基般

2 Android によるセンサプログラミング

石丸宗平
(株) ナノコネクト

Android とセンサ

■ 容易に使えるセンサ

Android では，搭載されたセンサを容易に取得するクラスライブラリが提供されています．センサを用いることで，Android 端末がどのような環境下で動作しているのかを知るための有用な手段となり，環境に対して何らかのレスポンスを返すアプリケーションを作成することができます．

■ 多彩なセンサ

標準的な Android のクラスライブラリは，表 -1 の定数が Sensor クラスで宣言されており，Android アプリケーションから容易にセンサを扱えるようになっています．これらのほかにも，ベンダが独自に搭載したセンサが，クラスライブラリとして提供されている場合もあります．

クラスライブラリ上では，iPhone 4 が搭載しているセンサの数より多くのセンサが定

定数名	センサ名称	備考
TYPE_ALL	すべてのセンサ	
TYPE_ACCELEROMETER	加速度センサ	
TYPE_AMBIENT_TEMPERATURE	温度センサ（室温）	Android4.0（API Level14 に相当） 新規追加
TYPE_GRABITY	重力センサ	Android2.3（API Level9 に相当） 新規追加
TYPE_GYROSCOPE	ジャイロセンサ	
TYPE_LIGHT	照度センサ	
TYPE_LINEAR_ACCELERATION	直線加速度センサ	Android2.3（API Level9 に相当） 新規追加
TYPE_MAGNETIC_FIELD	地磁気センサ	
TYPE_ORIENTATION	傾斜センサ	Android2.2（API Level8 に相当） 非推奨
TYPE_PRESSURE	圧力センサ	
TYPE_PROXIMITY	近接センサ	
TYPE_RELATIVE_HUMIDITY	湿度センサ	Android4.0（API Level14 に相当） 新規追加
TYPE_ROTATION_VECTOR	回転ベクトルセンサ	Android2.3（API Level9 に相当） 新規追加
TYPE_TEMPERATURE	温度センサ	Android4.0（API Level14 に相当） 非推奨

表 -1　センサクラスで定義されたセンサの定数

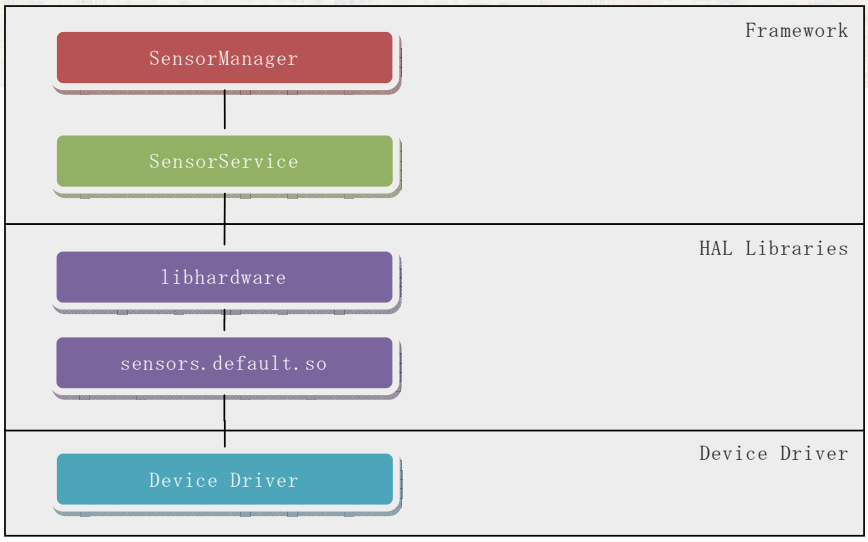

図 -1　センサに関するソフトウェアの構造

義されていますが，Android 端末にすべてのセンサが搭載されているとは限りません．センサを用いたアプリケーションを作成する際には，センサの存否に注意して実装する必要があります．また，Android プラットフォームはバージョンアップを重ねるごとに，非推奨のクラスやメソッドが増えており，比較的古い書籍や Web サイトでは，非推奨のものがそのまま使われているサンプルコードがありますので，新規にアプリケーションを作成する際には，非推奨のクラスやメソッドの使用を避けることを推奨します．

