端末電池消費に関するアプリガイドライン

Ver.1.0.0

KDDI 株式会社

2013 年 3 月 26 日

改版履歴

版数	改版内容	日付
1.0.0	新規作成	2013/03/26

1. 目次

2. はじめに

本ガイドラインは、Android のアプリ開発者に向けた端末の電池消費に関する内容を記載したドキュメントです。

2.1. 適用範囲

本ガイドラインは、Android スマートフォンに対応したアプリが対象となります。

2.2. 目的

スマートフォンの電池持ちがエンドユーザとなるお客様の利便性に大きく影響を与えることから、快適な環境をお届けするため、アプリの振る舞いを工夫することが必要不可欠です。
そのため、電池消費に影響を与えるスマートフォンの CPU、カメラ、通信モジュール、LCD、バックライト、各種センサー等のリソースをアプリが利用する際に参考にして頂きたい API の使用方法について紹介します。

2.3. 免責事項

本ガイドラインに基づいて開発されたアプリに関する一切の事項について、いかなる責任も当社は負うものではありません。
これは本ガイドの記載内容があくまで「参考」であり、ガイドラインを採用したことによるアプリの品質について当社は保証しないことを示します。

2.4. 用語

以下に本書に関連する用語を示します。

表 1 用語集

No.	用語	説明
1	GC(ガベージコレクション)	プログラムが動的に確保したメモリ領域のうち、不要になった領域を自動的に解放する機能です。
2	GZIP	GNU ZIP の略であり、Deflate アルゴリズムを用いた汎用のデータ圧縮のことを示します。

2.5. 参考資料

以下に本書に関連するドキュメント、及び参考資料を示します。

表 2 関連ドキュメント一覧

No.	名称	発行元
1	Android Developers Site http://developer.android.com/intl/ja/index.html	Google Inc.

2.6. 商標

「Android」は Google Inc. の商標または登録商標です。
その他、本書に記載されている会社名、製品名、サービス名は各社の商標または登録商標です。
本書ではコピーライト及び商標・登録商標表記はしておりません。

3. 通信リソースの効率的な利用

本章では、通信リソースを効率的に利用することで、電池消費を改善する手法を紹介します。

3.1. 通信頻度を減らす・通信時間を短くする手法

通信リソースを利用すると、電池を消費します。特に電波を送出するには送信電力が必要となり、端末の中で行われる通常の演算処理よりも、多くの電力が必要になります。
そのため、通信リソースの利用頻度を減らし、利用時間を短くすることで、電池消費の改善につながります。
しかし、通信頻度を減らすことでユーザビリティを損ねる可能性があります、たとえば、メールの更新を行う間隔を長くすると電池消費は改善されますが、利用者はメールの到着を間欠的にしか知ることができませんこのようなユーザビリティ上のデメリットがあるため、電池消費とユーザビリティのバランスを取りながら通信頻度・通信時間を制御する必要があります。

この章では、通信頻度を減らす手法と通信時間を短くする手法を紹介します。

3.1.1. 通信間隔の適正化

(1)ガイドライン
通信間隔を広げます。なぜならば、通信間隔を広げると通信リソースの利用回数を減らせるためです。
なお、通信間隔を広げる場合はユーザビリティを担保した上で最適化してください。

(2)ユースケース
一定間隔で通信する（定期的にアプリのバージョンをチェックする等）アプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
アプリが一定間隔で通信処理を行う場合は通信を行うための Thread において
Thread.sleep()により、通信間隔を最適に調整してください。

(5)サンプルコード

```
/** 通信間隔を 10 分に設定するサンプルです。 */
/** Thread.stop の利用は非推奨なのでフラグ制御などで適切に処理してください。 */
private boolean mFlg = true;
/** 必要に応じて通信用のスレッドを開始してください。 */
private class mThread extends Thread{
    @Override
    public void run() {
        while(mFlg){
            /** アプリ毎の処理を追加してください。 */
            try {
                /** 通信間隔の調整処理です。 */
                Thread.sleep(10 * 60 * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

(6)注意点
なし。

3.1.2. Push 通知の利用

(1)ガイドライン
通信契機に Push 通知を利用します。なぜならば、Polling 型よりも Push 通知を利用することでサーバー側との通信回数を最適化できるためです。

(2)ユースケース
通信契機を Push 通知に置き換え可能なアプリ対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
リアルタイムでのデータ受信を必要とするアプリの場合は
通信を行う契機として Push 通知を利用してください。
たとえば Push サービスとしては GCM(Google Cloud Messaging)があります。
参考サイト：http://developer.android.com/intl/ja/google/gcm/index.html

(5)サンプルコード
GCM については Google から詳細な情報が提供されているため、以下を参照してください。
http://developer.android.com/intl/ja/google/gcm/index.html

(6)注意点
なし。

3.1.3. リトライ頻度の適正化

(1)ガイドライン
通信失敗時のリトライや通信頻度を減らします。なぜならば、電波環境が悪化した際、すぐに再接続しても電波環境が改善しておらず、成功する確率は低くなります。接続のリトライも電波送信するために、電池消費に影響を与えます。電波状態が改善したあとに再接続を行うようにすることで、接続が成功する可能性が高くなり、リトライ回数を減らすことができるからです。

(2)ユースケース
通信失敗時に、自動的(ユーザ操作なし)にリトライを試みるアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
アプリが自動的(ユーザ操作なし)にリトライを行う場合は、リトライを行う Thread にて Thread.sleep()を用いて、通信失敗した際のリトライ間隔を広げることで電池消費が改善されます。アプリの仕様によりユーザビリティを損なわないように
1 回目失敗:1 分後、2 回目失敗:10 分後、3 回目失敗:30 分後などリトライの
適正な時間設定をしてください。

(5)サンプルコード

