Java Virtual Machine

LÓGR

Technická dokumentace[1]

[1] LaTeX ver., 29. listopadu 1999

http://www.ms.mff.cuni.cz/~mdvo4092/logr

Lógr team:

Jan Ducháček
Martin Dvořák
Vojtěch Filip
Martin Frýdl
Ondřej Martínek
Tomáš Sieger
Josef Štěpán

Obsah

Kapitola 1

Java & Lógr

1.1 Programovací jazyk Java

Java začala svůj život jako programovací jazyk určený na vývoj software pro spotřební elektroniku jako jsou topinkovače, mikrovlnné trouby a digitální diáře. Software pro spotřební zařízení má z pohledu návrháře určité jednotné požadavky. Musí být schopen pracovat na nových počítačových čipech ihned po jejich uvedení na trh - výrobci často mění čipy, které používají, pokud je nový čip finančně efektivnější nebo pokud nabízí nové funkce. Software rovněž musí být extrémně spolehlivý, protože jakmile spotřební produkt nefunguje nebo se poškodí následkem SW chyby, musí výrobce uhradit škodu.

Malý tým ve firmě Sun, který vedl James Gosling a který na tomto problému pracoval, brzy zjistil, že existující programovací jazyky jako je C/C++ nejsou pro tento úkol vhodné. Tak například program napsaný v jazyce C/C++ musí být přeložen vždy pro konkrétní počítačový čip. Jakmile se objeví nový čip, musí se program rekompilovat. Rovněž díky složitosti a nízké úrovni jazyka je obtížné napsat spolehlivý SW.

V důsledku toho zahájil Gosling v roce 1990 návrh nového programovacího jazyka, který by byl pro spotřební elektronický SW vhodnější. Tento jazyk, původně známý jako Oak, byl malý, spolehlivý a nezávislý na architektuře.

V roce 1993, kdy tým jazyka Java pokračoval ve vývoji nového jazyka se na Internetu objevil WWW a vzal jej útokem. Vývojáři jazyka Java si uvědomili, že takový jazyk, nezávislý na architektuře, jako je Java, bude ideální pro programování na Internetu, protože program může běžet na všech různých typech počítačů do něj připojených. Ve skutečnosti se tak Java navzdory všem původním cílům svého vývoje stala mocným prostředkem

pro programování na Internetu.

V „The Java Language: A White Paper“ charakterizuje SUN jazyk Java následujícím způsobem:

> *Java: jednoduchý, objektově orientovaný, distribuovaný, interpretovaný, robustní, bezpečný, nezávislý na architektuře, přenosný, vysoce výkonný, víceprocesní a dynamický jazyk.*

Tato smršť formálních pojmů ve skutečnosti poměrně elegantně charakterizuje jazyk Java. V dalším textu následuje popis některých zajímavých a důležitých rysů jazyka.

Kompilátor jazyka Java vytváří místo skutečného strojového kódu tzv. bajtový kód. Java je interpretovaný jazyk a tak, aby se program v jazyce Java mohl být spuštěn, musí se pomocí interpretu jazyka Java přeložené bajtové kódy vykonat. Bajtový kód jazyka Java představuje formát objektového souboru nezávislý na architektuře. Kód je tedy určen pro efektivní přenos programů na více platforem. Program v jazyce Java je možné spustit na kterémkoli systému, na němž běží interpret jazyka Java a systém run-time. Dohromady interpret a systém run-time vytváří virtuální stroj nazývaný virtuální stroj jazyka Java (Java Virtual Machine).

Nezávislost na architektuře je velkou částí přenositelnosti. Java však jde v tomto směru ještě dále, a to tím, že zajišťuje, že neexistují žádné 'implementačně závislé' aspekty specifikace jazyka. Java tak například explicitně určuje velikost každého z primitivních datových typů. Stejně jako jejich chování v aritmetických operacích.

V každém interpretovaném prostředí se standardní linkovací fáze vývoje programu do značné míry ztrácí. Pokud má Java vůbec nějakou linkovací fázi, jedná se pouze o proces zavedení nových tříd do prostředí, což je inkrementální proces. V důsledku toho podporuje Java rychlé prototypování a snadné experimentování, což vede k rychlejšímu vývoji programů. To je v příkrém kontrastu s tradičním, časově náročným procesem překládání, linkování a testování.

Jedním z nejvýznamnějších rozšíření ve smyslu robustnosti oproti klasickým programovacím jazykům jako C/C++ je paměťový model jazyka Java. Java nepodporuje ukazatele, díky čemuž je vyloučena možnost přepsání paměti a poškození dat. Protože Java nemá struktury, a protože pole a řetězce jsou objekty, nejsou ukazatele vůbec třeba. Java automaticky ošetří vytváření a rušení odkazů. Rovněž obsahuje automatické slučování volných kusů paměti (garbage collection), takže není vůbec nutné zabývat se správou paměti. Garbage collection v jazyce Java předchází vzniku neplatných odkazů

na ukazatele, dírám v paměti, visícím ukazatelům a dalším zhoubným chybám spojeným s dynamickou alokací a dealokací paměti.

Interpret jazyka Java rovněž provádí řadu kontrol v době běhu programu, jako je například ověřování, jestli jsou indexy polí a řetězců v definovaných mezích.

Dalším rysem jazyka Java, který přispívá k robustnějším programům, je zpracování výjimek. Výjimka je příznak, že došlo k určitému typu výjimečného stavu, jako je například chyba.

Řada obranných mechanismů v jazyce Java je určena k ochraně před nedůvěryhodnými aplikacemi. Run-time systém jazyka Java pomocí procesu verifikace bajtového kódu zajistí, že kód načtený přes síť nenarušuje žádná z jazykových omezení jazyka Java.

Java je interpretovaný jazyk, takže nikdy nebude stejně rychlá jako kompilovaný jazyk, jakým je například jazyk C. Ve skutečnosti je interpretovaná Java v průměru 20-krát pomalejší než jazyk C.

1.2 Co je Lógr

Jak bylo popsáno v předchozí kapitole, při klasickém přístupu je program napsaný v jazyce Java překladačem kompilován do bajtového kódu a uložen do souborů .class. Bajtový kód je poté interpretován virtuálním strojem JVM (Java Virtual Machine). To umožňuje jednou napsaný program spouštět na libovolné platformě, kde je JVM implementována. Obvykle je bohužel v důsledku interpretace běh takové aplikace pomalý a tak je nutné používat urychlovací techniky.

Interpret může při prvním průchodu nahradit kritickou instrukci odpovídající quick instrukcí, aby při pozdějších průchodech bylo její vykonání rychlejší.

Novější JVM také zpravidla obsahují JIT - 'just in time' kompilátor, který za běhu analyzuje kritické tj. často využívané části kódu a ty přeloží do nativního kódu procesoru, aby tak zrychlil jejich běh.

Další možností jsou přímé překlady ze zdrojových souborů .java do nativního kódu procesoru. Zde jsou možné dva přístupy.

Buď se vytvářejí samostatné aplikace, které však musí obsahovat značnou část run-time virtuálního stroje (jako například garbage collector). Tím pádem se i malá aplikace stane rozsáhlou. Cenou za kvalitní kód a to, že není potřeba JVM implementovaná pro danou platformu, je ztráta přenositelnosti.

Nebo se ze zdrojového souboru generují speciální soubory obsahující na-

tivní kód, které jsou obdobou souborů .class. K vykonání takových souborů se použije JVM, která už pracuje nad přeloženým nativním kódem a obsahuje potřebný runtime.

Velkou výhodou obou předchozích přístupů je možnost několikaúrovňové a tedy velmi kvalitní optimalizace generovaného kódu. Nevýhodou je pomalejší kompilace.

JVM Lógr kombinuje předchozí přístupy — překládá po jednotlivých třídách do nativního kódu procesoru, ale nikoliv ze zdrojových textů, ale ze souborů .class. Lógr (stejně jako JIT kompilátory) využívá formátu bajtového kódu, který byl navržen tak, aby generování kódu z něj nebylo obtížné a dávalo poměrně dobrý kód. Bajtový kód ze souborů .class, které má Lógr provést, tedy není interpretován, ale kompletně se přeloží do nativního kódu procesoru (v našem případě Intel). Jednou přeložené třídy se ukládají, a tak postačuje při jejich opětovném natažení zkontrolovat, zda se odpovídající .class soubor nezměnil. Překlad se provede pouze pokud odpovídající soubor .logr neexistuje nebo pokud byla detekována změna souboru .class a .logr se tak stal neaktuálním. Navenek se JVM Lógr chová jako klasická JVM (např. java z JDK). Zůstává veškeré její chování jako „lazy“ dotahování tříd. Veškerý „run-time“ je uvnitř JVM Lógr. Uživatel tedy může přímo, místo aby použil interpret java, použít logr. Nejsou nutné žádné kompilace ani úpravy kódu. Dále žádným způsobem nezanáší do `.class` souborů nativní kód - ten je naprosto izolován. Tím nenarušuje známé heslo technologie Java „write once run everywhere“. Generované soubory .logr slouží pouze pro potřebu JVM Lógr.

V JVM Lógr tedy zůstává výhoda přenositelnosti jazyka Java a nabízí JVM šitou na míru dané platformě, čímž umožňuje aplikacím rychlejší běh.

Kapitola 2

Organizace dokumentace

Základem technické dokumentace je specifikace Java Virtual Machine vydané firmou SUN [Spec97]. Tato specifikace je de facto horní vrstvou dokumentace, která definuje abstraktní architekturu JVM. Text, který právě čtete, je dokumentací vrstvy spodní. Je v něm popsáno, jak byla specifikaci odpovídající JVM implementována. Z předchozích řádků vyplývá, že [Spec97] je nedílnou součástí technické dokumentace. V důsledku toho se v následujícím textu předpokládá znalost termínů, datových struktur a mechanizmů v ní uvedených. Technická dokumentace se velmi často na specifikaci odkazuje. V těchto místech je odkaz explicitně uveden a zpravidla je upřesněn například číslem kapitoly.

Úvodní část dokumentace nastíní, jak vypadá jeden životního cyklus JVM Lógr. Budou zmíněny nejdůležitější součásti tvořící virtuální stroj a jejich spolupráce při vykonávání programů, aby se tak ujasnil obsah následujících kapitol.

V druhé části dokumentace je rozebrána organizace Lógr haldy a její datové struktury. Je popsán způsob práce s referencemi, datové struktury pro uložení tříd a jejich instancí v haldě, struktura zavaděčů tříd a systém garbage collection.

Třetí část obsahuje dokumentaci běhových struktur a vnitřních mechanismů virtuálního stroje. Pojednává o implementaci vláken, synchronizaci na úrovni vláken a jazyka Java, struktuře zásobníku, způsobu volání metod a přístupu na položky, přístupu do jádra JVM Lógr, interním systému pro volání nativních metod, práci s výjimkami, startu a ukončení běhu JVM.

Následuje dokumentace verifikátoru bajtového kódu a 'bytecode to native' překladače. Pro lepší orientaci ve zdrojových kódech JVM Lógr následuje kapitola obsahující použité konvence a stručný popis obsahu každého

souboru. Technickou dokumentaci uzavírá slovníček pojmů a přehled použité literatury.

Kapitola 3

Stručný popis životního cyklu

V této kapitole bude nastíněn jeden životní cyklus virtuálního stroje. Tato část dokumentace není v žádném případě formálním popisem běhu JVM, ale pouze úvodem k následujícím kapitolám. Snaží se na úvod čtenáři přiblížit hlavní části a vnitřní běhové mechanizmy JVM Lógr.

3.1 Start

Nechť je Lógr spuštěn takto:

```
% logr classContainingMain ChangeMyCase
```

Prvním argumentem programu logr je plně kvalifikované jméno třídy obsahující metodu

```
public static void main(String[] args)
```

která má být spuštěna. Jediným parametrem je pole řetězců, což jsou argumenty z příkazové řádky.

3.2 Inicializace

JVM Lógr se nejprve inicializuje, jak je popsáno v kapitole 5.11. Vytvoří se Lógr halda(LH), připraví se struktury nutné pro běh Linux vláken, správa Java výjimek, implicitní zavaděč tříd, struktury používané standardním rozhraním Java native interface (JNI) a garbage collector (GC).

Příprava na spuštění classContainingMain pokračuje zavedením systémových tříd. Zavedení je proces nalezení binární formy třídy nebo rozhraní. Buď přeložením za běhu, ale typicky získáním binární reprezentace již dříve přeložené.

3.3 Soubory .java, .class a .logr

Aby bylo možné vykonat program jazyka Java v JVM Lógr, je nutné nejprve přeložit soubor .java obsahující zdrojový kód třídy do souboru `.class` obsahujícího bajtový kód například překladačem javac firmy Sun.

Soubor `.class` (viz kapitola 4. [Spec97]) obsahuje binární reprezentaci právě jedné třídy nebo rozhraní. Jeho jméno bez přípony je stejné jako jméno třídy resp. rozhraní, které obsahuje.

Lógr pracuje nad soubory `.class` a vytváří z nich soubory `.logr` (kapitola 4.3). Struktura tohoto binárního souboru je podobná struktuře souboru `.class`. Hlavní odlišností je to, že soubor `.logr` neobsahuje bajtový kód, ale **nativní kód procesoru i386** přeložený překladačem BCC (kapitola 5.14). Soubor `.logr` tedy vzniká překladem souboru `.class`. Jeho formát je navržen tak, aby práce s ním byla efektivnější, než byla s původním souborem `.class`.

Vstupní `.class` soubor je nejprve prověřen verifikátorem bajtového kódu (kapitola 5.13). Pokud bajtový kód obsažený v `.class` souboru splňuje omezení jazyka Java, je akceptován a předložen ke zpracování kompilátoru BCC. Soubor `.logr` se následně zavádí do paměti virtuálního stroje.

Proces zavádění souborů `.class` je implementován ve třídě java.lang.classLoader a jejích podtřídách. Různé podtřídy java.lang.classLoader mohou implementovat vlastní zaváděcí politiky. Zavádění souborů `.logr` je implementováno ve vnitřních funkcích JVM (kapitola 4.4).

Ale zpět ke startu virtuálního stroje a zavádění systémových tříd. Nabízely se dva způsoby, jak tento úkol provést:

1. Buď přesně definovaný korektní rozvrh zavádění systémových tříd, který udává takové pořadí, aby nedocházelo k rekurzivnímu zavádění předků. Toto řešení má tu výhodu, že je efektivnější a tedy i rychlejší.

2. Námi zvolené řešení je zavádění klasickým „lazy“ způsobem. Má-li být zavedena třída, musí být zavedeni její předkové (pokud se tak dosud nestalo). Tento postup se aplikuje rekurzivně. Postupné zavádění tříd, které jsou právě potřebné, způsobí korektní inicializaci.

3.4 Zavedení třídy a vytvoření její reprezentace v paměti

Nejprve se musí třída zavést společně se svými předky do paměti. Následně je nutné vytvořit reprezentaci třídy v paměti a zapojit ji do struktur virtuálního stroje, tj. převést ji ze „syrového“ stavu do stavu, kdy ji bude možné používat (kapitola 4.2).

Třída je zavedena do Lógr haldy (LH), což je oblasti paměti virtuálního stroje, do které se ukládají Java třídy a jejich instance. Dvěma hlavními strukturami, které tvoří třídu v JVM, jsou FixedData a RuntimeData.

FixedData obsahují paměťové objekty třídy statické povahy jako kód metod třídy a strukturu NamePool, jejíž použití bude popsáno níže.

Při zavádění souboru `.logr` a vytváření struktury RuntimeData je nutné provést relokace kódu metod. Ke kódu jsou přiloženy tabulky, které udávají relativní adresy, na kterých je nutné relokovat. Relokují se např. přístupy na „this“ položky, přímé přístupy do polí, přístupy na „null“ či volání Logr Kernel Interface funkcí (kapitola 5.4). Logr Kernel Interface je tvořen sadou funkcí, které nahrazují instrukce bajtového kódu, které se nepřekládají. Jsou to takové instrukce, jejichž překlad do nativního kódu by byl neefektivní nebo nějakým způsobem problémový. Například alokace vícerozměrných polí nebo volání určitých typů metod.

FixedData dále obsahují strukturu NamePool. Po svém vytvoření se struktura NamePool používá při vyhledávání adresy začátku kódu Java metody a rovněž adresy, na které je uložena určitá položka. Podrobně jsou algoritmy vyhledávání popsány v kapitole 5.8.

RuntimeData naopak uchovávají paměťové objekty dynamického charakteru. Kromě jiného obsahují tabulku virtuálních metod, statická data třídy a strukturu ReferencePool.

Tabulka virtuálních metod (VMT) obsahuje informace o všech virtuálních metodách třídy (tedy i o zděděných). Při konstruování VMT se postupuje od předků k třídě, pro kterou se VMT staví. Protože určitá virtuální metoda je na stejném indexu ve VMT v každém z předků třídy, u předefinovaných metod se korektně přepisuje odpovídající index ve VMT. Položka VMT obsahuje referenci na kód, ve kterém metoda je, a dále relativní adresu metody v tomto kódu.

Uvnitř struktury RuntimeData se dále rezervuje prostor pro ReferencePool. V klasické JVM se metody resp. položky vyhledávají při každém použití vždy znovu (JDK toto vyhledávání částečně eliminuje použitím quick instrukcí bajtového kódu). Lógr má přidánu v každé třídě strukturu Refe-

rencePool, která obsahuje již jednou nalezené adresy kódu metod a položek. Tyto adresy byly nalezeny při prvním aktivním použití pomocí odpovídajících struktur NamePool. Při opakovaném použití se tedy používají přímo adresy ze struktur ReferencePool. Na struktury ReferencePool se odkazuje přímo z přeloženého kódu. Vyhledávání položek a metod je popsáno rovněž v kapitole 5.8.

3.5 Inicializace třídy

Před prvním aktivním použitím třídy classContainingMain se musí provést inicializace jejích statických položek. V jazyce Java má programátor možnost definovat inicializátory jak pro proměnné třídy, tak pro proměnné instancí. Proměnné třídy se inicializují v okamžiku, kdy je třída poprvé zavedena. Inicializační metoda pro proměnné třídy se nazývá statický inicializátor (kapitola 5.9.1). [1] V době překladu BCC vygeneruje volání Java metody, která je statickým inicializátorem třídy a ta se při prvním aktivním použití zavolá a třídu inicializuje.

3.6 Vytvoření instancí systémových tříd

U některých tříd postačují pouze statické metody a položky primitivních typů. Jinde tomu tak není, a proto je nutné vytvořit jejich instance (kapitola 5.9). Příkladem tříd, jejichž instance je nutné vytvořit, jsou například Class, Thread a String.

3.7 Vlákna

Jak již bylo zmíněno, inicializace vláken je prováděna rovněž při startu JVM. Pro implementaci vláken jsme použili vlákna operačního systému Linux. Java vlákna a Linux vlákna jsou spolu svázána jak je naznačeno v obrázku. Java vlákno má referenci na Linux vlákno a vice versa. Podrobná dokumentace je obsažena v kapitole 5.1.

[1] Otázka: Snad vygeneruje jen metodu, a ne volání, ne?
Odpověď: Neprázdná metoda provádějící inicializaci je vygenerována překladačem jazyka Java (javac). V JVM Lógr je pro jednoduchost statický inicializátor dogenerováván tak, aby byl přítomen v každé metodě (obsahuje alespoň ret - „prázdný inicializátor“). Potom BCC může vždy vložit volání tohoto inicializátoru, jelikož je vždy přítomen (a to je to generování volání). Alespoň myslím, že to tak je...

3.8 Argument pro main

Připraví se argument pro metodu main. Z parametrů, které dostala spouštěná třída na příkazové řádce, se vytvoří instance třídy String. Jak bylo již dříve uvedeno, jediným parametrem metody main je pole řetězců. Požadavek na implicitní zavaděč tříd o java.lang.String způsobí její natažení (pokud již není v paměti). Nakonec se vytvoří pole řetězců s jedinou položkou, kterou je `ChangeMyCase`.

3.9 Natažení třídy classContainingMain

JVM je inicializována, argumenty vykonávané třídy jsou připraveny, a tak je čas požádat implicitní zavaděč tříd o třídu classContainingMain, která obsahuje metodu main.

3.10 Spuštění metody main

Volání metody provádí trampolína. Je to volání statické metody Java. Všechny druhy volání metod jsou podrobně popsány v kapitole 5.8. Následuje vykonání metody main. Při jejím provádění se využívají struktury NamePool a ReferencePool pro volání dalších metod a přístup na položky. Nově vytvářené instance se ukládají do Lógr haldy a ve chvíli, kdy se stanou nedostupnými, je garbage collector odstraní (kapitola 4.5). Za běhu může dojít k určitému typu výjimečného stavu, jako je například chyba. V takovém případě dojde k vyhození výjimky.

3.11 Vytvoření výjimky

Pokud má dojít k vyhození výjimky, stane se toto: Nejdříve se musí vytvořit instance výjimky (provede se stejně jako vytvoření instance obyčejné třídy). Zavolá se speciální LKI funkce určená k ošetřování výjimek a jako parametr se jí předá reference na instanci výjimky, která právě nastala. Podrobně je hledání obsluhy výjimky popsáno v kapitole dokumentace 5.10.2. Může se stát, že výjimka vybublá až k rámci metody main. V tom případě je vyvolána metoda uncaughtException() v instanci třídy ThreadGroup, do níž patří právě běžící vlákno. Tato metoda vypíše patřičné informace a poté JVM skončí svoji práci.

3.12 Ošetření výjimky

Pokud je nalezen chráněný blok schopný zpracovat výjimku, vyčistí se pouze Java zásobník. Při čištění se jednak snižují počty referencí (tím zanikají odkazy ze zásobníku na paměťové objekty) a dále se přímo odstraňují primitivní typy. Když je struktura Java zásobníku vyčištěna, uloží se na něj reference na instanci výjimky a provede se odskok na adresu, kde je odpovídající catch blok (obsluha výjimky).

3.13 Konec metody main

Jakmile je kód metody main vykonán, provede se úklid Java zásobníku (jako po vykonání každé Java metody) tak, jak to bylo popsáno například v odstavci o bublání výjimek. Běh se vrátí do funkce, která spouštěla main, a tím je běh vlákna main ukončen (viz kapitola 5.1).

3.14 Ukončení běhu JVM

Nejdříve se ukončuje běh vláken. Ukončované vlákno zkontroluje, zda je posledním živým nedémonickým vláknem. Evidence vláken je vedena ve speciální systémové struktuře.

Každé vlákno má deskriptor, ve kterém je kromě jiného uvedeno, zda je ono samo démonem či nikoli. Pokud je tedy vlákno posledním žijícím nedémonem, je jeho povinností ukončit běh virtuálního stroje.

Ukončení se provede tak, že vlákno pošle všem démonům výjimku, kterou nelze odchytit. Postupně se uvolní systémové prostředky, které měli démoni v držení. Zánikem vláken běh JVM Lógr končí. Podrobně je celý postup popsán v kapitole 5.12.

Kapitola 4

Lógr halda

Běhový systém JVM Lógr pracuje nad Lógr haldou (LH), která je sdílena všemi vlákny. LH se vytváří při startu JVM. Je to datová oblast, ze které je alokována paměť pro datové struktury Java tříd, jejich instance a pole. V této kapitole bude nejprve popsána implementace LH samotné. Dále způsob, jakým jsou do LH Java třídy ukládány (kapitola 4.4), struktura jejich uložení (kapitola 4.2) a systém garbage collection provádějící recyklace a odstraňování tříd z paměti (kapitola 4.5).

4.1 Implementace LH

LH je tvořena jednou nebo více efemérními oblastmi proměnné délky. Efemérní oblasti jsou vytvářeny tak, jak rostou paměťové požadavky běhového systému. Paměť alokovaná haldou samotnou tedy nemusí být souvislá. Alokace jsou prováděny explicitně, dealokace, tj. recyklace paměti (až na výjimky), provádí garbage collector.

Deklarace struktur a symbolických konstant lze nalézt v Logr/include/referencetypes.h. Práce s LH se provádí pomocí funkcí z Logr/heap/reference.h a Logr/heap/reference.cc. Všechny funkce, které volá programátor pracující s haldou, jsou deklarovány zde. Podrobně se jim věnuje závěr kapitoly. V popisu algoritmů alokace reference a paměťového uzlu se používají metody a funkce, které programátor využívající haldu nikdy nevolá (je to popis interních mechanizmů). Ladící systém je v souboru Logr/heap/gcdebug.cc.

4.1.1 Alokace v LH

Alokaci v haldě lze rozdělit na dvě části — alokaci reference a alokaci uzlu paměti uvnitř efemérní oblasti.

Reference a její alokace

Do LH se přistupuje vždy zásadně užitím reference. Reference jsou dvojího druhu - InstRef a StatRef. Reference typu InstRef se používají pro odkazy na instance tříd, reference typu StatRef pro ostatní paměťové objekty. Další popis v kapitole 4.2. Struktura StatRef obsahuje následující položky:

.ptr	pozice v paměťového uzlu
.size	velikost bez servisní informace
.size	velikost bez servisní informace
.refCount	počet referencí z Java zásobníku
.lockCount	zámek bránící přesunům v haldě
.state	stav reference: 0 GC nepracuje s referencí 1 GC pracuje s referencí -1 reference není alokována
.name	jméno používané při ladění

.type	typ statické reference:
	`STATREF_VOID`
	`STATREF_FIXED_DATA`
	`STATREF_RUNTIME_DATA`
	`STATREF_ARRAY_HASHTABLE`
	`STATREF_CLASSLOADER_HASHTABLE`
	`STATREF_INTERFACE_METHOD_TABLE`

.fieldLock	zámek pro atomický přístup na dlouhé položky (long a double)
.finalizerFlags	příznaky, které používá garbage collector. Používají se pro uvolňování alokovaných objektů z interních struktur JVM a korektní volání ukončovačů instancí.

Struktura InstRef je prakticky stejná jako StatRef. Liší se pouze v tom, že neobsahuje type a navíc obsahuje následující položky:

.monitorOwner	vlastník monitoru
.monitorCount	kolikrát je monitor v držení stejným vláknem
.syncMutex	mutex

.syncCond	condition variable pro implementaci monitorů

Správa referencí obou typů je implementována ve třídě ReferenceVectors (Logr/include/gcreference.h a Logr/heap/gcreference.cc).
Třída obsahuje následující metody

StatRef *getStatRef()
alokace reference typu StatRef.

void putStatRef(StatRef *)
dealokace reference typu StatRef.

