Android におけるカスタム描画の詳細解析

本ブログ記事は ChatGPT-4o の助けを借りて作成されました。

紹介

このブログでは、`DrawActivity` クラスについて探求します。これは、Android アプリケーションでカスタム描画ビューを実装するための包括的な例です。各コンポーネントと使用されるアルゴリズムを分解し、それらがどのように連携して必要な機能を実現するのかを詳しく説明します。

目次

DrawActivity の概要

`DrawActivity` は、描画操作、画像の切り抜き、および他のコンポーネント（フラグメントや画像アップロードなど）とのやり取りを処理する主要なアクティビティです。ユーザーが描画、元に戻す、やり直す、画像を操作できるユーザーインターフェースを提供します。

```
public class DrawActivity extends Activity implements View.OnClickListener {
  // リクエストコードとフラグメント ID の定数
```

```
    public static final int CAMERA_RESULT = 1;
    public static final int CROP_RESULT = 2;
    public static final int DRAW_FRAGMENT = 0;
    public static final int RECOG_FRAGMENT = 1;
    public static final int RESULT_FRAGMENT = 2;
    public static final int WAIT_FRAGMENT = 3;
    public static final int MATERIAL_RESULT = 4;
    public static final String RESULT_JSON = "resultJson";
    public static final int INIT_FLOWER_ID = R.drawable.flower_b;
    public static final int LOGOUT = 0;
    public static final int IMAGE_RESULT = 0;

    // 画像と描画操作を処理する変数
    String baseUrl;
    DrawView drawView;
    Bitmap originImg;
    public static DrawActivity instance;
    View dir, clear, cameraView, materialView, scale;
    ImageView undoView, redoView;
    View upload;
    String cropPath;
    Tooltip toolTip;
    int curFragmentId = -1;
    int serverId = -1;
    private Bitmap resultBitmap;
    private RadioGroup radioGroup;
    Fragment curFragment;
    int curDrawMode;
    RadioButton drawBackBtn;
    private Activity cxt;
    Uri curPicUri;
}
```

Activity の初期化

Activity の作成時に、さまざまな初期化操作を実行します。これには、ビューの設定、初期画像の読み込み、イベントリスナーの設定などが含まれます。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  instance = this;
  cxt = this;
  cropPath = PathUtils.getCropPath();
  setContentView(R.layout.draw_layout);
  findView();
  setSize();
  initOriginImage();
  toolTip = new Tooltip(this);
  initUndoRedoEnable();
  setIp();
  initDrawmode();
}
```

このコードは、Android アプリケーションの `Activity` クラスにおける `onCreate` メソッドの実装です。以下に各ステップの説明を日本語で示します。

1. `super.onCreate(savedInstanceState);`: 親クラスの `onCreate` メソッドを呼び出し、アクティビティの基本的な初期化を行います。
2. `instance = this;`: 現在のアクティビティインスタンスを `instance` 変数に保存します。これにより、他のクラスやメソッドからこのアクティビティにアクセスできるようになります。
3. `cxt = this;`: 現在のコンテキスト（この場合はアクティビティ自身）を `cxt` 変数に保存します。コンテキストは、リソースやシステムサービスにアクセスするために使用されます。
4. `cropPath = PathUtils.getCropPath();`: `PathUtils` クラスの `getCropPath` メソッドを呼び出して、画像の切り抜きパスを取得し、`cropPath` 変数に保存します。
5. `setContentView(R.layout.draw_layout);`: `draw_layout` というレイアウトリソースをアクティビティのビューとして設定します。
6. `findView();`: レイアウト内のビュー（ボタン、テキストビューなど）を初期化するためのメソッドを呼び出します。

7. `setSize();`: ビューや画像のサイズを設定するためのメソッドを呼び出します。
8. `initOriginImage();`: 元の画像を初期化するためのメソッドを呼び出します。
9. `toolTip = new Tooltip(this);`: `Tooltip` クラスの新しいインスタンスを作成し、`toolTip` 変数に保存します。ツールチップは、ユーザーに追加情報を提供するために使用されます。
10. `initUndoRedoEnable();`: 元に戻す（Undo）とやり直す（Redo）の機能を初期化するためのメソッドを呼び出します。
11. `setIp();`: IP アドレスを設定するためのメソッドを呼び出します。
12. `initDrawmode();`: 描画モードを初期化するためのメソッドを呼び出します。

このコードは、アクティビティが作成される際に、必要な初期化処理を一連のメソッド呼び出しを通じて行っています。

findView()
このメソッドは、Activity で使用されるビューを初期化します。

