

《Android零基础入门课程》—— 涂涂IT学堂

第五章 Android应用开发核心知识点讲解

目标

- Android四大组件
- 数据存储
- 网络编程
- WebView
- Canvas
- 硬件与传感器

01 Android四大组件

1.1 Activity(活动)

- 1 官方解释: **Activity**是一个应用程序的组件，他在屏幕上提供了一个区域，允许用户在上面做一些交互性的操作， 比如打电话，照相，发送邮件，或者显示一个地图! **Activity**可以理解成一个绘制用户界面的窗口， 而这个窗口可以填满整个屏幕，也可能比屏幕小或者浮动在其他窗口的上方!
- 2 总结: 1. **Activity**用于显示用户界面，用户通过**Activity**交互完成相关操作 2. 一个App允许有多个**Activity**

- 1 继承**Activity**和**AppCompatActivity**区别
- 2 **AppCompatActivity**兼容了很多低版本的一些东西
- 3 **AppCompatActivity**相对于**Activity**的变化: 主界面带有**toolbar**的标题栏;

- **Activity** 创建流程

Activity的使用流程

①自定义Activity类名,继承Activity类或者它的子类

```
class MyActivity extends Activity{
```

②重写onCreate()方法,在该方法中调setContentView()设置要显示的视图

```
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}
```

③在AndroidManifest.xml对Activity进行配置

```
<activity
    android:icon = "图标"
    android:name = "类名"
    android:label = "Activity显示的标题"
    android:theme = "要应用的主题"></activity>
```

④启动Activity:调用startActivity(Intent);

```
Intent it = new
Intent(MainActivity.this,MyActivity.class) ;
startActivity(it);
```

⑤关闭Activity:调用finish,直接关闭当前Activity

```
我们可以把他写到启动第二个Activity的方法中,当启动第二个
Activity时,第一个Activity就会被关闭
finish() ;
```

• 启动一个Activity的几种方式

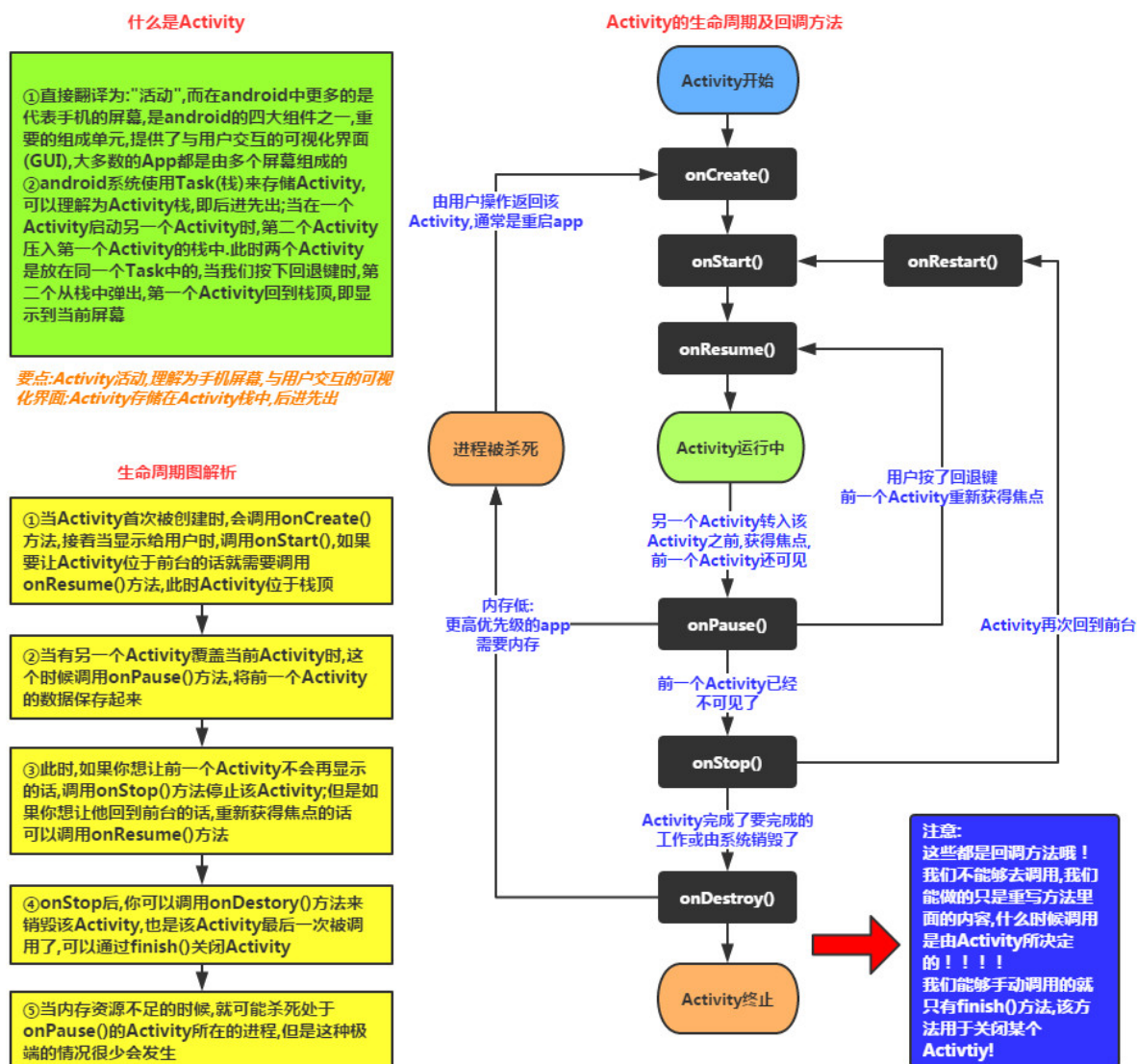
1. 显示启动

```
1  ①最常见的:
2  startActivity(new Intent(当前Act.this,要启动的Act.class));
3
4  ②通过Intent的ComponentName:
5  ComponentName cn = new ComponentName("当前Act的全限定类名","启动Act的全
6  限定类名") ;
7  Intent intent = new Intent() ;
8  intent.setComponent(cn) ;
9  startActivity(intent) ;
10
11 ③初始化Intent时指定包名:
12 Intent intent = new Intent("android.intent.action.MAIN");
13 intent.setClassName("当前Act的全限定类名","启动Act的全限定类名");
14 startActivity(intent);
```

