第15章

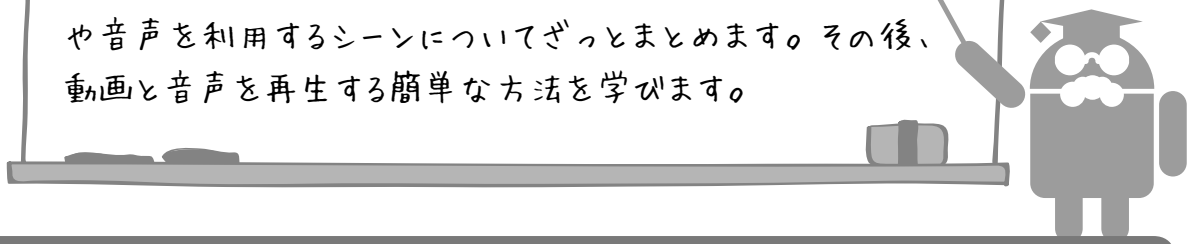
マルチメディア（AV）

著：中澤 慧

15-1 マルチメディアの利用方法

著：中澤 慧

マルチメディアAPIを利用することで、アプリの表現力を高めることができます。本節では、まずAndroidアプリ内で動画や音声を利用するシーンについてざっとまとめます。その後、動画と音声を再生する簡単な方法を学びます。



この節で学ぶこと

- アプリ内で映像、音声を利用する場面
- 動画データの基本構造
- 動画の再生に必要な処理
- 端末上に存在するムービーファイルを再生する方法

この節で出てくるキーワード一覧

MP4
H.264
AAC
MP3
ストリーミング
コマ落ち
MPEG-DASH
エンコード／デコード
mux/demux

この章に出てくるクラス

VideoView

15-1-1　アプリ内での代表的な映像 / 音声利用場面

はじめに、動画や音声というコンテンツが持つ特性を把握し、それを適切に、かつ効果的に利用するための方策を考えていきます。

まず、それなりの品質を確保した動画は大容量になりがちです。映画であれば2時間のDVDでも、データ量は4GBを超えてしまいます。アプリ中で動画を利用するのに、事前のロード時間が何時間にもなるとしたら、きっと多くの人はアプリを使うのをやめてしまうでしょう。基本的に、データ容量、読み込み方法、再生負荷という3点のバランスをうまく取る必要があります。

これらを考える材料として、Androidアプリの中で動画を利用する場面と、それぞれに求められる要件をざっとまとめてみました。場面ごとに重視すべきポイントと犠牲にしても良いポイントがあります。また、アプリの利用者に届けたい価値、つまりコンテンツの特性に大きく依存する部分もあります。

アプリの主役としての映像／音声コンテンツ

・映画やドラマといった動画コンテンツを配信する場合

映画配信のように動画コンテンツ自体がアプリの主役となる場合を考えてみましょう。この場合、可能な限り高い品質が求められるため、データ容量は大きくなりがちです。アプリ内にあらかじめ大容量データを格納して配信するよりは、アプリの実行中にネットワーク越しに動画データを読み込んで再生する方法が主流となります。

ネットワーク環境があまりよくない場合には、映像や音声の品質を下げてでもなるべく途中で停止することなく再生できることが求められます。とりわけ、YouTubeやVine、Facebookなど、比較的短時間の動画をインスタントに閲覧できることをウリとするアプリにおいては、この傾向が強くなっています。

発展的機能として、早送りや区間ループといった機能を作りこんで提供するもの、映像や音声にフィルタをかけて出力するものもあります。

・ネットワーク越しにストリーミング再生するUstreamやYouTube Liveといったリアルタイム配信コンテンツ

前述の映画やドラマと比較すると、映像コンテンツ自体がアプリの主役となる点は同じですが、ネットワーク越しの配信をよりスムーズに受けられることが重要という点が異なります。

このようなアプリにおいても、ネットワーク環境の影響を受けにくいアプリ作りが重要です。特に、映像部分よりも音声部分の配信を安定化することで、データ転送速度の遅い環境でも、比較的体感の良いコンテンツを提供することが可能となります。

バッファリングとは、ネットワーク転送速度が安定しない環境でリアルタイム配信を受ける場合、転送速度がいくら速くても配信タイミングよりも先を読み込むことができない一方、速度が低下したタイミングでは容易に再生データが不足して画面が停止する、といったことが発生します。これを軽減するために、再生する動画をある程度メモリ上に貯めこんでから再生をおこなう仕組みをバッファリングと呼びます。バッファ容量を大きく持つと動画再生を安定させられる反面、実時間からの遅延が大きくなるため、一概にバッファが大きなほうが良いとは言えません。

・ネットワークごしにストリーミング再生するインターネットラジオ系のもの

前述のリアルタイム配信動画よりも更に音声への要件が強いものです。

配信する内容によっては、音声配信のフォーマットを工夫することでデータ量をより低減できる場合もあります。データ転送速度の安定しないネットワーク環境に対する安定化策としてバッファリングは有効です。ただしその度合いは、聴取者に届くまでの遅延と、その許容範囲を考慮して調整する必要があります。例えば、リアルタイムの聴取者参加型クイズといったコンテンツを配信しようとする場合、音声の品質を犠牲にしてでも、バッファ時間を短めにおさえる必要があるでしょう。

・ 映像／音声コンテンツを組み合わせて表現するものや、映像制作／作曲をおこなうもの

DJやVJ、DTMといった領域です。USB経由で鍵盤を接続して作曲するアプリなどもここに含まれます。

Androidは、OSのサポートするアプリ間連携機能の中で、サウンド出力連携機能が弱いため、これらのアプリはiOSと比較するとあまり盛り上がっていません。

この場合、リアルタイムにデータをストリーミング再生するような用途はさほどありません。多くの場合、アプリ内にあらかじめ含まれている音源を組み合わせたり、事前にダウンロードしたデータを組み合わせたりします。これは、編集後の出力時に映像や音の途切れが発生したりすると、ユーザー体験が致命的に損なわれるためと考えられます。

・ 映像や音声を中軸とするゲーム

いわゆるリズムアクションゲームなど、ゲームに占める楽曲や効果音のウェイトが大きなものがこれにあたります。

前述のDJやVJと同様、プレイ途中でのフレーム落ちや再バッファリングの発生は著しくユーザー体験を損なうため、データは事前にダウンロードする形式が主流です。また、データ量が多くなりがちなので、データをダウンロードする方法に工夫が見られる場合があります。

