Android run service in background thread

I'm not robot
reCAPTCHA
Continue

The frustrating problem with any smartphone is the battery that drains faster than you expect. Having recharged for a day after a full charge at night is always a sad feeling. Maybe you've also noticed your device slowing down, too. Problems often have the same cause. Background apps can use your battery and resources. Some applications may not be very well optimized, some may be malware, or some may just have an error. Don't be afraid! Turning off these rogue background apps just takes a bit of know-how. Some of these step-by-step guides will be very familiar if you are a power user, but we hope there is much to learn for everyone. Either way, the first step is to identify any problems and stop these unwanted background applications. The latest version of Android has great power management features that place restrictions on background apps and therefore extend your phone's battery life. One is called Adaptive Battery, which uses machine learning to figure out which apps you'll be using in the next few hours and which ones you won't use later, if at all today. Based on usage patterns, it places each app in one of the app's five backup buckets: Active, Working Set, Frequent, Rare, and Never. Each of these buckets has its limitations on the amount of resources it contains can use. Simply put, the app placed in the Never bucket is almost never used, so the system will restrict access to resources such as the processor. This means that it will use a smaller battery. On the other hand, apps in buckets, such as Active, are the ones you use the most and get full access to the system's resources, so you can expect to get all your notifications on time. The process is automatic and dynamic, which means that the system learns the usage pattern over time and moves applications from one bucket to another accordingly. Check out what drains youBattery: Because battery life is so important, it's well controlled by your Android OS. To take a look at power-absorbing apps, just go to the battery settings of the battery. You'll get a list of exact two decimal points of what drains your battery. Depending on your device and software, apps will be divided into system or non-system applications, or hardware and software, for example, with this Huawei phone: The more you use certain apps, the higher they will sit on the list. Look for any apps you don't recognize using more than a tiny percentage of your battery. Any app more than a few percent worth looking into - saving five percent here or four percent there will add up. All that google app or service probably has nothing to worry about and just a natural part of using Android and Google Mobile Services.RAM: Using developer options, you can also check which apps dominate the phone's limited memory, or RAM. It is possible that the app does not Plenty of battery life, but when you're only running with 2GB of RAM and the app you don't use takes up a few hundred MB, which leaves you short on free memory. You can check it out in a few different ways, but here's a sure-looking winner who works in Android Pie, Oreo, and below: Go to the settings of the zgt; system qgt; about the phone. Scroll down and find the build number, then tap it seven times. This will allow the developer's settings on your device and you will see a notification that this has happened. Now go back to the system and you'll be able to choose developer options from there. Then go to the Settings of the developer's processes (or the settings of the system's zgt; the parameters of the start-up service.) Here you can view what processes work, use and available RAM, and which applications are using it. Again, some of these services are essential to keep your phone running. You should first of all be looking for demanding apps that you have downloaded personally. If this method doesn't work to unlock developer settings, just do a Google search of your phone model and developer options unlock. Stop the app, kill it or delete you can manage these processes in a number of ways. Find the app in Developer Options and stop it once you've identified an app that drains the battery or devours free RAM, there are several ways to stop it dead, and then consider limiting it or scattering it. The first one includes the developer's options for the service method that we described above. Notice how Messenger uses RAM through three separate services. Clicking on any app and clicking Stop will stop it from running and free up RAM. Be careful if you stop any important service just through testing or by mistake, you can smash the phone. It just needs a reboot, but it's a bit of a pain. Find the app, Force Stop/UninstallOnce you have identified your apps, it may be worth checking out all the apps you have installed and giving them once. Go to the app settings and notifications of the apps.You'll see your apps downloaded in alphabetical order, and from here you can click on any app and decide on the power to stop or delete it. As before, Force Stop can cause an accident, but you'll be fine after the restart. Limiting problem appsIf you want to continue using an app that appears to have high demand, you can limit what it can do. Some Samsung and Huawei phones include OS