第 34 章

使用SQLite本地数据库

相比使用JSON等简单的文件格式存取数据，使用大量或复杂数据集的应用通常需要更强大的数据处理能力。在RunTracker应用中，用户可持续跟踪定位自己的地理位置，并由此产生大量数据。Android提供了SQLite数据库来处理这些数据。以磁盘上单个文件的形式存在，SQLite是一个开源的跨平台库，并具有一套强大的关系型数据库API可供使用。

Android内置了操作SQLite的Java前端，该前端的`SQLiteDatabase`类负责提供`Cursor`实例形式的结果集。本章，我们将为RunTracker应用创建数据存储机制，以利用数据库存储旅行数据以及地理位置信息。同时还将新建一个旅程列表activity和fragment，以允许用户创建、跟踪多个旅程。

34.1 在数据库中存储旅程和地理位置信息

为存储数据到数据库，首先需定义数据库结构并打开数据库。这是一种常见任务，因此Android提供了一个帮助类。`SQLiteOpenHelper`类封装了一些存储应用数据的常用数据库操作，如创建、打开、以及更新数据库等。

在RunTracker应用中，以`SQLiteOpenHelper`为父类，我们将创建一个`RunDatabaseHelper`类。通过引用一个私有的`RunDatabaseHelper`类实例，`RunManager`类将为应用其他部分提供统一API，用于数据的插入、查询以及其他一些数据管理操作。而`RunDatabaseHelper`类会提供各种方法供`RunManager`调用，以实现其定义的大部分API。

设计应用的数据库存储API时，我们通常是为需创建和管理的各类数据库创建一个`SQLiteOpenHelper`子类。然后，再为需要访问的不同SQLite数据库文件创建一个子类实例。包括RunTracker在内，大多应用只需要一个`SQLiteOpenHelper`子类，并与其余应用组件共享一个类实例。

要创建数据库，首先应考虑它的结构。面向对象编程语言中，最常见的数据库设计模式是为应用数据模型层的每个类都提供一张数据库表。RunTracker应用中有两个类需要存储：`Run`和`Location`类。

因此我们需创建两张表：run和location表。一个旅程（Run）可包含多个地理位置信息（`Location`），所以location表将包含一个`run_id`外键与run表的`_id`列相关联。两张表的结构如图34-1所示。

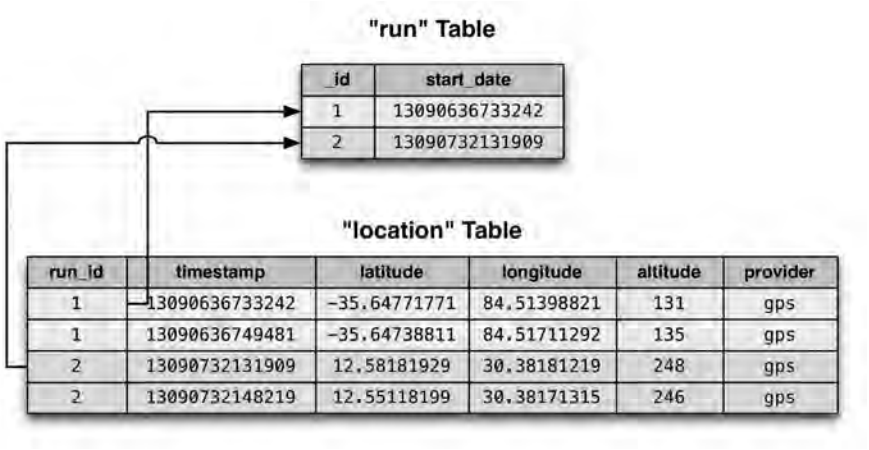

图34-1 RunTracker应用的数据库结构

添加代码清单34-1所示代码，新建RunDatabaseHelper类。

代码清单34-1 基本RunDatabaseHelper类（RunDatabaseHelper.java）

```
public class RunDatabaseHelper extends SQLiteOpenHelper {
    private static final String DB_NAME = "runs.sqlite";
    private static final int VERSION = 1;

    private static final String TABLE_RUN = "run";
    private static final String COLUMN_RUN_START_DATE = "start_date";

    public RunDatabaseHelper(Context context) {
        super(context, DB_NAME, null, VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // Create the "run" table
        db.execSQL("create table run (" +
            "_id integer primary key autoincrement, start_date integer)");
        // Create the "location" table
        db.execSQL("create table location (" +
            " timestamp integer, latitude real, longitude real, altitude real," +
            " provider varchar(100), run_id integer references run(_id))");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // Implement schema changes and data massage here when upgrading
    }

    public long insertRun(Run run) {
        ContentValues cv = new ContentValues();
        cv.put(COLUMN_RUN_START_DATE, run.getStartDate().getTime());
        return getWritableDatabase().insert(TABLE_RUN, null, cv);
    }
}
```