アプリの引き立て要素としての映像／音声

映像／音声コンテンツ自体を主役とするのではなく、他の主体となるコンテンツを補足したり世界観への没入感を高めるための道具として利用する場合です。

公共空間での音楽再生は、基本的にイヤフォンやヘッドフォンを着用することがほとんどで、スピーカーなどから音声出力をすることはハードルが高くなります。本書を読んでいる方の中でも、スマホの音設定は、自宅内を含めて常時マナーモードにしている方が多いでしょう。このため、アラームやリラックスサウンド集、一部のジョークア

プリなどを除いて、非ゲームのアプリ内では効果音やBGMは多用されません。

そのような中でも、映像や音声を効果的に利用できる箇所はいくつかあります。以下に挙げてみましょう。

・アプリ固有の通知効果音

LINEなどのメッセージングアプリは、他のアプリとは違う通知効果音を持っています。単純に端末のディスプレイをオフにした状態でもメッセージ着信をユーザーに知らせられるという以上に、独自の通知音とアプリをユーザーの脳内で関連付けてもらうという意味もあると考えられます。

通常はアプリ1つにつき1種類かつ1秒程度の効果音しか利用されないため、容量問題にはつながりにくいものです。

・ゲームの効果音やBGM

通常、アプリのダウンロードが完了して実行を開始した時点で、データが揃っていることが求められるものです。ゲームの規模によりますが、アプリ容量の圧迫原因となり得るため、なるべくデータ容量を削減することが求められます。

・ゲームのオープニング動画や演出動画

これも通常、アプリのダウンロードが完了して実行を開始した時点でデータが揃っていることが求められるものです。しかし、演出動画については、その種類や長さ、品質によってはアプリ起動後に追加データのダウンロードが必要になることも考えられます。

演出動画とは、プレイヤーがゲームの世界観に没入しやすいよう、ごく短い動画群を制作し、ポイントを絞って利用することがあります。これらを演出動画と呼びます。

15-1-2 動画データの基本構造

Androidでの動画の扱いを説明する前に、動画の構造に関する基礎知識をいくつか押さえておきましょう。そうすることで、各機能の持つ意味をより良く理解できるようになります。

動画の構成要素と構造を把握する

動画には映像と音声が含まれる場合がほとんどです。英語圏で制作されて日本で展開される映画やドラマといったコンテンツでは、英語と日本語の音声が含まれる場合も多いでしょう。また、字幕用のテキストデータが含まれる場合もあります。

このように、動画はある時間軸に従って同期を取って再生されることを想定する複数のストリームを集約したもの、と捉えることができます。複数のストリームを集約して扱いやすいようにひとつにまとめることを多重化=multiplex（略してmux）すると言

います。

多重化の必要性は、先頭から連続してしかデータを取り出せないストリームを考えると分かりやすいでしょう。ある動画を読み込んで再生するために最初に映像を全部ロードし、次に音声を全部ロードする、という手順が必要な場合、音声のロードが終わるまで映像も再生できないことになります。それでは、近年の動画ストリーミング再生のように、データの読み込みを始めてからすぐに再生を開始したいという要求にはマッチしません。

なお、動画中のストリームを多重化せずに、独立した複数のストリームのまま扱う方法もあります。後のコラムで簡単に紹介するMPEG-DASHがそれです。これは、複数の品質で用意された独立した映像ストリームと、同じく独立して用意された音声ストリームを再生側で適宜選択することで、可能な限り良い品質の映像を再生できる仕組みです。

さて、前述のように動画は複数のストリームから成ることがほとんどです。ある動画ファイルの中で映像部分はどこか、音声部分はどこか、といったことが再生ソフトウェアにとって分かりやすくなっている必要があります。また、例えば動画を途中から再生したい場合、動画ファイルの先頭からデータを読み直す必要がある構造では、ネットワーク越しのストリーミング再生時に不便です。

このように、配信側の都合と再生側の都合をうまく満足できるような動画形式（動画コンテナと呼ばれます）の必要性から、様々なものが開発されてきました。近年は、特にネットワーク越しにデータを読み込みながらスムーズに再生でき、途中でデータの破損があったとしてもなるべくその影響を抑えられるようなコンテナ形式が多く使われます。

なお、動画コンテナから映像ストリームと音声ストリームを適切に取り出すことをde-multiplex（略してdemux）と呼びます。

動画の要素としての映像／音声を把握する

前節では動画を構成する要素として映像・音声・テキスト字幕などを挙げました。この節では、映像・音声について、もう少し詳しくみていきます。

一般に、映像を表現するためのデータ量は膨大です。ここでは、いわゆるフルHD（1920×1080）解像度の映像を考えてみましょう。

フルHDの映像で、各ピクセルが赤、緑、青についてそれぞれ8bitのデータを持つと仮定すると、秒間30フレームの映像が持つデータ量は1秒間あたり180MB以上となります。このデータをそのまま記録するには、1時間あたり650GBものストレージが必要となります。いくら各種ストレージデバイスの価格が下がってきたとはいえ、この量を保持するのは現実的と言えません。

このため、様々な方法でデータ量を削減する工夫がおこなわれてきました。代表

的な方法としては、以下ものがあります。

- **人の目で区別しにくい色のデータを省略して知覚上問題ない範囲でデータを削る**
- **映像のあるフレームと前フレームとの差分を抽出してその分のみを記録する**
- **映像内のあるシーンにて一定の速度で移動する物体を検出して、その物体については移動速度情報のみを記録して実体分のデータを削る**

このような処理を加えたデータを生成することを「エンコード」と呼びます。近年動画配信などで多く利用されるH.264形式も、これらのデータ量削減策を駆使したエンコード形式です。なお、ここでは動画の話をしていますが、動画に限らず静止画やテキストデータでも、元データの容量を削減したり自身の用途に適合するように形を変えることをエンコードと呼びます。例えば「ビットマップデータをJPEGやPNGにエンコードする」といった表現も使われます。また、複数のファイルをひとつにまとめ、可能な限り容量を抑えるようにzipファイルを作成することもエンコードと言えます。

エンコードされたデータは、当然そのままでは画面表示などに使うことができません。このため、そのデータを利用する側では、使う側にとって都合の良い形に変換する必要があります。これをデコードと呼びます。デコードする際には、エンコードされたデータに合わせた方法を選ぶ必要があります。デコード手順が複雑なフォーマットは、スマートフォンでの再生に向かない場合もあります。

音声に関しても、映像と同様、そのままではデータ量が大きすぎるので、それをうまく削減するための形式が複数存在します。よく利用されている「AAC」や「MP3」という名前を聞いたことがあるかもしれません。

15-1-3 動画の再生に必要な処理の概略

動画コンテナと映像・音声フォーマットについての理解が進んだところで、Androidにおける動画再生を学んでいきましょう。ここではMP4コンテナに格納された「H.264」+「AAC」の動画を再生し、映像が画面に表示され、音声が端末のスピーカーに流れるまでを考えてみます。

