Android Concurrency

Amit.gulati@gmail.com

1

Android Threads

- Android Application Threads
  - Android application threads are based on the pthread in Linux.
  - Every Java Thread object in Android I backed by a pthread
    - Thread java class in Android makes a call to the pthread library using JNI.
  - Android system differentiates between 4 types of threads
    - Main/UI thread
    - Binder Thread
    - Background Thread
    - Worker Thread

9

9

Android Threads

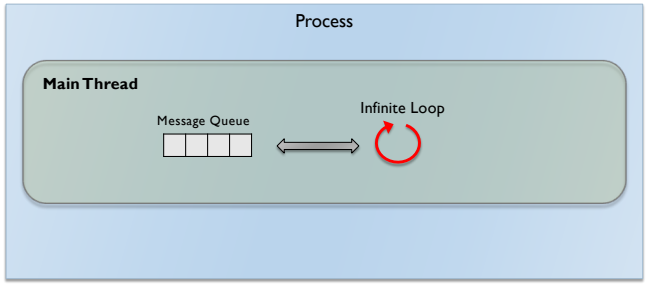
- Android Main/UI Thread
  - Created automatically when the application starts
  - Every Application will have **only one** UI Thread / Main Thread.
  - Responsible for executing Android components and updating the UI elements on the screen.
  - Sequential Event Handler Thread that can execute events sent from any other thread in the platform.

10

10

Android Threads

- Android Main Thread
  - Main Thread is basically a message loop.



11

11

Android Threads

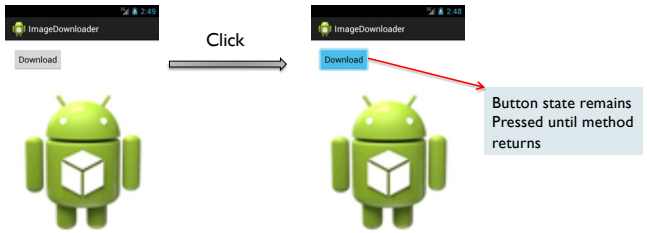
- Android Main Thread
  - Android refreshes the screen with the hardware refresh rate which is 55-60Hz
  - Main thread gets 16 Milliseconds to draw each frame
    - Measure
    - Layout
    - Draw
  - Jank
    - Main thread misses the 16 millisecond window
    - Dropped Frames
    - Choppy UI

12

12

Android Threads

- Android Main Thread
  - One of the reasons for Jank
    - Main Thread making blocking calls

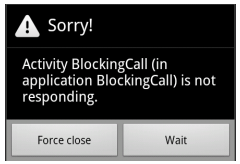


13

13

Android Threads

- Android Main Thread
  - Application Not Responding (ANR)
    - Worst case scenario
    - Main Thread makes a blocking call that takes longer than 5 seconds to return.



14

14

Android Threads

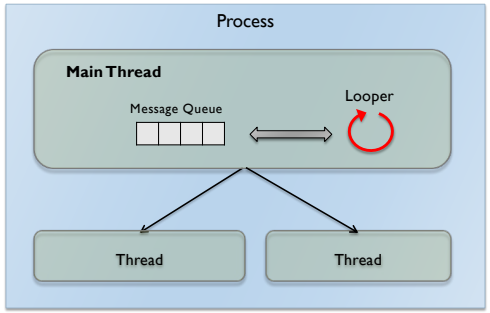
- How do we avoid ANR?
  - Not do any lengthy, time consuming, processor intensive operations in the main thread of the application.
    - Network IO
    - Image Encode/Decode
    - File IO
    - Database
    - CPU intensive computation
    - Image Filters/Processing

15

15

Android Threads

- How do we avoid ANR?
  - Move all blocking calls from the Main Thread to a secondary thread.



16

16

Android Threads

- Mechanisms to off load blocking calls to background threads in Android
  - Java Thread API
    - Java API provides classes for creating and managing multiple threads of execution.
    - Mechanism for Inter Thread communication is provided by Android.
  - Java Concurrent API
  - AsyncTask objects
    - Convenience class in Android API for creating and managing multiple threads in Android.
    - Safe mechanism to do something in the background and safely update the UI.

17

17

Android Threads

- Binder Threads
  - Binder threads are used for communicating between threads in different processes.
  - Each Android process maintains a thread pool of Binder Threads
  - Binder threads handle incoming requests from other processes: e.g., system services, intents, content providers, and services.
  - Most application need not handle Binder threads themselves and the Android system handles these.

