Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Nástroj pro analýzu Java memory heapu

Plzeň 2019 Martin Mach

PROHLÁŠENÍ

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 23. dubna 2019 . . . . . .

Martin Mach

ABSTRAKT

Abstrakt CZ

KLÍČOVÁ SLOVA

Klíčová slova CZ

ABSTRACT

Abstract EN

KEYWORDS

Keywords EN

MACH, Martin *Nástroj pro analýzu Java memory heapu*: diplomová práce. Plzeň: Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky, 2019. 44 s. Vedoucí práce byl Ing. Richard Lipka, Ph.D.

OBSAH

1 ÚVOD

Při vývoji programů se často můžeme setkat s neuspokojujícími parametry jejich běhu – ať už z hlediska času běhu nebo využití systémových zdrojů. Přes to, že cena hardware klesá a mzdy programátorů rostou, je často nutné přistoupit k optimalizaci existujícího kódu, namísto prostého přidání systémových prostředků [1][2]. To sice v některých případech pomoci může, nicméně problém neřeší – pouze jej oddaluje. Pokud chceme podobným problémům v budoucnu předejít a vyhnout se dalším investicím do výkonu, musíme problémový program upravit – optimalizovat.

Optimalizace může probíhat na několika úrovních. První z možností může být volba efektivnějšího algoritmu – můžeme zkontrolovat, zda jsme pro řešení problému zvolili vhodný algoritmus a zda neexistuje způsob, jakým bychom mohli dotyčné části řešit efektivněji. Tím je možné, především u vyššího počtu zpracovávaných prvků, dosáhnout výrazně rychlejšího běhu programu samotného. Dále je možné rychlost běhu ovlivnit zefektivněním (či prostou redukcí) přístupů k zařízením (IO), ať už se jedná o práci s diskem nebo síťovou činnost. Dále můžeme optimalizovat na úrovni využití systémových prostředků, typicky paměti (RAM). Právě optimalizaci paměťových nároků se v této práci budu věnovat.

I na snižování množství programem užité paměti lze pohlížet z několika různých úhlů. Navrhovaná řešení se mohou výrazně lišit dosaženými úsporami zdrojů, náročností či finanční nákladností. Jako první můžeme zvážit používané technologie. Změna programovacího jazyku je často natolik drahá, že kompletní přepsání dotyčného software ve většině případů nedává, i přes dosaženou optimalizaci, ekononomický smysl. V rámci námi používaného jazyku tak můžeme, kromě našeho kódu samotného, analyzovat používané frameworky a knihovny. V praxi se můžeme setkat s tím, že závislost na knihovně je přidána pouze z důvodu využití jedné či několika jejích funkcionalit. To je vhodné během vývoje z důvodu rychlosti; později je možné některé z těchto knihoven a jimi poskytované funkce nahradit implementací vlastní a teoreticky tak ušetřit jednotky, desítky či stovky megabytů paměti. Tím ovšem neoptimalizujeme data programu, ale velikost programu samotného.

Na řadu tak přichází právě programová data – konkrétně datové struktury a typy, které v aplikaci jako její autoři využíváme. A opět je možné k tomuto problému přistoupit z různých úhlů, lišících se pak především hloubkou a důsledností analýzy. Můžeme řešit, zda námi používané typy odpovídají povaze dat – například, zda rozsah celočíselného typu odpovídá maximální hodnotě dané veličiny.

V praxi mohou být následky nedůsledné optimalizace ještě vážnější. Systémy, které mají obsloužit vysoký počet požadavků za sekundu, je často nutné škálovat – vytvářet nové, nezávislé jednotky těcho systémů. Ty se poté mohou střídat o příchozí požadavky. Každá neoptimalizace je tedy znásobena počtem těchto instancí. To může naprosto zbytečně zvyšovat náklady na provoz; a to ať už v případě vlastního serveru či cloudových

služeb (tzv. serverless).

Cílem této práce je zanalyzovat, k jakým nedostatkům dochází (a zda vůbec) z pohledu neefektivního využití paměti při vývoji Java aplikací. Dále pak vytvořit nástroj, který na tyto nedostatky dokáže poukázat a uživateli napovědět, jakým způsobem by mohl použité prostředky svého programu redukovat a jeho běh tedy optimalizovat. Korektní fungování vytvořeného nástroje bude ověřeno na testovací aplikaci z pohledu správnosti a úplnosti výsledků. Následně pak také prověřeno na větších, rozšířených a běžně používaných Java aplikacích, za účelem posouzení efektivity použití nástroje v praxi, případně také v komerční sféře.

2 PROBLÉM SPRÁVY PAMĚTI

Při vytváření programu má jeho autor na výběr ze dvou způsobů správy paměti – spravované automaticky (typicky mechanismem typu Garbage Collector (GC) apod.) či manuálně, případně kombinací těchto přístupů. Ne každý jazyk nabízí oba – typicky je k dispozici pouze jeden z přístupů, často je správa GC vynucena. Vzhledem k tomu, že majorita nejpopulárnějších programovacích jazyků za poslední roky se řadí mezi vysokoúrovňové, na toto vynucení GC narazíme u většiny z nich, včetně Javy [4][5]. Výjimkou jsou populární nízkoúrovňovější jazyky typu C a C++.

2.1 Garbage collector

GC je nástroj, starající se o správu paměti programu – její přidělování, kontrolu a následné uvolnění, ať už pokud je jí málo a je zapotřebí jinde, v pravidelných intervalech nebo při jiných událostech. GC je obecný termín, tj. neodkazuje na žádnou konkrétní implementaci a způsob chování. Často je v rámci jednoho jazyka (respektive běhového prostředí) zároveň implementováno hned několik algoritmů GC a dle okolností je vybrán ten nejefektivnější a v danou chvíli nejvhodnější z nich. Některé algoritmy tak mohou běžet velmi rychle bez minimálního zásahu do běhu programu, zatímco jiné vyžadují pro svůj běh o něco delší čas. Často tak je nutné všechen běh kódu pozastavit; v takových případech toto spuštění GC nazýváme *stop-the-world* („zastavení světa“, běhu). Pro program je toto zastavení transparentní.

Obecně GC funguje tak, že si udržuje seznam referencí na jednotlivé objekty, respektive jejich počet. Pokud je některý z objektů dále nereferencovaný, při dalším běhu GC bude jím zabíraná paměť uvolněna. Nereferencovaným objektem rozumíme, že je nedosažitelný – nikdo na něj neukazuje. K takovým případům samozřejmě může docházet i v případě jazyků, které fungují bez GC, např. `C`. Pokud daný jazyk nezná jiný způsob, jak se k dané paměti opětovně dostat, dochází k tzv. *memory leakům*, tedy únikům paměti. V případě například výše zmíněného `C` nelze definitivně rozhodnout o nedostupnosti paměti – díky ukazatelové aritmetice je možné paměť zpětně dopočítat i v případě, že v jednom časovém okamžiku na něj žádný ukazatel v paměti programu neukazuje. Java koncept ukazatelové aritmetiky nezná, po odstranění poslední reference si tedy můžeme být jisti, že už znovu referencovat nikdy nepůjde. V souvislosti s *memory leaky* je nutné zdůraznit, že v tomto případě mám na mysli odstranění poslední reference v rámci paměti uživatelského programu. Java Virtual Machine (JVM), který paměť spravuje, adresu objektu stále zná a je tak schopný jej v rámci běhu GC odstranit.

3 STRUKTURA PAMĚTI PROGRAMU

Abychom mohli hledat v paměti a analyzovat nedostatky v rámci jejího využití, musíme nejprve porozumět její struktuře, různým typům oblastí a objektům v nich uložených. Přestože se mnoho následujících konceptů a pravidel vztahuje na ostatní jazyky (a v některých případech i na paměť spravovanou samotným operačním systémem), budu se primárně zaměřovat Javu. Specifikace JVM neobsahuje žádnou konkrétní podobu paměti či její rozložení, stejně tak jako nespecifikuje žádný konkrétní algoritmus GC a žádné optimalizace, které se mají nad běžícím kódem vykonávat. Tyto implementační detaily jsou ponechány na možnostech a tvůrčích schopnostech vývojářů, kteří standard implementují. Prostým požadavkem tak je korektní načtení dat ze souboru `class` a validní vykonání příslušných instrukcí.

JVM, stejně jako většina ostatních běhových prostředí a jazyků, rozeznává dva druhy datových typů – primitivní a referenční. Proměnné primitivního typu tak obsahují přímo danou hodnotu, zatímco proměnné referenčních typů referencují (tj. ukazují na) jinde umístěnou paměť, v níž se nachází objekt.

