Android: Programando para la vida (de la batería)

Resumen de la conferencia de Jeffrey Sharkey en Google I/O 2009

En el desarrollo de aplicaciones para móviles hay que tener en cuenta tres cosas a la hora de desarrollar aplicaciones: la vida de la batería, la vida de la batería y la vida de la batería. Después de todo, si la batería está agotada, nadie podrá utilizar tu aplicación. En este documento se describe cómo afectan las distintas redes a la vida de la batería, las formas correctas e incorrectas de utilizar características específicas de Android como los "wake locks", el por qué no puedes asumir que está bien consumir más memoria para ahorrar tiempo y más.

Contenido

más

Introducción

¿Por qué es importante tener en cuenta el consumo de batería?

Desde que desconectamos nuestro movil del cargador por la mañana y a lo largo del día, para funcionar, necesita consumir la energía almacenada. Cuando esta se termina, ya no se volverá a poder utilizar hasta que se vuelva a conectar. Las aplicaciones necesitan trabajar juntas para hacer un buen uso de los recursos a los que tienen acceso.

Utilizaremos las siguientes unidades:

- Gasto que se hace de energía: mA
- Capacidad: mAh

Veamos la capacidad de la batería de algunos dispositivos con Android:

- HTC Dream: 1150 mAh
- HTC Magic: 1350 mAh
- Samsung I7500: 1500 mAh
- Asus Eee PC: 5800 mAh

A mayor capacidad, mayor tamaño de batería, así que hay que escoger una buena relación tamaño / usabilidad. Al fin y al cabo, un dispositivo portatil deberá ser eso, portatil.

¿En qué se nos va el gasto de batería?

- Modo avión: 2 mA
- Modo espera 3G / EDGE: 5 mA
- Modo espera WIFI: 12 mA
- LCD normal: 90 mA
- CPU 50% - 100%: 110 mA
- Sensores: 80 mA
- GPS: 85 mA
- 3G transferencia máxima: 150 mA
- EDGE transferencia máxima: 250 mA
- WIFI transferencia máxima: 275 mA

Ejemplos de la vida real

Suponiendo que tenemos una batería con una capacidad de 1150mAh. Si dividimos este valor entre el consumo que tiene cada actividad podremos estimar la duración de la batería realizando esa actividad de forma contínua.

- Ver Youtube: 340 mA = 3,4 horas

- Navegar por la Web usando 3G: 225 mA = 5 horas
- Uso típico: 42 mA (de media) = 32 horas
- EDGE en modo espera: 5 mA = 9,5 días
- Modo avión en espera: 2 mA = 24 días

¿Qué es lo que más consume?

En todo lo alto de la lista están los procesos que se ejecutan cuando el teléfono está en modo espera. Tomemos como ejemplo una aplicación que se despierta cada 10 minutos para actualizar sus datos y que tarda 8 segundos en realizar esta labor con un consumo de 350 mA.

Consumo estimado en una hora:

3600 segundos * 5 mA = **5 mAh (en espera)**
6 veces * 8 segundos * 350 mA = **4,6mAh (actualizándose)**

Como se puede observar, esta inocente aplicación consume casi lo mismo actualizándose durante 48 segundos que durante 1 hora en modo espera.

Y esto no es todo, cada vez que actives tu servicio, desencadenarás la ejecución de otros (para aprovechar la activación de los elementos del dispositivo, que se encuentra en modo espera). Al final, despertamos a nuestro dispositivo para ejecutar un servicio (aplicación) durante 8 segundos y permanece trabajando (en backgrond) entre 15 y 20 segundos.

Otras cosas que consumen bastante batería son las transferencias largas de datos. Veamos el ejemplo de descargar una canción de 6MB:

- **EDGE** (90 kbps): 300 mA * 9,1 minutos = **45 mAh**
- **3G** (300 kbps): 210 mA * 2,7 minutos = **9,5 mAh**
- **WIFI** (1 Mbps): 330 mA * 48 segundos = **4,4 mAh**

Sacad vuestras propias conclusiones. En este caso, la tarea que menos batería gasta es la que antes finaliza.
(Los valores de consumo son distindos debido a que aquí se está incluyendo el gasto de CPU.)

