毕业设计（论文）外文资料翻译

外文出处 Mark Murphy.Beginning

Android 2　Chapter 33

Mapping with MapView and MapActivity

One of Google's most popular services-after search of course-is Google Maps, which lets you find everything from the nearest pizza parlor to directions from New York City to San Francisco (only 2,905 miles!), along with supplying street views and satellite imagery.

Most Android devices, not surprisingly, integrate Google Maps. For those that do, there is a mapping activity available to users directly from the main Android launcher. More relevant to you, as a developer, are MapView and MapActivity, which allow you to integrate maps into your own applications. Not only can you display maps, control the zoom level, and allow people to pan around, but you can tie in Android's location-based services (covered in Chapter 32) to show where the device is and where it is going.

Fortunately, integrating basic mapping features into your Android project is fairly easy. And there is also a fair bit of power available to you, if you want to get fancy.

Terms, Not of Endearment

Integrating Google Maps into your own application requires agreeing to a fairly lengthy set of legal terms. These terms include clauses that you may find unpalatable.

If you are considering Google Maps, please review these terms closely to determine if your intended use will not run afoul of any clauses. You are strongly recommended to seek professional legal counsel if there are any potential areas of conflict.

Also, keep your eyes peeled for other mapping options, based on other sources of map data, such as OpenStreetMap (http://www.openstreetmap.org/).

Piling On

As of Android l.5, Google Maps is not strictly part of the Android SDK. Instead, it is part of the Google APIs add-on, an extension of the stock SDK. The Android add-on system provides hooks for other subsystems that may be part of some devices but not others.

NOTE: Google Maps is not part of the Android open source project, and undoubtedly there will be some devices that lack Google Maps due to licensing issues. For example, at the time of this writing, the Archos 5 Android tablet does not have Google Maps.

By and large, the fact that Google Maps is in an add-on does not affect your day-to-day development. However, bear in mind the following:

- You will need to create your project with a suitable target to ensure the Google Maps APIs will be available.
- To test your Google Maps integration, you will also need an AVD that supports the Google Maps API.

The Bare Bones

Far and away the simplest way to get a map into your application is to create your own subclass of MapActivity. Like ListActivity, which wraps up some of the smarts behind having an activity dominated by a ListView, MapActivity handles some of the nuances of setting up an activity dominated by a MapView.

In your layout for the MapActivity subclass, you need to add an element named, at the time of this writing, com.google.android.maps.MapView. This is the "longhand" way to spell out the names of widget classes, by including the full package name along with the class name. This is necessary because MapView is not in the com.google.android.widget namespace. You can give the MapView widget whatever android:id attribute value you want, plus handle all the layout details to have it render properly alongside your other widgets.

However, you do need to have these two items:

- android:apiKey, which in production will need to be a Google Maps API key
- android:clickable="true", if you want users to be able to click and pan through your map

For example, from the Maps/NooYawk sample application, here is the main layout:

```
<?xml version="l.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <com.google.android.maps.MapView android:id="@+id/map"
    android:layout_width=" fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="<YOUR_ API_KEY>"
    android:clickable="true" />
</RelativeLayout>
```

We'll cover that mysterious apiKey later in this chapter, in the "The Key to It All" section. In addition, you will need a couple of extra things in your AndroidManifest.xml file:

- The INTERNET and ACCESS_COARSE_LOCATION permissions (the latter for use with the MyLocationOverlay class, described later in this chapter)
- Inside your <application>, a <uses-library> element with android:name ="com.google.android.maps", to indicate you are using one of the optional Android APIs

Here is the AndroidManifest.xml file for NooYawk:

```
<?xml version=”1.0” encoding=”utf-8”?>
<manifest xmlns:android=”http://schemas.android.com/apk/res/android”
    package="com.commonsware.android.maps”>
    <uses-permission android:name=”android. permission.INTERNET” />
    <uses-permission
android:name=”android.permission.ACCESS_COARES_LOCATION” />
      <application android:label="@string/app_name"
      android:icon="@drawable/cw">
      <uses-library    android:name=”com.google.android.maps"/>
      <activity android:name=".NooYawk”    android:label="@string/app_name">
          <intent-filter>
                      <action android:name="android.intent.action.MAIN" />
                      <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
      </activity>
   </application>
</manifest>
```

That is pretty much all you need for starters, plus to subclass your activity from MapActivity. If you were to do nothing else, and built that project and tossed it in the emulator, you would get a nice map of the world. Note, however, that MapActivity is abstract. You need to implement isRouteDisplayed() to indicate if you are supplying some sort of driving directions.

In theory, users could pan around the map using the D-pad. However, that's not terribly useful when they have the whole world in their hands.

Since a map of the world is not much good by itself, we need to add a few things, as described next.

