VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ZPĚTNÝ PŘEKLADAČ BAJTKÓDU JAZYKA JAVA

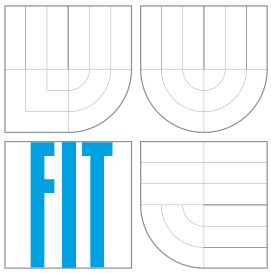
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAROMÍR HŘIBAL

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ZPĚTNÝ PŘEKLADAČ BAJTKÓDU JAZYKA JAVA

JAVA BYTECODE DISASSEMBLER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAROMÍR HŘIBAL
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYNĚK KŘIVKA, Ph.D.
SUPERVISOR

BRNO 2015

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2014/2015

Zadání bakalářské práce

Řešitel: **Hřibal Jaromír**
Obor: Informační technologie
Téma: **Zpětný překladač bajtkódu jazyka Java**
Java Bytecode Disassembler
Kategorie: Překladače

Pokyny:

1. Prostudujte a popište způsob překladu programů v jazyce Java ze zdrojových textů do bajtkódu a popište základní strukturu bajtkódu.
2. Prozkoumejte a proveďte rešerši existujících nástrojů na zpětný překlad do zdrojových textů v Javě včetně knihoven pro načítání Java bajtkódu.
3. Dle pokynů vedoucího navrhněte disassembler pro bajtkód (vč. ladicích informací) jazyka Java (verze 7), který bude pracovat nad zadaným projektem v jazyce Java.
4. Nástroj z předchozího bodu implementujte a dále implementujte možnost uživatelsky přívětivého zobrazení zdrojového kódu Javovské třídy současně s bajtkódem odpovídajícím jednotlivým programovým řádkům.
5. Zhodnoťte dosažené výsledky v kontextu existujících nástrojů a navrhněte možnosti dalšího vývoje projektu.

Literatura:

- Lindholm, T., et al.: *The Java Virtual Machine Specification, Java SE 7 Edition (Java Series)*. Addison-Wesley, 2013. Dostupné online: http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2014
Datum odevzdání: 20. května 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se zabývá popisem Java virtuálního stroje vycházejícího ze specifikace, zejména instrukční sadou a formátem spustitelných jednotek. Obecné části jsou konkretizovány na základě referenční implementace specifikace Java virtuálního stroje, kterou je Java HotSpot. Součástí práce je také implementace nástroje pro uživatelsky přívětivé zobrazení obsahu spustitelných jednotek spolu s ekvivalentním zdrojovým kódem, ze kterého byly generovány. Nástroj neplní roli dekompilátoru, ale předpokládá dostupnost zdrojových kódu a náležitých informací ve spustitelných jednotkách.

Abstract

This thesis is about Java Virtual Machine, especially about its instruction set and the format of units it can process. General parts are concretized with reference implementation of Java Virtual Machine specification, which is Java HotSpot. One part of this thesis is to create utility tool that allows the user to get user friendly view of the units Java Virtual Machine can handle and with corresponding source code from which these units were generated. This tool doesn't work as decompiler, but expects that the source code is available and needed debug informations are stored in appropriate units.

Klíčová slova

Java, HotSpot, Zpětný překlad, Java bajtkód, Java virtuální stroj

Keywords

Java, HotSpot, Disassembler, Java bytecode, Java Virtual Machine

Citace

Jaromír Hřibal: Zpětný překladač bajtkódu jazyka Java, bakalářská práce, Brno, FIT VUT v Brně, 2015

Zpětný překladač bajtkódu jazyka Java

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jaromír Hřibal
26. května 2015

Obsah

Kapitola 1

Úvod

Jedním z hlavních důvodů rozvoje IT sektoru, ke kterému v posledních dvou dekádách došlo, je vznik mnoha moderních programovacích jazyků, které umožňují snadněji a pohodlněji vytvářet počítačový software.

Mezi tyto jazyky se řadí také jazyk Java [48]. Vývoj tohoto objektově orientovaného, interpretovaného programovacího jazyka, jehož původní název byl Oak [50], započal v roce 1991 James Gosling [46]. Veřejnosti byl jazyk představen v roce 1996 společností Sun Microsystem [53], kdy byla uvolněna jeho první verze Java 1.0 [49]. Jazyk byl původně určen pro programování vestavěných systémů a důvodem vzniku byla tehdejší nespokojenost s možnostmi jazkya C++ [40] pro jejich programování (přenositelnost, paměťová náročnost, velký počet chyb plynoucích ze složitosti, nutnost vlastní správy paměti).

V současnosti je Java jeden z nejpoužívanějších programovacích jazyků, jehož popularita stále sílí a programátoři, kteří tento jazyk ovládají, patří mezi nejvyhledávanější. Její použití je od stolních počítačů po mobilních zařízení a vestavěné systémy obecně.

Hlavním důvodem úspěchu Javy je fakt, že programy v ni vytvořené jsou přenositelné a tedy mohou běžet na různých platformách, bez nutnosti znovu překládat zdrojové kódy (*"Write Once, Run Anywhere"*) [55]. Umožňuje tedy binární kompatibilitu, narozdíl např. od jazyka C [41], kde lze dosáhnout přenositelnosti pouze na úrovni zdrojových kódů. Mezi další důležité vlastnosti jazyka patří jednoduchost, syntaxe vycházející z dobře známých jazyků jako C a C++ a automatická správa paměti.

Protože programy napsané v jazyce Java jsou určeny pro různé platformy, není mezikód vzniklý překladem zdrojových kódu určen přímo pro procesor konkrétního zařízení, ale pro speciální program, který implementuje specifikaci Java Virtual Machine [35] *(dále jen JVM)* a dokáže tento mezikód vykonávat. Aby na konkrétním zařízení mohl být spuštěn program v jazyce Java, musí pro něho tedy existovat implementace JVM specifikace. První specifikace JVM vycházela z virtuálního stroje navrženého Jamesem Goslingem pro již zmíněný jazyk Oak, ze kterého se Java později vyvinula.

Java Virtual Machine umožňuje interpretovat mezikód, který se nazývá Java bajtkód (*z angl. Java bytecode*) [47] a je výstupem překladu zdrojových kódů v Javě. Výhodou využití virtuálního stroje pro interpretaci bajtkódu, který sám o sobě není pevně svázán s jazykem Java ani s konkrétní platformou, je, že umožňuje vytvořit překladače do bajtkódu i pro jiné jazyky, než je Java. Mezi takové jazyky se řadí např. Scala [52].

Vzhledem k tomu, kam až se Java od svého vzniku posunula, lze s nadsázkou přemýšlet o tom, že Java změnila svět.

1.1 Náplň práce

Tato práce se zabývá základním popisem JVM, vycházejícím ze specifikace, popisem formátu výstupních jednotek překladačů pro JVM a popisem jednotlivých typů instrukcí bajtkódu. Jedna kapitola je věnována podpoře JVM pro běh dynamicky typovaných jazyků.

V průběhu textu je u některých částí uveden příklad, jak konkrétní problematiku řeší referenční implementace JVM specifikace — 32 bitová verze HotSpot na platformě x86 [44]. Dvě kapitoly se navíc věnují popisu, jak HotSpot implementuje objekty a synchronizaci přístupu k objektům.

Součástí práce je také implementace nástroje pro inspekci souborů obsahujících bajtkód a zobrazení instrukcí bajtkódu v uživatelsky přívětivé podobě spolu částmi zdrojových kódů, ze kterých byl bajtkód generován. Dále jsou popsány některé existující nástroje pro manipulaci s bajtkódem.

1.2 Motivace

Přestože pro psaní programů bežících na JVM postačí zvládnutí syntaxe a sémantiky nějakého jazyka, pro který existuje překladač do bajtkódu, tak znalost bajtkódu a interního fungovování JVM značně přispěje k celkovému pochopení a přináší řadu výhod.

Výhody znalosti fungování JVM

- porozumění smyslu jednotlivých instrukcí a jak mohou být překládány různé jazykové konstrukce
- možnost odhalení chyby v překladači na základě vygenerovaného bajtkódu
- možnost přímé modifikace vygenerovaných Class file jednotek
- možnost tvorby softwarových komponent pro generování Class file jednotek
- možnost portace JVM na další platformy
- možnost vytvoření vlastní platformy s využitím JVM např. pro tvorbu realtime webových aplikací (vytvoření dynamicky typovaného jazyka zaměřeného na konkrétní doménu, zabudování webového serveru do JVM)

Tvorba softwarových komponent pro generování a manipulaci Class file jednotek je ve světě Javy poměrně častým jevem. Toto umožňuje implementovat např. aspektově orientované programování [2].

Kapitola 2

Java Virtual Machine

Java Virtual Machine (*dále jen JVM*) je obecný název pro software, který odpovídá specifikaci Java Virtual Machine [35]. Na JVM lze nahlížet jako na program, který dokáže korektně zpracovávat Class file jednotky, interpretovat instrukční sadu které se říká Java bajtkód (*z angl. Java bytecode*) a manipulovat s různými paměťovými oblastmi.

2.1 Paměťové oblasti

JVM používá několik paměťových oblastí, které jse popsány dále.

Java halda

Java halda (*z angl. Java heap*) je oblast společná pro všechna vlákna a slouží jako místo, odkud JVM alokuje potřebnou paměť pro vytváření polí a instancí tříd. Tato paměť je v automatické správě JVM a stará se o ni garbage collector.

HotSpot používá generační haldu (*z angl. generation heap*), kdy je halda rozdeléna na 2 částí a každá slouží pro ukládání objektů s různou délkou života. Tyto části se nazývají Young Generation a Old Generation [38].

Oblast Young Generation je dál rozdělena na tři časti - prostor pro nové objekty (*z angl. Eden*) a dva prostory (*z angl. Survivor Spaces*), kde jeden je vždy prázdný (*TO* prostor) a druhý obsahuje objekty, které přežily alespoň jeden cyklus garbage collectoru (*FROM* prostor).

Při cyklu garbage collectoru se vychází z množiny tzv. kořenových objektů (*z angl. Root Objects*), které pocházejí např. ze statických datových členů tříd a z objektů vyskystujících se na zásobnících vláken. Objekty, na které se lze z těchto kořenových objektů dostat skrze řetezec referencí jsou z části pro nové objekty a z *FROM* prostoru přesunuty do *TO* prosotoru.

Při dalším cyklu se proces opakuje, akorát si mezi sebou roli prohodí prostory *TO* a *FROM*, tedy *FROM* prostor se po skončení cyklu stává *TO* prostorem a opačně.

Při každém přesouvání objektů z *FROM* prostoru do *TO* prostoru mohou být některé objekty, které už přežily dostatečný počet cyklů, přesunuty do Old Generation.

Jednotlivá vlákna mohou mít z oblasti pro nové objekty vyhrazenu část prostoru, která je označována jako TLAB (*z angl. Thread Local Allocation Buffer*) [32][13] a ze které si dané vlákno alokuje prostor pro nové objekty. Díky tomu není nutné synchronizovat vytváření nových objektů z různých vláken a celý proces se tím zefektivní.

Method Area

Tato oblast slouží pro ukládání různých struktur jako např. Run-Time Constant Pool nebo bajtkód metod.

HotSpot objekty reprezentující metody, Run-Time Constant Pool a další interní struktury vytváří v oblasti nazývané Permanent Generation [38]. Tato oblast slouží k ukládání veškerých objektů, které povětšinou existují po celou dobu běhu programu. Např. java objekt reprezentující výjimku `java.lang.OutOfMemoryError` [3][8] je také alokován v této oblasti.

Programový čítač

Každé JVM vlákno vykonávající kód metody má svůj programový čítač (*z angl. program counter)*, který obsahuje adresu právě vykonávané instrukce. Pokud je metoda nativní, obsah programového čítače není definován.

HotSpot implementuje dva typy interpretů a to Cpp interpret (*z angl. Cpp Interpreter*) a Šablonový interpret (*z angl. Template Interpreter*). Cpp interpret si adresu právě vykonávané instrukce udržuje ve členu `BytecodeInterpreter._bcp` [5]. Šablonový interpret si udržuje adresu právě vykonávané instrukce v registru `esi` [7]. Šablonový intepret je výchozím interpretem a může být až 10x rychlejší [33].

Zásobník vlákna

Tento zásobník je vytvořen pro každé JVM vlákno a slouží pro ukládání aktivačních rámců při volání metod.

HotSpot nevytváří žádnou dodatečnou strukturu, ale používá přímo nativní zásobník vlákna.

Run-Time Constant Pool

Tato struktura je uložena v oblasti Method Area a reprezentuje část Class file formátu, kterou je `constant_pool` tabulka. Struktura vzniká ve chvíli, kdy JVM načítá Class file jednotku.

HotSpot reprezentuje Run-Time Constant Pool třídami `constantPoolOopDesc` [9] a `constantPoolCacheOopDesc` [10].

Třida `constantPoolOopDesc` obsahuje data jako konstanty (run-time reprezentace záznamu `CONSTANT_Integer_info` apod.) a `constantPoolCacheOopDesc` obsahuje data jako reference na instance třídy `methodOopDesc`, které reprezentují metody (záznam `CONSTANT_MethodRef_info` po dynamickém linkování).

Když se např. provádí instrukce `invokevirtual`, tak pokud patričný záznam v `constantPoolCacheOopDesc` ještě není slinkován, tak se tak učiní a další provádění instrukce, která se bude odkazovat na stejný záznam, už nevyžaduje linkování [4].

Rámce

Rámce (*z angl. Frames*) hrají důležitou roli při vykonávání bajkódu. Vždy je v jednom vlákně aktivní pouze jeden rámec a to rámec metody, která se právě vykonává. Hlavní část rámce tvoří zásobník operandů a pole lokálních proměnných. Po skončení metody dochází k odstranění rámce.

