# 第一章 Android部分

## 第1节 全面了解Activity

#Activity是什么？

　　我们都知道android中有四大组件（Activity 活动，Service 服务，Content Provider 内容提供者，BroadcastReceiver 广播接收器），Activity是我们用的最多也是最基本的组件，因为应用的所有操作都与用户相关，Activity 提供窗口来和用户进行交互。

　　官方文档这么说：

> An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View).

>

>

> 大概的意思（原谅我）：

>　　activity是独立平等的，用来处理用户操作。几乎所有的activity都是用来和用户交互的，所以activity类会创建了一个窗口，开发者可以通过setContentView(View)的接口把UI放到给窗口上。

　　Android中的activity全都归属于task管理 。task 是多个 activity 的集合，这些 activity 按照启动顺序排队存入一个栈（即“back stack”）。android默认会为每个App维持一个task来存放该app的所有activity，task的默认name为该app的packagename。

　　当然我们也可以在AndroidMainfest.xml中申明activity的taskAffinity属性来自定义task，但不建议使用，如果其他app也申明相同的task，它就有可能启动到你的activity，带来各种安全问题（比如拿到你的Intent）。

#Activity的内部调用过程

　　上面已经说了，系统通过堆栈来管理activity，当一个新的activity开始时，它被放置在堆栈的顶部和成为运行活动，以前的activity始终保持低于它在堆栈，而不会再次到达前台，直到新的活动退出。

　　还是上这张官网的activity\_lifecycle图：

　　![这里写图片描述](http://img.blog.csdn.net/20160425171711054)

- 首先打开一个新的activity实例的时候，系统会依次调用

> onCreate（） -> onStart() -> onResume() 然后开始running

　　running的时候被覆盖了（从它打开了新的activity或是被锁屏，但是它\*\*依然在前台\*\*运行， lost focus but is still visible），系统调用onPause();

>　该方法执行activity暂停，通常用于提交未保存的更改到持久化数据，停止动画和其他的东西。但这个activity还是完全活着（它保持所有的状态和成员信息，并保持连接到\*\*窗口管理器\*\*）

接下来它有三条出路

①用户返回到该activity就调用onResume()方法重新running

②用户回到桌面或是打开其他activity，就会调用onStop()进入停止状态（保留所有的状态和成员信息，\*\*对用户不可见\*\*）

③系统内存不足，拥有更高限权的应用需要内存，那么该activity的进程就可能会被系统回收。（回收onRause()和onStop()状态的activity进程）要想重新打开就必须重新创建一遍。

如果用户返回到onStop()状态的activity（又显示在前台了），系统会调用

> onRestart() -> onStart() -> onResume() 然后重新running

在activity结束（调用finish ()）或是被系统杀死之前会调用onDestroy()方法释放所有占用的资源。

> activity生命周期中三个嵌套的循环

- activity的完整生存期会在 onCreate() 调用和 onDestroy() 调用之间发生。

- activity的可见生存期会在 onStart() 调用和 onStop() 调用之间发生。系统会在activity的整个生存期内多次调用 onStart() 和onStop()， 因为activity可能会在显示和隐藏之间不断地来回切换。

- activity的前后台切换会在 onResume() 调用和 onPause() 之间发生。

因为这个状态可能会经常发生转换，为了避免切换迟缓引起的用户等待，这两个方法中的代码应该相当地轻量化。

## activity被回收的状态和信息保存和恢复过程

```

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

if(savedInstanceState!=null){ //判断是否有以前的保存状态信息

savedInstanceState.get("Key");

}

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

@Override

protected void onSaveInstanceState(Bundle outState) {

// TODO Auto-generated method stub

//可能被回收内存前保存状态和信息，

Bundle data = new Bundle();

data.putString("key", "last words before be kill");

outState.putAll(data);

super.onSaveInstanceState(outState);

}

@Override

protected void onRestoreInstanceState(Bundle savedInstanceState) {

// TODO Auto-generated method stub

if(savedInstanceState!=null){ //判断是否有以前的保存状态信息

savedInstanceState.get("Key");

}

super.onRestoreInstanceState(savedInstanceState);

}

}

```

> onSaveInstanceState方法

　　在activity　可能被回收之前　调用,用来保存自己的状态和信息，以便回收后重建时恢复数据（在onCreate()或onRestoreInstanceState()中恢复）。旋转屏幕重建activity会调用该方法，但其他情况在onRause()和onStop()状态的activity不一定会调用 ，下面是该方法的文档说明。

```

One example of when onPause and onStop is called and not this method is when a user navigates back from activity B to activity A: there is no need to call onSaveInstanceState on B because that particular instance will never be restored, so the system avoids calling it. An example when onPause is called and not onSaveInstanceState is when activity B is launched in front of activity A: the system may avoid calling onSaveInstanceState on activity A if it isn't killed during the lifetime of B since the state of the user interface of A will stay intact.

```

也就是说，系统灵活的来决定调不调用该方法，\*\*但是如果要调用就一定发生在onStop方法之前，但并不保证发生在onPause的前面还是后面。\*\*

> onRestoreInstanceState方法

　　这个方法在onStart 和 onPostCreate之间调用，在onCreate中也可以状态恢复，但有时候需要所有布局初始化完成后再恢复状态。

　　onPostCreate：一般不实现这个方法，当程序的代码开始运行时，它调用系统做最后的初始化工作。

#启动模式

##启动模式什么？

　　简单的说就是定义activity 实例与task 的关联方式。

##为什么要定义启动模式？

　　 为了实现一些默认启动（standard）模式之外的需求：

- 让某个 activity 启动一个新的 task （而不是被放入当前 task ）

- 让 activity 启动时只是调出已有的某个实例（而不是在 back stack 顶创建一个新的实例）

- 或者，你想在用户离开 task 时只保留根 activity，而 back stack 中的其它 activity 都要清空

##怎样定义启动模式？

　　定义启动模式的方法有两种：

### 使用 manifest 文件

　　在 manifest 文件中activity声明时，利用 activity 元素的 launchMode 属性来设定 activity 与 task 的关系。

```

<activity

．．．．．．

android:launchMode="standard"

>

．．．．．．．

</activity>

```

> 注意： 你用 launchMode 属性为 activity 设置的模式可以被启动 activity 的 intent 标志所覆盖。

####有哪些启动模式？

- "standard" （默认模式）

　　当通过这种模式来启动Activity时,　Android总会为目标 Activity创建一个新的实例,并将该Activity添加到当前Task栈中。这种方式不会启动新的Task,只是将新的 Activity添加到原有的Task中。

- "singleTop"

　　该模式和standard模式基本一致,但有一点不同:当将要被启动的Activity已经位于Task栈顶时,系统不会重新创建目标Activity实例,而是直接复用Task栈顶的Activity。

- "singleTask"

　　Activity在同一个Task内只有一个实例。

　　如果将要启动的Activity不存在,那么系统将会创建该实例,并将其加入Task栈顶；

　　如果将要启动的Activity已存在,且存在栈顶,直接复用Task栈顶的Activity。

　　如果Activity存在但是没有位于栈顶,那么此时系统会把位于该Activity上面的所有其他Activity全部移出Task,从而使得该目标Activity位于栈顶。

- "singleInstance"

　　无论从哪个Task中启动目标Activity,只会创建一个目标Activity实例且会用一个全新的Task栈来装载该Activity实例（全局单例）.

　　如果将要启动的Activity不存在,那么系统将会先创建一个全新的Task,再创建目标Activity实例并将该Activity实例放入此全新的Task中。

　　如果将要启动的Activity已存在,那么无论它位于哪个应用程序,哪个Task中;系统都会把该Activity所在的Task转到前台,从而使该Activity显示出来。

### 使用 Intent 标志

　　在要启动 activity 时，你可以在传给 startActivity() 的 intent 中包含相应标志，以修改 activity 与 task 的默认关系。

```

　　　　　Intent i = new Intent(this,ＮewActivity.class);

i.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);

startActivity(i);

```

####可以通过标志修改的默认模式有哪些？

- FLAG\_ACTIVITY\_NEW\_TASK

　　与"singleTask"模式相同，在新的 task 中启动 activity。如果要启动的 activity 已经运行于某 task 中，则那个 task 将调入前台。

- FLAG\_ACTIVITY\_SINGLE\_TOP

　　与 "singleTop"模式相同，如果要启动的 activity位于back stack 顶，系统不会重新创建目标Activity实例,而是直接复用Task栈顶的Activity。

- FLAG\_ACTIVITY\_CLEAR\_TOP

　　\*\*此种模式在launchMode中没有对应的属性值。\*\*如果要启动的 activity 已经在当前 task 中运行，则不再启动一个新的实例，且所有在其上面的 activity 将被销毁。

####　关于启动模式的一些建议

　　 一般不要改变 activity 和 task 默认的工作方式。 如果你确定有必要修改默认方式，请保持谨慎，并确保 activity 在启动和从其它 activity 返回时的可用性，多做测试和安全方面的工作。

# Intent Filter

　　android的3个核心组件——Activity、services、广播接收器——是通过intent传递消息的。intent消息用于在运行时绑定不同的组件。

　　在 Android 的 AndroidManifest.xml 配置文件中可以通过 intent-filter 节点为一个 Activity 指定其 Intent Filter，以便告诉系统该 Activity 可以响应什么类型的 Intent。

##intent-filter 的三大属性

###Action

　　一个 Intent Filter 可以包含多个 Action，Action 列表用于标示 Activity 所能接受的“动作”，它是一个用户自定义的字符串。

```

<intent-filter >

<action android:name="android.intent.action.MAIN" />

<action android:name="com.scu.amazing7Action" />

……

</intent-filter>

```

在代码中使用以下语句便可以启动该Intent 对象：

```

Intent i=new Intent();

i.setAction("com.scu.amazing7Action");

```

Action 列表中包含了“com.scu.amazing7Action”的 Activity 都将会匹配成功

###URL

　　在 intent-filter 节点中，通过 data节点匹配外部数据，也就是通过 URI 携带外部数据给目标组件。

```

<data android:mimeType="mimeType"

android:scheme="scheme"

android:host="host"

android:port="port"

android:path="path"/>

```

注意：只有data的所有的属性都匹配成功时 URI 数据匹配才会成功

###Category

　　为组件定义一个 类别列表，当 Intent 中包含这个类别列表的所有项目时才会匹配成功。

```

<intent-filter . . . >

<action android:name="code android.intent.action.MAIN" />

<category android:name="code　android.intent.category.LAUNCHER" />

</intent-filter>

```

##Activity 种 Intent Filter 的匹配过程

　　①加载所有的Intent Filter列表

　　②去掉action匹配失败的Intent Filter

　　③去掉url匹配失败的Intent Filter

　　④去掉Category匹配失败的Intent Filter

　　⑤判断剩下的Intent Filter数目是否为0。如果为0查找失败返回异常；如果大于0，就按优先级排序，返回最高优先级的Intent Filter

# 开发中Activity的一些问题

-

一般设置Activity为非公开的

```

<activity

．．．．．．

android:exported="false" />

```

注意：非公开的Activity不能设置intent-filter，以免被其他activity唤醒（如果拥有相同的intent-filter）。

- 不要指定activity的taskAffinity属性

- 不要设置activity的LaunchMode（保持默认）

　　注意Activity的intent最好也不要设定为FLAG\_ACTIVITY\_NEW\_TASK

- 在匿名内部类中使用this时加上activity类名（类名.this,不一定是当前activity）

- 设置activity全屏

　　在其 onCreate()方法中加入：

```

// 设置全屏模式

getWindow().setFlags(WindowManager.LayoutParams.FLAG\_FULLSCREEN, WindowManager.LayoutParams.FLAG\_FULLSCREEN);

// 去除标题栏

requestWindowFeature(Window.FEATURE\_NO\_TITLE);

## 第2节 Service全面总结

#什么是服务？

　　Service是一个应用程序组件，它能够在后台执行一些耗时较长的操作，并且不提供用户界面。服务能被其它应用程序的组件启动，即使用户切换到另外的应用时还能保持后台运行。此外，应用程序组件还能与服务绑定，并与服务进行交互，甚至能进行进程间通信（IPC）。 比如，服务可以处理网络传输、音乐播放、执行文件I/O、或者与content provider进行交互，所有这些都是后台进行的。

#Service 与 Thread 的区别

　　服务仅仅是一个组件，即使用户不再与你的应用程序发生交互，它仍然能在后台运行。因此，应该只在需要时才创建一个服务。

　　如果你需要在主线程之外执行一些工作，但仅当用户与你的应用程序交互时才会用到，那你应该创建一个新的线程而不是创建服务。 比如，如果你需要播放一些音乐，但只是当你的activity在运行时才需要播放，你可以在onCreate()中创建一个线程，在onStart()中开始运行，然后在onStop()中终止运行。还可以考虑使用AsyncTask或HandlerThread来取代传统的Thread类。

　　\*\*由于无法在不同的 Activity 中对同一 Thread 进行控制\*\*，这个时候就要考虑用服务实现。如果你使用了服务，它默认就运行于应用程序的主线程中。因此，如果服务执行密集计算或者阻塞操作，你仍然应该在服务中创建一个新的线程来完成（避免ANR）。

#服务的分类

##按运行分类

- 前台服务

　　前台服务是指那些经常会被用户关注的服务，因此内存过低时它不会成为被杀的对象。 前台服务必须提供一个状态栏通知，并会置于“正在进行的”（“Ongoing”）组之下。这意味着只有在服务被终止或从前台移除之后，此通知才能被解除。

　　例如，用服务来播放音乐的播放器就应该运行在前台，因为用户会清楚地知晓它的运行情况。 状态栏通知可能会标明当前播放的歌曲，并允许用户启动一个activity来与播放器进行交互。

　　要把你的服务请求为前台运行，可以调用startForeground()方法。此方法有两个参数：唯一标识通知的整数值、状态栏通知Notification对象。例如：

```

Notification notification = new Notification(R.drawable.icon, getText(R.string.ticker\_text),System.currentTimeMillis());

Intent notificationIntent = new Intent(this,ExampleActivity.class);

PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);

notification.setLatestEventInfo(this, getText(R.string.notification\_title),

getText(R.string.notification\_message), pendingIntent);

startForeground(ONGOING\_NOTIFICATION, notification);

```

　　要从前台移除服务，请调用stopForeground()方法，这个方法接受个布尔参数，表示是否同时移除状态栏通知。此方法不会终止服务。不过，如果服务在前台运行时被你终止了，那么通知也会同时被移除。

- 后台服务

##按使用分类

- 本地服务

　　用于应用程序内部，实现一些耗时任务，并不占用应用程序比如Activity所属线程，而是单开线程后台执行。

　　调用Context.startService()启动，调用Context.stopService()结束。在内部可以调用Service.stopSelf() 或 Service.stopSelfResult()来自己停止。

- 远程服务

　　用于Android系统内部的应用程序之间，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。可以定义接口并把接口暴露出来，以便其他应用进行操作。客户端建立到服务对象的连接，并通过那个连接来调用服务。调用Context.bindService()方法建立连接，并启动，以调用 Context.unbindService()关闭连接。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。

#Service生命周期

　　![这里写图片描述](http://img.blog.csdn.net/20160503165018575)

Service生命周期方法：

```

public class ExampleService extends Service {

int mStartMode; // 标识服务被杀死后的处理方式

IBinder mBinder; // 用于客户端绑定的接口

boolean mAllowRebind; // 标识是否使用onRebind

@Override

public void onCreate() {

// 服务正被创建

}

@Override

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

// 服务正在启动，由startService()调用引发

return mStartMode;

}

@Override

public IBinder onBind(Intent intent) {

// 客户端用bindService()绑定服务

return mBinder;

}

@Override

public boolean onUnbind(Intent intent) {

// 所有的客户端都用unbindService()解除了绑定

return mAllowRebind;

}

@Override

public void onRebind(Intent intent) {

// 某客户端正用bindService()绑定到服务,

// 而onUnbind()已经被调用过了

}

@Override

public void onDestroy() {

// 服务用不上了，将被销毁

}

}

```

> 请注意onStartCommand()方法必须返回一个整数。这个整数是描述系统在杀死服务之后应该如何继续运行。onStartCommand()的返回值必须是以下常量之一：

>START\_NOT\_STICKY

如果系统在onStartCommand()返回后杀死了服务，则不会重建服务了，除非还存在未发送的intent。 当服务不再是必需的，并且应用程序能够简单地重启那些未完成的工作时，这是避免服务运行的最安全的选项。

>START\_STICKY

如果系统在onStartCommand()返回后杀死了服务，则将重建服务并调用onStartCommand()，但不会再次送入上一个intent， 而是用null intent来调用onStartCommand() 。除非还有启动服务的intent未发送完，那么这些剩下的intent会继续发送。 这适用于媒体播放器（或类似服务），它们不执行命令，但需要一直运行并随时待命。

>START\_REDELIVER\_INTENT

如果系统在onStartCommand()返回后杀死了服务，则将重建服务并用上一个已送过的intent调用onStartCommand()。任何未发送完的intent也都会依次送入。这适用于那些需要立即恢复工作的活跃服务，比如下载文件。

　　服务的生命周期与activity的非常类似。不过，更重要的是你需密切关注服务的创建和销毁环节，因为后台运行的服务是不会引起用户注意的。

　　服务的生命周期——从创建到销毁——可以有两种路径：

- 一个started服务

　　这类服务由其它组件调用startService()来创建。然后保持运行，且必须通过调用stopSelf()自行终止。其它组件也可通过调用stopService() 终止这类服务。服务终止后，系统会把它销毁。

　　如果一个Service被startService 方法多次启动，那么onCreate方法只会调用一次，onStart将会被调用多次（对应调用startService的次数），并且系统只会创建Service的一个实例（因此你应该知道只需要一次stopService调用）。该Service将会一直在后台运行，而不管对应程序的Activity是否在运行，直到被调用stopService，或自身的stopSelf方法。当然如果系统资源不足，android系统也可能结束服务。

- 　一个bound服务

　　服务由其它组件（客户端）调用bindService()来创建。然后客户端通过一个IBinder接口与服务进行通信。客户端可以通过调用unbindService()来关闭联接。多个客户端可以绑定到同一个服务上，当所有的客户端都解除绑定后，系统会销毁服务。（服务不需要自行终止。）

　　如果一个Service被某个Activity 调用 Context.bindService 方法绑定启动，不管调用 bindService 调用几次，onCreate方法都只会调用一次，同时onStart方法始终不会被调用。当连接建立之后，Service将会一直运行，除非调用Context.unbindService 断开连接或者之前调用bindService 的 Context 不存在了（如Activity被finish的时候），系统将会自动停止Service，对应onDestroy将被调用。

![这里写图片描述](http://img.blog.csdn.net/20160503165622881)

　　这两条路径并不是完全隔离的。也就是说，你可以绑定到一个已经用startService()启动的服务上。例如，一个后台音乐服务可以通过调用startService()来启动，传入一个指明所需播放音乐的 Intent。 之后，用户也许需要用播放器进行一些控制，或者需要查看当前歌曲的信息，这时一个activity可以通过调用bindService()与此服务绑定。在类似这种情况下，stopService()或stopSelf()不会真的终止服务，除非所有的客户端都解除了绑定。

> 　　当在旋转手机屏幕的时候，当手机屏幕在“横”“竖”变换时，此时如果你的 Activity 如果会自动旋转的话，旋转其实是 Activity 的重新创建，因此旋转之前的使用 bindService 建立的连接便会断开（Context 不存在了）。

#在manifest中声明服务

　　无论是什么类型的服务都必须在manifest中申明，格式如下：

```

<manifest ... >

...

<application ... >

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

...

</application>

</manifest>

```

Service 元素的属性有：

> android:name　　-------------　　服务类名

>android:label　　--------------　　服务的名字，如果此项不设置，那么默认显示的服务名则为类名

>android:icon　　--------------　　服务的图标

>android:permission　　-------　　申明此服务的权限，这意味着只有提供了该权限的应用才能控制或连接此服务

>android:process　　----------　　表示该服务是否运行在另外一个进程，如果设置了此项，那么将会在包名后面加上这段字符串表示另一进程的名字

>android:enabled　　----------　　如果此项设置为 true，那么 Service 将会默认被系统启动，不设置默认此项为 false

>android:exported　　---------　　表示该服务是否能够被其他应用程序所控制或连接，不设置默认此项为 false

　　android:name是唯一必需的属性——它定义了服务的类名。与activity一样，服务可以定义intent过滤器，使得其它组件能用隐式intent来调用服务。如果你想让服务只能内部使用（其它应用程序无法调用），那么就不必（也不应该）提供任何intent过滤器。

　　此外，如果包含了android:exported属性并且设置为"false"， 就可以确保该服务是你应用程序的私有服务。即使服务提供了intent过滤器，本属性依然生效。

#startService 启动服务

　　从activity或其它应用程序组件中可以启动一个服务，调用startService()并传入一个Intent（指定所需启动的服务）即可。

```

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

startService(intent);

```

服务类：

```

public class MyService extends Service {

/\*\*

\* onBind 是 Service 的虚方法，因此我们不得不实现它。

\* 返回 null，表示客服端不能建立到此服务的连接。

\*/

@Override

public IBinder onBind(Intent intent) {

// TODO Auto-generated method stub

return null;

}

@Override

public void onCreate() {

super.onCreate();

}

@Override

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

//接受传递过来的intent的数据

return START\_STICKY;

};

@Override

public void onDestroy() {

super.onDestroy();

}

}

```

　　一个started服务必须自行管理生命周期。也就是说，系统不会终止或销毁这类服务，除非必须恢复系统内存并且服务返回后一直维持运行。 因此，服务必须通过调用stopSelf()自行终止，或者其它组件可通过调用stopService()来终止它。

#bindService 启动服务

　　当应用程序中的activity或其它组件需要与服务进行交互，或者应用程序的某些功能需要暴露给其它应用程序时，你应该创建一个bound服务，并通过进程间通信（IPC）来完成。

方法如下：

```

Intent intent=new Intent(this,BindService.class);

bindService(intent, ServiceConnection conn, int flags)

```

> 注意bindService是Context中的方法，当没有Context时传入即可。

在进行服务绑定的时，其flags有：

- Context.BIND\_AUTO\_CREATE

　　表示收到绑定请求的时候，如果服务尚未创建，则即刻创建，在系统内存不足需要先摧毁优先级组件来释放内存，且只有驻留该服务的进程成为被摧毁对象时，服务才被摧毁

- Context.BIND\_DEBUG\_UNBIND

　　通常用于调试场景中判断绑定的服务是否正确，但容易引起内存泄漏，因此非调试目的的时候不建议使用

- Context.BIND\_NOT\_FOREGROUND

　　表示系统将阻止驻留该服务的进程具有前台优先级，仅在后台运行。

服务类：

```

public class BindService extends Service {

// 实例化MyBinder得到mybinder对象；

private final MyBinder binder = new MyBinder();

/\*\*

\* 返回Binder对象。

\*/

@Override

public IBinder onBind(Intent intent) {

// TODO Auto-generated method stub

return binder;

}

/\*\*

\* 新建内部类MyBinder，继承自Binder(Binder实现IBinder接口),

\* MyBinder提供方法返回BindService实例。

\*/

　　public class MyBinder extends Binder{

public BindService getService(){

return BindService.this;

}

}

@Override

public boolean onUnbind(Intent intent) {

// TODO Auto-generated method stub

return super.onUnbind(intent);

}

}

```

启动服务的activity代码：

```

public class MainActivity extends Activity {

/\*\* 是否绑定 \*/

boolean mIsBound = false;

BindService mBoundService;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

doBindService();

}

/\*\*

\* 实例化ServiceConnection接口的实现类,用于监听服务的状态

\*/

private ServiceConnection conn = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

BindService mBoundService = ((BindService.MyBinder) service).getService();

}

@Override

public void onServiceDisconnected(ComponentName name) {

mBoundService = null;

}

};

/\*\* 绑定服务 \*/

public void doBindService() {

bindService(new Intent(MainActivity.this, BindService.class), conn,Context.BIND\_AUTO\_CREATE);

mIsBound = true;

}

/\*\* 解除绑定服务 \*/

public void doUnbindService() {

if (mIsBound) {

// Detach our existing connection.

unbindService(conn);

mIsBound = false;

}

}

@Override

protected void onDestroy() {

// TODO Auto-generated method stub

super.onDestroy();

doUnbindService();

}

}

```

> 注意在AndroidMainfest.xml中对Service进行显式声明

判断Service是否正在运行：

```

private boolean isServiceRunning() {

ActivityManager manager = (ActivityManager) getSystemService(ACTIVITY\_SERVICE);

　{

if ("com.example.demo.BindService".equals(service.service.getClassName())) 　　{

return true;

}

}

return false;

}

```

## 第3节 IntentService使用详解和实例介绍

#IntentService定义

　　IntentService继承与Service，用来处理异步请求。客户端可以通过startService(Intent)方法传递请求给IntentService。IntentService在onCreate()函数中通过HandlerThread单独开启一个线程来依次处理所有Intent请求对象所对应的任务。

　　这样以免事务处理阻塞主线程（ＡＮＲ）。执行完所一个Intent请求对象所对应的工作之后，如果没有新的Intent请求达到，则\*\*自动停止\*\*Service；否则执行下一个Intent请求所对应的任务。

　　IntentService在处理事务时，还是采用的Handler方式，创建一个名叫ServiceHandler的内部Handler，并把它直接绑定到HandlerThread所对应的子线程。 ServiceHandler把处理一个intent所对应的事务都封装到叫做\*\*onHandleIntent\*\*的虚函数；因此我们直接实现虚函数onHandleIntent，再在里面根据Intent的不同进行不同的事务处理就可以了。

另外，IntentService默认实现了Onbind（）方法，返回值为null。

使用IntentService需要实现的两个方法：

- 构造函数

　　IntentService的构造函数一定是\*\*参数为空\*\*的构造函数，然后再在其中调用super("name")这种形式的构造函数。因为Service的实例化是系统来完成的，而且系统是用参数为空的构造函数来实例化Service的

- 实现虚函数onHandleIntent

　　在里面根据Intent的不同进行不同的事务处理。

好处：处理异步请求的时候可以减少写代码的工作量，比较轻松地实现项目的需求。

#IntentService与Service的区别

　　Service不是独立的进程，也不是独立的线程，它是依赖于应用程序的主线程的，不建议在Service中编写耗时的逻辑和操作，否则会引起ANR。

　　IntentService 它创建了一个独立的工作线程来处理所有的通过onStartCommand()传递给服务的intents（把intent插入到工作队列中）。通过工作队列把intent逐个发送给onHandleIntent()。

　　不需要主动调用stopSelft()来结束服务。因为，在所有的intent被处理完后，系统会自动关闭服务。

　　 默认实现的onBind()返回null。

#IntentService实例介绍

　　首先是myIntentService.java

```

public class myIntentService extends IntentService {

//------------------必须实现-----------------------------

public myIntentService() {

super("myIntentService");

// 注意构造函数参数为空，这个字符串就是worker thread的名字

}

@Override

protected void onHandleIntent(Intent intent) {

//根据Intent的不同进行不同的事务处理

String taskName = intent.getExtras().getString("taskName");

switch (taskName) {

case "task1":

Log.i("myIntentService", "do task1");

break;

case "task2":

Log.i("myIntentService", "do task2");

break;

default:

break;

}

}

//--------------------用于打印生命周期--------------------

@Override

public void onCreate() {

Log.i("myIntentService", "onCreate");

super.onCreate();

}

@Override

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

Log.i("myIntentService", "onStartCommand");

return super.onStartCommand(intent, flags, startId);

}

@Override

public void onDestroy() {

Log.i("myIntentService", "onDestroy");

super.onDestroy();

}

}

```

然后记得在Manifest.xml中注册服务

```

<service android:name=".myIntentService">

<intent-filter >

<action android:name="cn.scu.finch"/>

</intent-filter>

</service>

```

最后在Activity中开启服务

```

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

// TODO Auto-generated method stub

super.onCreate(savedInstanceState);

//同一服务只会开启一个worker thread，在onHandleIntent函数里依次处理intent请求。

Intent i = new Intent("cn.scu.finch");

Bundle bundle = new Bundle();

bundle.putString("taskName", "task1");

i.putExtras(bundle);

startService(i);

Intent i2 = new Intent("cn.scu.finch");

Bundle bundle2 = new Bundle();

bundle2.putString("taskName", "task2");

i2.putExtras(bundle2);

startService(i2);

startService(i); //多次启动

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160513135411037)

　　IntentService在onCreate()函数中通过HandlerThread单独开启一个线程来依次处理所有Intent请求对象所对应的任务。

　　通过onStartCommand()传递给服务intent被\*\*依次\*\*插入到工作队列中。工作队列又把intent逐个发送给onHandleIntent()。

> 注意：

> 它只有一个工作线程，名字就是构造函数的那个字符串，也就是“myIntentService”，我们知道多次开启service，只会调用一次onCreate方法（创建一个工作线程），多次onStartCommand方法（用于传入intent通过工作队列再发给onHandleIntent函数做处理）。

## 第4节 Fragment 全解析

#1．概述

　　Fragment是Activity中用户界面的一个行为或者是一部分。主要是支持在大屏幕上动态和更为灵活的去组合或是交换UI组件，通过将activity的布局分割成若干个fragment，可以在运行时编辑activity的呈现，并且那些变化会被保存在由activity管理的后台栈里面。

　　\*\*Fragment必须总是被嵌入到一个activity之中\*\*，并且fragment的生命周期直接受其宿主activity的生命周期的影响。你可以认为fragment是activity的一个模块零件，它有自己的生命周期，接收它自己的输入事件，并且可以在activity运行时添加或者删除。

　　应该将每一个fragment设计为模块化的和可复用化的activity组件。也就是说，你可以在多个activity中引用同一个fragment，因为fragment定义了它自己的布局，并且使用它本身生命周期回调的行为。

#２．Fragment的生命周期

先看fragment生命周期图：

![这里写图片描述](http://img.blog.csdn.net/20160429134558410)

　　fragment所生存的activity生命周期直接影响着fragment的生命周期，由此针对activity的每一个生命周期回调都会引发一个fragment类似的回调。例如，当activity接收到onPause()时，这个activity之中的每个fragment都会接收到onPause()。

　　[这有Activity的详细说明](http://blog.csdn.net/amazing7/article/details/51244219)

　　Fragment有一些额外的生命周期回调方法（创建和销毁fragment界面）．

- onAttach()

　　当fragment被绑定到activity时调用（Activity会被传入）。

- onCreateView()

　　将本身的布局构建到activity中去（fragment作为activity界面的一部分）

- onActivityCreated()

　　当activity的onCreate()函数返回时被调用。

- onDestroyView()

　　当与fragment关联的视图体系正被移除时被调用。

- onDetach()

　　当fragment正与activity解除关联时被调用。

当activity接收到它的onCreate()回调时，activity之中的fragment接收到onActivityCreated()回调。

　　一旦activity处于resumed状态，则可以在activity中自由的添加或者移除fragment。因此，只\*\*有当activity处于resumed状态时\*\*，fragment的生命周期才可以独立变化。

fragment会在　activity离开恢复状态时　再一次被activity推入它的生命周期中。

\*\*管理fragment生命周期\*\*与管理activity生命周期很相像。像activity一样，fragment也有三种状态：

- Resumed

　　fragment在运行中的activity可见。

- Paused

　　另一个activity处于前台且得到焦点，但是这个fragment所在的activity仍然可见（前台activity部分透明，或者没有覆盖全屏）。

- Stopped

　　fragment不可见。要么宿主activity已经停止，要么fragment已经从activity上移除，但已被添加到后台栈中。一个停止的fragment仍然活着（所有状态和成员信息仍然由系统保留着）。但是，它对用户来讲已经不再可见，并且如果activity被杀掉，它也将被杀掉。

　　如果activity的进程被杀掉了，在activity被重新创建时，你需要恢复fragment状态。可以执行fragment的onSaveInstanceState()来保存状态（注意在fragment是在onCreate()，onCreateView()，或onActvityCreate()中进行恢复）。

　　在生命周期方面,activity与fragment之间一个\*\*很重要的不同\*\*，就是在各自的后台栈中是如何存储的。

　　当activity停止时，\*\*默认\*\*情况下activity被安置在由系统管理的activity后台栈中；

　　fragment仅当在一个事务被移除时，通过显式调用addToBackStack()请求保存的实例，该fragment才被置于由宿主activity管理的后台栈。

\*\*要创建一个fragment\*\*，必须创建一个fragment的子类。一般情况下，我们至少需要实现以下几个fragment生命周期方法：

> onCreate()

　　在创建fragment时系统会调用此方法。在实现代码中，你可以初始化想要在fragment中保持的那些必要组件，当fragment处于暂停或者停止状态之后可重新启用它们。

>onCreateView()

　　在第一次为fragment绘制用户界面时系统会调用此方法。为fragment绘制用户界面，这个函数必须要返回所绘出的fragment的根View。如果fragment没有用户界面可以返回空。

```

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container,Bundle savedInstanceState) {

// Inflate the layout for this fragment

return inflater.inflate(R.layout.example\_fragment, container, false);

}

```

inflate()函数需要以下三个参数：

①要inflate的布局的资源ID。

②被inflate的布局的父ViewGroup。

③一个布尔值，表明在inflate期间被infalte的布局是否应该附上ViewGroup（第二个参数container）。（在这个例子中传入的是false，因为系统已经将被inflate的布局插入到容器中（container）——传入true会在最终的布局里创建一个多余的ViewGroup。）

> onPause()

　　系统回调用该函数作为用户离开fragment的第一个预兆（尽管这并不总意味着fragment被销毁）。在当前用户会话结束之前，通常要在这里提交任何应该持久化的变化（因为用户可能不再返回）。

#3.将fragment添加到activity之中

　　可以通过在activity布局文件中声明fragment，用fragment标签把fragment插入到activity的布局中，或者是用应用程序源码将它添加到一个存在的ViewGroup中。

　　但fragment并不是一个定要作为activity布局的一部分，fragment也可以为activity隐身工作。

##3.1在activity的布局文件里声明fragment

　　可以像为view一样为fragment指定布局属性。例如：

```

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="horizontal"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<fragment android:name="com.example.test.FragmentOne"

android:id="@+id/fo"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" />

</LinearLayout>

```

　　fragment标签中的android:name 属性指定了布局中实例化的Fragment类。

　　当系统创建activity布局时，它实例化了布局文件中指定的每一个fragment，并为它们调用onCreateView()函数，以获取每一个fragment的布局。系统直接在<fragment>元素的位置插入fragment返回的View。

　　注意：每个fragment都需要一个唯一的标识，如果重启activity，系统可用来恢复fragment（并且可用来捕捉fragment的事务处理，例如移除）。为fragment提供ID有三种方法：

-

用android:id属性提供一个唯一的标识。

- 用android:tag属性提供一个唯一的字符串。

- 如果上述两个属性都没有，系统会使用其容器视图（view）的ID。

##3.2通过编码将fragment添加到已存在的ViewGroup中

　　在activity运行的任何时候，你都可以将fragment添加到activity布局中。

　　要管理activity中的fragment，可以使用FragmentManager。可以通过在activity中调用getFragmentManager()获得。使用FragmentManager 可以做如下事情，包括：

- 使用findFragmentById()（用于在activity布局中提供有界面的fragment）或者findFragmentByTag()获取activity中存在的fragment（用于有界面或者没有界面的fragment）。

- 使用popBackStack()（模仿用户的BACK命令）从后台栈弹出fragment。

- 使用addOnBackStackChangedListener()注册一个监听后台栈变化的监听器。

在Android中，对Fragment的事务操作都是通过FragmentTransaction来执行。操作大致可以分为两类：

- 显示：add() replace() show() attach()

- 隐藏：remove() hide() detach()

> 说明：

　　调用show() & hide()方法时，Fragment的生命周期方法并不会被执行，仅仅是Fragment的View被显示或者​隐藏。

>　　执行replace()时（至少两个Fragment），会执行第二个Fragment的onAttach()方法、执行第一个Fragment的onPause()-onDetach()方法，同时containerView会detach第一个Fragment的View。

>　　add()方法执行onAttach()-onResume()的生命周期，相对的remove()就是执行完成剩下的onPause()-onDetach()周期。

可以像下面这样从Activity中取得FragmentTransaction的实例：

```

FragmentManager fragmentManager = getFragmentManager()

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

```

可以用add()函数添加fragment，并指定要添加的fragment以及要将其插入到哪个视图（view）之中（注意commit事务）：

```

ExampleFragment fragment = new ExampleFragment();

fragmentTransaction.add(R.id.fragment\_container, fragment);

fragmentTransaction.commit();

```

##3.3添加没有界面的fragment

　　也可以使用fragment为activity提供后台动作，却不呈现多余的用户界面。

　　想要添加没有界面的fragment ，可以使用add(Fragment, String)（为fragment提供一个唯一的字符串“tag”，而不是视图（view）ID）。这样添加了fragment，但是，因为还没有关联到activity布局中的视图（view） ，收不到onCreateView()的调用。所以不需要实现这个方法。

　　对于无界面fragment，字符串标签是\*\*唯一识别\*\*它的方法。如果之后想从activity中取到fragment，需要使用findFragmentByTag()。

#4.fragment事务后台栈

　　在调用commit()之前，可以将事务添加到fragment事务后台栈中（通过调用addToBackStatck()）。这个后台栈由activity管理，并且允许用户通过按BACK键回退到前一个fragment状态。

　　下面的代码中一个fragment代替另一个fragment，并且将之前的fragment状态保留在后台栈中：

```

Fragment newFragment = new ExampleFragment();

FragmentTransaction transaction = getFragmentManager().beginTransaction();

transaction.replace(R.id.fragment\_container, newFragment);

transaction.addToBackStack(null);

transaction.commit();

```

> 注意：

>

>　　 如果添加多个变更事务（例如另一个add()或者remove()）并调用addToBackStack()，那么在调用commit()之前的所有应用的变更被作为一个单独的事务添加到后台栈中，并且BACK键可以将它们一起回退。

>

> 　　当移除一个fragment时，如果调用了addToBackStack()，那么之后fragment会被停止，如果用户回退，它将被恢复过来。

>

>　　调用commit()并不立刻执行事务，相反，而是采取预约方式，一旦activity的界面线程（主线程）准备好便可运行起来。然而，如果有必要的话，你可以从界面线程调用executePendingTransations()立即执行由commit()提交的事务。

>

> 　　只能在activity保存状态（当用户离开activity时）之前用commit()提交事务。如果你尝试在那时之后提交，会抛出一个异常。这是因为如果activity需要被恢复，提交后的状态会被丢失。对于这类丢失提交的情况，可使用commitAllowingStateLoss()

#５.与Activity交互

- Activity中已经有了该Fragment的引用，直接通过该引用进行交互。

-如果没引用可以通过调用fragment的函数findFragmentById()或者findFragmentByTag()，从FragmentManager中获取Fragment的索引，例如：

```

ExampleFragment fragment = (ExampleFragment) getFragmentManager().findFragmentById(R.id.example\_fragment);

```

- 在Fragment中可以通过getActivity得到当前绑定的Activity的实例。

- 创建activity事件回调函数，在fragment内部定义一个回调接口，宿主activity来实现它。

## 第5节 ContentProvider实例详解

#1.ContentProvider是什么？

　　ContentProvider（内容提供者）是Android的四大组件之一，管理android以结构化方式存放的数据，以相对安全的方式封装数据（表）并且提供简易的处理机制和统一的访问接口供\*\*其他程序\*\*调用。

　　Android的数据存储方式总共有五种，分别是：Shared Preferences、网络存储、文件存储、外储存储、SQLite。但一般这些存储都只是在单独的一个应用程序之中达到一个数据的共享，有时候我们需要操作其他应用程序的一些数据，就会用到ContentProvider。而且Android为常见的一些数据提供了默认的ContentProvider（包括音频、视频、图片和通讯录等）。

　　但注意ContentProvider它也只是一个中间人，真正操作的数据源可能是数据库，也可以是文件、xml或网络等其他存储方式。

#２.URL

　　URL（统一资源标识符）代表要操作的数据，可以用来标识每个ContentProvider，这样你就可以通过指定的URI找到想要的ContentProvider,从中获取或修改数据。

　　在Android中URI的格式如下图所示：

![这里写图片描述](http://img.blog.csdn.net/20160505154322045)

- Ａ

　　　schema，已经由Android所规定为：content://．

- Ｂ

　　　主机名（Authority），是URI的授权部分，是唯一标识符，用来定位ContentProvider。

> Ｃ部分和D部分：是每个ContentProvider内部的路径部分

- Ｃ

　　　指向一个对象集合，一般用表的名字，如果没有指定D部分，则返回全部记录。

- Ｄ

　　　指向特定的记录，这里表示操作user表id为7的记录。如果要操作user表中id为7的记录的name字段， D部分变为 \*\*/7/name\*\*即可。

> URI模式匹配通配符

>

> \*：匹配的任意长度的任何有效字符的字符串。

>

> ＃：匹配的任意长度的数字字符的字符串。

>

> 如：

>

> content://com.example.app.provider/\*

> 匹配provider的任何内容url

>

> content://com.example.app.provider/table3/#

> 匹配table3的所有行

##2.１MIME

　　MIME是指定某个扩展名的文件用一种应用程序来打开，就像你用浏览器查看PDF格式的文件，浏览器会选择合适的应用来打开一样。Android中的工作方式跟HTTP类似，ContentProvider会根据URI来返回MIME类型，ContentProvider会返回一个包含两部分的字符串。MIME类型一般包含两部分，如：

> text/html

text/css

text/xml

application/pdf

　　分为类型和子类型，Android遵循类似的约定来定义MIME类型，每个内容类型的Android MIME类型有两种形式：多条记录（集合）和单条记录。

　　集合记录：

```

vnd.android.cursor.dir/自定义

```

　　单条记录：

```

vnd.android.cursor.item/自定义

```

　　vnd表示这些类型和子类型具有非标准的、供应商特定的形式。Android中类型已经固定好了，不能更改，只能区别是集合还是单条具体记录，子类型可以按照格式自己填写。

　　在使用Intent时，会用到MIME，根据Mimetype打开符合条件的活动。

　　下面分别介绍Android系统提供了两个用于操作Uri的工具类：ContentUris和UriMatcher。

##2.２ ContentUris

　　ContetnUris包含一个便利的函数withAppendedId()来向URI追加一个id。

```

Uri uri = Uri.parse("content://cn.scu.myprovider/user")

Uri resultUri = ContentUris.withAppendedId(uri, 7);

//生成后的Uri为：content://cn.scu.myprovider/user/7

```

　　同时提供parseId(uri)方法用于从URL中获取ID:

```

Uri uri = Uri.parse("content://cn.scu.myprovider/user/7")

long personid = ContentUris.parseId(uri);

//获取的结果为:7

```

##2.３UriMatcher

　　UriMatcher本质上是一个文本过滤器，用在contentProvider中帮助我们过滤，分辨出查询者想要查询哪个数据表。

　　举例说明：

- 第一步，初始化：

```

UriMatcher matcher = new UriMatcher(UriMatcher.NO\_MATCH);

//常量UriMatcher.NO\_MATCH表示不匹配任何路径的返回码

```

- 第二步，注册需要的Uri：

```

//USER 和 USER\_ID是两个int型数据

matcher.addURI("cn.scu.myprovider", "user", USER);

matcher.addURI("cn.scu.myprovider", "user/#",USER\_ID);

//如果match()方法匹配content://cn.scu.myprovider/user路径，返回匹配码为USER

```

- 第三部，与已经注册的Uri进行匹配:

```

/\*

\* 如果操作集合，则必须以vnd.android.cursor.dir开头

\* 如果操作非集合，则必须以vnd.android.cursor.item开头

\* \*/

@Override

public String getType(Uri uri) {

Uri uri = Uri.parse("content://" + "cn.scu.myprovider" + "/user");

switch(matcher.match(uri)){

case USER:

return "vnd.android.cursor.dir/user";

case USER\_ID:

return "vnd.android.cursor.item/user";

}

}

```

#3.ContentProvider的主要方法

> public boolean onCreate()

　　ContentProvider创建后　或　打开系统后其它应用第一次访问该ContentProvider时调用。

> public Uri insert(Uri uri, ContentValues values)

　　外部应用向ContentProvider中添加数据。

> public int delete(Uri uri, String selection, String[] selectionArgs)

　　外部应用从ContentProvider删除数据。

> public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs)：

　　外部应用更新ContentProvider中的数据。

> public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)

　　供外部应用从ContentProvider中获取数据。

> public String getType(Uri uri)

　　该方法用于返回当前Url所代表数据的MIME类型。

#４.ContentResolver

　　ContentResolver通过URI来查询ContentProvider中提供的数据。除了URI以 外，还必须知道需要获取的数据段的名称，以及此数据段的数据类型。如果你需要获取一个特定的记录，你就必须知道当前记录的ID，也就是URI中D部分。

　　ContentResolver 类提供了与ContentProvider类相同签名的四个方法：

> public Uri insert(Uri uri, ContentValues values)　//添加

> public int delete(Uri uri, String selection, String[] selectionArgs)　//删除

>

> public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs)　//更新

>

> public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)//获取

实例代码：

```

ContentResolver resolver = getContentResolver();

Uri uri = Uri.parse("content://cn.scu.myprovider/user");

//添加一条记录

ContentValues values = new ContentValues();

values.put("name", "fanrunqi");

values.put("age", 24);

resolver.insert(uri, values);

//获取user表中所有记录

Cursor cursor = resolver.query(uri, null, null, null, "userid desc");

while(cursor.moveToNext()){

//操作

}

//把id为1的记录的name字段值更改新为finch

ContentValues updateValues = new ContentValues();

updateValues.put("name", "finch");

Uri updateIdUri = ContentUris.withAppendedId(uri, 1);

resolver.update(updateIdUri, updateValues, null, null);

//删除id为2的记录

Uri deleteIdUri = ContentUris.withAppendedId(uri, 2);

resolver.delete(deleteIdUri, null, null);

```

#5.ContentObserver

　　　 ContentObserver(内容观察者)，目的是观察特定Uri引起的数据库的变化，继而做一些相应的处理，它类似于数据库技术中的触发器(Trigger)，当ContentObserver所观察的Uri发生变化时，便会触发它.

下面是使用内容观察者监听短信的例子：

```

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

//注册观察者Observser

this.getContentResolver().registerContentObserver(Uri.parse("content://sms"),true,new SMSObserver(new Handler()));

}

private final class SMSObserver extends ContentObserver {

public SMSObserver(Handler handler) {

super(handler);

}

@Override

public void onChange(boolean selfChange) {

Cursor cursor = MainActivity.this.getContentResolver().query(

Uri.parse("content://sms/inbox"), null, null, null, null);

while (cursor.moveToNext()) {

StringBuilder sb = new StringBuilder();

sb.append("address=").append(

cursor.getString(cursor.getColumnIndex("address")));

sb.append(";subject=").append(

cursor.getString(cursor.getColumnIndex("subject")));

sb.append(";body=").append(

cursor.getString(cursor.getColumnIndex("body")));

sb.append(";time=").append(

cursor.getLong(cursor.getColumnIndex("date")));

System.out.println("--------has Receivered SMS::" + sb.toString());

}

}

}

}

```

同时可以在ContentProvider发生数据变化时调用

getContentResolver().notifyChange(uri, null)来通知注册在此URI上的访问者。

```

public class UserContentProvider extends ContentProvider {

public Uri insert(Uri uri, ContentValues values) {

db.insert("user", "userid", values);

getContext().getContentResolver().notifyChange(uri, null);

}

}

```

#6.实例说明

　　数据源是SQLite, 用ContentResolver操作ContentProvider。

![这里写图片描述](http://img.blog.csdn.net/20160505195143099)

Constant.java（储存一些常量）

```

public class Constant {

public static final String TABLE\_NAME = "user";

public static final String COLUMN\_ID = "\_id";

public static final String COLUMN\_NAME = "name";

public static final String AUTOHORITY = "cn.scu.myprovider";

public static final int ITEM = 1;

public static final int ITEM\_ID = 2;

public static final String CONTENT\_TYPE = "vnd.android.cursor.dir/user";

public static final String CONTENT\_ITEM\_TYPE = "vnd.android.cursor.item/user";

public static final Uri CONTENT\_URI = Uri.parse("content://" + AUTOHORITY + "/user");

}

```

DBHelper.java(操作数据库)

```

public class DBHelper extends SQLiteOpenHelper {

private static final String DATABASE\_NAME = "finch.db";

private static final int DATABASE\_VERSION = 1;

public DBHelper(Context context) {

super(context, DATABASE\_NAME, null, DATABASE\_VERSION);

}

@Override

public void onCreate(SQLiteDatabase db) throws SQLException {

//创建表格

db.execSQL("CREATE TABLE IF NOT EXISTS "+ Constant.TABLE\_NAME + "("+ Constant.COLUMN\_ID +" INTEGER PRIMARY KEY AUTOINCREMENT," + Constant.COLUMN\_NAME +" VARCHAR NOT NULL);");

}

@Override

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) throws SQLException {

//删除并创建表格

db.execSQL("DROP TABLE IF EXISTS "+ Constant.TABLE\_NAME+";");

onCreate(db);

}

}

```

　MyProvider.java(自定义的ContentProvider)

```

public class MyProvider extends ContentProvider {

DBHelper mDbHelper = null;

SQLiteDatabase db = null;

private static final UriMatcher mMatcher;

static{

mMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

mMatcher.addURI(Constant.AUTOHORITY,Constant.TABLE\_NAME, Constant.ITEM);

mMatcher.addURI(Constant.AUTOHORITY, Constant.TABLE\_NAME+"/#", Constant.ITEM\_ID);

}

@Override

public String getType(Uri uri) {

switch (mMatcher.match(uri)) {

case Constant.ITEM:

return Constant.CONTENT\_TYPE;

case Constant.ITEM\_ID:

return Constant.CONTENT\_ITEM\_TYPE;

default:

throw new IllegalArgumentException("Unknown URI"+uri);

}

}

@Override

public Uri insert(Uri uri, ContentValues values) {

// TODO Auto-generated method stub

long rowId;

if(mMatcher.match(uri)!=Constant.ITEM){

throw new IllegalArgumentException("Unknown URI"+uri);

}

rowId = db.insert(Constant.TABLE\_NAME,null,values);

if(rowId>0){

Uri noteUri=ContentUris.withAppendedId(Constant.CONTENT\_URI, rowId);

getContext().getContentResolver().notifyChange(noteUri, null);

return noteUri;

}

throw new SQLException("Failed to insert row into " + uri);

}

@Override

public boolean onCreate() {

// TODO Auto-generated method stub

mDbHelper = new DBHelper(getContext());

db = mDbHelper.getReadableDatabase();

return true;

}

@Override

public Cursor query(Uri uri, String[] projection, String selection,

String[] selectionArgs, String sortOrder) {

// TODO Auto-generated method stub

Cursor c = null;

switch (mMatcher.match(uri)) {

case Constant.ITEM:

c = db.query(Constant.TABLE\_NAME, projection, selection, selectionArgs, null, null, sortOrder);

break;

case Constant.ITEM\_ID:

c = db.query(Constant.TABLE\_NAME, projection,Constant.COLUMN\_ID + "="+uri.getLastPathSegment(), selectionArgs, null, null, sortOrder);

break;

default:

throw new IllegalArgumentException("Unknown URI"+uri);

}

c.setNotificationUri(getContext().getContentResolver(), uri);

return c;

}

@Override

public int update(Uri uri, ContentValues values, String selection,

String[] selectionArgs) {

// TODO Auto-generated method stub

return 0;

}

@Override

public int delete(Uri uri, String selection, String[] selectionArgs) {

// TODO Auto-generated method stub

return 0;

}

}

```

MainActivity.java(ContentResolver操作)

```

public class MainActivity extends Activity {

private ContentResolver mContentResolver = null;

private Cursor cursor = null;

@Override

protected void onCreate(Bundle savedInstanceState) {

// TODO Auto-generated method stub

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

TextView tv = (TextView) findViewById(R.id.tv);

mContentResolver = getContentResolver();

tv.setText("添加初始数据 ");

for (int i = 0; i < 10; i++) {

ContentValues values = new ContentValues();

values.put(Constant.COLUMN\_NAME, "fanrunqi"+i);

mContentResolver.insert(Constant.CONTENT\_URI, values);

}

tv.setText("查询数据 ");

cursor = mContentResolver.query(Constant.CONTENT\_URI, new String[]{Constant.COLUMN\_ID,Constant.COLUMN\_NAME}, null, null, null);

if (cursor.moveToFirst()) {

String s = cursor.getString(cursor.getColumnIndex(Constant.COLUMN\_NAME));

tv.setText("第一个数据： "+s);

}

}

}

```

最后在manifest申明

```

<provider android:name="MyProvider" android:authorities="cn.scu.myprovider" />

```

[本文中代码下载](http://download.csdn.net/detail/amazing7/9511234)

## 第6节 BroadcastReceiver使用总结

# BroadcastReceiver的定义

　　广播是一种广泛运用的在应用程序之间传输信息的机制，主要用来监听系统或者应用发出的广播信息，然后根据广播信息作为相应的逻辑处理，也可以用来传输少量、频率低的数据。

　　在实现开机启动服务和网络状态改变、电量变化、短信和来电时通过接收系统的广播让应用程序作出相应的处理。

　　BroadcastReceiver 自身并不实现图形用户界面，但是当它收到某个通知后， BroadcastReceiver 可以通过启动 Service 、启动 Activity 或是 NotificationMananger 提醒用户。

#BroadcastReceiver使用注意

　　当系统或应用发出广播时，将会扫描系统中的所有广播接收者，通过action匹配将广播发送给相应的接收者，接收者收到广播后将会产生一个广播接收者的实例，执行其中的onReceiver()这个方法；特别需要注意的是这个实例的生命周期只有10秒，如果10秒内没执行结束onReceiver()，系统将会报错。

　　在onReceiver()执行完毕之后，该实例将会被销毁，所以不要在onReceiver()中执行耗时操作，也不要在里面创建子线程处理业务（因为可能子线程没处理完，接收者就被回收了，那么子线程也会跟着被回收掉）；正确的处理方法就是通过in调用activity或者service处理业务。

#BroadcastReceiver的注册

　　BroadcastReceiver的注册方式有且只有两种，一种是静态注册（推荐使用），另外一种是动态注册，广播接收者在注册后就开始监听系统或者应用之间发送的广播消息。

\*\*接收短信广播示例\*\*：

定义自己的BroadcastReceiver 类

```

public class MyBroadcastReceiver extends BroadcastReceiver {

// action 名称

String SMS\_RECEIVED = "android.provider.Telephony.SMS\_RECEIVED" ;

public void onReceive(Context context, Intent intent) {

if (intent.getAction().equals( SMS\_RECEIVED )) {

// 一个receiver可以接收多个action的，即可以有多个intent-filter，需要在onReceive里面对intent.getAction(action name)进行判断。

}

}

}

```

##静态方式

　　在AndroidManifest.xml的application里面定义receiver并设置要接收的action。

```

< receiver android:name = ".MyBroadcastReceiver" >

< intent-filter android:priority = "777" >

<action android:name = "android.provider.Telephony.SMS\_RECEIVED" />

</ intent-filter >

</ receiver >

```

　　这里的priority取值是　-1000到1000，值越大优先级越高，同时注意加上系统接收短信的限权。

```

< uses-permission android:name ="android.permission.RECEIVE\_SMS" />

```

　　静态注册的广播接收者是一个常驻在系统中的全局监听器，当你在应用中配置了一个静态的BroadcastReceiver，安装了应用后而无论应用是否处于运行状态，广播接收者都是已经常驻在系统中了。同时应用里的所有receiver都在清单文件里面，方便查看。

　　要销毁掉静态注册的广播接收者，可以通过调用PackageManager将Receiver禁用。

##动态方式

　　在Activity中声明BroadcastReceiver的扩展对象，在onResume中注册，onPause中卸载.

```

public class MainActivity extends Activity {

MyBroadcastReceiver receiver;

@Override

protected void onResume() {

// 动态注册广播 (代码执行到这才会开始监听广播消息，并对广播消息作为相应的处理)

receiver = new MyBroadcastReceiver();

IntentFilter intentFilter = new IntentFilter( "android.provider.Telephony.SMS\_RECEIVED" );

registerReceiver( receiver , intentFilter);

super.onResume();

}

@Override

protected void onPause() {

// 撤销注册 (撤销注册后广播接收者将不会再监听系统的广播消息)

unregisterReceiver(receiver);

super.onPause();

}

}

```

##静态注册和动态注册的区别

　　1、静态注册的广播接收者一经安装就常驻在系统之中，不需要重新启动唤醒接收者；

　　动态注册的广播接收者随着应用的生命周期，由registerReceiver开始监听，由unregisterReceiver撤销监听，如果应用退出后，没有撤销已经注册的接收者应用应用将会报错。

　　2、当广播接收者通过intent启动一个activity或者service时，如果intent中无法匹配到相应的组件。

　　动态注册的广播接收者将会导致应用报错

　　而静态注册的广播接收者将不会有任何报错，因为自从应用安装完成后，广播接收者跟应用已经脱离了关系。

#发送BroadcastReceiver

发送广播主要有两种类型：

##普通广播

应用在需要通知各个广播接收者的情况下使用，如 开机启动

使用方法：sendBroadcast()

```

Intent intent = new Intent("android.provider.Telephony.SMS\_RECEIVED");

//通过intent传递少量数据

intent.putExtra("data", "finch");

// 发送普通广播

sendBroadcast(Intent);

```

　　普通广播是完全异步的，可以在同一时刻（逻辑上）被所有接收者接收到，所有满足条件的 BroadcastReceiver 都会随机地执行其 onReceive() 方法。

　　同级别接收是先后是随机的；级别低的收到广播；

　　消息传递的效率比较高，并且无法中断广播的传播。

##有序广播

应用在需要有特定拦截的场景下使用，如黑名单短信、电话拦截。

使用方法：

　sendOrderedBroadcast(intent, receiverPermission);

　receiverPermission ：一个接收器必须持以接收您的广播。如果为 null ，不经许可的要求（一般都为null）。

```

//发送有序广播

sendOrderedBroadcast(intent, null);

```

　　在有序广播中，我们可以在前一个广播接收者将处理好的数据传送给后面的广播接收者，也可以调用abortBroadcast()来终结广播的传播。

```

public void onReceive(Context arg0, Intent intent) {

　　//获取上一个广播的bundle数据

　　Bundle bundle = getResultExtras(true);//true：前一个广播没有结果时创建新的Bundle；false：不创建Bundle

　　bundle.putString("key", "777");

　　//将bundle数据放入广播中传给下一个广播接收者

　　setResultExtras(bundle);

　　//终止广播传给下一个广播接收者

　　abortBroadcast();

}

```

　　高级别的广播收到该广播后，可以决定把该广播是否截断掉。

　　同级别接收是先后是随机的，如果先接收到的把广播截断了，同级别的例外的接收者是无法收到该广播。

##异步广播

使用方法：sendStickyBroadcast() ：

　　发出的Intent当接收Activity（动态注册）重新处于onResume状态之后就能重新接受到其Intent.（the Intent will be held to be re-broadcast to future receivers）。就是说sendStickyBroadcast发出的最后一个Intent会被保留，下次当Activity处于活跃的时候又会接受到它。

发这个广播需要权限：

```

<uses-permission android:name="android.permission.BROADCAST\_STICKY" />

```

卸载该广播：

```

removeStickyBroadcast(intent);

```

　　在卸载之前该intent会保留，接收者在可接收状态都能获得。

##异步有序广播

　　使用方法：sendStickyOrderedBroadcast(intent, resultReceiver, scheduler,

initialCode, initialData, initialExtras)：

　　这个方法具有有序广播的特性也有异步广播的特性；

　　同时需要限权：

```

<uses-permission android:name="android.permission.BROADCAST\_STICKY" />

```

#总结

- 静态广播接收的处理器是由PackageManagerService负责，当手机启动或者新安装了应用的时候，PackageManagerService会扫描手机中所有已安装的APP应用，将AndroidManifest.xml中有关注册广播的信息解析出来，存储至一个全局静态变量当中。

- 动态广播接收的处理器是由ActivityManagerService负责，当APP的服务或者进程起来之后，执行了注册广播接收的代码逻辑，即进行加载，最后会存储在一个另外的全局静态变量中。需要注意的是：

　　 1、 这个并非是一成不变的，当程序被杀死之后，已注册的动态广播接收器也会被移出全局变量，直到下次程序启动，再进行动态广播的注册，当然这里面的顺序也已经变更了一次。

　　 2、这里也并没完整的进行广播的排序，只记录的注册的先后顺序，并未有结合优先级的处理。

- 广播发出的时候，广播接收者接收的顺序如下：

　　１．当广播为\*\*普通广播\*\*时，有如下的接收顺序：

　　无视优先级

　　动态优先于静态

　　同优先级的动态广播接收器，\*\*先注册的大于后注册的\*\*

　　同优先级的静态广播接收器，\*\*先扫描的大于后扫描的\*\*

　　２．如果广播为\*\*有序广播\*\*，那么会将动态广播处理器和静态广播处理器合并在一起处理广播的消息，最终确定广播接收的顺序：

　　优先级高的先接收

　　同优先级的动静态广播接收器，\*\*动态优先于静态\*\*

　　同优先级的动态广播接收器，\*\*先注册的大于后注册的\*\*

　　同优先级的静态广播接收器，\*\*先扫描的大于后扫描的\*\*

#一些常用的系统广播的action 和permission

- 开机启动

```

<action android:name="android.intent.action.BOOT\_COMPLETED"/>

```

```

<uses-permission android:name="android.permission.RECEIVE\_BOOT\_COMPLETED" />

```

- 网络状态

```

<action android:name="android.net.conn.CONNECTIVITY\_CHANGE"/>

```

```

<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE"/>

```

　　　网络是否可用的方法：

```

public static boolean isNetworkAvailable(Context context) {

ConnectivityManager mgr = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY\_SERVICE);

NetworkInfo[] info = mgr.getAllNetworkInfo();

if (info != null) {

for (int i = 0; i < info.length; i++) {

if (info[i].getState() == NetworkInfo.State.CONNECTED) {

return true;

}

}

}

return false;

}

```

- 电量变化

```

<action android:name="android.intent.action.BATTERY\_CHANGED"/>

```

BroadcastReceiver 的onReceive方法：

```

public void onReceive(Context context, Intent intent) {

int currLevel = intent.getIntExtra(BatteryManager.EXTRA\_LEVEL, 0); //当前电量

int total = intent.getIntExtra(BatteryManager.EXTRA\_SCALE, 1); //总电量

int percent = currLevel \* 100 / total;

Log.i(TAG, "battery: " + percent + "%");

}

```

## 第7节 Android异步任务机制之AsycTask

> 在Android中实现异步任务机制有两种方式，\*\*Handler\*\*和\*\*AsyncTask\*\*。

>

> Handler已经在上一篇文章 [异步消息处理机制（Handler 、 Looper 、MessageQueue）源码解析](http://blog.csdn.net/amazing7/article/details/51424038#reply) 说过了。

>

> 本篇就说说AsyncTask的异步实现。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

1、什么时候使用 AsnyncTask

</font>

</td>

</tr>

</table>

##

　　在上一篇文章已经说了，主线程主要负责控制UI页面的显示、更新、交互等。 为了有更好的用户体验，UI线程中的操作要求越短越好。

　　我们把耗时的操作（例如网络请求、数据库操作、复杂计算）放到单独的子线程中操作，以避免主线程的阻塞。但是在子线程中不能更新ＵＩ界面，这时候需要使用handler。

　　但如果耗时的操作太多，那么我们需要开启太多的子线程，这就会给系统带来巨大的负担，随之也会带来性能方面的问题。在这种情况下我们就可以考虑使用类AsyncTask来异步执行任务，不需要子线程和handler，就可以完成异步操作和刷新UI。

　　不要随意使用AsyncTask,除非你必须要与UI线程交互.默认情况下使用Thread即可,要注意需要将线程优先级调低.AsyncTask适合处理短时间的操作,长时间的操作,比如下载一个很大的视频,这就需要你使用自己的线程来下载,不管是断点下载还是其它的.