```
private void findView() {
  drawView = findViewById(R.id.drawView);
  undoView = findViewById(R.id.undo);
  redoView = findViewById(R.id.redo);
  scale = findViewById(R.id.scale);
  upload = findViewById(R.id.upload);
  clear = findViewById(R.id.clear);
  dir = findViewById(R.id.dir);
  materialView = findViewById(R.id.material);
  cameraView = findViewById(R.id.camera);

dir.setOnClickListener(this);
materialView.setOnClickListener(this);
undoView.setOnClickListener(this);
scale.setOnClickListener(this);
redoView.setOnClickListener(this);
clear.setOnClickListener(this);
cameraView.setOnClickListener(this);
upload.setOnClickListener(this);
initRadio();
}
```

setSize()
描画ビューのサイズを設定します。

```
private void setSize() {
  setSizeByResourceSize();
  setViewSize(drawView);
}
```

このコードは、Java で書かれたメソッド `setSize()` の定義です。このメソッドは、リソースサイズに基づいてサイズを設定し、その後、指定されたビュー（`drawView`）のサイズを設定する役割を持っています。具体的な処理内容は、`setSizeByResourceSize()` と `setViewSize(drawView)` という別のメソッドに委ねられています。

```
private void setSizeByResourceSize() {
  int width = getResources().getDimensionPixelSize(R.dimen.draw_width);
  int height = getResources().getDimensionPixelSize(R.dimen.draw_height);
  App.drawWidth = width;
  App.drawHeight = height;
}
```

このメソッドは、リソースから取得したサイズに基づいて描画サイズを設定するためのものです。具体的には、`R.dimen.draw_width` と `R.dimen.draw_height` で定義された寸法をピクセル単位で取得し、それらの値を `App` クラスの静的変数 `drawWidth` と `drawHeight` に設定しています。これにより、アプリケーション全体で使用される描画サイズがリソースファイルで定義された値に基づいて調整されます。

```
private void setViewSize(View v) {
  ViewGroup.LayoutParams lp = v.getLayoutParams();
  lp.width = App.drawWidth;
  lp.height = App.drawHeight;
  v.setLayoutParams(lp);
}
```

initOriginImage()
描画に使用する初期画像を読み込みます。

```
private void initOriginImage() {
  Bitmap bitmap = BitmapFactory.decodeResource(getResources(), INIT_FLOWER_ID);
```

```
  String imgPath = PathUtils.getCameraPath();
  BitmapUtils.saveBitmapToPath(bitmap, imgPath);
  Uri uri1 = Uri.fromFile(new File(imgPath));
  setImageByUri(uri1);
}
```

このコードは、初期の画像を設定するためのメソッドです。以下にその内容を説明します。

1. `Bitmap bitmap = BitmapFactory.decodeResource(getResources(), INIT_FLOWER_ID);`
    - `INIT_FLOWER_ID` で指定されたリソース ID からビットマップ画像をデコードします。
2. `String imgPath = PathUtils.getCameraPath();`
    - `PathUtils.getCameraPath()` メソッドを使用して、画像を保存するためのパスを取得します。
3. `BitmapUtils.saveBitmapToPath(bitmap, imgPath);`
    - デコードされたビットマップ画像を指定されたパスに保存します。
4. `Uri uri1 = Uri.fromFile(new File(imgPath));`
    - 保存された画像ファイルの URI を生成します。
5. `setImageByUri(uri1);`
    - 生成された URI を使用して、画像を設定します。

このメソッドは、指定されたリソースから画像を読み込み、それを指定されたパスに保存し、その画像を表示するために使用されます。

画像操作の処理

Activity は、URI を介した画像の設定、切り抜き、描画されたビットマップの保存など、さまざまな画像操作を処理します。

setImageByUri(Uri uri)
指定された URI から画像を読み込み、描画の準備を行います。

```
private void setImageByUri(final Uri uri) {
  new Handler().postDelayed(new Runnable() {
    @Override
    public void run() {
      curPicUri = uri;
      Bitmap bitmap = null;
      try {
        if (uri != null) {
          bitmap = BitmapUtils.getBitmapByUri(DrawActivity.this, uri);
        }
      } catch (Exception e) {
        e.printStackTrace();
      }

