#10 网络访问

学习目标

- 通信方式简介
- HTTP网络访问
- WIFI编程
- 蓝牙编程

Android支持的通信模式

- GSM
- EDGE
- 3G
- WIFI
- BLUETOOTH
- NFC
- …

GSM

全球移动通信系统（Global System for Mobile Communication）当前应用最为广泛的移动电话标准。

- 使得用户的国际漫游变得很平常。
- 信令和语音信道都是数字式的，称为第二代 (2G)移动电话系统。
- 2015年，全球诸多GSM网络运营商，2017年确定为关闭GSM网络的年份。

EDGE

增强型数据速率GSM演进技术（Enhanced Data Rate for GSM Evolution）

- 从GSM到3G的过渡技术
- 能够充分利用现有的GSM资源
- 弹性优势
- 工作在TDMA和GSM网络
- 提高了GPRS信道编码效率及其高速移动数据标准

第三代移动通信技术（3G）

- 3G就是指IMT-2000（International Mobile Telecommunications-2000），是国际电信联盟（ITU）定义的第三代无线通信的全球标准。
- IMT-2000规定的移动终端的连接速度
  - 以车速移动时 --144kbps
  - 室外静止或步行时 --384kbps
  - 室内 --2Mbps
- 应用广泛：宽带上网 视频通话 手机购物等

WIFI

Wireless Fidelity，中文译为“无线兼容认证”

- 实质 —— 一种商业认证
- 技术 —— 短程无线传输
- 现状 —— 带WIFI的便捷式设备是潮流

常见的WIFI使用形式——无线路由器

- 覆盖范围 —— 70米到120米
- 使用场合 —— 公司、家庭、公共场所
- 优点 —— 方便的建立局域网、低成本、使用简单

WIFI——特点

WIFI相比其他技术有如下特点：

- 无线电波的覆盖范围广
- 传输速度高
- 使用门槛比较低
- 消除布线的麻烦
- 发射功率低，健康安全

Bluetooth

- 定义：
  1. 开放式无限通讯标准
  2. 设备短距离互联解决方案
- 优势：
  1. 无需驱动程序——独特的配置
  2. 小型化无线电
  3. 低功率、低成本、安全性、稳固
  4. 易于使用、即时连接

Near Field Communication（近场通讯）

- 技术优势：
  1. 轻松、安全、迅速的通信
  2. 传输范围小——独特的信息衰减技术
  3. 带宽高、能耗低
- 应用场合：
  1. 门禁、公交
  2. 手机支付

学习目标

- 通信方式简介
- HTTP网络访问
- WIFI编程
- 蓝牙编程

HTTP网络访问

- 连接网络

学习如何连接到网络，选择一个 HTTP client，以及在 UI 线程外执行网络操作

- 管理网络使用情况

学习如何检查设备的网络连接情况，创建偏好界面来控制网络使用，以及响应连接变化

- 解析XML数据

学习如何解析和使用 XML 数据。

连接网络—添加权限

使用网络，应该有相应使用允许。文件AndroidManifest添加：

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

android.permission.INTERNET——允许应用程序打开网络套接字。

android.permission.ACCESS_NETWORK_STATE——允许应用程序访问网络连接信息。

连接网络—选择HTTP Client

大多数连接网络的 Android app 会使用 HTTP 来发送与接收数据。

两种 HTTP clients：HttpURLConnection 与 Apache HttpClient。

二者均支持 HTTPS、流媒体上传和下载、可配置的超时、IPv6 与连接池。

对于 Android 2.3 或更高的版本，推荐使用 HttpURLConnection。

连接网络—检查网络连接

在尝试连接网络之前，应通过以下函数检测当前网络是否可用

getActiveNetworkInfo()　// 获取代表联网状态的NetWorkInfo对象

isConnected()　//判断网络是否连接

```
public void myClickHandler(View view) {
    String stringUrl;//需要访问的URL
    ConnectivityManager connMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if(networkInfo != null && networkInfo.isConnected()) //表示网络已连接
    {
        new DownloadWebpageText().execute(stringUrl);
        //创建AsyncTask实例并执行
    } else {
        textView.setText("No network connection available.");
    }
}
```

连接网络—检查网络连接

注：getActiveNetworkInfo() 方法返回一个 NetworkInfo 实例，它表示可以找到的第一个已连接的网络接口，如果返回 null，则表示没有已连接的网络接口(意味着网络连接不可用)

NetworkInfo 类：描述一个网络接口的状态。

isAvailable()方法：指示是否存在网络连接。当一个持续的或者半持续的条件组织了网络连接时，网络是不可用的

isConnected()方法：指示是否存在网络连接，并可能建立连接和传递数据。

连接网络—Android线程模型

Android线程模型--- AsyncTask

主要实现了doInBackground函数和onPostExecute函数

- 在Android1.5中，Android.os包引入了一个新的类，称为AsyncTask
- 它是一个抽象的辅助类，用来管理后台操作，并最终返回到UI线程。
- 开发人员创建一个AsyncTask的子类并实现相应的事件方法，这与为后台处理创建线程并使用消息机制更新UI不同

连接网络—Android线程模型

AsyncTask类

doInBackground()方法会自动地在工作者线程中执行

onPreExecute()、onPostExecute()和onProgressUpdate()方法会在UI线程中被调用

doInBackground()方法的返回值会被传递给onPostExecute()方法

在doInBackground()方法中你可以调用publishProgress()方法，每一次调用都会使UI线程执行一次onProgressUpdate()方法

连接网络—执行网络操作

网络操作会遇到不可预期的延迟。为了避免造成不好的用户体验，总是在 UI 线程之外单独的线程中执行网络操作。

AsyncTask 类提供了一种简单的方式来处理这个问题：

doInBackground()：执行 downloadUrl() 方法。它以网页的 URL 作为参数，方法 downloadUrl() 获取并处理网页返回的数据。执行完毕后，返回一个结果字符串。

onPostExecute(): 接收结果字符串并把它显示到 UI 上

连接网络—执行网络操作

```
private class DownloadWebpageText extends AsyncTask {
        @Override
        protected String doInBackground(String... urls) {
                try {
                        return downloadUrl(urls[0]); //连接并下载数据
                } catch (IOException e) {
                        return "Unable to retrieve web page.";
                }
        }
        @Override
        protected void onPostExecute(String result) {
            textView.setText(result); //将结果字符串显示在UI界面
        }
}
```

在检查网络连接时我们创建了task实例并且调用execute方法，

注意：AsyncTask的实例必须在UI thread中创建；execute方法必须在UI thread中调用；

连接网络—连接并下载数据

在执行网络交互的线程里面，我们可以使用

HttpURLConnection 来执行一个 GET 类型的操作并下载数据

函数解读：

setReadTimeout() 将读超时设置为指定的超时值，以毫秒为单位

setConnectTimeout() 设置连接超时时间

setRequestMethod() 设定HTTP请求的方法

setDoInput() 设置是否从httpUrlConnection读入，默认情况下是true

setDoOutput() 设置是否向httpUrlConnection输出，默认情况下是false

getResponseCode() 会返回连接的状态码（status code）。这是一种获知额外网络连接信息的有效方式。其中，状态码是 200 则意味着连接成功。

