【eoeAndroid 特刊】

第六期：数据通信-成为 Android 数据流大师

发布版本： Ver 1.0.0(build 2009.07.26)

本篇简介

作为 Google 的手机，网络功能自然是其标榜的卖点，G1 上市以后，与相配合的网络功能也备受大家好评，可以说 Android 是离不开网络的，很多功能也都依靠网络才能完成。

Android 不仅自身具备了丰富的网络功能，同时也为我们的开发人员提供了丰富的接口。作为 Android 的开发人员来说，熟悉、并使用这些接口，无疑能为我们的应用增色不少。

但是使用网络应用，就必须对 Android 平台的数据通信做一定的了解，如果利用 SD 卡扩展我们的内存，如果利用蓝牙、wifi 来连通数据，如何使用 http 数据流，如果我们的开发人员能够熟练掌握这些技能，一定能做出更好的应用程序。

本期的特刊也只是一个引子，希望有兴趣的朋友，能够以这篇特刊为契机，拓宽自己的这方面的学习。

本期主要包含如下四方面的内容:

1. SD 的介绍、模拟器中模拟、代码中的使用
2. 蓝牙的使用、底层蓝牙的分析
3. Wifi 在 SDK 中相关的内容
4. http 协议的简单介绍等

这期结尾还有对 eoe 特刊小组成立的简单介绍，详细信息请访问

http://www.eoeandroid.com/viewthread.php?tid=1784

《Google Android 开发入门与实战》

- 作者：靳岩 姚尚朗
- 出版社：人民邮电出版社
- 书号：9787115209306
- 出版日期：2009 年 7 月
- 开本：16 开
- 页码：400

内容特点

- 国内第一本原创 Android 图书
- 完全基于 Android 最新的 SDK1.5
- 全书除了大量小型案例之外还包含了 5 个 Android 平台下的完整商业实例及源码分析，分别是 RSS 阅读器、基于 GoogleMap 的个人 GPS、豆瓣客户端、在线音乐播放器、手机信息助手
- 随书附赠的光盘中包含 300 分钟的详细教学视频以及 Android 开发必备的开发资源
- 读者对于此书内容的疑问可以访问 http://www.eoeandroid.com 社区，作者团队将会及时解答
- 马上订购！ 进入讨论区 试读下载

活动发起地址：

【eoeAndroid 特刊】策划 第六期：数据通信-成为 Android 数据流大师

http://www.eoeandroid.com/viewthread.php?tid=1421&extra=page%3D1

致谢：

本期特刊中的内容由以下几位朋友贡献，让我们对他们共享出的内容表示由衷得感谢

- leftmoon
- Iceskysl(http://iceskysl.1ster.com)
- Haiyangjy (http://haiyangjy.com)
- kenshin
- 游利卡(http://www.cnblogs.com/pcedb0189/)

目录

eoeMarket 中国第一款第三方软件发布平台

eoeAndroid 作为中国最棒的 Android 开发社区，已经受到了大家广泛的支持，而 eoeAndroid 推出的 eoeMarket 是为广大开发者，还有中国玩家提供了更加方便，更加好用的一个软件发布的平台。

首先，eoeMarket 目前是免费的，大家可以省去 Google Market 上注册繁琐的步骤，也不用担心申请无法通过之后，带来的信用卡锁定的麻烦。

其次，eoeMarket 是针对中国玩家、开发者的，是在中国法律保护下的软件平台。并能提供最好的本地化服务，一旦出现任何问题，都可以迅速得到解决，不再会有语言的障碍。

再次，eoeMarket 可以作为大家发布软件的一个很好的缓冲地。任何发布到 Android Market 上的产品，都可以先发布到 eoeMarket 上，首先得到充分的反馈，进行调整后发布，一定能在 Android Market 获得更好的反响和效益。

所以，eoeMarket 是所有我们广大开发者一个绝好的选择平台。

不仅是这些，eoeMarket 还有具有更加灵活的架构，可以根据我们不同的需求来进行调整。eoeMarket 甚至也可以做为一个软件或者其他产品的集中解决方案。非常适合作为国内个性化手机生产的一个集中的软件解决方案！

eoeMarket 的未来，将会继续为 eoeAndroid 社区的会员提供更好的服务，给于中国的开发者提供更好的渠道，我们的目标就是做中国 Android 开发者的门户，成为中国开发者应用发布的首选入口，让中国 Android 开发者可以在 Android 平台上真正淘到金子！

eoeAndroid 欢迎广大手机厂商，开源社区、玩家社区与我们合作，共同推进中国 Android 事业的发展，分享 Android 事业的成果

更多信息，尽情登陆 eoeAndroid 社区 eoeMarket 专区
http://www.eoeandroid.com/forumdisplay.php?fid=56

1. SD 卡概述及深层次介绍(google 来的)

By leftmoon

在这一部分中，准确的说，SD 卡应该是对外部存储卡的一个笼统的称呼，所以，如果你用的是 TF 卡，没关系，一样的。

SD 卡（Secure Digital Memory Card）是一种基于半导体快闪记忆器的新一代记忆设备。SD 卡由日本松下、东芝及美国 SanDisk 公司于 1999 年 8 月共同开发研制。大小犹如一张邮票的 SD 记忆卡，重量只有 2 克，但却拥有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性。

SD 卡在 24mm×32mm×2.1mm 的体积内结合了 SanDisk 快闪记忆卡控制与 MLC（Multilevel Cell）技术和 Toshiba（东芝）0.16u 及 0.13u 的 NAND 技术，通过 9 针的接口界面与专门的驱动器相连接，不需要额外的电源来保持其上记忆的信息。而且它是一体化固体介质，没有任何移动部分，所以不用担心机械运动的损坏。

以下是 SD 卡的特点：

High transfer rate for fast copying and downloading（高速复制与下载）

Large storage capacity, up to 2GB（大容量存储，最高可至 2G）

Built to last, with an operating shock rating of 2,000Gs, equivalent to a ten-foot drop**（防震设计，运行震动频率可达 2000 高斯，其震动强度相当于从 10 英尺的高度扔下，即 3.048 米。）

2.在模拟器如何创建、挂载、使用 SD 卡

By leftmoon

在 Android 开发包的 tools 目录下就有文件叫 mksdcard.exe，该命令是用来生成一个文件镜像，该文件可以供模拟器

模拟 SD 卡

我们可以通过创建一个磁盘镜像，并且在模拟器启动的时候装载进去，这样，在模拟器中，我们就可以使用 SD 卡了。

需要注意的是，我们只能通过启动模拟器的时候装载 SD 卡，在运行的时候不能移除 SD 卡（文档原文：you can not remove a simulated SD card from a running emulator，

不过，模拟器运行的时候可以通过设置-SD 卡-卸载 SD 卡来移除）。当 SD 卡装载后，就可以浏览 SD 卡的文件，也可以进行删除、复制操作。

模拟器支持 SDHC 卡，因此，我们可以创建超过 128M 的镜像文件

2.1 创建镜像文件

创建方式有两种 1

使用 android 命令

android create avd -n <avd_name> -t <targetID> -c <size>[K|M]