InstRef *getInstRef(), void putInstRef(InstRef *)
analogické funkce pro InstRef.

void clean()
privátní metoda provádějící optimalizaci vektorů referencí a položky.

a položky

.sVectorSize	velikost základního vektoru referencí typu StatRef
.sVectorIncrement	velikost vektoru při expanzi referencí StatRef
.statRefVecs	spojový seznam vektorů referencí StatRef
.sFL	free list nepoužitých referencí StatRef
.sShield	položka používaná zámkem aktivního čekání pro přístup do .sFL
.iVectorSize	
.iVectorIncrement	
.instRefVecs	
.iFL	
.iShield	analogické položky pro typ InstRef

Dále je funkci gcBody(), která je kódem vlákna garbage collection, umožněn přístup do této třídy a je tedy možné pracovat přímo s interními strukturami správy referencí.

Při startu JVM se vytvoří pro každý druh referencí základní vektor délky `GCREFVECS_START_SIZE` a vloží se do seznamu vektorů statRefVecs resp. instRefVecs. Dále se inicializují zámky a struktury free list.

Algoritmus alokace reference bude popsán například pro typ InstRef. Pro alokaci reference se použije metoda getInstRef(). Pokud jsou všechny

reference ve vektorech uvnitř instRefVecs již použity, vytvoří se nový inicializovaný vektor délky `GCREFVECS_INCREMENT` a provede se jeho zapojení do struktury free list. Free list je jednosměrný spojový seznam. Každý typ referencí má svoji vlastní strukturu free list. Nepoužité reference, tj. struktury StatRef resp. InstRef, jsou ve vektoru propojeny položkou .ptr. Díky tomu je irelevantní, zda vektory tvoří souvislý vektor nebo ne.

Dále je v nepoužité referenci nastaven `.refCount` na `REF_NOT_VALID`. Tento příznak využívá garbage collector při průchodu vektorem pro určení platnosti reference. Protože do struktury free list přistupují současně alokační funkce běžících vláken a garbage collector, operace nad seznamy jsou synchronizovány výhradním zámkem s aktivním čekáním.

Alokace reference je vypojením hlavy seznamu struktury free list, dealokace je vložením reference do hlavy seznamu. Tato strategie byla zvolena, aby se využíval pokud možno omezený okruh referencí a nedocházelo k fragmentaci obsahu vektorů.

InstRef se liší pouze tím, že provádí inicializace a kontroly zámků z knihovny pthread.

Privátní metoda clean() optimalizuje vektory referencí. Metoda organizuje free list tak, aby stoupal s rostoucím indexem ve vektoru referencí. Protože alokace nové reference se provádí vždy z hlavy seznamu free list, po určitém čase se v případě klesajícího počtu využitých referencí uvolní poslední vektor v seznamu vektorů, a je možné jej dealokovat. Na druhou stranu je ovšem sporné, zda je tato optimalizace přínosem:

1. Není vůbec jisté, že se poslední vektor úplně vyprázdní, aby bylo možné jej dealokovat.
2. Uspořádání struktury free list stojí procesorový čas. Proto je vhodné, aby byla alespoň volána v rámci démonických vláken.
3. Počet využitých referencí za běhu aplikace značně kolísá a tak je vektor s velkou pravděpodobností nutné alokovat v blízké budoucnosti znovu.

Metoda clean() je v aktuální verzi JVM Lógr zablokována, protože při testech se ukázalo, že nepřináší zrychlení. Jedinou její předností je zefektivnění práce s pamětí. Paměť využitá pro reference je ale řádově menší než např. paměť na haldě, ztráty jsou tedy zanedbatelné.

Alokace paměťového uzlu v efemérní oblasti

Systém řízení efemérních oblastí je implementován ve třídě GcHeap (Logr/include/gcheap.h a Logr/heap/gcheap.cc). Metody třídy umožňují alokace,

dealokace a změny velikosti paměťových uzlů. Dále třída umožňuje kontrovat integritu jak haldy jako celku, tak jednotlivých uzlů.

Halda implementovaná v GcHeap je rozšiřitelná. Při startu JVM se vytvoří iniciální efemérní oblast o velikosti dané parametrem příkazové řádky -ms (standardně 1MB). Za běhu se při zaplnění efemérních oblastí alokuje nová efemérní oblast dokud není překročen limit -ms (standardně 16MB).

Halda byla navržena tak, aby umožňovala rychlé elementární operace s důrazem na alokace. Dealokace totiž provádí asynchronní garbage collector vlákno a nezatěžuje tedy za běžných okolností běhový systém JVM. Strukturu free list tvoří 32 dvousměrných spojových seznamů volných paměťových uzlů. Každý obsahuje uzly, jejichž velikost je v určitém rozmezí a má výhradní zámek. Systém v principu umožňuje paralelní přístup až 29 vláken do haldy, to znamená, že může například několik vláken paralelně alokovat či dealokovat bez vzájemného z držování.

Halda je implementována v souborech Logr/include/gcheap.h a Logr-/include/gcheap.cc.

Paměťové uzly

Halda obsahuje alokované a volné paměťové uzly. Obrázku popisuje jejich strukturu.

Volný uzel je ohraničen na počátku a konci značkami `GCHEAP_CHUNK_FREE`. Na obou koncích je dále velikost uzlu včetně servisní informace. Velikost na konci se používá při defragmentaci volného místa haldy, která bude popsána níže. Poslední dvě položky jsou ukazatele na předchozí a následující uzel v odpovídajícím seznamu volných paměťových uzlů. Minimální velikost volného uzlu (případ kdy uzel obsahuje pouze servisní informaci) je `GCHEAP_SLACK_LIMIT` a je nejmenší alokovatelnou jednotkou v haldě.

Alokovaný uzel je ohraničen značkami `GCHEAP_CHUNK_FULL`. Servisní informace dále obsahuje pouze velikost uzlu včetně servisní informace.

Struktura free list

Free list je tabulka. Položkou je instance třídy FreeListItem:

`fake[]`	falešná hlavička se stejnou strukturou jako má paměťový uzel shield zámek chránící spojový seznam daného rozsahu před současným přístupem více vláken

bool get(int &, byte *&)

získání prvního uzlu ze spojového seznamu.

bool searchFor(int&, byte *&)
vyhledání paměťového uzlu požadované velikosti.

void listing()
výpis obsahu spojového seznamu volných paměťových uzlů.

Každá položka tabulky je hlavou seznamu volných paměťových uzlů v určitém rozmezí. Aby byla práce se spojovými seznamy jednodušší a rychlejší, struktura FreeListItem má stejnou (avšak falešnou) hlavičku jako paměťový uzel v haldě. Tabulka je organizována například takto:

Index v poli	Mocnina 2	Interval
0	31	nepoužit
1	30	nepoužit
2	28	nepoužit
...	...	...
10	21	nepoužit
11	20	$< 2^{20}, 2^{21-1} >$
12	19	$< 2^{19}, 2^{20-1} >$
13	18	$< 2^{18}, 2^{19-1} >$
...	...	...
28	4	$< 2^{4}, 2^{5-1} >$
29	3	nepoužit
30	2	nepoužit
31	1	nepoužit

Takto by vypadala tabulka pro velikost haldy 1,5MB. Horní intervaly nejsou použity, protože v haldě o této velikosti nikdy nebude tak velký volný uzel. Spodní intervaly jsou menší než nejmenší alokovatelná jednotka v haldě a jsou tedy rovněž zablokovány. Volné uzly těchto velikostí v haldě nikdy nevznikají. Pro určování intervalu při vkládání resp. odebírání volného paměťového uzlu ze struktury free list se používá hešovací funkce, která pro danou velikost vrátí index v poli.

Alokace
Parametrem metody alloc() třídy GcHeap je požadovaná velikost alokovaného uzlu size. Hešovací funkce vrátí pro size číslo intervalu ID. ID je nejbližší vyšší interval, který obsahuje paměťové uzly, jejichž velikost je větší nebo rovna size. Pokud je tedy seznam ID neprázdný stačí vypojit první uzel (v principu kterýkoli). Není nutné žádné prohledávání spojového seznamu ani

následná reorganizace. Ve skutečnosti je právě popsaný postup nejobvyklejší scénář alokace.

Pokud je seznam ID prázdný, postupuje se v cyklu směrem k seznamům, které obsahují uzly větších velikostí. Jestliže je některý z těchto seznamů neprázdný, opět stačí vypojit první uzel ze seznamu.

Teprve jsou-li všechny výše uvedené seznamy prázdné, prohledá se ten do kterého by svou velikostí size náležel. Pokud je takový uzel nalezen, vypojí se ze spojového seznamu.

Jestliže selhaly všechny předchozí pokusy, zkontroluje se horní limit velikosti haldy a pokud nebyl překročen, vytvoří se nová efemérní oblast s velikostí

$newEfemereAreaSize = min(max(size*10, ms), mx - allocatedBytes)$

kde ms a mx jsou parametry z příkazové řádky, allocatedBytes je počet alokovaných bajtů bez servisní informace.

V opačném případě metoda alloc() vrátí do vyšší vrstvy hodnotu, která způsobí vyhození výjimky OutOfMemoryError.

Při alokaci je zpravidla uzel nalezený v seznamu volných paměťových uzlů větší. Pokud je nepoužitá část větší než `GCHEAP_SLACK_LIMIT`, odlomí se a z fragmentu se vytvoří nový volný paměťový uzel, který se dle hešovací funkce vloží do odpovídajícího seznamu. Jestliže je menší než `GCHEAP_SLACK_LIMIT`, nechá se připojen k uzlu a zůstává nevyužit uvnitř alokovaného uzlu. Tím se zabraňuje fragmentaci haldy nepoužitelnými několikabajtovými uzly.

Dealokace

Až na výjimky volá metodu free() třídy GcHeap pouze garbage collector vlákno. Jejím parametrem je ukazatel na uvolňovaný paměťový uzel. Aby nedocházelo k fragmentaci volného místa, provádí se v době dealokace jeho defragmentace. Je nutné zabránit tomu, aby v haldě za sebou následovalo dva nebo více volných uzlů.

Nejprve se metoda pokusí zjistit zda uzel, který je bezprostředím předchůdcem uvolňovaného uzlu, je volný. To se zjistí ze značky na konci uzlu. Pokud je volný, podle velikosti, rovněž z konce uzlu, se získá od hešovací funkce interval ID ve kterém daný uzel je. Tento interval se zamkne. V této chvíli je zaručeno, že žádné jiné vlákno nepracuje s uzly z intervalu ID. Znovu se zkontrolují značky a velikost, protože mezi přečtením a zamčením mohlo jiné vlákno obsah uzlu změnit. Spojový seznam není nutné procházet. Z velikosti uzlu se určí relativní pozice položek odkazujících na předchozí a následující uzel. Provede se vyjmutí uzlu ze spojového seznamu. Dále se na-

staví značky na `GCHEAP_CHUNK_FULL`. Tím je uzel připraven k připojení k uvolňovanému uzlu. Stejně se postupuje v případě následníka.

Nakonec se přepočte velikost uvolňovaného uzlu (v případě, že jsme k němu připojili některého z jeho sousedů), podle ní se určí hešovací funkcí odpovídající spojový seznam a uzel, nyní už naformátovaný jako volný, se do něj vloží.

Expanze uzlu

Často je praktické roztáhnou velikost již alokovaného paměťového uzlu. Roztažení provádí metoda stretch() třídy GcHeap.

Pokud následkem omezení velikost na `GCHEAP_SLACK_LIMIT` zůstalo v uzlu dostatečné volné místo, je roztažení hotovo.

V opačném případě se zkontroluje, zda za roztahovaným uzlem je volný uzel dostatečné velikosti. Jestliže ano, provede se roztažení do něj (nepoužitý zbytek se případně odlomí a vytvoří se z něj volný uzel).

V nejhorším případě se alokuje nový paměťový uzel dostatečné velikosti a obsah starého se do něj překopíruje.

Smrštění uzlu

Další často používanou operací je zmenšení velikosti již alokovaného paměťového uzlu metodou shrink().

Provede se tedy změna velikosti a pokud je nepoužitý zbytek větší než `GCHEAP_SLACK_LIMIT`, vytvoří se z něj volný uzel.

Reentrantnost

Jak již bylo výše uvedeno, do haldy může přistupovat současně několik vláken a tak byly metody alloc(), free(), shrink() a stretch() implementovány jako reentrantní.

Kontrolní a ladící nástroje haldy

Třída GcHeap obsahuje rovněž metody vhodné pro ladění a kontrolu haldy. Zkontrolovat paměť lze metodou check(), nastavit oblasti volné paměti na konstantní hodnotu funkcí fillFree(), zkontrolovat toto nastavení lze metodou checkFillFree(), zkontrolovat uzel checkChunk() a procházet haldu od uzlu k uzlu metodou walker().

Nejdůležitější funkcí prověřující haldu je metoda check(). Metoda prochází paměť a kontroluje každý blok, jeho značky, ukazatele do struktury free list, velikost a další důležité parametry, které mohou být kritické. V případě detekce některé z výše uvedených chyb se volitelně uloží paměťový

obraz haldy funkcí dump(), dále JVM Lógr zašle sama sobě signál SIGSEGV (segmentation fault), v jehož důsledku dojde k ukončení programu a uložení paměťového obrazu procesu, tj. vytvoření souboru core.

Třída GcHeap

Jak bylo již několikrát zmíněno, jádro správy haldy je implementováno ve třídě GcHeap. Zde je přehled některých položek a metod:

void *alloc(int)
: alokace v paměťového uzlu

void shrink(void *, int)
: smrštění uzlu

void *stretch(void *, int)
: expanze uzlu

void free(void *)
: dealokace uzlu

bool isPtr(void *)
: kontrola ukazatele na volný uzel

bool isFreeChunk(void *)
: kontrola ukazatele na alokovaný uzel

bool isFullChunk(void *)
: je paměťový uzel obsazen?

void dumpChunk(void *)
: výpis obrazu uzlu

void dump(int)
: výpis paměťového obrazu haldy buď do souboru nebo na stderr

bool checkChunk(void *)
: kontrola uzlu

bool check()
: kontrola haldy

void info()
: výpis statistiky

void walker()
výpis průchodu haldou po uzlech

void listingFL()
výpis obsahu kompletní struktury free list

int coreLeft()
počet volných bajtů v haldě.

.Eas	seznam efemérních oblastí
.EAsCount	počet efemérních oblastí
.heapSizeSI	velikost haldy - součet velikostí efemérních oblastí včetně servisní informace
.allocatedBytesSI	počet alokovaných bajtů včetně servisní informace
.EAIgnoredBits	počet bitů ignorovaných při výpočtu indexu ve struktuře free list. Udává kolik horních intervalů je zablokováno.
.lastUsedInterval	index posledního použitého intervalu ve struktuře free list
.maxAllocatableChunk	maximální velikost alokovatelného uzlu včetně servisní informace
.heapStretchShield	zámek použitý při implementaci reentrantní metody stretch()

Programátorský interface LH

V předchozím textu byla popsána vnitřní implementace LH. Tato kapitola bude věnována funkcím horní vrstvy, tedy těm, které používá programátor pracující s LH. Jak již bylo uvedeno, deklarace struktur, symbolických konstant a funkcí lze nalézt v Logr/include/referencetypes.h, Logr/heap/reference.h a Logr/heap/reference.cc.
Halda je vytvořena a inicializována při startu JVM funkcí

```
int referenceInitialize(void)
```

Funkce vytvoří iniciální efemérní oblast Lógr haldy, základ servisní struktury referencí, inicializuje nulové reference (nullStatRef a nullInstRef - obdoba NULL), podle obsahu příkazové řádky případně spustí vlákno asynchronní garbage collection. Volitelně je inicializován ladící systém a systémová halda.
Úklid při ukončení běhu JVM provádí

```
int referenceCleanup(void)
```

Pokud je aktivní asynchronní garbage collection, pošle vláknu démona signál. Stejně proběhne zastavení ukončovacího vlákna. Uvolní se prostředky, které drží Lógr halda (dealokace efemérních oblastí apod.).
Horní vrstva obsahuje tyto funkce pro práci s haldou:

void *logrAlloc(int),void logrFree(void *)
: v případě, že se používá vlastní implementace systémové non Java haldy, používají se tyto funkce místo malloc() a free().

void referenceHeapDump(int)
: výpis paměťového obrazu haldy do souboru

void referenceHeapInfo(int)
: výpis statistiky

void referenceHeapWalker(int)
: výpis průchodu haldou po uzlech

int referenceHeapCheck(int)
: kontrola integrity haldy

int referenceHeapCoreLeft(int)
: počet volných bajtů na haldě

int referenceHeapCheckChunk(int,void *)
: kontrola uzlu

void referenceHeapDumpChunk(int,void *)
: výpis paměťového obrazu uzlu do souboru

A tyto funkce slouží pro práci s referencemi typu StatRef:

StatRef *newStatRef(int)
: získání nové StatRef reference

StatRef *newNamedStatRef(int, char *)
: získání nové pojmenované StatRef reference (používá se při ladění)

void setStatRefName(StatRef *, char *)
: nastavení jména reference

char *getStatRefName(StatRef *)
: získání jména reference

void deleteStatRef(StatRef *)
dealokace reference typu StatRef

void setStatRefType(StatRef *,int)
nastavení typu reference

void stretchStatRef(StatRef *, int)
roztažení uzlu v LH

void shrinkStatRef(StatRef *, int)
zmenšení uzlu v LH

void reallocStatRef(StatRef *, int)
změna velikosti uzlu v LH. Pokud je funkce volána s ukazatelem typu NULL, je ekvivalentní volání newStatRef(). Jestliže je volána s parametrem udávajícím velikost rovným 0, je ekvivalentní volání deleteStatRef().

void addStatRef(STAT_REF)
zvýšení počtu referencí

releaseStatRef(STAT_REF)
snížení počtu referencí

lockStatRef(STAT_REF)
zamčení proti přesunu

unlockStatRef(STAT_REF)
odemčení zámku bránícího přesunu

gcLockStatRef(STAT_REF)
zamčení provedené GC

gcUnlockStatRef(STAT_REF)
odemčení provedené GC

Používání počtu referencí, zámku proti přesunu a GC zámku je v kapitole 4.5. Funkce pro práci s referencemi typu InstRef jsou analogické. InstRef nemají funkci void setStatRefType(StatRef *,int), protože se dále nerozlišují. Navíc jsou implementovány následující funkce pro práci s instančními položkami:

InstRef *getRefField(InstRef **)
používá se pro natažení reference do lokální proměnné na zásobníku. Zvyšuje .refCount.

void putRefField(InstRef **, InstRef *)
provádí uložení reference v instanci

Podrobný popis předchozích dvou funkcí je v kapitole 4.5. Při ladění se používají funkce:

referenceNamedDebug(ACTION, NAME)
řízení ladícího systému se specifikací jména

referenceDebug(ACTION)
úprava ladícího systému.

Parametr action udává akci, kterou má ladící systém provést. Používá se disjunkce konstant deklarovaných v Logr/heap/reference.h. Nedílnou součástí správy haldy je garbage collector. Jeho implementace a řízení je popsáno v kapitole 4.5.

4.2 Reprezentace třídy v paměti

Každá třída nebo rozhraní je po překladu ze zdrojového kódu uložena v jednom .class souboru. Ten se v JVM Lógr překládá do souboru .logr, který je později zaveden do paměti (konkrétně do LH). V této kapitole bude popsáno, jak je rozhraní nebo třída v LH uložena. V další kapitole pak bude probrán způsob uložení v .logr souboru a proces zavedení do paměti.

Třída je v paměti uložena v nejjednodušším případě ve dvou datových strukturách: FixedData a RuntimeData. Struktura FixedData obsahuje především kód a identifikátory a je přímo načítána z .logr souboru. Provádějí se jen základní relokace v kódu. Díky tomu je možné ji při nedostatku paměti zrušit a později načíst znovu bez větších časových ztrát. Struktura RuntimeData je naproti tomu stavěna ze struktur uložených v .logr souboru a z paměťových struktur nadtřídy. Proto se její zahazování při nedostatku paměti nevyplatí.

Všechny struktury jsou deklarovány v souboru Logr/include/heapobjects.h.

4.2.1 O strukturách obecně

Ve strukturách se nikde nevyskytují ukazatele ve smyslu jazyka C. Vždy jde buď o reference typu StatRef nebo InstRef nebo o relativní pozice. Ty jsou ve většině případů počítány od začátku struktur, jen v některých případech relativně k adrese výskytu odkazu. Kromě toho jsou v některých

strukturách setříděné seznamy nebo stromy. Stromy jsou uloženy ve tvaru haldy (Heap-Shaped Trees), tzn. jsou plněny zleva doprava a shora dolů. Pro jejich procházení lze jednoduše používat pole. Funkce pro stavbu takovýchto stromů ze setříděných seznamů jsou uvedeny v Logr/util/hsbtree.c, prototypy v Logr/include/hsbtree.h.

FixedData

Na začátku této struktury je uložena její hlavička (FixedDataHeader). Ta ukazuje na některé další části. Na ostatní části vedou odkazy ze struktury RuntimeData (vyplňuje se podle .logr souboru). Pro rychlejší přístupy do struktur NamePool a ReferencePool musí mít hlavička velikost rovnou násobku velikosti struktury NamePoolEntry.

Hned za hlavičkou následuje pole struktur NamePoolEntry. Ty jsou spolu s polem struktur ReferencePoolEntry ze struktury RuntimeData používány pro volání metod a přístup k položkám (viz kapitola 5.8). Umístění hned za hlavičkou je důležité, protože struktura ReferencePool je též uložena hned za hlavičkou struktury RuntimeData a lze tak lehce přepočítávat ukazatele do obou struktur. Struktura NamePoolEntry obsahuje pouze relativní adresu identifikátoru cílové metody nebo položky (LogrStringMemberRef). Tato adresa je relativní vzhledem k adrese příslušné struktury NamePoolEntry a míří též do struktury FixedData.

Z hlavičky FixedDataHeader vede odkaz namePoolBitmap (relativní adresa) na pole bajtů, které reprezentuje typy záznamů v poli struktur NamePoolEntry (a tedy i ReferencePoolEntry). Každý záznam může být jednoho ze sedmi typů: odkaz na metodu, statickou metodu, metodu rozhraní, instanční položku, statickou položku, jméno třídy (využívá se pro funkce na alokaci objektů) a native metodu (jméno JNI metody ve sdílené knihovně). Záznamy jsou do bajtů ukládány od spodní poloviny. Délka pole struktur NamePoolEntry je uvedena v hlavičce FixedDataHeader v záznamu namePoolCount.

Ve struktuře FixedData je dále uložen kód všech metod dané třídy. Na něj jsou vedeny odkazy nejprve z .logr souboru a po jeho zavedení ze struktury RuntimeData. Jedinou výjimkou je metoda `<clinit>` (inicializátor třídy), na niž vede odkaz přímo z hlavičky FixedDataHeader (clinitOffset). Při zavádění kódu je třeba provádět relokace volání funkcí JVM, trampolín apod. (viz dále o zavádění tříd do paměti, .logr soubor). Před každou metodou je uvedena relativní pozice seznamu výjimek, které daná metoda může vyhazovat (ThrowsList). V tomto seznamu je uveden počet těchto výjimek (nula pokud žádné nevyhazuje) a seznam relativních pozic struktur Logr-

StringClassName, které identifikují typy výjimek. Tento seznam je uveden pouze pro účely Java Core Reflection, protože pro samotný běh JVM není potřeba.

V hlavičce FixedDataHeader je též uvedena relativní adresa stromu výjimek (exceptionTreeOffset) a jeho velikost (počet uzlů, exceptionTreeNodes). Každý uzel odpovídá rozsahem adres jedné metodě. Při hledání obsluhy výjimky se tak nejprve tímto stromem najde jméno metody, ve které výjimka nastala, a pak se pro danou metodu hledá obsluha. Tím se vlastně zrychlí vyhledávání bloků, kde výjimka nastala. Uzly stromu jsou typu ExceptionTreeEntry, který obsahuje relativní adresu metody, její délku, odkaz na její jméno (LogrStringMemberDef), počet jejích try bloků a odkaz na jejich seznam. Celý strom je postaven (utříděn) podle adresy metody (methodOffset).

Seznam try bloků se skládá z pole struktur typu ExceptionInfo, které obsahují vždy počáteční a koncovou pozici bloku, pozici obsluhy výjimky a typ výjimky, kterou daná obsluha odchytává (odkaz na identifikátor třídy, LogrStringClassName).

Další částí struktury FixedData jsou seznamy konstruktorů, privátních metod (statických i instančních), privátních položek (opět statických i instančních) a privátních konstruktorů. Pro všechny tyto seznamy jsou v hlavičce FixedDataHeader uvedeny počty a relativní pozice (pokud je počet nula, seznam chybí). Všechny seznamy jsou vlastně pole se záznamy typu MemberListEntry. Každý záznam obsahuje relativní pozici identifikátoru metody, položky či konstruktoru (LogrStringMemberDef) a vlastní relativní pozici příslušné metody, položky či konstruktoru. Pro metody a konstruktory tato pozice udává pozici kódu a pro položky to je relativní pozice v 'přidaných položkách'. Při vyhledávání je pak třeba k těmto pozicím ještě přičíst relativní pozici přidaných položek uvnitř struktury FixedData nebo uvnitř instance. Pro rychlejší vyhledávání jsou jednotlivé seznamy setříděny podle identifikátorů.

Poslední velkou částí struktury FixedData jsou seznamy různých identifikátorů. Pro reprezentaci různých typů identifikátorů se používají tři typy struktur.

Typ LogrStringClassName je určen pro názvy tříd. Kromě jména třídy uloženého jako řetězec UTF-8 znaků ukončený nulou obsahuje též hešovací kód tohoto řetězce a počet znaků jména po rozbalení do Unicode. Hešovací kód se používá pro rychlé vyhledávání v tabulkách zavaděče tříd (viz kapitola 4.4) a počet znaků pro rychlejší vytváření instancí třídy java.lang.String s daným řetězcem.

Dalším typem je LogrStringMemberDef používaný pro záznam jmen me-

tod, konstruktorů a položek dané třídy. Ve struktuře je uložen opět název (v Lógr JVM je spojen název a signatura) a počet znaků v Unicode. Kromě toho jsou zde uvedena přístupová práva (access flags) k danému 'objektu' a index do Unicode řetězce, který udává pozici signatury. To je zde kvůli vytváření záznamů pro Java Core Reflection, protože v některých případech nelze najít hranici mezi názvem a signaturou.

Posledním typem je LogrStringMemberRef sloužící k odkazům na metody, konstruktory a položky (i v jiných třídách). Ten je používán ve strukturách NamePoolEntry. Tento záznam obsahuje opět název spojený se signaturou v UTF-8. Dále obsahuje hešovací kód tohoto řetězce pro rychlé hledání metod, konstruktorů a položek uvnitř tříd. Kromě toho je též potřeba nějak identifikovat cílovou třídu (kde se má identifikátor hledat). K tomu slouží odkaz na LogrStringClassName. Tento odkaz je počítán relativně k příslušnému záznamu LogrStringMemberRef, ne od začátku struktury FixedData.

