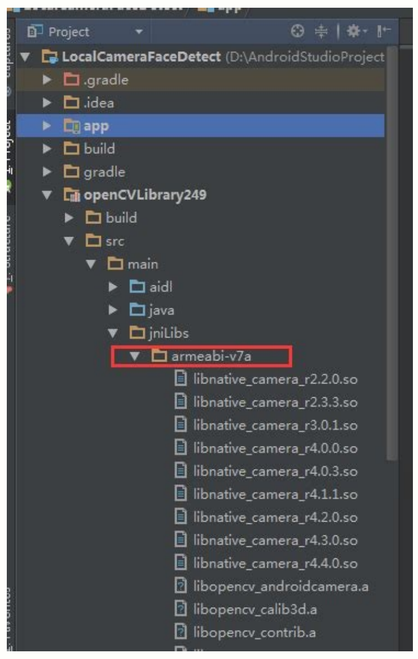
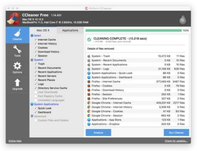
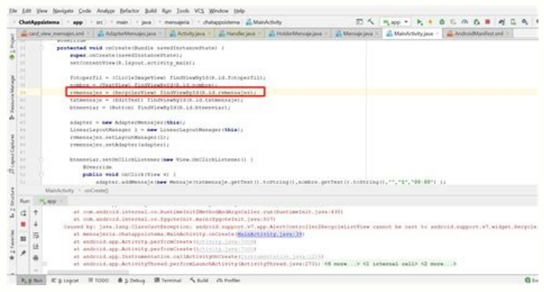
Java lang unsatisfiedlinkerror android studio

Continue







```
package com.chsoft.jnitest;

import android.app.Activity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.util.Log;

public class JniTestActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        String mes = "hello";//stringFromJNICPP();
        new AlertDialog.Builder(JniTestActivity.this).setMessage(mes).show();

    }

    public native String  stringFromJNICPP();

    static {
        try{
            System.loadLibrary("JniTest");
        }catch(Exception ex){
            Log.println(1, "Error", ex.getMessage());
        }
    }
}
```

Java.lang.unsatisfiedlinkerror android studio. Android studio java.lang.unsatisfiedlinkerror unable to load library 'corefoundation'. Android studio java.lang.unsatisfiedlinkerror dlopen failed. Android studio java.lang.unsatisfiedlinkerror dlopen failed library. Java.lang.unsatisfiedlinkerror no implementation found android studio. Android studio java.lang.unsatisfiedlinkerror dalvik.system.pathclassloader.

JNI is a native Java interface. Defines how bytecode compiled by Android from managed code (written in Java or Kotlin programming languages) interacts with native code (written in C/C++). JNI is vendor agnostic, supports code loading from dynamic shared libraries, and while unwieldy, is quite efficient at times. Note. Since Android compiles Kotlin to ART-friendly bytecode in a manner similar to the Java programming language, you can apply the recommendations on this page to both Kotlin and Java programming languages in terms of JNI architecture and related costs. To learn more, see Kotlin and Android. If you're not familiar with it yet, read the Java Native Interface Specification to understand how JNI works and what features are available. Some aspects of the interface are not immediately apparent on first reading, so the next few sections may be helpful. To view global JNI references and see where global JNI references are created and deleted, use the JNI Heap View in the Memory Profiler in Android Studio 3.2 and later. General tips Try to minimize the area of your JNI layer. There are several aspects to consider here. Your JNI solution should try to follow the following guidelines (listed below in order of importance, starting with the most important): Minimize resource sorting at the JNI level. Sorting by the JNI layer is associated with a non-trivial cost. Try to design an interface that minimizes the amount of data you need to order and how often you need to order it. If possible, avoid asynchronous communication between code written in a managed programming language and code written in C++. This will make it easier to maintain the JNI interface. You can usually simplify asynchronous UI updates by keeping the asynchronous update in the same language as the UI. For example, instead of calling a C++ function from the UI thread in Java code via JNI, it's better to do thatbetween two Java programming language threads, one of which makes a C++ blocking call and notifies the UI thread when the blocking call is complete. Reduce the number of threads that need or need to touch JNI. If you must use thread pools in both Java and C++, try to keep JNI communication between pool owners and not between individual worker threads. Keep your interface code in a small number of easily identifiable C++ and Java source locations to facilitate future conversion processes. If needed, consider using the JNI library for auto generation. JavaVM and JNIEnv JNI defines two main data structures "JavaVM" and "JNIEnv". Both are essentially pointers to pointers to function tables. (In C++, these are classes with a pointer to the function table and a member function for each JNI function that implicitly routes through the table.) JavaVM provides "calling interface" functions that you can use to create and destroy JavaVMs. In theory, a process can have multiple JavaVMs, but Android only allows one. JNIEnv provides most of the JNI functionality. All of your native functions take JNIEnv as the first argument. JNIEnv uses thread-local storage. Because of this, you cannot share JNIEnv between threads. If the code snippet has no other way to get its JNIEnv, you should free the JavaVM and use GetEnv to expose the JNIEnv thread. (Assuming it exists; see AttachCurrentThread below.) JNIEnv and JavaVM C declarations differ from C++ declarations. The attached jni.h file yields different type deficits depending on whether it's in C or C++. Because of this, it's not a good idea to include JNIEnv arguments in the header files that come with both languages. (Another possibility: if your header file requires #ifdef __cplusplus, you may have