

复旦大学计算机科学技术学院



编程方法与技术

C.1. 上节课复习

周扬帆

2021-2022第一学期

垃圾回收机制

□ 哪些内存需要回收

- 错误回收 → 程序出错
- 经典算法：引用计数法、可达性分析

□ 如何进行内存回收

- 标记清除算法、复制算法、标记整理算法
 - 标记清除：删了不挪
 - 标记整理：删了挪
 - 复制：活着的移动到新区域
- 分代回收算法
 - Eden, Survivor 0/1; Tenured Generation
- 垃圾回收器

关于能发布的程序

□ Java编译工具集

■ javac编译

■ jar打包

→ jar包是一种zip格式的文件

→ Manifest文件指定

■ 去哪找入口 **Main-Class:**

■ 去哪找其他jar依赖 **Class-Path:**

■ java运行

→ 可以指定虚拟机堆空间大小

■ Young generations

□ Eden/From Survivor/To Survivor

■ Old generation

→ 指定Permanent generation大小

关于能发布的程序

□ Java判断当前jvm版本等系统信息

```
Properties prop = public static Properties getProperties();  
prop.getProperties("os.version");
```

java.version	Java 运行时环境版本	java.io.tmpdir	默认的临时文件路径
java.vendor	Java 运行时环境供应商	java.compiler	要使用的 JIT 编译器的名称
java.vendor.url	Java 供应商的 URL	java.ext.dirs	一个或多个扩展目录的路径
java.home	Java 安装目录	os.name	操作系统的名称
java.vm.specification.version	Java 虚拟机规范版本	os.arch	操作系统的架构
java.vm.specification.vendor	Java 虚拟机规范供应商	os.version	操作系统的版本
java.vm.specification.name	Java 虚拟机规范名称	file.separator	文件分隔符 (在 UNIX 系统中是"/")
java.vm.version	Java 虚拟机实现版本	path.separator	路径分隔符 (在 UNIX 系统中是".")
java.vm.vendor	Java 虚拟机实现供应商	line.separator	行分隔符 (在 UNIX 系统中是"/n")
java.vm.name	Java 虚拟机实现名称	user.name	用户的账户名称
java.specification.version	Java 运行时环境规范版本	user.home	用户的主目录
java.specification.vendor	Java 运行时环境规范供应商	user.dir	用户的当前工作目录
java.specification.name	Java 运行时环境规范名称		
java.class.version	Java 类格式版本号		
java.class.path	Java 类路径		
java.library.path	加载库时搜索的路径列表		

关于能发布的程序

□ 使用Properties类读取配置

```
FileReader fr;  
try {  
    String fileName = "test.conf";  
    fr = new FileReader(fileName);  
    Properties props = new Properties();  
    props.load(fr);  
    if (props.containsKey("name")) {  
        System.out.println("name = " + props.getProperty("name"));  
    }  
    else {  
        ...  
    }  
} catch (IOException e) {  
    ...  
}
```

反射

- 当一个类被加载, JVM 便自动产生一个 Class 对象

```
public class Reflection {  
    public static void main(String args[]) {  
        Reflection r1 = new Reflection();  
        Reflection r2 = new Reflection();  
        Class clazz1 = r1.getClass();  
        Class clazz2 = r2.getClass();  
        Class clazz3 = Reflection.class;  
        System.out.println(clazz1 == clazz2);  
        System.out.println(clazz1 == clazz3);  
    }  
}
```

静态方法、静态块会不会被执行?

Reflection.class → 加载类信息

Reflection.doSomething() → 1. 加载类信息 2. 初始化

反射例子：调用多参数方法

`void printInfo(int i, String info);`

```
import java.lang.reflect.Method;
public static void main(String argc[]) {
    try {
        Class<?> clazz = Class.forName ("TestClass");
        TestInterface test = (TestInterface)clazz.newInstance();
        //Parameter list
        Class<?> parameterTypes[] = new Class[]
            {int.class, String.class };
        Method m = clazz.getMethod("printInfo", parameterTypes);
        m.invoke(test, new Object[]{1, "Hello"});
    } catch(Exception e) {
        System.out.println(e);
    }
}
```

