

Introducción al Desarrollo de Aplicaciones Android

Índice de contenido

Introducción.....	5
¿Qué es Android?.....	5
Características.....	5
Arquitectura Android.....	6
Aplicaciones.....	6
Framework.....	6
Librerías.....	7
Android en Tiempo de Ejecución.....	7
Linux Kernel.....	8
Fundamentos Android.....	8
Componentes de la aplicación.....	9
Actividades.....	9
Servicios.....	9
Proveedores de contenido.....	10
Receptor de notificaciones (Broadcast receiver).....	10
El archivo manifest.....	12
Recurso de una aplicación Android.....	15
Actividades.....	16
Crear una Actividad.....	17
Lanzar una Actividad Android.....	19
Ciclo de vida de la actividad.....	21
Tareas y Back Stack.....	30
Guardar el estado de la actividad.....	33
Gestión tareas Android.....	33
Gestión afinidades.....	37
Limpiar el back stack.....	38
Iniciar una tarea.....	39
Fragmentos.....	39
Diseños de Interfaz de Usuario dinámicos.....	40
Crear un fragmento.....	41
Añadir un fragmento a una interfaz de usuario.....	43
Añadir un fragmento a la actividad.....	44
Gestionar fragmentos.....	46
Realizar transacciones en los fragmentos.....	46
Comunicación con una actividad.....	48
Crear eventos callback a la actividad.....	48
Añadir items a la Barra de Acción.....	49
Gestionar el ciclo de vida del fragmento.....	50
Coordinación con el ciclo de vida de la actividad.....	51
Ejemplo de Actividad con dos fragmentos.....	51
Loaders.....	57
Ejecutar y Reiniciar un Loader.....	57
Ejecutar un Loader.....	57
Reiniciar un loader.....	58
Utilizar los métodos callback de LoaderManager.....	59
Ejemplo Loaders.....	61
Resumen del API Loader.....	63
Servicios Android.....	64

Crear un Servicio.....	66
Debería usarse un servicio o un hilo?.....	67
Declarar un servicio en el manifest.....	67
Crear un servicio iniciado.....	68
Enfocado a Android 1.6 o una versión más baja.....	68
Extender la clase IntentService.....	68
Extender la clase Service.....	70
Iniciar un servicio.....	72
Parar un servicio.....	72
Crear un Servicio vinculado (bound service).....	73
Mandar notificaciones al usuario.....	73
Ejecutar un servicio en primer plano.....	73
Gestionar el ciclo de vida de un servicio.....	74
Implementar los callbacks del ciclo de vida.....	75
Qué es un Servicio vinculado (bound service) y cómo se crea.....	77
Básicos.....	77
Crear un servicio vinculado.....	77
Cómo vincular los componentes de una aplicación a un servicio.....	82
Como gestionar el ciclo de vida de un servicio que está iniciado y que permite vinculación.....	84
Qué es un Content Provider.....	85
Básicos sobre Content Provider.....	85
Realizar una consulta al Content Provider.....	87
Realizar la consulta.....	87
Modificar los datos.....	89
Crear un Content Provider.....	92
Resumen del Content URI.....	94
Intents.....	95
Objetos Intent.....	95
Cómo funciona un Intent Filter.....	98
Intent filters (filtros intent).....	98
Procesos e Hilos en Android.....	102
Procesos.....	102
Hilos.....	104
Comunicación Interproceso.....	107
Interfaz de Usuario en Android.....	107
Jerarquía de Views.....	108
Layout.....	108
Widgets.....	109
Eventos de la IU.....	109
Menus.....	110
Temas avanzados.....	110
Declaración del Layout en Android.....	111
Escribir el XML.....	111
Cargar el recurso XML.....	112
Atributos.....	112
Posición del Layout.....	114
Tamaño, padding y márgenes.....	114
Creación de Menús para una Interfaz de Usuario.....	115
Crear un Recurso de Menú.....	115
Inflar un Recurso de Menú.....	116
Crear un Menú de Opciones.....	116
Crear un ContextMenu.....	119

Crear Submenús.....	120
Otras propiedades del Menú.....	121
Barra de Acción- Action Bar.....	125
Añadir la barra de acción.....	125
Añadir ítems de acción.....	126
Añadir un Action View.....	129
Añadir pestañas.....	130
Añadir navegación desplegable.....	131
Estilismo de la barra de acción.....	132
Mostrar un Diálogo en Android.....	134
Qué es un Diálogo.....	134
Mostrar un diálogo.....	134
Cerrar un Diálogo.....	135
Crear un AlertDialog.....	136
Crear un ProgressDialog.....	138
Crear un Dialog personalizado.....	139
Eventos de Interfaz de Usuario.....	141
Listeners de Eventos.....	141
Handlers de eventos.....	143
Modo Táctil.....	143
Manejar el focus.....	144
Notificaciones al Usuario.....	144
Notificaciones emergentes Toast.....	145
Notificación en la barra de estado.....	145
Diálogos de notificación.....	145
Notificaciones Toast.....	146
Básicos.....	146
Colocar el Toast.....	146
Crear un Toast View personalizado.....	147
Notificaciones en Barra de Estado.....	148
Básicos.....	149
Gestionar las notificaciones.....	149
Crear una notificación.....	150
Crear un View extendido personalizado.....	153

Introducción

¿Qué es Android?

En estos capítulos vamos a hacer una introducción práctica para desarrollar aplicaciones Android. Exploraremos los conceptos detrás de Android y el framework para construir una aplicación.

El primer paso en la programación Android es la descarga de SDK (kit de desarrollo de software). Para instrucciones e información, visite [Instalar SDK y ADT plugin](#).

Después de conseguir el SDK, comience por leer esta documentación. El documento "Fundamentos de la aplicación" es un buen sitio para empezar para aprender los temas básicos sobre el framework de la aplicación.

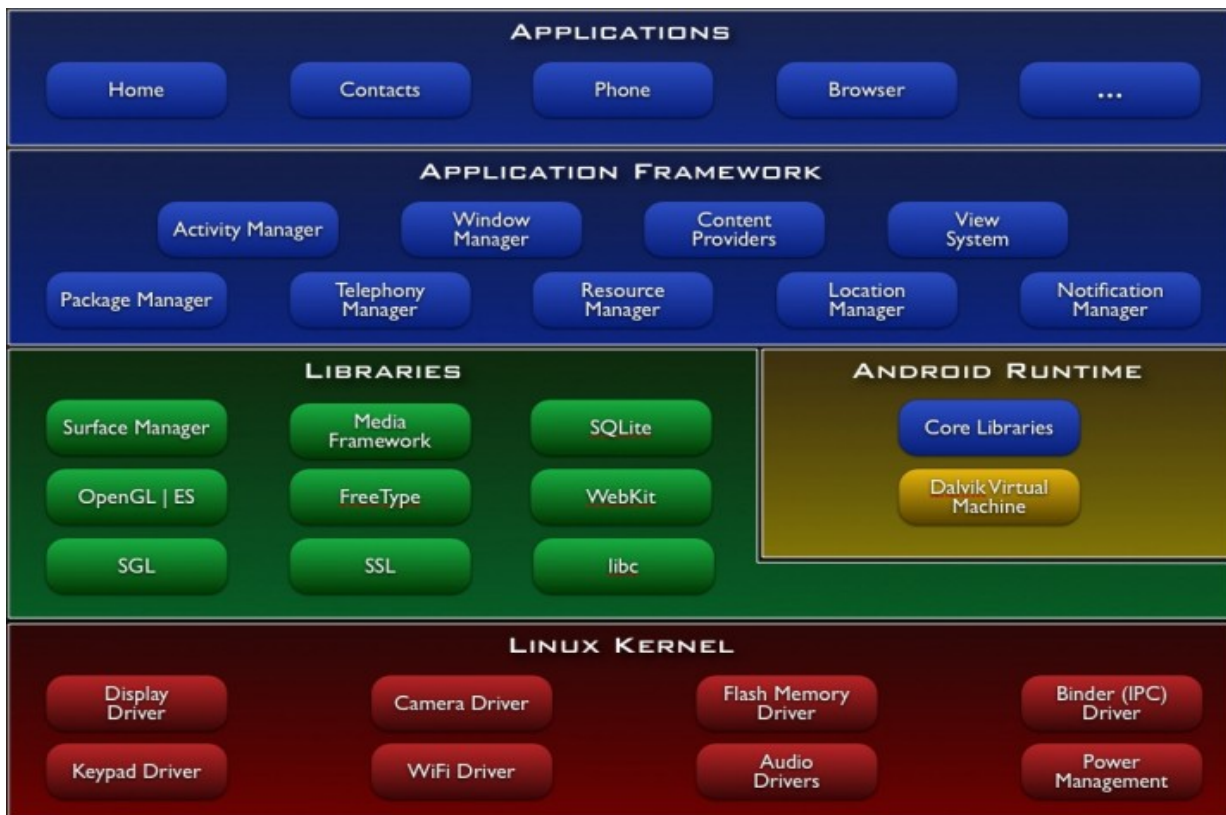
Android es un conjunto de software para dispositivos móviles que incluye un sistema operativo, middleware y aplicaciones clave. El SDK de Android proporciona las herramientas y APIs necesarios para comenzar a desarrollar aplicaciones en la plataforma Android usando el lenguaje de programación Java.

Características

- Marco de la aplicación que permite la reutilización y el reemplazamiento de los componentes
- Máquina virtual Dalvik optimizada para dispositivos móviles
- Navegador integrado basado en el motor WebKit
- Gráficos optimizados impulsados por una biblioteca 2D hecha a medida; gráficos 3D basados en las especificaciones OpenGL ES 1.0 (hardware con aceleración opcional)
- SQLite para almacenamiento estructurado de datos
- Apoyo a los medios de audio comunes, videos y formatos de imagen (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- GSM Telefonía (dependiente de hardware)
- Bluetooth, EDGE, 3G, y WiFi (dependiente de hardware)
- Cámara, GPS, brújula y acelerómetro (dependiente de hardware)
- Entorno de desarrollo completo que incluye un emulador de dispositivos, herramientas para la depuración, la memoria y de perfiles de rendimiento, y un plugin para el IDE de Eclipse

Arquitectura Android

El siguiente diagrama muestra los componentes principales del sistema operativo Android. Cada sección se describe con más detalle a continuación.



Aplicaciones

Android incluye un conjunto de aplicaciones básicas, como un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos, y otros. Todas las aplicaciones están escritas con el lenguaje de programación Java.

Framework

Al proporcionar una plataforma de desarrollo abierto, Android ofrece a los desarrolladores la capacidad de crear aplicaciones muy ricas e innovadoras. Los desarrolladores pueden tomar ventaja de los dispositivos de hardware, información de acceso a la localización, ejecutar servicios en segundo plano, poner alarmas, añadir las notificaciones de la barra de estado, y mucho, mucho más.

Los desarrolladores tienen pleno acceso a la API de un mismo marco utilizado por las aplicaciones básicas. La arquitectura de la aplicación está diseñada para simplificar la reutilización de componentes, cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación podrá hacer uso de esas capacidades (sujeto a restricciones de seguridad impuestas por el framework). Este mismo mecanismo permite que el usuario sustituya los componentes.

Detrás de todas las aplicaciones existe un conjunto de servicios y sistemas, incluyendo:

- Un amplio y extensible conjunto de Vistas que se puede utilizar para construir una aplicación, incluyendo las listas, redes, cajas de texto, botones, e incluso un navegador web embebido
- Proveedores de contenido que permiten a las aplicaciones acceder a datos de otras aplicaciones (por ejemplo, contactos), o compartir sus propios datos
- Un administrador de recursos, que facilita el acceso a recursos no-código como cadenas localizadas, gráficos y archivos de diseño
- Un gestor de notificaciones que permite a todas las aplicaciones mostrar alertas personalizadas en la barra de estado
- Un gestor de actividad que maneja el ciclo de vida de las aplicaciones y proporciona una navegación común backstack

Librerías

Android incluye un conjunto de librerías C / C + + utilizadas por los diversos componentes del sistema Android. Estas capacidades están expuestas a los desarrolladores a través del marco de aplicación para Android. Algunas de las librerías del núcleo son los siguientes:

- Sistema de biblioteca C una implementación derivada de la BSD de la biblioteca del sistema estándar de C (libc), en sintonía para los dispositivos Linux
- Librerías para los medios - basadas en PacketVideo's OpenCORE; las librerías soportan visiones y grabaciones de muchos de los formatos más populares de audio y video, así como archivos de imágenes estáticas, incluyendo MPEG4, H.264, MP3, AAC, AMR, JPG, y PNG
- Gestor de superficie - gestiona el acceso al subsistema de visualización y a las capas graficas 2D y 3D desde múltiples aplicaciones
- LibWebCore - un motor de un navegador web moderno que potencia el navegador Android y una vista web insertable
- SGL - el motor de base de gráficos 2D
- Librerías 3D - una implementación basada en OpenGL ES 1.0 API, las librerías utilizan o bien aceleración 3D por hardware (donde esté disponible) o el software 3D incluido; un rasterizador altamente optimizado
- FreeType - mapa de bits y representación de la fuente de vectores
- SQLite - un motor de base de datos relacional potente y ligera disponible para todas las aplicaciones

Android en Tiempo de Ejecución

Android incluye un conjunto de librerías principales que proporciona la mayor parte de la funcionalidad disponible en las librerías principales del lenguaje de programación Java.

Cada aplicación Android corre en su propio proceso, con su propia instancia de la máquina virtual de Dalvik. Dalvik ha sido desarrollada para que un dispositivo puede ejecutar varias máquinas virtuales de manera eficiente. La

máquina virtual Dalvik ejecuta archivos ejecutables en el formato Dalvik (.Dex) que está optimizado para dejar la mínima huella en la memoria. La MV se basa en registros, y corre clases compiladas con un compilador del lenguaje Java que las ha convertido en un formato. Dex con la herramienta ya incluida "dx"

La máquina virtual Dalvik se basa en Linux para la funcionalidad subyacente, como el threading y la gestión de memoria de bajo nivel.

Linux Kernel

Android se apoya en la versión 2.6 de Linux para los servicios del núcleo básicos, tales como la seguridad, la gestión de memoria, gestión de procesos, conjunto de red, y el modelo controlador. El kernel también actúa como una capa de abstracción entre el hardware y el resto del conjunto de software.

Fundamentos Android

Las aplicaciones Android están escritas en el lenguaje de programación Java. Las herramientas SDK de Android compilan el código—así como los datos y los archivos de recursos—dentro de un paquete Android, un archivo con un sufijo .apk. Todo el código dentro de un archivo .apk es considerado como una aplicación y es el archivo utilizado por los dispositivos Android utilizados para instalar la aplicación.

Una vez que se ha instalado en cada dispositivo, cada aplicación Android tiene su propia sandbox de seguridad:

- El sistema operativo de Android es un sistema Linux multi-usuario en el que cada aplicación es un usuario diferente.
- Por defecto, el sistema asigna a cada aplicación un ID a cada usuario Linux ID (el ID es utilizado solamente por el sistema y es desconocido para la aplicación). El sistema adjudica permisos para todos los archivos de una aplicación para que sólo el usuario ID asignado a la aplicación tenga acceso.
- Cada proceso tiene su propia máquina virtual (VM), por lo que el código de una aplicación se ejecuta aislada de otras aplicaciones.
- Por defecto, todas las aplicaciones se ejecutan en su propio proceso Linux.

Android comienza el proceso cuando cualquier de los componentes de la aplicación necesita ser ejecutados y lo termina cuando ya no se necesita o cuando el sistema necesita recuperar memoria para otras aplicaciones.

De esta manera, el sistema Android implementa el principio de menor privilegio. Esto es, que cada aplicación, por defecto, tiene acceso solamente a los componentes que necesita para hacer su trabajo. Esto crea un entorno muy seguro en el que una aplicación no puede tener acceso a partes del sistema para los que no tiene permiso.

Sin embargo, existen maneras para que una aplicación comparta datos con otras aplicaciones y para que una aplicación tenga acceso a servicios del sistema:

- Es posible hacer que dos aplicaciones compartan el mismo ID de usuario Linux, por lo que en este caso pueden acceder a los archivos de la otra aplicación. Para conservar recursos del sistema, las aplicaciones con el mismo ID de usuario también pueden llegar a ejecutarse en el mismo proceso Linux y

compartir la misma VM (las aplicaciones también deben ser firmadas con el mismo certificado).

- Una aplicación puede pedir permiso para acceder a los datos del dispositivo tales como los contactos del usuario, mensajes SMS, la memoria montable(tarjeta SD), cámara, Bluetooth y más. Todos los permisos de la aplicación deben ser concedidos por el usuario en el momento de la instalación.

Esto cubre la información básica con respecto a como una aplicación Android existe dentro de un sistema. El resto del documento da información sobre:

- Los componentes básicos del marco que definen la aplicación.
- El archivo manifest en el que se declaran los componentes y las funciones requeridas de los dispositivos para la aplicación.
- Recursos que están separados del código de la aplicación y permiten a la aplicación optimizar su comportamiento frente a una variedad de configuraciones de dispositivos.

Componentes de la aplicación

Los componentes de la aplicación son los bloques de construcción esenciales de una aplicación Android. Cada componente es un punto diferente por el cual el sistema puede entrar a la aplicación. No todos los componentes son puntos de entrada para el usuario y algunos dependen unos de otros pero cada uno existe como entidad propia y juega un determinado rol—cada uno es un bloque de construcción único que ayuda a definir el comportamiento global de la aplicación.

Existen cuatro tipos diferentes de componentes. Cada tipo tiene un propósito diferente y tiene un ciclo de vida diferente que define como el componente es creado y destruido.

A continuación se describen los cuatro tipos de componentes de la aplicación:

Actividades

Una actividad representa una pantalla con una interfaz de usuario. Por ejemplo, una aplicación de mail puede tener una actividad que muestra una lista de nuevos correos electrónicos, otra actividad para componer un correo electrónico, otra actividad para leer correos electrónicos. A pesar de que las actividades trabajan juntos para formar una experiencia de usuario cohesiva en la aplicación de correos electrónicos, cada uno de ellos es independiente con respecto al otro. Como tal, una aplicación diferente puede empezar cualquiera de estas actividades (si la aplicación de correo electrónico lo permite). Por ejemplo, una aplicación para una cámara puede empezar la actividad que redacta mails en la aplicación de correos electrónicos, para que el usuario pueda compartir una foto.

Una actividad es implementada como una subclase de Activity y se puede aprender más sobre ello en Activity.

Servicios

Un servicio es un componente que se ejecuta en un segundo plano para llevar

a cabo operaciones de largo recorrido o para llevar a cabo procesos remotos. Un servicio no suministra una interfaz de usuario. Por ejemplo, un servicio puede hacer que suene música a la vez que el usuario está en una diferente aplicación, o puede recopilar datos a través de la red sin bloquear la interacción de un usuario con una actividad. Otro componente, como un actividad, puede empezar un servicio y dejarlo ejecutándose o unirlo a otro para interactuar con él.

Un servicio es implementado como una subclase de `Servicio` y se puede aprender más sobre ello en `Servicios`.

Proveedores de contenido

Un proveedor de contenido gestiona un conjunto de datos compartidos de la aplicación. Se puede almacenar los datos en el sistema de archivos, en una base de datos SQLite, en la web, o en cualquier otro lugar persistente de almacenaje a la cual la aplicación tenga acceso. A través del proveedor de contenido, otras aplicaciones pueden preguntar o incluso modificar los datos (si el proveedor de contenido lo permite). Por ejemplo, el sistema Android tiene un proveedor de contenido que gestiona la información de contacto de usuario. De esta manera, cualquier aplicación con los permisos apropiados puede preguntar al proveedor de contenido información sobre una determinada persona.

Los proveedores de contenido también son útiles para leer y escribir datos que son privados para la aplicación y no se comparten. Por ejemplo, la aplicación `Note Pad` utiliza un proveedor de contenido para guardar notas.

Un proveedor de contenido es implementado como una subclase de `ContentProvider` y debe implementar un conjunto estándar de APIs que permite a otras aplicaciones realizar las transacciones. Para más información, ver `ContentProvider`.

Receptor de notificaciones (Broadcast receiver)

Un receptor de notificaciones es un componente que responde al sistema global de mensajes de notificación. Muchos mensajes originados por el sistema —por ejemplo, un mensaje informando que la pantalla se ha apagado, que la batería está baja, o que la imagen se ha capturado. Las aplicaciones también pueden iniciar esos mensajes— por ejemplo, para hacer saber a otras aplicaciones que parte de los datos han sido descargados al dispositivo y se pueden utilizar. A pesar de que los `BroadcastReceiver` no muestran una interfaz de usuario, pueden crear una notificación de barra de estado para avisar al usuario cuando ocurre un evento. Comunmente lo que ocurre es que un `BroadcastReceiver` es sólo una "puerta" a otros componentes y se pretende que realice muy poco trabajo. Por ejemplo, puede iniciar un servicio para realizar algún trabajo relacionado con el evento.

Un `BroadcastReceiver` es implementado como una subclase de `BroadcastReceiver` y cada notificación es entregada como un objeto `Intent`.

Un aspecto único del diseño del sistema Android es que cualquier aplicación puede ejecutar otro componente de otra aplicación. Por ejemplo, si quieres que el usuario capture una foto con el dispositivo de la cámara, probablemente

haya otra aplicación que lo haga y su aplicación lo puede utilizar, en vez de que usted mismo desarrolle una actividad para capturar fotos. No necesitas incorporar el código ni crear un link a él. En vez de eso, puedes simplemente ejecutar la actividad que captura las fotos en la aplicación de la cámara. Cuando finaliza, la foto se devuelve a su aplicación para que la puedas utilizar. Para el usuario, es como si la cámara formara parte de la aplicación.

Cuando el sistema ejecuta un componente, comienza el proceso para esa aplicación, (si no está funcionando ya) e instancia las clases que necesita el componente. Por ejemplo, si su aplicación ejecuta la aplicación en la aplicación de la cámara que captura la foto, la actividad se ejecuta en el proceso que pertenece a la aplicación de la cámara, no en el proceso de su aplicación. Por lo que a diferencia de otras aplicaciones en otros muchos sistemas, las aplicaciones Android no tienen un sólo punto de entrada (no existe una función `main()`, por ejemplo).

Dado que el sistema ejecuta cada aplicación en un proceso separado con permisos de archivo que restringen el acceso a otras aplicaciones, su aplicación no puede activar directamente un componente desde otra aplicación. El sistema Android, sin embargo, si que puede. Así, para activar el componente en otra aplicación, se debe mandar un mensaje al sistema especificando su intención de ejecutar un componente en particular. El sistema en ese momento activa el componente.

Activando Componentes

Tres de los cuatro tipos de componentes—actividades, servicios, y broadcast receivers—están activados por un mensaje asíncrono llamado intent. El Intent une componentes individuales entre sí en tiempo de ejecución (se puede pensar que son como mensajeros que piden una acción de otros componentes), tanto como si el componente pertenece a su aplicación o a otra.

Un Intent es creado con un objeto Intent, que define un mensaje para activar un componente específico o un tipo específico de componente—un intent puede ser explícito o implícito, respectivamente.

Para actividades y servicios, un Intent define la acción a ejecutar (por ejemplo, para "visualizar" o "mandar" algo) y puede especificar el URI de los datos sobre los que actuar (entre otras cosas que puede necesitar el componente ejecutado). Por ejemplo, un Intent puede transmitir una petición para que una actividad muestre una imagen o que abra una página. En algunos casos, se puede empezar una actividad para recibir un resultado, en cuyo caso, la actividad también devuelve un resultado en un Intent (por ejemplo, se puede emitir un Intent para permitir al usuario elegir un contacto personal y que te lo devuelva —el Intent devuelto incluye un URI apuntando al contacto escogido).

Para los broadcast receivers, el Intent simplemente define el mensaje mostrado (por ejemplo, un mensaje indicando que la batería está baja incluye sólo una cadena de acción conocida que indica "batería baja").

El otro tipo de componente, el proveedor de contenido, no se activa mediante Intent. Se activa cuando es diana de una petición ContentResolver. El "content

resolver" gestiona todas las transacciones directas con el proveedor de contenido para que el componente que está realizando las transacciones con el proveedor no necesite hacerlo y en vez de ello haga llamadas a los métodos del objeto ContentResolver. Esto deja una capa de abstracción entre el proveedor de contenido y el componente que pide la información (por seguridad).

Existen métodos diferentes para activar cada tipo de componente:

- Puede ejecutar una actividad (o darle algo nuevo para hacer) pasando un Intent al método startActivity() o al startActivityForResult() (cuando quieres que la actividad te devuelva un resultado).
- Puede ejecutar un servicio (o darle nuevas instrucciones a uno que ya se está ejecutando) pasando un Intent al método startService(). O lo puedes unir al servicio pasando un Intent al método bindService().
- Puede ejecutar un broadcast pasando un Intent a métodos como sendBroadcast(), sendOrderedBroadcast(), o sendStickyBroadcast().
- Puede ejecutar una consulta a un proveedor de contenido llamando a query() en un ContentResolver.

Para más información sobre como usar Intents, vea el documento Intents y Filtros de Intents . Más información sobre la activación de componentes específicos la puedes encontrar también en los siguientes documentos: Actividades, Servicios, BroadcastReceiver y Content Providers.

El archivo manifest

Antes de que el sistema Android pueda ejecutar un componente de la aplicación, el sistema debe saber que el componente existe leyendo el archivo AndroidManifest.xml de la aplicación (el archivo "manifest"). Su aplicación debe declarar todos sus componentes en este archivo, que debe estar en la raíz del directorio del proyecto de la aplicación.

El manifest realiza una serie de funciones además de la declaración de los componentes de la aplicación, tal y como:

- Identificación cualquier permiso de usuario que requiera la aplicación, como acceso a Internet o acceso de lectura de los contactos del usuario.
- Declaración del mínimo nivel del API requerida por la aplicación, basado en los APIs utilizados por la aplicación.
- Declaración de las características del hardware y software utilizados o necesitados por la aplicación, como la cámara, servicios de bluetooth, o una pantalla multitáctil.
- Las librerías API con las que la aplicación necesita ser linkeada (aparte de los APIs marco de Android), tal y como librería de Google Maps.

Componentes a declarar

La tarea principal del manifest es de informar al sistema sobre los componentes de la aplicación. Por ejemplo, un archivo manifest puede declarar una actividad como se describe a continuación:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:icon="@drawable/app_icon.png" ... >
        <activity android:name="com.example.project.ExampleActivity"
            android:label="@string/example_label" ... >
        </activity>
    </application>
</manifest>
```

En el elemento `<application>`, el atributo `android:icon` apunta a recursos para un icono que identifica la aplicación.

En el elemento `<activity>`, el atributo `android:name` especifica el nombre de la clase de la subclase `Activity` y el atributo `android:label` especifica una cadena de caracteres para utilizar como la etiqueta visible al usuario para la actividad.

Se debe declarar todos los componentes de la aplicación de la siguiente manera:

- `<activity>` elementos para actividades
- `<service>` elementos para servicios
- `<receiver>` elementos para los broadcast receivers
- `<provider>` elementos para los proveedores de contenido

Las actividades, servicios y proveedores de contenido que se incluyan en el código fuente pero no se declaran en el archivo manifest no son visibles al sistema y consecuentemente, no serán nunca ejecutados. Sin embargo los broadcast receivers pueden ser o bien declarados en el manifest o creados dinámicamente en el código (como objetos `BroadcastReceiver`) y registrados por el sistema al llamar al método `registerReceiver()`.

Declaración de la capacidades de los componentes

Tal y como se ha discutido anteriormente, en Activando los componentes, se puede utilizar un `Intent` para ejecutar actividades, servicios y broadcast receivers. Se puede hacer explícitamente llamando al componente objetivo (utilizando el nombre de la clase componente) en el intent. Sin embargo, el poder real de los intents se encuentra en el concepto de las acciones intent. Con las acciones intent puedes describir simplemente el tipo de acción que se quiere realizar (y opcionalmente, los datos sobre los que se quiere realizar la acción) y permitir al sistema encontrar un componente en el dispositivo que pueda realizar la acción y ejecutarla. Si existen múltiples componentes que pueden realizar la acción descrita por el intent, entonces el usuario selecciona cual quiere utilizar.

La manera en el que el sistema identifica los componentes que pueden responder al intent es comparando el intent recibido con los intent filters provistos en el archivo manifest de las otras aplicaciones del dispositivo.

Cuando se declara un componente en el manifest de la aplicación, se puede opcionalmente incluir los intent filters que declaran las capacidades del componente para que pueda responder a los intents de otras aplicaciones. Se puede declarar un intent filter para un componente añadiendo un elemento `<intent-filter>` como hijo del elemento de la declaración del

componente.

Por ejemplo, una aplicación de correo electrónico con una actividad para componer un nuevo correo electrónico puede declarar un intent filter en su manifest para responder a "mandar" intents (para mandar el correo electrónico). Una actividad en su aplicación puede crear un intent con la acción mandar (ACCION_MANDAR), la cual el sistema hace corresponder con la actividad mandar de la aplicación de correo electrónico y lo lanza cuando se llama al intent con el método startActivity().

Para saber más sobre crear intent filters, ver el documento de Intents and Intent Filters.

Declaración de requerimientos de la aplicación

Existen una variedad de dispositivos que funcionan con Android y no todos ellos tienen las mismas características y capacidades. Para prevenir que la aplicación sea instalada en dispositivos que carezcan de características necesarias para su aplicación, es importante que se defina un perfil claro en el archivo manifest definiendo los requerimientos de los dispositivos y del software que soporta su aplicación. La mayoría de estas declaraciones son sólo de carácter informativo y el sistema no lo lee, pero servicios externos Android Market los leen para proveer al usuario de filtrado cuando están buscando aplicaciones para sus dispositivos.

Por ejemplo, si su aplicación necesita una cámara y utiliza APIs introducidos en Android 2.1 (nivel API 7), se debería declarar como requerimientos en el archivo manifest. De esta manera, dispositivos que no tienen una cámara y tienen una versión Android anterior a 2.1 no podrán instalar su aplicación desde el Android Market.

Sin embargo, también puede declarar que su aplicación utiliza una cámara, pero no lo necesita. En ese caso, su aplicación debe comprobar en tiempo de ejecución si el dispositivo tiene cámara y debe desactivar cualquier característica que utilice la cámara que no esté disponible.

A continuación se describen las características más importantes de los dispositivos que debería considerar en el momento del diseño y desarrollo de la aplicación:

Tamaño de la pantalla y densidad

Para poder ordenar los dispositivos de acuerdo con su tipo de pantalla, Android define dos características para cada dispositivo: tamaño de la pantalla (las dimensiones físicas de la pantalla) y la densidad de la pantalla (la densidad de los píxeles en la pantalla, o puntos por pulgada). Para simplificar todos los diferentes tipos de configuraciones de pantalla, el sistema Android los agrupa en grupos seleccionados que lo hacen más fácil de encontrar.

Los tamaños de las pantallas son: pequeño, normal, grande, y extra grande. La densidad de la pantalla puede ser: baja densidad, densidad media, densidad alta, y densidad extra alta.

Por defecto, su aplicación es compatible con todos los tamaños de pantalla y densidades ya que el sistema Android hace los ajustes apropiados para el

diseño de la interfaz de usuario y para los recursos de imagen. Sin embargo, se debería crear diseños especializados para determinados tamaños de pantalla y proveer imágenes especializadas para densidades especiales, utilizando recursos de diseño alternativos, y declarando en el archivo manifest cuales son los tamaños de pantalla que su aplicación soporta con el elemento `<supports-screens>`.

Configuración para entrada de datos

Muchos dispositivos tienen un mecanismo diferente para que el usuario de entrada a datos; teclado, un trackball, o un control de navegación de 5 direcciones. Si su aplicación necesita un tipo específico de hardware para la entrada de datos, debe especificarlo en el archivo manifest con el elemento `<uses-configuration>`. Sin embargo, es raro que una aplicación necesite una configuración determinada para entrada de datos.

Características de los dispositivos

Existen muchas características de hardware y software que pueden o no existir en un dispositivo con Android, tal y como la cámara, un sensor ligero, bluetooth, una determinada version de OpenGL, o la fidelidad de la pantalla táctil. No debería asumir nunca que una característica determinada está disponible en todos los dispositivos Android (más que la disponibilidad de la librería estándar de Android), por lo que se debe declarar todas las características utilizadas por su aplicación con el elemento `<uses-feature>`.

Versión de la Plataforma

Los diferentes dispositivos Android suelen ejecutarse en diferentes versiones de la plataforma Android, como Android 1.6 o Android 2.3. Cada versión sucesiva normalmente APIs adicionales que no están disponibles en las anteriores versiones. Para indicar cual es el conjunto de APIs disponibles, cada versión de la plataforma especifica una nivel API (por ejemplo, Android 1.0 es API nivel 1 y Android 2.3 es API nivel 9). Si se utiliza cualquier APIs añadida a la plataforma después de la version 1.0, se debe declarar el mínimo nivel API Level en el que esos APIs se introdujeron utilizando el elemento `<uses-sdk>`.

Es importante que se declaren todos esos requerimientos en la aplicación, ya que, cuando se distribuye la aplicación en el Android Market, el Market utiliza estas declaraciones para filtrar cuales son las aplicaciones disponibles en cada dispositivo. Debido a esto, su aplicación debería estar disponible para dispositivos que cumplen con todas los requerimientos de la aplicación.

Recurso de una aplicación Android

Una aplicación Android está compuesta por algo más que el código—necesita recursos aparte del código fuente, como imágenes, archivos de audio, y cualquier recurso relacionado con la presentación visual de la aplicación. Por ejemplo, se deberían definir las animaciones, los menus, estilos, colores, y el diseño de las interfaces de usuario con los archivos XML. Utilizar los recursos de aplicación hace más fácil actualizar diferentes características de la aplicación sin modificar el código —mediante el conjunto de recursos alternativos— permite optimizar la aplicación para una variedad de configuraciones de dispositivo (como diferentes idiomas tamaño de pantallas).

Para cada recurso que se incluya en el proyecto Android, el kit de desarrollo SDK define un ID integer único, que se puede usar para referenciar el recurso del código de la aplicación o de cualquier otro recurso definido en el XML. Por ejemplo, si su aplicación contiene un archivo de imagen llamado `logo.png` (guardado en el directorio `res/drawable/`), el SDK genera un recurso ID llamado `R.drawable.logo`, que se puede utilizar para referenciar la imagen e insertar en él la interfaz de usuario.

Uno de los aspectos más importantes de tener los recursos separados del código, es poder tener la capacidad de proveer recursos alternativos para las diferentes configuraciones de los dispositivos. Por ejemplo, definiendo las cadenas de caracteres de la interfaz de usuario en el XML, así se pueden traducir a otros idiomas y guardar estas cadenas de caracteres en archivos separados. Así, basado en el clasificador de lenguaje adjunto al nombre del directorio del recurso (como por ejemplo `res/values-fr/` para caracteres franceses) y el lenguaje del usuario elegido, el sistema Android aplica la cadena de caracteres del lenguaje apropiado a tu interfaz de usuario.

Android soporta diferentes clasificadores para los recursos alternativos. El clasificador es un cadena corta de caracteres que se incluye en el nombre de los directorios de recursos para definir las configuraciones de los dispositivos para los cuales esos recursos se deberían utilizar. Como otro ejemplo, se deberían crear diferentes diseños para las actividades de la aplicación, dependiendo de la orientación de la pantalla y el tamaño. Por ejemplo, cuando la pantalla del dispositivo está en orientación vertical, se puede querer un diseño con los botones en vertical, pero cuando la pantalla está en orientación horizontal, los botones deben estar alineados horizontalmente. Para cambiar el diseño dependiendo de la orientación, se pueden definir dos diferentes diseños y aplicar el clasificador correspondiente a cada nombre del diseño en el directorio. De este modo el sistema automáticamente aplica el diseño apropiado dependiendo de la orientación actual del dispositivo actual.

Actividades

Una Actividad es un componente de la aplicación que nos ofrece una pantalla con la que los usuarios pueden interactuar para realizar una acción, como marcar un número, hacer una foto, mandar un correo electrónico, o ver un mapa. Cada actividad tiene una ventana en la que dibujar la interfaz de usuario. La ventana suele llenar la pantalla, pero puede ser más pequeña y flotar encima de las ventanas.

Una aplicación suele tener múltiples actividades que están vinculadas de manera poco consistente unas a otras. Normalmente, una actividad en una aplicación se especifica como la actividad "principal", la cual se muestra al usuario al lanzar por primera vez la aplicación. Cada actividad entonces puede empezar otra actividad para realizar diferentes acciones. Cada vez que comienza una nueva actividad, la actividad anterior se para, pero el sistema preserva la actividad en una pila (the "back stack"). Cuando se ejecuta una nueva actividad, se empuja al back stack y le da el focus al usuario. El back stack permite el mecanismo básico "ultimo dentro, primero fuera" por lo que, cuando el usuario ha terminado con la actividad actual y hace click en la tecla

BACK, se saca de la pila(y se destruye) y la actividad previa se reanuda. (El back stack es estudiado en más profundidad en el documento Tareas y Back Stack)

Cuando una actividad se para porque otra actividad empiece, se notifica del cambio en estado a través de los métodos de retorno del ciclo de vida de la actividades. Las actividades pueden recibir diferentes métodos de retorno, dependiendo del cambio en su estado—si el sistema lo está creando, parando, reanudando, o destruyéndolo—y cada retorno nos da la oportunidad de realizar trabajo específico apropiado para cada cambio de estado. Por ejemplo, cuando se para, su actividad debería soltar cualquier objeto grande, como las conexiones con la red o con la base de datos. Cuando la actividad se reanuda, se readquieren los recursos necesarios y se reanudan las actividades necesarias que fueron interrumpidas. Estos estados de transición son parte del ciclo de vida de la actividad.

El resto de este documento trata sobre los temas básicos sobre como desarrollar y utilizar una actividad, incluyendo una discusión completa sobre como funciona el ciclo de vida de la actividad, para que pueda manejar la transición entre los diferentes estados de la actividad.

Crear una Actividad

Para crear una actividad, debe crear la subclase `Actividad` (o una subclase existente). En su subclase, necesita implementar métodos de retorno a los que su sistema va a llamar cuando la actividad se mueva entre diferentes estados de su ciclo de vida, como creación de la actividad, paro, reanudación o destrucción. Los dos métodos de retorno más importantes son:

`onCreate()`

Debe implementar este método. El sistema lo llama cuando está creando su actividad. Dentro de su implementación, debe inicializar los componentes esenciales de su actividad. Aquí es donde debe llamar al método `setContentView()` para definir el layout de la interfaz de usuario de la actividad.

`onPause()`

El sistema llama a este método como primera indicación de que el usuario está abandonando su actividad (aunque no siempre significa que su actividad está siendo destruida). Aquí es donde debería comitear cualquier cambio que desea que persista más allá de la actual sesión de usuario (ya que el usuario puede que no retorne).

Existen otros diferentes métodos de retorno del ciclo de vida que debe utilizar para poder suministrar un proceso fluido entre actividades y para manejar interrupciones no esperadas que causen que su actividad se pare o incluso se destruya. Todos estos otros métodos se discuten posteriormente, en la sección sobre manejo del ciclo de vida de la actividad.

Implementación de una interfaz de usuario

La interfaz de usuario para una actividad se suministra mediante una jerarquía de objetos vista— derivada de la clase `View`. Cada vista controla un espacio rectangular particular dentro de la ventana de la actividad y puede responder a

la interacción del usuario. Por ejemplo, una vista puede ser un botón que inicia una acción cuando el usuario lo toca.

Android suministra una cantidad de vistas preparadas-y hechas que puede utilizar para diseñar y organizar el layout. Los "Widgets" son vistas que proveen unos elementos visuales(e interactivos) para la pantalla, como botones, campos de texto, checkbox, o simplemente imágenes. Los "Layouts" son vistas derivadas del ViewGroup que provee un model único layout model para cada uno de sus hijos vistas, como los "linear layout", un "grid layout", o un layout relativo. También puede hacer una subclase de las clases View y ViewGroup (o de las existentes subclases) para crear tus propios widgets y layouts y utilizarlos en tu propio layout.

La manera más común de definir un layout utilizando vistas es con un archivo XML en los recursos de su aplicación. De esta manera, puede mantener el diseño de su interfaz de usuario separado del código fuente que define el comportamiento de la actividad. Puede definir el layout de su IU para su actividad con el método setContentView(), pasando el recurso ID para el layout. Sin embargo, también puede crear un nuevo View en el código de su actividad y construir una jerarquía de vistas insertando un nuevo View un ViewGroup, y después usar ese layout pasando la raízViewGroup al método setContentView().

Para información sobre como crear una interfaz de usuario, ver la documentación en Interfaz de Usuario

Declaración de la actividad en el manifest.

Debe declarar su actividad en el archivo manifest para que se pueda acceder a ella desde el sistema. Para declarar su actividad, abra su archivo manifest e incluya un elemento <actividad> como un hijo del elemento <aplicación>. Por ejemplo:

```
<manifest ... >
    <application ... >
        <activity android:name=".ExampleActivity" />
        ...
    </application ... >
    ...
</manifest >
```

Existen otros diferentes atributos que puede incluir en este elemento, para definir propiedades como la etiqueta de la actividad, un icono para la actividad o una temática para darle estilo a la actividad de la IU. Ver la referencia al elemento <actividad> para más información sobre los diferentes atributos.

Utilización de filtros intent

Un elemento <actividad> también puede especificar varios filtros intent—utilizando el elemento— <filtro-intent>;para declarar como pueden otros componentes de la aplicación activarla.

Cuando se crea una nueva aplicación utilizando las herramientas de Android SDK, la actividad creada para usted automáticamente incluye un filtro intent que declara la respuesta de la actividad a la acción "main" y debería ser

colocada en la categoría de "launcher". El filtro-intent se ve así;

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

El elemento <acción> especifica que este es el punto de entrada "principal" a la aplicación. El elemento <categoría> especifica que la actividad debería ser listada en el launcher del sistema de la aplicación (para permitir a los usuarios lanzar esta actividad).

Si quiere que su aplicación sea autónoma y que otras aplicaciones no puedan activar sus actividades, entonces no necesita ningún otro filtro-intent. Sólo una actividad debería tener la acción "main" y la categoría de "launcher" como en el ejemplo anterior. Las actividades que no quiera que sean disponibles para otras aplicaciones no deberían tener ningún filtro-intent y los puede lanzar usted mismo utilizando intents explícitos (como se describe en la sección siguiente).

Sin embargo, si quiere que su actividad responda a intents implícitos que son entregados desde otras aplicaciones (y la suya propia), entonces debe definir filtros-intent adicionales para su actividad. Para cada tipo de intent al que quiera responder, debe incluir un <filtro-intent> que incluye un elemento <acción> y, opcionalmente, un elemento <categoría> y/o un elemento <data>. Estos elementos especifican el tipo de intent al que su actividad puede responder.

Lanzar una Actividad Android

Puede lanzar otra actividad llamando al método `startActivity()`, pasándole un Intent que describe la actividad que quiere lanzar. El intent especifica bien la actividad exacta que quiere lanzar o describe el tipo de acción que quiere realizar (y el sistema selecciona la actividad apropiada para usted, que puede ser incluso de una aplicación diferente). Un intent también puede llevar pequeñas cantidades de datos para ser utilizados por la actividad lanzada.

Cuando trabaja dentro de su propia aplicación, a menudo necesitará simplemente lanzar una actividad conocida. Puede hacerlo creando un intent que explícitamente defina la actividad que quiere lanzar, utilizando el nombre de la clase. Por ejemplo, así es como una actividad hace lanzar otra actividad llamada `SignInActivity`:

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

Sin embargo, la aplicación puede realizar una acción, como mandar un correo electrónico, un mensaje de texto, o actualizar el estado, utilizando datos de la actividad. En este caso, la aplicación puede no tener actividades que realicen esas acciones, por lo que puede utilizar otras aplicaciones de su dispositivo que si que realicen esas acciones. Aquí es donde los intents tienen su valor —puede crear un intent que describa la acción que se quiere realizar y el sistema lanza la actividad apropiada desde otra aplicación. Si existen multiples actividades

que pueden manejar el intent, el usuario podrá elegir la que quiere usar. Por ejemplo, si quiere permitir al usuario enviar un mensaje, puede crear el siguiente intent:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

El EXTRA_EMAIL añadido al intent es un string array de direcciones de email a los que se debería mandar el email. Cuando una aplicación de email responde a este intent, lee el string array suministrado en el extra y lo mete en el campo "to" del formulario de composición del email. En esta situación, la actividad del email de la aplicación se lanza y cuando el usuario ha terminado se relanza la actividad inicial.

Lanzar una actividad para un resultado

A veces, se quiere recibir un resultado de la actividad que se lanza. En este caso, se ejecuta la actividad llamando al método `startActivityForResult()` (en vez de `startActivity()`). Para recibir el resultado de la actividad, hay que implementar el método callback `onActivityResult()`. Cuando la actividad es realizada, retorna un resultado en un Intent al método `onActivityResult()`.

Por ejemplo, si quiere que el usuario elija uno de sus contactos para que la actividad pueda utilizar esa información del contacto, puede crear un intent y manejar el resultado;

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider URI
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST); }

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode ==
    PICK_CONTACT_REQUEST) {
        // Perform a query to the contact's content provider for the contact's name
        Cursor cursor = getContentResolver().query(data.getData(),
        new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}
```

Este ejemplo nos muestra la lógica básica que se debe utilizar en el método `onActivityResult()` para manejar el resultado de la actividad. La primera condición checkea si la petición se ha realizado con éxito—si así ha sido, entonces el `resultCode` será `RESULT_OK`—y si la petición a la que está respondiendo el resultado es conocida—en este caso, `requestCode` machea el segundo parámetro enviado por el método `startActivityForResult()`. Desde ahí, el código maneja el resultado de la actividad metiendo los datos devueltos a través de una query en una `queryIntent` (el parámetro `datos`).

Lo que ocurre es que un ContentResolver realiza una query a través del content provider que devuelve un Cursor que permite leer los datos enviados en el query.

Cerrar una actividad

Se puede cerrar una actividad llamando al método finish(). También se puede cerrar una actividad diferente que se haya arrancado previamente, llamando al método finishActivity().

Nota: En la mayoría de los casos, no se debería explícitamente cerrar una actividad utilizando estos métodos. Tal y como se explica en la siguiente sección sobre el ciclo de vida de la actividad, el sistema Android gestiona la vida de la actividad, por lo que no se necesita cerrar las actividades. Llamar a estos métodos podría tener un efecto adverso sobre el uso esperado y sólo debería utilizarse cuando no se quiera que el usuario vuelva a la instancia de esta actividad.

Ciclo de vida de la actividad

Gestionar el ciclo de vida de las actividades implementando los métodos callback es crucial para desarrollar una aplicación fuerte y flexible. El ciclo de vida de la actividad está directamente afectado por su asociación con otras actividades, con sus tareas y con su back stack.

Una actividad puede existir en tres estados:

Reanudada

La actividad está en primer plano de la pantalla y tiene el focus del usuario. (A este estado también se le llama "running".)

En pausa

Otra actividad está en primer plano y tiene el focus, pero sigue siendo visible. Es decir, otra actividad es visible encima de esta y esa actividad es parcialmente transparente y no cubre la pantalla totalmente. Una actividad en pausa está viva totalmente (el objeto Actividad es retenido en la memoria, mantiene la información sobre el estado y el miembro, y permanece unido al manager de la ventana), pero puede ser terminada por el sistema en situaciones de baja memoria.

Parada

La actividad está completamente oculta por otra actividad (la actividad está en "segundo plano"). Una actividad parada está también viva (el objeto Actividad es retenido en la memoria, mantiene toda la información del estado y del miembro, pero no está unido al gestor de la ventana. Sin embargo, no es visible al usuario y puede ser terminada por el sistema cuando se necesite memoria en algún otro lugar.

Si una actividad están en pausa o parada, el sistema puede sacarlo de la memoria bien preguntando si la termina (llamado al método finish()), o simplemente matando el proceso. Cuando la actividad es abierta de nuevo (después de haber sido terminada o muerta), debe de ser creada de nuevo.