to do extra work if anything in that header refers to JNIEnv.) Threads All threads are Linux kernel-scheduled threads. They arebe started from managed code (using Thread.start()), but can also be created elsewhere and then attached to JavaVM. For example, a thread started with pthread_create() or std::thread can be attached with AttachCurrentThread() or AttachCurrentThreadAsDaemon(). Until a thread is connected, it does not have a JNIEnv and cannot make JNI calls. It's usually best to use Thread.start() to create any thread that needs to call Java code. This ensures that you have enough stack space, that you are in the correct thread group, and that you are using the same ClassLoader as your Java code. It's also easier to set a debug thread name in Java than from native code (see pthread_setname_np() if you have pthread_t or thread_t, and std::thread::native_handle() if you have std::thread and need pthread_t). Attaching the originally created thread creates a java.lang.Thread object and adds it to the "main" thread group, making it visible to the debugger. Calling AttachCurrentThread() on an already attached thread does not work. Android does not suspend threads that execute native code. If garbage collection is in progress or the debugger has sent a suspend request, Android suspends the thread on the next JNI call. Threads attached to JNI must call DetachCurrentThread() before exiting. If it's inconvenient to code directly, you can use pthread_key_create() in Android 2.0 (Eclair) and later to define a destructor function to be called before the thread terminates, and call DetachCurrentThread() from there. (Use this key with pthread_setspecific() to store the JNIEnv in thread-local storage; that way it will be passed to your destructor as an argument.) jclass, jmethodID, and jfieldID If you want to access an object field from native code, follow these steps: Get a reference to the object classes using FindClass. Get the field ID using GetFieldID. Get the contents of an array usingB. GetIntField. To invoke a method, you must first get a reference to the class object, and then get the ID of the method. Identifiers are often just pointers to internal runtime data structures. It may take some string comparisons to find them, but once you get them, the actual field call or method call is very fast. When performance matters, it makes sense to check the values once and cache the results in the native code. Since there is only one JavaVM per process, it makes sense to store this data in a static local structure. Class references, field IDs, and method IDs are guaranteed until the class is released. Classes are only removed if all classes related to the ClassLoader can be garbage collected, which is rare but not impossible on Android. Note, however, that jclass is a class reference and must be protected by calling NewGlobalRef (see the next section). If you want to cache IDs when you load a class and automatically re-cache them when the class is deallocated and reloaded, the correct way to initialize IDs is to add a code snippet that looks like this to the appropriate class :companion object { / * * We use the static initializer class, which allows native code to cache some field offsets *. This native function finds and stores interesting* class/field/method IDs. roles in failure. */ private external fun nativeInit() init { nativeInit() } } /* * We use a class initializer that allows the native code to cache some * field offsets. This native function finds and stores interesting* class/field/method IDs. roles in failure. */ private static native void nativeInit(); static { nativeInit(); } In C/C++ code, create a nativeClassInit method that performs the ID lookup. The code runs once during class initialization. If the class is ever unloaded and then reloaded, it will run again. Local and global references Any argument passed to nativeand almost every object returned by a JNI function is a "local reference". That is, it is in effect for the duration of the current thread's current local method. Even though the object itself still works after the original method is returned, the reference is invalid. This applies to all jobect subclasses, including jclass, jstring, and jarray. (If extended JNI checks are enabled, the runtime will warn about most reference abuse.) The only way to get non-local references is to use the NewGlobalRef and NewWeakGlobalRef functions. If you want to keep the reference for a long time, you need to use a "global" reference. The NewGlobalRef function takes a local reference as an argument and returns a global reference. The global reference is guaranteed until DeleteGlobalRef is called. This pattern is typically used when caching the jclass returned by FindClass, for example: jclass localClass = env->FindClass("MyClass"); jclass