#14 网络访问

目录

1. 访问因特网（HTTP）

- HTTP

1. 超文本传输协议(HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议。

2. 最常见的从网络传输数据的方式就是使用HTTP

3. HTTP可以封装几乎所有类型的数据

1. 访问因特网（HTTP）

- Permission

要访问互联网， 首先需要设置好相应权限

文件AndroidManifest添加：

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

android.permission.ACCESS_NETWORK_STATE 主要是用于

访问ConnectivityManager (通常是管理网络访问连接状态)

1. 访问因特网（HTTP）

- 从Web读取数据

1. 创建一个新的URL对象
2. 为这个URL资源打开一个stream
3. 读取数据
4. 关闭InputStream

1. 访问因特网（HTTP）

- 从URL读取数据

1. 读取数据实例代码

```
URL text = new URL("http://sysu.github.io/");
InputStream inputStream = text.openStream();
byte[] bytes = new byte[250];
int readSize = inputStream.read(bytes);
Log.i("HTTP", "readSize = " + readSize);
Log.i("HTTP", "bText = " + new String(bytes));
inputStream.close();
```

2. LogCat观察输出结果

```
readSize = 250
bText = <!DOCTYPE html>
<head>
 <meta charset="utf-8" />
 <title>Homepage &#8212; OPENSYSU</title>
 <meta name="description" content="Open Source in SYSU">
 <link rel="alternate" type="application/atom+xml" href="/feed/index.xml" />
 <link rel="stylesheet" t
```

1. 访问因特网（HTTP）

- 从Web读取数据

1. 上述方法虽简单，但并不严谨
2. 没有很好的错误处理：如手机没有网络、服务器关闭、URL无效、用户操作超时
3. 因此，从一个URL读取数据值之前，往往需要了解更多的信息，例如，需要读取的数据到底有多大

1. 访问因特网（HTTP）

- 使用HttpURLConnection

1. HttpURLConnection可以对URL进行检查，避免错误地传输过多的数据
2. HttpURLConnection获取一些有关URL对象所引用的资源信息

如：HTTP状态、头信息、内容的长度、类型和日期时间等

1. 访问因特网（HTTP）

- 使用HttpURLConnection

1. 创建一个新的URL对象
2. 为这个URL资源打开Connection
3. 读取数据

1. 访问因特网（HTTP）

- 使用HttpURLConnection

1. HttpURLConnection 实例代码

```
URL text = new URL("http://sysu.github.io/");
HttpURLConnection httpURLConnection = (HttpURLConnection)
text.openConnection();
Log.i("HTTP", "respCode = " +
                httpURLConnection.getResponseCode());
Log.i("HTTP", "contentType = " +
                httpURLConnection.getContentType());
Log.i("HTTP", "content = " + httpURLConnection.getContent());
```

2. LogCat观察输出结果

```
I/HTTP: respCode = 200
I/HTTP: contentType = text/html; charset=utf-8
I/HTTP: content = buffer(com.android.okhttp.okio.GzipSource@744fe62).inputStream()
```

1. 访问因特网（HTTP）

- 解析从网络获取的数据

1. 大部分网络资源的传输存储在一种结构化的形式中， 通常会使用可拓展标记语言(Extensible Markup Language, XML) 或JSON（JavaScript Object Notation）
2. Android提供了一种快速而高效的XML Pull Parse, 是网络应用程序解析器的其中一个选择
3. Google公司发布Gson-Converter一个开放原始码的Java库, 方便了JSON与JAVA之间的转换

1. 访问因特网（HTTP）

- 解析从网络获取的数据

1. XML Example

```xml
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

2. JSON Example

```json
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

1. 访问因特网（HTTP）

- 解析从网络获取的XML

1. START_TAG:找到一个新的标记时（<tag>）返回
2. TEXT:当找到文本时返回（即<tag>TEXT</tag>）
3. END_TAG:找到标记的结束时（</tag>）返回
4. END_DOCUMENT:当到达XML文件末尾时返回

1. 访问因特网（HTTP）

- 解析从网络获取的XML

1. 创建URL实例

2. 从XmlPullParserFactory中获取一个XmlPullParser实例

```
URL text = new URL( “http://.....");
XmlPullParserFactory parserCreator=xmlPullParserFactory.newInstance();
XmlPullParser parser = parserCreator.newPullParser();
parser.setInput(text.openStream(), null);
status.setText("Parsing...");
```

1. 访问因特网（HTTP）

- 解析从网络获取的XML

若想获取 https://stackoverflow.com/feeds/ 中link Tag标签中的属性， 可以先创建好相应的XmlPullParser对象

```
XmlPullParserFactory factory = XmlPullParserFactory.newInstance();

factory.setNamespaceAware(true);

XmlPullParser parser = factory.newPullParser();
//获取XmlPullParser 实例

URL text = new URL("https://stackoverflow.com/feeds/");
//获取URL对象

parser.setInput(text.openStream(), null);
```

1. 访问因特网（HTTP）

- 解析从网络获取的XML

可以使用下方所示代码对上页例子解析

```
while(eventType != XmlPullParser.END_DOCUMENT) {
    String tagName = parser.getName();
    switch (eventType) {
        case XmlPullParser.START_TAG:
            if(tagName.compareTo("link") == 0) {
                System.out.println("rel attributeValue  : " + parser.getAttributeValue(null, "rel"));
                System.out.println("href attributeValue  :  " + parser.getAttributeValue(null, "href"));
            }
            break;
        case XmlPullParser.TEXT:
            break;

        case XmlPullParser.END_TAG:
            break;
        default:
            break;
    }
    eventType = parser.next();
}
```

解析结果：

```
rel attributeValue  : self
href attributeValue  :  https://stackoverflow.com/feeds/
rel attributeValue  : alternate
href attributeValue  :  https://stackoverflow.com/questions
rel attributeValue  : alternate
href attributeValue  :  https://stackoverflow.com/questions/47711286/gradient-for-l1-l2-in-sgd
```

2. Android线程机制

- 使用线程访问网络

1. 之前所提及到的网络操作方式会造成UI线程阻塞，直到网络操作完成为止
2. 如果在主线程中直接使用HTTP同步访问， AS会抛出 android.os.NetworkOnMainThreadException
3. 把一些耗时的操作从UI线程中移开，重新开启一个新的工作线程来执行这些任务,带给用户流畅的体验