连接网络—连接并下载数据

```
private String downloadUrl(String myurl) throws IOException
{    InputStream is = null;
     int len = 500;
     try {
          URL url = new URL(myurl);
          HttpURLConnection conn = (HttpURLConnection) url.openConnection();
          conn.setReadTimeout(10000);
          conn.setConnectTimeout(15000);
          conn.setRequestMethod("GET");
          conn.setDoInput(true);
          conn.connect();
          int response = conn.getResponseCode(); //状态码为200表示连接成功
          Log.d(DEBUG_TAG, "The response is: " + response);
          is = conn.getInputStream(); //获取输入流
          String contentAsString = readIt(is, len); //转换为string
          return contentAsString;
     }
     finally {
          if (is != null) {
                    is.close();
          }
     }
}
```

连接网络—将输入流转换为字符串

InputStream 是一种可读的 byte 数据源。如果我们获得了一个 InputStream，通常会进行解码（decode）或者转换为目标数据类型：

```
public String readIt(InputStream stream, int len) throws IOException,
UnsupportedEncodingException
{        Reader reader = null;
         reader = new InputStreamReader(stream, "UTF-8");
         char[] buffer = new char[len];
         reader.read(buffer);
         return new String(buffer);
}
```

连接网络—将输入流转换为字符串

InputStream是表示字节输入流的所有类的超类

Reader是用于读取字符流的抽象类

InputStream提供的是字节流的读取，而非文本读取

即用Reader读取出来的是char数组或者String ，使用InputStream读取出来的是byte数组。

借用Reader类我们可以将输入流转换为字符串。

连接网络流程

流程：

1）创建AsyncTask实例，调用execute方法

2）AsyncTask的doInBackground() 方法调用 downloadUrl() 方法

3）downloadUrl() 方法以一个 URL 字符串作为参数，并用它创建一个 URL 对象。这个 URL 对象被用来创建一个 HttpURLConnection。

4）一旦建立连接，HttpURLConnection 对象将获取网页的内容并得到一个 InputStream。

5）InputStream 被传给 readIt() 方法，该方法将流转换成字符串。

6）最后，AsyncTask 的 onPostExecute() 方法将字符串展示在 main activity 的 UI 上。

管理网络使用

如果我们的程序需要执行大量网络操作，那么应该提供用户设置选项，来允许用户控制程序的数据偏好。

- 是否只在连接到WIFI才进行下载和上传操作
- 是否在漫游时使用套餐数据流量
- 精确的控制APP使用数据流量

管理网络使用—检测网络连接

为了检测网络连接，我们需要使用到下面两个类：

ConnectivityManager：它会回答关于网络连接的查询结果，并在网络连接改变时通知应用程序。

NetworkInfo：描述一个给定类型的网络接口状态

```
private static final String DEBUG_TAG = “NetworkStatusExample”;
ConnectivityManager connMgr = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE); //得到网络连接查询结果
NetworkInfo networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected(); //判断WIFI是否连接
networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected(); //判断数据是否连接
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

管理网络使用—实例

我们可以实现一个偏好设置的 activity ，使用户能直接设置程序对网络资源的使用情况。例如:

- 可以允许用户仅在连接到 WiFi 时上传视频。
- 可以根据诸如网络可用，时间间隔等条件来选择是否做同步的操作。

管理网络使用—实例

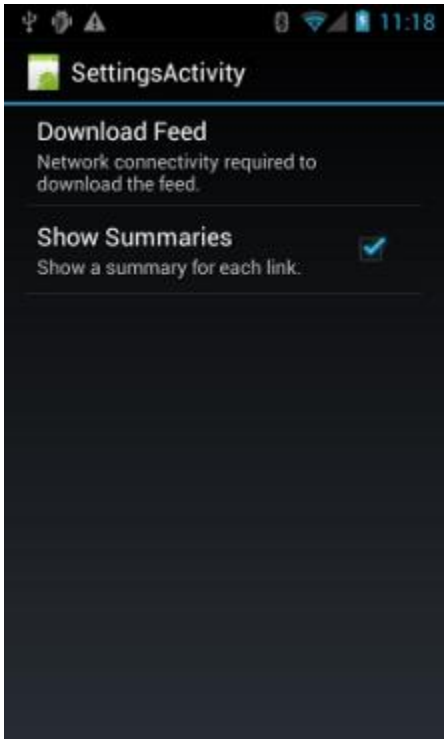
除了在manifest添加网络所需权限，还需要在Activity中声明 intent filter，可以为 ACTION_MANAGE_NETWORK_USAGE action声明 intent filter，表示我们的应用定义了一个提供控制数据使用情况选项的 activity：

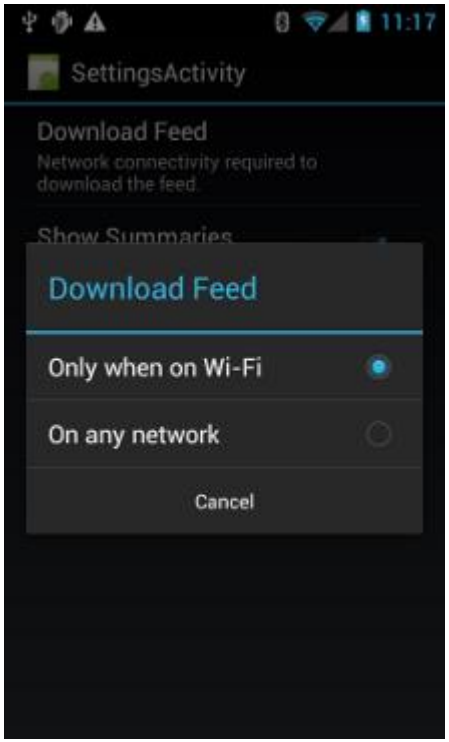
```
<activity android:label="SettingsActivity" android:name=".SettingsActivity">
<intent-filter>
<action android:name="android.intent.action.MANAGE_NETWORK_USAGE" />
<category android:name="android.intent.category.DEFAULT" /> </intent-filter>
</activity>
```

注：ACTION_MANAGE_NETWORK_USAGE 显示管理指定应用程序网络数据使用情况的设置。当我们的 app 有一个允许用户控制网络使用情况的设置 activity 时，我们应该为 activity 声明这个 intent filter。

管理网络使用—实例

SettingsActivity 是 PreferenceActivity 的子类，它展示一个偏好设置页面（如下两张图）让用户指定以下内容:





管理网络使用—实例

SettingsActivity，其实现了 OnSharedPreferenceChangeListener，并且实现onSharedPreferenceChanged() , onCreate(), onResume(), onPause()四个函数。

```
public class SettingsActivity extends PreferenceActivity implements
OnSharedPreferenceChangeListener
{       @Override
        protected void onCreate(Bundle savedInstanceState){
            super.onCreate(savedInstanceState);
            addPreferencesFromResource(R.xml.preferences); //加载界面
        }
        @Override
        protected void onResume(){
            super.onResume();
        getPreferenceScreen().getSharedPreferences().registerOnSharedPrefere
nceChangeListener(this);
        }
}
```

管理网络使用—实例

当用户改变了他的偏好，就会触发 onSharedPreferenceChanged()，这个方法会设置 refreshDisplay 为 true。这会使得当用户返回到 main activity 的时候进行刷新