18

18

Android Threads

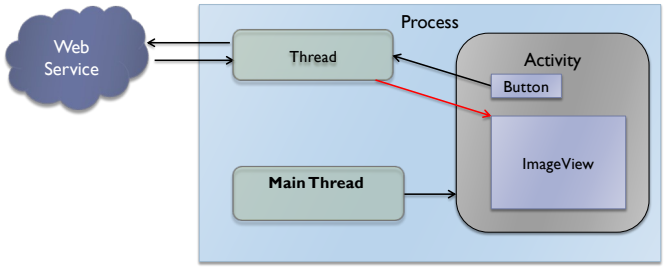
- Background Threads
  - All threads that the application explicitly creates are background threads.
  - A Background Thread get the same priority as the UI Thread / Main Thread.
  - Background Thread does not have permission to update the UI elements on the screen.
    - If a Background Thread tries to update UI element system generates a CalledFromWrongThread exception.

20

20

Android Threads

- Offload Task to Secondary Thread



21

21

Android Threads

- Secondary Threads and User Interface
  - Secondary Thread are unable to update User Interface Objects
    - If a Secondary Thread tries to access UI, you get a "CalledFromWrongThreadException"

```
AndroidRuntime FATAL EXCEPTION: Thread-219
AndroidRuntime android.view.ViewRootImpl$CalledFromWrongThreadException: Only the
               original thread that created a view hierarchy can touch its views.
AndroidRuntime     at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:4746)
AndroidRuntime     at android.view.ViewRootImpl.requestLayout(ViewRootImpl.java:823)
AndroidRuntime     at android.view.View.requestLayout(View.java:15473)
AndroidRuntime     at android.view.View.requestLayout(View.java:15473)
AndroidRuntime     at android.view.View.requestLayout(View.java:15473)
AndroidRuntime     at android.view.View.requestLayout(View.java:15473)
AndroidRuntime     at android.widget.RelativeLayout.requestLayout(RelativeLayout.java
           :318)
AndroidRuntime     at android.view.View.requestLayout(View.java:15473)
```

22

22

Android Threads

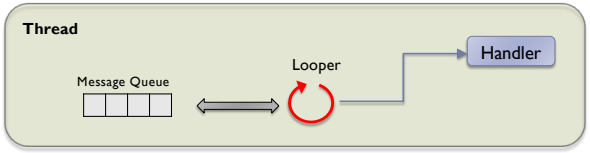
- Secondary Threads and User Interface
  - UI Thread
    - Thread that creates the UI objects.
  - Only the Thread that creates the UI object must update it.
    - Background thread needs to communicate thread for updating the UI.
  - Inter-thread communication
    - Using Handlers - Messages and Runnables.
    - AsyncTask

23

23

Android Threads

- Looper/Handler Framework
  - **Looper**
    - Message dispatcher associated with the Receiver thread.
    - Gets a reference to a message queue
    - Non-terminating loop that removes tasks from the Looper's MessageQueue and calls back to the Handler that enqueued them.



24

24

Android Threads

- Looper/Handler Framework
  - **Looper**
    - The Looper setup is done in the run method of the thread

```
class ConsumerThread extends Thread {
    @Override
    public void run() {
        Looper.prepare();

        Looper.loop(); }
}
```

25

25

Android Threads

- Looper/Handler Framework
  - **Looper**
    - Method used for stopping Looper processing messages

      public void **quit** ()

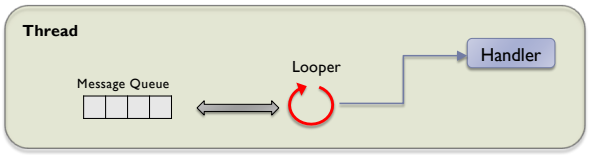
    - Ends the Looper.loop(), the thread can continue to run code placed after Looper.loop.
    - Once you stop the loop, the thread can no longer enqueue or handle messages.

26

26

Android Threads

- Looper/Handler Framework
  - **MessageQueue**
    - Unbounded linked list of messages to be processed on the receiver thread.
    - Thread can have at most one MessageQueue.



27

27

Android Threads

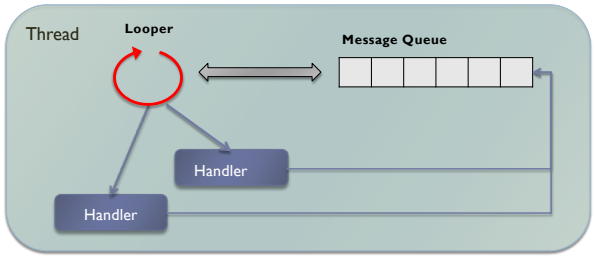
- Looper/Handler Framework
  - **Handler**
    - Help insert messages into the message queue and the message processing is done by Handler.
    - Handler cannot work without Looper
    - A thread can have multiple Handlers
    - Invoked from both the producer and consumer thread.
  - Handler responsibilities
    - Creating messages
    - Inserting messages into the queue
    - Processing messages on the receiver thread