■ センサに関するソフトウェアの構成

Android では，Android アプリケーションから容易にセンサを利用できるようにクラスライブラリが提供されています．そのクラスライブラリは，Java で作成されており，Java Native Interface（以下 JNI）を用いて，C / C++ などで書かれたネイティブコードのライブラリを呼び出しています．Android のクラスライブラリのソースコードを読むと，随所で native という修飾子があり，ここで JNI を用いてネイティブライブラリを呼び出しています☆1．

クラスライブラリに含まれる SensorManager クラスでは，JNI で「android_hardware_SensorManager.cpp」に含まれる関数を呼び出し，SensorService と通信を行います．その SensorService から，「libhardware」，「sensors.default.so☆2」のネイティブで書かれたライブラリが呼び出され，sensor.default.so の中でデバイスドライバへアクセスし，センサ値の読み込みや，センサ情報の取得などの処理が行われています（図 -1）．

評価ボードなどを用いて，新たにセンサを搭載するには，センサのデバイスドライバと「sensors.default.so」に相当するライブラリを新たに作成する必要があります． そのライブラリを作成するには，「hardware/sensors.h」に用意されたヘッダファイルを include し，ヘッダファイルで定義された関数を用意する必要があります．そして作成したライブラリ

☆1　Android Open Source Project のサイトから，閲覧することができます．http://source.android.com/
☆2　「sensors.default.so」の default には，機種固有の名称が入ります．

を，「system/lib」ディレクトリ以下に配置すると，SensorManager に認識され，Android アプリケーションから利用することができるようになります．

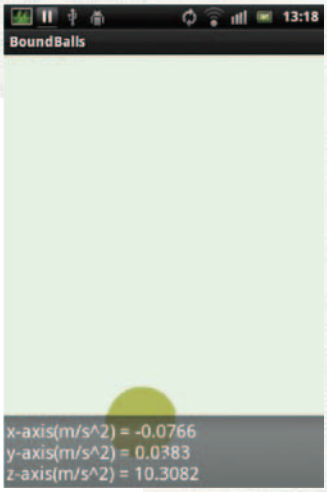

図 -2　アプリケーションのスクリーンショット

センサチュートリアル

ここからは，センサを使用したアプリケーションのソースコードを作成しながら，どのようにセンサを利用するかを解説していきます．本章で使用しているソースコードおよびリソースファイルなどのプロジェクトに関するファイルは，以下の Web サイトで配布しています．ファイルをダウンロードし，参考にしてください．

http://www.ipsj.or.jp/magazine/smartphone.html

■ このアプリケーションについて

このアプリケーションは，加速度センサの値を利用し，端末の傾きに応じてボールを動かすことができるアプリケーションです（図 -2）．このアプリケーションでは，加速度センサを使用しますので，エミュレータや，加速度センサを搭載していない端末では，ボールを動かすことができません．

- ターゲットは Android 2.1（API Level7 に相当）以上としています．
- センサから取得した端末の X 軸，Y 軸の加速度に応じて，それぞれのボールが動作します．
- ボールが画面の端に衝突した場合は跳ね返る処理を実装し，画面の外へ移動しないようにします．
- 画面の下側に，センサから入力された加速度の値を表示します．

■ プロジェクトの新規作成

まず，Android アプリケーションの開発環境をインストールしたパソコン上で，新規プロジェクトの作成を行います．Eclipse を起動し，以下の値を入力してプロジェクトを新規に作成します．そうすると，雛形からプロジェクトが新規に生成されます．作成された BoundBallActivity には，センサに関する処理を実装していきます．

