Arquitetura da Máquina Virtual Java

James de Souza
Instituto de Computação
Universidade Estadual de Campinas
RA 991899
jamesdesouza@gmail.com

ABSTRACT

Os diferentes tipos de arquiteturas de computadores e sistemas operacionais muitas vezes se tornam um impecílio para a portabilidade de programas. Dependendo da linguagem de programação utilizada, uma solução possível é utilizar um compilador para que o programa funcione na nova plataforma, mas muitas vezes é necessário realmente alterar o código-fonte para se obter o resultado esperado. O conceito das máquinas virtuais surge para tentar resolver o problema de portabilidade. Aumentando um nível de abstração, o sistemas agora passam a implementar código para a máquina virtual que tem a tarefa de converter os dados para a arquitetura real. Esse trabalho tem o objetivo de descrever a máquina virtual Java, mostrando a arquitetura, os tipos de dados, o conjunto de instruções e o modo como a máquina virtual trabalha em tempo de execução.

1. Introdução

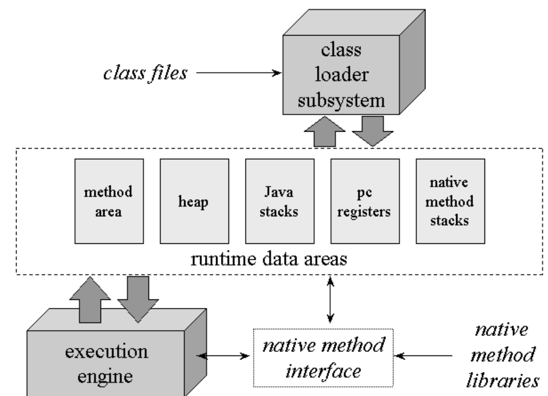
A máquina virtual Java é um dos principais componentes da plataforma Java e permite que o mesmo código possa ser executado em várias plataformas sem a necessidade de recompilação. A JVM (Java virtual machine) emula uma máquina real possuindo um conjunto de instruções próprio e atua em áreas de gerenciamento de memória. A JVM possui uma especificação que pode ser implementada nas diversas arquiteturas. Essa especificação visa não atender a nenhum tipo de tecnologia em específico, seja de hardware ou de sistema operacional. A máquina virtual não trabalha diretamente com a linguagem de programação Java, mas sim com o arquivo de formato class. Esse tipo de arquivo contém os bytecodes, que são as intruções para a máquina virtual e uma tabela de símbolos.

A JVM pode ser descrita através de quatro itens básicos: subsistemas, áreas de memória, tipos de dados e instruções.

2. Estrutura da JVM

O arquivo class contém as instruções que devem ser executadas na JVM. A implementação da JVM consiste basicamente na interpretação do arquivo class. Técnicas de otimização e gerenciamento de memória variam de implementação para implementação. A Figura 1 mostra uma visão geral com os principais componentes que fazem parte da JVM. Esse capítulo procura descrever cada um desses componentes.

Figura 1. Visão geral da arquitetura da JVM [2]



2.1. O class loader

O class loader é responsável por localizar e importar os dados binários para as classes. Ele também verifica a corretude das classes, inicializa a memória para as variáveis de classe e ajuda na resolução de símbolos. A JVM possui um bootstrap class loader que sabe como carregar as classes da API Java. A especificação da JVM não define como localizar essas classes e assim fica a cargo de cada implementação da JVM decidir esse procedimento. Um exemplo é o caso da implementação da JVM para Windows 98, da JDK1.1 da Sun, que busca as classes a partir de uma variável de ambiente chamada CLASSPATH.

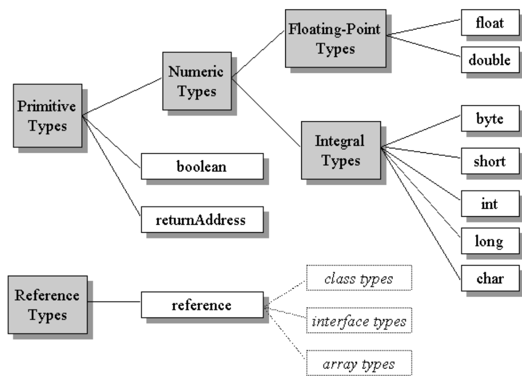
2.2. Tipos de dados

Os tipos de dados suportados pela JVM seguem a linguagem de programação, sendo do tipo primitivo ou referência. A JVM espera que a validação de tipos já esteja pronta quando os dados são executados. Essa validação normalmente é realizada pelo compilador.

