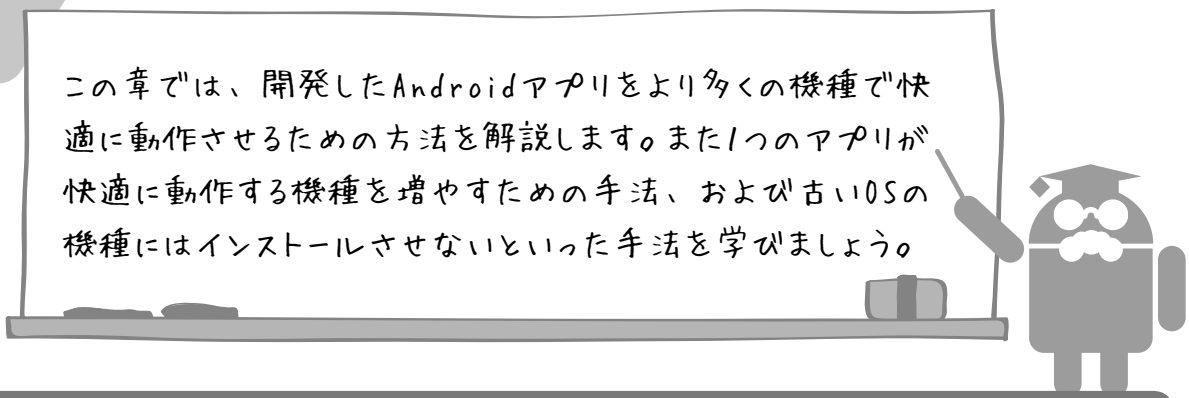
第19章

多機種対応

著：深見浩和

19 多機種対応

著：深見浩和



この節で学ぶこと

- 多機種対応の重要さ
- ピクセル密度の異なる機種でも見え方、ユーザー体験の変わらないレイアウトを作る方法
- ピクセル密度に応じた画像を用意する方法
- スクリーンサイズの違いに対応する方法
- OSバージョンの違いに対応する方法
- 横持ちに対応する方法
- エミュレーターで古いOSでの見え方や動作を確認する方法

この節で出てくるキーワード一覧

スクリーンサイズ
ピクセル密度(density)
dp(dip)
sp
dimen
layout-sw600dp / layout-sw720dp
screenOrientation
API Level
minSDKVersion

この章で出てくるクラスや定数

LayoutParams.MATCH_PARENT
LayoutParams.WRAP_CONTENT
Build.VERSION.SDK_INT

19-1 なぜ多機種対応が重要か

Androidを搭載している端末の種類はものすごい勢いで増加しています。端末メーカーは大きさやセンサーの有無などで特色を出そうとしています。そのため、機種毎に次のような違いが存在します。

- **OSのバージョン**
- **画面サイズ**
- **通話機能やセンサーなどの有無**

みなさんは、すばらしいユーザー体験を提供するアプリの開発者です。もし、このような機種毎の違いを考慮せずにアプリを開発した場合、すばらしいユーザー体験を一部の端末ユーザーのみにしか提供できなくなってしまいます。より多くのユーザーに開発したアプリを使用してもらうには、機種によってユーザー体験が低下しないようにする必要があります。この、機種毎の違いを考慮し、ユーザー体験の低下を防ぐ対応を、本章では「多機種対応」と呼ぶことにします。

19-2 多機種対応の考え方

多機種対応には、次の2つがあります。

- **快適に動作する機種を増やすための対応**
- **快適に動作しない機種にインストールさせない対応**

前者は、「ボタンが小さくて押しにくい機種がある」「画像がぼやけている」などといった問題を回避し、すばらしいユーザー体験を提供する端末を増やす対応です。後者は逆に、インストールを制限し、サポートする機種を絞る対応です。この対応が抜けていた場合、「お金を出してアプリを購入したのに、持っている端末では起動すらしなかった」といったことが起きる可能性が高くなります。