```
/** 1 分に対して moffset 値をかけてリトライ間隔を広げるサンプルです。 */
/** Thread.stop の利用は非推奨なのでフラグ制御などで適切に処理してください。 */
private boolean mFlg = true;
private int mOffset;
/** 必要に応じて通信用のスレッドを開始してください。 */
private class mThread extends Thread{
    @Override
    public void run() {
        while(mFlg){
            /** アプリ毎の処理を追記してください。 */
            /** 通信の結果に従って「mOffset」を調整してください。 */
            try {
                Thread.sleep((1 * mOffset) * 60 * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

(6)注意点
なし。

3.1.4. アプリ状態に応じた制御

(1)ガイドライン
アプリがバックグラウンド時は通信頻度を減らします。
なぜならば、バックグラウンド時にはフォアグランド時であるときほど、リアルタイムな操作を必要としないケースが多いため、ユーザビリティを担保したまま通信頻度を下げることができるからです。

(2)ユースケース
アプリがバックグラウンド時でも通信を行うアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
アプリがバックグラウンドで通信を行う場合は通信頻度を下げてください。
通信頻度はアプリの仕様によりユーザビリティを損なわないように最適化してください。
アプリがバックグラウンド状態であることを以下の手順で確認してください。
①ActivityManager.getRunningTasks(1)と List.get(0)を利用して ActivityManager.RunningTaskInfo を取得する。
②ActivityManager.RunningTaskInfo.topActivity で現在トップのアクティビティのコンポーネント名を取得してください。
③ComponentName.getPackageName()でパッケージ名を取得し、現在トップにあるアクティビティが自アプリのアクティビティでないか判定してください。

(5)サンプルコード

```
/** 通信間隔をフォアグラウンド状態 1 分、バックグラウンド状態 10 分に設定するサンプルです。 */
/** 自アプリのパッケージ名です。 */
private static final String TAG ="jp.co.kddi.battery.sample";
private boolean mFlg = true;
private ActivityManager mAm;
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mAm = (ActivityManager)getSystemService(ACTIVITY_SERVICE);
}
/** Thread.stop の利用は非推奨なのでフラグ制御などで適切に処理してください。 */
/** 必要に応じて通信用のスレッドを開始してください。 */
private class mThread extends Thread{
    @Override
    public void run() {
        while(mFlg){
            /** アプリ毎の処理を追記してください。 */
            try {
                /** アプリの状態を判定、通信頻度の調整処理です。*/
                List<ActivityManager.RunningTaskInfo>
                taskInfo = mAm.getRunningTasks(1);
                if(taskInfo.get(0).topActivity.getPackageName().equals(TAG)){
                    Thread.sleep(1 * 60 * 1000);
                }else{
                    Thread.sleep((1 * 10) * 60 * 1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

[AndroidManifest.xml]
トップアクティビティのコンポーネント名を取得するには android.permission.GET_TASKS を
追加する必要があります。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    〜
    <uses-permission android:name="android.permission.GET_TASKS"/>
</manifest>
```

(6)注意点
なし。

3.1.5. 圏外時の通信開始抑止

(1)ガイドライン
圏外状態の時は通信しないようにします。なぜならば、圏外時の通信は必ず失敗するのに、通信を行おうとして電池を消費するからです。

(2)ユースケース
通信を行うアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
アプリが一定頻度で通信を行う場合はアプリが優先ネットワークの状態を判定し、
圏外である場合は通信を行わないようにしてください。

【WiFi 及び MOBILE 共に圏外になった場合】
ConnectivityManager.getActiveNetworkInfo()で取得できる NetworkInfo が null となるため、これで圏外と判断してください。

また、通信を行うアプリが利用するネットワークを限定する場合は以下の方法で個別に圏外を判定してください。

【WIFI 利用不可(WIFI 圏外 or ユーザーWIFI オフ)の判定方法】
ConnectivityManager.getActiveNetworkInfo()で NetworkInfo を取得し、NetworkInfo.getTypeName()で“MOBILE"が返却されます。

【MOBILE の圏外判定方法】
TelephonyManager.listen()に PhoneStateListener と PhoneStateListener.LISTEN_SERVICE_STATE を指定することで PhoneStateListener.onServiceStateChanged()がコールバックされます。
このメソッドの引数である ServiceState から以下のメソッドを利用することで圏外・圏内の状態を取得することが可能です。
メソッド：ServiceState.getState()
圏内：ServiceState.STATE_IN_SERVICE
圏外：ServiceState.STATE_OUT_OF_SERVICE
。

(5)サンプルコード

```
/** WIFI 及び MOBILE が共に圏外の場合、通信を抑止するサンプルです。 */
/** Thread.stop の利用は非推奨なのでフラグ制御などで適切に処理してください。 */
private boolean mFlg = true;
private ConnectivityManager mCm;
private TelephonyManager mTm;
private String mNst;
private static final String IN_SV = "圏内";
private static final String OUT_SV = "圏外";
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mCm = (ConnectivityManager)getSystemService(CONNECTIVITY_SERVICE);
    mTm = (TelephonyManager)getSystemService(TELEPHONY_SERVICE);
    mTm.listen(mPhoneStateListener, PhoneStateListener.LISTEN_SERVICE_STATE);
}
/** MOBILE の通信状態を取得する処理です。*/
private PhoneStateListener mPhoneStateListener = new PhoneStateListener(){
    @Override
    public void onServiceStateChanged(ServiceState serviceState){
        if(serviceState.getState() == ServiceState.STATE_IN_SERVICE){
            mNst = IN_SV;
        }else{
            mNst = OUT_SV;
        }
    }
};
/** 必要に応じて通信用のスレッドを開始してください。 */
private class mThread extends Thread{
    @Override
    public void run() {
        while(mFlg){
            NetworkInfo info = mCm.getActiveNetworkInfo();
            if(info != null){
                if(info.getTypeName().equals("mobile")){
                    if(mNst.equals(IN_SV)){
                        /** アプリ毎の処理を追記してください。 */
                    }
                /** Wifi の時の処理です。 */
                }else{
                    /** アプリ毎の処理を追記してください。 */
                }
            }else{
                mNst = OUT_SV;
            }
        }
    }
}
```

