## 1.1 ABI 是什么

ABI 是 Application Binary Interface 的缩写。

不同 Android 手机使用不同的 CPU，因此支持不同的指令集。CPU 与指令集的每种组合都有其自己的应用二进制界面（或 ABI）。 ABI 可以非常精确地定义应用的机器代码在运行时如何与系统交互。 您必须为应用要使用的每个 CPU 架构指定 ABI。

**典型的 ABI 包含以下信息：**

- 机器代码应使用的 CPU 指令集。
- 运行时内存存储和加载的字节顺序。
- 可执行二进制文件（例如程序和共享库）的格式，以及它们支持的内容类型。
- 用于解析内容与系统之间数据的各种约定。这些约定包括对齐限制，以及系统如何使用堆栈和在调用函数时注册。
- 运行时可用于机器代码的函数符号列表 - 通常来自非常具体的库集。

## 1.2 如何在 gardle 中配置

默认情况下，cmake 会输出 4 种 ABI（"armeabi-v7a" , "arm64-v8a", "x86", "x86_64"），如下所示：


我们也可以通过 abiFilters 来指定我们需要的 ABI：

> abiFilters "armeabi", "armeabi-v7a" , "arm64-v8a", "x86", "x86_64", "mips", "mips64"

## 1.3 build_command.txt 详解

```
cmake的可执行文件在那个路径 （绝对路径）
Executable :
C:\Users\Administrator\AppData\Local\Android\Sdk\cmake\3.10.2.4988404\bin\cmake.exe
执行cmake时 携带的参数
arguments :
编译的源码放在哪个文件夹
-HE:\maniu\NativeTest\app\src\main\cpp
无效路径
-DCMAKE_FIND_ROOT_PATH=E:\maniu\NativeTest\app\.cxx\cmake\debug\prefab\armeabi-v7a\prefab
-DCMAKE_BUILD_TYPE=Debug
-
导入系统的库 如liblog.so  libjnigraphics.so
DCMAKE_TOOLCHAIN_FILE=C:\Users\Administrator\AppData\Local\Android\Sdk\ndk\21.0.6113669\build\cmake\android.toolchain.cmake
编译平台
-DANDROID_ABI=armeabi-v7a
NDK绝对路径
-DANDROID_NDK=C:\Users\Administrator\AppData\Local\Android\Sdk\ndk\21.0.6113669
Android最低平台版本
-DANDROID_PLATFORM=android-23
当前编译 生成so的版本
-DCMAKE_ANDROID_ARCH_ABI=armeabi-v7a
Debug下的NDK绝对路径
-
DCMAKE_ANDROID_NDK=C:\Users\Administrator\AppData\Local\Android\Sdk\ndk\21.0.6113669
Debug下打开Cmake命令输出
-DCMAKE_EXPORT_COMPILE_COMMANDS=ON
指定临时中间文件
-
DCMAKE_LIBRARY_OUTPUT_DIRECTORY=E:\maniu\NativeTest\app\build\intermediates\cmake\debug\obj\armeabi-v7a
指定语法解释器 ninja
-
DCMAKE_MAKE_PROGRAM=C:\Users\Administrator\AppData\Local\Android\Sdk\cmake\3.10.2.4988404\bin\ninja.exe
android 系统名称  Android
-DCMAKE_SYSTEM_NAME=Android
系统版本
-DCMAKE_SYSTEM_VERSION=23
最终生成的so库所在的地方
-BE:\maniu\NativeTest\app\.cxx\cmake\debug\armeabi-v7a
-GNinja
jvmArgs :

Build command args:
```

## 1.4 Linux平台下自己通过cmake 编译

> centos 7.8
>
> cmake 2.8.12
>
> g++

**安装cmake步骤**

> 1 yum install cmake
>
> 2 yum install gcc-c++

```
cmake_minimum_required(VERSION 2.8.12)
project(LS19 C)

set(CMAKE_C_STANDARD 99)

add_executable(LS19 main.c)
```

```
#include <stdio.h>

int main() {
    printf("main, World!\n");
    return 0;
}
```

### 1.4.1编译语法

```
cd source (CMakeLists.txt 所在目录)
cmake .
make
```

## 1.5 动态库与静态库区别

动态库的添加:

```
link_directories(${PROJECT_SOURCE_DIR}/lib) #添加动态连接库的路径
target_link_libraries(project_name -lmxnet ) #添加libmxnet.so12
```

