Runtime Code Generation with Byte Buddy

Hello World, Byte Buddy

ElementMatchers and Implementations

Mocking

AOP Agent with Byte Buddy

Using an Agent Without -javaagent

Conclusion

TOOLS

Runtime Code Generation with Byte Buddy

Create agents, run tools before main() loads, and modify classes on the fly.

by Fabian Lange

November 1, 2015

An often overlooked feature of the Java platform is the ability to modify a program's bytecode before it is executed by the JVM's interpreter or just-in-time (JIT) compiler. While this capability is used by tools, such as profilers and libraries that do object-relational mapping, it is rarely used by application developers. This represents untapped potential, because generating code at runtime allows for easy implementation of cross-cutting concerns such as logging or security, changing the behavior of third-party libraries—sometimes in the form of mocking—or writing performance data collection agents.

Unfortunately, generating bytecode at runtime has been difficult until recently. There are presently three major libraries for generating bytecode:

- ASM
- cglib
- Javassist

These libraries were all designed to write and modify specific bytecode instructions from Java code. But to be able to use them, you need to understand how bytecode works, which is quite different than understanding Java source code. In addition, these libraries are harder to use and test than Java code, because the Java compiler cannot verify whether, for example, the argument order of a method call matches its signature or whether it violates the *Java Language Specification*. Lastly, due to their age, these libraries do not all support the new Java features, such as annotations, generics, default methods, and lambdas.

The following example illustrates how you would implement a method that calls another static method with a single string parameter using the ASM library:

```
methodVisitor.visitVarInsn(Opcodes.ALOAD, 0);
methodVisitor.visitMethodInsn(
   Opcodes.INVOKESTATIC
   "com/instana/agent/Agent"
   "record"
   "(Ljava/lang/String;)V"
)
```

cglib and Javassist are not much different. They all require usage of bytecode and String representation of signatures, which as you can see looks more like assembly language, rather than Java.

Byte Buddy is a new library that takes a different approach to solving this problem. Byte Buddy's mission is to make runtime code generation accessible to developers who have little to no knowledge of Java instructions. The library also aims to support all Java features, and is not limited to generating dynamic implementations for interfaces, which is the approach used in the JDK's built-in proxy utilities. The Byte Buddy API abstracts away all bytecode operators behind plain old Java method calls. However, it retains a backdoor to the ASM library, on top of which Byte Buddy is implemented.

Note: All the examples in this article are using the 0.6 API of Byte Buddy.

Hello World, Byte Buddy

The following HelloWorld example from the Byte Buddy documentation (see **Listing 1**) presents everything you need to create a new class at runtime in a concise way.

Listing 1.

```
Class<? extends Object> clazz = new ByteBuddy
   .subclass(Object.class)
   .method(ElementMatchers.named("toString"))
   .intercept(FixedValue.value("Hello World!"
   .make()
   .load(getClass().getClassLoader(),
      ClassLoadingStrategy.Default.WRAPPER)
   .getLoaded();
assertThat(clazz.newInstance().toString(),
      is("Hello World!"));
```

All of Byte Buddy's APIs are builder-style fluent APIs supporting the functional style. You start off by telling Byte Buddy which class you want to subclass. While in this example you simply subclass `Object` you could subclass any non-final class, and

Byte Buddy will ensure that the generic return type will be `Class<? extends SuperClass>`. Now that you have a builder for your subclass, you can tell Byte Buddy to intercept calls to a method that is named `toString` and return a fixed value instead of calling the method that is already defined by `java.lang.Object`.

You might wonder about the term *intercept* here. Usually when you subclass something, you typically use the term *override* when you change the implementation of a superclass method in a subclass. *Intercept* is a term from aspect-oriented programming (AOP), which describes a more powerful concept of “what to do” when a method is called.

After you finish declaring how the subclass behaves, you invoke `make` to get a so-called `Unloaded` representation of your class. This representation behaves like a `.class` file and, in fact, it even supports functions to store the class file.

Finally, as shown in **Listing 1**, you load the class using a class loader and get a reference to the loaded class. When getting started with Byte Buddy, the `ClassLoadingStrategy` used to do this does not usually matter. However, there are situations in which you need a specific class loader to load the new class for visibility purposes or for enforcing a specific loading order.

Note that a class generated by Byte Buddy is indistinguishable from regular classes. Unlike other libraries or proxies, there are no traces left behind. The generated code fully resembles the code that a Java compiler would create for implementing such a subclass.