2. 隐世启动

1 通过Intent-filter的Action,Category或data来实现

• Activity 生命周期



• 组件间通信 Intent

```
1 Intent in = new Intent(FirstActivity.this, ThirdActivity.class);
2 //1. 传单个数据
3 in.putExtra("test", "TTIT");
4 in.putExtra("number", 100);
5 //2. 传多个数据
6 Bundle b = new Bundle();
7 b.putInt("number", 100);
8 b.putString("test", "TTIT");
9 in.putExtras(b);
10 startActivity(in);
```

1 1.FirstActivity启动ThirdActivity

```

2      startActivityForResult(in, 1001);
3  2.FirstActivity接受ThirdActivity返回的数据
4      @Override
5      protected void onActivityResult(int requestCode, int
resultCode, @Nullable Intent data) {
6          super.onActivityResult(requestCode, resultCode, data);
7          Log.e("tag", "requestCode =" + requestCode);
8          Log.e("tag", "resultCode =" + resultCode);
9          Log.e("tag", "data =" + data.getStringExtra("back"));
10     }
11  3.ThirdActivity设置返回的数据
12     Intent backIn = new Intent();
13     backIn.putExtra("back", "abcdef");
14     setResult(1002, backIn);

```

- Back Stack (回退堆栈)

```

1  Java栈Stack概念:
2  后进先出(LIFO)，常用操作入栈(push)，出栈(pop)，处于最顶部的叫栈顶，最底部叫栈底

```

```

1  Activity 管理机制:
2  1.我们的APP一般都是由多个Activity构成的，而在Android中给我们提供了一个Task(任务)的
概念，就是将多个相关的Activity收集起来，然后进行Activity的跳转与返回；
3  2.当切换到新的Activity，那么该Activity会被压入栈中，成为栈顶！而当用户点击Back
键，栈顶的Activity出栈，紧随其后的Activity来到栈顶！
4  3.Task是Activity的集合，是一个概念，实际使用的Back Stack来存储Activity，可以有多个
Task，但是 同一时刻只有一个栈在最前面，其他的都在后台

```

```

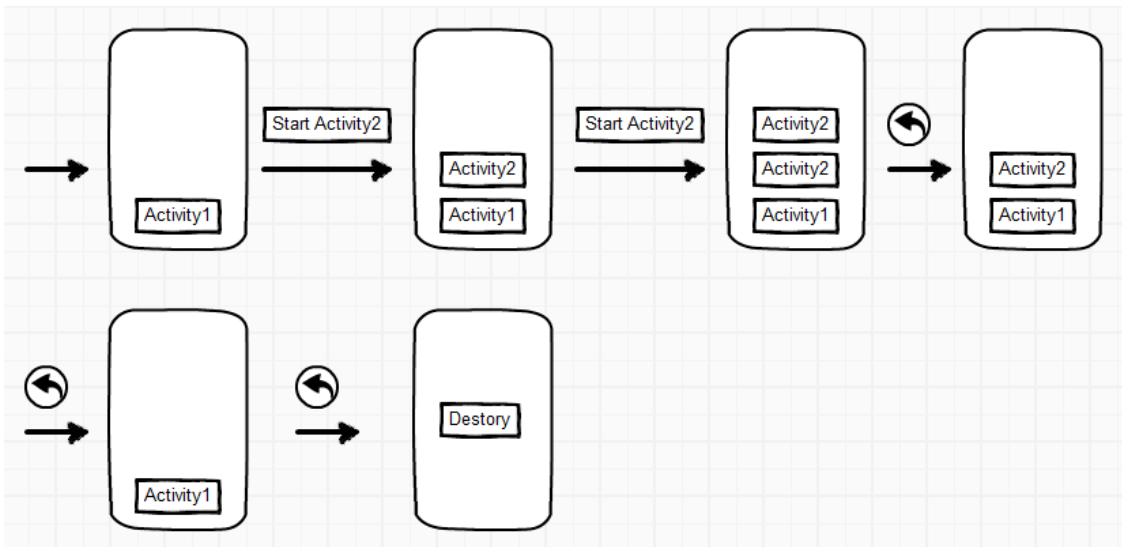
1  1.FLAG_ACTIVITY_NEW_TASK
2      默认启动标志，该标志控制创建一个新的Activity实例，首先会查找是否存在和被启动的
Activity具有相同的亲和性的任务栈 如果有，则直接把这个栈整体移动到前台，并保持栈中旧
activity的顺序不变，然后被启动的Activity会被压入栈，如果没有，则新建一个栈来存放被
启动的activity
3      Intent intent = new Intent(A.this, A.class);
4      intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
5      startActivity(intent);
6  2.FLAG_ACTIVITY_CLEAR_TOP
7      如果已经启动了四个Activity: A, B, C和D。在D Activity里，我们要跳到B
Activity，同时希望C finish掉,可以采用下面启动方式，这样启动B Activity，就会把D,
C都finished掉
8      Intent intent = new Intent(D.this, B.class);
9      intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
10     startActivity(intent);
11  3.FLAG_ACTIVITY_SINGLE_TOP
12     从名字中不难看出该Flag相当于Activity加载模式中的singleTop模式，即原来Activity
栈中有A、B、C、D这4个Activity实例，当在Activity D中再次启动Activity D时，
Activity栈中依然还是A、B、C、D这4个Activity实例。
13     Intent intent = new Intent(D.this, D.class);
14     intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
15     startActivity(intent);

