JAVA

Třídy a classloader

Přehled

- třídy se do VM natahují dynamicky
  - lze změnit
    - odkud se natahují
    - jak se natahují
- java.lang.ClassLoader
  - VM používá classloadery pro natahovaní tříd
- každá třída je natažena nějakým classloaderem
  - Class.getClassLoader()
- výjimka
  - třídy pro pole
    - vytvářeny automaticky, když je potřeba
    - Class.getClassLoader() vrací stejnou třídu jako pro elementy pole

Postup natahování tříd do VM

1. natažení třídy
   - classloader
   - může způsobit výjimky (potomci LinkageError)
     - ClassCircularityError
     - ClassFormatError
     - NoClassDefFoundError
   - může nastat i OutOfMemoryError
2. "linking"

Postup natahování tříd do VM

2. "linking"
   - verification
     - test, zda třída odpovídá Java Virtual Machine Specification
     - výjmky (potomci LinkageError)
       - VerifyError
   - preparation
     - vytvoření static atributů
       - ne inicializace
     - OutOfMemoryError
   - resolution
     - symbolické reference na další třídy
     - výjimky – IncompatibleClassChangeError a potomci
       - IllegalAccessError, InstantiationError, NoSuchFieldError, NoSuchMethodError, UnsatisfiedLinkError

Postup natahování tříd do VM

3. inicializace
4. vytvoření nové instance

Třída a classloader

- třída je ve VM určena nejen jménem ale i classloaderem
  - jedna třída natažená různými classloadery => z pohledu VM dvě různé třídy

- každý classloader má předka (teď ne dědičnost)
  - výjimka
    - bootstrap classloader
      - nemá předka
  - pokud se nespecifikuje => systémový classloader
- při natahování třídy classloader nejdříve deleguje natažení na předka a pokud ten třídu nenalezl, natáhne classloader třídu sám

Třída a classloader

- pokud jsou při natahování třídy potřeba další třídy, natahují se **stejným** classloaderem

Vlastní classloader

- potomek od java.lang.ClassLoader
  - předefinovat metodu findClass()

```
class MyClassLoader extends ClassLoader {
   public Class<?> findClass(String name) {
     byte[] b = loadClassData(name);
     return defineClass(name, b, 0, b.length);
   }

   private byte[] loadClassData(String name) {
     . . .
   }
}
```

- použití

```
Class.forName("trida", true, new MyClassLoader());
```

nebo

```
new MyClassLoader().loadClass("trida");
```

Vlastní classloader

- lze předefinovat i metody loadClass()
  - nevhodné
  - činnost metody
    - findLoadedClass()
      - test, zda je třída už natažená
    - deleguje natažení na rodičovský classloader
    - findClass()
    - resolveClass()