ElementMatchers and Implementations

When you use Byte Buddy to add or change behavior of classes, the most common task is to look up fields, constructors, and methods. To ease these tasks Byte Buddy comes with plenty of useful predefined `ElementMatcher`s, such as `hasParameter()` and `isAnnotatedWith()`, which check the method signature. It also has convenience aliases such as `isEquals()` and `isSetter()`, which use common Java naming patterns to match the method name. Using the predefined matchers allows for a concise description of the methods to intercept, which would otherwise be quite verbose to write. Additionally, it is possible to implement a custom `ElementMatcher` to cover any more complex use case.

Additionally, there exist many predefined replacement implementations to be used in `intercept()`. Two examples are `MethodCall`, which can invoke a different method using parameters, and `Forwarding`, which uses the identical parameters to call the same method on another object.

An even more powerful interception mechanism is represented by `MethodDelegation`: When delegating to a method, you can first execute your custom code, and then delegate the call to the original implementation. Additionally, you can also dynamically access the information of the original call site using the `@Origin` annotation, as shown in **Listing 2**. When delegating to other methods, you can also dynamically access the information of the original call site, as shown next.

Listing 2.

```
public static class Agent {
   public static String record(@Origin Method
      System.out.println(m + " called");
   }
}

Class<?> clazz = new ByteBuddy()
   .subclass(Object.class)
   .method(ElementMatchers.isConstructor())
   .intercept(MethodDelegation
      .to(Agent.class)
      .andThen(SuperMethodCall.INSTANCE))
   // & make instance;
```

`MethodDelegation` automatically looks up the best match of method signatures in case multiple interception targets are available. While the lookup is powerful and can be customized, I recommend keeping the lookup simple and understandable. After the method has been invoked, the original call continues, thanks to `andThen(SuperMethodCall.INSTANCE)`.

The target method can take a couple of annotated parameters. To access the arguments of the originating method, you can use `@Argument(position)` or `@AllParameters`. To obtain information about the originating method itself, you can use `@Origin`. The type of that parameter can be `java.lang.reflect.Method`, `java.lang.Class`, or even `java.lang.invoke.MethodHandle` (the latter, if used with Java 7 or later). These arguments provide information about where the method has been called from, which could be useful for debugging, or even about taking different code paths, in the event that the same method is an interception target for multiple methods.

To call the originating method or its super method from the target method, Byte Buddy provides `@DefaultCall` and `@SuperCall` parameters.

Mocking

Sometimes you want to write a unit test for a scenario that can happen at runtime, but you cannot provoke that scenario reliably for the purpose of the test, if at all. For instance, in **Listing 3**, the random number generator needs to produce a specific result for you to test the control flow.

Listing 3.

```
public class Lottery {
   public boolean win() {
      return random.nextInt(100) == 0;
   }
}

Random mockRandom = new ByteBuddy()
   .subclass(Random.class)
   .method(named("nextInt"))
   .intercept(value(0))
   // & make instance;

Lottery lottery = new Lottery(mockRandom);
assertTrue(lottery.win());
```

Byte Buddy provides various kinds of interceptors, so writing mocks, or spies, is easy. However, for more than a few mocks, I would recommend switching to a dedicated mocking library. In fact, version 2 of the popular mocking library Mockito is currently being rewritten to be based on Byte Buddy.

So far, I have used `subclass()` to create what is essentially a subclass on steroids. Byte Buddy has two other modes of operation: `rebase` and `redefine`. Both options change the implementation of the specified class; while `rebase` maintains existing code, `redefine` overwrites it. However, these modifications come with a limitation: to change already loaded classes, Byte Buddy needs to work as a Java agent (more on that shortly).

For usage in unit testing or other special cases in which you can ensure that Byte Buddy loads a class for the first time, you can change the implementation during load. For that, Byte Buddy supports a concept called `TypeDescription`, which represents Java classes in an unloaded state. You can populate a pool of them from the (not yet loaded) classpath and modify classes before loading them. For example, I can modify the `Lottery` class in **Listing 3**, as shown in **Listing 4**.

Listing 4.

```
TypePool pool = TypePool.Default.ofClassPath(
new ByteBuddy()
   .redefine(pool.describe("Lottery")
   .resolve(), ClassFileLocator.ForClassLoade
   .method(ElementMatchers.named("win"))
   .intercept(FixedValue.value(true))
   // & make and load;

assertTrue(new Lottery().win());
```

Note: You cannot use `Lottery.class` for the call to `describe` here, because this would load the class before Byte

Buddy can rewrite it. Once a Java class is loaded, it is not normally possible to unload that class.