globalClass = reinterpret_cast(env->NewGlobalRef(localClass)); All JNI methods accept both local and global references as arguments. References to the same object can have different values. For example, the values returned from successive calls to NewGlobalRef on the same object may differ. To determine whether two references refer to the same object, use the IsSameObject function. Never compare references with == in native code. One consequence of this is that object references should not be treated as constant or unique in native code. The 32-bit value representing an object may vary depending on the method call, and two different objects may have the same 32-bit value in subsequent calls. Don't use jobject values as keys. Developers must "over-allocate" local references. In practice, this means that if you create a large number of local references, for example when traversing an array of objects, you must free them.with DeleteLocalRef instead of letting JNI do it for you. The implementation is only needed to reserve space for 16 local links. So if you need more, either remove or use ServiateLocalCapacity/PushLocalFrame to reserve more. Note that jfieldID and jmethodID are opaque types, not object references, and should not be passed to NewGlobalRef. The raw data pointers returned by functions such as GetStringUTFChars and GetByteArrayElements are also not objects. (They can be passed between threads and are valid until the corresponding release call.) An unusual case deserves special mention. If you attach a local thread using AttachCurrentThread, your running code will never automatically release local references until the thread is detached. Any local links created must be removed manually. Generally, any native code that creates local references in a loop may need to be removed manually. Be careful when using global links. Global references may be unavoidable, but they are difficult to debug and can lead to hard-to-diagnose memory behavior. Other things being equal, a solution with fewer global pointers is probably better. UTF-8 and UTF-16 strings The Java programming language uses UTF-16. For convenience, JNI provides methods that also work with modified UTF-8 encoding. The modified encoding is useful for C code because it encodes \u0000 as 0xc0 0x80 instead of 0x00. The beauty is that you can rely on C-style null-terminated strings for use with standard libc string functions. The downside is that you can't pass arbitrary UTF-8 data to JNI and expect it to work correctly. It's usually faster to work with UTF-16 strings when possible. Currently Android does not require a copy for GetStringChars and GetStringUTFChars requires collation and conversion to UTF-8. Note that UTF-16 strings are not null-terminated and \u0000 are allowed, so you need to wait for the stringand the jchar pointer. Don't forget to release the received strings. String functions return jchar* or jbyte*, which are C-style pointers to primitive data, not local references. They are guaranteed to be valid until Release is called, meaning they will not be released when the native method returns. Data passed in NewStringUTF must be in modified UTF-8 format. A common mistake is to read character data from a file or network stream and pass it to NewStringUTF without filtering. If you don't know that the data is valid MUTF-8 (or 7-bit ASCII, which is a compatible subset), you need to remove the invalid characters or convert it to the correct modified form of UTF-8 . Otherwise, the UTF-16 conversion may produce unexpected results. CheckJNI - enabled by default for emulators - looks for strings and terminates the virtual machine if it receives malformed input. JNI array primitives provide functions to access the contents of array objects. While object arrays must be accessed input-by-input, primitive arrays can be read and written directly as if they were declared in C. Calls to the ArrayElements family allow the runtime to either return a pointer to the actual elements or allocate some storage and make a copy. In either case, the returned raw pointer is guaranteed to be valid until the appropriate Release call is made (that is, the array object is pinned and cannot be moved as part of array compression unless the data has been copied). Heap). You must release each received field. If the Get call fails, you must ensure that your code does not attempt to deallocate the NULL pointer later. You can determine whether data has been copied by passing a non-zero pointer to the isCopy argument. This is rarely useful. The Release call accepts a mode argument, which can be one of three values. Actions taken by the runtime depend ondid it return a pointer to the actual data or a copy of it: 0 Current: The array object is not pinned. Copy: The data is copied back. The copy