Введение в разработку под Android

Семинар 11.

Макаров И.О. 27.11.2020

На этом семинаре

- Объект Application
- Файлы приложения
- SharedPreferences
- SQLite базы данных
- ContentProvider
- Bitmap cache

Application

https://developer.android.com/reference/android/app/Application.html

Base class for maintaining global application state. You can provide your own implementation by creating a subclass and specifying the fully-qualified name of this subclass as the "android:name" attribute in your AndroidManifest.xml's <application> tag. The Application class, or your subclass of the Application class, is instantiated before any other class when the process for your application/package is created.

- onCreate, onTrimMemory, onTerminate for emulators
- singleton to access res, context and others

Note: There is normally no need to subclass Application. In most situations, static singletons can provide the same functionality in a more modular way. If your singleton needs a global context (for example to register broadcast receivers), include **Context.getApplicationContext()** as a **Context** argument when invoking your singleton's **getInstance()** method.

Access app-specific files

https://developer.android.com/training/data-storage/app-specific

In many cases, your app creates files that other apps don't need to access, or shouldn't access. The system provides the following locations for storing such *app-specific* files:

- **Internal storage directories:** These directories include both a dedicated location for storing persistent files, and another location for storing cache data. The system prevents other apps from accessing these locations, and on Android 10 (API level 29) and higher, these locations are encrypted. These characteristics make these locations a good place to store sensitive data that only your app itself can access.
- **External storage directories:** These directories include both a dedicated location for storing persistent files, and another location for storing cache data. Although it's possible for another app to access these directories if that app has the proper permissions, the files stored in these directories are meant for use only by your app. If you specifically intend to create files that other apps should be able to access, your app should store these files in the shared storage part of external storage instead.

When the user uninstalls your app, the files saved in app-specific storage are removed. Because of this behavior, you shouldn't use this storage to save anything that the user expects to persist independently of your app. For example, if your app allows users to capture photos, the user would expect that they can access those photos even after they uninstall your app. So you should instead use shared storage to save those types of files to the appropriate media collection.

Overview of shared storage

https://developer.android.com/training/data-storage/shared

Use shared storage for user data that can or should be accessible to other apps and saved even if the user uninstalls your app.
Android provides APIs for storing and accessing the following types of shareable data:

- **Media content:** The system provides standard public directories for these kinds of files, so the user has a common location for all their photos, another common location for all their music and audio files, and so on. Your app can access this content using the platform's MediaStore API.
- **Documents and other files:** The system has a special directory for containing other file types, such as PDF documents and books that use the EPUB format. Your app can access these files using the platform's Storage Access Framework.
- **Datasets:** On Android 11 (API level 30) and higher, the system caches large datasets that multiple apps might use. These datasets can support use cases like machine learning and media playback. Apps can access these shared datasets using the BlobStoreManager API.

SharedPreferences

https://developer.android.com/training/data-storage/shared-preferences

If you have a relatively small collection of key-values that you'd like to save, you should use the SharedPreferences APIs. A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them. Each SharedPreferences file is managed by the framework and can be private or shared.
This page shows you how to use the SharedPreferences APIs to store and retrieve simple values.

Note: The **SharedPreferences** APIs are for reading and writing key-value pairs, and you should not confuse them with the **Preference** APIs, which help you build a user interface for your app settings (although they also use **SharedPreferences** to save the user's settings). For information about the **Preference** APIs, see the Settings developer guide.

