# Android\_QA

## Java 部分（一）基础知识点

### 1. Java中 == 和 equals 和 hashCode 的区别

Java中的数据类型可分为两类，基本数据类型和引用类型。基本数据类型：byte、short、char、int、long、float、double、boolean。他们之间的比较用双等号（==），比较的是**值**。引用类型：类、接口、数组。当他们用双等号（==）进行比较的时候，比较的是他们在**内存中的存放地址**。对象是放在堆中的，栈中存放的是对象的引用（地址）。由此可见，**双等号是对栈中的值进行比较的**。如果要比较堆中对象是否相同，那么就要重写equals方法了。

默认情况下（没有覆盖equals方法）下的equals方法都是调用Object类的equals方法，而Object的equals方法主要是用于判断**对象的内存地址引用是不是同一个地址**（是不是同一个对象）。下面是Object类中的equals方法：

public boolean equals(Object obj) {   
 return (this == obj);   
}

定义的equals方法与==是等效的。

**但是**，要是类中覆盖了equals方法，那么就要根据具体代码来确定equals方法的作用了。**覆盖后的一般都是通过对象的内容是否相等来判断对象是否相等。**下面是String类对equals方法进行了重写：

public boolean equals(Object anObject) {   
 if (this == anObject) {   
 return true;   
 }   
 if (anObject instanceof String) {   
 String anotherString = (String)anObject;   
 int n = count;   
 if (n == anotherString.count) {   
 char v1[] = value;   
 char v2[] = anotherString.value;   
 int i = offset;   
 int j = anotherString.offset;   
 while (n-- != 0) {   
 if (v1[i++] != v2[j++])   
 return false;   
 }   
 return true;   
 }   
 }   
 return false;   
}

hashCode()方法返回的就是一个数值，从方法名上来看，其目的就是生成一个hash码，hash码的主要用途就是在**对对象进行散列的时候作为key输入**。

参考自：<http://blog.csdn.net/hla199106/article/details/46907725>

### 2. int、char、long 各占多少字节数

|  |  |  |
| --- | --- | --- |
| 类型 | 位数 | 字节数 |
| boolen | 8 | 1 |
| int | 32 | 4 |
| float | 32 | 4 |
| double | 64 | 8 |
| char | 16 | 2 |
| byte | 8 | 1 |
| short | 16 | 2 |
| long | 64 | 8 |

### 3. int 和 Integer 的区别

Java 为每一个基本数据类型都引入了对应的包装类，从Java 5 开始引入了自动装箱 / 拆箱机制，使得两者可以相互转化。**所以最基本的一点区别就是**：Integer 是int的包装类，int的初始值为零，而Integer的初值为null。int与Integer相比，会把Integer自动拆箱为int再去比。

参考自：<http://blog.csdn.net/login_sonata/article/details/71001851>

### 4. 谈谈对Java多态的理解

多态即：事物在运行过程中存在不同的状态。多态的存在有三个前提：要有继承关系、子类重写父类方法、父类数据类型的引用指向子类对象。弊端就是：不能使用子类特有的成员属性和子类特有的成员方法。如果非要用到不可，可以强制类型转换。

参考自：<https://item.btime.com/m_2s22uri6z16>

### 5. String、StringBuffer、StringBuilder的区别

String：字符串常量，使用字符串拼接时是不同的两个空间。

StringBuffer：字符串变量，线程安全，字符串拼接直接在字符串后追加。

StringBuilder：字符串变量，非线程安全，字符串拼接直接在字符串后追加。

StringBuilder执行效率高于StringBuffer高于String。String是一个常量，是不可变的，所以对于每次+=赋值都会创建一个新的对象，StringBuilder和StringBuffer都是可变的，当进行字符串拼接的时候采用append方法，在原来的基础上追加，所以性能要比String高，因为StringBuffer是线程安全的而StringBuilder是线程非安全的，所以StringBuilder的效率高于StringBuffer。对于大数据量的字符串拼接，采用StringBuffer、StringBuilder。另一种说法，JDK1.6做了优化，通过String声明的字符串在进行用+进行拼接时，底层调用的是StringBuffer，所以性能基本上和后两者没什么区别。

### 6. 什么是内部类？内部类的作用

Java 常见的内部类有四种：成员内部类、静态内部类、方法内部类和匿名内部类。

内部类可以很好的实现隐蔽，一般的非内部类，是不允许有private和protectd等权限的，但内部类（除方法内部类）可以通过这些修饰符来实现隐蔽。

内部类拥有外部类的访问权限（分静态与非静态情况），通过这一特性可以比较好的处理类之间的关联性，将一类事物的流程放在一起内部处理。

通过内部类可以实现多重继承，Java默认是单继承的，我们可以通过多个内部类继承实现多个父类，接着由于外部类完全可访问内部类，所以就实现了类似多继承的效果。

参考自：<https://www.jianshu.com/p/7fe3af7f0f2d>

### 7. 抽象类和接口的区别

抽象类使用abstract class 定义，抽象类既可以有抽象方法也可以有其他类型的方法，既可以有静态属性也可以有非静态属性。

接口使用interface定义，属性用public final static 修饰，如果没有写关键字，虚拟机默认会加上关键字。JDK8之后接口里的方法既可以有抽象方法也可以有默认方法。

抽象类是一种类别，具有构造方法。接口是一种行为规范，没有构造方法。抽象类中可以没有抽象方法，他有自己的构造方法但是不能直接实例化对象，所以必须被子类继承之后才有意义。

抽象类只能单继承，接口可以被多个类实现，接口和接口之间可以多继承。

抽象类可以由默认的方法实现，接口根本不存在方法的实现。实现抽象类使用extends关键字来继承抽象类，如果子类不是抽象类，它需要提供抽象类中所有抽象方法的实现。子类使用关键字implements来实现接口，它需要提供接口中所有声明的方法的实现。

抽象方法可以有public、protectd 和 default 这些修饰符，而接口方法默认修饰符是public，不可以使用其他修饰符。

抽象类比接口速度快，接口是稍微有点慢的，这是因为它需要时间去寻找类中实现的方法。

参考自：<http://www.importnew.com/12399.html>

### 8. 抽象类的意义

为子类提供一个公共的类型；封装子类中重复的内容；定义有抽象方法，子类虽然有不同的实现，但该方法的定义是一致的；

### 9. 抽象类与接口的应用场景

如果你拥有一些方法并且想让它们中的一些默认实现，那么就用抽象类；如果你想实现多继承，那么必须实用接口；如果基本功能在不断变化，那么就需要使用抽象类，如果不断改变基本功能并且使用接口，那么就要改变所有实现了该接口的类。

### 10. 抽象类是否可以没有方法和属性？

可以，抽象类可以没有方法和属性，但是含有抽象方法的类一定是抽象类。

### 11. 接口的意义

规范、扩展、回调。

对一些类是否具有某个功能非常关心，但不关心功能的具体实现，那么就需要这些类实现一个接口。

### 12. 泛型中的extends和super的区别

<? extends T>和<? super T>是泛型中的“通配符”和“边界”的概念。  
<? extends T>：是指“上界通配符”，不能往里存，只能往外取。  
<? super T>：是指“下界通配符”，不影响往里存，但往外取只能放在Object对象里。

**PECS原则：频繁往外读取内容，适合用上界Extends，经常往里插入的，适合用下界Super。**

参考自：[[Java]泛型中 extends 和 super 的区别？](https://itimetraveler.github.io/2016/12/27/%E3%80%90Java%E3%80%91%E6%B3%9B%E5%9E%8B%E4%B8%AD%20extends%20%E5%92%8C%20super%20%E7%9A%84%E5%8C%BA%E5%88%AB%EF%BC%9F/)

### 13. 父类的静态方法能否被子类重写？

不能，子类继承父类后，非静态方法覆盖父类的方法，父类的静态方法被隐藏。

### 14. 进程和线程的区别

进程是资源分配的基本单位，线程是处理器调度的基本单位。

进程拥有独立的地址空间，线程没有独立的地址空间，同一个进程内多个线程共享其资源。

线程划分尺度更小，所以多线程程序并发性更高。

一个程序至少有一个进程，一个进程至少有一个线程。

线程是属于进程的，当进程退出时该进程所产生的线程都会被强制退出并清除。

线程占用的资源要少于进程所占用的资源。

### 15. final、finally、finalize的区别

final用于声明属性、方法和类，分别表示属性不可变，方法不可覆盖，类不可继承。

finally是异常处理语句结构的一部分，表示总是执行。

finalize是Object类的一个方法，在垃圾收集器执行的时候会回调被回收对象的finalize()方法，可以覆盖此方法提供垃圾收集时其他资源的回收，例如关闭文件。

### 16. 序列化的方式

实现Serializable接口和实现Parcelable接口。

### 17. Serializable 和 Parcelable 的区别

Serializable 是 Java的序列化接口。特点是简单，直接实现该接口就行了，其他工作都被系统完成了，但是对于内存开销大，序列化和反序列化需要很多的 I/O 流操作。

Parcelable 是Android的序列化方式，主要用于在内存序列化上，采用该方法需要实现该接口并且手动实现序列化过程，相对复杂点。如果序列化到存储设备（文件等）或者网络传输，比较复杂，建议用Serializable 方式。

**最大的区别就是存储媒介的不同：**Serializable 使用IO读写存储在硬盘上，Parcelable 是直接在内存中读写，内存读写速度远大于IO读写，所以Parcelable 更加高效。Serializable在序列化过程中会产生大量的临时变量，会引起频繁的GC，效率低下。

### 18. 静态属性和静态方法是否可以被继承？是否可以被重写？以及原因。

静态属性和静态方法可以被继承，但是没有被重写而是被隐藏。

静态方法和属性是属于类的，调用的时候直接通过 类名.方法名 完成，不需要继承机制就可以调用。如果子类里面定义了静态方法和属性，那么这时候父类的静态方法或属性称之为“隐藏”。至于是否继承一说，子类是有继承静态方法和属性，但是跟实例方法和属性不太一样，存在隐藏这种情况。

多态之所以能够实现依赖于继承、接口和重写、重载（继承和重写最为关键）。**有了继承和重写就可以实现父类的引用指向不同子类的对象。** 重写的功能是：重写后子类的优先级高于父类的优先级，但是隐藏是没有这个优先级之分的。

静态属性、静态方法和非静态的属性都可以被继承和隐藏而不能被重写，因此不能实现多态，不能实现父类的引用可以指向不同子类的对象。

非静态方法可以被继承和重写，因此可以实现多态。

### 19. 静态内部类的设计意图

只是为了降低包的深度，方便类的使用，静态内部类适用于包含类当中，但又不依赖于外在的类，不用使用外在类的非静态属性和方法，只是为了方便管理类结构而定义。在创建静态内部类的时候，不需要外部类对象的引用。

非静态内部类有一个很大的优点：可以自由使用外部类中的变量和方法。

static class Outer {  
 class Inner {}  
 static class StaticInner {}  
}  
  
Outer outer = new Outer();  
Outer.Inner inner = outer.new Inner();  
Outer.StaticInner inner0 = new Outer.StaticInner();

参考自：<https://www.zhihu.com/question/28197253>

### 20. 成员内部类、静态内部类、方法内部类（局部内部类）和匿名内部类的理解，以及项目中的应用

成员内部类：

最普通的内部类，它的定义位于一个类的内部，这样看起来，成员内部类相当于外部类的一个成员，成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）。不过需要注意的是，当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。如果要访问外部类的同名成员，形式如下：

外部类.this.成员变量  
外部类.this.成员方法

虽然成员内部类可以无条件的访问外部类的成员，而外部类想访问成员内部类的成员却不是那么随心所欲了。成员内部类是依附外部类而存在的，也就是说，如果要创建成员内部类的对象，前提必须存在一个外部类对象。创建成员内部类对象的一般方式如下：

public class Test {  
 public static void main(String[] args) {  
 //第一种方式：  
 Outter outter = new Outter();  
 Outter.Inner inner = outter.new Inner(); //必须通过Outter对象来创建  
   
 //第二种方式：  
 Outter.Inner inner1 = outter.getInnerInstance();  
 }  
}  
   
class Outter {  
 private Inner inner = null;  
 public Outter() {  
   
 }  
   
 public Inner getInnerInstance() {  
 if(inner == null)  
 inner = new Inner();  
 return inner;  
 }  
   
 class Inner {  
 public Inner() {  
   
 }  
 }  
}

内部类可以拥有private、protected、public以及包访问权限。如果成员内部类用private修饰，则只能在外部类的内部访问，如果用public修饰，则任何地方都能访问，如果用protected修饰，则只能在同一个包下或者继承外部类的情况下访问，如果是默认访问权限，则只能是同一个包下访问。这一点和外部类有一点不一样，外部类只能被public和包访问两种权限修饰。由于成员内部类看起来像外部类的一个成员，所以可以像类的成员一样拥有多种修饰权限。

静态内部类：

静态内部类也是定义在另一个类里面的类，只不过在类的前面多了一个关键字static。静态内部类是不需要依赖外部类的，这点和类的静态成员属性有点相似，并且它不能使用外部类的非static成员变量或者方法。这点很好理解，因为在没有外部类的对象的情况下，可以创建静态内部类的对象，如果允许访问外部类的非static成员就会产生矛盾，因为外部类的非static成员必须依赖于具体的对象。

局部内部类：

局部内部类是定义在一个方法或者一个作用域里面的的类，它和成员内部类的区别在于局部内部类的访问权限仅限于方法内或者该作用域内。注意，局部内部类就像是方法里面的一个局部变量一样，是不能有public、protected、private以及static修饰符的。

匿名内部类：

匿名内部类不能有访问修饰符和static修饰符的，一般用于编写监听代码。匿名内部类是唯一一个没有构造器的类。正因为没有构造器，所以匿名内部类的适用范围非常有限，大部分匿名内部类用于接口回调。一般来说，匿名内部类用于继承其他类或者实现接口，并不需要增加额外的方法，只是对继承方法的实现或者是重写。

应用场景：

1. 最重要的一点，每个内部类都能独立的继承一个接口的实现，无论外部类是否已经继承了某个接口的实现，对于内部类都没有影响。内部类使得多继承的解决方案变得完整。
   1. 方便将存在一定逻辑关系的类组织在一起，又可以对外界隐蔽。
   2. 方便编写事件驱动程序
   3. 方便编写线程代码

总结：

对于成员内部类，必须先产生外部类的实例化对象，才能产生内部类的实例化对象。而非静态内部类不用产生外部类的实例化对象即可产生内部类的实例化对象。

创建静态内部类对象的一般形式：  
外部类类名.内部类类名 xxx = new 外部类类名.内部类类名()  
创建成员内部类对象的一般形式：  
外部类类名.内部类类名 xxx = new 外部类对象名.new 内部类类名()

参考自：<http://www.cnblogs.com/latter/p/5665015.html>

### 21. 谈谈对kotlin的理解

挖个坑，以后自己填！哼哼（傲娇脸

### 22. 闭包和局部内部类的区别

### 23. String转换成Integer的方式以及原理

String转int：

int i = Integer.parseInt(s);  
int i = Integer.valueOf(s).intValue();

int转String：

String s = i + "";  
String s = Integer.toString(i);  
String s = String.valueOf(i);

源码：

public static int parseInt(String s, int radix) throws NumberFormatException   
 {   
 /\*   
 \* WARNING: This method may be invoked early during VM initialization   
 \* before IntegerCache is initialized. Care must be taken to not use   
 \* the valueOf method.   
 \*/   
 // 下面三个判断好理解，其中表示进制的 radix 要在（2~36）范围内   
 if (s == null) {   
 throw new NumberFormatException("null");   
 }   
   
 if (radix < Character.MIN\_RADIX) {   
 throw new NumberFormatException("radix " + radix +   
 " less than Character.MIN\_RADIX");   
 }   
   
 if (radix > Character.MAX\_RADIX) {   
 throw new NumberFormatException("radix " + radix +   
 " greater than Character.MAX\_RADIX");   
 }   
   
 int result = 0; // 表示结果， 在下面的计算中会一直是个负数，   
 // 假如说 我们的字符串是一个正数 "7" ，   
 // 那么在返回这个值之前result保存的是 -7，   
 // 这个可能是为了保持正数和负数在下面计算的一致性   
 boolean negative = false;   
 int i = 0, len = s.length();   
   
   
 //limit 默认初始化为 最大正整数的 负数 ，假如字符串表示的是正数，   
 //那么result(在返回之前一直是负数形式)就必须和这个最大正数的负数来比较，判断是否溢出   
   
 int limit = -Integer.MAX\_VALUE;   
   
 int multmin;   
 int digit;   
   
 if (len > 0) { // 首先是对第一个位置判断，是否含有正负号   
 char firstChar = s.charAt(0);   
 if (firstChar < '0') { // Possible leading "+" or "-"   
 if (firstChar == '-') {   
 negative = true;   
   
 // 这里，在负号的情况下，判断溢出的值就变成了 整数的 最小负数了。   
 limit = Integer.MIN\_VALUE;   
   
 } else if (firstChar != '+')   
 throw NumberFormatException.forInputString(s);   
   
 if (len == 1) // Cannot have lone "+" or "-"   
 throw NumberFormatException.forInputString(s);   
 i++;   
 }   
   
 multmin = limit / radix;   
 // 这个是用来判断当前的 result 在接受下一个字符串位置的数字后会不会溢出。   
 // 原理很简单，为了方便，拿正数来说   
 // （multmin result 在计算中都是负数），假如是10   
 // 进制,假设最大的10进制数是 21，那么multmin = 21/10 = 2,   
 // 如果我此时的 result 是 3 ，下一个字符c来了，result即将变成   
 // result = result \* 10 + c;那么这个值是肯定大于 21 ，即溢出了，   
 // 这个溢出的值在 int里面是保存不了的，不可能先计算出   
 // result（此时的result已经不是溢出的那个值了） 后再去与最大值比较。   
 // 所以要通过先比较 result < multmin （说明result \* radix 后还比 limit 小）   
   
 while (i < len) {   
 // Accumulating negatively avoids surprises near MAX\_VALUE   
 digit = Character.digit(s.charAt(i++),radix);   
 if (digit < 0) {   
 throw NumberFormatException.forInputString(s);   
 }   
   
   
 // 这里就是上说的判断溢出，由于result统一用负值来计算，所以用了 小于 号   
 // 从正数的角度看就是 reslut > mulmin 下一步reslut \* radix 肯定是 溢出了   
 if (result < multmin) {   
 throw NumberFormatException.forInputString(s);   
 }   
   
   
   
 result \*= radix;   
   
 // 这里也是判断溢出， 由于是负值来判断，相当于 （-result + digit）> - limit   
 // 但是不能用这种形式，如果这样去比较，那么得出的值是肯定判断不出溢出的。   
 // 所以用 result < limit + digit 很巧妙   
 if (result < limit + digit) {   
   
 throw NumberFormatException.forInputString(s);   
 }   
 result -= digit; // 加上这个 digit 的值 （这里减就相当于加）   
 }   
 } else {   
 throw NumberFormatException.forInputString(s);   
 }   
 return negative ? result : -result; // 正数就返回 -result   
 }

参考自：<http://blog.csdn.net/stu_wanghui/article/details/38564177>

### 24. 面向对象思想

#### 设计原则 S.O.L.I.D

|  |  |
| --- | --- |
| 简写 | 翻译 |
| SRP | 单一职责原则 |
| OCP | 开放封闭原则 |
| LSP | 里氏替换原则 |
| ISP | 接口分离原则 |
| DIP | 依赖倒置原则 |

**单一职责原则**

修改一个类的原因应该只有一个

换句话说就是让一个类只负责一件事，当这个类需要做过多的事情的时候，就需要分解这个类。

如果一个类承担的职责过多，就等于把这些职责耦合在了一起，一个职责的变化可能会削弱这个类完成其它职责的能力。

**开放封闭原则**

类应该对扩展开放，对修改关闭

扩展就是添加新功能的意思，因此该原则要求在添加新功能时不需要修改代码。

符合开闭原则最典型的设计模式是装饰者模式，它可以动态地将职责附加到对象上，而不用去修改类的代码。

**里氏替换原则**

子类对象必须能够替换掉所有父类对象

继承是一种 IS - A 关系，子类需要能够当成父类来使用，并且需要比父类更特殊。

如果不满足这个原则，那么各个子类的行为上就会有很大差异，增加继承体系的复杂度。

**接口分离原则**

不应该强迫客户依赖于它们不用的方法

因此使用多个专门的接口比使用单一的总接口更好。

**依赖倒置原则**

高层模块不应该依赖于低层模块，两者都应该依赖于抽象。

抽象不应该依赖于细节，细节应该依赖于抽象。

高层模块包含一个应用程序中重要的策略选择和业务模块，如果高层模块依赖于低层模块，那么低层模块的改动就会直接影响到高层模块，从而迫使高层模块也需要改动。

依赖于抽象意味着：

* 任何变量都不应该持有一个指向具体类的指针或者引用
* 任何类都不应该从具体类派生
* 任何方法都不应该覆写它的任何基类中的已经实现的方法

#### 三大特性

**封装**

利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分离的独立实体。数据被保护在抽象数据类型的内部，尽可能的隐藏内部细节，只保留一些对外接口使之与外部发生联系。用户无需知道对象内部的细节，但可以通过对象对外提供的接口来访问该对象。