静态库的添加:

```
#为main添加共享库链接
#target_link_libraries(LS20  -lLS19)
add_library(staticFiled STATIC IMPORTED)
set_property(TARGET staticFiled PROPERTY IMPORTED_LOCATION
${PROJECT_SOURCE_DIR}/lib/libLS19.a)
#生成可执行文件
add_executable(LS20 main.c)
target_link_libraries(LS20 staticFiled ) #添加libmxnet.a
```

## 1.6 FFmpeg Cmake示例写法

```
cmake_minimum_required(VERSION 3.4.1)

#add libavcodec
add_library(avcodec
        SHARED
        IMPORTED
        )

SET_TARGET_PROPERTIES(
        avcodec
        PROPERTIES IMPORTED_LOCATION
        ${PROJECT_SOURCE_DIR}/ffmpeg/prebuilt/${ANDROID_ABI}/libavcodec.so
)

#add libavfilter
add_library(avfilter
        SHARED
        IMPORTED
        )

SET_TARGET_PROPERTIES(
        avfilter
        PROPERTIES IMPORTED_LOCATION
        ${PROJECT_SOURCE_DIR}/ffmpeg/prebuilt/${ANDROID_ABI}/libavfilter.so
)


#add libavformat
add_library(avformat
        SHARED
        IMPORTED
        )

SET_TARGET_PROPERTIES(
        avformat
        PROPERTIES IMPORTED_LOCATION
        ${PROJECT_SOURCE_DIR}/ffmpeg/prebuilt/${ANDROID_ABI}/libavformat.so
)


#add libavutil
add_library(avutil
        SHARED
        IMPORTED
        )

SET_TARGET_PROPERTIES(
        avutil
```

```
        PROPERTIES IMPORTED_LOCATION
        ${PROJECT_SOURCE_DIR}/ffmpeg/prebuilt/${ANDROID_ABI}/libavutil.so
)


#add libpostproc
add_library(postproc
        SHARED
        IMPORTED
        )

SET_TARGET_PROPERTIES(
        postproc
        PROPERTIES IMPORTED_LOCATION
        ${PROJECT_SOURCE_DIR}/ffmpeg/prebuilt/${ANDROID_ABI}/libpostproc.so
)

#add libswresample
add_library(swresample
        SHARED
        IMPORTED
        )

SET_TARGET_PROPERTIES(
        swresample
        PROPERTIES IMPORTED_LOCATION
        ${PROJECT_SOURCE_DIR}/ffmpeg/prebuilt/${ANDROID_ABI}/libswresample.so
)

#add libswscale
add_library(swscale
        SHARED
        IMPORTED
        )

SET_TARGET_PROPERTIES(
        swscale
        PROPERTIES IMPORTED_LOCATION
        ${PROJECT_SOURCE_DIR}/ffmpeg/prebuilt/${ANDROID_ABI}/libswscale.so
)

include_directories(ffmpeg/)


find_library(
        log-lib
        log)

add_library(
        ffmpeg-cmd
        SHARED
        ffmpeg/ffmpeg-cmd.cpp ffmpeg/ffmpeg.c ffmpeg/cmdutils.c
ffmpeg/ffmpeg_filter.c ffmpeg/ffmpeg_hw.c ffmpeg/ffmpeg_opt.c
)
target_link_libraries( # Specifies the target library.
        ffmpeg-cmd

        avcodec
```

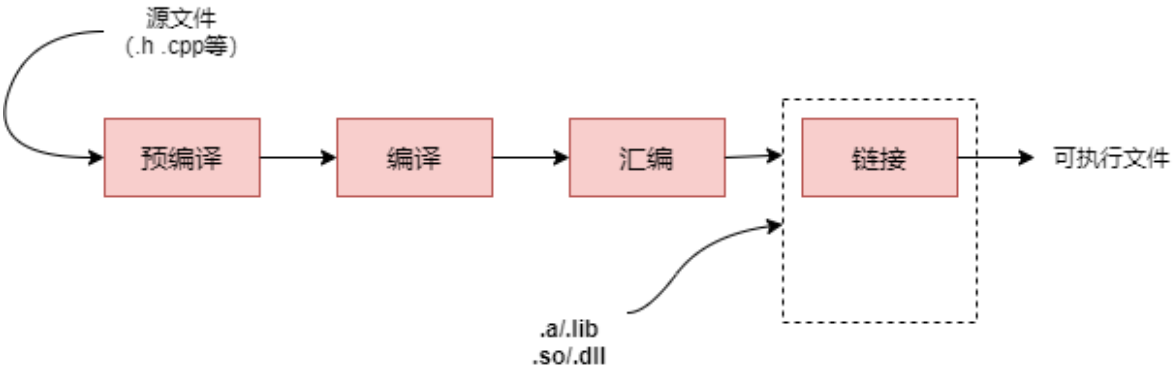
```
        swscale
        swresample
        postproc
        avutil
        avformat
        avfilter
        ${log-lib})
```

