Gson이 모든 후속 Guice 주입에 대해 인스턴스를 수정하여 해당 인스턴스의 필드를 들어오는 데이터로 덮어 쓴다. 이것은 분명히 바람직한 행동이 아니다. 따라서 매개변수 없는 생성자를 호출하여 클래스 인스턴스를 만든다. 또한 원시 타입, 열거형, 컬렉션, 세트, 맵 및 트리를 특별한 경우로 처리한다.

수정할 수 없는 타입을 지원하는 문제를 해결하기 위해 커스텀 Instance Creator를 사용한다. 따라서 기본 생성자를 정의하지 않는 라이브러리 유형 (예 : Money 클래스)을 사용하려는 경우, 요청시 더미 인스턴스를 반환하는 인스턴스 생성자를 등록 할 수 있다.

필드와 게터를 사용하여 Json 요소 표시

일부 Json 라이브러리는 유형의 getter를 사용하여 Json 요소를 추론합니다. 일시적, 정적 또는 합성이 아닌 모든 필드 (상속 계층 구조)를 사용하도록 선택했습니다. 모든 클래스가 적절한 이름의 getter로 작성되지 않았기 때문에 이렇게 했다. 더욱이, getXXX 또는 isXXX는 속성을 나타내기보다는 의미론적일 수 있다.

그러나 속성을 지원하는 좋은 주장도 있습니다. 우리는 Json 필드를 나타내는 대체 매핑으로 속성을 지원하기 위해 후자의 버전에서 Gson을 향상시킬 계획이다다. 현재 Gson은 필드 기반이다.

Gson의 대부분의 클래스가 final로 표시되는 이유는 무엇입니까?

Gson은 유연한 직렬화 및 역직렬화 변환기를 제공하여 상당히 확장 가능한 아키텍처를 제공하지만 Gson 클래스는 특별히 확장 가능하도록 설계되지 않았다. 최종 클래스가 아닌 클래스를 제공하면 사용자가 합법적으로 Gson 클래스를 확장 할 수 있었으며, 그 동작이 모든 후속 개정판에서 작동 할 것으로 기대했다. 우리는 클래스를 final로 표시하고 확장성을 허용하는 좋은 사용 사례가 나타날 때까지 기다림으로써 이러한 사용 사례를 제한하기로 결정했다. 클래스를 final로 표시하면 Java 컴파일러 및 가상 머신에 추가 최적화 기회를 제공하는 사소한 이점도 있습니다.

Gson에서 내부 인터페이스와 클래스가 많이 사용되는 이유는 무엇입니까?

Gson은 내부 클래스를 실질적으로 사용한다. 많은 공용 인터페이스도 내부 인터페이스다 (예시로 JsonSerializer.Context 또는 JsonDeserializer.Context 참조). 이것은 주로 스타일의 문제로 수행된다. 예를 들어, JsonSerializer.Context를 최상위 클래스 JsonSerializerContext로 옮겼지만 그렇게 하지 않도록 선택할 수 있다. 그러나 이름을 번갈아 변경해야하는 이유를 알려준다면, 이 철학을 변경할 수 있다.

Gson을 구성하는 두 가지 방법을 제공하는 이유는 무엇입니까?

Gson은 new Gson()을 호출하거나 GsonBuilder를 사용하는 두 가지 방법으로 구성 할 수 있다. 기본 옵션을 사용하고 코드 작성을 빠르게 진행하려는 Gson의 간단한 사용 사례를 처리하기 위해 인자없는 간단한 생성자를 제공하기로 결정했다. Formatter, Version Control 등과 같은 옵션을 사용하여 Gson을 구성해야 하는 다른 모든 상황에서는 빌더 패턴을 사용한다. 빌더 패턴을 사용하면 기본적으로 Gson의 생성자 매개 변수가 되는 항목에 대해 여러 선택적 설정을 지정할 수 있다.

Gson과 대체 접근 방식 비교

이러한 비교는 Gson을 개발하는 동안 수행 되었으므로 2007년 중반에서 후반까지 거슬러 올라갑니다.

