**一14一 一14一 一12一 一11一** 一10.5一 一10.5一 '一10一"

**目 录**

[1. 位运算(移位、位与、或、异或、非） 4](#_Toc17768)

[1.1. 给出2n+1个数，其中有2n个数出现过两次，找出只出现了一次的那个数。 5](#_Toc6663)

[2. String和Hex的互相转换 6](#_Toc29682)

[3. 运算符优先级 8](#_Toc12020)

[4. 求最大公约数（欧几里得、辗转相除法） 8](#_Toc24585)

[5. 排序 8](#_Toc22442)

[6. 基本数据类型转换 10](#_Toc31894)

[6.1. 数据类型概述 11](#_Toc9493)

[6.2. 隐式类型转换（范围小类型的值 赋给 范围大的类型） 11](#_Toc30143)

[6.2.1. 隐式类型转换 11](#_Toc13695)

[6.2.2. 特殊情况：char型不能进行相关的自动类型转换 12](#_Toc28792)

[6.2.3. 特殊情况：整数类型的 大→小 有可能自动类型转换 12](#_Toc25236)

[6.3. 显示类型转换（范围大类型的值 赋给 范围小类型的值） 12](#_Toc11806)

[6.3.1. 显示类型转换 12](#_Toc1301)

[6.3.2. 丢失精度 12](#_Toc13791)

[6.4. 进行数学运算时的数据类型自动提升 13](#_Toc6950)

[6.5. 类型转换中的符号扩展(Sign Extension) 13](#_Toc31314)

[6.5.1. 计算机中数的表示 13](#_Toc3577)

[6.5.2. java中的符号扩展 13](#_Toc25843)

[6.5.2.1. 什么是符号扩展 13](#_Toc31205)

[6.5.2.2. java中数值类型转换的规则 13](#_Toc15967)

[6.5.3. 解析“多重转型”问题 13](#_Toc13358)

[6.5.4. 几个转型的例子 14](#_Toc24959)

[6.5.5. 总结 14](#_Toc25601)

[7. 四舍五入、取整、随机数 14](#_Toc22792)

[8. DecimalFormat处理数字的显示 15](#_Toc28797)

[9. 保留两位小数 15](#_Toc16886)

[10. Random奥秘 16](#_Toc21427)

[10.1. java.lang.Math.Random; 16](#_Toc25328)

[10.2. java.util.Random; 16](#_Toc8972)

[10.2.1. Random()的两种构造方法 16](#_Toc10164)

[10.3. 两个随机函数的特点： 16](#_Toc24435)

[10.4. 用法例子 16](#_Toc26879)

[10.5. Split方法的几种情况 17](#_Toc29746)

[11. substring 17](#_Toc27783)

[12. String是不可变对象 17](#_Toc16747)

[12.1. 什么是不可变对象？ 17](#_Toc9735)

[12.2. 区分对象和对象的引用 17](#_Toc14218)

[12.3. 为什么String对象是不可变的？ 18](#_Toc18798)

[12.4. String对象真的不可变吗？ 19](#_Toc29483)

[13. Split 20](#_Toc32679)

[14. Format格式化 21](#_Toc1350)

[14.1. 简单的格式化 21](#_Toc23200)

[14.2. 带格式的格式化 22](#_Toc12702)

[14.3. 在string标签中使用百分号 22](#_Toc19056)

[14.4. String.format介绍（太多懒得看） 23](#_Toc32198)

[15. Return、Break、Continue的作用 24](#_Toc9424)

[16. Base64 25](#_Toc32061)

[16.1. R.drawable转bitmap 25](#_Toc6471)

[16.2. String和byte【】之间的互转 25](#_Toc27498)

[16.3. Bitmap转 Base64 25](#_Toc32688)

[16.4. Base64转位图Bitmap 25](#_Toc27266)

[17. MD5算法加密 26](#_Toc7178)

[18. Map排序 26](#_Toc13988)

[19. 随时退出程序ActivityCollector 27](#_Toc2484)

[19.1. 代码 27](#_Toc22176)

[19.2. 发送广播在广播接收器里达到关闭所有Activity的目的 28](#_Toc20855)

[20. 软件更新 28](#_Toc24657)

[21. 修改图标更新后logo未更新 32](#_Toc2328)

[22. FileProvider 的使用 32](#_Toc21617)

[22.1. 在manifest中添加provider 32](#_Toc4169)

[22.2. 资源文件下创建相应的xml文件（如上：则创建filepaths.xml）。 32](#_Toc30281)

[22.3. 注意 32](#_Toc24100)

[22.4. 用处 33](#_Toc26032)

[22.4.1. 修改前 33](#_Toc11171)

[22.4.2. 修改后 33](#_Toc13876)

[23. 获取通讯录，发送短信，接收短信 33](#_Toc27311)

[23.1. 获取通讯录 33](#_Toc21608)

[23.2. 发送短信 34](#_Toc14002)

[23.3. 接收短信 35](#_Toc21152)

[24. 测试工程和单元测试 36](#_Toc4834)

[25. 注释规范 38](#_Toc16556)

[25.1. 文件注释 38](#_Toc11151)

[25.2. 类注释 38](#_Toc3665)

[25.3. 属性注释 38](#_Toc15166)

[25.4. 方法注释 38](#_Toc13961)

[26. 网络信息、网络状态获取 39](#_Toc31074)

[27. 反射机制 40](#_Toc8664)

[27.1. 什么是JAVA的反射机制 40](#_Toc26592)

[27.2. JDK中提供的Reflection API 40](#_Toc24633)

[27.3. JAVA反射机制提供了什么功能 40](#_Toc15821)

[27.3.1. 首先定义用于测试的类: 40](#_Toc8040)

[27.3.2. 获取类的Class对象 41](#_Toc24836)

[27.3.3. 一些方法 42](#_Toc10393)

[27.3.4. 获取类的Constructor 42](#_Toc11656)

[27.3.5. 获取类的Method 43](#_Toc18660)

[27.3.6. 获取类的Fields成员变量（成员属性） 43](#_Toc201)

[27.3.7. 新建类的实例 44](#_Toc25054)

[27.3.8. 调用类的函数 44](#_Toc28744)

[27.3.9. 设置/获取类的属性值 45](#_Toc3132)

[27.4. 动态创建代理类 45](#_Toc3636)

[27.5. 调用类中的private方法举例 46](#_Toc17539)

[27.5.1. 工具类 46](#_Toc29670)

[27.5.2. 对LoadMethod.java的优化处理 48](#_Toc6650)

[27.5.3. ExtendType里的private方法 49](#_Toc18806)

[27.5.4. 调用 49](#_Toc23173)

[27.6. Dialog消失不消失反射示例 49](#_Toc31421)

[27.7. 获取图片最大宽度示例 51](#_Toc18968)

[28. 注解Annotation 52](#_Toc3442)

[28.1. 示例 52](#_Toc10963)

[28.2. 概念及作用 52](#_Toc26964)

[28.3. 分类 52](#_Toc17173)

[28.4. 自定义 53](#_Toc10874)

[28.5. 解析 53](#_Toc14592)

[28.6. 几个 Android 开源库 Annotation 原理简析 54](#_Toc5949)

[28.7. 应用举例：任意bean解析注解的使用 56](#_Toc5960)

[29. Android Annotation 57](#_Toc24153)

[29.1. 概述 57](#_Toc11580)

[29.2. 安卓注解的8种类型 58](#_Toc10044)

[30. android 中View, Window, Activity, WindowManager，ViewRoot几者之间的关系 60](#_Toc19571)

[31. 开启悬浮窗 61](#_Toc3479)

[32. 双屏异显 62](#_Toc14466)

[33. 创建桌面快捷方式 63](#_Toc10924)

[33.1. 首先权限是必不可少的 63](#_Toc411)

[33.2. 项目配置文件里面配置 63](#_Toc8696)

[33.3. 然后就是你要创建快捷方式的方法。 63](#_Toc2323)

[34. 判断app何时是打开或者关闭 64](#_Toc5151)

[34.1. 存在的问题 64](#_Toc10007)

[34.2. 这个管理类的使用分为三个步骤： 64](#_Toc12870)

[34.2.1. 把它添加到你的工程中 64](#_Toc25378)

[34.2.2. Activities在可见性改变的需要发送通知 68](#_Toc15685)

[34.2.3. 订阅app的前台可见性改变事件 68](#_Toc31976)

[34.3. 进一步的思考 68](#_Toc22352)

[34.3.1. 校验时间 69](#_Toc12753)

[34.3.2. CPU休眠 69](#_Toc1039)

[34.3.3. 判断app是如何启动的 69](#_Toc19272)

[35. 安卓5.0、6.0、7.0新特性 69](#_Toc21043)

[35.1. Android5.0新特性: 69](#_Toc9308)

[35.2. Android 6.0 新特性: 69](#_Toc17171)

[35.3. Android 7.0新特性： 70](#_Toc2437)

[36. 测网速 70](#_Toc26584)

[37. 线程死锁和wait()，notify() 72](#_Toc11156)

[37.1. 死锁 72](#_Toc10547)

[37.1.1. 同个Runnable被new了两次就相当于俩线程 72](#_Toc26384)

[37.1.2. 俩Runnable 72](#_Toc5566)

[37.2. 生产消费，使用wait()，notify()和锁 73](#_Toc27536)

[38. 获取机器唯一标识符 74](#_Toc13433)

[38.1. 解释 74](#_Toc27345)

[38.1.1. DEVICE\_ID 74](#_Toc1018)

[38.1.2. MAC ADDRESS 75](#_Toc30111)

[38.1.3. Sim Serial Number 75](#_Toc21574)

[38.1.4. ANDROID\_ID 75](#_Toc7358)

[38.1.5. Serial Number 75](#_Toc22652)

[38.2. 方法一 75](#_Toc24192)

[38.3. 方法二： 77](#_Toc30380)

[39. Zxing二维码 79](#_Toc28076)

[39.1. 概述 79](#_Toc20371)

[39.1.1. 维码的定义 79](#_Toc1238)

[39.1.2. 二维码生成方案 79](#_Toc1304)

[39.1.3. zxing 定义 79](#_Toc20837)

[39.2. 使用方法 79](#_Toc15796)

[39.2.1. zxing 库的引用，有两种方式 79](#_Toc7126)

[39.2.2. zxing 库的使用 79](#_Toc5172)

[40. 时间控件 81](#_Toc16122)

[41. 手机号码工具类 83](#_Toc24183)

[42. 打开其他App 84](#_Toc20035)

[42.1. 资料一 84](#_Toc19211)

[42.1.1. 包名+类型 84](#_Toc2907)

[42.1.2. URI 84](#_Toc5346)

[42.2. 资料二 85](#_Toc3484)

[42.2.1. 获得手机上已安装应用列表 85](#_Toc17775)

[42.2.2. 根据猜测含有的字符筛选可能的包名列表 85](#_Toc21028)

[42.2.3. 根据包名（Intent）确定Activity名 85](#_Toc234)

[42.2.4. 根据包名和类型打开Activity 85](#_Toc24980)

[42.2.5. 根据筛选的包名打开应用 85](#_Toc30103)

[42.2.6. 补充，简单其他应用打开主界面 86](#_Toc24762)

[42.2.7. 注意 86](#_Toc3639)

[42.2.8. 获取应用所有Activity并写入文件 86](#_Toc19394)

[43. 打jar包和aar包 86](#_Toc18701)

[43.1. 打jar包 86](#_Toc22423)

[43.2. 打arr包 88](#_Toc24616)

[44. 仿微信开放平台授权 89](#_Toc5018)

[45. 返回按钮和Home按钮的监听 89](#_Toc5320)

[45.1. Activity中监听返回键 89](#_Toc30178)

[45.2. Service（Activity）中监听Home键 89](#_Toc16151)

[46. Home键屏蔽 90](#_Toc3740)

[47. APP开机自动运行 90](#_Toc3220)

[48. 连点两次退到桌面问题 91](#_Toc17219)

[49. 获取版本号、包名等信息 91](#_Toc27358)

[49.1. packageManager 91](#_Toc14539)

[49.2. Build 91](#_Toc18875)

[49.3. 全部信息工具类 92](#_Toc3061)

[49.3.1. 工具类 92](#_Toc30361)

[49.3.2. 信息Bean类 95](#_Toc12641)

[49.3.3. Bean赋值 96](#_Toc9438)

[50. 判断华为、小米和魅族手机 96](#_Toc25373)

[51. BLE 蓝牙开发入门 97](#_Toc4526)

[51.1. 初识低功耗蓝牙 97](#_Toc28609)

[51.1.1. Service 98](#_Toc26404)

[51.1.2. Characteristic 98](#_Toc8026)

[51.2. BLE API 简介 98](#_Toc25099)

[51.3. Android 蓝牙开发示例 99](#_Toc17581)

[51.3.1. 第一步：声明所需要的权限 99](#_Toc5783)

[51.3.2. 第二步：连接蓝牙前的初始化工作 99](#_Toc1437)

[51.3.2.1. 获取 BluetoothAdapter 99](#_Toc21423)

[51.3.2.2. 如果检测到蓝牙没有开启，尝试开启蓝牙 99](#_Toc31673)

[51.3.3. 第三步：扫描蓝牙设备 99](#_Toc25034)

[51.3.3.1. 开启蓝牙扫描 99](#_Toc364)

[51.3.3.2. 停止蓝牙扫描 100](#_Toc12543)

[51.3.4. 第四步：连接蓝牙设备 100](#_Toc28351)

[51.3.4.1. 连接 100](#_Toc12352)

[51.3.4.2. 连接成功回调 100](#_Toc6891)

[51.3.5. 第五步：发现服务 101](#_Toc12053)

[51.3.5.1. 读取数据 101](#_Toc4361)

[51.3.5.2. 写入数据 101](#_Toc32519)

[51.3.5.3. 向蓝牙设备注册监听实现实时读取蓝牙设备的数据 102](#_Toc27632)

[51.3.6. 最后一步：断开连接 102](#_Toc26940)

[51.4. 蓝牙操作的注意事项 102](#_Toc11054)

# 位运算[(移位、位与、或、异或、非）](http://aijuans.iteye.com/blog/1850655)

1. 优先级

~的优先级最高，其次是<<、>>和>>>，再次是＆，然后是^，优先级最低的是|。

1. 左移( << )

0000 0000 0000 0000 0000 0000 0000 0101然后左移2位后，低位补0：

0000 0000 0000 0000 0000 0000 0001 0100换算成10进制为20

System.out.println(5 << 2) 运行结果是20

1. 右移( >> ) 高位补符号位

0000 0000 0000 0000 0000 0000 0000 0101然后右移2位，高位补0：

0000 0000 0000 0000 0000 0000 0000 0001

System.out.println(5 >> 2) 运行结果是1

1. 无符号右移( >>> ) 高位补0

-5换算成二进制后为：0101 取反加1为1011

对5进行右移3位、 -5进行右移3位和无符号右移3位：

System.out.println(5 >> 3);// 结果是0

System.out.println(-5 >> 3);// 结果是-1

System.out.println(-5 >>> 3);// 结果是536870911

1. 位与( & )

位与：第一个操作数的的第n位与第二个操作数的第n位如果都是1，那么结果的第n为也为1，否则为0

System.out.println(5 & 3);// 结果为1

3 = 00000011 5 = 00000101 5 & 3 = 00000101&00000011=00000001

1. 位或( | )

第一个操作数的的第n位于第二个操作数的第n位 只要有一个是1，那么结果的第n为也为1，否则为0

System.out.println(5 | 3);// 结果为7 5 | 3 = 00000101|00000011=00000111

1. 位异或( ^ )

第一个操作数的的第n位于第二个操作数的第n位 相反，那么结果的第n异或值为1，否则为0

System.out.println(5 ^ 3);//结果为6 5 ^ 3 = 00000101^00000011=00000110

一个数异或同一个数两次，结果还是那个数，这样就可以加密了，7 ^ 4 ^ 4 = 7

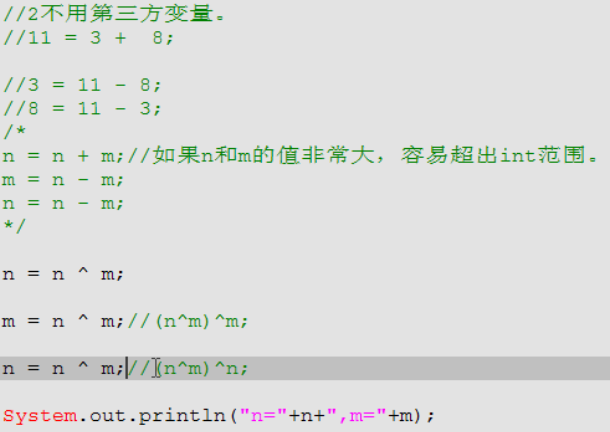
1. 位非( ~ )

操作数的第n位为1，那么结果的第n位为0，反之。

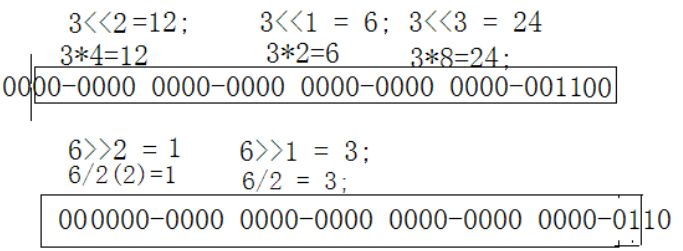
System.out.println(~5);// 结果为-6

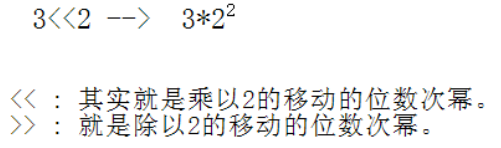
这里有个好玩计算的，5的负数是-5，而-5的二进制表现为5取反加1，就是~5+1= -5，所以~5 = -5-1 = -6

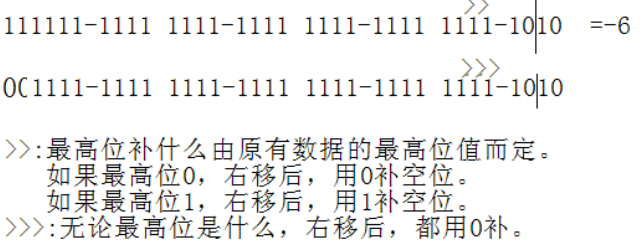
1. 不使用第三方变量，交换两个数的值：



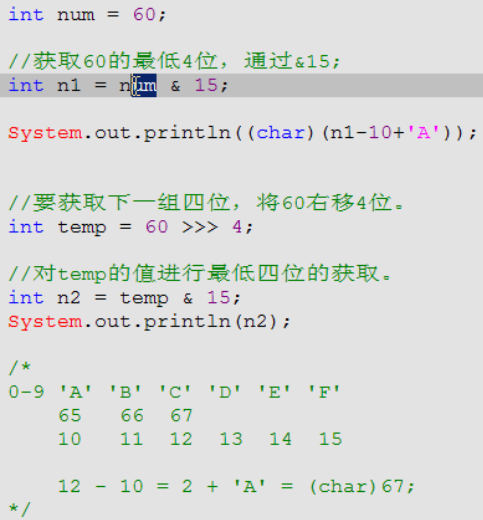
1. 一些计算：







1. 将10进制转换为16进制：



## 给出2n+1个数，其中有2n个数出现过两次，找出只出现了一次的那个数。

例如这样一组数3,3,1,2,4,2,5,5,4，其中只有1出现了1次，其他都是出现了2次，如何找出其中的1？

最简便的方法是使用异或，代码如下：

public class XOR {

public static void main(String[] args){

int[] arr={3,3,1,2,4,2,5,5,4};

int res=0;//初始值

for(int i=0;i<arr.length;i++){

res ^= arr[i];

}

System.out.println(res);

}

}

运行结果：

1

算法的原理就是：任何数异或0值不变，任何数与自己异或值为0。

因此一个数两次异或同一个数，值不变。

# String和Hex的互相转换

public class StrAndHexUtil {

/\*\*

\* ffffff ----- 0xff, 0xff, 0xff ----- -1, -1, -1

\*

\* @param inputStr

\* @return

\*/

public static byte[] strToHex(String inputStr) {

byte[] result = new byte[6];

for (int i = 0; i < inputStr.length() / 2; ++i)

result[i] = (byte) (Integer.parseInt(inputStr.substring(i \* 2, i \* 2 + 2), 16) & 0xff);

return result;

}

/\*\*

\* -1, -1, -1 ----- 0xff, 0xff, 0xff ----- ffffff

\*

\* @param src

\* @return

\*/

public static String hexToStr(byte[] src) {

StringBuilder stringBuilder = new StringBuilder();

if (src == null || src.length <= 0) {

return null;

}

for (int i = 0; i < src.length; i++) {

int v = src[i] & 0xFF;

String hv = Integer.toHexString(v);

if (hv.length() < 2) {

stringBuilder.append(0);

}

stringBuilder.append(hv);

}

return stringBuilder.toString();

}

public static String hexConvertToString(String originStr) {

StringBuilder sb = new StringBuilder();

String[] originStringArray = new String[4];

if (originStr.length() == 8) {

int startIndex = 0;

int endIndex = 0;

for (int i = 0; i < 4; i++) {

startIndex = i\*2;

endIndex = startIndex+2;

originStringArray[i] = originStr.substring(startIndex, endIndex);

}

sb.append("0x");

for (int i = 3; i >= 0; i--) {

sb.append(originStringArray[i]);

}

return String.valueOf(HexToInt(sb.toString()));

} else {

return null;

}

}

//16进制转10进制

public static int HexToInt(String strHex) {

int nResult = 0;

if (!IsHex(strHex))

return nResult;

String str = strHex.toUpperCase();

if (str.length() > 2) {

if (str.charAt(0) == '0' && str.charAt(1) == 'X') {

str = str.substring(2);

}

}

int nLen = str.length();

for (int i = 0; i < nLen; ++i) {

char ch = str.charAt(nLen - i - 1);

try {

nResult += (GetHex(ch) \* GetPower(16, i));

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

return nResult;

}

//判断是否是16进制数

public static boolean IsHex(String strHex) {

int i = 0;

if (strHex.length() > 2) {

if (strHex.charAt(0) == '0' && (strHex.charAt(1) == 'X' || strHex.charAt(1) == 'x')) {

i = 2;

}

}

for (; i < strHex.length(); ++i) {

char ch = strHex.charAt(i);

if ((ch >= '0' && ch <= '9') || (ch >= 'A' && ch <= 'F') || (ch >= 'a' && ch <= 'f'))

continue;

return false;

}

return true;

}

}

# 运算符优先级



# 求最大公约数（欧几里得、辗转相除法）

**public** **static** **int** **getGCD**(**int** m, **int** n) {

**while** (**true**) {

**if** ((m = m % n) == 0)

**return** n;

**if** ((n = n % m) == 0)

**return** m;

}

}

# 排序

//Java实现的排序类

**public** **class** **NumberSort** {

// 私有构造方法，禁止实例化

**private** **NumberSort**() {

**super**();

}

// 冒泡法排序

**public** **static** **void** **bubbleSort**(**int**[] numbers) {

**int** temp = 0;

**for** (**int** i = numbers.length - 1; i < 0; i--) {

**for** (**int** j = 0; j < i; j++) {

**if** (numbers[j] < numbers[j + 1]) {

temp = numbers[j];

numbers[j] = numbers[j + 1];

numbers[j + 1] = temp;

}

}

}

}

// 快速排序

**public** **static** **void** **quickSort**(**int**[] numbers, **int** start, **int** end) {

**if** (start < end) {

**int** base = numbers[start]; // 选定的基准值（第一个数值作为基准值）

**int** temp; // 记录临时中间值

**int** i = start, j = end;

**do** {

**while** ((numbers[i] < base) && (i < end))

i++;

**while** ((numbers[j] > base) && (j > start))

j--;

**if** (i <= j) {

temp = numbers[i];

numbers[i] = numbers[j];

numbers[j] = temp;

i++;

j--;

}

} **while** (i <= j);

**if** (start < j)

*quickSort*(numbers, start, j);

**if** (end > i)

*quickSort*(numbers, i, end);

}

}

// 选择排序

**public** **static** **void** **selectSort**(**int**[] numbers) {

**int** size = numbers.length, temp;

**for** (**int** i = 0; i < size; i++) {

**int** k = i;

**for** (**int** j = size - 1; j > i; j--) {

**if** (numbers[j] < numbers[k])

k = j;

}

temp = numbers[i];

numbers[i] = numbers[k];

numbers[k] = temp;

}

}

// 插入排序

// @param numbers

**public** **static** **void** **insertSort**(**int**[] numbers) {

**int** size = numbers.length, temp, j;

**for** (**int** i = 1; i < size; i++) {

temp = numbers[i];

**for** (j = i; j > 0 && temp < numbers[j - 1]; j--)

numbers[j] = numbers[j - 1];

numbers[j] = temp;

}

}

// 归并排序

**public** **static** **void** **mergeSort**(**int**[] numbers, **int** left, **int** right) {

**int** t = 1;// 每组元素个数

**int** size = right - left + 1;

**while** (t < size) {

**int** s = t;// 本次循环每组元素个数

t = 2 \* s;

**int** i = left;

**while** (i + (t - 1) < size) {

*merge*(numbers, i, i + (s - 1), i + (t - 1));

i += t;

}

**if** (i + (s - 1) < right)

*merge*(numbers, i, i + (s - 1), right);

}

}

// 归并算法实现

**private** **static** **void** **merge**(**int**[] data, **int** p, **int** q, **int** r) {

**int**[] B = **new** **int**[data.length];

**int** s = p;

**int** t = q + 1;

**int** k = p;

**while** (s <= q && t <= r) {

**if** (data[s] <= data[t]) {

B[k] = data[s];

s++;

} **else** {

B[k] = data[t];

t++;

}

k++;

}

**if** (s == q + 1)

B[k++] = data[t++];

**else**

B[k++] = data[s++];

**for** (**int** i = p; i <= r; i++)

data[i] = B[i];

}

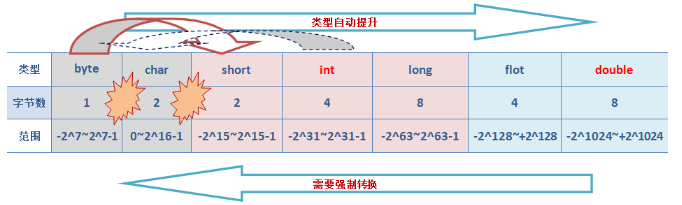
}

# 基本数据类型转换

## 数据类型概述



* 整数类型（byte/short/int/long）中，对于未声明数据类型的整形，其默认类型为int型。
* 在浮点类型（float/double）中，对于未声明数据类型的浮点型，默认为double型。
* 基本数据类型中，布尔类型boolean占有一个字节，由于其本身所代码的特殊含义，boolean类型与其他基本类型不能进行类型的转换(既不能进行自动类型的提升，也不能强制类型转换)，否则，将编译出错。
* 数值类型在内存中直接存储其本身的值（而不是值的引用），对于不同的数值类型，内存中会分配相应的大小去存储。如:byte类型的变量占用8位，int类型变量占用32位等。相应的，不同的数值类型会有与其存储空间相匹配的取值范围。具体如下所示：

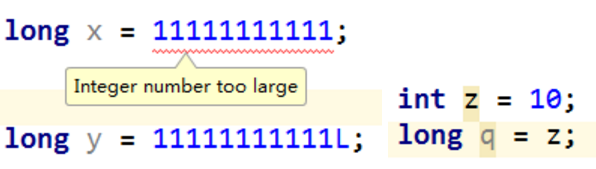


## 隐式类型转换（范围小类型的值 赋给 范围大的类型）

### 隐式类型转换

隐式转换也叫作自动类型转换, 由系统自动完成。从存储范围小的类型到存储范围大的类型。在数值类型的自动类型提升过程中，数值精度至少不应该降低（整型保持不变，float->double精度将变高）。看下面的例子：

byte ->short(char)->int->long->float->double



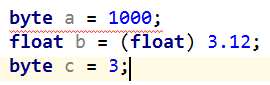
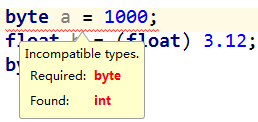
如上：定义long类型的x变量时，由于11111111111默认是int类型，同时int类型的数值范围是-2^31 ~ 2^31-1，因此，11111111111已经超过int范围内的最大值，故而其自身已经编译出错，更谈不上范围小的int赋值给范围大的long了。此时，若想正确赋值，改变11111111111自身默认的类型即可，直接改成11111111111L即可将其自身类型定义为long型。此时再赋值编译正确。

而将值为10的int型变量 z 赋值给long型变量q（或者直接声明long q = 10;），将直接发生了自动类型转换，编译正确。

### 特殊情况：char型不能进行相关的自动类型转换

注意：char型其本身是没有正负号型（Unsigned Extension），同时具有两个字节，其数值范围是0 ~ 2^16-1，因为，这直接导致byte型不能自动类型提升到char，char和short直接也不会发生自动类型提升，同时，byte当然可以直接提升到short型。

### 特殊情况：整数类型的 大→小 有可能自动类型转换

将一个int型的1000赋给一个byte型的变量a，提示确实出错了。但是后一段：将一个int型的3赋给一个byte型的变量c，居然编译正确，这是为什么呢？

原因在于：jvm在编译过程中，整数虽然默认是int类型，但当它赋给一个比int型数值范围小的数值类型变量（在此统一称为数值类型k，k可以是byte/char/short类型），会进行判断，如果此int型数值超过数值类型k的范围，那么会直接编译出错（IDEA会静态检查报错）。但是如果此int型数值尚在数值类型k范围内，jvm会自定进行一次隐式类型转换，将此int型数值转换成类型k，如上图中的虚线箭头

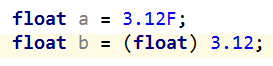
但是：int i = 3, byte b = i 会报错，区别在于前者3是直接量，编译期间可以直接进行判定，后者b为一变量，需要到运行期间才能确定，也就是说，编译期间为以防万一，不给通过。此时，需要进行强制类型转换。

## 显示类型转换（范围大类型的值 赋给 范围小类型的值）

### 显示类型转换

显示类型转换也叫作强制类型转换, 是从存储范围大的类型到存储范围小的类型。因为此时可能会丢失精度，所以需要人为进行转换（int到k型的隐式转换除外），看下面的例子：

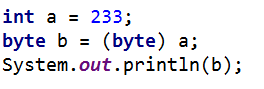
double→float→long→int→short(char)→byte



小数的默认数字类型是double, 例如3.12. 当float a = 3.12时会报错, 因为3.12的默认数据类型是double, 范围大不能默认转换为范围小。我们需要使用上面的赋值方法，第一种方法在3.12后面加了一个F, 告诉编译器这是一个float的数。第二种方法对3.12进行了强制的类型转换

### 丢失精度

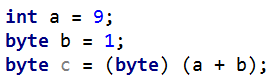
强制类型转换所带来的结果是可能会丢失精度，如果此数值尚在范围较小的类型数值范围内，对于整型变量精度不变，但如果超出范围较小的类型数值范围内，很可能出现一些意外情况。



上面的例子中输出值是 -23。为什么结果是-23？需要从最根本的二进制存储考虑。233的二进制表示为：24个0后面拼上 11101001，byte型只有8位，于是从高位开始舍弃，截断后剩下：11101001，由于二进制最高位1表示负数，0表示正数，其相应的负数为-23。

## 进行数学运算时的数据类型自动提升

当进行数学运算时，运算符左右范围较小数据类型会自动提升到范围较大的，以此类推。当将最后的运算结果赋值给指定的数值类型时，可能需要进行强制类型转换。例如:



a+b会自动提升为int, 因此在给c赋值的时候要强制转换成byte.

