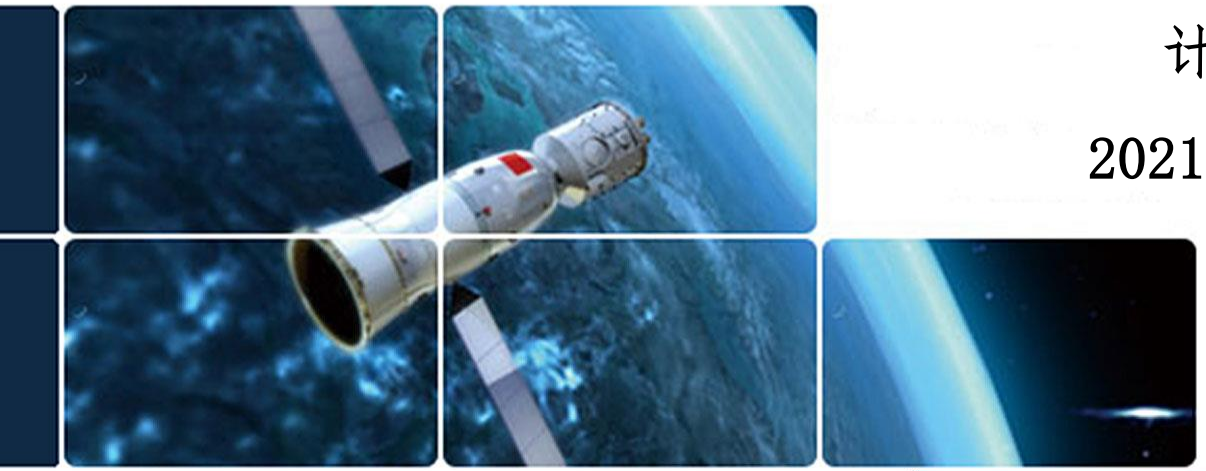
# 第8讲 移动互联网应用

# 网络通信技术

计算学部

2021年12月7日



# 案例中的网络连接问题




Q1：以什么方式连接?

A1：HTTP、TCP、UDP
需要点对点通信吗?

Q2：数据如何获取?

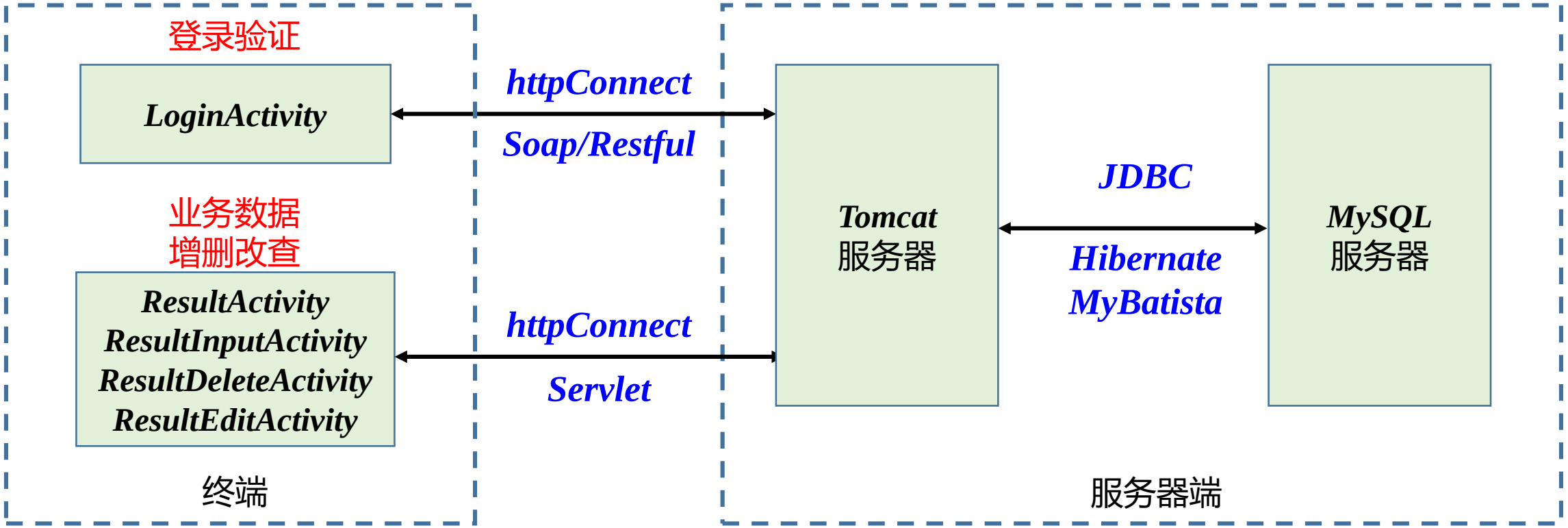
A2：直接访问数据库、通过数据服务访问数据库

Q3：服务器端的服务是什么?

