Software Básico

Apresentação do Leitor/Exibidor

Andrey Emmanuel Matrosov Mazépas – 16/0112362
Lincoln Abreu Barbosa – 14/0045023

Estrutura Geral do Leitor/Exibidor

Main

```cpp
#include "class_loader/class_loader.hpp"
#include "util/JvmException.hpp"
#include "util/commander.hpp"

int main (int argc, char *argv[ ]) {

    std::vector<std::string> commands(argv + 1, argv + argc);

    try {
        jvm::CommandState state = jvm::Commander::parse(commands);

        jvm::ClassLoader cl;
        cl.read(state.filename);

        if (state.shouldDescribe) {
            cl.show();
        }

    } catch (const jvm::JvmException& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

ClassLoader

```cpp
/**
 * Reads all the class file
 * @param file The file to extract the data
 */
void ClassLoader::read (std::basic_string<char> filename) {
    auto file = Reader();

    file.open(filename);

    read_version(file);
    read_cp(file);
    read_flags(file);
    read_interfaces(file);
    read_fields(file);
    read_methods(file);
    read_attributes(file);

    file.close();
}
```

ClassLoader

```cpp
void ClassLoader::show () {
    std::cout << "> .class" << std::endl << std::endl;

    print_version();
    print_cp();
    print_class_flags();
    print_this_class();
    print_super_class();
    print_interfaces();
    print_fields();
    print_methods();
    print_attributes();
}
```

Estrutura Geral do Leitor/Exibidor

- Projeto utiliza uma classe Reader() que possui métodos facilitadores para percorrer o arquivo, como *getNextByte()* ou *getNextHalfWord()*
- Base.hpp contém definições com os nomes que serão utilizados pela JVM.

```
#define MAGIC_NUMBER 0xCAFEBABE
#define BYTESIZE     8
#define HALFSIZE     16
#define WORDSIZE     32

#define T_ARRAY      1
#define T_OBJ        2
#define T_REF        3
#define T_BOOL       4
#define T_CHAR       5
#define T_FLOAT      6
#define T_DOUBLE     7
#define T_BYTE       8
#define T_SHORT      9
#define T_INT        10
#define T_LONG       11
#define T_RET        12
#define T_METHOD     13
#define T_INTERFACE  14
#define T_STRING     69
```

```
typedef uint8_t  u1;
typedef uint16_t u2;
typedef uint32_t u4;
typedef uint64_t u8;
typedef int8_t   i1;
typedef int16_t  i2;
typedef int32_t  i4;
typedef int64_t  i8;
```

Magic Number e Min/Max Version

- Toda classe Java precisa começar com um Magic Number 0xCAFEBABE.
- Min e Max numbers informam qual a versão do compilador utilizada para gerar aquele arquivo.

```
void ClassLoader::read_version (Reader& file) {
    magic_number = file.getNextWord();
    min_version = file.getNextHalfWord();
    max_version = file.getNextHalfWord();
}
```

Leitura da Constant Pool

- A próxima seção é dada pelo contador do pool de constantes seguida da tabela em si.
- São constantes todas as entradas imutáveis durante o runtime, como números, nomes de variáveis, referências, etc.

```
void ClassLoader::read_cp (Reader& file) {
    cp_count = (uint16_t)(file.getNextHalfWord() - 1);

    if (cp_count <= 0) {
        return;
    }

    constant_pool.fill(file, cp_count);
}
```

Leitura da Constant Pool

- Cada entrada da Tabela é precedida pelo identificador de tipo que informa o leitor qual tipo esperar e quantos bytes ler.

```
enum CP_TAGS : uint8_t {
    Class              = 7,
    FieldRef           = 9,
    MethodRef          = 10,
    InterfaceMethodRef = 11,
    String             = 8,
    Integer            = 3,
    Float              = 4,
    Long               = 5,
    Double             = 6,
    NameAndType        = 12,
    Utf8               = 1,
    MethodHandle       = 15,
    MethodType         = 16,
    InvokeDynamic      = 18
};
```

```
void ConstantPool::fill(Reader &reader, size_type cp_count) {
    reserve(cp_count);

    for (auto i = cp_count; i > 0; --i) {
        auto tag = reader.getNextByte();
        push_back(getNextEntry(reader, tag));
        if (tag == CP_TAGS::Double || tag == CP_TAGS::Long) {
            // these take two CP entries, the next entry is valid but unusable
            push_back(nullptr);
            --i;
        }
    }

    shrink_to_fit();
}
```

Leitura da Constant Pool

- Cada tipo necessita de um método específico para ser lido e impresso.