項目名	値
Project Name	BoundBall
Build Target	Android 2.1 (API Level7)
Application name	Bound Ball
Package name	jp.co.nanoconnect.boundBall
Create Activity	BoundBallActivity
Min SDK Version	7

また，このアプリケーションで利用する画像ファイルは，ダウンロードしたファイルの中にあらかじめ用意してありますので，以下のファイルを「Package Explorer」-「BoundBall」-「res」-「drawable-hdpi」内にコピーしておきます．

- icon.png
- label_background.9.png

（ダウンロードファイル snapshot_1 を参照）

■ クラスの新規作成

画面上でボールの描画を行う BoundBallView クラスは，SurfaceView クラスを継承して作成を行います．SurfaceView は，メインスレッドから独立したスレッドで描画を行うことができる View で，画面の再描画が頻繁に必要なアプリケーションを作成する際に用いられます．BoundBallView クラスは，BoundBallActivity クラスから受け取ったセンサの値を元に，ボールの移動，描画，衝突の判定を行うクラスとなります．

BoundBallView クラスの作成は，Eclipse の「Package Explorer」より，「BoundBall」プロジェクトの「src」-「jp.co.nanoconnect.boundball」を右クリックし，「Class」を選択します．表示されたダイアログに以下の値を入力し，「Constructors from superclass」にチェックを入れて「Finish」を押すと，SurfaceView クラスを継承した BoundBallView クラスが作成されます．

項目名	値
Name	BoundBallView
Superclass	android.view.SurfaceView

（ダウンロードファイル snapshot_2 を参照）

■ レイアウトファイルの作成

プロジェクトを新規作成した際に自動生成される「main.xml」ファイルの編集を行い，画面上に BoundBallView を表示するように修正します．まずは，Eclipse の「Package Explorer」から，「BoundBall」-「res」-「layout」-「main.xml」をダブルクリックし，編集画面を開きます．自動生成された状態では，「Hello World, BoundBallActivity!」を表示する「TextView」のみ配置されたレイアウトとなっていますので，その文字列を選択し，「右クリック」-「Delete」操作で削除します．

次に，編集画面の「Palette」-「Custom & Library Views」より「BoundBallView」をドラッグ＆ドロップします．レイアウトの幅，高さは，中の要素に合わせる「Wrap Content」に指定されていますので，編集画面の「BoundBallView」を選択し，「右クリック」-「Layout Width」を「Fill Parent」に指定します．同じように，「Layout Height」も「Fill Parent」に指定すると，画面全体に「BoundBallView」が配置されます（**図 -3参照**）．

そして，センサの値を表示する TextView を配置します．この TextView は，BoundBallView の上に重ねて表示しますので，View を縦か横かに一直線で並べる

図 -3　main.xml の編集画面 1

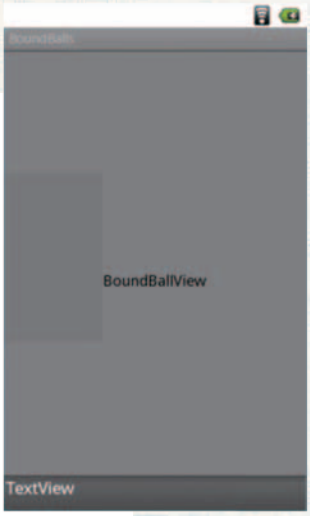

図 -4　main.xml の編集画面 2

LinearLayout では配置することができません．こういった場合には，RelativeLayout と呼ばれるレイアウトを用いて配置を行います．RelativeLayout は，親の ViewGroup や，特定の View に対して相対的に位置を指定するレイアウトで，View を重ねたり，変則的な配置のレイアウトを作成したりすることができます．Eclipse の「Outline」-「最上位の LinearLayout」を選択し，「右クリック」-「Change Layout」から，「New Layout Type」で「Relative Layout」を選択し，「OK」を押すことでレイアウトの種類を変更することができます．そして編集画面の「Palette」-「Form Widgets」から，「MediumText」を BoundBallView の左下寄りにドラッグ & ドロップします．すると，「TextView」のプロパティである「Layout align parent bottom」と「Layout align parent left」に「true」が設定され，親の View である「RelativeLayout」に対して左下に配置するという指定方法となります．