19-3 用語の定義

ここで、多機種対応の仕組みを使用する上で必要となる用語の定義を行います。定義する語は「スクリーンサイズ」「ピクセル密度」「dp」「sp」です。

最近の5インチ携帯電話は「large」に分類されそうな気がしますが、ピクセル密度が高いため「normal」に分類されます。この理由は後で説明します。

・スクリーンサイズ

物理的な画面のサイズです。対角線の長さで表し、単位としてインチを用います。Androidでは、「small」「normal」「large」「extra large」という4つのグループに分類されています。**表1**に、インチ数とグループの関係を示します。携帯電話の大部分は4〜5インチなので、まずは「normal」で正しく表示されることを目指しましょう。

グループ	インチ数
Small	2〜3.5インチ程度
Normal	3〜5インチ程度
Large	4〜7インチ程度
extra large	7〜10インチ程度

表1:インチ数とグループの関係

・ピクセル密度(density)

物理的な画面の中にピクセルがいくつ入っているかを表します。一般的には「dpi」(dot per inch)という単位で、1インチの中に何ピクセル入っているかで表します。Androidでは、「ldpi」「mdpi」「hdpi」「xhdpi」「xxhdpi」「xxxhdpi」というグループに分類されています。ピクセル密度の高い端末は、より精細な画像を表示することができます。

・dp(dip)

ピクセル密度に依存しない仮想的なピクセル単位(density-independent pixel)です。dp単位とピクセルの関係は、px = dp* (dpi / 160)で定義されています。このdpi / 160の部分とピクセル密度の関係を**表2**に示します。

ピクセル密度	dpi	dpi/160
Ldpi	120	0.75
Mdpi	160	1
Hdpi	240	1.5
Xhdpi	320	2
Xxhdpi	480	3
Xxxhdpi	640	4

表2:dpi/160とピクセル密度の関係

Androidでは、レイアウト時に長さの単位としてdpを用いると、実行時に端末のピクセル密度を用いて自動で適切なピクセル単位に変換してくれます。この機能により、ピクセル密度毎にレイアウトを作成する必要がなくなります。

・sp(Scale independent Pixels)

Android 4.0より、ユーザーは「設定」でフォントサイズを変更できるようになりました。

この単位を使用すると、ピクセル密度に加えてユーザーが設定したフォントサイズに応じてサイズが変化します。

19-4 ピクセル密度の違いに対応する

Androidアプリは、さまざまな画面サイズやピクセル密度の端末で実行されます。レイアウト作成時、長さの単位としてpx(ピクセル)を用いると、**図1**のように、ある機種では画面の幅いっぱいに表示されるのに、別の機種では画面の半分ほどのサイズで表示される、といった問題が発生します。

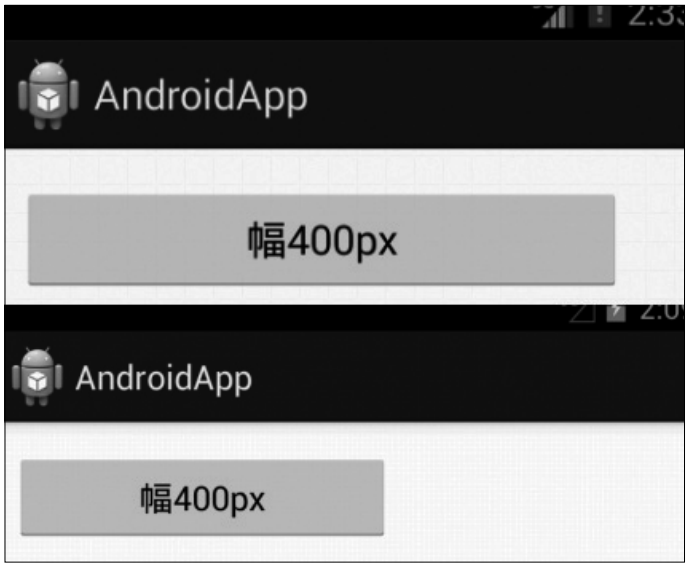


図1: ボタンの横幅をピクセルで指定した例

この問題を解決するために、Androidで提供されている仕組みを利用してピクセル密度に依存しないレイアウトを作成しましょう。Androidには、ピクセル密度に依存しないレイアウトを作成するための仕組みとして、次の4つが用意されています。