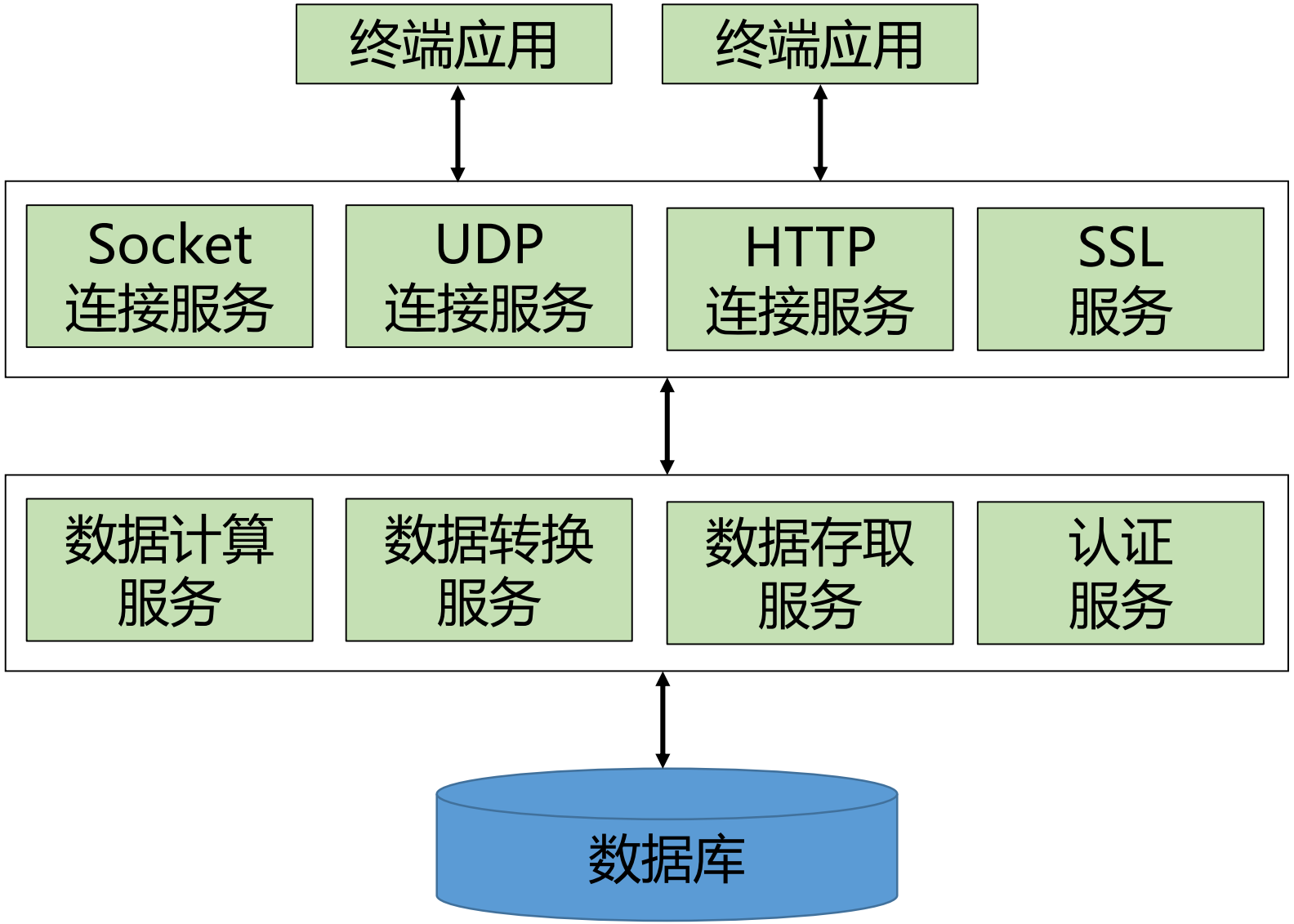
A3：Servlet、JSP或SOAP、Restful

如何验证用户　　业务数据来自哪里　　如何连接到服务

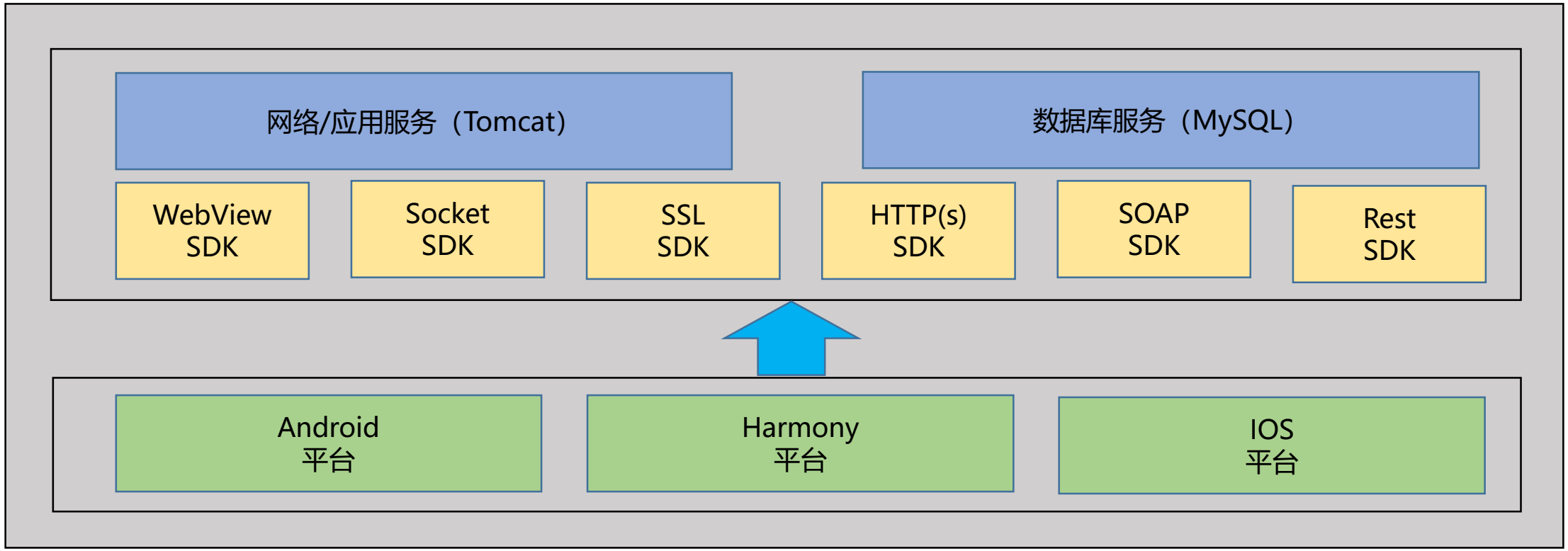
# 案例中的网络连接实现


登录验证
LoginActivity
业务数据
增删改查
ResultActivity
ResultInputActivity
ResultDeleteActivity
ResultEditActivity
终端
httpConnect
Soap/Restful
httpConnect
Servlet
Tomcat
服务器
JDBC
Hibernate
MyBatista
MySQL
服务器
服务器端

# 设计视图


终端应用
终端应用
Socket 连接服务
UDP 连接服务
HTTP 连接服务
SSL 服务
数据计算服务
数据转换服务
数据存取服务
认证服务
数据库

# 实现视图


网络/应用服务（Tomcat）
数据库服务（MySQL）
WebView SDK
Socket SDK
SSL SDK
HTTP(s) SDK
SOAP SDK
Rest SDK
Android 平台
Harmony 平台
IOS 平台

# 主要内容

- *WebView* 控件 – *onePage* 应用
- 访问*HTTP*资源
- *Socket*通信
- 数据报套接字和*SSL*
- *Harmony*平台的网络连接
- 蓝牙通信
- *Wifi* 通信

# WebView控件

- *WebView*是一个使用*WebKit*引擎的浏览器控件，可以用于显示本地或*Internet*上的网页。可将*WebView*当作一个完整的浏览器使用
- *WebView*支持*H5*、*HTML*、*CSS*等静态元素，也支持*JavaScript*，在*JavaScript*可以调用*Java*方法

# 用WebView控件浏览网页

1. 浏览网页是*WebView*的基本功能，通过*WebView*可直接装载任何有效网址

```
WebView webView = (WebView) findViewById(R.id.webview);
webView.loadUrl("http://www.hit.edu.cn");
```

2. *WebView*控件也可以浏览本地的网页文件或任何*WebView*支持的文件

```
String path = android.os.Environment.getExternalStorageDirectory().getPath();
String fileName1 = "file://" + path + "/test.html";
String fileName2 = "file://" + path + "/image.jpg";
webView.loadUrl(fileName2);
```

# 用WebView控件浏览网页

3. 除了可以浏览网页外，*WebView*也和大多数浏览器一样，可以缓存浏览历史网页，并可向前和向后浏览历史页面，也可以清除缓存内容

*webView.goBack()*
*webView.goForward()*
*webView.clearCache()*

**注意：** *WebView* 访问资源时，需要设置权限

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

# 用WebView控件装载HTML代码

4. ***WebView*** 控件可以直接装载 ***HTML***代码

   **(1)** ***public void loadData(String data, String mimeType , String encoding)***

   **(2)** ***public void loadDataWithBaseURL(String baseUrl, String data, String mimeType , String encoding, String historyUrl)***

5. ***WebView*** 默认情况下不支持*JavaScript*，如需支持*JavaScript*，需要进行设置

   - ***setJavaScriptEnabled(true);***
   - ***setWebChromeClient(new WebChromeClient())***，用于设置*JavaScript*处理器

# 主要内容

- ***WebView* 控件**
- 访问***HTTP***资源 – 大部分网络应用
- ***Socket***通信
- 数据报套接字和***SSL***
- ***Harmony***平台的网络连接
- 蓝牙通信
- ***Wifi*** 通信

# Http的访问基础

两个主要对象

- *HttpRequest*
- *HttpResponse*

两个主要方法

- *get()*
- *post()* → *formBody*

两个流

- *outputStream*：输出（发请求）
- *inputStream*：输入（看响应）

两种执行方式

- 同步
- 异步

# HttpURLConnection

- ***java.net.HttpURLConnection***类是访问***HTTP***资源的方式，***HttpURLConnection***完全取代了***HttpGet*** 和***HttpPost***，也是其他相关接口的基础（比如 ***okHttp*** 、 ***okGo***）

- 第一步：使用***java.net.URL***封装***HTTP***资源的***url***，使用***openConnection***方法获得***HttpUrlConnection***对象

```
URL url = new URL("http://www.hit.edu.cn/index.html");
HttpURLConnection httpURLConnection =
                    (HttpURLConnection)url.openConnection();
```

- 第二步：设置请求的方法，例如***GET***、***POST***等

```
httpURLConnection.setRequestMethod("POST");
```

- 第三步：设置输入输出及其它开关。如果要读取***HTTP***资源和向服务器端上传数据，需要将***setDoInput***方法参数设置为***true***，将***setDoOutput***方法参数设置为***true***

```
httpURLConnection.setDoInput(true);
httpURLConnection.setDoOutput(true);
httpURLConnection.setUseCaches(true|false); // 禁止使用缓存
```

- 第四步：设置***HTTP***请求头

```
httpURLConnection.setRequestProperty("Charset","UTF-8");
```

# HttpURLConnection

- 第五步：输入和输出数据。这是对*HTTP*资源的读写操作，也就是通过*InputStream*和*OutputStream*读取和写入数据

```
InputStream is = httpURLConnection.getInputStream();

OutputStream os = httpURLConnection.getOutputStream();
```

- 第六步：关闭输入输出流

```
is.close();

os.close();
```

## *Get/Post* 请求

```
OkHttpClient client = new OkHttpClient();

Request request = new Request.Builder()
                .get()
                .url("https:www.baidu.com")
                .build();

Call call = client.newCall(request);
```

```
FormBody formBody = new FormBody.Builder()
                .add("username", "admin")
                .add("password", "admin")
                .build();

final Request request = new Request.Builder()
                .url("http://www.jianshu.com/")
                .post(formBody)
                .build();
```

