Programação 3

Programação para Dispositivos Móveis Android

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Services

Canivete suíço!

Utilidade

- Operações que precisam continuar mesmo que o usuário deixe Activity, como fazer downloads longos ou tocar música
- Operações que precisam existir independente das activities estarem ativas ou não, como manter uma conexão de chat online
- Fornecer uma API local para APIs remotas, como as fornecidas por web services
- Realizar tarefas periódicas sem intervenção do usuário

Até mesmo widgets da tela inicial geralmente envolvem *services* para auxiliar em tarefas longas

Não são AsyncTasks melhorados...

Services

- Não interagem diretamente com o usuário (sem UI)
- Principais usos
  - Processar tarefas em segundo-plano
  - Dar suporte a comunicação inter-processos
    - remote method execution

Como *services* são criados e sobrevivem?

- Iniciados manualmente (via API Android)
- Uma activity conecta ao **`Service`** via *inter-process communication* (IPC)
- Vivem até que sejam explicitamente desligados ou caso o sistema precise de memória e o destrua
- Cuidado com o tempo, para não detonar a bateria

Como implementar *services*?

Implementando…

- Criar subclasse de **`Service`** e implementar alguns métodos
  - **`onCreate()`**
  - **`onStartCommand()`**
  - **`onBind()`**
  - **`onDestroy()`**

Métodos do ciclo de vida

- **`onCreate()`**
  - Chamado quando o **`Service`** é criado pela primeira vez, para procedimentos de configuração
  - Se o **`Service`** já está rodando, este método não é chamado
- **`onDestroy()`**
  - Chamado quando o **`Service`** não está mais sendo usado e vai ser destruído
  - Útil para liberar recursos como *threads*, *listeners* e *receivers* registrados, etc.

`onStartCommand()`

- Chamado sempre que algum componente requisita que o **`Service`** seja iniciado - **`startService()`**.

- É de responsabilidade de quem implementa definir quando o **`Service`** deve ser interrompido (opções são os métodos **`stopSelf`** e **`stopService`**).

`onBind()`

- Chamado quando um componente faz binding com o **`Service`** - **`bindService()`**.

- Deve ser fornecida uma interface para clientes se comunicarem com o **`Service`**, com base na interface **`IBinder`**

- Este método sempre deve ser implementado. Se não for desejável permitir binding, deve retornar **`null`**

Declarando **`Service`**

- Assim como outros componentes, AndroidManifest.xml
- O nome da classe corresponde ao nome do **`Service`**, similar a uma API pública, portanto devemos evitar mudar os nomes
- Segurança: explicit intents e **`android:exported`**

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".TesteServico" />
      ...
  </application>
</manifest>
```

http://android-developers.blogspot.com.br/2011/06/things-that-cannot-change.html

Sintaxe