HotSpot ukládá aktivační rámce přímo na nativním zásobníku vlákna. Struktura rámce se liší podle toho, jestli je použit Cpp interpret nebo Šablonový intepret [12]. Např. velikost zásobníku operandů u Šablonového interpretu se dynamicky mění, protože jeho vrchol je uložen v registru `esp` a vkládání hodnot se realizuje nativní instrukcí `push`. Naproti tomu u Cpp interpretu je velikost zásobníku fixně dána a manipuluje se s ním pomocí třídy `BytecodeInterpreter`. Oba typy rámců obsahují kromě oblasti pro lokální proměnné a operandy také hlavně oblast pro ukládání záznamů o zamčených objektech. Při volání interpretované metody se část rámce volajícího stane součástí nově vytvořeného rámce volaného. Toto je možné, protože lokální proměnné jsou uloženy na začátku rámce a zásobník operandů na konci. Protože parametry metody jsou očekávány v lokálních proměnných, může se konec zásobníku operandů volajícího stát začátkem lokálních proměnných volaného. Díky tomu není nutné hodnoty kopírovat a vytvoření nového rámce je rychlejší.

Aktivní rámec vlákna je možné identifikovat ze členu `JavaThread._anchor` [14].

Kapitola 3

Překlad zdrojových souborů

Program v jazyce Java se sestává z jednoho či více souborů s příponou `.java`, které obsahují definice tříd. Java nemá globální prostor, jako např. jazyk C a veškerý kód musí být rozdělen do tříd. Před spuštěním programu musí být zdrojové kódy nejprve přeloženy Java kompilátorem. Výstupem kompilátoru pro jeden zdrojový soubor je typicky jeden či více souborů s příponou `.class`, odpovídajících Class file formátu, které kromě dalších dat obsahují kód metod, který se nazývá Java bajtkód (*z angl. Java bytecode*). Každý `.class` soubor odpovídá definici jedné java třídy. Běhové prostředí Javy už pracuje pouze s `.class` soubory. Formát `.class` souborů, kterému se detailněji věnuje další kapitola, a Java bajtkód, je definován specifikací JVM.

Výhody Class file formátu

Překlad zdrojových souborů do jednotek odpovídajících Class file formátu má řadu výhod.

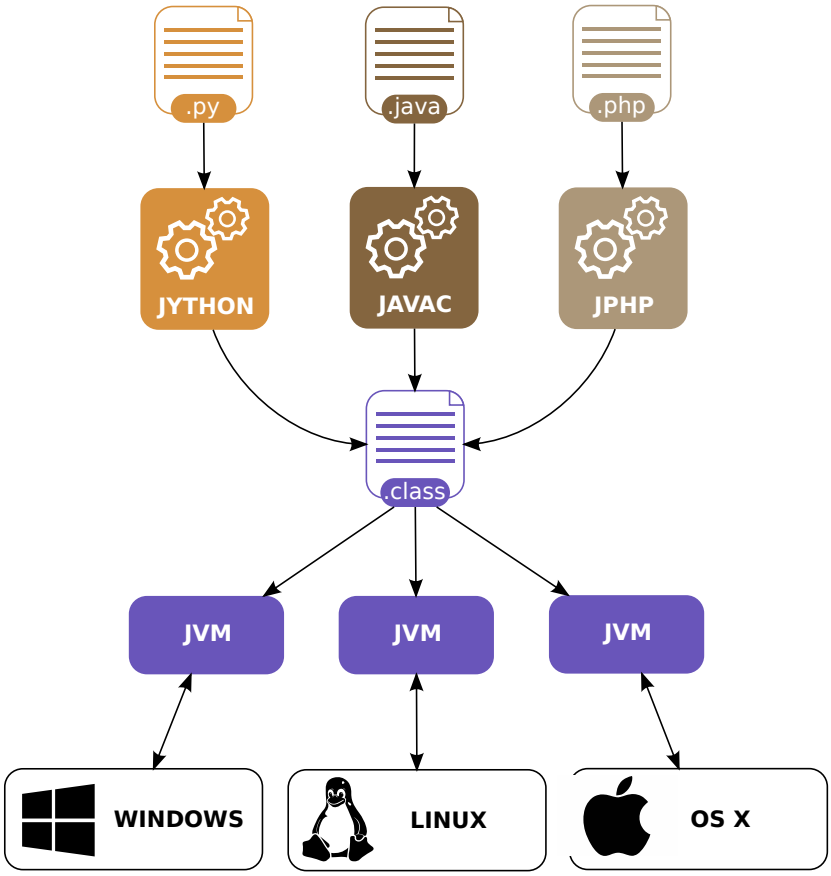
- **rychlejší interpretace** - není nutné zdrojové kódy převádět do mezikódu při každém prvním použití po spuštění programu, jako to standardně dělá např. jazyk PHP [51] se svými skripty
- **platformová nezávislost** - protože Class file formát není nijak svázán s konkrétní platformou, program je plně přenositelný a jedinou nutností je, aby pro danou platformu existovala implementace virtuálního stroje a případně tříd, které mají nativní metody
- **nezávislost na programovacím jazyce** - jakýkoliv programovací jazyk, pro který bude existovat překladač generující Class file formát, může benefitovat z vlastností JVM a používat ji jako své běhové prostředí. JVM pracuje pouze s Class file formátem a nepředpokládá použití konkrétního jazyka.

Ukázková definice třídy v jazyce Java

Listing 3.1: Definice třídy v jazyce Java

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Transformace zdrojových textů



Obrázek 3.1: Transformace ze zdrojových textů ke spuštění aplikace

3.1 Datové typy

JVM stejně jako Java rozděluje datové typy na dvě základní kategorie - primitivní typy a referenční typy.

Primitivní datové typy

Mezi primitivní datové typy patří datové typy pro celá čísla, čísla s plovoucí řádovou čárkou, datový typ pro logické hodnoty a datový typ `returnAddress`.

Překladač generuje pro různé primitivní typy jiné instrukce, není tedy nutné proměnné jakkoliv označovat.

Přehled primitivních datových typů

V závorce je uvedena velikost daného typu.

- **celočíselné datové typy** - mezi tyto datové typy patří `byte(8)`, `short(16)`, `int(32)`, `long(64)` a `char(16)`. Všechny tyto typy, s výjimkou typu `char`, jsou znaménkové a jejich defaultní hodnota je 0. Typ `char` je neznaménkový, slouží pro ukládání znaků v kódování UTF-16 [54] a jeho defaultní hodnota je `\u0000`.

- **datové typy s plovoucí řádovou čárkou** - mezi tyto typy patří `float(32)`, který odpovídá formátu IEEE 754 [45] s jednoduchou přesnosti, a `double(64)`, který odpovídá formátu IEEE 754 s dvojnásobnou přesností.
- **datový typ boolean** - slouží pro ukládání logických hodnot `true` a `false`. Defaultní hodnota je `false`. Pro tento datový typ neexistují žádné specializované instrukce a veškeré operace používají instrukce pro datový typ `int` nebo `byte`. Hodnota `false` je reprezentována jako 0 a hodnota `true` jako 1.
- **datový typ returnAddress** - tento typ je používan instrukcemi `jsr`, `ret` a `jsr_w` a reprezentuje ukazatel na instrukci metody. Tento typ nelze přímo použít.

Referenční datové typy

JVM rozeznává tří druhy referenčních datových typů. Defultní hodnotou těchto typů je speciální hodnota `null`.

- **reference na třidu** - hodnotou je reference na instanci třídy, která neimplementuje žádné rozhraní
- **reference na rozhraní** - hodnotou je reference na instanci třídy, která implementuje alespoň jedno rozhraní nebo reference na pole, které implementuje nějaké rozhraní
- **reference na pole** - hodnotou je reference na pole

3.2 Java bajtkód

Java bajtkód čítá celkem 201 instrukcí. Jednotlivé instrukce jsou relativně jednoduché. Každá instrukce se skládá z operačního kódu a případných operandů. Operační kód instrukce má velikost 1 bajt a jednoznačně identifikuje operaci, která se má provést a také počet operandů a jejich velikost. Výjimku tvoří instrukce s proměnnou délkou, jako např. instrukce `lookupswitch`, kde celkovou velikost nelze určit přímo z operačního kódu.

Pokud má operand větší velikost než 1 bajt, ukládá se jeho hodnota od nejvyššího bajtu po nejnižší (big-endian [43]).

K předchozím 201 instrukcím je zde navíc ještě jedna speciální instrukce s operačním kódem `0xca` a symbolickým názvem `breakpoint`. Tuto instrukci mohou použít debuggery [42] pro implementaci bodu přerušení (*z angl. breakpoint*) [39].

HotSpot má navíc ještě několik interních instrukcí [6] jako např. `_fast_agetfield`, které v průběhu programu nahrazují originální instrukce bajtkódu [15] a jejichž operandy mají takový význam, aby došlo ke zrychlení procesu interpretace.

3.2.1 Typy instrukcí

Většina instrukcí slouží pro přípravu zásobníku operandů pro hlavní typy instrukcí jako jsou instrukce pro volání metod či manipulaci s datovými členy tříd.

Některé instrukce existují ve dvou verzích a to s implicitním a explicitním operandem. Výhoda implicitního operandu je ušetření místa potřebného pro bajtkód a také o něco rychlejší interpretace, protože explicitní operand není potřeba načítat. Příkladem je např. instrukce `dload_1` a `dload`.

Datové typy `byte, boolean, short` a `char` nemají specializované instrukce jako typy `int, long, float` a `double` a jsou manipulovány pomocí instrukcí pro typ `int`. Jednu z mála výjimek představují instrukce pro manipulaci s prvky pole, kdy jsou použity specializované instrukce pro všechny typy kromě `boolean`, který v tomto případě používá instrukce typu `byte`.

HotSpot Jak už bylo uvedeno v předchozí kapitole, HotSpot ukládá rámce, a tedy i zásobník operandů a pole pro lokální proměnné, na nativním zásobníku vlákna.

Tento fakt určitým způsobem souvisí s tím, proč není nutné mít specializované instrukce pro manipulaci s datovými typy jako `short`, jejichž velikost je menší než 4 bajty.

Pokud jsou hodnoty těchto datových typů na zásobníku ukládány jako 4 bajtové, tedy velikost typu `int`, je automaticky vynuceno 4 bajtové zarovnání (předpokládá se, že bázová adresa zásobníku je také zarovnávna) a nebude docházet k neefektivnímu načítání hodnot. Tento jev je blíže popsán v jedné z dalších kapitol.

Naproti tomu v poli by bylo neefektivní ukládat menší datové typy jako `int`. Narozdíl od zásobníku operandů či pole pro lokální proměnné je datová část objektu, reprezentujícího pole, homogenní strukturou a slouží tedy pro ukládání hodnot pouze jednoho datového typu. Všechny typy pak mohou být ukládány ve svých velikostech a případné zarovnání postačí vynutit jen jednou před prvním prvkem. Proto pro manipulaci s prvky pole existují specializované instrukce i pro ostatní typy jako `char`.

Následuje přehled všech typů instrukcí vždy s několika představiteli. Stěžejní typy instrukcí jsou popsány všechny. Velikost operandů jednotlivých instrukců je vždy 1 bajt, v opačném případě je za dvojtečkou uveden výraz udávající velikost v bajtech (op:4 $\longrightarrow$ operand op má velikost 4 bajty).

Instrukce pro přesun hodnot mezi zásobníkem a lokálními proměnnými

- **(0x21) iload op1** - vloží na zásobník operandů `int` hodnotu, nacházející se v poli lokálních proměnných na indexu `op1`
- **(0x27) dload_1** - vloží na zásobník operandů `double` hodnotu, nacházející se v poli lokálních proměnných na indexu 1 (implicitní operand)
- **(0x38) fstore op1** - vyjme `float` hodnotu z vrcholu zásobníku operandů a uloží ji do pole lokálních proměnných na index `op1`

Instrukce pro uložení konstant na zásobník

- **(0x11) sipush op1 op2** - na zásobník operandů vloží `short` hodnotu získanou z operandů `op1` a `op2`
- **(0x12) ldc op1** - podle typu záznamu na indexu `op1` v Run-Time Constant Pool tabulce vloží na zásobník operandů `int`, `float` nebo referenci na objekt
- **(0x14) ldc2_w op1 op2** - podle typu záznamu v Run-Time Constant Pool tabulce na indexu získaného z operandů `op1` a `op2` vloží na zásobník operandů buď `long` nebo `double`
- **(0x1) aconst_null** - na zásobník operandů vloží hodnotu `null`

Aritmetické instrukce

JVM poskytuje instrukce pro operace sčítání, odčítání, násobení, dělení, zbytek po dělení, negace, bitový posun, bitový součet, bitový součin, bitová nonekvivalence, inkrementace lokální proměnné a porovnání

- **(0x61) ladd** - vyjme dvě `long` hodnoty ze zásobníku operandů, sečte je a součet vloží zpět jako `long` hodnotu
- **(0x84) iinc op1 op2** - k lokální proměnné na indexu `op1` přičte hodnotu danou operandem `op2`
- **(0x96) fcmpg a (0x95) fcmpl** - obě instrukce ze zásobníku operandů vyjmou dvě `float` hodnoty `A` a `B` a porovnají je. Na zásobník operandů poté vloží `int` hodnotu -1, pokud je `A` větší než `B`, nebo hodnotu 1 pokud je `A` menší než `B`, nebo hodnotu 0 pokud jsou si hodnoty `A` a `B` rovny.

  Rozdíl mezi těmito instrukcemi je v hodnotě vložené na zásobník operandů v situaci, kdy je hodnota `A` nebo `B` rovna `NaN`. Instrukce `fcmpg` v tomto případě vkládá hodnotu 1 a `fcmpl` hodnotu -1. Kompilátor vygeneruje instrukci `fcmpg` nebo `fcmpl` na základě použitého operátoru porovnání tak, aby nemohlo dojít k nejednoznačnosti výsledku.

  Následující tabulka ukazuje, jaké instrukce jsou generovány pro jednotlivé případy. TOS vyjadřuje stav vrcholu zásobníku (*z angl. Top Of Stack*).