・match_parentやwrap_content
・dp単位
・sp単位
・ピクセル密度に応じた画像ファイルの切り替え

match_parentやwrap_contentを適切に使用する

Viewの幅を指定する「android:layout_width」や、高さを指定する「android:layout_height」では、単位付き数値(たとえば100pxなど)の他に、「match_

parent」（親Viewのサイズに合わせる）や、「wrap_content」（内部のコンテンツの大きさに合わせる。たとえば、TextViewであれば文字列の長さや高さ）が使用できます。この「match_parent」や「wrap_content」はピクセル密度を考慮にいれた上でサイズを決定するため、使用できる箇所では積極的に使用していきましょう。例えば、レイアウトのXMLを次のように書きます。

match_parentとwrap_contentを指定したレイアウト

```
<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/login"
    />
```

Javaプログラム中で動的にViewを生成し、レイアウトに追加する場合、引数を2つ取るaddView()を使用します。第2引数にはレイアウトに応じたLayoutParamsオブジェクトを渡します。このオブジェクトを作成する際、幅や高さにWRAP_CONTENT定数やMATCH_PARENT定数を使用することで、ピクセル密度に依存しないレイアウトを行うことができます。

Javaプログラム中でmatch_parentを指定してViewを追加する

```
private void addButton(RelativeLayout parent, String label) {
    Button button = new Button(this);
    button.setText(label);

    RelativeLayout.LayoutParams params = new RelativeLayout.LayoutParams(
            RelativeLayout.LayoutParams.MATCH_PARENT,
            RelativeLayout.LayoutParams.WRAP_CONTENT
    );
    parent.addView(button, params);
}
```

marginやpaddingの単位にdpを使用する

「android:layout_margin」や「android:padding」の単位にpxを使用すると、ピクセル密度の高い端末で間隔が狭く見えたり、逆にピクセル密度の低い端末で間隔が広すぎたりしてしまいます。この問題を解決するため、「android:layout_margin」や「android:padding」には単位としてdpを使用しましょう。

dp単位を指定したレイアウトXML

```
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="8dp">
    <!-- 省略 -->
</LinearLayout>
```

Javaプログラム中でmarginやpaddingを指定する場合、使用できるのはピクセル単位のみです。これでは端末のピクセル密度によって見た目が変化してしまうので、端末のdensity(**表2**のdpi/160の値)を取得し、px = dp * (dpi / 160)でピクセル数に変換しましょう。端末のdensityは「getResources().getDisplayMetrics().density」で取得できます。変換時、毎回計算式をプログラム中に記述すると読みにくくなってしまうので、次のように計算するメソッドを用意すると便利です。

「dpi」とは一般的に「ppi」(pixel per inch)と呼ぶこともあります。

dp単位からpx単位に変換するメソッド

```
/**
 * dp単位の長さを渡すと、px単位に変換してくれます
 * @param dp dp単位での長さ
 * @return px単位に変換した長さ
 */
private int dp2px(int dp) {
    // densityは端末の(dpi / 160)の値
    final float scale = getResources().getDisplayMetrics().density;
    // px = dp * (dpi / 160) = dp * scaleで求める
    return (int)(dp * scale);
}
```

Javaプログラムでpaddingを指定する

```
private LinearLayout createLayout() {
    LinearLayout layout = new LinearLayout(this);
    layout.setOrientation(LinearLayout.VERTICAL);

    // dp単位を元にピクセル数を求める
    int horizontalPadding = dp2px(8); // 8dp
    int verticalPadding = dp2px(16); // 16dp

    layout.setPadding(
        horizontalPadding, // 左padding
        verticalPadding,   // 上padding
        horizontalPadding, // 右padding
        verticalPadding);  // 下padding
    return layout;
    }
```

文字サイズの単位にspを使用する

sp単位の説明で紹介しましたが、Android 4.0以降では、ユーザーは文字の大きさを設定で変更できるようになりました。この設定を反映させるため、TextViewやButtonの「android:textSize」には単位としてspを使用します。dpやpxを単位として使用した場合、せっかくユーザーが設定した文字サイズが反映されず、結果としてユーザー体験の低下につながります。文字サイズの単位にspを使用して、ユーザーの設定を反映できるようにしましょう。