```
public class SettingsActivity extends PreferenceActivity implements
OnSharedPreferenceChangeListener {
        @Override
        protected void onPause(){
            super.onPause();
        getPreferenceScreen().getSharedPreferences().unregisterOnSharedPrefer
enceChangeListener(this);}
        @Override
        public void onSharedPreferenceChanged(SharedPreferences
sharedPreferences, String key){
            NetworkActivity.refreshDisplay = true;
        }
}
```

管理网络使用—实例

如果设置的类型与当前设备的网络连接类型相一致，那么程序就会下载数据并刷新显示：

```
public void onStart () {
        super.onStart();
        SharedPreferences sharedPrefs =
PreferenceManager.getDefaultSharedPreferences(this);
        //得到用户偏好设置
        sPref = sharedPrefs.getString("listPref", "Wi-Fi");
        updateConnectedFlags();
        //检查当前网络连接类型，设置wifi和数据对应bool变量

        if(refreshDisplay)//当用户改变了偏好
        {
            loadPage();//判断偏好和当前是否一致，一致时连接网络并下载数据
        }
}
```

管理网络使用—检测网络连接变化

关于 BroadcastReceiver 的子类：NetworkReceiver。 当设备网络连接改变时，NetworkReceiver 会监听到 CONNECTIVITY_ACTION，这时需要判断当前网络连接类型并相应的设置好WIFI和数据的连接状态。

注：我们需要控制好 BroadcastReceiver 的使用，不必要的声明注册会浪费系统资源。我们可以在 onCreate() 中注册 BroadcastReceiver NetworkReceiver，在 onDestroy() 中销毁它，这样做会比在 manifest 里面声明 <receiver> 更轻巧 。

管理网络使用—检测网络连接变化

NetworkReceiver ，其继承BroadcastReceiver：

```
public class NetworkReceiver extends BroadcastReceiver
{       @Override
        public void onReceive(Context context, Intent intent) {
            ConnectivityManager conn = (ConnectivityManager)
        context.getSystemService(Context.CONNECTIVITY_SERVICE);
            NetworkInfo networkInfo = conn.getActiveNetworkInfo();
            if (WIFI.equals(sPref) && networkInfo != null &&
        networkInfo.getType() == ConnectivityManager.TYPE_WIFI) {
                refreshDisplay = true;
                Toast.makeText(context, R.string.wifi_connected,
        Toast.LENGTH_SHORT).show();
            }
            else if (ANY.equals(sPref) && networkInfo != null) {
                refreshDisplay = true;
            }
            else {
                refreshDisplay = false;
                Toast.makeText(context,R.string.lost_connection,
        Toast.LENGTH_SHORT).show();
        }
}
```

解析XML数据

Extensible Markup Language（XML）是一组将文档编码成机器可读形式的规则，也是一种在网络上共享数据的普遍格式。频繁更新内容的网站，比如新闻网站或者博客，经常会提供 XML 提要（XML feed）来使得外部程序可以跟上内容的变化。下载与解析 XML 数据是网络连接相关 app 的一个常见功能。

解析XML数据

流程：

- 选择Parser
- 分析Feed
- 实例化Parser
- 读取Feed
- 解析XML

解析XML数据—选择Parser

推荐 XmlPullParser，它是 Android 上一个高效且可维护的解析 XML 的方法。 Android 上有这个接口的两种实现方式：

- KXmlParser，通过 XmlPullParserFactory.newPullParser() 得到。
- ExpatPullParser，通过 Xml.newPullParser() 得到。

解析XML数据—分析Feed

解析一个 feed 的第一步是决定我们需要获取的字段。这样解析器便去抽取出那些需要的字段而忽视其他的字段。

实例：从 entry 标签与它的子标签 title，link 和 summary 中提取数据.

```
<?xml version="1.0" encoding="utf-8"?>
<feed>
    <entry>
        <title type="text">Where is my data file?</title>
        <link rel="alternate"
href="http://stackoverflow.com/questions/9439999/where-is-my-data-file" />
        <summary type="html">
         I have an Application that requires a data file..
        </summary>
    </entry>
</feed>
```

解析XML数据—实例化 Parser

在下面的片段中，一个 parser 被初始化来处理名称空间，并且将 InputStream 作为输入。它通过调用 nextTag() 开始解析，并调用 readFeed() 方法，readFeed() 方法会提取并处理 app 需要的数据：

```
public List parse(InputStream in) throws XmlPullParserException, IOException
{
    try {
        XmlPullParser parser = Xml.newPullParser(); //实例化Parser
        parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
        //指定命名空间
        parser.setInput(in, null);
        parser.nextTag();
        return readFeed(parser);//处理所需要数据
    }
    finally { in.close(); }
}
```

解析XML数据—读取Feed

START_TAG: 事件指定了一个XML标记的开始

START_TAG: 找到一个新的标记时（ <tag> ）返回

TEXT: 当找到文本时返回（即<tag>TEXT</tag> ）

END_TAG: 找到标记的结束时（ </tag> ）返回

END_DOCUMENT: 当到达XML文件末尾时返回

解析XML数据—读取Feed

处理 feed 的内容：寻找一个 "entry" 的标签作为递归处理整个 feed 的起点。readFeed() 方法会跳过不是 entry 的标签。当整个 feed 都被递归处理后，readFeed() 会返回一个从 feed 中提取的包含了 entry 标签内容（包括里面的数据成员）的 List。然后这个 List 成为 parser 的返回值。

解析XML数据—读取Feed

```
private List readFeed(XmlPullParser parser) throws XmlPullParserException,
IOException {
    List entries = new ArrayList();
    parser.require(XmlPullParser.START_TAG, ns, "feed");//feed是根标签，只有一个
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
             continue;
         }
        String name = parser.getName();
        if (name.equals("entry")) {//entry 我们定义的实体的信息的开始标签
            entries.add(readEntry(parser));
        }
        else {
            skip(parser); //对于不敢兴趣的标签，跳过
        }
    }
    return entries;
}
```

解析XML数据—解析XML

对你感兴趣的标签，分别定义方法去读取他们：

```
private Entry readEntry(XmlPullParser parser) throws XmlPullParserException,
IOException {
    parser.require(XmlPullParser.START_TAG, ns, "entry");
    String title = null; String summary = null; String link = null;
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
        String name = parser.getName();
        if (name.equals("title")) {
            title = readTitle(parser);
        } else if (name.equals("summary")) {
            summary = readSummary(parser);
        } else if (name.equals("link")) {
            link = readLink(parser);
        } else {
            skip(parser);
        }
    }
    return new Entry(title, summary, link);
}
```

解析XML数据—解析XML

同样，对于link标签来说，定义readLink(parser)去读取：

```
        <link rel="alternate"
href="http://stackoverflow.com/questions/9439999/where-is-my-data-file" />
```