28

28

Android Threads

- Looper/Handler Framework
  - **Handler**



29

29

Android Threads

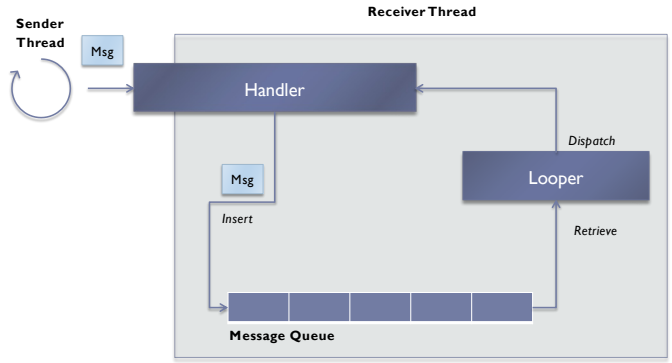
- Looper / Handler Framework
  - **Message**
    - Message is represented by android.os.Message class.
    - Message can either be a data message or task message.
      - Data Message is processed by the consumer thread
      - Task message is simply executed by the consumer thread when it is dequeued

30

30

Android Threads

- Looper / Handler Framework



31

31

Android Threads

- Looper / Handler Framework
  - Looper / Handler Framework
    - **Message**

Param	Type	Purpose
what	int	Message Identifier
arg1, arg2	int	Simple data values to handle the common use case of handing over integers
data	Bundle	Arbitrary object. If the object is handed off to a thread in another process, it has to implement Parcelable.
obj	Object	Arbitrary object. If the object is handed off to a thread in another process, it has to implement Parcelable.

32

32

Android Threads

- Looper / Handler Framework
  - **Message**
    - Methods used for attaching and retrieving objects to the Message instances
    - Attach Bundle object to a Message instance

      public void **setData** (Bundle *data*)

    - Retrieve Bundle object from Message instance

      Public Bundle **getData** ()

  - Example

```
String message = "Completed";
Bundle bundle = new Bundle();
bundle.putString("message", message);
Message msg = Message.obtain();
msg.setData(bundle);
```

```
public void handleMessage(Message msg) {
    Bundle bundle = msg.getData();
    String message =
            bundle.getString("message");
```

33

33

Android Threads

- Looper / Handler Framework
  - **Message**
    - Task Message
      - The task is represented by a java.lang.Runnable object to be executed on the receiver thread.
      - No callback method is called on the handler in the case of task message. The task method is simply executed.
    - A MessageQueue can contain any combination of data and task messages.

34

34

Android Threads

- Looper / Handler Framework
  - Binding a Handler object to a Thread
    - Handler is implicitly bound to the thread in which you create the instance of the Handler.
    - Example: Binding a Handler to the Main Thread

```
public class MainActivity extends Activity {

    Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {

        }
    };
```

35

35

Android Threads

- Looper / Handler Framework
  - Creating a Data Message for Handler
    - Handler class offers wrapper functions to create objects of the Message class.

Message **obtainMessage**(int what, int arg1, int arg2)

Message **obtainMessage**()

Message **obtainMessage**(int what, int arg1, int arg2, Object obj)

Message **obtainMessage**(int what, Object obj)

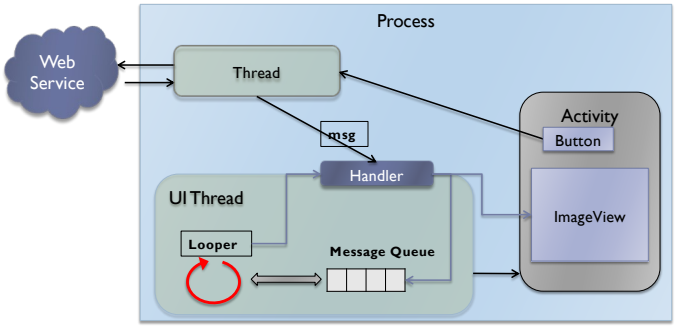
Message **obtainMessage**(int what)

36

36

Android Threads

- Looper / Handler Framework
  - **Sending a Data Message to Handler**



37

37

Android Threads

- Looper / Handler Framework
  - Sending Dats Messages to the Handler
    - Sending Message objects

Public boolean **sendMessage** (Message *msg*)

msg, Message object to be posted to the handler.

public boolean **sendMessageAtTime** (Message *msg*, long *uptimeMillis*)

uptimeMillis, The absolute time at which the message should be delivered, using the uptimeMillis() time base.

public final boolean **sendMessageDelayed** (Message *msg*, long *delayMillis*)

delayMillis, Time lapse before the message is delivered.