Exercising Your Control

You can find your MapView widget by findViewById(), just as with any other widget. The widget itself offers a getMapController() method. Between the MapView and MapController, you have a fair bit of capability to determine what the map shows and how it behaves. The following sections cover zoom and center, the features you will most likely want to use.

Zoom

The map of the world you start with is rather broad. Usually, people looking at a map on a phone will be expecting something a bit narrower in scope, such as a few city blocks.

You can control the zoom level directly via the setZoom() method on the MapController. This takes an integer representing the level of zoom, where 1 is the world view and 21 is the tightest zoom you can get. Each level is a doubling of the effective resolution: 1 has the equator measuring 256 pixels wide, while 21 has the equator measuring 268,435,456 pixels wide. Since the phone's display probably doesn't have 268,435,456 pixels in either dimension, the user sees a small map focused on one tiny corner of the globe. A level of 16 will show several city blocks in each dimension, which is probably a reasonable starting point for experimentation.

If you wish to allow users to change the zoom level, call setBuiltInZoomControls (true);, and the user will be able to zoom in and out of the map via zoom controls found at the bottom center of the map.

Center

Typically, you will need to control what the map is showing, beyond the zoom level, such as the user's current location or a location saved with some data in your activity. To change the map's position, call setCenter() on the MapController.

The setCenter() method takes a GeoPoint as a parameter. A GeoPoint represents a location, via latitude and longitude. The catch is that the GeoPoint stores latitude and longitude as integers representing the actual latitude and longitude multiplied by 1E6. This saves a bit of memory versus storing a float or double, and it greatly speeds up some internal calculations Android needs to do to convert the GeoPoint into a map position. However, it does mean you must remember to multiply the real-world latitude and longitude by 1E6.

Rugged Terrain

Just as the Google Maps service you use on your full-size computer can display satellite imagery, so can Android maps.

MapView offers toggleSatellite(), which, as the name suggests, toggles on and off the satellite perspective on the area being viewed. You can have the user trigger these via an options menu or, in the case of NooYawk, via key presses:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_S) {
                    map.setSatellite( ! map.isSatellite());
                return(true);
         }else if(keycode== KeyEvent.KEYCODE_Z) {
             Map.dispalyZoomControls(true);
             Return(true);
     }
      return(super.onKeyDown(keyCode, event));}
```

Layers upon Layers

If you have ever used the full-size edition of Google Maps, you are probably used to seeing things overlaid atop the map itself, such as pushpins indicating businesses near the location being searched. In map parlance (and, for that matter, in many serious graphic editors), the pushpins are on a layer separate from than the map itself, and what you are seeing is the composition of the pushpin layer atop the map layer.

Android's mapping allows you to create layers as well, so you can mark up the maps as you need to based on user input and your application's purpose. For example, NooYawk uses a layer to show where select buildings are located in the island of Manhattan.

Overlay Classes

Any overlay you want to add to your map needs to be implemented as a subclass of Overlay. There is an ItemizedOverlay subclass available if you are looking to add pushpins or the like; ItemizedOverlay simplifies this process.

To attach an overlay class to your map, just call getOverlays() on your MapView and add () your Overlay instance to it, as we do here with a custom SitesOverlay:

```
marker.setBounds(0, 0, marker.getIntrinsicWidth(),marker.getIntrinsicHeight());
map.getOverlays().add(new SitesOverlay(marker));
```

We will take a closer look at that marker in the next section.

Drawing the ItemizedOverlay

As the name suggests, ItemizedOverlay allows you to supply a list of points of interest to be displayed on the map-specifically, instances of OverlayItem. The overlay handles much of the drawing logic for you. Here are the minimum steps to make this work:

1. Override ItemizedOverlay<OverlayItem> as your own subclass (in this example, SitesOverlay).
2. In the constructor, build your roster of OverlayItem instances, and call populate() when they are ready for use by the overlay.
3. Implement size() to return the number of items to be handled by the overlay.
4. Override createItem() to return OverlayItem instances given an index.
5. When you instantiate your ItemizedOverlay subclass, provide it with a Drawable that represents the default icon (e.g., a pushpin) to display for each item.

The marker from the NooYawk constructor is the Drawable used for step 5. It shows a pushpin.

You may also wish to override draw() to do a better job of handling the shadow for your markers. While the map will handle casting a shadow for you, it appears you need to provide a bit of assistance for it to know where the bottom of your icon is, so it can draw the shadow appropriately.

For example, here is SitesOverlay:

```
  private class SitesOverlay extends ItemizedOverlay<OverlayItem> {
  private List<OverlayItem> items=new ArrayList<OverlayItem>();
  private Drawable marker=null;
  public SitesOverlay(Drawable marker){
          super(marker);
        this.marker=marker;
 items.add(new     OverlayItem(getPoint(40.748963847316034,-73.96807193756104),
                                                        "UN", "United Nations"));
 items.add(new      OverlayItem(getPoint(40.76866299974387,-73.98268461227417),
"Lincoln Center", "Home of Jazz at Lincoln Center"));
items.add                    (new                                    OverlayItem
(getPoint(40.765136435316755,-73.97989511489868),
                    "Carnegie Hall", "Where you go with practice, practice, practice"));
items.add(new      OverlayItem(getPoint(40.70686417491799,-74.01572942733765),
                    "The Downtown Club", "Original home of the Heisman Trophy"));
          populate();}
@Override
protected    OverlayItem createItem(int i) {
      return(items.get(i));}
@Override
public void draw(Canvas canvas, MapView mapView, boolean shadow)
             super.draw(canvas, mapView, shadow);
            boundCenterBottom(marker);}
@Override
protected boolean onTap(int i) {
     Toast.makeText(NooYawk.this,items.get(i).getSnippet(),
                        Toast.LENGTH_SHORT) .show();
     return(true);}
@Override
public int size() {
return(items.size());
    } }
```

Handling Screen Taps

An Overlay subclass can also implement onTap(), to be notified when the user taps the map, so the overlay can adjust what it draws. For example, in full-size Google Maps, clicking a pushpin pops up a bubble with information about the business at that pin's location. With onTap(), you can do much the same in Android.

The onTap() method for ItemizedOverlay receives the index of the OverlayItem that was clicked. It is up to you to do something worthwhile with this event.

In the case of SitesOverlay, as shown in the preceding section, onTap() looks like this:

```
@Override
```

```
protected boolean onTap(int i) {
   Toast.makeText(NooYawk.this,items.get(i).getSnippet(),
                                 Toast.LENGTH_SHORT).show();
      return(true),}
```

Here, we just toss up a short Toast with the snippet from the OverlayItem, returning true to indicate we handled the tap.

My, Myself, and MyLocationOverlay

Android has a built-in overlay to handle two common scenarios:

- Showing where you are on the map, based on GPS or other location-providing logic
- Showing where you are pointed, based on the built-in compass sensor, where available

All you need to do is create a MyLocationOverlay instance, add it to your MapView's list of overlays, and enable and disable the desired features at appropriate times.

The "at appropriate times" notion is for maximizing battery life. There is no sense in updating locations or directions when the activity is paused, so it is recommended that you enable these features in onResume() and disable them in onPause().

For example, NooYawk will display a compass rose using MyLocationOverlay. To do this, we first need to create the overlay and add it to the list of overlays:

```
me=new MyLocationOverlay(this, map);
map.getOverlays().add(me);
```

Then we enable and disable the compass rose as appropriate:

```
@Override
public void onResume() {
   super.onResume();
   me.enableCompass();}
@Override
public void onPause(){
   super.onPause();
   me.disableCompass();}
```

The Key to It All

If you actually download the source code for the book, compile the NooYawk project, install it in your emulator, and run it, you will probably see a screen with a grid and a couple of pushpins, but no actual maps.

That's because the API key in the source code is invalid for your development machine. Instead, you will need to generate your own API key(s) for use with your application.

Full instructions for generating API keys for development and production use can be found on the Android web site (http://code.google.com/android/add-ons/google -apis/mapkey.html). In the interest of brevity, let's focus on the narrow case of getting NooYawk running in your emulator. Doing this requires the following steps:

1. Visit the API key signup page and review the terms of service.
2. Reread those terms of service and make really sure you want to agree to them.
3. Find the MD5 digest of the certificate used for signing your debug-mode applications.
4. On the API key signup page, paste in that MD5 signature and submit the form.
5. On the resulting page, copy the API key and paste it as the value of apiKey in your MapView-using layout.

The trickiest part is finding the MD5 signature of the certificate used for signing your debug-mode applications. Actually, much of the complexity is merely in making sense of the concept.

All Android applications are signed using a digital signature generated from a certificate. You are automatically given a debug certificate when you set up the SDK, and there is a separate process for creating a self-signed certificate for use in your production applications. This signature process involves the use of the Java keytool and jar signer utilities. For the purposes of getting your API key, you only need to worry about keytool.

To get your MD5 digest of your debug certificate, if you are on Mac OS X or Linux, use the following command:

```
keytool -list -alias androiddebugkey –keystore ~ / .android/debug.keystore -storepass android -keypass android
```

On other development platforms, you will need to replace the value of the –keystore switch with the location for your platform and user account:

- On Windows XP, use C:\Documents and Settings\<user>\.android\debug.

keystore.

- On Windows Vista/Windows 7, use C:\Users\<user>\.android\debug.keystore (where <user> is your account name).

The second line of the output contains your MD5 digest, as a series of pairs of hex digits separated by colons.