优点：

* 降低耦合：可以独立开发、测试、优化和修改等
* 减轻维护的负担：可以更容易的被程序猿理解，并且在调试的时候可以不影响其他模块
* 有效地调节性能：可以通过剖析确定哪些模块影响了系统的性能
* 提高软件的可重用性
* 降低了构建大型系统的分享：即使整个系统不可用，但是这些独立的模块却有可能是可用的

**继承**

继承实现了 IS - A 关系，继承应该遵循里氏替换原则，子类对象必须能够替换掉所有父类对象。

**多态**

多态分为编译时多态和运行时多态。编译时多态主要指方法的重载，运行时多态指程序中定义的对象引用所指向的具体类型在运行期间才确定。

运行时多态有三个条件：

* 继承
* 覆盖
* 向上转型

### 25. 对象拷贝理解？深拷贝、浅拷贝的区别？

首先要先明白为什么需要使用克隆呢？

克隆的对象可能包含一些已经修改过的属性，而 new 出来的对象的属性都还是初始化时候的值，所以当需要一个新的对象来保存当前对象的 "状态" 就需要克隆了。

那如何实现对象克隆呢？有两种办法：

1. 实现 Cloneable 接口并重写 Object 类中的 clone() 方法
2. 实现 Serialiable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆

深拷贝和浅拷贝的区别是什么？

* 浅拷贝

当对象被复制的时候只复制它本身和其中包含的值类型的成员变量，而引用类型的成员变量并没有复制。

* 深拷贝

除了对象本身被复制外，对象所包含的所有成员变量也将被复制。

Java 默认的是浅拷贝，如果想实现深拷贝，就需要对象所包含的引用类型的成员变量也需要实现 Cloneable 接口，或者实现 Serialiable 接口。

### 26. Enumeration 和 Iterator 的区别？

* 接口不同

Enumeration 只能读取集合数据，而不能对数据进行修改；Iterator 除了读取集合的数据之外，也能对数据进行删除操作；Enumeration 已经被 Iterator 取代了，之所以没有被标记为 Deprecated，是因为在一些遗留类（Vector、Hashtable）中还在使用。

java public interface Enumeration<E> { boolean hasMoreElements(); E nextElement(); default Iterator<E> asIterator() { return new Iterator<>() { @Override public boolean hasNext() { return hasMoreElements(); } @Override public E next() { return nextElement(); } }; } }

java public interface Iterator<E> { boolean hasNext(); E next(); default void remove() { throw new UnsupportedOperationException("remove"); } default void forEachRemaining(Consumer<? super E> action) { Objects.requireNonNull(action); while (hasNext()) action.accept(next()); } }

* Iterator 支持 fail-fast 机制，而 Enumeration 不支持

Enumeration 是 JDK 1.0 添加的接口。使用到它的函数包括 Vector、Hashtable 等类，Enumeration 存在的目的就是为它们提供遍历接口。

Iterator 是 JDK 1.2 才添加的接口，它是为了 HashMap、ArrayList 等集合提供的遍历接口。Iterator 是支持 fail-fast 机制的。

Fail-fast 机制是指 Java 集合 (Collection) 中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。例如：当某个线程 A 通过 Iterator 去遍历某集合的过程中，若该集合的内容被其他线程所改变了，那么线程 A 访问集合时，就会抛出 ConcurrentModificationException 异常，产生 fail-fast 事件。

## Java 部分（二）高级知识点

### 1. 哪些情况下的对象会被垃圾回收机制处理掉？

垃圾回收机制中最基本的做法是分代回收。内存区域被划分为三代：新生代、老年代和永久代。对于不同世代可以使用不同的垃圾回收算法。一般来说，一个应用中的大部分对象的存活时间都很短，基于这一点，对于新生代的垃圾回收算法就可以很有针对性。

参考自：[Java — 垃圾收集灵魂拷问三连（一）](http://omooo.top/2017/12/25/Java%20---%20%E5%85%B3%E4%BA%8E%E5%9E%83%E5%9C%BE%E6%94%B6%E9%9B%86%E7%81%B5%E9%AD%82%E6%8B%B7%E9%97%AE%E4%B8%89%E8%BF%9E%EF%BC%88%E4%B8%80%EF%BC%89)

### 2. 讲一下常见的编码方式？

编码的原因：

计算机存储信息的最小单元是一个字节即八个bit，所以能表示的字符范围是0 - 225 个。然而要表示的符号太多了，无法用一个字节来完全表示，要解决这个矛盾必须需要一个新的数据结构char，从char到byte必须编码。编码方式规定了转化的规则，按照这个规则就可以让计算机正确的表示我们的字符。编码方式有以下几种：

ASCII码：

总共有128个，用一个字节的低七位表示，0 - 31是控制字符，如回车键、换行等等。32 - 126是打印字符，可以通过键盘输入并且能够显示出来。

ISO-8859-1：

ISO组织在ASCII码基础上又制定了一些列标准用来扩展ASCII编码，其中IS-8859-1涵盖了大多数西欧语言字符，单字节编码，总共表示256个字节，应用广泛。

GB2312、GBK：

双字节编码，表示汉字。

UTF-16：

UTF - 16具体定义了Unicode字符在计算机中的存取方法。UTF - 16用两个字节来表示Unicode转化格式，这个是定长的表示方法，不论什么字符都可以用两个字节表示，两个字节是十六个bit，所以叫UTF - 16.

UTF- 8：

UTF-16统一采用两个字节表示一个字符，虽然表示上非常简单方便，但是有很大一部分字符用一个字节就可以表示，显然是浪费了存储空间。于是UTF-8应运而生，UTF-8采用一种变长技术，每个编码区域有不同的字码长度，不同类型的字符可以是由1-6个字节组成。

参考自：[几种常见的编码格式](https://www.cnblogs.com/mlan/p/7823375.html)

### 3. UTF-8编码中中文占几个字节，int型几个字节？

中文占字节数可以是2、3和4个的，最高为4个字节。

参考自：<http://blog.csdn.net/hellokatewj/article/details/24325653>

### 4. 静态代理和动态代理的区别，什么场景使用？

参考自：<https://www.jianshu.com/p/2f518a4a4c2b>

### 5. Java的异常体系

异常简介：异常是阻止当前方法继续执行的问题，如：文件找不到、网络连接失败、非法参数等等。发现异常的理想时期是编译阶段，然而编译阶段不能找出所有的异常，余下的问题必须在运行期间。

Java中的异常分为可查异常和不可查异常。

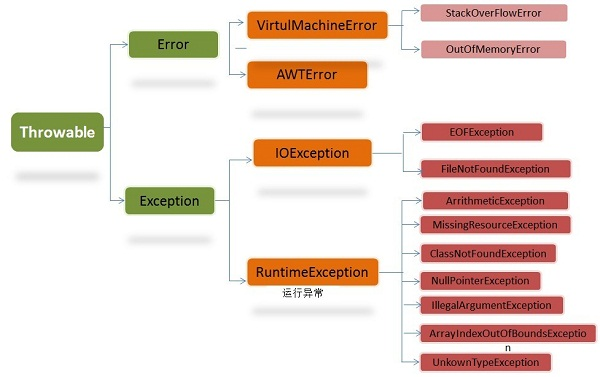
**可查异常：**

即编译时异常，只编译器在编译时可以发现的错误，程序在运行时很容易出现的异常状况，这些异常可以预计，所以在编译阶段就必须手动进行捕捉处理，即要用try-catch语句捕获它，要么用throws子句声明抛出，否者编译无法通过，如：IOException、SQLException以及用户自定义Exception异常。

**不可查异常：**

不可查异常包括\*\*运行时异常\*\*和\*\*错误\*\*，他们都是在程序运行时出现的。异常和错误的区别：异常能被程序本身处理，错误是无法处理的。\*\*运行时异常指的是\*\*程序在运行时才会出现的错误，由程序员自己分析代码决定是否用try...catch进行捕获处理。如空指针异常、类转换异常、数组下标越界异常等等。\*\*错误是指\*\*程序无法处理的错误，表示运行应用程序中较严重的问题，如系统崩溃、虚拟机错误、动态连接失败等等。大多数错误与代码编写者执行的操作无关，而表示代码运行时JVM出现的问题。

**Java异常类层次结构图：**



从上图可以看出Java通过API中Throwable类的众多子类描述各种不同的异常。因而，Java异常都是对象，是Throwable子类的实例。

**异常处理机制：**

当异常发生时，将使用new在堆上创建一个异常对象，对于这个异常对象，有两种处理方式：

1. 使用throw关键字将异常对象抛出，则当前执行路径被终止，异常处理机制将在其他地方寻找catch快对异常进行处理
2. 使用try...catch在当前逻辑中进行捕获处理

**throws 和 throw：**

throws：一个方法在声明时可以使用throws关键字声明可能会产生的若干异常

throw：抛出异常，并退出当前方法或作用域

使用 finally 清理：

在 Java 中的 finally 的存在并不是为了释放内存资源，因为 Java 有垃圾回收机制，因此需要 Java 释放的资源主要是：已经打开的文件或者网络连接等。

在 try 中无论有没有捕获异常，finally 都会被执行。

参考自：<https://www.jianshu.com/p/ffca876ce719>

### 6. 谈谈你对解析与分派的认识

[深入理解Java虚拟机之：多态性实现机制----静态分派与动态分配](https://www.jianshu.com/p/1976b01c07d2)

### 7. 修改对象A的equals方法的签名，那么使用HashMap存放这个对象实例的时候，会调用哪个equals方法？

### 8. Java中实现多态的机制是什么？

### 9. 如何将一个Java对象序列化到文件里？

### 10. 说说你对Java反射的理解

反射：在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 Java 语言的反射机制。

关于反射的基本使用可看：[Java-反射的理解与使用-(原创)](https://www.jianshu.com/p/2302ad5dbfad)

**反射是一种具有与类进行动态交互能力的一种机制**。

**反射的作用：**

在 Java 和 Android 开发中，一般情况下下面几种场景会用到反射机制：

* 需要访问隐藏属性或者调用方法改变程序原来的逻辑，这个在开发中是很常见的，由于一些原因，系统并没有开放一些接口出来，这个时候利用反射是一个有效的解决办法。
* 自定义注解，注解就是在运行时利用反射机制来获取的。
* 在开发中动态加载类，比如在 Android 中的动态加载解决65k问题等等，模块化和插件化都离不开反射。

**反射的工作原理**

我们知道，每个 Java 文件最终都会被编译成一个 .class 文件，这些 Class 对象承载了这个类的所有信息，包括父类、接口、构造函数、方法、属性等等，这些class文件在程序运行时会被 ClassLoader 加载到虚拟机中。当一个类被加载以后，Java 虚拟机就会在内存中自动产生一个 Class 对象，而一般情况下用 new 来创建对象，实际上本质都是一样的，只是底层原理对我们开发者透明罢了。有了 class 对象的引用，就相当于有了 Method、Field、Constructor 的一切信息，在 Java 中，有了对象的引用就有了一切，剩下就靠自己发挥了。

参考自：[Java反射以及在Android中的特殊应用](https://www.jianshu.com/p/8f394d90a85c)

### 11. 说说你对Java注解的理解

注解（Annotion）是一个接口（前加 @），程序可以通过反射来获取指定程序元素的Annotion对象，然后通过Annotion对象来获取注解里面的元数据。

注解可以用于创建文档，跟踪代码中的依赖性，甚至执行基本编译时检查。从某些方面看，Annotion就像修饰符一样被使用，并应用于包、类型、构造方法、方法、成员变量、参数、本地变量的声明中。

Annotation 的行为十分类似 public、final 这样的修饰符。每个 Annotation 具有一个名字和成员，每个 Annotation 的成员具有被称为 name=value 对的名字和值，name=value 装载了 Annotation 的信息，也就是说注解中可以不存在成员。

使用注解的基本规则：Annotation 不能影响程序代码的执行，无论增加、删除 Annotation，代码都始终如一的执行。

关于注解的基本使用可见：<https://www.jianshu.com/p/91393daaaf32>

参考自：<https://www.jianshu.com/p/8da24b7cf443>

从JDK5开始，Java增加了Annotation，注解是代码里的特殊标记，这些标记可以在编译、类加载、运行时被读取，并执行相应的处理。通过使用Annotation，开发人员可以在不改变原有逻辑的情况下，在源文件中嵌入一些补充的信息。

Annotation提供了一种为程序元素（包、类、构造器、方法、成员变量、参数、局部变量）设置元数据的方法。Annotation不能运行，**它只有成员变量，没有方法**。Annotation跟public、final修饰符一样，都是程序元素的一部分，Annotation不能作为一个程序元素使用。

#### 1. 定义Annotation

定义新的Annotation类型使用@interface关键字（在原有interface关键字前增加@符合），例如：

//定义注解  
public @interface Test{  
}  
//使用注解  
@Test  
public class MyClass{  
....  
}

**1.1 成员变量**

Annotation只有成员变量，没有方法。Annotation的成员变量在定义中以“无形参的方法”形式声明，其方法名定义了该成员变量的名字，其返回值定义了该成员变量的类型。例如：

//定义注解  
public @interface MyTag{  
 string name();  
 int age();  
 string id() default 0;  
}  
//使用注解  
public class Test{  
 @MyTag(name="红薯"，age=30)  
 public void info(){  
 ......  
 }  
}

一旦在Annotation里定义了成员变量后，使用该Annotation时就应该为该Annotation的成员变量指定值。

也可以在定义Annotation的成员变量时，为其指定默认值，指定成员变量默认值使用default关键字。

根据Annotation是否包含了成员变量，可以把Annotation分为以下两类：

* 标记Annotation

没有成员变量的Annotation被称为标记，这种Annotation仅用自身的存在与否来为我们提供信息，例如@override等

* 元数据Annotation

包含成员变量的Annotation。因为它可以接收更多的元数据，因此被称为元数据Annotation。

**1.2 元注解**

在定义Annotation时，也可以使用JDK提供的元注解来修饰Annotation定义。JDK提供了如下四个元注解（注解的注解，不是上诉的“元数据”）

* @Retention
* @Target
* @Documented
* @Inherited

**1.2.1 @Retention**

用于指定Annotation可以保留多长时间。

@Retention包含一个名为“value”的成员变量，该value成员变量是RetentionPolicy枚举类型。使用@Retention时，必须为其value指定值，value成员变量的值只能是如下三个：

* RetentionPolicy.SOURCE

Annotation只保留在源代码中，编译器编译时，直接丢弃这种Annotation

* RetentionPolicy.CLASS

编译器会把Annotation记录在class文件中，当运行Java程序时，JVM中不再保留该Annotation

* RetentionPolicy.RUNTIME

编译器把Annotation记录在class文件中，当运行Java程序时，JVM会保留该Annotation，程序可以通过反射获取该Annotation的信息。

//name=value形式  
//@Retention(value=RetentionPolicy.RUNTIME)  
  
//直接指定  
@Retention(RetentionPolicy.RUNTIME)  
public @interface MyTag{  
 String name() default "我兰";  
}

**1.2.2 @Target**

@Target指定Annotation用于修饰哪些程序元素。@Target也包含一个名为“value”的成员变量，该value成员变量类型为ElementType[]，同样为枚举类型，值有以下几个：

* ElementType.TYPE 能修饰类、接口或枚举类型
* ElementType.FIELD 能修饰成员变量
* ElementType.METHOD 能修饰方法
* ElementType.PARAMETER 能修饰参数
* ElementType.CONSTRUCTOR 能修饰构造器
* ElementType.LOCAL\_VARIABLE 能够修饰局部变量
* ElementType.ANNOTATION\_TYPE 能修饰注解
* ElementType.PACKAGE 能修饰包

//单个ElementType  
@Target(ElementType.FIELD)  
public @interface AnnTest {  
 String name() default "sunchp";  
}  
//多个ElementType  
@Target({ ElementType.FIELD, ElementType.METHOD })  
public @interface AnnTest {  
 String name() default "sunchp";  
}

**1.2.3 @Documented**

如果定义注解时，使用了@Documented修饰定义，则在用javadoc命令生成API文档后，所有使用该注解修饰的程序元素，将会包含该注解的说明。

**1.2.4 @Inherited**

指定Annotation具有继承性。

**1.3 基本Annotation**

* @Override

限定重写父类方法。对于子类中被@Override修饰的方法，如果存在对应的被重写的父类方法，则正确；如果不存在，则报错。@Override只能作用于方法，不能作用于其他程序元素。

* @Deprecated

用于表示某个程序元素（类、方法等等）已过时，如果使用被@Deprecated修饰的类或方法等，编译器会发出警号。

* @SuppressWarning

抑制编译器警号。指示被@SuppressWarning修饰的程序元素（以及该程序元素中的所有子元素，例如类以及该类中的方法...）取消显示指定的编译器警告，例如，常见的@SuppressWarning (value="unchecked")

* @SafeVarargs

告诉编译器忽略可变长度参数可能引起的类型转换问题，该注解修饰的方法必须为 static。

#### 2. 提取Annotation信息（反射）

当开发者使用了Annotation修饰了类、方法、Field等成员之后，这些Annotation不会自己生效，必须由开发者提供相应的代码来提取并处理Annotation信息，这些处理和提取Annotation的代码统称为APT（Annotation Processing Tool）。

JDK主要提供了两个类，来完成Annotation的提取：

* java.lang.annotation.Annotation

这个接口是所有Annotation类型的父接口

* java.lang.reflect.AnnotatedElement

该接口代表程序中被注解的程序元素

**2.1 java.lang.annotation.Annotation**

该接口源码：

package java.lang.annotation;  
  
public interface Annotation {  
  
 boolean equals(Object obj);  
  
 int hashCode();  
  
 String toString();  
  
 Class<? extends Annotation> annotationType();  
}

其中主要方法是annotationType()，用于返回该注解的java.lang.Class

**2.2 java.lang.reflect.AnnotatedElement**

接口源码：

package java.lang.reflect;  
  
import java.lang.annotation.Annotation;  
  
public interface AnnotatedElement {  
  
 //判断该程序元素上是否存在指定类型的注解，如果存在返回true，否则false  
 boolean isAnnotationPresent(Class<? extends Annotation> annotationClass);  
   
 //返回该程序元素上存在的指定类型的注解，如果该类型的注解不存在，则返回null  
 <T extends Annotation> T getAnnotation(Class<T> annotationClass);  
  
 //返回该程序元素上存在的所有注解  
 Annotation[] getAnnotations();  
  
 Annotation[] getDeclaredAnnotations();  
}

AnnotatedElement接口是所有程序元素（例如java.lang.Class、java.lang.reflect.Method、java.lang.reflect.Constructor等）的父接口。所以程序通过反射获取某个类的AnnotatedElement对象后，就可以调用该对象的isAnnotationPresent()、getAnnotation()等方法来访问注解信息。

**为了获取注解信息，必须使用反射知识。**

注：如果想要在运行时获取注解信息，在定义注解的时候，该注解必须要使用@Retention(RetentionPolicy.RUNTIME)修饰。

**2.3 示例**

**2.3.1 标记Annotation**

//定义注解  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface MyTag {  
  
}  
//注解处理  
class ProcessTool {  
 static void process(String clazz) {  
 Class targetClass = null;  
 try {  
 targetClass = Class.forName(clazz);  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 assert targetClass != null;  
 for (Method method : targetClass.getMethods()) {  
 if (method.isAnnotationPresent(MyTag.class)) {  
 System.out.println("被MyTag注解修饰的方法名：" + method.getName());  
 } else {  
 System.out.println("没有被MyTag注解修饰的方法名：" + method.getName());  
 }  
 }  
 }  
}  
//测试类  
public class TagTest {  
  
 @MyTag  
 public static void m1() {  
  
 }  
  
 public static void m2() {  
  
 }  
  
 public static void main(String[] args) {  
 ProcessTool.process("annotation.TagTest");  
 }  
}  
//输出  
没有被MyTag注解修饰的方法名：main  
被MyTag注解修饰的方法名：m1  
没有被MyTag注解修饰的方法名：m2  
没有被MyTag注解修饰的方法名：wait  
没有被MyTag注解修饰的方法名：wait  
没有被MyTag注解修饰的方法名：wait  
没有被MyTag注解修饰的方法名：equals  
没有被MyTag注解修饰的方法名：toString  
没有被MyTag注解修饰的方法名：hashCode  
没有被MyTag注解修饰的方法名：getClass  
没有被MyTag注解修饰的方法名：notify  
没有被MyTag注解修饰的方法名：notifyAll

