第12章

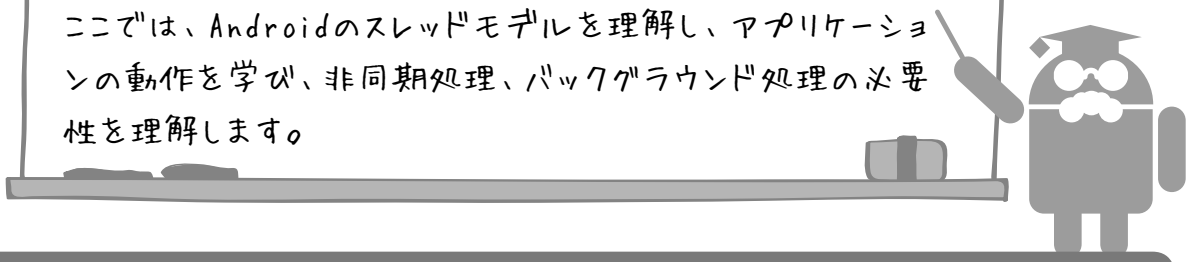
Androidアプリのバックグラウンド処理

著：日高正博

12-1 非同期処理とバックグラウンド処理

著：日高正博

非同期処理、バックグラウンド処理は、Androidプログラミングを学ぶ上で重要な考え方です。
ここでは、Androidのスレッドモデルを理解し、アプリケーションの動作を学び、非同期処理、バックグラウンド処理の必要性を理解します。



この節で学ぶこと

- スレッドの基礎
- Androidのスレッドモデルを理解する
- アプリケーションの動作モデルとエラー処理

この節で出てくるキーワード一覧

フォアグラウンド
バックグラウンド
プロセス
スレッド
UIスレッド
UIコンポーネント
ANR
Activity
Service

非同期処理の「非同期」とは、操作と処理が「同期」しないこと。つまり同じタイミングで行わなくてよいという意味です。コミュニケーションにたとえるなら、電話は「同期」しなければならない手段です。自分と相手が同じタイミングで会話しないと話ができませんよね？ 一方、メールやSNS、LINEは「非同期」の手段です。返事が早い人もいますが、メールした直後に返事がなくてもやりとりは可能です。コンピューターの処理も同様に、「同期」してすぐに反応するものと「非同期」で、あとから反応が返ってくるものがあります。

この章のねらい

本章では、Androidアプリケーションの非同期処理とバックグラウンド処理を解説します。UIスレッドで時間のかかる処理を行うと、その処理が終わるまでユーザーの操作を受け付けなくなります。Androidには時間のかかる処理、たとえばデータ通信やストレージアクセスなどを非同期で行う仕組みがあります。また、常駐型アプリケーションではServiceを使い、バックグラウンドで処理を行えます。ここではアプリケーションを作るために、Androidの持つ非同期処理やバックグラウンド処理の概要を理解したあ

と、AsyncTaskやServiceといったAPIについて学んでいきます。

目標は、非同期処理、バックグラウンド処理について、自分で最適な実装を選択できるようになることです。ひとくちに最適な手法といっても、アプリケーションの仕様や開発者の実力、プラットフォームの互換性など多くの要因に左右され、解はひとつではありません。誰かが「この手法がもっとも良かった」と言ったからといって自分にとっても最適であるか、わからないためです。各手法の特徴を学び、設計にあった取捨選択を目指します。

ソフトウェア開発において共通した要素があり、プログラミングでの定石をデザインパターンとよびます。開発における設計ノウハウをまとめたものとしては、書籍『オブジェクト指向における再利用のためのデザインパターン』が有名です。

12-1-1 フォアグラウンドとバックグラウンドとは

フォアグラウンドとは、ユーザーインターフェイス（UI）などの操作できる画面や処理、状態など、表面にあらわれている部分を指します。バックグラウンドとは、ユーザーにみせるUIの裏側で動作し、ユーザーに意識させずに行われる処理や状態のことです。バックグラウンドはUIを持たない性質上、ユーザーとの対話なしに行うタスク（処理）に向いています。

Androidでは「Activity」がフォアグラウンド処理を、「Service」がバックグラウンド処理を担当します。バックグラウンド処理を持つAndroidアプリケーションとしては、「ミュージック」や「Playミュージック」アプリなどが代表的です。

現在のコンピューターの多くは複数の作業を切り替えて実行するマルチタスク環境を持っており、それはAndroidも同様です。コンピューターのCPUやI/O（ファイル入出力）など限りあるリソースを効率的に利用するために、マルチタスク環境では待ち時間の合間に他のタスクを実行します。一般的に、フォアグラウンド処理はUIを担当します。そのため、バックグラウンド処理より優先度が高く設定されています。

12-1-2 プロセスとスレッドとは

プロセスとはアプリケーションごとの実行単位／実行環境です。基本的に1つのアプリケーションに対して1つのプロセスが割り当てられています。アプリを実行するために必要なリソース（メモリやCPUなど）は、プロセスごとに管理されます。また、プロセスは新しいプロセス（子プロセス）を生成できます。子プロセスは、生成元となったプロセス（親プロセス）とは異なるリソースを利用します。このため、複数のプロセスで同じメモリ空間を読み書きするといったリソースの共有はできません。これは、プログラムが動く仕組みを理解するうえで重要な考え方です。

スレッドとは処理の単位のことです。AndroidのActivityに関する処理は通常「main」スレッドで動作しています。「onCreate」「onPause」メソッド、各UIパーツ（ボタン、Viewなど）の描画処理は「main」スレッドで動いています。

「main」スレッドはUIスレッドと呼ばれることがあります。

デーモンスレッドとユーザースレッド

スレッドの種類は2つに分類されます。デーモンスレッドとは、プログラム終了時にスレッドの実行終了を待ちません。プログラム終了のタイミングでデーモンスレッドの処理は中断され、終わることになります。生成したスレッドで終了処理を意識しないで済むので使い勝手がよい側面があります。

ユーザースレッドは、デーモンスレッドの反対でプログラムを終わるときに、スレッドの実行終了を待ちます。プログラムを終了しようとしても、ユーザースレッドの処理が終わるまで終了できません(ユーザースレッドが生き残っている間は、プログラム実行状態です)。さきほど説明にでてきた、UIスレッドはユーザースレッドです。プログラム終了にあたっては、すべてのユーザースレッドできちんと終了処理がハンドリングできていないといけません。利点として、処理が中断してしまう恐れはないので紛失できない大事な処理などに適していることが挙げられます。

12-1-3 同期と非同期処理とは

同期と非同期は、複数の処理(スレッド)を同時に行う上で重要な考え方です。複数のスレッドを並行的に処理することをマルチスレッドと呼びます。同期処理とは同一スレッド上での動作を指します。また複数のスレッドが処理を待ち合わせることを「同期する」と表現します。

非同期処理では、複数のスレッドが阻害し合うことなく、独立して処理を行います。UIスレッドで行うには時間がかかりすぎるネットワーク通信やディスクアクセス、画像処理などは、積極的に非同期化すべきです。そうしないと、本来の役割であるUI描画をブロックしてしまい、画面が固まったようにみえるためです。

12-1-4 UIスレッドをブロックしない

Androidではユーザーの操作は、UIスレッドで処理されます(シングルスレッドモデル)。ボタンやテキストなどUIコンポーネントのインスタンスは、すべてUIスレッドで生成、操作しています。たとえばボタンを押したときに発生する「onClick」や、画面をタッチするときに発生する「onTouchEvent」は、UIスレッドで実行しています。

これらのメソッドで時間のかかる処理を行った場合、UIスレッドの他の処理はブロックされ、見た目に変化がなく固まった状態となります。一般的には、無応答時間(処理時間)が100～200ミリ秒(ms)を超えると応答が鈍いと感じます。処理時間を非同期化を検討する指標のひとつとするといいでしょう。

Androidアプリ開発では、原則として次の2つを守ってください。

- **UIスレッドで時間のかかる処理をしない**
- **UIスレッド以外からUIコンポーネントを操作しない**

アプリケーションが5秒以内に応答しなかった場合を「ANR（Application Not Responding）」と呼び、次のようなダイアログが表示されます（**図1**）。

図1： ANR（Application Not Responding）ダイアログ

「ANR」は、アプリケーションの異常を検出して、ユーザーが操作に困るような事態を防ぐ機構です。Androidは、アプリケーションの挙動を常時監視しています。5秒以上反応がないということは、アプリそのものが異常な状態（フリーズ、ハングアップ）となっています、このような状況は開発者として絶対に避けるべきです。

12-1-5 ANRが発生する要因

では「ANR」が起きるのは、どういう状況なのでしょうか。ここではサンプルコードを使って、実際にANRを発生させてみましょう。次の3つは、時間がかかる処理のなかでも代表的な処理です。

