# 手机应用平台软件开发

## 8、服务与多线程

# 什么是Service

SUN YAT-SEN UNIVERSITY

Service即“服务”，它与Activity属于同一等级的应用程序组件，都代表可执行的程序。不同的是Activity拥有前台运行的用户界面，而Service不能自己运行，需要通过某个Activity或者其他Context对象来调用。

# 什么是Service



Service在后台运行，它不能与用户直接进行交互。在默认情况下，Service运行在应用程序进程的主线程之中。

可以通过Context.startService()和Context.bindService()两种方式来启动Service。

# Service 简介

SUN YAT-SEN UNIVERSITY

- 通过Service可以使程序在退出之后仍然能够对事件或用户操作做出反应，或者在后台继续运行某些程序功能。
- Android赋予Services比处于不活动（inactivity）的Activities更高的优先级，所以它们的进程不会轻易被系统杀掉。

# Service 简介

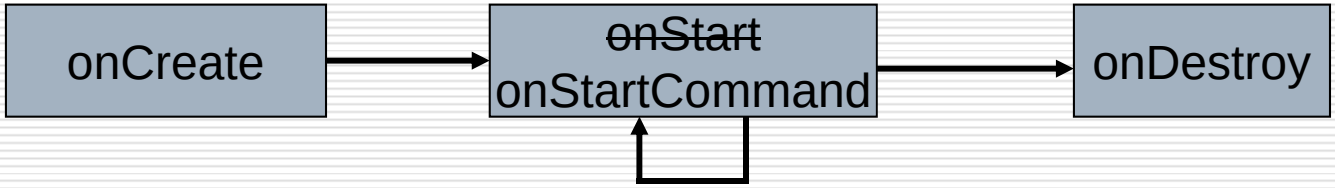
SUN YAT-SEN UNIVERSITY

- 在某些极端的情况下（例如为前台Activity提供资源），Service可能会被杀掉，但是只要有足够的资源，系统会自动重启Service。
- 在某些需要确保用户体验的情况下，（例如播放音乐）Service也可以被提高到与前台进程相同的优先级。
- Service是由其他Service，Activity或者Broadcast Receiver开始，停止和控制。

# Service 简介

## 启动模式一：通过startService启动（一旦启动，各自无关）

- 通过startService启动的Service的生命周期状态




- 通过调用Context.startService()启动，而以调用Context.stopService()结束。它可以调用Service.stopSelf() 或 Service.stopSelfResult()来自己停止。不论调用了多少次startService()方法，需要调用一次stopService()来停止服务。
- startService()启动和stopService()关闭服务，Service与访问者之间基本不存在太多关联，因此Service和访问者之间无法通讯和数据交换。

# 模式一：开始和停止Service



## 开启Service

- 在Activity中通过startService(Intent)开启一个Service。与Activity跳转类似。

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

- 其中MyService类是开发者自定义的继承Service的子类。
- 当第一次启动Service时，先后调用了onCreate(),onStart()这两个方法，当停止Service时，则执行onDestroy()方法。若Service已经启动，当再次启动Service时，不会在执行onCreate()方法，而是直接执行onStart()方法。

# 模式一：开始和停止Service



## 关闭Service

- 在Activity中通过stopService(Intent)关闭Service；

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

stopService(intent);
```

- 或者在Service中通过stopSelf()关闭自身。

# Service的实现

## 建立Android工程

- Activity：ServiceApplication.java 。程序入口，例程将在这个Activity中启动Service。
- Service：MyService.java(继承Service的子类)

  在大多情况，需要重写onCreate和onStartCommand方法

# Service的实现

## onStartCommand方法

- 在使用startService方法启动Service时被调用，在Service的生命周期内会多次被调用。
- onStartCommand方法代替了Android 2.0之前一直使用的onStart方法。通过onStartCommand方法，可以明确告诉操作系统，在用户调用stopService或者stopSelf方法显式停止之前被操作系统杀死的Service重启的时候要执行的操作。

# Service的实现



## MyService.java

```
public class MyService extends Service {
      @Override  // 必须实现的方法
      public IBinder onBind(Intent intent) {
            return null; }
      @Override  // 被启动时回调该方法
      public int onStartCommand(Intent intent, int flags, int startId) {
            return Service.START_STICKY;}
      @Override
     public boolean onUnbind(Intent intent) {
            return super.onUnbind(intent);}
      @Override// 被关闭之前回调该方法
     public void onDestroy() {
            super.onDestroy();}
}
```

# Service的实现



**ServiceApplication.java**

```
public class ServiceApplication extends Activity {
        @Override
        public void onCreate(Bundle savedInstanceState)
        {       super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
          // 启动指定的Service
                Intent intent = new Intent(this,MyService.class);
                startService(intent);
          }
}
```

# Service的实现

SUN YAT-SEN UNIVERSITY

## 在AndroidManifest.xml中注册这个Service

如果没有在此定义服务名称、访问权限，服务就无法被正确运行