**2.3.2 元数据Annotation**

//定义注解  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface MyTag {  
 String name() default "Omooo";  
  
 int age() default 21;  
}  
//注解处理  
class ProcessTool {  
 static void process(String clazz) {  
 Class targetClass = null;  
 try {  
 targetClass = Class.forName(clazz);  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 assert targetClass != null;  
 for (Method method : targetClass.getMethods()) {  
 if (method.isAnnotationPresent(MyTag.class)) {  
 MyTag tag = method.getAnnotation(MyTag.class);  
 System.out.println("方法：" + method.getName() + " 的注解内容为：" + tag.name() + " " + tag.age());  
 }  
 }  
 }  
}  
//测试类  
public class TagTest {  
  
 @MyTag  
 public static void m1() {  
  
 }  
  
 @MyTag(name = "当当猫", age = 20)  
 public static void m2() {  
  
 }  
  
 public static void main(String[] args) {  
 ProcessTool.process("annotation.TagTest");  
 }  
}  
//输出  
方法：m1 的注解内容为：Omooo 21  
方法：m2 的注解内容为：当当猫 20

#### 3. 注解本质

* 注解实质上会被编译器编译为接口，并且继承java.lang.annotation.Annotation接口
* 注解的成员变量会被编译器编译成同名的抽象方法
* 根据Java的class文件规范，class文件中会在程序元素的属性位置记录注解信息。

#### 4. 注解的意义

1. 为编译器提供辅助信息

Annotation可以为编译器提供额外的信息，以便于检测错误，抑制警告等。

1. 编译源代码时进行额外操作

软件工具可以通过处理Annotation信息来生成源代码，xml文件等等。

1. 运行时处理

有一些Annotation甚至可以在程序运行时被检测、使用。

总之，注解是一种元数据，起到了“描述、配置”的作用。

引自：<http://www.open-open.com/lib/view/open1423558996951.html>

### 12. 说说你对依赖注入的理解

依赖注入：可以通过这个服务来安全的注入组件到应用程序中，在应用程序部署的时候还可以选择从特定的接口属性进行注入。

参考自：

[用Dagger2在Android中实现依赖注入](https://www.jianshu.com/p/f713dd40e037)

[反射、注解与依赖注入总结](https://www.jianshu.com/p/24820bf3df5c)

### 13. 说一下泛型原理，并举例说明

### 14. Java中String的了解

### 15. String为什么要设计成不可变的？

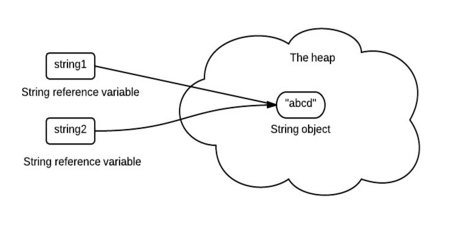
String 是 Java 中一个不可变类，所以他一旦被实例化就无法被修改。为什么要把String设计成不可变的呢？那就从内存、同步和数据结构层面上谈起。

**字符串池**

字符串池是方法区中的一部分特殊存储。当一个字符串被创建的时候，首先会去这个字符串池中查找。如果找到，直接返回对该字符串的引用。

下面的代码只会在堆中创建一个字符串：

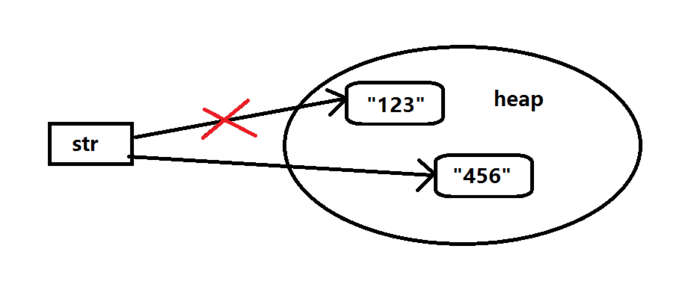
String string1 = "abcd";  
String string2 = "abcd";



**如果字符串可变的话，当两个引用指向同一个字符串时，对其中一个做修改就会影响另外一个。**

String str= "123";  
str = "456";

执行过程如下：



执行第一行代码时，在堆上新建一个对象实例 123，str 是一个指向该实例的引用，**引用包含的仅仅只是实例在堆上的内存地址而已**。执行第二行代码时，仅仅只是改变了 str 这个引用的地址，指向了另外一个实例 456。**给 String 赋值仅仅只是改变了它的引用而已，并不会真正的去改变它本来内存地址上的值**。这样的好处也是显而易见的，最简单的当存在多个 String 的引用指向同一个内存地址时，改变其中一个引用的值并不会其他引用的值造成影响。

那如果我们这样写呢：

String str1 = "123";  
 String str2 = new String("123");  
 System.out.println(str1 == str2);

结果是 false。JVM为了字符串的复用，减少字符串对象的重复创建，特别维护了一个字符串常量池。第一种字面量形式的写法，会直接在字符串常量池中查找是否存在值 123，若存在直接返回这个值的引用，若不存在则创建一个值123的String对象并存入字符串常量池中。而使用new关键字，则会直接才堆上生成一个新的String对象，并不会理会常量池中是否有这个值。所以本质上 str1 和 str2 指向的内存地址是不一样的。

那么，使用 new 关键字生成的 String 对象可以进入字符串常量池吗？答案是肯定的，String 类提供了一个 native 方法 intern() 用来将这个对象加入字符串常量池：

String str1 = "123";  
 String str2 = new String("123");  
 str2=str2.intern();  
 System.out.println(str1 == str2);

结果为 true。str2 调用 intern() 方法后，首先会在字符串常量池中寻找是否存在值为 123 的对象，若存在直接返回该对象的引用，若不存在，加入 str2 并返回。上诉代码中，常量池中已经存在了值为 123 的 str1 对象，则直接返回 str1 的引用地址，使得 str1 和 str2 指向同一个内存地址。

**缓存 Hashcode**

Java 中经常会用到字符串的哈希码。例如，在HashMap中，字符串的不可变能保证其Hashcode永远保持一致，避免了一些不必要的麻烦。这也就意味着每次使用一个字符串的hashcode的时候不用重新计算一次，更加高效。

**安全性**

String被广泛的使用在其他类中充当参数，如果字符串可变，那么类似操作就会可能导致安全问题。可变的字符串也可能导致反射的安全问题，因为它的参数也是字符串。

**不可变对象天生就是线程安全的**

因为不可变对象不能被改变，所以他们可以自由的在多个线程之间共享，不需要做任何同步处理。

总之，String被设计成不可变的主要目的是为了安全和高效。当然，缺点就是要创建多余的对象而并非改变其值。

参考自：

<https://www.jianshu.com/p/b1d62928552d>

<http://www.hollischuang.com/archives/1246>

### 16. Object类的equal和hashCode方法重写，为什么？

### 17. 为什么Java里的匿名内部类只能访问final修饰的外部变量？

因为匿名内部类最终会被编译成一个单独的类，而被该类使用的变量会以构造函数参数的形式传递给该类。如果变量不定义为final的，参数在匿名内部类中可以被修改，进而造成和外部的变量不一致的问题，为了避免这种不一致的情况，规定匿名内部类只能访问final修饰的外部变量。

### 18. Synchronized

synchronized，是Java中用于解决并发情况下数据同步访问的一个很重要的关键字。当我们想要保证一个共享资源在同一个时间只会被一个线程访问到时，我们可以在代码中使用synchronized关键字对类或者对象加锁。

在Java中，synchronized有两种使用方式，同步方法和同步代码块。

对于同步方法，JVM采用ACC\_SYNCHRONIZED标记符来实现同步。对于同步代码块，JVM采用monitorenter、monitorexit 两个指令来实现同步。

**同步方法** 方法级的同步是隐式的，同步方法的常量池中会有一个ACC\_SYNCHRNZED 标志，当某个线程要访问某个方法的时候，会检查是否有 ACC\_SYNCHORIZED ，如果有设置，则需要先获得监视器锁，然后开始执行方法，方法执行之后在释放监视器锁。这时如果其他线程来请求执行方法，会因为无法获得监视器锁而被阻断住。值得注意的是，如果方法在执行过程中发生了异常，并且方法内部并没有处理该异常，那么异常被抛到方法外面之前监视器锁会被自动释放。

**同步代码块** 同步代码块使用monitorenter和monitorexit两个指令实现。可以把执行monitorenter指令理解为加锁，执行monitorexit理解为释放锁。每个对象维护着一个记录被锁次数的计数器，未被锁定的对象的该计数器为0，当一个线程获得锁（执行monitorenter）后，该计数器自增变为1，当同一个线程再次获得该对象的锁的时候，计数器再次自增。当同一个线程释放锁（执行monitorexit指令）的时候，计数器在自减。当计数器为0的时候，锁将被释放，其他线程便可以获得锁。

**synchronized与原子性** 原子性是指一个操作是不可中断的，要全部执行完成，要不就都不执行。

线程是CPU调度的基本单位，CPU有时间片的概念，会根据不同的调度算法进行线程调度。当一个线程获得时间片之后开始执行，在时间片耗尽之后，就会失去CPU使用权。所以在多线程场景下，由于时间片在线程间轮换，就会发生原子性问题。

在Java中，为了保证原子性，提供了两个高级的字节码指令 monitorenter 和 monitorexit 。前面介绍过，这两个字节码指令，在Java中对应的关键字就是 synchronized。

通过 monitorenter 和 monitorexit 指令，可以保证被 synchronized修饰的代码在同一时间只能被一个线程访问，在锁未释放之前，无法被其他线程访问到。因此，在Java中可以使用 synchronized 来保证方法和代码块内的操作是原子性的。

线程一在执行monitorenter指令的时候，会对Monitor进行加锁，加锁后其他线程无法获得锁，除非线程一主动解锁。即使在执行过程中，由于某种原因，比如CPU时间片用完，线程一放弃了CPU，但是，他并没有进行解锁，而由于 synchorized 的锁是可以重入的，下一个时间片还是只能被他自己获取到，还是会继续执行代码，直到所有代码执行完，这就保证了原子性。

**synchroized与可见性** 可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存保存了该线程中是用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之后也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主内存之间进行数据同步进行。所以，就可能出现线程一修改了某个变量的值，但线程二不可见的情况。

前面我们介绍过，被 synchronized 修饰的代码，在开始执行时会加锁，执行完成后会进行解锁。而为了保证可见性，有一条规则是专业的：对一个变量解锁之前，必须先把变量同步到主内存中。这样解锁后，后续线程就可以访问到被修改后的值。

所以，被 synchronized 关键字锁住的对象，其值是具有可见性的。

**synchronized与有序性** 有序性即程序执行的顺序按照代码的先后顺序执行。

除了引入了时间片以外，由于处理器优化和指令重排，CPU还可能对输入代码进行乱序执行，这就可能存在有序性问题。

这里需要注意的是，synchronized 是无法禁止指令重排和处理器优化的，也就是说，synchronized 无法避免上述提到的问题。那么为什么还说synchronized 也提供了有序性保证呢？

这就要把有序性的概念扩展一下了，Java程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有操作都是天然有序的，如果在一个线程中观察另外一个线程，所有的操作都是无序的。

以上这句话也是《深入理解Java虚拟机》中的原句，但是怎么理解呢？这其实和 as-if-serial语义有关。

as-if-serial 语义的意思是指：不管怎么重排序（编译器和处理器为了提高并行度），单线程程序的执行结果都不能被改变，编译器和处理器无论如何优化，都必须遵守 as-if-serial语义。

简单来说，as-if-serial语义保证了单线程中，指令重排是有一定限制的，而只要编译器和处理器都遵守这个语义，那么就可以认为单线程程序是按照顺序执行的，当然，实际上还是有重排，只不过我们无需关心这种重排的干扰。

所以说，由于synchronized修饰的代码，同一时间只能被同一个线程访问，那么也就是单线程执行，所以可以保证其有序性。

**synchronized与锁优化**

无论是ACC\_SYNCHORIZED还是monitorenter、monitorexit都是基于Monitor实现的，在Java虚拟机（HotSpot）中，Monitor是基于C++实现的，由ObjectMonitor实现。

ObjectMonitor类中提供了几个方法，如 enter、exit、wait、notify、notifyAll 等。sychronized 加锁的原理，会调用 objectMonitor的enter方法，解锁的时候会调用 exit 方法。事实上，只有在JDK1.6之前，synchronized的实现才会直接调用ObjectMonitor的enter和exit，这种锁被称之为重量级锁。为什么说这种方式操作锁很重呢？

Java的线程是映射到操作系统原生线程之上的，如果要阻塞或唤醒一个线程就需要操作系统的帮忙，这就是要从用户态转换为核心态，因此状态转换需要花费很多的处理器时间，对于代码简单的同步块，状态转换消耗的时间有可能比用户代码执行的时间还要长，所以说synchroized是java语言中一个重量级的操纵。

所以，在JDK1.6中出现对锁进行了很多的优化，进而出现了轻量级锁、偏向锁、锁消除，适应性自旋锁等等，这些操作都是为了在线程之间更高效的共享数据，解决竞争问题。

### 19. volatile

**volatile用法** volatile通常被比喻成轻量级的synchronized，也是Java并发编程中比较重要的一个关键字，和synchronized不同，volatile是一个变量修饰符，只能用来修饰变量，无法修饰方法以及代码块。

volatile的用法比较简单，只需要在声明一个可能被多线程同时访问的变量时，使用volatile修饰就可以了。

**volatile原理** 为了提高处理器的执行速度，在处理器和内存之间增加了多级缓存来提升，但是由于引入了多级缓存，就存在缓存数据不一致的问题。

但是，对于volatile变量，当对volatile变量进行写操作的时候，JVM会向处理器发送一条Lock前缀的指令，将这个缓存中的变量回写到系统主存中。

但是就算回写内存，如果其他处理器缓存的值还是旧的，在执行计算操作就会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议。

**缓存一致性协议：**

每个处理器通过嗅探在总线上传播的数据来检测自己缓存的信息是不是过期了，当处理器发现自己缓存行对应的内存地址被修改了，就会将当前处理器的缓存行设置为无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

所以，如果一个变量被volatile所修饰的话，在每次数据变化之后，其值都会被强制刷新入主存。而其他处理器的缓存由于遵守了缓存一致性协议，也会把这个变量的值从主存加载到自己的缓存中，这就保证了一个volatile修饰的变量在多个缓存中是可见的。

**volatile与可见性** 可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程也能够立即看到修改的值。

前面在关于volatile原理的时候讲过，Java中的volatile关键字提供了一个功能，那就是被修饰的变量在被修改后可以立即同步到主存中，被其修饰的变量在每次使用之前都是从主内存中刷新。因此，可以使用volatile来保证多线程操作时变量的可见性。

**volatile与有序性** volatile禁止指令重排优化，这就保证了代码的程序会严格按照代码的先后顺序执行，这就保证了有序性。

**volatile与原子性** 在上面介绍synchronized的时候，提到过，为了保证原子性，需要通过字节码指令monitorenter和monitorexit，但是volatile和这两个指令没有任何关系。

**所以，volatile是不能保证原子性的。**

在以下两个场景中可以使用volatile替代synchronized：

1. 运算结果并不依赖变量的当前值，或者能够确保只有单一的线程会修改变量的值
2. 变量不需要与其他状态变量共同参与不变约束

除了以上场景，都需要使用其他方式来保证原子性，如synchronized或者concurrent包。

synchronized可以保证原子性、有序性和可见性，而volatile只能保证有序性和可见性。

### 20. 类加载流程双亲委托机制

**类加载流程**

1. 装载
2. 链接
3. 验证
4. 准备
5. 解析
6. 初始化

**双亲委托机制**

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才会自己去加载。

使用双亲委托模型的好处在于Java类随着它的类加载器一起具备类一种带有优先级的层次关系。例如类java.lang.Object，它存在rt.jar中，无论哪一个类加载器要加载这个类，最终都会委托处于模型最顶端的Bootstrap ClassLoader进行加载，因此Object类在程序的各类加载器环境中都是同一个类。相反，如果没有双亲委派模型，而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，那么系统中将会出现多个不同的Object类，程序将变得混乱。因此，如果开发者尝试编写一个与rt.jar类库中重名的Java类，可以正常编译，但是永远无法被加载运行。

1. 当前ClassLoader首先从自己已经加载的类中查询是否此类已经记载，如果已经加载则可以直接返回已经加载的类。

每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

1. 当前ClassLoader的缓存中没有找到被加载的类的时候，会委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到BootStrap ClassLoader。
2. 当所有的父类加载器都没有加载的时候，再有当前的类加载器加载，并将其放入它自己的缓存中，以便下次加载请求时直接返回。

为什么需要这样的委托机制呢？理解这个问题，我们要引入另外一个关于ClassLoader的概念“命名空间”，它是指要确定某一个类，需要类的全限定名以及加载此类的ClassLoader来共同确定。也就是说，即使两个类的全限定名相同，但是因为不同的ClassLoader加载了此类，那么在JVM中它是不同的类。明白了命名空间以后，我们再来看看委托模型。采用了委托模型以后加大了不同的ClassLoader的交互能力，比如上面说的，我们JDK本身提供的类库，比如HashMap、LinkedList等等，这些类由bootstrap类加载器加载以后，无论你程序中有多少个类加载器，那么这些类其实都是可以共享的，这样就避免了不同的类加载器加载了同样名字的不同类以后造成混乱。

## Java 部分（三）数据结构

### 1. 常用数据结构简介

### 2. 并发集合了解哪些？

### 3. HashMap

**简介**

HashMap 是一散列表，它存储的内容是键值对的映射。它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序确是不确定的。

HashMap最多只允许一条记录的键为null，允许多条记录的值为null。

HashMap使用hash算法进行数据的存储和查询，内部使用一个Entry表示键值对key-value。用Entry的数组保存所有键值对，Entry通过链表的方式链接后续的节点（1.8后会根据链表长度决定是否转换成红黑树），Entry通过计算key的hash值来决定映射到具体的哪个数组（也叫Bucket）中。

HashMap非线程安全，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致，如果需要满足线程安全，可以用Collections的synchronizedMap方法使得HashMap具有线程安全的能力，或者使用ConcurrentHaspMap。

**存储结构** HashMap是数组+链表+红黑树实现的。

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable {  
 private static final long serialVersionUID = 362498820763181265L;  
 /\*\*  
 \* HashMap 的默认初始容量为 16，必须为 2 的 n 次方 (一定是合数)  
 \*/  
 static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4;  
  
 /\*\*  
 \* HashMap 的最大容量为 2 的 30 次幂  
 \*/  
 static final int MAXIMUM\_CAPACITY = 1 << 30;   
  
 /\*\*  
 \* HashMap 的默认负载因子  
 \*/  
 static final float DEFAULT\_LOAD\_FACTOR = 0.75f;  
  
 /\*\*  
 \* 链表转成红黑树的阈值。即在哈希表扩容时，当链表的长度(桶中元素个数)超过这个值的时候，进行链表到红黑树的转变  
 \*/  
 static final int TREEIFY\_THRESHOLD = 8;  
  
 /\*\*  
 \* 红黑树转为链表的阈值。即在哈希表扩容时，如果发现链表长度(桶中元素个数)小于 6，则会由红黑树重新退化为链表  
 \*/  
 static final int UNTREEIFY\_THRESHOLD = 6;  
  
 /\*\*  
 \* HashMap 的最小树形化容量。这个值的意义是：位桶（bin）处的数据要采用红黑树结构进行存储时，整个Table的最小容量（存储方式由链表转成红黑树的容量的最小阈值）  
 \* 当哈希表中的容量大于这个值时，表中的桶才能进行树形化，否则桶内元素太多时会扩容，而不是树形化  
 \* 为了避免进行扩容、树形化选择的冲突，这个值不能小于 4 \* TREEIFY\_THRESHOLD  
 \*/  
 static final int MIN\_TREEIFY\_CAPACITY = 64;  
  
 /\*\*  
 \* Node 是 HashMap 的一个内部类，实现了 Map.Entry 接口，本质是就是一个映射 (键值对)  
 \* Basic hash bin node, used for most entries.  
 \*/  
 static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash; // 用来定位数组索引位置  
 final K key;  
 V value;  
 Node<K,V> next; // 链表的下一个node  
  
 Node(int hash, K key, V value, Node<K,V> next) { ... }  
  
 public final K getKey() { ... }  
 public final V getValue() { ... }  
 public final String toString() { ... }  
 public final int hashCode() { ... }  
 public final V setValue(V newValue) { ... }  
 public final boolean equals(Object o) { ... }  
 }  
  