```
<service android:enabled=["true" | "false"]
         android:exported=["true" | "false"]
         android:icon="drawable resource"
         android:isolatedProcess=["true" | "false"]
         android:label="string resource"
         android:name="string"
         android:permission="string"
         android:process="string" >
    . . .
</service>
```

http://developer.android.com/guide/topics/manifest/service-element.html

Duas formas de usar e comunicar com *services*

- Um cliente (geralmente uma **`Activity`**) tem duas maneiras de enviar requisições ou informações a um **`Service`**
  - enviar um comando e não estabelecer nenhum tipo de conexão entre o cliente e o **`Service`**
  - binding com o **`Service`**: estabelece canal de comunicação bidirecional que dura o tempo que o cliente necessitar

`startService()`

- Recebe um **`Intent`** como parâmetro, assim como **`startActivity()`**, com o papel de:
  - Identificar o **`Service`** a ser iniciado
  - Carregar informação adicional para o **`Service`**
- Usar Intent explícito!
- Chamada é assíncrona, não bloqueia o cliente

`onStartCommand()`

- Ao chamarmos **`startService()`** o **`Service`** é criado, se já não estiver rodando, e recebe o **`Intent`** via chamada a **`onStartCommand()`**

- O método **`onStartCommand()`** roda na thread principal, portanto…

- Qualquer tarefa que demore deve ser delegada a uma thread de segundo plano

Atenção: evite ANRs!

Atenção…

- Um **`Service` não** necessariamente roda em um processo separado

- Um **`Service` não** necessariamente corresponde a uma thread separada

Quando usar services vs. threads?

onStartCommand()

- O método **onStartCommand()** deve retornar um inteiro

