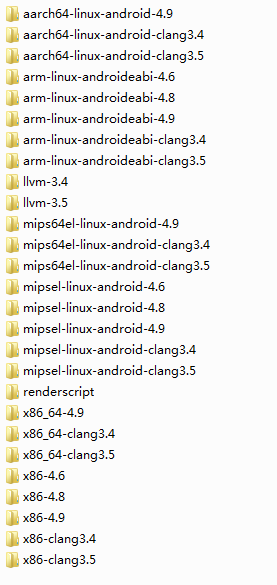
**总结：**

NDK的编译工具链有：



1. 当前编译armeabi-v7a armeabi

选择编译工具 arm-linux-androideabi-4.8

2. 当前编译 arm64-v8a

选择编译工具 aarch64-linux-android-4.9

不同工具 对应的底层库 stdint-gcc.h (在编译工具的目录下) 的基础类型定义都不一样

<NDK>\toolchains\aarch64-linux-android-4.9\prebuilt\windows-x86\_64\lib\gcc\aarch64-linux-android\4.9\include\stdint-gcc.h

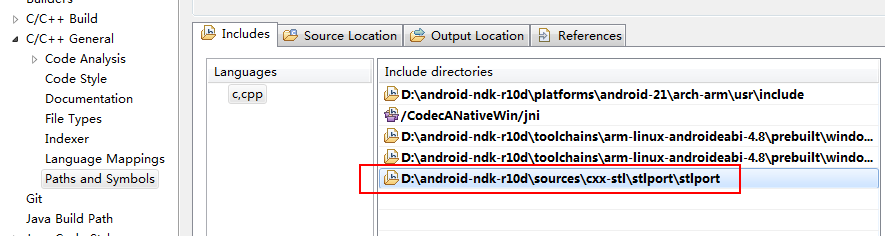
<NDK>\toolchains\arm-linux-androideabi-4.8\prebuilt\windows-x86\_64\lib\gcc\arm-linux-androideabi\4.8\include\stdint-gcc.h

所以格式化打印会不一样

int64\_t arm(32bit) int64\_t {aka long long int} %lld

arm64(64bit) int64\_t {aka long int} %ld

如果使用stlport 需要在Eclipse中导入下面的头文件(只是用作写代码时候的补全和提示)：



同时要在Application.mk添加 APP\_STL := stlport\_static

可供选择项目如下：

APP\_STL := stlport\_static --> static STLport library

对应动态库：<NDK>/sources/cxx-stl/stlport/libs/<架构>/ libstlport\_static.a

APP\_STL := stlport\_shared --> shared STLport library

对应动态库：<NDK>/sources/cxx-stl/stlport/libs/<架构>/libstlport\_shared.so

或者

<NDK>/sources/cxx-stl/stlport/libs/<armeabi/thumb>/ libstlport\_shared.so

只有 armeabi armeabi-v7a armeabi-v7a-hard 会有thumb对应的stlport库(默认 通过ndk-build V=1)

对应头文件：<NDK>/sources/cxx-stl/stlport/stlport

**APP\_STL := system**  --> default C++ runtime library(默认 如果不设置APP\_STL的话) 最小运行C++

对应库: libstdc++.a

APP\_STL:= gnustl\_static --> GUN C++ Extension

对应动态库： <NDK>/sources/cxx-stl/gnu-libstdc++/<$(TOOLCHAIN\_VERSION)>/libs/<架构>/

<NDK>/sources/cxx-stl/gnu-libstdc++/4.9/libs/arm64-v8a/ 或者

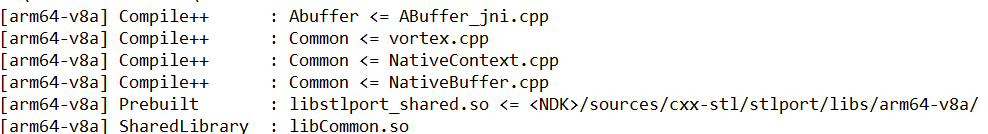
<NDK>/sources/cxx-stl/gnu-libstdc++/4.8/libs/armeabi/thumb/ (在armeabi armeabi-v7 支持thumb) 或者

