### 编译Android源码

#### 下载源码

repo sync

#### 选择 build 版本

repo init https:// -u ${build\_code}

#### 配置临时 (当前终端窗口) 环境变量

source build/envsteup.sh

#### 选择编译版本

lunch

#### 编译系统

m 整个编译

mm 编译当前目录 没有依赖会失败 m 整编后使用

mmm 指定路径目录进行编译

#### 启动模拟器

emulator

### Android Studio 阅读源码

1. source build/envsetup.sh 配置临时环境变量
2. mmm development/tools/idegen 编译生成 idegen
3. development/tools/idegen/idegen.sh 编译生成 android.ipr 和 android.iml 文件 到源码根目录
4. 编辑 android.iml 文件 排除 不需要的文件夹
5. 在 Android studio 中 打开 android.ipr 文件
6. 修改资源引用的顺序, 用来点击源码跳转到本地目录

### Android 内置 APP

1. 准备资源文件:
   * 在 packages/apps 目录 新建自己的 APP 资源文件
2. 在系统配置中增加自己的 APP 名称
   * 在 build/target/product/generic.mk
   * -generic\_no\_telephony.mk
   * -handheld\_system.mk
   * -handheld\_system\_ext.mk
   * -handheld\_vendor.mk
   * -handheld\_product.mk
   * -telephony.mk
   * -telephony\_system.mk
   * -telephony\_system\_ext.mk
   * -telephony\_vendor.mk
   * -telephony\_product.mk
   * **文件中配置的 PRODUCT\_PACKAGES 中增加自己的 APP (Android.mk 文件中配置的 LOCAL\_MODULE字段名称)**
   * 理论上这里的目录中的任何一个文件增加了都可以, 选择 handheld\_product.mk 文件作为app 载体

#### 没有源码 内置 APK 文件

内置的 apk 文件需要 包含 要运行的设备(虚拟机) 的 cpu 架构

armeabi / armeabi-v7a / arm64-v8a / x86 / x86\_64

# Android.mk 预置 APK   
# 当前目录  
LOCAL\_PATH := $(call my-dir)  
# 构建系统提供 指向以恶搞特殊的 GNU Markfile 类似恢复初始环境  
include $(CLEAR\_VARS)  
# 声明 需要被生成 的 module 名称   
LOCAL\_MODULE := DuLauncher  
LOCAL\_MODULE\_TAGS := optional  
# 此变量包含构建系统生成模块时所用的源文件列表  
LOCAL\_SRC\_FILES := dudu.apk  
  
LOCAL\_OVERRIDES\_PACKAGES := Home Launcher2 Launcher3 Launcher3QuickStep Launcher3Go CarLauncher  
  
LOCAL\_MODULE\_CLASS := APPS  
LOCAL\_MODULE\_SUFFIX := $(COMMON\_ANDROID\_PACKAGE\_SUFFIX)  
  
# 内置成核心应用，也就是内置到system/priv-app目录  
# LOCAL\_PRIVILEGED\_MODULE := true  
  
# 获得apk中 armeabi-v7a 下所有的so  
# 切换到 LOCAL\_PATH 目录下  
# 删除原有 lib  
# 解压  
# define get-all-libraries-module-name-in-subdirs  
  
# $(sort $(shell cd $(LOCAL\_PATH) ; rm -rf lib >/dev/null ; unzip $(LOCAL\_MODULE).apk 'lib/x86\_64/\*.so' -d . >/dev/null ; find -L $(1) -name "\*.so"))  
  
# endef  
LOCAL\_BUILT\_MODULE\_STEM := dudu.apk  
  
# ALL\_LIBRARIES\_MODULE\_NAME := $(call get-all-libraries-module-name-in-subdirs, lib/x86\_64)  
# LOCAL\_PREBUILT\_JNI\_LIBS := $(ALL\_LIBRARIES\_MODULE\_NAME)  
  