理解用Class数据构建形式参数表
用Object数组构建实际参数表

反射例子：动态加载类

```
public static void main(String argc[]) {  
    try {  
        URL [] urls = new URL[]{new URL("file:test.jar")};  
        URLClassLoader loader = new URLClassLoader(urls);  
        Class<?> clazz = loader.loadClass("TestClass");  
        TestInterface test = (TestInterface)clazz.newInstance();  
        //Parameter list  
        Class<?> parameterTypes[] = new Class[] {String.class };  
        Method m = clazz.getMethod("printInfo", parameterTypes);  
        m.invoke(test, "Hello");  
        loader.close();  
    } catch(Exception e) {  
        System.out.println(e);  
    }  
}
```

动态(运行时)实现类的加载

反射例子

□ 可用于实现插件

- 插件实现规定的interface
- 插件编译成jar包
- 运行时根据信息（如配置文件）获得插件类名
- 创建对象
- cast成规定的interface，并调用interface规定的接口

□ 插件的作用

- 方便（第三方）扩展程序
- 升级
- 针对需求（数据格式、用户需求等）改变程序行为
- 选择性加载（避免主程序过慢）

instanceof关键字

- ❑ 布尔计算(boolean)
- ❑ 用法object instanceof Class
- ❑ a instanceof B
 - 判断a是不是B这个类或者接口的实例
 - 返回true
 - 如果B是a对应的类 a = new B();
 - 或者, 是a对应的类的父类
 - 或者, 是a实现的接口

instanceof关键字

- 判断某个定义为父类的对象是不是事实上是一个子类的对象

```
class Dog extends Animal ...
```

```
Animal animal = new Dog();  
callSomething(animal); ...
```

```
...callsomething(Animal animal) {  
    if (animal instanceof Dog) {  
        ((Dog)animal).bark(); //安全调用Dog的方法  
    }  
}
```

