《嵌入式系统》第四次作业

基于 Android 两次实验项目，参考 Android 指导手册，完成下述任务：

1. 针对第一次 Android 实验中的 53 个程序样例 ：

a) 界面布局（5 个）

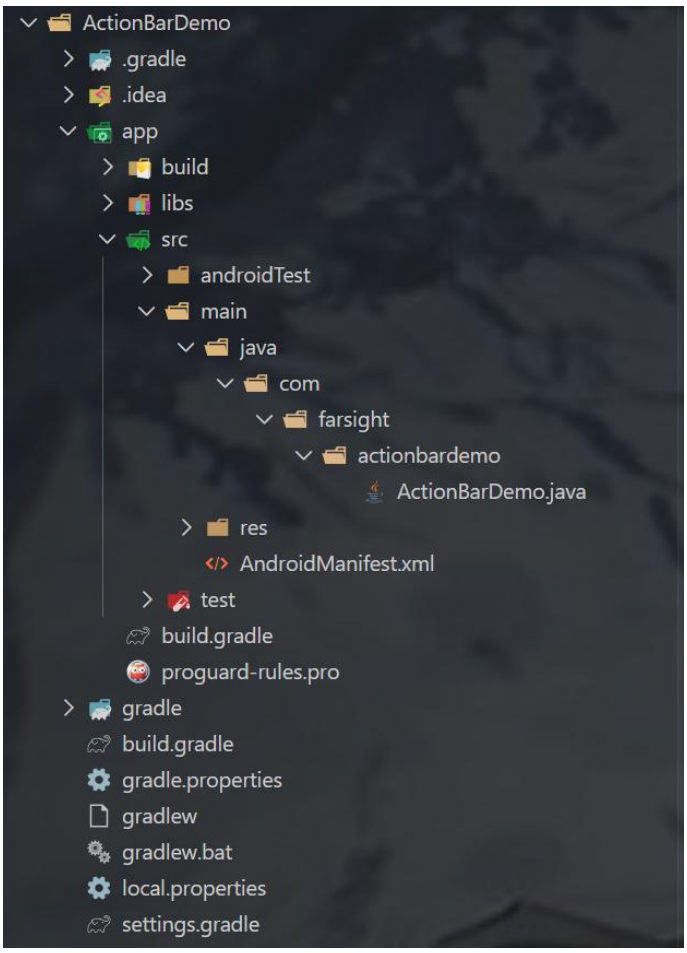
b) 界面控制（5 个）

c) 界面事件（2 个）

d) ActionBar 和菜单（6 个）

以 ActionBarDemo 为例分析：

项目结构如下：



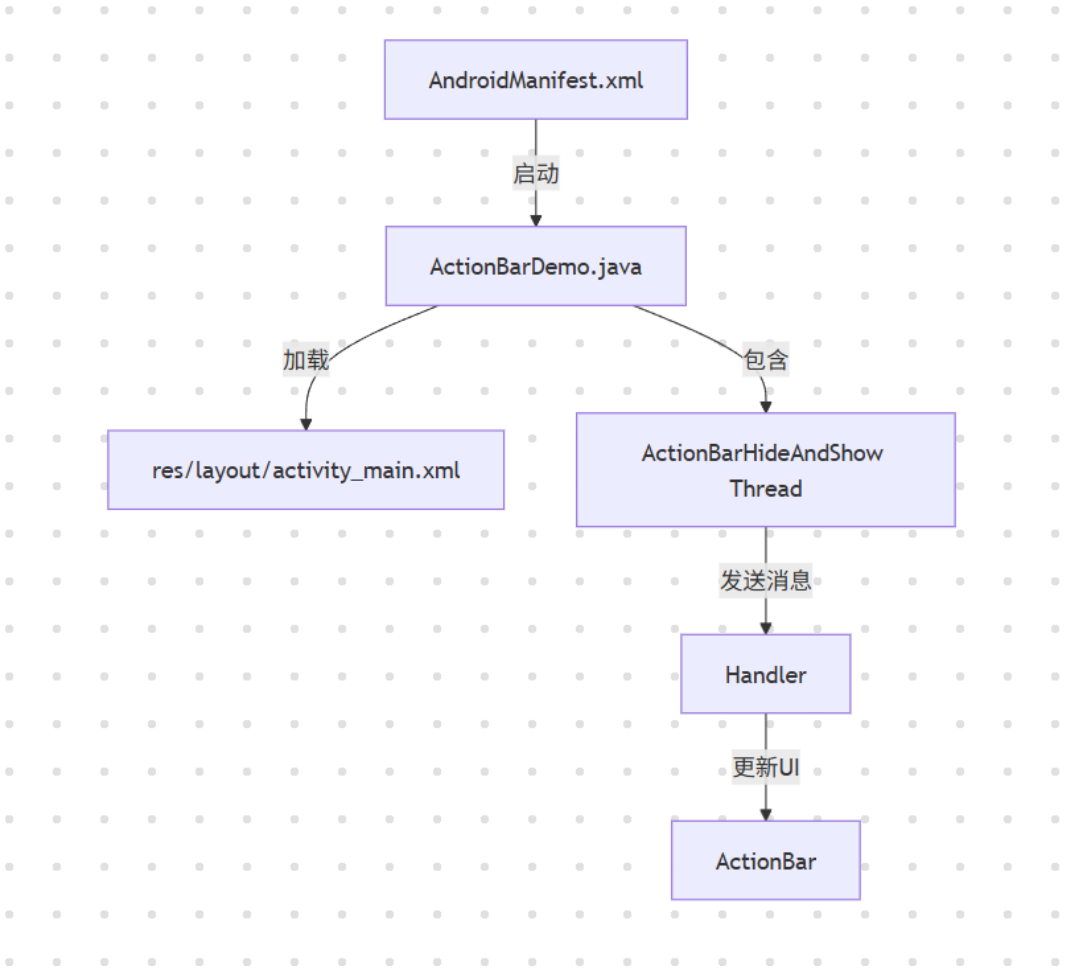
参考 Android 项目标准结构：

AndroidManifest.xml: 配置文件，注册了 ActionBarDemo Activity。

ActionBarDemo.java: 主程序逻辑，包含 Activity 生命周期管理和线程控制。

app/src/main/res/layout/activity_main.xml: 布局文件。

文件的调用关系如下



核心代码分析

涉及的模块 Activity, Handler (消息传递), ActionBar (UI 组件)

定义 Handler 用于处理子线程发送的消息，更新 UI

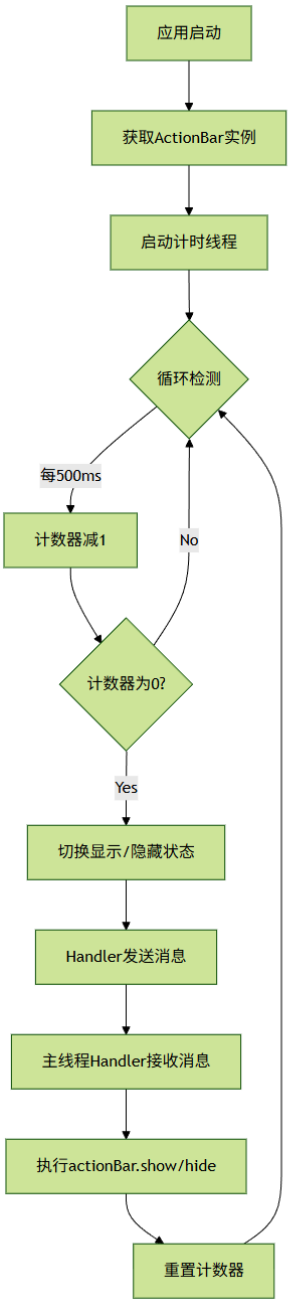
```java
public class ActionBarDemo extends AppCompatActivity
{
    // 定义Handler用于处理子线程发送的消息，更新UI
    @SuppressLint("HandlerLeak")
    private Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            switch (msg.what) {
                case HIDE: // 接收隐藏消息
                    if (actionBar != null) {
                        actionBar.hide(); // 核心：在主线程操作UI组件
                    }
                    break;
                case SHOW: // 接收显示消息
                    if (actionBar != null) {
                        actionBar.show(); // 核心：在主线程操作UI组件
                    }
                    break;
            }
        }
    };
```

```java
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // 获取ActionBar实例，这是Activity提供的系统组件
    actionBar = getSupportActionBar();

    // 开启后台线程
    actionBarHideAndShow = new ActionBarHideAndShow();
    threadOn = true;
    actionBarHideAndShow.start();
}
```

内部类：后台线程，模拟耗时操作或定时任务

```
private class ActionBarHideAndShow extends Thread
{
    @Override
    public void run() {
        while (threadOn) {
            try {
                sleep(500); // 每0.5秒检查一次
            } catch (InterruptedException e) { ... }
            count--;
            if (count == 0) { // 累计5秒
                count = 10;
                // 切换状态
                old = old == HIDE ? SHOW : HIDE;
                // 核心：通过Handler发送消息给主线程，不能直接在这里操作UI
                handler.sendEmptyMessage(old);
            }
        }
    }
}
```

方法调用的逻辑流程图

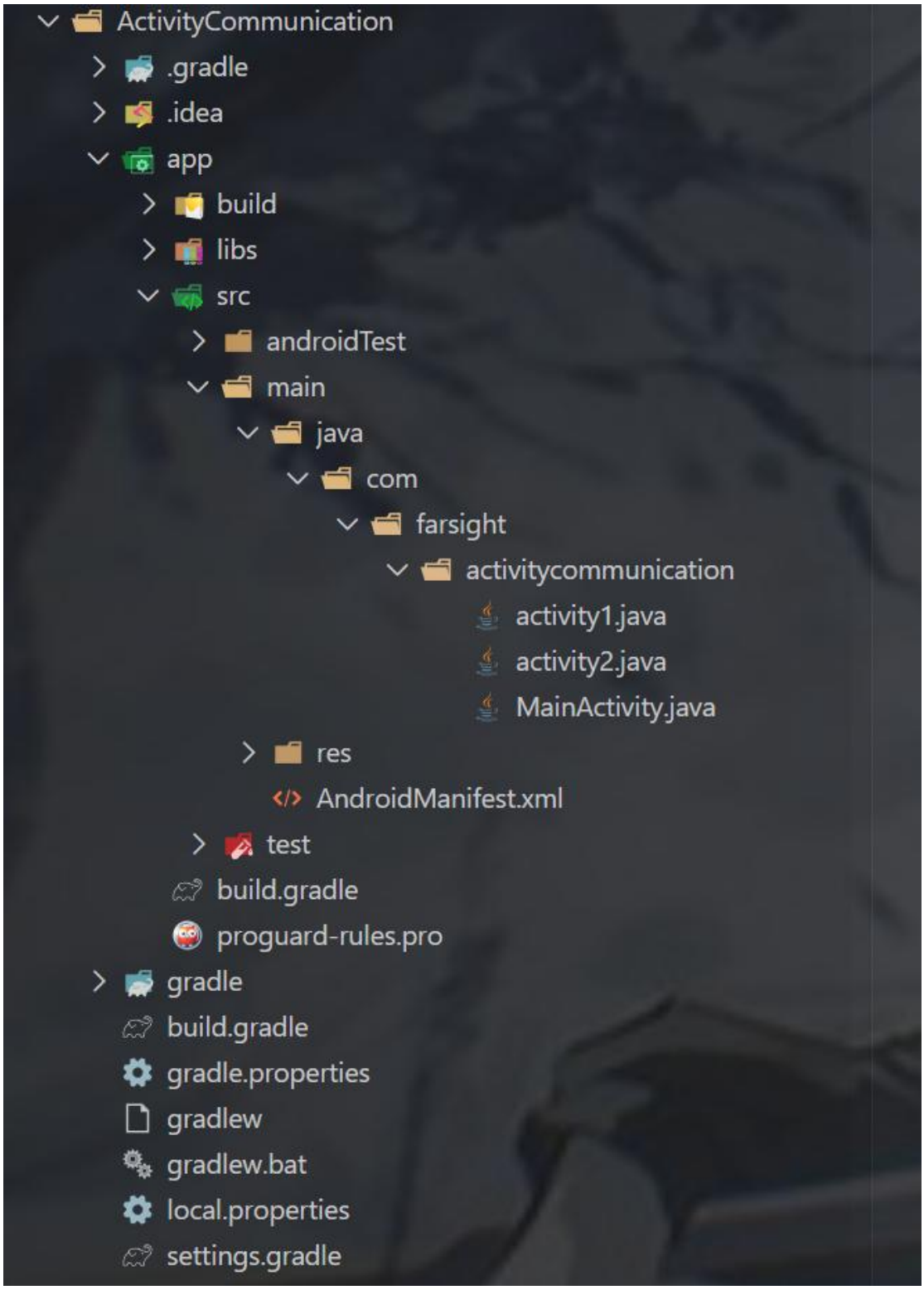


e) 9PatchDemo（1 个）

f) 组件间通讯（3 个）

以 ActivityCommunication 为例分析：

项目结构如下：



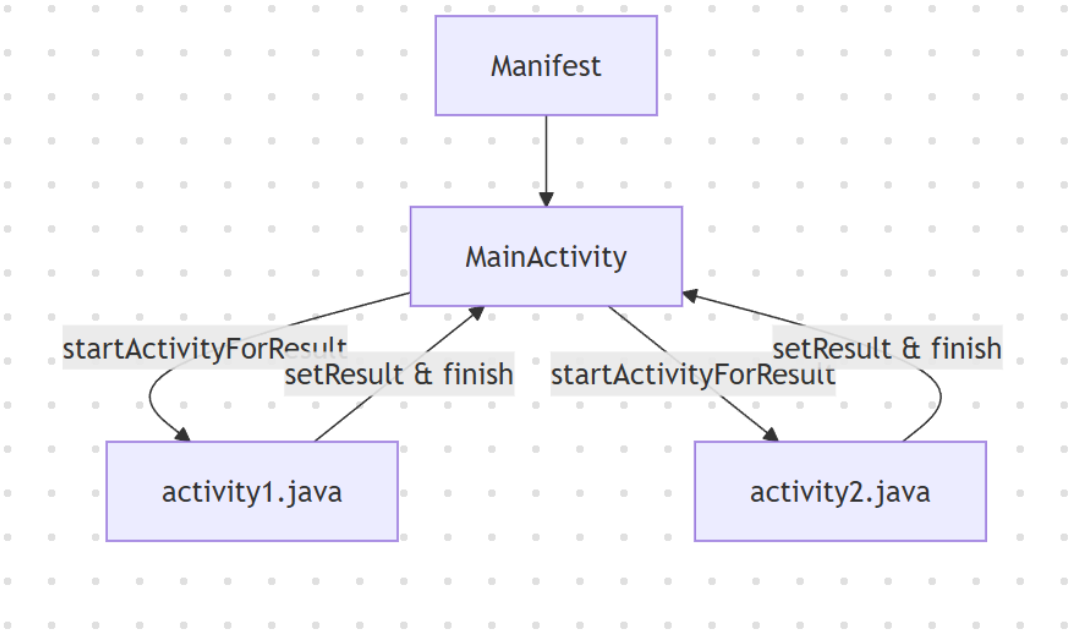
AndroidManifest.xml: 注册了 MainActivity, activity1, activity2。

