# C语言基础

## 基础介绍

#include<stdio.h>//include相当于java的import；stdio：可以理解为standard input output标准输入输出；.h头文件的后缀：包含一些函数

#include<stdlib.h>//导入标准C语言lib函数库

main()//相当于java的public static void main(String args[]){}函数入口

{

printf("Hello world !\n");//输出函数。C语言叫函数，JAVA叫方法。相当于java的system.out.print("Hello world !\n") C语言的\n回车换行。JAVA语言\r回车\n换行

system("calc"); //调用系统计算器

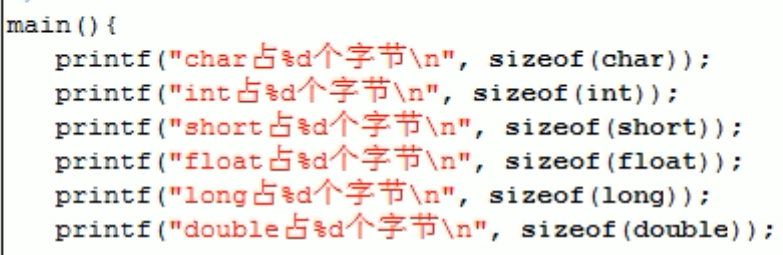
system("java World");//调用java字节码

system("pause");//让docs命令行执行pause命令，作用是让控制台停留

}

## 数据类型

Java中，打印的内容用+来链接。C中使用的是占位符

后面sizeof的结果会返回给前面的占位符！

char， int，float，double，long，short，signed，unsigned，void

计算类型的长度：sizeof("类型")返回int类型的长度

占位符:%d

printf("内容");

### java基本数据类型和C语言的区别

1.java中char类型长度为2个字节，C语言为1个字节

2.java中long类型长度为8个字节，C语言为4个字节

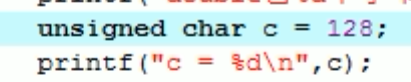
C99标准规定：long类型的规定，不小于整形，所以只要大于4个字节即可，有些系统long是8个字节

3.C语言中没有byte

4.C语言中没有boolean类型，0表示false，非0表示true

signed：有符号 -128~127 = -2^7 ~ 2^7-1

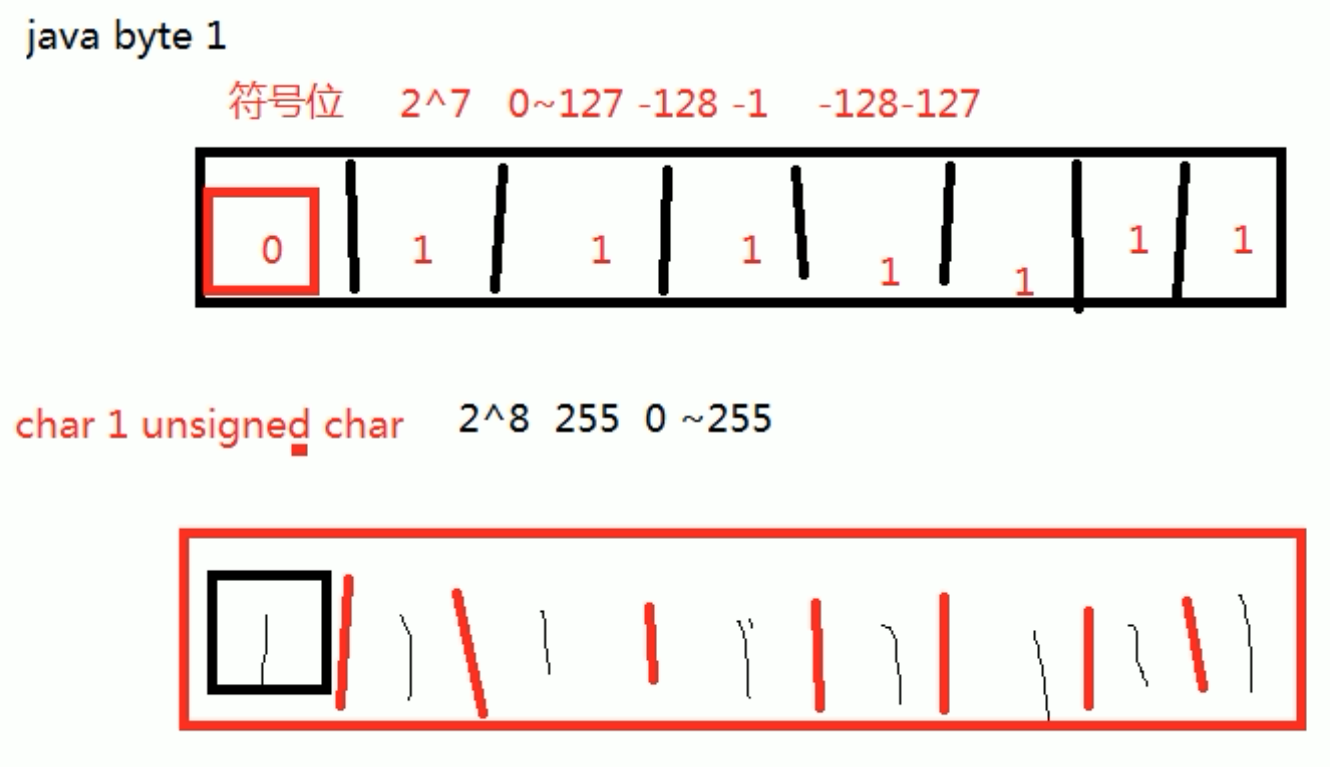
unsigned:无符号 0~255 = 0 ~ 2^8-1



signed只能用来修饰整形变量，char int short long

void 无类型，任意类型

#### signed和unsigned



### 判断

if(1){

printf("true\n");//非0表示true

}else{

printf("false\n");//0走这里

}

## 输出函数

printf("内容");

%d - int

%ld - long int

%c - char

%f - float

%u - 无符号数

%hd - 短整形

%lf - double

%x - 十六进制输出

%o - 八进制输出

%s - 字符串

在C语言中，默认保留小数点后六位

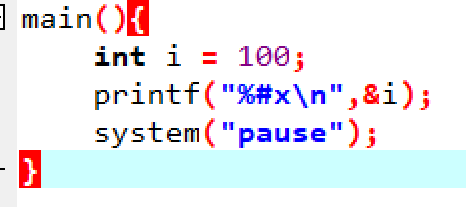
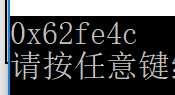
要想保留对应的位数，就需要在%后加".数字"；如：%.4f保留小数点后4位

十进制：12345678

二进制：101111000110000101001110

110000101001110

用短整形输出后会精度丢失

%#x 表示

相同的类型，要用相应的占位符去输出，否则精度丢失

//C语言的数组的括号写在变量的右边

char cArray[] = {'A','B'};

//在x前面加#号。十六进制结果默认加上0x

printf("cArray内存地址==%#x\n",&cArray);

printf("cArray内容==%s\n",cArray);

//指针方式输出字符串

char\* text = "I love you!";

## 输入函数

### 输入函数介绍

scanf("占位符"，内存地址);

&X取X的内存地址

%d - int

%ld - long int

%c - char

%f - float

%u - 无符号数

%hd - 短整形

%lf - double

%x - 十六进制输出

%o - 八进制输出

%s - 字符串

System.in()

### 输入函数一：

//在C语言中没有String类型，但是可以用char数组来表示

char cArray[] = {'H','E','L','L','O'};

int i;//C语言变量不能在for循环里面声明

for(i=0;i<5;i++){

printf("cArray[%d]==%c\n",i,cArray[i]);

}

printf("cArray的字符串是：%s \n",cArray);

### 输入函数二：

//在C语言中没有String类型，但是可以用char数组来表示

char cArray[5];

printf("请输入hello \n");

scanf("%s",&cArray);

int i;//C语言变量不能在for循环里面声明

for(i=0;i<5;i++){

printf("cArray[%d]==%c\n",i,cArray[i]);

}

printf("cArray的字符串是：%s \n",cArray);

### 输入函数三：

main(){

char cArray[]={'a','b','c','d','e','\0'}; //\0代表结束

//数组是一块连续的内存空间

printf("cArray的字符串是：%s \n",cArray);

char\* text = "I LOVE YOU!";

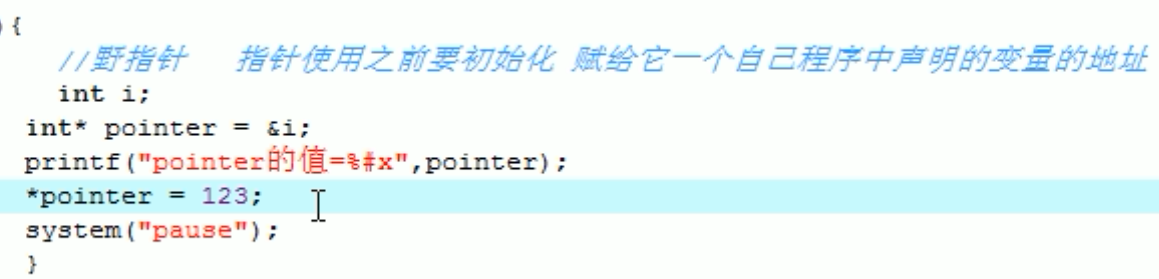
printf("%s",text);

system("pause");