Pro jednodušší přístup k typům LogrString jsou k dispozici tři funkce, které jako parametr dostávají ukazatel (obvykle na strukturu FixedData) a relativní pozici a vracejí ukazatel na odpovídající typ. Jde o funkce:

```
LogrStringClassName *getLogrStringClassName(void *fixedData,
                                            int offset)
LogrStringMemberDef *getLogrStringMemberDef(void *fixedData,
                                            int offset)
LogrStringMemberRef *getLogrStringMemberRef(void *fixedData,
                                            int offset)
```

RuntimeData

Tato struktura představuje hlavní „rozcestník“ každé třídy (nebo rozhraní). Je stavěna při zavádění třídy z údajů v .logr souboru a některé údaje se vyplňují i během dalšího „života“ třídy. Každá reference na třídu uvnitř Lógr JVM ukazuje právě na tuto strukturu (například instance, v zásobníku ukazatel na třídu, jejíž kód se právě provádí, apod.).

Na začátku je opět hlavička, struktura RuntimeDataHeader. Ta musí mít (ze stejných důvodů jako hlavička FixedDataHeader) velikost rovnou násobku velikosti struktury [1] NamePool. RuntimeDataHeader obsahuje kromě jiných záznamů (budou postupně probrány) i přístupová práva třídy, referenci na strukturu FixedData třídy, odkaz do této struktury na jméno třídy (LogrStringClassName) a další údaje ve formě bitů v položce .flags. Tyto

[1] ??? 1. ReferencePoolEntry, 2. je to vůbec podezřelý s těmi přepočty NamePool-ReferencePool

údaje jsou reprezentovány konstantami `RDF_INITIALIZED` a `RDF_LOADED`. První z nich říká, že třída byla již inicializována (byl zavolán statický inicializátor `<clinit>`) a druhý, že třída je již korektně zavedena do paměti.

Dále je v hlavičce uvedena reference na zavaděč tříd (tedy přímo na jeho hešovací tabulku (viz kapitola 4.4)).

classLoaderRef a reference na instanci třídy java.lang.Class pro danou třídu v položce javaLangClassRef (pokud neexistuje, je zde nullInstRef).

Pokud třída reprezentuje pole, je v hlavičce v položce componentTypeRef uvedena reference na třídu, která reprezentuje elementy tohoto pole (může to být opět pole). Pokud jde o pole primitivních typů, je v této položce speciální hodnota `T_BOOLEAN`, `T_BYTE`, `T_CHAR`, `T_INT`, `T_SHORT`, `T_LONG`, `T_FLOAT` nebo `T_DOUBLE`. Tyto hodnoty jsou definovány v Logr/include/lconst.h. Pro test, zda jde o pole primitivních typů, slouží makro `IS_PRIMITIVE_TYPE` z téhož souboru. Pokud o pole nejde, je tato položka inicializována na nullStatRef.

Pro účely inicializace třídy je v hlavičce uložen identifikátor vlákna, které třídu inicializuje.

Hned za hlavičkou struktury RuntimeData je uloženo pole struktur ReferencePoolEntry používaných k volání metod a k přístupům na položky. Toto pole musí co do pořadí položek odpovídat poli struktur NamePoolEntry ve struktuře FixedData. Pak je možné s pomocí funkce getNamePoolEntry získat z adresy struktur RuntimeData, FixedData a ReferencePoolEntry odpovídající strukturu NamePoolEntry.

Pro třídy je ve struktuře RuntimeData několik záznamů o nadtřídách. V první řadě je v hlavičce RuntimeDataHeader uvedena reference na přímou nadtřídu (directSuperClassRef). Ta je pro java.lang.Object nebo jakékoli rozhraní nastavena na nullStatRef. Kromě toho je ještě vytvořen strom ze všech předků, který se používá pro rychlou implementaci testů přetypování (checkcast, instanceof). Jde opět o strom ve tvaru haldy setříděný podle referencí (uzly jsou typu SuperClassTreeEntry, obsahují pouze referenci na třídu). Odkaz na strom je v hlavičce RuntimeDataHeader, položka superClasses, velikost je v položce superClassCount.

Dále je v hlavičce RuntimeDataHeader odkaz na strom všech implementovaných (pro rozhraní zděděných) rozhraní (interfaces, interfaceCount) a přímo implementovaných (zděděných) rozhraní (directInterfaces, directInterfaceCount). Přímo implementovaná rozhraní jsou ta rozhraní, která jsou přímo uvedena v class souboru (tedy i .logr souboru). Všechna implementovaná rozhraní jsou pak všichni předkové těchto rozhraní a všechna rozhraní, která implementuje nadtřída.

Seznam přímo implementovaných rozhraní je vystavěn jako pole struk-

tur DirectInterfaceListEntry, který obsahuje pouze referenci na rozhraní a je podle ní také utříděn. Tento seznam je použit pouze pro Java Core Reflection, v Lógr JVM se jinak nevyužívá.

Naproti tomu strom (ve tvaru haldy) všech rozhraní, sestavený ze struktur InterfaceTreeEntry, se využívá pro testy přetypování (checkcast, instanceof) a pro zaznamenávání odkazů na IMT. Každý záznam obsahuje referenci jednoho rozhraní (třídící klíč) a referenci odpovídající tabulky IMT. Tato tabulka se vystaví jen v případě, že nad třídou byla vyvolána metoda daného rozhraní. Jinak je v této referenci hodnota NULL (to platí i pokud jde o rozhraní, protože to nemá IMT).

Dále jsou ve struktuře RuntimeData uloženy statické položky třídy (včetně finálních, tedy i ty u rozhraní). Jejich pořadí určuje překladač bajtového kódu s tím, že nejdříve jsou uvedeny primitivní typy a pak reference. Kromě toho jsou ve statických položkách též odkazy (reference) na statické řetězce (řetězce přímo uvedené ve zdrojovém kódu jazyka Java). Tyto řetězce ovšem nemají žádné názvy položek, dá se na ně přistupovat pouze přímo z kódu (jsou vytvářeny při zavádění třídy). Odkaz na statické položky (staticDataOffset) a jejich velikost (staticDataCount) jsou uvedeny v hlavičce RuntimeDataHeader. Kromě toho je zde ještě odkaz na bitovou mapu (staticDataBitmap) popisující typy položek. Každý bit odpovídá jednomu záznamu typu word (zdvojený word tedy používá dva bity), kde hodnota 1 znamená referenci a hodnota 0 primitivní typ. Hodnoty jsou zapisovány od spodních bitů a délka bitové mapy je zaokrouhlena na nejbližší násobek čtyř bajtů.

Pro rychlejší vytváření instancí je ve struktuře RuntimeData též obraz celé inicializované instance dané třídy. Díky tomu se při vytváření instance nemusí procházet bitová mapa instance (též ve struktuře RuntimeData) a nastavovat jednotlivé záznamy na nulu či nullInstRef. Pořadí záznamů v instanci je opět určeno překladačem s tím, že nejdříve jsou primitivní typy a pak reference. Odkazy jsou v hlavičce RuntimeDataHeader. Na obraz instance vede odkaz instanceDataOffset, na bitovou mapu instanceDataBitmap (stejný systém jako u statických položek), velikost (počet položek) je v instanceDataCount. Navíc je v hlavičce ještě relativní pozice začátku přidaných instančních položek od začátku instance (posun kvůli položkám předků a hlavičce instance)(instanceDataDelta).

Poslední velkou částí struktury RuntimeData jsou tabulky neprivátních metod a položek (spolu s VMT). Tyto tabulky jsou určeny pro vyhledávání metod a položek ve třídě. Jsou celkem čtyři podle typu záznamů, které obsahují. To je proto, že již při překladu lze rozlišit volání statické a virtuální metody a přístup na statickou či instanční položku. Díky tomu je pak vy-

hledávání rychlejší, protože se hledá jen v jedné (správné) tabulce. Všechny tabulky jsou organizovány jako hešovací tabulka (Logr/include/hash.h), kde každý záznam vede (relativní pozice) na strukturu ColisionList. Ta obecně obsahuje pouze počet položek, za nímž následuje pole těchto položek. Položky vždy nějakým způsobem odkazují na název a signaturu (jsou podle něj též setříděny). Typy v polí se ovšem již liší podle typu tabulky (podle toho, pro co je určena). Prázdné kolizní seznamy (tj. seznamy s nulovým počtem položek) jsou sdíleny (je jen jeden). Na všechny tabulky vedou odkazy ze struktury RuntimeDataHeader. Pokud nějaká tabulka ve třídě neexistuje, je tento odkaz nulový.

Nejsložitější ze všech těchto struktur je MethodListEntry (odkaz methodTable), protože ta obsahuje pouze odkaz do VMT a odkaz do struktury FixedData na LogrStringMemberDef (jméno a signatura). Odkaz na třídu, která obsahuje jméno, je ale uveden ve VMT. Z tohoto důvodu je přístup na tento řetězec poněkud pomalejší (přes VMT). VMT (virtual method table) je pole struktur VmtEntry, které obsahují referenci na třídu (její strukturu RuntimeData) obsahující kód metody a relativní adresu této metody v odpovídající struktuře FixedData. Odkaz na VMT vede opět z hlavičky RuntimeDataHeader (položka vmt) a její velikost je uvedena tamtéž (virtualMethodCount). Zde je použit trik, že odkaz vmt neukazuje přímo na první položku VMT, ale o velikost jednoho záznamu (sizeof (VmtEntry)) před tabulku. To je proto, že pokud odkaz ze struktury ReferencePoolEntry má hodnotu nula, jde o nevyplněný záznam. Proto jsou všechny odkazy do VMT takto posunuté.

Ostatní tabulky mají víceméně stejný obsah. Vždy je v nich uvedena reference, ve které je název a signatura (LogrStringMemberDef) dané položky či statické metody, a relativní adresa názvu v odpovídající struktuře FixedData. Pro statickou metodu tato třída obsahuje též kód metody a v tabulce (záznam StaticMethodListEntry) je uvedena jeho relativní adresa. Na hešovací tabulku statických metod vede odkaz staticMethodTable z hlavičky RuntimeDataHeader. Statické položky (odkaz vede ze struktury StaticFieldListEntry, odkaz na tabulku je staticFieldTable) jsou umístěny přímo ve struktuře RuntimeData cílové třídy a ve struktuře StaticFieldListEntry je vždy uvedena jejich relativní adresa. Pro instanční položky se odkaz na třídu používá pouze pro určení třídy s názvem a signaturou metody a ve struktuře FieldListEntry (odkaz fieldTable) je uvedena relativní adresa položky v instanci. Poslední tabulkou je seznam metod rozhraní (InterfaceMethodListEntry). Tato tabulka se vyskytuje u rozhraní, u nichž se ale zase nevyskytuje tabulka virtuálních metod. Proto se pro odkaz na ni používá methodTable a počet metod je uveden ve virtualMethodCount (používá se

při vytváření IMT pro třídu). Záznam InterfaceMethodListEntry opět obsahuje odkaz tentokrát na rozhraní, ve kterém je LogrStringMemberDef se jménem a signaturou metody. Dále je tam relativní pozice tohoto jména a relativní pozice metody v IMT pro rozhraní.

Posledním záznamem v hlavičce RuntimeDataHeader je nameListSize. To je celková velikost všech kolizních seznamů všech tabulek. Je užíván pouze pro rychlejší určení prostoru, který třeba alokovat při zavádění třídy.

Tabulka metod rozhraní (Interface Method Table - IMT)

Odkaz na tuto tabulku vede ze struktury RuntimeData třídy. V ní je uvedeno 'mapování' metod rozhraní na virtuální metody třídy. Jde vlastně pouze o pole struktur typu ImtEntry, které obsahují relativní pozici záznamu pro metodu ve VMT třídy. Díky tomu mohou být tyto tabulky sdíleny. Více informací je v kapitole 5.8.

Instance

Každá instance má na svém začátku hlavičku definovanou strukturou InstanceHeader. Ta obsahuje pouze odkaz na třídu (strukturu RuntimeData), ke které instance patří. Hned za hlavičkou následují instanční položky dané třídy.

Překladač si pro každou třídu vygeneruje pořadí statických a instančních položek a pak se již odkazuje jen pomocí relativních adres od jejich začátku. Přidané položky jsou právě ty položky, které daná třída přidává ke zděděným.

Podle specifikace jazyka Java je sice rozhraní potomkem třídy java.lang.Object, ale to se projeví pouze v testech přetypování a v Java Core Reflection. Proto je v Lógr JVM toto chování implementováno přímo až v uvedených místech.

Pole

Z pohledu JVM je pole instancí zvláštního objektu, který je přímým potomkem třídy java.lang.Object, implementuje rozhraní Cloneable a Serializable a má navíc skrytou public final položku length (udržující informaci o počtu prvků v poli uložených) a přetíženou metodu Clone, která se chová stejně jako metoda Clone třídy Object, pouze místo výjimky CloneNotSupportedException vyhazuje výjimku InternalError, kterou není třeba programátorsky ošetřovat (překladač programátora nenutí na výjimku reagovat), a

tak je možno metodu Clone nad polem volat pohodlně. Položka length je dostupná pouze instrukcí bajtového kódu arraylength.

V JVM Lógr bylo možno řešit stavbu struktury RuntimeData pro pole několika způsoby:

1. „ručně“ - nejpracnější, avšak nejrychlejší varianta. V případě změny třídy Object nebo změny definice pole (např. nová norma) by však bylo nutno přepisovat značnou část kódu. Stavba struktury RuntimeData pro pole by navíc byla téměř totožná se stavbou struktury RuntimeData pro třídu. Značná redundance kódu.

2. „ručně“ s využitím struktury RuntimeData třídy Object - stále značná redundance kódu, neboť taková stavba je obdobná jako stavba struktury RuntimeData pro jakoukoli podtřídy třídy Object.

3. vytvořit falešný Lógr soubor popisující pole. Stavba struktury RuntimeData je pak totožná se stavbou jakékoli třídy (funkce loadLogrFile()). Odlišnosti pole oproti třídě lze následně vyřešit jednoduchým poopravením struktury RuntimeData.

Z důvodů elegantního a jednoduchého řešení jsme zvolili třetí možnost. Všechna pole mají společný vzorový .logr soubor, z něhož jednoduše vznikají struktury RuntimeData pro jednotlivé typy polí. Vzorový .logr soubor popisuje falešnou třídu se jménem `'['`. Toto jméno je pro konkrétní typ pole nahrazeno příslušným jménem pole, např. `'[[I'` (dvourozměrné pole datových typů int).

JVM Lógr si v hešovací tabulce udržuje všechny dosud vytvořené typy polí a při požadavku na strukturu RuntimeData pro pole nejdříve zjistí, zda taková struktura již v hešovací tabulce není. Pokud ano, vrátí ji. V případě, že v tabulce není, vytvoří vzorovou strukturu RuntimeData (funkce loadLogrFile), jíž upraví tak, aby popisovala pole, a upravenou strukturu poté zařadí do hešovací tabulky a vrátí žadateli.

Příklad:
Uložení třídy A v LH může vypadat například takto:

4.3 Soubor .logr

Soubory .logr jsou generovány překladačem bajtového kódu z .class souboru třídy. Při zavádění třídy je soubor .class načten a jsou z něj vytvořeny struktury RuntimeData a FixedData.

Při čtení následujícího textu je vhodné zároveň sledovat zdrojový kód se strukturou souboru .logr, který je uložen v Logr/include/logrfile.h.

4.3.1 Formát souboru .logr

Soubor .logr je uvozen hlavičkou LogrFileHeader. Hlavička na svém začátku obsahuje identifikátor .logr souboru (řetězec 'LOGR'), číslo verze, číslo překladu JVM (pro účely relokací), CRC-32 kód a délku zdrojového class souboru (použito pro test, zda soubor .logr odpovídá souboru .class), přístupová práva k třídě, relativní adresu jména třídy a jména předka (nula pokud třída nemá předka) uvnitř struktury FixedData (cíl je typu LogrFileClassName). Za hlavičkou je uložen obsah struktury FixedData dané třídy (velikost je uvedena v hlavičce souboru). Tato struktura je přímo načítána do paměti.

Za strukturou FixedData následuje seznam implementovaných či zděděných rozhraní. Jejich počet je uveden v hlavičce souboru (interfaceCount). Každé rozhraní je reprezentováno jednou strukturou LogrFileInterfaceInfo, která obsahuje pouze odkaz do struktury FixedData na strukturu LogrFileClassName obsahující název rozhraní.

Dále je uložen obraz primitivních statických položek. Ten má velikost primitiveStaticDataCount záznamů typu word. Kromě toho jsou v hlavičce souboru uvedeny počty referencí a statických řetězců uložených ve statických datech. Za obrazem primitivních typů leží bitová mapa statických dat. Každý bit je určen pro jeden záznam typu word ($0 = primitivntyp$, $1 = reference$), ukládání probíhá od spodních bitů, velikost je zaokrouhlena na hranici čtyř bajtů.

Za bitovou mapou statických dat následuje podobná konstrukce pro instanci. Nejdříve je uveden obraz primitivních typů instance (velikost primitiveInstanceDataCount) a za ním následuje bitová mapa instance. Navíc je v instanci ještě referenceInstanceDataCount záznamů typu reference.

Dále v souboru následují seznamy virtuálních metod nebo metod rozhraní (LogrFileMethodInfo), statických metod (LogrFileStaticMethodInfo), instančních položek (LogrFileFieldInfo) a statických položek (LogrFileStaticFieldInfo). Každý záznam obsahuje hašovací kód jména a signatury, odkaz na jméno a signaturu (LogrStringMemberDef) do struktury FixedData a relativní pozici kódu metody nebo relativní pozici položky uvnitř přidaných položek. Každý z těchto seznamů je setříděn nejdříve podle hešovacího kódu a pak podle jména a signatury. Počty záznamů v jednotlivých seznamech jsou uvedeny v hlavičce souboru.

Po seznamech metod a položek následuje v souboru seznam statických řetězců. Každý je reprezentován strukturou LogrFileStaticString, kde je uve-

den počet znaků v řetězci a pak následuje samotný řetězec zapsaný v Unicode (bez koncové nuly). Počet řetězců je zaznamenán v staticStringCount v hlavičce .logr souboru. Řetězce jsou po vytvoření přidávány do statických položek (za položky definované v class souboru, tj. za primitivní typy a reference) ve stejném pořadí, v jakém se vyskytují v tomto seznamu.

Poslední částí .logr souboru jsou různé relokační tabulky. Tabulky obsahují vždy odkaz do kódu, kde je třeba změnit hodnotu (relativní adresa ve struktuře FixedData), a většina z nich je ukončena záznamem s hodnotou -1.

První tabulkou je LogrFileMyThisRelocationEntry, která se používá pro relokace přímých přístupů na instanční položky třídy. Překladač do kódu umístí pouze relativní pozice v přidaných položkách a při načítání se tyto pozice posunou podle velikosti položek od předků.

Další tabulka je určena pro relokace přímých přístupů na položky předků . Tato tabulka je trochu zvláštní, protože obsahuje dva typy záznamů. Na první úrovni je vždy délka LogrStringMemberRef (typu int), který identifikuje cíl (odkaz na třídu se ignoruje, lze použít aktuální třídu), a za délkou ihned následuje tato struktura. Za ní pak následují záznamy typu LogrFileSuperThisRelocationEntry, které ukazují do kódu na místa přístupů na tyto položky. Tento seznam je ukončen záznamem s hodnotou -1, vnější seznam údajem o délce opět s hodnotou -1. Při relokaci se může stát, že cílová položka ve třídě neexistuje. V tom případě se přímo přepíše celá instrukce instrukcí na vyhození výjimky java.lang.NoSuchFieldError. Další tabulka se používá na relokaci přímých přístupů na statické položky (LogrFileStaticRelocationEntry). Použití je přesně stejné jako u LogrFileMyThisRelocationEntry. Každý záznam odkazuje na místo v kódu, kde je relativní pozice položky vzhledem k začátku statických dat (pořadí definuje překladač). K této hodnotě se pouze přičte relativní pozice statických dat ve struktuře RuntimeData. Seznam je opět ukončen záznamem s hodnotou -1.

Následující dvě tabulky slouží k relokaci přímých přístupů do polí, protože pole jsou potomky třídy java.lang.Object a při překladu není známa velikost instance této třídy (předchází data pole). Tabulka LogrFileArrayOffsetRelocationEntry slouží k relokaci přímých přístupů do pole (posunutí dat pole v instanci pole). Druhá tabulka (LogrFileArrayLengthRelocationEntry) je určena pro relokace přímého čtení délky pole. Ta je totiž uložena v instanci pole za položkami předka, tj. třídy java.lang.Object. Obě tabulky jsou opět ukončeny záznamem s hodnotou -1.

Poslední dvě tabulky jsou určeny pro relokace volání funkcí Lógr JVM z kódu a pro speciální referenci nullInstRef (implementace prázdné reference, null v jazyce Java). Obě se využijí pouze v případě, že číslo překladu JVM

je jiné než číslo uvedené v .logr souboru. V opačném případě není relokace nutná, protože adresy těchto objektů jsou stejné (na cílových adresách v kódu jsou adresy objektů platné pro dané číslo překladu JVM).

První tabulka (LogrFileStubRelocationEntry) je opět dvouúrovňová, kde na první úrovni je vždy číslo funkce (index do tabulky lkiStubList, soubor Logr/jvm/stublist.c) a za ním následuje seznam relativních pozic v kódu, kde se daná funkce volá. Obě úrovně jsou ukončeny záznamy s hodnotou -1.

Druhá tabulka (LogrFileNullRelocationEntry) slouží k relokacím 'konstanty' null jazyka Java. Záznam vždy obsahuje pozici v kódu, kde se používá. Seznam je ukončen hodnotou -1.

4.3.2 Načítání .logr souboru

Při načítání .logr souboru se nejdříve vytvoří struktura FixedData, která se zavede beze změn. Dále je nutné na základě zbytku souboru postavit strukturu RuntimeData. U této struktury je důležité nejdříve zjistit velikost, která je dána velikostí statických položek, obrazu instance (zjistí se podle velikosti instance předka a přidaných položek), bitových map statických a instančních dat a tabulek identifikátorů (zjišťuje se podle položky namePoolSize v hlavičce RuntimeDataHeader předka). Pro jednoduchost je tato velikost větší než skutečně potřebná, protože při stavbě tabulek identifikátorů pak některé z nich překrývají zděděné a tudíž se v tabulkách umisťují na jejich místa. Díky tomu se zmenší skutečně využité místo. Proto se po vytvoření struktury RuntimeData nevyužitý zbytek opět uvolní.

Pro stavbu tabulek se využívají jednotné funkce, kterým se jako parametr dodávají objekty (ve smyslu C++) a které umějí pracovat s jednotlivými typy identifikátorů (tj. zapisovat metody do VMT a do kolizního seznamu, položky do seznamu, atd.). Při vyplňování dochází u položek k posunům relativních pozic kvůli jejich umístění v instanci či struktuře RuntimeData. Při vyplňování se též testuje, zda třída může metody překrývat (na základě přístupových práv). Kromě toho je třeba otestovat, zda třída skutečně implementuje rozhraní, o kterých to tvrdí (tedy zda má skutečně všechny metody daného rozhraní). Tento test se rovnou využívá ke stavbě IMT pro dané rozhraní, protože postup testu a stavby je stejný. Pak je ještě nutné otestovat, zda neabstraktní třída nezdědila od svých předků nějaké abstraktní metody. V takovém případě musí být označena za abstraktní.

Během načítání se také musí vytvářet statické řetězce, tzn. instance třídy java.lang.String. S tím je jistý problém, protože například při načítání třídy java.lang.Object (předek java.lang.String) ještě není možné tyto instance

vytvářet. Proto se v tomto případě vytváří v paměti seznam takovýchto řetězců s odkazy do tříd, k nimž patří. Po načtení třídy java.lang.String se pak seznam projde a instance se vytvoří. Díky tomu může vzniknout problém, pokud se na statické řetězce přistupuje dříve, než vzniknou. V takovém případě je místo nich použita hodnota nullInstRef.

Poslední fází načtení souboru jsou relokace v kódu. Ty jsou ve většině případů triviální (přičtení hodnoty, výměna adresy). Jediná složitější relokace je přístup na zděděné položky, protože zde se položky musí nejprve vyhledávat.

To jsou ty položky, na které se přistupuje pomocí aktuální instance (this) a jméno položky se nevyskytuje ve třídě, pro kterou je .logr soubor vytvořen.

4.4 Zavaděče tříd

O tvorbě tříd na úrovni jazyka a JVM obecně se můžete dočíst ve [Spec97], v dokumentaci JDK - balík java.lang, třída ClassLoader, nebo přímo ve zdrojovém souboru třídy java.lang.ClassLoader.

Zavádění tříd v JVM Lógr vypadá následovně: zavaděč tříd je požádán o strukturu RuntimeData nějaké třídy T. Teď je třeba rozlišit, zda se má být třída zavedena vnitřním (v terminologii JDK 1.1 'default', v JDK 1.2 'bootstrap') nebo uživatelským (v JDK 1.1 'user', v JDK 1.2 'user' nebo 'system') zavaděčem tříd.

> Poznámka:
> struktura RuntimeData a instance třídy java.lang.Class jsou spolu spjaty a nesou stejnou informaci - popisují Java třídu.Struktura RuntimeData je vhodná pro použití JVM Lógr, zatímco instance třídy java.lang.Class je použita z 'vnějšího' - programátorského Java světa.

4.4.1 Zavádění vnitřním zavaděčem tříd

Vnitřní zavaděč tříd se nejprve snaží nalézt v hešovací tabulce strukturu RuntimeData požadované třídy T. Je-li úspěšný, struktura RuntimeData je vrácena. V opačném případě najde vnitřní zavaděč v lokálním souborovém systému .class soubor odpovídající požadované třídě a zjistí, zda byl tento soubor již přeložen do souboru .logr. Nebyl-li, bude přeložen nyní. Pak je .logr soubor zaveden funkcí loadLogrFile() (je vytvořena struktura RuntimeData) a tato vytvořená struktura je přidána do hešovací tabulky a vrácena.

