

Mac 下编译 libmono.so 和 DLL 加密详解

- Mac 下编译 libmono.so 和 DLL 加密详解

- 编译环境配置
 - 安装 HomeBrew
 - 按官网教程安装
 - 使用国内源安装
 - 使用 Ruby 脚本安装
 - 检查是否安装成功
 - 使用 HomeBrew 安装依赖
- 编译 libmono.so
 - 执行编译脚本
 - 编译脚本执行过程
 - configure 和 make
 - 错误查找过程
 - 编译选项优化
- DLL 加密与热更
 - 加密
 - 解密
 - 热更
 - 验证加密算法是否成功
- 总结

Unity 打出的安卓包为了防止反编译，需要对 Assembly-CSharp.dll 加密处理。Assembly-CSharp.dll 是由 libmono.so 运行时读取然后在 mono 虚拟机上执行，所以需要修改 libmono.so 源码，在加载 Assembly-CSharp.dll 前解密处理，然后重新编译出 libmono.so。

libmono.so 是由 Unity 官方 Fork 了开源的 Mono 编译出来的，Unity 官方也将其开源了，需要根据你的 Unity 版本下载对应分支的，这次我编译的是 Unity-2018.4 的，源码在这里：

- <https://github.com/Unity-Technologies/mono/tree/unity-2018.4>

本文不是傻瓜式教程告诉你如何编译的，而是用来讲述这个编译过程，附带我遇到的错误和解决思路。很多时候，照着别人的文档，甚至官方的，别人的操作成功了，自己的却一堆错，只有了解了这个编译过程，才能快速定位和解决问题。

编译环境配置

首先需要配置下 Mac 环境，编译 Untiy-Mono 需要安装一些编译脚本依赖的包。HomeBrew 是 MacOS 上的包管理工具，使用它安装这些依赖会很方便。

安装 HomeBrew

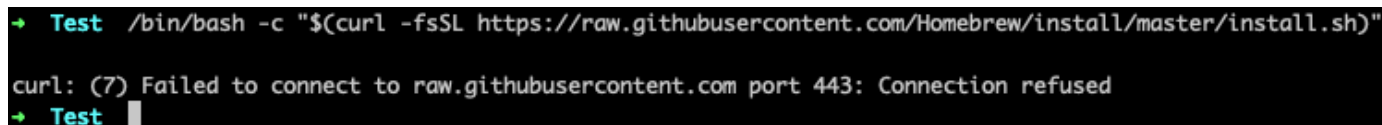
安装 HomeBrew 有时候不太顺利，这里提供 3 种安装方式，安装失败时可以切换试试。

按官网教程安装

官网中介绍的安装方式，执行下面的命令即可

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

我自己的 Mac 上最早就是这样安装的，但最近给另一台 Mac 配置环境时碰到了下图的错误，用 VPN 也不行。



```
→ Test /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
curl: (7) Failed to connect to raw.githubusercontent.com port 443: Connection refused
→ Test
```

使用国内源安装

在知乎上找到的一个国内源：

```
/bin/zsh -c "$(curl -fsSL https://gitee.com/cunkai/HomebrewCN/raw/master/Homebrew.sh)"
```

使用 Ruby 脚本安装

将 <https://gist.github.com/sunsetroads/11c35fb3caef2980041b1fcb07ab9a31> 的内容复制保存为 homebrew.rb，然后执行命令：

```
ruby homebrew.rb
```

检查是否安装成功

执行 `brew --help` 检查是否安装成功

```
➔ ~ brew --help
Example usage:
  brew search [TEXT|/REGEX/]
  brew info [FORMULA...]
  brew install FORMULA...
  brew update
  brew upgrade [FORMULA...]
  brew uninstall FORMULA...
  brew list [FORMULA...]
```

出现上图内容就说明 HomeBrew 安装成功了。

使用 HomeBrew 安装依赖

Mono-Unity 依赖下面这些包

- autoconf
- automake
- libtool
- pkg-config

使用 HomeBrew 一个个安装就行了：

```
brew install autoconf
```

编译 [libmono.so](#)