}

## 指针

指针必须要先在自己的程序里面声明单位。然后再把单位的地址给指针。因为直接声明指针后，系统会给指针随时一个地址值，也叫野指针！。这个地址值有可能正保存着其他应用的重要数据，比如保存着用户名密码。所以不能直接对野指针进行赋值！



指针就是内存地址

内存地址就是指针

int\* p是int类型的指针

char\*s是char类型的指针

不管什么类型的指针，都是内存地址！！

\*p是对应的内存地址的值

p是对应的内存地址

int i = 100;

int\*p = &i; //int\* p; p = &i;

int j = 8888;

\*p = j;相当于\*p=8888;一样的效果

p和j的内存地址不一样

p的内存地址和i是一样的。所以\*p改变了，i的值也变了

int\*p ;

\*p = 22;//错误！指针必须关联内存地址。不能直接赋值！！

p = &i;//正确

printf("p==%d\n",\*p);

//定义一个int类型的变量i，并且赋值为10

int i = 10;

//定义一个int类型的一级指针变量p

int\* p;

//把i的内存地址赋给p变量

p = &i;

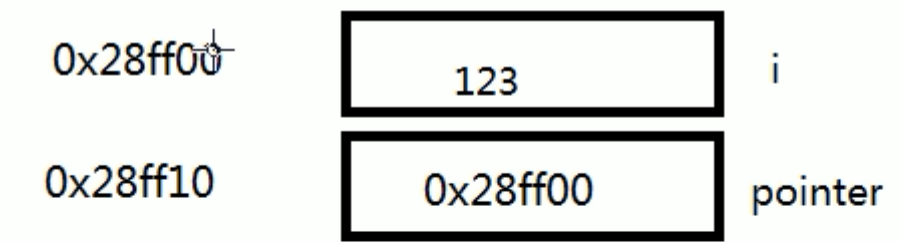
//指针取值\*p：把p变量对应的地址的值取出来

printf("\*p==%d\n",\*p);

\*p = 100;//改变\*p的值，实际上是改变\*p保持内存地址所对应的值！！！

printf("\*p==%d\n",\*p); //p对应的内存地址是&i，\*p是取对应内存的值，就是&i的值

printf("\*p==%d\n",i); // i的值是10。所以改变\*p=100，i也变成100！！



### 互换数字

JAVA中无法做到在方法中直接操作内存地址从而迅速改变值

C指针速度快！

void sitch(int a,int b){//传值无法改变值

int temp = a;

a = b;

b = temp;

printf("sitch 中a地址==%#x,sitch 中b地址==%#x\n",&a,&b);//地址都不一样

}

void sitch2(int\* a,int\* b){//传指针，即内存地址可以改变值！！

int temp = \*a;

\*a = \*b;

\*b = temp;

printf("a地址==%#x,b地址==%#x\n",a,b);//地址都不一样

}

### 指针的长度

/\*\*

指针的长度

64位系统指针长度是8

32位系统指针长度是4

\*/

## 多级指针

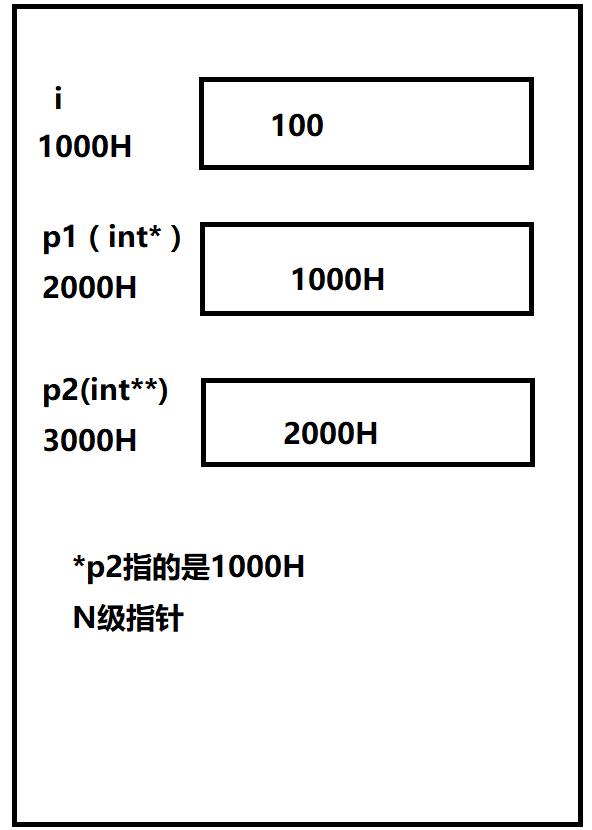
如果\*p=一个具体值，那么表示\*p是一个一级指针

如果\*p=一个内容地址，那么表示\*p是一个多级指针

有n个\*。那么值就等于n个\*p

之前的（n-1）\*p就是内存地址

始终记住：有N级指针，有多少个\*（n<=N），就是取多少个\*(n<=N-1)所对应内存地址的值



main(){

//

int i = 100;

int j = 200;

//一级指针。把i的内存地址给p

int\* p =&i;// int \*p; p = &i;的缩写

//二级指针。把p的内存地址给p1

int\*\* p1 = &p;

//多级指针。多个\*\*\*\*\*\*

printf("i==%d,p==%d,p1==%d\n",i,\*p,\*\*p1);

//\*p1指向的是p的内存地址。所以\*p1==p

printf("i==%#x,p==%#x,p1==%#x\n",&i,p,\*p1);

## 数组

1.数组的取值

2.数组的取地址

3.数组是一块连续的内存空间

4.数组的首地址，和数组的地址一样

main(){

char cArray[] = {'H','E','L','L','O'};

int iArray[] = {1,2,3,4,5};

//取数组的值

printf("cArray[0]==%c\n",cArray[0]);

printf("cArray[1]==%c\n",cArray[1]);

printf("iArray[0]==%d\n",iArray[0]);

printf("iArray[1]==%d\n",iArray[1]);

//取内存地址值

printf("cArray[0]内存地址==%#x\n",&cArray[0]);

printf("cArray[1]内存地址==%#x\n",&cArray[1]);

printf("cArray[2]内存地址==%#x\n",&cArray[2]);

printf("cArray[3]内存地址==%#x\n",&cArray[3]);

printf("cArray+0内存地址==%#x\n",cArray+0);

printf("cArray+1内存地址==%#x\n",cArray+1);

printf("cArray+2内存地址==%#x\n",cArray+2);

printf("cArray+3内存地址==%#x\n",cArray+3);

printf("iArray内存地址==%#x\n",&iArray);

printf("iArray[0]内存地址==%#x\n",&iArray[0]);

printf("iArray[1]内存地址==%#x\n",&iArray[1]);

printf("iArray[2]内存地址==%#x\n",&iArray[2]);

printf("iArray[3]内存地址==%#x\n",&iArray[3]);

//指针取值

printf("iArray==%d\n",\*iArray);

printf("iArray[0]==%d\n",\*iArray+0); // 错误，仅仅是取数组第一个值+1

printf("iArray[1]==%d\n",\*iArray+1);

printf("iArray[2]==%d\n",\*iArray+2);

printf("iArray[0]==%d\n",\*(iArray+0));//移动指针来取数组值

printf("iArray[1]==%d\n",\*(iArray+1));

printf("iArray[2]==%d\n",\*(iArray+2));

system("pause");

}

## 内存分配

### 静态内存

静态内存程序编译执行后系统自动分配，由系统自动释放，静态内存是栈分配的

### 动态内存

动态内存是开发者手动分配的，是堆分配

通过malloc或者new，在堆内存中创建，需要手动free或者delete释放内存

/\*\*

动态内存分配

\*/

void func(int\*\* address){

int i = 100;

int\* temp;

//malloc(int)-内存地址

temp = malloc(sizeof(int));

//把i对应的值，赋值给temp地址对应的值

\*temp = i;

//把一级指针的地址，修改成temp

\*address = temp;

free(temp); //手动释放内存

}

main(){

//定义int类型一级指针变量iPoint

int\* iPoint;

func(&iPoint);

printf("\*iPoint==%d\n",\*iPoint);

printf("\*iPoint==%d\n",\*iPoint);

printf("\*iPoint==%d\n",\*iPoint);

system("pause");

}

## Typedef别名

#include<stdio.h>

#include<stdlib.h>

/\*\*

别名

声明自定义数据类型，配合各种原有数据类型来达到简化编程的目的的类型定义关键字

在计算机编程语言中，用来为复杂的声明定义简答的别名

在类中把名字很长的方法用简写或者代替方式

\*/

typedef int i;//用i来代替int

typedef long l;

main(){

i m = 10;//用i来代替int i m == int m

l n = 123123123;

printf("%d\n",m);

printf("%d\n",n);

system("pause");