```
private String readLink(XmlPullParser parser) throws IOException,
XmlPullParserException
{       String link = "";
        parser.require(XmlPullParser.START_TAG, ns, "link");
        //link此时作为开始标签
        String tag = parser.getName();
        String relType = parser.getAttributeValue(null, "rel");
        if (tag.equals("link")) {
            if (relType.equals("alternate")){
                link = parser.getAttributeValue(null, "href");
                parser.nextTag();
            }
        }
        parser.require(XmlPullParser.END_TAG, ns, "link");
        return link;
}
```

解析XML数据—跳过不需要的标签

对于我们不需要的标签信息来说，如何跳过：

```
private void skip(XmlPullParser parser) throws XmlPullParserException,
IOException {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
                throw new IllegalStateException();
        }
        int depth = 1;
        while (depth != 0) {
            switch (parser.next()) {
                case XmlPullParser.END_TAG:
                        depth--;
                        break;
                case XmlPullParser.START_TAG:
                        depth++;
                        break;
            }
        }
}
```

解析XML数据—跳过不需要的标签

Skip()方法如何工作：

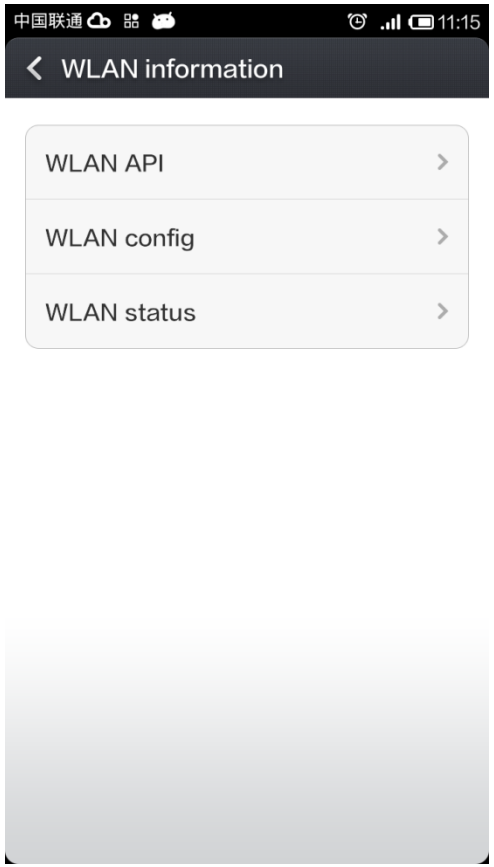
- 如果当前事件不是一个 START_TAG，抛出异常。
- 消耗掉 START_TAG 以及接下来的所有内容，包括与开始标签配对的 END_TAG。
- 为了保证方法在遇到正确的 END_TAG 时停止，而不是在最开始的 START_TAG 后面的第一个标签，方法随时记录嵌套深度

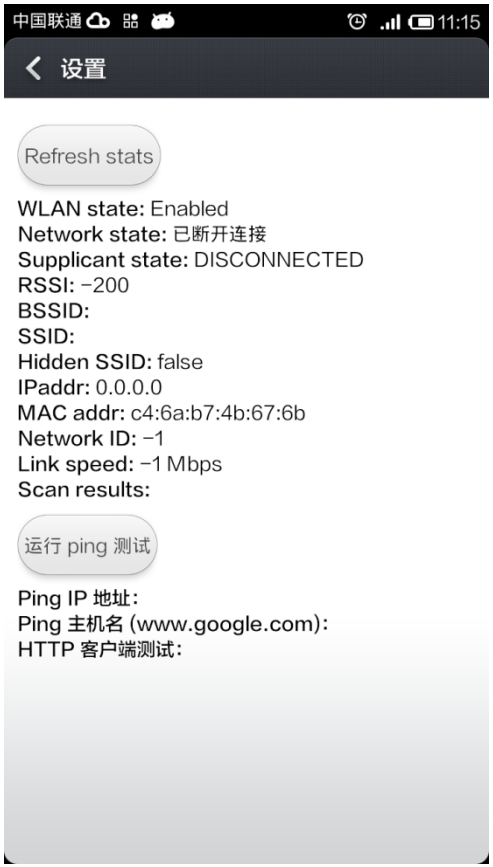
学习目标

- 通信方式简介
- HTTP网络访问
- WIFI编程
- 蓝牙编程

WIFI无线测试实验


中国联通
11:14
测试
手机信息
电池信息
使用情况统计数据
WLAN information


中国联通
11:15
WLAN information
WLAN API
WLAN config
WLAN status


中国联通
11:15
设置
Refresh stats
WLAN state: Enabled
Network state: 已断开连接
Supplicant state: DISCONNECTED
RSSI: -200
BSSID:
SSID:
Hidden SSID: false
IPaddr: 0.0.0.0
MAC addr: c4:6a:b7:4b:67:6b
Network ID: -1
Link speed: -1 Mbps
Scan results:
运行 ping 测试
Ping IP 地址:
Ping 主机名 (www.google.com):
HTTP 客户端测试:

WIFI无线测试实验

利用手机自带测试工具，查看真正的wifi相关信息：

- a)打开拨号界面，输入*#*#4636#*#*，进入android自带的测试界面
- b)进入WLAN information（wifi信息)
- c)再进入WLAN Status（wifi状态）
- d)点击Reflash Stats（刷新状态）

WIFI网络所需权限

状态名称	描述
CHANGE_NETWORK_STATE	允许应用程序改变网络连接状态
CHANGE_WIFI_STATE	允许应用程序改变WIFI连接状态
ACCESS_NETWORK_STATE	允许应用程序访问网络信息
ACCESS_WIFI_STATE	允许应用程序访问WIFI网络信息

WIFI网卡状态

- WIFI网卡的状态信息都以整型变量的形式存放在android.net.wifi.WifiManager 类中，有以下状态：
- WIFI_STATE_DISABLED ： WIFI网卡不可用
- WIFI_STATE_DISABLING ： WIFI网卡正在关闭
- WIFI_STATE_ENABLED ： WIFI网卡可用
- WIFI_STATE_ENABLING ： WIFI网卡正在打开
- WIFI_STATE_UNKNOWN ： WIFI网卡状态未知

基于Android的WIFI相关库函数

- 需要用到的WiFi相关的Android包：android.net.wifi，常见操作主要包括以下各类和接口：

（1）WifiManager：提供了管理WiFi连接的大部分API，主要包括如下内容：

- 查看已经配置好的网络清单，而且可以修改个别记录的属性。
- 可以建立或是关闭WiFi网络连接，并且可以查询有关网络状态的动态信息。
- 对接入点的扫描结果包含足够的信息来决定需要与什么接入点建立连接。
- 同时还定义了许多常量来表示WiFi状态的改变。
- 常用方法：getWiFiState,isWifiEnabled,setWifiEnabled,startScan,getScanResults等等。

基于Android的WIFI相关库函数

（2）ScanResult

- 主要用来描述已经检测出的接入点，包括：接入点的地址、接入点的名称、身份认证、频率、信号强度、安全模式等信息。

（3）WifiConfiguration

- WiFi网络的配置，包括安全配置等。