## 同步/异步 请求

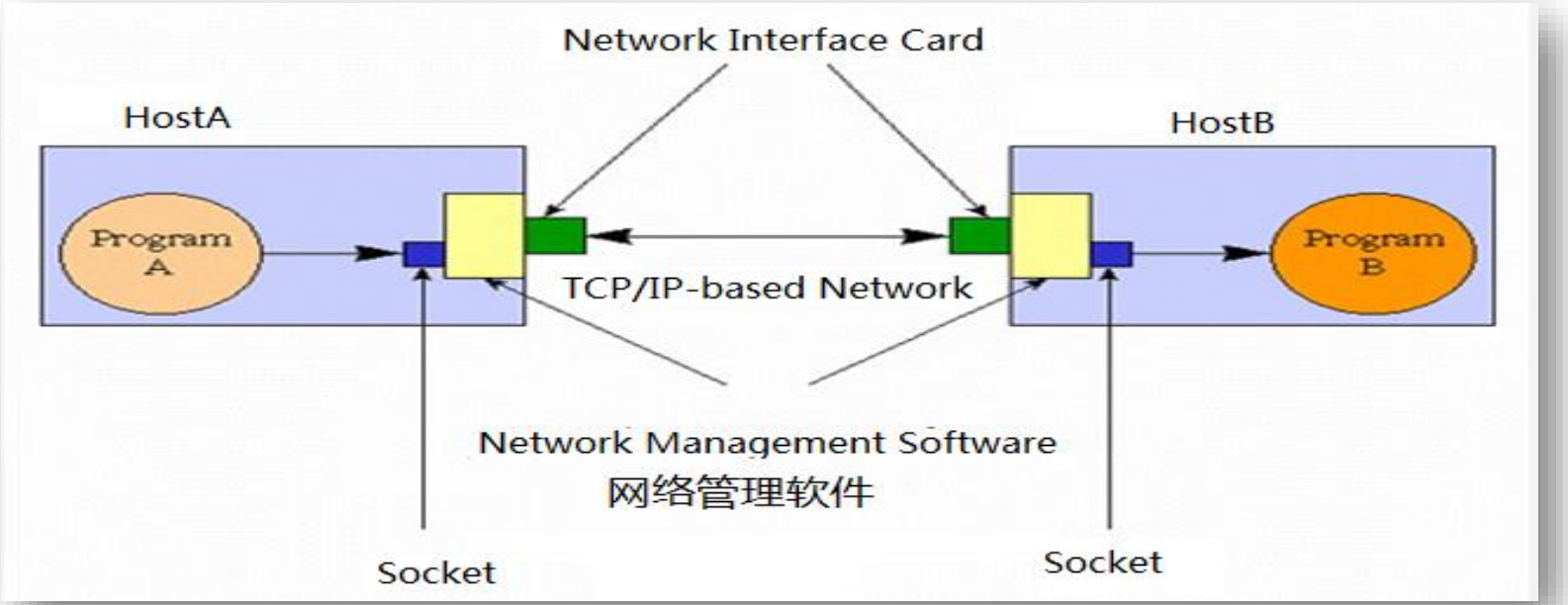
```
//同步调用,返回Response,会抛出IO异常
Response response = call.execute();

//异步调用,并设置回调函数
call.enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Toast.makeText(OkHttpActivity.this, "get failed", Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onResponse(Call call, final Response response) throws IOException {
        final String res = response.body().string();
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                contentTv.setText(res);
            }
        });
    }
});
```

# 主要内容

- ***WebView* 控件**
- 访问*HTTP*资源
- ***Socket*通信 – 端到端通信**
- 数据报套接字和*SSL*
- ***Harmony*平台的网络连接**
- 蓝牙通信
- ***Wifi* 通信**

# 套接字（Socket）的概念和原理

- ***Socket***可以看成在两个程序进行通讯连接中的一个端点，程序A将信息写入***Socket***，该***Socket***将信息发送给另一个***Socket***，使这段信息能传送到程序B




- Host A上的程序A将信息写入***Socket***， 被Host A的网络管理软件访问，并将信息通过网络接口卡发送到Host B， Host B网络接口卡接收信息后传送给其网络管理软件，再将信息保存到Host B的***Socket***，程序B从***Socket***读取信息

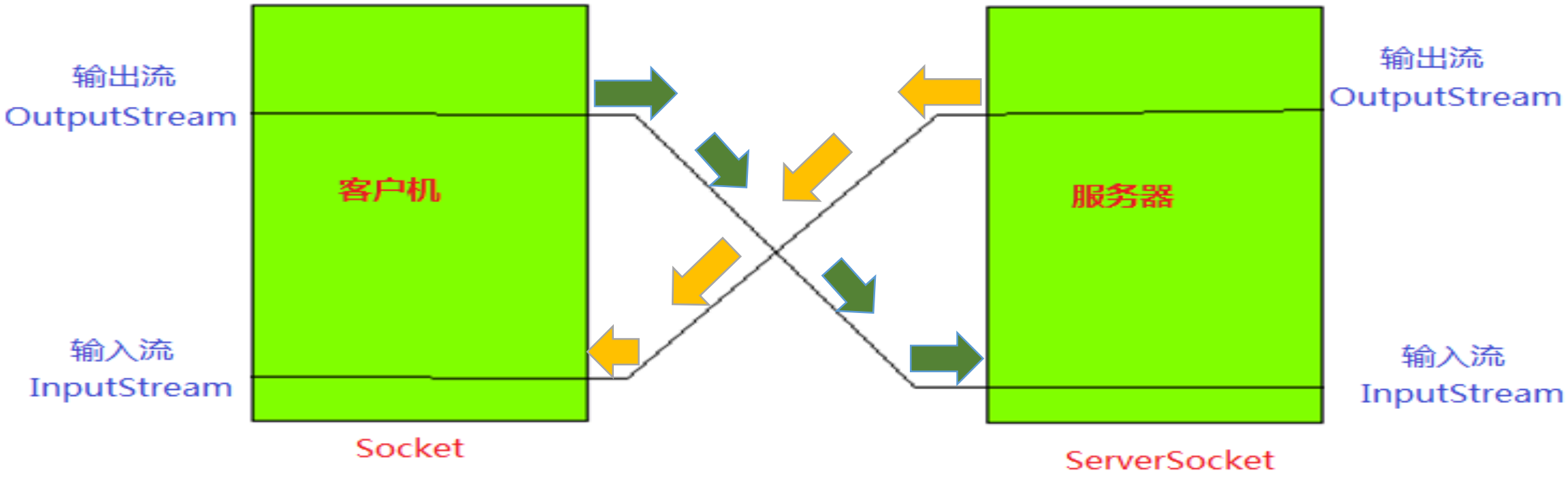
# 套接字（Socket）类

- ***Socket***构造函数，用于创建基于***Socket***的连接服务器端套接字的套接字
  - *Socket(InetAddress addr, int port)*
  - *Socket(String host, int port)*

- ***InetAddress***对象通过***addr***参数获得服务器主机IP地址，如果无IP地址与host匹配，抛出***UnknownHostException***异常，port表示服务器端口号

- 如果创建了***Socket***对象，可通过 *getInputStream()*方法获得输入流传送来的信息，也可通过 *getOutputStream()*方法获得输出流，实现消息发送

# 套接字（Socket）类


输出流
OutputStream
客户机
输入流
InputStream
Socket
服务器
输出流
OutputStream
输入流
InputStream
ServerSocket

# 连接服务器

- 客户端可以通过两种方式连接服务器：①IP方式；②域名方式
- 通过***Socket***类构造函数可将IP(或域名)及端口号作为参数传入***Socket***对象

```
Socket socket = new Socket(www.csdn.net,80);
```

- 还可通过*Socket.connect*方法指定连接参数

```
SocketAddress socketAddress = new InetSocketAddress("192.168.0.5",80);
Socket socket = new Socket();
socket.connect(socketAddress,50);  //50表示超时参数，单位ms
```

# 发送和接收数据

- *Socket.getInputStream*
- *Socket.getOutputStream*
- 分别获得***InputStream***和***OutputStream***对象，前者读取服务器端向客户端发送的数据，后者允许客户端向服务器端发送数据

- ***OutputStream***，***OutputStreamWriter***， ***BufferedWriter***的区别
  - ***OutputStream***表示输出字节流所有类的超类
    一般使用其子类***FileOutputStream***
  - ***OutputStreamWriter***：将写入字符编码成字节后写入字节流
  - ***BufferedWriter***：将文本先存储到缓冲区，加快字符写入的速度

- ***IutputStream***，***IutputStreamReader***， ***BufferedReader***的区别
  - ***IutputStream***表示输入字节流所有类的超类
    一般使用其子类***FileIutputStream***
  - ***IutputStreamReader***:将字节流中字节编码为字符
  - ***BufferedReader***：字符串缓冲读取类，用于加快读取字符速度

## 发送请求

```
Socket socket = new Socket("www.hit.edu.cn", 80);
OutputStream os = socket.getOutputStream(); ③输出字节流
OutputStreamWriter osw = new OutputStreamWriter(os); ②输出字符转字节
BufferedWriter bw = new BufferedWriter(osw) ①缓冲区写入字符
bw.write("GET / HTTP/1.1\r\nHost:www.hit.edu.cn\r\n\r\n");
bw.flush();
```

## 接收数据

```
InputStream is = socket.getInputStream(); ①读输入字节流
InputStreamReader isr = new InputStreamReader(is) ②字节转字符
BufferedReader br = new BufferedReader(isr); ③字符缓冲区输出
String s = "";
TextView textView = (TextView) findViewById(R.id.textview);
while ((s = br.readLine()) != null)
    textView.append(s + "\n");
socket.close();
```

# Socket选项

- 通过*Socket*选项可以指定*Socket*发送和接收数据的方式，比较常用的有8个选项（*java.net.SocketOptions*接口定义），可用***get***和***set***方法获得和设置这些选项
  - ***TCP_NODELAY***：延时设置
  - ***SO_REUSEADDR***：地址重用设置
  - ***SO_LINGER***：阻塞设置
  - ***SO_TIMEOUT***：超时设置
  - ***SO_SNDBUF***：发送缓冲区设置
  - ***SO_RCVBUF***：接收缓冲区设置
  - ***SO_KEEPALIVE***：活跃状态设置
  - ***SO_OOBINLINE***：紧急包发送设置

# TCP_NODELAY选项

- 默认情况下，客户端向服务器端发送数据时，会根据数据包的大小决定是否立即发送，当数据包中的数据很少时，系统会在发送前将小包合并到较大的包，然后一起发送出去，发送下一数据包时，系统会等待服务器对上一数据包的响应，当收到响应后再发送下一个数据包

- 对于网络速度较慢、实时性要求较高的场合，上述方式会使得客户端有明显的停顿现象，可使用***setTcpTodelay***和 ***setTcpNoDelay***实现控制