}

## 函数指针

感觉有点像给函数起个别名

#include<stdio.h>

#include<stdlib.h>

/\*\*

函数指针

指针指向函数

\*/

int add(int x,int y){

return x+y;

}

main(){

//定义函数指针

int (\*yangke)(int x,int y);

//函数指针赋值

yangke = add;

//使用函数指针

int result = yangke(30,18);

printf("result==%d\n",result);

system("pause");

}

## Unition联合体

所有字段共用一块内存空间

应用场景：各个类型之间方便转换

union Mix{

long i;//4个字节

int k;//4个字节

char s;//1个字节

};

main(){

printf("mix:%d\n",sizeof(union Mix));//所有字段共同一块内存空间，大小为4个字节

union Mix m;

m.i = 100;

m.k = 123;

printf("m.i = %d\n",m.i); //结果123

printf("m.k = %d\n",m.k); //结果123

system("pause");

}

## 枚举

.h>

/\*\*

枚举

值是递增效果，逐渐+1

默认是从0开始

\*/

enum WeekDay{

Monday,Tuesday

};

main(){

enum WeekDay day = Tuesday;

printf("%d\n",day);//结果是1

system("pause");

}

## 结构体指针

机构体不能有函数，只能有变量

### 结构体

#include<stdio.h>

#include<stdlib.h>

/\*\*

结构体

\*/

struct student{//结构体，类似java对象

int age;//4个字节

float score;//4个字节

char sex;//1个字节

};//结构体长度是最大的字节的倍数。4个字节\*3种类型=长度12

main(){

//使用结构体

struct student stu = {18,98.9,'W'};

//结构体取值

printf("stu.age==%d\n",stu.age);

printf("stu.score==%.1f\n",stu.score);

printf("stu.sex==%c\n",stu.sex);

//结构体赋值

stu.age = 20;

stu.score = 100;

stu.sex = 'M';

printf("stu.age==%d\n",stu.age);

printf("stu.score==%.1f\n",stu.score);

printf("stu.sex==%c\n",stu.sex);

//结构体的长度

printf("struct student长度==%d\n",sizeof(struct student));

system("pause");

}

### 结构体指针

main(){

//使用结构体

struct student stu = {18,98.9,'W'};

//结构体指针

struct student\* point = &stu;

//二级结构体指针

struct student\*\* point2 = &point;

//结构体指针取值要加括号（\*point）= stu

//(\*point).sex 等价于 point->sex

printf("stu.age==%d\n",(\*point).age);

printf("stu.score==%.1f\n",(\*point).score);

// printf("stu.sex==%c\n",(\*point).sex);

printf("stu.sex==%c\n",point->sex);

//赋值运算

//point->age = 888;

(\*point).age = 9999;

printf("stu.age==%d\n",(\*point).age);

//二级结构体指针取值

// (\*point2)->age

//(\*\*point2).age

printf("struct student\*\* point2==%d\n",(\*\*point2).age);

system("pause");

}

# C++

## 继承，多态

C++支持多继承

#pragma once

#include <iostream>

class Animal{

public:

// 添加virtual虚函数就有实现多态

virtual void test(){

printf("Animal test()\n");

}

// 纯virtual虚函数，子类必须实现！相当于java中的abstract

virtual void test1() = 0;//赋值为0，表示该函数在当前类中，没有实现！

};

class Animal2{

public:

void test(){

printf("Animal test()\n");

}

};

// public只能继承public的方法

// private只能继承private方法

// 多继承

class Dog : public Animal,Animal2{

public :

void test(){

// Animal::test(); //调用父类方法，相当于java中的super

printf("Dog test()\n");

}

void test1(){

printf("Dog test() 纯虚函数\n");

}

};

int main(){

Animal\* a = new Dog();//多态

// Animal\*a = new Animal();

// Dog d;

// a = &d;

a->test();

return 0;

}

virtual 修饰的函数 是支持多态的函数！

1. 构造函数永远不要设为虚函数，因为多态的灵活性
2. 析构函数声明为虚函数，多态可以释放子类的对象

## 泛型

1. 泛型方法声明template <typename T>
2. 泛型类声明template <class T,class E>

#include <iostream>

// 类似java中的泛型

template <typename T>

// 通过泛型避免创建太多的重载方法

T Max(T i,T j){

return i>j?i:j;

}

// 类似java的方法泛型

template <typename T>

T add(T i,T j){

return i+j;

}

// 类的泛型 类似java的类泛型

template <class T,class E>

class Q{

public:

E test(T i,E j){

return i+j;

}

};

int main(){

std::cout << Max(10,20) << std::endl;

std::cout << add(10,20) << std::endl;

Q<int,float> q;//声明的时候，传入泛型

std::cout << q.test(30,1.34f) << std::endl;

return 0;

}

## 强转

C的强转和java的一样

Tyep a = （Type）b；



1. const修饰符==Java中的final const\_cast就是强转final类型
2. static\_cast 静态类型的强转
3. dynamic\_cast 子类和父类的转换
4. reinterpret\_cast
5. Tyep a = static\_cast<Type>(b)；

### const\_cast

//const\_cast类型强转

const char\* a;

char\*b = const\_cast<char\*>(a);

### static\_cast编译时

1. 基础数据类型的转换 如： float转int，int转unsigned int等
2. 指针与void之间转换 如： float\*转void\*，Bean\*转void\*，函数指针转void\*
3. 子类指针/引用 与 父类指针/引用 转换

#### 父类转子类

// 当Parent加了virtual关键字的时候

// 强转后，还是调用Parent的方法！

// 如果Parent没有virtual关键字

// 则强转后，调用Child的方法

Parent \*p = new Parent;

Child \*c = static\_cast<Child\*>(p);

c->test();

结果就是：正常的多态相反！

#### 子类转父类

// 当Parent加了virtual关键字的时候

// 强转后，还是调用Child的方法！

// 如果Parent没有virtual关键字

// 则强转后，调用Parent的方法

Child \*c2 = new Child();

Parent \*p2 = static\_cast<Parent\*>(c2);

p2->test();

结果就是：正常的多态！

class Parent{

public:

virtual void test(){//加了虚函数，强转后，还是调用Parent的方法！

cout << "p" <<endl;

}

};

class Child : public Parent{

public:

void test(){

cout << "c" << endl;

}

};

int main(){

//static\_cast类型强转

// 父类转子类

Parent \*p = new Parent;

Child \*c = static\_cast<Child\*>(p);

// 当Parent加了virtual关键字的时候

// 强转后，还是调用Parent的方法！

// 如果Parent没有virtual关键字

// 则强转后，调用Child的方法

c->test(); // 结果恰好和加了虚函数多态是反的！

// 虚函数 当Parent的test方法前面加上virtual关键字

// 开启多态效果

Parent \*p1 = new Child();

p1->test(); // 结果调用的是 Child里面的test

// std::cout << \*a;

// std::cout << \*b;

}

### dynamic\_cast运行时

主要是 将基类指针引用 安全的转为派生类（父转子！）

在运行期对可疑的转型操作进行安全检查，仅对多态有效



## 容器

序列式 与 关联式

这两种容器都在标准函数库里面stl

序列式容器 ： 元素排列顺序与元素本身无关，由添加顺序决定！

vecotr

list

dequeue

queue

stack

priority queue

关联式容器 ：通过key-value的形式来存储

set

map

## 命名空间

相当于Java中的import

#include <iostream>

using namespace std;

// 声明命名空间 func1

namespace func1 {

void func() {

cout << "func 1" << endl;

}

}

// 声明命名空间 func2

namespace func2 {

void func() {

cout << "func 2" << endl;

}

}

int main()

{

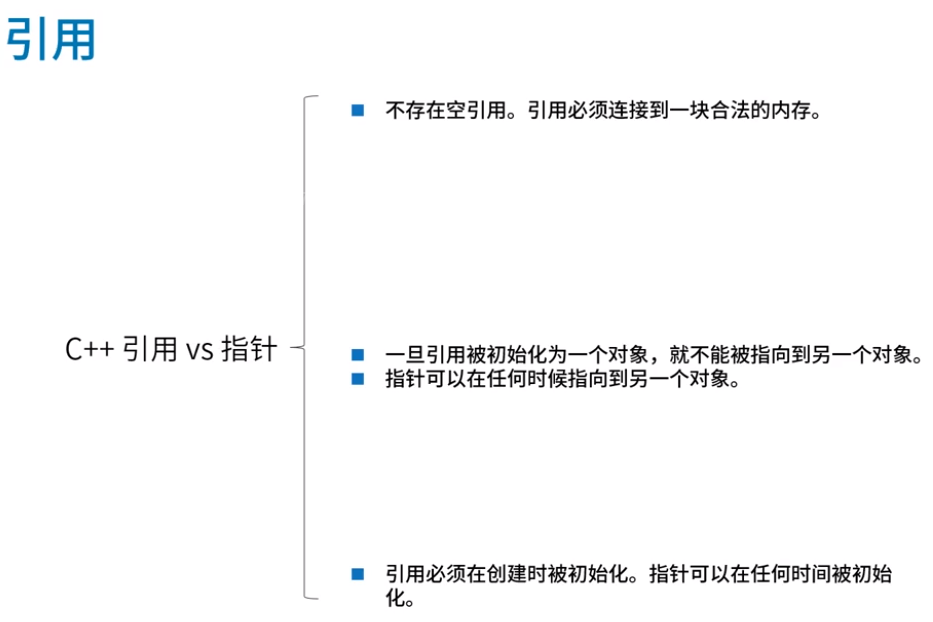
// 通过命名空间调用

func1::func();

func2::func();

