Java Byte Code

17. prosince 2020

Naprosto zásadní částí platformy Java je její virtuální stroj, Java Virtual Machine (JVM), který provádí Java Byte Code (JBC), do nejž jsou programy v Javě a jiných programovacích jazycích překládány. Platforma Java nabízí celou řadu nástrojů, které umožňují s JBC pracovat, analyzovat jej nebo vytvářet dokonce vlastní kód, čehož se využívá i mimo vývojová prostředí.

1 Analýza JBC

Exkurzi do světa JBC začneme nástrojem `javap`, který nám umožňuje zobrazit obsah jednotlivých souborů s třídami, které byly přeloženy do JBC (*class files*).

Předpokládejme, že máme jednoduchou třídu:

```
package cz.upol.pja.lecture10;

public class Point {
        private int x;
        private int y;
        public Point(int x, int y) {
                super();
                this.x = x;
                this.y = y;
        }
        public static Point create(int x, int y) {
                return new Point(x, y);
        }
        public int getX() {
                return x;
        }
        public void setX(int x) {
                this.x = x;
        }
        public int getY() {
                return y;
        }
```

```
        public void setY(int y) {
                this.y = y;
        }
        @Override
        public String toString() {
                return "Point [x=" + x + ", y=" + y + "]";
        }
}
```

V základu nám program `javap` zobrází elementární informace o daném souboru:

```
> javap Point.class
Compiled from "Point.java"
public class cz.upol.pja.lecture10.Point {
  public cz.upol.pja.lecture10.Point(int, int);
  public static cz.upol.pja.lecture10.Point create(int, int);
  public int getX();
  public void setX(int);
  public int getY();
  public void setY(int);
  public java.lang.String toString();
}
```

S dalšími přepínači je možné získat další informace jako je byte code jednotlivých metod. K tomu slouží přepínač `-c`.

```
> javap -c Point.class
Compiled from "Point.java"
public class cz.upol.pja.lecture10.Point {
  public cz.upol.pja.lecture10.Point(int, int);
    Code:
       0: aload_0
       1: invokespecial #11                 // Method java/lang/Object."<init>":()V
       4: aload_0
       5: iload_1
       6: putfield      #14                 // Field x:I
       9: aload_0
      10: iload_2
      11: putfield      #16                 // Field y:I
      14: return
```

Vzhledem k omezenému prostoru je zde ukázán jen byte code pro konstruktor. Vyzkoušejte si použití programu v rámci samostudia, ať vidíte úplný výpis metod.

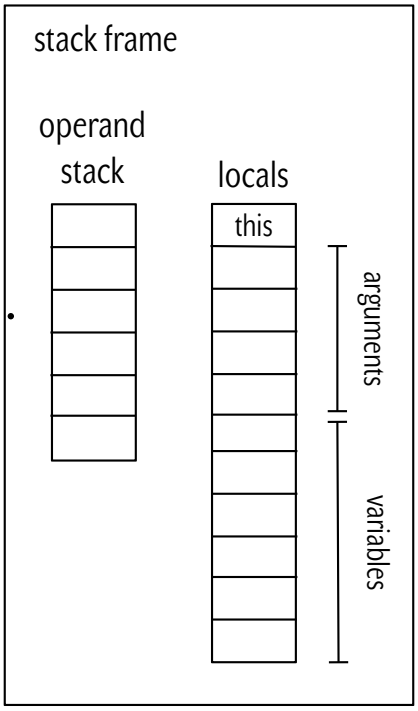
Z přeloženého souboru lze získat i další informace, např. jak odpovídají jednotlivé částí JBC kódu řádkům kódu, nebo jak vypadají jednotlivé zásobníkové rámce.

```
> javap -l Point.class
Compiled from "Point.java"
public class cz.upol.pja.lecture10.Point {
  public cz.upol.pja.lecture10.Point(int, int);
    LineNumberTable:
      line 18: 0
      line 19: 4
      line 20: 9
      line 21: 14
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      15     0  this   Lcz/upol/pja/lecture10/Point;
          0      15     1     x   I
          0      15     2     y   I
```

2 Jemný úvod do JBC

Na JVM můžeme nahlížet jako na zásobníkový procesor, kde vykonávaný kód se vztahuje vždy k nějaké metodě. Při každém vyvolání metody je alokován jeden zásobníkový rámec skládající se ze dvou částí (i) *operand stack*, kam jsou ukládány hodnoty, se kterými pracují jednotlivé instrukce JBC, a (ii) *locals*, což je oblast paměti, která má hned několik funkcí. V prvním slotu locals je uložen odkaz na aktuální objekt (`this`)[1], další sloty obsahují argumenty, které byly předané dané metodě, a zbylé sloty obsahují lokální proměnné alokované v rámci metody. V případě operand stacku i oblasti locals, každá hodnota zabírá právě jeden slot s výjimkou hodnot typu `double` a `long`, které zabírají sloty dva. Strukturu zásobníku ilustruje Obrázek 1.

[1] To neplatí, pokud se jedná o statickou metodu.



Obrázek 1: Zásobníkový rámec vykonávané metody

Jednotlivé instrukce JBC jsou docela jednoduché, obvykle mají žádný nebo jeden operand[2] a pracují s hodnotami uloženými na operand stacku.