- **CPUによる演算**
- **ストレージアクセス**
- **ネットワーク通信**

順番にどのようなコードがANRを引き起こすのか確認してみましょう。

はじめのCPUによる演算とは、目に見えて処理時間がかかるほどの膨大な計算量が要因です（次の「計算量の多い処理」を参照）。

計算量の多い処理

```
int i,j,sum=0;
for (j = 0; j < 1024*1024; j++) {
    for (i = 0; i < 1024*1024; i++) {
        sum++;
    }
}
```

サンプルコードでは、ループ回数が大きく非常に時間がかかっています。サンプルのような簡単なソースコードでは、時間がかかることが一目瞭然です。一般的に、複雑なコードになればなるほど処理に時間がかかります。

アプリケーション開発においては、すべての処理について理解し、把握していることが理想です。しかし、実際には用意されたAPIを使ったり、ライブラリを使ったりと、多くの要素が絡み合っています。開発者の意図どおり、遅滞なく実行されているかは、処理時間を計測していくことで確認できます。ANRの予防には計測が重要です。

次に、ストレージアクセスを見てみましょう。ストレージは設定値の保持や画像の一時保存などで利用されています。Android端末内のメモリストレージにデータを読み書きする場合、UIスレッドで処理しないほうがいいケースがあります。

ストレージアクセスの例

```
InputStream in;
String lineBuffer;

in = openFileInput("bigdata.txt");
BufferedReader reader= new BufferedReader(new InputStreamReader(in,"UTF-8"));
while( (lineBuffer = reader.readLine()) != null ){
    Log.d("FileAccess",lineBuffer);
}
```

サンプルコードではデータを読み込もうとしています。一般に書き込みは読み込みより時間がかかり、ストレージアクセス開始から終了まで操作ができなくなります。ごくわずかな時間で処理が完了する場合は、応答速度の観点からみると、UIスレッドで処理していいと言えます。

最後は、ネットワーク通信です。AndroidではUIスレッドでネットワーク通信を行った場合、エラーが発生します。

もちろん他スレッドで実行するほうが好ましいです。しかし、処理が複雑になると不具合が混入する可能性も増えます。分かりにくいソースコードになってしまうことは本意ではありません。

ネットワーク通信

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    String url="http://localhost:8888/test";
    HttpClient httpClient = new DefaultHttpClient();
    HttpPost post = new HttpPost(url);

    ArrayList <NameValuePair> params = new ArrayList <NameValuePair>();
    params.add( new BasicNameValuePair("content", "send message"));

    HttpResponse res = null;

    try {
        post.setEntity(new UrlEncodedFormEntity(params, "utf-8"));
        res = httpClient.execute(post);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

ネットワーク通信は、非常に時間がかかるため、UIスレッドでの実行が禁止されています。

Logcatの内容

```
E/AndroidRuntime(10396): java.lang.RuntimeException:
  Unable to start activity ComponentInfo{com.example.
  background.sample00_anr/com.example.background.sample00_anr.
  MainActivity}: android.os.NetworkOnMainThreadException
```

「Logcat」を確認すると「NetworkOnMainThreadException」エラーが発生していることがわかります。

12-1-6 Androidアプリはどうやって同時に動作しているのか？

Androidでは複数のアプリを同時に実行できます。原則として「Activity」は1つしか許可されませんが、バックグラウンドで「Service」が動作しています。本文中の言葉で言い換えると、Androidはマルチプロセス動作のための機構を持っているわけです。そのための機構は「ActivityManager」と呼ばれ、ユーザーへのインタラクションを管理しています。

「ActivityManager」は、アプリケーションを管理するため、次のようなことを行っています。

- **ライフサイクルに応じてActivity／Serviceを管理する**
- **優先度に応じてプロセスを制御する**
- **未使用のプロセスを終了させてメモリ空間を確保する**

ユーザーにストレスを感じさせないように調停し、場合によっては未使用のプロセス（アプリ）を終了します。たとえば、利用中のアプリがより多くのメモリを要求したとします。すると、バックグラウンドに存在しているアプリは終了し、利用できるメモリが増えます。

優先度についても良好な応答性が維持できるように制御されています。フォアグラウンドのActivityは、ユーザーインターフェイスを提供しており、すぐに反応できない状況を防ぐために、最も優先度が高いプロセスとして制御します。またバックグラウンドにある「Service」もメモリに余裕がある限り動作し続けます。メモリがひっ迫した場合は、「Service」であっても終了します。

処理の優先度は次のとおりです。

「Service」クラスの「startForeground」メソッドをつかい、通知バーに常駐することでフォアグラウンド動作が可能です。

- **フォアグラウンドのActivity**
- **フォアグラウンドのService**※
- **バックグラウンドのService**
- **バックグラウンドのActivity**

システムによる終了は低メモリ時に行われ、外部要因に依存します。これはOSが高負荷状態であるなどデバイスの状態や、他のアプリケーションの影響を受けることを示唆しています。フレームワークの挙動を理解してアプリの設計段階から考慮しておきましょう。

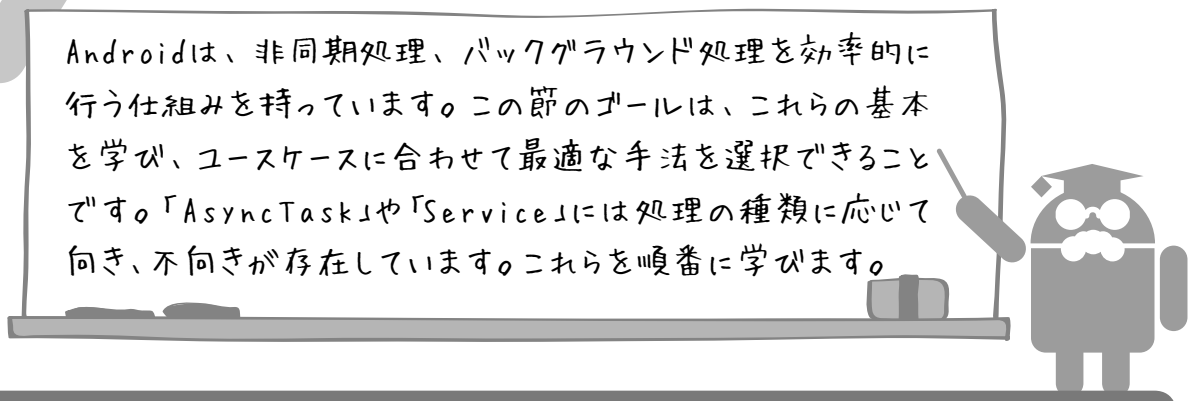
演習問題

実際にエラーを体験してみよう

- **UIスレッドの負荷を意図的にあげてANRを発生させる**
- **UIスレッド上でネットワーク通信を行い、「android.os.NetworkOnMainThreadException」エラーの発生を確認する**

12-2 さまざまな非同期処理

著：日高正博



この節で学ぶこと

- **AsyncTask、AsyncTaskLoader：もっともポピュラーな非同期処理**
- **Thread、Handler：スレッドとハンドラーによるスレッド間通信**
- **Service、IntentService：常駐型のバックグラウンド処理**

この節で出てくるキーワード一覧

AsyncTask
AsyncTaskLoader
LoaderCallbacks
LoaderManager
Thread
Runnable
ExecutorService
Handler
Service
ServiceConnection
IntentService

12-2-1 非同期処理を実現する各クラスの概要

AsyncTask

1回限りの非同期処理に向いています。Androidフレームワークの非同期用クラスです。UIスレッドからしか生成できません。非同期処理には、スレッドの生成や破棄

といった作業は不可欠ですが、煩雑です。「AsyncTask」クラスは、このようなスレッド管理をフレームワーク内部で完結しています。

AsyncTaskLoader

複数回繰り返される非同期処理に向いています。「AsyncTask」クラスを効率的に扱うための「Loader」クラスです。「Loader」とは、「AsyncTask」を読み込むためのクラスです。高速化やリソースの効率化の仕組みとしてキャッシュを持っています。複数回繰り返す処理に向き、UIスレッド以外からも生成できます。

Thread

Javaで提供されているクラスです。もっとも基本的なクラスで、スレッドの生成やスリープ、実行など基本的な操作ができます。ただし、Androidでは描画はUIスレッドでしか行えないため、処理結果をUIに反映する場合は「Handler」を理解する必要があります。自由度が高い反面、プログラムは複雑になります。

ExecutorService

Javaで提供されているクラスです。「ExecutorService」の実装方法は目的に応じて複数用意されており、単一のスレッドでの順次実行と複数のスレッドを使い分けられます。処理内容のほか、CPUやメモリなどのリソースに応じてどの実装（提供されたクラス）を使うかの選択が必要です。

Handler

「Handler」クラスは「Looper」と連携して、処理順序のスケジューリングや処理を別スレッドで実行するための機能を持っています。とくに、UIスレッドで描画処理を実行するために利用します。

