Programación multimedia y dispositivos móviles

8 Almacenamiento de información

IES Nervión
Miguel A. Casado Alías

Introducción

- En Android las aplicaciones pueden guardar información de varias maneras:
  - Pares de datos “Clave-Valor” (SharedPreferences)
  - Ficheros
  - Bases de datos SQLite
- En función del tipo y complejidad de la información a guardar se elegirá la forma de almacenamiento

PARTE 1:

SharedPreferences

¿Qué es “SharedPreferences”?

- Un objeto SharedPreferences apunta a un fichero que contiene pares **clave-valor** y proporciona métodos para leerlos y escribirlos
- No confundir con **Preferences**, que se usa para construir una interfaz de usuario que gestione las preferencias / ajustes de nuestra aplicación

Creación de ficheros SharedPreferences

- Si necesitamos varios ficheros para nuestra actividad e identificar cada uno con un nombre, usaremos el método **getSharedPreferences:**

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

- Si sólo necesitamos un fichero, usaremos **getPreferences:**

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

- Con MODE_PRIVATE indicamos que el fichero sólo es accesible por nuestra aplicación. MODE_WORLD_READABLE y MODE_WORLD_WRITEABLE dan acceso en lectura y escritura respectivamente a todo el mundo.
- Es mejor asegurarse que el nombre del fichero será único. Por ejemplo: com.example.myapp.PREFERENCE_FILE_KEY

Creación de ficheros SharedPreferences

- Si necesitamos varios ficheros para nuestra actividad e identificar cada uno con un nombre, usaremos el método **getSharedPreferences:**

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

- Si sólo necesitamos un fichero, usaremos **getPreferences:**

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

- Con MODE_PRIVATE indicamos que el fichero sólo es accesible por nuestra aplicación. MODE_WORLD_READABLE y MODE_WORLD_WRITEABLE dan acceso en lectura y escritura respectivamente a todo el mundo.
- Es mejor asegurarse que el nombre del fichero será único. Por ejemplo: com.example.myapp.PREFERENCE_FILE_KEY

Escritura en SharedPreferences

- Debemos crear un Editor, para ello llamamos al método **edit()** de nuestro SharedPreferences.
- Añadiremos pares clave-valor, llamando a métodos del tipo putInt , putString... del editor
- Por último haremos una llamada a commit sobre el editor
- Ejemplo:

```
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score),
              newHighScore);
editor.commit();
```

Lectura de SharedPreferences

- Usaremos métodos del tipo getInt , getString...
- Podemos proporcionar un valor por defecto para el caso de que no esté presente la clave solicitada
- Ejemplo:

```
long default =
getResources().getInteger(R.string.saved_high_score_default));
long highScore =
sharedPref.getInt(getString(R.string.saved_high_score),
default);
```

PARTE 2:

Ficheros

java.io.File

- Android usa un sistema de ficheros similar a los sistemas de ficheros de otras plataformas
- Para trabajar con el sistema de ficheros de Android usaremos la API java.io.File
- Se usarán ficheros para guardar e intercambiar información a través de la red, o para trabajar con recursos como imágenes, audio y video.

Dos zonas de almacenamiento

- Los sistemas Android siempre disponen de dos zonas de almacenamiento: almacenamiento interno y almacenamiento externo
- El almacenamiento interno es un almacenamiento no volátil, inherente al propio dispositivo
- El almacenamiento externo se empezó a llamar así porque hacía referencia al almacenamiento en un medio extraíble (generalmente tarjeta SD).
- Algunos dispositivos dividen el almacenamiento permanente en dos particiones (almacenamiento interno y almacenamiento externo). Trabajaremos igual tanto si el almacenamiento externo es extraíble como si no lo es