# 表示 APK 已经签名  
LOCAL\_CERTIFICATE := PRESIGNED  
  
# copy the apk to system/app/$(LOCAL\_MODULE)  
#$(shell cp $(LOCAL\_PATH)/$(LOCAL\_MODULE).apk $(TARGET\_OUT\_APPS)/$(LOCAL\_MODULE)/$(LOCAL\_MODULE).apk)  
# copy the nativelib to system/lib  
#$(shell cp $(LOCAL\_PATH)/lib/\* $(TARGET\_OUT)/lib)  
  
# 此变量 指向的构建脚本 会收集 在 LOCAL\_XXX 变量中提供的模块的所有相关信息, 以及确定  
include $(BUILD\_PREBUILT)

#### 有源码 内置 APP

配置资源

子目录 JNI 中有需要 配置 Android.mk

jni 模块在 S 以后, 需要指定 Header:

LOCAL\_HEADER\_LIBRARIES := jni\_headers

需要先编译 JNI 成 so

# Android.mk 编译 c++ 共享库  
  
LOCAL\_PATH := $(call my-dir)  
  
include $(CLEAR\_VARS)  
  
# 头文件  
LOCAL\_MODULE\_TAGS := optional  
  
#LOCAL\_LDFLAGS := -llog  
  
LOCAL\_HEADER\_LIBRARIES := jni\_headers  
  
LOCAL\_MODULE := libnative  
  
#LOCAL\_PRODUCT\_MODULE := true  
  
LOCAL\_SRC\_FILES := native-lib.cpp  
  
#LOCAL\_SDK\_VERSION := 21  
  
include $(BUILD\_SHARED\_LIBRARY)

编译项目 成 apk

# Android.mk 有源码的系统内置 app  
LOCAL\_PATH:= $(call my-dir)  
  
#预置Java库========================================================  
  
#因为项目使用了mmkv，先预处理mmkv使得下面的脚本能够引用  
#include $(CLEAR\_VARS)  
#LOCAL\_PREBUILT\_STATIC\_JAVA\_LIBRARIES := mmkv:libs/mmkv-1.0.16.aar  
#include $(BUILD\_MULTI\_PREBUILT)  
  
# ================================================================  
  
include $(CLEAR\_VARS)  
LOCAL\_USE\_AAPT2 := true  
LOCAL\_MODULE\_TAGS := optional  
  
LOCAL\_STATIC\_ANDROID\_LIBRARIES := \  
 androidx.fragment\_fragment \  
 androidx.appcompat\_appcompat \  
 com.google.android.material\_material \  
 androidx.legacy\_legacy-support-core-ui \  
 androidx.core\_core \  
 androidx.legacy\_legacy-support-v13  
  
# 源文件  
LOCAL\_SRC\_FILES := $(call all-java-files-under, src) \  
 $(call all-Iaidl-files-under, aidl) \  
 $(call all-kotlin-files-under, src)  
  
# aapt2  
LOCAL\_USE\_AAPT2 := true  
  
# 需要编译的资源  
LOCAL\_AAPT\_FLAGS += \  
 --auto-add-overlay  
# --extra-packages android.support.v7.appcompat \  
# --extra-packages android.support.v7.recyclerview  
  
# c 文件 拓展名  
#LOCAL\_CPP\_EXTENSION   
  
# Ｃ/C++ 文件  
# LOCAL\_SRC\_FILES += jni/native-lib.cpp  
  
# jni  
LOCAL\_JNI\_SHARED\_LIBRARIES := libnative  
  
# 资源文件  
LOCAL\_RESOURCE\_DIR += $(LOCAL\_PATH)/res  
  
# SDK 版本  
LOCAL\_SDK\_VERSION := current  
#   
# 引用的java库，OkHttp Glide等系统中存在的，系统不存在需要配置先预置Java库  
#LOCAL\_STATIC\_JAVA\_LIBRARIES := \  
# glide  
  