文字サイズにsp単位を指定したレイアウトXML

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="32sp"
    android:text="@string/title"/>
```

Javaプログラム中で文字の大きさを設定する場合は、TextViewやButtonの「setTextSize(int, float)」を使用します。第1引数には単位を指定します。sp単位であれば「TypedValue.COMPLEX_UNIT_SP」を指定します。第2引数には値を指定します。例えば、TextViewの文字サイズを24spにするには、次のように記述します。

Javaプログラムで文字サイズをsp単位で指定する

```
private TextView createTextView(String text) {
    TextView view = new TextView(this);
    view.setText(text);
    view.setTextSize(TypedValue.COMPLEX_UNIT_SP, 24);

    return view;
}
```

ピクセル密度に応じた画像を用意する

アプリ内で使用する画像は、「res/drawable-mdpi」フォルダーや、「res/drawable-xhdpi」フォルダーに入れておきます。「drawable-xhdpi」に入れた画像は、端末のピクセル密度がxhdpiの時に使用されます。

ここで、次のような状況を考えてみます。

1. res/drawable-mdpiフォルダーに48 x 48pxのicon.pngがある
2. res/drawable-xhdpiフォルダーにはicon.pngは無い
3. ImageViewのandroid:srcに@drawable/iconを指定する
4. アプリをピクセル密度xhdpiの端末で実行する

この場合、「res/drawable-xhdpi」フォルダーに「icon.png」ファイルがないので、代わりに「res/drawable-mdpi」フォルダーにある「icon.png」を2倍に拡大してImageViewに表示します。つまり、ピクセル密度の高い端末では画像が拡大されて、ぼやけてしまうことになります。このような事態を避けるため、ピクセル密度に応じた画像を用意し、適切なフォルダーに同名のファイルにして入れましょう。

ここで、「拡大すると画像がぼやけてしまうのなら、逆に『xxhdpi』向けの画像だけ用意すれば、後は縮小してくれるのでは?」と思うかもしれません。この場合、次のような理由により、ピクセル密度の低い端末でユーザー体験が低下してしまう恐れがあります。

携帯電話のスクリーンサイズの向上とともにピクセル密度も高くなっています。つまりピクセル密度の低い端末は、旧世代の端末であることが多く、(ピクセル密度の高い端末に比べ)処理能力が低い傾向にあります。処理能力に劣る端末で高解像度の画像を扱うと画像の縮小に時間がかかり、表示されるまでに時間がかかってしまうことになります。その結果、アプリケーションの動作が緩慢だと捉えられるかもしれません。このような挙動はエミュレーターでは実感しにくく、実際に実機で動かしてみるまで問題を認識できない可能性もあるでしょう。

19-5 スクリーンサイズの違いに対応する

スクリーンサイズには、そのグループごとに最小サイズが定義されています。**表3**にグループ毎の最小値を示します。例えばスクリーンサイズのグループがnormalであれば、470dp×320dpより大きいことが保証されます。

グループ	インチ数
small	426dp x 320dp
normal	470dp x 320dp
large	640dp x 480dp
xlarge	960dp x 720dp

表3: スクリーンサイズの最小値

