UD11 - Serialización y deserialización en Java

Contenido

Introducción a la serialización

La serialización es un concepto fundamental en la programación que se refiere al proceso de convertir un objeto en un formato que puede ser almacenado o transmitido y luego reconstruido en un objeto de nuevo. Este proceso es útil en una variedad de contextos, como la persistencia de datos, la programación de redes y la copia de objetos.

Imagina que tienes un objeto en tu programa que representa a un estudiante con su nombre y calificaciones. Si quieres guardar este objeto en un archivo para poder recuperarlo más tarde, necesitarás convertirlo a un formato que pueda ser escrito en un archivo. Este proceso de conversión es lo que llamamos serialización.

Del mismo modo, si quieres enviar este objeto a través de una red a otro programa o máquina, necesitarás convertirlo a un formato que pueda ser transmitido a través de la red. De nuevo, este proceso de conversión es la serialización.

Después de que el objeto ha sido serializado y almacenado o transmitido, necesitaremos convertirlo de nuevo a un objeto en algún momento. Este proceso de conversión inversa se llama deserialización.

En Java, la serialización y deserialización se pueden realizar utilizando varias técnicas y herramientas, incluyendo las clases incorporadas en Java, así como varias bibliotecas externas.

Serialización con Java nativo

Java proporciona soporte incorporado para la serialización de objetos a través de la interfaz “java.io.Serializable”. Para hacer que una clase sea serializable en Java, simplemente necesitamos implementar esta interfaz. No hay métodos que implementar, ya que “Serializable” es una interfaz de marcador que simplemente le dice a Java que una clase puede ser serializada.

Por ejemplo, considera la siguiente clase `Estudiante`:

```
import java.io.Serializable;

public class Estudiante implements Serializable {
    private String nombre;
    private int edad;

    // getters y setters
}
```

Para serializar un objeto de la clase “Estudiante”, podemos usar la clase “ObjectOutputStream” de Java:

```
Estudiante estudiante = new Estudiante();
estudiante.setNombre("Juan");
estudiante.setEdad(20);

try (FileOutputStream fileOut = new
FileOutputStream("estudiante.ser");
     ObjectOutputStream out = new ObjectOutputStream(fileOut)) {
    out.writeObject(estudiante);
} catch (IOException i) {
    i.printStackTrace();
}
```

Este código escribe el objeto “estudiante” a un archivo llamado “estudiante.ser”.

Para deserializar el objeto “Estudiante”, podemos usar la clase “ObjectInputStream” de Java:

```
Estudiante estudiante = null;

try (FileInputStream fileIn = new FileInputStream("estudiante.ser");
     ObjectInputStream in = new ObjectInputStream(fileIn)) {
    estudiante = (Estudiante) in.readObject();
} catch (IOException i) {
    i.printStackTrace();
} catch (ClassNotFoundException c) {
    System.out.println("Clase Estudiante no encontrada");
    c.printStackTrace();
}
```

Este código lee el objeto “Estudiante” del archivo “estudiante.ser”.

Es importante tener en cuenta que no todos los objetos pueden ser serializados. Solo los objetos de clases que implementan la interfaz “Serializable” (o su subinterfaz “Externalizable”) pueden ser serializados. Además, todos los campos en el objeto deben ser serializables. Si un campo no es serializable, debes marcarlo como “transient”.

Problemas con la serialización de Java

Aunque la serialización en Java es un concepto poderoso y útil, también tiene sus desafíos y problemas. Aquí hay algunos de los más comunes:

- Incompatibilidad de versiones: Si una clase serializable cambia (por ejemplo, si agregas, eliminas o cambias el tipo de un campo), entonces los objetos serializados de la versión anterior de la clase pueden no ser compatibles con la nueva versión de la clase. Java proporciona una solución parcial a este problema a través del uso de un serialVersionUID, pero esto no siempre es suficiente para resolver todas las incompatibilidades.
- Seguridad: La serialización puede ser una fuente de vulnerabilidades de seguridad, ya que permite que un atacante cree o modifique objetos de manera que el programador no anticipó. Por ejemplo, un atacante podría crear un objeto con un estado que viola las invariantes de la clase, o podría modificar un objeto serializado para ejecutar código arbitrario cuando se deserializa.
- Rendimiento: La serialización y deserialización pueden ser operaciones costosas en términos de rendimiento, especialmente para objetos grandes o complejos. Esto puede ser un problema en aplicaciones donde el rendimiento es crítico.
- Tamaño: Los objetos serializados pueden ser significativamente más grandes que los objetos en memoria, lo que puede ser un problema si estás trabajando con objetos grandes o si tienes restricciones de espacio de almacenamiento o ancho de banda.
- Transitoriedad: No todos los campos de un objeto son serializables. Si tienes campos que no son serializables o que no deben ser serializados por alguna razón (por ejemplo, porque contienen datos temporales o sensibles), debes marcarlos como transient. Sin embargo, esto puede complicar el diseño de tu clase y puede hacer que sea más difícil razonar sobre el estado de un objeto después de la deserialización.