Implementación de los callbacks de los ciclos de vida

Cuando una actividad transiciona dentro o fuera de los diferentes estados descritos anteriormente, se notifica a través de diferentes métodos callback. Todos los métodos callback son hooks que puedes sobrescribir para que hagan el trabajo adecuado cuando el estado de la actividad cambie. El siguiente esqueleto de la actividad incluye cada uno de los métodos fundamentales del ciclo de vida:

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus (this activity is about to be
        "paused").
    }
    @Override
    protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }
}
```

Nota: La implementación de estos métodos del ciclo de vida deben llamar siempre a la implementación de la superclase antes de llevar a cabo ninguna tarea, tal y como se muestra en los ejemplos anteriores.

Estos métodos en conjunto, definen el ciclo de vida completo de una actividad.

Mediante la implementación de estos métodos, se pueden monitorizar tres bucles anidados en el ciclo de vida de la actividad:

- El tiempo de vida completo de una actividad ocurre entre la llamada al método `onCreate()` y la llamada al método `onDestroy()`. La actividad debería realizar el setup del estado "global" (como la definición del layout) en el método `onCreate()`, y soltar el resto de los recursos en el método `onDestroy()`. Por ejemplo, si la actividad tiene un hilo activo en segundo plano para descargar datos desde la red, puede crear ese hilo en el método `onCreate()` y después parar el hilo en el método `onDestroy()`
- El tiempo de vida visible de una actividad ocurre entre la llamada a `onStart()` y la llamada al `onStop()`. En este tiempo, el usuario puede ver la actividad en la pantalla e interactuar con ella. Por ejemplo, se llama al `onStop()` cuando se lanza una nueva actividad y esta ya no es visible. Entre estos dos métodos, se pueden mantener los recursos necesarios para mostrar la actividad al usuario. Por ejemplo, puedes registrar un `BroadcastReceiver` en `onStart()` para monitorizar los cambios que impactan en el IU, y anular el registro en `onStop()` cuando el usuario ya no pueda ver lo que se está mostrando. El sistema puede llamar a `onStart()` y a `onStop()` varias veces durante el ciclo de vida de la actividad, mientras que la actividad alterna entre visible y no visible al usuario.
- El tiempo de vida en primer plano de una actividad ocurre entre la llamada al `onResume()` y la llamada al `onPause()`. Durante este tiempo, la actividad está por delante de todas las demás actividades en la pantalla y tiene el focus del usuario. Una actividad puede frecuentemente cambiar dentro y fuera del primer plano—por ejemplo, se llama al `onPause()` cuando el dispositivo entra en suspensión o cuando aparece un diálogo. Como este estado puede cambiar a menudo, el código en estos dos métodos debería ser suficientemente ligero para evitar transiciones lentas que hagan al usuario esperar.

La Figura 1 ilustra estos bucles y las rutas que una actividad puede hacer entre estados. Los rectángulos representan los métodos callback que se pueden implementar para realizar las operaciones cuando la actividad cambia entre estados.

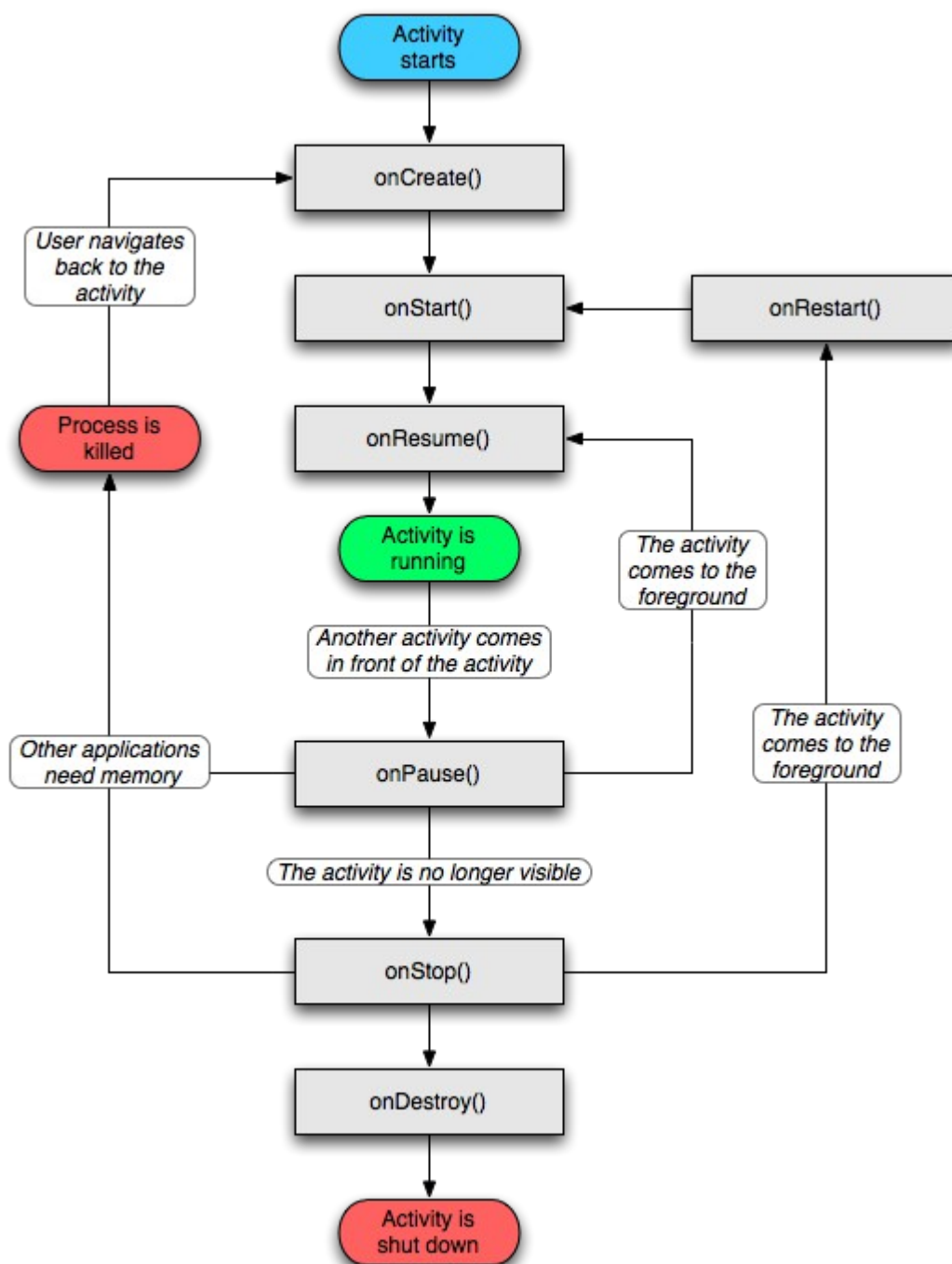


Figura 1. Ciclo de vida de la actividad

Los mismo métodos callback del ciclo de vida están listados en la tabla 1, que describe cada uno de los métodos callback con más detalle y localiza cada uno de ellos dentro del ciclo de vida global de la actividad, incluyendo la posibilidad de que el sistema puede terminar la actividad después de que el método callback finalice.

Tabla 1. Resumen de los métodos callback del ciclo de vida.

Método		Descripción	Terminable	Siguiente
onCreate()		Se llama cuando la actividad recién se crea. Aquí es donde se debería hacer el setup estático—la creación de vistas, la vinculación de los datos a las listas, etc. A este método se le pasa un objeto Bundle conteniendo el estado previo de la actividad, si el estado fue capturado. Seguido por el método onStart()	No	onStart()
onRestart()		Llamado después de que la actividad haya sido parada, y antes de que vuelva a empezar. Seguido por el método onStart()	No	onStart()
onStart()		Llamado justo antes de que la actividad sea visible para el usuario. Seguido por el método onResume() si la actividad funciona en primer plano, o onStop() si es invisible.	No	onResume() o onStop()
onResume()		Llamado justo antes de que la actividad empiece a interactuar con el usuario. En este punto la actividad está arriba del stack de la actividad, con el input del usuario yendo hacia él. Seguido por el método onPause().	No	onPause()
onPause()		Llamado cuando el sistema está por reanudar otra actividad. Este método se usa para hacer commit de cambios no guardados a datos de persistencia, parar animaciones y otras funciones que puedan estar consumiendo CPU, etc. Todo aquello que haga lo debería hacer rápido, ya que la siguiente actividad no se reanudará hasta que no vuelva. Seguido por onResume() si la actividad retorna al primer plano o por onStop() si se hace invisible al usuario.	Si	onResume() o onStop()
onStop()		Llamado cuando la actividad ya no es visible al usuario. Esto	Si	onRestart() o

Método		Descripción	Terminable	Siguiente
		puede ocurrir porque se esté destruyendo, o porque otra actividad (bien una existente o bien una nueva) ha sido reanudada y la esté cubriendo. Seguido por onRestart() si la actividad vuelve a interactuar con el usuario, o por onDestroy() si la actividad se va.		onDestroy()
onDestroy()		Llamado antes de que la actividad sea destruida. Esta es la última llamada recibida por la actividad. Se podría llamar porque la actividad se esté terminando (se ha llamado al método finish()), o porque el sistema esté destruyendo temporalmente la instancia de la actividad para guardar espacio. Se puede diferenciar entre estos dos escenarios con el método isFinishing().	Si	nada

La columna etiquetada "Terminable" indica si el sistema puede o no terminar el proceso en cualquier momento después de que el método vuelva, sin ejecutar otra línea del código de la actividad. Tres métodos están marcados con "si": (onPause(), onStop(), y onDestroy()). Esto es porque onPause() es el primero de los tres, (una vez que la actividad se ha creado) onPause() es el último método con la llamada garantizada antes de que se termine— si el sistema debe recuperar memoria en una emergencia, entonces el método onStop() y onDestroy() no tienen porque ser llamados. Por lo tanto se debería utilizar onPause() para almacenar datos de persistencia importantes. Sin embargo, se debería ser selectivo sobre que información se debe retener durante onPause(), ya que cualquier procedimiento bloqueante en este método bloquea la transición a la siguiente actividad y ralentiza las acciones del usuario.

Los métodos que están marcados con un "No" en la columna Terminable protegen el proceso y no podrá ser terminado desde el momento en el que son llamados. Así, una actividad se puede terminar desde el momento que el método onPause() retorna hasta que el método onResume() es llamado. No podrá ser de nuevo terminado hasta que onPause() es llamado de nuevo y retorna.

Nota: Una actividad que no es técnicamente "terminable" por su definición en la tabla 1 todavía puede ser terminada por el sistema—pero eso sólo ocurrirá en circunstancias extremas cuando no existe otro recurso.

Salvar el estado de la actividad

La introducción al Manejo del ciclo de vida de la Actividad menciona brevemente que cuando una actividad está en pausa o parada, el estado de la actividad es retenido. Esto es verdad dado que el objeto Actividad está todavía retenido en memoria cuando está en pausa o parado—toda la información sobre sus elementos y estado actual está todavía vivo. Así, cualquier cambio introducido por el usuario dentro de la actividad es retenido en la memoria, para que cuando la actividad retorne al primer plano (cuando se "reanude"), estos cambios todavía estén allí.

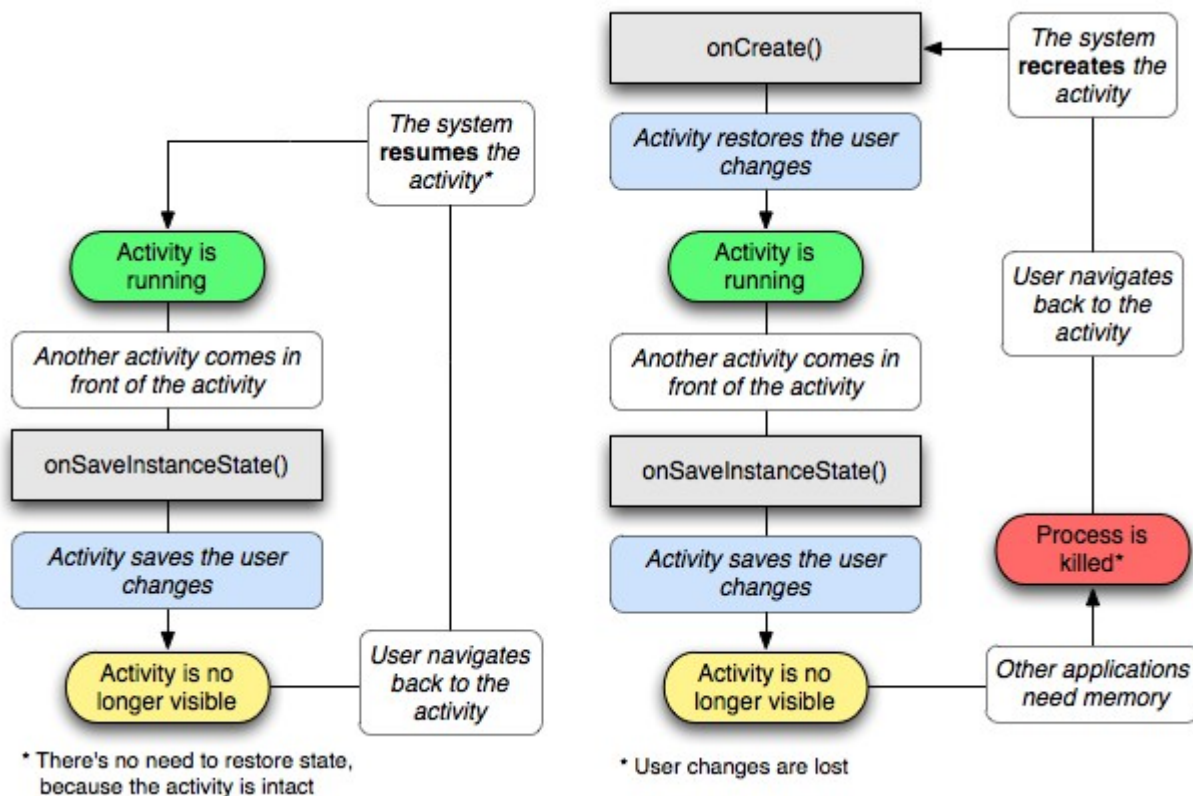


Figura 2. Las dos maneras en las que una actividad devuelve el focus al usuario con su estado intacto son cuando la actividad está parada, reanudada y el estado de la actividad se mantiene intacto (izquierda), o cuando la actividad es destruida, creada de nuevo y la actividad ha de restaurar el estado previo (derecha).

Sin embargo, cuando el sistema destruye una actividad para recuperar memoria, el objeto Actividad es destruido, por lo que el sistema no puede simplemente reanudarse con el estado intacto. En vez de esto, el sistema debe recrear el objeto Actividad si el usuario navega de vuelta. Sin embargo, el usuario no es consciente que el sistema ha destruido la actividad y que la ha vuelto a crear, y por lo tanto, espera que la actividad esté como estaba. En esta situación, se puede garantizar que la información importante sobre el estado de la actividad se preserve implementando un método callback adicional que permite guardar la información sobre el estado de la actividad y después restaurarla cuando el sistema vuelva a crear la actividad.

El método callback en el que se puede guardar la información sobre el actual estado de la actividad es `onSaveInstanceState()`. El sistema llama a este método antes de hacer a la actividad vulnerable de ser destruida y le pasa un

objeto Bundle. El Bundle es donde se puede almacenar información sobre el estado de la actividad como pares de name-value, utilizando métodos como `putString()`. Si el sistema termina el proceso de la actividad y el usuario retorna a la actividad, el sistema pasa el Bundle a `onCreate()` para que se pueda restaurar el estado de la actividad que se guardó durante `onSaveInstanceState()`. Si no hay información del estado que restaurar, entonces el Bundle que se pasa a `onCreate()` es null.

Nota: No hay garantía de que `onSaveInstanceState()` sea llamado antes de que la actividad sea destruida, dado que hay casos en el no es necesario guardar el estado (como cuando el usuario abandona la actividad con la tecla BACK, o en el caso de que el usuario está explícitamente cerrando la actividad). Si se llama al método, siempre se llama antes del método `onStop()` y posiblemente antes del método `onPause()`.

Sin embargo, si no se hace nada y no se implementa `onSaveInstanceState()`, el estado de la actividad se restaura a través de la implementación por defecto de la llamada al método `onSaveInstanceState()` de la clase Actividad. Específicamente, la implementación por defecto llama al método `onSaveInstanceState()` por cada Vista del layout, lo que permite a cada vista suministrar información de si misma que debería ser guardada. Casi todos los widgets en el framework de Android implementan este método, así, cualquier cambio visible a la interfaz de usuario es automáticamente guardado y restaurado cuando la actividad es creada de nuevo. Por ejemplo, el widget `EditText` guarda todo el texto introducido por el usuario y el widget `CheckBox` guarda si está seleccionado o no. Lo único que es necesario, es el suministro de un ID único (con el atributo `android:id`) por cada widget que se quiera guardar su estado.

También se puede hacer que una vista del layout explícitamente no guarde su estado seteando el atributo `android:saveEnabled` a "false" o llamando al método `setSaveEnabled()`. Normalmente, no se debería deshabilitar, pero se puede hacer si se quiere restaurar el estado de la actividad de la IU de forma diferente.

A pesar de que la implementación por defecto del método `onSaveInstanceState()` guarda información útil sobre la actividad del IU, se puede necesitar sobrescribirlo para guardar información adicional. Por ejemplo, se puede necesitar guardar valores de elementos que han cambiado durante la vida de la actividad (que pueden correlacionarse con valores restaurados en la IU, pero que los elementos que guardan esos valores de la IU no son restaurados por defecto).

Dado que la implementación por defecto de `onSaveInstanceState()` ayuda a guardar el estado de la IU, si se sobrescribe el método para guardar información adicional sobre el estado, se debería llamar siempre a la implementación de la superclase del método `onSaveInstanceState()` antes de nada.

Nota: Dado que `onSaveInstanceState()` no garantiza la llamada a este método, se debería utilizar solo para guardar el estado de transición de la actividad (el estado de la IU)—no se debería utilizar para almacenar datos de forma persistente. En su lugar, se debería utilizar el método `onPause()` para

almacenar datos de forma persistente (como los datos que deberían ser guardados en una base de datos) cuando el usuario abandona la actividad.

Una buena manera para probar la habilidad de la aplicación para restaurar el estado es rotar el dispositivo para que la orientación de la pantalla cambie. Cuando la orientación de la pantalla cambia, el sistema destruye y vuelve a crear la actividad para aplicar recursos alternativos que estén disponibles para la nueva orientación. Por esta razón, es muy importante que la actividad restaure completamente su estado cuando se vuelva a crear, ya que los usuarios rotan regularmente la pantalla mientras utiliza la aplicación.

Manejar los cambios de la configuración

Algunas configuraciones de dispositivos pueden cambiar en tiempo de ejecución (como la orientación de la pantalla, disponibilidad del teclado y el lenguaje). Cuando ese cambio ocurre, Android vuelve a arrancar la actividad que se está ejecutando (se llama al método `onDestroy()`, seguido de `onCreate()`). El comportamiento del rearranque está diseñado para ayudar a la adaptación de la aplicación a las nuevas configuraciones mediante la automática recarga de la aplicación con recursos alternativos. Si se diseña correctamente la aplicación para que gestione este evento, será más resistente hacia las eventos inesperados del ciclo de vida de la actividad.

La mejor manera de gestionar un cambio de configuración, como el cambio en la orientación de la pantalla, es preservar el estado de la aplicación utilizando `onSaveInstanceState()` y `onRestoreInstanceState()` (o `onCreate()`), tal y como es descrito en la sección anterior.

Coordinación de actividades

Cuando una actividad arranca otra, las dos sufren transiciones en sus ciclos de vida. La primera actividad se queda en pausa y se para (no se parará si sigue visible en segundo plano), mientras que la otra actividad es creada. En el caso de que estas actividades compartan datos guardados en disco o en otro lugar es importante entender que la primera actividad no se para completamente antes de que se cree la segunda. Es más, el proceso del arranque de la segunda actividad se sobrepone con el proceso de parar de la primera actividad.

El orden de los callbacks del ciclo de vida está bien definido, particularmente cuando las dos actividades están en el mismo proceso y una arranca a la otra. Aquí está el orden de las operaciones cuando la Actividad A arranca a la Actividad B:

1. El método de la Actividad A `onPause()` se ejecuta.
 2. Los métodos de la Actividad B `onCreate()`, `onStart()`, y `onResume()` se ejecutan en una secuencia. (La actividad B tiene el focus.)
 3. Si la Actividad A no es visible en la pantalla, su método `onStop()` se ejecuta.
- Esta secuencia predecible de los callbacks del ciclo de vida permiten que se maneje la transición de información de una actividad a otra. Por ejemplo, si se quiere escribir en base de datos cuando la primera actividad pare para que la siguiente actividad lo pueda leer, entonces se debería escribir en base de datos en el método `onPause()` en vez de en el método `onStop()`.

Tareas y Back Stack

Una aplicación contiene diferentes actividades. Cada actividad debería ser diseñada entorno a una acción específica que puede realizar el usuario y que pueden iniciar otras actividades. Por ejemplo, una aplicación de correo electrónico puede tener una actividad para mostrar una lista de los correos electrónicos nuevos. Cuando el usuario selecciona un email, se abre una nueva actividad para ver ese mail.

Una actividad también puede iniciar actividades existentes en otras aplicaciones del dispositivo. Por ejemplo, si la aplicación quiere mandar un correo electrónico, se puede definir un intent para que realice una acción "send" incluyendo datos, como la dirección de correo electrónico y el mensaje. En ese momento se abre una actividad de otra aplicación que va a gestionar este tipo de intent. En este caso, el intent es para mandar un correo electrónico, por lo que se inicia una actividad "redactar" de correo electrónico de la aplicación (si varias actividades mantienen al mismo intent, el sistema permite al usuario seleccionar cual quiere utilizar). Cuando se envía el email, se reanuda la actividad y parece como si la actividad de email forma parte de la aplicación. Aunque las actividades pueden ser de diferentes aplicaciones, Android mantiene esto transparente al usuario guardando ambas actividades en la misma tarea.

Una tarea es una colección de actividades con las cuales interactúa el usuario cuando quiere realizar determinado trabajo. Las actividades son ordenadas en un stack (el "back stack"); en el orden en el que se abre cada actividad.

La pantalla Inicio es el lugar de inicio para la mayoría de las tareas del dispositivo. Cuando el usuario hace click en un icono, la tarea de la aplicación va a un primer plano. Si no existe una tarea (la aplicación no se ha usado recientemente) se crea una nueva y la actividad "main" de la aplicación se abre como la actividad raíz en la pila.

Cuando la actividad en curso inicia otra, la nueva actividad es empujada arriba de la pila y obtiene el focus. La actividad anterior permanece en la pila, pero está parada. Cuando la actividad para, el sistema retiene el estado actual de la interfaz de usuario. Cuando el usuario hace click en la tecla BACK, la actividad en curso salta desde arriba de la pila (la actividad se elimina) y la actividad previa se reanuda (se restaura el estado previo de su IU). Las actividades en la pila nunca se reorganizan, solo se meten o sacan de la pila— se meten en la pila cuando se inicia la actividad en curso y son sacadas cuando el usuario la abandona utilizando la tecla BACK. De esta manera, el back stack funciona mediante la estructura de objeto; "último en entrar, primero en salir". La figura 1 muestra este comportamiento mediante una cronología de eventos mostrando, en cada momento, el progreso de las actividades y el back stack en curso.

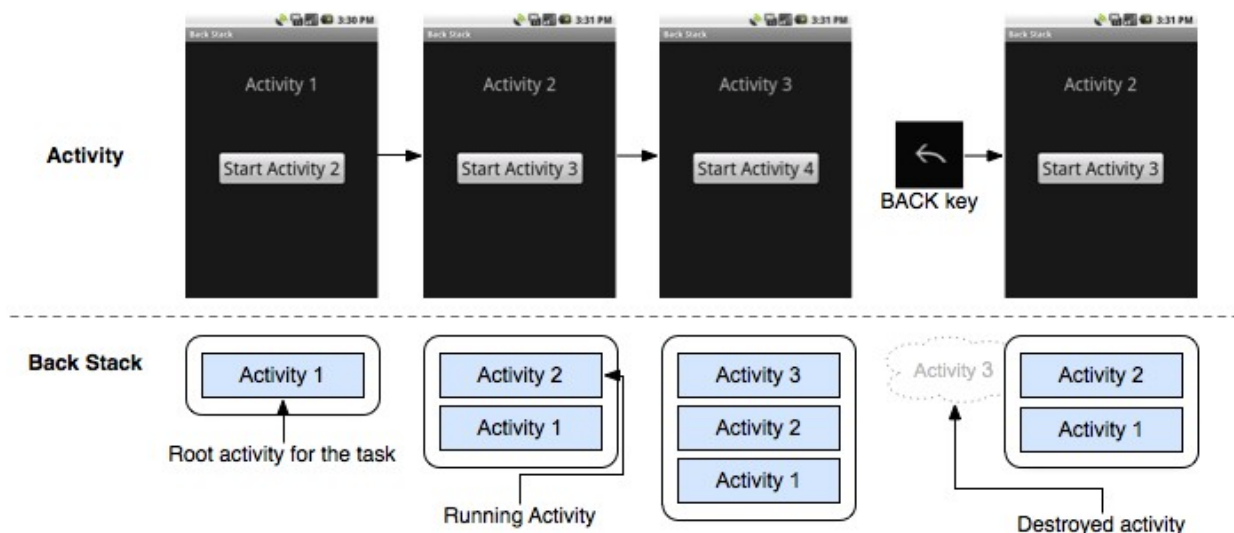


Figura 1. Representación de como cada nueva actividad en una tarea añade un item al back stack. Cuando el usuario hace click en la tecla BACK, se elimina la actividad en curso y se reanuda la actividad previa.

Si el usuario sigue haciendo click en la tecla BACK, cada actividad de la pila se saca, revelando la anterior, hasta que el usuario retorna a la pantalla de Inicio (o a cualquier actividad que estuviera ejecutándose en el comienzo de la tarea). Cuando todas las actividades son sacadas de la pila, la tarea deja de existir.

Una tarea es una unidad cohesiva que puede moverse a un "segundo plano" cuando el usuario comienza una nueva tarea o ir a la pantalla Inicio, a través de la tecla HOME. Mientras está en un segundo plano, todas las actividades en la tarea se paran, pero el back stack para la tarea se mantiene intacto— como se muestra en la figura 2, la tarea ha perdido el focus mientras que se está ejecutando otra tarea. Una tarea puede retornar a un "primer plano" por lo que los usuarios pueden volver al lugar donde lo había dejado. Imagínese, por ejemplo, que la tarea en curso (Tarea A) tiene tres actividades en su pila—dos por debajo de la actividad en curso. El usuario hace click en la tecla HOME, e inicia una nueva aplicación. Cuando la pantalla de inicio aparece, la tarea A se sitúa en un segundo plano. Cuando se inicia la nueva aplicación, el sistema inicia una tarea para esa aplicación (Tarea B) con su propia pila de actividades.

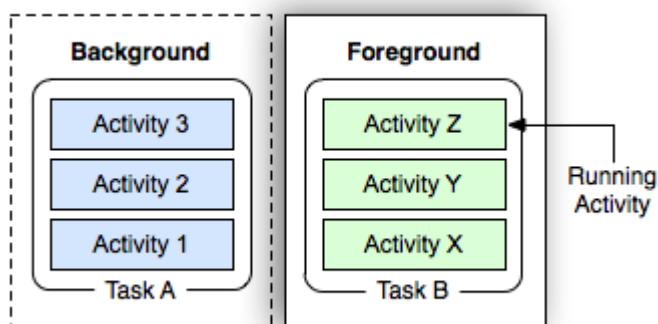


Figura 2. Dos tareas: Tarea A que está en un segundo plano, esperando a ser reanudada, mientras que la tarea B recibe la interacción del usuario en primer plano.

Después de interactuar con esa aplicación, el usuario retorna a la página de Inicio y selecciona la aplicación que originariamente había iniciado a la Tarea A. Ahora, la tarea A va a un primer plano— las tres actividades están intactas en su pila y se reanuda la actividad que está arriba de la pila. En este punto, el usuario puede volver a la tarea B si va a Inicio y selecciona el icono de la aplicación que empezó la tarea (o si hace click y mantiene la tecla Inicio para ver las tareas recientes y selecciona una). Esto es un ejemplo de multitasking en Android.

Nota: Se pueden manejar varias tareas en segundo plano. Sin embargo, si el usuario ejecuta varias tareas en segundo plano a la misma vez, el sistema puede empezar a eliminar algunas de estas actividades para recuperar memoria, perdiéndose con ello el estado de las actividades. Ver la siguiente sección sobre el Estado de la Actividad.

Dado que las actividades en el back stack nunca son reorganizadas, si la aplicación permite que los usuarios inicien una actividad concreta desde más de una actividad, se crea una nueva instancia de esa actividad y se mete en la pila (en vez de traer a la parte de arriba de la pila a una instancia anterior de la actividad). De esta manera, una actividad de la aplicación puede ser instanciada varias veces (incluso desde tareas diferentes), tal y como se muestra en la figura 3. Así, si el usuario navega hacia atrás usando la tecla BACK, aparecerá cada instancia de la actividad en el orden en el que fueron abiertas (cada una con su propio estado de IU). Sin embargo, se puede modificar este comportamiento si no se quiere instanciar una actividad más de una vez.

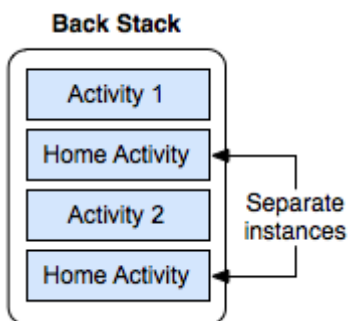


Figura 3. Una sola actividad es instanciada varias veces.

A continuación tenemos un resumen del comportamiento por defecto de las actividades y tareas:

- Cuando la actividad A inicia a la actividad B, la actividad A se para, pero el sistema mantiene su estado (como la posición del scroll y el texto introducido en los formularios). Si el usuario hace click en la tecla BACK durante la actividad B, la actividad A se reanuda con su estado restablecido.
- Cuando el usuario abandona una tarea haciendo click en la tecla Inicio, la actividad en curso se para y su tarea se mueve a un segundo plano. El sistema retiene en la tarea el estado de cada una de las actividades. Si el usuario reanuda la tarea seleccionando el icono que inicia la tarea, la tarea se mueve a un primer plano y reanuda la actividad de encima de la pila.
- Si el usuario hace click en la tecla BACK, la actividad en curso sale de la pila y es eliminada. Se reanuda la actividad anterior. Cuando una actividad es eliminada, el sistema no conserva el estado de la actividad.

- Las actividades pueden ser instanciadas varias veces, incluso desde otras tareas.

Guardar el estado de la actividad

Como se ha visto anteriormente, el comportamiento por defecto del sistema conserva el estado de una actividad cuando se para. De esta manera, cuando los usuarios navegan hacia atrás a una actividad anterior, la IU aparece de la manera en la que la dejaron. Sin embargo, se puede—y se debe— conservar el estado de las actividades utilizando métodos callback, en el caso de que la actividad sea destruida y deba ser recreada.

Cuando el sistema para una de las actividades (como cuando una actividad nueva se inicia o la tarea se mueve a un segundo plano), el sistema puede llegar a destruir la actividad completamente si necesita memoria. Cuando esto ocurre, se pierde la información sobre el estado de la actividad. Si esto ocurre, el sistema sabe que la actividad tiene un lugar en el back stack, pero cuando la actividad se mueve a la parte de arriba de la pila, el sistema debe recrearla (en vez de reanudarla). Para evitar perder el trabajo del usuario, se debe conservar proactivamente implementando el método callback `onSaveInstanceState()` de `Activity`.

Para más información sobre como guardar el estado de la actividad, ver el documento [Actividades](#).

Gestión tareas Android

La manera en la que Android gestiona las tareas y el back stack, —colocando todas las actividades iniciadas en la misma tarea y en una pila— "último en entrar, primero en salir"; funciona muy bien para la mayoría de las aplicaciones y no hay que preocuparse sobre como están asociadas las actividades con las tareas o como existen en el back stack. Sin embargo, se puede querer interrumpir el comportamiento normal. Quizás se necesite en la aplicación, que una actividad inicie una nueva tarea cuando se inicie (en vez de colocarse dentro de la tarea en curso); o, que cuando se inicie una actividad, se traiga hacia delante una instancia ya existente (en vez de crear una nueva instancia en la parte superior del back stack); o, que cuando el usuario deja la tarea, se eliminen todas las actividades exceptuando la actividad raíz.

Se puede hacer todo esto y más, con los atributos del elemento `<activity>` del manifest y con los flags en el intent que se pasan al método `startActivity()`.

A este efecto, los principales atributos `<activity>` que se pueden utilizar son:

- `taskAffinity`
- `launchMode`
- `allowTaskReparenting`
- `clearTaskOnLaunch`
- `alwaysRetainTaskState`
- `finishOnTaskLaunch`

Y los principales flags del intent que se pueden utilizar son:

- FLAG_ACTIVITY_NEW_TASK
- FLAG_ACTIVITY_CLEAR_TOP
- FLAG_ACTIVITY_SINGLE_TOP

En las siguientes secciones, vamos a ver como se pueden utilizar estos atributos del manifest y los flags del intent para definir como las actividades son asociadas con las tareas y como se comportan en el back stack.

Precaución: La mayoría de las aplicaciones no deberían interrumpir su comportamiento normal de actividades y tareas. Si se determina que es necesario hacerlo, hay que asegurarse que se testea la usabilidad de la actividad durante el inicio y cuando se navega de vuelta a ella, utilizando BACK desde otra actividad o tarea. Hay que asegurarse que se testea los diferentes comportamientos de navegación que pueden ser conflictivos con el comportamiento esperado del usuario.

Definir modo de inicio

Los modos de inicio te permiten definir como una nueva instancia de una actividad está asociada con la tarea en curso. Se pueden definir los diferentes modos de inicio de dos maneras:

•Utilizando el archivo manifest

Cuando se declara una actividad en el archivo manifest, se puede especificar como debe asociarse la actividad con las tareas.

•Utilizando flags de un intent

Cuando se llama al método startActivity(), se puede incluir un flag en el Intent que declara como (o si) la nueva actividad debería asociarse con la tarea en curso.

Así, si la Actividad A inicia la Actividad B, la Actividad B puede definir en su manifest como debe asociarse con la tarea en curso y la Actividad A también puede pedir como la Actividad B se debería asociar con la tarea en curso. Si ambas actividades definen como la Actividad B debe asociarse con una tarea, entonces la petición de la Actividad A (definida en el intent) será tomada en cuenta antes que la petición de la Actividad B (definida en su manifest).

Nota: Algunos de los modos de inicio disponibles en el manifest no están disponibles como flags de un intent y de la misma manera, algunos de los modos de inicio disponibles como flags de un intent no se pueden definir en el manifest.

Utilizar el archivo manifest

Cuando se declara una actividad en el archivo manifest, se puede especificar como se debe asociar la actividad con una tarea utilizando el atributo launchMode del elemento <activity>.

El atributo launchMode especifica una instrucción de como la actividad debe ser iniciada en una tarea. Existen cuatro diferentes modos de inicio que se pueden asignar al atributo launchMode :

1."standard" (modo por defecto)

El sistema crea una nueva instancia de la actividad en la tarea desde la cual se inició y le redirecciona el intent. La actividad puede ser instanciada varias

veces, cada instancia puede pertenecer a tareas diferentes y una tarea puede tener varias instancias.

2."singleTop"

Si una instancia de la actividad ya existe en la parte superior de la tarea en curso, el sistema redirecciona el intent hasta esa instancia a través de una llamada a su método `onNewIntent()`, en vez de crear una nueva instancia de la actividad. La actividad puede ser instanciada varias veces, cada instancia puede pertenecer a diferentes tareas, y una tarea puede tener varias instancias (pero solo si la actividad en la parte superior de la back stack no es una instancia ya existente de la actividad).

Por ejemplo, una tarea del back stack consta de una actividad raíz A con actividades B, C y D encima (la pila es A-B-C-D; D encima). Llega un intent para una actividad de tipo D. Si D tiene el modo por defecto "standard", se inicia una nueva instancia de la clase y la pila se convierte en A-B-C-D-D. Sin embargo, si el modo de inicio de D es "singleTop", la instancia existente de D entrega el intent a través del método `onNewIntent()`, y dado que está en la parte superior de la pila—la pila permanece como A-B-C-D. Sin embargo, si un intent llega para una actividad de tipo B, se añade una nueva instancia de B a la pila, aún si el modo de inicio es "singleTop".

Nota: Cuando se crea una nueva instancia de la actividad, el usuario puede hacer click en BACK para volver a la actividad previa. Pero cuando es una instancia ya existente de la actividad la que gestiona el nuevo intent, el usuario no puede hacer click en BACK para volver al estado que tenía la actividad antes de que el nuevo intent llegara con el `onNewIntent()`.

3."singleTask"

El sistema crea una nueva tarea e instancia la actividad en la raíz de la nueva actividad. Sin embargo, si una instancia de la actividad ya existe en una tarea por separado, el sistema redirecciona el intent hacia la instancia ya existente a través de la llamada al método `onNewIntent()`, en vez de crear una nueva instancia. Sólo puede existir a la vez una instancia de la actividad.

Nota: Aunque la actividad se inicia en una nueva tarea, la tecla BACK aún retorna al usuario a la actividad anterior.

4."singleInstance"

Igual que "singleTask", excepto que el sistema no inicia ninguna otra actividad dentro de la tarea que guarda la instancia. La actividad siempre es el único miembro de su tarea; cualquier actividad iniciada por esta se abre en una tarea aparte.

Otro ejemplo; el navegador de la aplicación Android declara que la actividad del navegador web se debería abrir en su propia tarea— esto lo hace especificando el modo de inicio `singleTask` en el elemento `<activity>`. Esto significa que si la aplicación pide que un intent abra el navegador de Android, su actividad no se sitúa en la misma tarea que la aplicación. En vez de esto, o bien una nueva tarea se inicia para el navegador o bien, si el navegador ya tiene una tarea ejecutándose en un segundo plano, la tarea se trae a un primer plano para manejar el intent nuevo.

Sin tener en cuenta si una actividad se inicia en una nueva tarea o en la misma

tarea en la que se inició la actividad, la tecla BACK siempre lleva al usuario a la actividad anterior. Sin embargo, si se inicia una actividad desde una tarea (Tarea A) que especifica el modo de inicio `singleTask`, entonces la actividad puede tener una instancia en un segundo plano que pertenece a la tarea que tiene su propio back stack (Tarea B). En este caso, cuando la Tarea B se trae al frente para manejar un nuevo intent, la tecla BACK primero navega hacia atrás a través de las actividades de la Tarea B antes de volver a la actividad que esté más arriba en la pila de la Tarea A. La figura 4 muestra este tipo de escenario.

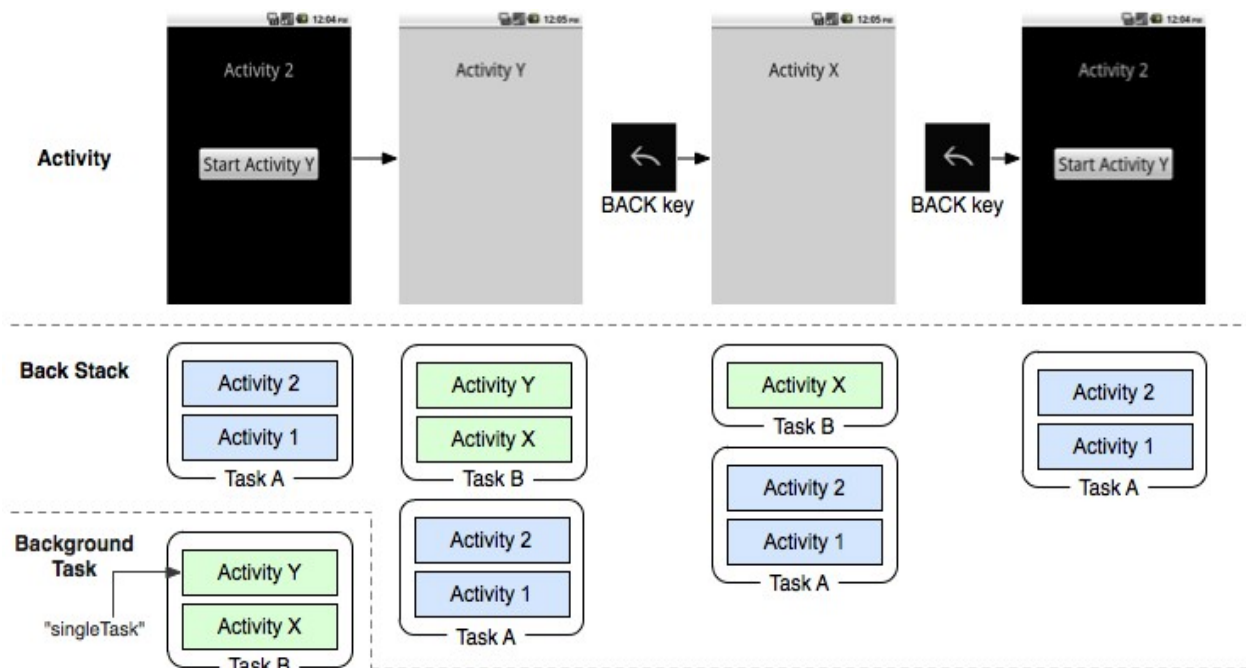


Figura 4. Representación de como una actividad con modo de inicio "singleTask" se añade al back stack. Si la actividad ya forma parte de una tarea en segundo plano con su propio back stack (Tarea B), entonces todo el back stack también viene a un primer plano, por encima de la tarea en curso (Tarea A).

Utilizar flags en los intent

Cuando se inicia una actividad, se puede modificar la asociación por defecto entre una actividad y su tarea mediante la inclusión de flags en el intent que quieras pasarle al método `startActivity()`. Los flags que se pueden utilizar para modificar el comportamiento por defecto son:

FLAG_ACTIVITY_NEW_TASK

Inicia la actividad en una nueva tarea. Si una tarea ya se está ejecutando para la actividad que se está iniciando, la tarea se trae a un primer plano con su último estado reanudado y la actividad recibe el nuevo intent a través del método `onNewIntent()`.

Esto consigue el mismo comportamiento que el valor del `launchMode` de "singleTask", del que hemos hablado en la sección anterior.

FLAG_ACTIVITY_SINGLE_TOP

Si la actividad que se inicia es la actividad en curso (la que está en la parte superior del back stack), entonces la instancia existente recibe una llamada al método `onNewIntent()`, en vez de crear una nueva instancia de la actividad.

Esto consigue el mismo comportamiento que el valor del `launchMode` de "singleTask".

FLAG_ACTIVITY_CLEAR_TOP

Si la actividad que se inicia ya se está ejecutando en la tarea en curso, en vez de iniciar una nueva instancia de la actividad, todas las actividades encima suya son destruidas y este intent es entregado a la instancia reanudada de la actividad (que ahora está en la parte superior) a través del método `onNewIntent()`.

No hay un valor del atributo `launchMode` que consigue este comportamiento.

FLAG_ACTIVITY_CLEAR_TOP

Se utiliza normalmente junto con `FLAG_ACTIVITY_NEW_TASK`. Cuando se usan juntos, estos flags son una manera de colocar una actividad ya existente dentro de otra tarea en una posición donde puede responder al intent.

Nota: Si el modo de inicio de la actividad asignada es "standard", esta también es eliminada de la pila y se inicia una nueva instancia en su lugar para gestionar el intent que llega. Esto es porque siempre se crea una nueva instancia para el nuevo intent cuando el modo de inicio es "standard".

Gestión afinidades

La afinidad indica la tarea a la que una actividad prefiere pertenecer. Por defecto, todas las actividades de la misma aplicación tienen afinidad la una por la otra. Por defecto, todas las actividades de la misma aplicación prefieren estar en la misma tarea. Sin embargo, se puede modificar esta afinidad por defecto de una actividad. Las actividades definidas en aplicaciones diferentes pueden compartir una afinidad, o las actividades definidas en la misma aplicación pueden tener asignadas diferentes afinidades de tareas.

Se pueden modificar las afinidades para cualquier actividad con el atributo `taskAffinity` del elemento `<activity>`.

El atributo `taskAffinity` adopta un valor string, que debe ser único, del nombre por defecto del paquete declarado en el elemento `<manifest>`, ya que el sistema utiliza ese nombre para identificar la afinidad por defecto de la tarea en la aplicación.

La afinidad entra en juego en dos circunstancias:

- Cuando el intent que inicia una actividad contiene el flag `FLAG_ACTIVITY_NEW_TASK`.

Una nueva actividad, es iniciada por defecto en una tarea de la actividad llamada `startActivity()`. Se inserta dentro del mismo back stack que el que llama. Sin embargo, si el intent que se pasa al `startActivity()` contiene el `FLAG_ACTIVITY_NEW_TASK`, el sistema busca una tarea diferente para hospedar a la nueva actividad. Muy a menudo, es una nueva tarea. Sin embargo, no tiene porque serlo. Si ya existe una tarea con la misma afinidad que la nueva actividad, la actividad se inicia dentro de esa tarea. Si no, comienza una nueva tarea.

Si este flag hace que una nueva actividad inicie una nueva tarea y el usuario hace click en HOME para abandonarla, debe existir una manera de que el

usuario navegue de vuelta a la tarea. Algunas entidades (como el gestor de notificaciones) siempre inician las actividades en una tarea externa, por lo que siempre utilizan `FLAG_ACTIVITY_NEW_TASK` en los intents que pasan al método `startActivity()`. Si existe una actividad que puede ser llamada por una entidad externa que pueda utilizar este flag, hay que tener en cuenta que el usuario puede llegar independientemente a la tarea iniciada a través de un icono de inicio (la actividad raíz de la tarea tiene un filtro intent `CATEGORY_LAUNCHER`; ver la sección [Iniciar una tarea](#)).

- Cuando una actividad tiene su atributo `allowTaskReparenting` seteado a `"true"`. En este caso, la actividad puede moverse desde la tarea que inicia a la tarea por la que tiene afinidad, cuando la tarea llega a un primer plano.

Ejemplo; una actividad que reporta las condiciones meteorológicas en las ciudades seleccionadas está definida como parte de una aplicación de viajes. Tiene la misma afinidad que otras actividades en la misma aplicación (la afinidad por defecto de la aplicación) y permite re-parenting con este atributo. Cuando una de las actividades inicia la actividad de informe meteorológicos, inicialmente pertenece a la misma tarea que la actividad. Sin embargo, cuando la tarea de la aplicación de viajes llega a un primer plano, la actividad del informe meteorológico se reasigna a esa tarea y se muestra en ella.

Consejo: Si un archivo `.apk` contiene más de una "aplicación" desde el punto de vista del usuario, es mejor utilizar el atributo `taskAffinity` para asignar diferentes afinidades a las actividades asociadas con cada "aplicación".

Limpiar el back stack

Si el usuario abandona una tarea por mucho tiempo, el sistema limpia todas las actividades de la tarea, excepto la actividad principal. Cuando el usuario retorna a la tarea, solo se restaura la actividad principal. El sistema se comporta de esta manera, porque después de un tiempo determinado, los usuarios normalmente han abandonado lo que estaban haciendo y vuelven a la tarea para iniciar algo nuevo.

Existen algunos atributos de las actividades que se pueden utilizar para modificar este comportamiento:

`alwaysRetainTaskState`

Si en la actividad principal de la tarea este atributo está seteado a `"true"`, el comportamiento por defecto descrito no ocurre. La tarea mantiene todas las actividades en su pila por un largo periodo de tiempo.

`clearTaskOnLaunch`

Si en la actividad principal de la tarea este atributo está seteado a `"true"`, la pila elimina todo excepto la actividad principal cada vez que el usuario abandona la tarea y retorna a ella. Dicho de otra manera, ocurre lo contrario que en `alwaysRetainTaskState`. El usuario siempre retorna a la tarea en su estado inicial, aunque haya abandonado la tarea sólo por un instante.

`finishOnTaskLaunch`

Este atributo es como `clearTaskOnLaunch`, pero trabaja en una sola actividad, no en una tarea completa. También puede hacer que una actividad se vaya, incluyendo la actividad principal. Cuando está seteada

a "true", la actividad forma parte de la tarea sólo durante la sesión en curso. Si el usuario se va y vuelve a la tarea, ya no existirá.

Iniciar una tarea

Se puede configurar una actividad como el punto de entrada para una tarea dándole un intent filter con "android.intent.action.MAIN" como la acción específica y "android.intent.category.LAUNCHER" como la categoría especificada.

Por ejemplo:

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

Un intent filter de este tipo hace que se muestren un icono y una etiqueta para la actividad en el iniciador de la aplicación, dando al usuario una manera de iniciar la actividad y de volver a la tarea que se crea en cualquier momento después del inicio.

Esta segunda habilidad es importante: los usuarios deben poder abandonar una tarea y volver utilizando este iniciador de la actividad. Por esta razón, los dos modos de inicio que marcan las actividades como siempre, iniciando una tarea "singleTask" y "singleInstance", deberían ser utilizadas sólo cuando la actividad tenga un ACTION_MAIN y un filtro CATEGORY_LAUNCHER. ¿Qué pasaría por ejemplo, si falta el filtro? Un intent lanza una actividad "singleTask" iniciando una nueva tarea, y el usuario gasta algún tiempo trabajando en esa tarea. El usuario hace click en HOME. La tarea se manda a un segundo plano y no es visible. Ya que no se representa en el iniciador de la aplicación, el usuario no tiene forma de volver a la tarea.

Para esos casos en los que no se quiere que el usuario pueda volver a una actividad, hay que setear el elemento de la <actividad> finishOnTaskLaunch a "true".

Fragmentos

Un fragmento representa un comportamiento o una parte de la interfaz de usuario en una Actividad. Se pueden combinar múltiples fragmentos en una sola actividad para construir una IU multi-pane y reutilizar un fragmento en múltiples actividades. Se puede pensar que un fragmento es como una sección modular de una actividad, que tiene su propio ciclo de vida, recibe sus propios eventos inputs, y los puede añadir o eliminar mientras la actividad se está ejecutando.

Un fragmento siempre debe estar dentro de una actividad y el ciclo de vida del fragmento está directamente afectado por el ciclo de vida de la actividad principal. Por ejemplo, cuando la actividad está en pausa, también lo están sus fragmentos y cuando la actividad se destruye, también son destruidos sus fragmentos. Sin embargo, cuando una actividad se está ejecutando (está en el

estado reanudado del ciclo de vida), se puede manipular cada fragmento independientemente, por lo que se pueden añadir o eliminar. Cuando se realiza una transacción así con el fragmento, también se puede añadir a un back stack gestionado por la actividad—cada entrada back stack en la actividad es un registro de la realización de una transacción de un fragmento. El back stack permite al usuario revertir una transacción con un fragmento (navegando hacia atrás), haciendo click en la tecla BACK.

Cuando se añade un fragmento como parte del layout de la actividad, vive en un ViewGroup dentro de la jerarquía de una vista de la actividad y define su propio layout de vistas. Se puede insertar un fragmento en el layout de la actividad mediante la declaración del fragmento en el archivo del layout de la actividad, como un elemento <fragment>, o desde el código de la aplicación, añadiéndolo a un ViewGroup existente. Sin embargo, no es necesario que un fragmento sea parte de un layout de una actividad; también se puede utilizar un fragmento como un trabajador invisible para la actividad.

Este documento describe como construir una aplicación para utilizar fragmentos, incluyendo como los fragmentos pueden mantener su estado cuando se añaden al back stack de la actividad, como comparten eventos con la actividad y otros fragmentos de la actividad, como contribuyen a la barra de acción, y más.

Diseños de Interfaz de Usuario dinámicos

Android introdujo fragmentos en Android 3.0 (nivel API "Honeycomb"), principalmente para soportar diseños de IU más dinámicos y flexibles en pantallas grandes, como las tablets. Como una pantalla de un tablet es mucho más grande que la de un teléfono móvil, existe más espacio para combinar e intercambiar componentes de IU. Los fragmentos permiten estos diseños sin que haya que gestionar cambios muy complejos en la jerarquía de la vista. Mediante la división del layout de una actividad en fragmentos, se puede modificar la apariencia de la actividad en tiempo de ejecución y preservar esos cambios en un back stack gestionado por una actividad.

Por ejemplo, una aplicación sobre noticias puede utilizar un fragmento para mostrar una lista de artículos a la izquierda y otro fragmento para mostrar un artículo a la derecha—los dos fragmentos aparecen en la actividad, juntos, y cada uno de los fragmentos tiene su propio conjunto de métodos callback del ciclo de vida y gestiona sus propios eventos input de usuario. Así, en vez de utilizar una actividad para seleccionar un artículo y otra actividad para leer el artículo, el usuario puede seleccionar un artículo y leerlo en la misma actividad, como se ilustra en la figura 1.

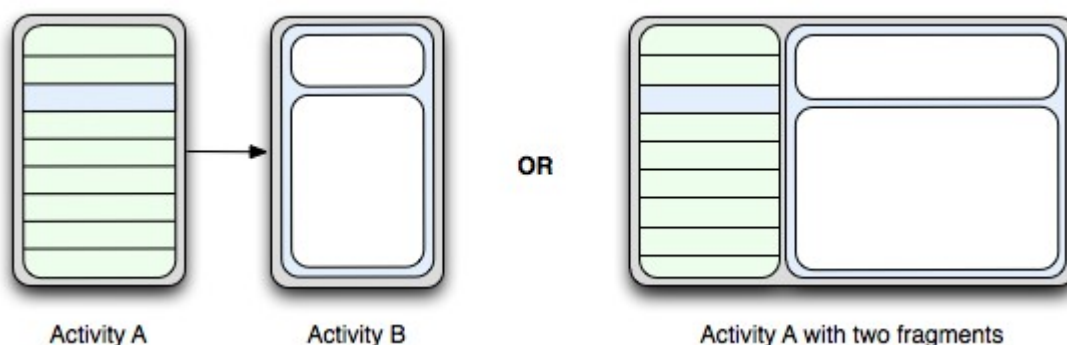


Figura 1. Un ejemplo de como dos módulos de IU que típicamente son separados en dos actividades pueden ser combinados en una sola actividad, utilizando fragmentos.

Un fragmento debería ser un componente modular y reutilizable en la aplicación. Como un fragmento define su propio layout y su propio comportamiento utilizando sus propios callbacks de ciclo de vida, se puede incluir un fragmento en múltiples actividades. Esto es especialmente importante ya que permite adaptar el uso del usuario a diferentes tamaños de pantalla. Por ejemplo, se puede incluir múltiples fragmentos en una actividad sólo cuando el tamaño de la pantalla es suficientemente grande, y cuando no lo es, se lanzan diferentes actividades que utilizan diferentes fragmentos.

Para seguir con el ejemplo de la aplicación de noticias; la aplicación puede incluir dos fragmentos en la Actividad A, cuando se ejecuta en una pantalla extra grande (un tablet, por ejemplo). Sin embargo, en una pantalla de tamaño normal (como por ejemplo, un teléfono), no hay sitio suficiente para los dos fragmentos, por lo que la Actividad A incluye solo el fragmento de la lista de artículos, y cuando el usuario elige un artículo, arranca la Actividad B, que incluye el fragmento que lee el artículo. Así, la aplicación soporta los dos patrones de diseño sugeridos en la figura 1.

Crear un fragmento

Para crear un fragmento, se debe crear una subclase de `Fragment` (o una subclase ya existente). La clase `Fragment` tiene código que se parece mucho a una `Activity`. Contiene métodos callback similares a una actividad, como el `onCreate()`, `onStart()`, `onPause()` y `onStop()`. De hecho, si se está modificando una aplicación Android ya existente para que utilice fragmentos, simplemente hay que mover código de los métodos callback de la actividad a los respectivos métodos callback del fragmento.

Normalmente, se deberían implementar al menos los siguientes métodos del ciclo de vida:

`onCreate()`

El sistema lo llama cuando crea un fragmento. Dentro de la implementación, se debería inicializar componentes esenciales del fragmento que se quiere retener cuando el fragmento está en pausa o parado, y después reanudado.

`onCreateView()`

El sistema lo llama cuando es el momento de que el fragmento dibuje la IU por primera vez. Para dibujar una IU para el fragmento, hay que devolver un `View` desde este método que es la raíz del layout del fragmento. Se puede devolver `null` si el fragmento no suministra una IU.

`onPause()`

El sistema llama a este método como primera indicación de que el usuario está abandonando el fragmento (esto no siempre significa que el fragmento esté siendo destruido). Aquí es donde normalmente se deberían comitear los cambios que quieren permanecer más allá de la actual sesión de usuario (ya que el usuario puede no volver).

La mayoría de las aplicaciones deberían implementar al menos estos tres

métodos para cada fragmento, pero también hay otros métodos callback que se deberían utilizar para manejar diferentes fases del ciclo de vida de los fragmentos. Todos los métodos callbacks del ciclo de vida son estudiados en profundidad más adelante, en la sección sobre Manejar el ciclo de vida de los fragmentos.

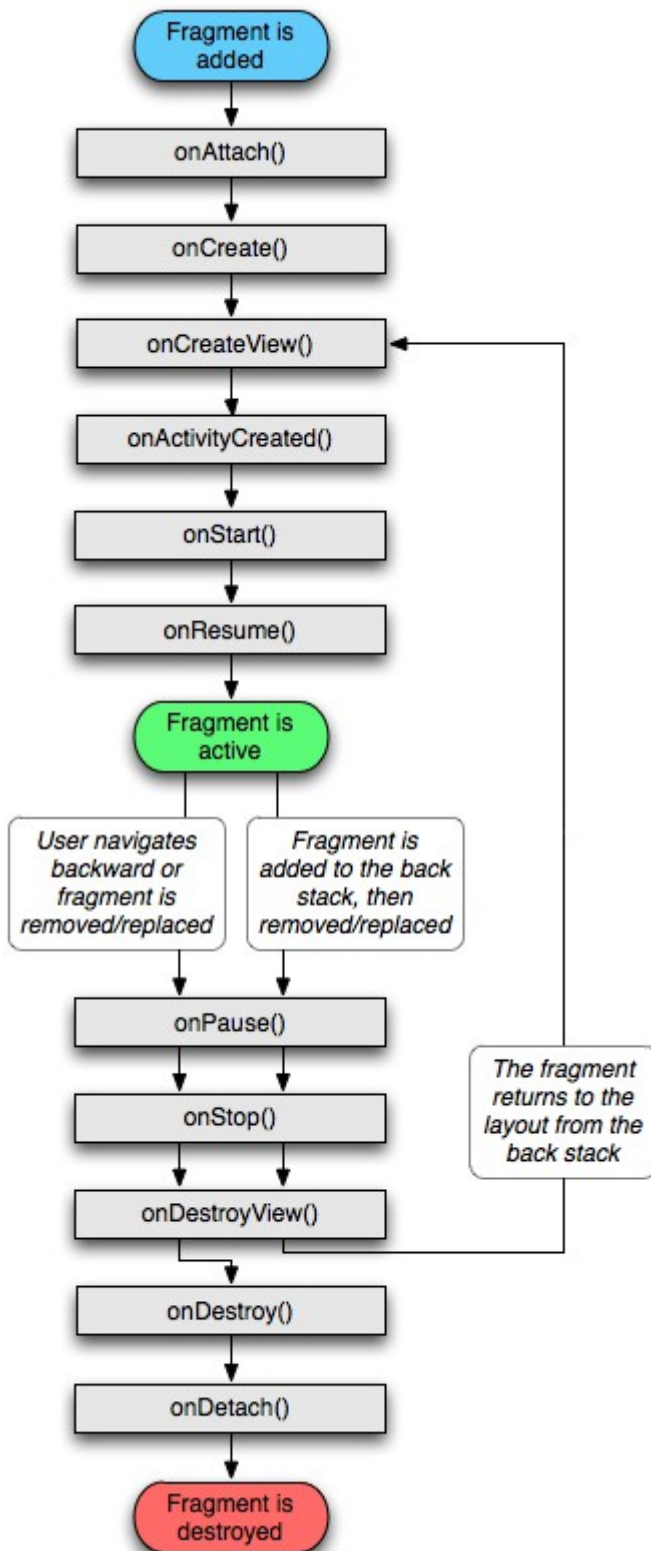


Figura 2. Ciclo de vida de un fragmento (mientras su actividad esté en ejecución).

También existen algunas subclases de las que se puede extender, en vez de la clase base Fragment:

DialogFragment

Nos muestra un diálogo flotante. Utilizar esta clase para crear un diálogo es una buena alternativa a utilizar los métodos de ayuda de los diálogos en la clase Activity, ya que se puede incorporar un fragmento diálogo en el back stack de los fragmentos gestionados por la actividad, permitiendo al usuario volver a un fragmento ya desechado.

ListFragment

Nos muestra una lista de items que son gestionados por un adaptador (como un SimpleCursorAdapter), similar a ListActivity. Nos suministra diferentes métodos para gestionar una lista de vistas, como el método callback onItemClick() para manejar eventos de click.

PreferenceFragment

Nos muestra una jerarquía de objetos Preference como una lista, similar a PreferenceActivity. Esto es útil cuando se crea una actividad "settings" para la aplicación.

Añadir un fragmento a una interfaz de usuario

Un fragmento es normalmente utilizado como parte de una actividad de IU y aporta su propio layout a la actividad.

Para aportar un layout al fragmento, se debe implementar el método callback onCreateView(), al que llama el sistema Android cuando es el momento de que el fragmento dibuje su layout. La implementación de este método debe devolver un View que es la raíz del layout del fragmento.

Nota: Si el fragmento es una subclase de ListFragment, la implementación por defecto devuelve un ListView desde onCreateView(), por lo que no necesita implementarlo.

Para devolver un layout desde onCreateView(), se puede inyectar desde un recurso del layout definido en el XML. Para ayudar con esto, onCreateView() suministra un objeto LayoutInflater.

Como ejemplo, tenemos una subclase de Fragment que carga un layout desde el archivo example_fragment.xml:

```
public static class ExampleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.example_fragment, container, false);
    }
}
```

El parámetro del container pasado al método onCreateView() es el padre ViewGroup (del layout de la actividad) en el que el layout del fragmento será insertado. El parámetro savedInstanceState es un Bundle que suministra datos sobre las instancias previas del fragmento, si el fragmento está siendo reanudado. Como reestablecer el estado es estudiado en mayor profundidad en

la sección sobre Gestionar el ciclo de vida del fragmento).

El método `inflate()` lleva tres argumentos:

- El ID del recurso del layout que se quiere inyectar.
- El `ViewGroup` que va a ser el padre del layout inyectado. Pasar el container es importante para que el sistema pueda aplicar los parámetros del layout al view raíz del layout afectado, especificado por el view padre en el que va.
- Debería existir un boolean indicando si el layout afectado se debería adjuntar al `ViewGroup` (segundo parámetro) durante la inyección. (En este caso, es `false` ya que el sistema ya está insertando el layout afectado en el container— si se pasa `true`, se crearía un view group redundante en el layout final.)

Ya hemos visto como crear un fragmento que suministra un layout. A continuación, se necesita añadir el fragmento a la actividad.

Añadir un fragmento a la actividad

Normalmente, un fragmento aporta una porción de IU a la actividad host, la cual está insertada como parte de la jerarquía global de la actividad. Existen dos maneras de añadir un fragmento al layout de la actividad:

- Declarar el fragmento dentro del archivo del layout de la actividad.

En este caso, se puede especificar propiedades del layout para el fragmento como si fuera un view. Como ejemplo, tenemos el archivo de un layout para una actividad con dos fragmentos:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.news.ArticleListFragment"
        android:id="@+id/list"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
    <fragment android:name="com.example.news.ArticleReaderFragment"
        android:id="@+id/viewer"
        android:layout_weight="2"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
</LinearLayout>
```

El atributo `android:name` en el `<fragment>` especifica la clase `Fragment` a instanciar en el layout.

Cuando el sistema crea este layout de la actividad, instancia cada fragmento especificado en el layout y llama al método `onCreateView()` para cada uno, para recuperar cada layout del fragmento. El sistema inserta el método `View` devuelto por el fragmento directamente en el lugar del elemento

<fragment>.

- Nota: Cada fragmento necesita un identificador único que el sistema puede usar para restablecer el fragmento si la actividad es reanudada (y que se puede utilizar para capturar el fragmento para realizar transacciones, como por ejemplo su eliminación). Existen tres maneras de suministrar un ID para un fragmento:

- Suministrar el atributo android:id con un ID único.
- Suministrar el atributo android:tag con un string único.
- Si no se aporta ninguno de estos dos, el sistema utiliza el ID del container view.
- También se puede, añadir el fragmento programáticamente a un ViewGroup existente.

En cualquier momento mientras se esté ejecutando la actividad, se pueden añadir fragmentos al layout de la actividad. Solamente se necesita especificar un ViewGroup en el que colocar el fragmento.

Para llevar a cabo transacciones del fragmento en la actividad (como añadir, eliminar, o sustituir un fragmento), se debe utilizar APIs de FragmentTransaction. Se puede conseguir una instancia de FragmentTransaction de la Activity de esta manera:

```
FragmentManager fragmentManager = getFragmentManager()  
FragmentTransaction fragmentTransaction =  
fragmentManager.beginTransaction();
```

Se puede añadir un fragmento utilizando el método add(), especificando el fragmento a añadir y el view en el que insertarlo. Por ejemplo:

```
ExampleFragment fragment = new ExampleFragment();  
fragmentTransaction.add(R.id.fragment_container, fragment);  
fragmentTransaction.commit();
```

El primero argumento que se pasa al método add() es el ViewGroup en el que el fragmento se tiene que insertar, especificado por el recurso ID, y el segundo parámetro es el fragmento a añadir.

Una vez que se han hecho los cambios con FragmentTransaction, se debe llamar al método commit() para que los cambios se efectuen.

Añadir un fragmento sin una IU

Los ejemplos anteriormente explicados nos muestran como añadir un fragmento a la actividad para suministrar una IU. Sin embargo, también se puede utilizar un fragmento para suministrar un comportamiento en segundo plano para la actividad sin una IU adicional.

Para añadir un fragmento sin una IU, hay que añadir el fragmento desde la actividad utilizando el método add(Fragment, String) (aportando un string único "tag" para el fragmento, en vez de un ID view). Esto añade el fragmento, pero, dado que no está asociado con un view en el layout de la actividad, no recibe una llamada al método onCreateView(), por lo que no se tiene que implementar ese método.

Suministrando un string "tag" para el fragmento no es estrictamente para los fragmentos no-IU—también se puede suministrar string "tags" a fragmentos

que si que tienen una IU—pero si el fragmento no tiene una IU, entonces el string "tag" es la única manera de identificarlo. Si se quiere conseguir el fragmento desde la actividad en algún momento, se necesita utilizar el método `findFragmentByTag()`.

Gestionar fragmentos

Para gestionar los fragmentos de la actividad, se necesita utilizar el `FragmentManager`. Para conseguirlo, hay que llamar al método `getFragmentManager()` de la actividad.

Algunas de las cosas que se puede hacer con el `FragmentManager` son:

- Conseguir fragmentos que existen en la actividad, a través de los métodos `findFragmentById()` (para fragmentos que suministran una IU en el layout de la actividad) o `findFragmentByTag()` para fragmentos que suministran o no una IU.
- Sacar fragmentos del back stack, con `popBackStack()`, simulando un comando BACK del usuario.
- Registrar un listener para cambios en el back stack, con el método `addOnBackStackChangeListener()`.

Como hemos visto en la sección anterior, también se puede utilizar `FragmentManager` para abrir un `FragmentTransaction`, que permite realizar transacciones, como añadir o eliminar fragmentos.

Realizar transacciones en los fragmentos

Una ventaja de usar fragmentos en la actividad es la habilidad de añadir, eliminar, sustituir y poder realizar otras acciones con ellos, en respuesta a la interacción del usuario. Cada conjunto de cambios que se hacen a la actividad se llama "transacción" y se puede hacer utilizando APIs en el `FragmentTransaction`. También se puede guardar cada transacción en un back pack gestionado por una actividad, permitiendo al usuario navegar hacia atrás a través de los cambios del fragmento (similar a cuando se navega hacia atrás a través de las actividades).

Se puede conseguir una instancia de `FragmentTransaction` desde el `FragmentManager` de esta manera:

```
FragmentManager fragmentManager = getFragmentManager();  
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
```

Cada transacción es un conjunto de cambios que se quiere realizar al mismo tiempo. Se pueden llevar a cabo todos los cambios para una transacción utilizando métodos como `add()`, `remove()` y `replace()`. Para aplicar la transacción a la actividad, se debe llamar al método `commit()`.

Antes de llamar al método `commit()`, también se puede querer llamar al método `addToBackStack()`, para añadir la transacción al back stack de las

transacciones de los fragmentos. Este back stack está gestionado por la actividad y permite al usuario retornar al estado previo del fragmento, haciendo click en la tecla BACK.

Como ejemplo, aquí podemos ver como se reemplaza un fragmento con otro, preservando el estado anterior en el back stack:

```
// Crear un nuevo fragmento y transacción
Fragment newFragment = new ExampleFragment();
FragmentManager transaction = getFragmentManager().beginTransaction();
// Reemplazar lo que esté en el fragment_container view con este fragmento,
// y añadir transacción al back stack
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);
//commit la transacción
transaction.commit();
```

En este ejemplo, newFragment reemplaza cualquier fragmento que esté en el container del layout (si hubiera) identificado con el ID R.id.fragment_container. Llamando al método addToBackStack(), la transacción reemplazada se guarda en el back stack para que el usuario pueda revertir la transacción y traer de vuelta el fragmento anterior apretando la tecla BACK.

Si se añaden múltiples cambios a la transacción (como otro add() o remove()) y llamar al método addToBackStack(), entonces todos los cambios realizados antes de la llamada al commit() son añadidos al back stack como una sola transacción y la tecla BACK lo invertirá de vuelta.

El orden en el que se añaden los cambios a un FragmentTransaction no importa, excepto si :

- Se debe llamar al commit() en ultimo lugar
- Si está añadiendo varios fragmentos al mismo container, el orden en el que se añade determina el orden en el que aparece en la vista de la jerarquía

Si no se llama al addToBackStack() cuando se realiza una transacción que elimina un fragmento, entonces ese fragmento es destruido cuando la transacción es comiteada y el usuario no puede volver a ella. Sin embargo, si llamas al addToBackStack() cuando se elimina un fragmento, entonces el fragmento es parado y será reanudado si el usuario vuelve a ella.

Llamando a commit() no se realiza la transacción inmediatamente. Se programa para que se ejecute en el hilo de la IU de la actividad (el hilo "main") tan pronto como el hilo lo pueda hacer. Si es necesario se puede llamar a executePendingTransactions() desde el thread del IU para que se ejecuten inmediatamente las transacciones presentadas en el commit(). Esto solo es necesario si la transacción es una dependencia para trabajos realizados en otros threads.

Advertencia: Se puede comitear una transacción utilizando commit() sólo si es anterior a la actividad y guardando su estado (cuando el usuario deje la actividad). Si se intenta comitear con posterioridad a este punto, se lanzará una excepción. Esto se produce porque se puede perder el estado después del commit si la actividad necesita ser reanudada. Para situaciones en las que no importa perder el commit, se utiliza commitAllowingStateLoss().

Comunicación con una actividad

A pesar de que un Fragment está implementado como un objeto que es independiente de una Activity y se puede utilizar dentro de varias actividades, una determinada instancia de un fragmento está directamente unida a la actividad que la contiene.

Específicamente, el fragmento puede acceder a la instancia de la Activity con `getActivity()` y puede realizar fácilmente tareas como encontrar un view en el layout de la actividad:

```
View listView = getActivity().findViewById(R.id.list);
```

Asimismo, la actividad puede llamar a métodos del fragmento consiguiendo una referencia al Fragment con `FragmentManager`, utilizando `findFragmentById()` o `findFragmentByTag()`. Por ejemplo:

```
ExampleFragment fragment = (ExampleFragment)
getFragmentManager().findFragmentById(R.id.example_fragment);
```

Crear eventos callback a la actividad

En algunos casos, se puede necesitar un fragmento para compartir eventos con la actividad. Una buena manera de hacerlo es definir una interfaz callback dentro del fragmento y hacer que la actividad host la implemente. Cuando la actividad recibe un callback a través de la interfaz, puede compartir la información, cuando sea necesario con otros fragmentos en el layout.

Por ejemplo, si una aplicación sobre noticias tiene dos fragmentos en la actividad—uno para mostrar una lista de artículos (fragmento A) y otro para mostrar un artículo (fragmento B)—el fragmento A debe notificar a la actividad cuando un item de la lista es seleccionado para que le pueda notificar al fragmento B que muestre el artículo. En este caso, la interfaz `OnArticleSelectedListener` es declarada dentro del fragmento A:

```
public static class FragmentA extends ListFragment {    ...
    // Container Activity must implement this interface

    public interface OnArticleSelectedListener {

        public void onArticleSelected(Uri articleUri);
    }

    ...
}
```

En ese momento la actividad que contiene al fragmento implementa la interfaz `OnArticleSelectedListener` y sobrescribe `onArticleSelected()` para notificar al fragmento B sobre el evento desde el fragmento A. Para asegurarse de que la actividad principal implementa esta interfaz, el método callback `onAttach()` del fragmento A (que el sistema llama cuando se añade el fragmento a la actividad) crea una instancia del `OnArticleSelectedListener` mediante el casteo de Activity que se pasa

en `onAttach()`:

```
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener mListener;

    ...
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        try {
            mListener = (OnArticleSelectedListener) activity;
        }
        catch (ClassCastException e) {
            throw new ClassCastException(activity.toString() + " must
implement OnArticleSelectedListener");
        }
        ...
    }
}
```

Si la actividad no ha implementado la interfaz, el fragmento tira una `ClassCastException`. Si se ha realizado con éxito, el componente `mListener` tiene una referencia a la implementación del método `OnArticleSelectedListener` de la actividad, para que el fragmento A pueda compartir eventos con la actividad mediante la llamada a métodos definidos por el `OnArticleSelectedListener` de la interfaz. Por ejemplo, si el fragmento A es una extensión de `ListFragment`, cada vez que el usuario hace click en una lista de items, el sistema llama al `onListItemClick()` en el fragmento, que llama a su vez al `onArticleSelected()` para compartir el evento con la actividad:

```
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener mListener;

    ...
    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        // Append the clicked item's row ID with the content provider Uri
        Uri noteUri = ContentUris.withAppendedId(ArticleColumns.CONTENT_URI,
id);

        // Send the event and Uri to the host activity
        mListener.onArticleSelected(noteUri);
    }
    ...
}
```

El parámetro `id` pasado al `onListItemClick()` es la fila ID del item elegido, que la actividad (u otro fragmento) utiliza para ir a buscar el artículo al `ContentProvider` de la aplicación.

Añadir items a la Barra de Acción

Los fragmentos pueden contribuir con items de menu al menu de Opciones de la actividad (y consecuentemente, a la barra de acción) mediante la implementación de `onCreateOptionsMenu()`. Para que este método reciba llamadas, se debe llamar al método `setHasOptionsMenu()` durante `onCreate()`, para indicar que el fragmento quiere añadir items al Menu de Opciones (en caso contrario, el fragmento no recibirá la llamada a `onCreateOptionsMenu()`).

Cualquier item que se añada al Menu de Opciones desde el fragmento está

unido al menu de items ya existente. El fragmento también recibe callbacks al `onOptionsItemSelected()` cuando un item del menu es seleccionado.

También se puede registrar una vista en el layout del fragmento para aportar un menu context llamando al `registerForContextMenu()`. Cuando el usuario abre el menu context, el fragmento recibe una llamada al `onCreateContextMenu()`. Cuando el usuario selecciona un item, el fragmento recibe una llamada al `onContextItemSelected()`.

Nota: A pesar de que el fragmento recibe un callback item- seleccionado por cada item del menu añadido, la actividad va a ser la primera en recibir el callback determinado cuando el usuario seleccione un item del menu. Si la implementación de la actividad del callback item- seleccionado, no gestiona el item seleccionado, entonces el evento se pasa al callback del fragmento. Esto es así con los Menu de Opciones y los Menu context.

Gestionar el ciclo de vida del fragmento.

Gestionar el ciclo de vida de un fragmento es como gestionar el ciclo de vida de una actividad. Como una actividad, el fragmento puede existir en tres estados:

Reanudado

El fragmento es visible en la actividad que se está ejecutando.

En pausa

Otra actividad está en primer plano y tiene el focus, pero la actividad en la que este fragmento vive está todavía visible (la actividad en primer plano es parcialmente transparente y no cubre la pantalla al completo).

Parado

El fragmento no es visible. O bien la actividad principal se ha parado o el fragmento se ha eliminado de la actividad, añadiéndose al back stack. Un fragmento parado está todavía vivo (toda la información del componente y su estado está retenido en el sistema). Sin embargo, no es ya visible para el usuario y será terminado si la actividad es terminada.

Tal y como sucede en una actividad, se puede retener el estado de un fragmento utilizando un Bundle, en caso de que el proceso de la actividad se termine y se necesita restaurar el estado del fragmento cuando la actividad se vuelva a crear. Se puede guardar el estado durante el callback `onSaveInstanceState()` y restaurarlo durante el `onCreate()`, el `onCreateView()`, o el `onActivityCreated()`.

La diferencia más importante entre el ciclo de vida de una actividad y un fragmento es como se almacenan en sus respectivos back stack. Una actividad se coloca en un back stack de actividades que es gestionada por el sistema cuando se para, por defecto (para que el usuario pueda navegar hacia atrás con la tecla BACK, como ya hemos visto en [Tareas y Back Stack](#)). Sin embargo, un fragmento es colocado en un back stack gestionado por la actividad principal sólo cuando se hace una petición explícitamente para que la instancia se guarde mediante la llamada a `addToBackStack()` durante la transacción que elimina el fragmento.

Por lo demás, gestionar el ciclo de vida de un fragmento es muy parecido a gestionar el ciclo de vida de una actividad. Por ello, se utilizan las mismas

prácticas para gestionar el ciclo de vida de una actividad como para los fragmentos.

Coordinación con el ciclo de vida de la actividad

El ciclo de vida de la actividad en el que vive el fragmento afecta directamente el ciclo de vida del fragmento, tanto que cada callback del ciclo de vida de la actividad resulta en una llamada similar por cada fragmento. Por ejemplo, cuando la actividad recibe `onPause()`, cada fragmento en la actividad recibe `onPause()`.

Los fragmentos aparte tienen unos callbacks de ciclo de vida extras, que gestionan una interacción única con la actividad para realizar acciones tales como construir y eliminar las IU de los fragmentos. Estos métodos callback adicionales son:

`onAttach()`

Llamado cuando el fragmento se ha asociado con la actividad (la Activity se pasa aquí).

`onCreateView()`

Llamado para crear la jerarquía de la vista asociada al fragmento.

`onActivityCreated()`

Llamado cuando el método `onCreate()` de la actividad retorna.

`onDestroyView()`

Llamado cuando la jerarquía de la vista asociada al fragmento se está eliminando.

`onDetach()`

Llamado cuando el fragmento está siendo disociado de la actividad.

El flujo del ciclo de vida del fragmento, y como está afectado por su actividad host, se ilustran en la figura 1. En esta figura, se puede ver como cada estado sucesivo de la actividad determina que métodos callback puede recibir un fragmento. Por ejemplo, cuando una actividad recibe el método callback `onCreate()`, un fragmento de la actividad sólo recibe al callback `onActivityCreated()`.

Ejemplo de Actividad con dos fragmentos

Aquí tenemos un ejemplo que recopila lo que hemos visto hasta ahora; una actividad que utiliza dos fragmentos para crear un layout de dos paneles.

La actividad que vemos a continuación incluye un fragmento que muestra una lista de títulos teatrales de Shakespeare y otro para mostrar un resumen de la obra cuando es seleccionado en una lista. También muestra las diferentes configuraciones de los fragmentos, basados en la configuración de la pantalla.

La principal actividad aplica un layout de la manera usual, en el método `onCreate()`:

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

El layout es fragment_layout.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment class="com.example.android.apis.app.FragmentLayout$TitlesFragment"
        android:id="@+id/titles" android:layout_weight="1"
        android:layout_width="0px" android:layout_height="match_parent" />
    <FrameLayout android:id="@+id/details" android:layout_weight="1"
        android:layout_width="0px" android:layout_height="match_parent"
        android:background="?android:attr/detailsElementBackground" />
</LinearLayout>
```

Utilizando este layout, el sistema instancia TitlesFragment (que nos da una lista de las obras) cuando la actividad carga el layout, mientras que el FrameLayout (donde va el fragmento que muestra el resumen de la obra) consume espacio en el lado derecho de la pantalla, pero está vacío al principio. No es hasta que el usuario selecciona un ítem de la lista que un fragmento es situado en el FrameLayout.

Sin embargo, no todas las configuraciones de pantalla son suficientemente anchas para mostrar uno al lado del otro las listas de las obras y el resumen. Esto hace que el layout que acabamos de ver es utilizado solamente para la configuración de vista apaisada (paisaje), guardándola en el res/layout-land/fragment_layout.xml.

Cuando, la pantalla está en orientación retrato, el sistema adopta el siguiente layout, que se guarda en res/layout/fragment_layout.xml:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment class="com.example.android.apis.app.FragmentLayout$TitlesFragment"
        android:id="@+id/titles"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</FrameLayout>
```

Este layout incluye solamente el TitlesFragment. Esto significa que, cuando el dispositivo está en orientación retrato, sólo se puede ver la lista de las obras. Esto hace que cuando el usuario hace click en una lista de ítems en esta configuración, la aplicación arrancará una nueva actividad para mostrar el resumen, en vez de cargar un segundo fragmento.

A continuación, podemos ver como se hace en las clases fragmento. Primero va TitlesFragment, que muestra la lista de obras de Shakespeare. Este fragmento extiende de ListFragment y se apoya en el para gestionar el trabajo de mostrar la lista.

Obsérvese que hay dos posibles comportamientos cuando el usuario cliquea

una lista de items: dependiendo de cual de los dos layouts esté activo, puede o bien crear y mostrar un nuevo fragmento para mostrar los detalles en la misma actividad (añadiendo el fragmento al FrameLayout), o bien empezar una nueva actividad (donde el fragmento se pueda mostrar).

```
public static class TitlesFragment extends ListFragment {
    boolean mDualPane;
    int mCurCheckPosition = 0;
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        // list con el static array de títulos
        setListAdapter(new ArrayAdapter<String>(getActivity(),
            android.R.layout.simple_list_item_activated_1, Shakespeare.TITLES));
        // Chequear si hay un marco para meter los detalles del fragmento
        // directamente en la IU contenedora.
        View detailsFrame = getActivity().findViewById(R.id.details);
        mDualPane = detailsFrame != null && detailsFrame.getVisibility() ==
View.VISIBLE;
        if (savedInstanceState != null) {
            // Restaurar ultimo estado para la posición chequeada.
            mCurCheckPosition = savedInstanceState.getInt("curChoice", 0);
        }
        if (mDualPane) {
            // En el modo dual-pane, la lista destaca el item seleccionado.
            getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
            // chequear que la IU en el estado correcto.
            showDetails(mCurCheckPosition);
        }
    }
    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("curChoice", mCurCheckPosition);
    }
    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        showDetails(position);
    }
    /**
     * Función de ayuda para mostrar los detalles de un item seleccionado, bien por
     * mostrar un fragmento en su lugar en la actual IU, o bien empezando
     * una nueva actividad en el que se muestre.
     */
    void showDetails(int index) {
        mCurCheckPosition = index;
        if (mDualPane) {
            // Podemos mostrar todo en su sitio con fragmentos, así que modificamos
            // la lista para destacar el item seleccionado y mostrar los datos.
            getListView().setItemChecked(index, true);
            // Chequear que fragmento es el que se está mostrando, reemplazar si fuera
            // necesario.
            DetailsFragment details =
            (DetailsFragment) getFragmentManager().findFragmentById(R.id.details);
            if (details == null || details.getShownIndex() != index) {
                // Crear nuevo fragmento para mostrar esta selección
                details = DetailsFragment.newInstance(index);
            }
            // Ejecutar una transacción, reemplazando cualquier fragmento ya existente
            // con el que está dentro del marco
            FragmentTransaction ft = getFragmentManager().beginTransaction();
            ft.replace(R.id.details, details);
            ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
            ft.commit();
        }
    }
}
```

```

}
} else {
// Si no hay que lanzar una nueva actividad para mostrar
// el diálogo del fragmento con el texto seleccionado.
    Intent intent = new Intent();
    intent.setClass(getActivity(), DetailsActivity.class);
    intent.putExtra("index", index);
    startActivity(intent);
}
}
}

```

El segundo fragmento, DetailsFragment muestra el resumen de la obra para el ítem seleccionado de la lista de TitlesFragment:

```

public static class DetailsFragment extends Fragment {
/**
 * Crear una nueva instancia de DetailsFragment, inicializada para
 * mostrar el texto en 'index'.
 */
    public static DetailsFragment newInstance(int index) {
        DetailsFragment f = new DetailsFragment();
        // Aportar el index input como argumento.
        Bundle args = new Bundle();
        args.putInt("index", index);
        f.setArguments(args);
        return f;
    }

    public int getShownIndex() {
        return getArguments().getInt("index", 0);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        if (container == null) {
            // Tenemos diferentes layouts, y en uno de ellos este marco
            // que contiene el fragmento no existe. El fragmento
            // todavía puede ser creado desde su estado guardado, pero no
            // hay razón para intentar crear su vista jerárquica ya que
            // no se mostrará. Notese que esto no es necesario--- podemos
            // simplemente ejecutar el código que está a continuación, donde se crea y
            // devuelve
            // la vista jerárquica; aunque no se usaría nunca.
            return null;
        }

        ScrollView scroller = new ScrollView(getActivity());
        TextView text = new TextView(getActivity());
        int padding =
            (int)TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP,4,
            getActivity().getResources().getDisplayMetrics());
        text.setPadding(padding, padding, padding, padding);
        scroller.addView(text);
        text.setText(Shakespeare.DIALOGUE[getShownIndex()]);
        return scroller;
    }
}

```

En la clase TitlesFragment se vió, que, si el usuario cliquea un ítem de la lista y el actual layout no incluye la vista de R.id.details (que es de donde pertenece

el DetailsFragment), entonces la aplicación ejecuta la actividad DetailsActivity para mostrar el contenido del item.

A continuación se ve el DetailsActivity, que simplemente inserta el DetailsFragment para mostrar el resumen de la obra seleccionado cuando la pantalla tiene la orientación paisaje:

```
public static class DetailsActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getResources().getConfiguration().orientation==
        Configuration.ORIENTATION_LANDSCAPE) {
            // si la pantalla está en modo paisaje, podemos mostrar
            // el diálogo in-line con la lista por lo que no necesitamos esta actividad.
            finish();
            return;
        }

        if (savedInstanceState == null) {
            // Durante la instalación inicial, conectar los detalles del fragmento.
            DetailsFragment details = new DetailsFragment();
            details.setArguments(getIntent().getExtras());
            getFragmentManager().beginTransaction().add(android.R.id.content,
            details).commit();
        }
    }
}
```

Obsérvese que esta actividad se termina a si misma si la configuración es paisaje, para que la actividad principal pueda tomar el mando y muestre el DetailsFragment junto con el TitlesFragment. Esto puede ocurrir si el usuario lanza el DetailsActivity durante la orientación retrato, pero luego rota a paisaje (lo que relanza la actividad actual).

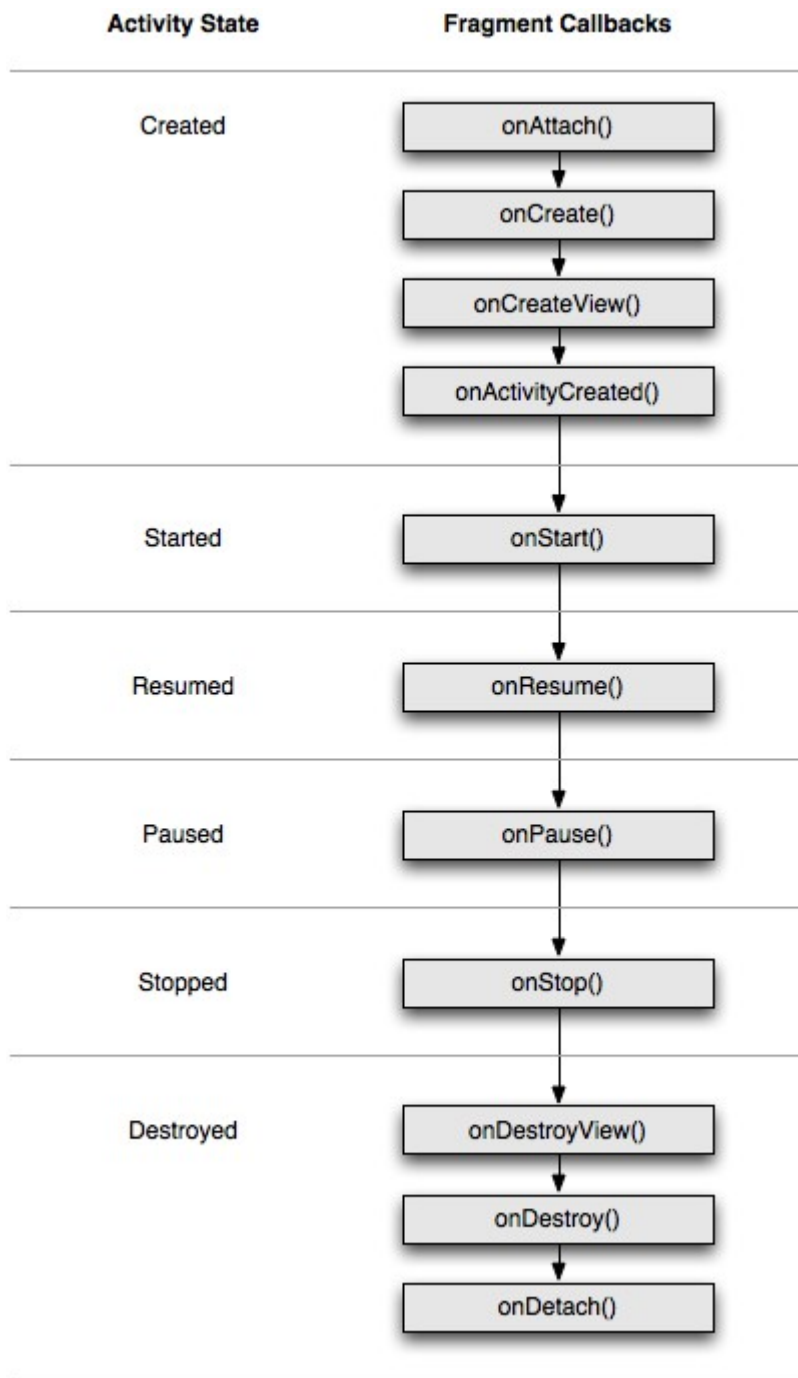


Figura 1. Como afecta el ciclo de vida de la actividad al ciclo de vida del fragmento.

Una vez que la actividad llega al estado reanudado (resumed), se pueden añadir o eliminar los fragmentos a la actividad. De esta manera, el ciclo de vida de un fragmento puede cambiar independientemente solo cuando la actividad se encuentra en el estado reanudado.

Sin embargo, cuando la actividad deja el estado de reanudado, el fragmento se mueve por su ciclo de vida empujado por la actividad.

Loaders

Los loaders fueron introducidos en Android 3.0, y facilitan la carga de datos de manera asincrónica en una actividad o fragmento. Los loaders tienen las siguientes características:

- Están disponibles para cada Activity y cada Fragment.
- Permiten carga de datos asincrónica.
- Monitorizan la fuente de datos y entregan los nuevos resultados cuando cambia el contenido.
- Reconectan automáticamente al cursor del último loader cuando se crean de nuevo después de un cambio en la configuración. Así, no necesitan hacer un re-query de los datos.

Ejecutar y Reiniciar un Loader

Esta sección describe como utilizar loaders en una aplicación Android. Una aplicación que utiliza loaders incluye lo siguiente:

- Una Activity or Fragment.
- Una instancia de LoaderManager.
- Un CursorLoader para cargar los datos respaldado por un ContentProvider. De manera alternativa, se puede implementar una subclase propia de Loader o de AsyncTaskLoader para cargar datos desde otras fuentes.
- Una implementación para LoaderManager.LoaderCallbacks. Aquí es donde se crea los nuevos loaders y donde se gestionan las referencias al los loaders ya existentes.
- Una manera de mostrar los datos de los loaders, como un SimpleCursorAdapter.
- Una fuente de datos, como un ContentProvider, cuando se utiliza un CursorLoader.

Ejecutar un Loader

El LoaderManager gestiona una o más instancias de Loader dentro de una Activity o de un Fragment. Sólo hay un LoaderManager por actividad o fragmento.

Se inicializa un Loader dentro del método onCreate() de la actividad, o dentro del método onActivityCreated() de un fragmento. Esto se hace de la siguiente manera:

```
// Preparar el loader. O bien reconectar con uno que ya existe,  
// o lanzar uno nuevo.  
getLoaderManager().initLoader(0, null, this);
```

El método initLoader() lleva los siguientes parámetros:

- Un ID único que identifica al loader. En este ejemplo el ID es 0.
- Argumentos opcionales que suministran al loader en construcción (null en este ejemplo).
- Una implementación de LoaderManager.LoaderCallbacks, que es llamado por el LoaderManager para reportar los eventos del loader. En este ejemplo, la

clase local implementa la interfaz LoaderManager.LoaderCallbacks y se pasa una referencia a si misma this.

La llamada al método initLoader() asegura la inicialización y la activación de un loader. Tiene dos posibles resultados:

- Si el loader especificado por el ID ya existe, se reutilizará el último loader creado.
- Si el loader especificado por el ID no existe, el método initLoader() desencadena la llamada al método onCreateLoader() del LoaderManager.LoaderCallbacks. Aquí es donde se implementa el código para instanciar y retornar un nuevo loader.

En cualquiera de los dos casos, la implementación de LoaderManager.LoaderCallbacks está asociada con el loader, y será llamada cuando cambie el estado de los loaders. Si cuando se produce la llamada, el estado está en started, y el loader pedido ya existe y ha generado los datos, el sistema llamará al método onLoadFinished() (en el método initLoader().)

Veáse que el método initLoader() retorna el Loader creado, aunque no se necesita capturar una referencia a él. El LoaderManager gestiona la vida del loader automáticamente. El LoaderManager arranca y para la carga siempre que sea necesario y mantiene el estado del loader así como su contenido asociado. Esto implica, que raramente se interactúa directamente con los loaders. Cuando suceden eventos particulares se debe utilizar los métodos del LoaderManager.LoaderCallbacks para intervenir en el proceso de carga. Para más detalle sobre este tema, ver Utilizar los métodos callback de LoaderManager.

Reiniciar un loader

Como acabamos de ver, cuando se utiliza el método initLoader(), utiliza un loader ya existente con el ID especificado, si es que existe. Si no existe, lo crea. Algunas veces se quieren eliminar los datos antiguos y volver a empezar.

Para eliminar los datos antiguos, se utiliza el método restartLoader(). Como ejemplo, tenemos la implementación del SearchView.OnQueryTextListener que reinicia el loader cuando el usuario cambia la consulta. Hay que reiniciar el loader para que pueda utilizar el filtro de búsqueda revisado para hacer una nueva query:

```
public boolean onQueryTextChanged(String newText) {
    // Llamado cuando el texto en la búsqueda en la barra de acción cambia.
    Actualiza
    // el filtro de búsqueda, y reinicia el loader para hacer una nueva consulta
    // con este filtro.
    mCurFilter = !TextUtils.isEmpty(newText) ? newText : null;
    getLoaderManager().restartLoader(0, null, this);
    return true;
}
```

Utilizar los métodos callback de LoaderManager.

LoaderManager.LoaderCallbacks es una interfaz callback que permite al cliente interactuar con el LoaderManager.

Los Loaders y en particular el CursorLoader, deben retener los datos después de haberse parado. Esto permite a las aplicaciones guardar los datos a través de los métodos onStop() y onStart() de la actividad y de los fragmentos, para que cuando el usuario retorna a una aplicación, no tenga que esperar para recargar los datos. Se utilizan los métodos LoaderManager.LoaderCallbacks para saber cuando crear un nuevo loader y para hacerle saber a la aplicación cuando es necesario dejar de utilizar un loader de datos.

Los LoaderManager.LoaderCallbacks incluyen estos métodos:

- onCreateLoader() — Instancia y retorna un nuevo Loader para un determinado ID.
- onLoadFinished() — Es llamado cuando un loader ya creado termina su carga.
- onLoaderReset() — Llamado cuando un loader ya creado está siendo reiniciado (y por tanto sus datos no están disponibles)

Estos métodos son descritos con más detalle en las siguientes secciones.

onCreateLoader

Cuando se intenta acceder a un loader (por ejemplo, a través de un initLoader()), se comprueba si el loader especificado por el ID existe o no. Si no existe, llama al método onCreateLoader() del LoaderManager.LoaderCallbacks. Aquí es donde se crea un nuevo loader. Normalmente sería un CursorLoader, pero se puede implementar una subclase Loader propia.

En el ejemplo, el método callback onCreateLoader() crea un CursorLoader. Se debe construir el CursorLoader utilizando su constructor, para lo que se necesita la información utilizada para realizar una consulta al ContentProvider. En concreto, se necesita:

- uri — El URI a recuperar por el content.
- projection — Una lista de las columnas a devolver. Si se pasa null devolverá todas las columnas, lo que es ineficiente.
- selection — Un filtro que declara que filas hay que devolver, siguiendo la cláusula SQL WHERE (excluyendo el WHERE). Si se pasa null devolverá todas las filas especificadas para el URI.
- selectionArgs — Se pueden incluir ?s en la selección, que serán reemplazados por los valores del selectionArgs, en orden de aparición en la selección. Los valores serán unidos como Strings.
- sortOrder — Cómo ordenar las filas, siguiendo el formato de la cláusula SQL ORDER BY (excluyendo el ORDER BY). Si se pasa null se usará el sort order por defecto, que puede ser sin orden.

Por ejemplo:

```
// Si null, este es el filtro que el usuario ha aportado.  
String mCurFilter;
```

```

...
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    // Es llamado cuando se necesita crear un nuevo Loader. A Este
    // ejemplo sólo tiene un Loader, por lo que no nos preocupamos por el ID.
    // Primero, hay que coger el URI base a utilizar dependiendo de si tenemos
    // algún filtro o no
    Uri baseUri;
    if (mCurFilter != null) {
        if (mCurFilter != null) {
            baseUri = Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI,
                Uri.encode(mCurFilter));
        } else {
            baseUri = Contacts.CONTENT_URI;
        }
        // Ahora se crea y se retorna un CursorLoader que se ocupará
        // de crear un Cursor para los datos mostrados.
        String select = "(" + Contacts.DISPLAY_NAME + " NOTNULL) AND ("
            + Contacts.HAS_PHONE_NUMBER + "=1) AND ("
            + Contacts.DISPLAY_NAME + " != ' ' )";
        return new CursorLoader(getActivity(), baseUri,
            CONTACTS_SUMMARY_PROJECTION, select, null,
            Contacts.DISPLAY_NAME + " COLLATE LOCALIZED ASC");
    }
}

```

onLoadFinished

Este método es llamado cuando un loader creado previamente ha terminado su carga. La llamada a este método se realiza antes de que se produzca la liberación de los últimos datos que nos trae el loader. En este punto se debe eliminar cualquier uso de los datos antiguos, pero no se debe hacer una liberación propia de los datos ya que el loader es el encargado de hacerlo.

El loader liberará los datos cuando la aplicación no los esté utilizando. Por ejemplo, si los datos son un cursor de un CursorLoader, no se debería llamar al método close(). Si el cursor se localiza en un CursorAdapter, se debería usar el método swapCursor() para que no se cierre el antiguo Cursor. Por ejemplo:

```

// Este es el Adapter siendo utilizado para mostrar la lista de datos.
SimpleCursorAdapter mAdapter;
...

public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    // Cambiar por el nuevo cursor. (El framework se encargará de
    // cerrar el antiguo cursor en el retorno.)
    mAdapter.swapCursor(data);
}

```

onLoaderReset

Este método es llamado cuando un loader que ha sido previamente creado se reinicia, haciendo que los datos no estén disponibles. Este callback permite que se sepa cuando los datos van a ser liberados para que se puedan eliminar sus referencias.

Esta implementación llama al método swapCursor() con null:

```

// Este es el Adapter utilizado para mostrar la lista de los datos.
SimpleCursorAdapter mAdapter;
...

public void onLoaderReset(Loader<Cursor> loader) {
    // Se llama cuando se vaya a cerrar el último Cursor suministrado al método

```

```
// onLoadFinished() definido arriba. Hay que estar seguros de que
// ya no se esté utilizando.
    mAdapter.swapCursor(null);
}
```

Ejemplo Loaders

Como ejemplo, a continuación tenemos la implementación completa de un Fragment que muestra el ListView que contiene los resultados de una consulta realizada a los contactos del content provider. Utiliza un CursorLoader para gestionar la consulta en el provider.