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

２、AsnyncTask原理

</font>

</td>

</tr>

</table>

##

　　AsyncTask主要有二个部分：一个是与主线程的交互，另一个就是线程的管理调度。虽然可能多个AsyncTask的子类的实例，但是AsyncTask的内部Handler和ThreadPoolExecutor都是进程范围内共享的，其都是static的，也即属于类的，类的属性的作用范围是CLASSPATH，因为一个进程一个VM，所以是AsyncTask控制着进程范围内所有的子类实例。

　　AsyncTask内部会创建一个进程作用域的线程池来管理要运行的任务，也就就是说当你调用了AsyncTask的execute()方法后，AsyncTask会把任务交给线程池，由线程池来管理创建Thread和运行Therad。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

３、AsyncTask介绍

</font>

</td>

</tr>

</table>

##

　　Android的AsyncTask比Handler更轻量级一些（只是代码上轻量一些，而实际上要比handler更耗资源），适用于简单的异步处理。

　　Android之所以有Handler和AsyncTask，都是为了不阻塞主线程（UI线程），因为UI的更新只能在主线程中完成，因此异步处理是不可避免的。

　　AsyncTask：对线程间的通讯做了包装，是后台线程和UI线程可以简易通讯：后台线程执行异步任务，将result告知UI线程。

使用AsyncTask分为两步：

①　继承AsyncTask类实现自己的类

```

public abstract class AsyncTask<Params, Progress, Result> {

```

> Params: 输入参数，对应excute()方法中传递的参数。如果不需要传递参数，则直接设为void即可。

>

> Progress：后台任务执行的百分比

>

> Result：返回值类型，和doInBackground（）方法的返回值类型保持一致。

②复写方法

最少要重写以下这两个方法：

- doInBackground(Params…)

　　在\*\*子线程\*\*（其他方法都在主线程执行）中执行比较耗时的操作，不能更新ＵＩ，可以在该方法中调用publishProgress(Progress…)来更新任务的进度。Progress方法是AsycTask中一个final方法只能调用不能重写。

- onPostExecute(Result)

　　使用在doInBackground 得到的结果处理操作UI， 在主线程执行，任务执行的结果作为此方法的参数返回。

有时根据需求还要实现以下三个方法：

- onProgressUpdate(Progress…)

　　可以使用进度条增加用户体验度。 此方法在主线程执行，用于显示任务执行的进度。

- onPreExecute()

　　这里是最终用户调用Excute时的接口，当任务执行之前开始调用此方法，可以在这里显示进度对话框。

- onCancelled()

　　用户调用取消时，要做的操作

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

４、AsyncTask示例

</font>

</td>

</tr>

</table>

##

按照上面的步骤定义自己的异步类：

```

public class MyTask extends AsyncTask<String, Integer, String> {

//执行的第一个方法用于在执行后台任务前做一些UI操作

@Override

protected void onPreExecute() {

}

//第二个执行方法,在onPreExecute()后执行，用于后台任务,不可在此方法内修改UI

@Override

protected String doInBackground(String... params) {

//处理耗时操作

return "后台任务执行完毕";

}

/\*这个函数在doInBackground调用publishProgress(int i)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 \*/

@Override

protected void onProgressUpdate(Integer... progresses) {

//"loading..." + progresses[0] + "%"

super.onProgressUpdate(progress);

}

/\*doInBackground返回时触发，换句话说，就是doInBackground执行完后触发

这里的result就是上面doInBackground执行后的返回值，所以这里是"后台任务执行完毕" \*/

@Override

protected void onPostExecute(String result) {

}

//onCancelled方法用于在取消执行中的任务时更改UI

@Override

protected void onCancelled() {

}

}

```

在主线程申明该类的对象，调用对象的execute（）函数开始执行。

```

MyTask ｔ= new MyTask();

t.execute();//这里没有参数

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

5、使用AsyncTask需要注意的地方

</font>

</td>

</tr>

</table>

##

- AsnycTask内部的Handler需要和主线程交互，所以AsyncTask的实例必须在UI线程中创建

- AsyncTaskResult的doInBackground(mParams)方法执行异步任务运行在子线程中，其他方法运行在主线程中，可以操作UI组件。

- 一个AsyncTask任务只能被执行一次。

- 运行中可以随时调用AsnycTask对象的cancel(boolean)方法取消任务，如果成功，调用isCancelled()会返回true，并且不会执行 onPostExecute() 方法了，而是执行 onCancelled() 方法。

- 对于想要立即开始执行的异步任务，要么直接使用Thread，要么单独创建线程池提供给AsyncTask。默认的AsyncTask不一定会立即执行你的任务，除非你提供给他一个单独的线程池。如果不与主线程交互，直接创建一个Thread就可以了。

## 第8节 Android启动过程图解

Android手机开机执行过程图：

![这里写图片描述](http://img.blog.csdn.net/20160603133621981)

从开机到桌面的过程为：

\*\*Bootloader\*\* ➪\*\*Kernel\*\* ➪\*\*Init进程\*\* ➪ \*\*Zygote\*\* ➪ \*\*SystemServer\*\* ➪ \*\*ServiceManager\*\* ➪ \*\*Home Launcher\*\*

　　Android服务包括系统服务和应用服务，系统服务是指Android系统在启动过程就已经启动实现了的服务，对于系统服务又分为Java服务和本地服务，Java服务是由Java代码编写而成，由SystemServer进程提供，而本地服务是由C/C++实现的服务，由Init进程在系统启动时启动的服务。应用服务是由开发者自行实现的某些特定服务。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

１、Bootloader

</font>

</td>

</tr>

</table>

##

　　当电源按下，引导芯片代码开始从预定义的地方（固化在ROM）开始执行。加载引导程序到RAM，然后执行。

　　BootLoader是在操作系统内核运行之前运行。可以初始化硬件设备、建立内存空间映射图，从而将系统的软硬件环境带到一个合适状态，以便为最终调用操作系统内核准备好正确的环境。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

２、Kernel

</font>

</td>

</tr>

</table>

##

　　Android内核启动时，会设置缓存、被保护存储器、计划列表，加载驱动。当内核完成系统设置，它首先在系统文件中寻找”init”文件，然后启动root进程或者系统的第一个进程。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

３、init进程

</font>

</td>

</tr>

</table>

##

　　init进程，它是一个由内核启动的用户级进程。内核自行启动（已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等）之后，就通过启动一个用户级程序init的方式，完成引导进程。init始终是第一个进程。

　　启动过程就是代码init.c中main函数执行过程：system\core\init\init.c在函数中执行了：\*\*文件夹建立\*\*，\*\*挂载\*\*，\*\*rc文件解析\*\*，\*\*属性设置\*\*，\*\*启动服务\*\*，\*\*执行动作\*\*，\*\*socket监听\*\*……

- rc文件解析

　　.rc文件是Android使用的初始化脚本文件 ，Android中有特定的格式以及规则。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

４、Zygote

</font>

</td>

</tr>

</table>

##

　　所有的应用程序进程以及系统服务进程（SystemServer）都是由Zygote进程孕育（fork）出来的，zygote本身是Native应用程序，与驱动内核无关。

　　我们知道，Android系统是基于Linux内核的，而在Linux系统中，所有的进程都是init进程的子孙进程，也就是说，所有的进程都是直接或者间接地由init进程fork出来的。Zygote进程也不例外，它是在系统启动的过程，由init进程创建的（在系统启动脚本system/core/rootdir/init.rc文件中）。

　　在Java中，不同的虚拟机实例会为不同的应用分配不同的内存。假如Android应用应该尽可能快地启动，但如果Android系统为每一个应用启动不同的Dalvik虚拟机实例，就会消耗大量的内存以及时间。因此，为了克服这个问题，Android系统创造了”Zygote”。Zygote是一个虚拟器进程，预加载以及初始化核心库类，让Dalvik虚拟机共享代码、降低内存占用和启动时间。

\*\*Zygote进程包含两个主要模块：\*\*

①. Socket服务端，该Socket服务端用于接收启动新的Dalvik进程命令。

②. Framework共享类及共享资源，当Zygote进程启动后，会装载一些共享类和资源，共享类是在preload-classes文件中定义的，共享资源是在preload-resources文件中定义。因为其他Dalvik进程是由Zygote进程孵化出来的，因此只要Zygote装载好了这些类和资源后，新的Dalvik进程就不需要在装载这些类和资源了，它们共享Zygote进程的资源和类。

\*\*Zygote启动分为两个阶段：\*\*

　①. \*\*虚拟机启动 --- 通过native启动\*\*

- startVm(&mJavaVM, &env) 　　启动虚拟机

- onVmCreated(env) 虚拟机启动后的初始化

- startReg(env) 注册JNI函数

- env->CallStaticVoidMethod(startClass, startMeth, strArray) 调用ZygoteInit类的main函数开创java世界

②. \*\*SystemServer进程 --- 通过Java启动\*\*

- registerZygoteSocket() 　为zygote进程注册监听socket

- preload() 加载常用的JAVA类和系统资源

- startSystemServer() 启动SystemServer进程

- runSelectLoopMode() 进入循环监听模式

- closeServerSocket() 进程退出时，关闭socket监听

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

５、启动系统服务

</font>

</td>

</tr>

</table>

##

　　Zygote创建新的进程去启动系统服务。你可以在ZygoteInit类的”startSystemServer”方法中找到源代码。

核心服务：

> 启动电源管理器；

>

创建Activity管理器；

>启动电话注册；

>启动包管理器；

>

>设置Activity管理服务为系统进程；

>

>启动上下文管理器；

>

>启动系统Context Providers；

>

>启动电池服务；

>

>启动定时管理器；

>

>启动传感服务；

>

>启动窗口管理器；

>

>启动蓝牙服务；

>

>启动挂载服务。

>

其他服务：

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

６、引导完成

</font>

</td>

</tr>

</table>

##

　　一旦系统服务在内存中跑起来了，Android就完成了引导过程。在这个时候“ACTION\_BOOT\_COMPLETED”开机启动广播就会发出去。

## 第9节 Android 自定义View入门

> 在android应用开发过程中，固定的一些控件和属性可能满足不了开发的需求，所以在一些特殊情况下，我们需要自定义控件与属性。

#一、实现步骤

　 1. 继承View类或其子类

　 2. 复写view中的一些函数

　３.为自定义View类增加属性（两种方式）

　４.绘制控件（导入布局）

　５.响应用户事件

　６.定义回调函数（根据自己需求来选择）

#二、哪些方法需要被重写

- onDraw()

　　view中onDraw()是个空函数，也就是说具体的视图都要覆写该函数来实现自己的绘制。对于ViewGroup则不需要实现该函数，因为作为容器是“没有内容“的（但必须实现dispatchDraw()函数，告诉子view绘制自己）。

- onLayout()

　　主要是为viewGroup类型布局子视图用的，在View中这个函数为空函数。

- onMeasure()

　　用于计算视图大小（即长和宽）的方式，并通过setMeasuredDimension(width, height)保存计算结果。

- onTouchEvent

　　定义触屏事件来响应用户操作。

还有一些不常用的方法：

> 　　onKeyDown 当按下某个键盘时

>

　　onKeyUp 当松开某个键盘时

　　onTrackballEvent 当发生轨迹球事件时

　　onSizeChange() 当该组件的大小被改变时

　　onFinishInflate() 回调方法，当应用从XML加载该组件并用它构建界面之后调用的方法

　　onWindowFocusChanged(boolean) 当该组件得到、失去焦点时

　　onAttachedToWindow() 当把该组件放入到某个窗口时

　　onDetachedFromWindow() 当把该组件从某个窗口上分离时触发的方法

　　onWindowVisibilityChanged(int): 当包含该组件的窗口的可见性发生改变时触发的方法

\*\*View的绘制流程\*\*

绘制流程函数调用关系如下图：

![这里写图片描述](http://img.blog.csdn.net/20160617150747985)

我们调用requestLayout()的时候，会触发measure 和 layout 过程，调用invalidate,会执行 draw 过程。

#三.自定义控件的三种方式

１. 继承已有的控件

　　当要实现的控件和已有的控件在很多方面比较类似, 通过对已有控件的扩展来满足要求。

２. 继承一个布局文件

　　一般用于自定义组合控件，在构造函数中通过inflater和addView()方法加载自定义控件的布局文件形成图形界面（不需要onDraw方法）。

３.继承view

　　通过onDraw方法来绘制出组件界面。

#四.自定义属性的两种方法

　　１．在布局文件中直接加入属性，在构造函数中去获得。

布局文件：

```

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

>

<com.example.demo.myView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

Text="@string/hello\_world"

/>

</RelativeLayout>

```

获取属性值：

```

public myView(Context context, AttributeSet attrs) {

super(context, attrs);

// TODO Auto-generated constructor stub

int textId = attrs.getAttributeResourceValue(null, "Text", 0);

String text = context.getResources().getText(textId).toString();

}

```

２．在res/values/ 下建立一个attrs.xml 来声明自定义view的属性。

可以定义的属性有：

```

<declare-styleable name = "名称">

//参考某一资源ID (name可以随便命名)

<attr name = "background" format = "reference" />

//颜色值

<attr name = "textColor" format = "color" />

//布尔值

<attr name = "focusable" format = "boolean" />

//尺寸值

<attr name = "layout\_width" format = "dimension" />

//浮点值

<attr name = "fromAlpha" format = "float" />

//整型值

<attr name = "frameDuration" format="integer" />

//字符串

<attr name = "text" format = "string" />

//百分数

<attr name = "pivotX" format = "fraction" />

//枚举值

<attr name="orientation">

<enum name="horizontal" value="0" />

<enum name="vertical" value="1" />

</attr>

//位或运算

<attr name="windowSoftInputMode">

<flag name = "stateUnspecified" value = "0" />

<flag name = "stateUnchanged" value = "1" />

</attr>

//多类型

<attr name = "background" format = "reference|color" />

</declare-styleable>

```

- attrs.xml进行属性声明

```

<?xml version="1.0" encoding="utf-8"?>

<resources>

<declare-styleable name="myView">

<attr name="text" format="string"/>

<attr name="textColor" format="color"/>

</declare-styleable>

</resources>

```

- 添加到布局文件

```

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

xmlns:myview="http://schemas.android.com/apk/com.example.demo"

>

<com.example.demo.myView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

myview:text = "test"

myview:textColor ="#ff0000"

/>

</RelativeLayout>

```

这里注意命名空间：

xmlns:前缀=”http://schemas.android.com/apk/res/包名（或res-auto）”，

前缀:TextColor　使用属性。

- 在构造函数中获取属性值

```

public myView(Context context, AttributeSet attrs) {

super(context, attrs);

// TODO Auto-generated constructor stub

TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.myView);

String text = a.getString(R.styleable.myView\_text);

int textColor = a.getColor(R.styleable.myView\_textColor, Color.WHITE);

a.recycle();

}

```

　或者：

```

public myView(Context context, AttributeSet attrs) {

super(context, attrs);

// TODO Auto-generated constructor stub

TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.myView);

int n = a.getIndexCount();

for(int i=0;i<n;i++){

int attr = a.getIndex(i);

switch (attr) {

case R.styleable.myView\_text:

break;

case R.styleable.myView\_textColor:

break;

}

}

a.recycle();

}

```

#五. 自定义随手指移动的小球(小例子)

<img src="http://img.blog.csdn.net/20160503143613554" width="210" height="334" />

实现上面的效果我们大致需要分成这几步

- 在res/values/ 下建立一个attrs.xml 来声明自定义view的属性

- 一个继承View并复写部分函数的自定义view的类

- 一个展示自定义view 的容器界面

1.自定义view命名为myView，它有一个属性值，格式为color、

```

<?xml version="1.0" encoding="utf-8"?>

<resources>

<declare-styleable name="myView">

<attr name="TextColor" format="color"/>

</declare-styleable>

</resources>

```

2.在构造函数获取获得view的属性配置和复写onDraw和onTouchEvent函数实现绘制界面和用户事件响应。

```

public class myView extends View{

//定义画笔和初始位置

Paint p = new Paint();

public float currentX = 50;

public float currentY = 50;

public int textColor;

public myView(Context context, AttributeSet attrs) {

super(context, attrs);

//获取资源文件里面的属性，由于这里只有一个属性值，不用遍历数组，直接通过R文件拿出color值

//把属性放在资源文件里，方便设置和复用

TypedArray array = context.obtainStyledAttributes(attrs,R.styleable.myView);

textColor = array.getColor(R.styleable.myView\_TextColor,Color.BLACK);

array.recycle();

}

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

//画一个蓝色的圆形

p.setColor(Color.BLUE);

canvas.drawCircle(currentX,currentY,30,p);

//设置文字和颜色，这里的颜色是资源文件values里面的值

p.setColor(textColor);

canvas.drawText("BY finch",currentX-30,currentY+50,p);

}

@Override

public boolean onTouchEvent(MotionEvent event) {

currentX = event.getX();

currentY = event.getY();

invalidate();//重新绘制图形

return true;

}

}

```

　　这里通过不断的更新当前位置坐标和重新绘制图形实现效果，要注意的是使用TypedArray后一定要记得recycle(). 否则会对下次调用产生影响。

　　![这里写图片描述](http://img.blog.csdn.net/20160503144335969)

３．把myView加入到activity\_main.xml布局里面

```

<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

xmlns:myview="http://schemas.android.com/apk/res-auto"

android:paddingBottom="@dimen/activity\_vertical\_margin"

android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

tools:context="finch.scu.cn.myview.MainActivity">

<finch.scu.cn.myview.myView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

myview:TextColor="#ff0000"

/>

</RelativeLayout>

```

４．最后是MainActivity

```

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

}

```

- 具体的view要根据具体的需求来，比如我们要侧滑删除的listview我们可以继承listview，监听侧滑事件，显示删除按钮实现功能。

## 第10节 Android 自定义ViewGroup入门实践

>　　 对自定义view还不是很了解的码友可以先看[自定义View入门](http://blog.csdn.net/Amazing7/article/details/51303289)这篇文章，本文主要对自定义ViewGroup的过程的梳理，废话不多说。

#1.View 绘制流程

　　ViewGroup也是继承于View，下面看看绘制过程中依次会调用哪些函数。

　![这里写图片描述](http://img.blog.csdn.net/20160617180933525)

说明：

- measure()和onMeasure()

　　在View.Java源码中：

```

public final void measure(int widthMeasureSpec,int heightMeasureSpec){

...

onMeasure

...

}

protected void onMeasure(int widthMeasureSpec,int heightMeasureSpec) {

setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),

getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));

}

```

　　可以看出measure()是被final修饰的，这是不可被重写。onMeasure在measure方法中调用的，当我们继承View的时候通过重写onMeasure方法来测量控件大小。

　　layout()和onLayout(),draw()和onDraw()类似。

- dispatchDraw()

　　View 中这个函数是一个空函数，ViewGroup 复写了dispatchDraw()来对其子视图进行绘制。自定义的 ViewGroup 一般不对dispatchDraw()进行复写。

- requestLayout()

　　当布局变化的时候，比如方向变化，尺寸的变化，会调用该方法，在自定义的视图中，如果某些情况下希望重新测量尺寸大小，应该手动去调用该方法，它会触发measure()和layout()过程，但不会进行 draw。

自定义ViewGroup的时候一般复写

> onMeasure()方法：

>

　　计算childView的测量值以及模式，以及设置自己的宽和高

>onLayout()方法，

>　对其所有childView的位置进行定位

View树：

![这里写图片描述](http://img.blog.csdn.net/20160617181010906)

　树的遍历是有序的，由父视图到子视图，每一个 ViewGroup 负责测绘它所有的子视图，而最底层的 View 会负责测绘自身。

- \*\*measure：\*\*

　　自上而下进行遍历，根据父视图对子视图的MeasureSpec以及ChildView自身的参数，通过

```

getChildMeasureSpec(parentHeightMeasure,mPaddingTop+mPaddingBottom，lp.height)

```

　　获取ChildView的MeasureSpec，回调ChildView.measure最终调用setMeasuredDimension得到ChildView的尺寸：

```

mMeasuredWidth 和 mMeasuredHeight

```

- \*\*Layout ：\*\*

　　　也是自上而下进行遍历的，该方法计算每个ChildView的ChildLeft,ChildTop；与measure中得到的每个ChildView的mMeasuredWidth 和 mMeasuredHeight，来对ChildView进行布局。

```

child.layout(left,top,left+width,top+height)

```

#２.onMeasure过程

　　measure过程会为一个View及所有子节点的mMeasuredWidth

和mMeasuredHeight变量赋值，该值可以通过getMeasuredWidth()和getMeasuredHeight()方法获得。

\*\*onMeasure过程传递尺寸的两个类：\*\*

- \*\*ViewGroup.LayoutParams\*\* （ViewGroup 自身的布局参数）

　　用来指定视图的高度和宽度等参数，使用 view.getLayoutParams() 方法获取一个视图LayoutParams，该方法得到的就是其所在父视图类型的LayoutParams，比如View的父控件为RelativeLayout，那么得到的 LayoutParams 类型就为RelativeLayoutParams。

> ①具体值

>

②MATCH\_PARENT 表示子视图希望和父视图一样大(不包含 padding 值)

>

③WRAP\_CONTENT 表示视图为正好能包裹其内容大小(包含 padding 值)

- \*\*MeasureSpecs\*\*

　　测量规格，包含测量要求和尺寸的信息，有三种模式:

> ①UNSPECIFIED

>

　　父视图不对子视图有任何约束，它可以达到所期望的任意尺寸。比如 ListView、ScrollView，一般自定义 View 中用不到

> ②EXACTLY

>

　　父视图为子视图指定一个确切的尺寸，而且无论子视图期望多大，它都必须在该指定大小的边界内，对应的属性为 match\_parent 或具体值，比如 100dp，父控件可以通过MeasureSpec.getSize(measureSpec)直接得到子控件的尺寸。

> ③AT\_MOST

>

　　 父视图为子视图指定一个最大尺寸。子视图必须确保它自己所有子视图可以适应在该尺寸范围内，对应的属性为 wrap\_content，这种模式下，父控件无法确定子 View 的尺寸，只能由子控件自己根据需求去计算自己的尺寸，这种模式就是我们自定义视图需要实现测量逻辑的情况。

#３.onLayout 过程

　　子视图的具体位置都是相对于父视图而言的。View 的 onLayout 方法为空实现，而 ViewGroup 的 onLayout 为 abstract 的，因此，如果自定义的自定义ViewGroup 时，必须实现 onLayout 函数。

　　在 layout 过程中，子视图会调用getMeasuredWidth()和getMeasuredHeight()方法获取到 measure 过程得到的 mMeasuredWidth 和 mMeasuredHeight，作为自己的 width 和 height。然后调用每一个子视图的layout(l, t, r, b)函数，来确定每个子视图在父视图中的位置。

#4.示例程序

先上效果图：

![这里写图片描述](http://img.blog.csdn.net/20160617215723283)

代码中有详细的注释，结合上文中的说明，理解应该没有问题。这里主要贴出核心代码。

FlowLayout.java中(参照阳神的慕课课程)

> onMeasure方法

```

@Override

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec)

{

// 获得它的父容器为它设置的测量模式和大小

int sizeWidth = MeasureSpec.getSize(widthMeasureSpec);

int modeWidth = MeasureSpec.getMode(widthMeasureSpec);

int sizeHeight = MeasureSpec.getSize(heightMeasureSpec);

int modeHeight = MeasureSpec.getMode(heightMeasureSpec);

// 用于warp\_content情况下，来记录父view宽和高

int width = 0;

int height = 0;

// 取每一行宽度的最大值

int lineWidth = 0;

// 每一行的高度累加

int lineHeight = 0;

// 获得子view的个数

int cCount = getChildCount();

for (int i = 0; i < cCount; i++)

{

View child = getChildAt(i);

// 测量子View的宽和高（子view在布局文件中是wrap\_content）

measureChild(child, widthMeasureSpec, heightMeasureSpec);

// 得到LayoutParams

MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams();

// 根据测量宽度加上Margin值算出子view的实际宽度（上文中有说明）

int childWidth = child.getMeasuredWidth() + lp.leftMargin + lp.rightMargin;

// 根据测量高度加上Margin值算出子view的实际高度

int childHeight = child.getMeasuredHeight() + lp.topMargin+ lp.bottomMargin;

// 这里的父view是有padding值的，如果再添加一个元素就超出最大宽度就换行

if (lineWidth + childWidth > sizeWidth - getPaddingLeft() - getPaddingRight())

{

// 父view宽度=以前父view宽度、当前行宽的最大值

width = Math.max(width, lineWidth);

// 换行了，当前行宽=第一个view的宽度

lineWidth = childWidth;

// 父view的高度=各行高度之和

height += lineHeight;

//换行了，当前行高=第一个view的高度

lineHeight = childHeight;

} else{

// 叠加行宽

lineWidth += childWidth;

// 得到当前行最大的高度

lineHeight = Math.max(lineHeight, childHeight);

}

// 最后一个控件

if (i == cCount - 1)

{

width = Math.max(lineWidth, width);

height += lineHeight;

}

}

/\*\*

\* EXACTLY对应match\_parent 或具体值

\* AT\_MOST对应wrap\_content

\* 在FlowLayout布局文件中

\* android:layout\_width="fill\_parent"

\* android:layout\_height="wrap\_content"

\*

\* 如果是MeasureSpec.EXACTLY则直接使用父ViewGroup传入的宽和高，否则设置为自己计算的宽和高。

\*/

setMeasuredDimension(

modeWidth == MeasureSpec.EXACTLY ? sizeWidth : width + getPaddingLeft() + getPaddingRight(),

modeHeight == MeasureSpec.EXACTLY ? sizeHeight : height + getPaddingTop()+ getPaddingBottom()

);

}

```

> onLayout方法

```

//存储所有的View

private List<List<View>> mAllViews = new ArrayList<List<View>>();

//存储每一行的高度

private List<Integer> mLineHeight = new ArrayList<Integer>();

@Override

protected void onLayout(boolean changed, int l, int t, int r, int b)

{

mAllViews.clear();

mLineHeight.clear();

// 当前ViewGroup的宽度

int width = getWidth();

int lineWidth = 0;

int lineHeight = 0;

// 存储每一行所有的childView

List<View> lineViews = new ArrayList<View>();

int cCount = getChildCount();

for (int i = 0; i < cCount; i++)

{

View child = getChildAt(i);

MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams();

int childWidth = child.getMeasuredWidth();

int childHeight = child.getMeasuredHeight();

lineWidth += childWidth + lp.leftMargin + lp.rightMargin;

lineHeight = Math.max(lineHeight, childHeight + lp.topMargin+ lp.bottomMargin);

lineViews.add(child);

// 换行，在onMeasure中childWidth是加上Margin值的

if (childWidth + lineWidth + lp.leftMargin + lp.rightMargin > width - getPaddingLeft() - getPaddingRight())

{

// 记录行高

mLineHeight.add(lineHeight);

// 记录当前行的Views

mAllViews.add(lineViews);

// 新行的行宽和行高

lineWidth = 0;

lineHeight = childHeight + lp.topMargin + lp.bottomMargin;

// 新行的View集合

lineViews = new ArrayList<View>();

}

}

// 处理最后一行

mLineHeight.add(lineHeight);

mAllViews.add(lineViews);

// 设置子View的位置

int left = getPaddingLeft();

int top = getPaddingTop();

// 行数

int lineNum = mAllViews.size();

for (int i = 0; i < lineNum; i++)

{

// 当前行的所有的View

lineViews = mAllViews.get(i);

lineHeight = mLineHeight.get(i);

for (int j = 0; j < lineViews.size(); j++)

{

View child = lineViews.get(j);

// 判断child的状态

if (child.getVisibility() == View.GONE)

{

continue;

}

MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams();

int lc = left + lp.leftMargin;

int tc = top + lp.topMargin;

int rc = lc + child.getMeasuredWidth();

int bc = tc + child.getMeasuredHeight();

// 为子View进行布局

child.layout(lc, tc, rc, bc);

left += child.getMeasuredWidth() + lp.leftMargin+ lp.rightMargin;

}

left = getPaddingLeft() ;

top += lineHeight ;

}

}

/\*\*

\* 因为我们只需要支持margin，所以直接使用系统的MarginLayoutParams

\*/

@Override

public LayoutParams generateLayoutParams(AttributeSet attrs)

{

return new MarginLayoutParams(getContext(), attrs);

}

```

> 以及MainActivity.java

```

public class MainActivity extends Activity {

LayoutInflater mInflater;

@InjectView(R.id.id\_flowlayout1)

FlowLayout idFlowlayout1;

@InjectView(R.id.id\_flowlayout2)

FlowLayout idFlowlayout2;

private String[] mVals = new String[]

{"Do", "one thing", "at a time", "and do well.", "Never", "forget",

"to say", "thanks.", "Keep on", "going ", "never give up."};

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

ButterKnife.inject(this);

mInflater = LayoutInflater.from(this);

initFlowlayout2();

}

public void initFlowlayout2() {

for (int i = 0; i < mVals.length; i++) {

final RelativeLayout rl2 = (RelativeLayout) mInflater.inflate(R.layout.flow\_layout, idFlowlayout2, false);

TextView tv2 = (TextView) rl2.findViewById(R.id.tv);

tv2.setText(mVals[i]);

rl2.setTag(i);

idFlowlayout2.addView(rl2);

rl2.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

int i = (int) v.getTag();

addViewToFlowlayout1(i);

rl2.setBackgroundResource(R.drawable.flow\_layout\_disable\_bg);

rl2.setClickable(false);

}

});

}

}

public void addViewToFlowlayout1(int i){

RelativeLayout rl1 = (RelativeLayout) mInflater.inflate(R.layout.flow\_layout, idFlowlayout1, false);

ImageView iv = (ImageView) rl1.findViewById(R.id.iv);

iv.setVisibility(View.VISIBLE);

TextView tv1 = (TextView) rl1.findViewById(R.id.tv);

tv1.setText(mVals[i]);

rl1.setTag(i);

idFlowlayout1.addView(rl1);

rl1.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

int i = (int) v.getTag();

idFlowlayout1.removeView(v);

View view = idFlowlayout2.getChildAt(i);

view.setClickable(true);

view.setBackgroundResource(R.drawable.flow\_layout\_bg);

}

});

}

```

> 这个项目源码已经上传，想要看源码的朋友可以

>点击 [FlowLayout](https://github.com/fanrunqi/FlowLayout)

>

> 如果有什么疑问可以给我留言，不足之处欢迎在github上指出，谢谢！

## 第11节 Android 缓存机制

>　　移动开发本质上就是手机和服务器之间进行通信，需要从服务端获取数据。反复通过网络获取数据是比较耗时的，特别是访问比较多的时候，会极大影响了性能，Android中可通过缓存机制来减少频繁的网络操作，减少流量、提升性能。

# 实现原理

　　把不需要实时更新的数据缓存下来，通过时间或者其他因素　来判别是读缓存还是网络请求，这样可以缓解服务器压力，一定程度上提高应用响应速度，并且支持离线阅读。

#Bitmap的缓存

　　在许多的情况下(像 ListView, GridView 或 ViewPager 之类的组件 )我们需要一次性加载大量的图片，在屏幕上显示的图片和所有待显示的图片有可能需要马上就在屏幕上无限制的进行滚动、切换。

　　像ListView, GridView 这类组件，它们的子项当不可见时，所占用的内存会被回收以供正在前台显示子项使用。垃圾回收器也会释放你已经加载了的图片占用的内存。如果你想让你的UI运行流畅的话，就不应该每次显示时都去重新加载图片。保持一些内存和文件缓存就变得很有必要了。

##使用内存缓存

　　通过预先消耗应用的一点内存来存储数据，便可快速的为应用中的组件提供数据，是一种典型的以\*\*空间换时间\*\*的策略。

　　LruCache 类（Android v4 Support Library 类库中开始提供）非常适合来做图片缓存任务 ，它可以使用一个LinkedHashMap 的强引用来保存最近使用的对象，并且当它保存的对象占用的内存总和超出了为它设计的最大内存时会把\*\*不经常使用\*\*的对象成员踢出以供垃圾回收器回收。

　　给LruCache 设置一个合适的内存大小，需考虑如下因素：

- 还剩余多少内存给你的activity或应用使用

- 屏幕上需要一次性显示多少张图片和多少图片在等待显示

- 手机的大小和密度是多少（密度越高的设备需要越大的 缓存）

- 图片的尺寸（决定了所占用的内存大小）

- 图片的访问频率（频率高的在内存中一直保存）

- 保存图片的质量（不同像素的在不同情况下显示）

具体的要根据应用图片使用的具体情况来找到一个合适的解决办法，一个设置 LruCache 例子：

```

private LruCache<String, Bitmap> mMemoryCache;

@Override

protected void onCreate(Bundle savedInstanceState) {

...

// 获得虚拟机能提供的最大内存，超过这个大小会抛出OutOfMemory的异常

final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);

// 用１／８的内存大小作为内存缓存

final int cacheSize = maxMemory / 8;

mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {

@Override

protected int sizeOf(String key, Bitmap bitmap) {

// 这里返回的不是item的个数，是cache的size（单位1024个字节）

return bitmap.getByteCount() / 1024;

}

};

...

}

public void addBitmapToMemoryCache(String key, Bitmap bitmap) {

if (getBitmapFromMemCache(key) == null) {

mMemoryCache.put(key, bitmap);

}

}

public Bitmap getBitmapFromMemCache(String key) {

return mMemoryCache.get(key);

}

```

　　当为ImageView加载一张图片时，会先在LruCache 中看看有没有缓存这张图片，如果有的话直接更新到ImageView中，如果没有的话，一个后台线程会被触发来加载这张图片。

```

public void loadBitmap(int resId, ImageView imageView) {

final String imageKey = String.valueOf(resId);

// 查看下内存缓存中是否缓存了这张图片

final Bitmap bitmap = getBitmapFromMemCache(imageKey);

if (bitmap != null) {

mImageView.setImageBitmap(bitmap);

} else {

mImageView.setImageResource(R.drawable.image\_placeholder);

BitmapWorkerTask task = new BitmapWorkerTask(mImageView);

task.execute(resId);

}

}

```

在图片加载的Task中，需要把加载好的图片加入到内存缓存中。

```

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {

...

// 在后台完成

@Override

protected Bitmap doInBackground(Integer... params) {

final Bitmap bitmap = decodeSampledBitmapFromResource(

getResources(), params[0], 100, 100));

addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);

return bitmap;

}

...

}

```

##使用磁盘缓存

　　内存缓存能够快速的获取到最近显示的图片，但不一定就能够获取到。当数据集过大时很容易把内存缓存填满（如GridView ）。你的应用也有可能被其它的任务（比如来电）中断进入到后台，后台应用有可能会被杀死，那么相应的内存缓存对象也会被销毁。 当你的应用重新回到前台显示时，你的应用又需要一张一张的去加载图片了。

　磁盘文件缓存能够用来处理这些情况，保存处理好的图片，当内存缓存不可用的时候，直接读取在硬盘中保存好的图片，这样可以有效的减少图片加载的次数。读取磁盘文件要比直接从内存缓存中读取要慢一些，而且需要在一个UI主线程外的线程中进行，因为磁盘的读取速度是不能够保证的，磁盘文件缓存显然也是一种以\*\*空间换时间\*\*的策略。

　　如果图片使用非常频繁的话，一个 ContentProvider 可能更适合代替去存储缓存图片，比如图片gallery 应用。

　　下面是一个DiskLruCache的部分代码：

```

private DiskLruCache mDiskLruCache;

private final Object mDiskCacheLock = new Object();

private boolean mDiskCacheStarting = true;

private static final int DISK\_CACHE\_SIZE = 1024 \* 1024 \* 10; // 10MB

private static final String DISK\_CACHE\_SUBDIR = "thumbnails";

@Override

protected void onCreate(Bundle savedInstanceState) {

...

// 初始化内存缓存

...

// 在后台线程中初始化磁盘缓存

File cacheDir = getDiskCacheDir(this, DISK\_CACHE\_SUBDIR);

new InitDiskCacheTask().execute(cacheDir);

...

}

class InitDiskCacheTask extends AsyncTask<File, Void, Void> {

@Override

protected Void doInBackground(File... params) {

synchronized (mDiskCacheLock) {

File cacheDir = params[0];

mDiskLruCache = DiskLruCache.open(cacheDir, DISK\_CACHE\_SIZE);

　 mDiskCacheStarting = false; // 结束初始化

　 mDiskCacheLock.notifyAll(); // 唤醒等待线程

}

return null;

}

}

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {

...

// 在后台解析图片

@Override

protected Bitmap doInBackground(Integer... params) {

final String imageKey = String.valueOf(params[0]);

// 在后台线程中检测磁盘缓存

Bitmap bitmap = getBitmapFromDiskCache(imageKey);

if (bitmap == null) { // 没有在磁盘缓存中找到图片

final Bitmap bitmap = decodeSampledBitmapFromResource(

getResources(), params[0], 100, 100));

}

// 把这个final类型的bitmap加到缓存中

addBitmapToCache(imageKey, bitmap);

return bitmap;

}

...

}

public void addBitmapToCache(String key, Bitmap bitmap) {

// 先加到内存缓存

if (getBitmapFromMemCache(key) == null) {

mMemoryCache.put(key, bitmap);

}

//再加到磁盘缓存

synchronized (mDiskCacheLock) {

if (mDiskLruCache != null && mDiskLruCache.get(key) == null) {

mDiskLruCache.put(key, bitmap);

}

}

}

public Bitmap getBitmapFromDiskCache(String key) {

synchronized (mDiskCacheLock) {

// 等待磁盘缓存从后台线程打开

while (mDiskCacheStarting) {

try {

mDiskCacheLock.wait();

} catch (InterruptedException e) {}

}

if (mDiskLruCache != null) {

return mDiskLruCache.get(key);

}

}

return null;

}

public static File getDiskCacheDir(Context context, String uniqueName) {

// 优先使用外缓存路径，如果没有挂载外存储，就使用内缓存路径

final String cachePath =

Environment.MEDIA\_MOUNTED.equals(Environment.getExternalStorageState()) ||

!isExternalStorageRemovable() ?getExternalCacheDir(context).getPath():context.getCacheDir().getPath();

return new File(cachePath + File.separator + uniqueName);

}

```

　　不能在UI主线程中进行这项操作，因为初始化磁盘缓存也需要对磁盘进行操作。上面的程序片段中，一个锁对象确保了磁盘缓存没有初始化完成之前不能够对磁盘缓存进行访问。

　　 内存缓存在UI线程中进行检测，磁盘缓存在UI主线程外的线程中进行检测，当图片处理完成之后，分别存储到内存缓存和磁盘缓存中。

##设备配置参数改变时加载问题

　　由于应用在运行的时候设备配置参数可能会发生改变，比如设备朝向改变，会导致Android销毁你的Activity然后按照新的配置重启，这种情况下，我们要避免重新去加载处理所有的图片，让用户能有一个流畅的体验。

　使用Fragment 能够把内存缓存对象传递到新的activity实例中，调用setRetainInstance(true)) 方法来保留Fragment实例。当activity重新创建好后， 被保留的Fragment依附于activity而存在，通过Fragment就可以获取到已经存在的内存缓存对象了，这样就可以快速的获取到图片，并设置到ImageView上，给用户一个流畅的体验。

下面是一个示例程序片段：

```

private LruCache<String, Bitmap> mMemoryCache;

@Override

protected void onCreate(Bundle savedInstanceState) {

...

RetainFragment mRetainFragment = RetainFragment.findOrCreateRetainFragment(getFragmentManager());

mMemoryCache = RetainFragment.mRetainedCache;

if (mMemoryCache == null) {

mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {

... //像上面例子中那样初始化缓存

}

mRetainFragment.mRetainedCache = mMemoryCache;

}

...

}

class RetainFragment extends Fragment {

private static final String TAG = "RetainFragment";

public LruCache<String, Bitmap> mRetainedCache;

public RetainFragment() {}

public static RetainFragment findOrCreateRetainFragment(FragmentManager fm) {

RetainFragment fragment = (RetainFragment) fm.findFragmentByTag(TAG);

if (fragment == null) {

fragment = new RetainFragment();

}

return fragment;

}

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// 使得Fragment在Activity销毁后还能够保留下来

setRetainInstance(true);

}

}

```

　　可以在不适用Fragment（没有界面的服务类Fragment）的情况下旋转设备屏幕。在保留缓存的情况下，你应该能发现填充图片到Activity中几乎是瞬间从内存中取出而没有任何延迟的感觉。任何图片优先从内存缓存获取，没有的话再到硬盘缓存中找，如果都没有，那就以普通方式加载图片。

参考：

[Caching Bitmaps](http://developer.android.com/training/displaying-bitmaps/cache-bitmap.html)

[LruCache](http://developer.android.com/reference/android/util/LruCache.html)

#使用SQLite进行缓存

　　网络请求数据完成后，把文件的相关信息（如url（一般作为唯一标示），下载时间，过期时间）等存放到数据库。下次加载的时候根据url先从数据库中查询，如果查询到并且时间未过期，就根据路径读取本地文件，从而实现缓存的效果。

　　注意：缓存的数据库是存放在/data/data/<package>/databases/目录下，是占用内存空间的，如果缓存累计，容易浪费内存，需要及时清理缓存。

#文件缓存

　　思路和一般缓存一样，把需要的数据存储在文件中，下次加载时判断文件是否存在和过期（使用File.lastModified()方法得到文件的最后修改时间，与当前时间判断），存在并未过期就加载文件中的数据，否则请求服务器重新下载。

　　注意，无网络环境下就默认读取文件缓存中的。

## 第12节 Android 数据存储五种方式使用与总结

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

1、概述

</font>

</td>

</tr>

</table>

##

　　Android提供了5种方式来让用户保存持久化应用程序数据。根据自己的需求来做选择，比如数据是否是应用程序私有的，是否能被其他程序访问，需要多少数据存储空间等，分别是：

①　使用SharedPreferences存储数据

②　文件存储数据

③　 SQLite数据库存储数据

④　使用ContentProvider存储数据

⑤　网络存储数据

Android提供了一种方式来暴露你的数据（甚至是私有数据）给其他应用程序 - ContentProvider。它是一个可选组件，可公开读写你应用程序数据。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

2、SharedPreferences存储

</font>

</td>

</tr>

</table>

##

　　SharedPreference类提供了一个总体框架，使您可以保存和检索的任何基本数据类型（ boolean, float, int, long, string）的持久键-值对（基于XML文件存储的“key-value”键值对数据）。

　　通常用来存储程序的一些配置信息。其存储在“data/data/程序包名/shared\_prefs目录下。

　　xml 处理时Dalvik会通过自带底层的本地XML Parser解析，比如XMLpull方式，这样对于内存资源占用比较好。

\*\*2.1\*\* 　我们可以通过以下两种方法获取SharedPreferences对象（通过Context）：

> ①　getSharedPreferences (String name, int mode)

　　当我们有多个SharedPreferences的时候，根据第一个参数name获得相应的SharedPreferences对象。

>②　getPreferences (int mode)

　　如果你的Activity中只需要一个SharedPreferences的时候使用。

这里的mode有四个选项：

```

Context.MODE\_PRIVATE

```

　　该SharedPreferences数据只能被本应用程序读、写。

```

Context.MODE\_WORLD\_READABLE

```

　　该SharedPreferences数据能被其他应用程序读，但不能写。

```

Context.MODE\_WORLD\_WRITEABLE

```

　　该SharedPreferences数据能被其他应用程序读和写。

```

Context.MODE\_MULTI\_PROCESS

```

　　sdk2.3后添加的选项，当多个进程同时读写同一个SharedPreferences时它会检查文件是否修改。

\*\*2.2\*\* 　向Shared Preferences中\*\*写入值\*\*

首先要通过 SharedPreferences.Editor获取到Editor对象；

然后通过Editor的putBoolean() 或 putString()等方法存入值；

最后调用Editor的commit()方法提交；

```

//Use 0 or MODE\_PRIVATE for the default operation

SharedPreferences settings = getSharedPreferences("fanrunqi", 0);

SharedPreferences.Editor editor = settings.edit();

editor.putBoolean("isAmazing", true);

// 提交本次编辑

editor.commit();

```

同时Edit还有两个常用的方法：

> editor.remove(String key) ：下一次commit的时候会移除key对应的键值对

>

>editor.clear()：移除所有键值对

\*\*2.3\*\* 　从Shared Preferences中\*\*读取值\*\*

　　读取值使用 SharedPreference对象的getBoolean()或getString()等方法就行了（没Editor 啥子事）。

```

SharedPreferences settings = getSharedPreferences("fanrunqi", 0);

boolean isAmazing= settings.getBoolean("isAmazing",true);

```

\*\*2.４\*\* 　Shared Preferences的优缺点

　　可以看出来Preferences是很轻量级的应用，使用起来也很方便，简洁。但存储数据类型比较单一（只有基本数据类型），无法进行条件查询，只能在不复杂的存储需求下使用，比如保存配置信息等。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

3、文件数据存储

</font>

</td>

</tr>

</table>

##

###3.1 使用内部存储

　　当文件被保存在内部存储中时，默认情况下，文件是应用程序私有的，其他应用不能访问。当用户卸载应用程序时这些文件也跟着被删除。

　　文件默认存储位置：/data/data/包名/files/文件名。

###3.1.1 创建和写入一个内部存储的私有文件：

①　调用Context的openFileOutput()函数，填入文件名和操作模式，它会返回一个FileOutputStream对象。

②　通过FileOutputStream对象的write()函数写入数据。

③　 FileOutputStream对象的close ()函数关闭流。

例如：

```

String FILENAME = "a.txt";

String string = "fanrunqi";

try {

FileOutputStream fos = openFileOutput(FILENAME, Context.MODE\_PRIVATE);

fos.write(string.getBytes());

fos.close();

} catch (Exception e) {

e.printStackTrace();

}

```

在 openFileOutput(String name, int mode)方法中

- name参数:　用于指定文件名称，不能包含路径分隔符“/” ，如果文件不存在，Android 会自动创建它。

- mode参数：用于指定操作模式，分为四种：

> Context.MODE\_PRIVATE = 0

　　为默认操作模式，代表该文件是私有数据，只能被应用本身访问，在该模式下，写入的内容会覆盖原文件的内容。

> Context.MODE\_APPEND = 32768

　　该模式会检查文件是否存在，存在就往文件追加内容，否则就创建新文件。

> Context.MODE\_WORLD\_READABLE = 1

　　表示当前文件可以被其他应用读取。

> MODE\_WORLD\_WRITEABLE

　　表示当前文件可以被其他应用写入。

###3.1.2 读取一个内部存储的私有文件：

① 调用openFileInput( )，参数中填入文件名，会返回一个FileInputStream对象。

② 使用流对象的 read()方法读取字节

③ 调用流的close()方法关闭流

例如：

```

String FILENAME = "a.txt";

try {

FileInputStream inStream = openFileInput(FILENAME);

int len = 0;

byte[] buf = new byte[1024];

StringBuilder sb = new StringBuilder();

while ((len = inStream.read(buf)) != -1) {

sb.append(new String(buf, 0, len));

}

inStream.close();

} catch (Exception e) {

e.printStackTrace();

}

```

其他一些经常用到的方法：

- getFilesDir()：　得到内存储文件的绝对路径

- getDir()：　在内存储空间中\*\*创建\*\*或\*\*打开一个已经存在\*\*的目录

- deleteFile()：　删除保存在内部存储的文件。

- fileList()：　返回当前由应用程序保存的文件的数组（内存储目录下的全部文件）。

###3.1.３　保存编译时的静态文件

　　如果你想在应用编译时保存静态文件，应该把文件保存在项目的　\*\*res/raw/\*\*　目录下，你可以通过 openRawResource()方法去打开它（传入参数R.raw.filename），这个方法返回一个 InputStream流对象你可以读取文件但是不能修改原始文件。

```

InputStream is = this.getResources().openRawResource(R.raw.filename);

```

###3.1.４　保存内存缓存文件

　　有时候我们只想缓存一些数据而不是持久化保存，可以使用getCacheDir（）去打开一个文件，文件的存储目录（ /data/data/包名/cache ）是一个应用专门来保存临时缓存文件的内存目录。

　　当设备的内部存储空间比较低的时候，Android可能会删除这些缓存文件来恢复空间，但是你不应该依赖系统来回收，要自己维护这些缓存文件把它们的大小限制在一个合理的范围内，比如1ＭＢ．当你卸载应用的时候这些缓存文件也会被移除。

##3.２ 使用外部存储（sdcard）

　　因为内部存储容量限制，有时候需要存储数据比较大的时候需要用到外部存储，使用外部存储分为以下几个步骤：

###3.2.1　添加外部存储访问限权

　　首先，要在AndroidManifest.xml中加入访问SDCard的权限，如下:

```

　<!-- 在SDCard中创建与删除文件权限 -->

<uses-permission android:name="android.permission.MOUNT\_UNMOUNT\_FILESYSTEMS"/>

<!-- 往SDCard写入数据权限 -->

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"/>

```

###3.2.２　检测外部存储的可用性

　　在使用外部存储时我们需要检测其状态，它可能被连接到计算机、丢失或者只读等。下面代码将说明如何检查状态：

```

//获取外存储的状态

String state = Environment.getExternalStorageState();

if (Environment.MEDIA\_MOUNTED.equals(state)) {

// 可读可写

mExternalStorageAvailable = mExternalStorageWriteable = true;

} else if (Environment.MEDIA\_MOUNTED\_READ\_ONLY.equals(state)) {

// 可读

} else {

// 可能有很多其他的状态，但是我们只需要知道，不能读也不能写

}

```

###3.2.3　访问外部存储器中的文件

\*\*１、如果 API 版本大于或等于８\*\*，使用

> getExternalFilesDir (String type)

　　该方法打开一个外存储目录，此方法需要一个类型，指定你想要的子目录，如类型参数DIRECTORY\_MUSIC和 DIRECTORY\_RINGTONES（传null就是你应用程序的文件目录的根目录）。通过指定目录的类型，确保Android的媒体扫描仪将扫描分类系统中的文件（例如，铃声被确定为铃声）。如果用户卸载应用程序，这个目录及其所有内容将被删除。

例如：

```

File file = new File(getExternalFilesDir(null), "fanrunqi.jpg");

```

\*\*２、如果API 版本小于 8\*\* （7或者更低）

> getExternalStorageDirectory ()

通过该方法打开外存储的根目录，你应该在以下目录下写入你的应用数据，这样当卸载应用程序时该目录及其所有内容也将被删除。

```

/Android/data/<package\_name>/files/

```

读写数据：

```

if(Environment.getExternalStorageState().equals(Environment.MEDIA\_MOUNTED)){

File sdCardDir = Environment.getExternalStorageDirectory();//获取SDCard目录 "/sdcard"

File saveFile = new File(sdCardDir,"a.txt");

//写数据

try {

FileOutputStream fos= new FileOutputStream(saveFile);

fos.write("fanrunqi".getBytes());

fos.close();

} catch (Exception e) {

e.printStackTrace();

}

//读数据

try {

FileInputStream fis= new FileInputStream(saveFile);

int len =0;

byte[] buf = new byte[1024];

StringBuffer sb = new StringBuffer();

while((len=fis.read(buf))!=-1){

sb.append(new String(buf, 0, len));

}

fis.close();

} catch (Exception e) {

e.printStackTrace();

}

}

```

　　我们也可以在　/Android/data/package\_name/cache/目录下做外部缓存。

部分翻译于：[android-data-storage](http://www.android-doc.com/guide/topics/data/data-storage.html)

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

４、 网络存储数据

</font>

</td>

</tr>

</table>

##

##HttpUrlConnection

　　HttpUrlConnection是Java.net包中提供的API，我们知道Android SDK是基于Java的，所以当然优先考虑HttpUrlConnection这种最原始最基本的API，其实大多数开源的联网框架基本上也是基于JDK的HttpUrlConnection进行的封装罢了，掌握HttpUrlConnection需要以下几个步骤：

1、将访问的路径转换成URL。

> URL url = new URL(path);

2、通过URL获取连接。

> HttpURLConnection conn = (HttpURLConnection) url.openConnection();

3、设置请求方式。

> conn.setRequestMethod(GET);

4、设置连接超时时间。

>conn.setConnectTimeout(5000);

5、设置请求头的信息。

> conn.setRequestProperty(User-Agent, Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0));

7、针对不同的响应码，做不同的操作（请求码200，表明请求成功，获取返回内容的输入流）

工具类：

```

public class StreamTools {

/\*\*

\* 将输入流转换成字符串

\*

\* @param is

\* 从网络获取的输入流

\* @return

\*/

public static String streamToString(InputStream is) {

try {

ByteArrayOutputStream baos = new ByteArrayOutputStream();

byte[] buffer = new byte[1024];

int len = 0;

while ((len = is.read(buffer)) != -1) {

baos.write(buffer, 0, len);

}

baos.close();

is.close();

byte[] byteArray = baos.toByteArray();

return new String(byteArray);

} catch (Exception e) {

Log.e(tag, e.toString());

return null;

}

}

}

```

###HttpUrlConnection发送GET请求

```

public static String loginByGet(String username, String password) {

String path = http://192.168.0.107:8080/WebTest/LoginServerlet?username= + username + &password= + password;

try {

URL url = new URL(path);

HttpURLConnection conn = (HttpURLConnection) url.openConnection();

conn.setConnectTimeout(5000);

conn.setRequestMethod(GET);

int code = conn.getResponseCode();

if (code == 200) {

InputStream is = conn.getInputStream(); // 字节流转换成字符串

return StreamTools.streamToString(is);

} else {

return 网络访问失败;

}

} catch (Exception e) {

e.printStackTrace();

return 网络访问失败;

}

}

```

###HttpUrlConnection发送POST请求

```

public static String loginByPost(String username, String password) {

String path = http://192.168.0.107:8080/WebTest/LoginServerlet;

try {

URL url = new URL(path);

HttpURLConnection conn = (HttpURLConnection) url.openConnection();

conn.setConnectTimeout(5000);

conn.setRequestMethod(POST);

conn.setRequestProperty(Content-Type, application/x-www-form-urlencoded);

String data = username= + username + &password= + password;

conn.setRequestProperty(Content-Length, data.length() + );

// POST方式，其实就是浏览器把数据写给服务器

conn.setDoOutput(true); // 设置可输出流

OutputStream os = conn.getOutputStream(); // 获取输出流

os.write(data.getBytes()); // 将数据写给服务器

int code = conn.getResponseCode();

if (code == 200) {

InputStream is = conn.getInputStream();

return StreamTools.streamToString(is);

} else {

return 网络访问失败;

}

} catch (Exception e) {

e.printStackTrace();

return 网络访问失败;

