How to implement android framework

所謂的框架(framework)就是一組設計模式(design pattern)的集合, 不過這裡並不會討論design pattern跟framework的理論, 最主要是以自己開發android framework的實作為主.在android的框架分為java framework跟native framework.在native framework中又有client和service的設計, 扯到client和service一定就避免不了binder的機制.而java framework和native framework中間會需要一層JNI來做溝通介面. 因此在開發android framework需要設計以下幾項.

1. Native service
2. Client
3. Execute
4. JNI
5. Java service
6. Java manager

以下就分別來介紹這幾項的設計.以下的設計會利用c++語言來實作

# Native service

一開始要先訂出native service的功能, 所以會先寫header file

|  |
| --- |
| // File: NativeService.h  #ifndef ANDROID\_NATIVE\_SERVICE\_H  #define ANDROID\_NATIVE\_SERVICE\_H    #include <utils/threads.h>  #include <utils/RefBase.h>  #include <binder/IInterface.h>  #include <binder/BpBinder.h>  #include <binder/Parcel.h>    namespace android {    class NativeService : public BBinder  {  mutable Mutex mLock;  int32\_t mNextConnId;  public:      enum {  INIT\_MODE = 1,  };    static int instantiate();  NativeService ();  virtual ~ NativeService ();  virtual status\_t onTransact(uint32\_t, const Parcel&, Parcel\*, uint32\_t);  };    }; //namespace  #endif |

所宣告的NativeService class會繼承自BBinder class, 原因是需要借由實作onTransact函數來達到Binder的機制.宣告好header file, 接下來就是實作此NativeService class.

|  |
| --- |
| // File: NativeService.cpp  #define LOG\_NDEBUG 0 //此define會打開ALOGV,若沒有這個define, 即使在code中有加ALOGV也無法看到message.  #define LOG\_TAG "NativeService" //此define是logcat的filter TAG.  #include <cutils/log.h>    #include "NativeService.h"  #include <binder/IServiceManager.h>  #include <binder/IPCThreadState.h>  namespace android {    static struct sigaction oldact;  static pthread\_key\_t sigbuskey;    int NativeService::instantiate()  {  ALOGV("NativeService instantiate");  //  int r = defaultServiceManager()->addService(String16("customize.native.service"), new NativeService()); //註冊到Service Manager所管理的service list中, 名子叫"customize.native.service".  ALOGV("NativeService r = %d/n", r);  return r;  }    NativeService:: NativeService ()  {  ALOGV("NativeService created");      }    NativeService::~ NativeService ()  {  pthread\_key\_delete(sigbuskey);  ALOGV("NativeService destroyed");  }    status\_t NativeService::onTransact(uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)  {    ALOGV("[%s]: code[%d]\n", \_\_FUNCTION\_\_, code);  switch(code)  {  case INIT\_MODE: {  pid\_t pid = data.readInt32();  int mode = data.readInt32();  int ret = mode + 1;  reply->writeInt32(ret);  return NO\_ERROR;  }  break;  default:  return BBinder::onTransact(code, data, reply, flags);  }  }      }; //namespace |

由實作onTransact函數中會發現其中的值會從data參數讀出來,在寫入reply參數.這兩個參數可以當作是Client和Service共同溝通的容器.Binder機制最基本的機制就是利用一塊共享記憶體來達到Client process和 Service process的溝通介面, 這裡的data和reply容器就是這塊共享記憶體. Native service的設計大致上完成. 接下來就來設計Client modle.

# Client

先訂出Client class

|  |
| --- |
| // File: CustClient.h  #ifndef ANDROID\_CUST\_CLIENT\_\_H  #define ANDROID\_CUST\_CLIENT\_H  #include <utils/RefBase.h>  #include <binder/IServiceManager.h>  #include <binder/IPCThreadState.h>    namespace android  {  class CustClient: public RefBase  {  public:  enum {  INIT\_MODE = 1,  };      virtual ~ CustClient ();  CustClient ();    static sp<IBinder> BpNativeService;  static const sp<IBinder> getNativeService();    int init(int);    void exit();    };  }; //namespace  #endif // ANDROID\_CUST\_CLIENT\_\_H |