MainActivity.java: 主界面，负责启动子 Activity 并接收结果。

activity1.java: 子界面 1，演示返回数据。

activity2.java：子界面 2，演示仅返回状态。

文件间的调用关系如下



代码剖析如下：

涉及模块: Activity, Intent (显式意图), Bundle (数据载体)

在 MainActivity.java 中

核心是按钮点击事件的处理

以下这个函数是用来启动子 Activity

```java
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    textView = (TextView)findViewById(R.id.textshow);
    Button button1 = (Button) findViewById(R.id.button1);
    Button button2 = (Button) findViewById(R.id.button2);

    button1.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            // 核心：创建显式Intent，指定从当前Context跳转到activity1类
            Intent intent = new Intent(MainActivity.this, activity1.class);
            // 核心：启动Activity并请求返回结果，请求码为SUBACTIVITY1
            startActivityForResult(intent, SUBACTIVITY1);
        }
    });

    button2.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent intent = new Intent(MainActivity.this,activity2.class);
            startActivityForResult(intent,SUBACTIVITY2);
        }
    });
}
```

然后在回调方法里接收子 Activity 返回的结果

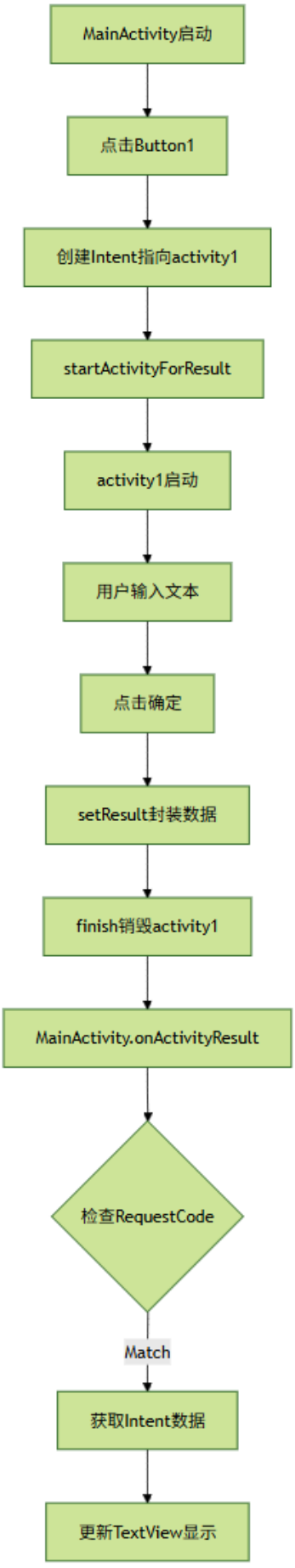
```java
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
    super.onActivityResult(requestCode, resultCode, data);

    switch(requestCode){
        case SUBACTIVITY1:// 辨别是哪个请求返回的
            if (resultCode == RESULT_OK){// 辨别处理结果状态
                // 核心：从返回的Intent中解析数据
                Uri uriData = data.getData();
                textView.setText(uriData.toString());
            }
            break;
        case SUBACTIVITY2:
            break;
    }
}
```

最后确认按钮点击事件

在 activity1.java 里

```java
public class activity1 extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity1);
        final EditText editText = (EditText)findViewById(R.id.text1);
        Button btnOK = (Button)findViewById(R.id.button1);
        Button btnCancel = (Button)findViewById(R.id.button2);

        btnOK.setOnClickListener(new View.OnClickListener(){
            public void onClick(View view){
                String uriString = editText.getText().toString();
                Uri data = Uri.parse(uriString);
                // 核心：创建Intent封装返回数据
                Intent result = new Intent(null, data);
                // 核心：设置结果码和数据Intent
                setResult(RESULT_OK, result);
                // 核心：销毁当前Activity，控制权返回给上一个Activity
                finish();
            }
        });

        btnCancel.setOnClickListener(new View.OnClickListener(){
            public void onClick(View view){
                setResult(RESULT_CANCELED, null);
                finish();
            }
        });
    }
}
```

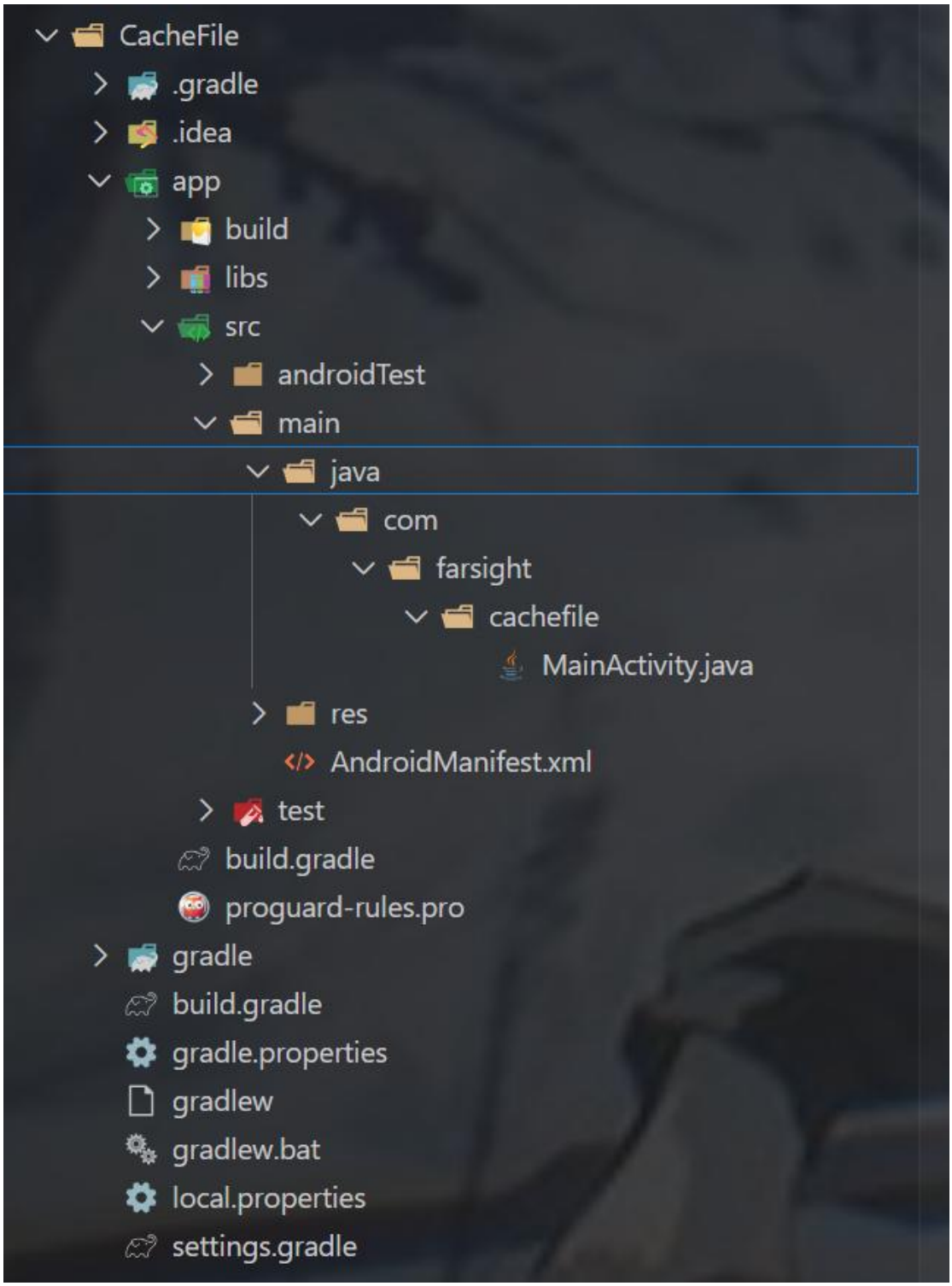
程序运行的逻辑流程图如下：


MainActivity启动
点击Button1
创建Intent指向activity1
startActivityForResult
activity1启动
用户输入文本
点击确定
setResult封装数据
finish销毁activity1
MainActivity.onActivityResult
检查RequestCode
Match
获取Intent数据
更新TextView显示

g) 数据库系统与访问（8 个）

以 CacheFile 为例分析：

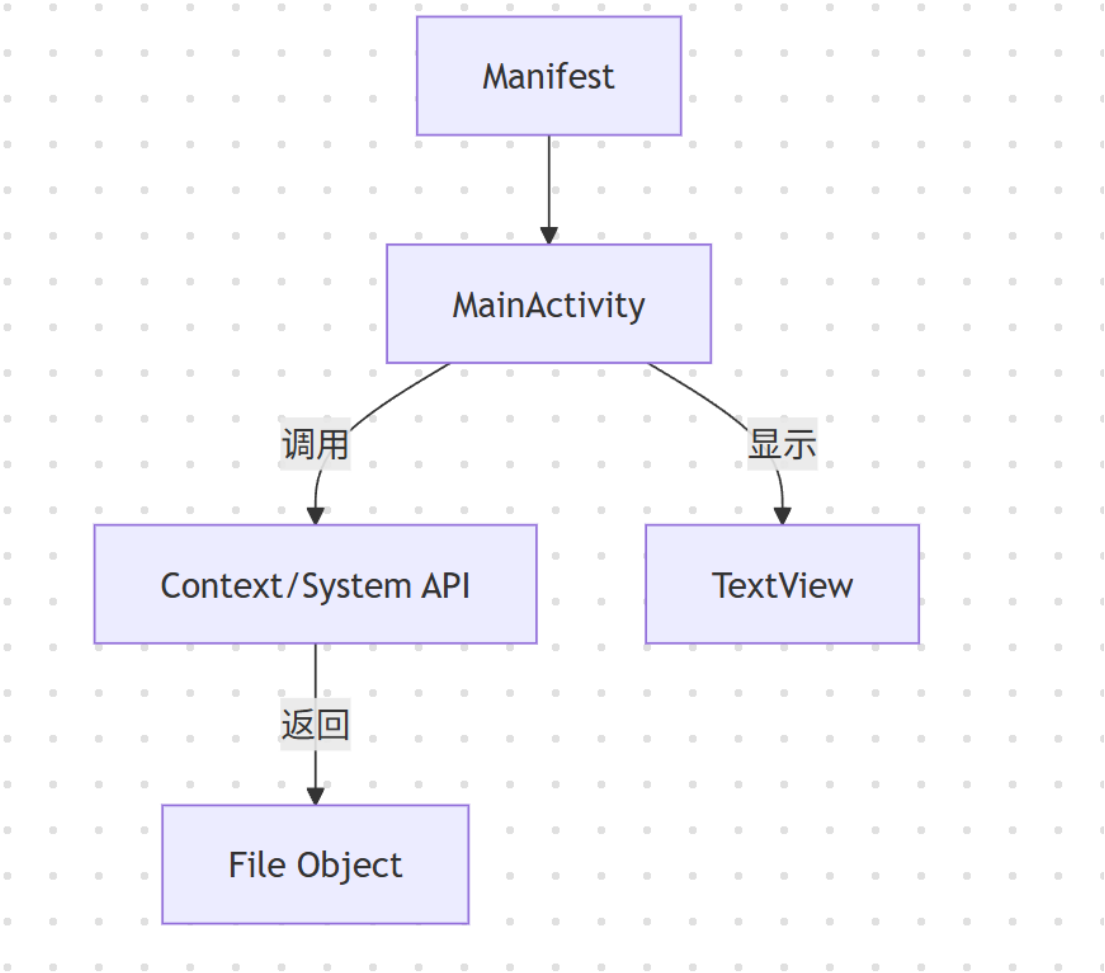
工程文件结构如下：



AndroidManifest.xml：注册 MainActivity。

MainActivity.java: 获取并显示缓存路径。

文件调用关系图如下



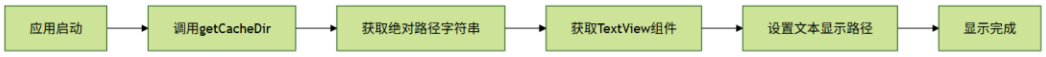
核心代码剖析：

涉及模块: Activity, Context (环境上下文), File (Java IO)

核心就是获取应用程序的内部缓存目录，然后拼接显示出来

```java
public class MainActivity extends AppCompatActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 核心：Context方法，获取应用程序专属的内部缓存目录
        // 路径通常为 /data/user/0/com.farsight.cachefile/cache
        File f = this.getCacheDir();

        // 核心：文件路径字符串拼接
        String path = f.getParent() + java.io.File.separator + f.getName();
        TextView tv = (TextView) findViewById(R.id.textView1);
        tv.setText(path);
    }
}
```