Ukažme si použití na jednoduché metodě, která inkrementuje celočíselnou hodnotu o jedna.

```
public static int inc(int n) {
    return n + 1;
}
```

Tato metoda by se dala v JBC napsat jako posloupnost instrukcí.

```
iload_0    // načte na zásobník první argument
iconst_1   // načte na zásobník konstantu 1
iadd       // sečte dvě hodnoty na zásobníku
ireturn    // provede návrat z metody a vrací celočíselnou hodnotu ze zásobníku
```

U tohoto příkladu stojí za zmínku dvě věci. (i) Jednotlivé instrukce mají svůj prefix, který určuje s jakým typem pracují (např. „i" značí hodnotu typu integer, „d" double apod.). Instrukční sada obsahuje instrukce pro práci s nejběžnějšími konstantami nebo operandy.

Výčet jednotlivých instrukcí jde nad rámec tohoto textu a je přiložen k tomuto semináři.

3 Generování kódu s Javassist

Velice mocnou knihovnou pro práci s JBC představuje knihovna Javassist. K tomuto semináři jsou připojeny zdrojové kódy, které s její pomocí umožňují generovat JBC kód jednotlivých tříd, který je následně

[2] Existují samozřejmě výjimky.

možné nechat vykonávat. Můžeme tak dynamicky vytvářet vlastní třídy, což se hodí, pokud chceme vytvořit něco jako vlastní překladač nebo potřebujeme generovat třídy, které odpovídají nějakému uživatelem zadanému vstupu.

Kód obsahuje čtyři typy příkladů zaměřené na (i) jednoduché metody, (ii) řídící struktury, (iii) volání metod a (iv) tvorbu plnohodnotných tříd.

3.1 Jednoduché metody

V příkladu `SimpleMethods.java` jsou ukázány implementace metod provádějící základní aritmetiku. Metoda z předchozího příkladu, provádějící inkrementaci, je realizována následovně.

```
private static void createIncMethod(ClassBuilder classBuilder) {
     MethodBuilder incMethod = classBuilder.createMethod("inc",
                                    Modifier.PUBLIC | Modifier.STATIC,
                                    CtClass.intType, CtClass.intType

     Bytecode bc = incMethod.createByteCode(2, 1);
     bc.addIconst(1); // vlozi konstantu na zasobnik (pouzije nejvhodnejsi operaci)
     bc.addIload(0);  // vlozi prvni operand (pouzije nejvhodnejsi operaci)
     bc.addOpcode(Opcode.IADD);
     bc.addOpcode(Opcode.IRETURN);
}
```

Jednotlivé instrukce jsou generovány voláním metod objektu typu `Bytecode`. Při vytváření tohoto objektu je nutné uvést velikost operand stacku (v našem případě se na zásobníku operandů budou nacházet nejvýše dvě hodnoty) a locals, kdy potřebujeme právě jeden slot pro uložení argumentu metody.

Ostatní podobné metody lze implementovat v podobném duchu.

3.2 Řídící struktury

Při implementaci řídících struktur (příklad `ControlStructure.java`) jako jsou podmínky či cykly narážíme na jeden problém, a to je nemožnost použít „návěští" jako například v inline assembleru. Pokud chceme provést skok, musíme to udělat mírně komplikovaněji.

Nejdříve do kódu vložíme instrukci skoku, aniž bychom uvedli cílovou adresu, a zapamatujeme si, kde se tato instrukce nachází. Když se v generovaném kódu dostaneme na místo, kam má být skok proveden, zjistíme jaká je aktuální pozice v kódu a instrukci skoku doplníme o danou adresu, jak ukazuje následující příklad.

```
int patchAddr = bc.currentPc(); // zapamatuje si misto s instrukci podmineneho skoku
bc.add(Opcode.IFLE);
bc.addGap(2); // vycleni misto, kam je pozdeji umistena skutecna adresa

// dalsi kod
```

```
// PATCH -- nastavi adresu u podmineneho skoku na konkretni hodnotu
int offset = bc.currentPc() - patchAddr;
bc.write(patchAddr + 1, offset >> 8);
bc.write(patchAddr + 2, offset);
```

Všimněte si, že JBC k nepoužívá absolutní adresy ale relativní, které se vztahují k adrese instrukce daného skoku.

3.3 Volání metod

Práce s objekty včetně volání metod (příklad `MethodCalls.java`) vyžaduje ještě o něco víc péče, jelikož potřebujeme jednoznačně identifikovat jednotlivé metody. Nejdříve musíme získat odpovídající třídu (k tomu slouží objekt typu `ClassPool`) a u každé metody musíme uvést jednoznačnou typovou signaturu. Dále je nutné volit odpovídající operaci pro volání metody, tj. rozlišovat instrukce `invokevirtual` (pro normální metody), `invokestatic` (pro statické metody) nebo `invokeinterface` (pro metody definované v rozhraní).

3.4 Tvorba plnohodnotných tříd

Poslední z příkladů (`Objects.java`) ukazuje vytvoření plnohodnotné třídy reprezentující bod v rovině. Tato třída má jak atributy, tak konstruktor, tak jednotlivé metody.

V tomto příkladu za povšimnutí stojí zejména konstruktor, kde na začátku konstruktoru musíme zavolat konstruktor předka s využitím `invokespecial`. Dále za povšimnutí stojí práce s atributy.

Nejdříve na zásobník uložíme hodnotu `this` a hodnotu, kterou chceme atributu přiřadit. A následně operací `putfield` nastavíme hodnotu. Podobně jako v případě metod je nezbytné identifikovat třídu, do které atribut patří a jeho typ, jak ukazuje následující kód.

```
bc.addAload(0);
bc.addIload(1);
bc.addPutfield(classBuilder.getGeneratedClass(), "x", typeSignature(CtClass.intType));
```