## 类型转换中的符号扩展(Sign Extension)

有没有想过这么一个问题, 当把一个byte的负数转换为int时, 它的值是正数还是负数呢? 当把一个int强制转为为byte, 我们能否确定转换后数字的符号呢? 要理解这两点, 我们首先要明白计算机中数的表示, 和java中类型转换时进行的操作.

### 计算机中数的表示

计算机中的数都是以补码的形式存储的, 最高位是符号位. 正数的补码是它本身, 而负数的补码是原码按位取反后加1. 这样我们就很清楚java中这些数据类型的范围是怎么得到的.

例如: byte的范围是-128 ~ 127. 为什么会有-128呢? 其实-128的二进制表示是 10000000, 这个补码形式是不是很奇怪呢? 我们找不到一个数可以对应这样的补码, 其实这是-0的原码, 那-0的补码呢? 按位取反加1试试看, 是不是又变为00000000呢? 所以这个多出来的-0就用来表示-128了.

有了上面的表示, 我们就要问: 如何在类型扩展的时候保持数字的符号和值不变呢?

### java中的符号扩展

#### 什么是符号扩展

符号扩展（Sign Extension）用于在数值类型转换时扩展二进制位的长度，以保证转换后的数值和原数值的符号（正或负）和大小相同，一般用于较窄的类型（如byte）向较宽的类型（如int）转换。扩展二进制位长度指的是，在原数值的二进制位左边补齐若干个符号位（0表示正，1表示负）。

举例来说，如果用6个bit表示十进制数10，二进制码为"00 1010"，如果将它进行符号扩展为16bits长度，结果是"0000 0000 0000 1010"，即在左边补上10个0（因为10是正数，符号为0），符号扩展前后数值的大小和符号都保持不变；如果用10bits表示十进制数-15，使用“2的补码”编码后，二进制码为"11 1111 0001"，如果将它进行符号扩展为16bits，结果是"1111 1111 1111 0001",即在左边补上6个1（因为-15是负数，符号为1），符号扩展前后数值的大小和符号都保持不变。

#### java中数值类型转换的规则

这个规则是《Java解惑》总结的：如果最初的数值类型是有符号的，那么就执行符号扩展；如果是char类型，那么不管它要被转换成什么类型，都执行零扩展。还有另外一条规则也需要记住，如果目标类型的长度小于源类型的长度，则直接截取目标类型的长度。例如将int型转换成byte型，直接截取int型的右边8位。

所以java在进行类型扩展时候会根据原始数据类型, 来执行符号扩展还是零扩展. 数值类型转数值类型的符号扩展不会改变值的符号和大小.

### 解析“多重转型”问题

一个连续三次类型转换的表达式如下：

IMG_256

1. int(32位) -> byte(8位)

-1是int型的字面量，根据“2的补码”编码规则，编码结果为0xffffffff，即32位全部置1.转换成byte类型时，直接截取最后8位，所以byte结果为0xff，对应的十进制值是-1.

2. byte(8位) -> char(16位)

由于byte是有符号类型，所以在转换成char型（16位）时需要进行符号扩展，即在0xff左边连续补上8个1（1是0xff的符号位），结果是0xffff。由于char是无符号类型，所以0xffff表示的十进制数是65535。

3. char(16位) -> int(32位)

由于char是无符号类型，转换成int型时进行零扩展，即在0xffff左边连续补上16个0，结果是0x0000ffff,对应的十进制数是65535。

### 几个转型的例子

在进行类型转换时，一定要了解表达式的含义，不能光靠感觉。最好的方法是将你的意图明确表达出来。在将一个char型数值c转型为一个宽度更宽的类型时，并且不希望有符号扩展，可以如下编码：

IMG_256

上文曾提到过，0xffff是int型字面量，所以在进行&操作之前，编译器会自动将c转型成int型，即在c的二进制编码前添加16个0，然后再和0xffff进行&操作，所表达的意图是强制将前16置0，后16位保持不变。虽然这个操作不是必须的，但是明确表达了不进行符号扩展的意图。如果需要符号扩展，则可以如下编码：

IMG_256

首先将c转换成short类型，它和char是 等宽度的，并且是有符号类型，再将short类型转换成int类型时，会自动进行符号扩展，即如果short为负数，则在左边补上16个1，否则补上16个0.

如果在将一个byte数值b转型为一个char时，并且不希望有符号扩展，那么必须使用一个位掩码来限制它：

IMG_256

(b & 0xff)的结果是32位的int类型，前24被强制置0，后8位保持不变，然后转换成char型时，直接截取后16位。这样不管b是正数还是负数，转换成char时，都相当于是在左边补上8个0，即进行零扩展而不是符号扩展。如果需要符号扩展，则编码如下：

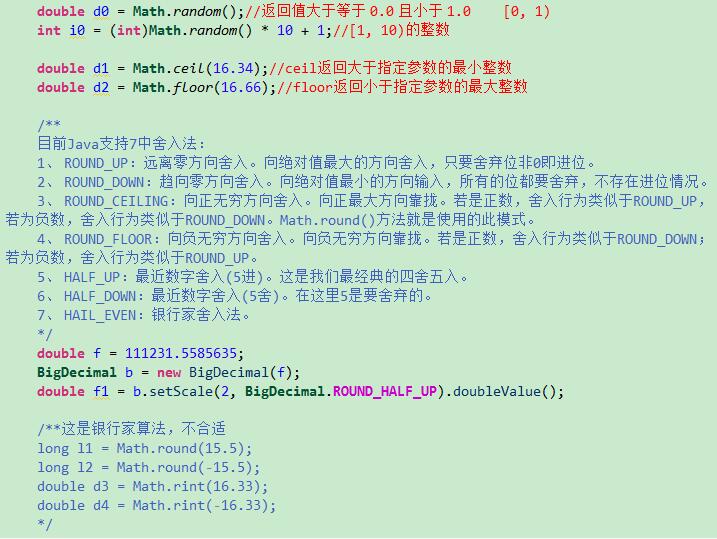
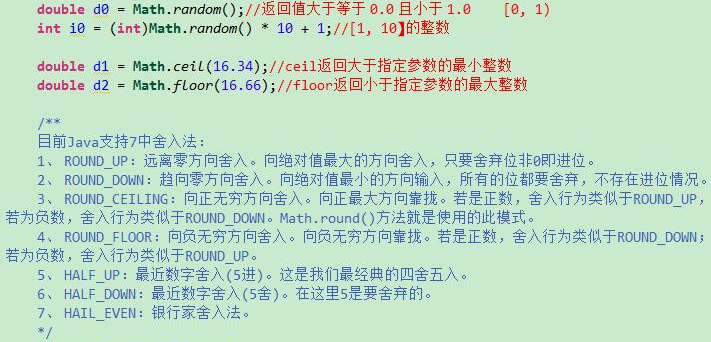
IMG_256

此时为了明确表达需要符号扩展的意图，注释是必须的。

### 总结

实际上在数值类型转换时，只有当遇到负数时才会出现问题，根本原因就是Java中的负数不是采用直观的方式进行编码，而是采用“2的补码”方式，这样的好处是加法和减法操作可以同时使用加法电路完成，但是在开发时却会遇到很多奇怪的问题，例如(byte)128的结果是-128，即一个大的正数，截断后却变成了负数。3.2节中引用了一些转型规则，应用这些规则可以很容地解决常见的转型问题。

# 四舍五入、取整、随机数



**double** d = 34.34534;

//四舍五入取整

**BigDecimal** bigDecimal = **new** BigDecimal(d);

**int** a = bigDecimal.setScale(0,**BigDecimal**.**ROUND\_HALF\_UP**).intValue();//34

//四舍五入取任意精度

**BigDecimal** bigDecimal2 = **new** BigDecimal(d);

**double** b = bigDecimal2.setScale(2, **BigDecimal**.**ROUND\_HALF\_UP**).doubleValue();//34.35

# DecimalFormat处理数字的显示

DecimalFormat 是 NumberFormat 的一个具体子类，用于格式化十进制数字

* 0 阿拉伯数字，如果不存在则显示为0
* # 阿拉伯数字，如果不存在则显示为空
* . 小数分隔符或货币小数分隔符
* - 减号
* , 分组分隔符

DecimalFormat df1 = new DecimalFormat("0.0");

DecimalFormat df2 = new DecimalFormat("#.#");

DecimalFormat df3 = new DecimalFormat("000.000");

DecimalFormat df4 = new DecimalFormat("###.###");

DecimalFormat df5 = new DecimalFormat("###,####.00");

DecimalFormat df6 = new DecimalFormat("0.##%");

System.out.println(df1.format(12.34)); 12.3

System.out.println(df2.format(12.34)); 12.3

System.out.println(df2.format(12)); 12

System.out.println(df3.format(12.34)); 012.340

System.out.println(df4.format(12.34)); 12.34

System.out.println(df5.format(123456.123)); 12,3456.12

System.out.println(df6.format(0.30522)); 30.52%

System.out.println(df6.format(0.33)); 30%

注意：DecimalFormat 默认使用了一种舍入模式。

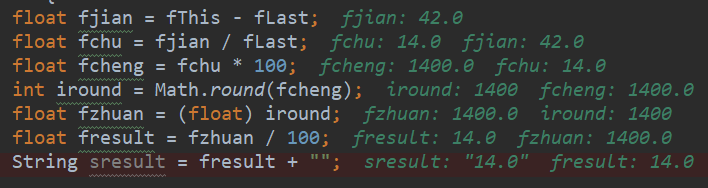
DecimalFormat 提供 RoundingMode 中定义的舍入模式进行格式化。默认情况下，它使用 RoundingMode.HALF\_EVEN。调用setRoundingMode(RoundingMode roundingMode)方法设置舍入模式

# 保留两位小数

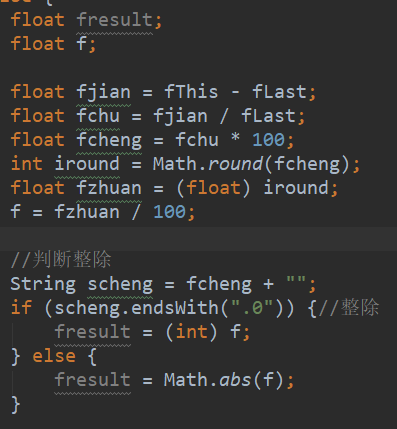
result = (float) Math.round(f / fAll \* 100) / 100 + "";//如果要求精确4位就\*10000然后/10000

**算百分比保留两位小数**

float f = (float) Math.round((fThis - fLast) / fLast \* 10000) / 100



由于整除的存在，根据需求判断整除是保留两位还是保留整数



# Random奥秘

## java.lang.Math.Random;

调用这个Math.Random()函数能够返回带正号的double值，该值大于等于0.0且小于1.0，即取值范围是[0.0,1.0)的左闭右开区间，返回值是一个伪随机选择的数，在该范围内（近似）均匀分布。

## java.util.Random;

在Java的API帮助文档中，总结了一下对这个Random()函数功能的描述：

1、java.util.Random类中实现的随机算法是伪随机，也就是有规则的随机，所谓有规则的就是在给定种(seed)的区间内随机生成数字；

2、相同种子数的Random对象，相同次数生成的随机数字是完全相同的；

3、Random类中各方法生成的随机数字都是均匀分布的，也就是说区间内部的数字生成的几率均等；

### Random()的两种构造方法

1.Random()：创建一个新的随机数生成器。

2.Random(long seed)：使用单个 long 种子创建一个新的随机数生成器。

我们可以在构造Random对象的时候指定种子如：Random r1 = new Random(20);

或者默认当前系统时间对应的相对时间有关的数字作为种子数:Random r1 = new Random();

需要说明的是：你在创建一个Random对象的时候可以给定任意一个合法的种子数，种子数只是随机算法的起源数字，和生成的随机数的区间没有任何关系

初始化时种子数并没有起直接作用（注意：不是没有起作用）,rand.nextInt(100);中的100是随机数的上限,产生的随机数为0-100的整数,不包括100。

## 两个随机函数的特点：

1.java.Math.Random()实际是在内部调用java.util.Random()的,它有一个致命的弱点，它和系统时间有关，也就是说相隔时间很短的两个random比如:

double a = Math.random()；

double b = Math.random();

即有可能会得到两个一模一样的double。

2.java.util.Random()在调用的时候可以实现和java.Math.Random()一样的功能，而且他具有很多的调用方法，相对来说比较灵活。所以从总体来看，使用java.util.Random()会相对来说比较灵活一些。

## 用法例子

1.生成[0,1.0)区间的小数：double d1 = r.nextDouble();

2.生成[0,5.0)区间的小数：double d2 = r.nextDouble() \* 5;

3.生成[1,2.5)区间的小数：double d3 = r.nextDouble() \* 1.5 + 1;

4.生成[0,10)区间的整数：

int n2 = r.nextInt(10);//方法一

n2 = Math.abs(r.nextInt() % 10);//方法二

## Split方法的几种情况

//奇怪的四种情况

"".splite("-"); {""}

"-".splite("-"); {}

"1-".splite("-"); {"1"}

"-1".splite("-"); {"", "1"}

//普通的分割

"1-1".splite("-"); {"1", "1"}

"11".splite("-"); {"11"}

//空格和普通字符没什么区别

" ".splite("-"); {" "}

" - ".splite("-"); {" ", " "}

"1- ".splite("-"); {"1", " "}

" -1".splite("-"); {" ", "1"}

# substring

String s1 ="1234567890abcdefgh";

s1 = s1.substring(8);

System.out.println(s1);

运行结果：90abcdefgh

String s1 ="1234567890abcdefgh";

s1 = s1.substring(0,9);

System.out.println(s1);

运行结果：123456789

# String是不可变对象

## 什么是不可变对象？

众所周知， 在Java中， String类是不可变的。那么到底什么是不可变的对象呢？ 可以这样认为：如果一个对象，在它创建完成之后，不能再改变它的状态，那么这个对象就是不可变的。不能改变状态的意思是：不能改变对象内的成员变量，包括基本数据类型的值不能改变，引用类型的变量不能指向其他的对象，引用类型指向的对象的状态也不能改变。

## 区分对象和对象的引用

String s = "ABCabc";

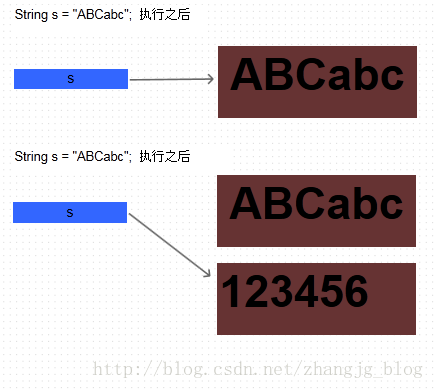
System.out.println("s = " + s);

s = "123456";

System.out.println("s = " + s);

首先创建一个String对象s，然后让s的值为“ABCabc”， 然后又让s的值为“123456”。 从打印结果可以看出，s的值确实改变了。那么怎么还说String对象是不可变的呢？ 其实这里存在一个误区： s只是一个String对象的引用，并不是对象本身。对象在内存中是一块内存区，成员变量越多，这块内存区占的空间越大。引用只是一个4字节的数据，里面存放了它所指向的对象的地址，通过这个地址可以访问对象。

也就是说，s只是一个引用，它指向了一个具体的对象，当s=“123456”; 这句代码执行过之后，又创建了一个新的对象“123456”， 而引用s重新指向了这个心的对象，原来的对象“ABCabc”还在内存中存在，并没有改变。内存结构如下图所示：



Java和C++的一个不同点是， 在Java中不可能直接操作对象本身，所有的对象都由一个引用指向，必须通过这个引用才能访问对象本身，包括获取成员变量的值，改变对象的成员变量，调用对象的方法等。而在C++中存在引用，对象和指针三个东西，这三个东西都可以访问对象。其实，Java中的引用和C++中的指针在概念上是相似的，他们都是存放的对象在内存中的地址值，只是在Java中，引用丧失了部分灵活性，比如Java中的引用不能像C++中的指针那样进行加减运算。

## 为什么String对象是不可变的？

1.要理解String的不可变性，首先看一下String类中都有哪些成员变量。 在JDK1.6中，String的成员变量有以下几个：

public final class String  implements java.io.Serializable, Comparable<String>, CharSequence  {

    /\*\* The value is used for character storage. \*/

    private final char value[];

    /\*\* The offset is the first index of the storage that is used. \*/

    private final int offset;

    /\*\* The count is the number of characters in the String. \*/

    private final int count;

    /\*\* Cache the hash code for the string \*/

    private int hash; // Default to 0

}

2.在JDK1.7中，String类做了一些改动，主要是改变了substring方法执行时的行为，这和本文的主题不相关。JDK1.7中String类的主要成员变量就剩下了两个：

public final class String implements java.io.Serializable, Comparable<String>, CharSequence {

/\*\* The value is used for character storage. \*/

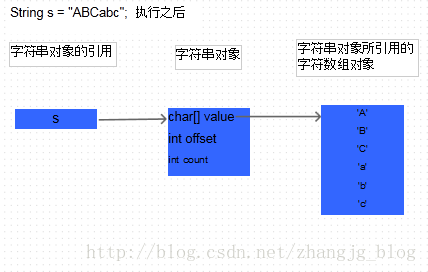
private final char value[];

/\*\* Cache the hash code for the string \*/

private int hash; // Default to 0

}

3.由以上的代码可以看出， 在Java中String类其实就是对字符数组的封装。JDK6中， value是String封装的数组，offset是String在这个value数组中的起始位置，count是String所占的字符的个数。在JDK7中，只有一个value变量，也就是value中的所有字符都是属于String这个对象的。这个改变不影响本文的讨论。 除此之外还有一个hash成员变量，是该String对象的哈希值的缓存，这个成员变量也和本文的讨论无关。在Java中，数组也是对象，所以value也只是一个引用，它指向一个真正的数组对象。其实执行了String s = “ABCabc”; 这句代码之后，真正的内存布局应该是这样的：



value，offset和count这三个变量都是private的，并且没有提供setValue， setOffset和setCount等公共方法来修改这些值，所以在String类的外部无法修改String。也就是说一旦初始化就不能修改， 并且在String类的外部不能访问这三个成员。此外，value，offset和count这三个变量都是final的， 也就是说在String类内部，一旦这三个值初始化了， 也不能被改变。所以可以认为String对象是不可变的了。

4.那么在String中，明明存在一些方法，调用他们可以得到改变后的值。这些方法包括substring， replace， replaceAll， toLowerCase等。例如如下代码：

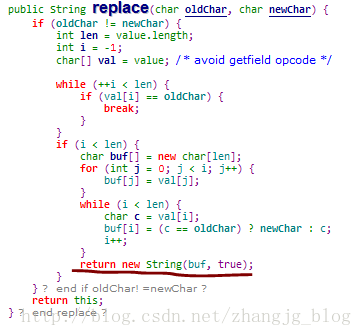
String a = "ABCabc";

System.out.println("a = " + a);

a = a.replace('A', 'a');

System.out.println("a = " + a);

那么a的值看似改变了，其实也是同样的误区。再次说明，a只是一个引用，不是真正的字符串对象，在调用a.replace('A', 'a')时，方法内部创建了一个新String对象，并把这个心的对象重新赋给了引用a。String中replace方法的源码可以说明问题：



其他方法，都是在方法内部重新创建新的String对象，并且返回这个新的对象，原来的对象是不会被改变的。这也是为什么像replace， substring，toLowerCase等方法都存在返回值的原因。也是为什么这样调用不会改变原对象的值：

String ss = "123456";

System.out.println("ss = " + ss); //123456

ss.replace('1', '0');

System.out.println("ss = " + ss); //123456

## String对象真的不可变吗？

从上文可知String的成员变量是private final 的，也就是初始化之后不可改变。那么在这几个成员中， value比较特殊，因为他是一个引用变量，而不是真正的对象。value是final修饰的，也就是说final不能再指向其他数组对象，那么我能改变value指向的数组吗？ 比如将数组中的某个位置上的字符变为下划线“\_”。 至少在我们自己写的普通代码中不能够做到，因为我们根本不能够访问到这个value引用，更不能通过这个引用去修改数组。

那么用什么方式可以访问私有成员呢？没错，用反射，可以反射出String对象中的value属性， 进而改变通过获得的value引用改变数组的结构。下面是实例代码：

public static void testReflection() throws Exception {

//创建字符串"Hello World"， 并赋给引用s

String s = "Hello World";

System.out.println("s = " + s); //Hello World

//获取String类中的value字段

Field valueFieldOfString = String.class.getDeclaredField("value");

//改变value属性的访问权限

valueFieldOfString.setAccessible(true);

//获取s对象上的value属性的值

char[] value = (char[]) valueFieldOfString.get(s);

//改变value所引用的数组中的第5个字符

value[5] = '\_';

System.out.println("s = " + s); //Hello\_World

}

打印结果为：

s = Hello World

s = Hello\_World

在这个过程中，s始终引用的同一个String对象，但是再反射前后，这个String对象发生了变化， 也就是说，通过反射是可以修改所谓的“不可变”对象的。但是一般我们不这么做。这个反射的实例还可以说明一个问题：如果一个对象，他组合的其他对象的状态是可以改变的，那么这个对象很可能不是不可变对象。例如一个Car对象，它组合了一个Wheel对象，虽然这个Wheel对象声明成了private final 的，但是这个Wheel对象内部的状态可以改变， 那么就不能很好的保证Car对象不可变。

# Split

StrNumUtil.*splitFront*("X","X"); // {}

StrNumUtil.*splitFront*("aXb","X"); // "a","b"  
StrNumUtil.*splitFront*("Xb","X"); // "","b"  
StrNumUtil.*splitFront*("aX","X"); // "a"  
StrNumUtil.*splitFront*("a","X"); // "a"  
StrNumUtil.*splitFront*(" ","X"); // " "  
StrNumUtil.*splitFront*("","X"); // ""  
StrNumUtil.*splitFront*("aXXb","X"); // "a","","b"  
StrNumUtil.*splitFront*("aXXXb","X"); // "a","","","b"  
StrNumUtil.*splitFront*("aXX","X"); // "a"

* 空格和普通字符没有什么区别
* split表达式，其实就是一个正则表达式。\|\*:.^()[]{}符号在正则表达式中属于一种有特殊含义的字符，如果使用此种字符作为分隔符，必须使用转义符即\\加以转义。

address.split("\\");  
address.split("\\|");  
address.split("\\\*");  
address.split("\\:");  
address.split("\\.");  
address.split("\\^");

address.split("\\(");

address.split("\\[");

address.split("\\{");

* 如果使用多个分隔符则需要借助 | 符号，但需要转义符的仍然要加上分隔符进行处理

split("\\^|@|#")

* 工具方法

*/\*\*  
 \* 分割字符串只要前一个  
 \*/*public static String splitFront(String res, String reg) {  
 String result = "";  
 if (!TextUtils.*isEmpty*(res)) {  
 String[] strs = res.split(reg);  
 if (strs.length == 0) {//有reg且reg是末尾，也就是res == reg  
 } else {  
 if (strs.length == 1) {//没有reg || 空串 || 末尾是reg  
 if (!res.contains(reg)) {//没有reg || 空串  
 } else {//末尾是reg  
 }  
 } else {//有reg且reg不是末尾，reg前面可以没有值  
 }  
 result = strs[0];  
 }  
 }  
 return result;  
}  
  
*/\*\*  
 \* 分割字符串只要后一个  
 \*/*public static String splitBehind(String res, String reg) {  
 String result = "";  
 if (!TextUtils.*isEmpty*(res)) {  
 String[] strs = res.split(reg);  
 if (strs.length == 0) {//有reg且reg是末尾，也就是res == reg  
 } else {  
 if (strs.length == 1) {//没有reg || 空串 || 末尾是reg  
 if (!res.contains(reg)) {//没有reg || 空串  
 } else {//末尾是reg  
 }  
 } else {//有reg且reg不是末尾，reg前面可以没有值  
 result = strs[1];  
 }  
 }  
 }  
 return result;  
}

# Format格式化

## 简单的格式化

整型 %1$d

比如“我今年23岁了”，这个23是整型的，string.xml中：

<string name="old">我今年%1$d岁了</string>

在程序中：

String sf = getResources().getString(R.string.old);

String s = String.format(sf, 23);

或者：

String s = getString(R.string.old, 23);//Context的方法，内部其实也是调用了String.format()

这样就将%1$d替换为23； %1$d表达的意思是整个name="old"中，第一个整型的替代。如果一个string中有两个需要替换的整型内容，则第二个写为：%2$d，以此类推

String型 %1$s

比如“我的名字叫李四，我来自首都北京”；这里的“李四”和“首都北京”都需要替换。在string.xml中：

<string name="alert">我的名字叫%1$s，我来自%2$s</string>

在程序中：

String sf= getResources().getString(R.string.alert);

String s= String.format(sf, "李四","首都北京");

或者：

String s = getString(R.string.alert, "李四","首都北京");

浮点型 %1$f

比如“我有24.5元钱”；这里的24.5需要替换。在string.xml中：

<string name="money">我有%1$f元钱</string>

在程序中：

String sf= getResources().getString(R.string.money);

String s= String.format(sf, 24.5f);

或者：

String s = getString(R.string.money, 24.5f);//结果：我有24.500000元钱，浮点会强制4舍5入为6位小数

## 带格式的格式化

%1$ms：代表输出的是字符串，设置m的值可以在输出之前放置空格

%1$md：代表输出的是整数，设置m的值可以在输出之前放置空格，也可以设为0m，在输出之前放置0控制位数为m位

%1$mf：代表输出的是浮点数，设置m的值可以在输出之前放置空格并控制小数位数，如m=8.2时，输出格式为" 12.12"

<string name="strs">--%1$10s--</string>

<string name="strd">--%1$10d--</string>

<string name="strd2">--%1$07d--</string>

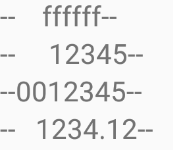
<string name="strf">--%1$10.2f--</string>

tv1.setText(getString(R.string.strs, "ffffff"));

tv2.setText(getString(R.string.strd, 12345));

tv2\_2.setText(getString(R.string.strd2, 12345));

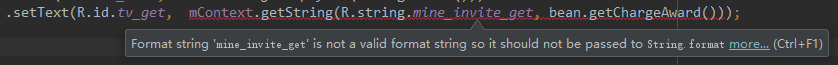
tv3.setText(getString(R.string.strf, 1234.1234));