判断是不是一个类自己的对象
`obj.getClass == 类名.class`

复旦大学计算机科学技术学院



编程方法与技术

C.2. HelloWorld Revisited

周扬帆

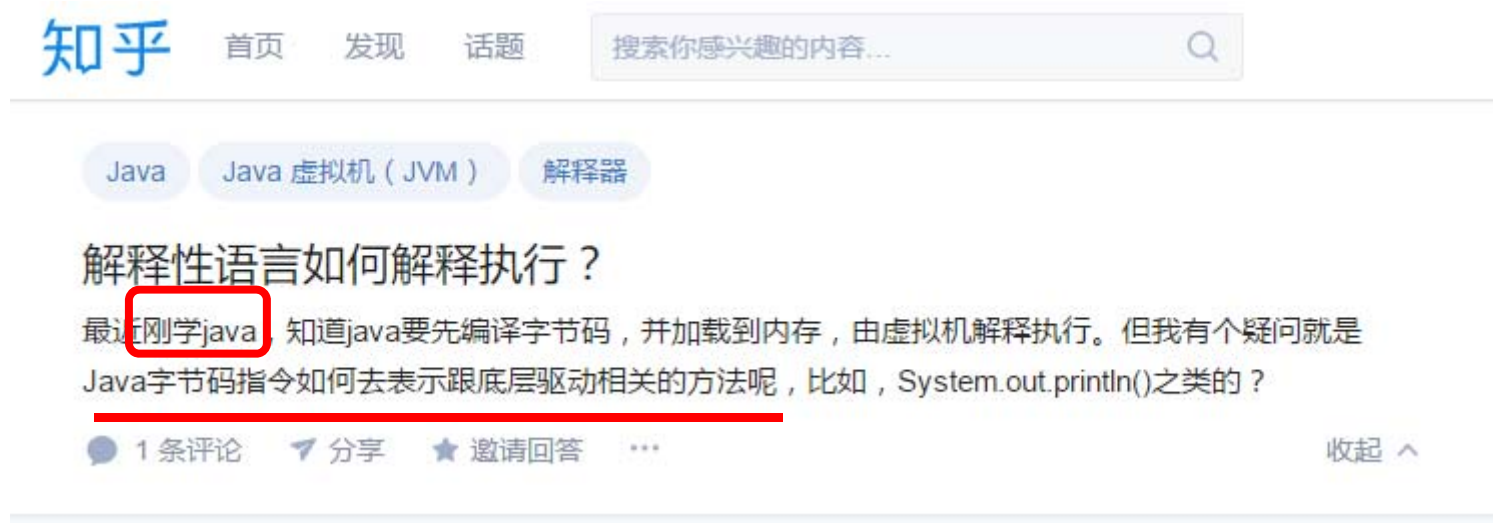
2021-2022第一学期

HelloWorld.java

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

- ❑ **javac将java变成bytecodes**
- ❑ **Jvm编译/解释执行bytecodes**
 - **往屏幕输出东西，怎么做？**
 - **不单单是CPU指令的翻译，还涉及CPU/MEM以外的东西**

HelloWorld.java



javap

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

□ 好奇一下class文件里的bytecodes

■ javap -c HelloWorld.class

→ javap: 自带的反汇编工具

```
1 Compiled from "HelloWorld.java"  
2 public class HelloWorld {  
3     public HelloWorld();  
4         Code:  
5         0: aload_0  
6         1: invokespecial #1          // Method java/lang/Object."<init>":()V  
7         4: return  
8  
9     public static void main(java.lang.String[]);  
10        Code:  
11        0: getstatic      #2          // Field java/lang/System.out:Ljava/io/PrintStream;  
12        3: ldc           #3          // String Hello, World!  
13        5: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V  
14        8: return  
15 }
```

就是简单做了方法调用

System类

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

□ 继续分析System.out是什么

■ java.lang.System

```
setOut0(new PrintStream(new BufferedOutputStream(fdOut, 128), true));
```

 new FileOutputStream(FileDescriptor.out);

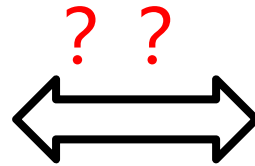
 new FileDescriptor(1);

没有太多magic, 就是往系统
默认file handler写东西

0: 标准输入
1: 标准输出
2: 标准错误输出

访问系统资源/功能

JVM: 一个跑在操作系统上的程序(jdk/bin/java)

