Android高级进阶

笔记本： Android
创建时间： 2015/12/15 14:52
标签： Java

Android高级进阶

2015年12月15日
14:52

一java代码优化

况下尤其如此，因为在 Android 2.2 中引入了实时（JIT）编译器。Dalvik JIT 编译器把 Dalvik 字节码编译成本地代码，这可以明显加快执行速度。JIT 编译器（有时简称 JIT）可以显著提高性能，因为：

- 本地代码直接由 CPU 执行，而不必由虚拟机解释执行；
- 本地代码可以为特定架构予以优化。

1，用循环迭代代替递归

递归算法需要消耗大量的栈控件，会产生过多的方法调用，严重影响效率，还有可能导致栈溢出。因此尽量使用循环迭代来实现相同的功能。

代码清单 1-1　简单的斐波那契数列递归实现

```
public class Fibonacci {
    public static long computeRecursively (int n)
    {
        if (n > 1) return computeRecursively(n-2) + computeRecursively(n-1);
        return n;
    }
}
```

代码清单 1-5　修改后的斐波那契数列的迭代实现

```
public class Fibonacci {
    public static long computeIterativelyFaster (int n)
    {
        if (n > 1) {
            long a, b = 1;
            n--;
            a = n & 1;
            n /= 2;
            while (n-- > 0) {
                a += b;
                b += a;
            }
            return b;
        }
        return n;
    }
}
```

2，android自带类和java类的选择

Android 定义了 SparseArray 类，当键是整数时，它比 HashMap 效率更高。因为 HashMap 使用的是 java.lang.Integer 对象，而 SparseArray 使用的是基本类型 int。因此使用 HashMap 会创建很多 Integer 对象，而使用 SparseArray 则可以避免。

Android 定义了多种类型的稀疏数组（sparse array）：SparseArray（键为整数，值为对象）、SparseBooleanArray（键为整数，值为 boolean）和 SparseIntArray（键为整数，值为整数）。

另外，如果不需要在数据结构内部处理同步，应该使用Arraylist而不是Vector。

一般情况下，你应该很熟悉 java.util 和 android.util 包，几乎所有的组件依赖这两个工具箱。每当新的 Android 版本发布，你都应该特别注意这些包的修改（添加和修改的类）并参考“API 的变化报告”，详见 http://d.android.com/sdk。我们将在第 5 章讨论 java.util.concurrent 中更多的数

Activity 的 onCreate() 方法一般会包含调用 setContentView 或任何其他负责展开资源的方法。因为展开资源是一个开销相对较大的操作，所以你可以通过降低布局（Layout，XML 文件定义应用的外观）复杂性来使资源展开加快。几个降低布局复杂性的步骤如下。

- 使用 RelativeLayout 代替嵌套 LinearLayouts，尽可能保持“扁平化”的布局。此外，减少创建的对象数量，也会让事件的处理速度加快。
- 使用 ViewStub 推迟对象创建（见 1.6.1 节）。

3，SQLite

使用一个数据库操作语句，而只改变参数，这样可以减少对sql语句的编译时间：

代码清单 1-21　SQLite 语句的编译

```
public void populateWithCompileStatement () {
    SQLiteStatement stmt = db.compileStatement("INSERT INTO cheese VALUES(?,?)");
    int i = 0;
    for (String name : sCheeseNames) {
        String origin = sCheeseOrigins[i++];
        stmt.clearBindings();
        stmt.bindString(1, name); // 替换第一个问号为 name
        stmt.bindString(2, origin); // 替换第二个问号为 origin
        stmt.executeInsert();
    }
}
```

因为只进行一次语句编译，而不是 650 次，并且绑定值是比编译更轻量的操作，所以这种方法明显快多了，建立数据库只用了 268 毫秒。这样还使代码更具可读性。

另外多个连续的操作尽量在一个事务过程中，这样避免创建事务和提交事务的多重时间，一次性事务是解决问题的较好办法。

```
public void populateWithCompileStatementOneTransaction () {
    try {
        db.beginTransaction();
        SQLiteStatement stmt = db.compileStatement("INSERT INTO cheese VALUES(?,?)");
        int i = 0;
        for (String name : sCheeseNames) {
            String origin = sCheeseOrigins[i++];
            stmt.clearBindings();
            stmt.bindString(1, name); // 替换第一个问号为 name
            stmt. bindString(2, origin);// 替换第二个问号为 origin
            stmt.executeInsert();
        }
        db.setTransactionSuccessful(); // 删除这一调用不会提交任何改动!
    } catch (Exception e) {
        // 在这里处理异常
    } finally {
        db.endTransaction(); //必须写在 finally 块
    }
}
```

二android NDK

1，NDK使用步骤

NDK 是以下 6 个组件的集合：

- 文档
- 头文件
- C/C++文件
- 预编译库
- 编译、链接、分析和调试代码的工具
- 示例应用程序

就定义而言，本地代码是具体到某个架构的。例如，Intel CPU 不接受 ARM 指令，反之亦然。因此，NDK 包括多个平台以及不同版本的预编译库。NDK r7 支持以下三个应用程序二进制接口（ABI）：

