北京交通大学

计算机专业实践与训练课程报告

小组：自学小组

成员：解仕奥（组长）胡劲鹰 孙龙

专业：计算机科学与技术

提交日期：2019年8月13日

目录

[1.设计内容](#_Toc7715_WPSOffice_Level1)  [3](#_Toc7715_WPSOffice_Level1)

[2.组员分工](#_Toc26110_WPSOffice_Level1) [4](#_Toc26110_WPSOffice_Level1)

[3.功能模块介绍](#_Toc26082_WPSOffice_Level1) [4](#_Toc26082_WPSOffice_Level1)

[3.1 地图定位以及跑步路线绘制](#_Toc12911_WPSOffice_Level2) [4](#_Toc12911_WPSOffice_Level2)

[3.2 目标设定](#_Toc22593_WPSOffice_Level2) [5](#_Toc22593_WPSOffice_Level2)

[3.3 跑步历史纪录储存](#_Toc17167_WPSOffice_Level2) [6](#_Toc17167_WPSOffice_Level2)

[3.4 速度，公里数，时间，步数显示](#_Toc10824_WPSOffice_Level2) [7](#_Toc10824_WPSOffice_Level2)

[3.5 步态识别](#_Toc2147_WPSOffice_Level2) [7](#_Toc2147_WPSOffice_Level2)

[4.前端代码原理](#_Toc15517_WPSOffice_Level1) [8](#_Toc15517_WPSOffice_Level1)

[4.1 地图实现模块](#_Toc21165_WPSOffice_Level2) [8](#_Toc21165_WPSOffice_Level2)

[4.2 主界面布局以及相关功能实现模块](#_Toc9777_WPSOffice_Level2) [9](#_Toc9777_WPSOffice_Level2)

[4.2.1地图显示实现代码原理](#_Toc26110_WPSOffice_Level3) [10](#_Toc26110_WPSOffice_Level3)

[4.2.2 跑步路径绘制实现代码原理](#_Toc26082_WPSOffice_Level3) [11](#_Toc26082_WPSOffice_Level3)

[4.2.3 相关控件说明代码](#_Toc15517_WPSOffice_Level3) [12](#_Toc15517_WPSOffice_Level3)

[4.3 步数统计代码原理](#_Toc13296_WPSOffice_Level2) [20](#_Toc13296_WPSOffice_Level2)

[4.4 三轴加速传感器代码原理](#_Toc21372_WPSOffice_Level2) [22](#_Toc21372_WPSOffice_Level2)

[5. 前后端数据传送代码原理](#_Toc4405_WPSOffice_Level3) [23](#_Toc4405_WPSOffice_Level3)

[6. 后端步态识别代码原理](#_Toc3194_WPSOffice_Level3) [25](#_Toc3194_WPSOffice_Level3)

[7. 遇到的问题及解决](#_Toc15088_WPSOffice_Level3) [26](#_Toc15088_WPSOffice_Level3)

[8. 总结感悟](#_Toc12911_WPSOffice_Level3) [26](#_Toc12911_WPSOffice_Level3)

1.设计内容

SuperSpeedRun App:

小组设计了一款运动App,本App借鉴乐跑App和一些其他的跑步App，拥有即时定位，跑步路线显示，跑步历史纪录，目标设定，速度，跑步的时间公里数等模块，并在基础上进行改进，增加了步态识别以及步数统计等功能，将前端的Android Studio设计与后端深度学习结合，打造一款受众面广的服务型App。

2.组员分工

首先，我们三个确定了App主题后，通过讨论确定了所包含的功能模块，并进行了明确的分工，在大体工作完成之后，我们一起讨论关于具体页面设计，logo，slogan等细节问题，在整个过程中，我们三个团结协作，有条不紊地完成了项目工作，每一个人都发挥了自己的作用。小组分工以及成员贡献比如下：