}

## 引用



# JNI与NDK开发

JNI是开发协议

Ndk是开发工具包

## JNI是什么

Jni是java native interface java本地开发接口

Jni可以让java与C/C++代码互调

### 为什么要用JNI

1. native coder执行高效：大量的运算（极品飞车），万能解码（ffmpeg），Opengl（3D渲染）
2. 通过JNI技术，可以扩展android手机的功能-wifi热点
3. 代码复用：ffmpeg，onencv（人脸识别库），7-zip
4. 使用场景：考虑特殊情况（效率/操作硬件）

### JNI开发流程

## NDK是什么

Native Develop Kits（本地开发工具包）

Android提供的用来做JNI开发工具包

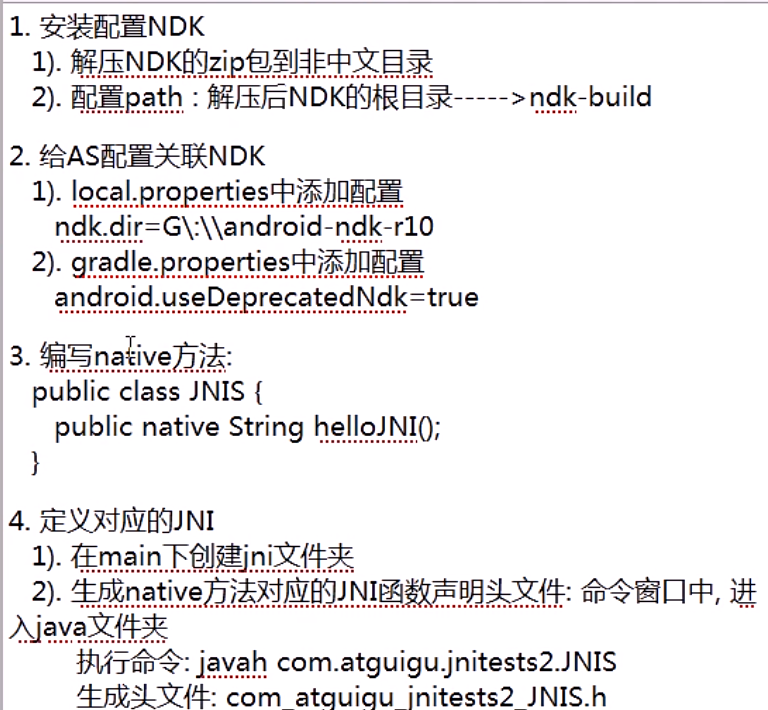
提高JNI开发效率：

生成代码

代码提示

多平台交叉编译

### 配置步骤



1. 环境变量path：C:\Android\android-ndk-r14b
2. AS的local.properties中添加配置(只要安装ndk，android studio自动添加)

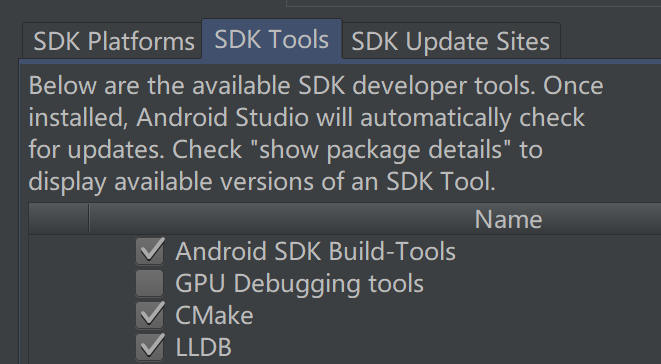
ndk.dir=C\:\\Android\\sdk\\ndk-bundle

1. gradle.properties中添加配置（需要手动添加）

#兼容老的NDK版本

android.useDeprecatedNdk=true

## Android3.0以后最新ndk添加方式：

安装LLDB和CMake

### 1.添加cpu支持类型和CMakeLists.txt路径！！

android {

compileSdkVersion 28

defaultConfig {

applicationId "com.mantoudaren.myjni"

minSdkVersion 21

targetSdkVersion 28

versionCode 1

versionName "1.0"

testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"

externalNativeBuild {

cmake {

cppFlags ""

abiFilters 'x86','x86\_64','armeabi-v7a','arm64-v8a'//填写cpu支持类型

}

}

}

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

}

}

externalNativeBuild {

cmake {

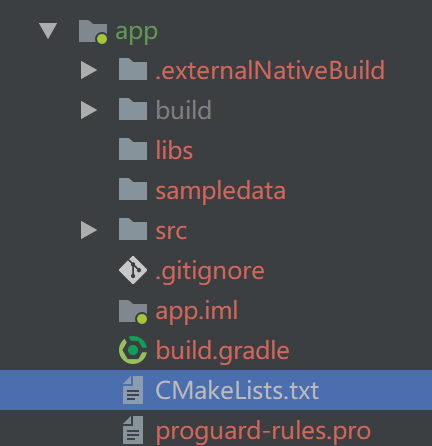
path "CMakeLists.txt"//添加CmakeLists.txt配置文件

}

}

}

### 2.在app目录下，新建file。文件名字为：CMakeLists.txt//和上面的path里面的名字一样



### 3.在CMakeLists.txt里面配置so文件

以CMakeLists.txt的位置为参考点！。相对于CMakeLists.txt，那么Hello.c就在src/main/jni/one目录下！

CMakeLists.txt相当于原来的Android.mk告诉系统C文件在哪里。和生成lib库的名字

cmake\_minimum\_required(VERSION 3.4.1)

add\_library( # Specifies the name of the library.

# 这里是你so的名字。刚才在 MainActivity里面要引用的

# System.*loadLibrary*("Hello");//动态加载链接库的名称

Hello

# Sets the library as a shared library.

SHARED

# Provides a relative path to your source file(s).

#这里是刚才 创建的c++ 代码的名字

#相对于CMakeLists.txt的位置。Hello.c的位置是src/main/jni/one/Hello.c

src/main/jni/one/Hello.c )

find\_library( # Sets the name of the path variable.

log-lib

# Specifies the name of the NDK library that

# you want CMake to locate.

log )

target\_link\_libraries( # Specifies the target library.

# 这里是你so的名字。刚才在 MainActivity里面要引用的 一样。

# System.*loadLibrary*("Hello");//动态加载链接库的名称

Hello

# Links the target library to the log library

# included in the NDK.

${log-lib} )

### 写C代码-生成头文件

在main目录下，新建jni/one目录 创建C代码

/\*\*

\* jstring:返回值//java类型前面加个j

\*

\* Java\_com\_test\_ndkdemo01\_JNI\_sayHello//Java\_全类名

\* com.test.ndkdemo01.JNI.sayHello

\* 把点.变成下划线\_

\*

\* JNIEnv\* env:里面有很多方法//这2个必须有

\* jobject jobj：谁调用了这个方法就是谁的实例 //这2个必须有

\*

\* @return

\*/

//

// Created by mantoudaren on 2018/8/16.

//

#include <jni.h>

#include <stdio.h>

#include <stdlib.h>

JNIEXPORT jstring JNICALL

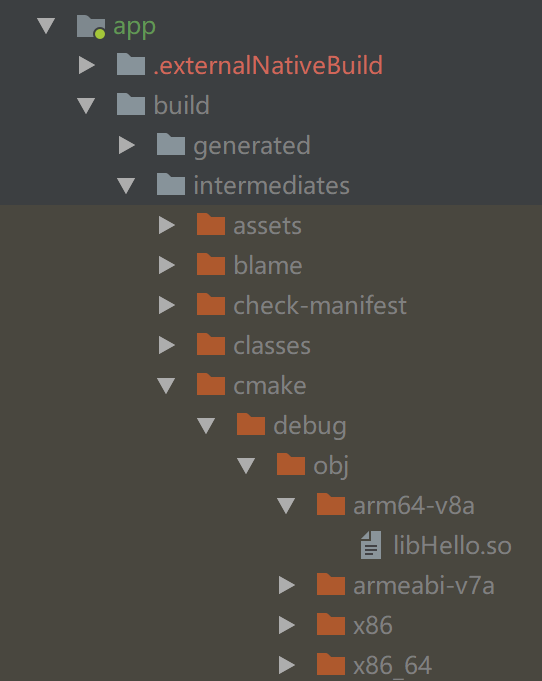
Java\_com\_mantoudaren\_myjni\_JNI\_stringFromC(JNIEnv \*env, jobject instance) {