Primitivních datových typů je v Javě několik druhů, konkrétně:

- numerické
  - celočíselné
    - `byte`
    - `short`
    - `int`
    - `long`
    - `char`
  - s plovoucí desetinnou čárkou
    - `float`
    - `double`
- `boolean`
- `returnAddress`
  - Speciální datový typ, sloužící jako pointer na `JVM` instrukce, konkrétně `jsr`, `ret` a `jsr_w`. V běžném programu jej nicméně nemůžeme jako vývojáři použít ani modifikovat; `JVM` jej však zná a využívá.

Referenční datové typy jsou potom následující:

- `class`
- `array`
- `interface`

Typ `array`, tedy pole, samozřejmě obsahuje seznam prvků určitého typu – ten nazýváme typem komponenty. Proměnná referenčního typu je ukazatelem do paměti – místa, kde je obsah objektu umístěn. Prázdný ukazatel, který nereferencuje žádné místo v paměti,

nazýváme `null`. Tato hodnota je výchozí pro všechny proměnné referenčního typu, dokud jim není přiřazena hodnota [16].

V násedujících odstavcích popisuji různé části paměti `JVM` – viz obrázek 1.

3.1 JVM stack

V případě, že v Javě mluvíme o stacku (zásobníku), typicky máme na mysli JVM stack. Kromě něj totiž ještě existuje nativní stack, který je vytvořený operačním systémem pro potřeby JVM samotného. Kromě toho je využívaný i pro některé nízkoúrovňovější činnosti. JVM stack je spravovaný běhovým prostředím samotným a slouží pro potřeby aplikací, které v rámci prostředí běží.

Pro každé vlákno aplikace je vytvořený stack. Do něj jsou ukládány lokální proměnné, hodnoty parametrů metod a návratové hodmoty. Rovněž se stará o volání metod, respektive o návrat do volací metody po zavolání metody jiné. Stejně jako v jiných jazycích (třeba C) je paměť lokálních proměnných automaticky uvolněna při odebírání hodnot ze stacku. Takto alokovanou paměť tak není nutné spravovat GC (či v případě C ručně uvolňovat).

Právě velikost JVM stacku je často omezující faktor při vývoji, na který v některých případech (např. nesprávně zastavovaná rekurze) narážíme. Jeho velikost můžeme nastavit jako parametr při spuštění programu – konkrétně `Xss` (respektive `-XX:ThreadStackSize`). Jeho výchozí velikost je závislá na velikosti dostupné (virtuální) paměti. V souvislosti s nedostatkem paměti a velikostí stacku se můžeme setkat se dvěma typy výjimek. `StackOverflowError` je vyhozena v případě, že během výpočtu narazíme na horní hranici velikosti stacku. `JVM` se případně může pokusit velikost dynamicky zvýšit. Pokud však během této operace narazí na velikost výše zmíněné paměti (což je už limitace nastavená operačním systémem a případně i hardwarovou konfigurací), je vyhozena výjimka jiná – `OutOfMemoryError`. Tato výjimka je rovněž produkována v případě, že ručně nastavíme velikost stacku takovou, že při jeho vytváření `JVM` narazí na limity paměti rovnou.

3.1.1 Rámce

Rámce (*frame*) přísluší právě jednomu stacku. Data v něm jako takovém jsou totiž neměnná; jsou pouze přidávány/odebírány odkazy na rámce. V každém okamžiku je pro jedno vlákno aktivní právě jeden rámec.

Lokální proměnné

Každý rámec obsahuje pole s lokálními proměnnými. Vzhledem k tomu, že je rámec vytvořen a vložen do JVM stacku při volání metody a znovu odebrán a zničen při návratu

z ní, právě toto pole zajišťuje uvolnění paměti alokovaných proměnných. Kromě proměnných primitivního typu jsou v tomto poli uloženy rovněž reference na objekty v heapu (viz dále), jejich uvolnění sníží počet referencí na daný objekt a v případě dosažení nulové hodnoty je tento objekt připraven na uvolnění pomocí GC.

Stack operandů

Tento stack slouží pro mezivýpočty při provádění operací, převážně matematických.

3.2 Program counter registr

Program counter (PC) je registr s adresou ukazující na operaci, která se má provést. Vzhledem k tomu, že Java umožňuje vícevláknový běh, má samozřejmě každé vlákno svůj PC.

3.3 Stack nativních metod

Pokud má daná implementace JVM podporovat i tzv. `native` metody, měla by obsahovat i stack nativních metod. Zdůrazňuji, že *měla* – specifikace zde benevolentně ponechává rozhodnutí na konkrétním řešení a tento stack nutně nevynucuje. Díky konceptu nativních metod je možné přímo z Javy volat kód napsaný např. v C.

Stejně jako JVM stack, i stack nativních metod v některých případech produkuje výjimky. Toto chování je totožné pro oba stacky; stejně tak typy těchto chybových stavů jsou stejné.

3.4 Heap

Heap, tj. *halda*, je část paměti, která je pro všechna vlákna programu společná a v Javě tomu není jinak. Díky tomu, že jsou všechny objekty alokovány právě v heapu, k nim můžeme přistupovat napříč metodami, objekty i vlákny. Díky tomu, že platnost v ní umístěných objektů není omezena žádným blokem platnosti (snad jen s výjimkou běhu programu samotného), nedojde k uvolnění jimi zabírané paměti automaticky při opuštění tohoto bloku tak, jak je tomu v případě stacku. Právě kvůli tomu nad heapem operuje GC, který nepotřebné objekty vyhledává a jejich paměť uvolňuje.

Kromě alokovaných objektů jsou rovněž v heapu uchovávány instanční proměnné. Právě z toho důvodu je nutné si dávat pozor na souběh při běhu ve více vláknech – všechna při přístupu k instanční proměnní manipulují se stejným objektem.

I haheaplda má samozřejmě omezenou velikost. Ta se dá nastavit dvěma přepínači:

- `Xms` – Výchozí velikost heap, s kterou Java nastartuje.

- `Xmx` – Maximální velikost.

V případě 32 bitového systému jsou horní hranicí pro maximální velikost heapu 4 Gb, stejně jako v případě velikosti RAM (bez použití různých triků a rozšíření, jako třeba *PAE*, apod.). Jak vyplývá z existence výše zmíněných přepínačů, JVM dokáže s velikostí heapu dynamicky manipulovat. Program, respektive prostředí pro něj, tak spustí s výchozí hodnotou velikosti a v případě potřeby ji rozšiřuje (nebo naopak zmenšuje) až do maxima. Pokud si běh programu žádá více paměti v heapu, než může JVM alokovat (tj. než má od systému k dispozici), vyhod, stejně jako v případě stacku, výjimku `OutOfMemoryError`.

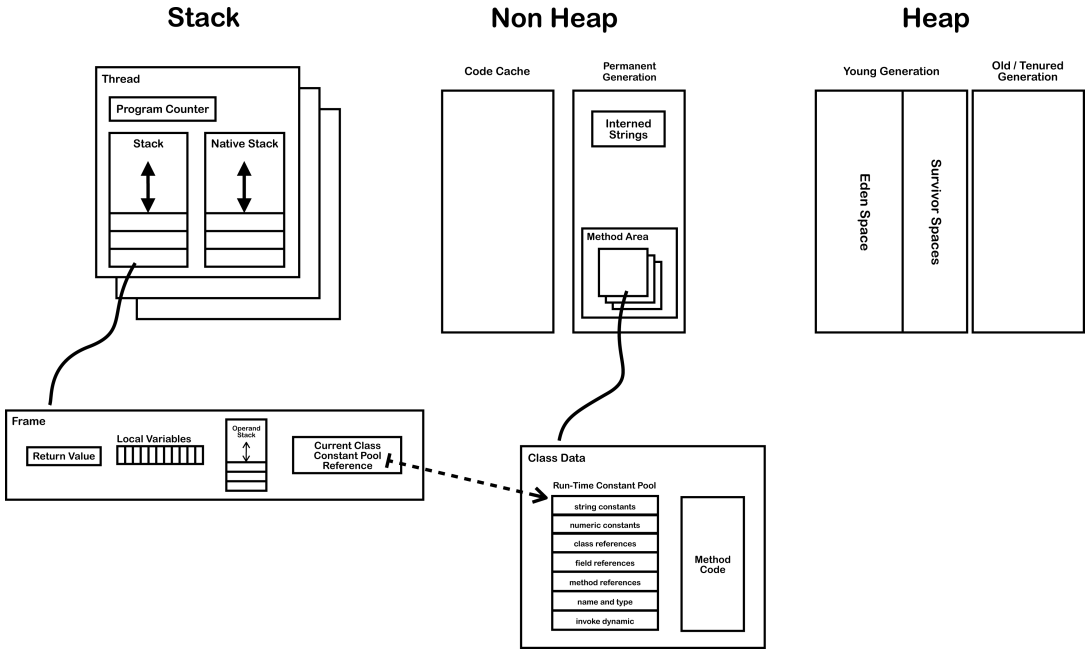
Heap je rozdělen dále do několika prostorů, jak je popsáno dále v kapitole 7.