```
CP_Integer::CP_Integer(Reader &reader) {
    _bytes = reader.getNextWord();
}

void CP_Integer::printToStream(std::ostream &os, ConstantPool &cp) {
    os << "Integer" << std::endl;
    os << "\t\t" << toString(cp) << std::endl;
}

std::string CP_Integer::toString(ConstantPool &cp) {
    return std::to_string(reinterpret_cast<int32_t&>(_bytes));
}
```

Flags de Acesso

- As flags de acesso são dois bytes após a constant pool para denotar permissões de acesso e as propriedades da classe ou interface do arquivo.
- Os bytes em seguida são índices da CP para a estrutura de informação da classe *this* e a sua *Super* Classe direta.

```
void ClassLoader::read_flags (Reader &file) {
    access_flags = file.getNextHalfWord();
    this_class = file.getNextHalfWord();
    super_class = file.getNextHalfWord();
}
```

Interfaces

- Começando com um contador, todas as interfaces definidas do arquivo estarão nessa seção. Contém um array de índices da constant pool para cada interface.

```
void ClassLoader::read_interfaces (Reader &file) {
    interfaces_count = file.getNextHalfWord();

    if (interfaces_count == 0) {
        return;
    }

    for (int i = 0; i < interfaces_count; ++i) {
        interfaces.emplace_back(file);
    }
}
```

Fields

- Da mesma forma que as interfaces, inicia-se com 2 bytes do contador seguido de um array de referências para a constant pool.
- Fields são instâncias ou propriedades definidas na classe ou interface, não incluindo aquelas herdadas pela *Super* classe.
- Cada entrada no array representa um Field, apontando para uma estrutura que contém os dados e algumas informações do Field enquanto outros metadados como o nome são guardados na Constant Pool.

Fields

```
namespace fields {

    enum Flags : uint32_t {
        PUBLIC    = 0x0001,    // Declared public; may be accessed from outside its package.
        PRIVATE   = 0x0002,    // Declared private; usable only within the defining class.
        PROTECTED = 0x0004,    // Declared protected; may be accessed within subclasses.
        STATIC    = 0x0008,    // Declared static.
        FINAL     = 0x0010,    // Declared final; never directly assigned to after object construction (JLS §17.5).
        VOLATILE  = 0x0040,    // Declared volatile; cannot be cached.
        TRANSIENT = 0x0080,    // Declared transient; not written or read by a persistent object manager.
        SYNTHETIC = 0x1000,    // Declared synthetic; not present in the source code.
        ENUM      = 0x4000     // Declared as an element of an enum.
    };
}
```

```
void FieldInfo::Read(Reader &reader, ConstantPool &cp) {
    access_flags = reader.getNextHalfWord();
    name_index = reader.getNextHalfWord();
    descriptor_index = reader.getNextHalfWord();
    attributes.fill(reader, cp);
}
```

```
void ClassLoader::read_fields (Reader &file) {
    fields_count = file.getNextHalfWord();

    if (fields_count == 0) {
        return;
    }

    for (int i = 0; i < fields_count; ++i) {
        fields.emplace_back(file, constant_pool);
    }
}
```

Métodos

- A seção de métodos possui, também, um contador indicando o número de métodos definidos na classe, não contando aqueles herdados, referenciados em um array de ponteiros para estruturas na CP.
- A estrutura método na CP contém diversas informações como lista de argumentos, tipo do return, valor da stack para acomodar todas suas variáveis e operandos e sua sequência de bytecode.

Métodos

- Os índices do nome e descritor são obtidos pelos primeiros bytes da estrutura do método, após as flags de acesso, e então retornados através da Constant Pool.