Service

バックグラウンド動作のためのクラスです。音楽再生のようにバックグラウンドで継続して処理を行うために利用します。スレッドとは異なります。

IntentService

「IntentService」はアプリケーションの動作状況に依存しないで、自分の仕事がなくなるまでバックグラウンドで処理を行います。「AsyncTask」と似ていますがアプリケーションがフォアグラウンドでなくなっても中断されない利点があります。

良い実装のための指標

良い設計の第一歩は、各手法の特徴を理解しておくことです。目安としては次の通りです。

- **1回限りの処理、または繰り返しの少ない処理：AsyncTask**
- **リスト表示するなど頻出する繰り返し処理：AsyncTaskLoader**
- **実行完了の保証が欲しい処理：IntentService**
- **アプリ終了後もバックグラウンド動作したい処理：Service**

プログラミングの目的は、さまざまなので、すべてのケースに当てはまるわけではありませんが、基本として理解しておくと応用の幅がひろがります。

12-2-2 AsyncTask

「AsyncTask」は非同期処理のための便利なヘルパークラスです。「AsyncTask」クラス内部では、非同期処理の為に「Thread」と「Hendler」が使われていますが、クラス内で隠蔽されており意識する必要はありません。

ヘルパークラスとは、処理を手助けする役割をまとめたクラスの総称です。

「AsyncTask」クラスは、非同期処理を4つのステップに区切っています。実行前、実行開始、実行中、実行後です。それぞれ「onPreExecute」メソッド、「doInBackground」メソッド、「onProgressUpdate」メソッド、「onPostExecute」メソッドが担当します（**図1**）。

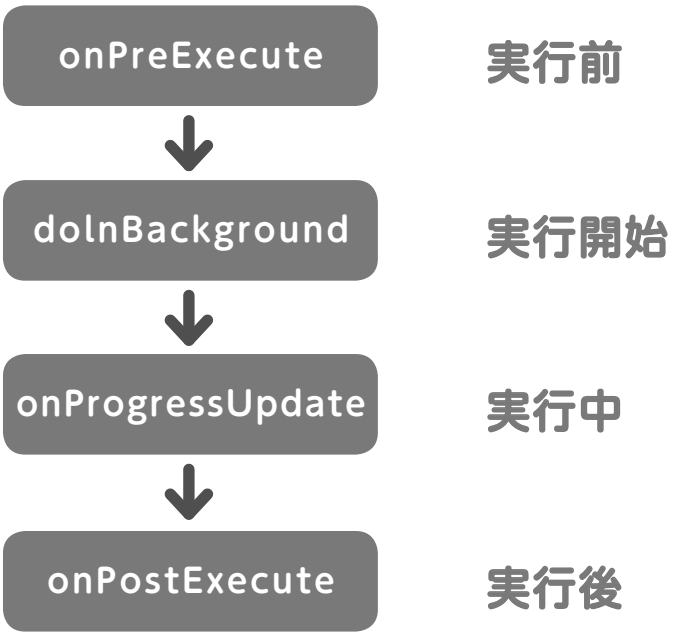


図1: AsyncTaskの処理シーケンス

「AsyncTask」クラスを使う際には次の特徴に注意してください。

- **UIスレッドでインスタンスを生成する**
- **実行済みのインスタンスは再利用できない**
- **onPostExecuteメソッドはUIスレッドで実行される**

「AsyncTask」クラス最大の利点は簡単に非同期処理できるところです。ただし、そのためにいくつかの制限があり、そのうちのひとつがUIスレッドでインスタンスを生成することです。もうひとつは実行済みのインスタンスは再利用できません。つまり、1度実行した「AsyncTask」は再度実行できません。破棄するのみです。

再利用性が低い設計になっていることからも、1回限定の処理に向いている(むしろ、そのために作られた)といえます。そしてもっとも嬉しい点は「onPostExecute」メソッドがUIスレッドで実行されることでしょう。

通常ならUIコンポーネントを操作するためには、「Handler」を使ってUIスレッドで処理するコードが必要です。「AsyncTask」ならそのような面倒なコードは不要というわけです。

AsyncTaskを使わない場合

ここでは非同期処理が必要な計算量の多い例として、画像処理を扱います。「Google Playストア」にもトイカメラ風の写真を撮るアプリなど面白い効果をもったカメラアプリがいくつも公開されていますが、今回はシンプルに用意されている画像をカラーからモノクロに変換するアプリを作ってみましょう(**図2**)。

図2: AsyncTaskサンプルアプリ

画面下の開始ボタンを押すと画像をモノクロに変換する処理を行います。

まずは「activity_main.xml」にレイアウトを作成します。

activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    ～省略～
    tools:context=".MainActivity" >

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:src="@drawable/ic_launcher_big" />

    <Button
        android:id="@+id/execButton"
        android:layout_below="@+id/imageView"
        ～省略～
        android:text="開始" />

    <Button
        android:id="@+id/countButton"
        android:layout_below="@+id/execButton"
        ～省略～
        android:text="カウントアップ" />

</RelativeLayout>
```

はじめは、「AsyncTask」を使わずにプログラミングしてみます。これは、計算量の多い処理があるとUIスレッドが止まり、ユーザーが操作できないことを確認するためです。操作できないことを体験するために、2番目のボタン「カウントアップ」を用意します。このボタンを押すと表示されているラベル（数字）がカウントアップしていくものです。

MainActivity.java

```
public class MainActivity extends Activity {

    private ImageView mImageView;
    private Bitmap mBitmap;
    private Button mCountButton;
    private Integer mCount = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // イメージの準備
        mBitmap = BitmapFactory.decodeResource(
                getResources(), R.drawable.ic_launcher_big);

        // 変換前のイメージを表示
        mImageView = (ImageView)findViewById(R.id.imageView);
        mImageView.setImageBitmap(mBitmap);

        // 押されるたびにカウントアップするボタン
        mCountButton =  (Button)findViewById(R.id.countButton);
        mCountButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View view) {
                mCount++;
                ((Button)view).setText(mCount.toString());
            }
        });
        ～省略～
    }
}
```

これで2番目のボタンである「カウントアップ」ボタンが配置できました。引き続き開始ボタンの処理を記述します。

MainActivity.java

```
    ～省略～
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ～省略～

        // 同期処理の開始
        Button execButton = (Button)findViewById(R.id.execButton);
        execButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View view) {
                // モノクロにする処理
                Bitmap out = mBitmap.copy(Bitmap.Config.ARGB_8888, true);

                int width = out.getWidth();
                int height = out.getHeight();
                int totalPixcel = width * height;

                int i, j;
                for (j = 0; j < height; j++) {
                    for (i = 0; i < width; i++) {
                        int pixelColor = out.getPixel(i, j);
                        int y = (int) (0.299 * Color.red(pixelColor) +
                                0.587 * Color.green(pixelColor) +
                                0.114 * Color.blue(pixelColor));
                        out.setPixel(i, j, Color.rgb(y, y, y));
                    }
                }

                // 変換が終わったので表示する
                mImageView.setImageBitmap(out);
            }
        });
    }
}
```

アプリを実行して、「開始」ボタンと「カウントアップ」ボタンを押してみましょう。しばらく待っていると、画像がカラーからモノクロへ変換されます。すべてUIスレッドで処理しているため、「開始」ボタンを押したあとは「カウントアップ」ボタンが操作できず、最悪の場合、ANRに陥ります。

ANRを確認するために「カウントアップ」ボタンを連続で押してみてください。

AsyncTaskを使う場合

前述のサンプルコードを変更して、画像処理部分を分離し、非同期化します。このサンプルコードでは画像処理という、比較的わかりやすい例を取り扱っていますが、実際のアプリ開発においては、重い処理を探し出して非同期化することになります。サンプルは処理を非同期化するスタンダードな手法として覚えておくといいでしょう。

「AsyncTask」のおもなメソッドは次の通りです。

メソッド	説　明
onPreExecute()	事前に準備する内容を記述する
doInBackground(Params...)	バックグラウンドで実行する
onProgressUpdate(Progress...)	進捗状況をUIスレッドで表示する
onPostExecute(Result)	バックグラウンド処理が完了し、UIスレッドに反映する

AsyncTaskのおもなメソッド

各メソッドの「Params」「Progress」「Result」は引数となるクラスの例です。実際には必要に応じて指定します。

MonochromeTask.java

```
public class MonochromeTask extends AsyncTask<Bitmap, Integer, Bitmap> {
    private ImageView mImageView;

    public MonochromeTask(ImageView imageView) {
        super();
        mImageView = imageView;
    }

