□□□□□□ को □□□□□□□ पर पोर्ट करना

मूल लेख का लिंक (□□□□)

प्रारंभ

जो छात्र अभी तक □□□□□□ के बारे में नहीं जानते हैं, वे पहले □□□□□□ के सामान्य बुनियादी कमांड्स को देख सकते हैं, और इसकी कार्यक्षमता का गहराई से अध्ययन कर सकते हैं—जैसे कि ऑडियो और वीडियो संश्लेषण, विभिन्न कोडेक्स का प्लेबैक, वीडियो कटिंग, कई छवियों से वीडियो बनाना और ऑडियो मिक्सिंग, फॉर्मेट परिवर्तन आदि। यह कई एप्लिकेशन परिदृश्यों में शक्तिशाली और मजेदार कार्यक्षमता प्रदान कर सकता है।

“डबिंग शो” जैसे ऐप्स के उदाहरण के रूप में, डबिंग न केवल मनोरंजक है, बल्कि अक्सर उत्पाद को और अधिक आकर्षक बना देता है। हमने एक डबिंग मॉड्यूल विकसित करने की योजना बनाई, और इसके लिए □□□□□□ को चुना, इसे □□□□□□□ प्लेटफॉर्म पर पोर्ट करने का प्रयास किया। इसमें लगभग 3-4 दिन लगे, कई संस्करणों का प्रयास किया, और ऑनलाइन कई लेखों में सफलता नहीं मिली, जब तक कि हमें “□□□□□□□ में □□□□□□ का उपयोग और इंटरफेस कॉल” नामक एक ट्यूटोरियल नहीं मिला, जिसके बाद हम इसे सफलतापूर्वक पूरा कर पाए। वास्तविक परीक्षण से पता चला कि केवल □□□□□□ 1.2 और □□□-□9 का संयोजन ही सफलतापूर्वक पोर्ट किया जा सकता है।

विचार

□□□□□□ एक □ भाषा में लिखा गया प्रोजेक्ट है, जिसमें एक `main()` फ़ंक्शन शामिल है। हमारा लक्ष्य है:

1. □□□□□□□ □□□ का उपयोग करके `libffmpeg.so` को कंपाइल करें।
2. इस लाइब्रेरी का उपयोग करके `ffmpeg.c` फ़ाइल को कंपाइल करें और इसमें संशोधन करें, मूल `main()` फ़ंक्शन का नाम बदलकर `video_merge(int argc, char **argv)` कर दें, ताकि □□□ से सीधे इस फ़ंक्शन को कॉल करके वीडियो संयोजन जैसे ऑपरेशन किए जा सकें।

उदाहरण के लिए, वीडियो संश्लेषण को निम्नलिखित तरीके से कार्यान्वित किया जा सकता है (जो कमांड लाइन `ffmpeg -i src1 -i src2 -y output` के अनुरूप है):

```
video_merge(5, argv); //    argv
```

पर्यावरण

- ऑपरेटिंग सिस्टम: □□□□□□ 12.04
- □□□□□□ संस्करण: 1.2
- □□□ संस्करण: □□□-□9

शुरू करने से पहले, कुछ संबंधित ट्यूटोरियल देखने की सलाह दी जाती है। यदि आपको कोई समस्या आती है, तो इस लेख को वापस देखें, ताकि आप बहुत अधिक गलत रास्ते पर न जाएं।

इंटरफ़ेस और □□□□□□□.□□ को संशोधित करना

□□□□□□ के लिए □□□ इंटरफ़ेस लिखते समय, `Android.mk` फ़ाइल को लिखने की आवश्यकता होती है ताकि लाइब्रेरी को लिंक किया जा सके और उपयोग करने योग्य `.so` फ़ाइल उत्पन्न की जा सके। कुछ उदाहरण `Android.mk` फ़ाइलें अलग-अलग वातावरण में सीधे चलाने में सफल नहीं हो सकती हैं। इसका उद्देश्य □□□ को यह बताना है कि कौन से स्रोत फ़ाइलों को संकलित करने की आवश्यकता है, किस लाइब्रेरी से लिंक करना है, आदि।