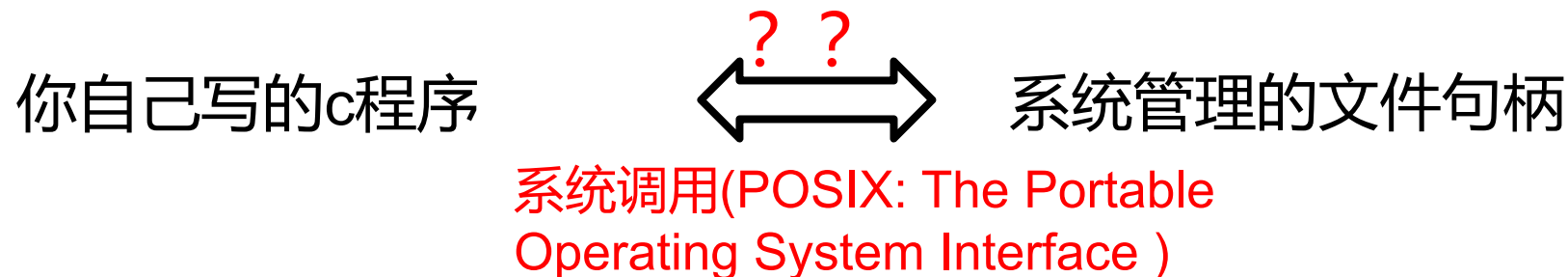


系统管理的文件句柄

没有太多magic, 就是往系统默认file handler写东西

0: 标准输入
1: 标准输出
2: 标准错误输出

访问系统资源/功能



函数原型：

```
#include <unistd.h>
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

函数参数：

-fd：要写入的文件的文件描述符

-buf：指向内存块的指针，从这个内存块中读取数据写入 到文件中

-count：要写入文件的字节个数

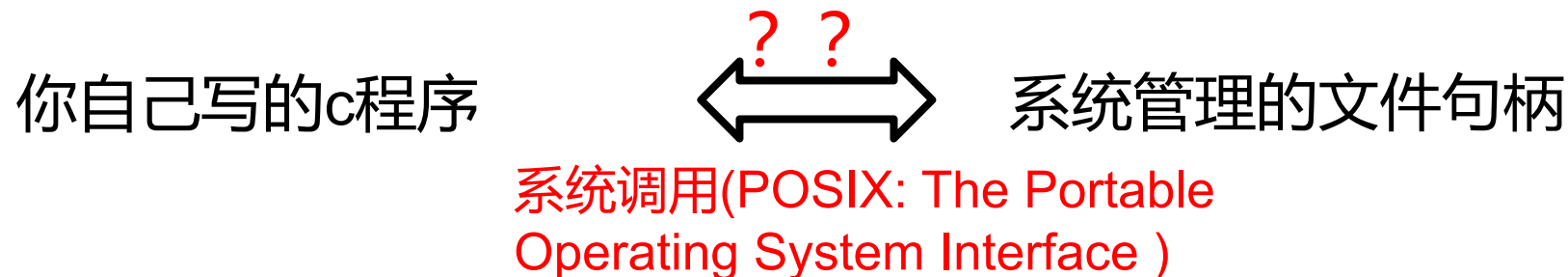
返回值

如果出现错误，返回-1

```
#include <unistd.h>
void main()
{
    write(1, "Hello, World!\n", 14);
}
```

```
svn@Jessica:~$ gcc testwrite.c -o testwrite
svn@Jessica:~$ ./testwrite
Hello, World!
svn@Jessica:~$
```

访问系统资源/功能



函数原型：

```
#include <unistd.h>
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

函数参数：

-fd：要写入的文件的文件描述符

-buf：指向内存块的指针，从这个内存块中读取数据写入 到文件中

-count：要写入文件的字节个数

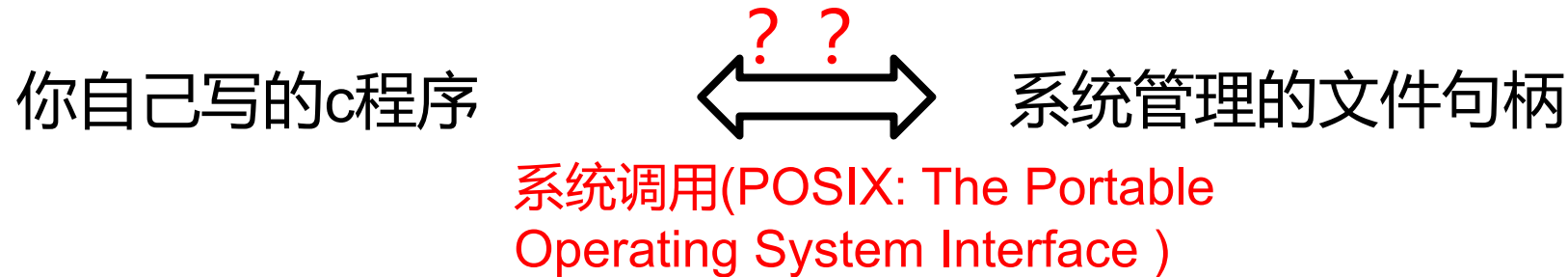
返回值

如果出现错误，返回-1

write实现在哪里？

```
svn@Jessica:~$ ldd testwrite
linux-vdso.so.1 => (0x00007ffc61d8c000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4c6a995000)
/lib64/ld-linux-x86-64.so.2 (0x000055814bed3000)
svn@Jessica:~$
```

