Optimizarea fisierelor Java

Abstract

In acest paper voi descrie incercarea mea de a crea un optimizator de spatiu pentru fisierele .class ale limbajului Java.

Acest optimizator se bazeaza pe analiza statica a fisierelor pentru eliminarea metodelor nefolosite din cadrul fisierelor.

Voi analiza structura fisierelor, voi explica modul de analiza alor si voi expune modul de eliminare a metodelor.

Introducere

Java

Java este un limbaj de programare orientat pe obiecte. Acesta a fost dezvoltat de catre Sun Microsystems (acum Oracle), iar prima versiune a aparut in anul 1995.

Jaza s-a bazat pe sintaxa limbajului C, si a introdus notiunea de “scrie o data, ruleaza peste tot” (eng. “write once, run everywhere”). Spre deosebire de C si de C++, care trebuiesc compilate pentru fiecare platforma tinta, Java a avut avantajul ca trebuie compilat o singura data, si va merge garantat pe toate platformele suportate de limbaj.

Java Bytecode

Solutia limbajului Java pentru a fi independent de platforma este de transforma codul intr-o reprezentare intermediara, in loc de direct in cod binary pentru o anumita arhitectura .

Compilatorul Java (`javac`), transforma codul Java intr-un limbaj intermediar, numit Java Bytecode.

Acest limbaj este un limbaj low-level, destinat in mod exclusiv procesarii de catre masini, spre deosebire de codul Java, care este destinat oamenilor.

Dupa ce compilatorul a procesat codul Java, provenit din fisere .java in format text, acesta salveaza rezultatul in fisiere de tip clasa (.class) in format binar.

Masina Virtuala Java (JVM)

Odata generate fisierele binare, acestea sunt executate pe o masina virtuala specifica limbajului Java - numita JVM sau `The JVM` (eng. Java Virtual Machine).

Aceasta masina virtuala are rolul de a citi fisierele de clasa binare si de a le interpreta.

Masina virtuala este implementata ca o "masina cu stiva" (eng. stack machine), unde toate instructiunile limbajului bytecode interactioneaza cu datele de pe o stiva controlata de aplicatie.

Masina virtuala insusi este implementata in C/C++, si este compilata in cod binar direct, dependent de arhitectura. Dezvoltatorii limbajului Java sunt responsabili pentru corectitudinea si siguranta masinii virtuale, in timp ce dezvoltatorii de aplicatii Java au garantia ca daca codul lor Java este corect, atunci acesta va rula la fel, deterministic, pe orice platforma.

In acest regard, limbajul Java poate fi vazut ca un limbaj interpretat. Comparand cu alte limbaje populare interpretate, ca de exemplu Python, Ruby, sau Perl, ne-am astepta ca si Java sa fie la fel de incet ca acestea [1]. Totusi, Java obtine performante mult mai bune decat aceastea. Acest fapt se datoreaza compilarii tocmai-la-timp (eng. just-in-time), in care atunci cand interpretorul observa o secventa de cod care este interpretata repetitiv de foarte multe ori, va genera direct cod binary pentru aceasta.

Fisierele de clasa

Fisierele de clasa Java sunt formate din 10 sectiuni[2]:

1. Constanta magica.
2. Versiunea fisierului.
3. Constantele clasei.
4. Permisiunile de acces.
5. Numele clasei din fisier.
6. Numele superclasei.
7. Interfetele pe care clasa le implementeaza.
8. Campurile clasei.
9. Metodele clasei.
10. Atribute ale clasei.

In continuare voi da o scurta descriere a formatului sectiunilor.

Sectiunile fiserelor clasa

Magic

Toate fiserele clasa trebuiesc sa inceapa cu un numar denumit constanta magica. Acesta este folosit pentru a identifica in mod unic ca acestea sunt intra-devar fisiere clasa. Numarul magic are o valoare memorabila: reprezentarea hexadecimala este `0xCAFEBABE`,

Versiunea

Versiunea unui fisier clasa este data de doua valori, versiunea majora `M` si versiunea minora `m`. Versiunea clasei este atunci reprezentata ca `M.m`. (e.g., `45.1`). Aceasta este folosita pentru a mentine compatibilitatea in cazul modificarilor masinii virtuale care interpreteaza clasa sau ale compilatorului care o genereaza.