以上代码可以看出，实现SQLiteOpenHelper子类需覆盖两个方法：onCreate(SQLiteDatabase)和onUpgrade (SQLiteDatabase, int, int)方法。在onCreate(...)方法中，应为新建数据库创建表结构。在onUpgrade(...)方法中，可执行迁移代码，实现不同版本间的数据库结构升级或转换。

通常还可以实现一个简单的构造方法，为超类提供必要的初始化参数。这里，我们传入了数据库文件名的常量、可选CursorFactory参数的null值，以及数据库版本号整型常量。

尽管对RunTracker应用来说并非必要，但SQLiteOpenHelper类有着管理不同版本数据库结构的能力。因此，其数据库版本号应为一个从1开始的递增整数值。实际应用中，每次对数据库结构做出调整后，我们都应增加版本号常量，并在onUpgrade(...)方法中编写代码，以处理不同版本间数据库结构或数据的必要改变。

这里，我们实现了onCreate(...)方法，用以在新建数据库上执行CREATE TABLE的两条SQL语句。同时还实现了insertRun(Run)方法，用以在run数据表中插入一条新纪录并返回其ID。run表只有旅行开始日期一个数据字段，此处通过ContentValues对象表示的栏位名与值的名值对，将long类型的开始日期存入到数据库中。

SQLiteOpenHelper类有两个访问SQLiteDatabase实例的方法：getWritableDatabase()和getReadableDatabase()方法。这里的使用模式是，需要可写数据库时，使用getWritableDatabase()方法；需要只读数据库时，则使用getReadableDatabase()方法。实践中，对于既定的SQLiteOpenHelper类实例，两种实现方法的调用将返回同样的SQLiteDatabase实例。但在某些罕见场景中，如磁盘空间满了，则可能无法获得可写数据库，而只能获得只读数据库。

为支持查询多张run数据库表并在应用中加以区分，需为Run类添加ID属性。在Run类中，添加代码清单34-2所示更新代码。

代码清单34-2　为Run类添加ID属性（Run.java）

```
public class Run {
    private long mId;
    private Date mStartDate;

    public Run() {
        mId = -1;
        mStartDate = new Date();
    }

    public long getId() {
        return mId;
    }

    public void setId(long id) {
        mId = id;
    }

    public Date getStartDate() {
        return mStartDate;
    }
```

接下来，需升级RunManager类以使用新数据库，也即应用各部分共享的数据存取API的更新。添加代码清单34-3所示代码，实现Run类数据的存储。

代码清单34-3 管理当前Run的数据（RunManager.java）

```
public class RunManager {
    private static final String TAG = "RunManager";

    private static final String PREFS_FILE = "runs";
    private static final String PREF_CURRENT_RUN_ID = "RunManager.currentRunId";

    public static final String ACTION_LOCATION =
        "com.bignerdranch.android.runtracker.ACTION_LOCATION";

    private static final String TEST_PROVIDER = "TEST_PROVIDER";

    private static RunManager sRunManager;
    private Context mAppContext;
    private LocationManager mLocationManager;
    private RunDatabaseHelper mHelper;
    private SharedPreferences mPrefs;
    private long mCurrentRunId;

    private RunManager(Context appContext) {
        mAppContext = appContext;
        mLocationManager = (LocationManager)mAppContext
            .getSystemService(Context.LOCATION_SERVICE);
        mHelper = new RunDatabaseHelper(mAppContext);
        mPrefs = mAppContext.getSharedPreferences(PREFS_FILE, Context.MODE_PRIVATE);
        mCurrentRunId = mPrefs.getLong(PREF_CURRENT_RUN_ID, -1);
    }

    ...

    private void broadcastLocation(Location location) {
        Intent broadcast = new Intent(ACTION_LOCATION);
        broadcast.putExtra(LocationManager.KEY_LOCATION_CHANGED, location);
        mAppContext.sendBroadcast(broadcast);
    }

    public Run startNewRun() {
        // Insert a run into the db
        Run run = insertRun();
        // Start tracking the run
        startTrackingRun(run);
        return run;
    }

    public void startTrackingRun(Run run) {
        // Keep the ID
        mCurrentRunId = run.getId();
        // Store it in shared preferences
        mPrefs.edit().putLong(PREF_CURRENT_RUN_ID, mCurrentRunId).commit();
        // Start location updates
        startLocationUpdates();
    }

    public void stopRun() {
        stopLocationUpdates();
        mCurrentRunId = -1;
        mPrefs.edit().remove(PREF_CURRENT_RUN_ID).commit();
    }
```

```
    private Run insertRun() {
        Run run = new Run();
        run.setId(mHelper.insertRun(run));
        return run;
    }
}
```

一起来看下刚添加到RunManager类的新方法。startNewRun()方法调用insertRun()方法，创建并插入新的Run到数据库中，然后传递给startTrackingRun(Run)方法并对其进行跟踪，最后再返回给调用者。在还没有可跟踪的旅程存在时，可在RunFragment类中使用startNewRun()方法，以响应Start按钮。