int originW = bitmap.getWidth();
int originH = bitmap.getHeight();
if (originW != App.drawWidth || originH != App.drawHeight) {
    float originRadio = originW * 1.0f / originH;
    float radio = App.drawWidth * 1.0f / App.drawHeight;
    if (Math.abs(originRadio - radio) < 0.01) {
        Bitmap originBm = bitmap;
        bitmap = Bitmap.createScaledBitmap(originBm, App.drawWidth, App.drawHeight, false);
        originBm.recycle();
    } else {
        cropIt(uri);
        return;
    }
}
ImageLoader imageLoader = ImageLoader.getInstance();
imageLoader.addOrReplaceToMemoryCache("origin", bitmap);
originImg = bitmap;
serverId = -1;
```

このコードは、ビットマップのサイズが指定された描画サイズ（`App.drawWidth` と `App.drawHeight`）と異なる場合に、ビットマップをスケーリングまたはクロップする処理を行っています。以下にその内容を説明します。

1. **オリジナルのビットマップサイズを取得**:
    - `originW` と `originH` にビットマップの幅と高さを取得します。
2. **サイズが異なる場合の処理**:
    - オリジナルのビットマップサイズが指定された描画サイズと異なる場合、以下の処理を行います。
    - オリジナルのアスペクト比 (`originRadio`) と指定された描画サイズのアスペクト比 (`radio`) を計算します。
    - 両者のアスペクト比がほぼ同じ場合（差が 0.01 未満)、ビットマップを指定された描画サイズにスケーリングします。
    - アスペクト比が異なる場合、`cropIt(uri)` を呼び出してビットマップをクロップし、処理を終了します。
3. **スケーリング後のビットマップをキャッシュに保存**:
    - スケーリングされたビットマップを `ImageLoader` のメモリキャッシュに保存します。
    - `originImg` にスケーリングされたビットマップを設定し、`serverId` を -1 に設定します。

このコードは、ビットマップのサイズを調整して、指定された描画サイズに合わせるための処理を行っています。

```
drawView.setSrcBitmap(originImg);
showDrawFragment(App.ALL_INFO);
curDrawMode = App.DRAW_FORE;

}, 500); }
```

上記のコードは、`drawView`に元の画像(`originImg`)を設定し、`App.ALL_INFO`を表示する描画フラグメントを表示し、現

cropIt(Uri uri)
画像の切り抜きアクティビティを開始します。

```java
public void cropIt(Uri uri) {
  Crop.startPhotoCrop(this, uri, cropPath, CROP_RESULT);
}
```

このコードは､指定されたURIの画像をトリミングするためのメソッドです｡`Crop.startPhotoCrop` メソッドを呼び出して、トリミング処理を開始します。`this` は現在のコンテキストを指し、`uri`

はトリミングする画像の URI、cropPath はトリミング後の画像の保存パス、CROP_RESULT はトリミング結果を受け取るためのリクエストコードです。

saveBitmap()
描画したビットマップをファイルに保存し、サーバーにアップロードします。

```
public void saveBitmap() {
  Bitmap handBitmap = drawView.getHandBitmap();
  Bitmap originBitmap = drawView.getSrcBitmap();
  saveBitmapToFileAndUpload(handBitmap, originBitmap);
}
```

このコードは、drawView から手書きのビットマップ (handBitmap) と元のビットマップ (originBitmap) を取得し、それらをファイルに保存してアップロードするメソッド saveBitmap を定義しています。コード自体は日本語に翻訳する必要はありませんが、その機能を説明すると以下のようになります。

- drawView.getHandBitmap(): drawView から手書きのビットマップを取得します。
- drawView.getSrcBitmap(): drawView から元のビットマップを取得します。
- saveBitmapToFileAndUpload(handBitmap, originBitmap): 取得した 2 つのビットマップをファイルに保存し、アップロードします。

このメソッドは、ユーザーが描画した内容と元の画像を保存してサーバーにアップロードするために使用されることが想定されます。

saveBitmapToFileAndUpload(Bitmap handBitmap, Bitmap originBitmap)
ビットマップをファイルに保存し、非同期でアップロードします。

```
private void saveBitmapToFileAndUpload(Bitmap handBitmap, Bitmap originBitmap) {
  final String originPath = PathUtils.getOriginPath();
  BitmapUtils.saveBitmapToPath(originBitmap, originPath);
  final String handPath = PathUtils.getHandPath();
  BitmapUtils.saveBitmapToPath(handBitmap, handPath);
  new AsyncTask<Void, Void, Void>() {
    boolean res;
    Bitmap foreBitmap;
    Bitmap backBitmap;
```

```
@Override
protected void onPreExecute() {
  super.onPreExecute();
  showWaitFragment();
}
```

上記のコードは、onPreExecute メソッドをオーバーライドしています。このメソッドは、バックグラウンドタスクが実行される前に呼び出されます。super.onPreExecute() を呼び出して親クラスの処理を実行した後、showWaitFragment() メソッドを呼び出して待機フラグメントを表示しています。

```
@Override
protected Void doInBackground(Void... params) {
    try {
        if (baseUrl == null) {
            throw new Exception("baseUrl が null です");
        }
        String jsonRes = UploadImage.upload(baseUrl, serverId, Web.STATUS_CONTINUE, originPath, handPath
        getJsonData(jsonRes);
        res = true;
    } catch (Exception e) {
        res = false;
        e.printStackTrace();
    }
    return null;
}
```

このコードは、バックグラウンドで実行される非同期タスクの一部です。baseUrl が null の場合に例外をスローし、それ以外の場合は UploadImage.upload メソッドを呼び出して画像をアップロードし、その結果を処理しています。エラーが発生した場合は res を false に設定し、例外のスタックトレースを出力します。