Constantele clasei

Tabela de constante este locul unde sunt stocate valorile literale constante ale clasei: * Numere intregi. * Numere cu virgula mobula. * Siruri de caractere, care pot reprezenta la randul lor: * Nume de clase. * Nume de metode. * Tipuri ale metodelor. * Informatii compuse din datele anterioare: * Referinta la o metoda a unei clase. * Referinta la o constanta a unei clase.

Toate celelalte tipuri de date compuse, cum ar fi metodele sau campurile, vor contine indecsi in tabela de constante.

Permisiunile de acces

Aceste permisiuni constau intr-o masca de bitsi, care reprezeinta operatiile permise pe aceasta clasa:

```
* daca clasa este publica, si poate fi accesta din afara pachetului acesteia.
* daca clasa este finala, si daca poate fi extinsa.
* daca invocarea metodelor din superclasa sa fie tratata special.
* daca este de fapt o interfata, si nu o clasa.
* daca este o clasa abstracta si nu poate fi instatiata.
```

Clasa curenta

Reprezinta un indice in tabela de constante, unde sunt stocate informatii despre clasa curenta.

Clasa super

Reprezinta un indice in tabela de consatante, cu informatii despre clasa din care a mostenit clasa curenta. Daca este 0, inseamna ca clasa curenta nu mosteneste nimic: singura clasa fara o superclasa este clasa `Object`.

E.g. pentru

```
class MyClass extends SuperClass implements Interface1, Interface {
    ....
}
```

Indicele corespunde lui `SuperClass`.

Interfetele

Reprezinta o colectie de indici in tabela de constante. Fiecare valoare de la acei indici reprezinta o interfata implementata in mod direct de catre clasa curenta. Interfetele apar in ordinea declarata in fisierele java.

E.g. pentru

```
class MyClass extends SuperClass implements Interface1, Interface2 {
    ...
}
```

Primul indice ar corespunde lui `Interface`, iar al doilea lui `Interface2`.

Campurile

Reprezinta informatii despre campurile (eng. fields) clasei: * Permisiunile de acces: daca este public sau privat, etc. * Numele campului. * Tipul campului. * Alte atribute: daca este deprecat, daca are o valoare constanta, etc.

Metodele

Reprezinta informtii despre toate metodele clasei, si include si constructorii:

```
* Permisiuni de acces: daca este public sau privat, daca este finala, daca este abstracta.
* Numele metodei.
* Tipul metodei.
* In caz ca nu este abstracta, byte codul metodei.
* Alte atribute:
    * Ce exceptii poate arunca.
    * Daca este deprecata.
```

Codul metodei este partea cea mai importanta, iar formatul acestuia urmeaza sa fie detaliat ulterior.

Atributele

Reprezinta alte informatii despre clasa, cum ar fi: * Clasele definite in interiorul acesteia. * In caz ca este o clasa anonima sau definita local, metoda in care este definita. * Numele fisierul sursa din care a fost compilata clasa.

In continuare, voi descrie din punct de vedere tehnic tipurile de date intalnite in fisierele de clasa:

Tipurile de baza

In formatul fisierelor clasa exista trei tipuri de baza, toate bazate pe intregi. In caz ca un intreg are mai multi octeti, acestia au ordinea de `big-endian`: cel mai semnificativ octet va fi mereu primul in memorie.

Nume	Semantica	Echivalentul in C
`u1`	intreg pe un octet, fara semn	`unsigned char` sau `uint8_t`
`u2`	intreg pe doi octeti, fara semn	`unsigned short` sau `uint16_t`
`u4`	intreg pe un octet, fara semn	`unsigned int` sau `uint32_t`

In codul sursa al proiectului, acestea sunt tratate astfel:

```
using u1 = uint8_t;
using u2 = uint16_t;
using u4 = uint32_t;
```

Tipuri de date compuse

cp_info

Fiecare constanta din tabela de constante incepe cu o eticheta de 1 octet, care reprezinta datele si tipul structurii. Continutul acesteia variaza in functie de eticheta, insa indiferent de eticheta, continutul trebuie sa aiba cel putin 2 octeti.