また，レイアウトを整えるため，「TextView」の「Layout Width」を「Fill Parent」に指定し，幅を画面一杯まで伸ばし，「右クリック」-「Properties」-「Background」から，「Drawable」の「label_background」を選択して半透明の背景画像を適用します（**図 -4** 参照）．

最後に，Java のソースコード上から，それぞれの View に対して，表示する文字列やセンサ値の受け渡しを行いますので，以下のように判別しやすい ID に変更しておきます．ID を変更するには，変更を行う View を選択し，「右クリック」-「Edit Id」から変更を行ってください．この状態でアプリケーションを実行すると，レイアウトファイルが変更され，何も描画されていない BoundBallView が表示されている状態となります．

初期 ID	変更後の ID
boundBallView1	view_bound_ball
textView1	text_values

（ダウンロードファイル snapshot_3 を参照）

■ 描画処理の実装

● SurfaceHolder.CallBack の実装

「BoundBallView」にて，描画処理を行うには，「SurfaceHolder.Callback」インタフェースを実装して，サーフェースが生成，変更，破棄されるタイミングでコールバックを受け取れるようにします．「BoundBallView」クラスを宣言している行で，「implements SurfaceHolder.Callback」を追加すると，「SurfaceHolder」の import 宣言がありませんので「SurfaceHolder cannot be resolved to a type」というエラーが発生します．「Quick fixes」よ

り，「Import 'SurfaceHolder'(android.view)」を選択することで import 宣言を追加することができます．次に「BoundBallView」クラスの宣言個所で，インタフェースに定義されたメソッドが用意されていない「The type BoundBallView must implement the inherited abstract method. . .」というエラーが発生しますので，「Quick fixes」より「Add unimplemented methods」を選択し，インタフェースで定義されたメソッドの自動生成を行います．

```
public class BoundBallView extends SurfaceView implements SurfaceHolder.Callback {
    public BoundBallView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width,
            int height) {
    }
    @Override
    public void surfaceCreated(SurfaceHolder holder) {
    }
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
    }
}
```

SurfaceHolder.Callback を実装しただけでは，コールバックされませんので，getHolder メソッドで SurfaceHolder インスタンスを取得し，addCallback メソッドでコールバック先の指定を行います．この処理は，BoundBallView クラスの，Context と AttributeSet を引数とするコンストラクタで処理を実装します．他のコンストラクタは，今回のアプリケーションでは使用しませんので削除しておきます．

```
public BoundBallView(Context context, AttributeSet attrs) {
    super(context, attrs);
    SurfaceHolder holder = getHolder();
    holder.addCallback(this);
}
```

（ダウンロードファイル snapshot_4 を参照）

● 描画スレッドの作成

ここでは，アプリケーションが終了するまで繰り返し描画を行うためのスレッドの作成を行います．クラスフィールドに，mDrawThread を宣言します．また，スレッドのメインループを制御するフラグも作成しておきます．

```
/** 描画スレッド */
private Thread mDrawThread;
/** スレッドのメインループ制御フラグ */
private boolean mRun;
```

宣言した mDrawThread インスタンスは，サーフェースが変更されたときに呼ばれる surfaceChanged メソッドで初期化を行います．そして，コンストラクタでは，Runnable を実装した無名クラスを指定し，この run メソッドに，メインループを作成し，ボールの移動処理，衝突処理，描画処理を実装していきます．そしてスレッドの生成後に，mRun に true を代入してスレッドを開始させます．

```
@Override
public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
    mRun = true;
    mDrawThread = new Thread(new Runnable() {
        @Override
        public void run() {
            while (mRun) {
                /* ループ処理 */
            }
        }
    });
    mDrawThread.start();
}
```