まず、コンテナごとデータを読み出します。SDカードなどのローカルストレージに保存されているデータはファイルシステムAPIを利用して読み込みます。ネットワーク越しに存在するデータはHTTPなどの方法で読み込みます。

次に、コンテナ内の映像と音声のストリームを分離します。既に述べたように、ファイル内の適切な位置から映像ストリームと音声ストリームを抽出します。コンテナデータのレベルで破損があった場合は、適切に破損部分をスキップした上で正常な

データを抽出します。

そして、映像をデコードしてフレームを取り出します。H.264の映像ストリームを、動画再生ソフトウェア内に含まれる機能またはOSの提供する機能を使ってデコードします。これにより、ディスプレイへ出力できる形式のフレームを生成します。

さらに、音声をデコードしてサンプルを取り出します。AACの音声ストリームを、動画再生ソフトウェア内に含まれる機能またはOSの提供する機能を使ってデコードします。これにより、サウンドチップに渡せるPCM形式のデータ列を作成します。

近代的なOSの上では、映像や音声を出力するアプリが複数動作することもあります。例えば、複数のアプリが同時に音を出す場合、何も考えずに両者の出力結果を足しあわせてサウンドチップへ流すと、音割れの原因になったりします。

このような問題を避けつつ、同時出力できるように音声を集約する処理をミキシングと呼びます。映像に関しても、例えばタスク一覧画面など、アプリが全画面表示されない場面でも再生中の動画を適切に表示するなどの処理がおこなわれます。これらは、いずれもOSの仕事です。

最終的に、ディスプレイドライバーを介して、映像を液晶ディスプレイなどに表示し、サウンドドライバーを介して、音声をスピーカーに出力します。

以上の構成、流れを**図1**に示します。

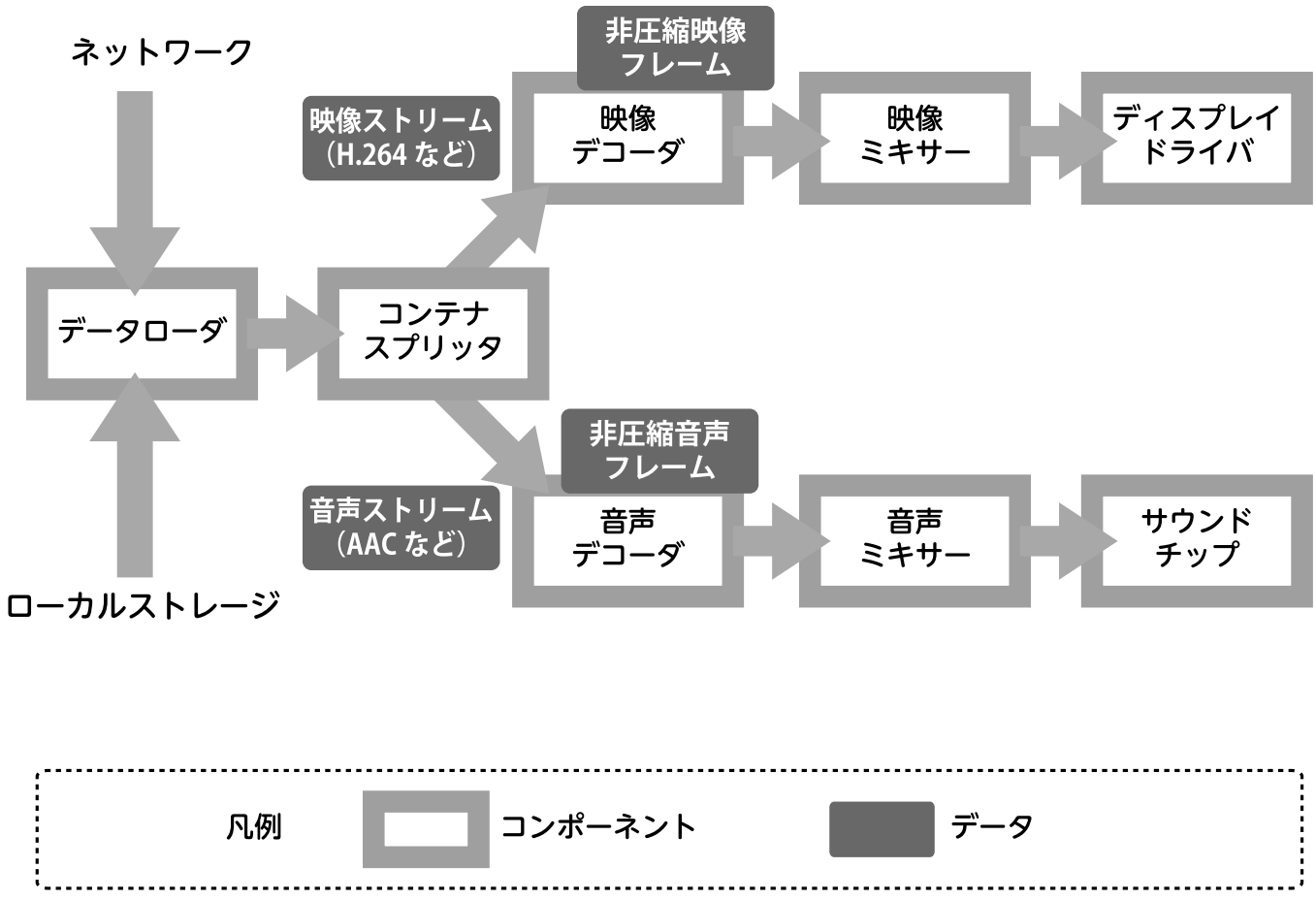


図1:マルチメディア再生データーフロー

映像出力と音声出力のタイミングがずれていると、動画の閲覧者は違和感をおぼえます。個人差はありますが、多くの場合3〜4フレームのずれが生じると違和感を持つとされます。特に、音声の出力にスピーカーではなくBluetooth接続のヘッドセットなどを利用すると、音声出力までの遅延は大きくなります。

利用する動画フォーマットによってはデコードにかかるCPU負荷が高すぎる場合もあります。一定範囲までは端末のCPUでリアルタイムに処理できますが、あまりにも負荷が大きなものは、フレームのデコードをスキップすることでリアルタイム性を確保するように処理されることも多くなっています。これをコマ落ちと呼びます。映像の動きが少ない場合は目立たないこともありますが、動きの激しいシーンで発生すると違和感を持ちやすいでしょう。

このように、快適な動画再生を実現するためには、解決するにあたって難易度の高い問題が多くあります。動画再生をおこなうアプリの開発者が皆このような問題に取り組むのは非常にハードルの高いものと言えます。