2. Android线程机制

- Android线程模型

1. 单线程模型常常会引起Android应用程序性能低下。如果在单线程下执行一些耗时的操作，如访问网络或查询数据库，会阻塞整个用户界面。

2. 如果阻塞应用程序的时间过长(在Android系统中为5秒钟)，Android会向用户提示如下信息

2. Android线程机制

- Android线程模型

1. 因此需要避免在UI线程中执行耗时的操作

2. 请看以下代码：按钮的单击事件从网络上下载一副图片并使用ImageView来展现这幅图片。

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            Bitmap b = loadImageFromNetwork();
            mImageView.setImageBitmap(b);
        }).start();
}
```

2. Android线程机制

- Android线程模型

1. 上页代码好像很好地解决了遇到的问题，因为它不会阻塞UI线程。

然而运行时，Android会提示程序因为异常而终止。Why?

2. 原因是代码违背了Android单线程模型的原则：Android UI操作并

不是线程安全的，并且这些操作必须在UI线程中执行。

3. LogCat中打印的日志信息就会发现这样的错误日志：

android.view.ViewRoot$CalledFromWrongThreadException: Only the original

thread that created a view hierarchy can touch its views.

2. Android线程机制

- Android线程模型

1. 因此，Andriod提供了几种在其他线程中访问UI线程的方法。

    - Handler
    - AsyncTask
    - RxJava

2. Android线程机制

- Android线程模型

1. 使用上页方法中的任何一种纠正前面的代码示例， 例如： 把Runnable添加至消息队列，并由UI线程来处理

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap b = loadImageFromNetwork();
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(b);
                }
            });
        }
    }).start();
}
```

2. Android线程机制

- Android线程模型--- AsyncTask

1. AsyncTask Example

```
private class ImageLoader extends AsyncTask<URL, String, String> {
      @Override
      protected String doInBackground(URL... params) {
         try {
            URL text = params[0];
            //执行代码，如网络访问，结果解析等
            publishProgress(“Test”);//调用onProgressUpdate
          } catch (Exception e) {
            Log.e("Net", "Failed", e);
            return "Finished with failure.";
          }
          return "Done...";
      }
      (接下页)
```

2. Android线程机制

- Android线程模型--- AsyncTask

```
    protected void onCancelled() {
        Log.e("Net", "Async task Cancelled");
    }

    protected void onPostExecute(String result) {
        mStatus.setText(result);//result是doInBackground中的
                                //return的值,本例中为”Done...”
    }

    protected void onPreExecute() {
        mStatus.setText("About to load URL");
    }

    protected void onProgressUpdate(String... values) {
        mStatus.setText(values[0]);//values[0]为doInBackg中
                                   //publishProgress(String)中的String
        super.onProgressUpdate(values);
    }
}
```

2. Android线程机制

- 使用线程访问网络

1. 新建线程

```
new Thread() {
    public void run() {
        try {
            //执行网络连接代码以及解析代码
            mHandler.post(new Runnable() {
                public void run() {
                    //把有关用户界面更新的内容提交回主线程
                }
            });
        } catch (Exception e) {//异常处理
        }
    }
}.start();
```

2. Android线程机制

- 使用线程访问网络

1. 详细代码示例

```
new Thread() {
    public void run() {
        try {
            mHandler.post(new Runnable() { //执行网络连接
                public void run() {
                    status.setText( “Parsing...");
                }
            });
            mHandler.post(new Runnable() { //执行解析代码
                public void run() {
                    status.setText( “Done...");
                }
            });
        } catch (Exception e) { //异常处理
    }}}.start();
```