# 指定apk的src目录  
# LOCAL\_SRC\_FILES := \  
# $(call all-java-files-under, src) \  
# $(call all-Iaidl-files-under, aidl)  
  
# 三方 的 so 文件  
# ifeq ($(strip $(TARGET\_ARCH)), arm64)  
# LOCAL\_PREBUILT\_JNI\_LIBS := libs/arm64-v8a/libc++\_shared.so  
# else ifeq ($(strip $(TARGET\_ARCH)), x86\_64)  
# LOCAL\_PREBUILT\_JNI\_LIBS := libs/x86\_64/libc++\_shared.so  
# else ifeq ($(strip $(TARGET\_ARCH)),arm)  
# LOCAL\_PREBUILT\_JNI\_LIBS := libs/armeabi-v7a/libc++\_shared.so  
# else  
# LOCAL\_PREBUILT\_JNI\_LIBS := libs/x86/libc++\_shared.so  
# endif  
  
# AndroidManifest.xml 文件  
LOCAL\_FULL\_LIBS\_MANIFEST\_FILE := $(LOCAL\_PATH)/AndroidManifest.xml  
  
# 要编译成apk的名字  
LOCAL\_PACKAGE\_NAME := LQKSystemApp  
# apk 签名  
LOCAL\_CERTIFICATE := platform  
# 不混淆  
LOCAL\_PROGUARD\_ENABLED := disabled  
  
include $(BUILD\_PACKAGE)  
  
# 执行当前文件子目录下的 mk 文件  
include $(call all-makefiles-under, $(LOCAL\_PATH))

#### 修改后 重新编译系统运行

m

### 自定义系统服务

#### 编辑自己的系统服务

###### 定义 服务相关的文件

**所有文件记得要 以 大驼峰方式命名**

一个 IXXXManager.aidl : 定义服务的功能

frameworks/base/core/java/android/app/IXXXManager.aidl

// IXXXManager.aidl  
package android.app;  
/\*\*  
\* System private API for talking with the activity manager service. This  
\* provides calls from the application back to the activity manager.  
\*  
\* {@hide}  
\*/  
interface IXXXManager {  
 String request(String msg);  
}

一个 XXXManager.java : 提供 Service 的实际 **调用**

frameworks/base/core/java/android/app/XXXManager.java

package android.app;  
import android.annotation.SystemService;  
import android.compat.annotation.UnsupportedAppUsage;  
import android.content.Context;  
import android.os.IBinder;  
import android.os.RemoteException;  
import android.annotation.Nullable;  
import android.os.ServiceManager;  
import android.util.Singleton;  
  
// 这个常量名称跟 Context 中定义的要一致  
@SystemService(Context.XXX\_SERVICE)  
public class XXXManager {  
   
 private Context mContext;  
   
 /\*\*  
 \* @hide  
 \*/  
 public XXXManager() {}  
   
 /\*\*  
 \* @hide  
 \*/  
 public static IXXXManager getService() {  
 return IXXXManagerSingleton.get();  
 }  
  
 @UnsupportedAppUsage  
 private static final Singleton<IXXXManager> IXXXManagerSingleton =  
 new Singleton<IXXXManager>() {  
 @Override  
 protected IXXXManager create() {  
 final IBinder b = ServiceManager.getService(Context.XXX\_SERVICE);  
 final ILXXXManager iXXXManager = IXXXManager.Stub.asInterface(b);  
 return iXXXManager;  
 }  
 };  
   
 @Nullable  
 public String request(@Nullable String msg) {  
 try {  
 return getService().request(msg);  
 } catch (RemoteException e) {  
 e.rethrowFromSystemServer();  
 }  
 return null;  
 }  
}

一个 XXXManagerService.java : 自定义服务的具体 **实现**

frameworks/base/services/core/java/com/android/service/xxx

package com.android.server.xxx;  
  
import android.annotation.Nullable;  
import android.app.IXXXManager;  
import android.os.RemoteException;  
  
public class XXXManagerService extends IXXXManager.Stub {  
 @Override  
 public String request(String msg) throws RemoteException {  
 return "XXXManagerService接收数据:" + msg;  
 }  
}

###### 增加 Context 常量 -> 服务名称

修改Context文件增加自己的常量:

frameworks/base/core/java/android/content/Context.java

/\*\* @hide \*/  
@StringDef(suffix = { "\_SERVICE" }, value = {  
 //...  
 ACTIVITY\_SERVICE,  
 //...  
 // 放在这个 value 哪都行  
 NAME\_SERVICE,  
})  
@Retention(RetentionPolicy.SOURCE)  
public @interface ServiceName {}  
  
