Android Developers

独立工具链

您可以独立使用 Android NDK 附带的工具链，或将其作为插件与现有 IDE 结合使用。如果您已有自己的构建系统，且仅需要调用交叉编译器功能以将对构建系统的支持添加到 Android，则这种灵活性非常有用。

一个典型用例是在 `CC` 环境变量中调用一个需要交叉编译器的开源库的 configure 脚本。

> 注：本页假设您非常了解编译、链接和低级架构。此外，对于大多数用例来说本页介绍的技术并不是必需的。在大多数情况下，我们建议您放弃使用独立工具链，而是坚持使用 NDK 构建系统。

本页内容

选择您的工具链

首先，您需要决定您的独立工具链将以哪个处理架构为目标。每个架构均对应不同的工具链名称，如表 1 所示。

表 **1.** `APP_ABI` 不同指令集的设置。

架构	工具链名称
基于 ARM	`arm-linux-androideabi-<gcc-version>`
基于 x86	`x86-<gcc-version>`
基于 MIPS	`mipsel-linux-android-<gcc-version>`
基于 ARM64	`aarch64-linux-android-<gcc-version>`
基于 X86-64	`x86_64-<gcc-version>`
基于 MIPS64	`mips64el-linux-android--<gcc-version>`

选择您的 Sysroot

接下来您需要做的是定义您的 *sysroot*（sysroot 是一个包含针对您的目标的系统标头和库的目录）。如需定义 sysroot，您必须知道原生支持的目标 Android API 级别；可用的原生 API 因 Android API 级别而异。

针对相应 Android API 级别 (https://developer.android.com/guide/topics/manifest/uses-sdk-element.html)的原生 API 位于 `$NDK/platforms/` 下；每个 API 级别目录又包含针对各种 CPU 和架构的子目录。以下示例显示如何针对 ARM 架构为以 Android 5.0（API 级别 21）为目标的构建定义 *sysroot*：

```
SYSROOT=$NDK/platforms/android-21/arch-arm
```

如需了解有关 Android API 级别及其支持的相应原生 API 的详细信息，请参阅 Android NDK 原生 API (https://developer.android.com/ndk/guides/stable_apis.html)。

调用编译器

调用编译器的方式有两种。其中一个方法很简单，大部分事务都由构建系统完成。另一种方法则较为复杂，但提供更多灵活性。

简单方法

最简单的构建方式是直接从命令行调用相应的编译器，使用 `--sysroot` 选项指明您的目标平台的系统文件位置。例如：

```
export CC="$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/ \
linux-x86/bin/arm-linux-androideabi-gcc-4.8 --sysroot=$SYSROOT"
$CC -o foo.o -c foo.c
```

尽管此方法很简单，但它缺少灵活性：它不允许您使用任何 C++ STL（STLport、libc++ 或 GNU libstdc++）。它也不支持例外或 RTTI。

对于 Clang，您需要执行两个额外的步骤：

1. 为目标架构添加适合的 `-target`，如表 2 所示。

   表 **2.** 架构和对应的 `-target` 的值。

架构	值
armeabi	`-target armv5te-none-linux-androideabi`
armeabi-v7a	`-target armv7-none-linux-androideabi`
arm64-v8a	`-target aarch64-none-linux-android`
x86	`-target i686-none-linux-android`
x86_64	`-target x86_64-none-linux-android`
mips	`-target mipsel-none-linux-android`

2. 通过添加 `-gcc-toolchain` 选项添加汇编程序和链接器支持，如以下示例所示：

```
-gcc-toolchain $NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/linux-x86_64
```

最终，一个使用 Clang 进行编译的命令可能如下所示：

```
export CC="$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/ \
linux-x86/bin/arm-linux-androideabi-gcc-4.8 --sysroot=$SYSROOT" -target \
armv7-none-linux-androideabi \
-gcc-toolchain $NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/linux-x86_64"
$CC -o foo.o -c foo.c
```

高级方法

NDK 提供 `make-standalone-toolchain.sh` shell 脚本以允许您从命令行执行定制的工具链安装。与简单方法 (#sm)中所述的程序相比，此方法为您提供更多灵活性。

脚本位于 `$NDK/build/tools/` 目录中，其中 `$NDK` 是 NDK 的安装根目录。下面展示了使用此脚本的示例：

```
$NDK/build/tools/make-standalone-toolchain.sh \
--arch=arm --platform=android-21 --install-dir=/tmp/my-android-toolchain
```

此命令创建一个名为 `/tmp/my-android-toolchain/` 的目录，包含一个 `android-21/arch-arm` sysroot 的副本，以及适用于 32 位 ARM 架构的工具链二进制文件的副本。