}

}

```

##HttpClient

　　HttpClient是开源组织Apache提供的Java请求网络框架，其最早是为了方便Java服务器开发而诞生的，是对JDK中的HttpUrlConnection各API进行了封装和简化，提高了性能并且降低了调用API的繁琐，Android因此也引进了这个联网框架，我们再不需要导入任何jar或者类库就可以直接使用，值得注意的是Android官方已经宣布不建议使用HttpClient了。

###HttpClient发送GET请求

1、 创建HttpClient对象

2、创建HttpGet对象，指定请求地址（带参数）

3、使用HttpClient的execute(),方法执行HttpGet请求，得到HttpResponse对象

4、调用HttpResponse的getStatusLine().getStatusCode()方法得到响应码

5、调用的HttpResponse的getEntity().getContent()得到输入流，获取服务端写回的数据

```

public static String loginByHttpClientGet(String username, String password) {

String path = http://192.168.0.107:8080/WebTest/LoginServerlet?username=

+ username + &password= + password;

HttpClient client = new DefaultHttpClient(); // 开启网络访问客户端

HttpGet httpGet = new HttpGet(path); // 包装一个GET请求

try {

HttpResponse response = client.execute(httpGet); // 客户端执行请求

int code = response.getStatusLine().getStatusCode(); // 获取响应码

if (code == 200) {

InputStream is = response.getEntity().getContent(); // 获取实体内容

String result = StreamTools.streamToString(is); // 字节流转字符串

return result;

} else {

return 网络访问失败;

}

} catch (Exception e) {

e.printStackTrace();

return 网络访问失败;

}

}

```

###HttpClient发送POST请求

1，创建HttpClient对象

2，创建HttpPost对象，指定请求地址

3，创建List，用来装载参数

4，调用HttpPost对象的setEntity()方法，装入一个UrlEncodedFormEntity对象，携带之前封装好的参数

5，使用HttpClient的execute()方法执行HttpPost请求，得到HttpResponse对象

6， 调用HttpResponse的getStatusLine().getStatusCode()方法得到响应码

7， 调用的HttpResponse的getEntity().getContent()得到输入流，获取服务端写回的数据

```

public static String loginByHttpClientPOST(String username, String password) {

String path = http://192.168.0.107:8080/WebTest/LoginServerlet;

try {

HttpClient client = new DefaultHttpClient(); // 建立一个客户端

HttpPost httpPost = new HttpPost(path); // 包装POST请求

// 设置发送的实体参数

List parameters = new ArrayList();

parameters.add(new BasicNameValuePair(username, username));

parameters.add(new BasicNameValuePair(password, password));

httpPost.setEntity(new UrlEncodedFormEntity(parameters, UTF-8));

HttpResponse response = client.execute(httpPost); // 执行POST请求

int code = response.getStatusLine().getStatusCode();

if (code == 200) {

InputStream is = response.getEntity().getContent();

String result = StreamTools.streamToString(is);

return result;

} else {

return 网络访问失败;

}

} catch (Exception e) {

e.printStackTrace();

return 访问网络失败;

}

}

```

参考：

　　[Android开发请求网络方式详解](http://www.2cto.com/kf/201501/368943.html)

##Android提供的其他网络访问框架

　　HttpClient和HttpUrlConnection的两种网络访问方式编写网络代码，需要自己考虑很多，获取数据或许可以，但是如果要将手机本地数据上传至网络，根据不同的web端接口，需要组织不同的数据内容上传，给手机端造成了很大的工作量。

　　目前有几种快捷的网络开发开源框架，给我们提供了非常大的便利。下面是这些项目Github地址，有文档和Api说明。

[android-async-http](https://github.com/loopj/android-async-http)

[http-request](https://github.com/kevinsawicki/http-request)

[okhttp](https://github.com/square/okhttp)

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

５、 SQLite数据库存储数据

</font>

</td>

</tr>

</table>

##

　　前面的文章[ SQLite的使用入门](http://blog.csdn.net/amazing7/article/details/51375012)已经做了详细说明，这里就不在多说了。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

６、 使用ContentProvider存储数据

</font>

</td>

</tr>

</table>

##

　　同样可以查看　[ContentProvider实例详解](http://blog.csdn.net/amazing7/article/details/51324022)

## 第13节 Android 异步消息处理机制源码解析

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

1、Handler的由来

</font>

</td>

</tr>

</table>

##

　　当程序第一次启动的时候，Android会同时启动一条主线程（ Main Thread）来负责处理与UI相关的事件，我们叫做UI线程。

　　Android的UI操作并不是线程安全的（出于性能优化考虑），意味着如果多个线程并发操作UI线程，可能导致线程安全问题。

　　为了解决Android应用多线程问题—Android平台只允许UI线程修改Activity里的UI组建，就会导致新启动的线程无法改变界面组建的属性值。

　　\*\*简单的说：\*\*当主线程队列处理一个消息超过5秒,android 就会抛出一个 ANP(无响应)的异常,所以,我们需要把一些要处理比较长的消息,放在一个单独线程里面处理,把处理以后的结果,返回给主线程运行,就需要用的Handler来进行线程建的通信。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

２、Handler的作用

</font>

</td>

</tr>

</table>

##

２.１　让线程延时执行

主要用到的两个方法：

- final boolean postAtTime(Runnable r, long uptimeMillis)

- final boolean postDelayed(Runnable r, long delayMillis)

２.２　让任务在其他线程中执行并返回结果

分为两个步骤：

- 在新启动的线程中发送消息

　　使用Handler对象的sendMessage()方法或者SendEmptyMessage()方法发送消息。

- 在主线程中获取处理消息

　　重写Handler类中处理消息的方法（void handleMessage(Message msg)），当新启动的线程发送消息时，消息发送到与之关联的MessageQueue。而Hanlder不断地从MessageQueue中获取并处理消息。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

３、Handler更新UI线程一般使用

</font>

</td>

</tr>

</table>

##

- 首先要进行Handler 申明，复写handleMessage方法( 放在主线程中)

```

private Handler handler = new Handler() {

@Override

public void handleMessage(Message msg) {

// TODO 接收消息并且去更新UI线程上的控件内容

if (msg.what == UPDATE) {

// 更新界面上的textview

tv.setText(String.valueOf(msg.obj));

}

super.handleMessage(msg);

}

};

```

- 子线程发送Message给ui线程表示自己任务已经执行完成，主线程可以做相应的操作了。

```

new Thread() {

@Override

public void run() {

// TODO 子线程中通过handler发送消息给handler接收，由handler去更新TextView的值

try {

//do something

Message msg = new Message();

msg.what = UPDATE;

msg.obj = "更新后的值" ;

handler.sendMessage(msg);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}.start();

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

4、Handler原理分析

</font>

</td>

</tr>

</table>

##

4.1 　Handler的构造函数

> ①　public　Handler()

②　public　Handler(Callbackcallback)

③　public　Handler(Looperlooper)

④　public　Handler(Looperlooper, Callbackcallback)

- 第①个和第②个构造函数都没有传递Looper，这两个构造函数都将通过调用Looper.myLooper()获取当前线程绑定的Looper对象，然后将该Looper对象保存到名为mLooper的成员字段中。

　　下面来看①②个函数源码：

```

113 public Handler() {

114 this(null, false);

115 }

127 public Handler(Callback callback) {

128 this(callback, false);

129 }

//他们会调用Handler的内部构造方法

188 public Handler(Callback callback, boolean async) {

189 if (FIND\_POTENTIAL\_LEAKS) {

190 final Class<? extends Handler> klass = getClass();

191 if ((klass.isAnonymousClass() ||klass.isMemberClass()

|| klass.isLocalClass()) &&

192 (klass.getModifiers() & Modifier.STATIC) == 0) {

193 Log.w(TAG, "The following Handler class should be static or leaks might occur: " +

194 klass.getCanonicalName());

195 }

196 }

197/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

198 mLooper = Looper.myLooper();

199 if (mLooper == null) {

200 throw new RuntimeException(

201 "Can't create handler inside thread that has not called Looper.prepare()");

202 }

203 mQueue = mLooper.mQueue;

204 mCallback = callback;

205 mAsynchronous = async;

206 }

```

　　我们看到暗红色的重点部分：

　　通过Looper.myLooper()获取了当前线程保存的Looper实例，又通过这个Looper实例获取了其中保存的MessageQueue（消息队列）。\*\*每个Handler 对应一个Looper对象，产生一个MessageQueue\*\*

- 第③个和第④个构造函数传递了Looper对象，这两个构造函数会将该Looper保存到名为mLooper的成员字段中。

　　下面来看③④个函数源码：

```

136 public Handler(Looper looper) {

137 this(looper, null, false);

138 }

147 public Handler(Looper looper, Callback callback) {

148 this(looper, callback, false);

149 }

//他们会调用Handler的内部构造方法

227 public Handler(Looper looper, Callback callback, boolean async) {

228 mLooper = looper;

229 mQueue = looper.mQueue;

230 mCallback = callback;

231 mAsynchronous = async;

232 }

```

- 第②个和第④个构造函数还传递了Callback对象，Callback是Handler中的内部接口，需要实现其内部的handleMessage方法，Callback代码如下:

```

80 public interface Callback {

81 public boolean More ...handleMessage(Message msg);

82 }

```

　　Handler.Callback是用来处理Message的一种手段，如果没有传递该参数，那么就应该重写Handler的handleMessage方法，也就是说为了使得Handler能够处理Message，我们有两种办法：

　1. 向Hanlder的构造函数传入一个Handler.Callback对象，并实现Handler.Callback的handleMessage方法

　2. 无需向Hanlder的构造函数传入Handler.Callback对象，但是需要重写Handler本身的handleMessage方法

　　　也就是说无论哪种方式，我们都得通过某种方式实现handleMessage方法，这点与Java中对Thread的设计有异曲同工之处。

4.2　Handle发送消息的几个方法源码

```

public final boolean sendMessage(Message msg)

{

return sendMessageDelayed(msg, 0);

}

```

```

public final boolean sendEmptyMessageDelayed(int what, long delayMillis) {

Message msg = Message.obtain();

msg.what = what;

return sendMessageDelayed(msg, delayMillis);

}

```

```

public final boolean sendMessageDelayed(Message msg, long delayMillis)

{

if (delayMillis < 0) {

delayMillis = 0;

}

return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);

}

```

```

public boolean sendMessageAtTime(Message msg, long uptimeMillis) {

MessageQueue queue = mQueue;

if (queue == null) {

RuntimeException e = new RuntimeException(

this + " sendMessageAtTime() called with no mQueue");

Log.w("Looper", e.getMessage(), e);

return false;

}

return enqueueMessage(queue, msg, uptimeMillis);

}

```

我们可以看出他们最后都调用了sendMessageAtTime（），然后返回了enqueueMessage方法，下面看一下此方法源码：

```

626 private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {

　　　　　　//把当前的handler作为msg的target属性

627 msg.target = this;

628 if (mAsynchronous) {

629 msg.setAsynchronous(true);

630 }

631 return queue.enqueueMessage(msg, uptimeMillis);

632 }

```

在该方法中有两件事需要注意：

1. msg.target = this

　　该代码将Message的target绑定为当前的Handler

2. queue.enqueueMessage

　　变量queue表示的是Handler所绑定的消息队列MessageQueue，通过调用queue.enqueueMessage(msg, uptimeMillis)我们将Message放入到消息队列中。

过下图可以看到完整的方法调用顺序：

![这里写图片描述](http://img.blog.csdn.net/20160516125245477)

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

５、Looper原理分析

</font>

</td>

</tr>

</table>

##

　　我们一般在主线程申明Handler，有时我们需要继承Thread类实现自己的线程功能，当我们在里面申明Handler的时候会报错。其原因是主线程中已经实现了两个重要的Looper方法，下面看一看ActivityThread.java中main方法的源码：

```

public static void main(String[] args) {

//......省略

5205 Looper.prepareMainLooper();//>

5206

5207 ActivityThread thread = new ActivityThread();

5208 thread.attach(false);

5209

5210 if (sMainThreadHandler == null) {

5211 sMainThreadHandler = thread.getHandler();

5212 }

5213

5214 AsyncTask.init();

5215

5216 if (false) {

5217 Looper.myLooper().setMessageLogging(new

5218 LogPrinter(Log.DEBUG, "ActivityThread"));

5219 }

5220

5221 Looper.loop();//>

5222

5223 throw new RuntimeException("Main thread loop unexpectedly exited");

5224 }

5225}

```

5.1　首先看prepare()方法

```

70 public static void prepare() {

71 prepare(true);

72 }

73

74 private static void prepare(boolean quitAllowed) {

　　　　　//证了一个线程中只有一个Looper实例

75 if (sThreadLocal.get() != null) {

76 throw new RuntimeException("Only one Looper may be created per thread");

77 }

78 sThreadLocal.set(new Looper(quitAllowed));

79 }

```

该方法会调用Looper构造函数同时实例化出MessageQueue和当前thread.

```

186 private Looper(boolean quitAllowed) {

187 mQueue = new MessageQueue(quitAllowed);

188 mThread = Thread.currentThread();

189 }

182 public static MessageQueue myQueue() {

183 return myLooper().mQueue;

184 }

```

　　prepare()方法中通过ThreadLocal对象实现Looper实例与线程的绑定。（不清楚的可以查看　[ThreadLocal的使用规则和源码分析](http://blog.csdn.net/amazing7/article/details/51313851)）

５.2 　loop()方法

```

109 public static void loop() {

110 final Looper me = myLooper();

111 if (me == null) {

112 throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");

113 }

114 final MessageQueue queue = me.mQueue;

115

118 Binder.clearCallingIdentity();

119 final long ident = Binder.clearCallingIdentity();

120

121 for (;;) {

122 Message msg = queue.next(); // might block

123 if (msg == null) {

124

125 return;

126 }

127

129 Printer logging = me.mLogging;

130 if (logging != null) {

131 logging.println(">>>>> Dispatching to " + msg.target + " " +

132 msg.callback + ": " + msg.what);

133 }

//重点\*\*\*\*

135 msg.target.dispatchMessage(msg);

136

137 if (logging != null) {

138 logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);

139 }

140

142 // identity of the thread wasn't corrupted.

143 final long newIdent = Binder.clearCallingIdentity();

144 if (ident != newIdent) {

145 Log.wtf(TAG, "Thread identity changed from 0x"

146 + Long.toHexString(ident) + " to 0x"

147 + Long.toHexString(newIdent) + " while dispatching to "

148 + msg.target.getClass().getName() + " "

149 + msg.callback + " what=" + msg.what);

150 }

151

152 msg.recycleUnchecked();

153 }

154 }

```

　　首先looper对象不能为空，就是说loop()方法调用必须在prepare()方法的后面。

　Looper一直在不断的从消息队列中通过MessageQueue的next方法获取Message，然后通过代码msg.target.dispatchMessage(msg)让该msg所绑定的Handler（Message.target）执行dispatchMessage方法以实现对Message的处理。

Handler的dispatchMessage的源码如下：

```

93 public void dispatchMessage(Message msg) {

94 if (msg.callback != null) {

95 handleCallback(msg);

96 } else {

97 if (mCallback != null) {

98 if (mCallback.handleMessage(msg)) {

99 return;

100 }

101 }

102 handleMessage(msg);

103 }

104 }

```

　　我们可以看到Handler提供了三种途径处理Message，而且处理有前后优先级之分：首先尝试让postXXX中传递的Runnable执行，其次尝试让Handler构造函数中传入的Callback的handleMessage方法处理，最后才是让Handler自身的handleMessage方法处理Message。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

６、如何在子线程中使用Handler

</font>

</td>

</tr>

</table>

##

　　Handler本质是从当前的线程中获取到Looper来监听和操作MessageQueue，当其他线程执行完成后回调当前线程。

　　子线程需要先prepare（）才能获取到Looper的，是因为在子线程只是一个普通的线程，其ThreadLoacl中没有设置过Looper，所以会抛出异常，而在Looper的prepare（）方法中sThreadLocal.set(new Looper())是设置了Looper的。

６.1　实例代码

　定义一个类实现Runnable接口或继承Thread类（一般不继承）。

```

class Rub implements Runnable {

public Handler myHandler;

// 实现Runnable接口的线程体

@Override

public void run() {

/\*①、调用Looper的prepare()方法为当前线程创建Looper对象并，

创建Looper对象时，它的构造器会自动的创建相对应的MessageQueue\*/

Looper.prepare();

/\*.②、创建Handler子类的实例，重写HandleMessage()方法，该方法处理除当前线程以外线程的消息\*/

myHandler = new Handler() {

@Override

public void handleMessage(Message msg) {

String ms = "";

if (msg.what == 0x777) {

}

}

};

//③、调用Looper的loop()方法来启动Looper让消息队列转动起来

Looper.loop();

}

}

```

注意分成三步：

１．调用Looper的prepare()方法为当前线程创建Looper对象，创建Looper对象时，它的构造器会创建与之配套的MessageQueue。

２．有了Looper之后，创建Handler子类实例，重写HanderMessage()方法，该方法负责处理来自于其他线程的消息。

３．调用Looper的looper()方法启动Looper。

　　然后使用这个handler实例在任何其他线程中发送消息，最终处理消息的代码都会在你创建Handler实例的线程中运行。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

７、总结

</font>

</td>

</tr>

</table>

##

\*\*Handler\*\*：

　　　　　　发送消息，它能把消息发送给Looper管理的MessageQueue。

　　　　　　处理消息，并负责处理Looper分给它的消息。

\*\*Message\*\*：

　　　　　　Handler接收和处理的消息对象。

\*\*Looper\*\*：

　　　　　　每个线程只有一个Looper，它负责管理对应的MessageQueue，会不断地从MessageQueue取出消息，并将消息分给对应的Hanlder处理。

　　　　　　主线程中，系统已经初始化了一个Looper对象，因此可以直接创建Handler即可，就可以通过Handler来发送消息、处理消息。 程序自己启动的子线程，程序必须自己创建一个Looper对象，并启动它，调用Looper.prepare()方法。

prapare()方法：保证每个线程最多只有一个Looper对象。

looper()方法：启动Looper，使用一个死循环不断取出MessageQueue中的消息，并将取出的消息分给对应的Handler进行处理。

MessageQueue：由Looper负责管理，它采用先进先出的方式来管理Message。

　　Handler的构造方法，会首先得到当前线程中保存的Looper实例，进而与Looper实例中的MessageQueue想关联。

　　Handler的sendMessage方法，会给msg的target赋值为handler自身，然后加入MessageQueue中。

## 第14节 Android View事件分发机制源码分析

> 在android开发中会经常遇到滑动冲突（比如ScrollView或是SliddingMenu与ListView的嵌套）的问题，需要我们深入的了解android事件响应机制才能解决，事件响应机制已经是android开发者必不可少的知识。

# 1.涉及到事件响应的常用方法构成

　　用户在手指与屏幕接触过程中通过MotionEvent对象产生一系列事件，它有四种状态：

- MotionEvent.ACTION\_DOWN　：手指按下屏幕的瞬间（一切事件的开始）

- MotionEvent.ACTION\_MOVE　：手指在屏幕上移动

- MotionEvent.ACTION\_UP　：手指离开屏幕瞬间

- MotionEvent.ACTION\_CANCEL 　：取消手势，一般由程序产生，不会由用户产生

　　Android中的事件onClick, onLongClick，onScroll, onFling等等，都是由许多个Touch事件构成的（一个ACTION\_DOWN， n个ACTION\_MOVE，1个ACTION\_UP）。

　　android 事件响应机制是先 \*\*分发\*\*（先由外部的View接收，然后依次传递给其内层的最小View）再 \*\*处理\*\* （从最小View单元（事件源）开始依次向外层传递。）的形式实现的。

　　复杂性表现在：可以控制每层事件是否继续传递（分发和拦截协同实现），以及事件的具体消费（事件分发也具有事件消费能力）。

# ２.android事件处理涉及到的三个重要函数

> \*\*事件分发：public boolean dispatchTouchEvent(MotionEvent ev)\*\*

　　　当有监听到事件时，首先由Activity进行捕获，进入事件分发处理流程。（因为activity没有事件拦截，View和ViewGroup有）会将事件传递给最外层View的dispatchTouchEvent(MotionEvent ev)方法，该方法对事件进行分发。

- return true ：表示该View内部消化掉了所有事件。

- return false ：事件在本层不再继续进行分发，并交由\*\*上层\*\*控件的onTouchEvent方法进行消费（如果本层控件已经是Activity，那么事件将被系统消费或处理）。

- 如果事件分发返回系统默认的 super.dispatchTouchEvent(ev)，事件将分发给本层的事件拦截onInterceptTouchEvent 方法进行处理

> \*\*事件拦截：public boolean onInterceptTouchEvent(MotionEvent ev)\*\*

- return true ：表示将事件进行拦截，并将拦截到的事件交由本层控件 的 onTouchEvent 进行处理；

- return false ：则表示不对事件进行拦截，事件得以成功分发到子View。并由子View的dispatchTouchEvent进行处理。

- 如果返回super.onInterceptTouchEvent(ev)，默认表示拦截该事件，并将事件传递给当前View的onTouchEvent方法，和return true一样。

> \*\*事件响应：public boolean onTouchEvent(MotionEvent ev)\*\*

　　在dispatchTouchEvent（事件分发）返回super.dispatchTouchEvent(ev)并且onInterceptTouchEvent（事件拦截返回true或super.onInterceptTouchEvent(ev)的情况下，那么事件会传递到onTouchEvent方法，该方法对事件进行响应。

- 如果return true，表示onTouchEvent处理完事件后消费了此次事件。此时事件终结；

- 如果return fasle，则表示不响应事件，那么该事件将会不断向上层View的onTouchEvent方法传递，直到某个View的onTouchEvent方法返回true，如果到了最顶层View还是返回false，那么认为该事件不消耗，则在同一个事件系列中，当前View无法再次接收到事件，该事件会交由Activity的onTouchEvent进行处理；

- 如果return super.dispatchTouchEvent(ev)，则表示不响应事件，结果与return false一样。

> 从以上过程中可以看出，dispatchTouchEvent无论返回true还是false，事件都不再进行分发，只有当其返回super.dispatchTouchEvent(ev)，才表明其具有向下层分发的愿望，但是是否能够分发成功，则需要经过事件拦截onInterceptTouchEvent的审核。事件是否向上传递处理是由onTouchEvent的返回值决定的。

![这里写图片描述](http://img.blog.csdn.net/20160428161104339)

（图来自网络）

# ３.View源码分析

　　Android中ImageView、textView、Button等继承于View但没有重写的dispatchTouchEvent方法，所以都用的View的该方法进行事件分发。

　　看View重要函数部分源码：

```

public boolean dispatchTouchEvent(MotionEvent event) {

//返回true,表示该View内部消化掉了所有事件。返回false，表示View内部只处理了ACTION\_DOWN事件，事件继续传递，向上级View(ViewGroup)传递。

if (mOnTouchListener != null && (mViewFlags & ENABLED\_MASK) == ENABLED &&

mOnTouchListener.onTouch(this, event)) {

//此处的onTouch方式就是回调的我们注册OnTouchListener时重写的onTouch()方法

return true;

}

return onTouchEvent(event);

}

```

　首先进行三个条件的判断：

（1）查看是否给button设置了OnTouchListener()事件；

（2）控件是否Enable；（控件默认都是enable的）

（3）button里面实现的OnTouchListener监听里的onTouch()方法是否返回true；

　如果条件都满足，则该事件被消耗掉，不再进入onTouchEvent中处理。否则将事件将交给onTouchEvent方法处理。

```

public boolean onTouchEvent(MotionEvent event) {

...

/＊ 当前onTouch的组件必须是可点击的比如Button，ImageButton等等，此处CLICKABLE为true，才会进入if方法，最后返回true。

如果是ImageView、TexitView这些默认为不可点击的View,此处CLICKABLE为false，最后返回false。当然会有特殊情况，如果给这些View设置了onClick监听器，此处CLICKABLE也将为true　　＊／

if (((viewFlags & CLICKABLE) == CLICKABLE ||

(viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE)) {

switch (event.getAction()) {

case MotionEvent.ACTION\_UP:

...

if (!post(mPerformClick)) {

performClick();// 实际就是回调了我们注册的OnClickListener中重新的onClick()方法

}

...

break;

case MotionEvent.ACTION\_DOWN:

...

break;

case MotionEvent.ACTION\_CANCEL:

...

break;

case MotionEvent.ACTION\_MOVE:

...

break;

}

return true;

}

return false;

}

```

```

public boolean performClick() {

...

／／

if (li != null && li.mOnClickListener != null) {

...

li.mOnClickListener.onClick(this);

return true;

}

return false;

}

```

```

public void setOnClickListener(OnClickListener l) {

if (!isClickable()) {

setClickable(true);

}

getListenerInfo().mOnClickListener = l;

}

```

> 只有我们注册OnTouchListener时重写的

> onTouch()方法中

>

> 返回false —> 执行onTouchEvent方法 —> 导致onClick()回调方法执行

>

返回true —> onTouchEvent方法不执行 —> 导致onClick()回调方法不会执行

# ４.ViewGroup源码分析

　　Android中诸如LinearLayout等的五大布局控件，都是继承自ViewGroup，而ViewGroup本身是继承自View，所以ViewGroup的事件处理机制对这些控件都有效。

部分源码：

```

public boolean dispatchTouchEvent(MotionEvent ev) {

final int action = ev.getAction();

final float xf = ev.getX();

final float yf = ev.getY();

final float scrolledXFloat = xf + mScrollX;

final float scrolledYFloat = yf + mScrollY;

final Rect frame = mTempRect;

//这个值默认是false, 然后我们可以通过requestDisallowInterceptTouchEvent(boolean disallowIntercept)方法

//来改变disallowIntercept的值

boolean disallowIntercept = (mGroupFlags & FLAG\_DISALLOW\_INTERCEPT) != 0;

//这里是ACTION\_DOWN的处理逻辑

if (action == MotionEvent.ACTION\_DOWN) {

//清除mMotionTarget, 每次ACTION\_DOWN都很设置mMotionTarget为null

if (mMotionTarget != null) {

mMotionTarget = null;

}

//disallowIntercept默认是false, 就看ViewGroup的onInterceptTouchEvent()方法

if (disallowIntercept || !onInterceptTouchEvent(ev)) { //第一点

ev.setAction(MotionEvent.ACTION\_DOWN);

final int scrolledXInt = (int) scrolledXFloat;

final int scrolledYInt = (int) scrolledYFloat;

final View[] children = mChildren;

final int count = mChildrenCount;

//遍历其子View

for (int i = count - 1; i >= 0; i--) { //第二点

final View child = children[i];

//如果该子View是VISIBLE或者该子View正在执行动画, 表示该View才

//可以接受到Touch事件

if ((child.mViewFlags & VISIBILITY\_MASK) == VISIBLE

|| child.getAnimation() != null) {

//获取子View的位置范围

child.getHitRect(frame);

//如Touch到屏幕上的点在该子View上面

if (frame.contains(scrolledXInt, scrolledYInt)) {

// offset the event to the view's coordinate system

final float xc = scrolledXFloat - child.mLeft;

final float yc = scrolledYFloat - child.mTop;

ev.setLocation(xc, yc);

child.mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;

//调用该子View的dispatchTouchEvent()方法

if (child.dispatchTouchEvent(ev)) {

// 如果child.dispatchTouchEvent(ev)返回true表示

//该事件被消费了，设置mMotionTarget为该子View

mMotionTarget = child;

//直接返回true

return true;

}

// The event didn't get handled, try the next view.

// Don't reset the event's location, it's not

// necessary here.

}

}

}

}

}

//判断是否为ACTION\_UP或者ACTION\_CANCEL

boolean isUpOrCancel = (action == MotionEvent.ACTION\_UP) ||

(action == MotionEvent.ACTION\_CANCEL);

if (isUpOrCancel) {

//如果是ACTION\_UP或者ACTION\_CANCEL, 将disallowIntercept设置为默认的false

//假如我们调用了requestDisallowInterceptTouchEvent()方法来设置disallowIntercept为true

//当我们抬起手指或者取消Touch事件的时候要将disallowIntercept重置为false

//所以说上面的disallowIntercept默认在我们每次ACTION\_DOWN的时候都是false

mGroupFlags &= ~FLAG\_DISALLOW\_INTERCEPT;

}

// The event wasn't an ACTION\_DOWN, dispatch it to our target if

// we have one.

final View target = mMotionTarget;

//mMotionTarget为null意味着没有找到消费Touch事件的View, 所以我们需要调用ViewGroup父类的

//dispatchTouchEvent()方法，也就是View的dispatchTouchEvent()方法

if (target == null) {

// We don't have a target, this means we're handling the

// event as a regular view.

ev.setLocation(xf, yf);

if ((mPrivateFlags & CANCEL\_NEXT\_UP\_EVENT) != 0) {

ev.setAction(MotionEvent.ACTION\_CANCEL);

mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;

}

return super.dispatchTouchEvent(ev);

}

//这个if里面的代码ACTION\_DOWN不会执行，只有ACTION\_MOVE

//ACTION\_UP才会走到这里, 假如在ACTION\_MOVE或者ACTION\_UP拦截的

//Touch事件, 将ACTION\_CANCEL派发给target，然后直接返回true

//表示消费了此Touch事件

if (!disallowIntercept && onInterceptTouchEvent(ev)) {

final float xc = scrolledXFloat - (float) target.mLeft;

final float yc = scrolledYFloat - (float) target.mTop;

mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;

ev.setAction(MotionEvent.ACTION\_CANCEL);

ev.setLocation(xc, yc);

if (!target.dispatchTouchEvent(ev)) {

}

// clear the target

mMotionTarget = null;

// Don't dispatch this event to our own view, because we already

// saw it when intercepting; we just want to give the following

// event to the normal onTouchEvent().

return true;

}

if (isUpOrCancel) {

mMotionTarget = null;

}

// finally offset the event to the target's coordinate system and

// dispatch the event.

final float xc = scrolledXFloat - (float) target.mLeft;

final float yc = scrolledYFloat - (float) target.mTop;

ev.setLocation(xc, yc);

if ((target.mPrivateFlags & CANCEL\_NEXT\_UP\_EVENT) != 0) {

ev.setAction(MotionEvent.ACTION\_CANCEL);

target.mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;

mMotionTarget = null;

}

//如果没有拦截ACTION\_MOVE, ACTION\_DOWN的话，直接将Touch事件派发给target

return target.dispatchTouchEvent(ev);

}

```

> 1、dispatchTouchEvent作用：决定事件是否由onInterceptTouchEvent来拦截处理。

返回super.dispatchTouchEvent时，由onInterceptTouchEvent来决定事件的流向

返回false时，会继续分发事件，自己内部只处理了ACTION\_DOWN

返回true时，不会继续分发事件，自己内部处理了所有事件（ACTION\_DOWN,ACTION\_MOVE,ACTION\_UP）

> 2、onInterceptTouchEvent作用：拦截事件，用来决定事件是否传向子View

返回true时，拦截后交给自己的onTouchEvent处理

返回false时，拦截后交给子View来处理

> 3、onTouchEvent作用：事件最终到达这个方法

返回true时，内部处理所有的事件，换句话说，后续事件将继续传递给该view的onTouchEvent()处理

返回false时，事件会向上传递，由onToucEvent来接受，如果最上面View中的onTouchEvent也返回false的话，那么事件就会消失

# ５.总结

- 如果ViewGroup找到了能够处理该事件的View，则直接交给子View处理，自己的onTouchEvent不会被触发；

- 可以通过复写onInterceptTouchEvent(ev)方法，拦截子View的事件（即return true），把事件交给自己处理，则会执行自己对应的onTouchEvent方法。

- 子View可以通过调用getParent().requestDisallowInterceptTouchEvent(true); 阻止ViewGroup对其MOVE或者UP事件进行拦截；

- 一个点击事件产生后，它的传递过程如下：

Activity->Window->View。顶级View接收到事件之后，就会按相应规则去分发事件。如果一个View的onTouchEvent方法返回false，那么将会交给父容器的onTouchEvent方法进行处理，逐级往上，如果所有的View都不处理该事件，则交由Activity的onTouchEvent进行处理。

- 如果某一个View开始处理事件，如果他不消耗ACTION\_DOWN事件（也就是onTouchEvent返回false），则同一事件序列比如接下来进行ACTION\_MOVE，则不会再交给该View处理。

- ViewGroup默认不拦截任何事件。

- 诸如TextView、ImageView这些不作为容器的View，一旦接受到事件，就调用onTouchEvent方法，它们本身没有onInterceptTouchEvent方法。正常情况下，它们都会消耗事件（返回true），除非它们是不可点击的（clickable和longClickable都为false），那么就会交由父容器的onTouchEvent处理。

- 点击事件分发过程如下 dispatchTouchEvent—->OnTouchListener的onTouch方法—->onTouchEvent-->OnClickListener的onClick方法。也就是说，我们平时调用的setOnClickListener，优先级是最低的，所以，onTouchEvent或OnTouchListener的onTouch方法如果返回true，则不响应onClick方法...

## 第15节 Android SQLite的使用入门

#概述

　　Android 也提供了几种方法用来保存数据，使得这些数据即使在程序结束以后依然不会丢失。这些方法有：

- 文本文件：

　　可以保存在应用程序自己的目录下，安装的每个app都会在/data/data/目录下创建个文件夹，名字和应用程序中AndroidManifest.xml文件中的package一样。

- SDcard保存：

- Preferences保存：

　　这也是一种经常使用的数据存储方法，因为它们对于用户而言是透明的，并且从应用安装的时候就存在了。

- Assets保存：

　　用来存储一些只读数据，Assets是指那些在assets目录下的文件，这些文件在你将你的应用编译打包之前就要存在，并且可以在应用程序运行的时候被访问到。

　　但有时候我们需要对保存的数据进行一些复杂的操作，或者数据量很大，超出了文本文件和Preference的性能能的范围，所以需要一些更加高效的方法来管理，从Android1.5开始，Android就自带SQLite数据库了。

　　SQLite它是一个独立的，无需服务进程，支持事务处理，可以使用SQL语言的数据库。

#SQLite的特性

1、 ACID事务

>ACID：

> 　　指数据库事务正确执行的四个基本要素的缩写。包含：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。一个支持事务（Transaction）的数据库，必需要具有这四种特性，否则在事务过程（Transaction processing）当中无法保证数据的正确性，交易过程极可能达不到交易方的要求。

2、 零配置 – 无需安装和管理配置

3、储存在单一磁盘文件中的一个完整的数据库

4、数据库文件可以在不同字节顺序的机器间自由的共享

5、支持数据库大小至2TB

6、 足够小, 大致3万行C代码, 250K

7、比一些流行的数据库在大部分普通数据库操作要快

8、简单, 轻松的API

9、 包含TCL绑定, 同时通过Wrapper支持其他语言的绑定

> http://www.sqlite.org/tclsqlite.html

10、良好注释的源代码, 并且有着90%以上的测试覆盖率

11、 独立: 没有额外依赖

12、 Source完全的Open, 你可以用于任何用途, 包括出售它

13、支持多种开发语言，C，PHP，Perl，Java，ASP.NET，Python

#Android 中使用 SQLite

　　Activites 可以通过 Content Provider 或者 Service 访问一个数据库。

##创建数据库

　　Android 不自动提供数据库。在 Android 应用程序中使用 SQLite，必须自己创建数据库，然后创建表、索引，填充数据。Android 提供了 SQLiteOpenHelper 帮助你创建一个数据库，你只要继承 SQLiteOpenHelper 类根据开发应用程序的需要，封装创建和更新数据库使用的逻辑就行了。

　　SQLiteOpenHelper 的子类，至少需要实现三个方法：

```

public class DatabaseHelper extends SQLiteOpenHelper {

/\*\*

\* @param context 上下文环境（例如，一个 Activity）

\* @param name 数据库名字

\* @param factory 一个可选的游标工厂（通常是 Null）

\* @param version 数据库模型版本的整数

\*

\* 会调用父类 SQLiteOpenHelper的构造函数

\*/

public DatabaseHelper(Context context, String name, CursorFactory factory, int version) {

super(context, name, factory, version);

}

/\*\*

\* 在数据库第一次创建的时候会调用这个方法

\*

\*根据需要对传入的SQLiteDatabase 对象填充表和初始化数据。

\*/

@Override

public void onCreate(SQLiteDatabase db) {

}

/\*\*

\* 当数据库需要修改的时候（两个数据库版本不同），Android系统会主动的调用这个方法。

\* 一般我们在这个方法里边删除数据库表，并建立新的数据库表.

\*/

@Override

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

//三个参数，一个 SQLiteDatabase 对象，一个旧的版本号和一个新的版本号

}

@Override

public void onOpen(SQLiteDatabase db) {

// 每次成功打开数据库后首先被执行

super.onOpen(db);

}

}

```

继承SQLiteOpenHelper之后就拥有了以下两个方法：

- getReadableDatabase() 　创建或者打开一个查询数据库

- getWritableDatabase()　创建或者打开一个可写数据库

```

DatabaseHelper database = new DatabaseHelper(context);//传入一个上下文参数

SQLiteDatabase db = null;

db = database.getWritableDatabase();

```

　　上面这段代码会返回一个 SQLiteDatabase 类的实例，使用这个对象，你就可以查询或者修改数据库。

SQLiteDatabase类为我们提供了很多种方法，而较常用的方法如下：

> (int) delete(String table,String whereClause,String[] whereArgs)

　　删除数据行

> (long) insert(String table,String nullColumnHack,ContentValues values)

　　 添加数据行

> (int) update(String table, ContentValues values, String whereClause, String[] whereArgs)

　　更新数据行

> (void) execSQL(String sql)

　　 执行一个SQL语句，可以是一个select或其他的sql语句

> (void) close()

　　 关闭数据库

> (Cursor) query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)

　　查询指定的数据表返回一个带游标的数据集。

各参数说明：

table：表名称

colums：列名称数组

selection：条件子句，相当于where

selectionArgs：条件语句的参数数组

groupBy：分组

having：分组条件

orderBy：排序类

limit：分页查询的限制

Cursor：返回值，相当于结果集ResultSet

> (Cursor) rawQuery(String sql, String[] selectionArgs)

　　运行一个预置的SQL语句，返回带游标的数据集（与上面的语句最大的区别就是防止SQL注入）

　　当你完成了对数据库的操作（例如你的 Activity 已经关闭），需要调用 SQLiteDatabase 的 Close() 方法来释放掉数据库连接。

##创建表和索引

　　为了创建表和索引，需要调用 SQLiteDatabase 的 execSQL() 方法来执行 DDL 语句。如果没有异常，这个方法没有返回值。

　　例如，你可以执行如下代码：

```

db.execSQL("CREATE TABLE user(\_id INTEGER PRIMARY KEY

AUTOINCREMENT, username TEXT, password TEXT);");

```

　　这条语句会创建一个名为 user的表，表有一个列名为 \_id，并且是主键，这列的值是会自动增长的整数。另外还有两列：username( 字符 ) 和 password( 字符 )。 SQLite 会自动为主键列创建索引。

　　通常情况下，第一次创建数据库时创建了表和索引。要 删除表和索引，需要使用 execSQL() 方法调用 DROP INDEX 和 DROP TABLE 语句。

##添加数据

　　有两种方法可以给表添加数据。

①可以使用 execSQL() 方法执行 INSERT, UPDATE, DELETE 等语句来更新表的数据。execSQL() 方法适用于所有不返回结果的 SQL 语句。例如：

```

String sql = "insert into user(username,password) values ('finch','123456');//插入操作的SQL语句

db.execSQL(sql);//执行SQL语句

```

②使用 SQLiteDatabase 对象的 insert()。

```

ContentValues cv = new ContentValues();

cv.put("username","finch");//添加用户名

cv.put("password","123456"); //添加密码

db.insert("user",null,cv);//执行插入操作

```

##更新数据（修改）

①使用SQLiteDatabase 对象的 update()方法。

```

ContentValues cv = new ContentValues();

cv.put("password","654321");//添加要更改的字段及内容

String whereClause = "username=?";//修改条件

String[] whereArgs = {"finch"};//修改条件的参数

db.update("user",cv,whereClause,whereArgs);//执行修改

```

该方法有四个参数：

　　表名；

　　列名和值的 ContentValues 对象；

　　可选的 WHERE 条件；

　　可选的填充 WHERE 语句的字符串，这些字符串会替换 WHERE 条件中的“？”标记，update() 根据条件，更新指定列的值.

②使用execSQL方式的实现

```

String sql = "update [user] set password = '654321' where username="finch";//修改的SQL语句

db.execSQL(sql);//执行修改

```

##删除数据

①使用SQLiteDatabase 对象的delete()方法。

```

String whereClause = "username=?";//删除的条件

String[] whereArgs = {"finch"};//删除的条件参数

db.delete("user",whereClause,whereArgs);//执行删除

```

②使用execSQL方式的实现

```

String sql = "delete from user where username="finch";//删除操作的SQL语句

db.execSQL(sql);//执行删除操作

```

##查询数据

①使用 rawQuery() 直接调用 SELECT 语句

```

Cursor c = db.rawQuery("select \* from user where username=?",new Stirng[]{"finch"});

if(cursor.moveToFirst()) {

String password = c.getString(c.getColumnIndex("password"));

}

```

　　返回值是一个 cursor 对象，这个对象的方法可以迭代查询结果。

如果查询是动态的，使用这个方法就会非常复杂。例如，当你需要查询的列在程序编译的时候不能确定，这时候使用 query() 方法会方便很多。

②通过query实现查询

　　query() 方法用 SELECT 语句段构建查询。

　　SELECT 语句内容作为 query() 方法的参数，比如：要查询的表名，要获取的字段名，WHERE 条件，包含可选的位置参数，去替代 WHERE 条件中位置参数的值，GROUP BY 条件，HAVING 条件。

　　除了表名，其他参数可以是 null。所以代码可写成：

```

Cursor c = db.query("user",null,null,null,null,null,null);//查询并获得游标

if(c.moveToFirst()){//判断游标是否为空

for(int i=0;i<c.getCount();i++){

c.move(i);//移动到指定记录

String username = c.getString(c.getColumnIndex("username");

String password = c.getString(c.getColumnIndex("password"));

}

}

```

###使用游标

　　不管你如何执行查询，都会返回一个 Cursor，这是 Android 的 SQLite 数据库游标，使用游标，你可以：

- 通过使用 getCount() 方法得到结果集中有多少记录；

- 通过 moveToFirst(), moveToNext(), 和 isAfterLast() 方法遍历所有记录；

- 通过 getColumnNames() 得到字段名；

- 通过 getColumnIndex() 转换成字段号；

- 通过 getString()，getInt() 等方法得到给定字段当前记录的值；

- 通过 requery() 方法重新执行查询得到游标；

- 通过 close() 方法释放游标资源；

例如，下面代码遍历 user表:

```

Cursor result=db.rawQuery("SELECT \_id, username, password FROM user");

result.moveToFirst();

while (!result.isAfterLast()) {

int id=result.getInt(0);

String name=result.getString(1);

String password =result.getString(2);

// do something useful with these

result.moveToNext();

}

result.close();

```

## 第16节 AIDL的使用情况和实例介绍

#AIDL是什么？

　　AIDL （Android Interface Definition Language）， Android接口定义语言，Android提供的IPC （Inter Process Communication，进程间通信）的一种独特实现。

#什么情况下要使用AIDL

　　使用AIDL只有在你允许来自不同应用的客户端跨进程通信访问你的service，并且想要在你的service种处理\*\*多线程\*\*的时候才是必要的。 如果你不需要执行不同应用之间的IPC并发，你应该通过实现Binder建立你的接口，或者如果你想执行IPC，但是不需要处理多线程。那么使用Messenger实现你的接口。

#定义一个AIDL接口的步骤

　　必须在一个.aidl文件中使用java编程语言语法定义你的AIDL接口，然后在提供service的应用中和任何绑定到这个service的应用中的源代码中（在src目录吓）保存它。

　　当你编译包含.aidl文件的应用时，Android SDK工具基于这个.aidl文件生成一个IBinder接口，并且把它保存到项目的gen目录吓．service必须恰当的实现这个IBinder接口 之后客户端应用可以绑定到这个服务上，然后从IBinder调用方法来执行IPC。

使用AIDL建立一个邻接的service需要遵循下面的步骤：

1.　建立.aidl文件

　　这个文件使用方法签名定义了语言接口

2.实现这个接口

　　Android SDk工具基于你的.aidl文件使用java语言生成一个接口 这个接口有一个内部抽象类，叫做Stub，它是继承Binder并且实现你AIDL接口的 你必须继承这个Stub类并且实现这些方法

3.暴露这个接口给客户端

　　实现一个service并且覆盖onBind()方法返回你的Stub实现类。

> 你的.aidl文件必须被复制到其他应用程序中来让他们访问你service的接口，你必须维护原始接口的支持（向后兼容）。

#用一个实例来分步骤说明

##在server项目中建立.aidl文件

![这里写图片描述](http://img.blog.csdn.net/20160504180944041)

　　AIDL使用一个简单的语法让你声明一个带有一个或者多个带有参数和返回值方法的接口 参数和返回值可以是任何类型，甚至是AIDL生成的接口。

　　IService.aidl

```

package com.example.aidl;

interface IService {

String hello(String name);

}

```

##在server项目中建立服务类

　　当你编译你的应用时，Android SDK工具生成一个.java接口文件用你的.aidl文件命名生成的接口包含一个名字为Stub的子类，这是一个它父类的抽象实现，并且声明了.aidl中所有的方法。

　　Stub也定义了一些辅助的方法，最显著的就是asInterface()，它是用来接收一个IBinder（通常IBinder传递给客户端的onServiceConnected()回调方法）并且返回一个Stub接口的实例 。

　　一旦你为service实现了接口，你需要把它暴露给客户端，这样他们才能绑定到上面 为了给你的service暴露接口，继承Service并且实现onBind()方法返回一个你实现生成的Stub类。

　　　AIDLService.java

```

public class AIDLService extends Service {

@Override

public void onCreate() {

super.onCreate();

}

@Override

public IBinder onBind(Intent intent) {

// Return the interface

return new IService.Stub() {

@Override

public String hello(String name) throws RemoteException {

// TODO Auto-generated method stub

return "hello"+name;

}

};

}

```

##在server项目AndroidManifest中申明Service

```

<service

android:name="com.example.service.AIDLService" >

　　 <intent-filter>

<action android:name="android.intent.action.AIDLService" />

　　</intent-filter>

</service>

```

##把server项目中的aidl文件带包拷贝到client项目中（包名要相同）

![这里写图片描述](http://img.blog.csdn.net/20160504181850678)

MainActivity.java

```

public class MainActivity extends Activity {

IService RemoteService; //监听服务

private ServiceConnection mConnection = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

// TODO Auto-generated method stub

Log.i("mConnection", service+"");

RemoteService = IService.Stub.asInterface(service);

try {

String s= RemoteService.hello("finch");

Toast.makeText(MainActivity.this, s, Toast.LENGTH\_LONG).show();

} catch (RemoteException e) {

e.printStackTrace();

}

}

@Override

public void onServiceDisconnected(ComponentName name) {

// TODO Auto-generated method stub

}

};

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

initService();

}

//连接服务

private void initService() {

Intent i = new Intent( );

i.setAction("android.intent.action.AIDLService");

boolean ret = bindService(i, mConnection, Context.BIND\_AUTO\_CREATE);

}

//断开服务

private void releaseService() {

unbindService(mConnection);

mConnection = null;

}

@Override

protected void onDestroy() {

super.onDestroy();

releaseService();

}

}

```

运行结果：

<img src="http://img.blog.csdn.net/20160504182305682" width="286" height="473" />

[文章中AIDL例子代码下载](http://download.csdn.net/detail/amazing7/9510133)

## 第17节 [Android] 名企面试题及答案整理（一）

- 下面异常是属于Runtime Exception 的是(abcd)(多选)

A、ArithmeticException

B、IllegalArgumentException

C、NullPointerException

D、BufferUnderflowException

解析：

Java提供了两类主要的异常:runtime exception和checked exception。checked 异常也就是我们经常遇到的IO异常，以及SQL异常都是这种异常。对于这种异常，JAVA编译器强制要求我们必需对出现的这些异常进行catch。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆catch块去处理可能的异常。

出现运行时异常后，系统会把异常一直往上层抛，一直遇到处理代码。如果没有处理块，到最上层，如果是多线程就由Thread.run()抛出，如果是单线程就被main()抛出。抛出之后，如果是线程，这个线程也就退出了。如果是主程序抛出的异常，那么这整个程序也就退出了。运行时异常是Exception的子类，也有一般异常的特点，是可以被Catch块处理的。只不过往往我们不对他处理罢了。也就是说，你如果不对运行时异常进行处理，那么出现运行时异常之后，要么是线程中止，要么是主程序终止。

编译时被检查的异常和运行时异常的区别：

　编译被检查的异常在函数内被抛出，函数必须要声明，否编译失败。

　声明的原因：是需要调用者对该异常进行处理。

　运行时异常如果在函数内被抛出，在函数上不需要声明。

- Math.round(11.5)等于多少(). Math.round(-11.5)等于多少(c)

A、11 ,-11 B、11 ,-12 C、12 ,-11 D、12 ,-12

解析：

Math.ceil()用作向上取整。

Math.floor()用作向下取整。

Math.round() 我们数学中常用到的四舍五入取整。

- 对一些资源以及状态的操作保存，最好是保存在生命周期的哪个函数中进行(d)

A、onPause() B、onCreate() C、 onResume() D、onStart()

解析：

![这里写图片描述](http://img.blog.csdn.net/20160411181000798)

系统杀死程序会调用onSaveInstanceState(Bundle)进行数据保存，这里保存的数据会出现在在程序下一次onStart（Bundle）这个Bundle中，onStart时可以将Bundle中数据取出。

- Intent传递数据时，下列的数据类型哪些可以被传递(abcd)(多选)

A、Serializable B、charsequence C、Parcelable D、Bundle

解析：

Serializable :将 Java 对象序列化为二进制文件的 Java 序列化技术是 Java系列技术中一个较为重要的技术点，在大部分情况下，开发人员只需要了解被序列化的类需要实现 Serializable 接口，使用ObjectInputStream 和 ObjectOutputStream 进行对象的读写。

charsequence :

实现了这个接口的类有：CharBuffer、String、StringBuffer、StringBuilder这个四个类。

CharBuffer为nio里面用的一个类，String实现这个接口理所当然，StringBuffer也是一个CharSequence，StringBuilder是Java抄袭C#的一个类，基本和StringBuffer类一样，效率高，但是不保证线程安全，在不需要多线程的环境下可以考虑。

提供这么一个接口，有些处理String或者StringBuffer的类就不用重载了。但是这个接口提供的方法有限，只有下面几个：charat、length、subSequence、toString这几个方法，感觉如果有必要，还是重载的比较好，避免用instaneof这个操作符。

Parcelable :

android提供了一种新的类型：Parcel。本类被用作封装数据的容器，封装后的数据可以通过Intent或IPC传递。 除了基本类型以

外，只有实现了Parcelable接口的类才能被放入Parcel中。

是GOOGLE在安卓中实现的另一种序列化,功能和Serializable相似,主要是序列化的方式不同

Bundle是将数据传递到另一个上下文中或保存或回复你自己状态的数据存储方式。它的数据不是持久化状态。

- 下列属于SAX解析xml文件的优点的是(b)

A、将整个文档树在内存中，便于操作，支持删除，修改，重新排列等多种功能

B、不用事先调入整个文档，占用资源少

C、整个文档调入内存，浪费时间和空间

D、不是长久驻留在内存，数据不是持久的，事件过后，若没有保存数据，数据就会消失

解析：

在Android中提供了三种解析XML的方式:SAX(Simple API XML),DOM(Document Objrect Model),以及Android推荐的Pull解析方式。

SAX： 是事件驱动型XML解析的一个标准接口，简单地说就是对文档进行顺序扫描，当扫描到文档（document）开始与结束、元素（element）开始与结束、文档（document）结束等地方时通知事件处理函数，由事件处理函数做相应动作，然后继续同样的扫描，直至文档结束。

DOM：即对象文档模型，它是将整个XML文档载入内存(所以效率较低，不推荐使用)，使用DOM API遍历XML树、检索所需的数据，每一个节点当做一个对象。

Pull：运行方式与 SAX 解析器相似。它提供了类似的事件，SAX解析器的工作方式是自动将事件推入事件处理器进行处理，因此你不能控制事件的处理主动结束；而Pull解析器的工作方式为允许你的应用程序代码主动从解析器中获取事件，正因为是主动获取事件，因此可以在满足了需要的条件后不再获取事件，结束解析。pull是一个while循环，随时可以跳出，而sax是只要解析了，就必须解析完成。

- 在android中使用Menu时可能需要重写的方法有(ac)。(多选)

A、onCreateOptionsMenu()

B、onCreateMenu()

C、onOptionsItemSelected()

D、onItemSelected()

解析：

android中有三种菜单

1.选项菜单Options menus :一个Activity只能有一个选项菜单，在按下Menu键时，显示在屏幕下方。

重写 onCreateContextMenu 用以创建上下文菜单

重写 onContextItemSelected 用以响应上下文菜单

2.上下文菜单Context menus :为Activity中的任何一个视图注册一个上下文菜单，“长按”出现。

重写 onCreateOptionsMenu 用以创建选项菜单

重写 onOptionsItemSelected 用以响应选项菜单

3.弹出式菜单Popup menus :依赖于Activity中的某个一个视图

- android 关于service生命周期的onCreate()和onStart()说法正确的是(ad)(多选题)

A、当第一次启动的时候先后调用onCreate()和onStart()方法

B、当第一次启动的时候只会调用onCreate()方法

C、如果service已经启动，将先后调用onCreate()和onStart()方法

D、如果service已经启动，只会执行onStart()方法，不在执行onCreate()方法

解析：

![这里写图片描述](http://img.blog.csdn.net/20160411181016027)

1). 被启动的服务的生命周期：如果一个Service被某个Activity 调用 Context.startService 方法启动，那么不管是否有Activity使用bindService绑定或unbindService解除绑定到该Service，该Service都在后台运行。如果一个Service被startService 方法多次启动，那么onCreate方法只会调用一次，onStart将会被调用多次（对应调用startService的次数），并且系统只会创建Service的一个实例（因此你应该知道只需要一次stopService调用）。该Service将会一直在后台运行，而不管对应程序的Activity是否在运行，直到被调用stopService，或自身的stopSelf方法。当然如果系统资源不足，android系统也可能结束服务。

2). 被绑定的服务的生命周期：如果一个Service被某个Activity 调用 Context.bindService 方法绑定启动，不管调用 bindService 调用几次，onCreate方法都只会调用一次，同时onStart方法始终不会被调用。当连接建立之后，Service将会一直运行，除非调用Context.unbindService 断开连接或者之前调用bindService 的 Context 不存在了（如Activity被finish的时候），系统将会自动停止Service，对应onDestroy将被调用。

3). 被启动又被绑定的服务的生命周期：如果一个Service又被启动又被绑定，则该Service将会一直在后台运行。并且不管如何调用，onCreate始终只会调用一次，对应startService调用多少次，Service的onStart便会调用多少次。调用unbindService将不会停止Service，而必须调用 stopService 或 Service的 stopSelf 来停止服务。

4). 当服务被停止时清除服务：当一个Service被终止（1、调用stopService；2、调用stopSelf；3、不再有绑定的连接（没有被启动））时，onDestroy方法将会被调用，在这里你应当做一些清除工作，如停止在Service中创建并运行的线程。

特别注意：

1、你应当知道在调用 bindService 绑定到Service的时候，你就应当保证在某处调用 unbindService 解除绑定（尽管 Activity 被 finish 的时候绑定会自动解除，并且Service会自动停止）；

2、你应当注意 使用 startService 启动服务之后，一定要使用 stopService停止服务，不管你是否使用bindService；

3、同时使用 startService 与 bindService 要注意到，Service 的终止，需要unbindService与stopService同时调用，才能终止 Service，不管 startService 与 bindService 的调用顺序，如果先调用 unbindService 此时服务不会自动终止，再调用 stopService 之后服务才会停止，如果先调用 stopService 此时服务也不会终止，而再调用 unbindService 或者 之前调用 bindService 的 Context 不存在了（如Activity 被 finish 的时候）之后服务才会自动停止；

4、当在旋转手机屏幕的时候，当手机屏幕在“横”“竖”变换时，此时如果你的 Activity 如果会自动旋转的话，旋转其实是 Activity 的重新创建，因此旋转之前的使用 bindService 建立的连接便会断开（Context 不存在了），对应服务的生命周期与上述相同。

5、在 sdk 2.0 及其以后的版本中，对应的 onStart 已经被否决变为了 onStartCommand，不过之前的 onStart 任然有效。这意味着，如果你开发的应用程序用的 sdk 为 2.0 及其以后的版本，那么你应当使用 onStartCommand 而不是 onStart。

- 下面是属于GLSurFaceView特性的是(abc)(多选)

A、管理一个surface，这个surface就是一块特殊的内存，能直接排版到android的视图view上。

B、管理一个EGL display，它能让opengl把内容渲染到上述的surface上。

C、让渲染器在独立的线程里运作，和UI线程分离。

D、可以直接从内存或者DMA等硬件接口取得图像数据

解析：

Android游戏当中主要的除了控制类外就是显示类View。SurfaceView是从View基类中派生出来的显示类。android游戏开发中常用的三种视图是：view、SurfaceView和GLSurfaceView。

　　View：显示视图，内置画布，提供图形绘制函数、触屏事件、按键事件函数等；必须在UI主线程内更新画面，速度较慢。

　　SurfaceView：基于view视图进行拓展的视图类，更适合2D游戏的开发；是view的子类，类似使用双缓机制，在新的线程中更新画面所以刷新界面速度比view快。

　　GLSurfaceView：基于SurfaceView视图再次进行拓展的视图类，专用于3D游戏开发的视图；是SurfaceView的子类，openGL专用。

GLSurfaceView提供了下列特性：

1.管理一个surface，这个surface就是一块特殊的内存，能直接排版到android的视图view上。

2.管理一个EGL display，它能让opengl把内容渲染到上述的surface上。

3.用户自定义渲染器(render)。

4 . 让渲染器在独立的线程里运作，和UI线程分离。

5.支持按需渲染(on-demand)和连续渲染(continuous)。

6.一些可选工具，如调试。

- 关于ContenValues类说法正确的是(a)

A、他和Hashtable比较类似，也是负责存储一些名值对，但是他存储的名值对当中的名是String类型，而值都是基本类型

　　B、他和Hashtable比较类似，也是负责存储一些名值对，但是他存储的名值对当中的名是任意类型，而值都是基本类型

　　C、他和Hashtable比较类似，也是负责存储一些名值对，但是他存储的名值对当中的名，可以为空，而值都是String类型

　　D、他和Hashtable比较类似，也是负责存储一些名值对，但是他存储的名值对当中的名是String类型，而值也是String类型

解析：

ContentValues 和HashTable类似都是一种存储的机制 但是两者最大的区别就在于，contenvalues Key只能是String类型，values只能存储基本类型的数据，像string，int之类的，不能存储对象这种东西。ContentValues 常用在数据库中的操作。

HashMap是Hashtable的轻量级实现（非线程安全的实现），他们都完成了Map接口，主要区别在于HashMap允许空（null）键值（key）,由于非线程安全，效率上可能高于Hashtable。HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。

- 下面退出Activity错误的方法是(c)

A、finish()

B、抛异常强制退出

C、System.exit()

D、onStop()

解析：

![这里写图片描述](http://img.blog.csdn.net/20160411181031423)

finish()：在你的activity动作完成的时候，或者Activity需要关闭的时候，调用此方法。当你调用此方法的时候，系统只是将最上面的Activity移出了栈，并没有及时的调用onDestory（）方法，其占用的资源也没有被及时释放。因为移出了栈，所以当你点击手机上面的“back”按键的时候，也不会再找到这个Activity。finish函数仅仅把当前Activity退出了，但是并没有释放他的资源。安卓系统自己决定何时从内存中释放应用程序。当系统没有可用内存到时候，会按照优先级，释放部分应用。

onDestory()：系统销毁了这个Activity的实例在内存中占据的空间。

在Activity的生命周期中，onDestory()方法是他生命的最后一步，资源空间等就被回收了。当重新进入此Activity的时候，必须重新创建，执行onCreate()方法。

System.exit(0)：退出整个应用程序（不仅仅是当前activity）。将整个进程直接Kill掉。

- 关于res/raw目录说法正确的是(a)

A、 这里的文件是原封不动的存储到设备上不会转换为二进制的格式

B、这里的文件是原封不动的存储到设备上会转换为二进制的格式

C、 这里的文件最终以二进制的格式存储到指定的包中

D、这里的文件最终不会以二进制的格式存储到指定的包中

解析：

res/raw和assets的相同点：

两者目录下的文件在打包后会原封不动的保存在apk包中，不会被编译成二进制。

res/raw和assets的不同点：

1.res/raw中的文件会被映射到R.java文件中，访问的时候直接使用资源ID即R.id.filename；assets文件夹下的文件不会被映射到R.java中，访问的时候需要AssetManager类。

- android中常用的四个布局是framlayout，linenarlayout，relativelayout和tablelayout。

- android 的四大组件是activiey，service，broadcast和contentprovide。

- activity一般会重载7个方法用来维护其生命周期，除了onCreate(),onStart(),onDestory() 外还有onpause,onresume,onstop，onrestart。

- android的数据存储的方式sharedpreference,文件,SQlite,contentprovider,网络。

- 程序运行的结果是：good and gbc

```

public classExample{

　　String str=new String("good");

　　char[]ch={'a','b','c'};

　　public static void main(String args[]){

　　Example ex=new Example();

　　ex.change(ex.str,ex.ch);

　　System.out.print(ex.str+" and ");

　　Sytem.out.print(ex.ch);

　　}

　　public void change(String str,char ch[]){

　　str="test ok";

　　ch[0]='g';

　　}

　　}

```

解析：

public void change(String str,char ch[])

str是按值传递，所以在函数中对它的操作只生效于它的副本，与原字符串无关。

ch是按址传递，在函数中根据地址，可以直接对字符串进行操作。

- 在android中，请简述jni的调用过程。

1)安装和下载Cygwin，下载 Android NDK

　　2)在ndk项目中JNI接口的设计

　　3)使用C/C++实现本地方法

　　4)JNI生成动态链接库.so文件

　 5)将动态链接库复制到java工程，在java工程中调用，运行java工程即可

- Android应用程序结构：

Linux Kernel(Linux内核)、Libraries(系统运行库或者是c/c++核心库)、Application Framework(开发框架包)、Applications (核心应用程序)

![这里写图片描述](http://img.blog.csdn.net/20160411181237328)

- 请继承SQLiteOpenHelper实现创建一个版本为1的“diaryOpenHelper.db”的数据库，同时创建一个 “diary” 表(包含一个\_id主键并自增长，topic字符型100长度， content字符型1000长度)，在数据库版本变化时请删除diary表，并重新创建出diary表。

```

public class DBHelper extends SQLiteOpenHelper{

　　public final static String DATABASENAME ="diaryOpenHelper.db";

　　public final static int DATABASEVERSION =1;

　　//创建数据库

　　public DBHelper(Context context,Stringname,CursorFactory factory,int version)

　　{

　　super(context, name, factory,version);

　　}

　　//创建表等机构性文件

　　public void onCreate(SQLiteDatabase db)

　　{

　　String sql ="create tablediary"+

　　"("+

　　"\_idinteger primary key autoincrement,"+

　　"topicvarchar(100),"+

　　"contentvarchar(1000)"+

　　")";

　　db.execSQL(sql);

　　}

　　//若数据库版本有更新，则调用此方法

　　public void onUpgrade(SQLiteDatabasedb,int oldVersion,int newVersion)

　　{

　　String sql = "drop table ifexists diary";

　　db.execSQL(sql);

　　this.onCreate(db);

　　}

　　}

```

- 页面上现有ProgressBar控件progressBar，请用书写线程以10秒的的时间完成其进度显示工作。

```

public class ProgressBarStu extends Activity {

　　private ProgressBar progressBar = null;

　　protected void onCreate(BundlesavedInstanceState) {

　　super.onCreate(savedInstanceState);

　　setContentView(R.layout.progressbar);

　　//从这到下是关键

　　progressBar = (ProgressBar)findViewById(R.id.progressBar);

　　Thread thread = new Thread(newRunnable() {

　　@Override

　　public void run() {

　　int progressBarMax =progressBar.getMax();

　　try {

　　while(progressBarMax!=progressBar.getProgress())

　　{

　　intstepProgress = progressBarMax/10;

　　intcurrentprogress = progressBar.getProgress();

　　progressBar.setProgress(currentprogress+stepProgress);

　　Thread.sleep(1000);

　　}

　　} catch(InterruptedException e) {

　　// TODO Auto-generatedcatch block

　　e.printStackTrace();

　　}

　　}

　　});

　　thread.start();

　　//关键结束

　　}

　　}

```

- onFreeze() renamed to onSaveInstanceState()，以便恢复在onCreate(Bundle)里面设置的状态。

- 如果后台的Activity由于某原因被系统回收了，onSaveInstanceState()在被系统回收之前（onPause()前面）保存当前状态。

当你的程序中某一个Activity A在运行时，主动或被动地运行另一个新的Activity B，这个时候A会执行onSaveInstanceState()。B完成以后又会来找A，这个时候就有两种情况：一是A被回收，二是A没有被回收，被回收的A就要重新调用onCreate()方法，不同于直接启动的是这回onCreate()里是带上了参数savedInstanceState;而没被收回的就直接执行onResume()，跳过onCreate()了。

- ContentProvider：

提供了我们在应用程序之前共享数据的一种机制，而我们知道每一个应用程序都是运行在不同的应用程序的，数据和文件在不同应用程序之间达到数据的共享不是没有可能，而是显得比较复杂，而正好Android中的ContentProvider则达到了这一需求，比如有时候我们需要操作手机里的联系人，手机里的多媒体等一些信息，我们都可以用到这个ContentProvider来达到我们所需。

1）、ContentProvider为存储和获取数据提供了统一的接口。ContentProvide对数据进行封装，不用关心数据存储的细节。使用表的形式来组织数据。

2）、使用ContentProvider可以在不同的应用程序之间共享数据。

3）、Android为常见的一些数据提供了默认的ContentProvider（包括音频、视频、图片和通讯录等）。

总的来说使用ContentProvider对外共享数据的好处是统一了数据的访问方式。

Uri为系统的每一个资源给其一个名字，比方说通话记录。每一个ContentProvider都拥有一个公共的URI，这个URI用于表示这个ContentProvider所提供的数据。

- 请解释下Android程序运行时权限与文件系统权限的区别。

运行时权限Dalvik( android授权)

文件系统 linux 内核授权

- 什么是ANR 如何避免它?