たとえば、最近の5インチフルHD(Full High Definition)端末は、ピクセル数は1920×1080pxですが、ピクセル密度が「xxhdpi」なので、サイズは640dp×380dpとなり、「large」ではなく「normal」に分類されます。

こうして多くの携帯電話端末が「normal」に分類されることになりますが、ここで示したサイズはあくまで最小値です。実際に販売されている機種の画面サイズ(dp値)はこれより大きいことがほとんどです。スクリーンサイズが「normal」では、長辺は最低470dp以上となっていますが、たとえばGoogleのリファレンス端末であるNex

us 5の長辺は690dpです。

この違いにより、「手元の開発用端末では想定していたレイアウトになるが、長辺が短い端末だと下がはみ出てしまい、一番下にあるOKボタンが押せない」といった問題が発生します（**図2**）。

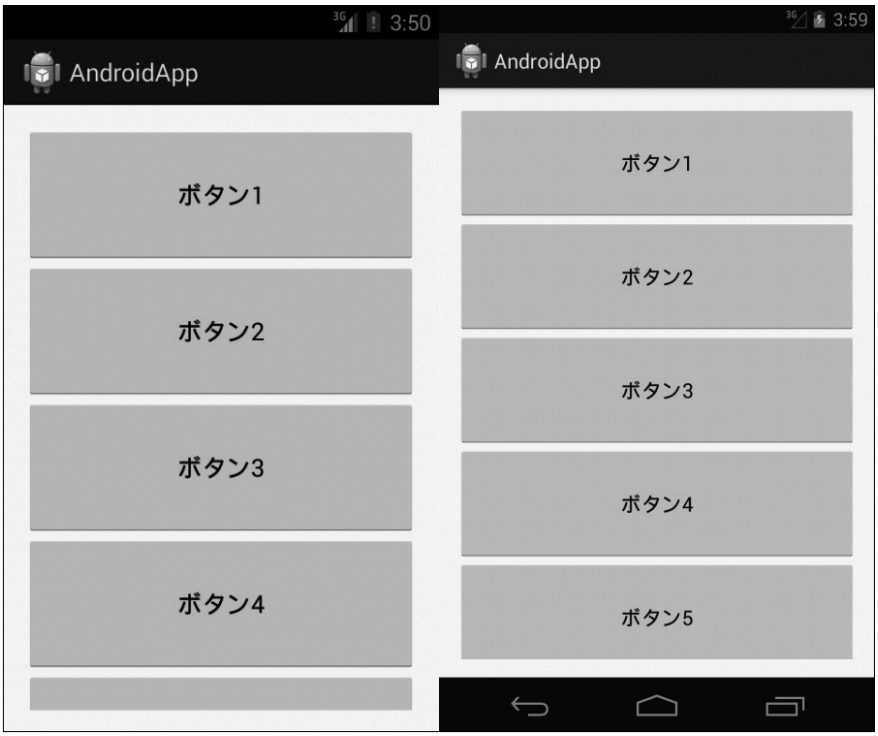


図2: ボタンが端末によってはみ出てしまう例

この問題を解決するため、レイアウト自体を470dp以内に収めるようにやり直すという手も考えられますが、さまざまな事情によりレイアウトを変更できないこともあるでしょう。その場合、レイアウト全体をScrollViewに入れましょう。長辺の短い機種では画面全体がスクロールするようになるため、はみ出てしまった部分が操作できないといった問題を回避できます。

19-6 タブレットに対応する

Androidアプリは、携帯電話だけでなくタブレットでも動作します。タブレットはスクリーンサイズが携帯電話に比べて大きいので、その大きさを活かしたレイアウトを提供しましょう。本節ではそれを実現するために、Androidで提供されている仕組みを2つ紹介します。

・7インチタブレット用や10インチタブレット用のレイアウトXMLを用意する
・marginやpaddingを@dimen/xxxxで指定する

タブレット用のレイアウトXMLを用意する

まず、タブレットでレイアウトそのものを変更する方法を紹介します。Androidには、「短い辺が500dp以上の端末の時は『res/layout-sw500dp』内のレイアウトを使用する」という仕組みが用意されています。多くの7インチタブレットは短辺が600dp以上なので、7インチタブレット用のレイアウトは「res/layout-sw600dp」フォルダに入れます。同様に10インチタブレットは短辺が720dp以上なので、10インチタブレット用のレイアウトXMLは、「res/layout-sw720dp」フォルダーに入れます。

横持ちに対応する

「res/layout-sw600dp」や「res/layout-sw720dp」が登場したので、縦持ちと横持ちで違うレイアウトを使用する方法も紹介します。この機能が必要となる場面として、縦持ちで上下に分割したレイアウトを作成した時などがあげられます。この時、上下に配置していたものを左右に配置した横持ち用のレイアウトを「res/layout-land」フォルダーに入れておくと、端末の向きに応じて自動的に使用するレイアウトXMLを切り替えてくれます。

marginやpaddingを@dimen/xxxxで指定する

次に、レイアウトはそのままで、余白を携帯電話とタブレットで変更する方法を紹介します。marginやpaddingはdp単位で指定しますが、タブレットの場合スクリーンサイズが大きいため、余白が思ったより少なく感じます。この問題を解決するため、Androidでは長さに名前を付け、レイアウトXMLでその名前をmarginやpaddingの値として指定することができます。

まず、「res/values」フォルダに「dimen.xml」ファイルを作成します。プロジェクトの作成方法によっては最初から作成されていることもあります。

「dimen.xml」は以下のようにします。これで、「text_vertical_marginという名前は8dpだよ」という定義を追加したことになります。

res/values/dimen.xml