A JVM possui tipos específicos de instruções para operar com diferentes tipos de dados. Por exenplo, existe a instrução iadd para soma de dois operandos do tipo int, enquanto que para a soma de dois operandos do tipo long, existe a instrução ladd.

A JVM suporta objetos. Uma instância de uma classe ou um vetor são considerados objetos e são manipulados pelo tipo de dados referência. A Figura 2 mostra as divisões e subdivisões dos tipos de dados suportados pela JVM.

Figura 2. Tipos de dados da JVM [2]



Os tipos primitivos da JVM podem ser do tipo numérico, do tipo booleano e o tipo returnAddress. Os tipo numéricos são subdivididos em tipos inteiros e em pontos flutuantes.

Dentre os tipos da categoria inteiros temos o tipo byte, de 8 bits, com sinal e complemento de 2. O tipo short possui 16 bits, sinal e complemento de 2. O tipo int possui 32 bits, sinal e complemento de 2. O long possui 64 bits, com sinal e complemento de 2. Já o tipo char possui 16 bits, sem sinal e representa caracteres Unicode. A Tabela 1 mostra os limites de valores para cada tipo.

Tabela 1. Limite de valores para os tipos inteiros

Tipo	Limite Inferior	Limite superior
byte	-128	127
short	-32768	32767
int	-2147483648	2147483647
long	-9223372036854775808	9223372036854775807
char	0	65535

Os pontos flutuantes são representados por dois tipos, o float e o double. Possuem tamanhos respectivos de 32 e 64 bits e seguem o padrão IEEE 754. O padrão IEEE 754 contempla números positivos e negativos, infinito negativo e positivo, o valor NaN (Not a Number) que representa um valor inválido, por exemplo, o resultado de uma divisão por zero.

Os valores do tipo booleano podem ser true ou false. Embora definido, existe pouco suporte para ele dentro da JVM. Tipos booleanos da linguagem são convertidos para o tipo int da JVM.

Os valores do tipo returnAddress representam ponteiros para instruções da JVM, como jsr, ret e jsr_w, sendo o único tipo primitivo que não está relacionado com os tipos da linguagem de programação Java.

A unidade de tamanho base para os tipos de dados é conhecida como palavra que é escolhida pelo implementador da JVM. Uma palavra deve ter no mínimo 32 bits e deve suportar os tipos byte, short, int, char, float, returnAddress e o tipo referência. Duas palavras precisam suportar os tipos long e double. As variáveis locais e outros elementos são baseadas em termos de palavras.

2.3. Área de dados em tempo de execução

Existem áreas de dados relacionadas a máquina virtual em geral e áreas específicas de cada thread.

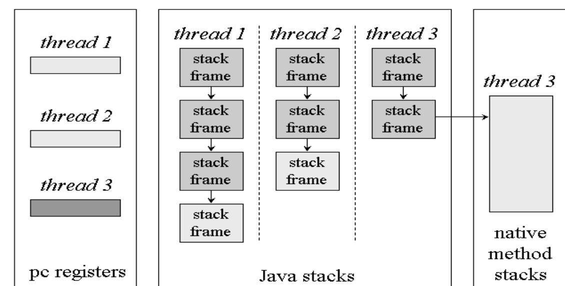
A máquina virtual suporta várias threads e cada thread possui o seu registrador pc (program counter). O pc possui o tamanho de uma palavra. Conforme a execução de um método de uma thread, o registrador pc contém o endereço da instrução atual. Um endereço pode ser um ponteiro nativo ou pode ser um offset em relação ao começo dos bytecodes de um método. Se a thread está executando um método nativo o valor do pc é indefinido.