    @Override
    protected Bitmap doInBackground(Bitmap... bitmap) {
        // 非同期で処理する
        Bitmap out = bitmap[0].copy(Bitmap.Config.ARGB_8888, true);

        int width = out.getWidth();
        int height = out.getHeight();
        int totalPixcel = width * height;

        int i, j;
        for (j = 0; j < height; j++) {
            for (i = 0; i < width; i++) {
                int pixelColor = out.getPixel(i, j);
                // モノクロ化
                int y = (int) (0.299 * Color.red(pixelColor) +
                        0.587 * Color.green(pixelColor) +
                        0.114 * Color.blue(pixelColor));
                out.setPixel(i, j, Color.rgb(y, y, y));
            }
        }
        return out;
    }

    @Override
    protected void onPostExecute(Bitmap result) {
        // 実行後にImageViewへ反映
        mImageView.setImageBitmap(result);
    }
}
```

ここで注目すべきは「extends AsyncTask<Bitmap, Integer, Bitmap>」の部分です。前述の引数では「Params」「Progress」「Result」が出てきましたが、ここで引数の型を指定しています。

1つ目の「Params」はバックグラウンド処理を実行する時に与える「execute」メソッドの引数の型です。2つ目の「Progress」は進捗状況を表示する「onProgressUpdate」メソッドの引数の型です。最後の「Result」はバックグラウンド処理の後に受け取る「onPostExecute」メソッドの引数の型です。

今回はビットマップ画像を与えてモノクロに変換されたビットマップを受け取ります。

「Activity」側で必要な処理は「MonochromeTask」クラス(「AsyncTask」のサブクラス)のインスタンスを生成することと、非同期処理を開始するために「execute」メソッドを呼び出すことです。

「execButton」の「OnClickListener」を次の通りに書き換えてみましょう。

MainActivity.java

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ～省略～
        Button execButton = (Button)findViewById(R.id.execButton);
        execButton.setOnClickListener(new OnClickListener() {

            MonochromeTask task =
                    new MonochromeTask(mImageView);

            @Override
            public void onClick(View view) {
                // モノクロにする処理
                task.execute(mBitmap);
            }
        });
        ～省略～
    }
```

「AsyncTask」を使わない場合と違い、UIスレッドをブロックしません。この状態で「カウントアップ」ボタンを押すと、問題なくカウントアップされていきます。

進捗を表示する

画像処理に関わらず、重い処理を行う時に見た目が固まるのは良くありません。何がおきているか、ユーザーが把握できず、不安になるためです。Windowsなどでファイルをコピーするときに出るプログレスバーがユーザビリティの良い例ですね。大量のファイルをコピーする時などプログレスバーが出ていないと本当にコピーしているのか不安になることがあります。

最後にプログレスバーによる進捗表示を追加してみましょう。

MainActivity.java

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ～省略～
    Button execButton = (Button)findViewById(R.id.execButton);
    execButton.setOnClickListener(new OnClickListener() {

        MonochromeTask task =
                new MonochromeTask(getParent(),mImageView);

        @Override
        public void onClick(View view) {
            // モノクロにする処理
            task.execute(mBitmap);
        }
    });
    ～省略～
}
```

「MonochromeTask（AsyncTask）」のコンストラクタに「getParent」メソッドで取得した「Activity」を渡します。

「MonochromeTask」では、進捗を表示するためのプログレスバーを「onPreExecute」メソッドで準備します。また、非同期処理を行う「doInBackground」メソッドから進捗を随時アップデートする「onProgressUpdate」メソッドを呼び出します。

MonochromeTask.java

```
public class MonochromeTask extends AsyncTask<Bitmap, Integer, Bitmap> {
    private ImageView mImageView;
    private ProgressDialog mDialog;
    private Context mContext;

    public MonochromeTask(Context context, ImageView imageView) {
        super();
        mContext = context;
        mImageView = imageView;
    }
    @Override
    protected void onPreExecute() {
        //処理前にプログレスバーを準備
        mDialog = new ProgressDialog(mContext);
        mDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
        mDialog.setIndeterminate(false);
        mDialog.setMax(100); //100%表記
        mDialog.show();
    }

    @Override
    protected Bitmap doInBackground(Bitmap... bitmap) {
        // 非同期で処理する
        Bitmap out = bitmap[0].copy(Bitmap.Config.ARGB_8888, true);

        int width = out.getWidth();
        int height = out.getHeight();
        int totalPixcel = width * height;

        int i, j;
        for (j = 0; j < height; j++) {
            for (i = 0; i < width; i++) {
                int pixelColor = out.getPixel(i, j);
                // モノクロ化
                int y = (int) (0.299 * Color.red(pixelColor) +
                        0.587 * Color.green(pixelColor) +
                        0.114 * Color.blue(pixelColor));
                out.setPixel(i, j, Color.rgb(y, y, y));
            }
            float percent = ((i + j*height) /
                    (float)totalPixcel ) * 100;
            onProgressUpdate((int)percent);
        }
        return out;
    }
    @Override
    protected void onProgressUpdate(Integer... progress) {
        // 実行中
        mDialog.setProgress(progress[0]);
    }

    @Override
    protected void onPostExecute(Bitmap result) {
        // 実行後にImageViewへ反映
        mDialog.dismiss();
        mImageView.setImageBitmap(result);
    }
}
```

おもな変更点は次のとおりです。進捗を表示するためにコンストラクタの引数にContextを追加しました。準備として「onPreExecute」メソッドでプログレスダイアログ（ProgressDialog）を用意しています。進捗を更新するために「onProgressUpdate」メソッドを追加し「doInBackground」メソッドから「onProgressUpdate」メソッドを呼び出しています。処理が完了したら「onPostExecute」メソッドでダイアログを非表示にして終了です。

「AsyncTask」は非同期処理を行えるもっとも手軽な手段です。しかし「ProgressDialog」のようにUIコンポーネントを使う場合は、思いの外、複雑になってしまいます。「Context」を持つ必要があることからも、Activityと密に結合せざるを得ません。

サンプルコードでは、UIコンポーネントの「ProgressDialog」を使って進捗表示する例を示しました。非同期処理をしているにも関わらず、ほかのUIを操作できなくなってしまいました。アプリとして、あまりうれしい挙動ではないでしょう。

実際「AsyncTask」が向いている処理は、ファイルのダウンロードや、データのネットワーク送信など進捗表示を必要としないバックグラウンド動作です。

12-2-3 AsyncTaskLoader

「AsyncTaskLoader」は、複数回繰り返される非同期処理に向いています。「AsyncTask」クラスを置き換えるものではありません。効率的に扱うための「Loader」クラスです。「Loader」とは、名前の通り読み込む（ロードする）機能を意味します。「Activity」や「Fragment」といったライフサイクルを考慮して作られている点も大きな利点です。

「AsyncTaskLoader」を利用する際には、次の特徴を覚えておきましょう。

- **LoaderCallbacksをつかってUIを制御するコードが分離できる**
- **高速化やリソースの効率化の仕組みとしてキャッシュを持てる**
- **UIスレッド以外からも生成できる**

「AsyncTask」と比較しやすいように、画像をモノクロに変換するサンプルコードを見てみましょう。

MonochromeTaskLoader.java

```
    public class MonochromeTaskLoader extends AsyncTaskLoader<Bitmap> {
    Bitmap mBitmap;

    public MonochromeTaskLoader(Context context, Bitmap image) {
        super(context);
        // 処理対象のビットマップ
        mBitmap = image;
    }

    @Override
    public Bitmap loadInBackground() {
        // 非同期で処理する
        Bitmap out = mBitmap.copy(Bitmap.Config.ARGB_8888, true);

        int width = out.getWidth();
        int height = out.getHeight();

        int i, j;
        for (j = 0; j < height; j++) {
            for (i = 0; i < width; i++) {
                int pixelColor = out.getPixel(i, j);
                // モノクロ化
                int y = (int) (0.299 * Color.red(pixelColor) +
                        0.587 * Color.green(pixelColor) +
                        0.114 * Color.blue(pixelColor));
                out.setPixel(i, j, Color.rgb(y, y, y));
            }
        }
        return out;
    }

    @Override
    protected void onStartLoading() {
        forceLoad();
    }
}
```

「loadInBackground」メソッドで画像のモノクロ変換を行っています。UIコンポーネントに関わる処理は後述の「LoaderCallbacks」が担当するため、非同期に実行される処理のみ記述しています。シンプルな構成です。

「onStartLoading」メソッドは「Loader」の開始時に呼び出されます。サンプルでは強制的に読み込みを始めていますが、条件を設けることも可能です。「Loader」は「resume」時など独自の判断に基づき、「onStartLoading」を呼び出しています（「onStartLoading」は直接呼び出してはいけません。かわりに「startLoading」が用意されています）。

またサンプルコードでは利用していませんが、「AsyncTaskLoader」には読み込み完了時に呼び出される「onStopLoading」メソッドやキャンセル用の「cancelLoad」メソッドが用意されています。

Activityから「AsyncTaskLoader」を呼び出す際は「LoaderManager」の

「initLoader」メソッドを利用します。

MainActivity.java

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    ～省略～

    mExecButton =  (Button)findViewById(R.id.execButton);
    mExecButton.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View view) {
            initAsyncTaskLoader(R.drawable.ic_launcher_big);
        }
    });

}

public void initAsyncTaskLoader(int resId) {
    Bundle args = new Bundle();
    args.putInt("drawableId", resId);
    // Loaderを初期化する
    getLoaderManager().initLoader(0, args, this);
}
```

