AddressSanitizer

AddressSanitizer (ASan) 是一种基于编译器的快速检测工具，用于检测原生代码中的内存错误。它与 Valgrind（Memcheck 工具）相差无几，不同之处在于，ASan：

- + 检测堆栈和全局对象是否有溢出
- - 不检测未初始化的读取和内存泄露
- + 速度更快（Valgrind 的 20-100x 与其相比，慢 2-3 倍）
- + 内存占用空间较少

本文档介绍了如何使用 AddressSanitizer 构建和运行 Android 平台的组成部分。如果您希望利用 AddressSanitizer 构建独立的（即 SDK/NDK）应用，请改为参阅 AddressSanitizerOnAndroid (https://github.com/google/sanitizers/wiki/AddressSanitizerOnAndroid) 公共项目网站。

AddressSanitizer 包括一个编译器 (**external/clang**) 和一个运行时库 (**external/compiler-rt/lib/asan**)。

注意：请立即使用当前的 master 分支获取对 SANITIZE_TARGET (#sanitize_target) 功能的访问权限，并获取利用 AddressSanitizer 构建整个 Android 平台的能力。否则，您将只能使用 **LOCAL_SANITIZE**。

使用 Clang 构建

要构建使用 ASan 进行测试的二进制文件，第一步是要确保您的代码是使用 Clang 进行构建的。默认情况下，系统会在 master 分支上完成这一步骤，因此您无需执行任何操作。如果您认为自己要测试的模块是使用 GCC 构建的，则可以向构建规则中添加 **LOCAL_CLANG:=true**，从而切换至 Clang。Clang 可以发现 GCC 遗漏的代码错误。

使用 AddressSanitizer 构建可执行文件

将 **LOCAL_SANITIZE:=address** 添加到可执行文件的构建规则中。

```
LOCAL_SANITIZE:=address
```

检测到错误时，ASan 会向标准输出文件和 **logcat** 发送一份详细报告，然后让相应进程崩溃。

使用 AddressSanitizer 构建共享库

根据 ASan 的工作原理，未采用 ASan 构建的可执行文件将无法使用采用 ASan 构建的库。

注意：如果 ASan 库加载到错误的进程，则在运行时，您会看到开头为 **_asan** 或 **_sanitizer** 的未解决错误符号信息。

要清理多个可执行文件（并非所有这些可执行文件都是使用 ASan 构建的）使用的共享库，您需要获取该库的 2 个副本。要获取副本，建议您针对相应的模块向 **Android.mk** 中添加以下内容：

```
LOCAL_SANITIZE:=address
LOCAL_MODULE_RELATIVE_PATH := asan
```

这样一来，系统会将库放置到 **/system/lib/asan**（而非 **/system/lib**）中。然后，使用以下方法运行您的可执行文件：**LD_LIBRARY_PATH=/system/lib/asan**

对于系统守护程序，将以下内容添加到 **/init.rc** 或 **/init.$device$.rc** 的相应部分。

```
setenv LD_LIBRARY_PATH /system/lib/asan
```

警告：**LOCAL_MODULE_RELATIVE_PATH** 设置会将您的库移动至 **/system/lib/asan**，这意味着，如果从头开始重写并重新构建，则会导致库从 **/system/lib** 中缺失，且很可能会产生无法启动的映像。这是当前构建系统存在的一个令人遗憾的限制。不要重写；而是进行 **make -j $N** 和 **adb sync**。

当通过读取 **/proc/$PID/maps** 显示相应进程时，验证其使用的是否为来自 **/system/lib/asan** 的库。如果不是，您可能需要停用 SELinux，如下所示：

```
$ adb root
$ adb shell setenforce 0
# restart the process with adb shell kill $PID
# if it is a system service, or may be adb shell stop; adb shell start.
```

更出色的堆栈跟踪

AddressSanitizer 使用基于框架指针的快速展开程序，针对程序中的每个内存分配和取消分配事件记录堆栈跟踪。大部分 Android 平台都未使用框架指针进行构建。因此，您通常仅会获得 1 个或 2 个有意义的框架。要解决此问题，请使用 ASan（推荐）或以下方法重新构建库：

```
LOCAL_CFLAGS:=-fno-omit-frame-pointer
```

```
LOCAL_ARM_MODE:=arm
```

或者在进程环境中设置 **ASAN_OPTIONS=fast_unwind_on_malloc=0**。后者可能对 CPU 要求极高，具体取决于负载。

符号化

最初，ASan 报告中包含对二进制文件和共享库中的偏移量的引用。您可以通过以下两种方法获取源文件和行信息：

- 确保 **/system/bin** 中有 llvm-symbolizer 二进制文件。Llvm-symbolizer 在 **third_party/llvm/tools/llvm-symbolizer** 的源文件中构建
- 通过 **external/compiler-rt/lib/asan/scripts/symbolize.py** 脚本过滤报告。