**解仕奥（贡献比 33%）**

1.前后端的连接以及传感器数据的传送

2.python后端深度学习实现步态识别（模型建立，训练，预测等）

3.ppt的制作以及课程报告撰写，答辩视频的录制

**孙龙（贡献比 33%）**

1.前端Android Studio交互界面设计

2.地图定位，跑步路线描绘等的功能

3.App后期的完善工作，程序演示视频的录制

**胡劲鹰：（贡献比 33%）**

1.步数统计功能的实现

2.通过Android Studio获得传感器数据

3.功能模块介绍

3.1 地图定位以及跑步路线绘制

****

**图 1 地图定位**

****

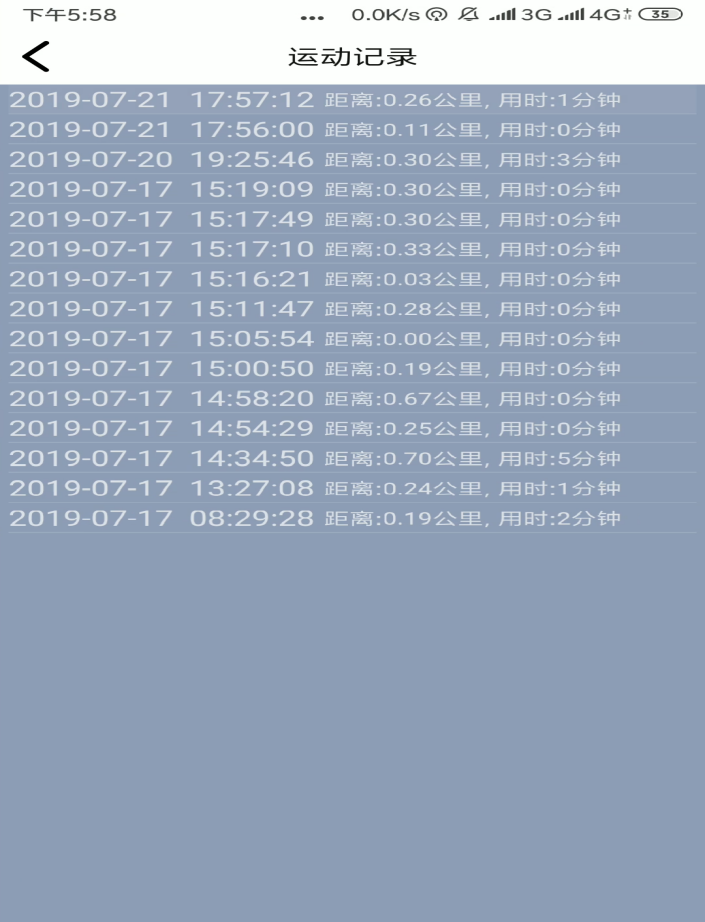
**图2 运动轨迹绘制**

3.2 目标设定

****

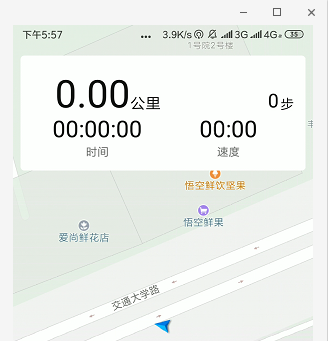
**图 3 跑步计划表**

3.3 跑步历史纪录储存

****

**图4 运动记录**

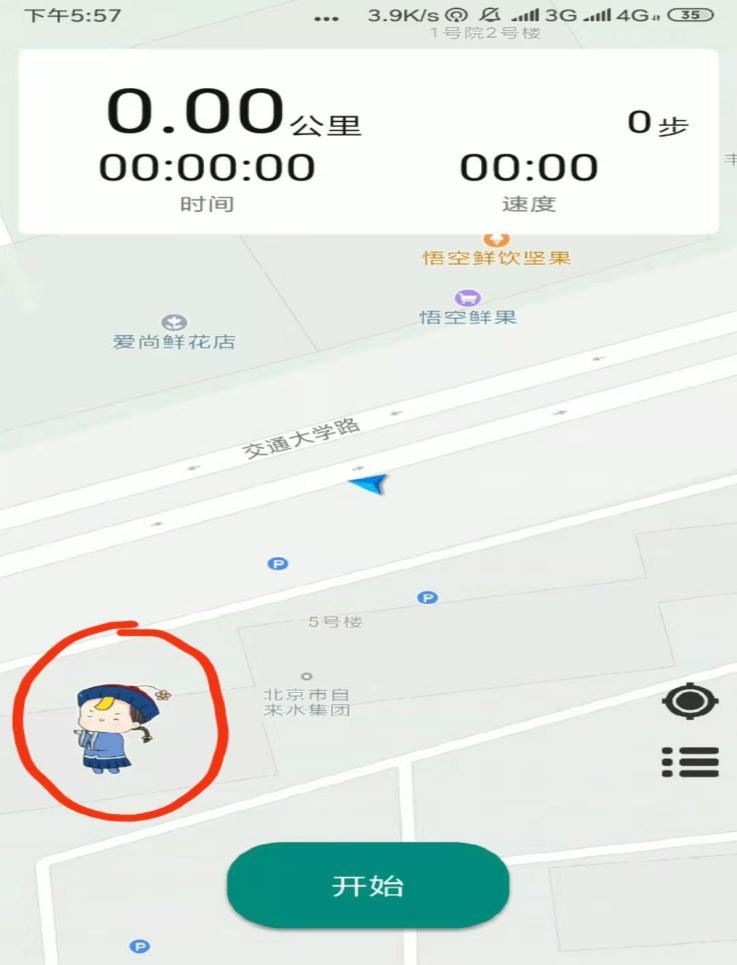
3.4 速度，公里数，时间，步数显示



**图5 速度等显示**

3.5 步态识别

通过表情包来实现六种步态的表述，分别为：坐着，站着，跑，走着，上楼，下楼

****

**图6 步态识别**

4.前端代码原理

4.1 地图实现模块

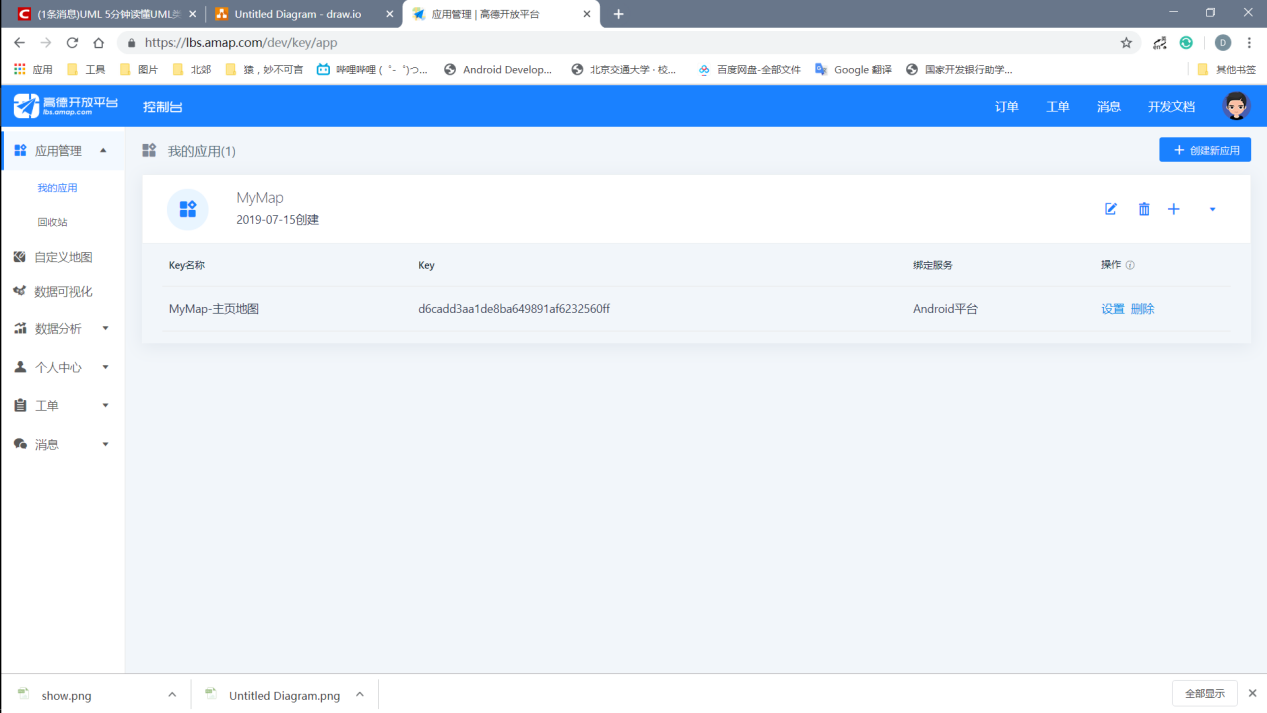
本模块主要基于高德地图SDK进行开发。

高德开放平台目前开放了Android 地图 SDK 以及 Android 地图 SDK 专业版两套地图SDK工具。

高德地图 Android SDK 是一套地图开发调用接口，开发者可以轻松地在自己的Android应用中加入地图相关的功能，包括：地图显示（含室内、室外地图）、与地图交互、在地图上绘制、兴趣点搜索、地理编码、离线地图等功能。