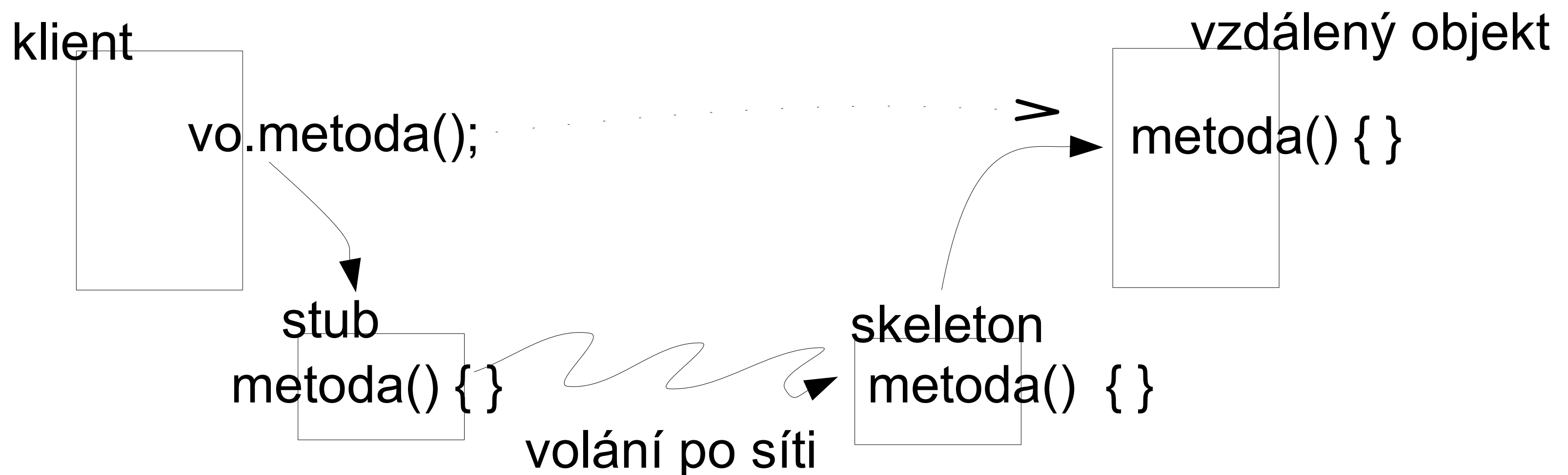
Příklady použití

- natažení tříd z různých zdrojů – např. ze sítě

```
private byte[] loadClassData(String name) throws
  ClassNotFoundException {
  URL url = new URL("........");
  URLConnection con = defURL.openConnection();
  InputStream is = con.getInputStream();
  ByteArrayOutputStream bo = new ByteArrayOutputStream();
  int a = is.read();
  while (a != -1) {
    bo.write(a);
    a = is.read();
  }
  is.close();
  byte[] ar = bo.toByteArray();
  bo.close();
  return ar;
}
```

Příklady použití

- RMI
  - automatické stahování stubů po síti



Příklady použití

- oddělení jmenných prostorů
  - *(od Java 9 – lepší řešení použít moduly)*
  - př. aplikační server
    - jedna VM
    - "aplikace" jsou natahovány vlastními classloadery
      - mohou využívat různé verze stejných knihoven
        - (různé třídy se stejnými názvy)
  - nastávají problémy, pokud spolu "aplikace" chtějí přímo komunikovat
    - řešení – pro komunikaci se používají interfacy a třídy natažené společným předkem (classloaderem)
    - nevhodné řešení – použití reflection API

Načítání dalších „zdrojů“

- classloader může načítat „cokoliv“
- načítání se řídí stejnými pravidly

- metody

```
URL getResource(String name)

InputStream getResourceAsStream(String name)

Enumeration<URL> getResources(String name)
```

Java

Service loader & provides

Přehled

- mějme nějaký interface pro nějakou „službu“
  - př. javax.xml.parsers.DocumentBuilderFactory
- různí „dodavatelé“ implementace služby
  - pokud bychom chtěli použít v programu, museli bychom jméno implementace mít napevno v kódu
    - při změně dodavatele nutno změnit kód
  - lépe => použijeme ServiceLoader<S>

Použití

- implementaci zabalit do JAR
  - přidat do JAR soubor META-INF/services/jmeno.interfacu.služby
    př: `META-INF/services/javax.xml.parsers.DocumentBuilderFactory`
  - v něm co řádek, to jméno třídy implementující interface
- v kódu
  - sl = ServiceLoader.load(interface.class, classloader);
  - sl.iterator()

- ServiceLoader je až od JDK 6
  - META-INF/services se používá už od JDK 1.3
    - bylo nutné si obdobu ServiceLoaderu napsat ručně