Cada thread possui uma pilha associada. A especificação da JVM permite que a pilha seja de tamanho fixo ou expandida dinamicamente conforme a execução. Caso seja necessário uma pilha maior do que a JVM permite é lançada uma exceção do tipo StackOverflowError. No caso da JVM ser expandida dinamicamente e não haver espaço suficiente de memória, uma exceção OutOfMemoryError é lançada.

A pilha guarda o estado do método invocado pela thread. O estado é formado por variáveis locais, parâmetros, valor de retorno e os cálculos realizados na execução do método. A pilha é muito utilizada já que a JVM não possui registradores para guardar valores intermediários durante a execução dos métodos. As pilhas são formadas por frames, que serão detalhados na seção 2.3.

A JVM oferece em sua arquitetura, suporte a execução de métodos nativos. Os métodos nativos podem acessar a área de dados da JVM mas pode também usar recursos do sistema nativo. Por sua própria natureza, normalmente são dependentes da plataforma que rodam. Quando existe a chamada de um método nativo, a JVM não cria um novo frame na pilha da thread e chama o método nativo diretamente. A Figura 3 mostra os componentes de memória em tempo de execução.

Figura 3. Memória em tempo de execução [2]



Uma outra área da memória da JVM é o heap. O heap é uma area de dados em tempo de execução onde é alocada memória para todas instâncias de classes e vetores. O heap é criado na inicialização da JVM. O tamanho pode variar durante a execução, conforme a implementação da JVM.

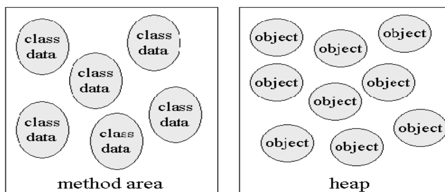
Dentro de uma aplicação existe apenas um heap. Duas aplicações rodando na mesma máquina virtual possuem heaps diferentes, já que considera-se que as aplicações estão sendo executadas em suas próprias máquinas exclusivas.

Existe instrução para alocar memória no heap, mas não existe instrução para realizar a desalocação. Para a desalocação do heap existe o garbage collector, que é um processo responsável por verificar objetos que não estão sendo mais utilizados e liberar o espaço da memória. A especificação da JVM não obriga o uso do garbage collector, ou seja, uma determinada implementação da JVM poderia escolher ter um tamanho fixo do heap e lançar uma exceção OutOfMemory quando esse espaço estivesse totalmente ocupado.

A JVM possui também uma área de métodos que é compartilhada por todas as threads. A área de métodos é criada na inicialização da máquina virtual.

A Figura 4 mostra a área de métodos e o heap.

Figura 4. Área de métodos e o heap [2]



A JVM mantém uma estrutura de dados chamada de Constant Pool. O constant pool é basicamente uma tabela de símbolos como em linguagens de programação convencional.

2.4. Frames

Um frame é utilizado para gravar dados e resultados parciais, executar acoplação dinâmica, retornar valores para métodos e lançar exceções.

Um frame é criado na chamada de um método e em uma determinada thread existe um único frame ativo. Ao se chamar um novo método, o frame atual passa o controle para o novo frame e volta a ser ativo ao receber o retorno do método.

Cada frame possui seu vetor de variáveis locais, que são endereçadas por índice. Um frame possui também uma pilha associada chamada de *operand stack*. O tamanho da operand stack é determinado em tempo de compilação. Essa pilha é vazia na criação do frame e vai sendo preenchida no decorrer da execução do método. É utilizada na preparação da chamada de outros métodos e para receber valores de retorno. A maioria das operações realizadas sobre essa pilha requer tipos de dados bem definidos pois existem poucas instruções que não se preocupam com o tipo dos dados.

Na finalização de um método o frame atual é utilizado para restaurar o estado do método que o invocou, ou seja, as variáveis locais, a operand stack e o pc, que é incrementado para a próxima instrução.

2.5. Exceções

Na JVM um catch ou finally da lingugem de programação é representado por um exception handler.

Um exception handler está associado a um determinado conjunto de linhas do código, possui um tipo de exceção associada e o lugar do código associado a ela.