Android上で、これらを思いっきりまとめて扱えるクラスが「VideoView」です。今回は、「VideoView」を使って簡単な動画再生を実現します。Android内部での詳細な処理コンポーネントについては、本章の後半で紹介します。

15-1-4 端末上に存在するムービーファイルを再生する

端末上のムービーファイルを再生する簡単なアプリを作ってみましょう。ここでは、アプリのパッケージ内に添付データとして含まれる動画を再生する、極力簡単な例を紹介します。

ディレクトリー一覧から指定のファイルを選択したり、ネットワーク上からストリーミング再生したり、といった複雑なことはおこないません。これにより、先に挙げた再生までの複雑な道のりを、Androidの機能で見事にショートカットできることがわかるでしょう。実用上も、ゲームのオープニング動画再生や、アプリ内で使い方を説明する動画再生程度なら、これで十分なケースも少なくありません。

プロジェクトを作成する

ここではSDKバージョン指定を4.4（KitKat）に揃えてプロジェクトを作成します（**図2**）。新規Activityの作成設定画面では、「Blank Activity」を選択してください。

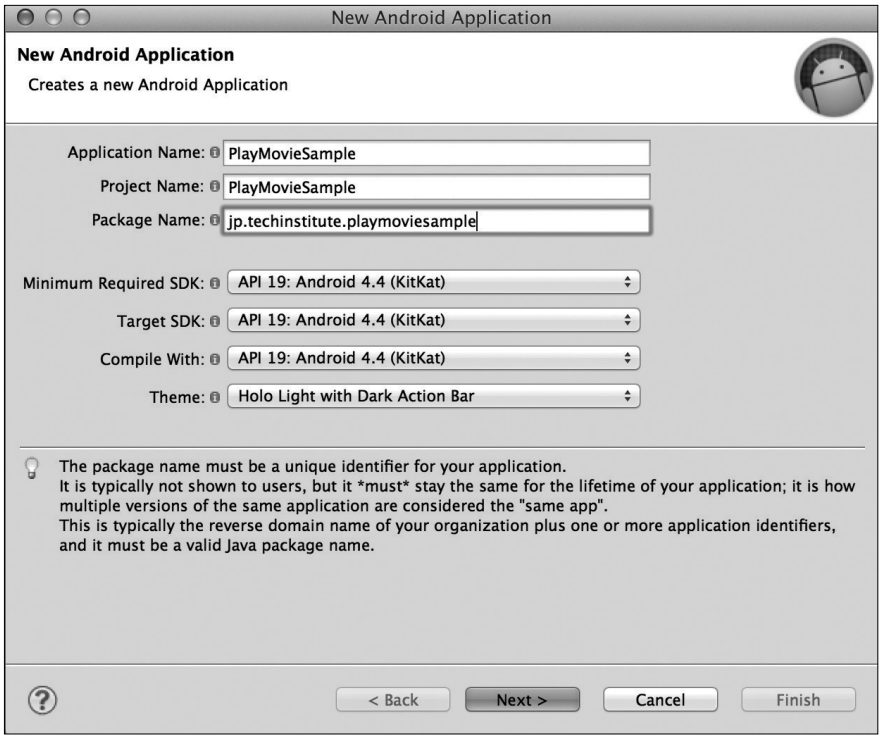


図2: プロジェクトの新規作成

次に、再生ボタンと動画表示用のビューのみを持つ単純なレイアウトを作成します（**リスト1**）。

リスト1:activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="${relativePackage}.${activityClass}" >

    <LinearLayout android:id="@+id/controlPanel" android:orientation="horizontal"
        android:layout_width="match_parent" android:layout_height="40dp" android:gravity="cent
er_horizontal"
        android:layout_margin="4dp">
        <Button android:id="@+id/playBtn"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="Play" />
    </LinearLayout>
    <VideoView android:id="@+id/videoView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@+id/controlPanel" />

</RelativeLayout>
```

必ずしもリストの通りに記述する必要はありません。あまり1行が長くなりすぎないように、適宜改行を入れてもかまいません。再生以外のボタンも適宜配置できるように、ボタンは「LinearLayout」内に置いています。「VideoView」と書かれているの

が、非常に簡単な手続きで動画を再生できるコンポーネントです。

次に、再生用の動画ファイルを「res/raw」ディレクトリー内に格納します。このとき、ファイル名は「samplemovie.mov」としましょう（**図3**）。

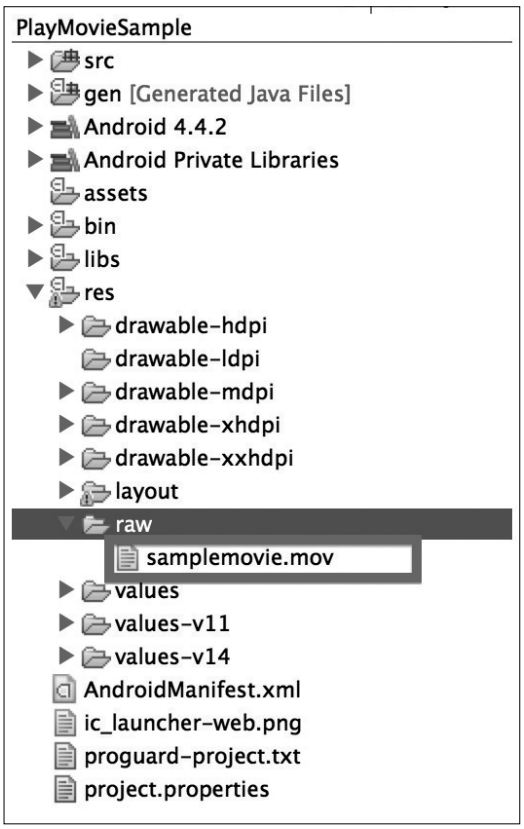


図3:再生する動画ファイルをプロジェクトに追加する

今回は動画再生をランドスケープモードでのみ扱うことを想定しています。このため、「AndroidManifest.xml」のActivity部分に、画面の向きを固定するため、以下のような記述を追加してください（**リスト2**）。

リスト2:AndroidManife

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="jp.techinstitute.playmoviesample"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="19"
        android:targetSdkVersion="19" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:screenOrientation="landscape"
            android:configChanges="keyboardHidden|orientation|screenSize" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

ここまでで準備はだいたい完了しました。仕上げにJavaのコードを書いていきましょう(**リスト3**)。

リスト3:MainActivity.java

```
package jp.techinstitute.playmoviesample;

import android.app.Activity;
(以下import文省略)

public class MainActivity extends Activity {

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                Button playBtn = (Button)findViewById(R.id.playBtn);
                playBtn.setOnClickListener(new OnClickListener() {