Poznámka:
Funkce loadLogrFile() při stavbě struktury RuntimeData žádá o všechny předky zaváděné třídy. Z toho pramení jediné nebezpečí - zacyklení v případě, že třída je sama sobě předkem. Zavaděč tříd si tedy musí kromě úspěšně zavedených tříd pamatovat i třídy, které zavádět začal, ale dosud nezavedl. Funkce getRuntimeData() toto řeší přidáním zvláštního záznamu do hešovací tabulky. Je-li funkce getRuntimeData() opětovně požádána o zavedení třídy, která se již zavádí, skončí takový pokus výjimkou ClassCircularityError. Aby výjimkou ClassCircularityError neskončil i požadavek na zavedení třídy, kterou právě zavádí jiné vlákno výpočtu, smí s vnitřním zavaděčem tříd pracovat vždy pouze jedno vlákno. Po dobu zavádění třídy (a tedy i zavádění všech jejích předků) nějakým vláknem je vnitřní zavaděč tříd pro ostatní vlákna uzamčen. Ač by bylo možno implementovat řešení, které vláknům umožní k zavaděči tříd konkurenční přístup, popisované řešení se obejde bez složitějších testů a je rychlejší (pokud by se vlákna v zavaděči střídala, ztratí se mnoho času jen na jejich synchronizaci).

Algoritmus zavádění třídy T vnitřním zavaděčem tříd:

1. Zamykání (je třeba kvůli úpravám hešovací tabulky). Pokud je již vnitřní zavaděč zamknut aktuálním vláknem, jde se do bodu 2). Jinak se vnitřní zavaděč zamkne pomocí zámku knihovny pthread. Žádné jiné vlákno nemá teď do zavaděče přístup.

2. Prohledání hešovací tabulky. Obsahuje-li hešovací tabulka již strukturu RuntimeData třídy T, zjišťuje se, zda je tato struktura falešná nebo již hotová. Je-li falešná, znamená to, že třída je sama svým vlastním předkem, a tuto chybu je třeba vrátit jako výjimku ClassCircularityError. Je-li struktura RuntimeData již hotová, stačí ji vrátit.

3. Přidání falešné struktury RuntimeData. Hešovací tabulka strukturu RuntimeData třídy T dosud neobsahuje. Do tabulky je přidána falešná struktura RuntimeData třídy T (nese informaci, že třída T je zaváděna).

4. Stavba struktury RuntimeData. V lokálním souborovém systému je nalezen soubor .class odpovídající třídě T a je zavolána funkce registerClass() (viz dále), která vrací hotovou strukturu RuntimeData.

5. Oprava hešovací tabulky. Falešné struktura RuntimeData třídy T v hešovací tabulce je nahrazena hotovou strukturou RuntimeData. Nastala-li při stavbě struktury RuntimeData chyba, je falešná struktura pouze z tabulky vyjmuta.

6. Odemykání. Byl-li vnitřní zavaděč v bodě 1) zamknut, bude nyní odemknut.

   Poznámka:
   Je-li vnitřní zavaděč volán rekurzivně z funkce loadLogrFile(), zamykání (bod 1)) ani odmykání (bod 6)) se neprovádí, neboť zavaděč byl zamknut při prvním, nerekurzivním volání. To ovšem požaduje, aby funkce loadLogrFile() používala pro rekurzivní volání funkce getRuntimeData() stále stejné vlákno. (Všimněte si, že tato podmínka u uživatelského zavaděče tříd splněna není!)

4.4.2 Zavádění uživatelským zavaděčem tříd

Z důvodů větší rychlosti není požadavkem na zavedení třídy uživatelským zavaděčem obtěžován přímo Java kód (metody třídy java.lang.ClassLoader), ale mezi Java kód a JVM

Lógr je vložena hešovací tabulka, která si pamatuje všechny třídy uživatelským zavaděčem zavedené.

Pozor: v následujícím textu se proto termínem 'uživatelský zavaděč' nebudou rozumět metody třídy java.lang.ClassLoader, ale funkce pracující s hešovací tabulkou!

Uživatelský zavaděč pracuje podobně jako zavaděč vnitřní, navíc je tu však třeba počítat s tím, že zavádění se účastní i uživatelem psaný Java kód.

Uživatelský zavaděč tříd se stejně jako vnitřní zavaděč nejprve snaží nalézt v hešovací tabulce strukturu RuntimeData požadované třídy T.

Je-li úspěšný, struktura RuntimeData je vrácena. V opačném případě je volána Java metoda loadClass() instance třídy java.lang.ClassLoader, která vrací instanci třídy java.lang.Class.

Tato instance je svázána se strukturou RuntimeData, která se nyní přidá do hešovací tabulky (pokud se struktura RuntimeData třídy T nedostala do tabulky mezitím jinou cestou).

Poznámka:
Funkce loadLogrFile() při stavbě struktury RuntimeData žádá o všechny předky zaváděné třídy. Z toho pramení jisté nebezpečí

- zacyklení v případě, že třída je sama sobě předkem. Uživatelský zavaděč tříd si ale s tímto problémem poradit nemůže, neboť není zaručeno, že pro rekurzivní volání funkce getRuntimeData() bude použito stejné vlákno. Pokud by byl uživatelský zavaděč uzamčen podobně jako zavaděč vnitřní pouze při prvním, nerekurzivním volání a rekurzivní volání by učinilo jiné vlákno, zavádění uvázne (vznikne 'deadlock'). Je tedy třeba před každým voláním funkce loadLogrFile() zavaděč odmykat a při každém rekurzivním volání funkce getRuntimeData() zavaděč znovu zamykat. Z toho plyne, že uživatelský zavaděč nemůže rozpoznat cyklus v dědičnosti tříd (nemůže si pamatovat zaváděné, ale dosud nezavedené třídy, protože nepozná, zda zavádění nějaké třídy T bylo způsobeno zaváděním nějakého předka třídy T, nebo se jedná o nezávislý požadavek).

Algoritmus zavádění třídy T uživatelským zavaděčem tříd:

1. Zamykání (je třeba kvůli úpravám hešovací tabulky). Uživatelský zavaděč tříd je uzamčen (nikdo jiný do něj nemá přístup).

2. Prohledání hešovací tabulky. Obsahuje-li hešovací tabulka již strukturu RuntimeData třídy T, je tato struktura vrácena.

3. Odemykání. Uživatelský zavaděč tříd je odemčen (aby mohl být při rekurzivním volání znovu uzamčen).

4. Volání Java metody loadCLass třídy java.lang.ClassLoader. Metoda loadClass() vrací instanci třídy java.lang.Class, jež je svázána se strukturou RuntimeData, která bude zařazena do hešovací tabulky.

5. Zamykání. Uživatelský zavaděč tříd je opětovně uzamčen.

6. Zařazení struktury RuntimeData do hešovací tabulky. Neobsahuje-li ještě hešovací tabulka strukturu RuntimeData třídy T, je tato struktura do tabulky přidána (struktura RuntimeData se ale do tabulky již dostat mohla, neboť tabulka byla odemčena).

7. Odemykání. Uživatelský zavaděč tříd je odemčen.

   Poznámka: Třída se může do hešovací tabulky dostat také prostřednictvím funkce defineClass.

registerClass

Pomocná funkce registerClass() dostává soubor .class, jejž přeloží do souboru .logr (nebyl-li již přeložen), který se pak snaží zavést funkcí loadLogrFile(). Výsledkem je postavená struktura RuntimeData, pro níž se následně vytvoří instance třídy java.lang.Class.

> Poznámka: Na disku může být vedle sebe uloženo více různých souborů .logr obsahujících třídu se stejným jménem (každá třída může být zavedena jiným zavaděčem). Proto je jméno souboru .logr doplněno o CRC32 kontrolní součet souboru .class původní třídy, o délku původní třídy a o údaj, zda se jedná o třídu zavedenou vnitřním nebo uživatelským zavaděčem (je-li třída stejného jména, ale s jiným obsahem, zavedena zavaděčem vnitřním, může být stará verze třídy (souboru .logr) případně smazána; je-li ale třída zaváděna zavaděčem uživatelským, musí být původní soubor .logr ponechán, neboť může být dosud využíván jiným uživatelským zavaděčem).

4.5 Garbage collection

Jazyk Java obsahuje automatické slučování nedostupných oblastí paměti — garbage collection. Garbage collection předchází vzniku neplatných odkazů na ukazatele, visícím ukazatelům a dalším zhoubným chybám spojeným s dynamickou alokací a dealokací paměti.

Garbage collector (GC) v JVM obvykle provádí dvě činnosti. Jednak vyhledává a uvolňuje instance tříd, které již nejsou dostupné, a dále provádí destrukci nepoužitých tříd.

4.5.1 Implementace GC

V JVM Lógr je implementován systém garbage collection pracující inkrementálním obarvovacím algoritmem s exaktním sledováním ukazatelů.

Při startu JVM je vytvořena servisní struktura GC a v závislosti na parametrech příkazové řádky je spuštěn asynchronní GC. Servisní strukturou je třída GarbageCollector. Při asynchronním běhu GC je vytvořeno démon vlákno s nízkou prioritou, které provádí kolekci na pozadí. Garbage collector provádí většinu své práce v době, kdy se nic jiného neděje. Obvykle pracuje v době nečinnosti, kdy se čeká na vstup od uživatele ve formě stisků kláves nebo událostí myši. Jediný moment, kdy se garbage collector musí spustit, i

když právě probíhá nějaká činnost s vysokou prioritou (tedy jediný moment, kdy se skutečně zpomalí systém), je ten, kdy není dostatek paměti.

Synchronní spuštění GC lze provést například metodou java.lang.Runtime.gc(). Nedílnou součástí procesu garbage collection je ukončování objektů, které lze rovněž synchronně vynutit pomocí java.lang.Runtime.finalize().

GC samozřejmě nepracuje odděleně, ale spolupracuje s běhovým systémem virtuálního stroje.

Hlavní motivací pro zavedení dvou typů referencí (InstRef a StatRef) byla snaha o zjednodušení a zefektivnění práce garbage collectoru. Reference typu InstRef se totiž odkazují výhradně na instance Java tříd v LH. Kdežto StatRef reference na ostatní datové struktury (jako například RuntimeData, FixedData apod.).

Protože GC provádí rovněž ukončování objektů, obsahují struktury referencí (InstRef a StatRef) speciální položku příznaků .finalizerFlags. Tyto příznaky se používají pro uvolňování alokovaných objektů z interních struktur JVM a korektní volání ukončovačů instancí. Po provedení kolekce zpracuje GC objekty, které mají být uvolněny podle finalizerFlags příznaků. Před uvolněním paměťového objektu podle nastavených příznaků obvolá funkce, které toto uvolnění provedou. Speciálním případem je příznak `FINALIZER_JAVA`, který odpovídá metodě ukončovače jazyka Java. Dle specifikace může být ukončovač nad danou instancí zavolán nejvýše jednou. Po jeho prvním zavolání se vynuluje příznak `FINALIZER_JAVA` a objekt postoupí do dalšího kola kolekce. Je tomu tak, protože uvnitř ukončovače mohl být ukazatel this předám jinam. Tím mohly vzniknout nové reference na objekt, který byl původně nedostupný a měl být uvolněn. Tyto nové reference se dohledají v dalším kole. Jestliže nemá paměťový objekt příznak `FINALIZER_JAVA` nastaven, ukončovač se nevolá. Obvolají se ovšem funkce vyřazující jej z interních struktur JVM a paměť se uvolní. Tento postup je korektní neboť v této fázi již nemůže dojít ke vzniku nových referencí.

Dalšími položkami referencí, se kterými pracuje systém GC, jsou dva druhy zámků. První z nich je položka struktury InstRef resp. StatRef .lockCount. Kromě jiného tento zámek slouží k zamčení objektu v LH. Tímto zámkem se aplikační vlákno chrání před přesunem paměťového objektu v haldě - GC musí tento zámek respektovat. Naopak systém GC může pro svoji potřebu zamknout referenci pomocí zámku v položce .state a pak s oblastí paměti libovolně manipulovat.

Samozřejmostí je spolupráce s překladačem, který vytváří kód pro správné počítání referencí (položka .refCount). Položka .refCount struktur InstRef resp. StatRef informuje o počtu referencí ze zásobníku. .refCount se

používá pro určení množiny kořenových ukazatelů, ze kterých se při kolekci vychází (viz níže). Běhový systém JVM používá pro práci s referencemi v instancích dvě funkce:

```
InstRef *getRefField(InstRef **refPtr)
```

která provádí natažení reference do lokální proměnné a

```
void putRefField(InstRef **refPtr, InstRef *ref)
```

jenž referenci do instance ukládá. Aby nedocházelo k chybám, položky v instancích se krátkodobě zamykají. Zámek vláknu signalizuje, že se s položkou pracuje, a zároveň zaručuje, že příchozí vlákno nemůže do práce s instancí žádným způsobem zasahovat.

4.5.2 Garbage collection instancí

GC obarvuje všechny paměťové objekty dosažitelné z množiny kořenových ukazatelů. Jak již bylo výše zmíněno, pracuje GC v této fázi pouze s referencemi typu InstRef. Množinu kořenových ukazatelů potom tvoří ty reference typu InstRef, které mají položku .refCount větší než nula (existuje na ně tedy odkaz ze zásobníku). Algoritmus pracuje s těmito množinami referencí typu InstRef (dále jen reference):

R
: množina kořenových referencí

Dále jsou reference obarvovány následujícími čtyřmi barvami:

zelená
: reference není dostupná z množiny kořenových referencí.

oranžová
: reference je dostupná, ale instance, na níž se odkazuje, nebyla dosud zcela zpracována.

červená
: reference je dostupná z množiny R a instance, na níž se odkazuje, byla již zpracována.

modrá
: reference je 'dirty'. Je to reference, která v průběhu kolekce vznikla nebo byla pozměněna mutátorem. Modrá reference vzniká přebarvením reference zelené.

Mutátor je aplikační vlákno, které zasahuje do datových struktur v LH a kolekci systému GC „kazí“. Jak bylo již uvedeno, takovým referencím se přiřadí modrá barva. Jak již bylo uvedeno

```
InstRef *getRefField(InstRef **refPtr)
```

je funkce, která provádí natažení reference do lokální proměnné. Jejím použitím mohou tedy vznikat nové odkazy na reference, které nebyly dosud pod kontrolou - nové kořenové reference. A funkce

```
void putRefField(InstRef **refPtr, InstRef *ref)
```

provádí uložení reference v instanci. Reference, která byla v instanci nahrazena, se tím může stát nedostupnou. To však v tomto okamžiku není pro algoritmus zajímavé. Taková reference bude případně uvolněna v dalším kole kolekce. V aktuální kolekci však nezpůsobí odstranění reference, na níž ještě existuje odkaz, a neporuší tím korektnost algoritmu. Reference vkládaná do instance se však mohla stát po dočasné nedostupnosti opět dostupnou. Je tedy nutné označit vše, na co ukazuje.

Z těchto důvodů obsahují obě funkce speciální kód, který v případě, že je GC aktivní, přistoupenou referenci přebarvují.

Algoritmus kolekce instancí

Algoritmus garbage collectoru pracuje v šesti běhových úrovních:

Runlevel 0

Je stav, během kterého se kolekce neprovádí.

Runlevel 1

Začátek kolekce. Provádí se iniciální nastavení.

1. Všechny reference jsou obarveny zelenou barvou.
2. .heapRefCount tj. počet referencí na instanci z vnitřku haldy je nastaveni na 0
3. Je vynulován příznak `FINALIZER_CANDIDATE` označující objekt, který je kandidátem na finalizaci, a příznak `FINALIZER_READY_TO_DIE` označující objekt, který je připraven pro odstranění z paměti a byl již finalizován.

Runlevel 2

Obarvovací fáze, během které všechny dosažitelné objekty získají červenou barvu.

Obarvovací algoritmus:

1. Algoritmus vychází z množiny kořenových ukazatelů - je provedena inicializace množiny R.
2. Pokud není R prázdná, vyjme se z ní reference r a provede se její rekurzivní obarvení. Pokud je množina R prázdná, algoritmus kolekce končí.
3. Algoritmus rekurzivního obarvení instance r:
   Algoritmus pracuje s referencí typu InstRef. V každé instanci provede následující trasování:
   (a) Rekurzivní obarvování instance právě začalo a nebylo dosud dokončeno. Instanci se tedy přiřadí oranžová barva.
   (b) V hlavičce instance, na niž se r odkazuje, je reference na strukturu RuntimeData mateřské třídy. Tato reference se použije pro získání bitové mapy. Bitová mapa popisuje položky instance. Pokud je nastaven bit na 0, je v odpovídající položce primitivní typ. V opačném případě obsahuje daná položka instance referenci.
   (c) Algoritmus prochází instanci. Tam kde je v instanci uložena reference (jak se zjistilo z bitové mapy), provede její zpracování.
   (d) Nejprve se zjistí barva instance. Pokud je červená resp. oranžová, zanoření do instance, na kterou se reference odkazuje, se neprovádí. Taková reference je totiž již rekurzivně obarvena resp. se na jejím obarvení pracuje. Pokud by se algoritmus zanořil, došlo by k jeho zacyklení. Jestliže je ovšem barva instance zelená resp. modrá, algoritmus se zanoří a pokračuje v bodě a) [2].
   (e) Tímto způsobem se projdou všechny položky instance, které jsou referencí. V této chvíli je reference I obarvena červenou barvou - byla již rekurzivně zpracována.
   (f) Nyní se obarví statické proměnné třídy, které je I instancí. Struktura RuntimeData dané instance obsahuje

[2] Opravit.

jak statické proměnné, tak popisnou bitovou mapu. Situace je analogická jako při obarvování instance. Statické proměnné se projdou podle bitové mapy a v případě, že je položka referencí, provede se její rekurzivní obarvení.

(g) Následuje vynoření.

4. Po zpracování všech kořenových referencí z R je první fáze obarvovacího algoritmu hotová. Reference na instance dosažitelné z množiny kořenových ukazatelů jsou obarveny červenou barvou, nedosažitelné reference barvou zelenou. Dále existují modré reference. Modrá reference vznikne ze zelené reference pokud během obarvovacího algoritmu přistoupí aplikační vlákno do instance, resp. se zvýší počet referencí na instanci. Tím je podchyceno oživování instancí v ukončovačích a patologické způsoby přepojování odkazů, které by vedly ke zmatení obarvovacího algoritmu a následnému odstranění takové instance z paměti.

Runlevel 3

Dokončení obarvování zpracováním modrých referencí.

Množina B modrých referencí se zpracuje stejným způsobem, jako byla zpracována množina R v Runlevel 2.

Obarvování referencí je nyní hotovo. Reference na instance dosažitelné z množiny kořenových ukazatelů jsou obarveny červenou barvou, nedosažitelné reference barvou zelenou. Modré reference již neexistují.[3]

Runlevel 4

Ukončení instancí.

1. Provede se výpočet položky .heapRefCount, který pro všechny zelené reference určí počet odkazů z haldy. Tato akce se provádí, aby bylo možné určit kořeny hierarchie zelených referencí. Jedině tak je možné určit správné pořadí volání ukončovačů.

[3] Otázka: Co když vzniknou právě teď via mutátor?
Odpověď: V tomto runlevelu už k vytváření modrých referencí vůbec nedochází. To bylo již dávno zastaveno (viz například níže uvedené tabulky obarvování).

2. Každé zelené referenci je dále nastaven příznak `FINALIZER_CANDIDATE` označující třídu, která by měla být ukončena.
3. Všechny zelené kořenové reference mají položku .heapRefCount rovnu 0. Ostatní reference nenulovou.
4. Vytvoří se samostatné ukončovací vlákno.

   Algoritmus ukončovacího vlákna:

   (a) Vyhledá všechny kořenové zelené reference, které mají nastaven příznak `FINALIZER_CANDIDATE` a provede jejich rekurzivní finalizaci.

   (b) Algoritmus vyjde od kořenové reference a pokračuje do hloubky pomocí bitové mapy, která je využívána stejně jako při obarvování.

   (c) Pokud má instance nastaven příznak `FINALIZER_JAVA`, zavolá se nejdříve její ukončovač. Následně se vynuluje příznak `FINALIZER_CANDIDATE`. Pokud se během ukončování změní barva reference ze zelené na modrou, znamená to, že se instance oživila (předala odkaz na sebe jinam a stala se tak dosažitelnou). V tomto případě se v rekurzivním ukončování aktuální instance nepokračuje. Položky níže v hierarchii se staly rovněž dosažitelnými. Provede se tedy pouze rekurzivní odstranění příznaků `FINALIZER_CANDIDATE`. Pokud změna barvy nenastala, pokračuje se v rekurzivním ukončování referencí.

   (d) Výše uvedené rekurzivní ukončení se provede v každé zelené kořenové referenci.

   (e) Zbývá provést detekci modré reference. Pokud během ukončování vznikla modrá reference (sama se oživila), kolekce v tomto okamžiku skončí. V důsledku oživení totiž došlo k významné změně vztahů v haldě.

   (f) V případě, že modrá reference neexistuje, provede se vyřazení všech instancí, na něž se odkazuje zelená reference, z vnitřních struktur JVM a jejich dealokace.

Jak plyne z předchozího textu, mutátory přebarvují v některých fázích reference ze zelené barvy na modrou. Následující tabulky práci s barvami upřesňují:

Tabulka 4.1: Vytváření nových referencí

Runlevel	Barva vytvořené reference
0	GREEN
1	GREEN
2	BLUE
3	RED
4	RED

Tabulka 4.2: Přebarvování při přístupech mutátoru

Runlevel	Přebarvení
0	.
1	.
2	$GREEN -> BLUE$
3	.
4	$GREEN -> BLUE$

Ukončení instancí

Podobně jako metoda konstruktoru provádí na objektu inicializaci, metoda ukončovače v jazyce Java provede na objektu činnosti spojené s jeho ukončením. Garbage collection automaticky uvolní paměťové prostředky, které objekty používají. Objekty si však mohou držet prostředky jiných druhů, jako jsou deskriptory k souborům nebo sockety. Garbage collection nemůže tyto prostředky uvolnit, musí být implementována metoda, která se postará o takové věci, jako je uzavření otevřených souborů, ukončení síťových spojení atd.

Ukončovač je virtuální metodou, nepřebírá žádné argumenty, nevrací žádnou hodnotu (tedy ani typu void) a musí se jmenovat finalize.

Poznámky k ukončovačům:

1. Pokud má objekt ukončovač, vyvolá se tato metoda před tím, než systém garbage collection objekt odstraní.

2. JVM Lógr může skončit bez odstranění (garbage collection) všech „visících“ objektů, takže některé ukončovače se nikdy nevyvolají. V tomto případě však veškeré „visící zdroje“ obvykle uvolní samotný operační systém.

3. Java nedává žádné záruku na to, kdy garbage collection skutečně proběhne, nebo v jakém pořadí se objekty budou odstraňovat. Proto se také Java nijak nezaručuje ohledně toho, kdy se zavolá ukončovač, ani v jakém pořadí se ukončovače budou volat, ani které vlákno bude ukončovač provádět.

4. Jakmile se vyvolá ukončovač, objekty se ihned neuvolní. To je proto, že metoda ukončovače může objekt znovu „oživit“, a to tak, že ukazatel this někam přiřadí a tím pádem vzniknou na objekt nové reference. Po zavolání metody finalize() se tudíž musí znovu zjistit, jestli je objekt bez odkazů, než je možné jej skutečně odstranit. I když se ale objekt „oživí“, metoda ukončovače se nikdy nevolá více než jednou.

5. Ukončovač může vyhodit výjimku. Pokud se v metodě ukončovače skutečně vyskytne neodchycená výjimka, systém tuto výjimku ignoruje.

V JVM Lógr je pro vykonávání ukončovačů zmocněno speciální vlákno nízké priority. Jak bylo výše uvedeno, GC provede jedno kolo kolekce a objekty, které by měly být uvolněny, předá ukončovacímu vláknu v Runlevel 4. To provede zpracování paměťových objektů podle finalizerFlags.

Pro ukončování objektů bylo použito samostatné vlákno především z těchto důvodů:

1. Metoda ukončovače může být napsána tak, že je neefektivní, zabere nezanedbatelně procesorového času nebo obsahuje chybu jako například nekonečný cyklus. Pokud by ukončovače zpracovávalo GC vlákno, mohlo by dojít k jeho odstavení (například v prvním případě kdy se může čekat na odezvu ze sítě apod.) nebo porušení. V implementovaném řešení může dojít k porušení ukončovacího vlákna, což zpravidla není fatální. Neodevzdání zdrojů zpravidla zatíží pouze OS, ale nenaruší běh JVM. I v případě porušení ukončovacího vlákna je OS schopen po doběhnutí JVM prostředky opět uvolnit.

2. GC vlákno není zatíženo ukončováním objektů a může tedy provádět to, k čemu je určeno - tedy vyhledávání nedostupných částí paměti.

Pokud ukončovací vlákno úspěšně ukončí svoji práci, provede se uvolnění paměťových uzlů v haldě.

Destrukce tříd

Třída jazyka Java může být uvolněna z paměti pokud jsou splněny následující podmínky:

1. Třída byla zavedena uživatelským zavaděčem tříd.
2. Neexistuje odkaz na instanci daného uživatelského zavaděče tříd.
3. Neexistuje odkaz na žádnou instanci třídy zavedenou daným uživatelským zavaděčem tříd.
4. Neexistuje další odkaz na instanci java.lang.Class pro dané třídy.

Jak plyne z předchozích podmínek lze uvolňovat pouze množiny tříd zavedené určitým uživatelským zavaděčem tříd. Při této práci pracuje narozdíl od recyklace tříd nejen s referencemi typu InstRef, ale rovněž s referencemi typu StatRef. Algoritmus destrukce tříd GC vyjde z tabulky systémového zavaděče tříd (odkaz na jeho tabulku je v uložen v globální proměnné defaultClassloader). Je to kořenová reference této kolekce - jsou z ní dostupné všechny paměťové objekty využívané třídami (nikoli instancemi). [4] V tabulce systémového zavaděče tříd se hledá odkaz na tabulku uživatelského zavaděče. Ta se pozná podle typu StatRef reference `CLASSLOADER_HASHTABLE`. Jestliže se získá taková tabulka, systém se pokusí ověřit výše uvedené podmínky. Situace je usnadněna tím, že GC zná přesné uložení třídy v paměti. Během prozkoumávání obsahu tabulky zavaděče může být nalezena reference na další zavaděč, který byl zaveden aktuálně zpracovávaným. V tom případě se pokračuje analogicky do nižší úrovně hierarchie. Jestliže se v nižší úrovni podaří ověřit platnost podmínek, provede se odstranění třídy z paměti. Hierarchie zavaděčů tříd se tedy ruší odspodu. Díky znalosti uložení v LH je možné provést tuto operaci standardním algoritmem. Odstranění této netriviální datové struktury z LH je tak poměrně efektivní.