3.5 Non-heap oblast

Kromě oblasti metod bsahuje pomocná data, která jsou potřeba pro interní zpracování a potřeby.

3.5.1 Oblast metod

Oblast, která uchovává data jednotlivých metod. Pro každou načtenou třídu uchovává strukturu se seznamem metod, statickými proměnnými a kódem metod a konstruktorů. Na jednu ze struktur třídy směřuje ukazatel z aktuálně zpracovávaného rámce v stacku, pomocí kterého může aktuálně vykonávaný kód přistupovat ke statickým členům třídy.



Obr. 1: Struktura Java paměti [20].

3.6 Runtime constant pool

Runtime constant pool (volně přeloženo jako *fond konstant pro běh*) je tabulka symbolů, která uchovává různé konstanty a informace nutné pro běh. Obsahuje následující položky.

Informace o třídách Obsahuje některé informace o samotných třídách a rozhraních.

- Pro třídy nebo rozhraní, která nejsou polem, obsahuje přímo jejich název
- Pro pole promitivního typu o `n` dimenzích začíná `n`-krát znakem `[`, pak následuje jméno komponenty pole. Pokud je toto navíc neprimitivního typu, jako prefix je přidán symbol `L` a postfix `;`.

Informace o třídních promenných Obsahuje název a typ.

Informace o metodách Obsahuje název a informace z hlavičky metody.

Informace o rozhraních Obsahuje názvy rozhraní a případně i jejich metody.

Různé typy parametrů, návratové typy atd.

3.6.1 Řetězce

Kromě toho jsou zde uloženy i řetězce (`String` (i když to neplatí univerzálně, některé řetězce budou uloženy přímo v heapu). Java při vytváření řetězců může použít existující literál uložený v paměti tak, aby tento řetězec zbytečně nezabíral v paměti místo dvakrát či vícekrát. Toto chování lze případně vynutit voláním metody `intern` nad daným řetězcem – viz ukázka 3.1.

Ukázka 3.1: Příklad uchovávání řetězců a metody `intern`

```
String s1 = "hello";
String s2 = "hello";
String s3 = new String("hello");

System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
System.out.println(s1 == s3.intern()); // true
```

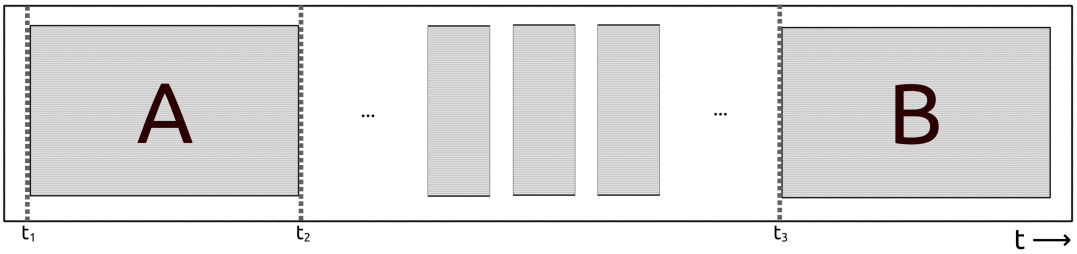
4 ANALÝZA PAMĚTI

Paměť je možné analyzovat za běhu programu nebo z jejího snímku – reprezentace paměti v určitém okamžiku běhu programu.

4.1 Analýza za běhu programu

Během běhu programu lze analyzovat aktuální obsah paměti – GC koneckonců nedělá nic jiného. Problémem tohoto přístupu je ovšem dopad na výkon. I samotný GC, ať už umožňující současný běh kódu nebo *stop-the-world*, má výrazný dopad na výkon oproti jazykům či běhovým prostředím bez něj [13]. Navíc, běh GC, stejně tak jako režie běhu samotného prostředí, jsou v drtivé většině případů v maximální možné míře optimalizovány, aby byl dopad na výkon samotného programu co nejmenší a jeho běh co nejrychlejší. Mohou využívat celou řadu nízkoúrovňových optimalizací, kterých – v některých jazycích, včetně Javy – dosáhnout prakticky nemůžeme. Z toho můžeme usoudit, že dopad analýzi paměti programu za jeho běhu by byl přinejmenším takový, jako dopad běhu GC; pravděpodobně však výrazně vyšší. Stejně tak musíme uvažovat, že běh naší analýzy – ať už pouze za účelem sběru dat pro pozdější zpracování – bude výrazně složitější (či výkonově náročnější), než běh GC.

Dalším důvodem pro vyhnutí se analýze za běhu programu je povaha řešeného problému a tedy implementovaného algoritmu. Protože potřebujeme provádět hloubkovou analýzu (všech či vybraných) objektů programu, jejich neustálá změna by tuto analýzu za běhu programu učinila nemožnou. Respektive bychom se nemohli vyhnout eventuálním falešně negativním či positivním hlášením – to si jednoduše můžeme představit. Jestliže s každým cyklem analýzy zpracujeme jeden objekt a během toho dojde ke změně dat některého z dalších objektů, pak by algoritmus zahlásil při zpracování dalších objektů falečně pozitivní nález. V žádném časovém okamžiku neexistovaly dva objekty se stejnými daty, algoritmus by je tak přesto označil.



Obr. 2: Problém analýzy paměti za běhu programu.

5 OPTIMALIZACE UŽITÍ PAMĚTI

Jak již bylo zmíněno v úvodu, využití paměti lze optimalizovat na několika úrovních. Vzhledem k tématu této práce se při uvažování optimalizací omezíme pouze na běžící Java aplkaci. Eliminaci případných zbytečných knihoven, frameworků a nástrojů se věnovat rovněž nemusíme. Analyzovat procentuální využití nabízených funkcí knihovny by mohlo být zajímavé, nicméně tento typ zbytečného užívání paměti nelze přímo označit za memory waste.

Můžeme tedy omezit samotné využití objektů – zvážit, zda je potřebujeme, případně zda neexistuje vhodnější způsob či struktura pro jejich uchovávání. Jejich alokaci v paměti bychom rovněž meli omezit na nejkratší možnou dobu, po kterou si budeme jisti jejich využitím. Nicméně i zde je vhodné najít vhodný poměr mezi optimalizací a zbytečným úklidem objektu, který za několik okamžiků znovu budeme vytvářet.

Nejpřímějším způsobem optimalizace užití paměti se může jevit odstranění objektů, které už nejsou zapotřebí a nikdo je tedy nevlastní. Takové objekty nicméně nemusíme v naší optimalizaci uvažovat. O jejich uvolnění z paměti se postará GC. Této vlastnosti jazyku tedy můžeme využít a nepotřebné objekty ručně odstraňovat – nastavit je jako `null`. K jejich uvolnění dojde i při opuštění aktuálního prostoru – oboru platnosti lokální proměnné. K takovému uvolnění nicméně nedochází přímo zapomocí analýzy GC, ale díky uložení lokálních proměnných v stacku, z kterého jsou při opuštění bloku odstraněny.

6 JAVA VIRTUAL MACHINE

Program napsaný v Javě běží typicky v některé z implementací JVM. JVM je program, který slouží jako běhové prostředí pro uživatelský kód – vykonává jeho instrukce a slouží tak jako prostředník mezi ním a operačním systémem (respektive jako interpret jeho kódu, který následně překládá do jiného jazyka, typicky strojového kódu dané architektury či platformy). JVM je možné si představit jako virtuální počítač – má vlastní instrukční sadu a na základě ní provádí operace nad pamětí. Díky tomu je možné jej, v případě potřeby, implementovat i jako CPU – taková hardwarová implementace se nazývá *Java procesor*.

JVM nemá ponětí o existenci Javy jako jazyku. Vidí pouze výsledek kompilace do bytecodu (viz dále), což je posloupnost operací z výše zmíněné instrukční sady. Jako analogii je možné zmínit instrukční sadu CPU a strojový kód – CPU také netuší, jaký vyšší programovací jazyk je původcem strojového kódu, a při podobné implementaci kompilátoru dvou jazyků by to ani neměl šanci zjistit. Stejně tak se chová JVM a bytecode – ostatně existují i další jazyky na platformě JVM, za všechny můžu jmenovat třeba populární Kotlin [17].

Některé implementace JVM umožňují přímý překlad do strojového kódu bez potřeby interpretace, např. GraalVM [3]. V takovém případě hovoříme o tzv. Ahead-Of-Time (AOT) přístupu, místo Just-In-Time (JIT) postupu implementovaného v moderních verzích častěji používaných JVM, např. HotSpot od společnosti Oracle [TODO zdroj, že HS používá JIT].