重启当前旅程的跟踪操作时，RunFragment类也会直接使用startTrackingRun(Run)方法。该方法将Run类传入的ID分别存储在实例变量和shared preferences中。通过这种存储方式，即使在应用完全停止的情况下，仍可重新取回ID。RunFragment类的构造方法负责完成此类工作。

最后，stopRun()方法停止更新地理位置数据，并清除当前旅程的ID。RunFragment类将使用该方法实施Stop按钮。

提到RunFragment类，现在是时候在该类中使用RunManager类的新方法了。参照代码清单34-4，完成代码更新。

代码清单34-4　更新启停按钮的相关代码（RunFragment.java）

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_run, container, false);

    ...

    mStartButton = (Button)view.findViewById(R.id.run_startButton);
    mStartButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            mRunManager.startLocationUpdates();
            mRun = new Run();
            mRun = mRunManager.startNewRun();
            updateUI();
        }
    });

    mStopButton = (Button)view.findViewById(R.id.run_stopButton);
    mStopButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            mRunManager.stopLocationUpdates();
            mRunManager.stopRun();
            updateUI();
        }
    });

    updateUI();

    return view;
}
```

接下来，为响应LocationManager的位置数据更新，需将Location对象插入数据库中。与插入Run对象类似，我们需在RunDatabaseHelper类和RunManager类中分别添加一个方法，用于插入当前run的地理位置数据。然而，与插入Run对象不同，无论用户界面是否可见，或应用是否运行，RunTracker应用都应在收到更新数据时实现地理位置数据的插入。为处理这种需求，独立的BroadcastReceiver是最好的选择。

首先，在RunDatabaseHelper类中添加insertLocation(long, Location)方法，如代码清单34-5所示。

代码清单34-5 插入地理位置数据到数据库中（RunDatabaseHelper.java）

```
public class RunDatabaseHelper extends SQLiteOpenHelper {
    private static final String DB_NAME = "runs.sqlite";
    private static final int VERSION = 1;

    private static final String TABLE_RUN = "run";
    private static final String COLUMN_RUN_START_DATE = "start_date";

    private static final String TABLE_LOCATION = "location";
    private static final String COLUMN_LOCATION_LATITUDE = "latitude";
    private static final String COLUMN_LOCATION_LONGITUDE = "longitude";
    private static final String COLUMN_LOCATION_ALTITUDE = "altitude";
    private static final String COLUMN_LOCATION_TIMESTAMP = "timestamp";
    private static final String COLUMN_LOCATION_PROVIDER = "provider";
    private static final String COLUMN_LOCATION_RUN_ID = "run_id";

    ...

    public long insertLocation(long runId, Location location) {
        ContentValues cv = new ContentValues();
        cv.put(COLUMN_LOCATION_LATITUDE, location.getLatitude());
        cv.put(COLUMN_LOCATION_LONGITUDE, location.getLongitude());
        cv.put(COLUMN_LOCATION_ALTITUDE, location.getAltitude());
        cv.put(COLUMN_LOCATION_TIMESTAMP, location.getTime());
        cv.put(COLUMN_LOCATION_PROVIDER, location.getProvider());
        cv.put(COLUMN_LOCATION_RUN_ID, runId);
        return getWritableDatabase().insert(TABLE_LOCATION, null, cv);
    }
```

现在，在RunManager类中添加代码，实现当前跟踪旅程地理位置数据的插入，如代码清单34-6所示。

代码清单34-6 插入当前旅程的地理位置数据（RunManager.java）

```
    private Run insertRun() {
        Run run = new Run();
        run.setId(mHelper.insertRun(run));
        return run;
    }

    public void insertLocation(Location loc) {
        if (mCurrentRunId != -1) {
            mHelper.insertLocation(mCurrentRunId, loc);
        } else {
            Log.e(TAG, "Location received with no tracking run; ignoring.");
```

```
        }
    }
}
```

最后，需挑选合适的地方调用insertLocation(Location)新方法。这里，我们选择了独立的Broadcast- Receiver。使用它完成此项任务，可保证location intent能够得到及时处理，而不用关心RunTraker应用的其余部分是否正在运行。这需要创建一个名为TrackingLocationReceiver的定制LocationReceiver子类，并在manifest配置文件中使用intent filter完成登记。

使用以下代码，创建TrackingLocationReceiver类。

代码清单34-7　简单优雅的TrackingLocationReceiver类（TrackingLocationReceiver.java）

```
public class TrackingLocationReceiver extends LocationReceiver {

    @Override
    protected void onLocationReceived(Context c, Location loc) {
        RunManager.get(c).insertLocation(loc);
    }

}
```

在manifest配置文件中登记TrackingLocationReceiver类，让其响应定制的ACTION_LOCATION操作，如代码清单34-8所示。

代码清单34-8　配置使用TrackingLocationReceiver（AndroidManifest.xml）

```
<application
  android:allowBackup="true"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme">

  ...