  *public boolean* ***getTcpNoDelay()***

  *public void* ***setTcpNoDelay(boolean on)***

# SO_REUSEADDR选项

- 通过该选项，可以使的多个***Socket***对象绑定在同一个端口，对于绑定在固定端口上的***Socket***可能抛出异常，使用该选项可避免抛出异常

  *public boolean* ***getReuseAddress****()*

  *public void* ***setReuseAddress(boolean on)***

- 使用该选项需要注意的事项
  - 必须在调用***bind***之前调用***set***方法打开该选项
  - 必须将绑定同一端口的所有***Socket***对象的***SO_REUSEADDR***选项打开

# SO_LINGER选项

- 该选项影响***Socket.close()***方法，默认情况下，调用***close***方法后，系统将立即返回。若此时还有未被送出去数据包，将被丢弃，如果将***linger***参数设置为某正数值n，在调用***close***方法后，系统将最多被阻塞n秒，n秒内系统尽量将未发送出去的数据包发送出去，如果超过n秒后还有未发送的数据包，这些数据包将全部丢失，***close***方法立即返回。linger设置为0与关闭***SO_LINGER***作用一样

  *public int* ***getSoLinger()***

  *public void* ***setSoLinger(boolean on, int linger)***

- 若底层***Socket***不支持***SO_LINGER***，会抛出异常，*linger*参数设为负数也会抛出异常。可以通过***getSoLinger***方法得到延迟关闭的时间，若返回-1，表明***SO_LINGER***关闭

# SO_TIMEOUT选项

- 通过该选项设置读数超时，当输入流的*read*方法被阻塞，如果设置*timeout*（单位为毫秒），那么系统等待*timeout*毫秒后会抛出*InterruptedIOException*异常

  *public int* ***getSoTimeout()***

  *public void* ***setSoTimeout(int timeout)***

- 如果将*timeout*参数设置为0，***read***方法会无限等待，直到服务端关闭***Socket***

- 当底层***Socket***不支持***SO_TIMEOUT***选项时，***get***和***set***方法均会抛出异常

# SO_SNDBUF选项

- 默认情况下，输出流的发送缓冲区是4096Byte，如该默认值不能满足要求，可用***setSendBufferSize***重新设置缓冲区大小，若将输出缓冲区设置过小，会导致数据传输过于频繁，降低网络传输效率

  *public int* ***getSendBufferSize****()*

  *public void* ***setSendBufferSize(int size)***

- 若底层***Socket***不支持***SO_SNDBUF***选项，***get***和***set***方法就会抛出异常

# SO_RCVBUF选项

- 默认情况下，输入流缓冲区大小为4096Byte，如果该默认值不能满足要求，可用***setReceiveBufferSize***重新设置缓冲区大小，若将输入缓冲区设置过小，会导致数据传输过于频繁，从而降低网络传输的效率

  *public int* ***getReceiveBufferSize****()*

  *public void* ***setReceiveBufferSize(int size)***

- 若底层***Socket***不支持***SO_RCVBUF***选项，***get***和***set***方法就会抛出异常

# SO_KEEPALIVE选项

- 将该选项打开，客户端*Socket*每隔一段时间就会利用空闲连接向服务器发送一个数据包，该数据包是为了监测服务器是否还处于活动状态，如果服务器未响应该数据包，一段时间后，客户端*Socket*再发送一个数据包。如果在一段时间内，服务器还没响应，客户端*Socket*将关闭

  *public int* ***getKeepAlive()***

  *public void* ***setKeepAlive(boolean on)***

- 若将该选项关闭，客户端*Socket*在服务器无效的情况下可能长时间不会关闭，该选项默认关闭

# SO_OOBINLINE选项

- 该选项打开，可通过***Socket.sendUrgentData***向服务器发送一个单字节数据，该单字节数据不需要经过数据缓冲区立即发出

  *public int* ***getOOBInLine()***

  *public void* ***setOOBInLine(boolean on)***

  *public void* ***sendUrgentData(int data)***

- 该选项必须在客户端和服务器端同时打开方可生效，否则无法正常使用

# 服务端Socket

- ***ServerSocket***对象用于将服务端IP地址和端口绑定，客户端可以通过IP地址和端口访问服务端程序，因此***ServerSocket***也叫服务端***Socket***，主要用于实现服务器程序

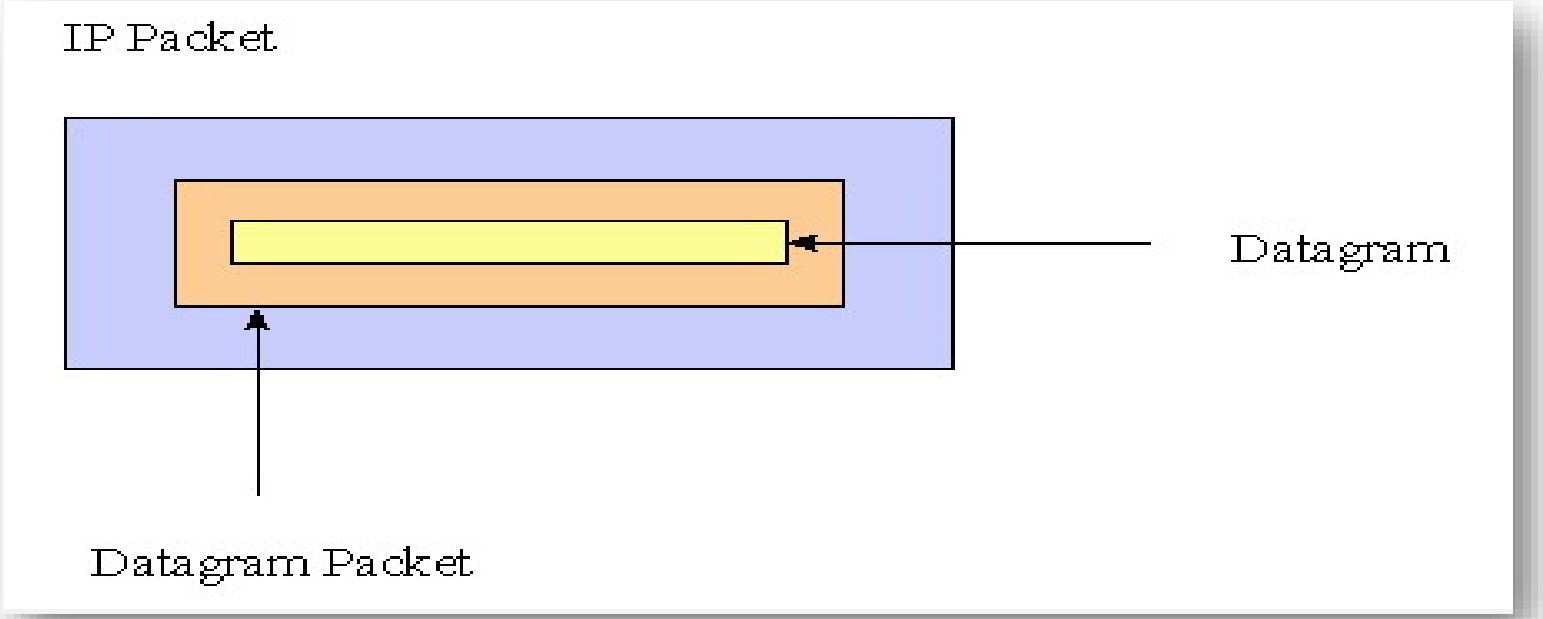
- 服务器程序一般部署在高端服务器或者***PC***上，但随着手机性能不断提高，也可将计算量不大的服务程序部署在手机上，作为移动服务器。部署服务的手机可通过IP地址与其它的计算机和智能设备进行通信

- ***ServerSocket***的应用包括：手机服务器、程序之间的通信等

# 主要内容

- ***WebView* 控件**
- 访问*HTTP*资源
- ***Socket*通信**
- 数据报套接字和*SSL* – 异步端到端通信
- ***Harmony*平台的网络连接**
- 蓝牙通信
- ***Wifi* 通信**

- ***Socket***连接建立需要一定时间开销，为减少这种开销，可使用数据报套接字(***datagram socket***)通信，又称为自寻址套接字，使用***UDP***发送寻址信息，通过数据报套接字可发送多***IP***包，地址信息包含在数据报文中，数据报又包含在***IP***包内



# 数据报套接字

- 数据报套接字包括三个类：
  - ***DatagramPacket***:表示存放数据的数据报，包括地址信息
  - ***DatagramSocket***: 表示接收和发送数据报的套接字
  - ***MulticastSocket***: 能进行多点传送的数据报套接字

- 在使用数据报之前，地址信息和数据报以字节数组的方式同时压入***DatagramPacket***类创建的对象中

- 发送数据的数据报需设定发送地址，接收数据的数据报无需设定地址

# 使用数据报套接字

第一步：创建**DatagramPacket**，数据包含在第一个参数当中

*DatagramPacket(byte [] data,int offset,int length,InetAddress remoteAddr,int remotePort)*

第二步：创建**DatagramSocket**，返回一个UDP套接字

*DatagramSocket(int localPort)*

第三步：发送和接受。*send()*方法用于发送**DatagramPacket**，一旦创建连接，数据报将发送到该套接字所连接的地址；*receive()*方法会阻塞等待，直到接收到数据报文，并将报文中的数据复制到指定的**DatagramPacket**实例