-c 选项表示创建 AVD 的时候，同时创建一个镜像文件

使用 mksdcard 命令

mksdcard <size> <file>，如

mksdcard 1024M E:¥sdcard1.iso

创建一个 1G 的 SD 卡镜像文件

2.2 装载 SD 卡

emulator -sdcard <filepath>，如：

emulator -sdcard E:¥sdcard.iso

2.3 复制文件

复制文件的命令为

adb push <local> <remote>

复制一个文件/目录到模拟器上

adb pull <remote> <local>

从模拟器上复制一个文件/目录到指定目录

如：将 aapt.exe 复制到 SD 卡上

E:\android-sdk\tools>adb push aapt.exe /sdcard

1303 KB/s (0 bytes in 11342941.008s)

将 SD 卡根目录下的 aapt.exe 文件复制到当前目录下，并且重命名为 a.exe

E:\android-sdk\tools>adb pull /sdcard/aapt.exe a.exe

1492 KB/s (0 bytes in 11342941.007s)

需要注意的是，目前模拟器还存在一些 BUG，复制中文名的文件会导致出错。

2.4 删除文件

删除文件需要使用 shell 命令，这个不作具体说明了，仅给出例子

E:\android-sdk\tools>adb shell

cd /sdcard

cd /sdcard

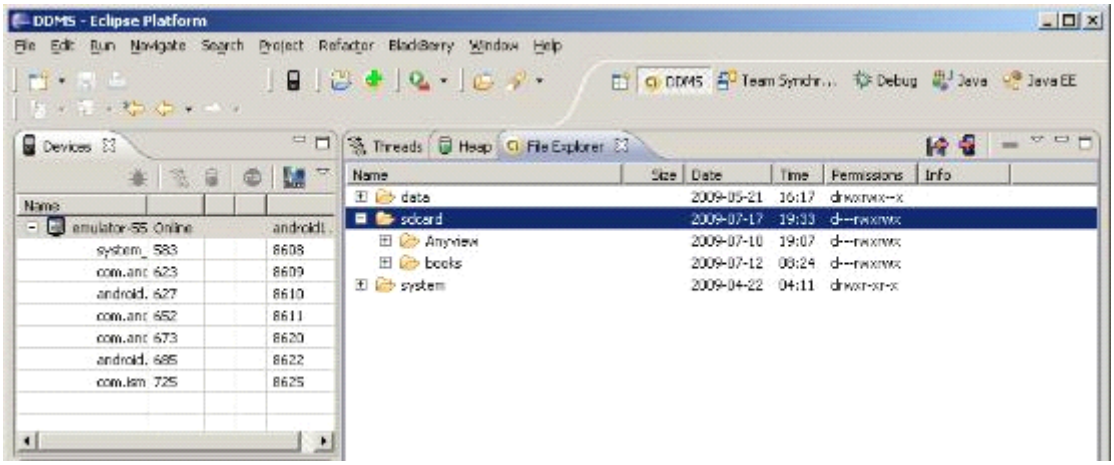
rm aapt.exe

rm aapt.exe

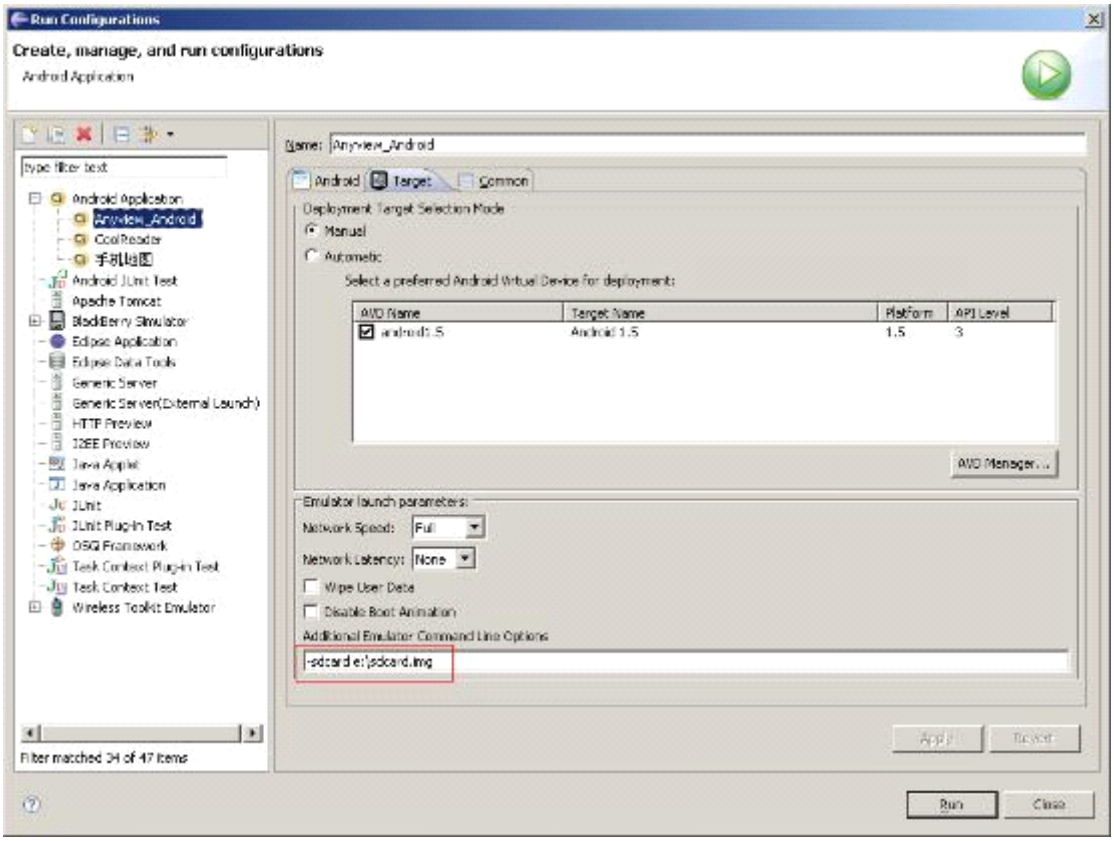
#

2.5eclispse

如果你对以上命令行操作深恶痛绝的话，更简单的方式是使用 Eclipse 中的 DDMS 来管理



在 Eclipse 中，如果需要在启动时自动加载，可以在 run 中指定参数，如图：



3.在代码中如何使用 SD 卡

By leftmoon

3.1SD 是否可用检查

有两种方式，一种是写一个监听函数，用来监听

```
private final BroadcastReceiver broadcastRec = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
                if (intent.getAction().equals("android.intent.action.MEDIA_MOUNTED")) { //SD 卡已经成功挂载
                        //有 SD 卡
                } else if (intent.getAction().equals("android.intent.action.MEDIA_REMOVED")
                                        ||
intent.getAction().equals("android.intent.action.ACTION_MEDIA_UNMOUNTED")
                                        ||
intent.getAction().equals("android.intent.action.ACTION_MEDIA_BAD_REMOVAL")) {//各种未挂载状态
                        //无 SD 卡
                        }                }        };
```

另一种在我看来，更简单一些：

```
File f = new File("/sdcard/");
f.exists();
```

3.2 卡上数据的读写

好了，看完了上面的，现在是不是想动手了？

在动手之前，我们先来看看，要对 SD 卡进行读写操作，Android 为我们准备了哪些类/接口。

接口：FileFilter

类：File、FileInputStream、FileOutputStream

下面我们依次对上面出场的做个简单的介绍吧，同时，也会给出若干代码来说明各个类如何使用

3.3 File

文件系统的一个抽象描述类，可以使用相对路径和绝对路径，该类提供了若干方法查询/修改文件信息

File file = new File("/sdcard/a.txt");

判断文件是否存在：

file.exists()

“文件”是目录还是文件

file.isDirectory()

如果文件为目录，如何列出子目录呢？

File[] files = file.listFiles();

对于文件，如果获得文件的一些信息

file.length(); //文件的长度

file.canRead(); //文件可以读吗

file.canWrite(); //文件可以写吗

file.delete(); //删除该文件

下面，给出一些片断代码来展示如何使用 File 类

//创建一个新文件

File file = new File("/sdcard/a.txt");

if (!file.exists()) {

try {

file.createNewFile();

```
} catch (IOException e) {

e.printStackTrace();

} }

//创建一个新文件夹

File file = new File("/sdcard/a/b/c");

if (!file.exists()) {

try {

file.mkdirs();

}
```

请注意，mkdirs()和 mkdir()的用法，mkdirs()表示，如果需要，会先创建上层目录，如上述代码中，如果 SD 卡根目录不存在目录 a，那么，使用 mkdir()会抛出异常

3.4 FileFilter

我们知道，listFiles()会列出当前目录下的文件，但是，另一个问题出来了，如果当前目录下文件杂乱无部分，而我们又不想列出所有文件，怎么办？FileFilter 提供了一个解决方法。我们只需要实现 public abstract boolean accept (File pathname)接口就可以了。

```
FileFilter filter = new FileFilter() {

        public boolean accept (File file)  {

                if (file.isFile() && file.getAbsolutePath().toLowerCase().endsWith(".txt")) {

                        return true;

                }

                return false;

        }

};

File[] files = file.listFiles(filter);
```

以上代码展示了如何只列出当前目录下的 TXT 文件。

3.5 FileInputStream

我们不仅需要对文件进行操作，还需要对文件的内容进行操作，这时候，FileInputStream 就登场了

FileInputStream 继承自 InputStream，但是 FileInputStream 使用了缓冲，以提高频繁读入数据时的性能。

使用 FileInputStream 也很简单

```
FileInputStream fis = new FileInputStream("/sdcard/a.txt");

byte[] abytes = new byte[1024]

int len = fis.read(abytes);
```

在上述代码中，我们打开了 SD 卡根目录下的 a.txt 文件，同时，读入 1K 的数据，需要注意的是，read 并不能保证一定读入期望长度的数据，我们需要对读入数据的长度进行检查。

3.6 FileOutputStream

FileOutputStream 与 OutputStream 的用法差不多，在些也不多说，给出一个代码吧：

```
FileOutputStream fos = new FileOutputStream("/sdcard/a.txt");

fos.write("Hello World!".getBytes());

fos.flush();
```

上述代码会向 a.txt 中写入 Hello World 字符，请注意，FileOutputStream 也使用了缓冲，因此，数据不会立即写入文件中，当系统认为需要写回数据的时候，真正的写回才发生，因此，如果为了让数据立即写回，需要使用 flush() 方法。

也许你会问到，我不想把文件原有的内容抹到，怎么？这个也简单，

```
FileOutputStream fos = new FileOutputStream("/sdcard/a.txt", true);
```

true 表示向文件后面追加数据

4.蓝牙简介

所谓蓝牙(Bluetooth)技术，实际上是一种短距离无线电技术，利用“蓝牙”技术，能够有效地简化掌上电脑、笔记本电脑和移动电话手机等移动通信终端设备之间的通信，也能够成功地简化以上这些设备与因特网 Internet 之间的通信，从而使这些现代通信设备与因特网之间的数据传输变得更加迅速高效，为无线通信拓宽道路。蓝牙采用分散式网络结构以及快跳频和短包技术，支持点对点及点对多点通信，工作在全球通用的 2.4GHz ISM（即工业、科学、医学）频段。其数据速率为 1Mbps。采用时分双工传输方案实现全双工传输。

蓝牙名称的来源

蓝牙 这个名称来自于第十世纪的一位丹麦国王 Harald Blatand , Blatand 在英文里的意思可以被解释为 Bluetooth(蓝牙)因为国王喜欢吃蓝梅，牙龈每天都是蓝色的所以叫蓝牙

更多信息请 Google 搜索：http://www.google.cn/search?hl=zh-CN&q=蓝牙&btnG=Google+搜索

5.蓝牙设置种常用的 Intent

By haiyangjy 收集

下面是在 bluetoothsettings.java 中注册蓝牙 Intent 的函数：

```
private boolean initBluetoothAPI() {

mIntentFilter =

// 跟远端蓝牙设备连接上时返回来的 intent

new IntentFilter(BluetoothIntent.REMOTE_DEVICE_CONNECTED_ACTION);

// 跟远端蓝牙设备断开时返回来的 intent
mIntentFilter.addAction(BluetoothIntent.REMOTE_DEVICE_DISCONNECTED_ACTION);

// 跟远端的蓝牙设备配对上时收到的 intent，不过前提是对方主动发起的配对

// 才能收到这个 intent

mIntentFilter.addAction(BluetoothIntent.BONDING_CREATED_ACTION);

// 本地蓝牙设备可用时收到的 Intent

mIntentFilter.addAction(BluetoothIntent.ENABLED_ACTION);

// 本地蓝牙设备不可用时收到的 Intent

mIntentFilter.addAction(BluetoothIntent.DISABLED_ACTION);

// 扫描到远端设备时收到的 intent mIntentFilter.addAction(BluetoothIntent.REMOTE_DEVICE_FOUND_AC
TION);

// 远端蓝牙设备消失时收到的 intent mIntentFilter.addAction(BluetoothIntent.REMOTE_DEVICE_DISAP
PEARED_ACTION);

// 远端蓝牙设备名称更换时收到的 intent，因为刚发现设备的时候还没有获取// 它的名称

mIntentFilter.addAction(BluetoothIntent.REMOTE_NAME_UPDATED_ACTION);
```

```
// 当有远端设备发起配对请求时收到的 intent
mIntentFilter.addAction(BluetoothIntent.PAIRING_REQUEST_ACTION);
// 蓝牙耳机状态改变时候到的 intent
mIntentFilter.addAction(BluetoothIntent.HEADSET_STATE_CHANGED_ACTION);
// 扫描设备结束
    mIntentFilter.addAction(BluetoothIntent.DISCOVERY_COMPLETED_ACTION);
// 扫描开始
mIntentFilter.addAction(BluetoothIntent.DISCOVERY_STARTED_ACTION);
// 蓝牙设备模式改变，表示本地蓝牙设备是否可以被查找
mIntentFilter.addAction(BluetoothIntent.MODE_CHANGED_ACTION);
// 有耳机插入
mIntentFilter.addAction(Intent.ACTION_HEADSET_PLUG);
```

5.1 启动蓝牙

在启动蓝牙的时候，要注意的地方是不能正常启动蓝牙的情况，因为正常启动的时候会返回 BluetoothIntent.ENABLED_ACTION 这个 Intent，当时当启动出现异常的时候是没有 Intent 返回的，android 使用回调函数来解决这个问题。

下面是在 bluetoothdeviceservice.java 里面 enable（(IBluetoothDeviceCallback callback） 的过程：

```
public synchronized boolean enable(IBluetoothDeviceCallback callback) {
        checkPermissionBluetoothAdmin();
        Log.d(TAG,"start enable! ");
        // Airplane mode can prevent Bluetooth radio from being turned on.
        if (mIsAirplaneSensitive && isAirplaneModeOn()) {
            return false;
        }
        if (mIsEnabled) {
            return false;
        }
        if (mEnableThread != null && mEnableThread.isAlive()) {
            return false;          }
    //    主要的启动过程是放在一个新起的线程里面，但是不管能不能启动
    //     仍然返回了 true
        mEnableThread = new EnableThread(callback);
        mEnableThread.start();
       //
        return true;          }

    private EnableThread mEnableThread;
    private class EnableThread extends Thread {
        private final IBluetoothDeviceCallback mEnableCallback;
        public EnableThread(IBluetoothDeviceCallback callback) {
```

```
            mEnableCallback = callback;          }

        public void run() {

            boolean res = enableNative();

            if (res) {                  mEventLoop.start();              }

            if (mEnableCallback != null) {

                try {                    // 通过回调函数来表明是否正常启动蓝牙设备

                    mEnableCallback.onEnableResult(res ?

                                        BluetoothDevice.RESULT_SUCCESS :

                                        BluetoothDevice.RESULT_FAILURE);

                } catch (RemoteException e) {}              }

            if (res) {                  mIsEnabled = true;

                                        mIsDiscovering = false;

                                        Intent intent = new Intent(BluetoothIntent.ENABLED_ACTION);

                                        mContext.sendBroadcast(intent);                  }

            }else{

                mIsEnabled = false;

                mIsDiscovering = false;

            }

            mEnableThread = null;

        }       } // 这个回调函数将被作为参数传进 bluetoothservice 里面的 enable（IBluetoothDeviceCallback
callback）

static class DeviceCallback extends IBluetoothDeviceCallback.Stub {

        Handler messageHandler;
```

```
        public void setHandler(Handler handler) {
            synchronized (this) {
                messageHandler = handler;
            }

        public void onEnableResult(int result) {
            switch(result) {

  //  启动不成功的时候执行
                case BluetoothDevice.RESULT_FAILURE:
 messageHandler.sendMessage(messageHandler.obtainMessage(EVENT_FAILED_BT_ENABLE,0));
                    break;
            }         }

//   配对完成时执行
public void onCreateBondingResult(String address, int result) {
            synchronized (this) {
                if (messageHandler != null) {
                    if (result == BluetoothDevice.RESULT_FAILURE) {
                        messageHandler.sendMessage(messageHandler.obtainMessage(
                                HANDLE_PAIRING_FAILED, address));
                    } else {
                        messageHandler.sendMessage(messageHandler.obtainMessage(
```

```
                        HANDLE_PAIRING_PASSED, address));

                }               }           }       }       };
```