程序运行的逻辑流程图如下

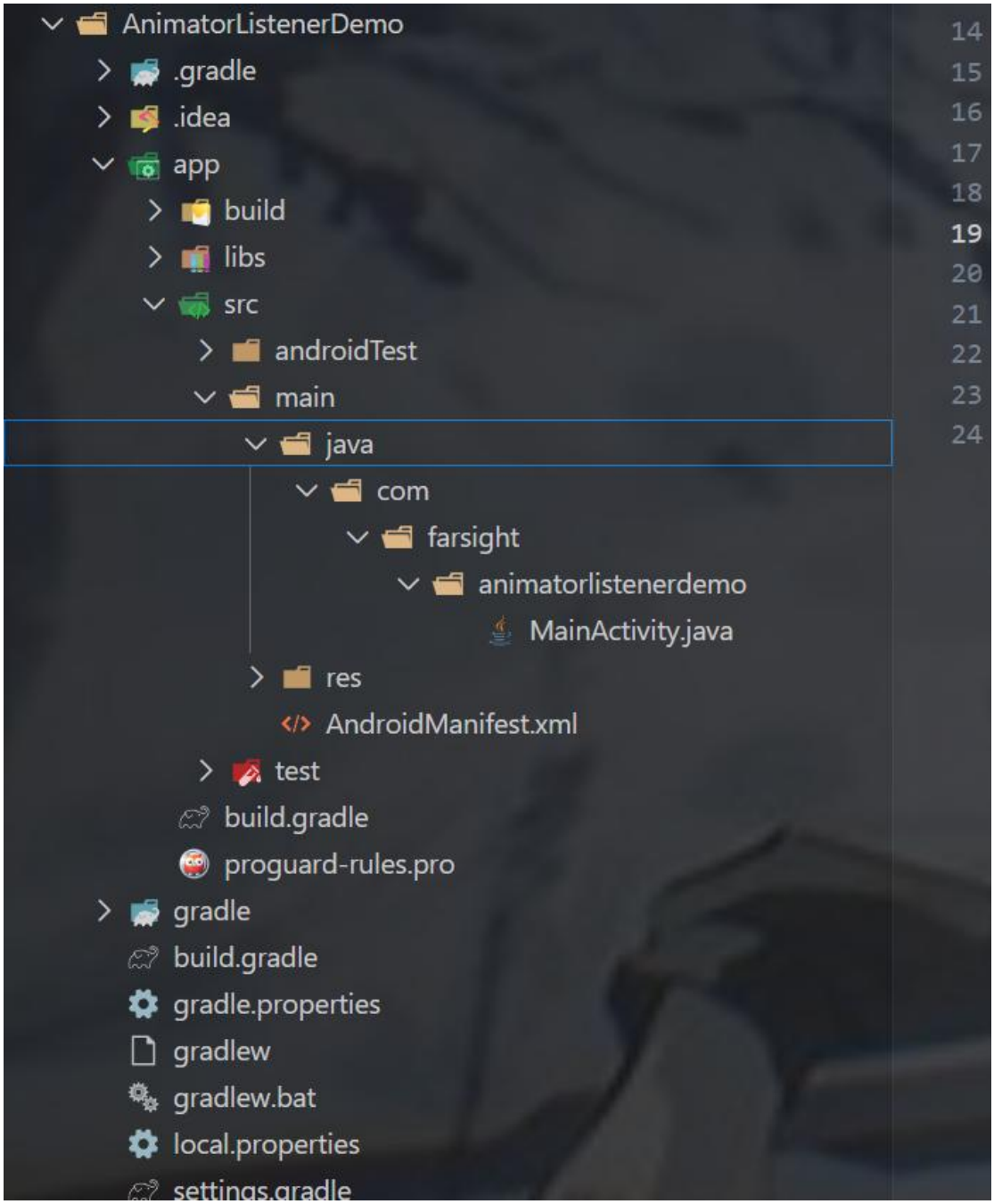


h) 多媒体开发（3 个）

i) 图形图像（13 个）

以 AnimatorListenerDemo 为例分析：

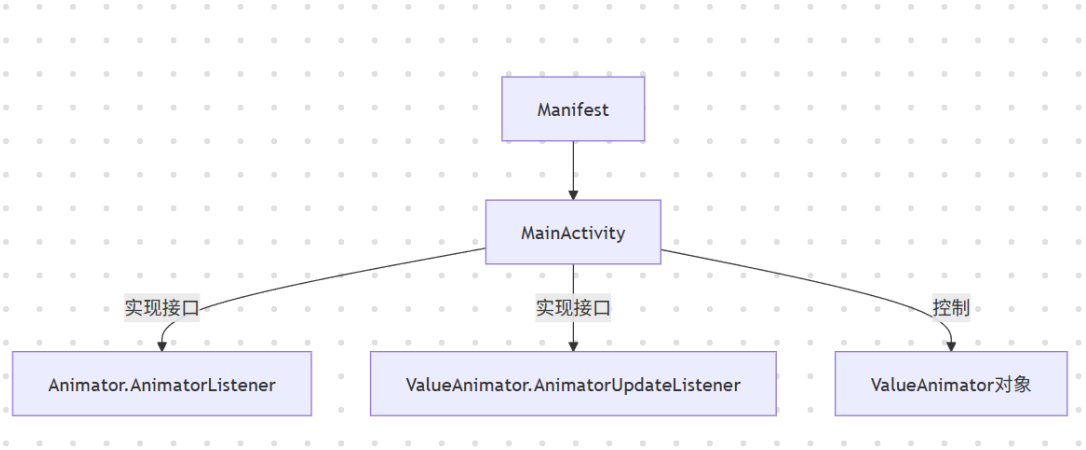
项目工程文件结构如下



AndroidManifest.xml: 注册 MainActivity。

MainActivity.java: 实现动画逻辑及多个监听接口。

文件之间的调用关系图如下



核心代码剖析

涉及模块: Activity, ValueAnimator (属性动画核心类)

对 MainActivity.java 中的类进行剖析

Activity 实现了多个动画监听接口，方便直接处理回调

```
public class MainActivity extends AppCompatActivity implements
        Animator.AnimatorListener, ValueAnimator.AnimatorUpdateListener,
        View.OnClickListener, Animator.AnimatorPauseListener
{
```

主监听函数

```java
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    txv = (TextView) findViewById(R.id.txv);

    // 核心：创建属性动画，值从0.0变化到10.0
    animator = ValueAnimator.ofFloat(0f, 10f);
    animator.setDuration(10000);// 持续10秒
    //更新事件监听 注册监听器
    animator.addUpdateListener(this);// 监听数值变化
    //暂停事件监听
    animator.addPauseListener(this);
    //继承自Animator的动画监听
    animator.addListener(this);// 监听开始/结束
    findViewById(R.id.btn_cancel).setOnClickListener(this);
    findViewById(R.id.btn_resume).setOnClickListener(this);
    findViewById(R.id.btn_pause).setOnClickListener(this);
    findViewById(R.id.btn_stop).setOnClickListener(this);
    findViewById(R.id.btn_start).setOnClickListener(this);
}
```

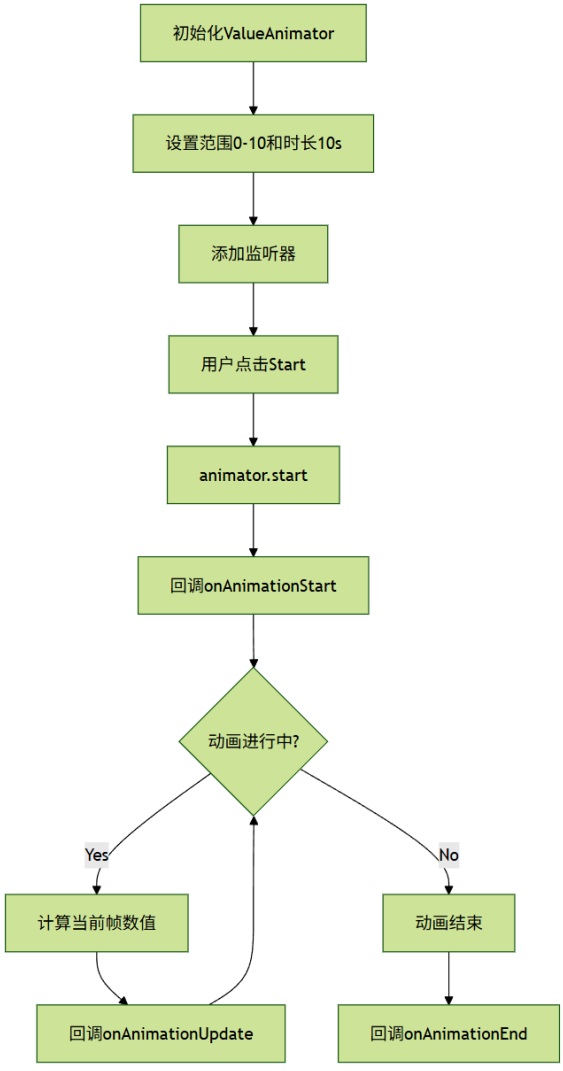
接口实现

```java
// 核心：实现AnimatorListener接口方法
@Override
public void onAnimationStart(Animator animation)
{
    if (animation == this.animator)
    {
        Log.d(TAG, "onAnimationStart");
    }
}
```

播放日志打印

```java
// 核心：实现AnimatorUpdateListener接口方法
// 动画每播放一帧，此方法被调用一次
@Override
public void onAnimationUpdate(ValueAnimator animation)
{
    // 获取当前动画的值并打印日志
    Log.d(TAG, "onAnimationUpdate: " + animation.getAnimatedValue());
    if (animation == this.animator)
    {
        Log.d(TAG, "onAnimationUpdate: " + animation.getAnimatedValue());
    }
}
```

程序工作逻辑流程图

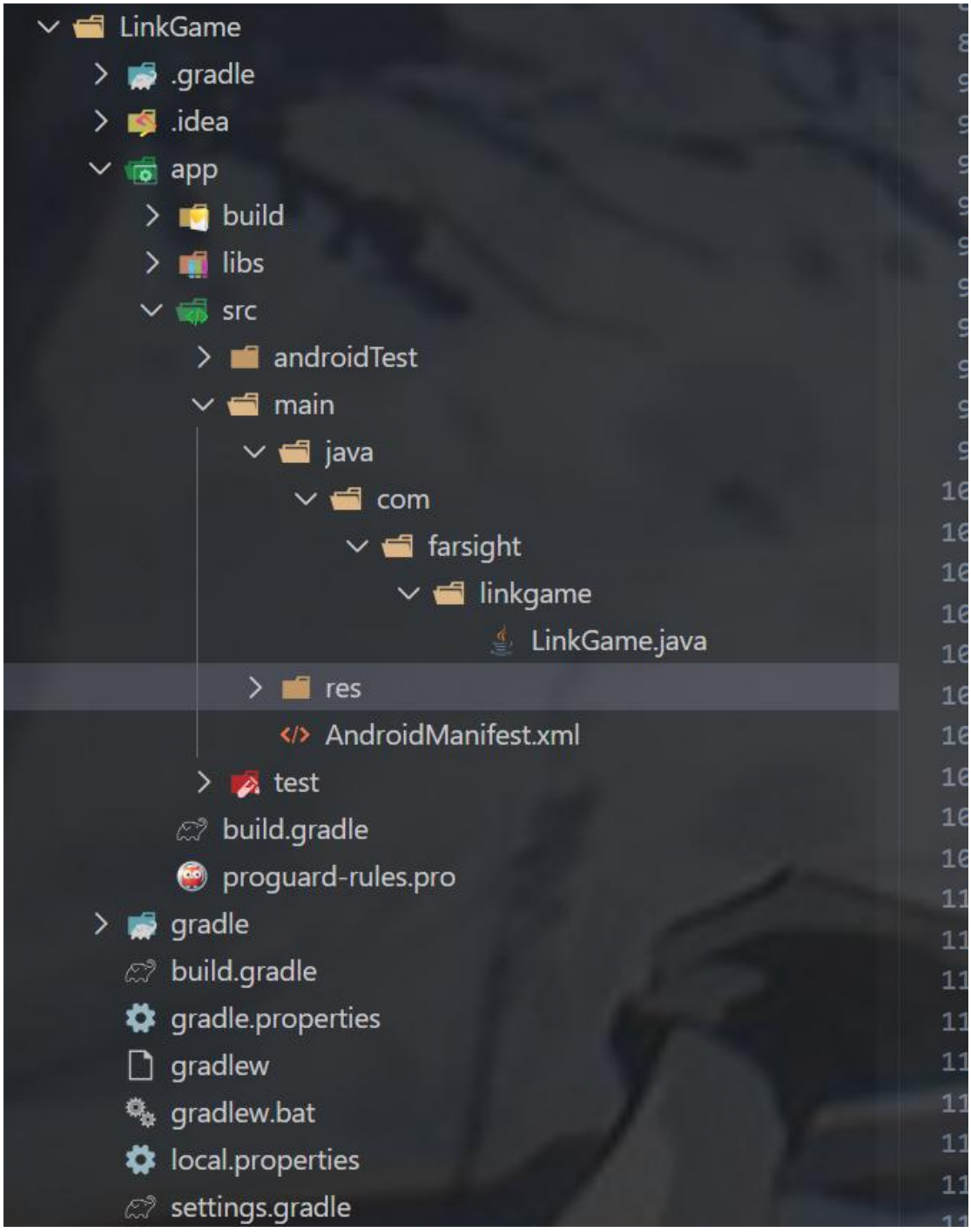


j) 网络通信（2 个）

k) Android 应用项目（5 个）

以 LinkGame 为例分析：

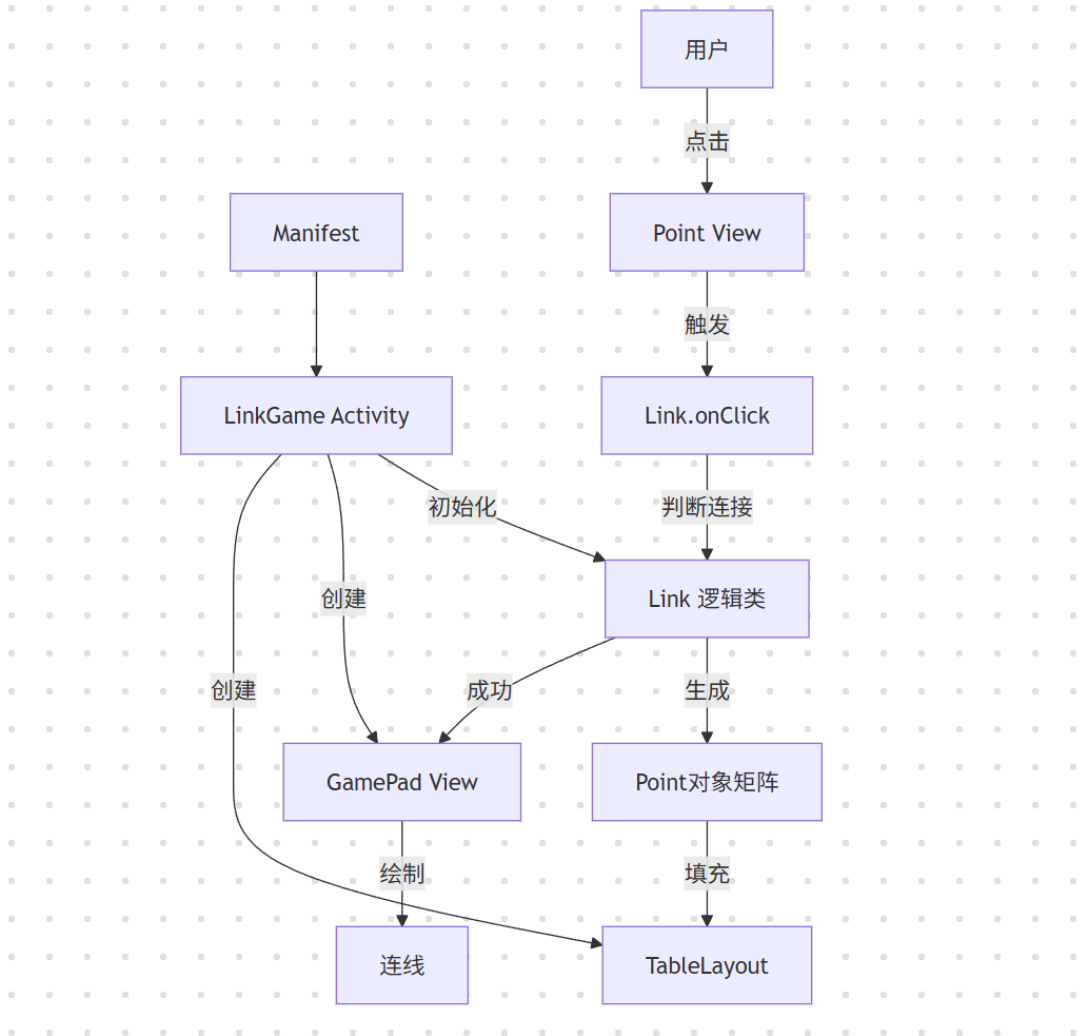
工程结构如下



AndroidManifest.xml: 注册 LinkGame Activity。

LinkGame.java: 包含主 Activity 以及 Point (方块类), Link (逻辑类), GamePad (绘图类)。

文件调用关系图如下



核心代码剖析如下

涉及模块: Activity, View (自定义 View), Canvas (绘图), Layout (动态布局)