// *TODO*

**return** (\*env)->NewStringUTF(env, "Hello from C!");

}

### 写Native方法并加载动态链接库

 lib+moduleName+.so文件

在JNI类中动态加载链接库。这个名称必须和moduleName一致

public class JNI {

{

System.*loadLibrary*("Hello");//动态加载链接库，只要在使用JNI类之前加载即可

}

*/\*\**

*\* 定义native方法*

*\* 调用C代码对应的方法*

*\** ***@return***

*\*/*

public native String sayHello();

}

### 使用

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.*activity\_main*);

String s = new JNI().sayHello();//调用java方法，访问C代码

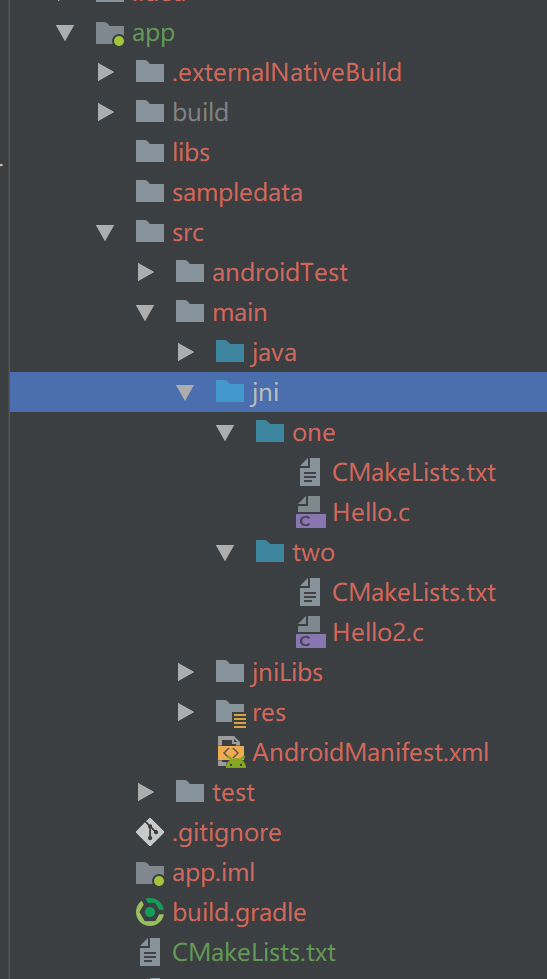
System.*out*.println("result=="+s);

}

## 关于多个C或者C++文件编译多个so的问题

核心思想：多个C文件要想生成多个so，必须要给每一个C文件配置对应的CMakeLists.txt。然后父目录的CMakeLists.txt加载子目录的CMakeLists.txt即可

### 每个C或者C++文件用目录层级单独隔离开。并且每一个C都单独配置CMakeLists.txt如图：



### 配置子目录的CMakeLists.txt

add\_library( # Specifies the name of the library.

# 这里是你so的名字。刚才在 MainActivity里面要引用的

Hello

# Sets the library as a shared library.

SHARED

# Provides a relative path to your source file(s).

#这里是刚才 创建的c++ 代码的名字

#因为Hello.c和子CMakeLists.txt在同一个目录里面。所以这里直接写Hello.c即可！！

Hello.c )

find\_library( # Sets the name of the path variable.

log-lib

# Specifies the name of the NDK library that

# you want CMake to locate.

log )

target\_link\_libraries( # Specifies the target library.

# 这里是你so的名字。刚才在 MainActivity里面要引用的 一样。

Hello

# Links the target library to the log library

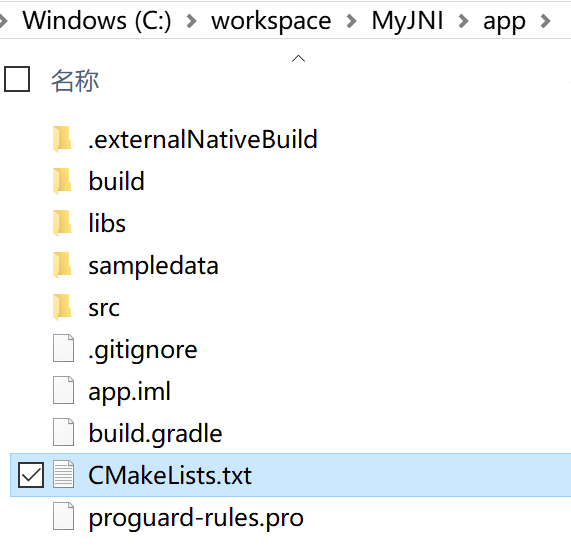
# included in the NDK.

${log-lib} )

### 配置父CMakeLists.txt

ADD\_SUBDIRECTORY(src/main/jni/one)关联子CMakeLists.txt的位置

这里以父CMakeLists.txt的位置为参考点！。相对于父CMakeLists.txt，那么子CMakeLists.txt就在src/main/jni/one目录下！

父CMakeLists.txt的位置和src是平级关系。

cmake\_minimum\_required(VERSION 3.4.1)

set(CMAKE\_LIBRARY\_OUTPUT\_DIRECTORY ${PROJECT\_SOURCE\_DIR}/src/main/jniLibs/${ANDROID\_ABI})

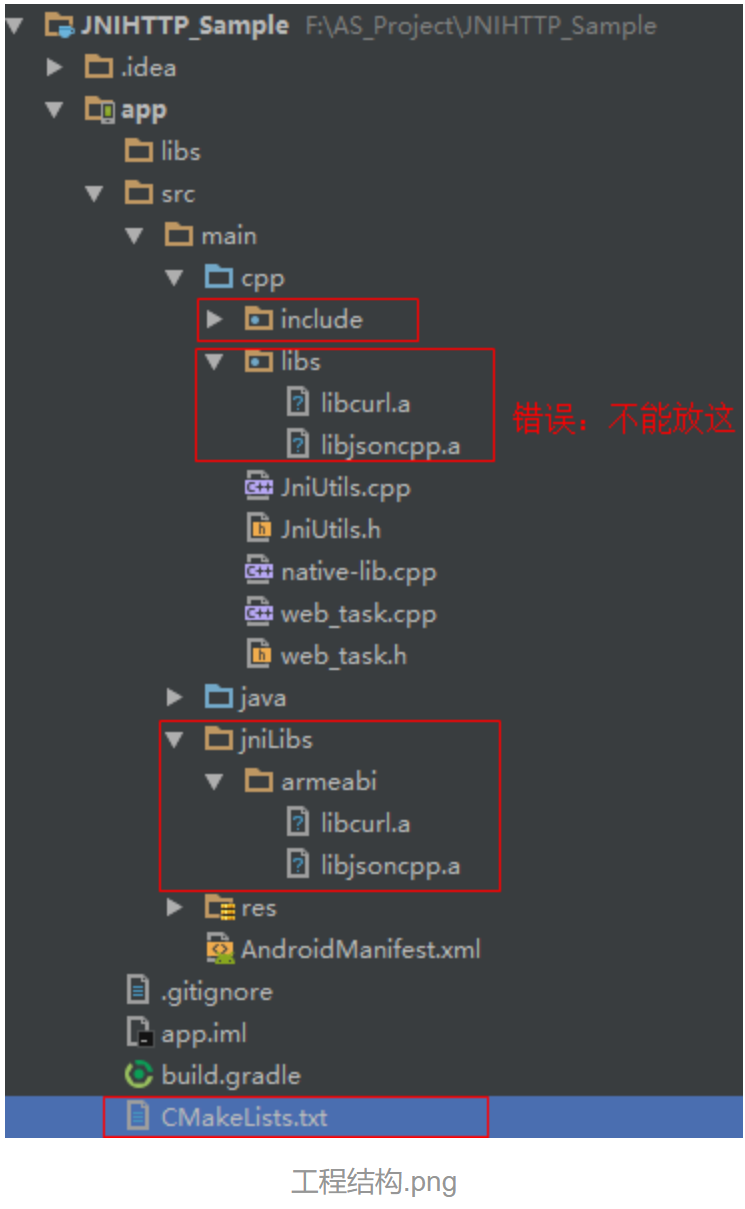
ADD\_SUBDIRECTORY(src/main/jni/one)

ADD\_SUBDIRECTORY(src/main/jni/two)

## CMakeLists.txt加载三方库

通过反编译工具，创建JNI类。注意JNI类和包名要和反编译后一致！！

假如我要依赖的第三方库是：libcurl.a和jsoncpp.a。



cmake\_minimum\_required(VERSION 3.4.1)

add\_library(native-lib

SHARED

src/main/cpp/native-lib.cpp

src/main/cpp/JniUtils.cpp

src/main/cpp/web\_task.cpp )//多个c++文件一起编译到一个名字为native-lib

的so文件

include\_directories( src/main/cpp/include/jsoncpp

src/main/cpp/include/curl

)//专门引入头文件的

add\_library(curl STATIC IMPORTED)//STATIC表示引入的是.a的库。如果是SHARED表示引入的是.so的库

set\_target\_properties(curl

PROPERTIES IMPORTED\_LOCATION

${CMAKE\_SOURCE\_DIR}/src/main/jniLibs/${ANDROID\_ABI}/libcurl.a)//库的拷贝位置在src/main/jniLibs目录下，${ANDROID\_ABI}是CPU类型的支持

add\_library(jsoncpp STATIC IMPORTED)//引入多个就拷贝多个

set\_target\_properties(jsoncpp

PROPERTIES IMPORTED\_LOCATION

${CMAKE\_SOURCE\_DIR}/src/main/jniLibs/${ANDROID\_ABI}/libjsoncpp.a)