高德地图 Android SDK 专业版是在 Android SDK 已有服务的基础上，新增支持了自定义地图在线加载、自定义地图元素纹理等功能，便于开发者完成基于自身场景的更深层、更个性化地图的开发需求。

通过获取key来使用高德地图SDK。

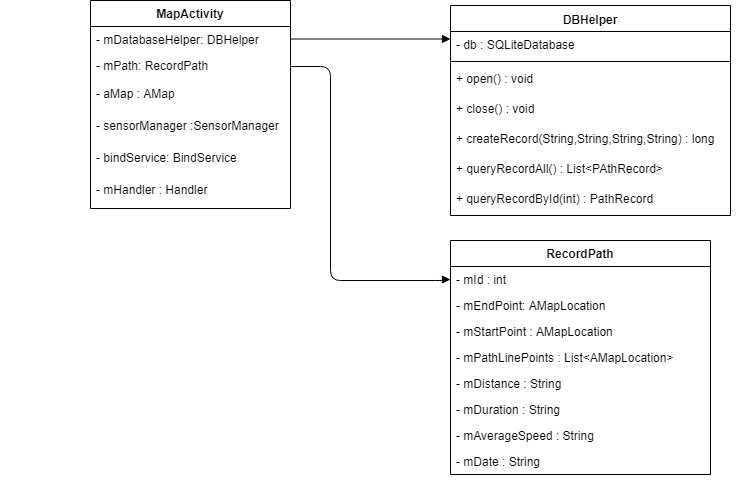


**图7 地图实现模块**

首先是地图界面，根布局采用FrameLayout，加入MapView 和RealativeLayout。这样就能使地图全屏显示，而控件悬浮在地图上，具有更好的视觉效果和使用体验。MapView为地图显示，在RelativeLayout中加入相关控件。

4.2 主界面布局以及相关功能实现模块

主要控件包括：定位按钮，记录列表按钮，开始按钮，公里数、步数、时间及速度展示视图。



**图 8 MapActivity的结构**

4.2.1地图显示实现代码原理

MapActivity中声明：

private AMap aMap;  
private AMapLocationClient mLocationClient;  
private AMapLocationClientOption mLocationOption;  
private LocationSource.OnLocationChangedListener mListener;

AMap对象为高德SDK中的地图对象，AMapLocationClient用来管理地图的相关属性，通过AMapLocationClientOption自定义定位模式，定位时间间隔等设置。

private void initAMapLocation() {  
 //初始化定位  
 mLocationClient = new AMapLocationClient(this);  
 //设置定位回调监听  
 mLocationClient.setLocationListener(mAMapLocationListener);  
 //初始化AMapLocationClientOption对象  
 mLocationOption = new AMapLocationClientOption();  
 // 设置定位场景，目前支持三种场景（签到、出行、运动，默认无场景）

mLocationOption.setLocationPurpose(AMapLocationClientOption.AMapLocationPurpose.*Transport*);

mLocationOption.setLocationMode(AMapLocationClientOption.AMapLocationMode.*Hight\_Accuracy*);  
 //设置定位间隔,单位毫秒,默认为2000ms，最低1000ms。  
 mLocationOption.setInterval(2000);  
 //设置是否返回地址信息（默认返回地址信息）  
 mLocationOption.setNeedAddress(true);  
 mLocationOption.setHttpTimeOut(20000);  
 if (null != mLocationClient) {  
 mLocationClient.setLocationOption(mLocationOption);  
 mLocationClient.startLocation();  
 }  
}

通过mLocationClient.setLocationListener(mAMapLocationListener); 设置对定位的监听，mAMapLocationListener 实现了onLocationChanged方法，

public AMapLocationListener mAMapLocationListener = new AMapLocationListener() {  
 @Override  
 public void onLocationChanged(AMapLocation amapLocation) {  
  
 if (amapLocation != null) {  
  
 if (amapLocation.getErrorCode() == AMapLocation.*LOCATION\_SUCCESS*) {  
 mListener.onLocationChanged(amapLocation);

// 更新当前的位置  
 LatLng lan = new LatLng(amapLocation.getLatitude(), amapLocation.getLongitude());  
 if (currentLatLng != null) {  
 double dis = AMapUtils.*calculateLineDistance*(lan, currentLatLng);  
 distance += dis;  
 speedText.setText(String.*format*("%4.2f", amapLocation.getSpeed()));  
 distanceText.setText(String.*format*("%4.2f", distance / 1000));  
 }  
 currentLatLng = lan;  
  
 if (isRunning) {  
 mPath.addpoint(amapLocation);  
 mPolyoptions.add(currentLatLng);  
 drawLines();  
 }  
  
 }  
  
 } else {  
 Log.*e*("AmapError", "location Error, ErrCode:"  
 + amapLocation.getErrorCode() + ", errInfo:" + amapLocation.getErrorInfo());  
 }  
 }  
  
};

currentLatLng为当前的位置（经纬度）

当定位成功，获得的状态码为SUCCESS时，将currentLatLng更新为新定位到的经纬度。

同时计算上次定位点与本次定位点的距离，更新总距离，将新的距离和当前的速度进行更新。

通过成员变量isRunning判断是否在进行记录，是则向mPath中添加新的点，最终mPath中将保存从开始到结束所有的定位点。

4.2.2 跑步路径绘制实现代码原理

drawlines方法进行路径的绘制

private void drawLines() {  
 if (mPolyoptions.getPoints().size() > 1) {  
 if (mPolyline != null) {  
 mPolyline.setPoints(mPolyoptions.getPoints());  
 } else {  
 mPolyline = aMap.addPolyline(mPolyoptions);  
 }  
 }  
}

mPolyoptions中保存着每次定位到的点，当定位的点超过一个时才能进行线的绘制，路径的绘制实质上是多个线段的绘制，在每次新的定位点与上一个定位点绘制一条线段，就形成了行进的路径。