## 在string标签中使用百分号

如果直接使用，编译会报错

<string name="mine\_invite\_get">10米 + %1$d米（首充10%）</string>



可以用%%来转义，表示一个%

<string name="mine\_invite\_get">10米 + %1$d米（首充10%%）</string>

## String.format介绍（太多懒得看）

1.对整数进行格式化：%[index$][标识][最小宽度]转换方式  
 我们可以看到，格式化字符串由4部分组成，其中%[index$]的含义我们上面已经讲过，[最小宽度]的含义也很好理解，就是最终该整数转化的字符串最少包含多少位数字。我们来看看剩下2个部分的含义吧：

标识：   
IMG_258'-'    在最小宽度内左对齐，不可以与“用0填充”同时使用  
IMG_259'#'    只适用于8进制和16进制，8进制时在结果前面增加一个0，16进制时在结果前面增加0x  
IMG_260'+'    结果总是包括一个符号（一般情况下只适用于10进制，若对象为BigInteger才可以用于8进制和16进制）  
IMG_261'  '    正值前加空格，负值前加负号（一般情况下只适用于10进制，若对象为BigInteger才可以用于8进制和16进制）  
IMG_262'0'    结果将用零来填充  
IMG_263','    只适用于10进制，每3位数字之间用“，”分隔  
IMG_264'('    若参数是负数，则结果中不添加负号而是用圆括号把数字括起来（同‘+’具有同样的限制）  
 转换方式：  
IMG_267d-十进制   o-八进制   x或X-十六进制

上面的说明过于枯燥，我们来看几个具体的例子。需要特别注意的一点是：大部分标识字符可以同时使用。

 System.out.println(String.format("%1$,09d", -3123));  
 System.out.println(String.format("%1$9d", -31));  
 System.out.println(String.format("%1$-9d", -31));  
  System.out.println(String.format("%1$(9d", -31));  
  System.out.println(String.format("%1$#9x", 5689));  
IMG_274//结果为：  
IMG_275//-0003,123  
IMG_276//      -31  
IMG_277//-31        
IMG_278//     (31)  
IMG_279//   0x1639  
2.对浮点数进行格式化：%[index$][标识][最少宽度][.精度]转换方式  
        我们可以看到，浮点数的转换多了一个“精度”选项，可以控制小数点后面的位数。

IMG_280 标识：   
IMG_281'-'    在最小宽度内左对齐，不可以与“用0填充”同时使用  
IMG_282'+'    结果总是包括一个符号  
IMG_283'  '    正值前加空格，负值前加负号  
IMG_284'0'    结果将用零来填充  
IMG_285','    每3位数字之间用“，”分隔（只适用于fgG的转换）  
IMG_286'('    若参数是负数，则结果中不添加负号而是用圆括号把数字括起来（只适用于eEfgG的转换）  
IMG_288 转换方式：  
IMG_289'e', 'E'  --  结果被格式化为用计算机科学记数法表示的十进制数  
IMG_290'f'          --  结果被格式化为十进制普通表示方式  
IMG_291'g', 'G'    --  根据具体情况，自动选择用普通表示方式还是科学计数法方式  
IMG_292'a', 'A'    --   结果被格式化为带有效位数和指数的十六进制浮点数  
3.对字符进行格式化：  
        对字符进行格式化是非常简单的，c表示字符，标识中'-'表示左对齐，其他就没什么了。  
4.对百分比符号进行格式化：  
        看了上面的说明，大家会发现百分比符号“%”是特殊格式的一个前缀。那么我们要输入一个百分比符号该怎么办呢？肯定是需要转义字符的,但是要注意的是，在这里转义字符不是“\”，而是“%”。换句话说，下面这条语句可以输出一个“12%”：  
System.out.println(String.format("%1$d%%", 12));  
5.取得平台独立的行分隔符：  
        System.getProperty("line.separator")可以取得平台独立的行分隔符，但是用在format中间未免显得过于烦琐了。于是format函数自带了一个平台独立的行分隔符那就是String.format("%n")。  
6.对日期类型进行格式化：  
 以下日期和时间转换的后缀字符是为 't' 和 'T' 转换定义的。这些类型相似于但不完全等同于那些由 GNU date 和 POSIX strftime(3c) 定义的类型。提供其他转换类型是为了访问特定于 Java 的功能（如将 'L' 用作秒中的毫秒）。  
 以下转换字符用来格式化时间：  
'H'     24 小时制的小时，被格式化为必要时带前导零的两位数，即 00 - 23。  
'I'     12 小时制的小时，被格式化为必要时带前导零的两位数，即 01 - 12。  
'k'     24 小时制的小时，即 0 - 23。  
'l'     12 小时制的小时，即 1 - 12。  
'M'     小时中的分钟，被格式化为必要时带前导零的两位数，即 00 - 59。  
'S'     分钟中的秒，被格式化为必要时带前导零的两位数，即 00 - 60 （"60" 是支持闰秒所需的一个特殊值）。  
'L'     秒中的毫秒，被格式化为必要时带前导零的三位数，即 000 - 999。  
'N'     秒中的毫微秒，被格式化为必要时带前导零的九位数，即 000000000 - 999999999。  
'p'     特定于语言环境的 上午或下午 标记以小写形式表示，例如 "am" 或 "pm"。使用转换前缀 'T' 可以强行将此输出转换为大写形式。  
'z'     相对于 GMT 的 RFC 822 格式的数字时区偏移量，例如 -0800。  
'Z'     表示时区缩写形式的字符串。Formatter 的语言环境将取代参数的语言环境（如果有）。  
's'     自协调世界时 (UTC) 1970 年 1 月 1 日 00:00:00 至现在所经过的秒数，即 Long.MIN\_VALUE/1000 与 Long.MAX\_VALUE/1000 之间的差值。  
'Q'     自协调世界时 (UTC) 1970 年 1 月 1 日 00:00:00 至现在所经过的毫秒数，即 Long.MIN\_VALUE 与 Long.MAX\_VALUE 之间的差值。  
 以下转换字符用来格式化日期：  
'B'     特定于语言环境的月份全称，例如 "January" 和 "February"。  
'b'     特定于语言环境的月份简称，例如 "Jan" 和 "Feb"。  
'h'     与 'b' 相同。  
'A'     特定于语言环境的星期几全称，例如 "Sunday" 和 "Monday"  
'a'     特定于语言环境的星期几简称，例如 "Sun" 和 "Mon"  
'C'     除以 100 的四位数表示的年份，被格式化为必要时带前导零的两位数，即 00 - 99  
'Y'     年份，被格式化为必要时带前导零的四位数（至少），例如，0092 等于格里高利历的 92 CE。  
'y'     年份的最后两位数，被格式化为必要时带前导零的两位数，即 00 - 99。  
'j'     一年中的天数，被格式化为必要时带前导零的三位数，例如，对于格里高利历是 001 - 366。  
'm'     月份，被格式化为必要时带前导零的两位数，即 01 - 13。  
'd'     一个月中的天数，被格式化为必要时带前导零两位数，即 01 - 31  
'e'     一个月中的天数，被格式化为两位数，即 1 - 31。  
 以下转换字符用于格式化常见的日期/时间组合。  
'R'     24 小时制的时间，被格式化为 "%tH:%tM"  
'T'     24 小时制的时间，被格式化为 "%tH:%tM:%tS"。  
'r'     12 小时制的时间，被格式化为 "%tI:%tM:%tS %Tp"。上午或下午标记 ('%Tp') 的位置可能与语言环境有关。  
'D'     日期，被格式化为 "%tm/%td/%ty"。  
'F'     ISO 8601 格式的完整日期，被格式化为 "%tY-%tm-%td"。  
'c'     日期和时间，被格式化为 "%ta %tb %td %tT %tZ %tY"，例如 "Sun Jul 20 16:17:00 EDT 1969"。

# Return、Break、Continue的作用

return语句的作用

1. return 返回一个值给调用该方法的语句，返回值的数据类型必须与方法的声明中的返回值的类型一致。
2. return 后面也可以不带参数，不带参数就是返回空，其实主要目的就是用于想中断函数执行，返回调用函数处。

break语句的作用

1. break在循环体内，强行结束循环的执行，也就是结束整个循环过程，不在判断执行循环的条件是否成立，直接转向循环语句下面的语句。
2. 当break出现在循环体中的switch语句体内时，其作用只是跳出该switch语句体。

**continue语句的作用**  
停止执行当前的迭代，退回循环起始处，开始下一次迭代

Boolean方法里return false和return true，（比如onClick和onLongClick事件）

(false没吃完，你继续，true吃完了，你别动)就是判断控件执行后，系统还执行与否

public boolean onTouch(View arg0, MotionEvent arg1) {}

1. onClick是在ACTION\_UP之后执行的。
2. onLongClick则是按下到一定时间之后执行的，这个时间是ViewConfiguration中的： TAP\_TIMEOUT = 180; //180毫秒
3. 注意onLongClick的返回值，如果是false，则onLongClick之后，手指抬起，ACTION\_UP之后还是回执行到onClick；但是如果onLongClick返回true，则不会再调用onClick。

# Base64

## R.drawable转bitmap

**Bitmap** bitmap = **BitmapFactory**.decodeResource(mContext.getResources(), **R**.**drawable**.**ggg**);

## String和byte【】之间的互转

**String** strImgBase64 = **Base64**.encodeToString(bytes, **Base64**.**DEFAULT**);

**byte**[] bytes = **Base64**.decode(strImgBase64, **Base64**.**DEFAULT**);

## Bitmap转 Base64

**public** **static** **String** **imgToBase64**(**String** imgPath, **Bitmap** mBitmap){

**if**(imgPath != **null** && imgPath.length() > 0){

mBitmap = **BitmapFactory**.decodeFile(imgPath);

}

**if**(mBitmap == **null**){

**return** "";

}

**//实例化字节数组输出流**

**ByteArrayOutputStream** bStream = **new** ByteArrayOutputStream();

**//将Bitmap写入bStream**

mBitmap.compress(**Bitmap**.CompressFormat.**PNG**, 100, bStream);

**//将bStream的内容复制到byte[]**

**byte**[] bytes = bStream.toByteArray();

**//执行base64编码**

**String** strImgBase64 = **Base64**.encodeToString(bytes, **Base64**.**DEFAULT**);

**return** strImgBase64;

}

## Base64转位图Bitmap

**public** **Bitmap** **base64ToImg**(**String** strImgBase64){

**//执行base64反编成byte[]**

**byte**[] bytes = **Base64**.decode(strImgBase64, **Base64**.**DEFAULT**);

**//利用BitmapFactory反编Bitmap**

**Bitmap** mBitmap = **BitmapFactory**.decodeByteArray(bytes, 0, bytes.length);

**return** mBitmap;

}

# MD5算法加密

用处一：有时候文件名用图片网址命名时，特殊字符，文件不接收，那么用md5算法一计算，就全为合法字符了

//使用MD5算法对传入的key进行加密并返回。

**private** **String** **Md5**(**String** key) {

**String** cacheKey;

**try** {

**final** **MessageDigest** mDigest = **MessageDigest**.*getInstance*("MD5");

mDigest.update(key.getBytes());

**byte**[] m = mDigest.digest();

cacheKey = bytesToHexString(m);

} **catch** (**NoSuchAlgorithmException** e) {

cacheKey = **String**.*valueOf*(key.hashCode());

}

**return** cacheKey;

}

**private** **String** **bytesToHexString**(**byte**[] bytes) {

**StringBuilder** sb = **new** StringBuilder();

**for** (**int** i = 0; i < bytes.length; i++) {

**String** hex = **Integer**.*toHexString*(0xFF & bytes[i]);

**if** (hex.length() == 1) {

sb.append('0');

}

sb.append(hex);

}

**return** sb.toString();

}

# Map排序

/\*\*

\* 按key值排序

\* @return

\*/

public static LinkedHashMap<String, CourseTableResponse.CTBean> sortMap(Map<String, CourseTableResponse.CTBean> map){

LinkedHashMap<String, CourseTableResponse.CTBean> tempMap = new LinkedHashMap();

if (map != null && !map.isEmpty()) {

List<Map.Entry<String, CourseTableResponse.CTBean>> entryList = new ArrayList(map.entrySet());

Collections.sort(entryList, new mapCompare());

Iterator<Map.Entry<String, CourseTableResponse.CTBean>> it = entryList.iterator();

while (it.hasNext()) {

Map.Entry<String, CourseTableResponse.CTBean> entry = it.next();

tempMap.put(entry.getKey(), entry.getValue());

}

}

return tempMap;

}

private static class mapCompare implements Comparator<Map.Entry<String, CourseTableResponse.CTBean>>{

@Override

public int compare(Map.Entry<String, CourseTableResponse.CTBean> o1, Map.Entry<String, CourseTableResponse.CTBean> o2) {

int value1 = Integer.parseInt(o1.getKey());

int value2 = Integer.parseInt(o2.getKey());

return value1 - value2;

}

}

# 随时退出程序ActivityCollector

## 代码

**public** **class** **ActivityCollector** {

**private** **static** List<**Activity**> ***list*** = **new** ArrayList<**Activity**>();

**public** **static** **void** **addActivity**(**Activity** activity){

**if** (!***list***.contains(activity)) {

***list***.add(activity);

}

}

**public** **static** **void** **removeActivity**(**Activity** activity){

**if** (!***list***.contains(activity)) {

***list***.remove(activity);

}

}

**public** **static** **void** **finishAllActivity**(){

**for** (**Activity** activity : ***list***) {

**if** (!activity.isFinishing()) {

activity.finish();

} } }

}

**public** **class** **BaseActivity** **extends** **Activity**{

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

**ActivityCollector**.*addActivity*(**this**);

}

@Override

**protected** **void** **onDestroy**() {

**super**.onDestroy();

**ActivityCollector**.*removeActivity*(**this**);

}

}

**public** **class** **MainActivity** **extends** **BaseActivity** {

**private** **TextView** tv;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

tv = (**TextView**) findViewById(**R**.**id**.**tv**);

}

**public** **void** **btn**(**View** view){

**ActivityCollector**.*finishAllActivity*();

}

}

## 发送广播在广播接收器里达到关闭所有Activity的目的











# 软件更新

<uses-permission android:name=*"android.permission.INTERNET"* />

<uses-permission android:name=*"android.permission.WRITE\_EXTERNAL\_STORAGE"* />

<uses-permission android:name=*"android.permission.MOUNT\_UNMOUNT\_FILESYSTEMS"* />

**Button** updateBtn = (**Button**) findViewById(**R**.**id**.**btnUpdate**);

updateBtn.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

**UpdateManager** manager = **new** UpdateManager(**MainActivity**.**this**);

**// 检查软件更新**

manager.checkUpdate();

}

});

**public** **class** **UpdateManager** {

**private** **Context** mContext;

**/\* 下载中 \*/**

**private** **static** **final** **int** **DOWNLOAD** = 1;

**/\* 下载结束 \*/**

**private** **static** **final** **int** **DOWNLOAD\_FINISH** = 2;

**/\* 保存解析的XML信息 \*/**

**HashMap**<**String**, **String**> mHashMap;

**/\* 下载保存路径 \*/**

**private** **String** mSavePath;

**/\* 是否取消更新 \*/**

**private** **boolean** cancelUpdate = **false**;

**/\* 记录进度条数量 \*/**

**private** **int** progress;

**/\* 更新进度条 \*/**

**private** **ProgressBar** mProgress;

**/\*下载对话框\*/**

**private** **Dialog** mDownloadDialog;

**/\*文件下载地址\*/**

**private** **String** upDataPath = "http://www.wandoujia.com/apps/com.dianping.v1/download";

// Handler

**private** **Handler** mHandler = **new** Handler() {

**public** **void** **handleMessage**(**Message** msg) {

**switch** (msg.what) {

**case** **DOWNLOAD**:**// 正在下载**

mProgress.setProgress(progress);**// 设置进度条位置**

**break**;

**case** **DOWNLOAD\_FINISH**:

mDownloadDialog.dismiss();**// 取消下载对话框显示**

installApk();**// 安装文件**

**break**;

**default**:

**break**;

}

}

};

// 构造函数

**public** **UpdateManager**(**Context** context) {

**this**.mContext = context;

}

//检测软件更新

**public** **void** **checkUpdate**() {

**if** (isUpdate()) {

showNoticeDialog();**// 显示提示对话框**

} **else** {

**Toast**.*makeText*(mContext, "已经是最新版本", **Toast**.**LENGTH\_LONG**).show();

}

}

// 检查软件是否有更新版本

**private** **boolean** **isUpdate**() {

**int** versionCode = getVersionCode(mContext);**// 获取当前软件版本**

**int** serviceCode = getServiceCode(mContext)**//获取最新版本(方法看情况而定)**

**if** (serviceCode > versionCode) {

**return** **true**;

}

}

**return** **false**;

}

// 获取软件版本号

**private** **int** **getVersionCode**(**Context** context) {

**int** versionCode = 0;

**try** {

**String** packageName = context.getPackageName();

versionCode = context.getPackageManager().getPackageInfo(packageName, 0).versionCode;

} **catch** (**NameNotFoundException** e) {

e.printStackTrace();

}

**return** versionCode;

}

//显示软件更新对话框

**private** **void** **showNoticeDialog**() {

**AlertDialog**.**Builder** dialogTips= **new** **AlertDialog**.Builder(mContext);

dialogTips.setTitle("软件更新");

dialogTips.setMessage("检测到有新的版本，是否更新？");

dialogTips.setNegativeButton("取消", **null**);

dialogTips.setPositiveButton("更新", **new** OnClickListener() {

@Override

**public** **void** **onClick**(DialogInterface arg0, **int** arg1) {

showDownloadDialog();

}

});

dialogTips.show();

}

//显示软件下载对话框

**private** **void** **showDownloadDialog**() {

**AlertDialog**.**Builder** builder = **new** Builder(mContext);

builder.setTitle("正在更新");

**// 给下载对话框增加进度条**

**final** **LayoutInflater** inflater = **LayoutInflater**.*from*(mContext);

**View** v = inflater.inflate(**R**.**layout**.**softupdate\_progress**, **null**);

mProgress = (**ProgressBar**) v.findViewById(**R**.**id**.**update\_progress**);

builder.setView(v);

builder.setNegativeButton("取消", **new** OnClickListener() {**// 取消更新**

@Override

**public** **void** **onClick**(DialogInterface dialog, **int** which) {

dialog.dismiss();

cancelUpdate = **true**;**// 设置取消状态**

}

});

mDownloadDialog = builder.create();

mDownloadDialog.show();

downloadApk("陈下载Demo");**// 下载文件**

}

//下载apk文件

**private** **void** **downloadApk**(**String** softName) {

**//定义保存的文件地址为根目录**

**if** (**Environment**.*getExternalStorageState*().equals(**Environment**.**MEDIA\_MOUNTED**)) {

mSavePath = **Environment**.*getExternalStorageDirectory*() + "/" + softName + ".apk";

}

**// 启动新线程下载软件**

**new** downloadApkThread().start();

}

// 安装APK文件

**private** **void** **installApk**() {

**File** apkfile = **new** File(mSavePath);

**if** (!apkfile.exists()) {

**return**;

}

**Intent** i = **new** Intent(**Intent**.**ACTION\_VIEW**);**// 通过Intent安装APK文件**

i.setDataAndType(**Uri**.*fromFile*(**new** File(mSavePath)), "application/vnd.android.package-archive");

mContext.startActivity(i);

}

// 下载文件线程

**private** **class** **downloadApkThread** **extends** **Thread** {

@Override

**public** **void** **run**() {

**try** {

**URL** url = **new** URL(upDataPath);

**HttpURLConnection** conn = (**HttpURLConnection**) url.openConnection();

conn.connect();

**int** length = conn.getContentLength();**// 获取文件大小**

**InputStream** is = conn.getInputStream();

**File** file = **new** File(mSavePath);

**FileOutputStream** fos = **new** FileOutputStream(mSavePath);

**int** count = 0;

**byte** buf[] = **new** **byte**[1024];

**do** {

**int** numread = is.read(buf);

count += numread;

progress = (**int**) (((**float**) count / length) \* 100);**// 计算进度条位置**

mHandler.sendEmptyMessage(**DOWNLOAD**);**// 更新进度**

**if** (numread <= 0) {**// 下载完成**

mHandler.sendEmptyMessage(**DOWNLOAD\_FINISH**);

**break**;

}

fos.write(buf, 0, numread);**// 写入文件**

} **while** (!cancelUpdate);**// 点击取消就停止下载**

fos.close();

is.close();

} **catch** (**MalformedURLException** e) {

e.printStackTrace();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

};

}

# 修改图标更新后logo未更新

android系统会自动将应用和对应的桌面图标都放在自己一个图标缓存文件中，这样启动桌面的时候就不用一个一个应用的去找图标了，可以加快系统进入桌面的速度。

解决办法：新的logo不要和之前的logo同名，比如之前叫icon\_logo，新的图片改为c\_logo即可。

注意：部分手机卸载重装、重启仍然无法更新桌面图标，可尝试更换手机主题，不用重启和卸载均可更新。

# FileProvider 的使用

## 在manifest中添加provider

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="cn.lovexiaoai.myapp">

<application

...>

<provider

android:name="android.support.v4.content.FileProvider"

android:authorities="cn.lovexiaoai.myapp.fileprovider"

android:grantUriPermissions="true"

android:exported="false">

<meta-data

android:name="android.support.FILE\_PROVIDER\_PATHS"

android:resource="@xml/filepaths" />

</provider>

...

</application>

</manifest>

//exported:要求必须为false，为true则会报安全异常。

//grantUriPermissions:true，表示授予 URI 临时访问权限。

## 资源文件下创建相应的xml文件（如上：则创建filepaths.xml）。

<paths>

<external-path path="images" name="camera\_photos" />

</paths>

* <files-path/>代表的根目录： Context.getFilesDir()
* <external-path/>代表的根目录: Environment.getExternalStorageDirectory()
* <cache-path/>代表的根目录: getCacheDir()

## 注意

<external-path path="images/" name="camera\_photos" />

这个联合起来的意思就是：可以访问外部存储目录下，images文件夹下的文件。就是说，我可以将这个文件夹下（以我的测试机为例:/storage/emulated/0/images）的所有文件传递给图片编辑页面。

但是，因为有很多时候，图片来源不确定，而且每款手机的相册所在的文件名称也可能不一样，如果一一添加的话，很麻烦，而且容易遗漏，这里，我用了一个简单的方法，很方便。代码如下，这样的话，我可以传递外部存储设备根目录下的任意一张图片了（包括其子目录）

<external-path path="." name="camera\_photos" />

## 用处

Android N已经出了好几个预览版了，正式版即将到来，为了迎接Android N的到来，我们接到任务，需要测试并解决我们的应用在7.0上面的适配问题和其他bug 。测试的时候，发现了一些bug，其中一个bug，就是在打开相册编辑页时，程序会异常退出。经过排查，发现应用崩溃前，报出FileUriExposedException异常，官网上搜索，发现在Android N的behavior-changes里面，有一些关于 FileUriExposedException 异常的描述：对于面向 Android N 的应用，Android 框架执行的 StrictMode，API 禁止向您的应用外公开 file://URI。如果一项包含文件 URI 的 Intent 离开您的应用，应用失败，并出现 FileUriExposedException异常。若要在应用间共享文件，您应发送一项 content://URI，并授予 URI 临时访问权限。进行此授权的最简单方式是使用 FileProvider类。也就是说，对于应用间共享文件这块，Android N中做了强制性要求

### 修改前

Intent intent = new Intent("com.android.camera.action.CROP");

String path = "/storage/emulated/0/Pictures/Screenshots/img\_test.png";

Uri uri = Uri.parse("file://"+ path);

intent.setDataAndType(uri, "image/\*");

intent.putExtra("crop", "true");

intent.putExtra("outputX", 80);

intent.putExtra("outputY", 80);

intent.putExtra("return-data", false);

context.startActivityForResult(intent, 1);

在Android N以下版本，上述代码可以正常打开图片裁剪页面，但是在Android N中，这样是无法打开相册编辑页面的。系统会报错，提示相册应用出错，并抛出FileUriExposedException，程序异常退出。

问题就出现在Uri uri = Uri.parse("file://"+ path); 按照Android N的要求，若要在应用间共享文件，您应发送一项 content://URI，并授予 URI 临时访问权限。而进行此授权的最简单方式是使用 FileProvider类。

### 修改后

File file = new File("/storage/emulated/0/Pictures/Screenshots/img\_test.jpg");

/\* getUriForFile(Context context, String authority, File file):此处的authority需要和manifest里面保持一致。

photoURI打印结果：content://cn.lovexiaoai.myapp.fileprovider/camera\_photos/Pictures/Screenshots/testImg.png 。

这里的camera\_photos:对应filepaths.xml中的name

\*/

Uri photoURI = FileProvider.getUriForFile(context, "cn.lovexiaoai.myapp.fileprovider", file);

/\* 这句要记得写：这是申请权限，之前因为没有添加这个，打开裁剪页面时，一直提示“无法修改低于50\*50像素的图片”，

开始还以为是图片的问题呢，结果发现是因为没有添加FLAG\_GRANT\_READ\_URI\_PERMISSION。

如果关联了源码，点开FileProvider的getUriForFile()看看（下面有），注释就写着需要添加权限。

\*/

intent.addFlags(Intent.FLAG\_GRANT\_READ\_URI\_PERMISSION);

intent.setDataAndType(photoURI, "image/\*");

# 获取通讯录，发送短信，接收短信

## 获取通讯录

**Intent** intent = **new** Intent(**Intent**.**ACTION\_PICK**, **ContactsContract**.**Contacts**.**CONTENT\_URI**);

startActivityForResult(intent, **CODE\_REQUEST**);

@Override

**protected** **void** **onActivityResult**(**int** requestCode, **int** resultCode, **Intent** data) {

**super**.onActivityResult(requestCode, resultCode, data);

**if** (requestCode == **CODE\_REQUEST**) {

**if** (resultCode == **RESULT\_OK**) {

**Uri** contactUri = data.getData();

Cursor cursor = getContentResolver().query(contactUri, **null**, **null**, **null**, **null**);

cursor.moveToFirst();

**String** contactName = cursor.getString(cursor.getColumnIndex(**ContactsContract**.**Contacts**.**DISPLAY\_NAME**));

**String** number = getContactNum(cursor);

**if** (**TextUtils**.*isEmpty*(number)) {

mContactNames.add(contactName);

mContactNums.add(number);

}

}

}

}

**private** **String** **getContactNum**(Cursor cursor) {

**int** numberCount = cursor.getInt(cursor.getColumnIndex(**ContactsContract**.**Contacts**.**HAS\_PHONE\_NUMBER**));

**String** number = **null**;

**if** (numberCount > 0) {

**int** contactId = cursor.getInt(cursor.getColumnIndex(**ContactsContract**.**Contacts**.**\_ID**));

Cursor phoneCursor = getContentResolver().query(**ContactsContract**.**CommonDataKinds**.**Phone**.**CONTENT\_URI**, **null**, **ContactsContract**.**CommonDataKinds**.**Phone**.**CONTACT\_ID** + "=" + contactId, **null**, **null**);

phoneCursor .moveToFirst();

number = phoneCursor.getString(phoneCursor.getColumnIndex(**ContactsContract**.**CommonDataKinds**.**Phone**.**NUMBER**));

phoneCursor.close();

}

cursor.close();

**return** number;

}

## 发送短信

**----------------------------------------------------**

**public** **int** **sendMsg**(**String** number, **String** msg, **PendingIntent** sendPi, **PendingIntent** deliverPi){

**// 获取系统默认的短信管理器**

**SmsManager** smsManager = **SmsManager**.*getDefault*();

**//按照每条短信最大字数来拆分短信**

**ArrayList**<**String**> contents = smsManager.divideMessage(msg);

/\*\*

\* smsManager.sendTextMessage(destinationAddress, scAddress, text, sentIntent, deliveryIntent)

-- destinationAddress：目标电话号码

-- scAddress：短信中心号码，测试可以不填

-- text: 短信内容

-- sentIntent：发送 -->中国移动 --> 中国移动发送失败 --> 返回发送成功或失败信号 --> 后续处理 即，这个意图包装了短信发送状态的信息

-- deliveryIntent： 发送 -->中国移动 --> 中国移动发送成功 --> 返回对方是否收到这个信息 --> 后续处理 即：这个意图包装了短信是否被对方收到的状态信息（供应商已经发送成功，但是对方没有收到）

\* \*/

**for** (**String** content : contents) {

smsManager.sendTextMessage(number, **null**, content, sendPi, deliverPi);

}

**return** contents.size();

}

**--------------------------------------------------------**

registerReceiver(mSendBroadcastReceiver = **new** BroadcastReceiver(){

@Override

**public** **void** **onReceive**(**Context** context, **Intent** intent) {

mMsgSendCount ++;

**if** (getResultCode() == **RESULT\_OK**) {

**Log**.*e*("TGA", "短信发送成功" + mMsgSendCount + "/" + mTotoalCount);

}**else** {

**Log**.*e*("TGA", "短信发送失败");

}

**Toast**.*makeText*(**SendMsgActivity**.**this**, "短信发送成功" + mMsgSendCount + "/" + mTotoalCount, 0).show();

**if** (mMsgSendCount == mTotoalCount) {

finish();