CustClient class 之所以要繼承自RefBase是因為想利用Android本身的Strong pointer來做記憶體管理.裡面還宣告了init函數給其他的modle呼叫.例外有兩個static的成員,是希望一開始有類別就有這兩個成員.而非載入此物件才有. 接下來看CustClient class的實作

|  |
| --- |
| // File: CustClient.cpp  #define LOG\_NDEBUG 0  #define LOG\_TAG " CustClient "  #include <cutils/log.h>  #include " CustClient.h"    namespace android  {  sp<IBinder> CustClient::BpNativeService = NULL;    CustClient:: CustClient ()  {  ALOGV("[%s]: enter\n", \_\_FUNCTION\_\_);  getNativeService();  }    CustClient::~ CustClient ()  {  ALOGV("[%s]: enter\n", \_\_FUNCTION\_\_);    exit();    }    void CustClient::exit()  {  ALOGV("[%s]: enter\n", \_\_FUNCTION\_\_);  //reset  BpNativeService.clear();  BpNativeService = NULL;    }      int CustClient::init(int mode)  {  ALOGV("[%s]: enter\n", \_\_FUNCTION\_\_);  Parcel data, reply;  int ret;    data.writeInt32(getpid());  data.writeInt32(mode);  ALOGE("[%s]: BpNativeService::create remote()->transact(), mode = %d\n", \_\_FUNCTION\_\_, mode);  BpNativeService ->transact(**INIT\_MODE**, data, &reply);  ret = reply.readInt32();  ALOGV("[%s]: ret = %d\n", \_\_FUNCTION\_\_, ret);  return ret;  }  const sp<IBinder> CustClient::getNativeService()  {  ALOGV("[%s]: enter\n",\_\_FUNCTION\_\_);  if(BpNativeService == NULL) {  sp<IServiceManager> sm = defaultServiceManager();  sp<IBinder> binder = sm->getService(String16(**"customize.native.service"**));  ALOGV("[%s]: defaultServiceManager %p\n",\_\_FUNCTION\_\_, sm.get());  if (binder == 0) {  ALOGW("getATKLedService not published, waiting...");  return NULL;  }  ALOGV("[%s]: binder %p\n",\_\_FUNCTION\_\_, binder.get());  BpNativeService = binder;    }  ALOGV("[%s]: BpNativeService %p\n",\_\_FUNCTION\_\_, BpNativeService.get());  return BpNativeService;  }    }; //namespace |

在CustClient class提供的init函數裡就會希望能呼叫NativeService來完成相對應的功能, 所以會借由 BpNativeService 這個指標物件來達成這個目標, 由其宣告可知, BpNativeService 是一個IBinder的指標物件, IBinder在Binder的世界是個Proxy,在這裡顧名思義就是NativeService代理人.在android framework中的Client要利用service 代理人來傳送給service的需求, 都會用transact函數搭配函數中的第一個參數code來告訴service. 而這個service代理人就是由service manager所提供的.回到程式碼來說明. 在CustClient建構子會去呼叫getNativeService函數, getNativeService函數的實作就是利用service manager來獲取Native service的proxy, 獲取的方法就是利用在Native Service自己跟Service manager註冊的"customize.native.service"的名子.這時候CustClient就已經擁有Native Service的代理人了, 一旦CustClient的init函數被呼叫,就會經由BpNativeService ->transact(INIT\_MODE, data, &reply); 這一行把需求送到Native service, 等到Native service執行完會把結果放在reply這個容器. 這裡有一點要說明的就是transact中的code會跟Native Service實作的onTransact函數中的code相匹配. 寫完Native service 跟CustClient,接下來就是編輯Android.mk file. 此檔案是一個make file, 用來告訴compiler要建構哪些檔案,要建構成何種目的檔.

|  |
| --- |
| //Android.mk  LOCAL\_PATH := $(call my-dir)  include $(CLEAR\_VARS)  LOCAL\_MODULE\_TAGS := optional  # our own branch needs these headers  LOCAL\_C\_INCLUDES +=  LOCAL\_PRELINK\_MODULE := false  LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_SHARED\_LIBRARIES)  LOCAL\_LDLIBS := -llog  LOCAL\_SHARED\_LIBRARIES := \  liblog \  libcutils \  libutils \  libbinder    LOCAL\_SRC\_FILES := \  NativeService.cpp \  CustClient.cpp  LOCAL\_MODULE := libNativeService  **include $(BUILD\_SHARED\_LIBRARY)** |

這些程式碼我將建構成動態程式庫,所以最後我加上include $(BUILD\_SHARED\_LIBRARY)這行指令將其建構成一個shared library(libNativeService.so)

程式庫是需要被人使用的,無法自己主動執行,所以需要時做一個執行檔(Execute).我將取名為CustServer.