这里以 mono-unity-2018.4 为例，下载后在桌面新建文件夹 Test/T，将下载下来的源码放入，编译脚本运行后会在 mono-unity-2018.4 上级目录安装依赖，这样建目录会方便查看依赖包。

执行编译脚本

进入工程根目录 mono-unity-2018.4，执行编译脚本

`./external/buildscripts/build_runtime_android.sh` 开始编译：

```
➔ Desktop cd Test/T/mono-unity-2018.4
➔ mono-unity-2018.4 pwd
/Users/zhangning/Desktop/Test/T/mono-unity-2018.4
➔ mono-unity-2018.4 ./external/buildscripts/build_runtime_android.sh
Environment:
  Host      = macosx
  Temporary = /var/folders/d4/084t8_0113917tv4z910v6qm0000gn/T/
  Home      = /Users/zhangning
```

等了挺久，然后编译失败了：

```
checking for arm-eabi-linux-gcc... /Users/zhangning/android-ndk_auto-r10e/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86_64
/bin/arm-linux-androideabi-gcc --sysroot=/Users/zhangning/android-ndk_auto-r10e/platforms/android-9/arch-arm
checking for arm-eabi-linux-gcc... (cached) /Users/zhangning/android-ndk_auto-r10e/toolchains/arm-linux-androideabi-4.8/prebuilt/darw
in-x86_64/bin/arm-linux-androideabi-gcc --sysroot=/Users/zhangning/android-ndk_auto-r10e/platforms/android-9/arch-arm
checking whether the C compiler works... no
configure: error: in `/Users/zhangning/Desktop/Test/T/mono-unity-2018.4':
configure: error: C compiler cannot create executables
See `config.log' for more details
Configure FAILED!
```

查看日志还发现了时间久的原因，下载了 NDK-r10e 后，又去下载了 NDK-r16，总共 1G 多的文件，比较浪费时间了：

```
Environment:
  Host      = macosx
  Temporary = /var/folders/d4/084t8_0113917tv4z910v6qm0000gn/T/
  Home      = /Users/zhangning

  $ANDROID_NDK_ROOT = /Users/zhangning/android-ndk_auto-r10e

Installing NDK 'r16b':
  Currently installed = r10e (64-bit)
```

在网上搜了一会，没有好的解决办法，决定看下编译脚本的执行过程，来查找报错的根本原因。

PS：这一步还可能提示缺少什么包，按提示执行 `brew install` 包名 即可。

编译脚本执行过程

`build_runtime_android.sh` 就是入口脚本，先忽略掉杂要信息，看下它的关键内容：

```

function clean_build_krait_patch
{
    # 检查是否有下载 krait-signal-handler, 并执行 build.pl
    KRAIT_PATCH_PATH="${CWD}/../../android_krait_signal_handler/build"
    local KRAIT_PATCH_REPO="git://github.com/Unity-Technologies/krait-signal-handler"
    git clone --branch "master" "$KRAIT_PATCH_REPO" "$KRAIT_PATCH_PATH"
    (cd "$KRAIT_PATCH_PATH" && ./build.pl)
}

function clean_build
{
    make clean && make distclean

    ./configure

    if [ "$?" -ne "0" ]; then
        echo "Configure FAILED!"
        exit 1
    fi

    make && echo "Build SUCCESS!" || exit 1
}

perl ${BUILDSRIPTSDIR}/PrepareAndroidSDK.pl -ndk=r10e -env=envsetup.sh && source envsetup.sh

clean_build_krait_patch

clean_build "$CCFLAGS_ARMv7_VFP" "$LDFLAGS_ARMv7" "$OUTDIR/armv7a"

```

首先执行的 `perl ${BUILDSRIPTSDIR}/PrepareAndroidSDK.pl` , 注意这里传入的参数 `ndk-r10e`。然后执行 `clean_build_krait_patch` , 先去下载了 `krait-signal-handler` 包, 然后执行里面的 `build.pl` :

```

sub BuildAndroid
{
    PrepareAndroidSDK::GetAndroidSDK(undef, undef, "r16b");
    system('$ANDROID_NDK_ROOT/ndk-build clean');
    system('$ANDROID_NDK_ROOT/ndk-build');
}