```
<application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".ServiceApplication“  android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <service android:name=".MyService"
            android:process="serviceApplication"/>
</application>
```

# 模式二：通过bindService启动



1. 当程序通过startService（）和stopService（）启动、关闭Service时，Service和访问者基本上不存在太多关联，也无法进行通信和数据交换。

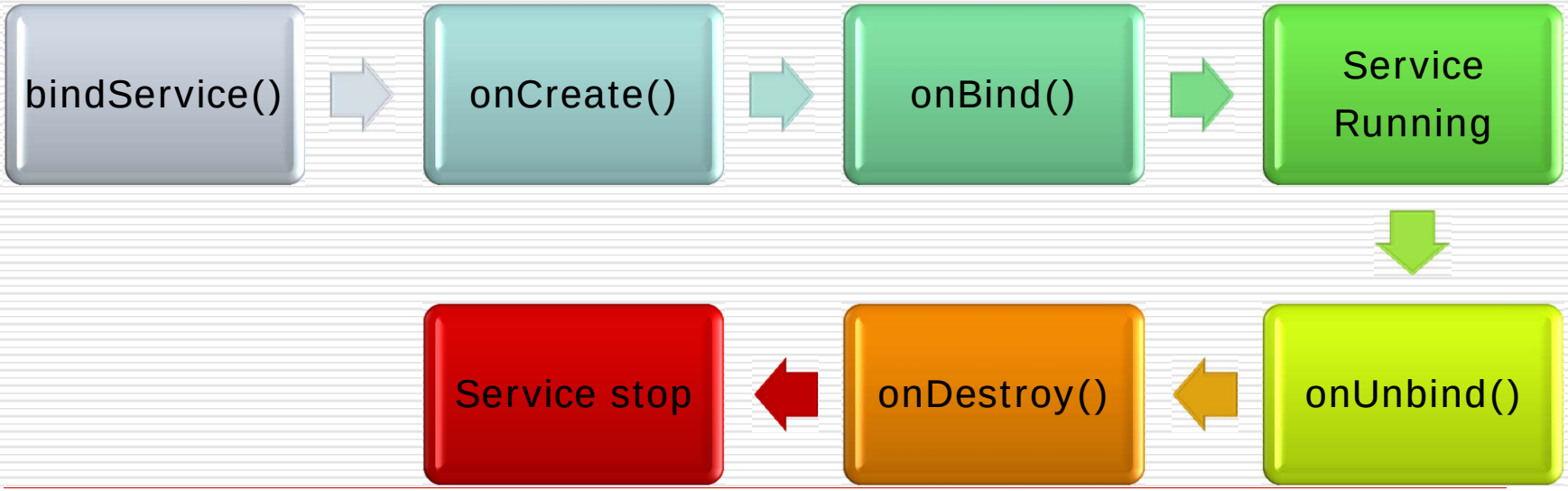
2. 若Service和访问者之间需要进行方法调用或数据交换，则应该使用bindService(intent service, ServiceConnection conn, int flags)方式

   - **service**：通过Intent指定要启动的Service；
   - **conn**：该对象用于监听访问者与Service之间的连接情况。当访问者连接成功时将回调ServiceConnection对象的onServiceConnected(ComponentName name,Ibinder service)方法；
   - **flags**：绑定时是否自动创建Service。（自动或不自动创建）

# 模式二：通过bindService启动



- 通过Context.bindService()方法启动服务，通过Context.unbindService()关闭服务。多个客户端可以绑定至同一个服务。如果服务此时还没有加载，bindService()会先加载它。
- bindService启动的Service的生命周期



# 模式二：通过bindService启动

SUN YAT-SEN UNIVERSITY

- **onBind()**只有采用Context.bindService()方法启动服务时才会回调该方法。该方法在调用者与服务绑定时被调用。
- 当调用者与服务已经绑定，多次调用Context.bindService()方法并不会导致该方法被多次调用。
- 采用Context.bindService()方法启动服务时，只能调用onUnbind()方法解除调用者与服务解除，服务结束时会调用onDestroy()方法。
- 可被其他应用程序复用，比如天气预报服务，其他应用程序不需要再写这样的服务，调用已有的即可。

# 绑定Activity和Service



- 由于Service没有界面，所以用户控制Service需要通过另外一个Activity来接收用户输入。
- 通过绑定Activity与Service，可以实现Activity与Service之间的交互。例如在Activity中控制音乐播放Service对音乐进行开始/停止，快进/快退等操作。

# 绑定Activity和Service

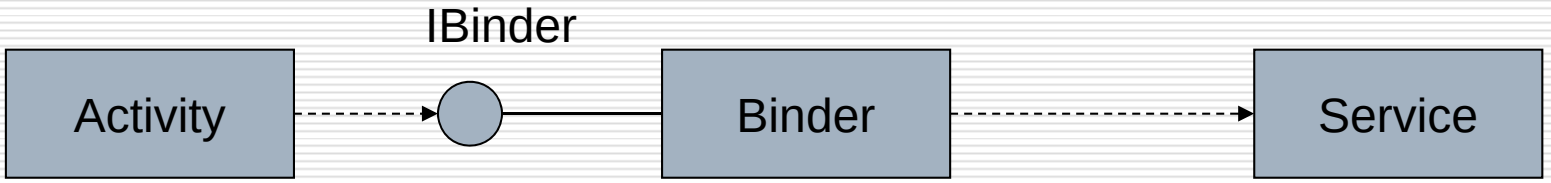
SUN YAT-SEN UNIVERSITY

## Service运行在本地进程中

与Activity属于同一进程，则只要在Activity中获得指向Service的引用，就可以像调用成员对象的方法一样去调用Service的方法。

# 绑定Activity和Service

## Activity和Service交互示意图



# PlayBindMusic.java（调用端代码）

SUN YAT-SEN UNIVERSITY