You can create a new shared preference file or access an existing one by calling one of these methods:

- getSharedPreferences() — Use this if you need multiple shared preference files identified by name, which you specify with the first parameter. You can call this from any Context in your app.
- getPreferences() — Use this from an Activity if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

SharedPreferences

https://developer.android.com/training/data-storage/shared-preferences

To retrieve values from a shared preferences file, call methods such as getInt() and getString(), providing the key for the value you want, and optionally a default value to return if the key isn't present. For example:

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.integer.saved_high_score_default_key);
int highScore = sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue);
```

To write to a shared preferences file, create a SharedPreferences.Editor by calling edit() on your SharedPreferences. Pass the keys and values you want to write with methods such as putInt() and putString(). Then call apply() or commit() to save the changes. For example:

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score_key), newHighScore);
editor.apply();
```

apply() changes the in-memory SharedPreferences object immediately but writes the updates to disk asynchronously. Alternatively, you can use commit() to write the data to disk synchronously. But because commit() is synchronous, you should avoid calling it from your main thread because it could pause your UI rendering.

Note: You can edit shared preferences in a more secure way by calling the **edit()** method on an **EncryptedSharedPreferences** object instead of on a **SharedPreferences** object. To learn more, see the guide on how to work with data more securely.

SQLite

https://developer.android.com/training/data-storage/sqlite
https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase
https://developer.android.com/training/data-storage/room

Saving data to a database is ideal for repeating or structured data, such as contact information. This page assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the android.database.sqlite package.

Caution: Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:

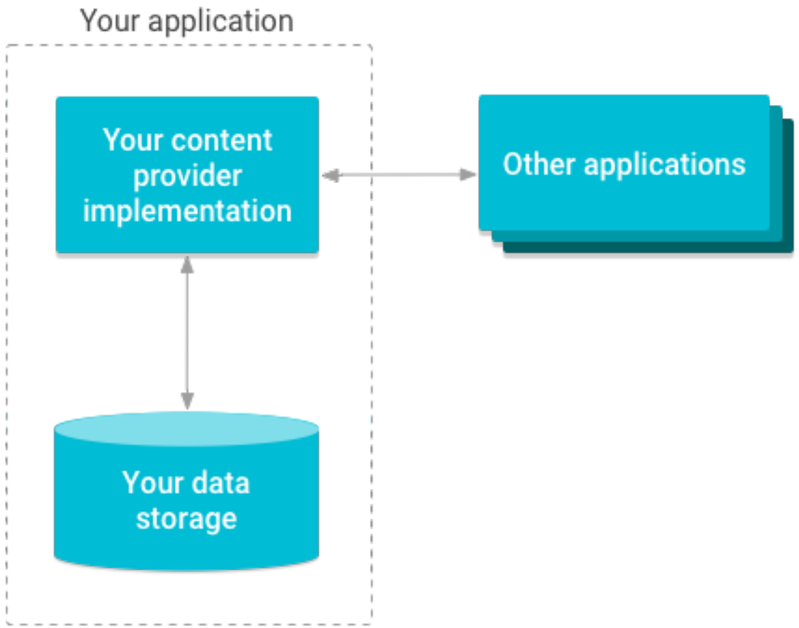
- There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
- You need to use lots of boilerplate code to convert between SQL queries and data objects.

For these reasons, we **highly recommended** using the Room Persistence Library as an abstraction layer for accessing information in your app's SQLite databases.

ContentProvider

https://developer.android.com/guide/topics/providers/content-provider-basics

Content providers can help an application manage access to data stored by itself, stored by other apps, and provide a way to share data with other apps. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process. Implementing a content provider has many advantages. Most importantly you can configure a content provider to allow other applications to securely access and modify your app data as illustrated in figure 1.



Caching Bitmaps

https://developer.android.com/topic/performance/graphics/cache-bitmap

- Memory cache
- Disk cache
- Soft & Weak references https://habr.com/ru/post/169883/

Note: For most cases, we recommend that you use the Glide library to fetch, decode, and display bitmaps in your app. Glide abstracts out most of the complexity in handling these and other tasks related to working with bitmaps and other images on Android. For information about using and downloading Glide, visit the Glide repository on GitHub.

Дома

https://developer.android.com/courses/fundamentals-training

- Изучить руководство по Room: http://d.android.com/training/data-storage/room
- Изучить основы работы с SQLite
- Изучить основы ContentProvider’a