Tabulka 3.1: Generování instrukcí `fcmpg` a `fcmpl`

výraz	**porovnání**	**ok/fail TOS**	**skok**
$A == B$	`fcmpg/fcmpl`	0/1,-1	`ifne fail_offset`
$A > B$	`fcmpg`	-1/0,1	`ifge fail_offset`
$A < B$	`fcmpl`	1/0,-1	`ifle fail_offset`
$A >= B$	`fcmpg`	-1,0/1	`ifgt fail_offset`
$A <= B$	`fcmpl`	1,0/-1	`iflt fail_offset`

- **(0x7e) iand** - vyjme ze zásobníku operadnů dvě `int` hodnoty, provede jejich bitový součin a výsledek vloží na zásobník operandů jako `int`

Instrukce pro konverzi mezi číselnými datovými typy

Lze provádět konverzi z menšího datového typu na větší, v tom případě může dojít ke ztrátě informace, nebo z většího datového typu na menší. Ke ztrátě informace může dojít i při převádění mezi datovými typy stejných velikostí, pokud je jeden typ celočíselny a druhý s plovoucí řádovou čárkou. Podporované jsou následující konverze:

- int $\longrightarrow$ long, float, double, byte, short, char
- long $\longrightarrow$ float, double, int
- double $\longrightarrow$ int, long, float
- float $\longrightarrow$ double, int, long

- **(0x8a) l2d** - vyjme ze zásobníku operadnů `long` hodnotu a podle standardu IEEE 754 provede konverzi na `double`. Výsledná hodnota je vložena na zásobník operandů.
- **(0x92) i2c** - vyjme ze zásobníku operandů `int` hodnotu, ořízne ji na velikost typu `char` a provede neznaménkové rozšíření na typ `int`. Výsledná `int` hodnota je vložena na zásobník operandů.

Instrukce pro vytváření instancí tříd

Pouze jedna instrukce `new`.

- **(0xbb) new op1 op2** - vytvoří instanci třídy identifikované ze záznamu typu `CONSTANT_Class_info`. Záznam je nalezen pod indexem daným operandy `op1` a `op2` a nesmí identifikovat rozhraní. Reference na instanci je vložena na zásobník operandů. Datové členy instance jsou inicializovány na své výchozí hodnoty, ale samotná instance ještě není plně incializována, protože nebyl vyvolán konstruktor.

Instrukce pro manipulaci s datovými členy tříd

Pro přístup ke statickým členům se používají instrukce `getstatic` a `putstatic`. Pro přístup k instančním členům se používá dvojice instrukcí `getfield` a `putfield`.

Při ukládání hodnot do datového členu musí být vkládané hodnoty kompatibilní s deskriptorem tohoto členu.

- **(0xb2) gestatic op1 op2** - vloží na zásobníku operandů hodnotu statického datového členu třídy, která je identifikována skrze záznam `CONSTANT_FieldRef_info`. Tento záznam se nachází se na indexu získaného z operandů `op1` a `op2`
- **(0xb3) putstatic op1 op2** - vloží hodnotu vyjmutou z vrcholu zásobníku operandů do statického datového členu třídy, identifikované skrze záznam `CONSTANT_FieldRef_info`. Tento záznam se nachází na indexu získaného z operandů `op1` a `op2`
- **(0xb4) getfield op1 op2** - vloží hodnotu datového členu objektu, odkazovaného referencí vyjmutou ze zásobníku operandů, na zásobník operandů. Datový člen je identifikována záznamem `CONSTANT_FieldRef_info`, který se nachází na indexu získaného z operandů `op1` a `op2`
- **(0xb5) putfield op1 op2** - z vrcholu zásobníku operandů vyjme hodnotu a referenci na objekt, do jehož datového členu tuto hodnotu uloží. Datový člen je identifikován záznamem `CONSTANT_FieldRef_info`, který se nachází na indexu získaného z operandů `op1` a `op2`.

Instrukce pro vytváření polí

Tři instrukce `newarray, anewarray, multianewarray`.

- **(0xbc) newarray op1** - vytvoří jednodimenzionální pole primitivního typu, který je dán hodnotou operandu `op1`. Počet prvků pole musí být uložen na zásobníku operandů jako `int` hodnota, která je poté vyjmuta a nahrazena referencí na nově vzniklé pole. Všechny prvky jsou incializovány svými výchozími hodnotami. Možné hodnoty operandu `op1` jsou 4 až 11, udávající primitivní datové typy `boolean, char, float,`

`double, byte, short, int, long` v tomto pořadí. Pro vícedimenzionální pole primitivního typu se používá instrukce `anewarray` nebo `multianewarray`, protože už se jedná o pole referencí.

- **(0xbd) anewarray op1 op2** - vytvoří jednodimenzální pole objektů, přesněji řečeno referencí na objekty. Postup je stejný jako u instrukce `newarray`, s tím rozdílem, že operandy `op1` a `op2` udávají index záznamu `CONSTANT_Class_info`, který identifikuje třídu, rozhraní, nebo pole.

- **(0xc5) multianewarray op1 op2 op3** - slouží pro vytvoření vicedimenzionálního pole.

  Počet dimenzí je dán operandem `op3` a počty prvků jednotlivých dimenzí musí být uloženy jako `int` hodnoty na zásobníku operandů. Počty prvků jsou vkládány v opačném pořadí, než se vyskytují ve zdrojovém kódu, tedy poslední vložená hodnota určuje počet prvků poslední dimenze (`new int[50][20] -> bipush 50; bipush 20`).

  Operandy `op1` a `op2` udávají index záznamu `CONSTANT_Class_info`, který musí identifikovat třídu pole, jehož dimenze je minimálně stejně velká jako hodnota operandu `op3`.

  Prvky dimenze `X` jsou vždy inicializovány referencemi na pole, jejichž typ je dán typem dimenze `X+1`. Pokud není počet prvků dimenze `X+1` uveden, což se stane ve chvíli, kdy je hodnota operandu `op3` menší než počet dimenzí udávaný deskriptorem třídy pole, jsou prvky dimenze `X` incializovány hodnotou `null` a proces končí.

  S tímto souvisí, kdy je při deklaraci proměnné typu vícedimenzionální pole použita instrukce `anewarray`, místo předpokládané instrukce `multianewarray`. Pokud není uveden počet prvků pro druhou dimenzi, jsou prvky první dimenze incializované hodnotou `null` a z tohoto pohledu se jedná o stejnou operaci, kterou provádí instrukce `anewarray`, tedy vytvoření pole objektů. Použití instrukce `anewarray` může být v tomto případě efektivnější.

  Na zásobníku operandů je na závěr vložena reference na nově vzniklé pole. Počty prvků jednotlivých dimenzí jsou vyjmuty.

  Následující tabulka ukazuje použité instrukce pro inicializaci polí.

Tabulka 3.2: Inicializace polí a použité instrukce

inicializace	**instrukce**
`int[] a = new int[10];`	newarray
`int[][] a = new int[10][];`	anewarray
`int[][][] a = new int[10][10][];`	multianewarray
`Object[] a = new Object[10];`	anewarray
`Object[][] a = new Object[10][10];`	multianewarray

Instrukce pro manipulaci s prvky pole

Tyto instrukce slouží pro přesouvání hodnot mezi zásobníkem operandů a prvky polí. Polem jsou myšleny objekty reprezentující pole, nikoli pole lokálních proměnných.

- **(0x31) daload** - ze zásobníku operandů vyjme index a referenci na pole a následně do zásobníku operandů vloží hodnotu, nacházející se v odkazovaném poli na daném indexu. Prvky pole musí být typu `double`.
- **(0x54) bastore** - ze zásobníku operandů vyjme referenci na pole, index a hodnotu. Tato hodnota je poté uložena do odkazovaného pole na daný index. Pole musí být typu `boolean` nebo `byte`.

Instrukce pro přímou manipulaci se zásobníkem

- **(0x57) pop** - z vrcholu zásobníku operandů vyjme 4 bajtovou hodnotu
- **(0x59) dup** - zduplikuje 4 bajtovou hodnotu na vrcholu zásobníku operandů
- **(0x5f) swap** - prohodí dvě 4 bajtové hodnoty na vrcholu zásobníku operandů

Instrukce skoku

JVM poskytuje instrukce pro podmíněné i nepodmíněné skoky a dále dvě speciální instrukce `tableswitch` a `lookupswitch`, které slouží pro efektivnější implementaci jazykového konstruktu `switch`.

Offset, udávající relativní adresu cíle skoku, se vždy počítá od adresy operačního kódu instrukce.

- **(0xa7) goto op1 op1** - realizuje nepodmíněný skok na offset daný operandy `op1` a `op2`.
- **(0xa8) jsr op1 op2** - realizuje nepodmíněný skok. Vloží na zásobník operandů hodnotu typu `returnAddress`, obsahující adresu následující instrukce a skočí na offset daný operandy `op1` a `op2`.
- **(0xa9) ret op1** - realizuje nepodmíněný skok. Skočí na adresu nacházející se v poli lokálních proměnných na indexu `op1`. Hodnota `returnAddress`, vložena na zásobník operandů např. instrukcí `jsr` proto musí být ze zásobníku operandů přesenuta do pole lokálních proměnných.
- **(0x99) ifeq op1 op2** - vyjme `int` hodnotu ze zásobníku operandů a pokud je tato hodnota 0, tak realizuje podmíněný skok na offset daný operandy `op1` a `op2`.
- **(0xa1) if_icmplt op1 op2** - ze zásobníku operandů vyjme `int` hodnoty `A` a `B` a pokud je `B` $<$ `A`, provede podmíněný skok na offset daný operandy `op1` a `op2`
- **(0xaa) tableswitch [padding] op1:4 op2:4 op3:4 offsets:(op2-op1+1)*4** - tato instrukce s proměnnou velikostí umožňuje efektivněji implementovat konstrukt `switch`, pokud jednotlivé `case` bloky používají číselné hodnoty, které následují hned za sebou (např. všechny hodnoty z intervalu 0-5).

  Za operačním kódem instrukce je vynuceno zarovnání přidáním 0 až 3 bajtů, aby se operandy vyskytovali na adrese, která je při odečtení adresy první instrukce metody dělitelná hodnotou 4 `((&op1-&method)/4==0.0)`.

  Operand `op1` udává offset `default` bloku.

Operandy `op2` a `op3` udávají nejmenší a největší hodnotu, které se vyskytují v `case` blocích.

Hodnoty offsetů, reprezentované operandem `offsets`, vyjadřují relativní skok od adresy operačního kódu instrukce `tableswitch`.

Při provádění instrukce se nejprve vyjme `int` hodnota ze zásobníku a zkontroluje se, jestli je menší než operand `op2` nebo větší než operand `op3`. V takovém případě se použije offset `default` bloku daný operandem `op1`. V opačném případě se od hodnoty odečte operátor `op2` a výsledek se použije jako index do tabulky `offsets` pro zjištění offsetu bloku, který se má vykonat.

Nalezený offset se přičtě k adrese instrukce `tableswitch` a na tomto místě se pokračuje v interpretaci.

Podmínkou je, jak bylo nazačátku zmíněno, aby hodnoty použité v `case` podmínkách byly všechny z určitého intervalu a ani jedna hodnota nechyběla. V opačném případě je tu instrukce `lookupswitch`.

- **(0xab) lookupswitch [padding] op1:4 op2:4 A-B-pairs:op2*8** - tato instrukce, stejně jako `tableswitch`, slouží pro podporu implementace konstruktu `switch` a má také proměnnou délku.

  Operand `op1` udává offset `default` bloku a musí být zarovnín stejně jako v případě `tableswitch`.

  Nyní se `lookupswitch` začíná lišit. Operand `op2` obsahuje počet dvojic `A:B`, které následují za ním.

  Symbol `A` reprezentuje `int` hodnotu použitou v `case` podmínce a symbol `B` reprezentuje offset `case` bloku. Tyto dvojice jsou seřazeny podle hodnoty symbolu `A` od nejmenší po největší.

  Při provádění instrukce se ze zásobníku vyjme `int` hodnota, která se postupně porovnáva s hodnotami `A` a při shodě se jako offset použije odpovídající `B`.

  Nalezený offset se přičte k adrese instrukce `lookupswitch` a na výsledné adrese se pokračuje v interpretaci.

Instrukce pro volání metod

JVM poskytuje několik instrukcí pro volání metod a to `invokevirtual, invokespecial, invokestatic` a `invokedynamic`. Všechny instrukce předpokládají, že jsou parametry metody vloženy na na zásobníku operandů. Při volání jsou ze zásobníku operandů vyjmuty a vloženy do pole lokálních proměnných volané metody. První parametr vkládaný na zásobník operandů má v poli lokálních proměnných index 0, druhý index 1 atd.. V případě instančních metod je prvním parametre na indexu 0 vždy objekt, na kterém je metoda volána.

- **(0xb6) invokevirtual op1 op2** - tato instrukce je použita pro volání nestatické neprivátní metody, která není konstruktorem a není metodou předka. Metoda je identifikována ze záznamu `CONSTANT_Methodref_info`, ležícího na indexu, který je dán operandy `op1` a `op2`.

- **(0xb9) invokeinterface op1 op2 op3 0** - tato instrukce slouží pro volání metod rozhraní (když je datový typ proměnné rozhraní). Metoda je identifikována ze záznamu

`CONSTANT_InterfaceMethodref_info`, ležícího na indexu, který je dán operadny `op1` a `op2`. Operand `op3` musí mít nenulovou hodnotu.

- **(0xb7) invokespecial op1 op2** - tato instrukce slouží pro volání metod předka (volání metody přes klíčové slovo `super`), privátních metod a konstruktorů. Metoda je identifikována ze záznamu
  `CONSTANT_Methodref_info`, ležícího na indexu, který je dán operandy `op1` a `op2`.

- **(0xb8) invokestatic op1 op2** - tato instrukce se používá pro volání statických metod. Metoda je identifikována ze záznamu `CONSTANT_Methodref_info`, ležícího na indexu, který je dán operandy `op1` a `op2`.