```
private void getJsonData(String jsonRes) throws Exception {
    JSONObject json = new JSONObject(jsonRes);
    if (serverId == -1) {
        serverId = json.getInt(Web.ID);
    }
```

```
        String foreUrl = json.getString(Web.FORE);
        String backUrl = json.getString(Web.BACK);
        String resultUrl = json.getString(Web.RESULT);
        foreBitmap = Web.getBitmapFromUrlByStream1(foreUrl, 0);
        backBitmap = Web.getBitmapFromUrlByStream1(backUrl, 0);
        resultBitmap = Web.getBitmapFromUrlByStream1(resultUrl, 0);
    }

    @Override
    protected void onPostExecute(Void aVoid) {
        super.onPostExecute(aVoid);
        if (res) {
            showRecogFragment(foreBitmap, backBitmap);
        } else {
            Utils.toast(DrawActivity.this, R.string.server_error);
            recogNo();
        }
    }
```

このコードは、非同期タスクの実行が完了した後に呼び出される `onPostExecute` メソッドをオーバーライドしています。`res` が `true` の場合、`showRecogFragment` メソッドを呼び出して認識フラグメントを表示します。`res` が `false` の場合、サーバーエラーのメッセージを表示し、`recogNo` メソッドを呼び出します。

```
      }.execute();
    }
```

フラグメント管理

Activity は、描画、認識、待機など、アプリケーションのさまざまな状態を処理するために、異なる fragment を管理します。

showDrawFragment(int infoId)
描画フラグメントを表示します。

```
private void showDrawFragment(int infoId) {
  curFragmentId = DRAW_FRAGMENT;
  curFragment = new DrawFragment(infoId);
  showFragment(curFragment);
}
```

このコードは、DrawFragment を表示するためのメソッドです。infoId を引数として受け取り、新しい DrawFragment インスタンスを作成し、それを表示します。curFragmentId には DRAW_FRAGMENT が設定され、curFragment には新しく作成された DrawFragment が代入されます。その後、showFragment メソッドを呼び出して、フラグメントを表示します。

showWaitFragment()
待機フラグメントを表示します。

```
private void showWaitFragment() {
  curFragmentId = WAIT_FRAGMENT;
  showFragment(new WaitFragment());
}
```

このコードは、showWaitFragment というプライベートメソッドを定義しています。このメソッドは、curFragmentId に WAIT_FRAGMENT という定数を設定し、WaitFragment という新しいフラグメントを表示するために showFragment メソッドを呼び出します。

showFragment(Fragment fragment)
指定された fragment で現在の fragment を置き換えます。

```
private void showFragment(Fragment fragment) {
  FragmentTransaction trans = getFragmentManager().beginTransaction();
  trans.replace(R.id.rightLayout, fragment);
  trans.commit();
}
```

このコードは、指定されたフラグメントを表示するためのメソッドです。FragmentTransaction を使用して、R.id.rightLayout に指定されたレイアウト内の現在のフラグメントを新しいフラグメントに置き換え、変更をコミットしています。

イベント処理

Activity は、ボタンクリックやメニュー選択など、さまざまなユーザーインタラクションを処理します。

onClick(View v)
異なるビューのクリックイベントを処理します。

```
@Override
public void onClick(View v) {
  int id = v.getId();
  if (id == R.id.drawOk) {
    if (drawView.isDrawFinish()) {
      saveBitmap();
    } else {
      Utils.alertDialog(this, R.string.please_draw_finish);
    }
  } else if (id == R.id.recogOk) {
    recogOk();
  } else if (id == R.id.recogNo) {
    recogNo();
  } else if (id == R.id.dir) {
    Utils.getGalleryPhoto(this, IMAGE_RESULT);
  } else if (id == R.id.clear) {
    clearEverything();
  } else if (id == R.id.undo) {
    drawView.undo();
  } else if (id == R.id.redo) {
    drawView.redo();
  } else if (id == R.id.camera) {
    Utils.takePhoto(cxt, CAMERA_RESULT);
  } else if (id == R.id.material) {
    goMaterial();
  } else if (id == R.id.upload) {
    com.lzw.commons.Utils.goActivity(cxt, PhotoActivity.class);
  } else if (id == R.id.scale) {
    cropIt(curPicUri);
```