访问系统资源/功能



□ 系统调用的实现

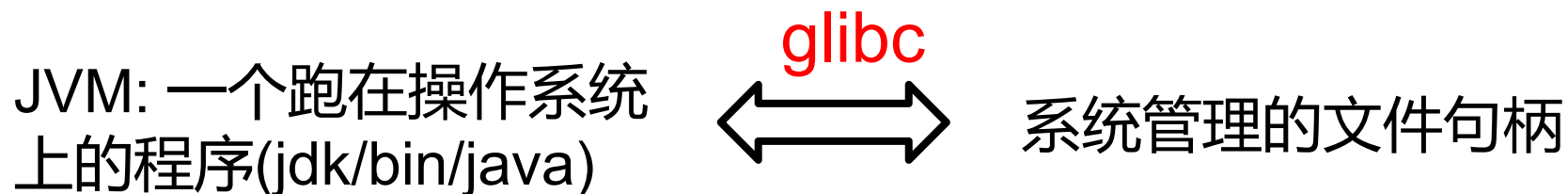
- Linux: glibc GNU C Library
- Android: Bionic – Google C Standard Library

□ 程序一般都通过C标准库函数，进行系统调用，访问系统资源/功能

- jvm也不例外

□ 这就是为什么说c是**native**(原生/本地)的

访问系统资源/功能



```
svn@Jessica:~$ ldd /usr/bin/java
linux-vdso.so.1 => (0x00007ffdeefdb000)
libjli.so => /usr/lib/jvm/java-8-openjdk-amd64/lib/amd64/jli/libjli.so (0x00007f3d6ef61000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3d6eb98000)
libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f3d6e97d000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f3d6e779000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f3d6e55b000)
/lib64/ld-linux-x86-64.so.2 (0x000055def392e000)
svn@Jessica:~$
```

没有太多magic, 就是往系统
默认file handler写东西

- 0: 标准输入
- 1: 标准输出
- 2: 标准错误输出

Output的过程

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

□ javac将java变成bytecodes

□ Jvm编译/解释执行bytecodes

- 和系统打交道的事, jvm有某种机制去调用C的库
- java.lang.System

```
setOut0(new PrintStream(new BufferedOutputStream(fdOut, 128), true));
```

```
public void write(byte b[]) throws IOException {  
    writeBytes(b, 0, b.length, append);  
}
```

```
new FileOutputStream(FileDescriptor.out);
```

```
new FileDescriptor(1);
```

```
private native void writeBytes(byte b[], int off, int len, boolean append)  
    throws IOException;
```

Output的过程

C代码

```
JNIEXPORT void JNICALL
Java_java_io_FileOutputStream_writeBytes(JNIEnv *env,
    jobject this, jbyteArray bytes, jint off, jint len) {
    writeBytes(env, this, bytes, off, len, fos_fd);
}
```

glibc

⇒ write()

setOut0(new PrintStream(new BufferedOutputStream(fdOut, 128), true));

```
public void write(byte b[]) throws IOException {
    writeBytes(b, 0, b.length, append);
}
```

new FileOutputStream(FileDescriptor.out);

new FileDescriptor(1);

```
private native void writeBytes(byte b[], int off, int len, boolean append)
    throws IOException;
```

复旦大学计算机科学技术学院



编程方法与技术

C.3. JNI入门

周扬帆

2021-2022第一学期

JNI: Java Native Interface

□ 设计目的

■ 进行系统调用(glibc.so), 访问系统资源/功能

- 输入、输出
- 文件系统
- 系统环境
- 网络 ...

■ 调用c实现的库, 目的?

- 性能
- Legacy代码
- 已写好的代码
- 干Java干不了的事
- 让人看不懂 😊

JNI: Java Native Interface

□ 很多语言都有类似的接口

■ node.js: node-gyp, 编译成module

```
var addon = require('./build/Release/hello');  
  
console.log(addon.hello());
```

■ python: ctypes, 编译成so

```
import ctypes  
so = CDLL("./libhello.so")  
so.test()
```

JNI HelloWorld

```
1 package com.cudroid.jni;
2 public class HelloJNI {
3     public native void sayHelloWorld();
4     public static void main(String[] args){
5         System.loadLibrary("libHelloWorld");
6         HelloJNI hello = new HelloJNI();
7         hello.sayHelloWorld();
8     }
9 }
```

- ❑ 方法声明加上**native**关键字
 - 不需要在java写上实现代码
- ❑ 用System.loadLibrary加载实现了这个方法/函数的动态链接库
- ❑ 像调用普通java方法一样调用native方法