buffer is released. JNI_COMMIT True: Does nothing. Copy: The data is copied back. The copy buffer is not freed. JNI_ABORT Current: The array object is not attached. Older recordings are not interrupted. Copy: the copy buffer is released; all its changes will be lost. One reason to check the isCopy flag is to know if you need to call Release with JNI_COMMIT after making changes to the array - if you alternate between making changes and running code that uses the contents of the array, you can skip inactive commits. Another possible reason to check the flag is efficient JNI_ABORT handling. For example, you might want to get an array, modify it in place, pass parts to other functions, and then roll back the changes. If you know that JNI is creating a new copy for you, there is no need to create another "editable" copy. If JNI gives you the original, you must make your own copy. A common mistake (reproduced in the code example) is to assume that you can omit the Release call if *isCopy is false. That's not the point. If the copy buffer has not been allocated, the original memory must be attached and cannot be moved using the memory release utility. Also note that the JNI_COMMIT flag will not release the field and you will have to call Release again with a different flag. Range Calls There is an alternative to calls like GetArrayElements and GetStringChars that can be very useful when all you want to do is copy data. Consider the following: jbyte* data = env->GetByteArrayElements(array, NULL); if (data != NULL) { memcpy(buffer, data, length); env->ReleaseByteArrayElements(array, data, JNI_ABORT); } This will fetch the array, copy the first byte-only elements from it, and then free the array. Depending on the implementation, the call to Get will be appended or copiedtable contents. The code copies the data (perhaps a second time) and then calls Release; in this case, JNI_ABORT guarantees that there is no chance of getting a third copy. The same can be achieved more easily: env->GetByteArrayRegion(array, 0, len, buffer); This has several advantages: one JNI call is required instead of 2, thus reducing overhead. No pinning or additional data copies required. Reduces the risk of developer errors – no risk of forgetting to call a release if something goes wrong. Similarly, you can use SetArrayRegion to copy data into an array, and GetStringRegion or GetStringUTFRegion to copy characters from a string. Exceptions Most JNI functions should not be called while waiting for an exception. Your code is expected to catch an exception (using the ExceptionCheck or ExceptionOccurred function return value) and return or clear and handle the exception. The only JNI functions you can call when expecting an exception are: DeleteGlobalRef DeleteLocalRef DeleteWeakGlobalRef ExceptionCheck ExceptionClear ExceptionDescribe ExceptionOccurred MonitorExit PopLocalFrame PushLocalFrame ReleaseArrayElements ReleaseReleaseReleaseCriticalRelease. For example, if NewString returns a non-NULL value, you don't need to check for an exception. However, calling a method (with a function like CallObjectMethod) should always check for an exception because the return value is invalid if an exception is thrown. Note that exceptions from interpreted code do not clear native stack frames, and Android does not yet support C++ exceptions. The JNI Throw and ThrowNew statements simply set the exception pointer to the current thread. Returning to the managed code, the exception is caught and handled accordingly. Native code can "catch" the file.by calling ExceptionCheck or ExceptionOccurred and clearing it with ExceptionClear. As usual, throwing unhandled exceptions can cause problems. There are no built-in functions to manipulate the Throwable object itself. So if you want to (say) get an exception string you need to find the Throwable class, the method id for getMessage "()Ljava/lang/String ;", call it and if the result is not NULL then use GetStringUTFChars to get something to pass to printf (3) or equivalent. JNI extended validation checks for very few errors. Mistakes usually lead to failure. Android also provides a mode called CheckJNI where JavaVM and JNIEnv function table pointers are switched to function tables that perform an extended set of checks before calling the default implementation. Additional checks include: Arrays: try to allocate a negative sized array. Bad pointers: passing an invalid jarray/jclass/jobject/jstring to a JNI call, or passing a NULL pointer to a JNI call with a non-nullable argument. Class names: Pass any class name except java/lang/String style to the JNI call. Critical Calls: Making a JNI call between a "critical" receipt and the corresponding release. Direct ByteBuffers: Passing invalid arguments to NewDirectByteBuffer. Exceptions: Making a JNI call