LinkGame.java

核心就是俩个类，一个是判断两个点是否可以连接的 LinkPnt 函数，另一个就是绘制界面的 OnDraw 函数

```
public boolean LinkPnt(Point P1, Point P2){
```

逻辑：先判断是否在同一直线(LineX，LineY)，再判断是否可以通过一个转弯连接，最后判断两个转弯

示例：判断是否可以通过“先纵向再横向”的一个转弯连接

(P1.x, P1.y+1) -> (P1.x, P2.y) 纵向扫描

(P1.x, P2.y) -> (P2.x, P2.y) 横向扫描

```
//P1下方1点（y+1）先纵向再横向是否可连接。（因为起点P1不为空，所以检测其下方一点）
if( LineY(P1.x, (P1.y+1), P2.y) && LineX(P1.x, P2.y, P2.x) ) {
    line[0][0] = xPadding + LinkGame.PIC_SIZE * P1.x;
    line[0][1] = yPadding + LinkGame.PIC_SIZE * P1.y;
    if(p1.x != p2.x){
        line[1][0] = xPadding + LinkGame.PIC_SIZE * P1.x;
        line[1][1] = yPadding + LinkGame.PIC_SIZE * P2.y;
        line[2][0] = xPadding + LinkGame.PIC_SIZE * P2.x;
        line[2][1] = yPadding + LinkGame.PIC_SIZE * P2.y;
    }else{
        line[1][0] = xPadding + LinkGame.PIC_SIZE * P2.x;
        line[1][1] = yPadding + LinkGame.PIC_SIZE * P2.y;
    }
    return true;
}
```

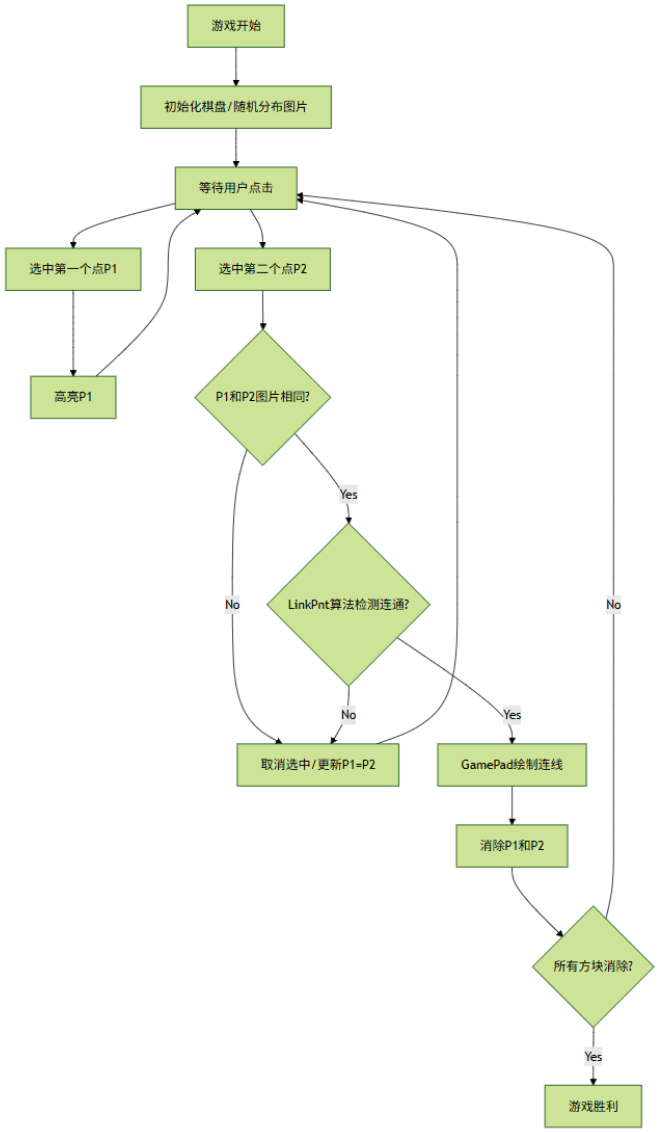
其他路径检测逻辑大同小异

GamePad 类

通过反复绘制来实现交互的效果

```
class GamePad extends View
{
    private int[][] point = new int[4][2];
    private Paint paint= new Paint();
    public GamePad(Context context) {
        super(context);
        paint.setColor(Color.WHITE);
        paint.setStrokeWidth(1);
        // TODO Auto-generated constructor stub
    }
    public void setLine(int[] p1, int[] p2, int[] p3, int[] p4){
        point[0] = p1;
        point[1] = p2;
        point[2] = p3;
        point[3] = p4;
    }
    protected void onDraw(Canvas canvas) {
        // 核心：在Canvas上绘制连接线
        if(point[0][0] != 0 && point[1][0] != 0)
            canvas.drawLine(point[0][0], point[0][1], point[1][0], point[1][1], paint);
        if(point[1][0] != 0 && point[2][0] != 0)
            canvas.drawLine(point[1][0], point[1][1], point[2][0], point[2][1], paint);
        if(point[2][0] != 0 && point[3][0] != 0)
            canvas.drawLine(point[2][0], point[2][1], point[3][0], point[3][1], paint);
        // 绘制完后清空坐标，避免残留
        for(int i = 0; i < 4; i++)
            for(int j = 0; j < 2; j++){
                point[i][j] = 0;
            }
        invalidate();// 请求重绘
    }
}
```

程序运行的逻辑流程图

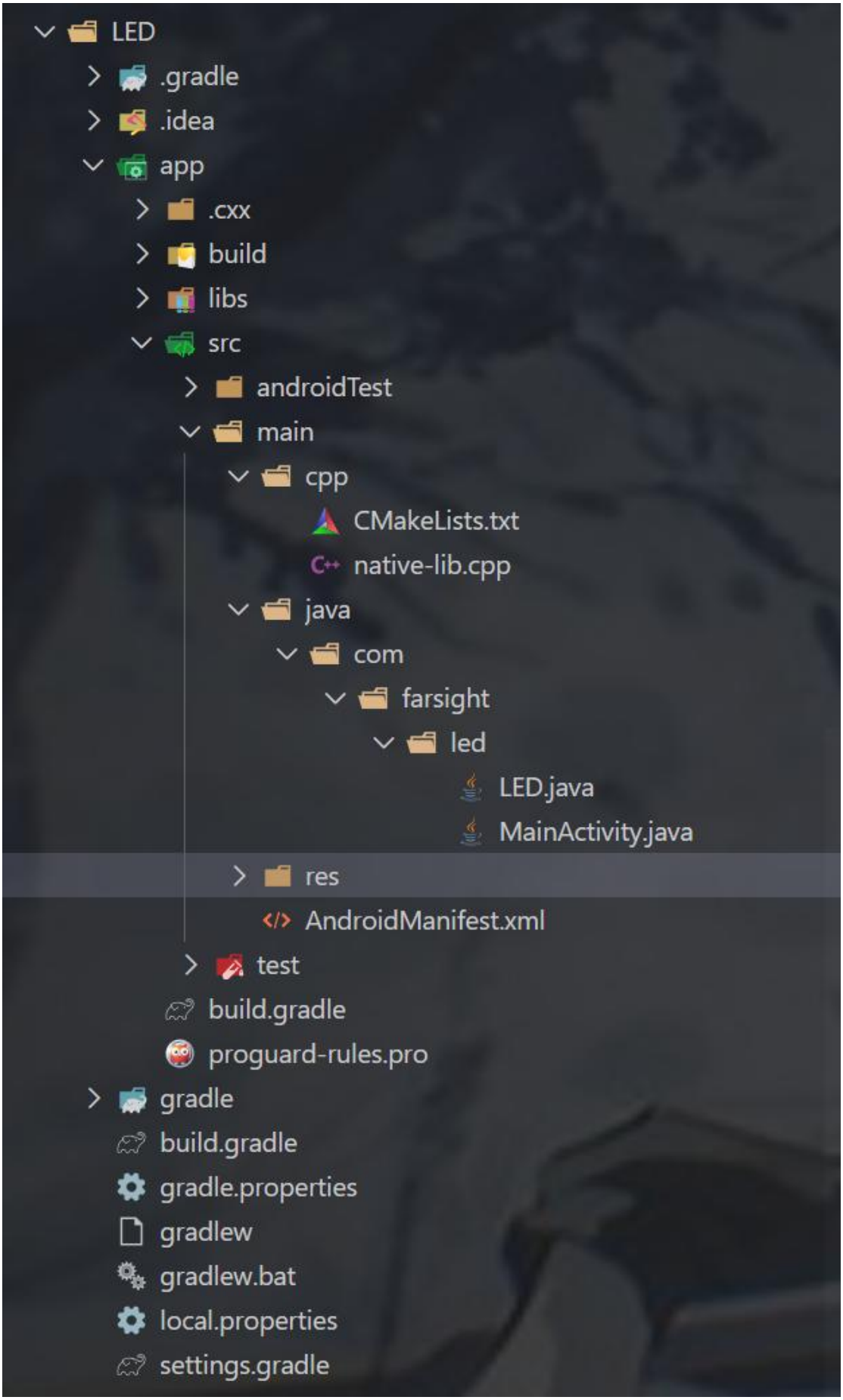


从 d)中任选 1 个+f)中任选 1 个+g)中任选一个+i)中任选 1 个+k)中任选 1 个案例，共计 5 个案例，参考 Android 指导手册第 6 章 （程序设计基础中的 6.1.1Android 项目目录结构.和 6.1.2Android 应用解析），进行上述案例的工程文件结构以及核心代码中对于不同模块（activity,intent,content provider, service）的实际应用剖析。

2. **针对第二次 Android 实验的 4 个实验样例：**

a) LED 灯

工程文件结构如下


LED
.gradle
.idea
app
.cxx
build
libs
src
androidTest
main
cpp
CMakeLists.txt
native-lib.cpp
java
com
farsight
led
LED.java
MainActivity.java
res
AndroidManifest.xml
test
build.gradle
proguard-rules.pro
gradle
build.gradle
gradle.properties
gradlew
gradlew.bat
local.properties
settings.gradle

MainActivity.java: 主界面逻辑，处理按钮点击事件。

LED.java: JNI 接口类，声明 native 方法并加载 .so 库。

native-lib.cpp: C++ 实现代码，调用 Linux 系统调用（open, ioctl）操作设备节点 /dev/leds_ctl。

AndroidManifest.xml: 权限和 Activity 注册。

MainActivity.java 如何驱动外设:

实例化对象: 创建 LED 类的实例 led。

事件监听: 为两个 ImageButton 设置 OnClickListener。

调用流程: 当用户点击按钮时，先判断当前灯的状态，然后依次调用 led.open() 打开设备，led.LedOn1() / led.LedOff1() 控制亮灭，最后调用 led.close() 关闭设备。

UI 反馈: 同时更新按钮的背景图片（setBackground）以显示开关状态。

驱动流程图如下

代码剖析如下：

配置文件方面：

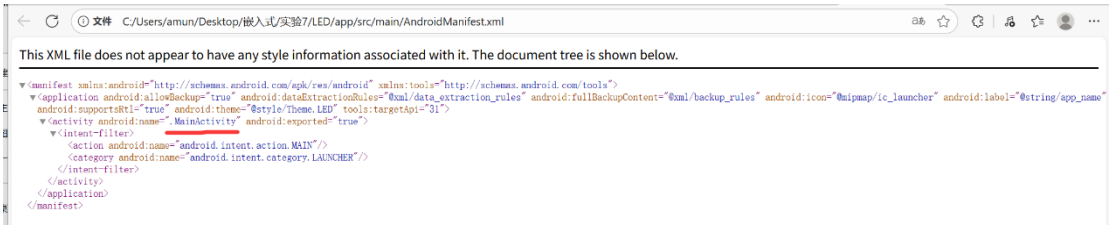
CMakeLists.txt（Native 库构建配置）

将 native-lib.cpp 编译为名为 libled.so 的共享库，供 Java 层通过

System.loadLibrary("led")加载，同时链接 Android 系统的 log 库，支持在 C++ 中打印调试日志

```
# For more information about using CMake with Android Studio, read the
# documentation: https://d.android.com/studio/projects/add-native-code.html

# Sets the minimum version of CMake required to build the native library.

cmake_minimum_required(VERSION 3.22.1)

# Declares and names the project.

project("led")

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
        led

        # Sets the library as a shared library.
        SHARED

        # Provides a relative path to your source file(s).
        native-lib.cpp)

# Searches for a specified prebuilt library and stores the path as a
# variable. Because CMake includes system libraries in the search path by
# default, you only need to specify the name of the public NDK library
# you want to add. CMake verifies that the library exists before
# completing its build.

find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)

# Specifies libraries CMake should link to your target library. You
# can link multiple libraries, such as libraries you define in this
# build script, prebuilt third-party libraries, or system libraries.

target_link_libraries( # Specifies the target library.
        led

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib})
```

AndroidManifest.xml 中 MainActivity 作为应用启动后的首个界面，负责 UI 展示与交互



Java 层代码分析：

LED.java 封装了所有硬件操作的 Native 方法

LED.java

JDK"Android Studio default JDK"缺失

```java
package com.farsight.led;

public class LED
{
    static
    {
        System.loadLibrary("led");
    }
    public native int open();
    public native int close();
    public native int LedOn1();
    public native int LedOff1();
    public native int LedOn2();
    public native int LedOff2();
}
```

