[SCA] Crash 安全应急方案 - 设计文档

Ownership

Product Manager	*The PIC of Product manager*
Project Manager	*The PIC of Project Manager*
Native Dev	*The PIC of Android or iOS Dev*
RN Dev	*The PIC of React native dev*
Server Dev	*The PIC of non-native dev*
Designer	*The PIC of designer*
QA	*The PIC of QA*

Resources

PRD	*The link of PRD(s), created by product manager*
Figma	*The link of UI design*
Transify	*The link of transify in https://transify.seagroup.com*

Git Repo(Optional)	*The git repo or branch for this project if necessary to indicate*
Dependent Service Doc(Optional)	*SDK integration doc, official guideline, etc.*
Project Schedule Page (Optional)	*Usually created by project manager, timeline of the document*
Feasibility Study Doc (Optional)	*Usually feasibility study is ahead of PRD.*

Background

对于Android应用来说，运行过程中发生错误则会发生崩溃（Crash，即抛出未处理的异常），系统的默认处理是关闭或重启应用。这对于用户来说是一个相当糟糕的体验，如果启动过程中崩溃甚至会导致应用不可用。预防崩溃是开发工作的重中之重，但一旦崩溃发生，我们还需要一套应急方案，特别是对于启动崩溃，我们希望也能保证用户可以一定程度上使用我们的应用。

Overall Design (Optional)

Brand new

- Current

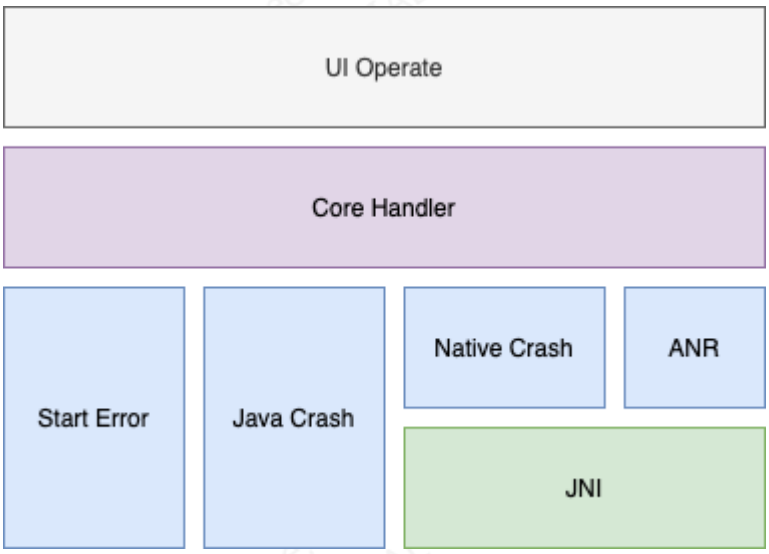
目前的崩溃完全走的Android系统的默认处理流程，崩溃后杀掉应用进程，或根据用户操作重启应用。对于重启仍然崩溃的问题无能为力。

- Improvement

拦截系统处理流程，根据情况选择不同的处理方式，包括但不限于清空应用数据、降级兜底和热更新。

调研文档：https://docs.google.com/document/d/1QknqmAJ6auXYFduOnx2DVG-3zz2HAipMTdy8BVE2Y20/edit?usp=sharing

Code organization



底层捕获三种崩溃情况，并屏蔽系统的处理，传递到上层的Crash Handler。Crash Handler调起安全进程，展示操作界面，操作界面根据用户操作进行后续操作。

- 后续操作包括关闭应用、重启应用和清除应用数据，都是常规操作，可以统一实现。
- 降级兜底，可能跟业务绑定，如旧的业务比较稳定可作为兜底逻辑，新上的业务就作为非兜底逻辑，一旦降级应用就会回退到旧的业务，不走新的逻辑。本框架只需要提供一个全局的兜底开关管理，具体的兜底策略需要业务自行实现。
- 热修复或覆盖安装，这里可能涉及到不同的热修复策略，也不宜统一实现。本框架可以考虑提供给业务方自定义处理的接口，如自定义一个应用升级的处理实现，去下载特定的补丁包或全量包。