Quando uma exceção é lançada, a JVM procura por um handler para a exceção e se encontra, salta para o código associado ao handler. Caso não encontre nenhum handler, o frame atual é encerrado, o controle é tranferindo para o frame que realizou a chamada e segue a cadeia de chamadas até que se encontre um handler. Em último caso, não havendo nenhum handler, a thread é terminada. Dentro do arquivo class existe uma tabela com os exception handlers para cada método.

3. Conjunto de instruções

Uma instrução da JVM consiste de um opcode de um byte e pode ter argumentos e dados que serão usados na operação. A opção de ter a instrução em apenas um byte permite maior simplicidade e limita o número de instruções.

A maioria das instruções possui tipos específicos associados, ou seja, tipos de dados diferentes podem possuir instruções diferentes para realizar uma determinada operação. O opcode mnemônico possui uma letra associada ao tipo do dado: i para int, l para long, s para short, b para byte, c para char, d para double e a para uma referência.

Existem três opcodes reservados para uso interno da JVM. Os dois primeiros, números 254 e 255, impdep1 e impdep2, provêem a capacidade para implementações de funcionalidade específica para software e hardware, respectivamente. O terceiro opcode reservado, 202, possui o mnemônico breakpoint para possível verificações de programas.

Abaixo alguns exemplos de instruções da JVM:

ladd – soma dois valores do tipo long. Os dois operandos são retirados no topo da pilha. O resultado é colocado no topo da pilha.

lload – carrega o valor de uma variável local para o topo da pilha.

lstore – carrega o valor do topo da pilha para a variável determinada pelo índice passado como parâmetro.

lshl - executa um deslocamento a esquerda do operando 1 de acordo com os 6 bits mais baixos do operando 2. O resultado é colocado no topo da pilha.

i2b – converte um int para um byte.

iaload – carrega um valor int de um vetor. O valor da referência do vetor e o índice são retirados do topo da pilha. O valor recuperado do vetor é colocado topo da pilha.

imul – multiplica dois valores do tipo int. Os dois valores são retirados e o resultado é colocado no topo da pilha.

ddiv – realiza a divisão entre dois valores do tipo double. O resultado é colocado no topo da pilha.

dreturn – retorna um valor double de um método. O método deve ter como retorno um valor do tipo double.

dsub – realiza a subtração entre dois valores do tipo double. O resultado é colocado na pilha.

3.1. Instruções do tipo load e store

As instruções do tipo load e store têm o papel de transferir dados entre as variáveis locais e a pilha do frame (operand stack).

O load tem a função de carregar o valor de uma variável para o topo da pilha. As instruções do tipo store tem a função de carregar valores do topo da pilha para variáveis locais. Exitem funções também para carregar constantes para o topo da pilha.

Uma instrução do tipo load possui várias formas, dependendo do tipo dos dados que serão utilizados na operação. Mesmo uma operação para um tipo específico de dado, por exemplo, iload, possui variações. Por exemplo, iload_0, iload_1, ou seja, a forma iload_<n>, que representa a operação iload com o operando n.

3.2. Instruções aritméticas

As instruções aritméticas basicamente trabalham com a pilha (operand stack), retirando os valores e colocando o resultado no topo da pilha. As instruções aritméticas dividem-se naquelas relacionadas a valores inteiros e naquelas relacionadas com pontos-flutuantes. Na JVM dentro dessas categorias ainda existe a subdivisão por tipos de dados.

As instruções aritméticas são:

add: iadd, ladd, fadd, dadd

subtract: isub, lsub, fsub, dsub

multiply: imul, lmul, fmul, dmul.

divide: idiv, ldiv, fdiv, ddiv.

remainder: irem, lrem, frem, drem.

negate: ineg, lneg, fneg, dneg.

shift: ishl, ishr, iushr, lshl, lshr, lushr.

bitwise OR: ior, lor.

bitwise AND: iand, land.

bitwise exclusive OR: ixor, lxor.

local variable increment: iinc.

comparison: dcmpg, dcmpl, fcmpg, fcmpl, lcmp.

A JVM não indica overflow durante uma operação entre inteiros. Apenas as operações idiv, ldiv, irem e lrem lançam uma exceção se o divisor for zero.

As instruções de ponto flutuante não geram exceções. Quando ocorre um overflow, é produzido um valor infinito com sinal. Uma operação com underflow produz um valor não normalizado ou um zero sinalizado. Uma operação que produz um valor indefinido resulta em um NaN.