while waiting for an exception. JNIEnv*s: Using JNIEnv* from the wrong thread. jfieldIDs: using NULL jfieldID or using jfieldID to set a field to a value of the wrong type (e.g. or vice versa, or using jfieldID from one class with instances of another class. jmethodIDs: Wrong type jmethodID used when calling Call*Method JNI: Invalid return type, mismatch static/non-static, wrong type for "this" (for non-static calls) or wrong class (for static calls. Connections). Reference: Using DeleteGlobalRef/DeleteLocalRef ontype of help. Release Modes: Passing an incorrect release mode to the release call (anything other than 0, JNI_ABORT, or JNI_COMMIT). Type Safety: Returning an incompatible type from your own method (for example, returning a StringBuilder from a method declared to return a String). UTF-8: Passing an invalid modified UTF-8 byte sequence to a JNI call. (Methods and fields are not yet checked for availability: access restrictions do not apply to native code.) There are several ways to enable CheckJNI. If you are using an emulator, CheckJNI is enabled by default. If you have a rooted device, you can use the following command sequence to restart the runtime with CheckJNI enabled: adb shell stop adb shell setprop dalvik.vm.checkjni true adb shell start In each of these cases, you will see something like the logcat -Output when running runtime :D AndroidRuntime: CheckJNI enabled If you have a regular device, you can use the following command: adb shell setprop debug.checkjni 1 This will not affect already running applications, but all applications that are started to run CheckJNI integrated. (Change this property to a different value, or simply reload CheckJNI to disable CheckJNI again.) In this case, the next time you run the program, you'll see something like this in the logcat output: D Activating CheckJNI late You can also use android :debuggable. in the application manifest to enable CheckJNI only for your application. Note that the Android build tools do this automatically for certain build types. Native Libraries You can load native code from shared libraries using the System.loadLibrary standard library. In practice, older versions of Android had bugs in PackageManager that made installing and updating native libraries unreliable. The ReLinker project provides solutions to this and other native library loading problems. Call System.loadLibrary (or ReLinker.loadLibrary) from the static class initializer. The argument is the "raw" library name, so load ityou would go to "fubar". If you only have one class with native methods, it makes sense to call System.loadLibrary in the static initializer for that class. Otherwise, you may want to call from the application to be told that the library is always and always loaded first. There are two ways the runtime can find your custom methods. You can either register them explicitly with RegisterNatives or let the runtime find them dynamically with dlsym. The advantage of RegisterNatives is that you can look ahead to characters and have smaller and faster shared libraries without having to export anything other than JNI_OnLoad. The advantage of a runtime that can recognize your functions is that it takes a little less code to write. To use RegisterNatives: Provide JNIEXPORT function jint JNI_OnLoad(JavaVM* vm, void* is reserved). In JNI_OnLoad, register all native methods using RegisterNatives. Build with -fvisibility=hidden so that only your JNI_OnLoad will be exported from your library. This produces faster, smaller code and avoids potential conflicts with other libraries loaded in your application (but produces a less useful stack trace if your application crashes in native code). The static initializer should look like this: Companion object { init { System.loadLibrary("fubar") } } static { System.loadLibrary("fubar"); } The JNI_OnLoad function should look something like this when written in C++: JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved) { JNIEnv* env; if (vm->GetEnv(reinterpret_cast(&env), JNI_VERSION_1_6) != JNI_OK) { return JNI_ERR; } // Find your class. For this to work, JNI_OnLoad is called from the correct classloader context. jclass c = env->FindClass("com/example/app/package/MyClass"); If (c == nullptr) return JNI_ERR; // Register the native methods of your class. Methods Static Constant JNINativeMethod[] = {{"nativeFoo", "()V", reinterpret_cast(nativeFoo)}, {"nativeBar",reinterpret_cast(nativeBar)}, }; int rc = env->RegisterNatives(c, methods, sizeof(methods)/sizeof(JNINativeMethod)); if (rc != JNI_OK) return rc; return JNI_VERSION_1_6; } To use "recognition" of native methods instead, they must be named in a specific way (see the JNI specification for details). This means that if the method signature is incorrect, you won't know about it until the first actual