```
  }
}
```

このコードは、Android アプリケーションにおけるボタンクリックイベントを処理するための `onClick` メソッドの実装です。各ボタンの ID に応じて異なるアクションが実行されます。以下に各条件分岐の説明を日本語で示します。

- `R.id.drawOk`: 描画が完了しているかどうかを確認し、完了していればビットマップを保存します。完了していない場合は、ユーザーに描画を完了するよう促すダイアログを表示します。
- `R.id.recogOk`: `recogOk` メソッドを呼び出します。
- `R.id.recogNo`: `recogNo` メソッドを呼び出します。
- `R.id.dir`: ギャラリーから写真を取得します。
- `R.id.clear`: すべてをクリアします。
- `R.id.undo`: 描画ビューで最後の操作を取り消します。
- `R.id.redo`: 描画ビューで最後に取り消した操作を再実行します。
- `R.id.camera`: カメラで写真を撮影します。
- `R.id.material`: `goMaterial` メソッドを呼び出します。
- `R.id.upload`: `PhotoActivity` に遷移します。
- `R.id.scale`: 現在の画像 URI をトリミングします。

このコードは、ユーザーの操作に応じて適切な処理を行うための典型的なパターンを示しています。

onActivityResult(int requestCode, int resultCode, Intent data)
他のアクティビティの結果を処理します。例えば、画像の選択やトリミングなどです。

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
  if (resultCode != RESULT_CANCELED) {
    Uri uri;
    switch (requestCode) {
      case IMAGE_RESULT:
        if (data != null) {
          setImageByUri(data.getData());
        }
        break;
      case CAMERA_RESULT:
```

```
        setImageByUri(Utils.getCameraUri());
        break;
      case CROP_RESULT:
        uri = Uri.fromFile(new File(cropPath));
        setImageByUri(uri);
        break;
      case MATERIAL_RESULT:
        setImageByUri(data.getData());
    }
  }
}
```

このコードは、Android アプリケーションでのアクティビティ結果を処理するためのメソッドです。以下にその内容を説明します。

- `onActivityResult` メソッドは、他のアクティビティから結果を受け取ったときに呼び出されます。
- `resultCode` が `RESULT_CANCELED` でない場合、つまり結果がキャンセルされていない場合に処理が行われます。
- `requestCode` に応じて、どのアクティビティからの結果かを判断し、適切な処理を行います。
  - `IMAGE_RESULT`: 画像選択の結果を受け取り、選択された画像の URI を使って画像を設定します。
  - `CAMERA_RESULT`: カメラで撮影した画像の URI を使って画像を設定します。
  - `CROP_RESULT`: 切り抜き処理が行われた画像の URI を使って画像を設定します。
  - `MATERIAL_RESULT`: マテリアルデザイン関連の結果を受け取り、その URI を使って画像を設定します。

このメソッドは、画像選択やカメラ撮影、画像の切り抜きなど、ユーザーが画像を操作した後の処理を行うために使用されます。

元に戻すとやり直しの機能

Activity は、描画操作の取り消し（Undo）とやり直し（Redo）の機能を提供します。

initUndoRedoEnable()
コールバック関数を設定して、元に戻す（Undo）とやり直す（Redo）機能を初期化します。

```
void initUndoRedoEnable() {
  drawView.history.setCallBack(new History.CallBack() {
    @Override
    public void onHistoryChanged() {
      setUndoRedoEnable();
      if (curFragmentId != DRAW_FRAGMENT) {
        showDrawFragment(curDrawMode);
      }
    }
  });
}
```

このコードは、`initUndoRedoEnable` メソッドを定義しています。このメソッドは、`drawView.history` にコールバックを設定し、履歴が変更されたときに `setUndoRedoEnable` メソッドを呼び出し、現在のフラグメントが `DRAW_FRAGMENT` でない場合に `showDrawFragment` メソッドを呼び出して描画フラグメントを表示します。

```
void setUndoRedoEnable() {
  redoView.setEnabled(drawView.history.canRedo());
  undoView.setEnabled(drawView.history.canUndo());
}
```

カスタム DrawView

`DrawView` は、描画操作、タッチイベント、およびズームを処理するためのカスタムビューです。

onTouchEvent(MotionEvent event)
描画とスケーリングのためのタッチイベントを処理します。

```
@Override
public boolean onTouchEvent(MotionEvent event) {
  if (!scaleMode) {
    handleDrawTouchEvent(event);
  } else {
    handleScaleTouchEvent(event);
  }
```

```
  return true;
}
```

このコードは、onTouchEvent メソッドをオーバーライドしており、タッチイベントを処理するためのものです。scaleMode が false の場合、handleDrawTouchEvent メソッドが呼び出され、scaleMode が true の場合、handleScaleTouchEvent メソッドが呼び出されます。最後に、true を返すことで、イベントが処理されたことを示しています。

```
private void handleDrawTouchEvent(MotionEvent event) {
  int action = event.getAction();
  float x = event.getX();
  float y = event.getY();
  if (action == MotionEvent.ACTION_DOWN) {
    path.moveTo(x, y);
  } else if (action == MotionEvent.ACTION_MOVE) {
    path.quadTo(preX, preY, x, y);
  } else if (action == MotionEvent.ACTION_UP) {
    Matrix matrix1 = new Matrix();
    matrix.invert(matrix1);
    path.transform(matrix1);
    paint.setStrokeWidth(strokeWidth * 1.0f / totalRatio);
    history.saveToStack(path, paint);
    cacheCanvas.drawPath(path, paint);
    paint.setStrokeWidth(strokeWidth);
    path.reset();
  }
  setPrev(event);
  invalidate();
}
```