- **(0xba) invokedynamic op1 op2 0 0** - tato instrukce slouží pro podporu dynamicky typovaných jazyků a její využití je popsáno v samostané kapitole. Java kompilátor ji negeneruje.

Následující tabulka předpokládá existencí třídy `AA extends BB implements IAA` a zobrazuje použité instrukce při volání různých metod z těla metody třídy `AA` (pomyslná `this` reference se tedy odkazuje na objekt třídy `AA`). Statické metody jsou vždycky volány pomocí `invokestatic` a konstruktory pomocí `invokespecial`, nejsou proto uvedeny.

Tabulka 3.3: Použité instrukce pro různá volání metod

přístup	**výraz**	**instrukce**
public/protected	`aa_method()`	invokevirtual
private	`aa_method()`	invokespecial
public/protected	`bb_method()`	invokevirtual
public/protected	`super.bb_method()`	invokespecial
public	`iaa_method()`	invokevirtual
public	`((IAA)this).iaa_method()`	invokeinterface

Instrukce pro předávání návratových hodnot

Tyto instrukce slouží pro předávání návratových hodnot metod a jsou rozlišeny podle typu hodnoty, kterou vrací.

Návratová hodnota se musí nacházet na vrcholu zásobníku opernadů, odkud je vyjmuta a umístěna do zásobníku operandů volajícího (s výjimkou instrukce `return`).

- **(0xac) ireturn** - slouží pro vrácení hodnot typu `boolean, byte, short, char` nebo `int`.

- **(0xb1) return** - slouží pro vrácení z metody, jejíž návratová hodnota je typu `void` a tedy nevrací žádnou hodnotu

Instrukce pro vyhození výjimky

JVM poskytuje pouze jednou instrukci `athrow`.

- **(0xbf) athrow** - na vrcholu zásobníku musí být reference na instanci třídy `Throwable` nebo nějakého jejího potomka.

Při provádění této instrukce je tato reference ze zásobníku vyjmuta a začne se hledat blok kódu v aktuální metodě, který by tuto výjimku mohl obsloužit. Pokud je takový blok nalezen, je výjimka vložena zpět na zásobník, který je předtím vymazán a začne se vykonát kód tohoto bloku. Poté může metoda dál pokračovat. V opačném případě je metoda ukončena a pokračuje se prohledáváním metody volajícího.

Bloky pro obsluhu výjimek jsou popsány v atributu `Code`, který se vyskytuje v atributech metody.

Instrukce pro podporu synchronizace

JVM poskytuje dvojici instrukcí `monitorenter` a `monitorexit`. S každým objektem je asociován monitor, nad kterým tyto instrukce operují.

- **(0xc2) monitorenter** - tato instrukce vyjme ze zásobníku referenci na objekt a pokusí se odkazovaný objekt zamknout
- **(0xc3) monitorexit** - na zásobníku musí být reference na objekt, jehož monitor musí být vlastněn aktuálním vláknem. Při provádění této instrukce je tato reference vyjmuta a počet vstupů do monitoru odkazovaného objektu je snížen o 1. Pokud je nyní hodnota čítače vstupů 0, vlákno již monitor nevlastní a další vlákna se mohou pokusit si ho přívlastnit.

Další instrukce

Zde je popsáno několik dalších zajímavých instrukcí.

- **(0xbe) arraylength** - ze zásobníku operandů vyjme referenci na pole a vloží `int` hodnotu, udavájící délku pole
- **(0xc1) instanceof op1 op2** - ze zásobníku operandů vyjme referenci a pokud je reference stejného typu jako typ identifikovaný ze záznamu `CONSTANT_Class_info`, který se nachází na indexu daného operandy `op1` a `op2`, pak na zásobník operandů vloží hodnotu 1. V opačném případě vloží na zásobník operandů hodnotu 0.

Kapitola 4

Class file formát

Class file formát popisuje strukturu binární jednotky reprezentující zkompilovanou třídu nebo rozhraní. Tato jednotka je zpravidla výstupem překladače, ale může být také vygenerována např. pomocí knihovny pro manipulaci s bajtkódem. Všechny vícebajtové hodnoty jsou uloženy ve formátu big-endian [43], kdy jsou jednotlivé bajty ukládány od nejvíce významného po nejméně významný. Hodnoty nejsou nijak zarovnány a jsou umístěny hned za sebou. Z pohledu návrhu disassembleru je pochopení tohoto formátu nejdůležitější částí.

4.1 Struktura class file formátu

Následující tabulka zobrazuje strukturu class file formátu. Datové typy `u1`, `u2` a `u4` mají velikosti 1, 2 a 4 bajty.

Tabulka 4.1: Struktura class file formátu

typ	**název**	**počet**
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

- **magic** - speciální hodnota `0xCAFEBABE`, identifikující class file formát
- **minor_version, major_version** - dohromady `(major_version:minor_version)` tvoří číslo určující verzi class file formátu

- **constant_pool_count** - tato hodnota je rovna počtu záznamů v `constant_pool` tabulce plus 1
- **constant_pool** - tato tabulka obsahuje různé záznamy, které jsou popsány později
- **access_flag** - maska příznaků, určujících přístupová práva a vlastnosti jednotky

Tabulka 4.2: Přehled možných příznaků jednotky

název	**hodnota**	**klíčové slovo**	**popis**
ACC_PUBLIC	0x0001	public	přístup i z vnějšku balíčku
ACC_FINAL	0x0010	final	nelze dědit
ACC_SUPER	0x0020		pouze pro zpětou kompatibilitu
ACC_INTERFACE	0x0200	interface	jedná se o rozhraní
ACC_ABSTRACT	0x0400	abstract	nesmí být instancováno
ACC_SYNTHETIC	0x1000		vygenerováno kompilátorem
ACC_ANNOTATION	0x2000	@interface	jedná se o anotaci
ACC_ENUM	0x4000	enum	jedná se o výčtový typ

- **this_class** - index do `constant_pool` tabulky, kde se nachází záznam typu `CONSTANT_Class_info`, který reprezentuje tuto jednotku
- **super_class** - pokud je tato hodnota 0, musí tato jednotka reprezentovat třídu `Object`, což je jediná třída bez přímého rodiče. V opačném případě je to index do `constant_pool` tabulky, kde se nachází záznam typu `CONSTANT_Class_info`, který reprezentuje přímého předka.

  Pokud tato jednotka reprezentuje rozhraní, `super_class` index musí vést na záznam reprezentující třídu `Object`.
- **interfaces_count** - udává počet přímých rozhraní této jednotky
- **interfaces** - každá hodnota tohoto pole musí být indexem do `constant_pool` tabulky, kde se nechází záznam `CONSTANT_Class_info`, reprezentující přímé rozhraní této jednotky. Pole je seřazené tak, aby reflektovalo pořadí, v jakém byly rozhraní uváděny v deklaraci třídy (zleva doprava)
- **fields_count** - hodnota udává počet záznamů v tabulce `fields`
- **fields** - tato tabulka musí být složena ze záznamů typu `field_info`, které poskytují informace o datových členech této jednotky. Zděděné prvky zahrnuty nejsou.
- **methods_count** - udává počet záznamů v tabulce `methods`
- **methods** - tato tabulka musí být složena ze záznamů typu `method_info`, které poskytují informace o metodách jednotky. Kromě normálních metod jsou zahrnuty i konstruktory, statické metody a statické inicializační metody. Tabulka nezahrnuje zděděné metody.
- **attributes_count** - udává počet záznamů v tabulce `attributes`
- **attributes** - tato tabulka musí být složena ze záznamů typu `attribute_info`

4.2 Constant Pool

Constant Pool, reprezentovaný v popisu struktury class file formátu jako tabulka `constant_pool`, je tvořen různými typy záznamů, které obsahují data potřebná pro korektní nahrání jednotky a interpretaci bajtkódu metod.

Záznam je obecně reprezentován datovým typem `cp_info` a konkrétní záznamy si pak lze představit jako přetypování hodnoty tohoto obecného typu na typ konkrétní. Constant Pool zabírá největší část jednotky.

Tabulka 4.3: Struktura záznamu `cp_info`

typ	**název[počet]**	**popis**
u1	tag	identifikuje typ záznamu
u1	info[2+]	data specifická pro typ záznamu

4.3 Datové členy

Každý datový člen je popsán záznamem typu `field_info`.

Tabulka 4.4: Struktura záznamu `field_info`

typ	**název[počet]**	popis
u2	access_flags	přístupová práva a vlastnosti
u2	name_index	`CONSTANT_Utf8_info` reprezentující nekvalifikované jméno
u2	descriptor_index	`CONSTANT_Utf8_info` reprezentující deskriptor
u2	attributes_count	počet atributů
attribute_info	attributes[attributes_count]	atributy datového členu

Tabulka 4.5: Přehled všech flagů

název	**hodnota**	**klíčové slovo**	**popis**
ACC_PUBLIC	0x0001	public	přístup i z vnějšku balíčku
ACC_PRIVATE	0x0002	private	přístup pouze uvnitř definující třídy
ACC_PROTECTED	0x0004	protected	přístup i z potomků
ACC_STATIC	0x0008	static	přístup přes třídu
ACC_FINAL	0x0010	final	nelze opakovaně měnit
ACC_VOLATILE	0x0040	volatile	nelze cachovat
ACC_TRANSIENT	0x0080	transient	neserializovatelné
ACC_SYNTHETIC	0x1000		vygenerováno, není ve zdrojovém kódu
ACC_ENUM	0x4000		`enum` prvek

Následuje přehled všech konkrétních Constant Pool záznamů.

Tabulka 4.6: Typy Constant Pool záznamů

tag	název	popis
7	CONSTANT_ Class_info	Reprezentuje jednotku. `name_index:CONSTANT_Utf8_info` - plně kvalifikované jméno.
9	CONSTANT_ Fieldref_info	Reprezentuje datový člen. `class_index:CONSTANT_Class_info` - třída členu, `name_and_type_index:CONSTANT_NameAndType_info` - název a deskriptor
10	CONSTANT_ Methodref_info	Reprezentuje metodu. `class_index:CONSTANT_Class_info` - třída metody, `name_and_type_index:CONSTANT_NameAndType_info` - název a deskriptor
8	CONSTANT_ String_info	Reprezentuje String konstantu. `string_index:CONSTANT_Utf8_info` - data
11	CONSTANT_ InterfaceMethodref_info	Reprezentuje metodu rozhraní. `class_index:CONSTANT_Class_info` - třída metody, `name_and_type_index:CONSTANT_NameAndType_info` - název a deskriptor
3	CONSTANT_ Integer_info	Reprezentuje `int` konstantu. `bytes` - data
4	CONSTANT_ Float_info	Reprezentuje `float` konstantu. `bytes` - data v IEEE 754
3	CONSTANT_ Long_info	Reprezentuje `long` konstantu. `high_bytes, low_bytes` - data
6	CONSTANT_ Double_info	Reprezentuje `double` konstantu. `high_bytes, low_bytes` - data v IEEE 754
12	CONSTANT_ NameAndType_info	Reprezentuje název a deskriptor členu nebo metody. `name_index:CONSTANT_Utf8_info` - název, `descriptor_index:CONSTANT_Utf8_info` - deskriptor
1	CONSTANT_ Utf8_info	Reprezentuje řetězec znaků. `length` - délka, `bytes` - data
15	CONSTANT_ MethodHandle_info	Obecná funkcionalita (metoda, manipulace s datovým členem). `reference_kind` - určuje chování (např. `REF_invokeVirtual`), `reference_index:CONSTANT_FieldRef_info` nebo `CONSTANT_MethodRef_info` nebo `CONSTANT_InterfaceMethodref_info`
16	CONSTANT_ MethodType_info	Typ metody. `descriptor_index:CONSTANT_Utf8_info` - deskriptor
18	CONSTANT_ InvokeDynamic_info	Odkazován z `invokedynamic` instrukce. `bootstrap_method_attr_index` - index do `bootstrap_methods` v `BootstrapMethods` atributu jednotky, `name_and_type_index:CONSTANT_NameAndType_info` - název a deskriptor volané metody

4.4 Metody

Každá metoda je reprezentována záznamem `method_info`.

Tabulka 4.7: Struktura záznamu `method_info`

typ	**název[počet]**	popis
u2	access_flags	přístupová práva a vlastnosti
u2	name_index	`CONSTANT_Utf8_info` reprezentující nekvalifikované jméno
u2	descriptor_index	`CONSTANT_Utf8_info` reprezentující deskriptor
u2	attributes_count	počet atributů
attribute_info	attributes[attributes_count]	atributy metody

Tabulka 4.8: Přehled všech flagů

název	**hodnota**	**klíčové slovo**	**popis**
ACC_PUBLIC	0x0001	public	přístup i z vnějšku balíčku
ACC_PRIVATE	0x0002	private	přístup pouze uvnitř definující třídy
ACC_PROTECTED	0x0004	protected	přístup i z potomků
ACC_STATIC	0x0008	static	přístup přes třídu
ACC_FINAL	0x0010	final	nelze překrýt
ACC_SYNCHRONIZED	0x0020	synchronized	synchronizovaný přístup
ACC_BRIDGE	0x0040		generováno kompilátorem
ACC_VARARGS	0x0080		proměnný počet argumentů
ACC_NATIVE	0x0100	native	neimplementováno v Javě
ACC_ABSTRACT	0x0400	abstract	bez implementace
ACC_STRICT	0x0800	strictfp	FP-strict floating-point mód
ACC_SYNTHETIC	0x1000		vygenerováno, není ve zdrojovém kódu

4.5 Atributy

Atribut je reprezentován obecným záznamem `attribute_info`. Konkrétní atribut si pak lze představit jako přetypování obecného typu `attribute_info` na typ konkrétní (např. `SourceFile`). Kompilátory mohou podle potřeby generovat vlastní atributy. Pokud JVM tyto atributy nerozpozná, musí je ignorovat.