Estos son solo algunos de los problemas con la serialización en Java. Afortunadamente, hay varias técnicas y herramientas disponibles para ayudar a mitigar estos problemas.

Ejercicio

Este ejercicio consiste en escribir un programa en Java que extraiga información de una URL de búsqueda de google, extraerá el protocolo, el dominio, ruta de acceso, y termino de búsqueda.

Por ejemplo:

```
String uri = "https://www.google.com/search?q=Java+21";
```

Salida:

Protocolo= https

Dominio= www.google.com

Ruta de acceso= search

Termino de búsqueda= Java+21 (o Java 21 si sustituyes el + por un espacio.)

Librerias externas

En el desarrollo de software, rara vez creamos todo desde cero. A menudo, utilizamos código que otras personas han escrito y compartido para realizar tareas comunes o complejas. Este código compartido se conoce como una "librería".

Una librería externa es una librería que no forma parte del lenguaje de programación estándar o del entorno de ejecución. Por ejemplo, en Java, las clases y las interfaces que vienen con el JDK son parte de la biblioteca estándar de Java, mientras que algo como la biblioteca Jackson para trabajar con JSON es una librería externa.

Para usar una librería externa en nuestro proyecto, generalmente necesitamos descargarla e incluirla en nuestro classpath. Esto puede ser un proceso manual, pero a menudo se automatiza con herramientas de gestión de dependencias como Maven o Gradle.

Maven es una herramienta de gestión de proyectos y comprensión de proyectos. Puede usarse para construir y gestionar cualquier proyecto basado en Java. Maven ayuda a manejar las dependencias, es decir, las librerías externas que nuestro proyecto necesita.

Para usar Maven, especificamos nuestras dependencias en un archivo llamado pom.xml. Maven luego descarga automáticamente estas dependencias y las incluye en el classpath de nuestro proyecto.

Aquí vemos un ejemplo de cómo se ve un archivo pom.xml:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.9.8</version>
    </dependency>
  </dependencies>
  ...
</project>
```

Gradle es otra herramienta de gestión de proyectos y de sistemas de construcción que se utiliza en el desarrollo de software. Al igual que Maven, Gradle puede manejar la descarga y la inclusión de dependencias en nuestro proyecto. Sin embargo, Gradle utiliza un lenguaje de script basado en Groovy en lugar de XML para su configuración.

Aquí tienes un ejemplo de cómo se ve un archivo build.gradle:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile 'com.fasterxml.jackson.core:jackson-databind:2.9.8'
}
```

Tanto Maven como Gradle son herramientas poderosas que pueden hacer mucho más que simplemente manejar dependencias, pero la gestión de dependencias es una de sus características más utilizadas y útiles.

Nota: En Java se sigue usando maven, mientras que en el desarrollo de aplicaciones móviles se usa más Gradle. https://mvnrepository.com/

Introducción a JSON

JSON, que significa JavaScript Object Notation, es un formato de texto ligero para el intercambio de datos. Aunque originalmente se derivó de JavaScript, es un formato de datos independiente del lenguaje que es fácil de leer y escribir para humanos y fácil de analizar y generar para máquinas.

Un objeto JSON es un conjunto de pares de nombre/valor, donde el nombre es una cadena y el valor puede ser una cadena, un número, un objeto JSON, un array, un booleano o null. Aquí tienes un ejemplo de un objeto JSON que representa a un estudiante:

```
{
    "nombre": "Juan",
    "edad": 20,
    "asignaturas": ["Matemáticas", "Física", "Química"],
    "aprobado": true
}
```

JSON es muy similar a la sintaxis de los objetos en JavaScript. Sin embargo, JSON es un formato de datos, no un lenguaje de programación, por lo que no tiene variables, funciones, bucles o cualquier otra característica de los lenguajes de programación.

JSON es ampliamente utilizado para la serialización y transmisión de datos en aplicaciones web entre el servidor y el cliente. También es comúnmente utilizado para almacenar datos en archivos o bases de datos.

En Java, podemos trabajar con JSON utilizando varias bibliotecas, como Jackson, Gson y JSON-P. Estas bibliotecas nos permiten serializar objetos a JSON y deserializar JSON a objetos de una manera fácil y conveniente.

Serialización con Jackson

Jackson es una de las bibliotecas más populares para trabajar con JSON en Java. Proporciona una interfaz fácil de usar para serializar objetos a JSON y deserializar JSON a objetos.