<NDK>/sources/cxx-stl/gnu-libstdc++/4.8/libs/armeabi/

对应头文件：<NDK>/sources/cxx-stl/gnu-libstdc++/4.9/include

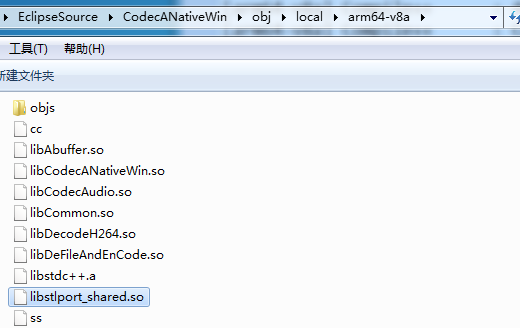
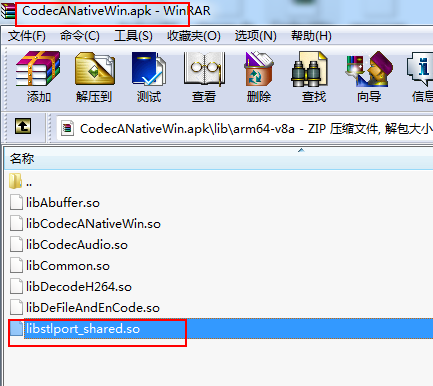
APP\_STL:= gnustl\_shared

编译时候会看到如下打印(使用 APP\_STL := stlport\_shared



1. 同时在obj目录 和 libs目录都可以看到对应so ，打包的apk文件中也会包含这个libstlport\_shared.so ，因为在系统库目录/system/lib中不包含这个动态库

2. 如果是APP\_STL := stlport\_static 直接使用libstlport\_static.a编译到我们自己的库或者执行文件中 没有拷贝到libs obj 或者apk里面

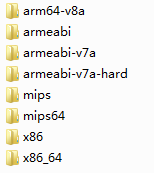
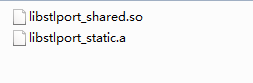
无符号(libs目录) 带符号(obj目录)

stlport\_static 79,440 525,221

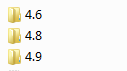
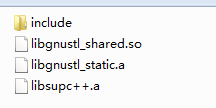
stlport\_shared 17,728 184,544 (使用动态链接库 明显减少了)

对于stlport标准库 目前提供的架构支持：

<NDK>\sources\cxx-stl\stlport\libs\

每一个架构都会对应一个静态库 一个动态库

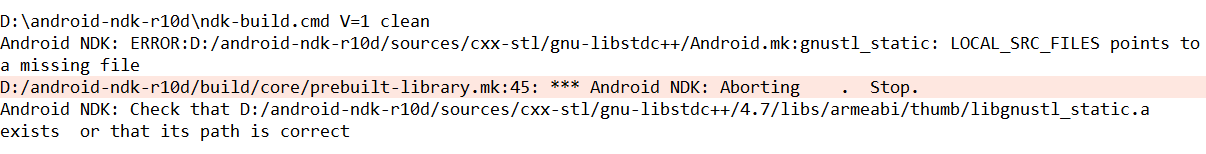
对于gun cpp 库 目前 根据 编译器版本 和 架构

编译器版本架构  对应的动态库 和 静态库 及 支持异常和rtti的静态库 

Application.mk可以指定版本NDK\_TOOLCHAIN\_VERSION，但是不一定所有编辑工具都有对应版本，

比如NDK\_TOOLCHAIN\_VERSION = 4.7

目前可以改成 NDK\_TOOLCHAIN\_VERSION = 4.9



**说明：**

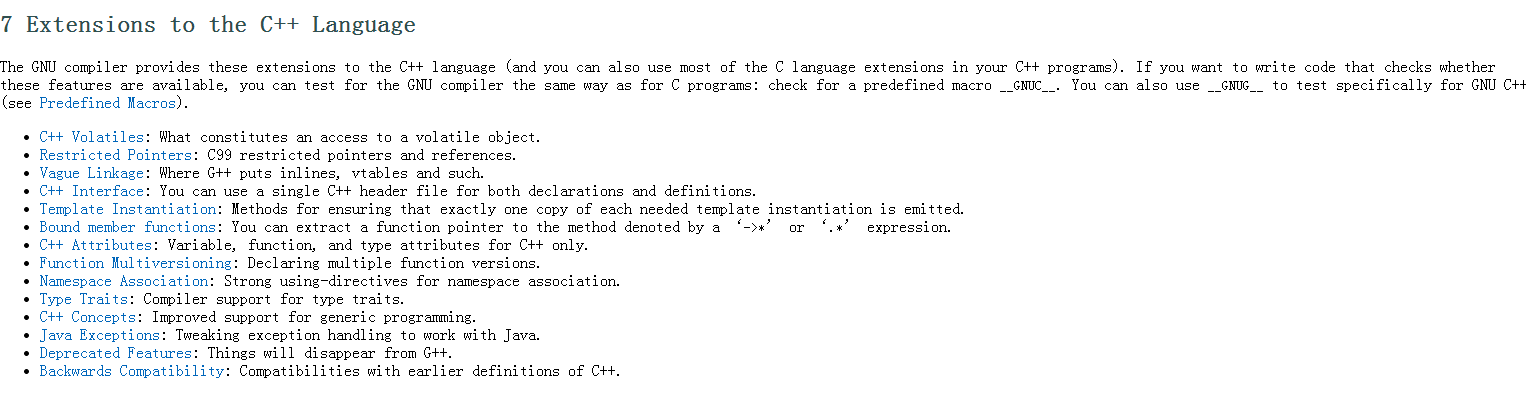
C++有Standard C++ 和 GNU C++两个版本 c99 and gnu99

