Tracing Memory Accesses

Jmtrace supports running a JAR tracing all the shared memory access. It is written in Kotlin based on Javassist library for editing bytecodes in Java.

Algorithm

Steps

1. Use Javassist `ClassPool` api to load a JAR;
2. Add a written `Translator` instance for editing bytecodes;
3. Find the main class of this JAR, and run it with Javassist `Loader` api.

Translator

When running JAR, Javassist will trigger the `onLoad` method before each class being loaded. Then, we can modify the bytecode of this class in the `onLoad` method.

Iterate all the methods in the comming class, and then use `CtMethod::instrument(CodeConverter)` api to modify shared memory access. Generally, there are two types of shared memory access, array and field.

Array Access

There are 7 types of array in Java:

- `byte` / `boolean`
- `char`
- `double`
- `float`
- `int`
- `long`
- `java.lang.Object`

Before running JAR, I manually create a new class `__MTrace_Array__` to handle array access statements.

Generate read / write methods for each type of these 7 types of array. These generated methods look like:

```java
// Handle read int array
public static int read_int(java.lang.Object obj, int index) {
  long threadId = Thread.currentThread().getId();
  String objId = Integer.toHexString(System.identityHashCode(obj));
  System.err.println("R " + threadId + " " + objId + " int[" + index + "]");
  int[] arr = (int[]) obj;
  return arr[index];
}

// Handle write int array
public static void write_int(java.lang.Object obj, int index, int value) {
  long threadId = Thread.currentThread().getId();
  String objId = Integer.toHexString(System.identityHashCode(obj));
  System.err.println("W " + threadId + " " + objId + " int[" + index + "]");
  int[] arr = (int[]) obj;
  arr[index] = value;
}
```

For all the loaded classes and their methods, use `CodeConverter::replaceArrayAccess` api to replace the array access with our generated methods.

Field Access

Field access statements are a bit different from array access. Because we cannot know all possible fields and generate corresponding tracing methods for them. Solution is using `CtMethod::instrument(ExprEditor)` api to store all the field access in the comming method. Then generate field read / write methods. They look like:

```
// Read field
public static ${type name} read_${field name}(java.lang.Object target) {
  ${declaring class name} recv = (${declaring class name}) target;
  long threadId = Thread.currentThread().getId();
  String objId = Integer.toHexString(System.identityHashCode(target));
  System.err.println("R " + threadId + " " + objId + " ${declaring class name}.${field name}");
  return recv.${field name};
}

// Write field
public static void write_${filed name}(java.lang.Object target, ${type name} value) {
  ${declaring class name} recv = (${declaring class name}) target;
  long threadId = Thread.currentThread().getId();
  String objId = Integer.toHexString(System.identityHashCode(target));
  System.err.println("W " + threadId + " " + objId + " ${declaring class name}.${filed name}");
  recv.${filed name} = value;
}
```

Notice that the `${...}` will be replaced with concrete field information. In the real implementation, to hack private field, I use reflection instead.