Desarrollo de apps con Android

11.Transferencia eficiente de datos y JobScheduler

Transferir datos de manera eficiente

Consumo de recursos en transferencias de datos

- La radio inalámbrica consume batería.
  - El dispositivo se puede quedar sin batería.
  - Necesidad de permitir la carga del dispositivo.
- La transferencia de datos hace uso de los planes de datos del usuario.
  - Tiene un coste real para los usuarios.

Precarga de datos

- Descargar todos los datos que se supone que se van a necesitar en un periodo de tiempo determinado de una sola vez, a través de un única conexión y con toda la capacidad posible.
- Si se hace bien, se reducirá el coste de batería y la latencia.
- Si se calcula mal, puede llegar a consumir demasiada batería y ancho de banda de datos.

Monitorizar el estado de conectividad

- La radio Wi-Fi (alta frecuencia) utiliza menos batería y tiene más ancho de banda que la radio inalámbrica (baja frecuencia).
- Utilizar `ConnectivityManager` para determinar qué radio está activa y así seguir una estrategia u otra.

Monitorizar el estado de la batería

- Esperar a que se den ciertas condiciones para realizar operaciones intensivas sobre la batería.
- `BatteryManager` hace una transmisión (broadcast) de todos los detalles de carga y de la batería en un `Intent` de broadcast.
- Utilizar un `BroadcastReceiver` registrado para las acciones de estado de la batería.
- `Descripción general Broadcast`

JobScheduler

Introducción

El usuario puede configurar el uso que una app puede hacer de la batería de su dispositivo:

- **No restringido:** La app puede usar la batería en segundo plano sin restricciones (puede penalizar bastante la duración de la batería).
- **Optimizado:** Se optimiza el uso de la batería en función del uso que el usuario haga de la app. Opción recomendada para la mayoría de las apps.
- **Restringida:** Restringe el uso de la batería cuando la app se ejecuta en segundo plano. En este caso, es posible que la app no funcione según lo previsto y que las notificaciones se puedan recibir con retraso.

¿Qué es JobScheduler?

- Se utiliza para la programación eficiente de tareas en segundo plano *(background)*.
- Se basa en condiciones, no en una planificación temporal (ej., estado de la batería y conectividad).
- Agrupa las tareas de cara a minimizar el consumo de batería.
- Es mucho más eficiente que `AlarmManager`.
- Otra alternativa a `JobScheduler` es la clase `WorkManager` (más detalle).

Componentes de un JobScheduler

- `JobService`: Clase del servicio donde se inicia la tarea.
- `JobInfo`: Clase que sigue el patrón Builder para establecer las condiciones que se han de cumplir para realizar la tarea.
- `JobScheduler`: Clase abstracta que permite programar y cancelar tareas, lanzar un servicio.

JobService

JobService

- En la clase `JobService` es donde se implementa la tarea a realizar.
- Sobrescribir los siguientés métodos:
  - `onStartJob(JobParameters params)`
  - `onStopJob(JobParameters params)`
- **Se ejecuta en el hilo principal.**

onStartJob (JobParameters params) (1/2)

- Implementar en este método el trabajo a realizar.
- Llamado por el sistema Android cuando se cumplen las condiciones establecidas.
- Se ejecuta en el hilo principal.
- **Ejecutar en otro hilo todo el trabajo más pesado.**

onStartJob (JobParameters params) (2/2)

`Devuelve un valor booleano.`

`FALSE`: Trabajo terminado.

`TRUE`:

- El trabajo se ha llevado a otro hilo *(worker thread)*.
- Debe invocar al método `jobFinished()` desde cualquier otro hilo.
  - Pasar el objeto `JobParameters` desde `onStartJob()`.
  - Pasar el booleano `wantsReschedule`: **true** si se desea reprogramar el trabajo posteriormente.

onStopJob(JobParameters params)

- Invocado por el sistema Android si considera que debe terminar la ejecución de un determinado trabajo.
- Puede darse el caso si los requisitos especificados no se pueden volver a cumplir.
- Se ejecuta antes que `jobFinished(JobParameters, boolean)`.
- Devolver `TRUE` para reprogramar, FALSE para abandonar la tarea.