Interface Design

对外提供的方法接口包括初始化方法，兜底状态获取方法，以及自定义处理方案的接口。

Detailed Design

1. 触发时机

比较普遍的触发是上述提到的捕获到崩溃时，但由于要捕获所有崩溃比较困难，尤其在启动时期，所以我们可以参考天猫APP的安全模式方案，增加一个启动异常的触发时机。另外，应用在后台的崩溃，我们希望让用户无感知，这种情况下只需要拦截系统处理，直接kill应用进程。所以触发时机是：

- 应用在前台。
- 启动异常或捕获到崩溃。

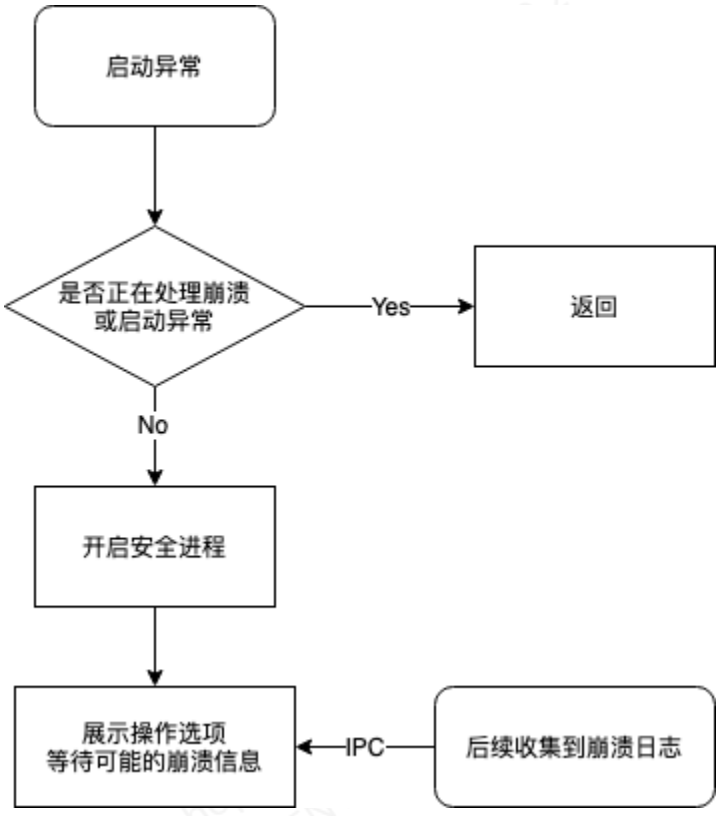
2. 启动异常

启动异常的判断方案：APP启动时记录一个flag值，满足以下条件时，将flag值清空：

- APP正常启动10秒
- 用户正常退出应用
- 用户主动从前台切换到后台

如果在启动阶段发生异常，则flag值不会清空，通过flag值就可以判断出客户端是否异常退出，每次异常退出，flag值都会+1。

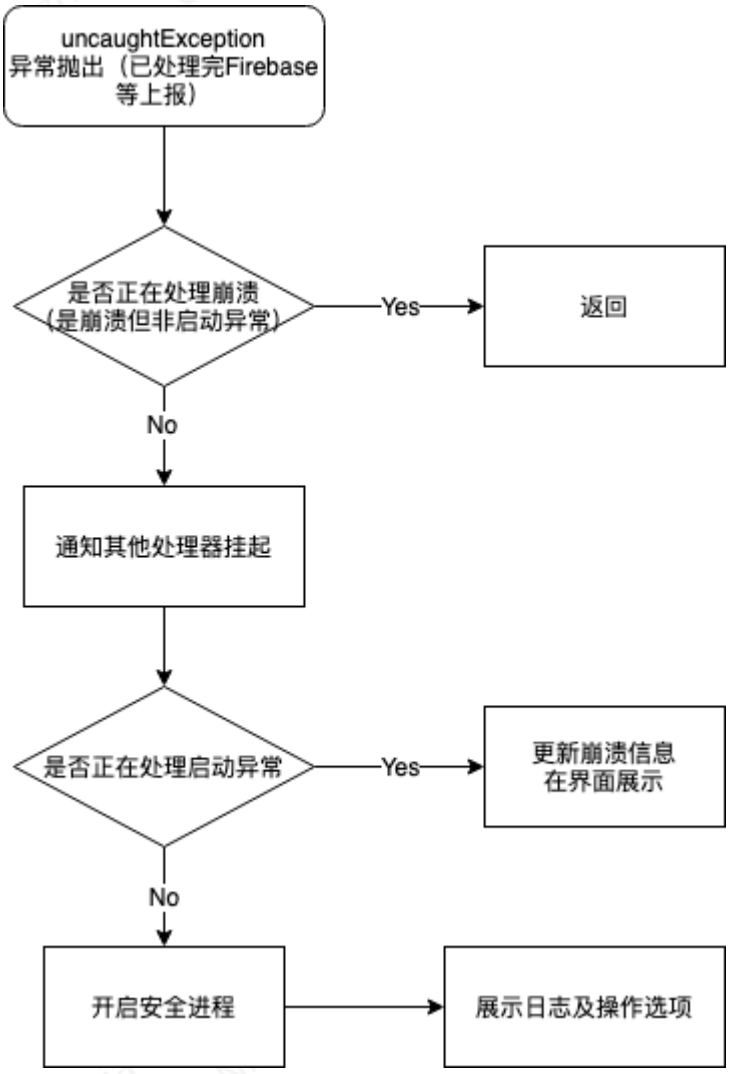
这样实现的启动异常无法拿到异常的信息，来分析原因，所以我们希望如果我们处理启动异常时，能够继续监测崩溃，从而来获取异常的原因。处理流程如下：



3. Java Crash

Java层的崩溃作为目前主要的崩溃，是需要重点优先处理的。

3.1 Java Crash的处理流程



整个Java层崩溃的处理流程，如上图所示。起始点是捕获到Java层的崩溃；然后是防止重复处理崩溃，如果是在处理启动异常则仍往下执行；接着是防止处理过程中，发生其他崩溃，导致重复处理，由于这种情况非常少，所以选择挂起其他的处理器进行忽略，这个步骤需要在一个统一的调度器中处理；接着需要开启一个安全进程来处理后续操作，来给用户展示操作界面等。

3.2 UncaughtExceptionHandler

（此处简称Handler）

