Kryo

Kryo is a fast and efficient binary object graph serialization framework for Java.

It is designed to be simple to use and highly configurable, with a small memory footprint.

he Kryo framework can serialize and deserialize objects from the standard Java library, including collections and enums, as well as objects not part of the standard library but registered with the framework.

Introduction to Kryo

Kryo is often used in applications where performance and memory usage are critical, such as in video games, data processing pipelines, or distributed systems.

Such performance and efficiency can be achieved due to the fact that Kryo serializes data in binary format and has internal optimizations, including object caching and field registration.

Kryo caches objects that have already been serialized to reduce the time needed to serialize them again.

Kryo requires the fields to be registered before they are serialized, which allows for efficient serialization and deserialization.

Unlike JSON or XML, the binary format is not human-readable. So, in addition to the obvious advantages, this is also a disadvantage of Kryo: this can make it more challenging to debug serialized data or inspect the contents of a serialized file manually.

Another downside is that Kryo is not thread-safe, because it uses internal data structures and caches that are not protected against concurrent access. For example, Kryo uses a cache to store information about previously serialized classes, and this cache is not thread-safe.

If multiple threads attempt to access the cache at the same time, they may overwrite each other's changes or cause other unexpected behavior.

Since Kryo is not thread-safe, it is important to use proper synchronization when sharing instances of Kryo among multiple threads or to use a thread-local instance of Kryo or a pool of reusable instances.

Furthermore, creating a new instance of Kryo can be relatively expensive, as it requires allocating memory and initializing various internal data structures,

such as a cache for storing information about previously serialized classes, which is populated by scanning the class path and analyzing the fields and properties of available classes.

These internal data structures are only necessary during the serialization/deserialization process and are not needed once the process is completed.

To avoid these issues, using a thread-local instance of Kryo or a pool of reusable Kryo instances that can be shared among multiple threads is advisable.

Object serialization and deserialization

Let's look at an example of serialization and deserialization using Kryo. To start using Kryo in your application with Maven, use this dependency entry in your pom.xml:

```
<dependency>
    <groupId>com.esotericsoftware</groupId>
    <artifactId>kryo</artifactId>
    <version>5.4.0</version>
</dependency>
```

To use Kryo in an application with Gradle, use the following dependency entry:

```
implementation group: 'com.esotericsoftware', name: 'kryo', version: '5.4.0'
```

Imagine we want to serialize and deserialize an object of a City class:

```
public class City {
    String cityName;
    Country country;

    public City() {}

    public City(String cityName, Country country) {
        this.cityName = cityName;
        this.country = country;
    }

    @Override
    public String toString() {
        return this.cityName + " is in " + this.country.countryName +
                "(" + this.country.continentName + ")";
    }

    @Override
    public boolean equals(Object o) {
```

```
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        City city = (City) o;
        return this.country.equals(city.country) &&
this.cityName.equals(city.cityName);
    }
}

public class Country {
    String countryName;
    String continentName;

    public Country() {}

    public Country(String countryName, String continentName) {
        this.countryName = countryName;
        this.continentName = continentName;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Country country = (Country) o;
        return this.countryName.equals(country.countryName)
                && this.continentName.equals(country.continentName);
    }
}
```

The following code shows how to serialize and deserialize an object with the help of Kryo:

```
public class Main {
    public static void main(String[] args) throws Exception {
        // Create a new instance of Kryo
        Kryo kryo = new Kryo();

        // All classes that Kryo will read or write must be registered beforehand
        kryo.register(City.class);
        kryo.register(Country.class);

        // Create a new instance of custom class
        City city = new City("Berlin", new Country("Germany", "Eurasia"));

        //try-with-resources
        try (Output output = new Output(new FileOutputStream("data.bin"))) {
            // Serialize the object to a file
            kryo.writeObject(output, city);
```

```
            // Object in array of byte
            byte[] bytes = output.toBytes();
            // Print the object byte array
            System.out.println("Byte array: " + Arrays.toString(bytes));
        } catch (IOException e) {
            System.err.println("An error occurred while writing to the file: " +
e.getMessage());
        }
        //try-with-resources
        try (Input input = new Input(new FileInputStream("data.bin"))) {
            // Deserialize the object from the file
            City city2 = kryo.readObject(input, City.class);
            // Show object after deserialization
            System.out.println("Object after deserialization: " + city2);
            // Show that serialized object is the same after deserialization
            System.out.println("Are objects equal before and after serialization: "
                    + city.equals(city2));
        } catch (IOException e) {
            System.err.println("An error occurred while reading from the file: " +
e.getMessage());
        }
    }
}
```

This is what is displayed in the console:
Byte array: [66, 101, 114, 108, 105, -18, 12, 69, 117, 114, 97, 115, 105, -31, 71, 101, 114, 109, 97, 110, -7]
Object after deserialization: Berlin is in Germany(Eurasia)
Are objects equal before and after serialization: true

It's very important to close the Input and Output classes after their use. When an instance of Input or Output is created, it opens an underlying stream or allocates a buffer in memory. These resources need to be freed after use. Closing the Input or Output instance releases these resources and prevents memory leaks.
The **try-with-resources** statement is a feature in Java that allows automatic resource management for Java objects that implement the java.lang.AutoCloseable interface. It ensures that the resources are properly closed, even if an exception is thrown

Class registration

we called the register(City.class) and register(Country.class) methods.

Kryo uses class registration to keep track of the classes that have been serialized and deserialized and to improve performance by reusing previously registered class information.