JVM se stará o načtení kódu ze souboru `.class`, dále o jeho verifikaci, spuštění a zároveň poskytuje tomuto kódu prostředí, v rámci kterého může běžet.

6.1 Java Bytecode

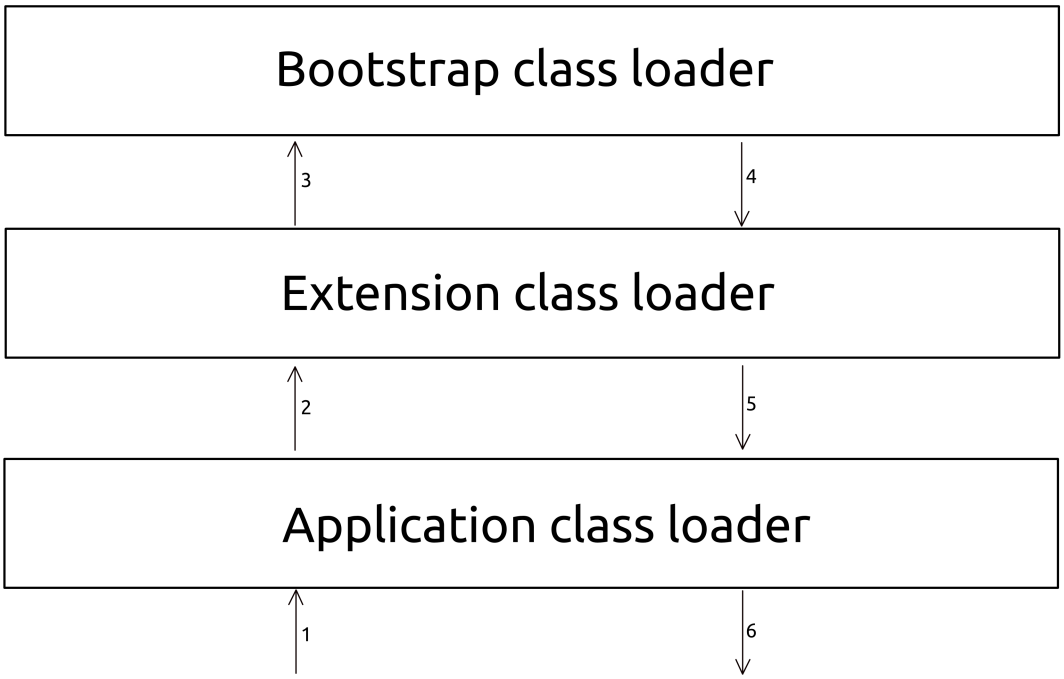
Java využívá dvoufázový překlad, tj. samotný zdrojový kód vytvořený programátorem je nejprve přeložen do *bytecodu* (či česky bytekód). Bytecode je (či by měl být) platformě nezávislý soubor instrukcí, který následně JVM vykoná v prostředí architektury a platformy, na které je spuštěn. To znamená, že zdrojový kód v jazyce Java (či kompatibilních jazycíc využívajících stejné prostředí, např. Kotlin), uložený typicky v souboru `.java`, je přeložen do bytecodu – typicky s typu `.class`. Takové soubory je následně, obecně vzato, možné přenést na jinou platformu či architekturu a jestliže se zde nachází kompatibilní JVM, je možné jej bez úpravy na daném systému vykonat a tedy program spustit. Bytecode se může přenášet formou klasických jednotlivých souborů `.class` či v zabalené formě, tj. archiv typu `.jar` (což je de facto pouze zip archiv s předem danou strukturou a volitelně přidanými informacemi – manifestem, metadaty, podpůrnými soubory (*resources*)).

6.2 Class loader

Jak už jsem zmínil, JVM se stará o načítání dat ze souboru `.class`. Konkrétně se o toto načítání stará objekt nazvaný *class loader*. Ten, typicky, načítá třídy ze souborového systému, konkrétně z cesty (nebo z více cest) definované v proměnné `CLASSPATH`. Nicméně, díky tomu, že koncept class loaderů je poměrně abstraktní, je možné načítat třídy například přes síť, z paměti nebo je dynamicky vytvářet za běhu programu dle potřeby. Každá načtená třída (konkrétně objek typu `Class`) obsahuje referenci na objekt class loaderu, který ji zavedl do programu – konkrétně jej lze obdržet pomocí metody `getClassLoader()` (v případě primitivních typů vrací `null`). `ClassLoader` je definovaný jako abstraktní třída, lze tedy definovat vlastní loadery dle potřeby.

Pokud `JVM` narazí na třídu, podívá se do setu načtených tříd. Jestliže se tam nachází, vezme ji (instanci třídy `Class`) a použije. Pokud ne, požádá o její načtení. Systém class loaderů funguje na principu delegování v následujícím pořadí (typicky, pokud jej neupravíme), jak zobrazuje obrázek 3 [18] [19]. V případě, že třídu žádný z boot loaderů nenalezne, je produkována výjimka `ClassNotFoundException`.

1. Aplikace požádá *Application class loader* o načtení třídy.
2. Ten zavolá *Extension class loader*.
3. *Extension class loader* zavolá *Bootstrap class loader*.
4. *Bootstrap class loader* funguje v rámci výchozích *JDK* a *JRE* knihoven. Pokud třídu nalezne, tak ji vrátí. V opačném případě volá zpět *Extension class loader*.
5. *Extension class loader* se podívá v rámci rozšíření *ext* Javy. V případě neúspechu volá *Application class loader*.
6. *Application class loader* načítá třídy z `CLASSPATH` (a třídy specifikované v Manifestu atd.)



Obr. 3: Postup volání hierarchie class loaderů.

7 SPRÁVA A STRUKTURA JAVA HEAPU

Paměťovému modelu v Javě se věnuje *JSR-133*, nicméně přesně nespecifikuje konkrétní rozdělení paměti a způsob jejího přidělování a uvolňování. Následný popis se tedy věnuje implementaci od společnosti Oracle – HotSpot. Zde je paměť rozdělena na 2 logické celky – *young generation space* a *old generation space*. Tato paměť, tedy heap, je rozdělena pouze v rámci JVM a následně je mapována na skutečnou fyzickou paměť [6].

Young generation space, tedy doslova "prostor mladé generace", je dále rozdělen na *eden space* a prostory *S0* a *S1*. Eden space slouží k vytváření nových instancí objektů, je zde tedy vyhrazena část paměti nově vytvořenému objektu. Pokud v tomto prostoru není volno, proběhne uvolnění paměti (viz dále) přesunutím některých objektů do S0. Každý takový objekt obsahuje informaci o tom, kolik takových uvolnění daný objekt „přežil".

Po určitém počtu takových přežití (či jinak také povýšení) je objekt přenesen do objektů staré generace, konkrétně *Tenured space*.

Toto rozdělení objektů do jednotlivých prostorů se jeví jako zbytečná komplikace, má však řadu výhod. První z nich je rychlost – nejvíce operací uvolnění je prováděno právě nad eden spacem, který je z prostorů nejmenší. Dále jsou tak objekty rozdělovány do skupin s podobnou charakteristikou (podobný věk, podobný počet a styl referencí apod.), na kterými je poté možné spustit rozdílné, pro dané skupiny specifické algoritmy pro jejich uvolnění.

7.1 Garbage collection

Spuštění GC v Javě nelze vynutit ručně. Systému lze *doporučit* jeho exekuci voláním metody `System.gc()`; JVM se tím ale nemusí řídit a toto volání jednoduše ignorovat [7].

I v Javě můžeme narazit na problém úniků paměti, tzv. *memory leaků*. Typicky se tento problém týká nízkoúrovňových jazyků typu C, nebo takových jazyků, kde je správa paměti v kompletní kompetenci autora programu. Často dojde k „zapomenutí" některého ukazatele. Jeho smazáním se paměť stává nedostupnou a protože v daném jazyku není GC, bude uvolněna teprve ukončením programu – operačním systémem samotným. Toto chování je nebezpečné, protože pokud program poběží dlouhou dobu a bude alokovat paměť bez jejího následného uvolnění, dříve nebo později narazí na limit kladený ze strany operačního systému. Rovněž může jeho provozování být nepříjemné pro provozovatele programu, protože i když jeho běh operační systém neukončí, program bude zabírat zbytečně velké množství paměti.

Ve spojení s GC by tedy nemělo k únikům paměti typicky dojít. V Javě k nim může dojít především při nedůsledném používání vlastních zavaděčů tříd – *class loaderů*. Za únik paměti můžeme rovněž považovat neuzavřený popisovač otevřeného souboru, databáze či

jiného zdroje. Pokud k němu ztratíme přístup, např. po vyhození výjimky bez uzavření tohoto popisovače v bloku `catch` či lépe `finally`, ztrácíme tím, spolu s popisovačem, i menší množství paměti. Při častějším výskytu problému ale v tomto případě pravděpodobně narazíme na horní limit popisovačů Javy či operačního systému – ani tento zdroj není neomezený.