開始させたスレッドは，アプリケーションが終了するときに停止させなければなりません．スレッドの停止は，surfaceDestroyed メソッドで mRun に false を代入し，停止を行います．その際には，join メソッドを用いて，スレッドが終了するのを待ちます．

```
@Override
public void surfaceDestroyed(SurfaceHolder holder) {
    boolean retry = true;
    mRun = false;
    while (retry) {
        try {
            mDrawThread.join();
            retry = false;
        } catch (InterruptedException e) {
            /* NOP */
        }
    }
}
```

（ダウンロードファイル snapshot_5 を参照）

● Ball クラスの作成

次に，ボールの位置や速度の値を保持する Ball クラスの作成を行います．今回のアプリケーションでは，Ball クラスは，BoundBallView クラスのみで使用しますので，BoundBallView クラスのインナークラスとして定義します．この Ball クラスには，ボールの中心座標 (X，Y)，速度 (X，Y)，半径，描画のスタイルをクラスフィールドに宣言し，他のインスタンスから，値へ直接アクセスするようにしています．

```
private class Ball {
    /** 中心座標 X */
    public float mPx;
    /** 中心座標 Y */
    public float mPy;
    /** 速度 X */
    public float mDx;
    /** 速度 Y */
    public float mDy;
    /** 半径 */
    public float mRadius;
    /** 描画のスタイル */
    public Paint mPaint;

    public Ball(float x, float y, float r, int color) {
        this.mPx = x;
        this.mPy = y;
        this.mRadius = r;
        mPaint = new Paint();
        mPaint.setColor(color);
        mPaint.setAntiAlias(true);
    }
}
```

（ダウンロードファイル snapshot_6 を参照）

● Ball インスタンスの生成と描画処理

まず始めに，BoundBallView クラスの，surfaceChanged メソッドが呼ばれたタイミングで，Ball クラスのインスタンス "mBall" を初期化します．初期の位置は，画面の中央で，半径，色は定数で定義を行っています．そのほか，サーフェースの高さや幅，背景の描画スタイルなどをクラスフィールドで定義しておきます．

次に描画スレッド内のメインループ内に描画処理を実装します．その際に，描画を行う Canvas と呼ばれるインスタンスに対して，lockCanvas メソッドでロックを掛け，ほかから描画されないように排他処理を行います．そして描画が完了したタイミングで取得したロックは，unlockCanvasAndPost メソッドでアンロックします．描画の処理は，doDraw メソッドを定義し，そのメソッド内で描画を行います．描画は，Canvas クラスに用意された drawPaint メソッドで画面全体の塗りつぶし，drawCircle メソッドで円の中心座標，半径，描画スタイルを指定して描画を行っています．この段階でアプリケーションを実行すると，画面上に背景と，中央にボールが描かれるようになります．

```
mDrawThread = new Thread(new Runnable() {
    @Override
    public void run() {
        Canvas canvas = null;
        SurfaceHolder holder = getHolder();
        while (mRun) {
            try {
                canvas = holder.lockCanvas();
                if (canvas != null) {
                    doDraw(canvas);
                }
            } finally {
                if (canvas != null) {
                    holder.unlockCanvasAndPost(canvas);
                }
            }
        }
    }
});
```

```
private void doDraw(Canvas canvas) {
    canvas.drawPaint(mBgStyle);
    canvas.drawCircle(mBall.mPx, mBall.mPy, mBall.mRadius, mBall.mPaint);
}
```

（ダウンロードファイル snapshot_7 を参照）

■ 移動処理の実装

加速度センサから値を受け取り，ボールの移動処理を実装します．加速度センサ値を受け取るメソッドは，setAcceleration メソッドを用意し，クラスフィールドに値を持つようにします．ボールの移動処理は，描画スレッドのメインループ内で位置の更新を行う doUpdate メソッドを呼び出し，移動処理は，センサから入力された X 軸，Y 軸方向の加速度から，毎フレーム移動量を計算し，移動処理を行います．また，画面の範囲外に出てしまう場合には，速度の符号を反転させ，画面内で跳ねかえるように実装します．