- armeabi
- armeabi-v7a
- x86

从 Java 调用 C/C++函数其实很简单，只需以下几个步骤。
(1) 必须在 Java 代码中声明本地方法。
(2) 需要实现 Java 本地接口（JNI）粘合层。
(3) 必须创建 Android makefile 文件。
(4) 必须用 C/C++实现本地方法。
(5) 必须编译本地库。
(6) 必须加载本地库。

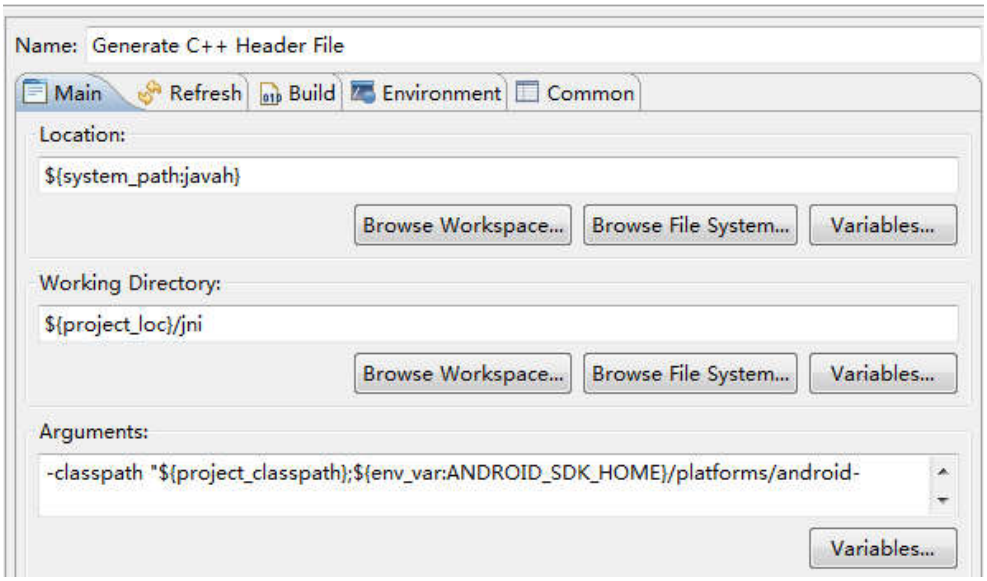
1，java代码中声明要使用的本地方法(C++实现)

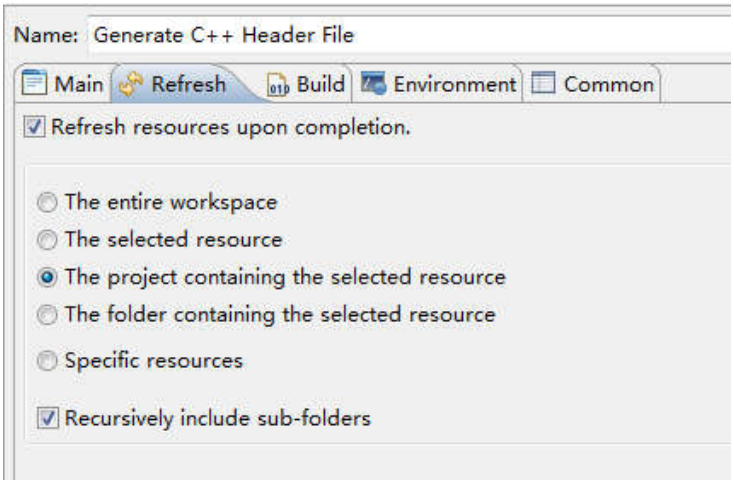
```
public class Fibonacci {
    static {
        System.loadLibrary("fibonacci"); // 加载 libfibonacci.so
    }

    public static native long recursiveNative (int n);
}
```

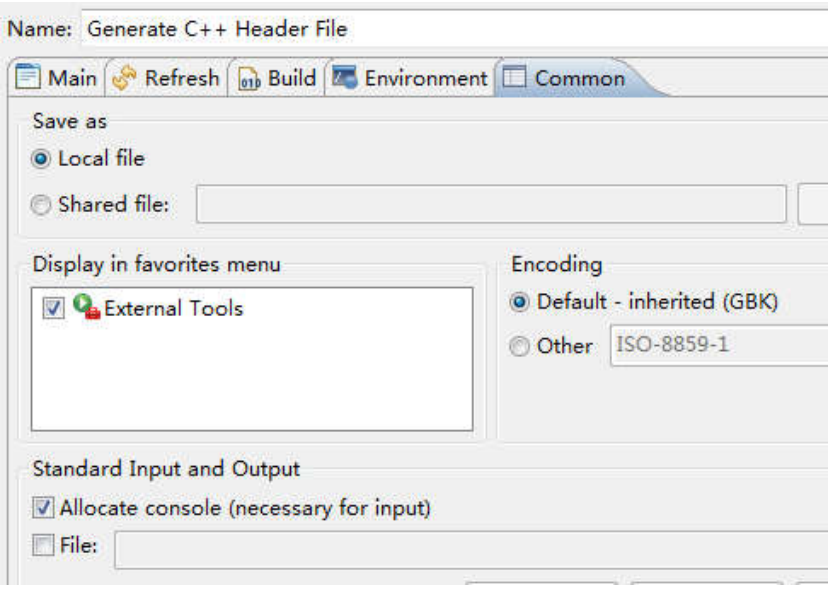
2，实现JNI粘合层：使用javah命令，生成的头文件中方法名为包名+方法名，之间用_连接

```
cd ~/workspace/MyFibonacciApp
mkdir jni
javah -classpath bin -jni -d jni com.apress.proandroid.Fibonacci
```