# 不能向下面格式写，会出现 Error:error: '../../../../src/main/libs/libjsoncpp.a', needed by '../obj/armeabi/libnative-lib.so', missing and no known rule to make it

# add\_library(curl STATIC IMPORTED)

# set\_target\_properties(curl

# PROPERTIES IMPORTED\_LOCATION

# ${CMAKE\_SOURCE\_DIR}/src/main/libs/libcurl.a)

# add\_library(jsoncpp STATIC IMPORTED)

# set\_target\_properties(jsoncpp

# PROPERTIES IMPORTED\_LOCATION

# ${CMAKE\_SOURCE\_DIR}/src/main/libs/libjsoncpp.a)

find\_library( # Sets the name of the path variable.

log-lib

log )

target\_link\_libraries( # Specifies the target library.

native-lib//需要把引入的每一个库名字加进来

jsoncpp//需要把引入的每一个库名字加进来

curl//需要把引入的每一个库名字加进来

${log-lib} )

### 注意：

1，要注明第三方so文件运用了哪些Cpu

externalNativeBuild {

cmake {

cppFlags ""

abiFilters 'x86\_64','arm64-v8a'//第三方so文件运用了哪些Cpu。这里要写明

}

}

2，在引入第三方so文件之前，必须添加本地native-lib.cpp文件！删除会造成无法编译！

add\_library( # Sets the name of the library.

native-lib

# Sets the library as a shared library.

SHARED

# Provides a relative path to your source file(s).

src/main/cpp/native-lib.cpp )

#上面为初始话本机的native-lib，必须要有！！下面是加载第三方的so！

add\_library(Hello SHARED IMPORTED)

set\_target\_properties(Hello

PROPERTIES IMPORTED\_LOCATION

${CMAKE\_SOURCE\_DIR}/src/main/jniLibs/${ANDROID\_ABI}/libHello.so)

3，main目录下，创建cpp目录，下面创建native-lib.cpp。



4，native-lib.cpp里面可以为空，但是不能没有！可以不在类中创建native方法，也可以不加载System.loadlibary（ native-lib）

5，所有第三方和本地库的cpu必须全部保持一致

建议是：

* 为了减小 apk 体积，只保留 armeabi 和 armeabi-v7a 两个文件夹，并保证这两个文件夹中 .so 数量一致
* 对只提供 armeabi 版本的第三方 .so，原样复制一份到 armeabi-v7a 文件夹

## 动态生成C语言方法

### 老方法：

/\* DO NOT EDIT THIS FILE - it is machine generated \*/

#include <jni.h>

/\* Header for class com\_test\_javacallc\_MyJNI \*/

#ifndef \_Included\_com\_test\_javacallc\_MyJNI

#define \_Included\_com\_test\_javacallc\_MyJNI

#ifdef \_\_cplusplus

**extern** "C" {

#endif

/\*

\* Class: com\_test\_javacallc\_MyJNI

\* Method: add

\* Signature: (II)I

\*/

JNIEXPORT jint JNICALL Java\_com\_test\_javacallc\_MyJNI\_add

(JNIEnv \*, jobject, jint, jint);//方法应该怎么写

/\*

\* Class: com\_test\_javacallc\_MyJNI

\* Method: sayHello

\* Signature: (Ljava/lang/String;)Ljava/lang/String;

\*/

JNIEXPORT jstring JNICALL Java\_com\_test\_javacallc\_MyJNI\_sayHello

(JNIEnv \*, jobject, jstring);

/\*

\* Class: com\_test\_javacallc\_MyJNI

\* Method: increaseArrayEles

\* Signature: ([I)[I

\*/

JNIEXPORT jintArray JNICALL Java\_com\_test\_javacallc\_MyJNI\_increaseArrayEles

(JNIEnv \*, jobject, jintArray);

/\*

\* Class: com\_test\_javacallc\_MyJNI

\* Method: checkPwd

\* Signature: (Ljava/lang/String;)I

\*/

JNIEXPORT jint JNICALL Java\_com\_test\_javacallc\_MyJNI\_checkPwd

(JNIEnv \*, jobject, jstring);

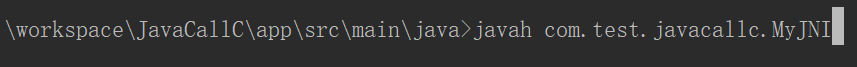
#ifdef \_\_cplusplus

}

#endif

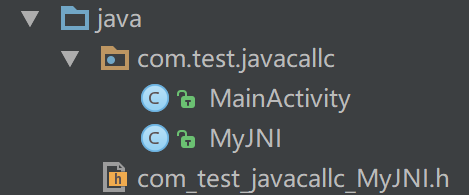
#endif

1. 在NDKDemo\app\src\main\java



javah com.test.javacallc.MyJNI

Javah 空格《native方法所在的全类名》

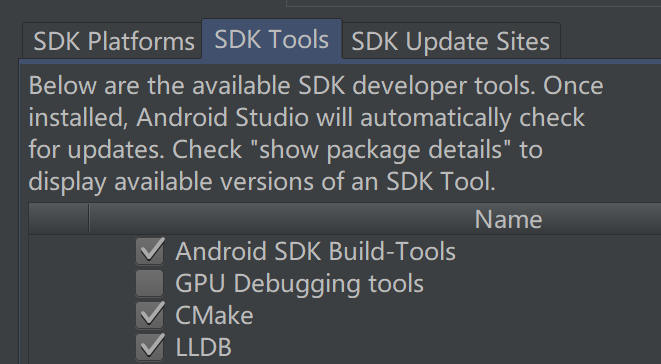


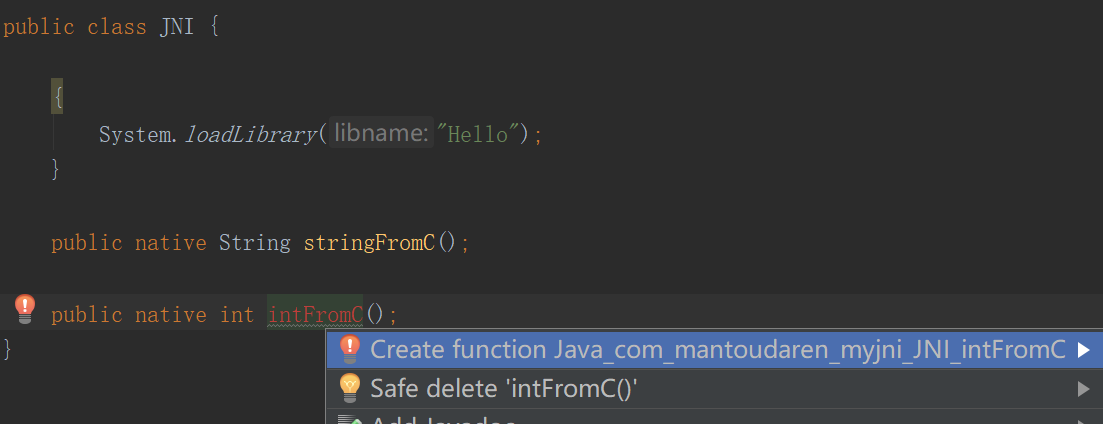
1. 在项目build/intermaediates/classes/debug/下执行