このコードは、タッチイベントを処理して描画を行うメソッドです。以下にその動作を説明します。

1. **タッチイベントの取得**:
   - event.getAction() でタッチイベントのアクション（押下、移動、離す）を取得します。
   - event.getX() と event.getY() でタッチ位置の座標を取得します。
2. **ACTION_DOWN**:

- タッチが開始された場合、`path.moveTo(x, y)` でパスの開始点を設定します。

3. **ACTION_MOVE**:
   - タッチが移動した場合、`path.quadTo(preX, preY, x, y)` で前回の座標から現在の座標までの曲線を描画します。

4. **ACTION_UP**:
   - タッチが終了した場合、以下の処理を行います：
     - `Matrix` オブジェクトを作成し、現在のマトリックスを反転させます。
     - パスに反転したマトリックスを適用します。
     - ペイントのストローク幅を調整し、履歴にパスとペイントを保存します。
     - キャンバスにパスを描画し、ペイントのストローク幅を元に戻します。
     - パスをリセットします。

5. **前回の座標を更新**:
   - `setPrev(event)` で前回の座標を更新します。

6. **再描画**:
   - `invalidate()` を呼び出して、ビューを再描画します。

このメソッドは、ユーザーが画面に指で描画する際の動作を制御し、描画結果をキャンバスに反映します。

```
private void handleScaleTouchEvent(MotionEvent event) {
  switch (event.getActionMasked()) {
    case MotionEvent.ACTION_POINTER_DOWN:
      lastFingerDist = calFingerDistance(event);
      break;
    case MotionEvent.ACTION_MOVE:
      if (event.getPointerCount() == 1) {
        handleMove(event);
      } else if (event.getPointerCount() == 2) {
        handleZoom(event);
      }
      break;
    case MotionEvent.ACTION_UP:
    case MotionEvent.ACTION_POINTER_UP:
      lastMoveX = -1;
```

```
      lastMoveY = -1;
      break;
    default:
      break;
  }
}
```

このコードは、タッチイベントを処理するためのメソッドです。以下に各ケースの説明を日本語で示します。

- **ACTION_POINTER_DOWN**: 2 本目の指が画面に触れたときに、指の間の距離を計算して `lastFingerDist` に保存します。
- **ACTION_MOVE**: 指が動いたときに、指の本数に応じて処理を分岐します。1 本の場合は `handleMove` メソッドを呼び出し、2 本の場合は `handleZoom` メソッドを呼び出します。
- **ACTION_UP** および **ACTION_POINTER_UP**: 指が離れたときに、最後の移動位置をリセットします。
- **default**: その他のアクションでは何も処理しません。

```
private void handleMove(MotionEvent event) {
  float moveX = event.getX();
  float moveY = event.getY();
  if (lastMoveX == -1 && lastMoveY == -1) {
    lastMoveX = moveX;
    lastMoveY = moveY;
  }
  moveDistX = (int) (moveX - lastMoveX);
  moveDistY = (int) (moveY - lastMoveY);
  if (moveDistX + totalTranslateX > 0 || moveDistX + totalTranslateX + curBitmapWidth < width) {
    moveDistX = 0;
  }
  if (moveDistY + totalTranslateY > 0 || moveDistY + totalTranslateY + curBitmapHeight < height) {
    moveDistY = 0;
  }
  status = STATUS_MOVE;
  invalidate();
  lastMoveX = moveX;
```

```
  lastMoveY = moveY;
}
```

このコードは、タッチイベントを処理してビットマップの移動を制御するメソッドです。以下にその動作を説明します。

1. **イベント座標の取得**: `event.getX()` と `event.getY()` を使用して、現在のタッチ位置を取得します。
2. **初期位置の設定**: `lastMoveX` と `lastMoveY` が初期値（-1）の場合、現在の位置を初期位置として設定します。
3. **移動距離の計算**: 現在の位置と前回の位置の差を計算し、`moveDistX` と `moveDistY` に格納します。
4. **移動範囲の制限**: ビットマップが画面の境界を超えないように、移動距離を調整します。もし移動後の位置が画面の外に出る場合、移動距離を 0 に設定します。
5. **状態の更新**: 移動中であることを示すために、`status` を `STATUS_MOVE` に設定します。
6. **再描画の要求**: `invalidate()` を呼び出して、ビューを再描画します。
7. **前回の位置の更新**: 現在の位置を `lastMoveX` と `lastMoveY` に保存し、次のイベント処理に備えます。