```
public class PlayBindMusic extends Activity implements
OnClickListener {
    private Button playBtn,stopBtn,pauseBtn,exitBtn;
    private BindMusicService musicService;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.bind_music_service);
        findView();bindButton();
        connection();}
    private void findView() {
        playBtn = (Button) findViewById(R.id.play);
        stopBtn = (Button) findViewById(R.id.stop);
        pauseBtn = (Button) findViewById(R.id.pause);
        exitBtn = (Button) findViewById(R.id.exit);}
    private void bindButton() {
        playBtn.setOnClickListener(this);
        stopBtn.setOnClickListener(this);
        pauseBtn.setOnClickListener(this);
        exitBtn.setOnClickListener(this);}
    private void connection(){
        Intent intent = new Intent(this,BindMusicService.class);
        bindService(intent, sc, Context.BIND_AUTO_CREATE);    }
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.play:
            musicService.play();break;
        case R.id.stop:
            if(musicService != null){
                musicService.stop();}break;
        case R.id.pause:
            if(musicService != null){
                musicService.pause();
            }break;
        case R.id.exit:
            this.finish();break;}
    }
    private ServiceConnection sc = new ServiceConnection() {
        public void onServiceDisconnected(ComponentName
name) {
            musicService = null;}
        public void onServiceConnected(ComponentName name,
IBinder service) {
            musicService =
((BindMusicService.MyBinder)(service)).getService();}
    };
}
```

onServiceConnection的输入参数IBinder就是Service中onBind返回的Binder对象。通过IBinder对象就可以实现Activity和Service进程间通讯

# BindMusicService.java（服务端代码）

SUN YAT-SEN UNIVERSITY

```
public class BindMusicService extends Service {
    private MediaPlayer mediaPlayer;
    private final IBinder binder = new MyBinder();
    public class MyBinder extends Binder {
        BindMusicService getService() {
            return BindMusicService.this;}}
    @Override
    public IBinder onBind(Intent intent) {
        return binder;}
    @Override
    public void onCreate() {
        super.onCreate();}
    @Override
    public void onDestroy() {
        super.onDestroy();
        if(mediaPlayer != null){
            mediaPlayer.stop();
            mediaPlayer.release();}}

    public void play() {
        if (mediaPlayer == null) {
            mediaPlayer = MediaPlayer.create(this, R.raw.tmp);
            mediaPlayer.setLooping(false);}
        if (!mediaPlayer.isPlaying()) {
            mediaPlayer.start();}}
    public void pause() {
        if (mediaPlayer != null && mediaPlayer.isPlaying()) {
            mediaPlayer.pause();
        }
    }

    public void stop() {
        if (mediaPlayer != null) {
            mediaPlayer.stop();
            try {
                mediaPlayer.prepare();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