  <receiver android:name=".LocationReceiver"
  <receiver android:name=".TrackingLocationReceiver"
    android:exported="false">
    <intent-filter>
      <action android:name="com.bignerdranch.android.runtracker.ACTION_LOCATION"/>
    </intent-filter>
  </receiver>

</application>
```

完成以上代码添加后，该是运行应用查看辛苦工作成果的时候了。现在，RunTracker应用已可持续跟踪用户旅程（除非人为停止），即使应用被终止或已退出。为确认应用运行是否符合预期，可在TrackingLocationReceiver类的onLocationReceived(...)方法中添加日志代码，然后开启旅程跟踪，接着终止或退出应用并同时观察LogCat窗口。

34.2　查询数据库中的旅程列表

现在，RunTracker应用可将新的旅程及它们的地理位置信息写入数据库，但正如前文所述，

每次点击Start按钮，RunFragment都会创建一条新的旅程记录。本小节，我们将创建新的activity和fragment，以显示旅程列表，并允许用户创建新的旅程记录和查看现有旅程记录。实现界面类似于CriminalIntent应用的crime记录列表显示界面，不过有一点不同的是，列表中的数据来自于数据库而不是应用内存或文件存储。

查询SQLiteDatabase可返回描述结果的Cursor实例。CursorAPI使用简单，且可灵活支持各种类型的查询结果。Cursor将结果集看做是一系列的数据行和数据列，但仅支持String以及原始数据类型的值。

然而，作为Java程序员，我们习惯于使用对象来封装模型层数据，如Run和Location对象。既然我们已经有了代表对象的数据库表，如果能从Cursor中取得这些对象的实例就再好不过了。

为实现以上目标，本章我们将使用一个名为CursorWrapper的内置Cursor子类。CursorWrapper类设计用于封装当前的Cursor类，并转发所有的方法调用给它。CursorWrapper类本身没有多大用途，但作为超类，它为创建适用于模型层对象的定制cursor打下了良好基础。

更新RunDatabaseHelper类，添加新的queryRuns()方法，并返回一个RunCursor，从而列出按日期排序的旅程列表。这实际就是CursorWrapper模式的一个例子。

代码清单34-9 查询旅程记录（RunDatabaseHelper.java）

```
public class RunDatabaseHelper extends SQLiteOpenHelper {
    private static final String DB_NAME = "runs.sqlite";
    private static final int VERSION = 1;

    private static final String TABLE_RUN = "run";
    private static final String COLUMN_RUN_ID = "_id";
    private static final String COLUMN_RUN_START_DATE = "start_date";

    ...

    public RunCursor queryRuns() {
        // Equivalent to "select * from run order by start_date asc"
        Cursor wrapped = getReadableDatabase().query(TABLE_RUN,
                null, null, null, null, null, COLUMN_RUN_START_DATE + " asc");
        return new RunCursor(wrapped);
    }

    /**
     * A convenience class to wrap a cursor that returns rows from the "run" table.
     * The {@link getRun()} method will give you a Run instance representing
     * the current row.
     */
    public static class RunCursor extends CursorWrapper {

        public RunCursor(Cursor c) {
            super(c);
        }

        /**
         * Returns a Run object configured for the current row,
         * or null if the current row is invalid.
         */
        public Run getRun() {
```

```
            if (isBeforeFirst() || isAfterLast())
                return null;
            Run run = new Run();
            long runId = getLong(getColumnIndex(COLUMN_RUN_ID));
            run.setId(runId);
            long startDate = getLong(getColumnIndex(COLUMN_RUN_START_DATE));
            run.setStartDate(new Date(startDate));
            return run;
        }
    }
}
```

RunCursor只定义了两个方法：一个简单的构造方法和getRun()方法。getRun()方法首先检查确认cursor未越界，然后基于当前记录的列值创建并配置Run实例。RunCursor的使用者可遍历结果集里的行数，并对每一行调用getRun()方法，以此得到一个对象而不是一堆原始数据。

因此，RunCursor主要负责将run表中的各个记录转化为Run实例，并按要求对结果进行组织排序。

queryRuns()方法执行SQL查询语句，提供未经处理的cursor给新的RunCursor，然后返回给queryRuns()方法调用者。现在可以在RunManager类中使用queryRuns()方法了，后面我们还会在RunListFragment类中用到它。

代码清单34-10　代理旅程查询（RunManager.java）