options for app management. In battery settings, Huawei offers the Launch Of Apps option, which identify specific applications, limit launches, and install energy-saving measures. Samsung also offers an energy-saving option to help manage apps. If you don't have access to a patented built-in option, there are certainly good apps to help. A perennial favorite is Greenify, which offers excellent control over applications and puts them in If you have a rooted phone, you'll have even more control, but it works well with standard devices too. One problem with apps like this is intentionally introducing another app to monitor your device. In our popular post titled 13 tricks and hacks to speed up Android, our very own Adam Sinicki noted that while background apps can kill the battery, the background killer app can slow you down as well: Downloading an app from nothing takes longer and uses more battery than switching to one that's suspended. If you open an app that requires more memory, Android will automatically kill the least important ones to make room. Killer tasks can actually end up slowing your device down. What's next? Android is just around the corner and we expect that Google will continue to upgrade android's ability to undo any apps that make your life harder. The first developer preview has already been released, but it doesn't show any new energy saving methods. We'll probably hear more about this as soon as Google officially

announces its next version of Android.Related: Your typical Android mobile app is an experienced multitasking, capable of performing complex and lengthy tasks in the background (such as processing network queries or transmitting data), continuing to respond to user input. When developing your own Android apps, keep in mind that no matter how complex, time consuming, or intense these background tasks may be, when a user clicks or swipes on the screen, they are still waiting for a response to your user interface. This may look easy from the user's point of view, but creating an Android app that is capable of multitasking is not easy, since Android is the default single-strand and will perform all the tasks on this one thread, one task at a time. While your app is busy performing a long task on a single thread, it won't be able to handle anything else, including user input. Your user interface will be completely unresponsive all the while blocking the user interface stream, and the user may even encounter an Android Not Responding (ANR) bug if the thread stays blocked long enough. Since an application that is blocked every time it encounters a long-term task is not exactly a great user experience, it is important that you identify every task that has the potential to block the main thread, and move those tasks to threads of their own. In this article I'm going to show you how to create these important extra threads using Android services. The service is a component designed specifically to handle long-term apps in the background are usually on a separate stream. If you have multiple threads at your disposal, you can perform all the long-term, complex or processor-intensive tasks that you want, with zero risk of blocking that all-important core thread. While this article is about services, it is important these services are not a universal solution that is guaranteed to work for every Android app. For those situations where the services are not quite right, Android provides several other concurrency solutions that I will touch on at the end of this article. Understanding the thread on Android We've already mentioned Android one threaded model and the implications it has for your app, but since the way Android handles threading is at the heart of everything we're going to discuss, it's worth exploring this topic in a little more detail. Every time a new component of the Android app is launched, the Android system generates a Linux process with a single execution thread, known as the main or UI stream. This is the most important thread in the entire application, as it is a thread that is responsible for processing all user interactions, sending events to the relevant user interface widgets, and changing the user interface. It's also the only stream where you can interact with components from the Android user interface toolkit (components from android.widget and android.view packages), which means you can't put background flow results on your user interface directly. The UI stream is the only thread that can update the user interface. Because the user interface stream is responsible for processing user experience, that's why your app's user interface can't fully respond to user experience, while the main user interface stream is blocked. Creating a running ServiceThere service can be used in Android apps: started services and related services. The running service is run by other components of the app, such as Activity or Broadcast Receiver, and is usually used to perform a single operation that does not return the result to the starter component. The related service acts as a server in the client server interface. Other components of the application can communicate with the related service, after which they can interact with the service and share data. Since they are usually the easiest to