「LoaderManager」クラスの「initLoader」メソッドは3つの引数をもち、ローダーID、ローダーに渡す引数、読み込みイベントを受け取る「LoaderCallbacks」の3つを指定します。サンプルコードではIDは0番、引数にモノクロに変換するdrawableIDを指定しています。「LoaderCallbacks」は、ここでは「Activity」に実装するため「this」と自分自身を指定します。

「LoaderCallbacks」とはローダーの読み込みを行うためのインターフェースです。クライアント（呼び出し元である「Activity」）がローダーの管理者である「LoaderManager」とやりとりする仕組みで、次のメソッドの実装が必要です（**表1**）。

メソッド	説　明
onCreateLoader	新しくLoaderが生成されるとき
onLoadFinished	Loaderでの非同期処理が終了するとき
onLoaderReset	Loaderがリセットされるとき

表1：「LoaderCallbacks」で実装が必要なメソッド

次のコードで「LoaderCallbacks」インターフェイスの実装をみてみましょう。

MainActivity.java

```
public class MainActivity extends Activity implements LoaderCallbacks<Bitmap>{
    ～省略～
    // LoaderCallbacks
    @Override
    public Loader<Bitmap> onCreateLoader(int id, Bundle args) {
        if(args != null) {
            int resId = args.getInt("drawableId");
            Bitmap bitmap = BitmapFactory
                    .decodeResource(getResources(), resId);
            // 非同期処理を開始する
            return new MonochromeTaskLoader(this, bitmap);
        }
        return null;
    }

    @Override
    public void onLoadFinished(Loader<Bitmap> loader, Bitmap bitmap) {
        // 画像を更新する
        mImageView.setImageBitmap(bitmap);
        mImageView.invalidate();
    }

    @Override
    public void onLoaderReset(Loader<Bitmap> loader) {
        // リセットされた場合の処理
    }
}
```

「onCreateLoader」メソッドでは「Loader」のIDと引数を受け取ります。第2引数の「Bundle」から「drawableID」を取得して「MonochromeTaskLoader」を生成しています。

また引数は「LoaderManager」クラスの「initLoader」メソッドの第1引数、第2引数と同等です。サンプルコードでは利用していませんがIDをつかって処理を変えたり、「Loader」を切り替えたり、処理条件を変更できます。

「onLoadFinished」メソッドでは、モノクロ変換が完了した画像を「ImageView」に設定しています。第1引数でLoaderを受け取ります。ローダー側で必要な処理があればこのメソッド内で実行可能です。

「onLoaderReset」メソッドはサンプルコードでは利用していません。リセット時の挙動（UIコンポーネントの操作など）を記述するといいでしょう。

「AsyncTaskLoader」では「LoaderManager」「Loader」「LoaderCallbacks」が登場するようになりました。「AsyncTask」にくらべると構成が複雑です。しかし、サンプルコードではUIコンポーネントの操作が「LoaderCallbacks」にまとめられるなど、それぞれの責務が明確となっています。「AsyncTask」のサンプルコードと見比べても、すっきりしたと感じるでしょう（じつは、コード量は増えていますが、読みやすく感じるのなら良い構成といえます）。

それぞれの役割は**表2**のように理解するといいでしょう。

ここでは「Loader<Bitmap>」という型になっていますが「MonochromeTaskLoader」が「AsyncTaskLoader<Bitmap>」を継承していることを思い出しましょう。ここで「<Bitmap>」はジェネリクス（型変数）で型を指定します。「AsyncTaskLoader」はさらに「Loader<D>」を継承しているため、上位のクラスで表記できるわけです。

「ContentResolver」へアクセスするための「CursorLoader」も「AsyncTaskLoader」を継承しています。

クラス	説　明
`LoaderManager`	ActivityやFragmentに対応したライフサイクル管理
`LoaderCallbacks`	LoaderManagerとLoaderが連携するためのインターフェイス
`Loader`	非同期処理を行う

表2:「AsyncTaskLoader」の実現に必要な要素

ここまでで「AsyncTaskLoader」を解説しました。「Loader」という仕組みは非同期処理を汎用的に使うために作られたといえます。「LoaderManager」により管理されるため、独自の実装（たとえばActivityやFragmentなどのライフサイクルから切り離した設計）では利用しにくいことも覚えておきましょう。

12-2-4 Threadと Handler

ここまで何度も繰り返してきましたが、Androidアプリケーションはシングルスレッドモデルを採用しています。いかにUIスレッドをブロックしないかが開発のポイントです。「AsyncTask」「AsyncTaskLoader」は、非同期処理のための便利なクラスですが、これらのクラスも大本をたどれば「Thread」クラスと「Handler」クラスで作られています。これらは非同期処理の基本ともいえる機能で、マルチスレッドで処理を行うには避けて通れません。

「Thread」はJavaで提供されているクラスです。もっとも基本的なクラスで、スレッドの生成、スリープ、実行など基本的な操作ができます。処理の実行単位といって差し支えありません。自由度が高い一方で、実装コストも高くなります。

「Handler（ハンドラー）」は、「Looper（ルーパー）」と連携して動作し、キューを使った処理順序のスケジューリングや、処理を別スレッドで実行するための機能を持っています。

「Looper」とは繰り返し処理をする機構です。要求された作業を順番に処理します。すべて終えたら次の作業が発生するまで待機します。

- **UIスレッドなど指定のスレッドのLooperと連携する**
- **任意のスレッドにメッセージ、処理を送信できる**

キューとは？

キュー（待ち行列）は、データ構造の一種です。データを先入れ先出し（FIFO：First In First Out）で保持します。先に入れたデータから取り出します。キューに従ってデータ、メッセージを管理することで順番とおりに処理できる仕組みです。

ここでは、別スレッドで1秒ごとカウントアップして、結果を「TextView」に反映するコードを考えてみます（**図3**）。

図3: 1秒ごとカウントアップするサンプルアプリ

実際にサンプルコードで確認してみましょう。「Handler」では処理とメッセージが送れますが、最初に解説する方法は「Thread」で非同期処理を行い、処理の完了時にメッセージを通知する手法です。

MainActivity.java

```
public class MainActivity extends Activity implements Runnable{

    private Thread mThread;
    private Handler mHandler;
    private TextView mTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mTextView = (TextView)findViewById(R.id.textView);

        mHandler = new Handler(){
            //メッセージ受信
            public void handleMessage(android.os.Message message) {
                //メッセージの表示
                String text = (String) message.obj;
                mTextView.setText( text );
                //メッセージの種類に応じてswitch文で制御すれば
                //イベント制御に利用可能
            };
        };
    }
  ～省略～
}
```

「onCreate」メソッド内で「Handler」のインスタンスを生成します。「Handler」は、生成されたスレッドの「Looper」に紐づけられます。このサンプルでは暗黙的にUIスレッドで処理する「Handler」を作成しています。「handleMessage」メソッドでは、「TextView」を操作していますが、UIスレッド上で実行するのでシングルスレッドモデルに違反しません。

実際に非同期処理を行う「Thread」は、次のとおりです。まずは開始と終了処理のみを確認してみましょう。

MainActivity.java

```
public class MainActivity extends Activity implements Runnable{

    private Thread mThread;
    private Handler mHandler;
    private TextView mTextView;

    ～省略～
    @Override
    public void onResume(){
        super.onResume();

        mThread = new Thread(this);
        //スレッド処理を開始
        if(mThread != null ){
            mThread.start();
        }
    }

    @Override
    public void onPause(){
        super.onPause();
        //スレッドを削除
        mThread = null;
    }
    ～省略～
}
```

「onResume」メソッドでスレッドを開始し「onPause」メソッドで処理を終了しています。アプリケーションがフォアグラウンドにいる間は動作し続けます。「onResume」メソッド内にある「Thread」のインスタンスを生成するタイミングで、コンストラクタに「this（Activity）」を指定しています。これは「MainActivity」のクラス宣言にあるとおり、「Activity」に実装した「Runnable」インターフェイスを与えています（これについては後述します）。