-std=c++11

-std=gnu++11

GUN C++的扩展: 很多情况下 扩展并不会与Standard冲突

<http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Extensions.html>



C++异常和RTTI（Run Time Type Information）需要库 libsupc++  arm-linux-androideabi-g++ .... -lsupc++ 用 -fno-exception 和 -fno-rtti 关闭

( gnu-libstdc++ 下的 libsupc++.a )

C++异常支持 (http://www.jianshu.com/p/07856720339e)

Android.mk配置

LOCAL\_MODULE := module

LOCAL\_CPP\_FEATURES += exceptions

include $(BUILD\_SHARED\_LIBRARY)

Application.mk配置

APP\_STL := gnustl\_shared

APP\_CPPFLAGS += -fexceptions

C++RTTI支持

Android.mk配置

LOCAL\_MODULE := module

LOCAL\_CPP\_FEATURES += rtti

include $(BUILD\_SHARED\_LIBRARY)

Application.mk配置

APP\_STL := gnustl\_shared

APP\_CPPFLAGS += -frtti



**最小运行C++ 库**： LOCAL\_LDLIBS := -lstdc++ 实际在固件 system/lib/libstdc++.so)

./platforms/android-14/arch-arm/usr/lib/libstdc++.a

./platforms/android-14/arch-arm/usr/lib/libstdc++.so

..........

./platforms/android-21/arch-arm/usr/lib/libstdc++.a

./platforms/android-21/arch-arm/usr/lib/libstdc++.so

**APP\_STL** := gnustl\_static  
指定application里**要链接的标准c++库**。如果不指定，默认情况下，使用**最小化的c++运行时系统库**。

但是似乎APP\_STL只是对动态库有效，如果是要编译出可以执行程序，则没有效果(据说是ndkr6的一个bug)，需要显示地指定用gnustl\_static

如Android.mk里：  
**LOCAL\_STATIC\_LIBRARIES** := libTest1 libTest2 **gnustl\_static**

从android-ndk-r6b\sources\cxx-stl\gnu-libstdc++\Android.mk内容：

include $(CLEAR\_VARS)

LOCAL\_MODULE := **gnustl\_shared**

LOCAL\_SRC\_FILES := $(TOOLCHAIN\_VERSION)/libs/$(TARGET\_ARCH\_ABI)/libgnustl\_shared$(TARGET\_SONAME\_EXTENSION)

# For armeabi\*, choose thumb mode unless LOCAL\_ARM\_MODE := arm

ifneq (,$(filter armeabi%,$(TARGET\_ARCH\_ABI)))

ifneq (arm,$(LOCAL\_ARM\_MODE))

ifneq (arm,$(TARGET\_ARM\_MODE))

LOCAL\_SRC\_FILES:= $(TOOLCHAIN\_VERSION)/libs/$(TARGET\_ARCH\_ABI)/thumb/libgnustl\_shared$(TARGET\_SONAME\_EXTENSION)

endif

endif

endif

LOCAL\_EXPORT\_CPPFLAGS := $(gnustl\_exported\_cppflags)

LOCAL\_EXPORT\_C\_INCLUDES := $(gnustl\_exported\_c\_includes)

LOCAL\_EXPORT\_LDLIBS := $(call host-path,$(LOCAL\_PATH)/$(TOOLCHAIN\_VERSION)/libs/$(TARGET\_ARCH\_ABI)/libsupc++$(TARGET\_LIB\_EXTENSION))

include $(PREBUILT\_SHARED\_LIBRARY)