Para que un aplicación acceda a los contactos del usuario, el manifest debe incluir el permiso READ_CONTACTS.

```
public static class CursorLoaderListFragment extends ListFragment
    implements OnQueryTextListener, LoaderManager.LoaderCallbacks<Cursor> {
    // Este es el Adapter utilizado para mostrar la lista de datos.
    SimpleCursorAdapter mAdapter;

    // Si es non-null, este es el filtro que ha suministrado el
    usuario.
    String mCurFilter;

    @Override public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        // Dar texto a mostrar si no hay datos. En una aplicación
        // real vendrán de un recurso.
        setEmptyText("No phone numbers");

        // Existe un menu item para mostrar en la barra de acción.
        setHasOptionsMenu(true);

        // Crear un adapter vacio que será utilizado para mostrar los
        datos cargados.
        mAdapter = new SimpleCursorAdapter(getActivity(),
            android.R.layout.simple_list_item_2,
            null,
            new String[] { Contacts.DISPLAY_NAME,
                Contacts.CONTACT_STATUS },
            new int[] { android.R.id.text1,
                android.R.id.text2 }, 0);
        setListAdapter(mAdapter);

        // Preparar el loader; o bien se reconecta con uno ya
        existente,
        // o iniciar uno nuevo.
        getLoaderManager().initLoader(0, null, this);
    }

    @Override public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
        // Colocar un item de barra de acción para la búsqueda.
        MenuItem item = menu.add("Search");
        item.setIcon(android.R.drawable.ic_menu_search);
        item.setShowAsAction(MenuItem.SHOW_AS_ACTION_IF_ROOM);
        SearchView sv = new SearchView(getActivity());
        sv.setOnQueryTextListener(this);
    }
}
```

```

        item.setActionView(sv);
    }

    public boolean onQueryTextChanged(String newText) {
        // Se llama cuando el texto en la búsqueda de la barra de
        acción cambia.
        // Actualizar el filtro de búsqueda, y reiniciar el loader para
        //realizar una nueva consulta con este filtro.
        mCurFilter = !TextUtils.isEmpty(newText) ? newText : null;
        getLoaderManager().restartLoader(0, null, this);
        return true;
    }

    @Override public boolean onQueryTextSubmit(String query) {
        //
        return true;
    }

    @Override public void onListItemClick(ListView l, View v, int position,
long id) {
        // Insertar aquí comportamiento deseado.
        Log.i("FragmentComplexList", "Item clicked: " + id);
    }

    // Estos son las filas de Contactos que se devolverán.
    static final String[] CONTACTS_SUMMARY_PROJECTION = new
String[] {
        Contacts._ID,
        Contacts.DISPLAY_NAME,
        Contacts.CONTACT_STATUS,
        Contacts.CONTACT_PRESENCE,
        Contacts.PHOTO_ID,
        Contacts.LOOKUP_KEY,
    };

    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        // Es llamado cuando se necesita crear un nuevo Loader;
        // Este ejemplo sólo tiene un Loader, por lo que no hay que
preocuparse por el ID.
        // Primero, escoger el URI base a utilizar según si estamos
filtrando o no.
        Uri baseUri;
        if (mCurFilter != null) {
            baseUri = Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI,
Uri.encode(mCurFilter));
        } else {
            baseUri = Contacts.CONTENT_URI;
        }

        // Crear y retornar un CursorLoader que se encargará
        // de crear un Cursor para los datos mostrados.
        String select = "(" + Contacts.DISPLAY_NAME + " NOTNULL) AND
("
                                + Contacts.HAS_PHONE_NUMBER + "=1) AND
("
                                + Contacts.DISPLAY_NAME + " != ' ' )";
        return new CursorLoader(getActivity(), baseUri,
CONTACTS_SUMMARY_PROJECTION, select,
null,
                                Contacts.DISPLAY_NAME + " COLLATE
LOCALIZED ASC");
    }

    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        // Colocar el nuevo cursor. (El framework se encargará de

```

```

cerrar
        // el antiguo cursor en el retorno.)
        mAdapter.swapCursor(data);
    }

    public void onLoaderReset(Loader<Cursor> loader) {
        // Es llamado cuando el ultimo Cursor suministrado al
        onLoadFinished()
        // vaya a ser cerrado; Se necesita saber cuando
        // no se está utilizando.
        mAdapter.swapCursor(null);
    }
}

```

Resumen del API Loader

Existen muchas clases e interfaces que utilizan loaders en una aplicación. Se resumen en esta tabla:

Clases/Interfaz	Descripción
<u>LoaderManager</u>	<p>Una clase abstracta asociada con un <u>Activity</u> o un <u>Fragment</u> para gestionar uno o más instancias de <u>Loader</u>. Esto ayuda a una aplicación a gestionar operaciones de larga duración junto con los ciclos de vida de una <u>Activity</u> o de un <u>Fragment</u>; su uso más común es con un <u>CursorLoader</u>, sin embargo las aplicaciones son libres de escribir sus propios loaders para cargar otros tipos de datos.</p> <p>Sólo hay un <u>LoaderManager</u> por actividad o fragmento. Pero un <u>LoaderManager</u> puede tener varios loaders.</p>
<u>LoaderManager.LoaderCallbacks</u>	Es una interfaz callback para que el cliente pueda interactuar con el <u>LoaderManager</u> . Por ejemplo, se utiliza el método <u>callback.onCreateLoader()</u> para crear un nuevo loader.
<u>Loader</u>	Es una clase abstracta que realiza carga asincrónica de datos. Esta es la clase base para un loader. Normalmente se usaría el <u>CursorLoader</u> , pero también se puede implementar una clase propia. Mientras los loaders estén activos deben monitorizar la fuente de los datos y entregar nuevos resultados cuando el contenido cambie.
<u>AsyncTaskLoader</u>	Es un loader abstracto que suministra un <u>AsyncTask</u> para realizar el trabajo.
<u>CursorLoader</u>	Subclase de <u>AsyncTaskLoader</u> que lanza una consulta a <u>ContentResolver</u> y retorna un <u>Cursor</u> . Esta clase implementa el protocolo de <u>Loader</u> de la manera estandar para lanzar una consulta a los cursores. Lo hace sobre <u>AsyncTaskLoader</u> que realiza la consulta cursor en un hilo en segundo plano para que no bloquee la IU de la aplicación. Utilizar este loader es la mejor manera de cargar datos de manera asíncrona desde un <u>ContentProvider</u> . Se hace en vez de lanzar una consulta gestionada a través de un fragmento o de un API de una actividad.

Las clases e interfaces de la tabla anterior son los componentes esenciales que se utilizarán para implementar un loader en una aplicación. No se necesitan todas para cada loader que se cree, pero se necesita una referencia al LoaderManager para que inicialice un loader y una implementación de una clase Loader como por ejemplo el CursorLoader.

Servicios Android

Un Servicio es un componente de la aplicación que puede realizar operaciones en segundo plano de larga duración sin una interfaz de usuario. Otro componente de la aplicación puede empezar un servicio y continuará ejecutándose en un segundo plano aún si el usuario cambia de aplicación. Además, un componente se puede unir a un servicio para interactuar con él e incluso realizar comunicación inter-proceso (IPC).

Por ejemplo, un servicio puede gestionar en un segundo plano; transacciones de red, música, realizar archivos I/O, o interactuar con un content provider.

Un servicio puede estar en dos modos:

Iniciado

Un servicio está "iniciado" cuando un componente de una aplicación (como una actividad) lo inicia llamando a un método startService(). Una vez iniciado, un servicio puede ejecutarse en un segundo plano indefinidamente, incluso si se elimina el componente que lo inició. Normalmente, un servicio iniciado realiza una sola operación y no devuelve un resultado al que le llama. Por ejemplo, puede descargar o subir un archivo a través de la red. Al finalizar esta operación, el servicio se debería parar solo.

Vinculado (Bound)

Un servicio está "vinculado" cuando un componente de una aplicación se une a él mediante la llamada al método bindService(). Un servicio Bound ofrece una interfaz cliente-servidor que permite a los componentes interactuar con el servicio, mandar peticiones, obtener resultados, e incluso hacerlo a través de procesos con comunicación inter-proceso (IPC). Un servicio Bound se ejecuta solamente si otro componente de la aplicación está unido a él. Varios componentes se pueden unir al servicio al mismo tiempo, pero cuando todos se desvinculan; se elimina el servicio.

A pesar de que esta documentación habla sobre estos dos tipos de servicios por separado, el servicio puede trabajar de las dos maneras—puede iniciarse (para ejecutarse indefinidamente) y también puede permitir la unión. Esto va a depender de si se implementan un par de métodos callback: onStartCommand() para permitir que los componentes lo inicien, onBind() para permitir la unión.

Cualquier componente de la aplicación puede utilizar el servicio (incluso desde una aplicación separada), de la misma manera que cualquier componente puede utilizar una actividad— iniciándolo con un Intent . Esto ocurre tanto en el caso de que la aplicación sea iniciada, unida o ambas. Sin embargo, se puede declarar un servicio como privado en el archivo manifest y así bloquear el acceso desde otras aplicaciones. Sobre este tema se abre en más

profundidad en la sección sobre Declarar el servicio en el manifest.

Precaución: Un servicio se ejecuta en el hilo principal del proceso del host—el servicio no crea su propio hilo y no se ejecuta en un proceso separado (a no ser que se especifique lo contrario). Esto significa que, si el servicio va a realizar cualquier trabajo intensivo de CPU u operaciones bloqueantes (como playback del MP3 o trabajo de red), se debería crear un nuevo hilo dentro del servicio para realizar ese trabajo. Si se utiliza un hilo por separado, se reducirá el riesgo de errores de Aplicación No Responde (ANR) y el hilo principal de la aplicación puede seguir dedicado a la interacción del usuario con las actividades.

Crear un Servicio

Para crear un servicio, se debe crear una subclase de Service (o una de sus ya existentes subclases).

En la implementación, se debe sobrescribir algunos de los métodos callbacks que gestionan los aspectos clave de los servicios del ciclo de vida y suministrar un mecanismo para que los componentes se unan al servicio.

Los métodos callback más importantes que se deben sobrescribir son:

onStartCommand()

El sistema llama a este método cuando otro componente (como una actividad), pide que el servicio se inicie, llamando al método startService(). Una vez que el método se ejecuta, el servicio se inicia y puede ejecutarse en un segundo plano indefinidamente. Si esto se implementa, hay que parar el servicio cuando termine su trabajo, mediante la llamada al método stopSelf() o al método stopService(). (Si solamente se quiere hacer el binding, no hay que implementar este método.)

onBind()

El sistema llama a este método cuando otro componente quiere unirse al servicio (como para realizar RPC), llamando al método bindService(). En la implementación de este método, se debe suministrar una interface para que los clientes la usen para comunicarse con el servicio, retornando un IBinder. Siempre se debe implementar este método, pero en el caso de que no se quiera permitir el binding (unión), se retorna null.

onCreate()

Cuando se crea el servicio por primera vez, el sistema llama a este método para realizar procedimientos de set-up (antes de llamar al método onStartCommand() o al método onBind()). Si el servicio ya está siendo ejecutado, no se llama a este método.

onDestroy()

Cuando el servicio ya no está siendo utilizado y se está eliminando el sistema llama a este método. El servicio debería ser implementado para limpiar cualquier tipo de recursos como hilos, listeners registrados, receptores, etc. Este es la última llamada que recibe el servicio.

Si un componente inicia un servicio a través de la llamada a startService() (que a su vez llama a onStartCommand()), entonces el servicio sigue ejecutándose hasta que o bien se para a si mismo con stopSelf() u otro componente lo para llamando al método stopService().

Si un componente llama al método bindService() para crear un servicio (y el método onStartCommand() no es llamado), entonces el servicio se ejecuta solamente mientras que el componente esté unido a él. Una vez que se desvincula el servicio de todos los clientes, el sistema lo destruye.

El sistema Android forzará a que un servicio se pare solamente cuando la memoria sea mínima y deba recuperar recursos del sistema para la actividad que tenga el focus del usuario. Si el usuario está vinculado a una actividad que tiene el focus del usuario, tendrá menos probabilidad de ser parada, y si el servicio está declarado para que se ejecute en primer plano, entonces casi nunca se parará. Si, por el contrario, el servicio se inició y es de larga duración, entonces el sistema bajará su posición en la lista de tareas en segundo plano y el servicio será susceptible de ser parado— si el servicio es iniciado, entonces se debe diseñar para que el sistema pueda reiniciarlo. Si el sistema para el servicio, lo reinicia tan pronto como se puedan conseguir los recursos de nuevo (aunque esto también depende del valor que se retorne del método onStartCommand(), como se verá más tarde).

En las secciones siguientes, se hablará sobre como crear cada tipo de servicio y como utilizarlos

desde otros componentes de la aplicación.

Debería usarse un servicio o un hilo?

Un servicio es un componente que puede ejecutarse en un segundo plano aunque el usuario no esté interactuando con la aplicación. Por esta razón sólo se debería crear un servicio cuando este se necesite.

Si necesita realizar un trabajo fuera del hilo principal, pero solamente mientras el usuario esté interaccionando con la aplicación, entonces debería crear un nuevo hilo en vez de un servicio. Por ejemplo, si se quiere poner música solamente mientras la actividad se esté ejecutando, se puede crear un hilo en el método `onCreate()`, empezar a ejecutarlo en `onStart()`, y pararlo en `onStop()`. Se puede utilizar también `AsyncTask` o `HandlerThread`, en vez de la clase tradicional `Thread`.

Hay que recordar que si se utiliza un servicio, se ejecuta por defecto en el hilo principal de la aplicación, por lo que si realizan operaciones intensivas o bloqueantes hay que crear un nuevo hilo dentro del servicio.

Declarar un servicio en el manifest

Tal y como se hace con las actividades (y otros componentes), los servicios se deben declarar en el archivo manifest de la aplicación.

Para declarar el servicio, hay que añadir un elemento `<service>` como hijo del elemento de la `<application>`.

Por ejemplo:

```
<manifest ... >
...
  <application ... >
    <service android:name=".ExampleService" />
    ...
  </application>
</manifest>
```

Existen otros atributos que se pueden incluir en el elemento `<service>` para definir propiedades tal y como los permisos necesarios para empezar el servicio y el proceso en el que el servicio debería ser ejecutado. Ver elemento referencia del `<service>` para más información.

Tal y como ocurre con una actividad, un servicio puede definir filtros intent que permiten que otros componentes llamen al servicio utilizando intents implícitos. Declarando los filtros intents, los componentes de cualquier otra aplicación instalada en el dispositivo del usuario pueden, potencialmente, iniciar el servicio siempre que el servicio declare un filtro intent que coincida con el intent que se pasa a través del `startService()` de la otra aplicación.

Si se quiere utilizar el servicio localmente (no lo utilizan otras aplicaciones), entonces no se necesita (ni se debe) suministrar ningún filtro intent. Sin esos filtros intent, se debe iniciar el servicio utilizando un intent que explícitamente nombre la clase del servicio. Más adelante se detalla más información sobre como iniciar un servicio.

Además, el servicio será privado a la aplicación solamente si se incluye el atributo `android:exported` y se setea a "false". Esto es efectivo incluso si el servicio suministra filtros intent.

Crear un servicio iniciado

Un servicio iniciado es uno que es iniciado por otro componente mediante la llamada al método `startService()`, que resulta en una llamada al método `onStartCommand()` del servicio.

Cuando un servicio es iniciado, tiene un ciclo de vida que es independiente al componente que lo inició y por lo tanto el servicio se puede ejecutar indefinidamente en un segundo plano, incluso si se destruye el componente que lo inició. Debido a esto, el servicio debería pararse a si mismo una vez ejecutado su función, mediante la llamada al método `stopSelf()`, o bien otro componente debería poder pararlo mediante la llamada al método `stopService()`.

Un componente de la aplicación (como una actividad) puede iniciar un servicio llamando a `startService()` y pasando un `Intent` que especifica el servicio e incluye cualquier dato necesario para el servicio. El servicio recibe este `Intent` en el método `onStartCommand()`.

Por ejemplo, si una actividad necesita guardar datos en una base de datos online, la actividad puede iniciar un servicio y entregarle los datos que quiere guardar, pasándole un `Intent` al método `startService()`. El servicio recibe el `Intent` en el método `onStartCommand()`, se conecta a Internet y realiza la transacción a la base de datos. Cuando la transacción se ha realizado, el servicio se para a si mismo y se destruye.

Precaución: Un servicio se ejecuta, por defecto, en el mismo proceso en el que está declarado en la aplicación y en el mismo hilo que esa aplicación. Por lo que, si el servicio realiza operaciones intensivas o bloqueantes mientras que el usuario interactúa con una actividad de la misma aplicación, el servicio ralentizará los procesos de la actividad. Para evitar este impacto, se debería iniciar un nuevo hilo dentro del servicio.

Existen dos clases de las que se puede extender para crear un servicio iniciado:

Service

Esta es la clase base para todos los servicios. Cuando se extiende de esta clase, es importante crear un nuevo hilo en el que hacer todo el trabajo del servicio, ya que el servicio, si no, por defecto, utiliza el hilo principal de la aplicación, lo que va a ralentizar los procesos de cualquier actividad que esté ejecutando la aplicación.

IntentService

Esta es una subclase de `Service` que utiliza un hilo trabajador para manejar todas las peticiones de inicio, una por una. Esta es la mejor opción si no se necesita que el servicio maneje varias peticiones simultáneamente. Lo que hay que hacer en este caso es implementar `onHandleIntent()`, que recibe el `Intent` para cada petición para que se pueda hacer todo el trabajo correspondiente al segundo plano.

Enfocado a Android 1.6 o una versión más baja

Si está desarrollando una aplicación para Android 1.6 o una versión más baja, se necesita implementar el método `onStart()`, en vez del método `onStartCommand()` (en Android 2.0, `onStart()` está en desuso en favor del método `onStartCommand()`).

Para más información sobre compatibilidad con versiones más viejas que Android 2.0 ver la documentación `onStartCommand()`.

Las siguientes secciones describen como se puede implementar un servicio utilizando cada una de estas clases.

Extender la clase `IntentService`

Ya que la mayoría de los servicios iniciados no necesitan manejar varias peticiones simultáneamente (lo que puede resultar un escenario multihilo peligroso), es mejor implementar el servicio utilizando la clase `IntentService`.

El IntentService hace lo siguiente:

- Crea un hilo trabajador por defecto que ejecuta todos los intents entregados a onStartCommand(), separados del hilo principal de la aplicación.
- Crea una cola de trabajo que pasa a la implementación de onHandleIntent() un intent cada vez, por lo que nunca hay que preocuparse por el multihilo.
- Para el servicio después de que todas las peticiones de inicio hayan sido gestionadas, por lo que nunca hay que llamar al método stopSelf().
- Aporta la implementación por defecto de onBind(), que devuelve null.
- Aporta la implementación por defecto de onStartCommand() que envía al intent a la cola de trabajo y después a la implementación del onHandleIntent().

Todo esto se suma al hecho de que todo lo que hay que hacer es implementar onHandleIntent() para realizar el trabajo dado por el cliente. Aunque, también se necesita suministrar un pequeño constructor para el servicio.

Aquí hay un ejemplo de la implementación de IntentService:

```
public class HelloIntentService extends IntentService {
    /**
     * Se necesita un constructor, y debe llamar al super constructor de IntentService(String)
     * con un nombre para el hilo trabajador.
     */
    public HelloIntentService() {
        super("HelloIntentService");
    }
    /**
     * El IntentService llama a este método desde el hilo trabajador por defecto
     * con el intent que inició el servicio. Cuando este método retorna, el IntentService
     * como corresponde, para el servicio.
     */
    @Override
    protected void onHandleIntent(Intent intent) {
        // Normalmente aquí se llevaría a cabo algún trabajo, como descargar un fichero.
        // Para este ejemplo, dormimos 5 segundos.
        long endTime = System.currentTimeMillis() + 5*1000;
        while (System.currentTimeMillis() < endTime) {
            synchronized (this) {
                try {
                    wait(endTime - System.currentTimeMillis());
                } catch (Exception e) {
                }
            }
        }
    }
}
```

Esto es todo lo que se necesita: un constructor y una implementación de onHandleIntent().

Si se quiere sobrescribir los métodos callback tales como onCreate(), onStartCommand(), o onDestroy(), hay que asegurarse de llamar a la implementación del super para que el IntentService pueda manejar adecuadamente la vida del hilo trabajador.

Por ejemplo, onStartCommand() debe devolver la implementación por defecto (que es como el intent se entrega a onHandleIntent()):

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();
    return super.onStartCommand(intent, flags, startId);
}
```

Aparte del método `onHandleIntent()`, el único método desde donde no se necesita llamar a la super clase es `onBind()` (pero solamente hay que implementarlo si el servicio permite el vínculo).

En la siguiente sección, se verá como se implementa el mismo tipo de servicio cuando se extiende la clase base `Service`, lo que resulta en más código, pero que puede ser apropiado si se necesita manejar varias peticiones de inicio simultáneas.

Extender la clase Service

Como se vió en la sección anterior, utilizar `IntentService` hace que la implementación de un servicio iniciado sea muy simple. Sin embargo, si se necesita que el servicio realice un proceso multihilo (en vez de peticiones de inicio de proceso a través de una cola de trabajo), se puede extender de la clase `Service` para manejar cada `Intent`.

El siguiente código es un ejemplo de implementación de la clase `Service` que realiza exactamente la misma función que en el ejemplo anterior utilizando `IntentService`.

Para cada petición de inicio, utiliza un hilo trabajador para realizar el trabajo y procesa solamente una petición cada vez.

```
public class HelloService extends Service {
    private Looper mServiceLooper;
    private ServiceHandler mServiceHandler;

    // Handler que recibe mensajes del hilo
    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
    }

    @Override
    public void handleMessage(Message msg) {
        // Normalmente aquí se llevaría a cabo algún trabajo, como descargar un fichero.
        // Para este ejemplo, dormimos 5 segundos.
        long endTime = System.currentTimeMillis() + 5*1000;
        while (System.currentTimeMillis() < endTime) {
            synchronized (this) {
                try {
                    wait(endTime - System.currentTimeMillis());
                } catch (Exception e) {
                }
            }
        }

        // Parar el servicio utilizando el startId, para que no se pare
        // el servicio en la mitad de la gestión de otro trabajo
        stopSelf(msg.arg1);
    }

    @Override
    public void onCreate() {
        // Ejecutar el hilo que inicia el servicio. Vease que se crea un
        // un hilo separado ya que el servicio normalmente se ejecuta en el hilo principal
        // del proceso, que no queremos bloquear. También lo hacemos
        // una prioridad del segundo plano para que el trabajo intensivo de la CPU no interrumpa nuestra IU.
        HandlerThread thread = new HandlerThread("ServiceStartArguments",
            Process.THREAD_PRIORITY_BACKGROUND);
        thread.start();

        // coger el looper del HandlerThread y usarlo para nuestro Handler
    }
}
```

```

mServiceLooper = thread.getLooper();
mServiceHandler = new ServiceHandler(mServiceLooper);
}
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

    // Para cada petición de inicio, mandar un mensaje para iniciar el trabajo y entregar
    // el ID de inicio para saber que petición estamos parando cuando se acabe el trabajo
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    mServiceHandler.sendMessage(msg);

    // Si se para después de volver de aquí, reiniciar.
    return START_STICKY;
}

@Override
public IBinder onBind(Intent intent) {
    // No se lleva a cabo el binding, por lo que retornamos null
    return null;
}

@Override
public void onDestroy() {
    Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
}
}

```

Como se puede ver, es mucho más trabajo que utilizando IntentService.

Sin embargo, de esta manera se pueden llevar a cabo varias peticiones simultaneamente, manejando cada llamada a onStartCommand(). Esto no es lo que hace este ejemplo, pero si es lo que se quiere hacer, se puede crear un nuevo hilo para cada petición y ejecutarlos inmediatamente (en vez de esperar a que la petición anterior finalice.)

Nótese que el método onStartCommand() debe retornar un integer. El integer es un valor que describe como el sistema debería continuar el servicio en el evento (como se ha explicado anteriormente la implementación por defecto de IntentService lo gestiona por tí, aunque lo puedes modificar). El valor que retorna el método onStartCommand() debe de ser una de las siguientes constantes:

START_NOT_STICKY

Si el sistema para el servicio después del retorno de onStartCommand(), no se debe recrear el servicio, a no ser que haya intents pendientes de entregar. Esta es la opción más segura para evitar ejecutar el servicio cuando no es necesario y cuando la aplicación simplemente puede reiniciar cualquiera de los trabajos no terminados.

START_STICKY

Si el sistema para el servicio después del retorno de onStartCommand(), se debe recrear el servicio y llamar a onStartCommand(), pero no re-entregar los últimos intent. En vez de esto el sistema llama al método onStartCommand() con un intent null, a no ser que haya intents pendientes para iniciar el servicio, en cuyo caso se entregan dichos intents. Esto es adecuado para los media players (o servicios similares) que no estén ejecutando comandos, pero que estén ejecutándose indefinidamente y esperando a un trabajo.

START_REDELIVER_INTENT

Si el sistema para el servicio después del retorno de onStartCommand(), se debe recrear el servicio y llamar al método onStartCommand() con el último intent que fue entregado al servicio. Cualquier intent pendiente es entregado por turno. Esto es adecuado para servicios

que están activamente realizando una función y que deben ser inmediatamente reanudados, como el de descargar un fichero.

Iniciar un servicio

Se puede iniciar un servicio desde una actividad o desde otro componente pasando un Intent (especificando el servicio a ser iniciado) al método startService(). El sistema Android llama al método onStartCommand() del servicio y le pasa el Intent. (No se debería llamar nunca directamente al método onStartCommand().)

Ejemplo: una actividad puede empezar el servicio de ejemplo de la sección anterior (HelloService) utilizando un intent explícito con el método startService():

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

El método startService() retorna inmediatamente y el sistema Android llama al método onStartCommand() del servicio. Si el servicio no está ya ejecutándose, el sistema llama primero al método onCreate(), y después al método onStartCommand().

Si el servicio no suministra binding, el intent entregado con el método startService() es el único modo de comunicación entre el componente de la aplicación y el servicio. Sin embargo, si se quiere que el servicio devuelva un resultado, entonces el cliente que inicia el servicio puede crear un PendingIntent para un broadcast (con getBroadcast()) y entregarlo al servicio en el Intent que inicia el servicio. El servicio, entonces, puede usar el broadcast para entregar el resultado.

Si tenemos varias peticiones para iniciar el servicio, tendremos sus correspondientes llamadas al método onStartCommand() del servicio. Sin embargo, solo se necesita una petición para parar el servicio (con stopSelf() o stopService()).

Parar un servicio

Un servicio iniciado debe gestionar su propio ciclo de vida. Esto significa, que el sistema no para o elimina el servicio a no ser que tenga que recuperar memoria del sistema y el servicio continúa después del retorno de onStartCommand(). Por lo tanto, el servicio debe pararse a si mismo, mediante la llamada al método stopSelf() u otro componente puede pararlo mediante la llamada al método stopService().

Una vez que se ha hecho la petición de parar con el método stopSelf() o con el método stopService(), el sistema destruye el servicio tan pronto como puede.

Sin embargo, si el servicio gestiona varias peticiones a la vez a onStartCommand(), entonces no se debería parar el servicio después de procesar una petición de inicio ya que mientras tanto se ha podido recibir una petición de inicio nueva (parar al final de la primera petición terminaría con la segunda). Para evitar este problema, se puede usar el método stopSelf(int) para asegurarse que la petición de parar el servicio siempre está basada en la petición de inicio más reciente. Cuando se llama al método stopSelf(int), se pasa el ID de la petición de inicio (el startId entregado al onStartCommand()) correspondiente a la petición de parar. Si el servicio recibe una nueva petición de inicio antes de poder llamar al stopSelf(int), entonces el ID no coincidiría y el servicio no se pararía.

Precaución: Es importante que la aplicación pare sus servicios cuando acabe con sus funciones, para evitar gastar los recursos del sistema y para evitar consumir la batería. Si fuera necesario, otros componentes pueden parar el servicio mediante la llamada al método stopService(). Incluso si habilita el binding para el servicio, se debe siempre parar el servicio si recibe una llamada al método onStartCommand().

Para más información sobre los ciclos de vida del servicio, ver las secciones sobre Gestionar el ciclo de vida de un Servicio.

Crear un Servicio vinculado (bound service)

Un servicio vinculado es uno que permite unirse a los componentes de la aplicación llamando al método `bindService()` para crear una conexión de larga duración (y generalmente no permite a los componentes iniciarlo mediante la llamada a `startService()`).

Se debería crear un servicio vinculado cuando se quiere interactuar con el servicio desde actividades u otros componentes de la aplicación o cuando se quiere exponer algunas funcionalidades de la aplicación a otras aplicaciones, a través de comunicación inter-proceso (IPC).

Para crear un servicio vinculado, se debe implementar el método callback `onBind()` para que devuelva un `IBinder` que define la interfaz para la comunicación con el servicio. Los componentes de otras aplicaciones pueden de esta manera llamar al método `bindService()` para recuperar la interfaz y empezar a llamar a métodos del servicio. El servicio vive solamente para servir al componente de la aplicación que está unido a él, por lo que cuando no hay componentes unidos al servicio, el sistema lo destruye (no se necesita parar un servicio vinculado en el camino, se debe hacer cuando el servicio es iniciado a través del método `onStartCommand()`).

Para crear un servicio vinculado, lo primero que se debe hacer es definir la interfaz que especifica como el cliente se puede comunicar con el servicio. Esta interfaz entre el servicio y el cliente debe de ser una implementación del `IBinder` y es lo que el servicio debe devolver desde el método callback `onBind()`. Una vez que el cliente recibe el `IBinder`, puede empezar a interactuar con el servicio a través de la interfaz.

Se pueden vincular varios clientes al servicio al mismo tiempo. Cuando un cliente ha terminado de interactuar con el servicio, llama al método `unbindService()` para desvincularse. Una vez que no existan clientes vinculados al servicio, el sistema destruye el servicio.

Existen diferentes maneras de implementar un servicio vinculado y la implementación resulta más complicada que un servicio iniciado, por lo que los detalles sobre los servicios vinculados aparecen en un documento separado sobre "Servicios Vinculados (Bound Services)".

Mandar notificaciones al usuario

Una vez que se está ejecutando, un servicio puede notificar al usuario los eventos utilizando Notificaciones Toast o Notificaciones de la barra de estado.

Una notificación toast es un mensaje que aparece por un momento en la superficie de la ventana en curso y después desaparece, mientras que una notificación de barra de estado es un icono en la barra de estado con un mensaje, que el usuario puede seleccionar para llevar a cabo una acción (como iniciar una actividad).

Normalmente, una notificación en la barra de estado es la mejor técnica para que una vez finalizado un trabajo (como la descarga de un archivo) en segundo plano, el usuario pueda actuar sobre él. Cuando el usuario selecciona la notificación desde la vista expandida, la notificación puede empezar una actividad (como ver el archivo descargado).

Véase la documentación sobre Notificaciones Toast o Notificaciones de barra de estado para más información.

Ejecutar un servicio en primer plano

Un servicio en primer plano es un servicio que el usuario tiene presente y por tanto no es un candidato a ser parado por el sistema cuando tenga baja memoria. Un servicio en primer plano debe suministrar una notificación para la barra de estado, que estará localizado debajo de la cabecera "en

curso" lo que significa que la notificación no puede ser descartada a no ser que el servicio sea parado o eliminado del primer plano.

Ejemplo; un reproductor de música que reproduce música desde un servicio debería ser configurado para ser ejecutado en primer plano, ya que el usuario es consciente de la operación. La notificación en la barra de estado puede indicar la canción en curso y permitir al usuario a lanzar una actividad que interactúe con el reproductor de música.

Para pedir que el servicio se ejecute en primer plano, hay que llamar al método `startForeground()`. Este método lleva dos parámetros: un integer que identifica unívocamente la notificación y la Notificación para la barra de estado.

Por ejemplo:

```
Notification notification = new Notification(R.drawable.icon, getText(R.string.ticker_text),
    System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);
notification.setLatestEventInfo(this, getText(R.string.notification_title),
    getText(R.string.notification_message), pendingIntent);
startForeground(ONGOING_NOTIFICATION, notification);
```

Para eliminar el servicio del primer plano, hay que llamar al método `stopForeground()`. Este método lleva un boolean, indicando si hay que eliminar también la notificación de la barra de estado. Este método no para el servicio.

Sin embargo, si se para el servicio mientras se está ejecutando en primer plano, la notificación también se elimina.

Nota: Los métodos `startForeground()` y `stopForeground()` fueron introducidos en Android 2.0 (API nivel 5). Para ejecutar servicios en primer plano en versiones más antiguas de la plataforma, se debe utilizar el método `setForeground()`—ver la documentación sobre `startForeground()` para información sobre como obtener compatibilidad hacia atrás.

Para más información sobre notificaciones, ver Crear notificaciones de barra de estado.

Gestionar el ciclo de vida de un servicio

El ciclo de vida de un servicio es mucho más simple que el de una actividad. Sin embargo, es muy importante saber como un servicio es creado y destruido, ya que un servicio puede ser ejecutado en un segundo plano sin que el usuario lo sepa.

El ciclo de vida de un servicio—desde que es creado hasta cuando es destruido—puede seguir dos caminos diferentes:

- Un servicio iniciado

El servicio es creado cuando otro componente llama al método `startService()`. El servicio en ese momento se ejecuta indefinidamente y debe pararse a si mismo llamando al método `stopSelf()`. Otro componente también puede parar el servicio mediante la llamada al método `stopService()`. Cuando se para el servicio, el sistema lo elimina.

- Un servicio vinculado(bound service)

El servicio se crea cuando otro componente (un cliente) llama al método `bindService()`. El cliente en ese momento se comunica con el servicio a través de la interfaz `IBinder`. El cliente puede cerrar la conexión llamando al `unbindService()`. Varios clientes se pueden vincular al mismo servicio y cuando todos se desvinculan, el sistema destruye el servicio. (El servicio no necesita pararse a si mismo).

Estos dos caminos no están separados completamente. Se puede vincular a un servicio que ya estaba

iniciado mediante un `startService()`. Por ejemplo, un servicio de música en segundo plano se puede iniciar llamando al `startService()` con un `Intent` que identifica la música a reproducir. Posteriormente, cuando el usuario quiera ejercer control sobre el reproductor u obtener información sobre la canción en curso, una actividad se puede vincular al servicio llamando al método `bindService()`. En casos como este, `stopService()` o `stopSelf()` no paran el servicio hasta que todos los clientes se desvinculan.

Implementar los callbacks del ciclo de vida

Como en una actividad, los servicios tienen métodos callbacks de ciclo de vida que se pueden implementar para monitorizar cambios en el estado del servicio y realizar funciones en los momentos oportunos.

La siguiente estructura del servicio nos muestra cada uno de los métodos del ciclo de vida:

```
public class ExampleService extends Service {
    int mStartMode;    // Indica como comportarse si el servicio es parado
    IBinder mBinder;   // interface para clientes que vinculan
    boolean mAllowRebind; // indica si se debería utilizar onRebind

    @Override
    public void onCreate() {
        // El servicio se está creando
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // El servicio se está iniciando, por la llamada al startService()
        return mStartMode;
    }

    @Override
    public IBinder onBind(Intent intent) {
        // Un cliente se está vinculando al servicio con bindService()
        return mBinder;
    }

    @Override
    public boolean onUnbind(Intent intent) {
        // Todos los clientes se han desvinculado con unbindService()
        return mAllowRebind;
    }

    @Override
    public void onRebind(Intent intent) {
        // Un cliente se está vinculando al servicio con bindService(),
        // después de que se llame al método onUnbind()
    }

    @Override
    public void onDestroy() {
        // El servicio ya no se utiliza y está siendo eliminado
    }
}
```

Nota: A diferencia de los métodos callback del ciclo de vida de una actividad, no se necesita llamar a la implementación de la superclase de estos métodos callback.

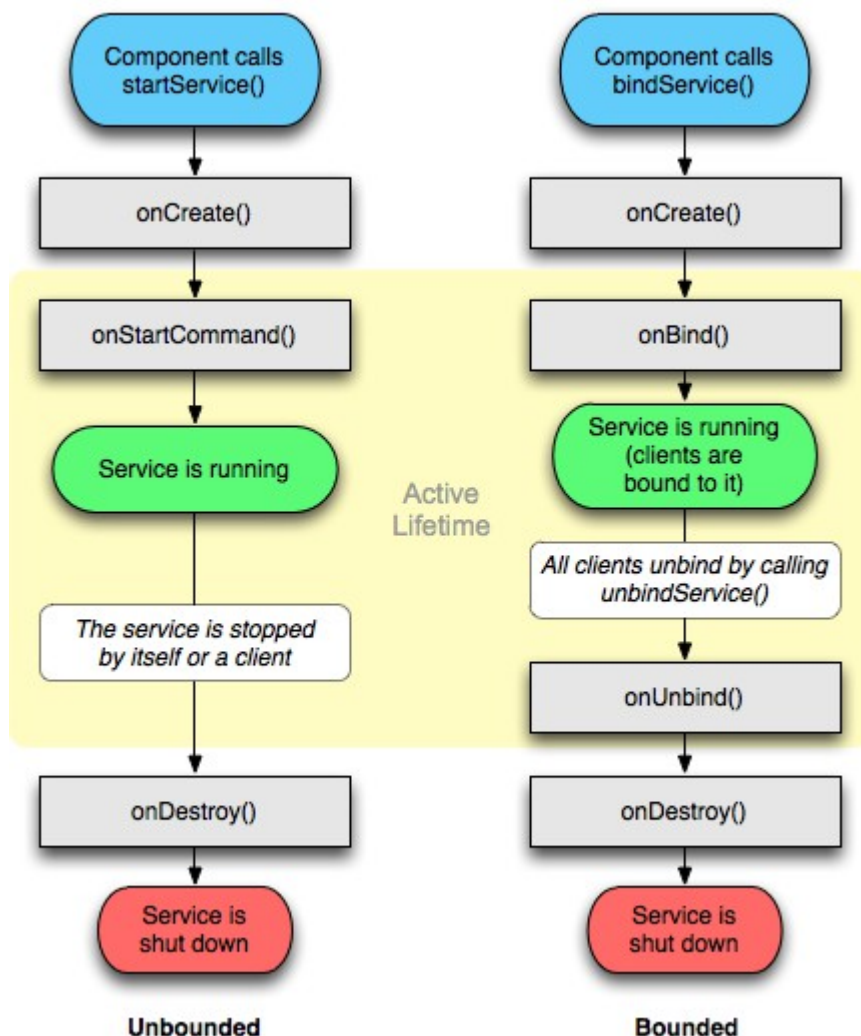


Figura 2. Ciclo de vida de un servicio. El diagrama de la izquierda muestra el ciclo de vida cuando el servicio es creado con `startService()` y el diagrama a la derecha muestra el ciclo de vida cuando un servicio es creado con `bindService()`.

Implementando estos métodos, se pueden monitorizar dos loops anidados del ciclo de vida del servicio:

- El tiempo completo de un servicio ocurre entre el tiempo en el que el método `onCreate()` es llamado y el tiempo en el que `onDestroy()` vuelve. Como ocurre en una actividad, un servicio hace su configuración inicial en `onCreate()` y suelta todos los recursos restantes en `onDestroy()`. Ejemplo; un servicio de reproducción de música podría crear el hilo donde se va a reproducir la música en `onCreate()` y después parar el hilo en `onDestroy()`.

Los métodos `onCreate()` y `onDestroy()` son llamados para todos los servicios, bien si son creados por `startService()` o por `bindService()`.

- El tiempo de vida activo de un servicio empieza con una llamada a bien `onStartCommand()` o `onBind()`. A cada método se le pasa el `Intent` que se había pasado a bien el `startService()` o al `bindService()`, respectivamente. Si el servicio es iniciado, el tiempo de vida activo acaba al mismo tiempo que acaba el tiempo de vida completo (el servicio está todavía activo aún después de que retorne `onStartCommand()`). Si el servicio está vinculado, el tiempo de vida activo acaba cuando retorna `onUnbind()`.

Nota: A pesar de que un servicio iniciado se para por la llamada a bien el método `stopSelf()` o al

método `stopService()`, no existe un callback correspondiente al servicio (no hay un callback `onStop()`). Por lo que, a no ser que el servicio esté vinculado a un cliente, el sistema lo destruye cuando el servicio es parado—`onDestroy()` es el único callback recibido.

La figura 2 ilustra los métodos callback más usuales para el servicio. A pesar de que la figura separa los servicios que son creados por el método `startService()` de los que son creados por `bindService()`, hay que tener en cuenta que cualquier servicio, sin tener en cuenta como se haya iniciado, puede potencialmente permitir a los clientes vincularse a ellos. Así, un servicio que se inició con `onStartCommand()` (por un cliente llamando al `startService()`) puede todavía recibir una llamada al método `onBind()` (cuando un cliente llama al `bindService()`).

Para más información sobre como crear un servicio que puede vincularse ver el documento sobre Servicios Vinculados (Bound Services) que incluye más información sobre el método callback `onRebind()` en la sección sobre Gestionar el ciclo de vida de los servicios vinculados.

Qué es un Servicio vinculado (bound service) y cómo se crea.

Un servicio vinculado es el servidor en una interfaz cliente-servidor. Permite que componentes (como las actividades) se vinculen a un servicio, manden peticiones, reciban respuestas, y realicen comunicaciones inter-procesos (IPC). Este tipo de servicio solamente existe mientras sirve a otro componente de una aplicación y no se ejecuta indefinidamente en un segundo plano.

Este documento muestra como crear un servicio vinculado, incluyendo como vincular componentes de la aplicación al servicio. Para ver información general sobre servicios, como entregar notificaciones desde un servicio, configurar el servicio para que se ejecute en primer plano, etc, ver el documento Servicios.

Básicos

Un servicio vinculado es una implementación de la clase `Service` que le permite a otras aplicaciones vincularse así como interactuar con él. Para poder vincularse a un servicio, se debe implementar el método callback `onBind()`. Este método devuelve un objeto `IBinder` que define la interfaz que los clientes pueden usar para interactuar con el servicio

Un cliente se puede vincular a un servicio llamando al método `bindService()`. Al hacerlo debe aportar una implementación del `ServiceConnection`, que monitoriza la conexión con el servicio. El método `bindService()` retorna inmediatamente sin un valor, pero cuando el sistema Android crea la conexión entre el cliente y el servicio, llama al método `onServiceConnected()` de `ServiceConnection`, para entregar el `IBinder` que el cliente va a poder utilizar para comunicarse con el servicio.

Varios clientes se pueden conectar al servicio a la vez. Sin embargo, el sistema llama al método `onBind()` del servicio para recuperar el `IBinder` sólo cuando se vincula el primer cliente. El sistema entrega ese mismo `IBinder` a cualquier otro cliente que se vincula, sin volver a llamar al método `onBind()`.

Cuando el último cliente se desvincula del servicio, el sistema destruye el servicio (a no ser que el servicio haya sido iniciado por el método `startService()`).

Cuando se implementa el servicio vinculado, la parte más importante es definir la interfaz que devuelve el método callback `onBind()`. Existen varias maneras de definir la interfaz del servicio `IBinder` que discutimos a continuación.

Crear un servicio vinculado.

Cuando se crea un servicio que puede ser vinculado, se debe suministrar un `IBinder` que aporta la interfaz de programación que los clientes pueden utilizar para interactuar con el servicio. Existen tres maneras de definir la interfaz:

Extender la clase Binder

Si el servicio es privado de la aplicación y se ejecuta en el mismo proceso que el cliente (lo que es común), se debe crear una interfaz extendiendo la clase Binder y devolviendo una instancia suya desde el método onBind(). El cliente recibe el Binder y lo puede usar para acceder directamente a métodos públicos disponibles o bien en la implementación del Binder o incluso del Service.

Esta es la mejor técnica cuando el servicio trabaja en segundo plano de la aplicación. La única razón para no crear la interfaz de esta manera es si el servicio es utilizado por otras aplicaciones o en procesos separados.

Utilizar un mensajero (Messenger)

Si se necesita que la interfaz trabaje en varios procesos, se puede crear una interfaz para el servicio con un Messenger. De esta manera, el servicio define un Handler que responde a diferentes tipos de objetos Message. Este Handler es la base de un Messenger que puede compartir un IBinder con el cliente, permitiendo mandar al cliente comandos al servicio utilizando objetos Message. Adicionalmente, el cliente puede definir un Messenger propio para que el servicio pueda mandar mensajes de vuelta.

Esta es la manera más simple de realizar una comunicación inter-proceso (IPC), ya que el Messenger pone en cola todas las peticiones en un hilo único para que el servicio tenga seguridad en hilos (thread-safe).

Utilizar AIDL

El AIDL (Lenguaje de Definición de Interfaz Android) realiza todo el trabajo para descomponer objetos en primitivos que pueda entender el sistema operativo y reunirlos a través de los procesos para realizar IPC. La técnica anterior, que utiliza un Messenger, tiene como estructura de base un AIDL. Como se menciona anteriormente, el Messenger crea una cola con todas las peticiones del cliente en un solo hilo, para que el servicio reciba una sola petición a la vez. Si, por el contrario, se quiere que el servicio gestione varias peticiones a la vez, entonces se puede usar directamente AIDL. En este caso, el servicio debe ser capaz de hacer un trabajo multi-hilo y tiene que ser desarrollado de manera thread-safe.

Para usar directamente AIDL, se debe crear un archivo .aidl que define la interfaz de programación. Las herramientas de Android SDK utilizan este archivo para generar una clase abstracta que implementa la interfaz y gestiona la IPC, que se pueden extender dentro del servicio.

Nota: La mayoría de las aplicaciones no deberían utilizar AIDL para crear un servicio vinculado, ya que pueden necesitar capacidades multi-hilo y pueden crear una implementación más complicada. Como tal, AIDL no es adecuado para la mayoría de las aplicaciones y este documento no detalla como utilizarlo en los servicios.

Extender la clase Binder

Si el servicio es utilizado solamente por la aplicación local y no necesita trabajar a través de procesos, se puede implementar una clase Binder propia que le da al cliente acceso a métodos públicos en el servicio.

Nota: Esto funciona sólo si el cliente y el servicio están en la misma aplicación y proceso, lo que es bastante común. Esto funcionaría, por ejemplo, para una aplicación de música que necesita vincular una actividad a su propio servicio que está ejecutando música en un segundo plano.

Así se configura:

1. En el servicio, crear una instancia de Binder que bien:

- contenga métodos públicos a los que el cliente pueda llamar
- devuelva la instancia del Service en curso, la cual tiene métodos públicos a los que el cliente puede

llamar

- o bien, devuelve una instancia de otra clase hosteada por el servicio con métodos públicos a los que el cliente puede llamar

2.Devolver esta instancia del Binder desde el método callback onBind().

3.En el cliente, recibir el Binder desde el método callback onServiceConnected() y hacer llamadas al servicio vinculado utilizando los métodos suministrados.

Nota: La razón por la que el servicio y el cliente debe estar en la misma aplicación es para que el cliente pueda castear el objeto devuelto y llamar su API correspondiente. El servicio y el cliente deben estar también en el mismo proceso.

Como ejemplo, aquí tenemos un servicio que suministra acceso de clientes a métodos en servicios a través de la implementación del Binder:

```
public class LocalService extends Service {
    // Binder dado a los clientes
    private final IBinder mBinder = new LocalBinder();
    // Generador aleatorio de números
    private final Random mGenerator = new Random();
    /**
     * Clase utilizada por el cliente Binder. Como se sabe que este servicio
     * se ejecuta siempre en el mismo proceso que sus clientes, no se necesita tratar con el IPC.
     */
    public class LocalBinder extends Binder {
        LocalService getService() {
            // Devolver esta instancia del LocalService para que los clientes puedan llamar a métodos públicos
            return LocalService.this;
        }
    }
    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    /** método para clientes */
    public int getRandomNumber() {
        return mGenerator.nextInt(100);
    }
}
```

El LocalBinder suministra el método getService() para que los clientes recuperen la instancia del LocalService en curso. Esto permite a los clientes llamar a los métodos públicos del servicio. Los clientes, por ejemplo, pueden llamar al método getRandomNumber() desde el servicio.

Aquí se muestra una actividad que se vincula al LocalService y llama al método getRandomNumber() cuando se hace click en un botón:

```
public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;

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

    @Override
    protected void onStart() {
        super.onStart();

        // Se vincula al LocalService
    }
}
```

```

Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
}

@Override
protected void onStop() {
    super.onStop();
    // se desvincula del servicio
    if (mBound) {
        unbindService(mConnection);
        mBound = false;
    }
}

/** Se llama cuando se hace click en un botón (el botón en el archivo layout se une a
 * este método con el atributo android:onClick) */
public void onClick(View v) {
    if (mBound) {
        // Llama a un método desde el LocalService.
        // Sin embargo, si en esta llamada hubiera algo suspendido, entonces la petición debería
        // realizarse en un hilo por separado para evitar la ralentización de la actividad.
        int num = mService.getRandomNumber();
        Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT).show();
    }
}

/** Define los callback pasados al método bindService() para la vinculación de los servicios*/
private ServiceConnection mConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName className,
        IBinder service) {
        // Se ha hecho la vinculación al LocalService, el cast del IBinder y un get de la instancia del LocalService
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        mBound = false;
    }
};
}

```

El ejemplo anterior muestra como el cliente se une al servicio utilizando la implementación del ServiceConnection y del método callback onServiceConnected().

Nota: En el ejemplo anterior no se desvincula explícitamente del servicio, pero todos los clientes deberían desvincularse en el momento adecuado (cuando la actividad se pone en pausa).

Utilizar un Messenger

Si se necesita que el servicio se comunique con procesos remotos, entonces se puede usar un Messenger para suministrar la interfaz para el servicio. Esta técnica permite que se realice una comunicación interproceso (IPC) sin la necesidad de usar AIDL.

A continuación hay un resumen sobre como utilizar un Messenger:

- El servicio implementa un Handler que recibe un callback para cada llamada de un cliente.
- El Handler es utilizado para crear un objeto Messenger (que es una referencia al Handler).
- El Messenger crea un IBinder que el servicio devuelve a los clientes desde onBind().
- Los clientes utilizan el IBinder para instanciar el Messenger (que referencia el Handler del

servicio), que el cliente utiliza para mandar objetos Message al servicio.

- El servicio recibe cada Message en su Handler—específicamente, en su método handleMessage(). De esta manera, no hay "métodos" del servicio para que el cliente los llame. En vez de esto, el cliente entrega "mensajes" (objetos Message) que el servicio recibe en su Handler.

Aquí hay un ejemplo simple de un servicio que utiliza la interfaz Messenger:

```
public class MessengerService extends Service {
    /** Comando para que el servicio muestre un mensaje */
    static final int MSG_SAY_HELLO = 1;
    /**
     * Handler de mensajes que vienen de los clientes.
     */
    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_SAY_HELLO:
                    Toast.makeText(getApplicationContext(), "hello!", Toast.LENGTH_SHORT).show();
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }

    /**
     * Diana que publicamos para que los clientes manden mensajes al IncomingHandler.
     */
    final Messenger mMessenger = new Messenger(new IncomingHandler());

    /**
     * Cuando se vincula al servicio, se devuelve una interfaz a nuestro mensajero
     * para mandar mensajes al servicio.
     */

    @Override
    public IBinder onBind(Intent intent) {
        Toast.makeText(getApplicationContext(), "binding", Toast.LENGTH_SHORT).show();
        return mMessenger.getBinder();
    }
}
```

Nótese que el método handleMessage() en el Handler es donde el servicio recibe el Message entrante y decide que hacer, basándose en el miembro what.

Todo lo que un cliente necesita hacer es crear un Messenger basándose en el IBinder devuelto por el servicio y mandar un mensaje utilizando send(). Aquí tenemos, como ejemplo, una simple actividad que se vincula al servicio y entrega el mensaje MSG_SAY_HELLO al servicio:

```
public class ActivityMessenger extends Activity {
    /** Mensajero para comunicarse con el servicio.
     */
    Messenger mService = null;
    /** Flag indicando si se ha llamado a bind en el servicio.
     */
    boolean mBound;
    /**
     * Clase para interactuar con la interfaz principal del servicio.
     */
    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            // Se llama a este método cuando la conexión con el servicio ha sido
            // establecida, dándonos el objeto que podemos utilizar
            // para interactuar con el servicio. Nos estamos comunicando con el
        }
    };
}
```

```

// servicio utilizando un Messenger, por lo que aquí obtenemos una representación
// del lado del cliente del objeto IBinder.
mService = new Messenger(service);
mBound = true;
}

public void onServiceDisconnected(ComponentName className) {
// Se llama a este método cuando la conexión con el servicio ha sido
// desconectada inesperadamente-- su proceso ha sido terminado.
mService = null;
mBound = false;
}

};

public void sayHello(View v) {
if (!mBound) return;
// Crear y mandar un mensaje al servicio, utilizando un valor 'what'.
Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
try {
    mService.send(msg);
} catch (RemoteException e) {
    e.printStackTrace();
}
}

}

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

@Override
protected void onStart() {
super.onStart();
// Vincular al servicio
bindService(new Intent(this, MessengerService.class), mConnection,
Context.BIND_AUTO_CREATE);
}

@Override
protected void onStop() {
super.onStop();
// Desvincular del servicio
if (mBound) {
    unbindService(mConnection);
    mBound = false;
}
}
}

```

Nótese que este ejemplo no muestra como el servicio puede responder al cliente. Si quiere que el servicio responda, entonces se necesita crear también un Messenger en el cliente. Cuando el cliente recibe el callback onServiceConnected(), manda un Message al servicio que incluye el Messenger del cliente en el parámetro replyTo del método send().

Cómo vincular los componentes de una aplicación a un servicio

Los componentes de una aplicación (zclientes) pueden vincularse a un servicio llamando al método bindService(). El sistema Android llama en ese momento al método onBind() del servicio, que devuelve un IBinder para interactuar con el servicio.

La vinculación es asíncrona. bindService() devuelve inmediatamente y no devuelve al cliente el IBinder. Para recibir el IBinder, el cliente debe crear una instancia de ServiceConnection y

pasárselo a `bindService()`. El `ServiceConnection` incluye un método callback al que llama el sistema para entregar el `IBinder`.

Nota: Solamente las actividades, servicios y los content providers pueden vincularse a un servicio — no se puede vincular desde un broadcast receiver a un servicio.

Para vincularse a un servicio desde un cliente, se debe:

1. Implementar el `ServiceConnection`.

La implementación debe sobrescribir dos métodos callback:

`onServiceConnected()`

El sistema lo llama para entregar el `IBinder` devuelto por el método `onBind()` del servicio.

`onServiceDisconnected()`

El sistema Android lo llama cuando la conexión al servicio se pierde inesperadamente, como por ejemplo cuando el servicio se bloquea o se elimina. No se llama cuando el cliente se desvincula.

2. Llamar al método `bindService()`, pasando la implementación del `ServiceConnection`.

3. Cuando el sistema llama al método callback `onServiceConnected()` se puede iniciar la llamada al servicio, utilizando métodos definidos por la interfaz.

4. Para desconectarse del servicio, hay que llamar al `unbindService()`.

5. Cuando el cliente es destruido, se desvinculará del servicio, pero se debería desvincular siempre que se termine la interacción con el servicio o cuando la actividad se pone en pausa para que el servicio se puede apagar mientras no se esté usando.

El siguiente snippet conecta el cliente con el servicio creado anteriormente a través de la extensión de la clase `Binder`, por lo que tiene que hacer simplemente es castear el `IBinder` devuelto a la clase `LocalService` y pedir una instancia de `LocalService`:

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Se llama cuando se establece la conexión con el servicio
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Dado que se ha producido la vinculación a un servicio
        // que se está ejecutando en nuestro proceso, se puede
        // castear el IBinder a una clase concreta y acceder directamente a ella.
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;    }

    // Se llama cuando la conexión con el servicio se desconecta inesperadamente
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "onServiceDisconnected");
        mBound = false;
    }
};
```

Con este `ServiceConnection`, el cliente se puede vincular a un servicio pasándoselo al `bindService()`. Por ejemplo:

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

- El primer parámetro del `bindService()` es un `Intent` que explícitamente nombra al servicio al que se vincula (a pesar de que el intent puede ser implícito).
- El segundo parámetro es el objeto `ServiceConnection`.
- El tercer parámetro es un flag indicando las opciones para la vinculación. Para crear el servicio si no está todavía vivo, debería ser `BIND_AUTO_CREATE`. Otros posibles valores son `BIND_DEBUG_UNBIND` y `BIND_NOT_FOREGROUND`.