5.2 关闭过程

```
public synchronized boolean disable() {

        checkPermissionBluetoothAdmin();



        if (mEnableThread != null && mEnableThread.isAlive()) {

            return false;

        }

        if (!mIsEnabled) {

            return true;

        }

        if(!disableNative()){

            Log.d(TAG,"disableNative false ");

          return false;

        }
```

```
        mEventLoop.stop();
        mIsEnabled = false;
        mIsDiscovering = false;
        Intent intent = new Intent(BluetoothIntent.DISABLED_ACTION);
        mContext.sendBroadcast(intent);
        return true;
    }
```

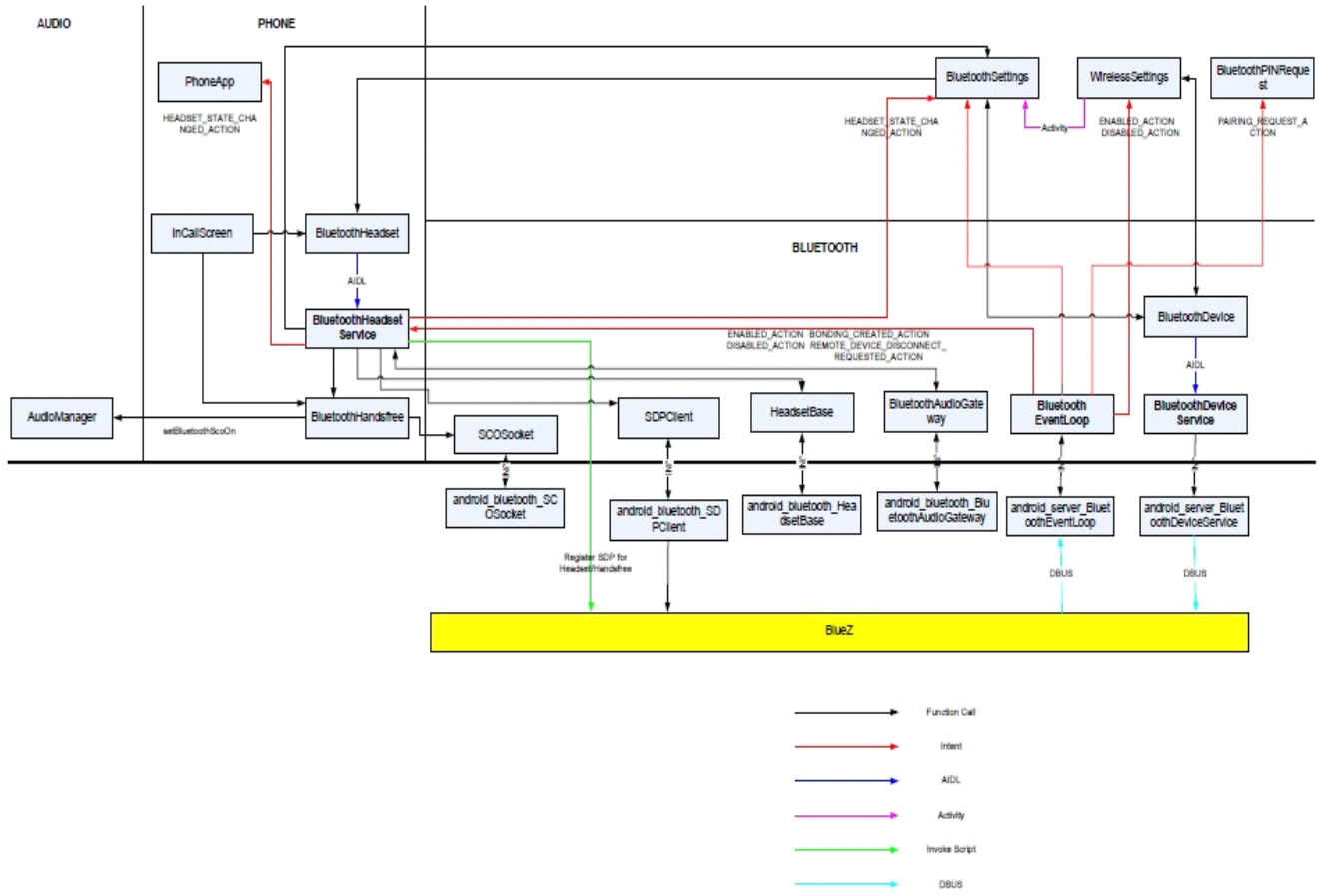
5.3 配对过程

```
private void doPair(Preference pref, String address) {
        pref.setEnabled(false);
        pref.setSummary(STR_PAIRING);
        if (mPinEdit != null){
            String strPIN = mPinEdit.getText().toString();

            mBluetooth.writePinCode(address, strPIN);
            mBluetooth.createBonding(address, sDeviceCallback);
        }    }
```

最后根据配对的结果执行回调函数的 onCreateBondingResult。

6.蓝牙模块



在 SystemServer 启动的时候，会生成一个 BluetoothDeviceService 的实例，

```
// Skip Bluetooth if we have an emulator kernel

// TODO: Use a more reliable check to see if this product should

// support Bluetooth - see bug 988521

if (SystemProperties.get("ro.kernel.qemu").equals("1")) {

Log.i(TAG, "Registering null Bluetooth Service (emulator)");
```

```
ServiceManager.addService(Context.BLUETOOTH_SERVICE, null);
} else if (factoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL) {
Log.i(TAG, "Registering null Bluetooth Service (factory test)");
ServiceManager.addService(Context.BLUETOOTH_SERVICE, null);
} else {
Log.i(TAG, "Starting Bluetooth Service.");
bluetooth = new BluetoothDeviceService(context);
bluetooth.init();
ServiceManager.addService(Context.BLUETOOTH_SERVICE, bluetooth);
int bluetoothOn = Settings.System.getInt(mContentResolver,
Settings.System.BLUETOOTH_ON, 0);
if (bluetoothOn > 0) {
bluetooth.enable(null);
}
}
```

BluetoothDeviceService 会生成一个 BluetoothEventLoop 实例，它们两者均通过 DBUS 来和 BlueZ 通信。BluetoothDeviceService 是通过 DBUS 向 BlueZ 发送命令，而命令的返回结果则是由 BlueZ 通过 DBUS 传回给 BluetoothEventLoop 的（具体交互请参见 BlueZ 的 dbus_api.txt ） ,BlueZ 也会通过 DBUS 向 BluetoothEventLoop 发送一些事件通知。

BluetoothEventLoop 和外部的接口是通过预先定义的 Intent，

初始的时候蓝牙是没有使能的，要通过 BluetoothSettings 或者 WirelessSettings 来打开蓝牙设备，然后通过 BluetoothSettings 去查找附近的其他蓝牙设备，找到后可以建立 RFCOMM 连

接和配对。

6.1 蓝牙耳机

Android 实现了对 Headset 和 Handsfree 两种 profile 的支持。其实现核心是 BluetoothHeadsetService，在 PhoneApp 创建的时候会启动它。

```
if (getSystemService(Context.BLUETOOTH_SERVICE) != null) {
mBtHandsfree = new BluetoothHandsfree(this, phone);
startService(new Intent(this, BluetoothHeadsetService.class));
} else {
// Device is not bluetooth capable
mBtHandsfree = null;
}
```

BluetoothHeadsetService 通过接收 ENABLED_ACTION、BONDING_CREATED_ACTION 、DISABLED_ACTION 和 REMOTE_DEVICE_DISCONNECT_REQUESTEDACTION 来改变状态，它也会监听 Phone 的状态变化。

```
IntentFilter filter = new IntentFilter(BluetoothIntent.BONDING_CREATED_ACTION);
filter.addAction(BluetoothIntent.REMOTE_DEVICE_DISCONNECT_REQUESTED_ACTION);
filter.addAction(BluetoothIntent.ENABLED_ACTION);
filter.addAction(BluetoothIntent.DISABLED_ACTION);
registerReceiver(mBluetoothIntentReceiver, filter);
mPhone.registerForPhoneStateChanged(mStateChangeHandler,
```

PHONE_STATE_CHANGED, null);

BluetoothHeadsetService 收到 ENABLED_ACTION 时，会先向 BlueZ 注册 Headset 和 Handsfree 两种 profile （ 通过执行 sdptool 来实现的， 均作为 Audio Gateway ）， 然后让 BluetoothAudioGateway 接收 RFCOMM 连接，让 BluetoothHandsfree 接收 SCO 连接（这些操作都是为了让蓝牙耳机能主动连上 Android）。

```
if (action.equals(BluetoothIntent.ENABLED_ACTION)) {
// SDP server may not be ready, so wait 3 seconds before
// registering records.
// TODO: Use a different mechanism to register SDP records,
// that actually ACK's on success, so that we can retry rather
// than hardcoding a 3 second guess.
mHandler.sendMessageDelayed(mHandler.obtainMessage(REGISTER_SDP_
RECORDS),3000);
mAg.start(mIncomingConnectionHandler);
mBtHandsfree.onBluetoothEnabled();
}
```

BluetoothHeadsetService 收到 DISABLED_ACTION 时，会停止 BluetoothAudioGateway 和 BluetoothHandsfree。

```
if (action.equals(BluetoothIntent.DISABLED_ACTION)) {
```

```
mBtHandsfree.onBluetoothDisabled();

mAg.stop(); }
```

6.2Android 跟蓝牙耳机建立连接有两种方式。

1. Android 主动跟蓝牙耳机连

BluetoothSettings 中和蓝牙耳机配对上之后， BluetoothHeadsetService 会收到 BONDING_CREATED_ACTION，这个时候 BluetoothHeadsetService 会主动去和蓝牙耳机建立 RFCOMM 连接。

```
if (action.equals(BluetoothIntent.BONDING_CREATED_ACTION)) {

if (mState == BluetoothHeadset.STATE_DISCONNECTED) {

// Lets try and initiate an RFCOMM connection

try {

mBinder.connectHeadset(address, null);

} catch (RemoteException e) {}

} }
```

RFCOMM 连接的真正实现是在 ConnectionThread 中，它分两步，第一步先通过 SDPClient 查询蓝牙设备时候支持 Headset 和 Handsfree profile。

```
// 1) SDP query

SDPClient client = SDPClient.getSDPClient(address);

if (DBG) log("Connecting to SDP server (" + address + ")...");

if (!client.connectSDPAsync()) {
```

```
Log.e(TAG, "Failed to start SDP connection to " + address);

mConnectingStatusHandler.obtainMessage(SDP_ERROR).sendToTarget();

client.disconnectSDP();

return;

}

if (isInterrupted()) {

client.disconnectSDP();

return;

}

if (!client.waitForSDPAsyncConnect(20000)) { // 20 secs

if (DBG) log("Failed to make SDP connection to " + address);

mConnectingStatusHandler.obtainMessage(SDP_ERROR).sendToTarget();

client.disconnectSDP();

return;

}

if (DBG) log("SDP server connected (" + address + ")");

int headsetChannel = client.isHeadset();

if (DBG) log("headset channel = " + headsetChannel);

int handsfreeChannel = client.isHandsfree();

if (DBG) log("handsfree channel = " + handsfreeChannel);

client.disconnectSDP();
```