JNI HelloWorld

```
package com.cudroid.jni;  
public class HelloJNI {  
    public native void sayHelloWorld();  
    public static void main(String[] args){  
        System.loadLibrary("libHelloWorld");  
        HelloJNI hello = new HelloJNI();  
        hello.sayHelloWorld();  
    }  
}
```

□ 生成头文件

□ javah com.cudroid.jni.HelloJNI

- 去com/cudroid/jni目录下找HelloJNI.java
- 生成com_cudroid_jni_HelloJNI.h

JNI HelloWorld

□ 生成的com_cudroid_jni_HelloJNI.h

```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class com_cudroid_jni_HelloJNI */
4
5  #ifndef _Included_com_cudroid_jni_HelloJNI
6  #define _Included_com_cudroid_jni_HelloJNI
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Class:      com_cudroid_jni_HelloJNI
12  * Method:     sayHelloWorld
13  * Signature:  ()V
14  */
15 JNIEXPORT void JNICALL Java_com_cudroid_jni_HelloJNI_sayHelloWorld
16     (JNIEnv *, jobject);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif
22
```

函数
signature

JNI HelloWorld

□ 生成的com_cudroid_jni_HelloJNI.h

```
package com.cudroid.jni;  
public class HelloJNI {  
    public native void sayHelloWorld();  
    public static void main(String[] args){  
        System.loadLibrary("libHelloWorld");  
        HelloJNI hello = new HelloJNI();  
        hello.sayHelloWorld();  
    }  
}
```

```
13  * Signature: (J)V  
14  */  
15  JNIEXPORT void JNICALL Java_com_cudroid_jni_HelloJNI_sayHelloWorld  
16      (JNIEnv *, jobject);  
17  
18  #ifdef __cplusplus  
19  }  
20  #endif  
21  #endif  
22
```

包名 类名 方法名

JNI HelloWorld

```
#include "jni.h"
#include "com_cudroid_jni_HelloJNI.h"
JNIEXPORT void JNICALL
Java_com_cudroid_jni_HelloJNI_sayHelloWorld (JNIEnv* env, jobject obj)
{
    printf("Hello, World!\n");
    return;
}
```

□ 实现native的方法

- include 生成的头文件和jni.h
- 实现头文件声明的方法

□ 编译成so

- gcc -shared HelloJNI.c -o libHelloWorld.so

JNI HelloWorld

```
1 package com.cudroid.jni;
2 public class HelloJNI {
3     public native void sayHelloWorld();
4     public static void main(String[] args){
5         System.loadLibrary("libHelloWorld");
6         HelloJNI hello = new HelloJNI();
7         hello.sayHelloWorld();
8     }
9 }
```

- ❑ 用System.loadLibrary加载libHelloWorld.so
- ❑ 像调用普通java方法一样调用native方法
 - 输出: Hello, World!

JNIEXPORT void JNICALL

Java_com_cudroid_jni_HelloJNI_sayHelloWorld (JNIEnv*env, jobject obj)

{

printf("Hello, World!\n");

return;

}

JNI局限性

❑ 兼容性？

- 继承C的“平台相关”局限
- Linux编译的so不能在Windows下被java loadLibrary

❑ 鲁棒性robustness?

- 继承C的各种问题: 空指针/野指针, 内存泄漏, 等等

❑ 替代方案

- 跨进程: Java和C实现为不同程序, 进程间通信
→ C/S

❑ 很多时候无法用替代方案

- 效率考虑
- Java需要调用系统功能/访问系统数据 (e.g., System.out.println)
- ...

思考

- 理解Java等语言为什么需要JNI等和C打交道的机制

复旦大学计算机科学技术学院



编程方法与技术

C.4. JNI入门++

周扬帆

2021-2022第一学期

Java对动态链接库的加载

```
1 package com.cudroid.jni;
2 public class HelloJNI {
3     public native void sayHelloWorld();
4     public static void main(String[] args){
5         System.loadLibrary("libHelloWorld");
6         HelloJNI hello = new HelloJNI();
7         hello.sayHelloWorld();
8     }
9 }
```

□ System.loadLibrary可以实现的位置

- 静态块
- 使用之前

利弊?

