Crash监控

Crash（应用崩溃）是由于代码异常而导致 App 非正常退出，导致应用程序无法继续使用，所有工作都停止的现象。发生 Crash 后需要重新启动应用（有些情况会自动重启），而且不管应用在开发阶段做得多么优秀，也无法避免 Crash 发生，特别是在 Android 系统中，系统碎片化严重、各 ROM 之间的差异，甚至系统Bug，都可能会导致Crash的发生。

在 Android 应用中发生的 Crash 有两种类型，Java 层的 Crash 和 Native 层 Crash。这两种Crash 的监控和获取堆栈信息有所不同。

Java Crash

Java的Crash监控非常简单，Java中的Thread定义了一个接口：`UncaughtExceptionHandler`；用于处理未捕获的异常导致线程的终止（**注意：catch了的是捕获不到的**），当我们的应用crash的时候，就会走`UncaughtExceptionHandler.uncaughtException`，在该方法中可以获取到异常的信息，我们通过`Thread.setDefaultUncaughtExceptionHandler`该方法来设置线程的默认异常处理器，我们可以将异常信息保存到本地或者是上传到服务器，方便我们快速的定位问题。

```
public class CrashHandler implements Thread.UncaughtExceptionHandler{

    private static final String FILE_NAME_SUFFIX = ".trace";
    private static Thread.UncaughtExceptionHandler mDefaultCrashHandler;
    private static Context mContext;

    private CrashHandler(){
    }


    public static void init(@NonNull Context context){
        //默认为：RuntimeInit#KillApplicationHandler
        mDefaultCrashHandler = Thread.getDefaultUncaughtExceptionHandler();
        Thread.setDefaultUncaughtExceptionHandler(this);
        mContext = context.getApplicationContext();
    }

    /**
     * 当程序中有未被捕获的异常，系统将会调用这个方法
     *
     * @param t 出现未捕获异常的线程
     * @param e 得到异常信息
     */
    @Override
    public void uncaughtException(Thread t, Throwable e){
        try{
            //自行处理：保存本地
            File file = dealException(e);
            //上传服务器
            //......
        } catch (Exception e1){
            e1.printStackTrace();
        } finally{
            //交给系统默认程序处理
```

```
            if (mDefaultCrashHandler != null){
                mDefaultCrashHandler.uncaughtException(t, e);
            }
        }
    }

    /**
     * 导出异常信息到SD卡
     *
     * @param e
     */
    private File dealException(Thread t,Throwable e) throw Exception{
        String time = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new
Date());
        File f = new
File(context.getExternalCacheDir().getAbsoluteFile(),"crash_info");
        if (!f.exists()) {
            f.mkdirs();
        }
        File crashFile = new File(f, time + FILE_NAME_SUFFIX);
        File file = new File(PATH + File.separator + time + FILE_NAME_SUFFIX);

        //往文件中写入数据
        PrintWriter pw = new PrintWriter(new BufferedWriter(new
FileWriter(file)));
        pw.println(time);
        pw.println("Thread: "+ t.getName());
        pw.println(getPhoneInfo());
        e.printStackTrace(pw); //写入crash堆栈
        pw.close();
        return file;
    }

     private String getPhoneInfo() throws PackageManager.NameNotFoundException{
        PackageManager pm = mContext.getPackageManager();
        PackageInfo pi = pm.getPackageInfo(mContext.getPackageName(),
PackageManager.GET_ACTIVITIES);
        StringBuilder sb = new StringBuilder();
        //App版本
        sb.append("App Version: ");
        sb.append(pi.versionName);
        sb.append("_");
        sb.append(pi.versionCode + "\n");

        //Android版本号
        sb.append("OS Version: ");
        sb.append(Build.VERSION.RELEASE);
        sb.append("_");
        sb.append(Build.VERSION.SDK_INT + "\n");

        //手机制造商
        sb.append("Vendor: ");
        sb.append(Build.MANUFACTURER + "\n");

        //手机型号
        sb.append("Model: ");
        sb.append(Build.MODEL + "\n");
```

```
        //CPU架构
        sb.append("CPU: ");
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP){
            sb.append(Arrays.toString(Build.SUPPORTED_ABIS));
        } else {
            sb.append(Build.CPU_ABI);
        }
        return sb.toString();
    }
}
```