第二步才是去真正建立 RFCOMM 连接。

```
// 2) RFCOMM connect

mHeadset = new HeadsetBase(mBluetooth, address, channel);

if (isInterrupted()) {

return;

}

int result = mHeadset.waitForAsyncConnect(20000, // 20 secs

mConnectedStatusHandler);

if (DBG) log("Headset RFCOMM connection attempt took " +

(System.currentTimeMillis() - timestamp) + " ms");

if (isInterrupted()) {

return;

}

if (result < 0) {

Log.e(TAG, "mHeadset.waitForAsyncConnect() error: " + result);

mConnectingStatusHandler.obtainMessage(RFCOMM_ERROR).sendToTarget();

return;

} else if (result == 0) {

Log.e(TAG, "mHeadset.waitForAsyncConnect() error: " + result +

"(timeout)");

mConnectingStatusHandler.obtainMessage(RFCOMM_ERROR).sendToTarget();

return;

} else {
```

```
if (DBG) log("mHeadset.waitForAsyncConnect() success");
mConnectingStatusHandler.obtainMessage(RFCOMM_CONNECTED).sendToTarget();
}
```

当 RFCOMM 连接成功建立后，BluetoothHeadsetDevice 会收到 RFCOMM_CONNECTED 消息，它会调用 BluetoothHandsfree 来建立 SCO 连接，广播通知 Headset 状态变化的 Intent（PhoneApp 和 BluetoothSettings 会接收这个 Intent）。

```
case RFCOMM_CONNECTED:
// success
if (DBG) log("Rfcomm connected");
if (mConnectThread != null) {
try {
mConnectThread.join();
} catch (InterruptedException e) {
Log.w(TAG, "Connect attempt cancelled, ignoring
RFCOMM_CONNECTED", e);
return;
}
mConnectThread = null;
}
setState(BluetoothHeadset.STATE_CONNECTED,
BluetoothHeadset.RESULT_SUCCESS);
mBtHandsfree.connectHeadset(mHeadset, mHeadsetType);
```

```
break;
```

BluetoothHandsfree 会先做一些初始化工作，比如根据是 Headset 还是 Handsfree 初始化不同的 ATParser，并且启动一个接收线程从已建立的 RFCOMM 上接收蓝牙耳机过来的控制命令（也就是 AT 命令），接着判断如果是在打电话过程中，才去建立 SCO 连接来打通数据通道。

```
/* package */ void connectHeadset(HeadsetBase headset, int headsetType) {
mHeadset = headset;
mHeadsetType = headsetType;
if (mHeadsetType == TYPE_HEADSET) {
initializeHeadsetAtParser();
} else {
initializeHandsfreeAtParser();
}
headset.startEventThread();
configAudioParameters();
if (inDebug()) {
startDebug();
}
if (isIncallAudio()) {
audioOn();
```

```
}}
```

建立 SCO 连接是通过 SCOSocket 实现的

```
/** Request to establish SCO (audio) connection to bluetooth
* headset/handsfree, if one is connected. Does not block.
* Returns false if the user has requested audio off, or if there
* is some other immediate problem that will prevent BT audio.
*/
/* package */ synchronized boolean audioOn() {
mOutgoingSco = createScoSocket();
if (!mOutgoingSco.connect(mHeadset.getAddress())) {
mOutgoingSco = null;
}
return true;
}
```

当 SCO 连接成功建立后，BluetoothHandsfree 会收到 SCO_CONNECTED 消息，它就会去调用 AudioManager 的 setBluetoothScoOn 函数，从而通知音频系统有个蓝牙耳机可用了。到此，Android 完成了和蓝牙耳机的全部连接。

```
case SCO_CONNECTED:
if (msg.arg1 == ScoSocket.STATE_CONNECTED && isHeadsetConnected()
&&
mConnectedSco == null) {
```

```
if (DBG) log("Routing audio for outgoing SCO conection");

mConnectedSco = (ScoSocket)msg.obj;

mAudioManager.setBluetoothScoOn(true);

} else if (msg.arg1 == ScoSocket.STATE_CONNECTED) {

if (DBG) log("Rejecting new connected outgoing SCO socket");

((ScoSocket)msg.obj).close();

mOutgoingSco.close();

}

mOutgoingSco = null;

break;
```

2. 蓝牙耳机主动跟 Android 连

首先 BluetoothAudioGateway 会在一个线程中收到来自蓝牙耳机的 RFCOMM 连接，然后发送消息给 BluetoothHeadsetService。

```
mConnectingHeadsetRfcommChannel = -1;

mConnectingHandsfreeRfcommChannel = -1;

if

(waitForHandsfreeConnectNative(SELECT_WAIT_TIMEOUT) == false) {

if (mTimeoutRemainingMs > 0) {

try {

Log.i(tag, "select thread timed out, but " +

mTimeoutRemainingMs + "ms of

waiting remain.");
```

```
Thread.sleep(mTimeoutRemainingMs);

} catch (InterruptedException e) {

Log.i(tag, "select thread was interrupted (2),

exiting");

mInterrupted = true;

}       }       }
```

BluetoothHeadsetService 会根据当前的状态来处理消息，分 3 种情况，第一是当前状态是非连接状态，会发送 RFCOMM_CONNECTED 消息，后续处理请参见前面的分析。

```
case BluetoothHeadset.STATE_DISCONNECTED:

// headset connecting us, lets join

setState(BluetoothHeadset.STATE_CONNECTING);

mHeadsetAddress = info.mAddress;

mHeadset = new HeadsetBase(mBluetooth, mHeadsetAddress,

info.mSocketFd,

info.mRfcommChan,

mConnectedStatusHandler);

mHeadsetType = type;

mConnectingStatusHandler.obtainMessage(RFCOMM_CONNECTED).sendToTarget();

break;
```

如果当前是正在连接状态， 则先停掉已经存在的 ConnectThread，并直接调用 BluetoothHandsfree 去建立 SCO 连接。

```
case BluetoothHeadset.STATE_CONNECTING:
```

```
// If we are here, we are in danger of a race condition
// incoming rfcomm connection, but we are also attempting an
// outgoing connection. Lets try and interrupt the outgoing
// connection.
mConnectThread.interrupt();
// Now continue with new connection, including calling callback
mHeadset = new HeadsetBase(mBluetooth, mHeadsetAddress,
info.mSocketFd,
info.mRfcommChan,
mConnectedStatusHandler);
mHeadsetType = type;
setState(BluetoothHeadset.STATE_CONNECTED,
BluetoothHeadset.RESULT_SUCCESS);
mBtHandsfree.connectHeadset(mHeadset, mHeadsetType);
// Make sure that old outgoing connect thread is dead.
break;
```

如果当前是已连接的状态，这种情况是一种错误 case，所以直接断掉所有连接。

```
case BluetoothHeadset.STATE_CONNECTED:
if (DBG) log("Already connected to " + mHeadsetAddress + ", disconnecting
" +
info.mAddress);
```

```
mBluetooth.disconnectRemoteDeviceAcl(info.mAddress);

break;
```

蓝 牙 耳 机 也可能会主动发起 SCO 连接， BluetoothHandsfree 会接收到一个 SCO_ACCEPTED 消息，它会去调用 AudioManager 的 setBluetoothScoOn 函数，从而通知音频系统有个蓝牙耳机可用了。到此，蓝牙耳机完成了和 Android 的全部连接。

```
case SCO_ACCEPTED:

if (msg.arg1 == ScoSocket.STATE_CONNECTED) {

if (isHeadsetConnected() && mAudioPossible && mConnectedSco ==

null) {

Log.i(TAG, "Routing audio for incoming SCO connection");

mConnectedSco = (ScoSocket)msg.obj;

mAudioManager.setBluetoothScoOn(true);

} else {

Log.i(TAG, "Rejecting incoming SCO connection");

((ScoSocket)msg.obj).close();

}

} // else error trying to accept, try again

mIncomingSco = createScoSocket();

mIncomingSco.accept();

break;
```

7.Wi-Fi 简介

Wi-Fi 是一个无线网路通信技术的品牌，由 Wi-Fi 联盟(Wi-Fi Alliance)所持有。目的是改善基于 IEEE 802.11 标准的无线网路产品之间的互通性。

现时一般人会把 Wi-Fi 及 IEEE 802.11 混为一谈。甚至把 Wi-Fi 等同于无线网际网路。

Wi-Fi 联盟成立于 1999 年，当时的名称叫做 Wireless Ethernet Compatibility Alliance (WECA)。在 2002 年 10 月，正式改名为 Wi-Fi Alliance。

通俗说法：

WIFI 就是一种无线联网的技术，以前通过网络连接电脑，而现在则是通过无线电波来连网；常见的就是一个无线路由器，那么在这个无线路由器的电波覆盖的有效范围都可以采用 WIFI 连接方式进行联网，如果无线路由器连接了一条 ADSL 线路或者别的上网线路，则又被称为“热点”。

现在市面上上常见的无线路由器多为 54M 速度，再上一个等级就是 108M 的速度，当然这个速度并不是你上互联网的速度，上互联网的速度主要是取决于 WIFI 热点的互联网线路。

更多资料可以 Google 搜索 wifi

http://www.google.cn/search?hl=zh-CN&q=wifi&btnG=Google+搜索

简单来说，wifi 就是一种数据连通的方式，就像我们用普通网线联网一样。形象一点，我们要到一个地方，我们必须选择一种交通工具，是选择汽车、火车，还是其他，wifi 就相当于我们的一种交通工具而已。

Wifi 的应用领域很广，现在绝大部分的笔记本，还有很多电子产品上都内置了 wifi 模块。这些都不是很复杂的应用，如果有兴趣把自己的机器拆了，就可以很轻松得看到里面的 wifi 模块，只是很简单的一小块芯片。

对于 Android 平台来说，wifi 是一种选择的方式，因为 wifi 的速度更快，所以直接使用 wifi 进行数据连通对于我们来说更加方便，但同时，因为手机还承担了接打电话还有手法短信的一些工作，所以如何处理好 wifi 还有其他程序之间的协调也是我们要做的事情。

另外 SDK 中也提供了一些现成的 API，可以让我们直接来进行一些数据通信的处理

8.Android SDK 中 Android.net.wifi 简要分析

By 游利卡

Android 已经帮助我们做好了一些可以直接调用的类
打开 andriod.net.wifi 我们可以看到 Android 提供的几个类。

游利卡的建议是，在看这一部分之前，最好能用一下 PC 上的 wifi，这样对我们的理解会很有帮助！

这里列了很多，但是大致可以分为四个主要的类 ScanResult wifiConfiguration WifiInfo WifiManager

(1)ScanResult，主要是通过 wifi 硬件的扫描来获取一些周边的 wifi 热点的信息。

在我们进行 wifi 搜索的时候，一般会搜到这些信息，首先是接入点名字、接入点信息的强弱、还有接入点使用的安全模式，是 WPA、WPE。

打开这个类，我们可以看到以下几个信息

BSSID	接入点的地址，这里主要是指小范围几个无线设备相连接所获取的地址，比如说两台笔记本通过无线网卡进行连接，双方的无线网卡分配的地址
SSID	网络的名字，当我们搜索一个网络时，就是靠这个来区分每个不同的网络接入点
Capabilities	网络接入的性能，这里主要是来判断网络的加密方式等
Frequency	频率，每一个频道交互的 MHz 数
Level	等级，主要来判断网络连接的优先数。

这里只提供了一个方法，就是将获得信息编程字符串 toString()

(2)wifiConfiguration 在我们连通一个 wifi 接入点的时候，需要获取到的一些信息。大家可以跟我们有线的设备进行对比一下。

这里的数据相对来说比较复杂一下

六个子类

WifiConfiguration.AuthAlgorthm	用来判断加密方法
WifiConfiguration.GroupCipher	获取使用 GroupCipher 的方法来进行加密
WifiConfiguration.KeyMgmt	获取使用 KeyMgmt 进行
WifiConfiguration.PairwiseCipher	获取使用 WPA 方式的加密
WifiConfiguration.Protocol	获取使用哪一种协议进行加密
wifiConfiguration.Status	获取当前网络的状态