También se consume bastante batería cuando el movil cambia de celda telefónica (por decirlo fácil: de antena) debido a que el módulo de Radio debe

realizar tareas de asociación con la nueva celda, además de todos los eventos (BroadcastIntents) que se generan en base a esto y que pueden despertar la ejecución de otras aplicaciones.

Para terminar, señalar que parsear XML es bastante costoso, así como utilizar expresiones regulares sin JIT (la máquina virtual dalvik todavía no lo soporta)

Veamos a continuación varios consejos para mejorar nuestra aplicación y conseguir que genere menor gasto.

Redes

Como hemos visto anteriormente, tenemos que intentar minimizar el gasto a la hora de descargar datos de la red. ¿Cómo?

Esperar a una conexión 3G o WIFI

Es mucho mejor disminuir el tiempo de descarga de los datos, así el gasto de batería nos saldrá más barato. Además, podemos hacer una aplicación inteligente que no consuma datos en caso de estar usando una conexión de datos movil en roaming, estableciendo esta acción en su menú de configuración correspondiente.

Veamos el código para hacerlo:

Lo primero es comprobar que hay conexión y que, en caso de ejecutarse este código en un servicio, comprobar que el usuario tiene activada la descarga en background en el menú de Ajustes de sincronización de datos (Esto es nuevo en Android 1.5 Cupcake). Todos los servicios de Google para Android comprueban este valor de configuración antes de utilizar una conexión de datos en background, así que tu aplicación también debería comprobarlo y actuar en consecuencia.

```
ConnectivityManager  mConnectivity;
TelephonyManager mTelephony;

NetworkInfo info = mConnectivity.getActiveNetworkInfo();
if (info == null || !mConnectivity.getBackgroundDataSetting()) return false;
```

Antes de utilizar la conexión de datos, comprobamos el tipo, y en caso de que sea 3G, que no estemos en roaming.

```
// Sólo conectar si hay WIFI o 3G (sin roaming)
int netType = info.getType();
int netSubtype = info.getSubtype();

if (netType == ConnectivityManager.TYPE_WIFI) {
    return info.isConnected();
} else if (netType == ConnectivityManager.TYPE_MOBILE
            && netSubtype == TelephonyManager.NETWORK_TYPE_UMTS
            && !mTelephony.isNetworkRoaming()) {
    return info.isConnected();
} else {
    return false;
}
```

Como consejo, si estamos desarrollando una aplicación que necesita, por ejemplo, subir un vídeo, o un elemento pesado podemos postponer la carga hasta que el teléfono se encuentre en una red 3G o WIFI. Lo normal sería que no hubiera actuación por parte del usuario final, aunque tampoco es malo avisar al usuario de que realizar esa acción consumirá muchísima batería y preguntar si realmente quiere continuar con el proceso de envío.

Utilizar un analizador de datos eficiente

Analizar datos es una tarea bastante común para una aplicación que necesita alimentarse de datos descargados de Internet. Esta tarea requiere un mayor gasto de batería en base al tipo de analizador y al formato de los datos a analizar (XML, JSON, ...)