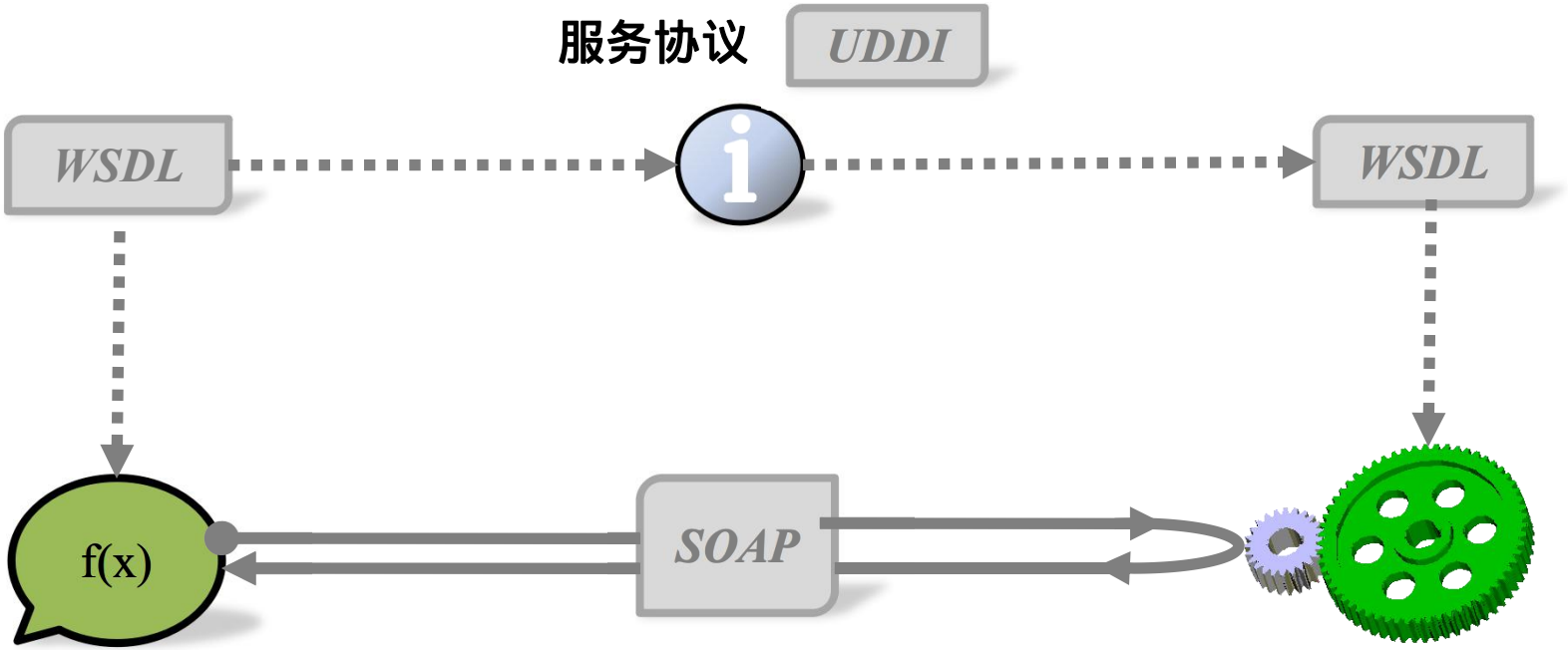
3. Web服务

- Web服务概述

1. Web服务是一种面向服务架构（SOA）的技术，通过标准的Web协议提供服务，目的是保证不同平台的应用服务可以互操作。
2. Web服务（Web service）应当是一个软件系统，用以支持网络间不同机器的互动操作。网络服务通常是许多应用程序接口（API）所组成的，它们透过网络，例如国际互联网（Internet）的远程服务器端，执行客户所提交服务的请求。

3. Web服务

- Web服务的架构



3. Web服务

- Web服务概述

1. Web服务主要用到以下几个核心技术和规范。
    - SOAP： 表示信息交换的协议.
    - WSDL: Web服务描述语言
    - UDDI: 一个用来发布和搜索Web服务的协议。

3. Web服务

- Web服务概述

1. SOAP(Simple Object Access Protocol)

   - SOAP技术把基于HTTP的Web技术与XML的可扩展性相结合，实现异构程序和平台之间的互操作性，使应用能够被不同的用户所访问。

2. WSDL(Web Services Description Language)

   - WSDL描述Web服务的公共接口。这是一个基于XML的关于如何与Web服务通讯和使用的服务描述；也就是描述与目录中列出的Web服务进行交互时需要绑定的协议和信息格式

3. Web服务

- Web服务概述

3. UDDI(Universal Description, Discovery, and Integration)

  - UDDI是一个基于XML的跨平台的描述规范，可以使世界范围内的企业在互联网上发布自己所提供的服务。

3. Web服务

- Web服务调用原理

1. 服务提供者首先建立Web服务，然后把服务发布给所有用户。
2. 任何平台上的用户可以通过阅读其WSDL文档生成一个SOAP请求消息。这个SOAP消息嵌入到一个HTTP POST请求中发送到Web服务所在的Web服务器。
3. Web服务器把请求转发给Web服务请求处理器，请求处理器解析SOAP请求，然后调用Web服务生成相应的SOAP应答。
4. Web服务器得到SOAP应答后通过HTTP送回客户端。

3. Web服务

- Web服务调用原理-高层接口

1. 使用高层接口，不需要知道SOAP和XML的任何信息，就可以生成和使用一个Web服务。

2. Soap Tookit 2.0 通过提供SoapClient和SoapServer两个COM对象来完成这些功能。

   在客户端，只要生成一个soapclient实例，并用WSDL作为参数来调用其中的mossoapinit方法。soapclient对象会自动解析WSDL文件，并在内部生成所有web service的方法和参数信息

3. Web服务

- Web服务调用原理-低层接口

注： 使用低层接口必须对SOAP和XML有所了解。这种接口可以对SOAP的处理过程进行控制，特别是要做特殊处理的时候。

1. 创建一个HttpConnector对象负责HTTP连接。
2. 创建SoapSerializer对象，用于生成SOAP消息。
3. SOAP消息作为Playload通过HttpConnector被发送到服务端。
4. 生成一个SoapReader对象，负责读取服务端返回的SOAP消息。

4. 使用Ksoap2访问WebService

- Ksoap2概述

1. 在Android SDK中并没有提供调用WebService的库，因此，需要使用第三方类库（KSOAP2）来调用WebService。
2. Ksoap2是一个SOAP Web service客户端包。主要用于资源受限制的Java环境如Applets或J2ME应用程序（CLDC/CDC/MIDP）。

4. 使用Ksoap2访问WebService

- Android Studio 添加Ksoap2

1. 在Project中的build.gradle中添加Ksoap2源

maven { url 'https://oss.sonatype.org/content/repositories/ksoap2-android-releases/' }

```
allprojects {
    repositories {
        maven { url 'https://oss.sonatype.org/content/repositories/ksoap2-android-releases/' }
        jcenter()
    }
}
```

2. 在app的build.gradle中添加Ksoap2依赖

compile 'com.google.code.ksoap2-android:ksoap2-android:3.6.1'

4. 使用Ksoap2访问WebService

- Ksoap2概述

1. 指定WebService的URL，命名空间和调用的方法名

```
String url = "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx";
String nameSpace = "http://WebXml.com.cn/";
String methodName = "getSupportCity";
```

2. 命名空间和调用方法可参考网站

http://www.webxml.com.cn/WebServices/WeatherWebService.asmx

http://www.webxml.com.cn/WebServices/WeatherWebService.asmx?WSDL

4. 使用Ksoap2访问WebService

- Ksoap2实例

1. 从WSDL查看WebService的Namespace。 进入

   http://www.webxml.com.cn/WebServices/WeatherWebService.asmx?WSDL

```
<?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:tns="http://WebXml.com.cn/"
  xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" targetNamespace="http://WebXml.com.cn/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"><a href="http://www.webxml.com.cn/"
```

NameSpace

4. 使用Ksoap2访问WebService

- Ksoap2实例
  1. http://www.webxml.com.cn/WebServices/WeatherWebService.asmx 查看WebService提供的方法

4. 使用Ksoap2访问WebService

- Ksoap2实例

1. 设置调用方法的参数值，这一步是可选的，如果方法没有参数，可以省略这一步。设置方法的参数值的代码如下

```
SoapObject request = new SoapObject(nameSpace, methodName);
request.addProperty("param1", "value1");
request.addProperty("param2", "value2");
```

2. addProperty方法的第1个参数虽然表示调用方法的参数名，但该参数值并不一定与服务端的WebService类中的方法参数名一致，只要设置参数的顺序一致即可。

4. 使用Ksoap2访问WebService

- Ksoap2实例
  1. 进入

     http://www.webxml.com.cn/WebServices/WeatherWebService.asmx?op=getWeatherbyCityName