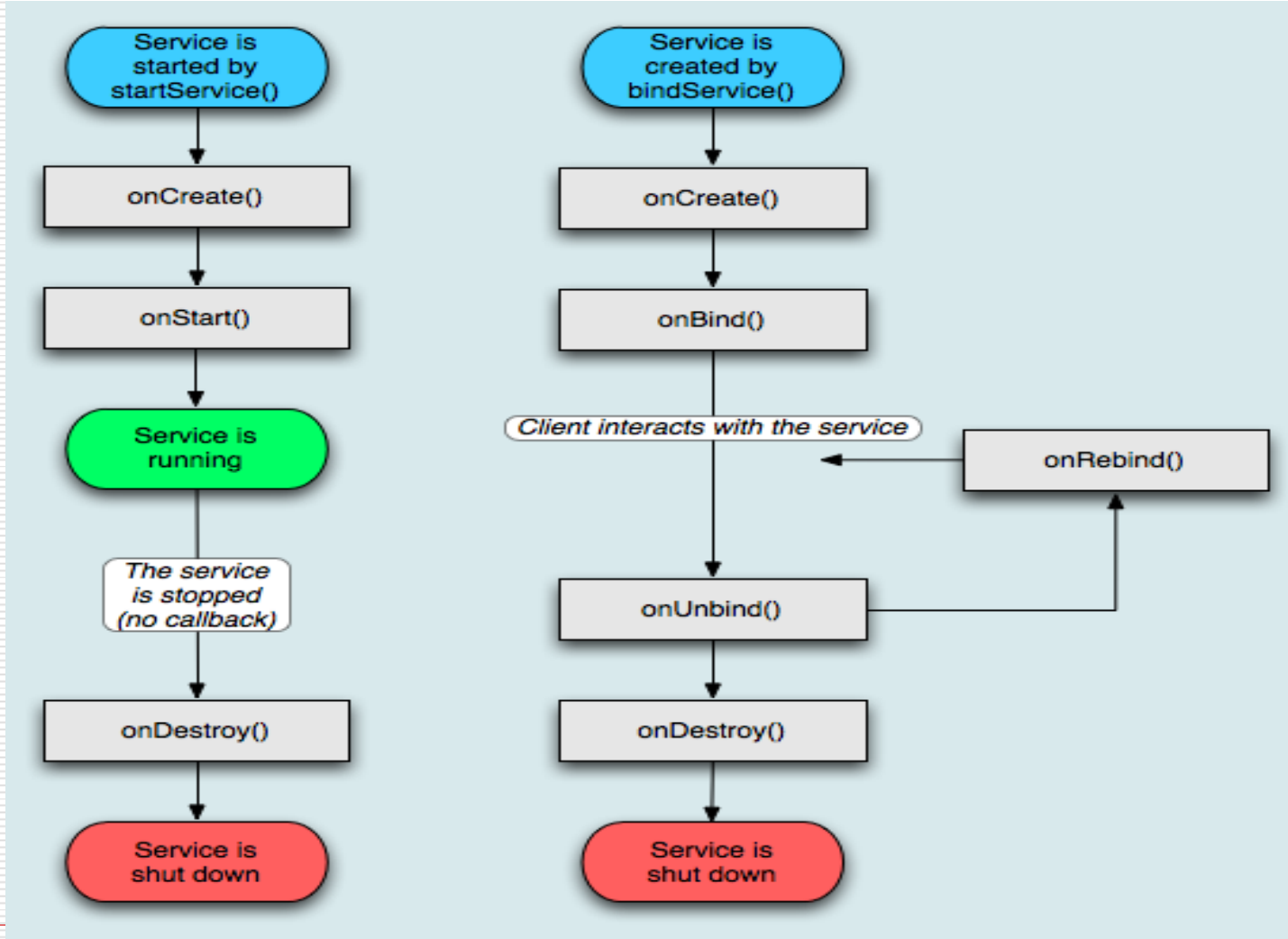
# BindMusicService.java（服务端代码）

SUN YAT-SEN UNIVERSITY

对于Service的onBind（）方法所返回的IBinder对象来说，它可以被当成该Service组件所返回的代理对象，Service允许客户端通过该IBinder对象来访问Service内部的数据，实现客户端与Service之间的通信。

# 两种启动模式对比




Service is started by startService()
onCreate()
onStart()
Service is running
The service is stopped (no callback)
onDestroy()
Service is shut down
Service is created by bindService()
onCreate()
onBind()
Client interacts with the service
onRebind()
onUnbind()
onDestroy()
Service is shut down

# 耗时操作的处理



## Service本身存在两个问题

- Service不是一个单独的进程，它和应用程序在同一个进程中。
- Service不是一个线程，所以应该避免在Service里面进行耗时的操作；

把耗时的操作直接放在Service的onStart方法中，会出现Application Not Responding！

Service不是一个线程，不能直接处理耗时的操作。如果有耗时操作在Service里，就必须开启一个单独的线程来处理。

# IntentService

SUN YAT-SEN UNIVERSITY

- IntentService使用队列的方式将请求的Intent加入队列，然后开启一个worker thread(线程)来处理队列中的Intent，对于异步的startService请求，IntentService会处理完成一个之后再处理第二个，每一个请求都会在一个单独的worker thread中处理，不会阻塞应用程序的主线程。
- 如果有耗时的操作可以在Service里面开启新线程或使用IntentService来处理

# IntentService

SUN YAT-SEN UNIVERSITY

- 创建单独的worker线程来处理所有的Intent请求；
- 创建单独的worker线程来处理onHandleIntent（）方法实现的代码，开发者无需处理多线程问题；
- 当所有请求处理完成后，IntentService会自动停止，开发者无需调用stopSelf（）停止该Service；
- 提供了一个onBind()方法的默认实现，它返回null
- 提供了一个onStartCommand()方法的默认实现，它将Intent先传送至工作队列，然后从工作队列中每次取出一个传送至onHandleIntent()方法，在该方法中对Intent对相应的处理

# IntentService

SUN YAT-SEN UNIVERSITY

```
public class ServiceDemoActivity extends Activity {

    /** Called when the activity is first created. */

    @Override

    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);

        startService(new Intent(this,MyService.class));//主界面阻塞，最终会出现Application not responding

        //连续两次启动IntentService，会发现应用程序不会阻塞，而且最重的是第二次的请求会再第一个请求结束之后运行(这个证实了
IntentService采用单独的线程每次只从队列中拿出一个请求进行处理)

        startService(new Intent(this,MyIntentService.class));

        startService(new Intent(this,MyIntentService.class));

    }

}
```