  - **START_NOT_STICKY** - se o sistema mata o **Service** após este método, o **Service** não é criado novamente, a não ser que ainda existam intents pendentes.

  - **START_STICKY** - se o sistema interrompe o **Service** após este método, o **Service** é criado novamente, e **onStartCommand()** é chamado com um intent nulo.

  - **START_REDELIVER_INTENT** - se o sistema interrompe o **Service** após este método, o **Service** é criado novamente, e **onStartCommand()** é chamado com o último intent entregue.

Encerrando Services

- Por padrão, **`startService()`** inicia um **`Service`** que roda até que seja encerrado explicitamente

- Uma maneira de encerrar **`Service`** é por meio de **`stopService()`**, passando o mesmo **`Intent`** utilizado para iniciá-lo

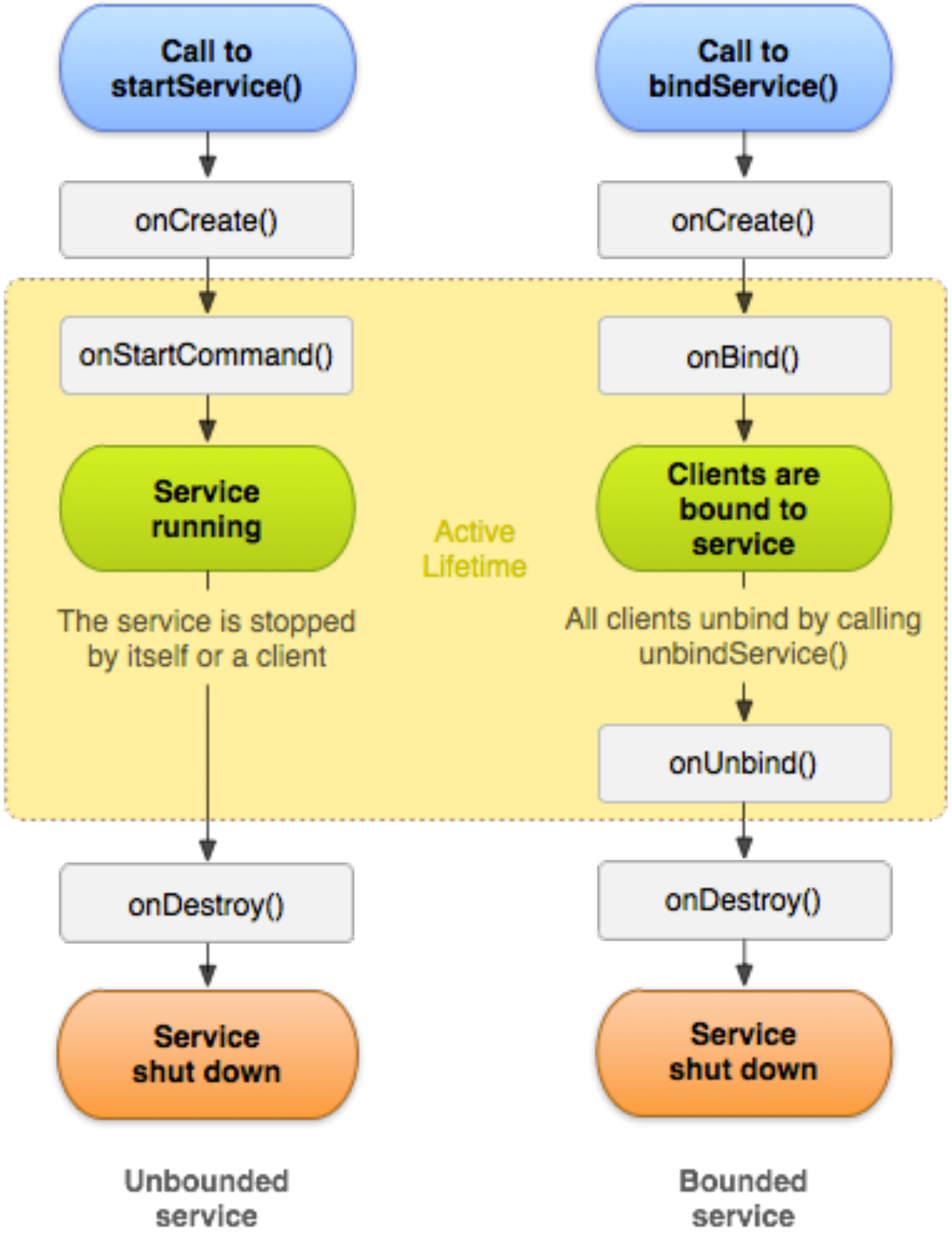
- Outra possibilidade é o próprio **`Service`** chamar **`stopSelf()`** ao encerrar o trabalho realizado

Comunicando-se a partir de um **`Service`**

- Fazer Broadcast de intents
- Pending intents
- Event Buses
- Messengers
- Notifications

Ciclo de Vida de Service

- O ciclo de vida é mais simples que o de Activity
- No entanto, importante prestar atenção em como **`Service`** é criado e destruído, dado que geralmente roda em background
- O ciclo varia de acordo com o tipo de **`Service`**


Call to startService()
Call to bindService()
onCreate()
onCreate()
onStartCommand()
onBind()
Service running
Clients are bound to service
Active Lifetime
The service is stopped by itself or a client
All clients unbind by calling unbindService()
onUnbind()
onDestroy()
onDestroy()
Service shut down
Service shut down
Unbounded service
Bounded service