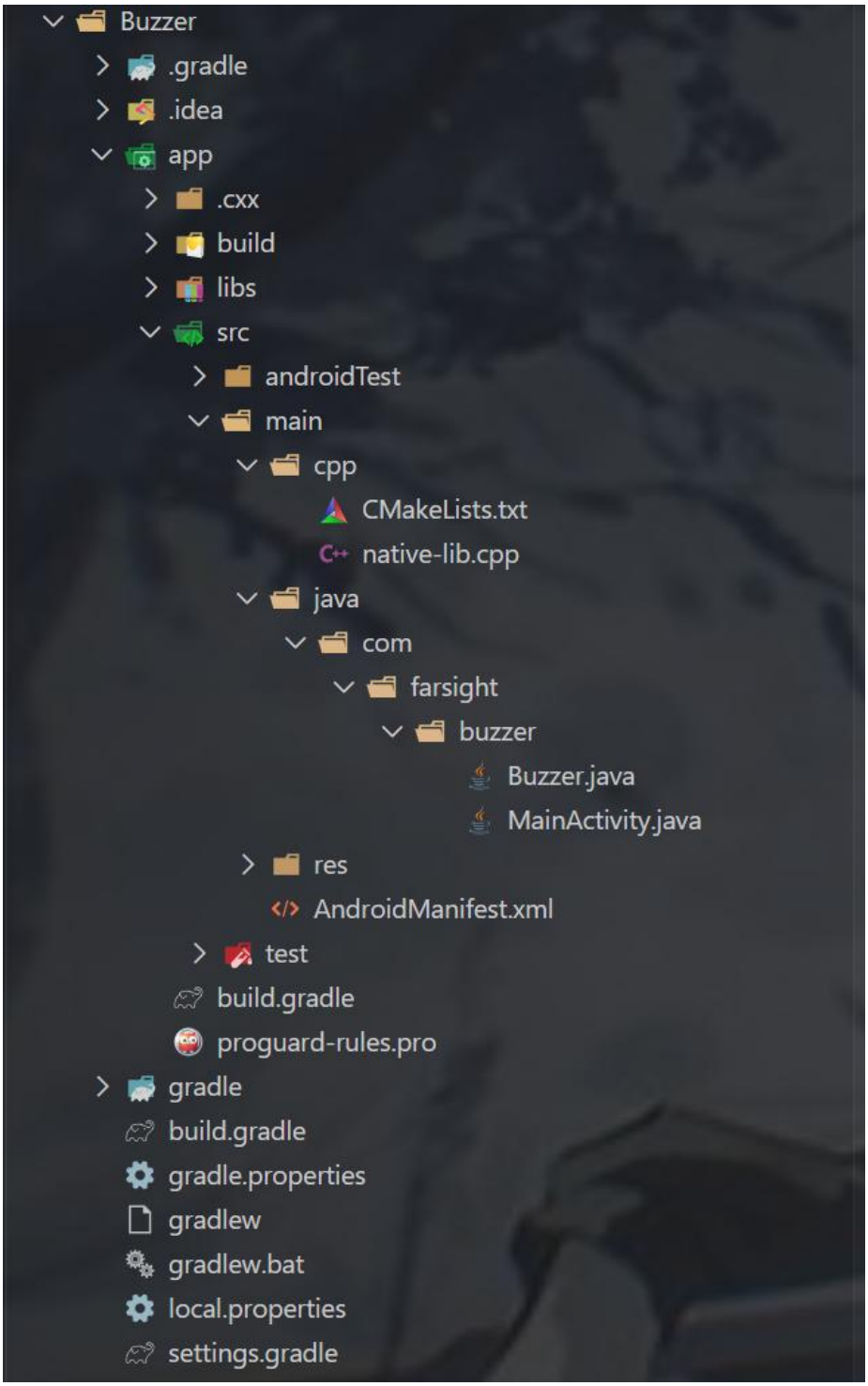
MainActivity.java 负责 UI 初始化、点击事件处理、LED 状态管理

```java
public class MainActivity extends AppCompatActivity {  0 个用法
    LED led = new LED();  // 实例化JNI接口类  0 个用法
    // 记录两颗LED的当前状态（默认关闭）
    boolean IsLight_1_On = false;  0 个用法
    boolean IsLight_2_On = false;  0 个用法
    private ImageButton LedButton_1;  // LED1控制按钮  0 个用法
    private ImageButton LedButton_2;  // LED2控制按钮  0 个用法

    @Override  0 个用法
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);  // 加载UI布局
        // 初始化按钮控件（通过id关联布局文件中的ImageButton）
        LedButton_1 = (ImageButton)findViewById(R.id.buttonOne);
        // LED1按钮点击事件
        LedButton_1.setOnClickListener(new View.OnClickListener() {
            @Override  0 个用法
            public void onClick(View view) {
                if (IsLight_1_On) {  // 当前亮→切换为灭
                    ((ImageButton)view).setBackground(getDrawable(R.drawable.pic_bulboff));
                    led.open();       // 打开设备
                    led.LedOff1();    // 关闭LED1
                    led.close();      // 关闭设备
                } else {  // 当前灭→切换为亮
                    ((ImageButton)view).setBackground(getDrawable(R.drawable.pic_bulbon));
                    led.open();       // 打开设备
                    led.LedOn1();     // 开启LED1
                    led.close();      // 关闭设备
                }
                IsLight_1_On = !IsLight_1_On;  // 切换状态标记
            }
        });
```

native-lib.cpp 通过 JNI 技术实现 Java 层声明的 Native 方法，直接与 Linux 内核驱动交互，控制 LED 硬件

b) 蜂鸣器

实验工程结构如下



MainActivity.java: 主界面，包含“鸣叫”和“停止”按钮。

Buzzer.java: JNI 接口类，加载 libbuzzer.so。

native-lib.cpp: C++ 实现，操作设备节点 /dev/buzzer_ctl。

MainActivity.java 如何驱动外设:

事件处理: 实现了 View.OnClickListener 接口。

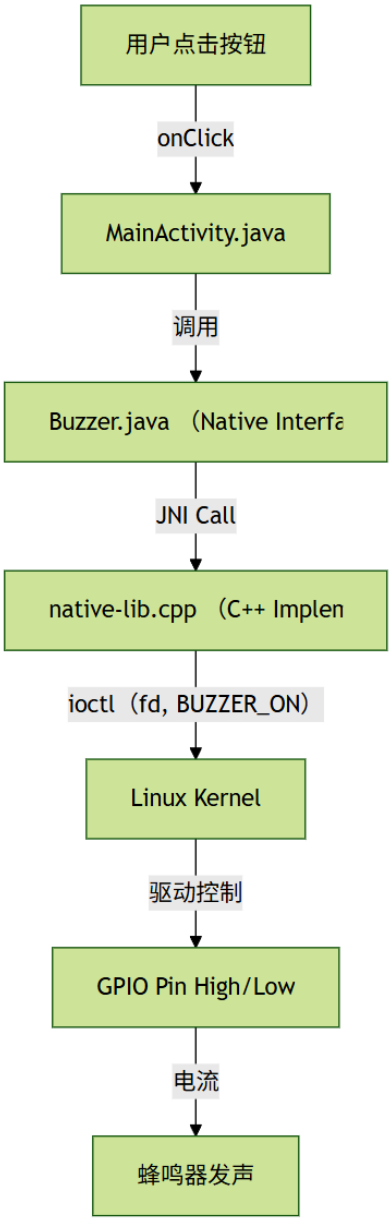
控制逻辑:

点击 button1 (Start): 调用 buzzer.open() -> buzzer.BuzzerOn() -> buzzer.close()。

点击 button2 (Stop): 调用 buzzer.open() -> buzzer.BuzzerOff() -> buzzer.close()。

错误处理: 如果 open() 返回 -1，弹出 Toast 提示“设备打开失败”。

程序驱动流程逻辑图如下

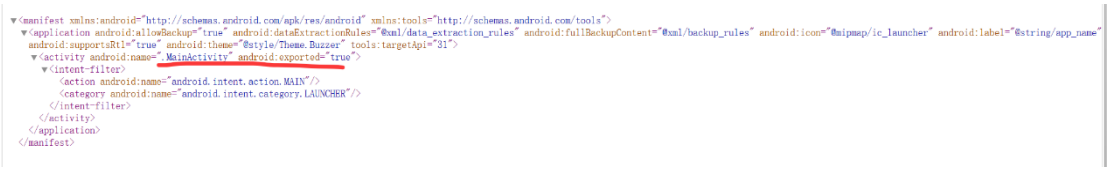


代码剖析

CMakeLists.txt 作为 Native 层编译的核心配置文件，其逻辑与 LED 实验一致，仅针对蜂鸣器项目调整了库名和项目名

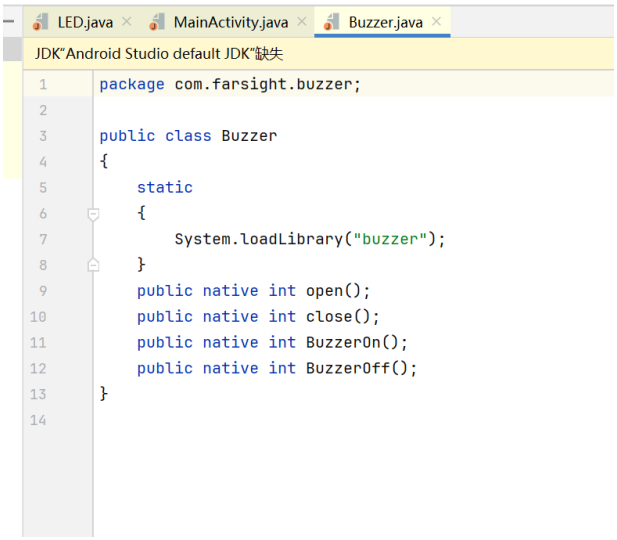
```
# For more information about using CMake with Android Studio, read the
# documentation: https://d.android.com/studio/projects/add-native-code.html

# Sets the minimum version of CMake required to build the native library.

cmake_minimum_required(VERSION 3.22.1)

# Declares and names the project.

project("buzzer")

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
        buzzer

        # Sets the library as a shared library.
        SHARED

        # Provides a relative path to your source file(s).
        native-lib.cpp)

# Searches for a specified prebuilt library and stores the path as a
# variable. Because CMake includes system libraries in the search path by
# default, you only need to specify the name of the public NDK library
# you want to add. CMake verifies that the library exists before
# completing its build.

find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)

# Specifies libraries CMake should link to your target library. You
# can link multiple libraries, such as libraries you define in this
# build script, prebuilt third-party libraries, or system libraries.

target_link_libraries( # Specifies the target library.
        buzzer

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib})
```

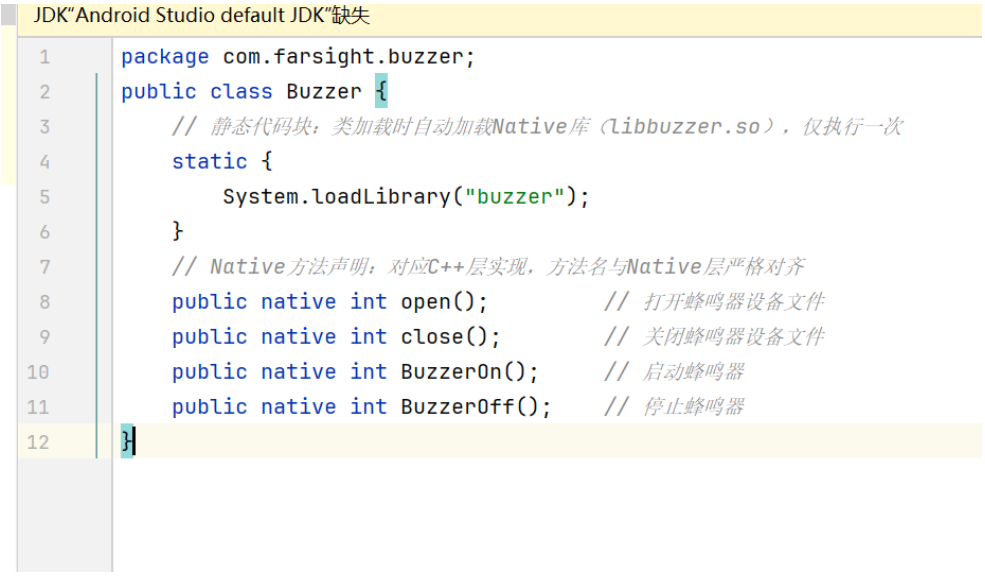
AndroidManifest.xml 跟 led 的一样



Java 层代码分析：

Buzzer.java 封装了所有硬件操作的 Native 方法





MainActivity.java

先初始化 UI 界面，包括 START 和 STOP 俩个按钮

```java
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    Buzzer buzzer = new Buzzer();  // 实例化JNI接口类
    private Button start;  // "START"按钮（控制蜂鸣器启动）
    private Button stop;   // "STOP"按钮（控制蜂鸣器停止）

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);  // 加载UI布局
        // 初始化按钮控件（通过id关联布局文件中的Button）
        start = (Button) findViewById(R.id.button1);
        stop = (Button) findViewById(R.id.button2);
        // 为两个按钮设置点击事件监听器（当前Activity实现OnClickListener接口）
        start.setOnClickListener(this);
        stop.setOnClickListener(this);
    }
```

两个按钮的处理逻辑

```java
    // 统一处理按钮点击事件（通过switch-case区分按钮id）
    @Override
    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.button1:  // 点击"START"按钮
                if (buzzer.open() == -1) {  // 检查设备是否打开成功
                    Toast.makeText(this, "设备打开失败！", Toast.LENGTH_SHORT).show();
                    return;  // 打开失败则终止后续操作
                }
                buzzer.BuzzerOn();  // 发送启动蜂鸣器命令
                buzzer.close();     // 关闭设备，释放资源
                break;
            case R.id.button2:  // 点击"STOP"按钮
                if (buzzer.open() == -1) {  // 检查设备是否打开成功
                    Toast.makeText(this, "设备打开失败！", Toast.LENGTH_SHORT).show();
                    return;  // 打开失败则终止后续操作
                }
                buzzer.BuzzerOff();  // 发送停止蜂鸣器命令
                buzzer.close();      // 关闭设备，释放资源
                break;
        }
    }
}
```

native-lib.cpp 通过 JNI 技术实现 Java 层声明的 Native 方法，直接与蜂鸣器驱动交互

```cpp
#include <jni.h>
#include <string>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <android/log.h>
#include <sys/ioctl.h>
#include <unistd.h>

// 定义蜂鸣器控制的IOCTL命令（与内核驱动约定：魔数为'b'，1=启动，0=停止）
#define BUZZER_ON   _IO('b',1)
#define BUZZER_OFF  _IO('b',0)

int fd = 0;  // 全局设备文件描述符，记录蜂鸣器设备的打开状态

// JNI方法：打开蜂鸣器设备文件（对应Java层Buzzer.open()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_buzzer_Buzzer_open(JNIEnv* env, jobject) {
    // 打开Linux设备节点/dev/buzzer_ctl（字符设备，由蜂鸣器驱动提供）
    fd = open("/dev/buzzer_ctl", O_RDWR);
    if (fd < 0) {  // 打开失败（文件描述符<0）
        __android_log_print(ANDROID_LOG_INFO, "serial", "open /dev/buzzer_ctl Error");
    }
    return fd;  // 返回文件描述符（Java层通过返回值判断是否打开成功）
}

// JNI方法：启动蜂鸣器（对应Java层Buzzer.BuzzerOn()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_buzzer_Buzzer_BuzzerOn(JNIEnv* env, jobject) {
    ioctl(fd, BUZZER_ON);  // 向驱动发送"启动蜂鸣器"IOCTL命令
    return 0;
}

// JNI方法：停止蜂鸣器（对应Java层Buzzer.BuzzerOff()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_buzzer_Buzzer_BuzzerOff(JNIEnv* env, jobject) {
    ioctl(fd, BUZZER_OFF);  // 向驱动发送"停止蜂鸣器"IOCTL命令
    return 0;
}

// JNI方法：关闭蜂鸣器设备文件（对应Java层Buzzer.close()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_buzzer_Buzzer_close(JNIEnv* env, jobject) {
    if (fd > 0) {  // 仅当文件描述符有效时（已打开），执行关闭操作
        close(fd);
    }
    return 0;
}
```