在Android里，应用程序的响应性是由Activity Manager和Window Manager系统服务监视的。当它监测到以下情况中的一个时，Android就会针对特定的应用程序显示ANR：

在5秒内没有响应输入的事件（例如，按键按下，屏幕触摸）

BroadcastReceiver在10秒内没有执行完毕。

Android应用程序通常是运行在一个单独的线程（例如，main）里。这意味着你的应用程序所做的事情如果在主线程里占用了太长的时间的话，就会引发ANR对话框，因为你的应用程序并没有给自己机会来处理输入事件或者Intent广播。

在主线程里尽量的少做事情，比如高耗时的计算和网络、数据库等潜在的耗时操作都应该放在子线程来完成。

# 第二章 JAVA

## 第1节 ArrayList、LinkedList、Vector的区别

首先我们来看一下继承关系：

![这里写图片描述](http://img.blog.csdn.net/20160408150531095)

- 我们可以看出ArrayList、LinkedList、Vector都实现了List的接口。

接下来分别看一下三个数据结构的说明。

> public class \*\*ArrayList<E>\*\* extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, Serializable

>List 接口的\*\*大小可变数组\*\*的实现。实现了所有可选列表操作，并\*\*允许包括 null 在内的所有元素\*\*。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。（此类大致上等同于 Vector 类，除了\*\*此类是不同步的\*\*。）

>

每个 ArrayList 实例都有一个容量。该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向 ArrayList 中不断添加元素，其容量也自动增长。并未指定增长策略的细节，因为这不只是添加元素会带来分摊固定时间开销那样简单。

>在添加大量元素前，应用程序可以使用 ensureCapacity 操作来增加 ArrayList 实例的容量。这可以减少递增式再分配的数量。

>

List list = Collections.synchronizedList(new ArrayList(...));

>此类的 iterator 和 listIterator 方法返回的迭代器是快速失败的：在创建迭代器之后，除非通过迭代器自身的 remove 或 add 方法从结构上对列表进行修改，否则在任何时间以任何方式对列表进行修改，迭代器都会抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为无法得到保证，因为一般来说，不可能对是否出现不同步并发修改做出任何硬性保证。快速失败迭代器会尽最大努力抛出 ConcurrentModificationException。因此，为提高这类迭代器的正确性而编写一个依赖于此异常的程序是错误的做法：迭代器的快速失败行为应该仅用于检测 bug。

- 然后是LinkedList

> public class \*\*LinkedList\*\*<E> extends AbstractSequentialList<E>

implements List<E>, Deque<E>, Cloneable, Serializable

>List 接口的链接列表实现。实现所有可选的列表操作，并且允许所有元素（\*\*包括 null\*\*）。除了实现 List 接口外，LinkedList 类还为在列表的开头及结尾 get、remove 和 insert 元素提供了统一的命名方法。这些操作允许将链接列表用作堆栈、队列或双端队列。

>此类实现 Deque 接口，为 add、poll 提供先进先出队列操作，以及其他堆栈和双端队列操作。

>所有操作都是按照\*\*双重链接列表\*\*的需要执行的。在列表中编索引的操作将从开头或结尾遍历列表（从靠近指定索引的一端）。

>注意，此实现\*\*不是同步\*\*的。如果多个线程同时访问一个链接列表，而其中至少一个线程从结构上修改了该列表，则它必须 保持外部同步。（结构修改指添加或删除一个或多个元素的任何操作；仅设置元素的值不是结构修改。）这一般通过\*\*对自然封装该列表的对象进行同步操作来完成\*\*。如果不存在这样的对象，则应该使用 Collections.synchronizedList 方法来“包装”该列表。最好在创建时完成这一操作，以防止对列表进行意外的不同步访问，如下所示：

> List list = Collections.synchronizedList(new LinkedList(...));

>

此类的 iterator 和 listIterator 方法返回的迭代器是快速失败 的：在迭代器创建之后，如果从结构上对列表进行修改，除非通过迭代器自身的 remove 或 add 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒将来不确定的时间任意发生不确定行为的风险。

>注意，迭代器的快速失败行为不能得到保证，一般来说，存在不同步的并发修改时，不可能作出任何硬性保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的方式是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

- 最后是Vector

> public class \*\*Vector\*\*<E> extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, Serializable

> Vector 类可以\*\*实现可增长的对象数组\*\*。与数组一样，它包含可以使用整数索引进行访问的组件。但是，Vector 的大小可以根据需要增大或缩小，以适应创建 Vector 后进行添加或移除项的操作。

> 每个向量会试图通过维护 capacity 和 capacityIncrement 来优化存储管理。capacity 始终至少应与向量的大小相等；这个值通常比后者大些，因为随着将组件添加到向量中，其存储将按 capacityIncrement 的大小增加存储块。应用程序可以在插入大量组件前增加向量的容量；这样就减少了增加的重分配的量。

> 由 Vector 的 iterator 和 listIterator 方法所返回的迭代器是快速失败的：如果在迭代器创建后的任意时间从结构上修改了向量（通过迭代器自身的 remove 或 add 方法之外的任何其他方式），则迭代器将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就完全失败，而不是冒着在将来不确定的时间任意发生不确定行为的风险。Vector 的 elements 方法返回的 Enumeration 不是 快速失败的。

> 注意，迭代器的快速失败行为不能得到保证，一般来说，存在不同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的方式是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测 bug。

> 从 Java 2 平台 v1.2 开始，此类改进为可以实现 List 接口，使它成为 Java Collections Framework 的成员。与新 collection 实现不同，\*\*Vector 是同步的\*\*。

- 区别

ArrayList 本质上是一个可改变大小的\*\*数组\*\*.当元素加入时,其大小将会动态地增长.内部的元素可以直接通过get与set方法进行访问.元素顺序存储 ,\*\*随机访问很快，删除非头尾元素慢，新增元素慢而且费资源\*\* ,较适用于无频繁增删的情况 ,比数组效率低，如果不是需要可变数组，可考虑使用数组 ,\*\*非线程安全\*\*.

LinkedList 是一个\*\*双链表\*\*,在添加和删除元素时具有比ArrayList更好的性能.但在get与set方面弱于ArrayList. 适用于 ：没有大规模的随机读取，大量的增加/删除操作.\*\*随机访问很慢，增删操作很快\*\*，不耗费多余资源 ,允许null元素,\*\*非线程安全.\*\*

Vector （类似于ArrayList）但其是\*\*同步\*\*的，开销就比ArrayList要大。如果你的程序本身是线程安全的，那么使用ArrayList是更好的选择。

Vector和ArrayList在更多元素添加进来时会请求更大的空间。Vector每次请求其大小的双倍空间，而ArrayList每次对size增长50%.

## 第2节 Collection包结构，与Collections的区别

> Collection家族

![这里写图片描述](http://img.blog.csdn.net/20160408150531095)

Collection是集合继承结构中的顶层接口

Collections 是提供了对集合进行操作的强大方法的工具类 ，它包含有各种有关集合操作的静态多态方法。此类不能实例化

## 第3节 Excption与Error包结构,OOM和SOF

>Java 异常类继承关系图

![这里写图片描述](http://img.blog.csdn.net/20160412143252629)

（一）Throwable

　　Throwable 类是 Java 语言中所有错误或异常的超类。只有当对象是此类或其子类之一的实例时，才能通过 Java 虚拟机或者 Java throw 语句抛出，才可以是 catch 子句中的参数类型。

　　Throwable 类及其子类有两个构造方法，一个不带参数，另一个带有 String 参数，此参数可用于生成详细消息。

　　Throwable 包含了其线程创建时线程执行堆栈的快照。它还包含了给出有关错误更多信息的消息字符串。

Java将可抛出(Throwable)的结构分为三种类型：

　　错误(Error)

　　运行时异常(RuntimeException)

　　被检查的异常(Checked Exception)

　１.\*\*Error\*\*

　　Error 是 Throwable 的子类，用于指示合理的应用程序不应该试图捕获的严重问题。大多数这样的错误都是异常条件。

　　和RuntimeException一样， 编译器也不会检查Error。

　　当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误，程序本身无法修复这些错误的。

　２.\*\*Exception\*\*

　　Exception 类及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。

　　 对于可以恢复的条件使用\*\*被检查异常\*\*（Exception的子类中除了RuntimeException之外的其它子类），对于程序错误使用运行时异常。

>①　ClassNotFoundException

当应用程序试图使用以下方法通过字符串名加载类时：

Class 类中的 forName 方法。

ClassLoader 类中的 findSystemClass 方法。

ClassLoader 类中的 loadClass 方法。

但是没有找到具有指定名称的类的定义，抛出该异常。

②　CloneNotSupportedException

当调用 Object 类中的 clone 方法复制对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。重写 clone 方法的应用程序也可能抛出此异常，指示不能或不应复制一个对象。

> ③　IOException

当发生某种 I/O 异常时，抛出此异常。此类是失败或中断的 I/O 操作生成的异常的通用类。

－EOFException

　　当输入过程中意外到达文件或流的末尾时，抛出此异常。

此异常主要被\*\*数据输入流\*\*用来表明到达流的末尾。

注意：其他许多输入操作返回一个特殊值表示到达流的末尾，而不是抛出异常。

－FileNotFoundException

　　当试图打开指定路径名表示的文件失败时，抛出此异常。

在不存在具有指定路径名的文件时，此异常将由 FileInputStream、FileOutputStream 和 RandomAccessFile 构造方法抛出。如果该文件存在，但是由于某些原因不可访问，比如试图打开一个只读文件进行写入，则此时这些构造方法仍然会抛出该异常。

－MalformedURLException

　　抛出这一异常指示出现了错误的 URL。或者在规范字符串中找不到任何合法协议，或者无法解析字符串。

－UnknownHostException

　　指示主机 IP 地址无法确定而抛出的异常。

>④　RuntimeException

　　 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。可能在执行方法期间抛出但未被捕获的 RuntimeException 的任何子类都无需在 throws 子句中进行声明。

　　 Java编译器不会检查它。当程序中可能出现这类异常时，还是会编译通过。

　　 虽然Java编译器不会检查运行时异常，但是我们也可以通过throws进行声明抛出，也可以通过try-catch对它进行捕获处理。

－ArithmeticException

当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。

－ClassCastException

　　当试图将对象强制转换为不是实例的子类时，抛出该异常。

例如：Object x = new Integer(0);

－LllegalArgumentException

　　抛出的异常表明向方法传递了一个不合法或不正确的参数。

－IllegalStateException

　　在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。

－IndexOutOfBoundsException

　　指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。

应用程序可以为这个类创建子类，以指示类似的异常。

－NoSuchElementException

　　由 Enumeration 的 nextElement 方法抛出，表明枚举中没有更多的元素。

－NullPointerException

　　当应用程序试图在需要对象的地方使用 null 时，抛出该异常。这种情况包括：

调用 null 对象的实例方法。

访问或修改 null 对象的字段。

将 null 作为一个数组，获得其长度。

将 null 作为一个数组，访问或修改其时间片。

将 null 作为 Throwable 值抛出。

应用程序应该抛出该类的实例，指示其他对 null 对象的非法使用。

（二） SOF （堆栈溢出 StackOverflow）

> StackOverflowError 的定义：

> 当应用程序递归太深而发生堆栈溢出时，抛出该错误。

>

因为栈一般默认为1-2m，一旦出现死循环或者是大量的递归调用，在不断的压栈过程中，造成栈容量超过1m而导致溢出。

栈溢出的原因：

递归调用

大量循环或死循环

全局变量是否过多

数组、List、map数据过大

（三）Android的ＯＯＭ（Out Of Memory）

　　当内存占有量超过了虚拟机的分配的最大值时就会产生内存溢出（VM里面分配不出更多的page）。

一般出现情况：加载的图片太多或图片过大时、分配特大的数组、内存相应资源过多没有来不及释放。

解决方法：

①在内存引用上做处理

软引用是主要用于内存敏感的高速缓存。在jvm报告内存不足之前会清除所有的软引用，这样以来gc就有可能收集软可及的对象，可能解决内存吃紧问题，避免内存溢出。什么时候会被收集取决于gc的算法和gc运行时可用内存的大小。

　②对图片做边界压缩，配合软引用使用

　③显示的调用GC来回收内存

```

if(bitmapObject.isRecycled()==false) //如果没有回收

bitmapObject.recycle();

```

　④优化Dalvik虚拟机的堆内存分配

　　１．增强程序堆内存的处理效率

```

//在程序onCreate时就可以调用 即可

private final static floatTARGET\_HEAP\_UTILIZATION = 0.75f;

VMRuntime.getRuntime().setTargetHeapUtilization(TARGET\_HEAP\_UTILIZATION);

```

　2 .设置堆内存的大小

```

private final static int CWJ\_HEAP\_SIZE = 6\* 1024\* 1024 ;

//设置最小heap内存为6MB大小

VMRuntime.getRuntime().setMinimumHeapSize(CWJ\_HEAP\_SIZE);

```

⑤ 用LruCache 和 AsyncTask<>解决

　　从cache中去取Bitmap，如果取到Bitmap，就直接把这个Bitmap设置到ImageView上面。

　　如果缓存中不存在，那么启动一个task去加载（可能从文件来，也可能从网络）。

## 第4节 HashMap和HashTable的区别

- HashTable

>

Hashtable继承于Dictionary字典，实现Map接口

> 键、值都不能是空对象

>多次访问，映射元素的顺序相同

>线程安全

>

>

>hash算法 ，Hashtable则直接利用key本身的hash码来做验证

>

>数据遍历的方式 Iterator （支持fast-fail）和 Enumeration （不支持fast-fail）

>

> 缺省初始长度为11，内部都为抽象方法，需要 它的实现类一一作自己的实现

备注：程序在对 collection 进行迭代时，某个线程对该 collection 在结构上对其做了修改，这时迭代器就会抛出 ConcurrentModificationException 异常信息，从而产生 fail-fast。

- HashMap

> HashMap继承于AbstractMap抽象类

>

> 键和值都可以是空对象

>

多次访问，映射元素的顺序可能不同

> 非线程安全

HashMap可以通过下面的语句进行同步：

Map m = Collections.synchronizeMap(hashMap);

>

> 检测是否含有key时，HashMap内部需要将key的hash码重新计算一边再检测

>

> 数据遍历的方式 Iterator （支持fast-fail）

>

> 缺省初始长度为16，其内部已经实现了Map所需 要做的大部分工作， 它的子类只需要实现它的少量方法

## 第5节 HashMap源码分析

#1.概述

　　Hashmap继承于AbstractMap，实现了Map、Cloneable、java.io.Serializable接口。它的key、value都可以为null，映射不是有序的。

　　Hashmap不是同步的，如果想要线程安全的HashMap，可以通过Collections类的静态方法synchronizedMap获得线程安全的HashMap。

```

Map map = Collections.synchronizedMap(new HashMap());

```

（除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）

HashMap 中两个重要的参数：“初始容量” 和 “加载因子”。

> \*\*容量\*\* 是哈希表中桶的数量，初始容量 只是哈希表在创建时的容量

>

> \*\*加载因子\*\* 是哈希表在其容量自动增加之前可以达到多满的一种尺度（默认0.75）。

>

>当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构，桶数Ｘ２）。

>

>加载因子越大,填满的元素越多,好处是,空间利用率高了,但:冲突的机会加大了.反之,加载因子越小,填满的元素越少, 好处是:冲突的机会减小了,但:空间浪费多了.

#2.HashMap的数据结构

　　Hashmap本质是\*\*数组加链表\*\*。通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样，然后再计算出数组下标，如果多个key对应到同一个下标，就用链表串起来，新插入的在\*\*前面\*\*。

　　结构图如下（通过key查找value）：

![这里写图片描述](http://img.blog.csdn.net/20160429162312540)

（图来源于网络）

先来看看HashMap中Entry类的代码：

```

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

// 指向下一个节点

Entry<K,V> next;

final int hash;

// 构造函数。

// 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"

Entry(int h, K k, V v, Entry<K,V> n) {

value = v;

next = n;

key = k;

hash = h;

}

public final K getKey() {

return key;

}

public final V getValue() {

return value;

}

public final V setValue(V newValue) {

V oldValue = value;

value = newValue;

return oldValue;

}

// 判断两个Entry是否相等

// 若两个Entry的“key”和“value”都相等，则返回true。

// 否则，返回false

public final boolean equals(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry e = (Map.Entry)o;

Object k1 = getKey();

Object k2 = e.getKey();

if (k1 == k2 || (k1 != null && k1.equals(k2))) {

Object v1 = getValue();

Object v2 = e.getValue();

if (v1 == v2 || (v1 != null && v1.equals(v2)))

return true;

}

return false;

}

// 实现hashCode()

public final int hashCode() {

return (key==null ? 0 : key.hashCode()) ^

(value==null ? 0 : value.hashCode());

}

public final String toString() {

return getKey() + "=" + getValue();

}

// 当向HashMap中添加元素时，绘调用recordAccess()。

// 这里不做任何处理

void recordAccess(HashMap<K,V> m) {

}

// 当从HashMap中删除元素时，绘调用recordRemoval()。

// 这里不做任何处理

void recordRemoval(HashMap<K,V> m) {

}

}

```

我们可以理解HashMap就是一个Entry数组，Entry对象中包含了键和值。

#3.HashMap源码分析

HashMap共有4个构造函数,如下：

> HashMap()

构造一个具有默认初始容量 (16) 和默认加载因子 (0.75) 的空 HashMap。

>HashMap(int initialCapacity)

构造一个带指定初始容量和默认加载因子 (0.75) 的空 HashMap。

>HashMap(int initialCapacity, float loadFactor)

构造一个带指定初始容量和加载因子的空 HashMap。

>HashMap(Map<　 extends K, extends V> m)

构造一个映射关系与指定 Map 相同的新 HashMap。

HashMap提供的API方法：

> void clear()

从此映射中移除所有映射关系。

> Object clone()

返回此 HashMap 实例的浅表副本：并不复制键和值本身。

>boolean containsKey(Object key)

如果此映射包含对于指定键的映射关系，则返回 true。

> boolean containsValue(Object value)

如果此映射将一个或多个键映射到指定值，则返回 true。

> Set entrySet()

返回此映射所包含的映射关系的 Set＜Map.Entry＞ 视图。

> V get(Object key)

返回指定键所映射的值；如果对于该键来说，此映射不包含任何映射关系，则返回 null。

> boolean isEmpty()

如果此映射不包含键-值映射关系，则返回 true。

> Set keySet()

返回此映射中所包含的键的 Set＜Ｋ＞ 视图。

>V put(K key, V value)

在此映射中关联指定值与指定键。

> void　 putAll(Map< extends K, extends V> m)

将指定映射的所有映射关系复制到此映射中，这些映射关系将替换此映射目前针对指定映射中所有键的所有映射关系。

> V remove(Object key)

从此映射中移除指定键的映射关系（如果存在）。

> int 　size()

返回此映射中的键-值映射关系数。

> Collection values()

返回此映射所包含的值的 Collection<V> 视图。

HashMap源码：

```

package java.util;

import java.io.\*;

public class HashMap<K,V>

extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

{

// 默认的初始容量（容量为HashMap中桶的数目）是16，且实际容量必须是2的整数次幂。

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

// 最大容量（必须是2的幂且小于2的30次方，传入容量过大将被这个值替换）

static final int MAXIMUM\_CAPACITY = 1 << 30;

// 默认加载因子

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

// 存储数据的Entry数组，长度是2的幂。

// HashMap是采用拉链法实现的，每一个Entry本质上是一个单向链表

transient Entry[] table;

// HashMap的大小，它是HashMap保存的键值对的数量

transient int size;

// HashMap的阈值，用于判断是否需要调整HashMap的容量（threshold = 容量\*加载因子）

int threshold;

// 加载因子实际大小

final float loadFactor;

// HashMap被改变的次数

transient volatile int modCount;

// 指定“容量大小”和“加载因子”的构造函数

public HashMap(int initialCapacity, float loadFactor) {

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

// HashMap的最大容量只能是MAXIMUM\_CAPACITY

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

if (loadFactor <= 0 || Float.isNaN(loadFactor))

throw new IllegalArgumentException("Illegal load factor: " +

loadFactor);

// 找出“大于initialCapacity”的最小的2的幂

int capacity = 1;

while (capacity < initialCapacity)

capacity <<= 1;

// 设置“加载因子”

this.loadFactor = loadFactor;

// 设置“HashMap阈值”，当HashMap中存储数据的数量达到threshold时，就需要将HashMap的容量加倍。

threshold = (int)(capacity \* loadFactor);

// 创建Entry数组，用来保存数据

table = new Entry[capacity];

init();

}

// 指定“容量大小”的构造函数

public HashMap(int initialCapacity) {

this(initialCapacity, DEFAULT\_LOAD\_FACTOR);

}

// 默认构造函数。

public HashMap() {

// 设置“加载因子”

this.loadFactor = DEFAULT\_LOAD\_FACTOR;

// 设置“HashMap阈值”，当HashMap中存储数据的数量达到threshold时，就需要将HashMap的容量加倍。

threshold = (int)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);

// 创建Entry数组，用来保存数据

table = new Entry[DEFAULT\_INITIAL\_CAPACITY];

init();

}

// 包含“子Map”的构造函数

public HashMap(Map<? extends K, ? extends V> m) {

this(Math.max((int) (m.size() / DEFAULT\_LOAD\_FACTOR) + 1,

DEFAULT\_INITIAL\_CAPACITY), DEFAULT\_LOAD\_FACTOR);

// 将m中的全部元素逐个添加到HashMap中

putAllForCreate(m);

}

static int hash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

// 返回索引值

// h & (length-1)保证返回值的小于length

static int indexFor(int h, int length) {

return h & (length-1);

}

public int size() {

return size;

}

public boolean isEmpty() {

return size == 0;

}

// 获取key对应的value

public V get(Object key) {

if (key == null)

return getForNullKey();

// 获取key的hash值

int hash = hash(key.hashCode());

// 在“该hash值对应的链表”上查找“键值等于key”的元素

for (Entry<K,V> e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

// 获取“key为null”的元素的值

// HashMap将“key为null”的元素存储在table[0]位置！

private V getForNullKey() {

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null)

return e.value;

}

return null;

}

// HashMap是否包含key

public boolean containsKey(Object key) {

return getEntry(key) != null;

}

// 返回“键为key”的键值对

final Entry<K,V> getEntry(Object key) {

// 获取哈希值

// HashMap将“key为null”的元素存储在table[0]位置，“key不为null”的则调用hash()计算哈希值

int hash = (key == null) ? 0 : hash(key.hashCode());

// 在“该hash值对应的链表”上查找“键值等于key”的元素

for (Entry<K,V> e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

return e;

}

return null;

}

// 将“key-value”添加到HashMap中

public V put(K key, V value) {

// 若“key为null”，则将该键值对添加到table[0]中。

if (key == null)

return putForNullKey(value);

// 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

// 若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

// 若“该key”对应的键值对不存在，则将“key-value”添加到table中

modCount++;

addEntry(hash, key, value, i);

return null;

}

// putForNullKey()的作用是将“key为null”键值对添加到table[0]位置

private V putForNullKey(V value) {

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

// 这里的完全不会被执行到!

modCount++;

addEntry(0, null, value, 0);

return null;

}

// 创建HashMap对应的“添加方法”，

// 它和put()不同。putForCreate()是内部方法，它被构造函数等调用，用来创建HashMap

// 而put()是对外提供的往HashMap中添加元素的方法。

private void putForCreate(K key, V value) {

int hash = (key == null) ? 0 : hash(key.hashCode());

int i = indexFor(hash, table.length);

// 若该HashMap表中存在“键值等于key”的元素，则替换该元素的value值

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {

e.value = value;

return;

}

}

// 若该HashMap表中不存在“键值等于key”的元素，则将该key-value添加到HashMap中

createEntry(hash, key, value, i);

}

// 将“m”中的全部元素都添加到HashMap中。

// 该方法被内部的构造HashMap的方法所调用。

private void putAllForCreate(Map<? extends K, ? extends V> m) {

// 利用迭代器将元素逐个添加到HashMap中

for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator(); i.hasNext(); ) {

Map.Entry<? extends K, ? extends V> e = i.next();

putForCreate(e.getKey(), e.getValue());

}

}

// 重新调整HashMap的大小，newCapacity是调整后的单位

void resize(int newCapacity) {

Entry[] oldTable = table;

int oldCapacity = oldTable.length;

if (oldCapacity == MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return;

}

// 新建一个HashMap，将“旧HashMap”的全部元素添加到“新HashMap”中，

// 然后，将“新HashMap”赋值给“旧HashMap”。

Entry[] newTable = new Entry[newCapacity];

transfer(newTable);

table = newTable;

threshold = (int)(newCapacity \* loadFactor);

}

// 将HashMap中的全部元素都添加到newTable中

void transfer(Entry[] newTable) {

Entry[] src = table;

int newCapacity = newTable.length;

for (int j = 0; j < src.length; j++) {

Entry<K,V> e = src[j];

if (e != null) {

src[j] = null;

do {

Entry<K,V> next = e.next;

int i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

} while (e != null);

}

}

}

// 将"m"的全部元素都添加到HashMap中

public void putAll(Map<? extends K, ? extends V> m) {

// 有效性判断

int numKeysToBeAdded = m.size();

if (numKeysToBeAdded == 0)

return;

// 计算容量是否足够，

// 若“当前实际容量 < 需要的容量”，则将容量x2。

if (numKeysToBeAdded > threshold) {

int targetCapacity = (int)(numKeysToBeAdded / loadFactor + 1);

if (targetCapacity > MAXIMUM\_CAPACITY)

targetCapacity = MAXIMUM\_CAPACITY;

int newCapacity = table.length;

while (newCapacity < targetCapacity)

newCapacity <<= 1;

if (newCapacity > table.length)

resize(newCapacity);

}

// 通过迭代器，将“m”中的元素逐个添加到HashMap中。

for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator(); i.hasNext(); ) {

Map.Entry<? extends K, ? extends V> e = i.next();

put(e.getKey(), e.getValue());

}

}

// 删除“键为key”元素

public V remove(Object key) {

Entry<K,V> e = removeEntryForKey(key);

return (e == null ? null : e.value);

}

// 删除“键为key”的元素

final Entry<K,V> removeEntryForKey(Object key) {

// 获取哈希值。若key为null，则哈希值为0；否则调用hash()进行计算

int hash = (key == null) ? 0 : hash(key.hashCode());

int i = indexFor(hash, table.length);

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

// 删除链表中“键为key”的元素

// 本质是“删除单向链表中的节点”

while (e != null) {

Entry<K,V> next = e.next;

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {

modCount++;

size--;

if (prev == e)

table[i] = next;

else

prev.next = next;

e.recordRemoval(this);

return e;

}

prev = e;

e = next;

}

return e;

}

// 删除“键值对”

final Entry<K,V> removeMapping(Object o) {

if (!(o instanceof Map.Entry))

return null;

Map.Entry<K,V> entry = (Map.Entry<K,V>) o;

Object key = entry.getKey();

int hash = (key == null) ? 0 : hash(key.hashCode());

int i = indexFor(hash, table.length);

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

// 删除链表中的“键值对e”

// 本质是“删除单向链表中的节点”

while (e != null) {

Entry<K,V> next = e.next;

if (e.hash == hash && e.equals(entry)) {

modCount++;

size--;

if (prev == e)

table[i] = next;

else

prev.next = next;

e.recordRemoval(this);

return e;

}

prev = e;

e = next;

}

return e;

}

// 清空HashMap，将所有的元素设为null

public void clear() {

modCount++;

Entry[] tab = table;

for (int i = 0; i < tab.length; i++)

tab[i] = null;

size = 0;

}

// 是否包含“值为value”的元素

public boolean containsValue(Object value) {

// 若“value为null”，则调用containsNullValue()查找

if (value == null)

return containsNullValue();

// 若“value不为null”，则查找HashMap中是否有值为value的节点。

Entry[] tab = table;

for (int i = 0; i < tab.length ; i++)

for (Entry e = tab[i] ; e != null ; e = e.next)

if (value.equals(e.value))

return true;

return false;

}

// 是否包含null值

private boolean containsNullValue() {

Entry[] tab = table;

for (int i = 0; i < tab.length ; i++)

for (Entry e = tab[i] ; e != null ; e = e.next)

if (e.value == null)

return true;

return false;

}

// 克隆一个HashMap，并返回Object对象

public Object clone() {

HashMap<K,V> result = null;

try {

result = (HashMap<K,V>)super.clone();

} catch (CloneNotSupportedException e) {

// assert false;

}

result.table = new Entry[table.length];

result.entrySet = null;

result.modCount = 0;

result.size = 0;

result.init();

// 调用putAllForCreate()将全部元素添加到HashMap中

result.putAllForCreate(this);

return result;

}

// Entry是单向链表。

// 它是 “HashMap链式存储法”对应的链表。

// 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

// 指向下一个节点

Entry<K,V> next;

final int hash;

// 构造函数。

// 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"

Entry(int h, K k, V v, Entry<K,V> n) {

value = v;

next = n;

key = k;

hash = h;

}

public final K getKey() {

return key;

}

public final V getValue() {

return value;

}

public final V setValue(V newValue) {

V oldValue = value;

value = newValue;

return oldValue;

}

// 判断两个Entry是否相等

// 若两个Entry的“key”和“value”都相等，则返回true。

// 否则，返回false

public final boolean equals(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry e = (Map.Entry)o;

Object k1 = getKey();

Object k2 = e.getKey();

if (k1 == k2 || (k1 != null && k1.equals(k2))) {

Object v1 = getValue();

Object v2 = e.getValue();

if (v1 == v2 || (v1 != null && v1.equals(v2)))

return true;

}

return false;

}

// 实现hashCode()

public final int hashCode() {

return (key==null ? 0 : key.hashCode()) ^

(value==null ? 0 : value.hashCode());

}

public final String toString() {

return getKey() + "=" + getValue();

}

// 当向HashMap中添加元素时，绘调用recordAccess()。

// 这里不做任何处理

void recordAccess(HashMap<K,V> m) {

}

// 当从HashMap中删除元素时，绘调用recordRemoval()。

// 这里不做任何处理

void recordRemoval(HashMap<K,V> m) {

}

}

// 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。

void addEntry(int hash, K key, V value, int bucketIndex) {

// 保存“bucketIndex”位置的值到“e”中

Entry<K,V> e = table[bucketIndex];

// 设置“bucketIndex”位置的元素为“新Entry”，

// 设置“e”为“新Entry的下一个节点”

table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

// 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小

if (size++ >= threshold)

resize(2 \* table.length);

}

// 创建Entry。将“key-value”插入指定位置，bucketIndex是位置索引。

// 它和addEntry的区别是：

// (01) addEntry()一般用在 新增Entry可能导致“HashMap的实际容量”超过“阈值”的情况下。

// 例如，我们新建一个HashMap，然后不断通过put()向HashMap中添加元素；

// put()是通过addEntry()新增Entry的。

// 在这种情况下，我们不知道何时“HashMap的实际容量”会超过“阈值”；

// 因此，需要调用addEntry()

// (02) createEntry() 一般用在 新增Entry不会导致“HashMap的实际容量”超过“阈值”的情况下。

// 例如，我们调用HashMap“带有Map”的构造函数，它绘将Map的全部元素添加到HashMap中；

// 但在添加之前，我们已经计算好“HashMap的容量和阈值”。也就是，可以确定“即使将Map中

// 的全部元素添加到HashMap中，都不会超过HashMap的阈值”。

// 此时，调用createEntry()即可。

void createEntry(int hash, K key, V value, int bucketIndex) {

// 保存“bucketIndex”位置的值到“e”中

Entry<K,V> e = table[bucketIndex];

// 设置“bucketIndex”位置的元素为“新Entry”，

// 设置“e”为“新Entry的下一个节点”

table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

size++;

}

// HashIterator是HashMap迭代器的抽象出来的父类，实现了公共了函数。

// 它包含“key迭代器(KeyIterator)”、“Value迭代器(ValueIterator)”和“Entry迭代器(EntryIterator)”3个子类。

private abstract class HashIterator<E> implements Iterator<E> {

// 下一个元素

Entry<K,V> next;

// expectedModCount用于实现fast-fail机制。

int expectedModCount;

// 当前索引

int index;

// 当前元素

Entry<K,V> current;

HashIterator() {

expectedModCount = modCount;

if (size > 0) { // advance to first entry

Entry[] t = table;

// 将next指向table中第一个不为null的元素。

// 这里利用了index的初始值为0，从0开始依次向后遍历，直到找到不为null的元素就退出循环。

while (index < t.length && (next = t[index++]) == null)

}

}

public final boolean hasNext() {

return next != null;

}

// 获取下一个元素

final Entry<K,V> nextEntry() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

Entry<K,V> e = next;

if (e == null)

throw new NoSuchElementException();

// 注意！！！

// 一个Entry就是一个单向链表

// 若该Entry的下一个节点不为空，就将next指向下一个节点;

// 否则，将next指向下一个链表(也是下一个Entry)的不为null的节点。

if ((next = e.next) == null) {

Entry[] t = table;

while (index < t.length && (next = t[index++]) == null)

}

current = e;

return e;

}

// 删除当前元素

public void remove() {

if (current == null)

throw new IllegalStateException();

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

Object k = current.key;

current = null;

HashMap.this.removeEntryForKey(k);

expectedModCount = modCount;

}

}

// value的迭代器

private final class ValueIterator extends HashIterator<V> {

public V next() {

return nextEntry().value;

}

}

// key的迭代器

private final class KeyIterator extends HashIterator<K> {

public K next() {

return nextEntry().getKey();

}

}

// Entry的迭代器

private final class EntryIterator extends HashIterator<Map.Entry<K,V>> {

public Map.Entry<K,V> next() {

return nextEntry();

}

}

// 返回一个“key迭代器”

Iterator<K> newKeyIterator() {

return new KeyIterator();

}

// 返回一个“value迭代器”

Iterator<V> newValueIterator() {

return new ValueIterator();

}

// 返回一个“entry迭代器”

Iterator<Map.Entry<K,V>> newEntryIterator() {

return new EntryIterator();

}

// HashMap的Entry对应的集合

private transient Set<Map.Entry<K,V>> entrySet = null;

// 返回“key的集合”，实际上返回一个“KeySet对象”

public Set<K> keySet() {

Set<K> ks = keySet;

return (ks != null ? ks : (keySet = new KeySet()));

}

// Key对应的集合

// KeySet继承于AbstractSet，说明该集合中没有重复的Key。

private final class KeySet extends AbstractSet<K> {

public Iterator<K> iterator() {

return newKeyIterator();

}

public int size() {

return size;

}

public boolean contains(Object o) {

return containsKey(o);

}

public boolean remove(Object o) {

return HashMap.this.removeEntryForKey(o) != null;

}

public void clear() {

HashMap.this.clear();

}

}

// 返回“value集合”，实际上返回的是一个Values对象

public Collection<V> values() {

Collection<V> vs = values;

return (vs != null ? vs : (values = new Values()));

}

// “value集合”

// Values继承于AbstractCollection，不同于“KeySet继承于AbstractSet”，

// Values中的元素能够重复。因为不同的key可以指向相同的value。

private final class Values extends AbstractCollection<V> {

public Iterator<V> iterator() {

return newValueIterator();

}

public int size() {

return size;

}

public boolean contains(Object o) {

return containsValue(o);

}

public void clear() {

HashMap.this.clear();

}

}

// 返回“HashMap的Entry集合”

public Set<Map.Entry<K,V>> entrySet() {

return entrySet0();

}

// 返回“HashMap的Entry集合”，它实际是返回一个EntrySet对象

private Set<Map.Entry<K,V>> entrySet0() {

Set<Map.Entry<K,V>> es = entrySet;

return es != null ? es : (entrySet = new EntrySet());

}

// EntrySet对应的集合

// EntrySet继承于AbstractSet，说明该集合中没有重复的EntrySet。

private final class EntrySet extends AbstractSet<Map.Entry<K,V>> {

public Iterator<Map.Entry<K,V>> iterator() {

return newEntryIterator();

}

public boolean contains(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry<K,V> e = (Map.Entry<K,V>) o;

Entry<K,V> candidate = getEntry(e.getKey());

return candidate != null && candidate.equals(e);

}

public boolean remove(Object o) {

return removeMapping(o) != null;

}

public int size() {

return size;

}

public void clear() {

HashMap.this.clear();

}

}

// java.io.Serializable的写入函数

// 将HashMap的“总的容量，实际容量，所有的Entry”都写入到输出流中

private void writeObject(java.io.ObjectOutputStream s)

throws IOException

{

Iterator<Map.Entry<K,V>> i =

(size > 0) ? entrySet0().iterator() : null;

// Write out the threshold, loadfactor, and any hidden stuff

s.defaultWriteObject();

// Write out number of buckets

s.writeInt(table.length);

// Write out size (number of Mappings)

s.writeInt(size);

// Write out keys and values (alternating)

if (i != null) {

while (i.hasNext()) {

Map.Entry<K,V> e = i.next();

s.writeObject(e.getKey());

s.writeObject(e.getValue());

}

}

}

private static final long serialVersionUID = 362498820763181265L;

// java.io.Serializable的读取函数：根据写入方式读出

// 将HashMap的“总的容量，实际容量，所有的Entry”依次读出

private void readObject(java.io.ObjectInputStream s)

throws IOException, ClassNotFoundException

{

// Read in the threshold, loadfactor, and any hidden stuff

s.defaultReadObject();

// Read in number of buckets and allocate the bucket array;

int numBuckets = s.readInt();

table = new Entry[numBuckets];

init(); // Give subclass a chance to do its thing.

// Read in size (number of Mappings)

int size = s.readInt();

// Read the keys and values, and put the mappings in the HashMap

for (int i=0; i<size; i++) {

K key = (K) s.readObject();

V value = (V) s.readObject();

putForCreate(key, value);

}

}

// 返回“HashMap总的容量”

int capacity() { return table.length; }

// 返回“HashMap的加载因子”

float loadFactor() { return loadFactor; }

}

```

- 在public V get(Object key)方法中：

　　如果key不为null，则先求的key的hash值，根据hash值找到在table中的索引，在该索引对应的单链表中查找是否有键值对的key与目标key相等，有就返回对应的value，没有则返回null。

　　如果key为null，则直接从哈希表的第一个位置table[0]对应的链表上查找。记住，key为null的键值对永远都放在以table[0]为头结点的链表中，当然不一定是存放在头结点table[0]中。

- public V put(K key, V value) 方法中：

　　 如果key不为null，则同样先求出key的hash值，根据hash值得出在table中的索引，而后遍历对应的单链表，如果单链表中存在与目标key相等的键值对，则将新的value覆盖旧的value，并将旧的value返回，如果找不到与目标key相等的键值对，或者该单链表为空，则将该键值对插入到改单链表的头结点位置（每次新插入的节点都是放在头结点的位置），该操作是有addEntry方法实现的，它的源码如下：

```

void addEntry(int hash, K key, V value, int bucketIndex) {

Entry<K,V> e = table[bucketIndex]; //如果要加入的位置有值，将该位置原先的值设置为新entry的next,也就是新entry链表的下一个节点

table[bucketIndex] = new Entry<>(hash, key, value, e);

if (size++ >= threshold) //如果大于临界值就扩容

resize(2 \* table.length); //以2的倍数扩容

}

```

　　参数bucketIndex就是indexFor函数计算出来的索引值，第2行代码是取得数组中索引为bucketIndex的Entry对象，第3行就是用hash、key、value构建一个新的Entry对象放到索引为bucketIndex的位置，并且将该位置原先的对象设置为新对象的next构成链表。第4行和第5行就是判断put后size是否达到了临界值threshold，如果达到了临界值就要进行扩容，HashMap扩容是扩为原来的两倍。

　　如果key为null，则将其添加到table[0]对应的链表中，由putForNullKey（）实现。

```

// putForNullKey()的作用是将“key为null”键值对添加到table[0]位置

private V putForNullKey(V value) {

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

// 如果没有存在key为null的键值对，则直接题阿见到table[0]处!

modCount++;

addEntry(0, null, value, 0);

return null;

}

```

- resize扩容方法

```

void resize(int newCapacity) {

Entry[] oldTable = table;

int oldCapacity = oldTable.length;

if (oldCapacity == MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return;

}

Entry[] newTable = new Entry[newCapacity];

transfer(newTable);//用来将原先table的元素全部移到newTable里面

table = newTable; //再将newTable赋值给table

threshold = (int)(newCapacity \* loadFactor);//重新计算临界值

}

```

　　它新建了一个HashMap的底层数组，而后调用transfer方法，将就HashMap的全部元素添加到新的HashMap中（要重新计算元素在新的数组中的索引位置）。

　　扩容是需要进行数组复制的，非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

```

// 将HashMap中的全部元素都添加到newTable中

void transfer(Entry[] newTable) {

Entry[] src = table;

int newCapacity = newTable.length;

for (int j = 0; j < src.length; j++) {

Entry<K,V> e = src[j];

if (e != null) {

src[j] = null;

do {

Entry<K,V> next = e.next;

int i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

} while (e != null);

}

}

}

```

- 计算哈希值的方法

```

static int hash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

```

移位的操作使hash值的计算效率更高。

- 由hash值找到对应索引的方法

```

static int indexFor(int h, int length) {

return h & (length-1);

}

```

　　HashMap中则通过h&(length-1)的方法来代替取模，同样实现了均匀的散列，但效率要高很多，这也是HashMap对Hashtable的一个改进。

　　length为2的整数次幂的话，h&(length-1)就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；

> length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间。

#4.HashMap和ConcurrentHashMap的区别

　　ConcurrentHashMap是在hashMap的基础上，将数据分为多个segment(类似hashtable)，默认16个（concurrency level），然后在每一个分段上都用锁进行保护，从而让锁的粒度更精细一些，并发性能更好。

## 第6节 Hashcode的作用

关于HashCode的官方文档定义：

> hashcode方法返回该对象的哈希码值。支持该方法是为哈希表提供一些优点，例如，java.util.Hashtable 提供的哈希表。 hashCode 的常规协定是： 在 Java 应用程序执行期间，在同一对象上多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是对象上 equals 比较中所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。

如果根据 equals(Object) 方法，两个对象是相等的，那么在两个对象中的每个对象上调用 hashCode 方法都必须生成相同的整数结果。

以下情况不 是必需的：如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么在两个对象中的任一对象上调用 hashCode 方法必定会生成不同的整数结果。但是，程序员应该知道，为不相等的对象生成不同整数结果可以提高哈希表的性能。

实际上，由 Object 类定义的 hashCode 方法确实会针对不同的对象返回不同的整数。（这一般是通过将该对象的内部地址转换成一个整数来实现的，但是 JavaTM 编程语言不需要这种实现技巧。）

当equals方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。

- 以上这段官方文档的定义，我们可以抽出成以下几个关键点：

1、hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的；

2、如果两个对象相同，就是适用于equals(java.lang.Object) 方法，那么这两个对象的hashCode一定要相同；

3、如果对象的equals方法被重写，那么对象的hashCode也尽量重写，并且产生hashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点；

4、两个对象的hashCode相同，并不一定表示两个对象就相同，也就是不一定适用于equals(java.lang.Object) 方法，只能够说明这两个对象在散列存储结构中，如Hashtable，他们“存放在同一个篮子里”。

再归纳一下就是hashCode是用于查找使用的，而equals是用于比较两个对象的是否相等的。

- 总的来说，Java中的集合（Collection）有两类，一类 是List，再有一类是Set。前者集合内的元素是有序的，元素可以重复；后者元素无序，但元素不可重复。要想保证元素不重复，可两个元素是否重复应该依据什么来判断呢？这就是Object.equals方法了。但是，如果每增加一个元素就检查一 次，那么当元素很多时，后添加到集合中的元素比较的次数就非常多了。也就是说，如果集合中现在已经有1000个元素，那么第1001个元素加入集合时，它 就要调用1000次equals方法。这显然会大大降低效率。

于是，Java采用了哈希表的原理。哈希算法也称为散列算法，是 将数据依特定算法直接指定到一个地址上。这样一来，当集合要添加新的元素时，先调用这个元素的hashCode方法，就一下子能定位到它应该放置的物理位置上。如果这个位置上没有元素，它就可以 直接存储在这个位置上，不用再进行任何比较了；如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散 列其它的地址。所以这里存在一个冲突解决的问题。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。

所以，Java对于eqauls方法和hashCode方法是这样规定的：

　　1、如果两个对象相同，那么它们的hashCode值一定要相同；

　　2、如果两个对象的hashCode相同，它们并不一定相同

## 第7节 Map、Set、List、Queue、Stack的特点与用法

- Map

> 键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射到一个值。

>某些映射实现可明确保证其顺序，如 TreeMap 类；另一些映射实现则不保证顺序，如 HashMap 类。

>Map中元素，可以将key序列、value序列单独抽取出来。

使用keySet()抽取key序列，将map中的所有keys生成一个Set。

使用values()抽取value序列，将map中的所有values生成一个Collection。

为什么一个生成Set，一个生成Collection？那是因为，key总是独一无二的，value允许重复。

- Set

>一个不包含重复元素的 collection。更确切地讲，set 不包含满足 e1.equals(e2) 的元素对 e1 和 e2，并且最多包含一个 null 元素。

>

> 不可随机访问包含的元素

>

只能用Iterator实现单向遍历

>

> Set 没有同步方法

- List

> 可随机访问包含的元素

元素是有序的

可在任意位置增、删元素

不管访问多少次，元素位置不变

允许重复元素

用Iterator实现单向遍历，也可用ListIterator实现双向遍历

- Queue

> 先进先出

>

> Queue使用时要尽量避免Collection的add()和remove()方法，而是要使用offer()来加入元素，使用poll()来获取并移出元素。它们的优点是通过返回值可以判断成功与否，add()和remove()方法在失败的时候会抛出异常。 如果要使用前端而不移出该元素，使用element()或者peek()方法。

值得注意的是LinkedList类实现了Queue接口，因此我们可以把LinkedList当成Queue来用。

> Queue 实现通常不允许插入 null 元素，尽管某些实现（如 LinkedList）并不禁止插入 null。即使在允许 null 的实现中，也不应该将 null 插入到 Queue 中，因为 null 也用作 poll 方法的一个特殊返回值，表明队列不包含元素。

- Stack

> 后进先出

>

> Stack继承自Vector（可增长的对象数组），也是同步的

> 它通过五个操作对类 Vector 进行了扩展 ，允许将向量视为堆栈。它提供了通常的 push 和 pop 操作，以及取堆栈顶点的 peek 方法、测试堆栈是否为空的 empty 方法、在堆栈中查找项并确定到\*\*堆栈顶\*\*距离的 search 方法。

- 用法

> 如果涉及到堆栈、队列等操作，应该考虑用List；

> 对于需要快速插入，删除元素，应该使用LinkedList；

> 如果需要快速随机访问元素，应该使用ArrayList。

> 如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高

## 第8节 Object有哪些公用方法？

> 官方文档 http://docs.oracle.com/javase/8/docs/api/

![这里写图片描述](http://img.blog.csdn.net/20160406172807848)

![这里写图片描述](http://img.blog.csdn.net/20160406172424112)

protected Object clone()

创建并返回此对象的一个副本。

boolean equals(Object obj)

指示其他某个对象是否与此对象“相等”。

protected void finalize()

当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

Class getClass()

返回此 Object 的运行时类。

int hashCode()

返回该对象的哈希码值。

void notify()

唤醒在此对象监视器上等待的单个线程。

void notifyAll()

唤醒在此对象监视器上等待的所有线程。

String toString()

返回该对象的字符串表示。

void wait()

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。

void wait(long timeout)

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。

void wait(long timeout, int nanos)

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

## 第9节 Override和Overload的使用规则和区别

\*\*重写(Override)\*\*

　　重写是子类对父类的允许访问的方法的实现过程进行重新编写！

　　\*\*返回值和形参都不能改变\*\*。即外壳不变，重写内在实现！

　　重写的好处在于\*\*子类可以根据需要，定义特定于自己的行为\*\*。

也就是说子类能够根据需要实现父类的方法。

　　在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：

```

class Animal{

public void move(){

System.out.println("Animal　can move");

　 }

}

class Pig extends Animal{

public void move(){

System.out.println("Pig can Climb tree");

}

}

public class Test{

public static void main(String args[]){

Animal a = new Animal();

Animal b = new Pig ();

a.move();// 执行 Animal 类的方法

b.move();//执行 Pig 类的方法

}

}

```

以上实例编译运行结果如下：

```

Animal　can move

Pig can Climb tree

```

　　从上面的例子中可以看出，b 申明的是Animal类 的引用，但运行的是Pig类的move方法。

　　这是因为由于\*\*在编译阶段，只是检查参数的引用类型\*\*。

然而在\*\*运行时，Java虚拟机(JVM)指定对象的类型并且运行该对象的方法。\*\*

　　在上面的例子中，之所以能编译成功，是因为Animal类中存在move方法，然而运行时，运行的是特定对象（Pig 对象）的方法。

看下面的例子：

```

class Animal{

public void move(){

System.out.println("Animal　can move");

}

}

class Pig extends Animal{

public void move(){

System.out.println("Pig can Climb tree");

}

public void bark(){

System.out.println("Pig can bray");

}

}

public class Test{

public static void main(String args[]){

Animal a = new Animal();

Animal b = new Pig();

a.move();// 执行 Animal 类的方法

b.move();//执行 Pig 类的方法

b.bark();

}

}

```

以上实例编译运行结果会报如下错误：

```

Test.java:30: cannot find symbol

symbol : method bark()

location: class Animal

b.bark();

^

```

这是一个编译错误，因为b的引用类型Animal 类中没有bark方法。

\*\*方法的重写规则\*\*

\*\*参数列表必须完全\*\*与被重写方法的\*\*相同\*\*；

\*\*返回类型\*\*必须完全与被重写方法的返回类型\*\*相同\*\*；

\*\*访问权限不能比父类中被重写的方法的访问权限更高。\*\*例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected 或 private。

父类的成员方法只能被它的子类重写。

声明为\*\*final的方法不能被重写\*\*。

声明为\*\*static的方法不能被重写，但是能够被再次声明\*\*。

子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。

子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。

重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。

但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。

\*\*构造方法不能被重写\*\*。

如果不能继承一个方法，则不能重写这个方法( 父类的private方法)。

\*\*重载(Overload)\*\*

　　重载(overloading) 是在同一个类里面，\*\*方法名字相同，而参数不同，返回类型可以相同也可以不同\*\*的多个方法。

每个重载的方法（只能重载构造函数）都必须有一个独一无二的参数类型列表。

重载规则：

被重载的方法\*\*必须改变参数列表\*\*；

被重载的方法\*\*可以\*\*改变返回类型；

被重载的方法\*\*可以\*\*改变访问修饰符；

被重载的方法可以声明新的或更广的检查异常；

方法能够在同一个类中或者\*\*在一个子类\*\*中被重载。

```

public class Test{

public int test(){

System.out.println("Overload1");

return 1;

}

public void test(int a){

System.out.println("Overload2");

}

//以下两个参数类型顺序不同

public String test(int a,String s){

return "Overload3" + s;

}

public String test(String s,int a){

return "Overload4"+s;

}

public static void main(String[] args){

Test t = new Test();

System.out.println(t.test());

t.test(1);

System.out.println(t.test(1,"test3"));

System.out.println(t.test("test4",1));

}

}

```

\*\*重写与重载之间的区别\*\*

重载方法：

参数列表 必须修改

返回类型 可以修改

异常 可以修改

访问修饰符 可以修改

重写方法：

参数列表 不能改

返回类型 不能改

异常 可以减少或删除，不能抛出新的或更广的异常

访问修饰符 只能降低限制，不能变成高限制

> 限制由低到高 ： public 、protected 、private

## 第10节 Switch能否用string做参数？

1 . 在jdk 7 之前，switch 只能支持 byte、short、char、int 这几个基本数据类型和其对应的封装类型。switch后面的括号里面只能放int类型的值，但由于byte，short，char类型，它们会 \*\*自动\*\* 转换为int类型（精精度小的向大的转化），所以它们也支持。

对于精度比int大的类型，long、float、double，不会自动转换成int。要想使用就得加强转如（int）long。

另外boolean类型不参与转换，任何类型不能转换为boolean型.