Notas adicionales

A continuación se detallan unas notas sobre como vincularse a un servicio:

- Se deberían manejar siempre las excepciones DeadObjectException, que son lanzadas cuando la conexión se rompe. Esta es la única excepción lanzada por métodos remotos.
- Los objetos son referencias enumeradas a través de los procesos.
- Se debería emparejar los vinculados y desvinculados durante los momentos correspondientes de ejecución y parada del ciclo de vida de los clientes. Por ejemplo:
 - Si solamente se necesita interactuar con el servicio mientras la actividad es visible, se debería hacer la vinculación durante onStart() y la desvinculación durante onStop().
 - Si se quiere que la actividad reciba respuestas mientras está parado en un segundo plano, se puede vincular durante onCreate() y desvincular durante onDestroy(). Hay que tener en cuenta que esto implica que la actividad necesita usar al servicio durante todo el tiempo de ejecución (incluso en un segundo plano), por lo que si el servicio está en otro proceso, esto hace que se aumente el peso del proceso y se aumente la probabilidad de que el sistema lo termine.

Nota: No se debería vincular y desvincular durante los métodos onResume() y onPause() de la actividad, ya que estos callbacks son llamados en cada transición del ciclo de vida y se deberían mantener al mínimo estas transiciones. Si diversas actividades de la aplicación se vinculan al mismo servicio y no existe transición entre dos de esas actividades, el servicio puede ser destruido y recreado cuando la actividad en curso se desvincule (durante una pausa) antes de que la siguiente se vincule (durante la reanudación).

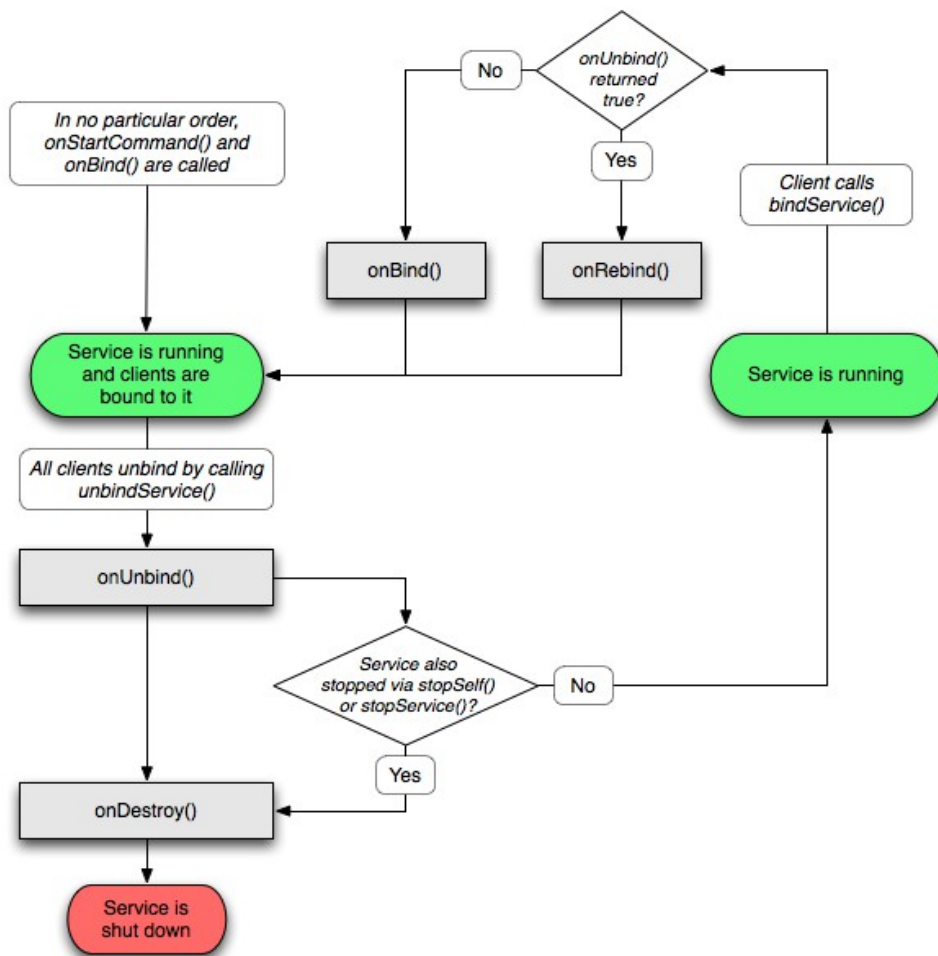
Como gestionar el ciclo de vida de un servicio que está iniciado y que permite vinculación.

Cuando un servicio está desvinculado de todos los clientes, el sistema Android lo destruye (a no ser que haya sido iniciado con onStartCommand()). De esta manera, no se tiene que gestionar el ciclo de vida del servicio si es puramente un servicio vinculado—el sistema Android lo gestiona por ti basándose en si está vinculado a alguno de los clientes.

Sin embargo, si se escoge implementar el método callback onStartCommand(), se debe parar el servicio explícitamente, ya que el servicio está considerado como iniciado. En este caso, el servicio se ejecuta hasta que el servicio se para a si mismo con el método stopSelf() o si otro componente llama al método stopService(), sin importar si está vinculado a algún cliente.

Además, si el servicio está iniciado y acepta ser vinculado, cuando el sistema llama al método onUnbind(), se puede devolver opcionalmente true si la próxima vez que un cliente se vincule a un servicio, se quiere recibir una llamada al método onRebind() (en vez de recibir una llamada al onBind()). El método onRebind() devuelve void, pero el cliente sigue recibiendo el IBinder en su callback onServiceConnected().

La figura 1, ilustra la lógica de este tipo de ciclo de vida.



Qué es un Content Provider

Los content provider almacenan y recuperan datos haciéndolos accesibles a todas las aplicaciones. Son la única manera de compartir datos a través de las aplicaciones; no existe un área de almacén común al que puedan acceder todos los paquetes Android.

Android viene con un número determinado de "content provider" para tipos de datos comunes (audio, video, imágenes, información personal de contactos, etc.) Se pueden ver algunos de ellos en el paquete `android.provider`. Se pueden hacer consultas para conseguir los datos que tienen (aunque para algunos de ellos, se deben conseguir permisos adecuados).

Si se quiere hacer públicos los datos, se tiene dos opciones: Se puede crear un content provider (una subclase de `ContentProvider`) o se pueden añadir los datos a un provider ya existente — si existe uno que controle el mismo tipo de datos y se tiene permiso para escribir en él.

Este documento es una introducción para utilizar los content providers. Primero habla sobre los fundamentos, después detalla como hacer una consulta a un content provider, como modificar los datos controlados por un provider y por último como crear un content provider propio.

Básicos sobre Content Provider

Como almacena el content provider los datos es a gusto del diseñador. Pero todos los content providers implementan una interfaz común para realizar consultas al provider así como para añadir, alterar o borrar datos.

Los clientes utilizan indirectamente una interfaz, generalmente a través de objetos `ContentResolver`. Se consigue el `ContentResolver` llamando al método `getContentResolver()` desde la implementación

de una actividad o de otro componente de la aplicación:

```
ContentResolver cr = getContentResolver();
```

A partir de este momento se puede utilizar los métodos del ContentResolver para interactuar con el content provider que se quiera.

Cuando se inicia una consulta, el sistema Android identifica al content provider diana de la consulta y se asegura que se está ejecutando. El sistema instancia todos los objetos ContentProvider; no necesita hacerlo uno mismo. De hecho, no existe una interacción directa con los objetos ContentProvider. Normalmente, existe una sola instancia de cada tipo de ContentProvider, pero esta se puede comunicar con varios objetos ContentResolver en diferentes aplicaciones y procesos. La interacción entre procesos es manejada por las clases ContentResolver y ContentProvider.

El modelo de datos

Los content providers exponen sus datos en una tabla sencilla en un modelo de base de datos , donde cada fila es un registro y cada columna es un tipo de dato y significado particular. En el ejemplo a continuación, tenemos como se exponen los datos sobre información de personas y sus números de teléfono:

_ID	NUMBER	NUMBER_KEY	LABEL	NAME	TYPE
13	(425) 555 6677	425 555 6677	Kirkland office	B. Vain	TYPE_WORK
44	(212) 555-1234	212 555 1234	NY apartment	A. Dylan	TYPE_HOME
45	(212) 555-6657	212 555 6657	Downtown office	A. Dylan	TYPE_MOBILE
53	201.555.4433	201 555 4433	Love Nest	Rex Cars	TYPE_HOME

Cada registro incluye un campo numérico _ID que identifica unívocamente el registro dentro de la tabla. Los IDs pueden ser utilizados para machear registros en tablas relacionadas— ejemplo; para encontrar un número de teléfono de una persona en una tabla y fotos de esa persona en otra.

Una consulta devuelve un objeto Cursor que se mueve de un registro a otro y de columna en columna para leer los contenidos de cada campo. Tiene métodos especializados para leer cada tipo de dato. Para leer un campo, se debe conocer el tipo de dato que contiene el campo.

URIs

Cada content provider expone un URI público (envuelto como un objeto Uri) que indentifica unívocamente su conjunto de datos. Un content provider que controla varios conjuntos de datos (múltiples tablas), tiene un URI por cada uno. Todas las URIs de los providers empiezan con el string "content://". El esquema content: identifica que los datos están controlados por un content provider.

Si se está definiendo un content provider, es una buena idea definir una constante para su URI, para simplificar el código del cliente y para que las futuras actualizaciones sean más sencillas. Android define las constantes CONTENT_URI para todos los providers que vengan con la plataforma. Ejemplo; el URI de la tabla que machea los números de teléfono con personas y el URI de la tabla que guarda las fotos de las personas (los dos controlados por el content provider Contacts) son:

```
android.provider.Contacts.Phones.CONTENT_URI  
android.provider.Contacts.Photos.CONTENT_URI
```

La constante URI se utiliza en todas las interacciones con el content provider. Cada método ContentResolver toma el URI como su primer argumento. Es lo que identifica a que

provider se tiene que dirigir el `ContentResolver` y que tabla del provider es el objetivo.

Realizar una consulta al Content Provider

Se necesitan tres datos para hacer una consulta al content provider:

- El URI que identifica al provider
- Los nombres de los campos que se quiere recibir
- Los tipos de datos para esos campos

Si se está haciendo la consulta a un registro en particular, también se necesita el ID de ese registro.

Realizar la consulta

Para hacer la consulta al content provider, se puede usar bien el método `ContentResolver.query()` o el método `Activity.managedQuery()`. Ambos métodos reciben el mismo conjunto de argumentos, y los dos devuelven el objeto `Cursor`.

El primer argumento tanto para `query()` como para `managedQuery()` es el URI provider — la constante `CONTENT_URI` que identifica un `ContentProvider` particular y un conjunto de datos.

Para restringir una consulta a un sólo registro, se puede agregar el valor del `_ID` para ese registro al URI— esto significa colocar un string que coincida con el ID del último segmento de la ruta del URI. Si, por ejemplo, el ID es 23, el URI sería:

`content://.../23`

Existen algunos métodos que ayudan; especialmente `ContentUris.withAppendedId()` y `Uri.withAppendedPath()`, que hacen fácil añadir un ID a un URI. Los dos son métodos estáticos que devuelven un objeto URI con el ID añadido. Así, por ejemplo, si buscamos el registro 23 en la base de datos de contactos, se construiría la consulta de la siguiente manera:

```
import android.provider.Contacts.People;
import android.content.ContentUris;
import android.net.Uri;
import android.database.Cursor;

// Utilizar el método ContentUris para conseguir la base del URI para los contactos con _ID==23.
Uri myPerson = ContentUris.withAppendedId(People.CONTENT_URI, 23);

// Como alternativa, se puede utilizar el método Uri para conseguir la base del URI.
// Lleva un string en vez de un integer.
Uri myPerson = Uri.withAppendedPath(People.CONTENT_URI, "23");

// Se consulta por ese registro específico:
Cursor cur = managedQuery(myPerson, null, null, null, null);
```

Los otros argumentos de los métodos `query()` y `managedQuery()` delimitan la consulta. Estos son:

- Los nombres de las columnas a devolver. Un valor null devuelve todas las columnas. De lo contrario, sólo se devuelven las columnas que son listadas por su nombre. Todos los content providers que vienen con la plataforma definen constantes para sus columnas. Ejemplo; la clase `android.provider.Contacts.Phones` define constantes para los nombres de las columnas en la tabla de los teléfonos que hemos visto anteriormente — `_ID`, `NUMBER`, `NUMBER_KEY`, `NAME`, etc.
- Un filtro detallando que registros hay que devolver, una cláusula `WHERE` con formato de SQL (excluyendo el `WHERE` mismo). El valor null devuelve todas las filas (a no ser que el URI limite la consulta a un sólo registro).
- Argumentos de selección.
- Un orden clasificadorio para las filas devueltas, una cláusula `ORDER BY` con formato de SQL

(excluyendo el ORDER BY mismo). El valor null devuelve los registros en el orden por defecto de la tabla, que puede estar desordenada. Veamos un ejemplo de consulta que devuelve una lista de nombres de contactos y sus números de teléfonos principales:

```
import android.provider.Contacts.People;
import android.database.Cursor;
// Crear un array especificando las columnas a devolver.
String[] projection = new String[] {
    People._ID,
    People._COUNT,
    People.NAME,
    People.NUMBER
};

// Obtener la base del URI para la tabla People en el content provider Contacts.
Uri contacts = People.CONTENT_URI;

// Hacer la consulta.
Cursor managedCursor = managedQuery(contacts,
    projection, // qué columnas hay que devolver
    null,       // qué filas hay que devolver (todas)
    null,       // argumentos de selección (ninguno)
    // Ordenar los resultados de manera ascendente por su nombre
    People.NAME + " ASC");
```

Esta consulta recupera datos de la tabla People del content provider Contacts. Recupera, el nombre, el número de teléfono principal y el ID unívoco de cada contacto. También informa sobre el número de registros devueltos a través del campo `_COUNT` de cada registro.

Las constantes de los nombres de las columnas están definidas en varias interfaces `_ID` y `_COUNT` en BaseColumns, `NAME` en PeopleColumns, y `NUMBER` en PhoneColumns. La clase Contacts.People implementa cada una de estas interfaces, por lo que el ejemplo de código que acabamos de ver se podría referir a ellas usando simplemente el nombre de la clase.

Lo que devuelve una consulta

Una consulta devuelve un conjunto de registros del cero en adelante. Los nombres de las columnas, el orden por defecto, y los tipos de datos son específicos de cada content provider. Cada provider tiene una columna `_ID`, que guarda un ID numérico unívoco para cada registro. Cada provider puede informar sobre el número de registros devueltos a través de la columna `_COUNT`; su valor es el mismo para todos los registros.

A continuación tenemos el conjunto de resultados de la consulta de la sección anterior:

<code>_ID</code>	<code>_COUNT</code>	<code>NAME</code>	<code>NUMBER</code>
44	3	A. Dylan	212 555 1234
13	3	B. Vain	425 555 6677
53	3	Rex Cars	201 555 4433

Los datos recuperados son expuestos por un objeto Cursor que puede ser usado para recorrer el conjunto de resultados hacia adelante o hacia atrás. Sólo se puede utilizar este objeto para leer los datos. Para añadir, modificar, o eliminar los datos, se debe utilizar el objeto ContentResolver.

Leer los datos recuperados

El objeto Cursor devuelto por la consulta permite el acceso a un recordset de resultados. Si se hace

la consulta por un registro específico mediante el ID, este conjunto contendrá solamente un valor. Si no, puede contener varios valores. (Si no existen coincidencias, puede estar vacío.)

Se pueden leer datos de campos específicos del registro, pero hay que conocer el tipo de dato del campo, ya que el objeto `Cursor` tiene un método diferente para leer cada tipo de dato — como `getString()`, `getInt()`, y `getFloat()`. (Sin embargo, para la mayoría de tipos, si se llama al método que lee strings, el objeto `Cursor` devolverá la representación de los datos mediante un `String`.) El `Cursor` te permite pedir la columna de los nombres a partir del índice de la columna, o el número del índice de la columna de los nombres.

El siguiente código muestra como leer nombres y números de teléfono de la consulta que hemos visto anteriormente:

```
import android.provider.Contacts.People;

private void getColumnData(Cursor cur){
    if (cur.moveToFirst()) {

        String name;
        String phoneNumber;
        int nameColumn = cur.getColumnIndex(People.NAME);
        int phoneColumn = cur.getColumnIndex(People.NUMBER);
        String imagePath;

        do {
            // Obtener los valores del campo
            name = cur.getString(nameColumn);
            phoneNumber = cur.getString(phoneColumn);

            // Hacer algo con los valores
            ...
        } while (cur.moveToNext());
    }
}
```

En general, las pequeñas cantidades de datos (desde 20 a 50k o menos) son directamente metidas en la tabla y se pueden leer llamando al método `Cursor.getBlob()`. Devuelve un array de bytes.

Si la entrada a la tabla es un content: URI, no se debería intentar abrir y leer el archivo directamente (puede fallar por problemas con los permisos). En vez de esto, se llama al método `ContentResolver.openInputStream()` para obtener un objeto `InputStream` que puedes utilizar para leer los datos.

Modificar los datos

Los datos guardados por un content provider se pueden modificar a través de:

- Añadir nuevos registros
- Añadir nuevos valores a los registros existentes
- Actualización de registros ya existentes a través de un Batch
- Borrar registros

Todas las modificaciones de los datos son realizadas utilizando los métodos `ContentResolver`. Algunos content providers requieren permisos más restrictivos para escribir datos de los que necesitan para leerlos. Si no se tiene permisos para escribir a un content provider, los métodos `ContentResolver` fallarán.

Añadir registros

Para añadir un nuevo registro a un content provider, primero hay que establecer un "Map" (colección) de parejas key-value en el objeto `ContentValues`, donde cada key coincide con el

nombre de la columna en el content provider y el valor es el valor deseado para el nuevo registro en esa columna. Después hay que llamar al método `ContentResolver.insert()` y pasarle el URI del provider y el map `ContentValues`. Este método devuelve el URI completo del nuevo registro — esto es, el URI del provider, con el ID añadido para el nuevo registro. En este momento se puede utilizar el URI para hacer una consulta y obtener el Cursor del nuevo registro, para después modificar el registro. A continuación tenemos un ejemplo:

```
import android.provider.Contacts.People;
import android.content.ContentResolver;
import android.content.ContentValues;

ContentValues values = new ContentValues();

// Añadir Abraham Lincoln a los contactos y hacerle favorito.
values.put(People.NAME, "Abraham Lincoln");
// 1 = el nuevo contacto se añade a favoritos
// 0 = el nuevo contacto no es añadido a favoritos
values.put(People.STARRED, 1);

Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
```

Añadir nuevos valores

Una vez que el registro existe, se le puede añadir nueva información o modificar la información existente. En el ejemplo anterior el siguiente paso sería añadir nueva información al nuevo contacto — como un número de teléfono o una dirección de correo electrónico.

La mejor manera de añadir información a un registro en la base de datos Contacts es añadir el nombre de la tabla al URI donde los nuevos datos acuden para buscar el registro, y utilizar el URI modificado para añadir los nuevos valores. Cada tabla de contactos tiene una constante `CONTENT_DIRECTORY` con este propósito. El siguiente código continúa con el ejemplo anterior, añadiéndole un número de teléfono y una dirección de correo electrónico para el registro recién creado:

```
Uri phoneUri = null;
Uri emailUri = null;

// Añadir un número de teléfono para Abraham Lincoln. Empezar con el URI para
// el nuevo registro que acaba de devolver el método insert(); acaba con el _ID
// del nuevo registro, por lo que no se tiene que añadir el ID.
// Añadir el nombre de la tabla de teléfonos a este URI,
// y utilizar el URI resultante para insertar el número de teléfono.
phoneUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);

values.clear();
values.put(People.Phones.TYPE, People.Phones.TYPE_MOBILE);
values.put(People.Phones.NUMBER, "1233214567");
getContentResolver().insert(phoneUri, values);

// Añadir una dirección de correo electrónico de la misma manera.
emailUri = Uri.withAppendedPath(uri, People.ContactMethods.CONTENT_DIRECTORY);

values.clear();
// ContactMethods.KIND se utiliza para distinguir diferentes tipos de
// métodos de contacto, como el email, IM, etc.
values.put(People.ContactMethods.KIND, Contacts.KIND_EMAIL);
values.put(People.ContactMethods.DATA, "test@example.com");
values.put(People.ContactMethods.TYPE, People.ContactMethods.TYPE_HOME);
getContentResolver().insert(emailUri, values);
```

Se pueden colocar pequeñas cantidades de datos binarios en una tabla llamando a la versión del método `ContentValues.put()` que recibe un array de bytes. Esto funciona para un audio clip corto o un icono. Sin embargo, si se tiene que añadir una gran cantidad de datos binarios, como una fotografía o una canción completa, hay que poner un content: URI para los datos de la tabla y llamar al método `ContentResolver.openOutputStream()` con el URI del archivo. (Esto hace que el content provider almacene los datos en un archivo y registre la ruta del archivo en un campo hidden del registro.)

Para esto, el content provider `MediaStore`, que es el provider principal que reparte imágenes, audio y datos de video, utiliza una convención especial: El mismo URI utilizado con `query()` o `managedQuery()` para obtener la meta-información de los datos binarios (como la captura de una fotografía o la fecha en la que se hizo) se utiliza con `openInputStream()` para obtener los propios datos. De la misma manera, el mismo URI que se utiliza con `insert()` para meter la meta-información en un registro `MediaStore` se utiliza con `openOutputStream()` para colocar allí los datos binarios.

El siguiente código ilustra esta convención:

```
import android.provider.MediaStore.Images.Media;
import android.content.ContentValues;
import java.io.OutputStream;

// Guardar el nombre y la descripción de la imagen en un map ContentValues.
ContentValues values = new ContentValues(3);
values.put(Media.DISPLAY_NAME, "road_trip_1");
values.put(Media.DESCRPTION, "Day 1, trip to Los Angeles");
values.put(Media.MIME_TYPE, "image/jpeg");

// Añadir un nuevo registro sin el bitmap, pero con los valores ya seteados.
// insert() devuelve el URI del nuevo registro.
Uri uri = getContentResolver().insert(Media.EXTERNAL_CONTENT_URI, values);

// Obtener un handle al nuevo archivo para ese registro, y guardar los datos en él.
// Aquí, el sourceBitmap es un objeto Bitmap que representa un archivo a guardar en la base de datos.
try {
    OutputStream outputStream = getContentResolver().openOutputStream(uri);
    sourceBitmap.compress(Bitmap.CompressFormat.JPEG, 50, outputStream);
    outputStream.close();
} catch (Exception e) {
    Log.e(TAG, "excepción mientras se escribe la imagen", e);
}
```

Actualización de registros mediante Batch

Para actualizar un grupo de registros mediante un batch (por ejemplo, cambiar "NY" a "New York" en todos los campos), llamar al método `ContentResolver.update()` con los valores y columnas que se quiere cambiar.

Borrar un registro

Para borrar un solo registro, llamar al método `ContentResolver.delete()` con el URI de una fila específica.

Para borrar varias filas, llamar al método `ContentResolver.delete()` con el URI del tipo del registro a borrar (por ejemplo, `android.provider.Contacts.People.CONTENT_URI`) y una cláusula SQL WHERE definiendo que filas borrar. (Precaución: Hay que incluir una cláusula WHERE válida si se está borrando un tipo general ya que si no se corre el riesgo de borrar más registros de los que se quiere!).

Crear un Content Provider

Para crear un content provider, se debe:

- Configurar un sistema para almacenar los datos. La mayoría de los content providers almacenan los datos utilizando los métodos de almacenaje de los archivos Android o base de datos SQLite, pero se puede almacenar los datos de la manera en que se quiera. Android proporciona la clase SQLiteOpenHelper para ayudarnos a crear una base de datos y un SQLiteDatabase para gestionarla.
- Extender la clase ContentProvider para proporcionar el acceso a los datos.
- Declarar el content provider en el archivo manifest de la aplicación. (AndroidManifest.xml).

A continuación describimos como hacer los dos últimos apartados;

Extender la clase ContentProvider

Se define una subclase ContentProvider para encapsular los datos. Se implementan seis métodos abstractos declarados en la clase ContentProvider:

```
query()
insert()
update()
delete()
getType()
onCreate()
```

El método query() debe devolver un objeto Cursor que itera sobre los datos requeridos. El propio Cursor es una interface, pero Android proporciona objetos Cursor ya preparados que se pueden usar. SQLiteCursor, por ejemplo, puede iterar sobre los datos almacenados en una base de datos SQLite. Se obtiene el objeto Cursor llamando a cualquiera de los métodos query() de la clase SQLiteDatabase. Existen otras implementaciones de Cursor — como el MatrixCursor — para datos no almacenados en una base de datos.

Ya que estos métodos ContentProvider pueden ser llamados desde varios objetos ContentResolver en diferentes procesos e hilos, se deben implementar usando "thread-safety" (seguridad en hilos).

Como cortesía, también se puede llamar al método ContentResolver.notifyChange() para notificar a los listeners cuando hay modificaciones de datos.

Aparte de definir la subclase, existen otros pasos que se deben de tomar para simplificar el trabajo de los clientes y hacer la clase más accesible:

- Definir un public static final Uri llamado CONTENT_URI. Este es el string que representa el content: URI completo que maneja el content provider. Se debe definir un string único para este valor. La mejor solución es utilizar el nombre completo de la clase del content provider (en minúsculas). Así, por ejemplo, el URI de una clase TransportationProvider puede ser definida de la siguiente manera:

```
public static final Uri CONTENT_URI =
    Uri.parse("content://com.example.codelab.transportationprovider");
```

Si el provider tiene subtablas, hay que definir también constantes CONTENT_URI para cada una de las subtablas. Estos URIs deben de tener todos la misma autoridad (ya que esto identifica al content provider) y ser diferenciados solamente por sus rutas. Por ejemplo:

```
content://com.example.codelab.transportationprovider/train
content://com.example.codelab.transportationprovider/air/domestic
content://com.example.codelab.transportationprovider/air/international
```

Para una perspectiva general de content: URIs, ver el "Resumen del Content URI" al final de este documento.

- Definir el nombre de las columnas que serán devueltas al cliente por el content provider. Si se está utilizando una base de datos subyacente, estos nombres de columnas son idénticas a los nombres de las columnas de la base de datos SQL que representan. También se definen las constantes `string public static` que los clientes pueden utilizar para especificar las columnas tanto en las consultas como en otras instrucciones.

Hay que incluir un columna integer llamada `"_id"` (con la constante `_ID`) para los IDs de los registros. Este campo debe de existir tanto si se tiene o no otro campo unívoco (como el URL). Si se está utilizando la base de datos SQLite el campo `_ID` debe de ser del tipo:

`INTEGER PRIMARY KEY AUTOINCREMENT`

El descriptor `AUTOINCREMENT` es opcional. Pero sin él, SQLite incrementa el campo contador ID en uno con respecto al mayor número existente en la columna. Si se borra la última fila, la siguiente fila añadida tendrá el mismo ID que la fila borrada. `AUTOINCREMENT` evita esto haciendo que SQLite tome el siguiente valor más alto se borre o no.

- Documentar cuidadosamente el tipo de datos de cada columna. Los clientes necesitan esta información para leer los datos.

- Si se maneja un nuevo tipo de datos, se debe definir un nuevo tipo de MIME para devolver en la implementación del `ContentProvider.getType()`. El tipo depende en parte de si el `content: URI` enviado al `getType()` limita la petición o no a un registro específico. Existe un formato tipo de MIME para un registro único y otro para múltiples registros. Hay que utilizar los métodos `Uri` para determinar que se está pidiendo. A continuación tenemos los formatos generales para cada tipo:

- Para un sólo registro:

`vnd.android.cursor.item/vnd.yourcompanyname.contenttype`

Como por ejemplo, una petición para el registro 122 del tren, como este URI,

`content://com.example.transportationprovider/trains/122`

puede devolver este tipo de MIME:

`vnd.android.cursor.item/vnd.example.rail`

- Para registros múltiples:

`vnd.android.cursor.dir/vnd.yourcompanyname.contenttype`

Como por ejemplo, una petición de todos los registros del tren, como el siguiente URI,

`content://com.example.transportationprovider/trains`

puede devolver este tipo de MIME:

`vnd.android.cursor.dir/vnd.example.rail`

- Si se revelan datos en bytes que son demasiado grandes para poner en una tabla — tal y como un archivo bitmap grande — el campo que expone esos datos a los clientes debería contener un `string content: URI`. Este es el campo que da el acceso al archivo con los datos a los clientes. El registro también debería tener otro campo, `"_data"` que lista la ruta exacta del archivo en el dispositivo. No se espera que el cliente lea este campo, si no el `ContentResolver`. El cliente llamará al `ContentResolver.openInputStream()` del campo de cara al usuario que contiene el URI para ese item. El `ContentResolver` pedirá el campo `"_data"` para ese registro, y dado que tiene más permisos que el cliente, debería poder acceder directamente a ese archivo y devolver al cliente un wrapper de lectura para ese archivo.

Declarar el content provider

Para informar al sistema Android sobre el content provider desarrollado, hay que declararlo con un

elemento<provider> en el archivo AndroidManifest.xml de la aplicación. Los content providers que no están declarados en el manifest no son visibles para el sistema Android.

El atributo name es el nombre cualificado de la subclase ContentProvider. El atributo authorities es la parte del content: URI que identifica al provider. Ejemplo: si la subclase ContentProvider es AutoInfoProvider, el elemento<provider> se verá de la siguiente manera:

```
<provider android:name="com.example.autos.AutoInfoProvider"
          android:authorities="com.example.autos.autoinfoprovider"
          .../>
</provider>
```

Nótese que el atributo authorities omite la parte de la ruta del content: URI. Ejemplo: Si AutoInfoProvider controlara subtablas para diferentes tipos de autos o diferentes fabricantes,

```
content://com.example.autos.autoinfoprovider/honda
content://com.example.autos.autoinfoprovider/gm/compact
content://com.example.autos.autoinfoprovider/gm/suv
```


estas rutas no serían declaradas en el manifest. El atributo authorities es lo que identifica al provider, no a la ruta; el provider puede interpretar la parte de la ruta en el URI de cualquier forma que se quiera.

Otros atributos <provider> pueden configurar permisos para la lectura y escritura de datos, pueden hacer que un icono y un texto que se muestran a los usuarios, habiliten y deshabiliten el provider, etc. Si se setea el atributo multiprocess a "true" los datos no necesitan ser sincronizados entre las múltiples versiones ejecutadas del content provider y esto permite crear una instancia del provider en cada proceso del cliente, eliminando la necesidad de llevar a cabo IPC.

Resumen del Content URI

A continuación tenemos una recapitulación de las partes más importantes de un content URI:

content://com.example.transportationprovider/trains/122



A.Prefijos estándar que indican que los datos son controlados por un content provider. No se modifican nunca.

B.Es el atributo authorities, que identifica al content provider. Para terceras aplicaciones, tiene que ser un nombre de una clase completamente calificada (en minúscula) para asegurarse la unicidad. El atributo authorities se declara en el atributo authorities del elemento <provider>:

```
<provider android:name=".TransportationProvider"
          android:authorities="com.example.transportationprovider"
          ... >
```

C.La ruta que utiliza el content provider para determinar el tipo de datos pedidos. Puede tener una longitud de cero en adelante. Si el content provider expone solamente un tipo de datos (sólo trenes, por ejemplo), será cero. Si el content provider expone varios tipos de datos, incluyendo subtipos, puede tener una longitud de varios segmentos — como por ejemplo, "land/bus", "land/train", "sea/ship", y "sea/submarine" para dar cuatro posibilidades.

D.El ID del registro específico pedido (si es que se ha pedido alguno). Este es el valor del _ID del registro pedido. Si la petición no está limitada a un sólo registro, este segmento y la barra final se omiten:

```
content://com.example.transportationprovider/trains
```

Intents

Tres de los componentes de una aplicación — actividades, servicios, y los broadcast receivers — son activados a través de mensajes, llamados intents.

Los mensajes Intent son una estructura para la vinculación tardía en tiempo de ejecución entre componentes de una misma aplicación o de aplicaciones diferentes. El propio objeto Intent, es una estructura pasiva de datos que contiene una descripción abstracta de una operación a realizar — o, en el caso de los broadcasts, una descripción de algo que ha ocurrido y está siendo anunciado.

Existen mecanismos diferentes para entregar los intents a cada tipo de componente:

- Un objeto Intent es pasado a Context.startActivity() o a Activity.startActivityForResult() para lanzar una actividad o para que una actividad ya existente haga algo nuevo. (También puede ser pasado a Activity.setResult() para devolver información a la actividad que llamó al startActivityForResult().)
- Un objeto Intent es pasado a Context.startService() para instanciar un servicio o entregar nuevas instrucciones a un servicio en curso. De forma similar, un intent se puede pasar a un Context.bindService() para establecer una conexión entre el componente que llama y el servicio objetivo. Puede, opcionalmente, iniciar un servicio que no se esté ejecutando.
- Los objetos intent pasados a cualquiera de los métodos broadcast (como por ejemplo Context.sendBroadcast(), Context.sendOrderedBroadcast(), o Context.sendStickyBroadcast()) son entregados a todos los broadcast receivers que estén interesados.

En cada caso, el sistema Android encuentra la actividad, servicio o broadcast receiver apropiado para responder al intent, instanciándolo si hiciera falta. No hay solapamiento entre estos sistemas de mensajería: los broadcast intents sólo son entregados a los broadcast receivers, nunca a las actividades ni a los servicios. Un intent pasado al método startActivity() se entrega solamente a una actividad, nunca a un servicio ni a un broadcast receiver, etc.

Objetos Intent

Un objeto Intent es un paquete de información. Contiene información de interés para el componente que recibe el intent, (como la acción a realizar y los datos afectados) además de la información de interés para el sistema Android (como la categoría del componente que debería manejar el intent y las instrucciones de como lanzar una actividad objetivo).

Puede contener los siguiente:

Nombre del componente

El nombre del componente que debe manejar el intent. Este campo es un objeto ComponentName— una combinación de un nombre de una clase del componente objetivo (por ejemplo "com.example.project.app.FreneticActivity") y el nombre del paquete seteado en el archivo manifest de la aplicación en donde está el componente (por ejemplo, "com.example.project"). La parte del paquete del nombre del componente y el nombre del paquete seteado en el manifest no tienen porque coincidir.

El nombre del componente es opcional. Si está seteado, el objeto Intent se entrega a una instancia de la clase designada. Si no está seteado, Android utiliza otra información del objeto Intent para localizar un objetivo adecuado.

El nombre del componente está seteado por el método setComponent(), setClass(), o setClassName() y leído por getComponent().

Action (Acción)

Un string que da nombre a la acción que se va a realizar — o, en el caso de los broadcast intents, la acción que ya ha tenido lugar y está siendo reportada.

La clase Intent define un número de constantes de acción, incluyendo las siguientes:

Constante	Componente Objetivo	Action (Acción)
ACTION_CALL	actividad	Iniciar una llamada de teléfono.
ACTION_EDIT	actividad	Mostrar datos para que el usuario los edite.
ACTION_MAIN	actividad	Se ejecuta como la actividad inicial de una tarea, sin input de datos ni output.
ACTION_SYNC	actividad	Sincronización de datos entre servidor y el dispositivo móvil.
ACTION_BATTERY_LOW	broadcast receiver	Un aviso de que la batería está baja.
ACTION_HEADSET_PLUG	broadcast receiver	Unos auriculares han sido insertados o sacados del dispositivo.
ACTION_SCREEN_ON	broadcast receiver	La pantalla ha sido encendida.
ACTION_TIMEZONE_CHANGED	broadcast receiver	La configuración del huso horario ha sido cambiada.

Ver la descripción de la clase [Intent](#) para una lista de constantes predefinidas para acciones genéricas. Las otras acciones se definen en el API de Android.

También se pueden definir los strings de las acciones para activar los componentes de la aplicación. Los que se inventen nuevos deben incluir como prefijo el paquete de la aplicación — por ejemplo: "com.example.project.SHOW_COLOR".

La acción determina como se estructura el resto del intent — particularmente los datos y los campos extras — igual que el nombre de un método determina un conjunto de argumentos y el valor que retorna. Por esta razón, es una buena idea utilizar nombres de acciones que sean muy específicos y emparejarlos a otros campos del intent. En otras palabras, en vez de definir una acción aislada, es bueno definir un protocolo entero para los objetos Intent que van a manejar los componentes que se crean.

La acción en un objeto Intent está seteado por el método [setAction\(\)](#) y leído por el método [getAction\(\)](#).

Datos

La URI de los datos sobre la que actuar y el tipo de MIME de esos datos. Diferentes acciones se emparejan con diferentes tipos de especificaciones de datos. Por ejemplo, si el campo de la acción es ACTION_EDIT, el campo de los datos contendría el URI del documento que sería mostrado para editar. Si la acción es ACTION_CALL, el campo de los datos sería un URI con el teléfono al que llamar. Si la acción es ACTION_VIEW y el campo de los datos es un http: URI, la actividad receptora sería llamada para descargar y mostrar los datos a los que se refiere el URI.

Cuando se machea un intent con un componente capaz de manejar los datos, es importante saber el tipo de dato (el tipo de MIME) además de su URI. Por ejemplo, si un componente es capaz de mostrar una imagen no debería ser llamado para ejecutar un archivo de audio.

En muchos casos, el tipo de datos puede ser inferido desde el URI— particularmente desde el content: URIs, que indica que los datos están localizados en el dispositivo y controlados por el content provider (ver la [información sobre content provider](#)).

Pero los tipos también pueden ser explícitamente seteados en el objeto Intent. El método `setData()` especifica los datos solo como un URI, el método `setType()` los especifica como un tipo de MIME, y `setDataAndType()` los especifica tanto como un URI como un tipo de MIME. El URI es leído por el método `getData()` y el tipo por el método `getType()`.

Categoría

Un string conteniendo información adicional sobre el tipo de componente que debería manejar el intent. Se pueden colocar en el objeto Intent el número de descripciones de categoría que sean necesarios. Como lo hace con las acciones, la clase Intent define varias categorías de constantes, incluyendo estas:

Constante	Significado
CATEGORY_BROWSABLE	La actividad objetivo puede ser llamada con seguridad por el navegador para mostrar datos referenciados por un link — por ejemplo, una imagen o un mensaje de correo electrónico.
CATEGORY_GADGET	La actividad puede ser embebida dentro de otra que sea un host de gadgets.
CATEGORY_HOME	La actividad muestra la pantalla de inicio, la primera pantalla que el usuario ve cuando el dispositivo se enciende o cuando se hace click en la tecla HOME.
CATEGORY_LAUNCHER	La actividad puede ser la actividad inicial de una tarea y está en el nivel más alto del lanzador de la aplicación.
CATEGORY_PREFERENCE	La actividad objetivo es un panel preferente.

Para conocer la lista completa de las categorías, ver la descripción de la clase `Intent`.

El método `addCategory()` coloca una categoría en un objeto Intent, `removeCategory()` elimina una categoría previamente añadida, y `getCategories()` obtiene el conjunto de todas las categorías que están en el objeto en ese momento.

Extras

Parejas key-value para información adicional que deberían ser entregadas al componente que maneja el intent. Igual que algunos actions están emparejados con tipos determinados de datos URIs, algunos están emparejados con unos extras determinados. Por ejemplo, un intent `ACTION_TIMEZONE_CHANGED` tiene un extra "time-zone" que identifica el nuevo huso horario, y `ACTION_HEADSET_PLUG` tiene un extra "state" que indica si los auriculares están insertados o no, así como un extra "name" para el tipo de auricular. Si se quisiera inventar una acción `SHOW_COLOR`, el valor del color, sería seteado en una pareja extra key-value.

El objeto Intent tiene una serie de métodos `put...()` que insertan varios tipos de datos extra y un conjunto de métodos `get...()` que leen los datos. Estos métodos son paralelos a los que tienen los objetos `Bundle`. De hecho, los extras se pueden instalar y leer como si fueran Bundle utilizando los métodos `putExtras()` y `getExtras()`.

Flags

Flags de varios tipos. Muchos ellos informan al sistema Android de como lanzar una actividad (por ejemplo, a qué tarea pertenece la actividad) y como tratarla después del lanzamiento (por ejemplo, si pertenece o no a la lista de actividades recientes). Todos estos flags están definidos en la clase Intent.

El sistema Android y las aplicaciones que vienen con la plataforma utilizan objetos Intent tanto para mandar broadcasts originados por el sistema como para activar a los componentes definidos por el sistema. Para ver como estructurar un intent para activar un componente del sistema, consultar la [lista de intents](#).

Cómo funciona un Intent Filter

Los Intents pueden ser divididos en dos grupos:

- Los intents explícitos designan el componente objetivo por su nombre (el nombre del campo del componente, del que hemos hablado previamente, tiene seteado un valor). Los intents explícitos se usan generalmente para mensajes internos de la aplicación — como una actividad que inicia un servicio subordinado o lanza una actividad hermana.
- Los intents implícitos no nombran un componente objetivo (el campo para el nombre del componente está en blanco). Los intents implícitos son utilizados frecuentemente para activar componentes en otras aplicaciones.

Android entrega un intent explícito a una instancia de la clase objetivo. Para determinar que componente debe obtener el intent sólo importa el nombre del componente.

Se necesita una estrategia diferente para los intents implícitos:

En ausencia de un objetivo designado, el sistema Android debe encontrar el mejor componente (o componentes) para manejar el intent — una sola actividad o servicio para realizar la acción pedida o el conjunto de broadcast receivers para responder al mensaje del broadcast. Utiliza los intent filters para comparar los contenidos del objeto Intent con las estructuras asociadas con componentes que potencialmente pueden recibir intents. Los filtros anuncian las capacidades del componente y delimitan los intents que pueden manejar. Le dan a los componentes la posibilidad de recibir intents implícitos del tipo anunciado. Si un componente no tiene ningún filtro intent (intent filters), solamente puede recibir intents explícitos. Un componente con filtros puede recibir tanto intents implícitos como explícitos.

Cuando un objeto es testeado con un intent filter, sólo se miran tres aspectos de un objeto Intent:

- acción
- datos (tanto URI como el tipo de datos)
- categoría

Los extras y los flags no intervienen en la decisión de que componente recibe un intent.

Intent filters (filtros intent)

Para informar al sistema que intents implícitos pueden manejar, las actividades, servicios y los broadcast receivers pueden tener uno o más intent filters. Cada filtro filtra los intents por el tipo deseado, dejando fuera los intents no deseados — pero sólo descarta los intents implícitos (aquellos que no nombran la clase objetivo). Un intent explícito siempre se entrega a su objetivo, sin importar lo que contenga; no se filtra. Pero en el caso de los intent implícitos, sólo se entregan a un componente si puede pasar a través de uno de los filtros de los componentes.

Un componente tiene filtros separados para cada trabajo que realiza, cada cara que presenta al usuario. Por ejemplo, la actividad Note Editor de la aplicación Note Pad tiene dos filtros — uno para iniciarse con una nota específica que el usuario puede ver o editar, y otra para iniciarse con una nota nueva en blanco que el usuario puede rellenar y guardar.

Un intent filter es una instancia de la clase [IntentFilter](#). Sin embargo, dado que el sistema Android debe conocer las capacidades del componente antes de lanzarlo, los intent filters no están en código java, si no en el archivo manifest de la aplicación (AndroidManifest.xml) como elementos [<intent-](#)

filter>. (La única excepción serían los filtros para los broadcast receivers que están registrados dinámicamente mediante la llamada al `Context.registerReceiver()`; se crean directamente como objetos `IntentFilter`.)

Un filtro tiene campos paralelos a los campos acción, datos y categoría de un objeto `Intent`. Un `intent` implícito se testea frente al filtro en esas tres áreas. Para ser entregado al componente dueño del filtro, debe pasar los tres tests. Si no pasa alguno de los tres tests, el sistema Android no lo entregará al componente — por lo menos no con las bases de ese filtro. Sin embargo, dado que un componente puede tener varios `intent filters`, un `intent` que no pasa por alguno de los filtros del componente puede pasar por otro.

A continuación se describen cada uno de los tres tests:

Test de acción

Un elemento `<intent-filter>` en el archivo `manifest` lista las acciones como subelementos `<action>`. Por ejemplo:

```
<intent-filter ... >
  <action android:name="com.example.project.SHOW_CURRENT" />
  <action android:name="com.example.project.SHOW_RECENT" />
  <action android:name="com.example.project.SHOW_PENDING" />
  ...
</intent-filter>
```

Como se muestra en el ejemplo, mientras que un objeto `Intent` nombra una sola acción, un filtro puede listar más de uno. La lista no puede estar vacía; un filtro debe contener al menos un elemento `<action>` element, ya que si no bloquea todos los `intents`.

Para pasar el test, la acción especificada en el objeto `Intent` debe coincidir con alguno de las acciones listadas en el filtro. Si el objeto o el filtro no especifican una acción, el resultado es el siguiente:

- Si el filtro no lista ninguna acción, el `intent` no coincidirá con nada, por lo que todos los `intents` fallarán el test. Ningún `intent` puede pasar el filtro.
- De otro lado, un objeto `Intent` que no especifica ninguna acción, pasará automáticamente el test — esto ocurrirá mientras que el filtro contenga al menos una acción.

Test de categoría

Un elemento `<intent-filter>` también lista las categorías como subelementos. Por ejemplo:

```
<intent-filter ... >
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  ...
</intent-filter>
```

Nótese que las constantes descritas anteriormente para acciones y categorías no se utilizan en el archivo `manifest`. Se utilizan los valores completos del string. Por ejemplo, el string `"android.intent.category.BROWSABLE"` del ejemplo anterior corresponde a la constante `CATEGORY_BROWSABLE` de la que hemos hablado con anterioridad en el apartado de OBJETOS INTENT. De forma parecida, el string `"android.intent.action.EDIT"` corresponde a la constante `ACTION_EDIT`.

Para que un `intent` pase el test de categoría, cada categoría del objeto `Intent` debe coincidir con una categoría en el filtro. El filtro puede listar categorías adicionales, pero no puede omitir

ninguna que esté en el intent.

