

Android.mk 文档规范

Android.mk 编译文件是用来向 Android NDK 描述你的 C,C++源代码文件的, 这篇文档描述了它的语法。在阅读下面的内容之前, 假定你已经阅读了 docs/OVERVIEW.TXT 文件, 了解了它们的角色和用途。

一、概述

一个 Android.mk file 用来向编译系统描述你的源代码。具体来说:

(1) 该文件是 GNU Makefile 的一小部分, 会被编译系统解析一次或更多次的 build 系统。因此, 您应尽量减少您声明的变量, 不要认为某些变量在解析过程中不会被定义。

(2) 这个文件的语法允许把你的源代码组织成模块, 一个模块属下列类型之一:

- 1) 静态库
- 2) 共享库

且只有共享库将被安装/复制到您的应用软件包, 虽然静态库能被用于生成共享库。

可以在每一个 Android.mk file 中定义一个或多个模块, 你也可以在几个模块中使用同一个源代码文件。

编译系统为你处理许多细节问题。例如, 你不需要在你的 Android.mk 中列出头文件和依赖文件。NDK 编译系统将会为你自动处理这些问题。这也意味着, 在升级 NDK 后, 你应该得到新的 toolchain/platform 支持, 而且不需要改变你的 Android.mk 文件。

注意, 这个语法同公开发布的 Android 平台的开源代码很接近, 然而编译系统实现他们的方式却是不同的, 这是故意这样设计的, 可以让程序开发人员重用外部库的源代码更容易。

在描述语法细节之前, 咱们来看一个简单的"hello world"的例子, 比如, 下面的文件:

sources/helloworld/helloworld.c
sources/helloworld/Android.mk

'helloworld.c'是一个 JNI 共享库, 实现返回"hello world"字符串的原生方法。相应的 Android.mk 文件会象下面这样:

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

include $(CLEAR_VARS)

LOCAL_MODULE:= helloworld

LOCAL_SRC_FILES := helloworld.c

include $(BUILD_SHARED_LIBRARY)
```

解释一下这几行代码:

LOCAL_PATH := \$(call my-dir)

一个 Android.mk file 首先必须定义好 LOCAL_PATH 变量。它用于在开发树中查找源文件。在这个例子中, 宏函数 'my-dir', 由编译系统提供, 用于返回当前路径(即包含 Android.mk file 文件的目录)。

include \$(CLEAR_VARS)

CLEAR_VARS 由编译系统提供(可以在 android 安装目录下的/build/core/config.mk 文件看到其定义, 为 CLEAR_VARS:= \$(BUILD_SYSTEM)/clear_vars.mk), 指定让 GNU MAKEFILE 为你清除许多 LOCAL_XXX 变量(例如 LOCAL_MODULE, LOCAL_SRC_FILES,

LOCAL_STATIC_LIBRARIES, 等等...), 除 LOCAL_PATH。这是必要的, 因为所有的编译控制文件都在同一个 GNU MAKE 执行环境中, 所有的变量都是全局的。

LOCAL_MODULE := helloworld

LOCAL_MODULE 变量必须定义, 以标识你在 Android.mk 文件中描述的每个模块。名称必须是唯一的, 而且不包含任何空格。注意编译系统会自动产生合适的前缀和后缀, 换句话说, 一个被命名为 'foo' 的共享库模块, 将会生成 'libfoo.so' 文件。注意: 如果把库命名为 'libhelloworld', 编译系统将不会添加任何的 lib 前缀, 也会生成 libhelloworld.so, 这是为了支持来源于 Android 平台的源代码的 Android.mk 文件。

LOCAL_SRC_FILES := helloworld.c

LOCAL_SRC_FILES 变量必须包含将要编译打包进模块中的 C 或 C++ 源代码文件。不用在这里列出头文件和包含文件, 编译系统将会自动找出依赖型的文件。

【注意, 默认的 C++ 源码文件的扩展名是 '.cpp'。指定一个不同的扩展名也是可能的, 只要定义 LOCAL_DEFAULT_CPP_EXTENSION 变量, 不要忘记开始的小圆点(也就是定义为 '.cxx', 而不是 'cxx')】

include \$(BUILD_SHARED_LIBRARY)

BUILD_SHARED_LIBRARY 是编译系统提供的变量, 指向一个 GNU Makefile 脚本(应该就是在 build/core 目录下的 shared_library.mk), 负责收集自从上次调用 'include \$(CLEAR_VARS)' 以来, 定义在 LOCAL_XXX 变量中的所有信息, 并且决定编译什么, 如何正确地去。并根据其规则生成静态库。同理对于静态库。

在 sources/samples 目录下有更复杂一点的例子, 写有注释的 Android.mk 文件。

二、参考

这是一份你应该在 Android.mk 中依赖或定义的变量列表, 可以定义其他变量为自己使用, 但是 NDK 编译系统保留下列变量名:

- 以 LOCAL_ 开头的名字 (例如 LOCAL_MODULE)
- 以 PRIVATE_, NDK_ 或 APP_ 开头的名字 (内部使用)
- 小写名字 (内部使用, 例如 'my-dir')

如果为了方便在 Android.mk 中定义自己的变量, 建议使用 MY_ 前缀, 一个小例子:

```
MY_SOURCES := foo.c

ifneq ($(MY_CONFIG_BAR),)
    MY_SOURCES += bar.c
endif

LOCAL_SRC_FILES += $(MY_SOURCES)
```

1. GNU Make 变量

这些 GNU Make 变量在你的 Android.mk 文件解析之前, 就由编译系统定义好了。注意在某些情况下, NDK 可能分析 Android.mk 几次, 每一次某些变量的定义会有不同。

(1) **CLEAR_VARS:** 指向一个编译脚本, 几乎所有未定义的 LOCAL_XXX 变量都在 "Module-description" 节中列出。必须在开始一个新模块之前包含这个脚本: include \$(CLEAR_VARS)

(2) **BUILD_SHARED_LIBRARY:** 指向编译脚本, 收集所有的在 LOCAL_XXX 变量中提供的信息, 并且决定如何把列出的源代码文件编译成一个共享库。注意, 必须至少在包含这个文件之前定义 LOCAL_MODULE 和 LOCAL_SRC_FILES。使用例子:

```
include $(BUILD_SHARED_LIBRARY)
#这将生成一个名为 lib$(LOCAL_MODULE).so 的文件
```

(3) **BUILD_STATIC_LIBRARY:** 一个 BUILD_SHARED_LIBRARY 变量用于编译一个静态库。静态库不会复制到的 project/packages 中，但是能够用于编译共享库，（看下面描述的 LOCAL_STATIC_LIBRARIES 和 LOCAL_STATIC_WHOLE_LIBRARIES）。使用例子：

```
include $(BUILD_STATIC_LIBRARY)
#注意，这将会生成一个名为 lib$(LOCAL_MODULE).a 的文件
```

(4) **TARGET_ARCH:** 目标 CPU 平台的名称，和 android 开放源码中指定的那样。如果是 arm，表示要生成 ARM 兼容的指令，与 CPU 架构的修订版无关。

(5) **TARGET_PLATFORM:** Android.mk 解析的时候，目标 Android 平台的名称。详情可参考 development/ndk/docs/stable-apis.txt。

android-3 -> Official Android 1.5 system images

android-4 -> Official Android 1.6 system images

android-5 -> Official Android 2.0 system images

(6) **TARGET_ARCH_ABI:** 暂时只支持两个 value，armeabi 和 armeabi-v7a。在现在的版本中一般把这两个值简单的定义为 arm，通过 android 平台内部对它重定义来获得更好的匹配。其他的 ABI 将在以后的 NDK 版本中介绍，它们会有不同的名字。注意所有基于 ARM 的 ABI 都会把 'TARGET_ARCH' 定义成 'arm'，但是会有不同的 'TARGET_ARCH_ABI'。

(7) **TARGET_ABI:** 目标平台和 ABI 的组合，它事实上被定义成 \$(TARGET_PLATFORM)-\$(TARGET_ARCH_ABI)，在想要在真实的设备中针对一个特别的目标系统进行测试时，会有用。在默认的情况下，它会是 'android-3-arm'。

2. 模块描述变量

下面的变量用于向编译系统描述你的模块。你应该定义在 'include \$(CLEAR_VARS)' 和 'include \$(BUILD_XXXX)' 之间。正如前面描写的那样，\$(CLEAR_VARS) 是一个脚本，清除所有这些变量，除非在描述中显式注明。

(1) **LOCAL_PATH:** 这个变量用于给出当前文件的路径。必须在 Android.mk 的开头定义，可以这样使用：

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

这个变量不会被 \$(CLEAR_VARS) 清除，因此每个 Android.mk 只需要定义一次(即使在一个文件中定义了几个模块的情况下)。

(2) **LOCAL_MODULE:** 这是模块的名字，它必须是唯一的，而且不能包含空格。必须在包含任一的 \$(BUILD_XXXX) 脚本之前定义它。模块的名字决定了生成文件的名字。例如，如果一个一个共享库模块的名字是，那么生成文件的名字就是 lib.so。但是，在的 NDK 生成文件中(或者 Android.mk 或者 Application.mk)，应该只涉及(引用)有正常名字的其他模块。

(3) **LOCAL_SRC_FILES:** 这是要编译的源代码文件列表。只要列出要传递给编译器的文件，因为编译系统自动计算依赖。注意源代码文件名称都是相对于 LOCAL_PATH 的，你可以使用路径部分，例如：