मैंने एक "दो बार संकलन, फिर लिंक" की विधि अपनाई है:

1. पहले `myffmpeg` नामक एक साझा लाइब्रेरी को संकलित करें।
2. फिर एक अन्य मॉड्यूल `ffmpeg-jni` में, `myffmpeg` को लिंक करें और अंत में आवश्यक `.so` फ़ाइल उत्पन्न करें।

इसके अलावा, कंपाइल किए गए `libffmpeg.so` को `jni` डायरेक्टरी में रखना आवश्यक है, ताकि लिंक करते समय इसे ढूंढा जा सके।

□□□□□□ को डीबग करना

□□□□□□ को पोर्ट करने के बाद, □□□ के माध्यम से फ़ंक्शन को कॉल करने के लिए, अक्सर □ लेयर में डीबग करने की आवश्यकता होती है। यदि हम कमांड लाइन की तरह विस्तृत लॉग आउटपुट देख सकें, तो समस्याओं का पता लगाना और भी आसान हो जाएगा।

□□□□□□□□ में, □□□□ दबाकर `av_log` जैसे कॉल स्थान पर क्लिक करने से, आप `ffmpeg/libavutil/log.c` में `av_log_default_callback` फ़ंक्शन के कार्यान्वयन को ट्रैक कर सकते हैं। यह □□□□□□□□ के `__android_log_print` को कॉल करके □□□□□□ में प्रिंट करता है। इन आउटपुट को देखकर, आप □□□□□□ के आंतरिक स्थिति को जान सकते हैं, जिसका उपयोग संश्लेषण विफलता, विशिष्ट कोडेक का समर्थन न करने जैसी समस्याओं को हल करने के लिए किया जा सकता है।

कभी-कभी, □□□□□□ अपवाद (□□□□□□□□□□) फेंक सकता है जिससे ऐप क्रैश हो सकता है। इस समस्या का पता लगाने के लिए आप निम्नलिखित कमांड का उपयोग कर सकते हैं:

```
adb shell logcat | ndk-stack -sym obj/local/armeabi
```

(यह कोड ब्लॉक है और इसे अनुवादित नहीं किया जाना चाहिए।)

यदि □□□□□□ के मूल `main()` फ़ंक्शन के अंत में `exit(0)` है, तो कृपया इसे कमेंट कर दें, अन्यथा यह एप्लिकेशन को बंद कर देगा।

मेमोरी लीक और सर्विस समाधान

मेमोरी लीक एक सामान्य समस्या है जो सॉफ्टवेयर डेवलपमेंट में होती है, खासकर जब हम लंबे समय तक चलने वाले सर्विसेज (□□□-□□□□□) का उपयोग करते हैं। यह समस्या तब होती है जब प्रोग्राम द्वारा आवंटित मेमोरी को सही ढंग से रिलीज़ नहीं किया जाता है, जिससे समय के साथ मेमोरी की खपत बढ़ती जाती है और अंततः सिस्टम की परफॉर्मेंस प्रभावित होती है।

मेमोरी लीक के कारण

1. **रिसोर्सेज का सही ढंग से रिलीज़ न होना**: जब ऑब्जेक्ट्स या रिसोर्सेज को सही ढंग से डिस्पोज़ या रिलीज़ नहीं किया जाता है, तो वे मेमोरी में जमा होते रहते हैं।
2. **सर्विसेज का लंबे समय तक चलना**: सर्विसेज जो लंबे समय तक चलती हैं, वे अक्सर मेमोरी लीक का कारण बनती हैं, खासकर यदि उनमें रिसोर्सेज का प्रबंधन सही ढंग से नहीं किया गया हो।
3. **इवेंट लिसनर्स का सही ढंग से न हटना**: यदि इवेंट लिसनर्स को सही ढंग से नहीं हटाया जाता है, तो वे मेमोरी लीक का कारण बन सकते हैं।