LOCAL\_MODULE := **gnustl\_static**

LOCAL\_SRC\_FILES := $(TOOLCHAIN\_VERSION)/libs/$(TARGET\_ARCH\_ABI)/libgnustl\_static$(TARGET\_LIB\_EXTENSION)

# For **armeabi\***, choose **thumb mode** unless LOCAL\_ARM\_MODE := arm **TARGET\_ARCH\_ABI 设置成armeabi-v7a 而且 LOCAL\_ARM\_MODE TARGET\_ARM\_MODE都不是arm**

ifneq (,$(filter armeabi%,$(TARGET\_ARCH\_ABI)))

ifneq (arm,$(LOCAL\_ARM\_MODE))

ifneq (arm,$(TARGET\_ARM\_MODE))

LOCAL\_SRC\_FILES:= $(TOOLCHAIN\_VERSION)/libs/$(TARGET\_ARCH\_ABI)/**thumb**/**libgnustl\_static**$(TARGET\_LIB\_EXTENSION) 这里选择的是thumb目录下的

endif

endif

endif

LOCAL\_EXPORT\_CPPFLAGS := $(gnustl\_exported\_cppflags)

LOCAL\_EXPORT\_C\_INCLUDES := $(gnustl\_exported\_c\_includes)

include $(PREBUILT\_STATIC\_LIBRARY)

可知，**gnustl\_static其实就是LOCAL\_MODULE 名字（所以可以在LOCAL\_STATIC\_LIBRARIES中添加gnustl\_static）**，指定了c++库用gnu的libstdc++.a静态库  
在ndk-build 加参数V=1可以查看编译时的详细参数输出，如NDK自带的gcc的路径

其他方法可以看"NDK Applicaton.mk使用手册"，比如设置 APP\_STL

[NDK Application.mk使用手册](http://blog.csdn.net/flydream0/article/details/7433786)

**APP\_CFLAGS**

可以在Application.mk文件中修改此编译选项参数，此参数可以覆盖Android.mk文件中的相同定义，从而不必去Android.mk文件中修改.

需要注意地是，这个选项里的路径必须是以顶层NDK目录为相对路径，如存在以下两个文件：

sources/foo/Android.mk

sources/bar/Android.mk

如果在foo/Android.mk文件中想要添加bar目录，那么得这样便用：

**APP\_CFLAGS += -I**sources/bar

如果这样使用:

APP\_CFLAGS += -I$(LOCAL\_PATH)/../bar

将会变成'-I$NDK\_ROOT/../bar'，从而出现并不是你想要的结果.

在android-ndk-1.5\_r1时，此选项中只在C代码中被支持，而C＋＋不支持，不过现在已经被纠正了

**APP\_CXXFLAGS**

APP\_CPPFLAGS的别名，将来此变量将会被抛弃。

**APP\_CPPFLAGS**

C++代码的编译选项。在android-ndk-1.5\_r1版本中，此变量只适用于C++，但是现在可以同时适用于C和C++。

**APP\_ABI**

在默认情况下，NDK会使用'armeabi' ABI 来生成二进制机器码，这是基于ARMv5TE的浮点运算CPU，这可以通过使用此变量来选项不同的ABI(Application Binary Interface).

例如:支持基于armv7 FPU指令集的设备:

APP\_ABI := armeabi-v7a

支持IA-32指令集：

APP\_ABI := x86

同时支持三种：

APP\_ABI := armeabi armeabi-v7a x86

从NDK-r7版本后，同时支持三种还可以这样写：

APP\_ABI := all

有关ABI更详细的内容请参考文档：**docs/CPU-ARCH-ABIS.html**

**APP\_STL**

C++标准库

在默认情况下，NDK通过Androoid自带的最小化的C++运行库(system/lib/libstdc++.so)来提供标准C++头文件.然而，NDK提供了可供选择的C++实现，你可以通过此变量来选择使用哪个或链接到你的程序。

APP\_STL := stlport\_static --> static STLport library

APP\_STL := stlport\_shared --> shared STLport library

**APP\_STL := system**  --> default C++ runtime library

APP\_STL:= gnustl\_static --> GUN C++ Extension

APP\_STL:= gnustl\_shared

更多详细内容请参考文档docs/CPLUSPLUS-SUPPORT.html。

**APP\_GNUSTL\_FORCE\_CPP\_FEATURES**

在先前的NDK版本中，当使用GNU libstdc++ runtime运行库（通过设置APP\_STL变量为gnustl\_static或gnustl\_shared）都会强制支持异常和RTTI,在有些极少情况下可能会出现问题，同时会使生成的机器码包含不必须的内容。

这种问题在NDK r7b中得到解决，但是这也意味着如果你真的需要支持异常和RTTI 的话，则必须显式声明。要么通过APP\_CPPFLAGS，要么通过LOCAL\_CPPFLAGS，或LOCAL\_CPP\_FEATURES。本变量就是为了解决此问题的.