```
void ClassLoader::read_methods (Reader &file) {
    methods_count = file.getNextHalfWord();

    if (methods_count == 0) {
        return;
    }

    for (int i = 0; i < methods_count; ++i) {
        MethodInfo currentMethod(file, constant_pool);
        auto name = currentMethod.getName(constant_pool);
        auto descriptor = currentMethod.getDescriptor(constant_pool);
        methods.insert({name + descriptor, currentMethod});
    }

}
```

```
void MethodInfo::Read(Reader &reader, ConstantPool &cp) {
    access_flags = reader.getNextHalfWord();
    name_index = reader.getNextHalfWord();
    descriptor_index = reader.getNextHalfWord();
    attributes.fill(reader, cp);
}

std::string MethodInfo::getName(ConstantPool &cp) {
    return cp[name_index]->toString(cp);
}

std::string MethodInfo::getDescriptor(ConstantPool &cp) {
    return cp[descriptor_index]->toString(cp);
}
```

Atributos

- A última seção é um array, precedido de um contador, de atributos gerais da classe. A documentação oficial define 5 atributos críticos para a interpretação correta do .class, atributos extras serão ignorados caso não compreendidos pela JVM.

  - ConstantValue
  - Code
  - StackMapTable
  - Exceptions
  - BootstrapMethods

```
void ClassLoader::read_attributes (Reader &file) {
    attributes.fill(file, constant_pool);
}
```

Atributos

- O método fill irá iterar sobre o arquivo o número de vezes definido pela quantidade de atributos conhecidos detectados.
- Os atributos são guardados em vetores específicos dependendo do nome.
- Atributos de tipo desnecessário/desconhecidos são ignorados pela JVM.

```
void AttributeInfo::fill(Reader &reader, ConstantPool &cp) {
    auto attr_count = reader.getNextHalfWord();
    reserve(attr_count);
    while(attr_count--) {
        auto name_index = reader.getNextHalfWord();
        auto attr_length = reader.getNextWord();
        auto name = cp[name_index]->toString(cp);

        // instantiate the attribute and initialize with data from reader
        if (name == "Code") {
            auto codePtr = std::make_shared<AttrCode>(reader, cp);
            Codes.push_back(codePtr);
            push_back(codePtr);
        } else if (name == "Exceptions") {
            auto exceptionsPtr = std::make_shared<AttrExceptions>(reader, cp);
            Exceptions.push_back(exceptionsPtr);
            push_back(exceptionsPtr);
        } else if (name == "ConstantValue") {
            auto constantValuePtr = std::make_shared<AttrConstantValue>(reader, cp);
            ConstValues.push_back(constantValuePtr);
            push_back(constantValuePtr);
        } else {
            // In this case, the attribute is of a type we won't read
            // Add a nullptr and skip the attribute's bytes
            emplace_back();
            reader.skipBytes(attr_length);
        }
    }
}
```

Mnemônicos de instruções

- O atributo Code de um método contém, entre outros dados, o código a ser executado em instruções mnemônicas para JVM.

```
struct AttrCode : public AttrEntry {
    typedef struct {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table_entry;

    u2 max_stack;
    u2 max_locals;
    std::vector<u1> code_bytes;
    CodeInfo code;
    std::vector<exception_table_entry> exception_table;
    AttributeInfo attributes;

    AttrCode(Reader &reader, ConstantPool &cp);
    void printToStream(std::ostream &ostream, ConstantPool &pool, std::string &prefix) override;
};
```

```
[02] main (16)
    Descriptor: ([Ljava/lang/String;)V (17)
    Flags:
        Public
        Static
    Attributes count:1
        Code:
            0: iconst_3
            1: istore_1
            2: iconst_4
            3: istore_2
            4: iload_2
            5: iload_1
            6: iadd
            7: istore_3
            8: iload_3
            9: iload_2
            10: iload_1
            11: invokestatic 2
            14: iadd
            15: istore_3
            16: getstatic 3
            19: ldc 4
            21: invokevirtual 5
            24: getstatic 3
            27: iload_3
            28: invokevirtual 6
            31: return
```