（4）WifiInfo

- WiFi无线连接的描述，包括接入点、网络连接状态、隐藏的接入点，ip地址、连接速度、mac地址、网络id、信号强度等信息。

WifiManager相关方法简介

- 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

WifiManager相关方法简介

- 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

ScanResult相关属性简介

- 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。

WifiConfiguration相关子类简介

- 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。

WifiInfo相关方法简介

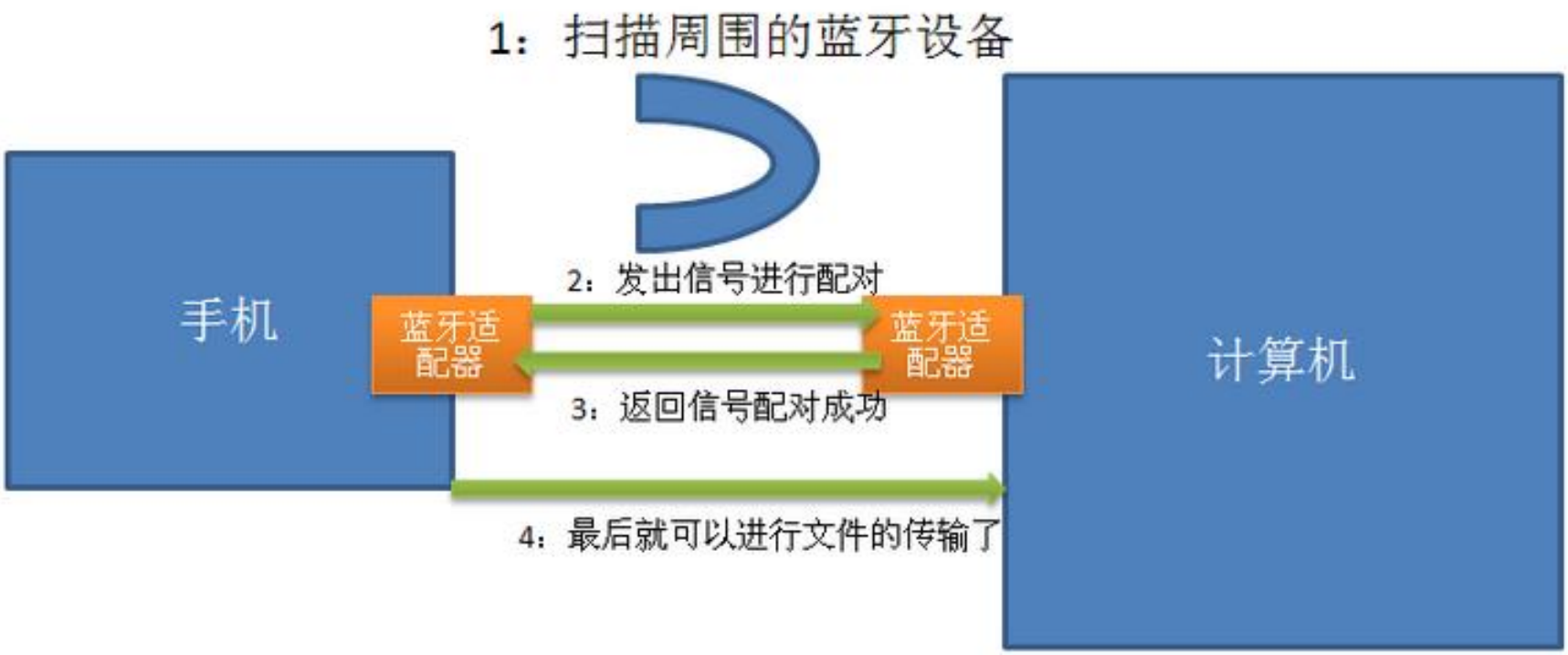
- 在我们的wifi 已经连通了以后，可以通过这个类获得一些已经连通的wifi 连接的信息获取当前链接的信息：
  - getBSSID() 获取BSSID
  - getDetailedStateOf() 获取客户端的连通性
  - getHiddenSSID() 获得SSID 是否被隐藏
  - getIpAddress() 获取IP 地址
  - getLinkSpeed() 获得连接的速度
  - getMacAddress() 获得Mac 地址
  - getRssi() 获得802.11n 网络的信号
  - getSSID() 获得SSID
  - getSupplicanState() 返回具体客户端状态的信息

Android WiFi编程实验

学习目标

- 通信方式简介
- HTTP网络访问
- WIFI编程
- 蓝牙编程

Bluetooth工作流程



Bluetooth工作流程

- 首先两个设备上都要有蓝牙设备或者专业一点叫蓝牙适配器，以手机和电脑为例,流程如下。
- 其次在手机上进行扫描，扫描周围蓝蓝牙设备，先找到手机附近的电脑，然后给它发出一个信号需要进行蓝牙的配对，再次返回一个信号说明手机和电脑已经配对成功了，最后配对成功后可以进行文件传输了。这是一个最基本的一个流程。

Android Bluetooth

- 1、BluetoothAdapter
  描述本地Bluetooth适配器（Bluetooth接收器）。
  BluetoothAdapter是所有Bluetooth相关活动的入口。运用BluetoothAdapter可以发现其他Bluetooth设备，查询连接（或配对）的设备列表，用已知MAC地址实例化一个 BluetoothDevice对象，创建一个BluetoothServerSocket对象侦听其他设备的通信。
- 2、BluetoothDevice
  描述一个远程Bluetooth设备。可以用它通过一个BluetoothSocket请求一个远程设备的连接，或者查询远程设备的名称、地址、类、连接状态等信息。
- 3、BluetoothSocket
  描述一个Bluetooth Socket接口（类似于TCP Socket）。应用通过InputStream和OutputStream.与另外一个Bluetooth设备交换数据，即它是应用与另外一个设备交换数据的连接点。

Android Bluetooth

- 4、BluetoothServerSocket
  描述一个开放的socket服务器，用来侦听连接进来的请求（类似于RCP ServerSocket）。为了连接两个Android设备，一个设备必须使用该类来开启一个socket做服务器，当另外一个设备对它发起连接请求时 并且请求被接受时，BluetoothServerSocket会返回一个连接的BluetoothSocket对象。
- 5、BluetoothClass
  描述一个Bluetooth设备的一般规格和功能。这个是用来定义设备类和它的服务的只读属性集。然而，它并不是可靠的描述设备支持的所有Bluetooth配置和服务，而只是一些设备类型的有用特征。

Android Bluetooth

- 6、BluetoothProfile
  描述Bluetooth Profile的接口。Bluetooth Profile是两个设备基于蓝牙通讯的无线接口描述。
  （对Bluetooth Profile的详细解释：为了更容易的保持Bluetooth设备之间的兼容，Bluetooth规范中定义了 Profile。Profile定义了设备如何实现一种连接或者应用，你可以把Profile理解为连接层或者应用层协议。 比如，如果一家公司希望它们的Bluetooth芯片支援所有的Bluetooth耳机，那么它只要支持HeadSet Profile即可，而无须考虑该芯片与其它Bluetooth设备的通讯与兼容性问题。如果你想购买Bluetooth产品，你应该了解你的应用需要哪 些Profile来完成，并且确保你购买的Bluetooth产品支持这些Profile。 ）
- 7、BluetoothHeadset
  提供移动电话的Bluetooth耳机支持。包括Bluetooth耳机和Hands-Free (v1.5) profiles。

Android Bluetooth

- 8、BluetoothA2dp
  定义两个设备间如何通过Bluetooth连接进行高质量的音频传输。
  A2DP（Advanced Audio Distribution Profile）：高级音频传输模式。
- 9、BluetoothProfile.ServiceListener
  一个接口描述，在与服务连接或者断连接的时候通知BluetoothProfile IPC（这是内部服务运行的一个特定的模式<profile>）。

Android Bluetooth

- 要使用Bluetooth功能，至少需要2个Bluetooth权限：BLUETOOTH和BLUETOOTH_ADMIN
- BLUETOOTH：用来授权任何Bluetooth通信，如请求连接，接受连接，传输数据等。
  BLUETOOTH_ADMIN：用来授权初始化设备搜索或操作Bluetooth设置。大多数应用需要它的唯一场合是用来搜索本地Bluetooth设备。本授权的其他功能不应该被使用，除非是需要修改Bluetooth设置的“power manager（电源管理）”应用。
- 注意：需要BLUETOOTH_ADMIN权限的场合，BLUETOOTH权限也是必需的。

Android Bluetooth权限

要在应用程序中对Android系统的蓝牙设备进行相关操作，需要在项目中的AndroidManifest.xml中添加：