38

38

Android Threads

- Looper / Handler Framework
  - Processing Messages (Data Messages)
    - Override the method named handleMessage

public void **handleMessage**(Message msg)

msg, Message received.

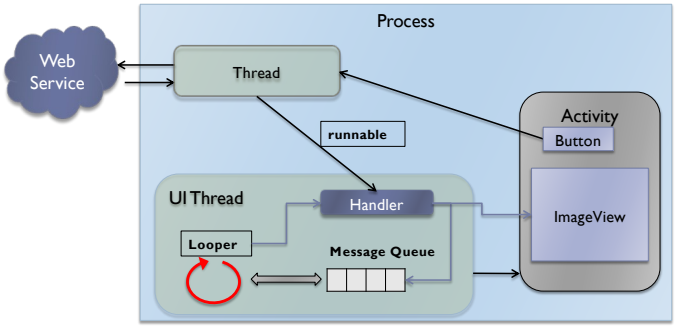
```
Handler handler = new Handler(){
    @Override
    public void handleMessage(@NonNull Message msg) {
        super.handleMessage(msg);
    }
};
```

39

39

Android Threads

- Looper / Handler Framework
  - Posting Runnable objects on Handler



40

40

Android Threads

- Looper / Handler Framework
  - Posting Runnable Objects to Handler
    - Runnable Objects are Instances of classes that implement the java.lang.Runnable interface.
    - Methods used for posting Runnable object to Handler

public final boolean **post** (Runnable *runnable*)

runnable, Instance of Runnable object to be executed in the thread associated with Handler.

public final boolean **postAtTime** (Runnable *runnable*, long *uptimeMillis*)

uptimeMillis, The absolute time at which the runnable object should be posted, using the uptimeMillis() time base.

41

41

Android Threads

- Thread with Looper and Handler
  - Java Threads by default do not provide a Looper.
  - To create a Looper for a regular Java Thread, we need to sub-class it and over-ride the run method for Thread

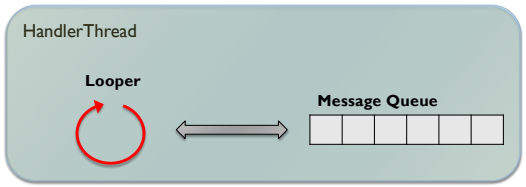
```
class LooperThread extends Thread {
      public Handler mHandler;

      public void run() {
          Looper.prepare();

          mHandler = new Handler() {
              public void handleMessage(Message msg) {
                  // process incoming messages here
              }
          };

          Looper.loop();
      }
  }
```

42

42

Android Threads

- Handler Thread
  - Handler Thread provides a message loop.
  - Helper class for creating a Thread that includes a Looper.
  - Looper can create Handler objects for handling messages.



43

43

Android Threads

- Handler Thread
  - android.os.HandlerThread class represents a Handler Thread.
  - Constructors for HandlerThread

public **HandlerThread** (String *name*)

public **HandlerThread** (String *name*, int *priority*)

name, Name for the Thread being created.
priority, The priority to run the thread at.
The value supplied must be from *Process* and not from *java.lang.Thread*.

44

44

Android Threads

- Using HandlerThread
  - Create a Sub-class of android.os.**HandlerThread** class.

```
public class ImageDownloaderRunner extends HandlerThread {

    public ImageDownloaderRunner(String name) {
        super(name);
    }
}
```

  - Creating and instance of ImageDownloaderRunner will give us a Thread with a Looper.

45

45

Android Threads

- Handler Thread
  - Method that HandlerThread class provides to execute some setup before Looper loops.

protected void **onLooperPrepared** ()

```
public class ImageDownloaderRunner extends HandlerThread {

    public ImageDownloaderRunner(String name) {
        super(name);
    }

    @Override
    protected void onLooperPrepared() {
        //onLooperPrepared() of base class i.e. HandlerThread
        //is empty and hence we don't have to call super
        super.onLooperPrepared();

    }
}
```

46

46

Android Threads in Android

- Handler Thread
  - Creating a Handler for the Looper in HandlerThread

```
public class ImageDownloaderRunner extends HandlerThread {

    Handler handler;

    public ImageDownloaderRunner(String name) {
        super(name);
    }

    @Override
    protected void onLooperPrepared() {
        handler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                super.handleMessage(msg);
            }

        };
    }
}
```

47

47

Android Threads

- Handler Thread
  - Terminating a Handler Thread

public boolean **quit** ()

public boolean **quitSafely** ()

48

48

Android Threads

- AsyncTask
  - Introduced in Android 1.5
  - Convenient way of offloading time consuming operation to the background and safely updating the UI.
  - Asynchronous Task handles all the dirty work of
    - Thread creation, and management.
    - Inter-Thread communication
  - Programmers need not worry about handler, messages, etc.
  - Programmer has to think about 3 things
    - What to do before the background operation?
    - What background operations?
    - What to do after the background operatin?