[AndroidManifest.xml]
優先ネットワークを取得するには android.permission.ACCESS_NETWORK_STATE **を追加する必要があります。**

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    〜
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
</manifest>
```

(6)**注意点**

なし。

3.1.6. HTTP 通信時のデータ圧縮

(1)ガイドライン
HTTP通信を利用する場合は GZIP 圧縮をサーバーに要求することでデータ圧縮を有効にします。なぜならば、データを圧縮することで、通信時間を短くできるためです。

(2)ユースケース
HTTP 通信を行うアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
アプリが HTTP 通信をする場合は、GZIP 圧縮を有効にしてください。
HttpGet.addHeader()に"Accept-Encoding"、"gzip,deflate"を指定してください。
HTTP レスポンス取得は GZIPInputStream.GZIPInputStream()で GZIP のデータの解凍・取得をしてください。
なお、サーバーが GZIP 圧縮に対応していない場合、HTTP レスポンスヘッダに"gzip"が存在しません。
この場合、GZIP 圧縮なしと判断して通常のデータ受信処理をしてください。

(5)サンプルコード

```
/** GZIP 圧縮の指定・受信データの展開をするサンプルです。*/
/** GZIP に対応したサイトを指定する。 */
private static final String URL = "http://www.XXX.com";
/** Thread.stop の利用は非推奨なのでフラグ制御などで適切に処理してください。 */
private boolean mFlg = true;
/** 必要に応じて通信用のスレッドを開始してください。 */
private class mThread extends Thread{
    @Override
    public void run() {
        DefaultHttpClient httpClient = new DefaultHttpClient();
        HttpGet req = new HttpGet(MainActivity.URL);
        /** GZIP 圧縮を有効にする処理です。 */
        req.addHeader("Accept-Encoding", "gzip,deflate");
        while(mFlg){
            try {
                HttpResponse execute = httpClient.execute(req);
                switch (execute.getStatusLine().getStatusCode()) {
                case HttpStatus.SC_OK:
                    InputStream in;
                    if (isGZipHttpResponse(execute)) {
                        /** GZIP 圧縮をサポートしたサーバーからの受信処理です。 */
                        in = new GZIPInputStream(execute.getEntity().getContent());
                    } else {
                        /** GZIP 圧縮をサポートしていないサーバーからの受信処理です。 */
                        in = execute.getEntity().getContent();
                    }
                    /** 読み込んだデータを必要に応じて処理してください。 */
                    break;
                default:
                    break;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
private boolean isGZipHttpResponse(HttpResponse response) {
    Header header = response.getEntity().getContentEncoding();
    if (header == null) {
        return false;
    }
    String value = header.getValue();
    return(value.contains("gzip"));
}
```

```
[AndroidManifest.xml]
通信を行う場合は、android.permission.INTERNET パーミッションを追加する必要が
あります。

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    〜
    <uses-permission android:name="android.permission.INTERNET"/>
</manifest>
```

.

(6)注意点

対象サイトが GZIP 圧縮に対応している場合に限ります。

3.2. 通信を行うタイミングを複数のアプリで同期させる手法

通信を行うアプリが複数存在する場合、アプリがそれぞれのタイミングで通信を開始すると電池を多く消費します。なぜならば、通信の要求があるごとに通信リソースが利用され、通信モジュールが継続して利用状態となる可能性があります。たとえば、A、B、C の各アプリが 10 分間隔(非同期)で各々1 分の通信する場合、全てが重ならないケースにおいて合計で 3 分の端末の通信リソースを占有します。しかし、同期がとれていれば、1 分間の通信リソースを占有するだけにとどまります。
そのため、消費電力を減らす（端末の通信リソースの利用時間を短縮）には、複数のアプリでタイミングを合わせることが効果的です。

この章では、一定頻度で通信を行うアプリを対象に複数のアプリで通信タイミングを同期させる手法を紹介します。

3.2.1. 複数アプリによる同期の通信

(1)ガイドライン
複数の通信アプリの通信開始タイミングを同期させます。通信を同期して行うと端末の通信リソースの利用時間を短くできるためです。

(2)ユースケース
一定頻度で通信する（定期的にアプリのバージョンをチェックする等）アプリに対して、適用を推奨します。

(3)適用可能 AndroidOS バージョン
1.5 以上(API Level 3)

(4)実現方法
通信開始タイミングに厳密な精度を求めないアプリは、
AlarmManager.setInexactRepeating()に AlarmManager.RTC_WAKEUP を使用し、通信タイミングを同じ API を利用した他のアプリと同期させてください。
AlarmManager.RTC_WAKEUP 設定は RTC(Real Time Clock)の時刻を基準にアプリにインテントを通知します。また、スリープ中に AlarmManager のタイマーが満了した場合は、スリープが解除されてアプリにインテントが通知されます。

AlarmManager.setInexactRepeating()の引数：intervalMillis には以下の指定が可能です。
・AlarmManager.INTERVAL_FIFTEEN_MINUTES：15 分
・AlarmManager.INTERVAL_HALF_HOUR：30 分
・AlarmManager.INTERVAL_HOUR：60 分
・AlarmManager.INTERVAL_HALF_DAY：12 時間
・AlarmManager.INTERVAL_DAY：24 時間

(5)サンプルコード

```
/** AlarmManager に 15 分間隔でインテント送信設定を行うサンプルです。*/
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    AlarmManager AlrMng = (AlarmManager)getSystemService(ALARM_SERVICE);
    Intent intent = new Intent(this, SampleBCReceiver.class);
    PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0, intent,
                                        PendingIntent.FLAG_UPDATE_CURRENT);
    long currentTimeMillis = System.currentTimeMillis();
    /** PendingIntent を 15 分間隔で送信する設定処理です。*/
    AlrMng.setInexactRepeating(AlarmManager.RTC_WAKEUP,
                                        currentTimeMillis + 1000,
                                        AlarmManager.INTERVAL_FIFTEEN_MINUTES,
                                        pendingIntent);
}
```