Interno vs Externo: Comparativa

- Interno:
  - Siempre disponible
  - Los ficheros guardados aquí sólo son accesibles por tu aplicación
  - Al desinstalar la aplicación, se eliminan todos sus ficheros almacenados aquí
- Externo:
  - No siempre disponible (el usuario puede montar la tarjeta SD como almacenamiento USB para otro dispositivo...)
  - Todos pueden leer
  - Cuando el usuario desinstala, los ficheros almacenados aquí sólo se eliminan si están guardados en el directorio proporcionado por **getExternalFilesDir()**

Interno vs Externo: Conclusiones

- El almacenamiento interno es la mejor opción cuando no queremos que otros usuarios o aplicaciones puedan acceder a los datos de nuestra aplicación
- Usaremos el almacenamiento externo para guardar datos que queremos compartir con otras aplicaciones o que queremos que permanezcan aunque la aplicación se desinstale (por ej, un archivo de música editado por nuestra aplicación...)
- Por defecto las aplicaciones se instalan en el almacenamiento interno, pero este comportamiento se puede modificar con la propiedad **android:installLocation** en nuestro AndroidManifest.xml. Más información aquí

Pedir permiso para almacenar externamente

- WRITE_EXTERNAL_STORAGE es el permiso que hay que solicitar en el AndroidManifest para poder escribir en la memoria externa
- En un futuro muy cercano, será necesario pedir el permiso READ_EXTERNAL_STORAGE para leer de la memoria externa (aunque si se pide el permiso de escritura, implícitamente se tendrá permiso de lectura también)
- VER EJEMPLOS
- No hace falta ningún permiso para leer o escribir en la memoria interna (en el directorio correspondiente a la aplicación, claro)

Pedir permiso para almacenar externamente

- WRITE_EXTERNAL_STORAGE es el permiso que hay que solicitar en el AndroidManifest para poder escribir en la memoria externa
- En un futuro muy cercano, será necesario pedir el permiso READ_EXTERNAL_STORAGE para leer de la memoria externa (aunque si se pide el permiso de escritura, implícitamente se tendrá permiso de lectura también)
- VER EJEMPLOS
- No hace falta ningún permiso para leer o escribir en la memoria interna (en el directorio correspondiente a la aplicación, claro)

Guardar en memoria interna

- **getFilesDir()** nos devuelve el directorio interno disponible para nuestra aplicación. Así pues podemos hacer:
  - File fich = new File(context.getFilesDir(), nombre_fichero);
- A partir de ahí abriríamos un flujo de salida o de entrada según nos conveniese (FileOutputStream, FileInputStream)
- También podríamos haber utilizado **openFileOutput** para obtener un FileOutputStream que escribe en un fichero del directorio interno de nuestra aplicación
- Si queremos usar ficheros temporales, nos podemos valer de **getCacheDir()** para obtener nuestro directorio interno temporal y de **createTempFile** para crear el fichero
- VER EJEMPLOS

Guardar en memoria externa

- Como la memoria externa puede no estar disponible, siempre hay que comprabar su disponibilidad antes de hacer uso de ella. Si **getExternalStorageState()** nos devuelve MEDIA_MOUNTED la podemos usar en escritura y lectura, si nos devuelve MEDIA_MOUNTED_READ_ONLY sólo podemos leer.
- **getExternalStoragePublicDirectory** nos devuelve el directorio en el que podemos guardar datos que permanecerán aunque la aplicación se desinstale. Estos datos serán accesibles para otras aplicaciones (públicos)
- Al método anterior le pasamos una constante (p.ej: DIRECTORY_MUSIC) para categorizar los datos a guardar
- **getExternalFilesDir** nos devuelve un directorio externo en el que podemos almacenar datos “privados”, que se borrarán al desinstalar la aplicación. VER EJEMPLOS

¿Cuánto espacio hay libre?

- File dispone de los métodos **getFreeSpace()** y **getTotalSpace()** que nos indican respectivamente el espacio libre y el espacio total
- Sin embargo, no tenemos garantizado poder escribir tantos bytes como nos devuelva getFreeSpace()
- No estamos obligados a comprobar el espacio libre antes de escribir, podemos capturar una IOException si no podemos escribir por falta de espacio (a veces no sabemos el tamaño de los datos a escribir...)