我们需要拦截系统的Handler，但不能影响Firebase的Handler。由于系统只保留最后一次设置的Handler，所以设置的时机很重要。经阅读Firebase的源码，Firebase在完成上报之后会无条件执行上一次设置的Handler，所以我们设置的时机必须在Firebase初始化之前。Firebase初始化是在ContentProvider中，我们必须在Application的attachBaseContext中初始化并设置我们的Handler。

3.3 安全进程

用于我们需要给用户展示一个操作界面，我们可以通过打开Activity实现开启安全子进程。需要把在主进程获取的日志信息带到这个Activity中，考虑到存在信息延迟到达的情况（启动异常+捕获崩溃），可以将该Activity的启动模式设为SingleTop，新的信息到来时只需要再startActivity把信息带入即可，在onNewIntent 刷新页面显示，也完成了IPC跨进程通信。这里要注意，包含界面展示的信息在内，传入的信息大小不能超过1M，我们可以把这个信息量阈值设成100KB。

3.4 日志信息获取

日志需包含崩溃的调用栈信息，对于Java Crash，可以利用在捕获崩溃时获取到的异常对象Throwable，通过其printStackTrace方法获取stack trace信息。

```
//stack trace

StringWriter sw = new StringWriter();
```

```
PrintWriter pw = new PrintWriter(sw);

throwable.printStackTrace(pw);

String stacktrace = sw.toString();
```

除此之外，我们还希望得到更多信息，如发生崩溃进程和线程的ID和Name，崩溃的时间，软件版本和系统版本等。另外，还可以加入一些自定义的信息，如给外部一个异常时的回调，可以加入一些自定义的信息展示。

```
public interface IErrorDealingListener {

    void executing(Throwable t, String appearanceStr);

}
```