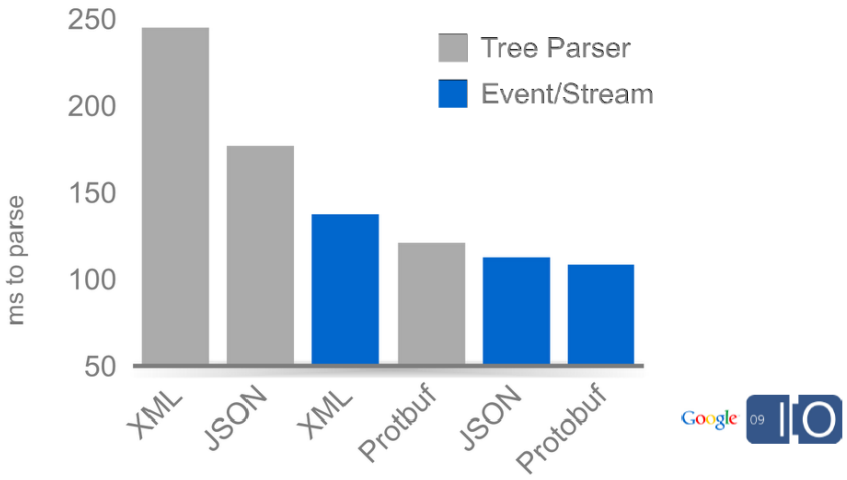
Tenemos dos tipos de analizadores:

- **Analizadores de árboles**. Cargan el documento por completo en memoria manteniendo una estructura en arbol, generan mucha basura que luego el sistema operativo tiene que limpiar, con el consecuente gasto extra de batería. Para realizar esta tarea, hay que analizar uno a uno todos los elementos e ir generando paralelamente la estructura de datos en memoria. Una vez finalizado el análisis el acceso a los elementos en memoria es relativamente rápido, pero... ¿Merece la pena tanto gasto?
- **Analizadores "al vuelo"**. Involucran al programador en el proceso de análisis mediante callbacks. Por ejemplo: "Se ha encontrado el inicio del documento", "tag XXX encontrado con los atributos...". De esta forma, el proceso de análisis puede ser bastante eficiente, ya que sólo se procesan los datos que se necesitan.

Veamos los formatos de datos:

- **XML**. Formato para el intercambio de datos bastante popular en Internet que en dispositivos móviles puede llegar a ser un problema.
- **JSON**. Bastante útil cuando el receptor de esos datos es un motor de Javascript.
- **Protobuf**. Formato de datos binario desarrollado y utilizado por Google. Al ser binario hay muchísima menor sobrecarga añadida a los datos a enviar/recibir.

¿Qué hacer para ahorrar energía? Utilizar un analizador de datos y un formato de datos eficiente.



Tiempos obtenidos descargando y analizando un RSS de 6 elementos repetidamente durante 60 segundos y haciendo una media.

Datos clave:

- Utilizar analizadore "al vuelo" en vez de analizadores de árboles. Por ejemplo, para JSON tenemos el analizador Jackson.
- Tener en cuenta los formatos binarios que pueden facilmente mezclar datos binarios y texto en una sola petición. En la aplicación "Market" se puede ver cómo los iconos de las aplicaciones se van cargando en grupos. Esto es debido a que se unen varias peticiones de imágenes en una sola petición que será la que se haga al servidor, y en cuya respuesta tendremos todas las imágenes solicitadas. (En una sóla respuesta)
- Como hemos visto en el ejemplo del "Market" del punto anterior, utilizando formatos binarios reducimos el número de peticiones que se realicen al servidor, y esto se nota en el consumo de energía.

Utilizar GZIP para texto

Se trata de que el servidor comprima los datos de texto antes de enviarlos al cliente, de esta forma, se reduce sustancialmente el tamaño de datos a descargar y el tiempo de obtención de los mismos. Esto se utiliza en la actualizada en muchos navegadores Web que aceptan recibir las páginas comprimidas.

En Android, la librería GZIP es muy eficiente ya que la descompresión se ejecuta mediante código nativo y es bastante rápida.

Vamos a suponer que en el siguiente código hacemos una petición Web y obtenemos una respuesta comprimida en GZIP.