## 1.7 什么是库

库是写好的，成熟的，可以复用的代码，一般程序运行都需要依赖许多底层库文件。

本质上来说库是一种可可执行代码的二进制形式，可以被操作系统载入内存执行，库有两种：静态库（.a、.lib）和动态库（.so、.so）。

静态、动态是指链接，将一个程序编译成可执行程序步骤如下：




静态库和动态库的区别主要来自【链接阶段】如何处理库，链接成可执行文件，分别称为静态链接和动态链接

程序编译过程

```
一步到位编译：gcc main.c -o main

预处理 -E (.i) 编译 -S (.s) 汇编-c (.o) 连接-o

预处理

gcc -E main.c -o main.i -E:仅执行编译预处理

-o:将结果输出并指定输出文件的文件名

编译为汇编代码

gcc -S main.i -o main.s -S：将C代码转换为汇编代码

汇编：

gcc -c main.c -o main.o -c：仅执行编译操作，不进行连接操作
```

连接：

```
gcc main.o -o main
```

### 1.7.1 静态库

所谓静态库，是因为在链接阶段，会将汇编生成的目标文件.o与引用到的库一起链接打包到可执行文件中，对应的链接方式成为静态链接。

静态库与汇编生成的目标文件一起链接成为可执行文件，那么可以得出，静态库的格式跟.o文件格式相似，其实一个静态库可以简单看成是一组目标文件（.o/.ojb文件）的集合，即很多目标文件经过压缩打包后形成的一个文件。

- 优点
  - 程序在运行时与函数库就没有关系，移植方便
- 缺点
  - 浪费空间和资源，所有相关的目标文件与牵涉的函数库被链接合成一个可执行文件

### 1.7.2 Linux下创建与使用静态库

**Linux下静态库命名规则**

必须是**lib{your_library_name}.a**：lib为前缀，中间是静态库名，扩展名为.a

**创建静态库（.a）**

下面以一个简单四则运算C++类为例，将其编译为静态库给他人用。

头文件

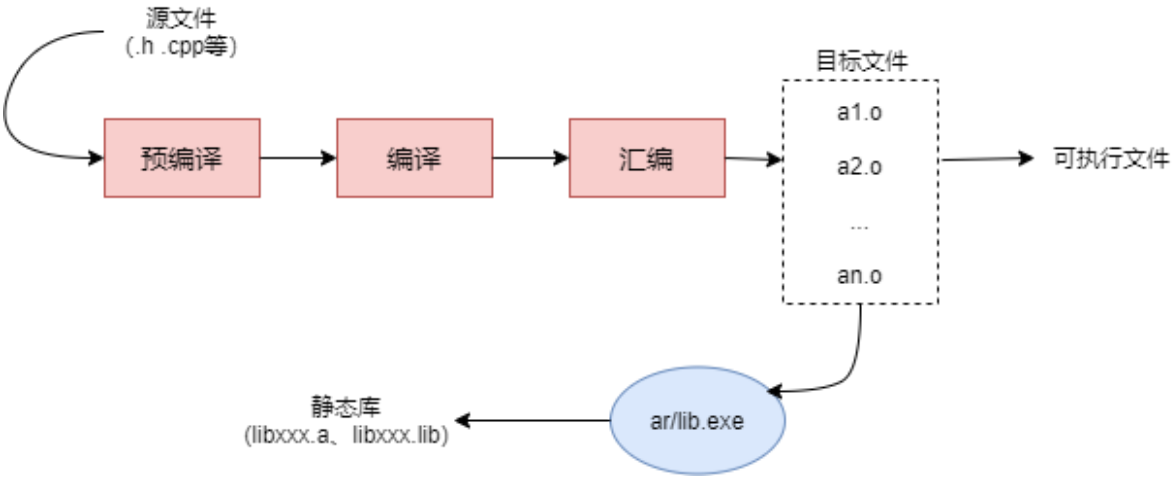
```
class StaticMath {
  public:
    static double add(double a, double b);
    static double sub(double a, double b);
    static double mul(double a, double b);
    static double div(double a, double b);
    void print();
};
```