AOP Agent with Byte Buddy

In the following example, I create a performance monitoring and logging agent. It will intercept calls to JAX-WS endpoints and print how long the call took. Such an agent needs to follow conventions explained in the Javadoc for `java.lang.instrument`. It is launched using the `-javaagent` command line argument and executed before the actual `main` method (hence, the name `premain`). Usually agents install a hook for themselves, which is triggered before the regular program loads classes. This bypasses the limitation of not being able to change loaded classes. Agents are stackable, and you can use as many as you like. **Listing 5** shows the code for an agent.

Listing 5.

```
public class Agent {
   public static void premain(String args, In
      new AgentBuilder.Default()
         .rebase(isAnnotatedWith(Path.class))
         .transform((b, td) ->
            b.method(
               isAnnotatedWith(GET.class)
               .or(isAnnotatedWith(POST.class
            .intercept(to(Agent.class)))
         .installOn(inst);
}

@RuntimeType public static Object profile(@Or
   @SuperCall Callable<?> c)
   throws Exception {
      long start = System.nanoTime();
      try {
         return c.call();
      } finally {
         long end = System.nanoTime();
         System.out.println("Call to " + m +
            + (end - start) +" ns");
      }
   }
}
```

After obtaining a default `AgentBuilder`, I tell it which classes it should `rebase`. This example will modify only classes having the `javax.ws.rs.Path` annotation. Next, I tell the builder how to `transform` those classes. In this example, the agent will intercept calls to either `GET` or `POST` annotated methods and delegate to the `profile` method. For this to work, the agent needs to be hooked into the `Instrumentation` using `installOn()`.

The profile method itself uses three annotations: `RuntimeType`, to tell Byte Buddy that the return type `Object` needs to be adjusted to the real return type used by the method it intercepts; `Origin`, to obtain a reference to the actual method intercepted, which is used to print its name; and `SuperCall`, to actually perform the original method call. In contrast to the previous example, I need to perform the super call myself, because I want to be able to have my code executed before and after the method call—so that I can perform the timing.

Comparing the way Byte Buddy implements method interception to the default Java `InvocationHandler`, you can see that the Byte Buddy method is much more optimized due to the fact that the interception will pass in only the required arguments, while `InvocationHandler` must fulfill the following interface:

```
Object invoke(Object proxy, Method method, Ob
```

This benefit is especially noticeable for primitive arguments or return types, which need to be autoboxed. The additional `RuntimeType` annotation causes Byte Buddy to reduce any boxing to a minimum. Even though the JVM mostly optimizes away simple boxings, this is not always true for complex interfaces such as that of the `InvocationHandler`.

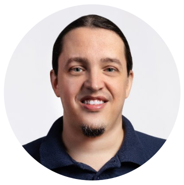
Using an Agent Without -javaagent

Using an agent to generate and modify code at runtime is a powerful technique; however, forcing the `-javaagent` argument to make it work is sometimes inconvenient. Byte Buddy comes with a handy convenience feature that uses the Java Attach API, which originally was designed to load diagnostic tooling at runtime. It attaches the agent to the currently running JVM. You need the additional `byte-buddy-agent.jar` file, which contains the utility class `ByteBuddyAgent`. With that, you invoke `ByteBuddyAgent.installOnOpenJDK()`, which does the same thing that starting the JVM with `-javaagent` did. The only other difference with this approach is that you do not invoke `installOn (inst)`, but rather you invoke `installOnByteBuddyAgent()`.

Conclusion

Despite the existence of dynamic proxies in the JDK and three popular third-party bytecode-manipulation libraries, Byte Buddy fills an important gap. Its fluent API uses generics, so you do not lose track of the actual type you are modifying, which can easily happen using other approaches. Byte Buddy also comes with a rich set of matchers, transformers, and implementations, and it enables their use via lambdas, which results in relatively concise and readable code.

As a result, Byte Buddy is fully understandable by developers who are not accustomed to reading bytecodes and working at low levels. With the upcoming version 0.7, Byte Buddy will support all the infrastructure around generic types. This way, Byte Buddy allows for easy interaction with generic types and type annotations even at runtime. As someone who writes a lot of bytecode-handling code, I both recommend and use this library. [Byte Buddy received a Duke's Choice Award at 2015's JavaOne conference. —*Ed*]



Fabian Lange

Fabian Lange (@CodingFabian) is a lead agent developer and performance geek at Instana, where he is building IT operations solutions. He is also a JavaOne Rock Star speaker.