```
private Run insertRun() {
    Run run = new Run();
    run.setId(mHelper.insertRun(run));
    return run;
}

public RunCursor queryRuns() {
    return mHelper.queryRuns();
}

public void insertLocation(Location loc) {
    if (mCurrentRunId != -1) {
        mHelper.insertLocation(mCurrentRunId, loc);
    } else {
        Log.e(TAG, "Location received with no tracking run; ignoring.");
    }
}
```

34.3　使用 CursorAdapter 显示旅程列表

为在用户界面显示旅程列表，应首先完成一些准备工作。如代码清单34-11所示，为应用新建一个名为RunListActivity的默认activity。然后按照代码清单34-12所示代码，在manifest配置文件中将其设置为默认activity。暂时忽略有关RunListFragment的错误提示。

代码清单34-11　强大的RunListActivity（RunListActivity.java）

```
public class RunListActivity extends SingleFragmentActivity {

    @Override
```

```
        protected Fragment createFragment() {
            return new RunListFragment();
        }

    }
```

代码清单34-12 配置声明RunListActivity（AndroidManifest.xml）

```
    <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme">
      <activity android:name=".RunActivity"
      <activity android:name=".RunListActivity"
        android:label="@string/app_name">
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
      <activity android:name=".RunActivity"
                android:label="@string/app_name" />
      <receiver android:name=".TrackingLocationReceiver"
        android:exported="false">
        <intent-filter>
```

现在可以创建RunListFragment的基本结构了，如代码清单34-13所示。当前，cursor的加载与关闭分别发生在onCreate(Bundle)和onDestroy()方法中，但我们不推荐这种做法，因为这不仅强制数据库查询在主线程（UI）上执行，在极端情况下甚至会引发糟糕的ANR（应用无响应）。下一章，我们将从原生的实现转变到另一种实现模式，也即使用Loader将数据库查询转移到后台运行。

代码清单34-13 基本RunListFragment（RunListFragment.java）

```
    public class RunListFragment extends ListFragment {

        private RunCursor mCursor;

        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            // Query the list of runs
            mCursor = RunManager.get(getActivity()).queryRuns();
        }

        @Override
        public void onDestroy() {
            mCursor.close();
            super.onDestroy();
        }

    }
```

如果不能将数据提供给关联着RunListFragment的ListView，仅有RunCursor用处也不大。Android API（以及支持库）中的CursorAdapter类恰好可以解决这一问题。这里，我们只需创建CursorAdapter的一个子类，并实现其提供的几个方法，然后CursorAdapter会帮我们完成创建和

重复使用显示视图的所有逻辑处理。

更新RunListFragment类，实现一个RunCursorAdapter类，如代码清单34-14所示。

代码清单34-14　实现RunCursorAdapter类（RunListFragment.java）

```
public class RunListFragment extends ListFragment {

    private RunCursor mCursor;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Query the list of runs
        mCursor = RunManager.get(getActivity()).queryRuns();
        // Create an adapter to point at this cursor
        RunCursorAdapter adapter = new RunCursorAdapter(getActivity(), mCursor);
        setListAdapter(adapter);
    }

    @Override
    public void onDestroy() {
        mCursor.close();
        super.onDestroy();
    }

    private static class RunCursorAdapter extends CursorAdapter {

        private RunCursor mRunCursor;

        public RunCursorAdapter(Context context, RunCursor cursor) {
            super(context, cursor, 0);
            mRunCursor = cursor;
        }

        @Override
        public View newView(Context context, Cursor cursor, ViewGroup parent) {
            // Use a layout inflater to get a row view
            LayoutInflater inflater = (LayoutInflater)context
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
            return inflater
                .inflate(android.R.layout.simple_list_item_1, parent, false);
        }

        @Override
        public void bindView(View view, Context context, Cursor cursor) {
            // Get the run for the current row
            Run run = mRunCursor.getRun();

            // Set up the start date text view
            TextView startDateTextView = (TextView)view;
            String cellText =
                context.getString(R.string.cell_text, run.getStartDate());
            startDateTextView.setText(cellText);
        }
    }
}
```

CursorAdapter类的构造方法需要一个Context、一个Cursor以及一个整数flag参数。为提

倡使用loader，大多数flag已被废弃或有一些问题存在，因此，这里传入的值为0。另外，为避免后面的类型转换，我们将RunCursor存储在一个实例变量中。

接下来，实现newView(Context, Cursor, ViewGroup)方法，并返回一个代表cursor中当前数据行的View。这里我们实例化系统资源android.R.layout.simple_list_item_1，得到一个简单的TextView组件。列表中所有视图看上去都一样，因此这里就完成了列表视图的准备。

当需要配置视图显示cursor中的一行数据时，CursorAdapter将调用bindView(View, Context,Cursor)方法。该方法需要的View参数应该总是前面newView()方法返回的View。

bindView(...)方法的实现相对简单。首先，从RunCursor中取得当前行的Run（这里的RunCursor 是CursorAdapter已经定位的cursor）。然后，假定传入的视图是一个TextView，我们对其进行配置以显示Run的简单描述。

完成全部代码添加后，运行RunTracker应用。我们应该能看到之前创建的旅程记录列表（假定本章前面已实现数据库插入代码，并启动了旅程跟踪。）。如果还没有任何旅程记录，也无需担心，我们可通过下一节添加的UI界面来创建新的旅程。

34.4 创建新旅程

如同CrimininalIntent应用中的处理，使用选项菜单或操作栏菜单项，添加可创建旅程的用户界面非常容易。首先需创建菜单资源，如代码清单34-15所示。

代码清单34-15 用于添加旅程的选项菜单（run_list_options.xml）

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
  <item android:id="@+id/menu_item_new_run"
    android:showAsAction="always"
    android:icon="@android:drawable/ic_menu_add"
    android:title="@string/new_run"/>
</menu>
```