# Execute

|  |
| --- |
| // File: CustServer.cpp  //#define LOG\_NDEBUG 0  #define LOG\_TAG "CustServer"  #include <cutils/log.h>  #include <sys/types.h>  #include <unistd.h>  #include <grp.h>  #include <binder/IPCThreadState.h>  #include <binder/ProcessState.h>  #include <binder/IServiceManager.h>  #include <utils/Log.h>  #include <NativeService.h>    using namespace android;    int main(int argc, char\*\* argv)  {  sp<ProcessState> proc(ProcessState::self());  sp<IServiceManager> sm = defaultServiceManager();  ALOGI("ServiceManager: %p", sm.get());    NativeService::instantiate();    ProcessState::self()->startThreadPool();  IPCThreadState::self()->joinThreadPool();  return 0;  } |

作法很簡單就是一執行就去呼叫NativeService的instantiate函數將Native service註冊到Service manager的service list中. 其Android.mk實作如下.

|  |
| --- |
| LOCAL\_PATH := $(call my-dir)  include $(CLEAR\_VARS)  LOCAL\_MODULE\_TAGS := optional  # our own branch needs these headers  LOCAL\_C\_INCLUDES += \  $(TOP)/vendor/altek/frameworks/native/libNativeService    LOCAL\_PRELINK\_MODULE := false  LOCAL\_LDLIBS := -llog  LOCAL\_SHARED\_LIBRARIES := \  liblog \  libcutils \  libutils \  libbinder \  libNativeService    LOCAL\_SRC\_FILES := CustServer.cpp  LOCAL\_MODULE := CustServer  **include $(BUILD\_EXECUTABLE)** |

由於會呼叫到NativeService class的instantiate函數,所以要指定LOCAL\_C\_INCLUDES和LOCAL\_SHARED\_LIBRARIES,在最後加上include $(BUILD\_EXECUTABLE)這行指令表示要建構出CustServer 執行檔.產生了執行檔總要有地方去執行吧. 在android framework中的執行程序都是一些server process. 而且都是一開機就會被system執行起來.這些server我分為兩類,一類是在進入主畫面才執行, 一類是在剛開機就執行.一般Native server都是在剛開機才執行. 所以可以在init.rc中去加以下的指令.

|  |
| --- |
| #  #CustServer service (start)  #  service CustServer /system/bin/CustServer  user system  group system  #  #CustServer service (end)  # |

在編輯init.rc會需要先懂一點Android Init Language, 由於Android Init Language並不是這裡的重點,所以就不在這裡介紹,有興趣可以參考一下\system\core\init\readme.txt 以上的寫法是告訴Android init process去啟動CustServer執行檔並取名為CustServer,且uid和gid都是system.

到這裡算是把Native service和對應的Cust Client建構完成.接下來就是跟java framework的溝通介面JNI(Java Native Interface.)

# JNI

在Android中的JNI function table只能對應一個java class. 其實作如下:

|  |
| --- |
| //com\_custframework\_server\_custservice\_jni.cpp  #define LOG\_NDEBUG 0  #define LOG\_TAG "cust\_jni"  #include <CustClient.h>  #include "jni.h"  #include "JNIHelp.h"  #include <cutils/log.h>  #include <assert.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  using namespace android;  static const char\* const kClassName = "com/custframework/server/CustService";    /\*  \* The method below are not thread-safe and not intended to be  \*/  static CustClient \*pCustClient = NULL;  static jint  custsvr\_init (JNIEnv \*env, jclass clazz, jint mode)  {  ALOGV("[%s] ++\n", \_\_FUNCTION\_\_);    if(pCustClient == NULL) {  pCustClient = new CustClient();  }    pCustClient ->init(mode);    ALOGV("[%s] --\n", \_\_FUNCTION\_\_);  return 0;  }  static jint  custsvr\_release(JNIEnv \*env, jobject thiz)  {  ALOGV("[%s] ++\n", \_\_FUNCTION\_\_);  jint ret = 0;    if(pCustClient!= NULL) {  ALOGV("[%s]: finish...\n", \_\_FUNCTION\_\_);  delete pCustClient;  pCustClient = NULL;  }    ALOGD("[%s] --\n", \_\_FUNCTION\_\_);  return ret;  }  **static JNINativeMethod gMethods[] = {**  **{"\_custsvr\_init", "(I)I", (void\*) custsvr\_init },**  **{"\_custsvr\_release", "()I", (void\*) custsvr\_release },**  **};**  static int registerMethods(JNIEnv\* env) {  jclass clazz;    /\* look up the class \*/  clazz = env->FindClass(kClassName);  if (clazz == NULL) {  ALOGE("Can't find class %s\n", kClassName);  return -1;  }    /\* register all the methods \*/  if (env->RegisterNatives(clazz, gMethods,  sizeof(gMethods) / sizeof(gMethods[0])) != JNI\_OK)  {  ALOGE("Failed registering methods for %s\n", kClassName);  return -1;  }  /\* fill out the rest of the ID cache \*/  return 0;  }  // ----------------------------------------------------------------------------  /\*  \* This is called by the VM when the shared library is first loaded.  \*/  jint JNI\_OnLoad(JavaVM\* vm, void\* reserved) {  JNIEnv\* env = NULL;  jint result = -1;  ALOGV("[%s]: enter\n", \_\_FUNCTION\_\_);  if (vm->GetEnv((void\*\*) &env, JNI\_VERSION\_1\_4) != JNI\_OK) {  ALOGE("ERROR: GetEnv failed\n");  goto bail;  }  assert(env != NULL);  if (registerMethods(env) != 0) {  ALOGE("ERROR: PlatformLibrary native registration failed\n");  goto bail;  }  /\* success -- return valid version number \*/  result = JNI\_VERSION\_1\_4;  bail:  return result;  } |