与 Linux 实现的对比

Android 与 Linux 设备控制对比（以 Buzzer 为例）

在 Linux 嵌入式开发中，我们通常编写一个 C 语言应用程序（如 buzzer_test）在终端运行；而在 Android 中，我们需要构建一个完整的 App。

比较维度	Linux C 应用驱动案例	Android App 驱动案例
用户入口	命令行终端 (Shell)	触摸屏图形界面 (Activity)
执行方式	./buzzer_test 1 (参数控制)	点击按钮触发事件回调
代码层级	单层: C 代码直接调用内核 API	三层: Java (UI) -> JNI (桥接) -> C++ (内核调用)
权限管理	依赖 Shell 用户权限 (通常是 root)	依赖 AndroidManifest 权限及系统签名 (System App)
生	进程启动 ->	Activity onCreate -> onClick -> onDestroy

比较维度	Linux C 应用驱动案例	Android App 驱动案例
命周期	执行 -> 退出	

代码实现对比

Linux C 方式:

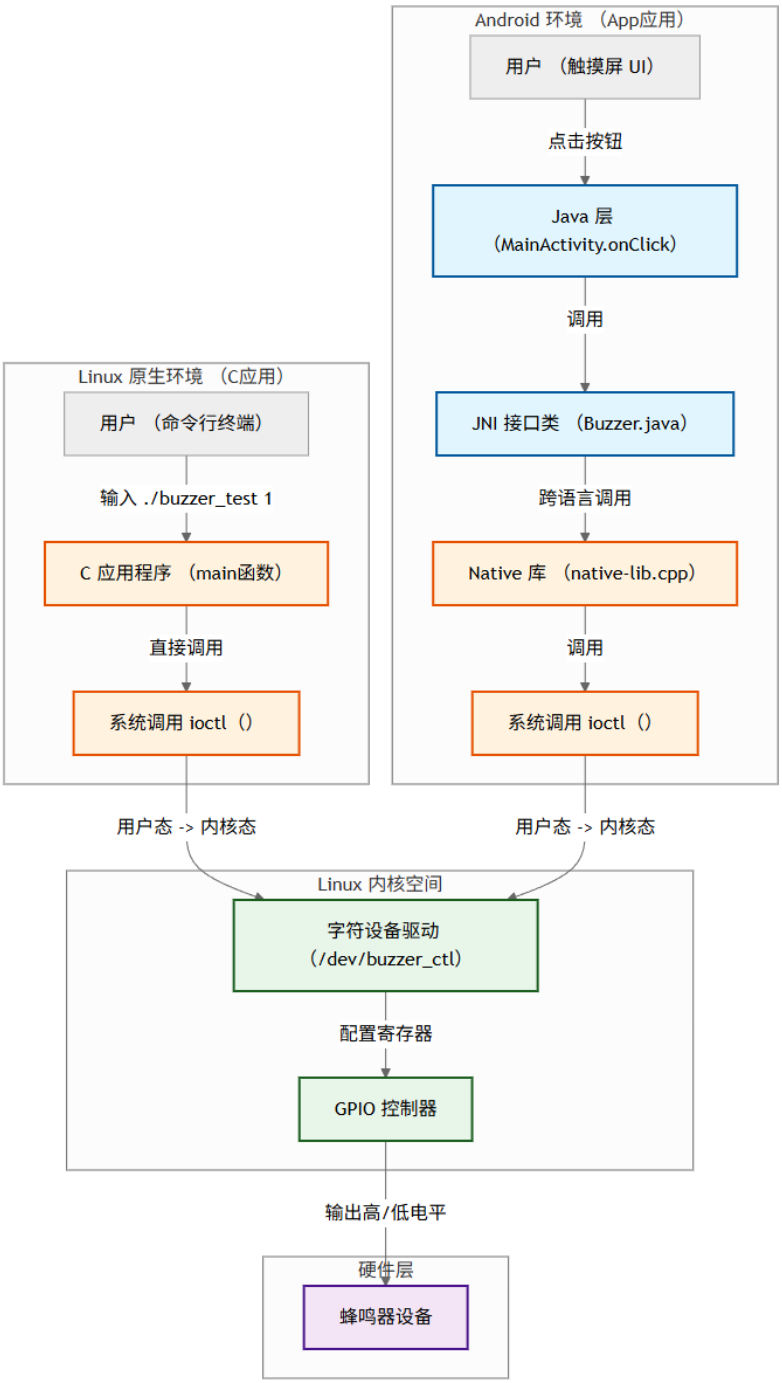
buzzer_test.c

```c
int main(int argc, char **argv) {
    int fd = open("/dev/buzzer_ctl", O_RDWR);
    int cmd = atoi(argv[1]); // 从命令行参数获取命令
    if (cmd == 1)
        ioctl(fd, BUZZER_ON); // 直接控制
    else
        ioctl(fd, BUZZER_OFF);
    close(fd);
    return 0;
}
```

Android 方式:需要跨越虚拟机边界，步骤繁琐但用户体验好。

```
// Java
buzzer.BuzzerOn();
    ↓ (JNI)
// C++
ioctl(fd, BUZZER_ON);
```

蜂鸣器在 Android 环境与 Linux 原生环境下的驱动控制对比流程图:



入口差异：

Linux：用户通过 Shell 命令行（如 ./buzzer_test 1）与程序交互，入口是 main 函数。

Android：用户通过触摸屏点击按钮，入口是 Android 框架的回调方法 onClick。

调用层级：

Linux: 路径非常短，C 代码直接发起系统调用，效率高，结构简单。

Android: 路径较长，需要经过 Java 虚拟机 (Dalvik/ART) -> JNI 桥接 -> Native C++ 代码，最后才发起系统调用。这是为了保证 Android 应用层的平台无关性和安全性。

殊途同归：

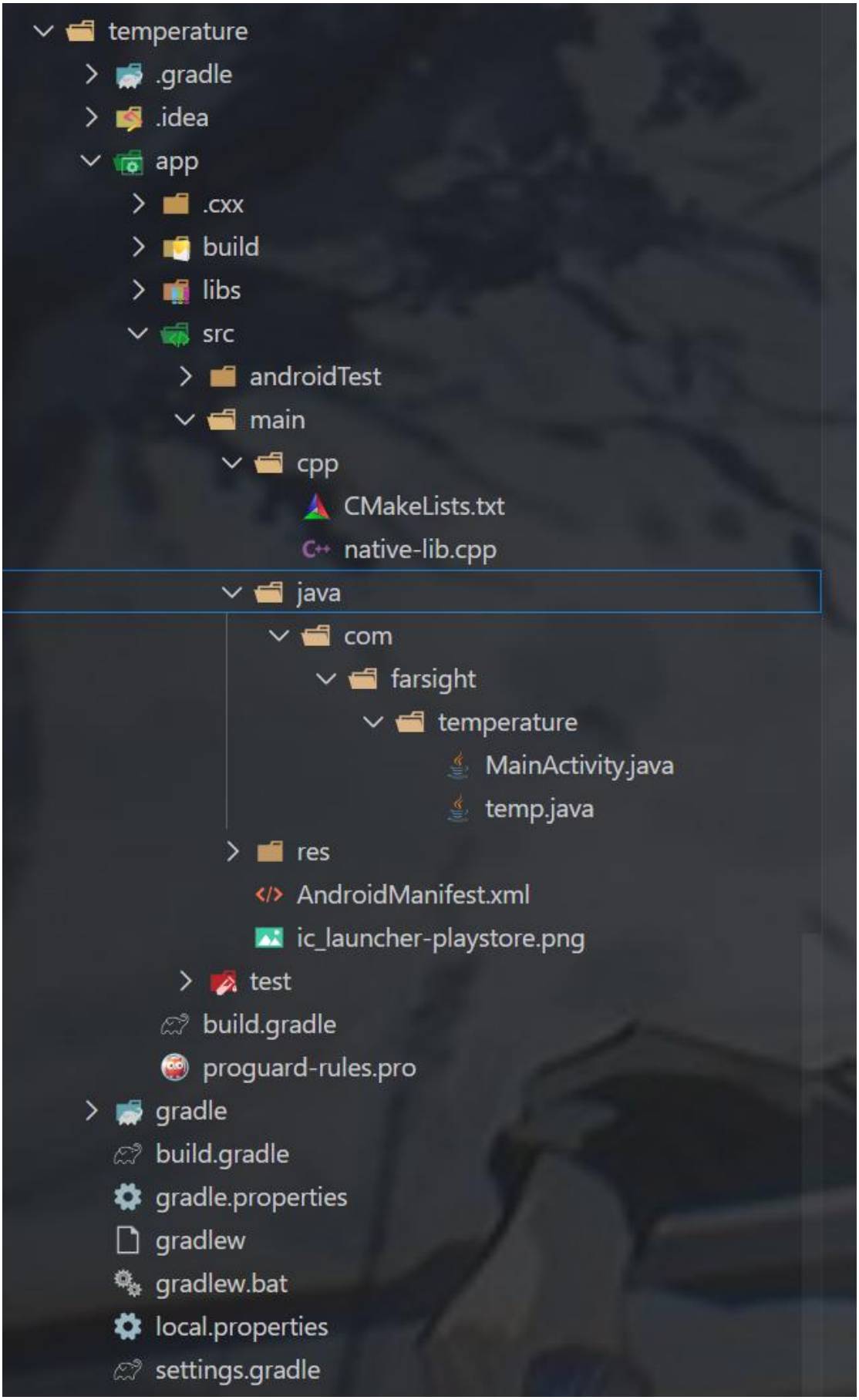
无论上层如何封装，最终都汇聚到 Linux 内核空间。

两者都使用相同的系统调用接口（open, ioctl, close）来操作同一个设备节点（/dev/buzzer_ctl）。

底层的驱动程序和硬件响应是完全一致的。

c) 温度采集

实验工程文件结构如下


temperature
.gradle
.idea
app
.cxx
build
libs
src
androidTest
main
cpp
CMakeLists.txt
native-lib.cpp
java
com
farsight
temperature
MainActivity.java
temp.java
res
AndroidManifest.xml
ic_launcher-playstore.png
test
build.gradle
proguard-rules.pro
gradle
build.gradle
gradle.properties
gradlew
gradlew.bat
local.properties
settings.gradle

MainActivity.java: 主界面，显示温度值，包含定时刷新逻辑。

temp.java: JNI 接口类，加载 libtemperature.so。

native-lib.cpp: C++ 实现，操作设备节点 /dev/temp。

MainActivity.java 如何驱动外设:

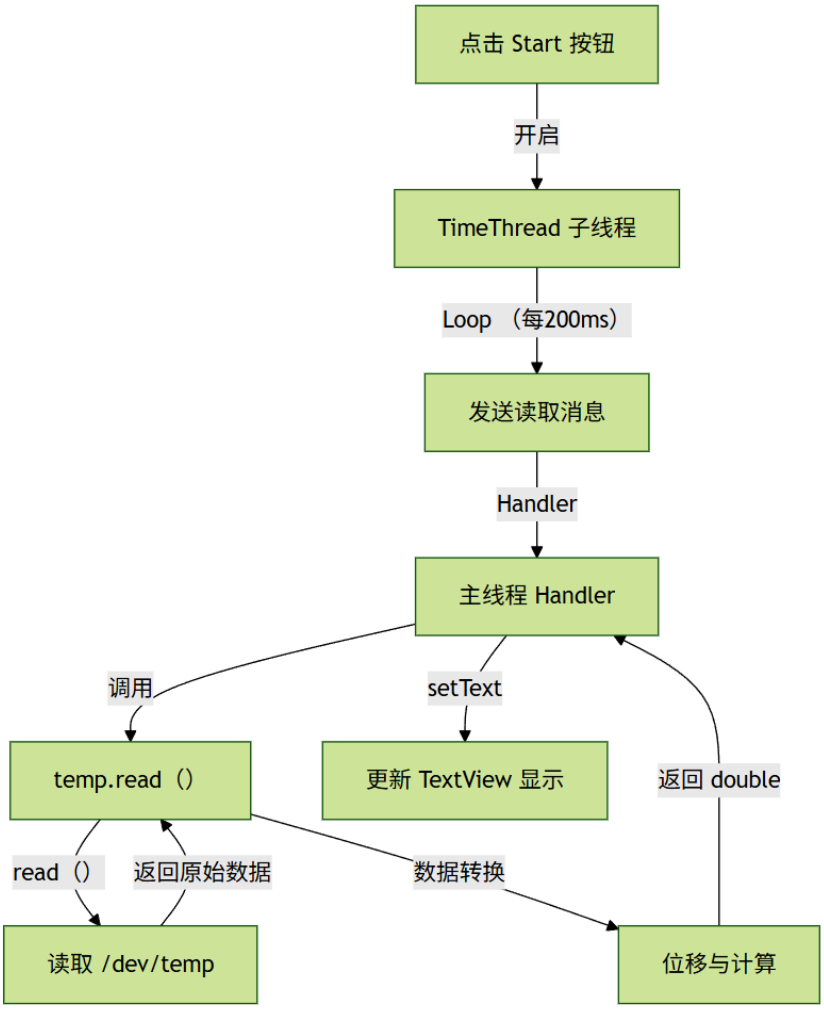
多线程轮询: 不同于 LED 的按需触发，温度采集需要持续读取。Activity 中定义了一个内部类 TimeThread 继承自 Thread。