Aproape toate tipurile de constante ocupa un singur slot in tabela. Insa, din motive istorice, unele constante ocupa doua sloturi.

Totodata, tot din motive istorice, tabela este indexata de la 1, si nu de la 0, cum sunt celelalte.

Tipurile de constante

`CONSTANT_Class`

Corespunde valorii etichetei de 7 si contine un indice spre un alt camp in tabela de constante, de tipul `CONSTANT_Utf8` - un sir de caractere. Acel sir de caractere va contine numele clasei.

`CONSTANT_Fieldref`

Corespunde valorii etichetei de 9 si contine o referinta spre campul unei clase. Referinta conta in doi indici, amandoi care arata spre tabela de contante. Primul indice arata spre o constanta `CONSTANT_Class`, care reprezinta clasa sau interfata careia apartine metoda. Al doilea indice arata spre o constanta `CONSTANT_NameAndType`, care contine informatii despre numele si tipul campului.

`CONSTANT_Methodref`

Corespunde valorii etichetei de 10 si contine o referinta spre metoda unei clase. Are o structura identica cu `CONSTANT_Fieldref`, doar ca primul indice arata neaparat spre o clasa, in timp ce al doilea indice arata spre numele si tipul metodei.

`CONSTANT_InterfaceMethodref`

Corespunde valorii etichetei de 11 si contine o refereinta spre metoda unei interfete. Are o structura identica cu `CONSTANT_Methodref`, doar ca primul indice arata spre o interfata.

`CONSTANT_String`

Corespunde valorii etichetei de 8 si reprezinta un sir de caractere. Contine un indice, catre o structura de tipul `CONSTANT_Utf8`.

`CONSTANT_Integer`

Corespunde valorii etichetei de 3 si contine un intreg pe 4 octeti.

`CONSTANT_Float`

Corespunde valorii etichetei de 4 si contine un numar cu virgula mobila pe 4 octeti.

`CONSTANT_Long`

Corespunde valorii etichetei de 5 si contine un intreg pe 8 octeti. Din motive istorice, ocupa 2 spatii in tabela de constante.

`CONSTANT_Double`

Corespunde valorii etichetei de 6 si contine un numar cu virgula mobila pe 8 octeti. Din motive istorice, ocupa 2 spatii in tabela de constante.

`CONSTANT_NameAndType`

Corespunde valorii etichetei de 12. Descrie numele si tipul unui camp sau al unei metode, fara informatii despre clasa. Contine doi indici, amandoi catre structuri de tipul `CONSTANT_Utf8`. Primul reprezinta numele, iar al doilea tipul.

`CONSTANT_Utf8`

Corespunde valorii etichetei de 1. Reprezinta un sir de caractere encodat in formatul UTF-8. Contine un intreg `length`, de tipul `u2`, si apoi `length` octeti care descriu sirul in sine. Din cauza ca este encodat ca UTF-8, un singur caracter poate fi format din mai multi octeti.

`CONSTANT_MethodHandle`

Corespunde valorii etichetei de 15 si contine o referinte catre un camp, o metoda de clasa, sau o metoda de interfata.

`CONSTANT_MethodType`

Corespunde valorii etichetei de 16 si contine un indice catre o constanta `CONSTANT_UTf8`, ce reprezinta tipul unei metode.

`CONSTANT_InvokeDynamic`

Corespunde valorii etichetei de 18 si este folosit de catre `JVM` pentru a invoka o metoda polimorfica.