2 .我们也可以用枚举类型实现switch可传入string参数

```

public enum En{

a,b,c

}

public static void main(String[] args) {

En t = En.a;

function(t);

}

public static void function(En type){

switch (type) {

case a:

System.err.println("a");

break;

case b:

System.err.println("b");

break;

case c:

System.err.println("c");

break;

default:

break;

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160406161357320)

3 . jdk7之后java加入了switch对string的支持,就不用枚举来实现啦！

```

public static void main(String[] args) {

String s = "a";

switch (s) {

case "a":

System.err.println("a");

break;

case "b":

System.err.println("b");

break;

case "c":

System.err.println("c");

break;

default:

break;

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160406161357320)

## 第11节 ThreadLocal的使用规则和源码分析

#ThreadLocal是什么？

　　线程局部变量，访问某个变量的每个线程都有自己的局部变量，它独立于变量的初始化副本。

　　它的功用非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，是每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量。

　　ThreadLocal 实例通常是类中的 private static 字段，它们希望将状态与某一个线程（例如，用户 ID 或事务 ID）相关联。每个线程都保持对其线程局部变量副本的隐式引用，只要线程是活动的并且 ThreadLocal 实例是可访问的；在线程消失之后，其线程局部实例的所有副本都会被垃圾回收（除非存在对这些副本的其他引用）。

#ThreadLocal的接口方法

- protected T initialValue()　：返回此线程局部变量的初始值

　　线程第一次使用 get() 方法访问变量时将调用此方法，但如果线程之前调用了 set(T) 方法，则不会对该线程再调用 initialValue 方法。通常，此方法对每个线程最多调用一次，但如果在调用 get() 后又调用了 remove()，则可能再次调用此方法。

　　该实现返回 null；如果程序员希望线程局部变量具有 null 以外的值，则必须为 ThreadLocal 创建子类，并重写此方法。通常将使用匿名内部类完成此操作。

- public T get()　：返回此线程局部变量的当前线程的值

　　如果变量没有用于当前线程的值，则先将其初始化为调用 initialValue() 方法返回的值。

- public void set(T value)　：将此线程局部变量的当前线程副本中的值设置为指定值

　　大部分子类不需要重写此方法，它们只依靠 initialValue() 方法来设置线程局部变量的值。

- public void remove()　：移除此线程局部变量当前线程的值

　　如果此线程局部变量随后被当前线程 读取，且这期间当前线程没有 设置其值，则将调用其 initialValue() 方法重新初始化其值。这将导致在当前线程多次调用 initialValue 方法。

　如果希望线程局部变量初始化其它值，那么需要自己实现ThreadLocal的子类并重写该方法，通常使用一个内部类对ThreadLocal进行实例化。

　比如下面的例子，SerialNum类为每一个类分配一个序号：

```

public class SerialNum {

// The next serial number to be assigned

private static int nextSerialNum = 0;

private static ThreadLocal serialNum = new ThreadLocal() {

protected synchronized Object initialValue() {

return new Integer(nextSerialNum++);

}

}

public static int get() {

return ((Integer) (serialNum.get())).intValue();

}

}

```

#ThreadLocal的内部实现

　　先看看ThreadLocal类的部分源码：

```

public class ThreadLocal<T> {

//省略...

126 protected T initialValue() {

127 return null;

128 }

159 public T get() {

160 Thread t = Thread.currentThread();

//当前线程为key存入ThreadLocalMap 中

161 ThreadLocalMap map = getMap(t);

162 if (map != null) {

163 ThreadLocalMap.Entry e = map.getEntry(this);

164 if (e != null) {

165 @SuppressWarnings("unchecked")

166 T result = (T)e.value;

167 return result;

168 }

169 }

170 return setInitialValue();

171 }

199 public void set(T value) {

200 Thread t = Thread.currentThread();

201 ThreadLocalMap map = getMap(t);

202 if (map != null)

203 map.set(this, value);

204 else

205 createMap(t, value);

206 }

219 public void remove() {

220 ThreadLocalMap m = getMap(Thread.currentThread());

221 if (m != null)

222 m.remove(this);

223 }

｝

```

　　我们可以很明显的看出ThreadLocal把线程和线程局部变量存在ThreadLocalMap中，而ThreadLocalMap是ThreadLocal的静态类部类，我们来看看ThreadLocalMap 的部分源码：

```

308 static class Entry extends 　　　　　WeakReference<ThreadLocal<?>> {

309

310 Object value;

311

312 Entry(ThreadLocal<?> k, Object v) {

313 super(k);

314 value = v;

315 }

316 }

```

　　这个Map的key是ThreadLocal对象的弱引用，当要抛弃掉ThreadLocal对象时，垃圾收集器会忽略这个key的引用而清理掉ThreadLocal对象 。

　　那么到底是ThreadLocal还是Thread持有ThreadLocalMap对象的引用呢？

我们在Thread源码中发现：

```

180 /\* ThreadLocal values pertaining to this thread. This map is maintained

181 \* by the ThreadLocal class. \*/

182 ThreadLocal.ThreadLocalMap threadLocals = null;

```

　　ThreadLocalMap变量属于Thread的内部属性,不同的Thread拥有完全不同的ThreadLocalMap变量.

　　 Thread中的ThreadLocalMap变量的值是在ThreadLocal对象进行set或者get操作时创建的.

　　 在创建ThreadLocalMap之前,会首先检查当前Thread中的ThreadLocalMap变量是否已经存在,如果不存在则创建一个；如果已经存在,则使用当前Thread已创建的ThreadLocalMap.

#ThreadLocal如何做到线程安全

　　从上面的分析我们可以得出：

- 因为每个Thread在进行对象访问时,访问的都是各自线程自己的ThreadLocalMap，所以保证了Thread与Thread之间的数据访问隔离。

- 不同的ThreadLocal实例操作同一Thread时，ThreadLocalMap在存储时采用当前ThreadLocal的实例作为key来保证数据访问隔离（上面源码Entry处可以看出）。

举个例子说明：

```

public class Test

{

static ThreadLocal<Integer> local=new ThreadLocal<Integer>(){

protected synchronized Integer initialValue(){

return 0;

}

};

public static void main( String args[]){

testRun t = new testRun();

new Thread(t).start();

new Thread(t).start();

}

public static class testRun implements Runnable{

@Override

public void run() {

// TODO Auto-generated method stub

for(int i=5 ;i>0;i--){

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+"::"+local.get());

int localValue=local.get();

local.set(++localValue);

}

}

}

}

```

输出结果为：

```

Thread-0::0

Thread-1::0

Thread-0::1

Thread-1::1

Thread-1::2

Thread-0::2

Thread-1::3

Thread-0::3

Thread-0::4

Thread-1::4

```

#TheadLocal模式与同步机制的区别

- 同步机制采用了“以时间换空间”的方式,提供一份变量,让不同的线程排队访问.而ThreadLocal采用了“以空间换时间”的方式,为每一个线程都提供一份变量的副本,从而实现同时访问而互不影响。

- Java中的synchronized是一个保留字,它依靠JVM的锁机制来实现临界区的函数或者变量的访问中的原子性.在同步机制中,通过对象的锁机制保证同一时间只有一个线程访问变量.此时,被用作“锁机制”的变量是多个线程共享的；

　　而ThreadLocal会为每一个线程维护一个和该线程绑定的变量的副本，从而隔离了多个线程的数据，每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。

- 同步机制是为了同步多个线程对相同资源的并发访问，是为了多个线程之间进行通信的有效方式。

　　而ThreadLocal是隔离多个线程的数据共享，从根本上就不在多个线程之间共享资源（变量），这样当然不需要对多个线程进行同步了。

　　所以，如果你需要进行多个线程之间进行通信，则使用同步机制。如果需要隔离多个线程之间的共享冲突，可以使用ThreadLocal。

## 第12节 ThreadPool用法与示例

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

1、ThreadPool的优点

</font>

</td>

</tr>

</table>

##

　　在java.util.concurrent包下，提供了一系列与线程池相关的类。合理的使用线程池，可以带来多个好处：

（1）降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗；

（2）提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行；

（3）提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

　　线程池可以应对突然大爆发量的访问，通过有限个固定线程为大量的操作服务，减少创建和销毁线程所需的时间。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

1、ThreadPool的创建

</font>

</td>

</tr>

</table>

##

　　我们一般通过Executors类下的四个成员函数创建相应的线程池：

```

//创建一个定时任务的线程池

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(4);

//创建单线程的线程池

ExecutorService singleThreadPool = Executors.newSingleThreadExecutor();

//创建缓存线程池（重用先前的线程）

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

//创建一个带有固定线程的线程池

ExecutorService threadPool = Executors.newFixedThreadPool(4);

```

我们可以从源码看出，上面四个方法会调用ThreadPoolExecutor的构造方法创建线程池。

```

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

```

创建一个ThreadPoolExecutor线程池一般需要以下几个参数：

- corePoolSize（线程池的基本大小）：

　　当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。

- maximumPoolSize（线程池最大大小）：

　　线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。

- keepAliveTime（线程活动保持时间）：

　　线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。

- TimeUnit（线程活动保持时间的单位）：

　　可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)等。

- workQueue（任务队列）：

　　用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列：ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockingQueue

- threadFactory：

　　用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。

- handler（饱和策略）：

　　当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。

　　我们尽量优先使用Executors提供的静态方法来创建线程池，如果Executors提供的方法无法满足要求，再自己通过ThreadPoolExecutor类来创建线程池。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

２、向线程池提交任务

</font>

</td>

</tr>

</table>

##

有两种方式：

①　我们可以使用execute提交的任务，但是execute方法没有返回值，所以无法判断任务是否被线程池执行成功。通过以下代码可知execute方法输入的任务是一个Runnable类的实例。

```

threadsPool.execute(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

}

});

```

②　使用submit 方法来提交任务，它会返回一个future,那么我们可以通过这个future来判断任务是否执行成功，通过future的get方法来获取返回值，get方法会阻塞住直到任务完成，而使用get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

```

Future<Object> future = executor.submit(harReturnValuetask);

try {

Object s = future.get();

} catch (InterruptedException e) {

// 处理中断异常

} catch (ExecutionException e) {

// 处理无法执行任务异常

} finally {

// 关闭线程池

executor.shutdown();

}

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

３、线程池的关闭

</font>

</td>

</tr>

</table>

##

　　我们可以通过调用线程池（ExecutorService的对象）的shutdown或shutdownNow方法来关闭线程池，它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的区别，shutdownNow首先将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而shutdown只是将线程池的状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程。

　　只要调用了这两个关闭方法的其中一个，isShutdown方法就会返回true。当所有的任务都已关闭后,才表示线程池关闭成功，这时调用isTerminaed方法会返回true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用shutdown来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

４、线程池的工作流程

</font>

</td>

</tr>

</table>

##

　　![这里写图片描述](http://img.blog.csdn.net/20160519124839474)

从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

1、首先线程池判断基本线程池是否已满（ corePoolSize ）没满，创建一个工作线程来执行任务。满了，则进入下个流程。

2、其次线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。

3、最后线程池判断整个线程池是否已满（< maximumPoolSize ？）？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

也就是说，线程池优先要创建出基本线程池大小（corePoolSize）的线程数量，没有达到这个数量时，每次提交新任务都会直接创建一个新线程，当达到了基本线程数量后，又有新任务到达，优先放入等待队列，如果队列满了，才去创建新的线程（不能超过线程池的最大数maxmumPoolSize）。

　　线程池创建线程时，会将线程封装成工作线程Worker，Worker在执行完任务后，还会无限循环获取工作队列里的任务来执行。我们可以从Worker的run方法里看到这点：

```

public void run() {

try {

Runnable task = firstTask;

firstTask = null;

while (task != null || (task = getTask()) != null) {

runTask(task);

task = null;

}

} finally {

workerDone(this);

}

}

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

５、线程池的监控

</font>

</td>

</tr>

</table>

##

　　通过继承线程池并重写线程池的beforeExecute，afterExecute和terminated方法，我们可以在任务执行前，执行后和线程池关闭前干一些事情。如监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

６、线程池的例子

</font>

</td>

</tr>

</table>

##

示例一：

```

public class DifferentKindsThreadPool {

public static void main(String[] args) {

// displayScheduledThreadPool();

// displaySingleThreadPool();

// displayCachedThreadPool();

displayThreadPool();

}

/\*\*

\* 创建一个定时任务的线程池

\*/

public static void displayScheduledThreadPool() {

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(4);

//它可以向固定线程池一样执行任务

distributeTaskForThreadPool(scheduledThreadPool);

//这是它的特殊之处，可以定时任务

scheduledThreadPool.schedule(

new Runnable() {

@Override

public void run() {

System.out.println("开始执行任务1");

}

},

5,

TimeUnit.SECONDS);

//每隔2秒再次重新执行任务

scheduledThreadPool.scheduleAtFixedRate(

new Runnable() {

@Override

public void run() {

System.out.println("开始执行任务2");

}

},

5,

3,

TimeUnit.SECONDS);

}

/\*\*

\* 创建只有一个线程的线程池，如果线程终止，

\* 他将会创建一个新的线程加入到池子中，这

\* 个线程池会保证池子中始终有一个线程

\*/

public static void displaySingleThreadPool() {

ExecutorService singleThreadPool = Executors.newSingleThreadExecutor();

distributeTaskForThreadPool(singleThreadPool);

}

/\*\*

\* 创建一个可根据需要创建线程的线程池，但是

\* 当先前创建的线程可得到时就会重用先前的线

\* 程，如果不存在可得到的线程，一个新的线程

\* 将被创建并被加入到池子中。60秒没有被用到

\* 的线程将被终止并从缓存中移除

\*/

public static void displayCachedThreadPool() {

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

distributeTaskForThreadPool(cachedThreadPool);

}

/\*\*

\* 创建一个带有固定线程的线程池

\*/

public static void displayThreadPool() {

// 创建一个带有4个固定线程的线程池

ExecutorService threadPool = Executors.newFixedThreadPool(4);

distributeTaskForThreadPool(threadPool);

}

/\*\*

\* 为线程池分配8个任务，使其驱动

\* @param threadPool

\*/

public static void distributeTaskForThreadPool(ExecutorService threadPool) {

// 让线程池驱动8个任务

for (int i = 1; i <= 8; i++) {

// 由于内部类里面不能放一个非final的变量，所以我把i的值赋予task

final int task = i;

threadPool.execute(new Runnable() {

@Override

public void run() {

System.out.println("我是" + Thread.currentThread().getName()

+ "，" + "拿到了第" + task + "个任务，我开始执行了");

}

});

}

}

}

```

示例二：

```

public class BankCount {

public synchronized void addMoney(int money){//存钱

System.out.println(Thread.currentThread().getName() + ">存入：" + money);

}

public synchronized void getMoney(int money){//取钱

System.out.println(Thread.currentThread().getName() + ">取钱：" + money);

}

}

```

```

public class BankTest {

public static void main(String[] args) {

final BankCount bankCount = new BankCount();

ExecutorService executor = Executors.newFixedThreadPool(10);

executor.execute(new Runnable() {//存钱线程

@Override

public void run() {

int i = 5;

while(i-- > 0){

bankCount.addMoney(200);

try {

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

});

Future<?> future = executor.submit(new Runnable() {//取钱线程

@Override

public void run() {

int i = 5;

while(i-- > 0){

try {

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

bankCount.getMoney(200);

}

}

});

try {

Object res = future.get();

System.out.println(res);

} catch (InterruptedException e) {

// 处理中断异常

e.printStackTrace();

} catch (ExecutionException e) {

// 处理无法执行任务异常

e.printStackTrace();

}finally{

// 关闭线程池

executor.shutdown();

}

}

}

```

## 第13节 equals与==的区别

- == 是一个运算符。

equals则是string对象的方法。

- java中 \*\*值类型\*\* 是存储在内存中的\*\*栈\*\*中。

而\*\*引用类型\*\*在栈中仅仅是存储引用类型变量的地址，而其本身则存储在\*\*堆\*\*中。所以字符串的内容相同，引用地址不一定相同，有可能创建了多个对象。

- ==操作比较的是两个变量的值是否相等，对于引用型变量表示的是两个变量在堆中存储的地址是否相同，即栈中的内容是否相同。

- equals将此字符串与指定的对象比较。当且仅当该参数不为 null，并且是与此对象表示相同字符序列的 String 对象时，结果才为 true。即堆中的内容是否相同。==比较的是2个对象的地址（栈中），而equals比较的是2个对象的内容（堆中）。所以当equals为true时，==不一定为true。

> 下面是String类equals方法源码，它复写了类 Object 中的 equals方法。

```

public boolean equals(Object anObject) {

965 if (this == anObject) {

966 return true;

967 }

968 if (anObject instanceof String) {

969 String anotherString = (String)anObject;

970 int n = value.length;

971 if (n == anotherString.value.length) {

972 char v1[] = value;

973 char v2[] = anotherString.value;

974 int i = 0;

975 while (n-- != 0) {

976 if (v1[i] != v2[i])

977 return false;

978 i++;

979 }

980 return true;

981 }

982 }

983 return false;

984 }

```

　　上面已经说到equals是比较两个对象的内容，我们可以看到方法中，先是比较两个String对象是否为同一对象，如果是就直接返回true（两个对象为同一对象那他们的内容必然相等）。

　　如果不是同一对象，先确定传入的对象是否是String类型，如果是，则比较两对象的字符序列（String类内部存储是用char[]实现的，可以查看源码了解），遍历过程中只要有一个字符不相同，就返回false，否则返回true。这里\*\*注意\*\*比较次数为第一个String对象的长度n，而不是传入的String对象参数的长度。

## 第14节 try catch finally，try里有return，finally还执行么？

\*Condition 1：\* 如果try中\*\*没有异常\*\*且try中\*\*有return\*\* （执行顺序）

```

try ---- finally --- return

```

\*Condition 2：\* 如果try中\*\*有异常\*\*并且try中\*\*有return\*\*

```

try----catch---finally--- return

```

总之 finally 永远执行！

\*Condition 3：\* try中有异常，try-catch-finally里都没有return ，finally 之后有个return

```

try----catch---finally

```

try中有异常以后，根据java的异常机制先执行catch后执行finally，此时错误异常已经抛出，程序因异常而终止，所以你的return是不会执行的

\*Condition 4：\* 当 try和finally中都有return时，finally中的return会覆盖掉其它位置的return（多个return会报unreachable code，编译不会通过）。

\*Condition 5：\* 当finally中不存在return，而catch中存在return，但finally中要修改catch中return 的变量值时

```

int ret = 0;

try{

throw new Exception();

}

catch(Exception e)

{

ret = 1; return ret;

}

finally{

ret = 2;

}

```

最后返回值是1，因为return的值在执行finally之前已经确定下来了

## 第15节 九种基本数据类型的大小，以及他们的封装类

　　 变量就是申请内存来存储值。也就是说，当创建变量的时候，需要在内存中申请空间。

内存管理系统根据变量的类型为变量分配存储空间，分配的空间只能用来储存该类型数据。

因此，通过定义不同类型的变量，可以在内存中储存整数、小数或者字符。

Java的两大数据类型:

(一).内置数据类型(基本数据类型)

　　1 六种数字类型 ( byte, short, int, long, float, double) + void

　　　　　　　　　 8 16 32 64 32 64 位

　　2 一种字符类型 char

　　　　　　　　　　16位Unicode字符

　　3 一种布尔型 boolean

　　　　　　　　　　1位

(二).引用数据类型

　　引用类型变量由类的构造函数创建，可以使用它们访问所引用的对象。这些变量在声明时被指定为一个特定的类型。变量一旦声明后，类型就不能被改变了。

　　对象、数组都是引用数据类型，所有引用类型的默认值都是null。

基本数据类型只能按值传递，而封装类按引用传递。

　　Void无返回值类型，作为伪类型对应类的对象，也被认为是 基本数据类型

## 第16节 从源码分析String、StringBuffer与StringBuilder区别和联系

- String

> public final class \*\*String\*\* extends Object

implements Serializable, Comparable<String>, CharSequence

String 类代表字符串。Java 程序中的所有字符串字面值（如 "abc" ）都作为此类的实例实现。

>字符串是常量；它们的值在创建之后不能更改。字符串缓冲区支持可变的字符串。因为 String \*\*对象是不可变的，所以可以共享\*\*。例如：

>

String str = "abc"

等效于：

>

char data[] = {'a', 'b', 'c'};

String str = new String(data);

>

下面给出了一些如何使用字符串的更多示例：

>

System.out.println("abc");

String cde = "cde";

System.out.println("abc" + cde);

String c = "abc".substring(2,3);

String d = cde.substring(1, 2);

>String 类包括的方法可用于检查序列的单个字符、比较字符串、搜索字符串、提取子字符串、创建字符串副本并将所有字符全部转换为大写或小写。大小写映射基于 Character 类指定的 Unicode 标准版。

>Java 语言提供对字符串串联符号（"+"）以及将其他对象转换为字符串的特殊支持。\*\*字符串串联是通过 StringBuilder\*\*（或 StringBuffer）类及其 \*\*append 方法\*\*实现的。字符串转换是通过 toString 方法实现的，该方法由 Object 类定义，并可被 Java 中的所有类继承。有关字符串串联和转换的更多信息，请参阅 Gosling、Joy 和 Steele 合著的 The Java Language Specification。

>除非另行说明，否则将 null 参数传递给此类中的构造方法或方法将抛出 NullPointerException。

>String 表示一个 UTF-16 格式的字符串，其中的增补字符 由代理项对 表示（有关详细信息，请参阅 Character 类中的 Unicode 字符表示形式）。索引值是指 char 代码单元，因此增补字符在 String 中占用两个位置。

>String 类提供处理 Unicode 代码点（即字符）和 Unicode 代码单元（即 char 值）的方法。

- StringBuffer

>public final class \*\*StringBuffer\*\* extends Object

implements Serializable, CharSequence

>\*\*线程安全的可变字符序列\*\*。一个类似于 String 的字符串缓冲区，但不能修改。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。

>可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。

>StringBuffer 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。

>例如，如果 z 引用一个当前内容为 "start" 的字符串缓冲区对象，则此方法调用 z.append("le") 会使字符串缓冲区包含 "startle"，而 z.insert(4, "le") 将更改字符串缓冲区，使之包含 "starlet"。

>通常，如果 sb 引用 StringBuilder 的一个实例，则 sb.append(x) 和 sb.insert(sb.length(), x) 具有相同的效果。

>当发生与源序列有关的操作（如源序列中的追加或插入操作）时，该类只在执行此操作的字符串缓冲区上而不是在源上实现同步。

>每个字符串缓冲区都有一定的容量。只要字符串缓冲区所包含的字符序列的长度没有超出此容量，就无需分配新的内部缓冲区数组。如果内部缓冲区溢出，则此容量自动增大。从 JDK 5 开始，为该类补充了一个单个线程使用的等价类，即 StringBuilder。与该类相比，通常应该优先使用 StringBuilder 类，因为它支持所有相同的操作，但由于它不执行同步，所以速度更快。

- StringBuilder

>public final class \*\*StringBuilder\*\* extends Object

implements Serializable, CharSequence

>\*\*一个可变的字符序列\*\*。此类提供一个与 StringBuffer 兼容的 API，但\*\*不保证同步\*\*。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。

>在 StringBuilder 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串生成器中。append 方法始终将这些字符添加到生成器的末端；而 insert 方法则在指定的点添加字符。

>将 StringBuilder 的实例用于多个线程是不安全的。如果需要这样的同步，则建议使用 StringBuffer。

1 .上String、StringBuffer与StringBuilder的原代码中的部分代码

```

//String

public final class String

implements java.io.Serializable, Comparable<String>, CharSequence

{

private final char value[];

/\*\*

\* Allocates a new {@code String} so that it represents the sequence of

\* characters currently contained in the character array argument. The

\* contents of the character array are copied; subsequent modification of

\* the character array does not affect the newly created string.

\*

\* @param value

\* The initial value of the string

\*/

public String(String original) {

this.value = Arrays.copyOf(value, value.length);

// 把传入构造函数original字符串切分成字符数组并赋给value[];

}

}

//StringBuffer

public final class StringBuffer extends AbstractStringBuilder

implements java.io.Serializable, CharSequence

{

/\*\*

\* Constructs a string buffer initialized to the contents of the

\* specified string. The initial capacity of the string buffer is

\* <code>16</code> plus the length of the string argument.

\*

\* @param str the initial contents of the buffer.

\* @exception NullPointerException if <code>str</code> is <code>null</code>

\*/

public StringBuffer(String str) {

super(str.length() + 16); //继承父类的构造器，并创建一个大小为str.length()+16的value[]数组

append(str); //将str切分成字符序列并加入到value[]中

}

public synchronized StringBuffer append(String str) {

super.append(str);

return this;

}

}

//StringBuilder

public final class StringBuilder

extends AbstractStringBuilder

implements java.io.Serializable, CharSequence

{

/\*\*

\* Constructs a string builder initialized to the contents of the

\* specified string. The initial capacity of the string builder is

\* <code>16</code> plus the length of the string argument.

\*

\* @param str the initial contents of the buffer.

\* @throws NullPointerException if <code>str</code> is <code>null</code>

\*/

public StringBuilder(String str) {

super(str.length() + 16);

append(str);

}

public StringBuilder append(String str) {

super.append(str);

return this;

}

}

//AbstractStringBuilder

abstract class AbstractStringBuilder implements Appendable, CharSequence {

/\*\*

\* The value is used for character storage.

\*/

char[] value;

/\*\*

\* Creates an AbstractStringBuilder of the specified capacity.

\*/

AbstractStringBuilder(int capacity) {

value = new char[capacity];

}

public AbstractStringBuilder append(String str) {

if (str == null) str = "null";

int len = str.length();

ensureCapacityInternal(count + len);

str.getChars(0, len, value, count);

count += len;

return this;

}

}

```

2 .我们可以看出在String中构造器传入的字符串对象被切分成字符序列，存放在其私有final的类变量value[]中。

new String("java")使得value[]={'j','a','v','a'}，之后这个String对象中的value[]再也不能改变了，只能读，所以是\*\*线程安全的\*\*。

StringBuffer和StringBuilder都继承了AbstractStringBuilder 这个抽象类，在他们的构造函数中传入字符串，会调用父类AbstractStringBuilder中对应的构造函数和append函数，父类构造函数新建一个长度为（字符串长度+末尾附加16个空元素）的数组然后通过父类append函数把字符串转换成字符数组value[]中（这里的char[] value 没有final，StringBuffer和StringBuilder中的\*\*字符串是可变的\*\*）。以后通过append()方法将新字符串加入value[]末尾。这样也就改变了value[]的内容和大小了。

StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，所以是\*\*线程安全的\*\*。

而\*\*StringBuilder不是线程安全的\*\*。

①

```

String s = "ja" +"va"

```

②

```

String s1="ja";

StringBuffer sb=new StringBuffer("va");

sb.append(s1);

```

③

```

String s1="ja";

String s2 = "va";

String s = s1 +s2

```

> 在编译阶段就能够确定的字符串常量，完全没有必要创建String或StringBuffer对象。直接使用字符串常量的"+"连接操作效率最高。

> 时间上 ①<③

>

StringBuffer对象的append效率要高于两个String对象的"+"连接操作。

时间上 ②<③

> 一般来说 执行时间上从快到慢 StringBuilder、StringBuffer、String

>

> 非多线程操作字符串缓冲区建议使用 StringBuilder。

- 抽象类和接口

> 抽象类中可以定义一些子类的公共方法，子类只需要增加新的功能，不需要重复写已经存在的方法；

>

> 而接口中只是对方法的申明和常量的定义。

>

## 第17节 多线程下生产者消费者问题的五种同步方法实现

　　生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。

　　生产者消费者模式的优点

- 解耦

- 支持并发

- 支持忙闲不均

　　解决方法可分为两类：

　　（1）用信号量和锁机制实现生产者和消费者之间的同步；

　　　　　- wait() / notify()方法

　　　　　- await() / signal()方法

　　　　　- BlockingQueue阻塞队列方法

　　　　　- Semaphore方法

　　（2）在生产者和消费者之间建立一个管道。（一般不使用，缓冲区不易控制、数据不易封装和传输）

　　　　　- PipedInputStream / PipedOutputStream

## 1 wait() / notify()方法实现

　　wait() / nofity()方法是Object里面的两个方法（[Object有哪些公用方法？](http://blog.csdn.net/amazing7/article/details/51219376)），所有Object的子类都可以使用这两个方法。

　　wait()方法：

　　在其他线程调用此对象的 notify() 方法前，导致当前线程等待。当前线程必须拥有此对象监视器。该线程发布对此监视器的所有权并等待，直到其他线程通过调用 notify 方法，或 notifyAll 方法通知在此对象的监视器上等待的线程醒来。然后该线程将等到重新获得对监视器的所有权后才能继续执行。

　　此方法只应由作为此对象监视器的所有者的线程来调用。

　　notify()方法：

　　唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。

直到当前线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。

　　此方法只应由作为此对象监视器的所有者的线程来调用。通过以下三种方法之一，线程可以成为此对象监视器的所有者：

　　－通过执行此对象的同步实例方法。

　　－通过执行在此对象上进行同步的 synchronized 语句的正文。

　　－对于 Class 类型的对象，可以通过执行该类的同步静态方法。

一次只能有一个线程拥有对象的监视器。

代码示例：

```

public class Test

{

private static Integer count = 0;

private final Integer FULL = 5;

private static String lock = "lock";

public static void main(String[] args)

{

Test t = new Test();

new Thread(t.new Producer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Producer()).start();

new Thread(t.new Consumer()).start();

}

class Producer implements Runnable

{

@Override

public void run()

{

for (int i = 0; i < 5; i++)

{

try {

Thread.sleep(1000);

} catch (InterruptedException e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

synchronized (lock)

{

while (count == FULL)

{

try {

lock.wait();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

count++;

System.out.println(Thread.currentThread().getName() + "produce:: " + count);

lock.notifyAll();

}

}

}

}

class Consumer implements Runnable

{

@Override

public void run()

{

for (int i = 0; i < 5; i++)

{

try {

Thread.sleep(1000);

} catch (InterruptedException e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

synchronized (lock)

{

while (count == 0)

{

try {

lock.wait();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

count--;

System.out.println(Thread.currentThread().getName()+ "consume:: " + count);

lock.notifyAll();

}

}

}

}

}

```

对象的监视器对锁对象的锁定（也就是代码中的lock对象），注意是调用锁对象的wait() / nofity()方法。

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160423102010734)

##2 await() / signal()方法

　　await()/signal()是对wait()/notify()的改进，功能更加强大，更适用于高级用户，synchronized是托管给JVM执行的，而lock是java写的控制锁的代码。

　　ReentrantLock（实现lock接口）相对于synchronized多了三个高级功能：

　　①等待可中断

　　　　在持有锁的线程长时间不释放锁的时候,等待的线程可以选择放弃等待.

> tryLock(long timeout, TimeUnit unit)

　　　②公平锁

　　　　按照申请锁的顺序来一次获得锁称为公平锁.synchronized的是非公平锁,ReentrantLock可以通过构造函数实现公平锁.

> new RenentrantLock(boolean fair)

　　公平锁和非公平锁。这2种机制的意思从字面上也能了解个大概：即对于多线程来说，公平锁会依赖线程进来的顺序，后进来的线程后获得锁。而非公平锁的意思就是后进来的锁也可以和前边等待锁的线程同时竞争锁资源。对于效率来讲，当然是非公平锁效率更高，因为公平锁还要判断是不是线程队列的第一个才会让线程获得锁。

　　③绑定多个Condition

　　　通过多次newCondition可以获得多个Condition对象,可以简单的实现比较复杂的线程同步的功能.通过await(),signal();

一般情况下都是用synchronized原语实现同步，除非下列情况使用ReentrantLock

　　　①某个线程在等待一个锁的控制权的这段时间需要中断

　　　②需要分开处理一些wait-notify，ReentrantLock里面的Condition应用，能够控制notify哪个线程

　　　 ③具有公平锁功能，每个到来的线程都将排队等候

更多了解：[线程同步的方法：sychronized、lock、reentrantLock分析](http://blog.csdn.net/amazing7/article/details/51219714)

下面是使用ReentrantLock来实现生产者消费者问题

代码示例：

```

public class Test {

private static Integer count = 0;//缓冲区

private final Integer FULL = 5;

final Lock lock = new ReentrantLock(); //获得可重入锁

final Condition put = lock.newCondition();

final Condition get = lock.newCondition();

public static void main(String[] args) {

Test t = new Test();

new Thread(t.new Producer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Producer()).start();

}

class Producer implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(1000);

} catch (InterruptedException e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

lock.lock();

try {

while (count == FULL) {

try {

put.await();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

count++;

System.out.println(Thread.currentThread().getName() + "produce:: " + count);

get.signal();

} finally {

lock.unlock();

}

}

}

}

class Consumer implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(1000);

} catch (InterruptedException e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

lock.lock();

try {

while (count == 0) {

try {

get.await();

} catch (Exception e) {

// TODO: handle exception

e.printStackTrace();

}

}

count--;

System.out.println(Thread.currentThread().getName()+ "consume:: " + count);

put.signal();

} finally {

lock.unlock();

}

}

}

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160423101751985)

##3 BlockingQueue阻塞队列方法

　　BlockingQueue 实现主要用于生产者-使用者队列，但它另外还支持 Collection 接口。是线程安全的，所有排队方法都可以使用内部锁或其他形式的并发控制来自动达到它们的目的。

　　BlockingQueue 方法以四种形式出现，对于不能立即满足但可能在将来某一时刻可以满足的操作，这四种形式的处理方式不同：第一种是抛出一个异常，第二种是返回一个特殊值（null 或 false，具体取决于操作），第三种是在操作可以成功前，无限期地阻塞当前线程，第四种是在放弃前只在给定的最大时间限制内阻塞。下表中总结了这些方法：

　　![这里写图片描述](http://img.blog.csdn.net/20160423103531099)

　　主要说说用于阻塞的那个方法

　　put()方法：将指定元素插入此队列中，将等待可用的空间（如果有必要）。

　　take()方法：获取并移除此队列的头部，在指定的等待时间前等待可用的元素（如果有必要）。

代码示例：

```

public class Test {

private static Integer count = 0;

final BlockingQueue<Integer> bq = new ArrayBlockingQueue<Integer>(5);//容量为5的阻塞队列

public static void main(String[] args) {

Test t = new Test();

new Thread(t.new Producer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Producer()).start();

}

class Producer implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(1000);

} catch (Exception e) {

e.printStackTrace();

}

try {

bq.put(1);

count++;

System.out.println(Thread.currentThread().getName() + "produce:: " + count);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

class Consumer implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(1000);

} catch (InterruptedException e1) {

e1.printStackTrace();

}

try {

bq.take();

count--;

System.out.println(Thread.currentThread().getName()+ "consume:: " + count);

} catch (Exception e) {

// TODO: handle exception

e.printStackTrace();

}

}

}

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160423104129979)

## 4 Semaphore方法实现同步

　　信号量（Semaphore）维护了一个许可集。在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore 只对可用许可的号码进行计数，并采取相应的行动。

Semaphore 通常用于限制可以访问某些资源（物理或逻辑的）的线程数目。

　　注意，调用 acquire() 时无法保持同步锁，因为这会阻止将项返回到池中。信号量封装所需的同步，以限制对池的访问，这同维持该池本身一致性所需的同步是分开的。

代码示例：

```

public class Test

{

int count = 0;

final Semaphore put = new Semaphore(5);//初始令牌个数

final Semaphore get = new Semaphore(0);

final Semaphore mutex = new Semaphore(1);

public static void main(String[] args) {

Test t = new Test();

new Thread(t.new Producer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Producer()).start();

}

class Producer implements Runnable

{

@Override

public void run()

{

for (int i = 0; i < 5; i++)

{

try

{

Thread.sleep(1000);

}

catch (Exception e)

{

e.printStackTrace();

}

try

{

put.acquire();//注意顺序

mutex.acquire();

count++;

System.out.println(Thread.currentThread().getName() + "produce:: " + count);

}

catch (Exception e)

{

e.printStackTrace();

}

finally

{

mutex.release();

get.release();

}

}

}

}

class Consumer implements Runnable

{

@Override

public void run()

{

for (int i = 0; i < 5; i++)

{

try

{

Thread.sleep(1000);

}

catch (InterruptedException e1)

{

e1.printStackTrace();

}

try

{

get.acquire();//注意顺序

mutex.acquire();

count--;

System.out.println(Thread.currentThread().getName()+ "consume:: " + count);

}

catch (Exception e)

{

e.printStackTrace();

}

finally

{

mutex.release();

put.release();

}

}

}

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160423110117024)

　　注意同步令牌（notFull.acquire()）必须在互斥令牌（mutex.acquire()）前面获得，如果先得到互斥锁再发生等待，会造成死锁。

##5 PipedInputStream / PipedOutputStream

　　这个类位于java.io包中，是解决同步问题的最简单的办法，一个线程将数据写入管道，另一个线程从管道读取数据，这样便构成了一种生产者/消费者的缓冲区编程模式。PipedInputStream/PipedOutputStream只能用于多线程模式，用于单线程下可能会引发死锁。

代码示例：

```

public class Test {

final PipedInputStream pis = new PipedInputStream();

final PipedOutputStream pos = new PipedOutputStream();

public static void main(String[] args) {

Test t = new Test();

new Thread(t.new Producer()).start();

new Thread(t.new Consumer()).start();

}

class Producer implements Runnable{

@Override

public void run() {

try {

pis.connect(pos);

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

try {

while(true){ //不断的产生数据

int n = (int)(Math.random()\*255);

System.out.println(Thread.currentThread().getName()+"produce::"+n);

pos.write(n);

pos.flush();

}

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}finally {

try {

pis.close();

pos.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

class Consumer implements Runnable{

@Override

public void run() {

int n;

try {

while(true){

n = pis.read();

System.out.println(Thread.currentThread().getName()+"consume::"+n);

}

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}finally{

try {

pis.close();

pos.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160423094138234)

　　从结果上看出也可以实现同步，但一般不使用，因为缓冲区不易控制、数据不易封装和传输。

## 第18节 实现多线程的两种方法

Java提供了两种创建线程方法：

-

通过实现Runable接口；

- 通过继承Thread类本身。

1 .声明实现 Runnable 接口的类，该类然后实现 run 方法。然后可以分配该类的实例，在创建 Thread 时作为一个参数来传递并启动。例如，计算大于某一规定值的质数的线程可以写成：

```

class PrimeRun implements Runnable {

long minPrime;

PrimeRun(long minPrime) {

this.minPrime = minPrime;

}

public void run() {

// compute primes larger than minPrime

. . .

}

}

```

然后，下列代码会创建并启动一个线程：

```

PrimeRun p = new PrimeRun(143);

new Thread(p).start();

```

２．将类声明为 Thread 的子类。该子类应重写 Thread 类的 run 方法。接下来可以分配并启动该子类的实例。

```

class PrimeThread extends Thread {

long minPrime;

PrimeThread(long minPrime) {

this.minPrime = minPrime;

}

public void run() {

// compute primes larger than minPrime

. . .

}

}

```

然后，下列代码会创建并启动一个线程：

```

PrimeThread p = new PrimeThread(143);

p.start();

```

　　当 Java 虚拟机启动时，通常都会有单个非守护线程（它通常会调用某个指定类的 main 方法）。Java 虚拟机会继续执行线程，直到下列任一情况出现时为止：

-

调用了 Runtime 类的 exit 方法，并且安全管理器允许退出操作发生。

- 非守护线程的所有线程都已停止运行，无论是通过从对 run 方法的调用中返回，还是通过抛出一个传播到 run 方法之外的异常。

３．使用和区别

　　Runable源码：

```

public interface Runnable {

/\*\*

\* When an object implementing interface <code>Runnable</code> is used

\* to create a thread, starting the thread causes the object's

\* <code>run</code> method to be called in that separately executing

\* thread.

\* <p>

\* The general contract of the method <code>run</code> is that it may

\* take any action whatsoever.

\*

\* @see java.lang.Thread#run()

\*/

public abstract void run();

}

```

　　Thread 类实现了 Runnable。激活的意思是说某个线程已启动并且尚未停止。此外，Runnable 为非 Thread 子类的类提供了一种激活方式。通过实例化某个 Thread 实例并将自身作为运行目标，就可以运行实现 Runnable 的类而无需创建 Thread 的子类。\*\*大多数情况下，如果只想重写 run() 方法，而不重写其他 Thread 方法\*\*，那么应使用 Runnable 接口。这很重要，因为\*\*除非程序员打算修改或增强类的基本行为，否则不应为该类创建子类\*\*。

　　继承Thread类实现多线程，要求放入多线程中的类不能继承其他类（Java的单继承特性），如果需要请用 Runnable 实现（接口可以多实现并不影响继承其他类）。

　　一个实现Runnable接口的类可以放在多个线程中执行，多个线程可以去执行同一资源；而继承Thread只能实现多个线程分别去处理自己的资源。（通过Runnable创建的多个线程可以由编程人员传入同一个Runnable对象,即执行同一个run方法，而通过Thread创建的多线程它们运行的都是自己的run方法）。

## 第19节 接口（Interface）与 抽象类 （Abstract）使用规则和区别

# 接口（Interface）

> 是\*\*抽象方法的集合\*\*，接口通常以interface来声明。

>

> 一个类通过继承接口的方式，从而来继承接口的抽象方法。

\*\*接口并不是类\*\*，编写接口的方式和类很相似，但是它们属于不同的概念。

>\*\*类描述对象的属性和方法。接口则包含类要实现的方法\*\*。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

>\*\*接口无法被实例化，但是可以被实现\*\*。一个实现接口的类，必须实现接口内所描述的所有方法（\*\*所有方法都是抽象的方法\*\*），否则就必须声明为抽象类。

>接口\*\*没有构造方法，支持多重继承\*\*，不能包含成员变量，除了static和final变量。

接口的申明方法

```

[可见度] interface 接口名称 [extends 其他的类名] {

//任何类型 final, static 字段

public [返回类型] [函数名] ();

// 抽象方法（所有都是 public）

}

```

　　接口是\*\*隐式抽象\*\*的，当声明一个接口和接口中方法的时候，不必使用abstract关键字。接口中的方法都是公有的。

　　类使用implements关键字实现接口。在类声明中，Implements关键字放在class声明后面。

```

... implements 接口名称[, 其他接口, 其他接口..., ...] ...

```

重写接口中的方法时，需注意：

　　类在实现接口的方法时，\*\*不能抛出强制性异常\*\*，只能在接口中，或者继承接口的抽象类　中抛出该强制性异常。

　　类在重写方法时要\*\*保持一致的方法名\*\*，并且应该保持相同或者相兼容的返回值类型。

　　一个类只能继承一个类，但是\*\*能实现多个接口\*\*，同时一个接口能继承另一个接口。

　　\*\*一个接口能继承另一个接口\*\*，和类之间的继承方式比较相似。接口的继承使用　extends　关键字，\*\*子接口继承父接口的方法\*\*。

# 抽象类 （Abstract）

> 　在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，\*\*并不是所有的类都是用来描绘对象的\*\*，如果一个类中\*\*没有包含足够的信息\*\*来描绘一个具体的对象，这样的类就是抽象类。

>　　抽象类除了\*\*不能实例化对象\*\*之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。抽象类必须被继承，才能被使用。（它没有足够信息描绘一个对象，只有等子类来继承完善后才能使用）

在Java语言中使用abstract class来定义抽象类。

```

public abstract class Employee

{

／／成员变量

／／成员方法，可以是抽象的或实现的（可以没有抽象方法）

［访问权限修饰符］ abstract ［返回类型］　［函数名］();

｝

```

声明抽象方法会造成以下两个结果：

　　如果一个类包含抽象方法，那么该类必须是抽象类。但是一个抽象类可以没有抽象方法，只是把类申明为抽象的。

　　任何子类\*\*必须重写父类的抽象方法\*\*（一句话抽象方法都要重写，只不过接口的方法全部都是抽象的而已），或者声明自身为抽象类。

# 区别

　　接口不是类，抽象类是一个功能不齐全的类，都不能实例化对象。

　　一个类可以实现（implements）多个接口。一个类只能继承（extends）一个抽象类。

　　 接口没有构造函数，所有方法都是 public abstract的，一般不定义成员变量。（所有的成员变量都是 static final ，而且必须显示初始化）。

　　抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

　　一个实现接口的类，必须实现接口内所描述的所有方法（\*\*所有方法都是抽象的方法\*\*），否则就必须声明为抽象类。

　　如果一个类包含抽象方法，那么该类必须是抽象类。任何子类必须重写父类的抽象方法，或者声明自身为抽象类。

## 第20节 方法锁、对象锁和类锁的意义和区别

> 首先的明白java中锁的机制

>

> synchronized

> 　　在修饰代码块的时候需要一个reference对象作为锁的对象.

　　在修饰方法的时候默认是当前对象作为锁的对象.

　　在修饰类时候默认是当前类的Class对象作为锁的对象.

> [线程同步的方法：sychronized、lock、reentrantLock分析](http://blog.csdn.net/amazing7/article/details/51219714)

#方法锁（synchronized修饰方法时）

通过在方法声明中加入 synchronized关键字来声明 synchronized 方法。

synchronized 方法控制对类成员变量的访问：

　　每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，\*\*方法一旦执行，就独占该锁\*\*，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中\*\*至多只有一个\*\*处于可执行状态，从而有效避免了类成员变量的访问冲突。

#　对象锁（synchronized修饰方法或代码块）

　　当一个对象中有synchronized method或synchronized block的时候调用此对象的同步方法或进入其同步区域时，就必须先获得对象锁。如果此对象的对象锁已被其他调用者占用，则需要等待此锁被释放。（方法锁也是对象锁）

　　java的所有对象都含有1个互斥锁，这个锁由JVM自动获取和释放。线程进入synchronized方法的时候获取该对象的锁，当然如果已经有线程获取了这个对象的锁，那么当前线程会等待；synchronized方法正常返回或者抛异常而终止，JVM会自动释放对象锁。这里也体现了用synchronized来加锁的1个好处，\*\*方法抛异常的时候，锁仍然可以由JVM来自动释放。\*\*

对象锁的两种形式：

```

public class Test

{

// 对象锁：形式1(方法锁)

public synchronized void Method1()

{

System.out.println("我是对象锁也是方法锁");

try

{

Thread.sleep(500);

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

// 对象锁：形式2（代码块形式）

public void Method2()

{

synchronized (this)

{

System.out.println("我是对象锁");

try

{

Thread.sleep(500);

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

｝

```

# 类锁(synchronized 修饰静态的方法或代码块)

　　由于一个class不论被实例化多少次，其中的静态方法和静态变量在内存中都\*\*只有一份\*\*。所以，一旦一个静态的方法被申明为synchronized。此类所有的实例化对象在调用此方法，共用同一把锁，我们称之为类锁。

　　\*\*对象锁是用来控制实例方法之间的同步，类锁是用来控制静态方法（或静态变量互斥体）之间的同步。\*\*

　　类锁只是一个概念上的东西，并不是真实存在的，它只是用来帮助我们理解锁定实例方法和静态方法的区别的。

　　java类可能会有很多个对象，但是只有1个Class对象，也就是说类的不同实例之间共享该类的Class对象。Class对象其实也仅仅是1个java对象，只不过有点特殊而已。由于每个java对象都有1个互斥锁，而类的静态方法是需要Class对象。所以所谓的类锁，不过是Class对象的锁而已。获取类的Class对象有好几种，最简单的就是［类名.class］的方式。

```

public class Test

{

　　 // 类锁：形式1

public static synchronized void Method1()

{

System.out.println(＂我是类锁一号＂);

try

{

Thread.sleep(500);

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

// 类锁：形式2

public void Method２()

{

synchronized (Test.class)

{

System.out.println(＂我是类锁二号＂);

try

{

Thread.sleep(500);

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

｝

```

## 第21节 的四种引用，强弱软虚，用到的场景

> 从JDK1.2版本开始，把对象的引用分为四种级别，从而使程序能更加灵活的控制对象的生命周期。这四种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

1 . 强引用(StrongReference)

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。如代码String s=”abc”中变量s就是字符串对象”abc”的一个强引用。只要你给强引用对象s赋空值null,该对象就可以被垃圾回收器回收。因为该对象此时不再含有其他强引用。

2 . 弱引用(WeakReference)

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象（s），不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象，弱引用非常适合存储元数据。另一个使用弱引用的例子是WeakHashMap，它是除HashMap和TreeMap之外，Map接口的另一种实现。WeakHashMap有一个特点：map中的键值(keys)都被封装成弱引用，也就是说一旦强引用被删除，WeakHashMap内部的弱引用就无法阻止该对象被垃圾回收器回收。

如下代码创建弱引用：

```

Counter counter = new Counter(); // strong reference - line 1

WeakReference<Counter> weakCounter = new WeakReference<Counter>(counter); //weak reference

counter = null; // now Counter object is eligible for garbage collection

```

3 . 软引用(SoftReference)

如果一个对象（如 s）只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

可以使用如下代码创建软引用：

```

Counter prime = new Counter(); // prime holds a strong reference

SoftReference soft = new SoftReference(prime) ; //soft reference variable has SoftReference to Counter Object created at line 2

prime = null; // now Counter object is eligible for garbage collection but only be collected when JVM absolutely needs memory

```

4 . 虚引用(PhantomReference)

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用 \*\*必须\*\* 和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

如下代码创建虚引用：

```

DigitalCounter digit = new DigitalCounter(); // digit reference variable has strong reference – line 3

PhantomReference phantom = new PhantomReference(digit); // phantom reference to object created at line 3

digit = null;

```

弱引用、软引用\*\*可以和\*\*一个引用队列（ReferenceQueue）联合使用，如果其所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

在创建任何弱引用、软引用和虚引用的过程中你可以通过如下代码提供引用队列ReferenceQueue。

```

ReferenceQueue refQueue = new ReferenceQueue(); //reference will be stored in this queue for cleanup

DigitalCounter digit = new DigitalCounter();

PhantomReference<DigitalCounter> phantom = new PhantomReference<DigitalCounter>(digit, refQueue);

```

## 第22节 线程同步的方法：sychronized、lock、reentrantLock分析

>　　如果你向一个变量写值，而这个变量接下来可能会被另一个线程所读取，或者你从一个变量读值，而它的值可能是前面由另一个线程写入的，此时你就必须使用同步。

- \*\*sychronized\*\*

　 Java语言的\*\*关键字\*\*，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码，它是在 \*\*软件层面依赖JVM实现同步\*\*。

　　 synchronized 方法或语句的使用提供了对与每个对象相关的隐式监视器锁的访问，但却强制所有锁获取和释放均要出现在一个块结构中：当获取了多个锁时，它们必须以相反的顺序释放，且必须在与所有锁被获取时相同的词法范围内释放所有锁。

> 　　通过在方法声明中加入 synchronized关键字来声明 synchronized 方法。

>

　　synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能

执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行

状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有一个能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突（只要所有可能访问类成员变量的方法均被声明为 synchronized）

>

>

　　synchronized 方法的缺陷：若将一个大的方法声明为synchronized 将会大大影响效率，典型地，若将线程类的方法 run() 声明为　synchronized ，由于在线程的整个生命期内它一直在运行，因此将导致它对本类任何 synchronized 方法的调用都永远不会成功。

　　解决synchronized 方法的缺陷

> 通过 synchronized关键字来声明synchronized 块。

```

synchronized(lock) {

// 访问或修改被锁保护的共享状态

}

```

> 其中的代码必须获得对象 syncObject （类实例或类）的锁方能执行。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。

　　当两个并发线程访问同一个对象中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

　　当一个线程访问对象的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该对象中的非synchronized(this)同步代码块。其他线程对对象中所有其它synchronized(this)同步代码块的访问将被阻塞。

　　如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续） synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。

　　在修饰代码块的时候需要一个reference对象作为锁的对象.

　　在修饰方法的时候默认是当前对象作为锁的对象.

　　在修饰类时候默认是当前类的Class对象作为锁的对象.