Tabulka 4.9: Struktura obecného záznamu `attribute_info`

typ	**název[počet]**	popis
u2	attribute_name_index	`CONSTANT_Utf8_info` reprezentující název atributu
u4	attribute_length	velikost položky `info` v bajtech
u1	info[attribute_length]	data atributu

Následuje přehled známých atributů. Sloupec **výskyt** vyjadřuje informaci o tom, kde se daný atribut může vyskytovat. Možné hodnoty jsou **F** - atributy datových členů, **M** - atributy metod, **C** - atributy jednotky, **B** - atributy `Code` atributu.

Tabulka 4.10: Přehled známých atributů

název	**výskyt**	**popis**
ConstantValue	F	hodnota konstantního datového členu
Code	M	instrukce bajtkódu
StackMapTable	B	použit při verifikaci
Exceptions	M	informace o výjimkách
InnerClasses	C	informace o třídách, které nejsou členy balíčku
EnclosingMethod	C	pouze pro lokální nebo anonymní třídu
Synthetic	CMF	daný prvek je vygenerován
Signature	CMF	signatura
SourceFile	C	název souboru se zdrojovým kódem jednotky
SourceDebugExtension	C	rozšířené debug informace
LineNumberTable	B	mapování bajtkódu na řádky zdrojového kódu
LocalVariableTable	B	informace o lokálních proměnných
LocalVariableTypeTable	B	signatura lokálních proměnných
Deprecated	CMF	prvek už by neměl být používán
RuntimeVisible-Annotations	CMF	runtime přístupné anotace
RuntimeInvisble-Annotations	CMF	runtime nepřístupné anotace
RuntimeVisible-ParameterAnnotations	M	runtime přístupné anotace parametrů metody
RuntimeInvisible-ParameterAnnotations	M	runtime nepřístupné anotace parametrů metody
BootstrapMethods	C	informace o bootstrap metodách používaných při provádění instrukce `invokedynamic`

4.6 Deskriptor

Deskriptor je řetězec složen z několika značek(**B**-byte, **C**-char, **D**-double, **F**-float, **I**-int, **J**-long, **S**-short, **Z**-boolean, **V**-void, **L***ClassName*;-reference na instanci třídy *ClassName*, [-dimenze pole), identifikujících typ datového členu, metody nebo lokální proměnné. *ClassName* je plně kvalifikované jméno jednotky, kde tečky jsou nahrazeny lomítkem (`java.lang.Object` ⟶ `java/lang/Object`).

Tabulka 4.11: Syntaxe deskriptoru popsaná PERL kompatibilními regulárními výrazy

alias	**regulární výraz**
BaseType	[BCDFIJSZ]
ObjectType	L*ClassName*;
ComponentType *ArrayType*	*FieldType* \[*ComponentType*
FieldType	*BaseType*|*ObjectType*|*ArrayType*
VoidType	V
ParameterType	*FieldType*
ReturnType	*FieldType*|*VoidType*
MethodType	\(*ParameterType**\)*ReturnType*

Deskriptor datového členu nebo lokální proměnné odpovídá zástupnému jménu *FieldType* a deskriptor metody zástupnému jménu *MethodType*.

Např. deskriptor datového členu `Object[][][]` je *[[[Ljava/lang/Object;* a deskriptor metody `short (int a, double b)` je *(ID)S*.

Kapitola 5

Dynamicky typované jazyky a Java Virtual Machine

Java Virtual Machine (*dále jen JVM*) kromě staticky typovaných jazyků jako Java[1] umožňuje běh i pro jazyky typované dynamicky. U těchto jazyků kontrola typů neprobíhá při překladu, ale v průběhu vykonávání kódu. Podporu dynamicky typovaných jazyků zajišťuje zejména instrukce `invokedynamic`. Příkladem takového jazyka je např. jazyk Golo [56].

Když kompilátor dynamicky typovaného jazyka kompiluje kód do bajtkódu, tak v místech, kde neví, jaký je skutečný typ proměnné (např. parametr metody), generuje při volání metody na dané proměnné místo běžných instrukcí pro volání metody instrukci `invokedynamic`.

Výskyt této instrukce se nazývá *dynamic call site* [16]. Tato instrukce se neodkazuje na záznam `CONSTANT_Methodref_info` nebo `CONSTANT_InterfaceMethodref_info`, protože třída není známa, ale na záznam `CONSATNT_InvokeDynamic_info`.

Když interpret poprvé interpretuje konkrétní `invokedynamic` instrukci, musí nejprve provést dynamické linkování, tedy zjistit, kterou metodu má vlastně zavolat. Ze záznamu `CONSTANT_InvokeDynamic_info`, odkazovaného instrukcí `invokedynamic`, se odvodí tzv. *call site specifier* [16], což je záznam v `BootstrapMethods` atributu `ClassFile` záznamu.

Call site specifier obsahuje index do Constant Pool tabulky na záznam `CONSTANT_MethodHandle_info` a případné dodatečné parametry pro tzv. bootstrap metodu.

Bootstrap metoda je uživatelsky definovaná metoda, v případě nějakého kompilátoru dynamického jazyka tedy spíše generována kompilátorem, než explicitně implementována programátorem aplikace, jejímž úkolem je vrátit tzv. *call site object* [16], což je instance potomka třídy `java.lang.CallSite` [18].

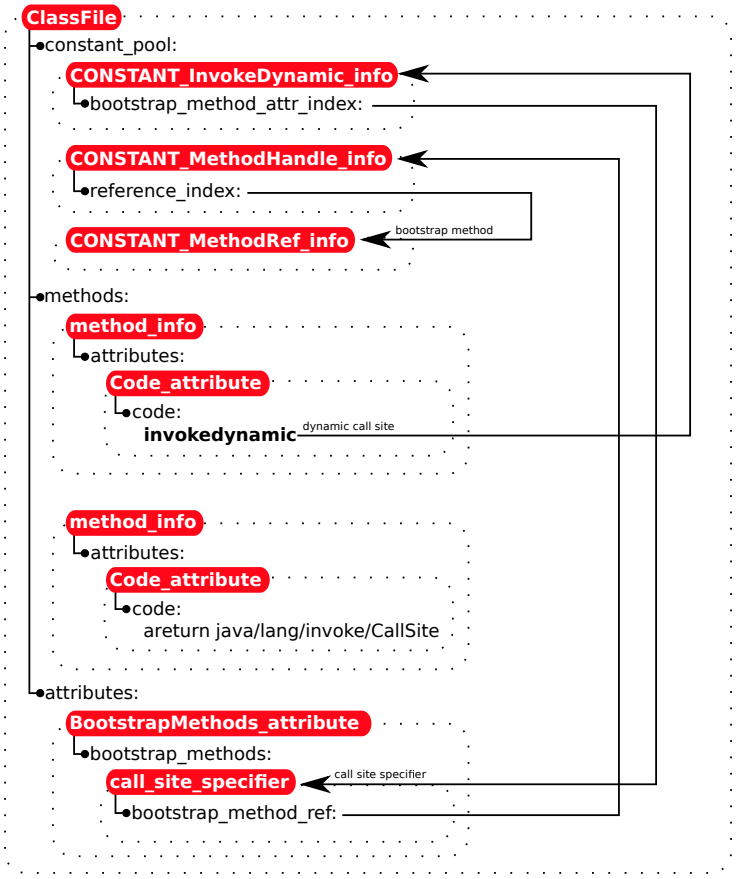
`CONSTANT_MethodRef_info` záznam, reprezentující bootstrap metodu, je získán z `CONSTANT_MethodHandle_info` záznamu odkazovaného z *call site specifier záznamu.* Položka `reference_kind` u tohoto `CONSTANT_MethodHandle_info` záznamu musí mít hodnotu `REF_invokeStatic` nebo `REF_newInvokeSpecial`.

Call site object slouží jako nepřímá reference na tzv. *call site target* , reprezentovaný instancí třídy `java.lang.MethodHandle` [19].

Výhodou tohoto přístupu, kdy bootstrap metoda nevrací přímo instanci `java.lang.MethodHandle` je v tom, že *call site object* může změnit svůj *call site target*, tedy

[1] Ve skutečnosti Java není úplně striktně staticky typovaný jazyk, protože podporuje runtime přetypování, kdy se kontroluje, jestli je objekt skutečně daného typu. Dalším příkladem dynamické kontroly typů je operátor instanceof.

být znovu slinkován s jinou instancí `java.lang.MethodHandle` a konkrétní výskyt instrukce `invokedynamic` díky tomu může v průběhu života programu spuštět různé metody.



Obrázek 5.1: Dynamické linkování `invokedynamic` instrukce

Kapitola 6

Reprezentace objektů v HotSpot

Všechny objekty v HotSpot, které jsou alokovány na Java haldě[1], jsou reprezentovány potomky třídy `oopDesc` a sdílejí několik charakteristik, vyplývajících z tohoto společného předka [24]:

1. Každý objekt obsahuje jedno slovo pro správu objektu (*z angl. mark word* [17])
2. Každý objekt má odkaz na svoji třídu (metaobjekt) (*z angl. klass pointer* [17])

Předchozí dvě charakteristiky dohromady formují hlavičku objektu (*z angl. object header* [17]). U objektů reprezentujících pole je navíc součástí hlavičky ještě délka pole.

Většina těchto objektů spravuje určitá data, která jsou pro daný typ objektu specifická. Např. u Java objektu reprezentovaného instancí třídy `instanceOopDesc` to jsou datové členy, deklarované ve zdrojovém kódu Java třídy tohoto objektu. Tato data jsou uložena s případným zarovnáním hned za objektem, který je spravuje.

Data mohou být obecně dvojího typu. Buď se jedná pouze o netypovanou paměť, kterou objekt spravuje (např. datová část již zmíněné instance třídy `instanceOopDesc`) nebo jde o instanci nějaké třídy (např. instance třídy `klassOopDesc` má ve své datové části instance potomků třídy `Klass`).

Druhý způsob je možný díky použití C++ operátoru `new` s umístěním (*z angl. placemenet new*), který umožňuje vytvořit instanci na konkrétní adrese [23].

Typicky metaobjekt vytvářeného objektu požádá Java haldu o dostatečné množství paměti pro uložení samotného objektu a dat, které má spravovat a poté vytvářený objekt inicializuje a případně pomocí operátoru `new` s umístěním vytvoří v jeho datové části správovaný objekt.

Typy objektů na Java haldě

Následující tabulka popisuje základní typy objektů na Java haldě [22].

[1] Přestože oblast Permanent Generation z určitého pohledu není součástí Java haldy [36], v této kapitole budu v termínem Java halda označovat i oblast Permanent Generation, protože třída `CollectedHeap`, která poskytuje rozhraní pro manipulaci s haldou, obsahuje i metody jako `permanent_obj_allocate`, sloužící pro vytváření objektů v Permanent Generation. Komentář k této třídě říká *"CollectedHeap is an implementation of a java heap for HotSpot* [21].

Tabulka 6.1: Typy objektů na java haldě

třída	**popis**
`instanceOopDesc`	instance java třídy
`objArrayOopDesc`	pole objektů
`typeArrayOopDesc`	pole primitivních typů
`methodOopDesc`	metoda
`constMethodOopDesc`	části metody, které se němění (např. bajtkód)
`methodDataOopDesc`	profilovací data metody
`constantPoolOopDesc`	run-time constant pool
`constantPoolCacheOopDesc`	run-time constant pool záznamy po slinkování
`markOopDesc`	slovo pro správu objektu
`klassOopDesc`	třída objektu (metaobjekt)

Určitou výjimku tvoří třída `markOopDesc`, která reprezentuje slovo pro správu objektu. Tato třída nemá žádné členy a obsahuje pouze metody pro manipulaci s tímto slovem. Samotné slovo pro správu je ve třídě `oopDesc` deklarováno jako ukazatel na `markOopDesc`. Tento ukazatel ve skutečnosti neobsahuje žádnou adresu, ale samotná data slova pro správu. Když je přes tento ukazatel zavolána metoda třídy `markOopDesc`, ukazatel `this`, který standardně obsahuje adresu objektu, na kterém je metoda volána, obsahuje adresu onoho slova pro správu objektu. Je to určitý "trik", který umožňuje manipulovat se slovem pro správu bez nutností explicitně ho některé z metod předávat (je přístupné přes `*this`).

6.1 Metaobjekty a jejich hierarchie

Jak již bylo uvedeno, každý objekt na java haldě má odkaz na svoji třídu (metaobjekt). Tento odkaz je reprezentován ukazatelem na instanci třídy `klassOopDesc`. Data spravována tímto typem objektu jsou instance potomků třídy `Klass`. Ve skutečnosti jsou to právě potomci třídy `Klass`, kteří reprezentují metaobjekty.

Protože instance třídy `klassOopDesc` je uložena na Java haldě a potomci třídy `oopDesc` mají odkaz na svůj metaobjekt, tak také všechny instance třídy `klassOopDesc` mají odkaz na svůj metaobjekt.

Z předešlého faktu lze vypozorovat jistou podobnost mezi jazykem Smalltalk, ve kterém vše je objekt, tedy i třídy jsou objekty a každý objekt má svoji třídu [31].

HotSpot objekty reprezentuje podobným způsobem, kdy každý objekt ma svůj metaobjekt, metaobjekty jsou také objekty a tedy i metaobjekt má svůj metaobjekt.

Pro potřeby určité klasifikace metaobjektů zavedu následujcí termíny.

- **metametaobjekt** - metaobjekt jiného metaobjektu (třída třídy)
- **metaobjekt metametaobjektu** - metaobjekt metametaobjektu (třída metametaobjektu)

Metametaobjekt je reprezentován instancí potomka třídy `klassKlass`. Narozdíl od jazyku Smalltalk ale jedna instance potomka třídy `klassKlass` funguje jako metametaobjekt pro více metaobjektů, takže podobnost s jazykem Smalltalk, kde každá třída má svoji vlastní metatřídu, není úplná.

Metaobjekty obecně slouží pro vytváření a manipulaci objektů v hierarchii pod nimi .