Por lo tanto, un objeto Intent sin categorías debería siempre pasar el test, sin tener en cuenta lo que esté en el filtro. Esto pasa siempre pero con una excepción; Android trata a todos los intents implícitos pasados al método `startActivity()` como si contuvieran al menos una categoría: "android.intent.category.DEFAULT" (la constante `CATEGORY_DEFAULT`).

Por lo tanto, las actividades que quieran recibir intents implícitos deben incluir "android.intent.category.DEFAULT" en sus intents filters. (Filtros con configuraciones "android.intent.action.MAIN" y "android.intent.category.LAUNCHER" son la excepción. Marcan actividades que empiezan nuevas tareas y que están representadas en la pantalla de inicio. Pueden incluir "android.intent.category.DEFAULT" en la lista de categorías, pero no es necesario.)

Test de datos

Como ocurre con la acción y la categoría, la especificación de datos para un intent filter está contenido en un subelemento. Y como en esos casos, el subelemento puede aparecer varias veces, o ninguna. Por ejemplo:

```
<intent-filter . . . >
  <data android:mimeType="video/mpeg" android:scheme="http" . . . />
  <data android:mimeType="audio/mpeg" android:scheme="http" . . . />
  . . .
</intent-filter>
```

Cada elemento `<data>` puede especificar un URI y un tipo de datos (tipo media MIME). Existen atributos diferentes — scheme, host, port, y path — para cada parte del URI:

scheme://host:port/path

For example, in the following URI,

content://com.example.project:200/folder/subfolder/etc

el scheme es el "content", el host es "com.example.project", el port es "200", y la ruta es "folder/subfolder/etc". El host y el port juntos constituyen el URI authority; si no se especifica un host, el port es ignorado.

Cada uno de estos atributos es opcional, pero no son independientes unos de otros. Para que una autoridad tenga significado, se debe especificar también un scheme. Para que una ruta tenga sentido, tanto el scheme como la autoridad deben ser especificados.

Cuando el URI en un objeto Intent se compara con una especificación URI en un filtro, se compara sólo con las partes del URI mencionados en el filtro. Por ejemplo, si un filtro especifica solo un scheme, todas las URIs con ese scheme machean con el filtro. Si un filtro especifica un scheme y una autoridad pero no una ruta, todos los URIs con el mismo scheme y la misma autoridad machearán, sin tener en cuenta sus rutas. Si un filtro especifica un scheme, una autoridad y una ruta, sólo machearán con los URIs con el mismo scheme, autoridad y ruta. Sin embargo, una especificación en la ruta en el filtro puede contener comodines y así no necesitar más que una coincidencia parcial de la ruta.

El atributo type de un elemento `<data>` especifica el tipo MIME de los datos. Es más común en filtros que en URI. Tanto el objeto Intent como el filtro pueden usar un comodín "*" para el campo subtipo — por ejemplo, "text/*" o "audio/*" — indicando cualquier coincidencia con los subtipos.

El test de los datos compara tanto el URI como el tipo de dato en el objeto Intent con el URI y tipo de dato especificados en el filtro. Las reglas son las siguientes:

- Un objeto intent que no contiene ni el URI ni un tipo de dato sólo pasará el test si el filtro no especifica, tampoco, ningún URI o tipo de datos.
- Un objeto Intent que contiene un URI pero no contiene tipos de datos (y un tipo no puede ser inferido desde el URI) pasará el test sólo si su URI coincide con un URI en el filtro y el filtro

asimismo, no especifica el tipo. Este será el caso de URIs como mailto: y tel: que no hacen referencia a los datos actuales.

- c. Un objeto Intent que contiene un tipo de dato pero no un URI pasará el test sólo si el filtro lista los mismos tipos de datos y si no especifica un URI.
- d. Un objeto intent que contiene tanto un URI como un tipo de dato (o un tipo de dato inferido del URI) pasará la parte del test sobre los tipos de datos, sólo si el tipo coincide con un tipo listado en el filtro. Pasará la parte del test del URI bien si el URI coincide con un URI en el filtro o si tiene un content: o un file: URI y el filtro no especifica un URI. En otras palabras, un componente soportará datos content: y file: si su filtro lista solo un tipo de dato.

Si un intent puede pasar a través de los filtros de más de una actividad o servicio, se preguntará al usuario que componente activar. Se lanzará una excepción si no se encuentra un objetivo.

Nota; no se puede confiar en un intent filter en cuanto a seguridad se refiere. Mientras controla que un componente reciba solamente ciertos tipos de intents implícitos, no evita que los intents explícitos tengan al componente como objetivo. A pesar de que un filtro restringe los intents que un componente va a manejar a ciertas acciones y fuentes de datos, alguien podría juntar un intent explícito con una acción y fuente de datos diferentes y poner al componente como objetivo.

Casos más comunes

La última regla mostrada anteriormente para el test de datos, regla (d), refleja la expectativa de que los componentes pueden obtener datos locales desde un archivo o content provider. Por lo tanto, sus filtros pueden listar solamente un tipo de dato y no necesitan nombrar explícitamente los schemes content: y file:. Este es un caso típico. Un elemento <data> como el siguiente, por ejemplo, le indica a Android que el componente puede conseguir datos en formato imagen de un content provider y mostrarlo:

```
<data android:mimeType="image/*" />
```

Dado que la mayoría de los datos disponibles son dispensados por los content providers, los filtros que especifican un tipo de dato pero no un URI, son los más comunes.

Otra configuración común son los filtros con un scheme y un tipo de dato. Por ejemplo, un elemento <data> como el siguiente, indica a Android que el componente puede obtener datos de video desde la red y mostrarlo:

```
<data android:scheme="http" android:type="video/*" />
```

Ver, por ejemplo, qué hace la aplicación del navegador cuando el usuario navega siguiendo un link de una página web. Primero trata de mostrar los datos (como ocurriría si el link estuviera en una página HTML). Si no puede mostrar los datos, junta el intent implícito con el scheme y el tipo de dato e intenta iniciar una actividad que pueda realizar el trabajo. Si no existe, pide al gestor de descargas que descargue los datos. Eso lo pone bajo el control de un content provider, por lo que potencialmente puede responder un pool más amplio de actividades (aquellas con filtros que sólo nombran el tipo de dato).

La mayoría de las aplicaciones, tienen una forma de empezar de cero, sin referencias a ningún dato en particular. Las actividades que pueden iniciar aplicaciones tienen filtros con un "android.intent.action.MAIN" especificado como la acción. Para ser representados en el lanzador de la aplicación, especifican la categoría "android.intent.category.LAUNCHER":

```
<intent-filter . . . >
  <action android:name="code android.intent.action.MAIN" />
  <category android:name="code android.intent.category.LAUNCHER" />
</intent-filter>
```

Utilizar mapeo con intents

Los Intents son mapeados con los intent filters no sólo para descubrir un componente objetivo a activar, pero también para obtener información sobre el conjunto de componentes del dispositivo. Por ejemplo, el sistema Android rellena el lanzador de la aplicación, la primera pantalla que muestra las aplicaciones que están disponibles para ser lanzadas por el usuario, encontrando todas las actividades con intent filters que especifican la acción "android.intent.action.MAIN" y la categoría "android.intent.category.LAUNCHER" (como descrito en la sección anterior). En ese momento muestra los iconos y etiquetas de las actividades que están en el lanzador. De manera similar, descubre la pantalla inicio, buscando la actividad con "android.intent.category.HOME" en su filtro.

La aplicación puede utilizar el mapeo con los intent de manera similar. El PackageManager tiene un conjunto de métodos query...() que devuelven todos los componentes que pueden aceptar un intent determinado, y una serie de métodos similares de resolve...() que determinan que componente responderá mejor al intent. Por ejemplo, el método queryIntentActivities() devuelve un listado de todas las actividades que pueden realizar el intent pasado como argumento, y el método queryIntentServices() devuelve un listado parecido de servicios. Ninguno de los métodos activa componentes; sólo listan los que si que pueden responder. Existe un método similar para los broadcast receivers; queryBroadcastReceivers()

Procesos e Hilos en Android

Cuando un componente de una aplicación se inicia y la aplicación no tiene otros componentes ejecutándose, el sistema Android inicia un nuevo proceso con un solo hilo de ejecución. Por defecto, todos los componentes de la misma aplicación se ejecutan en el mismo proceso e hilo (llamado hilo "main"). Si un componente de una aplicación se inicia y ya existe un proceso para esa aplicación (ya que otro componente de la aplicación existe), se inicia el componente dentro de ese proceso y utiliza el mismo hilo de ejecución. Sin embargo, se puede hacer que los diferentes componentes de la aplicación se ejecuten en diferentes procesos y crear hilos adicionales para cualquier proceso.

Este documento detalla como trabajan los procesos y los hilos en una aplicación Android.

Procesos

Por defecto, todos los componentes de la misma aplicación se ejecutan en el mismo proceso y la mayoría de las aplicaciones no deberían cambiarlo. Sin embargo, si se quiere controlar a qué proceso pertenece un componente, se puede hacer en el archivo manifest.

La anotación en el manifest para cada tipo de elemento del componente—<activity>, <service>, <receiver>, y <provider>—soporta un atributo `android:process` que puede especificar el proceso en el que ese componente debería ejecutarse. Se puede setear este atributo para que el componente se ejecute en su propio proceso o para que algunos componentes compartan un proceso mientras otros no lo hacen. También se puede setear `android:process` para que componentes de diferentes aplicaciones se ejecuten en el mismo proceso—siempre que las aplicaciones compartan el mismo ID de usuario Linux y firmen con los mismos certificados.

El elemento <application> también soporta un atributo `android:process`, para setear un valor por defecto que aplica a todos los componentes.

En un momento determinado, Android puede decidir terminar un proceso. Esto puede pasar, si por ejemplo, la memoria es baja y se necesita para otros procesos más inmediatos para el usuario. Cuando se termina un proceso, los componentes de la aplicación que se están ejecutando en él, se eliminan. El proceso se reinicia para esos componentes cuando hay trabajo para ellos.

Cuando se decide que procesos hay que terminar, el sistema Android sopesa la importancia relativa de cada uno para el usuario. Por ejemplo, terminará antes un proceso que hostea actividades que ya no son visibles en la pantalla, que un proceso que hostea actividades visibles. La decisión de si

terminar un proceso o no, depende del estado de los componentes que se están ejecutando en ese proceso. A continuación se detallan las reglas que se usan para decidir que procesos se terminan:

Ciclo de vida del proceso

El sistema Android intenta mantener el proceso de una aplicación el máximo de tiempo posible, pero eventualmente necesita eliminar procesos para recuperar memoria para procesos nuevos o más importantes. Para determinar cuales mantener y cuales eliminar, el sistema coloca cada proceso en una "jerarquía de importancia" basada en los componentes que se están ejecutando en el proceso y el estado de esos componentes. Primero se eliminan los procesos que tiene menos importancia, después los siguientes en importancia y así hasta que sea necesario para recuperar los recursos del sistema.

Existen cinco niveles en la jerarquía. La siguiente lista nos detalla los diferentes tipos de procesos en orden de importancia (el primer proceso es el más importante y es el que se termina en último lugar):

1. Proceso en primer plano

Un proceso requerido para lo que el usuario está haciendo en ese momento. Un proceso se considera que está en primer plano si cualquiera de las siguientes condiciones se cumplen:

- Hostea un Activity con la que el usuario está interactuando (se ha llamado al método onResume() del Activity).
- Hostea a un Service que está vinculado a la actividad con la que el usuario está interactuando.
- Hostea un Service que se está ejecutando "en primer plano"—el servicio ha llamado al método startForeground().
- Hostea a un Service que está ejecutando a uno de sus métodos callback de su ciclo de vida (onCreate(), onStart(), o onDestroy()).
- Hostea a un BroadcastReceiver que está ejecutando su método onReceive().

Generalmente, sólo existen unos pocos procesos en primer plano en un tiempo determinado. Son finalizados en último término, cuando la memoria es tan baja que no se puede seguir con su ejecución. Finalizar algunos de estos procesos es necesario para que la interfaz de usuario continúe respondiendo.

2. Procesos visibles

Un proceso que no tiene ningún componente en primer plano, pero que todavía puede afectar a lo que el usuario ve en la pantalla. Se considera un proceso visible si cualquiera de las siguientes condiciones se cumple:

- Hostea un Activity que no está en primer plano, pero todavía es visible al usuario (su método onPause() ha sido llamado). Esto puede ocurrir, por ejemplo, si la actividad en primer plano ha iniciado un diálogo, lo que permite que se vea la actividad anterior.
- Hostea un Service que está vinculado a una actividad visible (o en primer plano).

Un proceso visible está considerado extremadamente importante y no será terminado a no ser que sea necesario para que los procesos en primer plano puedan continuar ejecutándose.

3. Proceso de Servicio

Un proceso que está ejecutando un servicio que ha sido iniciado con el método startService() y que no se puede catalogar en ninguna de las otras dos categorías anteriores. A pesar de que los procesos de servicio no están directamente unidos a nada que el usuario ve, están generalmente haciendo cosas importantes para él (como reproducir música o descargar data de la red), por lo que el sistema los mantiene en ejecución a no ser que no haya suficiente memoria para mantenerlos junto con los procesos visibles y los que están en primer plano.

4. Procesos en segundo plano

Un proceso que contiene una actividad que no es visible en ese momento para el usuario (el método onStop() de la actividad ha sido llamado). Estos procesos no tienen impacto directo en el

usuario, y el sistema los puede terminar en cualquier momento para recuperar memoria para un proceso de servicio, visible o de primer plano. Normalmente existen varios procesos de segundo plano en ejecución, por lo que se guardan en una lista LRU (least recently used-el menos usado recientemente) para asegurar que el proceso con la última actividad que ha visto el usuario, sea la última en ser terminada. Si una actividad implementa sus métodos de ciclo de vida correctamente, y guarda su estado en curso, cuando se termine su proceso no existirá un efecto visible en el usuario, ya que cuando el usuario navegue de vuelta a la actividad, la actividad restaura todo su estado visible. Ver la documentación sobre [Activities](#) para información sobre guardar y restaurar el estado.

5. Procesos vacíos

Un proceso que no contiene ningún componente activo de la aplicación. La única razón para mantener vivo este tipo de proceso es para el caché, para mejorar el tiempo de inicio la siguiente vez que un componente necesite ejecutarlo. El sistema termina frecuentemente con estos procesos para equilibrar los recursos globales del sistema entre los procesos de caché y los cachés kernel subyacentes.

Android clasifica un proceso en el máximo nivel que puede, basándose en la importancia de los componentes que están en ese momento activos en el proceso. Por ejemplo, si un proceso hostea un servicio y una actividad visible, el proceso se clasifica como un proceso visible y no como un proceso de servicio.

Además, la clasificación de un proceso puede ser incrementado si otros procesos son dependientes de él—un proceso que esté sirviendo a otro proceso nunca puede ser clasificado en un nivel más bajo que el proceso al que sirve. Por ejemplo, si un content provider en un proceso A está sirviendo a un cliente en un proceso B, o si un servicio en un proceso A está vinculado a un componente de un proceso B, el proceso A siempre será considerado por lo menos tan importante como el proceso B.

Dado que un proceso que ejecuta un servicio es clasificado en un nivel más alto que el proceso que tiene actividades en un segundo plano, una actividad que inicia una operación a largo plazo es bueno que inicie un [Service](#) para esa operación, en vez de crear un hilo —sobre todo si la operación dura más que la actividad.

Por ejemplo; una actividad que está subiendo una foto a la web debería empezar un servicio para realizar la subida para que ésta pueda continuar en un segundo plano a pesar de que el usuario deje la actividad. Utilizar un servicio garantiza que la operación tendrá al menos prioridad como "proceso de servicio" sin tener en cuenta lo que le pase a la actividad. Esta es la misma razón por la que los broadcast receivers deberían utilizar servicios en vez de poner en hilos las operaciones que son grandes consumidoras de tiempo.

Hilos

Cuando se lanza una aplicación, el sistema crea un hilo de ejecución para la aplicación, llamado "main." Este hilo es muy importante porque se encarga de enviar los eventos a los widgets apropiados de la interfaz de usuario, incluyendo los eventos de gráfica. También es el hilo con el que la aplicación interactúa con los componentes del toolkit de Android (componentes de los paquetes [android.widget](#) y [android.view](#)). Por esto, el hilo principal se llama también hilo del IU.

El sistema no crea un hilo diferente para cada instancia de un componente. Todos los componentes que se ejecutan en el mismo proceso son instanciados en el hilo del IU, y las llamadas del sistema a cada componente se hacen desde ese hilo. Consecuentemente, los métodos que responden a los callbacks del sistema (como por ejemplo el método [onKeyDown\(\)](#) para informar sobre las acciones del usuario o un método callback de ciclo de vida) siempre se ejecutan en el hilo del IU del proceso.

Cuando, por ejemplo, un usuario toca un botón en la pantalla, el hilo del IU de la aplicación envía ese evento al widget, que a su vez setea el nuevo estado clickeado y envía una petición invalidada a la cola de eventos. El hilo del IU saca la petición de la cola y notifica al widget que tiene que retirarse.

Cuando la aplicación realiza un trabajo intenso para responder a la interacción del usuario, este modelo de un sólo hilo puede producir un bajo rendimiento a no ser que se implemente apropiadamente la aplicación. Si todo está ocurriendo en el hilo del IU, cuando se realizan operaciones largas tal y como acceso a la red o consultas a base de datos, éstas pueden bloquear toda la IU. Cuando se bloquea el hilo, no se pueden repartir más eventos, ni siquiera los eventos de gráfica. Desde la perspectiva del usuario, la aplicación parece que se ha colgado. Si el hilo del IU se bloquea por más de unos pocos segundos (normalmente 5), al usuario le aparece el famoso diálogo "aplicación no responde" (ANR). El usuario puede decidir cerrar la aplicación e incluso desinstalarla.

Además, el toolkit de la IU de Android no es thread-safe. No se debe manipular la IU desde un hilo — se debe hacer desde el hilo de la IU.

Existen dos reglas a seguir en el modelo Android de un solo hilo:

- 1.No bloquear el hilo de la IU
- 2.No acceder al toolkit de la IU de Android desde fuera del hilo de la IU

Hilo trabajador (worker threads)

Dado el modelo de un sólo hilo descrito anteriormente, es vital para la respuesta de la IU de la aplicación que no se bloquee el hilo de la IU. Si existen operaciones no instantáneas a realizar, hay que asegurarse de llevarlas a cabo en hilos separados (hilos "en segundo plano" o "trabajadores").

A continuación, tenemos un ejemplo de código para un click listener que descarga una imagen desde un hilo diferente y lo muestra en un ImageView:

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");
            mImageView.setImageBitmap(b);
        }
    }).start();
}
```

Al principio, puede parecer que funciona perfectamente, ya que crea un nuevo hilo para manejar las operaciones de la red. Sin embargo, viola la segunda regla del modelo de un sólo hilo: no acceder al toolkit de la IU desde fuera del hilo de la IU —este ejemplo modifica el ImageView desde el hilo trabajador en vez desde el hilo de la IU.

Esto puede resultar en un comportamiento no definido e inesperado, que puede resultar difícil de localizar.

Para solucionar este problema, Android ofrece diferentes maneras de acceder al hilo de la IU desde otros hilos. Aquí tenemos una lista de métodos que pueden ayudar:

- Activity.runOnUiThread(Runnable)
- View.post(Runnable)
- View.postDelayed(Runnable, long)

Por ejemplo, se puede arreglar el código anterior utilizando el método View.post(Runnable):

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap = loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    });
}
```

```
    }  
    }).start();  
}
```

Esta implementación es thread-safe: la operación de la red se hace desde un hilo separado mientras que el ImageView se manipula desde el hilo de la IU.

Sin embargo, con el aumento de la complejidad de la operación, este tipo de código puede complicarse bastante y ser difícil de mantener. Para manejar interacciones más complicadas con un hilo trabajador, se puede considerar utilizar un Handler en el hilo trabajador, para procesar mensajes entregados por el hilo de la IU. Aunque, quizás la mejor solución sea, extender la clase AsyncTask, que simplifica la ejecución de las tareas de los hilos trabajadores que necesitan interactuar con la IU.

Utilizar AsyncTask

AsyncTask te permite realizar trabajo asincrónico en la IU. Realiza operaciones bloqueantes en un hilo trabajador y publica los resultados en el hilo de la IU, sin necesitar un manejo externo de los hilos o handlers.

Para utilizarlo, se debe utilizar la subclase AsyncTask e implementar el método callback doInBackground(), que se ejecuta en un pool de hilos en segundo plano. Para actualizar la IU, se debe implementar el método onPostExecute(), que entrega el resultado desde el método doInBackground() y se ejecuta en el hilo de la IU, por lo que se puede actualizar de forma segura la IU.

A continuación se puede ejecutar la tarea llamando al método execute() desde el hilo de la IU.

Por ejemplo, se puede implementar el anterior ejemplo utilizando el AsyncTask de la siguiente manera:

```
public void onClick(View v) {  
    new DownloadImageTask().execute("http://example.com/image.png"); }  
private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {  
    /** El sistema llama a este método para realizar el trabajo en un hilo trabajador y  
    * le entrega los parámetros dados a AsyncTask.execute() */  
    protected Bitmap doInBackground(String... urls) {  
        return loadImageFromNetwork(urls[0]);    }  
    /** El sistema lo llama para realizar trabajo en el hilo de la IU y entrega  
    * el resultado desde doInBackground() */  
    protected void onPostExecute(Bitmap result) {  
        mImageView.setImageBitmap(result);  
    }  
}
```

Ahora la IU es segura y el código más simple, dado que separa el trabajo en la parte en la que se debería realizar en un hilo trabajador y la parte en la que se debería realizar en un hilo de la IU.

Sería conveniente leer la parte de AsyncTask para un entendimiento completo de como utilizar esta clase, pero a continuación tenemos un resumen rápido de cómo trabaja:

- Se pueden especificar el tipo de parámetros, valores de progreso y el valor final de la tarea, utilizando genéricos.
- El método doInBackground() se ejecuta automáticamente en un hilo trabajador.
- Los métodos onPreExecute(), onPostExecute(), y onProgressUpdate() son llamados desde el hilo de la IU.
- El valor devuelto por el método doInBackground() se manda al método onPostExecute()
- Se puede llamar al método publishProgress() en cualquier momento desde doInBackground() para ejecutar el método onProgressUpdate() en el hilo de la IU

- Desde cualquier hilo, se puede cancelar la tarea en cualquier momento.

Precaución: Otro problema con el que nos podemos encontrar cuando utilizamos un hilo trabajador es el que una actividad se reinicie inesperadamente debido a cambios de configuración en tiempo de ejecución (como cuando el usuario cambia la orientación de la pantalla), que pueden destruir el hilo.

Métodos thread-safe

En algunas situaciones, los métodos que se implementan pueden ser llamados desde más de un hilo, y por lo tanto tienen que ser thread-safe.

Esto ocurre fundamentalmente con los métodos que pueden ser llamados de forma remota—como los métodos en un servicio vinculado. Cuando la llamada a un método implementado en un IBinder se origina en el mismo proceso en el que se ejecuta el IBinder, el método se ejecuta en el hilo del que llama. Sin embargo, cuando la llamada se origina en otro proceso, el método se ejecuta en un hilo escogido de un pool de hilos que el sistema mantiene en el mismo proceso que el IBinder (no se ejecuta en el hilo de la IU del proceso).

Por ejemplo, mientras que un método onBind() de un servicio sería llamado desde el hilo de la IU del proceso del servicio, los métodos implementados en el objeto devuelto por el método onBind() (por ejemplo, una subclase que implementa métodos RPC) serían llamados desde los hilos en el pool. Dado que un servicio puede tener más de un cliente, más de un hilo de un pool puede captar al mismo tiempo, el mismo método IBinder. Por lo tanto, los métodos IBinder deben ser implementados como thread-safe.

De forma similar, un content provider puede recibir peticiones de datos originados en otros procesos. A pesar de que las clases ContentResolver y ContentProvider ocultan los detalles de como se gestiona la comunicación interproceso, los métodos ContentProvider que responden a esas peticiones—los métodos query(), insert(), delete(), update(), y getType()—se llaman desde un pool de hilos en el proceso del content provider, no desde el hilo de la IU del proceso. Ya que estos métodos pueden ser llamados desde muchos hilos a la vez, también deben ser implementados como thread-safe.

Comunicación Interproceso

Android ofrece un mecanismo para una comunicación interproceso (IPC) utilizando procedimientos de llamada remota (RPCs), en la cual un método es llamado por una actividad u otro componente de la aplicación, pero se ejecuta de forma remota (en otro proceso), devolviendo cualquier resultado al que le ha llamado. Esto implica la descomposición de la llamada de un método y de sus datos a un nivel en que el sistema operativo lo pueda entender, transmitiéndolo desde el proceso local y el espacio de direcciones al proceso remoto y espacio de direcciones, con una posterior composición y reconstrucción de la llamada en ese lugar. Los valores devueltos son transmitidos en dirección contraria. Android suministra todo el código para realizar estas transacciones IPC, para que nos podamos centrar en la definición e implementación de la interfaz RPC.

Para realizar IPC, la aplicación se debe vincular a un servicio, utilizando el método bindService(). Para más información, ver el apartado de Servicios.

Interfaz de Usuario en Android

En una aplicación Android, la interfaz de usuario se desarrolla utilizando objetos View y ViewGroup. Existen varios tipos de view y view groups, cada uno siendo descendiente de la clase View.

Los objetos View son las unidades básicas de expresión de la interfaz de usuario en la plataforma de Android.

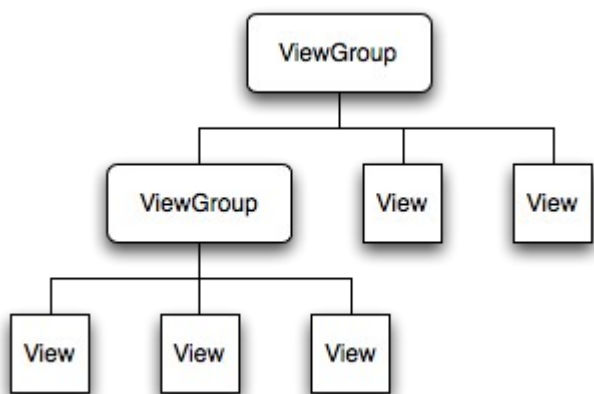
La clase View sirve de base para las subclases llamadas "widgets," que ofrecen objetos de la IU totalmente implementados, como los campos de texto y los botones.

La clase `ViewGroup` sirve de base a las subclases llamadas "layouts," que ofrecen diferentes tipos de arquitectura de layout, como la lineal, tabular o relativa.

Un objeto `View` es una estructura de datos cuyas propiedades almacenan los parámetros del layout así como el contenido para un área rectangular específica de la pantalla. Un objeto `View` maneja las medidas, el layout, la gráfica, el cambio de focus, scrolling así como las interacciones que se produzcan en el área rectangular correspondiente de la pantalla. Dado que es un objeto de la interfaz de usuario, un `View` también es un punto de interacción entre el usuario y el receptor de los eventos.

Jerarquía de Views

Como se muestra en el diagrama siguiente, en la plataforma Android, se define una actividad de IU utilizando una jerarquía de nodos `Views` y `ViewGroups`. Este árbol de jerarquía puede ser tan simple o complicado como se necesite, y se puede construir bien utilizando los widgets y layouts predefinidos de Android, o bien utilizando los `Views` personalizados creados por uno mismo.



Para adjuntar el árbol de jerarquía de views a la pantalla con vistas a la renderización, la Actividad debe llamar al método `setContent View()` y pasarle la referencia al objeto nodo raíz. El sistema Android recibe esta referencia y la utiliza para invalidar, medir y dibujar el árbol. El nodo raíz de la jerarquía pide que sus nodos hijos se dibujen a sí mismos — a su vez, cada nodo `view group` es responsable de llamar a cada uno de sus `views` hijos para que se dibujen a sí mismos. Los hijos pueden pedir un tamaño y una localización dentro del padre, pero el objeto padre tiene la decisión final sobre como de grande debe ser cada hijo. Android parsea los elementos del layout en orden (desde la parte superior del árbol jerárquico), instanciando los `Views` y añadiéndolos a los padres. Dado que estos son dibujados en orden, si existen elementos con posiciones solapadas, el último que se dibuje estará situado encima de los que se han dibujado previamente en ese mismo espacio.

Layout

La manera más común de definir un layout y de expresar una jerarquía `view` es con un archivo layout XML. XML ofrece una estructura para el layout que es legible para el ser humano, como el HTML. Cada elemento en el XML es un objeto `View` o `ViewGroup` (o descendiente). Los objetos `View` son las hojas en el árbol y los objetos `ViewGroup` son las ramas (ver la figura anterior sobre la jerarquía `View`).

El nombre de un elemento XML es correspondiente a la clase Java que representa. Por lo tanto un elemento `<TextView>` crea en la IU un `TextView`, y un elemento `<LinearLayout>` crea un `view group LinearLayout`. Cuando se carga un recurso layout, el sistema Android inicializa estos objetos de tiempo de ejecución, correspondientes a los elementos del layout.

Por ejemplo, a continuación podemos ver un layout vertical simple con un `view` texto y un botón:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>

```

Obsérvese que el elemento `LinearLayout` contiene tanto el `TextView` como el botón. Dentro se puede anidar otro `LinearLayout` (u otro tipo de view group), para alargar la jerarquía view y crear un layout más complejo.

Truco: También se pueden dibujar objetos `View` y `ViewGroups` con código java, utilizando los métodos `addView(View)` que insertan dinámicamente nuevos objetos `View` y `ViewGroup`.

Existen varias maneras en las que se pueden distribuir los views. Se pueden estructurar view groups e hijos view de muchas maneras utilizando diferentes tipos de view groups. Dentro de los view groups definidos que ofrece Android (llamados layouts) se incluyen los `LinearLayout`, `RelativeLayout`, `TableLayout`, `GridLayout` y otros. Cada uno de ellos ofrece un conjunto único de parámetros de layout que son usados para definir las posiciones de los views hijos y de la estructura del layout.

Widgets

Un widget es un objeto `View` que sirve de interfaz para la interacción con el usuario. Android nos ofrece un conjunto de widgets completamente implementados para construir fácilmente la IU, como botones, campos de texto, etc. Otros widgets ofrecidos por Android son más complejos, como un reloj o un control de zoom. También se pueden crear elementos personalizados propios, bien definiendo un objeto `View` propio o bien extendiendo y combinando widgets ya existentes.

Eventos de la IU

Una vez que se han añadido algunos `Views`/widgets a la IU, hay que saber como interaccionan con el usuario, para poder realizar acciones. Para ser informados sobre los eventos de la IU, hay que hacer dos cosas:

- Definir un evento listener y registrarlo con el `View`. Esta es la manera más frecuente de escuchar a los eventos. La clase `View` contiene una colección de interfaces anidadas llamadas `On<algo>Listener`, cada uno con un método callback llamado `On<algo>()`. Por ejemplo, `View.OnClickListener` (para manejar "clicks" en un `View`), `View.OnTouchListener` (para manejar en un `View` eventos al tocar la pantalla) y `View.OnKeyListener` (para manejar dentro de un `View` los clicks de las teclas en el dispositivo). Por lo tanto si se quiere que el `View` sea informado cuando se le haga "click" (como cuando se selecciona el botón), hay que implementar el `OnClickListener` y definir el método callback `onClick()` (donde se realiza la acción cuando se hace el click), y registrarlo en el `View` con el `setOnClickListener()`.
- Sobrecribir un método callback ya existente para el `View`. Esto es lo que se debe hacer cuando se ha implementado una clase `View` propia y se quiere escuchar eventos específicos que ocurren dentro de ella. Los eventos que se pueden manejar de esta manera incluyen cuando se toca la pantalla (`onTouchEvent()`), cuando se mueve el trackball (`onTrackballEvent()`), o cuando se toca una tecla del dispositivo (`onKeyDown()`). Esto permite definir el comportamiento por defecto de cada evento dentro de la clase `View` personalizada y determinar si el evento debería ser pasado a otro `View` hijo.

Esto, de nuevo, son callbacks a la clase View, por lo que la única ocasión de definirlos es cuando se desarrolla un componente personalizado.

Menus

Los menús de las aplicaciones son otra parte importante de una aplicación de IU. Los menús ofrecen una interfaz confiable que muestran las funciones y configuraciones de la aplicación. El menú más común de una aplicación se muestra cuando se hace click en la tecla MENU del dispositivo. Sin embargo, también se puede añadir Context Menus, que se muestran cuando el usuario aprieta y mantiene el dedo encima de un item.

Los menús también se estructuran utilizando una jerarquía View, pero uno no define esta estructura. Lo que se debe hacer es definir para la actividad, los métodos callback `onCreateOptionsMenu()` o `onCreateContextMenu()` y declarar los items que se quieran incluir en el menu. En el momento apropiado, Android creará automáticamente la jerarquía View necesaria para el menú y dibujará en ella cada uno de los items del menú.

Los menús también manejan sus propios eventos, por lo que no es necesario registrar los listeners de los eventos en los items del menú. Cuando se selecciona un item del menú, el método `onOptionsItemSelected()` o el método `onContextItemSelected()` serán llamados por el framework.

Como en el layout de la aplicación, también existe la opción de declarar los items del menu en el archivo XML.

Temas avanzados

Una vez estudiados los temas fundamentales sobre como crear una interfaz de usuario, a continuación se detalla opciones avanzadas para crear una interfaz más compleja.

Adaptadores (Adapters)

Si se quiere rellenar un view group con información que no se puede harcodear, se puede vincular el view con la fuente de datos externa. Para hacerlo, se utilizan un AdapterView como view group y cada View hijo se inicializa y se rellena con datos desde el Adaptador.

El objeto AdapterView es una implementación del ViewGroup que determina sus views hijos basándose en un determinado objeto Adapter. El Adaptador actúa como un mensajero entre la fuente de datos (por ejemplo un array de strings externos) y el AdapterView, que lo muestra. Existen diferentes implementaciones de la clase Adapter para tareas específicas, como por ejemplo el CursorAdapter para leer datos de la base de datos desde un Cursor, o un ArrayAdapter para leer desde un array arbitrario.

Estilos y temáticas

Puede ser que no se esté satisfecho con los widgets estándar. Se pueden crear de estilos y temáticas propios.

- Un estilo es un conjunto de uno o más atributos de formato que se pueden aplicar como unidad a elementos individuales en el layout. Por ejemplo, se puede definir un estilo que especifica un determinado color y tamaño para el texto y sólo aplicarlo a elementos View específicos.
- Una temática es un conjunto de uno o más atributos de formato que se pueden aplicar como una unidad a todas las actividades en una aplicación o a una sola. Por ejemplo, se puede definir una temática que determina el color del marco de la ventana y del panel de fondo y determina el tamaño del texto y los colores para los menús. Esta temática puede ser aplicada a actividades específicas o a la aplicación completa.

Los estilos y las temáticas son recursos. Se pueden usar los estilos por defecto que Android ofrece así como sus recursos temáticos o bien se pueden declarar unos estilos y recursos temáticos

personalizados propios.

Declaración del Layout en Android

El layout es la arquitectura para la interfaz de usuario en una Actividad. Define la estructura del layout y tiene todos los elementos que le aparecen al usuario. Se puede declarar el layout de dos maneras:

- Declarar los elementos de la IU en el XML. Android proporciona un vocabulario XML sencillo que corresponde a las clases y subclases View, como los correspondientes a los widgets y layouts.
- Instanciar los elementos layout en tiempo de ejecución. La aplicación puede crear programáticamente objetos View y ViewGroup (y manipular sus propiedades).

El framework Android da la flexibilidad para usar cualquiera de estos dos métodos para declarar y manejar la IU de la aplicación. Por ejemplo, se pueden declarar los layouts por defecto en el XML, incluyendo los elementos que aparecerán y sus propiedades. Se puede añadir código en la aplicación para modificar en tiempo de ejecución el estado de los objetos de la pantalla, incluyendo los declarados en XML.

La ventaja de declarar la IU en el XML es que permite separar mejor la presentación de la aplicación del código que controla su comportamiento. Las descripciones de la IU son externas al código de la aplicación, lo que significa que se puede modificar o adaptar sin tener que modificar el código fuente para después recompilarlo. Se puede crear, por ejemplo, layouts XML para diferentes orientaciones de pantalla, para diferentes tamaños de pantalla del dispositivo y diferentes idiomas. Además, declarar el layout en el XML hace más fácil visualizar la estructura de la IU, por lo que resulta más fácil debuggear. Este documento se centra en enseñar como declarar el layout en el XML. Si está interesado en instanciar objetos View en tiempo de ejecución, ver la documentación sobre las clases [ViewGroup](#) y [View](#).

En general, el vocabulario XML para declarar elementos de la IU concuerda con la estructura y nombre de las clases y los métodos, donde los nombre de los elementos corresponden con los nombres de las clases y los nombres de los atributos corresponden con los métodos. De hecho, la correspondencia puede ser tan directa, que se puede adivinar que atributo del XML corresponde con el método de la clase o que clase corresponde con un elemento dado del XML. Sin embargo, no todo el vocabulario es idéntico. En algunos casos, hay pequeñas diferencias en los nombres. El elemento EditText tiene un atributo text que corresponde a un EditText.setText().

Escribir el XML

La documentación sobre el API para las clases relacionadas con la IU incluye un listado de los atributos XML disponibles que corresponden a los métodos de las clases, incluyendo los atributos heredados.

Utilizando el vocabulario del XML de Android, se puede diseñar rápidamente layouts de IU así como los elementos de pantalla que contienen, de la misma manera en que se crea páginas web en HTML— con una serie de elementos anidados.

Cada archivo de un layout contiene un elemento raíz, que debe ser un objeto View o ViewGroup. Una vez que se ha definido el elemento raíz, se pueden añadir más objetos layout o widgets como elementos hijos para construir gradualmente una jerarquía View que define el layout. Aquí tenemos, por ejemplo, un layout XML que usa un [LinearLayout](#) vertical para guardar un [TextView](#) y un [Button](#):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```



```

        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical" >
<TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
<Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>

```

Después de declarar el layout en el XML, hay que guardar el archivo con extensión .xml en el directorio del proyecto `res/layout/`, para que pueda compilar correctamente.

Cargar el recurso XML

Cuando se compila la aplicación, cada archivo layout XML se compila a un recurso `View`. Se debe cargar el recurso del layout desde el código de la aplicación, en la implementación del callback `Activity.onCreate()`. Se hace mediante la llamada al método `setContentView()`, pasándole la referencia al recurso del layout en la forma: `R.layout.layout_file_name`. A continuación tenemos un ejemplo de si el XML del layout se guarda en `main_layout.xml`, para una actividad se carga de esta manera:

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}

```

En la actividad, el método callback `onCreate()` es llamado por el framework de Android cuando se lanza la actividad (ver el apartado sobre el ciclo de vida, en el documento sobre [Activities](#)).

Atributos

Cada objeto `View` y `ViewGroup` soporta su propia variedad de atributos XML. Algunos atributos son específicos para un objeto `View` (`TextView`, por ejemplo, soporta el atributo `textSize`), pero estos atributos son heredados también por cualquier objeto `View` que extienda de esta clase. Algunos son comunes a todos los objetos `View`, ya que son heredados de la clase raíz `View` (como el atributo `id`). Otros atributos son considerados como "parámetros layout," que son atributos que describen determinadas orientaciones layout del objeto `View`, definidas por el objeto padre `ViewGroup`.

ID

Cualquier objeto `View` puede tener un integer ID asociado a él, que identifica unívocamente el `View` dentro del árbol. Cuando la aplicación se compila, este ID se referencia como un integer, pero el ID, en realidad, es un string en el atributo `id` del archivo XML del layout. Este es un atributo común a todos los objetos `View` (definidos por la clase `View`) y se utiliza con frecuencia. La sintaxis para el ID dentro de un tag XML es la siguiente:

```

android:id="@+id/my_button"

```

El símbolo arroba (`@`) al principio del string indica que el parseador XML tiene que pasar por allí y expandir el resto del string ID e identificarlo como un recurso ID. El símbolo (`+`) significa que es un nombre nuevo de un recurso que debe ser creado y añadido a nuestros recursos (en el archivo `R.java`). El framework de Android ofrece otros recursos ID. Cuando se referencia un recurso ID Android, no se necesita el símbolo (`+`), pero hay que escribir el nombre del paquete de la

siguiente manera:

```
android:id="@android:id/empty"
```

Con el nombre del paquete android en su lugar, estamos referenciando un ID de la clase de recursos android.R, en vez de uno de la clase de recursos local.

Para crear views y sus referencias a ellos desde la aplicación, se hace de esta manera:

1. Definir un view/widget en el archivo layout y asignarle un ID único:

```
<Button android:id="@+id/my_button"  
android:layout_width="wrap_content"  
android:layout_height="wrap_content"  
android:text="@string/my_button_text"/>
```

2. Crear una instancia del objeto view y capturarlo desde el layout (normalmente en el método `onCreate()`):

```
Button myButton = (Button) findViewById(R.id.my_button);
```

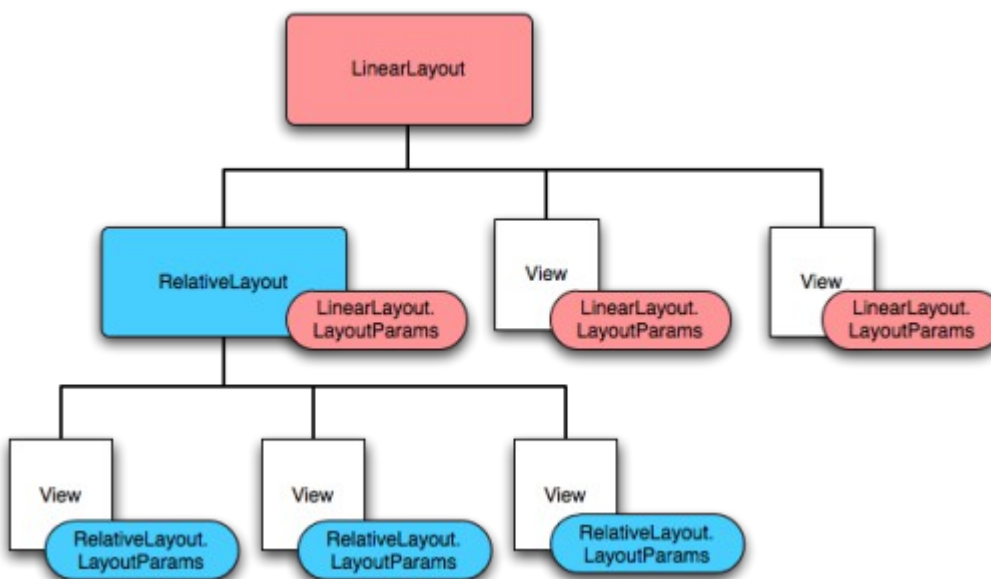
Es importante definir IDs para los objetos view cuando se crea un RelativeLayout. En un layout relativo, los sibling views pueden definir su layout relativo respecto a otro sibling view (view hermano), que está referenciado por un ID único.

No es necesario que un ID sea unívoco en todo el árbol, pero debe ser unico dentro de la parte del árbol en la que se busca (lo que puede ser todo el árbol, por lo tanto es mejor que sea completamente único cuando sea posible).

Parámetros del Layout

Los atributos del XML del layout llamados `layout_*` algo definen parámetros del layout para el View que son apropiados para el ViewGroup en el que está.

Cada ViewGroup implementa una clase anidada que extiende ViewGroup.LayoutParams. Esta subclase contiene tipos de propiedad que definen el tamaño y posición para cada view hijo, apropiado para el view group. Como se ve en el siguiente gráfico, el view group padre define los parámetros del layout para cada view hijo (incluyendo el view group hijo).



Visualización de una jerarquía view con parámetros layout asociados a cada view.

Observe que cada subclase `LayoutParams` tiene su propia sintaxis para setear los valores. Cada elemento hijo debe definir los `LayoutParams` que son apropiados para su padre, aunque también puede definir diferentes `LayoutParams` para sus propios hijos.

Todos los view group incluyen un ancho y una altura (`layout_width` y `layout_height`), y se necesita cada view para definirlos. Muchos `LayoutParams` también incluyen márgenes y bordes opcionales.

Se puede especificar el ancho y la altura con medidas exactas, pero no se suele hacer muy a menudo. Normalmente, se usan una de estas constantes para setear el ancho o la altura:

- `wrap_content` le dice al view que se dimensione según lo necesario para adecuarse a su contenido.
- `fill_parent` (renombrado `match_parent` en el nivel 8 del API) le dice al view que se agrande todo lo que le permita el view group padre.

En general, no se recomienda utilizar unidades absolutas como los píxeles para especificar el ancho o la altura del un layout. Es mejor utilizar, medidas relativas como las unidades pixel independientes de densidad (dp), `wrap_content`, o `fill_parent`, ya que ayuda a garantizar que se muestre correctamente la aplicación en una gran variedad de tamaños de pantallas.

Posición del Layout

Los view tienen la geometría de un rectángulo. Tiene una localización, expresada como un par de coordenadas `left` y `top`, y dos dimensiones, expresadas como el ancho y la altura. La unidad de la localización es el pixel.

Es posible conseguir la localización del view llamando a los métodos `getLeft()` y `getTop()`. El primero devuelve X y el último devuelve Y. Estos métodos devuelven la localización del view relativa a sus padres. Cuando, por ejemplo, `getLeft()` devuelve 20, eso significa que el view está situado a 20 píxeles a la derecha del borde de la izquierda del padre.

Existen varios métodos para evitar cálculos innecesarios como; `getRight()` y `getBottom()`. Estos métodos devuelven las coordenadas de los bordes de la derecha y de abajo del rectángulo que representa el view. Llamar, por ejemplo, al método `getRight()` es similar al siguiente cálculo: `getLeft() + getWidth()`.

Tamaño, padding y márgenes

El tamaño de un view se expresa con un ancho y una altura. Un view tiene dos pares de valores de ancho y de alto.

El primer par es conocido como `measured width` (ancho medido) y `measured height` (altura medida). Estas dimensiones definen cómo de grande va a ser un view con respecto al padre. Las dimensiones medidas pueden ser obtenidas llamando a los métodos `getMeasuredWidth()` y `getMeasuredHeight()`.

El segundo par es conocido como `width` (ancho) y `height` (alto), o a veces como `drawing width` (ancho de dibujo) y `drawing height` (altura de dibujo). Estas dimensiones definen el tamaño actual del view en la pantalla, el tamaño durante el tiempo en el que se dibuja y el tamaño después del layout. Estos valores, pueden ser diferentes a la altura medida y el ancho medido, aunque no tienen porque serlo. El ancho y la altura pueden ser obtenidos llamando a los métodos `getWidth()` y `getHeight()`.

Para medir estas dimensiones, el view tiene en cuenta el padding. El padding se expresa en píxeles tanto para la parte izquierda, como para la derecha, como para la parte de abajo del view. El padding puede ser usado para compensar en un determinado número de píxeles, el contenido del view. Un padding izquierdo de 2, por ejemplo, empujará el contenido del view 2 píxeles hacia la derecha desde el borde izquierdo. El padding puede ser seteado utilizando el método `setPadding(int, int, int, int)` y consultado llamando al método `getPaddingLeft()`, `getPaddingTop()`, `getPaddingRight()` y `getPaddingBottom()`.

A pesar de que un view puede definir un padding, no proporciona soporte para márgenes. Sin embargo, los view groups si que proporcionan ese soporte. Para más información ver [ViewGroup](#) y [ViewGroup.MarginLayoutParams](#).

Creación de Menús para una Interfaz de Usuario

Los menús son una parte importante de una actividad de una interfaz de usuario, que aportan a los usuarios una manera sencilla de realizar acciones. Android ofrece un framework sencillo para que añadir los menús estándar a la aplicación.

Existen tres tipos de menús para aplicaciones:

Menú de opciones

Colección de ítems de menú para una actividad, que aparece cuando el usuario toca el botón de MENU. Cuando la aplicación se ejecuta en Android 3.0 o en una versión posterior, se pueden colocar los accesos rápidos para seleccionar los ítems de un menú en una Barra de Acción, como "ítems de acción."

Menú de Contexto

Menú emergente que aparece cuando el usuario hace click en un view que está registrado para aportar un menú de contexto.

Submenú

Lista de ítems en un menú que aparece cuando el usuario hace click en un ítem de un menú que contiene un menú anidado.

Este documento muestra como crear cada tipo de menú, utilizando el XML para definir el contenido del menú y los métodos callback de la actividad para que respondan cuando el usuario selecciona un ítem.

Crear un Recurso de Menú

En vez de instanciar un Menú en el código de la aplicación, se debe definir un menú y todos sus ítems en un XML, y después inflar el recurso de menú (cargarlo como un objeto) en el código de la aplicación. Utilizar el recurso de menú para definir el menú es una buena práctica ya que separa el contenido del menú, del código de la aplicación. También es más fácil visualizar la estructura y contenido de un menú en un XML.

Para crear un recurso de menú, hay que crear un archivo XML dentro del directorio del proyecto res/menu/ y construir el menú con los siguientes elementos:

<menu>

Define un Menu, que es un contenedor para un menú de ítems. Un elemento de <menu> debe ser el nodo raíz del archivo y guarda uno o más <item> y elementos de <group>.

<item>

Crea un MenuItem, que representa un sólo ítem en un menú. Este elemento puede contener un elemento de <menu> anidado para crear un submenú.

<group>

Es un container invisible y opcional para elementos <item> elements. Permite clasificar los ítems de un menú de manera que compartan propiedades como el estado activo y la visibilidad. Ver la sección sobre Grupos de Menú.