```

这里 `krait-signal-handler` 中 `build.pl` 传入的参数是 `r16b`, 也就是说, 构建脚本依赖的 NDK 版本和 `krait-signal-handler` 依赖的不一致, 导致了重复下载, 所以要去把 `build.pl` 中的 `r16b` 改为 `r10e`。

编译脚本安装了依赖的环境后, 接着往下执行 `clean_build` :

```
make clean && make distclean

./configure

if [ "$?" -ne "0" ]; then
    echo "Configure FAILED!"
    exit 1
fi

make && echo "Build SUCCESS!" || exit 1
```

./configure、make、make install 命令这些都是典型的使用 GNU 的 AUTOCONF 和 AUTOMAKE 产生的程序的安装步骤，这里需要了解 configure 和 make 命令。

configure 和 make

在 Linux 下安装一个应用程序时，一般先运行脚本 configure，然后用 make 来编译源程序，在运行 make install，最后运行 make clean 删除一些临时文件。

configure 是一个 shell 脚本，它可以自动设定源程序以符合各种不同平台上 Unix 系统的特性，并且根据系统参数及环境产生合适的 Makefile 文件或是 C 的头文件 (header file)，让源程序可以很方便地在这些不同的平台上被编译链接。

运行 configure 脚本，就可产生出符合 GNU 规范的 Makefile 文件了，然后就可以运行 make 进行编译，再运行 make install 进行安装了，这里只需要编译。

引用自 <https://www.cnblogs.com/tinywan/p/7230039.html>

错误查找过程

了解了编译脚本的执行过程后，可以开始根据执行的 log 找编译失败的原因了。

在上面编译失败的 log 中可以看到一句 make: *** No rule to make target 'clean'. Stop :

```
obj/local/armeabi-v7a/objs/krait-signal-handler/krait_signal_handler.o:2: *** missing separator. Stop.
obj/local/armeabi-v7a/objs/krait-signal-handler/krait_signal_handler.o:2: *** missing separator. Stop.
updating: libkrait-signal-handler.a (deflated 59%)
updating: build.txt (deflated 3%)
make: *** No rule to make target 'clean'. Stop.
rm: android_cross.cache: No such file or directory
```

不熟悉 make 命令时还以为缺少什么环境，但其实编译失败和这个没关系，这是 configure 执行失败导致没有正常生成 Makefile，make 命令找不到 Makefile 文件后提示的，问题是出在 configure 脚本里。

中间的日志都可以忽略，直接看最后的报错：

```
checking for arm-eabi-linux-gcc... /Users/zhangning/android-ndk_auto-r10e/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86_64
/bin/arm-linux-androideabi-gcc --sysroot=/Users/zhangning/android-ndk_auto-r10e/platforms/android-9/arch-arm
checking for arm-eabi-linux-gcc... (cached) /Users/zhangning/android-ndk_auto-r10e/toolchains/arm-linux-androideabi-4.8/prebuilt/darw
in-x86_64/bin/arm-linux-androideabi-gcc --sysroot=/Users/zhangning/android-ndk_auto-r10e/platforms/android-9/arch-arm
checking whether the C compiler works... no
configure: error: in `/Users/zhangning/Desktop/Test/T/mono-unity-2018.4':
configure: error: C compiler cannot create executables
See `config.log' for more details
Configure FAILED!
```

这里提示 C compiler cannot create executables，检查了系统的 gcc 和 clang，都是可以正常编译 C 程序的，网上也找不到解决办法。于是去 config.log 查看更详细的信息。