*void send(DatagramPacket packet)*

*void receive(DatagramPacket packet)*

# Android的SSL通信

- *Android SDK*实现了通用***SSL***功能。具体应用时，将证书文件和***CA***信任列表加载到对应***KeyStore***中，用***KeyStore***初始化需要打开的***SSLContext***，然后建立连接，使用***BufferReader***和***BufferWriter***就能实现SSL加密传输

- ***Android***的私钥和信任证书的格式必须是***BKS***格式的，通过配置本地***JDK***，***keytool***可以生成***JKS/BKS***格式的私钥和信任证书

- 双向消息认证配置要求

***Server***：

1) ***KeyStore***：保存服务端的私钥
2) ***Trust KeyStore***：保存客户端的授权证书

***Clinet***：

1) ***KeyStore***：保存客户端的私钥
2) ***Trust KeyStore***：保存服务端的授权证书

- 生成密钥文件

```
keytool –import –alias serverkey –file server.crt –keystore tclient.keystore
keytool -genkey -alias clientkey -keystore kclient.keystore
keytool -export -alias clientkey -keystore kclient.keystore -file client.crt
keytool -import -alias clientkey -file client.crt -keystore tserver.keystore
```

# 主要内容

- ***WebView* 控件**
- 访问*HTTP*资源
- ***Socket*通信**
- 数据报套接字和*SSL*
- ***Harmony*平台的网络连接**
- 蓝牙通信
- ***Wifi* 通信**

*HarmonyOS* 网络管理模块主要提供以下功能：

1. 数据连接管理：网卡绑定，打开URL，数据链路参数查询
2. 数据网络管理：指定数据网络传输，获取数据网络状态变更，数据网络状态查询
3. 流量统计：获取蜂窝网络、所有网卡、指定应用或指定网卡的数据流量统计值
4. HTTP 缓存：有效管理HTTP缓存，减少数据流量

| 权限名 | 权限描述 |
| --- | --- |
| ohos.permission.GET_NETWORK_INFO | 获取网络连接信息。 |
| ohos.permission.SET_NETWORK_INFO | 修改网络连接状态。 |
| ohos.permission.INTERNET | 允许程序打开网络套接字，进行网络连接。 |

# 使用当前网络打开URL连接

使用当前连接打开一个URL链接需要的接口：

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| *NetManager* | *getInstance(Context context*) | 获取网络管理的实例对象 |
| *hasDefaultNet()* | 查询当前是否有默认可用的数据网络 | |
| *getDefaultNet()* | 获取当前默认的数据网络句柄 | |
| *addDefaultNetStatusCallback(NetStatusCallback callback*) | 获取当前默认的数据网络状态变化 | |
| *setAppNet(NetHandle netHandle*) | 应用绑定该数据网络 | |
| *NetHandle* | *openConnection(URL url*, *Proxy proxy*) *throws IOException* | 使用该网络打开一个 URL 链接 |

# 使用当前网络打开URL连接

1.调用 *NetManager.getInstance(Context)*获取网络管理的实例对象

2.调用 *NetManager.getDefaultNet()*获取默认的数据网络

3.调用 *NetHandle.openConnection()*打开一个 *URL*

4.通过 *URL* 链接实例访问网站

```
1    NetManager netManager = NetManager.getInstance(null);
2     if (!netManager.hasDefaultNet()) {
3.        return;
4.    }
5.    NetHandle netHandle = netManager.getDefaultNet();
6.   // 可以获取网络状态的变化
7.   NetStatusCallback callback = new NetStatusCallback() {
8.   // 重写需要获取的网络状态变化的override函数
9.   }
10.   netManager.addDefaultNetStatusCallback(callback);
```

```
11   // 通过openConnection来获取URLConnection
12   try {
13       HttpURLConnection connection = null;
14       String urlString = "http://www.hit.edu.cn/";
15       URL url = new URL(urlString);
16       URLConnection urlConnection = netHandle.
17             openConnection (url, java.net.Proxy.NO_PROXY);
18       if (urlConnection instanceof HttpURLConnection) {
19             connection = (HttpURLConnection) urlConnection;
20        }
21       connection.setRequestMethod("GET");
22       connection.connect();
23       // 之后可进行url的其他操作
24   } finally {
25         connection.disconnect();
26   }
```

# 使用当前网络进行Socket数据传输

使用当前连接进行***Socket*** 数据传输需要如下接口：

| 类名 | 接口名 | 功能描述 |
| --- | --- | --- |
| *NetManager* | *getByName(String host)* | 解析主机名，获取其 *IP* 地址 |
| *bindSocket(Socket socket)* | 绑定 *Socket* 到该数据网络 | |
| *NetHandle* | *bindSocket(DatagramSocket socket)* | 绑定 *DatagramSocket* 到该数据网络 |

# 使用当前网络进行Socket数据传输

1.调用 ***NetManager.getInstance(Context)***获取网络管理的实例对象

2.调用 ***NetManager.getDefaultNet()***获取默认的数据网络

3.调用 ***NetHandle.bindSocket()***绑定网络

4.使用 ***Socket*** 发送数据

```
1.  NetManager netManager = NetManager.getInstance(null);
2.  if (!netManager.hasDefaultNet()) {
3.      return;
4.   }
5.   NetHandle netHandle = netManager.getDefaultNet();
```

```
6.  // 通过Socket绑定来进行数据传输
7.  try {
8.     InetAddress address = netHandle.getByName("www.hit.edu.cn");
9.     DatagramSocket socket = new DatagramSocket();
10.    netHandle.bindSocket(socket);
11.    byte[] buffer = new byte[1024];
12.    DatagramPacket request = new DatagramPacket(buffer, buffer.length,
                                                   address, port);
13.  // buffer赋值
14.  // 发送数据
15. socket.send(request);
16. } catch(IOException e) {
17. e.printStackTrace();
18. }
```

# 流量统计

| 接口名 | 功能描述 |
| --- | --- |
| *getCellularRxBytes()* | 获取蜂窝数据网络的下行流量 |
| *getCellularTxBytes()* | 获取蜂窝数据网络的上行流量 |
| *getAllRxBytes()* | 获取所有网卡的下行流量 |
| *getAllTxBytes()* | 获取所有网卡的上行流量 |
| *getUidRxBytes(int uid)* | 获取指定UID的下行流量 |
| *getUidTxBytes(int uid)* | 获取指定UID的上行流量 |
| *getIfaceRxBytes(String nic)* | 获取指定网卡的下行流量 |
| *getIfaceTxBytes(String nic)* | 获取指定网卡的上行流量 |

# 管理HTTP缓存

- 重复打开一个相同网页时，可优先从缓存文件里读取内容，减少数据流量，降低设备功耗，提升应用性能
- 管理 HTTP 缓存的功能主要由 HttpResponseCache 类提供

| 接口名 | 功能描述 |
| --- | --- |
| *install(File directory, long size)* | 使能 HTTP 缓存，设置缓存保存目录及大小 |
| *getInstalled()* | 获取缓存实例 |
| *flush()* | 立即保存缓存信息到文件系统中 |
| *close()* | 关闭缓存功能 |
| *delete()* | 关闭并清除缓存内容 |

# 主要内容

- ***WebView* 控件**
- 访问*HTTP*资源
- ***Socket*通信**
- 数据报套接字和*SSL*
- ***Harmony*平台的网络连接**
- 蓝牙通信
- ***Wifi* 通信**

# 认识蓝牙通信技术

- 蓝牙是一种重要的短距离无线通信协议
- 蓝牙采用工作频率2.4GHz，ISM（即工业、科学和医学）频段
- 蓝牙协议分为4层：即核心协议层、电缆替代协议层、电话控制协议层和其它协议层，最重要的是核心协议层

# Android平台的蓝牙编程

- ***Android*** 平台支持蓝牙网络协议栈，实现蓝牙设备之间的数据传输

- 蓝牙设备之间的通信主要包括了四个步骤：
  - 设置蓝牙设备
  - 寻找局域网内可能或者匹配的设备
  - 连接设备
  - 实现设备之间的数据传输

- ***Android***所有关于蓝牙开发的类都在***android.bluetooth***包中

# BluetoothAdapter类

- 代表一个本地蓝牙适配器，是蓝牙交互的入口点
- ***BluetoothAdapter***的作用
  - (1) 发现其他蓝牙设备
  - (2) 查询绑定了的设备
  - (3) 使用已知MAC地址实例化一个蓝牙设备
  - (4) 建立一个***BluetoothServerSocket***

# BluetoothAdapter类

- ***BluetoothAdapter***的方法很多，常用方法包括：
  - ***getDefaultAdapter()***获取默认BluetoothAdapter
  - ***getState()***获取本地蓝牙适配器当前状态
  - ***enable()***打开蓝牙
  - ***getAddress()***获取本地蓝牙地址
  - ***getName()***获取本地蓝牙名称
  - ***getRemoteDevice(String address)***根据蓝牙地址获取远程蓝牙设备
  - ***isDiscovering()***判断当前是否正在查找设备，是返回true
  - ***isEnabled()***判断蓝牙是否打开，已打开返回true，否则返回false
  - ***cancelDiscovery()***，取消发现，不再继续搜索设备
  - ***disable()***关闭蓝牙