スレッドで処理している部分は次のとおりです。

MainActivity.java

```
public class MainActivity extends Activity implements Runnable{

    private Thread mThread;
    private Handler mHandler;
    private TextView mTextView;
    ～省略～

    //スレッドによる更新処理
    public void run() {
        long time = System.currentTimeMillis();
        long count = 0;

        while (mThread != null) {
            long now = System.currentTimeMillis();
            if(now - time > 1000){
                //Message msg = new Message();
                //Message msg = Message.obtain();
                Message msg = mHandler.obtainMessage();
                msg.obj = new String("ループが"+ count + "回終了しました");

                //ハンドラへのメッセージ送信
                mHandler.sendMessage(msg);

                //スレッドの利用変数を初期化
                time = now;
                count++;
            }
        }
    }
}
```

「Runnable」インターフェイスの「run」メソッドでシステムの時間を数えています。ここでは無限ループとならないように「mThread」変数の有無を「while」の条件としています。1秒以上の差があれば「Handler」に対してメッセージ（「TextView」に表示したい文字列）を送信しています。

メッセージの送受信ではなく、UIスレッドで動作させたい処理そのものを送る場合は、次のサンプルコードのようになります。

MainActivity.java

```
public class MainActivity extends Activity implements Runnable{

    private Thread mThread;
    private Handler mHandler;
    private TextView mTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mTextView = (TextView)findViewById(R.id.textView);

        mHandler = new Handler();
    }
    ～省略～
    //スレッドによる更新処理
    public void run() {
        long time = System.currentTimeMillis();
        long count = 0;

        while (mThread != null) {
            long now = System.currentTimeMillis();
            if(now - time > 1000){
                final String text = new String("ループが"+
                        count + "回終了しました");
                mHandler.post(new Runnable() {

                    @Override
                    public void run() {
                        // UIスレッドで動作する
                        mTextView.setText( text );
                    }
                });

                //スレッドの利用変数を初期化
                time = now;
                count++;
            }
        }
    }
}
```

非同期処理を行っている「Thread」から「Handler」クラスの「post」メソッドを使ってUIスレッドで実行したい処理を記述しています。「AsyncTask」の「onPostExecute」のようですね。「AsyncTask」とは違い、何度でも呼び出すことができます。

「post」メソッドの引数は「Runnable」です。「Thread」を生成した際も引数が「Runnable」であったことを覚えているかもしれません。「Runnable」は、Javaでの実行単位です。

「Handler」と「Thread」が連携することでメッセージの送受信や非同期処理、結果の通知が可能です。「Handler」クラスの実装をまとめると**表3**のとおりとなります。

メソッド	説　明
sendMessageメソッド	メッセージを送信する
handleMessageメソッド	メッセージを受信する
postメソッド	Runnableをつかって処理を送信する

表3:「Handler」クラスのおもなメソッド

いずれもキューで管理されるため先入れ先出しです。呼ばれた順番に処理されます。同じ「Handler」への操作であれば、あとから「post」した処理が先に実行されるようなことはありません。

「Thread」クラスでは、次の点に注意して利用しましょう。

- **Runnableで非同期化したい処理を記述する**
- **処理の回数ごとThreadを生成するとコストが高い**
- **startメソッドで開始できるが停止時のstopは非推奨**

「Thread」を停止する「stop」メソッドは構造上の欠陥から非推奨となります。「interrupt」メソッドによる割り込みが適切ですが、より深いプログラミングの知識が必要です。サンプルコードでは複雑になることを避けるため「mThread」インスタンスで代用しています。どのように終了を定義しているか、再度読み直してみてください。

また単純な非同期処理であれば「Thread」を使う必要はありません。「Thread」の生成はコストが高く、頻繁に繰り返すとパフォーマンスに悪影響を及ぼします。

じつは、Activityには「runOnUiThread」メソッドという、「post」メソッドを暗黙的に使う便利な機能が備わっています。「Handler」が不要になるわけではありませんが、覚えておいて損はありません。

12-2-5 Service

Androidでバックグラウンド動作する、常駐型アプリケーションを作成する場合には、「Service」の知識が欠かせません。ここでは「Service」のライフサイクル（**図4**）を解説したあと、実際の利用上の注意点に触れていきます。

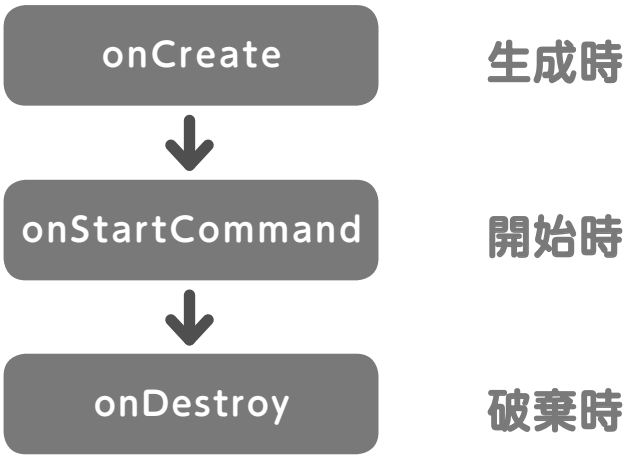


図4: Serviceのライフサイクル

ActivityはContextを継承しているため、Activityも同じメソッドを持っています。

「Service」のライフサイクルは、次の3つの状態があります（**表4**）。

状態	説　明
onCreate	生成時
onStartCommand	開始時
onDestroy	破棄時

表4:「Service」のライフサイクルの3種の状態

「Service」の実行方法によってライフサイクルが異なります。以下の説明は、それぞれ「Context」クラスの「startService」と「bindService」メソッドを使った場合です。

- **startServiceメソッドとstopServiceメソッド**

「Service」全般として実行中は「Service」から「Activity」へ「Intent」の発行が可能です。サービス起動後は「Activity」から「Service」を制御する経路がありません。「Service」の生存期間は「Activity」に依存せず、明示的に「stopService」が呼ばれるまで動き続けます。

- **bindServiceメソッドとunbindServiceメソッド**

「バインド(bind)」という仕組みを使い、「ActvitiyとService」でコネクションを確立します。バインドを使うことで「Activity」から「Service」を制御できます。「Service」の生存期間はコネクションに依存します。コネクションが切断されると「Service」は終了します。

はじめに一般的なサービスのライフサイクル、「startService」による「Service」起動を解説します。

startServiceによるService起動

「Context」クラスの「startService」メソッドによる実行の場合、「Service」のライフサイクルは「onCreate」「onStartCommand」「onDestroy」の3つのコールバックメソッドが呼び出されます。

- **public void onCreate()**
- **public void onDestroy()**
- **public int onStartCommand(Intent intent, int flags, int startId)**

「onCreate」メソッドはServiceのインスタンス生成時（複数回「startService」メソッドを実行した場合は初回のみ）呼び出されます。「onStartCommand」メソッドでは「startService」で送られた「Intent」を受けとります。「stopService」メソッドに

よるサービス終了のタイミングで「onDestroy」メソッドが呼ばれます。「Activity」で「stopService」を呼ばずにアプリケーションを終了した場合は、「Service」は終了せず、バックグラウンドで動き続けます。

サンプルとして次のように「Service」を実行するボタンを準備し（**図5**）、サービスの動作を確認してみましょう。

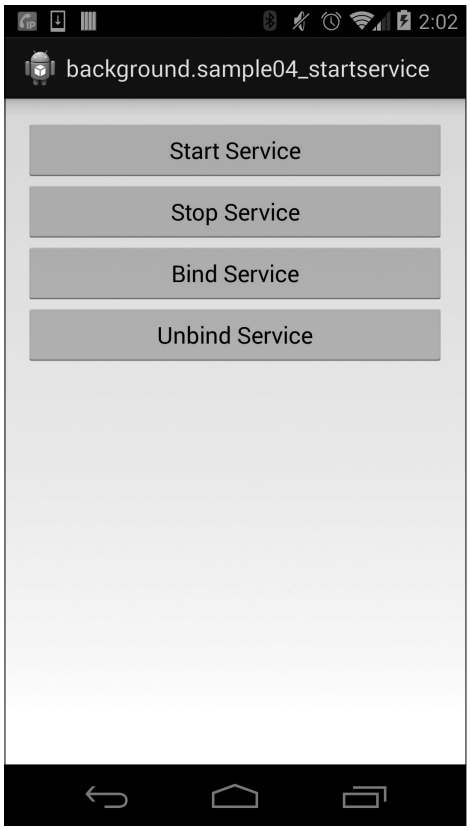


図5: Serviceを実行するサンプルアプリ

MyService.java

```
public class MyService extends Service {

    static final String TAG="LocalService";

    @Override
    public void onCreate() {
        Log.i(TAG, "onCreate");
        Toast.makeText(this, "MyService#onCreate", Toast.LENGTH_SHORT).show();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.i(TAG, "onStartCommand Received start id " + startId + ": " + intent);
        Toast.makeText(this, "MyService#onStartCommand", Toast.LENGTH_SHORT).show();
        //明示的にサービスの起動、停止が決められる場合の返り値
        return START_STICKY;
    }

    @Override
    public void onDestroy() {
        Log.i(TAG, "onDestroy");
        Toast.makeText(this, "MyService#onDestroy", Toast.LENGTH_SHORT).show();
    }
}
```