In cod `C++`, am reprezentat `cp_info` astfel:

```
struct cp_info {
    enum class Tag : u1 {
        CONSTANT_Class = 7,
        CONSTANT_Fieldref = 9,
        CONSTANT_Methodref = 10,
        CONSTANT_InterfaceMethodref = 11,
        CONSTANT_String = 8,
        CONSTANT_Integer = 3,
        CONSTANT_Float = 4,
        CONSTANT_Long = 5,
        CONSTANT_Double = 6,
        CONSTANT_NameAndType = 12,
        CONSTANT_Utf8_info = 1,
        CONSTANT_MethodHandle = 15,
        CONSTANT_MethodType = 16,
        CONSTANT_InvokeDynamic = 18,
    };

    Tag tag;
    std::vector<u1> data;
};
```

Iar structurile folosite pentru obiectivul propus au fost reprezentate astfel:

```
struct CONSTANT_Methodref_info {
    cp_info::Tag tag;
    u2 class_index;
```

```
    u2 name_and_type_index;
};
struct CONSTANT_Class_info {
    cp_info::Tag tag;
    u2 name_index;
};
struct CONSTANT_NameAndType_info {
    cp_info::Tag tag;
    u2 name_index;
    u2 descriptor_index;
};
```

field_info

Fiecare camp din cadrul unei clase este reprezentat printr-o structura de tipul `field_info`.

In cod `C++`, aceasta structura a fost reprezentata astfel:

```
struct field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    std::vector<attribute_info> attributes;
};
```

Unde: * `name_index` este o intrare in tabela de constante unde se afla o constanta de tipul `CONSTANT_Utf8`. * `descriptor_index` arata spre o constanta de tipul `CONSTANT_Utf8` si reprezinta tipul campului.

method_info

Fiecare metoda a unei clase/interfete este descrisa prin aceasta structura.

In cod `C++`, am implementat-o asa:

```
struct method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    std::vector<attribute_info> attributes;
};
```

Unde `name_index` si `descriptor_index` au aceeasi interpretare ca si la `field_info`.

Daca metoda nu este abstracta, atunci in vectorul `attributes` se va gasi un attribut de tipul `Code`, care contine bytecode-ul corespunzator acestei metode.

attribute_info

In `C++`, a fost implementata astfel:

```
struct attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    std::vector<u1> info;
};
```

Numele atributului determina modul in care octetii din vectorul `info` sunt interpretati. Pentru intentiile noastre, atributul de interes este cel de cod:

Code_attribute

```
struct Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;

    u2 max_stack;
    u2 max_locals;

    u4 code_length;
    std::vector<u1> code;

    u2 exception_table_length;
    struct exception {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    };
    std::vector<exception> exception_table; // of length exception_table_length.
    u2 attributes_count;
    std::vector<attribute_info> attributes; // of length attributes_count.
};
```

Aceasta structura este piesa centrala a lucrarii. In continuare, o voi descrie detaliat:

- `max_stack`: Reprezinta adancimea maxima a stivei masinii virtuale cand aceasta bucata de cod este interpretata.
- `max_locals`: Reprezinta numarul maxim de variabile locale alocate in acelasi timp cand aceasta bucata de cod este interpretata.

- `code`: Codul metodei.
- `exception_table`: Exceptiile pe care le poate arunca metoda.

`Code`

Vectorul `code` din cadrul atributului `Code` reprezinta bytecode-ul propriu-zis al metodei.

Acest vector contine instructiunile care sunt executate de catre masina virtuala.

JVM-ul ruleaza ca o masina cu stiva, iar toate instructiunile opereaza pe aceasta stiva. Reultatul rularii unei instructiuni este modificarea stivei: scoaterea si adaugarea de elemente in varful acesteia.

Instructiunile au in general formatul [3]:

```
nume_instr
operand1
operand2
...
```

cu un numar variabil de operanzi, prezenti in mod explicit in vectorul de `cod`.

Fiecarui instructiuni ii corespunde un octet, denumit opcode. Fiecare operand este fie cunoscut la compilare, fie calculat in mod dinamic la rulare.

Cele mai multe operatii nu au niciun operand dat in mod explicit la nivelul instructiunii: ele lucreaza doar cu valorile din varful stivei la momentul executarii codului.

De exemplu:

Instructiunea `imul` are octetul `104` sau `0x68`. Acestea da pop la doua valori din varful stivei: `value1` si `value2`. Amandoua valorile trebuie sa fie de tipul `int`. Rezultatul este inmultirea celor doua valori: `result = value1 * value2`, si este pus in varful stivei.