```
/** AlarmManager からインテント受信する側のサンプルです。 */
public class SampleBCReceiver extends BroadcastReceiver{
    @Override
    public void onReceive(Context context, Intent intent) {
        mThread thread = new mThread();
        thread.start();
    }
    private class mThread extends Thread{
        @Override
        public void run() {
            /** アプリ毎の処理を追記してください。 */
        }
    }
}
```

(6)注意点

・AlarmManager.setInexactRepeating()に指定可能な AlarmManager.RTC は、スリープ中にインテントがアプリに通知されませんが、通信環境上、好ましくないタイミングでアプリにインテントが通知される恐れがあり、エンドユーザとなるお客様の快適な利用を損なう可能性があるため、極力利用を避けてください。

・インテント通知の時刻指定は、通信環境が混み合い快適な利用を損なう可能性があるため、こちらも極力利用を避けてください。

3.3. 通信を行うタイミングを複数の端末で分散させる手法

複数の端末が同じタイミングで通信すると回線が混み合うため、1回の通信時間が長くなり、電池を多く消費する可能性があります。たとえば、一定時刻に通信をするアプリが複数の端末にインストールされているケースでは、その全ての端末で同時刻に通信が発生すると通信回線が混み合います。

この章では、一定頻度で通信を行うアプリを複数の端末間で通信タイミングを分散させる手法を紹介します。

3.3.1. 乱数による分散

(1)ガイドライン
通信頻度にランダムなオフセット値を付加します。複数端末の通信タイミングを分散させることによって快適な通信をできる可能性が高くなります。

(2)ユースケース
一定頻度で通信するアプリに対して、適用を推奨します。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
通信開始タイミングに厳密な精度を求めないアプリは、
通信を行うための Thread において Thread.sleep()に
Random.nextInt()の乱数値をオフセットとして利用し、通信開始時間を分散させてください。

(5)サンプルコード

```
/** 通信間隔に 10 秒以下のランダム値を設定するサンプルです。*/
/** Thread.stop の利用は非推奨なのでフラグ制御などで適切に処理してください。 */
private boolean mFlg = true;
/** 必要に応じて通信用のスレッドを開始してください。 */
private class mThread extends Thread{
    @Override
    public void run() {
        while(mFlg){
            /** アプリ毎の処理を追記してください。 */
            Random Rndm = new Random();
            int mRdmOffset = Rndm.nextInt(10);
            try {
                /** 10 秒以下のランダム値を設定する処理です。 */
                Thread.sleep(mRdmOffset * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

(6)注意点
なし。

3.3.2. アラームマネージャによる分散

(1)ガイドライン
通信の開始契機にアラームマネージャを利用します。
なぜならば、それぞれの端末の起動時を起点として通信タイミングを分散できるため、快適な通信をできる可能性が高くなります。

(2)ユースケース
一定頻度で通信するアプリに対して、適用を推奨します。

(3)適用可能 AndroidOS バージョン
1.5 以上(API Level 3)

(4)実現方法
インテントを通信開始契機とする場合、AlarmManager.setInexactRepeating()に AlarmManager.ELAPSED_REALTIME_WAKEUP を使用してください。
これにより処理を開始するためのインテントの送信時間を分散させ、通信開始タイミングを分散させてください。
AlarmManager.ELAPSED_REALTIME_WAKEUP 設定は端末の起動時間(スリープ時間を含む)を基準にアプリにインテントを通知します。
また、スリープ中に AlarmManager のタイマーが満了した場合は、スリープが解除されてアプリにインテントが通知されます。

AlarmManager.setInexactRepeating()の引数：intervalMillis には以下の指定が可能です。
・AlarmManager.INTERVAL_FIFTEEN_MINUTES：15 分
・AlarmManager.INTERVAL_HALF_HOUR：30 分
・AlarmManager.INTERVAL_HOUR：60 分
・AlarmManager.INTERVAL_HALF_DAY：12 時間
・AlarmManager.INTERVAL_DAY：24 時間

(5)サンプルコード

```
/** AlarmManager に 15 分間隔でインテントを送る設定を行うサンプルです。*/
@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      AlarmManager AlrMng = (AlarmManager)getSystemService(ALARM_SERVICE);
      Intent intent = new Intent(this, SampleBCReceiver.class);
      PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0, intent,
                                                PendingIntent.FLAG_UPDATE_CURRENT);
      long currentTimeMillis = System.currentTimeMillis();
      /** PendingIntent を 15 分間隔で送信する設定処理です。 */
      AlrMng.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                                 currentTimeMillis + 1000,
                                 AlarmManager.INTERVAL_FIFTEEN_MINUTES,
                                 pendingIntent);
}
```

```
/** AlarmManager からインテントを受信する側のサンプルです。 */
public class SampleBCReceiver extends BroadcastReceiver{
      @Override
      public void onReceive(Context context, Intent intent) {
            mThread thread = new mThread();
            thread.start();
      }
      private class mThread extends Thread{
            @Override
            public void run() {
                  /** アプリ毎の処理を追記してください。 */
            }
      }
}
```

(6)注意点

・AlarmManager.setInexactRepeating()に指定可能な AlarmManager.ELAPSED_REALTIME は、スリープ中にインテントがアプリに通知されませんが、通信環境上、好ましくないタイミングでアプリにインテントが通知される恐れがあり、エンドユーザとなるお客様の快適な利用を損なう可能性があるため、極力利用を避けてください。

・インテント通知の時刻指定は、通信環境が混み合い快適な利用を損なう可能性があるため、こちらも極力利用を避けてください。

3.3.3. 端末ロック解除による分散

(1)ガイドライン
端末ロック中(画面オフ含む)に通信を保留している場合は、通信再開タイミングを端末ロック解除にします。
なぜならば、端末ロック解除はエンドユーザであるお客様個々の意図した操作によって任意の時間に発生するため、快適な通信をできる可能性が高くなります。