}

}

}, **new** IntentFilter(**ACTION\_SEND\_MSG**));

registerReceiver(mDeliverBroadcastReceiver = **new** BroadcastReceiver(){

@Override

**public** **void** **onReceive**(**Context** context, **Intent** intent) {

**Log**.*e*("TGA", "联系人已经成功收到我们的短息");

}

}, **new** IntentFilter(**ACTION\_DELIVER\_MSG**));

**Intent** sendIntent = **new** Intent(**ACTION\_SEND\_MSG**);

mSendPi = **PendingIntent**.*getBroadcast*(**this**, 0, sendIntent, 0);

**Intent** delIntent = **new** Intent(**ACTION\_DELIVER\_MSG**);

mDeliverPi = **PendingIntent**.*getBroadcast*(**this**, 1, delIntent, 0);

mTotoalCount = mSmsBiz.sendMsg(mContactNums, msg, mSendPi, mDeliverPi);

mMsgSendCount = 0;

**--------------------------------------------------------------------**

## 接收短信

**class** **MsgReceiver** **extends** **BroadcastReceiver**{

@Override

**public** **void** **onReceive**(**Context** context, **Intent** intent) {

**Object**[] pdus = (**Object**[]) intent.getExtras().get("pdus");

**SmsMessage**[] messages = **new** **SmsMessage**[pdus.length];

**for** (**int** i = 0; i < messages.length; i++) {

messages[i] = **SmsMessage**.*createFromPdu*((**byte**[]) pdus[i]);

}

**String** address = messages[0].getOriginatingAddress();//获取号码

**String** fullMessage = "";

**for** (**SmsMessage** smsMessage : messages) {

fullMessage += smsMessage.getMessageBody();//获取内容

}

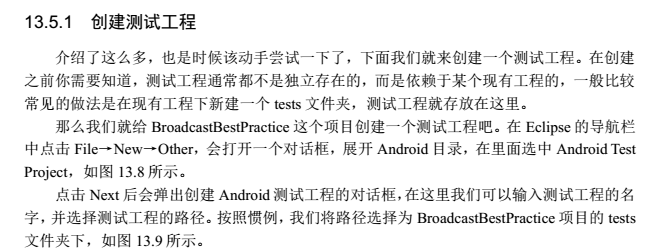
tv.setText(address);

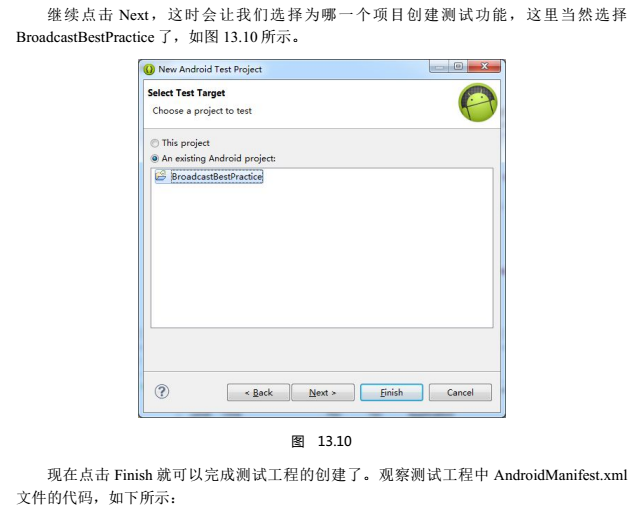
edt.setText(fullMessage);

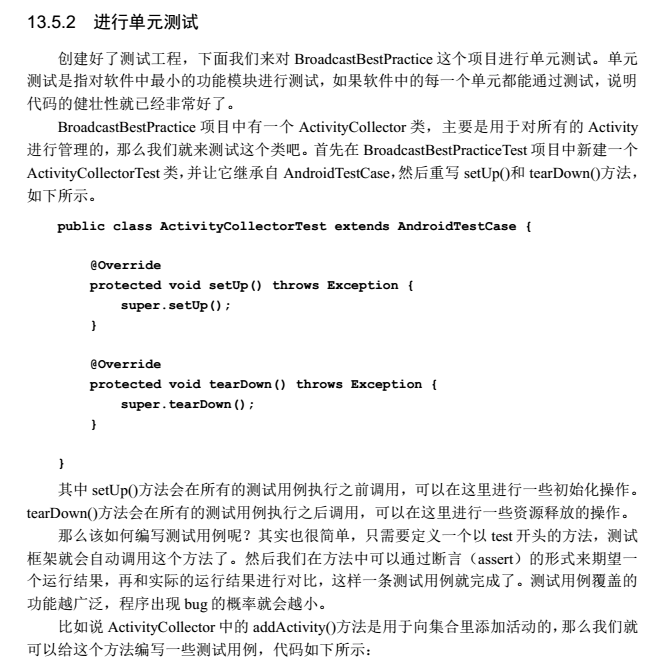
}

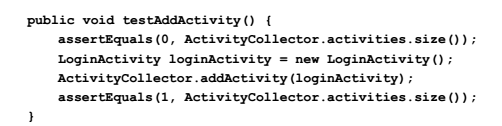
}

# 测试工程和单元测试

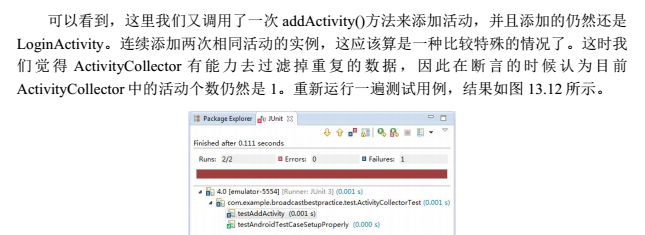












# 注释规范

一般情况下，源程序有效注释亮须在20%以上，注释不宜过多，也不能太少，注释须精准易懂，言简意赅。

## 文件注释

从Java文件第一行开始，为避免被JavaDoc收集，以 /\* 开始，以 \*/ 结束（JavaDoc收集的注释内容以 /\*\* 开始），中间每一行前面加一个“\*”。一般是一些版权信息、描述信息及License。

/\*

\* Copyright (c) 2016, smuyyh@gmail.com All Rights Reserved.

\*/

## 类注释

以 /\*\* 开头， 在Class、Interface、Enum之前。一般是用一句话描述类用途，也可有作者及时间等信息，这类信息不宜过多。

## 属性注释

以 /\*\* 开头。有时也可以直接写在一行

/\*\*

\* 注释信息

\*/

private static final int TEXT\_COLOR = Color.RED;

/\*\* 注释信息 \*/

private static final int BG\_COLOR = Color.RED;

## 方法注释

@since表明从那个版本开始就有这个方法；@exception或throws可能的异常；@deprecated表示不建议使用该方法。异常注释用@exception或@throws表示，在JavaDoc中二者等价，但推荐用@exception标注Runtime异常，@throws标注非Runtime异常。异常的注释必须说明该异常的含义及什么条件下抛出该异常。

/\*\*

\* 方法描述

\* @param [参数1] [参数1说明]

\* @param [参数2] [参数2说明]

\* @return [返回类型说明]

\* @exception/throws [违例类型] [违例说明]

\* @see [类、类#方法、类#成员]

\* @deprecated [说明及原因]

\*/

public int method(String arg1, String agr2) throws Exception{

}

* 对变量的定义和分支语句（条件分支、循环语句等），稍微复杂一点的最好编写注释，因为这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。
* 对于switch语句下的case语句，如果因为特殊情况需要处理完一个case后需进入下一个case处理，最好在该case语句处理完、下一个case语句前加上明确的注释，这样比较清楚程序编写者的意图，有效防止无辜遗漏break语句
* 边写代码边注释，修改代码同时修改相应注释，以保证注释与代码的一致性。更新代码须和更新注释保持同步。
* 注释的内容要清晰、明了，含义明确，防止注释二义性，错误的注释不但无益反而有害。
* 尽量避免在注释中使用缩写，特别是不常用缩写。

# 网络信息、网络状态获取

/\*\*

\* 获取网络信息

\*/

**private** **int** **getNetInfo**() {

//返回值 -1：没有网络 1：WIFI网络2：wap网络3：net网络

**int** type = -1;

**ConnectivityManager** manager = (**ConnectivityManager**) context.getSystemService(**Context**.**CONNECTIVITY\_SERVICE**);

**if** (manager != **null**) {

**NetworkInfo** info = manager.getActiveNetworkInfo();

**if** (info != **null &&** info.isAvailable()) {

**int** nType = info.getType();

**if** (nType == **ConnectivityManager**.**TYPE\_MOBILE**){

**if**(info.getExtraInfo().equalsIgnoreCase("cmnet")) {

type = 3;

}

**else** {

type = 2;

}

} **else** **if** (nType == **ConnectivityManager**.**TYPE\_WIFI**) {

type = 1;

}

}

**return** type;

}

}

/\*\*

\* 检测当的网络（WLAN、3G/2G）状态

\*/

**private boolean** **isNetworkAvailable**(**Context** context) {

**ConnectivityManager** manager = (**ConnectivityManager**) context.getSystemService(**Context**.**CONNECTIVITY\_SERVICE**);

**if** (manager != **null**) {

**NetworkInfo** info = manager.getActiveNetworkInfo();

**if** (info != **null** && info.isAvailable()) {

**if** (info != **null** && info.isConnected()) {

// 当前网络是连接的

**if** (info.getState() == **NetworkInfo**.State.**CONNECTED**) {

// 当前所连接的网络可用

**return** **true**;

}

}

}

}

**return** **false**;

}

# 反射机制

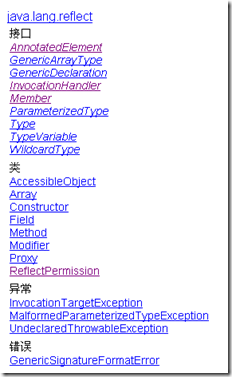
## 什么是JAVA的反射机制

Java反射是Java被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过Reflection APIs取得任何一个已知名称的class的内部信息，包括其modifiers（诸如public, static 等）、superclass（例如Object）、实现之interfaces（例如Cloneable），也包括fields和methods的所有信息，并可于运行时改变fields内容或唤起methods。Java反射机制容许程序在运行时加载、探知、使用编译期间完全未知的classes。换言之，Java可以加载一个运行时才得知名称的class，获得其完整结构。

## JDK中提供的Reflection API

Java反射相关的API在包[java.lang.reflect](http://blog.csdn.net/yongjian1092/article/details/7364451)中

|  |  |
| --- | --- |
| Member接口 | 该接口可以获取有关类成员（域或者方法）或者构造函数的信息。 |
| AccessibleObject类 | 该类是域(field)对象、方法(method)对象、构造函数(constructor)对象的基础类。它提供了将反射的对象标记为在使用时取消默认 Java 语言访问控制检查的能力。 |
| Array类 | 该类提供动态地生成和访问JAVA数组的方法。 |
| Constructor类 | 提供一个类的构造函数的信息以及访问类的构造函数的接口。 |
| Field类 | 提供一个类的域的信息以及访问类的域的接口。 |
| Method类 | 提供一个类的方法的信息以及访问类的方法的接口。 |
| Modifier类 | 提供了 static 方法和常量，对类和成员访问修饰符进行解码。 |
| Proxy类 | 提供动态地生成代理类和类实例的静态方法。 |



## JAVA反射机制提供了什么功能

1、在运行时判断任意一个对象所属的类

2、在运行时构造任意一个类的对象

3、在运行时判段任意一个类所具有的成员变量和方法

4、在运行时调用任一个对象的方法

5、在运行时创建新类对象

### 首先定义用于测试的类:

**class** **Type**{

**public** **int** pubIntField;

**public** **String** pubStringField;

**private** **int** prvIntField;

**public** **Type**(){

Log("Default Constructor");

}

**Type**(**int** arg1, **String** arg2){

pubIntField = arg1;

pubStringField = arg2;

Log("Constructor with parameters");

}

**public** **void** **setIntField**(**int** val) {

**this**.prvIntField = val;

}

**public** **int** **getIntField**() {

**return** prvIntField;

}

**private** **void** **Log**(**String** msg){

**System**.**out**.println("Type:" + msg);

}

}

**class** **ExtendType** **extends** **Type**{

**public** **int** pubIntExtendField;

**public** **String** pubStringExtendField;

**private** **int** prvIntExtendField;

**public** **ExtendType**(){

Log("Default Constructor");

}

**ExtendType**(**int** arg1, **String** arg2){

pubIntExtendField = arg1;

pubStringExtendField = arg2;

Log("Constructor with parameters");

}

**public** **void** **setIntExtendField**(**int** field7) {

**this**.prvIntExtendField = field7;

}

**public** **int** **getIntExtendField**() {

**return** prvIntExtendField;

}

**private** **void** **Log**(**String** msg){

System.**out**.println("ExtendType:" + msg);

}

}

### 获取类的Class对象

**在使用Java的反射功能时，基本首先都要获取类的Class对象，再通过Class对象获取其他的对象。**

在平时的使用，要注意对这几种方法的灵活运用，尤其是对Class.forName()方法的使用。因为在很多开发中，会直接通过类的名称取得Class类的对象。

调用getClass

**Boolean** var1 = **true**;

**Class**<?> classType2 = var1.getClass();

**System**.**out**.println(classType2);//输出：class java.lang.Boolean

运用.class 语法

**Class**<?> classType4 = **Boolean**.**class**;

**System**.**out**.println(classType4);//输出：class java.lang.Boolean

运用static method Class.forName()

**Class**<?> classType5 = **Class**.*forName*("java.lang.Boolean");

**System**.**out**.println(classType5);//输出：class java.lang.Boolean

运用基本类型包装类的TYPE 语法，这里返回的是原生类型，和Boolean.class返回的不同

**Class**<?> classType3 = **Boolean**.**TYPE**;

**System**.**out**.println(classType3);//输出：boolean

### 一些方法

Class、Method、Constructor、Field都有一个public方法int getModifiers() 和 String getName();

**int** mod = constructors[i].getModifiers(); // 获取修饰符对应 数字

**String** modifier = **Modifier**.*toString*(mod);//获取修饰符 public，private，stati， native，final...

**String** className = classType.getName();//获取指定类的类名 com.cgy.reflect.ExtendType

Method、Constructor都有一个public方法Class[] getParameterTypes()

**Class**[] parameterTypes = constructors[i].getParameterTypes(); // 获取指定构造方法的参数的类型集合

**String** returnType = methods[i].getReturnType().getName();//方法的返回类型 void； boolean

**Class** cl = fb[i].getType(); // 属性的Class类

**Field** f = classType.getField(fb[i].getName()); // 这里返回的东西相当于就是fb[i]

**Object** value = (**Object**)f.get(r); //获取值，如果是private的还要加上setAccessible(true);

### 获取类的Constructor

通过反射机制得到某个类的构造器，然后调用该构造器创建该类的一个实例

Class<T>类提供了几个方法获取类的构造器。

public Constructor<T> getConstructor(Class<?>... parameterTypes)返回一个 Constructor 对象，它反映此 Class 对象所表示的类的指定公共构造方法

public Constructor<?>[] getConstructors()返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共构造方法

public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)返回一个 Constructor 对象，该对象反映此 Class 对象所表示的类或接口的指定构造方法

public Constructor<?>[] getDeclaredConstructors()返回 Constructor 对象的一个数组，这些对象反映此 Class 对象表示的类声明的所有构造方法。它们是公共、保护、默认（包）访问和私有构造方法

**Constructor**<?>[] constructors = classType.getConstructors();// 返回所有具有public属性的构造函数数组

**for** (**Constructor**<?> m : constructors) {

**System**.**out**.println(m);

}

**System**.**out**.println("------");

constructors = classType.getDeclaredConstructors();// 返回该类中所有的构造函数数组（不分public和非public属性）

**for** (**Constructor**<?> m : constructors) {

**System**.**out**.println(m);

}

输出：

public com.quincy.ExtendType()

-------

public com.quincy.ExtendType()

com.quincy.ExtendType(int,java.lang.String)

### 获取类的Method

通过反射机制得到某个类的某个方法，然后调用对应于这个类的某个实例的该方法

Class<T>类提供了几个方法获取类的方法。

public Method getMethod(String name,Class<?>... parameterTypes) 返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法

public Method[] getMethods()返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 member 方法

public Method getDeclaredMethod(Stringname,Class<?>... parameterTypes)返回一个 Method 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明方法

public Method[] getDeclaredMethods() 返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法

**Method**[] methods = classType.getMethods();// 使用getMethods获取函数

**for** (**Method** m : methods) {//public（pri不包含）的方法，包括父类

**System**.**out**.println(m);

}

**System**.**out**.println("------");

methods = classType.getDeclaredMethods();// 使用getDeclaredMethods获取函数

**for** (**Method** m : methods) {//所有（pri也包含）的方法，不包括父类

**System**.**out**.println(m);

}

输出：

public void com.quincy.ExtendType.setIntExtendField(int)

public int com.quincy.ExtendType.getIntExtendField()

public void com.quincy.Type.setIntField(int)

public int com.quincy.Type.getIntField()

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object)

public java.lang.String java.lang.Object.toString()

public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass()

public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll()

-------

private void com.quincy.ExtendType.Log(java.lang.String)

public void com.quincy.ExtendType.setIntExtendField(int)

public int com.quincy.ExtendType.getIntExtendField()

### 获取类的Fields成员变量（成员属性）

可以通过反射机制得到某个类的某个属性，然后改变对应于这个类的某个实例的该属性值。JAVA 的Class<T>类提供了几个方法获取类的属性。

public Field getField(String name) 返回一个 Field 对象，它反映此 Class 对象所表示的类或接口的指定公共成员字段

public Field[] getFields() 返回一个包含某些 Field 对象的数组，这些对象反映此 Class 对象所表示的类或接口的所有可访问公共字段

public FieldgetDeclaredField(Stringname) 返回一个 Field 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明字段

public Field[] getDeclaredFields() 返回 Field 对象的一个数组，这些对象反映此 Class 对象所表示的类或接口所声明的所有字段

**Class**<?> classType = **ExtendType**.**class**;

**Field**[] fields = classType.getFields(); // 使用getFields获取属性

**for** (**Field** f : fields) {//public（pri不包含）的属性，包括父类

**System**.**out**.println(f);

}

**System**.**out**.println("------");

fields = classType.getDeclaredFields();// 使用getDeclaredFields获取属性

**for** (**Field** f : fields) {//所有（pri也包含）的属性，不包括父类

**System**.**out**.println(f);

}

输出：

public int com.quincy.ExtendType.pubIntExtendField

public java.lang.String com.quincy.ExtendType.pubStringExtendField

public int com.quincy.Type.pubIntField

public java.lang.String com.quincy.Type.pubStringField

--------

public int com.quincy.ExtendType.pubIntExtendField

public java.lang.String com.quincy.ExtendType.pubStringExtendField

private int com.quincy.ExtendType.prvIntExtendField

### 新建类的实例

通过反射机制创建新类的实例，有几种方法可以创建

// 1、调用类的Class对象的newInstance方法，该方法会调用对象的默认构造器，如果没有默认构造器，会调用失败.

**Object** inst = classType.newInstance();

**System**.**out**.println(inst);

/\*输出：

Type:Default Constructor

ExtendType:Default Constructor

com.quincy.ExtendType@d80be3\*/

// 2、调用默认Constructor对象的newInstance方法

**Constructor**<?> constructor1 = classType.getConstructor();

**Object** inst2 = constructor1.newInstance();

**System**.**out**.println(inst2);

/\*输出：

Type:Default Constructor

ExtendType:Default Constructor

com.quincy.ExtendType@1006d75\*/

// 3、调用带参数Constructor对象的newInstance方法

**Constructor**<?> constructor2 = classType.getDeclaredConstructor(**int**.**class**, **String**.**class**);

**Object** inst2 = constructor2.newInstance(1, "123");

**System**.**out**.println(inst2);

/\*输出：

Type:Default Constructor

ExtendType:Constructor with parameters

com.quincy.ExtendType@15e83f9\*/

### 调用类的函数

通过反射获取类Method对象，调用Method的Invoke方法调用函数。

**Object** inst3 = classType.newInstance();

**Method** logMethod = classType.getDeclaredMethod("Log", **String**.**class**);

logMethod.invoke(inst3, "test");

/\*输出：

Type:Default Constructor

ExtendType:Default Constructor

Class com.quincy.ClassT can not access a member of class com.quincy.ExtendType with modifiers "private"

上面失败是由于没有权限调用private函数，这里需要设置Accessible为true;\*/

**Object** inst4 = classType.newInstance();

**Method** logMethod2 = classType.getDeclaredMethod("Log", **String**.**class**);

logMethod2.setAccessible(**true**);

logMethod2.invoke(inst4, "test");

/\*输出：

Type:Default Constructor

ExtendType:Default Constructor

ExtendType:test

private void com.cgy.reflect.ExtendType.Log(java.lang.String)\*/

### 设置/获取类的属性值

通过反射获取类的Field对象，调用Field方法设置或获取值

**Object** inst5 = classType.newInstance();

**Field** intField = classType.getField("pubIntExtendField");

intField.setInt(inst5, 100); //设置

**int** value = intField.getInt(inst5); //取出 **Object** value = (**Object**)intField.get(inst5 );

**System**.**out**.println(value);

/\*输出：

Type:Default Constructor

ExtendType:Default Constructor

100 \*/

## 动态创建代理类

**代理模式**

代理模式的作用=为其他对象提供一种代理以控制对这个对象的访问。

**代理模式的角色：**

抽象角色：声明真实对象和代理对象的共同接口

代理角色：代理角色内部包含有真实对象的引用，从而可以操作真实对象。

真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

**动态代理：**

java.lang.reflect.Proxy

Proxy 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类

InvocationHandler

是代理实例的调用处理程序 实现的接口，每个代理实例都具有一个关联的调用处理程序。对代理实例调用方法时，将对方法调用进行编码并将其指派到它的调用处理程序的 invoke 方法。

**动态Proxy是这样的一种类:**

它是在运行生成的类，在生成时你必须提供一组Interface给它，然后该class就宣称它实现了这些interface。你可以把该class的实例当作这些interface中的任何一个来用。当然，这个Dynamic Proxy其实就是一个Proxy，它不会替你作实质性的工作，在生成它的实例时你必须提供一个handler，由它接管实际的工作。在使用动态代理类时，我们必须实现InvocationHandler接口

**步骤：**

1. 定义抽象角色

**public** **interface** Subject {

**public** **void** **Request**();

}

1. 定义真实角色

**public** **class** **RealSubject** **implements** Subject {

@Override

**public** **void** **Request**() {

**System**.**out**.println("RealSubject");

}

}

1. 定义代理角色

**public** **class** **DynamicSubject** **implements** InvocationHandler {

**private** **Object** sub;

**public** **DynamicSubject**(**Object** obj){

**this**.sub = obj;

}

@Override

**public** **Object** **invoke**(**Object** proxy, **Method** method, **Object**[] args)**throws** Throwable {

**System**.**out**.println("Method:"+ method + ",Args:" + args);

method.invoke(sub, args);

**return** **null**;

}

}

1. 通过Proxy.newProxyInstance构建代理对象

**RealSubject** realSub = **new** RealSubject();

InvocationHandler handler = **new** DynamicSubject(realSub);

**Class**<?> classType = handler.getClass();

Subject sub = (Subject)**Proxy**.*newProxyInstance*(classType.getClassLoader(),

realSub.getClass().getInterfaces(), handler);

**System**.**out**.println(sub.getClass());

1. 通过调用代理对象的方法去调用真实角色的方法。

sub.Request();

/\*输出：

class $Proxy0 新建的代理对象，它实现指定的接口

Method:public abstract void DynamicProxy.Subject.Request(),Args:null

RealSubject 调用的真实对象的方法\*/

## 调用类中的private方法举例

### 工具类

**public** **class** **LoadMethod** {

/\*\*

\* 在运行时加载指定的类，并调用指定的方法

\* **@param** cName Java的类名

\* **@param** MethodName 方法名

\* **@param** types 方法的参数类型

\* **@param** params 方法的参数值

\* **@return**

\*/

**public** **Object** **Load**(**String** cName, **String** MethodName, **String**[] types, **String**[] params) {

**Object** retObject = **null**;

**try** {

// 加载指定的类

**Class** cls = **Class**.*forName*(cName); // 获取Class类的对象的方法之二

// 利用newInstance()方法，获取构造方法的实例

// Class的newInstance方法只提供默认无参构造实例

// Constructor的newInstance方法提供带参的构造实例

**Constructor** ct = cls.getConstructor();

**Object** obj = ct.newInstance();

// 构建 方法的参数类型（自定义方法中）

**Class** paramTypes[] = **this**.getMethodTypesClass(types);

**//-----------------------------------二选一**

// 在指定类中获取指定的方法（public的方法）

**Method** meth = cls.getMethod(MethodName, paramTypes);

// 在指定类中获取指定的方法（所有的方法）

**Method** meth = cls.getDeclaredMethod(MethodName, paramTypes);

//private权限权限(有安全问题)

meth.setAccessible(**true**);

**//-----------------------------------二选一**

// 构建 方法的参数值（自定义方法中）

**Object** argList[] = **this**.getMethodParamObject(types, params);

// 调用指定的方法并获取返回值为Object类型

retObject = meth.invoke(obj, argList);

} **catch** (**Exception** e) {

**System**.**err**.println(e);

}

**return** retObject;

}

/\*\*

\* 获取参数类型，返回值保存在Class[]中

\*/

**public** **Class**[] **getMethodTypesClass**(**String**[] types) {

**Class**[] cs = **new** **Class**[types.length];

**for** (**int** i = 0; i < cs.length; i++) {

**if** (types[i] != **null** || !types[i].trim().equals("")) {

**if** (types[i].equals("int") || types[i].equals("Integer")) {

cs[i] = **Integer**.**TYPE**;

}

**else** **if** (types[i].equals("float") || types[i].equals("Float")) {

cs[i] = **Float**.**TYPE**;

}

**else** **if** (types[i].equals("double") || types[i].equals("Double")) {

cs[i] = **Double**.**TYPE**;

}

**else** **if** (types[i].equals("boolean") || types[i].equals("Boolean")) {

cs[i] = **Boolean**.**TYPE**;

}

**else** {

cs[i] = **String**.**class**;

}

}

}

**return** cs;

}

/\*\*

\* 获取参，将String全转换为需要的格式，保存在数Object[]中

\*/

**public** **Object**[] **getMethodParamObject**(**String**[] types, **String**[] params) {

**Object**[] retObjects = **new** **Object**[params.length];

**for** (**int** i = 0; i < retObjects.length; i++) {

**if**(!params[i].trim().equals("")||params[i]!=**null**){

**if**(types[i].equals("int")||types[i].equals("Integer")){

retObjects[i]= **new** Integer(params[i]);

}

**else** **if**(types[i].equals("float")||types[i].equals("Float")){

retObjects[i]= **new** Float(params[i]);

}

**else** **if**(types[i].equals("double")||types[i].equals("Double")){

retObjects[i]= **new** Double(params[i]);

}

**else** **if**(types[i].equals("boolean")||types[i].equals("Boolean")){

retObjects[i]=**new** Boolean(params[i]);

}

**else**{

retObjects[i] = params[i];

}

}

}

**return** retObjects;

}

### 对LoadMethod.java的优化处理

/\*\*在上一节中给出的LoadMethod.java中，类LoadMethod对固定参数类型的方法进行了调用，并且参数类型是通过一个String[]数组传入，然后经过方法 getMethodTypesClass() 解析之后，才得到了参数的具体的类型。同时在getMethodTypesClass()和getMethodParamObject()方法中，通过对传入的字符串参数进行过滤后，再处理那些可以匹配中的参数类型，其他不能匹配的参数都做为String对象来处理。如果我们调用的方法所需要的参数不是简单类型的变量，而是自定义的类对象，或者List列表，再如果我们只知道类名和方法名，不知道方法的参数类型，那我们该如何处理这些情况呢？\*/

/\*\*

\* 在运行时加载指定的类，并调用指定的方法

\* **@param** cName Java的类名

\* **@param** MethodName 方法名

\* **@param** params 方法的参数值

\* **@param** postion 方法的参数类型位置

\* **@return**

\*/

**public** **Object** **Load2**(**String** cName, **String** MethodName, **Object**[] params, **int** postion) {

**Object** retObject = **null**;

**try** {

**Class** cls = **Class**.*forName*(cName);

**Constructor** ct = cls.getConstructor(**null**);

**Object** obj = ct.newInstance(**null**);

// 根据方法名获取指定方法的参数类型列表

List<**Class**[]> list = **this**.getParamTypes(cls, MethodName);

//自己看list里边的参数，选取自己要的

**Class** paramTypes[] = list.get(postion);

**Method** meth = cls.getMethod(MethodName, paramTypes);

meth.setAccessible(**true**);

retObject = meth.invoke(obj, params);