我在這裡的JNI實作兩個jni function, custsvr\_init和custsvr\_release. 其中會有個jni function table用來註冊這兩個function給java class呼叫. 其table中的一些member定義不是這裡的重點,所以就不在這裡詳細解答.在前面有說到一個jni function table只對應一個java class, 在這裡對應的java class我指定為static const char\* const kClassName = "com/custframework/server/CustService"; 所以待會會實作一個java的CustService class. 其Android.mk實作如下:

|  |
| --- |
| LOCAL\_PATH:= $(call my-dir)  include $(CLEAR\_VARS)  # [optional, user, eng]  # eng = required  # optinal = no install on target  LOCAL\_MODULE\_TAGS := optional  # This is the target being built.  LOCAL\_MODULE:= libcust\_jni  # Target install path.  LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_SHARED\_LIBRARIES)  # All of the source files that we will compile.  LOCAL\_SRC\_FILES:= \  com\_custframework\_server\_custservice\_jni.cpp  # All of the shared libraries we link against.  LOCAL\_SHARED\_LIBRARIES := \  libandroid\_runtime \  libnativehelper \  libcutils \  libutils \  libNativeService  # No static libraries.  LOCAL\_STATIC\_LIBRARIES :=  # Also need the JNI headers.  LOCAL\_C\_INCLUDES += \  $(JNI\_H\_INCLUDE) \  vendor/altek/frameworks/native/libNativeService  # No specia compiler flags.  LOCAL\_CFLAGS +=  # Don't prelink this library. For more efficient code, you may want  # to add this library to the prelink map and set this to true.  LOCAL\_PRELINK\_MODULE := false  **include $(BUILD\_SHARED\_LIBRARY)** |

一般JNI都是建構成一個動態程式庫.所以加了include $(BUILD\_SHARED\_LIBRARY)指令, 由於會需要呼叫到CustClient class的成員函數, 所以需要指定LOCAL\_C\_INCLUDES和LOCAL\_SHARED\_LIBRARIES.接下來就來實作jni相對應的java class也就是Java service.

# Java Service

實作如下:

|  |
| --- |
| //CustService.java  package com.custframework.server; //需要與在jin指定的package name一樣.  import android.content.Context;  import atkframework.hardware.IHapticService;  import android.os.Binder;  import android.os.ParcelFileDescriptor;  import android.os.RemoteException;  import android.os.IBinder;  import android.util.Config;  import android.util.Log;  import java.util.ArrayList;  /\*\*  \* Class that manages the device's sensors. It register clients and activate  \* the needed sensors. The sensor events themselves are not broadcasted from  \* this service, instead, a file descriptor is provided to each client they  \* can read events from.  \*/  //需要與在jin指定的class name一樣.  public final class CustService {  static final String TAG = CustService.class.getSimpleName();  private static final boolean DEBUG = true;  static {  //裡面的字串就是jni 程式庫的名字去掉前面的lib和後面的.so所得到的, 例如lib**cust\_jni**.so  System.loadLibrary("cust\_jni");  }    public CustService (Context context) {  PrintI("CustService startup");    }      public void setType(int mode) {  PrintI("setType mode = " + mode);  \_custsvr\_init (mode);  }      public int release() {  PrintI("release");  int ret = \_custsvr\_release();  return ret;  }      private void PrintI(String str) {    if(DEBUG) Log.i(TAG, str);  }    private native final int \_custsvr\_init(int mode);  private native final int \_custsvr\_release();  } |