# BluetoothDevice类

- ***BluetoothDevice***代表一个远端的蓝牙设备，使用它可请求远端蓝牙设备连接或获取远端蓝牙设备的名称、地址、种类和绑定状态（其信息封装在***bluetoothSocket***中）
- 获取***BluetoothDevice***的目的就是要创建***BluetoothSocket***对象
- ***createRfcommSocketToServiceRecord(UUID uuid)***根据UUID创建并返回一个***BluetoothSocket***

# BluetoothSocket类

- ***BluetoothSocket***代表一个蓝牙套接字（类似于***Socket***），是应用程序通过输入、输出流与其他蓝牙设备通信的连接点
- 一共5个方法
  - ***connect()***，与服务器建立连接
  - ***getInptuStream()***，获取输入流
  - ***getOutputStream()***，获取输出流
  - ***getRemoteDevice()***，获取远程设备，获取***bluetoothSocket***指定连接的那个远程蓝牙设备
  - ***close()***，与服务器断开连接

# BluetoothServerSocket类

- ***BluetoothServerSocket***表示服务器套接字，用于打开服务连接，监听可能到来的连接请求。为了连接两个蓝牙设备，必须有一个设备作为服务器打开一个服务套接字。当远端设备发起连接请求，并已连接成功，***BluetoothServerSocket***类将会返回一个***BluetoothSocket***

- 包括三个方法
  - ***accept()***和***accept(int timeout)***，返回一个***BluetoothSocket***
  - ***close()***关闭连接

# 打开和关闭蓝牙设备

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

//获取本地蓝牙适配器
BluetoothAdapter bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();


//请求打开蓝牙设备
Intent enableIntent = new
                    Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
startActivityForResult(enableIntent, 1);
```

# 搜索蓝牙设备

- 与其它蓝牙设备通信之前需要搜索周围的蓝牙设备
- 如设备需要被其它蓝牙设备搜索到，需要进入蓝牙设置界面进行相关设置
- 如手机已和某设备绑定，***getBondedDevices***方法可获得已经绑定的蓝牙设备列表
- ***startDiscodery***方法可用于搜索周围的蓝牙设备，搜到的蓝牙设备通过广播返回，因此需要注册广播接收器来获得已经搜索到的蓝牙设备

# 蓝牙数据传输

- 蓝牙传输数据的方式与***Socket***类似，在网络中使用***Socket***和***ServerSocket***控制客户端和服务端的数据读写
- 蓝牙通信也由客户端和服务端***Socket***完成
  - 蓝牙客户端***Socket***是***BluetoothSocket***
  - 蓝牙服务端***Socket***是***BluetoothServerSocket***
  - 两个类都位于***android.bluetooth***包中
  - 无论是客户端***Socket***还是服务器端***Socket***，都需要一个***UUID***进行标识

# Harmony平台蓝牙支持

- ***HarmonyOS*** 将蓝牙主要分为传统蓝牙和低功耗蓝牙（通常称为 ***BLE***，***Bluetooth Low Energy***）。传统蓝牙指版本 3.0 以下蓝牙，低功耗蓝牙指版本 4.0 以上蓝牙

- 蓝牙配对方式有两种：蓝牙协议 2.0 以下支持 PIN 码（Personal Identification Number，个人识别码）配对，蓝牙协议 2.1 以上支持简单配对

- 蓝牙打开需要 ***ohos.permission.USE_BLUETOOTH*** 权限
- 蓝牙扫描需要***ohos.permission.LOCATION***权限

  和***ohos.permission.DISCOVER_BLUETOOTH***权限

## *HarmonyOS* 传统蓝牙

（1）传统蓝牙本机管理：打开和关闭蓝牙、设置和获取本机蓝牙名称、扫描和取消扫描周边蓝牙设备、获取本机蓝牙对其他设备的连接状态、获取本机蓝牙已配对的蓝牙设备列表

（2）传统蓝牙远端设备操作：查询远端蓝牙设备名称和 MAC 地址、设备类型和配对状态，以及向远端蓝牙设备发起配对

## *HarmonyOS* 低功耗蓝牙

（1）区分中心设备和外围设备：前者负责扫描外围设备和发现广播，后者负责发送广播

（2）区分GATT（通用属性配置文件）服务端与 GATT 客户端：建立连接后，一台设备作为 GATT 服务端，另一台设备作为 GATT 客户端

（3）BLE 扫描和广播：根据指定状态获取外围设备、启动或停止BLE扫描、广播

# 传统蓝牙本机管理

- 传统蓝牙本机管理主要是针对蓝牙本机的基本操作

| 接口名 | 功能描述 |
| --- | --- |
| *getDefaultHost(Context context)* | 获取BluetoothHost实例，去管理本机蓝牙操作 |
| *enableBt()* | 打开本机蓝牙 |
| *disableBt()* | 关闭本机蓝牙 |
| *setLocalName(String name)* | 设置本机蓝牙名称 |
| *getLocalName()* | 获取本机蓝牙名称 |
| *getBtState()* | 获取本机蓝牙状态 |
| *startBtDiscovery()* | 发起蓝牙设备扫描 |
| *cancelBtDiscovery()* | 取消蓝牙设备扫描 |
| *isBtDiscovering()* | 检查蓝牙是否在扫描设备中 |
| *getProfileConnState(int profile)* | 获取本机蓝牙profile对其他设备的连接状态 |
| *getPairedDevices()* | 获取本机蓝牙已配对的蓝牙设备列 |

## 1. 打开蓝牙

（1）调用 ***getDefaultHost(Context context)*** 获取***BluetoothHost*** 实例，管理本机蓝牙

（2）调用 ***enableBt()*** 打开蓝牙

（3）调用 ***getBtState()***查询蓝牙是否打开

```
1.    // 获取蓝牙本机管理对象
2.    BluetoothHost bluetoothHost = BluetoothHost.getDefaultHost(context);
3.    // 调用打开接口
4.    bluetoothHost.enableBt();
5.    // 调用获取蓝牙开关状态接口
6.    int state = bluetoothHost.getBtState();
```

## 2. 蓝牙扫描

（1）注册广播 ***BluetoothRemoteDevice.EVENT_DEVICE_DISCOVERED***

（2）调用 ***startBtDiscovery()*** 扫描外围设备

（3）如需获取扫描到的设备，须实现广播方法***onReceiveEvent()***

```
1.  //开始扫描
2.  mBluetoothHost.startBtDiscovery();
3.  //接收系统广播
4.  public class MyCommonEventSubscriber extends CommonEventSubscriber {
5.      @Override
6.      public void onReceiveEvent(CommonEventData var){
7.          Intent info = var.getIntent();
8.          if(info == null) return;
9.          //获取系统广播的action
10.         String action = info.getAction();
11.         //判断是否为扫描到设备的广播
12.         if(action == BluetoothRemoteDevice.EVENT_DEVICE_DISCOVERED){
13.             IntentParams myParam = info.getParams();
14.             BluetoothRemoteDevice device =
(BluetoothRemoteDevice)myParam.getParam(BluetoothRemoteDevice.REMOTE_DEVICE_PARAM_DE
15.         }
16.     }
17. }
```

# 传统蓝牙远端设备操控

- 传统蓝牙远端管理操作主要是针对远端蓝牙设备的基本操作

| 接口名 | 功能描述 |
| --- | --- |
| *getDeviceAddr()* | 获取远端蓝牙设备地址 |
| *getDeviceClass()* | 获取远端蓝牙设备类型 |
| *getDeviceName()* | 获取远端蓝牙设备名称 |
| *getPairState()* | 获取远端设备配对状态 |
| *startPair()* | 向远端设备发起配对 |

# 传统蓝牙远端设备操作

(1) 调用*getDefaultHost(Context context)*获取*BluetoothHost* 实例，管理本机蓝牙

(2) 调用 *enableBt()* 打开蓝牙

(3) 调用 *startBtDiscovery()*扫描设备

(4) 调用 *startPair()*发起配对

```
1.    // 获取蓝牙本机管理对象
2.    BluetoothHost bluetoothHost = BluetoothHost.getDefaultHost(context);
3.    // 调用打开接口
4.    bluetoothHost.enableBt();
5.    // 调用扫描接口
6.    bluetoothHost.startBtDiscovery();
7.    //设置界面会显示出扫描结果列表，点击蓝牙设备去配对
8.    BluetoothRemoteDevice device = bluetoothHost.getRemoteDev(TEST_ADDRESS);
9.    device.startPair();
```

## 根据指定状态获取外围设备、启动或停止 BLE 扫描、广播

| 接口名 | 功能描述 |
| --- | --- |
| *startScan(List<BleScanFilter> filters)* | 进行 BLE 蓝牙扫描，并使用 filters 对结果进行过滤 |
| *stopScan()* | 停止 BLE 蓝牙扫描 |
| *getDevicesByStates(int[] states)* | 根据状态获取连接的外围设备 |
| *BleCentralManager(BleCentralManagerCallback callback)* | 获取中心设备管理对象 |
| *onScanCallback(BleScanResult result)* | 扫描到 BLE 设备的结果回调 |
| *onStartScanFailed(int resultCode)* | 启动扫描失败的回调 |
| *BleAdvertiser(Context context, BleAdvertiseCallback callback)* | 用于获取广播操作对象 |
| *startAdvertising(BleAdvertiseSettings settings, BleAdvertiseData advData, BleAdvertiseData scanResponse)* | 进行 BLE 广播，第一个参数为广播参数，第二个为广播数据，第三个参数是扫描和广播数据参数的响应 |
| *stopAdvertising()* | 停止 BLE 广播 |
| *startResultEvent(int result)* | 广播回调结果 |

# 中心设备BLE扫描

(1) 继承*BleCentralManagerCallback*实现*onScanCallback*和*onStartScanFailed*回调函数

(2) 调用*BleCentralManager()*获取中心设备管理对象

(3) 获取扫描过滤器，调用*startScan()*扫描BLE设备，在回调中获取扫描到的BLE设备

