ByteBuddy

Runtime bytecode generation refers to the dynamic creation and execution of code within a running program.

This means that your program can adapt to various scenarios without the need for manual code changes and recompilation, relying solely on user input, runtime conditions, or evolving requirements. This technique empowers programs to become more flexible and adaptable.

Runtime bytecode generation usage example

Runtime bytecode generation can be a powerful tool in frameworks and libraries that require dynamic and flexible behaviour modification.

1. Dynamic proxies: frameworks such as Java's Dynamic Proxy API employ bytecode generation to construct proxy instances that can intercept method calls and override the default behavior.
2. Mocking and testing: runtime bytecode generation can be valuable for creating mock objects during unit testing. Testing frameworks like Mockito utilize bytecode generation to dynamically create proxy objects that mimic the behavior of dependencies, enabling isolated and controlled testing scenarios.
3. Dynamic code generation: frameworks like Apache Spark use bytecode generation to create custom functions and expressions for distributed data processing. This approach enables the efficient execution of tasks on extensive datasets, facilitating the processing of large-scale data more effectively.

Overall, runtime bytecode generation is a powerful technique that enables frameworks and libraries to offer dynamic and customizable behavior, such as creating proxies, overriding the default behavior, and creating data model classes.

Generating bytecode dynamically has historically been a challenging task, although recent advancements have made it more accessible.

Currently, there are a few prominent libraries available for bytecode generation: **ASM, cglib, Javassist.**
These libraries are specifically designed to write and modify bytecode instructions within Java code.

However, using them requires a solid understanding of bytecode and these libraries present difficulties in terms of usability and testing compared to Java

code.
Additionally, not all of them support newer Java features such as annotations, generics, default methods, and lambdas.

Byte Buddy is an innovative library that offers a solution to this challenge. Its primary objective is to democratize runtime code generation for developers who may not possess extensive familiarity with Java instructions.

Compare to other approaches: reflection

When considering runtime techniques for code manipulation, one approach that comes to mind is the use of the **reflection** API.
Reflection is a powerful feature that allows for the examination of code entities such as classes, methods, and fields during runtime. It provides the capability to inspect and modify these elements dynamically, even if they were not known or accessible at compile time.

However, it's important to understand the difference between reflection and bytecode generation approaches.
Both techniques enable runtime manipulation of code, but they serve different purposes. Reflection focuses on inspecting and modifying existing code entities, whereas bytecode generation is concerned with dynamically creating new code.

While both reflection and bytecode generation enable runtime code manipulation, there are some drawbacks to using reflection.
First, reflection is slower compared to direct method invocations or field accesses. The extra steps involved in looking up methods and executing additional reflection code introduce overhead and result in decreased performance.
Second, reflection is not type-safe. The compiler cannot verify if the reflection code is using the correct types of objects, which can lead to errors that are only discovered at runtime.

On the other hand, runtime code generation overcomes these issues.
It allows for the dynamic creation of code at runtime, combining the flexibility of dynamic languages with static type-checking.
This means that developers can still benefit from the strong typing provided by Java while enjoying the advantages of dynamic code generation.

Example: class creation with Byte Buddy

First, we need to include the Byte Buddy dependency in the project. Here's an example of a Maven dependency:

```
<dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy</artifactId>
    <version>1.14.5</version>
</dependency>
```

Gradle dependency:

```
implementation group: 'net.bytebuddy', name: 'byte-buddy', version: '1.14.5'
```

Before moving on to the creation of a dynamic class example, let’s create a helper class – GreetClass, with one simple method:

```
package org.example;

public class GreetClass {
    public static String sayHello() {
        return "Hello from method!";
    }
}
```

Let’s create a class dynamically:

```
package org.example;

import net.bytebuddy.ByteBuddy;
import net.bytebuddy.implementation.FixedValue;
import net.bytebuddy.matcher.ElementMatchers;

public class Main {
    public static void main(String[] args) throws Exception {
        Class<?> type = new ByteBuddy()
                .subclass(Object.class)
                .name("MyCustomClassName")
                .defineMethod("myCustomMethodName", String.class,
Modifier.PUBLIC)
                .intercept(MethodDelegation.to(GreetClass.class))
                .defineField("myCustomFieldName", String.class, Modifier.PUBLIC)
                .make()
                .load(
                        Main.class.getClassLoader(),
ClassLoadingStrategy.Default.WRAPPER)
                .getLoaded();

        Object instance = type.getDeclaredConstructor().newInstance();

        Method m = type.getDeclaredMethod("myCustomMethodName", null);
```

```
        System.out.println(m.invoke(instance));

        Field fieldX = type.getDeclaredField("myCustomFieldName");
        fieldX.set(instance, "Hello from field!");
        System.out.println(fieldX.get(instance));
    }
}
```

Output:
Hello from method!
Hello from field!

1. The code uses the ByteBuddy class to create a new subclass of Object. This is achieved by calling the subclass method on ByteBuddy and passing Object.class as the superclass.
2. The name() method is used to specify the name of the dynamically created class. In this case, the name MyCustomClassName is assigned.
3. The defineMethod() method is used to define a new method within the dynamically created class. In this case, the method name is myCustomMethodName, with a return type of String and the public modifier.
4. The intercept() method is used to specify how the defined method should be intercepted. In our case, we intercept the method call and delegate it to the GreetClass class.
5. The defineField() method is used to define a new field. Here, a field named myCustomFieldName is defined with a type of String and the public modifier.
6. The make() method is used to generate the bytecode for the dynamic class.
7. The load() method is called on the dynamic class object, passing Main.class.getClassLoader(). The class is loaded into the JVM along with specifying the class loading strategy.
8. The getLoaded() method is used to return an instance of the loaded class.
9. A new instance of the loaded class is created by calling getDeclaredConstructor().newInstance().
10. Then, a reference to the myCustomMethodName method is obtained using getDeclaredMethod().
11. The method is invoked on the instance using the invoke() method, and the result is printed.
12. Then, a reference to the field myCustomFieldName is obtained using type.getDeclaredField("myCustomFieldName"). Finally, the value is assigned using the set() method of the Field class, and the result is printed.

Conclusion

- Bytecode generation allows for the dynamic creation of code at runtime.
- nstead of manipulating existing code entities, like in reflection, bytecode generation involves generating new code instructions on the fly.
- This approach is often used in advanced scenarios where code needs to be generated dynamically based on certain conditions or requirements.
- Such as implementing dynamic proxies, creating mock objects for testing, or overriding default behavior.