 /\*\*  
 \* 哈希桶数组，分配的时候，table的长度总是2的幂  
 \*/  
 transient Node<K,V>[] table;  
  
 /\*\*  
 \* Holds cached entrySet(). Note that AbstractMap fields are used  
 \* for keySet() and values().  
 \*/  
 transient Set<Map.Entry<K,V>> entrySet;  
  
 /\*\*  
 \* HashMap 中实际存储的 key-value 键值对数量  
 \*/  
 transient int size;  
  
 /\*\*  
 \* 用来记录 HashMap 内部结构发生变化的次数，主要用于迭代的快速失败机制  
 \*/  
 transient int modCount;  
  
 /\*\*  
 \* HashMap 的门限阀值/扩容阈值，所能容纳的 key-value 键值对极限，当size>=threshold时，就会扩容  
 \* 计算方法：容量capacity \* 负载因子load factor   
 \*/  
 int threshold;  
  
 /\*\*  
 \* HashMap 的负载因子  
 \*/  
 final float loadFactor;  
}  
  
作者：野狗子嗷嗷嗷  
链接：https://www.jianshu.com/p/b40fd341711e  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

Node[] table的初始化长度是16，负载因子默认是0.75，threshold是HashMap所能容纳的最大键值对。threshold=length\*loadFactor，也就是说当HashMap存储的元素数量大于threshold时，HashMap就会进行扩容的操作。

size这个字段其实很好理解，就是HashMap中实际存在的键值对数量。而modCount字段主要用来记录HashMap内部结构发生变化的次数，主要用于迭代的快速失败。

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方，这是一种非常规的设计，常规的设计是把桶的大小设计为素数，相对来说，素数导致冲突的概率要小。HashTable初始化桶大小为11，这就是桶大小设计为素数的应用（HashTable扩容后不能保证还是素数）。HashMap采用这种非常规的设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap定位哈希桶索引位置时，也加入了高位参与运算的过程。

**功能实现**

解决Hash冲突的hash()方法：

HashMap的hash计算时先计算hashCode()，然后进行二次hash。

// 计算二次Hash  
int hash = hash(key.hashCode());  
  
// 通过Hash找数组索引  
int i = hash & (tab.length-1);  
  
static final int hash(Object key) {  
 int h;  
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}

它总是通过h&(table.length-1)来得到该对象的保存位置，而HashMap底层数组的长度总是2的n次方，这样保证了计算得到的索引值总是位于table数组的索引之内。

**put方法**

1. 对key的hashCode做hash，然后计算index
2. 如果没碰撞，就直接放到bucket里，如果碰撞了，以链表的形式存在buckets后
3. 如果碰撞导致链表过长（大于等于TREEIFY\_THRESHOLD=8），就把链表转换为红黑树
4. 如果节点已经存在，就替换，保证key的唯一性
5. 如果bucket满了（超过loadFactor\*currentCapacity），就要resize

具体步骤：

1. 如果table没有使用过的情况，（tab=table）==null||(n=length)==0，则以默认大小进行一次resize
2. 计算key的hash值，然后获取底层table数组的第（n-1）&hash的位置的数组索引tab[i]处的数据，即hash对n取模的位置，依赖的是n为2的次方的这一条件
3. 先检查该bucket第一个元素是否是和插入的key相等（如果是同一个对象则肯定equals）
4. 如果不相等并且是TreeNode的情况，调用TreeNode的put方法
5. 否则循环遍历链表，如果找到相等的key跳出循环否则达到最后一个节点时将新的节点添加到链表最后，当前面找到了相同的key的情况下替换这个节点的value为新的value
6. 如果新增了key-value对，则增加size并且判断是否超过了threshold，如果超过了则需要进行resize扩容

public V put(K key, V value) {  
 // 对key的hashCode()做hash  
 return putVal(hash(key), key, value, false, true);  
}  
  
/\*\*  
 \* Implements Map.put and related methods  
 \*  
 \* @param hash hash for key  
 \* @param key the key  
 \* @param value the value to put  
 \* @param onlyIfAbsent if true, don't change existing value  
 \* @param evict if false, the table is in creation mode.  
 \* @return previous value, or null if none  
 \*/  
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  
 boolean evict) {  
 Node<K,V>[] tab; Node<K,V> p; int n, i;  
 // table为空或者length=0时，以默认大小扩容，n为table的长度   
 if ((tab = table) == null || (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 // 计算index，并对null做处理，table[i]==null  
 if ((p = tab[i = (n - 1) & hash]) == null)  
 // (n-1)&hash 与Java7中indexFor方法的实现相同，若i位置上的值为空，则新建一个Node，table[i]指向该Node。  
 // 直接插入  
 tab[i] = newNode(hash, key, value, null);  
 else {  
 // 若i位置上的值不为空，判断当前位置上的Node p 是否与要插入的key的hash和key相同  
 Node<K,V> e; K k;  
 // 若节点key存在，直接覆盖value  
 if (p.hash == hash &&  
 ((k = p.key) == key || (key != null && key.equals(k))))  
 e = p;  
 // 判断table[i]该链是否是红黑树，如果是红黑树，则直接在树中插入键值对  
 else if (p instanceof TreeNode)  
 // 不同，且当前位置上的的node p已经是TreeNode的实例，则再该树上插入新的node  
 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
 // table[i]该链是普通链表，进行链表的插入操作  
 else {  
 // 在i位置上的链表中找到p.next为null的位置，binCount计算出当前链表的长度，如果继续将冲突的节点插入到该链表中，会使链表的长度大于tree化的阈值，则将链表转换成tree。  
 for (int binCount = 0; ; ++binCount) {  
 // 如果遍历到了最后一个节点，说明没有匹配的key，则创建一个新的节点并添加到最后  
 if ((e = p.next) == null) {  
 p.next = newNode(hash, key, value, null);  
 // 链表长度大于8转换为红黑树进行处理  
 if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st  
 treeifyBin(tab, hash);  
 break;  
 }  
 // 遍历过程中若发现 key 已经存在直接覆盖 value 并跳出循环即可  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 break;  
 p = e;  
 }  
 }  
 // 已经存在该key的情况时，将对应的节点的value设置为新的value  
 if (e != null) { // existing mapping for key  
 V oldValue = e.value;  
 if (!onlyIfAbsent || oldValue == null)  
 e.value = value;  
 afterNodeAccess(e);  
 return oldValue;  
 }  
 }  
 ++modCount;  
 // 插入成功后，判断实际存在的键值对数量 size 是否超多了最大容量 threshold，如果超过，进行扩容  
 if (++size > threshold)  
 resize();  
 afterNodeInsertion(evict);  
 return null;  
}

**get方法** get(key)方法时获取key的hash值，计算hash&(n-1)得到在链表数组中的位置first=tab[hash&(n-1)]，先判断first的key是否是参数key相等，不等就遍历后面的链表找到相同的key值返回对应的value值即可。

public V get(Object key) {  
 Node<K,V> e;  
 return (e = getNode(hash(key), key)) == null ? null : e.value;  
}  
  
// 根据哈希表元素个数与哈希值求模（使用的公式是 (n - 1) &hash）得到 key 所在的桶的头结点，如果头节点恰好是红黑树节点，就调用红黑树节点的 getTreeNode() 方法，否则就遍历链表节点  
final Node<K,V> getNode(int hash, Object key) {  
 Node<K,V>[] tab; Node<K,V> first, e; int n; K k;  
 if ((tab = table) != null && (n = tab.length) > 0 &&  
 (first = tab[(n - 1) & hash]) != null) {  
 if (first.hash == hash && // always check first node  
 ((k = first.key) == key || (key != null && key.equals(k))))  
 return first;  
 if ((e = first.next) != null) {  
 if (first instanceof TreeNode)  
 return ((TreeNode<K,V>)first).getTreeNode(hash, key);  
 do {  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 return e;  
 } while ((e = e.next) != null);  
 }  
 }  
 return null;  
}

**resize方法** 扩容就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组。

具体步骤：

1. 首先计算resize()后的新的capacity和threshold，如果原有的capacity大于零，则将capacity增加一倍，否则设置成默认的capacity
2. 创建新的数组，大小是新的capacity
3. 将旧的数组元素放置到新数组中

final Node<K,V>[] resize() {  
 // 将字段引用copy到局部变量表，这样在之后的使用时可以减少getField指令的调用  
 Node<K,V>[] oldTab = table;  
 // oldCap为原数组的大小或当空时为0  
 int oldCap = (oldTab == null) ? 0 : oldTab.length;  
 int oldThr = threshold;  
 int newCap, newThr = 0;  
 if (oldCap > 0) {  
 if (oldCap >= MAXIMUM\_CAPACITY) {  
 // 如果超过最大容量1>>30，无法再扩充table，只能改变阈值  
 threshold = Integer.MAX\_VALUE;  
 return oldTab;  
 }  
 // 新的数组的大小是旧数组的两倍  
 else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&  
 oldCap >= DEFAULT\_INITIAL\_CAPACITY)  
 // 当旧的的数组大小大于等于默认大小时，threshold也扩大一倍  
 newThr = oldThr << 1;  
 }  
 else if (oldThr > 0) // initial capacity was placed in threshold  
 newCap = oldThr;  
 else { // zero initial threshold signifies using defaults  
 // 初始化操作  
 newCap = DEFAULT\_INITIAL\_CAPACITY;  
 newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);  
 }  
 if (newThr == 0) {  
 float ft = (float)newCap \* loadFactor;  
 newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?  
 (int)ft : Integer.MAX\_VALUE);  
 }  
 threshold = newThr;  
 @SuppressWarnings({"rawtypes","unchecked"})  
 // 创建容量为newCap的newTab，并将oldTab中的Node迁移过来，这里需要考虑链表和tree两种情况。  
 Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  
 table = newTab;  
 // 将原数组中的数组复制到新数组中  
 if (oldTab != null) {  
 for (int j = 0; j < oldCap; ++j) {  
 Node<K,V> e;  
 if ((e = oldTab[j]) != null) {  
 oldTab[j] = null;  
 if (e.next == null)  
 // 如果e是该bucket唯一的一个元素，则直接赋值到新数组中  
 newTab[e.hash & (newCap - 1)] = e;  
 else if (e instanceof TreeNode)  
 // split方法会将树分割为lower 和upper tree两个树，如果子树的节点数小于了UNTREEIFY\_THRESHOLD阈值，则将树untreeify，将节点都存放在newTab中。  
 // TreeNode的情况则使用TreeNode中的split方法将这个树分成两个小树  
 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  
 else { // preserve order 保持顺序  
 // 否则则创建两个链表用来存放要放的数据，hash值&oldCap为0的(即oldCap的1的位置的和hash值的同样的位置都是1，同样是基于capacity是2的次方这一前提)为low链表，反之为high链表, 通过这种方式将旧的数据分到两个链表中再放到各自对应余数的位置  
 Node<K,V> loHead = null, loTail = null;  
 Node<K,V> hiHead = null, hiTail = null;  
 Node<K,V> next;  
 do {  
 next = e.next;  
 // 按照e.hash值区分放在loTail后还是hiTail后  
 if ((e.hash & oldCap) == 0) {  
 // 运算结果为0的元素，用lo记录并连接成新的链表  
 if (loTail == null)  
 loHead = e;  
 else  
 loTail.next = e;  
 loTail = e;  
 }  
 else {  
 // 运算结果不为0的数据，用li记录  
 if (hiTail == null)  
 hiHead = e;  
 else  
 hiTail.next = e;  
 hiTail = e;  
 }  
 } while ((e = next) != null);  
 // 处理完之后放到新数组中  
 if (loTail != null) {  
 loTail.next = null;  
 // lo仍然放在“原处”，这个“原处”是根据新的hash值算出来的  
 newTab[j] = loHead;  
 }  
 if (hiTail != null) {  
 hiTail.next = null;  
 // li放在j+oldCap位置  
 newTab[j + oldCap] = hiHead;  
 }  
 }  
 }  
 }  
 }  
 return newTab;  
}

**size方法** HashMap的大小很简单，不是实时计算的，而是每次新增加Entry的时候，size就递增。删除的时候就递减，空间换时间的做法，因为它不是线程安全的，完全可以这么做，效率高。

**面试问题**

**1. 构造函数中initialCapacity与loadFactor两个参数** ​ HashMap(int initialCapacity,float loadFactor)：构造一个指定容量和负载因子的空HashMap。

这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中桶的数量，初始容量是创建哈希表时的容量，负载因子是哈希表在其容量增加之前可以达到多满的一种尺度，它衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的填充程度越高，反之越小。

**2. size为什么必须是2的整数次幂** ​ 这是为了服务key映射到index的hash算法的，公式 index=hashcode(key)&(length-1)。HashMap中数组的size必须是2的幂，是为了将key的hash值均匀的分布在数组的索引上。HashMap中使用indexFor方法来计算key所在的数组的索引，实现逻辑为key的hash值与数组长度值减一进行与运算，代码如下：

static int indexFor(int h, int length) {  
 return h & (length - 1);  
 }

**3. HashMap的key为什么一般用字符串比较多，能用其他对象，或者自定义的对象嘛？** ​ 能用其他对象，必须是不可变的，但是自实现的类必须重写equals()和hashCode()方法，否则会调用默认的Object类的对应方法。

**4. HashMap的key和value都能为null嘛？如果key为null，那么它是怎么样查找值的？** ​ 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，由putForNullKey()实现，记住，key为null的键值对永远都放在以table[0]为头节点的链表中。

**5. 使用HashMap时一般使用什么类型的元素作为Key？** ​ 一般是String、Integer，这些类是不可变的，并且这些类已经规范的复写了hashCode以及equals方法，作为不可变类天生是线程安全的，而且可以很好的优化比如可以缓存hash值，避免重复计算等等。

**6. HashTable和HashMap的区别有哪些？** ​ 都实现了Map接口，主要区别在于：线程安全性，同步以及性能。

* HashMap是非线程安全的，效率肯定高于线程安全的HashTable
* HashMap允许null作为一个entry的key或者value，而HashTable不允许
* HashMap把HashTable的contains方法去掉了，改成了containsVaule和containsKey
* HashTable和HashMap扩容的方法不一样，HashTable中的hash数组默认大小是11，扩容方式是old x 2+1，而HashMap中hash数组的默认大小是16，而且一定是2的指数，扩容时old x 2

## Java 部分（四）线程、多线程和线程池

### 1. 开启线程的三种方式？

#### 继承Thread类创建线程类

1. 定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run方法称为执行体。
2. 创建Thread子类的实例，即创建了线程对象。
3. 调用线程对象的start方法来启动该线程。

public class FirstThreadTest extends Thread {  
 int i = 0;  
  
 //重写run方法，run方法的方法体就是现场执行体  
 public void run() {  
 for (; i < 100; i++) {  
 System.out.println(getName() + " " + i);  
  
 }  
 }  
  
 public static void main(String[] args) {  
 for (int i = 0; i < 100; i++) {  
 System.out.println(Thread.currentThread().getName() + " : " + i);  
 if (i == 20) {  
 new FirstThreadTest().start();  
 new FirstThreadTest().start();  
 }  
 }  
 }  
  
}

#### 通过Runnable接口创建线程类

1. 定义runnable接口的实现类，并重写该接口的run方法，该run方法的方法体同样是该线程的线程执行体。
2. 创建Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。
3. 调用线程对象的start方法来开启该线程。

public class RunnableThreadTest implements Runnable {  
 private int i;  
  
 public void run() {  
 for (i = 0; i < 100; i++) {  
 System.out.println(Thread.currentThread().getName() + " " + i);  
 }  
 }  
  
 public static void main(String[] args) {  
 for (int i = 0; i < 100; i++) {  
 System.out.println(Thread.currentThread().getName() + " " + i);  
 if (i == 20) {  
 RunnableThreadTest rtt = new RunnableThreadTest();  
 new Thread(rtt, "新线程1").start();  
 new Thread(rtt, "新线程2").start();  
 }  
 }  
  
 }  
  
}

#### 通过Callable和Future创建线程

1. 创建Callable接口的实现类，并实现call方法，该call方法将作为线程执行体，并且有返回值。
2. 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call方法的返回值。
3. 使用FutureTask对象作为Thread对象的target创建并启动新线程。
4. 调用FutureTask对象的get方法来获取子线程执行结束后的返回值，调用get方法会阻塞线程。

public class CallableThreadTest implements Callable<Integer> {  
  
 public static void main(String[] args) {  
 CallableThreadTest ctt = new CallableThreadTest();  
 FutureTask<Integer> ft = new FutureTask<>(ctt);  
 for (int i = 0; i < 100; i++) {  
 System.out.println(Thread.currentThread().getName() + " 的循环变量i的值" + i);  
 if (i == 20) {  
 new Thread(ft, "有返回值的线程").start();  
 }  
 }  
 try {  
 System.out.println("子线程的返回值：" + ft.get());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Override  
 public Integer call() throws Exception {  
 int i = 0;  
 for (; i < 100; i++) {  
 System.out.println(Thread.currentThread().getName() + " " + i);  
 }  
 return i;  
 }  
}

#### 创建线程的三种方式的对比

**采用实现Runnable、Callable接口的方式创建多线程时：**

优势：

线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好的体现面向对象的思想。

劣势：

稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

**使用继承Thread类的方式创建多线程时：**

优势：

编写简单，如果需要访问当前线程，则不需使用Thread.currentThread方法，直接使用this即可获得当前线程。

劣势：

线程类已经继承了Thread类，所以不能在继承其他父类。

### 2. 线程和进程的区别？

### 3. 为什么要有线程，而不是仅仅用进程？

线程增加进程的并发度，线程能更有效的利用多处理器和多内核。

### 4. run() 和 start() 方法的区别？

主要区别在于当程序调用start方法一个新线程将会被创建，并且在run方法中的代码将会在新线程上运行，然而在你直接调用run方法的时候，程序并不会创建新线程，run方法内部的代码将在当前线程上运行。  
  
另外一个区别在于，一旦一个线程被启动，你不能重复调用该Thread对象的start方法，调用已经启动线程的start方法将会报IIIegalStateException异常，而重复调用run方法是没有问题的。

### 5. 如何控制某个方法允许并发访问线程的个数？

Semaphore semaphore = new Semaphore(5);//线程run中只有5个线程可并发访问  
semaphore.acquire();//申请一个信号  
semaphore.release();//释放一个信号