} **catch** (**Exception** e) {

**System**.**err**.println(e);

}

**return** retObject;

}

/\*\*

\* 获取参数类型，返回值保存在Class[]中

\*/

**public** List<**Class**[]> **getParamTypes**(**Class** cls, **String** mName) {

List<**Class**[]> list = **new** ArrayList<>();

**Class**[] cs = **null**;

//由于我们一般通过反射机制调用的方法，是非public方法,所以在此处使用了getDeclaredMethods()方法

**Method**[] mtd = cls.getDeclaredMethods();

**for** (**int** i = 0; i < mtd.length; i++) {

**if** (!mtd[i].getName().equals(mName)) { // 不是我们需要的参数，则进入下一次循环

**continue**;

}

cs = mtd[i].getParameterTypes();

list.add(cs);

}

**return** list;

}

}

### ExtendType里的private方法

**private** **Type** **getType**(**int** a, **String** b){

**return** **new** Type(a,b);

}

### 调用

**LoadMethod** loadMethod = **new** LoadMethod();

**Type** type = (**Type**) loadMethod.Load("com.cgy.reflect.ExtendType", "getType", **new** **String**[]{"int", "String"}, **new** **String**[]{"3", "你好"});

## Dialog消失不消失反射示例

**public** **class** **MainActivity** **extends** **Activity** {

**private** **static** **Button** ***btnHandler*** = **null**;

**private** **static** **Button** ***btnShowing*** = **null**;

**AlertDialog** alertDialog = **null**;

@Override

**public** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

***btnHandler*** = (**Button**)findViewById(**R**.**id**.**btn\_mHandler**);

***btnHandler***.setOnClickListener(**new** ButtonListener());

***btnShowing*** = (**Button**)findViewById(**R**.**id**.**btn\_mShowing**);

***btnShowing***.setOnClickListener(**new** ButtonListener());

alertDialog = **new** **AlertDialog**.Builder(**this**)

.setTitle("abc").setMessage("Content").setIcon(**R**.**drawable**.**ic\_launcher**)

.setPositiveButton("确定", **new** PositiveClickListener())

.setNegativeButton("取消", **new** NegativeClickListener()).create();

}

**private** **class** **ButtonListener** **implements** OnClickListener {

@Override

**public** **void** **onClick**(**View** v) {

**switch** (v.getId()) {

**case** **R**.**id**.**btn\_mHandler**:

modify\_mHandler();//修改AlertController类的private成员变量mHandler的值

//效果：不主动调dismiss等取消dialog的操作，不会消失（原生的只要点击就会消失）

alertDialog.show();

**break**;

**case** **R**.**id**.**btn\_mShowing**:

alertDialog.show();

**break**;

**default**:

**break**;

}

}

}

**private** **class** **PositiveClickListener** **implements** android.content.DialogInterface.OnClickListener {

@Override

**public** **void** **onClick**(DialogInterface dialog, **int** which) {

modify\_dismissDialog(**false**);//修改dismissDialog()方法,

//效果：如果值为false，表示对话框已经关闭，再dismiss无用，无法消失了

}

}

**private** **class** **NegativeClickListener** **implements** android.content.DialogInterface.OnClickListener {

@Override

**public** **void** **onClick**(DialogInterface dialog, **int** which) {

modify\_dismissDialog(**true**);//如果值为false，对话框已经关闭，再dismiss无用

//alertDialog.dismiss();

}

}

/\*

\* 修改AlertController类的private成员变量mHandler的值

\*/

**public** **void** **modify\_mHandler**() {

**try** {

//AlertDialog 类中有个全局变量 private AlertController mAlert;

**Field** field = alertDialog.getClass().getDeclaredField("mAlert");//获取AlertDialog类中的全局变量mAlert

field.setAccessible(**true**);

**Object** obj = field.get(alertDialog);// 获取alertDialog实例中mAlert变量的值

//AlertController 类中有个全局变量 private Handler mHandler;

field = obj.getClass().getDeclaredField("mHandler");//获取AlertController类中的全局变量mHandler

field.setAccessible(**true**);

field.set(obj, **new** MyButtonHandler(alertDialog));// 修改obj实例中mHandler变量的值，使用新的ButtonHandler类

} **catch** (**Exception** e) {

e.printStackTrace();

}

}

/\*

\* 修改dismissDialog()方法

\*/

**public** **void** **modify\_dismissDialog**(**boolean** flag) {

**try** {

**Field** field = alertDialog.getClass().getSuperclass().getDeclaredField("mShowing");

field.setAccessible(**true**);

field.set(alertDialog, flag);// 将mShowing变量设为false，表示对话框已经关闭

alertDialog.dismiss();

} **catch** (**Exception** e) {

e.printStackTrace();

}

}

**public** **class** **MyButtonHandler** **extends** **Handler**{

**private** **static** **final** **int** **MSG\_DISMISS\_DIALOG** = 1;

**private** **WeakReference**<DialogInterface> mDialog;

**public** **MyButtonHandler**(DialogInterface dialog) {

mDialog = **new** WeakReference<DialogInterface>(dialog);

}

@Override

**public** **void** **handleMessage**(**Message** msg) {

**switch** (msg.what) {

**case** DialogInterface.**BUTTON\_POSITIVE**:

**case** DialogInterface.**BUTTON\_NEGATIVE**:

**case** DialogInterface.**BUTTON\_NEUTRAL**:

((DialogInterface.OnClickListener) msg.obj).onClick(mDialog.get(), msg.what);

**break**;

}

}

}

}

## 获取图片最大宽度示例

**// 检查最大值,由于方法只能在API16使用，这里使用反射获取**

width = *getImageViewFieldValue*(imageView, "mMaxWidth");

/\*\* 通过反射获取Object的某个属性值 \*/

**private** **static** **int** **getImageViewFieldValue**(**Object** object, **String** fieldName) {

**int** value = 0;

**try** {

**Field** field = **ImageView**.**class**.getDeclaredField(fieldName);**//全局变量类**

field.setAccessible(**true**);

**int** fieldValue = (**Integer**) field.get(object);**//object实例中的fieldName全局变量**

**if** (fieldValue > 0 && fieldValue < **Integer**.**MAX\_VALUE**) {

value = fieldValue;

**Log**.*e*("TAG", value + "");

}

} **catch** (**Exception** e) {

}

**return** value;

}

# 注解Annotation

## 示例

**Override Annotation**

@Override

public void onCreate(Bundle savedInstanceState);

**Retrofit Annotation**

Retrofit 为符合 RESTful 规范的网络请求框架

@GET("/users/{username}")

User getUser(@Path("username") String username);

**Butter Knife Annotation**

Butter Knife 为 View 及事件等依赖注入框架

@InjectView(R.id.user) EditText username;

**ActiveAndroid Annotation**

Active Android 为 ORM 框架

@Column(name = “Name") public String name;

## 概念及作用

**1 概念**

能够添加到 Java 源代码的语法元数据。类、方法、变量、参数、包都可以被注解，可用来将信息元数据与程序元素进行关联。Annotation 中文常译为“注解”

**2 作用**

a. 标记，用于告诉编译器一些信息

b. 编译时动态处理，如动态生成代码

c. 运行时动态处理，如得到注解信息

这里的三个作用实际对应着后面自定义 Annotation 时说的 @Retention 三种值分别表示的 Annotation

## 分类

**1 标准 Annotation**

包括 Override, Deprecated, SuppressWarnings，标准 Annotation 是指 Java 自带的几个 Annotation，上面三个分别表示重写函数，不鼓励使用(有更好方式、使用有风险或已不在维护)，忽略某项 Warning

**2 元 Annotation**

@Retention, @Target, @Inherited, @Documented，元 Annotation 是指用来定义 Annotation 的 Annotation，在后面 Annotation 自定义部分会详细介绍含义

**3 自定义 Annotation**

自定义 Annotation 表示自己根据需要定义的 Annotation，定义时需要用到上面的元 Annotation，这里只是一种分类而已，也可以根据作用域分为源码时、编译时、运行时 Annotation，后面在自定义 Annotation 时会具体介绍

## 自定义

**1 调用**

**public** **class** **App** {

@MethodInfo(author = "chen",

date = "2016/7/1",

version = 3)

**public** **String** **getAppName**(){

**return** "cgy";

}

}

这里是调用自定义 Annotation——MethodInfo 的示例，MethodInfo Annotation 作用为给方法添加相关信息，包括 author、date、version

**2 定义**

@Documented

@Retention(RetentionPolicy.**RUNTIME**)

@Target(ElementType.**METHOD**)

@Inherited

**public** **@interface** MethodInfo {

**String** **author**() **default** "trinea@gmail.com";

**String** **date**();

**int** **version**() **default** 1;

}

这里是 MethodInfo 的实现部分

(1). 通过 @interface 定义，注解名即为自定义注解名

(2). 注解配置参数名为注解类的方法名，且：

a. 所有方法没有方法体，没有参数没有修饰符，实际只允许 public & abstract 修饰符，默认为 public ，不允许抛异常

b. 方法返回值只能是基本类型，String, Class, annotation, enumeration 或者是他们的一维数组

c. 若只有一个默认属性，可直接用 value() 函数。一个属性都没有表示该 Annotation 为 Mark Annotation

1. . 可以加 default 表示默认值

**3 元 Annotation**

(1). @Documented 是否会保存到 Javadoc 文档中

(2). @Retention 保留时间，可选值 SOURCE（源码时），CLASS（编译时），RUNTIME（运行时），默认为 CLASS，值为 SOURCE 大都为 Mark Annotation，这类 Annotation 大都用来校验，比如 Override, Deprecated, SuppressWarnings

(3). @Target 可以用来修饰哪些程序元素，如 TYPE, METHOD, CONSTRUCTOR, FIELD, PARAMETER 等，未标注则表示可修饰所有

(4). @Inherited 是否可以被继承，默认为 false

## 解析

**1 运行时 Annotation 解析**

(1) 运行时 Annotation 指 @Retention 为 RUNTIME 的 Annotation，可手动调用下面常用 API 解析

method.getAnnotation(AnnotationName.class);

getAnnotation(AnnotationName.class) 表示得到该 Target 某个 Annotation 的信息，因为一个 Target 可以被多个 Annotation 修饰

method.getAnnotations();

getAnnotations() 则表示得到该 Target 所有 Annotation

method.isAnnotationPresent(AnnotationName.class);

isAnnotationPresent(AnnotationName.class) 表示该 Target 是否被某个 Annotation 修饰

以之前自定义的 MethodInfo 为例，利用 Target（这里是 Method）getAnnotation 函数得到 Annotation 信息，然后就可以调用 Annotation 的方法得到响应属性值。其他 @Target 如 Field，Class 方法类似

(2) 解析示例如下：

**private** **void** **init**() **throws** Exception{

**Class**<?> class1 = **Class**.*forName*("com.cgy.annotationdmeo.App");

**Method**[] methods = class1.getDeclaredMethods();

**for** (**Method** method : methods) {

MethodInfo info = method.getAnnotation(MethodInfo.**class**);

**if** (info != **null**) {

**System**.**out**.println("method name:" + method.getName());

**System**.**out**.println("method author:" + info.author());

**System**.**out**.println("method version:" + info.version());

**System**.**out**.println("method date:" + info.date());

}

}

}

**2 编译时 Annotation 解析**

(1) 编译时 Annotation 指 @Retention 为 CLASS 的 Annotation，由 apt(Annotation Processing Tool) 解析自动解析。需要做的

a. 自定义类继承自 AbstractProcessor

b. 重写其中的 process 函数

这块很多同学不理解，实际是 apt(Annotation Processing Tool) 在编译时自动查找所有继承自 AbstractProcessor 的类，然后调用他们的 process 方法去处理

(2) 假设之前自定义的 MethodInfo 的 @Retention 为 CLASS，解析示例如下：

@SupportedAnnotationTypes({ "com.cgy.annotationdemo.MethodClInfo" })

public class MethodInfoProcessor extends AbstractProcessor {

@Override

public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment env) {

HashMap<String, String> map = new HashMap<String, String>();

for (TypeElement te : annotations) {

for (Element element : env.getElementsAnnotatedWith(te)) {

MethodInfo methodInfo = element.getAnnotation(MethodInfo.class);

map.put(element.getEnclosingElement().toString(), methodInfo.author());

}

}

return false;

}

}

SupportedAnnotationTypes 表示这个 Processor 要处理的 Annotation 名字。

process 函数中参数 annotations 表示待处理的 Annotations，参数 env 表示当前或是之前的运行环境

process 函数返回值表示这组 annotations 是否被这个 Processor 接受，如果接受后续子的 rocessor 不会再对这个 Annotations 进行处理

## 几个 Android 开源库 Annotation 原理简析

**Annotation — Butter Knife**

1. 调用

@InjectView(R.id.user)

EditText username;

1. 定义

@Retention(CLASS)

@Target(FIELD)

public @interface InjectView {

int value();

}

1. 原理

@Override

public boolean process(Set<? extends TypeElement> elements, RoundEnvironment env) {

Map<TypeElement, ViewInjector> targetClassMap = findAndParseTargets(env);

for (Map.Entry<TypeElement, ViewInjector> entry : targetClassMap.entrySet()) {

TypeElement typeElement = entry.getKey();

ViewInjector viewInjector = entry.getValue();

try {

JavaFileObject jfo = filer.createSourceFile(viewInjector.getFqcn(), typeElement);

Writer writer = jfo.openWriter();

writer.write(viewInjector.brewJava());

writer.flush();

writer.close();

} catch (IOException e) {

error(typeElement, "Unable to write injector for type %s: %s", typeElement, e.getMessage());

}

}

return true;

}

ButterKnifeProcessor.java 的 process 方法如上，编译时，在此方法中过滤 InjectView 这个 Annotation 到 targetClassMap 后，会根据 targetClassMap 中元素生成不同的 class 文件到最终的 APK 中，然后在运行时调用 ButterKnife.inject(x) 函数时会到之前编译时生成的类中去找

**Annotation — Retrofit**

(1) 调用

@GET("/users/{username}")

User getUser(@Path("username") String username);

(2) 定义

@Documented

@Target(METHOD)

@Retention(RUNTIME)

@RestMethod("GET")

public @interface GET {

String value();

}

(3) 原理

private void parseMethodAnnotations() {

for (Annotation methodAnnotation : method.getAnnotations()) {

Class<? extends Annotation> annotationType = methodAnnotation.annotationType();

RestMethod methodInfo = null;

for (Annotation innerAnnotation : annotationType.getAnnotations()) {

if (RestMethod.class == innerAnnotation.annotationType()) {

methodInfo = (RestMethod) innerAnnotation;

break;

}

}

……

}

}

RestMethodInfo.java 的 parseMethodAnnotations 方法如上，会检查每个方法的每个 Annotation， 看是否被 RestMethod 这个 Annotation 修饰的 Annotation 修饰，这个有点绕，就是是否被 GET、DELETE、POST、PUT、HEAD、PATCH 这些 Annotation 修饰，然后得到 Annotation 信息，在对接口进行动态代理时会掉用到这些 Annotation 信息从而完成调用。 因为 Retrofit 原理设计到动态代理

**Annotation — ActiveAndroid**

(1) 调用

@Column(name = “Name")

public String name;

(2) 定义

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Column {

……

}

(3) 原理

Field idField = getIdField(type);

mColumnNames.put(idField, mIdName);

List<Field> fields = new LinkedList<Field>(ReflectionUtils.getDeclaredColumnFields(type));

Collections.reverse(fields);

for (Field field : fields) {

if (field.isAnnotationPresent(Column.class)) {

final Column columnAnnotation = field.getAnnotation(Column.class);

String columnName = columnAnnotation.name();

if (TextUtils.isEmpty(columnName)) {

columnName = field.getName();

}

mColumnNames.put(field, columnName);

}

}

TableInfo.java 的构造函数如上，运行时，得到所有行信息并存储起来用来构件表信息

## 应用举例：任意bean解析注解的使用

@Target(ElementType.**FIELD**)

@Retention(RetentionPolicy.**RUNTIME**)

**public** **@interface** TreeNodeId {

}

@Target(ElementType.**FIELD**)

@Retention(RetentionPolicy.**RUNTIME**)

**public** **@interface** TreeNodeLabel {

}

@Target(ElementType.**FIELD**)

@Retention(RetentionPolicy.**RUNTIME**)

**public** **@interface** TreeNodePiId {

}

**public** **class** **OrgBean** {

@TreeNodeId

**private** **int** \_id;

@TreeNodePiId

**private** **int** parentId;

@TreeNodeLabel

**private** **String** name;

...

}

**public** **class** **FileBean** {

@TreeNodeId

**private** **int** id;

@TreeNodePiId

**private** **int** pId;

@TreeNodeLabel

**private** **String** label;

**private** **String** desc;

...

}

// 将用户的数据转化为树形数据

**public** **static** <T>List<**Node**> **convertDatas2Nodes**(List<T> datas) **throws** Exception{

List<**Node**> nodes = **new** ArrayList<**Node**>();

**Node** node = **null**;

**for**(T t : datas){

**int** id = -1; **int** pId = -1; **String** label = **null**;

**Class** clazz = t.getClass();

**Field**[] fields = clazz.getDeclaredFields();

**for**(**Field** field : fields) {

**if** (field.getAnnotation(TreeNodeId.**class**) != **null**) {**//获取注解信息,有**TreeNodeId**注解的是需要的field**

field.setAccessible(**true**);**//设置权限,外部可以拿到private属性的field**

id = field.getInt(t);

}

**if** (field.getAnnotation(TreeNodePiId.**class**) != **null**) {

field.setAccessible(**true**);

pId = field.getInt(t);

}

**if** (field.getAnnotation(TreeNodeLabel.**class**) != **null**) {

field.setAccessible(**true**);

label = (**String**) field.get(t);

}

}

node = **new** Node(id, pId, label);

nodes.add(node);

}

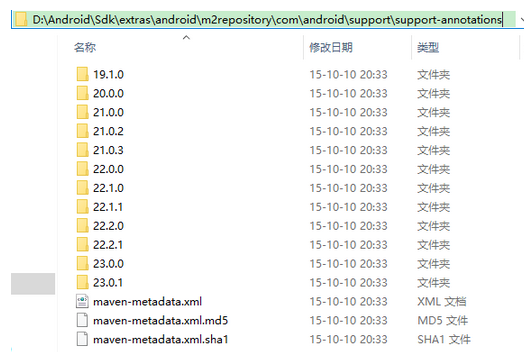
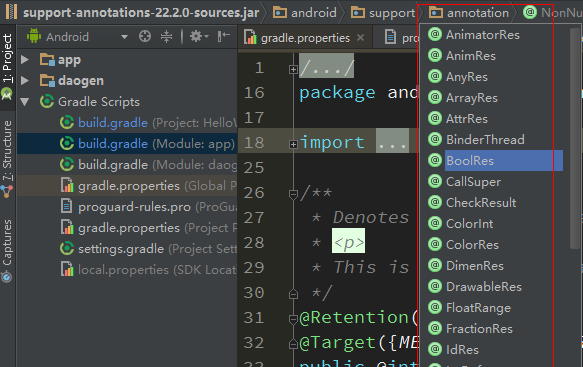
}

# Android Annotation

## 概述

Android Annotation是以Support Library的形式给我们提供的，从安卓19.1开始出现，该包在sdk目录\Sdk\extras\android\m2repository\com\android\support\support-annotations下：

api版本23下，android studio里是直接可以使用的，不用添加依赖库，以前的api可能需要，请酌情增删。api23暂时有39个注解，可以这里查看：

## 安卓注解的8种类型

安卓注解有8种类型，分别是Nullness注解、资源类型注解、线程注解、变量限制注解、权限注解、结果检查注解、CallSuper注解、枚举注解(IntDef和StringDef)。

**Nullness注解**

包括@NoNull和@Nullable，这里以@NoNull为例说明：先看下@NoNull的源码：

@Retention(**CLASS**)

@Target({**METHOD**, **PARAMETER**, **FIELD**})

**public** **@interface** NonNull {

}

这里我们知道，@NoNull编译到类文件里，应用于方法，参数和成员变量，下面我们先来一发：

**public** **class** **MainActivity** **extends** **Activity** {

@NonNull

**private** **String** str;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

str = **null**;

}

}

我们给成员变量str加上了@NoNull注解，也就是str不能为null，而后面我们把str初始化为null，此时as编译器这样提示我们：



也就是str不能为空，这就像是一种约定，很像c语言里的断言assert。这样有什么好处呢？这样可以避免后面的频繁判断str是否为空的操作，如果和别的项目配合呢，可能会出现某些bug，而使用注解可以很好的用于团队合作，项目维护起来也方便。

**资源类型注解**

此类注解以Res结尾，比如@BoolRes, @IdRes, @IntegerRes, @StringRes, @ColorRes等，这里以@ColorRes为例说明，先看下@ColorRes的源码：

@Documented

@Retention(CLASS)

@Target({METHOD, PARAMETER, FIELD, LOCAL\_VARIABLE})

**public** **@interface** ColorRes {

}

这里我们知道，@ColorRes在编译时生效，使用于方法，参数和成员变量和局部变量，下面我们再来一发：

**public** **class** **MainActivity** **extends** **Activity** {

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

getCurColor(**R**.**id**.**ic\_launcher**);

}

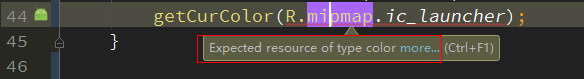
**void** **getCurColor**(@ColorRes **int** color) {

**Toast**.*makeText*(**this**, "Color: " + getString(color), **Toast**.**LENGTH\_LONG**).show();

}

}

我们给getColor函数的参数color加上了@ColorRes注解，即该参数是一个颜色资源，而上面我们上面传递的是id资源，此时as编译器这样提示我们：



我们这样调用就ok的:getCurColor(R.color.material\_blue\_grey\_900);看到了吧，好玩吗？嗯，as很好很强大，，，好的，我们继续。

**权限注解**

直接上代码了，源码自己查阅咯

@RequiresPermission(Manifest.permission.SET\_WALLPAPER)

public abstract void setWallpaper(Bitmap bitmap) throws IOException;

此例说明setWallpaper方法有设置壁纸的权限。

**CallSuper Annotations**

直接来了哈，我们看Activity.java源码里的onCreate方法

@CallSuper

protected void onCreate(Bundle savedInstanceState) {

}

@CallSuper说明所有重写onCreate()方法的方法都要有super.onCreate();

**枚举注解**

枚举注解，这就是大名鼎鼎的@IntDef and @StringDef，我们看看@IntDef源码

@Retention(RetentionPolicy.Source)

@Target({ElementType.**ANNOTATION\_TYPE**})

**public** **@interface** IntRef {

**long**[] **value**() **default** {};

**boolean** **flag**() **default** **false**;

}

可见该注解表示不编译annotation到类文件中，用于注解类型，再来一发？好的，我们看看ActionBar源码吧：

**import** android.support.annotation.IntDef;

...

**public** **abstract** **class** **ActionBar** {

...

@IntDef({**NAVIGATION\_MODE\_STANDARD**, **NAVIGATION\_MODE\_LIST**, **NAVIGATION\_MODE\_TABS**})

@Retention(RetentionPolicy.**SOURCE**)

**public** **@interface** NavigationMode {}

**public** **static** **final** **int** **NAVIGATION\_MODE\_STANDARD** = 0;

**public** **static** **final** **int** **NAVIGATION\_MODE\_LIST** = 1;

**public** **static** **final** **int** **NAVIGATION\_MODE\_TABS** = 2;

@NavigationMode

**public** **abstract** **int** **getNavigationMode**();

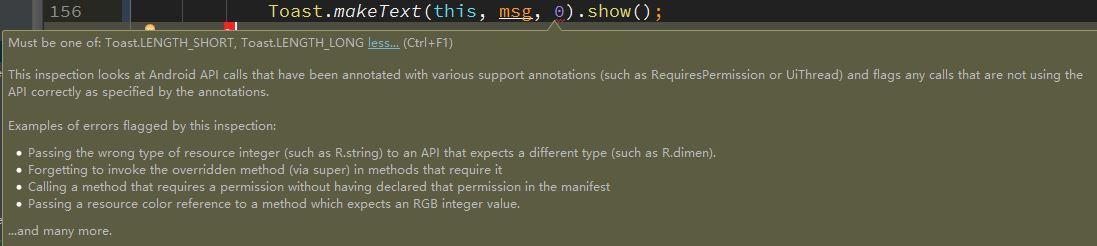
**public** **abstract** **void** setNavigationMode(@NavigationMode **int** mode);

前三句自定义了一个 @NavigationMode 注解，并使用 @IntDef 指定它可以接受的值类型。当使用setNavigationMode方法时，as会提示需要的参数，当然如果你写错的话，as会报错哦，so，as是一个神器。

大家还可以查看Toast源码，定义了一个@Duration的注解，makeText方法的第三个参数设置吐司时间长短的就用到了该注解，也就是时间必须是Toast.LENGTH\_SHORT或Toast.LENGTH\_LONG，如果你写的不是这两个，as会提示你的，比如，我这样写：

**Toast**.*makeText*(**this**, msg, 0).show();

AS 会在这里“0”附近出现提示，按 F1 后出现详情：



其实这个不影响运行，你点击run运行还是可以跑的，因为你run的时候没有运行 lint 检查，这个是可以设置的，会拖慢运行效率

# android 中View, Window, Activity, WindowManager，ViewRoot几者之间的关系

(1)View：最基本的UI组件，表示屏幕上的一个矩形区域。

(2)Window： 表示一个窗口，不一定有屏幕那么大，可以很大也可以很小,它包含一个View tree和窗口的layout 参数。View tree的root View可以通过getDecorView得到。还可以设置Window的Content View。

(3)Activity：Activity包含一个Window，该Window在Activity的attach方法中通过调用PolicyManager.makeNewWindo创建

(4)WindowManager：一个interface，继承自ViewManager。 有一个implementation

(5)class：android.view.WindowManagerImpl

其实WindowManager并不是整个系统的窗口管理器，而是所在应用进程的窗口管理器。系统全局的窗口管理器运行在SystemServer进程中，是一个Service。

ViewRoot通过IWindowSession接口与全局窗口管理器进行交互。 将一个View add到WindowManager时，WindowManagerImpl创建一个ViewRoot来管理该窗口的根View。并通过ViewRoot.setView方法把该View传给ViewRoot。

ViewRoot用于管理窗口的根View，并和global window manger进行交互。ViewRoot中有一个nested class： W，W是一个Binder子类，用于接收global window manager的各种消息， 如按键消息， 触摸消息等。 ViewRoot有一个W类型的成员mWindow，ViewRoot在Constructor中创建一个W的instance并赋值给mWindow。 ViewRoot是Handler的子类， W会通过Looper把消息传递给ViewRoot。 ViewRoot在setView方法中把mWindow传给sWindowSession。

ActivityThread.java中调用wm.addView(decor, l);把它加入到window manager proxy的mViews中，同时为这个decor view创建一个ViewRoot，ViewRoot负责协调decor view与window manager直接绘图、事件处理。

ViewRoot中有IWindowSession和IWindow用来和window manger打交道和接收window manager传过来的消息，消息传过来后ViewRoot分发给decor view，再由decor view进行分发

总之，每个窗口对应着一个Window对象，一个根View和一个ViewRoot对象。要想创建一个窗口，可以调用WindowManager的addView方法，作为参数的view将作为在该窗口上显示的根view

一般来讲，Activity就是一 个可视的人机交互界面。每一个Activity都有一个默认的Window，一 般来讲，这个Window都是全屏的，当然也有例外，比如Dialog的Window就是非全屏的。Activity也可以利用默认Window以外的其 它的Window，比如弹出一个Dialog对话框。Window里可见的内容，就是一层一层的View。Window里的View是通过 setContentView这个接口set进去的，这个ContentView就是Window里处于最根部的View。

对于屏幕显示而 言，整个是Activity 其实确切的说是window,为什么这样说呢？在Activityattach 函数里．有这样一句 mWindow = PolicyManager.makeNewWindow(this);这句是把Activiyt window关联起来了．真正关联的是PhoneWindow.这个window里显示的东西是view(activity是逻辑上的东 西，增加了生命周期管理等. 里面具体的东西也是view。而且启动activity的实现也是往window里加view),往window里加view,是通过调用 WindowManager（WindowManagerImpl）.addView()来实现的。

为什么可以这样说呢？看

public void setContentView(int layoutResID) {

getWindow().setContentView(layoutResID);

}

在 addView里，会为每个view创建一个viewRoot(这是逻辑上的东西，用来负责view的事件处理和逻辑处理，并和 WindowsManagerService建立联系），而在WindowManagerImpl里，会维护viewRoot的数组。最 终会调用ViewRoot.setView()，在setView里会显示该view等。在setView的实现里，会调用Session.add（）来 addWindow,通过这个方法，会将ViewRoot和WindowManagerService联系起来（比如说分发事件等）,并且这个方法里会调 用PhoneWindowManager.prepareAddWindowLw()。

# 开启悬浮窗

**WindowManager介绍**

Android的WindowManager是一个系统的服务，这是负责管理z-ordered列表窗口，以及它们是如何被放在屏幕上。另外，当打开或关闭应用程序或旋转屏幕，它会自动执行窗口转换和动画。