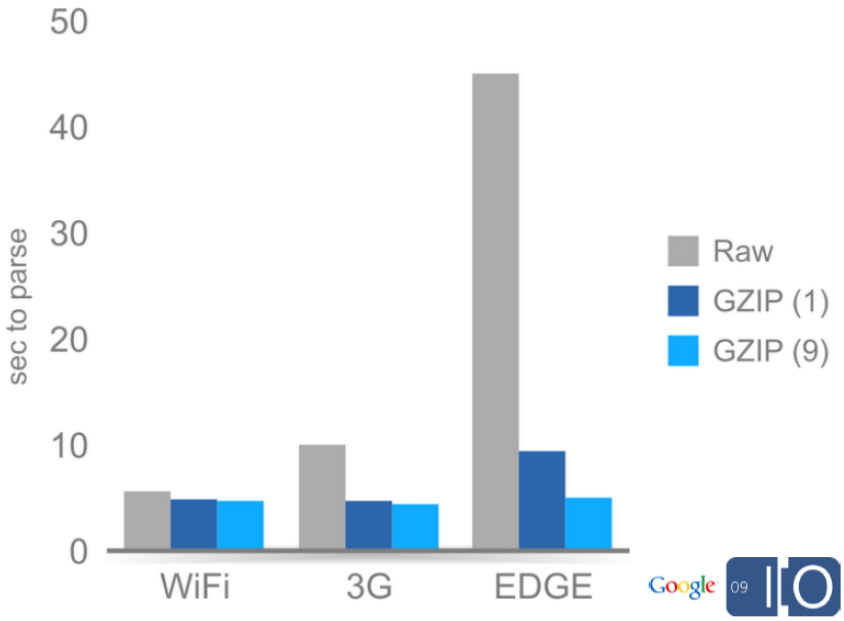
```
import java.util.zip.GZIPInputStream;

HttpGet request = new HttpGet("http://example.com
/contenidogzip");
HttpResponse resp = new DefaultHttpClient().execute(request);
HttpEntity entity = resp.getEntity();
InputStream compressed = entity.getContent();
```

Sólo necesitamos añadir la siguiente línea para trabajar con un InputStream, esta vez, con datos descomprimidos.

```
InputStream rawData = new GZIPInputStream(compressed);
```

¿Cuanto mejoramos al trabajar con datos de texto comprimidos?



Tiempos medidos en múltiples pruebas descargando un RSS de 1800 elementos de texto. El valor que hay entre paréntesis al lado de GZIP es el nivel de compresión, siendo 1 el más bajo y 9 el más alto.

Como se observa en la gráfica, con una conexión rápida casi ni se nota la diferencia entre contenido comprimido o no. Ahora bien, cuando entramos en las conexiones móviles ya empezamos a encontrar una gran diferencia, sobre todo usando EDGE. Como todos ya sabemos, aunque el gasto energético de una conexión EDGE sea inferior a una conexión WIFI, el estar mucho más tiempo descargando/enviando contenido implica un gasto muchísimo mayor.

Aplicaciones en primer plano

Vamos a ver qué podemos hacer con las aplicaciones con las que el usuario interactúa activamente.

WakeLocks

Las aplicaciones de Android pueden hacer que el algun elemento del dispositivo permanezca activo. Para hacer esto podemos obtener un WakeLock del nivel que necesitemos y cuando terminemos, lo liberamos.

Hay varios niveles de WakeLock que fuerzan el estado activo de los siguiente elementos:

- La CPU.
- La pantalla.
- La luz del teclado.

No se deben adquirir WakeLock a menos que realmente lo necesitemos. Hay que utilizar el nivel más bajo posible y asegurarlos de que lo liberamos tan pronto como podamos. Si no lo hacemos, Android pensará que estamos utilizándolo y estaremos consumiendo mucha batería.