```
<resources>
    <!-- デフォルトで作成された場合はこの2つは残しておきます -->
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>

    <!-- 次の行を追加 -->
    <dimen name="text_vertical_margin">8dp</dimen>
</resources>
```

「res/layout」内のレイアウトxmlでは、例えば「@dimen/text_vertical_margin」のように、「@dimen/xxxx」で長さの値として使用できます。

@dimen/xxxxを使用する例

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/text1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/title1"/>
    <TextView
        android:id="@+id/text2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/text1"
        android:layout_marginTop="@dimen/text_vertical_margin"
        android:text="@string/title2"/>
</RelativeLayout>
```

最後に、タブレット用の「dimen.xml」を「res/values-sw600dp」フォルダー内に作成します。ここでは、余白を4倍にしてみます。もし10インチタブレット用の設定を追加したい場合は、「dimen.xml」を「res/values-sw720dp」フォルダー内に作成します。

res/values-sw600dp/dimen.xml

```
<resources>
    <!-- デフォルトで作成された場合はこの2つは残しておきます -->
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">64dp</dimen>
    <dimen name="activity_vertical_margin">64dp</dimen>

    <!-- 次の行を追加 -->
    <dimen name="text_vertical_margin">32dp</dimen>
</resources>
```

19-7 OSバージョンの違いに対応する

AndroidはOSのバージョンアップに伴い、新しいAPIがどんどん追加されています。各APIは追加されたOSのバージョンに対応するAPI Levelが設定されています。

たとえば、API Level 8のAPIはAndroid 2.2で新たに追加されたAPIで、Android2.1以前の端末では使用することができません。

開発するAndroidアプリで使用できるAPIは、原則、サポートすると決めた最低のAPI Levelまでのものだけです。たとえば、サポートする最低のAPI Levelを10

(Android 2.3)と決めた上で、ArrayAdapterクラスのaddAllメソッドを呼ぶコードを書いたとします。このAPIは意外にもAPI Level 11で追加されたAPIなので、Android 2.3でこのメソッドを呼ぼうとすると、NoSuchMethodErrorが発生して、アプリはクラッシュしてしまいます。

しかし、「Android 2.3をサポートしたいが、Android 4.0以上でもよりよいユーザー体験を提供したい」という理由で、API Level 14以上のAPIを使用したい、といった状況も発生します。代表的な例はNotificationです。

これを実現するために、実行時に端末のAPI Levelを調べ、「高API LevelのAPIが使用できる時だけ使用する」といった処理を書くことができます。

例えば、API Level 14以上の時のみ、Notificationに大きなアイコンを設定するには、次のように記述します。

API Levelを調べて分岐する

```
private Notification createNotification(Bitmap bigIcon) {
    // API Level 14(IceCreamSandwich)前後で処理を分ける
    if (Build.VERSION.SDK_INT <
            Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
        Notification notification = new Notification();
        notification.tickerText = "通知";
        // 中略
        return notification;
    } else {
        return new Notification.Builder(this)
                .setTicker("通知")
                // 中略
                .setLargeIcon(bigIcon)
                .getNotification();
    }
}
```

19-8 インストール可能なOSのバージョンを制限する

「19-7 OSバージョンの違いに対応する」でも紹介しましたが、開発するAndroidアプリが使用するAPI Levelの最低値をあらかじめ指定する必要があります。ADTでは「AndroidManifest.xml」内の「android:minSdkVersion」で指定します。アプリをインストールしようとした際、端末はサポートしているAPI LevelとminSdkVersionの比較を行い、サポート外であればエラーを表示し、インストールを抑制します。なおGoogle Playでは、ある端末にインストールできないアプリは、その端末から見たアプリのリストから除外されます。

19-9 画面の向きを固定する

携帯電話向けアプリで「横持ちではすばらしいユーザー体験が提供できないので、画面の向きは縦持ちで固定したい」といったケースもあるでしょう。画面の向きを固定するには、「AndroidManifest.xml」のactivity要素に「android:screenOrientation」属性を追加します。

画面の向きを縦持ち固定にする

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name"
    android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