### 6. 在 Java 中 wait() 和 sleep() 方法的区别

### 7. 谈谈 wait/notify 关键字的理解

都是本地final方法，调用之前持有对象锁。  
  
wait，线程进入挂起状态，释放对象锁。  
  
notify，通知唤醒wait队列中的线程（某个）

### 8. 什么导致线程阻塞？

Thread.sleep t.join 等待输入

### 9. 线程如何关闭？

### 10. 讲一下 Java 中的同步的方法

### 11. 数据一致性如何保证？

### 12. 如何保证线程安全？

### 13. 如何实现线程同步？

### 14. 两个进程同时要求读写，能不能实现？如何防止进程同步？

### 15. 线程间操作 List

### 16. Java 中对象的生命周期

### 17. 说说线程池

**线程池的好处**

1. 线程池的重用

线程的创建和销毁的开销是巨大的，而通过线程池的重用大大减少了这些不必要的开销，当然既然少了那么多消耗内存的开销，其线程执行速度也是突飞猛进的提升。

1. 控制线程池的并发数

控制线程池的并发数可以有效地避免大量的线程争夺CPU资源而造成阻塞。

1. 线程池可以对线程进行管理

线程池可以提供定时、定期、单线程、并发数控制等功能。

**线程池的详解**

1. ThreadPoolExecutor

```java

public ThreadPoolExecutor(int corePoolSize,  
int maximumPoolSize,  
long keepAliveTime,  
TimeUnit unit,  
BlockingQueue workQueue,  
ThreadFactory threadFactory,  
RejectedExecutionHandler handler) ```

这里是七个参数（更多的是用五个参数的构造方法）。

**corePoolSize**：线程池中核心线程的数量。

**maxinumPoolSize**：线程池中最大的线程数量。

**keepAliveTime**：非核心线程的超时时长，当系统中非核心线程闲置时间超过keepAliveTime之后，则会被回收。如果ThreadPoolExecutor的allowCoreThreadTimeOut属性设置为true，则该参数也表示核心线程的超时时长。

**unit**：第三个参数的单位，有毫秒、秒、分等等。

**workQueue**：线程池中的任务队列，该队列主要用来存储已经被提交但尚未执行的任务。存储在这里的任务是由ThreadPoolExecutor的execute方法提交来的。

threadFactory：为线程池提供创建新线程的功能，这个我们一般使用默认即可。

**handler**：拒绝策略，当线程无法执行新任务（一般是由于线程池中的线程数量已经达到最大数或者线程池关闭导致的）。默认情况下，当线程池无法处理新线程时，会抛出一个RejectedExecutionException。

有以下：

* 当currentSize < corePoolSize 时，直接启动一个核心线程并执行任务
* 当currentSize >= corePoolSize，并且 workQueue 未满时，添加进来的任务会被安排到workQueue中等待执行
* 当workQueue已满，但是currentSize < maxinumPoolSize 时，会立即开启一个非核心线程来执行任务
* 当 currentSize >= corePoolSize、workQueue已满，并且currentSize > maxinumPoolSize 时，调用handler默认会抛出异常

1. 其他线程池
2. FixedThreadPool

* 有固定数量线程的线程池，其中corePoolSize = maxinumPoolSize，且keepAliveTime为0，适合线程稳定的场所。

1. SingleThreadPool

* corePoolSize = maxinumPoolSzie = 1 且 keepAliveTime 为0，适合线程同步操作的场所。

1. CachedThreadPool

* corePoolSize = 0，maximunPoolSize = Integer.MAX\_VALUE（2的32次方-1）。

1. ScheduledThreadPool

* 是一个具有定时定期执行任务功能的线程池。

### 18. 并发编程面临的挑战

1. 上下文切换
2. 死锁
3. 资源限制

参考：[并发编程面临的挑战](https://www.jianshu.com/p/f3e69850addf)

## Android 部分（一）基础知识点

### 1. 四大组件是什么？

四大组件：Activity、Service、BroadcastReceiver、ContentProvider。

### 2. 四大组件的生命周期和简单用法

### **Activity：**

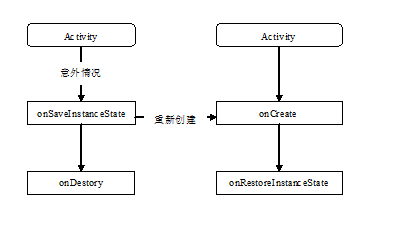
典型的生命周期好像没什么可说的，主要说一下特殊情况下的生命周期。

1. 横竖屏的切换

在横竖屏切换的过程中，会发生Activity被销毁并被重建的过程。

在了解这种情况下的生命周期，首先应该了解这两个回调：onSaveInstance State 和 onRestoreInstance State。

在Activity由于异常情况下终止时，系统会调用 onSaveInstanceState 来保存当前 Activity 的状态。这个方法的调用是在onStop之前，它和onPause没有既定的时序关系，该方法只有在Activity被异常终止的情况下调用。当异常终止的Activity被重建之后，系统会调用onRestoreInstanceState，并且把Activity销毁时onSaveInstanceState方法所保存的Bundle对象参数同时传递给onRestoreInstanceState和onCreate方法。因为，可以通过onRestoreInstanceState方法来恢复Activity的状态，该方法的调用时机是在onStart之后。其中，onCreate和onRestoreInstanceState方法来恢复Activity状态的区别：onRestoreInstanceState回调则表明其中Bundle对象非空，不用加非空判断，而onCreate需要非空判断，建议使用onRestoreInstanceState。



横竖屏切换的生命周期：onPause() --> onSaveInstanceState() --> onStop() --> onDestory() --> onCreate() --> onStart() --> onRestoreInstanceState() --> onResume()   
  
可以通过在AndroidManifest文件的Activity中指定如下属性：  
  
android:configChanges = "orientation| screenSize"  
  
来避免横竖屏切换时，Activity的销毁和重建，而是回调了下面的方法：

@Override  
 public void onConfigurationChanged(Configuration newConfig) {  
 super.onConfigurationChanged(newConfig);  
 }

1. 资源内存不足导致优先级低的Activity被杀死

优先级：前台Activity > 可见但非前台Activity > 后台Activity

**启动模式：**

Android 提供了四种Activity启动方式：

标准模式：standard

栈顶复用模式：singleTop

栈内复用模式：singleTask

单例模式：singleInstance

1. 标准模式 standard

每启动一次Activity，就会创建一个新的Activity实例并置于栈顶。谁启动了这个Activity，那么这个Activity就运行在启动它的那个Activity所在的栈中。

特殊情况下，如果在Service或Application中启动一个Activity，其并没有所谓的任务栈，可以使用标记位Flag来解决。解决办法：为待启动的Activity指定FLAG\_ACTIVITY\_NEW\_TASK标记位，创建一个新栈。

1. 栈顶复用模式 singleTop

如果需要新建的Activity位于任务栈栈顶，那么此Activity的实例就不会重建，而是复用栈顶的实例。并回调：

java @Override protected void onNewIntent(Intent intent) { super.onNewIntent(intent); }

由于不会重建一个Activity实例，则不会回调其他生命周期方法。

应用场景：在通知栏点击收到的通知，然后需要启动一个Activity，这个Activity就可以用singleTop，否则每次点击都会新建一个Activity。

1. 栈内复用模式 singleTask

该模式是一种单例模式，即一个栈内只有一个该Activity实例。该模式，可以通过在AndroidManifest文件的Activity中指定该Activity需要加载到哪个栈中，即singleTask的Activity可以指定想要加载的目标栈。singleTask和taskAffinity配合使用，指定开启的Activity加入到哪个栈中。

xml <activity android:name=".Activity1" android:launchMode="singleTask" android:taskAffinity="com.lvr.task" android:label="@string/app\_name"> </activity>

关于taskAffinity的值：每个Activity都有taskAffinify属性，这个属性指出了它希望进入的Task。如果一个Activity没有显式的指明该Activity的taskAffinity，那么它的这个属性就等于Application指明的taskAffinity，如果Application也没有指明，那么该taskAffinity的值就等于包名。

执行逻辑：

在这种模式下，如果Activity指定的栈不存在，则创建一个栈，并把创建的Activity压入栈内。如果Activity指定的栈存在，如果其中没有该Activity实例，则会创建Activity并压入栈顶，如果其中有该Activity实例，则把该Activity实例之上的Activity杀死清除出战，重用并让该Activity实例处在栈顶，然后调用onNewIntent()方法。

应用场景：

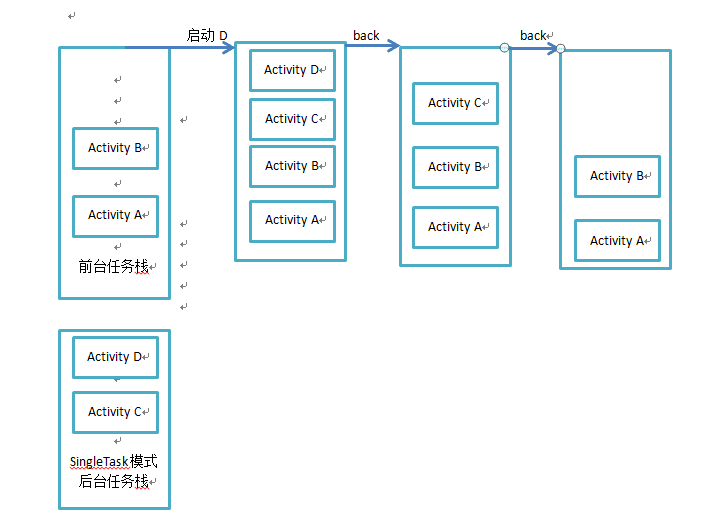
在大多数App的主页，对于大部分应用，当我们在主界面点击返回按钮都是退出应用，那么当我们第一次进入主界面之后，主界面位于栈底，以后不管我们打开了多少个Activity，只要我们再次回到主界面，都应该使用将主界面Activity上所有的Activity移除的方式来让主界面Activity处于栈顶，而不是往栈顶新加一个主界面Activity的实例，通过这种方式能够保证退出应用时所有的Activity都能被销毁。

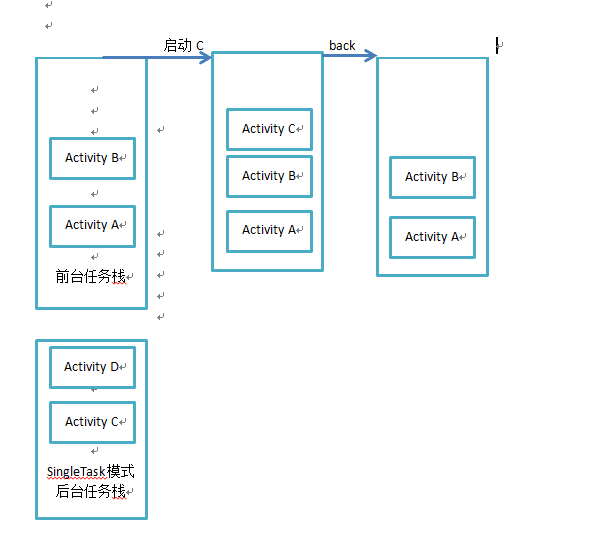
1. 单例模式 singleInstance

作为栈内复用的加强版，打开该Activity时，直接创建一个新的任务栈，并创建该Activity实例放入栈中。一旦该模式的Activity实例已经存在于某个栈中，任何应用在激活该Activity时都会重用该栈中的实例。

应用场景：呼叫来电界面

**特殊情况：前台栈和后台栈的交互**





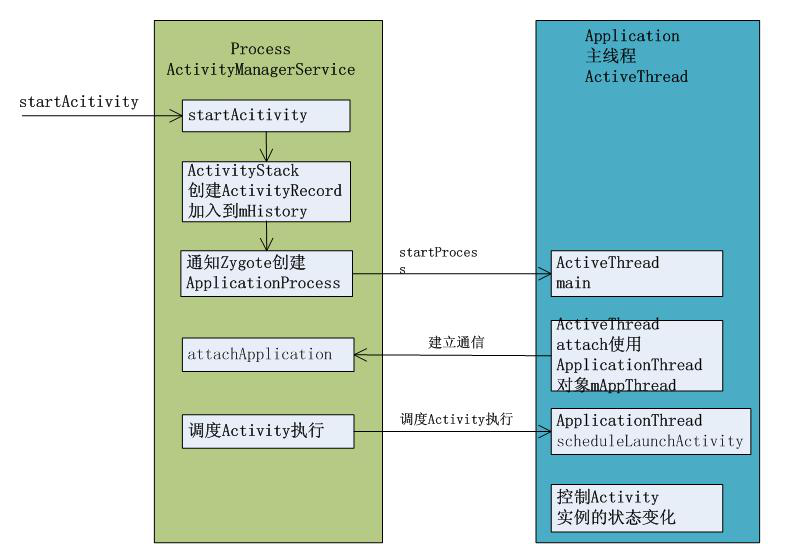
调用SingleTask模式的后台任务栈的Activity，会把整个栈的Activity压入当前栈的栈顶。singleTask会具有clearTop特性，会把之上的栈内Activity清除。

**Activity的Flags：**

Activity的Flags很多，这里介绍集中常用的，用于设定Activity的启动模式，可以在启动Activity时，通过Intent.addFlags()方法设置。

1. FLAG\_ACTIVITY\_NEW\_TASK 即 singleTask
2. FLAG\_ACTIVITY\_SINGLE\_TOP 即 singleTop
3. FLAG\_ACTIVITY\_CLEAR\_TOP 当他启动时，在同一个任务栈中所有位于它之上的Activity都要出栈。如果和singleTask模式一起出现，若被启动的Activity已经存在栈中，则清除其之上的Activity，并调用该Activity的onNewIntent方法。如果被启动的Activity采用standard模式，那么该Activity连同之上的所有Activity出栈，然后创建新的Activity实例并压入栈中。

**Activity的启动过程**



**应用启动过程**

1. Launcher通过Binder进程间通信机制通知AMS，它要启动一个Activity
2. AMS通过Binder进程间通信机制通知Launcher进入Paused状态
3. Launcher通过Binder进程间通信机制通知AMS，它已经准备就绪进入Paused状态，于是AMS就创建一个新的线程，用来启动一个ActivityThread实例，即将要启动的Activity就是在这个ActivityThread实例中运行
4. ActivityThread通过Binder进程间通信机制将一个ApplicationThread类型的Binder对象传递给AMS，以便以后AMS能够通过这个Binder对象和它进行通信
5. AMS通过Binde进程间通信机制通知ActivityThread，现在一切准备就绪，它可以真正执行Activity的启动操作了

### Service

是Android中实现程序后台运行的解决方案，它非常适用于去执行那些不需要和用户交互而且还要长期运行的任务。Service默认并不会运行在子线程中，它也不运行在一个独立的进程中，它同样执行在UI线程中，因此，不要在Service中执行耗时的操作，因此，不要在Service中执行耗时的操作，除非你在Service中创建了子线程来完成耗时操作。

#### Service种类划分

按运行地点分类：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类别 | 区别 | 优点 | 缺点 | 应用 |
| 本地服务(Local Service) | 该服务依附在主进程中 | 服务依附于主进程而不是独立的进程，这样在一定程度上节约了资源，因为Local服务是在同一个进程，所以不需要IPC和AIDL。相应bindService会方便很多。 | 主进程被kill后，服务便会终止 | 如：音乐播放等不需要常驻的服务 |
| 远程服务(Remote Service) |  | 服务为独立的进程，对应进程名格式为所在包名加上你指定的 android:process 字符串。由于是独立的进程，因此在Activity所在进程被kill的时候，该服务依然在运行，不受其他进程影响，有利于为多个进程提供服务具有较高的灵活性。 | 该服务是独立的进程，会占用一定的资源，并且使用AIDL进行IPC稍麻烦 | 一些提供系统服务的Service，这种Service是常住的 |

安运行类型分类：

|  |  |  |
| --- | --- | --- |
| 类别 | 区别 | 应用 |
| 前台服务 | 会在通知栏显示onGoing的Notification | 当服务被终止的时候，通知栏的Notification也会消失，这样对于用户有一定的通知作用。常见的如音乐播放服务 |
| 后台服务 | 默认的服务即为后台服务，即不会在通知一栏显示onGoing的Notification | 当服务被终止的时候，用户是看不到效果的。某些不需要运行或者终止提供的服务，如天气更新、日期同步等等 |

按使用方式分类：

|  |  |
| --- | --- |
| 类别 | 区别 |
| startService启动的服务 | 主要用于启动一个服务执行后台任务，不进行通信，停止服务使用stopService |
| bindService启动的服务 | 启动的服务要进行通信，停止服务使用unbindService |
| 同时使用startService、bindService启动的服务 | 停止服务应同时使用stopService与unbindService |

#### Service生命周期

startService() --> onCreate() --> onStartCommand() --> Service running --> onDestory()

bindService() --> onCreate() --> onBind() --> Service running --> onUnbind() --> onDestory()

**onCreate()：**

系统在Service第一次创建时执行此方法，来执行**只运行一次**的初始化工作，如果service已经运行，这个方法不会调用。

**onStartCommand()：**

每次客户端调用startService()方法启动该Service都会回调该方法(**多次调用**)，一旦这个方法执行，service就启动并且在后台长期运行，通过调用stopSelf()或stopService()来停止服务。

**onBind()：**

当组件调用bindService()想要绑定到service时，系统调用此方法(**一次调用**)，一旦绑定后，下次在调用bindService()不会回调该方法。在你的实现中，你必须提供一个返回一个IBinder来使客户端能够使用它与service通讯，你必须总是实现这个方法，但是如果你不允许绑定，那么你应返回null

**onUnbind()：**

当前组件调用unbindService()，想要解除与service的绑定时系统调用此方法(**一次调用**，一旦解除绑定后，下次再调用unbindService()会抛异常)

**onDestory()：**

系统在service不在被使用并且要销毁的时候调用此方法（**一次调用**）。service应在此方法中释放资源，比如线程，已注册的监听器、接收器等等。

#### 三种情况下Service的生命周期

1. startService / stopService

生命周期：onCreate --> onStartCommand --> onDestory

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，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，还有一种方法可以关闭服务，在设置中，通过应用 --> 找到自己应用 --> 停止。

注意：

第一次startService会触发onCreate和onStartCommand，以后在服务运行过程中，每次startService都只会触发onStartCommand

不论startService多少次，stopService一次就会停止服务

1. bindService / unbindService

生命周期：onCreate --> onBind --> onUnbind --> onDestory

如果一个Service在某个Activity中被调用bindService方法启动，不论bindService被调用几次，Service的onCreate方法只会执行一次，同时onStartCommand方法始终不会调用。

当建立连接后，Service会一直运行，除非调用unbindService来解除绑定、断开连接或调用该Service的Context不存在了（如Activity被finish --- **即通过bindService启动的Service的生命周期依附于启动它的Context**），系统会在这时候自动停止该Service。

注意：

第一次bindService会触发onCreate和inBind，以后在服务运行过程中，每次bindService都不会触发任何回调

1. 混合型

当一个Service再被启动(startService)的同时又被绑定(bindService)，该Service将会一直在后台运行，不管调用几次，onCreate方法始终只会调用一次，onStartCommand的调用次数与startService调用的次数一致（使用bindService方法不会调用onStartCommand）。同时，调用unBindService将不会停止Service，必须调用stopService或Service自身的stopSelf来停止服务。

#### 三种情况下的应用场景

如果你只是想启动一个后台服务长期进行某项任务，那么使用startService便可以了。

如果你想与正在运行的Service取的联系，那么有两种方法，一种是使用broadcast，另外是使用bindService。前者的缺点是如果交流较为频繁，容易造成性能上的问题，并且BroadcastReceiver本身执行代码的时间是很短的（也许执行到一半，后面的代码便不会执行），而后者则没有这些问题，因此我们肯定选择使用bindService（这个时候便同时使用了startService和bindService了，这在Activity中更新Service的某些运行状态是相当有用的）

如果你的服务只是公开了一个远程接口，供连接上的客户端（Android的Service是C/S架构）远程调用执行方法。这个时候你可以不让服务一开始就运行，而只用bindService，这样在第一次bindService的时候才会创建服务的实例运行它，这会节约很多系统资源，特别是如果你的服务是Remote Service，那么该效果会越明显。

### BroadcastReceiver

广播接收器

作用：用于监听 / 接收 应用发出的广播消息，并作出响应

应用场景：

1. 不同组件之间的通信（包括应用内 / 不同应用之间）
2. 与Android系统在特定情况下的通信，如当电话呼入时，网络可用时
3. 多线程通信

#### 实现原理

* 使用了观察者模式：基于消息的发布/订阅事件模型。
* 模型中有三个角色：消息订阅者（广播接收者）、消息发布者（广播发布者）和消息中心（AMS，即Activity Manager Service）

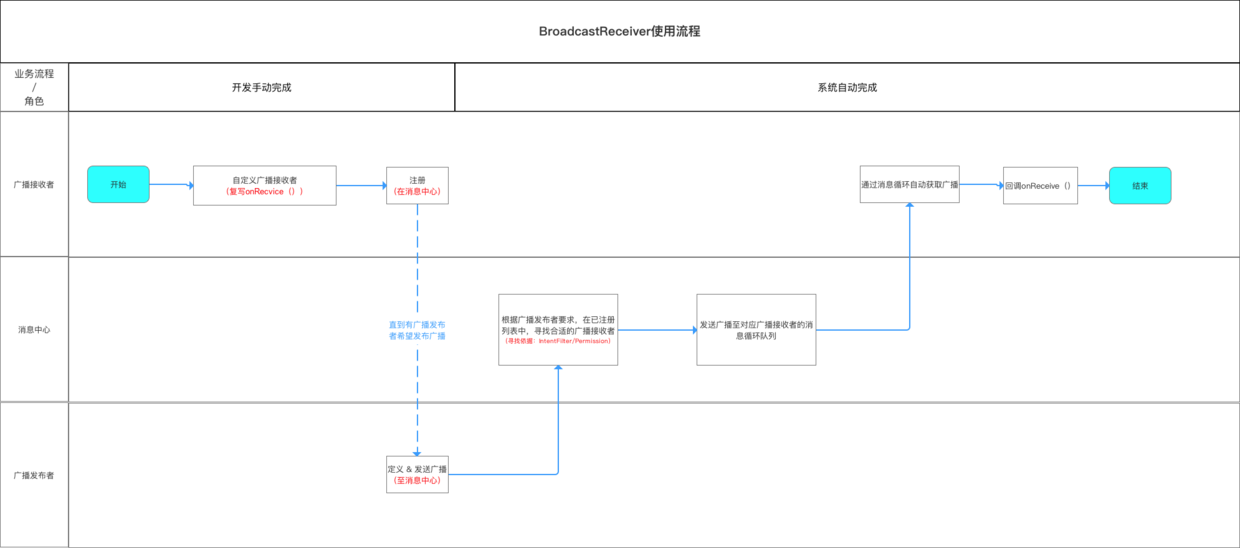


* 原理描述

1. 广播接收者通过Binder机制在AMS注册
2. 广播发送者通过Binder机制向AMS发送广播
3. AMS根据广播发送者要求，在已注册列表中，寻找合适的广播接收者，寻找依据：IntentFilter / Permission
4. AMS将广播发送到合适的广播接收者相应的消息循环队列
5. 广播接收者通过消息循环拿到此广播，并回调onReceive()

注意：广播发送者和广播接收者的执行是异步的，发出去的广播不会关心有没有接收者接收，也不确定接收者何时能接受到。

#### 具体使用



#### 自定义广播接收者BroadcastReceiver

* 继承至BroadcastReceiver基类
* 重写onReceiver()方法

广播接收器收到相应广播后，会自动回调onReceiver()方法；一般情况下，onReceiver方法会涉及与其他组件之间的交互，如发送Notification、启动service等等；默认情况下，广播接收器运行在UI线程，因此onReceiver方法不能执行耗时操作，否则可能ANR

#### 广播接收器注册

注册方式分为两种：静态注册和动态注册。

**静态注册：**

在AndroidManifest.xml里通过标签声明：

<receiver  
 android:enabled=["true" | "false"]  
 //此broadcastReceiver能否接收其他App的发出的广播  
 //默认值是由receiver中有无intent-filter决定的：如果有intent-filter，默认值为true，否则为false  
 android:exported=["true" | "false"]  
 android:icon="drawable resource"  
 android:label="string resource"  
 //继承BroadcastReceiver子类的类名  
 android:name=".mBroadcastReceiver"  
 //具有相应权限的广播发送者发送的广播才能被此BroadcastReceiver所接收；  
 android:permission="string"  
 //BroadcastReceiver运行所处的进程  
 //默认为app的进程，可以指定独立的进程  
 //注：Android四大基本组件都可以通过此属性指定自己的独立进程  
 android:process="string" >  
  
 //用于指定此广播接收器将接收的广播类型  
 //本示例中给出的是用于接收网络状态改变时发出的广播  
 <intent-filter>  
 <action android:name="android.net.conn.CONNECTIVITY\_CHANGE" />  
 </intent-filter>  
</receiver>

**动态注册：**

在代码中通过调用Context的registerReceiver()方法进行动态注册BroadcastReceiver：

@Override  
protected void onResume() {  
 super.onResume();  
  
 //实例化BroadcastReceiver子类 & IntentFilter  
 mBroadcastReceiver mBroadcastReceiver = new mBroadcastReceiver();  
 IntentFilter intentFilter = new IntentFilter();  
  
 //设置接收广播的类型  
 intentFilter.addAction(android.net.conn.CONNECTIVITY\_CHANGE);  
  
 //调用Context的registerReceiver（）方法进行动态注册  
 registerReceiver(mBroadcastReceiver, intentFilter);  
}  
  
  
//注册广播后，要在相应位置记得销毁广播  
//即在onPause() 中unregisterReceiver(mBroadcastReceiver)  
//当此Activity实例化时，会动态将MyBroadcastReceiver注册到系统中  
//当此Activity销毁时，动态注册的MyBroadcastReceiver将不再接收到相应的广播。  
@Override  
protected void onPause() {  
 super.onPause();  
 //销毁在onResume()方法中的广播  
 unregisterReceiver(mBroadcastReceiver);  
}

注意：

动态广播最好在Activity的onResume()注册，onPause()注销，否则会导致内存泄漏，当然，重复注册和重复注销也不允许。

**两种注册方式的区别：**

|  |  |  |
| --- | --- | --- |
| 注册方式 | 特点 | 应用场景 |
| 静态注册（常驻广播） | 常驻，不受任何组件的生命周期影响（应用程序关闭后，如果有信息广播来，程序依旧会被系统调用）缺点：耗电、占内存 | 需要时刻监听广播 |
| 动态注册（非常驻广播） | 非常驻，灵活，跟随组件的生命周期变化（组件结束 = 广播结束，在组件结束前，必须移除广播接收器） | 需要特定时刻监听广播 |

#### 广播发送者向AMS发送广播

##### 广播的发送

* 广播是用意图（Intent）标识
* 定义广播的本质：定义广播所具备的意图（Intent）
* 广播发送：广播发送者将此广播的意图通过sendBroadcast()方法发送出去

##### 广播的类型

广播的类型主要分为5类：

* 普通广播（Normal Broadcast）
* 系统广播（System Broadcast）
* 有序广播（Ordered Broadcast）
* 粘性广播（Sticky Broadcast）
* App应用内广播（Local Broadcast）

**普通广播：**

即开发者自身定义Intent的广播（最常用），发送广播使用如下：

Intent intent = new Intent();  
//对应BroadcastReceiver中intentFilter的action  
intent.setAction(BROADCAST\_ACTION);  
//发送广播  
sendBroadcast(intent);

* 若被注册了的广播接收者中注册时IntentFilter的action与上述匹配，则会接收此广播（即进行回调onReceiver()），如下mBroadcastReceiver则会接收上述广播：

xml <receiver //此广播接收者类是mBroadcastReceiver android:name=".mBroadcastReceiver" > //用于接收网络状态改变时发出的广播 <intent-filter> <action android:name="BROADCAST\_ACTION" /> </intent-filter> </receiver>

* 若发送广播有相应权限，那么广播接收者也需要相应权限

##### 系统广播

* Android中内置了很多系统广播：只要涉及到手机的基本操作（如开关机、网络状态变化、拍照等等），都会发出相应的广播
* 每个广播都有特定的IntentFilter（包括具体的action）Android系统广播action如下：

|  |  |
| --- | --- |
| 系统操作 | action |
| 监听网络变化 | android.net.conn.CONNECTIVITY\_CHANGE |
| 关闭或打开飞行模式 | Intent.ACTION\_AIRPLANE\_MODE\_CHANGED |
| 充电时或电量发生变化 | Intent.ACTION\_BATTERY\_CHANGED |
| 电池电量低 | Intent.ACTION\_BATTERY\_LOW |
| 电池电量充足（即从电量低变化到饱满时会发出广播） | Intent.ACTION\_BATTERY\_OKAY |
| 系统启动完成后（仅广播一次） | Intent.ACTION\_BOOT\_COMPLETED |
| 按下照相时的拍照按键（硬件按键）时 | Intent.ACTION\_CAMERA\_BUTTON |
| 屏幕锁屏 | Intent.ACTION\_CLOSE\_SYSTEM\_DIALOGS |
| 设备当前设置被改变时（设备语言、设备方向等） | Intent.ACTION\_CONFIGURATION\_CHANGED |
| 插入耳机时 | Intent.ACTION\_HEADSET\_PLUG |
| 未正确移除SD卡但已取出来时（正确移除方法：设置SD卡和设备内存--卸载SD卡） | Intent.ACTION\_MEDIA\_BAD\_REMOVAL |
| 插入外部存储装置（如SD卡） | Intent.ACTION\_MEDIA\_CHECKING |
| 成功安装APK | Intent.ACTION\_PACKAGE\_ADDED |
| 成功删除APK | Intent.ACTION\_PACKAGE\_REMOVE |
| 重启设备 | Intent.ACTION\_REBOOT |
| 屏幕被关闭 | Intent.ACTION\_SCREEN\_OFF |
| 屏幕被打开 | Intent.ACTION\_SCREENT\_ON |
| 关闭系统时 | Intent.ACTION\_SHUTDOWN |
| 重启设备 | Intent.ACTION\_REBOOT |

注：当使用系统广播时，只需要在注册广播接收者时定义相关的action即可，并不需要手动发送广播，当系统有相关操作时会自动进行系统广播

##### 有序广播

* 定义：发送出去的广播被接收者按照先后顺序接收，有序是针对广播接收者而言的
* 广播接收者接收广播的顺序规则（同时面向静态和动态注册的广播接收者）
* 按照Priority属性值从大到小排序
* Priority属性相同者，动态注册的广播优先
* 特点：
* 接收广播按顺序接收
* 先接收的广播接收者可以对广播进行截断，即后接收的广播接收者不在接收到此广播
* 先接收的广播接收者可以对广播进行修改，那么后接收的广播接收者将接收到被修改后的广播
* 具体使用有序广播的使用过程和普通广播非常类似，差异仅在于广播的发送方式：sendOrderedBroadcast(intent);

##### APP应用内广播

* Android中的广播可以跨App直接通信(exported对于有intent-filter情况下默认为true)
* 冲突可能出现的问题：
* 其他App针对性发出与当前App intent-filter 相匹配的广播，由此导致当前App不断接收广播并处理；
* 其他App注册与当前App一致的intent-filter用于接收广播，获取广播具体信息。即会出现安全性&效率性的问题
* 解决方案 使用App应用内广播（Local Broadcast）
* App应用内广播可以理解为一种局部广播，广播的发送者和接收者都同属于一个App
* 相比于全局广播（普通广播），App应用内广播优势体现在：安全性高 & 效率高
* 具体使用1 将全局广播设置成局部广播
* 注册广播时将exported属性设置为false，使得非本App内部发出的此广播不被接受
* 在广播发送和接收时，增设相应权限permission，用于权限验证
* 发送广播时指定该广播接收器所在的包名，此广播将只会发送到此包中的App内与之相匹配的有效广播接收器中，通过intent.setPackage(packageName)指定包名
* 具体使用2 使用封装好的LocalBroadcastManager类

使用方式上与全局广播几乎相同，只是注册 / 取消注册广播接收器和发送广播时将参数的context变成了LocalBroadcastManager的单一实例

注：对于LocalBroadcastManager方式发送的应用内广播，只能通过LocalBroadcastManager动态注册，不能静态注册。