执行 configure 时会把执行过程的详细信息输出到 config.log，终端中输出的只是一份简要的，这两个文件都位于 mono-unity-2018.4 根目录下。打开 config.log，在里面搜 C compiler cannot create executables，可以看到出现这个错误前发生了一些 error。

```
configure:4511: $? = 0
configure:4500: /Users/zhangning/android-ndk_auto-r10e/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-gcc --sysroot=/Users/
arm-linux-androideabi-gcc: error: unrecognized command line option '-V'
arm-linux-androideabi-gcc: fatal error: no input files
compilation terminated.
configure:4511: $? = 1
configure:4500: /Users/zhangning/android-ndk_auto-r10e/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-gcc --sysroot=/Users/
arm-linux-androideabi-gcc: error: unrecognized command line option '-qversion'
arm-linux-androideabi-gcc: fatal error: no input files
compilation terminated.
configure:4511: $? = 1
configure:4531: checking whether the C compiler works
configure:4553: /Users/zhangning/android-ndk_auto-r10e/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-gcc --sysroot=/Users/
/Users/zhangning/android-ndk_auto-r10e/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86_64/bin/../lib/gcc/arm-linux-androideabi/4.8/../../../../arm-linux-
error: cannot find -lkrait-signal-handler
collect2: error: ld returned 1 exit status
configure:4557: $? = 1
configure:4595: result: no
configure: failed program was:
| /* confdefs.h */
| #define PACKAGE_NAME ""
| #define PACKAGE_TARNAME ""
| #define PACKAGE_VERSION ""
| #define PACKAGE_STRING ""
| #define PACKAGE_BUGREPORT ""
| #define PACKAGE_URL ""
| #define PACKAGE "mono"
| #define VERSION "2.6.5"
| /* end confdefs.h. */
|
| int
| main ()
| {
|
| ...;
| ...return 0;
| }
configure:4600: error: in `/Users/zhangning/Desktop/Test/T/mono-unity-2018.4':
configure:4602: error: C compiler cannot create executables
```

来看这些 error，首先是 -V 和 -qversion 问题，提示的是 arm-linux-androideabi-gcc 不支持这些参数，这个输出在 configure 的 4500 行，看下对应的代码：

```
# Provide some information about the compiler.
$as_echo "$as_me:${as_lineno-$LINENO}: checking for C compiler version" >&5
set X $ac_compile
ac_compiler=$2
for ac_option in --version -v -V -qversion; do
{ { { ac_try="$ac_compiler $ac_option >&5"
case "($ac_try" in
  *\"* | *\\* | *\\*) ac_try_echo=$ac_try;;
  *) ac_try_echo=$ac_try;;
esac
eval ac_try_echo="\"$as_me:${as_lineno-$LINENO}: $ac_try_echo\""
```

这段代码是用来检查 arm-linux-androideabi-gcc 版本的，尝试了 --version -v -V -qversion 四个参数，使用 --veesion 和 -v 就可以获取到了，其他的参数不适用也无所谓，这个并不是真正的错误原因。

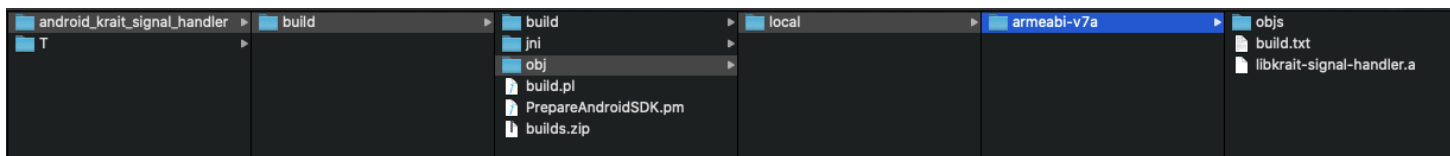
根据 error 发生的位置继续查看 configure 源码，从 4531 行到 4602 行，代码有点多，需要联系上下文才能理解。这段代码作用是来检查 NDK 中的 arm-linux-androideabi-gcc 编译器是否正常，判断的标准是用它编译一段简单的 C 程序，然后查看是否生成了可执行文件。这里最终没有生成，所以抛出了个 C compiler cannot create executables 错误。

config.log 里提到发生了一个 ld 链接器错误，cannot find -lkrait-signal-handler，忽略掉次要信息后，configure:4553 这一行是这样的：

```
arm-linux-androideabi-gcc -L/Test/T/mono-unity-2018.4/../../android_krait_signal_handler/build/obj/loca
```