Dintre cele peste o suta de instructiuni, noi suntem preocupati doar de 5 dintre acestea: cele care au de a face cu invocarea unei metode.

invokedynamic

Format:

```
invokedynamic
index1
index2
0
0
```

Opcode-ul corespunzator acestei instructiuni este `186` sau `0xba`.

Index1 si index2 sunt doi octeti sunt compusi in

```
index = (index1 << 8) | index2
```

Indicele compus reprezinta o intrare in tabela de constante. La locatia respectiva trebuie sa se afle o structura de tipul `CONSTANT_MethodHandle`

invokeinterface

Format:

```
invokeinterface
index1
index2
count
0
```

Opcode-ul corespunzator este `185` sau `0xb9`. `index1` si `index2` sunt folositi, in mod similar ca la `invokedynamic`, pentru a construi un `indice` in tabela de constante.

La pozitia respectiva in tabela, trebuie sa se regaseasca o strutura de tipul `CONSTANT_Methodref`.

`count` trebuie sa fie un octet fara semn diferit de 0. Acest operand descrie numarul argumentelor metodei, si este necesar din motive istorice: aceasta informatie poate fi dedusa din tipul metodei.

TODO(ericpts): add resolution order.

invokespecial

Format:

```
invokespecial
index1
index2
```

Opcode-u corespunzator este `183` sau `0xb7`. La fel ca la `invokeinterface`, este format un indice in tabela de constante, catre o structura `CONSTANT_Methodref`.

Aceasta instructiune este folosita pentru a invoca constructorii claselor.

invokestatic

Format:

```
invokestatic
index1
index1
```

Opcode-ul corespunzator este `184` sau `0xb8`. Instructiunea este invocata pentru a invoke o metoda statica a unei clase.

La fel ca la `invokeinterface`, este construit un indice compus, si folosit pentru a indexa tabela de constante.

invokevirtual

Format:

```
invokevirtual
index1
index1
```

Opcode-ul corespunzator este `182` sau `0xb6`, iar interpretarea este la fel ca la `invokeinterface`.

Aceasta este cea mai comuna instructiune de invocare de functii.

Dupa ce numele si tipul metodei, cat si clasa `C` de care apartine aceasta sunt rezolvate, masina virtuala cauta metoda respectiva in clasa referentiata. In caz ca o gaseste, cautarea se termina. In caz negativ, JVM va continua cautarea recursiv din superclasa lui `C`.

In `C++`, am reprezentat aceste instructiuni de interes astfel:

```
enum class Instr {
    invokedynamic = 0xba,
    invokeinterface = 0xb9,
    invokespecial = 0xb7,
    invokestatic = 0xb8,
    invokevirtual = 0xb6,
};
```

ClassFile

Folosind definitiile anterioare, putem descrie un fisier de clasa binar in C++:

```
struct ClassFile {
    u4 magic; // Should be 0xCAFEBABE.

    u2 minor_version;
    u2 major_version;

    u2 constant_pool_count;
    std::vector<cp_info> constant_pool;

    u2 access_flags;

    u2 this_class;
    u2 super_class;

    u2 interface_count;
    std::vector<interface_info> interfaces;

    u2 field_count;
```

```
    std::vector<field_info> fields;

    u2 method_count;
    std::vector<method_info> methods;

    u2 attribute_count;
    std::vector<attribute_info> attributes;
};
```

Studiu de caz

In continuare, voi exemplica structura unui fisier clasa cu un exemplu.