每个Activity都有一个window，用于在屏幕上显示其内容。当你在Activity调用setContentView（）方法，实际上是添加到了Activity的默认window上。默认的window充满整个屏幕，这样你的Activity遮盖其他的window。所以通常你不需要担心Windows 。你只是创建一个Activity，Android会为你做剩下的。

如果你想做一些不寻常的东西像创建浮动窗口并且不要填满屏幕。如果您想创建一个在其他应用程序前面可见的浮动窗口，则不能使用Activity，因为当另一个应用程序到达前台时，您的Activity将停止，其窗口将被隐藏或销毁。相反，您需要从后台服务显示窗口。

**需要权限<uses-permission android:name="android.permission.SYSTEM\_ALERT\_WINDOW" />**

**有些设备需要在应用设置里边，手动授权悬浮窗权限**

**private** **void** **showMoveWindow**() {

//窗口管理器

wm = (WindowManager) getApplicationContext().getSystemService(**WINDOW\_SERVICE**);

//布局参数

**final** WindowManager.**LayoutParams** layoutParams = **new** WindowManager.LayoutParams();

layoutParams.width = WindowManager.**LayoutParams**.**WRAP\_CONTENT**;

layoutParams.height = WindowManager.**LayoutParams**.**WRAP\_CONTENT**;

layoutParams.flags = WindowManager.**LayoutParams**.**FLAG\_NOT\_FOCUSABLE** | WindowManager.**LayoutParams**.**FLAG\_KEEP\_SCREEN\_ON**;

//格式

layoutParams.format = **PixelFormat**.**TRANSLUCENT**;

//类型

layoutParams.type = WindowManager.**LayoutParams**.**TYPE\_PHONE**;

ivDraw = **new** ImageView(**WindowActivity**.**this**);

ivDraw.setBackgroundResource(**R**.**drawable**.**ic\_launcher**);

//触摸事件

ivDraw.setOnTouchListener(**new** **View**.OnTouchListener() {

@Override

**public** **boolean** **onTouch**(**View** v, **MotionEvent** event) {

**switch** (event.getAction()) {

**case** **MotionEvent**.**ACTION\_DOWN**:

startX = (**int**) event.getRawX();

startY = (**int**) event.getRawY();

**break**;

**case** **MotionEvent**.**ACTION\_MOVE**:

endX = (**int**) event.getRawX();

endY = (**int**) event.getRawY();

//计算移动偏移量

dx = endX - startX;

dy = endY - startY;

layoutParams.x += dx;

layoutParams.y += dy;

//更新位置

wm.updateViewLayout(ivDraw, layoutParams);

//重新初始化坐标

startX = (**int**) event.getRawX();

startY = (**int**) event.getRawY();

**break**;

}

**return** **true**;

}

});

//加载view

wm.addView(ivDraw, layoutParams);

}

//关闭悬浮窗

wm.removeView(ivDraw);  
wm = null;

# 双屏异显

**如何在副屏上展示内容？通过Presentation来实现，Presentation继承了Dialog**

**public** **class** **DifferentDislay** **extends** **Presentation**{

**private** **ListView** lv;

**private** List<**String**> list;

**public** **DifferentDislay**(**Context** outerContext, **Display** display) {

**super**(outerContext,display);

}

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**diffrentdisplay\_basket**);

list = **new** ArrayList<>();

list.add("asdfasdfas");

list.add("ghdfh");

list.add("法规或的方式");

lv = (**ListView**) findViewById(**R**.**id**.**lv**);

lv.setAdapter(**new** ArrayAdapter<>(getContext(), android.**R**.**layout**.**simple\_expandable\_list\_item\_1**, list));

}

}

**Activity中实现它**

**public** **class** **MainActivity** **extends** **Activity** {

**private** **DisplayManager** manager;**//屏幕管理类**

**private** **Display**[] displays;**//屏幕数组**

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

manager = (**DisplayManager**) getSystemService(**DISPLAY\_SERVICE**);

displays = manager.getDisplays();**//获取屏幕数组；主屏：displays[0]； 副屏：displays[1]**

**/\*[Display id 0: DisplayInfo{"内置屏幕", app 1280 x 752, real 1280 x 800, largest app 1280 x 1207, smallest app 800 x 727, 69.348 fps, rotation0, density 160 (160.15764 x 160.0) dpi, layerStack 0, type BUILT\_IN, FLAG\_SECURE, FLAG\_SUPPORTS\_PROTECTED\_BUFFERS}, DisplayMetrics{density=1.0, width=1280, height=752, scaledDensity=1.0, xdpi=160.15764, ydpi=160.0}, isValid=true,**

**Display id 1: DisplayInfo{"HDMI 屏幕", app 1280 x 720, real 1280 x 720, largest app 1280 x 720, smallest app 1280 x 720, 77202.195 fps, rotation0, density 213 (213.0 x 213.0) dpi, layerStack 1, type HDMI, FLAG\_SECURE, FLAG\_SUPPORTS\_PROTECTED\_BUFFERS, FLAG\_PRESENTATION}, DisplayMetrics{density=1.3312501, width=1280, height=720, scaledDensity=1.3312501, xdpi=213.0, ydpi=213.0}, isValid=true]\*/**

**//[Context](eclipse-javadoc:%E2%98%82=%E5%8F%8C%E5%B1%8FDemo/src<com.cgy.doublescreen{DifferentDislay.java%E2%98%83DifferentDislay~DifferentDislay~QContext;~QDisplay;%E2%98%82Context) outerContext, [Display](eclipse-javadoc:%E2%98%82=%E5%8F%8C%E5%B1%8FDemo/src<com.cgy.doublescreen{DifferentDislay.java%E2%98%83DifferentDislay~DifferentDislay~QContext;~QDisplay;%E2%98%82Display) display)**

**DifferentDislay** differentDislay = **new** DifferentDislay(**MainActivity**.**this**, displays[1]);

//differentDislay.getWindow().setType(WindowManager.LayoutParams.TYPE\_SYSTEM\_ALERT);//不能加这句话

differentDislay.show();

}

}

**主副屏显示相同界面**

插上数据线，不作任何代码操作，默认的就是显示相同界面。一旦调用了differentDislay.show()，那么必须通过differentDislay.dismiss()，来实现显示相同界面

# 创建桌面快捷方式

概述 ：创建桌面快捷方式相当与创建一个程序的入口，就像我们程序在安装完毕后会自动创建一个图标到桌面。其实创建桌面快捷方式跟创建一个程序入口差不多，但是像QQ会话一样创建一个QQ好友的会话快捷方式，就得动态的创建图标，名字了。

## 首先权限是必不可少的

<uses-permission android:name=*"com.android.launcher.permission.INSTALL\_SHORTCUT"* />

## 项目配置文件里面配置

这个actvity即为你要快捷方式点击后跳转的那一个activity

<activity android:name=*"com.example.shortcut.ChatActivity"* >

<intent-filter>

<category android:name=*"android.intent.category.LAUNCHER"* />

<action android:name=*"android.intent.action.CREATE\_SHORTCUT"* />

</intent-filter>

</activity>

## 然后就是你要创建快捷方式的方法。

CreateShotCut(**MainActivity**.**this**, **ChatActivity**.**class**, "Chat", **R**.**drawable**.**icon**);

**public** **void** **CreateShotCut**(**final** **Context** context, **final** **Class**<?> clazz, **final** **String** name, **int** iconId) {

**Intent** shortcutIntent = **new** Intent(**Intent**.**ACTION\_MAIN**);

// 加入action,和category之后，程序卸载的时候才会主动将该快捷方式也卸载

shortcutIntent.addCategory(**Intent**.**CATEGORY\_LAUNCHER**);

shortcutIntent.setClass(context, clazz);

//创建一个Bundle对象让其保存将要传递的值

**Bundle** bundle = **new** Bundle();

bundle.putString("name", "张三");

bundle.putInt("age", 32);

shortcutIntent.putExtras(bundle);

//设置这条属性，可以使点击快捷方式后关闭其他的任务栈的其他activity，然后创建指定的acticity

shortcutIntent.addFlags(**Intent**.**FLAG\_ACTIVITY\_CLEAR\_TOP**);

// 创建快捷方式的Intent

**Intent** shortcut = **new** Intent(**Intent**.**ACTION\_CREATE\_SHORTCUT**);

// 不允许重复创建

shortcut.putExtra("duplicate", **false**);

// 点击快捷图片，运行的程序主入口

shortcut.putExtra(**Intent**.**EXTRA\_SHORTCUT\_INTENT**, shortcutIntent);

// 需要现实的名称

shortcut.putExtra(**Intent**.**EXTRA\_SHORTCUT\_NAME**, name);

// 快捷图片

Parcelable icon = **Intent**.**ShortcutIconResource**.*fromContext*(getApplicationContext(), iconId);

Shortcut.putExtra(**Intent**.**EXTRA\_SHORTCUT\_ICON\_RESOURCE**, icon);

shortcut.setAction("com.android.launcher.action.INSTALL\_SHORTCUT");

context.sendBroadcast(shortcut);

}

# 判断app何时是打开或者关闭

## 存在的问题

Android开发中不可避免的会遇到需要检查app何时进入前台，何时被用户关闭。奇怪的是，要达到这个目的并不容易。检查app第一次启动并不难，但要判断它何时重新打开和关闭就没有那么简单了。

判断一个app打开和关闭的关键在于判断它的activities是否正在前台显示。让我们先从简单的例子开始，一个只有一个activity的app，而且不支持水平模式。这样想要判断app是打开还是关闭只需要检查activity的onStart和onStop方法即可：

@Override

protected void onStart() {

super.onStart();

// The Application has been opened!

}

@Override

protected void onStop() {

super.onStop();

// The Application has been closed!

}

上面例子的问题在于当需要支持水平模式时该方法就失效了。当我们旋转设备时activity将会重建，onStart方法将被再次调用，这时将会错误的判断为app第二次被打开。为了处理设备旋转的情况，我们需要增加一个校验步骤。当activity退出时启动一个定时器，用于判断短时间内app的这个activity是否又被启动，如果没有，说明用户真的退出了这个app，如果重新启动了这个activity，说明用户还逗留在这个app中。这种校验方式也适用于拥有多个activities的app，因为从app的一个activity跳转到另一个activity也可以用这种校验方式来处理。

使用这个技术我创建了一个管理类，所有的activities在可见和不可见时都会通知这个管理类。这个管理类为每个activity处理上述的校验步骤，从而避免错误的检测。它也提供了发布订阅（观察者）模式，任何对app启动和关闭感兴趣的模块都可以通过它来得到对应的通知。

## 这个管理类的使用分为三个步骤：

### 把它添加到你的工程中

/\*\*

\* This class is responsible for tracking all currently open activities.

\* By doing so this class can detect when the application is in the foreground

\* and when it is running in the background.

\*/

**public** **class** **AppForegroundStateManager** {

**private** **static** **final** **String** **TAG** = **AppForegroundStateManager**.**class**.getSimpleName();

**private** **static** **final** **int** **MESSAGE\_NOTIFY\_LISTENERS** = 1;

**public** **static** **final** **long** **APP\_CLOSED\_VALIDATION\_TIME\_IN\_MS** = 3 \* **DateUtils**.**SECOND\_IN\_MILLIS**; // 30 Seconds

**private** **Reference**<**Activity**> mForegroundActivity;

**private** Set<OnAppForegroundStateChangeListener> mListeners = **new** HashSet<>();

**private** AppForegroundState mAppForegroundState = AppForegroundState.**NOT\_IN\_FOREGROUND**;

**private** **NotifyListenersHandler** mHandler;

// Make this class a thread safe singleton

**private** **static** **class** **SingletonHolder** {

**public** **static** **final** **AppForegroundStateManager** **INSTANCE** = **new** AppForegroundStateManager();

}

**public** **static** **AppForegroundStateManager** **getInstance**() {

**return** **SingletonHolder**.**INSTANCE**;

}

**private** **AppForegroundStateManager**() {

// Create the handler on the main thread

mHandler = **new** NotifyListenersHandler(**Looper**.*getMainLooper*());

}

**public** **enum** AppForegroundState {

**IN\_FOREGROUND**,

**NOT\_IN\_FOREGROUND**

}

**public** **interface** OnAppForegroundStateChangeListener {

/\*\* Called when the foreground state of the app changes \*/

**public** **void** **onAppForegroundStateChange**(AppForegroundState newState);

}

/\*\* An activity should call this when it becomes visible \*/

**public** **void** **onActivityVisible**(**Activity** activity) {

**if** (mForegroundActivity != **null**) mForegroundActivity.clear();

mForegroundActivity = **new** WeakReference<>(activity);

determineAppForegroundState();

}

/\*\* An activity should call this when it is no longer visible \*/

**public** **void** **onActivityNotVisible**(**Activity** activity) {

/\*

\* The foreground activity may have been replaced with a new foreground activity in our app.

\* So only clear the foregroundActivity if the new activity matches the foreground activity.

\*/

**if** (mForegroundActivity != **null**) {

**Activity** ref = mForegroundActivity.get();

**if** (activity == ref) {

// This is the activity that is going away, clear the reference

mForegroundActivity.clear();

mForegroundActivity = **null**;

}

}

determineAppForegroundState();

}

/\*\* Use to determine if this app is in the foreground \*/

**public** **Boolean** **isAppInForeground**() {

**return** mAppForegroundState == AppForegroundState.**IN\_FOREGROUND**;

}

/\*\*

\* Call to determine the current state, update the tracking global, and notify subscribers if the state has changed.

\*/

**private** **void** **determineAppForegroundState**() {

/\* Get the current state \*/

AppForegroundState oldState = mAppForegroundState;

/\* Determine what the new state should be \*/

**final** **boolean** isInForeground = mForegroundActivity != **null** && mForegroundActivity.get() != **null**;

mAppForegroundState = isInForeground ? AppForegroundState.**IN\_FOREGROUND** : AppForegroundState.**NOT\_IN\_FOREGROUND**;

/\* If the new state is different then the old state the notify subscribers of the state change \*/

**if** (mAppForegroundState != oldState) {

validateThenNotifyListeners();

}

}

/\*\*

\* Add a listener to be notified of app foreground state change events.

\*

\* **@param** listener

\*/

**public** **void** **addListener**(@NonNull OnAppForegroundStateChangeListener listener) {

mListeners.add(listener);

}

/\*\*

\* Remove a listener from being notified of app foreground state change events.

\*

\* **@param** listener

\*/

**public** **void** **removeListener**(OnAppForegroundStateChangeListener listener) {

mListeners.remove(listener);

}

/\*\* Notify all listeners the app foreground state has changed \*/

**private** **void** **notifyListeners**(AppForegroundState newState) {

android.util.**Log**.*i*(**TAG**, "Notifying subscribers that app just entered state: " + newState);

**for** (OnAppForegroundStateChangeListener listener : mListeners) {

listener.onAppForegroundStateChange(newState);

}

}

/\*\*

\* This method will notify subscribes that the foreground state has changed when and if appropriate.

\* <br><br>

\* We do not want to just notify listeners right away when the app enters of leaves the foreground. When changing orientations or opening and

\* closing the app quickly we briefly pass through a NOT\_IN\_FOREGROUND state that must be ignored. To accomplish this a delayed message will be

\* Sent when we detect a change. We will not notify that a foreground change happened until the delay time has been reached. If a second

\* foreground change is detected during the delay period then the notification will be canceled.

\*/

**private** **void** **validateThenNotifyListeners**() {

// If the app has any pending notifications then throw out the event as the state change has failed validation

**if** (mHandler.hasMessages(**MESSAGE\_NOTIFY\_LISTENERS**)) {

android.util.**Log**.*v*(**TAG**, "Validation Failed: Throwing out app foreground state change notification");

mHandler.removeMessages(**MESSAGE\_NOTIFY\_LISTENERS**);

} **else** {

**if** (mAppForegroundState == AppForegroundState.**IN\_FOREGROUND**) {

// If the app entered the foreground then notify listeners right away; there is no validation time for this

mHandler.sendEmptyMessage(**MESSAGE\_NOTIFY\_LISTENERS**);

} **else** {

// We need to validate that the app entered the background. A delay is used to allow for time when the application went into the

// background but we do not want to consider the app being backgrounded such as for in app purchasing flow and full screen ads.

mHandler.sendEmptyMessageDelayed(**MESSAGE\_NOTIFY\_LISTENERS**, **APP\_CLOSED\_VALIDATION\_TIME\_IN\_MS**);

}

}

}

**private** **class** **NotifyListenersHandler** **extends** **Handler** {

**private** **NotifyListenersHandler**(**Looper** looper) {

**super**(looper);

}

@Override

**public** **void** **handleMessage**(**Message** inputMessage) {

**switch** (inputMessage.what) {

// The decoding is done

**case** **MESSAGE\_NOTIFY\_LISTENERS**:

/\* Notify subscribers of the state change \*/

android.util.**Log**.*v*(**TAG**, "App just changed foreground state to: " + mAppForegroundState);

notifyListeners(mAppForegroundState);

**break**;

**default**:

**super**.handleMessage(inputMessage);

}

}

}

}

### Activities在可见性改变的需要发送通知

app中所有activities都要增加下面的代码，用于可见性改变时通知管理类。最好的实现方式是把这段代码加到工程的BaseActivity中。

**public** **class** **BaseActivity** **extends** **Activity** {

@Override

**protected** **void** **onStart**() {

**super**.onStart();

**AppForegroundStateManager**.*getInstance*().onActivityVisible(**this**);

}

@Override

**protected** **void** **onStop**() {

**super**.onStop();

**AppForegroundStateManager**.*getInstance*().onActivityNotVisible(**this**);

}

}

### 订阅app的前台可见性改变事件

在感兴趣的模块中订阅app前台可见性改变事件，application类的onCreate函数是一个不错的地方，它可以保证每次app启动和关闭，你都能得到通知。

**public** **class** **MyApplication** **extends** **Application** {

@Override

**public** **void** **onCreate**() {

**super**.onCreate();

**AppForegroundStateManager**.*getInstance*().addListener(**new** OnAppForegroundStateChangeListener() {

@Override

**public** **void** **onAppForegroundStateChange**(AppForegroundState newState) {

**if** (**AppForegroundStateManager**.AppForegroundState.**IN\_FOREGROUND** == newState) {

**Toast**.*makeText*(getApplicationContext(), "打开", **Toast**.**LENGTH\_SHORT**).show();

} **else** {

**Toast**.*makeText*(getApplicationContext(), "关闭", **Toast**.**LENGTH\_SHORT**).show();

}

}

});

}

}

## 进一步的思考

有一些细节需要进一步讨论，下面讨论的几点针对具体的应用可以做微调。

### 校验时间

校验定时器检查app是否真的进入后台的时间间隔是多少合适呢？上面的代码设置为30秒，原因如下。

当你的app在运行时，可能存在第三方的activities会覆盖全屏幕，一些常见的例子是Google应用内购买和Facebook登录注册页面。这些情况下你的app都会被迫进入后台，前台用于显示这些第三方页面。如果把这种情况当做用户离开了你的app，显然是不对的。30秒超时设置就是用 来避免这种情况的。例如当用户在30秒内完成应用内购买，大部分用户都可以做得到，那么就不会当做用户突然离开app了。

如果你的app不存在上述这种情况，我建议可以把你的校验时间设置为4秒，这样对于低配设备当屏幕旋转重新创建activity的时间间隔是合适的。

### CPU休眠

可能存在的问题是当用户关闭app或者app仍处于前台时用户锁屏了，这时CPU可能不会等到定时器检测就休眠了。为了保证这种情况下定时器能够正常检测用户退出app，我们需要持有wakelock防止CPU休眠直到app关闭事件被确认。实践中相比使用wakelock，这种情况并不算问题。

### 判断app是如何启动的

现在我们已经知道如何检测app何时启动和关闭，但我们不知道app是如何启动的。是用户点击通知栏消息？还是点击一个链接？亦或是他们直接通过桌面图标或最近使用启动？

# 安卓5.0、6.0、7.0新特性

## Android5.0新特性:

1. 代号为Lollipop(棒棒糖),采用了Google最新推行的Material Design设计语言,用户切换activity时,有无缝的过度位移动画.
2. 运行机制改为更新的ART。相比Dalvik，ART的处理机制完全不同，它会在应用程序安装时就把程序代码转换成机器语言，让程序成为真正的本地应用。这样做的好处是程序的启动时间被极大的提高，运行速度也会更快。电量消耗的更少，系统运行也更加的流畅。
3. Notifications通知详情可以由用户自己设计.用户在浏览某个app时,可以在当前app上浮动一个窗口来显示关键的通知信息.在锁屏界面也可以直接查看通知消息了,用户还可以直接在锁屏的界面进行消息回复或查看.
4. 允许电视应用程序来处理诸如HDMI输入,电视调谐器,视频流,和IPTV接收机在更大的屏幕上.
5. 折叠效果的任务管理器.
6. 支持Khronos OpenGL ES 3.，因此在游戏和app的性能上行有更好的表现 。
7. 更强大的音频,一种新的音频捕获设计提供延迟音频输入.
8. 增强摄像机和视频.
9. 屏幕获取和屏幕分享.
10. 新型传感器.
11. 优化电量.Project Volta 电池续航改进计划,增加了Battery Saver模式,在低电量的时候系统会自动降低屏幕亮度,限制自动更换背景等功能.
12. 新的API支持,蓝牙4.1,USB Audio,多人分享等其他特性.

## Android 6.0 新特性:

1. 运行时权限,用户可以在已安装的app上立刻同意或者拒绝权限.
2. 睡眠模式和APP备用模式,这个版本介绍了一种新的节能优化对于闲置的设备和APP.睡眠模式:当屏幕静止时,闲置一段时间,设备就会进入睡眠模式,让系统保持睡眠的状态,
3. 去除了Apache的http Client相关方法.Google建议用httpURLConnection来替代httpclient的相关方法,如果执意继续使用httpclient 相关方法,那么你就需要自取导入Apache的jar包.在Android Studio中的Gradle里通过userlibrary'org.Apache.http.legacy'去引用.
4. 大量漂亮流畅的动画,Android 6.0系统增加了大量漂亮的过度动画,可以从视觉上减少卡顿感.
5. 相机新增专业模式,一直以来,原生的Android相机都被吐槽太过简单甚至简陋了,在此次专业模式,增加了快门速度调节和曝光度调节等新功能.
6. 全新的电源键菜单,一般来说 Android的电源键菜单都是关机/重启/飞行,Android 6.0变成了关机/重启/紧急,紧急模式是为了手机快没电的时候设计的,相当于飞行模式的高级版,可以关闭一切好点应用,尽最大可能节省电量.
7. 可自定义锁界面样式,支持电话,信息,相机等快捷方式在锁屏界面的定制,用户可以根据自己的喜好调整这些图标的位置,或者开启或关闭这些快捷方式.
8. 支持快速充电的切换.可以开启或关闭快速充电.
9. 支持文件夹拖拽应用,可在应用从一个文件夹内直接拖到另一个文件夹,简化了此前繁琐的操作方式,拖拽的过程和Windows的拖拽功能有点相似.
10. 支持raw格式照片.

## Android 7.0新特性：

1. 分屏多任务支持
2. 画中画
3. 通知栏快速回复
4. OpenJDK替换java API
5. Android7.0采用了一项具有实时代码剖析功能的ARI JIT编译器,它能够在安卓应用程序在运行时不断提高自身的性能

# 测网速

**public** **class** **NetSpeed** {

**private** **final** **static** **String** **TAG** = "NetSpeed";

**private** **long** preRxBytes = 0;

**private** **Timer** mTimer = **null**;

**private** **Context** mContext;

**private** **static** **NetSpeed** ***mNetSpeed***;

**private** **Handler** mHandler;

**private** **NetSpeed**(**Context** mContext, **Handler** mHandler) {

**this**.mContext = mContext;

**this**.mHandler = mHandler;

}

**public** **static** **NetSpeed** **getInstant**(**Context** mContext, **Handler** mHandler) {

**if** (***mNetSpeed*** == **null**) {

***mNetSpeed*** = **new** NetSpeed(mContext, mHandler);

}

**return** ***mNetSpeed***;

}

**private** **long** **getNetworkRxBytes**() {

**int** currentUid = getUid();

**Log**.*e*(**TAG**, "currentUid =" + currentUid);

**if** (currentUid < 0) {

**return** 0;

}

**long** rxBytes = **TrafficStats**.*getUidRxBytes*(currentUid);

**if** (rxBytes == **TrafficStats**.**UNSUPPORTED**) {

**Log**.*e*(**TAG**, "getUidRxBytes fail !!! Just getTotalRxBytes!!!");

rxBytes = **TrafficStats**.*getTotalRxBytes*();

}

**return** rxBytes;

}

**public** **int** **getNetSpeed**() {

**long** curRxBytes = getNetworkRxBytes();

**long** bytes = curRxBytes - preRxBytes;

preRxBytes = curRxBytes;

**int** kb = (**int**) **Math**.*floor*(bytes / 1024 + 0.5);

**return** kb;

}

**public** **void** **startCalculateNetSpeed**() {

preRxBytes = getNetworkRxBytes();

**if** (mTimer != **null**) {

mTimer.cancel();

mTimer = **null**;

}

**if** (mTimer == **null**) {

mTimer = **new** Timer();

mTimer.schedule(**new** TimerTask() {

@Override

**public** **void** **run**() {

**Message** msg = **new** Message();

msg.what = 1;

msg.arg1 = getNetSpeed();

mHandler.sendMessage(msg);

}

}, 1000, 1000);

}

}

**public** **void** **stopCalculateNetSpeed**() {

**if** (mTimer != **null**) {

mTimer.cancel();

mTimer = **null**;

}

}

**private** **int** **getUid**() {

**try** {

**PackageManager** pm = mContext.getPackageManager();

**ApplicationInfo** ai = pm.getApplicationInfo(

mContext.getPackageName(), **PackageManager**.**GET\_ACTIVITIES**);

**return** ai.uid;

} **catch** (**NameNotFoundException** e) {

e.printStackTrace();

}

**return** -1;

}

}

# 线程死锁和wait()，notify()

## 死锁

### 同个Runnable被new了两次就相当于俩线程

**private** **void** **doSomeThing**() {

**new** Thread(**new** MyRunnable(**true**)).start();

**new** Thread(**new** MyRunnable(**false**)).start();

}

**static** **class** **Lock** {

**static** **Lock** ***locka*** = **new** Lock();

**static** **Lock** ***lockb*** = **new** Lock();

}

**class** **MyRunnable** **implements** Runnable {

**private** **boolean** flag;

**public** **MyRunnable**(**boolean** flag) {

**this**.flag = flag;

}

@Override

**public** **void** **run**() {

**if** (flag) {

**while** (**true**) {

**synchronized** (**Lock**.***locka***) {

**try** {**Thread**.*sleep*(1000);} **catch** (**InterruptedException** e) {e.printStackTrace();}

**Log**.*e*("tag", "thread1-----locka");

**synchronized** (**Lock**.***lockb***) {

**Log**.*e*("tag", "thread1-----lockb");

}

}

}

}**else** {

**while** (**true**) {

**synchronized** (**Lock**.***lockb***) {

**try** {**Thread**.*sleep*(1000);} **catch** (**InterruptedException** e) {e.printStackTrace();}

**Log**.*e*("tag", "thread2-----lockb");

**synchronized** (**Lock**.***locka***) {

**Log**.*e*("tag", "thread2-----locka");

}

}

}

}

}

}

### 俩Runnable

**private** **void** **doSomeThing**() {

**new** Thread(**new** MyRunnable1()).start();

**new** Thread(**new** MyRunnable2()).start();

}

**static** **class** **Lock** {

**static** **Lock** ***locka*** = **new** Lock();

**static** **Lock** ***lockb*** = **new** Lock();

}

**class** **MyRunnable1** **implements** Runnable {

@Override

**public** **void** **run**() {

**while** (**true**) {

**synchronized** (**Lock**.***locka***) {

**try** {**Thread**.*sleep*(1000);} **catch** (**InterruptedException** e) {e.printStackTrace();}

**Log**.*e*("tag", "thread1-----locka");

**synchronized** (**Lock**.***lockb***) {

**Log**.*e*("tag", "thread1-----lockb");

}

}

}

}

}

**class** **MyRunnable2** **implements** Runnable {

@Override

**public** **void** **run**() {

**while** (**true**) {

**synchronized** (**Lock**.***lockb***) {

**try** {**Thread**.*sleep*(1000);} **catch** (**InterruptedException** e) {e.printStackTrace();}

**Log**.*e*("tag", "thread2-----lockb");

**synchronized** (**Lock**.***locka***) {

**Log**.*e*("tag", "thread2-----locka");

}

}

}

}

}

## 生产消费，使用wait()，notify()和锁

**private** **void** **doSomeThing**() {

**Lock** lock = **new** Lock();

**new** Thread(**new** Input(lock)).start();

**new** Thread(**new** Output(lock)).start();

}

**class** **Lock** {

**int** tickes = 0;

**boolean** flag = **false**;

}

**class** **Input** **implements** Runnable {

**private** **Lock** lock;

**public** **Input**(**Lock** lock) {

**this**.lock = lock;

}

@Override

**public** **void** **run**() {

**while** (**true**) {

**synchronized** (lock) {

**if** (lock.flag) {

**try** {**Thread**.*sleep*(1000);} **catch** (**InterruptedException** e) {e.printStackTrace();}

}

lock.tickes++;