Gson과 org.json 라이브러리 비교

org.json은 클래스에서 toJson() 메서드를 작성하는 데 사용할 수 있는 훨씬 낮은 수준의 라이브러리다. Gson을 직접 사용할 수 없는 경우 (반영과 관련된 플랫폼 제한 때문일 수 있음) org.json을 사용하여 각 객체에서 toJson 메서드를 직접 코딩 할 수 있다.

Gson과 org.json.simple 라이브러리 비교

org.json.simple 라이브러리는 org.json 라이브러리와 매우 유사하므로 상당히 낮은 수준이다. 이 라이브러리의 핵심 문제는 예외를 잘 처리하지 못한다는 것이다. 어떤 경우에는 예외를 먹는 것처럼 보이지만 다른 경우에는 예외가 아닌 "오류"가 발생한다.

커스텀 직렬화와 역직렬화

때로는 기본 표현이 원하는게 아닐 것이다. 라이브러리 클래스 (DateTime, 등등) 을 다룰 때 이런 경우가 종종 있다. Gson은 개인의 커스텀 직렬화 변환기와 역직렬화 변환기 등록할 수 있다. 이건 다음 두 부분을 정의함으로써 끝난다.

- Json Serializers : 객체에 대한 커스텀 직렬화 정의가 필요하다.
- Json Deserializers : 유형에 대한 커스텀 역직렬화 정의가 필요하다.
- Instance Creators : 인자가 없는 생성자가 사용되거나 deserializer가 등록된다면 필요하지 않는다.

```
GsonBuilder gson = new GsonBuilder();
gson.registerTypeAdapter(MyType2.class, new MyTypeAdapter());
gson.registerTypeAdapter(MyType.class, new MySerializer());
gson.registerTypeAdapter(MyType.class, new MyDeserializer());
gson.registerTypeAdapter(MyType.class, new MyInstanceCreator());
```

registerTypeAdapter 호출은 유형 어댑터가 이러한 인터페이스들 중 하나 이상을 구현하는지 확인하고 모든 인터페이스에 대해 등록한다.

직렬화 변환기 작성

JodaTime DateTime 클래스에 대한 커스텀 직렬화 변환기를 작성하는 방법에 대한 예제이다.

```
private class DateTimeSerializer implements JsonSerializer<DateTime> {
  public JsonElement serialize(DateTime src, Type typeOfSrc, JsonSerializationContext context) {
    return new JsonPrimitive(src.toString());
```

```
  }
}
```

Gson은 직렬화하는 동안 DateTime 객체 내에서 실행될 때 serialize()를 호출한다.

역직렬화 변환기 작성

JodaTime DateTime 클래스에 대한 커스텀 역직렬화 변환기를 작성하는 방법에 대한 예제이다.

```
private class DateTimeDeserializer implements JsonDeserializer<DateTime> {
  public DateTime deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context)
      throws JsonParseException {
    return new DateTime(json.getAsJsonPrimitive().getAsString());
  }
}
```

Gson은 DateTime 객체 내에서 JSON 문자열 조각이 역직렬화를 필요로 할 때 deserialize를 호출한다.

직렬화 변환기와 역직렬화 변환기에 미세한 점

종종 원시(raw) 유형에 일치하는 모든 제네릭 유형에 대해 단일 핸들러를 등록하길 원한다.

- 예를 들어, id 표현/번역에 대해 Id 클래스를 가진다 가정하자.(즉, 내부 표현과 외부 표현)
- 모든 제네릭 유형에 대한 같은 직렬화를 있는 Id<T> 유형
  - 기본적으로 id값을 작성해야 한다.
- 역직렬화는 매우 유사하지만 정확하게 동일하지는 않다.
  - Id<T>의 인스턴스를 반환하는 새로운 Id(Class<T>, String)을 호출해야만 한다.

Gson은 이에 대해 단일 핸들러 등록하는 것을 지원한다. 특정 제네릭 유형(예, Id<RequiresSpecialHandling> 특별한 핸들링이 필요하다)에 대해 특정 핸들러를 등록할 수 있다. toJson()과 fromJson()에 대한 Type 파라미터는 같은 원시 유형에 일치하는 모든 제네릭 유형에 대한 단일 핸들러를 작성하는 것을 돕기 위한 제네릭 유형 정보를 포함한다.