实现

```
#include "StaticMath.h"
#include <iostream>

double StaticMath::add(double a, double b) {
  return a + b;
}

double StaticMath::sub(double a, double b) {
  return a - b;
}

double StaticMath::mul(double a, double b) {
  return a * b;
```

```
}

double StaticMath::div(double a, double b) {
  return a / b;
}

void StaticMath::print() {
  std::cout << "Static Math Library" << std::endl;
}
```

Linux通过**ar**工具，Windows下vs使用lib.exe，将目标文件压缩到一起，并且对其进行编号和索引，以便于查找和检索。一般创建静态库的步骤如下：




创建静态库过程

- 将代码文件编译为目标文件.o（StaticMath.o）

```
g++ -c StaticMath.cpp
```

- 通过ar命令将目标文件打包为.a静态文件

```
ar -crv libstaticmath.a StaticMath.o
```

生成静态库libstaticmath.a

**使用静态库**

- 编写测试代码

```
#include "StaticMath.h"
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
  double a = 10;
  double b = 2;
  cout << "a + b = " << StaticMath::add(a, b) << endl;
  cout << "a - b = " << StaticMath::sub(a, b) << endl;
  cout << "a * b = " << StaticMath::mul(a, b) << endl;
  cout << "a / b = " << StaticMath::div(a, b) << endl;
  StaticMath sm;
  sm.print();
```

```
    return 0;
}
```

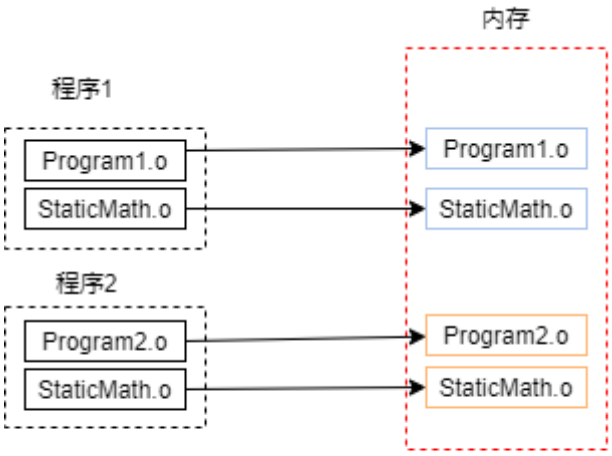
Linux环境下使用静态库，只需要在编译的时候，指定静态库搜索路径（-L选项）和指定库名（-l选项）

```
# 编译
g++ -o staticmatch StaticMathTest.cpp -L/home/username/googletest/mybuild/mytest
-lstaticmath
# 执行
./staticmatch
```

## 1.8 动态库

**为什么需要动态库?**

- 静态库会造成空间浪费，如下图：

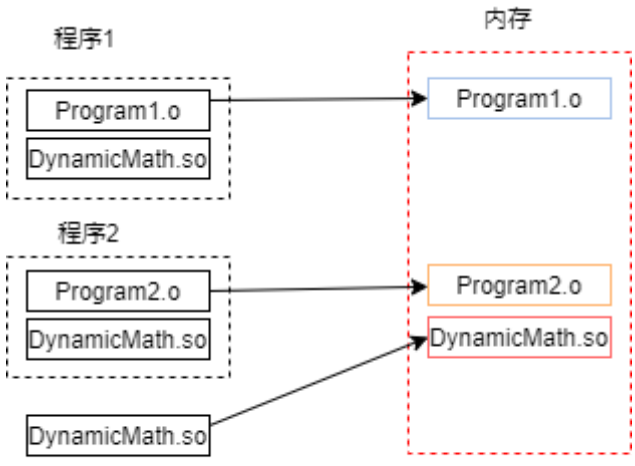



静态库在内存中存在多分拷贝导致空间浪费，假设，静态库占用1M内存，有1000个这样的程序，将占用1GB的空间

- 静态库对程序更新、部署和发布会带来麻烦，如果静态库更新，则所有使用它的应用程序都需要重新编译、发布给用户（一个小的改动，可能导致整个程序重新下载）。

动态库优点

- 可以实现进程之间资源共享（因此动态库也称为共享库），如下图：




动态库在内存中只存在一份拷贝，避免静态库浪费空间的问题

- 使程序升级变得简单

## 1.9 Linux下创建与使用动态库

**Linux下动态库命名规则**
命名形式为libxxx.so，前缀是lib，后缀名为“.so”

**创建静态库（.so）**

类似四则运算代码

头文件

```
class DynamicMath {
  public:
    static double add(double a, double b);
    static double sub(double a, double b);
    static double mul(double a, double b);
    static double div(double a, double b);
    void print();
};
```