Borrar ficheros

- Podemos borrar los ficheros que ya no necesitemos con el método **delete()** de File
- En el caso de tratarse de ficheros almacenados internamente, también podemos borrarlos llamando a: myContext.deleteFile(nombre_fichero)

PARTE 3:

Bases de Datos

SQLite

- Android incorpora SQLite en su runtime
- Cualquier aplicación Android puede crear y usar BDs SQLite
- SQLite consume poca memoria y su velocidad es aceptable
- Se desvia poco del standard SQL-92
- La principal diferencia con otros SGBD está en la gestión de los tipos. Aunque al crear las tablas se indica el tipo de dato para cada columna, posteriormente el sistema deja introducir cualquier tipo de dato en cualquier columna (p.ej: enteros en cadenas). Esto se denomina *manifest typing*

El contrato

- Una **clase contrato** es un contenedor para constantes que definen nombres de URIs, tablas y columnas.
- Permite usar las mismas constantes en toda la aplicación. Si el nombre de una columna cambia, no tendremos que cambiar nada en el código, solo en la clase contrato.
- Se suele crear una clase interna por cada tabla, y dentro de ella se enumeran sus columnas
- Al implementar **BaseColumns**, nuestras clases heredarán un atributo llamado _ID (clave primaria) cuyo valor será "_id". **Es necesario que nuestras tablas tengan una columna "_id"** si queremos usarlas con CursorAdaptor (y derivados)
- Ver Ejemplo 8.1

SQLiteOpenHelper

- Para crear una BD usaremos una subclase de SQLiteOpenHelper. Principales métodos:
  - Un constructor, que llamará al constructor de la clase padre. Éste recibirá el contexto, el nombre de la BD, un CursorFactory (null normalmente), y un entero que representa la versión de nuestra BD
  - onCreate, al que se le pasa una SQLiteDatabase que rellenaremos con nuestras tablas y datos iniciales
  - onUpgrade, que recibe un objeto SQLiteDatabase, así como los números de la antigua versión y la nueva. Normalmente, aunque es hacer “trampas”, nuestro proceso de actualización consistirá en eliminar las tablas y crearlas de nuevo.
  - onDowngrade, similar al anterior pero para “desactualizar”.
- Ver Ejemplo 8.2

Acceder a la BD

- Debemos instanciar nuestra subclase de SQLiteOpenHelper
- Usaremos getReadableDatabase o getWritableDatabase
- Esos métodos aplicados sobre un objeto SQLIteOpenHelper nos devolverán una SQLiteDatabase en modo lectura o escritura respectivamente
- Para cerrar la conexión con la BD, llamaremos al método close sobre nuestro objeto SQLIteOpenHelper
- Ver Ejemplo 8.3

execSQL

- Es un método de SQLIteOpenHelper
- Sirve para ejecutar **sentencias SQL que no devuelvan ningún resultado**, por ejemplo: INSERT, UPDATE, DROP
- Sin embargo, hay otros métodos alternativos para llevar a cabo tareas de inserción, actualización y borrado de filas de una BD:
  - insert
  - update
  - delete

Consultas

- Para consultar una BD tenemos dos opciones:
  - rawQuery : lanza sentencia SELECT que le digamos
  - query : recibe una serie de parámetros que indican la tabla a consultar, las columnas resultado, las columnas de la clausula WHERE, los valores para las columnas de la clausula WHERE, la clausula GROUP BY, la clausula HAVING, la clausula ORDER
- El resultado de la consulta lo podremos recoger en un cursor. Ver ejemplo 8.4
- Podemos usar un SimpleCursorAdapter para construir una lista (o un Spinner, o un GridView...) con los datos de una consulta. Ver ejemplo 8.5 . **NO OLVIDAR "_ID" EN EL CURSOR, DE LO CONTRARIO FALLARÁ**