```

Activity启动模式：

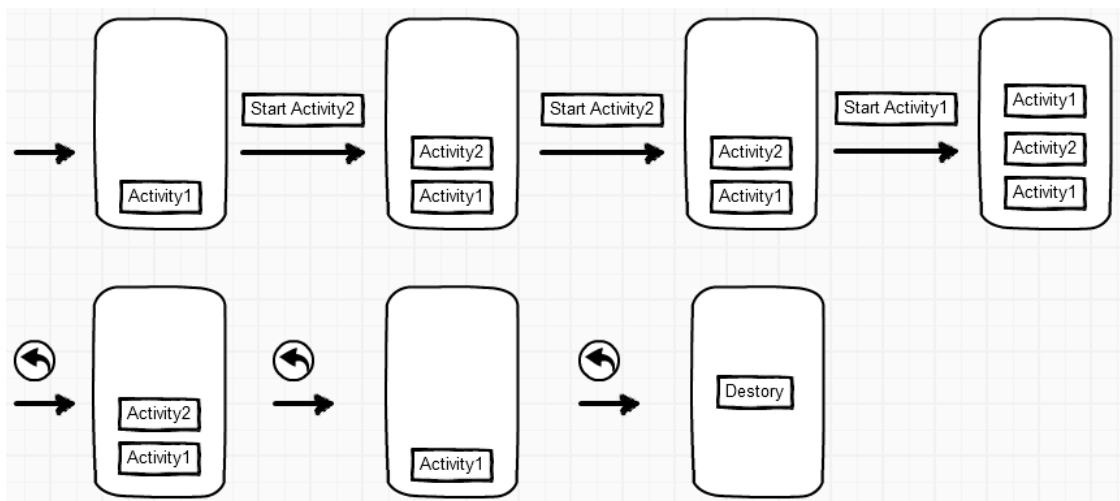
- 1 模式详解:
- 2 **standard**模式:
- 3 标准启动模式，也是**activity**的默认启动模式。在这种模式下启动的**activity**可以被多次实例化，即在同一个任务中可以存在多个**activity**的实例，每个实例都会处理一个**Intent**对象。如果**Activity A**的启动模式为**standard**，并且**A**已经启动，在**A**中再次启动**Activity A**，即调用**startActivity (new Intent (this, A.class))**，会在**A**的上面再次启动一个**A**的实例，即当前的栈中的状态为**A-->A**。
- 4
- 5 **singleTop**模式:
- 6 如果一个以**singleTop**模式启动的**Activity**的实例已经存在于任务栈的栈顶，那么再启动这个**Activity**时，不会创建新的实例，而是重用位于栈顶的那个实例，并且会调用该实例的**onNewIntent()**方法将**Intent**对象传递到这个实例中。举例来说，如果**A**的启动模式为**singleTop**，并且**A**的一个实例已经存在于栈顶中，那么再调用**startActivity (new Intent (this, A.class))**启动**A**时，不会再次创建**A**的实例，而是重用原来的实例，并且调用原来实例的**onNewIntent()**方法。这时任务栈中还是这有一个**A**的实例。如果以**singleTop**模式启动的**activity**的一个实例 已经存在与任务栈中，但是不在栈顶，那么它的行为和**standard**模式相同，也会创建多个实例。
- 7
- 8 **singleTask**模式:
- 9 只允许在系统中有一个**Activity**实例。如果系统中已经有了一个实例，持有这个实例的任务将移动到顶部，同时**intent**将被通过**onNewIntent()**发送。如果没有，则会创建一个新的**Activity**并置放在合适的任务中。
- 10
- 11 **singleInstance**模式:
- 12 保证系统无论从哪个**Task**启动**Activity**都只会创建一个**Activity**实例,并将它加入新的**Task**栈顶 也就是说被该实例启动的其他**activity**会自动运行于另一个**Task**中。当再次启动该**activity**的实例时，会重用已存在的任务和实例。并且会调用这个实例的**onNewIntent()**方法，将**Intent**实例传递到该实例中。和**singleTask**相同，同一时刻在系统中只会存在一个这样的**Activity**实例。

standard模式:



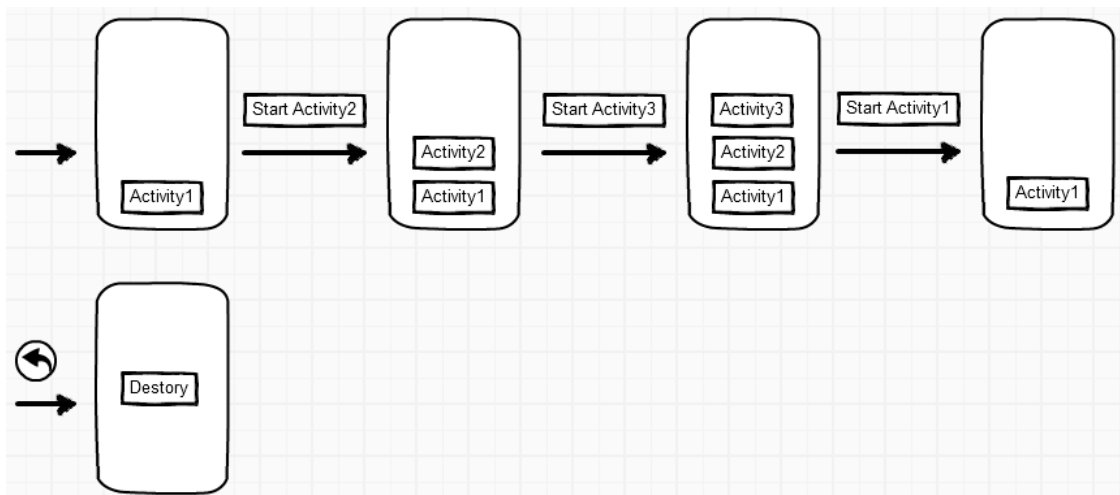
singleTop模式:

- 1 在该模式下，如果栈顶**Activity**为我们要新建的**Activity**（目标**Activity**），那么就不会重复创建新的**Activity**。



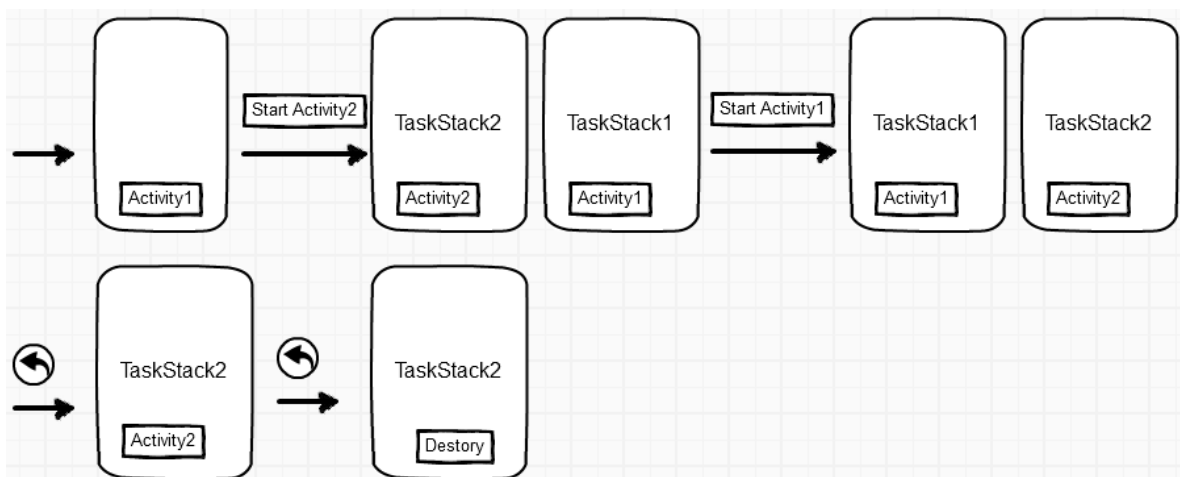
singleTask模式：

- 1 与singleTop模式相似，只不过singleTop模式是只是针对栈顶的元素，而singleTask模式下，如果task栈内存在目标Activity实例，则：
- 2 将task内的对应Activity实例之上的所有Activity弹出栈。
- 3 将对应Activity置于栈顶，获得焦点。



singleInstance（全局唯一）模式：

- 1 是我们最后的一种启动模式，也是我们最恶心的一种模式：在该模式下，我们会为目标Activity分配一个新的affinity，并创建一个新的Task栈，将目标Activity放入新的Task，并让目标Activity获得焦点。新的Task有且只有这一个Activity实例。如果已经创建过目标Activity实例，则不会创建新的Task，而是将以前创建过的Activity唤醒（对应Task设为Foreground状态）