若使用android studio，classpath为HugeAPP\app2\build\intermediates\classes\debug
在eclipse中配置javah调用命令，这样就不需要每次输入命令了：







3，实现C++源文件：实现本地方法

```
#include "com_apress_proandroid_Fibonacci.h"
#include <stdint.h>

static uint64_t recursive (unsigned int n) {
     if (n > 1) return recursive(n-2) + recursive(n-1);
     return n;
}
/*
 * Class:     com_apress_proandroid_Fibonacci
 * Method:    recursiveNative
 * Signature: (I)J
 */
jlong JNICALL
Java_com_apress_proandroid_Fibonacci_recursiveNative
   (JNIEnv *env, jclass clazz, jint n) {
     return recursive(n);
}
```

JNI层的所有函数有一个共同点：它们的第一个参数都是JNIEnv*类型（JNIEnv对象的指针）。JNIEnv的对象是JNI环境本身，使用它可以与虚拟机交互（如果需要的话）。第二个参数，当方法被声明为静态时为jclass类型，否则为jobject类型。

4，创建Application.mk和android.mk：在jni目录下，同样头文件和源文件也在该目录下。之后进行编译就可以了。编译之后生成.so本地库，就可以调用其中的本地方法了。

为防止调用本地库时失败，可以提供一个默认的java实现，当找不到C++本地方法时，调用java方法：

```
public class Fibonacci {
    private static final boolean useNative;
    static {
        boolean success;
        try {
            System.loadLibrary("fibonacci"); // 加载 libfibonacci.so
            success = true;
        } catch (Throwable e) {
            success = false;
        }
        useNative = success;
    }

     public static long recursive (int n) {
        if (useNative) return recursiveNative(n);
        return recursiveJava(n);
    }

    private static long recursiveJava (int n) {
        if (n > 1) return recursiveJava(n-2) + recursiveJava(n-1);
        return n;
    }

    private static native long recursiveNative (int n);
}
```

2，Application.mk/android.mk

2.1 变量和格式

表2-3　Application.mk的变量

变　量	含　义
APP_PROJECT_PATH	项目路径
APP_MODULES	模块编译列表
APP_OPTIM	设置程序为“release”或“debug”版
APP_CFLAGS	C/C++编译器选项
APP_CXXFLAGS	废弃，用APP_CPPFLAGS替代
APP_CPPFLAGS	C++编译器选项
APP_BUILD_SCRIPT	除了jni/Android.mk外，使用其他构建脚本
APP_ABI	编译代码输出的ABI列表
APP_STL	指定使用哪种C++标准模板库，可选择“system”、“stlport_static”、“stlport_shared”或“gnustl_static”
STLPORT_FORCE_REBUILD	如果打算用代码构建STLport而不是使用预编译库，设置为true

表2-5　Android.mk可以定义的变量

变　量	含　义
LOCAL_PATH	Android.mk的路径，可以设置为$(call my-dir)
LOCAL_MODULE	模块名称
LOCAL_MODULE_FILENAME	重新定义库的名称（可选）
LOCAL_SRC_FILES	模块要编译的文件列表
LOCAL_CPP_EXTENSION	重新定义C++源文件的扩展名（默认是.cpp）
LOCAL_C_INCLUDES	追加到include搜索路径的路径列表
LOCAL_CFLAGS	C和C++文件的编译器选项
LOCAL_CXXFLAGS	废弃，用LOCAL_CPPFLAGS替代
LOCAL_CPPFLAGS	C++文件编译器选项
LOCAL_STATIC_LIBRARIES	模块链接的静态库列表
LOCAL_SHARED_LIBRARIES	模块运行时依赖的共享库列表
LOCAL_WHOLE_STATIC_LIBRARIES	和LOCAL_STATIC_LIBRARIES相似，不过使用的是--whole-archive选项
LOCAL_LDLIBS	其他链接选项代码清单（例如，用-lGLESv2链接OpenGL ES 2.0库）
LOCAL_ALLOW_UNDEFINED_SYMBOLS	这个变量设置为true（默认为false）允许未定义的符号
LOCAL_ARM_MODE	编译的指令模式（ARM或Thumb）
LOCAL_ARM_NEON	允许使用NEON高级SIMD指令/内联
LOCAL_DISABLE_NO_EXECUTE	禁用NX位（默认是false，即启用NX）
LOCAL_EXPORT_CFLAGS LOCAL_EXPORT_CPPFLAGS LOCAL_EXPORT_C_INCLUDES LOCAL_EXPORT_LDLIBS	导出变量到依赖此模块的模块（就是在LOCAL_STATIC_LIBRARY或LOCAL_SHARED_LIBRARY里列出此模块）
LOCAL_FILTER_ASM	允许执行shell命令以过滤汇编文件