# IntentService

SUN YAT-SEN UNIVERSITY

```
public class MyService extends Service {
@Override
    public void onCreate() {
        super.onCreate();  }
@Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
        //经测试，Service里面是不能进行耗时的操作的，必须要手动开启一个工作线程来处理耗时操作
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 此处进行耗时的操作，这里只是简单地让线程睡眠了1s
                try {   Thread.sleep(1000); } catch (Exception e) {
                    e.printStackTrace();    } }
        }).start();
        return START_STICKY;     }
@Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

# IntentService

SUN YAT-SEN UNIVERSITY

```
public class MyIntentService extends IntentService {
    public MyIntentService() {
        super("yyyyyyyyyyy");
    }
@Override
    protected void onHandleIntent(Intent intent) {
        // 经测试，IntentService里面是可以进行耗时的操作的
        //IntentService使用队列的方式将请求的Intent加入队列，然后开启一个worker thread(线程)来处理队列中的Intent
        //对于异步的startService请求，IntentService会处理完成一个之后再处理第二个
        System.out.println("onStart");
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("睡眠结束");
    }
}
```

# Service运行在远程进程中

Android系统中，各种应用程序都运行在自己的进程中，操作系统也对进程空间进行保护，一个进程不能直接访问另一个进程的内存空间，进程之间一般无法进行数据交换。所以进程间进行数据交互需要利用Android操作系统提供的进程间通讯（IPC）机制来实现。

# IPC机制——IBinder

SUN YAT-SEN UNIVERSITY

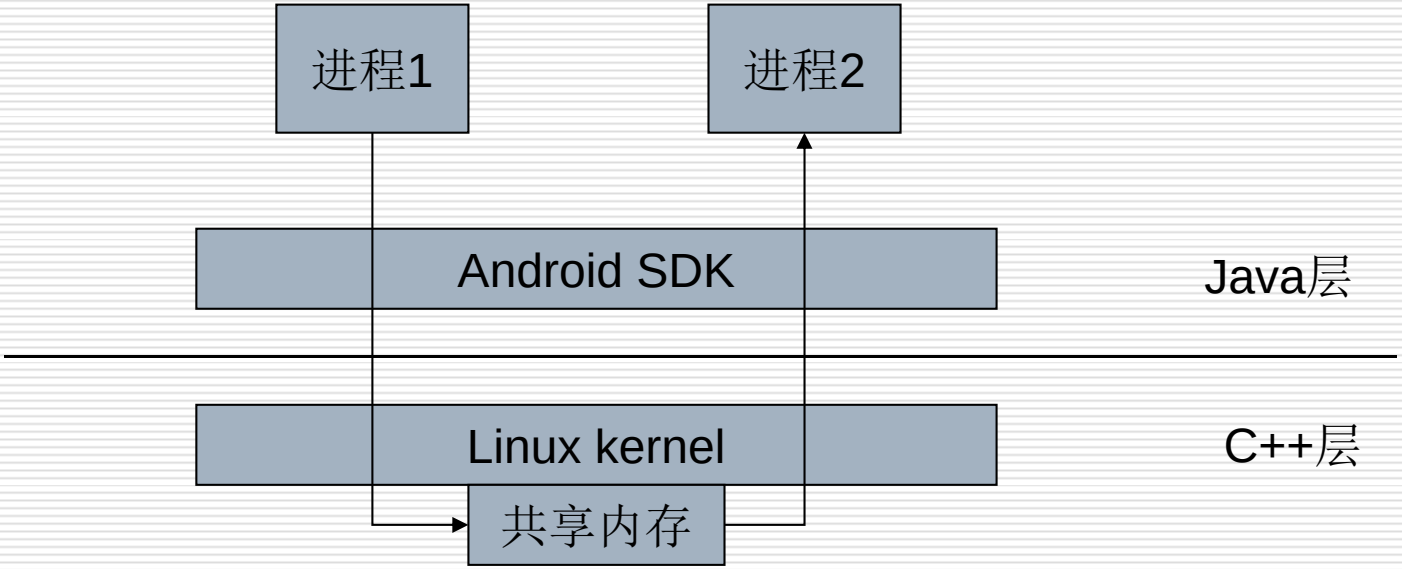
1. IBinder是远程对象的基本接口，该接口描述了与远程对象交互的抽象协议，是为高效率进行进程间通讯设计的轻量级远程过程调用机制的核心。通常并不直接实现该接口，而是继承自Binder父类，一个继承了Binder的类，它的对象就可以被远程的进程使用了(前提是远程进程获取了这个类的对象【对象的引用】（若一个Service中有一个继承了Stub的类的对象，那么这个对象中的方法就可以在Activity中使用）；
2. 当需要在一个类中有多个Stub对象，它们都要给远程交互的类的实例，这个时候可以考虑使用RemoteCallbackList<>

（docs/reference/android/os/RemoteCallbackList.html）。

# IPC机制——IBinder



Binder实质上是以IPC（Inter-Process Communication，进程间通信）框架为基础。可以简单按下图理解，其实质就是通过共享内存实现进程间的通讯。