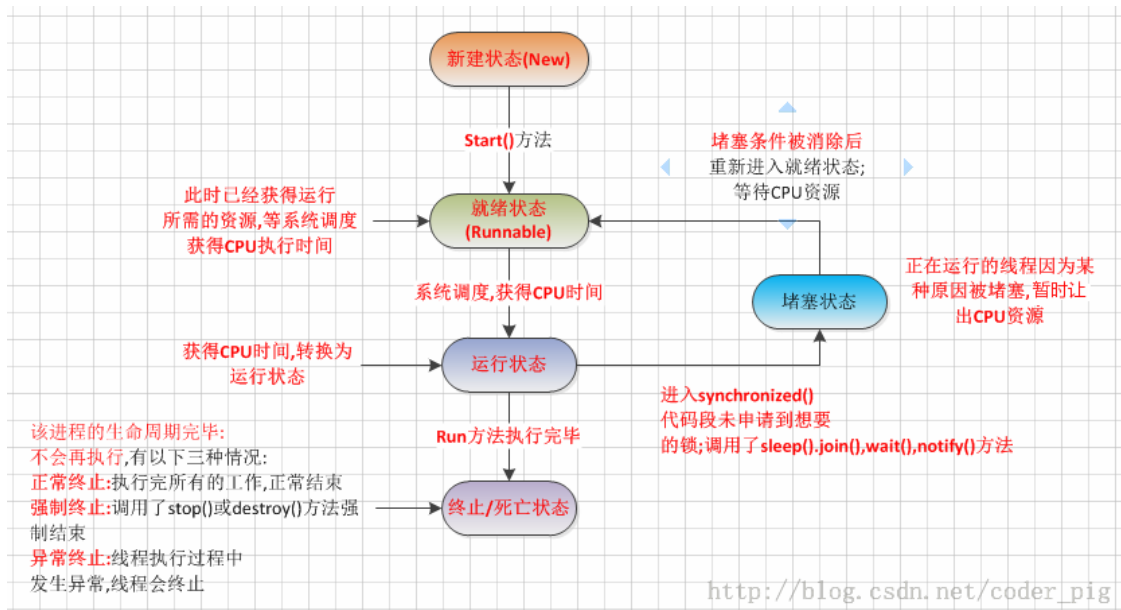


1.2 Service(服务)

- 线程的相关概念

- **程序**：为了完成特定任务，用某种语言编写的一组指令集合(一组**静态代码**)
- **进程**：**运行中的程序**，系统调度与资源分配的一个**独立单位**，操作系统会为每个进程分配一段内存空间！程序的依次动态执行，经历代码的加载，执行，执行完毕的完整过程！
- **线程**：比进程更小的执行单元，每个进程可能有多条线程，**线程**需要放在一个**进程**中才能执行，**线程**由**程序**负责管理，而**进程**则由**系统**进行调度！
- **多线程的理解**：**并行**执行多个条指令，将**CPU时间片**按照调度算法分配给各个线程，实际上是**分时**执行的，只是这个切换的时间很短，用户感觉到"同时"而已！

- 线程的生命周期



- 创建线程的三种方式

1. 继承Thread类

```
1 public class MyThread extends Thread{
2
3     @Override
4     public void run() {
5         // TODO Auto-generated method stub
6         //super.run();
7         doSomething();
8     }
9
10    private void doSomething() {
11        // TODO Auto-generated method stub
12        System.out.println("我是一个线程中的方法");
13    }
14 }
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```

22     }
23
24     private static void doSomething() {
25         // TODO Auto-generated method stub
26     }
27 }

```

2. 实现Runnable接口

```

1  public class RunnableThread implements Runnable{
2
3      @Override
4      public void run() {
5          // TODO Auto-generated method stub
6          doSomething();
7      }
8
9      private void doSomething() {
10         // TODO Auto-generated method stub
11         System.out.println("我是一个线程方法");
12     }
13 }
14 =====
15 =====
16 public class NewThread {
17     public static void main(String[] args) {
18         Runnable runnable=new RunnableThread();
19         Thread thread=new Thread(runnable);
20         thread.start();//开启一个线程方法
21         //以下的方法可与上边的线程并发执行
22         doSomething();
23     }
24
25     private static void doSomething() {
26         // TODO Auto-generated method stub
27     }
28 }

```

3. 实现Callable接口和Future创建线程

```

1  public class CallableThread implements Callable<String>{
2
3      @Override
4      public String call() throws Exception {
5          // TODO Auto-generated method stub
6          doSomething();
7          return "需要返回的值";
8      }
9
10     private void doSomething() {
11         // TODO Auto-generated method stub
12         System.out.println("我是线程中的方法");
13     }
14 }