(2)ユースケース
端末ロック時は通信を保留し、ロック解除時に通信を再開するようなアプリに対して、適用を推奨します。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
スリープ時に通信を保留するようなアプリは通信を再開するタイミングを端末ロック解除にしてください。
端末ロック解除は BroadcastReceiver で Intent.ACTION_USER_PRESENT を受け取り判断してください。

(5)サンプルコード

```
/** 端末ロック解除通知を受信するサンプルです。 */
@Override
protected void onResume() {
    super.onResume();
    IntentFilter filter = new IntentFilter();
    /** 端末ロック解除をインテントフィルタに登録する処理です。 */
    filter.addAction(Intent.ACTION_USER_PRESENT);
    registerReceiver(mUserPresentBCReceiver, filter);
}

@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(mUserPresentBCReceiver);
}

private BroadcastReceiver mUserPresentBCReceiver = new BroadcastReceiver(){
    @Override
    public void onReceive(Context context, Intent intent) {
        String a = intent.getAction();
        /** 端末ロック解除インテントの判定処理です。 */
        if(a.equals(Intent.ACTION_USER_PRESENT)){
            /** アプリ毎の処理を追記してください。 */
        }
    }
};
```

(6)注意点
なし。

4. 液晶バックライトの効率的な利用

本章では、液晶バックライトの点灯制御と輝度制御を利用した電池消費を改善する手法を紹介します。

4.1. 液晶バックライトの点灯制御

通常アプリはバックライトの点灯制御を意識する必要はありません。なぜならば、バックライトの点灯制御はシステム側で行っているからです。
アプリでのバックライトの常時点灯制御は必要性を十分検討の上、極力利用を避けてください。
なぜなら、ゲームなどのアプリでバックライト常時点灯されていると、ゲームプレイ中以外の画面でも消灯できなるためです。
具体的には以下の API などを利用する場合が該当します。
①ウェイクロックの取得：PowerManager.newWakeLock
②バックライト常時点灯：Window.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)
ゲームプレイ中以外の画面で、常時点灯を解除することにより電池消費を改善する余地があります。

4.2. 液晶バックライトの輝度を下げる手法

液晶バックライトは輝度設定によって電池消費が変化します。なぜならば、輝度設定によってバックライトが必要とする電力が異なるためです。たとえば、単純にゲームなどバックライトを点灯するアプリでも輝度を下げることで電池消費を改善する余地があります。

本章ではアプリによるバックライトの輝度を下げる手法を紹介します。

4.2.1. アプリ状態に応じた輝度制御

(1)ガイドライン
バックライトの輝度を低く設定します。なぜならば、輝度を低くすることでバックライトが利用する電池消費を下げることができるからです。

(2)ユースケース
大容量のデータをダウンロード中や、Bluetooth のデバイス探索中等、長時間にわたり画面遷移が発生しないシーンを有するアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.5 以上(API Level 3)

(4)実現方法
長時間にわたり画面遷移が発生しないシーンにおいて、輝度を落とすことを許容されているアプリはユーザビリティを損なわない範囲で、輝度を調整してください。

Activity.getWindow()で Window を取得してください。
Window.getAttributes()を利用して WindowManager.LayoutParams を取得してください。
LayoutParams.screenBrightness()メソッドに輝度を設定してください。（輝度：0.0（暗い）～1.0（明るい））
上記で取得した Window.setAttributes()に WindowManager.LayoutParams.screenBrightness を指定して設定した輝度を設定してください。

(5)サンプルコード

```
/** 輝度の変更処理をメソッド化したサンプルです。 */
private void ChgBrightness(float param){/** 引数は 0.0 から 1.0 を指定してください。 */
    Window win = getWindow();
    WindowManager.LayoutParams layoutParam = win.getAttributes();
    layoutParam.screenBrightness = param;
    win.setAttributes(layoutParam);
}
```

(6)**注意点**

輝度は機種によって設定値が同じでも明るさが異なる可能性がある為、確認の上でユーザビリティを損なわない範囲で設定してください。

5. 各種デバイスの効率的な利用

本章では、各種デバイス（加速度センサー、地磁気センサー、GPS、カメラなど）を効率的に利用することで電池消費を改善する手法を紹介します。

5.1. 各種センサーの利用時間を短くする手法

各種センサーを利用すると電池を消費します。なぜならば、デバイスからデータを受け取るための処理が必要となるからです。たとえば、地磁気センサーを利用する場合はデバイスから定期的にデータを受け取るための処理が発生します。

この章では、各種センサーを効率的に利用する方法を紹介します。

5.1.1. データ取得間隔の適正化

(1)ガイドライン

センサーからのデータ取得間隔について制限がない場合はできるだけ長くします。なぜならば、センサーの取得タイミングごとに端末が省電力状態から起動状態となるため、データ取得間隔が短いと省電力の時間が短くなってしまうためです。

※データ取得に厳密なリアルタイム性が要求されるアプリは除きます。

(2)ユースケース

センサーから定期的に情報を取得して画面に反映するアプリに対して、適用を推奨します。

(3)適用可能 AndroidOS バージョン

1.5 以上(API Level 3)

(4)実現方法

各種センサーから定期的にデータを取得したい場合、SensorManager クラスの registerListener()メソッドを使用してデータ取得用のリスナーとデータ取得間隔を指定してください。

データ取得間隔を指定するための定数と取得間隔の関係は下記表の通りです。

表 3 センサー取得間隔

SensorManager の定数	取得間隔
SENSOR_DELAY_FASTEST	0msec
SENSOR_DELAY_GAME	20msec
SENSOR_DELAY_NORMAL	60msec
SENSOR_DELAY_UI	200msec