有两个选项供选择，也可同时都选择.

exceptions -> 强制**所有模块**支持异常.

rtti -> 强制所有模块支持rtti.

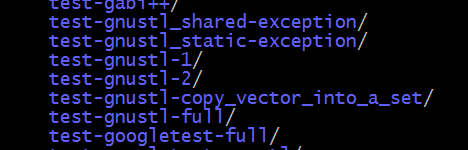
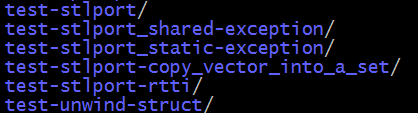
例如:

**APP\_GNUSTL\_FORCE\_CPP\_FEATURES := exceptions rtti**

**或者**

jni/Android.mk **LOCAL\_CPPFLAGS += -frtti -fexceptions** cpp flag还可以是 -std=gnu++98 -std=c++11

NDK目录下 android-ndk-r10d/tests/device 有很多测试C++ 特性的代码：

**独立编译链工具**

作用：

1. 可以通过ndk提供的arm-linux-androideabi-g++ 或者 aarch64-linux-androdeabli-g++直接编译出在android设备上运行的执行程序，其使用的库是ndk中的c/c++等native库

2. 独立出来的toolchain可是直接在Makefile中使用，而不需要自己去重新编写Android.mk文件，减少了很多麻烦，毕竟用自己熟悉的Makefile能解决问题。

$ export PATH=/<替换成安装目录>/bin:$PATH

(下面的配置CC CXX CXXFLAGS是为了./configure脚本中 会获取)

$ export CC=arm-linux-androideabi-gcc (arm-linux-androideabi-gcc已经添加在PATH中)

$ export CXX=arm-linux-androideabi-g++

$ export CXXFLAGS="-lstdc++" (std 需要使用 gunstl(默认))

执行完以上设置环境变量的命令之后，就可以直接编译了

执行 ./configure 然后 make 得到的就是可以在Android设备上运行的程序,不用再设定sysroot , 可以使用 STL，异常，RTTI

创建方式：

在 build/tools 目录准备了一个 make-standalone-toolchain.sh 的程序来让用户创建标准的交叉编译工具链(Cross Toolchain)

echo WARNING: make-standalone-toolchain.sh will be removed in r13. Please try \

后续应该改用make\_standalone\_toolchain.py now to make sure it works for your needs.

/mnt/android-ndk-r9c/build/tools/make-standalone-toolchain.sh

--platform=android-19

--toolchain=arm-linux-androideabi-4.8

--stl=stlport

--install-dir=/mnt/my\_ndk\_toolchain

出现以下打印：

dir=/mnt/my\_ndk\_toolchain

Copying prebuilt binaries...

Copying sysroot headers and libraries...

Copying libstdc++ headers and libraries...

Copying files to: /mnt/my\_ndk\_toolchain

Cleaning up...

Done.

说明独立的工具链成功，对执行的命令进行简单说明：

参数说明：

官方说明文件：<https://developer.android.com/ndk/guides/standalone_toolchain.html>

usage: make\_standalone\_toolchain.py [-h] --arch

{arm,arm64,mips,mips64,x86,x86\_64}

[--api API]

[--stl {gnustl,libc++,stlport}] [--force]

[-v]

[--package-dir PACKAGE\_DIR | --install-dir INSTALL\_DIR]

Creates a toolchain installation for a given Android target. The output of

this tool is a more typical cross-compiling toolchain. It is indended to be

used with existing build systems such as autotools.

optional arguments:

-h, --help show this help message and exit

--arch {arm,arm64,mips,mips64,x86,x86\_64}

--api API Target the given API version.

--stl {gnustl,libc++,stlport}

C++ STL to use.

--force Remove existing installation directory if it exists.

-v, --verbose Increase output verbosity.

--package-dir PACKAGE\_DIR

Build a tarball and install it to the given directory.

If neither --package-dir nor --install-dir is

specified, a tarball will be created and installed to

the current directory.