Para usar Jackson, primero necesitamos agregarlo a nuestro proyecto. Si estás utilizando Maven, puedes hacerlo agregando la dependencia a tu archivo pom.xml

Una vez que Jackson está en nuestro proyecto, podemos usar la clase ObjectMapper para serializar y deserializar objetos. Aquí tienes un ejemplo de cómo podríamos serializar un objeto Estudiante a JSON:

```
import com.fasterxml.jackson.databind.ObjectMapper;

public class App {
    public static void main(String[] args) {
        Estudiante estudiante = new Estudiante();
```

```
        estudiante.setNombre("Juan");
        estudiante.setEdad(20);

        ObjectMapper objectMapper = new ObjectMapper();
        try {
            String json = objectMapper.writeValueAsString(estudiante);
            System.out.println(json);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Este código crea un objeto Estudiante, lo serializa a una cadena JSON utilizando ObjectMapper, y luego imprime la cadena JSON.

Para deserializar la cadena JSON de nuevo a un objeto Estudiante, podemos usar el método readValue de ObjectMapper:

```
import com.fasterxml.jackson.databind.ObjectMapper;

public class App {
    public static void main(String[] args) {
        String json = "{\"nombre\":\"Juan\",\"edad\":20}";

        ObjectMapper objectMapper = new ObjectMapper();
        try {
            Estudiante estudiante = objectMapper.readValue(json,
Estudiante.class);
            System.out.println(estudiante.getNombre());
            System.out.println(estudiante.getEdad());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Este código lee una cadena JSON, la deserializa a un objeto Estudiante utilizando ObjectMapper, y luego imprime el nombre y la edad del estudiante.

Introducción a XML

XML, que significa eXtensible Markup Language, es un lenguaje de marcado que define un conjunto de reglas para codificar documentos de manera que sean tanto legibles por humanos como procesables por máquinas. Aunque XML es similar a HTML en apariencia, no son lo mismo. Mientras que HTML se utiliza para describir la estructura y la apariencia de una página web, XML se utiliza para describir y transportar datos.

Un documento XML consta de elementos, cada uno de los cuales comienza con una etiqueta de inicio y termina con una etiqueta de fin. Los elementos pueden tener atributos y pueden contener otros elementos. Aquí tienes un ejemplo de un documento XML que representa a un estudiante:

```
<estudiante>
    <nombre>Juan</nombre>
    <edad>20</edad>
    <asignaturas>
        <asignatura>Matemáticas</asignatura>
        <asignatura>Física</asignatura>
        <asignatura>Química</asignatura>
```

```
    </asignaturas>
    <aprobado>true</aprobado>
</estudiante>
```

XML es muy descriptivo y autoexplicativo, lo que lo hace fácil de leer y entender para los humanos. Sin embargo, también es muy verboso, lo que puede hacer que los documentos XML sean grandes y lentos de procesar.

En Java, podemos trabajar con XML utilizando varias bibliotecas, como JAXP (Java API for XML Processing), JAXB (Java Architecture for XML Binding) y Xerces. Estas bibliotecas nos permiten parsear documentos XML, generar documentos XML, y mapear entre objetos Java y documentos XML.

Serialización con JAXB

JAXB, que significa Java Architecture for XML Binding, es una biblioteca que proporciona una forma de mapear entre objetos Java y documentos XML. Con JAXB, podemos serializar objetos a XML y deserializar XML a objetos de una manera fácil y conveniente.

Para usar JAXB, primero necesitamos agregarlo a nuestro proyecto. Si estás utilizando Maven, puedes hacerlo agregando la dependencia a tu archivo pom.xml

Una vez que JAXB está en nuestro proyecto, podemos usar las clases JAXBContext y Marshaller para serializar un objeto a XML. Aquí tienes un ejemplo de cómo podríamos serializar un objeto Estudiante a XML:

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;

public class App {
    public static void main(String[] args) {
        Estudiante estudiante = new Estudiante();
        estudiante.setNombre("Juan");
        estudiante.setEdad(20);

        try {
            JAXBContext context =
JAXBContext.newInstance(Estudiante.class);
            Marshaller marshaller = context.createMarshaller();
            marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
true);
            marshaller.marshal(estudiante, System.out);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Este código crea un objeto Estudiante, lo serializa a XML utilizando Marshaller, y luego imprime el XML.

Para deserializar el XML de nuevo a un objeto Estudiante, podemos usar la clase Unmarshaller:

```java
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;
import java.io.StringReader;

public class App {
    public static void main(String[] args) {
        String xml =
"<estudiante><nombre>Juan</nombre><edad>20</edad></estudiante>";

        try {
            JAXBContext context =
JAXBContext.newInstance(Estudiante.class);
            Unmarshaller unmarshaller = context.createUnmarshaller();
            Estudiante estudiante = (Estudiante)
unmarshaller.unmarshal(new StringReader(xml));
            System.out.println(estudiante.getNombre());
            System.out.println(estudiante.getEdad());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Este código lee una cadena XML, la deserializa a un objeto Estudiante utilizando Unmarshaller, y luego imprime el nombre y la edad del estudiante.