method call. All FindClass calls to JNI_OnLoad resolve the class in the context of the classloader used to load the shared library. When called from other contexts, FindClass uses a classloader associated with a method at the top of the Java stack, or if there isn't one (because the call comes from its own thread, which has just been added), it uses "system". . class loader. The system class loader doesn't know your application's classes, so you can't use FindClass to find your classes in that context. This makes JNI_OnLoad a convenient place to look up and store classes: if you have a valid jclass, you can use it from any attached stream. To support architectures that use 64-bit pointers, use a long field instead of an int field if you store a pointer to a local structure in a Java field. Unsupported Features/Backwards Compatibility All JNI 1.6 features are supported with the following exception: DefineClass is not implemented. Android doesn't use Java bytecodes or class files, so passing binary class data won't work. For backward compatibility with older versions of Android, you may need: Dynamic search for built-in functions Prior to Android 2.0 (Eclair), the "$" character was not correctly converted to "_00024" when searching for method names. To work around this, you need to use explicit registration or move your methods from inner classes. Thread Detachment Prior to Android 2.0 (Eclair) it was not possible to use the pthread_key_create destructor function to avoid "disconnect thread first".Check. (The runtime also uses the pthread key's destructor function, so there will be a race to see who gets called first.) Weak global references Before Android 2.2 (Froyo), weak global references were not implemented. Older versions categorically reject attempts to use them. You can use Android platform version constants to check for support. Prior to Android 4.0 (Ice Cream Sandwich), weak global references could only be passed to NewLocalRef, NewGlobalRef, and DeleteWeakGlobalRef. (The spec strongly encourages programmers to hard-reference weak global references before doing anything with them, so this shouldn't be restrictive at all.) Starting with Android 4.0 (Ice Cream Sandwich), weak global references can be used in the same way as and all others. JNI link. Local References Before Android 4.0 (Ice Cream Sandwich), local references were actually direct references. Ice Cream Sandwich added the direction needed to support better garbage collectors, but this means many JNI bugs are undetectable in older versions. For more information, see Changes to local JNI references in ICS. In Android versions prior to Android 8.0, the number of local links is limited by a version-specific limit. Starting with Android 8.0, Android supports unlimited local connections. Determining the reference type with GetObjectRefType Before Android 4.0 (Ice Cream Sandwich) it was not possible to correctly implement GetObjectRefType due to the use of direct pointers (see above). Instead, we used a heuristic that looks at the weak global variable table, the arguments, the local variable table, and the global variable table, in that order. When it first encounters your direct pointer, it will report that your reference is of the type it is examining. This meant, for example, that if you called GetObjectRefType on a global jclass that happened to be identical to the jclass passed as an implicit argument to your static native method, you would get a JNILocalRefType and not a JNIGlobalRefType. When working with native codeit is not uncommon to see the following failure: java.lang.UnsatisfiedLinkError: Library foo not found In some cases it means what it says: library not found. In other cases, the library exists but cannot be opened with dlopen(3) and the error can be found in the detailed exception message. Common reasons for "library not found" exceptions: The library does not exist or is not available to the application. Use adb shell ls -l to check its existence and permissions. The library is not built with NDK. This may cause dependencies on functions or libraries that are not on the device. Another error class UnsatisfiedLinkError looks like this: java.lang.UnsatisfiedLinkError: myfunc at Foo.myfunc(Native Method) at Foo.main(Foo.java:10) In the logcat you will see: W/dalvikvm(880): No implementation . found for its LFoo;.myfunc ()V This means that the runtime tried to find a match method but failed. Some common reasons are: Library not loading. Check the logcat output for library loading messages. Method not found due to name or signature mismatch. This is usually due to the following reasons: It is not possible to declare C++ functions with external "C" and appropriate visibility (JNIEXPORT) for lazy method lookups. Note that before Ice Cream Sandwich the JNIEXPORT