请注意，工具链二进制文件不依赖或包含主机特有的路径，换句话说，您可以将它们安装在任意位置中，甚至

移动它们（如果需要）。

默认情况下，构建系统使用 32 位、基于 ARM 的 GCC 4.8 工具链。不过，您可以通过将 `--arch=<toolchain>` 指定为选项来指定一个不同的值。表 3 显示将用于其他工具链的值：

表 **3.** 工具链和对应的值，使用 `--arch`。

工具链	值
mips64 编译器	`--arch=mips64`
mips GCC 4.8 编译器	`--arch=mips`
x86 GCC 4.8 编译器	`--arch=x86`
x86_64 GCC 4.8 编译器	`--arch=x86_64`
mips GCC 4.8 编译器	`--arch=mips`

或者，您可以使用 `--toolchain=<toolchain>` 选项。表 4 显示您可以为 `<toolchain>` 指定的值：

表 **4.** 工具链和对应的值，使用 `--toolchain`.

工具链	值
arm	• `--toolchain=arm-linux-androideabi-4.8` • `--toolchain=arm-linux-androideabi-4.9` • `--toolchain=arm-linux-android-clang3.5` • `--toolchain=arm-linux-android-clang3.6`
x86	• `--toolchain=x86-linux-android-4.8` • `--toolchain=x86-linux-android-4.9` • `--toolchain=x86-linux-android-clang3.5` • `--toolchain=x86-linux-android-clang3.6`
mips	• `--toolchain=mips-linux-android-4.8` • `--toolchain=mips-linux-android-4.9` • `--toolchain=mips-linux-android-clang3.5` • `--toolchain=mips-linux-android-clang3.6`

arm64	• `--toolchain=aarch64-linux-android-4.9` • `--toolchain=aarch64-linux-android-clang3.5` • `--toolchain=aarch64-linux-android-clang3.6`
x86_64	• `--toolchain=x86_64-linux-android-4.9` • `--toolchain=x86_64-linux-android-clang3.5` • `--toolchain=x86_64-linux-android-clang3.6`
mips64	• `--toolchain=mips64el-linux-android-4.9` • `--toolchain=mips64el-linux-android-clang3.5` • `--toolchain=mips64el-linux-android-clang3.6`

> 注：表 4 并不是一个详尽的列表。其他组合可能也有效，但未经验证。

您也可以使用以下两种方法之一复制 Clang/LLVM 3.6：您可以将 `-clang3.6` 附加到 `--toolchain` 选项，以便 `--toolchain` 选项看上去如以下示例所示：

```
--toolchain=arm-linux-androideabi-clang3.6
```

您也可以在命令行上添加 `-llvm-version=3.6` 作为单独的选项。

> 注：无需指定特定版本，您也可以使用 `<version>`，其默认使用可用的 Clang 最高版本。

默认情况下，构建系统针对 32 位主机工具链进行构建。您可以指定一个 64 位主机链代替它。表 5 显示针对不同平台将与 `-system` 一起使用的值。

表 **5.** 主机工具链和对应的值，使用 `-system`。

主机工具链	值
64 位 Linux	`-system=linux-x86_64`
64 位 MacOSX	`-system=darwin-x86_64`
64 位 Windows	`-system=windows-x86_64`

如需了解有关指定 64 或 32 位指令主机工具链的详细信息，请参阅 64 位和 32 位工具链 (https://developer.android.com/ndk/guides/ndk-build.html#6432)。

您可以指定 `--stl=stlport` 以复制 `libstlport`，而不是使用默认的 `libgnustl`。如果您执行此操作并想链

接共享库，则必须以显式方式使用 `-lstlport_shared`。此要求与必须为 GNU `libstdc++` 使用 `-lgnustl_shared` 相似。

同样，您可以指定 `--stl=libc++` 复制 LLVM libc++ 标头和库。如需链接共享库，您必须以显式方式使用 -lc++_shared。

您可以直接进行这些设置，如以下示例所示：

```
export PATH=/tmp/my-android-toolchain/bin:$PATH
export CC=arm-linux-androideabi-gcc   # or export CC=clang
export CXX=arm-linux-androideabi-g++  # or export CXX=clang++
```

请注意，如果您忽略 `-install-dir` 选项，则 `make-standalone-toolchain.sh` shell 脚本在 `tmp/ndk/<toolchain-name>.tar.bz2` 中创建一个 tarball。此 tarball 让您可以轻松存档和重新分发二进制文件。

此独立工具链还提供了一个额外优势，即：它包含一个 C++ STL 库的工作中副本以及工作中例外和 RTTI 支持。

如需了解更多选项和详细信息，请使用 `--help`。

使用 Clang

您可以使用 `--llvm-version=<version>` 选项在独立安装中安装 Clang 二进制文件。`<version>` 是 LLVM/Clang 版本号，如 `3.5` 或 `3.6`。例如：

```
build/tools/make-standalone-toolchain.sh \
--install-dir=/tmp/mydir \
--toolchain=arm-linux-androideabi-4.8 \
--llvm-version=3.6
```