Tenemos dos formas de hacer esta tarea correctamente:

1. Utilizar el atributo android:keepScreenOn="true" en el Layout de nuestra actividad. De esta forma, mientras se esté visualizando esta actividad la pantalla permanecerá encendida y cuando dejemos de verla porque pasamos a otra actividad o la cerremos, Android lo liberará automaticamente.
2. En el caso de un WakeLock para mantener activa la CPU, si conocemos cuanto tiempo necesitaremos (el peor caso) ese WakeLock activo para realizar alguna tarea de procesamiento (como parsear un XML) lo iniciaremos indicando ese tiempo de tal modo que Android lo liberará si ve que ha pasado el tiempo establecido y no ha sido liberado.

Reciclar objetos de Java

Otra forma que tenemos de ahorrar batería es reciclando objetos de Java, especialmente los complejos que gestionan búfers en memoria.

Todos sabemos que Android tiene un recolector de basura, pero cuanto menos basura se genere, mayor será el ahorro de energía.

Para reciclar algunos objetos, podemos utilizar clases del Framework de Android que nos ayuden a realizar esta tarea. Por ejemplo:

- Si creamos y destruimos bastantes objetos (que no deberíamos) XmlPullParse o Bitmap, podemos usar las clases XmlPullParserFactory o BitmapFactory que mejorarán la eficiencia de este proceso.

- Si utilizamos expresiones regulares, podemos reciclar el objeto Matcher usando: Matcher.reset(newString)
- Si utilizamos un StringBuilder, podemos reciclarlo (para volver a utilizarlo)

mediante: StringBuilder.setLength(0)

Ya que la interfaz de usuario corre en un sólo hilo, podemos compartir objetos entre los métodos (de los eventos ocurridos en la interfaz) sin riesgo a tener problemas de sincronización.

En un ListView se utilizan muchas estrategias de reciclado de objetos.

Utilizar localización por red telefónica

Es mucho más barato que utilizar localización por GPS. Los tiempos de inicialización del GPS pueden variar mucho dependiendo del entorno en el que nos encontremos, por ejemplo, si estamos en un campo abierto la sincronización con los satélites puede tardar unos segundos, mientras que si estamos en un sitio cerrado puede llegar a no ocurrir nunca y todo el tiempo que tenemos el GPS activado estamos malgastando batería.

De media, estos son los tiempos de inicio:

- GPS: 25 segundos * 140 mA = 1 mAh
- Red: 2 segundos * 180 mA = 0.2 mAh

En Android 1.5 se utiliza AGPS cuando hay red disponible para reducir el tiempo de arranque del GPS. ¿Cómo? En vez de estar atento a recibir la señal de posición de cada satélite, que puede tardar hasta dos minutos, en base a la posición obtenida de la Red, realiza una petición por Internet para conocer la información de los satélites necesaria en vez de tener que esperar a recogerla toda.

El tiempo de sincronización del GPS con los satélites varía dependiendo del entorno y de la precisión que queramos. Con el GPS ocurre igual que con los wake-locks, las actualizaciones de posición pueden continuar después del onPause(), así que asegúrate de detenerlas.

Si todas las aplicaciones dejaran de utilizar los recursos correctamente, como en el caso del GPS, los usuarios lo podrían dejar activado en los ajustes del teléfono.

Utilizar aritmética con número reales (decimales, en punto flotante) sale caro

Los dispositivos actuales no tienen unidad de punto flotante en sus CPUs, de tal forma que hacer operaciones en punto flotante es caro. Si trabajaste con un MapActivity o MapView habrás visto algo llamado GeoPoint. Es como la latitud y longitud, que son número decimales, solo que están multiplicados por 10, de tal forma que podemos tratarlos como números enteros que ya son más baratos para trabajar en el dispositivo.

```
 // Geopoint devuelve el valor 37392778, -122041944
double lat = GeoPoint.getLatitudeE6() / 1E6;
double lon = GeoPoint.getLongitudeE6() / 1E6;
```