```
LOCAL_SRC_FILES := foo.c
                  toto/bar.c
```

【注意：在生成文件中都要使用 UNIX 风格的斜杠(/)。windows 风格的反斜杠不会被正确的处理。】

(4) `LOCAL_CPP_EXTENSION`: 这是一个可选变量,用来指定 C++代码文件的扩展名,默认是'.cpp',但是可以改变它,比如:

```
LOCAL_CPP_EXTENSION := .cxx
```

(5) `LOCAL_C_INCLUDES`: 路径的可选配置,是从根目录开始的所有源文件(C, C++ and Assembly)。比如:

```
LOCAL_C_INCLUDES := sources/foo
```

或者:

```
LOCAL_C_INCLUDES := $(LOCAL_PATH)/../foo
```

需要在任何包含 `LOCAL_CFLAGS`/`LOCAL_CPPFLAGS` 标志之前。

(6) `LOCAL_CFLAGS`: 可选的编译器选项,在编译 C 代码文件的时候使用。这可能是有用的,指定一个附加的包含路径(相对于 NDK 的顶层目录),宏定义,或者编译选项。

注意: 不要在 `Android.mk` 中改变 `optimization/debugging` 级别,只要在 `Application.mk` 中指定合适的信息,就会自动地为你处理这个问题,在调试期间,会让 NDK 自动生成有用的数据文件。

(7) `LOCAL_CXXFLAGS`: 与 `LOCAL_CFLAGS` 相同,针对 C++源文件。

(8) `LOCAL_CPPFLAGS`: 与 `LOCAL_CFLAGS` 相同,但是对 C 和 C++ source files 都适用。

(9) `LOCAL_STATIC_LIBRARIES`: 应该链接到这个模块的静态库列表(使用 `BUILD_STATIC_LIBRARY` 生成),这仅仅对共享库模块才有意义。

(10) `LOCAL_SHARED_LIBRARIES`: 这个模块在运行时要依赖的共享库模块列表,在链接时需要,在生成文件时嵌入的相应的信息。注意: 这不会附加列出的模块到编译图,也就是仍然需要在 `Application.mk` 中把它们添加到程序要求的模块中。

(11) `LOCAL_LDLIBS`: 编译模块时要使用的附加的链接器选项。这对于使用 '-l' 前缀传递指定库的名字是有用的。例如,下面将告诉链接器生成的模块要在加载时刻链接到 `/system/lib/libz.so`

```
LOCAL_LDLIBS := -lz
```

可查看 `docs/STABLE-APIS.TXT` 获取使用 NDK 发行版能链接到的开放的系统库列表。

(12) `LOCAL_ALLOW_UNDEFINED_SYMBOLS`: 默认情况下,在试图编译一个共享库时,任何未定义的引用将导致一个“未定义的符号”错误。这对于在源代码文件中捕捉错误会有很大的帮助。然而,如果因为某些原因,需要不启动这项检查,可把这个变量设为 'true'。注意相应的共享库可能在运行时加载失败。(这个一般尽量不要去设为 true)。

(13) `LOCAL_ARM_MODE`: 默认情况下,arm 目标二进制会以 thumb 的形式生成(16 位),你可以通过设置这个变量为 arm 如果你希望你的 module 是以 32 位指令的形式。

'arm' (32-bit instructions) mode. E.g.:

```
LOCAL_ARM_MODE := arm
```

【注意: 同样可以在编译的时候告诉系统编译特定的类型】

(14) `LOCAL_SRC_FILES := foo.c bar.c.arm` 这样就告诉系统总是将 `bar.c` 以 `arm` 的模式编译，下面是 GNU Make ‘功能’ 宏，必须通过使用 `'$(call)'` 来求值，他们返回文本化的信息。

(15) `my-dir`: 返回当前 `Android.mk` 所在的目录路径，相对于 NDK 编译系统的顶层。这是有用的，在 `Android.mk` 文件的开头如此定义：

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

(16) `all-subdir-makefiles`: 返回一个位于当前 `'my-dir'` 路径的子目录列表。例如，看下面的目录层次：

```
sources/foo/Android.mk
sources/foo/lib1/Android.mk
sources/foo/lib2/Android.mk
```

如果 `sources/foo/Android.mk` 包含一行：

```
include $(call all-subdir-makefiles)
```

那么它就会自动包含 `sources/foo/lib1/Android.mk` 和 `sources/foo/lib2/Android.mk`。这项功能用于向编译系统提供深层次嵌套的代码目录层次。注意，在默认情况下，NDK 将会只搜索在 `sources/*/Android.mk` 中的文件。

(17) `this-makefile`: 返回当前 Makefile 的路径(即这个函数调用的地方)

(18) `parent-makefile`: 返回调用树中父 Makefile 路径。即包含当前 Makefile 的 Makefile 路径。

(19) `grand-parent-makefile`

3. Android.mk 使用模板

在一个 `Android.mk` 中可以生成多个可执行程序、动态库和静态库。

(1) 编译应用程序的模板：

```
#Test Exe

LOCAL_PATH := $(call my-dir)

#include $(CLEAR_VARS)

LOCAL_SRC_FILES := main.c

LOCAL_MODULE := test_exe

#LOCAL_C_INCLUDES :=

#LOCAL_STATIC_LIBRARIES :=

#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_EXECUTABLE)
```

【注：‘:=’是赋值的意思，‘\$’是引用某变量的值】

`LOCAL_SRC_FILES` 中加入源文件路径，`LOCAL_C_INCLUDES` 中加入所需要包含的头文件路径，`LOCAL_STATIC_LIBRARIES` 加入所需要链接的静态库 (*.a) 的名称，`LOCAL_SHARED_LIBRARIES` 中加入所需要链接的动态库 (*.so) 的名称，`LOCAL_MODULE`

表示模块最终的名称，BUILD_EXECUTABLE 表示以一个可执行程序的方式进行编译。

(2) 编译静态库的模板：

```
#Test Static Lib
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_SRC_FILES := \
    helloworld.c

LOCAL_MODULE:= libtest_static

#LOCAL_C_INCLUDES :=

#LOCAL_STATIC_LIBRARIES :=

#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_STATIC_LIBRARY)
```

和上面相似，BUILD_STATIC_LIBRARY 表示编译一个静态库。

(3) 编译动态库的模板：

```
#Test Shared Lib

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_SRC_FILES := helloworld.c

LOCAL_MODULE := libtest_shared

TARGET_PRELINK_MODULES := false

#LOCAL_C_INCLUDES :=

#LOCAL_STATIC_LIBRARIES :=

#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_SHARED_LIBRARY)
```

和上面相似，BUILD_SHARED_LIBRARY 表示编译一个共享库。

以上三者的生成结果分别在如下目录中，generic 依具体 target 会变：

out/target/product/generic/obj/EXECUTABLE
out/target/product/generic/obj/STATIC_LIBRARY
out/target/product/generic/obj/SHARED_LIBRARY

每个模块的目标文件夹分别为：

- 1) 可执行程序：XXX_intermediates
- 2) 静态库： XXX_static_intermediates
- 3) 动态库： XXX_shared_intermediates

另外，在 Android.mk 文件中，还可以指定最后的目标安装路径，用 LOCAL_MODULE_PATH

和 `LOCAL_UNSTRIPPED_PATH` 来指定。不同的文件系统路径用以下的宏进行选择：

`TARGET_ROOT_OUT`：表示根文件系统。

`TARGET_OUT`：表示 `system` 文件系统。

`TARGET_OUT_DATA`：表示 `data` 文件系统。

用法如：

```
LOCAL_MODULE_PATH := $(TARGET_ROOT_OUT)
```