新建菜单需要new_run字符串资源，因此将其添加到strings.xml中。

代码清单34-16 添加New Run字符串资源（strings.xml）

```
  <string name="stop">Stop</string>
  <string name="gps_enabled">GPS Enabled</string>
  <string name="gps_disabled">GPS Disabled</string>
  <string name="cell_text">Run at %1$s</string>
    <string name="new_run">New Run</string>
</resources>
```

然后，参照代码清单34-17，在RunListFragment类中创建选项菜单，并响应菜单项的选择。

代码清单34-17 通过选项菜单新建旅程记录（RunListFragment.java）

```
public class RunListFragment extends ListFragment {
    private static final int REQUEST_NEW_RUN = 0;

    private RunCursor mCursor;
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setHasOptionsMenu(true);
    // Query the list of runs
    mCursor = RunManager.get(getActivity()).queryRuns();
    // Create an adapter to point at this cursor
    RunCursorAdapter adapter = new RunCursorAdapter(getActivity(), mCursor);
    setListAdapter(adapter);
}

...

@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    super.onCreateOptionsMenu(menu, inflater);
    inflater.inflate(R.menu.run_list_options, menu);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case R.id.menu_item_new_run:
        Intent i = new Intent(getActivity(), RunActivity.class);
        startActivityForResult(i, REQUEST_NEW_RUN);
        return true;
    default:
        return super.onOptionsItemSelected(item);
    }
}

@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (REQUEST_NEW_RUN == requestCode) {
        mCursor.requery();
        ((RunCursorAdapter)getListAdapter()).notifyDataSetChanged();
    }
}

private static class RunCursorAdapter extends CursorAdapter {

    private RunCursor mRunCursor;
```

以上代码实现中，只需解读一处新知识点，即在用户重新返回应用页面时，可使用onActivityResult(...)方法强行重新加载列表数据。需再次提醒的是，我们并不推荐此处在主线程上重新查询cursor的做法。在下一章的学习中，我们会使用Loader进行替换。

34.5　管理现有旅程

接下来，我们来实现查看任意旅程列表项的明细信息。要查看明细，RunFragment需要传入的旅程ID参数（argument）的支持。RunActivity负责托管RunFragment，因此它也需要旅程ID附加信息。

首先，将argument附加给RunFragment，并添加一个newInstance(long)便利方法，如代码

清单34-18所示。

代码清单34-18 附加旅程ID参数（RunFragment.java）

```
public class RunFragment extends Fragment {
    private static final String TAG = "RunFragment";
    private static final String ARG_RUN_ID = "RUN_ID";

    ...

    private TextView mStartedTextView, mLatitudeTextView,
        mLongitudeTextView, mAltitudeTextView, mDurationTextView;

    public static RunFragment newInstance(long runId) {
        Bundle args = new Bundle();
        args.putLong(ARG_RUN_ID, runId);
        RunFragment rf = new RunFragment();
        rf.setArguments(args);
        return rf;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

然后，在RunActivity中分情形使用fragment，如果intent包含RUN_ID附加信息，则使用newInstance(long)方法创建RunFragment；如不包含，则仍使用默认的构造方法。如代码清单34-19所示。

代码清单34-19 添加旅程ID附加信息（RunActivity.java）

```
public class RunActivity extends SingleFragmentActivity {
    /** A key for passing a run ID as a long */
    public static final String EXTRA_RUN_ID =
        "com.bignerdranch.android.runtracker.run_id";

    @Override
    protected Fragment createFragment() {
        return new RunFragment();
        long runId = getIntent().getLongExtra(EXTRA_RUN_ID, -1);
        if (runId != -1) {
            return RunFragment.newInstance(runId);
        } else {
            return new RunFragment();
        }
    }

}
```

最后，在RunListFragment中响应列表项选择，使用已选旅程ID启动RunActivity，如代码清单34-20所示。

代码清单34-20 通过onListItemClick(...)方法启动现有旅程activity（RunListFragment.java）

```
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    // The id argument will be the Run ID; CursorAdapter gives us this for free
```