代码清单 2-13　Application.mk 指定三种 ABI

```
APP_ABI := armeabi armeabi-v7a x86
```

代码清单 2-16　在 Android.mk 中指定两个模块

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := fibonacci
LOCAL_ARM_MODE := thumb
LOCAL_SRC_FILES := com_apress_proandroid_Fibonacci.c fibonacci.c
include $(BUILD_SHARED_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := fibonarmcci
LOCAL_ARM_MODE := arm
LOCAL_SRC_FILES := com_apress_proandroid_Fibonacci.c fibonacci.c
include $(BUILD_SHARED_LIBRARY)
```

Application.mk 用于指定整个应用的共同变量，Android.mk 用来指定想构建什么模块以及如何构建，具体细节比较琐碎。表 2-5 列出了 NDK r6 中可用的变量。

2.2 高级配置

编译静态库：

```
LOCAL_PATH := $(call my-dir)

#
# 3rd party AVI library
#
include $(CLEAR_VARS)

LOCAL_MODULE    := avilib
LOCAL_SRC_FILES := avilib.c platform_posix.c

include $(BUILD_STATIC_LIBRARY)
```

使用已经编译好的so库重新构建：

```
LOCAL_PATH := $(call my-dir)

#
# 3rd party prebuilt AVI library
#
include $(CLEAR_VARS)

LOCAL_MODULE    := avilib
LOCAL_SRC_FILES := libavilib.so

include $(PREBUILT_SHARED_LIBRARY)
```

编译linux程序，可以直接复制到android中使用：

```
include $(CLEAR_VARS)

LOCAL_MODULE    := module
LOCAL_SRC_FILES := module.c

LOCAL_STATIC_LIBRARIES := avilib

include $(BUILD_EXECUTABLE)
```

3，JNI数据类型

3.1 java类型和JNI中本地类型的对应

Table 3-1. Java Primitive Data Types

Java Type	JNI Type	C/C++ Type	Size
Boolean	Jboolean	unsigned char	Unsigned 8 bits
Byte	Jbyte	char	Signed 8 bits
Char	Jchar	unsigned short	Unsigned 16 bits
Short	Jshort	short	Signed 16 bits
Int	Jint	Int	Signed 32 bits
Long	Jlong	long long	Signed 64 bits
Float	Jfloat	float	32 bits
Double	Jdouble	double	64 bits

Table 3-2. Java Reference Type Mapping

Java Type	Native Type
java.lang.Class	jclass
java.lang.Throwable	jthrowable
java.lang.String	jstring
Other objects	jobject
java.lang.Object[]	jobjectArray
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
short[]	jshortArray
int[]	jintArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
Other arrays	Jarray

3.2 String类

- 新建字符串：

```
jstring javaString;
javaString = (*env)->NewStringUTF(env, "Hello World!");
```

- 在C++中使用string，常会导致性能问题，建议将java字符串转化为C语言的Char*后进行运算：

```
// JNI 粘合层 (C 文件中)
void JNICALL Java_com_apress_proandroid_MyClass_doSomethingWithString
  (JNIEnv *env, jclass clazz, jstring s)
{
    const char* str = (*env)->GetStringUTFChars(env, s, NULL);
    if (str != NULL) {
        // 用 str 字符串在这里做些事情

        // 记得释放字符串，不要犯内存泄漏这个常见错误
        (*env)->ReleaseStringUTFChars(env, s, str);
    }
}
```

内存分配从来不是免费午餐，应该在代码中尽可能使用 GetStringRegion 和 GetStringUTFRegion。这样做，可以得到以下好处：

- 避免可能的内存分配；
- 复制 String 要用的一部分到预先分配的缓冲区（可能在栈上）；
- 不需要释放字符串，避免了忘记释放字符串的风险。

3.3 数组类型

- 新建数组：

```
jintArray javaArray;
javaArray = (*env)->NewIntArray(env, 10);
if (0 != javaArray) {
    /* You can now use the array. */
}
```

- 访问数组元素：

方法1：转化成C数组格式，完成操作后再转化为ava数组

```
jint nativeArray[10];
(*env)->GetIntArrayRegion(env, javaArray, 0, 10, nativeArray);

(*env)->SetIntArrayRegion(env, javaArray, 0, 10, nativeArray);
```

方法2：直接获取数组的指针，通过指针进行操作

```
jint* nativeDirectArray;
jboolean isCopy;

nativeDirectArray = (*env)->GetIntArrayElements(env, javaArray, &isCopy);


(*env)->ReleaseIntArrayElements(env, javaArray, nativeDirectArray, 0);
```

4 域和方法的相互调用

4.1 获取java中的变量

JNI中的域指的是java代码中定义的变量，静态变量的成为静态域，非静态的变量成为实例域。这两种域都可以在本地C++代码中调用。

```
public class JavaClass {
    /** Instance field */
    private String instanceField = "Instance Field";