                        @Override
                        public void onClick(View arg0) {
                                VideoView videoView = (VideoView)findViewById(R.id.videoView);
                                String movieUri = "android.resource://" + getPackageName() + "/" +
R.raw.samplemovie;                                                                              ❶
                                videoView.setVideoURI(Uri.parse(movieUri));
                                videoView.start();
                        }
                });
        }

}
```

パッケージのインポートでエラーが出る部分は、普段の通りEclipseのインポート補助機能を利用して適宜解消してください。今回の動画再生に特有のコードは、❶の赤枠で囲った4行のみです。XMLでのレイアウト上に存在する「VideoView」コンポーネントを取得し、setVideoURIメソッドで再生対象動画を指定し、startメソッドで再生を開始しています。

ここまで書けたら、アプリを実機に転送して動作を確認してみましょう。

演習問題

再生ボタン以外に一時停止ボタンを用意してみましょう。VideoViewコンポーネントのpauseメソッドがヒントです。

一時停止ボタンに加えて、停止ボタンを用意してみましょう。停止の場合には次の再生時に動画の先頭から再生が始まるようにするのが一般的です。

まとめ

Androidアプリ開発に限らず、動画やサウンドを利用する上で身につけておくべき基礎知識を紹介しました。

加えて、Androidアプリ内でなるべく少ないコードを用いて動画を再生する方法を紹介しました。

check!

凝ったムービー再生

本節ではアプリ内にあらかじめ組み込まれた動画の再生を扱いました。それに対して、オンラインの動画を自由自在に再生したい場合には、知るべきことや書くべきコードの性質が変わってきます。また、前の章で学んだネットワークや認証の知識も関連してきます。ネットワーク越しのデータ読み込みは速度が安定しない場合もしばしばあります。これを見越して適切なバッファリング処理をおこなったりする必要があります。

動画の読み込み元についても考える必要があります。Web上で公開されている動画でも、アプリから自由に再生して良いというわけではない場合があります。例えば、YouTubeはアプリ向けのSDKを公開しているので、これを利用します（「https://developers.google.com/youtube/android/player/」）。

動画の容量は大きくなりがちです。ネットワーク経由の配信サービスをおこなっているサービス主体は、その配信コストを負担しているので、コンテンツに対するタダ乗りのような挙動をするアプリを好まない傾向にあります。加えて、動画は違法アップロードやダウンロードといった問題と関わりがちのエリアでもあります。自身のアプリをユーザーにどのように使って欲しいか、という点だけでなく、そのアプリがおこなうことはフェアであるか、適法であるか、といったことにも気を配ることが重要です。

Android上での動画再生については、既に紹介した「VideoView」クラスを利用する以外の方法もあります。2014年のGoogle I/Oにて、「ExoPlayer」というプロジェクトが発表されました。これは、Android標準の「VideoView」よりも多くの動画コンテナをサポートしつつ、ネットワーク越しのストリーミングにおけるパフォーマンスも改善するソフトウェアコンポーネントを提供するという意欲的なものです。きめ細かなAPIを提供しており、動画再生に関わる多くの部分を「デフォルトのまま利用したい箇所はデフォルトの実装を使いつつ、こだわりたい箇所のみアプリの開発者が手を入れる」という形にしてくれます。「ExoPlayer」のプロジェクトには「https://github.com/google/ExoPlayer/」にてアクセスできます（プロジェクト自体の紹介は「http://developer.android.com/guide/topics/media/exoplayer.html」にあります）。

この中では、ネットワークの転送速度に応じて動的に再生する動画の品質を切り替える仕様である「MPEG-DASH」をサポートしています。この制御を自前のソフトウェアできちんとおこなうのは大変なのですが、アプリ開発者が組み込み可能なコンポーネントとして提供されることで、導入のハードルが下がります。動画再生機能にこだわりたいアプリにとって、「ExoPlayer」を利用することにはいくつものメリットがあります。

このように期待のプロジェクトなのですが、一方でまだ発展途上でもあります。機能が動作せずにアプリがクラッシュする端末も多い状態です。現時点で「ExoPlayer」を使いこなして実用的なアプリを開発するには、おかしな動作へ行き当たった際に自らライブラリのソースコードをチェックするような覚悟が必要でしょう。

check!

マルチメディアフォーマットの色々

本文で扱った動画コンテナは「MP4」のみですが、それ以外にも多くのものがあります。例えば、「MPEG2-PS」「 MPEG2-TS」「AVI, Matroska」「Ogg」などです。

動画コンテナは、基本的にOSや利用コンポーネントが対応するものの中から選ぶことになります。この中で、ネットワーク越しのストリーミングに強い形式か否か、テキスト字幕などの特殊ストリームを配置しやすい形式か否か、などの軸で評価して決定します。

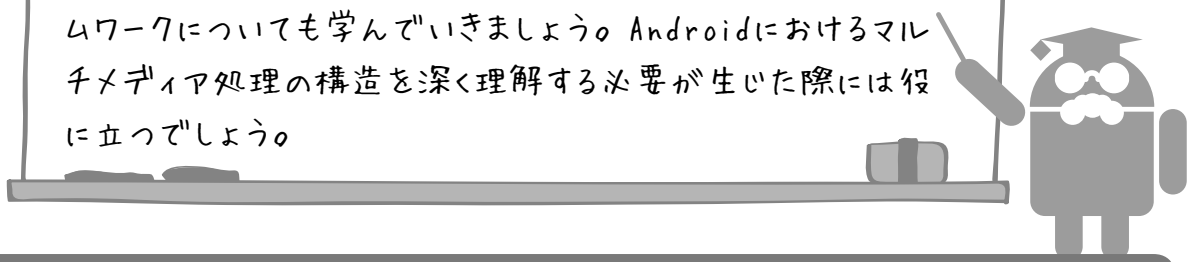
動画コンテナの中に含まれる映像・音声の形式も様々です。映像では「H.263」「H.264」「AVC」「VP8」「WebP」などが、音声では「AAC」「MP3」「Ogg Vorbis」「FLAC」などが利用されます。変わり種として、人の話し声に特化した省容量形式として「Speex」があります。例えばインターネットラジオのトーク部分などに利用すると、他の形式と比較して省容量でも聞き取りやすい結果を得られる傾向にあります。

Android端末で広くサポートされるフォーマットはAndroidのデベロッパサイトに掲載されています（「http://developer.android.com/guide/appendix/media-formats.html」）。参考にしてください。

15-2 Androidのマルチメディアフレームワーク解説

著：中澤 慧

本章の前半で、一般的な動画再生のためのフローを学びました。これをより深める形で、Androidのマルチメディアフレームワークについても学んでいきましょう。Androidにおけるマルチメディア処理の構造を深く理解する必要が生じた際には役に立つでしょう。



この節で学ぶこと

・Androidのマルチメディアフレームワークの概要
・マルチメディアAPIを使ったアプリの作成方法

この節で出てくるキーワード一覧

MediaPlayer
Stagefright
MediaCodec
コーデック
DataSource
MediaExtractor
OpenMAX-IL
SurfaceFlinger
AudioFlinger

この章に出てくるクラス

SurfaceView
SoundPool
MediaPlayer
SurfaceHolder

15-2-1 Androidのマルチメディアフレームワーク概要

本節は、読者が動画データの基本構造を把握している前提で書いています。この部分の理解に不安がある場合は、本章の前半に戻って読み返してみてください。

なお、Androidのマルチメディアフレームワークには「DRM（デジタル権利管理）」の機能が含まれますが、これを含めると説明範囲が広くなりすぎるため、今回は扱いません。

前の節で演習に利用した「VideoView」は、動画再生に必要な多くの手間を省略して簡単に扱えるようにしたものです。この中で利用されているのが「MediaPlayer」というクラスです。このクラスは、動画／音声ストリームのdemuxに始まり、「H.264」や「AAC」といった圧縮形式のデコード、画面やスピーカーへの出力までをおこなっています。これをブラックボックスとして扱うこともできますが、より深く知っていることで便利な局面もあります。本節では、Javaの世界から扱える「MediaPlayer」の裏側に迫ってみます。

Androidのメディアフレームワークのコンポーネント概略を**図4**に示します。

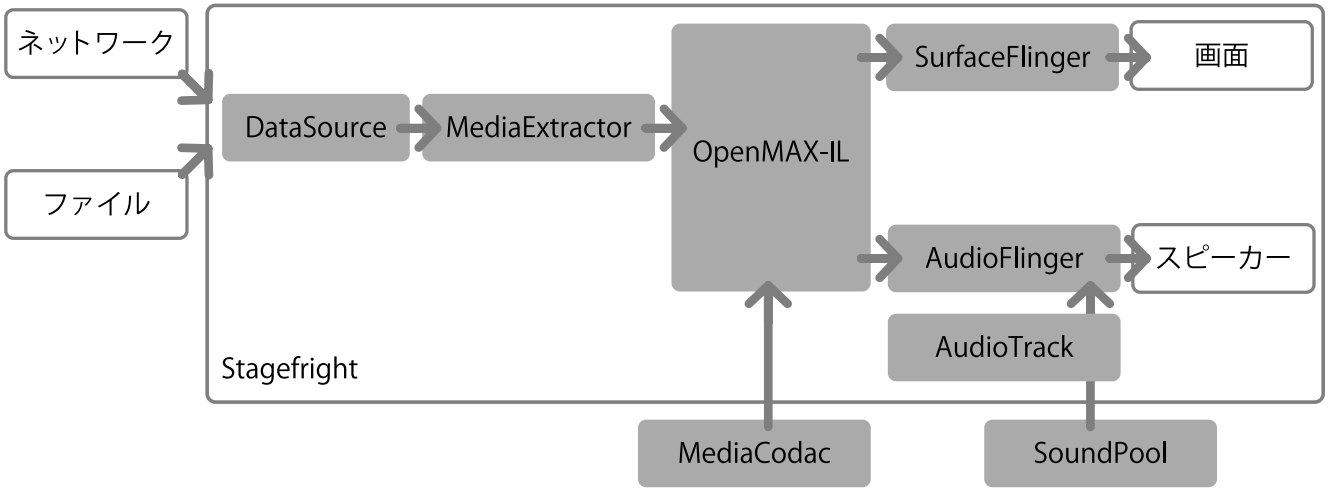


図4: Stagefright周辺の構造図

以下、この中に含まれるコンポーネントについて、順に説明します。

・Stagefright