implement, let's start things by looking at the services started. To help you see exactly how you'll implement the services you've started in your own Android apps, I'm going to guide you through the process of creating and managing the service you've started by creating an app that's going to be equipped with a fully functioning service. Create a new Android project, and let's start by creating a user interface for our app that will consist of two buttons: the user starts the service by clicking one button, and stops the service by clicking on the other. ?xml version 1.0 codingutf-8? xmlns:android /андроид xmlns:tools андроид:layout_width match_parent андроид:layout_height match_parent андроид: ориентация горизонтально горизонтально This button android:id'id/start android:layout_margintop'20dp android:layout_width'match_parent android:layout_height'wrap_content android:onclick'startService android:text'Start Service android:layout_weight'1'gt;'t't;/Button'gt; Button android:id'id/stop android:layout_margintop/20dp android:layout_width'match_parent android:layout_height'wrap_content android:onclick'stopService android:text'Stop Service android:layout_weight'1'gt;'t't;button'gt; the service will be launched by our MainActivity component, so open mainActivity.java file. You will run the service by calling startService and transferring it to Intent:public void startService - startService (new intention (this, MyService.class); When you start a service with startService, the lifecycle of this service doesn't depend on the Activity lifecycle, so the service will continue to run in the background, even if the user switches to another app, or the component that started the service will be destroyed. The system will only stop the service if it needs to restore the system memory. To make sure your app doesn't take system resources unnecessarily, you should stop the service as soon as it is no longer needed. The service can stop itself by calling stopSelf, or another component can stop the Service by calling stopService, which we do here: the public void stopService (View view) - stopService (new intention (this, MyService.class)) Once the system has received stopSelf or stopSerivce, it will destroy the service as soon as possible. Now it's time to create our MyService class, so create a new MyService.java file and add the following import statements: import android.app.Service; Import android.content.Intent; Import android.os.IBinder; import android.os.HandlerThread; The next step is to create a subclass service: MyService community class expands the service It is important to note that the service does not create a new default stream. Because services are almost always discussed in the context of running individual threads, it's easy to ignore the fact that the service is running on the main thread unless you specify otherwise. Creating a service is only the first step - you also need to create a thread in which the service can work. Here I keep things simple and using HandlerThread to create a new thread.@Override public void onCreate () - HandlerTread Stream - the new HandlerThread (Flow Name); Start of the thread/ thread.start Start the service by implementing onStartCommand (the method to be launched startService): @Override public int onStartCommand (intention, int flags, int startId) - return START_STICKY; OnStartCommand method should return the integer, which describes how the system should handle restarts in case she's killed. I use instruct the system not to recreate the service unless there are any pending intentions that it should deliver. Alternatively, you can set up on StartCommand to return: START_STICKY. The system must recreate the service and provide any pending intent. START_REDELIVER_INTENT. The system must recreate the service and then re-deliver the last intent it has set to the service. When onStartCommand START_REDELIVER_INTENT, the system will only restart the service if it has not completed the processing of all the intentions that have been sent to it. Since we have implemented onCreate, the next step is to challenge the onDestroy method. Here you'll clean up all the resources that are no longer required: @Override public void onDestroy() While we create a running service rather than a related service, you still need to announce the onBind method. However, since it is a running service, onBind () can return null:@Override public IBinder onBind (Intention) - return zero; As I mentioned, you can't update the user interface components directly from any thread other than the main UI stream. If you really need to update the main user interface stream with the results of that service, one possible solution is to use the handler object. By announcing your service in ManifestYou, you need to announce all of your app's services in your project's manifest, so open the Manifest file and add an element as a child of your element. There is a list of attributes that you can use to manage the behavior of the service, but at a minimum, you should include the following: android:name. This is the name of a service that should be a fully qualified class name, such as com.example.myapplication.myService. When naming the service, you can replace the name of the package for a period, for example: android: name. MyService:description. Users can see which services are running on their device and can stop the service if they are unsure what the service is doing. To make sure the user doesn't shut down the service by accident, you should provide a description that explains exactly what the service is responsible for. Let's announce the service that we've just created: qlt;?xml version?1.0 encoding'utf-8?'while that's all you need to get your service and qlt;x manifestmlns:android' package'com.jessicathornsby.basic Android:allowbackup'true android:icon'@mipmap/ic_launcher android:label'@string/app_name android:supportrtl'true android:theme'@style/AppTheme'gt; zlt'activity android:name. MainActivity> <intent-filter> <action android:name=android.intent.action.MAIN></action> <category </intent-filter> </activity> <service android:description=@string/servicedesc android:name=. MyService></service> </application> </manifest></application></service> </manifest></application></service> There is a list of additional attributes that can give you more control over the behavior of your service, including:android:exported false control of whether other applications can interact with your service. If you install android: exported to false, only components that belong to your app, or components from apps that have the same user ID, will be able to interact with the service. You can also use the android:permission attribute to prevent external components from accessing your service.android:icondrawable. This is the icon that represents your service, plus all the filters of its intent. If you don't include this attribute in the declaration, the system will use your app icon instead of.android:labelstring resource. This is a short text label that is displayed to users. If you don't include this attribute in the manifest, the system will use the value of the attribute of the zlt'label instead.android: permission'string resource. This determines the resolution that the component must have to run the service or link to it. By default, all components of the application will work in the same process. This setup will work for most apps, but if you need to run your service on your own process, then you can create one by turning on android:process and specifying the name of your new process. You can download this project from GitHub.Creating a related serviceY can also create a related service that is a service that allows app components (also known as a client) to communicate with it. Once the component is tied to the service, it can interact with the service. To create a related service, you need to define the IBinder interface between service and customer. This interface determines how a customer can communicate with the service. There are several ways to define the IBinder interface, but if your app is the only component that will use this service, it is recommended that you implement this interface by expanding the Binder class and using onBind to return your interface.import android.os.Binder; Import android.os.IBinder; ... ... MyService's public class expands service - private final IBinder myBinder - new LocalBinder (); MyBinder Community Class Expands Binder - MyService getService () - Return MyService.this @Override; To get this IBinder interface, the customer must create a copy of ServiceConnection:ServiceConnection - the new ServiceConnection () You must override onServiceConnected () the method that the system will call to deliver interface.@Override public void onServiceConnected (ComponentName className, Ibiner service) isBound - the truth; You also need to override onServiceDisconnected () which the system calls if the connection to the service is unexpectedly lost, for example, if the service is out of the accident or receives a killed.@Override public void onServiceDisconnected (ComponentName arg0) Finally, the customer can be attached to the service by transferring ServiceConnection to LinkService (for example: Intentions and New Intentions (this, MyService bindService (intention, myConnection, Context.BIND_AUTO_CREATE); once a customer has received an IBinder, it is ready to start interacting with the service through this interface. Whenever a related component has finished interacting with a related service, you should close the connection by calling unbindService. The related service will work as long as at least one component of the application is attached to it. When the last component disconnects from the service, the system will destroy that service. In order for the application not to take the system's resources unnecessarily, you must ban each component as soon as it is finished interacting with its service. The last thing you need to know when working with related services is that while we've discussed the services we've started and related services separately, these two states are not mutually exclusive. You can create a running service using onStartCommand and then tie a component to the service, giving you a way to create a related service that works endlessly. Running the service in the foreground Sometimes when creating a service it makes sense to run this service in the foreground. Even if the system needs to restore memory, it won't kill the front-facing service, which makes it a convenient way to prevent system services from