```
1.// 实现扫描回调
2.public class ScanCallback implements BleCentralManagerCallback{
3.@Override
5.    public void onScanCallback(BleScanResult var1) {
6.          results.add(var1);}
7.    @Override
8.    public void onStartScanFailed(int var1) {
9.          HiLog.info(TAG,"Start Scan failed,Code:" + var1);
10.    }
11. }
12. // 获取中心设备管理对象
13. private ScanCallback centralManagerCallback = new ScanCallback();
14. private BleCentralManager centralManager = new BleCentralManager(centralManagerCallback);
15. // 创建扫描过滤器然后开始扫描
16. List<BleScanFilter> filters = new ArrayList<BleScanFilter>();
17. centralManager.startScan(filters);
```

# 外围设备进行BLE广播

（1）继承*advertiseCallback*类实现*startResultEvent*回调，用于获取广播结果

（2）调用*BleAdvertiser()*获取广播对象，构造广播参数和广播数据

（3）调用*startAdvertising()*接口开始BLE广播

```
1. // 实现 BLE 广播回调
2. private BleAdvertiseCallback advertiseCallback = new
BleAdvertiseCallback()    {
3.   @Override
4.   public void startResultEvent(int result) {
5.       if(result == BleAdvertiseCallback.RESULT_SUCC){
6.           // 开始 BLE 广播成功
7.       }
8.      else {
9.          // 开始 BLE 广播失败
10.      }
11. }
12.  };
13.// 获取 BLE 广播对象
14. private BleAdvertiser advertiser = new
                    BleAdvertiser(this,advertiseCallback);
```

```
15. // 创建 BLE 广播参数和数据
16. private BleAdvertiseData data = new BleAdvertiseData.Builder()
17. .addServiceUuid(SequenceUuid.uuidFromString(Server_UUID))
18. .addServiceData(SequenceUuid.uuidFromString(Server_UUID),new
                byte[]{0x11}) // 添加广播数据内容
19.  .build();
20.  private BleAdvertiseSettings advertiseSettings = new
                        BleAdvertiseSettings.Builder()
21.  .setConnectable(true) // 设置是否可连接广播
22.  .setInterval(BleAdvertiseSettings.INTERVAL_SLOT_DEFAULT)
23.  .setTxPower(BleAdvertiseSettings.TX_POWER_DEFAULT)
24.  .build();
25.  // 开始广播
26.   advertiser.startAdvertising(advertiseSettings,data,null);
```

# 主要内容

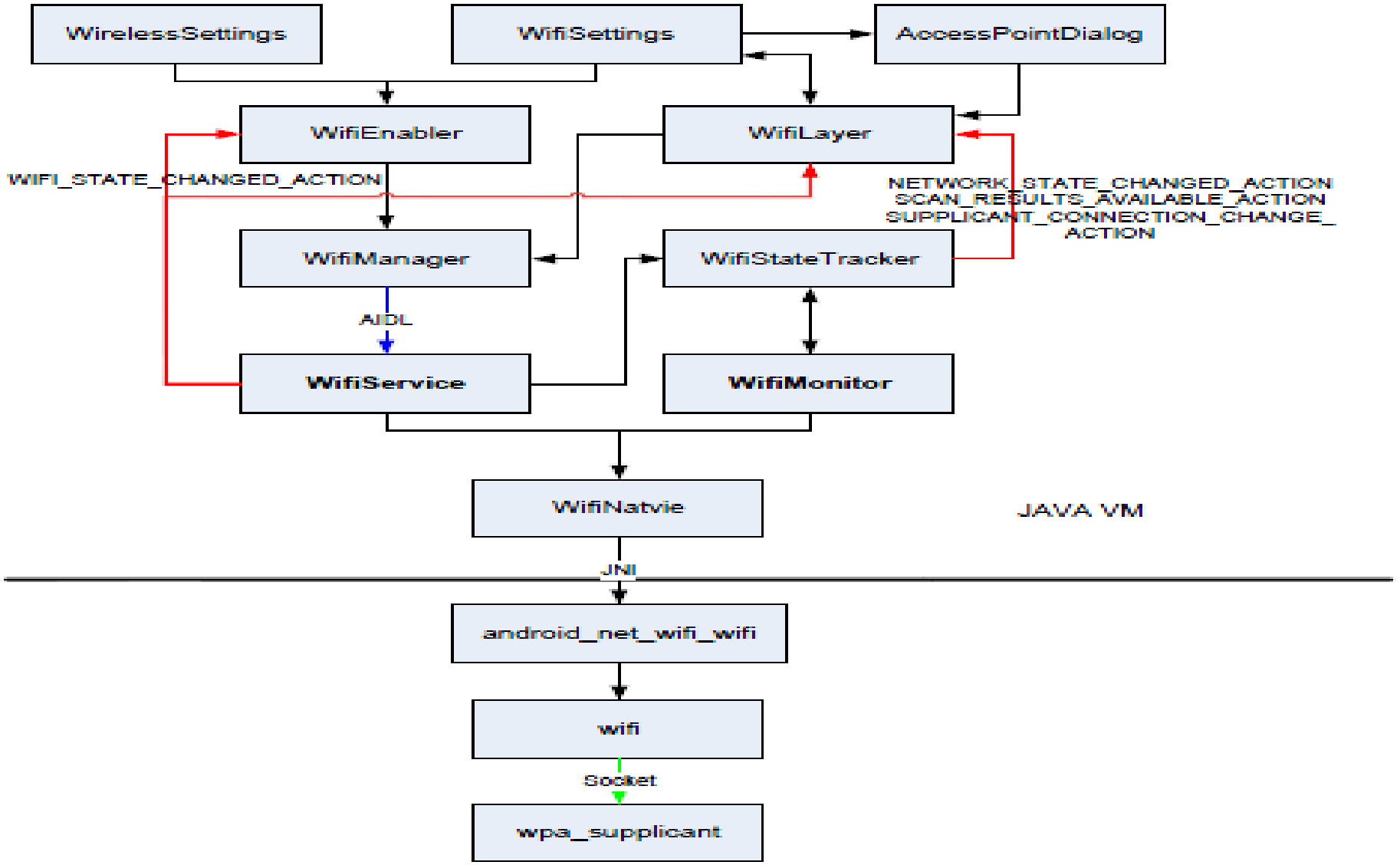
- ***WebView* 控件**
- 访问*HTTP*资源
- ***Socket*通信**
- 数据报套接字和*SSL*
- ***Harmony*平台的网络连接**
- 蓝牙通信
- ***Wifi* 通信**

# Android Wifi简介

- ***Wi-Fi*** 的英文全称为***wireless fidelity***，在无线局域网的范畴是指“无线相容性认证”，是一种无线联网的技术，通常通过无线路由器实现，在无线路由器电波覆盖的有效范围均可采用***WIFI***连接方式进行联网

- ***Wi-Fi （802.11G）***通常在2.4Ghz频段工作，所支持的速度最高达54Mbps

# Android Wifi系统架构



# Wifi网卡的状态

- ***WIFI***网卡状态表示：
  - ***0 --> WIFI_STATE_DISABLING***
  - ***1 --> WIFI_STATE_DISABLED***
  - ***2 --> WIFI_STATE_ENABLING***
  - ***3 --> WIFI_STATE_ENABLED***
  - ***4 --> WIFI_STATE_UNKNOWN***
- 其中0表示网卡正在关闭；1表示网卡不可用，2表示网卡正在打开，3表示网卡可用，4表示未知网卡状态

# Wifi权限设置

```
<uses-ermissionandroid:name="android.permission.CHANGE_NETWORK_STATE">
</uses-permission>  //修改网络状态的权限

<uses-permissionandroid:name="android.permission.CHANGE_WIFI_STATE">
</uses-permission>  //修改wifi状态的权限

<uses-ermissionandroid:name="android.permission.ACCESS_NETWORK_STATE">
</uses-permission>  //访问网络权限