对于上述加密感兴趣的朋友，可以在网上搜索相关的内容。

(3)WifiInfo 在我们的 wifi 已经连通了以后，可以通过这个类获得一些已经连通的 wifi 连接的信息

获取当前链接的信息，这里信息就比较简单了，
这里简单介绍一下这里的方法:

getBSSID()	获取 BSSID
getDetailedStateOf()	获取客户端的连通性，
getHiddenSSID()	获得 SSID 是否被隐藏
getIpAddress()	获取 IP 地址
getLinkSpeed()	获得连接的速度
getMacAddress()	获得 Mac 地址
getRssi()	获得 802.11n 网络的信号
getSSID()	获得 SSID
getSupplicanState()	返回具体客户端状态的信息

(4)wifiManager 这个不用说，就是用来管理我们的 wifi 连接，这里已经定义好了一些类，可以供我们使用

这里来说相对复杂，里面的内容比较多，但是通过字面意思，我们还是可以获得很多相关的信息。这个类里面预先定义了许多常量，我们可以直接使用，不用再次创建。

这里还是简单介绍一下这里的方法

addNetwork(WifiConfiguration config)	通过获取到的网络的链接状态信息，来添加网络
calculateSignalLevel(int rssi , int numLevels)	计算信号的等级
compareSignalLevel(int rssiA, int rssiB)	对比连接 A 和连接 B
createWifiLock(int lockType, String tag)	创建一个 wifi 锁，锁定当前的 wifi 连接
disableNetwork(int netId)	让一个网络连接失效
disconnect()	断开连接
enableNetwork(int netId, Boolean disableOthers)	连接一个连接
getConfiguredNetworks()	获取网络连接的状态
getConnectionInfo()	获取当前连接的信息
getDhcpInfo()	获取 DHCP 的信息
getScanResulats()	获取扫描测试的结果
getWifiState()	获取一个 wifi 接入点是否有效
isWifiEnabled()	判断一个 wifi 连接是否有效
pingSupplicant()	ping 一个连接，判断是否能连通
ressociate()	即便连接没有准备好，也要连通
reconnect()	如果连接准备好了，连通
removeNetwork()	移除某一个网络
saveConfiguration()	保留一个配置信息

setWifiEnabled()	让一个连接有效
startScan()	开始扫描
updateNetwork(WifiConfiguration config)	更新一个网络连接的信息

此外 wifiManaer 还提供了一个内部的子类，也就是 wifiManagerLock

WifiManagerLock 的作用是这样的，在普通的状态下，如果我们的 wifi 的状态处于闲置，那么网络的连通，将会暂时中断。但是如果我们把当前的网络状态锁上，那么 wifi 连通将会保持在一定状态，当然接触锁定之后，就会恢复常态。

9.在 Android 中扫描 wifi 热点演示实例教程

1、首先新建了布局模板 XML 文件 vifi.xml,代码很简单，如下：

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.          android:orientation="vertical" android:layout_width="fill_parent"
4.          android:layout_height="fill_parent">
5.
6.          <TextView android:id="@+id/wifi"
7.          android:layout_width="fill_parent"
8.          android:layout_height="wrap_content"
9.          android:text="@string/hello" />
10.
11. </LinearLayout>
```

复制代码

2、写 java 代码，新建个 Activity，代码如下：

```
1.  package com.eoeandroid.demo.testcode;
2.
3.  import java.util.List;
4.
5.  import android.app.Activity;
6.  import android.content.BroadcastReceiver;
7.  import android.content.Context;
8.  import android.content.Intent;
9.  import android.content.IntentFilter;
10. import android.net.wifi.ScanResult;
11. import android.net.wifi.WifiManager; (这里引入两个 wifi 的类)
12. import android.os.Bundle;
```

```
13.  import android.view.Menu;
14.  import android.view.MenuItem;
15.  import android.widget.TextView;
16.
17.  public class WifiTester extends Activity {
18.          TextView mainText;
19.          WifiManager mainWifi;    (请注意这里,我这里就是我们调用的 WifiManager 类)
20.          WifiReceiver receiverWifi;
21.          List<ScanResult> wifiList;
22.          StringBuilder sb = new StringBuilder();
23.
24.          public void onCreate(Bundle savedInstanceState) {
25.                  super.onCreate(savedInstanceState);
26.                  setContentView(R.layout.vifi);
27.                  setTitle("eoe 教程: Wifi Test.  -by:IceskYsl");
28.                  mainText = (TextView) findViewById(R.id.wifi);
29.                  mainWifi = (WifiManager) getSystemService(Context.WIFI_SERVICE);(初始化 WifiManager)
30.
31.
32.                  receiverWifi = new WifiReceiver();
33.                  registerReceiver(receiverWifi, new IntentFilter(
34.                                  WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
35.                  mainWifi.startScan();(开始扫描 wifi 网络)
36.                  mainText.setText("\nStarting Scan...\n");
37.          }
38.
39.          public boolean onCreateOptionsMenu(Menu menu) {
40.                  menu.add(0, 0, 0, "Refresh");
41.                  return super.onCreateOptionsMenu(menu);
42.          }
43.
```

```
44.          public boolean onMenuItemSelected(int featureId, MenuItem item) {
45.                  mainWifi.startScan();(开始扫描 wifi 网络)
46.                  mainText.setText("Starting Scan");
47.                  return super.onMenuItemSelected(featureId, item);
48.          }
49.
50.          protected void onPause() {
51.                  unregisterReceiver(receiverWifi);(注销 Wifi)
52.                  super.onPause();
53.          }
54.
55.          protected void onResume() {
56.                  registerReceiver(receiverWifi, new IntentFilter(
57.                                  WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
58.                   (注册 Wifi)
59.                  super.onResume();
60.          }
61.
62.          class WifiReceiver extends BroadcastReceiver {
63.          // 这里 wifiReciver 继承了 BroadcastReceiver ,使用了标准的 BroadcastReceiver 方法
64.                  public void onReceive(Context c, Intent intent) {
65.                          sb = new StringBuilder();
66.                          wifiList = mainWifi.getScanResults();
67.                          for (int i = 0; i < wifiList.size(); i++) {
68.                                  sb.append(new Integer(i + 1).toString() + ".");
69.                                  sb.append((wifiList.get(i)).toString());
70.                                  sb.append("\n\n");
71.                          }
72.                          mainText.setText(sb);
73.                  }
74.          }
```

```
75.  }
```

复制代码

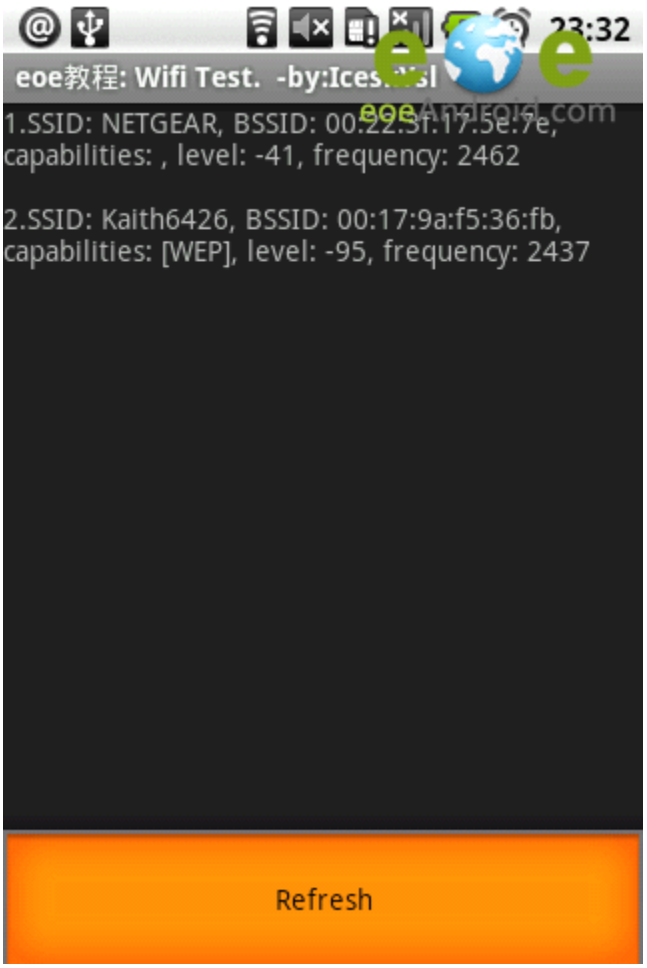
3、申请相关权限，代码如下：

```
1.  <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"></uses-permission>
2.  <uses-permission android:name="android.permission.ACCESS_CHECKIN_PROPERTIES"></uses-permission>
3.  <uses-permission android:name="android.permission.WAKE_LOCK"></uses-permission>
4.  <uses-permission android:name="android.permission.INTERNET"></uses-permission>
5.  <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"></uses-permission>
6.  <uses-permission android:name="android.permission.MODIFY_PHONE_STATE"></uses-permission>
```

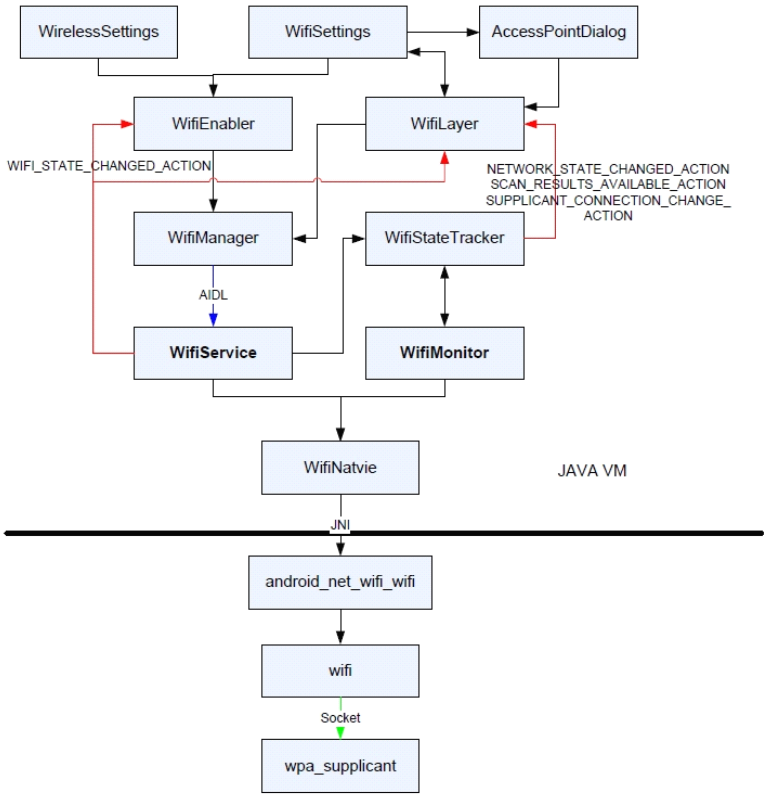
复制代码

4、OK，就这些，允允许下，效果看附件图片，我扫描到我这有 2 个热点。

其他的大家自己发挥，不赘述了 。



10. wifi 模块分析



10.1 初始化

在 SystemServer 启动的时候，会生成一个ConnectivityService 的实例，

```
try {
Log.i(TAG, "Starting Connectivity Service.");
ServiceManager.addService(Context.CONNECTIVITY_SERVICE, new
ConnectivityService(context));
} catch (Throwable e) {
Log.e(TAG, "Failure starting Connectivity Service", e);
}
```

ConnectivityService 的构造函数会创建WifiService，

```
if (DBG) Log.v(TAG, "Starting Wifi Service.");
mWifiStateTracker = new WifiStateTracker(context, handler);
WifiService wifiService = new WifiService(context, mWifiStateTracker);
ServiceManager.addService(Context.WIFI_SERVICE, wifiService);
```

WifiStateTracker 会创建WifiMonitor 接收来自底层的事件，WifiService 和WifiMonitor 是整

个模块的核心。WifiService 负责启动关闭wpa_supplicant、启动关闭WifiMonitor 监视线程和把命令下发给wpa_supplicant，而WifiMonitor 则负责从wpa_supplicant 接收事件通知。

10.2 连接 AP

1. 使能 WIFI

WirelessSettings 在初始化的时候配置了由WifiEnabler 来处理Wifi 按钮，