コールバックメソッドのみ実装した簡単な「Service」です。「Service」を使う際は、以下のように「AndroidManifest.xml」に「service」要素を追加します。

AndroidManifest.xml

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

以下のActivityでは「Start Service」ボタンで開始、「Stop Service」ボタンで終了します。

MainActivity.java

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Button btn = (Button) findViewById(R.id.startButton);
        btn.setOnClickListener(btnListener);//リスナの登録

        btn  = (Button) findViewById(R.id.stopButton);
        btn.setOnClickListener(btnListener);//リスナの登録
    }

    private OnClickListener btnListener = new OnClickListener() {
        public void onClick(View v) {

            switch(v.getId()){

            case R.id.startButton://startServiceでサービスを起動
                startService(new Intent(MainActivity.this, MyService.class));
                break;
            case R.id.stopButton://stopServiceでサービスの終了
                stopService(new Intent(MainActivity.this, MyService.class));
                break;
            }
        }
    };
}
```

「OnClickListener」でサービスの起動、終了を行います。「startService」メソッド、「stopService」メソッドの引数には「Intent」が必要です。ここでは「Intent(Context, Class)」コンストラクタで「Intent」を生成して「Service」を呼び出しています。

「Start Service」「Stop Service」ボタンの順番で押して「Service」のライフサイクルを確認してみます。

Serviceの動作

```
INFO/LocalService(1619): onCreate
INFO/LocalService(1619): onStartCommand Received start id 1:
  Intent { cmp=jp.androidopentextbook.background.sample04/.MyService }
INFO/LocalService(1619): onDestroy
```

「Start Service」「Start Service」「Stop Service」の順番でボタンを押すと「onCreate」メソッドの呼び出しが初回のみであることがわかります。

「onCreate」メソッドの呼び出し状態

```
INFO/LocalService(1619): onCreate
INFO/LocalService(1619): onStartCommand Received start id 1:
  Intent { cmp=jp.androidopentextbook.background.sample04/.MyService }
INFO/LocalService(1619): onStartCommand Received start id 2:
  Intent { cmp=jp.androidopentextbook.background.sample04/.MyService }
INFO/LocalService(1619): onDestroy
```

「Activity」と比べると単純なライフサイクルですが、「Service」の終了を忘れると、ずっと動作し続けます。バックグラウンド動作中は、通常より電池を消耗したり、動作が緩慢になったり、他のアプリにも影響を及ぼしたりします。目に見えない処理だからこそ、ライフサイクルを十分に意識しましょう。

bindServiceによるService起動

「bind（バインド）」という仕組みを使って「Actvitiy」と「Service」を接続するケースでは、ライフサイクルに多少の違いがあります。**図6**にあるとおり「onStartCommand」メソッドが呼び出されることはありません。

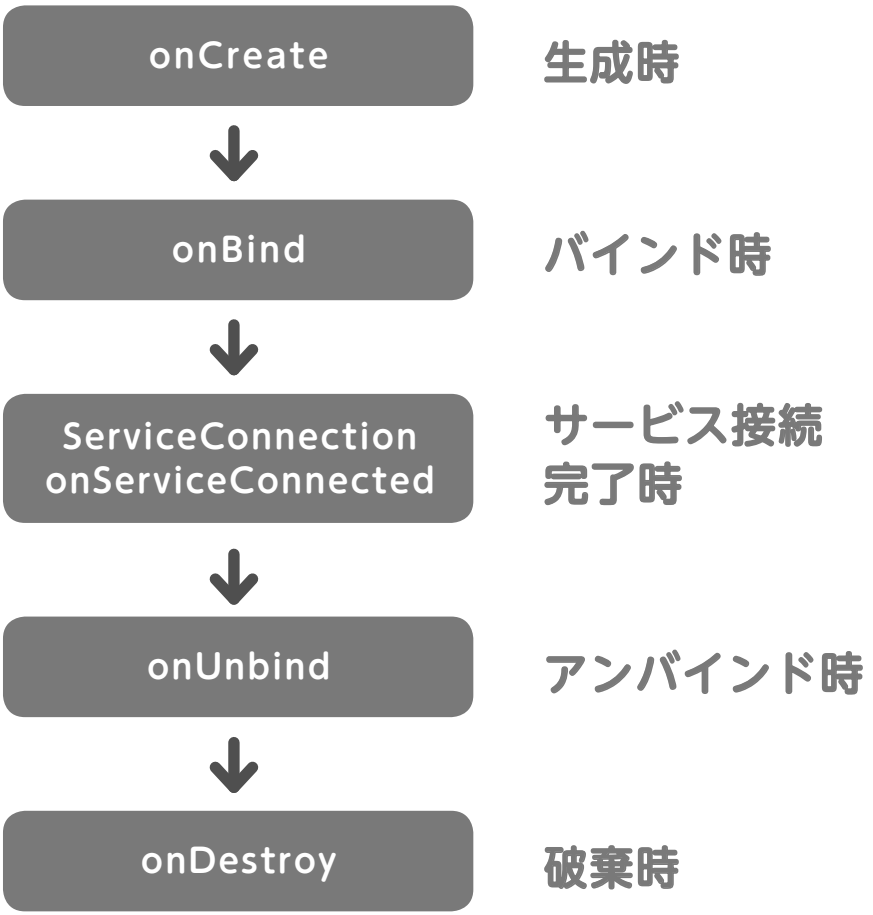


図6: Serviceのライフサイクル（Bind時）

通常の「onCreate」「onDestroy」メソッドに加えて、バインド時は次のコールバックメソッドを利用します。

- **public abstract IBinder onBind (Intent intent)**
- **public boolean onUnbind (Intent intent)**
- **public void onRebind (Intent intent)**

「onBind」はバインド（接続時）、「Unbind」はバインド解除（切断時）に呼び出されます。「onRebind」メソッドは図中には現れていませんが、「Unbind」後に再接続する場合、「onRebind」メソッドを利用できます。使用には「onUnbind」メソッドの返り値を「true」に設定します。

バインドするサービスを作る

「Bind」「Unbind」に対応した「Service」を作ってみましょう。バインドするためには「IBinder」クラスを実装します。バインダーと呼ばれる仕組みで「Service」と「Activity」をつなぐインターフェイスを提供します。サンプルコードは次のとおりです。

MyService.java

```
public class MyService extends Service {
    // onCreate、onStartCommand、onDestroyはstartServiceと同じなので省略
    /*
     * 以下はBind時に追加するコード
     * */
    //サービスに接続するためのBinder
    public class MyServiceLocalBinder extends Binder {
        //サービスの取得
        MyService getService() {
            return MyService.this;
        }
    }
    //Binderの生成
    private final IBinder mBinder = new MyServiceLocalBinder();

    @Override
    public IBinder onBind(Intent intent) {
        Toast.makeText(this, "MyService#onBind"+ ": " + intent, Toast.LENGTH_SHORT).show();
        Log.i(TAG, "onBind" + ": " + intent);
        return mBinder;
    }

    @Override
    public void onRebind(Intent intent){
        Toast.makeText(this, "MyService#onRebind"+ ": " + intent, Toast.LENGTH_SHORT).show();
        Log.i(TAG, "onRebind" + ": " + intent);
    }

    @Override
    public boolean onUnbind(Intent intent){
        Toast.makeText(this, "MyService#onUnbind"+ ": " + intent, Toast.LENGTH_SHORT).show();
        Log.i(TAG, "onUnbind" + ": " + intent);

        //onUnbindをreturn trueでoverrideすると次回バインド時にonRebildが呼ばれる
        return true;
    }
}
```

「Service」をバインドするために「IBinder」を生成して「onBind」メソッドの返り値として設定しています。「onBind」「onRebind」「onUnbind」メソッドそれぞれで動作を確認しやすいように「Toast」を表示します。

「Service」を起動する「Activity」側も変更します。以下のようにボタンを追加して「OnClickListener」に「bindService」「unbindService」メソッドを追加します。

MainActivity.java