请注意，Clang 二进制文件与 GCC 二进制文件一起复制，因为它们依赖于相同的汇编程序、链接器、标头、库以及 C++ STL 实现。

此操作也将在 `<install-dir>/bin/@` 下安装两个名为 `clang` 和 `clang++` 的脚本。这些脚本使用默认目标架构标志调用真实的 `clang` 二进制文件。换句话说，它们无需任何修改就能运行，并且您只需设置指向它们的 `CC` `CXX` 环境变量就可以在您自己的构建中使用它们。

调用 Clang

在一个使用 `llvm-version=3.6` 构建的 ARM 独立安装中，在 Unix 系统上调用 Clang (http://clang.llvm.org/) 采用

单行形式。例如：

```
`dirname $0`/clang36 -target armv5te-none-linux-androideabi "$@"
```

`clang++` 以相同方式调用 `clang++31`。

Clang 以 ARM 为目标

针对 ARM 进行构建时，Clang 基于是否存在 `-march=armv7-a` 和/或 `-mthumb` 选项更改目标：

表 **5.** 可指定的 `-march` 值及其生成的目标。

`-march` 值	生成的目标
`-march=armv7-a`	`armv7-none-linux-androideabi`
`-mthumb`	`thumb-none-linux-androideabi`
`-march=armv7-a` 和 `-mthumb`	`thumbv7-none-linux-androideabi`

如果您希望，您也可以替换为您自己的 `-target`。

`-gcc-toolchain` 选项不是必需的，因为在独立软件包中，Clang 在预定义的相对位置中查找 `as` 和 `ld`。

`clang` 和 `clang++` 应能够轻松替换 makefile 中的 `gcc` 和 `g++`。如有疑问，添加下列选项以验证他们是否正确运行。

- `-v`，用于转储与编译器驱动程序问题有关的命令
- `-###`，用于转储命令行选项，包括以隐式方式预定义的选项。
- `-x c < /dev/null -dM -E`，用于转储预定义的预处理器定义
- `-save-temps`，用于比较 `*.i` 或 `*.ii` 预处理文件。

如需了解有关 Clang 的详细信息，请参阅 http://clang.llvm.org/ (http://clang.llvm.org/)，尤其是 GCC 兼容性部分。

ABI 兼容性

默认情况下，ARM 工具链生成的机器代码应与官方 Android `armeabi` ABI (https://developer.android.com/ndk/guides/abis.html) 兼容。

我们建议使用 `-mthumb` 编译器标志以强制生成 16 位 Thumb-1 指令（默认成为 32 位 ARM 指令）。

如果您要以 armeabi-v7a ABI 为目标，则必须设置下列标志：

```
CFLAGS= -march=armv7-a -mfloat-abi=softfp -mfpu=vfpv3-d16
```

第一个标志启用 Thumb-2 指令。第二个标志启用硬件 FPU 指令，同时确保系统在核心寄存器中传递浮点参数，这对于 ABI 兼容性至关重要。

> 注：在 r9b 以前的 NDK 版本中，请勿单独使用这些标志。您必须同时设置所有标志或一个都不设置。否则，可能导致无法预测的行为和崩溃。

如需使用 NEON 指令，您必须更改 `-mfpu` 编译器标志：

```
CFLAGS= -march=armv7-a -mfloat-abi=softfp -mfpu=neon
```

请注意，按照 ARM 规范，此设置强制使用 `VFPv3-D32`。

另外，确保向链接器提供以下两个标志：

```
LDFLAGS= -march=armv7-a -Wl,--fix-cortex-a8
```

第一个标志指示链接器选取为 armv7-a 定制的 `libgcc.a`、`libgcov.a` 和 `crt*.o`。在某些 Cortex-A8 实现中，需要第二个标志作为 CPU 错误的解决方法。

自 NDK 版本 r9b 开始，获取或返回双精度值或浮点值的所有 Android 原生 API 都具有用于 ARM 的 `attribute((pcs("aapcs")))`。这让您可以在 `-mhard-float`（其表示 `-mfloat-abi=hard`）中编译用户代码，并仍与符合 softfp ABI 的 Android 原生 API 关联。如需了解有关此操作的详细信息，请参阅 `$NDK/tests/device/hard-float/jni/Android.mk` 中的注释。

如果您要在 x86 上使用 NEON intrinsics，构建系统可以使用与标准 ARM NEON 内联函数标头具有相同名称 `arm_neon.h` 的特殊 C/C++ 语言标头将它们转换为原生 x86 SSE 内联函数。