由于可以使用主机上的符号化库，因此第二种方法可以提供更多数据（即 file:line 位置）。

应用中的 AddressSanitizer

AddressSanitizer 无法了解 Java 代码的情况，但可以检测 JNI 库中的错误。为此，您需要使用 ASan 构建可执行文件，在此情况下是 **/system/bin/app_process(*32*|*64*)**。这样一来，便可以同时启用设备上所有应用中的 ASan，这会给设备带来一点压力，但 2GB RAM 设备可以从容处理任何情况。

向 **frameworks/base/cmds/app_process** 中的 app_process 构建规则添加常规 **LOCAL_SANITIZE:=address**。暂时忽略同一文件中的 **app_process__asan** 目标（如果当您阅读该文档时仍存在于文件中）。修改 **system/core/rootdir/init.zygote(*32*|*64*).rc** 中的 Zygote 记录，以添加以下行：

```
setenv LD_LIBRARY_PATH /system/lib/asan:/system/lib
setenv ASAN_OPTIONS
allow_user_segv_handler=true
```

构建，进行 adb 同步，fastboot 刷写启动，然后重新启动。

使用 wrap 属性

上一部分中的方法将 AddressSanitizer 放置到了系统的每个应用中（实际上是放置到了 Zygote 进程的每个子级元素中）。可以仅运行一个（或几个）具有 ASan 的应用，从而占用部分内存空间，使应用启动速度变慢。

为实现这一目标，您可以借助“wrap”属性（用于在 Valgrind 下运行应用的同一属性）启动应用。下面是在 ASan 下运行 Gmail 应用的示例：

```
$ adb root
$ adb shell setenforce 0  # disable SELinux
$ adb shell setprop wrap.com.google.android.gm "asanwrapper"
```

在这种情况下，asanwrapper 会将 **/system/bin/app_process** 重写至 **/system/bin/asan/app_process**（使用 AddressSanitizer 构建）。此外，它还会在动态库搜索路径的开头处添加 **/system/lib/asan**。这样一来，借助 asanwrapper 运行应用时，与 **/system/lib** 中的普通库相比，系统更倾向于使用 **/system/lib/asan** 中用 ASan 进行测试的库。

同样，如果发现错误，应用会崩溃，且系统会将报告记录到日志中。

SANITIZE_TARGET

master 分支支持立即使用 AddressSanitizer 构建整个 Android 平台。

在同一构建树中运行以下命令。

```
$ make -j42
$ SANITIZE_TARGET=address make -j42
```

在此模式下，**userdata.img** 中包含其他库，必须也刷写到设备上。请使用以下命令行：

```
$ fastboot flash userdata && fastboot flashall
```

写入时，现今的 Nexus 和 Pixel 设备会启动到该模式中的界面。

其工作原理是构建两组共享库：**/system/lib** 中的常规库（第一次 make 调用），**/data/asan/lib** 中使用 ASan 进行测试的库（第二次 make 调用）。第二次构建中的可执行文件会覆盖第一次构建中的可执行文件。通过使用 PT_INTERP 中的“/system/bin/linker_asan”，使用 ASan 进行测试的可执行文件会获得一个不同的库搜索路径，该路径会在 **/system/lib** 前添加 **/data/asan/lib**。

如果 **$SANITIZE_TARGET** 值已更改，则构建系统会重写中间对象目录。这样一来，系统便会强制重新构建所有目标，同时保留 **/system/lib** 下已安装的二进制文件。

以下目标不能使用 ASan 进行构建：

- 静态关联的可执行文件。
- **LOCAL_CLANG:=false** 目标

- 不会针对 `SANITIZE_TARGET=address` 进行 ASan 操作的 `LOCAL_SANITIZE:=false`

在 SANITIZE_TARGET 构建中，系统会跳过此类可执行文件，且会将第一次 make 调用中的版本留在 `/system/bin` 中。

此类库只是未使用 ASan 进行构建，但它们仍然可以包含一些来自自己依赖的静态库的 ASan 代码。

支持文档

AddressSanitizerOnAndroid (https://github.com/google/sanitizers/wiki/AddressSanitizerOnAndroid) 公共项目网站

AddressSanitizer 和 Chromium (https://www.chromium.org/developers/testing/addresssanitizer)

其他 Google 清理程序 (https://github.com/google/sanitizers)