# 跨进程调用



为了解决进程间数据共享的问题。需要通过把对象拆分成操作系统能理解的简单形式，伪装成本地对象进行跨界访问，为此就需要跨进程通信的双方约定一个统一的接口。由于编写这种接口的方法具有很大的共性，Android提供了AIDL工具来辅助完成接口的编写工作。

# 跨进程调用



AIDL(Android Interface Definition Language，即Android 接口描述语言)属于IDL语言的一种，借助它可以快速地生成接口代码，使得在同一个Android设备上运行的两个进程之间可以通过内部通信进程进行交互。如果需要在一个进程中(假设为一个Activity)访问另一个进程中(假设为一个Service)某个对象的方法，就可以使用AIDL来生成接口代码并传递各种参数。

# AIDL简介

SUN YAT-SEN UNIVERSITY

AIDL语法与Java接口非常类似，但定义接口的源代码必需以.aidl为结尾。开发人员定义的AIDL接口仅定义了进程之间的通信接口，Service端、客户端都需要使用Android SDK安装目录下的platform-tools子目录下的aidl.exe为该接口提供实现。若采用ADT工具进行开发，则ADT工具会自动为该AIDL接口生成实现。

# AIDL简介

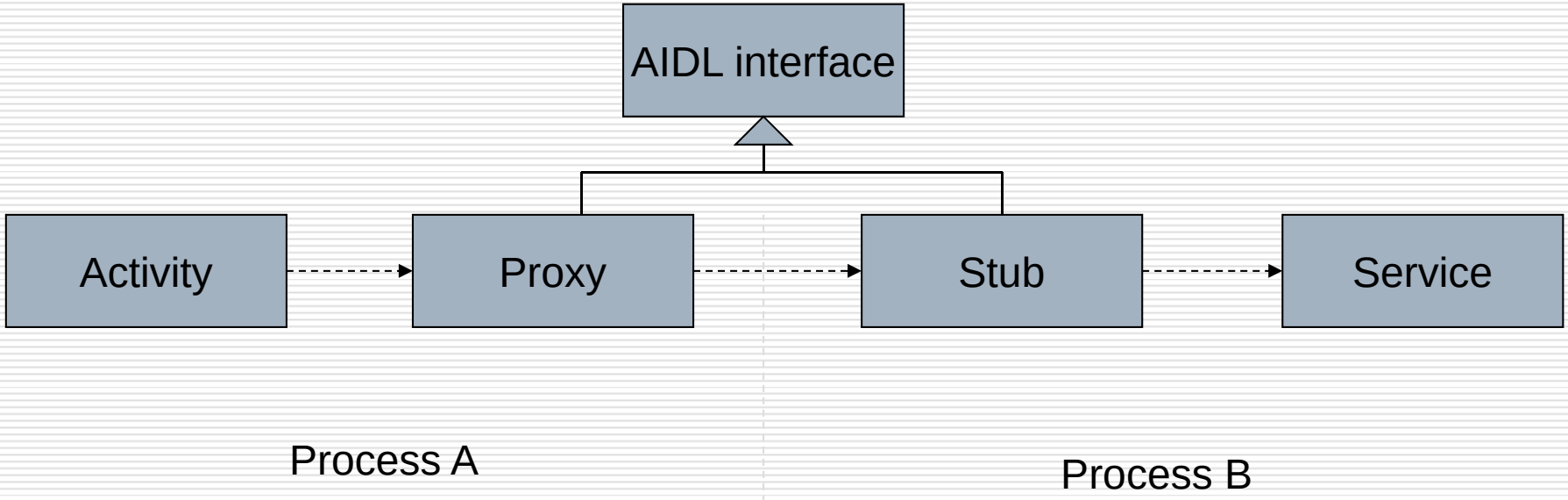
SUN YAT-SEN UNIVERSITY

- AIDL（Android Interface Description Language）弥补了IBinder接口单一的缺点。
- ADT会根据这个描述文件自动生成一个底层基于IBinder机制，表层提供描述文件所定义方法的接口类。

# AIDL简介



AIDL接口代码结构如下，采用了COM组件开发Proxy/Stub结构，这种代理设计模式多用于远程对象的调用。



# 跨进程调用



跨进程调用通常是以服务端提供服务供客户端调用的形势存在的。Android的远程Service调用与Java RMI基本类似，一样都是先定义一个远程调用接口，然后为该接口提供一个实现类；

使用AIDL，服务端需要以aidl文件的方式提供服务接口，AIDL工具将生成一个对应的java接口对象，并且在生成的接口中包含一个供客户端调用的stub服务桩类，Stub对象就是远程对象的本地代理。服务端的实现类需要提供返回stub服务桩类对象的方法。使用时，客户端通过onBind方法得到服务端stub服务桩类的对象，之后就可以像使用本地对象一样使用它了。

# AIDL开发注意事项

SUN YAT-SEN UNIVERSITY

1. 接口名和aidl文件名相同。
2. 接口和方法前不用加访问权限修饰符public,private,protected等,也不能用final,static。
3. Aidl默认支持的类型包括java基本类型（int、long、boolean等）和（String、List、Map、CharSequence），使用这些类型时不需要import声明。对于List和Map中的元素类型必须是Aidl支持的类型。如果使用自定义类型作为参数或返回值，自定义类型必须实现Parcelable接口。
4. 自定义类型和AIDL生成的其它接口类型在aidl描述文件中，应该显式import，即便在该类和定义的包在同一个包中。
5. 在aidl文件中所有非Java基本类型参数必须加上in、out、inout标记，以指明参数是输入参数、输出参数还是输入输出参数。
6. Java原始类型默认的标记为in,不能为其它标记。

# AIDL样例程序

SUN YAT-SEN UNIVERSITY

**Client（Activity）程序调用远端Server（Service）的方法：String callServer(String path);**

1、建立服务端工程（无Activity的Android项目）；

2、建立包，并在包下建立AIDL文件（ITest.aidl），