Kryo uses the unique ID assigned to the class during registration to identify the class when reading the serialized data.

If you try to serialize an object of a class that is not registered with Kryo, the serialization process will fail.

Kryo will not be able to generate a unique ID for the class or store information about the class's fields and properties. When trying to serialize the object, the program will throw the java.lang.IllegalArgumentException.

However, there is a method setRegistrationRequired(false) that allows disabling registration:

```
public class Main {
    public static void main(String[] args){
        Kryo kryo = new Kryo();

        // Disable registration
        kryo.setRegistrationRequired(false);

        City city = new City("Berlin", new Country("Germany", "Eurasia"));
        try (Output output = new Output(new FileOutputStream("data.bin"))) {
            kryo.writeObject(output, city);
            byte[] bytes = output.toBytes();
            System.out.println("Byte array: " + Arrays.toString(bytes));
        } catch (IOException e) {
            System.err.println("An error occurred while writing to the file: " +
e.getMessage());
        }

        // Show unregistered classes ID
        System.out.println("City class ID to identify an object's class when
serializing it : "
                + kryo.getRegistration(City.class).getId());
        System.out.println("Country class ID to identify an object's class when
serializing it : "
                + kryo.getRegistration(Country.class).getId());
    }
}
```

```
Byte array: [66, 101, 114, 108, 105, -18, 1, 0, 111, 114, 103, 46, 101, 120, 97, 109,
112, 108,
101, 46, 67, 111, 117, 110, 116, 114, -7, 69, 117, 114, 97, 115, 105, -31, 71, 101, 114,
109,
97, 110, -7]
City class ID to identify an object's class when serializing it : -1
Country class ID to identify an object's class when serializing it : -1
```

Output if we registered the Country and City classes:
Byte array: [66, 101, 114, 108, 105, -18, 12, 69, 117, 114, 97, 115, 105, -31, 71, 101, 114, 109,
97, 110, -7]
City class ID to identify an object's class when serializing it : 9
Country class ID to identify an object's class when serializing it : 10

When a class is registered, Kryo associates a unique integer ID with the class, which it uses to identify the class when serializing and deserializing objects. If a class is not registered with Kryo, the serializer will write the fully qualified class name of the unregistered class to the serialized data.

Class registration has a lot of advantages. By assigning a unique numeric ID to each registered class, the size of the serialized byte array is reduced by half.

Registering classes with Kryo allows it to cache information about the class and its fields so that it doesn't have to perform the same computation repeatedly.

This leads to faster serialization and deserialization times and lower memory usage.

Default serializer

Kryo has a default serializer for most common types, such as primitives, arrays, collections, and enums. By default, Kryo has more than 50 default serializers for various JRE classes.

One of Kryo's features is the ability to automatically choose the most appropriate serializer from a list of default serializers for a given object based on its type.

This can improve performance and reduce the amount of code required to handle serialization.

Let's modify the Country class and change the continentName field's type to enum:

```
class Country {
    String countryName;
    Continent continentName;

    public Country() {
    }

    public Country(String countryName, Continent continentName) {
```

```
        this.countryName = countryName;
        this.continentName = continentName;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Country country = (Country) o;
        return this.countryName.equals(country.countryName)
                && this.continentName.equals(country.continentName);
    }

    public enum Continent {
        AFRICA, NORTH_AMERICA, SOUTH_AMERICA, ANTARCTICA, EURASIA,
AUSTRALIA
    }
}
```

After registering Country.Continent.class and serializing the City object, we receive the following byte array:

```
Byte array: [66, 101, 114, 108, 105, -18, 12, 5, 71, 101, 114, 109, 97, 110, -7]
```

When serializing an object to a byte array, the size of the array will depend on the size of the object and the serialization method used. Enumerations can often be represented more compactly than other types of objects, which confirms the reduced size of the byte array.

Serializer addition

Kryo allows you to specify a custom serializer for a specific type during registration.

This means that when an object of that type is encountered during serialization or deserialization, the specified serializer will be used instead of the default one:

```
Kryo kryo = new Kryo();
kryo.register(SomeClass.class, new SomeSerializer());
kryo.register(AnotherClass.class, new AnotherSerializer());
```

To implement a custom serializer for a custom class in Kryo, you need to create a new class that implements the Serializer interface and overrides the write() and read() methods:

```
public class SomeSerializer extends Serializer<SomeClass> {
  @Override
  public void write(Kryo kryo, Output output, SomeClass someClass) {
    output.writeString(someClass.getValue());
```

```
    }
    @Override
    public SomeClass read(Kryo kryo, Input input, Class<? extends SomeClass>
type) {
        return new SomeClass(input.readString());
    }
```

Conclusion

- Kryo is a binary serialization library, meaning that the serialized data is not in a human-readable format. The serialized data is a binary representation of the object's state, which is designed to be compact and efficient for storage and transmission but is not easily readable by humans.
- Kryo is not a thread-safe library. This means that multiple threads accessing the same Kryo instance concurrently can lead to unexpected behavior or errors.
- It is generally best practice to register classes with Kryo to ensure that the serialization and deserialization process runs smoothly and efficiently.
- Kryo is a high-performance Java serialization library that provides efficient and flexible serialization and deserialization of objects. It offers features such as automatic selection of serializers based on object type and the ability to register custom serializers for specific types. This allows for improved control over the serialization process, as well as optimization of space and performance for specific types.