默认情况下，x86 ABI 最大支持 SIMD 的 SSSE3，且标头涵盖 NEON 函数的 93% 左右（1869 个（总数为 2009 个））。

如果以 MIPS ABI 为目标，您不必使用任何特定的编译器标志。

如需有关 ABI 支持的详细信息，请参阅 x86 支持 (https://developer.android.com/ndk/guides/x86.html)。

警告和限制

Windows 支持

Windows 二进制文件不依赖于 Cygwin。这种独立性让它们的运行速度更快。不过，代价是它们不理解 Cygwin 路径规范，如 `cygdrive/c/foo/bar`，但可以理解 `C:/foo/bar`。

NDK 构建系统确保所有从 Cygwin 传递到编译器的路径可自动转换，同时管理其他复杂性。如果您有自定义构建系统，您可能需要自己解决这些复杂性。

如需有关为 Cygwin/MSys 贡献支持的信息，请访问 android-ndk 论坛 (https://groups.google.com/forum/#!forum/android-ndk)。

wchar_t 支持

Android 平台在 Android 2.3（API 级别 9）之前并没有真正地支持 `wchar_t`。这种情况产生多个结果：

- 如果您的目标平台为 Android 2.3 或更高版本，则 `wchar_t` 的大小为 4 字节，且大多数 `wide-char` 函数可在 C 库中获取（多字节编码/解码函数和 `wsprintf`/`wsscanf` 除外）。
- 如果您以任意较低的 API 级别为目标，则 `wchar_t` 的大小为 1 字节，且任何 wide-char 函数均无法运行。

我们建议您不要依赖 `wchar_t` 类型，并改用更好的表示形式。 Android 中提供的此支持目的只是为了帮助您迁移现有代码。

例外、RTTI 和 STL

默认情况下，工具链二进制文件支持 C++ 例外和 RTTI。在构建源时，如需停用 C++ 例外和 RTTI（例如，为了生成更轻量的机器代码），则使用 `-fno-exceptions` 和 `-fno-rtti`。

如需将这些功能与 GNU libstdc++ 结合使用，您必须以显式方式与 libsupc++ 进行关联。为此，链接二进制文件时请使用 `-lsupc++`。例如：

```
arm-linux-androideabi-g++ .... -lsupc++
```

如果使用 STLport 或 libc++ 库，那么您不需要执行此操作。

C++ STL 支持

独立工具链包含一个 C++ 标准模板库实现的副本。此实现适用于 GNU libstdc++、STLport 或 libc++，具体取决于您为前面所述的 `--stl=<name>` 选项所指定的内容。如需使用这个 STL 实现，您需要将您的项目与正确的库进行关联：

- 使用 `-lstdc++` 以链接任意实现的静态库版本。这样做可确保将所有必需的 C++ STL 代码添加到您最终的二进制文件。如果您仅生成一个共享库或可执行文件，则此方法为理想之选。

  这是我们建议的方法。

- 替代方法是使用 `-lgnustl_shared` 链接 GNU `libstdc++` 的共享库版本。如果您使用此选项，您还必须确保将 `libgnustl_shared.so` 复制到您的设备以正确加载您的代码。表 6 显示对于每个工具链类型此文件的位置。

  > 注：GNU libstdc++ 依据 GPLv3 许可证授权，具有一个链接例外。如果您不能符合其要求，则无法在您的项目中重新分发共享库。

- 使用 `-lstlport_shared` 链接 STLport 的共享库版本。如果您这样做，您需要确保您也将 `libstlport_shared.so` 复制到您的设备以正确加载您的代码。表 6 显示对于每个工具链此文件的位置：

  Table 6. 可指定的 `-march` 值及其生成的目标。

工具链	位置
arm	`$TOOLCHAIN/arm-linux-androideabi/lib/`
arm64	`$TOOLCHAIN/aarch64-linux-android/lib/`
x86	`$TOOLCHAIN/i686-linux-android/lib/`
x86_64	`$TOOLCHAIN/x86_64-linux-android/lib/`
mips	`$TOOLCHAIN/mipsel-linux-android/lib/`
mips64	`$TOOLCHAIN/mips64el-linux-android/lib/`

> 注：如果您的项目包含多个共享库或可执行文件，那么，您必须链接一个共享库 STL 实现。否则，此构建系统不会定义特定的全局唯一性，从而导致不可预测的运行时行为。此行为可能包括崩溃和未能正确捕捉异常。

这些库不只是称为 `libstdc++.so` 的原因是此名称在运行时与系统自身的最小 C++ 运行时冲突。为此，构

建系统强制为 GNU ELF 库指定一个新名称。静态库没有这个问题。