Aquí tenemos un ejemplo llamado game_menu.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/new_game"
        android:icon="@drawable/ic_new_game"
        android:title="@string/new_game" />
  <item android:id="@+id/help"
        android:icon="@drawable/ic_help"
        android:title="@string/help" />
</menu>
```

Este ejemplo define un menú con dos items. Cada item incluye los atributos:

android:id

Un recurso ID único, que permite a la aplicación reconocer el item cuando el usuario lo selecciona.

android:icon

Una referencia a una gráfica que se usa como icono del item.

android:title

Una referencia a un string que se usa como título del item.

Existen más atributos que se pueden incluir en un `<item>`, incluyendo algunos que especifican como el item puede aparecer en la Barra de Acción.

Inflar un Recurso de Menú

Desde el código de la aplicación, se puede inflar un recurso de menú (convertir un recurso XML en un objeto programable) utilizando `MenuInflater.inflate()`. Como ejemplo, tenemos el siguiente código que infla el `archivogame_menu.xml` definido anteriormente, en el método callback `onCreateOptionsMenu()`, para utilizar el menú como la actividad de Menú de Opciones:

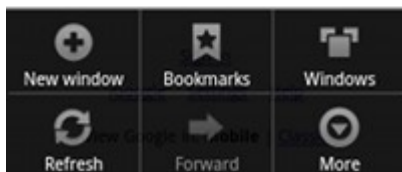
```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.game_menu, menu);
    return true;
}
```

El método `getMenuInflater()` devuelve un `MenuInflater` a la actividad. Con este objeto, se puede llamar al método `inflate()`, que infla un recurso de menú a un objeto `Menu`. En este ejemplo, el recurso de menú definido por `game_menu.xml` es inflado al `Menu` que se ha pasado al método `onCreateOptionsMenu()`. (Este método callback para el Menú de Opciones se discute con detalle en la siguiente sección.)

Crear un Menú de Opciones

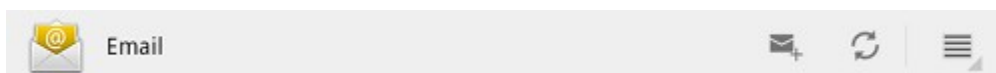
El Menú de Opciones es donde se deben incluir la actividad básica de las acciones y los items de navegación básicos (por ejemplo, un botón para abrir las configuraciones de la aplicación). Los items en el Menú de Opciones son accesibles de dos maneras diferentes: el botón MENU o en la Barra de Acción (en dispositivos con Android 3.0 o superior).

Cuando el dispositivo tiene un sistema operativo Android 2.3 o inferior, el Menú de Opciones aparece en la parte de abajo de la pantalla, como se muestra en la siguiente imagen. Cuando se abre, la primera parte visible del Menú de Opciones es el menú de iconos. Guarda los primeros seis items de menú. Si se añaden más de seis items al menú de opciones, Android coloca el sexto item y los siguientes en el menú overflow, que el usuario puede abrir si hace click en el item del menú "More".



Pantallazo del Menú de Opciones en el Navegador.

En las versiones de Android 3.0 y superiores, los items del Menú de Opciones se colocan en la Barra de Acción, que aparece en la parte superior de la actividad reemplazando la barra del título. Todos los items del Menú de Opciones, se sitúan por defecto en el menu overflow, que el usuario puede abrir si hace click sobre el icono del menú que está en la parte derecha de la Barra de Acción. Sin embargo, se pueden colocar directamente items del menú en la Barra de Acción como "items de acción," para acceso instantáneo, como se muestra en la siguiente imagen



Pantallazo de la Barra de Acción en la aplicación de email, con dos items de acción del menú de opciones, más el menu overflow

Cuando el sistema Android crea por primera vez el menú de opciones, llama al método `onCreateOptionsMenu()` de la actividad. Se sobrescribe este método en la actividad y se rellena el `Menu` que se pasa como argumento a este método, inflando un recurso del menú como se describe en Inflar un recurso del menú. Por ejemplo:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.game_menu, menu);
    return true;
}
```

También se puede rellenar el menú en el código, utilizando `add()` para añadir items al `Menu`.

Nota: En Android 2.3 e inferiores, el sistema llama al método `onCreateOptionsMenu()` para crear el menú de opciones cuando el usuario lo abre por primera vez, pero en la versión de Android 3.0 o superior, el sistema lo crea en el momento en el que se crea la actividad, para rellenar la Barra de Acción.

Respuesta a la acción del usuario

Cuando el usuario selecciona un item del menú del menú de opciones (incluyendo los items de acción de la barra de acción), el sistema llama al método `onOptionsItemSelected()` de la actividad. Este método pasa el `MenuItem` elegido por el usuario. Se puede identificar el item del menú

llamando al método `getItemId()`, que devuelve el ID único para ese ítem del menú (definido por el atributo `android:id` en el recurso del menú o mediante un `integer` dado al método `add()`). Se puede machear este ID con los ítems del menú conocidos y realizar la acción apropiada. Por ejemplo:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle item selection
    switch (item.getItemId()) {
        case R.id.new_game:
            newGame();
            return true;
        case R.id.help:
            showHelp();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

En este ejemplo, el método `getItemId()` obtiene el ID del ítem seleccionado y el `switch` compara el ID con los IDs de los recursos asignados a los ítems del menú en el XML de recursos. Cuando uno de los casos del `switch` nos devuelve un ítem del menú, devuelve `true` para indicar que la selección del ítem se ha llevado a cabo. Si no, el caso `default` pasa el ítem del menú a la super clase, por si pudiera manejar el ítem seleccionado. (Si se ha extendido directamente la clase `Activity`, la super clase devuelve `false`, pero es una buena práctica pasar a la super clase ítems no manejados en vez de devolver directamente `false`.)

Además, Android 3.0 añade la posibilidad de que se defina el comportamiento `on-click` para un ítem del menú en el XML del recurso del menú, utilizando el atributo `android:onClick`. De esta manera no se necesita implementar el método `onOptionsItemSelected()`. Utilizando el atributo `android:onClick`, se puede especificar un método al que llamar cuando el usuario selecciona el ítem del menú. La actividad debe implementar el método especificado en el atributo `android:onClick` para que acepte un parámetro único `MenuItem`—cuando el sistema llama a este método, le pasa el ítem seleccionado.

Truco: Si la aplicación contiene varias actividades y algunas de ellas ofrecen el mismo menú de opciones, hay que considerar crear una actividad que no implemente nada más que los métodos `onCreateOptionsMenu()` y `onOptionsItemSelected()`. Después habría que extender esta clase para cada actividad que comparta el mismo menú de opciones. De esta manera, sólo se tiene que gestionar un grupo de líneas de código para manejar las acciones de menú y cada clase hija hereda los mismos comportamientos del menú.

Si se quiere añadir ítems del menú a uno de las actividades hijas, se sobrescribe el método `onCreateOptionsMenu()` en esa actividad. Hay que llamar al `super.onCreateOptionsMenu(menu)` para crear los ítems del menú originales y después añadir los ítems del menú con el método `menu.add()`. También se puede sobrescribir el comportamiento de la super clase para ítems individuales.

Cambiar los ítems del menú en tiempo de ejecución

Una vez que la actividad se crea, se llama al método `onCreateOptionsMenu()` una sola vez, tal y como se describe anteriormente. El sistema guarda y reutiliza el `Menu` que se ha definido en este método hasta que la actividad se destruya. Si se quiere cambiar el menú de opciones en cualquier momento después de creado, se debe sobrescribir el método `onPrepareOptionsMenu()`. Este método pasa el objeto `Menu` tal y como existe en este momento. Esto es útil si se quiere eliminar, añadir, deshabilitar o habilitar los ítems del menú dependiendo del estado en curso de la aplicación.

En Android 2.3 y versiones anteriores, el sistema llama al método `onPrepareOptionsMenu()` cada vez que el usuario abre el menú de opciones.

En Android 3.0 y superiores, se debe llamar al método `invalidateOptionsMenu()` cuando se quiere actualizar el menú, ya que el menú siempre está abierto. El sistema llamará al método `onPrepareOptionsMenu()` para que se puede actualizar los items del menú.

Nota: No se debería cambiar los items en el menú de opciones basándose en el `View` que está en ese momento en el focus. Cuando se está en modo táctil (cuando el usuario no está utilizando un trackball o d-pad), los views no pueden tener el focus, por lo que no se puede utilizar el focus como base para modificar los items en el menú de opciones. Si se quiere suministrar items de menú que sean sensibles al contexto de un `View`, hay que utilizar un `Context Menu`.

Si se está desarrollando para una versión Android 3.0 o superior, hay que asegurarse de leer [Utilizar la Barra de Acciones](#).

Crear un ContextMenu

Un `ContextMenu` es conceptualmente similar al menú mostrado cuando el usuario realiza "click en el botón derecho" en un PC. Se debe utilizar un context menu para darle acceso al usuario a acciones que pertenecen a un item específico en la interfaz de usuario. En Android, el context menu se muestra cuando el usuario realiza un "largo click" (pulsar y mantener) en un item.

Se puede crear un context menu para cualquier `View`, aunque los context menus son más usados para los items en un `ListView`. Cuando el usuario realiza un largo click en un item de un `ListView` y la lista está registrada para suministrar un context menu, el item de la lista muestra al usuario que un context menu está disponible mediante un cambio de color del fondo—cambia de naranja a blanco antes de abrir el context menu.

Para que un `View` suministre un context menu, se debe "registrar" el view para un context menu. Hay que llamar al método `registerForContextMenu()` y pasarle el `View` que se quiere dar al context menu. Cuando este `View` recibe el largo click, muestra el context menu.

Para definir el aspecto y el comportamiento del context menu, hay que sobrescribir los métodos callback de la actividad context menu, `onCreateContextMenu()` y `onContextItemSelected()`.

Aquí tenemos un ejemplo con `onCreateContextMenu()` que utiliza el recurso de menú `context_menu.xml`:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                               ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.context_menu, menu);
}
```

`MenuInflater` es utilizado para inflar el context menu desde un recurso de menú. (También se puede utilizar el método `add()` para añadir items de menú.) Los parámetros del método callback incluyen el `View` que el usuario ha seleccionado y un objeto `ContextMenu.ContextMenuInfo` que aporta información adicional sobre el item seleccionado. Se pueden utilizar estos parámetros para determinar que context menu crear, aunque en este ejemplo, todos los context menus para la actividad son iguales.

Cuando el usuario selecciona un item del context menu, el sistema llama al método `onContextItemSelected()`. Aquí hay un ejemplo de como se puede manejar los items seleccionados:


```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
    switch (item.getItemId()) {
        case R.id.edit:
            editNote(info.id);
            return true;
        case R.id.delete:
            deleteNote(info.id);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}

```

La estructura de este código es similar al ejemplo de "Crear un menú de opciones", en el que `getItemId()` obtiene el ID de los ítems seleccionados del menú y un `switch` machea el ítem a los IDs que están definidos en el recurso del menú. Igual que en el ejemplo del menú de opciones, la sentencia `default` llama a la super clase por si ésta puede manejar los ítems del menú que no son gestionados aquí.

En este ejemplo, el ítem seleccionado es un ítem del `ListView`. Para realizar una acción en el ítem seleccionado, la aplicación tiene que saber el ID de la lista para ese ítem seleccionado (su posición en la `ListView`). Para obtener este ID, la aplicación llama al método `getMenuInfo()`, que devuelve un objeto `AdapterView.AdapterContextMenuInfo` que incluye el ID de la lista para el ítem seleccionado en el campo `id`. Los métodos locales `editNote()` y `deleteNote()` aceptan que este ID de la lista realice una acción con los datos especificados por el ID de la lista.

Nota: Los ítems en un context menu no soportan iconos o accesos directos

Registrar un ListView

Si la actividad utiliza un `ListView` y se quiere que todos los ítems de la lista suministren un context menu, hay que registrar todos esos ítems pasando un `ListView` al `registerForContextMenu()`. Si, por ejemplo, se está utilizando un `ListActivity`, hay que registrar todos los list items de esta manera:

```
registerForContextMenu(getListView());
```

Crear Submenús

Un submenú es un menú que el usuario puede abrir si selecciona un ítem en otro menú. Se puede añadir un submenú a cualquier menú (exceptuando un submenú). Los submenús son útiles cuando la aplicación tiene muchas funciones que pueden ser organizados en temas, como por ejemplo los ítems en una barra de menú en un PC (File, Edit, View, etc.)

Cuando se crea el recurso de menú, se puede crear un submenú añadiendo un elemento `<menu>` como hijo de un `<item>`.

Por ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/file"
        android:icon="@drawable/file"
        android:title="@string/file" >
        <!-- "file" submenu -->
        <menu>
            <item android:id="@+id/create_new"
                android:title="@string/create_new" />
            <item android:id="@+id/open"
                android:title="@string/open" />
        </menu>
    </item>
</menu>
```

Cuando el usuario selecciona un ítem de un submenú, el método callback del ítem seleccionado del correspondiente menú padre, recibe el evento. Si, por ejemplo, el menú anterior se utiliza como un menú de opciones, cuando un ítem del submenú se selecciona se llama al método `onOptionsItemSelected()`.

También se puede utilizar el método `addSubMenu()` para añadir dinámicamente `SubMenu` a un ya existente `Menu`. Esto devuelve el nuevo objeto `SubMenu`, al que se puede añadir ítems de un submenú, utilizando el método `add()`.

Otras propiedades del Menú

A continuación vemos otras propiedades que se pueden aplicar a la mayoría de los ítems del menú.

Grupos del menú

Un grupo de un menú es una colección de ítems de un menú que comparten determinadas características. Con un grupo se puede:

- Mostrar u ocultar todos los ítems con el método `setGroupVisible()`
- Habilitar o deshabilitar todos los ítems con el método `setGroupEnabled()`
- Especificar si todos los ítems son chequeables con `setGroupCheckable()`

Se puede crear un grupo anidando elementos `<item>` dentro de un elemento `<group>` en el recurso de menú o especificando un ID de un grupo con el método `add()`.

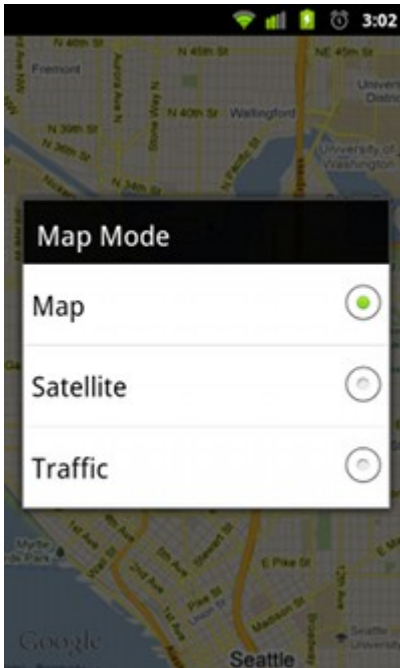
Aquí tenemos un ejemplo de un recurso de menú que incluye un grupo:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/item1"
        android:icon="@drawable/item1"
        android:title="@string/item1" />
    <!-- menu group -->
    <group android:id="@+id/group1">
        <item android:id="@+id/groupItem1"
            android:title="@string/groupItem1" />
        <item android:id="@+id/groupItem2"
            android:title="@string/groupItem2" />
    </group>
</menu>
```

Los ítems que están en el grupo se muestran igual que el primer ítem que no está en el grupo—los tres ítems del menú son siblings (hermanos). Sin embargo, se pueden modificar las características de los dos ítems del grupo referenciando el ID del grupo y utilizando los métodos listados anteriormente.

Radio Buttons

Un menú puede resultar útil como interfaz para activar o desactivar opciones, utilizando un checkbox para opciones individuales o radio buttons para grupos de opciones que se excluyen mutuamente entre sí. La imagen a continuación muestra un submenú con ítems chequeables con radio buttons.



Pantallazo de un submenú con radio buttons.

Nota: Los ítems de menú en el Icon Menu (del menú de opciones) no pueden mostrar un checkbox o un radio button. Si se escoge hacer los ítems del Icon Menu chequeables, se debe indicar manualmente el estado chequeado cambiando el icono y/o texto cada vez que el estado cambie.

Se puede definir el comportamiento chequeable para los ítems del menú utilizando el atributo `android:checkable` en el elemento `<item>`, o para el grupo completo con el atributo `android:checkableBehavior` en el elemento `<group>`. En este ejemplo a continuación, todos los ítems en este grupo son chequeables con un radio button:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item android:id="@+id/red"
            android:title="@string/red" />
        <item android:id="@+id/blue"
            android:title="@string/blue" />
    </group>
</menu>
```

El atributo `android:checkableBehavior` acepta cualquiera de los siguientes:

`single`

Uno sólo de los ítems del grupo pueden ser chequeados (radio buttons)

`all`

Todos los ítems pueden ser chequeados (checkboxes)

`none`

Ninguno de los ítems pueden ser chequeados

Se puede aplicar por defecto un estado chequeado utilizando el atributo `android:checked` en el elemento `<item>` y cambiarlo en el código con el método `setChecked()`.

Cuando se selecciona un ítem chequeable, el sistema llama al método callback correspondiente al ítem seleccionado (como el método `onOptionsItemSelected()`). Aquí es donde se debe setear el estado del checkbox, ya que el checkbox o el radio button no cambia su estado automáticamente. Se puede obtener el estado actual del ítem (tal y como estaba antes de que el usuario lo seleccionara) con el método `isChecked()` y setear el estado chequeado con el método `setChecked()`. Por ejemplo:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.vibrate:
        case R.id.dont_vibrate:
            if (item.isChecked()) item.setChecked(false);
            else item.setChecked(true);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Si no se setea el estado chequeado de esta manera, el estado visible del ítem (el checkbox o radio button) no cambiará cuando el usuario lo seleccione. Cuando sí se setea el estado, la actividad preserva el estado chequeado del ítem para que cuando el usuario abra el menú más adelante, el estado chequeado que se ha seteado sea visible.

Nota: Los ítems chequeables de un menú se deben usar en una sesión y no guardarlos después de que la aplicación haya sido destruida. Si se tienen configuraciones de la aplicación que se quieran guardar para el usuario, se deben guardar utilizando las Preferencias compartidas.

Accesos directos

Para facilitar el acceso rápido a los ítems en un menú de opciones cuando el dispositivo del usuario tiene un teclado, se pueden añadir teclas de acceso directo utilizando y/o números, con los atributos `android:alphabeticShortcut` y `android:numericShortcut` en el elemento `<item>` element. También se pueden utilizar los métodos `setAlphabeticShortcut(char)` y `setNumericShortcut(char)`. Las teclas de acceso directo no son sensibles a las mayúsculas o minúsculas.

Si, por ejemplo, hacemos que el carácter "s" sea un acceso directo a un ítem de menú "save", cuando se abre el menú (o mientras que el usuario mantenga apretado el botón MENU) y el usuario hace click en la tecla "s", se seleccionará el ítem del menú "save".

Este acceso directo se muestra en un ítem de un menú, debajo del nombre del ítem (exceptuando en los ítems del menú de iconos, que sólo se muestran si el usuario mantiene apretado el botón MENU).

Nota: Las teclas de acceso directo para los ítems de menú sólo funcionan en dispositivos con teclado. Los accesos directos no pueden añadirse a los ítems en un Context Menu.

Añadir dinámicamente intents del menú

En alguna ocasión se puede necesitar que un ítem de menú lance una actividad utilizando un Intent (bien si es una actividad de la aplicación o de otra aplicación). Cuando se conoce el intent que se quiere usar y se tiene el ítem del menú específico que inicia el intent, se puede ejecutar el intent con el método `startActivity()` durante el método callback correspondiente del ítem seleccionado (como el callback `onOptionsItemSelected()`).

Sin embargo, si no se está seguro de que el dispositivo del usuario contiene una aplicación que maneje el intent, añadir un ítem que lo llame puede resultar en un ítem no funcional, ya que el intent

puede que no lance la actividad. Para solventar esto, Android permite añadir dinámicamente los items de menú cuando Android encuentra en el dispositivo actividades que manejan el intent.

Para añadir items de menú basándose en las actividades disponibles que aceptan un intent:

1. Definir un intent con la categoría CATEGORY_ALTERNATIVE y/o CATEGORY_SELECTED_ALTERNATIVE, más cualquier otro requisito.

2. Llamar al método Menu.addIntentOptions(). Android busca cualquier aplicación que pueda realizar el intent y los añade al menú.

Si no existen aplicaciones instaladas que sirvan al intent, no se añade ningún item.

Nota: CATEGORY_SELECTED_ALTERNATIVE se utiliza para manejar el elemento seleccionado en ese momento en la pantalla. Sólo debería ser utilizado cuando se crea un menú con el método onCreateContextMenu().

Por ejemplo:

```
@Override
public boolean onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);

    // Crear un intent que describe los requisitos a completar, que hay que añadir
    // en el menú. La aplicación ofrecida debe incluir un valor de categoría Intent.CATEGORY_ALTERNATIVE.
    Intent intent = new Intent(null, dataUri);
    intent.addCategory(Intent.CATEGORY_ALTERNATIVE);

    // Buscar y rellenar el menú con aplicaciones adecuadas.
    menu.addIntentOptions(
        R.id.intent_group, // Grupo del menú al que se le añadirán los nuevos items.
        0, // ID único del item (ninguno)
        0, // Orden para los items (ninguno)
        this.getComponentName(),
        // El nombre de la actividad en curso
        null, // Items específicos a colocar en primer lugar (ninguno)
        intent, // Intents creados anteriormente que describen nuestros requisitos
        0, // Flags adicionales para controlar items (ninguno)
        null); // Array de MenuItem's que se correlacionan con items específicos (ninguno)

    return true;
}
```

Para cada actividad encontrada que aporte un intent filter que machee con un intent definido, se añade un item de menú, utilizando el valor del intent filter android:label como título del item de menú y el icono de la aplicación como icono del item de menú. El método addIntentOptions() devuelve el número de items añadidos.

Nota: Cuando se llama al método addIntentOptions(), sobrescribe todos los items de menú según el grupo de menú especificado en el primer argumento.

Permitir que la actividad sea añadida a otros menús.

Para que sean incluidas en el menú de otras aplicaciones, se necesita definir como siempre, un intent filter, incluyendo los valores de CATEGORY_ALTERNATIVE y/o CATEGORY_SELECTED_ALTERNATIVE para la categoría del intent filter category. Por ejemplo:

```
<intent-filter label="Resize Image">
    ...
    <category android:name="android.intent.category.ALTERNATIVE" />
    <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
</intent-filter>
```

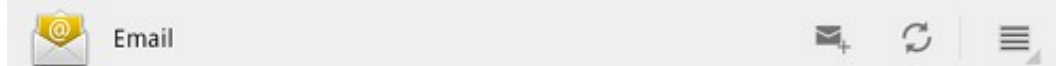
...
</intent-filter>

Para leer más sobre como escribir intent filters ver la sección [Intent Filters](#).

Barra de Acción- Action Bar

La barra de acción es un widget para actividades que reemplaza la tradicional barra del título en la parte superior de la pantalla. Por defecto, la barra de acción incluye el logo de la aplicación en el lado izquierdo, seguido del título de la actividad y cualquier item disponible del menú de opciones en el lado derecho. La barra de acción ofrece varias características útiles, incluyendo la habilidad para:

- Mostrar items del [Menú de Opciones](#) directamente en la barra de acción, como "items de acción"—aportando acceso instantáneo a las acciones clave para el usuario.
- Los items de menú que no aparecen como items de acción se sitúan en el overflow menu, y se muestran mediante una lista desplegable en la barra de acción.
- Proporciona pestañas para navegar entre [fragmentos](#).
- Proporciona una lista desplegable para la navegación.
- Proporciona "action views" interactivos en lugar de items de acción (como una caja de búsqueda).



Un pantallazo de la barra de acción en una aplicación de email, conteniendo items de acción para redactar un nuevo email y refrescar la bandeja de entrada.

Añadir la barra de acción

La barra de acción se incluye por defecto en todas las actividades cuyo objetivo es Android 3.0 o superior. Más específicamente, todas las actividades que utilizan el nuevo tema "holográfico" incluyen la barra de acción, y cualquier aplicación que tiene como objetivo Android 3.0 recibe automáticamente este tema. Se considera que una aplicación tiene como "objetivo" Android 3.0 cuando se ha seteado bien el atributo `android:minSdkVersion` o bien el atributo `android:targetSdkVersion` en el elemento `<uses-sdk>` a "11" o mayor que 11. Por ejemplo:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.helloworld"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-sdk android:minSdkVersion="4"
        android:targetSdkVersion="11" />
    <application ... >
        ...
    </application>
</manifest>
```

En este ejemplo, la aplicación necesita una versión mínima de nivel API 4 (Android 1.6), pero también tiene como objetivo API nivel 11 (Android 3.0). De esta manera, cuando se instala la aplicación en un dispositivo con Android 3.0 o superior, el sistema aplica el tema holográfico a cada actividad, y por tanto, cada actividad incluye la barra de acción.

Sin embargo, si se quiere utilizar los APIs de la barra de acción, como añadir pestañas o modificar los estilos de la barra de acción, hay que setear el `android:minSdkVersion` a "11", para poder acceder a la clase [ActionBar](#).

Eliminar la barra de acción

Si se quiere eliminar la barra de acción para una actividad en concreto, hay que crear una temática personalizada que extienda de Theme.Holo, y dentro de esta temática, setear la propiedad estilo android:windowActionBar a "false". Por ejemplo:

```
<style name="MyTheme" parent="android:Theme.Holo.Light">
    <item name="android:windowActionBar">false</item>
    <item name="android:windowNoTitle">true</item>
</style>
```

También se puede ocultar la barra de acción en tiempo de ejecución llamando al método hide(), y volverlo a mostrar llamando al método show(). Por ejemplo:

```
ActionBar actionBar = getActionBar();
actionBar.hide();
```

Cuando la barra de acción se oculta, el sistema ajusta el contenido de la actividad para rellenar todo el espacio disponible de la pantalla.

Nota: Si se elimina la barra de acción utilizando una temática, la ventana no permitirá la barra de acción en ningún caso, por lo que no se puede añadir en tiempo de ejecución—si se llama al método getActionBar() devolverá null.

Añadir ítems de acción

Un ítem de acción es simplemente un ítem de menú del menú de opciones que debería aparecer en la barra de acción. Un ítem de acción puede incluir un icono y/o texto. Si un ítem de menú no aparece como un ítem de acción, el sistema lo coloca en un overflow menu, que el usuario puede abrir con el icono del menú en la parte derecha de la barra de acción.



Figura 2. Pantallazo de una barra de acción con dos ítems de acción y el overflow menu.

Cuando la actividad se inicia, el sistema rellena la barra de acción y el overflow menu llamando al método onCreateOptionsMenu(). Tal y como se detalla en la sección Crear menús, es en este método callback donde se define el menú de opciones para la actividad.

Se puede hacer que un ítem del menú aparezca como un ítem de acción—si existe lugar para él—desde el recurso de menú mediante la declaración android:showAsAction="ifRoom" para el elemento <item>. De esta manera, el ítem del menú aparece como acceso directo en la barra de acción sólo si existe lugar para ello. Si no hay suficiente lugar, el ítem se coloca en el overflow menu (mostrado por el icono del menú en la parte derecha de la barra de acción).

También se puede declarar un ítem del menú para que aparezca como un ítem de acción en el código de la aplicación, llamando al método setShowAsAction() en el MenuItem y pasándole SHOW_AS_ACTION_IF_ROOM.

Si el ítem del menú tiene un título y un icono, el ítem de acción, por defecto sólo muestra el icono. Si se quiere incluir el texto con el ítem de acción, hay que añadir el flag "with text": en el XML, añadir withText al atributo android:showAsAction o, utilizar en el código de la aplicación el flag SHOW_AS_ACTION_WITH_TEXT cuando se llama al método setShowAsAction().

La figura 2 muestra una barra de acción que tiene dos ítems de acción con su texto y el icono para el overflow menu.

A continuación tenemos un ejemplo de cómo se puede declarar un ítem de menú como un ítem de acción en un archivo recurso de menú:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_add"
        android:icon="@drawable/ic_menu_save"
        android:title="@string/menu_save"
        android:showAsAction="ifRoom|withText" />
</menu>
```

En este caso, los dos flags `ifRoom` y `withText` están seteados, por lo que cuando este ítem aparece como un ítem de acción, incluye el texto del título junto con el icono.

Un ítem de menú colocado en la barra de acción dispara los mismos métodos callback que otros ítems del menú de opciones. Cuando el usuario selecciona un ítem de acción, la actividad recibe una llamada al método `onOptionsItemSelected()`, pasándole el ítem ID.

Nota: Si se añade un ítem del menú desde un fragmento, se llama al método `onOptionsItemSelected()` correspondiente para ese fragmento. Si la actividad lo maneja primero, el sistema llama al método `onOptionsItemSelected()` de la actividad antes de llamar al fragmento.

También se puede declarar un ítem para que aparezca siempre como un ítem de acción, pero se debe evitar, ya que puede crear una IU recargada debido a la presencia de demasiados ítems de acción y pueden llegar a chocar con otros elementos de la barra de acción.

Para más información sobre los menús, ver el documento sobre [Crear menús](#).

Utilizar el icono de la aplicación como un ítem de acción

Por defecto, el icono de la aplicación aparece en la barra de acción en el lado izquierdo. También responde a la interacción del usuario (cuando el usuario lo pulsa, responde visualmente de la misma manera que los ítems de acción) y es nuestra responsabilidad hacer algo cuando el usuario lo pulsa.

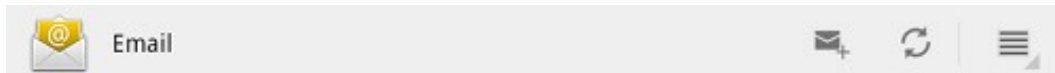


Figura 3. Barra de acción de correo electrónico, con el icono de la aplicación a la izquierda.

Cuando el usuario pulsa el icono, el comportamiento normal de la aplicación sería volver a la actividad "home" o al estado inicial (como cuando la actividad no ha cambiado, pero los fragmentos si). Si el usuario ya está en HOME o en el estado inicial, no es necesario hacer nada.

Cuando el usuario pulsa el icono, el sistema llama al método `onOptionsItemSelected()` de la actividad con el `IDandroid.R.id.home`. Hay que añadir una condición al método `onOptionsItemSelected()` para escuchar `android.R.id.home` y realizar la acción apropiada, como iniciar la actividad home o sacar de la pila transacciones recientes del fragmento.

Si al pulsar el icono de la aplicación se vuelve a la actividad home, se debería incluir el flag `FLAG_ACTIVITY_CLEAR_TOP` en el `Intent`. Con este flag, si la actividad que se está iniciando ya existe en la tarea en curso, entonces todas las actividades por encima de ella son eliminadas y es traída adelante. Se debe favorecer esta acción, ya que ir "home" es una acción que es equivalente a "ir atrás" y no se debería crear una nueva instancia de la actividad home. De lo contrario, se puede acabar con una larga pila de actividades en la tarea en curso.

Aquí, tenemos un ejemplo de la implementación de `onOptionsItemSelected()` que vuelve a la actividad "home" de la aplicación:


```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case android.R.id.home:
            // icono de la aplicación pulsado en la barra de acción; ir a inicio
            Intent intent = new Intent(this, HomeActivity.class);
            intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
            startActivity(intent);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}

```

Utilizar el icono de la aplicación para navegar "up"

También se puede utilizar el icono de la aplicación para proporcionar la navegación "up" al usuario. Esto es especialmente útil cuando la aplicación está compuesta de actividades que generalmente aparecen en un orden determinado y se quiere facilitar al usuario la navegación hacia arriba en la jerarquía de la actividad (sin tener en cuenta como se entró en la actividad en curso).

La forma de responder a este evento es la misma que cuando se navega al inicio (como se detalla anteriormente, exceptuando que se inicia una actividad diferente, basada en la actividad en curso). Todo lo que hay que hacer para indicar al usuario que el comportamiento es diferente es setear la barra de acción a "show home as up." ("mostrar inicio como arriba."); Se puede hacer llamando al método `setDisplayHomeAsUpEnabled(true)` en la `ActionBar` de la actividad. Cuando se hace esto, el sistema dibuja el icono de la aplicación con una flecha indicando el comportamiento arriba, tal y como se muestra en la figura 4.

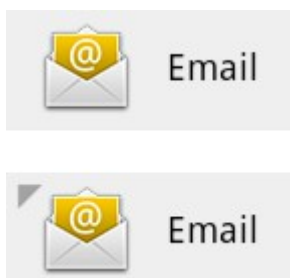


Figura 4. El icono estándar para la aplicación de email (arriba) y el icono "up" (abajo).

Aquí tenemos un ejemplo de como se puede mostrar el icono de la aplicación como una acción "up":

```

@Override
protected void onStart() {
    super.onStart();
    ActionBar actionBar = this.getActionBar();
    actionBar.setDisplayHomeAsUpEnabled(true);
}

```

La actividad, después de esto, debería responder cuando el usuario pulsa el icono desde el `onOptionsItemSelected()`, buscando el ID `android.R.id.home` (como se muestra anteriormente). En este caso, cuando se navega hacia arriba, es aún más importante utilizar el flag `FLAG_ACTIVITY_CLEAR_TOP` en el `Intent`, para que no se cree una nueva instancia de la actividad padre si ya existe.

Añadir un Action View

Un action view es un widget que aparece en la barra de acción como un sustituto de un ítem de acción. Si, por ejemplo, tenemos un ítem en el menú de opciones para "Search", se puede añadir un action view para el ítem que proporciona un widget SearchView en la barra de acción cuando el ítem se habilita como ítem de acción.

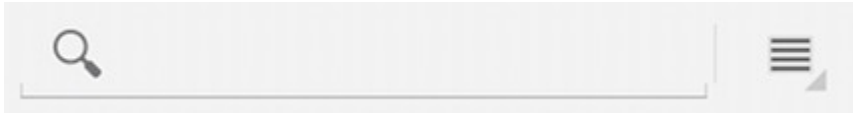


Figura 5. Un action view con un widget SearchView.

Cuando se añade un action view para un ítem de menú, es importante que se permita al ítem comportarse como un ítem de menú normal cuando no aparece en la barra de acción. Un ítem del menú, para realizar una búsqueda por ejemplo, debe, por defecto, mostrar un diálogo Android de búsqueda, pero si el ítem se coloca en la barra de acción, el action view aparece con un widget SearchView.

La mejor manera de declarar un action view para un ítem es hacerlo en el recurso del menú, utilizando los atributos `android:actionLayout` o `android:actionViewClass`:

- El valor para `android:actionLayout` debe de ser un puntero a un recurso de tipo archivo layout. Por ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_search"
        android:title="Search"
        android:icon="@drawable/ic_menu_search"
        android:showAsAction="ifRoom"
        android:actionLayout="@layout/searchview" />
</menu>
```

- El valor para `android:actionViewClass` debe de ser un nombre completo de la clase para el View que se quiere usar. Por ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_search"
        android:title="Search"
        android:icon="@drawable/ic_menu_search"
        android:showAsAction="ifRoom"
        android:actionViewClass="android.widget.SearchView" />
</menu>
```

Se debe incluir `android:showAsAction="ifRoom"` para que el ítem aparezca como un action view cuando haya espacio. Si es necesario, se puede forzar el ítem a aparecer siempre como un action view seteando `android:showAsAction` a "always".

Cuando el ítem del menú se muestra como un action item, su action view aparece en vez del icono y/o del título. Sin embargo, si no hay suficiente espacio en la barra de acción, el ítem aparece en el overflow menu como un ítem de menú normal y su respuesta debe de ser desde el método callback onOptionsItemSelected().

Cuando la actividad se inicia, el sistema rellena la barra de acción y el overflow menu llamando al onCreateOptionsMenu(). Después de inflar el menú en este método, se pueden conseguir elementos en un action view (para quizá adjuntar listeners) llamando al método findItem() con el ID del ítem de menú y después getActionView() en el MenuItem devuelto. Por ejemplo, el widget de búsqueda de los ejemplos anteriores se consigue de esta manera:

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.options, menu);
    SearchView searchView = (SearchView) menu.findItem(R.id.menu_search).getActionView();
    // setear los listeners apropiados para searchView
    ...
    return super.onCreateOptionsMenu(menu);
}

```

Añadir pestañas

La barra de acción puede mostrar pestañas que permiten al usuario navegar entre diferentes fragmentos de la actividad. Cada pestaña puede tener un título y/o un icono.



Figura 6. Pantallazo de pestañas en la barra de acción, de la aplicación Honeycomb Gallery.

El layout debe incluir un View en el que se muestra cada Fragment asociado con una pestaña. El view debe de tener un ID para referenciarlo desde el código.

Para añadir pestañas a la barra de acción:

- Crear una implementación de ActionBar.TabListener para gestionar los eventos de interacción de las pestañas de la barra de acción. Se deben implementar todos los métodos: onTabSelected(), onTabUnselected(), y onTabReselected(). Cada método callback pasa el ActionBar.Tab que ha recibido el evento y un FragmentTransaction para que se realice las transacciones de fragmento (añadir o eliminar fragmentos). Por ejemplo:

```

private class MyTabListener implements ActionBar.TabListener {
    private TabContentFragment mFragment;

    // Se llama para crear una instancia del listener cuando se añade una nueva pestaña
    public MyTabListener(TabContentFragment fragment) {
        mFragment = fragment;
    }
    public void onTabSelected(Tab tab, FragmentTransaction ft) {
        ft.add(R.id.fragment_content, mFragment, null);
    }
    public void onTabUnselected(Tab tab, FragmentTransaction ft) {
        ft.remove(mFragment);
    }
    public void onTabReselected(Tab tab, FragmentTransaction ft) {
        // no hacer nada
    }
}

```

- Esta implementación de ActionBar.TabListener añade un constructor que guarda el Fragment asociado a una pestaña para que cada callback pueda añadir o eliminar ese fragmento.
- Conseguir el ActionBar, llamando al método getActionBar() desde la Actividad durante onCreate() (hay que asegurarse que se hace después de llamar a setContentview()).
- Llamar a setNavigationMode(NAVIGATION_MODE_TABS) para habilitar el modo pestaña para ActionBar.
- Crear cada pestaña para la barra de acción:

- Crear un nuevo ActionBar.Tab llamando al método newTab() en el ActionBar.
- Añadir título y/o un icono para la pestaña llamando al método setText() y/o setIcon().
Truco: Estos métodos devuelven la misma instancia de ActionBar.Tab, por lo que se pueden encadenar las llamadas.
- Declarar el ActionBar.TabListener para usarlo en la pestaña, pasándole una instancia de la implementación del método setTabListener().
- Añadir cada ActionBar.Tab a la barra de acción llamando al método addTab() en el ActionBar y pasándole el ActionBar.Tab.
- Ejemplo: El siguiente código combina los pasos 2-5 para crear dos pestañas y añadirlas a la barra de acción:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    // configuración para pestañas de la barra de acción
    final ActionBar actionBar = getActionBar();
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
    // eliminar el título de la actividad para crear espacios para pestañas
    actionBar.setDisplayShowTitleEnabled(false);
    // instanciar un fragmento para la pestaña
    Fragment artistsFragment = new ArtistsFragment();
    // añadir una nueva pestaña y setear el título y el listener
    actionBar.addTab(actionBar.newTab().setText(R.string.tab_artists)
        .setTabListener(new TabListener(artistsFragment)));
    Fragment albumsFragment = new AlbumsFragment();
    actionBar.addTab(actionBar.newTab().setText(R.string.tab_albums)
        .setTabListener(new TabListener(albumsFragment)));
}
```

Todo lo que ocurre cuando se selecciona una pestaña debe ser definido en los métodos callback del ActionBar.TabListener. Cuando se selecciona una pestaña, recibe una llamada al método onTabSelected() y es aquí donde se debe añadir el fragmento apropiado al view correspondiente del layout, utilizando add() con el FragmentManager proporcionado. Cuando, una pestaña es deseleccionada (porque otra pestaña se ha seleccionado), se debe eliminar el fragmento del layout, utilizando remove().

Precaución: No se debe llamar al método commit() para estas transacciones—el sistema ya lo hace y se puede lanzar una excepción si se llama desde nuestro código. No se debe añadir estas transacciones de fragmento al back stack.

Si se para la actividad, se debe guardar la pestaña seleccionada para que cuando el usuario vuelva a la aplicación, se pueda abrir la pestaña. Cuando sea momento de guardar el estado, se puede hacer una consulta a la pestaña seleccionada en curso con el método getSelectedNavigationIndex(). Este método devuelve la posición en el índice de la pestaña seleccionada.

Precaución: Es importante que se guarde el estado de cada fragmento, para que cuando el usuario intercambie fragmentos con las pestañas, y después vuelva a un fragmento previo, aparezca como lo dejaron. Para más información sobre como guardar el estado del fragmento, ver la sección Fragmentos.

Añadir navegación desplegable

Se puede proporcionar una lista desplegable en la barra de acción como otro modo de navegación en la actividad. Ejemplo; la lista desplegable puede proporcionar modos alternativos para ordenar el contenido en la actividad o para cambiar la cuenta del usuario.

A continuación tenemos una lista de pasos para habilitar la navegación desplegable:

- Crear un [SpinnerAdapter](#) que proporciona la lista de items seleccionables para usar por el layout y el desplegable cuando se dibuja cada item de la lista.
- Implementar [ActionBar.OnNavigationItemSelectedListener](#) para definir el comportamiento cuando el usuario selecciona un item de la lista.
- Habilitar el modo navegación para la barra de acción con [setNavigationMode\(\)](#). Por ejemplo:

```
ActionBar actionBar = getActionBar();
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
```

Nota: Esto se debe realizar durante el método [onCreate\(\)](#) de la actividad.

- Setear el callback para la lista desplegable con [setListNavigationCallbacks\(\)](#). Por ejemplo:

```
actionBar.setListNavigationCallbacks(mSpinnerAdapter, mNavigationCallback);
```

Este método tiene como parámetro [SpinnerAdapter](#) y [ActionBar.OnNavigationItemSelectedListener](#).

Esta es la configuración básica. En la implementación de [SpinnerAdapter](#) y [ActionBar.OnNavigationItemSelectedListener](#) es donde se hace la mayor parte del trabajo. Existen muchas maneras de implementarlos para definir la funcionalidad de la navegación desplegable. Esta implementación de varios tipos de [SpinnerAdapter](#) está fuera del alcance de este documento (para más información ver la clase [SpinnerAdapter](#)).

Estilismo de la barra de acción

La barra de acción es el encabezamiento de la aplicación y un punto de interacción primario para el usuario, por lo que es importante poder modificar el diseño para integrarlo con el diseño de la aplicación. Existen varias maneras de hacerlo.

Para modificaciones simples del [ActionBar](#), se pueden utilizar los siguientes métodos:

[setBackgroundDrawable\(\)](#)

Setea un drawable (gráfico) para usar como fondo de la barra de acción. El gráfico debe ser una imagen [Nine-patch](#), una [forma](#), o un [color sólido](#), para que el sistema cambie el tamaño según el tamaño de la barra de acción (no se debería utilizar una imagen bitmap con tamaño fijo).

[setDisplayUseLogoEnabled\(\)](#)

Habilita el uso de una imagen alternativa en la barra de acción (un "logo"), en vez del icono por defecto de la aplicación. Un logo suele ser una imagen más detallada y más ancha que representa a la aplicación. Cuando está habilitado, el sistema utiliza la imagen del logo definida para la aplicación (o para una actividad individual) en el archivo manifest, con el atributo [android:logo](#). Se cambiará el tamaño del logo si es necesario para que tenga la misma altura que la barra de acción. (La mejor práctica es diseñar el logo con el mismo tamaño que el icono de la aplicación).

Para personalizaciones más complejas, se puede usar el framework de Android estilos y temas para cambiar el estilo de la barra de acción.

La barra de acción tiene dos temas estándar, "oscuro" y "claro". El oscuro se aplica con el tema holográfico por defecto, como se especifica en el [Theme.Holo](#). Si se quiere un fondo blanco con texto oscuro, se puede aplicar a la actividad, el tema [Theme.Holo.Light](#) en el archivo manifest. Por ejemplo:

```
<activity android:name=".ExampleActivity"
    android:theme="@android:style/Theme.Holo.Light" />
```

Para más control, se puede sobrescribir bien el tema [Theme.Holo](#) o bien el tema [Theme.Holo.Light](#) y aplicar estilos customizados a determinados aspectos de la barra de acción.

Se pueden customizar algunas propiedades de la barra de acción, como por ejemplo:

android:actionBarTabStyle

Estilo para las pestañas de la barra de acción.

android:actionBarTabBarStyle

Estilo para la barra que aparece debajo de las pestañas en la barra de acción.

android:actionBarTabTextStyle

Estilo para el texto en las pestañas.

android:actionDropDownStyle

Estilo para la lista desplegable utilizada por el overflow menu y la navegación desplegable.

android:actionButtonStyle

Estilo para la imagen de fondo utilizada por los botones de la barra de acción.