- \*\*lock\*\*

　　Lock 接口实现提供了比使用 synchronized 方法和语句可获得的更广泛的锁定操作。此实现允许更灵活的结构，可以具有差别很大的属性，可以支持多个相关的 Condition 对象。在硬件层面依赖特殊的CPU指令实现同步更加灵活。

> 什么是Condition ？

> Condition 接口将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set（wait-set）。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。

　　虽然 synchronized 方法和语句的范围机制使得使用监视器锁编程方便了很多，而且还帮助避免了很多涉及到锁的常见编程错误，但有时也需要以更为灵活的方式使用锁。例如，某些遍历并发访问的数据结果的算法要求使用 "hand-over-hand" 或 "chain locking"：获取节点 A 的锁，然后再获取节点 B 的锁，然后释放 A 并获取 C，然后释放 B 并获取 D，依此类推。Lock 接口的实现允许锁在不同的作用范围内获取和释放，并允许以任何顺序获取和释放多个锁，从而支持使用这种技术。

　　随着灵活性的增加，也带来了更多的责任。不使用块结构锁就失去了使用 synchronized 方法和语句时会出现的锁自动释放功能。在大多数情况下，应该使用以下语句：

```

　　　Lock l = ...; //lock接口的实现类对象

l.lock();

try {

// access the resource protected by this lock

} finally {

l.unlock();

}

```

在java.util.concurrent.locks包中有很多Lock的实现类，常用的有ReentrantLock、ReadWriteLock（实现类ReentrantReadWriteLock）.它们是具体实现类，不是java语言关键字。

-

\*\*ReentrantLock\*\*

　　一个可重入的互斥锁 Lock，它具有与使用 synchronized 方法和语句所访问的\*\*隐式监视器锁\*\*相同的一些基本行为和语义，但功能更强大。

　　最典型的代码如下：

```

class X {

private final ReentrantLock lock = new ReentrantLock();

// ...

public void m() {

lock.lock(); // block until condition holds

try {

// ... method body

} finally {

lock.unlock()

}

}

}

```

> 重入性：指的是同一个线程多次试图获取它所占有的锁，请求会成功。当释放锁的时候，直到重入次数清零，锁才释放完毕。

>

　　ReentrantLock 的lock机制有2种，\*\*忽略中断锁\*\*和\*\*响应中断锁\*\*，这给我们带来了很大的灵活性。比如：如果A、B 2个线程去竞争锁，A线程得到了锁，B线程等待，但是A线程这个时候实在有太多事情要处理，就是 一直不返回，B线程可能就会等不及了，想中断自己，不再等待这个锁了，转而处理其他事情。这个时候ReentrantLock就提供了2种机制，第一，B线程中断自己（或者别的线程中断它），但是ReentrantLock 不去响应，继续让B线程等待，你再怎么中断，我全当耳边风（synchronized原语就是如此）；第二，B线程中断自己（或者别的线程中断它），ReentrantLock 处理了这个中断，并且不再等待这个锁的到来，完全放弃。

　　ReentrantLock相对于synchronized多了三个高级功能：

　　①等待可中断

　　　　在持有锁的线程长时间不释放锁的时候,等待的线程可以选择放弃等待.

```

tryLock(long timeout, TimeUnit unit)

```

　　　②公平锁

　　　　按照申请锁的顺序来一次获得锁称为公平锁.synchronized的是非公平锁,ReentrantLock可以通过构造函数实现公平锁.

```

new RenentrantLock(boolean fair)

```

　　公平锁和非公平锁。这2种机制的意思从字面上也能了解个大概：即对于多线程来说，公平锁会依赖线程进来的顺序，后进来的线程后获得锁。而非公平锁的意思就是后进来的锁也可以和前边等待锁的线程同时竞争锁资源。对于效率来讲，当然是非公平锁效率更高，因为公平锁还要判断是不是线程队列的第一个才会让线程获得锁。

　　③绑定多个Condition

　　　通过多次newCondition可以获得多个Condition对象,可以简单的实现比较复杂的线程同步的功能.通过await(),signal();

- \*\*synchronized和lock的用法与区别\*\*

　synchronized是托管给JVM执行的，而lock是java写的控制锁的代码。

　synchronized原始采用的是CPU悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低。

　Lock用的是乐观锁方式。每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。

　ReentrantLock必须在finally中释放锁，否则后果很严重，编码角度来说使用synchronized更加简单，不容易遗漏或者出错。

　ReentrantLock提供了可轮询的锁请求，他可以尝试的去取得锁，如果取得成功则继续处理，取得不成功，可以等下次运行的时候处理，所以不容易产生死锁，而synchronized则一旦进入锁请求要么成功，要么一直阻塞，所以更容易产生死锁。

　synchronized的话，锁的范围是整个方法或synchronized块部分；而Lock因为是方法调用，可以跨方法，灵活性更大

　一般情况下都是用synchronized原语实现同步，除非下列情况使用ReentrantLock

　　　①某个线程在等待一个锁的控制权的这段时间需要中断

　　　②需要分开处理一些wait-notify，ReentrantLock里面的Condition应用，能够控制notify哪个线程

　　　 ③具有公平锁功能，每个到来的线程都将排队等候

## 第23节 集合框架的层次结构和使用规则梳理

> 在Java语言中，Java语言的设计者对常用的数据结构和算法做了一些规范（接口）和实现（具体实现接口的类）。所有抽象出来的数据结构和操作（算法）统称为Java集合框架（JavaCollectionFramework）。

>

Java程序员在具体应用时，不必考虑数据结构和算法实现细节，只需要用这些类创建出来一些对象，然后直接应用就可以了，这样就大大提高了编程效率。

#概述

- 什么是框架？

> 类库的集合

- 什么是集合？

> 存放数据的容器

- 集合框架用来干什么？

> 用来表示和操作的统一的架构

集合框架包含了两部分：一部分是类，一部分是接口（用来规范相似功能的类）

主要结构图：

![这里写图片描述](http://img.blog.csdn.net/20160427155321833)

　　Java集合工具包位于Java.util包下，包含了很多常用的数据结构，如数组、链表、栈、队列、集合、哈希表等。

　　Java集合框架的使用重点：List列表、Set集合、Map映射、迭代器（Iterator、Enumeration）、工具类（Arrays、Collections）。

#Collection

　　 Collection是List、Set等集合高度抽象出来的接口，它包含了这些集合的基本操作，实现该接口及其子接口的所有类都可应用clone()方法，并是序列化类。

主要实现关系层次图：

　　　![这里写图片描述](http://img.blog.csdn.net/20160427155936816)

Collection的常用方法：

- 添加

boolean add(E e);

boolean addAll(Collection<> c)

- 删除

boolean remove(E e);

boolean removeAll(Collection<> c) ;

void clear();

- 判断

boolean contains(Object o)

boolean containsAll(Collection<> c)

boolean isEmply();

- 获取

int size();

Iterator<E> iterator();

- 其他

boolean retainAll(Collection coll);

Object[] toArray();

##List

　　List接口通常表示一个列表，\*\*可随机访问\*\*包含的元素，元素是\*\*有序的\*\*，可在任意位置增、删元素，不管访问多少次，元素位置不变，\*\*允许重复\*\*元素。

　　用Iterator实现单向遍历，也可用ListIterator实现双向遍历。

List特有的常见方法：

- 添加

void add(index,element);

void add(index,collection);

- 删除

Object remove(index);

- 获取

Object get(index);

int indexOf(object);

int lastIndexOf(object);

Last subList(from,to);

- 修改

Object set(index,element);

###LinkedList

　　内部是\*\*链表\*\*数据结构，\*\*随机访问很慢\*\*，\*\*增删操作很快\*\*，不耗费多余资源，\*\*允许null元素\*\*，\*\*非线程安全\*\* ．如果多个线程同时访问一个List，则必须自己实现访问同步。

```

List list = Collections.synchronizedList(new LinkedList(...));

```

###ArrayList

　　内部是\*\*数组\*\*数据结构，元素顺序存储。新增元素改变List大小时，内部会新建一个数组，在将添加元素前将所有数据拷贝到新数组中 。\*\*随机访问很快\*\*，\*\*删除非头尾元素慢，新增元素慢\*\*而且费资源 。较适用于无频繁增删的情况，可增长数组，50%延长。比数组效率低，如果不是需要可变数组，可考虑使用数组，\*\*非线程安全\*\*。

###Vector

　　另一种ArrayList，具备ArrayList的特性。所有方法都是\*\*线程安全\*\*的（双刃剑，和ArrayList的主要区别） 比ArrayList效率低。

####Stack

　　LIFO的数据结构，Stack继承自Vector，也是\*\*同步的\*\*。

##Set

　　通常表示一个集合，接口中的方法和Collection中的一致。是一个不允许重复元素的Collection，\*\*可以有一个空元素\*\*，\*\*不可随机\*\*访问包含的元素，只能用Iterator实现单向遍历，Set\*\*不同步\*\*。

###HashSet

　　内部数据结构是哈希表（HashMap），元素是\*\*无序的\*\*，迭代访问元素的顺序和加入的顺序不同 .多次迭代访问，元素的顺序可能不同，\*\*非线程安全\*\*。

####LinkedHashSet

　　基于HashMap和链表的Set实现，迭代访问元素的顺序和加入的顺序相同，多次迭代访问，元素的\*\*顺序不变\*\*。

###SortedSet

　　加入SortedSet的所有元素必须实现Comparable接口，元素是\*\*有序的\*\*。

####TreeSet

　　基于TreeMap实现的SortedSet，可以对Set集合中的元素进行排序，排序后按\*\*升序\*\*排列元素，\*\*非线程安全\*\*。

　　两种排序方式：

- 让元素自身具备比较功能，\*\*元素\*\*就需要实现Compareable方法覆盖compareTo方法。

- 让集合自身具备比较功能，定义一个类实现Comparator接口，覆盖compare方法。

#Map

　　是一个映射接口，Map没有继承Collection接口，Map提供key到value的映射。一个Map中不能包含相同的key，每个key只能映射一个value。

　　抽象类AbstractMap通过适配器模式实现了Map接口中的大部分函数，TreeMap、HashMap、WeakHashMap等实现类都通过继承AbstractMap来实现，但不常用的HashTable直接实现了Map接口。

　　实现关系层次图：

![这里写图片描述](http://img.blog.csdn.net/20160427163254642)

map中常用的方法：

- 添加

value put(key,value):返回前一个和key关联的值，如果没有则返回null

- 删除

void clear():清空map集合。

value remove(key):根据指定的key删除这个键值对

- 判断

boolean containKey(key)

boolean containValue(value)

boolean isEmpty();

- 获取

value get(key);通过键获取值，如果没有该键返回null，当然可以通过返回null来判断是否包含指定键

int size()获取键值对个数。

取出Map集合中的所有值的方法：

- 通过keySet方法获取map中所有的键所在的Set集合，再通过Set的迭代器获取到每一个键，再对每一个键获通过map集合的get方法取其对应值即可。

```

Map map = new LinkedHashMap();

map.put("1", "第一个数");

map.put("2", "第二个数");

map.put("3", "第三个数");

for (Object obj : map.keySet()) {

String key = (String) obj;

String value = (String) map.get(key);

}

```

- 通过map集合的entryset方法也可取出map集合当中所有的键值对，返回的就是Map.Entry对象，再通过Map.Entry对象的getkey和getvalue方法就可以取出对应的键和值。

```

Map map = new LinkedHashMap();

map.put("1", "one");

map.put("2", "two");

map.put("3", "three");

for (Object obj : map.entrySet()) {

Entry entry = (Entry) obj;

String key = (String) entry.getKey();

String value = (String) entry.getValue();

}

```

- 通过Iterator遍历keySet或entryset

```

Map map = new HashMap();

map.put("1", "one");

map.put("2", "two");

map.put("3", "three");

Set set = map.keySet();

Iterator it = set.iterator();

while (it.hasNext()) {

String key = (String) it.next();

String value = (String) map.get(key);

}

```

```

Map map = new HashMap();

map.put("1", "one");

map.put("2", "two");

map.put("3", "three");

Set set = map.entrySet();

Iterator it = set.iterator();

while (it.hasNext()) {

Entry entry = (Entry) it.next();

String key = (String) entry.getKey();

String value = (String) entry.getValue();

}

```

##Hashtable

　　用作键的对象必须实现了hashcode()、equals()方法，也就是说只有Object及其子类可用作键，Hashtable继承于Dictionary字典，实现Map接口。\*\*键、值都不能是空对象\*\*，多次访问，映射元素的\*\*顺序相同\*\* ，\*\*线程安全\*\*。

###Properties

　　继承于Hashtable，\*\*键和值都是字符串\*\*。

##HashMap

　　内部结构是哈希表，\*\*键和值都可以是空对象，不保证映射的顺序\*\*，多次访问，映射元素的顺序可能不同，\*\*非线程安全\*\*。

###LinkedHashMap

　　继承于HashMap ，多次访问，映射元素的\*\*顺序是相同的\*\*，性能比HashMap差。

##WeakHashMap

　　当某个键不再正常使用时，垃圾收集器会移除它，即便有映射关系存在，\*\*非线程安全\*\*。

##SortedMap

　　键按\*\*升序\*\*排列，所有键都必须实现Comparable接口。

###TreeMap

　　内部结构是二叉树（基于红黑树实现），\*\*不同步\*\*，可以对Map集合中的键进行排序。

#Iterator

　　Iterator是遍历\*\*集合\*\*的迭代器（不能遍历Map，只用来遍历Collection），Collection的实现类都实现了iterator()函数，它返回一个Iterator对象。

　　ListIterator则专门用来遍历List。

　　而Enumeration则是JDK1.0时引入的，作用与Iterator相同，但它的功能比Iterator要少，它只能在Hashtable、Vector和Stack中使用。

　　Arrays和Collections是用来操作数组、集合的两个工具类。

#使用技巧

- 需要\*\*唯一\*\*用Set

　　　　　　　且需要指定顺序：TreeSet

　　　　　　　　　　　不需要：HashSet

　　　　　　　得到与存储一致的顺序：LinkedHashSet

- 不需要\*\*唯一\*\*用List

　　　　　　　需要频繁增删动作用：LinkedList

　　　　　　　不需要：ArrayList

- 如果涉及到堆栈、队列等操作，应该考虑用List；

- 对于需要快速插入，删除元素，应该使用LinkedList；

- 如果需要快速随机访问元素，应该使用ArrayList。

- 如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高。

- 如果多个线程可能同时操作一个类，应该使用同步的类。 要特别注意对哈希表的操作，作为key的对象要正确复写equals和hashCode方法。 尽量返回接口而非实际的类型，如返回List而非ArrayList，这样如果以后需要将ArrayList换成LinkedList时，客户端代码不用改变，这就是针对抽象编程。

## 第24节 面向对象的三个特征与含义

> 对象

> 是类的一个实例（对象不是找个女朋友），有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。

>

> 类

> 是一个模板，它描述一类对象的行为和状态。

1 . \*\*封装性\*\*

　　将对象的状态信息尽可能的隐藏在对象内部，只保留有限的接口和方法与外界进行交互，从而避免了外界对对象内部属性的破坏。

　　Java中使用访问控制符来保护对类、变量、方法和构造方法的访问。

　　Java支持4种不同的访问权限。

　　默认的，也称为default，在同一包内可见，不使用任何修饰符。

　　私有的，以private修饰符指定，在同一类内可见。

　　共有的，以public修饰符指定，对所有类可见。

　　受保护的，以protected修饰符指定，对同一包内的类和所有子类可见。

２. \*\*继承\*\*

　　 java通过继承创建分等级层次的类，可以理解为一个对象从另一个对象获取属性的过程。

　　 类的继承是单一继承，也就是说，一个子类只能拥有一个父类

　　 下面的做法是不合法的：

```

public class extends Animal, Mammal{}

```

　　但是我们可以用多继承接口来实现,　如：

```

public class Apple extends Fruit implements Fruit1, Fruit2{}

```

　　继承中最常使用的两个关键字是extends（用于基本类和抽象类）和implements（用于接口）。

　　 注意：子类拥有父类所有的成员变量，但对于父类private的成员变量却没有访问权限，这保障了父类的封装性。

　　 下面是使用关键字extends实现继承。

```

public class Animal{

}

public class Mammal extends Animal{

}

public class Reptile extends Animal{

}

public class Dog extends Mammal{

}

```

　　通过使用关键字extends，子类可以继承父类所有的方法和属性，但是无法使用 private(私有) 的方法和属性。

我们通过使用instanceof 操作符能够确定一个对象是另一个对象的一个分类。

```

public class Dog extends Mammal{

public static void main(String args[]){

Animal a = new Animal();

Mammal m = new Mammal();

Dog d = new Dog();

System.out.println(m instanceof Animal);

System.out.println(d instanceof Mammal);

System.out.println(d instanceof Animal);

}

}

```

结果如下：

```

true

true

true

```

Implements关键字使用在类继承接口的情况下， 这种情况不能使用关键字extends。

```

public interface Animal {}

public class Mammal implements Animal{

}

public class Dog extends Mammal{

}

```

可以使用 instanceof 运算符来检验Mammal和dog对象是否是Animal类的一个实例。

```

interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{

public static void main(String args[]){

Mammal m = new Mammal();

Dog d = new Dog();

System.out.println(m instanceof Animal);

System.out.println(d instanceof Mammal);

System.out.println(d instanceof Animal);

}

}

```

运行结果如下：

```

true

true

true

```

３．\*\*多态\*\*

　　多态是同一个行为具有多个不同表现形式或形态的能力。

　　多态性是\*\*对象多种表现形式的体现\*\*

　　比如：我到宠物店说"请给我一只宠物"，服务员给我小猫、小狗或者蜥蜴都可以，我们就说"宠物"这个对象就具备多态性。

例子

```

public interface Vegetarian{}

public class Animal{}

public class Deer extends Animal implements Vegetarian{}

```

因为Deer类具有多重继承，所以它具有多态性。

访问一个对象的唯一方法就是通过引用型变量 (编译时变量)。

引用型变量只能有一种类型，一旦被声明，引用型变量的类型就不能被改变了。

引用型变量不仅能够被重置为其他对象，前提是这些对象没有被声明为final。还可以引用和它类型相同的或者相兼容的对象。它可以声明为类类型或者接口类型。

```

Deer d = new Deer();

Animal a = d;

Vegetarian v = d;

Object o = d;

```

所有的引用型变量d,a,v,o都指向堆中相同的Deer对象。

我们来看下面这个例子：

```

public class Animal {

public String name = "父类name";

public void move(){

System.out.println("父类move");

}

public void content(){

System.out.println("父类content");

}

}

```

```

public class Bird extends Animal{

public String name = "子类name";

@Override

public void move() {

// TODO Auto-generated method stub

System.out.println("子类move");

}

public void content(){

System.out.println("子类content");

}

}

```

```

public class Test {

public static void main(String[] args) {

Animal a = new Animal();

System.out.println(a.name);

a.move();

a.content();

System.out.println("----------------------");

Animal b = new Bird(); //向上转型由系统自动完成

//编译时变量 运行时变量

System.out.println(b.name);

b.move();

b.content();

System.out.println("----------------------");

Bird c = new Bird();

System.out.println(c.name);

c.move();

c.content();

}

}

```

运行结果：

```

父类name

父类move

父类content

----------------------

父类name

子类move

子类content

----------------------

子类name

子类move

子类content

```

说明：Bird类继承Animal并重写了其方法。

　　 因为Animal b = new Bird()，编译时变量和运行时变量不一样，所以多态发生了。可以从最后的运行结果中看出，调用了\*\*父类的成员变量name和子类重写后的两个方法\*\*。

　　 上面继承说了，子类可以调用父类所有非private的方法和属性。因为name是一个String的对象，\*\*与方法不同，对象的域不具有多态性\*\*。通过引用变量来访问其包含的实例变量时，系统总是视图访问它编译时类型所定义的变量，而不是他运行时类型所定义的变量。

　　那么问题来了，如果我们把Animal的成员变量换成private，那会不会去调用Bird类的成员变量name来打印输出呢？

　　![这里写图片描述](http://img.blog.csdn.net/20160413153304267)

也就是说　\*\*系统访问的始终是去访问编译时类型所定义的变量\*\*。

> 　　重写定义：子类对父类的允许访问的方法的实现过程进行重新编写！返回值和形参都不能改变。即外壳不变，核心重写！

## 第25节 static的作用和意义

# static 静态修饰符

　　在程序中任何变量或者代码都是在编译时由系统自动分配内存来存储的。static修饰符表示静态的，在类加载时Jvm会把它放到\*\*方法区\*\*，被本类以及本类的所有实例所共用。在编译后所分配的内存会\*\*一直存在\*\*，直到程序退出内存才会释放这个空间。如果一个被所有实例共用的方法被申明为static，那么就可以节省空间，不用每个实例初始化的时候都被分配到内存。

> java类被加载过程

>类装载器把一个类装入Java虚拟机中，要经过三个步骤来完成：

>①加载（以二进制形式来生成Class对象）

>②链接（又分为验证、准备和解析）

>　　　　校验：检查导入类或接口的二进制数据的正确性；

>　　　　准备：给类的静态变量分配并初始化存储空间；

>　　　　解析：将符号引用转成直接引用；

>③初始化（激活类的静态变量和静态代码块、初始化Java代码）

##静态变量

　　Static关键字修饰成员变量被称为静态变量（也叫作类变量，同时　局部变量也能被声明为static），静态变量在内存中只有一个拷贝（节省内存，方便对象之间共享值），JVM只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问（当然也可以通过对象来访问）。因为静态变量被类的所有实例共用，所以非线程安全的。

　　未被Static修饰的成员变量叫作实例变量，每创建一个实例，就会为实例变量分配一次内存，实例变量在内存中可以有多个拷贝（但互相不影响，更加灵活）。

```

//静态变量的申明

private static int count = 0;

public static String str;

```

## 静态方法

　　静态方法可以直接通过类名调用，任何的实例也都可以调用。

　　\*\*只能\*\*访问所属类的静态成员变量和成员方法，静态方法中也不能用this和super关键字。

　　类似于静态变量，静态方法也属于类，不属于实例的。

```

//静态方法的申明

public static void s(int param) {

......

}

```

## 静态代码块

　　静态代码块就是在类加载器加载对象时，要执行的一组语句。静态块只会在类加载到内存中的时候执行一次，位置可以随便放，如果static代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。

```

static{

//在类被加载的时候用于初始化资源，仅能访问静态变量和静态方法

System.out.println("StaticExample static block");

}

```

## 静态类

　　\*\*只能在内部类中定义静态类\*\*，静态内部类与外层类绑定，即使没有创建外层类的对象，它一样存在。

　　静态类的方法可以是静态的方法也可以是非静态的方法，静态的方法可以在外层通过静态类调用，而非静态的方法必须要创建类的对象之后才能调用。

　　只能引用外部类的static成员变量（也就是类变量）。

　　如果一个内部类不是被定义成静态内部类，那么在定义成员变量或者成员方法的时候，是不能够被定义成静态的。

```

public class OuterClass {

public static class InnerClass{

InnerClass(){

System.out.println("静态内部类");

}

}

}

```

## 第26节 多态实现的JVM调用过程

> 多态

>

>同一个类的不同表现形态，不同的形态是通过其不同的子类体现

>java通过将子类对象引用赋值给超类对象变量, 来实现动态方法调用。

>

>[面向对象的三个特征与含义](http://blog.csdn.net/amazing7/article/details/51219687)

下面看例子：

```

public class A{

public String name = "父类name";

public void move(){

System.out.println("父类move");

}

}

```

```

public class B extends A{

public String name = "子类name";

@Override

public void move() {

// TODO Auto-generated method stub

System.out.println("子类move");

}

}

```

```

public class Test {

public static void main(String[] args) {

A a = new B();

a.move();

}

}

```

\*\*类B是类A的子类， A a = new B() 编译时变量和运行时变量不一样，所以多态发生了。\*\*

① A a 作为一个引用类型数据，存储在JVM栈的\*\*本地变量表\*\*中。

② new B()作为\*\*实例对象数据\*\*存储在堆中

　　 Ｂ的对象实例数据（接口、方法、field、对象类型等）的地址也存储在堆中

　　 Ｂ的对象的类型数据（对象实例数据的地址所执行的数据）存储在方法区中，方法区中 对象类型数据 中有一个指向该类方法的\*\*方法表\*\*。

③Java虚拟机规范中并未对引用类型访问具体对象的方式做规定，目前主流的实现方式主要有两种：

　１. 通过句柄访问

![这里写图片描述](http://img.blog.csdn.net/20160414163216948)

　　在这种方式中，JVM堆中会专门有一块区域用来作为句柄池，存储相关句柄所执行的实例数据地址（包括在堆中地址和在方法区中的地址）。这种实现方法由于用句柄表示地址，因此十分\*\*稳定\*\*。

２.通过直接指针访问

![这里写图片描述](http://img.blog.csdn.net/20160414163449343)

　　通过直接指针访问的方式中，reference中存储的就是对象在堆中的实际地址，在堆中存储的对象信息中包含了在方法区中的相应类型数据。这种方法最大的优势是速度快，在HotSpot虚拟机中用的就是这种方式。

④实现过程

首先虚拟机通过reference类型（A的引用）查询java栈中的 本地变量表，得到堆中的 对象类型数据的地址，从而找到方法区中的 对象类型数据（B的对象类型数据） ，然后查询方法表定位到实际类（B类）的方法运行。

# 第三章 数据结构

## 第1节 九大基础排序总结与对比

##

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

一、对比分析图

</font>

</td>

</tr>

</table>

<div align=center>

<img src="http://img.blog.csdn.net/20160607144411150" width="1131" height="590" />

</div>

- 均按从小到大排列

- k代表数值中的"数位"个数

- n代表数据规模

- m代表数据的最大值减最小值

> 稳定性：稳定排序算法会让原本有相等键值的纪录维持相对次序。也就是如果一个排序算法是稳定的，当有两个相等键值的纪录R和S，且在原本的列表中R出现在S之前，在排序过的列表中R也将会是在S之前。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

二、冒泡排序

</font>

</td>

</tr>

</table>

##

#概述

　　冒泡排序通过重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来，直到没有再需要交换的元素为止（对n个项目需要O(n^2)的比较次数）。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

#实现步骤

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2. 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

3. 针对所有的元素重复以上的步骤，除了最后一个。

4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

![这里写图片描述](http://img.blog.csdn.net/20160511102808244)

　　冒泡排序为一列数字进行排序的过程

#实现性能

- 最差时间复杂度

> O(n^2)

- 最优时间复杂度

> O(n)

- 平均时间复杂度

> O(n^2)

- 最差空间复杂度

> 总共O(n)，需要辅助空间O(1)

#Java实现

```

public static void main(String[] args) {

int[] number = {95,45,15,78,84,51,24,12};

bubble\_sort(number);

for(int i = 0; i < number.length; i++) {

System.out.print(number[i] + " ");

}

}

public static void bubble\_sort(int[] arr) {

int temp, len = arr.length;

for (int i = 0; i < len - 1; i++)

for (int j = 0; j < len - 1 - i; j++)

if (arr[j] > arr[j + 1]) {

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

三、选择排序

</font>

</td>

</tr>

</table>

##

#选择排序

　　常用的选择排序方法有简单选择排序和堆排序，这里只说简单选择排序，堆排序后面再说。

#简单选择排序

　　设所排序序列的记录个数为n，i　取　1,2,…,n-1　。

　　从所有n-i+1个记录（Ri,Ri+1,…,Rn）中找出排序码最小（或最大）的记录，与第i个记录交换。执行n-1趟 后就完成了记录序列的排序。

以排序数组｛3，2，1，4，6，5｝为例

![这里写图片描述](http://img.blog.csdn.net/20160511111047864)

![这里写图片描述](http://img.blog.csdn.net/20160511111210100)

#简单选择排序性能

　　在简单选择排序过程中，所需移动记录的次数比较少。最好情况下，即待排序记录初始状态就已经是正序排列了，则不需要移动记录。

　　最坏情况下，即待排序记录初始状态是按第一条记录最大，之后的记录从小到大顺序排列，则需要移动记录的次数最多为3（n-1）。

　　简单选择排序过程中需要进行的比较次数与初始状态下待排序的记录序列的排列情况\*\*无关\*\*。

　　当i=1时，需进行n-1次比较；当i=2时，需进行n-2次比较；依次类推，共需要进行的比较次数是(n-1)+(n-2)+…+2+1=n(n-1)/2，即进行\*\*比较操作的时间复杂度为O(n^2)\*\*，进行移动操作的时间复杂度为\*\*O(n)\*\*。

　　简单选择排序是不稳定排序。

#简单选择排序Java实现

```

public static void main(String[] args) {

int[] number = {3,1,2,8,4,5,24,12};

SimpleSort(number);

for(int i = 0; i < number.length; i++) {

System.out.print(number[i] + " ");

}

}

public static void SimpleSort(int[] arr) {

int length=arr.length;

int temp;

for(int i=0;i<length-1;i++){

int min=i;

for(int j=i+1;j<length;j++){ //寻找最小的数

if(arr[j]<arr[min]){

min =j;

}

}

if(min!=i){

temp = arr[min];

arr[min]=arr[i];

arr[i]=temp;

}

}

}

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

四、希尔排序

</font>

</td>

</tr>

</table>

##

#概述

　　希尔排序法(缩小增量法) 属于插入类排序，是将整个无序列分割成若干小的子序列分别进行插入排序的方法。

　　把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

- 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率。

- 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。

#实现过程

　　先取一个正整数d1小于n，把所有序号相隔d1的数组元素放一组，组内进行直接插入排序；然后取d2小于d1，重复上述分组和排序操作；直至di=1，即所有记录放进一个组中排序为止。

　　例如，假设有这样一组数[ 13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10 ]，如果我们以步长为5开始进行排序，我们可以通过将这列表放在有5列的表中来更好地描述算法，这样他们就应该看起来是这样：

> 13 14 94 33 82

25 59 94 65 23

45 27 73 25 39

10

然后我们对每列进行排序：

> 10 14 73 25 23

13 27 94 33 39

25 59 94 65 82

45

　　将上述四行数字，依序接在一起时我们得到：[ 10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45 ].这时10已经移至正确位置了，然后再以3为步长进行排序：

> 10 14 73

25 23 13

27 94 33

39 25 59

94 65 82

45

排序之后变为：

> 10 14 13

25 23 33

27 25 59

39 65 73

45 94 82

94

最后以1步长进行排序（此时就是简单的插入排序了）。

#实现效率

　　希尔排序是一个不稳定的排序，其时间复杂度受\*\*步长（增量）\*\*的影响。

　　空间复杂度：　Ｏ(1)

　　时间复杂度：　平均 O(n^1.3)

　　　　　　　　　最好　O(n)

　　　　　　　　　最坏　O(n^2)

#Java实现

```

public static void shellSort(int[] a) {

int gap = 1, i, j, len = a.length;

int temp;//插入排序交换值的暂存

//确定初始步长

while (gap < len / 3){

gap = gap \* 3 + 1;

}

for (; gap > 0; gap /= 3){//循环遍历步长，最后必为1

for (i = gap; i < len; i++) {//每一列依次向前做插入排序

temp = a[i];

//每一列中在a[i]上面且比a[i]大的元素依次向下移动

for (j = i - gap; j >= 0 && a[j] > temp; j -= gap){

a[j + gap] = a[j];

}

//a[i]填补空白，完成一列中的依次插入排序

a[j + gap] = temp;

}

}

}

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

五、归并排序

</font>

</td>

</tr>

</table>

##

#１.概述

　　归并排序，是创建在归并操作上的一种有效的排序算法该算法是采用分治法（Divide and Conquer）的一个非常典型的应用，且各层分治递归可以同时进行。

　　即先使每个子序列有序，再将两个已经排序的序列合并成一个序列的操作。若将两个有序表合并成一个有序表，称为二路归并。

例如：

> 设有数列{6，202，100，301，38，8，1}

初始状态：6,202,100,301,38,8，1

第一次归并后：{6,202},{100,301},{8,38},{1}，比较次数：3；

第二次归并后：{6,100,202,301}，{1,8,38}，比较次数：4；

第三次归并后：{1,6,8,38,100,202,301},比较次数：4；

总的比较次数为：3+4+4=11,；

\*\*逆序数\*\*为14；

![这里写图片描述](http://img.blog.csdn.net/20160517150920429)

　　　　　　归并排序示意图

#２.效率

　　归并排序速度仅次于快速排序，为稳定排序算法（即相等的元素的顺序不会改变），一般用于\*\*对总体无序，但是各子项相对有序的数列\*\*.

　　时间复杂度为\*\*O(nlogn)\*\*

　　空间复杂度为 \*\*O(n)\*\*

归并排序比较占用内存，但却是一种效率高且稳定的算法。

#3.迭代实现

##3.１实现原理

①申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列

②设定两个指针，最初位置分别为两个已经排序序列的起始位置

③比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置

④重复步骤③直到某一指针到达序列尾

⑤将另一序列剩下的所有元素直接复制到合并序列尾

##3.２Java代码

```

public static void main(String[] args) {

int [] arr ={6,5,3,1,8,7,2,4};

merge\_sort(arr);

for(int i : arr){

System.out.println(i);

}

}

public static void merge\_sort(int[] arr) {

int len = arr.length;

//用于合并的临时数组

int[] result = new int[len];

int block, start;

//两两合并后块大小变大两倍 (注意最后一次block等于len)

for(block = 1; block <=len ; block \*= 2) {

//把整个数组分成很多个块，每次合并处理两个块

for(start = 0; start <len; start += 2 \* block) {

int low = start;

int mid = (start + block) < len ? (start + block) : len;

int high = (start + 2 \* block) < len ? (start + 2 \* block) : len;

//两个块的起始下标及结束下标

int start1 = low, end1 = mid;

int start2 = mid, end2 = high;

//开始对两个block进行归并排序

while (start1 < end1 && start2 < end2) {

result[low++] = arr[start1] < arr[start2] ? arr[start1++] : arr[start2++];

}

while(start1 < end1) {

result[low++] = arr[start1++];

}

while(start2 < end2) {

result[low++] = arr[start2++];

}

}

//每次归并后把结果result存入arr中，以便进行下次归并

int[] temp = arr;

arr = result;

result = temp;

}

}

```

#4.递归实现

##4.１实现原理

假设序列共有n个元素

①将序列每相邻两个数字进行归并操作，形成floor(n/2)个序列，排序后每个序列包含两个元素。

②将上述序列再次归并，形成floor(n/4)个序列，每个序列包含四个元素

③重复步骤②，直到所有元素排序完毕

##4.２Java代码

```

public static void main(String[] args) {

int [] arr ={6,5,3,1,8,7,2,4};

int len = arr.length;

int[] reg = new int[len];

merge\_sort\_recursive(arr,reg,0,len-1);

for(int i : arr){

System.out.println(i);

}

}

static void merge\_sort\_recursive(int[] arr, int[] reg, int start, int end) {

if (start >= end)

return;

int len = end - start, mid = (len >> 1) + start;

int start1 = start, end1 = mid;

int start2 = mid + 1, end2 = end;

//递归到子序列只有一个数的时候，开始逐个返回

merge\_sort\_recursive(arr, reg, start1, end1);

merge\_sort\_recursive(arr, reg, start2, end2);

//-------合并操作，必须在递归之后（子序列有序的基础上）----

int k = start;

while (start1 <= end1 && start2 <= end2)

reg[k++] = arr[start1] < arr[start2] ? arr[start1++] : arr[start2++];

while (start1 <= end1)

reg[k++] = arr[start1++];

while (start2 <= end2)

reg[k++] = arr[start2++];

//借用reg数组做合并，然后把数据存回arr中

for (k = start; k <= end; k++)

arr[k] = reg[k];

}

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

六、快速排序

</font>

</td>

</tr>

</table>

##

#基本思想

　　快速排序（Quicksort）是对冒泡排序的一种改进，又称划分交换排序（partition-exchange sort。

　　快速排序使用分治法（Divide and conquer）策略来把一个序列（list）分为两个子序列（sub-lists）。

步骤为：

①．从数列中挑出一个元素，称为"基准"（pivot）

②．重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。

③．递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序

![这里写图片描述](http://img.blog.csdn.net/20160519164311224)

使用快速排序法对一列数字进行排序的过程

#排序效率

　　在平均状况下，排序n个项目要Ο(n log n)次比较。在最坏状况下则需要Ο(n2)次比较，但这种状况并不常见。事实上，快速排序通常明显比其他Ο(n log n)算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。

最差时间复杂度 Ο(n^2)

最优时间复杂度 Ο(n log n)

\*\*平均时间复杂度Ο(n log n)\*\*

最差空间复杂度 根据实现的方式不同而不同

#Java实现

```

public static void main(String[] args) {

int [] arr = {8,1,0,4,6,2,7,9,5,3};

quickSort(arr,0,arr.length-1);

for(int i :arr){

System.out.println(i);

}

}

public static void quickSort(int[]arr,int low,int high){

if (low < high) {

int middle = getMiddle(arr, low, high);

quickSort(arr, low, middle - 1);

quickSort(arr, middle + 1, high);

}

}

public static int getMiddle(int[] list, int low, int high) {

int tmp = list[low];

while (low < high) {

while (low < high && list[high] >= tmp) {

high--;

}

list[low] = list[high];

while (low < high && list[low] <= tmp) {

low++;

}

list[high] = list[low];

}

list[low] = tmp;

return low;

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160519181157295)

分析：

![这里写图片描述](http://img.blog.csdn.net/20160519180035248)

> 取８为中值，红色箭头表示low，绿色箭头表示high

①从high开始向前扫描到第一个比８小的值与８交换。

②从low向后扫描第一比8大的值与8交换。

③重复①②过程只到，high=low完成一次快速排序，然后递归子序列。

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

七、堆排序

</font>

</td>

</tr>

</table>

##

#浅析堆

　　堆排序(Heapsort)是指利用堆这种数据结构所设计的一种排序算法，它是选择排序的一种。可以利用\*\*数组\*\*的特点快速定位指定索引的元素。堆分为大根堆和小根堆，是完全二叉树。大根堆的要求是每个节点的值都不大于其父节点的值。

　　由于堆中每次都只能删除第0个数据，通过　取出第０个数据再执行堆的删除操作、重建堆（实际的操作是将最后一个数据的值赋给根结点，然后再从根结点开始进行一次从上向下的调整。），然后再取，如此重复实现排序。

\*\*堆的操作：\*\*

![这里写图片描述](http://img.blog.csdn.net/20160523164108748)

在堆的数据结构中，堆中的最大值总是位于根节点。堆中定义以下几种操作：

- 最大堆调整（Max\_Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点

- 创建最大堆（Build\_Max\_Heap）：将堆所有数据重新排序

- 堆排序（HeapSort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算

\*\*堆的存储：\*\*

![这里写图片描述](http://hi.csdn.net/attachment/201108/22/0\_1314014706gZqn.gif)

通常堆是通过一维数组来实现的。在数组起始位置为0的情形中：

- 父节点i的左子节点在位置(2\*i+1);

- 父节点i的右子节点在位置(2\*i+2);

- 子节点i的父节点在位置floor((i-1)/2);

#Java代码实现

```

public class HeapSort {

private static int[] sort = new int[]{1,0,10,20,3,5,6,4,9,8,12,17,34,11};

public static void main(String[] args) {

buildMaxHeapify(sort);

heapSort(sort);

print(sort);

}

private static void buildMaxHeapify(int[] data){

//没有子节点的才需要创建最大堆，从最后一个的父节点开始

int startIndex = getParentIndex(data.length - 1);

//从尾端开始创建最大堆，每次都是正确的堆

for (int i = startIndex; i >= 0; i--) {

maxHeapify(data, data.length, i);

}

}

/\*\*

\* 创建最大堆

\* @param data

\* @param heapSize需要创建最大堆的大小，一般在sort的时候用到，因为最多值放在末尾，末尾就不再归入最大堆了

\* @param index当前需要创建最大堆的位置

\*/

private static void maxHeapify(int[] data, int heapSize, int index){

// 当前点与左右子节点比较

int left = getChildLeftIndex(index);

int right = getChildRightIndex(index);

int largest = index;

if (left < heapSize && data[index] < data[left]) {

largest = left;

}

if (right < heapSize && data[largest] < data[right]) {

largest = right;

}

//得到最大值后可能需要交换，如果交换了，其子节点可能就不是最大堆了，需要重新调整

if (largest != index) {

int temp = data[index];

data[index] = data[largest];

data[largest] = temp;

maxHeapify(data, heapSize, largest);

}

}

/\*\*

\* 排序，最大值放在末尾，data虽然是最大堆，在排序后就成了递增的

\* @param data

\*/

private static void heapSort(int[] data) {

//末尾与头交换，交换后调整最大堆

for (int i = data.length - 1; i > 0; i--) {

int temp = data[0];

data[0] = data[i];

data[i] = temp;

maxHeapify(data, i, 0);

}

}

/\*\*

\* 父节点位置

\* @param current

\* @return

\*/

private static int getParentIndex(int current){

return (current - 1) >> 1;

}

/\*\*

\* 左子节点position注意括号，加法优先级更高

\* @param current

\* @return

\*/

private static int getChildLeftIndex(int current){

return (current << 1) + 1;

}

/\*\*

\* 右子节点position

\* @param current

\* @return

\*/

private static int getChildRightIndex(int current){

return (current << 1) + 2;

}

private static void print(int[] data){

for (int i = 0; i < data.length; i++) {

System.out.print(data[i] + " |");

}

}

}

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

八、桶排序

</font>

</td>

</tr>

</table>

##

#1.概念

桶排序（Bucket sort）或所谓的箱排序，是一个排序算法。

　　假设有一组长度为N的待排关键字序列K[1....n]。首先将这个序列划分成M个的子区间(桶) 。然后基于某种映射函数 ，将待排序列的关键字k映射到第i个桶中(即桶数组B的下标 i) ，那么该关键字k就作为B[i]中的元素。接着对每个桶B[i]中的所有元素进行比较排序(可以使用快排)。然后依次枚举输出B[0]....B[M]中的全部内容即是一个有序序列。

桶排序的步骤：

①设置一个定量的数组当作空桶子。

②寻访序列，并且把项目一个一个放到对应的桶子去。

③对每个不是空的桶子进行排序。

④从不是空的桶子里把项目再放回原来的序列中。

#２.性能

数据结构 \*\*数组\*\*

最差时间复杂度 　 O(n^2)

平均时间复杂度 　O(n+k)

最差空间复杂度　O(n\*k)

　　平均情况下桶排序以线性时间运行，桶排序是\*\*稳定\*\*的，排序非常快,但是同时也非常耗空间,基本上是最\*\*耗空间\*\*的一种排序算法。

　　对N个关键字进行桶排序的时间复杂度分为两个部分：

①循环计算每个关键字的桶映射函数，这个时间复杂度是O(N)。

②利用先进的比较排序算法对每个桶内的所有数据进行排序，其时间复杂度为 ∑ O(Ni\*logNi) 。其中Ni 为第i个桶的数据量。

　　很显然，第②部分是桶排序性能好坏的决定因素。尽量减少桶内数据的数量是提高效率的唯一办法(因为基于比较排序的最好平均时间复杂度只能达到O(N\*logN)了)。因此，我们需要尽量做到下面两点：

> ① 映射函数f(k)能够将N个数据平均的分配到M个桶中，这样每个桶就有[N/M]个数据量。

②尽量的增大桶的数量。极限情况下每个桶只能得到一个数据，这样就完全避开了桶内数据的“比较”排序操作。 当然，做到这一点很不容易，数据量巨大的情况下，f(k)函数会使得桶集合的数量巨大，空间浪费严重。这就是一个时间代价和空间代价的权衡问题了。

#3.java实现

![这里写图片描述](http://img.blog.csdn.net/20160607102041634)

　　对０～１之间的一组浮点数进行升序排序：

BucketSort.java

```

public class BucketSort {

/\*\*

\* 对arr进行桶排序，排序结果仍放在arr中

\*/

public static void bucketSort(double arr[]){

//-------------------------------------------------分桶-----------------------------------------------

int n = arr.length;

//存放桶的链表

ArrayList bucketList[] = new ArrayList [n];

//每个桶是一个list，存放此桶的元素

for(int i =0;i<n;i++){

//下取等

int temp = (int) Math.floor(n\*arr[i]);

//若不存在该桶，就新建一个桶并加入到桶链表中

if(null==bucketList[temp])

bucketList[temp] = new ArrayList();

//把当前元素加入到对应桶中

bucketList[temp].add(arr[i]);

}

//-------------------------------------------------桶内排序-----------------------------------------------

//对每个桶中的数进行插入排序

for(int i = 0;i<n;i++){

if(null!=bucketList[i])

insert(bucketList[i]);

}

//-------------------------------------------------合并桶内数据-----------------------------------------------

//把各个桶的排序结果合并

int count = 0;

for(int i = 0;i<n;i++){

if(null!=bucketList[i]){

Iterator iter = bucketList[i].iterator();

while(iter.hasNext()){

Double d = (Double)iter.next();

arr[count] = d;

count++;

}

}

}

}

/\*\*

\* 用插入排序对每个桶进行排序

\* 从小到大排序

\*/

public static void insert(ArrayList list){

if(list.size()>1){

for(int i =1;i<list.size();i++){

if((Double)list.get(i)<(Double)list.get(i-1)){

double temp = (Double) list.get(i);

int j = i-1;

for(;j>=0&&((Double)list.get(j)>(Double)list.get(j+1));j--)

list.set(j+1, list.get(j)); //后移

list.set(j+1, temp);

}

}

}

}

}

```

测试代码：

```

public static void main(String[] args) {

double arr [] ={0.21,0.23,0.76,0.12,0.89};

BucketSort.bucketSort(arr);

for(double a:arr){

System.out.println(a);

}

}

```

输出结果：

![这里写图片描述](http://img.blog.csdn.net/20160607105154789)

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

九、基数排序

</font>

</td>

</tr>

</table>

##

#原理

　　基数排序（Radix sort）是一种非比较型整数排序算法，其原理是将\*\*整数按位数切割\*\*成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。

　　将所有待比较数值（正整数）统一为\*\*同样的数位长度\*\*，数位较短的数前面\*\*补零\*\*。然后，从\*\*最低位\*\*开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

#效率

　　基数排序的时间复杂度是\*\*O(k·n)\*\*，其中n是排序元素个数，k是数字位数。注意这不是说这个时间复杂度一定优于O(n·log(n))，k的大小取决于数字位的选择和待排序数据所属数据类型的全集的大小；k决定了进行多少轮处理，而n是每轮处理的操作数目。

　　基数排序基本操作的代价较小，k一般不大于logn，所以基数排序一般要\*\*快过基于比较的排序\*\*，比如快速排序。

　　最差空间复杂度是\*\*O(k·n)\*\*

#Java实现

　　![这里写图片描述](http://img.blog.csdn.net/20160607115546499)

　　现在有数组：278，109，63,930,589,184,505,269,8,83　。根据各位数将数组划分为10个链表(当然其中的某些链表可能不含有元素)

\*\*第一次分配\*\*：

> 0:930

1:

2:

3:63,83

4:184

5:505

6：

7:

8:278,8

9:109,589,269

\*\*第一次收集后的数组\*\*：

> 930,63,83,184,505，278,8,109,589,269

\*\*第二次分配\*\*：

> 0:505,8,109

1:

2:

3:930

4:

5:

6：63,269

7:278

8:83,184，589

9:

\*\*第二次收集后的数组\*\*：

>505,8,109，930，63,269，278，83,184，589

\*\*第三次分配：\*\*

> 0:8,63,83

1:109,184

2:278,269

3:

4:

5:505,589

6:

7:

8:

9:930

最后得到序列：

> 8,63,83,109,184，269,278,505,589,930

基数排序其实是利用多关键字先达到局部有序，再调整达到全局有序。

代码实现：

```

public class Test {

public static void main(String[] args) {

int[] array = {278,109,63,930,589,184,505,269,8,83};

radixSort(array);

for(double a : array){

System.out.println(a);

}

}

public static void radixSort(int[] array){

//------------------------------------------确定排序的趟数----------------------------------

int max=array[0];

for(int i=1;i<array.length;i++){

if(array[i]>max){

max=array[i];

}

}

int time=0;

while(max>0){

max/=10;

time++;

}

//----------------------------------------初始化10个链表用户分配时暂存-------------------------------

List<List<Integer>> list=new ArrayList<List<Integer>>();

for(int i=0;i<10;i++){

List<Integer> item=new ArrayList<Integer>();

list.add(item);

}

//-----------------------------------------进行time次分配和收集-------------------------------------

for(int i=0;i<time;i++){

//分配元素;

for(int j=0;j<array.length;j++){

int index = array[j]%(int)Math.pow(10, i+1)/(int)Math.pow(10, i);

list.get(index).add(array[j]);

}

//收集元素;

int count=0;

for(int k=0;k<10;k++){

if(list.get(k).size()>0){

for(int a : list.get(k)){

array[count]=a;

count++;

}

//清除数据，以便下次收集

list.get(k).clear();

}

}

}

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160607135057719)

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

十、插入排序

</font>

</td>

</tr>

</table>

##

#概述

　　将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于\*\*少量数据\*\*的排序，是\*\*稳定\*\*的排序方法。

　　插入排序又分为　直接插入排序　和 折半插入排序。

#直接插入排序

　　把待排序的纪录按其关键码值的大小逐个插入到一个已经排好序的有序序列中，直到所有的纪录插入完为止，得到一个新的有序序列。

##Java实现

```

public static void insertSort(int a[]){

int j; //当前要插入值的位置

int preJ; //依次指向j前的位置

int key; //后移时来暂存要插入的值

//从数组的第二个位置开始遍历值

for(j=1;j<a.length;j++){

key=a[j];

preJ=j-1;

//a[preJ]比当前值大时，a[preJ]后移一位

while(preJ>=0 && a[preJ]>key){

a[preJ+1]=a[preJ]; //将a[preJ]值后移

//这里注意: a[preJ+1]=a[j]=key,把插入值已经存在了 key中

//等于说, 留出来一个空白位置来实现依次后移（不会造成数据丢失问题）

preJ--; //preJ前移

}

//找到要插入的位置或已遍历完成（(preJ=0）

a[preJ+1]=key; //将当前值插入 空白位置

}

}

```

　　备注很清楚，我就不多说了．．．．

##效率分析

　　空间复杂度O(1)

　　平均时间复杂度O(n^2)

>最差情况：反序，需要移动n\*(n-1)/2个元素 ，运行时间为O(n^2)。

最好情况：正序，不需要移动元素，运行时间为O(n)．

#折半插入排序

　　直接插入排序中要把\*\*插入元素\*\*与\*\*已有序序列元素\*\*依次进行比较，效率非常低。

　　折半插入排序,使用使用\*\*折半查找\*\*的方式寻找插入点的位置, 可以减少比较的次数,但移动的次数不变, 时间复杂度和空间复杂度和直接插入排序一样，在元素较多的情况下能提高查找性能。

##Java实现

```

private static void binaryInsertSort(int[] a)

{

//从数组的第二个位置开始遍历值

for(int i = 1; i < a.length; i++) {

int key = a[i]; //暂存要插入的值

int pre = 0; //有序序列开始和结尾下标申明

int last = i - 1;

// 折半查找出插入位置 a[pre]

while(pre <= last) {

int mid = (pre + last) / 2;

if(key < a[mid]) {

last = mid - 1;

} else {

pre = mid + 1;

}

}

//a[i]已经取出来存放在key中，把下标从pre + 1到 i-1的元素依次后移

for(int j = i; j >= pre + 1; j--) {

a[j] = a[j - 1];

}

//把值插入空白位置

a[pre] = key;

}

}

```

直接插入排序是，比较一个后移一个；

折半插入排序是，先找到位置，然后一起移动；

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

十一、补充

</font>

</td>

</tr>

</table>

###1. 快排的partition函数

　　作用：给定一个数组arr[]和数组中任意一个元素a，重排数组使得a左边都小于它，右边都不小于它。

```

// A[]为数组，start、end分别为数组第一个元素和最后一个元素的索引

// povitIndex为数组中任意选中的数的索引

static int partition(int A[], int start, int end, int pivotIndex){

int i = start, j = end, pivot = A[pivotIndex];

swap<int>(A[end], A[pivotIndex]);

while(i < j){

while(i < j && A[i] <= pivot) ++i;

while(i < j && A[j] >= pivot) --j;

if(i < j) swap<int>(A[i], A[j]);

}

swap<int>(A[end], A[i]);

return i;

}

```

###2. 冒泡排序的改进

思路：

①、加一个标志位，当某一趟冒泡排序没有元素交换时，则冒泡结束，元素已经有序，可以有效的减少冒泡次数。

```

/\*\*

\* 引入标志位，默认为true

\* 如果前后数据进行了交换，则为true，否则为false。如果没有数据交换，则排序完成。

\*/

public static int[] bubbleSort(int[] arr){

boolean flag = true;

int n = arr.length;

while(flag){

flag = false;

for(int j=0;j<n-1;j++){

if(arr[j] >arr[j+1]){

//数据交换

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

//设置标志位

flag = true;

}

}

n--;

}

return arr;

}

```

②、记录每一次元素交换的位置，当元素交换的位置在第0个元素时，则排序结束。

###3.快排优化

① 快速排序在处理小规模数据时的表现不好，这个时候可以改用插入排序。

②对于一个每个元素都完全相同的一个序列来讲，快速排序也会退化到 O(n^2)。要将这种情况避免到，可以这样做：

　　在分区的时候，将序列分为 3 堆，一堆小于中轴元素，一堆等于中轴元素，一堆大于中轴元素，下次递归调用快速排序的时候，只需对小于和大于中轴元素的两堆数据进行排序，中间等于中轴元素的一堆已经放好。

## 第2节 AVL树和AVL旋转、哈夫曼树和哈夫曼编码

#1. AVL树

　　AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是O（log n）。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

　　节点的平衡因子是它的左子树的高度减去它的右子树的高度（有时相反）。带有平衡因子1、0或 -1的节点被认为是平衡的。带有平衡因子 -2或2的节点被认为是不平衡的，并需要\*\*重新平衡\*\*这个树。平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。

##１.２AVL旋转

　　AVL树的基本操作一般涉及运作同在不平衡的二叉查找树所运作的同样的算法。但是要进行预先或随后做一次或多次所谓的"AVL旋转"。

　　以下图表以 \*\*四列\*\* 表示四种情况，每行表示在该种情况下要进行的操作。在左左和右右的情况下，只需要进行一次旋转操作；在左右和右左的情况下，需要进行两次旋转操作。

> Root是失去平衡树的根节点，Pivot是旋转后重新平衡树的根节点

![这里写图片描述](http://img.blog.csdn.net/20160612143136956)

- 插入

　向AVL树插入，可以透过如同它是未平衡的二叉查找树一样，把给定的值插入树中，接着自底往上向根节点折回，于在插入期间成为不平衡的所有节点上进行旋转来完成。

- 删除

　从AVL树中删除，可以透过把要删除的节点向下旋转成一个叶子节点，接着直接移除这个叶子节点来完成。

- 搜寻

　可以像普通二叉查找树一样的进行，所以耗费O(log n)时间，因为AVL树总是保持平衡的。

##１.３AVL节点数计算

高度为h的AVL树，节点数N最多 ![这里写图片描述](https://wikimedia.org/api/rest\_v1/media/math/render/svg/f933ccc9d17f70cd03d6aa6fbf61b34a0d47ff62)。

![这里写图片描述](http://img.blog.csdn.net/20160612143832921)

想起了什么？　Fibonacci 是不是．．．可以根据节点数ｎ求出相应的高度ｈ。

#2.哈夫曼树

　　哈夫曼树又称最优二叉树，是一种\*\*带权路径长度最短的二叉树\*\*。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的 路径长度（若根结点为0层，叶结点到根结点的路径长度为叶结点的层数）。

　　树的带权路径长度记为WPL= (W1\*L1+W2\*L2+W3\*L3+...+Wn\*Ln)，N个权值Wi(i=1,2,...n)构成一棵有N个叶结点的二叉树，相应的叶结点的路径长度为Li(i=1,2,...n)。可以证明哈夫曼树的WPL是最小的。

#2.1哈夫曼树的构造

　　假设有n个权值，则构造出的哈夫曼树有n个叶子结点。 n个权值分别设为 w1、w2、…、wn，则哈夫曼树的构造规则为：

①　将w1、w2、…，wn看成是有n 棵树的森林(每棵树仅有一个结点)；

②在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

③从森林中删除选取的两棵树，并将新树加入森林；

④重复②、③步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

#2.２哈夫曼编码

　　哈夫曼编码是可变字长编码(VLC)的一种。该方法完全依据字符出现概率来构造异字头的平均长度最短的码字，有时称之为最佳编码。

![这里写图片描述](http://img.blog.csdn.net/20160612151017097)

> 有A,B,C,D,E五个字符，出现的频率（即权值）分别为5,4,3,2,1，对其进行哈夫曼编码。

首先把A,B,C,D,E构造成一棵哈夫曼树（２.１有说明）；

通过从哈夫曼树根结点开始，对左子树分配代码“0”，右子树分配代码“1”，一直到达叶子结点为止，然后将从树根沿每条路径到达叶子结点的代码排列起来，便得到了哈夫曼编码。

![这里写图片描述](http://img.blog.csdn.net/20160612151250872)

所以各字符对应的编码为：A->11,B->10,C->00,D->011,E->010

霍夫曼编码是一种无前缀编码。解码时不会混淆。其主要应用在数据压缩，加密解密等场合。

## 第3节 B(B-)树、B+树、B树

#1.　B树

　　Ｂ树及Ｂ－树，是一种自平衡的树，能够\*\*保持数据有序\*\*。这种数据结构能够让查找数据、顺序访问、插入数据及删除的动作，都在对数时间内完成。

　　B树，概括来说是一个一般化的\*\*二叉查找树\*\*（binary search tree），可以拥有\*\*多于2个子节点\*\*。与自平衡二叉查找树不同，B树为系统大块数据的读写操作做了优化。B树减少定位记录时所经历的中间过程，从而加快存取速度。B树这种数据结构可以用来描述外部存储。这种数据结构常被应用在数据库和文件系统的实作上。

##1.1　B树的性质

M为树的阶数，B-树或为空树，否则满足下列条件：

1.　定义任意非叶子结点最多只有M个儿子；且M>2；

2.　根结点的儿子数为[2, M]；

3.　除根结点以外的非叶子结点的儿子数为[M/2, M]；

4.　每个结点存放至少M/2-1（取上整）和至多M-1个关键字；（至少2个关键字）

5.　非叶子结点的关键字个数=指向儿子的指针个数-1；

6.　非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；

7.　非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；

8.　所有叶子结点位于同一层；

如：（M=3）

![这里写图片描述](http://img.blog.csdn.net/20160612115550177)

　　B-树的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点。

##1.２　B树的运用场景

- 保持\*\*键值有序\*\*，以顺序遍历

- 使用\*\*层次化的索引来最小化磁盘读取\*\*

- 使用\*\*不完全填充\*\*的块来加速插入和删除

- 通过优雅的遍历算法来保持\*\*索引平衡\*\*

另外，B树通过保证内部节点至少半满来\*\*最小化空间浪费\*\*。一棵B树可以处理任意数目的插入和删除。

#２.　B＋树

　　B+ 树是一种树数据结构，是一个\*\*n叉树\*\*，每个节点通常有多个孩子，一颗B+树包含\*\*根节点、内部节点和叶子节点\*\*。根节点可能是一个叶子节点，也可能是一个包含两个或两个以上孩子节点的节点。

　　B+ 树通常用于\*\*数据库和操作系统的文件系统\*\*中。NTFS, ReiserFS, NSS, XFS, JFS, ReFS 和BFS等文件系统都在使用B+树作为\*\*元数据索引\*\*。B+ 树的特点是能够保持\*\*数据稳定有序\*\*，其插入与修改拥有较稳定的对数时间复杂度。B+ 树元素自底向上插入。

##２.1　B＋树的性质

B+树是B-树的变体，也是一种多路搜索树，其定义基本与B-树同，除了：

１.　非叶子结点的子树指针与关键字个数相同；

２.　非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

３.　为所有叶子结点增加一个链指针；

４.　所有关键字都在叶子结点出现；

![这里写图片描述](http://img.blog.csdn.net/20160612134221890)

##２.２　B＋树与Ｂ树的区别

1.　所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

2.　不可能在非叶子结点命中；

3.　非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；

4.　\*\*更适合文件索引系统\*\*；

　　 B+的搜索与B-树也基本相同，区别是B+树只有达到叶子结点才命中（B-树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；

#３.　B\*树

B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针；

![这里写图片描述](http://img.blog.csdn.net/20160612134707432)

　　 B\*树定义了非叶子结点关键字个数至少为(2/3)\*M，即块的最低使用率为2/3（代替B+树的1/2）；

B+树的分裂：

> 当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

B\*树的分裂：

>当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针；

所以，B\*树分配新结点的概率比B+树要低，\*\*空间使用率更高\*\*；

#4 总结

- B-树：

　　多路搜索树，每个结点存储M/2到M个关键字，非叶子结点存储指向关键字范围的子结点；所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；

- B+树：

　　在B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点

中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；

- B\*树：

　　在B+树基础上，为非叶子结点也增加链表指针，将结点的最低利用率从1/2提高到2/3；

## 第4节 Hash表、Hash函数及冲突解决

#１.Hash表

　　哈希表（Hash table，也叫散列表），是根据key而\*\*直接\*\*进行访问的数据结构。也就是说，它通过把key映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

　　以数据中每个元素的关键字K为自变量，通过散列函数H（k）计算出函数值，以该函数值作为一块连续存储空间的的单元地址，将该元素存储到函数值对应的单元中。

　　哈希表存储的是键值对，其查找的时间复杂度与元素数量多少无关，哈希表在查找元素时是通过计算哈希码值来定位元素的位置从而直接访问元素的，因此，哈希表查找的时间复杂度为O（1）。

#2.哈希表的构造方法

##2.1直接定址法

　　取关键字或者关键字的某个\*\*线性函数\*\*值作为哈希地址,即

　　H(Key)=Key或者H(Key)=a\*Key+b(a,b为整数)

　　这种散列函数也叫做自身函数.如果H(Key)的哈希地址上已经有值了,那么就往下一个位置找,直到找到H(Key)的位置没有值了就把元素放进去.

　　此法仅适合于：地址集合的大小 等于 关键字集合的大小

##2.２ 数字分析法

　　分析一组数据,比如一组员工的出生年月,这时我们发现出生年月的前几位数字一般都相同,因此,出现冲突的概率就会很大,但是我们发现年月日的后几位表示月份和具体日期的数字差别很大,如果利用后面的几位数字来构造散列地址,则冲突的几率则会明显降低.

　　因此数字分析法就是找出\*\*数字的规律\*\*,尽可能利用这些数据来构造冲突几率较低的散列地址.

　　此法适于:能预先估计出全体关键字的每一位上各种数字出现的频度。

##2.３ 平方取中法

　　以关键字的平方值的中间几位作为存储地址（哈希地址）。求“关键字的平方值” 的目的是“扩大差别” ，同时平方值的中间各位又能受到整个关键字中各位的影响。

　　此法适于:关键字中的每一位都有某些数字重复出现频度很高的现象。

##2.４ 折叠法

　　 将关键字分割成若干部分，然后取它们的叠加和为哈希地址。两种叠加处理的方法：移位叠加:将分 割后的几部分低位对齐相加；间界叠加:从一端沿分割界来回折叠，然后对齐相加。

　　此法适于：关键字的\*\*数字位数特别多\*\*。

##2.５随机数法

　　设定哈希函数为:H(key) = Random(key)其中，Random 为伪随机函数

此法适于：对\*\*长度不等的关键字\*\*构造哈希函数。

##2.６除留余数法

　　取关键字被某个不大于散列表表长m的数p除后所得的余数为散列地址.即

　　哈希函数为:H(key) = key MOD p ( p≤m )，其中， m为表长，p 为不大于 m 的素数。

#３.哈希表冲突解决方法

　　哈希表处理冲突主要有\*\*开放寻址法\*\*、\*\*再散列法\*\*、\*\*链地址法\*\*（拉链法）和建立一个\*\*公共溢出区\*\*四种方法。

　　通过构造性能良好的哈希函数，可以减少冲突，但一般不可能完全避免冲突，因此解决冲突是哈希法的另一个关键问题。

　“处理冲突” 的实际含义是：\*\*为产生冲突的关键字寻找下一个哈希地址。\*\*

##３.１开放定址法

　　一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。

###３.１.1线性探测

　　冲突发生时，\*\*顺序查看\*\*表中下一单元，直到找出一个空单元或查遍全表。

　　公式：

```

fi(key) = (f(key)+di) MOD m (di=1,2,3,......,m-1)

```

###３.１.２二次探测法

　　冲突发生时，在表的左右进行跳跃式探测，双向寻找到可能的空位置。

公式：

```

fi(key) = (f(key)+di) MOD m (di = 12, -12, 22, -22,……, q2, -q2, q <= m/2)

```

###３.１.３随机探测法

　　在冲突时，对于位移量 di 采用随机函数计算得到，我们称之为随机探测法。

　公式：

```

fi(key) = (f(key)+di) MOD m (di是一个随机数列)

```

　　线性探测再散列容易产生“二次聚集”，即在处理同义词的冲突时又导致非同义词的冲突。

　　线性探测再散列的优点是：只要哈希表不满，就一定能找到一个不冲突的哈希地址，而二次探测再散列和伪随机探测再散列则不一定。

##３.２链地址法

　　 将所有哈希地址相同的记录都链接在同一链表中。各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况。

　　 处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；

##３.３再哈希法

这种方法是同时构造多个不同的哈希函数：

Hi=RH1（key），i=1，2,3，…,n.

当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

##３.４建立公共溢出区

　　这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表.(注意：在这个方法里面是把元素分开两个表来存储)

## 第5节 KMP的一个简单解释

#1.概述

　　KMP算法的关键是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。

　　给定两个字符串T和W，长度分别为m和n，判断W是否在T中出现，如果出现则返回出现的位置。常规方法是遍历T的每一个位置，然后从该位置开始和W进行匹配，但是这种方法的复杂度是O(mn)。kmp算法通过一个O(n)的预处理，使匹配的复杂度降为O(m+n)。

#2.思路

![这里写图片描述](http://img.blog.csdn.net/20160617180047231)

> 我们要在

> 字符串\*\*Ｔ＝“ＡＢＣＤＡＢＣＤＡＢＤＥ”\*\*中查找

> 字符串\*\*Ｗ＝＂ＡＢＣＤＡＢＤ＂\*\*出现的位置。

![这里写图片描述](http://img.blog.csdn.net/20160617180126140)

　　很明显可以看出前６位都是匹配的，然而第７位不匹配，一般的做法就是从Ｔ[i]（i从1到9）开始和W开始循环比较。直到i=4，也就是图2，比较成功返回下标4。

![这里写图片描述](http://img.blog.csdn.net/20160614111238202)

　　这样比较无异效率很低，然而我们发现在字符串"ABCDAB"之中有两个"AB"，第一次比较失败之后，又循环比较了四次，把W字符串从第一个"AB"的位置移动到了第二个"AB"处（显然这四次比较毫无意义）。于是Knuth、Morris、Pratt这三个人就想了个办法去掉了这些无意义的比较。

　　然而是怎么实现的呢？就要从Ｗ字符串的部分匹配值（相等的前缀和后缀）说起。

##2.１部分匹配值