取得間隔は SENSOR_DELAY_FASTEST が最短で、SENSOR_DELAY_UI が最長です。

(5)サンプルコード

```
/** センサーに取得間隔を設定するサンプルです。*/
private SensorManager mSm;
private Sensor mSs;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mSm = (SensorManager)getSystemService(SENSOR_SERVICE);
    mSs = mSm.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
}
@Override
protected void onResume() {
    super.onResume();
    mSm.registerListener(mSensorEventListener,
    /** センサーの取得頻度の設定処理です。 */
    mSs, SensorManager.SENSOR_DELAY_UI);
}
@Override
protected void onStop() {
    super.onStop();
    mSm.unregisterListener(mSensorEventListener);
}
private SensorEventListener mSensorEventListener = new SensorEventListener (){
    @Override
    public void onSensorChanged(SensorEvent event) {
        if(event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD){
            /** 磁気センサーのデータ取得方法です。 */
            float x = event.values[0];
            float y = event.values[1];
            float z = event.values[2];
        }
    }
    @Override
    public void onAccuracyChanged(Sensor arg0, int arg1) {
    }
};
```

(6)注意点

上記(4)の表に記載した取得時間は、SensorManager クラスにハードコーディングされている値です。
将来的な OS バージョンにより指定時間が変更される可能性があります。

5.2. 位置測位に GPS デバイス以外を利用する手法

位置測位を行うと電池を消費します。なぜならば、位置測位のために GPS デバイス(GPS 測位)又は通信リソース(ネットワーク測位)が利用され、電池消費するためです。
それぞれの測位の特徴として GPS 測位は環境により概ね高い精度で測位可能ですが、ネットワーク測位より多く電池を消費します。そのため、位置測位を行う場合は必要に応じて使い分ける必要があります。
たとえば、高い精度での位置測位がそれほど求められないアプリの場合はネットワーク測位を利用します。

この章では、効率的に測位を行う手法を紹介します。

5.2.1. ネットワーク測位

(1)ガイドライン
高い精度での位置測位がそれほど求められないアプリの場合はネットワーク位置測位を利用します。なぜならば、ネットワーク測位は GPS 測位と比べて電池消費が少ないからです。

(2)ユースケース
高い精度での位置測位がそれほど求められないアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
高い測位精度を必要としない場合は LocationManager.requestLocationUpdates()の引数：provider にネットワーク測位(LocationManager.NETWORK_PROVIDER)を指定してください。

(5)サンプルコード

```
/** 位置測位方法にネットワーク測位を設定するサンプルです。*/
private LocationManager mLocationMng;
@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      mLocationMng = (LocationManager)getSystemService(LOCATION_SERVICE);
}
@Override
protected void onResume() {
      super.onResume();
      /** ネットワーク測位を 1 分間隔で行う場合の設定処理です。     */
      mLocationMng.requestLocationUpdates(
                                                LocationManager.NETWORK_PROVIDER,
                                                1 * 60 * 1000,
                                                0,
                                                mLocationListener);
}
@Override
protected void onStop() {
      super.onStop();
      mLocationMng.removeUpdates(mLocationListener);
}

private LocationListener mLocationListener = new LocationListener() {
      @Override
      public void onLocationChanged(Location location) {
            double lat = location.getLatitude();
            double lon = location.getLongitude();
      }
      @Override
      public void onStatusChanged(String provider, int status, Bundle extras){}
      @Override
      public void onProviderEnabled(String provider){}
      @Override
      public void onProviderDisabled(String provider){}
};
```

[AndroidManifest.xml]
ネットワーク測位を行う場合は、android.permission.ACCESS_COARSE_LOCATION 追加する必要があります。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      〜
      <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
</manifest>
```

(6)注意点
なし。

5.2.2. 基地局測位

(1)ガイドライン
ネットワーク測位よりも低い精度で問題ない場合は基地局測位を利用します。なぜならば、基地局測位はネットワーク測位と比べて電池消費が少ないからです。

(2)ユースケース
高い精度での位置測位がそれほど求められないアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
高い測位精度を必要としない場合は TelephonyManager.getCellLocation()を利用してください。

(5)サンプルコード

```
/** 位置測位方法に基地局測位を設定するサンプルです。*/
private TelephonyManager mTelMng;
@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      mTelMng = (TelephonyManager)getSystemService(TELEPHONY_SERVICE);
}

/** 基地局測位処理です。 */
private void getTelephonyLocation() {
      CellLocation cellLoc = mTelMng.getCellLocation();
      /** CDMA の場合 */
      if(cellLoc instanceof CdmaCellLocation) {
            int lat = ((CdmaCellLocation)cellLoc).getBaseStationLatitude();
            int lon = ((CdmaCellLocation)cellLoc).getBaseStationLongitude();
      }
}
```

[AndroidManifest.xml]
基地局測位を行う場合は、android.permission.ACCESS_COARSE_LOCATION 追加する必要があります。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      〜
      <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
</manifest>
```

(6)注意点

本サンプルでは CDMA の場合であす。現状の OS バージョンでは LTE での基地局測位はサポートされておりません。

5.3. カメラデバイスの利用時間を短くする手法

カメラデバイスは電池消費が多いデバイスの 1 つです。そのため、このデバイスの利用は可能な限り短くすることが電池消費の改善に効果的です。たとえば、アプリがバックグラウンドになった時にはその都度、カメラデバイスを停止します。

この章では、カメラデバイスの電池消費を改善する手法を紹介します。

5.3.1. 必要最低限の利用制御

(1)ガイドライン
カメラデバイスの利用時間を短くします。なぜならば、カメラデバイスは起動しているだけで多く電池消費するからです。

(2)ユースケース
カメラデバイスを利用するアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
カメラを使用するアプリは、必要な時のみカメラをオンしてください。
たとえばフォアグラウンドになった時、
Activity.onResume()で Camera.open()を呼び出し、カメラデバイスを起動してください。
バックグラウンドになった時 Activity.onPause()で Camera.release()を呼び出し、カメラを停止してください。

(5)サンプルコード

```
/** カメラデバイスをオン･オフするサンプルです。*/
private Camera mCam;
@Override
public void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	setContentView(R.layout.activity_main);
}
@Override
protected void onResume() {
	super.onResume();
	mCam = Camera.open();
}
@Override
protected void onPause() {
	super.onPause();
	/** カメラリソースの開放処理です。 */
	mCam.release();
}
```

```
[AndroidManifest.xml]
カメラデバイスを使用する場合は、android.permission.CAMERA パーミッション
を追加する必要が有ります。
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
	～
	<uses-permission android:name="android.permission.CAMERA"/>
</manifest>
```