实现

```
#include "DynamicMath.h"
#include <iostream>

double DynamicMath::add(double a, double b) {
  return a + b;
}

double DynamicMath::sub(double a, double b) {
  return a - b;
}

double DynamicMath::mul(double a, double b) {
  return a * b;
}

double DynamicMath::div(double a, double b) {
  return a / b;
}

void DynamicMath::print() {
  std::cout << "DynamicMath Math Library" << std::endl;
}
```

与静态库不同，创建动态库不需要打包工具（ar，lib.exe），直接使用编译器即可创建动态库。

- 生成目标文件，加编译选项-fpic

```
g++ -fPIC -c DynamicMath.cpp
```

- 生成动态库，加编译选项-shared

```
g++ -shared -o libdynmath.so DynamicMath.o
```

上面两个命令也可以合并为一个

```
g++ -fPIC -shared -o libdynmath.so DynamicMath.cpp
```

**使用静态库**

- 编写测试代码

```
#include "DynamicMath.h"
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
  double a = 10;
  double b = 2;
  cout << "a + b = " << DynamicMath::add(a, b) << endl;
  cout << "a - b = " << DynamicMath::sub(a, b) << endl;
  cout << "a * b = " << DynamicMath::mul(a, b) << endl;
  cout << "a / b = " << DynamicMath::div(a, b) << endl;
  DynamicMath dyn;
  dyn.print();
  return 0;
}
```

由于动态库是在程序运行时进行链接，所以在程序运行时需要让系统能够找到动态库，系统一般会依次搜索：环境变量LD_LIBRARY_PATH、/etc/ld.so.cache文件列表、/lib、/usr/lib目录找到库文件后将其载入内存。因此主要有三种方法来设置动态库路径。

- 将动态库绝对路径加入环境变量LD_LIBRARY_PATH
- 将动态库绝对路径加入/etc/ld.so.cache文件中，步骤如下：
  - 编辑/etc/ld.so.conf文件，加入文件所在目录的路径
  - 运行ldconfig，重建/etc/ld.so.cache
- 将动态库移到/lib或/usr/lib中

```
# 编译
g++ -o dynamicmath DynamicMathTest.cpp -L/home/dgh/googletest/mybuild/mytest -
ldynmath
# 执行
./dynamicmath # 需要先设置好动态库路径
```

clang编译c文件

下载最新的clang版本，地址：[http://www.llvm.org/releases/download.html#3.7.0](http://www.llvm.org/releases/download.html#3.7.0)

然后编写测试用的c代码，保存为main.c.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("main World!");
    system("pause");
    return 0;
}
```

使用Win + R，切换到main.c.c的目录下，然后执行clang --verbose main.c.c会遇到错误

**1 静态链接库的优点**

(1) 代码装载速度快，执行速度略比动态链接库快；

(2) 只需保证在开发者的计算机中有正确的.LIB文件，在以二进制形式发布程序时不需考虑在用户的计算机上.LIB文件是否存在及版本问题，可避免so地狱等问题。

**2 动态链接库的优点**

(1) 更加节省内存并减少页面交换；

(2) so文件与EXE文件独立，只要输出接口不变（即名称、参数、返回值类型和调用约定不变），更换so文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性；

(3) 不同编程语言编写的程序只要按照函数调用约定就可以调用同一个so函数；

(4)适用于大规模的软件开发，使开发过程独立、耦合度小，便于不同开发者和开发组织之间进行开发和测试。

**3 不足之处**

(1) 使用静态链接生成的可执行文件体积较大，包含相同的公共代码，造成浪费；

(2) 使用动态链接库的应用程序不是自完备的，它依赖的so模块也要存在，如果使用载入时动态链接，程序启动时发现so不存在，系统将终止程序并给出错误信息。而使用运行时动态链接，系统不会终止，但由于so中的导出函数不可用，程序会加载失败；速度比静态链接慢。当某个模块更新后，如果新模块与旧的模块不兼容，那么那些需要该模块才能运行的软件，统统撕掉。这在早期Windows中很常见。