Cachear valores cuando se realiza trabajo DPI con DisplayMetrics. Si trabajas en tu propia View, no necesitas preocuparte a la hora de escalar a diferentes densidades de dispositivos. Un dispositivo puede tener una pantalla de mayor densidad y esta densidad estar representada por un número real. Si escribes tu propia View, puedes precachearla.

```
float density = getResources().getDisplayMetrics(). density;
int actualWidth = (int)(bitmap.getWidth() * density);
```

Sensores

Si vas a trabajar con el acelerómetro o sensores para juegos, hay cuatro tasas distintas:

- Normal: 10 mA (Utilizada para detección de orientación)
- Interfaz de usuario (UI): 15 mA (Se comprueba el sensor una vez por segundo)
- Juego (Game): 80 mA
- Muy rápido (Fastest): 90 mA

El mismo coste para el sensor del acelerómetro, magnético y de orientación.

Aplicaciones en background

Vamos a hablar de servicios en general. Si vienes de UNIX o Windows, cuando pensamos en un servicio, pensamos en algo que iniciamos de fondo y continúa ejecutándose, casi como un demonio (daemon) esperando a que llegue alguna petición. Se ejecuta todo el tiempo. Esto funciona muy bien en el mundo del escritorio, pero en el mundo movil, cada uno de estos procesos tienen una sobrecarga asociada. Veamos por ejemplo, algunos dispositivos que primero salieron, por ejemplo, el HTC Dream. Cuando el sistema inicia tenemos una determinada cantidad de RAM en el sistema. Una parte se la queda el Kernel, otra el Framework. Cuando finalmente termina de iniciar, nos queda para las aplicaciones unos 30 o 40 megabytes de RAM. Una cosa que hay que tener en cuenta es que cada proceso en el dispositivo, por muy poco código que tenga, utilizará unos 2 megabytes de RAM sólo para iniciarse, sin tener en cuenta la memoria que pueda necesitar según las tareas que realice.

Si tienes un servicio que se ejecuta de fondo constantemente el sistema terminará matándolo si el usuario está realizando alguna tarea, como por ejemplo, navegando por Internet y el navegador requiera más memoria. Una vez esta memoria quede libre, volverá a iniciar los servicios eliminados para que continúen realizando su trabajo. Y aquí tenemos otra sobrecarga a la hora de tener que detener e iniciar el servicio.

Nos hacemos la siguiente pregunta. Si no podemos tener nuestro proceso ejecutándose siempre de fondo, si no podemos utilizar el modelo de servicio (Windows) / demonio (UNIX), que funcionan bien en escritorio. ¿Cómo debería escribir servicios?

Hay dos métodos primarios que podemos utilizar:

- Si tenemos un punto definido en el futuro en el que necesitemos despertar a nuestro servicio, digamos, actualizar un RSS cada 15 minutos o media hora, podemos especificar una alarma para que el sistema nos despierte y realicemos la tarea en ese punto futuro.
- Otro método es utilizar un receptor de señales, de tal forma que funcionemos en base a eventos. Por ejemplo, si el usuario pasa de una red EDGE a una 3G, cualquier aplicación podrá recibir esta notificación y realizar una tarea pertinente. No hay que olvidarse de deshabilitar la escucha de eventos una vez que ya no estemos interesados en ellos.

Una vez realizada la tarea, asegúrate de llamar a stopSelf para indicarlo, de tal forma que el sistema pueda matar a tu proceso.

En Android 1.5 se añadió setInexactRepeating() para especificar alarmas. Si realmente no nos importa adelantarnos o atrasar la ejecución establecida en un periodo de tiempo, podemos utilizar setInexactRepeating() de tal forma que Android intentará agrupar la ejecución de todas las aplicaciones que pueda en un momento exacto de tiempo. De esta forma, evitamos tener que despertar el dispositivo cada vez que una aplicación tenga que realizar una tarea.

Una buena costumbre sería comprobar el estado de la batería antes de realizar alguna tarea bastante costosa, de tal forma que si queda poca, no haremos nada. A esto se llama programar aplicaciones inteligentes.