(6)注意点
なし。

6. その他の工夫

本章では、描画、演算、ガベージコレクション（GC）軽減、ブロードキャストインテント受信を効率的に利用することで電池消費を改善する手法を紹介します。

6.1. 描画・演算・GC・ブロードキャストインテント受信の工夫による手法

描画処理、小数点演算、描画、ブロードキャストインテント受信処理は適切に行うことは電池消費の改善につながります。なぜならば 1 回 1 回の処理による電力消費は微々たるものですが繰り返し、長い時間に実行されることで無視できない消費電力になる場合があるためです。
たとえば、多くのブロードキャストインテントを受信可能にしている場合、、多くの機会で受信時の処理が実行されるからです。

この章では、描画処理、小数点演算、GC 軽減、ブロードキャストインテント受信に関する手法を紹介します。

6.1.1. 液晶バックライトオフ時の描画抑止

(1)ガイドライン
バックライトオフ時は描画をしないようにします。なぜならば、バックライトオフ時はユーザに画面が見えてないため描画回数を減らすことができるからです。

(2)ユースケース
バックライトオン・オフを意識せず、任意に画面の描画を繰り返し行う可能性があるアプリに対して、適用を推奨します。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
画面描画を行うアプリは、バックライトオフ時の描画を極力しないようにしてください。
たとえば BroadcastReceiver での Intent.ACTION_SCREEN_OFF の受信から
Intent ACTION_SCREEN_ON を受信するまでは画面の描画をしないようにしてください。

(5)サンプルコード

```
/** バックライトの点灯･消灯状態を取得するサンプルです。*/
/** 任意の描画をフラグで抑止する。 */
private boolean isBackLightOFF=false;
private BroadcastReceiver mBroadcastReceiver = new BroadcastReceiver(){
    @Override
    public void onReceive(Context context, Intent intent) {
        String a = intent.getAction();
        /** バックライト点灯･消灯を判定、状態を保持する処理です。 */
        if(a.equals(Intent.ACTION_SCREEN_ON)){
            isBackLightOFF = false;
        }else if(a.equals(Intent.ACTION_SCREEN_OFF)){
            isBackLightOFF = true;
        }
    }
};
@Override
protected void onResume() {
    super.onResume();
    IntentFilter filter = new IntentFilter();
    /** バックライト点灯･消灯をインテントフィルタに登録する処理です。 */
    filter.addAction(Intent.ACTION_SCREEN_ON);
    filter.addAction(Intent.ACTION_SCREEN_OFF);
    this.registerReceiver(mBroadcastReceiver, filter);
}
```

(6)注意点

なし。

6.1.2. ハードウェアアクセラレータの使用

(1)ガイドライン

ハードウェアアクセラレータを有効にします。なぜならば、これによりほとんどの描画において GPU が利用され、描画が高速化・効率化されるからです。

(2)ユースケース

ビュー(View)、ウィジェット(widget)、Canvas などを利用して画面描画を行うアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン

1.0 以上(API Level 1)

(4)実現方法

アプリは AndroidManifest.xml に対して以下の方法でハードウェアアクセラレータを有効にしてください。

・android 4.X 未満の場合は<application>に「android:hardwareAccelerated="true"」を設定してください。

・android 4.X 以上の場合は<application>に特別な理由なく「android:hardwareAccelerated="false"」を設定しないでください。

(5)サンプルコード

[AndroidManifest.xml]

android 4.X 未満の場合は、「android:hardwareAccelerated="true"を追加してください。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    <application
        android:hardwareAccelerated="true"
    〜
    </application>

</manifest>
```

(6)注意点

ハードウェアアクセラレータのオン・オフはアプリで制御可能ですが実際に利用可能であるかは端末に依存します。

6.1.3. レンダリングの実行

(1)ガイドライン
画像エフェクト(カラー⇒モノクロ)などは renderscript の利用を控えます。なぜならば、このようなエフェクトでは renderscript のパフォーマンスが活かせないからです。

(2)ユースケース
画像エフェクトを行うアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
renderscript の使用は控えて、たとえば画像エフェクト(カラー⇒モノクロ)では Canvas を利用してください。

(5)サンプルコード

```
/** Canvas を利用したレンダリング(カラー⇒モノクロ)のサンプルです。*/
private Bitmap mBitmapOut;
private ColorMatrix colorMtrx;
private Paint paint;
@Override
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      /** 任意のレイアウトを指定してください。 */
      LinearLayout layout = (LinearLayout) findViewById(R.id.layout);
      CustomImgViewColor viewColor = new CustomImgViewColor(getApplicationContext());
      layout.addView(viewColor);
      CustomImgViewMono viewMono = new CustomImgViewMono(getApplicationContext());
      layout.addView(viewMono);
      /** 任意のサンプル画像を指定してください。 */
      Bitmap bitmapIn = loadBitmap(R.drawable.sample);
      viewColor.setImageBitmap(bitmapIn);
      colorMtrx = new ColorMatrix();
      paint = new Paint();
      /** 任意のサンプル画像を指定してください。 */
      mBitmapOut = loadBitmap(R.drawable.sample);
      viewMono.setImageBitmap(mBitmapOut);
}