49

49

Android Threads

- AsyncTask
  - android.os.**AsyncTask** class
  - AsyncTask is a Generic/Template class

    class AsyncTask<*Params* , *Progress* , *Result*>
  - Programmer provides operations for background and for Main thread.
    - By overriding methods provided by the AsyncTask class.
    - The AsycTask object runs operations meant for background thread in a background thread and others in the main UI thread.
  - Must be created in the UI Thread for proper UI Thread and background Thread communication

50

50

Android Threads

- Using AsyncTask
  - Create a sub-class of AsyncTask<Params, Progress, Result>
    - AsyncTask is a Generic/Template class.
    - Requires the following template parameters to be provided
    - ***Params***: Type of the parameters sent to the task upon execution.
    - ***Progress***: Type of the progress units published during the background computation.
    - ***Result***: Type of the result of the background computation.

Not all types are always used by an asynchronous task.
To mark a type as unused, simply use the type **Void**

```
public class ImageDownloaderTask extends
             AsyncTask<Void, Void, Void>
```

51

51

Android Threads

- Using AsyncTask
  - Create a sub-class of AsyncTask
  - Method you must override when creating a Sub-class of AsyncTask

protected abstract Result **doInBackground** (Params... *params*)

Result, Returns a value that is the same type and what is specified in the AsyncTask template parameters.

params, Array of parameters, type of the value is same as what is specified in the template parameters.

- Example

```
public class ImageDownloaderTask extends AsyncTask<Void, Void, Void> {

    @Override
    protected Void doInBackground(Void... arg0) {
        return null;
    }
}
```

52

52

Android Threads

- Using AsyncTask
  - Methods of AsyncTask we must over-ride.

    protected abstract Result **doInBackground** (Params... *params*)

    - Invoked on a background thread.
    - Method where background work takes place.
    - Calls on UI objects must not be invoked in this method.

53

53

Android Threads

- Using AsyncTask
  - Creating an AsyncTask

```
//create a new task
ImageDownloaderTask task = new ImageDownloaderTask();
```

    - This just creates the AsyncTask, but does not start execution.

  - Method used for Executing AsyncTask

    AsyncTask<Params, Progress, Result> **execute** (Params... *params*)

    params, Variable number of parameters to pass to the AsyncTask.

    - Tasks are executed serially using a single background thread.

54

54

Android Threads

- Using AsyncTask
  - Method used for Executing AsyncTask

AsyncTask<Params, Progress, Result> **executeOnExecutor** (Executor exec, Params... params)

exec, Reference to Executor that will execute the Task.
SERIAL_EXECUTOR, Execute tasks using single thread.
THREAD_POOL_EXECUTOR, Multiple threads from a thread pool will be used to execute tasks.

params, Variable number of parameters to pass to the AsyncTask.

55

55

Android Threads

- Using AsyncTask
  - Methods of AsyncTask we can over-ride

protected void **onPreExecute**()

- Invoked on the UI thread immediately before the task is executed.
- Used for creating/attaching to UI element that will be updated while the background task is running.

protected void **onPostExecute** (Result *result*)

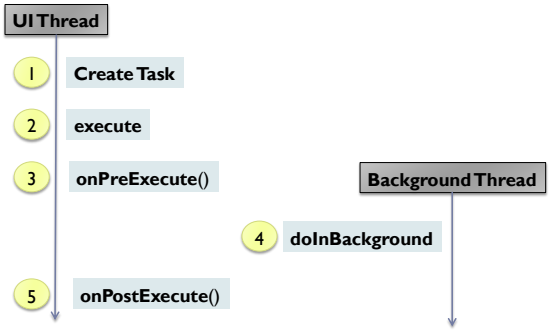
- Invoked on the UI thread after the **doInBackground** method returns.
- Used for updating the UI after the background processing is complete.
- *result*, Type of the parameter is specified in the template definition. Value for the parameter is the value returned by the **doInBackground** method.