V Javě je několik různých implementací GC. Ty se liší rychlostí a efektivitou běhu a každý se může specializovat na jinou činnost [14]. Z toho některé GC dokáží většinu práce odvést simultánně s během aplikace (tzv. konkurentní GC); jiné vyžadují dočasné zastavení aplikace.

7.1.1 CMS GC

Algoritmus CMS GC prozkoumává paměťový prostor heapu pomocí běhu ve více vláknech (tak, aby byl pokud možno co nejrychlejší). Skenuje ho tak, aby nalezl nepoužívané objekty. Ty postupně označí pro uvolnění a nakonec nad nimi provede iteraci a všechny uvolní. Takový typ běhu GC je vhodný zejména pro aplikace, které preferují pokud možno co nejkratší dobu jeho běhu. Zároveň se aplikace dělí s GC o výkon procesoru na všech jeho vláknech – CMS je význačný tím, že se co nejvíce práce pokouší vykonat přímo za běhu aplikace. Její běh zastavuje během jednoho cyklu svého běhu dvakrát – poprvé pro označení dosažitelných objektů, tzv. *initial mark pause* fáze. Druhá pauza aplikace proběhne po souběžném běhu zároveň s aplikací a označí zbylé objekty, které nebyly během konkurentního běhu nalezeny. Tato fáze se nazývá *remark pause* [15].

7.1.2 Seriový GC

Typ běhu GC, který zastaví všechna vlákna a provede nad nimi operaci uvolnění. Vzhledem k typu běhu je vhodný pouze pro menší aplikace nebo jednovláknové stroje. Všechny operace uvolnění běží pouze v jednom vlákně – tato vlastnost jej činí zároveň velikce efektivním, protože není zapotřebí žádná synchronizace ani mezivláknová komunikace. Díky tomu, že zmrazí všechna ostatní vlákna, je vysloveně nevhodný pro aplikace, které se na vícevláknový běh spoléhají – typicky například server, který musí poslouchat na portu a obsluhovat příchozí provoz.

7.1.3 Paralelní GC

Je výchozí volbou pro JVM (pokud jej umožňuje hardware), až do Javy 8 (včetně). Jak vyplývá z názvu, jeho běh je rozdělen do více vláken, narozdíl od seriového GC. Je určen pro aplikace velkého rozsahu, které běží ve více vláknech (a na systémech, které tento běh umožňují). Konkrétní počet vláken, ve kterých GC běží, je zvolen automaticky na základě

dostupných zdrojů – konkrétně paralelních vláken, které je schopen daný procesor obsluhovat zároveň. Kromě toho je tento počet rovněž možné nastavit ručně (pomocí parametru `-XX:ParallelGCThreads`).

Při běhu paralelního GC má tento typ priority – kritéria, která musí splnit. Jejich priorita je v následujícím pořadí (sestupně):

Maximální čas běhu Nastavuje horní časový limit (v milisekundách), jak dlouho má být trvání jedné pauzy. Výchozí hodnota dobu trvání neomezuje. Pokud ručně nastavíme kratší čas, než je pro algoritmus optimální, můžeme výrazně snížit propustnost algoritmu.

Propustnost Poměr času stráveného zpracováním garbage kolekce oproti času stráveného mimo ni (tj. času využívaného aplikací). Např. hodnota 9 nastaví tento poměr na 10% ($\frac{1}{1+9}$). Výchozí hodnota je 1%.

Minimální *footprint* (stopa, využití prostředků) Zajišťuje dodržení velikosti heapu.

Při běhu GC je postupováno shora, tj. na propustnost je brán ohled pouze za podmínky, že je splněna maximální doba běhu. Analogicky je splněna podmínka minimálního footprintu.

7.1.4 G1 GC

G1 znamená *Garbage First* a je výchozí volbou pro algoritmus od Javy 9 (včetně). Je určený pro heapy větší velikosti, které pak rozdělí do několika oblastí – regionů. Ty poté paralelně prohledává; prioritu přikládá regionům, které mají nejvyšší poměr zbytečných objektu (což odhaduje heuristicky). Oprito parelelnímu GC nabízí stabilnější běh algoritmu a kratší pauzy.

7.2 Heap Dump

Heap dump je textová nebo binární reprezentace paměti, kterou je možné uložit na disk a zachycuje aktuální stav aplikace. Při vytváření je činnost aplikace pozastavena. Dump je možné následně analyzovat a dále zpracovávat, je tak možné prozkoumat vnitřní stav aplikace v určitém bodě a např. řešit příčiny neočekávaného chování.

7.2.1 Vytvoření dumpu

Prostředků k vytvoření dumpu je několik. Při správném nastavení (pomocí parametru `HeapDumpOnOutOfMemoryError`) k němu dojde při nedostatku paměti zcela automaticky. Mezi manuální způsoby vytvoření patří primárně nástroj *JMAP*, který je publikován spolu se standardní distribucí Oracle JVM. Při použití tohoto nástroje je nutné naprosto přesně

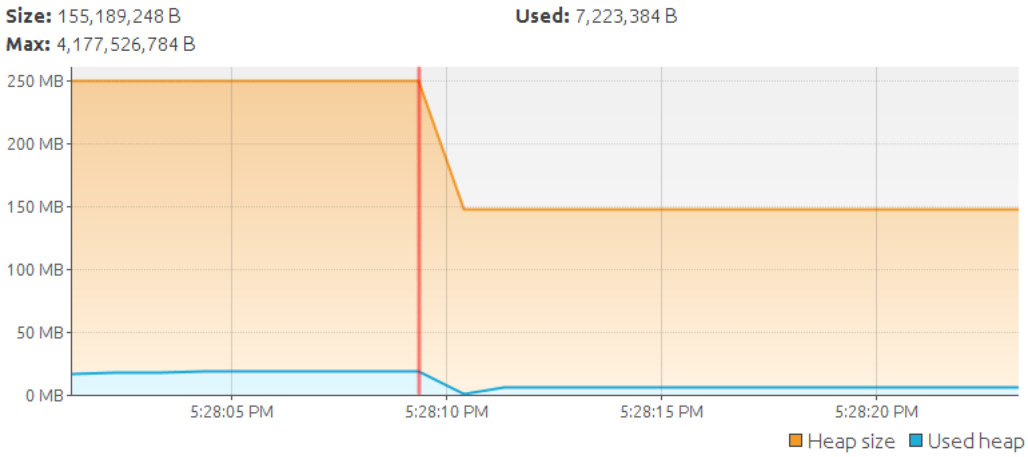
dodržet číslo verze JMAP a JRE, pod kterým cílová aplikace běží – dumpování rozdílných verzí není podporováno, je nutné dodržet rovnost verzí (major, minor i update).

Mezi další způsoby vytvoření dumpu patří různé nástroje, debuggery a profilery typu Eclipse MAT, VisualVM nebo Java Mission Control (viz dále). Výhodou těchto nástrojů je, že dokáží zvládnout vytváření dumpu i napříč verzemi a dokonce implementacemi (z Oracle JDK na OpenJDK apod.)

Další možnost je využít některou z knihovních funkcí a vytvářet tak dump programově. Zde je možné využít např. MBeans. Nízkoúrovňovou možností je poté například použití Unixového nástroje *gcore*, respektive *GDB*, který se postará o vytvoření dumpu paměti procesu (pod daným Process Identifier – Identifikátor procesu (PID)), tzv. *core dump*. Z něj lze memory dump vyextrahovat. Pokročilejší nástroje typu VisualVM umí pracovat i napřímo s core dumpem.

7.2.2 Vztah dumpu vůči paměti procesu

Heap Dump přímo odpovídá části paměťového prostoru procesu, resp. heapu. Je tedy přímým otiskem části fyzické paměti tak, jak je uložena, v určitém časovém okamžiku. Toto je možné si experimentálně ověřit – jak bylo zmíněno výše, z otisku fyzické paměti je možné heap dump získat. Je tedy evidentní, že při jeho vytvoření některým z výše uvedených způsobů nedochází k žádným úpravám a snímek je vytvořen „tak jak je“.



Obr. 4: Využití paměti a provedení garbage kolekce před vytvořením dumpu (červeně).

7.3 Zpracování dumpu

Při zpracování dumpu je nutné zohlednit fakt, že je vyexportovaný kompletní paměťový prostor [TODO zdroj] a nachází se zde tedy i data objektů, které nás nutně nemusí zají-

mat – typicky knihovny nebo objekty Javy. Toto je možné zohlednit a filtrovat na základě jmenného prostoru (namespace), do kterého objekt patří.

Pro práci nad heapem (typicky ve formátu HPROF) je možné využít některou z implementací dotazovacího jazyku OQL (Object Query Language).