启动采集: 点击 Start 按钮，设置标志位 sensorflag = true，调用 tmp.open()，并启动线程。

数据获取：线程每隔 200ms 发送消息给 Handler。Handler 接收消息后调用 tmp.read() 获取温度数据（double 类型）。

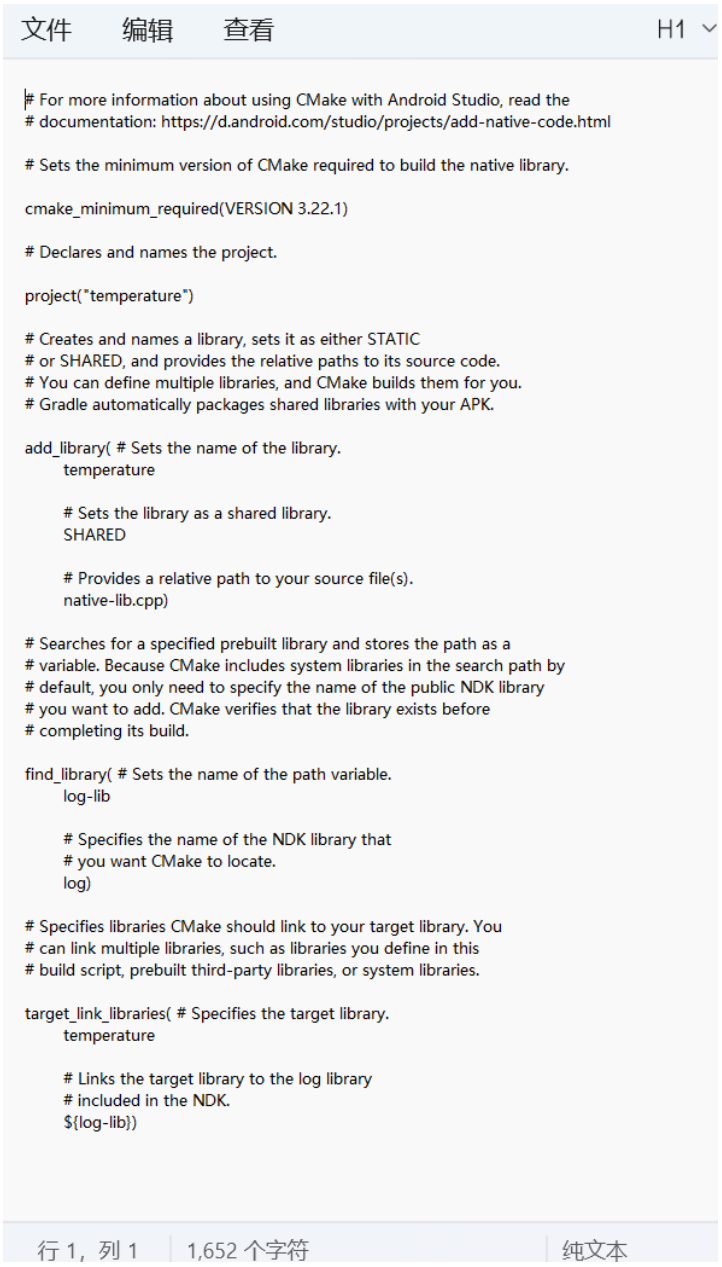
UI 更新: 将获取的温度值更新到 TextView 上。

程序调用的流程图



代码剖析如下：

CMakeLists.txt 同之前两个，只改了项目名跟库名



AndroidManifest.xml

```
<manifest ...>
    <application
        android:theme="@style/Theme.Temperature"  # 应用主题（与项目名匹配）
        android:icon="@mipmap/logoi"              # 应用图标
        android:label="@string/app_name"          # 应用名称（显示为"temperature"）
        android:supportsRtl="true"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"          # 主活动类名
            android:exported="true">              # 允许外部启动（作为应用入口）
            <intent-filter>
                <!-- 声明为应用启动入口 -->
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Java 层代码分析：

temp.java 也是接口，封装了读取温度等方法

JDK"Android Studio default JDK"缺失

```
1  package com.farsight.temperature;
2  public class temp {
3      // 静态代码块：类加载时自动加载Native库（libtemperature.so），仅执行一次
4      static {
5          System.loadLibrary("temperature");
6      }
7      // Native方法声明：与C++层实现严格对齐
8      public native int open();          // 打开温度传感器设备文件
9      public native double read();       // 读取温度数据（返回double类型，适配温度精度）
10     public native int close();         // 关闭温度传感器设备文件
11 }
```

MainActivity.java 整合 UI 控制、多线程数据采集、UI 更新，确保传感器信息实时显示

先初始化各种对象

```java
public class MainActivity extends AppCompatActivity {  0 个用法
    TextView val;          // 温度值显示文本框  0 个用法
    Button start_btn;      // "打开传感器"按钮  0 个用法
    Button close_btn;      // "关闭传感器"按钮  0 个用法
    Boolean sensorflag = false;  // 传感器状态标记（false=未打开，true=已打开）  0 个用法
    double data = 0;       // 存储读取的温度值  0 个用法
    temp tmp = new temp(); // 实例化JNI接口类  0 个用法

    @Override  0 个用法
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);  // 加载UI布局
        // 初始化控件（通过id关联布局文件）
        val = findViewById(R.id.text);
        start_btn = findViewById(R.id.start_btn);
        close_btn = findViewById(R.id.close_btn);
```

点击 UI 的“打开传感器”，期中的交互方法如下

```java
// "打开传感器"按钮点击事件
start_btn.setOnClickListener(new View.OnClickListener() {
    @Override  0 个用法
    public void onClick(View view) {
        if (!sensorflag) {  // 仅当传感器未打开时执行
            sensorflag = true;  // 更新状态标记
            tmp.open();         // 打开传感器设备
            new TimeThread().start();  // 启动实时读取线程
        }
    }
});
```

关闭传感器方法，也是接收到信号后修改更新状态标记来达到目的

```
        // "关闭传感器"按钮点击事件
        close_btn.setOnClickListener(new View.OnClickListener() {
            @Override  0 个用法
            public void onClick(View view) {
                if (sensorflag) {  // 仅当传感器已打开时执行
                    tmp.close();       // 关闭传感器设备
                    sensorflag = false; // 更新状态标记
                }
            }
        });
    }
```

多线程实现数据实时更新

```java
// 实时数据采集线程：每隔200ms发送一次数据读取请求，避免阻塞UI线程
public class TimeThread extends Thread {  0 个用法
    @Override  0 个用法
    public void run() {
        super.run();
        do {  // 循环执行（通过sensorflag控制启停）
            if (sensorflag) {  // 传感器已打开：发送读取数据消息
                try {
                    Thread.sleep(200);  // 间隔200ms，平衡实时性与性能
                    Message msg = new Message();
                    msg.what = 1;  // 消息标识：读取温度
                    handler.sendMessage(msg);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } else {  // 传感器未打开：发送更新UI提示消息
                try {
                    Thread.sleep(200);
                    Message msg = new Message();
                    msg.what = 2;  // 消息标识：提示未打开
                    handler.sendMessage(msg);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } while (true);  // 无限循环，通过sensorflag控制逻辑分支
    }
}
```

```java
// Handler：接收线程消息，更新UI（Android UI更新需在主线程执行）
private Handler handler = new Handler(new Handler.Callback() {  0 个用法
    @Override  0 个用法
    public boolean handleMessage(Message msg) {
        switch (msg.what) {
            case 1:  // 读取温度成功：更新文本框显示温度值（保留3位小数）
                data = tmp.read();
                val.setText(String.format("%.3f", data) + "°C");
                break;
            case 2:  // 传感器未打开：显示提示文本
                val.setText("传感器未打开");
                break;
        }
        return false;
});
```

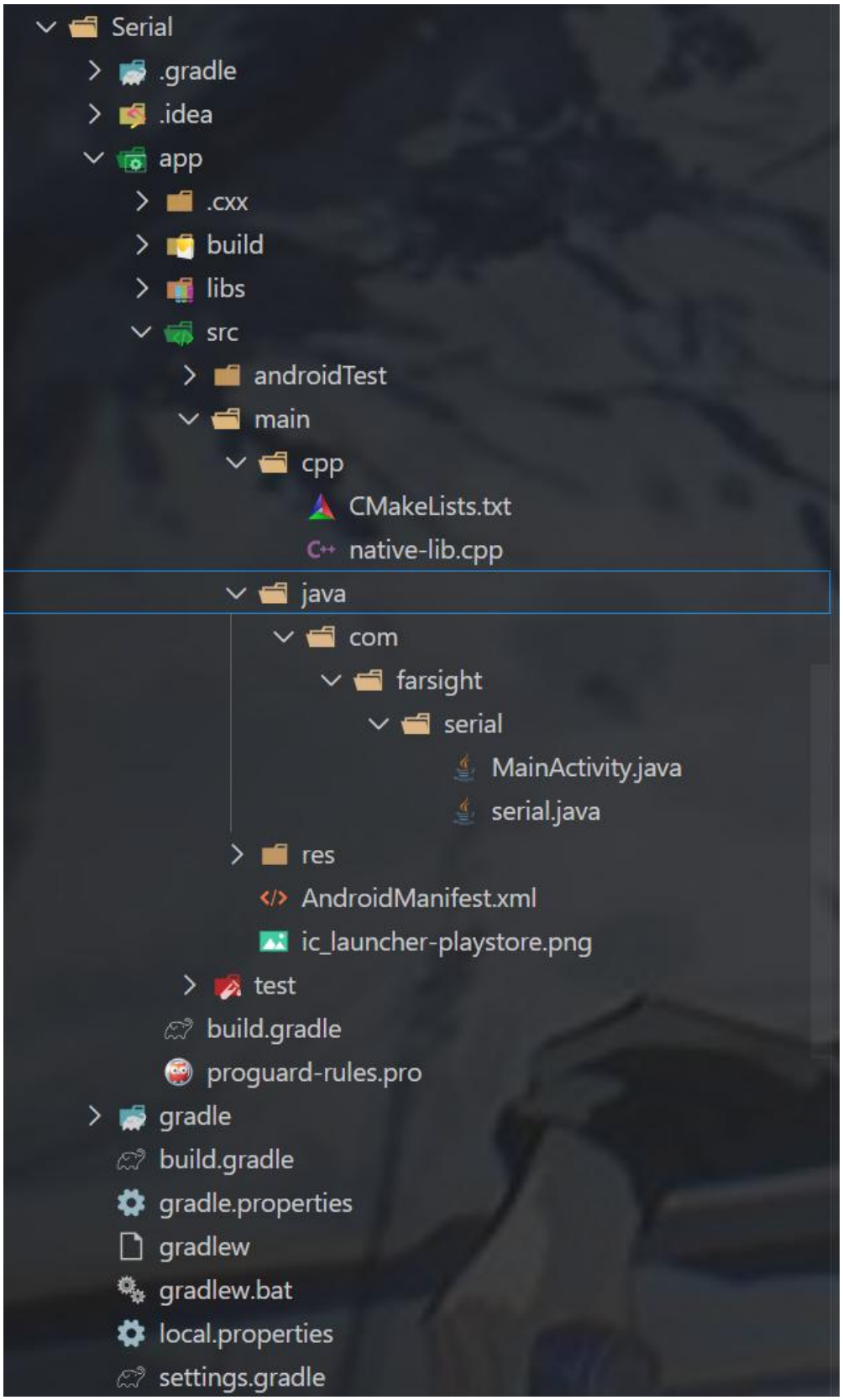
Native 层代码分析：

native-lib.cpp 与 LED、蜂鸣器的 “控制类” 操作不同，温度传感器属于 “数据采集类” 硬件，核心通过 read 系统调用获取原始数据并转换为温度值

```cpp
#include <jni.h>
#include <string>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <android/log.h>
#include <unistd.h>

int fd = 0;    // 全局设备文件描述符，关联温度传感器设备
int data = 0;  // 存储传感器原始数据（整型）

// JNI方法：打开温度传感器设备文件（对应Java层temp.open()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_temperature_temp_open(JNIEnv* env, jobject) {
    // 打开Linux设备节点/dev/temp（字符设备，由温度传感器驱动提供）
    fd = open("/dev/temp", O_RDWR);
    if (fd < 0) {  // 打开失败（文件描述符<0）
        __android_log_print(ANDROID_LOG_INFO, "temp", "open /dev/temp Error");
    }
    else {  // 打开成功
        __android_log_print(ANDROID_LOG_INFO, "temp", "open /dev/temp success");
    }
    return 0;
}

// JNI方法：读取温度数据（对应Java层temp.read()）
extern "C" JNIEXPORT jdouble JNICALL
Java_com_farsight_temperature_temp_read(JNIEnv* env, jobject) {
    double ret_val;  // 存储转换后的温度值（浮点型）
    // 读取设备数据：从fd关联的设备读取sizeof(data)字节到data变量
    read(fd, (char*)&data, sizeof(data));
    // 原始数据转换为温度值：(data >> 5) → 右移5位（丢弃低5位噪声/无效数据），*0.125 → 按传感器精度缩放
    ret_val = ((data >> 5) * 0.125);
    // 打印转换后的温度值到Logcat（标签为"temp"），便于调试
    __android_log_print(ANDROID_LOG_INFO, "temp", "%.3f", ret_val);
    return ret_val;  // 返回温度值给Java层
}