このメソッドは、ユーザーが画面をタッチしてビットマップを移動させる際に、ビットマップが画面の外に出ないように制御する役割を果たします。

```
private void handleZoom(MotionEvent event) {
  float fingerDist = calFingerDistance(event);
  calFingerCenter(event);
  if (fingerDist > lastFingerDist) {
    status = STATUS_ZOOM_OUT;
  } else {
    status = STATUS_ZOOM_IN;
  }
  scaledRatio = fingerDist * 1.0f / lastFingerDist;
  totalRatio = totalRatio * scaledRatio;
  if (totalRatio < initRatio) {
    totalRatio = initRatio;
```

```
  } else if (totalRatio > initRatio * 4) {
    totalRatio = initRatio * 4;
  }
  lastFingerDist = fingerDist;
  invalidate();
}
```

onDraw(Canvas canvas)
ビューの現在の状態を描画します。

```
@Override
protected void onDraw(Canvas canvas) {
  super.onDraw(canvas);
  if (scaleMode) {
    switch (status) {
      case STATUS_MOVE:
        move(canvas);
        break;
      case STATUS_ZOOM_IN:
      case STATUS_ZOOM_OUT:
        zoom(canvas);
        break;
      default:
        if (cacheBm != null) {
          canvas.drawBitmap(cacheBm, matrix, null);
          canvas.drawPath(path, paint);
        }
    }
  } else {
    if (cacheBm != null) {
      canvas.drawBitmap(cacheBm, matrix, null);
      canvas.drawPath(path, paint);
    }
  }
}
```

このコードは、Java で書かれた `onDraw` メソッドのオーバーライドです。このメソッドは、

Canvas オブジェクトを使用して描画を行います。scaleMode が有効な場合、status に応じて異なる描画処理を行います。STATUS_MOVE の場合は move メソッドを、STATUS_ZOOM_IN または STATUS_ZOOM_OUT の場合は zoom メソッドを呼び出します。それ以外の場合は、cacheBm が null でなければ、cacheBm を matrix に従って描画し、path を paint で描画します。scaleMode が無効な場合も同様に、cacheBm が null でなければ、cacheBm と path を描画します。

move(Canvas canvas)
ズーム中の移動操作を処理します。

```
private void move(Canvas canvas) {
  matrix.reset();
  matrix.postScale(totalRatio, totalRatio);
  totalTranslateX = moveDistX + totalTranslateX;
  totalTranslateY = moveDistY + totalTranslateY;
  matrix.postTranslate(totalTranslateX, totalTranslateY);
  canvas.drawBitmap(cacheBm, matrix, null);
}
```

このコードは、Java で Canvas 上にビットマップを移動させるためのメソッドです。以下にその内容を説明します。

1. `matrix.reset();`
   行列（Matrix）をリセットして初期状態に戻します。
2. `matrix.postScale(totalRatio, totalRatio);`
   行列にスケーリング（拡大縮小）を適用します。totalRatio は現在の拡大率を表します。
3. `totalTranslateX = moveDistX + totalTranslateX;`
   X 軸方向の移動距離を更新します。moveDistX は今回の移動量で、totalTranslateX は累積された移動量です。
4. `totalTranslateY = moveDistY + totalTranslateY;`
   Y 軸方向の移動距離を更新します。moveDistY は今回の移動量で、totalTranslateY は累積された移動量です。
5. `matrix.postTranslate(totalTranslateX, totalTranslateY);`
   行列に平行移動を適用します。totalTranslateX と totalTranslateY は、それぞれ X 軸と Y 軸方向の累積移動量です。

6. `canvas.drawBitmap(cacheBm, matrix, null);`
   更新された行列を使用して、キャッシュされたビットマップ（`cacheBm`）を Canvas に描画します。

このメソッドは、ビットマップを指定された距離だけ移動させ、Canvas 上に描画するための処理を行います。

zoom(Canvas canvas)
ズーム操作を処理します。

```
private void zoom(Canvas canvas) {
  matrix.reset();
  matrix.postScale(totalRatio, totalRatio);
  int scaledWidth = (int) (cacheBm.getWidth() * totalRatio);
  int scaledHeight = (int) (cacheBm.getHeight() * totalRatio);
  int translateX;
  int translateY;
  if (curBitmapWidth < width) {
    translateX = (width - scaledWidth) / 2;
  } else {
    translateX = (int) (centerPointX + (totalTranslateX - centerPointX) * scaledRatio);
    if (translateX > 0) {
      translateX = 0;
    } else if (scaledWidth + translateX < width) {
      translateX = width - scaledWidth;
    }
  }
  if (curBitmapHeight < height) {
    translateY = (height - scaledHeight) / 2;
  } else {
    translateY = (int) (centerPointY + (totalTranslateY - centerPointY) * scaledRatio);
    if (translateY > 0) {
      translateY = 0;
    } else if (scaledHeight + translateY < height) {
      translateY = height - scaledHeight;
    }
  }
```

```
  matrix.postTranslate(translateX, translateY);
  canvas.drawBitmap(cacheBm, matrix, null);
}

Y = height - scaledHeight;
    }
  }
  totalTranslateX = translateX;
  totalTranslateY = translateY;
  curBitmapWidth = scaledWidth;
  curBitmapHeight = scaledHeight;
  matrix.postTranslate(translateX, translateY);
  canvas.drawBitmap(cacheBm, matrix, null);
}
```