「MediaPlayer」の本体が、「Stagefright」と呼ばれるソフトウェアです。これはAndroid 2.2から利用されているもので、Javaの世界から呼び出されたメディア再生の大半をC++のネイティブ側の世界で処理してくれます。画面への映像出力やサウンドドライバーへの音声出力までの面倒を見てくれる、「MediaPlayer」の肝となる部分です。

これ以下のコンポーネントは、「Stagefright」の中心となってメディア処理をおこ

なうものです。更なる詳細を知りたい場合は、この節の最後に参考URLを記載していますので参照ください。

・MediaCodec

コーデック（codec）は、エンコーダ・デコーダをまとめて扱う用語です。メディアコーデックは、マルチメディアデータをエンコードしたりデコードしたりする機能を持つソフトウェアやハードウェアを意味します。

Javaの世界から、後述する「OpenMAX」のコーデックを利用して映像や音声のデコードをおこなう窓口にあたるクラスが「MediaCodec」です。

・DataSource

端末上のローカルファイルや、HTTP経由などで動画のデータを読み出してくる役割を持ちます。ここで読み出されたデータは、後述する「MediaExtractor」に渡されて利用されます。

・MediaExtractor

メディアのストリームをファイルなどのストリーム内から抽出する役割を持ちます。前節で述べた、多重化されたストリームのdemuxをおこないます。Androidの標準機能としては、例えば「MPEG4」のストリームの内部から、「H.264」映像ストリームなどを抽出する機能を持ちます。

他にも、「MP3」のファイルストリームから「MP3」のメディアストリームを抽出するような機能も持っています。

・OpenMAX-IL

映像と音声のエンコードやデコードをおこなうための枠組みです。「H.264」や「AAC」のデータ列を渡すと、適宜映像フレームや音声サンプルを返してくれるすぐれものです。「OpenMAX-IL」の仕様自体はオープンなもので、「Khronos Group」という団体によって標準化されています。

Androidでは「OpenMAX-IL」の仕様にほぼ準拠した内部APIを実装しており、それを使ってメディアのデコードをおこないます。「OpenMAX-IL」の仕組みを使って呼び出されるメディアデコード部分では、全てのAndroid端末に対してGoogleが共通して提供している標準デコーダを利用できます。しかしこれはソフトウェアによるデコーダです。

動画の圧縮にはかなり複雑な演算が必要となります。それをデコードする際にも相応のメモリを必要とし、CPUの負担も大きなものとなりがちです。特に、動画の解像度が高かったりフレームレートが高い場合はなおさらです。

このような映像デコーダをソフトウェアで実装すると、再生負荷の面で問題が発生しやすくなります。複数のCPUコアを利用して処理負荷を低減しないとコマ落ち

を発生させてしまったり、再生はできたとしても消費電力が大きく、発熱が大きすぎたり、バッテリの持ちが悪化したりします。

このような状況を改善するために、現在流通しているほとんどの端末には、「H.264」のハードウェアデコーダが搭載されています。この専用回路を「OpenMAX-IL」経由で利用することにより、CPU負荷を抑えつつスムーズな再生が実現できます。

・SurfaceFlinger

前半で述べたとおり、Androidにおいてもディスプレイへの出力をおこなうアプリは同時に複数存在し得ます。これをうまくミックスして出力してくれるのが「SurfaceFlinger」です。アプリ開発時に意識することは少ないものですが、映像出力フォーマットの設定ミスなどを起こした際に、システムが出力する例外メッセージの出力元として目にする機会があるかもしれません。

・AudioFlinger

アプリ内では複数の音声出力をおこなうことが考えられます。例えば、BGMを再生しつつ効果音を鳴らすなどです。これら複数の再生処理の仲介をしてミキシングを適切におこなう役割を持つのが「AudioFlinger」です。通常、アプリ開発中に「AudioFlinger」への出力を意識することはありません。

・AudioTrack

再生に利用されるサウンドデータはファイルから読み出される場合もあればネットワークからストリーミングされる場合もあり、またプログラム内で動的に生成されることも考えられます。これらのデータ形式は多彩で、例えばモノラル／ステレオや出力周波数といった設定を適切におこなわなければ、システム上で正常に再生できません。

この設定をおこない、PCMのサウンド再生をプログラマとOSの間で仲介する役割を持つのが「AudioTrack」です。アプリ内で動的に生成したサウンドを再生したい場合には、このクラスを利用します。

・SoundPool

「MediaPlayer」クラスは、基本的に実行時の利用メモリー削減のため「なるべくその時々に必要な映像／音声データのみを圧縮データからデコードする」という仕組みを採用しています。そのため、例えばゲームで使う効果音のように、「プログラマが指示したらなるべく早く再生をおこなう」ことが求められる音声の再生には向きません。

そうした場合に利用されるのが「SoundPool」です。「SoundPool」は、処理対象のサウンドをロード時にすべてメモリ上へ展開します。このため、アプリ内から再生指示を出してからのデコード処理が必要なく、反応性の良い出力が可能です。

反面、デメリットもあります。サウンドをすべてメモリー上へ展開することは、少ないメモリーしか搭載しない端末においてはアプリの他の部分で利用できるメモリー余裕を圧迫します。例えば、ステレオ/44.1kHz/16bitのサウンドは、1分間ぶんで約10MBのメモリーを必要とします。

Javaからマルチメディアフレームワークを扱う

「Stagefright」の持つ多くの機能は、Android 4.1ではJavaの世界からも呼び出せるようになりました。「MediaCodec」や「MediaExtractor」のクラスが「android.media」内に追加されています。

実は、それらを使って従来よりも良いパフォーマンスを実現するメディアプレイヤーの仕組みを作り上げたものが、前節のコラムで紹介した「ExoPlayer」です。

以上、メディアフレームワークの構成要素を駆け足で見てきました。更に詳細を知りたい場合には、英語の資料になりますが、以下のURLを参照してください。

http://events.linuxfoundation.org/sites/events/files/slides/Android%20builders%20summit%20-%20The%20Android%20media%20framework%20-%20final.pdf

https://source.android.com/devices/media.html

https://source.android.com/devices/audio_terminology.html

MediaPlayerには多彩な利用方法が用意されています。これを実際に利用してアプリを開発する際には、以下のレファレンスを読むのが良いでしょう。

http://developer.android.com/reference/android/media/MediaPlayer.html

15-2-2 マルチメディア API を使ったアプリを作ってみる

ここでは、前の節で「VideoView」を利用して作成していた動画再生機能を、「MediaPlayer」を利用して作り直してみましょう。

本節内で紹介した「SoundPool」の利用により、再生開始時に効果音を鳴らすようにもしてみます。今回は、効果音として「se.ogg」を用意しました。「samplemovie.mov」と同様に、あらかじめ「res/raw」ディレクトリへコピーしておいてください。

まずはレイアウトから編集していきます。**リスト4**は、「MediaPlayer」を使う場合のレイアウトXMLです。

リスト4:activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="${relativePackage}.${activityClass}" >

    <LinearLayout android:id="@+id/controlPanel" android:orientation="horizontal"
        android:layout_width="match_parent" android:layout_height="40dp" android:gravity="cent
er_horizontal"
        android:layout_margin="4dp">
        <Button android:id="@+id/playBtn"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="Play" />
    </LinearLayout>
    <SurfaceView android:id="@+id/videoSurfaceView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@+id/controlPanel" />

</RelativeLayout>
```