（ダウンロードファイル snapshot_8 を参照）

■ センサ処理の実装

● SensorManager の取得

```
/* SensorManager の取得 */
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

SensorManager は，BoundBallActivity の onResume メソッドで取得しています．Activity クラスは，ライフサイクルを持っており，onResume メソッドは，アプリケーションが実行状態になる直前に呼ばれます．それに対して onPause メソッドは，Activity が隠れる直前に呼ばれます．Android は，電話の着信や，他のアプリケーションのポップアップなどの割込みや，ホームボタンを押してホーム画面を表示する，バックライトの消灯ボタンを押すなど，アプリケーションが中断されることがしばしばありますので，最小の範囲でセンサを利用する処理を組み込まなければ，画面が隠れている間もセンサの値を取

り続け，バッテリーを大量に消費するアプリケーションとなってしまいます．バックグラウンドでセンサの値を取り続けなければならない場合を除いて，一般的には onResume メソッドと，onPause メソッドでセンサリスナの取得と解放処理を実装します．

SensorManager は，new 演算子で初期化せずに，Context クラスで定義された getSystemService メソッドを用いて Context から取得します．getSystemService メソッドは，さまざまなシステムサービスを取得することができるメソッドで，返り値が Object 型であるため，SensorManager 型へキャストする必要があります．

（ダウンロードファイル snapshot_9 を参照）

● 端末に搭載されているセンサの取得

```
@Override
protected void onResume() {
    ...
    /* 端末に搭載されているセンサの取得 */
    List<Sensor> sensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
    ...
}
```

Android の端末は，種類によってさまざまなセンサが搭載されており，どのようなセンサが搭載されているかを調べるには，SensorManager クラスの getSensorList メソッドを使用し，Sensor のリストを取得します．getSensorList メソッドの引数では，どのようなセンサを取得するかを，Sensor クラスで定義された "TYPE_" から始まる定数で指定します．端末に搭載されているすべてのセンサのリストを取得するには，Sensor クラスの "TYPE_ALL" を指定することで，取得することが可能です．

（ダウンロードファイル snapshot_10 を参照）

● センサリスナのセット

```
@Override
protected void onResume() {
    ...
    /* センサリスナのセット */
    if (s.getType() == Sensor.TYPE_ACCELEROMETER) {
        mSensorManager.registerListener(this, s, SensorManager.SENSOR_DELAY_GAME);
    }
    ...
}
```

取得したセンサのリストから，使用するセンサをセンサの種類を表す変数で判断し，そのセンサをリスナに渡します．センサの種類は，Sensor クラスの getType メソッドを使用することで取得することが可能です．

利用するセンサを見つけた場合，SensorManager クラスの，registerListener メソッドで，リスナにセンサを登録します．第 1 引数は，SensorEventListener を実装したインスタンス，第 2 引数に値を監視する Sensor インスタンス，第 3 引数で，センサをサンプリングする周期を指定します．この周期は，SensorManager クラスで宣言された "SENSOR_DELAY" から始まる定数を使用します．SENSOR_DELAY_FASTEST は，遅延時間を取らずに，最速

の時間で値を取得し，逆に SENSOR_DELAY_NORMAL は，定義された定数の中で最も遅い周期で値を取得します．取得の周期を早くすれば，更新を早くすることが可能ですが，消費電力量も増えます．アプリケーションの種類によってパフォーマンスと消費電力量の兼ね合いから，最適な遅延時間を指定する必要があります．

（ダウンロードファイル snapshot_11 を参照）

● センサ値の取得

```
@Override
public void onAccuracyChanged(Sensor sensor, int i) {
    Log.d("onAccuracyChanged. sensor = " + sensor.getName() + ", accuracy = " + i);
}
```