NDK Crash

相对于Java的Crash，NDK的错误无疑更加让人头疼，特别是对初学NDK的同学，不说监控，就算是错误堆栈都不知道怎么看。

Linux信号机制

信号机制是Linux进程间通信的一种重要方式，Linux信号一方面用于正常的进程间通信和同步，另一方面它还负责监控系统异常及中断。当应用程序运行异常时，Linux内核将产生错误信号并通知当前进程。当前进程在接收到该错误信号后，可以有三种不同的处理方式。

- 忽略该信号；
- 捕捉该信号并执行对应的信号处理函数（信号处理程序）；
- 执行该信号的缺省操作（如终止进程）；

当Linux应用程序在执行时发生严重错误，一般会导致程序崩溃。其中，Linux专门提供了一类crash信号，在程序接收到此类信号时，缺省操作是将崩溃的现场信息记录到核心文件，然后终止进程。

常见崩溃信号列表：

信号	描述
SIGSEGV	内存引用无效。
SIGBUS	访问内存对象的未定义部分。
SIGFPE	算术运算错误，除以零。
SIGILL	非法指令，如执行垃圾或特权指令
SIGSYS	糟糕的系统调用
SIGXCPU	超过CPU时间限制。
SIGXFSZ	文件大小限制。

一般的出现崩溃信号，Android系统默认缺省操作是直接退出我们的程序。但是系统允许我们给某一个进程的某一个特定信号注册一个相应的处理函数（**signal**），即对该信号的默认处理动作进行修改。因此NDK Crash的监控可以采用这种信号机制，捕获崩溃信号执行我们自己的信号处理函数从而捕获NDK Crash。

墓碑

> 此处了解即可，普通应用无权限读取墓碑文件，墓碑文件位于路径*/data/tombstones/*下。解析墓碑文件与后面的***breakPad***都可使用`addr2line`工具。

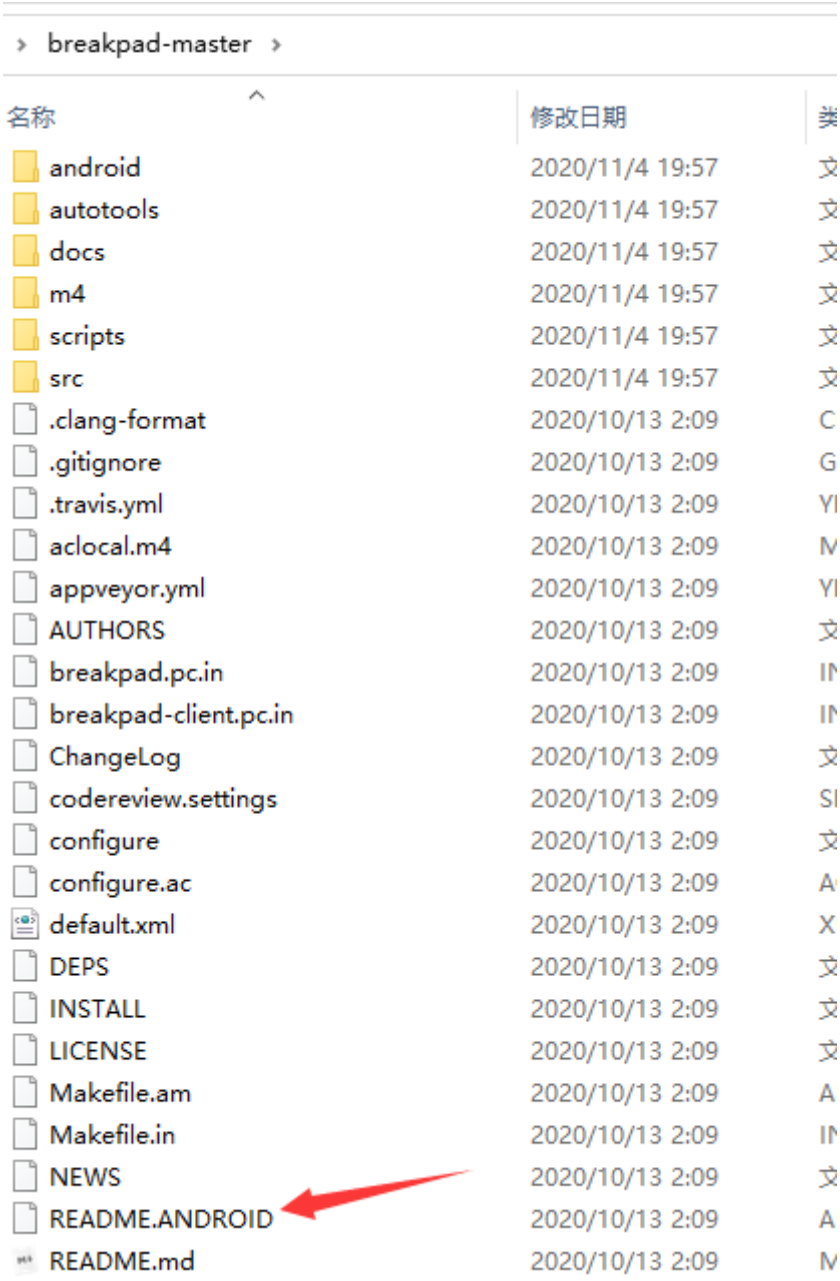
Android本机程序本质上就是一个Linux程序，当它在执行时发生严重错误，也会导致程序崩溃，然后产生一个记录崩溃的现场信息的文件，而这个文件在Android系统中就是 **tombstones** 墓碑文件。