3.3. Instruções para conversão de dados

A JVM suporta instruções para conversões de tipos numéricos: i2l, i2f, i2d, l2f, l2d, f2d, que representam respectivamente, conversão de int para long, int para float, int para double, long para float, long para double e float para double. Conversões de tipos int e long para os tipos float e double podem ocasionar perda de precisão. O arredondamento utilizado para esse caso é o modo de arredondamento para o valor mais próximo de acordo com a especificação IEEE 754.

3.4. Criação e manipulação de objetos

A JVM possui instruções para manipulação de objetos como new para criar uma instância da classe e newarray para criação de vetores. Não existe instrução para desalocação de um objeto.

Existem também as instruções para acesso a campos de classes e de instâncias de classes, como getfield, putfield, getstatic e putstatic.

As operações relacionadas com load de vetores na pilha são: baload, caload, saload, iaload, laload, faload, daload, aaload e as operações de store relacionadas: bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore.

Ainda referente a manipulação de objetos, temos as instruções arraylength para obtenção do tamanho de vetores e as instruções instanceof e checkcast para verificação das propriedades da classe.

3.5. Gerenciamento da pilha (operand stack)

As instruções de gerenciamento da pilha são muito importantes devido ao funcionamento das diversas instruções que dependem do uso da pilha.

Algumas instruções abaixo:

pop – retira o valor que está no topo da pilha.

dup – faz uma cópia do valor que está no topo da pilha e também coloca o valor no topo da pilha.

dup_x1 – faz uma cópia do valor que está no topo da pilha e coloca o valor duas posições abaixo do topo da pilha.

swap – troca de posição os dois valores que estão no topo da pilha.

3 .7. Instruções de controle

As operações de controle podem mudar a sequência de execução de um conjunto de instruções.

Existem as instruções que transferem o controle de maneira condicional ou de maneira incondicional.

As instruções condicionais são: ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpn, tableswitch, lookupswitch.

As instruções de controle incondicionais são: goto, goto_w, jsr, jsr_w, ret.

3 .8. Instruções de chamada e retorno de métodos

As instruções para chamada de métodos são listadas abaixo.

- invokevirtual - é a forma padrão para a chamada de um método

- invokeinterface - realiza a chamada de um método de uma interface procurando o método implementado por um objeto particular de um ambiente de execução

- invokespecial - realiza a chamada de métodos especiais como métodos privados ou métodos da super classe

- invostatic - faz a chamada de métodos de classe (static)

As instruções para retorno são divididas dependendo do tipo do retorno em ireturn, para boolean, byte, char, short e int; as demais instruções são lreturn, freturn , dreturn e areturn. Para métodos do tipo void existe a instrução return.

3 .9. Sincronização

A JVM permite a sincronização de métodos ou sequência de instruções.

A sincronização de sequência de instruções é realizada através de duas instruções: monitorenter e monitorexit.

O compilador deve garantir a execução de um monitorexit para cada monitorenter, devendo prover o tratamento mesmo que o método não seja executado em sua sequência normal. Assim devem ser gerados também exception handlers para o caso de haver alguma exceção dentro do código.

4. Limitações da JVM

O formato do arquivo class está associado diretamente ao funcionamento da JVM. Assim para um melhor entendimento da JVM e suas limitações é importante conhecer um pouco da estrutura desse tipo de arquivo.

Um arquivo class começa com um número chamado magic que identifica o formato de arquivo class. Esse número corresponde ao valor 0xCAFEBABE.

Os valores minor_version e major_version determinam se uma JVM pode suportar um arquivo class se a versão do formato desse arquivo estiver dentro da sua faixa suportada.

Algumas flags de acesso, access_flags, determinam algumas propriedades da classes contidas no arquivo, como ACC_FINAL, que quando contendo um determinado valor (0x001), não permite que a classe tenha subclasses; ACC_INTERFACE distingue entre uma interface e uma classe; ACC_ABSTRACT diz se uma classe pode ser instanciada ou não.

O campo field_count contém o número de variáveis de classe ou de instância. A infomações sobre as variáveis estão no vetor fields [].