```



```

14 }
15 =====
16 =====
17 public class NewThread {
18     public static void main(String[] args) {
19         Callable<String> callable=new CallableThread();
20         FutureTask<String> futureTask=new FutureTask<String>
21         (callable);
22         Thread thread=new Thread(futureTask);
23         thread.start();//开启一个线程方法
24         //以下的方法可与上边的线程并发执行
25         doSomething();
26         try {
27             futureTask.get();//获取线程返回值
28         } catch (InterruptedException | ExecutionException e) {
29             // TODO Auto-generated catch block
30             e.printStackTrace();
31         }
32     }
33
34     private static void doSomething() {
35         // TODO Auto-generated method stub
36     }
37 }

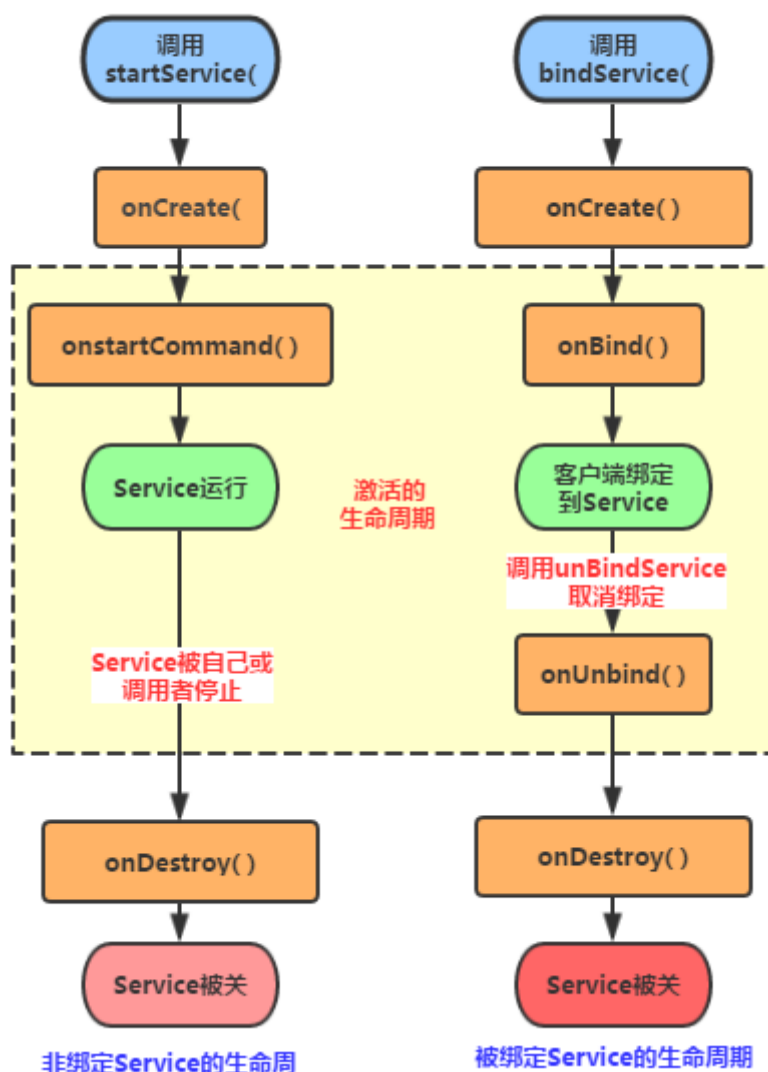
```

4. Service与Thread线程的区别

- 1 其实他们两者并没有太大的关系，不过有很多朋友经常把这两个混淆了！ **Thread**是线程，程序执行的最小单元，分配CPU的基本单位！ 而**Service**则是Android提供一个允许长时间驻驻后台的一个组件，最常见的 用法就是做轮询操作！或者想在后台做一些事情，比如后台下载更新！ 记得别把这两个概念混淆！

- Service的生命周期

Service的生命周期图



- 1 生命周期函数解析:
- 2 1) onCreate(): 当Service第一次被创建后立即回调该方法, 该方法在整个生命周期 中只会调用一次!
- 3 2) onDestroy(): 当Service被关闭时会回调该方法, 该方法只会回调一次!
- 4 3) onStartCommand(intent,flag,startId): 早期版本是onStart(intent,startId), 当客户端调用startService(Intent)方法时会回调, 可多次调用startService方法, 但不会再创建新的Service对象, 而是继续复用前面产生的Service对象, 但会继续回调 onStartCommand()方法!
- 5 IBinder onBind(intent): 该方法是Service都必须实现的方法, 该方法会返回一个 IBinder对象, app通过该对象与Service组件进行通信!
- 6 4) onUnbind(intent): 当该Service上绑定的所有客户端都断开时会回调该方法!

- service启动方式:

- 1 1) startService() 启动Service
- 2 2) bindService() 启动Service
- 3 PS: 还有一种, 就是启动Service后, 绑定Service!

- startService启动Service

- 1 ①首次启动会创建一个Service实例,依次调用onCreate()和onStartCommand()方法,此时Service 进入运行状态,如果再次调用startService启动Service,将不会再创建新的Service对象,系统会直接复用前面创建的Service对象,调用它的onStartCommand()方法!
- 2 ②但这样的Service与它的调用者无必然的联系,就是说当调用者结束了自己的生命周期,但是只要不调用stopService,那么Service还是会继续运行的!
- 3 ③无论启动了多少次Service,只需调用一次stopService即可停掉Service