Jedním z důvodů zavedení tohoto rozdělení na metaobjekty je i snaha o minimalizaci paměťových nároků jednotlivých Java objektů, které už sami o sobě mají 8 bajtovou hlavičku. Místo vytváření virtuálních metod ve třídách jako `instanceOopDesc`, což by znamenalo další 4 bajty pro ukazatel do C++ tabulky virtuálních metod, jsou virtuální funkce přesunuty do `*Klass` tříd a tím dochází k úspoře místa.

Typy metaobjektů

Následující tabulka zobrazuje objekty a jejich metaobjekty.

Všechny objekty se jménem končícím na `*Klass` jsou uloženy v datové části instance třídy `klassOopDesc`.

V prvním sloupci tabulky je v závorce vždy uveden počet instancí v systému a typ metaobjektu (**M**-metaobjekt, **MM**-metametaobjekt, **MMM**-metaobjekt metametaobjektu).

Např. instancí třídy `typeArrayKlass` je jen 8, jedna pro každý primitivní datový typ, který je přístupný z programovacího jazyka `(booelan, byte, short, char, int, float, long, double)` a slouží jako metaobjekt pole primitivního typu.

Tabulka 6.2: Objekty a jejich metaobjekty

metaobjekt(počet-typ)	**objekt**	**popis**
`klassKlass(1-MMM)`	`klassKlass`, `instanceKlassKlass`, `objArrayKlassKlass`, `typeArrayKlassKlass`, `methodKlass`, `constMethodKlass`, `methodDataKlass`, `constantPoolKlass`, `constantPoolCacheKlass`	fixní bod
`instanceKlass(J-M)`	`instanceOopDesc`	reprezentace java třídy
`instanceMirrorKlass(1-M)`	`instanceOopDesc`	reprezentace `java.lang.Class` třídy
`instanceRefKlass(K-M)`	`instanceOopDesc`	reprezentace potomků `java.lang.ref.Reference` třídy
`objArrayKlass(L-M)`	`objArrayOopDesc`	třída pole objektů
`typeArrayKlass(8-M)`	`typeArrayOopDesc`	třída pole primitivního typu
`methodKlass(N-M)`	`methodOopDesc`	třída metody
`constMethodKlass(N-M)`	`constMethodOopDesc`	
`methodDataKlass(N-M)`	`methodDataClass`	
`instanceKlassKlass(1-MM)`	`instanceKlass`, `instanceMirrorKlass`, `instanceRefKlass`	třída c++ reprezentace java třídy
`objArrayKlassKlass(1-MM)`	`objArrayKlass`	třída c++ reprezentace třídy pole objektů
`typeArrayKlassKlass(1-MM)`	`typeArrayKlass`	třída c++ reprezentace třídy pole primitivních typů
`constantPoolKlass(1-M)`	`constantPoolOopDesc`	
`constantPoolCacheKlass(1-M)`	`constantPoolCacheOopDesc`	

klassKlass je speciální třída, která v hierarchii metaobjektů slouží jako koncový bod a její metaobjekt je ona sama. Z každého objektu se na tuto třídu lze dostat v maximálně 3

cyklech (nejdelší cesta je
`objekt->metaobjekt->metametaobjekt->metaobjekt_metametaobjektu`).

instanceKlass

Nejzajímavější třídou je `instanceKlass`, která představuje C++ reprezentaci java třídy a obsahuje kromě dalších členů hlavně tabulku virtuálních metod a tabulku metod rozhraní.

Třída `instanceKlass` má dva potomky, `instanceMirrorKlass` a `instanceRefKlass`. Třída `instanceMirrorKlass` je použita místo třídy `instanceKlass` u java objektů třídy `java.lang.Class`, jejíž instance kromě vlastních datových členů obsahují také statické datové členy třídy. Instance `instanceOopDesc` reprezentující java object třídy `java.lang.Class` je ze třídy `instanceKlass` dostupný přes prvek `_java_mirror`.

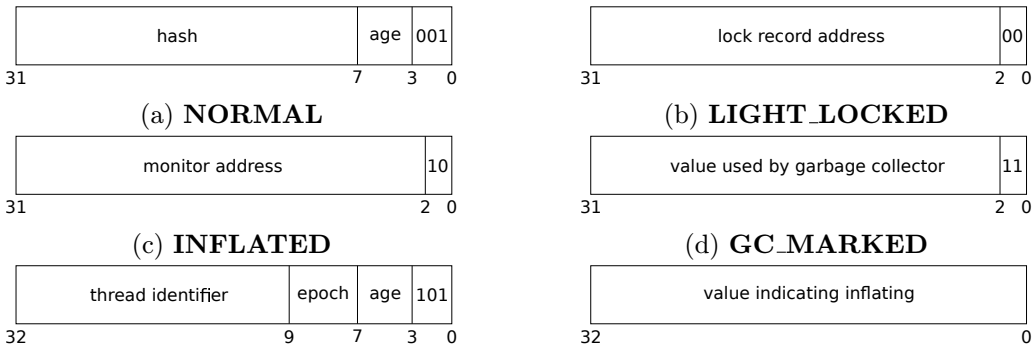
Třída `instanceRefKlass` je použita místo třídy `instnaceKlass` u instancí potomků java třídy `java.lang.ref.Reference`. Tyto java třídy mají specifickou funkcionalitu, protože více spolupracují s garbage collectorem. Jejich instance se používají ve chvíli, kdy je potřeba udržovat referenci na objekt, ale v případě, že se jedná o poslední existující referenci, tak objekt dál neudržovat naživu. Typické použití je v různých cache strukturách. Jednotliví potomci se potom líší zvolenou politikou, kdy referenci už nedovolí použít [37].

6.2 Slovo pro správu objektu

Slovo pro správu objektu (*dále jen SPSO*) má velikost 4 bajty, slouží k ukládání informací o stavu objektu a je využíváno několika subsystémy (garbage collector, synchronizace, uložení hashe objektu). Interpretace hodnoty SPSO se v čase mění podle aktuálního stavu objektu, který je odvoditelný ze tří nejnižších bitů [11].

Stavy SPSO

Následující obrázky zobrazují možné stavy SPSO.



(a) **NORMAL** (b) **LIGHT_LOCKED**

(c) **INFLATED** (d) **GC_MARKED**

(e) **BIASED/BIASABLE** (f) **INFLATING**

Obrázek 6.1: Možné stavy SPSO

Tabulka 6.3: Popis stavů SPSO

stav	popis
NORMAL	tento stav je výchozím stavem SPSO nového objektu při použití Lightweight Locking synchronizace.
LIGHT_LOCKED	tento stav je aktivní, pokud je objekt zamknutý použitím Lightweight Locking synchronizace
INFLATED	tento stav je aktivní, když je objekt zamknutý pomocí Heavyweight Locking synchronizace
GC_MARKED	tento stav může být aktivní pouze při cyklu garbage collectoru
BIASABLE	tento stav je výchozím stavem SPSO nového objektu při použití Store-Free Biased Locking synchronizace
BIASED	tento stav je aktivní, pokud je objekt zamknutý pomocí Store-Free Biased Locking synchronizace
INFLATING	tento stav je aktivní, když je instalován monitor (před přechodem do stavu **INFLATED**

- **hash** - tato hodnota je u objektu nastavena až ve chvíli, kdy dojde k explicitnímu volání statické metody `System.identityHashCode`, jinak má tato část hodnotu 0
- **věk (age)** - tato hodnota umožňuje garbage collectoru zjistit, kolik objekt přežil cyklů. Hodnota je navýšena při každém kopírování v rámci New Generation a po dosažení určité hodnoty je objekt přesunut do Old Generation.
- **adresa záznamu zámku (lock record address)** - ukazatel na záznam zámku, který se nachází na zásobníku vlákna, které objekt vlastní
- **adresa monitoru (monitor address)** - ukazatel na objekt, reprezentující monitor
- **hodnota používaná garbage collectorem (value used by garbage collector)** - např. ukazatel (*z angl. forwarding pointer*) na kopii objektu po zkopírování Copy Collectorem
- **identifikátor vlákna (thread identifier)** - tato hodnota je buď 0, v tom případě je objekt tzv. anonymně biasován (SPSO ve stavu **BIASABLE**), nebo obsahuje identifikátor vlákna, které ho vlastní (SPSO ve stavu **BIASED**)
- **hodnota signalizující instalaci monitoru (value indicating inflating)** - speciální hodnota, podle které ostatní vlákna mohou poznat, že právě probíhá instalace monitoru
- **epoch** - tato hodnota umožňuje invalidovat stav **BIASED** u všech objektů určité třídy

6.3 Struktura objektu v paměti

Jak bylo uvedeno v předešlé části, každý Java objekt má slovo pro správu a odkaz na svoji třídu. Obě tyto hodnoty mají velikost 4 bajty.

Každý objekt, tedy i ten, co nemá žádné další datové členy, má tedy velikost minimálně 8 bajtů.

Dalším pravidlem je, že objekty jsou v paměti zarovnány na hranici 8 bajtů, což spolu s uspořádáním datových členů objektu zajišťuje efektivnější přístup do paměti [20]. Konkrétně uvažovaná architektura IA-32 pracuje efektivněji, pokud jsou data v paměti zarovnána na hranici 4 bajtů, protože není potřeba data velikosti větší než 1 bajt číst ve více paměťových cyklech a poté je rekonstruovat (např. čtení 4 bajtové hodnoty z adresy 6 by způsobilo načtení hodnoty z adresy 4 a z adresy 8 a následnou rekonstrukci) [25].

Toto souvisí s konstrukcí procesorů i pamětí DRAM. Adresová sběrnice by mohla mít např. pouze 30 bitů a zbylé 2 bity uvažovat jako vynulované, tedy že načítaná hodnota je zarovnána na 4 bajty. Některé platformy dokonce způsobují chybu sběrnice, pokud dochází k pokusu o přístup do nezarovnané paměti.

S předchozím odstavcem souvisí, jak jsou objekty v paměti uloženy. Bylo by neefektivní jednotlivé datové členy objektu ukládat v takovém pořadí, v jakém jsou deklarované v Java třídě, protože by lehce došlo k takovému uspořádání, že by každý přístup k datovému členu vyžadoval více paměťových přístupu a následnou rekonstrukci (např. prokládání jedno bajtových a čtyř bajtových hodnot). HotSpot ukládá všechny datové typy tak, aby byly zarovnány na svých velikostech.

Následující dvě podkapitoly popisují dva možné způsoby uspořádání objektu v paměti.

Instance tříd, jejichž přímým rodičem je java.lang.Object

Objekty, jejichž třídy přímo rozšiřují třídu `java.lang.Object` mají své datové členy uspořádané následujícím způsobem:

1. double a long (velikost 8 bajtů)
2. int a float (velikost 4 bajty)
3. short a char (velikost 2 bajty)
4. boolean a byte (velikost 1 bajt)
5. [případné zarovnání na 4 bajty] - pokud celková velikost všech datových členů typu short, char, boolean a byte není dělitelná čtyřmi beze zbytku, je přídáno zarovnání, aby následující reference byly pořád efektivně zarovnány
6. reference (velikost 4 bajty)
7. [případné zarovnání na 8 bajtů] - protože jsou objekty zarovnány na 8 bajtů, je na konci objektu určitý nevyužitý prostor, pokud celková velikost všech datových členu, slova pro správu objektu a odkazu na třídu není dělitelná osmi beze zbytku. Např. pokud objekt na adrese 0 zabírá 12 bajtů, stejně nelze další objekt alokovat na následující volné adrese 12, ale až na adrese 15.

Instance tříd, jejichž přímým rodičem není java.lang.Object

Datové členy různých tříd v hierarchii dědění nejsou nikdy prokládány mezi sebou a členy potomka vždy následují za členy rodiče.

Z předchozí podkapitoly vyplývá, že objekt, jehož přímý předek je `java.lang.Object`, je na závěr doplněn o dál nevyužívaný prostor.

Bylo by neefektivní nejprve za poslední datový člen předka doplnit nevyužívaný prostor až do zarovnání na 8 bajtu a poté umísťovat datové členy aktuální třídy. Místo toho HotSpot zavádí trochu jiná pravidla pro nevyužitý prostor za posledním datovým členem předka a pro uspořádání datových členů aktuální třídy.

1. přidání nevyužívaného prostoru za poslední datový člen předka až do zarovnání na 4 bajty
2. (a) pokud je zarovnání zároveň 8 bajtové nebo třída nemá žádné datové členy typu long a double
    i. double a long (velikost 8 bajtů)
    ii. int a float (velikost 4 bajty)
    iii. short a char (velikost 2 bajty)
    iv. boolean a byte (velikost 1 bajt)
    v. [případné zarovnání na 4 bajty]
    vi. reference (velikost 4 bajty)

   (b) pokud třída má datové členy typu long nebo double
    i. vyplnění 4 bajtové mezery datovými členy typu int, float, char, short, byte, reference v tomto pořadí
    ii. [případné zarovnání na 8 bajtů]
    iii. double a long (velikost 8 bajtů)
    iv. int a float (velikost 4 bajty)
    v. short a char (velikost 2 bajty)
    vi. boolean a byte (velikost 1 bajt)
    vii. [případné zarovnání na 4 bajty]
    viii. reference (velikost 4 bajty)

   (c) pokud třída nemá datové členy typu long nebo double
    i. int a float (velikost 4 bajty)
    ii. short a char (velikost 2 bajty)
    iii. boolean a byte (velikost 1 bajt)
    iv. [případné zarovnání na 4 bajty]
    v. reference (velikost 4 bajty)
3. (a) pokud v hierarchii dedění následuje další třída, opět postupovat jako v bodě 1

   (b) pokud je aktuální třída koncová, tak zarovnání na 8 bajtů

Java Object Layout je utilita, která umožňuje zobrazit, jak bude v paměti vypadat uspořádání instance určité třídy [1]. Následující příklad zobrazuje výstup utility Java Object Layout pro třídu `B` a demonstruje předchozí pravidla.