```
private void initToggles() {
mWifiEnabler = new WifiEnabler(
this,
(WifiManager) getSystemService(WIFI_SERVICE),
(CheckBoxPreference) findPreference(KEY_TOGGLE_WIFI));
```

当用户按下Wifi 按钮后，Android 会调用WifiEnabler 的onPreferenceChange，再由WifiEnabler 调用WifiManager 的setWifiEnabled 接口函数，通过AIDL，实际调用的是WifiService 的 setWifiEnabled 函数，WifiService 接着向自身发送一条MESSAGE_ENABLE_WIFI 消息，在处理该消息的代码中做真正的使能工作：首先装载WIFI 内核模块（该模块的位置硬编码为 "/system/lib/modules/wlan.ko" ）， 然后启动wpa_supplicant （ 配置文件硬编码为 "/data/misc/wifi/wpa_supplicant.conf"），再通过WifiStateTracker 来启动WifiMonitor 中的监视线程。

```
private boolean setWifiEnabledBlocking(boolean enable) {
final int eventualWifiState = enable ? WIFI_STATE_ENABLED :
WIFI_STATE_DISABLED;
updateWifiState(enable ? WIFI_STATE_ENABLING : WIFI_STATE_DISABLING);
if (enable) {
if (!WifiNative.loadDriver()) {
Log.e(TAG, "Failed to load Wi-Fi driver.");
updateWifiState(WIFI_STATE_UNKNOWN);
return false;
}
if (!WifiNative.startSupplicant()) {
WifiNative.unloadDriver();
Log.e(TAG, "Failed to start supplicant daemon.");
updateWifiState(WIFI_STATE_UNKNOWN);
return false;
}
mWifiStateTracker.startEventLoop();
}
// Success!
persistWifiEnabled(enable);
updateWifiState(eventualWifiState);
```

```
return true;
}
```

当使能成功后，会广播发送WIFI_STATE_CHANGED_ACTION 这个Intent 通知外界WIFI 已经成功使能了。WifiEnabler 创建的时候就会向Android 注册接收 WIFI_STATE_CHANGED_ACTION，因此它会收到该Intent，从而开始扫描。

```
private void handleWifiStateChanged(int wifiState) {
if (wifiState == WIFI_STATE_ENABLED) {
loadConfiguredAccessPoints();
attemptScan();
}
```

10.3 查找 AP

扫描的入口函数是WifiService 的startScan，它其实也就是往wpa_supplicant 发送SCAN 命令。

```
static jboolean android_net_wifi_scanCommand(JNIEnv* env, jobject clazz)
{
jboolean result;
// Ignore any error from setting the scan mode.
// The scan will still work.
(void)doBooleanCommand("DRIVER SCAN-ACTIVE", "OK");
result = doBooleanCommand("SCAN", "OK");
(void)doBooleanCommand("DRIVER SCAN-PASSIVE", "OK");
return result;
}
```

当wpa_supplicant 处理完SCAN 命令后，它会向控制通道发送事件通知扫描完成，从而 wifi_wait_for_event 函数会接收到该事件，由此WifiMonitor 中的MonitorThread 会被执行来出来这个事件，

```
void handleEvent(int event, String remainder) {
case SCAN_RESULTS:
mWifiStateTracker.notifyScanResultsAvailable();
break;
```

WifiStateTracker 则接着广播发送SCAN_RESULTS_AVAILABLE_ACTION 这个Intent

```
case EVENT_SCAN_RESULTS_AVAILABLE:
mContext.sendBroadcast(new
Intent(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
```

WifiLayer 注册了接收SCAN_RESULTS_AVAILABLE_ACTION 这个Intent，所以它的相关处理函数handleScanResultsAvailable 会被调用，在该函数中，先会去拿到SCAN 的结果（最终是往wpa_supplicant 发送SCAN_RESULT 命令并读取返回值来实现的），

List<ScanResult> list = mWifiManager.getScanResults();
对每一个扫描返回的AP，WifiLayer 会调用WifiSettings 的onAccessPointSetChanged 函数，从而最终把该AP 加到GUI 显示列表中。

```
public void onAccessPointSetChanged(AccessPointState ap, boolean added) {
AccessPointPreference pref = mAps.get(ap);
if (added) {
if (pref == null) {
pref = new AccessPointPreference(this, ap);
mAps.put(ap, pref);
} else {
pref.setEnabled(true);
}
mApCategory.addPreference(pref);
}}
```

10.4 配置 AP 参数

当用户在 WifiSettings 界面上选择了一个AP 后，会显示配置AP 参数的一个对话框，

```
public boolean onPreferenceTreeClick(PreferenceScreen preferenceScreen, Preference
preference) {
if (preference instanceof AccessPointPreference) {
AccessPointState state = ((AccessPointPreference)
preference).getAccessPointState();
showAccessPointDialog(state, AccessPointDialog.MODE_INFO);
}}
```

10.5 连接

当用户在 AcessPointDialog 中选择好加密方式和输入密钥之后，再点击连接按钮，Android 就会去连接这个AP。

```
private void handleConnect() {
String password = getEnteredPassword();
if (!TextUtils.isEmpty(password)) {
mState.setPassword(password);
}
mWifiLayer.connectToNetwork(mState);
}
```

WifiLayer 会先检测这个AP 是不是之前被配置过，这个是通过向wpa_supplicant 发送 LIST_NETWORK 命令并且比较返回值来实现的

```
// Need WifiConfiguration for the AP
WifiConfiguration config = findConfiguredNetwork(state);
```

如果wpa_supplicant 没有这个AP 的配置信息，则会向wpa_supplicant 发送ADD_NETWORK 命令来添加该AP，

```
if (config == null) {
// Connecting for the first time, need to create it
config = addConfiguration(state,
ADD_CONFIGURATION_ENABLE|ADD_CONFIGURATION_SAVE);
}
```

ADD_NETWORK 命令会返回一个ID ，WifiLayer 再用这个返回的ID 作为参数向 wpa_supplicant 发送ENABLE_NETWORK 命令，从而让wpa_supplicant 去连接该AP。

```
// Make sure that network is enabled, and disable others
mReenableApsOnNetworkStateChange = true;
if (!mWifiManager.enableNetwork(state.networkId, true)) {
Log.e(TAG, "Could not enable network ID " + state.networkId);
error(R.string.error_connecting);
return false;
}
```

10.6 配置 IP 地址

当 wpa_supplicant 成功连接上AP 之后，它会向控制通道发送事件通知连接上AP 了，从而 wifi_wait_for_event 函数会接收到该事件，由此WifiMonitor 中的MonitorThread 会被执行来出来这个事件，

```
void handleEvent(int event, String remainder) {
case CONNECTED:
handleNetworkStateChange(NetworkInfo.DetailedState.CONNECTED,
remainder);
break;
```

WifiMonitor 再调用WifiStateTracker 的notifyStateChange，WifiStateTracker 则接着会往自身发送EVENT_DHCP_START 消息来启动DHCP 去获取IP 地址，

```
private void handleConnectedState() {
setPollTimer();
mLastSignalLevel = -1;
if (!mHaveIPAddress && !mObtainingIPAddress) {
```

```
mObtainingIPAddress = true;
mDhcpTarget.obtainMessage(EVENT_DHCP_START).sendToTarget();
}
}
```

然后再广播发送NETWORK_STATE_CHANGED_ACTION 这个Intent

```
case EVENT_NETWORK_STATE_CHANGED:
if (result.state != DetailedState.DISCONNECTED || !mDisconnectPending) {
intent = new
Intent(WifiManager.NETWORK_STATE_CHANGED_ACTION);
intent.putExtra(WifiManager.EXTRA_NETWORK_INFO,
mNetworkInfo);
if (result.BSSID != null)
intent.putExtra(WifiManager.EXTRA_BSSID, result.BSSID);
mContext.sendStickyBroadcast(intent);
}
break;
```

WifiLayer 注册了接收NETWORK_STATE_CHANGED_ACTION 这个Intent，所以它的相关处理函数handleNetworkStateChanged 会被调用，

当 DHCP 拿到IP 地址之后，会再发送EVENT_DHCP_SUCCEEDED 消息，

```
private class DhcpHandler extends Handler {
public void handleMessage(Message msg) {
switch (msg.what) {
case EVENT_DHCP_START:
if (NetworkUtils.runDhcp(mInterfaceName, mDhcpInfo)) {
event = EVENT_DHCP_SUCCEEDED;
}
```

WifiLayer 处理EVENT_DHCP_SUCCEEDED 消息， 会再次广播发送 NETWORK_STATE_CHANGED_ACTION 这个Intent，这次带上完整的IP 地址信息。