4.2.3 相关控件说明代码

**1.主界面逻辑设计**

通过点击屏幕下方的开始按钮进行记录运动数据，点击开始之后按钮变为结束按钮。

由两个Button实现。首先为startButton和stopButton设置监听，这里通过让MapActivity实现OnClickListener，直接将Button的监听器都绑定为this。

@Override  
public void onClick(View v) {  
 switch (v.getId()) {  
 case R.id.*button\_locate*:  
 Log.*d*(*TAG*, "button\_locate..");  
 location();  
 break;  
  
 case R.id.*button\_list*:  
 Intent intent = new Intent(MapActivity.this, RecordActivity.class);  
 startActivity(intent);  
 break;  
  
 case R.id.*button\_start*:  
 start();  
 break;  
  
 case R.id.*button\_stop*:  
 stop();  
 break;  
 }  
}

在定位按钮被点击时，调用location方法，location方法调用animateCamera方法实现相机的平滑移动，将相机移动到当前的位置。

如果currentLatLng为空，说明onLocaitonChange方法中amapLocation为null或者定位的状态不为SUCCESS，说明定位未成功，此时进行提示。

当list按钮被点击时，跳转到RecordActivity，通过显式Intent实现。

private void location() {  
 if (currentLatLng == null) {  
 Toast.*makeText*(getApplicationContext(), "正在搜索位置...", Toast.*LENGTH\_SHORT*).show();  
 } else {  
 aMap.animateCamera(CameraUpdateFactory.*newCameraPosition*(new CameraPosition(currentLatLng, 250, 0, 0)), 500, null);  
 }  
}

**2.开始按钮代码原理**

Start方法：

首先将上一次的记录清空，设置距离、时间为0。启动计步服务。

设置开始时间，同时隐藏startButton，显示stopButton，使用Timer+TimerTask来完成计时任务。

private void start() {  
 if (mPath != null) {  
 mPath = null;  
 }  
 aMap.clear(true);  
 mPath = new PathRecord();  
 distance = 0;  
 isRunning = true;  
 DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.*CHINA*);  
 mPath.setDate(df.format(new Date()));  
 Toast.*makeText*(getApplicationContext(), "开始时间:" + mPath.getDate(), Toast.*LENGTH\_SHORT*).show();  
  
 Intent intent = new Intent(MapActivity.this, BindService.class);  
  
 // 开始计步  
 isBind = bindService(intent, serviceConnection, Context.*BIND\_AUTO\_CREATE*);  
 Log.*d*("isbind",""+isBind);  
 startService(intent);  
  
 startTime = System.*currentTimeMillis*();  
 Log.*d*(*TAG*, "button\_start..");  
 if (startButton.getVisibility() == View.*VISIBLE*) {  
 startButton.setVisibility(View.*INVISIBLE*);  
 stopButton.setVisibility(View.*VISIBLE*);  
 }  
 speedText.setText("0.00");  
 distanceText.setText("0.00");  
 // 计时  
 if (mTimer == null) {  
 mTimer = new Timer();  
 }  
 if (mTimerTask == null) {  
 mTimerTask = new TimerTask() {  
 @Override  
 public void run() {  
 Message message = new Message();  
 message.what = 0;  
 mHandler.sendMessage(message);  
 }  
 };  
 }  
 mTimer.schedule(mTimerTask, 0, 1000);  
  
}

在TimerTask中使用message向Handler传递消息，通过Handler来改变时间的显示。

timeText.setText(TimeUtil.*getFormateTime*(System.*currentTimeMillis*() - startTime));

其中TimeUtil中的getFormatTime将时间的毫秒数转为00：00：00的格式化的字符串。

**3.停止按钮代码原理**

Stop方法：

stop按钮被点击，隐藏stop按钮，显示start按钮，设置endTime，解除计步服务的绑定（停止计步），保存记录的PathRecord对象到数据库中，接着将定时任务取消。

private void stop() {  
 if (stopButton.getVisibility() == View.*VISIBLE*) {  
 stopButton.setVisibility(View.*INVISIBLE*);  
 startButton.setVisibility(View.*VISIBLE*);  
 }  
 endTime = System.*currentTimeMillis*();  
 isRunning = false;  
 save();  
 unbindService(serviceConnection);  
 distance = 0;  
 Log.*d*(*TAG*, "button\_stop");  
 if (mTimer != null) {  
 mTimer.cancel();  
 mTimer = null;  
 }  
 if (mTimerTask != null) {  
 mTimerTask.cancel();  
 mTimerTask = null;  
 }  
 // new  
// aMap.clear();  
  
 }

**4.跑步记录保存代码原理**

Save方法：

private void save() {  
 List<AMapLocation> list = mPath.getPathline();  
 if (list != null && list.size() > 0) {  
 mDatabaseHelper = new DBHelper(this);  
 mDatabaseHelper.open();  
 String duration = getDuration();  
 double distance = getDistance();  
 String average = getAverage(distance);  
 String pathlineSring = LocUtil.*getPathLineString*(list);  
 AMapLocation firstLocaiton = list.get(0);  
 AMapLocation lastLocaiton = list.get(list.size() - 1);  
 String stratpoint = LocUtil.*amapLocationToString*(firstLocaiton);  
 String endpoint = LocUtil.*amapLocationToString*(lastLocaiton);  
 mDatabaseHelper.createrecord(String.*valueOf*(distance), duration, average,  
 pathlineSring, stratpoint, endpoint, mPath.getDate());  
 mDatabaseHelper.close();  
 Toast.*makeText*(getApplicationContext(), "保存记录成功!", Toast.*LENGTH\_SHORT*).show();  
 } else {  
 Toast.*makeText*(MapActivity.this, "距离过短..", Toast.*LENGTH\_SHORT*).show();  
 }  
}

这里通过PathRecord对象获取定位点的列表，使用DBHelper类进行数据库的访问，将记录的时间、距离、速度开始结束点、定位点的列表保存到SQLite数据库中，如果定位失败或者只有一个点，则不进行保存。

## PathRecord类

// 起点  
private AMapLocation mStartPoint;  
// 终点  
private AMapLocation mEndPoint;  
// 保存所有的点  
private List<AMapLocation> mPathLinePoints = new ArrayList<>();  
// 总距离  
private String mDistance;  
// 持续时间  
private String mDuration;  
// 平均速度  
private String mAverageSpeed;  
// 开始时间  
private String mDate;  
  