  2. 查看 getWeatherbyCityName

WebService读入参数格式定义

WebService输出参数格式定义

SOAP 1.1

以下是 SOAP 1.2 请求和响应示例。所显示的**占位符**需替换为实际值。

```
POST /WebServices/WeatherWebService.asmx HTTP/1.1
Host: webservice.webxml.com.cn
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://WebXml.com.cn/getWeatherbyCityName"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:x
  <soap:Body>
    <getWeatherbyCityName xmlns="http://WebXml.com.cn/">
      <theCityName>string</theCityName>
    </getWeatherbyCityName>
  </soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:x
  <soap:Body>
    <getWeatherbyCityNameResponse xmlns="http://WebXml.com.cn/">
      <getWeatherbyCityNameResult>
        <string>string</string>
        <string>string</string>
      </getWeatherbyCityNameResult>
    </getWeatherbyCityNameResponse>
  </soap:Body>
</soap:Envelope>
```

4. 使用Ksoap2访问WebService

- Ksoap2实例

```
POST /WebServices/WeatherWebService.asmx HTTP/1.1
Host: webservice.webxml.com.cn
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://WebXml.com.cn/getWeatherbyCityName"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w
  <soap:Body>
    <getWeatherbyCityName xmlns="http://WebXml.com.cn/">
      <theCityName>string</theCityName>
    </getWeatherbyCityName>
  </soap:Body>
</soap:Envelope>
```

把读入的String
赋给变量theCityName

```
SoapObject request = new SoapObject(nameSpace, methodName);
request.addProperty("theCityName", "广州");
```

4. 使用Ksoap2访问WebService

- Ksoap2实例

1. 生成调用WebService方法的SOAP请求信息。该信息由SoapSerializationEnvelope对象描述，代码如下：

```
SoapSerializationEnvelope envelope =
new SoapSerializationEnvelope(SoapEnvelope.VER11);
envelope.bodyOut = request;
envelope.dotNet = true; /*访问.NET的WebService必须加上这行*/
```

2. 创建SoapSerializationEnvelope对象时需要通过SoapSerializationEnvelope类的构造方法设置SOAP协议的版本号。该版本号需要根据服务端WebService的版本号设置。在创建SoapSerializationEnvelope对象后，还需要设置SoapSerializationEnvelope类的bodyOut属性.

4. 使用Ksoap2访问WebService

- Ksoap2实例

1. 创建HttpTransportSE对象。通过HttpTransportSE类的构造方法可以指定WebService的WSDL文档的URL，代码如下：

```
HttpTransportSE ht = new HttpTransportSE (url );
```

2. 使用call方法调用WebService方法，代码如下：

```
ht.call(ServiceNamespace + MethodName, envelope);
```

call方法的第1个参数是完整的方法名,前面加上命名空间，第2个参数就是在之前创建的SoapSerializationEnvelope对象。

4. 使用Ksoap2访问WebService

- Ksoap2实例

1. 获得WebService方法的返回结果，代码如下：

```
SoapObject result = (SoapObject) envelope.bodyIn;
SoapObject detail = (SoapObject)
result.getProperty("getWeatherbyCityNameResult");
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="htt
  <soap:Body>
    <getWeatherbyCityNameResponse xmlns="http://WebXml.com.cn/">
      <getWeatherbyCityNameResult>
        <string>string</string>
        <string>string</string>
      </getWeatherbyCityNameResult>
    </getWeatherbyCityNameResponse>
  </soap:Body>
</soap:Envelope>
```

结果由两个string组成，返回getWeatherCityNameResult

4. 使用Ksoap2访问WebService

- Ksoap2实例

1. 返回值是对象时:利用第3步创建的SOAP序列化封装对象获得的Web服务的返回结果，并强制类型转换为SoapObject类。

```
SoapObject SO = (SoapObject) en.getResponse();
```

2. 若读取的类型是int或单个string时，则是SoapPrimitive，不是SoapObject

```
SoapObject result = (SoapObject) envelope.bodyIn;
SoapPrimitive detail = (SoapPrimitive)
result.getProperty("addResult");
```

4. 使用Ksoap2访问WebService

- 小结

1. KSOAP调用Web服务需要运用HttpTransport类，实际上是调用了HttpConnection作为网络连接。

2. 在KSOAP调用Web服务的时候，如果由于某种原因，Web服务不能立即返回，Android界面上的组件仍然需要处于活动状态供用户使用，不能造成阻塞。

3. 为了防止UI组件的阻塞，KSOAP调用Web服务的时候，必须另起一个线程。

5. REST

- SOAP与REST对比

SOAP作为一种古老的Web服务技术，短期内还不会退出历史舞台。但因仅支持XML, 和较为复杂的解析操作是其缺点。

与SOAP相比， REST 使用了标准 HTTP ，因此其创建客户端，开发 API，编写文档都会更加简单

5. REST

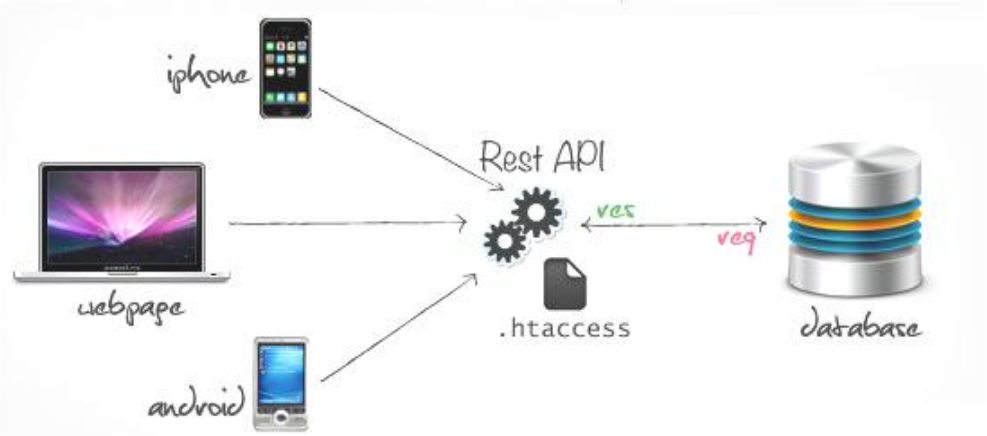
- REST(Representational State Transfer)
  1. 是Roy Thomas Fielding博士于2000年在他的博士论文中提出来的一种万维网软件架构风格，目的是便于不同软件/程序在网络（例如互联网）中互相传递信息。
  2. 资源是由URI来指定。对资源的操作包括获取、创建、修改和删除资源，这些操作正好对应HTTP协议提供的GET、POST、PUT和DELETE方法。
  3. 用 HTTP Status Code传递Server的状态信息。比如最常用的 200 表示成功，500 表示Server内部错误等。