```
package cn.sysu.edu.ss;

interface ITest {

   String  callServer(String path);}
```

Eclipse将自动生成相应的类（ITest.java）；

# AIDL样例程序

SUN YAT-SEN UNIVERSITY

3、创建服务类（继承service），并创建内部类，继承于ITest.stub（Eclipse根据接口文件自动生成的类）

```
public class AIDLServer extends Service {

    private ServiceBinder serviceBinder = new ServiceBinder();

    @Override

    public IBinder onBind(Intent intent) {

     return serviceBinder;}//返回服务端Stub对象

     //内部类定义，继承自Itest.Stub

    public class ServiceBinder extends ITest.Stub{

    @Override

     //实现接口方法

     public String callServer(String path) throws RemoteException {

     return "服务器端被调用    "+path; }};}
```

# AIDL样例程序

4、修改服务的AndroidManifest.xml文件

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="cn.sysu.edu.ss"
  android:versionCode="1"
  android:versionName="1.0" >
  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="17" />
  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
  <service android:name=".AIDLServer" >
    <intent-filter>
      <action android:name="cn.sysu.edu.ss.AIDLServer" />
    </intent-filter>
  </service>
    </application>
</manifest>
```

## 客户端开发

1、创建Activity，添加按钮和按钮监听函数；

2、将服务端的aidl文件与包拷贝到客户端；

3、创建内部类

```
private ServiceConnection serviceConnection = new ServiceConnection() {
        @Override
         public void onServiceDisconnected(ComponentName name) {
              local = null; }
         @Override
          // 获取远程Service的onBind方法返回的对象的代理
         public void onServiceConnected(ComponentName name, IBinder service) {
              local = ITest.Stub.asInterface(service);  }
    };
```

其中 `private ITest local;`

# AIDL样例程序

SUN YAT-SEN UNIVERSITY

## 客户端开发

```
public class MainActivity extends Activity {
    private Button btn;
    private ITest local;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
         //绑定远程的服务，远程服务将返回远程服务对象的本地引用：local
        this.bindService(new Intent("cn.sysu.edu.ss.AIDLServer"),this.serviceConnection, BIND_AUTO_CREATE);
        btn = (Button) findViewById(R.id.button1);
        btn.setOnClickListener(new OnClickListener() {
         @Override
          public void onClick(View v) {
          try {Toast.makeText(getApplicationContext(),local.callServer("from client"),Toast.LENGTH_LONG ).show();
            } catch (RemoteException e) {
                Log.e("ClientActivity", e.toString()); }}});}
    private ServiceConnection serviceConnection = new ServiceConnection() { //此处省略.....};
    @Override
    protected void onDestroy() {
        super.onDestroy();
        this.unbindService(serviceConnection);
    }}
```

# 多线程应用-背景



1. 为了提供良好的用户体验，我们必须保证程序有高响应性，所以不能在UI线程中进行耗时的计算或I/O操作。
2. Android操作系统在下面情况下会强制关闭程序
   - UI线程在5秒内没有响应
   - 广播对象不能再10秒内完成onReceive方法

因此需要采用多线程的方法，将大规模的计算放在后台线程中进行计算，然后将计算结果再显示到前台。

```
Thread mThread = new Thread()
    {
    @Override
    public void run()
    {
        timeConsumingProcess();
    }
    };
mThread.start()
```

# 新建用户线程



- 由于Android操作系统的线程安全机制，不能在非UI线程中重绘UI，所以在用户线程中进行类似更改进度条，修改TextView文字等操作都会造成程序强制关闭（FC）
- Android提供了两种方法解决上述问题：
  a) Handler
  b) AsyncTask

# Handler机制

SUN YAT-SEN UNIVERSITY

- Android操作系统在UI线程中，缺省维护该MessageQueue和一个Looper。
- **Looper**伪码

```
while(true)
    {
        Msg =getFirstMessage(MessageQueue)
    if(Msg != null)
        processMessage()
}
```

# Handler机制

SUN YAT-SEN UNIVERSITY

- Looper通过一个死循环，当有消息Message加入队列时，通过FIFO的顺序处理消息。
- 一个Message中包括了处理Message的Handler对象还有消息内容。
- 这种机制对应这设计模式中的命令模式
- Handler与UI线程是同一个线程，所以我们在用户线程中完成计算之后，可以通过向消息队列加入一个消息，通知特定的Handler去更改UI。