// id，主键  
private int mId = 0;

PathRecord类中保存着记录的信息，轨迹的所有点的列表，持续时间，开始日期，距离等。

对外提供getter和setter。

同时重写了toString方法，以便进行展示。

**DBHelper类**

DBHelper类负责处理访问数据库的事件，通过静态Sting常量保存字段名及建表语句。

各个字段对应PathRecord类的各个成员。因为PathRecord类中的速度、时间等都以String类型存储，故建表时对应的字段也是String类型。

public static class DatabaseHelper extends SQLiteOpenHelper {  
 public DatabaseHelper(Context context) {  
 super(context, *DATABASE\_NAME*, null, *DATABASE\_VERSION*);  
 }  
  
 @Override  
 public void onCreate(SQLiteDatabase db) {  
 db.execSQL(*RECORD\_CREATE*);  
 }  
  
 @Override  
 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
 }  
}

通过静态内部类DatabaseHelper继承SQLiteOpenHelper，在重写的onCreate方法中执行建表语句，

create table if not exists record

.....

这样将在没有record表时创建它。

通过DatabaseHelper的对象来进行SQLite数据库的增删改查操作。

public long createrecord(String distance, String duration,  
 String averagespeed, String pathline, String stratpoint,  
 String endpoint, String date) {  
 ContentValues args = new ContentValues();  
 args.put("distance", distance);  
 args.put("duration", duration);  
 args.put("averagespeed", averagespeed);  
 args.put("pathline", pathline);  
 args.put("stratpoint", stratpoint);  
 args.put("endpoint", endpoint);  
 args.put("date", date);  
 return db.insert(*RECORD\_TABLE*, null, args);  
}

向record表中加入一条新的数据。

public List<PathRecord> queryRecordAll() {  
 List<PathRecord> allRecord = new ArrayList<>();  
 Cursor allRecordCursor = db.query(*RECORD\_TABLE*, getColumns(), null,  
 null, null, null, null);  
 while (allRecordCursor.moveToNext()) {  
 PathRecord record = new PathRecord();  
 record.setId(allRecordCursor.getInt(allRecordCursor  
 .getColumnIndex(DBHelper.*KEY\_ROWID*)));  
 record.setDistance(allRecordCursor.getString(allRecordCursor  
 .getColumnIndex(DBHelper.*KEY\_DISTANCE*)));  
 record.setDuration(allRecordCursor.getString(allRecordCursor  
 .getColumnIndex(DBHelper.*KEY\_DURATION*)));  
 record.setDate(allRecordCursor.getString(allRecordCursor  
 .getColumnIndex(DBHelper.*KEY\_DATE*)));  
 String lines = allRecordCursor.getString(allRecordCursor  
 .getColumnIndex(DBHelper.*KEY\_LINE*));  
 record.setPathline(LocUtil.*parseLocations*(lines));  
 record.setStartpoint(LocUtil.*parseLocation*(allRecordCursor  
 .getString(allRecordCursor  
 .getColumnIndex(DBHelper.*KEY\_STRAT*))));  
 record.setEndpoint(LocUtil.*parseLocation*(allRecordCursor  
 .getString(allRecordCursor  
 .getColumnIndex(DBHelper.*KEY\_END*))));  
 allRecord.add(record);  
 }  
 Collections.*reverse*(allRecord);  
 return allRecord;  
}

通过查询数据库得到的Cursor对象对结果集进行遍历，以一条结果的各字段的值来构建给一个PathRecord对象，遍历所有结果完成得到一个List，将List翻转，得到按时间逆序排列的记录列表。

类似的，通过id查询就是通过id调用query方法，只不过要加上约束条件where id = recordId

## TraceRePlay类

private TraceRePlayHandler mTraceHandler;  
private static final int *TRACE\_MOVE* = 1;  
private static final int *TRACE\_FINISH* = 2;  
private List<LatLng> mTraceList;  
private int mIntervalMillisecond;  
private TraceRePlayListener mTraceUpdateListener;  
private boolean mStop = false;

TraceRePlay类中的成员变量如图

TraceRePalyHandler继承Handler，这里将TraceRePalyHandler声明为静态内部类，静态内部类不能访问外部类的非静态成员，在内部类中定义外部类的弱引用来进行消息的响应动作。

TraceRePlay类实现了Runnable接口，在run方法中进行轨迹的绘制，即遍历list中的所有点，向handler发送消息，进行轨迹的绘制。具体的绘制由回调接口实现者进行。

TaceRePlayLisenter是一个回调的接口，用来处理轨迹结束和轨迹在重放过程中的事件。

在android中有很多方法的回调使用，比如activity和adapter，listener之间的回调。

回调机制就是将一个方法的定义和其具体的实现分开，做什么由使用者通过实现回调接口来决定。

回调机制可以通过定义内部的接口或者公共的接口实现，在构造器中加入接口的引用作为参数，或者提供一个setListener之类的方法来进行回调的注册，在类的内部通过mListener=lisenter进行赋值，这样就可以使用回调的接口，而不必关心具体的功能实现。TraceRePlay类中的回调接口就定义了轨迹的更新和结束轨迹的方法，而将具体的实现交给调用TraceRePlay。

static class TraceRePlayHandler extends Handler {  
 WeakReference<TraceRePlay> mTraceRePaly;  
  
 public TraceRePlayHandler(TraceRePlay traceRePlay) {  
 super(Looper.*getMainLooper*());  
 mTraceRePaly = new WeakReference<>(traceRePlay);  
 }  
  
 @Override  
 public void handleMessage(Message message) {  
 super.handleMessage(message);  
 TraceRePlay trace = mTraceRePaly.get();  
 switch (message.what) {  
 case *TRACE\_MOVE*:  
 LatLng latLng = (LatLng) message.obj;  
 if (trace.mTraceUpdateListener != null) {  
 trace.mTraceUpdateListener.onTraceUpdating(latLng);  
 }  
 break;  
 case *TRACE\_FINISH*:  
 if (trace.mTraceUpdateListener != null) {  
 trace.mTraceUpdateListener.onTraceUpdateFinish();  
 }  
 break;  
 default:  
 break;  
 }  
 }  
}