```

"前缀"指除了最后一个字符以外，一个字符串的全部头部组合；

"后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

```

　　"部分匹配值"就是"前缀"和"后缀"的最长的共有元素的\*\*长度\*\*。以Ｗ字符串"ABCDABD"为例：

> "A"的前缀和后缀都为空集，共有元素的长度为0；

>

"AB"的前缀为[A]，后缀为[B]，共有元素的长度为0；

>"ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度0；

>

"ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；

>"ABCDA"的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为"A"，长度为1；

>

"ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为"AB"，长度为2；

>"ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为0。

从上面我们就得出了字符串Ｗ的部分匹配表：

![这里写图片描述](http://img.blog.csdn.net/20160614113209853)

我们把部分匹配表存放在一个next[]数组中，那部分匹配值究竟怎么用呢？

我们可以根据\*\*部分匹配值计算出向后移动的位数\*\*，避免了一位位的比较：

> \*\*移动位数 = 已匹配的字符数 - 对应的部分匹配值\*\*

以图１中为例：

　前６位已经匹配\*\*（已匹配的字符数 ＝６）\*\*；

　next[5] = 2 \*\*(对应的部分匹配值 = 2)\*\*;

我们可以得出移动位数为 6-2=4位。

>　　 可以看出next数组的作用，就是在我们匹配失败的时候，确定我们子串需要往后移动的距离，而避免我们的主串指针进行回退。这样可以保证主串在只遍历一遍的情况下找到子串。因此KMP算法的重点就是如何快速的求出这个next数组。

##2.2 求next数组

　　next数组是只与子串有关与主串无关的，它记录的是子串到每个字符处那个公共前缀（或后缀）的最大长度。

![这里写图片描述](http://img.blog.csdn.net/20160614150241191)

　　 假设我们已经求出的next[i-1] = ｊ，即Ｗ从０到i-1处这段字符串中，最大的相等的前缀和后缀长度为ｊ。

　　 那如何求得next[i]呢？

这是后就要分为两种情况了：

①、如果Ｗ[i]==W[j]

　　显然前缀往后再加上一个字符之后依然会和后缀往后加上一个字符相等，此时

> \*\*next[i] = next[i-1] + 1，即next[i] = ｊ+1\*\*

②、如果Ｗ[i]！=W[j]

　　那么 Ｗ[0-i]这段字符串 中，\*\*最大的相等\*\*的前缀和后缀的\*\*长度\*\*必然\*\*小于等于\*\*ｊ。 （以前是j，现在加了一个不同字符，所以不可能大于j）。

　　从上面的Ｗ字符串的前缀分析可以看出，\*\*短字符串的前缀必是长字符串前缀的子集\*\*。

　　既然长的匹配值不行了，我们只能回溯到短的匹配。把ｊ下标循环向前（下标０方向）移动，直到\*\*W[j]==W[i]或者j==0\*\*为止（回到第一种情况）。

　　那么ｊ怎么移动呢？当前Ｗ[i]！=W[j]，那么就求0到j-1的最长匹配值（next[j - 1]），再比较这个最长匹配串的末尾字符的下一位是否等于w[i]，不相等再循环。

上代码：

```

private static int[] getNextArray(String s){

char[] ch = s.toCharArray();

int i,j;

int[] next = new int[ch.length];

for(i = 1,j = 0; i < ch.length; i++){

while(j > 0 && ch[i] != ch[j]){

j = next[j - 1]; //j <next[j - 1],所以向前移动

}

if(ch[i] == ch[j]){

j++;

}

next[i] = j;

}

return next;

}

```

Ｗ字符串"ABCDABD"为例，运行结果如下：

![这里写图片描述](http://img.blog.csdn.net/20160614160131451)

#3.KMP

　　\*\*其实进行next数组求解的过程，类似于主串和子串进行匹配的过程，只不过是在next数组求解过程中，是子串和子串自己进行比较而已。\*\*

因此整个KMP算法的代码过程如下：

```

/\*\*

\* @param str1 被匹配的字符串

\* @param str2 子串

\* @return 布尔值

\*/

public static boolean kmp(String str1,String str2){

char[] strA = str1.toCharArray();

char[] strB = str2.toCharArray();

int[] next = getNextArray(str2);

int i,j; //这里i是从0开始的

for(i = 0,j = 0; i < strA.length; i++){

while(j > 0 && strA[i] != strB[j])

j = next[j-1];

if(strA[i] == strB[j]){

j++;

}

//匹配成功

if(j == strB.length){

return true;

}

}

return false;

}

private static int[] getNextArray(String s){

char[] ch = s.toCharArray();

int i,j;

//数组初始全部为0，所以next[0]=0

int[] next = new int[ch.length];

for(i = 1,j = 0; i < ch.length; i++){

while(j > 0 && ch[i] != ch[j]){

//j <next[j - 1],所以向前移动

j = next[j - 1];

}

if(ch[i] == ch[j]){

j++;

}

next[i] = j;

}

return next;

}

public static void main(String[] args) {

String T ="ＡＢＣＤＡＢＣＤＡＢＤＥ";

String W = "ＡＢＣＤＡＢＤ";

System.out.println(kmp(T,W));

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160614161443423)

## 第6节 二分查找与变种二分查找

#１.二分查找

　　二分搜索（binary search），也称折半搜索（half-interval search）、对数搜索（logarithmic search），是一种在\*\*有序数组\*\*中查找某一特定元素的搜索算法。搜索过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到。这种搜索算法每一次比较都使搜索范围缩小一半。

　　除直接在一个数组中查找元素外，可用在插入排序中。

##1.1 复杂度分析

　![这里写图片描述](http://img.blog.csdn.net/20160611105752324)

- 时间复杂度

　　折半搜索每次把搜索区域减少一半，时间复杂度为Ｏ(Log n) 。（n代表集合中元素的个数）

- 空间复杂度

　　 迭代：O(1)　　递归：O(log n)

##1.2 示例代码

```

// 递归版本

int binary\_search(const int arr[], int start, int end, int key) {

if (start > end)

return -1;

int mid = start + (end - start) / 2; //直接平均可能会溢位

if (arr[mid] > key)

return binary\_search(arr, start, mid - 1, key);

if (arr[mid] < key)

return binary\_search(arr, mid + 1, end, key);

return mid;

}

```

```

// while循环

int binary\_search(const int arr[], int start, int end, int key) {

int mid;

while (start <= end) {

mid = start + (end - start) / 2; //直接平均可能会溢位

if (arr[mid] < key)

start = mid + 1;

else if (arr[mid] > key)

end = mid - 1;

else

return mid;

}

return -1;

}

```

#2. 变种二分查找

##2.1 改变数组

　　升序数组a经过循环右移后，二分查找给定元素x的位置。

> 如a={1,2,3,4,5,6,7}，循环移动后a={5,6,7,1,2,3,4}

##2.２思路

　　移动后的数组分为两部分，两部分内部都是升序排列的，中值mid必然属于这两部分之一。然后根据下标判断查找值是否属于这一部分，循环缩小mid值，直到arr[mid]等于查找值就返回。

##2.3代码实现

```

public static void main(String[] args){

int arr[]={5,6,7,1,2,3,4};

int out = VariantBinaryFind(arr,arr.length-1,1);

System.out.println("下标为： "+ out);

}

/\*\*

\* @param a 数组

\* @param len 数组长度

\* @param x 查找的元素

\* @return 返回查找值得下标，不存在返回 -1

\*/

public static int VariantBinaryFind(int arr[],int len,int x){

int left = 0;

int right = len;

while(left<=right){

int mid = left + (right -left)/2;

if(arr[mid]==x){

return mid;

}

if(arr[mid] >= arr[left]) //mid在左边序列

{

if(arr[left] > x || arr[mid] < x)

left = mid + 1; //x在右边序列

else

right = mid - 1; //x在左边序列

}

else //mid在右边序列

{

if(arr[right] < x || arr[mid] > x)

right = mid - 1; //x在左边序列

else

left = mid + 1; //x在右边序列

}

}

return -1;

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160611115200340)

## 第7节 二叉树前中后、层次遍历算法

#1.二叉树

　　二叉树（Binary tree）是每个节点最多有两个子树的树结构。二叉树的子树有左右之分，次序不能颠倒。

　　二叉树的第i层至多有 ![这里写图片描述](https://wikimedia.org/api/rest\_v1/media/math/render/svg/de838b503259acc792dd682654445984ea6e4c6d)个结点；

　　深度为k的二叉树至多共有![这里写图片描述](https://wikimedia.org/api/rest\_v1/media/math/render/svg/76af5b5bc7d056faef7eef0c6efeea3a16adc156)个结点；

　　对任何一棵二叉树T，如果其终端结点数为![这里写图片描述](https://wikimedia.org/api/rest\_v1/media/math/render/svg/63584d203ecb012a7bcb90f422408bbfe4018956)，度为2的节点数为 ![这里写图片描述](https://wikimedia.org/api/rest\_v1/media/math/render/svg/840e456e3058bc0be28e5cf653b170cdbfcc3be4)，则 ![这里写图片描述](https://wikimedia.org/api/rest\_v1/media/math/render/svg/aa746d23c7cab9729c356a8e12e734e03de9fa64)。

　　与树不同，树的节点个数至少为1，而二叉树的节点个数可以为0；树中节点的最大度数没有限制，而二叉树节点的最大度数为2；树的节点无左、右之分，而二叉树的结点有左、右之分。

##1.1前(先)序、中序、后序遍历

　　遍历二叉树：L、D、R分别表示遍历左子树、访问根结点和遍历右子树，则先(根)序遍历二叉树的顺序是DLR，中(根)序遍历二叉树的顺序是LDR，后(根)序遍历二叉树的顺序是LRD。还有按层遍历二叉树。这些方法的时间复杂度都是O(n)，n为结点个数。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/f/fe/Bitree.JPG)

用二叉树表示上述表达式：a+b\*(c-d)-e/f

- 先序遍历的序列是：-+a\*b-cd/ef

- 中序遍历的序列是：a+b\*c-d-e/f

- 后序遍历的序列是：abcd-\*+ef/-

##1.２二叉树的存储

###1.２.1顺序存储

　　二叉树可以用\*\*数组或线性表\*\*来存储，而且如果这是\*\*满二叉树\*\*，这种方法不会浪费空间。用这种紧凑排列，如果一个结点的索引为i，它的子结点能在索引2i+1和2i+2找到，并且它的父节点（如果有）能在索引floor((i-1)/2)找到（假设根节点的索引为0）。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/8/86/Binary\_tree\_in\_array.svg/370px-Binary\_tree\_in\_array.svg.png)

###1.２.２二叉链表存储

　　二叉树通常用树结点结构来存储。有时也包含指向唯一的父节点的指针。如果一个结点的子结点个数小于2，一些子结点指针可能为空值，或者为特殊的哨兵结点。 使用链表能避免顺序储存浪费空间的问题，算法和结构相对简单，但使用二叉链表，由于缺乏父链的指引，在找回父节点时需要重新扫描树得知父节点的节点地址。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/8/82/Eclb.jpg)

###1.２.３三叉链表存储

　　改进于二叉链表，增加父节点的指引，能更好地实现节点间的访问，不过算法相对复杂。 当二叉树用三叉链表表示时，有N个结点，就会有N+2个空指针。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/6/60/3clb.jpg)

##1.３将n叉树转换为二叉树

一般有序树和二叉树之间有一一映射关系，能进行相互转换。

n叉树转换为二叉树的方法：二叉树中结点x的左子结点为n叉树中结点x的左子结点；二叉树中结点x的右子结点为n叉树中结点x的第一个右边的同级结点y。

例如，在左边的树中，A有6个子结点{B,C,D,E,F,G}。它能被转换成右边的二叉树。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/0/02/Nary\_to\_binary\_tree\_conversion.png)

- 将一棵树转换为二叉树的方法：

1、在兄弟之间加一连接；

2、对每个结点，除了其左孩子外，去除其与其余孩子之间的联系；

3、以树的根结点为轴心，将整树顺时针转45度。

##1.4二叉树遍历（Java）

节点描述：

```

private class Node{

// 左节点

private Node leftChild;

// 右节点

private Node rightChild;

// 节点对应的值

private int data;

public Node(int data){

this.leftChild = null;

this.rightChild = null;

this.data = data;

}

}

```

###1.4.1前中后续遍历（递归）

递归遍历：

```

/\*

\*前序遍历二叉树

\* \*/

public void preOrder(Node node){

if(node != null){

System.out.print(node.data);

preOrder(node.leftChild);

preOrder(node.rightChild);

}

}

/\*

\*中序遍历二叉树

\* \*/

public void inOrder(Node node){

if(node != null){

inOrder(node.leftChild);

System.out.print(node.data);

inOrder(node.rightChild);

}

}

/\*

\*后序遍历二叉树

\* \*/

public void postOrder(Node node){

if(node != null){

postOrder(node.leftChild);

postOrder(node.rightChild);

System.out.print(node.data);

}

```

###1.4.2前中后续遍历（非递归）

```

/\*\*

\*

\* 【前序】

\* 利用栈实现循环先序遍历二叉树

\* 这种实现类似于图的深度优先遍历（DFS）

\* 维护一个栈，将根节点入栈，然后只要栈不为空，出栈并访问，接着依次将访问节点的右节点、左节点入栈。

\* 这种方式应该是对先序遍历的一种特殊实现（看上去简单明了），但是不具备很好的扩展性，在中序和后序方式中不适用

\*/

public static void preOrderStack(Node root){

if(root==null)return;

Stack<Node> s=new Stack<Node>();

s.push(root);

while(!s.isEmpty()){

Node temp=s.pop();

System.out.println(temp.value);

if(temp.right!=null) s.push(temp.right);

if(temp.left!=null) s.push(temp.left);

}

}

/\*\*

\*

\* 【中序】

\* 利用栈模拟递归过程实现循环中序遍历二叉树

\* 访问的时间是在左子树都处理完直到null的时候出栈并访问。

\*/

public static void inOrderStack(Node root){

if(root==null)return;

Stack<Node> s=new Stack<Node>();

while(root!=null||!s.isEmpty()){

while(root!=null){

s.push(root);//先访问再入栈

root=root.left;

}

root=s.pop();

System.out.println(root.value);

root=root.right;//如果是null，出栈并处理右子树

}

}

/\*\*

\*

\* 【后续】

\* 后序遍历不同于先序和中序，它是要先处理完左右子树，然后再处理根(回溯)，所以需要一个记录哪些节点已经被访问的结构(可以在树结构里面加一个标记)，这里可以用map实现

\*/

public static void postOrderStack(Node root){

if(root==null)return;

Stack<Node> s=new Stack<Node>();

Map<Node,Boolean> map=new HashMap<Node,Boolean>();

s.push(root);

while(!s.isEmpty()){

Node temp=s.peek();

if(temp.left!=null&&!map.containsKey(temp.left)){

temp=temp.left;

while(temp!=null){

if(map.containsKey(temp))break;

else s.push(temp);

temp=temp.left;

}

continue;

}

if(temp.right!=null&&!map.containsKey(temp.right)){

s.push(temp.right);

continue;

}

Node t=s.pop();

map.put(t,true);

System.out.println(t.value);

}

}

```

###1.4.3广度优先遍历（层次遍历）

```

/\*\*

\* @param root 树根节点

\* 层序遍历二叉树，用队列实现，先将根节点入队列，只要队列不为空，然后出队列，并访问，接着讲访问节点的左右子树依次入队列

\*/

public static void levelTravel(Node root){

if(root==null)return;

Queue<Node> q=new LinkedList<Node>();

q.add(root);

while(!q.isEmpty()){

Node temp = q.poll();

System.out.println(temp.value);

if(temp.left!=null)q.add(temp.left);

if(temp.right!=null)q.add(temp.right);

}

}

```

## 第8节 图的BFS、DFS、prim、Dijkstra算法

#1.图的定义

　　图G是由顶点的有穷集合，以及顶点之间的关系组成，顶点的集合记为V，顶点之间的关系构成边的集合E，G=(V,E).

　　如果给图的每条边规定一个方向，那么得到的图称为\*\*有向图\*\*，其边也称为有向边。在有向图中，与一个节点相关联的边有出边和入边之分，而与一个有向边关联的两个点也有始点和终点之分。相反，边没有方向的图称为\*\*无向图\*\*。

#２.图的存储

　　一般图的存储有两种方式

　　1)邻接表：需要保存一个顺序存储的顶点表和每个顶点上的边的链接表。

　　2)相邻矩阵：用一个矩阵来保持边的情况

![这里写图片描述](http://img.blog.csdn.net/20160613103617205)

下图展示了两种表示上面这个图的方法：

![这里写图片描述](http://img.blog.csdn.net/20160613103702050)

#３.图的遍历

　　①　DFS（Depth First Search）深度优先搜索，为每个顶点设立一个“访问标志”。首先将图中每个顶点的访问标志设为 FALSE, 之后搜索图中每个顶点，如果未被访问，则以该顶点为起始点，进行遍历。

　　若当前访问的顶点的邻接顶点有未被访问的，则任选一个访问之。反之，退回到最近访问过的顶点；直到与起始顶点相通的全部顶点都访问完毕；

> 遍历图的过程实质上是对每个顶点查找其邻接点的过程，所耗费的时间取决于所采用的存储结构。

对图中的每个顶点至多调用1次DFS算法，因为一旦某个顶点已访问过，则不再从它出发进行搜索。

> 邻接链表表示：查找每个顶点的邻接点所需时间为O(e)，e为边(弧)数，算法时间复杂度为O(n+e)

> 数组表示：查找每个顶点的邻接点所需时间为O(n2)，n为顶点数，算法时间复杂度为O(n2)

　　②　BFS（Breadth First Search）广度优先遍历，从图的某一结点出发，首先依次访问该结点的所有邻接顶点 Vi1, Vi2, …, Vin 再按这些顶点被访问的先后次序依次访问与它们相邻接的所有未被访问的顶点，重复此过程，直至所有顶点均被访问为止。

##３.１遍历代码（邻接矩阵）

```

// 邻接矩阵存储图

public class Graph {

// 顶点数

private int number = 9;

// 记录顶点是否被访问

private boolean[] flag;

// 顶点

private String[] vertexs = { "A", "B", "C", "D", "E", "F", "G", "H", "I" };

// 边

private int[][] edges = {

{ 0, 1, 0, 0, 0, 1, 1, 0, 0 },

{ 1, 0, 1, 0, 0, 0, 1, 0, 1 },

{ 0, 1, 0, 1, 0, 0, 0, 0, 1 },

{ 0, 0, 1, 0, 1, 0, 1, 1, 1 },

{ 0, 0, 0, 1, 0, 1, 0, 1, 0 },

{ 1, 0, 0, 0, 1, 0, 1, 0, 0 },

{ 0, 1, 0, 1, 0, 1, 0, 1, 0 },

{ 0, 0, 0, 1, 1, 0, 1, 0, 0 },

{ 0, 1, 1, 1, 0, 0, 0, 0, 0 }

};

// 图的深度遍历操作(递归)

void DFSTraverse() {

flag = new boolean[number];

for (int i = 0; i < number; i++) {

if (flag[i] == false) {// 当前顶点没有被访问

DFS(i);

}

}

}

// 图的深度优先递归算法

void DFS(int i) {

flag[i] = true;// 第i个顶点被访问

System.out.print(vertexs[i] + " ");

for (int j = 0; j < number; j++) {

if (flag[j] == false && edges[i][j] == 1) {

DFS(j);

}

}

}

// 图的广度遍历操作

void BFSTraverse() {

flag = new boolean[number];

Queue<Integer> queue = new LinkedList<Integer>();

for (int i = 0; i < number; i++) {

if (flag[i] == false) {

flag[i] = true;

System.out.print(vertexs[i] + " ");

queue.add(i);

while (!queue.isEmpty()) {

int j = queue.poll();

for (int k = 0; k < number; k++) {

if (edges[j][k] == 1 && flag[k] == false) {

flag[k] = true;

System.out.print(vertexs[k] + " ");

queue.add(k);

}

}

}

}

}

}

public static void main(String[] args) {

Graph graph = new Graph();

System.out.println("-----------DFS-----------------");

graph.DFSTraverse();

System.out.println();

System.out.println("-----------BFS-----------------");

graph.BFSTraverse();

}

}

```

运行结果：

![这里写图片描述](http://img.blog.csdn.net/20160613112121895)

#4.最小生成树prim算法

　　从单一顶点开始，普里姆算法按照以下步骤逐步扩大树中所含顶点的数目，直到遍及连通图的所有顶点。

1. 输入：一个\*\*加权连通图\*\*，其中顶点集合为V，边集合为E；

2. 初始化：Vnew = {x}，其中x为集合V中的任一节点（起始点），Enew = {}；

3. 重复下列操作，直到Vnew = V：

3.1 在集合E中选取权值最小的边（u, v），其中u为集合Vnew中的元素，而v则是V中没有加入Vnew的顶点（如果存在有多条满足前述条件即具有相同权值的边，则可任意选取其中之一）；

3.2 \*\*将v加入集合Vnew\*\*中，将（u, v）加入集合Enew中；

4. 输出：使用集合Vnew和Enew来描述所得到的最小生成树。

> 此为原始的加权连通图。每条边一侧的数字代表其权值。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/a/a8/Prim\_Algorithm\_0.svg/200px-Prim\_Algorithm\_0.svg.png)

> 顶点D被任意选为起始点。顶点A、B、E和F通过单条边与D相连。A是距离D最近的顶点，因此将A及对应边AD以高亮表示。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Prim\_Algorithm\_1.svg/200px-Prim\_Algorithm\_1.svg.png)

> 下一个顶点为距离D或A最近的顶点。B距D为9，距A为7，E为15，F为6。因此，F距D或A最近，因此将顶点F与相应边DF以高亮表示。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/f/f5/Prim\_Algorithm\_2.svg/200px-Prim\_Algorithm\_2.svg.png)

> 算法继续重复上面的步骤。距离A为7的顶点B被高亮表示。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/b/b4/Prim\_Algorithm\_3.svg/200px-Prim\_Algorithm\_3.svg.png)

> 在当前情况下，可以在C、E与G间进行选择。C距B为8，E距B为7，G距F为11。E最近，因此将顶点E与相应边BE高亮表示。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/7/7e/Prim\_Algorithm\_4.svg/200px-Prim\_Algorithm\_4.svg.png)

> 这里，可供选择的顶点只有C和G。C距E为5，G距E为9，故选取C，并与边EC一同高亮表示。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/e/e3/Prim\_Algorithm\_5.svg/200px-Prim\_Algorithm\_5.svg.png)

> 顶点G是唯一剩下的顶点，它距F为11，距E为9，E最近，故高亮表示G及相应边EG。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/9/99/Prim\_Algorithm\_6.svg/200px-Prim\_Algorithm\_6.svg.png)

> 现在，所有顶点均已被选取，图中绿色部分即为连通图的最小生成树。在此例中，最小生成树的权值之和为39。

![这里写图片描述](https://upload.wikimedia.org/wikipedia/commons/thumb/b/b9/Prim\_Algorithm\_7.svg/200px-Prim\_Algorithm\_7.svg.png)

##4.1 java代码实现

```

public class Prim {

//结点集

public static List<Vertex> vertexList = new ArrayList<Vertex>();

//边集

public static List<Edge> EdgeList = new ArrayList<Edge>();

//已经访问过的结点集

public static List<Vertex> containVertexList = new ArrayList<Vertex>();

public static void main(String[] args) {

primTree();

}

public static void primTree(){

//初始化图

buildGraph();

//起始点

Vertex start = vertexList.get(0);

containVertexList.add(start);

for(int n=0;n<vertexList.size()-1;n++){

//临时节点a

Vertex temp = new Vertex(start.key);

Edge tempedge = new Edge(start,start,1000);

for(Vertex v : containVertexList){

for(Edge e : EdgeList){

//找出相邻最小边

if(e.start==v && !containVertex(e.end)){

if(e.Len<tempedge.Len){

temp = e.end;

tempedge = e;

}

}

}

}

//把该点加入

containVertexList.add(temp);

}

//打印输出

Iterator it = containVertexList.iterator();

while(it.hasNext()){

Vertex v =(Vertex) it.next();

System.out.println(v.key);

}

}

public static void buildGraph() {

Vertex v1 = new Vertex("a");

Prim.vertexList.add(v1);

Vertex v2 = new Vertex("b");

Prim.vertexList.add(v2);

Vertex v3 = new Vertex("c");

Prim.vertexList.add(v3);

Vertex v4 = new Vertex("d");

Prim.vertexList.add(v4);

Vertex v5 = new Vertex("e");

Prim.vertexList.add(v5);

addEdge(v1, v2, 6);

addEdge(v1, v3, 7);

addEdge(v2, v3, 8);

addEdge(v2, v5, 4);

addEdge(v2, v4, 5);

addEdge(v3, v4, 3);

addEdge(v3, v5, 9);

addEdge(v5, v4, 7);

addEdge(v5, v1, 2);

addEdge(v4, v2, 2);

}

public static void addEdge(Vertex a, Vertex b, int w) {

Edge e = new Edge(a, b, w);

Prim.EdgeList.add(e);

}

public static boolean containVertex(Vertex vte){

for(Vertex v : containVertexList){

if(v.key.equals(vte.key))

return true;

}

return false;

}

}

class Vertex {

String key;

Vertex(String key){

this.key = key;

}

}

class Edge{

Vertex start;

Vertex end;

int Len;

Edge(Vertex start,Vertex end,int key){

this.start = start;

this.end = end;

this.Len = key;

}

}

```

#５.Dijkstra算法

　　使用了\*\*广度优先搜索\*\*解决非负权有向图的\*\*单源最短路径\*\*问题，算法最终得到一个最短路径树（一个节点到其他所有节点的最短路径）。该算法常用于路由算法或者作为其他图算法的一个子模块。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。

算法思想：

> 　　设G=(V,E)是一个带权有向图，把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合（用S表示，初始时S中只有一个源点，以后每求得一条最短路径 , 就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了）

>

> 　　第二组为其余未确定最短路径的顶点集合（用U表示），按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

算法步骤：

a.　初始时，S只包含源点，即S＝{v}，v的距离为0。U包含除v外的其他顶点，即:U={其余顶点}，若v与U中顶点u有边，则（u,v）正常有权值，若u不是v的出边邻接点，则（u,v）权值为∞。

b.　从U中选取一个距离v最小的顶点k，把k，加入S中（该选定的距离就是v到k的最短路径长度）。

c.　以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值，修改后的距离值的顶点k的距离加上边上的权。

d.　重复步骤b和c直到所有顶点都包含在S中。

　　![这里写图片描述](http://img.blog.csdn.net/20160613154953498)

　　Dijkstra算法演示

##５.１Dijkstra算法（Java实现）

```

public class Dijkstra {

private static int M = 10000; //此路不通

public static void main(String[] args) {

//邻接矩阵

int[][] weight = {

{0,10,M,30,100},

{M,0,50,M,M},

{M,M,0,M,10},

{M,M,20,0,60},

{M,M,M,M,0}

};

int start=0;

int[] shortPath = dijkstra(weight,start);

for(int i = 0;i < shortPath.length;i++)

System.out.println("从"+start+"出发到"+i+"的最短距离为："+shortPath[i]);

}

public static int[] dijkstra(int[][] weight, int start) {

//接受一个有向图的权重矩阵，和一个起点编号start（从0编号，顶点存在数组中）

//返回一个int[] 数组，表示从start到它的最短路径长度

int n = weight.length; //顶点个数

int[] shortPath = new int[n]; //保存start到其他各点的最短路径

String[] path = new String[n]; //保存start到其他各点最短路径的字符串表示

for(int i=0;i<n;i++)

path[i]=new String(start+"-->"+i);

int[] visited = new int[n]; //标记当前该顶点的最短路径是否已经求出,1表示已求出

//初始化，第一个顶点已经求出

shortPath[start] = 0;

visited[start] = 1;

for(int count = 1; count < n; count++) { //要加入n-1个顶点

int k = -1; //选出一个距离初始顶点start最近的未标记顶点

int dmin = Integer.MAX\_VALUE;

for(int i = 0; i < n; i++) {

if(visited[i] == 0 && weight[start][i] < dmin) {

dmin = weight[start][i];

k = i;

}

}

//将新选出的顶点标记为已求出最短路径，且到start的最短路径就是dmin

shortPath[k] = dmin;

visited[k] = 1;

//以k为中间点，修正从start到未访问各点的距离

for(int i = 0; i < n; i++) {

if(visited[i] == 0 && weight[start][k] + weight[k][i] < weight[start][i]) {

weight[start][i] = weight[start][k] + weight[k][i];

path[i] = path[k] + "-->" + i;

}

}

}

for(int i = 0; i < n; i++) {

System.out.println("从"+start+"出发到"+i+"的最短路径为："+path[i]);

}

System.out.println("=====================================");

return shortPath;

}

}

```

运行结果：

```

从0出发到0的最短路径为：0-->0

从0出发到1的最短路径为：0-->1

从0出发到2的最短路径为：0-->3-->2

从0出发到3的最短路径为：0-->3

从0出发到4的最短路径为：0-->3-->2-->4

=====================================

从0出发到0的最短距离为：0

从0出发到1的最短距离为：10

从0出发到2的最短距离为：50

从0出发到3的最短距离为：30

从0出发到4的最短距离为：60

```

## 第9节 字符串操作

> 在JAVA语言中，字符串数据实际上由String类所实现的。Java字符串类分为两类：一类是在程序中不会被改变长度的不变字符串；二类是在程序中会被改变长度的可变字符串。Java环境为了存储和维护这两类字符串提供了 String和StringBuffer两个类。

#String的一些常用操作

##字符串创建

```

String　str="This is a String";

```

或者

```

String　str=new String（"This is a String");

```

　　第一种的的效率比较高，详细的可以看：

　　[String、StringBuffer与StringBuilder区别和联系（源码）](http://blog.csdn.net/amazing7/article/details/51219397)

##字符串长度

```

String str= "This is a String";

int len =str.length();

```

##指定字符或子字符串的位置

```

String str="Thisis a String";

Int index1 =str.indexOf("i"); //2

Int index２＝str.lastIndexOf("i")； //12

```

##判断两个字符串是否相等

```

String str="This　is a String";

Boolean result=str.equals("This is another String");

```

##得到指定位置的字符

```

String str="This　is a String";

char chr=str.charAt(3);

```

##截取子字符串

```

str＝str.substring(int beginIndex);

```

　截取掉str从首字母起长度为beginIndex的字符串，将剩余字符串赋值给str；

```

str＝str.substring(int beginIndex，int endIndex);

```

　截取str中从beginIndex开始至endIndex结束时的字符串，并将其赋值给str;

##字符串合并

```

String str="This　is a String";

String str1=str.concat("Test"); //str1="This　is a String Test"

```

##字符串大小写的转换

```

str.toLowerCase();

```

大写转小写

```

str.toUpperCase();

```

小写转大写

##去掉字符串中开头和结尾处的空格

```

String str="This　is a String ";

String str1=str.trim(); //str1="This is a String"

```

#StringBuffer的一些常用操作

##初始化

```

StringBuffer s = new StringBuffer();

```

这样初始化出的StringBuffer对象是一个空的对象。

如果需要创建带有内容的StringBuffer对象，则可以使用：

```

StringBuffer s = new StringBuffer(“abc”);

```

##append方法

　　追加内容到当前StringBuffer对象的末尾，类似于字符串的连接。调用该方法以后，StringBuffer对象的内容也发生改变。

```

StringBuffer sb = new StringBuffer(“abc”);

sb.append(true);

```

对象sb的值将变成”abctrue”

##deleteCharAt方法

删除指定位置的字符，然后将剩余的内容形成新的字符串

```

StringBuffer sb = new StringBuffer(“Test”);

sb. deleteCharAt(1);

```

对象sb的值变为”Tst”

还存在一个功能类似的delete方法：

```

public StringBuffer delete(int start,int end)

```

该方法的作用是删除指定区间以内的所有字符，包含start，不包含end索引值的区间。

##insert方法

在StringBuffer对象中插入内容，然后形成新的字符串。

```

StringBuffer sb = new StringBuffer(“TestString”);

　　sb.insert(4,false);

```

在对象sb的索引值4的位置插入false值，形成新的字符串，则执行以后对象sb的值是”TestfalseString”。

##reverse方法

将StringBuffer对象中的内容反转，然后形成新的字符串

```

StringBuffer sb = new StringBuffer(“abc”);

sb.reverse();

```

对象sb中的内容将变为”cba”

##setCharAt方法

修改对象中指定索引值位置的字符为新的字符

```

StringBuffer sb = new StringBuffer(“abc”);

sb.setCharAt(1,’D’);

```

sb的值将变成”aDc”

##trimToSize方法

将StringBuffer对象的中存储空间缩小到和字符串长度一样的长度，减少空间的浪费。

#常见问题

- 字符串比较,使用 "==" 还是 equals() ?

　　[源码分析equals与==的区别](http://blog.csdn.net/amazing7/article/details/51219323)

- 在switch语句中使用String作为case条件?

　　[Switch能否用string做参数？](http://blog.csdn.net/amazing7/article/details/51219315)

- String转换为数字

```

int age = Integer.parseInt("10");

long id = Long.parseLong("190"); // 假如值可能很大.

```

　　非常大的数字请使用Long

- 通过空白字符拆分字符串

```

String[] strArray = aString.split("\\s+");

```

　　String 的 split()方法接收的字符串会被当做正则表达式解析,

　 "\s"代表空白字符,如空格" "（tab制表符"\t", 换行"\n",回车"\r"）

　　而编译器在对源代码解析时,也会进行一次\*\*字面量转码\*\*,所以需要"\\s".

- 将String转换为日期

```

SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");

String str = "2016-05-07";

Date date = format.parse(str);

System.out.println(format.format(date));

```

## 第10节 数组与链表的优缺点和区别

#概述

　　数组　是将元素在内存中\*\*连续存放\*\*，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要\*\*快速访问数据，很少插入和删除元素，就应该用数组。\*\*

　　链表　中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起，每个结点包括两个部分：一个是存储 数据元素 的　数据域，另一个是存储下一个结点地址的 指针。

　　如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要\*\*经常插入和删除元素你就需要用链表\*\*。

#内存存储区别

- 数组从\*\*栈\*\*中分配空间, 对于程序员方便快速,但自由度小。

- 链表从\*\*堆\*\*中分配空间, 自由度大但申请管理比较麻烦.

#逻辑结构区别

- 数组必须事先定义\*\*固定的长度\*\*（元素个数），不能适应数据动态地增减的情况。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费。

- 链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删除数据项时，需要移动其它数据项）

#总结

1、存取方式上，数组可以顺序存取或者随机存取，而链表只能顺序存取；

2、存储位置上，数组逻辑上相邻的元素在物理存储位置上也相邻，而链表不一定；

3、存储空间上，链表由于带有指针域，存储密度不如数组大；

4、按序号查找时，数组可以随机访问，时间复杂度为O(1)，而链表不支持随机访问，平均需要O(n)；

5、按值查找时，若数组无序，数组和链表时间复杂度均为O(1)，但是当数组有序时，可以采用折半查找将时间复杂度降为O(logn)；

6、插入和删除时，数组平均需要移动n/2个元素，而链表只需修改指针即可；

7、空间分配方面：

　　数组在静态存储分配情形下，存储元素数量受限制，动态存储分配情形下，虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且如果内存中没有更大块连续存储空间将导致分配失败；

　　链表存储的节点空间只在需要的时候申请分配，只要内存中有空间就可以分配，操作比较灵活高效；

## 第11节 红黑树

#1. 概述

　　红黑树（Red Black Tree） 是一种自平衡二叉查找树，红黑树和AVL树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。

　　它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，并且在实践中是高效的： 它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。

#2. 性质

　红黑树是每个节点都带有颜色属性的\*\*二叉查找树\*\*，颜色为红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

１、节点是红色或黑色。

２、根是黑色。

３、所有叶子都是黑色（叶子是NIL节点）。

４、每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）

５、从任一节点到其每个叶子的所有简单路径（不包含重复的点的路径）都包含相同数目的黑色节点。

下面是一个具体的红黑树的图例：

![这里写图片描述](http://img.blog.csdn.net/20160612153918096)

#3. 操作

　　因为每一个红黑树也是一个特化的二叉查找树，因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。然而，在红黑树上进行插入操作和删除操作会导致不再匹配红黑树的性质。恢复红黑树的性质需要少量（O(log n)）的\*\*颜色变更\*\*（实际是非常快速的）和不超过三次树\*\*旋转\*\*（对于插入操作是两次）。虽然插入和删除很复杂，但操作时间仍可以保持为O(log n)次。

##3.１插入操作

　　我们首先以二叉查找树的方法增加节点并标记它为红色。（如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整。）下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语叔父节点来指一个节点的父节点的兄弟节点。

注意：

> 性质1和性质3总是保持着。

性质4只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。

性质5只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。

![这里写图片描述](http://img.blog.csdn.net/20160612154806763)

　　在下面将要插入的节点标为N，N的父节点标为P，N的祖父节点标为G，N的叔父节点标为U。在图中展示的任何颜色要么是由它所处情形这些所作的假定，要么是假定所暗含（imply）的。

> \*\*情形1\*\*:

　　\*\*新节点N位于树的根上\*\*，没有父节点。在这种情形下，我们把它重绘为黑色以满足性质2。因为它在每个路径上对黑节点数目增加一，性质5匹配。

> \*\*情形２\*\*:

　　\*\*新节点的父节点P是黑色\*\*，所以性质4没有失效（新节点是红色的）。在这种情形下，树仍是有效的。性质5也未受到威胁，尽管新节点N有两个黑色叶子子节点；但由于新节点N是红色，通过它的每个子节点的路径就都有同通过它所替换的黑色的叶子的路径同样数目的黑色节点，所以依然满足这个性质。

> \*\*情形３\*\*:

　　\*\*如果父节点P和叔父节点U二者都是红色\*\*，（此时新插入节点N做为P的左子节点或右子节点都属于情形3，这里右图仅显示N做为P左子的情形）则我们可以将它们两个重绘为黑色并重绘祖父节点G为红色（用来保持性质5）。现在我们的新节点N有了一个黑色的父节点P。因为通过父节点P或叔父节点U的任何路径都必定通过祖父节点G，在这些路径上的黑节点数目没有改变。但是，红色的祖父节点G可能是根节点，这就违反了性质2，也有可能祖父节点G的父节点是红色的，这就违反了性质4。

　　为了解决这个问题，我们在祖父节点G上递归地进行情形1的整个过程。（把G当成是新加入的节点进行各种情形的检查）

![这里写图片描述](http://img.blog.csdn.net/20160612155222957)

> \*\*情形４\*\*:

　　\*\*父节点P是红色而叔父节点U是黑色或缺少\*\*，并且新节点N是其父节点P的\*\*右子节点\*\*而父节点P又是其父节点的左子节点。在这种情形下，我们进行一次\*\*左旋转\*\*调换新节点和其父节点的角色;接着，我们按情形5处理以前的父节点P以解决仍然失效的性质4。注意这个改变会导致某些路径通过它们以前不通过的新节点N（比如图中1号叶子节点）或不通过节点P（比如图中3号叶子节点），但由于这两个节点都是红色的，所以性质5仍有效。

![这里写图片描述](http://img.blog.csdn.net/20160612155611583)

> \*\*情形５\*\*:

　　\*\*父节点P是红色而叔父节点U是黑色或缺少，新节点N是其父节点的左子节点\*\*，而父节点P又是其父节点G的左子节点。在这种情形下，我们进行针对祖父节点G的一次右旋转；在旋转产生的树中，以前的父节点P现在是新节点N和以前的祖父节点G的父节点。我们知道以前的祖父节点G是黑色，否则父节点P就不可能是红色（如果P和G都是红色就违反了性质4，所以G必须是黑色）。我们切换以前的父节点P和祖父节点G的颜色，结果的树满足性质4。性质5也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过祖父节点G，现在它们都通过以前的父节点P。在各自的情形下，这都是三个节点中唯一的黑色节点。

![这里写图片描述](http://img.blog.csdn.net/20160612155811854)

##3.２删除操作

　　对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们找到要么在它的左子树中的最大元素、要么在它的右子树中的最小元素，并把它的值转移到要删除的节点中。

　　我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为N（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为S。在下面的示意图中，我们还是使用P称呼N的父亲，SL称呼S的左儿子，SR称呼S的右儿子。

> \*\*情形1\*\*:

　　\*\*N是新的根\*\*。在这种情形下，我们就做完了。我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

> \*\*情形２\*\*:

　　\*\*S是红色\*\*。在这种情形下我们在N的父亲上做\*\*左旋转\*\*，把红色兄弟转换成N的祖父，我们接着对调N的父亲和祖父的颜色。完成这两个操作后，尽管所有路径上黑色节点的数目没有改变，但现在N有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色S的一个儿子），所以我们可以接下去按情形4、情形5或情形6来处理。

　![这里写图片描述](http://img.blog.csdn.net/20160612160739148)

![这里写图片描述](http://img.blog.csdn.net/20160418153051177)

> \*\*情形３\*\*:

　　\*\*N的父亲、S和S的儿子都是黑色的。\*\*在这种情形下，我们简单的重绘S为红色。结果是通过S的所有路径，它们就是以前不通过N的那些路径，都少了一个黑色节点。因为删除N的初始的父亲使通过N的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过P的所有路径现在比不通过P的路径少了一个黑色节点，所以仍然违反性质5。要修正这个问题，我们要从情形1开始，在P上做\*\*重新平衡\*\*处理。

![这里写图片描述](http://img.blog.csdn.net/20160612160910088)

> \*\*情形４\*\*:

　　\*\*S和S的儿子都是黑色，但是N的父亲是红色\*\*。在这种情形下，我们简单的交换N的兄弟和父亲的颜色。这不影响不通过N的路径的黑色节点的数目，但是它在通过N的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。

![这里写图片描述](http://img.blog.csdn.net/20160612161147669)

> \*\*情形５\*\*:

　　S是黑色，S的左儿子是红色，S的右儿子是黑色，而N是它父亲的左儿子。在这种情形下我们在S上做\*\*右旋转\*\*，这样S的左儿子成为S的父亲和N的新兄弟。我们接着交换S和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在N有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了情形6。N和它的父亲都不受这个变换的影响。

![这里写图片描述](http://img.blog.csdn.net/20160612161350762)

![这里写图片描述](http://img.blog.csdn.net/20160418153412158)

> \*\*情形６\*\*:

　　 S是黑色，S的右儿子是红色，而N是它父亲的左儿子。在这种情形下我们在N的父亲上做左旋转，这样S成为N的父亲（P）和S的右儿子的父亲。我们接着交换N的父亲和S的颜色，并使S的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以性质3没有被违反。但是，N现在增加了一个黑色祖先：要么N的父亲变成黑色，要么它是黑色而S被增加为一个黑色祖父。所以，通过N的路径都增加了一个黑色节点。

此时，如果一个路径不通过N，则有两种可能性：

- 它通过N的新兄弟。那么它以前和现在都必定通过S和N的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。

- 它通过N的新叔父，S的右儿子。那么它以前通过S、S的父亲和S的右儿子，但是现在只通过S，它被假定为它以前的父亲的颜色，和S的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。

在任何情况下，在这些路径上的黑色节点数目都没有改变。所以我们恢复了性质4。在示意图中的白色节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。

![这里写图片描述](http://img.blog.csdn.net/20160612161557794)

## 第12节 队列和栈

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

一、队列

</font>

</td>

</tr>

</table>

##

#队列

　　队列是一种操作受限制的线性表，它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作。进行插入操作的端称为队尾，进行删除操作的端称为队头。

　　队列中没有元素时，称为空队列。在队列这种数据结构中，最先插入的元素将是最先被删除的元素；反之最后插入的元素将是最后被删除的元素，因此队列又称为“先进先出”（FIFO—first in first out）的线性表。

队列空的条件：front=rear

队列满的条件： rear = MAXSIZE

#顺序队列

　　建立顺序队列结构必须为其静态分配或动态申请一片连续的存储空间，并设置两个指针进行管理。一个是队头指针front，它指向队头元素；另一个是队尾指针rear，它指向下一个入队元素的存储位置．

　　![这里写图片描述](http://img.blog.csdn.net/20160510130505383)

顺序队列中的溢出现象：

- 下溢：当队列为空时，做出队运算产生的溢出现象。“下溢”是正常现象，常用作程序控制转移的条件。

- 真上溢：当队列满时，做进栈运算产生空间溢出的现象。“真上溢”是一种出错状态，应设法避免。

- 假上溢：由于入队和出队操作中，头尾指针只增加不减小，致使被删元素的空间永远无法重新利用。当队列中实际的元素个数远远小于向量空间的规模时，也可能由于尾指针已超越向量空间的上界而不能做入队操作。该现象称为"假上溢"现象。

##java中Queue

　　Queue继承于Collection，除了基本的 Collection 操作外，队列还提供其他的插入、提取和检查操作。每个方法都存在两种形式：一种抛出异常（操作失败时），另一种返回一个特殊值（null 或 false，具体取决于操作）。

　　![这里写图片描述](http://img.blog.csdn.net/20160510131629258)

- add()和offer()都是将指定的元素插入队列

　　　　add() 在成功时返回 true，如果当前没有可用的空间，则抛出 　　IllegalStateException。

　　　　offer()当使用有容量限制的队列时,无法插入元素，而只是抛出一个异常。

- element() 和 peek() 返回但不移除队列的头，如果队列为空，peek()返回 null，element()抛出异常。

- remove() 和 poll() 方法可移除和返回队列的头，它们仅在队列为空时其行为有所不同：remove() 方法抛出一个异常，而 poll() 方法则返回 null。

一般使用：

```

public class TestQueue

{

/\*\*

\* @param args

\* @author JavaAlpha

\* Info 测试队列

\*/

public static void main(String[] args)

{

Queue<String> queue = new LinkedList<String>();

queue.offer("1");//插入元素

queue.offer("2");

queue.offer("3");

//打印元素个数

System.out.println("queue.size()"+queue.size());

//遍历打印所有的元素（按插入顺序）

for (String string : queue)

{

System.out.println(string);

}

}

}

```

结果：

```

queue.size()3

1

2

3

```

　　JDK中包含了 双向顺序队列ArrayDeque、双向链式队列LinkedList和PriorityQueue优先队列。

　　ArrayDeque包括顺序栈和顺序队列，LinkedList包含链式栈和链式队列。ArrayDeque和LinkedList都是线程不安全的。

java顺序队列的实现：

```

package lang;

import java.io.Serializable;

import java.util.Arrays;

public class ArrayQueue<T> implements Serializable{

private static final long serialVersionUID = 7333344126529379197L;

private int DEFAULT\_SIZE = 10;

private int capacity;//保存数组的长度

private Object[] elementData;//定义一个数组用于保存顺序队列的元素

private int front = 0;//队头

private int rear = 0;//队尾

//以默认数组长度创建空顺序队列

public ArrayQueue() {

capacity = DEFAULT\_SIZE;

elementData = new Object[capacity];

}

//以一个初始化元素来创建顺序队列

public ArrayQueue(T element) {

this();

elementData[0] = element;

rear++;

}

public ArrayQueue(int initSize) {

elementData = new Object[initSize];

}

/\*\*

\* 以指定长度的数组来创建顺序队列

\* @param element 指定顺序队列中第一个元素

\* @param initSize 指定顺序队列底层数组的长度

\*/

public ArrayQueue(T element, int initSize) {

this.capacity = initSize;

elementData = new Object[capacity];

elementData[0] = element;

rear++;

}

/\*\*

\* @Title: size

\* @Description: 获取顺序队列的大小

\* @return

\*/

public int size() {

return rear - front;

}

/\*\*

\* @Title: offer

\* @Description: 入队

\* @param element

\*/

public void offer(T element) {

ensureCapacity(rear + 1);

elementData[rear++] = element;

}

private void ensureCapacity(int minCapacity) {

//如果数组的原有长度小于目前所需的长度

int oldCapacity = elementData.length;

if (minCapacity > oldCapacity) {

int newCapacity = (oldCapacity \* 3) / 2 + 1;

if (newCapacity < minCapacity)

newCapacity = minCapacity;

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

}

/\*\*

\* @Title: poll

\* @Description: 出队

\* @return

\*/

public T poll() {

if (isEmpty()) {

throw new IndexOutOfBoundsException("空队列异常");

}

//保留队列的front端的元素的值

T oldValue = (T) elementData[front];

//释放队列的front端的元素

elementData[front++] = null;

return oldValue;

}

/\*\*

\* @Title: peek

\* @Description: 返回队列顶元素，但不删除队列顶元素

\* @return

\*/

public T peek() {

if (isEmpty()) {

throw new IndexOutOfBoundsException("空队列异常");

}

return (T) elementData[front];

}

/\*\*

\* @Title: isEmpty

\* @Description: 判断顺序队列是否为空队列

\* @return

\*/

public boolean isEmpty() {

return rear == front;

}

/\*\*

\* @Title: clear

\* @Description: 清空顺序队列

\*/

public void clear() {

//将底层数组所有元素赋为null

Arrays.fill(elementData, null);

front = 0;

rear = 0;

}

public String toString() {

if (isEmpty()) {

return "[]";

} else {

StringBuilder sb = new StringBuilder("[");

for (int i = front; i < rear; i++) {

sb.append(elementData[i].toString() + ", ");

}

int len = sb.length();

return sb.delete(len - 2, len).append("]").toString();

}

}

}

```

java链式队列的实现

```

package lang;

import java.io.Serializable;

public class LinkQueue<T> implements Serializable{

private static final long serialVersionUID = -6726728595616312615L;

//定义一个内部类Node，Node实例代表链队列的节点。

private class Node {

private T data;//保存节点的数据

private Node next;//指向下个节点的引用

//无参数的构造器

public Node() {

}

//初始化全部属性的构造器

public Node(T data, Node next) {

this.data = data;

this.next = next;

}

}

private Node front;//保存该链队列的头节点

private Node rear;//保存该链队列的尾节点

private int size;//保存该链队列中已包含的节点数

/\*\*

\* <p>Title: LinkQueue </p>

\* <p>Description: 创建空链队列 </p>

\*/

public LinkQueue() {

//空链队列，front和rear都是null

front = null;

rear = null;

}

/\*\*

\* <p>Title: LinkQueue </p>

\* <p>Description: 以指定数据元素来创建链队列，该链队列只有一个元素</p>

\*/

public LinkQueue(T element) {

front = new Node(element, null);

//只有一个节点，front、rear都指向该节点

rear = front;

size++;

}

/\*\*

\* @Title: size

\* @Description: 获取顺序队列的大小

\* @return

\*/

public int size() {

return size;

}

/\*\*

\* @Title: offer

\* @Description: 入队

\* @param element

\*/

public void offer(T element) {

//如果该链队列还是空链队列

if (front == null) {

front = new Node(element, null);

rear = front;//只有一个节点，front、rear都指向该节点

} else {

Node newNode = new Node(element, null);//创建新节点

rear.next = newNode;//让尾节点的next指向新增的节点

rear = newNode;//以新节点作为新的尾节点

}

size++;

}

/\*\*

\* @Title: poll

\* @Description: 出队

\* @return

\*/

public T poll() {

Node oldFront = front;

front = front.next;

oldFront.next = null;

size--;

return oldFront.data;

}

/\*\*

\* @Title: peek

\* @Description: 返回队列顶元素，但不删除队列顶元素

\* @return

\*/

public T peek() {

return rear.data;

}

/\*\*

\* @Title: isEmpty

\* @Description: 判断顺序队列是否为空队列

\* @return

\*/

public boolean isEmpty() {

return size == 0;

}

/\*\*

\* @Title: clear

\* @Description: 清空顺序队列

\*/

public void clear() {

//将front、rear两个节点赋为null

front = null;

rear = null;

size = 0;

}

public String toString() {

//链队列为空链队列时

if (isEmpty()) {

return "[]";

} else {

StringBuilder sb = new StringBuilder("[");

for (Node current = front; current != null; current = current.next) {

sb.append(current.data.toString() + ", ");

}

int len = sb.length();

return sb.delete(len - 2, len).append("]").toString();

}

}

public static void main(String[] args) {

LinkQueue<String> queue = new LinkQueue<String>("aaaa");

//添加两个元素

queue.offer("bbbb");

queue.offer("cccc");

System.out.println(queue);

//删除一个元素后

queue.poll();

System.out.println("删除一个元素后的队列：" + queue);

//再次添加一个元素

queue.offer("dddd");

System.out.println("再次添加元素后的队列：" + queue);

//删除一个元素后，队列可以再多加一个元素

queue.poll();

//再次加入一个元素

queue.offer("eeee");

System.out.println(queue);

}

}

```

#循环队列

　　为充分利用向量空间，克服"假溢出"现象的方法是：将向量空间想象为一个首尾相接的圆环，并称这种向量为循环向量。存储在其中的队列称为循环队列（Circular Queue）。这种循环队列可以以单链表的方式来在实际编程应用中来实现。

##循环队列的队空与队满的条件

初始化建空队时， front=rear=0

当队空时：front=rear

当队满时：front=rear 亦成立

因此只凭等式front=rear无法判断队空还是队满。 有两种方法处理上述问题：

1、另设一个标志位以区别队列是空还是满。

2、少用一个元素空间，约定以“队列头指针front在队尾指针rear的下一个位置上”作为队列“满”状态的标志。即：

\*\*队空时： front=rear\*\*

\*\*队满时： (rear+1)%maxsize=front\*\*

front指向队首元素，rear指向队尾元素的下一个元素（始终指向空）。

　循环队列的Java实现:

```

package lang;

import java.io.Serializable;

import java.util.Arrays;

public class LoopQueue<T> implements Serializable{

private static final long serialVersionUID = -3670496550272478781L;

private int DEFAULT\_SIZE = 10;

private int capacity;//保存数组的长度

private Object[] elementData;//定义一个数组用于保存循环队列的元素

private int front = 0;//队头

private int rear = 0;//队尾

//以默认数组长度创建空循环队列

public LoopQueue() {

capacity = DEFAULT\_SIZE;

elementData = new Object[capacity];

}

//以一个初始化元素来创建循环队列

public LoopQueue(T element) {

this();

elementData[0] = element;

rear++;

}

/\*\*

\* 以指定长度的数组来创建循环队列

\* @param element 指定循环队列中第一个元素

\* @param initSize 指定循环队列底层数组的长度

\*/

public LoopQueue(T element, int initSize) {

this.capacity = initSize;

elementData = new Object[capacity];

elementData[0] = element;

rear++;

}

//获取循环队列的大小

public int size() {

if (isEmpty()) {

return 0;

}

return rear > front ? rear - front : capacity - (front - rear);

}

//插入队列

public void add(T element) {

if (rear == front && elementData[front] != null) {

throw new IndexOutOfBoundsException("队列已满的异常");

}

elementData[rear++] = element;

//如果rear已经到头，那就转头

rear = rear == capacity ? 0 : rear;

}

//移除队列

public T remove() {

if (isEmpty()) {

throw new IndexOutOfBoundsException("空队列异常");

}

//保留队列的rear端的元素的值

T oldValue = (T) elementData[front];

//释放队列的rear端的元素

elementData[front++] = null;

//如果front已经到头，那就转头

front = front == capacity ? 0 : front;

return oldValue;

}

//返回队列顶元素，但不删除队列顶元素

public T element() {

if (isEmpty()) {

throw new IndexOutOfBoundsException("空队列异常");

}

return (T) elementData[front];

}

//判断循环队列是否为空队列

public boolean isEmpty() {

//rear==front且rear处的元素为null

return rear == front && elementData[rear] == null;

}

//清空循环队列

public void clear() {

//将底层数组所有元素赋为null

Arrays.fill(elementData, null);

front = 0;

rear = 0;

}

public String toString() {

if (isEmpty()) {

return "[]";

} else {

//如果front < rear，有效元素就是front到rear之间的元素

if (front < rear) {

StringBuilder sb = new StringBuilder("[");

for (int i = front; i < rear; i++) {

sb.append(elementData[i].toString() + ", ");

}

int len = sb.length();

return sb.delete(len - 2, len).append("]").toString();

}

//如果front >= rear，有效元素为front->capacity之间、0->front之间的

else {

StringBuilder sb = new StringBuilder("[");

for (int i = front; i < capacity; i++) {

sb.append(elementData[i].toString() + ", ");

}

for (int i = 0; i < rear; i++) {

sb.append(elementData[i].toString() + ", ");

}

int len = sb.length();

return sb.delete(len - 2, len).append("]").toString();

}

}

}

public static void main(String[] args) {

LoopQueue<String> queue = new LoopQueue<String>("aaaa", 3);

//添加两个元素

queue.add("bbbb");

queue.add("cccc");

//此时队列已满

System.out.println(queue);

//删除一个元素后，队列可以再多加一个元素

queue.remove();

System.out.println("删除一个元素后的队列：" + queue);

//再次添加一个元素，此时队列又满

queue.add("dddd");

System.out.println(queue);

System.out.println("队列满时的长度：" + queue.size());

//删除一个元素后，队列可以再多加一个元素

queue.remove();

//再次加入一个元素，此时队列又满

queue.add("eeee");

System.out.println(queue);

}

}

```

#阻塞队列（BlockingQueue）

##几种主要的阻塞队列

　　自从Java 1.5之后，在java.util.concurrent包下提供了若干个阻塞队列，主要有以下几个：

　　ArrayBlockingQueue：基于数组实现的一个阻塞队列，在创建ArrayBlockingQueue对象时必须制定容量大小。并且可以指定公平性与非公平性，默认情况下为非公平的，即不保证等待时间最长的队列最优先能够访问队列。

　　LinkedBlockingQueue：基于链表实现的一个阻塞队列，在创建LinkedBlockingQueue对象时如果不指定容量大小，则默认大小为Integer.MAX\_VALUE。

　　PriorityBlockingQueue：以上2种队列都是先进先出队列，而PriorityBlockingQueue却不是，它会按照元素的优先级对元素进行排序，按照优先级顺序出队，每次出队的元素都是优先级最高的元素。注意，此阻塞队列为无界阻塞队列，即容量没有上限（通过源码就可以知道，它没有容器满的信号标志），前面2种都是有界队列。

　　　DelayQueue：基于PriorityQueue，一种延时阻塞队列，DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue也是一个无界队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

##阻塞队列方法对比

　1.　非阻塞队列中的几个主要方法：

　　add(E e):将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则会抛出异常；

　　remove()：移除队首元素，若移除成功，则返回true；如果移除失败（队列为空），则会抛出异常；

　　offer(E e)：将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则返回false；

　　poll()：移除并获取队首元素，若成功，则返回队首元素；否则返回null；

　　peek()：获取队首元素，若成功，则返回队首元素；否则返回null

　　对于非阻塞队列，一般情况下建议使用offer、poll和peek三个方法，不建议使用add和remove方法。因为使用offer、poll和peek三个方法可以通过返回值判断操作成功与否，而使用add和remove方法却不能达到这样的效果。注意，非阻塞队列中的方法都没有进行同步措施。

2.　阻塞队列中的几个主要方法：

　　阻塞队列包括了非阻塞队列中的大部分方法，上面列举的5个方法在阻塞队列中都存在，但是要注意这5个方法在阻塞队列中都进行了同步措施。除此之外，阻塞队列提供了另外4个非常有用的方法：

　　put(E e)

　　take()

　　offer(E e,long timeout, TimeUnit unit)

　　poll(long timeout, TimeUnit unit)

　　put方法用来向队尾存入元素，如果队列满，则等待；

　　take方法用来从队首取元素，如果队列为空，则等待；

　　offer方法用来向队尾存入元素，如果队列满，则等待一定的时间，当时间期限达到时，如果还没有插入成功，则返回false；否则返回true；

　　poll方法用来从队首取元素，如果队列空，则等待一定的时间，当时间期限达到时，如果取到，则返回null；否则返回取得的元素；

阻塞队列内部是通过Object.wait()、Object.notify()实现的，下面来看

使用阻塞队列实现的生产者-消费者的例子：

```

public class Test {

private static Integer count = 0;

final BlockingQueue<Integer> bq = new ArrayBlockingQueue<Integer>(5);//容量为5的阻塞队列

public static void main(String[] args) {

Test t = new Test();

new Thread(t.new Producer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Consumer()).start();

new Thread(t.new Producer()).start();

}

class Producer implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(1000);

} catch (Exception e) {

e.printStackTrace();

}

try {

bq.put(1);

count++;

System.out.println(Thread.currentThread().getName() + "produce:: " + count);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

class Consumer implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(1000);

} catch (InterruptedException e1) {

e1.printStackTrace();

}

try {

bq.take();

count--;

System.out.println(Thread.currentThread().getName()+ "consume:: " + count);

} catch (Exception e) {

// TODO: handle exception

e.printStackTrace();

}

}

}

}

}

```

<table>

<tr>

<td bgcolor=#f4f31a>

<font color=#00aaff size=5 face="微软雅黑">

二、栈

</font>

</td>

</tr>

</table>

##

> 栈（stack）又名堆栈，它是一种运算受限的线性表。其限制是仅允许在表的一端进行插入和删除运算。这一端被称为栈顶，相对地，把另一端称为栈底。向一个栈插入新元素又称作进栈、入栈或压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素；从一个栈删除元素又称作出栈或退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素。

栈的示意图：

![这里写图片描述](http://img.blog.csdn.net/20160510143210148)

#java中的Stack

　　Stack 类表示后进先出（LIFO）的对象堆栈。它通过五个操作对类 Vector 进行了扩展 ，允许将向量视为堆栈。它提供了通常的 push 和 pop 操作，以及取堆栈顶点的 peek 方法、测试堆栈是否为空的 empty 方法、在堆栈中查找项并确定到堆栈顶距离的 search 方法。首次创建堆栈时，它不包含项。

　　Deque 接口及其实现提供了 LIFO 堆栈操作的更完整和更一致的 set，应该优先使用此 set，而非此类。例如：

```

Deque<Integer> stack = new ArrayDeque<Integer>();

```

成员方法：

> E push(E item)

把项压入堆栈顶部。

>E pop()

移除堆栈顶部的对象，并作为此函数的值返回该对象。

>E peek()

查看堆栈顶部的对象，但不从堆栈中移除它。

>boolean empty()

测试堆栈是否为空。

>int search(Object o)

返回对象在堆栈中的位置，以 1 为基数。

　　Stack继承于Vector，Vector本身是一个可增长的对象数组。

Stack并不要求其中保存数据的唯一性，当Stack中有多个相同的item时，调用search方法，只返回与查找对象equal并且离栈顶最近的item与栈顶间距离。

　　以下是Stack源码分析：

##empty（）

　判断stack是否为空，就需要有一个变量来计算当前栈的长度，如果该变量为0，则表示该栈为空。

源码：

```

public boolean empty() {

return size() == 0;

}

```

size()方法在父类Vector中实现了，在Vector里面有一个变量elementCount来表示容器里元素的个数。如果为0，则表示容器空。

```

public synchronized int size() {

return elementCount;

}

```

##peek（）

返回栈顶端的元素，如果栈为空的话，则要抛出异常。

```

public synchronized E peek() {

int len = size();

if (len == 0)

throw new EmptyStackException();

return elementAt(len - 1);

}

```

elementAt方法也是在Vector里面实现的，实际上是用一个elementData的Object数组来存储元素的。

```

public synchronized E elementAt(int index) {

if (index >= elementCount) {

throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);

}

return elementData(index);

}

@SuppressWarnings("unchecked")

E elementData(int index) {

return (E) elementData[index];

}

```

##pop

　将栈顶的元素弹出来，如果栈里有元素，就取最顶端的那个，否则就要抛出异常。

```

public synchronized E pop() {

　 E obj;

　 int len = size();

　 obj = peek();

　 removeElementAt(len - 1);

　 return obj;

}

```

通过peek()取到顶端的元素之后，我们需要用removeElementAt()方法将最顶端的元素移除。 removeElementAt()方法定义在vector中。

```

public synchronized void removeElementAt(int index) {

modCount++;

if (index >= elementCount) {

throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);

}

else if (index < 0) {

throw new ArrayIndexOutOfBoundsException(index);

}

int j = elementCount - index - 1;

if (j > 0) {

System.arraycopy(elementData, index + 1, elementData, index, j);

}

elementCount--;

elementData[elementCount] = null; /\* to let gc do its work \*/

}

```

　　这里用待删除元素的后面元素依次覆盖前面一个元素。这样，就相当于将数组的实际元素长度给缩短了。

##push（）

　　将数据入栈

```

public E push(E item) {

addElement(item);

return item;

}

```

　将要入栈的元素放到数组的末尾，再将数组长度加1就可以了。addElement（）方法也在vector中（好父亲啊）。

```

public synchronized void addElement(E obj) {

modCount++;

ensureCapacityHelper(elementCount + 1);

elementData[elementCount++] = obj;

}

private void ensureCapacityHelper(int minCapacity) {

// overflow-conscious code

if (minCapacity - elementData.length > 0)

grow(minCapacity);

}

private void grow(int minCapacity) {

// overflow-conscious code

int oldCapacity = elementData.length;

int newCapacity = oldCapacity + ((capacityIncrement > 0) ?

capacityIncrement : oldCapacity);

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

elementData = Arrays.copyOf(elementData, newCapacity);

}

private static int hugeCapacity(int minCapacity) {

if (minCapacity < 0) // overflow

throw new OutOfMemoryError();

return (minCapacity > MAX\_ARRAY\_SIZE) ?