56

56

Android Threads

- Using AsyncTask
  - Steps of Async Task



57

57

Android Threads

- Using AsyncTask
  - Steps of Async Task

```
public class ImageDownloaderTask extends AsyncTask<Void, Void, Void> {

    @Override
    protected void onPreExecute() {
        super.onPreExecute();
        //initialize the UI
    }

    @Override
    protected Void doInBackground(Void... arg0) {
        //do the background processing
        return null;
    }

    @Override
    protected void onPostExecute(Void result) {
        super.onPostExecute(result);
        //update the UI once background processing is
        //complete
    }
}
```

58

58

Android Threads

- Using AsyncTask
  - Methods of AsyncTask we can over-ride

protected void **onProgressUpdate** (Progress... *progress*)

- Invoked on the UI thread after **publishProgress**(Progress...) is invoked.
- This method is used to update the UI elements like progress bar etc.

progress, Type of data is specified by the template definition.
Values for this parameter are provided by the **publishProgress** method call.

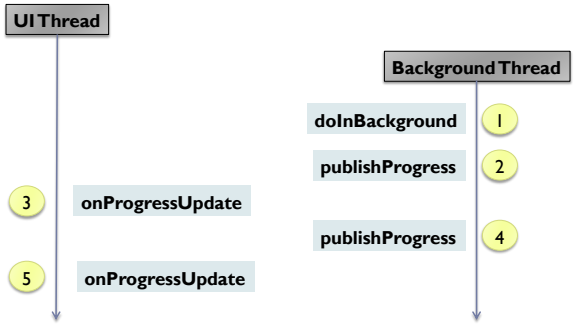
void **publishProgress** (Progress... values)

59

59

Android Threads

- Using AsyncTask
  - Steps of Async Task



60

60

Services

amit.gulati@gmail.com

61

Introduction

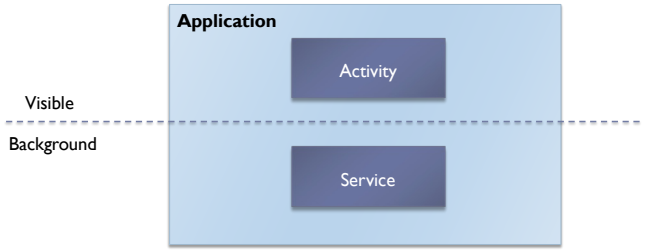
- What is a Service?
  - Android Application component