- BindService启动Service

- 1 ①当首次使用bindService绑定一个Service时,系统会实例化一个Service实例,并调用其onCreate()和onBind()方法,然后调用者就可以通过IBinder和Service进行交互了,此后如果再次使用bindService绑定Service,系统不会创建新的Service实例,也不会再调用onBind()方法,只会直接把IBinder对象传递给其他后来增加的客户端!
- 2 ②如果我们解除与服务的绑定,只需调用unbindService(),此时onUnbind和onDestory方法将会被调用!这是一个客户端的情况,假如是多个客户端绑定同一个Service的话,情况如下 当一个客户完成和service之间的互动后,它调用 unbindService() 方法来解除绑定。当所有的客户端都和service解除绑定后,系统会销毁service。(除非service也被startService()方法开启)
- 3 ③另外,和上面那张情况不同,bindService模式下的Service是与调用者相互关联的,可以理解为"一条绳子上的蚂蚱",要死一起死,在bindService后,一旦调用者销毁,那么Service也立即终止!
- 4 通过BindService调用Service时调用的Context的bindService的解析
bindService(Intent service,ServiceConnection conn,int flags)
- 5 service:通过该intent指定要启动的Service
- 6 conn:ServiceConnection对象,用户监听访问者与Service间的连接情况,连接成功回调该对象中的onServiceConnected(ComponentName,IBinder)方法;如果Service所在的宿主由于异常终止或者其他原因终止,导致Service与访问者间断开 连接时调用onServiceDisconnected(ComponentName)方法,主动通过unBindService() 方法断开并不会调用上述方法!
- 7 flags:指定绑定定时是否自动创建Service(如果Service还未创建),参数可以是0(不自动创建),BIND_AUTO_CREATE(自动创建)

- StartService启动Service后bindService绑定

- 1 如果Service已经由某个客户端通过startService()启动,接下来由其他客户端 再调用bindService() 绑定到该Service后调用unbindService()解除绑定最后在 调用bindService()绑定到Service的话,此时所触发的生命周期方法如下:
- 2 onCreate() -> onStartCommand() -> onBind() -> onUnbind() -> onRebind()
- 3 PS:前提是:onUnbind()方法返回true!!! 这里或许部分读者有疑惑了,调用了unbindService后Service不是应该调用 onDestory()方法么!其实这是因为这个Service是由我们的startService来启动的,所以你调用onUnbind()方法取消绑定,Service也是不会终止的!
- 4 得出的结论:假如我们使用bindService来绑定一个启动的Service,注意是已经启动的Service!!! 系统只是将Service的内部IBinder对象传递给Activity,并不会将Service的生命周期与Activity绑定,因此调用unBindService()方法取消绑定时,Service也不会被销毁!

1.3 BroadcastReceiver 广播接收器

- 前言

- 1 为了方便Android系统各个应用程序及程序内部进行通信,Android系统引入了一套广播机制。各个应用程序可以对感兴趣的广播进行注册,当系统或者其他程序发出这条广播的时候,对发出的广播进行注册的程序便能够收到这条广播。为此,Android系统中有一套完整的API,允许程序只有发送和接受广播。

- 在Android系统中，主要有两种基本的广播类型：

- 标准广播 (Normal Broadcasts)

```
1  是一种完全异步执行的广播，在广播发出之后，所有的广播接收器会在同一时间接收到这条广播，广播无法被截断。  
2  发送方式：  
3  Intent intent=new Intent("com.example.dimple.BROADCAST_TEST");  
4  sendBroadcast(intent);
```

- 有序广播 (Ordered Broadcasts)

```
1  是一种同步执行的广播，在广播发出之后，优先级高的广播接收器可以优先接收到这条广播，并可以在优先级较低的广播接收器之前截断停止发送这条广播。  
2  发送方式：  
3  Intent intent=new Intent("com.example.dimple.BROADCAST_TEST");  
4  sendOrderBroadcast(intent, null);//第二个参数是与权限相关的字符串。
```

- 注册广播

```
1  在Android的广播接收机制中，如果需要接收广播，就需要创建广播接收器。而创建广播接收器的方法就是新建一个类（可以是单独新建类，也可以是内部类（public）） 继承自  
BroadcastReceiver  
2  
3  class myBroadcastReceiver extends BroadcastReceiver{  
4  
5      @Override  
6      public void onReceive(Context context, Intent intent) {  
7          //不要在广播里添加过多逻辑或者进行任何耗时操作,因为在广播中是不允许开辟  
          线程的，当onReceiver( )方法运行较长时间(超过10秒)还没有结束的话,那么程序会报错  
          (ANR)，广播更多的时候扮演的是一个打开其他组件的角色,比如启动Service,Notification  
          提示，Activity等!  
8          }  
9  }
```

- 动态注册和静态注册的区别

```
1  动态注册的广播接收器可以自由的控制注册和取消，有很大的灵活性。但是只能在程序启动之后才能收到广播，此外，不知道你注意到了没，广播接收器的注销是在onDestroy()方法中的。所以广播接收器的生命周期是和当前活动的生命周期一样。  
2  静态注册的广播不受程序是否启动的约束，当应用程序关闭之后，还是可以接收到广播。
```

- 两种方式注册广播：

- 动态注册

```
1 所谓动态注册是指在代码中注册。步骤如下：
2  - 实例化自定义的广播接收器。
3  - 创建IntentFilter实例。
4  - 调用IntentFilter实例的addAction()方法添加监听的广播类型。
5  - 最后调用Context的registerReceiver(BroadcastReceiver, IntentFilter)动态的注册广播。
6  PS:这里提醒一点，如果需要接收系统的广播（比如电量变化，网络变化等等），别忘记在AndroidManifest配置文件中加上权限。
7
8 另外，动态注册的广播在活动结束的时候需要取消注册：
9  @Override
10 protected void onDestroy() {
11     super.onDestroy();
12     unregisterReceiver(myBroadcastReceiver);
13 }
```