// 定义服务的关键字  
public static final String NAME\_SERVICE="service\_name";

###### 注册系统服务 : ServiceManage.java

frameworks/base/services/java/com/android/server/SystemServer.java

import com.android.server.xxx.XXXManagerService;  
  
//...  
// 这是 8.0  
// private void startOtherServices(){  
// Android 12  
private void startOtherServices(@NonNull TimingsTraceAndSlog t) {  
   
 //...  
   
 t.traceBegin("StartXXXManagerService");  
 ServiceManager.addService(Context.XXX\_SERVICE,new XXXManagerService());  
 t.traceEnd();  
   
 //...  
}

###### 注册系统服务获取器: SystemServiceRegistry.java

frameworks/base/core/java/android/app/SystemServiceRegistry.java

import android.app.XXXManager;  
import android.app.IXXXManager;  
  
//...  
  
static{  
 //...  
   
 // 注册自己的服务  
 registerService(Context.XXX\_SERVICE, XXXManager.class,  
 new CachedServiceFetcher<XXXManager>() {  
 @Override  
 public XXXManager createService(ContextImpl ctx) throws ServiceNotFoundException {  
 return new XXXManager();  
 }});  
   
 //...  
}

###### 修改 SELinux 权限

system/sepolicy/private/

system/sepolicy/prebuilts/api/31.0/private/

**这里需要修改所有 api 中的 文件 需要所有编译版本以及更高的版本中的文件**

**service\_contexts : 配置 自定义服务的 selinux 角色**

activity u:object\_r:activity\_service:s0  
# 配置自定义服务selinux角色  
lance u:object\_r:xxx\_service:s0

service.te: 定义服务的类型

#配置自定义服务类型  
type xxx\_service, app\_api\_service, ephemeral\_app\_api\_service, system\_server\_service,  
service\_manager\_type;

untrusted\_app\_all.te : 配置可访问服务的访问权限

#允许所有app使用自定义服务  
allow untrusted\_app\_all xxx\_service:service\_manager find;

###### 更新并重新编译

# 需要先回到 源码根目录  
# 更新api  
make update-api  
# 编译  
m  
# 运行模拟器  
emulator

###### 查看服务是否启动成功

// 列出所有服务 并筛选 出包含 xxx 的服务  
adb shell service list | grep xxx

#### 使用自己的系统服务

###### 双亲委托机制

定义一个同包名的类文件, 使用双亲委托机制先加载系统中的 XXXManager, 实际调用的也都是系统的类和方法

**定义的文件 包名, 类名, 方法 (入参和返回值) 必须和系统文件完全相同**

###### 生成自己的sdk

// 编译自己的 sdk  
make sdk  
  
// 再次运行一个 cmd 的 时候会报 api 文档不匹配  
// 运行 make api-stubs-docs-non-updatable-update-current-api

###### 复制并替换SDK

编译后的系统源码中的文件到 开发 sdk 的目录下

### SEAndroid

#### DAC 和 MAC

DAC

MAC

#### FdBus 编译配置

##### Host ( Ubuntu )

###### protocol

cd ~/workspace  
git clone https://github.com/protocolbuffers/protobuf.git #get protobuf source code  
cd protobuf;git submodule update --init --recursive  
mkdir -p build/install;cd build #create directory for out-of-source build  
cmake -DCMAKE\_INSTALL\_PREFIX=install -DBUILD\_SHARED\_LIBS=1 ../cmake  
make -j4 install #build and install to build/install directory

###### fdbus

cd ~/workspace  
git clone https://github.com/jeremyczhen/fdbus.git #get fdbus source code  
cd fdbus;mkdir -p build/install;cd build #create directory for out-of-source build  
cmake -DSYSTEM\_ROOT=~/workspace/protobuf/build/install -DCMAKE\_INSTALL\_PREFIX=install ../cmake  
PATH=~/workspace/protobuf/build/install/bin:$PATH make #set PATH to the directory where protoc can be found

### FdBus 跨系统通信

#### 搭建 FdBus server 端

编译 probuf