Listing 6.1: Třída `A`

```
public class A {
    byte a;



}
```

Listing 6.2: Třída `B`

```
class B extends A {
    char a;
    byte b;
    A c;
    double d;
}
```

Tabulka 6.4: Výstup utility Java Object Layout pro třídu `B`

offset	velikost	typ	název	popis
0	8		(hlavička)	slovo pro správu, odkaz na třídu
8	1	byte	A.a	
9	3		(zarovnání)	následují členy B - zarovnání na 4b
12	2	char	B.a	B má double - vyplnění 4 bajtové mezery
14	1	byte	B.b	
15	1		(zarovnání)	
16	8	double	B.d	
24	4	A	B.c	
28	4		(zarovnání)	finální zarovnání na 8 bajtů
Celková velikost objektu: 32 bajtů				
Počet nevyužitých bajtů: 8				

Pole

Pole má navíc ke standardní hlavičce objektu ještě 4 bajtovou hodnotu, reprezentující délku pole. Hlavička má pak velikost 12 bajtů. U pole typu int, float, short, char, byte nebo pole objektů následují data hned za hlavičkou. U polí typu double nebo long je nejdřív přidáno 4 bajtové zarovnání. Na závěr je poté ještě přidáno případné zarovnání, aby celková velikost byla dělitelná 8.

Kapitola 7

Synchronizace přístupu k objektům v HotSpot

JVM používá blokově strukturovanou synchronizaci pomocí páru instrukcí `monitorenter` a `monitorexit`. Pro každé obdržení zámku tedy existuje odpovídající uvolnění. Kompilátor by neměl nikdy generovat kód, který předchozí podmínku nesplňuje. V opačném případě interpret dokáže tuto situaci detekovat a ošetřit, případně vyhodit výjimku.

Předchozí fakt je jedna z invariant, umožňující fungování algoritmů, které HotSpot využívá pro synchronizaci přístupu k objekům. HotSpot implementuje dva způsoby synchronizace, tzv. Lightweight Locking a Store-Free Biased Locking (SFBL).

7.1 Lightweight Locking

Tento algoritmus, dále jen LL, je postaven na hypotéze, že většina pokusů o obrdžení zámku z různých vláken neprobíhá ve stejný moment a tedy nedochází ke stavu (contention), kdy by bylo skutečně nutné použít nějaké synchronizační primitivum poskytované operačním systémem. Toto by mohlo např. vyžadovat přepnutí do jádra operačního systému, což je poměrně náročná operace a celkově v daný moment, kdy nedochází ke skutečnému střetu dvou či více vláken, zbytečně neefektivní.

Použití algoritmu lze detekovat ze tří nejnižších bitů slova pro správu objektu (SPSO), které mají binární hodnotu 001 (stav **NORMAL**) v případě, že zámek objektu není vlastněn žádným vláknem. Pokud nějaké vlákno již zámek vlastní, mají spodní dva bity binární hodnotu 00 (stav **LIGHT_LOCKED**) a předchozí bit už se neuvažuje.

Když se vlákno pokouší zamknout objekt, vytvoří se nejprve v aktivním rámci právě vykonávané metody, v oblasti určené pro záznamy o držených zámcích, nový záznam.

Do záznamu se uloží adresa objektu a aktuální hodnota SPSO. Tato kopie SPSO se nazývá displaced mark word. Pomocí CAS proběhne pokus o uložení adresy záznamu zámku do SPSO. Toto uložení proběhne uspěšně pouze pokud je objekt ve stavu **NORMAL**. Uspěšné uložení znamená přechod objektu do stavu **LIGHT_LOCKED**. V opačném případě dojde ke kontrole, jestli již objekt ve stavu **LIGHT_LOCKED** je a jestli adresa záznamu zámku v SPSO ukazuje na zásobník aktuálního vlákna.

HotSpot rámce ukládá na nativním zásobníku vlákna, proto lze podle předchozí kontroly poznat, jestli je objekt ve stavu **LIGHT_LOCKED** zamčen aktuálním vláknem a jestli se tedy jedná o rekurzivní zamčení.

V případě rekurzivního zamčení se do záznamu o zámku na místo displaced mark word

uloží hodnota 0, která později při uvolňování zámku umožňuje detekovat, že se jedná o rekurzivní uvolňení. V SPSO zůstává adresa nejstaršího záznamu zámku.

Pokud adresa záznamu zámku v SPSO neodkazuje do oblasti zásobníku akuálního vlákna, znamená to, že je objekt vlastněn jiným vlákem a musí dojít k přechodu na Heavyweight locking.

Pří odemykání objektu jsou prohledávány záznamy zámků v aktuálním rámci, dokud není nalezen záznam, jehož objekt se shoduje s odemykaným objektem. Pokud je hodnota zazálohovaného SPOS v tomto záznamu nula, znamená to, že se jedná o rekurzivní odemykání a nic se neděje. V opačném případě je pomocí CAS proveden pokus o obnovení SPOS odemykaného objektu do původního stavu. Pokud CAS neuspěje, znamená to, že mezitím objekt přešel do stavu **INFLATED**, protože se ho pokoušely zamykat další vlánka.

7.2 Store-Free Biased Locking

Tento algoritmus, dále jen SFBL, je výchozím algoritmem pro synchronizaci přístupu k objektům v HotSpot.

SFBL vychází ze stejné hypotézy jako LL, ale navíc předpokládá, že většina pokusů o obdržení zámku již zamknutého objektu je prováděna vláknem, které už k tomuto objektu zámek vlastní, tedy že se jedná o rekurzivní zamykání. Na základé této myšlenky je zefektivněn opakovaný přístup stejného vlákna k objektu, ke kterému se mezitím jiné vlákno nepokusilo přistoupit.

Konkrétně je zredukován počet přístupů do paměti na minimum a kromě úvodního zamknutí objektu se dále nepužívají atomické instrukce, což je rozdíl oproti LL, které je používá pokaždé, viz. popis algoritmu.

Nevýhodou tohoto přístupu je potom situace, kdy skutečně dojde k pokusu o zamknutí objektu, který je již zamknutý jiným vláknem. V tom případě je nutné přivést JVM do tzv. bezpečného bodu (safepoint), aby mohla být vykonána logika pro převod objektu ze stavu **BIASED** do stavu **LIGHT_LOCKED**. Termín pro převod ze SFBL na LL se nazývá bias revoke. K tomuto převodu nemusí dojít pouze při pokusu o zamčení objektu jiným vláknem, než které objekt vlastní, ale např. při zavolání metody `System.identityHashCode`, která generuje hash objektu, který má být uložen v SPSO a protože není možné ukládat v SPSO současně identifikátor vlákna vlastnící objekt a hash objektu, je nutné provést bias revoke. Obecně se po provedení bias revoke objekt nachází buď ve stavu **NORMAL** nebo **LIGHT_LOCKED**. Když je poté obejekt ve stavu typickém pro algoritmus LL, další proces synchronizace už se řídí dle LL algoritmu.

HotSpot se na základě různých heuristik dokáže automaticky přizpůsobit průběhu programu a preferovat LL před SFBL pro určitý datový typ, pokud často dochází ke stavu, kdy je nutné provést revoke bias. Tento proces se nazývá bulk revocation.

Dále HotSpot dokáže invalidovat všechny objekty určitého typu, které jsou ve stavu **BIASED**, ale objekt už ve skutečnosti vláknem zamčen není. Tento process se nazývá bulk rebiasing.

Bulk Rebiasing Tento proces využívá epoch ze SPSO objektu ve stavu **BIASED/BIASABLE**. Když je potřeba hromadně invalidovat vlastníky všech objektů určitého typu, které jsou ve stavu BIASED, přivede se JVM do bezpečného bodu, změní se epoch v prototypu SPOS (prototype mark word), který je vlastněn třídou objektu a určuje výchozí hodnotu SPOS pro nově vytvořené objekty. Tato změna způsobí, že při pokusu o zamčení

biasovatelného objektu, kdy se kontroluje, jestli epoch v SPSO objektu má stejnou hodnotu jako epoch v prototypu SPSO, bude možné detekovat, že objekt už není ve stavu BIASED, ale pouze BIASABLE a vlákno se může pokusit objekt zamknout pomocí CAS. Toto je nutné protože jakmile se vláknu podaří jednou zamknout biasovatelný objekt, zůstává jeho vlastníkem (bias owner) už napořád, čímž je elminována CAS operace při odemykání objektu, kdy by bylo jinak potřeba do SPOS odemykaného objektu opět vložit hodnotu 0 na místo identifikátoru vlákna, aby se objekt nejevil jako ve stavu BIASED, ale pouze BIASABLE.

Bulk Revocation Tento proces se využívá při potřebě konvertovat všechny objekty určitého typu, využívající SFBL, na použití LL. Typicky když JVM zdetekuje, že na objektech určitého typu dochází často k pokusu obdržet zámek, který už je vlastněn jiným vláknem. Toto se děje např. u algoritmů producer-consumer, kdy více vláken potřebuje manipulovat s nějakou sdílenou strukturou, jako fronta apod.. Před operací se JVM přivede do bezpečného bodu a prototyp SPOS se změní z **BIASABLE** na **NORMAL**. Protože v daný moment mohou existovat objekty, které jsou ve stavu BIASED a jejich zámek vlastní nějaké vlákno, musí být manipulovány zásobníky vláken, které vlastní zámky těchto objektů, tak, aby se objekty byly ve validním **LIGHT_LOCKED** stavu. Toto je jeden z důvodů, proč musí být JVM v bezpečném bodě, jinak by hrozili synchronizační problémy plynoucí z manipulace se zásobníkem cizích vláken.

7.3 Heavyweight Locking

Hevyweight locking přichází na řadu ve chvíli, kdy LL detekuje potřebu využít opravdové synchronizační primitivum pro odstavení vláken, které se pokouší zamknout objekt ve stavu **LIGHT_LOCKED**, jehož zámek ale není vlastněn vláknem, které se objekt pokouší zamknout. Biasovatelný objekt s tímto algoritmem nikdy přímo použit není, ale nejprve se provede operace revoke bias, která objekt převede do validního stavu LL algoritmu a ostatní logika už je pak vykonávána v rámci LL.

Sychronizační primitivum je reprezentováno instancí třídy `ObjectMonitor`. Tento monitor může být získán z lokálního seznamu pro konkrétní vlákno, což je efektivnější, protože není nutná synchronizace, nebo z globálního seznamu. Zamykaný objekt je nejprve pomocí CAS převeden do dočasného stavu **INFLATING**, podle kterého případně algoritmus pozná, že už se nějaké vlákno pokouší nainstalovat monitor a použije spin lock pro čekání, dokud tento dočasný stav neskončí.

Po skončení dočasného stavu **INFLATING** je objekt ve stavu **INFLATED**, což signalizuje, že v SPSO objektu je uložena adresa monitoru. Monitor si před změnou SPSO zazálohuje, takže může detekovat, jestli vlákno pokoušející se objekt zamknout je vlákno, které zámek skutečně vlastní a neblokovat toto vlákno ve frontě vláken, které čekájí na uvolnění zámku objektu. Toto je umožněno, protože před přechodem do stavu **INFLATING** byl objekt ve stavu **LIGHT_LOCKED** a tedy jeho SPSO obsahovala adresu nejstaršího záznamu zámku, ležícího na zásobníku vlákna vlastnícího tento zámek a která slouží pro detekci rekurzivního zamykání, viz. LL.

7.4 Compare-And-Swap (CAS)

Compare-and-swap je atomická operace, která umožňuje uložit do paměti hodnotu **A** pouze za předpokladu, že aktuální hodnota v této paměti je stejná, jako hodnota **B**. Typicky je pak vrácena hodnota C, která v paměti byla před změnou, podle čeho se dá poznat, jestli operace proběhla úspěšně (`success = (C == B ? true : false)`).

Atomičnost zajišťuje, že nemůže nastat situace typická při vykonání stejné funkcionality pomocí neatomických instrukcí, kdy mezi různými dvěma instrukcemi může jiný procesor, popřípadě jiné vlákno na stejném procesoru, které bylo aktivováno operačním systémem, paměť modifikovat a tak není nikdy možné s jistotou určit, že hodnota v paměti je pořád stejná, jako hodnota nahraná do registru předešlou instrukcí.

Kapitola 8

Existující nástroje pro práci s Bajtkódem

8.1 Knihovny pro manipulaci s Bajtkódem

Vzhledem k faktu, že analýza a další druhy manipulace s Bajtkódem jsou, jak již bylo v úvodu nastíněno, poměrně časté operace, vzniklo v průběhu let hned několik nástrojů, které umožňují tyto úlohy provést bez nutnosti vymýšlení vlastního řešení. Mezi typické operace, které lze s Bajtkódem provádět, patří:

- **transformace** - změna jíž existujícího class souboru. Nemusí se jednat pouze o modifikaci instrukcí metod, ale např. o změnu constant poolu.
- **generování** - sestavení výstupu, který odpovídá class formátu, ale nenáleží mu žádné zdrojové kódy
- **analýza** - různé druhy zpracování, které mohou mít výstupy např. v podobě uživatelsky přívětivého pohledu na obsah class souboru.

BCEL

Knihovna BCEL (The Byte Code Engineering Library) je ve správě Apache Foundation a jedná se o jednu z nejstarších knihoven pro manipulaci s Bajtkódem. Přestože tuto knihovnu využívá řada nástrojů, není v současnosti již příliš vyvíjena.

Javassist

Javassist (Java Programming Assistant) je knihovna, kterou vyvíjí Shigeru Chiba. Tato knihovna je specifická tím, že umožňuje používat javovské příkazy pro generování kódu a tedy není nutná znalost instrukcí JVM.

ASM

ASM knihovna od společnosti Objectweb je v současnosti nejaktivněji vyvíjena. Poskytuje poměrně jednoduché a intuitivní rozhraní, které umožňuje využít dva různé způsoby zpracování class souborů.