- explicitně podporován moduly (od Java 9)
  - definice v module-info.java

Bytecode

Bytecode

- The Java Virtual Machine Specification,

- https://docs.oracle.com/javase/specs/

Formát class souboru

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Formát class souboru

- magic
  - 0xCAFEBABE
- version

Java SE	class file format version range
1.0.2	45.0 ≤ *v* ≤ 45.3
1.1	45.0 ≤ *v* ≤ 45.65535
1.2	45.0 ≤ *v* ≤ 46.0
1.3	45.0 ≤ *v* ≤ 47.0
1.4	45.0 ≤ *v* ≤ 48.0
5.0	45.0 ≤ *v* ≤ 49.0
6	45.0 ≤ *v* ≤ 50.0
7	45.0 ≤ *v* ≤ 51.0
8	45.0 ≤ *v* ≤ 52.0
9	45.0 ≤ *v* ≤ 53.0
10	45.0 ≤ *v* ≤ 54.0
11	45.0 ≤ *v* ≤ 55.0

Kód

- zásobníkový assembler

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) { ; }
}


Method void spin()
 0     iconst_0      // Push int constant 0
 1     istore_1      // Store into local variable 1 (i=0)
 2     goto 8        // First time through don't increment
 5     iinc 1 1      // Increment local variable 1 by 1 (i++)
 8     iload_1       // Push local variable 1 (i)
 9     bipush 100    // Push int constant 100
11     if_icmplt 5   // Compare and loop if less than (i<100)
14     return        // Return void when done
```

Instrukce

- opcodes
- velikost – 1 byte
  - max 256 možných instrukcí
    - ne všechny jsou použité
- kategorie instrukcí
  - load a store (aload_0, istore,...)
  - aritmetické a logic (ladd, fcmpl,...)
  - konverzie typů (i2b, d2i,...)
  - vytváření objektů a manipulace s nimi (new, putfield)
  - manipulace s operandy na zásobníku (swap, dup2,...)
  - řízení běhu (ifeq, goto,...)
  - volání metod a návrat (invokespecial, areturn,...)

Instrukce

- invokedynamic
  - od Java 7
  - podpora pro překlad dynamických jazyků do Java bytekódu

  - od Java 8 použita i pro překlad lambda výrazů

Bytecode

- nástroje pro práci s byte kódem
  - ASM, BCEL, SERP, ...
- ASM
  - http://asm.ow2.org/
    - manipulace s bytekódem
    - vytváření nových tříd
    - upravování existujících

JNI
(Java Native Interface)

Přehled

- integrace nativního kódu (v C, C++,...) do Javy
- integrace Java kódu do nativního kódu

- obvyklé použití
  - platformě závislé operace

Příklad

```
class HelloWorld {
    public native void displayHelloWorld();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

Příklad

- javac -h <output_dir> HelloWorld.java
    - (starý způsob – javah -jni HelloWorld)
  - > HelloWorld.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    displayHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
  (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

Příklad

- naimplementovat nativní metodu

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env,
                                        jobject obj) {
    printf("Hello world!\n");
}
```

Příklad

- přeložit nativní kód (HelloWorld.c)
  - výsledek – sdílená knihovna (.so, .dll)
  - UNIX
    - ```
      gcc -shared -Wall -fPIC -o libhello.so
      -I/java/include -I/java/include/linux HelloWorld.c
      ```
  - Windows
    - ```
      cl -Ic:\java\include -Ic:\java\include\win32
                      -LD HelloWorldImp.c -Fehello.dll
      ```
- spustit program
  - java HelloWorld

Mapování typů

Java Type	Native Type	Size in bits
boolean	jboolean	8, unsigned
byte	jbyte	8
char	jchar	16, unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	n/a

Mapování typů

- "neprimitivní" typy
  - jobject
  - jstring
  - jclass
  - jthrowable
  - jarray
  - jobjectArray
  - jbooleanArray
  - jintArray
  - .....Array

Přístup ke Stringům

- typ **jstring** nelze přímo použít
  - převést na char*

```
Java_.....(JNIEnv *env, jobject obj, jstring prompt) {
  char *str = (*env)->GetStringUTFChars(env, prompt, 0);
  printf("%s", str);
  (*env)->ReleaseStringUTFChars(env, prompt, str);
  ...
```

  - nový string

```
char buf[128];
...
jstring jstr = (*env)->NewStringUTF(env, buf);
```

Přístup k polím