A continuación tenemos un ejemplo de un archivo recurso que define un tema customizado para la barra de acción, según el tema estándar Theme.Holo:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- tema aplicado a la aplicación o actividad -->
    <style name="CustomActionBar" parent="android:style/Theme.Holo.Light">
        <item name="android:actionBarTabTextStyle">@style/customActionBarTabTextStyle</item>
        <item name="android:actionBarTabStyle">@style/customActionBarTabStyle</item>
        <item name="android:actionBarTabBarStyle">@style/customActionBarTabBarStyle</item>
    </style>
    <!-- estilo para el texto de la pestaña -->
    <style name="customActionBarTabTextStyle">
        <item name="android:textColor">#2966c2</item>
        <item name="android:textSize">20sp</item>
        <item name="android:typeface">sans</item>
    </style>
    <!-- estilo para las pestañas -->
    <style name="customActionBarTabStyle">
        <item name="android:background">@drawable/actionbar_tab_bg</item>
        <item name="android:paddingLeft">20dp</item>
        <item name="android:paddingRight">20dp</item>
    </style>
    <!-- estilo para barra de pestañas -->
    <style name="customActionBarTabBarStyle">
        <item name="android:background">@drawable/actionbar_tab_bar</item>
    </style>
</resources>
```

Nota: Para cambiar la imagen de fondo de la pestaña, dependiendo del estado de la pestaña en curso (seleccionada, pulsada, no seleccionada), el recurso dibujable utilizado debe ser un state list drawable. Hay que asegurarse de que el tema customizado declare un tema padre, desde el que hereda todos los estilos que no están declarados explícitamente en el tema.

Se puede aplicar en el archivo manifest, el tema customizado a toda la aplicación o a actividades individuales de esta manera:

```
<application android:theme="@style/CustomActionBar"
    ... />
```

Si se quiere crear un tema customizado para la actividad que elimine completamente la barra de acción, hay que utilizar los siguientes atributos de estilo:

android:windowActionBar

Para eliminar la barra de acción, setear esta propiedad de estilo a false.

android:windowNoTitle

Para eliminar también el título de la barra, setear esta propiedad de estilo a true.

Mostrar un Diálogo en Android

Qué es un Diálogo

Un diálogo es una pequeña ventana que aparece delante de la actividad en curso. La actividad que está detrás pierde el focus y el diálogo es el que recibe todas las interacciones con el usuario. Los diálogos son utilizados para las notificaciones que deben interrumpir al usuario y para realizar tareas cortas que se relacionan directamente con la aplicación en curso (como una barra de progreso o un login prompt).

La clase Dialog es la clase base para crear diálogos. Sin embargo, no se debe instanciar directamente un Dialog. Se debe utilizar una de las siguientes subclases:

AlertDialog

Un diálogo que puede manejar cero, uno, dos o tres botones y/o una lista de items seleccionables que pueden incluir casillas de verificación (checkboxes) o botones de opción (radio buttons). El AlertDialog es capaz de construir la mayoría de los diálogos de interfaz de usuario y es el tipo de diálogo más recomendado.

ProgressDialog

Un diálogo que muestra una barra de progreso o una cursor de espera. Como es una extensión de AlertDialog, también soporta botones.

DatePickerDialog

Un diálogo que permite al usuario seleccionar una fecha.

TimePickerDialog

Un diálogo que permite al usuario seleccionar una hora.

Si se quiere personalizar un diálogo, se puede extender del objeto base Dialog o de cualquiera de las subclases de la lista anterior y definir un nuevo layout.

Mostrar un diálogo

Un diálogo siempre es creado y mostrado como parte de un Activity. Siempre se deberían crear los diálogos dentro del método callback onCreateDialog(int) del Activity. Cuando se utiliza este callback, el sistema Android automáticamente maneja el estado de cada diálogo y los engancha al Activity, haciéndolo el "dueño" de cada diálogo. De esa manera, cada diálogo hereda determinadas propiedades del Activity. Ejemplo; cuando se abre un diálogo, la tecla Menú nos muestra el menú de opciones definido para el Activity y las teclas del sonido modifican el audio usado por el Activity.

Nota: Si se decide crear un diálogo fuera del método onCreateDialog(), no estará unido al Activity. Se puede unir al Activity con el método setOwnerActivity(Activity).

Cuando se quiere mostrar un diálogo, hay que llamar al método showDialog(int) y pasarle un integer que identifica unívocamente al diálogo que se quiere mostrar.

Cuando se pide un diálogo por primera vez, Android llama al método onCreateDialog(int) del Activity, que es donde se debe instanciar al Dialog. A este método callback se le pasa el mismo ID que se le había pasado a showDialog(int). Después de crear el Dialog, devuelve el objeto al final del método.

Antes de que el diálogo se muestre, Android llama también al método callback opcional onPrepareDialog(int, Dialog). Hay que definir este método si se quiere cambiar cualquiera de las propiedades del diálogo cada vez que se abre. Se llama a este método cada vez que se abre el diálogo, mientras que onCreateDialog(int) sólo se llama la primera vez que se abre el diálogo. Si no se define el método onPrepareDialog(), el diálogo permanecerá igual que la última vez que se abrió.

A este método también se le pasa el ID del diálogo, junto con el objeto Dialog que se ha creado en `onCreateDialog()`.

La mejor manera de definir los métodos callback `onCreateDialog(int)` y `onPrepareDialog(int, Dialog)` es con una sentencia switch que chequea el parámetro id que se pasa al método. Cada caso debe chequear con un ID único de diálogo y después crear y definir el Dialogo correspondiente. Ejemplo; un juego que utiliza dos diálogos diferentes: uno para indicar que el juego está en pausa y otro para indicar que el juego ha terminado. Primero, hay que definir un ID integer para cada diálogo:

```
static final int DIALOG_PAUSED_ID = 0;
static final int DIALOG_GAMEOVER_ID = 1;
```

Después, definir el callback `onCreateDialog(int)` con un switch case para cada ID:

```
protected Dialog onCreateDialog(int id) {
    Dialog dialog;
    switch(id) {
        case DIALOG_PAUSED_ID:
            // hacer el trabajo para definir el Dialogo pausa
            break;
        case DIALOG_GAMEOVER_ID:
            // hacer el trabajo para definir el Dialogo juego terminado
            break;
        default:
            dialog = null;
    }
    return dialog;
}
```

Nota: En este ejemplo, no hay código dentro de las sentencias case porque el procedimiento para definir el Diálogo está fuera del alcance de esta sección.

Cuando sea el momento de mostrar cualquiera de los diálogos, llamar al método `showDialog(int)` con el ID del diálogo:

```
showDialog(DIALOG_PAUSED_ID);
```

Cerrar un Diálogo

Antes de cerrar un diálogo, se puede desechar llamando al método `dismiss()` del objeto Dialog. Si fuera necesario también se puede llamar al método `dismissDialog(int)` de la clase Activity, que llama al método `dismiss()` del Dialog.

Si se está utilizando el método `onCreateDialog(int)` para gestionar el estado de los diálogos (como hemos visto en la sección anterior), cada vez que se desecha el diálogo, el Activity retiene el estado del objeto Dialog. Si se decide que ya no se necesita este objeto o que es necesario limpiar el estado, se debe llamar al método `removeDialog(int)`. Esto eliminará cualquier referencia interna al objeto y si se está mostrando el diálogo, lo desechará.

Utilizar dismiss listeners

Si se quiere que la aplicación realice determinadas funciones en el momento en el que cierra un diálogo, se debe unir un dismiss listener al diálogo.

Primero hay que definir la interfaz `DialogInterface.OnDismissListener`. Esta interfaz sólo tiene un método, `onDismiss(DialogInterface)`, que será llamado cuando el diálogo sea cerrado. Después hay que pasar la implementación del OnDismissListener al método `setOnDismissListener()`.

Los diálogos también pueden ser "cancelados." Este es un caso especial en el que se indica que el

diálogo ha sido explícitamente cancelado por el usuario. Esto ocurrirá cuando el usuario pulse el botón "back" para cerrar el diálogo, o si el diálogo llama explícitamente al método `cancel()` (quizá desde un botón "Cancel" del diálogo). Cuando se cancela un diálogo, se avisa igualmente al `OnDismissListener`, pero si quiere ser informado de que el diálogo ha sido explícitamente cancelado (y no desechado normalmente), entonces hay que registrar un `DialogInterface.OnCancelListener` con un `setOnCancelListener()`.

Crear un AlertDialog

Un `AlertDialog` es una extensión de una clase `Dialog`. Es capaz de construir la mayoría de los diálogos de interfaz de usuario y es el tipo de diálogo más recomendado. Se debe utilizar para diálogos que usan las siguientes propiedades:

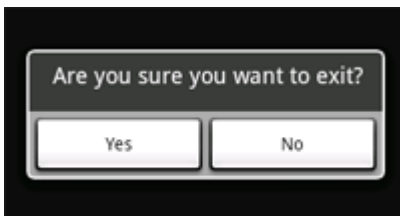
- Un título
- Un mensaje de texto
- Uno, dos o tres botones
- Una lista de items seleccionables (con casillas de verificación (checkboxes) o botones de opción (radio buttons) opcionales)

Para crear un `AlertDialog`, utilizar la subclase `AlertDialog.Builder`. Obtener un Builder con `AlertDialog.Builder(Context)` y después usar los métodos públicos de la clase para definir todas las propiedades del `AlertDialog`. Después de usar el Builder, hay que recuperar el objeto `AlertDialog` con el método `create()`.

Los siguientes temas muestran como definir varias propiedades del `AlertDialog` utilizando la clase `AlertDialog.Builder`. Si se usa cualquiera de los siguientes códigos dentro del método callback `onCreateDialog()`, se puede devolver el objeto `Dialog` resultante para mostrar el diálogo.

Añadir botones

Para crear un `AlertDialog` con botones a cada lado como el mostrado a continuación, hay que utilizar los métodos `set...Button()`:



```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Are you sure you want to exit?")
    .setCancelable(false)
    .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            MyActivity.this.finish();
        }
    })
    .setNegativeButton("No", new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            dialog.cancel();
        }
    });
AlertDialog alert = builder.create();
```

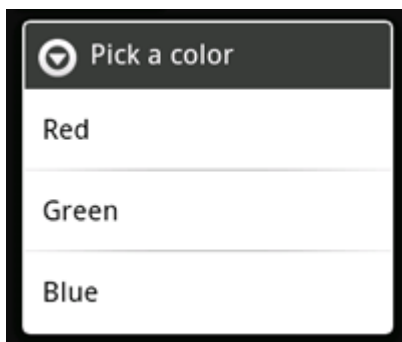
Primero, añadir un mensaje que vaya en el diálogo con `setMessage(CharSequence)`. Después, empezar con el encadenamiento de métodos y setear el diálogo para que sea no cancelable (así el usuario no podrá cerrar el diálogo con el botón back) con el método `setCancelable(boolean)`. Para cada botón, utilizar uno de los métodos `set...Button()` como el `setPositiveButton()`, que acepta el

nombre del botón y un [DialogInterface.OnClickListener](#) que define la acción a realizar cuando el usuario seleccione un botón.

Nota: Sólo se puede añadir un tipo de botón al AlertDialog. Esto significa que, no se puede tener más de un botón "positivo". Esto limita a tres el número de botones posibles: positivo, neutro y negativo. Estos nombres son irrelevantes técnicamente para la funcionalidad actual de los botones, pero pueden ayudar para saber cual hace que.

Añadir una lista

Para crear un AlertDialog con una lista de items seleccionables como la que se ve a continuación, hay que utilizar el método `setItems()`:



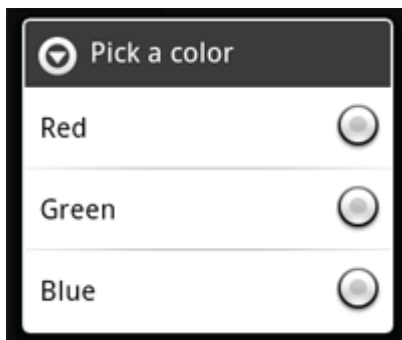
```
final CharSequence[] items = {"Red", "Green", "Blue"};

AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Pick a color");
builder.setItems(items, new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int item) {
        Toast.makeText(getApplicationContext(), items[item], Toast.LENGTH_SHORT).show();
    }
});
AlertDialog alert = builder.create();
```

Primero, añadir un título al diálogo con `setTitle(CharSequence)`. Después, añadir una lista de items seleccionables con `setItems()`, que acepta el array de items que va a mostrar y un [DialogInterface.OnClickListener](#) que define la acción a realizar cuando el usuario selecciona un item.

Añadir casillas de verificación (checkboxes) o botones de opción (radio buttons)

Para crear una lista de items con opción-múltiple (casillas de verificación) o items de una sola opción (botones de opción) dentro del diálogo, hay que utilizar los métodos `setMultiChoiceItems()` y `setSingleChoiceItems()` respectivamente. Si se crea una de estas listas seleccionables en el método callback `onCreateDialog()`, Android gestiona el estado de la lista. Mientras el Activity esté activo, el diálogo recuerda los items que han sido previamente seleccionados, pero cuando el usuario sale del Activity, la selección se pierde.



Nota: Para guardar la selección cuando el usuario abandona o pone en pausa el Activity, se debe guardar y restablecer la configuración a través del ciclo de vida de la actividad. Para guardar de manera permanente las selecciones, aún cuando el proceso del Activity está completamente parado, hay que guardar las configuraciones con alguna de las técnicas de almacenaje de datos.

Para crear un AlertDialog con una lista de items de una sola opción como el mostrado a la derecha, hay que utilizar el mismo código del ejemplo previo, pero reemplazando el método `setItems()` con el método `setSingleChoiceItems()`:

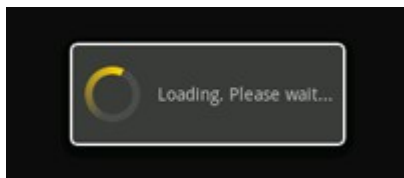
```
final CharSequence[] items = {"Red", "Green", "Blue"};

AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Pick a color");
builder.setSingleChoiceItems(items, -1, new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int item) {
        Toast.makeText(getApplicationContext(), items[item], Toast.LENGTH_SHORT).show();
    }
});
AlertDialog alert = builder.create();
```

El segundo parámetro en el método `setSingleChoiceItems()` es un integer para el método `checkedItem`, que indica la posición de la lista por defecto del item seleccionado. Hay que utilizar "-1" para indicar que ningún item debería ser seleccionado por defecto.

Crear un ProgressDialog

Un ProgressDialog es una extensión de la clase AlertDialog que puede mostrar una animación representando el progreso en forma de por ejemplo una rueda para una tarea con un progreso no definido, o una barra de progreso para una tarea con una progresión definida. El diálogo puede tener botones, como uno para cancelar la descarga.



Abrir un diálogo de progreso puede ser tan sencillo como llamar al método `ProgressDialog.show()`. Ejemplo; el diálogo de progreso mostrado a la derecha se consigue sin tener que gestionar el diálogo a través del método callback `onCreateDialog(int)` como se muestra a continuación:

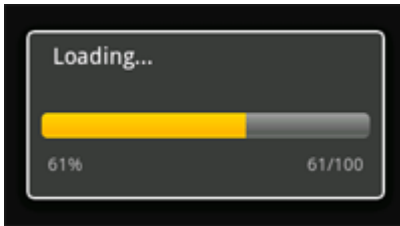
```
ProgressDialog dialog = ProgressDialog.show(MyActivity.this, "",
    "Loading. Please wait...", true);
```

El primer parámetro es la aplicación Context, el segundo es un título para el diálogo (se queda vacío), el tercero es el mensaje, y el último parámetro es si el progreso es indeterminado (sólo es relevante cuando se quiere crear una barra de progreso; que se hablará en la sección siguiente).

El estilo por defecto de un diálogo de progreso es una rueda rodando. Si se quiere crear una barra de progreso que muestre el progreso de carga con granulación hay que escribir un poco más de código, como se ve en la sección siguiente.

Mostrar una barra de progreso

Para mostrar la progresión con una barra de progreso animada:



1. Inicializar el ProgressDialog con el constructor de la clase, `ProgressDialog(Context)`.
2. Setear el estilo del progreso a "STYLE_HORIZONTAL" con el método `setProgressStyle(int)` y setear cualquier otra propiedad, como el mensaje.
3. Cuando se quiera mostrar el diálogo hay que llamar al método `show()` o devolver el ProgressDialog del callback `onCreateDialog(int)`.
4. Se puede incrementar la cantidad de progreso mostrado en la barra llamando bien al método `setProgress(int)` con un valor del porcentaje completado en ese momento o bien al método `incrementProgressBy(int)` con un valor incremental que se añade al porcentaje total completado en ese momento.

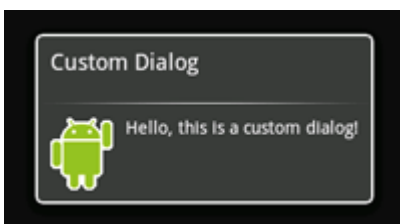
Ejemplo;

```
ProgressDialog progressDialog;
progressDialog = new ProgressDialog(mContext);
progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
progressDialog.setMessage("Loading...");
progressDialog.setCancelable(false);
```

La mayoría del código que se necesita para crear un diálogo de progreso está involucrado en el proceso que lo actualiza. Puede ser necesario crear un segundo hilo en la aplicación para que realice este trabajo y para que después informe del progreso al hilo del Activity de la IU con un objeto `Handler`.

Crear un Dialog personalizado

Si se quiere un diseño personalizado para el diálogo, se puede crear un layout propio para la ventana del diálogo con elementos de layout y de widget. Después de definir el layout, hay que pasar el objeto View o el ID del layout al método `setContentView(View)`.



Como ejemplo, aquí tenemos como crear el diálogo mostrado arriba:

1. Crear un layout XML guardado como `custom_dialog.xml`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout_root"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    >
```

```

<ImageView android:id="@+id/image"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:layout_marginRight="10dp"
/>
<TextView android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:textColor="#FFF"
/>
</LinearLayout>

```

Este XML define un ImageView y un TextView dentro de un LinearLayout.

2. Hay que setear el layout que acabamos de ver como el content view del diálogo y definir el contenido para los elementos ImageView y TextView:

```

Context mContext = getApplicationContext();
Dialog dialog = new Dialog(mContext);
dialog setContentView(R.layout.custom_dialog);
dialog.setTitle("Custom Dialog");
TextView text = (TextView) dialog.findViewById(R.id.text);
text.setText("Hello, this is a custom dialog!");
ImageView image = (ImageView) dialog.findViewById(R.id.image);
image.setImageResource(R.drawable.android);

```

Después de instanciar el Dialog, hay que setear el layout personalizado como el content view del diálogo con el método setContentView(int), pasándole el ID del layout. Ahora que el Dialog tiene un layout definido, se puede capturar los objetos View del layout con el método findViewById(int) y modificar su contenido.

3. Ahora ya se puede mostrar el diálogo tal y como se describe en Mostrar un Diálogo.

Un diálogo hecho con la clase base Dialog debe tener un título. Si no se llama al método setTitle(), el espacio utilizado por el título permanece vacío, pero visible. Si no se quiere un título, entonces hay que crear un diálogo personalizado utilizando la clase AlertDialog. Un AlertDialog se crea más fácilmente con la clase AlertDialog.Builder, pero en este caso no se tiene acceso al método setContentView(int) utilizado anteriormente. Sin embargo, se puede utilizar el método setView(View). Este método acepta un objeto View, por lo que se necesita inflar el objeto raíz View del layout desde el XML.

Para inflar el XML del layout, hay que recuperar el LayoutInflater con el método getLayoutInflater() (o el método getSystemService()), y después llamar al método inflate(int, ViewGroup), donde el primer parámetro es el ID del layout y el segundo es el ID del View raíz. En este punto, se puede utilizar el layout inflado para encontrar los objetos View en el layout y definir el contenido para los elementos ImageView y TextView. Después hay que instanciar el AlertDialog.Builder y setear el layout inflado para el diálogo con el setView(View).

A continuación tenemos un ejemplo de como crear un layout personalizado en un AlertDialog:

```

AlertDialog.Builder builder;
AlertDialog alertDialog;

Context mContext = getApplicationContext();
LayoutInflater inflater = (LayoutInflater) mContext.getSystemService(LAYOUT_INFLATER_SERVICE);
View layout = inflater.inflate(R.layout.custom_dialog,
    (ViewGroup) findViewById(R.id.layout_root));

TextView text = (TextView) layout.findViewById(R.id.text);
text.setText("Hello, this is a custom dialog!");

```

```
ImageView image = (ImageView) layout.findViewById(R.id.image);
image.setImageResource(R.drawable.android);

builder = new AlertDialog.Builder(mContext);
builder.setView(layout);
alertDialog = builder.create();
```

Eventos de Interfaz de Usuario

En Android, existe más de una manera de interceptar los eventos producidos por la interacción del usuario con la aplicación. Los eventos que se producen dentro de la interfaz de usuario, hay que capturarlos del objeto View específico con el que el usuario interacciona. La clase View proporciona los medios para hacerlo.

Dentro de las diferentes clases de View que se utilizan para componer el layout, existen varios métodos callback públicos que son útiles para los eventos de la IU. Estos métodos son llamados por el framework de Android cuando se lleva a cabo la acción correspondiente en ese objeto. Por ejemplo, cuando se pulsa un View (como un botón), se llama al método `onTouchEvent()` de ese objeto. Para poder interceptarlo, hay que extender la clase y sobrescribir el método. Hay que tener en cuenta que extender todos los objetos View para manejar un evento así no es práctico. Esta es la razón por la que la clase View tiene una colección de interfaces anidadas con callbacks que son más fácilmente definidas. Estas interfaces, llamadas listeners de eventos, son la manera correcta de capturar la interacción del usuario con la IU.

Normalmente se utilizan los listeners de eventos para capturar la interacción del usuario, pero en algún momento se puede necesitar extender una clase View para construir un componente personalizado. Se puede necesitar extender la clase `Button` para hacer algo más acorde a las necesidades. En este caso, se podrá definir los comportamientos por defecto del evento utilizando la clase handlers de eventos que veremos en este capítulo.

Listeners de Eventos

Un listener de un evento es una interfaz de la clase `View` que contiene un sólo método callback. Estos métodos serán llamados por el framework Android cuando el View al que se le ha registrado el listener sea activado mediante la interacción con el usuario a través del item de la IU.

Los siguientes métodos callback están incluidos en las interfaces de listener de eventos:

`onClick()`

Se llama desde el `View.OnClickListener`. Se llama cuando el usuario bien pulsa el item (cuando está en modo táctil), o cuando pulsa el item con las teclas de navegación o con el trackball y pulsa la tecla "enter".

`onLongClick()`

Se llama desde el `View.OnLongClickListener`. Se llama cuando el usuario bien pulsa el item (cuando está en modo táctil), o cuando pulsa el item con las teclas de navegación o con el trackball y pulsa la tecla "enter" (por un segundo).

`onFocusChange()`

Se llama desde el `View.OnFocusChangeListener`. Se llama cuando el usuario navega por encima del item o cuando se aleja, utilizando las teclas de navegación o el trackball.

`onKey()`

Se llama desde el `View.OnKeyListener`. Se llama cuando el usuario está con el focus en el item y pulsa o deja de pulsar una tecla del dispositivo.

onTouch()

Se llama desde el View.OnTouchListener. Se llama cuando el usuario realiza una cualquier evento que tenga que ver con pulsar, incluyendo pulsar, dejar de pulsar, o cualquier movimiento en la pantalla (dentro de los límites del item).

onCreateContextMenu()

Se llama desde el View.OnCreateContextMenuListener. Se llama cuando se está construyendo un menú contextual (como resultado de un "largo click") sostenido en el tiempo. Ver los detalles en los menús contextuales en Crear Menús.

Estos métodos son los únicos de sus respectivas interfaces. Para definir uno de estos métodos y manejar los eventos, hay que implementar la interfaz anidada en el Activity o definirla en una clase anónima. Después, hay que pasar una instancia de la implementación al método View.set...Listener() correspondiente. (Por ejemplo, llamar al método setOnClickListener() y pasarle la implementación del OnClickListener.)

El ejemplo a continuación muestra como hacer un on-click listener para un botón.

```
// Crear una implementación anónima de OnClickListener
private OnClickListener mCorkyListener = new OnClickListener() {
    public void onClick(View v) {
        // hacer algo cuando se pulsa el botón
    }
};

protected void onCreate(Bundle savedInstanceState) {
    ...
    // Capturar el botón desde el layout
    Button button = (Button)findViewById(R.id.corky);
    // registrar el listener onClick con la implementación antes realizada
    button.setOnClickListener(mCorkyListener);
    ...
}
```

También se puede implementar el OnClickListener como parte del Activity. Esto evitará la carga de la clase extra y la asignación del objeto. Por ejemplo:

```
public class ExampleActivity extends Activity implements OnClickListener {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        Button button = (Button)findViewById(R.id.corky);
        button.setOnClickListener(this);
    }

    // Implementar el callback OnClickListener
    public void onClick(View v) {
        // hacer algo cuando se pulsa el botón
    }
    ...
}
```

Obsérvese que el callback onClick() del ejemplo anterior no tiene valor de retorno, aunque otros métodos de listener de eventos tienen que devolver un boolean. Esto dependerá del evento. A continuación explicamos las razones en los casos en los que devuelven un boolean:

- onLongClick() - Devuelve un boolean para indicar si se ha manejado el evento o no. Se devuelve true para indicar que se ha manejado el evento y se debería parar; devuelve false si no se ha manejado el evento y/o el evento debería continuar en otro listener on-click.

- onKey() - Devuelve un boolean para indicar si se ha manejado el evento o no. Se devuelve true para indicar que se ha manejado el evento y se debería parar; devuelve false si no se ha manejado el evento y/o el evento debería continuar en otro listener on-key.

- onTouch() -Devuelve un boolean para indicar si se ha manejado el evento o no. Se devuelve true para indicar que se ha manejado el evento y se debería parar; devuelve false si no se ha manejado el evento y/o el evento debería continuar en otro listener.

Hay que recordar que los eventos de las teclas siempre son entregados al View que tiene el focus en ese momento. Se entregan primero empezando desde la parte superior de la jerarquía del View, hacia abajo, hasta que llegan el destino apropiado. Si el View(o un hijo del View) tiene en ese momento el focus, se puede ver como el evento viaja a través del método dispatchKeyEvent(). Como alternativa a capturar a los eventos de las teclas a través del View, se puede recibir todos los eventos dentro del Activity con los métodos onKeyDown() y onKeyUp().

Handlers de eventos

Si se está desarrollando un componente personalizado a partir de View, se puede definir métodos callback como handlers de eventos por defecto. A continuación tenemos los callbacks más comunes utilizados para manejar los eventos:

- onKeyDown(int, KeyEvent) - Se llama cuando ocurre un nuevo evento de teclado.

- onKeyUp(int, KeyEvent) - Se llama cuando se suelta una tecla.

- onTrackballEvent(MotionEvent) - Se llama cuando un ocurre un evento referente a cualquier movimiento del trackball.

- onTouchEvent(MotionEvent) - Se llama cuando se toca la pantalla.

- onFocusChanged(boolean, int, Rect) - Se llama cuando el view gana o pierde el focus.

Existen otros métodos que pueden impactar directamente en el manejo de los eventos, que no forman parte de la clase View:

- Activity.dispatchTouchEvent(MotionEvent) - Permite al Activity interceptar todos los eventos referentes a tocar la pantalla antes de que se reparten a la ventana.

- ViewGroup.onInterceptTouchEvent(MotionEvent) - Permite al ViewGroup controlar los eventos mientras se reparten a los Views hijos.

- ViewParent.requestDisallowInterceptTouchEvent(boolean) - Se llama sobre un View padre para indicar que no debería interceptar eventos táctiles con onInterceptTouchEvent(MotionEvent).

Modo Táctil

Cuando el usuario está navegando por la interfaz de usuario con teclas direccionales o con un trackball, es necesario darle el focus a los items accionables (como los botones) para que el usuario pueda ver qué va a aceptar el input. Si el dispositivo tiene capacidades táctiles y el usuario empieza interactuando con la interfaz tocándola, entonces no es necesario destacar los items, ni darle el focus a un View en particular. Este será el llamado "modo táctil."

En un dispositivo táctil, una vez que el usuario toca la pantalla, el dispositivo entra en modo táctil. Desde ese momento, sólo los Views con isFocusableInTouchMode() true podrán recibir el focus, como por ejemplo los widgets editores de texto. Otros Views táctiles, como los botones, no recibirán el focus cuando son tocados, lo que harán será lanzar sus listeners on-click cuando son pulsados.

En el momento en el que el usuario pulsa una tecla direccional o hace scroll con un trackball, el dispositivo abandona el modo táctil y encuentra un view que agarre el focus. En este momento el usuario puede recuperar la interacción con la interfaz de usuario sin tocar la pantalla.

El estado del modo táctil se mantiene durante todo el sistema (todas las ventanas y actividades). Para obtener el estado en curso, se puede llamar al método isInTouchMode() para ver si el dispositivo está en modo táctil o no.

Manejar el focus

El framework manejará el movimiento rutinario del focus en respuesta al input del usuario. Esto incluye cambiar el focus cuando se eliminan u ocultan los Views o cuando se encuentra disponible un nuevo View. Los View indican si pueden recibir el focus con el método `isFocusable()`. Para cambiar que el View pueda recibir o no el focus hay que llamar al método `setFocusable()`. Cuando se está en modo táctil, se puede saber si un View permite el focus mediante el método `isFocusableInTouchMode()`. Se puede cambiar esto con `setFocusableInTouchMode()`.

El movimiento del focus está basado en un algoritmo que encuentra el vecino más cercano en una dirección dada. En algunas ocasiones, el algoritmo por defecto puede que no coincida con el esperado por el desarrollador. En estas situaciones, se pueden sobrescribir determinados atributos en el archivo layout: `nextFocusDown`, `nextFocusLeft`, `nextFocusRight`, y `nextFocusUp`. Hay que añadir uno de estos atributos al View a partir del cual parte el focus y definir el valor del atributo para que sea el id del View al que hay que entregar el focus. Por ejemplo:

```
<LinearLayout
    android:orientation="vertical"
    ... >
    <Button android:id="@+id/top"
        android:nextFocusUp="@+id/bottom"
        ... />
    <Button android:id="@+id/bottom"
        android:nextFocusDown="@+id/top"
        ... />
</LinearLayout>
```

En este layout vertical, si navegamos hacia arriba a partir del primer botón no iríamos a ningún lado, tampoco si navegamos hacia abajo desde el segundo botón. Pero una vez que el botón superior ha definido el de abajo como el `nextFocusUp` (y vice versa), el focus navegacional irá desde arriba hacia abajo y desde abajo hacia arriba.

Si se quiere declarar en la IU un View como que puede recibir el focus (cuando normalmente no lo haría), añadir el atributo XML `android:focusable` al View en la declaración del layout. Setear el valor `true`. También se puede declarar un View como que puede recibir el focus si está en modo táctil con `android:focusableInTouchMode`.

Para pedir que un View en particular reciba el focus, llamar al método `requestFocus()`.

Para escuchar a los eventos del focus (ser notificado cuando un View recibe o pierde el focus) utilizar el método `onFocusChange()`, como se detalla en la sección `Listeners de Eventos`.

Notificaciones al Usuario

Pueden surgir diferentes tipo de situaciones que necesitan ser notificadas al usuario. Algunos eventos necesitan que el usuario responda y otros no. Por ejemplo:

- Cuando se termina un evento como el de guardar un archivo, debería aparecer un pequeño mensaje de confirmación informando que se ha guardado correctamente.
- Si la aplicación se está ejecutando en un segundo plano y necesita la atención del usuario, la aplicación debe crear una notificación que permite al usuario responder a su conveniencia.
- Si la aplicación está realizando algún tipo de trabajo por lo que el usuario tiene que esperar (como cargar un archivo), la aplicación debe mostrar una barra de progreso.

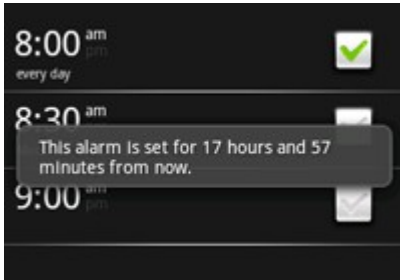
Se pueden realizar cada una de estas tareas de notificación mediante diferentes técnicas:

- Una notificación emergente "Toast", para mensajes cortos que vienen desde un segundo plano.
- Una notificación de barra de estado, para recordatorios que vienen de un segundo plano y requieren la respuesta del usuario.

- Una notificación de diálogo para notificaciones relacionadas con un Activity. Este documento resumen cada una de estas técnicas para notificar al usuario.

Notificaciones emergentes Toast

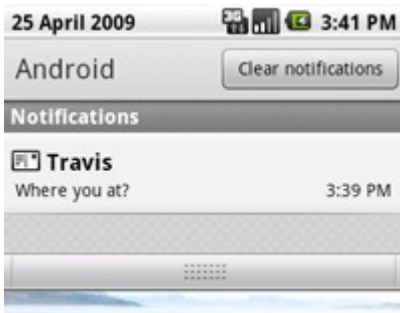
Una notificación emergente "Toast" es un mensaje que salta y rellena solamente el espacio necesario para el mensaje por lo que el activity del usuario permanece visible e interactivo. La notificación salta y se va automáticamente, sin aceptar interacción de eventos. Ya que las notificaciones emergentes pueden ser creadas de un Service en segundo plano, aparecen incluso si la aplicación no es visible.



Un toast es lo más conveniente para mensajes de texto, como "File saved," cuando se está seguro de que el usuario está atendiendo a la pantalla. Un toast no puede aceptar la interacción de eventos; si se quiere que el usuario responda y realice una acción, hay que considerar utilizar una notificación de barra de estado.

Notificación en la barra de estado

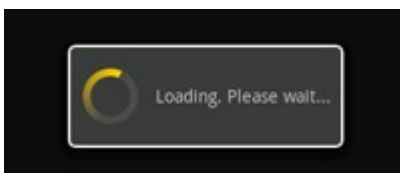
Una notificación en la barra de estado añade un icono a la barra de estado del sistema (con un mensaje opcional "ticker-text") y un mensaje expandido en la ventana "Notificaciones". Cuando el usuario selecciona el mensaje expandido, Android lanza un Intent que es definido por la notificación (normalmente para lanzar un Activity). También se puede configurar la notificación para avisar al usuario con un sonido, una vibración o luces parpadeantes en el dispositivo.



Este tipo de notificación es ideal cuando la aplicación está funcionando en un Service en un segundo plano y necesita notificar sobre el evento al usuario. Si se necesita avisar al usuario sobre un evento que ocurre mientras el Activity tiene el focus, hay que utilizar la notificación de diálogo.

Diálogos de notificación

Un diálogo es una pequeña ventana que aparece delante del Activity en curso. El activity que está detrás pierde el focus y el diálogo acepta todas las interacciones del usuario. Los diálogos se utilizan normalmente para notificaciones y actividades cortas directamente relacionadas con la aplicación que está en curso.

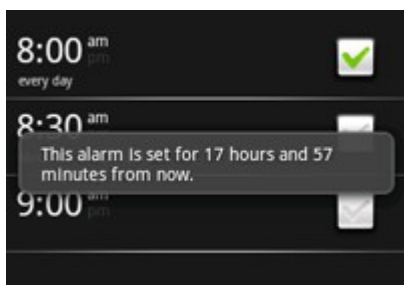


Un diálogo se debería utilizar cuando se necesita mostrar una barra de progreso o un mensaje corto que necesita confirmación del usuario (como una alerta con botones "OK" y "Cancel"). También se pueden utilizar como componentes integrales de la IU de la aplicación. Para más detalles sobre los tipos de diálogos, incluyendo sus usos para notificaciones, ver [Crear Diálogos](#).

Notificaciones Toast

Una notificación emergente (toast) es un mensaje que aparece en la ventana. Sólo ocupa el espacio necesario para el mensaje y la actividad en curso permanece visible e interactiva. La notificación aparece y desaparece automáticamente y no acepta eventos de interacción.

El pantallazo debajo muestra un ejemplo de notificación toast que aparece en la aplicación Alarma. Una vez que la alarma se enciende, se muestra un toast para indicar que la alarma está seteada.



Un toast se puede crear y mostrar a partir de un [Activity](#) o [Service](#). Si se crea una notificación toast a partir de un Service, aparecerá delante del Activity que tiene el focus.

Si se necesita que el usuario responda a la notificación, hay que utilizar una [Notificación en la barra de estado](#).

Básicos

Primero hay que instanciar un objeto [Toast](#) con uno de los métodos [makeText\(\)](#). Este método tiene tres parámetros: la aplicación [Context](#), el mensaje de texto, y la duración del toast. Devuelve un objeto Toast inicializado. Se puede mostrar la notificación toast con [show\(\)](#), tal y como se muestra en el siguiente ejemplo:

```
Context context = getApplicationContext();
CharSequence text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

Este ejemplo muestra todo lo que se necesita para la mayoría de las notificaciones toast. Normalmente no se necesita nada más. Las siguientes secciones, describen como hacer para, si además, se quiere colocar el toast de manera diferente o utilizar un layout propio en vez de un mensaje de texto simple.

Como vemos aquí, también se pueden encadenar los métodos:

```
Toast.makeText(context, text, duration).show();
```

Colocar el Toast

La notificación toast estándar aparece cerca de la parte de abajo de la pantalla, centrada horizontalmente. Se puede cambiar la posición con el método [setGravity\(int, int, int\)](#).

Acepta tres parámetros: una constante [Gravity](#), constant, una posición-x offset, y una posición-y offset.

Ejemplo; si se decide que el toast aparezca en la esquina superior izquierda, se puede setear el

gravity de la siguiente manera:

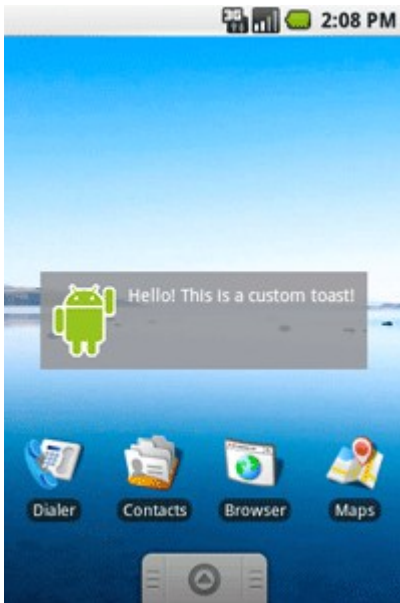
```
toast.setGravity(Gravity.TOP|Gravity.LEFT, 0, 0);
```

Si se quiere mover la posición a la derecha, hay que incrementar el valor del segundo parámetro. Para moverlo hacia abajo, hay que incrementar el valor del último parámetro.

Crear un Toast View personalizado

Si un texto simple no es suficiente, se puede crear un layout personalizado para la notificación toast. Para crear un layout personalizado, hay que definir un layout de un View, bien en un XML o en el código de la aplicación, y pasarle el objeto raíz View al método setView(View).

Como ejemplo tenemos a continuación como se puede crear un layout para el toast visible en el pantallazo abajo con el siguiente XML (guardado como toast_layout.xml):



```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toast_layout_root"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:background="#DAAA"
    >
    <ImageView android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_marginRight="10dp"
        />
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:textColor="#FFF"
        />
</LinearLayout>
```

Obsérvese que el ID del elemento LinearLayout es "toast_layout". Se debe utilizar este ID para inflar el layout a partir del XML, como se muestra a continuación:

```

LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.toast_layout,
                             (ViewGroup) findViewById(R.id.toast_layout_root));

ImageView image = (ImageView) layout.findViewById(R.id.image);
image.setImageResource(R.drawable.android);
TextView text = (TextView) layout.findViewById(R.id.text);
text.setText("Hello! This is a custom toast!");

Toast toast = new Toast(getApplicationContext());
toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();

```

Primero, recuperar el `LayoutInflater` con `getLayoutInflater()` (o `getSystemService()`), y después inflar el layout a partir del XML utilizando `inflate(int, ViewGroup)`. El primer parámetro es el ID del layout y el segundo es el View raíz. Se puede utilizar este layout inflado para encontrar más objetos View en el layout, para después capturarlos y definir el contenido para los elementos `ImageView` y `TextView`. Finalmente, crear un nuevo `Toast` con `Toast(Context)` y setearle algunas de las propiedades del toast, como la posición y la duración. Después se llama al método `setView(View)` y se le pasa el layout inflado. Ahora ya se puede mostrar el toast con el layout personalizado, llamando al método `show()`.

Nota: No utilizar el constructor público para un `Toast` a no ser que se vaya a definir el layout con un `setView(View)`. Para crear un `Toast`, si no se tiene un layout personalizado que se pueda utilizar, se debe usar `makeText(Context, int, int)`.

Notificaciones en Barra de Estado

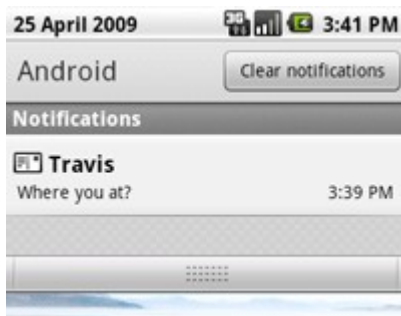
Una notificación en la barra de estado añade un icono a la barra de estado del sistema (con un mensaje de texto ticker opcional) y un mensaje extendido en la ventana "Notificaciones". Cuando el usuario selecciona el mensaje extendido, Android lanza un `Intent` que es definido por la notificación (para lanzar un `Activity`). También se puede configurar la notificación para alertar al usuario con un sonido, una vibración, o con luces.

Una notificación en la barra de estado puede ser utilizado en todos los casos en los que un `Service` en segundo plano alerte sobre un evento que requiere una respuesta. Este tipo de `Service` no debería nunca lanzar un `Activity` por si mismo para recibir una interacción del usuario. El `Service` debería crear una notificación en la barra de estado que lance el `Activity` cuando sea seleccionado por el usuario.

El pantallazo a continuación muestra la barra de estado con un icono de una notificación en la parte de la izquierda.



El siguiente pantallazo muestra el mensaje extendido de la notificación en la ventana de "Notificaciones". El usuario puede ver la ventana de notificaciones (o seleccionando del menú de opciones de inicio Notificaciones).



Básicos

Un Activity o un Service puede iniciar una notificación en la barra de estado. Dado que un Activity puede realizar acciones sólo cuando está activo y con el focus, se debe crear las notificaciones en la barra de estado a partir de un Service. De esta manera, la notificación puede ser creada desde un segundo plano, mientras que el usuario está utilizando otra aplicación o mientras que el dispositivo está en pausa. Para crear una notificación, se deben utilizar dos clases: Notification y NotificationManager.

Usar una instancia de la clase Notification para definir las propiedades de la notificación en la barra de estado, como el icono en la barra de estado, el mensaje extendido y las configuraciones extras como por ejemplo el sonido. El NotificationManager es un servicio del sistema Android que ejecuta y gestiona todas las notificaciones. El NotificationManager no se instancia. Para darle la notificación, se debe recuperar una referencia al NotificationManager mediante el método getSystemService() y después, cuando se quiere notificar al usuario, se le pasa el objeto Notification con el método notify().

Para crear una notificación en la barra de estado:

1. Obtener una referencia al NotificationManager:

```
String ns = Context.NOTIFICATION_SERVICE;  
NotificationManager mNotificationManager = (NotificationManager) getSystemService(ns);
```

2. Instanciar la notificación:

```
int icon = R.drawable.notification_icon;  
CharSequence tickerText = "Hello";  
long when = System.currentTimeMillis();  
  
Notification notification = new Notification(icon, tickerText, when);
```

3. Definir el mensaje extendido de la notificación y el Intent:

```
Context context = getApplicationContext();  
CharSequence contentTitle = "My notification";  
CharSequence contentText = "Hello World!";  
Intent notificationIntent = new Intent(this, MyClass.class);  
PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);  
  
notification.setLatestEventInfo(context, contentTitle, contentText, contentIntent);
```

4. Pasar la notificación al NotificationManager:

```
private static final int HELLO_ID = 1;  
  
mNotificationManager.notify(HELLO_ID, notification);
```

Ya está. El usuario ya ha sido notificado.

Gestionar las notificaciones

El NotificationManager es un servicio del sistema que gestiona todas las notificaciones. Se debe recuperar una referencia a él mediante el método getSystemService() Por ejemplo:

```
String ns = Context.NOTIFICATION_SERVICE;
NotificationManager mNotificationManager = (NotificationManager) getSystemService(ns);
```

Cuando se quiere mandar la notificación en la barra de estado, hay que pasar el objeto Notification al NotificationManager con el método notify(int, Notification). El primer parámetro es el ID unívoco para la notificación y el segundo es el objeto Notification. El ID identifica unívocamente la notificación en la aplicación. Esto es necesario si se necesita actualizar la notificación o (si la aplicación gestiona diferentes tipos de notificaciones) para seleccionar la acción apropiada cuando el usuario vuelve a la aplicación a través del Intent definido en la notificación.

Para limpiar la notificación en la barra de estado cuando el usuario la selecciona de la ventana de notificaciones, hay que añadir el flag "FLAG_AUTO_CANCEL" al objeto Notification. También se puede limpiar de manera manual con el método cancel(int), pasándole el ID de la notificación, o bien limpiando todas las notificaciones con el método cancelAll().

Crear una notificación

Un objeto Notification define los detalles del mensaje de notificación que se muestra en la barra de estado y en la ventana de "Notificaciones", y cualquier otro tipo de alertas, como de sonido o de luces.

Una notificación en la barra de estado requiere lo siguiente:

- Un icono para la barra de estado
- Un título y un mensaje extendido para la vista extendida (a no ser que se defina una vista extendida personalizada)
- Un PendingIntent, que va a ser lanzado cuando se selecciona la notificación

Las configuraciones opcionales para la notificación en la barra de estado incluyen:

- Un mensaje "ticker-text" para la barra de estado
- Sonido como alerta
- Vibración
- Configuración de luces LED

El kit de inicio para una nueva notificación incluye el constructor Notification(int, CharSequence, long) y el método setLatestEventInfo(Context, CharSequence, CharSequence, PendingIntent). Estos definen todas las configuraciones necesarias para una notificación. El siguiente código muestra una configuración básica de una notificación:

```
int icon = R.drawable.notification_icon;    // icon from resources
CharSequence tickerText = "Hello";         // ticker-text
long when = System.currentTimeMillis();    // notification time
Context context = getApplicationContext(); // application Context
CharSequence contentTitle = "My notification"; // expanded message title
CharSequence contentText = "Hello World!";  // expanded message text

Intent notificationIntent = new Intent(this, MyClass.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);

// las siguientes dos líneas inicializan la notificación, utilizando las configuraciones antes vistas.
Notification notification = new Notification(icon, tickerText, when);
notification.setLatestEventInfo(context, contentTitle, contentText, contentIntent);
```

Actualizar la notificación

Se puede actualizar la información en la notificación en la barra de estado según vayan ocurriendo diferentes eventos en la aplicación. Cuando por ejemplo, llega un nuevo SMS antes de que se hayan leído mensajes que llegaron antes, la aplicación de mensajes actualiza la notificación ya existente para mostrar el número total de mensajes recibidos. Esta práctica de actualizar una notificación ya

existente es mucho mejor que añadir una nueva notificación al `NotificationManager` ya que evita abarrotar la ventana de notificaciones.

Ya que cada notificación se identificada unívocamente mediante su ID por el `NotificationManager`, se puede modificar la notificación con nuevos valores llamando al método `setLatestEventInfo()`, cambiar algunos de los valores en los campos de la notificación, y después llamar de nuevo al método `notify()`.

Se puede modificar cada uno de las propiedades con los atributos del objeto (exceptuando el Context y el título y texto del mensaje extendido). El mensaje de texto se modifica llamando a `setLatestEventInfo()` con nuevos valores para `contentTitle` y `contentText`. Después se llama a `notify()` para actualizar la notificación. (La modificación del título y el texto no tendrá efecto si se ha creado una vista extendida personalizada.)

Añadir un sonido

Se puede alertar al usuario con el sonido por defecto de la notificación (definido por el usuario) o bien con un sonido especificado por la aplicación.

Para usar el sonido por defecto, hay que añadir "DEFAULT_SOUND" al campo `defaults`:

```
notification.defaults |= Notification.DEFAULT_SOUND;
```

Para utilizar un sonido diferente con las notificaciones, hay que pasar un Uri de referencia al campo `sound`. El siguiente ejemplo utiliza un archivo de audio conocido guardado en la tarjeta SD del dispositivo:

```
notification.sound = Uri.parse("file:///sdcard/notification/ringer.mp3");
```

En el siguiente ejemplo, el archivo de audio es escogido a partir del `MediaStore's ContentProvider` interno:

```
notification.sound = Uri.withAppendedPath(Audio.Media.INTERNAL_CONTENT_URI, "6");
```

En este caso, se conoce el ID del archivo media ("6") y se adjunta al `Uri` del content. Si no se conoce el ID exacto, se puede consultar todos los media disponibles en el `MediaStore` con un `ContentResolver`. Para más información sobre como utilizar un `ContentResolver` ver la documentación sobre `Content Providers`.

Si se quiere que el sonido se repita hasta que el usuario responda a la notificación o hasta que sea cancelada, hay que añadir "FLAG_INSISTENT" al campo `flags`.

Nota: Si el campo `defaults` incluye "DEFAULT_SOUND", el sonido por defecto sobrescribe cualquier sonido definido por el campo `sound`.

Añadir vibración

Se puede alertar al usuario con el diseño de la vibración por defecto o con un diseño definido por la aplicación.

Para utilizar el diseño por defecto, hay que añadir "DEFAULT_VIBRATE" al campo `defaults`:

```
notification.defaults |= Notification.DEFAULT_VIBRATE;
```

Para definir un diseño de vibración propio, hay que pasar un array de valores `long` al campo `vibrate field`:

```
long[] vibrate = {0,100,200,300}; notification.vibrate = vibrate;
```

El array de valores `long` define los tiempos de vibración del diseño alternativo (en milisegundos). El

primer valor es cuanto tiempo tiene que pasar (apagado) antes de empezar, el segundo valor es la longitud de la primera vibración, el tercero es la siguiente longitud de apagado, etc. El diseño puede ser tan largo como se quiere pero no se puede setear para que se repita.

Nota: Si el campo defaults incluye "DEFAULT_VIBRATE", la vibración por defecto sobrescribe a cualquier vibración definida en el campo vibrate.

Añadir luces parpadeantes

Para alertar al usuario con luces LED, se puede implementar el diseño de luces por defecto, o definir un diseño y color propios.

Para utilizar las configuraciones por defecto, hay que añadir "DEFAULT_LIGHTS" al campo defaults:

```
notification.defaults |= Notification.DEFAULT_LIGHTS;
```

Para definir el color y el diseño, hay que definir el valor para el campo ledARGB (para el color), el campo ledOffMS(longitud de tiempo con las luces apagadas, en milisegundos), el ledOnMS (tiempo con las luces encendidas, en milisegundos), y añadir "FLAG_SHOW_LIGHTS" al campo flags:

```
notification.ledARGB = 0xff00ff00;
notification.ledOnMS = 300;
notification.ledOffMS = 1000;
notification.flags |= Notification.FLAG_SHOW_LIGHTS;
```

En este ejemplo, la luz verde parpadea repetidamente durante 300 milisegundos y se apaga durante un segundo. Los LEDs del dispositivo no soportan todos los colores del espectro y no todos los dispositivos soportan los mismos colores, por lo que el hardware pondrá lo que más se parezca a lo configurado. El verde es el color más común para las notificaciones.

Más propiedades

Se pueden añadir más propiedades a las notificaciones utilizando campos y flags de Notification. A continuación se incluyen algunas propiedades útiles:

"FLAG_AUTO_CANCEL" flag

Añadir esto al campo flags para cancelar automáticamente la notificación después de que haya sido seleccionado desde la ventana de notificaciones.

"FLAG_INSISTENT" flag

Añadir esto al campo flags para repetir el audio hasta que el usuario responda.

"FLAG_ONGOING_EVENT" flag

Añadir esto al campo flags para agrupar la notificación debajo del título "actual" en la ventana de notificaciones. Esto indica que la aplicación está en curso— su proceso está todavía en ejecución en un segundo plano, aún cuando la aplicación no es visible (como con música o una llamada).

"FLAG_NO_CLEAR" flag

Añadir esto al campo flags para indicar que la notificación no debería ser eliminada por el botón "Limpiar notificaciones". Esto es particularmente útil si la notificación está en curso.

campo number

Este valor indica el número de eventos en curso representados por la notificación. El número apropiado se escribe encima del icono de la barra de estado. Si se quiere utilizar este campo, se debe empezar con "1" cuando se crea el Notification por primera vez. (Si se cambia el valor de cero a cualquier número superior durante una actualización, el número no se mostrará.)

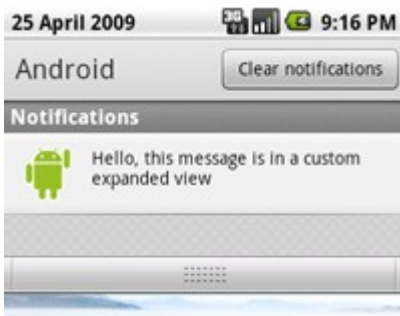
campo iconLevel

Este valor indica el nivel en curso de [LevelListDrawable](#) utilizado por el icono de la notificación. Se puede animar el icono en la barra de estado cambiando este valor para que corresponda con el gráfico definido en [LevelListDrawable](#). Para más información ver el documento sobre [LevelListDrawable](#).

Para más información sobre propiedades adicionales para personalizar la aplicación ver la clase [Notification](#).

Crear un View extendido personalizado

Por defecto, el view extendido utilizado en la ventana "Notificaciones" incluye un título y un mensaje de texto. Estos son definidos por los parámetros `contentTitle` y `contentText` del método `setLatestEventInfo()`. Sin embargo, también se puede definir un layout personalizado para el view extendido utilizando [RemoteViews](#). El pantallazo mostrado a continuación muestra un ejemplo de un view extendido personalizado que utiliza un `ImageView` y un `TextView` en un `LinearLayout`.



Para definir un layout propio para el mensaje extendido, hay que instanciar un objeto [RemoteViews](#) y pasarlo al campo `contentView` de la notificación. Hay que pasar el [PendingIntent](#) al campo `contentIntent`.

A continuación tenemos un ejemplo de como crear un view extendido personalizado:

- Crear el XML del layout para el view extendido. Crear por ejemplo, un archivo layout llamado `custom_notification_layout.xml` y desarrollarlo de la siguiente manera:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="3dp"
    >
    <ImageView android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_marginRight="10dp"
        />
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:textColor="#000"
        />
</LinearLayout>
```

Este layout es utilizado para el view extendido, pero el contenido del `ImageView` y del `TextView` todavía hay que definirlo en la aplicación. [RemoteViews](#) ofrece métodos útiles que permiten definir este contenido.

- Hay que utilizar, en el código de la aplicación, los métodos de [RemoteViews](#) para definir la

imagen y el texto. Después hay que pasar el objeto RemoteViews al campo contentView de la notificación, como se muestra en este ejemplo:

```
RemoteViews contentView = new RemoteViews(getPackageName(),
R.layout.custom_notification_layout);
contentView.setImageDrawableResource(R.drawable.notification_image);
contentView.setTextViewText(R.id.text, "Hello, this message is in a custom expanded view");
notification.contentView = contentView;
```

Como se muestra aquí, hay que pasar el nombre del paquete de la aplicación y el ID del layout al constructor del RemoteViews. Después, hay que definir el contenido para el ImageView y el TextView, utilizando los métodos `setImageDrawableResource()` y `setTextViewText()`. En cada caso, hay que pasar el ID del objeto View que se quiere setear, junto con el valor para ese View. Por último, se pasa el objeto RemoteViews al Notification a través del campo contentView.

- Como no se necesita el método `setLatestEventInfo()` cuando se utiliza un view personalizado, se debe definir el Intent para la notificación con el campo `contentIntent` tal y como se muestra en este ejemplo:

```
Intent notificationIntent = new Intent(this, MyClass.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);
notification.contentIntent = contentIntent;
```

- Ahora se puede mandar la notificación de manera común:

```
mNotificationManager.notify(CUSTOM_VIEW_ID, notification);
```

La clase RemoteViews también incluye métodos que se pueden utilizar para añadir fácilmente al view extendido de la notificación un [Chronometer](#) o un [ProgressBar](#). Para más información sobre como crear layouts personalizados con RemoteViews, ver la clase [RemoteViews](#).

Nota: Cuando se crea un view extendido personalizado, se debe prestar especial cuidado para que el layout personalizado funcione correctamente en las diferentes orientaciones y resoluciones del dispositivo. Este consejo se debe aplicar a todos los layouts de los Views creados en Android, pero especialmente en este caso ya que el estado real del layout está muy limitado. No se debe hacer el layout personalizado muy complejo y hay que asegurarse de testearlo en varias configuraciones diferentes.