BreakPad

Google breakpad是一个跨平台的崩溃转储和分析框架和工具集合，其开源地址是：https://github.com/google/breakpad。breakpad在Linux中的实现就是借助了Linux信号捕获机制实现的。因为其实现为C++，因此在Android中使用，必须借助NDK工具。

引入项目

将Breakpad源码下载解压，首先查看README.ANDROID文件。



打开 **README.ANDROID**

```
If you're using the ndk-build build system, you can follow
these simple steps:
      导入 android/google_breakpad/Android.mk到自己项目的Android.mk中
  1/ Include android/google_breakpad/Android.mk from your own
     project's Android.mk

     This can be done either directly, or using ndk-build's
     import-module feature.

  2/ Link the library to one of your modules by using:
     在自己的项目Android.mk中加入下面配置，依赖breakpad
     LOCAL_STATIC_LIBRARIES += breakpad_client

NOTE: The client library requires a C++ STL implementation,
      which you can select with APP_STL in your Application.mk

      It has been tested succesfully with both STLport and GNU libstdc++
```

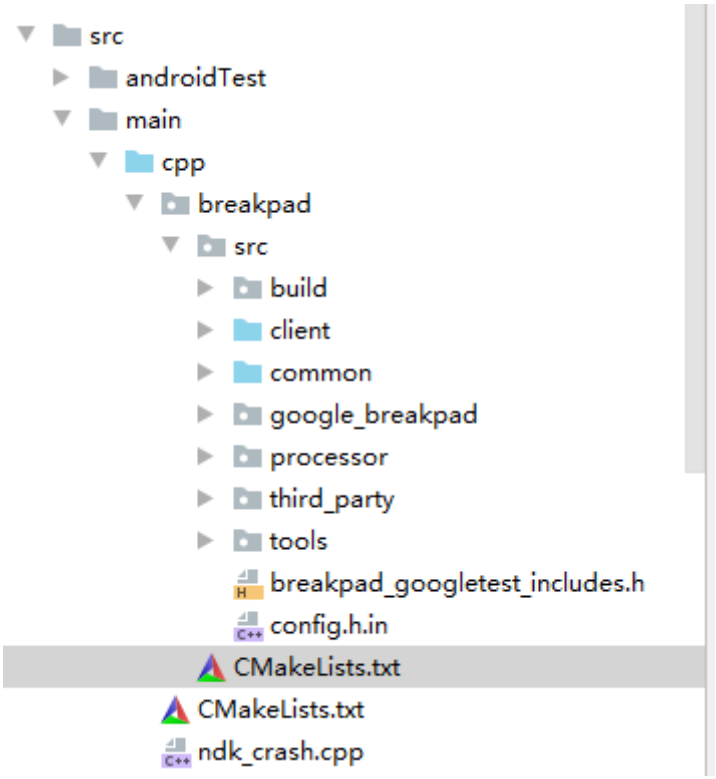
按照文档中的介绍，如果我们使用Android.mk 非常简单就能够引入到我们工程中，但是目前NDK默认的构建工具为：CMake，因此我们做一次移植。查看android/google_breakpad/Android.mk

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

include $(CLEAR_VARS)

#最后编译出 libbreakpad_client.a
LOCAL_MODULE := breakpad_client
#指定c++源文件后缀名
LOCAL_CPP_EXTENSION := .cc