```java //注册应用内广播接收器 //步骤1：实例化BroadcastReceiver子类 & IntentFilter mBroadcastReceiver mBroadcastReceiver = new mBroadcastReceiver(); IntentFilter intentFilter = new IntentFilter();

//步骤2：实例化LocalBroadcastManager的实例 localBroadcastManager = LocalBroadcastManager.getInstance(this);

//步骤3：设置接收广播的类型 intentFilter.addAction(android.net.conn.CONNECTIVITY\_CHANGE);

//步骤4：调用LocalBroadcastManager单一实例的registerReceiver（）方法进行动态注册 localBroadcastManager.registerReceiver(mBroadcastReceiver, intentFilter);

//取消注册应用内广播接收器 localBroadcastManager.unregisterReceiver(mBroadcastReceiver);

//发送应用内广播 Intent intent = new Intent(); intent.setAction(BROADCAST\_ACTION); localBroadcastManager.sendBroadcast(intent); ```

##### 粘性广播

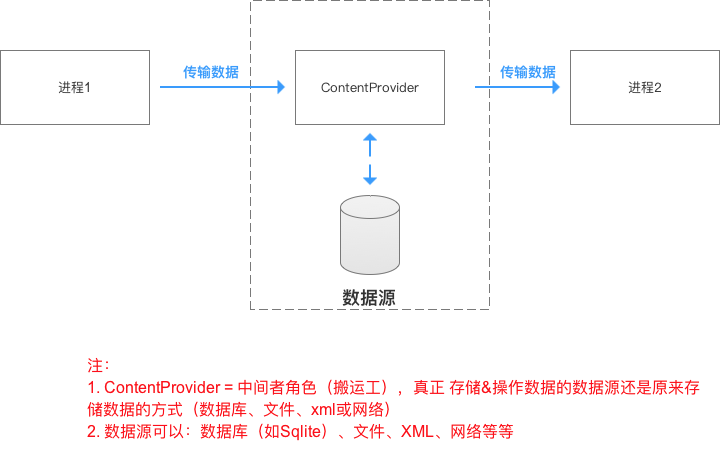
Android 5.0 & API 21中已经失效

### ContentProvider

内容提供者

#### 作用

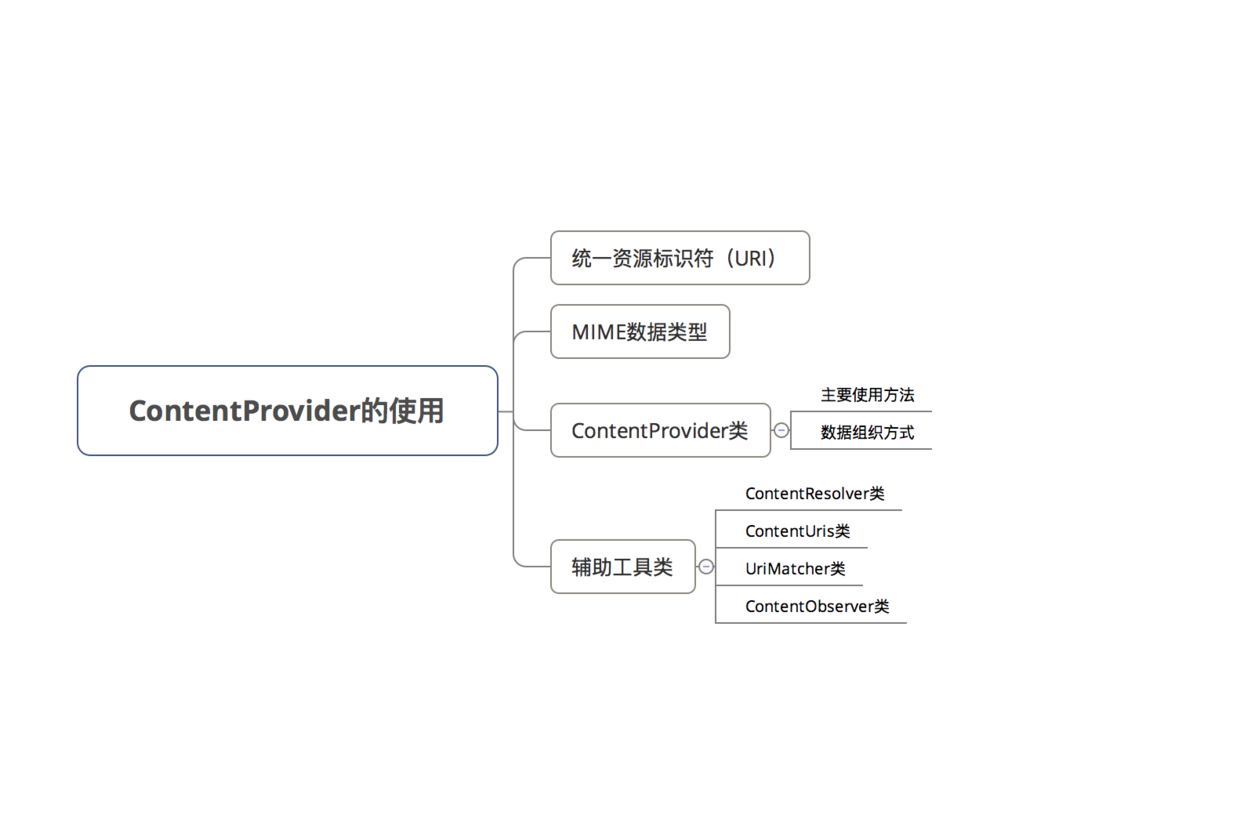
进程间进行数据交互&共享，即跨进程通信



#### 原理

ContentProvider的底层采用Android中的Binder机制

#### 具体使用



##### 统一资源标识符（RUI）

作用：唯一标识ContentProvider & 其中的数据，外界进程通过URI找到对应的ContentProvider & 其中的数据，再进行数据操作

具体使用：

URI分为系统预置 & 自定义，分别对应系统内置的数据（如通讯录、日程表等等）和自定义数据库。



// 设置URI  
Uri uri = Uri.parse("content://com.carson.provider/User/1")   
// 上述URI指向的资源是：名为 `com.carson.provider`的`ContentProvider` 中表名 为`User` 中的 `id`为1的数据  
  
// 特别注意：URI模式存在匹配通配符\* & ＃  
  
// \*：匹配任意长度的任何有效字符的字符串  
// 以下的URI 表示 匹配provider的任何内容  
content://com.example.app.provider/\*   
// ＃：匹配任意长度的数字字符的字符串  
// 以下的URI 表示 匹配provider中的table表的所有行  
content://com.example.app.provider/table/#

##### MIME数据类型

* MIME，即多功能Internet邮件扩充服务。它是一种多用途网际邮件扩充协议。MIME类型就是设定某种扩展名的文件用一种应用程序来打开的方式类型，当该扩展名文件被访问的时候，浏览器会自动使用指定应用程序来打开。多用于指定一些客户端自定义的文件名，以及一些媒体文件打开方式。
* 作用：指定某个扩展名的文件用某种应用程序来打开。
* ContentProvider根据URI返回MIME类型

ContentProvider.getType(uri);

* MIME类型组成

每种MIME类型由两部分组成=类型+子类型

MIME类型是一个包含两部分的字符串，例：

text / html 类型为text 子类型为html

text/css text/xml 等等

### 3. Context的理解？

Android应用模型是基于组件的应用设计模式，组件的运行要有一个完整的Android工程环境。在这个工程环境下，Activity、Service等系统组件才能够正常工作，而这些组件并不能采用普通的Java对象创建方式，new一下就能创建实例了，而是要有它们各自的上下文环境，也就是Context，Context是维持Android程序中各组件能够正常工作的一个核心功能类。

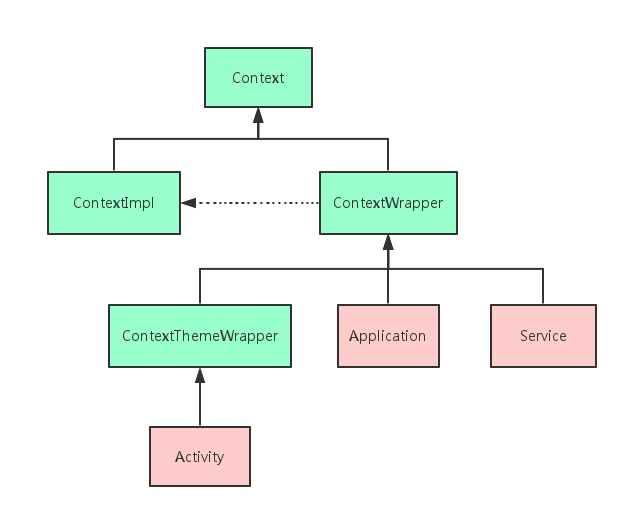
**如何生动形象的理解Context？**

一个Android程序可以理解为一部电影，Activity、Service、BroadcastReceiver和ContentProvider这四大组件就好比戏了的四个主角，它们是剧组（系统）一开始定好的，主角并不是大街上随便拉个人（new 一个对象）都能演的。有了演员当然也得有摄像机拍摄啊，它们必须通过镜头（Context）才能将戏传给观众，这也就正对应说四大组件必须工作在Context环境下。那么Button、TextView等等控件就相当于群演，显然没那么重用，随便一个路人甲都能演（可以new一个对象），但是它们也必须在面对镜头（工作在Context环境下），所以Button mButtom = new Button(context) 是可以的。

**源码中的Context**

public abstract class Context {  
}

它是一个纯抽象类，那就看看它的实现类。

[](

它有两个具体实现类：ContextImpl和ContextWrapper。

其中ContextWrapper类，是一个包装类而已，ContextWrapper构造函数中必须包含一个真正的Context引用，同时ContextWrapper中提供了attachBaseContext()用于给ContextWrapper对象指定真正的Context对象，调用ContextWrapper的方法都会被转向其包含的真正的Context对象。ContextThemeWrapper类，其内部包含了与主题Theme相关的接口，这里所说的主题就是指在AndroidManifest,xml中通过android:theme为Application元素或者Activity元素指定的主题。当然，只有Activity才需要主题，Service是不需要主题的，所以Service直接继承与ContextWrapper，Application同理。而ContextImpl类则真正实现了Context中的所有函数，应用程序中所调用的各种Context类的方法，其实现均来源于该类。**Context得两个子类分工明确，其中ContextImpl是Context的具体实现类，ContextWrapper是Context的包装类。** Activity、Application、Service虽都继承自ContextWrapper（Activity继承自ContextWrapper的子类ContextThemeWrapper），但它们初始化的过程中都会创建ContextImpl对象，由ContextImpl实现Context中的方法。

**一个应用程序有几个Context？**

在应用程序中Context的具体实现子类就是：Activity、Service和Application。那么Context数量=Activity数量+Service数量+1。那么为什么四大组件中只有Activity和Service持有Context呢？BroadcastReceiver和ContextPrivider并不是Context的子类，它们所持有的Context都是其他地方传过去的，所以并不计入Context总数。

**Context能干什么？**

Context能实现的功能太多了，弹出Toast、启动Activity、启动Service、发送广播、启动数据库等等都要用到Context。

TextView tv = new TextView(getContext());  
  
ListAdapter adapter = new SimpleCursorAdapter(getApplicationContext(), ...);  
  
AudioManager am = (AudioManager) getContext().getSystemService(Context.AUDIO\_SERVICE);getApplicationContext().getSharedPreferences(name, mode);  
  
getApplicationContext().getContentResolver().query(uri, ...);  
  
getContext().getResources().getDisplayMetrics().widthPixels \* 5 / 8;  
  
getContext().startActivity(intent);  
  
getContext().startService(intent);  
  
getContext().sendBroadcast(intent);

**Context的作用域**

虽然Context神通广大，但并不是随便拿到一个Context实例就可以为所欲为，它的使用还是有一些规则限制的。由于Context的具体实例是由ContextImpl类去实现的，因此在绝大多数场景下，Activity、Service和Application这三种类型的Context都是可以通用的。不过有几种场景比较特殊，比如启动Activity，还有弹出Dialog。出于安全原因的考虑，Android是不允许Activity或Dialog凭空出现的，一个Activity的启动必须要建立在另一个Activity的基础之上，也就是以此形成返回栈。而Dialog则必须在一个Activity上面弹出（除非是System Alert类型的Dialog），因此在这种场景下，我们只能使用Activity类型的Context，否则将会报错。



从上图我们可以发现Activity所持有的Context的作用域最广，无所不能，因此Activity继承至ContextThemeWrapper，而Application和Service继承至ContextWrapper，很显然ContextThemeWrapper在ContextWrapper的基础上又做了一些操作使得Activity变得更强大。着重讲一下不推荐使用的两种情况：

1. 如果我们用ApplicationContext去启动一个LaunchMode为standard的Activity的时候会报错：

android.util.AndroidRuntimeException: Calling startActivity from outside of an Activity context requires the FLAG\_ACTIVITY\_NEW\_TASK flag. Is this really what you want?