Jak je vidět, je algoritmus destrukce tříd časově náročný. Provádí se zpravidla až v krajních situacích, aby zbytečně nezatěžoval běhový systém virtuálního stroje. Na druhou stranu je nutností v systémech, v nichž se hojně využívá uživatelských zavaděčů tříd a které běží nepřetržitě.

V současné verzi JVM Lógr není mechanizmus destrukce tříd (stejně jako například v JDK 1.0.2) plně implementován, nicméně běhový systém jako takový je připraven na jeho doplnění.

Zdrojové kódy servisní struktury jsou obsaženy v Logr/include/gcollector.h a Logr/heap/gcollector.cc, tělo garbage collectoru potom v Logr/-heap/gc.cc. V souborech Logr/heap/gcfinalize.h a Logr/heap/gcfinalize.cc je implementována finalizace.

[4] Poznámka: To nejde, musí se procházet instance tříd ClassLoader. V tabulce systémového classloaderu tabulky uživatelských loaderů nejsou.
Áááááha..... o tom už jsi mi něco říkal — upravím až si to ujasním(e).

Kapitola 5

Běhový systém

Předchozí text se věnoval LH, nad kterou běhový systém pracuje. Byla rozebrána implementace LH, způsob uložení datových struktur jazyka Java i mechanizmy jejich vytváření.

Následující kapitoly se věnují běhovému systému JVM Lógr. Bude v nich popsán 'život' virtuálního stroje počínaje implementací vláken přes interní mechanizmy jádra až po 'bytecode to native' kompilátor.

5.1 Vlákna

Jazyk Java umožňuje konkurenční běh více vláken, jejich vzájemnou synchronizaci a komunikaci. Vlákna jazyka Java lze implementovat dvěma možnými způsoby: kooperativním přepínáním vláken v rámci jedno-vláknového procesu nebo mapováním Java vláken na native vlákna operačního systému. Většina verzí JDK firmy Sun (JDK 1.0 a 1.1) využívá první způsob (green threads), v posledních verzích (JDK 1.2) však začínají používat native vlákna.

Pro vlákna JVM Lógr byly následující možnosti:

1. Použít kooperativní přepínání vláken v rámci jednoho procesu. Tento způsob má výhodu ve snadné kooperaci vláken, odpadá většina synchronizací, jelikož vlákna nikdy neběží skutečně zároveň, a jsou přesně definované body přepnutí vláken. Nevýhodou je těžší dodržení spravedlivého přidělení času vláknům a především nemožnost běhu na více procesorech (JVM v tomto případě využije pouze jeden procesor).

2. Použít native vlákna, konkrétně knihovnu LinuxThreads, která implementuje vlákna podle normy POSIX 1003.1c a která je již součástí

standardní knihovny glibc. Výhodou použití knihovny LinuxThreads je využití standardu (a tím zjednodušení přenositelnosti) a využití již hotového kódu (odpadá implementace vlastní knihovny vláken). Nevýhodou je nutnost přizpůsobení se logice knihovny, která může být v některých ohledech nevyhovující. Použití native vláken navíc vyžaduje mnohem důslednější synchronizaci. Další nevýhodou je, že knihovna LinuxThread je poměrně nová a může tedy potencionálně obsahovat chyby, které by později znesnadňovaly její použití.

3. Další možností by bylo napsat vlastní knihovnu nativních vláken. Tu by bylo možno udělat na míru vláknům jazyka Java, ovšem byla by velice nevýhodná v ohledech přenositelnosti a vyžadovala by příliš velké úsilí na její vytvoření.

V JVM Lógr bylo nakonec použito mapování vláken jazyka Java na vlákna operačního systému Linux (LinuxThreads). Mapování je provedeno 1:1, navíc je použito několik vláken pro garbage collector.

5.1.1 Start vlákna

Java vlákno vzniká při volání metody start() třídy Thread. V ten okamžik je nutno vytvořit native vlákno a svázat jej s Java vláknem. Metoda start() je native a obsahuje kód, který pouze spustí nové Linux vlákno, předá mu referenci na příslušnou instanci třídy Thread a této instanci do 'skryté' instanční položky zaznamená identifikaci Linux vlákna. Nově spuštěné Linux vlákno provede provázání Linux vlákno - Java vlákno a zaznamená si referenci na Thread do lokálních proměnných vlákna.

Dále probíhají inicializace nutné pro spuštění bajtového kódu (funkce startThread v Logr/jvm/threads.c, konkrétně metody run() třídy Thread. Dojde k vytvoření instance struktury RtExceptionInfo, která je nutná pro správu výjimek (viz kapitola 5.10), a spustí metodu run().

5.1.2 Ukončení vlákna

Po návratu z metody run() se otestují výjimky (metoda nemusela být korektně ukončena - mohla 'probublat' výjimka) a pokud nějaká nastala, je vypsána. Nakonec je instance RtExceptionInfo vyřazena ze spojového seznamu, odalokována a Linux vlákno je ukončeno. Před ukončením (voláním `pthread_exit()`) je ještě zkontrolováno, zda ukončované vlákno není poslední Java vlákno, případně zda všechna zbývající vlákna nejsou označena jako démon. V takovém případě je potřeba ukončit celou JVM.

Ukončení vlákna metodou stop()

Volání metody stop() nad instancí třídy Thread je možné poslat vláknu výjimku java.lang.ThreadDeath (podle specifikace by mělo jít zasílat i libovolného jiného potomka java.lang.Throwable, ovšem JVM Lógr umožňuje zasílat pouze ThreadDeath). ThreadDeath je poté v cílovém vláknu vyhozena a 'probublává' zásobníkem dokud není odchycena, nebo se nedostane zpět do funkce startThread(). ThreadDeath není vypsána (jedná se o korektní, i když nedoporučené, ukončení běhu vlákna) a vlákno je ukončeno. Zasílání ThreadDeath funguje v JVM následovně: parametr metody stop() je ignorován a cílovému vláknu je ve struktuře RtExceptionInfo zaznamenáno, že nastala asynchronní výjimka. Tím činnost stop() končí. Při testování, zda nastala výjimka (funkce checkException()), je nejprve otestována asynchronní výjimka a pokud je nastaveno, že ano, je vytvořena instance třídy ThreadDeath, a dále už se s ní zachází jako s normální výjimkou (respektive chybou). ThreadDeath je vždy zaváděn systémovým zavaděčem tříd.

5.2 Synchronizace vláken

Je nutné rozlišit synchronizaci na dvou různých úrovních - vnitřní synchronizace virtuálního stroje a synchronizace na úrovní jazyka Java. Vnitřní synchronizace je provedena běžnými synchronizačními prostředky Linux vláken. Synchronizační prostředky z jazyka Java je potřeba na vnitřní prostředky JVM převést.

5.2.1 Synchronizace jazyka Java

Jazyk Java používá synchronizační primitivum monitor. Pro programátora jsou k použití následující prostředky: synchronized metoda nebo blok (vnitřně totožné, pouze se synchronizuje nad jiným objektem - odpovídá dvojici funkcí monitorEnter() a monitorExit()), instanční metody wait(), notify() a notifyAll() třídy Object. Navíc má třída Thread instanční metodu interrupt(), kterou je ovšem nedoporučeno používat.

Vlákno může do monitoru vstoupit pouze voláním funkce monitorEnter(). Opustit monitor lze voláním funkce monitorExit() nebo dočasně metodou wait(). wait() uvolní monitor, uspí vlákno a čeká na některou z následujících událostí: uplyne požadovaný čas (pokud je metodě wait() zadán), jiné vlákno, které momentálně vlastní monitor, volá notify() nebo notifyAll(), jiné vlákno volá nad čekajícím vláknem interrupt(). Není-li jiné vlákno v

monitoru, vlákno opět do monitoru vstoupí a pokračuje ve výpočtu, jinak čeká na uvolnění monitoru.

5.2.2 Vnitřní implementace

Každá instance (přesněji reference - struktura InstRef) obsahuje položky:

.monitorOwner	které vlákno je v monitoru
.monitorCount	kolikrát je vlákno v monitoru - monitorEnter() lze volat jedním vláknem opakovaně
syncMutex	zámek, kterým se řídí přístup do monitoru (vstupem do monitoru je zámek uzamčen)
syncCond	'condition variable' - použito pro implementaci metody wait()

Struktura RtExceptionInfo obsahuje (mimo jiné) položky:

waitOn	ukazuje na objekt, nad kterým byla zavolána metoda wait()
interrupted	použito pro rozlišení, zda čekání ve wait() bylo přerušeno voláním notify () nebo interrupt()

Při prvním vstupu vlákna do monitoru (volání monitorEnter()) je uzamčen zámek syncMutex a monitorOwner je nastaven na identifikaci aktuálního vlákna. Pokud vlákno již v monitoru je a vstupuje do něj opakovaně, pouze je zvýšen čítač monitorCount. Výstup z monitoru (monitorExit()) snižuje monitorCount a při hodnotě 0 odemyká syncMutex.

5.2.3 Implementace metod wait(), notify(), notifyAll()

Všechny tyto funkce může volat pouze vlastník monitoru. Funkce `wait()` si nejprve zaznamená aktuálního vlastníka monitoru (položky monitorOwner a monitorCount z InstRef), nastaví položku waitOn struktury RtExceptionInfo na aktuální objekt a zavolá funkci `pthread_cond_wait()` nad položkou syncCond ze struktury InstRef aktuálního objektu, případně `pthread_cond_timedwait()` pro wait() s parametrem času. Tím se vlákno uspí, uvolní zámek syncMutex a čeká na probuzení.

Probuzení může přijít od funkce `notify()` (`notifyAll()`) nebo interrupt(). Proto je po probuzení testována položka interrupted struktury

RtExceptionInfo, a v případě, že probuzení nastalo od `interrupt()`, je vyhozena výjimka InterruptedException.

Poté jsou nastaveny monitorOwner a monitorCount na původní hodnotu a waitOn na nullInstRef a `wait()` je ukončen. Metoda `notify()` volá `pthread_cond_signal()`, metoda `notifyAll()` `pthread_cond_broadcast()`, čímž vzbudí jedno nebo všechna čekající vlákna. Nejprve však musí aktuální vlákno opustit monitor, pak může jedno z probuzených vláken pokračovat v činnosti.

5.2.4 Metoda interrupt()

Tato metoda umožňuje probudit jiné vlákno čekající v metodě `wait()` nebo `sleep()`. Proto je i metoda `sleep()` implementována pomocí volání `pthread_cond_timedwait()` (nad RtExceptionInfo.sleepCondVar). Metoda `interrupt()` nalezne instanci RtExceptionInfo a nad jejími položkami sleepCondVar resp. `waitOn->syncCond` volá `pthread_cond_signal()` resp. `pthread_cond_broadcast()`. Ještě nastaví položku interrupted, aby bylo možno detekovat, zda `wait()` a `sleep()` byly ukončeny korektně nebo voláním `interrupt()`.

5.3 Java zásobník

S každým Java vláknem, které je v JVM vytvořeno, je asociován Java zásobník (Java stack viz kapitola 3.5.2 [Spec97]). Ten je vytvořen ve chvíli, kdy se nové vlákno vytváří. Na Java zásobníku se ukládají Java rámce (Java frames). Jde o obdobu rámců, které generují překladače jiných jazyků (např. C/Pascal). V jednom Java rámci (dále jen rámci) je uložen kontext výpočtu v jedné metodě, který tvoří lokální proměnné, mezivýsledky rozpracovaných operací a návratová adresa do předchozí metody. Pokud je během provádění kódu metody volána jiná metody, je pro ní vytvořen nový rámec, který se umístí na vrchol Java zásobníku.

JVM pracuje vždy pouze s naposledy vloženým rámcem. Rámec má (na rozdíl od výše zmíněných jazyků) po dobu své existence konstantní velikost, která je určena počtem lokálních proměnných metody a prostorem pro uložení mezivýsledků. Java zásobník může mít jak staticky omezenou, tak dynamicky se měnící velikost.

Protože JVM je navržena jako procesor bez datových registrů, berou všechny instrukce bajtového kódu (strojový jazyk JVM) své operandy z části rámce zvané Operand Stack. Ta svojí funkcí napodobuje zásobník reálných procesorů, odlišuje se od něj však jednou důležitou vlastností. Jednotlivé

instrukce bajtového kódu sice odebírají operandy z vrcholu struktury Operand Stack, ale vždy ze stejného místa. Představíme-li si Operand Stack jako pole, bere konkrétní instrukce své operandy stále ze stejných indexů. A program v bajtovém kódu musí být napsán tak, aby to vždy byl vrchol struktury Operand Stack. Z toho vyplývá, že pro danou metodu je velikost struktury Operand Stack pevně dána.

Pro uložení lokálních proměnných metody jazyka Java je použita část rámce nazvaná Local Variables. Tato oblast je také použita k předávání parametrů. Právě zavolaná metoda má své skutečné parametry uloženy v lokálních proměnných a to lokální proměnnou s indexem 0 počínaje (instanční metody zde mají uložený skrytý parametr 'this').

JVM rozlišuje následující datové typy, které mohou být uloženy v rámci (respektive jeho příslušných částech). Tyto typy plně odpovídají datovým typům definovaným specifikací jazyka Java.

Int	celé znaménkové číslo (32 bitů). Jako typ Int vystupují i další datové typy (Byte,Char a Boolean) v nespecializovaných instrukcích.
Float	reálné číslo (32 bitů).
Reference	reference na instanci objektu nebo hodnota NULL.
Návratová hodnota	návratová hodnota používaná dvojící instrukcí JSR/RET

Všechny tyto typy zabírají jednu položku Local Variables nebo jednu položku struktury Operand Stack.

Long	celé znaménkové číslo (64 bitů).
Double	reálné číslo (64 bitů).

Oba tyto typy zabírají dvě po sobě jdoucí položky Local Variables nebo struktury Operand Stack. Není kladen požadavek na nějaké speciální zarovnání.

O každé položce Local Variables nebo struktury Operand Stack je třeba vědět jaký typ obsahuje nebo to, že doposud nebyla použita (je neinicializovaná).

5.3.1 Implementace v JVM Lógr

Volbu implementace rámců v JVM Lógr ovlivnilo několik faktorů:

- potřeba těsné spolupráce v C (resp. C++) napsané JVM a do assembleru přeloženého kódu metod jazyka Java.

- jednoduchá a rychlá práce s rámci (jako celkem) a také s jednotlivými položkami (ať už daty nebo typovými informacemi)

Tvorba rámců v paměti alokované JVM byla vyloučena pro obtížnou propagaci informací o rámci do assemblerových metod. JVM Lógr vytváří své rámce na zásobníku, podobně jako jiné jazyky, a to za použití pro ně běžné konvence. Jednotlivé rámce (jak C tak assembler) jsou provázány pomocí opakovaného ukládání registru EBP.

Tvrdší oříšek však představuje uložení vlastních datových položek a hlavně pak jejich typové informace. JVM Lógr stačí odlišit položku s typem reference od všech ostatních. Postupně se zvážilo několik návrhů. (Např. ukládat datové položky do dvou zásobníků (rozuměj datová struktura) rostoucích proti sobě. Na jeden z nich by se ukládaly primitivní datové typy (to jest číselné typy a návratová hodnota), na druhý pak reference. Tato možnost byla zamítnuta pro svou 'registrovou' náročnost a obtížné uložení (respektive) obnovení stavu rámce, obnovení ukazatelů na zásobníky atd.)

Konečná implementace rámce je následující. Začátek (hlavičku rámce, která je uložena směrem od vrcholu zásobníku) tvoří parametry, s nimiž je metoda volána. Všechny metody jazyka Java v JVM Lógr mají následující signaturu v jazyce C.

```
long long javaMethod (RuntimeData,ExceptionInfo)
```

Kde RuntimeData je odkaz na skutečnou třídu, nad kterou je volána metoda. ExceptionInfo je struktura (pro každé vlákno specifická), která nese informace o výjimkách. Protože trampolíny v JVM Lógr nevědí, jaká metoda je jejich prostřednictvím volána, ani jaký typ vrací, očekávají vždy nejdelší možnou návratovou hodnotu long long. (Jedná se o rozšíření jazyka C o 64-bitový celočíselný typ. Kompilátor gcc, pro který je JVM Lógr určen, toto rozšíření podporuje.)

Dále je v hlavičce rámce uložena návratová adresa do JVM Lógr trampolíny, jejíž prostřednictvím byla tato metoda vyvolána. Následuje provázání rámců, pomocí následující sekvence instrukcí:

```
pushl $ebp
movl $esp,$ebp
```

(Zápis je v `AT&T` assembleru, který je v OS Linux používán.)

Protože stejným kódem začínají i metody v C, jsou pomocí registru EBP (EBP vždy ukazuje, na předchozí EBP uložené na zásobníku) dostupné i předchozí rámce.

Pro podporu zpracování výjimek je dále uložen celkový počet datových položek tohoto rámce, a kolik z nich patří do části Local Varibales. Toto rozdělení je nutné zachovat, protože při vyhození výjimky jsou datové položky struktury Operand Stack odstraněny, zatímco datové položky z Local Varibles jsou zachovány.

Odlišení primitivních typů a referencí umožňuje následující bitové pole příznaků. Nastavení bitu na 1 označuje, že odpovídající položka je reference. Volba bitového pole byla učiněna s cílem šetřit paměť, s odstupem času se však ukazuje, že úspora paměti v rámci je vykompenzována delším a méně přehledným kódem metod.

Zbytek rámce tvoří uložené datové položky. Část Local Variables má vždy plnou velikost (i když ne všechny položky jsou využity). Z části Operand Stack jsou přítomny jen využité položky. Vrchol struktury Operand Stack aktuálního (tedy posledního) rámce je totožný s vrcholem zásobníku.

Následují obrázky s přesným popisem Java rámce, příkladem uložení dat v Java rámci a podobou zásobníku během volání Java metody v jiné Java metodě.

Popis Java rámce:

proměnná na (CPU) zásobníku	popis proměnné
...	
registr EBP size=1 x 32b	Uschované EBP z minulé metody (trampolína z LKI). Odpovídá kódu typicky generovanému překladačem: PUSH EBP MOV EBP,ESP
resolved THIS size=32b	Linux adresa instance THIS.
Fields Total size=32b	Velikost Local Variables + maximální velikost struktury Operand Stack. Velikost je udána v počtu položek (položka á 32b).
Fields from LVARS size=32b	Určuje, kolik položek patří do oblasti LVARS.
Field Type size=n x 32b	Pomocné pole příznaků, rozliší položky na primitivní datové typy a reference.
LVARS size=32b na položku	Pole Local Variables.
OSTACK size=32b na položku	Vrchol struktury Operand Stack. Je totožný s vrcholem (CPU) zásobníku.

Poznámky:

1. Ve všech těchto schématech zásobník roste směrem k dolnímu okraji.
2. Označení v tabulce: $? \Leftrightarrow 0 \vee 1$

Příklad uložení dat v Java rámci:

proměnná na (CPU) zásobníku	
hodnota proměnné	význam hodnoty
...	
Fields Total	
3	
Fields from LVARS	
2	
Field Type	
?????001 ???????? ???????? ????????	Bez ohledu na skutečný typ položek v datové oblasti a jejich příslušnost k LVARS či OSTACK můžeme říci, že první položka je reference, další dvě jsou primitivní datové typy, či jeden dlouhý primitivní datový typ (PDT).
LVARS	
`[reference]` `[int]`	První položka LVARS je reference, typické pro instanční metody, kde se jedná o THIS. Podle 'Fields from LVARS' vidíme, že velikost LVARS je 2, a podle předchozí položky 'Field Type' víme, že druhá položka je PDT, například int.
OSTACK	
`[int]`	Část OSTACK je v tomto příkladě tvořená jedinou položkou, což snadno určíme jako, 'Fields Total - Fields from LVARS'. Tato položka je rovněž primitivní datový typ, a aby byl příklad poněkud smysluplný, je to opět int.

Zásobník při volání Java metody:

proměnná na (CPU)zásobníku	popis proměnné
...	
Previous Frame	Metoda asociovaná s tímto rámcem provedla volání jiné metody. Volání metody je nahrazeno voláním trampolíny z LKI s parametry:
THIS	Nad kterým objektem se metoda vyvolává. POZOR! pro statické metody tento parametr chybí. POZOR! trampolíny se liší i podle toho, zda voláme ze statické či instanční metody.
CallPoolIDX	Odkaz do struktury CallPool (viz kapitola 5.8).
RET ADDR	Kód uloží návratovou adresu a zavolá trampolína nepřímo (instrukce JMP).
EBP	Jsme v kódu trampolíny, standardní chování funkcí v C je schovat registr EBP. Ten ukazoval na začátek rámce, kde Java kód uschoval minulý EBP.
...	Rezervováno pro kód trampolíny.
ExceptionInfo	Trampolína volá Java metodu. Všechny Java metody v JVM Lógr se volají stejně: `return_type` (JAVA Func) (`RUNTIMEDATA`, `EXCEPTION_INFO`) Parametr `EXCEPTION_INFO` je Linux ukazatel na strukturu, která je součástí mechanismu asynchronních výjimek (též JNI ignorované výjimky či chyby).
RUNTIMEDATA	Odkaz do LH, určuje třídu, která implementuje kód metody.
EBP	Opět se dostává ke slovu Java a opakuje se situace z první tabulky.

5.4 Logr Kernel Interface

Logr Kernel Interface (dále jen LKI) slouží k přístupu do jádra JVM Lógr z kompilovaného kódu. Tyto přístupy slouží k volání vnitřních funkcí JVM. Přes LKI se provádějí operace, které nemůže provádět překompilovaný kód přímo, nebo operace, jejichž začlenění do kódu by bylo příliš prostorově náročné. Dále instrukce, jejichž kompilace do nativního kódu by byla neefektivní nebo nějakým způsobem problémová, například alokace vícerozměrných polí, volání určitých typů metod, kontrola přetypování (check cast) apod.

LKI funkce pracují se zásobníkem obdobně jako bajtové instrukce se strukturou Operand Stack (viz odpovídající instrukce, kapitola 6 [Spec97]). Instrukce dostávají hodnoty ze zásobníku (Operand Stack) a operandy, které jsou též připraveny na zásobník. Zavolaná LKI funkce tedy dostane vše jako parametry (podle 'C' konvence). Uvnitř LKI funkce se navíc využívá znalosti struktury zásobníku pro zjišťování dalších dat, především aktuální struktury RuntimeData.

Návratová hodnota z LKI funkcí je předávána běžnou 'C' konvencí v registru 'eax', případně v dvojici registrů 'eax' a 'edx' pro návratový typ 'long long'.

LKI funkce mohou vyhazovat výjimky. K tomu je použita dvojice funkcí createSynchException() a throwAbsolute(). Pro throwAbsolute() platí omezení, že může být volána pouze přímo z LKI funkce, ne z žádné další funkce volané z LKI, protože počítá s určitou strukturou zásobníku.

Typy LKI funkcí:

lkiNew	vytváření instancí tříd a polí
lkiInvoke	volání Java metod (viz trampolíny)
lkiArrayAccess	přístup do polí
lkiFieldAccess	přístup do položek tříd
lkiException	testování a vyhazování výjimek
lkiMonitor	vstup a výstup z monitoru
lkiMath	některé matematické instrukce
lkiNative	funkce pro hledání JNI native funkcí a pro čištění paměti po návratu z JNI native funkce

Zdrojový kód obsahují soubory Logr/include/lki.h, Logr/jvm/lki.c, Logr/-heap/new.h a Logr/heap/new.c.

5.5 Native metody v JVM Lógr

V JVM Lógr používáme dva typy nativních metod — jednak standardní JNI (Java Native Interface), použitelný na uživatelské native metody, a Logr Native Interface (LNI). LNI je úzce svázán s JVM, a je použit pouze pro systémové třídy, které potřebují využívat vnitřní funkce JVM.

5.5.1 Jak poznat, který native interface použít

Toto je rozlišeno již při kompilaci souboru .class do souboru .logr. K souboru .class může být přidán soubor popisující změny v jeho kompilaci. Je v něm možno určit jednak právě native metody, které mají být volány pomocí LNI, a dále je možno do souboru .logr přidat položky, které jsou prostředky jazyka Java nedostupné a které je poté možno využívat z nativních metod. Definiční soubory se nacházejí v adresáři Logr/lni/description a mají název `package_Class.def`.

5.5.2 Popis definičního souboru

Komentáře začínají na začátku řádky znaky //. Soubor obsahuje dvě sekce uvozené :fields a :methods.

Sekce :fields obsahuje přidané položky ve tvaru 'název signatura'. Do souboru .logr bude položka přidána jako 'název', mezera v začátku názvu zaručí její nedostupnost prostředky jazyka Java.

Sekce :methods obsahuje seznam native metod (musí být native již v původním souboru .class), které se nebudou volat přes JNI, ale přes LNI. Metody jsou v definičním souboru ve tvaru 'metoda signatura'.

5.5.3 Příklad definičního souboru

```
// soubor: java_lang_Thread.def
// definicni soubor pro tridu java.lang.Thread

:fields

threadId I
// pridana polozka _threadId typu integer
// - obsahuje identifikaci Linux vlakna
```

```
:methods

registerNatives ()V
currentThread ()Ljava/lang/Thread;
// ...a dalsi metody

// konec
```

5.6 Logr Native Interface

Logr Native Interface (LNI) je navržen tak, aby umožňoval co nejrychlejší spuštění a běh native metod a z nich přímý přístup do JVM. Z hlediska bajtového kódu se jedná o normální Java metodu, je tedy volána přes trampolínu, ale místo přeloženého bajtového kódu je kód, který spustí LNI metodu.

5.6.1 Spuštění LNI metody

Spouštěcí kód LNI metody vytvoří na zásobníku strukturu LNIEnv vyplní ji (referencí na aktuální instanci struktury RuntimeData, referencí na this a referencí na aktuální instanci struktury RtExceptionInfo - obojí se zjistí ze zásobníku - viz kapitola 5.3). Zjistí ukazatel na první parametr volané metody v Java zásobníku (což je u nestatické metody reference na this). Poté volá native LNI metodu s parametry: ukazatel na strukturu LNIEnv, ukazatel na první Java parametr. Po návratu odstraní strukturu LNIEnv ze zásobníku a pokračuje dál běžným způsobem (návratem do trampolíny).

Adresa native LNI metody je přímo v přeloženém bajtovém kódu pomocí relokace.