在最底下有兩行用native修飾的宣告函數\_custsvr\_init和\_custsvr\_release,這兩個函數名子一定要跟在jni fucntion table的第一個member同樣名子,在java中由native修飾的宣告函數都是實作在native layer.所以會有一個jni function table來做映射. 接下來直接看CustManager java的實作, 用意是希望可以利用CustManager class來對個個的自訂service做個包裝.

# Java Manager

實作如下:

|  |
| --- |
| //CustManager.java  package cust.framework.manager;  import android.content.Context;  import android.os.Binder;  import android.os.Bundle;  import android.os.Parcelable;  import android.os.ParcelFileDescriptor;  import android.os.Process;  import android.os.RemoteException;  import android.os.Handler;  import android.os.Message;  import android.os.ServiceManager;  import android.util.Log;  import com.custframework.server.CustService;  /\*\*  \* Class that lets you access the Customize Services.  \*/  public final class CustManager  {  private static final String TAG = " CustManager ";  private static final boolean DEBUG = true;    //cust service service  public static CustService getCustService(Context context, int mode) {  PrintI("####getCustService ####");  CustService custService = new CustService (context);  custService.setType(type);  return custService;  }      private static void PrintI(String str) {    if(DEBUG) Log.i(TAG, str);  }  } |

實作完CustService.java和CustManager.java接下來就來實作Android.mk

|  |
| --- |
| LOCAL\_PATH := $(call my-dir)  # the library  # =============================================================  include $(CLEAR\_VARS)  LOCAL\_SRC\_FILES := \  $(call all-subdir-java-files)  LOCAL\_MODULE\_TAGS := optional  LOCAL\_MODULE := custframework  include $(BUILD\_JAVA\_LIBRARY) |

經過android建構之後會產生custframework.jar檔案.到此會有以下的建構產生的檔案.

* libNativeService.so: 包含CustClient和NativeService的函式庫
* CustServer: 用來啟動Customize native service並等待上層需求.
* libcust\_jni.so: 作為java呼叫CustClient函數的橋梁.
* custframework.jar: 提供給app使用的一個java函數庫.

哪開發應用層的如何使用這個custframework.jar? 在開發應用層會有兩類,一類是用平台的環境去做開發,是謂系統內建應用程式, 另一類是Eclipse開發環境去做開發.以下就大概的來說明如何在這兩種環境去使用custframework.jar.

* **系統內建應用程式開發**

1. 編輯jar config file

|  |
| --- |
| //com.custframework.server.xml  <?xml version="1.0" encoding="utf-8"?>  <permissions>  <library name=" com.custframework.server "  file="/system/framework/custframework.jar"/>  </permissions> |

此設定檔是用來設定使用此jar file的permission.也是說應用層想使用這個custframework.jar, 需要指定此jar lib file name " com.custframework.server ".

1. 編輯Android.mk

|  |
| --- |
| LOCAL\_PATH := $(my-dir)  ########################  include $(CLEAR\_VARS)  LOCAL\_MODULE := com.custframework.server.xml  LOCAL\_MODULE\_TAGS := optional  LOCAL\_MODULE\_CLASS := ETC  # This will install the file in /system/etc/permissions  #  LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_ETC)/permissions  LOCAL\_SRC\_FILES := $(LOCAL\_MODULE)  include $(BUILD\_PREBUILT) |

在android中一般跟permissions有關的設定檔都會放在裝置的/system/etc/permissions下.

* **Eclipse開發環境開發**

1. add jar to project

一般由平台建構出來的jar file裡面包的是被優化過的dex file,而不是一些.class file. 所以需要自己把jar file中的classes.dex取出來.(jar file是一個壓縮檔,解壓縮之後就看到classes.dex). 再利用dex2jar轉成classes\_dex2jar.jar, 然後在rename為custframework.jar, 這時候的jar file裡就都是一些.class file.將此jar file 複製到app project下的libs 資料夾下. 在app project上按右鍵,點選 Build Path\Configure BuildPath, 在點Add JARs按鈕選擇剛剛複製的custframework.jar.

1. copy jni so file to libs/armeabi

在libs下新增一個資料夾名為armeabi,將libcust\_jni.so複製到裡面.