O campo methods[] é um vetor de uma estrutura que contém todas as informações dos métodos da classe ou da interface. Esse vetor engloba apenas os métodos da classe e não inclui os métodos das super classes.

Esses são apenas poucos exemplos dos dados que estão contidos em um arquivo do tipo class.

O fato das propriedades dos arquivos do tipo class possuírem tamanho fixo indicam as limitações da JVM:

- o número máximo de variáveis locais em um frame é limitado a 65535

- o número de campos é determinado pelo campo fields_count que suporta no máximo o valor numérico 65535

- a operand stack também possui o limite de 65535 valores. Esse valor é determinado pelo campo max_stack

- o limite do número de dimensões de um vetor é 255

- o número de parâmetros em um método também está limitado a 255

- o nome dos métodos, campos e outras constantes do tipo string são limitados a 65535 caracteres de 16 bits

5. Exemplos de bytecode

Nessa seção é apresentado uma sequência de instruções da JVM. É feita uma comparação entre código da linguagem de programação Java e o bytecode gerado [2].

```
class Act {
    public static void doMathForever() {
        int i = 0;
        for (;;) {
            i += 1;
            i *= 2;
        }
    }
}
    // Bytecode stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
    // método void doMathForever()
    // Coluna esquerda: offset da instrução em relação ao início do método
    // Coluna do centro: mnemônico das instruções e operandos
    // Coluna direita: comentário
    0  iconst_0      // 03
    1  istore_0      // 3b
    2  iinc 0, 1     // 84 00 01
    5  iload_0       // 1a
    6  iconst_2      // 05
    7  imul          // 68
    8  istore_0      // 3b
    9  goto 2        // a7 ff f9
```

6. Otimizações

Inicialmente a JVM foi considerado lenta pelo fato dos bytecodes serem interpretados. Atualmente existem algumas técnicas que ajudam a melhorar esse desempenho.

Uma das possibilidades é de executar os bytecodes em hardware sem a necessidade de interpretação pela JVM e compilação para uma determinada CPU. Algumas empresas têm trabalhado em torno da arquitetura picoJava. Essa arquitetura além de prover os bytecodes padrões da linguagem Java provê algumas outras instruções de bytecode para suportar códigos C/C++, também funções de baixo nível de hardware, como escrita na memória, controle de cache e acesso a registradores.

Outra possibilidade de otimização, que é muito utilizada na prática, são os compiladores Just-In-Time (JIT).

O JIT é um gerador de código que converte bytecodes em código de máquina nativo. Isso possibilita uma execução muito mais rápida comparando com a execução que passa pelo interpretador.

Quando um método é chamado pela primeira vez o JIT converte os bytecodes para código nativo e grava esse resultado.

Existem algumas técnicas mais sofisticadas, por exemplo, onde a JVM passa a monitorar a execução de código e coletar estatísticas para a geração de código nativo mais otimizado. A otimização passa a ser maior pois o código nativo é gerado de acordo com a máquina em específico podendo aproveitar melhor seus recursos, como por exemplo, o conjunto de registradores.

7. Conclusão

A arquitetura da JVM foi projetada para ser simples e de fácil implementação.

A independência de hardware e sistema operacional permite uma grande flexibilidade e ao mesmo abre espaço para implementações específicas.

A especificação da JVM não impõe muitas regras, deixando que os implementadores possam tratar alguns problemas da melhor maneira para a arquitetura em específico.

A utilização de bytecodes, de certa maneira desvinculado da linguagem de programação, permite que outras linguagens também possam ser executadas na JVM.

Inicialmente considerada lenta, por ser interpretada, já apresenta soluções como implementação em hardware e compilação com compiladores mais sofisticados e voltados a arquiteturas específicas.

8. Bibliografia

[1] Sun Microsystems, Inc. The Java Virtual Machine Specification*, 1999*.

[2] B. Venners. Inside the Java Virtual Machine. Artima Software, Inc., 2003.

[3] Tom R. Halfhill, How To Soup Up Java, 1998.

[4] Austin C. e Pawlan M., Advanced Programming for the Java 2 Platform, 1999.

[5] Apache Software Foundation, Byte Code Engineering Library, 2003