◦ 静态注册

```
1 <receiver
2   android:name="com.ttitt.core.broadcastreceiver.MyBRReceiver2">
3     <intent-filter>
4       <action
5         android:name="com.example.broadcasttest.MY_BROADCAST" />
6     </intent-filter>
7 </receiver>
```

1.4 ContentProvider(内容提供者)

- ContentProvider应用场景：

我们想在自己的应用中访问别的应用，或者说一些ContentProvider暴露给我们的一些数据，比如手机联系人，短信等！我们想对这些数据进行读取或者修改，这就需要用到ContentProvider了！

我们自己的应用，想把自己的一些数据暴露出来，给其他的应用进行读取或操作，我们也可以用到ContentProvider，另外我们可以选择要暴露的数据，就避免了我们隐私数据的泄露！

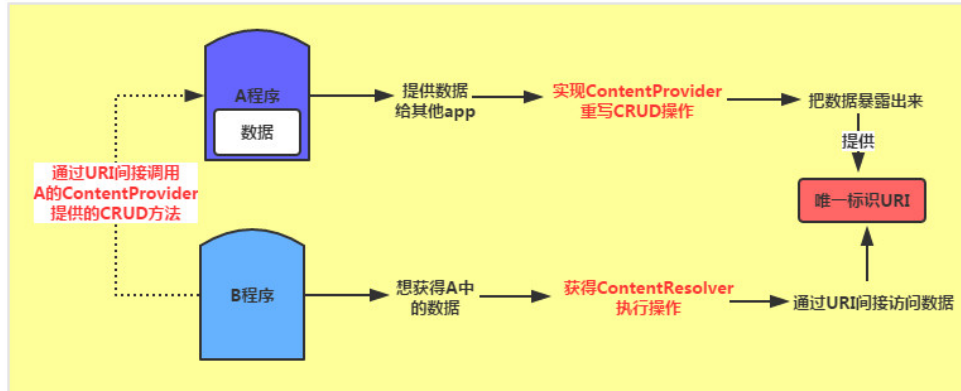
- ContentProvider概念讲解

ContentProvider(内容提供者)

ContentProvider的概述:

当我们想允许自己的应用的数据允许别的应用进行读取操作,我们可以让我们的App实现ContentProvider类,同时注册一个Uri,然后其他应用只要使用ContentResolver根据Uri就可以操作我们的app中的数据了!而数据不一定是数据库,也可能是文件,xml或者其他,但是SharedPreferences使用基于数据库模型的简单表格来提供其中的数据!

ContentProvider的执行原理



URI简介:

专业名词叫做:通用资源标识符,而你也可以类比为网页的域名,我们暂且就把他叫做资源定位符吧,就是定位资源所在路径的而在本节ContentProvider中,Uri灰常重要,我们分析一个简单的例子吧:
`content://com.jay.example.providers.myprovider/word/2`

分析:

`content`:协议头,这个是规定的,就像`http`,`ftp`等一样,规定的,而ContentProvider规定的是`content`开头的接着是`provider`所在的全限定类名
`word`:代表资源部分,如果想访问`word`所有资源,后面的2就不用写了,直接写`word`
`2`:访问的是`word`资源中`id`为2的记录

附加

当然,上面也说过数据不仅仅来自于数据库,有时也来源于文件,xml或者网络等其他存储方式,但是依旧可以使用上面这种URI定义方式:
比如:当表示的xml文件时:`~/word/detail`表示`word`节点下的`detail`结点
另外:URI还提供一个`parse()`方法将字符串转换为URI
eg:`Uri uri = Uri.parse("Content://~");`

- ContentProvider的URI

`content://com.example.transportationprovider/trains/122`

A

B

C

D

<http://blog.csdn.net/>

- 1 主要分三个部分: **scheme**, **authority** and **path**。scheme表示上图中的content://, **authority**表示B部分, **path**表示C和D部分。
- 2 A部分: 表示是一个Android内容URI, 说明由ContentProvider控制数据, 该部分是固定形式, 不可更改的。
- 3 B部分: 是URI的授权部分, 是唯一标识符, 用来定位ContentProvider。格式一般是自定义ContentProvider类的完全限定名称, 注册时需要用到, 如:
`com.example.transportationprovider`
- 4 C部分和D部分: 是每个ContentProvider内部的路径部分, C和D部分称为路径片段, C部分指向一个对象集合, 一般用表的名字, 如: `/trains`表示一个笔记集合; D部分指向特定的记录, 如: `/trains/122`表示id为122的单条记录, 如果没有指定D部分, 则返回全部记录。

- 使用系统提供的ContentProvider

- 1 打开模拟器的file explorer/data/data/com.android.providers.contacts/databases/contact2.db 导出后使用SQLite图形工具查看, 三个核心的表:raw_contact表, data表, mimetypes表

- 自定义ContentProvider

- 1 我们自己的应用, 想把自己的一些数据暴露出来, 给其他的应用进行读取或操作, 我们也可以用 到ContentProvider, 另外我们可以选择要暴露的数据, 就避免了隐私数据的泄露!

自定义ContentProvider流程解析