5.6.2 Běh LNI metody

LNI metoda si své parametry zjišťuje přímo z Java zásobníku ze znalosti umístění prvního Java parametru (makro `GET_PARAM`). Navíc dostane parametr LNIEnv, ze kterého se zjistí aktuální instanci struktur RuntimeData a RtExceptionInfo. Volat může libovolnou vnitřní funkci JVM a navíc může využívat funkce LNI pro volání Java metod (lniInvoke), přístupy na položky (lniFieldAccess) a přístupy do polí (lniArrayAccess).
Jednotlivé LNI funkce mají názvy odvozeny z třídy a názvu:

```
package_Class_method(LNIEnv *env, void *params)
```

5.7 Java Native Interface

JNI je standardní rozhraní pro psaní native metod v jazyce Java. Podrobná dokumentace je obsažena v [JNI97]. V JVM Lógr je implementován JNI binárně kompatibilní s JNI z JDK firmy SUN, což umožňuje použít sdílené knihovny JNI metod z JDK.

5.7.1 Spuštění JNI metody

Z hlediska bajtového kódu se jedná, stejně jako v případě LNI, o normální Java metodu, je tedy volána přes trampolínu, ale místo přeloženého bajtového kódu je kód, který spouští JNI metodu. Spouštěcí kód JNI metody vytvoří na zásobníku struktury LJNIEnv a vyplní ji:

fields	= NULL
methods	= NULL
localRefs	= NULL
actRtData	reference na strukturu RuntimeData, zjistí ze zásobníku
excInfo	reference na strukturu RtExceptionInfo, zjistí ze zásobníku
currentThis	zjistí ze zásobníku

Dále připraví parametry - ukazatel na LJniEnv a Java parametry v obráceném pořadí (v 'C' konvenci). Pro každý parametr typu reference, pokud není null, vytvoří na zásobníku strukturu `_jobject` a vyplní ji:

ref	předávaná reference
next	= NULL
prev	= NULL

Pokud je parametr null, `_jobject` se nevytváří a předává se NULL.

Dále spouštěcí kód ze struktury ReferencePool zjistí ukazatel na JNI funkci. Pokud není položka struktury ReferencePool vyplněna, zavolá se lkiFindNativeFunction (adresa známa přes relokaci), která položku vyplní: projde všechny uživatelsky natažené sdílené knihovny a hledá v nich požadovanou metodu. (V JDK 1.2 je procházení knihoven vázáno na aktuální zavaděč tříd, v JVM Lógr je ponecháno prohledávání všech knihoven, podle JDK 1.1).

Zavolá se JNI funkce a po návratu se zavolá LKI funkce lkiJniClear s parametrem LJniEnv. Ta provede vyčištění seznamu lokálních referencí. Návratová hodnota JNI metody se předá běžnou 'C' konvencí zpět do trampolíny.

Seznamy struktur fieldId a methodId se ponechávají v paměti, jelikož knihovny firmy SUN počítají s tím, že fieldId a methodId přežije trvale v paměti. Tato vlastnost není uvedena ve specifikaci, proto byla zjištěna až v pokročilé fázi implementace a není tedy implementována optimálně (struktury nejsou uvolňovány z paměti).

5.7.2 Struktury JNI

LJNIEnv

Struktura předávaná JNI metodě, obsahuje ukazatel na poleJNI funkcí, které lze z JNI metody volat, a další data, která využívají JNI funkce. Položky této struktury využitelné v JNI metodě musí být binárně kompatibilní s JNIEnv z JDK. Binární kompatibilitu nenaruší rozšíření struktury o další položky neviditelné z JNI metody, je tedy možné do ní uložit další potřebné informace pro JNI funkce implementované v JVM Lógr.

Jedná se o ukazatele na spojové seznamy struktur fieldId, methodId a `_jobject` a dále o reference na aktuální instance struktur RuntimeData a RtExceptionInfo a referenci na aktuální Java instanci.

_jobject

Struktura obalující referenci na Java objekty (InstRef). Instance struktur `_jobject` jsou zařazeny ve spojovém seznamu, aby při ukončení JNI metody bylo možno všem použitým referencím snížit čítač referencí.

fieldId, methodId

Protože native metody nemohou používat strukturu ReferencePool (neví umístění požadované položky a ta tam ani ve většině případů nemusí být), potřebují pro přístup na položky a volání metod nějakou identifikaci, přes kterou se odkazovat. Proto se použijí struktury fieldId a methodId, což je vlastně jedna položka ze struktury ReferencePool. Pouze je potřeba tyto struktury naplnit (podobně jako položky ve struktuře ReferencePool při prvním přístupu). Toto řeší JNI funkce GetFieldID() a GetMethodID(), které vytvářejí instance struktur fieldId a methodId. Přístup na položky a volání metod je potom pomocí funkcí, které jako jeden z parametrů dostávají právě

fieldId nebo methodId. Podle původních předpokladů bylo nutno při ukončení JNI metody struktury fieldId a methodId uvolnit, proto jsou řazeny do spojového seznamu, který měl být v LKI funkci lkiJniClear uvolněn. Jelikož však struktury mohou být uloženy v globální proměnné a později požadovány, seznam je ponechávána v paměti.

Všechny jmenované struktury jsou ve zdrojových souborech Logr/jvm/-jni.c, Logr/include/lni.h a Logr/include/ljni.h.

5.8 Volání metod a přístup na položky

Tato kapitola vysvětluje, jakým způsobem je v JVM Lógr implementováno volání Java metod a přístup na položky tříd a rozhraní.

5.8.1 Volání metod

Specifikace JVM rozlišuje celkem čtyři instrukce pro volání metod: invokeinterface, invokestatic, invokespecial a invokevirtual. Všechny pracují tak, že na Java zásobníku jsou uloženy parametry (případně i odkaz na instanci, nad níž se metoda vyvolává) a jako operand instrukce se předává plné jméno volané metody (fully qualified name) a signatura metody (popis typů parametrů).

V Lógr JVM se každá instrukce volání přeloží ve volání speciální funkce z jádra JVM (trampolína), které se předá identifikátor cílové metody. Ten zpočátku odpovídá plnému jménu a signatuře jako v bajtovém kódu, ale po prvním volání pomocí dané 'instance' instrukce se tento symbolický odkaz nahradí přímým odkazem na volanou metodu, takže příští volání již bude rychlejší.

Uvedené nahrazení odkazu je velké zjednodušení, protože vzhledem k možnosti posunu dat v Lógr haldě (a tedy i kódu, neboť kód je v haldě též) se nesmějí používat přímé adresy. Z tohoto důvodu byly zavedeny dvě struktury (NamePool a ReferencePool), které spolu úzce souvisí. Trampolína pak dostává jen odkaz na příslušný záznam v těchto strukturách.

Struktury jsou dvě, protože byla oddělena statická (NamePool) a dynamická (ReferencePool) část uložených informací. Statická část je uložena ve struktuře FixedData (viz dále o uložení tříd v paměti) a je možné ji bez následků vyhazovat z paměti (pro účely stránkování). Naopak dynamická část je ve struktuře RuntimeData a i když by bylo možné ji také zahazovat, je lepší ji mít stále někde uloženou, protože její opětovná inicializace podle statické části je časově náročná.

Obě části jsou uloženy u třídy, která danou metodu volá. Protože ta obvykle volá více metod, je u ní uloženo několik těchto struktur. Zde je ale důležité jejich pořadí. To si musí ve statické a dynamické části odpovídat, protože trampolína dostává pouze adresu dynamické části (přesněji její relativní pozici od začátku struktury RuntimeData dané třídy) a v případě potřeby z ní vypočítává adresu statické části.

Trampolína nejprve najde podle svého parametru dynamickou část volacích údajů a podívá se, zda jsou platné (tj. není v nich uvedena nula). Pokud platné jsou, dojde k pokusu o zavolání metody. Pokud ne, trampolína se snaží najít odpovídající statickou část a podle zde uvedených údajů (plné jméno a signatura metody) vyplní dynamickou část. Pokud se metodu nepodaří najít nebo třída nemá dostatek práv na její vyvolání, je vyhozena výjimka java.lang.NoSuchMethodError. V tomto případě se struktura ReferencePool nevyplňuje. Není ani možné uložit do ní poznámku o selhání (tj. aby se příště již nehledalo a rovnou se vyhodila výjimka), protože při příštím volání by hledání mohlo uspět. Tato situace nastane například při použití uživatelského zavaděče tříd. Při prvním požadavku na vyvolání metody A.m()V dojde k pokusu o zavedení třídy A.

> Poznámka:
>
> Zápis A.m()V znamená, že jde o metodu definovanou jako void m() ve třídě A. Jde o zápis signatury metody, bližší informace najdete ve [Spec97].

Uživatelský zavaděč prohlásí, že třída neexistuje. V tom případě ani metoda neexistuje a vyhodí se výjimka. Při druhém volání téže metody však může zavaděč třídu zavést a tím pádem (pokud má skutečně danou metodu) metoda vyvolána být může. Pokud by ve struktuře ReferencePool byla poznámka o nenalezení metody, volání by již vždy selhalo.

5.8.2 Seznamy metod a jejich vyhledávání

Pro rychlé vyhledávání je u každé třídy seznam všech jejích metod, a to včetně zděděných. Seznam je rozdělen na dvě části a pro implementaci je použita hešovací tabulka. Rozdělení na dvě části vzniklo proto, že volání statických a virtuálních metod (případně metod rozhraní) používá různé instrukce bajtového kódu (lze tedy rozhodovat o typu cílové funkce už při překladu) a není pak nutné hledat v seznamu všech metod. Vzhledem k tomu, že seznam obsahuje též metody zděděné, není nutné při volání hledat metodu u předků dané třídy.

Kromě toho došlo k oddělení seznamu privátních metod (jak statických, tak virtuálních), protože ty se volají pomocí speciální instrukce invokespecial a nejsou přístupné z jiných tříd. Díky tomu není nutné je vyhledávat a lze použít přímo relativní odkazy na jejich kód. Jejich seznam byl zachován kvůli funkcím Java Core Reflection a případnému volání pomocí JNI.

Navíc byl oddělen též seznam konstruktorů, protože ty se nedědí (jsou svázány se svou třídou). Také to zrychluje vyhledávání.

5.8.3 Implementace jednotlivých instrukcí volání

Teď budou probrány jednotlivé typy volání a jejich implementace v Lógr JVM. Ještě je třeba poznamenat, že existují dvě sady trampolín: volané ze statických a z instančních metod. Obě sady mají stejné názvy funkcí až na příponu FromStatic. Rozdíl a přesný způsob volání budou popsány dále.

invokestatic

Tato instrukce slouží k volání neprivátních statických metod. Implementována je trampolínou lkiInvokeStatic v Logr/jvm/stubs.S. Podle údajů ve struktuře NamePool se nejprve zavede do paměti cílová třída. V seznamu jejích neprivátních statických metod se pak hledá cílová metoda. Pokud není nalezena, dojde k vyhození výjimky java.lang.NoSuchMethodError. Jinak je do struktury ReferencePool zapsána reference na třídu, která obsahuje kód volané metody, a relativní adresa této metody od začátku struktury FixedData té třídy. Vyhledávání provádí funkce findStaticMethod() z Logr/jvm/pools.c. Pak se provede volání.

invokevirtual

Tato instrukce vyvolává virtuální metody. Implementaci obstarává trampolína lkiInvokeVirtual.

Vyhledávání je podobné jako u invokestatic, jen funkce se jmenuje findMethod a do struktury ReferencePool se ukládá relativní adresa metody v tabulce virtuálních metod (Virtual Method Table, VMT).

Každá třída obsahující alespoň jednu virtuální metodu (třeba i abstraktní) má ve své struktuře RuntimeData uloženu tabulku virtuálních metod. V ní jsou uvedeny odkazy na kód všech virtuální metod dané třídy. Tabulka se staví při zavedení třídy do paměti z VMT předka (nadtřídy) a seznamu nových či překrytých metod, tedy těch, pro které přímo tato třída obsahuje kód (jsou uvedeny k souboru .logr pro danou třídu). Při stavbě se nejprve překopíruje VMT od předka, na jejíž konec se pak přidávají odkazy

na nové metody a mění se odkazy na překryté. Odkaz se skládá pouze z reference na třídu obsahující kód metody a z relativní adresy metody ve struktuře FixedData této třídy.

Díky tomu má jedna metoda (tedy metoda s daným jménem a signaturou) v celé hierarchii tříd stále stejnou pozici ve VMT, danou jejím umístěním ve VMT první třídy, která ji definuje. Myslí se tedy od místa prvního výskytu. V jiných podstromech se může vyskytovat stejnojmenná metoda, která ovšem má jinou pozici ve VMT.

V uvedeném příkladu jsou metody ClassB1.method ()V a ClassB2.method()V zcela nezávislé, protože jejich společný předek ClassA metodu method ()V neobsahuje. V obou třídách tedy na pohled tatáž metoda může mít různé umístění ve VMT. Ovšem celý podstrom pod třídou ClassB1 (nebo ClassB2) musí mít tuto metodu ve VMT na stejném místě.

Volání pak probíhá tak, že se zjistí třída instance, nad kterou se metoda vyvolává, v ní se najde VMT (tj. najde se VMT dané instance) a z pozice uvedené ve struktuře ReferencePool se přečte cílová třída (tj. ta, která obsahuje kód metody) a relativní adresa cílové metody v rámci struktury FixedData této třídy. Na tu se pak provede volání.

Ještě je třeba zmínit implementaci abstraktních metod. Podle specifikace JVM je při pokusu o vyvolání abstraktní metody vyhozena výjimka java.lang.AbstractMethodError. To lze zařídit například testováním abstraktnosti metody při každém volání. V Lógr JVM ale překladač bajtového kódu pro každou abstraktní metodu vytvoří speciální kód, který slouží pouze k vyhození této výjimky. Metoda pak sice existuje, ale pokud je zavolána, chová se přesně podle specifikace. Je tak zrychleno volání metod, protože se nemusí testovat jejich abstraktnost.

Obrázek demonstruje schéma volání metody void a.fce2 (int,int) z třídy C, když kód je ve třídě B.

invokespecial

Toto je instrukce, která se používá při volání privátních metod, metod zděděných z předka a pro volání konstruktorů (konstruktor je z pohledu bajtového kódu metoda jako každá jiná, jen má speciální název ¡init¿). V Lógr JVM se tato instrukce rozděluje do několika různých případů (tedy i trampolín).

Prvním případem je volání privátních metod. To se v Lógr JVM provádí pomocí trampolín lkiInvokePrivateVirtual, lkiInvokePrivateStatic a lkiInvokePrivateConstructor. To, která z nich se má použít, je známo již při překladu, protože metoda je privátní, tj. je definována ve třídě, ve které je

použita, a lze tedy zjistit její typ. Trampolíny tohoto typu se od ostatních liší tím, že nepoužívají strukturu ReferencePool, ale jako parametr dostávají přímo relativní adresu kódu cílové metody ve struktuře FixedData aktuální třídy (protože ta jediná je smí volat).

Dalším případem jsou metody zděděné od předka (volání super.metoda).

O jejich volání se stará trampolína lkiInvokeSuper. Ta se chová velmi podobně jako lkiInvokeVirtual jen s tím rozdílem, že bere adresu metody ne z VMT dané instance, ale z předka třídy, která má kód metody, kterou by volala lkiInvokeVirtual. Vzhledem k tomu, že statické metody nemají předky (tedy původní, překrytý, kód), neexistuje trampolína lkiInvokeSuperFromStatic pro volání ze statické metody. Pro vyhledávání slouží findMethod.

Posledním případem je volání konstruktorů, a to i zděděných. Jde vlastně o obdobu trampolíny lkiInvokeStatic, protože konstruktor je přímo svázán s nějakou konkrétní třídou, ne instancí. Rozdíl je pouze v tom, že tato trampolína (lkiInvokeConstructor a lkiInvokeSuperConstructor) navíc používá referenci na instanci (konstruktor se vyvolává nad instancí). Pro vyhledávání se používá funkce findConstructor. Funkce lkiInvokeSuperConstructorFromStatic opět neexistuje, protože zděděné konstruktory je možné volat opět pouze z konstruktorů a to nejsou statické metody (i když jsou jim podobné).

invokeinterface

Toto je poslední a nejsložitější volací instrukce (alespoň v implementaci Lógr JVM). Slouží k volání metod nad rozhraními. Jmenuje se lkiInvokeInterface a pro hledání metod používá funkci findInterfaceMethod. Ta hledá metodu standardním způsobem v daném rozhraní a výsledek ukládá do struktury ReferencePool. Pro implementaci této instrukce byla zavedena speciální tabulka metod rozhraní (Interface Method Table, IMT), která je velmi podobná tabulce virtuálních metod. Strukturu této tabulky (tedy pozice metod včetně zděděných) definuje každé rozhraní nezávisle na ostatních (tj. nepřidává metody do zděděné tabulky). Struktura IMT se tedy nedědí jako v případě VMT u tříd a to proto, že rozhraní používají mnohonásobnou dědičnost a v tom případě nelze tabulky dědit.

Vyplněná tabulka je uložena u tříd, které rozhraní implementují. Vzhledem k tomu, že v tabulce jsou pouze odkazy do VMT, lze takto vytvořené tabulky sdílet v jedné linii dědičnosti tříd (tedy spíš opačně, aby bylo možné tabulky sdílet, je v nich pozice ve VMT). Proto se každá IMT vytváří pro první třídu, která dané rozhraní implementuje, a zděděné třídy používají pouze odkaz na tuto tabulku. Každá třída pak má tolik IMT, kolik implementuje tříd. IMT pro třídu se staví až v okamžiku, kdy se nad třídou vyvolá

nějaká metoda daného rozhraní.

Do struktury ReferencePool se ukládá pozice metody v IMT daného rozhraní a reference na toto rozhraní. Tato reference se pak použije při hledání IMT ve třídě, nad kterou instancí se metoda volá.

Obrázek ukazuje schéma volání metody rozhraní void fce2(int,int) na instancí třídy A z třídy C.

Přesný postup akcí při volání

Nyní předpokládejme, že již byl nalezen kód metody (tj. třída, ve které je uložen, a jeho relativní pozice od začátku struktury FixedData této třídy). Dále budou popsány akce nutné pro úspěšné vyvolání metody. Popsané akce mohou být prováděny již v době hledání kódu metody, protože některé úkony s ním spojené jsou identické.

Protože v Lógr JVM se všechny struktury a kód mohou v paměti přesouvat, je nutné použité objekty vždy zamykat proti přesunu. Nejdříve se tedy uzamknou struktury RuntimeData a FixedData cílové třídy. Ty jsou během vykonávání kódu z této třídy stále zamčené. Mohou se odemknout jen při volání jiných metod nebo některých funkcí jádra JVM. Pro instanční metody a konstruktory se navíc ještě zamyká instance, nad níž se metoda vyvolává.

Pak se odemykají struktury RuntimeData a FixedData třídy, jejíž kód způsobil toto volání. Z tohoto důvodu se trampolína musí volat speciálním způsobem. Jako další parametr se jí předává relativní pozice návratové adresy (vzhledem ke struktuře FixedData právě prováděné třídy) a trampolína se nevolá klasickou instrukcí CALL, ale skokem (JMP). Pokud volající metoda byla instanční, je odemčena její instance. To je rozdíl trampolín FromStatic od obyčejných.

Protože cílová adresa je relativní vzhledem ke struktuře FixedData, musí se přičíst k její adrese. To lze již beztrestně udělat, protože tato struktura se již nemůže přesunout. Pak následuje volání (CALL) cílové metody. Zde lze použít klasické volání, protože návratová adresa (tedy trampolína) se v paměti nepřesouvá.

Při návratu se provádí opačný postup. Opět se musí uzamknout volající třída, odemknout volaná a z relativní pozice vypočítat cílová adresa návratu.

Přesné rozložení parametrů na zásobníku je uvedeno v kapitole o překladu volání.

Zvažované alternativy

Při vymýšlení způsobu implementace volání metod se kromě uvedeného řešení vyskytly i jiné varianty, které byly nakonec zamítnuty.

Jednou z možností bylo úplné zrušení struktur NamePool a ReferencePool a použití přímého volání s relokací. Kód cílové metody (nebo pozice ve VMT či IMT) by se nezjišťoval až při prvním volání metody z třídy, ale při zavádění třídy do paměti by se současně zavedly i všechny třídy, na které daná třída směruje volání. Tato varianta by ovšem vedla k přílišnému zpomalení startu JVM, protože by se musely zavést prakticky všechny třídy, které by bylo možné nějakým způsobem použít. To by mělo též značné nároky na paměť a bylo by to zcela nevhodné pro použití v síti (Internet), kdy by se stahovaly i třídy, které se pak za běhu programu vůbec nemusí používat. Tato možnost též koliduje se specifikací JVM, protože chyby typu „Nenalezena třída“ se mají oznamovat až v okamžiku skutečné potřeby této třídy a ne s předstihem. To by vedlo k nutnosti zaznamenávat si nenalezené třídy a všechny odkazy na ně nahradit vyhozením výjimky (nebo poznámkou, že se JVM má pokusit o opakované zavedení třídy, což je vlastně podobné použité variantě). Kromě toho třída na vzdáleném serveru nemusí být při startu programu momentálně přístupná (například z důvodu přetížení serveru), ale později za běhu ano. JVM pak chybně usoudí, že třída nebyla nalezena (nebo na ni bude čekat a start se opět zpomalí). Ještě horší variantou by bylo natahovat všechny třídy již při překladu a víceméně vytvořit jeden spustitelný soubor obsahující všechny třídy. V tom případě by k zavedení kódu bylo možné použít přímo operační systém, ale zase by se třídy musely překládat zvlášť pro každý program. Tyto opětovné překlady by pak příliš zpomalovaly běh, protože by nebylo možné používat již jednou přeložené třídy. Například systémové třídy se překládají jen při prvním spuštění a jinak se jen používají (pokud se nezměnily).

5.8.4 Přístup na položky tříd, rozhraní a instancí

Pro přístup na položky používá bajtový kód čtyři instrukce: na čtení/zápis a pro statické/instanční: getfield, putfield, getstatic a putstatic. Jako parametr se vždy předává identifikátor třídy, do které položka patří, a jméno a signatura této položky. U instančních položek je navíc potřeba reference na instanci, ke které se operace vztahuje.

V Lógr JVM se tyto instrukce realizují několika způsoby. Nejjednodušší je přístup na privátní položky (instanční i statické). Na tyto položky lze přistupovat jen v případě, že se provádí kód té třídy, ve které jsou definovány. V takovém případě je tedy známa adresa položky v paměti až na jisté posunutí. Toto posunutí je dáno v případě instančních položek počtem a typem položek zděděných od předka (při zavádění třídy je známé, provede se posunutí v operandu instrukce) a v případě statických položek pouze

velikostí údajů ve struktuře RuntimeData předcházejících statické položky (opět známé při zavádění - je tedy vidět, že statické položky jsou uloženy ve struktuře RuntimeData. Pro úspěšný přístup k položkám je též potřeba uzamknout paměťový objekt obsahující položku proti přesunu. V případě (privátních) statických položek to není nutné, protože struktury RuntimeData a FixedData třídy, jejíž kód se provádí, jsou vždy zamčeny. U instančních položek je situace složitější. Proto se zde rozlišuje přístup na položky od aktuální instance (this v jazyce Java) a na ostatní. V prvním případě opět není třeba zamykat, protože instance this je vždy zamčena. V druhém případě se o korektní uzamčení a odemčení stará přímo přeložený kód. Ovšem detekce přístupu do aktuální instance není jednoduchá.

Druhou variantou je přístup na zděděné instanční položky. K tomu se používá stejná metoda (přímý přístup), jen místo jednoduché relokace se provádí vyhledání položky ve třídě (zjištění její pozice v instanci). Zde se může stát, že položka v dané třídě chybí. V tom případě se místo přístupu (instrukce MOV) přepíše instrukcí na vyvolání výjimky java.lang.NoSuchFieldError (CALL). Opět je možné takto přistupovat do aktuální i jiné instance.

Poslední možností je přístup na ostatní položky (i když takto lze přistupovat na všechny kromě privátních). Jde o podobnou metodu jako bylo volání metod. Opět se používají speciální funkce na přístup a jako parametr se předává odkaz na strukturu ReferencePool (a též NamePool). Podobně jako u volání funkcí se položka vyhledává ve třídě a její pozice se pak uloží do struktury ReferencePool. Díky tomu může být další přístup mnohem rychlejší. V tomto případě se ale paměťový objekt s položkou zamyká vždy. Pro vyhledávání položek slouží funkce findField a findStaticField a pro přístup pak funkce lkiPut/GetField/Static s příponou Ref, D nebo žádnou podle typu položky (reference, zdvojený word, word). Zde je nutné ještě zmínit existenci tzv. finálních položek (final). Tyto položky jsou specifikací jazyka určeny pouze pro čtení, avšak jejich inicializace se provádí uvnitř konstruktoru (nebo statického inicializátoru). Díky tomu nelze zápis do nich přímo zakázat. V Lógr JVM byla zvolena cesta dvou sad funkcí na zápis do položek. Jedna z nich (lkiPutInitField/Static) umožnuje zápis do položky vždy a je určena pro použití v konstruktorech druhá (lkiPutField/Static) kontroluje atribut final a zapisuje pouze v případě, že není nastaven. Na základě výsledku testu se pak buď vyplní (není nastaven) nebo nevyplní (je nastaven) odpovídající struktura ReferencePool (díky tomu se při selhání vždy vyhledává položka znovu, je tedy pomalejší, ale méně časté). Protože první z funkcí uspěje nezávisle na atributu final (tj. vyplní strukturu), musí být pro každou ze zápisových funkcí použita jiná struktura ReferencePool (a tedy

i NamePool). Na druhou stranu ale lze použít pro čtení a první variantu zápisu tutéž strukturu.

Zvolený způsob implementace finálních položek umožňuje opakovaný zápis do těchto položek, ale pouze v konstruktorech. To nepřináší žádné bezpečnostní problémy, protože funkce umožňující tento přístup je možné volat pouze z konstruktorů a pouze nad aktuální instancí (to zaručuje překladač bajtového kódu).

Lepším způsobem je zapisovat do finálních položek přímo jako do privátních, protože je možné takto zapisovat jen v konstruktorech třídy, která položky definuje, tedy při překladu je známo přesné umístění položek až na posunutí. Tato lepší možnost byla odhalena až později a proto není překladačem využívána (všechny zápisy do finálních položek se provádějí přes uvedené funkce).

Ještě je třeba zmínit přesný postup při čtení a zápisu položek typu reference (kvůli garbage collection) a dlouhých typů (long, double), které, pokud mají nastaven bit volatile, se musejí chovat atomicky. V Lógr JVM se pro jednotnost přístupu (a specifikace JVM to navíc doporučuje) provádějí nad dlouhými typy pouze atomické operace.

5.8.5 Přístup k polím

Pro přístup do polí z Java kódu jsou určeny LKI funkce lkiArrayLoad¡typ¿ a lkiArrayStore¡typ¿.