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity"...>

    <service
        android:name=".MyService" />

</application>
```

62

Introduction

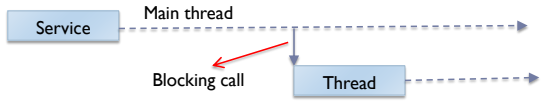
- What is a Service?
  - Background component
    - Does not have an associated User Interface
    - Used for implementing Application functionality that runs in the background



63

Introduction

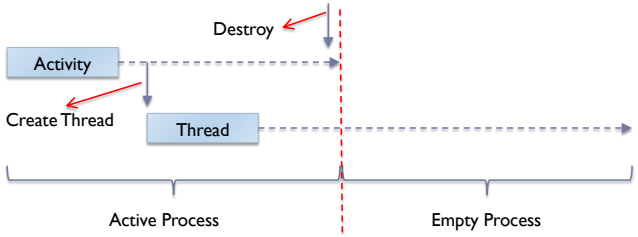
- What is a Service?
  - Runs in the main thread of the Application
    - Blocking call in Service component will block the main thread.
    - Offloading blocking calls to secondary threads required.



64

Service Component

- Why Service Component? (vs Thread)
  - Thread life cycle is independent of Application component



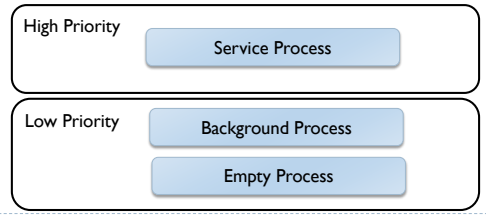
  - Application with background thread (and no alive components) is marked as an Empty process.

65

65

Service Component

- Why Service Component? (vs Thread)
  - Thread life cycle is independent of Application component
    - Application having a Service running (and no live Activities) is marked as a Service process.

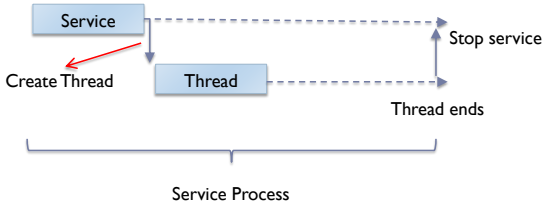


66

66

Service Component

- Why Service Component? (vs Thread)
  - Service Lifecycle can be tied to lifetime of Thread



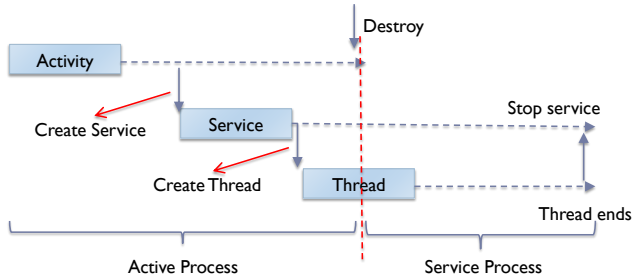
  - Application priority is Service process while the thread is running.

67

67

Service Component

- Service the preferred way for background task

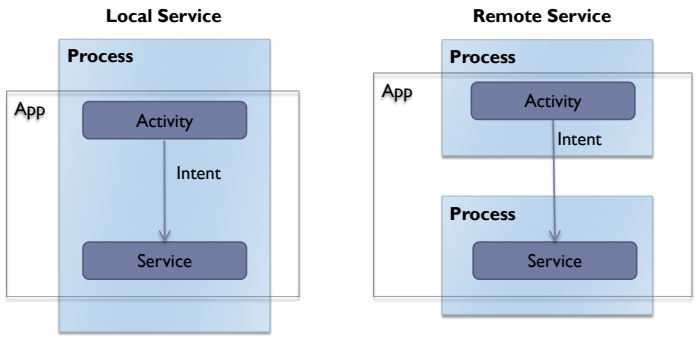


68

68

Service Component

- Service and Process



69

69

Types of Services

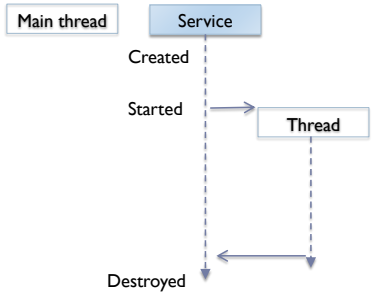
- Unbound Service (Started Service)
  - Allows applications to run code in background until it is complete or is explicitly stopped.
  - Execution model of Unbound service is sequential
    - Starts, does something and then stops.
  - Generally runs in the same process as other application components.
  - Example:
    - Service that synchronizes data with server.

70

70

Types of Services

- Unbound Service (Started Service)



71

71

Types of Services

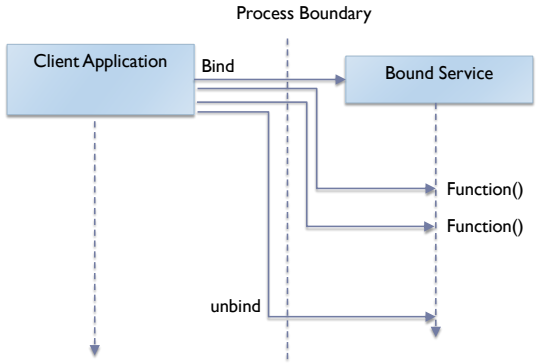
- Bound Service
  - Remote Java Object
    - Exposes a set of methods as a java Interface.
  - Clients bind/connect to a "bound service" and execute methods exposed by the Bound service.
  - Execution model of Unbound service is not sequential
    - Service is started when a client binds to it and when all clients are unbound from the service, it may be shut down by the system.
  - Useful when functionality needs to be shared with other applications on the device.
  - Example:
    - Google Play Service

72

72

Types of Services

- Bound Service

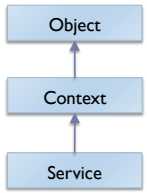


73

73

Service Component

- Service class
  - Android framework provides a class that encapsulated the Service component.



74

74

Service Component

- Creating an Unbound Service
  - Create a subclass of Service class

```
public class MyService extends Service {
    @Override
    public IBinder onBind(Intent intent) {
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

  - Register in Manifest file

androidmanifest.xml

```
<service
    android:name=".MyService" />
```

75

75

Service Component

- Launching Unbound Service
  - Method for launching Service in Context class

ComponentName **startService** (Intent *service*)

service, Intent that will be used for launching the service

  - Method does not block

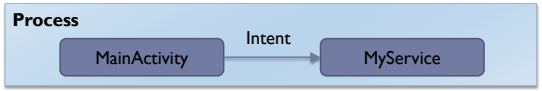
```
Intent intent = new Intent(this, MyService.class);
startService(intent);
```

76

76

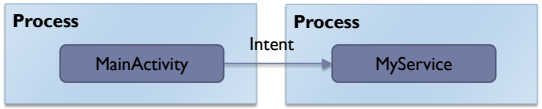
Service Content

- Unbound Service and Process
  - By default Service launches in the same process