Javah –classpath . –jni空格《native方法所在的全类名》



### 3.0后：

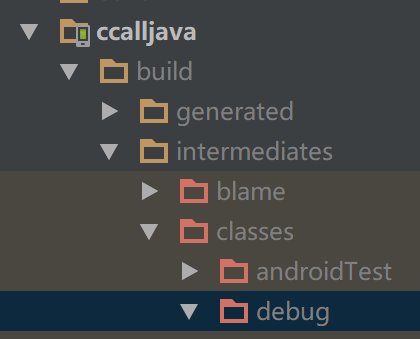
安装LLDB和CMake后，直接自动生成！



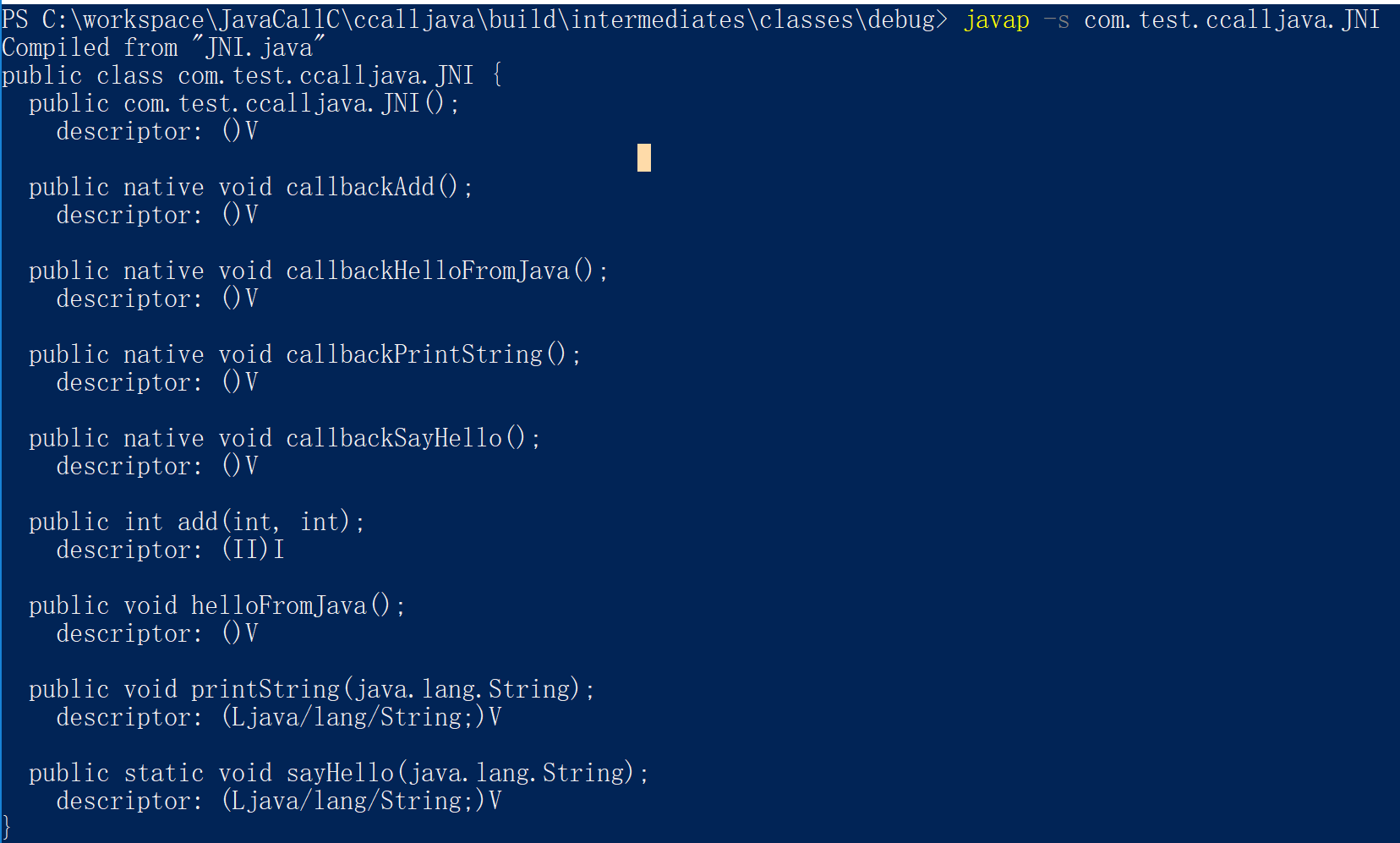
## 查看方法签名

方法签名的作用是：当C调JAVA时候，通过反射获取方法的时候，需要传入方法签名！

jmethodID mid = (\*env)->GetMethodID(env,clsstring, "getBytes", "(Ljava/lang/String;)[B");//最后一个参数就是方法签名

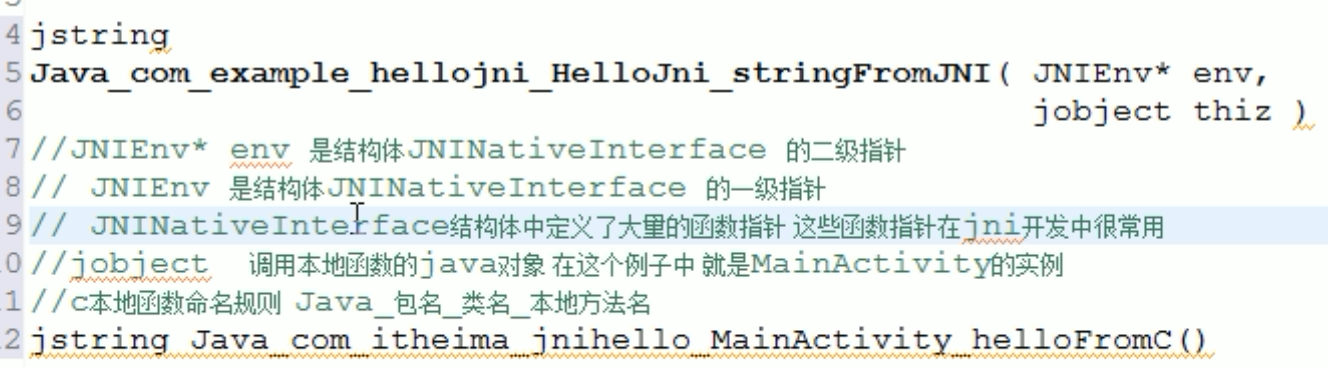
在debug目录下，输入javap –s 全类名即可查看签名

方法签名是在class下去查看。所以方法写入后，必须编译成class，才能查看到签名



## JAVA调用C

解释了为什么是 (\*env)->函数指针



protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.*activity\_main*);

int add = new MyJNI().add(10, 22);//在java中直接执行native方法即可

System.*out*.println("result==" + add);

String str = new MyJNI().sayHello("我是");

System.*out*.println("result==" + str);

int[] array = {1,2,3};

int[] resultArray = new MyJNI().increaseArrayEles(array);

for (int i = 0; i < resultArray.length; i++) {

System.*out*.println("result==" + resultArray[i]);

}

int checkPwd = new MyJNI().checkPwd("123456");

System.*out*.println("checkPwd==" + checkPwd);

}

#include "com\_test\_javacallc\_MyJNI.h"

#include <string.h>

/\*\*

\* jstring to char\* jstring转换char\*

\* @param env

\* @param jstr

\* @return

\*/

**char**\* \_JString2CStr(JNIEnv\* env, jstring jstr)

{

**char**\* rtn = NULL;

jclass clsstring = (\*env)->FindClass(env,"java/lang/String");

jstring strencode = (\*env)->NewStringUTF(env,"utf-8");

jmethodID mid = (\*env)->GetMethodID(env,clsstring, "getBytes", "(Ljava/lang/String;)[B");

jbyteArray barr= (jbyteArray)(\*env)->CallObjectMethod(env,jstr, mid, strencode);

jsize alen = (\*env)->GetArrayLength(env,barr);

jbyte\* ba = (\*env)->GetByteArrayElements(env,barr, JNI\_FALSE);

**if** (alen > 0)

{

rtn = (**char**\*)malloc(alen + 1);

memcpy(rtn, ba, alen);

rtn[alen] = 0;

}

(\*env)->ReleaseByteArrayElements(env,barr, ba, 0);

**return** rtn;

}

jint Java\_com\_test\_javacallc\_MyJNI\_add(JNIEnv \*env, jobject jobj, jint ji, jint jj) {

**int** result = ji + jj;

**return** result;//返回2个数字相加

}

JNIEXPORT jstring JNICALL Java\_com\_test\_javacallc\_MyJNI\_sayHello

(JNIEnv \*env, jobject jobj, jstring str) {

//jstring要转换成char\*

**char** \*fromJava = \_JString2CStr(env, str);

**char** \*fromC = " add I am from C";

strcat(fromJava, fromC);//把拼接的结果放在第一个参数

**return** (\*env)->NewStringUTF(env, fromJava);//最后把char转换成jstring

}

JNIEXPORT jintArray JNICALL Java\_com\_test\_javacallc\_MyJNI\_increaseArrayEles

(JNIEnv \* env, jobject jobj, jintArray jarray){

jsize size = (\*env)->GetArrayLength(env,jarray);//取数组长度

jint \*intArray = (\*env)->GetIntArrayElements(env,jarray, JNI\_FALSE);//取数组每个元素

**int** i;

**for** (i = 0; i < size; ++i) {

\*(intArray+i) += 10;//数组每个元素内容加10

}

**return** jarray;

}

JNIEXPORT jint JNICALL Java\_com\_test\_javacallc\_MyJNI\_checkPwd

(JNIEnv \* env, jobject jobj, jstring jstr){

//假如服务器的密码是123456

**char**\* origin = "123456";

**char**\* fromUser = \_JString2CStr(env,jstr);

//比较字符串函数

**int** code = strcmp(origin,fromUser);

**if**(code==0){

**return** 200;

} **else**{

**return** 400;

}

}

## C调用JAVA

核心：在C语言中通过反射来调用JAVA中的代码

一般只有在C代码中有耗时操作。然后需要回调java，给用户显示耗时进度。的时候，才用C调JAVA

### 调用java，有参有返回值

如果本地方法和要回调的java方法在同一个类里可以直接用 jni传过来的java对象 就是jobj

其中第三步：可以直接使用jobj。因为谁调用了JNI，jobj就是谁。Jobj就等同于JNI对象

    \* 当回调的方法跟本地方法不在一个类里 需要通过刚创建的字节码对象手动创建一个java对象。调用创建的Method

    \* jobject obj =(\*env)->AllocObject(env,claz);