Jejich činnost je následující:

nejprve je otestováno, zda index odpovídá velikosti pole. Pokud ne, je vyhozena výjimka ArrayIndexOutOfBoundsException. Poté je přečtena/zapsána hodnota z pole, u referencí pomocí get/putRefField. Pozice odkud/kam se má číst/zapisovat se vypočte

$$InstRef -> ptr + aeaOffset + index * size$$

kde ptr je aktuální umístění pole v paměti, aeaOffset je konstanta, kde v instanci pole začínají prvky pole (viz kapitola 4.2.1), size je velikost typu prvků pole. Během celé operace přístupu do pole je pole uzamčeno proti posunu v paměti pomocí lockInstRef().

5.8.6 Přístup na položky z JVM

Zevnitř JVM je potřeba přistupovat na některé položky Java objektů, a to buď na skutečné Java položky nebo na položky přidané (pomocí popisného souboru pro LNI - viz kapitola 5.6). Přístup je potřeba provádět přes

strukturu ReferencePool, ovšem ReferencePool je vždy pro konkrétní třídu a konkrétní přístupy, které třída provádí (viz kapitola o strukturách Pool). Proto je potřeba vytvořit strukturu ReferencePool, která je globální pro celou JVM.

K této 'fiktivní' struktuře ReferencePool se při startu JVM vytváří a vyplňuje struktura NamePool. Při přístupech přes ReferencePool se provádí test, zda je příslušná položka vyplněna ('lazy' přístup - stejně jako při ostatních přístupech do položek) a po vyplnění podle struktury NamePool je možno přistupovat. Vyplnění 'fiktivní' struktury vyžaduje natažení tříd do nichž struktura ukazuje, ovšem těchto tříd není mnoho (několik málo systémových tříd viz Logr/jvm/fictive.c).

Speciální položkou je položka pro volání konstruktoru výjimek. Jelikož se používá velké množství výjimek a bylo by zbytečné vytvářet pro každou vlastní položku (konstruktor není virtuální, není tedy možné použít položku vyplněnou na předka), byl zvolen přístup, který je sice pomalejší, ale přehlednější: vždy před voláním konstruktoru výjimky je položka 'fiktivní' struktury ReferencePool znovu vyplněna pro konkrétní výjimku, je tedy před voláním konstruktoru nutno volat funkci registerExcConstructor(), která položku vyplní.

Přístup se poté provádí přes LNI přístupové funkce, které dostávají jako jeden z parametrů index do 'fiktivní' struktury ReferencePool.

5.9 Vytvoření instance

Vytvoření instance třídy spočívá v alokování potřebného místa v Lógr haldě a inicializování. Inicializace provede překopírování připraveného obrazu iniciálního stavu instance ze struktury RuntimeData. V tomto obrazu jsou inicializované položky instance nastaveny na iniciální hodnotu, ostatní jsou nastaveny na nulu či nullInstRef.

U polí se inicializuje položka 'length' na aktuálně vytvářenou velikost pole, prvky pole jsou inicializovány na nulu (či nullInstRef), případně na zvolenou hodnotu (stejnou pro všechny prvky).

5.9.1 Statický inicializátor třídy

V jazyce Java je možné vytvářet statické inicializátory (v JVM pojmenované `<clinit>`), což jsou vlastně takové konstruktory tříd. V některých případech ho dokonce sám vytváří překladač (javac), protože jím vytváří inicializované statické položky (hlavně objekty), nebo dokonce i konstanty (static final).

V Lógr JVM je navíc pro jednoduchost vždy přítomen (alespoň prázdný, s jedinou instrukcí ret).

Podle specifikace JVM je statický inicializátor volán při prvním tzv. aktivním použití (active use). Za aktivní použití se považuje přístup na statickou nebo instanční položku, vytvoření instance a volání metody. Prakticky to znamená pouze přístup na statickou položku, volání konstruktoru a volání statické metody. Ostatní případy jsou zahrnuty v popsaných. Pro přístup na instanční položku je nejprve nutné vytvořit instanci a při vytvoření instance se vždy musí volat konstruktor. Virtuální metody lze opět volat jen pomocí instance.

Díky tomu je statický inicializátor volán jen v případě přístupu ke statické položce nebo při volání konstruktoru či statické metody. Aby mohla jedna z možností nastat, je nutné nejprve odpovídající 'objekt' najít, tj. vyplnit pro něj strukturu ReferencePoolEntry. Ta se ve většině případů vyplňuje těsně před přístupem či voláním. Proto se statický inicializátor volá při hledání, tj. z funkcí findStaticField, findConstructor a findStaticMethod. V těchto místech volání stačí, protože přímé přístupy z kódu nejsou možné z jiné třídy (viz relokace v kapitole o 4.3), jedinou výjimkou je přístup na zděděné instanční položky, ale v tom případě již musel být volán konstruktor předka.

Podle specifikace JVM se vlákna při volání statického incializátoru synchronizují pomocí zámku nad instancí třídy java.lang.Class pro danou třídu. Při startu JVM ovšem pro některé třídy instance java.lang.Class neexistuje (např. pro java.lang.Object nebo i pro samotnou třídu java.lang.Class, protože pro vytvoření instance je třeba volat konstruktor a před jeho voláním se musí volat statický inicializátor). Právě kvůli tomuto problému je možné volat statický inicializátor též bez synchronizace, ale v tomto případě je testováno, zda inicializuje právě jedno vlákno (vlastník v hlavičce RuntimeDataHeader). Pokud ne, dojde k ukončení JVM (protože tato situace nastává pouze pro několik tříd při startu, tj. dokud se nezavede třída java.lang.Class). Po inicializování třídy se nastaví bit `RDF_INITIALIZED` v hlavičce RuntimeDataHeader třídy.

Implementace volání statického incializátoru je v souboru Logr/jvm/pool.c, funkce initializeClass. Funkce volající tuto inicializaci nejdříve testují bit `RDF_INITIALIZED`, aby initializeClass nevolaly zbytečně. Kvůli synchronizaci tento bit testuje též initializeClass při zamčené instanci java.lang.Class pro danou třídu.

5.10 Výjimky

Jazyk Java podporuje zpracování výjimek (viz kapitola 2.16 [Spec97]). Všechny výjimky jsou potomci třídy Throwable a dále jsou rozlišovány jako error (potomci třídy Error) a exception (potomci třídy Exception). Uvnitř JVM Lógr nejsou výjimky na error a exception rozlišovány, proto se v dalším textu bude hovořit pouze o výjimkách a bude tím myšlen error i exception.

Dále lze výjimky rozlišit na synchronní a asynchronní. Synchronní výjimky jsou vyhazovány na přesně definovaných místech programu příkazem throw. Asynchronní výjimka je libovolná výjimka (typicky však pouze ThreadDeath) zaslaná jedním vláknem jinému vláknu (viz kapitola 5.1 - Ukončení vlákna metodou stop()).

Implementace výjimek v JVM LÓGR

5.10.1 Detekce výjimky

Výskyt výjimky je detekován na těchto místech:

- každá LKI funkce detekuje výskyt synchronní výjimky, kterou způsobila sama LKI funkce nebo funkce z ní volaná.
- trampolíny testují výskyt jakékoliv výjimky (synchronní, asynchronní, suspend).
- v přeloženém bajtovém kódu při každém zpětném skoku je testován výskyt asynchronních výjimek (a suspend). Tím se zaručí odchycení asynchronních výjimek i pokud se běh metody delší dobu nedostane do žádné trampolíny. Nezaručí se tím okamžité odchycení, ale to specifikace povoluje.

Informace o vyhozené výjimce jsou uloženy ve struktuře RtExceptionInfo (soubor Logr/include/exception.h). Instance této struktury je vytvořena pro každé vlákno při jeho startu (viz kapitola 5.1 - Start vlákna).

RtExceptionInfo obsahuje záznam pro synchronní výjimku, pro asynchronní výjimku (pouze čítač výjimek ThreadDeath, zasílání jiných asynchronních výjimek není možné) a dále záznamy pro další asynchronní události, konkrétně suspend (volání suspend() a resume() nad instancí třídy Thread) a interrupt (volání interrupt()).

Instance struktury RtExceptionInfo je na místech, kde má být testováno vyhození výjimky, kontrolována, zda neobsahuje výjimky nebo asynchronní události. Rozlišuje se, odkud je test prováděn, zda z bajtového kódu nebo

zevnitř z JVM (případně z native metod). Pokud je RtExceptionInfo testováno uvnitř JVM (funkce checkSynchException()), testuje se pouze synchronní výjimka. Pokud je zpracována, je z RtExceptionInfo vymazána (deleteSynchException()), aby se nepropagovala dále. Asynchronních událostí se netestují proto, aby se nekomplikovalo testování výjimek rozlišováním jejich druhů (jelikož, pokud by byla odchycena asynchronní výjimka, muselo by se ještě opět testovat, zda nenastala předpokládaná výjimka synchronní) a aby nedošlo k zablokování vlákna uvnitř JVM voláním suspend().

Pokud je výjimka testována z bajtového kódu nebo trampolíny, testuje se nejprve, zda nastala jakákoliv výjimka (test položky flags v RtExceptionInfo na nenulovou hodnotu). Pokud ano, je rozlišen typ (funkce isException()) a to následujícím způsobem:

- pokud se jedná o synchronní výjimku, funkce isExceptin vrací TRUE a poté je spuštěno probublávání (viz níže).
- pokud se jedná o asynchronní výjimku (ThreadDeath), je převedena na synchronní a pokračuje se jako v předchozím bodě.
- pokud se jedná o suspend, vlákno je uspáno (voláním `pthread_cond_wait()` nad položkou suspendCondVar struktury RtExceptionInfo) a čeká se na jeho probuzení (resume). Pokud během „spánku“ nedošlo k asynchronní výjimce, funkce vrací FALSE.

Přístup k asynchronním záznamům v RtExceptionInfo (příznaky a čítače asynchronních výjimek) musí být synchronizován, jelikož ke stejnému záznamu může přistupovat více vláken.

5.10.2 Bublání výjimky zásobníkem a hledání její obsluhy

Přesný popis mechanizmu výjimek je možno nalézt ve [Spec97] kapitola 2.16. Tento dokument popisuje pouze vyhledání obsluhy (bloku catch) již vytvořené výjimky. Výjimkou se rozumí instance třídy java.lang.Throwable nebo některé její podtřídy.
Situace z pohledu zásobníku volání (zásobník roste dolů):

```
...
Java metoda
trampolína
Java metoda   toto je AKTUÁLNÍ Java metoda - ta, do níž
              se šíří výjimka
trampolína    která právě zjistila výjimku
```

Funkce throwAbsolute resp. throwRelative bývají volány z trampolíny (trampolína předává řízení mezi Java metodami) v případě, že trampolína zjistí výjimku (výjimku např. způsobí Java metoda volaná z trampolíny). Funkce throwAbsolute resp. throwRelative mají za úkol nalézt v aktuální Java metodě obsluhu vyhozené výjimky a předat jí řízení. Pokud taková obsluha v aktuální metodě není, je třeba vyčistit Java rámec aktuální metody a předat řízení trampolíně, která aktuální metodu zavolala.

Na vrcholu zásobníku volání je tedy vlastní funkce throwAbsolute resp. throwRelative, pod ní je trampolína a teprve pod tou se nachází Java rámec metody, z níž je vyhozena výjimka.
Příklad 1:
(zásobník volání:)

```
...
Java metoda M1
trampolína
Java metoda M2
trampolína          (zde je zjištěna výjimka)
throwRelative
```

Po skončení throwRelative, byla obsluha nalezena přímo v metodě M2:

```
...
Java metoda M1
trampolína
Java metoda M2
```

Příklad 2:
(zásobník volání:)

```
...
Java metoda M1
trampolína
Java metoda M2
trampolína       (zde je zjištěna výjimka)
throwRelative
```

Po skončení throwRelative nebyla obsluha v metodě M2 nalezena.

```
...
Java metoda M1
trampolína       (zde je zjištěna výjimka)
throwRelative
```

Po skončení throwRelative byla obsluha konečně nalezena v metodě M1.

```
...
Java metoda M1
```

Poznámka:

Není nutné, aby funkce throwAbsolute resp. throwRelative byla volána z trampolíny. Pouze je požadováno, aby mezi Java metodou a funkcí throwAbsolute resp. throwRelative byla v zásobníku volání právě jedna funkce. (Musí být definované, jak hluboko se nachází rámec Java metody.)

Algoritmus hledání obsluhy výjimky vypadá takto:

1. Vyčištění struktury Operand Stack v Java rámci aktuální Java metody. Po vyčištění zůstává ve struktuře Operand Stack pouze instance vyhozené výjimky, ostatní data (např. mezivýsledky operace, při níž byla výjimka vyhozena) jsou zahozeny. Struktura Local Variables je ponechána beze změny.

2. Prohledání obsluh v aktuální metodě. Je-li obsluha odpovídající vyhozené výjimce nalezena, je této obsluze předáno řízení.

3. Obsluha v aktuální metodě nalezena nebyla. Teď je třeba vyčistit celý Java rámec aktuální metody a metodu násilně opustit. Byla-li metoda označena jako synchronized, byl při vstupu do ní zamknut synchronizační zámek (u instančních metod jejich vlastní zámek, u metod statických instanční zámek instance třídy java.lang.Class). Zamknutý synchronizační zámek je nyní při násilném ukončení metody třeba odemknout. Řízení se poté předává v zásobníku volání o úroveň výše - do trampolíny, která aktuální metodu zavolala. Trampolína zjistí, že výjimka nebyla dosud obsloužena a znovu zavolá funkci throwRelative resp. throwAbsolute, pokračuje se tedy bodem 1).

   Poznámka:

   Metoda main (první spouštěná Java metoda) je v JVM Lógr volána ze zvláštní startovací funkce, která zachytí jakoukoli výjimku. To zajišťuje správnost výše uvedeného algoritmu (není třeba testovat, zda aktuální metodou je již main() a končit bublání).

Technické detaily

Položky struktur Operand Stack a Local Variables v Java rámci jsou dvojího druhu — primitivní datové typy a reference. K jejich rozlišení slouží bitová mapa umístěná rovněž v Java rámci. Tato bitová mapa je použita při čištění:

- referencím se sníží čítač referencí
- primitivní datové typy se jednoduše „zapomenou“

Pozice, do níž se výjimka vyhazuje, je místo, odkud byla volána trampolína, která výjimka zjistila.
Obsluhy výjimek vypadají následovně:

```
<startovní pozice>
<koncová pozice>
<typ výjimky>
```

Obsluha „chrání“ přeložený kód v úseku <startovní pozice, koncová pozice> a zachytává instance třídy <typ výjimky> nebo její podtřídy. (Existuje ještě zvláštní hodnota typu výjimky, která zachytává vše.) Obsluhy výjimek se nemohou částečně překrývat: jsou buď disjunktní, nebo jedna obsahuje druhou. V JVM Lógr jsou navíc obsluhy výjimek setříděny vzestupně podle délky kódu, který „chrání“. To zaručuje, že lineární průchod takto setříděným seznamem postupně nachází obsluhy ve správném uspořádání vzhledem k inkluzi (tj. nejdříve najde ty, které nejtěsněji obmykají pozici, do níž se výjimka šíří). Každá obsluha „chránící“ pozici, do níž je výjimka vyhozena, je prověřena, zda zachytává výjimku typu <typ výjimky>. Pokud ano, je nalezené obsluze předáno řízení. Pokud ne, hledá se v setříděném seznamu další obsluha „chránící“ pozici, do níž se výjimka vyhazuje.

V čem se liší throwRelative a throwAbsolute

Trampolíny nejsou volány instrukcí call, ale sledem instrukcí

```
push <návratová adresa>
jmp <adresa trampolíny>
```

To proto, aby návratová adresa mohla být relativní vzhledem k začátku struktury FixedData, v níž je umístěn kód metody, z níž se trampolína volá (FixedData se totiž mohou v haldě pohnout). Naproti tomu např. LKI funkce jsou volány instrukcí

```
call
```

(která ukládá na zásobník absolutní návratovou adresu). Funkce throwAbsolute je volána v případě absolutní návratové adresy (je tedy volána např. z LKI funkcí), throwRelative je použita v případě návratové adresy relativní (tedy z trampolín).

5.11 Start JVM

Před spuštěním metody main() Java programu je nutné inicializovat celou JVM. Nejprve se zjistí nastavené parametry a proměnné prostředí, které určí jak parametrizovat další inicializace. Poté proběhnou inicializace jednotlivých částí JVM v pořadí:

1. reference
   systém pro práci s referencemi, Lógr halda, garbage collector
2. threads
   inicializace nutné pro běh Linux vláken
3. exception
   inicializace struktur pro správu Java výjimek
4. class loader
   inicializace struktur standardního zavaděče tříd
5. fict inicializace
   'fiktivních' položek tříd
6. jni
   připravuje struktury pro spuštění JNI metod a funkcí

Dále se zjistí jméno třídy obsahující metodu main() a parametry, ze kterých se vytvoří instance třídy String. Tyto parametry se uloží do pole, které se později předá spouštěné metodě main().

Následuje natažení třídy s metodou main() a třídy Thread, od které se vytvoří instance a zavolá konstruktor. To navíc vyžaduje natažení a inicializaci třídy ThreadGroup a 'podvodnou' inicializaci třídy Thread, protože není možné vytvořit instanci Thread bez již existující instance Thread. Pokud žádná z předchozích operací nezpůsobila výjimku, vytvoří se Linux vlákno a v něm je spuštěna funkce startMainThread(). Tím startovací vlákno končí svoji činnost, uspí se a čeká na probuzení posledním Java vláknem, aby poté provedlo ukončení JVM.

Funkce startMainThread provádí takřka stejné inicializace jako start jakéhokoliv dalšího Java vlákna (viz kapitola 5.1 - start vlákna).

5.11.1 Popis jednotlivých inicializací

referenceInitialize

Vytvoří iniciální efemérní oblast Lógr haldy, vytvoří základ servisní struktury referencí, inicializuje nulové reference (nullStatRef a nullInstRef), podle obsahu příkazové řádky případně spustí vlákno asynchronní garbage collection.

threadsInitialize

Inicializuje některé klíče pro přístup do lokálních proměnných vláken, inicializují se zámky a čítač nedémonických vláken.

exceptionInitialize

Zde se inicializují další klíče a zámky nutné pro synchronizaci funkcí na správu výjimek. Dále se vytvoří hlava seznamu struktur RtExceptionInfo, přes kterou se poté dohledávají instance RtExceptionInfo patřící konkrétnímu vláknu (pro předání asynchronní události). Poté je ještě inicializován seznam výjimek vyhazovatelných zevnitř JVM.

defaultClassLoaderInit

Inicializuje vnitřní zavaděč tříd. Funkce postaví prázdnou hešovací tabulku a zavede třídu java.lang.Class potřebnou pro zavádění všech tříd (pro každou strukturu RuntimeData je třeba vytvořit instanci třídy java.lang.Class). Zavádění třídy java.lang.Class si rekurzivně žádá zavádění dalších tříd. Pro žádnou z těchto rekurzivně zaváděných tříd nemůže být instance třídy java.lang.Class vytvořena, a tak po úspěšném zavedení třídy java.lang.Class je nutné hešovací tabulku projít (nevíme, jaké třídy byly rekurzivně zavedeny) a pro každou třídu v tabulce nalezenou je třeba dokončit vytvoření instance třídy java.lang.Class. Dále je zavedena třída java.lang.String potřebná pro tvorbu řetězců a poté je zavolána funkce assignStaticStrings, která dokončí tvorbu statických řetězců v dosud zavedených třídách.

fictInitialize

Vytvoří globální 'fiktivní' strukturu ReferencePool, určenou pro přístup na položky Java objektů z JVM, a k ní příslušnou strukturu NamePool, která se ihned vyplní. jniInitialize Provede inicializace pole ukazatelů na JNI funkce,

seznamu globálních referencí a zámku pro výlučný přístup k tomuto seznamu. Poté provede natažení sdílené knihovny 'libjava.so', která obsahuje native funkce knihoven napsané pomocí JNI.

5.12 Ukončení běhu JVM

JVM má být ukončena v okamžiku, kdy neběží žádné Java vlákno, nebo pokud běží pouze vlákna označená jako démonická.

Při startu každého nedémonického vlákna je zvyšován čítač nedémonických vláken, tedy při ukončování vlákna je snadné zjistit, zda se jedná o poslední nedémonické vlákno a JVM se má ukončit.

Při skončení Java vlákna (konec metody run() třídy Thread) se provedou všechny čistící akce, zlikviduje se RtExceptionInfo vlákna, instanci třídy Thread se nastaví, že je vlákno ukončeno. Nyní se zjistí, zda existují ještě nějaká další nedémonická vlákna a pokud ano, Linux vlákno je ukončeno (volání `pthread_exit()`). Pokud ne, je potřeba ukončit všechna démonická vlákna (to se provede zasláním asynchonní výjimky ThreadDeath všem těmto vláknům) a ukončit běh JVM. To se zajistí tím, že se probudí hlavní (startovací) vlákno JVM a aktuální vlákno se ukončí. Hlavní vlákno po probuzení konečně ukončí běh JVM, tedy uvolní Lógr haldu, provede deinicializace všech částí JVM, uvolní systémové prostředky a ukončí svoji činnost.

5.13 Verifikace bajtového kódu

Verifikace bajtového kódu je proces statické kontroly kódu, při kterém je ověřována správnost použití instrukcí bajtového kódu metody. Tím je umožněn následný úspornější překlad do strojového kódu a také rychlejší vykonávání. Detailní popis omezení kladených na bajtový kód metod a původní návrh implementace je uveden v kapitolách 4.8 a 4.9 [Spec97].

Implementace verifikace v JVM Lógr se trochu liší od původního návrhu z [Spec97]. Hlavní změnou je rozdělení kódu do souvislých bloků instrukcí, které neobsahují žádná větvení vykonávání. Proto se může stav rámce ukládat pouze pro tyto bloky a ne pro každou instrukci, jak je tam uvedeno. Každý takový blok obsahuje adresu svého začátku, pomocné příznaky, ukazatel na následující blok a počáteční stav rámce při vstupu do bloku. Stav rámce během verifikace aktuálního bloku je uchováván v tzv. aktivním rámci.

Stav rámce se skládá z ukazatele na vrchol zásobníku a z popisu typu hodnot na zásobníku a v lokálních proměnných. Popis typu položky rámce

je jedna z následujících hodnot:

TEMPTY	prázdná položka
TUNSTABLE	nestabilní, a proto nepoužitelná
TINT	'byte', 'char', 'boolean', 'short', 'int'
TFLOAT	'float'
TLONG	'long'
TDOUBLE	'double'
THALF	druhých 32bitu 'long' nebo 'double'
TADDRESS	'returnAddress'
TREF	inicializovaná reference
TUREF	neinicializovaná reference 'this' (vyšší hodnoty 'x' značí neinicializovanou referenci vytvořenou instrukcí NEW na adrese 'x - TUREF - 1')

Verifikace probíhá ve dvou hlavních fázích a prvotním předpokladem je, že struktura a jednotlivé položky (např. Constant pool) příslušného .class souboru jsou platné a v pořádku.

5.13.1 Fáze 1:

Bajtový kód metody je rozdělen na souvislé bloky tak, aby:

1. první blok začínal na adrese 0
2. všechny přenosy vykonávání (tj. podmíněné a nepodmíněné skoky) mířily na začátek bloku
3. obsluha přerušení začínala vždy na začátku bloku

Dále se kontroluje, že kód obsahuje pouze instrukce, které mohou být v souboru .class, že cíle všech přenosů vykonávání jsou uvnitř kódu metody, a že 'WIDE' prefix se nachází pouze před instrukcemi, které to povolují.

5.13.2 Fáze 2:

Rámec bloku začínajícího na adrese 0 se nastaví tak, že zásobník je prázdný a v lokálních proměnných jsou popisy argumentů metody (a to i s případnou referencí 'this' u virtuálních metod a konstruktorů). Tento blok se vloží do seznamu bloků určených pro verifikaci.
Kostra algoritmu verifikace bloku:

Opakuj, dokud seznam bloků pro verifikaci není prázdný:

1. nastav aktuální blok na první blok ze tohoto seznamu
2. nastav aktivní rámec podle rámce aktuálního bloku
3. pro každou instrukci:
   (a) spoj (viz kapitola 4.9.2 [Spec97]) aktivní rámec s rámcem počátečního bloku všech obsluh výjimek, které chrání tuto instrukci
   (b) zkontroluj její požadovaný počáteční stav rámce a další specifické požadavky a omezení (kapitoly 4.8 a 6.4 [Spec97])
   (c) simuluj její efekt na aktivním rámci a u přesunů vykonávání spoj aktivní rámec s rámcem cílového bloku
4. spoj aktivní rámec s rámcem následujícího bloku, pokud vykonávání může přejít přes poslední instrukci aktuálního bloku
5. konec

Kromě bloku začínajícího na adrese 0 se blok může dostat do seznamu pro verifikaci pouze, pokud se při spojování jeho rámec změnil.

Dále se kontroluje, že bloky začínají vždy na začátku instrukcí, a že kód metody nekončí uprostřed poslední instrukce.

Porušuje-li bajtový kód jakékoliv omezení na něj kladené, je okamžitě odmítnut, verifikace ukončena a vyhozena výjimka java.lang.VerifyError.

Poznámky:

1. V současné verzi JVM Lógr není správně implementována kontrola omezení kladených na instrukce JSR a RET (viz kapitola 4.9.6 [Spec97]).
2. Rozšíření popisu inicializované reference (TREF) v Java rámci o ukazatel na řetězec s názvem jejího skutečného typu by umožnilo verifikátoru provádět další kontroly staticky před vykonáváním některých instrukcí.

5.14 Překladač bajtového kódu

Dokumentace starého překladače zde není uvedena, protože je velmi pochybného obsahu. Hlavním důvodem ale je, že nyní pracuje sám velký Vojta na vynikajícím, superrychlém a vysoce optimalizujícím překladači nové generace.

Kapitola 6

Nástroje pro ladění

6.1 Jádro ladícího systému

Základem ladícího systému je sada funkcí a maker deklarovaných v souboru Logr/include/debuglog.h. Tyto funkce provádějí logovací výpisy a ukládají je buď do systémového logu, nebo je vypisují na stderr. Systémové logování se provádí dle proměnné prostředí `LOG_USER` na úrovni definované proměnnou prostředí `LOG_PERROR`.

Každý typ výpisu je identifikován celým číslem. To umožňuje určovat, které výpisy mají být zobrazovány. Selekce výpisů je provedena pomocí proměnné prostředí nebo řetězce obsahujícího tyto číselné identifikátory. Následně se zobrazují pouze výpisy s identifikátory, které byly aktivovány.

