Android Essentials

Listas avanzadas

Listas con layouts personalizados

FOLDER IT

Si queremos mostrar algo más complejo en un listview necesitamos otros elementos

En este caso todo el trabajo deberá ser desarrollado por clases propias, extendiendo las que nos brinda android.

El primer paso que necesitamos es comenzar con el “inflation” de cada fila.

Listas con layouts personalizados

FOLDER IT

El “Inflation” implica el acto de convertir un layout `XML` en un árbol real de objetos `View`. El proceso es bastante tedioso

- Tomar un elemento
- Instanciar el `View` especificado
- Evaluar los atributos
- Convertirlos en llamadas apropiadas a método setters
- Iterar sobre todos los elementos hijos y luego repetir

Android brinda una clase denominada `LayoutInflater` que nos permite realizar todas estas acciones de manera ordenada y eficiente.

En las listas podemos aplicar el proceso de “inflate” a cada fila mostrada en la lista, por lo que podremos usar sencillamente una parte del layout XML para describir cómo se supone deberían mostrarse las filas.

Reuso de filas

FOLDER IT

Cada vez que el usuario se mueve en la lista, debemos crear en tiempo de ejecución, nuevos objetos `View` para mostrar las nuevas filas lo cual es negativo en términos de uso de CPU y la percepción de pérdida de performance.

No solo es negativo para la experiencia de usuario, sino que afectará al uso de la batería, cada ciclo de CPU extra que usamos consume batería.

Y además, tenemos un trabajo extra de consumo de CPU debido a la ejecución del GC que debe limpiar todos los objetos extras que hemos creado.

Reuso de filas

La principal ventaja es que evitamos el proceso costoso de “inflation”.

Según estadísticas citadas en el google I/O 2010, reciclar los elementos de un `ListView` permite tener una performance de hasta 150% más rápido comparado con aquellas aplicaciones que no lo utilizan.

No solo es rápido sino que usa mucha menos memoria.

Patrón Holder

FOLDER IT

Otro método costoso es el `findViewById()` , el cual implica recorrer el XML hasta encontrar el `View` que queremos.

Para evitar esto lo que se puede hacer es invocar al método solo la primer vez que sea necesario y guardar las referencias en memoria. Así cuando se necesita se retorna el `View` que tenemos en caché.

Esto de denomina `ViewHolder`

Aprovecharemos que todos los objetos tienen los métodos `getTag()` y `setTag()` lo cual nos permitirá asociar un objeto arbitrario con un widget

Y Así nos ahorramos las llamadas a `findViewById()`

Patrón Holder

FOLDER IT

Aunque el uso de holder ayuda a mejorar la performance, esta mejora no es dramática

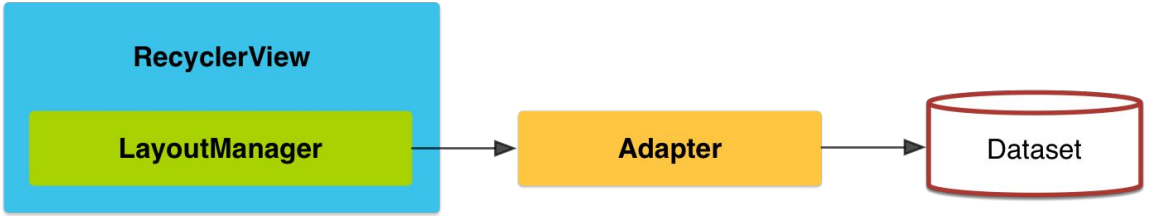
El reciclado de filas nos permite mejorar un 150%, y el uso del holder nos da una mejora de 175% sobre la situación inicial

Recycler View

A partir de la API 21 (Android 5.0 Lollypop) se incorporó este nuevo componente que viene a reemplazar a los `ListView` y `GridView`

Se sustenta en varios componentes, los principales son:

`RecyclerView.Adapter`
`RecyclerView.ViewHolder`
`LayoutManager`
`ItemDecoration`
`ItemAnimator`



Como lo que hemos visto, se apoya en un adapter para la manipulación de los datos, pero en esta ocasión nos “obliga” a usar el ViewHolder

RecyclerView vs ListView

FOLDER IT

Si necesitamos una lista con `choiceMode` tendremos que usar si o si un `ListView`

El RecyclerView carece de la opción de `choiceMode` y dificulta la selección de datos tanto simples como múltiples

En caso de querer usar un `RecyclerView` con `choiceMode` deberemos implementar el tema de la selección o multiselección todo nosotros o usar alguna librería externa que ya implemente todo por nosotros

RecyclerView vs ListView

FOLDER IT

El recyclerView es mejor para manejar adiciones y sustracciones de ítems

Tiene un algoritmo que infiere la menor cantidad de pasos necesarios para transformar una lista en otra

RecyclerView

FOLDER IT