# 强制构建系统以 32 位 arm 模式生成模块的对象文件
LOCAL_ARM_MODE := arm

# 需要编译的源码
LOCAL_SRC_FILES := \
    src/client/linux/crash_generation/crash_generation_client.cc \
    src/client/linux/dump_writer_common/thread_info.cc \
    src/client/linux/dump_writer_common/ucontext_reader.cc \
    src/client/linux/handler/exception_handler.cc \
    src/client/linux/handler/minidump_descriptor.cc \
    src/client/linux/log/log.cc \
    src/client/linux/microdump_writer/microdump_writer.cc \
    src/client/linux/minidump_writer/linux_dumper.cc \
    src/client/linux/minidump_writer/linux_ptrace_dumper.cc \
    src/client/linux/minidump_writer/minidump_writer.cc \
    src/client/minidump_file_writer.cc \
    src/common/convert_UTF.cc \
    src/common/md5.cc \
    src/common/string_conversion.cc \
    src/common/linux/breakpad_getcontext.S \
    src/common/linux/elfutils.cc \
    src/common/linux/file_id.cc \
    src/common/linux/guid_creator.cc \
    src/common/linux/linux_libc_support.cc \
    src/common/linux/memory_mapped_file.cc \
    src/common/linux/safe_readlink.cc
```

```
#导入头文件
LOCAL_C_INCLUDES        := $(LOCAL_PATH)/src/common/android/include \
                           $(LOCAL_PATH)/src \
                           $(LSS_PATH) #注意这个目录
#导出头文件
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_C_INCLUDES)
#使用android ndk中的日志库log
LOCAL_EXPORT_LDLIBS     := -llog

#编译static静态库-》类似java的jar包
include $(BUILD_STATIC_LIBRARY)
```

> 注意：mk文件中 LOCAL_C_INCLUDES 的 **LSS_PATH** 是个坑

对照Android.mk文件，我们在自己项目的cpp(工程中C/C++源码)目录下创建breakpad目录，并将下载的breakpad源码根目录下的src目录全部复制到我们的项目中：



接下来在breakpad目录下创建**CMakeList.txt**文件:

```
cmake_minimum_required(VERSION 3.4.1)
#对应android.mk中的 LOCAL_C_INCLUDES
include_directories(src src/common/android/include)
#开启arm汇编支持，因为在源码中有 .S文件（汇编源码）
enable_language(ASM)

#生成 libbreakpad.a 并指定源码，对应android.mk中 LOCAL_SRC_FILES+LOCAL_MODULE
add_library(breakpad STATIC
        src/client/linux/crash_generation/crash_generation_client.cc
        src/client/linux/dump_writer_common/thread_info.cc
        src/client/linux/dump_writer_common/ucontext_reader.cc
        src/client/linux/handler/exception_handler.cc
```

```
        src/client/linux/handler/minidump_descriptor.cc
        src/client/linux/log/log.cc
        src/client/linux/microdump_writer/microdump_writer.cc
        src/client/linux/minidump_writer/linux_dumper.cc
        src/client/linux/minidump_writer/linux_ptrace_dumper.cc
        src/client/linux/minidump_writer/minidump_writer.cc
        src/client/minidump_file_writer.cc
        src/common/convert_UTF.cc
        src/common/md5.cc
        src/common/string_conversion.cc
        src/common/linux/breakpad_getcontext.S
        src/common/linux/elfutils.cc
        src/common/linux/file_id.cc
        src/common/linux/guid_creator.cc
        src/common/linux/linux_libc_support.cc
        src/common/linux/memory_mapped_file.cc
        src/common/linux/safe_readlink.cc)

#链接 log库，对应android.mk中 LOCAL_EXPORT_LDLIBS
target_link_libraries(breakpad log)
```

在cpp目录下（breakpad同级）还有一个CMakeList.txt文件，它的内容是：

```
cmake_minimum_required(VERSION 3.4.1)

#引入breakpad的头文件（api的定义）
include_directories(breakpad/src breakpad/src/common/android/include)
#引入breakpad的cmakelist，执行并生成libbreakpad.a (api的实现，类似java的jar包)
add_subdirectory(breakpad)

#生成libbugly.so 源码是：ndk_crash.cpp(我们自己的源码，要使用breakpad)
add_library(
        bugly
        SHARED
        ndk_crash.cpp)


target_link_libraries(
        bugly
        breakpad #引入breakpad的库文件(api的实现)
        log)
```

此时执行编译，会在`#include "third_party/lss/linux_syscall_support.h"`报错，无法找到头文件。此文件从：https://chromium.googlesource.com/external/linux-syscall-support/+/refs/heads/master 下载（**需要翻墙**）放到工程对应目录即可。

`ndk_crash.cpp`源文件中的实现为：