```
@Override
public void onSensorChanged(SensorEvent sensorevent) {
    /* センサ値の取得 */
    if (sensorevent.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        mBallView.setAcceleration(sensorevent.values);
        showSensorValues(sensorevent.values);
    }
}
```

```
private void showSensorValues(float[] values) {
   if (values == null) return;
   NumberFormat nf = new DecimalFormat("0.0000");
   StringBuilder sb = new StringBuilder();
   sb.append("x-axis(m/s^2) = ").append(nf.format(values[0]).append(BR)
      .append("y-axis(m/s^2) = ").append(nf.format(values[1])).append(BR)
      .append("z-axis(m/s^2) = ").append(nf.format(values[2]));
   mSensorValuesView.setText(sb.toString());
}
```

センサ値は，SensorEventListener インタフェースを実装することで，センサの値を取得することが可能になります．SensorEventListener インタフェースには，onSensorChanged メソッドと，onAccuracyChanged メソッドが宣言されており，このインタフェースを実装したクラスでは，2 つのメソッドを宣言する必要があります．このアプリケーションでは，BoundBallActivity クラスで SensorEventListener インタフェースを実装していますので，それぞれのメソッドを BoundBallActivity クラスにて宣言しています．

onSensorChanged では，引数として SensorEvent インスタンスを受け取ります．この SensorEvent インスタンスの中に，イベントが発生したセンサのインスタンス (sensor) や，そのセンサの値 (values)，精度 (accuracy)，タイムスタンプ (timestamp) の情報が格納されています．これらは，可視性が public で宣言されていますので，直接参照して値の取得を行います．このアプリケーションでは，センサから入力された float 型の配列を，BoundBallView にそのまま渡しています．そして showSensorValues メソッドで，センサの値を表示する TextView に，センサ値を文字列化したものを表示しています．

onAccuracyChanged は，センサの精度に変更があった場合に呼ばれるメソッドです．この

アプリケーションでは，精度を考慮しておりませんので，ログの出力処理のみ実装しています．

（ダウンロードファイル snapshot_12 を参照）

● センサリスナの解除

```
@Override
protected void onPause() {
    Log. d("onPause. ");
    /* センサリスナの解除 */
    mSensorManager. unregisterListener(this);
    super. onPause();
}
```

センサリスナをセットすると，明示的に解除のメソッドが呼ばれるまでセンサの値を取り続けます．したがいまして解除の処理を行わないと，アプリケーション終了後もセンサの値を取り続け，バッテリーの電力を大量に消費してしまいます．

今回のアプリケーションでは，Activity が表示されていない状態では，センサの値を取得する必要がありませんので，onResume メソッドにてセンサリスナの解除処理を行っています．

センサリスナを解除するには，SensorManager クラスの，unregisterListener メソッドで解除を行います．unregisterListener メソッドの引数は，SensorEventListenr インタフェースを実装したインスタンスを指定します．

（ダウンロードファイル snapshot_13 を参照）

■ 画面の方向指定

このアプリケーションは，端末を傾けて動作させるアプリケーションですので，画面の方向が縦，横に切り替わると動作に支障をきたします．したがいまして AndroidManifest.xml にて，画面の方向を縦方向に固定する設定を行います．AndroidManifest.xml を開き，「Application」-「Application Nodes」にある BoundBallActivity を選択し，「Screen orientation」を「portrait」に変更します．

（ダウンロードファイル snapshot_14 を参照）

これでセンサを用いたアプリケーションは完成となります．ほかにもさまざまなセンサが用意されておりますので，ほかのセンサを用いたアプリケーションの作成にもチャレンジしてみてください．

参考文献

1）（株）ブリリアントサービス：Android Hacks - プロが教えるテクニック & ツール，オライリー・ジャパン，東京（2010）．

2）Android Developers，http://developer.android.com/

（2011 年 10 月 3 日受付）

石丸宗平　ishimaru@nanoconnect.co.jp

（株）ナノコネクト Android アプリ開発部 主任．Android アプリケーションの教育や Android アプリケーションの開発に従事．