समाधान

1. **सर्विसेज का सही प्रबंधन**: सर्विसेज को सही ढंग से शुरू और बंद करना महत्वपूर्ण है। यह सुनिश्चित करें कि सर्विसेज द्वारा उपयोग की जाने वाली सभी रिसोर्सेज को सही ढंग से रिलीज़ किया जाए।

```
public class MyService extends Service {
    @Override
    public void onDestroy() {
        super.onDestroy();
        //
    }
}
```

2. **इवेंट लिसनर्स का सही प्रबंधन**: इवेंट लिसनर्स को सही ढंग से हटाना सुनिश्चित करें, खासकर जब वे अब आवश्यक न हों।

```
public class MyActivity extends AppCompatActivity {
    private MyListener listener;
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();
        listener = null; //
    }
}
```

3. **मेमोरी प्रोफाइलिंग टूल्स का उपयोग**: □□□□□□□□ □□□□□□□ जैसे टूल्स का उपयोग करके मेमोरी लीक की पहचान करें और उन्हें ठीक करें।

   ```
   adb shell dumpsys meminfo <package_name>
   ```

4. □□□□□□□□□□□□□□□ **का उपयोग**: यदि आपको किसी ऑब्जेक्ट को लंबे समय तक रखने की आवश्यकता है, तो `WeakReference` का उपयोग करें ताकि गार्बेज कलेक्टर उसे मेमोरी से हटा सके।

   ```
   WeakReference<MyObject> weakRef = new WeakReference<>(myObject);
   ```

मेमोरी लीक को समझना और उन्हें ठीक करना एक महत्वपूर्ण कौशल है जो आपके एप्लिकेशन की परफॉर्मेंस और उपयोगकर्ता अनुभव को बेहतर बना सकता है। सही प्रबंधन और टूल्स का उपयोग करके आप मेमोरी लीक से बच सकते हैं और अपने एप्लिकेशन को अधिक कुशल बना सकते हैं।

संश्लेषण पूरा होने के बाद, यदि फिर से कॉल करने पर `"INVALID HEAP ADDRESS IN dlfree ffmpeg"` त्रुटि आती है, तो यह अधिकतर □□□□□□ के मेमोरी को पूरी तरह से मुक्त न करने के कारण होता है। एक समझौता समाधान यह है कि संश्लेषण प्रक्रिया को एक अलग `Service` में रखा जाए, और संश्लेषण पूरा होने के बाद उस □□□□□□□□ को समाप्त कर दिया जाए, ताकि संसाधनों को साफ किया जा सके।

```
<!-- AndroidManifest.xml -->
<service android:name=".FFmpegService" />
```

`Receiver` को पंजीकृत करके और संश्लेषण पूरा होने के बाद स्वयं □□□□□□□□ को समाप्त करके, आप बार-बार कॉल होने पर मेमोरी समस्याओं से बच सकते हैं।

संभावित समस्याएं

- □ □□□ **फ़ाइल प्लेबैक में समस्या**
  कुछ डिवाइस (जैसे □□□□□□ 2□) में डिफ़ॉल्ट `MediaPlayer` के माध्यम से □□□ एन्कोडेड ऑडियो प्ले करने में समस्या हो सकती है।

- **एन्कोडर समर्थन की कमी**
  यदि □□□-□□, □□3 आदि को सपोर्ट करना है, तो □□□□□□ को कंपाइल करते समय संबंधित विकल्पों को मैन्युअल रूप से सक्षम करना होगा। यदि कंपाइल स्क्रिप्ट संबंधित लाइब्रेरी या हेडर फ़ाइल नहीं ढूंढ पाती है, तो यह त्रुटि के साथ समाप्त हो जाएगी।