// JNI方法：关闭温度传感器设备文件（对应Java层temp.close()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_temperature_temp_close(JNIEnv* env, jobject) {
    if (fd > 0) {  // 仅当文件描述符有效时（已打开）执行关闭操作
        close(fd);
        __android_log_print(ANDROID_LOG_INFO, "temp", "close /dev/temp success");
    }
    return 0;
}
```

d)　串口通信

实验工程结构如下



MainActivity.java: 复杂的 UI，包含波特率/串口号选择 (Spinner)、发送/接收显示区。

serial.java: JNI 接口类，加载 libserial.so。

native-lib.cpp: C++ 实现，操作 /dev/ttyS0 或 /dev/ttyS4，并配置 termios 结构体。

MainActivity.java 如何驱动外设:

配置参数: 通过 Spinner 选择串口号（如 /dev/ttyS0）和波特率。

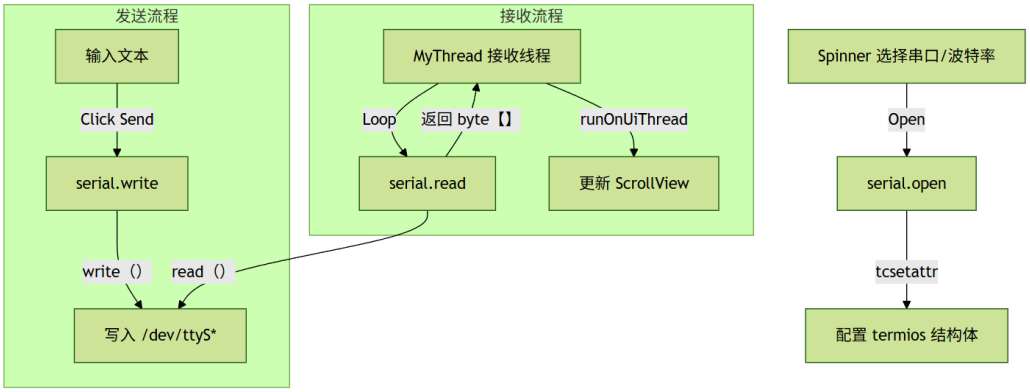
打开串口: 点击 Open 按钮，调用 com.open(port, baudrate)。

数据收发:

发送: 获取 EditText 内容，转换为字节数组，调用 com.write()。

接收: 同样需要开启子线程不断轮询 com.read()，将读取到的字节数组转换为字符串并追加显示在 ScrollView 中。

驱动流程逻辑图如下:



代码剖析

CMakeLists.txt 同上，只改了项目名和库名

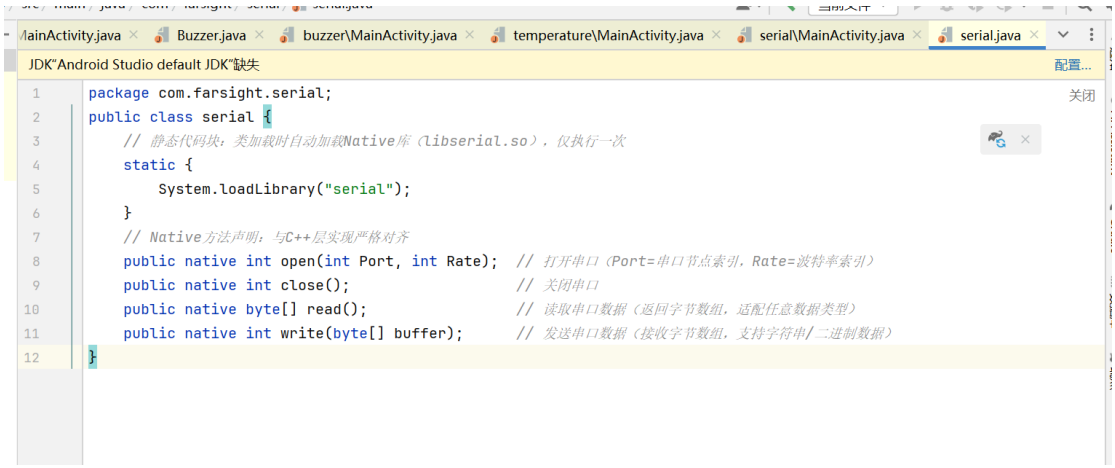
```
# 最小CMake版本要求（适配Android Studio编译环境）
cmake_minimum_required(VERSION 3.22.1)
# 项目名称（与串口功能匹配，便于识别）
project("serial")
# 编译共享库（SHARED），库名为"serial"，关联源文件native-lib.cpp
add_library(serial SHARED native-lib.cpp)
# 查找Android系统日志库（log-lib），支持Native层打印调试日志
find_library(log-lib log)
# 链接目标库（serial）与日志库（log-lib），确保调试日志功能可用
target_link_libraries(serial ${log-lib})
```

AndroidManifest.xml

```
        android:theme="@style/Theme.Serial"  # 应用主题（与项目名匹配）
        android:icon="@mipmap/ic_launcher"   # 应用图标
        android:label="@string/app_name"      # 应用名称（显示为"Serial"）
        android:supportsRtl="true"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"      # 主活动类名
            android:exported="true">          # 允许外部启动（作为应用入口）
            <intent-filter>
                <!-- 声明为应用启动入口 -->
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Java 层代码分析：

serial.java 封装了一些函数接口



MainActivity.java

先定义各种变量和控件并初始化

```
public class MainActivity extends AppCompatActivity implements OnClickListener {
    private EditText ET1;           // 发送数据输入框
    private Button SEND, OPENSERIAL, CLOSESERIAL, CLEAR;  // 功能按钮
    private TextView msglist;       // 收发数据显示文本框
    private ScrollView sv;          // 滚动视图（避免数据溢出）
    private Spinner spinner, spinner2;  // 串口节点选择、波特率选择控件
    private List<String> data_list, data_list2;  // Spinner数据源
    private ArrayAdapter<String> arr_adapter, arr_adapter2;  // Spinner适配器
    private int serial;             // 串口节点索引（0=/dev/ttyS0，1=/dev/ttyS4）
    private int Baudrate;           // 波特率索引（0=2400~6=115200）
    serial com = new serial();      // 实例化JNI接口类
    MyThread myThread = null;       // 串口数据接收线程

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);  // 加载UI布局
        // 初始化控件（通过id关联布局文件）
        initView();
        // 初始化Spinner（串口节点+波特率选择）
        initSpinner();
        // 绑定按钮点击事件
        bindClickEvent();
    }
```

初始化串口波特率等参数以便使用串口调试助手连接

```
private void initSpinner() {
    // 串口节点数据源（对应Native层的/dev/ttyS0和/dev/ttyS4）
    data_list = new ArrayList<>();
    data_list.add("/dev/ttyS0");
    data_list.add("/dev/ttyS4");
    // 波特率数据源（对应Native层的7种波特率选项）
    data_list2 = new ArrayList<>();
    data_list2.add("2400");
    data_list2.add("4800");
    data_list2.add("9600");
    data_list2.add("19200");
    data_list2.add("38400");
    data_list2.add("57600");
    data_list2.add("115200");

    // 串口节点Spinner适配
    arr_adapter = new ArrayAdapter<>(this, android.R.layout.simple_spinner_item, data_list);
    arr_adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    spinner.setAdapter(arr_adapter);
    spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
        @Override
        public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
            serial = position;  // 记录串口节点索引（0/1）
        }
        @Override
        public void onNothingSelected(AdapterView<?> parent) {}
    });

    // 波特率Spinner适配
    arr_adapter2 = new ArrayAdapter<>(this, android.R.layout.simple_spinner_item, data_list2);
    arr_adapter2.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    spinner2.setAdapter(arr_adapter2);
    spinner2.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
        @Override
        public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
            Baudrate = position;  // 记录波特率索引（0~6）
        }
        @Override
        public void onNothingSelected(AdapterView<?> parent) {}
    });

    // 默认选择：串口/dev/ttyS4（索引1），波特率115200（索引6）
    spinner.setSelection(1, true);
    spinner2.setSelection(6, true);
}
```

按钮点击事件的函数逻辑

```java
@Override
public void onClick(View v) {
    switch (v.getId()) {
        case R.id.open_serial:  // 打开串口
            int ret = com.open(serial, Baudrate);  // 传入参数打开串口
            if (ret > 0) {
                Toast.makeText(this, "串口打开成功", Toast.LENGTH_LONG).show();
                // 更新按钮状态（防止重复打开）
                OPENSERIAL.setEnabled(false);
                CLOSESERIAL.setEnabled(true);
                SEND.setEnabled(true);
                myThread = new MyThread();  // 启动数据接收线程
                myThread.start();
            }
            break;
        case R.id.close_serial:  // 关闭串口
            if (myThread != null) {
                com.close();  // 关闭串口设备
                myThread = null;  // 终止接收线程
            }
            // 重置按钮状态
            SEND.setEnabled(false);
            OPENSERIAL.setEnabled(true);
            CLOSESERIAL.setEnabled(false);
            break;
        case R.id.send1:  // 发送数据
            String sendData = ET1.getText().toString() + "\n";  // 获取输入数据，加换行符
            com.write(sendData.getBytes());  // 转换为字节数组发送
            // 更新显示（标记"发送"）
            msglist.append("[发送]" + sendData);
            ET1.setText("");  // 清空输入框
            // 滚动到最新数据
            sv.post(() -> sv.fullScroll(ScrollView.FOCUS_DOWN));
            break;
        case R.id.clear:  // 清空显示
            msglist.setText("");
            break;
    }
}
```

最后是串口接受线程，将接受的信息从串口传到实验箱并显示出来

```
    // 串口数据接收线程：异步读取串口数据，避免阻塞UI线程
    class MyThread extends Thread {
        @RequiresApi(api = Build.VERSION_CODES.KITKAT)
        @Override
        public void run() {
            while (true) {
                byte[] recvData = com.read();  // 读取串口数据
                if (recvData == null) break;  // 数据为空则退出循环（串口关闭）
                // 字节数组转换为字符串（默认UTF-8编码）
                String recvStr = new String(recvData);
                String finalRecvStr = recvStr;
                // 主线程更新UI（Android UI更新需在主线程执行）
                runOnUiThread(() -> {
                    msglist.append("[接收]" + finalRecvStr + "\n");  // 标记"接收"
                    // 滚动到最新数据
                    sv.post(() -> sv.fullScroll(ScrollView.FOCUS_DOWN));
                });
            }
            // 线程结束后重置按钮状态（主线程）
            runOnUiThread(() -> {
                OPENSERIAL.setEnabled(true);
                CLOSESERIAL.setEnabled(false);
            });
        }
    }
}
```

Native 层代码分析：

native-lib.cpp Native 层通过 Linux 串口标准接口 (termios 结构体) 配置通信参数，

实现数据双向传输

```c
#include <jni.h>
#include <string.h>
#include <termios.h>  // 串口配置核心头文件
#include <android/log.h>
#include <unistd.h>
#include <fcntl.h>

#undef TCSAFLUSH
#define TCSAFLUSH TCSETSF
#ifndef _TERMIOS_H_
#define _TERMIOS_H_
#endif

int fd = 0;  // 全局串口设备文件描述符

// JNI方法：打开串口（对应Java层serial.open()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_serial_serial_open(JNIEnv* env, jobject thiz, jint port, jint rate) {
    if (fd <= 0) {
        // 根据port索引选择串口设备节点
        if (port == 0) {
            fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NONBLOCK);  // 非阻塞模式
            __android_log_print(ANDROID_LOG_INFO, "serial", "open /dev/ttyS0");
        }
        else if (port == 1) {
            fd = open("/dev/ttyS4", O_RDWR | O_NOCTTY | O_NONBLOCK);
            __android_log_print(ANDROID_LOG_INFO, "serial", "open /dev/ttyS4");
        }
        else {
            __android_log_print(ANDROID_LOG_INFO, "serial", "串口节点参数错误");
            return -1;
        }

        if (fd > 0) {
            __android_log_print(ANDROID_LOG_INFO, "serial", "串口打开成功，fd=%d", fd);
            struct termios ios;  // 串口配置结构体
            tcgetattr(fd, &ios);  // 获取当前串口配置

            // 1. 禁用特殊字符处理（避免回车/换行转换）
            ios.c_oflag &= ~(INLCR | IGNCR | ICRNL | ONLCR | OCRNL);
            ios.c_iflag &= ~(ICRNL | IXON | INLCR | IGNCR | ICRNL | ONLCR | OCRNL);
            // 2. 设置波特率（根据rate索引匹配）
            switch (rate) {
            case 0: cfsetospeed(&ios, B2400); cfsetispeed(&ios, B2400); break;
            case 1: cfsetospeed(&ios, B4800); cfsetispeed(&ios, B4800); break;
            case 2: cfsetospeed(&ios, B9600); cfsetispeed(&ios, B9600); break;
            case 3: cfsetospeed(&ios, B19200); cfsetispeed(&ios, B19200); break;
            case 4: cfsetospeed(&ios, B38400); cfsetispeed(&ios, B38400); break;
            case 5: cfsetospeed(&ios, B57600); cfsetispeed(&ios, B57600); break;
            case 6: cfsetospeed(&ios, B115200); cfsetispeed(&ios, B115200); break;
            default: cfsetospeed(&ios, B9600); cfsetispeed(&ios, B9600); break;
            }
            // 3. 配置数据位、校验位、停止位
            ios.c_cflag |= (CLOCAL | CREAD);  // 启用本地连接、允许读取数据
            ios.c_cflag &= ~PARENB;  // 禁用校验位（NONE）
            ios.c_cflag &= ~CSTOPB;  // 禁用2位停止位（使用1位）
            ios.c_cflag &= ~CSIZE;   // 清除数据位配置
            ios.c_cflag |= CS8;      // 设置8位数据位
            ios.c_lflag = 0;         // 禁用本地模式（原始数据模式）
            // 4. 刷新串口缓冲区，应用新配置
            tcflush(fd, TCIFLUSH);
            tcsetattr(fd, TCSANOW, &ios);  // 立即生效
        }
    }
    return fd;  // 返回文件描述符（>0表示成功）
}
```

```cpp
// JNI方法：关闭串口（对应Java层serial.close()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_serial_serial_close(JNIEnv* env, jobject thiz) {
    if (fd > 0) close(fd);  // 关闭设备，释放文件描述符
    fd = -1;  // 重置文件描述符
    return 1;
}

// JNI方法：读取串口数据（对应Java层serial.read()）
extern "C" JNIEXPORT jbyteArray JNICALL
Java_com_farsight_serial_serial_read(JNIEnv* env, jobject thiz) {
    unsigned char buffer[512];  // 数据缓冲区（512字节，适配大部分场景）
    memset(buffer, 0, sizeof(buffer));  // 清空缓冲区
    int len = 0;

    if (fd > 0) {
        len = read(fd, buffer, sizeof(buffer));  // 读取串口数据（非阻塞模式）
        if (len > 0) {
            // 创建jbyteArray，将缓冲区数据复制到Java层
            jbyteArray result = env->NewByteArray(len);
            env->SetByteArrayRegion(result, 0, len, reinterpret_cast<const jbyte*>(buffer));
            // 释放局部引用（避免内存泄露）
            env->ReleaseByteArrayElements(result, env->GetByteArrayElements(result, JNI_FALSE), 0);
            return result;
        }
        usleep(500);  // 无数据时休眠500微秒，降低CPU占用
    }
    return nullptr;  // 无数据或串口关闭，返回空
}

// JNI方法：发送串口数据（对应Java层serial.write()）
extern "C" JNIEXPORT jint JNICALL
Java_com_farsight_serial_serial_write(JNIEnv* env, jobject thiz, jbyteArray buf) {
    if (fd <= 0 || buf == nullptr) return -1;
    // 获取Java层byte数组长度与数据
    jsize len = env->GetArrayLength(buf);
    jbyte* buffer = env->GetByteArrayElements(buf, JNI_FALSE);
    // 写入串口数据，返回实际写入字节数
    int writeLen = write(fd, buffer, len);
    // 释放Java层数组引用
    env->ReleaseByteArrayElements(buf, buffer, 0);
    return writeLen;
}
```

针对不同 GPIO，进行各个案例的工程文件结构分析以及主活动文件 MainActivity.java 如何完成对于外设的驱动。

3. 代码剖析要求：

a) 第一次 Android 实验：

i. 作图描述每个工程中各个文件之间的调用关系；

ii. 核心代码中对于不同模块（activity,intent,content provider, service）的实际应用剖析（代码注解+流程图）

b) 第二次 Android 实验：

i. 工程文件中如何完成对于 GPIO 通用外部设备的驱动？（可采用流程图，代码加注等形式）

ii. 比较 Android 对于设备的控制与 linux 设备驱动案例的不同（从 LED 灯、蜂鸣器、温度采集中任选一个比对分析）。（可采用流程图，代码加注等形式）

选择蜂鸣器

4. 作业提交：

a) 命名：姓名+学号+4.pdf

b) 提交至数字化教学平台 course.xmu.edu.cn

c) 截止时间：12 月 21 晚 12:00