```
case EVENT_DHCP_SUCCEEDED:
mWifiInfo.setIpAddress(mDhcpInfo.ipAddress);
setDetailedState(DetailedState.CONNECTED);
intent = new
Intent(WifiManager.NETWORK_STATE_CHANGED_ACTION);
intent.putExtra(WifiManager.EXTRA_NETWORK_INFO, mNetworkInfo);
mContext.sendStickyBroadcast(intent);
break;
```

至此为止，整个连接过程完成。

问题：

目前的实现不支持 Ad-hoc 方式。

11.了解 HTTP

11.1 HTTP 协议是什么？

HTTP 是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于 1990 年提出，经过几年的使用与发展，得到不断地完善和扩展。目前在 WWW 中使用的是 HTTP/1.0 的第六版，HTTP/1.1 的规范化工作正在进行之中，而且 HTTP-NG(Next Generation of HTTP)的建议已经提出。
简单来说，就是一个基于应用层的通信规范：双方要进行通信，大家都要遵守一个规范，这个规范就是 HTTP 协议。

11.2 HTTP 协议的主要特点？

1.支持客户/服务器模式；
2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快；
3.灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记；
4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间；
5.无状态：HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快；

11.3 HTTP 协议能做什么？

很多人首先一定会想到：浏览网页。没错，浏览网页是 HTTP 的主要应用，但是这并不代表 HTTP 就只能应用于网页的浏览。HTTP 是一种协议，只要通信的双方都遵守这个协议，HTTP 就能有用武之地。我们今天要讲的就是在 Android 中通过 HTTP 交换数据（获取和提交）。

11.4 HTTP 协议如何工作？

大家都知道一般的通信流程：首先客户端发送一个请求(request)给服务器，服务器在接收到这个请求后将生成一个响应(response)返回给客户端。

12.HTTP 协议详解

12.1 HTTP 协议详解之 URL 篇

http（超文本传输协议）是一个基于请求与响应模式的、无状态的、应用层的协议，常基于 TCP 的连接方式，HTTP1.1 版本中给出一种持续连接的机制，绝大多数的 Web 开发，都是构建在 HTTP 协议之上的 Web 应用。

HTTP URL (URL 是一种特殊类型的 URI，包含了用于查找某个资源的足够的信息)的格式如下：
http://host[":"port][abs_path]
http 表示要通过 HTTP 协议来定位网络资源；host 表示合法的 Internet 主机域名或者 IP 地址；port 指定一个端口号，
为空则使用缺省端口 80；abs_path 指定请求资源的 URI；如果 URL 中没有给出 abs_path，那么当它作为请求 URI 时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。
eg:
1、输入：www.eoeandroid.com
浏览器自动转换成：http://www.eoeandroid.com/

12.2 HTTP 协议详解之请求篇

http 请求由三部分组成，分别是：请求行、消息报头、请求正文

1、**请求行**以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本，格式如下：
Method Request-URI HTTP-Version CRLF
其中 Method 表示请求方法；Request-URI 是一个统一资源标识符；HTTP-Version 表示请求的 HTTP 协议版本；CRLF 表示回车和换行（除了作为结尾的 CRLF 外，不允许出现单独的 CR 或 LF 字符）。

请求方法（所有方法全为大写）有多种，各个方法的解释如下：
GET 请求获取 Request-URI 所标识的资源
POST 在 Request-URI 所标识的资源后附加新的数据
HEAD 请求获取由 Request-URI 所标识的资源的响应消息报头
PUT 请求服务器存储一个资源，并用 Request-URI 作为其标识
DELETE 请求服务器删除 Request-URI 所标识的资源
TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断
CONNECT 保留将来使用
OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求

应用举例：
GET 方法：在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用 GET 方法向服务器获取资源，
eg:GET /form.html HTTP/1.1 (CRLF)

POST 方法：要求被请求服务器接受附在请求后面的数据，常用于提交表单。

eg：POST /reg.jsp HTTP/ (CRLF)
Accept:image/gif,image/x-xbit,... (CRLF)
...
HOST:www.eoeandroid.com (CRLF)
Content-Length:22 (CRLF)
Connection:Keep-Alive (CRLF)
Cache-Control:no-cache (CRLF)
(CRLF) //该 CRLF 表示消息报头已经结束，在此之前为消息报头
user=jeffrey&pwd=1234 //此行以下为提交的数据

HEAD 方法与 GET 方法几乎是一样的，对于 HEAD 请求的回应部分来说，它的 HTTP 头部中包含的信息与通过 GET 请求所得到的信息是相同的。利用这个方法，不必传输整个资源内容，就可以得到 Request-URI 所标识的资源的信息。该方法常用于测试超链接的有效性，是否可以访问，以及最近是否更新。

2、请求报头后述
3、请求正文(略)

12.3 HTTP 协议详解之响应篇

在接收和解释请求消息后，服务器返回一个 HTTP 响应消息。
HTTP 响应也是由三个部分组成，分别是：状态行、消息报头、响应正文

1、**状态行**格式如下：
HTTP-Version Status-Code Reason-Phrase CRLF
其中，HTTP-Version 表示服务器 HTTP 协议的版本；Status-Code 表示服务器发回的响应状态代码；Reason-Phrase 表示状态代码的文本描述。
状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：
1xx：指示信息--表示请求已接收，继续处理
2xx：成功--表示请求已被成功接收、理解、接受
3xx：重定向--要完成请求必须进行更进一步的操作
4xx：客户端错误--请求有语法错误或请求无法实现
5xx：服务器端错误--服务器未能实现合法的请求
常见状态代码、状态描述、说明：
200 OK //客户端请求成功
400 Bad Request //客户端请求有语法错误，不能被服务器所理解
401 Unauthorized //请求未经授权，这个状态代码必须和 WWW-Authenticate 报 //头域一起使用
403 Forbidden //服务器收到请求，但是拒绝提供服务
404 Not Found //请求资源不存在，eg：输入了错误的 URL
500 Internal Server Error //服务器发生不可预期的错误
503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后， //可能恢复正常
eg：HTTP/1.1 200 OK （CRLF）

2、响应报头后述

3、响应正文就是服务器返回的资源的内容

12.4 HTTP 协议详解之消息报头篇

HTTP 消息由客户端到服务器的请求和服务器到客户端的响应组成。请求消息和响应消息都是由开始行（对于请求消息，开始行就是请求行，对于响应消息，开始行就是状态行），消息报头（可选），空行（只有 CRLF 的行），消息正文（可选）组成。

HTTP 消息报头包括普通报头、请求报头、响应报头、实体报头。
每一个报头域都是由名字+“：”+空格+值 组成，消息报头域的名字是大小写无关的。

1、普通报头
在普通报头中，有少数报头域用于所有的请求和响应消息，但并不用于被传输的实体，只用于传输的消息。
eg：
Cache-Control 用于指定缓存指令，缓存指令是单向的（响应中出现的缓存指令在请求中未必会出现），且是独立的（一个消息的缓存指令不会影响另一个消息处理的缓存机制），HTTP1.0 使用的类似的报头域为 Pragma。
请求时的缓存指令包括：no-cache（用于指示请求或响应消息不能缓存）、no-store、max-age、max-stale、min-fresh、only-if-cached;
响应时的缓存指令包括：public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age、s-maxage.
eg：为了指示 IE 浏览器（客户端）不要缓存页面，服务器端的 JSP 程序可以编写如下：response.sehHeader ("Cache-Control","no-cache");
//response.setHeader("Pragma","no-cache");作用相当于上述代码，通常两者//合用
这句代码将在发送的响应消息中设置普通报头域：Cache-Control:no-cache

Date 普通报头域表示消息产生的日期和时间

Connection 普通报头域允许发送指定连接的选项。例如指定连接是连续，或者指定“close”选项，通知服务器，在响应完成后，关闭连接

2、请求报头
请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。
常用的请求报头
Accept
Accept 请求报头域用于指定客户端接受哪些类型的信息。eg：Accept：image/gif，表明客户端希望接受 GIF 图象格式的资源；Accept：text/html，表明客户端希望接受 html 文本。
Accept-Charset
Accept-Charset 请求报头域用于指定客户端接受的字符集。eg：Accept-Charset:iso-8859-1,gb2312.如果在请求消息中没有设置这个域，缺省是任何字符集都可以接受。

Accept-Encoding

Accept-Encoding 请求报头域类似于 Accept，但是它是用于指定可接受的内容编码。eg: Accept-Encoding:gzip.deflate. 如果请求消息中没有设置这个域服务器假定客户端对各种内容编码都可以接受。

Accept-Language

Accept-Language 请求报头域类似于 Accept，但是它是用于指定一种自然语言。eg：Accept-Language:zh-cn.如果请求消息中没有设置这个报头域，服务器假定客户端对各种语言都可以接受。

Authorization

Authorization 请求报头域主要用于证明客户端有权查看某个资源。当浏览器访问一个页面时，如果收到服务器的响应代码为 401（未授权），可以发送一个包含 Authorization 请求报头域的请求，要求服务器对其进行验证。

Host（发送请求时，该报头域是必需的）

Host 请求报头域主要用于指定被请求资源的 Internet 主机和端口号，它通常从 HTTP URL 中提取出来的，eg：

我们在浏览器中输入：http://www.guet.edu.cn/index.html

浏览器发送的请求消息中，就会包含 Host 请求报头域，如下：

Host：www.guet.edu.cn

此处使用缺省端口号 80，若指定了端口号，则变成：Host：www.guet.edu.cn:指定端口号

User-Agent

我们上网登陆论坛的时候，往往会看到一些欢迎信息，其中列出了你的操作系统的名称和版本，你所使用的浏览器的名称和版本，这往往让很多人感到很神奇，实际上，服务器应用程序就是从 User-Agent 这个请求报头域中获取到这些信息。User-Agent 请求报头域允许客户端将它的操作系统、浏览器和其它属性告诉服务器。不过，这个报头域不是必需的，如果我们自己编写一个浏览器，不使用 User-Agent 请求报头域，那么服务器端就无法得知我们的信息了。

请求报头举例：

GET /form.html HTTP/1.1 (CRLF)

Accept:image/gif,image/x-xbitmap,image/jpeg,application/x-shockwave-flash,application/vnd.ms-excel,application/vnd.ms-

powerpoint,application/msword,*/* (CRLF)

Accept-Language:zh-cn (CRLF)

Accept-Encoding:gzip,deflate (CRLF)

If-Modified-Since:Wed,05 Jan 2007 11:21:25 GMT (CRLF)

If-None-Match:W/"80b1a4c018f3c41:8317" (CRLF)

User-Agent:Mozilla/4.0(compatible;MSIE6.0;Windows NT 5.0) (CRLF)

Host:www.guet.edu.cn (CRLF)

Connection:Keep-Alive (CRLF)

(CRLF)

3、响应报头

响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对 Request-URI 所标识的资源进行下一步访问的信息。

常用的响应报头

Location

Location 响应报头域用于重定向接受者到一个新的位置。Location 响应报头域常用在更换域名的时候。

Server

Server 响应报头域包含了服务器用来处理请求的软件信息。与 User-Agent 请求报头域是相对应的。下面是

Server 响应报头域的一个例子：

Server：Apache-Coyote/1.1

WWW-Authenticate

WWW-Authenticate 响应报头域必须被包含在 401（未授权的）响应消息中，客户端收到 401 响应消息时候，并发送 Authorization 报头域请求服务器对其进行验证时，服务端响应报头就包含该报头域。

eg：WWW-Authenticate:Basic realm="Basic Auth Test!" //可以看出服务器对请求资源采用的是基本验证机制。

4、实体报头

请求和响应消息都可以传送一个实体。一个实体由实体报头域和实体正文组成，但并不是说实体报头域和实体正文要在一起发送，可以只发送实体报头域。实体报头定义了关于实体正文（eg：有无实体正文）和请求所标识的资源的元信息。

常用的实体报头

Content-Encoding

Content-Encoding 实体报头域被用作媒体类型的修饰符，它的值指示了已经被应用到实体正文的附加内容的编码，因而要获得 Content-Type 报头域中所引用的媒体类型，必须采用相应的解码机制。Content-Encoding 这样用于记录文档的压缩方法，eg：Content-Encoding：gzip

Content-Language

Content-Language 实体报头域描述了资源所用的自然语言。没有设置该域则认为实体内容将提供给所有的语言阅读

者。eg：Content-Language:da

Content-Length

Content-Length 实体报头域用于指明实体正文的长度，以字节方式存储的十进制数字来表示。

Content-Type

Content-Type 实体报头域用语指明发送给接收者的实体正文的媒体类型。eg：

Content-Type:text/html;charset=ISO-8859-1

Content-Type:text/html;charset=GB2312

Last-Modified

Last-Modified 实体报头域用于指示资源的最后修改日期和时间。

Expires

Expires 实体报头域给出响应过期的日期和时间。为了让代理服务器或浏览器在一段时间以后更新缓存中(再次访问曾访问过的页面时，直接从缓存中加载，缩短响应时间和降低服务器负载)的页面，我们可以使用 Expires 实体报头域指定页面过期的时间。eg：Expires：Thu，15 Sep 2006 16:23:12 GMT

HTTP1.1 的客户端和缓存必须将其他非法的日期格式（包括 0）看作已经过期。eg：为了让浏览器不要缓存页面，我们也可以利用 Expires 实体报头域，设置为 0，jsp 中程序如下：response.setDateHeader("Expires","0");

13. Android 实例

通过前面的分析，我们可以了解到可以 GET,POST,HEAD 等等方法获取（提交）数据，这里以我们最常用使用的 GET 和 POST 为例，说明如何在 Anroid 通过 HTTP 交换数据。

13.1 通过 GET 请求获取数据

```
/**
* Sends an HTTP GET request to a url
* @param endpoint
*              - The URL of the server. (Example:
*              " http://www.yahoo.com/search")
* @param requestParameters
*              - all the request parameters (Example:
*              "param1=val1&param2=val2"). Note: This method will add the
*              question mark (?) to the request - DO NOT add it yourself
* @return - The response from the end point
*/
public static String sendGetRequest(String endpoint,
String requestParameters) {
Log.i("sendGetRequest", endpoint);
String result = null;
if (endpoint.startsWith("http://")) {
// Send a GET request to the servlet
try {
// Construct data
StringBuffer data = new StringBuffer();
// Send data
String urlStr = endpoint;
if (requestParameters != null && requestParameters.length() > 0) {
urlStr += "?" + requestParameters;
}
Log.i("urlStr", urlStr);
URL url = new URL(urlStr);
URLConnection conn = url.openConnection();

// Get the response
BufferedReader rd = new BufferedReader(new InputStreamReader(
conn.getInputStream()));
StringBuffer sb = new StringBuffer();
String line;
while ((line = rd.readLine()) != null) {
```

```
sb.append(line);
}
rd.close();
result = sb.toString();
} catch (Exception e) {
e.printStackTrace();
}}
Log.i("sendGetRequest", result);
return result;
}
```

比如再举个实际的例子，通过指定 URL 的图片，获取图片数据，并转换成 Android 的支持的图片格式，相关代码如下：