前回は「VideoView」を利用していた部分を「SurfaceView」へと変更しましょう。「android:id」要素も変更しますので注意してください。この時点で、Java側のコードとの間で整合しない箇所が出てくるため、プロジェクト自体はエラーとなりますが慌てずに続けて下さい。

今回は、「AndroidManifest.xml」の変更はありませんのでそのままJavaコードの編集にとりかかります。少々長くなりますので、前後半に分けて説明します。

リスト5:MacinActivity.javaの前半

```
package jp.techinstitute.playmoviesample;

import java.io.IOException;
（以下import文省略）

public class MainActivity extends Activity implements SurfaceHolder.Callback {
        SoundPool soundPool;
        int POOL_MAX = 4;
        int seId;
        MediaPlayer player = null;
        SurfaceHolder sh;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                soundPool = new SoundPool(POOL_MAX, AudioManager.STREAM_MUSIC, 0);
                seId = soundPool.load(getApplicationContext(), R.raw.se, 1);
                Button playBtn = (Button)findViewById(R.id.playBtn);
                playBtn.setOnClickListener(new OnClickListener() {
```

```
                @Override
                public void onClick(View arg0) {
                        soundPool.play(seId, 1.0f, 1.0f, 0, 0, 1.0f);
                        player.seekTo(0);
                        player.start();
                }
        });
        SurfaceView view = (SurfaceView) findViewById(R.id.videoSurfaceView);
        sh = view.getHolder();
        sh.addCallback(this);
    }
```

少々量が増えましたね。処理の構造を把握せずにいきなり書き写すのは危険かもしれません。

このリスト5のプログラムは、主に以下の3種類の処理により成り立っています。

- **「SoundPool」を利用した再生ボタンタップ時の効果音再生**
- **動画の表示先である「SurfaceView」をJava側コードで適切に処理できるようハンドラを取得と維持**
- **「MediaPlayer」による動画再生**