```
Java_..._sumArray(JNIEnv *env, jobject obj,
                                        jintArray arr) {
  int i, sum = 0;
  jsize len = (*env)->GetArrayLength(env, arr);
  ...
  jint *body = (*env)->GetIntArrayElements(env,arr,0);
  for (i=0; i<len; i++) {
    sum += body[i];
  }
  ...
  (*env)->ReleaseIntArrayElements(env, arr, body, 0);
  return sum;
}
```

Přístup k metodám

```
JNIEXPORT void JNICALL
Java_..._nativeMethod(JNIEnv *env, jobject obj, jint i) {
  jclass cls = (*env)->GetObjectClass(env, obj);
  jmethodID mid = (*env)->GetMethodID(env, cls, "method", "(I)V");
  (*env)->CallVoidMethod(env, obj, mid, i);
}
```

- signatura metody
  - program **javap**
    - javap -s trida

Signatura	Java typ
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L*třída;*	třída
[*typ*	typ []
(*parametry*)*navratový-typ*	metoda

Přístup k metodám

- GetMethodID()
- GetStaticMethodID()

- CallVoidMethod()
- CallIntMethod()
- ....
- CallStaticVoidMethod()
- CallStaticIntMethod()
- ....

Přístup k atributům

- obdobně jako k metodám

- fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
- fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");

- si = (*env)->GetStaticIntField(env, cls, fid);
- jstr = (*env)->GetObjectField(env, obj, fid);

- SetObjectField
- SetStaticIntField
- ...

Ošetření výjimek

```
jmethodID mid = ....
...
(*env)->CallVoidMethod(env, obj, mid);
jthroable exc = (*env)->ExceptionOccurred(env);
if (exc) {
   (*env)->ExceptionDescribe(env);
   (*env)->ExceptionClear(env);
   ....
}
```

Podpora synchronizace

```
...
(*env)->MonitorEnter(env, obj);
     synchronizovaný blok
(*env)->MonitorExit(env, obj);
...
```

- wait() a notify() nejsou přímo podporovány
  - lze je ale zavolat jako každou jinou metodu

C++

- C
  - ```
    jstring jstr = (*env)->GetObjectArrayElement
                                      (env, arr, i);
    ```
- C++
  - ```
    jstring jstr = (jstring) env->GetObjectArrayElement
                                              (arr, i);
    ```

JNA
(Java Native Access)

Přehled

- není součást JDK
- https://github.com/java-native-access/jna

- automatické mapování mezi Javou a nativními knihovnami
  - podpora pro „všechny“ platformy

- není potřeba psát ručně žádný nativní kód
  - interně používá JNI

JNA „Hello world“

```
import com.sun.jna.Library;
import com.sun.jna.Native;


public interface JNAApiInterface extends Library {
  JNAApiInterface INSTANCE = (JNAApiInterface)
Native.load((Platform.isWindows() ? "msvcrt" : "c"),
JNAApiInterface.class);

  void printf(String format, Object... args);
}



public static void main(String args[]) {
  JNAApiInterface jnaLib = JNAApiInterface.INSTANCE;
  jnaLib.printf("Hello World");
}
```

Mapování typů

Native Type	Size	Java Type
char	8-bit integer	byte
short	16-bit integer	short
wchar_t	16/32-bit character	char
int	32-bit integer	int
int	boolean value	boolean
long	32/64-bit integer	NativeLong
long long	64-bit integer	long
float	32-bit FP	float
double	64-bit FP	double
char*	C string	String
void*	pointer	Pointer

Mapování typů

- pole (ukazatel) → pole
- struct → … extends Structure
- předávání parametrů referencí
  - char **bufp → PointerByReference bufp
  - int* lenp → IntByReference lenp
- …
- JNAerator
  - https://github.com/nativelibs4java/JNAerator
  - generování hlaviček Java metod

Verze prezentace AJ03.cz.2019.01
Tato prezentace podléhá licenci Creative Commons Uveďte autora-Neužívejte komerčně 4.0 Mezinárodní License.