Ejemplo de código de JobService

```
public class MyJobService extends JobService {
  private UpdateAppsAsyncTask updateTask =
                          new UpdateAppsAsyncTask();
  @Override
  public boolean onStartJob(JobParameters params) {
    updateTask.execute(params);
    return true;
  }

  @Override

  public boolean onStopJob(JobParameters jobParameters) {
     return true;
  }
}
```

Registro de JobService

```
<service
    android:name=".NotificationJobService"
    android:permission=
        "android.permission.BIND_JOB_SERVICE"/>
```

JobInfo

JobInfo

- Establece las condiciones de la ejecución.
- Objeto `JobInfo.Builder`.

Objeto JobInfo.Builder

- **Parámetro 1:** ID de la tarea.
- **Parámetro 2:** Componente de Servicio.
- **Parámetro 3:** `JobService` que hay que lanzar.

```
JobInfo.Builder builder = new JobInfo.Builder(
        JOB_ID,
        new ComponentName(getPackageName(),
        NotificationJobService.class.getName()));
```

Establecer condiciones

setRequiredNetworkType(int networkType)

setBackoffCriteria(long initialBackoffMillis, int backoffPolicy)

setMinimumLatency(long minLatencyMillis)

setOverrideDeadline(long maxExecutionDelayMillis)

setPeriodic(long intervalMillis)

setPersisted(boolean isPersisted)

setRequiresCharging(boolean requiresCharging)

setRequiresDeviceIdle(boolean requiresDeviceIdle)

setRequiredNetworkType()

`setRequiredNetworkType(int networkType)`

- `NETWORK_TYPE_NONE`: Por defecto, no se requiere conectividad.
- `NETWORK_TYPE_ANY`: Requiere que haya conectividad.
- `NETWORK_TYPE_NOT_ROAMING`: Requiere conectividad que no sea roaming.
- `NETWORK_TYPE_UNMETERED`: Requiere conectividad Wi-Fi o con datos ilimitados.

setMinimumLatency()

`setMinimumLatency(long minLatencyMillis)`

- El número mínimo de milisegundos que hay que esperar antes de terminar la tarea.

setOverrideDeadline()

`setOverrideDeadline(long maxExecutionDelayMillis)`

- El número máximo de milisegundos que hay que esperar antes de ejecutar la tarea, incluso cuando no se dan otras condiciones.

setPeriodic()

`setPeriodic(long intervalMillis)`

- Repite la tarea tras transcurrir un determinado periodo de tiempo.
- Pasar el intervalo de repetición.
- Es mutuamente exclusivo con las condiciones `setMinimumLatency()` y `setOverrideDeadline()`.
- No se garantiza que se ejecute la tarea en el periodo indicado.

setPersisted()

`setPersisted(boolean isPersisted)`

- Establece si el trabajo persiste aunque se reinicie el sistema Android.
- Pasar `True` o `False`.
- Requiere el permiso `RECEIVE_BOOT_COMPLETED`.

setRequiresCharging()

`setRequiresCharging(boolean requiresCharging)`

- Establece si el dispositivo debe estar enchufado o no.
- Pasar `True` o `False`.
- Por defecto a `False`.

setRequiresDeviceIdle()

`setRequiresDeviceIdle(boolean requiresDeviceIdle)`

- Establece si el dispositivo está en “modo desocupado”.
- La definición de “modo desocupado” en Android es un tanto ambigua, pero se entiende que es cuando el dispositivo no se está utilizando, y no lo ha estado durante un tiempo.
- Utilizarlo para tareas que requieren recursos muy pesados.
- Pasar `True` o `False`.
- Por defecto a `False`.

Ejemplo de código JobInfo

```
JobInfo.Builder builder = new JobInfo.Builder(
        JOB_ID, new ComponentName(getPackageName(),
        NotificationJobService.class.getName()))

    .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED)
        .setRequiresDeviceIdle(true)
        .setRequiresCharging(true);
JobInfo myJobInfo = builder.build();
```

JobScheduler

Programando la tarea

1. Obtener un objeto `JobScheduler` del sistema.
2. Invocar al método `schedule()` de `JobScheduler`, con el objeto `JobInfo`.

```
mScheduler =
    (JobScheduler)getSystemService(JOB_SCHEDULER_SERVICE);
mScheduler.schedule(myJobInfo);
```