```
//
// Created by Administrator on 2020/11/4.
//


#include <jni.h>
#include <android/log.h>
```

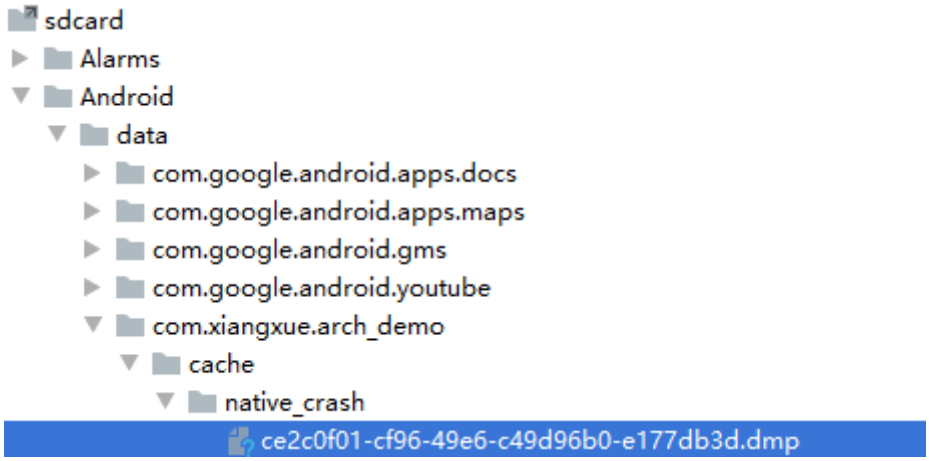
```
#include "breakpad/src/client/linux/handler/minidump_descriptor.h"
#include "breakpad/src/client/linux/handler/exception_handler.h"

bool DumpCallback(const google_breakpad::MinidumpDescriptor &descriptor,
                  void *context,
                  bool succeeded) {
    __android_log_print(ANDROID_LOG_ERROR, "ndk_crash", "Dump path: %s",
descriptor.path());
    //如果回调返回true，Breakpad将把异常视为已完全处理，禁止任何其他处理程序收到异常通知。
    //如果回调返回false，Breakpad会将异常视为未处理，并允许其他处理程序处理它。
    return false;
}

extern "C"
JNIEXPORT void JNICALL
Java_com_enjoy_crash_CrashReport_initBreakpad(JNIEnv *env, jclass type, jstring
path_) {
    const char *path = env->GetStringUTFChars(path_, 0);
    //开启crash监控
    google_breakpad::MinidumpDescriptor descriptor(path);
    static google_breakpad::ExceptionHandler eh(descriptor, NULL, DumpCallback,
NULL, true, -1);
    env->ReleaseStringUTFChars(path_, path);
}

//测试用
extern "C"
JNIEXPORT void JNICALL
Java_com_enjoy_crash_CrashReport_testNativeCrash(JNIEnv *env, jclass clazz) {

    int *i = NULL;
    *i = 1;
}
```

注意JNI方法的方法名对应了java类，创建Java源文件：`com.enjoy.crash.CrashReport`

```
package com.enjoy.crash;

import android.content.Context;

import java.io.File;

public class CrashReport {

    static {
        System.loadLibrary("bugly");
    }


    public static void init(Context context) {
        //开启java监控
        Context applicationContext = context.getApplicationContext();
        CrashHandler.init(applicationContext);

         //开启ndk监控
        File file = new File(context.getExternalCacheDir(), "native_crash");
        if (!file.exists()) {
```

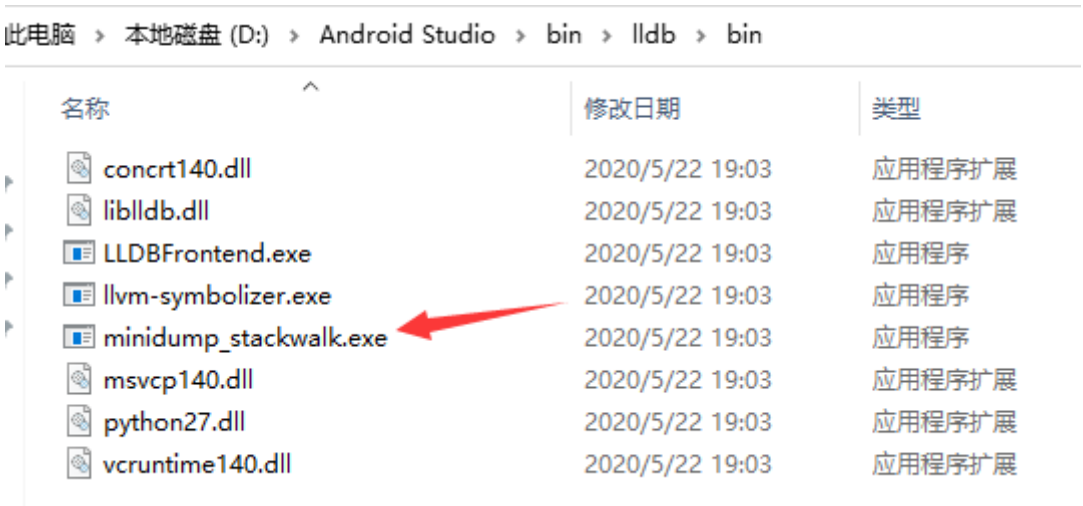
```
            file.mkdirs();
        }
        initBreakpad(file.getAbsolutePath());
    }