--install-dir INSTALL\_DIR

Install toolchain to the given directory instead of

packaging.

./make-standalone-toolchain.sh --help

WARNING: make-standalone-toolchain.sh will be removed in r13. Please try make\_standalone\_toolchain.py now to make sure it works for your needs.

HOST\_OS=linux

HOST\_EXE=

HOST\_ARCH=x86\_64

HOST\_TAG=linux-x86\_64

HOST\_NUM\_CPUS=1

BUILD\_NUM\_CPUS=2

Usage: make-standalone-toolchain.sh [options]

Generate a customized Android toolchain installation that includes

a working sysroot. The result is something that can more easily be

used as a standalone cross-compiler, e.g. to run configure and

make scripts. (也就是生成了一个编译器 专门用来编译Android的交叉编译器)

Valid options (defaults are in brackets):

--help Print this help.

--verbose Enable verbose mode.

--dryrun Set to dryrun mode.

--toolchain=<name> Specify toolchain name 指定交叉编译器名字 /android-ndk-r10d/toolchains 目录下的一个

--use-llvm Use LLVM. ?? ndk-r10d 目前支持 llvm-3.5 llvm-3.4 ??

--stl=<name> Specify C++ STL [gnustl] C++ STL特性实现库 默认gunstl stlport代表C++库将静态链接，stlport\_shared将动态链接

--arch=<name> Specify target architecture 目标机 架构

--abis=<list> Specify list of target ABIs.

--ndk-dir=<path> Take source files from NDK at <path> [.] NDK目录

--package-dir=<path> Place package file in <path> [/tmp/ndk-hhl] 安装成压缩包

--install-dir=<path> Don't create package, install files to <path> instead. 不打包到--package-dir指定目录的tar包 而是直接到到目录上创建 sysroot 环境和 工具链

--platform=<name> Specify target Android platform/API level. [android-9] 比如 android-19

编译工具链：



LLVM：

作为开源领域第二大的编译器，甚至在某些方面专门为了替代GCC而实现的LLVM来说，近些年它已经在不断的接近甚至准备超越GCC了,LLVM可以编译出接近与GCC接近相同性能的运行码,C++实现的编译器工具链,以C++编写而成，用于优化以任意程序语言编写的程序的编译时间(compile-time)、链接时间(link-time)、运行时间(run-time)以及空闲时间(idle-time)。

我的配置 :

#!/bin/bash

INSTALL\_DIR=`pwd`/my\_android\_toolchain

./build/tools/make\_standalone\_toolchain.py \

--arch arm64 \

--stl gnustl \

--api 21 \

--install-dir $INSTALL\_DIR

翻译文档：

$NDK/build/tools/make\_standalone\_toolchain.py \

--arch arm --api 21 --install-dir /tmp/my-android-toolchain

这个命令将会创建 /tmp/my-android-toolchain/ 目录 并包含一份android-21/arch-arm的拷贝,作为sysroot，并且是arm 架构 32bits (路径可以任意指定)

--arch 选项是必须的

但是 --stl选项是默认为gnustl  相当于默认 --stl=gnustl --api选项默认是架构的最低版本(当前32bit是android-9 64bit是android-21)

与旧的工具不同，Clang(Clang是一个C语言、C++、Objective-C、C++语言的轻量级编译器, 基于[LLVM](http://baike.baidu.com/view/2995210.htm))总是拷贝到工具链中(toolchain)，每个独立工具链都可以使用Clang和GCC

可以使用--stl=stlport 来拷贝 libstlport 到工具链，而不是默认的 libgnustl.

1. 如果使用--stl=stlport 制作独立工具链，在使用的时候，需要显示指定 -lstlport\_shared 来编译 ；

2. 如果使用--stl=libgnustl(GNU libstdc++.)制作独立工具链，在使用的时候，需要显示指定  -lgnustl\_shared来编译 ；

3. 如果使用--stl=libc++ 拷贝LLVM的libc++头文件和库来制作独立工具链，与gnustl and stlport不同，在使用时候不需要指定 -lc++\_shared作为动态库，如果要静态链接，就要使用-static-libstdc++在链接时候(这跟一般的主机编译器一样)，另外，可能还经常需要传递-latomic (比如 arm-linux-g++ -latomic <filename> 当使用<atomic> )