    \* 再通过这个对象来回调java的方法

需要注意的是 如果创建的是一个activity对象 回调的方法还包含上下文 这个方法行不通!!!会报空指针异常

解决办法就是：

方式一：

把native方法，拷贝到activity对象下面，然后直接使用jobj。而不是通过反射取activity对象！！

方式二：

把activtiy里面需要更新UI的方法拷贝到JNI类里面，把context通过JNI构造方法传进去，然后还是直接使用jobj

/\*\*

\* 让C代码调用java中的add方法

\* Java中JNI类的 public int add(int x, int y)方法

\*/

**void** Java\_com\_test\_ccalljava\_JNI\_callbackAdd

(JNIEnv \* env, jobject jobj) {

//1.反射得到字节码

//jclass (\*FindClass)(JNIEnv\* , const char\*)

jclass jclzz = (\*env)->FindClass(env, "com/test/ccalljava/JNI");//注意全类名的.用/分隔

//2.反射得到方法

//jmethodID (\*GetMethodID)(JNIEnv\*,jclass,const char\*,const char\*)

//env,类的字节码，方法名，参数,方法签名！

//方法有2个形参(int,int)而且返回值为int 所以是(II)I

jmethodID jmethod = (\*env)->GetMethodID(env, jclzz, "add", "(II)I");

//3.反射得到实例化对象

//jobject (\*AllocObject)(JNIEnv\*,jclass)

jobject object = (\*env)->AllocObject(env, jclzz);//这里可以直接使用参数的jobj

//4.通过反射调用方法

//jint (\*CallIntMethod)(JNIEnv\*,jobject,jmetodID,...)

jint result = (\*env)->CallIntMethod(env, object, jmethod,22, 33);

//成功调用了public int add(int x, int y)方法

LOGI("result == %d\n",result);

};

### 调用java，无参无返回值

/\*\*

\* 让C代码调用java中的helloFromJava方法

\* Java中JNI类的 public void helloFromJava()方法

\*/

JNIEXPORT **void** JNICALL

Java\_com\_test\_ccalljava\_JNI\_callbackHelloFromJava

(JNIEnv\* env,jobject jobj){

jclass jclzz = (\*env)->FindClass(env, "com/test/ccalljava/JNI");

jmethodID jmethod = (\*env)->GetMethodID(env, jclzz, "helloFromJava", "()V");

jobject object = (\*env)->AllocObject(env, jclzz);

//void (\*CallVoidMethod)(JNIEnv\*, jobject, jmethodID, ...);

//没有参数，最后一个就不写！！

(\*env)->CallVoidMethod(env, object, jmethod);

};

### 调用java，有参无返回值

/\*\*

\* 让C代码调用java中的printString方法

\* Java中JNI类的 public void printString(String s)方法

\*/

JNIEXPORT **void** JNICALL

Java\_com\_test\_ccalljava\_JNI\_callbackPrintString

(JNIEnv\* env,jobject jobj){

jclass jclzz = (\*env)->FindClass(env, "com/test/ccalljava/JNI");

jmethodID jmethod = (\*env)->GetMethodID(env, jclzz, "printString", "(Ljava/lang/String;)V");

jobject object = (\*env)->AllocObject(env, jclzz);

//void (\*CallVoidMethod)(JNIEnv\*, jobject, jmethodID, ...);

**char**\* cStr = "asdfasdfsf";

//jstring (\*NewStringUTF)(JNIEnv\*, const char\*); char转jstring

(\*env)->CallVoidMethod(env, object, jmethod,(\*env)->NewStringUTF(env,cStr));

};

### 调用java，静态方法，有参无返回值

/\*\*

\* 让C代码调用java中的static方法

\* Java中JNI类的 public static void sayHello(String s)方法

\*/

JNIEXPORT **void** JNICALL Java\_com\_test\_ccalljava\_JNI\_callbackSayHello

(JNIEnv \* env, jobject jobj){

jclass jclzz = (\*env)->FindClass(env, "com/test/ccalljava/JNI");

//jmethodID (\*GetStaticMethodID)(JNIEnv\*, jclass, const char\*, const char\*);

jmethodID jmethod = (\*env)->GetStaticMethodID(env, jclzz, "sayHello", "(Ljava/lang/String;)V");

//jobject object = (\*env)->AllocObject(env, jclzz);//静态方法不需要实例化

//void (\*CallStaticVoidMethod)(JNIEnv\*, jclass, jmethodID, ...);

**char**\* cStr = "HAHA";

//jstring (\*NewStringUTF)(JNIEnv\*, const char\*); char转jstring

(\*env)->CallStaticVoidMethod(env,jclzz,jmethod,(\*env)->NewStringUTF(env,cStr));

};

## 打印C代码log

1. 注册log

ndk{

moduleName "ccalljava"//so文件：lib+moduleName+.so

abiFilters "armeabi","armeabi-v7a","x86"//cpu类型

}

ndk{

ldLibs "log"//注册LOG

}

1. 拷贝到C代码下

#include <android/log.h>

// 宏定义类似java 层的定义,不同级别的Log LOGI, LOGD, LOGW, LOGE, LOGF。 对就Java中的 Log.i log.d

#define LOG\_TAG "MyLogC" // 这个是自定义的LOG的标识

//#undef LOG // 取消默认的LOG

#define LOGI(...) \_\_android\_log\_print(ANDROID\_LOG\_INFO,LOG\_TAG, \_\_VA\_ARGS\_\_)

#define LOGD(...) \_\_android\_log\_print(ANDROID\_LOG\_DEBUG,LOG\_TAG, \_\_VA\_ARGS\_\_)

#define LOGW(...) \_\_android\_log\_print(ANDROID\_LOG\_WARN,LOG\_TAG, \_\_VA\_ARGS\_\_)

#define LOGE(...) \_\_android\_log\_print(ANDROID\_LOG\_ERROR,LOG\_TAG, \_\_VA\_ARGS\_\_)

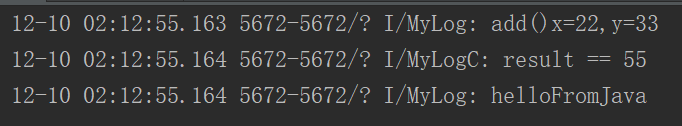
#define LOGF(...) \_\_android\_log\_print(ANDROID\_LOG\_FATAL,LOG\_TAG, \_\_VA\_ARGS\_\_)

1. 使用LOG

jint result = (\*env)->CallIntMethod(env, object, jmethod,22, 33);

//成功调用了public int add(int x, int y)方法

LOGI("result == %d\n",result);//按照C语言打印的方式展示LOG内容

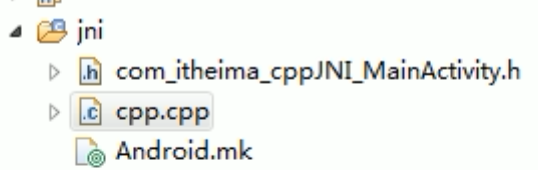
## C++

C++当中的env是一级指针，所以直接使用env->newStringUTF调用方法

C当中env是二级指针，所以使用(\*env)->newStringUTF调用方法



C++的方法需要声明，所以需要引入头文件。

把头文件和CPP放在一起即可！

## JNI中更新UI

核心思想。就是让更新UI的方法和native绑定在一起！

需要注意的是 如果创建的是一个activity对象 回调的方法还包含上下文 这个方法行不通!!!会报空指针异常

解决办法就是：把native方法，拷贝到activity对象下面，然后直接使用jobj。而不是通过反射取activity对象！！

### Native方法拷贝到UI类下面：

1.在Activity中直接填写native方法

private native void callbackshowToast();//让native方法和activity关联！

private void showToast(){

Toast.*makeText*(MainActivity.this,"111",Toast.*LENGTH\_SHORT*).show();

}

2.在C代码反射中，使用jobject

谁调用了native方法，jobject就是谁

因为native方法和showToast方法在同一个类中，所以直接使用jobj

JNIEXPORT **void** JNICALL

Java\_com\_test\_ccalljava\_MainActivity\_callbackShowToast

(JNIEnv\* env,jobject jobj){//使用jobj对象

jclass jclzz = (\*env)->FindClass(env, "com/test/ccalljava/MainActivity");

jmethodID jmethod = (\*env)->GetMethodID(env, jclzz, "ShowToast", "()V");

//jobject object = (\*env)->AllocObject(env, jclzz); //activity不能当成普通类来new！！

//void (\*CallVoidMethod)(JNIEnv\*, jobject, jmethodID, ...);

//没有参数，最后一个就不写！！

(\*env)->CallVoidMethod(env, jobj, jmethod);//activity调用了native方法，所以直接使用jobj

};

### UI方法拷贝到JNI类下面：

通过构造方法，把context传进入

public class JNI {

{

System.*loadLibrary*("Hello");

}

private Context mContext;

public JNI(Context context) {

mContext = context;

}

public void showToast(){//把要更新UI的方法，拷贝到JNI类下面

Toast.*makeText*(mContext,"更新UI",Toast.*LENGTH\_SHORT*).show();

}

public native String stringFromC();

}

因为native方法和showToast方法在同一个类中，所以直接使用jobj