5. REST

- REST(Representational State Transfer)

用HTTP协议里的动词来实现资源的添加，修改，删除等操作。即通过HTTP动词来实现资源的状态扭转：

1. GET　用来获取资源
2. POST　用来新建资源
3. PUT　用来更新资源
4. DELETE　用来删除资源。



5. REST

- REST(Representational State Transfer)

REST全称是表述性状态转移，那究竟指的是什么的表述？其实指的就是资源。**任何事物，只要有被引用到的必要，它就是一个资源。资源可以是实体(例如手机号码)，也可以只是一个抽象概念(例如价值) **。下面是一些资源的例子:

- 某用户的手机号码
- 某用户可以办理的优惠套餐
- 某手机号码的潜在价值

5. REST

- REST(Representational State Transfer)

要让一个资源可以被识别，需要有个唯一标识，在Web中这个唯一标识就是URI(Uniform Resource Identifier)。 URI既可以看成是资源的地址，也可以看成是资源的名称。如果某些信息没有使用URI来表示，那它就不能算是一个资源， 只能算是资源的一些信息而已。URI的设计应该遵循可寻址性原则，具有自描述性，需要在形式上给人以直觉上的关联。

5. REST

- REST(Representational State Transfer)

这里以github网站为例，给出一些还算不错的URI：

GET /zoos：列出所有动物园

POST /zoos：新建一个动物园

GET /zoos/ID：获取某个指定动物园的信息

PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）

PATCH /zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）

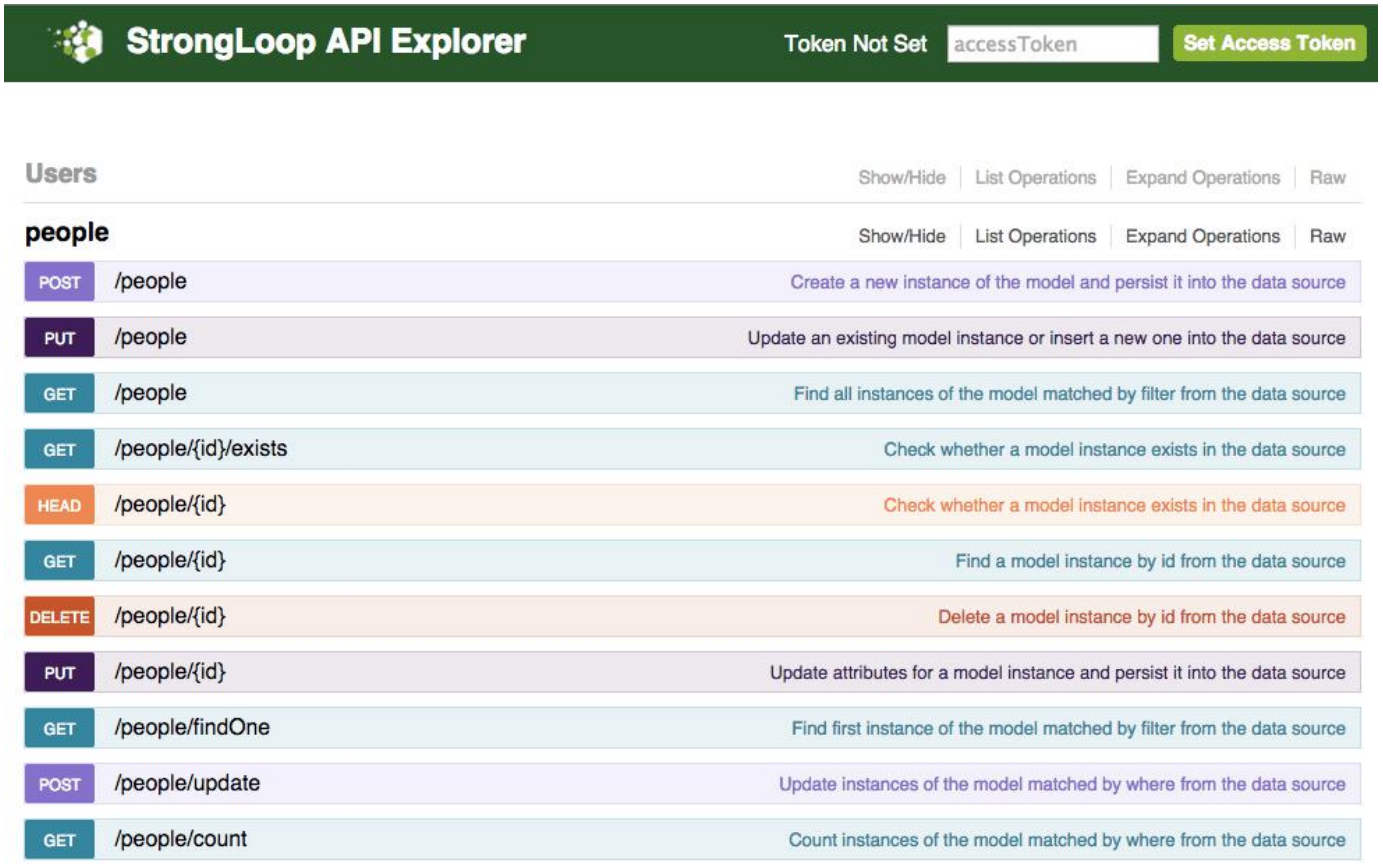
DELETE /zoos/ID：删除某个动物园

GET /zoos/ID/animals：列出某个指定动物园的所有动物

DELETE /zoos/ID/animals/ID：删除某个指定动物园的指定动物

5. REST

- REST(Representational State Transfer)



5. REST

- REST-过滤信息（Filtering/QueryParams）

  如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。下面是一些常见的参数。

  - ?limit=10：指定返回记录的数量
  - ?offset=10：指定返回记录的开始位置。
  - ?page=2&per_page=100：指定第几页，以及每页的记录数。
  - ?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序。
  - ?animal_type_id=1：指定筛选条件