    /** Static field */
    private static String staticField = "Static Field";
```

下面是针对这两种变量的获取方法：

首先需要获取class对象：

Listing 3-21. Getting the Class from an Object Reference

```
jclass clazz;
clazz = (*env)->GetObjectClass(env, instance);
```

实例域：
获取FiledID：
```
jfieldID instanceFiledID=(*env)->getFieldID(env,class,"instanceField","Lava/lang/String;");
```
通过FieldID获取变量：
```
Jstring instanceField=(*env)->GetObjectField(env,instance, instanceFiledID);
```
静态域：
获取FieldID：
```
jfieldID staticFieldID=(*env)->getStaticFieldID(env,clazz,"staticField","Ljava/lang/String;");
```
通过FieldID获取静态变量
```
Jstring staticField=(*env)->GetStaticObjectField(env,clazz, staticFieldID);
```

4.2 调用java中的方法

和变量一样，方法分为实例方法和静态方法，其调用方式也有一定的差别。

```
Public class JavaClass{
    Private String instanceMethod(){ return "hahaha";}
    Private static String staticMethod(){ return "heehhe";}
}
```

和获取变量一样，首先需要获取class对象

实例方法：
获取MethodID：
```
jmethodID instanceID=(*env)->getMethodID(env,class,"instanceMethod","Lava/lang/String;");
```
通过MethodID调用方法：
```
Jstring result=(*env)->CallStringMethod(env,instance, instanceID);
```
静态方法：
获取MethodID：
```
jMethodID staticID=(*env)->getStaticMethodID(env,clazz,"staticMethod","Ljava/lang/String;");
```
通过MethodID调用静态方法
```
Jstring result=(*env)->CallStaticStringMethod(env,clazz, staticID);
```

4.3 域和方法的描述符

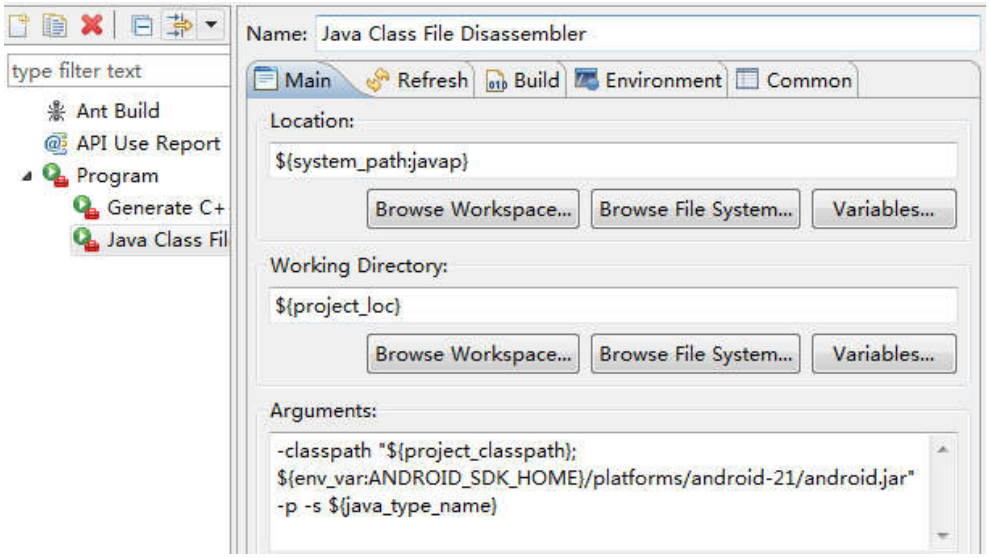
上面两节获取FieldID和MethodID的时候最有一个参数为描述符，分别表示变量的类型和方法的返回值。具体的类型和对应描述符如下：

Java Type	Signature
Boolean	Z
Byte	B
Char	C
Short	S
Int	I
Long	J
Float	F
Double	D
fully-qualified-class	Lfully-qualified-class;
type[]	[type
method type	(arg-type)ret-type

可以通过javap命令自动的生成类型描述符，命令行方法格式如下：

```
javap -classpath bin/classes -p -s com.example.hellojni.HelloJni
```

也可以在eclipse中定义tool用于调用（和javah类似）：



5，异常处理

- 捕获异常

```
Jthrowable ex=(*env)->ExceptionOccurred(env);//获取异常
If(0!=ex){ //判断如果有异常
    (*env)->ExceptionClear(env); //清除异常状态
    //异常处理代码
}
```

- 抛出异常

由于需要使用java中的异常类，所以需要先通过FindClass方法获取异常类，然后抛出：

```
Jclass clazz=(*env)->FindClass(env,"java/lang/NullPointerException"); //获取异常类
If(0!=clazz){
    (*env)->ThrowNew(env,clazz,"异常提示信息" ); //抛出异常
}
```

6，多线程

需要注意以下两点：

- 本地引用无法在多线程中共享，只有全局变量共享可以在多线程中使用。

全局变量：

```
jclass localClazz;
jclass globalClazz;
...
localClazz = (*env)->FindClass(env, "java/lang/String");
globalClazz = (*env)->NewGlobalRef(env, localClazz);
...
(*env)->DeleteLocalRef(env, localClazz);