## RecordActivity和RecordAdapter

RecordActivity使用listView显示本地数据库存储的所有记录。

在RecordAdapter中：

private Context mContext;  
private List<PathRecord> mRecordList;  
  
public RecordAdapter(Context context, List<PathRecord> list) {  
 this.mContext = context;  
 this.mRecordList = list;  
}

RecordAdapter类继承BaseAdapter，作为RecordActivity中的listView的适配器。通过mContext保存Context对象，mRecordList保存所有点。通过构造器进行初始化。

@Override  
public View getView(int position, View convertView, ViewGroup parent) {  
 ViewHolder holder;  
 if (convertView == null) {  
 holder = new ViewHolder();  
 convertView = View.*inflate*(mContext, R.layout.*record\_item*, null);  
 holder.date = convertView.findViewById(R.id.*date*);  
 holder.record = convertView.findViewById(R.id.*record*);  
  
 convertView.setTag(holder);  
 } else {  
 holder = (ViewHolder) convertView.getTag();  
 }  
  
 PathRecord item = mRecordList.get(position);  
 holder.date.setText(item.getDate());  
 holder.record.setText(item.toString());  
 return convertView;  
}

在getView方法中通过inflate根据布局文件的id加载成View，并返回，使用ViewHolder来保存View，当convertView不为空时，直接使用Holder中的View即可，加快listView的加载速度。

再回到RecordActivity，通过DBHelper类来查询出所有的记录，将其保存到列表中，用其构建RecordAdapter。

4.3 步数统计代码原理

从MainActivity进入，在构造函数中绷定并开启BindService服务，其中绷定是为了方便数据交换，启动是为了使activity不为活跃状态计步服务在后台也能一直运行。而BindService服务实现了SensorEventListener接口，SensorEventListener接口是计步传感器的一个回调接口。开启这个子线程后，子线程中开始计步。先通过startStepDetector()获取并判断SDK版本，SDK版本高于19的机型一般实现了用于记录用户行走步数的Sensor.TYPE\_STEP\_COUNTER传感器，所以用addCountStepListener()直接开启计步传感器计步，在计步传感器数据变化时，通过系统回调onSensorChanged(SensorEvent event)获取步数数据。而版本低于19的机型，我们通过加速度传感器的数据来计算步数。用addBasePedometer()启动加速度传感器计步，先为传感器注册监听器，获得加速度后调用StepCount进行计算处理，通过接口反馈回来步数。



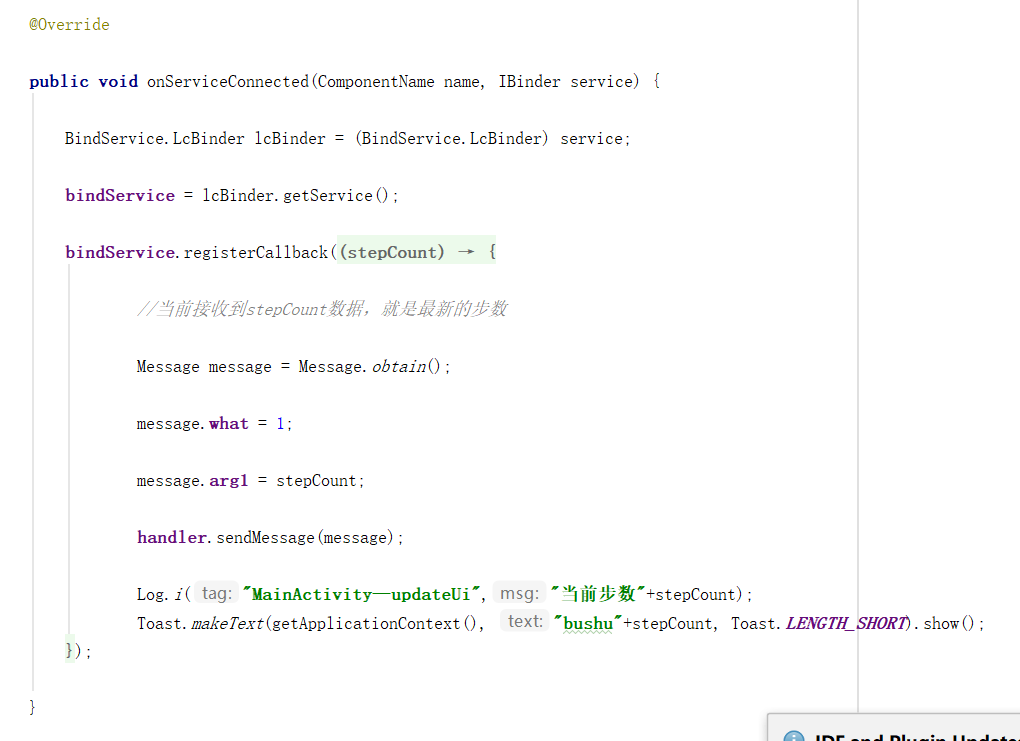
**图9 根据版本选择传感器**

StepCount通过计算加速度的波峰波谷来判断人是否走一步，即如果检测到了波峰，且波峰间时间差满足阈值的条件，则判定为1步。同时，若符合时间差条件，且波峰波谷差值大于initialValue，则将该差值纳入阈值的计算中。波峰由当前点和前一个点的下降上升趋势、持续上升次数以及最小阈值四个条件进行检测，波谷由图像特征进行检测。阈值由波峰波谷差值进行动态更新，并通过均值将其梯度化在范围内。另外，为提高app的实用性，规定运动在9步以内或9步以内停顿超过3s的情况视为无效运动。



图 10 通过加速度传感器数据计算步数

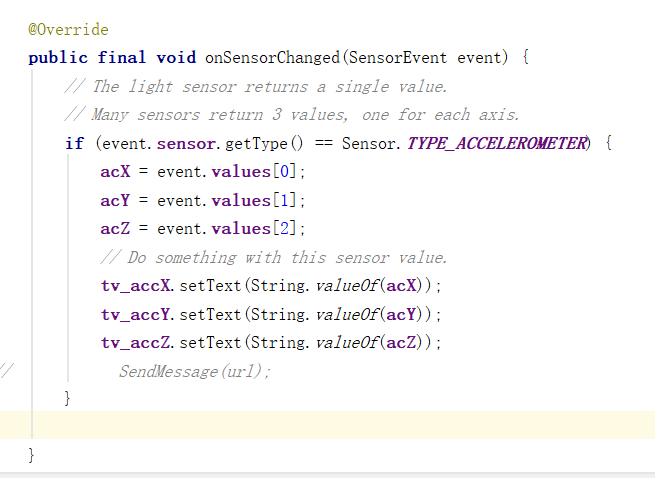
得到当前步数后调用updateNotification()发送更新步数通知，通过接口回调的方式将数据传递给调用者，实现与MainAcitivity的数据交互。而MainActivity通过IBinder service获取服务的实例，调用服务的方法或者数据来构造和绷定服务数据交换的桥梁，从onServiceConnected中接收到步数，通过消息和消息队列来更新UI，将步数stepCount显示出来。另外，在app关闭前，BindService需要解除绑定，否则下次Activity切换到活动界面的时候会重新开启一个新的计步线程。