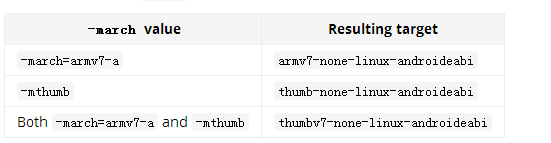
使用Clang

Clang是自动拷贝到独立工具链

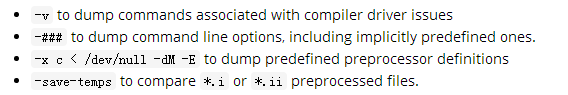
Clang 拷贝与GCC同样版本，因为他们依赖同样的assembler, linker, 头文件, 库文件, and C++ STL实现

同时会安装两个wrapper脚本，名为clang 和 clang++ 在安装目录的bin下，这两个脚本回调clang库 使用对应目标架构标记 (the correct target architecture flags)

当使用Clang来编译ARM的工具 ，Clang会根据-march=选项来改变目标平台



clang and clang++  应该在Makefile中使用 gcc and g++ ，如果有怀疑的话，可以使用下面方法打印编译命令



ABI兼容性 ABI Compatibility （Application Binary Interface, or *ABI* ）

默认 ARM Clang独立工具链 目标给armeabi-v7a ABI

GCC独立工具链 目标给 armeabi

都可以通过-march 编译选项来指定 ， Clang还可以通过-target选项来控制

为了用 gcc 来编译armeabi-v7a平台程序，必须传递如下选项(标记)给编译器：-march=armv7-a -mfloat-abi=softfp -mfpu=vfpv3-d16.

-march=armv7-a 为 armv7架构

-mfloat-abi=softfp 使能硬件FPU浮点数指定 确保系统会传递浮点参数到cpu对应寄存器

建议使用-mthumb编译选项，来强制产生16-bit Thumb-2 指令 (Thumb-1 for armeabi).如果不指定，将会产生32bit ARM指令

如果使用neon请必须使用 : -mfpu=neon. 选项，同时这将会强制使用VFPv3-D32

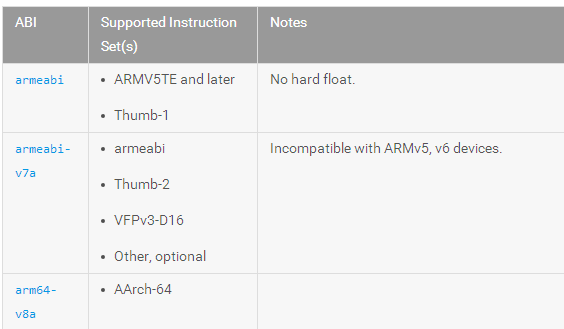
同时 必须传递如下选项给连接器 -march=armv7-a -Wl,--fix-cortex-a8.

第一个标记 指示 连接器 选择 armv7-a的库

第二个标记 解决部分 Cortex-A8 实现的CPU bug (-Wl选项告诉编译器将后面的参数传递给链接器)

如果是其他 ABIs，不需要指定特别的编译标记(complier flag)

关于更多的ABI支持，可以看<https://developer.android.com/ndk/guides/abis.html>



Exceptions和RTTI

工具链默认支持C++异常和RTTI ,如果要禁止，使用 -fno-exceptions and -fno-rtti.

如果使用 GNU libstdc++ 连接时候需要指定  libsupc++ arm-linux-androideabi-g++ .... **-lsupc++**

如果使用 STLport or libc++  就不需要指定

C++ STL支持：

libgnustl\_shared.so  libstlport\_shared.so   libc++\_shared.so  如果是动态连接的话，必须包含在APP中，因为系统库路径上没有这些，运行可能出错

3、测试程序

hello.cpp

#include <iostream>

#include <string>

int main(int argc, char \*\*argv)

{

std::string str = "hello, ndk! this is my own toolchain! ^-^";

std::cout << str << std::endl;

return 0;

}

Makefile

rm=/bin/rm -f

CC=/mnt/my\_ndk\_toolchain/bin/arm-linux-androideabi-g++

PROGNAME = main

INCLUDES= -I.

CFLAGS = $(INCLUDES) -g -fPIC -D\_FILE\_OFFSET\_BITS=64 -D\_LARGE\_FILE

OBJS = hello.o

LDFLAGS =

all :$(PROGNAME)

%.o: %.cpp

$(CC) $(CFLAGS) -c -o $@ $<

$(PROGNAME) : $(OBJS)

@echo "Linking $(PROGNAME)......"