  - Launching Service in a different process

```
<service
    android:name=".MyService"
    android:process=":com.technapse.service"/>
```



77

77

Service Component

- Stopping an Unbound Service
  - Method for stopping Service in Context class

boolean **stopService** (Intent *service*)

service, Intent that will be used for finding the service to stop.

boolean, If there is a service matching the given Intent that is already running, then it is stopped and true is returned; else false is returned.

```
Intent intent = new Intent(this, MyService.class);
stopService(intent);
```

78

78

Service Component

- Stopping an Unbound Service
  - Method for stopping Service in Service class

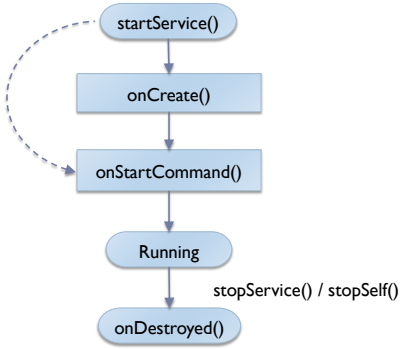
```
void stopSelf ()
```

79

79

Service Lifecycle

- Unbound Service



80

80

Service Lifecycle

- Unbound Service Life-cycle Callback
  - onCreate

```
public void onCreate()
```

  - Called only once during the life time of a service, i.e. when the service is created.
  - Signals that the Service has been created and is ready for execution.

81

81

Service Lifecycle

- Unbound Service Life-cycle Callback
  - onStartCommand

```
public int onStartCommand (Intent intent, int flags, int startId)
```

  - Unbound service should override this method and provide service code.
  - Return value, flags, and startID are used for handling service termination.

intent,	Launching Intent
flags,	Options for handling service termination (because of low resources).
startId,	A unique identifier provided by the runtime for this start request. If the process is terminated and restarted, onStartCommand is called with the same start ID.

82

82

Service Lifecycle

- Unbound Service Life-cycle
  - onDestroy

```
public void onDestroy()
```

  - Called by the system to notify a Service that it is no longer used and is being removed.

83

Service Lifecycle

- Service Termination

```
public int onStartCommand (Intent intent, int flags, int startId)
```

  - START_STICKY:
    - Service will be restarted.
    - Intent that was not fully handled (process was terminated while Intent was being processed), will not be re-delivered.
    - If Queued Intents (Intents for service that was terminated) available, they will be delivered, and onStartCommand will be called with

intent => Intent for start service

flags => START_FLAG_RETRY

84

Service Lifecycle

- Service Termination

  public int **onStartCommand** (Intent *intent*, int *flags*, int *startId*)

  - START_STICKY:
    - If no Queued Intents available, onStartCommand will be called with

      intent => NULL

      flags => START_FLAG_RETRY
  - START_NOT_STICKY
    - Service will be restarted only if there were pending start requests when the process was terminated

      intent => Intent for start service

85

85

Service Lifecycle

- Service Termination

  public int **onStartCommand** (Intent *intent*, int *flags*, int *startId*)

  - START_REDELIVER_INTENT
    - Service will be restarted and receives both pending requests and requests that were previously started and had no chance to finish.
    - If pending requests are delivered

      intent => Intent for start service

      flags => START_FLAG_RETRY
    - If previously started requests are redelivered

      flags => START_FLAG_REDELIVERY

86

86

Service Component

- Bound Service Life cycle

  public void **onCreate**()

  - Same as for unbound service.

  public IBinder **onBind**(Intent *intent*)

  - Return the communication channel to the service. May return null if clients can not bind to the service.

  intent, Intent used to bind to the service.

  public boolean **onUnbind** (Intent *intent*)

  - Called when all clients have disconnected from a particular interface published by the service.

87

87

Service Component

- Bound Service Life cycle.

  public void **onRebind**()

  - Called when a client requests a bind to the service after previous clients have unbound.

  public void **onDestroy**()

  - Same as bound service.

88

88

Service Component

- Intent Service
  - IntentService is a helper class that provides the following functionality
    - Creates a default worker thread that executes all intents delivered to onStartCommand() separate from your application's main thread.
    - Creates a work queue that passes one intent at a time to your onHandleIntent() implementation.
    - Stops the service after all start requests have been handled, so you never have to call stopSelf().
    - Provides default implementation of onBind() that returns null.

89

89

Service Component

- Intent Service
  - Creating an Intent Service

```
public class SimpleIntentService extends IntentService {

    public SimpleIntentService(String name) {
        super(name);
    }

    @Override
    public void onCreate() {
        super.onCreate();

    }

    @Override
    protected void onHandleIntent(Intent arg0) {

    }
}
```

90

90