(*env)->DeleteGlobalRef(env, globalClazz);
```

弱型全局变量：会被垃圾回收

```
jclass weakGlobalClazz;
weakGlobalClazz = (*env)->NewWeakGlobalRef(env, localClazz);


(*env)->DeleteWeakGlobalRef(env, weakGlobalClazz);
```

- JNIENV接口只适用于每个本地方法，不能缓存或者被其他线程使用

本地代码如何实现java中同步操作对象？如下面示例：

Java：

```
synchronized(obj) {
    /* Synchronized thread-safe code block. */
}
```

C：

```
if (JNI_OK == (*env)->MonitorEnter(env, obj)) {
    /* Error handling. */
}

/* Synchronized thread-safe code block. */

if (JNI_OK == (*env)->MonitorExit(env, obj)) {
    /* Error handling. */
}
```

本地线程需要连接到Java虚拟机上才能与Java代码进行通信：

```
JavaVM* cachedJvm;
...
JNIEnv* env;
...
/* Attach the current thread to virtual machine. */
(*cachedJvm)->AttachCurrentThread(cachedJvm, &env, NULL);

/* Thread can communicate with the Java application
   using the JNIEnv interface. */

/* Detach the current thread from virtual machine. */
(*cachedJvm)->DetachCurrentThread(cachedJvm);
```

三 多线程与同步

```
Thread thread = new Thread("thread name") {
    @Override
    public void run() {
        // 在这里做事情
    }
};
thread.setPriority(Thread.MAX_PRIORITY); // 最高优先级 (比 UI 线程高)
thread.start();
```

如果未指定优先级，会使用默认值。Thread 类定义了三个常量：

- MIN_PRIORITY (1)
- NORM_PRIORITY (5)—— 默认优先级
- MAX_PRIORITY (10)

如果应用设置的线程优先级超出取值范围，也就是说，小于 1 或大于 10，那么会抛出 IllegalArgumentException 异常。

1针对多核处理器的优化：

多数情况下，不用操心设备有多少个核心，如果处理器是多核的，这些线程会自动分配到不同的核心上。为了最有效的使用CPU提高性能，需要特别为多核定制算法。

代码清单 5-15　获取处理器核心数量

```
// Galaxy Tab 10.1 或 BeBox Dual603 返回 2，Nexus S 或 Logitech Revue 返回 1
final int proc = Runtime.getRuntime().availableProcessors();
```

Java定义里ExcutorService接口，可以使用这个接口来调度任务：

```
private static final int proc = Runtime.getRuntime().availableProcessors();
private static final ExecutorService executorService =
Executors.newFixedThreadPool(proc + 2);
```

```
public static BigInteger recursiveFasterBigIntegerAndThreading (int n) {
    int proc = Runtime.getRuntime().availableProcessors();
    if (n < 128 || proc <= 1) {
        return recursiveFasterBigInteger(n);
    }

    final int m = (n / 2) + (n & 1);
    Callable<BigInteger> callable = new Callable<BigInteger>() {
        public BigInteger call() throws Exception {
            return recursiveFasterBigInteger(m);
        }
    };
    Future<BigInteger> ffM = executorService.submit(callable); // 尽早提交第一个任务

    callable = new Callable<BigInteger>() {
        public BigInteger call() throws Exception {
            return recursiveFasterBigInteger(m-1);
        }
    };
    Future<BigInteger> ffM_1 = executorService.submit(callable); // 提交第二个任务

// 得到各部分结果，进行合并
BigInteger fM, fM_1, fN;

try {
    fM = ffM.get(); // 获取第一个子问题的结果（阻塞调用）
} catch (Exception e) {
    // 如果有异常，在当前线程计算 fM
    fM = recursiveFasterBigInteger(m);
}
try {
    fM_1 = ffM_1.get(); // 获取第二个子问题的结果（阻塞调用）
} catch (Exception e) {
    // 如果有异常，在当前线程计算 fM
    fM_1 = recursiveFasterBigInteger(m-1);
}
```

即使把问题分解为子问题并将子问题分派到不同线程，可能性能也不一定有提升。有可能是数据之间有依赖，不得不进行同步，线程可能会花费一些或大部分时间在等待数据。此外，性能提升也不一定如你所愿。虽然相对于单核心，理论上期望双核心处理器性能是两倍，四核处理器是四倍，但现实给出的是另一番景象。

四 性能评估

1，时间度量

Java和android提供了一下API，用于测量时间和性能：

- System.currentTimeMillis
- System.nanoTime
- Debug.threadCpuTimeNanos
- SystemClock.currentThreadTimeMillis
- SystemClock.elapsedRealtime
- SystemClock.uptimeMillis

注意 即使有些方法返回时间用纳秒表示，但这并不意味着精度是纳秒级的。实际精度取决于平台，设备之间可能会有所不同。同样，System.currentTimeMillis()返回的毫秒数也不保证毫秒的精度。

尽管 System.currentTimeMillis()可以作为测量时间的手段，但不建议使用这种方法，原因如下：

- 其精度和准确度可能不够；
- 更改系统时间会影响结果。

最好使用 System.nanoTime()，因为它提供了更好的精度和准确度。

代码清单 6-1 测量时间