7.4 Binární formát dumpu heapu

Binární formát dumpu Java heapu (*hprof*) má specifikovaný formát, který je popsán v tabulkách 1 a 2.

7.4.1 Hlavička

Velikost	Popis
$n + 1$ B	Formát a verze dumpu délky n. Velikost není specifikována, pole bytů je ukončeno hodnotou `null`.
4 B	Velikost identifikátorů. Mohou mít stejnou velikost jako ukazatele hostitelského systému, ale není to nutné.
4 B	Část data (high-order word) z Unix timestampu.
4 B	Část data (low-order word) z Unix timestampu.

Tab. 1: Hlavička `hprof` binárního formátu.

7.4.2 Tělo

Tělo formátu dumpu je složeno ze sekvence (či pole) položek následujícího formátu.

Velikost	Popis
1 B	Tag – typ záznamu – viz tabulka 3.
4 B	Čas – počet milisekund, které uplynule od Unix timestampu v hlavičce.
4 B	Délka – počet bytů n, které následují a slouží jako obsah záznamu.
n B	Obsah záznamu.

Tab. 2: Tělo `hprof` binárního formátu.

Typy záznamů

Tabulka s tělem obsahuje pole záznamů, v nichž je typ záznamu – pole *tag*. Celkově je podporováno 13 následujících typů – z toho většina má ještě, jako obsah, další – více či méně komplexní – strukturu.

Tag	Popis
0x01	Dump řetězce.
0x02	Načtení třídy.
0x03	Odstranění (ve smyslu opaku načtění) třídy.
0x04	Rámec (položka) stacku.
0x05	Stack trace.
0x06	Alokace.
0x07	Souhrn některých hodnot heapu.
0x0A	Počátek vlákna.
0x0B	Konec vlákna
0x0C/0x1C	Heap dump, respektive heap dump segment.
0x2C	Konec heap dumpu.
0x0D	CPU vzorkování.
0x0E	Hodnoty některých nastavení.

Tab. 3: Typy položek `hprof` binárního formátu.

Typy dumpovaných hodnot

Některé z výše zmíněných typů (které jsem dále nerozepisoval) už obsahují přímo data programu (ať už uživatelská nebo pomocná). Proto jsou definovány typy těchto dat, které jsou shrnuty v tabulce 4. Pole jsou definována pomocí řetězcu s jejich názvem tak, jak by se objevily v Javě, tedy např. `int[][]`.

Označení	Datový typ
2	`Object`.
4	`Boolean`.
5	`Char`.
6	`Float`.
7	`Double`.
8	`Byte`.
9	`Short`.
10	`Int`.
11	`Long`.

Tab. 4: Datové typy položek `hprof` binárního formátu.

8 EXISTUJÍCÍ NÁSTROJE PRO ANALÝZU HEAPU

Analýza dumpu – ať už manuální či automatická – je využívána v případech, kdy nám debugování na úrovni kódu už nestačí a potřebujeme se podívat, jakým způsobem se program chová na pozadí. Některé typy problémů navíc pomocí debuggeru odhalíme jen těžko (nebo vůbec) a musíme tak zvolit nízkoúrovnější způsob, který dokáže odhalit problém napříč více částmi aplikace. Rovněž je taková analýza vhodná v okamžiku, kdy dojde k pádu (zejména v důsledku nedostatku paměti), kdy chceme zjistit příčinu pádu „post mortem“. Můžeme tak odhalit problematické místo v programu a podrobná analýza nám tak umožní problém opravit.

TODO přidat licence a další info.

8.1 Eclipse MAT

IBM Eclipse Memory Analyser Tooling je open source nástroj pro analýzu Java paměti. Po spuštění umožňuje otevřít již vygenerovaný dump Java heapu, umí jej ale i vytvořit. V rámci analýzy nabízí 2 konkrétní volby – analýzu memory leaků a memory bloatu – tj. neefektivního využití paměti a zbytečného plýtvání.

MAT se analýzou memory bloatu přibližuje zamýšlenému výsledku této práce, bohužel ale nenabízí kompletní funkcionalitu v této oblasti. Omezuje se pouze na efektivní práci s řetězci (kterou už obsahuje Java, respektive JVM v základu, viz TODO) a dalšími základnímu typy, např. Map. Cílem práce je ale zpracování všech možných objektů, tato funkcionalita by se tak dala případně rozšířit.

Nástroj je založený na platformě Eclipse RCP (Rich Client Platform), respektive OSGi. Díky tomu je poměrně snadno rozšiřitelný, což je ale vyváženo velkým rozsahem aplikace, který vývoj a rozšíření naopak lehce komplikuje. Buildovacím nástrojem je zde Maven.

8.2 VisualVM

Open source profiler pro Java platformu. Patří mezi nejpoužívanější profilery, respektive nástroje pro analýzu výkonu v Javě.

Po spuštění nabízí klasické funkce typické pro profilery, tj. využití paměti, zatížení CPU, počet objektů a vláken a podobné statistiky. Kromě toho obsahuje celou řadu dalších funkcí, jako provedení garbage kolekce (její vyžádání, explicitně vyvolání GC není možné) nebo vytvoření heap dumpu.

Požadovanou funkcionalitu VisualVM v zásadě neposkytuje, umožňuje pouze k nahlédnutí tabulku s informacemi – kolik bylo vytvořeno instancí jaké třídy, respektive jimi

okupovanou paměť. V programu využít podporu pro OQL syntaxi, což je možné využít, nicméně tento přístup nelze považovat za dostačující.

Pro build je využíván nástroj Ant a v současné době je vyžadována Java verze 7 a vyšší.

8.3 Java Mission Control

Nástroj poskytovaný přímo spolu s distribucí Oracle JVM od verze 7 (konkrétně 7 Update 40 – 7u40), což je jeho výhodou. Mezi jeho možnosti patří např. využití paměti jednotlivými součástmi Java paměti a také umožňuje zobrazit jednotlivé instance objektů, nicméně neumožňuje jejich další analýzu.

8.4 JProfiler

Komerční profiler, přesto velice používaný. Standardní licence stojí v době psaní 409 euro, akademická potom 179 euro. Je možné zažádat o licenci pro open source produkty, nicméně ta je podmíněna již vydanou verzí a existující webovou stránkou, což činí jakékoliv použití tohoto profileru v rámci práce nepraktickým. Profiler je používán především v komerční sféře, díky svým možnostem a dle výše uvedeného testu i nejvyšší úspěšností v odhalování bugů.

8.5 JHAT

Nástroj, který je přímou součásti distribuce Oracle JVM od verze 6. Nebyl nikdy oficiálně podporován a od počátku byl označen jako experimentální nástroj, z těchto důvodů byl tedy v Javě 11 naprosto odstraněn [8][9]. V rámci verzí Javy, které jej obsahují, ho lze využít jako Command Line Interface (CLI) aplikaci, která dokáže dump vytvoření pomocí např. *JMAP* otevřít. Následně vytvoří webový server, jehož prostřednictvím poskytuje data získaná ze zpracovaného dumpu. Tato data je možné si poté zobrazit pomocí webového prohlížeče; zajímavým příkladem možného využití je následné rozparsování těchto dat jakožto formátu HTML a jejich další využití. Program je tedy možné využít jako prostředníka pro zpracování [10]. Kromě „prostého“ zobrazení webové stránky umí *JHAT* rovněž poskytovat zpracování pomocí jazyka Object Query Language (OQL).

8.6 OQL

OQL zmíním i jako samostatný způsob zpracování. Jedná se o jazyk, který slouží pro obecnou manipulaci s objekty, resp. s objektovými dokumenty [11][12]. Na první pohled

si nelze nevšimnou jeho podobnosti s dotazovým jazykem SQL. Neomezuje se tedy pouze na zpracování Java heapu (či obecně paměti), ale je standardem, který pro tento účel lze využít. Z toho je možné usoudit, že standard jako takový pro zpracování nestačí – je nutné využít některou z jeho implementací, která takové rozhraní přístupu k Java heapu umožňuje. Jednou z nich je právě výše zmíněný *JHAT*.

V příkladu 8.1 je možné vidět práci s OQL.

Ukázka 8.1: Příklad OQL

```
select s
from java.lang.String s
where s.value.length >= 100
```

9 MOŽNOSTI ANALÝZY

9.1 Rovnost objektů

Rovnost dvou či více objektů se dá definovat a zjišťovat různými způsoby. Je ale nutné si uvědomit, že v nejhorším případě, tj. pokud chceme najít všechny nadbytečné kopie každého objektu, je složitost této operace $O(2^n)$. Bylo by tedy vhodné se zamyslet, zda neexistuje způsob, jak počet porovnání snížit, případně navrhnout jednoduchou heuristiku, která by dokázala rychle ohodnotit, zda má vůbec smysl pokračovat v podrobnějším porovnání. V následujících případech tedy uvažujme objekty A, B, jejich třídy C_A a C_B a proměnné obou instancí $F_A^0..F_A^n$, respektive $F_B^0..F_B^n$. Hodnotu dané instanční proměnné definujeme jako $V(F)$. Pro potřeby algoritmu předpokládejme, že jsou tyto proměnné v určitém, předem očekávatelném pořadí, např. podle abecedy dle jejich názvu (nebo dle Unicode kódu pro speciální znaky).