4. Native Crash

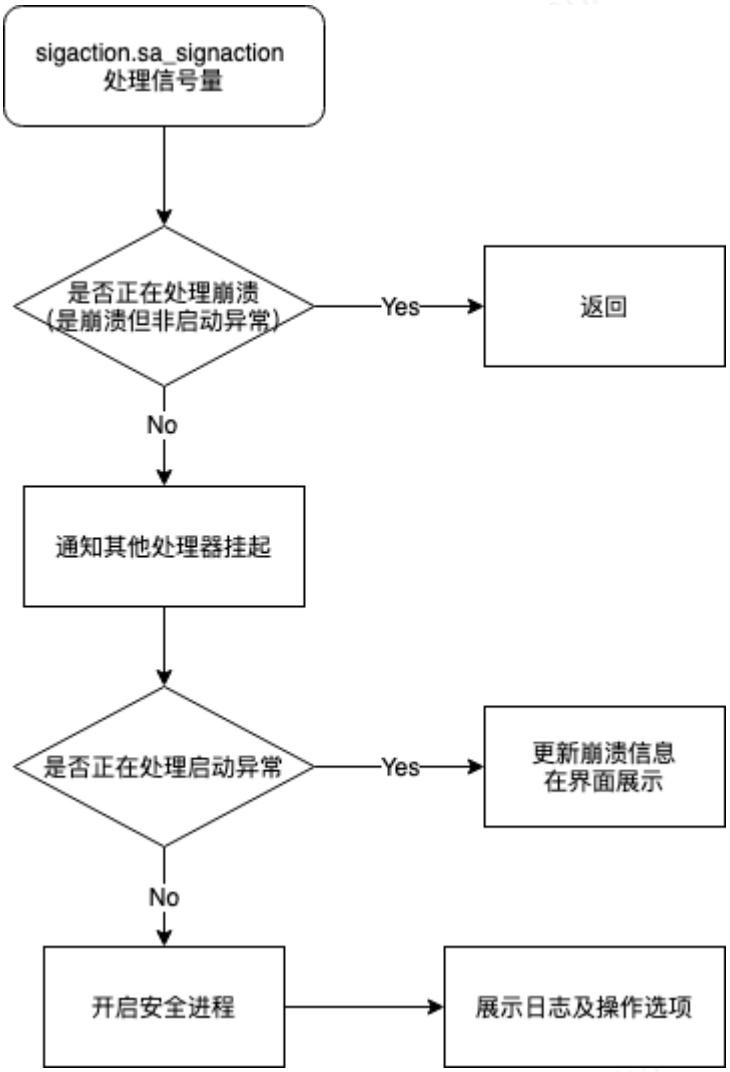
Native崩溃，主要是C层的崩溃，来源于so库的错误。捕获是基于信号量的处理。

4.1 信号量

Native崩溃涉及的信号量，可以参考xCrash的选择，比较全面。包括 SIGABRT、SIGBUS、SIGFPE、SIGILL、SIGSEGV、SIGTRAP、SIGSYS、SIGSTKFLT。

4.2 Native Crash的处理流程

处理全流程类似Java的崩溃：



其中的难点在于信号量的处理，在C层实现，发生崩溃后需通过JNI回调Java的方法。

4.3 Native信号处理实现

相关知识参考：Android平台Native代码的崩溃捕获机制及实现。使用signal库的signaction函数注册handler，可以监测信号量的到来。

还有几个要点：利用alarm函数防止死锁和zombie进程，注册前先设置额外的栈空间，合理地安排旧的处理器，防止覆盖原有的处理。

4.4 Native崩溃信息

经调研，此部分涉及的东西较多，也比较容易踩坑。光从signal库收集到信号时的信息，只能得到信号量、错误码等信息，对分析崩溃原因帮助不大。要获取调用栈还需要费一番功夫。

```
/*信号处理函数*/

void (*sa_sigaction)(const int code, siginfo_t *const si, void * const sc)

siginfo_t {

  int     si_signo;    /* Signal number 信号量 */

  int     si_errno;    /* An errno value */

  int     si_code;     /* Signal code 错误码 */

  }
```