JNI设计考虑

- ❑ Java的基本数据类型/c的基本数据类型
- ❑ Java的String能不能传给c
- ❑ Java的数组能不能传给c
- ❑ Java的对象引用能不能传给c
- ❑ c怎么抛出异常给java

JNI设计考虑

- ❑ **Java的基本数据类型/c的基本数据类型**
- ❑ **Java的String能不能传给c**
- ❑ **Java的数组能不能传给c**
- ❑ **Java的对象引用能不能传给c**
- ❑ **c怎么抛出异常给java**

基本数据类型

□ Java的基本数据类型/C的基本数据类型

Java类型 直接映射的c类型 实际c的类型 说明

boolean	jboolean	unsigned char	无符号，8 位
byte	jbyte	signed char	有符号，8 位
char	jchar	unsigned short	无符号，16 位
short	jshort	short	有符号，16 位
int	jint	long	有符号，32 位
long	jlong	__int64	有符号，64 位
float	jfloat	float	32 位
double	jdouble	double	64 位
void	void	N/A	N/A

基本数据类型

□ Java的基本数据类型/C的基本数据类型

```
public class HelloJNI {  
    public native boolean testBSIL(short s, int i, long l);  
    public native byte testBC(char c);  
    public native float testFD(double d);  
}
```

char	jchar	unsigned short	无符号, 16 位
short	jshort	short	有符号, 16 位
int	jint	long	有符号, 32 位

```
JNIEXPORT jboolean JNICALL Java_HelloJNI_testBSIL  
(JNIEnv *, jobject, jshort, jint, jlong);
```

```
JNIEXPORT jbyte JNICALL Java_HelloJNI_testBC  
(JNIEnv *, jobject, jchar);
```

```
JNIEXPORT jfloat JNICALL Java_HelloJNI_testFD  
(JNIEnv *, jobject, jdouble);
```


JNI设计考虑

- ❑ Java的基本数据类型/c的基本数据类型
- ❑ Java的String能不能传给c
- ❑ Java的数组能不能传给c
- ❑ Java的对象引用能不能传给c
- ❑ c怎么抛出异常给java

String的相互传递

□ String: jstring

```
public native String echo(String str);
```

```
JNIEXPORT jstring HelloJNI_echo  
(JNIEnv * env, jobject obj, jstring str)  
{
```

```
    int i;  
    char returnbuf[128] = "Hello, World!";
```

```
    const jbyte *buffer;
```

```
    buffer = (*env)->GetStringUTFChars(env, str, NULL);
```

把这个参数来访问运行环境: JVM

把String转成数组

```
    ...
```

```
    (*env)->ReleaseStringUTFChars(env, str, buffer);
```

release内存

```
    ...
```

```
    return (*env)->NewStringUTF(env, returnbuf);
```

```
}
```

构建String对象返回

为什么要通过JVM处理?

JNI设计考虑

- Java的基本数据类型/c的基本数据类型
- Java的String能不能传给c
- **Java的数组能不能传给c**
- Java的对象引用能不能传给c
- c怎么抛出异常给java

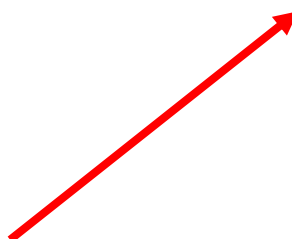
数组的相互传递

□ 数组: jarray / jintArray

`public native` int sum(int []arr);

```
JNIEXPORT jint HelloJNI_sum
(JNIEnv * env, jobject obj, jintArray arr)
{
    jint buf[10];
    jint i, sum = 0;
    (*env)->GetIntArrayRegion(env, arr, 0, 10, buf);
    for (i = 0, i < 10, i++) {
        sum += buf[i];
    }
    return sum;
}
```

jarray不是c的数组对象,
只能通过jvm来转换成数组



`jsize len = (*env)->GetArrayLength(env, arr);`

数组的相互传递

□ 数组: jarray / jintArray

```
public native int sum(int [ ]arr);

JNIEXPORT jint HelloJNI_sum
(JNIEnv *env, jobject obj, jintArray arr)
{
    jint *carr;
    jint i, sum = 0;
    carr = (*env)->GetIntArrayElements(env, arr, NULL); 数组转换
    if (carr == NULL) {
        return 0; /* exception occurred */
    }
    for (i=0; i<10; i++) {
        sum += carr[i];
    }
    (*env)->ReleaseIntArrayElements(env, arr, carr, 0); 释放内存
    return sum;
}

jsize len = (*env)->GetArrayLength(env, arr);
```