Más allá de Android 1.5

Android 1.5 mantiene un historial del consumo de recursos del dispositivo que realizan las aplicaciones: CPU, red, wake-locks... En versiones posteriores, se le dará la opción a los usuarios de conocer qué aplicaciones tiene instaladas que más consumo realizan de batería, de tal forma que sea decisión del usuario continuar utilizándola o desinstalársela. (Es una forma de obligar a los desarrolladores a programar aplicaciones con consumo eficiente de batería)

En la versión 1.5 podemos acceder a una aplicación muy básica oculta, debido a que es simplemente para ver que realmente se están recogiendo esos datos, escribiendo lo siguiente dentro del cuadro para marcar un número de teléfono:

#*#4636#*#

Considera realizar aplicaciones flexibles, de tal forma que si tiene que actualizar información usando Internet, dejar al usuario poder configurar cada cuanto tiempo quiere que se realice esta actualización. De esta forma, le estarás dando al usuario una posibilidad de controlar el gasto de batería que tu aplicación realice.

A modo de resumen, veamos los puntos más importantes:

- Utilizar un analizador eficiente y GZIP para realizar un mejor uso de la red y de CPU.
- Los servicios que duermen y consultan son malos, en vez de esto, utiliza <receiver> y AlarmManager.
  - Desactiva los elementos del manifest cuando no sea necesario seguir utilizándolos.
  - Despiértate junto con las demás aplicaciones (alarmas inexactas).
- Espera a un estado de batería y de red lo suficientemente buenos antes de realizar alguna tarea bastante costosa en batería o red.
- Dar opción a los usuarios sobre el comportamiento de fondo de una aplicación.

Enlaces

Google I/O - Coding for Life -- Battery Life, That Is [video]
Descargar transparencias de la conferencia [pdf]

http://www.desarrolladores-android.com
http://www.android-spa.com

Comentarios

Accede para escribir un comentario.

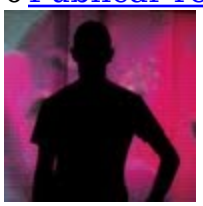
Emilio Murado

Felicidades

Dos de los mejores artículos//tutoriales que he leído sobre Android son tuyos. Realmente elaborados y muy, muy útiles.

Felicidades por ellos.

20/03/2011 21:03
Informe de comentarios abusivos
0 Publicar respuesta a este comentario ▼



Modo 6.6.6

Excelente articulo

Hola!
Muchas gracias por un articulo tan claro y conciso.
Me ha sido de mucha ayuda los ejemplos.

Suerte!

30/08/2010 10:37
Informe de comentarios abusivos
0 Publicar respuesta a este comentario ▼

MEGadeath

PowerManager

Hola como estas, super interesante el articulo, muy dificil de encontrar informacion tan valiosa como esta en español, de verdad te felicito.

La verdad he implementado esto de los wakeLock, pues al finalizar un Activity de reproduccion de videos, la aplicacion se queda como en standBy, pero la verdad no se como corroborar que el PowerManager.waKelock esta realizando algo en el telefono o en su defecto en el simulador.

Gracias por la atencion prestada.

Última modificación: 09/01/2010 16:22
Informe de comentarios abusivos
0 Ver/publicar las respuestas (1) a este comentario ▼

Fabrixio Terán

Portátil

¿Puedo sugerir algo? Lo que haría falta sería al menos que se indicara que se trata de la batería del portátil en la intro. Está bien explicado y presentado, bastantes datos técnicos ;), pero el lector tiene que inferir de que se trata unos 5 minutos antes de entenderlo. Sólo eso, y alguna referencia o bibliografía no vendía mal.

No necesitas publicar mi comentario, con que las sugerencias sean leídas me basta. Saludos, y buen trabajo!

Última modificación: 25/07/2009 20:09
Informe de comentarios abusivos
0 Ver/publicar las respuestas (1) a este comentario ▼