```
        Intent i = new Intent(getActivity(), RunActivity.class);
        i.putExtra(RunActivity.EXTRA_RUN_ID, id);
        startActivity(i);
    }

    private static class RunCursorAdapter extends CursorAdapter {
```

以上代码中有一处处理得比较精妙。因为我们指定了run_id表中的ID字段，CursorAdapter检测到该字段并将其作为id参数传递给了onListItemClick(...)方法。我们也因此能够直接将其作为附加信息传递给了RunActivity。真的好方便!

可惜，方便到此结束。要知道，仅使用ID参数启动RunFragment，我们还是无法显示现有的旅程信息。为在用户界面上显示已记录的旅程信息，还需查询数据库获取现有旅程的详细信息，其中包括它的最近一次记录的地理位置信息。

幸运的是，我们之前已做过类似的工作。在RunDatabaseHelper类中，添加queryRun(long)方法，返回指定ID旅程记录的RunCursor，如代码清单34-21所示。

代码清单34-21　查询单个旅程记录（RunDatabaseHelper.java）

```
    public RunCursor queryRun(long id) {
        Cursor wrapped = getReadableDatabase().query(TABLE_RUN,
                null, // All columns
                COLUMN_RUN_ID + " = ?", // Look for a run ID
                new String[]{ String.valueOf(id) }, // with this value
                null, // group by
                null, // having
                null, // order by
                "1"); // limit 1 row
        return new RunCursor(wrapped);
    }
```

传递给query(...)方法的许多参数很容易理解。我们从TABLE_RUN表中获取了所有数据列，然后以ID列过滤它们。这里，我们使用单元素字符串数组处理传递进来的ID参数并提供给“where”子句。限制查询只能返回一条记录，然后将结果封装到RunCursor中并返回。

然后，在RunManager类中，添加一个getRun(long)方法，并封装queryRun(long)方法的返回结果。如果有数据存在，则从第一条记录取出Run对象，如代码清单34-22所示。

代码清单34-22　实现getRun(long)方法（RunManager.java）

```
    public Run getRun(long id) {
        Run run = null;
        RunCursor cursor = mHelper.queryRun(id);
        cursor.moveToFirst();
        // If you got a row, get a run
        if (!cursor.isAfterLast())
            run = cursor.getRun();
        cursor.close();
        return run;
    }
```

以上代码中，从queryRun(long)方法取回RunCursor后，getRun(long)方法尝试从RunCursor的第一条记录获取Run对象。该方法首先让RunCursor移动到结果集的第一行。如有记录存

在，isAfterLast()方法会返回false值，这样我们就能从第一条记录安全获取到Run对象。

由于新方法的调用者无法接触到RunCursor，在返回旅程记录前，应记得调用RunCursor的close()方法，以便于数据库尽快从内存中释放与cursor相关联的资源。

完成以上工作后，现在，我们可更新 RunFragment类，以实现对现有旅程记录的管理。具体代码调整如代码清单34-23所示。

代码清单34-23 管理现有旅程记录（RunFragment.java）

```
public class RunFragment extends Fragment {
    private static final String TAG = "RunFragment";
    private static final String ARG_RUN_ID = "RUN_ID";

    private BroadcastReceiver mLocationReceiver = new LocationReceiver() {

        @Override
        protected void onLocationReceived(Context context, Location loc) {
            if (!mRunManager.isTrackingRun(mRun))
                return;
            mLastLocation = loc;
            if (isVisible())
                updateUI();
        }

        @Override
        protected void onProviderEnabledChanged(boolean enabled) {
            int toastText = enabled ? R.string.gps_enabled : R.string.gps_disabled;
            Toast.makeText(getActivity(), toastText, Toast.LENGTH_LONG).show();
        }

    };

    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        mRunManager = RunManager.get(getActivity());<

        // Check for a Run ID as an argument, and find the run
        Bundle args = getArguments();
        if (args != null) {
            long runId = args.getLong(ARG_RUN_ID, -1);
            if (runId != -1) {
                mRun = mRunManager.getRun(runId);
            }
        }
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_run, container, false);

        ...
```

```
            mStartButton = (Button)view.findViewById(R.id.run_startButton);
            mStartButton.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    mRun = mRunManager.startNewRun();
                    if (mRun == null) {
                        mRun = mRunManager.startNewRun();
                    } else {
                        mRunManager.startTrackingRun(mRun);
                    }
                    updateUI();
                }
            });

            ...

            return view;
        }

        ...

        private void updateUI() {
            boolean started = mRunManager.isTrackingRun();
            boolean trackingThisRun = mRunManager.isTrackingRun(mRun);

            if (mRun != null)
                mStartedTextView.setText(mRun.getStartDate().toString());

            int durationSeconds = 0;
            if (mRun != null && mLastLocation != null) {
                durationSeconds = mRun.getDurationSeconds(mLastLocation.getTime());
                mLatitudeTextView.setText(Double.toString(mLastLocation.getLatitude()));
                mLongitudeTextView.setText(Double.toString(mLastLocation.getLongitude()));
                mAltitudeTextView.setText(Double.toString(mLastLocation.getAltitude()));
            }
            mDurationTextView.setText(Run.formatDuration(durationSeconds));

            mStartButton.setEnabled(!started);
            mStopButton.setEnabled(started);
            mStopButton.setEnabled(started && trackingThisRun);
        }