这是因为非Activity类型的Context并没有所谓的任务栈，所以待启动的Activity就找不到栈了。解决这个问题的方法就是为待启动的Activity指定FLAG\_ACTIVITY\_NEW\_TASK标记位，这样启动的时候就为它创建一个新的任务栈，而此时Activity是以singleTask模式启动的。所有这种用Application启动Activity的方式都不推荐，Service同Application。

1. 在Application和Service中去LayoutInflate也是合法的，但是会使用系统默认的主题样式，如果你自定义了某些样式可能不会被使用，这种方式也不推荐使用。

一句话总结：**凡是跟UI相关的，都应该使用Activity作为Context来处理；其他的一些操作，Service、Activity、Application等实例都可以，当然了注意Context引用的持有，防止内存泄露。**

**如何获取Context？**

有四种方法：

1. View.getContext 返回当前View对象的Context对象，通常是当前正在展示的Activity对象。
2. Activity.getApplicationContext 获取当前Activity所在的进程的Context对象，通常我们使用Context对象时，要优先考虑这个全局的进程Context。
3. ContextWrapper.getBaseContext() 用来获取一个ContextWrapper进行装饰之前的Context，可以使用这个方法，这个方法在实际开发中使用的不多，也不建议使用。
4. Activity.this 返回当前Activity实例，如果是UI控件需要使用Activity作为Context对象，但是默认的Toast实际上使用ApplicationContext也可以。

**getApplication()和getApplicationContext()的区别？**

其内存地址是一样的。Application本身就是一个Context，这里获取getApplicationContext得到的结果就是Application本身的实例。getApplication方法的语义性很强，就是用来获取Application实例的，但是这个方法只有在Activity和Service中才能调用的到。那么也许在绝大多数情况下我们都是在Activity或者Service中使用Application，但是如果在一些其他的场景，比如BroadcastReceiver中也想获取Application实例，这时就可以借助getApplicationContext方法了。

public class MyReceiver extends BroadcastReceiver{  
 @Override  
 public void onReceive(Contextcontext,Intentintent){  
 Application myApp= (Application)context.getApplicationContext();  
 }  
}

### 4. AsyncTask详解

#### Android中的线程

在操作系统中，线程是操作系统调度的最小单元，同时线程又是一种受限的系统资源，即线程不可能无限制的产生，并且**线程的创建和销毁都会有相应的开销**。当系统中存在大量的线程时，系统会通过时间片轮转的方式调度每个线程，因此线程不可能做到绝对的并行。

如果在一个进程中频繁的创建和销毁线程，显然不是高效地做法。正确的做法是采用线程池，一个线程池会缓存一定数量的线程，通过线程池就可以避免因为频繁创建和销毁线程所带来的系统开销。

#### AsyncTask简介

AsyncTask是一个抽象类，它是由Android封装的一个轻量级异步类，它可以在线程池中执行后台任务，然后把执行的进度和最终结果传递给主线程并在主线程中更新UI。

AsyncTask的内部封装了两个线程池（SerialExecutor和THREAD\_POOL\_EXECUTOR）和一个Handle（InternalHandler）。

其中SerialExecutor线程池用于任务的排队，让需要执行的多个耗时任务，按顺序排列，THREAD\_POLL\_EXECUTOR线程池才真正的执行任务，InternalHandler用于从工作线程切换到主线程。

##### AsyncTask的泛型参数

AsyncTask的类声明如下：

public abstract class AsyncTask<Params,Progress,Result>

AsyncTask是一个抽象泛型类。

Params：开始异步任务时传入的参数类型

Progress：异步任务执行过程中，返回下载进度值的类型

Result：异步任务执行完成后，返回的结果类型

如果AsyncTask确定不需要传递具体参数，那么这三个泛型参数可以用Void来代替。

##### AsyncTask的核心方法

**onPreExecute()**

这个方法会在**后台任务开始执行之前调用，在主线程执行**。用于进行一些界面上的初始化操作，比如显示一个进度条对话框等等。

**doInBackground(Params...)**

这个方法中的所有代码都会在子线程中运行，我们应该在这里去处理所有的耗时任务。

任务一旦完成就可以通过return语句来将任务的执行结果进行返回，如果AsyncTask的第三个泛型参数指定的是Void，就可以不返回任务执行结果。**注意，这个方法中是不可以进行UI操作的，如果需要更新UI元素，比如说反馈当前任务的执行进度，可以调用publishProgress(Progress ...)方法来完成。**

**onProgressUpdate(Progress...)**

当在后台任务中调用了publishProgress(Progress...)方法后，这个方法就很快被调用，方法中携带的参数就是在后台任务中传递过来的。**在这个方法中可以对UI进行操作，在主线程中进行，利用参数中的数值就可以对界面元素进行相应的更新。**

**onPostExecute(Result)**

当doInBackground(Params...)执行完毕并通过return语句进行返回时，这个方法就很快被调用。返回的数据会作为参数传递到此方法中，**可以利用返回的数据来进行一些UI操作，在主线程中进行，比如说提醒任务执行的结果，以及关闭掉进度条对话框等等。**

上面几个方法的调用顺序为：onPreExecute() --> doInBackground() --> publishProgress() --> onProgressUpdate() --> onPostExecute()

如果不需要执行更新进度则为：onPreExecute() --> doInBackground() --> onPostExecute()

除了上面四个方法，AsyncTask还提供了onCancelled()方法，**它同样在主线程中执行，当异步任务取消时，onCancelled会被调用，这个时候onPostExecute()则不会调用，但是，AsyncTask的cancel()方法并不是真正的去取消任务，只是设置这个任务为取消状态，我们需要在doInBackground()判断终止任务。就好比想要终止一个线程，调用interrupt()方法，只是进行标记为中断，需要在线程内部进行标记判断然后中断线程。**

##### 使用AsyncTask的注意事项

1. 异步任务的实例必须在UI线程中创建，即AsyncTask对象必须在UI线程中创建。
2. execute(Params ... params)方法必须在UI线程中调用。
3. 不要手动调用onPreExecute()、doInBackground(Params ... params)、onProgressUpdate()、onPostExecute() 这几个方法。
4. 不能在doInBackground()中更改UI组件信息。
5. 一个任务实例只能执行一次，如果执行第二次将会抛异常。

### 5. Android虚拟机以及编译过程

#### 什么是Dalvik虚拟机？

Dalvik是Google公司自己设计用于Android平台的Java虚拟机，它是Android平台的重要组成部分，支持dex格式的Java应用程序的运行。dex格式是专门为Dalvik设计的一种压缩格式，适合内存和处理器速度有限的系统。Google对其进行了特定的优化，是的Dalvik具有**高效、简洁、节省资源**的特点。从Android系统架构图知，Dalvik虚拟机运行在Android的运行时库层。

Dalvik作为面向Linux、为嵌入式操作系统设计的虚拟机，主要负责完成对象生命周期、堆栈管理、线程管理、安全和异常管理，以及垃圾回收等。另外，Dalvik早期并没有JIT编译器，知道Android2.2才加入了对JIT的技术支持。

#### Dalvik虚拟机的特点

体积小，占用内存空间小。

专有的DEX可执行文件格式，体积更小，执行速度更快。

常量池采用32位索引值，寻址类方法名、字段名，常量更快。。

基于寄存器架构，并拥有一套完整的指令系统。

提供了对象生命周期管理，堆栈管理，线程管理，安全和异常管理以及垃圾回收等重要功能。

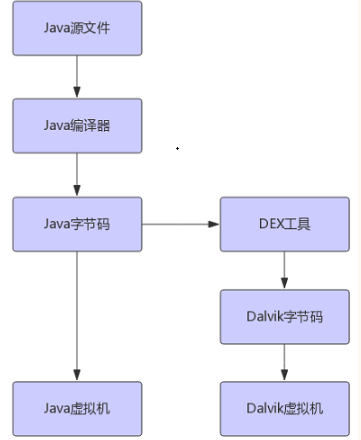
所有的Android程序都运行在Android系统进程里，每个进程对应着一个Dalvik虚拟机实例。

#### Dalvik虚拟机与Java虚拟机的区别

Dalvik虚拟机与传统的Java虚拟机有着许多不同点，两者并不兼容，它们显著的不同点主要表现在以下几个方面：

**Java虚拟机运行的是Java字节码，Dalvik虚拟机运行的是Dalvik字节码。**

传统的Java程序经过编译，生成Java字节码保存在class文件中，Java虚拟机通过解码class文件中的内容来运行程序。而Dalvik虚拟机运行的是Dalvik字节码，所有的Dalvik字节码由Java字节码转换而来，并被打包到一个DEX可执行文件中。Dalvik虚拟机通过解码DEX文件来执行这些字节码。



**Dalvik可执行文件体积小，Android SDK中有一个叫dx的工具负责将Java字节码转换为Dalvik字节码。**

消除其中的冗余信息，重新组合形成一个常量池，所有的类文件共享同一个常量池。由于dx工具对常量池的压缩，使得相同的字符串常量在DEX文件中只出现一次，从而减小了文件的体积。

简单来讲，dex格式文件就是将多个class文件中公有的部分统一存放，去除冗余信息。

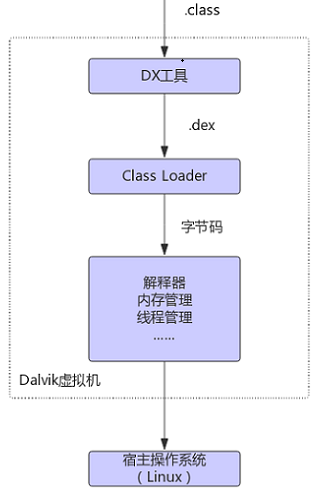
**Java虚拟机与Dalvik虚拟机架构不同，这也是Dalvik与JVM之间最大的区别。**

**Java虚拟机基于栈架构。**程序在运行时虚拟机需要频繁的从栈上读取或写入数据，这个过程需要更多的指令分配与内存访问次数，会耗费不少CPU时间，对于像手机设备资源有限来说，这是相当大的一笔开销。

**Dalvik虚拟机基于寄存器架构。**

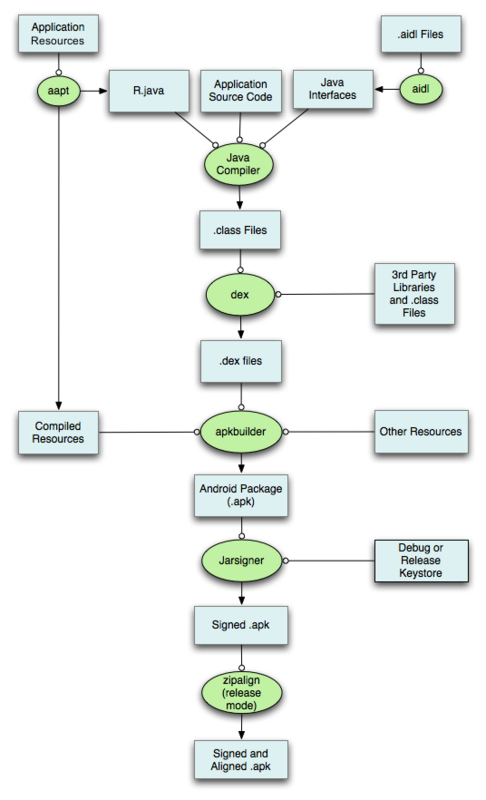
数据的访问通过寄存器间直接传递，这样的访问方式比基于栈方式要快很多。

#### Dalvik虚拟机的结构



一个应用首先经过DX工具将class文件转换成Dalvik虚拟机可以执行的dex文件，然后由类加载器加载原生类和Java类，接着由解释器根据指令集对Dalvik字节码进行解释、执行。最后，根据dvm\_arch参数选择编译的目标机体系结构。

#### Android APK编译打包流程



1. AAPT（Android Asset Packaging Tools）工具会打包应用中的资源文件，如AndroidManifest.xml、layout布局中的xml等，并将xml文件编译成二进制形式，当然assets文件夹中的文件不会被编译，图片以及raw文件夹中的资源也会保持原有的形态，需要注意的是raw文件夹中的资源也会生成资源ID。AAPT编译完成后会生成R.java文件。
2. AIDL工会将所有的aidl接口转换为java接口。
3. 所有的Java源代码、R文件、接口都会编译器编译成.class文件。
4. Dex工具会将上述产生的.class文件以及第三方库和其他class文件转化为dex（Dalvik虚拟机可执行文件）文件，dex文件最终会被打包进APK文件。
5. apkbuilder会把编译后的资源和其他资源文件同dex文件一起打入APK中。
6. 生成APK文件之后，，需要对其签名才能安装到设备上，平时测试都会使用debug keystore，当发布应用时必须使用release版的keystore对应用进行签名。
7. 如果对APK正式签名，还需要使用zipalign工具对APK进行对齐操作，这样做的好处是当应用运行时能提高速度，但是会相应的增加内存开销。

**总结：编译 --> DEX --> 打包 --> 签名和对齐**

#### ART虚拟机与Dalvik虚拟机的区别

##### 什么是ART？

ART代表Android Runtime，其处理应用程序执行的方式完全不同于Dalvik，Dalvik是依靠一个Just-In-Time（JIT）编译器去解释字节码。开发者编译后的应用代码需要通过一个解释器在用户的设备上运行，这一机制并不高效，但让应用能更容易在不同硬件和架构上运行。ART则完全改变了这套做法，在应用安装时就预编译字节码到机器语言，这一机制叫Ahead-Of-Time（AOT）编译。在移除解释代码这一过程后，应用程序执行将更加效率。启动更快。

##### ART优点：

1. 系统性能的显著提升。
2. 应用启动更快、运行更快、体验更流畅、触摸反馈更及时。
3. 更长的电池续航能力
4. 支持更低的硬件。

##### ART缺点：

1. 更大的存储空间占用，可能会增加10%-20%
2. 更长的应用安装时间

#### ART虚拟机相对于Dalvik虚拟机的提升

##### 预编译

在Dalvik中，如同其他大多数JVM一样，都采用的是JIT来做及时翻译（动态翻译），将dex或odex中并排的Dalvik code（或者叫smali指令集）运行态翻译成native code去执行。JIT的引入使得Dalvik提升了3-6倍的性能。

而在ART中，完全抛弃了Dalvik的JIT，使用了AOT直接在安装时将其完全翻译成native code。这一技术的引入，使得虚拟机执行指令的速度又一重大提升。

##### 垃圾回收机制

首先介绍下Dalvik的GC过程，主要有四个过程：

1. 当GC被触发时候，其会去查找所有活动的对象，这个时候整个程序与虚拟机内部的所有线程就会挂起，这样目的是在较少的堆栈里找到所引用的对象。
2. GC对符合条件的对象进行标记。
3. GC对标记的对象进行回收。
4. 恢复所有线程的执行现场继续运行。

**Dalvik这么做的好处是，当pause了之后，GC势必是相当快速的，但是如果出现GC频繁并且内存吃紧势必会导致UI卡顿、掉帧、操作不流畅等等。**

后来ART改善了这种GC方式，主要的改善点在将其**非并发过程改成了部分并发，还有就是对内存的重新分配管理。**

当ART GC发生时：

1. GC将会锁住Java堆，扫描并进行标记。
2. 标记完毕释放掉Java堆的锁，并且挂起所有线程。
3. GC对标记的对象进行回收。
4. 恢复所有线程的执行继续运行。
5. **重复2-4直到结束。**

可以看出整个过程做到了部分并发使得时间缩短，GC效率提高两倍。

##### 提高内存使用，减少碎片化

Dalvik内存管理特点是：内存碎片化严重，当然这也是标记清除算法带来的弊端。

**ART的解决：** 在ART中，它将Java分了一块空间命名为 Large-Object-Space，这个内存空间的引入用来专文存放大对象，同时ART又引入了 moving collector 的技术，即将不连续的物理内存快进行对齐。对齐之后内存碎片化就得到了很好的解决。Large-Object-Space的引入是因为moving collector对大块内存的位移时间成本太高。据官方统计，ART的内存利用率提高了10倍左右，大大提高了内存的利用率。

### 6. 进程保活方案

#### 保活的两个方案

* 提高进程优先级，降低进程被杀死的概率
* 在进程被杀死后，进行拉活

##### 进程的优先级，划分五级：

1. 前台进程（Foreground process）
2. 可见进程（Visible process）
3. 服务进程（Service process）
4. 后台进程（Background process）
5. 空进程（Empty process）

**前台进程一般有以下特点：**

* 拥有用户正在交互的Activity（已调用onResume）
* 拥有某个Service，后者绑定到用户正在交互的Activity
* 拥有正在前台运行的Service（服务已调用startForeground）
* 拥有一个正执行生命周期回调的Service（onCreate，onStart或onDestory）
* 拥有其正执行其onReceive()方法的BroadcastReceiver

#### 提高进程优先级

* 利用Activity提升权限

监控手机锁屏解锁事件，在屏幕锁屏时启动1像素的Activity，在用户解锁时将Activity销毁，注意该Activity需设计成用户无感知。

* Notification提升权限

Android中Service的优先级为4，通过setForeground接口可以将后台Service设置为前台Service，使进程的优先级由4提升为2，从而是进程的优先级仅仅低于用户当前正在交互的进程，与可见进程优先级一致，使进程被杀死的概率大大降低。

#### 进程死后拉活

* 利用系统广播拉活

在发生特定系统事件时，系统会发出相应的广播，通过在AndroidManifest中静态注册对应的广播监听器，即可在发生响应事件时拉活。

* 利用第三方应用广播拉活

该方案总的设计思想与接收系统广播类似，不同的是该方案为接收第三方Top应用广播。通过反编译第三方Top应用，如微信、支付宝等等，找出它们外发的广播，在应用中进行监听，这样当这些应用发出广播时，就会将我们的应用拉活。

* 利用系统Service机制拉活

将Service设置为START\_STICKY，利用系统机制在Service挂掉后自动拉活。

* 利用Native进程拉活

利用Linux中的fork机制创建Native进程，在Native进程中监控主进程的存活，当主进程挂掉后，在Native进程中立即对主进程进行拉活。

原理：在Android中所有进程和系统组件的生命周期受ActivityManagerService的统一管理。而且，通过Linux的fork机制创建的进程为纯Linux进程，其生命周期不受Android管理。

* 利用 JobScheduler机制拉活

在Android5.0以后系统对Native进程等加强了管理，Native拉活方式失效。系统在Android5.0以后版本提供了 JobScheduler接口，系统会定时调用该进程以使应用进行一些逻辑操作。

* 利用账号同步机制拉活

Android系统的账号同步机制会定期同步账号进行，该方案目的在于利用同步机制进行进程的拉活。

#### 其他有效的拉活方案

相互唤醒。

**参考：**

[进程保活方案1](https://www.jianshu.com/p/845373586ac1)

[Android进程保活招式大全](http://geek.csdn.net/news/detail/95035)

### 7. Android 消息机制

#### 消息机制简介

在Android中使用消息机制，我们首先想到的就是Handler。没错，Handler是Android消息机制的上层接口。我们通常只会接触到Handler和Message开完成消息机制，其实内部还有两大助手共同完成消息传递。