private class CustomImgViewColor extends ImageView {
      public CustomImgViewColor(Context context) {
            super(context);
      }
      @Override
      protected void onDraw(Canvas canvas) {
            super.onDraw(canvas);
      }
}
private class CustomImgViewMono extends ImageView {
      public CustomImgViewMono(Context context) {
            super(context);
      }
      @Override
      protected void onDraw(Canvas canvas) {
            colorMtrx.setSaturation(0);
            ColorMatrixColorFilter cmcf = new ColorMatrixColorFilter(colorMtrx);
            paint.setColorFilter(cmcf);
            canvas.drawBitmap(mBitmapOut,75,0, paint);
      }
}
private Bitmap loadBitmap(int resource) {
      final BitmapFactory.Options options = new BitmapFactory.Options();
      options.inPreferredConfig = Bitmap.Config.ARGB_8888;
      return BitmapFactory.decodeResource(getResources(), resource, options);
}
```

(6)注意点

なし。

6.1.4. 固定小数点演算の回避

(1)ガイドライン
小数点演算は浮動小数点演算を利用します。なぜならば、浮動小数点演算は固定小数点演算に比べて高速であるため、効率よく処理を実行できるからです。

(2)ユースケース
浮動小数点演算による演算精度がアプリの仕様、要件で許容されていて、かつ、頻繁に小数点演算を行うアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
double や、float を利用した浮動小数点演算を利用し、BigDecimal による固定小数点演算の利用は控えてください。

(5)サンプルコード

```
/** 任意のメソッドでの浮動小数点演算例のサンプルです。 */
double    x=1.111111111;
double    y=1.111111111;
double    z;
z = x + y;
```

(6)注意点
なし。

6.1.5. GC (ガベージコレクション) の軽減

(1)ガイドライン
クラスインスタンス生成は必要なだけ行い、可能な限り生成済みのインスタンスを利用します。なぜならばこれにより GC 実行回数が減るからです。

(2)ユースケース
クラスインスタンスを使いまわすことができるアプリ(たとえば、画像のレンダリング等で、座標計算や矩形表示を頻繁に行うアプリ)に対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
画像のレンダリング等で、座標計算や矩形表示を頻繁に行うアプリは以下の方法で処理をしてください。
Point、Rect は View を利用し、view.onDraw()を再利用してください。

(5)サンプルコード

```
/** Paint、RectF 利用した GC 軽減のサンプルです。*/
private Paint mPaint;
private RectF mRect;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    /** 任意のレイアウトを指定してください。 */
    LinearLayout layout = (LinearLayout) findViewById(R.id.layout);
    CustomView view = new CustomView(getApplicationContext());
    layout.addView(view);
    /** 一度だけインスタンスを生成するようにしてください。 */
    mPaint = new Paint();
    mRect = new RectF();
}

private class CustomView extends View {
    public CustomView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }
    public CustomView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    public CustomView(Context context) {
        super(context);
    }
    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        mRect.right = getWidth();
        mRect.bottom = getHeight();
        mPaint.setColor(Color.MAGENTA);
        canvas.drawRoundRect(mRect, 10f, 10f, mPaint);
    }
}
```

(6) 注意点
なし。

6.1.6. ブロードキャストインテントの制御

(1)ガイドライン
ブロードキャストインテントの受信を少なくします。なぜならば、ブロードキャストインテント受信を行うとその都度処理が実行されるからです。

(2)ユースケース
アプリの状態により必要なインテントが異なる（たとえば、A画面表示中は必要だが、B画面表示中は不要、等）シーンを有するアプリに対して、適用してください。

(3)適用可能 AndroidOS バージョン
1.0 以上(API Level 1)

(4)実現方法
アプリの状態により必要なインテントが異なる（たとえば、A画面表示中は必要だが、B画面表示中は不要、等）場合は、AndroidManifest.xml における intent-filter 定義には常に通知が必要な Intent のみを定義し、特定のシーンでのみ通知必要なインテントは
Context.registerReceiver()/unregisterReceiver()を使用して動的に受信するブロードキャストインテントを指定してください。

(5)サンプルコード

```
/** インテント送信側のサンプルです。 */
/** アプリ毎の任意のインテントを指定してください。 */
private static final String SAMPLE_BC_ACTION = "jp.co.kddi.battery.sample.SampleAction";
private IntentFilter filter;
private SampleBCReceiver MyReceiver;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    MyReceiver = new SampleBCReceiver();
    filter = new IntentFilter(SAMPLE_BC_ACTION);
}
@Override
protected void onResume() {
    super.onResume();
    registerReceiver(MyReceiver, filter);
}
@Override
protected void onPause() {
    super.onPause();
    /** 該当アクティビティを抜ける時にレシーバーを削除する処理です。 */
    unregisterReceiver(MyReceiver);
}
```

```
public class SampleBCReceiver extends BroadcastReceiver{
    private static final String SAMPLE_BC_ACTION = "jp.co.kddi.battery.sample.SampleAction";
    @Override
    public void onReceive(Context context, Intent intent){
        String a = intent.getAction();
        if(a.equals(SAMPLE_BC_ACTION)){
            /** アプリ毎の処理を追記してください。 */
        }
    }
}
```

(6)注意点
なし。

7. 付録

7.1. 電池レベルを取得する方法

本章では、アプリによる電池レベルの取得方法を紹介します。
各アプリで電池レベルを意識して処理を改善する場合にご参考ください。
電池レベルの取得は BroadcastReceiver で Intent.ACTION_BATTERY_CHANGED を受け取り、intent.getIntExtra(BatteryManager.EXTRA_LEVEL,0)メソッドを利用して取得してください。取得できる値は 0～100(%)です。

【サンプルコード】

```
/** 電池レベルを取得するサンプルです。 */
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}

@Override
protected void onResume() {
    super.onResume();
    /** 電池レベル通知の登録処理です。 */
    IntentFilter filter = new IntentFilter();
    filter.addAction(Intent.ACTION_BATTERY_CHANGED);
    registerReceiver(mBroadcastReceiver, filter);
}

@Override
protected void onStop() {
    super.onStop();
    unregisterReceiver(mBroadcastReceiver);
}

private BroadcastReceiver mBroadcastReceiver = new BroadcastReceiver(){
    @Override
    public void onReceive(Context context, Intent intent) {
        String a = intent.getAction();
        /** インテントの種別を判定し、電池レベルを取得する処理です。 */
        if(a.equals(Intent.ACTION_BATTERY_CHANGED)){
            int blv = intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 0);
        }
    }
};
```

以上