まず、再生ボタンをタップした際におこなわれる「SoundPool」クラスに関連する記述のみを赤枠で囲ってみます。

リスト6:MacinActivity.javaの前半(SoundPool関連)

```
package jp.techinstitute.playmoviesample;

import java.io.IOException;
(以下import文省略)

public class MainActivity extends Activity implements SurfaceHolder.Callback {
        SoundPool soundPool;
        int POOL_MAX = 4;
        int seId;
        MediaPlayer player = null;
        SurfaceHolder sh;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                soundPool = new SoundPool(POOL_MAX, AudioManager.STREAM_MUSIC, 0);
                seId = soundPool.load(getApplicationContext(), R.raw.se, 1);
                Button playBtn = (Button)findViewById(R.id.playBtn);
                playBtn.setOnClickListener(new OnClickListener() {
```

```
                @Override
                public void onClick(View arg0) {
                        soundPool.play(seId, 1.0f, 1.0f, 0, 0, 1.0f);
                        player.seekTo(0);
                        player.start();
                }
        });
        SurfaceView view = (SurfaceView) findViewById(R.id.videoSurfaceView);
        sh = view.getHolder();
        sh.addCallback(this);
    }
```

ここで指定している各種定数は、最大同時発音数設定やサウンドのボリューム指定、再生速度などです。基本的に、「SoundPool」を初期化→loadメソッドでサウンドをロード→playメソッドでサウンドを再生、という流れになっています。

次に、動画を表示する先である「SurfaceView」をJava側で適切に処理するための部分を赤枠で囲ってみます。

リスト7：MacinActivity.javaの前半（SurfaceView関連）

```
package jp.techinstitute.playmoviesample;

import java.io.IOException;
（以下import文省略）

public class MainActivity extends Activity implements SurfaceHolder.Callback {
        SoundPool soundPool;
        int POOL_MAX = 4;
        int seId;
        MediaPlayer player = null;
        SurfaceHolder sh;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                soundPool = new SoundPool(POOL_MAX, AudioManager.STREAM_MUSIC, 0);
                seId = soundPool.load(getApplicationContext(), R.raw.se, 1);
                Button playBtn = (Button)findViewById(R.id.playBtn);
                playBtn.setOnClickListener(new OnClickListener() {

                        @Override
                        public void onClick(View arg0) {
                                soundPool.play(seId, 1.0f, 1.0f, 0, 0, 1.0f);
                                player.seekTo(0);
                                player.start();
                        }
                });
                SurfaceView view = (SurfaceView) findViewById(R.id.videoSurfaceView);
                sh = view.getHolder();
                sh.addCallback(this);
    }
```

「SurfaceView」から参照保持用の「SurfaceHolder」を取得してActivity内のフィールドに保存している点、Activity自体が「SurfaceHolder.Callback」インタフェースを実装するように変更されている点に注目してください。

なお、「SurfaceHolder.Callback」に該当するメソッド群は、現時点ではまだ実装していないためコンパイルエラーとなります。

前半部分の最後として、MediaPlayer関連を見てみましょう。

リスト8:MacinActivity.javaの前半(MediaPlayer関連)

```
package jp.techinstitute.playmoviesample;

import java.io.IOException;
(以下import文省略)

public class MainActivity extends Activity implements SurfaceHolder.Callback {
        SoundPool soundPool;
        int POOL_MAX = 4;
        int seId;
        MediaPlayer player = null;
        SurfaceHolder sh;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                soundPool = new SoundPool(POOL_MAX, AudioManager.STREAM_MUSIC, 0);
                seId = soundPool.load(getApplicationContext(), R.raw.se, 1);
                Button playBtn = (Button)findViewById(R.id.playBtn);
                playBtn.setOnClickListener(new OnClickListener() {

                        @Override
                        public void onClick(View arg0) {
                                soundPool.play(seId, 1.0f, 1.0f, 0, 0, 1.0f);
                                player.seekTo(0);
                                player.start();
                        }
                });
                SurfaceView view = (SurfaceView) findViewById(R.id.videoSurfaceView);
                sh = view.getHolder();
                sh.addCallback(this);
        }
```

ここでは、フィールドとしての宣言と、ボタンのタップ時の再生処理をおこなっています。startメソッドの前にseekToメソッドを呼んでいるのは、2回目以降の再生ボタンタップ時に動画再生位置を先頭へ戻すためです。

ここまでの記述が完了したら、後半のコードを見てみましょう。

リスト9:MacinActivity.javaの後半

```
@Override
        public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2, int arg3) {
        }

        @Override
        public void surfaceCreated(SurfaceHolder arg0) {
                player = new MediaPlayer();
                String movieUri = "android.resource://" + getPackageName() + "/" + R.raw.samplemovie;
                try {
                        player.setDataSource(getApplicationContext(), Uri.parse(movieUri));
                        player.setDisplay(sh);
                        player.prepare();
                } catch (IllegalArgumentException e) {
                        e.printStackTrace();
                } catch (SecurityException e) {
                        e.printStackTrace();
                } catch (IllegalStateException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

        @Override
        public void surfaceDestroyed(SurfaceHolder arg0) {
        }

}
```

前半で出てきた「SurfaceHolder.Callback」に該当する3つのメソッド、surfaceChanged、surfaceCreated、surfaceDestroyedが実装されています。

処理の実体が書かれているのはsurfaceCreatedです。これは、アプリ画面への表示準備ができた時点で呼び出されるメソッドです。ここで「MediaPlayer」の生成をおこない、「VideoView」でおこなっていたのと同様に動画リソースを設定し、再生準備をおこなっています。

「VideoView」を利用する場合と大きく異なるのは、setDisplayメソッドを呼び出して、レイアウトで定義した「SurfaceView」を、明示的に「MediaPlayer」の出力先として指定しているところです。

ここまでの実装が完了したら、コンパイルして実機に転送してみてください。動画再生と再生時の効果音再生が確認できるはずです。

演習問題

以下のような課題に取り組んでみましょう。

- **アプリとしての完成度を高めるためには、リソースの扱いに注意を払う必要があります。動画再生に利用するコンポーネントは、システム上で限られたリソースです。他のアプリの動作を妨げないように、「SurfaceHolder.Callback」のonDestroyedコールバック時や、アプリが終了した際に呼び出されるonDestroyメソッド内で、「MediaPlayer」のreleaseメソッドを用いてリソースの解放処理をおこなうようにしてみましょう**
- **再生ボタンを押す度に最初から動画が再生される仕様を変更し、再生ボタンは再生/一時停止の切り替えにしてみましょう。あわせて、停止ボタンとループ再生ボタンも用意してみましょう。ループ再生には「MediaPlayer」のsetLoopingメソッドを利用します**

まとめ

この節では、Androidの裏側でメディア再生を担う機能について概観しました。Android 4.1以上をターゲットとする場合には、特に強力なメディア再生関連機能を利用できます。

世の中にメディア再生アプリは既に多く存在しますが、新規アプリでそれらにキャッチアップし、特徴的な機能で追い抜こうと考える際には、本節の理解が大いに役立つでしょう。

また、メディア再生をアプリ内でワンポイント要素として利用する際にも、「SoundPool」などの理解は役立つでしょう。