数组的相互传递

□ 数组: jarray / jintArray

```
JNIEXPORT jintArray HelloJNI_sum
(JNIEnv *env, jobject obj)
{
    jintArray javaArray;
    jint nativeArr[3] = {21,22,23};
    javaArray = (*env)->NewIntArray(env,3);
    (*env)->SetIntArrayRegion(env,javaArray,0,3,nativeArr);
    return javaArray;
}
```

创建数组

拷贝数组

JNI设计考虑

- ❑ Java的基本数据类型/c的基本数据类型
- ❑ Java的String能不能传给c
- ❑ Java的数组能不能传给c
- ❑ **Java的对象引用能不能传给c**
- ❑ c怎么抛出异常给java

访问对象

□ java对象: jobject

```
private String name = "I am at Java" ;
```

```
JNIEXPORT jint HelloJNI_test  
(JNIEnv *env, jobject obj)  
{
```

```
    jfieldID nameFieldId ;
```

获得Java层该对象实例的类引用

```
    jclass cls = env->GetObjectClass(obj);
```

获得属性句柄

```
    nameFieldId = env->GetFieldID(cls , "name" , "Ljava/lang/String;");
```

```
    if(nameFieldId == NULL)
```

```
    { ... }
```

获得值

```
    jstring javaNameStr = (jstring)env->GetObjectField(obj ,nameFieldId);
```

```
    ...
```

```
}
```


访问对象

□ java对象: jobject

```
public void callback(String fromNative){ ...}
```

```
JNIEXPORT jint HelloJNI_test (JNIEnv *env, jobject obj)
{
    jclass cls = env->GetObjectClass(obj);           获得Java层该对象实例的类引用
    jmethodID callbackID = env->GetMethodID(cls, "callback", "(Ljava/lang/String;)V");  获得方法句柄
    if(callbackID == NULL)
    {
        ...
    }
    jstring native_desc = env->NewStringUTF(" I am Native");
    env->CallVoidMethod(obj, callbackID, native_desc);  调用
}
```

JNI设计考虑

- ❑ Java的基本数据类型/c的基本数据类型
- ❑ Java的String能不能传给c
- ❑ Java的数组能不能传给c
- ❑ Java的对象引用能不能传给c
- ❑ c怎么抛出异常给java

异常抛出

□ Java异常处理

```
public native void throwit() throws IllegalArgumentException;;
```

```
JNIEXPORT void HelloJNI_throwit (JNIEnv *env, jobject obj)
```

```
{  
    jclass newExcCls;  
    newExcCls = (*env)->FindClass(env, "java/lang/IllegalArgumentException");  
    if(newExcCls == NULL)  
    {  
        ...  
    }  
    (*env)->ThrowNew(env, newExcCls, "thrown from C code");  
}
```

获得Java层该异常类

抛出异常

思考

- ❑ 思考JNI的优缺点
- ❑ 思考什么情景适合使用JNI
- ❑ 思考还有哪些Java与其他语言通讯的方法

复旦大学计算机科学技术学院



编程方法与技术

C.5.作业：JNI/反射/普通方法调用比较

周扬帆

2021-2022第一学期

练习

□ 设计三种方法调用

- JNI
- 反射
- 普通

□ 比较性能差异

□ ?

- 自行设计，比较出来不同场景（如函数体运算复杂但调用少，函数简单但需要多次调用等，访问大对象，实例化/非实例化调用等）下的性能差距
- 交：代码和report(简短，清晰说明这么比的motivation及结论)