${CC} ${LDFLAGS} -o ${PROGNAME} ${OBJS}

@echo "Linking Success!"

clean:

$(rm) \*.o $(PROGNAME)

编译后得到可执行文件:main，adb push到嵌入式Android平台后，./main运行，得到以下结果：

root@android :/data # ./main

hello, ndk! this is my own toolchain! ^-^

**异常， RTTI 和 STL**

NDK 工具链默认支持C++异常和RTTI（Run Time Type Information），可以用 -fno-exception 和 -fno-rtti 关闭（生成的机器码更小）

注意： 如果要用这两个特性，需要显式链接 libsupc++。例如： arm-linux-androideabi-g++ .... -lsupc++

NDK 提供了 libstdc++，因而可以用 STL，但需要显式链接 libstdc++ ( gcc ... -lstdc++)。不过在将来可以不用手动指定这个链接参数。

**最快速编译 执行程序：**

$ export CC="$NDK/toolchains/<name>/prebuilt/<host-system>/bin/<prefix>gcc --sysroot=$SYSROOT"

$ $CC -o foo.o -c foo.c

要确保该程序没有链接ndk未提供的共享库 所以 该方法的缺陷就是  C++ STL（STLport 或 GNU libstdc++ ），也不能使用异常机制和RTTI

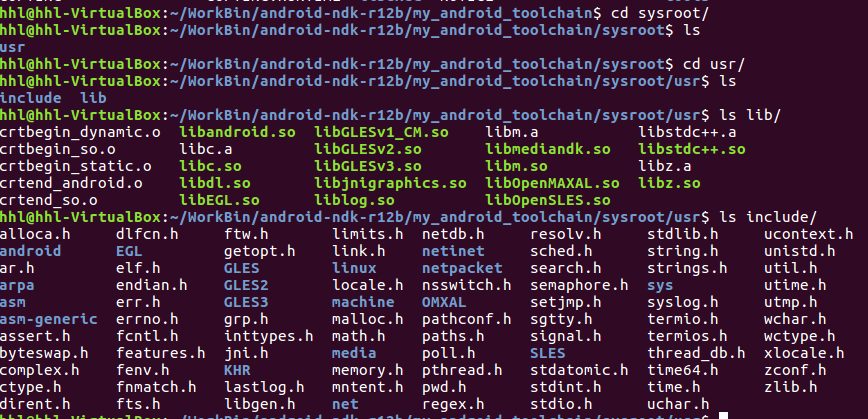
可以直接在后面加上 -lstdc++ ,

gcc 的sysroot 选项

sysroot 选项设定 gcc 在编译源码的时候，寻找头文件和库文件的根目录。可以这样调用 gcc --sysroot=/tmp/gcc-arm (及其他选项)。NDK 根目录下的 platforms 目录中的各个子目录的路径都可以直接传给 gcc --sysroot=<dir>。为了简化操作，可以在linux系统的命令终端执行以下命令，设置SYSROOT环境变量，$NDK是ndk的根目录。

$ SYSROOT=$NDK/platforms/android-8/arch-arm

A sysroot is a directory containing the system headers and libraries for your target. 编译器根据sysroot指定的路径 在编译时 找到头文件和库(对应目标)



ndk 同时支持 arm5 和 arm7，一般只用 arm5就好了。arm7是高端一点的，NDK 默认也是 arm5 。

推荐加上 -mthumb 选项给gcc，来生成 16-bit Thumb-1 指令。

如果要用 arm7，可以设定 CFLAGS='-march=armv7-a -mfloat-abi=softfp'， 使用 Thumb-2 指令，且这两个选项不能分开！

#!/bin/bash

NDK=/home/cxx/ndk/android-ndk-r10c

SYSROOT=$NDK/platforms/android-19/arch-arm/

TOOLCHAIN=$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/linux-x86\_64

function build\_one

{

./configure \

--prefix=$PREFIX \

--enable-shared \

--disable-static \

--disable-doc \

--disable-ffserver \

--enable-gpl\

--enable-libx264 \

--enable-encoder=libx264 \

--enable-decoder=h264 \

--enable-cross-compile \

--cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \

--target-os=linux \

--arch=arm \

--sysroot=$SYSROOT \

--extra-cflags="-Os -fpic $ADDI\_CFLAGS" \

--extra-ldflags="$ADDI\_LDFLAGS" \

$ADDITIONAL\_CONFIGURE\_FLAG

}

CPU=arm

PREFIX=$(pwd)/android/$CPU

ADDI\_CFLAGS="-marm -I../x264/android/arm/include -DANDROID "

ADDI\_LDFLAGS="-L../x264/android/arm/lib"

build\_one