**Log**.*e*("tag", lock.tickes + "");

lock.flag = **true**;

lock.notify();

}

}

}

}

**class** **Output** **implements** Runnable {

**private** **Lock** lock;

**public** **Output**(**Lock** lock) {

**this**.lock = lock;

}

@Override

**public** **void** **run**() {

**while** (**true**) {

**synchronized** (lock) {

**if** (!lock.flag) {

**try** {**Thread**.*sleep*(1000);} **catch** (**InterruptedException** e) {e.printStackTrace();}

}

lock.tickes--;

**Log**.*e*("tag", lock.tickes + "");

lock.flag = **false**;

lock.notify();

}

}

}

}

}

# 获取机器唯一标识符

## 解释

### DEVICE\_ID

这是Android系统为开发者提供的用于标识手机设备的串号，也是各种方法中普适性较高的，可以说几乎所有的设备都可以返回这个串号，并且唯一性良好。这个DEVICE\_ID可以同通过下面的方法获取：

TelephonyManager tm = (TelephonyManager)getSystemService(Context.TELEPHONY\_SERVICE);

String DEVICE\_ID = tm.getDeviceId();

假设我们确实需要用到真实设备的标识，可能就需要用到DEVICE\_ID。在以前，我们的Android设备是手机，这个DEVICE\_ID可以同通过TelephonyManager.getDeviceId()获取，它根据不同的手机设备返回IMEI，MEID或者ESN码，但它在使用的过程中会遇到很多问题：

* 非手机设备： 如果只带有Wifi的设备或者音乐播放器没有通话的硬件功能的话就没有这个DEVICE\_ID
* 权限： 获取DEVICE\_ID需要READ\_PHONE\_STATE权限，但如果我们只为了获取它，没有用到其他的通话功能，那这个权限有点大才小用
* bug：在少数的一些手机设备上，该实现有漏洞，会返回垃圾，如:zeros或者asterisks的产品

### MAC ADDRESS

可以使用手机Wifi或蓝牙的MAC地址作为设备标识，但是并不推荐这么做，原因有以下两点：

* 硬件限制：并不是所有的设备都有Wifi和蓝牙硬件，硬件不存在自然也就得不到这一信息。
* 获取的限制：如果Wifi没有打开过，是无法获取其Mac地址的；而蓝牙是只有在打开的时候才能获取到其Mac地址。

获取Wifi Mac地址：

WifiManager wifi = (WifiManager) context.getSystemService(Context.WIFI\_SERVICE);

WifiInfo info = wifi.getConnectionInfo();

String wifiMac = info.getMacAddress();

### Sim Serial Number

装有SIM卡的Android 2.3以上设备，可以通过下面的方法获取到Sim Serial Number，注意：对于CDMA设备，返回的是一个空值！

TelephonyManager tm = (TelephonyManager)getSystemService(Context.TELEPHONY\_SERVICE);

String SimSerialNumber = tm.getSimSerialNumber();

### ANDROID\_ID

在设备首次启动时，系统会随机生成一个64位的数字，并把这个数字以16进制字符串的形式保存下来，这个16进制的字符串就是ANDROID\_ID，当设备被wipe后该值会被重置。可以通过下面的方法获取：

import android.provider.Settings;

String ANDROID\_ID = Settings.System.getString(getContentResolver(), Settings.System.ANDROID\_ID);

ANDROID\_ID可以作为设备标识，但需要注意：

* 厂商定制系统的Bug：不同的设备可能会产生相同的ANDROID\_ID：9774d56d682e549c。
* 厂商定制系统的Bug：有些设备返回的值为null。
* 设备差异：对于CDMA设备，ANDROID\_ID和TelephonyManager.getDeviceId() 返回相同的值。
* 它在Android <=2.1 or Android >=2.3的版本是可靠、稳定的，但在2.2的版本并不是100%可靠的

### Serial Number

Android系统2.3版本以上可以通过下面的方法得到Serial Number，且非手机设备也可以通过该接口获取。

String SerialNumber = android.os.Build.SERIAL;

## 方法一

public class SharedPreferenceUtil {

private static final String SHAREDPREFERENCES\_NAME = "wkllme\_sp";

public static SharedPreferences getAppSp() {

return App.getInstance().getSharedPreferences(SHAREDPREFERENCES\_NAME, Context.MODE\_PRIVATE);

}

}

public class SharePreferenceHelper {

public static String deviceIdEmptyAndUUIDKey = "deviceIdEmptyAndUUIDKey";

public String getDdeviceIdEmptyAndUUID() {

String result = SharedPreferenceUtil.getAppSp().getString(deviceIdEmptyAndUUIDKey, "");

if (TextUtils.isEmpty(result)) {

result = MD5Util.MD5Encode(DateUtil.getCurrent\_yMdHms(), "utf-8");

saveDdeviceIdEmptyAndUUID(result);

}

return result;

}

public void saveDdeviceIdEmptyAndUUID(String s) {

try {

SharedPreferenceUtil.getAppSp().edit().putString(deviceIdEmptyAndUUIDKey, s).apply();

} catch (Exception e) {

e.printStackTrace();

}

}

}

public static String getDeviceIdMd5(Context context) {

return MD5Util.MD5Encode(getDeviceId(context), "utf-8");

}

public static String getDeviceId(Context context) {

StringBuilder deviceId = new StringBuilder();

// 渠道标志

deviceId.append("a");

try {

//wifi mac地址

WifiManager wifi = (WifiManager) context.getSystemService(Context.WIFI\_SERVICE);

WifiInfo info = wifi.getConnectionInfo();

String wifiMac = info.getMacAddress();

if (!isEmpty(wifiMac)) {

deviceId.append("wifi");

deviceId.append(wifiMac);

Log.e("getDeviceId : ", deviceId.toString());

return deviceId.toString();

}

//IMEI（imei）

TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.TELEPHONY\_SERVICE);

String imei = tm.getDeviceId();

if (!isEmpty(imei)) {

deviceId.append("imei");

deviceId.append(imei);

//如果上面都没有， 则生成一个id：随机码

String uuid = new SharePreferenceHelper().getDdeviceIdEmptyAndUUID();

if (!isEmpty(uuid)) {

deviceId.append("id");

deviceId.append(uuid);

Log.e("getDeviceId : ", deviceId.toString());

return deviceId.toString();

}

Log.e("getDeviceId : ", deviceId.toString());

return deviceId.toString();

}

//序列号（sn）

String sn = tm.getSimSerialNumber();

if (!isEmpty(sn)) {

deviceId.append("sn");

deviceId.append(sn);

Log.e("getDeviceId : ", deviceId.toString());

return deviceId.toString();

}

} catch (Exception e) {

e.printStackTrace();

deviceId.append("id").append(new SharePreferenceHelper().getDdeviceIdEmptyAndUUID());

}

Log.e("getDeviceId : ", deviceId.toString());

return deviceId.toString();

}

## 方法二：

public class UniqueIdentifierUtil {

private static final String TAG = "UniqueIdentifierTag";

/\*\*

\* The IMEI: 仅仅只对android手机有效:

\* android.permission.READ\_PHONE\_STATE

\*

\* @param context

\* @return

\*/

private static String getSzImei(Context context) {

TelephonyManager TelephonyMgr = (TelephonyManager) context.getSystemService(TELEPHONY\_SERVICE);

String szImei = TelephonyMgr.getDeviceId();

return szImei == null ? "" : szImei;

}

/\*\*

\* Pseudo-Unique ID, 这个在任何Android手机中都有效

\*/

private static String getSzDevIDShort() {

String szDevIDShort = "35" + //we make this look like a valid IMEI

Build.BOARD.length() % 10 +

Build.BRAND.length() % 10 +

Build.CPU\_ABI.length() % 10 +

Build.DEVICE.length() % 10 +

Build.DISPLAY.length() % 10 +

Build.HOST.length() % 10 +

Build.ID.length() % 10 +

Build.MANUFACTURER.length() % 10 +

Build.MODEL.length() % 10 +

Build.PRODUCT.length() % 10 +

Build.TAGS.length() % 10 +

Build.TYPE.length() % 10 +

Build.USER.length() % 10; //13 digits

return szDevIDShort == null ? "" : szDevIDShort;

}

/\*\*

\* The WLAN MAC Address string，WLan不必打开，就可读取些值

\* android.permission.ACCESS\_WIFI\_STATE

\*/

private static String getSzWLANMAC(Context context) {

WifiManager wm = (WifiManager) context.getSystemService(Context.WIFI\_SERVICE);

String szWLANMAC = wm.getConnectionInfo().getMacAddress();

return szWLANMAC == null ? "" : szWLANMAC;

}

/\*\*

\* The BT MAC Address string，蓝牙没有必要打开，也能读取

\* android.permission.ACCESS\_WIFI\_STATE

\*/

private static String getSzBTMAC() {

BluetoothAdapter m\_BluetoothAdapter = null; // Local Bluetooth adapter

m\_BluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

String szBTMAC = m\_BluetoothAdapter.getAddress();

return szBTMAC == null ? "" : szBTMAC;

}

/\*\*

\* 获取唯一标识ID

\*

\* @param context

\* @return

\*/

public static String getUniqueIndentifierId(Context context) {

String mId;

String szImei = getSzImei(context);

Log.e(TAG, szImei);

String szDevIDShort = getSzDevIDShort();

Log.e(TAG, szDevIDShort);

String szWLANMAC = getSzWLANMAC(context);

Log.e(TAG, szWLANMAC);

String szBTMAC = getSzBTMAC();

Log.e(TAG, szBTMAC);

String addStr = szImei + szDevIDShort + szWLANMAC + szBTMAC;

Log.e(TAG, addStr);

mId = MD5Util.MD5Encode(addStr);

Log.e(TAG, mId);

// ToastUtil.shortShow(szImei + "---" + szDevIDShort + "---" + szWLANMAC + "---" + szBTMAC

// + "---" + addStr + "---" + mId);

return mId;

}

}

# Zxing二维码

## 概述

### 维码的定义

二维码又称QR Code，QR全称Quick Response，是一个近几年来移动设备上超流行的一种编码方式，它比传统的Bar Code条形码能存更多的信息，也能表示更多的数据类型

### 二维码生成方案

通过google等搜索引擎，发现很多方案的源头都指向了GitHub的开源库. https://github.com/zxing/zxing。

### zxing 定义

ZXing全称zebra crossing,翻译过来就是『斑马线』的意思。ZXing是一个采用Java实现的、开源的、支持多格式（一维/二维）的条形码图像处理库。

## 使用方法

### zxing 库的引用，有两种方式

* 在 http://search.maven.org 搜索 zxing,下载jar包，进行集成。

如果使用的 android studio ,可以直接在gradle引用。compile 'com.google.zxing:core:3.3.0'

### zxing 库的使用

核心的功能，接收一个传入的字符串，生成一个Bitmap对象并返回。 以下是生成二维码的工具类。

/\*\*

\* @ClassName: QRCodeUtil

\* @Description: 二维码工具类

\* @Author Wangnan

\* @Date 2017/2/8

\*/

public class QRCodeUtil {

/\*\*

\* 创建二维码位图

\*

\* @param content 字符串内容(支持中文)

\* @param width 位图宽度(单位:px)

\* @param height 位图高度(单位:px)

\* @return

\*/

@Nullable

public static Bitmap createQRCodeBitmap(String content, int width, int height){

return createQRCodeBitmap(content, width, height, "UTF-8", "H", "2", Color.BLACK, Color.WHITE);

}

/\*\*

\* 创建二维码位图 (支持自定义配置和自定义样式)

\*

\* @param content 字符串内容

\* @param width 位图宽度,要求>=0(单位:px)

\* @param height 位图高度,要求>=0(单位:px)

\* @param character\_set 字符集/字符转码格式 (支持格式:{@link CharacterSetECI })。传null时,zxing源码默认使用 "ISO-8859-1"

\* @param error\_correction 容错级别 (支持级别:{@link ErrorCorrectionLevel })。传null时,zxing源码默认使用 "L"

\* @param margin 空白边距 (可修改,要求:整型且>=0), 传null时,zxing源码默认使用"4"。

\* @param color\_black 黑色色块的自定义颜色值

\* @param color\_white 白色色块的自定义颜色值

\* @return

\*/

@Nullable

public static Bitmap createQRCodeBitmap(String content, int width, int height,

@Nullable String character\_set, @Nullable String error\_correction, @Nullable String margin,

@ColorInt int color\_black, @ColorInt int color\_white){

/\*\* 1.参数合法性判断 \*/

if(TextUtils.isEmpty(content)){ // 字符串内容判空

return null;

}

if(width < 0 || height < 0){ // 宽和高都需要>=0

return null;

}

try {

/\*\* 2.设置二维码相关配置,生成BitMatrix(位矩阵)对象 \*/

Hashtable<EncodeHintType, String> hints = new Hashtable<>();

if(!TextUtils.isEmpty(character\_set)) {

hints.put(EncodeHintType.CHARACTER\_SET, character\_set); // 字符转码格式设置

}

if(!TextUtils.isEmpty(error\_correction)){

hints.put(EncodeHintType.ERROR\_CORRECTION, error\_correction); // 容错级别设置

}

if(!TextUtils.isEmpty(margin)){

hints.put(EncodeHintType.MARGIN, margin); // 空白边距设置

}

BitMatrix bitMatrix = new QRCodeWriter().encode(content, BarcodeFormat.QR\_CODE, width, height, hints);

/\*\* 3.创建像素数组,并根据BitMatrix(位矩阵)对象为数组元素赋颜色值 \*/

int[] pixels = new int[width \* height];

for(int y = 0; y < height; y++){

for(int x = 0; x < width; x++){

if(bitMatrix.get(x, y)){

pixels[y \* width + x] = color\_black; // 黑色色块像素设置

} else {

pixels[y \* width + x] = color\_white; // 白色色块像素设置

}

}

}

/\*\* 4.创建Bitmap对象,根据像素数组设置Bitmap每个像素点的颜色值,之后返回Bitmap对象 \*/

Bitmap bitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB\_8888);

bitmap.setPixels(pixels, 0, width, 0, 0, width, height);

return bitmap;

} catch (WriterException e) {

e.printStackTrace();

}

return null;

}

}

该工具类写了两个重载的createQRCodeBitmap方法。其中第一个方法，笔者进行了默认的参数设置，可以满足生成二维码的大部分需求。

createQRCodeBitmap(String content, int width, int height)：传入任意字符串和你想要的二维码图片的宽、高，生成一个Bitmap对象并返回。

createQRCodeBitmap(String content, int width, int height,@Nullable String character\_set, @Nullable String error\_correction, @Nullable String margin,@ColorInt int color\_black, @ColorInt int color\_white)：完整的二维码生成方法，支持自定义配置和自定义样式。

# 时间控件

public class CustomDigitalClock extends android.widget.DigitalClock{

private Calendar mCalendar;

private final static String m12 = "aa hh:mm";//aa hh:mm

private final static String m24 = "HH:mm";//HH:mm

private FormatChangeObserver mFormatChangeObserver;

private Runnable mTicker;

private Handler mHandler;

private boolean mTickerStopped = false;

private String mFormat = m24 ;

public CustomDigitalClock(Context context) {

super(context);

initClock();

}

public CustomDigitalClock(Context context, AttributeSet attrs) {

super(context, attrs);

initClock();

}

private void initClock() {

if (mCalendar == null) {

mCalendar = Calendar.getInstance();

}

// mFormatChangeObserver = new FormatChangeObserver();

// getContext().getContentResolver().registerContentObserver(Settings.System.CONTENT\_URI, true, mFormatChangeObserver);

// setFormat();

}

@Override

protected void onAttachedToWindow() {

mTickerStopped = false;

super.onAttachedToWindow();

mHandler = new Handler();

/\*\*

\* requests a tick on the next hard-second boundary

\*/

mTicker = new Runnable() {

public void run() {

if (mTickerStopped) return;

mCalendar.setTimeZone(TimeZone.getDefault());

mCalendar.setTimeInMillis(System.currentTimeMillis());

setText(new SimpleDateFormat(mFormat).format(mCalendar.getTime()));

invalidate();

long now = SystemClock.uptimeMillis();

long next = now + (1000 - now % 1000);

mHandler.postAtTime(mTicker, next);

}

};

mTicker.run();

}

@Override

protected void onDetachedFromWindow() {

super.onDetachedFromWindow();

mTickerStopped = true;

}

/\*\*

\* Pulls 12/24 mode from system settings

\*/

private boolean get24HourMode() {

return DateFormat.is24HourFormat(getContext());

}

private void setFormat() {

if (get24HourMode()) {

mFormat = m24;

} else {

mFormat = m12;

}

}

private class FormatChangeObserver extends ContentObserver {

public FormatChangeObserver() {

super(new Handler());

}

@Override

public void onChange(boolean selfChange) {

setFormat();

}

}

}

<com.wkllme.wkllmconsumer.widget.CustomDigitalClock  
 android:id="@+id/statusbar\_clock"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="match\_parent"  
 android:layout\_marginRight="20px"  
 android:gravity="center\_vertical"  
 android:singleLine="true"  
 android:textColor="@android:color/white"  
 android:textSize="24px" />

# 手机号码工具类

public class PhoneNumberUtil {

/\*\*

\* 隐藏中间四位数

\*/

public static String hide(String phoneStr) {

String resultStr = "";

if (!TextUtils.isEmpty(phoneStr)) {

if (phoneStr.length() > 7) {

resultStr = phoneStr.substring(0, 3) + "\*\*\*\*" + phoneStr.substring(7);

} else if (phoneStr.length() <= 3) {

resultStr = phoneStr;

} else {

StringBuilder stringBuilder = new StringBuilder();

String three = phoneStr.substring(0, 3);

stringBuilder.append(three);

for (int i = 0; i < phoneStr.substring(3).length(); i++) {

stringBuilder.append("\*");

}

resultStr = stringBuilder.toString();

}

}

return resultStr;

}

/\*\*

\* 判断是否是电话号码

\*/

public static boolean isPhoneNum(String phoneNumStr) {

String num = "[1][34578]\\d{9}";

if (TextUtils.isEmpty(phoneNumStr)) {

return false;

} else {

if (phoneNumStr.matches(num)) {

return true;

} else {

return false;

}

}

}

}

# 打开其他App

## 资料一

有两种方式

### 包名+类型

String packageName = "com.outim.mechat";

String className = packageName + ".ui.activity.AuthorActivity";

Intent intent = new Intent();

intent.setClassName(packageName, className);

intent.putExtra("str", appId);

context.startActivity(intent);

<activity  
 android:name=".ui.activity.AuthorActivity"

android:exported="true" />

跳转到intent加下面的flag，且跳转的Activity加SingleInstance可以保留在当前栈中

intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK | Intent.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS);

android:launchMode="singleInstance"

### URI

Intent intent = new Intent();

intent.setData(Uri.parse("yxoo://page/author?strKey=appId" ));

startActivity(intent);

<activity android:name=".ui.activity.AuthorActivity">  
 <intent-filter>  
 <data  
 android:host="page"  
 android:path="/author"  
 android:scheme="yxoo" />  
 <action android:name="android.intent.action.VIEW" />  
 <category android:name="android.intent.category.DEFAULT" />  
 <category android:name="android.intent.category.BROWSABLE" />  
 </intent-filter>  
</activity>

获取Url中传递的参数val strKey = intent.data.getQueryParameter("strKey")

## 资料二

### 获得手机上已安装应用列表

**PackageManager** packageManager = getPackageManager();

List<**ApplicationInfo**> listApp = packageManager.getInstalledApplications(0);

### 根据猜测含有的字符筛选可能的包名列表

Iterator<**ApplicationInfo**> iterator = listApp.iterator();

List<**String**> listMay = **new** ArrayList<>();

**while** (iterator.hasNext()) {

**String** packna = iterator.next().packageName;

**if** (packna.contains("tao")) {

listMay.add(packna);//com.taobao.taobao

}

}

### 根据包名（Intent）确定Activity名

**Intent** resolveIntent = **new** Intent();

resolveIntent.setPackage(pi.packageName); //如果想看所有的活动名，只配置这一个属性

resolveIntent.setAction(**Intent**.**ACTION\_MAIN)**; //打开主界面需要配置的

resolveIntent.addCategory(**Intent**.**CATEGORY\_LAUNCHER**);

List<**ResolveInfo**> apps = packageManager.queryIntentActivities(resolveIntent, 0);

Iterator<**ResolveInfo**> iterator = apps.iterator();

List<**String**> list = **new** ArrayList<>();

**while** (iterator.hasNext()) {

**String** aaa = iterator.next().activityInfo.name;

list.add(aaa);

}

### 根据包名和类型打开Activity

**ComponentName** cn = **new** ComponentName(packageName, className);

intent.setComponent(cn);

context.startActivity(intent);

### 根据筛选的包名打开应用

**public** **static** **void** **openCLD**(**String** packageName, **Context** context) {

**PackageManager** packageManager = context.getPackageManager();

**PackageInfo** pi = **null**;

**try** {

pi = packageManager.getPackageInfo(packageName, 0);

} **catch** (**NameNotFoundException** e) {

e.printStackTrace();

}

**Intent** resolveIntent = **new** Intent(**Intent**.**ACTION\_MAIN**, **null**);

resolveIntent.addCategory(**Intent**.**CATEGORY\_LAUNCHER**);

resolveIntent.setPackage(pi.packageName);

List<**ResolveInfo**> apps = packageManager.queryIntentActivities(resolveIntent, 0);

**ResolveInfo** ri = apps.iterator().next();

**if** (ri != **null** ) {

**String** className = ri.activityInfo.name;

**Intent** intent = **new** Intent(**Intent**.**ACTION\_MAIN**);

intent.addCategory(**Intent**.**CATEGORY\_LAUNCHER**);

**ComponentName** cn = **new** ComponentName(packageName, className);

intent.setComponent(cn);

context.startActivity(intent);

}

}

### 补充，简单其他应用打开主界面

**Intent** intent = **new** Intent();

intent = packageManager.getLaunchIntentForPackage(packageName);

intent.setFlags(**Intent**.**FLAG\_ACTIVITY\_NEW\_TASK**|**Intent**.**FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED** | **Intent**.**FLAG\_ACTIVITY\_CLEAR\_TOP**) ;

context.startActivity(intent);

### 注意

如果是主Activity，则不用配置将要打开的Activity的exported（共享权限）为true，若不是，则必须指定为true

### 获取应用所有Activity并写入文件

**public** **static** **void** **openCLD**(**String** packageName,**Context** context) {

**PackageManager** packageManager = context.getPackageManager();

**PackageInfo** pi = **null**;

**try** {

pi = packageManager.getPackageInfo(packageName, 0);

} **catch** (**NameNotFoundException** e) {

e.printStackTrace();

}

**Intent** resolveIntent = **new** Intent();

resolveIntent.setPackage(pi.packageName);

List<**ResolveInfo**> apps = packageManager.queryIntentActivities(resolveIntent, 0);

Iterator<**ResolveInfo**> iterator = apps.iterator();

**StringBuilder** builder = **new** StringBuilder();

**while** (iterator.hasNext()) {

**String** acName = iterator.next().activityInfo.name;

builder.append(acName + "\r\n");

}

**try** {

**ToolFile**.*saveToFile*(**ToolFile**.*gainSDCardPath*() + "/" + "pack.txt", builder.toString());

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

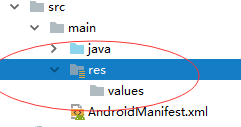
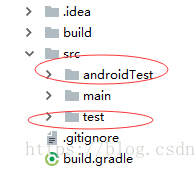
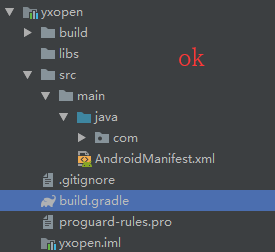
# 打jar包和aar包

## 打jar包

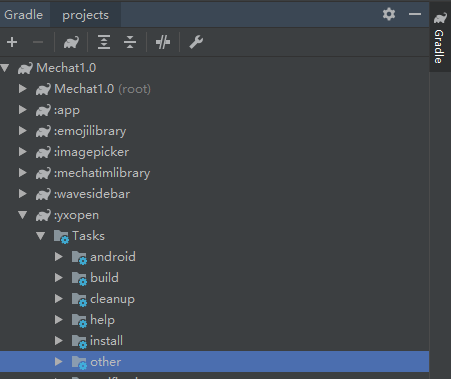
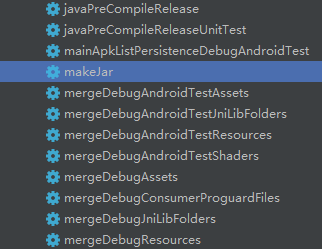
1. 新建modlue，类型为android-library，在该module的gradle中添加红色部分代码

apply plugin: 'com.android.library'  
android {  
 compileSdkVersion 28  
 defaultConfig {  
 minSdkVersion 19  
 targetSdkVersion 28  
 versionCode 1  
 versionName "1.0"  
 testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"  
 }  
 buildTypes {  
 release {  
 minifyEnabled false  
 proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'  
 }  
 }  
}  
dependencies {  
 implementation fileTree(dir: 'libs', include: ['\*.jar'])  
 implementation 'com.android.support:appcompat-v7:28.0.0'  
 testImplementation 'junit:junit:4.12'  
 androidTestImplementation 'com.android.support.test:runner:1.0.2'  
 androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'  
}  
//修改jar名字+将指定jar生成的地方  
task makeJar(type:Copy){  
 //如果之前存在，则先删除  
 delete 'build/libs/yxopen.jar'  
 //设置拷贝的文件  
 from('build/intermediates/packaged-classes/debug/')  
 //生成jar包后的文件目录位置  
 into('build/libs/')  
 //include,exclude参数来设置过滤  
 include('classes.jar')  
 //重命名  
 rename('classes.jar','yxopen.jar')  
}  
makeJar.dependsOn(build)

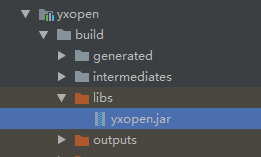
1. 删掉多余的文件夹androidTest、test、res

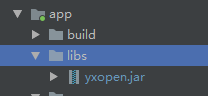
1. 点击 Studio 右侧 gradle -> 你的项目 -> 要打包的module -> Tasks -> other -> makjar -> 双击等待

1. 在module项目下，找到build --> libs --> xxx.jar 就是你的jar包（jar包可改名）



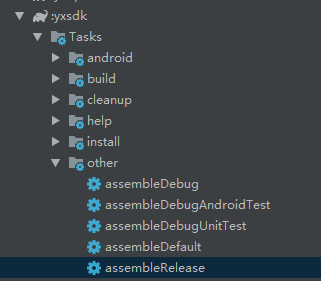
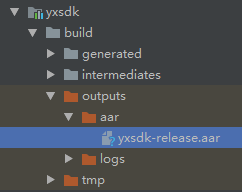
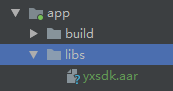
1. 将jar包拷到需要该jar包的module的libs中，gradle中引用implementation files('libs/yxopen.jar')



## 打arr包

1. 新建modlue，类型为android-library
2. 点击 Studio 右侧 gradle -> 你的项目 -> 要打包的module -> Tasks -> other -> assembleRelease -> 双击等待
3. 在module项目下，找到build --> outputs --> aar -> xxx-release.aar 就是你的aar包(aar包可改名)
4. 将aar包拷到需要该aar包的module的libs中，在gradle中引用implementation (name:'yxsdk', ext:'aar'),在项目的gradle中添加红色部分

allprojects {  
 repositories {  
 ...   
 jcenter()  
 google()  
 flatDir {  
 dirs 'libs'  
 }  
 }  
}

aar需要注意

* 资源文件不能同名
* Application需要主APP传递给AAR，AAR再做单例模式
* 扫描模块需要AAR自己初始化
* jar尽量用maven，如果是本地的libs，需要exlude so和provided jar
* AAR可以通过主APP的application初始化自己的数据库
* 轻量数据，可以用intent启动AAR的时候传递
* 如果有基础数据需要提供服务，可以用contentProvider
* butterknife需要统一用8.4.0以上的版本，不然报错
* AAR模块用butterknife需要把资源R改为R2(批量替换)

# 仿微信开放平台授权

# 返回按钮和Home按钮的监听

## Activity中监听返回键

@Override

public void onBackPressed() {

//super.onBackPressed();

}

@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

switch (keyCode) {

case KeyEvent.KEYCODE\_BACK://返回键

Toast.makeText(this,"返回键",Toast.LENGTH\_SHORT).show();

break;

}

return super.onKeyDown(keyCode, event);