    // C++: Java_com_enjoy_crash_CrashReport_initBreakpad
    private static native void initBreakpad(String path);

    // C++: Java_com_enjoy_crash_CrashReport_testNativeCrash
    public static native void testNativeCrash();

    public static int testJavaCrash() {
        return 1 / 0;
    }

}
```

此时，如果出现NDK Crash，会在我们指定的目录：`/sdcard/Android/Data/[packageName]/cache/native_crash`下生成NDK Crash信息文件。



Crash解析

采集到的Crash信息记录在minidump文件中。minidump是由微软开发的用于崩溃上传的文件格式。我们可以将此文件上传到服务器完成上报，但是此文件没有可读性可言，要将文件解析为可读的崩溃堆栈需要按照breakpad文档编译`minidump_stackwalk`工具，而Windows系统编译个人不会。不过好在，无论你是 Mac、windows还是ubuntu在 Android Studio 的安装目录下的`bin\lldb\bin`里面就存在一个对应平台的`minidump_stackwalk`。



使用这里的工具执行：

```
minidump_stackwalk xxxx.dump > crash.txt
```

打开`crash.txt`内容为：

```
Operating system: Android
                  0.0.0 Linux 4.4.124+ #1 SMP PREEMPT Wed Jan 30 07:13:09 UTC
2019 i686
CPU: x86  // abi类型
     GenuineIntel family 6 model 31 stepping 1
     3 CPUs

GPU: UNKNOWN

Crash reason:  SIGSEGV  //内存引用无效 信号
Crash address: 0x0
Process uptime: not available

Thread 0 (crashed)  //crashed：出现crash的线程
 0  libbugly.so + 0x1feab //crash的so与寄存器信息
    eip = 0xd5929eab   esp = 0xffa85f30   ebp = 0xffa85f38   ebx = 0x0000000c
    esi = 0xd71a3f04   edi = 0xffa86128   eax = 0xffa85f5c   ecx = 0xefb19400
    edx = 0x00000000   efl = 0x00210286
    Found by: given as instruction pointer in context
 1  libart.so + 0x5f6a18
    eip = 0xef92ea18   esp = 0xffa85f40   ebp = 0xffa85f60
    Found by: previous frame's frame pointer

Thread 1
......
```

接下来使用 Android NDK 里面提供的 addr2line 工具将寄存器地址转换为对应符号。addr2line 要用和自己 so 的 ABI 匹配的目录，同时需要使用有符号信息的so(一般debug的就有)。

> 因为我使用的是模拟器x86架构，因此addr2line位于：
>
> Android\Sdk\ndk\21.3.6528147\toolchains\x86-4.9\prebuilt\windows-x86_64\bin\i686-linux-android-addr2line.exe

```
i686-linux-android-addr2line.exe -f -C -e libbugly.so 0x1feab
```

```
C:\Users\Admin\AppData\Local\Android\Sdk\ndk\21.3.6528147\toolchains\x86-4.9\prebuilt\windows-x86_64\bin>i686-linux-android-addr2line.exe
-f -C -e C:\Users\Admin\Desktop\project\crash\build\intermediates\cmake\debug\obj\x86\libbugly.so 0x1feab
Java_com_enjoy_crash_CrashReport_testNativeCrash
C:/Users/Admin/Desktop/project/crash/src/main/cpp/ndk_crash.cpp:35
```

```
30  extern "C"
31  JNIEXPORT void JNICALL
32  CrashReport.testNativeCrash(JNIEnv *env, jclass CrashReport clazz) {
33
34      int *i = NULL;
35      *i = 1;
36  }
```