19-10 エミュレーターでテストする

開発したAndroidアプリが正しく多機種対応できているかを、アプリを動作させて確認しましょう。OSバージョンやピクセル密度の異なる端末を多数用意するのは費用、手間の面で現実的ではありません。Android SDK付属のエミュレーターを活用しましょう。

エミュレーターのシステムイメージをインストールする

古いOSのエミュレーターを作成するために、「Android SDK Manager」でOSバージョン毎にシステムイメージをインストールします。**図3**に、Android 4.0.3（API Level 15）のシステムイメージをインストールする例を示します。

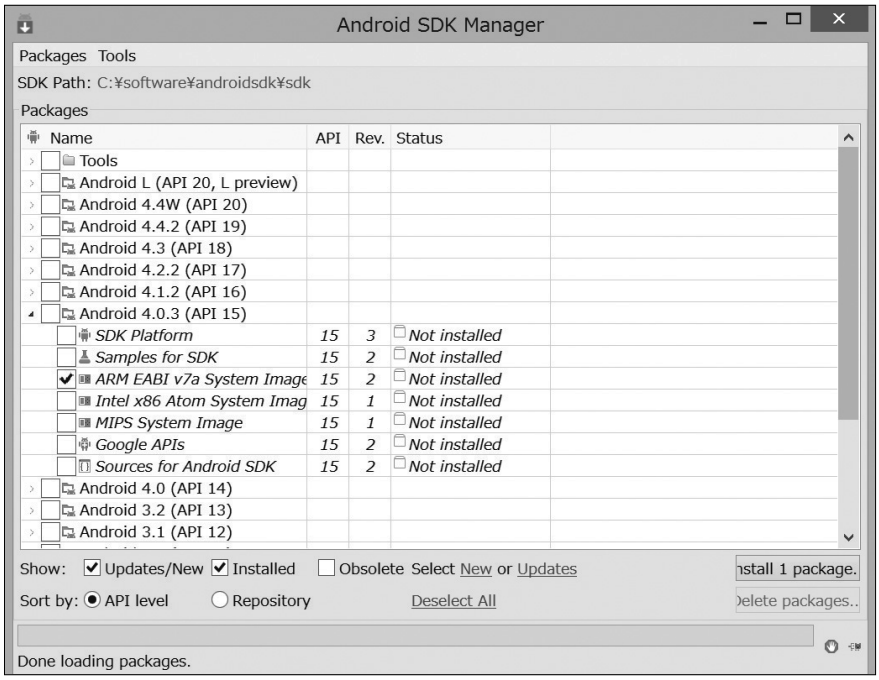


図3:システムイメージのインストール

エミュレーターのスクリーンサイズとピクセル密度を指定する

次に、テスト対象となるエミュレーターを作成しましょう。「Android Virtual Device Manager」を起動し、「Create」ボタンをクリックします。クリックすると、**図4**のようなダイアログが表示されるので、「Device」欄で端末の種類とピクセル密度を、「Skin」欄でスクリーンサイズを指定します。**表4**にSkinとピクセル数の関係を示します。