意思是使用 arm-linux-androideabi-gcc 编译 conftest.c，并链接库 krait-signal-handler。

GCC 会在 -L 选项后紧跟着的基本名称的基础上自动添加前缀 lib、后缀 .a，这里基本名称为 krait-signal-handler。现在去 -L 后面的路径下看下是否存在 libkrait-signal-handler.a：



这时发现 libkrait-signal-handler.a 是存在的，只是前面的路径不对，configure 脚本以为 libkrait-signal-handler.a 位于 armabi 下，但实际编译出来的在 armabi-v7a，问题找到了，新建个 armabi 的目录将 krait-signal-handler.a 放入，再执行编译脚本。

等待终端刷屏了近 10 分钟，输出了下面的信息：

```
Build SUCCESS!
Build failed? Android SHARED library cannot be found... Found      2 libs under builds/embedruntimes/android
total 0
drwxr-xr-x  3 zhangning  staff  96  4 29 15:38 armv7a
drwxr-xr-x  3 zhangning  staff  96  4 29 15:40 x86
→ mono-unity-2018.4
```

编译成功了，在 builds/embedruntimes/android 下可以看到不同 CPU 架构下的 libmono.so。

编译选项优化

修改 ./external/buildscripts/build_runtime_android.sh 文件，在这个脚本中修改

-fpic -g -funwind-tables \ 为 -fpic -O2 -funwind-tables \， -g 打出来的 libmono.so 是 debug 版本的，文件会比较大。

修改 ./external/buildscripts/build_runtime_android_x86.sh，在这个脚本中把 -fpic -g \ 修改为 -fpic \，这个修改据说是因为 x86 的编译选项和 arm 不一样，不去掉 -g 一些手机上进入游戏会卡死。

如果只需要 armv7a 和 x86 的, 可以在 build_runtime_android.sh 中注释掉下面两项:

- clean_build "\$CCFLAGS_ARMv5_CPU" "\$LD_FLAGS_ARMv5" "\$OUTDIR/armv5"
- clean_build "\$CCFLAGS_ARMv6_VFP" "\$LD_FLAGS_ARMv5" "\$OUTDIR/armv6_vfp"

DLL 加密与热更

加密

在导出 Android 的工程的时候对 Assembly-CSharp.dll 进行加密, 具体做法是直接修改 Assembly-CSharp.dll 的二进制内容, 这里使用 C# 进行简单加密测试:

```
void EncryptDLL()
{
    Debug.Log ("EncryptDLL");
    if (!Directory.Exists (eclipseProPath)) {
        Debug.LogError("eclipse project not exist");
        return;
    }
    string inpath = eclipseProPath + "/assets/bin/Data/Managed/Assembly-CSharp.dll";
    if (File.Exists (inpath)) {
        byte[] bytes = File.ReadAllBytes (inpath);
        bytes [0] += 1;
        File.WriteAllBytes (inpath, bytes);
    } else {
        Debug.LogError("dll打开失败, 加密失败 path="+inpath);
    }
}
```

使用 file 命令查看加密后的 dll:

```
➔ Desktop file /Users/zhangning/Desktop/Assembly-CSharp.dll
/Users/zhangning/Desktop/Assembly-CSharp.dll: data
➔ Desktop
```

可以看到系统已经识别不出来这个 dll 了, 只把它看作 data 数据。

解密

修改 mono-unity-2018.4/mono/metadata/ 下的 image.c。这个文件包含有载入 DLL 的方法: mono_image_open_from_data_with_name, 在这个方法的入口处加入以下代码:

```
if(name != NULL && strstr(name,"Assembly-CSharp.dll") {
    data[0]-=1; //上面的加密算法是对第一个字节 +1, 这里需要 -1 来还原。
}
```