První nápovědou samozřejmě může být porovnání tříd obou objektů – C_A a C_B. Pokud platí $C_A = C_B$, zřejmě má smysl se porovnáváním dále zabývat. V případě jejich nerovnosti není ale možné další porovnání zavrhnout, porovnat je nutné (či možné) i jejich rodiče. V případě dědičnosti dvou tříd nelze automaticky vyloučit – pokud potomek k rodičovské třídě nepřidává žádná data, pouze funkcionalitu, pak se z pohledu dat v paměti objekty rovnají, i když jsou sémanticky odlišné.

Definujeme-li tedy funkci pro zjištění přímého rodiče $P(C)$, potom rovnost objektů (z pohledu jejich dat) lze vyjádřit jako

$$E_C(C_A, C_B) \Leftrightarrow C_A = C_B \vee E_C(P(C_A), C_B) \vee E_C(C_A, P(C_B)). \tag{9.1}$$

Samozřejmě je nutné definovat i zastavovací podmínku, v případě Javy by tedy jeden z parametrů nesměl být třída typu *Object*. Formálně je tedy možné tuto rovnost definovat jako

$$E_C(C_A, C_B) = P_c(C_A) \cap P_c(C_B) \notin \emptyset, \tag{9.2}$$

kde $P_c(C)$ je množina třídy samotné a všech jejich rodičů bez "univerzálního předka" všech tříd, v tomto případě *Object*:

$$P_c(C) = \{C, P(C), P(P(C)), \ldots, Object\} \setminus Object. \tag{9.3}$$

Tím máme nastavnu podmínku nutnou – bez rovnosti (tak, jak je výše definována) tříd nemá smysl porovnávat objekty. Nyní se můžeme zaměřit na porovnávání objektů a jejich proměnných samotných. Vzhledem k tomu, že máme zaručenu rovnost tříd, můžeme

očekávat zhruba stejnou strukturu, názvy a typy instančních proměnných. Nicméně nemůžeme očekávat přímou rovnost těchto parametrů, díky tomu, jak jsem výše zadefinoval rovnost proměnných. Proto platí:

$$F_A \subset F_B \vee F_A \subset .F_B \tag{9.4}$$

Naším cílem je ovšem porovnání rovnosti. Musí tedy platit:

$$\|F_A\| = \|F_B\|. \tag{9.5}$$

Tato operace je pouhé porovnání dvou celočíselných hodnot (plus pravděpodobně zjištění velikosti pole), což je zanedbatelná operace, která algoritmus nezpomalí. Tímto je zajištěno, že porovnávané objekty patří pod stejnou třídu či jejího potomka, který nepřidává žádné instanční proměnné, tj. je z pohledu paměti zaměnitelný. To samozřejmě neznamená, že je stejný z pohledu sémantiky či funkcionality; to je však cena za to, že hledáme maximální možnou úsporu. Toto je ovšem funkcionalita, která je jednoduše regulovatelná, tj. v algoritmu je možnost přidat parametr, který bude ovlivňovat způsob porovnávání rovnosti tříd a případně zajistí, že takto budou označeny pouze třídy skutečně stejné, bez ohledu na dědičnost.

Nyní jsou tedy objekty stejných (dle výše definovaných měřítek) tříd a mají stejný počet proměnných. Vzhledem k tomu, že proměnnou nelze při dědičnosti odebrat ani pozměnit. Maximálně je možné ji pouze překrýt, případně změnit pomocí reflexe, což jsou okrajové případy, jejich řešení by algoritmus pouze zbytečně zpomalovalo.

Nyní je tedy možné definovat rovnost objektů následovně:

$$E(A,B) \Leftrightarrow E_C(C_A, C_B) \wedge \forall F_A^i, F_B^j \epsilon F_A, F_B : i = j : F_A^i = F_B^j. \tag{9.6}$$

Jinak řečeno – objekty A, B jsou si rovny právě tehdy, když jsou si rovny jejich třídy pomocí výše navrhnuté metody a zároveň platí, že jsou si rovny všechny dvojice proměnných daných objektů.

Teoreticky by samozřejmě bylo možné vyhledávat i napříč různými třídami, pouze na základě rovnosti všech hodnot proměnných. To by ovšem nedávalo velký smysl. Objekty, které algoritmus má vyhledat, jsou kandidáty pro odstranění z programu a nahrazení zbylou, jedinou instancí. To by v případě nerovnosti tříd nebylo možné. Naproti tomu, v případě nerovnosti tříd, kdy je ovšem jedna z nich součástí cesty dědičnosti třídy druhé, to možné je – samozřejmě ale ne vždy. Minimálně to ale dává prostor se zamyslet, zda je skutečně nutné používat dědičnost, když jsou data obou objektů naprosto totožná. Pokud chceme dodržovat sémanticky správný pohled na objekty, nemusíme je kopírovat; stačí uchovávat pouze jedinou instanci a na správných místech použít přetypování (což je de facto polymorfismus).

9.2 Rovnost složitějších objektů

Výše navrhnutý indikátor rovnosti ovšem postačuje pouze pro objekty tříd, jejichž proměnné jsou všechny primitivního typu. To ovšem neplatí v běžném programu netriviálního rozsahu prakticky nikdy. Už například prosté použití řetězce, tedy třídy `String`, tuto podmínku nesplňuje.

Jak bylo popsáno v kapitole 3, referenční – tedy neprimitivní – typy jsou odkazovány pomocí jejich adresy. Výše definovaný algoritsmus by tak porovnával ji – pouze celočíselnou hodnotu. Takový algoritmus ovšem ve skutečnosti není o nic lepší, než prosté porovnání bytů daného objektu přímo v dumpu. Je tedy nutné zvolit vhodnější přístup.

9.2.1 Porovnávání referencovaných objektů

V případě, že je typ jedné z proměnných třídy jinou třídou (a proměnná tedy obsahuje referenci na instanci této třídy nebo `null`), měli bychom porovnat i rovnost tohoto odkazovaného objektu s objektem odkazovaným z druhé instance. Zobecníme tedy výše definovanou rovnost na funkci $E_F(f_1, f_2)$:

$$E(A,B) \Leftrightarrow E_C(C_A, C_B) \wedge \forall F_A^i, F_B^j \epsilon F_A, F_B : i = j : E_F(F_A^i, F_B^j) \tag{9.7}$$

a tuto funkci definujeme jako:

$$E_F(f_1, f_2) = \begin{cases} f_1 = f_2 & \text{pokud je typ proměnné primitivní} \\ E(f_1, f_2) & \text{pokud je typ proměnné referenční} \end{cases}. \tag{9.8}$$

Z těchto vztahů ale vyplývají dva problémy:

- Není zajištěna ochrana proti zacyklení v rámci rekurze.
- Pro rekurzi není nastavena zastavovací podmínka.

První problém můžeme vyřešit jednoduše v rámci původní rovnice, její menší úpravou:

$$E(A,B) \Leftrightarrow E_C(C_A, C_B) \wedge \forall F_A^i, F_B^j \epsilon F_A, F_B : i = j \wedge D(A, B, F_A^i, F_B^j) : E_F(F_A^i, F_B^j) \tag{9.9}$$

A pomocnou funkci $D(A, B, f_1, f_2)$, která zjišťuje prostou odlišnost objektů (na úrovni reference, adresy) a slouží tedy jako ochrana proti zacyklení:

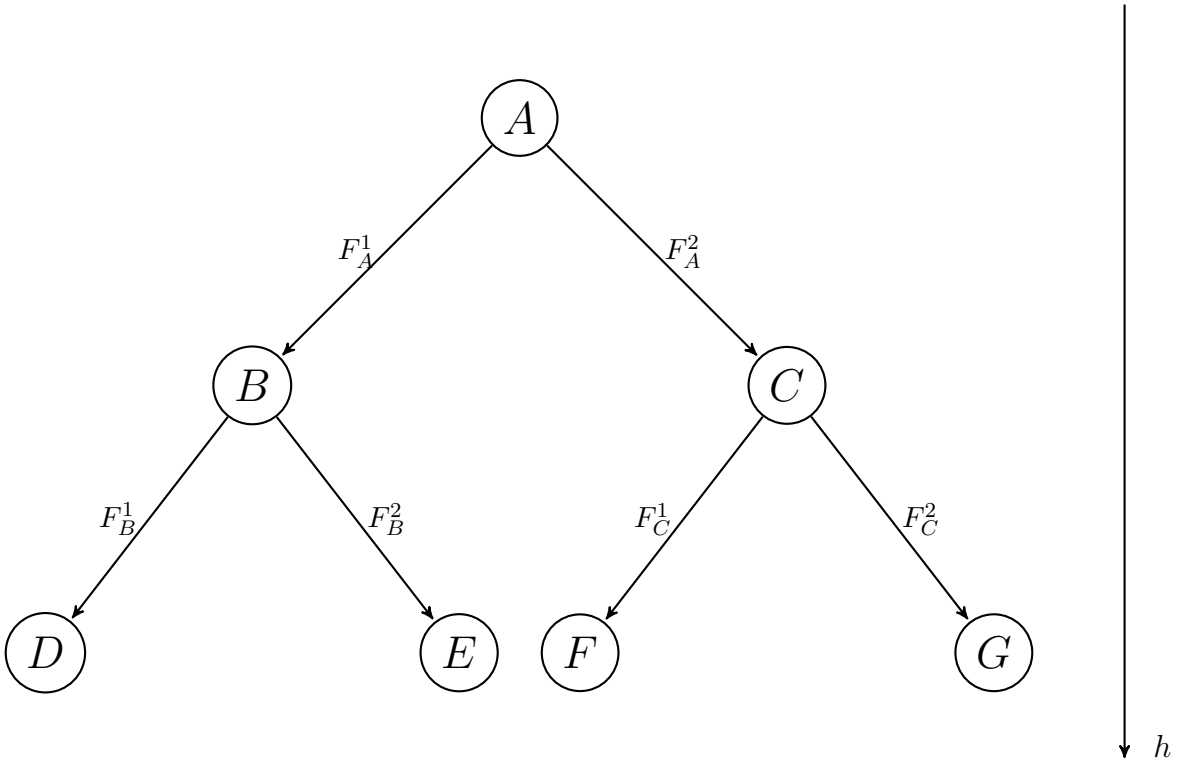
$$D(A, B, f_1, f_2) = f_1 \neq A \vee f_1 \neq B \vee f_2 \neq A \vee f_2 \neq B \tag{9.10}$$

Problém druhý, tedy zastavovací podmínku, můžeme vyřešit pouhým počítáním hloubky (zanoření) porovnávání a přerušením tohoto zanořování v určité hloubce – ideálně nastavitelné externě tak, abychom mohli volitelně ovlivňovat rychlost, a nevyhnutelně tak i přesnost, algoritmu. Formálně je tento problém tedy možné definovat tak, že porovnávání

referenčních typů budeme chápat jako strom, jehož rozvoj ukončíme při dosažení určité hloubky h.

Tento strom je vidět na obrázku 5. Mějme dva objekty, A_1 a A_2. Oba mají po 2 proměnných, $F^1_{A_1}, F^2_{A_1}$, respektive $F^1_{A_2}, F^2_{A_2}$. Označme operaci tohoto porovnání jako A a porovnání dvojic třídních proměnných jako F^1_A, F^2_A. Hloubka zanoření je v tuto chvíli $h = 0$. Rekurzivně takto budeme sestupovat, dále tedy provádíme porovnání B a C obdobně, jako porovnání A, tentokrát s hloubkou zanoření $h = 1$.

Takto bychom sestupovali (dále s operacemi v hloubce $h = 2$), dokud bychom nenarazili na horní limit h_{max}. V ten okamžik se můžeme zachovat ke všem proměnným jako k primitivním typům a dále rekurzivně nesestupovat.



Obr. 5: Strom porovnávání objektů

9.2.2 Porovnávání řetězců

Porovnání řetězce jako takového je poměrně jednoduché, kromě vlastního řešení, které může být více či méně efektivní, můžeme byty převést opět do objektu typu `String` a využít metodu `equals`, kterou Java nabízí.

Nicméně, pro zrychlení porovnávání můžeme využít výše zmíněného poznatku, že se Java v některých případech snaží uchovávat řetězce v tabulce symbolů. V takovém případě

by stačilo pouze porovnat adresu, na kterou proměnná typu `String` ukazuje. To je prosté porovnání celočíselných hodnot, tedy triviální operace.

9.2.3 Porovnávání polí a kolekcí

Porovnávání polí je opět poměrně jednoduché a znamená prosté iterování nad všemi položkami pole a jejich porovnání podle výše zmíněných metod.

Musíme se pouze zamyslet, jakým způsobem chceme toto iterování provádět – respektive, zda jsme ochotni vyměnit rychlost za přesnost. Prvky pole mohou být v různém pořadí. V takovém případě se problém redukuje na problém řazení, kterey je v nejlepším případě $O(NlogN)$ (např. pro *Quick sort*). Nabízí se ovšem otázka, zda k takovému kroku chceme přistupovat – implementace programu může být na pořadí prvků závislá a pokud bychom označili objekty s takovými poli jako duplikáty, mohlo by se jednat o falešně positivní případ. Rovnost polí $E_A(f_1, f_2)$, kde f_1, f_2 jsou třídní proměnné typu pole, obsahující prvky $p_1, p_2, ..., p_N$ je tedy možné definovat jako:

$$E_A(f_1, f_2) = \forall f_1^i, f_2^j \epsilon f_1, f_2 : i = j : E_F(f_1^i, f_2^j) \tag{9.11}$$

V případě kolekcí takto jednoduše rozhodnout možné není. Musíme si uvědomit, že přestože některé třídy sdílejí určité společné rozhraní (třeba `Collection`, `Map` apod.), jejich implementace uvnitř záleží na typu struktury, kterou zvolili pro reprezentaci ukládaných dat. V případě třídy `ArrayList` je možné takový problém redukovat na problém porovnávání pole; naopak v případě `LinkedList` je nutné několikanásobné porovnání N objektů. To lze samozřejmě také redukovat na problém porovnání pole a dívat se na propojení listu na – de facto – iterátor. Nicméně už taková operace vyžaduje mezikrok navíc.

9.3 Efektivita využití polí a kolekcí

TODO

10 ANALÝZA

11 IMPLEMENTACE

12 OVĚŘENÍ IMPLEMENTACE

13 ZÁVĚR

SEZNAM OBRÁZKŮ

,

SEZNAM ZKRATEK

JVM Java Virtual Machine

AOT Ahead-Of-Time

JIT Just-In-Time

GC Garbage Collector

PID Process Identifier – Identifikátor procesu

CLI Command Line Interface

OQL Object Query Language

LITERATURA

[1] TODO Citace Moore's law

[2] TODO Citace mzdy programátorů

[3] TODO https://www.graalvm.org/docs/getting-started/#native-images

[4] Developer Survey Results 2018 - Most Popular Technologies. Stack Overflow Insights [online]. 2018 [cit. 2019-03-02]. Dostupné z: https://insights.stackoverflow.com/survey/2018#most-popular-technologies

[5] TIOBE Index for March 2019. TIOBE - The Software Quality Company [online]. 2019 [cit. 2019-03-02]. Dostupné z: https://www.tiobe.com/tiobe-index/

[6] JSR-133. JSR-133: JavaTM Memory Model and Thread Specification: Proposed Final Draft. 2004. Dostupné z: https://download.oracle.com/otndocs/jcp/memory_model-1.0-pfd-spec-oth-JSpec/

[7] TODO https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#gc()

[8] TODO https://openjdk.java.net/jeps/241

[9] TODO https://docs.oracle.com/en/java/javase/11/migrate/migration-guide.pdf

[10] TODO https://docs.oracle.com/javase/7/docs/technotes/tools/share/jhat.html

[11] TODO http://www.csd.uwo.ca/courses/CS4411b/pdf02manuals/oql.pdf

[12] TODO https://books.google.cz/books?id=bAA9Z5WLdjQC&dq=OQL+specification&hl=cs&source=gbs_navlinks_s

[13] TODO http://www.cs.utexas.edu/users/mckinley/papers/mmtk-sigmetrics-2004.ps

[14] TODO Mastering Java 11: Develop modular and secure Java applications using concurrency and advanced JDK libraries, 2nd Edition

[15] TODO https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html

[16] TODO https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html

[17] TODO https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-1.html#jvms-1.2

[18] TODO https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html

[19] TODO Enterprise Java Security: Building Secure J2EE Applications Od autorů: Marco Pistoia, Larry Koved, Nataraj

[20] TODO https://medium.com/@govinda_raj/all-you-need-to-know-jvm-36c0b59ba969

[21] TODO http://hg.openjdk.java.net/jdk6/jdk6/jdk/raw-file/tip/src/share/demo/jvmti/hprof/manual.html

SEZNAM PŘÍLOH

A PŘÍLOHY