6. Retrofit2访问WebService

- Retrofit简介

1. Retrofit是 Square公司方便APP访问服务器API所开发的库， 基于REST规范。
2. Retrofit 是对 OkHttp 的封装，提供了使用注解更简单的构建各种请求，配置各种参数的方式。本质发起网络请求的还是OkHttp，但 Retrofit 让这一操作更加的简单优雅。

6. Retrofit2访问WebService

- Retrofit依赖添加

使用Retrofit需要添加上相应的依赖

```
compile 'com.squareup.retrofit2:retrofit:2.1.0'//retrofit
//下面两个是RxJava 和RxAndroid
compile 'io.reactivex:rxjava:1.1.0'
compile 'io.reactivex:rxandroid:1.1.0'
compile 'com.squareup.retrofit2:converter-gson:2.1.0'//转换器，请求结果转换成Model
compile 'com.squareup.retrofit2:adapter-rxjava:2.1.0'//配合Rxjava 使用
```

6. Retrofit2访问WebService

- Retrofit注解(annotation)与请求方法

Retrofit通过给访问接口方法添加相应的注解来表示该方法对应于HTTP的哪种请求

请求方法	请求方法
@GET	表明这是get请求
@POST	表明这是post请求
@PUT	表明这是put请求
@DELETE	表明这是delete请求
@PATCH	表明这是一个patch请求，该请求是对put请求的补充，用于更新局部资源
@HEAD	表明这是一个head请求
@OPTIONS	表明这是一个option请求

6. Retrofit2访问WebService

- Retrofit使用

Retrofit使用分成简单的五个部分

1. 定义Model类

2. 定义访问API接口

3. 创建Retrofit对象

4. 使用创建好的Retrofit创建访问实例

5. 通过访问实例发送请求， 获取Response数据， GsonConverter会自动将JSON数据转换为定义好的Model

6. Retrofit2访问WebService

- Retrofit使用

定义Model类

通过访问API获取所需的JSON数据后， 需要将这些数据转换为我们所需要的Model。 可以访问URL查看返回的数据后， 根据项目需要， 定义相应的Model, 然后在Retrofit调用build()前

调用addConverterFactoryGsonConverterFactory.create()); 添加GSON Converter。

添加后， 在请求成功后得到的JSON数据会自动转换为定义好的model对象

6. Retrofit2访问WebService

- Retrofit使用

定义Model类

如： 我们需要提取右图中的login,

id, blog等数据可以定义如下Model

```
Class User {
    int id;
    String login;
    String blog;
}
```


```
{
 "login": "hellokenlee",
 "id": 8079836,
 "avatar_url": "https://avatars1.githubusercontent.com/u/8079836?v=4",
 "gravatar_id": "",
 "url": "https://api.github.com/users/hellokenlee",
 "html_url": "https://github.com/hellokenlee",
 "followers_url": "https://api.github.com/users/hellokenlee/followers",
 "following_url": "https://api.github.com/users/hellokenlee/following{/other_user}",
 "gists_url": "https://api.github.com/users/hellokenlee/gists{/gist_id}",
 "starred_url": "https://api.github.com/users/hellokenlee/starred{/owner}{/repo}",
 "subscriptions_url": "https://api.github.com/users/hellokenlee/subscriptions",
 "organizations_url": "https://api.github.com/users/hellokenlee/orgs",
 "repos_url": "https://api.github.com/users/hellokenlee/repos",
 "events_url": "https://api.github.com/users/hellokenlee/events{/privacy}",
 "received_events_url": "https://api.github.com/users/hellokenlee/received_events",
 "type": "User",
 "site_admin": false,
 "name": "KenLee",
 "company": null,
 "blog": "https://hellokenlee.github.io/",
 "location": "Guangzhou, China",
 "email": null,
 "hireable": null,
 "bio": "A CS student studying in Sun Yat-Sen University.",
 "public_repos": 24,
 "public_gists": 0,
 "followers": 12,
 "following": 7,
 "created_at": "2014-07-06T09:00:05Z",
 "updated_at": "2017-11-20T11:55:05Z"
}
```


因为添加了GSON Converter, 在后续调用API方法获取数据后， Converter会将该JSON数据转换为定义的model 对象

6. Retrofit2访问WebService

- Retrofit使用

定义相应HTTP API访问接口

```
public interface GitHubService {
    @GET("users/{user}")
    Call<User> getUser(@Path("user") String user);
}
```

将URL中{user}替换成传入的user参数

6. Retrofit2访问WebService

- Retrofit使用

构建Retrofit对象

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .addConverterFactoryGsonConverterFactory.create());
    .build();
```

设置访问服务端的baseUrl

创建API访问接口， 并调用接口函数获取相应的数据

```
GitHubService service = retrofit.create(GitHubService.class);
Call<User> userCall= service.getUser("octocat");
```

6. Retrofit2访问WebService

- Retrofit使用

当请求成功后， 会调用onResponse回调函数。通过response.body()可以获取返回的User 数据， 如：

```
userCall.enqueue(new Callback<User>() {
    @Override
     public void onResponse(Call<User> call, Response<User> response) {
        User receiver = response.body(); //Response body中的JSON数据已经转换为了User类型
     }
     @Override
        public void onFailure(Call<User> call, Throwable t) {

        }
});
```

6. Retrofit2访问WebService

- Retrofit使用

在访问接口的定义中会涉及到下列几个参数

1. PATH - 用于URL的参数替换

2. Body - 用于Post, Put请求的数据携带

3. Query - urlQuery的参数携带如 api/v1/user/list?limit=100&offset=10

表示limit=100, offset=10

6. Retrofit2访问WebService

- Retrofit-PATH

请求路径中可以包含参数，并在参数中使用 @PATH 注解来动态改变路径， 如下例子所示：

```
@PUT( "api/v2/baby/doEdit/{PATH_BABY_ID}")
Call<BaseResponse> putEditBaby(@Path(PATH_BABY_ID) Long babyId);
```

使用注解 @Path(PATH_BABY_ID) Long babyId 即可改变路径中 {PATH_BABY_ID} 请求时的值

6. Retrofit2访问WebService

- Retrofit-Body

发送 POST 、PUT 请求时通常需要携带数据进HTTP body内以返回给请求的客户端，使用 @Body 注解添加数据进HTTP Request中，如下：向服务端POST一个Model为Data的数据