# Handler实现

SUN YAT-SEN UNIVERSITY

```
/ * ServiceApplication.java * /
 Handler mHandler = new Handler()
 {@Override
      public void handleMessage(Message msg) {
      super.handleMessage(msg);
      switch(msg.what)
      {//在此处判断消息类型并更新UI}}   };
Thread mThread = new Thread(){
      @Override
      public void run(){
      timeConsumingProcess();  //进行耗时操作
      //定义接收消息的Handler对象，并将消息加入队列
      mHandler.obtainMessage(type).sendToTarget();   }
};
```

与UI同一线程的消息处理器Handler，专门负责处理非UI线程发送过来的各种消息，更新UI。

非UI线程负责耗时工作，将不同类型的消息发送给上面定义的Handler。

# AsyncTask概述

SUN YAT-SEN UNIVERSITY

- AsyncTask是在Android SDK 1.5之后推出的一个方便编写后台线程与UI线程交互的辅助类。
- AsyncTask的内部实现是一个线程池，每个后台任务会提交到线程池中的线程执行，然后使用Thread+Handler的方式调用回调函数（对程序员透明）。
- AsyncTask抽象出后台线程运行的五个状态，分别是：1、准备运行，2、正在后台运行，3、进度更新，4、完成后台任务，5、取消任务，对于这五个阶段，AsyncTask提供了五个回调函数。

# AsyncTask可重载的回调函数



- **准备运行：onPreExecute()**

  该回调函数在任务被执行之后立即由UI线程调用。这个步骤通常用来建立任务，在用户接口（UI）上显示进度条。

- **正在后台运行：doInBackground(Params...)**

  该回调函数由后台线程在onPreExecute()方法执行结束后立即调用。通常在这里执行耗时的后台计算。计算的结果必须由该函数返回，并被传递到onPostExecute()中。在该函数内也可以使用publishProgress(Progress...)来发布一个或多个进度单位(unitsof progress)。这些值将会在onProgressUpdate(Progress...)中被发布到UI线程。

# AsyncTask可重载的回调函数



- 进度更新：**onProgressUpdate(Progress...)**

  该函数由UI线程在publishProgress(Progress...)方法调用完后被调用。一般用于动态地显示一个进度条。

- 完成后台任务：**onPostExecute(Result)**

  当后台计算结束后调用。后台计算的结果会被作为参数传递给这一函数。

- 取消任务：**onCancelled ()**

  在调用AsyncTask的cancel()方法时调用

# AsyncTask的构造函数



- AsyncTask中有三个参数，如class MyTask extends AsyncTask<参数1,参数2,参数3>{}

  a) **参数1:**向后台任务执行过程方法传递参数的类型

  b) **参数2:**在后台任务执行过程中，要求主UI线程处理中间状态，通常是一些UI处理中传递的参数类型，后台计算执行过程中，进度单位（progress　units）的类型。（就是后台程序已经执行了百分之几了。）

  c) **参数3:**后台任务执行完返回时的参数类型

- AsyncTask并不总是需要使用上面的全部3种类型。标识不使用的类型很简单，只需要使用Void类型即可。

# AsyncTask使用

SUN YAT-SEN UNIVERSITY

**ServiceApplication.java**

```
class MyTask extends AsyncTask<Integer,Integer,Integer>
{     @Override
      protected Integer doInBackground(Integer... params) {
      //重写该函数，实现后台处理大规模计算
      return null;}
      @Override
      protected void onProgressUpdate(Integer... values) {
      super.onProgressUpdate(values);
      //重写该回调函数，更新UI
}}
```

很清晰的一套模板方法设计模式，只需要重写关键事件，不用去了解底层的多线程并发的实现机制

# AsyncTask使用

SUN YAT-SEN UNIVERSITY

```
private class myAsync extends
AsyncTask<Void, Integer, Void>
    {  int duration = 0;
       int current = 0;
       @Override
//在doInBackground里执行耗时操作，在适当的时候调用publishProgress(n)来设置进度条进度。
protected Void doInBackground(Void... params) {
     do {  current+=10;
          try {  publishProgress(current);
//参数类型是 AsyncTask<Void, Integer, Void>中的Integer决定的，onProgressUpdate中可以得到这个值去更新UI主线程，这里是异步线程
Thread.sleep(1000);
if(mProgressBar.getProgress() >= 100){
                    break;  }
                } catch (Exception e) {  }
            } while
(mProgressBar.getProgress() <= 100);
return null;
}

 @Override
protected void onProgressUpdate(Integer... values) {super.onProgressUpdate(values);
mProgressBar.setProgress(values[0]);
//注意：这个函数在doInBackground调用publishProgress时触发，虽然调用时只有一个参数，但是这里取到的是一个数组,所以要用progesss[0]来取值，第n个参数就用progress[n]来取值。
      }
 }
```

# Questions?