Integer.MAX\_VALUE :

MAX\_ARRAY\_SIZE;

}

```

##search（）

　找到一个最靠近栈顶端的匹配元素，然后返回这个元素到栈顶的距离

```

public synchronized int search(Object o) {

int i = lastIndexOf(o);

if (i >= 0) {

return size() - i;

}

return -1;

}

```

对应在vector里面的实现也相对容易理解：

```

public synchronized int lastIndexOf(Object o) {

return lastIndexOf(o, elementCount-1);

}

public synchronized int lastIndexOf(Object o, int index) {

if (index >= elementCount)

throw new IndexOutOfBoundsException(index + " >= "+ elementCount);

if (o == null) {

for (int i = index; i >= 0; i--)

if (elementData[i]==null)

return i;

} else {

for (int i = index; i >= 0; i--)

if (o.equals(elementData[i]))

return i;

}

return -1;

}

```

　　lastIndexOf是从数组的末端往前遍历，如果找到这个对象就返回。如果到头了，还未找到就返回个-1。

##栈和队列的区别

　　队列是FIFO的（先进先出），堆栈是FILO的（现今后出）

　　栈是限定只能在表的一端进行插入和删除操作的线性表。 队列是限定只能在表的一端进行插入和在另一端进行删除操作的线性表

　　栈只能从头部取数据，也就最先放入的需要遍历整个栈最后才能取出来，而且在遍历数据的时候还得为数据开辟临时空间；

队列基于地址指针进行遍历，而且可以从头或尾部开始遍历，但不能同时遍历，无需开辟临时空间，因为在遍历的过程中不影像数据结构，速度要快的多。

# 第四章 算法

## 第1节 二叉搜索树与双向链表

## 题目描述

> 输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

##输入描述

> 二叉搜索树

##输出描述

> 排序的双向链表

##题目分析

节点描述：

```

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

```

　　二叉查找树（英语：Binary Search Tree），也称二叉搜索树、有序二叉树（英语：ordered binary tree），排序二叉树（英语：sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

- 任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

- 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

- 任意节点的左、右子树也分别为二叉查找树；

- 没有键值相等的节点。

> 解法一（递归）　运行时间：32ms　占用内存：550k

```

//递归调用 左 根 右 遍历

public class Solution {

//双向链表的左边头结点和右边头节点

TreeNode leftHead = null;

TreeNode rightHead = null;

public TreeNode Convert(TreeNode pRootOfTree) {

//递归调用叶子节点的左右节点返回null

if(pRootOfTree==null) return null;

//第一次运行时，它会使最左边叶子节点为链表第一个节点

Convert(pRootOfTree.left);

if(rightHead==null){

leftHead= rightHead = pRootOfTree;

}else{

//把根节点插入到双向链表右边，rightHead向后移动

rightHead.right = pRootOfTree;

pRootOfTree.left = rightHead;

rightHead = pRootOfTree;

}

//把右叶子节点也插入到双向链表（rightHead已确定，直接插入）

Convert(pRootOfTree.right);

//返回左边头结点

return leftHead;

}

}

```

　　找到树最左边的叶子节点，以该节点为双向链表的第一个节点，然后根据　左　根　右　遍历依次把节点插入到双向链表右边并移动rightHead指针 ，最后返回leftHead。

> 解法二　运行时间：31ms　占用内存：503k

```

import java.util.Stack;

public class Solution {

public TreeNode Convert(TreeNode pRootOfTree) {

if(pRootOfTree == null) return pRootOfTree;

//双向链表的左边头结点和右边头节点

TreeNode leftHead = null;

TreeNode rightHead = null;

Stack<TreeNode> s = new Stack<TreeNode>();

while(pRootOfTree != null || !s.isEmpty()){

if(pRootOfTree != null) {

//依次遍历到最左边叶节点

s.push(pRootOfTree);

pRootOfTree = pRootOfTree.left;

} else {

pRootOfTree = s.pop();

if(rightHead == null)

//以最左边叶子节点为链表头

leftHead = rightHead = pRootOfTree;

else {

rightHead.right = pRootOfTree;

pRootOfTree.left = rightHead;

rightHead = pRootOfTree;

}

pRootOfTree = pRootOfTree.right;

}

}

return leftHead;

}

}

```

　　思路一样，以最左边叶子节点为链表头，借用堆栈遍历链表。

## 第2节 二叉树中 和为某值 的所有路径

## 题目描述

> 输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径

##输入描述

> 一颗二叉树和一个整数

##输出描述

> 二叉树中结点值的和为输入整数的所有路径

##题目分析

节点描述：

```

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

```

> 解法 　运行时间：31ms　占用内存：652k

```

public class Solution {

//用于存储路径集合

ArrayList<ArrayList<Integer>> paths=new ArrayList<ArrayList<Integer>>();

public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {

if(root==null) return paths;

//递归调用

SearchPath(new ArrayList<Integer>(),root,target);

return paths;

}

public void SearchPath(ArrayList<Integer> path,TreeNode root,int target){

if(root==null) return;

//把当前节点加入路径，同时target要减去当前节点的值

//这里path和target都是值传递的局部变量

path.add(root.val);

target=target-root.val;

if(root.left==null&&root.right==null){

//target为零的叶子节点对应的路径就是所求路径，把它加入到路径集合中

if(target==0){

paths.add(path);

}

return;

}

//不是叶子节点继续向下遍历，拷贝一个路径

ArrayList<Integer> path2=new ArrayList<>();

path2.addAll(path);

//递归左右节点

SearchPath(path,root.left,target);

SearchPath(path2,root.right,target);

}

}

```

　　递归遍历整棵树，每到一个节点时，把节点加入路径并target减去当前节点的值。如果为叶子节点，判断target是否为零，为零就是一条成功的路径。不是叶子节点就继续向左右子节点遍历。

## 第3节 二叉树的镜像

## 题目描述

> 操作给定的二叉树，将其变换为源二叉树的镜像。

##输入描述

二叉树的镜像定义：

源二叉树

8

/ \

6 10

/ \ / \

5 7 9 11

镜像二叉树

8

/ \

10 6

/ \ / \

11 9 7 5

##输出描述

> 源二叉树的镜像

##题目分析

节点描述：

```

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

```

> 解法　　运行时间：38ms 　占用内存：688k

```

public class Solution {

public void Mirror(TreeNode root) {

if(root==null) return;

//交换左右子节点

TreeNode tempNode;

tempNode = root.left;

root.left = root.right;

root.right = tempNode;

Mirror(root.left);

Mirror(root.right);

}

}

```

　　思路：从根节点开始依次递归遍历整棵树，如果节点不为空，就交换它的左右子节点（对象的赋值，相应的左右指针也会改变）。

## 第4节 二维数组中的查找

# 题目描述

>　　在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

##输入描述

> array： 待查找的二维数组

target：查找的数字

## 输出描述

> 查找到返回true，查找不到返回false

## 题目分析

![这里写图片描述](http://img.blog.csdn.net/20160423182044824)

　　根据题意画了个简单的图，首先我们要确定查找开始的位置，因为它是二维的数组，它的下标变化是两个方向的，根据四个边界点来分析。

> Ａ：向下　增　，向右　增

>Ｂ：向左　减　，向下　增

>Ｃ：向上　减　，向右　增

>Ｄ：向左　减　，向上　减

　　可以看出从B 或Ｃ点开始查找刚好可以构建一个二叉查找树。

> 　　二叉查找树（Binary Search Tree），（又：二叉搜索树，二叉排序树）它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。

　　先确定二维数组的行数和列数，把 查找值 与 二叉查找树的根节点（Ｂ或者　Ｃ）开始比较，如果相等返回true，小于查找左子树，大于就查找右子树。如果遍历超过数组边界，就返回 false。

> 以C为根节点查找

```

public class Solution {

public boolean Find(int [][] array,int target) {

int i = array.length -1;

int m = array[0].length -1;

int j = 0;

while(i>=0 && j<=m){

if(target == array[i][j]){

return true;

}else if(target >array[i][j]){

j++;

}else{

i--;

}

}

return false;

}

}

```

> 以B为根节点查找

```

public class Solution {

public boolean Find(int [][] array,int target) {

int i = array.length -1;

int j = array[0].length -1;

int n = 0;

while(j>=0 && n<=i){

if(target == array[n][j]){

return true;

}else if(target >array[n][j]){

n++;

}else{

j--;

}

}

return false;

}

}

```

## 第5节 二进制中1的个数

## 题目描述

> 输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。

##输入描述

> 一个整数

##输出描述

> 该数二进制表示中1的个数

##题目分析

> 解法一　　运行时间：29m 　占用内存：629k

```

public int NumberOf1(int n) {

String s=Integer.toBinaryString(n);

char[] c=s.toCharArray();

int j=0;

for(int i=0;i<c.length;i++){

if(c[i]=='1'){

j++;

}

}

return j;

}

}

```

解析：

> public static String toBinaryString(int i)

>

以二进制（基数 2）无符号整数形式返回一个整数参数的字符串表示形式。

①先把整数转换成二进制字符串

②把字符串转换成字符数组

③遍历该数组，判断每位是否为1，为1 计数加1。

④遍历完成返回1的个数

> 解法二 运行时间：30ms 占用内存：629k

```

public class Solution {

public int NumberOf1(int n) {

int count =0;

while(n!=0){

count++;

n = n&(n-1);

}

return count;

}

}

```

解析：

> 如果一个整数不为0，那么这个整数至少有一位是1。如果我们把这个整数减1，那么原来处在整数最右边的1就会变为0，原来在1后面的所有的0都会变成1(如果最右边的1后面还有0的话)。其余所有位将不会受到影响。

举个例子：

①二进制数　　1100

② 减１后，得　1011

③1100&1011=1000

对比①和③你会发现，\*\*把一个整数减去1，再和原整数做与运算，会把该整数最右边一个1变成0.那么一个整数的二进制有多少个1，就可以进行多少次这样的操作。\*\*

## 第6节 从上往下打印二叉树

## 题目描述

> 从上往下打印出二叉树的每个节点，同层节点从左至右打印。

##输入描述

> 二叉树的根节点

##输出描述

> 从上到下，从左到右打印出整棵二叉树

##题目分析

节点描述

```

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

```

> 解法一 　运行时间：28ms　占用内存：636k

```

import java.util.\*;

public class Solution {

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {

ArrayList<Integer> list =new ArrayList<Integer>();

if(root==null) return list;

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

while(!queue.isEmpty()){

TreeNode t = queue.poll();

list.add(t.val);

if(t.left!=null) queue.add(t.left);

if(t.right!=null) queue.add(t.right);

}

return list;

}

}

```

　　其实就是一个宽度优先搜索，新建一个list用于顺序存放值，一个queue用来存放节点。每次出队一个节点把节点值存入list中，同时把它的左右子节点依次入队。这样形成一个从上到下从左到右的层次遍历，也就是宽度优先搜索。

> 解法二 　运行时间：32ms　占用内存：503k

```

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {

ArrayList<Integer> listVal =new ArrayList<Integer>();

if(root==null) return listVal;

ArrayList<TreeNode> listNode =new ArrayList<TreeNode>();

listNode.add(root);

for(int i=0;i<listNode.size();i++){

TreeNode t = listNode.get(i);

listVal.add(t.val);

if(t.left!=null) listNode.add(t.left);

if(t.right!=null) listNode.add(t.right);

}

return listVal;

}

}

```

　这个和第一种解法的思路是一样的，只是队列换成链表来实现了。

## 第7节 从尾到头打印链表

## 题目描述

> 输入一个链表，从尾到头打印链表每个节点的值。

##输入描述

>输入为链表的表头

##输出描述

> 输出为需要打印的“新链表”的表头

##题目分析

链表节点类

```

/\*\*

\* public class ListNode {

\* int val;

\* ListNode next = null;

\*

\* ListNode(int val) {

\* this.val = val;

\* }

\* }

\*

\*/

```

> 解法一

　　根据Stack的特点 后进先出（LIFO），可以先遍历链表，把值存入堆栈中。然后新建一个链表对象做为返回，再把堆栈中的值依次拿出来存入新链表中, 就实现了倒序。

> 运行时间：29ms　　　占用内存：503k

```

import java.util.\*;

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

Stack<Integer> st = new Stack<Integer>();

while(listNode!=null){

st.push(listNode.val);

listNode = listNode.next;

}

ArrayList<Integer> list = new ArrayList<Integer>();

while(!st.empty()){

list.add(st.pop());

}

return list;

}

}

```

　　注意Stack 申明时要确定存入数据的具体类型（Integer），否则　pop（）函数返回的是Object对象，需要　强转。

> 解法二

　　直接新建一个ArrayList对象 list，遍历链表把值存入list中。再调用Collections工具类的反转函数把list反转，最后返回。

> 运行时间：35ms　　占用内存：654k

```

import java.util.ArrayList;

import java.util.Collections;

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

ArrayList<Integer> list= new ArrayList<Integer>();

if(listNode!=null)

{

list.add(listNode.val);

ListNode nextnode=listNode.next;

while(nextnode!=null){

list.add(nextnode.val);

nextnode=nextnode.next;

}

Collections.reverse(list);

}

return list;

}

}

```

这是　Collections.reverse（）函数源码，性能比第一种差一点。

```

374 public static void More ...reverse(List<?> list) {

375 int size = list.size();

376 if (size < REVERSE\_THRESHOLD || list instanceof RandomAccess) {

377 for (int i=0, mid=size>>1, j=size-1; i<mid; i++, j--)

378 swap(list, i, j);

379 } else {

380 // instead of using a raw type here, it's possible to capture

381 // the wildcard but it will require a call to a supplementary

382 // private method

383 ListIterator fwd = list.listIterator();

384 ListIterator rev = list.listIterator(size);

385 for (int i=0, mid=list.size()>>1; i<mid; i++) {

386 Object tmp = fwd.next();

387 fwd.set(rev.previous());

388 rev.set(tmp);

389 }

390 }

391 }

```

> 解法三

　　新建两个ArrayList对象 list1和list2，遍历链表把值存入list1。再把list1从后到前 遍历，把值放入list2中（实现反转），返回list2；

> 运行时间：28ms 占用内存：629k

```

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

ArrayList<Integer> list1 = new ArrayList<Integer>();

ArrayList<Integer> list2 = new ArrayList<Integer>();

while(listNode!=null){

list1.add(listNode.val);

listNode=listNode.next;

}

for(int i=list1.size()-1;i>=0;i--){

list2.add(list1.get(i));

}

return list2;

}

}

```

## 第8节 判断二叉搜索树的后序遍历序列

## 题目描述

> 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。

##输入描述

> 整数数组

##输出描述

> 布尔值

##题目分析

什么是二叉搜索树？

> 二叉查找树（Binary Search Tree），（又：二叉搜索树，二叉排序树）它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。

分析：

BST的后序遍历的合法序列满足：

对于一个序列S，最后一个元素是根元素，假设为x；

如果去掉根元素剩下的序列为T，那么T满足：前一段（左子树）小于x，后一段（右子树）大于x，且这两段（子树）也是合法的后序序列。

> 解法一 　运行时间：32ms　占用内存：503k

```

public class Solution {

public boolean VerifySquenceOfBST(int [] sequence) {

if(sequence==null || sequence.length==0) return false;

return IsTreeOfBST(sequence,0,sequence.length-1);

}

public boolean IsTreeOfBST(int [] sequence,int start,int end){　　 //子数组为空

if(end<=start) return true;

int i=0;

for(;i<end;i++){//找到左右子树分割点

if(sequence[i]>sequence[end]) break;

}

for(int j=i;j<end;j++){

if(sequence[j]<sequence[end]) return false;

}

//递归

return IsTreeOfBST(sequence,0,i-1)&&IsTreeOfBST(sequence,i,end-1);

}

}

```

　　采用递归的思想，先找出根节点，左子树元素都必须比根节点小，右子树节点都比根节点大,否则返回false.

得到子树（子序列）的两种方法：

①用下标把数组　逻辑分为几个子数组（这里采用的是这种）

②用工具类Arrays把数组分割

> 解法二 　运行时间：38ms　占用内存：654k

```

import java.util.Arrays;

public class Solution {

public boolean VerifySquenceOfBST(int [] sequence) {

int len = sequence.length;

if(sequence==null || len==0) return false;

int i=0;

for(;i<len-1;i++){//寻找左右子树分界点

if(sequence[i]>sequence[len-1]) break;

}

for(int j=i;j<len-1;j++){

if(sequence[j]<sequence[len-1]) return false;

}

boolean left=true;//左右子树标志，默认为true

boolean right=true;

if(i>0){//判断是否还有左子树

left = VerifySquenceOfBST(Arrays.copyOfRange(sequence,0,i));

}

if(i<len-1){//判断是否还有右子树

right = VerifySquenceOfBST(Arrays.copyOfRange(sequence,i,len-1));

}

return left&&right;//返回递归结果

}

}

```

　　思路还是和解法一相同，只是用工具类Arrays把数组分割，效率比解法一慢一些。

## 第9节 判断栈的弹出序列

## 题目描述

> 输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4，5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。

##输入描述

> 压入序列

##输出描述

> 判断是否为出栈序列的boolean值

##题目分析

> 解法一　　运行时间：26ms 　占用内存：629k

```

public class Solution {

public boolean IsPopOrder(int [] pushA,int [] popA) {

if(pushA.length==0 || popA.length==0){

return false;

}

Stack<Integer> stack = new Stack<Integer>();

int j=0;

for(int i=0;i<pushA.length;i++){

stack.push(pushA[i]);

while(j<popA.length && stack.peek()==popA[j]){

j++;

stack.pop();

}

}

return stack.isEmpty();

}

}

```

①判断数组长度，如果为０，返回false

②栈的特性：每次入栈，只要栈中还有元素，就可以多次出栈。

③如果最后栈为空，则这个序列就是弹出序列，否则不是

## 第10节 判断树B是不是树A的子结构

## 题目描述

> 输入两颗二叉树A，B，判断B是不是A的子结构。

##输入描述

> 两颗二叉树A，B

##输出描述

> boolean值

##题目分析

树节点声明：

```

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

```

> 解法 　运行时间：30ms 　占用内存：629k

```

public class Solution {

　　　//判断root1为根节点的树 是否包含 root2为根的树

public boolean HasSubtree(TreeNode root1,TreeNode root2) {

　　//如果其中有一个节点为空就返回false

if (root1 == null || root2 == null) return false;

// 如果root2为根节点的树 是　root1为根的树的子树，返回true;否则依次遍历左右子节点再做判断

　　　return　isSubtree(root1,root2)||

　　　　　　　　　HasSubtree(root1.left,root2)||

　　　　　　　　　　HasSubtree(root1.right,root2);

}

//判断root2为根节点的树 是否为 root1为根的树的子树

public boolean isSubtree(TreeNode root1, TreeNode root2) {

if (root2 == null) return true;

if (root1 == null) return false;

if (root2.val == root1.val) {

//如果根节点相同分别遍历左右子树，看是否相同

　　　　　　　 return isSubtree(root1.left, root2.left)&& 　 　　　　　　　　　　　　isSubtree(root1.right, root2.right);

}else {

return false;

}

}

}

```

首先需要递归root1树，找到与root2根一样的节点；

找到相同的根节点后，要判断是否子树（及左右节点都相等），都相等返回true，否则返回false;

## 第11节 包含min函数的栈

## 题目描述

> 定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的min函数。

##输入描述

> 栈结构

##输出描述

> min函数

##题目分析

> 解法一 　　运行时间：33ms　占用内存：528k

```

import java.util.Stack;

public class Solution {

Stack<Integer> stack = new Stack<Integer>();

Stack<Integer> min = new Stack<Integer>();

public void push(int node) {

stack.push(node);

if(min.isEmpty()){

min.push(node);

}else{

min.push(min.peek() > node ? node : min.peek());

}

}

public void pop() {

stack.pop();

min.pop();

}

public int top() {

return stack.peek();

}

public int min() {

return min.peek();

}

}

```

　　思路：实现两个栈，stack用来存储数据，min用来保存最小值；为了使min栈顶元素一直是当前stack栈中的最小值，要在stack入栈时进行判断，如果入栈值比当前min栈顶值更小，则把该值入为min新的栈顶，否则再\*\*重复入一次min栈顶元素\*\*（为了stack和min同时出栈，虽然出栈值可能不一样，但并不影响min栈顶是当前stack中的最小元素）。

　　在min函数中min.peek()就行了，返回栈顶但不出栈。

> 解法二 　运行时间：33ms 　占用内存：654k

```

import java.util.Stack;

import java.util.Iterator;

public class Solution {

Stack<Integer> stack = new Stack<Integer>();

public void push(int node) {

stack.push(node);

}

public void pop() {

stack.pop();

}

public int top() {

return stack.peek();

}

public int min() {

Iterator<Integer> iterator = stack.iterator();

int min = stack.peek();

int temp = 0;

while(iterator.hasNext()){

temp = iterator.next();

if(temp<min){

min = temp;

}

}

return min;

}

}

```

　　因为stack实现了Collection接口，我们用迭代器遍历所有元素，求出最小值返回。

> 解法三　运行时间：28ms　占用内存：629k

```

public class Solution {

Stack<Integer> s = new Stack<Integer>();

ArrayList<Integer> a = new ArrayList<Integer>();

public void push(int node) {

a.add(s.push(node));

}

public void pop() {

a.remove(s.pop());

}

public int top() {

return s.peek();

}

public int min() {

Object[] b = a.toArray();

Arrays.sort(b);

return (int)b[0];

}

}

```

　　把每次入栈元素都存在一个list中，求最小值时把list转换成数组，用Arrays工具类排序（从小到大），然后返回数组第一个元素就是最小值。

## 第12节 反转链表

## 题目描述

> 输入一个链表，反转链表后，输出链表的所有元素。

##输入描述

> 链表

##输出描述

> 反转链表

##题目分析

　　节点申明：

```

public class ListNode {

int val;

ListNode next = null;

ListNode(int val) {

this.val = val;

}

}

```

这里的头结点也有val值，它不是一个空节点。

> 解法一（递归） 运行时间：31ms 　 占用内存：688k

```

public class Solution {

public ListNode ReverseList(ListNode head) {

if(head==null || head.next==null){

return head;

}

ListNode p=head.next;

ListNode newHead=ReverseList(p);

p.next = head;

head.next =null;

return newHead;

}

}

```

　　把整个链表看成只有两个节点（\*\*头结点\*\*和\*\*剩下的节点\*\*），反转只用　剩下的节点　的尾指针指向头结点，头结点的指针指向空。再把　剩下的节点　看成两个节点，依次递归即可。

> 解法二　运行时间：32ms　占用内存：629k

```

import java.util.\*;

public class Solution {

public ListNode ReverseList(ListNode head) {

if(head==null || head.next==null){

return head;

}

Stack stack = new Stack();

ListNode in = head;

ListNode out = head;

while(in!=null){

stack.push(in.val);

in = in.next;

}

while(out!=null){

out.val = (int)stack.pop();

out = out.next;

}

return head;

}

}

```

　　链表的结构和指针不变，把整个链表的值（val）反转存入即可（利用Stack存值）。

> 解法三 　运行时间：29ms　占用内存：629k

```

import java.util.\*;

public class Solution {

public ListNode ReverseList(ListNode head) {

if(head==null || head.next==null){

return head;

}

ListNode cur=head;

ListNode pre=null;

ListNode next=null;

while(cur!=null)

{

next=cur.next;

cur.next=pre;

pre=cur;

cur=next;

}

return pre;

}

}

```

　　链表的值不变，利用中转节点pre来改变指针，使最后一个节点依次指向第一个节点。

## 第13节 变态跳台阶

## 题目描述

> 一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法

##输入描述

> 台阶数

##输出描述

> 跳法数

##题目分析

- 设n阶的跳数为f(n)

- 当n=1时，f(1) = 1

- 当n=2时，分为最后一步 跳2阶和跳1阶 两种情况，有f(2)=f(0)+f(1)=1+1=2

- 当n=3时，分为最后一步 跳3阶、跳2阶和跳1阶 三种情况，有f(3)=f(0)+f(1)+f(2)=1+1+2=4

- 有 　　 f(n) = f(n-1)+f(n-2)+...+f(1) + f(0)成立

同时有　f(n-1)＝f(n-2)+...+f(1) + f(0)　成立，可得出f(n)=2f(n-1) (n>=2)

![这里写图片描述](http://img.blog.csdn.net/20160427145606607)

很明显可以得出递推公式：

　　　　 | 1 　　　　(n=0 )

f(n) = 　 | 1 　　　　 (n=1 )

　　　　 | 2\*f(n-1)　　(n>=2)

> 解法一 　运行时间：35ms 　 占用内存：654k

```

public class Solution {

public int JumpFloorII(int target) {

if(target<=1) return 1;

return 2\*JumpFloorII(target-1);

}

}

```

> 解法二 　运行时间：34ms　 占用内存：654k

```

public class Solution {

public int JumpFloorII(int target) {

　　 if(target<=1) return 1;

return 1<<(target-1);

}

}

```

　　换一种思路想一下：一共有ｎ个台阶，最后一个台阶是一定要跳上去的，其他的　ｎ－１个可跳可不跳，一共有多少总情况？

> ２（ｎ－１）

这里用移位实现乘法，时间上要快一些！

## 第14节 合并两个排序链表

## 题目描述

> 输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

##输入描述

>两个单调递增的链表

##输出描述

> 一个单调不减的链表

##题目分析

节点申明：

```

public class ListNode {

int val;

ListNode next = null;

ListNode(int val) {

this.val = val;

}

}

```

> 解法一 （递归）　运行时间：31ms　 占用内存：629k

```

public class Solution {

public ListNode Merge(ListNode list1,ListNode list2) {

if(list1==null) return list2;

if(list2==null) return list1;

ListNode mergeHead=null;

if(list1.val<=list2.val){

mergeHead = list1;

mergeHead.next = Merge(list1.next,list2);

}else{

mergeHead = list2;

mergeHead.next = Merge(list1,list2.next);

}

return mergeHead;

}

}

```

　　首先，判断链表是否为空，如果其中一个为空就直接返回另一个链表。

　　因为合并后的链表要满足单调不减，比较两个链表当前头结点，值较小的赋值给合并链表的当前节点，依次递归下一个节点。

　　依次返回合并链表的当前节点。

> 解法二　　运行时间：31ms 　占用内存：629k

```

public class Solution {

public ListNode Merge(ListNode list1,ListNode list2) {

if(list1==null) return list2;

if(list2==null) return list1;

ListNode mList=new ListNode(0);

ListNode point =mList;

while(list1!=null && list2!=null){

if(list1.val<=list2.val){

point.next=list1;

list1=list1.next;

}else{

point.next = list2;

list2=list2.next;

}

point = point.next;

}

if(list1==null){

point.next=list2;

}else{

point.next=list1;

}

return mList.next;

}

}

```

①首先，判断链表是否为空，如果其中一个为空就直接返回另一个链表。

②申明合并链表的头结点和指针节点

③循环比较两个链表的头结点的值，让当前的point节点指向值较小的那个节点，该节点等于其下一个节点，同时point向后移。

④判别循环后，如果有链表没遍历完，直接加在point后面

⑤返回头结点的下一个节点（头结点是新建的节点）

> 这个思想很简单：就是新建一个头结点，依次比较两个链表的节点的值，用next指针把他们依次串联起来。

## 第15节 复杂链表的复制

## 题目描述

> 输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点）。

##输入描述

> 一个复杂链表头结点

##输出描述

> 该链表的复制链表头结点

##题目分析

　　节点描述：

```

public class RandomListNode {

int label;

RandomListNode next = null;

RandomListNode random = null;

RandomListNode(int label) {

this.label = label;

}

}

```

> 解法一 运行时间：36ms　占用内存：566k

```

public class Solution {

public RandomListNode Clone(RandomListNode pHead)

{

if(pHead==null) return pHead;

RandomListNode copyHead = new RandomListNode(pHead.label);

copyHead.random = pHead.random;

copyHead.next = Clone(pHead.next);

return copyHead;

}

}

```

　　用递归的方式复制每一个节点。

> 解法二 　运行时间：31ms　占用内存：692k

```

public class Solution {

public RandomListNode Clone(RandomListNode pHead)

{

if(pHead==null) return pHead;

//拷贝新链表头节点

RandomListNode copyHead = CopyNode(pHead);

//新链表指针节点

RandomListNode copyPoint = copyHead;

while(pHead.next!=null){

//把拷贝的节点加到新链表

copyPoint.next = CopyNode(pHead.next);

//两个链表指针向后移动

copyPoint = copyPoint.next;

pHead = pHead.next;

}

return copyHead;

}

//拷贝节点函数

public RandomListNode CopyNode(RandomListNode node){

RandomListNode temp = new RandomListNode(node.label);

temp.next = node.next;

temp.random = node.random;

return temp;

}

}

```

　循环遍历链表把每个节点拷贝到新链表中。

## 第16节 字符串中空格替换

## 题目描述

> 请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

##输入描述

> str 是StringBuffer的对象

##输出描述

> 返回替换后的字符串的String对象

##题目分析

　　\*\*首先要对输入参数进行判空和越界判断\*\*（任何算法题都要注意）。

> 解法一 （运行时间：36ms 占用内存：688k）

```

public class Solution {

public String replaceSpace(StringBuffer str) {

if(str==null){

return null;

}

StringBuffer sb = new StringBuffer();

char [] strChar = str.toString().toCharArray();

for(int i=0;i<strChar.length;i++){

if(strChar[i]==' '){

sb.append("%20");

}else{

sb.append(strChar[i]);

}

}

return sb.toString();

}

}

```

　　把传入的StringBuffer 对象 str 转换成字符串 再转换成\*\*字符数组\*\*,申请一个新的StringBuffer 对象 sb来存最后的结果，遍历整个字符数组，如果当前字符为空格则把　＂％２０＂加入到sb中，否则直接追加当前字符，最后把sb转换成字符串返回。

> 解法二 （运行时间：34ms 占用内存：503k）

```

public class Solution {

public String replaceSpace(StringBuffer str) {

if(str == null){

return null;

}

for(int i =0;i<str.length();i++){

char c = str.charAt(i);

if(c==' '){

str.replace(i,i+1,"%20");

}

}

return str.toString();

}

}

```

> public StringBuffer replace(int start, int end, String str)

>

　　使用给定 String 中的字符替换此序列的子字符串中的字符。该子字符串从指定的 start 处开始，一直到索引 end - 1 处的字符，如果不存在这种字符，则一直到序列尾部。\*\*先将子字符串中的字符移除，然后将指定的 String 插入 start。\*\*（如果需要，序列将延长以适应指定的字符串。）

参数：

start - 起始索引（包含）。

end - 结束索引（\*\*不包含\*\*）。

str - 将替换原有内容的字符串。

返回：

此对象。

源码（可以看出它调用的是父类的方法）

```

public synchronized StringBuffer More ...replace(int start, int end, String str) {

453 toStringCache = null;

454 super.replace(start, end, str);

455 return this;

456 }

```

父类AbstractStringBuilder

```

822 public AbstractStringBuilder More ...replace(int start, int end, String str) {

823 if (start < 0)

824 throw new StringIndexOutOfBoundsException(start);

825 if (start > count)

826 throw new StringIndexOutOfBoundsException("start > length()");

827 if (start > end)

828 throw new StringIndexOutOfBoundsException("start > end");

829

830 if (end > count)

831 end = count;

832 int len = str.length();

833 int newCount = count + len - (end - start);

834 ensureCapacityInternal(newCount);

835

836 System.arraycopy(value, end, value, start + len, count - end);

837 str.getChars(value, start);

838 count = newCount;

839 return this;

840 }

```

　　可以看出StringBuffer类 提供了 插入的函数，值得注意的是 StringBuilder 也是继承于AbstractStringBuilder这个抽象类的。所以它也有这个方法，但是String并没有。

> 解法三（不推荐，运行时间：38ms 占用内存：654k）

```

public class Solution {

public String replaceSpace(StringBuffer str) {

return str.toString().replaceAll("\\s", "%20");

}

}

```

## 第17节 字符串的顺序全排列

## 题目描述

> 输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串abc,则打印出由字符a,b,c所能排列出来的所有字符串abc,acb,bac,bca,cab和cba。 结果请按字母顺序输出。

##输入描述

> 输入一个字符串,长度不超过9(可能有字符重复),字符只包括大小写字母。

##输出描述

> 顺序输出字符串的所有排列

##题目分析

　　这是一个字符串全排列的问题，把全部序列存在TreeSet中默认可得到字典顺序。

> TreeSet

　　　　基于TreeMap实现的SortedSet，可以对Set集合中的元素进行排序，排序后按升序排列元素（缺省是按照自然排序），非线程安全。

思路：

　　固定一个字符串之后，之后再将问题变小，只需求出后面子串的排列个数就可以得出结果，然后依次将后面的字符串与前面的交换，再递归子串的排列结果，最后当所有字符都固定结束递归。

下面这张图很清楚的给出了递归的过程：

![这里写图片描述](http://img.blog.csdn.net/20160523115821386)

> 解法　运行时间：131ms　占用内存：1477k

```

import java.util.\*;

public class Solution {

//用于最后返回结果

ArrayList<String> list = new ArrayList<>();

//遍历的时候来存储序列实现排序

TreeSet<String> set = new TreeSet<>();

public ArrayList<String> Permutation(String str) {

if(str==null || str.length()==0) return list;

Permutation(str.toCharArray(),0);

//容器转换

list.addAll(set);

return list;

}

public void Permutation(char[] s,int index){

if(s==null ||s.length==0 || index<0 || index>s.length-1) return ;

if(index==s.length-1){//递归固定到最后一个位置，把该串加入集合

set.add(new String(s));

}else{//固定前index+1个字符，递归后面所有可能的子串

for(int i = index;i<s.length;i++){

swap(s,index,i);//交换一次形成一个子串

Permutation(s,index+1);

swap(s,i,index);//复原使下次循环产生下一个子串

}

}

}

public void swap(char[] s,int i,int j){

char temp = s[i];

s[i] = s[j];

s[j] = temp;

}

}

```

## 第18节 数组中出现次数超过一半的数字

## 题目描述

> 数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

##输入描述

> 数字数组

##输出描述

> 出现的次数超过数组长度的一半的数字

##题目分析

> 解法一（以暴制暴） 运行时间：32ms　占用内存：528k

```

public class Solution {

public int MoreThanHalfNum\_Solution(int [] array) {

if(array.length==0 || array==null) return 0;

int temp;

for(int i=0;i<array.length;i++){

temp = array[i];

int count=0;

for(int j =0;j<array.length;j++){

if(array[j]==temp){

count++;

}

}

if(count>(array.length/2)){

return temp;

}

}

return 0;

}

}

```

　　时间复杂度O(n^2), 遍历数组中每一个数的出现次数，第一个次数大于长度一半 的返回。

> 解法二 　运行时间：31ms　占用内存：629k

```

public class Solution {

public int MoreThanHalfNum\_Solution(int [] array) {

if(array.length==0 || array==null) return 0;

int temp=array[0],count=1;

//找出出现次数最多的数

for(int i=1;i<array.length;i++){

if(array[i]==temp){

count++;

}else{

count--;

}

if(count==0){

temp=array[i];

count=1;

}

}

//由于可能中间夹着其他数，所以count不准确，重新确认count

count=0;

for(int i=0;i<array.length;i++){

if(array[i]==temp){

count++;

}

}

if(count\*2>array.length){

return temp;

}

return 0;

}

}

```

　　时间复杂度Ｏ(n) （ps：虽然时间上美什么卵用），分为两步：

1. 先确定 出现次数最多的数

② 遍历求该数出现的次数

## 第19节 斐波那契数列

## 题目描述

> 大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项。

##输入描述

> 一个整数n

##输出描述

> 斐波那契数列的第n项。

##题目分析

###什么是斐波那契数列？

> 斐波那契数列（Fibonacci sequence），又称黄金分割数列

>

> 在数学上，斐波纳契数列以如下被以递归的方法定义：\*\*F（0）=0，F（1）=1\*\*，F（n）=F(n-1)+F(n-2)（n≥2，n∈N\*）

>

> 指的是这样一个数列：0、1、1、2、3、5、8、13、21、34、……

###递归？

　　看到这个的第一想法就是用递归，当ｎ＜２时返回ｎ，ｎ＞＝２时返回ｆ(n-1)+f(n-2), 于是就来试一下...

```

public class Solution {

public int Fibonacci(int n) {

if(n<2){

return n;

}

return Fibonacci(n-1)+Fibonacci(n-2);

}

}

```

> 运行超时:您的程序未能在规定时间内运行结束，请检查是否循环有错或算法复杂度过大。

为什么呢？

　　因为重复计算，比如：

```

ｆ(4) = ｆ(3) + ｆ(2);

　 = ｆ(2) + ｆ(1) + ｆ(1) + ｆ(0);

　 = ｆ(1) + ｆ(0) + ｆ(1) + ｆ(1) + ｆ(0);

```

求ｆ(4)就要计算三次ｆ(1)和两次ｆ(0)，显然这是不行的。

> 解法 (动态规划)

>

> 运行时间：27ms 占用内存：629k

```

public class Solution {

public int Fibonacci(int n) {

int i = 0, j = 1;

for(;n>0;n--){

j += i;

i = j-i;

}

return i;

}

}

```

根据n的大小，从f(0)=i 和 f(1)=j 从头开始遍历整个序列

> 有f(n)=f(n-1)+f(n-2) （\*\*n≥2\*\*，n∈N\*）

>

j+=i, 使j成为新的f(n-1)

> i = j-i ，使i成为f(n-2)

>完成后，返回 f(n-2)

注意：java中

> while(n--)

会报编译错误：

> required: boolean

found: int

## 第20节 旋转数组的最小数字

## 题目描述

> 把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个\*\*非递减序列\*\*的一个旋转，输出旋转数组的最小元素。例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。

##输入描述

> 一个非递减序列的一个旋转数组

##输出描述

> 输出旋转数组的最小元素

##题目分析

　　原数组最小的值必然是第一个，旋转后数组是两个非递减数组的拼接，只要找到第二个非递减数组的第一个元素就是最小值。

　　遍历旋转数组，只要有array[i+1]小于array[i],那么array[i+1]必是最小值（这里不能取等，比如［２，２，１，２，２］取等就返回２了，至于特殊情况［１，１，１，１，１］遍历完没有结果，在最后return array[0]就可以了）。

```

import java.util.ArrayList;

public class Solution {

public int minNumberInRotateArray(int [] array) {

if(array.length==0){

return 0;

}

for(int i =0;i<array.length-2;i++){

if(array[i+1] < array[i]){

return array[i+1];

}

}

return array[0];

}

}

```

　　注意：

　　题目要求数组 [] 要求返回 0，其实我觉得该返回 -1；

　　数组不要越界，也不要直接求一个一维数组的最小值（就是先设第一个最小，遍历数组比它小的就赋值为最小，最后返回），你觉得别人给你说半天旋转都是空话吗？

## 第21节 浮点数的整数次方

## 题目描述

> 给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

##输入描述

> base，exponent

##输出描述

> base的exponent次方

##题目分析

　　首先要注意，指数正负和零的情况判别：

　　①任何数的０次方等于０

　　②０不能做除数（也就是指数为负时，基数不能为０）

> 解法一　　运行时间：27ms　占用内存：636k

```

public class Solution {

public double Power(double base, int exponent) {

int flag = exponent;//接下来exponent会改变

if(exponent==0){

return 1;

}

if(exponent<0){

exponent = -exponent;

if(base==0){

throw new RuntimeException("分母不能为0");

}

}

double result = 1;

for(int i=0;i<exponent;i++){

result\*=base;

}

return flag < 0 ? 1 / result : result;

}

}

```

> 解法二 　运行时间：30ms　占用内存：510k

　把指数以二进制的形式表达：

举例（10^13）有10^1101 = 10^0001\*10^0100\*10^1000。

通过&1和>>1来逐位读取1101，为1时将该位代表的乘数累乘到最终结果。

```

public class Solution {

public double Power(double base, int exponent) {

int flag = exponent;//接下来exponent会改变

if(exponent==0){

return 1;

}

if(exponent<0){

exponent = -exponent;

if(base==0){

throw new RuntimeException("分母不能为0");

}

}

double result = 1;

while(exponent!=0){

if((exponent&1)==1){

result\*=base;

}

exponent>>=1;

base \*=base; //指数右移一位，底数翻倍

}

return flag < 0 ? 1 / result : result;

}

}

```

> 解法三 　运行时间：33ms 　占用内存：629k

　　调用库函数

```

public class Solution {

public double Power(double base, int exponent) {

return Math.pow(base,exponent);

}

}

```

因为pow最终会调用一个native方法，所以时间上还是可以的。

## 第22节 用两个栈实现队列

## 题目描述

> 用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

##题目分析

　　用两个栈实现ＦＩＦＯ

![这里写图片描述](http://img.blog.csdn.net/20160424194132456)

　　情况一：如果出栈时stack2为空，则直接把stack1全部出栈到stack2之后，返回stack2.pop()就可以得出 出栈结果。

![这里写图片描述](http://img.blog.csdn.net/20160424194432779)

　　情况二：已经出栈过，stack2中有数据。stack1中也有数据，现在要出栈。就要先把stack1的全部数据出栈到stack2，然后再返回stack2.pop()的值就行。

　　实现代码：

> 运行时间：27ms 占用内存：503k

```

import java.util.Stack;

public class Solution {

Stack<Integer> stack1 = new Stack<Integer>();

Stack<Integer> stack2 = new Stack<Integer>();

public void push(int node) {

stack1.push(node);

}

public int pop() {

if(stack2.isEmpty()){

while(!stack1.isEmpty()){

stack2.push(stack1.pop());

}

}

return stack2.pop();

}

}

```

![这里写图片描述](http://img.blog.csdn.net/20160424195440070)

　值得注意的是：

　　isEmpty()是 stack父类vector中的方法，stack中判空的函数是 empty()，虽然都是判断当前集合的元素个数是否为0。但是isEmpty()是同步的方法，empty()不是线程同步的。并且用isEmpty()的时间效率更高！

## 第23节 矩形覆盖

## 题目描述

> 我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？

##输入描述

> 一个大矩形

##输出描述

> 覆盖的方法数

##题目分析

>设 被n个2\*1的小矩形无重叠地覆盖的方法总数为 f(n)

- 当n=1时，明显f(1)=1;

- 当n=2时，只能两个都横着或两个都竖着放，有f(2)=2;

- 当小矩形个数为n，来覆盖这个\*\*2\*n\*\*的大矩形。第一步只有两种放法：

　①竖着放，那么剩下的摆放总数为 f(n-1)

![这里写图片描述](http://img.blog.csdn.net/20160428100300772)

　　　②横着放，那么剩下的摆放总数为 f(n-２)。因为它下面的那块也跟随着它的摆放而确定（必须是一个横着放的小矩形）。

![这里写图片描述](http://img.blog.csdn.net/20160428100709583)

很容易看出满足斐波那契数列。

> 斐波那契数列（Fibonacci sequence），又称黄金分割数列

> 在数学上，斐波纳契数列以如下被以递归的方法定义：F（0）=0，F（1）=1，F（n）=F(n-1)+F(n-2)（n≥2，n∈N\*）

> 指的是这样一个数列：0、1、1、2、3、5、8、13、21、34、……

可以得出递推公式：

　　　　 | 1 　　　　(n=0 )

f(n) = 　 | 1 　　　　 (n=1 )

　　　　 | f(ｎ-1)＋f(n-2)　　(n>=2)

> 解法一 （递归） 　运行时间：924ms 　占用内存：654k

```

public class Solution {

public int RectCover(int target) {

if(target<=1) return 1;

return RectCover(target-1)+RectCover(target-2);

}

}

```

递归效率不高，重复计算多，比如：

```

ｆ(4) = ｆ(3) + ｆ(2);

　 = ｆ(2) + ｆ(1) + ｆ(1) + ｆ(0);

　 = ｆ(1) + ｆ(0) + ｆ(1) + ｆ(1) + ｆ(0);

```

求ｆ(4)就要计算三次ｆ(1)和两次ｆ(0)，显然这是不行的。

> 解法二（动态规划）　　运行时间：29ms　占用内存：629k

```

public class Solution {

public int RectCover(int target) {

if(target<=1) return 1;

int i =1;//f(0)

int j =1;//f(1)

for(;target>=2;target--){

j+=i;

i=j-i;

}

return j;

}

}

```

显然这个快很多，n>=2时，根据 f(n)=f(ｎ-1)＋f(n-2)进行依次计算，最后得出 f（target）并返回。

## 第24节 调整数组顺序使奇数位于偶数前面

## 题目描述

> 输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

##输入描述

> 整数数组

##输出描述

> 奇数在前偶数在后的，调整后的数组

##题目分析

> 解法一　　运行时间：26ms 　 占用内存：526k

```

import java.util.ArrayList;

import java.util.List;

public class Solution {

public void reOrderArray(int [] array) {

　　if(array.length==0 || array==null){

return;

}

List<Integer> oddList = new ArrayList<Integer>();

List<Integer> evenList = new ArrayList<Integer>();

for(int i=0;i<array.length;i++){

if(array[i]%2==0){

evenList.add(array[i]);

}else{

oddList.add(array[i]);

}

}

oddList.addAll(evenList);

for(int i=0;i<array.length;i++){

array[i] = oddList.get(i);

}

}

}

```

　　这个思路比较简单，遍历array把奇数和偶数分别放在两个新建的数组中，然后合并再赋值给array就行了。

> 解法二　运行时间：27ms 　占用内存：503k

```

public class Solution {

public void reOrderArray(int [] array) {

if(array.length==0 || array==null){

return;

}

for(int i=0;i<array.length-1;i++){

for(int j=0;j<array.length-i-1;j++){

if(array[j]%2==0 && array[j+1]%2==1){

int temp = array[j];

array[j] = array[j+1];

array[j+1] = temp;

}

}

}

}

}

```

　　类似于冒泡排序，以为用到array[j+1]数组不要越界。

## 第25节 跳台阶

## 题目描述

> 一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

##输入描述

> 台阶级数 target

##输出描述

> 多少种跳法

##题目分析

- 假设跳上n阶台阶时有f(n)种跳法

- 要跳上ｎ阶只能从ｎ－１阶或是ｎ－２阶跳上去

- 那么有ｆ（ｎ）＝ｆ（ｎ－１）＋ｆ（ｎ－２）成立，这符合[斐波那契数列](http://blog.csdn.net/amazing7/article/details/51251040)

- 显然n=1时 f(1)=1，n=2时f(2)=2，n=3时f(3)=f(1)+f(2)=3, 我们得出递推公式：

　　　　 | 1, (n=1)

f(n) = 　 | 2, (n=2)

　　　　 | f(n-1)+f(n-2) ,(n>2,n为整数)

> 解法一（递归）　　运行时间：444ms 占用内存：629k

```

public class Solution {

public int JumpFloor(int target) {

if(target<=0) return 0;

if(target<=2) return target;

return JumpFloor(target -1)+JumpFloor(target -2);

}

}

```

注意输入限制，在上一题中说到了递归调用效率不高，<font color=##ee2fcf size=5 face="微软雅黑"> 不推荐 </font>

> 解法二 （动态规划） 运行时间：33ms 占用内存：654k

```

public class Solution {

public int JumpFloor(int target) {

if(target<=0) return 0;

if(target<=2) return target;

int i =1;

int j =2;

for(;target>2;target--){

j+=i;

i=j-i;

}

return j;

}

}

```

　　首先，可以明显看出运行时间只有递归的十分之一不到。n=1时 f(1)=1，n=2时f(2)=2直接返回.

根据n的大小，从f(１)=i 和 f(２)=j 从头开始遍历整个序列，由ｆ（ｎ）＝f(n-1)+f(n-2)　（n>2），依次求得后面的结果，最后求得ｆ（target）并返回。

//变态跳台阶

## 题目描述

> 一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法

##输入描述

> 台阶数

##输出描述

> 跳法数

##题目分析

- 设n阶的跳数为f(n)

- 当n=1时，f(1) = 1

- 当n=2时，分为最后一步 跳2阶和跳1阶 两种情况，有f(2)=f(0)+f(1)=1+1=2

- 当n=3时，分为最后一步 跳3阶、跳2阶和跳1阶 三种情况，有f(3)=f(0)+f(1)+f(2)=1+1+2=4

- 有 　　 f(n) = f(n-1)+f(n-2)+...+f(1) + f(0)成立

同时有　f(n-1)＝f(n-2)+...+f(1) + f(0)　成立，可得出f(n)=2f(n-1) (n>=2)

![这里写图片描述](http://img.blog.csdn.net/20160427145606607)

很明显可以得出递推公式：

　　　　 | 1 　　　　(n=0 )

f(n) = 　 | 1 　　　　 (n=1 )

　　　　 | 2\*f(n-1)　　(n>=2)

> 解法一 　运行时间：35ms 　 占用内存：654k

```

public class Solution {

public int JumpFloorII(int target) {

if(target<=1) return 1;

return 2\*JumpFloorII(target-1);

}

}

```

> 解法二 　运行时间：34ms　 占用内存：654k

```

public class Solution {

public int JumpFloorII(int target) {

　　 if(target<=1) return 1;

return 1<<(target-1);

}

}

```

　　换一种思路想一下：一共有ｎ个台阶，最后一个台阶是一定要跳上去的，其他的　ｎ－１个可跳可不跳，一共有多少总情况？

> ２（ｎ－１）

这里用移位实现乘法，时间上要快一些！

## 第26节 重建二叉树

## 题目描述

> 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

##输入描述

>int [] pre　前序遍历序列

>int [] in　　中序遍历序列

##输出描述

>重建后二叉树的根节点

##题目分析

> 二叉树节点定义

```

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

```

思路：

- 先求出根节点（前序序列第一个元素）

- 将根节点带入到中序遍历序列中求出左右子树的中序遍历序列

- 通过左右子树的中序序列元素集合的\*\*长度\*\*带入前序遍历序列可以求出左右子树的前序序列

- 左右子树的前序序列第一个元素分别是根节点的左右儿子

- 然后 \*\*递归\*\*

```

public class Solution {

public TreeNode reConstructBinaryTree(int [] pre,int [] in) {

return reConstructBTree(pre,0,pre.length-1,in,0,in.length-1);

}

private TreeNode reConstructBTree(int [] pre,int startPre,int endPre,int [] in,int startIn,int endIn) {

if(startPre>endPre||startIn>endIn)

return null;

TreeNode root=new TreeNode(pre[startPre]);

for(int i=startIn;i<=endIn;i++)

if(in[i]==root.val){

root.left=reConstructBTree(pre,startPre+1,startPre+i-startIn,in,startIn,i-1);

root.right=reConstructBTree(pre,i-startIn+startPre+1,endPre,in,i+1,endIn);

}

return root;

}

}

```

## 第27节 链表中倒数第k个结点

## 题目描述

> 输入一个链表，输出该链表中倒数第k个结点。

##输入描述

> 链表

##输出描述

> 倒数第k个结点

##题目分析

节点描述：

```

public class ListNode {

int val;

ListNode next = null;

ListNode(int val) {

this.val = val;

}

}

```

> 解法一 运行时间：35ms 占用内存：503k

```

public class Solution {

public ListNode FindKthToTail(ListNode head,int k) {

if(head==null || k<=0){

return null;

}

ListNode last = head;

for(int i=1;i<k;i++){

last = last.next;

if(last==null){

return null;

}

}

while(last.next!=null){

head = head.next;

last = last.next;

}

return head;

}

}

```

　　①首先判断边界，新建一个节点last，开始指向head节点；

　　②向后遍历Ｋ－１次使它与head节点相差ｋ个节点（如果链表长度小于ｋ就返回null）；

　　③两个节点同时向后遍历，直到last指向最后一个节点为止。

　　④返回现在的head节点就是倒数第ｋ个节点。

> 解法二 　运行时间：34ms　占用内存：629k

```

public class Solution {

public ListNode FindKthToTail(ListNode head,int k) {

　　　　　if(head==null || k<=0){

return null;

}

ListNode p = head;

ListNode q = head;

int i = 0;

for (; p != null; i++) {

if (i >= k){

q = q.next;

}

p = p.next;

}

return i < k ? null : q;

}

}

```

　　与解法一是同一种思路，不过只进行一次遍历，事件复杂度O(n)。

## 第28章 顺时针打印矩阵

## 题目描述

> 输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下矩阵：

> 1 2 3 4

> 5 6 7 8

> 9 10 11 12

> 13 14 15 16

则依次打印出数字1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

##输入描述

> 一个矩阵

##输出描述

>矩阵的顺序打印序列

##题目分析

> 解法 　 运行时间：33ms　占用内存：629k

```

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> printMatrix(int [][] matrix) {

if(matrix == null){

return null;

}

ArrayList<Integer> list=new ArrayList<Integer> ();

int row = matrix.length;

int col = matrix[0].length;

int left=0,right=col-1,up=0,down=row-1;

while(left<=right&&up<=down){

//左上到右上

for(int i=left;i<=right;i++){

list.add(matrix[up][i]);

}

//右上到右下

for(int i=up+1;i<=down;i++){

list.add(matrix[i][right]);

}

//右下到左下，判断是否与 左上到右上 重复

if(up!=down){

for(int i=right-1;i>=left;i--){

list.add(matrix[down][i]);

}

}

//左下到左上，判断是否与 右上到右下 重复

if(left!=right){

for(int i=down-1;i>up;i--){

list.add(matrix[i][left]);

}

}

left++;right--;up++;down--;

}

return list;

}

}

```

这代码看似比较多，其实思路很简单：

①判空

②分别得到矩阵的行数和列数

1. 定初始转一圈的上下左右四个点的位置
2. 根据这四个点依次把数据添加到list中,每转一圈要改变点的位置
3. 返回list