**图 11 更新UI**

4.4 三轴加速传感器代码原理

先用getSystemService获取传感器服务，使用getDefaultSensor()方法从传感器服务中指定待监听的加速度传感器，再用registerListener()方法添加加速度传感器的监听器，即可回调得到传感器获取的三轴加速度的数据，存储到JSONObject类型的object中，为避免负荷太大，使用Android定时器Timer.schedule定时获取三轴数据。用完后清空Timer的任务队列并注销传感器的监听器。



**图 12 获取三轴加速度**



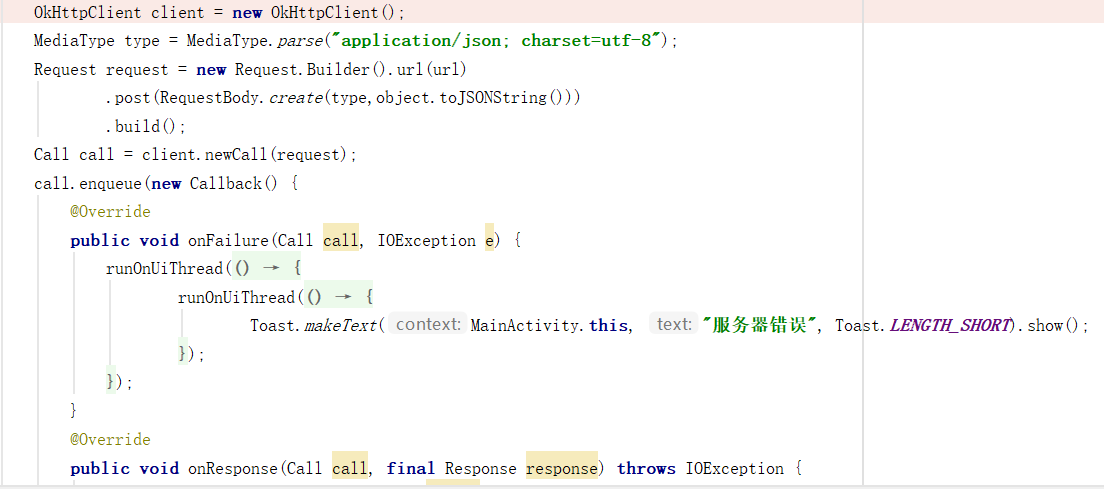
**图 13定时获取**

1. 前后端数据传送代码原理

5.1 前端发送请求，接收后端的响应

**OkHttp框架：**

先创建OkHttpClient客户端对象client，再调用newCall(request)创建真正发起请求的类call，实例化request对象，定好ulr请求链接、post请求方式和RequestBody请求体了，请求体中，为方便python的数据接收，使用toJSONString是将object转化为Json字符串类型。再调用OkHttpClient的newCall方法，把request对象传进去，然后执行enqueue异步方法请求。在CallBack的onResponse方法中对后端返回的数据进行判断，对应转化为坐着、站着、走着、跑着、上楼、下楼六种状态并实时更新在屏幕上。



**图 14 OKHttp实现前后端通信**

5.2 后端接收请求，发送响应给前端

**Django框架：**

Django是一个开放源代码的Web应用框架，由Python写成,采用了MT'V的框架模式.即Model,View,Template组成.许多成功的网站和APP都基于Django.说到底,其实Django内部就是对 Socket 连接的强大封装.Django框架可用于前后端通信，接收前端的请求，返回一个响应。