<uses-permissionandroid:name="android.permission.ACCESS_WIFI_STATE">
</uses-permission>   //访问wifi权限
```

# Wifi网卡的操作过程

Android开发Wifi主要包括以下几个类和接口

- ***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、信号强度等信息
- ***WifiConfiguration***： ***WiFi***网络的配置，包括安全配置等
- ***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***：提供了管理WiFi连接的大部分API

# Wifi网卡的操作过程

- 对***wifi***网卡进行操作需要通过***WifiManger***对象来进行

*WifiManger wifiManger = (WifiManger)Context.getSystemService(Service.WIFI_SERVICE);*

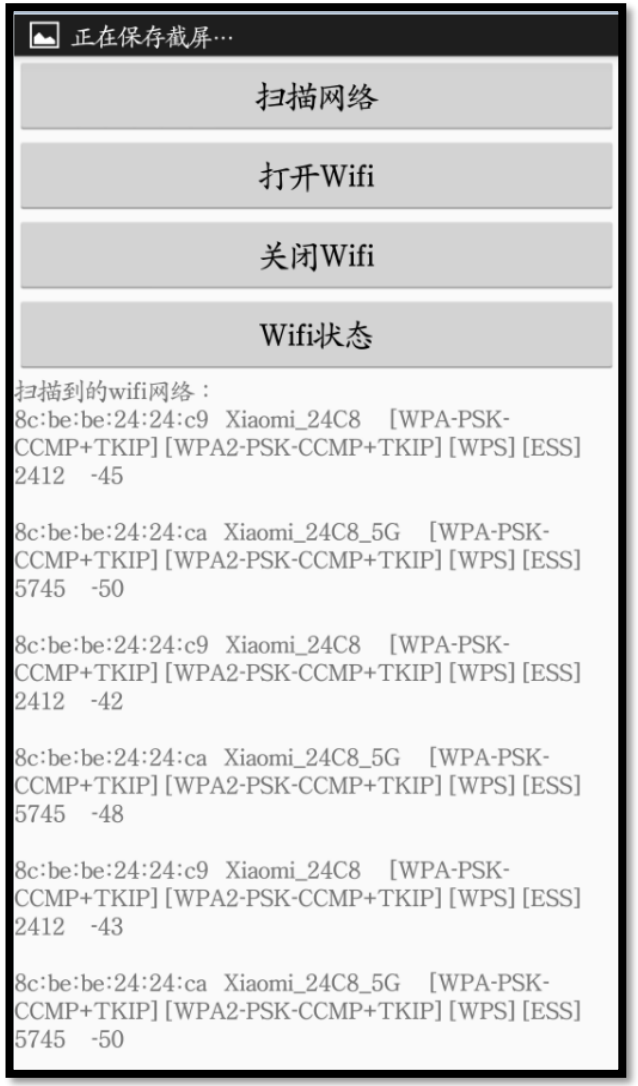
- 打开wifi网卡： *wifimanger.setWifiEnabled(true);*
- 关闭wifi网卡： *wifiManger.setWifiEnabled(false);*
- 获取网卡的当前的状态： *wifiManger.getWifiState();*
- 添加一个配置好的网络连接:*wifiManger.addNetwork();*
- 计算信号的强度:*wifiManger.calculateSignalLevel();*
- 比较两个信号的强度:*wifiManger.compareSignalLevel();*
- 创建一个WiFi锁:*wifiManger.createWifiLock();*
- 从接入点断开:*wifiManger.disconnect();*
- 扫描已存在的接入点:*wifiManger.startScan();*
- 更新已经配置好的网络:*wifiManger.updateNetwork();*

# Wifi工作步骤

- ***Wifi*** 启动
- 开始扫描
- 显示扫描的***AP***
- 配置***AP***
- 连接***AP***
- 获取IP地址
- 上网

# 管理手机wifi


9:49 PM
扫描网络
打开Wifi
关闭Wifi
Wifi状态
当前没有扫描到Wifi网络


正在保存截屏…
扫描网络
打开Wifi
关闭Wifi
Wifi状态
扫描到的wifi网络：
8c:be:be:24:24:c9 Xiaomi_24C8 [WPA-PSK-CCMP+TKIP] [WPA2-PSK-CCMP+TKIP] [WPS] [ESS] 2412 -45
8c:be:be:24:24:ca Xiaomi_24C8_5G [WPA-PSK-CCMP+TKIP] [WPA2-PSK-CCMP+TKIP] [WPS] [ESS] 5745 -50
8c:be:be:24:24:c9 Xiaomi_24C8 [WPA-PSK-CCMP+TKIP] [WPA2-PSK-CCMP+TKIP] [WPS] [ESS] 2412 -42
8c:be:be:24:24:ca Xiaomi_24C8_5G [WPA-PSK-CCMP+TKIP] [WPA2-PSK-CCMP+TKIP] [WPS] [ESS] 5745 -48
8c:be:be:24:24:c9 Xiaomi_24C8 [WPA-PSK-CCMP+TKIP] [WPA2-PSK-CCMP+TKIP] [WPS] [ESS] 2412 -43
8c:be:be:24:24:ca Xiaomi_24C8_5G [WPA-PSK-CCMP+TKIP] [WPA2-PSK-CCMP+TKIP] [WPS] [ESS] 5745 -50

# Harmony WLAN基础功能

- 获取 *WLAN* 状态，查询 *WLAN* 是否打开
- 获取连接态详细信息，包括连接信息、IP信息等
- 发起扫描并获取扫描结果
- 获取设备国家码；获取设备是否支持指定的能力

| 接口名 | 功能描述 | 所需权限 |
| --- | --- | --- |
| *getInstance(Context context)* | 获取WLAN功能管理对象实例 | NA |
| *isWifiActive()* | 获取当前WLAN打开状态 | ohos.permission.GET_WIFI_INFO |
| *scan()* | 发起 WLAN 扫描 | ohos.permission.SET_WIFI_INFOohos.permission.LOCATION |
| *getScanInfoList()* | 获取上次扫描结果 | ohos.permission.GET_WIFI_INFOohos.permission.LOCATION |
| *isConnected()* | 获取当前 WLAN 连接状态 | ohos.permission.GET_WIFI_INFO |
| *getLinkedInfo()* | 获取当前的 WLAN 连接信息 | ohos.permission.GET_WIFI_INFO |
| *getIpInfo()* | 获取当前连接的 WLAN IP 信息 | ohos.permission.GET_WIFI_INFO |
| *getSignalLevel(int rssi, int band)* | 通过 RSSI 与频段计算信号格数 | NA |
| *getCountryCode()* | 获取设备的国家码 | ohos.permission.LOCATIONohos.permission.GET_WIFI_INFO |
| *isFeatureSupported(long featureId)* | 获取设备是否支持指定的特性 | ohos.permission.GET_WIFI_INFO |

# Harmony WLAN P2P功能

1.发现对端设备　2.建立与移除群组　3.向对端设备发起连接　4.获取*P2P*相关信息

| 接口名 | 功能描述 | 所需权限 |
| --- | --- | --- |
| ***init(EventRunner eventRunner, WifiP2pCallback callback)*** | 初始化 P2P 的信使，当且仅当信使成功初始化，P2P 的其他功能才可以正常使用 | ohos.permission.GET_WIFI_INFOohos.permission.SET_WIFI_INFO |
| ***discoverDevices(WifiP2pCallback callback)*** | 搜索附近可用的 P2P 设备 | ohos.permission.GET_WIFI_INFO |
| ***stopDeviceDiscovery(WifiP2pCallback callback)*** | 停止搜索附近的 P2P 设备 | ohos.permission.GET_WIFI_INFO |
| ***createGroup(WifiP2pConfig wifiP2pConfig, WifiP2pCallback callback)*** | 建立 P2P 群组 | ohos.permission.GET_WIFI_INFO |
| ***removeGroup(WifiP2pCallback callback)*** | 移除 P2P 群组 | ohos.permission.GET_WIFI_INFO |
| ***requestP2pInfo(int requestType, WifiP2pCallback callback)*** | 请求 P2P 相关信息，如群组信息、连接信息、设备信息等 | ohos.permission.GET_WIFI_INFO |
| ***connect(WifiP2pConfig wifiP2pConfig, WifiP2pCallback callback)*** | 向指定设备发起连接 | ohos.permission.GET_WIFI_INFO |
| ***cancelConnect(WifiP2pCallback callback)*** | 取消向指定设备发起的连接 | ohos.permission.GET_WIFI_INFO |

# Harmony WLAN P2P消息通知

- ***WLAN*** 消息通知（***Notification***）是 ***HarmonyOS*** 内部或者与应用之间跨进程通讯的机制，注册者在注册消息通知后，一旦符合条件的消息被发出，注册者即可接收到该消息并获取消息中附带的信息

| 描述 | 通知名 | 附加参数 |
| --- | --- | --- |
| WLAN状态 | usual.event.wifi.POWER_STATE | active_state |
| WLAN扫描 | usual.event.wifi.SCAN_FINISHED | scan_state |
| WLAN RSSI变化 | usual.event.wifi.RSSI_VALUE | rssi_value |
| WLAN连接状态 | usual.event.wifi.CONN_STATE | conn_state |
| Hotspot状态 | usual.event.wifi.HOTSPOT_STATE | hotspot_active_state |
| Hotspot连接状态 | usual.event.wifi.WIFI_HS_STA_JOIN usual.event.wifi.WIFI_HS_STA_LEAVE | - |
| P2P状态 | usual.event.wifi.p2p.STATE_CHANGE | p2p_state |
| P2P连接状态 | usual.event.wifi.p2p.CONN_STATE_CHANGE | linked_infonet_infogroup_info |
| P2P设备列表变化 | usual.event.wifi.p2p.PEERS_STATE_CHANGE | peers_list |
| P2P搜索状态变化 | usual.event.wifi.p2p.PEER_DISCOVERY_STATE_CHANGE | peers_discovery |
| P2P当前设备变化 | usual.event.wifi.p2p.CURRENT_DEVICE_CHANGE | p2p_device |
| P2P群组信息变化 | usual.event.wifi.p2p.GROUP_STATE_CHANGED | - |

# The End