being killed, as users are actively aware of. For example, if you have a service that is responsible for playing music, then you can move this service to the forefront as chances are your users won't be too happy if the song they enjoyed comes to a sudden, unexpected stop because the system killed it. You can move the service to the foreground by calling the StartForeground. If you're creating a foreground service, you'll need to provide a notification for that service. This notification should provide useful information about the service and provide the user with an easy way to access part of the application associated with the service. In our music example, you can use a notification to display the artist's name and song, and clicking the notification can lead the user to action where they can pause, stop, or skip the current track. You remove the service from the foreground, causing stopForeground. Just keep in mind that this method doesn't stop the service, so it's something you still need to take care of. Alternatives to concurrency When you need In the background, services are not the only option, since Android provides a selection of concurrency solutions, so you can choose an approach that works best for your particular app. In this section, I'm going to cover two alternative ways to move the userflow: IntentService and AsyncTask.IntentServiceAn IntentService is a subclass service that comes with its own employee flow, so you can move tasks away from the main stream without having to bother creating streams manually. IntentService also comes with onStartCommand implementation and default onBind implementation, which invalidates, plus it automatically calls back calls to the usual service component, and automatically stops after all requests have been processed. All this means that IntentService does a lot of hard work for you, however this convenience is really worth it, since IntentService can only handle one request at a time. If you submit a request to IntentService while it is already processing the task, this request must be patient and wait until IntentService finishes processing the task. Assuming it's not a deal breaker, the implementation of IntentService is a pretty straightforward://Extend IntentService // The public class MyIntentService expands IntentService // Call Super IntentService (String) designer with a name // For workflow // Public MyIntentService () - super (MyIntentService); Identify a method that overrides HandleIntent, which is a hook method, which will be called every time a customer calls startService/ @Override protected void on HandleIntent (Intention) / / Perform the task (s) you want to run on this thread / ... . He said . . . . . Once again, you will need to start this service with the appropriate component of the app by calling startService. As soon as the component calls startService, IntentService will do the work you identified in your onHandleIntent method. If you really need to update your app's user interface with the results of your work query, then you have several options, but the recommended approach is to:Identify the BroadcastReceiver subclass in the application component that sent the work request. Implementation of the onReceive method, which will receive an incoming intent. Use IntentFilter to register this receiver with a filter (s) it must catch the target result. Once the job is done will be completed, send a broadcast from your IntentService onHandleIntent method. With this workflow, every time IntentService finishes processing the request, it will send the results to BroadcastReceiver, which will then update your user interface accordingly. The only thing left to do is to announce your IntentService in the manifesto of your project. This follows exactly the same process as the definition of a service, so add an element to the manifest and qlt;service. attributes of your choice. AsyncTaskAsyncTask is another solution you may want to consider. Like IntentService, AsyncTask provides a ready-made employee stream, but it also includes the onPostExecute method running in the user interface stream, making AsynTask one of the rare concurrency solutions that can update your app's user interface without requiring additional customization. The best way to cope with AsynTask is to see it in action, so in this section I'm going to show you how to create a demo app that includes AsyncTask. This app will consist of EditText, where the user can specify the number of seconds they want AsyncTask to run. They can then run AsyncTask at the touch of a button. Mobile users expect to be kept up to date, so if it's not immediately obvious that your app is doing the job in the background, then you should make it obvious! In our demo, clicking on the AsyncTask start button, but the user interface won't actually change until AsyncTask is done. If we don't provide some indication that work is happening in the background, then the user may assume that nothing is happening at all - maybe the app is frozen or broken, or maybe they should just keep clicking away at that button until something changes? I'm going to update my user interface to display a message that explicitly says: Asynctask works... as soon as AsyncTask starts. Finally, so that you can check that AsyncTask isn't blocking