上記のコードは、画像のスケーリングと位置調整を行い、キャンバスに描画する処理を行っています。具体的には、`Y = height - scaledHeight;` で画像の Y 座標を調整し、`matrix.postTranslate(translateX, translateY);` で画像を指定された位置に移動させています。最後に、`canvas.drawBitmap(cacheBm, matrix, null);` で調整された画像をキャンバスに描画しています。

ヒストリ管理

`History` クラスは、描画の履歴を管理し、元に戻す（Undo）とやり直す（Redo）機能を実現します。

saveToStack(Path path, Paint paint)
現在のパスとペイントをスタックに保存します。

```
public void saveToStack(Path path, Paint paint) {
  Draw draw = new Draw();
  draw.path = new Path(path);
  draw.paint = new Paint(paint);
  saveToStack(draw);
}
```

このコードは、指定された Path と Paint オブジェクトを使用して新しい Draw オブジェクトを作成し、それをスタックに保存するメソッドです。Draw クラスは、path と paint というフィールドを持っていると推測されます。このメソッドは、Draw オブジェクトをスタックに保存するために、別の saveToStack メソッドを呼び出しています。

```
public void saveToStack(Draw draw) {
  curPos++;
  while (histroy.size() > curPos) {
    histroy.pop();
  }
  histroy.push(draw);
  if (callBack != null) {
    callBack.onHistoryChanged();
  }
}
```

このコードは、Draw オブジェクトをスタックに保存するメソッドです。以下にその動作を説明します。

1. curPos をインクリメントします。これは、現在の位置を更新するためです。
2. histroy のサイズが curPos より大きい場合、histroy から要素を取り除きます。これにより、新しい Draw オブジェクトを追加する前に、古い履歴をクリアします。
3. histroy に新しい Draw オブジェクトをプッシュします。
4. callBack が null でない場合、callBack.onHistoryChanged() を呼び出して、履歴が変更されたことを通知します。

このメソッドは、履歴管理を行う際に使用されることが多いです。

getBitmapAtDraw(int n)
指定されたポイントの状態を表すビットマップを返します。

```
public Bitmap getBitmapAtDraw(int n) {
  Canvas canvas = new Canvas();
  Bitmap bm = Utils.getCopyBitmap(srcBitmap);
  canvas.setBitmap(bm);
  for (int i = 0; i <= n; i++) {
    Draw draw = histroy.get(i);
    canvas.drawPath(draw.path, draw.paint);
```

```
  }
  return bm;
}
```

undo()
元に戻す操作を実行します。

```
public Bitmap undo() throws UnsupportedOperationException {
  if (canUndo()) {
    curPos--;
    if (callBack != null) {
      callBack.onHistoryChanged();
    }
    return getBitmapAtDraw(curPos);
  } else {
    throw new UnsupportedOperationException(" 取り消す記録がありません");
  }
}
```

redo()
再実行操作を実行します。

```
public Bitmap redo() throws UnsupportedOperationException {
  if (canRedo()) {
    curPos++;
    if (callBack != null) {
      callBack.onHistoryChanged();
    }
    return getBitmapAtDraw(curPos);
  } else {
    throw new UnsupportedOperationException(" リドゥ可能な記録がありません");
  }
}
```

canUndo()
アンドゥが可能かどうかを確認します。

```
public boolean canUndo() {
  return curPos > 0;
}
```

このコードは、canUndo メソッドを定義しています。このメソッドは、curPos（現在の位置）が 0 より大きい場合に true を返し、それ以外の場合に false を返します。これにより、元に戻す操作が可能かどうかを判定します。

canRedo()
再実行が可能かどうかを確認します。

```
public boolean canRedo() {
  return curPos + 1 < histroy.size();
}
```

このコードは、canRedo メソッドを定義しています。このメソッドは、現在の位置（curPos）に 1 を加えた値が履歴（histroy）のサイズよりも小さいかどうかをチェックします。もし小さい場合、true を返し、それ以外の場合は false を返します。これは、ユーザーが「やり直し」操作を行えるかどうかを判断するために使用されます。

結論

DrawActivity とその関連クラスは、Android でカスタム描画ビューを実装するための包括的な例を提供します。これには、タッチイベントの処理、描画履歴の管理、そして fragment や非同期タスクなどの他のコンポーネントとの統合など、さまざまな技術が含まれています。各コンポーネントとアルゴリズムを理解することで、これらの技術を活用して、強力でインタラクティブな描画機能を自身のアプリに組み込むことができます。