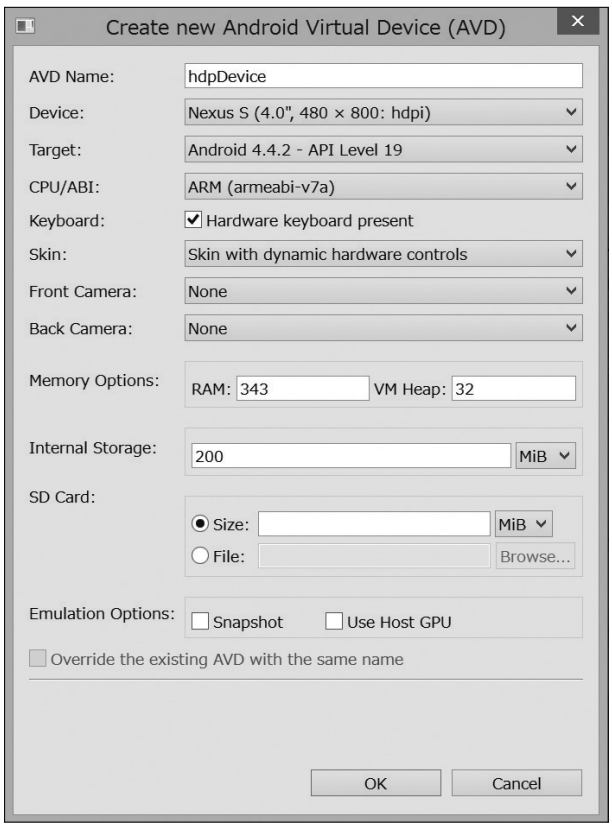


図4:エミュレーターの新規作成

Skin	ピクセル数
Skin with dynamic hardware controls	Deviceで指定したサイズ
No skin	Deviceで指定したサイズ
HVGA	320 x 480
QVGA	240 x 320
WQVGA400	240 x 400
WQVGA432	240 x 432
WSVGA	1024 x 600
WVGA800	480 x 800
WVGA854	480 x 854
WXGA720	720 x 1280
WXGA800	1280 x 800

表4: Skinとピクセル数の関係

ディスプレイ上にインチ数を指定してエミュレーターを起動する

エミュレーターの作成時にスクリーンサイズを指定しますが、指定したサイズをそのままPCのディスプレイで表示すると、場合によってははみ出てしまい、画面全体を表示できないことがあります。これは、エミュレーターで設定したピクセル密度と、PCディスプレイのピクセル密度が異なるためです。

この問題を解決するため、エミュレーターの起動時に、指定したインチ数でディスプレイに表示するための設定を行います。「Android Virtual Device Manager」で「Start」ボタンを押すと、**図5**のようなダイアログが表示されます。ここで「Scale display to real size」にチェックをつけ、インチ数とディスプレイのdpiを指定してエミュレーターを起動します。

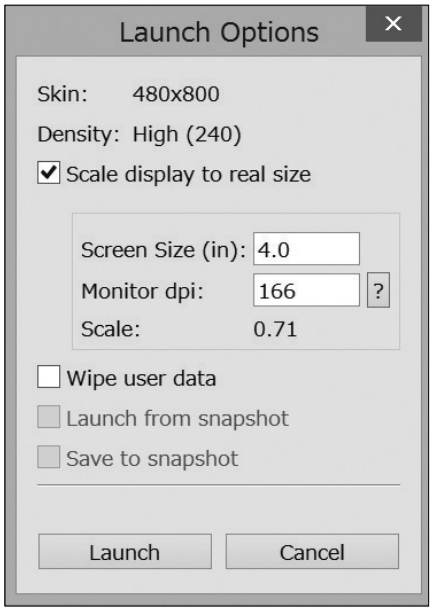


図5: インチ数を指定する

もしディスプレイのdpi値が分からない場合は、入力欄の右側にある「?」をクリックしましょう。ディスプレイのインチ数と解像度を指定するだけで、自動的にdpi値を計算してくれます（**図6**）。この設定により、たとえばディスプレイ上で4インチ（約10cm）のエミュレーターを起動することができます。

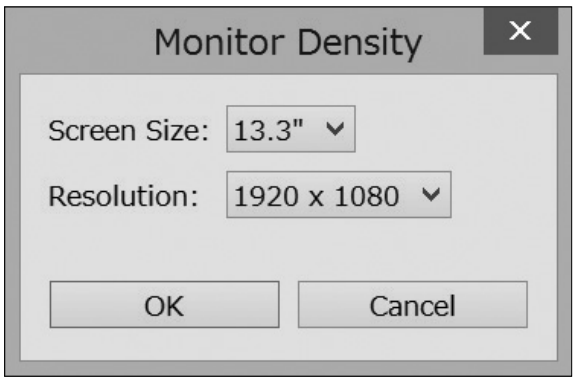


図6: ディスプレイのdpiを調べる

演習問題

図7のようなログイン画面を作成し、エミュレーターでスクリーンサイズやピクセル密度を変えて表示させてみましょう。

図7: 作成するログイン画面

まとめ

本章では、開発したアプリをユーザー体験を低下させずに、より多くの端末に提供するための手法を紹介しました。すべての端末をサポートするのは不可能なので、「Android 4.0以上で、縦持ちのみサポートする」のようにサポートする範囲を決めて、その中でよりよいユーザー体験を提供するよう心がけましょう。