注意 dll 名字的判断不能少，因为工程中还存在其他没有加密的 dll。

热更

热更 dll 后，需要在 mono_image_open_from_data_with_name 里读取本地的 dll 进行加载，在 image.c 里添加读取文件的函数：

```
static FILE *OpenFileWithPath(const char *path)
{
    const char *fileMode = "rb";
    return fopen (path, fileMode);
}
static char *ReadStringFromFile(const char *pathName, int *size)
{
    FILE *file = OpenFileWithPath (pathName);
    if (file == NULL)
    {
        return 0;
    }
    fseek (file, 0, SEEK_END);
    int length = ftell(file);
    fseek (file, 0, SEEK_SET);
    if (length < 0)
    {
        fclose (file);
        return 0;
    }
    *size = length;
    char *outData = g_try_malloc (length);
    int readLength = fread (outData, 1, length, file);
    fclose(file);
    if (readLength != length)
    {
        g_free (outData);
        return 0;
    }
    return outData;
}
```

mono_image_open_from_data_with_name 中添加代码：

```

MonoImage *
mono_image_open_from_data_with_name (char *data, guint32 data_len, gboolean need_copy, I
{
    int dataszie = 0;
    //original name like /data/app/package-name-1.apk/assets/bin/Data/Managed/Assembly-C#
    g_message("mono: origianl path = %s\n", name);
    if (name != NULL && strstr(name, "Assembly-CSharp.dll"))
    {
        char *_name = "/storage/emulated/0/Android/data/包名/files/Android/Assembly-CSha
        char *dllBytes = ReadStringFromFile(_name, &dataszie);
        if (dataszie > 0)
        {
            g_message("mono: new");
            data = dllBytes;
            data_len = dataszie;
        }
        data[0] -= 1; //上面的加密算法是对第一个字节 +1, 这里需要 -1 来还原。
    }
    // 下面是 mono_image_open_from_data_with_name 原有的代码
    ...
}

```

注意 dll 的路径不要写错了，这里是直接写死

验证加密算法是否成功

由于无法直接执行 [libmono.so](#)，这里将解密相关内容拿出来作为一个 C 程序，这样就不用重复出包来验证加密算法了。代码如下：

```

#include <stdio.h>
#include <stdlib.h>

static FILE *OpenFileWithPath(const char *path)
{
    const char *fileMode = "rb";
    return fopen (path, fileMode);
}

static char *ReadStringFromFile(const char *pathName, int *size)
{
    FILE *file = OpenFileWithPath (pathName);
    if (file == NULL)
    {
        return 0;
    }
    fseek (file, 0, SEEK_END);
    int length = ftell(file);
    fseek (file, 0, SEEK_SET);
    if (length < 0)
    {
        fclose (file);
        return 0;
    }
    *size = length;
    char *outData = malloc (length);
    int readLength = fread (outData, 1, length, file);
    fclose(file);
    if (readLength != length)
    {
        free(outData);
        return 0;
    }
    return outData;
}

int main(int argc, const char * argv[]) {

    int data_len = 0;
    int dataszie = 0;
    //将这个路径替换为你加密后的 dll
    char *data = ReadStringFromFile("/Users/zhangning/Desktop/Assembly-CSharp.dll", &data_len);

    char *_name = "/storage/emulated/0/Android/data/vn.funtap.tinhkiem.mobile3d/files/Assembly-CSharp.dll";
    char *dllBytes = ReadStringFromFile(_name, &dataszie);

    if (dataszie > 0)
    {
        data = dllBytes;
        data_len = dataszie;
    }
}

```

```
}  
    //在这里替换你的解密算法  
    data[0] -= 1;  
  
    FILE *fp = NULL;  
    //将解密后的 dll 输出为 test.dll  
    fp = fopen("/Users/zhangning/Desktop/test.dll", "wb");  
    fwrite(data, 1, data_len, fp);  
    fclose(fp);  
    return 0;  
}
```

运行上方的解密代码后，用 file 命令查看 test.dll：

```
→ Desktop file /Users/zhangning/Desktop/test.dll  
/Users/zhangning/Desktop/test.dll: PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows
```

显示这样的结果就说明 dll 解密成功了，加密算法是没问题的。然后修改 image.c 重新编译就行了。

总结

一番折腾后，终于重新编译出需要的 [libmono.so](#)，中间踩了不少坑，教训是遇到不熟悉的领域里的问题时，不要总想着直接复制粘贴找答案，不要急，慢慢看代码梳理流程，才能查找出问题的真正原因。另外，有时会遇到一些特殊 BUG，需要控制变量多次测试，这时要用 Git 做好测试的版本管理，清楚地记录每次修改的内容和结论。