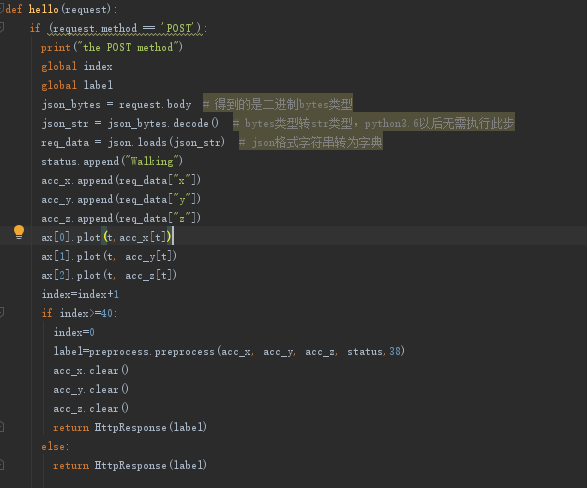
步骤：

1.首先判断请求方式是否为post，post请求包含两部分，请求头（header）和请求体（body）。

2.我们得到的前端数据为json类型的，因此我们需要首先得到request.body，我们在经过转化之后，将其转化为字典格式

3.获取x轴加速度，y轴加速度，z轴加速度

4.返回响应，一个int型的label值

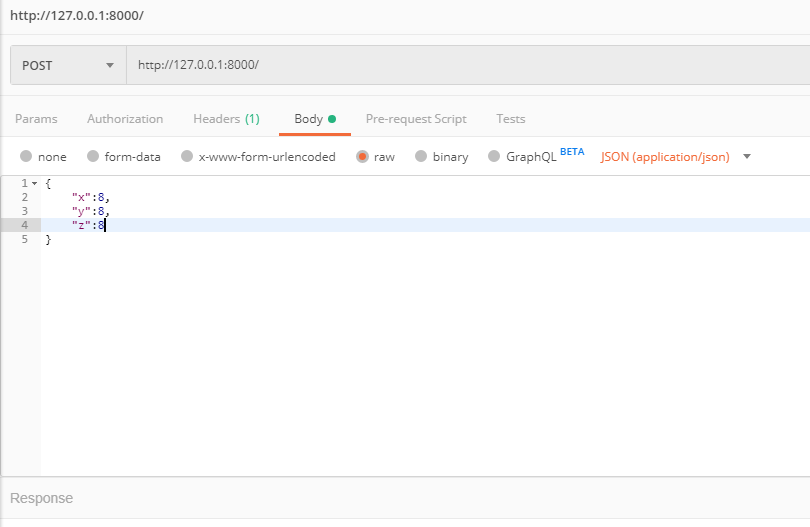


**图 15 Django框架**

传送的url值为：主机号+端口号（Django默认端口号为8000）

5.3 postman 检验端口

Postman用于检验前后端的数据传送，请求格式设为post，在body中将数据格式转化为json格式，若正常传送，应该返回label值



**图 16 postman检验**

1. 后端步态识别代码原理
   1. 模型选择

SVM算法：

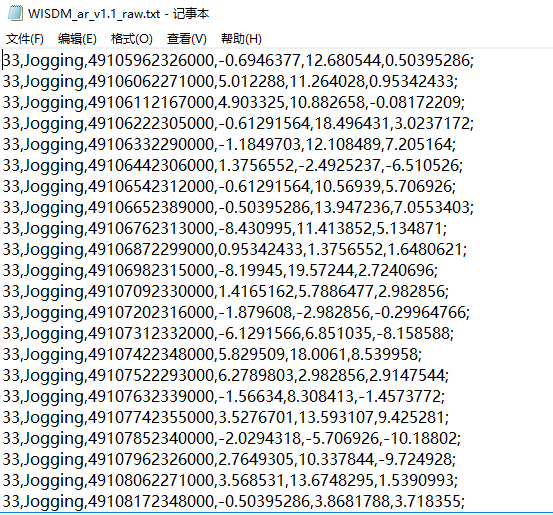
一个能够将不同类样本在样本空间分隔的超平面。换句话说，给定一些标记好的训练本（监督式学习），SVM算法输出一个最优化的超分隔平面。SVM为典型的用于多分类的预测模型，而在一些经典的分类算法中，比如：随机森林，kNN算法中，SVM算法对于本题目表现更好，而Lib SVM是台湾大学林智仁(Lin Chih-Jen)教授等开发设计的一个简单、易于使用和快速有效的SVM模式识别与回归的软件包，他不但提供了编译好的可在Windows系列系统的执行文件，还提供了源代码，方便改进、修改以及在其它操作系统上应用；该软件对SVM所涉及的参数调节相对比较少，提供了很多的默认参数，利用这些默认参数可以解决很多问题,因此为了简便有效起见，我们选择了LIB SVM模型。

* 1. 数据处理

6.2.1初始数据集

WISDM\_ar\_latest.tar.gz 数据集

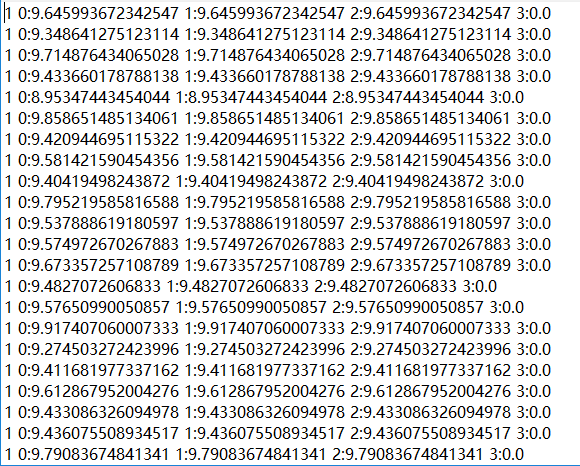
（每列数据分别代表了年龄，状态，编号，x轴加速度，y轴加速度，z轴加速度）



**图 17 初始数据集**

6.2.2数据预处理

将原始数据集转化为模型接受的格式



**图 18 数据预处理**

6.2.3 特征提取及模型训练

选择了一个周期内的平均值，最大值，最小值，标准差这四个特征进行训练，通过SVM自带的划分训练集和测试集的方法，将数据集进行随机划分，之后将训练集带入svm-train进行训练，svm-train函数中会自动打印经过K折交叉验证之后的准确率，得到模型后通过svm-save进行模型保存，在模型优化调参的时候调用grid.py是一个封装好的调参文件，最后通过svm-predict进行预测。

* 1. 模型预测

我们将模型返回来的结果对应转化为相应的表情包，增强了本App的趣味性，同时，我们通过测试发现我们训练得到的模型表现良好，能够给用户提醒，特别是学生，长时间坐着不运动，或者是一直在玩不学习等，有较好的监督提醒功能，可以考虑放入后台运行。



**图19步态识别**

1. 遇到的问题及解决

1.在activity之间，activity和Service之间传递数据主要通过Intent进行，其中Intent分为显式和隐式，显式Intent指定目标的类名，隐式Intent则不直接指定，通过Action的方式来判断，所有在intent-filter中具有对应的action的activity或者service都可以响应。

Activity内部传递消息需要注意，不能在非UI线程（主线程）修改UI，如果必须这么做，可以通过handler的方式进行，通过定义自己的Handler，实现handlMessage方法，根据what值来分辨不同的消息，message对象还可以传递数据，message.obj是一个Object类型的对象，可以将需要传递的对象赋值给obj，在handler中进行类型转换即可。

2.在前后端传送数据的过程中，必须处于同一个局域网下，否则会出现服务器错误等情况，同时url的设置为主机号+端口号

3.一开始想使用flask框架来进行前后端通信，实现了后端给网页发送信息，但是不知道如何和手机端通信，所以换用Django框架，前端使用OkHttp框架，实现了前后端交互。

4.使用onTick方法倒计时秒数实现定时执行一次方法，但调用失败了，所以直接创建一个定时器(Timer类对象),并通过Timer.schedule(TimerTask task,Date time) 方法定时发送数据。

8.总结感悟

通过本次项目的实践，我们的动手操作能力得到了增强，不管是前端Android Studio 还是后端的python ，以前我们只是单纯的写前端或者后端，这是第一次尝试前端后端通信，在做的过程中遇到了很多bug，但是我们一起讨论解决了这些问题，我们更加明白了小组团结协作的重要性，每个人有明确的分工，遇到困难，大家一起绞尽脑汁地解决，高效率地完成了此次项目，非常感谢本次课程。