```
～省略～
private OnClickListener btnListener = new OnClickListener() {
    public void onClick(View v) {

        switch(v.getId()){

        case R.id.StartButton://startServiceでサービス起動
            startService(new Intent(MainActivity.this, MyService.class));
            break;

        case R.id.StopButton://stopServiceでサービス終了
            stopService(new Intent(MainActivity.this, MyService.class));
            break;

        case R.id.bindButton://サービスをバインドするメソッドを呼び出す
            doBindService();
            break;

        case R.id.unbindButton://アンバインドするメソッドを呼び出す
            doUnbindService();
            break;

        default:
            break;
        }

    }
};
～省略～
```

「doBindService」メソッドと「doUnbindService」メソッドは処理が長くなるため、独自のメソッドを作成しています。この時点では何も処理しないメソッドとして追加してください。

「startService」メソッドの場合と異なり、バインドするためには「ServiceConnection」クラスが必要です。「ServiceConnection」はサービスと接続するための機能を担っています。「Activity」側で「ServiceConnection」クラスのインスタンスを生成して「ServiceConnection」に必要なメソッドを用意します。

- **public void onServiceConnected(ComponentName className, IBinder service)**
- **public void onServiceDisconnected(ComponentName className)**

「onServiceConnected」メソッドは「Activity」と「Service」のコネクションが確立した際に呼び出されます。引数の「IBinder service」で、サービスのバインダーを受け取れます。バインダー経由で「Service」のインスタンスを取得することが可能です。つまり「Activity」から直接「Service」側のメソッド呼び出しができるようになり

ます。

「ServiceConnection」の「onServiceDisconnected」メソッドは、プロセスのクラッシュなど意図しないサービスの切断が発生した場合に利用されます（呼び出されること自体、あまり好ましい状況ではありません）。

MainActivity.java

```
public class MainActivity extends Activity {

    //取得したServiceの保存
    private MyService mBoundService;
    private boolean mIsBound;

    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {

            // サービスとの接続確立時に呼び出される
            Toast.makeText(MainActivity.this, "Activity:onServiceConnected",
                    Toast.LENGTH_SHORT).show();

            // サービスはIBinder経由でアクセス可能になる
            mBoundService = ((MyService.MyServiceLocalBinder) service).getService();
            // 必要であればmBoundServiceを使ってバインドしたサービスのメソッドを呼び出す
        }

        public void onServiceDisconnected(ComponentName className) {
            // サービスとの切断（異常系処理）
            // プロセスのクラッシュなど意図しないサービスの切断が発生した場合に呼ばれる。
            mBoundService = null;
            Toast.makeText(MainActivity.this, "Activity:onServiceDisconnected",
                    Toast.LENGTH_SHORT).show();
        }
    };

    void doBindService() {
        // サービスとの接続を確立する。明示的にServiceを指定
        // (特定のサービスを指定する必要がある。
        // 他のアプリケーションから知ることができない =ローカルサービス)
        bindService(new Intent(MainActivity.this,
                MyService.class), mConnection, Context.BIND_AUTO_CREATE);
        mIsBound = true;
    }

    void doUnbindService() {
        if (mIsBound) {
            // コネクションの解除
            unbindService(mConnection);
            mIsBound = false;
        }
    }
}
```

このサンプルコードの「doBindService」メソッドはサービスにバインドするためのメソッドです。「doUnbindService」メソッドはサービスをアンバインド（切断）するメソッドです。

「bindService」と「unbindService」メソッドどちらにも引数に「ServiceConnection」クラスを利用しています。「ServiceConnection」では、サービスとの接続時、切断時の処理を記述しています。今回のサンプルではどちらも「Toast」を使ってログを画面上に表示します。

最後にサンプルアプリを起動して「Bind Service」→「Unbind Service」の順番でボタンを押してみましょう。ログでライフサイクルを確認できます。

ログの例

```
INFO/LocalService(1619): onCreate
INFO/LocalService(1619): onBind: Intent {
  cmp=jp.androidopentextbook.background.sample04/.MyService }
INFO/LocalService(1619): onUnbind: Intent {
  cmp=jp.androidopentextbook.background.sample04/.MyService }
INFO/LocalService(1619): onDestroy
```

「Service」の利用では「startService」「stopService」メソッドを使う場合と「bindService」「unbindService」メソッドの2つの方法があることを解説しました。前者は「Service」の生存期間が「Activity」に依存しないことから、音楽再生のようにバックグラウンドで常時動いてほしいものが適切です。しかしながら起動後は「Service」を細かく操作できないため「onStartCommand」メソッドの引数をもとに制御することになります。

後者の生存期間は、コネクションに依存します。これはサーバーとの通信など利用者が多い機能（各Activityで共通化した処理）と相性が良いでしょう。そのための「ServiceConnection」が存在していて、「Activity」と密な連携が可能です。

どちらの場合においても「Service」は独自のライフサイクルをもって動作しています。「Activity」や「Fragment」の制御とは切り離した設計が重要です。

12-2-6 IntentService

「IntentService」はアプリケーションのバックグラウンドで処理を行うための方法です。名前のとおり「Service」を拡張していますので、基本的な扱い方は「Service」と同様です。フォアグラウンドの動作と関係なく（非同期で）動作できますが、そのために専用のサービスを用意すると煩雑になってしまいます。

「IntentService」を使えば、これらの処理を簡略化、簡単に実行することができます。「AsyncTask」もAndroidアプリケーションのUIスレッドを保護する手法の一つです。動作は似ていますが、「IntentService」はアプリケーションの動作状況に依存しないで、自分の仕事がなくなるまでバックグラウンドで処理を行える利点があります。アプリケーションがフォアグラウンド（前面）に位置していなくても、処理が中断されることがありません。また「IntentService」は「AsyncTask」と異なり再利用（複数

呼び出し)も可能です。逆に処理中のプログレスバーを表示するなど、そのつど結果をUIコンポーネントを通じて反映する場合は、「AsyncTask」のほうが手軽です。「IntentService」と「AsyncTask」は非同期処理の特性に応じて使い分けるほうがいいでしょう。

ボタンを押すと「IntentService」を実行するサンプルコードで動作を見ていきましょう(**図7**)。

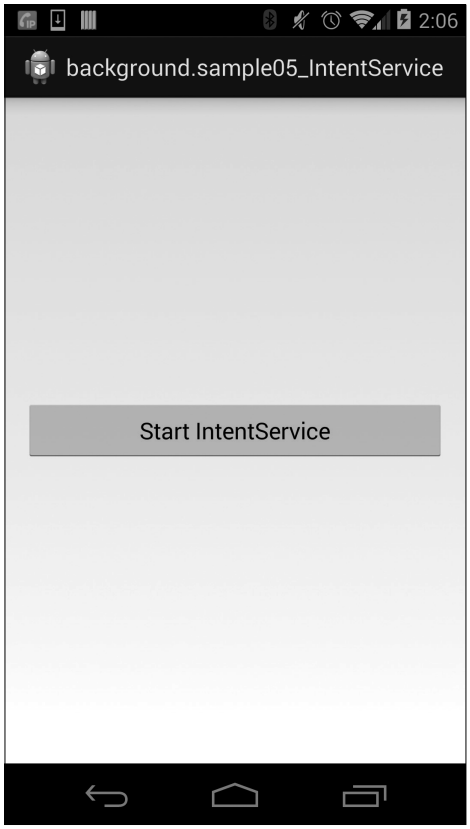


図7: IntentServiceを実行するサンプルアプリ

「IntentService」はサービスですので、以下のように最初に「AndroidManifest」に登録します。

AndroidManifest.xml

```
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="jp.androidopentextbook.background.sample05.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

「service」要素に「MyIntentService」を追加します。

「IntentService」のソースコードは次のとおりです。コンストラクタと非同期処理を行う「onHandleIntent」メソッドのみのシンプルな構成です。

MyIntentService.java

```
public class MyIntentService extends IntentService{

    public MyIntentService(String name) {
        super(name);
    }

    public MyIntentService(){
        // ActivityのstartService(intent);で呼び出されるコンストラクタはこちら
        super("MyIntentService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        // 非同期処理を行うメソッド。タスクはonHandleIntentメソッド内で実行する
        Log.d("IntentService","onHandleIntent Start");
        Log.d("IntentService","intent msg:" +
          intent.getStringExtra("IntentServiceCommand") );
    }
}
```

「IntentService」のコンストラクタが2つありますが、動作には引数がない「MyIntentService()」コンストラクタが必要です。「IntentService」が同時に処理できる作業はひとつで「onHandleIntent」メソッドが非同期処理を行うメソッドです。

「IntentService」の内部では、起動したあと自動的にワーカースレッド（仕事をするために用意された専用のスレッド）によりキューイングが行われています。「Service」はワーカースレッドを通じて処理を逐次実行していきます。

サンプルコードを実行すると、次のログが表示されます。

ログの例

```
DEBUG/IntentService(478): onHandleIntent Start
DEBUG/IntentService(478): intent msg:TestText
```

「IntentService」は、「Service」として動作することで確実にバックグラウンドで動作します。「AsyncTask（またはThreadをつかった非同期処理）」では、アプリケーションが非表示になる際に処理が中断される恐れがありました。確実に実行しなければならない作業では積極的に使っていくべきでしょう。

「AsyncTask」では、アプリケーションが非表示になる際に処理が中断されるため、onPauseメソッドなどで停止処理を組み込んで対応する、もしくはライフサイクルを考慮したAsyncTaskLoaderを使います。