```
long startTime = System.nanoTime();

// 这里是待测量的操作

long duration = System.nanoTime() - startTime;

System.out.println("Duration: " + duration);
```

Android提供的Debug.threadCpuTimeNanos()方法值测量当前线程中所花费的时间，比system更准确。

代码清单 6-3 使用 Debug.threadCpuTimeNanos()

```
long startTime = Debug.threadCpuTimeNanos();
// 警告：这可能会返回-1，如果系统不支持此操作

// 暂停 1 秒钟（其他线程会在这段时间内调度运行）
try {
    TimeUnit.SECONDS.sleep(1);
    // 和 Thread.sleep(1000)是一样的；
} catch (InterruptedException e) {
    e.printStackTrace();
}

long duration = Debug.threadCpuTimeNanos() - startTime;

Log.i(TAG, "Duration: " + duration + " nanoseconds");
```

2，方法调用跟踪

Android 提供了 Debug.startMethodTracing()方法来创建跟踪文件，然后用 Traceview 工具调试和分析应用。Debug.startMethodTracing()方法有 4 个变种：

- startMethodTracing()
- startMethodTracing(String traceName)
- startMethodTracing(String traceName, int bufferSize)
- startMethodTracing(String traceName, int bufferSize, int flags)

代码清单 6-5 启用跟踪

```
Debug.startMethodTracing("/sdcard/awesometrace.trace");

// 需要跟踪的操作
BigInteger fN = Fibonacci.computeRecursivelyWithCache(100000);

Debug.stopMethodTracing();
```

获取跟踪文件后，用TraceView打开，包含以下信息：

- Name： 方法名。
- Incl %： 此方法中占的时间百分比（包含子方法）。
- Inclusive： 此方法所花毫秒数（包含子方法）。
- Excl %： 此方法所占时间百分比（不包含子方法）。
- Exclusive： 此方法所花毫秒数（不包含子方法）。
- Calls+RecurCalls/Total： 调用和递归调用次数 。
- Time/Call： 平均每次调用时间。

五 延长电池续航

1，广播接收器开关控制

定义一个标记变量，在应用处于前台运行时才启动广播：

```
private void enableBatteryReceiver(boolean enabled) {
    PackageManager pm = getPackageManager();
    ComponentName receiverName = new ComponentName(this, BatteryReceiver.class);
    int newState;

    if (enabled) {
        newState = PackageManager.COMPONENT_ENABLED_STATE_ENABLED;
    } else {
        newState = PackageManager.COMPONENT_ENABLED_STATE_DISABLED;
    }

    pm.setComponentEnabledSetting(receiverName, newState, PackageManager.DONT_KILL_APP);
}

@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(mBatteryChangedReceiver);

    enableBatteryReceiver(false); // 电池接收器现在被禁用

    // 应用没有在前台时注销接收器以省功耗
}

@Override
protected void onResume() {
    super.onResume();
    if (mBatteryChangedReceiver == null) {
        createBatteryReceiver();
    }
    registerReceiver(mBatteryChangedReceiver, new
        IntentFilter(Intent.ACTION_BATTERY_CHANGED));

    enableBatteryReceiver(true);  // 电池接收器现在已启用
}
```

2，WakeLock屏幕长亮控制

控制屏幕长亮可以使用android:keepscreenon，不需要配置权限和释放资源。使用下面的WakeLock就需要这些步骤了。

在某些情况下，一些应用程序即使用户长时间不与设备交互，也要阻止设备进入休眠状态，来保持良好的用户体验。最简单的例子（也是最可能遇到的），就是当用户观看设备上的视频或电影时。这种情况下，CPU 需要做视频解码，同时屏幕保持开启，让用户能够观看。此外，视频播放时屏幕不能变暗。

Android 为这种场景设计了 WakeLock 类，如代码清单 7-19 所示。

代码清单 7-19　创建 WakeLock

```
private void runInWakeLock(Runnable runnable, int flags) {
    PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);

    PowerManager.WakeLock wl = pm.newWakeLock(flags, "My WakeLock");

    wl.acquire();

    runnable.run();

    wl.release();
}
```

注意　应用需要 WAKE_LOCK 的权限来使用 WakeLock 对象。

系统的行为取决于创建 WakeLock 对象时传入的标记（flags）参数。Android 的定义了以下标记：

- PARTIAL_WAKE_LOCK（CPU 开）
- SCREEN_DIM_WAKE_LOCK（CPU 开、暗色显示）
- SCREEN_BRIGHT_WAKE_LOCK（CPU 开、明亮显示）
- FULL_WAKE_LOCK（CPU 开、明亮显示、键盘开）

这些标记可以结合使用：

- ACQUIRE_CAUSES_WAKEUP（打开屏幕和键盘）；
- ON_AFTER_RELEASE（WakeLock 释放后继续保持屏幕和键盘开启片刻）。

六 界面布局优化

1，基本优化方法

1. 建议使用RelativeLayout，减少linearLayout布局的使用
2. 使用<include>标签重用布局

代码清单 8-7　多次包含布局

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <include android:id="@+id/myid1" android:layout="@layout/mylayout" android:layout_margin="9dip" />
    <include android:id="@+id/myid2" android:layout="@layout/mylayout" android:layout_margin="3dip" />
    <include android:id="@+id/myid3" android:layout="@layout/mylayout" />

</LinearLayout>
```