- **संश्लेषण गति**
  10 सेकंड का 1280×720 वीडियो और ऑडियो मिक्स करने में कई सेकंड से लेकर एक मिनट तक का समय लग सकता है। उपयोगकर्ता अनुभव के लिए, शायद उपयोगकर्ता को पहले सुनने देना और फिर अंतिम संश्लेषण करने का निर्णय लेना बेहतर होगा।

डबिंग शो के विशिष्ट कार्यान्वयन में, सामान्य प्रथाएं हैं: 1. मूल वीडियो, सबटाइटल्स, और "डबिंग के लिए आवश्यक खंडों को हटा दिया गया" ऑडियो फ़ाइल को पहले से डाउनलोड कर लें। 2. उपयोगकर्ता की आवाज़ रिकॉर्ड करें, संश्लेषण के दौरान केवल रिकॉर्डिंग और म्यूट खंडों को मर्ज करना आवश्यक होता है। 3. यदि स्थानीय संश्लेषण समय से संतुष्ट नहीं हैं, तो ऑडियो-वीडियो डेटा को सर्वर पर अपलोड करने का विकल्प चुन सकते हैं, जहां सर्वर साइड पर संश्लेषण किया जाता है, और पूरा होने के बाद इसे डाउनलोड कर लें।

□□□□□□□□ में □□□ का उपयोग करना

`ndk-build` को कमांड लाइन में दर्ज करना जरूरी नहीं है। बस □□□□□□□□ में प्रोजेक्ट पर राइट-क्लिक करें, □□□□□□□□ □□□□□ □ □□□ □□□□□□ □□□□□□□□ चुनें, और फिर हर बार "□□□" पर क्लिक करने पर `ndk-build` स्वचालित रूप से निष्पादित हो जाएगा।

एक क्लिक में □□□ हेडर फ़ाइल जनरेट करें

□□□ फ़ंक्शन हेडर फ़ाइल लिखना थोड़ा जटिल हो सकता है, लेकिन `javah` कमांड का उपयोग करके इसे स्वचालित रूप से जेनरेट किया जा सकता है।
□□□□□□□□ में, इसे एक एक्सटर्नल टूल के रूप में कॉन्फ़िगर किया जा सकता है, और निम्नलिखित कमांड का उपयोग करके हेडर फ़ाइल जेनरेट की जा सकती है:

```
javah -jni -classpath bin/classes -d jni com.example.ffmpeg.MyFFmpeg
```

(यह कमांड □□□ हेडर फ़ाइल बनाने के लिए उपयोग की जाती है और इसे अनुवादित करने की आवश्यकता नहीं है।)

निष्पादित करने के बाद, `jni` डायरेक्टरी में `com_example_ffmpeg_MyFFmpeg.h` जैसी फ़ाइल उत्पन्न होगी। फिर आपको बस अपने □ कोड में `#include` करना होगा और संबंधित फ़ंक्शन को लागू करना होगा।

सारांश

□□□□□□□□ पर □□□□□□□ को पोर्ट करने में कई प्रकार के ज्ञान शामिल हैं, जैसे कि □□□ वातावरण कॉन्फ़िगरेशन, □/□++ कंपाइल और लिंक, □□□ कॉल, ऑडियो और वीडियो कोडेक आदि। यदि आपको संश्लेषण करने में समस्या हो रही है, कुछ प्रारूपों का समर्थन नहीं है, या लिंकिंग में त्रुटियां आ रही हैं, तो कॉन्फ़िगरेशन और लॉग आउटपुट की जांच करने की आवश्यकता है। आशा है कि यह लेख आपको कुछ समस्याओं से बचने में मदद करेगा। यदि आप भी □□□□□□□ का उपयोग कर रहे हैं, तो कृपया टिप्पणियों में अपने अनुभव या समस्याएं साझा करें, ताकि हम एक-दूसरे से सीख और सहयोग कर सकें।