Codul Java este urmatorul:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("project1 - hello world");
        foo();
    }

    public static void foo() {
        System.out.println("project1 - foo()");
    }
}
```

Compilatorul folosit este `openjdk-11`. Clasa a fost utilizata folosind utilitarul `javap` [4], care este de asemenea inclus in pachetul `openjdk-11`.

In primul rand, tabela de constante:

```
Constant pool:
   #1 = Methodref          #8.#18         // java/lang/Object."<init>":()V
   #2 = Fieldref           #19.#20        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #21            // project1 - hello world
   #4 = Methodref          #22.#23        // java/io/PrintStream.println:(Ljava/lang/String;
   #5 = Methodref          #7.#24         // Main.foo:()V
   #6 = String             #25            // project1 - foo()
   #7 = Class              #26            // Main
   #8 = Class              #27            // java/lang/Object
   #9 = Utf8               <init>
  #10 = Utf8               ()V
  #11 = Utf8               Code
  #12 = Utf8               LineNumberTable
  #13 = Utf8               main
  #14 = Utf8               ([Ljava/lang/String;)V
```

```
  #15 = Utf8               foo
  #16 = Utf8               SourceFile
  #17 = Utf8               Main.java
  #18 = NameAndType        #9:#10         // "<init>":()V
  #19 = Class              #28            // java/lang/System
  #20 = NameAndType        #29:#30        // out:Ljava/io/PrintStream;
  #21 = Utf8               project1 - hello world
  #22 = Class              #31            // java/io/PrintStream
  #23 = NameAndType        #32:#33        // println:(Ljava/lang/String;)V
  #24 = NameAndType        #15:#10        // foo:()V
  #25 = Utf8               project1 - foo()
  #26 = Utf8               Main
  #27 = Utf8               java/lang/Object
  #28 = Utf8               java/lang/System
  #29 = Utf8               out
  #30 = Utf8               Ljava/io/PrintStream;
  #31 = Utf8               java/io/PrintStream
  #32 = Utf8               println
  #33 = Utf8               (Ljava/lang/String;)V
```

In acest format, namespace-urile imbricate sunt reprezentate prin `/`.

Informatii despre clasa:

```
Classfile Main.class
  Last modified May 28, 2018; size 520 bytes
  MD5 checksum 248b729dfe4b4bc8da895944d30fdc28
  Compiled from "Main.java"
public class Main
  minor version: 0
  major version: 55
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #7                          // Main
  super_class: #8                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 3, attributes: 1
```

Constructorul clasei:

```
{
  public Main();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
```

```
        line 1: 0
```

Metoda `main(String[] args)`:

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: (0x0009) ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=1, args_size=1
     0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStrea
     3: ldc           #3                  // String project1 - hello world
     5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/
     8: invokestatic  #5                  // Method foo:()V
    11: return
  LineNumberTable:
    line 3: 0
    line 4: 8
    line 5: 11
```

Metoda `foo()`:

```
public static void foo();
descriptor: ()V
flags: (0x0009) ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=0, args_size=0
     0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStrea
     3: ldc           #6                  // String project1 - foo()
     5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/
     8: return
  LineNumberTable:
    line 8: 0
    line 9: 8
```

Implementare

Deserializare

Prima problema intalnita in construirea optimizatorului este serializarea si deserializarea fisierelor clasa. Problema aceasta a fost rezolvata folosind clasa `ClassReader`:

```
/// This class handles the parsation (deserialization and serialization) of
/// Java's .class files.
/// Normal usage should be:
/// 1. Reading the binary data (for example, from a file on disk)
```

```
/// 2. Instantiating this ClassReader.
/// 3. Parsing the actual file.
struct ClassReader {
  private:
    /// The binary representation of the class being parser.
    BytesParser m_bparser;

    /// The class file that is being populated as the parsing progresses.
    ClassFile m_cf;

  public:
    /// Initialize the reader, with the binary `data` of the class file.
    ClassReader(std::vector<uint8_t> data);

    /// Parse an entire class file.
    /// This is the method that you most likely want to use.
    ClassFile deserialize();

  private:
    /// Parses a constant from the data buffer, and returns the data
    /// and how many slots it takes up in the constant table.
    cp_info parse_cp_info();

    /// Parses a field_info struct from the data buffer.
    field_info parse_field_info();

    /// Parses a method_info struct from the data buffer.
    method_info parse_method_info();

    /// Asserts that `idx` is an index into the constant pool, tagged with
    /// `tag`.
    void expect_cpool_entry(int idx, cp_info::Tag tag) const;
};
```


[1] https://github.com/trizen/language-benchmarks

[2] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html

[3] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

[4] https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html