```java
public class ExampleService extends Service {
    int mStartMode;       // indicates how to behave if the service is killed
    IBinder mBinder;      // interface for clients that bind
    boolean mAllowRebind; // indicates whether onRebind should be used

    @Override
    public void onCreate() {
        // The service is being created
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // The service is starting, due to a call to startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
        return mBinder;
    }
    @Override
    public boolean onUnbind(Intent intent) {
        // All clients have unbound with unbindService()
        return mAllowRebind;
    }
    @Override
    public void onRebind(Intent intent) {
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }
    @Override
    public void onDestroy() {
        // The service is no longer used and is being destroyed
    }
}
```

Mudanças de Configuração

- **`Service`** por padrão é alheio a mudanças de configuração, continua rodando simplesmente

- Se desejar observar mudanças pode sobrescrever o método **`onConfigurationChanged()`**

- Importante estar ciente apenas no caso de armazenar localmente ou fazer caching de informações de configuração, como locale, por ex.

Dica:

IntentService

`IntentService`

- Usa uma thread de background, então pode fazer operações de rede e demorar mais…
- Automaticamente se encerra ao terminar tarefa
- Uma **`Activity`** pode, basicamente, enviar um comando via **`startService()`** e ‘esquecer’
- Obviamente, existem situações onde a **`Activity`** desejará saber quando terminou a tarefa…

Ao pensarmos em um download, imaginamos que armazenaremos o arquivo em uma pasta, possivelmente pública...

Particularidades…

- Se estivermos em Android 6.0+, precisaremos ter permissão antes de iniciar o **`Service`**

- Um **`Service`** não pode pedir permissão (pode checar, no entanto…)

Matando Serviços

- O sistema interrompe **`Service`** apenas quando a memória está baixa e é necessário recuperar recursos

- Se o **`Service`** está ligado a uma activity que interage com o usuário, é menos provável que o **`Service`** seja interrompido

- Da mesma forma, **`Service`** declarado como de primeiro plano quase nunca é eliminado

- Um **`Service`** eliminado pelo sistema é reiniciado assim que recursos se tornam disponíveis novamente

E bound services?

Bound Services

- Permite que outros componentes de sua aplicação ou de outras aplicações interajam com o **`Service`**

- *Binding* permite que o **`Service`** exponha uma API, na forma de um objeto "*binder*", com métodos de sua escolha

- Isto pode ser feito intra-aplicação ou entre aplicações, permitindo comunicação entre processos

Criando Bound Services

- Ao criar um **`Service`** que permite binding, é necessário fornecer um **`IBinder`** que define a interface que clientes podem usar para interagir
  - Estendendo a classe **`Binder`**
  - Usando um **`Messenger`**
  - Usando **`AIDL`**

Implementando Binding

- Requer esforço nos dois lados, cliente e **`Service`**
- O **`Service`** precisa implementar o método **`onBind()`**, que retorna **`null`** normalmente…
- O cliente precisa solicitar o binding, ao invés de (ou além de) iniciar o **`Service`**

O lado do **`Service`**

- O binding é fornecido por meio da implementação de **`onBind()`**
- O **`Service`** implementa uma subclasse de **`Binder`** que representa a API exposta
- Para **`Service`** local, o **`Binder`** pode ter métodos, parâmetros, tipos de retorno e exceções como desejável
- Para **`Service`** remoto, a implementação do **`Binder`** é mais restrita para dar suporte a IPC
- O método **`onBind()`** retorna uma instância do **`Binder`**

O lado do cliente

- Clientes chamam **`bindService()`**, fornecendo um **`Intent`** e implementação de **`ServiceConnection`**

- O cliente não sabe nada sobre o status do *binding* até que o sistema chame o método **`onServiceConnected()`** de **`ServiceConnection`**

- Este método indica que o *binding* foi estabelecido, e no caso de *local services*, retorna o **`IBinder`** retornado no método **`onBind()`**, que o cliente usa para se comunicar com o **`Service`**

BIND_AUTO_CREATE

flag

`unbindService()`

- Eventualmente, cliente chama **`unbindService()`** para indicar que não precisa mais se comunicar com o **`Service`**

- Normalmente chamado no método **`onStop()`** de **`Activity`**

- Após chamar este método, não é seguro utilizar o objeto **`Binder`** no cliente

- Se não existem outros clientes ligados ao **`Service`**, Android o encerra também, liberando memória

- O objeto **`ServiceConnection`** também define o método **`onServiceDisconnected()`**.

Estendendo **`Binder`**

- Quando o **`Service`** é usado apenas pela aplicação local e não precisa lidar com comunicação entre processos

- Implementa a própria classe **`Binder`**, que fornece acesso direto aos métodos públicos do **`Service`**

```java
public class LocalService extends Service {
    // Binder given to clients
    private final IBinder mBinder = new LocalBinder();
    // Random number generator
    private final Random mGenerator = new Random();

    /**
     * Class used for the client Binder.  Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
    public class LocalBinder extends Binder {
        LocalService getService() {
            // Return this instance of LocalService so clients can call public methods
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    /** method for clients */
    public int getRandomNumber() {
      return mGenerator.nextInt(100);
    }
}
```

```
public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to LocalService
        Intent intent = new Intent(this, LocalService.class);
        bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {
        super.onStop();
        // Unbind from the service
        if (mBound) {
            unbindService(mConnection);
            mBound = false;
        }
    }
```

```
/** Called when a button is clicked (the button in the layout file attaches to
  * this method with the android:onClick attribute) */
public void onButtonClick(View v) {
    if (mBound) {
        // Call a method from the LocalService.
        // However, if this call were something that might hang, then this request should
        // occur in a separate thread to avoid slowing down the activity performance.
        int num = mService.getRandomNumber();
        Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT).show();
    }
}

/** Defines callbacks for service binding, passed to bindService() */
private ServiceConnection mConnection = new ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName className,
            IBinder service) {
        // We've bound to LocalService, cast the IBinder and get LocalService instance
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        mBound = false;
    }
};
```