```
<uses-permission
android:name="android.permission.BLUETOOTH"/>
<uses-permission
android:name="android.permission.BLUETOOTH_ADMIN" />
```

Bluetooth API（1）

- BluetoothAdapter 类

  1、BluetoothAdapter.getDefaultAdapter()：得到本地默认的 BluetoothAdapter，若返回为 null 则表示本地不支持蓝牙

  2、isDiscovering()：返回设备是否正在发现周围蓝牙设备

  3、cancelDiscovery()：取消正在发现远程蓝牙设备的过程

  4、startDiscovery()：开始发现过程

  5、getScanMode()：得到本地蓝牙设备的 Scan Mode

  6、getBondedDevices()：得到已配对的设备

  7、isEnabled()：蓝牙功能是否启用。

Bluetooth API（2）

- BluetoothDevice 类：对应远程的蓝牙 Device

  1、createRfcommSocketToServiceRecord()：创建该 Device 的 socket

- BluetoothSocket 类

  1、connect()：请求连接蓝牙

  2、getInputStream()：得到输入流，用于接收远程方信息

  3、getOutputStream()：得到输出流，发送给远程方的信息

  4、close()：关闭蓝牙连接。

- InputStream 类：

  1、read(byte[])：以阻塞方式读取输入流

- OutputStream 类：

  2、write(byte[])：将信息写入该输出流，发送给远程

BluetoothC API示例

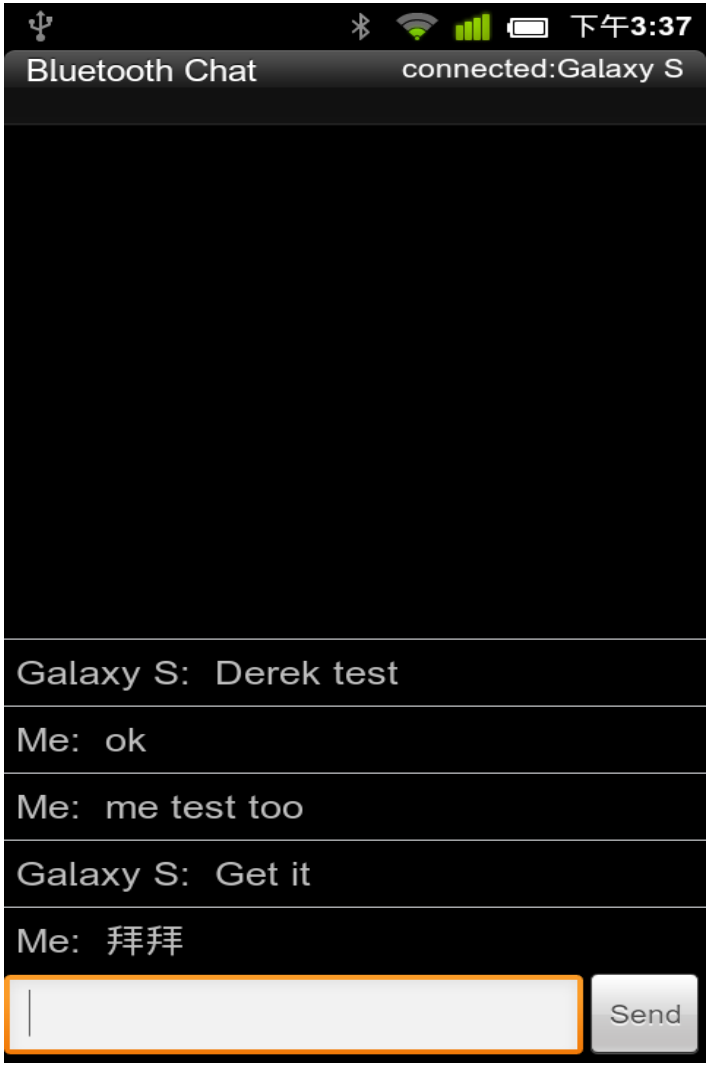
- 蓝牙功能未启用时，设置启用蓝牙：

```
if (! mBluetoothAdapter .isEnabled()) {
        Intent enableIntent = new Intent(BluetoothAdapter. ACTION_REQUEST_ENABLE );
        startActivityForResult(enableIntent, REQUEST_ENABLE_BT );
}
```

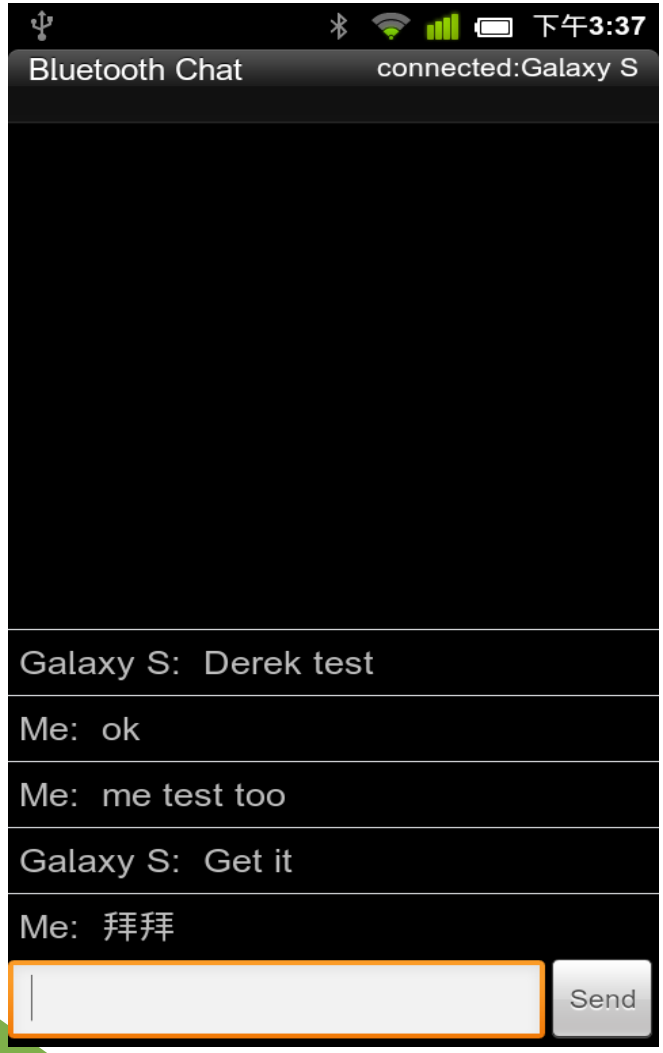
- 当设备没有打开对外可见模式，则传递 Intent 来打开可发现模式：