```
// 显示网络上的图片
public static Bitmap returnBitMap(String url) {
Log.i("returnBitMap", "url=" + url);
URL myFileUrl = null;
Bitmap bitmap = null;
try {
myFileUrl = new URL(url);
} catch (MalformedURLException e) {
e.printStackTrace();
}
try {
HttpURLConnection conn = (HttpURLConnection) myFileUrl
.openConnection();
conn.setDoInput(true);
conn.connect();
InputStream is = conn.getInputStream();
bitmap = BitmapFactory.decodeStream(is);
is.close();
} catch (IOException e) {
e.printStackTrace();
}
return bitmap;
}
```

13.2 通过 POST 提交数据

很多情况下，需要通过 POST 将本地的数据发送给服务器，如下是一段实际的代码

```
public void MyFunction{
HttpClient httpclient = new DefaultHttpClient();
```

```
//你的 URL
 HttpPost httppost = new HttpPost("http://www.eoeandroid.com/post_datas.php");
 try {
   List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(2);
//Your DATA
   nameValuePairs.add(new BasicNameValuePair("id", "12345"));
   nameValuePairs.add(new BasicNameValuePair("stringdata", "eoeAndroid.com is Cool!"));
   httppost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
   HttpResponse response;
   response=httpclient.execute(httppost);
 } catch (ClientProtocolException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
 } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
 }
}
```

13.3. 小结

本篇文部分主要介绍了 HTTP 相关的概念，协议，特点等，最后通过几段代码，介绍了如何在 android 上通过 HTTP GET 获取数据，以及通过 HTTP POST 提交数据。
通过本篇学习，希望你可以了解 HTTP，并学会在 Android 中通过 HTTP 交换数据。

14.编后语

还是要对大家说声抱歉，第一次完全负责这次特刊的发布，结果还是不能按时！不过好在的是，在周日还是发了发出来了！

这几天发生了很多事情，对于我呢！终于跨出了人生的一大步，论坛的第一期的董事会招募也很成功，第二期的结果页更加出乎我意料。论坛的工作，终于不用都是我一个人包办了，而且相信有这么多朋友的加入，也能把论坛做的更好。

而这次把时间拖这么唱的最重要的原因，就是我在准备 eoeAndroid 董事会第三期的招募!

这期最主要的任务就是 eoe 特刊小组的招募了！

其实如果按照现在的思路，也能这么做下去，但是终究不长久，而且这些很容易让别人模仿，结果又是山寨一堆，对大家的学习也影响很不好。所以要变，要迎来改变！只有改变才能求生存！

首先，就是我们要继续扩大特刊的范围，要让更多朋友来加入。

没有奖励机制很难吸引到很多人，那么以后的特刊，我们要给为每一位对特刊做出贡献的朋友提供方便，比如他的求职。而且如果我们的特刊能做大做强，能够吸引真是的 RMB，那有什么理由不能变成大家口袋里的东西呢？如果少，起码多几个 Q 币也是可以的吗！

另外还有我们的合作方式，我们没有等级，只要有能力就可以来做！

当然其实最感谢的还是海阳哥他们，能给我过生日，特别感谢他们！都不知道该说什么话了！

好了，我们下期见！

15. 其他

BUG 提交 如果你发现文档中翻译不妥的地方，请到如下地址反馈，我们会定期更新、发布更新后的版本

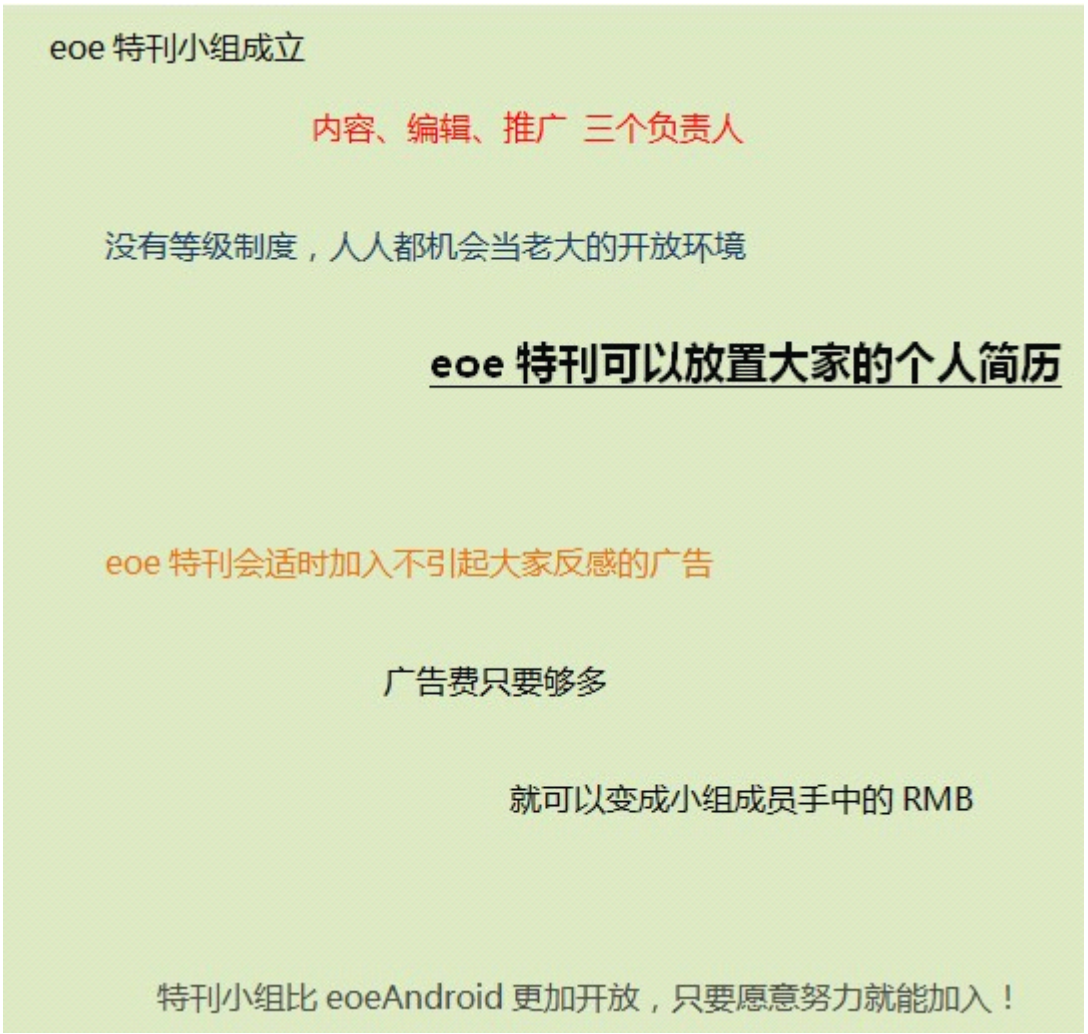
http://www.eoeandroid.com/viewthread.php?tid=753

资源下载： 本期文部分中包含的源码请在如下地址下载

http://www.eoeandroid.com/viewthread.php?tid=753

加入 eoe 特刊小组，为你的将来磨平道路

下面只是广告，详细信息请浏览页面：
http://www.eoeandroid.com/viewthread.php?tid=1784



eoe 特刊小组从今天开始正式成立，并吸纳全中国有识之士，eoe 特刊小组欢迎你的加入！并也希望借助 eoe 特刊的力量，帮助大家提高、进步，或者是~~~~

废话不多说，有意加入朋友，请立即访问

http://www.eoeandroid.com/viewthread.php?tid=1784

关于 eoeAndroid

当前，3G 商业，传统互联网与移动互联网也呈现出全业务发展的融合趋势，电信与互联网行业已经踏入继单机计算时代、传统互联网时代之后的第三个纪元。

由于看好移动互联网和 Android 手机平台的商业前景，同时也拥有专业而独特的产品、技术服务能力，我们聚集了一群热爱 Android 的技术英才，组建了 eoeMobile 团队。

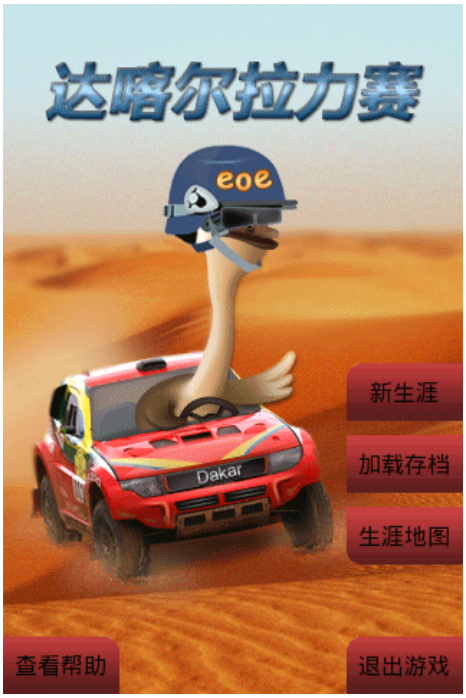
eoeMobile 是一支专注于 Android 平台应用开发、产品运营和相关商业与技术服务的团队，立志于建立中国最大的 Android 应用开发专业社区 eoeAndroid.com，想为 Android 在中国的发展尽自己的微薄之力。

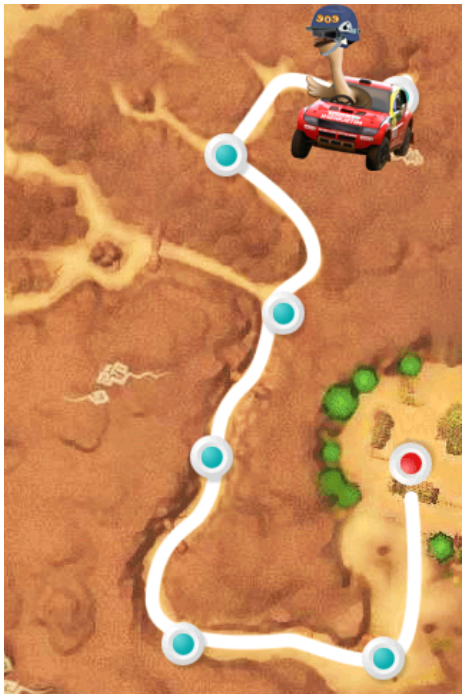
17. 广告

eoeDakar

eoeMobile 团队最新出品的一款游戏软件，以真实的达喀尔拉力赛为基准，赛途中会碰到很多的障碍物，但不是所有人都能顺利通过这些赛段哦！而游戏最大的乐趣也就在这里了！

第一个公测版已经放出，在 Android Market 上搜索 eoeDakar 即可获取软件！

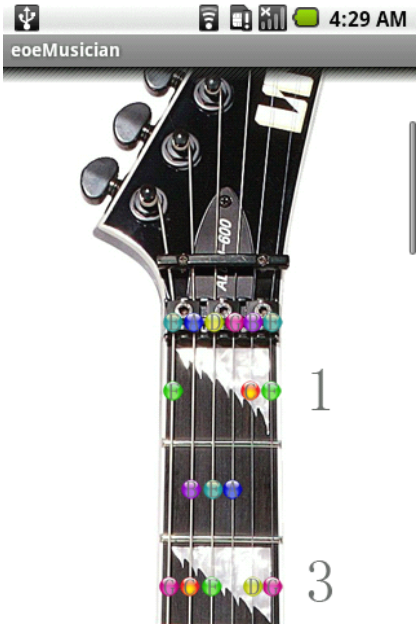




eoeMusician

eoeMobile 是一 eoeMobile 团队制作的一款最专业的软件，继承了多种专业的音乐功能。

这款软件是游利卡见过的 Android 平台上最专业的一款音乐软件了，其中包括手机吉他、架子鼓、音阶查询器、节拍器等等非常多的专业的应用，而且很多功能都酷炫到不行！



感受 eoeMusican 的魅力，快去 Android Market 上下载吧！