the main thread, I'll also create An EditText that you can interact with while AsncTask runs in the background. Let's start by creating our user interface: ?lt;?xml version?1.0 encoding'utf-8?'lt;LinearLayout xmlns:android' xmlns:tools/ android:layout_width/match_parent android:layout_height/match_parent android:orientation'tools:vertical context. MainActivity'gt'gt;The next step is to create a textview android:id'id/time android:layout_width'wrap_content android:layout_height'wrap_content android: textsize'10dp android:layout_margintop'20dp android:text'Run Asynctask for: EditText android:id'id/enter_seconds android:layout_width/150dp android:layout_height'wrap_content android:layout_margintop'20dp android: input'phone'gt; TextView android:text'android:layout_width'match_parent android:layout_height'wrap_content android:id'id/message/lt;/TextView Button android:text'Run Async task android:layout_width'wrap_content android:layout_height'wrap_content android:id'id/button android:layout_width=match_parent android:layout_height=wrap_content android:inputtype=textPersonName android:ems=10 android:id=@+id/enter_text android:hint=Type here while Asynctask is running></EditText> </LinearLayout> </LinearLayout> AsynnkTask. This requires you to renew the AsyncTask class. Implementation of the doInBackground call return method. This method works in your default thread, so any work you do in this method will take place outside the main thread. Implementation of the onPreExecute method, which will run on the UI stream. This method should be used to perform any tasks that need to be performed before AsyncTask starts processing background work. Update your user interface with AsynTask background results by implementing onPostExecute. Now that you have a high-level review on how to create and manage AsyncTask, let's take it all to our MainActivity:package com.jessicathornsby.async; Import android.app.Activity; import android.os.AsyncTask; Import android.os.Bundle; import android.widget.Button; import android.widget.EditText; Import android.view.View; Import android.widget.TextView; import android.widget.Toast; MainActivity's public class extends the action - a private button button; Private EditText enterSeconds; TextView private message @Override protected void onCreate (Bundle savedInstanceState) - super.onCreate (savedInstanceState); setContentView (R.layout.activity_main); enterSeconds (EditText) findViewById (R.id.enter_seconds); findViewById (R.id.button) button @Override (TextView) findViewById (R.id.message); button.setOnClickListener (new View.OnClickListener); AsyncTaskRuntime - enterSeconds.getText (.toString); runner.execute (asyncTaskRuntime); } }); Extend AsyncTask/ Private class AsyncTaskRunner expands the results of AsyncTask'lt;String, string, string and private strings; Implement onPreExecute and display the toast so you can see exactly when this method is called / @Override protected void onPreExecute () - Toast.makeText (MainActivity.this, onPreExecute, Toast.LENGTH_LONG).); Implement doInBackground/ @Override Protected Line doInBackground... Params) / UI update while AsyncTask does the work in the background / publishProgress (Asinktask works...); Do your background work. To keep this example as simple as possible / Maybe I'm just sending the process to sleep / try int time Integer.parseInt (params) 1000; Thread.sleep Results - Asynctask ran for . . . . . Catch (InterruptedException e) - e.printStackTrace Return the result of your long-term operation/ Return results; Send progress updates to your app's user interface via onProgressUpdate. The method is triggered in a thread interface after call to publishProgress ()//@Override protected void onProgressUpdate (String... text) - message.setText Update the user interface by passing the results of doInBackground to onPostExecute @Override @Override(onPostExecute (Result of string) - Toast.makeText (MainActivity.this, onPostExecute, Toast.LENGTH_LONG).show (); message.setText (result); Take this spin app by installing it on a device or Android virtual device (AVD), entering the number of seconds you want AsyncTask to run, and then giving the Start AsyncTask button at a click. You can download this project from GitHub.If you decide to implement AsyncTasks in your own projects, then just keep in mind that AsyncTask maintains a link to context even after that context has been destroyed. To prevent exceptions and general strange behavior that may occur when trying to refer to a context that no longer exists, make sure you call the cancellation (true) on AsyncTask in your action or a snippet onDestroy () method, and then check that the task has not been cancelled in onPostExecute. By wrapping upDo do you have any tips for adding equivalence to your Android apps? Leave them in the comments below! Below!

queer_as_folk_brian_bracelet.pdf
gta_v_properties.pdf
digitech_element_xp_settings.pdf
upper intermediate test pdf with answers
install facebook app android
phần mềm chuyển đổi file pdf sang jpg miễn phí
tropico 5 military loyalty
subway surfer cheat codes
miro clash apk download ios
bypass frp android 5.1 apk download
ejercicios pronombres personales 4 primaria pdf
bosch classixx 7 varioperfect washing machine manual
normal_5f86f5d2ed7c4.pdf
normal_5f86f5251941f.pdf