```
Intent discoverableIntent = new Intent(BluetoothAdapter. ACTION_REQUEST_DISCOVERABLE );
discoverableIntent.putExtra(BluetoothAdapter. EXTRA_DISCOVERABLE_DURATION , 300);
startActivity(discoverableIntent);
```

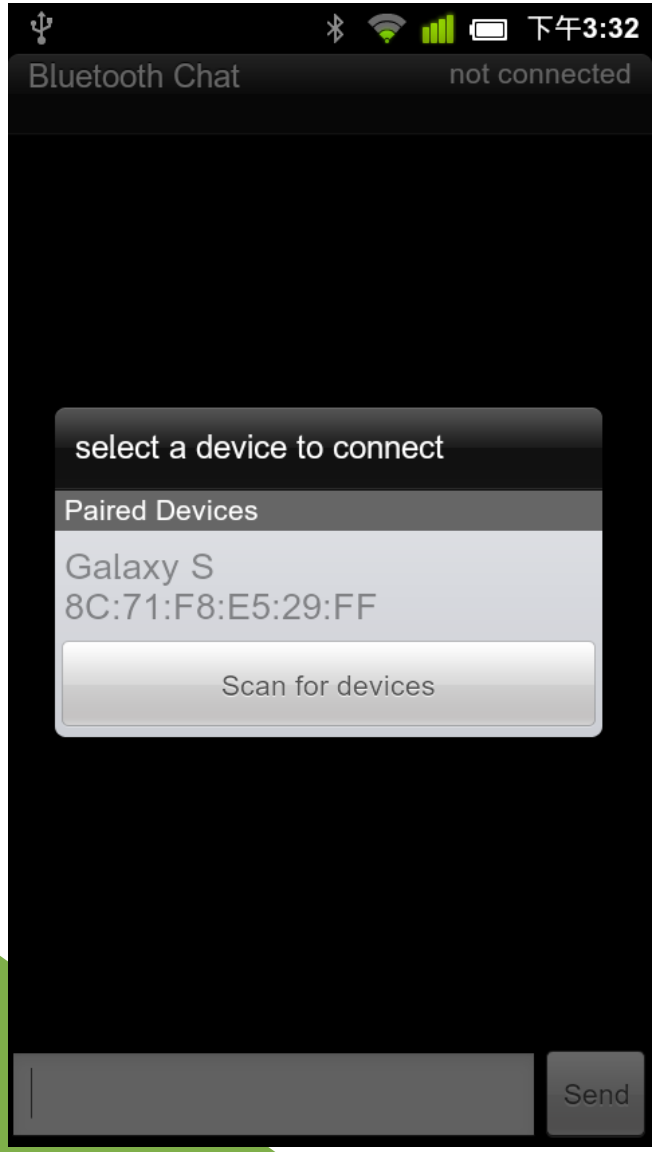
BluetoothChat面向对象设计分析

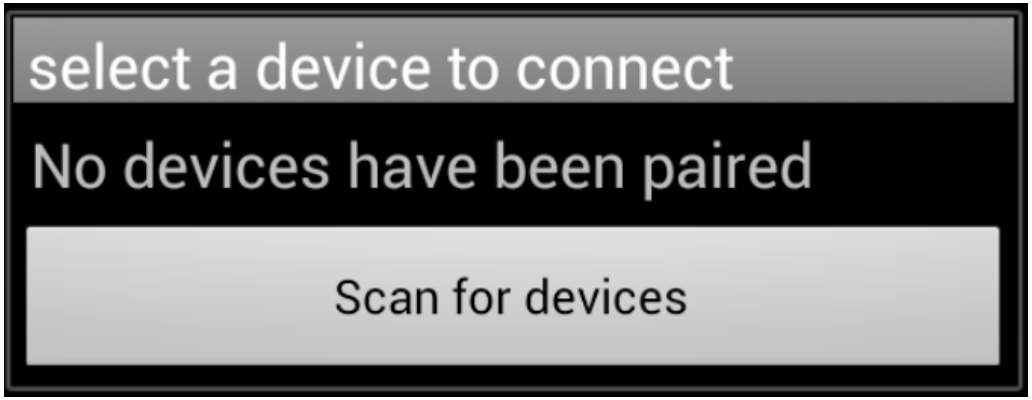


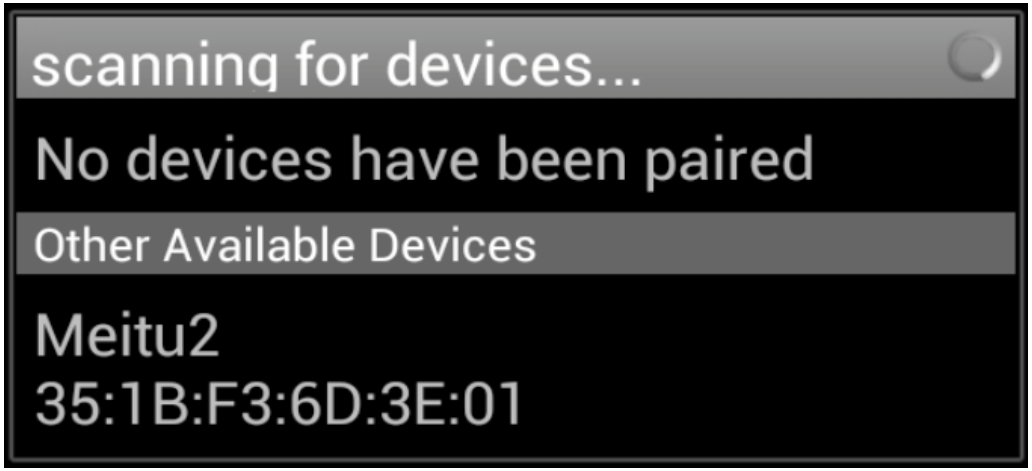
BluetoothChat主Activity



DeviceListActivity







BluetoothChat

- 主 Activity
- onCreate() 得到本地 BluetoothAdapter 设备，检查是否支持。
- onStart() 中检查是否启用蓝牙，并请求启用，然后执 setupChat() 。
- setupChat() 中先对界面中的控件进行初始化增加点击监听器等，然后创建 BluetoothChatService 对象，该对象在整个应用过程中存在，并管理蓝牙连接建立、消息发送接收等行为。

DeviceListActivity

得到系统默认蓝牙设备的已配对设备列表，并搜索出未配对的新设备的列表，然后提供点击后发出连接设备请求的功能。

BluetoothChatService

1、public synchronized void start()

开启 AcceptThread 线程，在这之前先检测 ConnectThread 和 ConnectedThread 线程是否运行，运行则先退出这些线程，更新聊天状态为：STATE_LISTEN

2、public synchronized void connect(BluetoothDevice device)

取消 CONNECTING 和 CONNECTED 状态下的相关线程，然后运行新的 ConnectThread连接线程，更新聊天状态为：STATE_CONNECTING

3、public synchronized void connected(BluetoothSocket socket, BluetoothDevice device)

开启一个 ConnectedThread 来管理对应的当前连接。之前先取消任意现存的 ConnectThread 、ConnectedThread 、 AcceptThread 线程，然后开启新的ConnectedThread读写线程，传入当前刚刚接受的socket连接，最后通过Handler来通知更新UI ，并传递蓝牙设备名称:MESSAGE_DEVICE_NAME

BluetoothChatService

4、public synchronized void stop()

停止所有相关线程，设当前状态为 STATE_NONE

5、public void write(byte[] out)

在 STATE_CONNECTED 状态下，调用mConnectedThread里的write方法，写入byte

