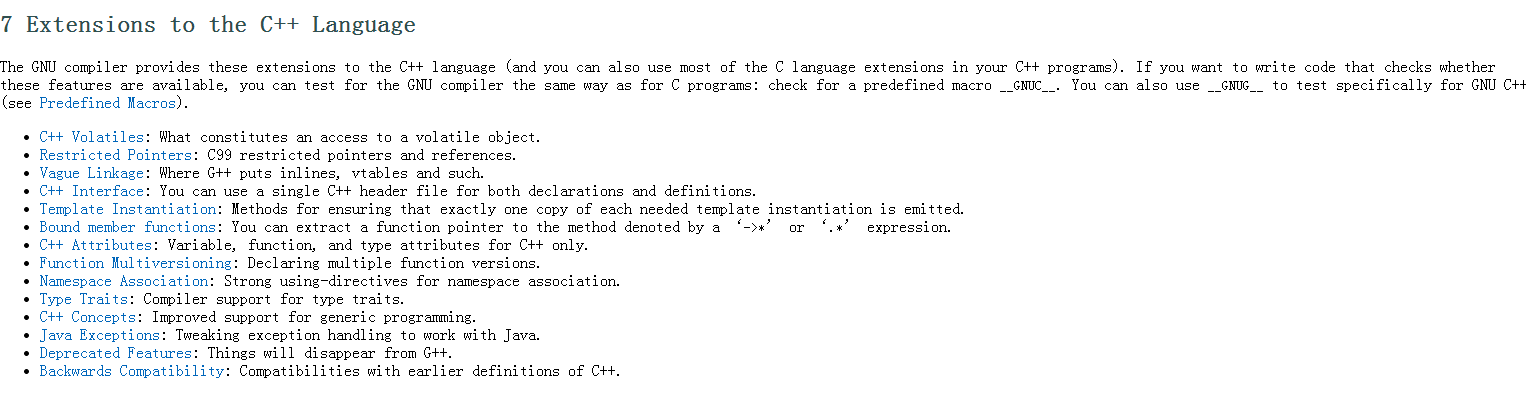
C++有Standard C++ 和 GNU C++两个版本 c99 and gnu99

-std=c++11

-std=gnu++11

GUN C++的扩展: 很多情况下 扩展并不会与Stand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 )

**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

根据Android NDK 产生独立编译链工具

GNUSTL\_DIR=$NDK\_DIR/$GNUSTL\_SUBDIR/$GCC\_VERSION

GNUSTL\_LIBS=$GNUSTL\_DIR/libs

STLPORT\_DIR=$NDK\_DIR/$STLPORT\_SUBDIR

STLPORT\_LIBS=$STLPORT\_DIR/libs

LIBCXX\_DIR=$NDK\_DIR/$LIBCXX\_SUBDIR

LIBCXX\_LIBS=$LIBCXX\_DIR/libs

case $ARCH in

x86|x86\_64|mips|mips64)

LIBCXX\_SUPPORT\_LIB=gabi++

;;

\*)

LIBCXX\_SUPPORT\_LIB=**libc++abi arm用这个**

;;

esac

......

case $STL in

gnustl)

copy\_directory "**$GNUSTL\_DIR**/include" "$ABI\_STL\_INCLUDE"

;;

libcxx|libc++)

copy\_directory "$LIBCXX\_DIR/libcxx/include" "$ABI\_STL\_INCLUDE"

copy\_directory "$SUPPORT\_DIR/include" "$ABI\_STL\_INCLUDE"

if [ "$LIBCXX\_SUPPORT\_LIB" = "gabi++" ]; then

copy\_directory "$STLPORT\_DIR/../gabi++/include" "$ABI\_STL\_INCLUDE/../../gabi++/include"

copy\_abi\_headers gabi++ cxxabi.h unwind.h unwind-arm.h unwind-itanium.h gabixx\_config.h

elif [ "$LIBCXX\_SUPPORT\_LIB" = **"libc++abi"** ]; then

copy\_directory "$LIBCXX\_DIR/..**/llvm-libc++abi/**libcxxabi/include" "$ABI\_STL\_INCLUDE/../../llvm-libc++abi/include"