```
@PUT(PUT_EDIT_BABY)
Call<BaseResponse> putEditBaby(@Body Data data);
```

```
{
  name:'test',
  age:12,
  links:[
   {
      rel:'parent',
      href:'/people/john'
   }
]
}
```

使用注解 (@Body Data data)携带body数据

6. Retrofit2访问WebService

- Retrofit-Query

在HTTP GET请求常需携带相应的query参数

比如： api/v1/user/list?limit=100&offset=10

第一种方法使用 @Query 注解，如 @Query("userId") Long userId 的形式。这种形式可以传递 null 值，如果某个参数为 null，将不会拼接在 url 后面。

```
Call<UserBabyRelationResp> getBabyRelationList
 (@Query("limit") int limit, @Query("offset") Long offset);
```

6. Retrofit2访问WebService

- Retrofit-Query

在HTTP GET请求常需携带相应的query参数

比如： api/v1/user/list?limit=100&offset=10

第二种使用 @QueryMap 注解，如 @QureyMap

Map<String,String> params 的形式。这种形式传递一个 map 作为参数，但是 map 中 value 不能为 null，否则会抛出异常。

```
Call<UserBabyRelationResp> getBabyRelationList
 (@QueryMap Map<String,String> map);
```

6. Retrofit2访问WebService

- Retrofit-Call

1. Retrofit2 有了新的类型:Call, 语法与okHttp基本一模一样。即：

```
Call<User> repos = service.getUser("octocat");
```

2. 每一个 call 对象实例只能被用一次，所以说 request 和 response 都是一一对应的。你其实可以通过 Clone 方法来创建一个一模一样的实例，这个开销是很小的。

6. Retrofit2访问WebService

- Retrofit-Call

同步与异步

同步与异步概念在多线程章节中已做相应介绍

同步:提交请求->等待处理(这个期间无法进行其他操作) -> 处理完毕返回

异步：请求通过事件触发->等待处理(期间仍然可以进行其他操作)->处理完毕回调

6. Retrofit2访问WebService

- Retrofit-Call

Call 同步调用-获取城市Id为101010100的天气情况

```
ApiService apiService = retrofit.create(ApiService.class);
Call<ResponseBody> callObject=apiService.getWeather("101010100");
try {
        System.out.println(callObject.execute().body().string());
        System.out.println("这是同步调用后的打印");
}catch (IOException e) {
        e.printStackTrace();
}
System.out.println("同步调用后")
```

```
I/System.out: 这是同步调用后的打印
I/System.out: 同步调用后
```

6. Retrofit2访问WebService

- Retrofit-Call

Call 异步调用-获取城市Id为101010100的天气情况

```
callObject.enqueue(new Callback<ResponseBody>() {
        @Override
        public void onResponse(Call<ResponseBody> call, Response<ResponseBody>
response) {
        try {
            System.out.println("这是异步调用的结果");
          }catch (IOException e) {
             e.printStackTrace();
          }
        }
        @Override
        public void onFailure(Call<ResponseBody> call, Throwable t) {
        }
});
System.out.println("异步调用后")
```

I/System.out: 异步调用后
I/System.out: 这是异步调用后的打印

6. Retrofit2访问WebService

- Retrofit开发参考网站

1. Retrofit + RxJava + OkHttp 让网络请求变的简单-基础篇

   http://www.jianshu.com/p/5bc866b9cbb9

2. Retrofit + RxJava + OkHttp 让网络请求变的简单-封装篇

   http://www.jianshu.com/p/811ba49d0748

3. RxJava 与 Retrofit 结合的最佳实践

   http://gank.io/post/56e80c2c677659311bed9841

7. 搭建Java版WebService

- WebService

做Android开发，不可避免会涉及到客户端开发，我们怎么样来实现一个服务端，怎么样来实现一个客户端，并相互传递数据。就算调用别人的服务时，也能知道大概是怎么样实现的。

WebService一般分为.Net版和Java版，今天主要来实现Java版的WebService，.Net版本的还是比较简单的。

7. 搭建Java版WebService

- WebService-Java版

有下列几种方式配置WebService

1. JAX-WS： Jax-WS是Java1.6中才有的,新的WebService模式,基于注解的方式配置WebService,很类似Asp中的WebService,难度已经比Xfire方式的配置降低了很多.
2. REST(JAX-RS) : 是一个Java编程语言的应用程序接口,支持按照表象化状态转变 (REST)架构风格创建Web服务
3. Xfire（已过时）

8. Android WiFi开发

- Wi-Fi

  WiFi是一种段程无线传输技术，能够在数百英尺范围内支持互联网接入的无线电信号。随着技术的发展，以及IEEE802.11a和IEEE802.11g等标准的出现，现在IEEE802.11这个标准已被统称作Wi-Fi。从应用层面来说，要使用Wi-Fi，用户首先要有Wi-Fi兼容的用户端装置。

8. Android WiFi开发

- 操作Wi-Fi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

8. Android WiFi开发

- WifiManager

要在应用程序中对Android系统的WiFi设备进行相关操作，需要在项目中的AndroidManifest.xml中选择性地添加如下几句用于声明权限的语句：

```
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE">
</uses-permission>
<uses-permission
android:name="android.permission.ACCESS_CHECKIN_PROPERTIES">
</uses-permission>
<uses-permission
android:name="android.permission.WAKE_LOCK"></uses-permission>
<uses-permission
android:name="android.permission.CHANGE_WIFI_STATE">
</uses-permission>
```

8. Android WiFi开发

- android.net.wifi

ScanResult	用于描述一个已经被检测到的wifi接入点。
WifiConfiguration	该类代表了一个已经配置好的wifi网络，包括了该网络的一些安全设置。例如接入点密码，接入点通讯所采用的安全标准。
WifiConfiguration.AuthAlgorithm	公认的IEEE 802.11标准认证算法。

8. Android WiFi开发

- Android WiFi相关类介绍

WifiConfiguration.GroupCipher	公认的组密码。
WifiConfiguration.KeyMgmt	公认的密钥管理方案。
WifiConfiguration.PairwiseCipher	公认的用于WPA的成对密码标准。
WifiConfiguration.Protocol	公认的安全协议
WifiConfiguration.Status	网络所可能存在的状态。

8. Android WiFi开发

- Android WiFi相关类介绍

WifiInfo	描述了各个wifi连接的状态，该连接是否处于活动状态或者是否处于识别过程中。
WifiManager	这个类比较重要。它提供了用于管理wifi连接的各种主要API。详见表后说明。
WifiManager.MulticastLock	允许应用程序接收wifi的多播数据包。
WifiManager.WifiLock	允许应用程序永久地保持wifi连接（防止系统自动回收）。

8. Android WiFi开发

- WifiManager

1. Android 操作WiFi的重要类——WifiManager，这个类提供了最主要的用于管理wifi连接的API。通过调用Context.getSystemService(Context.WIFI_SERVICE)方法来得到系统提供的WifiManager