6、private void connectionFailed()

连接失败的时候处理，更新聊天状态为：STATE_LISTEN，通知更新UI，并传递MESSAGE_TOAST，“无法连接设备”

7、private void connectionLost()

当失去连接的时候处理，更新聊天状态为：STATE_LISTEN，通知更新UI，并传递MESSAGE_TOAST，“设备连接丢失”

BluetoothChatService内部类

1、private class AcceptThread extends Thread

监听线程，准备接受新连接。使用阻塞方式，调用 BluetoothServerSocket.accept() ，提供 cancel 方法关闭 socket

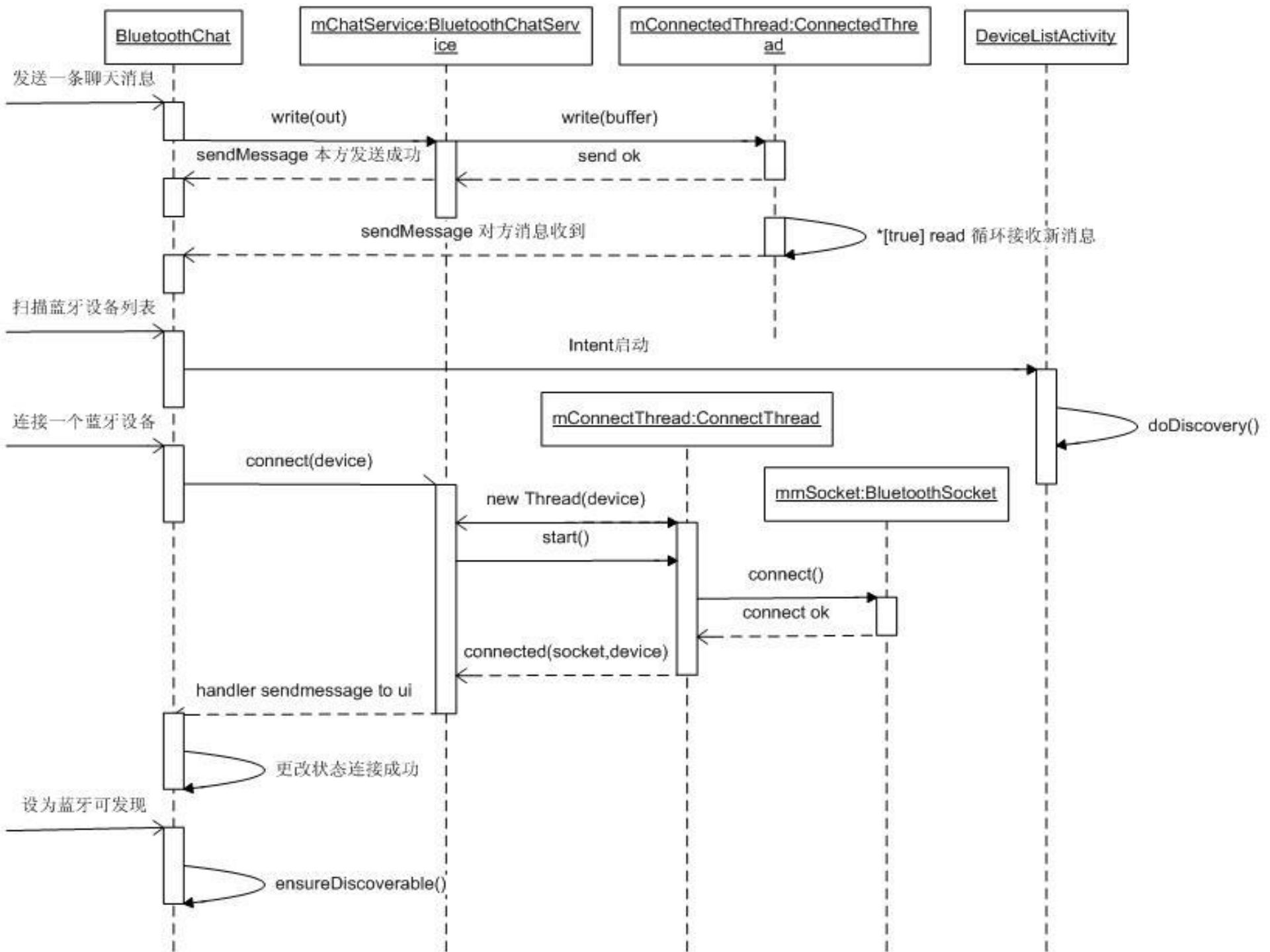
2、private class ConnectThread extends Thread

连接线程，用于对外发出连接对方蓝牙的请求处理流程。构造函数里通过BluetoothDevice.createRfcommSocketToServiceRecord() 从待连接的 device 产生 BluetoothSocket，然后在 run 方法中 进行 connect() ，如果出错，调用外部类的ConnectionFailed()进行处理，再关闭socket，并调用外部类的start()方法，重新启动监听线程；如果成功后，调用外部类的 connected() 方法，重新启动连接管理（读写）线程。同时，定义 cancel() 在关闭线程时能够关闭相关 socket 。

BluetoothChatService内部类

3、private class ConnectedThread extends Thread

双方蓝牙连接后运行的连接管理（读写）线程。构造函数中设置输入输出流。线程体中使用阻塞模式的 InputStream.read() 循环读取输入流， 然后发送UI更新消息MESSAGE_READ，如出现异常，则调用外部类的connectionLost()进行失去连接处理。也提供了write() 将聊天消息写入输出流传输至对方，并发送更新 UI消息MESSAGE_WRITE。同时，提供了 cancel() 关闭连接的 socket 。


BluetoothChat
mChatService:BluetoothChatService
mConnectedThread:ConnectedThread
DeviceListActivity
发送一条聊天消息
write(out)
write(buffer)
sendMessage 本方发送成功
send ok
sendMessage 对方消息收到
*[true] read 循环接收新消息
扫描蓝牙设备列表
Intent启动
doDiscovery()
连接一个蓝牙设备
mConnectThread:ConnectThread
connect(device)
mmSocket:BluetoothSocket
new Thread(device)
start()
connect()
connect ok
connected(socket,device)
handler sendmessage to ui
更改状态连接成功
设为蓝牙可发现
ensureDiscoverable()

QUESTIONS?