macro was incorrect, so using the new GCC with the old jni.h will not work. You can use arm-eabi-nm to display the symbols as they appear in the library; if they look wrong (something like _Z15Java_Foo_myfuncP7_JNIEnvP7_jclass instead of Java_Foo_myfunc) or if the character type is lowercase "t" instead of uppercase "T", you need to fix the declaration. With a clear registration of minor errors when entering the method signature. Make sure what you pass in the log call matches the signature in the log file. Note that "B" is a byte and "Z" is a boolean value. The class name components in the signature start with "L" and end with ";","/" to separate package/class names and use "$" to separate internal class names (say Ljava/util/Map$Entry;). Using javah to automatically generate JNI headers can help avoid some problems. FAQ: Why can't FindClass find my class? (Most of these tips apply equally to errors when looking up methods with GetMethodID or GetStaticMethodID or fields with GetFieldID or GetStaticFieldID.) Make sure the class name string is in the correct format. JNI class names start with the package name and are separated by a slash, for example java/lang/String. If you're looking for an array class, you need to start with the correct number of square brackets and wrap the class with "L" and ";" so that a one-dimensional String array looks like this: [Ljava /language/ string;. If you are looking for an inner class, use "$" instead of ".". In general, using javapu in a .class file is a good way to find out the internal name of your class. If you enable code compression, make sure you set up code storage. Setting appropriate retention rules is important because Code Shrink may otherwise remove classes, methods, or fields that are only used with JNI. If the class name looks correct, you may have a problem with the classloader. FindClass wants to start looking for classes in the classloader that are related to your code. It will inspect the call stack, which will look something like this: Foo.myfunc(native method) Foo.main(Foo.java:10) The topmost method is Foo.myfunc. FindClass finds the ClassLoader object associated with class Foo and uses it. It usually does what you want. You may run into problems if you create the thread yourself (perhaps by calling pthread_create and then attaching it with AttachCurrentThread). There are no stack images in your application now. If you call FindClass from this thread, JavaVM will start in the "system" classloader, not the one associated with your application, so attempts to find application-specific classes will fail. There are several ways to do thisSomething like this: search for FindClass once in JNI_OnLoad and save class references for later use. All FindClass calls made as part of a JNI_OnLoad execution use the class loader associated with the function that called System.loadLibrary (this is a special rule that makes library initialization easier). When your application code loads the library, FindClass will use the correct class loader. Pass the class instance to the functions that need it by declaring a custom method to get the class argument and then passing it to Foo.class. Cache a reference to the ClassLoader object in an appropriate location and call loadClass directly. It takes some effort. You may find yourself in a situation where you need to access a large buffer of raw data from both managed and native code. Common examples are the manipulation of bitmaps or sound samples. There are two main approaches. You can store data in byte[]. This provides very fast access from managed code. However, there is no guarantee on your own side that you will be able to access the data without having to copy it. In some implementations, GetByteArrayElements and GetPrimitiveArrayCritical return actual pointers to the raw data on the managed heap, but in others, they allocate a buffer on their own heap and copy the data. An alternative is to store the data in a direct byte buffer. They can be created using java.nio.ByteBuffer.allocateDirect or the JNI NewDirectByteBuffer function. Unlike regular byte buffers, memory is not allocated on the managed heap and is always available directly from native code (get the address using GetDirectBufferAddress). Depending on how direct byte buffer access is implemented, data access from managed code can be very slow. Which one to use depends on two factors: Will most of the data be accessed through code written in Java or C/C++? Finally, when data is passed to the system API, what form should it be in?For example, if the data is passed to a function that takes a byte [, the processing may not be reasonable.) If there is no clear winner, use an even byte buffer. Support for these is built right into JNI and should improve performance in future releases. graduation.