```
WifiManager   mWifiManager = (WifiManager)
context.getSystemService(Context.WIFI_SERVICE);
```

8. Android WiFi开发

- WifiManager

1. 已经配置好的网络连接列表。这个列表可以被用户查看或者更新，而且可以通过它来修改个别接入点的属性；
2. 如果当前有连接存在的话，可以得到当前正处于活动状态的wifi连接的控制权，可以通过它建立或者断开连接，并且可以查询该网络连接的动态信息；
3. 通过对已经扫描到的接入点的足够信息来进行判断，得出一个最好的接入点进行连接。
4. 定义了很多用于系统广播通知的常量，它们分别代表了WiFi状态的改变。

8. Android WiFi开发

- WifiConfiguration相关子类简介
  1. WifiConfiguration.AuthAlgorthm 用来判断加密方法。
  2. WifiConfiguration.GroupCipher 获取使用GroupCipher 的方法来进行加密。
  3. WifiConfiguration.KeyMgmt 获取使用KeyMgmt 进行。
  4. WifiConfiguration.PairwiseCipher 获取使用WPA 方式的加密。
  5. WifiConfiguration.Protocol 获取使用哪一种协议进行加密。
  6. wifiConfiguration.Status 获取当前网络的状态。

8. Android WiFi开发

- WifiInfo相关方法简介

  在连接上WiFi后可以通过这个类获得一些已经连通的WiFi 连接的信息

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

8. Android WiFi开发

- WiFi相关编程

打开无线网卡

```
public void openNetCard() {
    if (!mWifiManager.isWifiEnabled()) {
        mWifiManager.setWifiEnabled(true);
    }
}
```

关闭无线网卡

```
public void closeNetCard() {
    if (mWifiManager.isWifiEnabled()) {
        mWifiManager.setWifiEnabled(false);
    }
}
```

8. Android WiFi开发

- WiFi相关编程

检查网卡状态

```
public void checkNetCardState() {
    if (mWifiManager.getWifiState() == 0) {
        Log.i(TAG, "网卡正在关闭");
    } else if (mWifiManager.getWifiState() == 1) {
        Log.i(TAG, "网卡已经关闭");
    } else if (mWifiManager.getWifiState() == 2) {
        Log.i(TAG, "网卡正在打开");
    } else if (mWifiManager.getWifiState() == 3) {
        Log.i(TAG, "网卡已经打开");
    } else {
        Log.i(TAG, "---_---晕.....没有获取到状态---_---");
    }
}
```

扫描网络

```
public void scan() {
    mWifiManager.startScan();
    listResult = mWifiManager.getScanResults();
    if (listResult != null) {
        Log.i(TAG, "当前区域存在无线网络，请查看扫描结果");
    } else {
        Log.i(TAG, "当前区域没有无线网络");
    }
}
```

8. Android WiFi开发

- WiFi相关编程

扫描结果

```
public String getScanResult() {
    // 每次点击扫描之前清空上一次的扫描结果
    if (mStringBuffer != null) {
        mStringBuffer = new StringBuffer();
    }
    // 开始扫描网络
    scan();
    listResult = mWifiManager.getScanResults();
    if (listResult != null) {
        for (int i = 0; i < listResult.size(); i++) {
            mScanResult = listResult.get(i);
            mStringBuffer = mStringBuffer.append("NO.").append(i + 1)
                    .append(" :").append(mScanResult.SSID).append("->")
                    .append(mScanResult.BSSID).append("->")
                    .append(mScanResult.capabilities).append("->")
                    .append(mScanResult.frequency).append("->")
                    .append(mScanResult.level).append("->")
                    .append(mScanResult.describeContents()).append("\n\n");
        }
    }
    Log.i(TAG, mStringBuffer.toString());
    return mStringBuffer.toString();
}
```

8. Android WiFi开发

- WiFi相关编程

连接WiFi

```
public void connect() {
    mWifiInfo = mWifiManager.getConnectionInfo();

}
```

WiFi连接状态

```
public void checkNetWorkState() {
    if (mWifiInfo != null) {
        Log.i(TAG, "网络正常工作");
    } else {
        Log.i(TAG, "网络已断开");
    }
}
```

断开WiFi

```
public void disconnectWifi() {
    int netId = getNetworkId();
    mWifiManager.disableNetwork(netId);
    mWifiManager.disconnect();
    mWifiInfo = null;
}
```

8. Android WiFi开发

- 实例代码

```
WifiManager   wifiManager = (WifiManager)
                                    context.getSystemService(Context.WIFI_SERVICE);
WifiInfo wifiInfo = wifiManager.getConnectionInfo();
System.out.println("WiFi Status : " + WifiConfiguration.Status.CURRENT);
System.out.println("BSSID : " + wifiInfo.getBSSID());
System.out.println("is hidden SSID :  "  + wifiInfo.getHiddenSSID());
System.out.println("IP Address : " + wifiInfo.getIpAddress());
System.out.println("Link Speed " + wifiInfo.getLinkSpeed());
System.out.println("MAC Address : " + wifiInfo.getMacAddress());
System.out.println("RSSI : " + wifiInfo.getRssi());
System.out.println("SSID : " + wifiInfo.getSSID());
```

8. Android WiFi开发

- 实例代码 - 打印结果

I/System.out: WiFi Status : 0

I/System.out: BSSID : 01:80:c2:00:00:03

I/System.out: is hidden SSID： false

I/System.out: IP Address : 251854858(25.185.48.58)

I/System.out: Link Speed 0

I/System.out: MAC Address : 02:00:00:00:00:00

I/System.out: RSSI : -55

I/System.out: SSID : "WiredSSID"

Questions?