    }
```

本着用户至上的原则，我们可以让RunFragment从数据库加载当前旅程的最近一次地理位置信息。这与加载Run的处理非常类似，但我们会新建LocationCursor来处理Location对象。

在RunDatabaseHelper类中，首先添加查询旅程记录最近一次地理位置信息的方法，然后再添加LocationCursor内部类，如代码清单34-24所示。

代码清单34-24　查询旅程记录的最近一次地理位置（RunDatabaseHelper.java）

```
    public LocationCursor queryLastLocationForRun(long runId) {
        Cursor wrapped = getReadableDatabase().query(TABLE_LOCATION,
                null, // All columns
                COLUMN_LOCATION_RUN_ID + " = ?", // limit to the given run
                new String[]{ String.valueOf(runId) },
```

```
            null, // group by
            null, // having
            COLUMN_LOCATION_TIMESTAMP + " desc", // order by latest first
            "1"); // limit 1
    return new LocationCursor(wrapped);
}


// ... After RunCursor ...

public static class LocationCursor extends CursorWrapper {

    public LocationCursor(Cursor c) {
        super(c);
    }

    public Location getLocation() {
        if (isBeforeFirst() || isAfterLast())
            return null;
        // First get the provider out so you can use the constructor
        String provider = getString(getColumnIndex(COLUMN_LOCATION_PROVIDER));
        Location loc = new Location(provider);
        // Populate the remaining properties
        loc.setLongitude(getDouble(getColumnIndex(COLUMN_LOCATION_LONGITUDE)));
        loc.setLatitude(getDouble(getColumnIndex(COLUMN_LOCATION_LATITUDE)));
        loc.setAltitude(getDouble(getColumnIndex(COLUMN_LOCATION_ALTITUDE)));
        loc.setTime(getLong(getColumnIndex(COLUMN_LOCATION_TIMESTAMP)));
        return loc;
    }
}
```

LocationCursor与RunCursor使用目的相同，但LocationCursor封装了用于从location数据表中返回记录的cursor，并将记录各字段转换为Location对象的属性。请注意这里的微妙之处，在该实现代码中，Location的构造方法需要定位服务提供者的名字，因此在设置其他属性前，我们应首先从当前数据行获取其名称。

queryLastLocationForRun(long)方法与queryRun(long)方法十分相似，但前者可找到与指定旅程相关联的最近一次地理位置信息，并将结果封装到LocationCursor中。

类似queryRun(long)方法的处理，我们应在RunManager中新建方法以封装queryLastLocationForRun(long)方法，并返回cursor中某条记录的Location，如代码清单34-25所示。

代码清单34-25 取得旅程的最近一次地理位置信息（RunManager.java）

```
public Location getLastLocationForRun(long runId) {
    Location location = null;
    LocationCursor cursor = mHelper.queryLastLocationForRun(runId);
    cursor.moveToFirst();
    // If you got a row, get a location
    if (!cursor.isAfterLast())
        location = cursor.getLocation();
    cursor.close();
    return location;
}
```

在创建了fragment后，我们即可在RunFragment中使用getLastLocation ForRun(...)方

法，以获取当前旅程的最近一次地理位置信息，如代码清单34-26所示。

代码清单34-26　获取当前旅程的最近一次地理位置信息（RunFragment.java）

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);
    mRunManager = RunManager.get(getActivity());

    // Check for a Run ID as an argument, and find the run
    Bundle args = getArguments();
    if (args != null) {
        long runId = args.getLong(ARG_RUN_ID, -1);
        if (runId != -1) {
            mRun = mRunManager.getRun(runId);
            mLastLocation = mRunManager.getLastLocationForRun(runId);
        }
    }
}
```

完成了这么多工作后，现在，RunTracker应既能创建并跟踪尽可能多的旅程（只要存储空间和电力够用），又能展示给用户准确明了的旅程及地理位置信息。旅程跟踪愉快！

34.6　挑战练习：识别当前跟踪的旅程

按照应用的当前实现，用户仅可通过手动点开列表项，以查看start和stop按钮状态的方式，来识别当前跟踪的旅程。如果有更便捷的方式，用户体验会更好。

作为简单的练习，请调整应用的列表项显示界面，为当前跟踪旅程的列表项添加识别图标，或区分于其他列表项的颜色。

作为更具挑战的练习，请使用通知信息告诉用户当前正跟踪的旅程，并在用户点击后，启动RunActivity。