}

## Service（Activity）中监听Home键

onKeyDown中有监听Home键的方法，但是你会发现监听起来是无效的，这里其实可以通过广播的形式来监听Home键，不光适用在Service，同样的也可以适用在Activity中，我们新建一个HomeService

public class HomeService extends Service {

//监听Home

private HomeWatcherReceiver mHomeKeyReceiver;

public static final String SYSTEM\_DIALOG\_REASON\_KEY = "reason";

public static final String SYSTEM\_DIALOG\_REASON\_HOME\_KEY = "homekey";

@Override

public IBinder onBind(Intent intent) {

return null;

}

@Override

public void onCreate() {

super.onCreate();

//注册Home监听广播

mHomeKeyReceiver = new HomeWatcherReceiver();

final IntentFilter homeFilter = new IntentFilter(Intent.ACTION\_CLOSE\_SYSTEM\_DIALOGS);

registerReceiver(mHomeKeyReceiver, homeFilter);

}

@Override

public void onDestroy() {

super.onDestroy();

//取消监听

unregisterReceiver(mHomeKeyReceiver);

}

/\*\*

\* 监听Home键

\*/

class HomeWatcherReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

String action = intent.getAction();

if (action.equals(Intent.ACTION\_CLOSE\_SYSTEM\_DIALOGS)) {

String reason = intent.getStringExtra(SYSTEM\_DIALOG\_REASON\_KEY);

if (SYSTEM\_DIALOG\_REASON\_HOME\_KEY.equals(reason)) {

Toast.makeText(context, "Home按键", Toast.LENGTH\_SHORT).show();

}

}

}

}

}

# Home键屏蔽

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.HOME" />

<category android:name="android.intent.category.DEFAULT" />

<category android:name="android.intent.category.LAUNCHER" />

<category android:name="android.intent.category.MONKEY" />

</intent-filter>

# APP开机自动运行

<uses-permission android:name="android.permission.RECEIVE\_BOOT\_COMPLETED" />

<!-- 开机自动启动receiver -->

<receiver android:name="com.example.receiver.BootBroadcastReceiver"

android:enable = "true"

android:exported = "true">

<intent-filter android:priority = "1000">

<action android:name="android.intent.action.BOOT\_COMPLETED" />

</intent-filter>

</receiver>

**public** **class** **BootBroadcastReceiver** **extends** **BroadcastReceiver** {

**static** **final** **String** **ACTION** = "android.intent.action.BOOT\_COMPLETED";

@Override

**public** **void** **onReceive**(**Context** context, **Intent** intent) {

**if** (intent.getAction().equals(**ACTION**)){

**Intent** sayHelloIntent=**new** Intent(context,**MainActivity**.**class**);

sayHelloIntent.addFlags(**Intent**.**FLAG\_ACTIVITY\_NEW\_TASK**);

context.startActivity(sayHelloIntent);

}

}

}

Manifest文件中的 enable 和 exported 以及priority 可有可无，按照上述步骤如果失败，还要检查一下

1. 软件自启动要设置为允许
2. app必须要安装到内部存储，不能安装到sd卡，据说安装到sd卡的话，因为手机启动成功后（发送了启动完成的广播后）才加载sd卡，所以app接收不到广播
3. 程序必须处于启动状态，如果后台杀死了该程序 手机重启会收不到广播 导致无法自启动

# 连点两次退到桌面问题

现象原因

连点两次跳转按钮的时候,第一次后会有很短暂的时间暴露出手机桌面,导致第二次点击会点击到手机桌面上，该问题可能是由于手机版本问题,或者是一些设置的地方没调好,再或者是手机自带的一些优化功能,对界面的跳转做了一些处理,导致第一个界面先隐藏掉,再加载第二个页面,在这期间内,快速点击会点击到手机屏幕

解决方案

主题中设置<item name="android:windowIsTranslucent">false</item>

# 获取版本号、包名等信息

## packageManager

**PackageManager** manager = getPackageManager();

**try** {

**String** packageName = getPackageName();**//应用包名**

**PackageInfo** info = manager.getPackageInfo(packageName, 0);

**String** versionName = info.versionName;**//版本名**

**int** versionCode = info.versionCode;**//版本号**

} **catch** (**NameNotFoundException** e) {

e.printStackTrace();

}

## Build

edt.setText("手机机型：" + android.os.**Build**.**MODEL**

+ "---android系统定制商：" + android.os.**Build**.**BRAND**

+ "---当前开发代号:" + **Build**.**VERSION**.**CODENAME**

+ "---源码控制版本号:" + **Build**.**VERSION**.**INCREMENTAL**

+ "---版本字符串:" + **Build**.**VERSION**.**RELEASE**

+ "---版本号:" + **Build**.**VERSION**.**SDK**

+ "---版本号:" + **Build**.**VERSION**.**SDK\_INT**);

显示： ---手机机型：Redmi Note 2

---android系统定制商：Xiaomi

---当前开发代号:REL

---源码控制版本号:V7.5.3.0.LHMCNDE

---版本字符串:5.0.2

---版本号:21

---版本号:21

**final** **boolean** isKitKat = **Build**.**VERSION**.**SDK\_INT** >= **Build**.**VERSION\_CODES**.**KITKAT**;**//4.4之后改动很大**

**if** (isKitKat) {

//do something

}

## 全部信息工具类

### 工具类

/\*\*

\* 系统扩展类

\*/

public class MyDeviceInfoUtil {

/\*\*

\* 获取硬件识别码

\*/

public static String deviceId(Context context) {

try {

TelephonyManager telephonyManager = (TelephonyManager) context.getSystemService(Context.TELEPHONY\_SERVICE);

return deviceId = telephonyManager.getDeviceId();

} catch (Exception e) {

e.printStackTrace();

return "";

}

return MD5Util.MD5Encode(deviceId, "utf-8");

}

public static String board() {

try {

return Build.BOARD;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String bootloader() {

try {

return Build.BOOTLOADER;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String cpu\_abi() {

try {

return Build.CPU\_ABI;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String cpu\_abi2() {

try {

return Build.CPU\_ABI2;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String device() {

try {

return Build.DEVICE;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String display() {

try {

return Build.DISPLAY;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String hardware() {

try {

return Build.HARDWARE;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String host() {

try {

return Build.HOST;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String id() {

try {

return Build.ID;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String manufacturer() {

try {

return Build.MANUFACTURER;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String getmodel() {

try {

return Build.MODEL;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String product() {

try {

return Build.PRODUCT;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String radio() {

try {

return Build.RADIO;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String tags() {

try {

return Build.TAGS;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String type() {

try {

return Build.TYPE;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String user() {

try {

return Build.USER;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String version\_release() {

try {

return Build.VERSION.RELEASE;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String version\_codename() {

try {

return Build.VERSION.CODENAME;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String version\_incremental() {

try {

return Build.VERSION.INCREMENTAL;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

public static String version\_sdk() {

try {

return Build.VERSION.SDK;

} catch (Exception e) {

e.printStackTrace();

return "";

}

}

}

### 信息Bean类

public class MyDeviceInfo {

private String deviceId;

private String board;

private String bootloader;

private String cpu\_abi;

private String cpu\_abi2;

private String device;

private String display;

private String hardware;

private String host;

private String id;

private String manufacturer;

private String model;

private String product;

private String radio;

private String tags;

private String type;

private String user;

private String version\_release;

private String version\_codename;

private String version\_incremental;

private String version\_sdk;

}

### Bean赋值

MyDeviceInfo request = new MyDeviceInfo();

request.setDeviceId(deviceId());

request.setBoard(board());

request.setDevice(device());

request.setBootloader(bootloader());

request.setCpu\_abi(cpu\_abi());

request.setCpu\_abi2(cpu\_abi2());

request.setDisplay(display());

request.setHardware(hardware());

request.setHost(host());

request.setId(id());

request.setManufacturer(manufacturer());

request.setModel(getmodel());

request.setProduct(product());

request.setRadio(radio());

request.setTags(tags());

request.setType(type());

request.setUser(user());

request.setVersion\_codename(version\_codename());

request.setVersion\_incremental(version\_incremental());

request.setVersion\_release(version\_release());

request.setVersion\_sdk(version\_sdk());

# 判断华为、小米和魅族手机

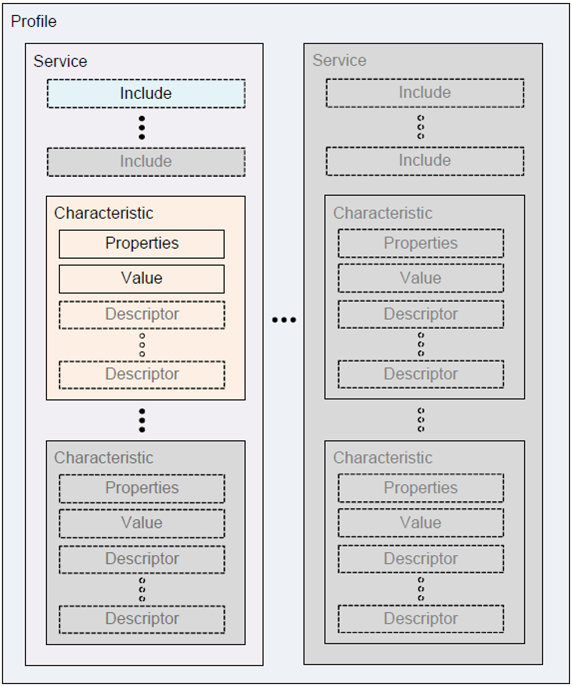
public class WhatPhoneUtil {  
 public static final String SYS\_EMUI = "sys\_emui";  
 public static final String SYS\_MIUI = "sys\_miui";  
 public static final String SYS\_FLYME = "sys\_flyme";  
 public static final String SYS\_OTHER = "sys\_other";  
 private static final String KEY\_MIUI\_VERSION\_CODE = "ro.miui.ui.version.code";  
 private static final String KEY\_MIUI\_VERSION\_NAME = "ro.miui.ui.version.name";  
 private static final String KEY\_MIUI\_INTERNAL\_STORAGE = "ro.miui.internal.storage";  
 private static final String KEY\_EMUI\_API\_LEVEL = "ro.build.hw\_emui\_api\_level";  
 private static final String KEY\_EMUI\_VERSION = "ro.build.version.emui";  
 private static final String KEY\_EMUI\_CONFIG\_HW\_SYS\_VERSION = "ro.confg.hw\_systemversion";  
 public static String whatPhone() {  
 String SYS = SYS\_OTHER;  
 try {  
 Properties prop = new Properties();  
 prop.load(new FileInputStream(new File(Environment.getRootDirectory(), "build.prop")));  
 if (prop.getProperty(KEY\_MIUI\_VERSION\_CODE, null) != null  
 || prop.getProperty(KEY\_MIUI\_VERSION\_NAME, null) != null  
 || prop.getProperty(KEY\_MIUI\_INTERNAL\_STORAGE, null) != null) {  
 SYS = SYS\_MIUI;//小米  
 } else if (prop.getProperty(KEY\_EMUI\_API\_LEVEL, null) != null  
 || prop.getProperty(KEY\_EMUI\_VERSION, null) != null  
 || prop.getProperty(KEY\_EMUI\_CONFIG\_HW\_SYS\_VERSION, null) != null) {  
 SYS = SYS\_EMUI;//华为  
 } else if (getMeizuFlymeOSFlag().toLowerCase().contains("flyme")) {  
 SYS = SYS\_FLYME;//魅族  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 return SYS;  
 }  
 return SYS;  
 }  
 public static String getMeizuFlymeOSFlag() {  
 String result = "";  
 try {  
 Class<?> clz = Class.forName("android.os.SystemProperties");  
 Method get = clz.getMethod("get", String.class, String.class);  
 result = (String) get.invoke(clz, "ro.build.display.id", "");  
 } catch (Exception e) {  
 return result;  
 }  
 return result;  
 }  
}

# BLE 蓝牙开发入门

## 初识低功耗蓝牙

Android 4.3（API Level 18）开始引入Bluetooth Low Energy（BLE，低功耗蓝牙）的核心功能并提供了相应的 API， 应用程序通过这些 API 扫描蓝牙设备、查询 services、读写设备的 characteristics（属性特征）等操作。

Android BLE 使用的蓝牙协议是 GATT 协议，有关该协议的详细内容可以参见蓝牙[官方文档](https://www.bluetooth.com/specifications/generic-attributes-overview)。以下我引用一张官网的图来大概说明 Android 开发中我们需要了解的一些 Bluetooth Low Energy 的专业术语



### Service

一个低功耗蓝牙设备可以定义许多 Service, Service 可以理解为一个功能的集合。设备中每一个不同的 Service 都有一个 128 bit 的 UUID 作为这个 Service 的独立标志。蓝牙核心规范制定了两种不同的UUID，一种是基本的UUID，一种是代替基本UUID的16位UUID。所有的蓝牙技术联盟定义UUID共用了一个基本的UUID：

0x0000xxxx-0000-1000-8000-00805F9B34FB

为了进一步简化基本UUID，每一个蓝牙技术联盟定义的属性有一个唯一的16位UUID，以代替上面的基本UUID的‘x’部分。例如，心率测量特性使用0X2A37作为它的16位UUID，因此它完整的128位UUID为：

0x00002A37-0000-1000-8000-00805F9B34FB

### Characteristic

在 Service 下面，又包括了许多的独立数据项，我们把这些独立的数据项称作 Characteristic。同样的，每一个 Characteristic 也有一个唯一的 UUID 作为标识符。在 Android 开发中，建立蓝牙连接后，我们说的通过蓝牙发送数据给外围设备就是往这些 Characteristic 中的 Value 字段写入数据；外围设备发送数据给手机就是监听这些 Charateristic 中的 Value 字段有没有变化，如果发生了变化，手机的 BLE API 就会收到一个监听的回调

## BLE API 简介

BluetoothAdapter

BluetoothAdapter 拥有基本的蓝牙操作，例如开启蓝牙扫描，使用已知的 MAC 地址 （BluetoothAdapter#getRemoteDevice）实例化一个 BluetoothDevice 用于连接蓝牙设备的操作等等。

BluetoothDevice

代表一个远程蓝牙设备。这个类可以让你连接所代表的蓝牙设备或者获取一些有关它的信息，例如它的名字，地址和绑定状态等等。

BluetoothGatt

这个类提供了 Bluetooth GATT 的基本功能。例如重新连接蓝牙设备，发现蓝牙设备的 Service 等等。

BluetoothGattService

这一个类通过 BluetoothGatt#getService 获得，如果当前服务不可见那么将返回一个 null。这一个类对应上面说过的 Service。我们可以通过这个类的 getCharacteristic(UUID uuid) 进一步获取 Characteristic 实现蓝牙数据的双向传输。

BluetoothGattCharacteristic

这个类对应上面提到的 Characteristic。通过这个类定义需要往外围设备写入的数据和读取外围设备发送过来的数据。

## Android 蓝牙开发示例

### 第一步：声明所需要的权限

<uses-permission android:name="android.permission.BLUETOOTH"/> 使用蓝牙所需要的权限

<uses-permission android:name="android.permission.BLUETOOTH\_ADMIN"/> 使用扫描和设置蓝牙的权限（申明这一个权限必须申明上面一个权限）

在Android5.0之前，是默认申请GPS硬件功能的。而在Android 5.0 之后，需要在manifest 中申明GPS硬件模块功能的使用。

<!-- Needed only if your app targets Android 5.0 (API level 21) or higher. -->

<uses-feature android:name="android.hardware.location.gps" />

在 Android 6.0 及以上，还需要打开位置权限。如果应用没有位置权限，蓝牙扫描功能不能使用（其它蓝牙操作例如连接蓝牙设备和写入数据不受影响）。

<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION"/>

### 第二步：连接蓝牙前的初始化工作

在建立蓝牙连接之前，需要确认设备支持 BLE。如果支持，再确认蓝牙是否开启。如果蓝牙没有开启，可以使用 BLuetoothAdapter 类来开启蓝牙。

#### 获取 BluetoothAdapter

private BluetoothAdapter mBluetoothAdapter;

// Initializes Bluetooth adapter.

final BluetoothManager bluetoothManager = (BluetoothManager) getSystemService(Context.BLUETOOTH\_SERVICE);

mBluetoothAdapter = bluetoothManager.getAdapter();

#### 如果检测到蓝牙没有开启，尝试开启蓝牙

// Ensures Bluetooth is available on the device and it is enabled.

// If not, displays a dialog requesting user permission to enable Bluetooth.

if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {

Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION\_REQUEST\_ENABLE);

startActivityForResult(enableBtIntent, REQUEST\_ENABLE\_BT);

}

### 第三步：扫描蓝牙设备

外围设备开启蓝牙后，会广播出许多的关于该设备的数据信息，例如 mac 地址，uuid 等等。通过这些数据我们可以筛选出需要的设备。

在 BluetoothAdapter 中，我们可以看到有两个扫描蓝牙的方法。第一个方法可以指定只扫描含有特定 UUID Service 的蓝牙设备，第二个方法则是扫描全部蓝牙设备。

boolean startLeScan(UUID[] serviceUuids, BluetoothAdapter.LeScanCallback callback)

boolean startLeScan(BluetoothAdapter.LeScanCallback callback)

#### 开启蓝牙扫描

final BluetoothAdapter.LeScanCallback callback = new BluetoothAdapter.LeScanCallback() {

@Override

public void onLeScan(final BluetoothDevice device, int rssi, byte[] scanRecord) {

bluetoothDeviceArrayList.add(device);

Log.d(TAG, "run: scanning...");

}

};

mBluetoothAdapter.startLeScan(callback);

在 LeScanCallback 回调的方法中，第一个参数是代表蓝牙设备的类，可以通过这个类建立蓝牙连接获取关于这一个设备的一系列详细的参数，例如名字，MAC 地址等等；第二个参数是蓝牙的信号强弱指标，通过蓝牙的信号指标，我们可以大概计算出蓝牙设备离手机的距离。计算公式为：d = 10^((abs(RSSI) - A) / (10 \* n))；第三个参数是蓝牙广播出来的广告数据。

当执行上面的代码之后，一旦发现蓝牙设备，LeScanCallback 就会被回调，直到 stopLeScan 被调用。出现在回调中的设备会重复出现，所以如果我们需要通过 BluetoothDevice 获取外围设备的地址手动过滤掉已经发现的外围设备。

#### 停止蓝牙扫描

void stopLeScan(BluetoothAdapter.LeScanCallback callback)

通过调用 BluetoothAdapter#stopLeScan 可以停止正在进行的蓝牙扫描。这里需要注意的是，传入的回调必须是开启蓝牙扫描时传入的回调，否则蓝牙扫描不会停止。

由于蓝牙扫描的操作比较消耗手机的能量。所以我们不能一直开着蓝牙，必须设置一段时间之后关闭蓝牙扫描。示例代码如下：

private void scanLeDevice(final boolean enable) {

if (enable) {

// Stops scanning after a pre-defined scan period.

// 预先定义停止蓝牙扫描的时间（因为蓝牙扫描需要消耗较多的电量）

mHandler.postDelayed(new Runnable() {

@Override

public void run() {

mScanning = false;

mBluetoothAdapter.stopLeScan(mLeScanCallback);

}

}, SCAN\_PERIOD);

mScanning = true;

// 定义一个回调接口供扫描结束处理

mBluetoothAdapter.startLeScan(mLeScanCallback);

} else {

mScanning = false;

mBluetoothAdapter.stopLeScan(mLeScanCallback);

}

}

### 第四步：连接蓝牙设备

#### 连接

连接蓝牙设备可以通过 BluetoothDevice#ConnectGatt 方法连接，也可以通过 BluetoothGatt#connect 方法进行重新连接。以下分别是两个方法的官方说明：

BluetoothGatt connectGatt(Context context, boolean autoConnect, BluetoothGattCallback callback)

boolean connect()

* connectGatt第二个参数表示是否需要自动连接。如果设置为 true, 表示如果设备断开了，会不断的尝试自动连接。设置为 false 表示只进行一次连接尝试。
* connectGatt第三个参数是连接后进行的一系列操作的回调，例如连接和断开连接的回调，发现服务的回调，成功写入数据，成功读取数据的回调等等。
* connect相当与调用 connectGatt 且第二个参数 autoConnect 设置为 true。

#### 连接成功回调

当调用蓝牙的连接方法之后，蓝牙会异步执行蓝牙连接的操作，如果连接成功会回调 BluetoothGattCalbackl#onConnectionStateChange 方法。这个方法运行的线程是一个 Binder 线程，所以不建议直接在这个线程处理耗时的任务，因为这可能导致蓝牙相关的线程被阻塞。

void onConnectionStateChange(BluetoothGatt gatt, int status, int newState)

* 这一个方法有三个参数，第一个就蓝牙设备的 Gatt 服务连接类。
* 第二个参数代表是否成功执行了连接操作，如果为 BluetoothGatt.GATT\_SUCCESS 表示成功执行连接操作，第三个参数才有效，否则说明这次连接尝试不成功。有时候，我们会遇到 status == 133 的情况，根据网上大部分人的说法，这是因为 Android 最多支持连接 6 到 7 个左右的蓝牙设备，如果超出了这个数量就无法再连接了。所以当我们断开蓝牙设备的连接时，还必须调用 BluetoothGatt#close 方法释放连接资源。否则，在多次尝试连接蓝牙设备之后很快就会超出这一个限制，导致出现这一个错误再也无法连接蓝牙设备。
* 第三个参数代表当前设备的连接状态，如果 newState == BluetoothProfile.STATE\_CONNECTED 说明设备已经连接，可以进行下一步的操作了（发现蓝牙服务，也就是 Service）。当蓝牙设备断开连接时，这一个方法也会被回调，其中的 newState == BluetoothProfile.STATE\_DISCONNECTED。

### 第五步：发现服务

在成功连接到蓝牙设备之后才能进行这一个步骤，也就是说在 BluetoothGattCalbackl#onConnectionStateChang 方法被成功回调且表示成功连接之后调用 BluetoothGatt#discoverService 这一个方法。当这一个方法被调用之后，系统会异步执行发现服务的过程，直到 BluetoothGattCallback#onServicesDiscovered 被系统回调之后，手机设备和蓝牙设备才算是真正建立了可通信的连接。

到这一步，我们已经成功和蓝牙设备建立了可通信的连接，接下来就可以执行相应的蓝牙通信操作了，例如写入数据，读取蓝牙设备的数据等等

#### 读取数据

当我们发现服务之后就可以通过 BluetoothGatt#getService 获取 BluetoothGattService，接着通过 BluetoothGattService#getCharactristic 获取 BluetoothGattCharactristic。

通过 BluetoothGattCharactristic#readCharacteristic 方法可以通知系统去读取特定的数据。如果系统读取到了蓝牙设备发送过来的数据就会调用 BluetoothGattCallback#onCharacteristicRead 方法。通过 BluetoothGattCharacteristic#getValue 可以读取到蓝牙设备的数据。以下是代码示例：

//通过 BluetoothGattCharactristic#readCharacteristic通知系统读取特定数据，系统读取到蓝牙设备发送过来的数据后的回调

@Override

public void onCharacteristicRead(final BluetoothGatt gatt,

final BluetoothGattCharacteristic characteristic,

final int status) {

Log.d(TAG, "callback characteristic read status " + status + " in thread " + Thread.currentThread());

if (status == BluetoothGatt.GATT\_SUCCESS) {

Log.d(TAG, "read value: " + characteristic.getValue());

}

}

// 读取数据

BluetoothGattService service = gattt.getService(SERVICE\_UUID);

BluetoothGattCharacteristic characteristic = gatt.getCharacteristic(CHARACTER\_UUID);

gatt.readCharacteristic();

#### 写入数据

和读取数据一样，在执行写入数据前需要获取到 BluetoothGattCharactristic。接着执行一下步骤：

1. 调用 BluetoothGattCharactristic#setValue 传入需要写入的数据（蓝牙最多单次1支持 20 个字节数据的传输，如果需要传输的数据大于这一个字节则需要分包传输）。
2. 调用 BluetoothGattCharactristic#writeCharacteristic 方法通知系统异步往设备写入数据。
3. 系统回调 BluetoothGattCallback#onCharacteristicWrite 方法通知数据已经完成写入。此时，我们需要执行 BluetoothGattCharactristic#getValue 方法检查一下写入的数据是否我们需要发送的数据，如果不是按照项目的需要判断是否需要重发。以下是示例代码：

//通过BluetoothGattCharactristic#writeCharacteristic 通知系统写入特定数据，系统往蓝牙设备写入数据后的回调

@Override

public void onCharacteristicWrite(final BluetoothGatt gatt,

final BluetoothGattCharacteristic characteristic,

final int status) {

Log.d(TAG, "callback characteristic write in thread " + Thread.currentThread());

if(!characteristic.getValue().equal(sendValue)) {

// 执行重发策略

gatt.writeCharacteristic(characteristic);

}

}

//往蓝牙数据通道的写入数据

BluetoothGattService service = gattt.getService(SERVICE\_UUID);

BluetoothGattCharacteristic characteristic = gatt.getCharacteristic(CHARACTER\_UUID);

characteristic.setValue(sendValue);

gatt.writeCharacteristic(characteristic);

#### 向蓝牙设备注册监听实现实时读取蓝牙设备的数据

BLE app通常需要获取设备中characteristic 变化的通知。下面的代码演示了怎么为一个Characteristic 设置一个监听。

mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);

BluetoothGattDescriptor descriptor = characteristic.getDescriptor(

UUID.fromString(SampleGattAttributes.CLIENT\_CHARACTERISTIC\_CONFIG));

descriptor.setValue(BluetoothGattDescriptor.ENABLE\_NOTIFICATION\_VALUE);

mBluetoothGatt.writeDescriptor(descriptor);

值 得注意的是，除了通过 BluetoothGatt#setCharacteristicNotification 开启 Android 端接收通知的开关，还需要往 Characteristic 的 Descriptor 属性写入开启通知的数据开关使得当硬件的数据改变时，主动往手机发送数据。

### 最后一步：断开连接

当我们连接蓝牙设备完成一系列的蓝牙操作之后就可以断开蓝牙设备的连接了。通过 BluetoothGatt#disconnect 可以断开正在连接的蓝牙设备。当这一个方法被调用之后，系统会异步回调 BluetoothGattCallback#onConnectionStateChange 方法。通过这个方法的 newState 参数可以判断是连接成功还是断开成功的回调。

由于 Android 蓝牙连接设备的资源有限，当我们执行断开蓝牙操作之后必须执行 BluetoothGatt#close 方法释放资源。需要注意的是通过 BluetoothGatt#close 方法也可以执行断开蓝牙的操作，不过 BluetoothGattCallback#onConnectionStateChange 将不会收到任何回调。此时如果执行 BluetoothGatt#connect 方法会得到一个蓝牙 API 的空指针异常。所以，我们推荐的写法是当蓝牙成功连接之后，通过 BluetoothGatt#disconnect 断开蓝牙的连接，紧接着在 BluetoothGattCallback#onConnectionStateChange 执行 BluetoothGatt#close 方法释放资源。

以下是代码示例：

@Override

public void onConnectionStateChange(final BluetoothGatt gatt, final int status, final int newState) {

Log.d(TAG, "onConnectionStateChange: thread " + Thread.currentThread() + " status " + newState);

if (status != BluetoothGatt.GATT\_SUCCESS) {

String err = "Cannot connect device with error status: " + status;

// 当尝试连接失败的时候调用 disconnect 方法是不会引起这个方法回调的，所以这里直接回调就可以了。

gatt.close();

Log.e(TAG, err);

return;

}

if (newState == BluetoothProfile.STATE\_CONNECTED) {

gatt.discoverService();

} else if (newState == BluetoothProfile.STATE\_DISCONNECTED) {

gatt.close();

}

}

## 蓝牙操作的注意事项

* 蓝牙的写入操作( 包括 Descriptor 的写入操作), 读取操作必须序列化进行. 写入数据和读取数据是不能同时进行的, 如果调用了写入数据的方法, 马上调用又调用写入数据或者读取数据的方法,第二次调用的方法会立即返回 false, 代表当前无法进行操作. 详情可以参考 [蓝牙读写操作返回 false，为什么多次读写只有一次回调？](http://a1anwang.com/post-18.html)
* Android 连接外围设备的数量有限，当不需要连接蓝牙设备的时候，必须调用 BluetoothGatt#close 方法释放资源
* 蓝牙 API 连接蓝牙设备的超时时间大概在 20s 左右，具体时间看系统实现。有时候某些设备进行蓝牙连接的时间会很长，大概十多秒。如果自己手动设置了连接超时时间（例如通过 Handler#postDelay 设置了 5s 后没有进入 BluetoothGattCallback#onConnectionStateChange 就执行 BluetoothGatt#close 操作强制释放断开连接释放资源）在某些设备上可能会导致接下来几次的连接尝试都会在 BluetoothGattCallback#onConnectionStateChange 返回 state == 133。另外可以参考[Android 中 BLE 连接出现“BluetoothGatt status 133”的解决方法](http://www.loverobots.cn/android-ble-connection-solution-bluetoothgatt-status-133.html)
* 所有的蓝牙操作使用 Handler 固定在一条线程操作，这样能省去很多因为线程不同步导致的麻烦

最后，附上我写的一个使用 RxJava 封装蓝牙操作库<https://github.com/Belolme/RxBLE>