6.1.1 Funkce a makra pro řízení ladícího systému

DBGstart(string)

Začátek ladění, řetězec string je ve stejném formátu jako hodnota `DEBUG_VAR`. Formát výpisu je dán obsahem proměnné `DEBUG_FORMVAR` nebo `(FILE:LINE)`.

DBGsetFormat(formatString)

Nastavuje formátovací řetězec, stejný formát jako `DEBUG_FORMVAR`.

DBGstop()

Ukončení práce ladícího systému.

DBGaddString(string)

Přidání řetězce obsahujícího číselné identifikátory k seznamu aktivovaných. Řetězec má stejný formát, jako proměnná `DEBUG_VAR`.

DBGremoveString(string)

Odstraní intervaly, které jsou uvedeny v řetězci string ze seznamu aktivních číselných identifikátorů. Řetězec má stejný formát jako proměnná `DEBUG_VAR`.

DBGallowed(ID)

Vrací nenulovou hodnotu pokud číselný identifikátor ID je aktivován. `DBG_ASSERT(expression)` Klasický assert(), který je 'vypnut' v případě, že se ladící systém nepoužívá.

6.1.2 Makra provádějící výstup

Makra, která provádějí výstup jsou dvou základních typů:

DBGxprintf(id,format,args...)

Provádí výpis do logu. Argumenty funkce jsou stejné jako printf jazyka C, navíc první parametr funkce je číselným identifikátorem.

DBGaxprintf(format,args...)

Provádí výpis vždy.

Znak 'x' v názvech výše uvedených maker určuje co vypsat před hláškou:

x	Výpis
nic	použit formátovací řetězec ladícího systému
t	(THREAD:FILE:LINE)
f	(FILE:FUNCTION:LINE)
tf	(THREAD:FILE:FUNCTION:LINE)

6.1.3 Direktivy jazyka C pro řízení ladícího systému

Direktiva	Akce
`DEBUG_VERSION`	Pokud je definován, ladící systém se používá. V opačném případě jsou všechna ladící makra ignorována.
`DEBUG_NAME`	Pod jakým jménem logovat do systémového logu. Pokud není definována, systémový log se nepoužívá.
`DEBUG_VAR`	Direktiva, která obsahuje jméno proměnné prostředí určující co zobrazovat a co logovat.
`DEBUG_FORM_VAR`	Direktiva, která obsahuje jméno proměnné prostředí definující formát výpisu.

Hodnota proměnné prostředí definované direktivou `DEBUG_VAR` je čárkami oddělený seznam číselných hodnot nebo jejich intervalů. Hodnoty mohou být zapsány v decimálním, hexadecimálním nebo osmičkovém formátu dle konvencí jazyka C. Intervaly jsou zapisovány jako

$$IDfrom - IDto$$

Navíc lze specifikovat, zda interval má být vložen ('i') či vyřazen ('x'). Řetězec může vypadat například takto:

$$1000 - 2000x1050 - 1500i1200$$

a výsledek jeho transformace následovně

$$1000 - 1049, 1200, 1501 - 2000$$

Hodnota proměnné prostředí definované direktivou `DEBUG_FORM_VAR` je stejná jako formátovací řetězec funkce printf jazyka C. Jsou však podporovány pouze následující specifikátory formátu:

Formátovač	Výpis
`%F`	aktuální soubor (`__FILE__`)
`%n`	aktuální jméno souboru (`__FILE__` bez cesty)
`%f`	aktuální funkce (`__PRETTY_FUNCTION__`)
`%l`	aktuální řádka (`__LINE__`)
`%t`	aktuální vlákno (`pthread_self()`)
`%i`	ID tohoto výpisu

6.1.4 Makra volitelného vykonání kódu

DBGsection(id,code)

code je kód, který má být vykonán v případě, že je aktivní číselný identifikátor id. Použití může vypadat například takto:

```
DBGsection(2000,(recordList = new List));
```

Tabulku číselných identifikátorů naleznete v příloze A.

6.1.5 Použití

Ve zdrojových kódech lze použít ladící systém jak bylo popsáno výše.
JVM Lógr přeložená s ladícími informacemi umožňuje provádět výše uvedené ladící výpisy. Implicitně se řetězec číselných deskriptorů získává z proměnné prostředí LOGRDEBUG. Dále lze specifikovat pomocí parametru -d příkazové řádky o co se má tento řetězec doplnit. Pokud není proměnná prostředí definována, použije se pouze parametr z příkazové řádky.

6.2 Ladění práce s referencemi

Pro ladění práce s referencemi byly implementovány níže uvedené funkce. Tyto funkce provádějí záznam a vyhodnocování operací, které s referencemi pracují. Umožňují dostat pod kontrolu práci se zámky, vytváření resp. rušení odkazů a samozřejmě vytváření a rušení referencí samotných.

Reference lze pojmenovávat pomocí řetězců jazyka C. Funkce potom umožňují jednak okamžitý výpis operací (který lze filtrovat jménem) a dále jejich následné vyhodnocení.

Typická seance vypadá tak, že je zapnuto 'nahrávání' prováděných akcí. Testovaná část programu běží a operace se ukládají do interního logu. Nahrávání lze přerušovat.

Po ukončení nahrávání lze vypsat všechny prováděné akce. Tento výpis je obohacen o místo ve zdrojových kódech kde se reference např. zamkla, jak se hodnota zámku touto operací změnila a další informace.

Rovněž lze provést vyhodnocení změny stavu referencí v průběhu nahrávání. Výpis s touto volbou pro každou referenci, se kterou se v průběhu nahrávání pracovalo, zobrazí:

- její stav před a po nahrávání
- jak se změnily počty odkazů a hodnoty zámků

- její jméno

Výpis je možné filtrovat použitím jména reference.
Systém lze použít po aktivaci číselného identifikátoru 2000.

```
referenceDebug(action)
referenceNamedDebug(action, name)
```

Jsou funkce, které řídí systém pro ladění práce s referencemi. Parametr action obsahuje disjunkci příznaků, které určují jaká operace se má provést. Tabulka obsahuje použitelné příznaky:

Příznak	**Akce**
START_RECORDING	zapne nahrávání
STOP_RECORDING	vypne nahrávání
DESTROY_RECORD	smaže záznam
PRINT_COMPLETE_RECORD	vypíše kompletní záznam
PRINT_COMPREHENSION	vypíše velmi stručnou statistiku
PRINT_SINGLE_RECORD	vypíše všechny operace provedené s referencí daného jména během nahrávání
PRINT_CHANGE	pro každou referenci vypíše jak se změnila v průběhu nahrávání
PRINT_TRIM_CHANGE	pro každou referenci, která se v průběhu nahrávání změnila, vypíše jak se změnila
START_PRN	zapne okamžité vypisování operací
STOP_PRN	vypne okamžité vypisování operací

Pomocí parametru name lze specifikovat referenci, jíž se prováděná akce týká.

6.3 Ladění práce s pamětí

V kapitole 4 byly popsány funkce, které lze použít při ladění práce s pamětí. Funkce umožňují jak kontrolu integrity haldy jako celku, tak kontrolu integrity jednotlivých vnitřních oblastí. Kontroluje se integrita formátovacích a servisních datových struktur haldy. Volitelně lze pomocí funkcí fillFree() a checkFillFree() kontrolovat konzistenci volné paměti.
Obraz haldy resp. oblasti lze uložit do souboru nebo vypsat na stderr.

6.4 Nastavování ladícího bodu v Java metodě

Pro ladění Java metod jsou implementovány následující funkce.

```
void setBreakpoint (int n, char *className, char *methodName)
```

Tato funkce podle jména třídy a metody (v UTF8) nastaví n-tý ladící bod. Vstupní kód do Java metody je přepsán specielním kódem, který odskočí do funkce breakpoint() (viz soubor Logr/jvm/breakpoint.c).

To umožňuje nastavit ladící bod například programem gdb na tuto metodu a odtud pokračovat v trasování.
Nastavení ladícího bodu je bezpečné v 'singlethread' módu.

```
void clearBreakpoint (int n)
```

Odstraní n-tý ladicí bod.

```
void breakpoint (int stackPtr)
```

Funkce do které se 'odskočí' při dosažení ladícího bodu.

6.5 Nastavení ladícího z jazyka Java

K tomu aby bylo možné nastavovat a rušit ladící bod z běžícího programu v jazyce Java slouží třída z Logr/jvm/LogrDebugger.java. Tato třída tedy umožňuje nastavení a zrušení ladícího bodu v Java metodě z jazyka Java. Obsahuje čtyři nativní metody.

```
public static native void setBreakpoint (int n,
                                          String className,
                                          String methodName)
```

Je obdoba metody uvedené v předchozím odstavci, která nastaví n-tý ladící bod na metodu jazyka Java dle jména třídy a metody. Java program tak zajistí průchod C funkcí breakpoint(), na kterou je opět možné nastavit ladící bod programem gdb.

```
public static native void clearBreakpoint (int n)
```

Odstraní n-tý ladící bod.
Třída je doplněna o následující funkce, které umožňují měnit řetězce číselných identifikátorů.

```
public static native void addString (String s)
```

Přidá číselné identifikátory do seznamu aktivních.

```
public static native void removeString (String s)
```

Odstraní číselné identifikátory ze seznamu aktivních.

Příloha A

Ladící systém

Tabulka číselných identifikátorů ladícího systému.

Identifikátor	**Prováděný výpis**
1000	výpis natahovaných .logr souborů
1001	dosažený řádek ve zdrojovém souboru při natahování souboru .logr.
1002	průběžný výpočet velikostí vytvářených struktur při natahování .logr
1003	výpis obsahu natahovaných tříd
1004	výpis struktur souboru .logr
1005	stavění hešovacích tabulek .logr souboru
1006	vytváření statických řetězců a jejich zařazování do struktury RuntimeData
1050	hledání ve strukturách Pool při přístupu na položky a volání metod
1051	vytváření IMT hlášek
1052	volání `<clinit>`
1053	checkCast
1200	java.lang.Reflect: ladění getField & java.lang.Field
1201	java.lang.Reflect: ladění getMethod & java.lang.Method
1202	java.lang.Reflect: getConstructor & java.lang.Constructor
1100	trasování invoke

Identifikátor	Prováděný výpis
2000	práce s referencemi, řízení GC, kontrolní mechanismy haldy
2005	linked list
2006	Lógr halda
2007	servisní struktura GC
2008	kolekce GC
2009	správa referencí
5000	BCC: parametry a návratové hodnoty compile()
5005	BCC: návěští (výpis po skončení první fáze)
5010	BCC: položky .class, výpis hodnot vytvářených struktur
5020	BCC: položky .logr, výpis hodnot vytvářených struktur
5030	BCC: detailní výpis v průběhu kompilace
5040	BCC: kompletace kódu
5100	BCC: podrobný výpis chyb
6101	LKI field access
6201	LNI field access
6211	LNI invoke
6301	JNI field access
6311	JNI invoke
7100	throw relative/ throw absolute
7200	defaultClassLoaderInit
7220	getRuntimeData
7240	registerClass
7260	printHashTableContent
7280	defaultClassLoaderDeinit
7300	arrayLoaderInit
7320	getArrayRuntimeDataInternal a createRuntimeDataForOneDimension
7340	getArrayRuntimeData
7400	lNewArray
7420	lkiNewArray
7430	lNewArrayRef
7440	lMultiNewArray
7450	lkiMultiNewArray

Identifikátor	**Prováděný výpis**
7500	newJavaLangStringFromUnicode
7520	newJavaLangStringFromUtf8
7530	internStringTableInitialize
7540	internStringTableCleanup
7550	getInternStringRepresentation
7560	removeInternStringRepresentation
7570	printInternStringTable
7600	`MUTEX_INIT`, `MUTEX_DESTROY`, `COND_INIT`, `COND_DESTROY`
7610	`MUTEX_LOCK`, `MUTEX_UNLOCK`, `COND_SIGNAL`, `COND_BROADCAST`, `COND_WAIT`
7700	java.lang.Class
7750	java.lang.ClassLoader
7790	java.lang.System
7799	java.util.Resource.Bundle

Příloha B

Soubory a konvence

B.1 Přípony souborů

Soubory v instalaci JVM Lógr používají následující přípony:

Typ souboru	Přípona
zdrojový kód v jazyce Java	.java
bajtový kód jazyka Java	.class
nativní kód vykonávaný JVM Lógr	.logr
hlavičkový soubor jazyka C/C++	.h
zdrojový kód v jazyce C	.c
zdrojový kód v jazyce C++	.cc
shell skript	.sh
zdrojový kód v assembleru	.s
zdrojový kód v assembleru, který má být před přeložením zpracován C preprocesorem	.S

B.2 Názvy běžných souborů

Název	Obsah
Makefile.*	konfigurační soubory programu make
readme	preferovaný název pro soubor obsahující popis obsahu adresáře

B.3 Zdrojové soubory

B.3.1 Kódovací konvence

Typ	Konvence
Primitivní typy	celé malými písmeny.
Ostatní typy	první písmeno a první písmeno každého slova velkým, ostatní písmena malá.
Proměnné a funkce	první písmeno malé, první písmeno každého dalšího slova velké, ostatní písmena malá.

Poznámka:

Zkratky v názvech jsou slova (tedy malými písmeny, případně první velké).

Makra a konstanty	všechna písmena velká, slova oddělena podtržítkem.

B.4 Adresářová struktura

Adresář	Obsah
Logr/bcc	implementace kompilátoru a verifikátoru
Logr/doc	dokumentace
Logr/heap	implementace Lógr haldy
Logr/include	hlavičkové soubory
Logr/jni	implementace JNI metod
Logr/jvm	implementace JVM
Logr/lni	implementace LNI metod
Logr/tests	testy
Logr/util	pomocné funkce

B.4.1 Adresář Logr/include/

`Logr/include/arrayaccess.h`
Makra pro přístup do polí. Jsou využita v LKI, LNI a JNI.

`Logr/include/arrayloader.h`
Tvorba struktury RuntimeData pro pole.

`Logr/include/bcc.h`
Deklarace funkce compile() pro volání překladače.

`Logr/include/bcc_help.h`
Deklarace pomocných funkcí pro překladač.

`Logr/include/bcci.h`
Deklarace datových struktur překladače.

`Logr/include/breakpoint.h`
Nastavení ladícího bodu do Java metody.

`Logr/include/build.h`
Globální proměnná s číslem sestavení.

`Logr/include/bytecode.h`
Operační kódy bajtových instrukcí.

`Logr/include/classloader.h`
Definice hešovací tabulky zavaděče tříd (vnitřní zavaděč i zavaděč uživatelský mají stejný formát tabulky).

`Logr/include/debuglog.h`
Makra pro vnitřní ladicí systém.

`Logr/include/dfile.h`
Deklarace metody pro parsování LNI definičních souborů.

`Logr/include/elfripper.h`
Vyřezávání kódu z binárního souboru ELF.

`Logr/include/exception.h`
Seznam výjimek vyhazovaných z JVM, struktura RtExceptionInfo pro správu výjimek a asynchronních událostí, deklarace funkcí pro práci s výjimkami.

`Logr/include/fdefs.h`
Deklarace struktur pro vlastní překlad bajtového kódu.

`Logr/include/fictive.h`
Seznam položek 'fiktivní' struktury ReferencePool.

`Logr/include/function.h`
Deklarace metody překládající bajtový kód.

`Logr/include/gclist.h`
Nesynchronizovaný, dvousměrný spojový seznam.

`Logr/include/gcreference.h`
Práce s referencemi typu StatRef i InstRef.

`Logr/include/gcheap.h`
Lógr halda.

`Logr/include/gc.h, Logr/include/gc.h`
Servisní struktura a hlavička funkce, která tvoří tělo garbage collector vlákna.

`Logr/include/gcfinalize.h`
Hlavička funkce, která je tělem vlákna ukončovače objektů.

`Logr/include/hash.h`
Deklarace hešovacích funkcí použitých v hešovacích tabulkách.

`Logr/include/heapobjects.h`
Struktura objektů ukládaných do Lógr haldy.

`Logr/include/hsbtree.h`
Stavba binárních stromů.

`Logr/include/invoke.h`
Deklarace funkcí trampolíny.

`Logr/include/lconsts.h`
Globální konstanty.

`Logr/include/listclass.h`
Deklarace funkcí pro výpis struktur RuntimeData a FixedData pro ladění.

`Logr/include/ljni.h`
Struktura LJniEnv nahrazující JNIEnv pro JNI, assembler pro přípravu parametrů pro volání trampolín z JNI.

`Logr/include/lki.h`
Deklarace všech LKI funkcí.

`Logr/include/lni.h`
LNI funkce používané pro přístup do JVM z LNI metod.

`Logr/include/lnimethods.h`

Deklarace všech LNI metod.

`Logr/include/logrfile.h`
Struktura .logr souboru.

`Logr/include/logrload.h`
Deklarace funkcí a konstanty pro zavádění .logr souboru.

`Logr/include/monitor.h`
Funkce pro vstup a výstup ze synchronizačního primitiva monitor.

`Logr/include/nativethreads.h`
Makra obalující volání funkcí Linux vláken.

`Logr/include/new.h`
Deklarace funkcí pro vytváření instancí tříd a polí.

`Logr/include/pools.h`
Vyplňování struktury ReferencePool.

`Logr/include/primitiveclasses.h`
Deklarace funkcí pro vytváření instancí třídy java.lang.Class pro popis primitivních typů.

`Logr/include/reference.h`
Struktury a funkce horní vrstvy Lógr haldy, referencí a garbage collection.

`Logr/include/referencetypes.h`
Struktury referencí.

`Logr/include/threads.h`
Deklarace funkcí pro startování vláken, globální klíče pro přístup do lokálních dat vláken (thread specific), zámky.

`Logr/include/throw.h`
Deklarace funkcí pro probublávání zásobníku.

`Logr/include/utf8.h`
Deklarace převodních funkcí mezi řetězci ve formátu Unicode a UTF8.

`Logr/include/utils.h`
Deklarace různých menších funkcí: vytváření a práce s instancemi třídy java.lang.String, vytváření struktur NamePoolEntry.

`Logr/include/verifier.h`

Deklarace funkce provádějící verifikaci bajtového kódu.

`Logr/include/version.h`
Číslo verze.

`Logr/include/jdk`
Hlavičkové souboru JNI metod - vypůjčeno z JDK firmy Sun.

B.4.2 Adresář Logr/bcc/bcc.c

`Logr/bcc/bcc.c`
Načtení souboru .class.

`Logr/bcc/bcc2.c`
Tvorba datových struktur souboru .logr.

`Logr/bcc/bcc_help.c`
Pomocné funkce překladače.

`Logr/bcc/test.c`
Spouštění samotného překladače.

`Logr/bcc/bytecode.c`
Seznam instrukcí bajtového kódu a jejich vlastností. Používán verifikátorem.

`Logr/bcc/dfile.c`
Parsování LNI definičních souborů

`Logr/bcc/elfripper.c`
Načítá z relokovatelného souboru ELF spustitelný kód, jména globálních funkcí ze segmentu .text a tabulku relokací. Použito v překladači.

`Logr/bcc/function.c`
Překladač bajtového kódu.

`Logr/bcc/jnires.c`
Zjišťování názvu JNI native metody.

`Logr/bcc/native.S, Logr/bcc/native2.S`
Assemblerové prototypy instrukcí bajtového kódu.

`Logr/bcc/verifier.c`
Implementace verifikátoru.

B.4.3 Adresář Logr/heap/

`Logr/heap/gcdebug.cc`
Funkce určené pro ladění práce s referencemi. Tento soubor je vložen do reference.cc.

`Logr/heap/gcheap.cc`
Lógr halda.

`Logr/heap/gclist.cc`
Nesynchronizovaný dvousměrný spojový seznam. Obsahuje funkce pro práci se spojovým seznamem jako se zásobníkem resp. frontou.

`Logr/heap/gcollector.cc, Logr/heap/gc.cc, Logr/heap/gcfinalize.cc`
Implementace metod garbage collector vlákna a ukončovače objektů.

`Logr/heap/gcreference.cc`
Alokace a uvolňování referencí.

`Logr/heap/new.c`
Vytváření instancí tříd a polí (polí primitivních typů, polí referencí, vícedimenzionálních polí).

`Logr/heap/reference.cc`
Hlavní a nejdůležitější soubor pracující s Lógr haldou.. Obsahuje funkce nejvyšší úrovně pro práci s referencemi, haldou a řízení garbage collection.

`Logr/heap/referencedefines.c`
Tento soubor je spuštěn v době kompilace. Generuje soubor referencedefines.h, v němž jsou definovány konstanty, které se mohou měnit. Například relativní posuny ve strukturách, části kódu předpočítávající konstanty užívané v makrech.

`Logr/heap/utils.c`
Různé menší funkce: vytváření a práce s instancemi třídy java.lang.String, vytváření struktur NamePoolEntry

B.4.4 Adresář Logr/jni/ libjava/

Implementace Java metod pomocí JNI rozhraní.

B.4.5 Adresář Logr/jvm/

`Logr/jvm/LogrDebugger.java`
Spouštění ladících výpisů přímo z Java programu.

`Logr/jvm/arrayaccess.c`
LKI funkce pro přístup do polí - čtení a zápis prvků pole, zjišťování velikosti pole. Většina funkcí je implementována pomocí přístupových maker z arrayaccess.h

`Logr/jvm/arrayloader.c`
Vytváření struktur RuntimeData, FixedData a instance třídy java.lang.Class popisující konkrétní typ pole. Kostra struktury je vytvořena funkcí loadLogrFile zavádějící soubor .logr. Z koster struktur jsou následně vytvořeny struktury plnohodnotné.

`Logr/jvm/breakpoint.c`
Nastavení ladícího bodu do Java metod.

`Logr/jvm/classloader.c`
Vytváření struktur RuntimeData, FixedData a instance třídy java.lang.Class popisující konkrétní třídu. Struktury RuntimeData a FixedData jsou vytvořeny funkcí loadLogrFile ze souboru .logr. Pokud soubor .logr nebyl dosud vytvořen, je volána funkce compile(), která soubor .logr vyprodukuje ze souboru .class. Struktura RuntimeData je zařazena do hešovací tabulky (JVM si pomocí hešovací tabulky pamatuje, které třídy již byly zavedeny a nezavádí je opakovaně).

`Logr/jvm/exception.c`
Vytváření a testování výjimek.

`Logr/jvm/fictive.c`
Funkce pro správu 'fiktivní' struktury ReferencePool pro přístup na položky a metody zevnitř JVM.

`Logr/jvm/fieldaccess.c`
LKI funkce pro přístup k položkám objektů.

`Logr/jvm/javaparse.c`
Zpracování příkazové řádky.

`Logr/jvm/jni.c`
Implementace JNI rozhraní.

`Logr/jvm/jvmstart.c`
Startování JVM, volání veškerých inicializací a startování prvního Java vlákna.

`Logr/jvm/listclass.c`
Výpis struktur RuntimeData a FixedData pro ladění, ladění trampolín.

`Logr/jvm/lki.c`
Implementace některých LKI funkcí, které si nezasloužily vlastní zdrojový soubor.

`Logr/jvm/lni.c`
Implementace některých funkcí LNI rozhraní.

`Logr/jvm/logrload.cc`
Natahování .logr souboru, stavba struktur FixedData a RuntimeData.

`Logr/jvm/nativethreads.c`
Volání funkcí Linux vláken.

`Logr/jvm/pools.c`
Vyplňování struktur ReferencePool, CheckCast, stavba tabulky IMT.

`Logr/jvm/stublist.c`
Seznam LKI a LNI funkcí.

`Logr/jvm/stubs.S`
Trampolíny.

`Logr/jvm/testloader.c`
Test natahování tříd.

`Logr/jvm/threads.c`
Inicializační funkce vláken - tyto funkce jsou spouštěny voláním `pthread_create()` a obsahují inicializace pro běh Java vlákna.

`Logr/jvm/throw.c`
Vybublávání výjimek. Obsluha vyhozené výjimky se hledá v aktuální metodě. Není-li v aktuální metodě nalezena, hledá se dále v metodách uložených v zásobníku volání výše.

B.4.6 Adresář Logr/lni/

`Logr/lni/description/`
Adresář s popisnými soubory pro LNI.

`Logr/lni/java_lang:`
Implementace native metod pomocí LNI.

B.4.7 Adresář Logr/util/

`Logr/util/asmdefines.c`
Generuje hlavičkový soubor s konstantami offsetů do C struktur, pro použití z assembleru.

`Logr/util/crc.c`
Výpočet CRC32.

`Logr/util/debuglog.c`
Ladící systém.

`Logr/util/hash.c`
Hešovací funkce použitá v hešovacích tabulkách. Počítá hešovací kódy jmen tříd a jmen položek tříd.

`Logr/util/hsbtree.c`
Stavba binárních stromů.

`Logr/util/inlines.c`
Obsahuje funkce, které jsou inline v .h souborech. Pokud se nepřekládá s optimalizací, musí být všechny tyto funkce přeloženy v .o souboru.

`Logr/util/utf8.c`
Převodní funkce mezi řetězci ve formátu Unicode a UTF8, výpočet délky rozbaleného a zabaleného řetězce.

B.5 CVS

Pro správu zdrojových souborů byl zvolen systém CVS - „Concurrent Versions System“, který umožňuje a podporuje práci s mnoha zdrojovými soubory více programátory současně. Umožňuje správu verzí jednotlivých zdrojových souborů i celého projektu, odstraňování konfliktů, pokud více členů týmu pracuje se stejným souborem, vzdálenou práci s archivem souborů. Viz 'man cvs'.

Literatura

[Mikro94] Brandejs Michal: *Mikroprocesory INTEL - Pentium a spol.*, Grada, 1994

[Nut96] Flanagan David: *Java in a Nutshell*, O'REILLY, 1996

[Lang96] Gosling James, Joy Bill, Steele Guy: *The Java Languague Specification*, Addison-Wesley, 1996

[Refl97] JavaSoft: *Java Core Reflection, API and specification*, 1997

[Ser97] JavaSoft: *Java Object Serializaton Specificaton*, 1997

[RMI97] JavaSoft: *Java Remote Method Invocation Specification*, 1997

[Inn97] JavaSoft: *Inner Classes Specification*, 1997

[JNI97] Liang Sheng: *Java Native Interface Specification*, JavaSoft, 1997

[Spec97] Lindholm Tim, Yellin Frank: *The Java Virtual Machine Specification*, Addison-Wesley, 1997

[Nat96] Yellin Frank: *The Java Native Code API*, 1996

Zdroje

Projekt Kaffe	http://www.kaffe.org
Linux port JDK 1.2	http://www.blackdown.org, http://java.sun.com
JDK 1.0beta	http://java.sun.com
BISS-AWT	http://www.biss-net.com