4.4.1 子进程与子线程

先要创建子进程和子线程，保证dump过程顺利完成并回调Java层，用unistd库的fork方法和pthread库的pthread_create方法分别可实现。

4.4.2 获取pc值

即程序的绝对地址。信号处理函数第三个参数sc是uc_mcontext的结构体，这个结构体的定义是平台相关，需要区分不同平台处理。对应的pc值：

- x86-64架构：uc_mcontext.gregs[REG_RIP]
- arm架构：uc_mcontext.arm_pc

4.4.3 函数名称

获取共享库名称和相对偏移地址，以获取问题函数名称。考虑使用dlfcn库的dladdr函数。

```
int dladdr(void *addr, Dl_info *info);

typedef struct {

  const char *dli_fname;  /* Pathname of shared object that contains address */

  void       *dli_fbase;      /* Base address at which shared object is loaded */

  const char *dli_sname;  /* Name of symbol whose definition overlaps addr */

  void       *dli_saddr;      /* Exact address of symbol named in dli_sname */

} Dl_info;
```

4.4.4 调用栈信息

若要进一步获取调用栈信息，需要利用/proc/self/maps 读取共享库的地址范围，从而确定函数对应的栈帧，进一步得到其母函数的栈帧，以此类推可以得到全部调用栈。Android 5.0以上可以使用libunwind库来实现。

若要更进一步获取对应Java层的调用栈，可以考虑从C层把线程名称传回到Java层，在Java层找到对应的线程，并dump出堆栈信息。

5. ANR

ANR问题，经过调研，也是通过处理Native层的信号量来实现捕获。处理流程可以参考Native崩溃。

5.1 ANR信号量处理

ANR问题处理参考：ANR信号拦截与处理。处理的信号量为SIGQUIT，实现上跟Native的崩溃有一个区别。注册信号处理器之前，必须解除系统的信号量屏蔽，用的是signal库的pthread_sigmask函数。

5.2 ANR信息获取

ANR信息都由系统写入了/data/anr/traces.txt文件，文件包含信息较多，我们没必要都导入界面展示。简单来说，我们可能需要获取ANR的发生时间，对应的界面以及近期的Logcat信息，这些信息完全可以回调Java层来获取。

6. 应急处理操作

6.1 系统默认处理

保留系统默认弹框的处理，对于Java崩溃很好实现，只需要保留系统的Handler对象，选择该操作后执行。Native Crash和ANR不太好实现，可以默认处理为退出应用。

6.2 清除数据

清除应用数据可考虑清空特定目录，然后退出应用。参考：https://www.cnblogs.com/Jason-Jan/p/7908477.html

参考源码退出应用的代码：

```
android.os.Process.killProcess(android.os.Process.myPid());

System.exit(10);
```

6.3 兜底开关

兜底功能酌情使用。全局开关存储在SP文件下，开启兜底开关后清空数据就能恢复。可扩展项，这个兜底开关可以做成后台配置。

6.4 自定义逻辑

在操作列表加入类似如下的实现对象：

```
public interface IOperation {

   String operationName();

   void execute(Context context, @ErrorType int type);

}
```

附 开发过程遇到的问题及解决

问题1: 初始化必须在Application的attachBaseContext回调中进行，先于ContentProvider初始化，不影响Firebase上报。但该回调中的Context对象，getApplicationContext为空，导致某些框架无法初始化。

存储框架无法初始化，可以修改存储框架的初始化，兼容Context.getApplicationContext为空的情况。

问题2: Java崩溃后，AMS处理完崩溃回调后，就会在finally代码块必定执行杀死进程的代码，导致处理崩溃时的异步操作失败，如bindService。

可以尽量改为同步操作。不建议阻塞线程来保持住进程，可能导致未知的问题。可以把一些异步操作提前到初始化时进行。

问题3: 自定义的后续操作，支持在指定进程进行。需要进行跨进程通信。

把后续操作的代码块，定义为AIDL接口的实现，可进行IPC传递。

问题4: 进程间无法传递异常，异常都会被Binder捕获，然后仅传递RemoteException。

进程间只传递异常信息，崩溃进程在被杀前，需要获取必要的异常信息。

问题5: 用代码强行杀死进程时，若进程有Activity没有正常关闭，binder会启动讣告机制，由AMS重启进程并重启Activity，导致关闭应用失败。

关闭应用前，需要保证清空Activity栈，即所有Activity都正常finish，并且阻止新的Activity被打开。