copy\_abi\_headers llvm-libc++abi cxxabi.h libunwind.h unwind.h

else

dump "ERROR: Unknown libc++ support lib: $LIBCXX\_SUPPORT\_LIB"

exit 1

fi

;;

stlport)

copy\_directory "**$STLPORT\_DIR/stlport**" "$ABI\_STL\_INCLUDE"

copy\_directory "$STLPORT\_DIR/../gabi++/include" "$ABI\_STL\_INCLUDE/../../gabi++/include"

copy\_abi\_headers gabi++ cxxabi.h unwind.h unwind-arm.h unwind-itanium.h gabixx\_config.h

;;

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。

从Android NDK中独立toolchain步骤（系统为Ubuntu(32位)）：

1、下载Android NDK

地址：http://developer.android.com/tools/sdk/ndk/index.html ，下载对应版本

个人网盘共享链接：http://pan.baidu.com/s/1hqGIV9E

2、独立toolchain

把NDK压缩包解压到系统，如/mnt目录下，后在/mnt目录下建立文件夹my\_ndk\_toolchain，然后再/mnt目录下执行以下命令：

/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：

/mnt/android-ndk-r9c/build/tools/make-standalone-toolchain.sh：执行NDK目录下make-standalone-toolchain.sh脚本；

--platform：API版本选择 指工具链将使用哪个版本的Android API，可cd /mnt/android-ndk-r9c/platform中查看，我这里使用的是Android-19；

--toolchain: 平台选择 指独立出来的工具链哪种用途的编译，arm(arm-linux-androideabi-4.8),X86(x86-4.8)或MIPS(mipsel-linux-android-4.8)，可cd toolchains中查看并选择适合的类型，我这里使用的是嵌入式；

--stl:指工具链支持C++ stl，stlport代表C++库将静态链接，stlport\_shared将动态链接；

--install-dir:指安装目录；

--system:主机系统类型

注意：因为我使用的是32-bit Ubuntu，独立工具链默认是32位，所以在参数中没有指定系统类型，如果是64-bit Linux系统，需加入--system=linux-x86\_64 或MacOSX加入--system=darwin-x86\_64。

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

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

--llvm-version=<ver> Specify LLVM version ?? 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 还可以是stlport 可以查看make-standalone-toolchain.sh中

android-ndk-r10d/sources/cxx-stl 目录下

gnu-libstdc++(下面还分为4.6 4.8 4.9/libs/<arch>/\*so \*.a 根据编译器的名字 --toolchain )

arm-linux-androideabi-4.6 arm-linux-androideabi-4.8 arm-linux-androideabi-4.9

x86-4.6 x86-4.8 x86-4.9

stlport(没有版本)

--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

--system=<name> Specify host system [windows] 主机系统

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

--install-dir=<path> Don't create package, install files to <path> instead. 安装到目录 ，并不打包到--package-dir指定目录的tar包

--platform=<name> Specify target Android platform/API level. [android-3] API版本

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。

**第二个例子：**

android ndk 提供脚本，允许自己定制一套工具链。例如：

$NDK/build/tools/make-standalone-toolchain.sh --platform=android-5 --install-dir=/tmp/my-android-toolchain [ --arch=x86 ]

将会在/tmp/my-android-toolchain 中创建 sysroot 环境和 工具链。--arch 选项选择目标程序的指令架构，默认是为 arm。

如果不加 --install-dir 选项，则会创建 /tmp/ndk/<toolchain-name>.tar.bz2。

运行之后，这样使用：

$ export PATH=/tmp/my-android-toolchain/bin:$PATH

$ export CC=arm-linux-androideabi-gcc

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

$ export CXXFLAGS="-lstdc++"

执行完以上设置环境变量的命令之后，就可以直接编译了（例如，执行 ./configure 然后 make 得到的就是 arm 程序了）。

不用再设定 sysroot, CC 了。而且，可以使用 STL，异常，RTTI。

**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

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

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

**APP\_CFLAGS**

编译选项

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** +=  **-f**rtti -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：