První způsob (Core API) je podobný zpracování XML souborů metodou SAX, kde ASM, používající návrhový vzor Visitor, očekává třídy implementující určité rozhraní, jehož

jednotlivé metody reprezentují možnou reakci na výskyt nějakého prvku class formátu. Výhodou tohoto přístupu je menší paměťová náročnost, protože není nutné v paměti udržovat celkovou reprezentaci class souboru. Tato metoda ovšem neumožňuje nějakou komplexnější analýza, kdy je nutné se např. zpětně používat již zpracované prvky. Druhý způsob (Tree API) je podobný zpracování XML souborů metodou DOM, kdy je celý class soubor načten do paměti a výsledná struktura umožňuje přistupovat ke všem dostupným prvkům libovolně. Výhodou je možnost komplexní analýzy, nevýhodou poté větší paměťová náročnost.

8.2 Disassemblery Bajtkódu

Disassemblery bajtkódu umožňují uživatelsky přívětivě zobrazit obsah `.class` souborů, zejména `Code` atribut metody, který obsahuje jednotlivé instrukce.

Pro tento uživatelsky přívětivý pohled na bajtkód zavádím termín *uživatelský bajkód*.

Následující přehled popisuje některé existující disassemblery.

Javap

Tento nástroj je spouštěn z příkazové řádky a standardně se nainstaluje při instalaci Java platformy od společnosti Oracle nebo OpenJDK. Jeho typické použití je spuštnění s parametrem `-v` následovaným názevem `.class` souboru , kdy se na výstup vypíší obecné informace o jednotce, obsah Constant Pool tabulky a detailní přehled všech implementovaných metod včetně uživatelského bajtkódu [28].

Příklad spuštění nástroje Javap

```
javap -v Main.class
```

Bytecode Visualizer

Tato nástroj ze skupiny utilit Dr. Garbage tools, vyvíjené developery Sergejem Alekseevem, Peterem Palagou a Sebastianem Reschkem, představuje plugin do vývojového prostředí Eclipse [27].

Po instalaci asociuje soubory s přípnou `.class` se svým editorem. Po otevření `.class` souboru např. skrze průzkumník projektu (*z angl. Package Explorer*), který zobrazuje i výstupní soubory překladu (*View menu* → *Customize View* → *Java output folder*), se editor otevře.

Standardně se zobrazují deklarace a uživatelský bajtkód. V nastavení pluginu (*Window* → *Preferences* → *Dr. Garbage Bytcode* → *Visualizer* → *General*) je možné nastavit např. zobrazování Constant Pool tabulky nebo tabulky mapování instrukcí na čísla řádků zdrojového kódu (LineNumberTable atribut). Unikátnější vlastností je zobrazování grafu toku vykonávání (*z angl. Control Flow Graph*), ze kterého lze vypozorovat, kde se tok větví např. kvůli instrukcím podmíněných skoků.

Bytecode Outline

Tento diassembler také funguje jako plugin do vývojového prostředí Eclipse, používá knihovnu ASM a vyvíjí ho Andrey Loskutov [26].

Po instalaci je možné zobrazit zobrazit okno (*Window* → *Show View* → *Other* → *Java* → *Bytecode*), které zobrazuje uživatelský bajtkód pro aktivní `.java` soubory. Neumožňuje ale zobrazit obsah Constant Pool tabulky. Unikátnější vlastností je možnost porovnat `.class` soubory. Celkově na mě udělal větší dojem Bytecode Visualizer.

Bytecode Viewer

Tato samostatná aplikace naprogramovaná v jazyce Java slouží zároveň jako dekompilátor a jejím autorem je Kalen Kinloch [57].

Program v sobě integruje existující nástroje, jako např. Procyon pro dekompilaci a Krakatau pro získání uživatelsky přívětivého bajtkódu. Celkově je aplikace poměrně velká a kromě zmíněného dekompilátoru či disassembleru disponuje i dalšími vlastnostmi, jako např. možnost psaní skriptů v jazyce Groovy pro potřeby analyzování kódu.

Krakatau

Krakatau je sada tří nástrojů - dekompilátor, disassembler a nástroj pro vytváření `.class` souborů. Tyto nástoje jsou realizovány jako skripty v jazyce Python a spouštějí se z příkazové řádky. Krakataou je vyvíjena Robertem Groosem [30].

Výstup disassembleru je podobný jako u nástroje Javap, ale Javap mi příjde přehldnější.

Příklad spuštění disassembleru Krakatau

```
python diassemble.py -cpool -out adresar Main.class
```

JBE - Java Bytecode Editor

Tato samostatná aplikace je naprogramována v jazyce Java a jejím autorem je Ando Saabas [29].

Aplikace má jednoduché a přehledné uživatelské prostředí a kromě zobrazení jednotlivých částí `.class` souborů, mezi kterými se dá navigovat pomocí položek v levém sloupci, umožňuje i přímou modifikaci kódu metod zapisováním symbolických názvů a operandů jednotlivých instrukcí.

Kapitola 9

Závěr

Práci jsem nestihl dokončit, takže počítám s opakovanou obhajobou. Jako důvody nezdaru bych identifkoval zbytečný zájem o ne úplně nutnou problematiku ve snaze nemít teoretickou část sestávanou pouze ze specifikace Java Virtual Machine a mít tam něco navíc. Bohužel pro sebe jsem začal spíše z opačného konce a nakonec jsem nestihl dokončit nutné podmínky pro možnost úspěšné obhajoby. Samotná aplikace také není dokončena a výsledná vyexportovaná Eclipse 4 aplikace, obsažena na dodaném nosiči, má na některých systémech hodně podivné chování, což už jsem taky nestihl vyřešit. Nedokončenou kapitolu o tvorbě vlastního nástroje jsem odstranil.

Literatura

[1]

[2] Aspect Oriented Programming with Spring [online]. http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html, [cit. 2015-05-09].

[3] Java 7 API: java.lang.OutOfMemoryError [online]. http://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html, [cit. 2015-05-09].

[4] OpenJDK 7 HotSpot sources: interpreter/bytecodeIntepreter.cpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/interpreter/bytecodeInterpreter.cpp#L2308, [cit. 2015-05-09].

[5] OpenJDK 7 HotSpot sources: interpreter/bytecodeInterpreter.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/interpreter/bytecodeInterpreter.hpp#L120, [cit. 2015-05-09].

[6] OpenJDK 7 HotSpot sources: interpreter/bytecodes.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/interpreter/bytecodes.hpp#L247, [cit. 2015-05-09].

[7] OpenJDK 7 HotSpot sources: interpreter/templateInterpreter.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/interpreter/templateInterpreter.cpp#L1181, [cit. 2015-05-09].

[8] OpenJDK 7 HotSpot sources: memory/universe.cpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/memory/universe.cpp#L1040, [cit. 2015-05-09].

[9] OpenJDK 7 HotSpot sources: oops/constantPoolOop.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/oops/constantPoolOop.hpp#L88, [cit. 2015-05-09].

[10] OpenJDK 7 HotSpot sources: oops/cpCacheOop.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/oops/cpCacheOop.hpp#L320, [cit. 2015-05-09].

[11] OpenJDK 7 HotSpot sources: oops/markOop.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/oops/markOop.hpp#L37, [cit. 2015-05-09].

[12] OpenJDK 7 HotSpot sources: runtime/frame.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/runtime/frame.hpp#L471, [cit. 2015-05-09].

[13] OpenJDK 7 HotSpot sources: runtime/thread.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/runtime/thread.hpp#L245, [cit. 2015-05-09].

[14] OpenJDK 7 HotSpot sources: runtime/thread.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/runtime/thread.hpp#L738, [cit. 2015-05-09].

[15] OpenJDK 7 HotSpot sources: x86/templateTable_x86_32.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/x86/templateTable_x86_32.cpp#L2273, [cit. 2015-05-09].

[16] Chapter 6. The Java Virtual Machine Instruction Set [online]. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.invokedynamic, [cit. 2015-05-10].

[17] HotSpot Glossary of Terms [online]. http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html, [cit. 2015-05-10].

[18] Java 7 API: java.lang.invoke.CallSite [online]. http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/CallSite.html, [cit. 2015-05-10].

[19] Java 7 API: java.lang.invoke.MethodHandle [online]. http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandle.html, [cit. 2015-05-10].

[20] Java Objects Memory Structure [online]. http://www.codeinstructions.com/2008/12/java-objects-memory-structure.html, [cit. 2015-05-10].

[21] OpenJDK 7 HotSpot sources: gc_interface/collectedHeap.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/gc_interface/collectedHeap.hpp#L35, [cit. 2015-05-10].

[22] OpenJDK 7 HotSpot sources: oops [online]. https://github.com/hrib/openjdk7-hotspot/tree/master/oops, [cit. 2015-05-10].

[23] OpenJDK 7 HotSpot sources: oops/klass.cpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/oops/klass.cpp#L182, [cit. 2015-05-10].

[24] OpenJDK 7 HotSpot sources: oops/oop.hpp [online]. https://github.com/hrib/openjdk7-hotspot/blob/master/oops/oop.hpp#L64, [cit. 2015-05-10].

[25] Intel 64 and IA-32 Architectures Software Developers Manual: Alignment of Words, Doublewords, Quadwords, and Double Quadwords - 4-2 Vol.1 [online]. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, [cit. 2015-05-11].

[26] Bytecode Outline plugin for Eclipse [online]. http://andrei.gmxhome.de/bytecode/index.html, [cit. 2015-05-12].

[27] Bytecode Visualizer [online]. http://www.drgarbage.com/bytecode-visualizer/, [cit. 2015-05-12].

[28] javap - The Java Class File Disassembler [online]. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html, [cit. 2015-05-12].

[29] JBE - Java Bytecode Editor [online]. http://set.ee/jbe/, [cit. 2015-05-15].

[30] Krakatau Bytecode Tools [online]. https://github.com/Storyyeller/Krakatau, [cit. 2015-05-15].

[31] Classes and metaclasses [online]. http://pharo.gforge.inria.fr/PBE1/PBE1ch14.html, March 2014 [cit. 2015-05-10].

[32] Austin, R.: What are Thread Local Allocation Buffers ? [online]. http://robsjava.blogspot.cz/2013/03/what-are-thread-local-allocation-buffers.html, 2013-17-03 [cit. 2015-05-09].

[33] Dinn, A.: OpenJDK Architecture, slide 21 [online]. http://www.dcs.gla.ac.uk/~jsinger/pdfs/sicsa_openjdk/OpenJDKArchitecture.pdf, March 2014 [cit. 2015-05-09].

[34] Hohensee, P.: The Hotspot Java Virtual Machine [online]. http://www.cs.princeton.edu/picasso/mats/HotspotOverview.pdf, [cit. 2015-05-10].

[35] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: The Java Virtual Machine Specification [online]. https://docs.oracle.com/javase/specs/jvms/se7/html/, 2013-02-28 [cit. 2015-05-09].

[36] Masumitsu, J.: Presenting the Permanent Generation [online]. https://blogs.oracle.com/jonthecollector/entry/presenting_the_permanent_generation, 2006-11-28 [cit. 2015-05-10].

[37] Nicholas, E.: Understanding Weak References [online]. https://weblogs.java.net/blog/2006/05/04/understanding-weak-references, May 4, 2006 [cit. 2015-05-10].

[38] Sun Microsystems, Inc.: Memory Management in the Java HotSpot Virtual Machine [online]. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf, 2006 [cit. 2015-05-09].

[39] Wikipedia: Breakpoint [online]. http://cs.wikipedia.org/wiki/Breakpoint, [cit. 2015-05-09].

[40] Wikipedia: C++ [online]. http://cs.wikipedia.org/wiki/C%2B%2B, [cit. 2015-05-09].

[41] Wikipedia: C (programovací jazyk) [online]. http://cs.wikipedia.org/wiki/C_(programovac%C3%AD_jazyk), [cit. 2015-05-09].

[42] Wikipedia: Debugger [online]. http://cs.wikipedia.org/wiki/Debugger, [cit. 2015-05-09].

[43] Wikipedia: Endianita [online]. http://cs.wikipedia.org/wiki/Endianita, [cit. 2015-05-09].

[44] Wikipedia: HotSpot [online]. http://en.wikipedia.org/wiki/HotSpot), [cit. 2015-05-09].

[45] Wikipedia: IEEE 754 [online]. http://cs.wikipedia.org/wiki/IEEE_754, [cit. 2015-05-09].

[46] Wikipedia: James Gosling [online]. http://cs.wikipedia.org/wiki/James_Gosling, [cit. 2015-05-09].

[47] Wikipedia: Java bytecode [online]. http://en.wikipedia.org/wiki/Java_bytecode, [cit. 2015-05-09].

[48] Wikipedia: Java (programovací jazyk) [online]. http://cs.wikipedia.org/wiki/Java_(programovac%C3%AD_jazyk), [cit. 2015-05-09].

[49] Wikipedia: Java version history [online]. http://en.wikipedia.org/wiki/Java_version_history, [cit. 2015-05-09].

[50] Wikipedia: Oak (programming language) [online]. http://en.wikipedia.org/wiki/Oak_%28programming_language%29, [cit. 2015-05-09].

[51] Wikipedia: PHP [online]. http://cs.wikipedia.org/wiki/PHP, [cit. 2015-05-09].

[52] Wikipedia: Scala (programovací jazyk) [online]. http://cs.wikipedia.org/wiki/Scala_(programovac%C3%AD_jazyk), [cit. 2015-05-09].

[53] Wikipedia: Sun Microsystems [online]. http://cs.wikipedia.org/wiki/Sun_Microsystems, [cit. 2015-05-09].

[54] Wikipedia: UTF-16 [online]. http://cs.wikipedia.org/wiki/UTF-16, [cit. 2015-05-09].

[55] Wikipedia: Write once, run anywhere [online]. http://en.wikipedia.org/wiki/Write_once,_run_anywhere, [cit. 2015-05-09].

[56] WWW stránky: Golo - a lightweight dynamic language for the JVM [online]. http://golo-lang.org/, [cit. 2015-05-10].

[57] WWW stránky: http://bytecodeviewer.com/ [online]. http://bytecodeviewer.com/, [cit. 2015-05-15].