3. }ViewStub推迟初始化

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

  <ViewStub
      android:id="@+id/mystubid"
      android:inflatedId=" @+id/myid"
      android:layout="@layout/mylayout" />
```

代码清单 8-10　在代码中展开布局

```
ViewStub stub = (ViewStub) findViewById(R.id.mystubid);
View inflatedView = stub.inflate();// inflatedView 定义在 mylayout.xml 的布局中
```

代码清单 8-11　在代码中用 setVisibility()展开布局

```
View view = findViewById(R.id.mystubid);
view.setVisibility(View.VISIBLE); // ViewStub 取代展开的布局
view = findViewById(R.id.myid); // 现在要得到展开的视图
```

2，布局调试

Layoutopt 是一个优化布局的工具，它可以帮助开发者分析采用的布局是否合理，并给出修改意见。其用法是：

```
layoutopt <directories/files to analyze>
```

具体方法如下：

```
layoutopt res/layout-land
layoutopt res/layout/main.xml
```

Layoutopt可以指出有问题的代码所在的位置和问题原因，还可以给出优化建议。

Hierarchyviewer 允许开发者调试和优化用户界面。通过 Hierarchyviewer 开发者可以清晰地看到当前设备的 UI 界面的实际布局和控件属性，这在复杂界面的调试中显得非常有用，可以帮助开发者快速定位问题。当然出于安全性考虑，Hierarchyviewer 仅能优化 debug 模式

在 Hierarchyviewer 的视图界面中，开发者可以清晰地看到布局文件的层次关系。针对特定的 UI 控件，开发者可以清晰地看到控件在屏幕上的位置以及各属性的值。

七，android测试

1，Monkey压力测试

Monkey工具可以模拟各种按键、触屏、轨迹球、导航、activity等事件，基本用法如下。Monkey测试仅能模拟系统事件来检测程序中的语法bug，对于深层次的语义bug和网络功能都无法进行有效的测试。

```
adb shell monkey [options] <event-count>// 特定事件
adb shell monkey -p your.package.name -v 50000//50000 随机事件
```

设置Intent-filter：

```
<activity android:name="Hello">
    <intent-filter>
     <action android:name="android.intent.action.MAIN"/>
    <category android:name
="android.intent.category.MONKEY"/>
    </intent-filter>
  </activity>
```

在进行 Monkey 压力测试的过程中，Monkey 会因为应用崩溃、网络超时及一些无法处理的异常而停止测试。建议在对网络应用进行压力测试时，忽略网络超时的情况，方法如下：

```
#adb shell monkey -p your.package.name -v 50000 --ignore-timeouts
```

如果希望忽略应用崩溃的情况，那么可执行如下方法：

```
#adb shell monkey -p your.package.name -v 50000 --ignore-crashes
```

2，Iunit回归测试

根据约束的不同，测试可以分为 SmallTest、MediumTest、LargeTest、AndroidOnly、SideEffect、UiThreadTest、BrokenTest、Smoke、Suppress 等。其中 AndroidOnly、表示测试项仅适用于 Android；SideEffect 表示测试项具有副作用；UiThreadTest 表示测试项在 UI 主线程中运行；BrokenTest 表示测试项需要修复；Smoke 表示测试项为冒烟测试；Suppress 表示该测试项不应出现在测试用例中。其他测试类型的具体约束如表 12-1 所示。

表 12-1　测试类型的约束

功　能	SmallTest	MediumTest	LargeTest
网络接入	No	仅 localhost	Yes
数据库	No	Yes	Yes
文件系统接入	No	Yes	Yes
使用外部系统	No	Discouraged	Yes
多线程	No	Yes	Yes
休眠	No	Yes	Yes
系统属性	No	Yes	Yes
时间约束	60	300	900+

1，JUnit实现

1，构建manifest配置文件

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.calculator2.tests">
    <application>
        <uses-library android:name="android.test.runner" />
    </application>
// 仅在源代码下可用
    <instrumentation android:name="CalculatorLaunchPerformance"
        android:targetPackage="com.android.calculator2"
```

2，上面使用的是默认的runner文件。InstrumentationTestRunner是Junit测试的入口文件，在某些情况下可以自定义实现，其中最重要的是getAllTests方法。

```
public class MusicPlayerFunctionalTestRunner extends InstrumentationTestRunner {
    @Override
    public TestSuite getAllTests() {
        TestSuite suite = new InstrumentationTestSuite(this);
        suite.addTestSuite(TestSongs.class);
        suite.addTestSuite(TestPlaylist.class);
        suite.addTestSuite(MusicPlayerStability.class);
        return suite;
    }
    @Override
    public ClassLoader getLoader() {
        return MusicPlayerFunctionalTestRunner.class.getClassLoader();
    }
}
```

3，具体的测试代码