#### 消息机制模型

消息机制主要包含：MessageQueue、Handler、Looper和Message这四大部分。

* Message

需要传递的消息，可以传递数据

* MessageQueue

消息队列，但是它的内部实现并不是用的队列，实际上是通过一个单链表的数据结构来维护消息列表，因为单链表在插入和删除上比较有优势。主要功能是向消息池传递消息（MessageQueue.enqueueMessage）和取走消息池的消息（MessageQueue.next）

* Handle

消息辅助类，主要功能是向消息池发送各种消息事件（Handler.sendMessage）和处理相应消息事件（Handler.handleMessage）

* Looper

不断循环执行（Looper.loop），从MessageQueue中读取消息，按分发机制将消息分发给目标处理者。

#### 消息机制的架构

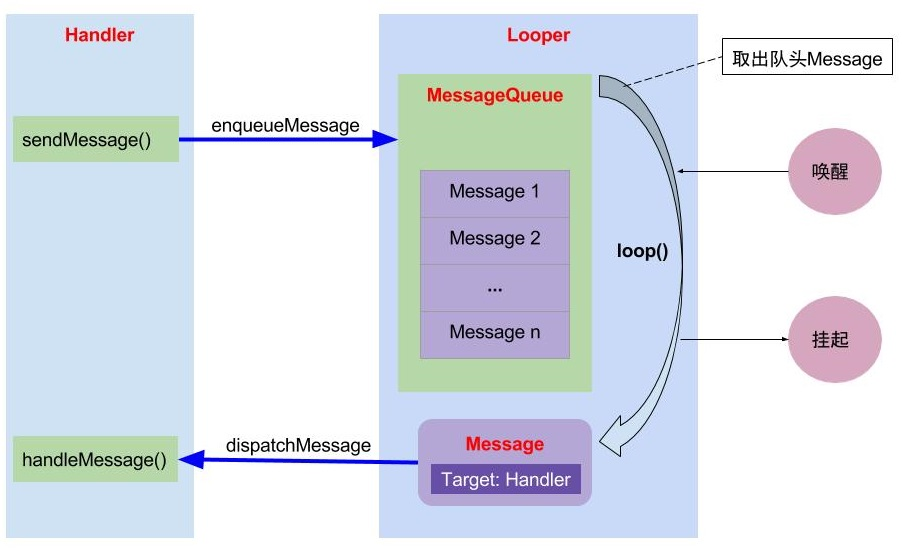
**运行流程：**

在子线程执行完耗时操作，当Handler发送消息时，将会调用MessageQueue.enqueueMessage，向消息队列中添加消息。当通过Looper.loop开启循环后，会不断的从线程池中读取消息，即调用MessageQueue.next，然后调用目标Handler（即发送该消息的Handler）的dispatchMessage方法传递消息，然后返回到Handler所在线程，目标Handler收到消息，调用handleMessage方法，接收消息，处理消息。

**MessageQueue、Handler和Looper三者之间的关系：**

每个线程中只能存在一个Looper，Looper是保存在ThreadLocal中。主线程已经创建一个Looper，所以在主线程中不需要在创建Looper，但是在其他线程中需要创建Looper。每个线程中可以有多个Handler，即一个Looper可以处理来自多个Handler的消息。Looper中维护一个MessageQueue，来维护消息队列，消息队列中的Message可以来自不同的Handler。

#### 总结



Android消息机制之ThreadLocal的工作原理：

Looper中还有一个特殊的概念，那就是ThreadLocal，ThreadLocal并不是线程，它的作用是可以在每个线程中存储数据。大家知道，Handle创建的时候会采用当前线程的Looper来构造消息循环系统，那么Handle内部如何获取当前线程的Looper呢？这就要使用ThreadLocal了，ThreadLocal可以在不同的线程之中互不干扰的存储并提供数据，通过ThreadLocal可以轻松的获取每个线程的Looper。当然，需要注意的是，线程默认是没有Looper的，如果需要使用Handler就必须为线程创建Looper。大家经常提到的主线程，也叫UI线程，它就是ActivityThread，ActivityThread被创建时就会初始化Looper，这也是在主线程默认可以使用Handle的原因。

ThreadLocal是一个线程内部的数据存储类，通过它可以在指定的线程中存储数据，数据存储以后，只有在指定线程中可以获取到存储的数据，对于其他线程来说无法获取到数据。在日常开发中用到ThreadLocal的地方很少，但是在某些特殊的场景下，通过ThreadLocal可以轻松的实现一些看起来很复杂的功能，这一点在Android源码中也有所体现，比如Looper、ActivityThread以及AMS中都用到了ThreadLocal。具体到ThreadLocal的使用场景，这个不好统一来描述，一般来说，当某些数据是以线程为作用域并且不同线程具有不同的数据副本的时候，就可以采用ThreadLocal。比如对于Handle来说，它需要获取当前线程的Looper，很显然Looper的作用域就是线程并且不同线程具有不同的Looper，这个时候通过ThreadLocal就可以轻松的实现Looper在线程中的存取。

ThreadLocal另外一个使用场景是复杂逻辑下的对象传递，比如监听器的传递，有些时候一个线程中的任务过于复杂，这可能表现为函数调用栈比较深以及代码入口的多样性，在这种情况下，我们又需要监听器能够贯穿整个线程的执行过程，这个时候可以怎么做呢？其实就可以采用ThreadLocal，采用ThreadLocal可以让监听器作为线程内的局部对象而存在，在线程内部只要通过get方法就可以获取到监听器。而如果不采用ThreadLocal，那么我们能想到的可能就是一下两种方法：

1. 将监听器通过参数的形式在函数调用栈中进行传递

在函数调用栈很深的时候，通过函数参数来传递监听器对象几乎是不可接受的

1. 将监听器作为静态变量供线程访问

这是可以接受的，但是这种状态是不具有可扩充性的，比如如果同时有两个线程在执行，那就需要提供两个静态的监听器对象，如果有十个线程在并发执行呢？提供十个静态的监听器对象？这显然是不可思议的。而采用ThreadLocal每个监听器对象都在自己的线程内部存储，根本就不会有这个问题。

ThreadLocal<Boolean> threadLocal = new ThreadLocal<>();  
threadLocal.set(true);  
 Log.i(TAG, "getMsg: MainThread"+threadLocal.get());  
 new Thread("Thread#1"){  
 @Override  
 public void run() {  
 threadLocal.set(false);  
 Log.i(TAG, "run: Thread#1" + threadLocal.get());  
 }  
 }.start();  
 new Thread("Thread#2"){  
 @Override  
 public void run() {  
 Log.i(TAG, "run: Thread#2" + threadLocal.get());  
 }  
 }.start();  
   
输出：true、false、null

虽然在不同线程中访问的是同一个ThreadLocal对象，但是它们获取到的值却是不一样的。不同线程访问同一个ThreadLocal的get方法，ThreadLocal内部会从各自的线程中取出一个数组，然后再从数据中根据当前ThreadLocal的索引去查找出对应的value值，很显然，不同线程中的数组是不同的，这就是为什么通过ThreadLocal可以在不同的线程中维护一套数据副本并且彼此互不干扰。

ThreadLocal是一个泛型类，里面有两个重要方法：get()和set()方法。它们所操作的对象都是当前线程的localValues对象的table数组，因此在不同线程中访问同一个ThreadLocal的get和set方法，它们对ThreadLocal所做的读写操作仅限于各自线程的内部，这就是为什么ThreadLocal可以在多个线程中互不干扰的存储和修改数据。

<https://blog.csdn.net/singwhatiwanna/article/details/48350919>

### 8. Window、Activity、DecorView以及ViewRoot之间的关系

#### 职能简介

**Activity**

Activity并不负者视图控制，它只是控制生命周期和处理事件。真正控制视图的是Window。一个Activity包含了一个Window，Window才是真正代表一个窗口。**Activity就像一个控制器，统筹视图的添加与显示，以及通过其他回调方法，来与Window以及View进行交互。**

**Window**

Window是视图的承载器，内部持有一个DecorView，而这个DecorView才是view的跟布局。Window是一个抽象类，实际在Activity中持有的是其子类PhoneWindow。PhoneWindow中有个内部类DecorView，通过创建DecorView来加载Activity中设置的布局。Window通过WindowManager将DecorView加载其中，并将DecorView交给ViewRoot，进行视图绘制以及其他交互。

**DecorView**

DecorView是FrameLayout的子类，它可以被认为是Android视图树的根节点视图。

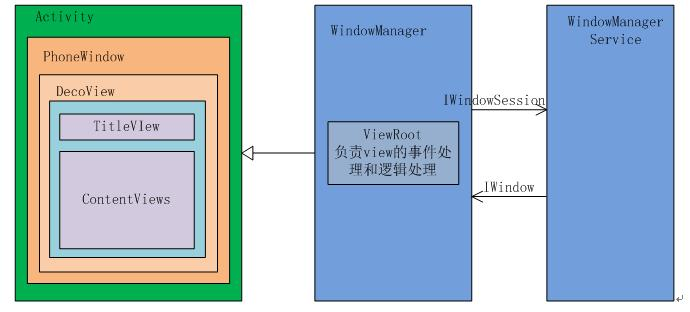
DecorView作为顶级View，一般情况下它内部包含一个竖直方向的LinearLayout，**在这个LinearLayout里面有上下三个部分，上面是个ViewStub，延迟加载的视图（应该是设置ActionBar，根据Theme设置），中间的是标题栏（根据Theme设置，有的布局没有），下面是内容栏。**在Activity中通过setContentView所设置的布局文件其实就是被加到内容栏之中的，成为其唯一子View。

**ViewRoot**

ViewRoot可能比较陌生，但是其作用非常重大。所有View的绘制以及事件分发等交互都是通过它来执行或传递的。

ViewRoot对应ViewRootImpl类，它是连接WindowManagerService和DecorView的纽带，View的三大流程（测量、布局、绘制）均通过ViewRoot来完成。

ViewRoot并不属于View树的一份子。从源码实现上来看，它既是非View的子类，也是非View的父类，但是，它实现了ViewParent接口，这让它可以作为View的名义上的父视图。RootView继承了Handler类，可以接收事件并分发，Android的所有触屏事件，按键事件、界面刷新等事件都是通过ViewRoot来进行分发的。



#### 总结

Activity就像个控制器，不负责视图部分。Window像个承载器，装着内部视图。DecorView就是个顶级视图，是所有View的最外层布局。ViewRoot像个连接器，负者沟通，通过硬件感知来通知视图，进行用户之间的交互。

### 9. Android事件分发机制

[图解 Android 事件分发机制](https://www.jianshu.com/p/e99b5e8bd67b)

简述：

* Activity --> ViewGroup --> View 责任链模式
* dispatchTouchEvent 和 onTouchEvent 一旦 return true，事件就停止传递了（到达终点，没有谁再能收到这个事件），对于 return true 我们经常说事件被消费了，消费了的意思就是事件走到这里就是终点，不会往下传，没有谁能在收到这个事件了。
* dispatchTouchEvent 和 onTouchEvent return false 的时候事件都会回传给父控件的 onTouchEvent处理。对于dispatchTouchEvent返回false的含义应该是：事件停止往子View传递和分发同时开始往父控件回溯（父控件的onTouchEvent开始从下往上回传直到某个onTouchEvent return true），事件分发机制就像递归，return false 的意义就是递归停止然后开始回溯。
* 对于onTouchEvent return false就比较简单了，它就是不消费事件，并让事件继续往父控件的方向从下往上流动。
* oninterceptTouchEvent，用于事件拦截，只存在于ViewGroup中，如果返回true就会交给自己的onTouchEvent处理，如果不拦截就是往子控件往下传递。默认是不会去拦截的，因为子View也需要这个事件，所以onInterceptTouchEvent拦截器return super和false是一样的，事件往子View的dispatchTouchEvent传递。
* 对于ViewGroup，dispatchTouchEvent，之前说的return true就是终结传递，return false就是回溯到父View的onTouchEvent。那么ViewGroup怎样通过dispatchTouchEvent方法能把事件分发到自己的onTouchEvent处理呢？return false 和 true 都不行，那么只能通过 onInterceptTouchEvent把事件拦截下来给自己的onTouchEvent，所以ViewGroup的dispatchTouchEvent方法的super默认实现就是去调用onInterceptTouchEvent，记住这一点。

**总结：**

* 对于dispatchTouchEvent，onTouchEvent，return true 是终结事件传递，return false是回溯父View的onTouchEvent方法
* ViewGroup想把事件分发给自己的onTouchEvent处理，需要拦截器onInterceptTouchEvent方法return true把事件拦截下来
* ViewGroup的拦截器onInterceptTouchEvent默认是不拦截的，所以return super和return false是一样的
* View没有拦截器，为了让View可以把事件分发给自己的onTouchEvent处理，View的dispatchTouchEvent默认实现（super）就是把事件分发给自己的onTouchEvent

### 10. dp、sp、px的理解以及相互转换

**px**

像素，对应于屏幕上的实际像素。在画分割线的可以用到，用其他单位画可能很模糊。

**dp、dip**

与密度无关的像素单位，基于屏幕的物理尺寸的抽象单位，在160dpi的屏幕上1dp相当于1px，一般用于空间大小的单位。

**sp**

与缩放无关的像素单位，类似dp，不用之处在于它还会根据用户字体大小配置而缩放。开发中指定字体大小时建议使用sp，因为它会根据屏幕密度和用户字体配置而适配

DisplayMetrics dm = getResources().getDisplayMetrics();  
int dpi = dm.densityDpi; // 屏幕Dpi  
int width = dm.widthPixels; // 当前屏幕可用空间的宽(像素)  
int height = dm.heightPixels; // 高(像素)  
float density = dm.density; // 屏幕密度  
float scale = dm.scaledDensity; // 字体缩放比例  
  
// 获得设备的配置信息  
Configuration c = getResources().getConfiguration();  
int widthDp = c.screenWidthDp; // 当前屏幕可用空间的宽(dp)  
int heightDp = c.screenHeightDp; // 高(dp)  
int dpi2 = c.densityDpi; // 屏幕Dpi  
float fontScale = c.fontScale; // 字体缩放比例

/\*\*  
 \* 将px值转换为dip或dp值  
 \* @param pxValue  
 \* @return  
 \*/  
 public static int px2dip(Context context, float pxValue) {  
 final float scale = context.getResources().getDisplayMetrics().density;  
 return (int) (pxValue / scale + 0.5f);  
 }  
  
 /\*\*  
 \* 将dip或dp值转换为px值  
 \*  
 \* @param dipValue  
 \* @return  
 \*/  
 public static int dip2px(Context context, float dipValue) {  
 final float scale = context.getResources().getDisplayMetrics().density;  
 return (int) (dipValue \* scale + 0.5f);  
 }  
  
 /\*\*  
 \* 将px值转换为sp值  
 \* @param pxValue  
 \* @return  
 \*/  
 public static int px2sp(Context context, float pxValue) {  
 final float fontScale = context.getResources().getDisplayMetrics().scaledDensity;  
 return (int) (pxValue / fontScale + 0.5f);  
 }  
  
 /\*\*  
 \* 将sp值转换为px值  
 \* @param spValue  
 \* @return  
 \*/  
 public static int sp2px(Context context, float spValue) {  
 final float fontScale = context.getResources().getDisplayMetrics().scaledDensity;  
 return (int) (spValue \* fontScale + 0.5f);  
 }

### 11. RelativeLayout和LinearLayout在实现效果同等的情况下使用哪个？为什么？

[Android中RelativeLayout和LinearLayout性能分析](https://www.jianshu.com/p/8a7d059da746)

选择LinearLayout，因为RelativeLayout在measure过程需要两次。

//RelativeLayout源码  
 View[] views = mSortedHorizontalChildren;  
 int count = views.length;  
 for (int i = 0; i < count; i++) {  
 /\*\*/  
 measureChildHorizontal(child, params, myWidth, myHeight);  
 }  
 /\*\*/  
 for (int i = 0; i < count; i++){  
 /\*\*\*/  
 measureChild(child, params, myWidth, myHeight);  
 }  
  
 //LinearLayout  
 @Override  
 protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
 if (mOrientation == VERTICAL) {  
 measureVertical(widthMeasureSpec, heightMeasureSpec);  
 } else {  
 measureHorizontal(widthMeasureSpec, heightMeasureSpec);  
 }  
 }

从源码我们发现RelativeLayout会对子View做两次measure。这是为什么呢？首先RelativeLayout中子View的排列方式是基于彼此的依赖关系，而这个依赖关系可能和布局中View的顺序并不相同，在确定每个子View的位置的时候，就需要先给所有的子View排序一下。又因为RelativeLayout允许A B两个子View，横向上B依赖于A，纵向上A依赖于B，所以需要横向纵向分别进行一次排序测量。

RelativeLayout另外一个性能问题：

View的measure方法里对绘制过程做了一个优化，如果我们的子View没有要求强制刷新，而父View给子View的传入值也没有变化，也就说子View的位置没有变化，就不会做无谓的measure。但是上面已经说了RelativeLayout要做两次measure，而在做横向测量时，纵向的测量结果尚未完成，只好暂时使用myHeight传入子View系统，假如子View的Height不等于（设置了margin）myHeight的高度，那么measure中优化则不起作用，这一过程将进一步影响RelativeLayout的绘制性能。而LinearLayout则无这方面的担忧，解决这个问题也很好办，如果可以，尽量使用padding代替margin。

**结论**

* RelativeLayout会让子View调用两次onMeasure，LinearLayout再有weight时，也会调用子View两次onMeasure
* RelativeLayout的子View如果高度和RelativeLayout不同，则会引发效率问题。当子View很复杂时，这个问题会更加严重。如果可以，尽量使用padding代替margin
* 在不影响层级深度的情况下，使用LinaerLayout和FrameLayout而不是RelativeLayout

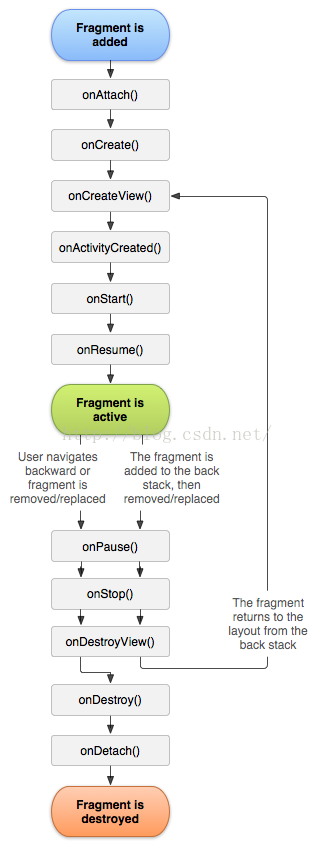
### 12. 布局相关的 <merge>、<viewstub> 控件作用及实现原理

[从源码角度分析ViewStub 疑问与原理](https://blog.csdn.net/androiddevelop/article/details/46632323)

* ViewStub本身是一个视图，会被添加到界面上，之所以看不到是因为其设置了隐藏与不绘制
* 当调用infalte或者ViewStub.setVisibilty(View.VISIBLE)时（两个都使用infalte逻辑），先从父视图上把当前ViewStub删除，再把加载的android:layout视图添加上
* 把ViewStub LayoutParams 添加到加载的android:layout视图上，而其根节点的LayoutParams设置无效
* ViewStub是指用来占位的视图，通过删除自己并添加android:layout视图达到懒加载效果

### 13. Fragment详解

#### 生命周期



#### Fragment使用方式

**静态使用**

步骤：

1. 创建一个类继承Fragment，重写onCreateView方法，来确定Fragment要显示的布局
2. 在Activity的布局xml文件中添加< fragment >标签，并声明name属性为Fragemnt的全限定名，**而且必须给该控件(fragment)一个id或者tag**

**动态使用**

步骤：

1. 在Activity的布局xml文件中添加一个FrameLayout，此控件是Fragment的容器。
2. 准备好Fragment并实例化，获取Fragment管理器对象 FragmentManager。
3. 然后通过Fragment管理器调用beginTransaction()，实例化FragmentTransaction对象，即开启事务。
4. 执行事务并提交。FragmentTransaction对象有以下几种方法：

* add() 向Activity中添加一个Fragment
* remove() 从Activity中移除一个Fragment，如果被移除的Fragment没有添加到回退栈，这个Fragment实例将会被销毁
* replace() 使用一个Fragment替换当前的，实际上就是remove + add
* hide() 隐藏当前的Fragment，仅仅设为不可见，并不会销毁
* show() 显示之前隐藏的Fragment
* detach() 会将view从UI中移除，和remove()不同，此时fragment的状态依然由FragmentManager维护
* attach() 重建View视图，附加到UI上并显示
* commit() 提交事务

**回退栈**

Fragment的回退栈是用来保存每一次Fragment事务发生的变化，如果将Fragment任务添加到回退栈，当用户点击后退按钮时，将会看到上一次保存的Fragment，一旦Fragment完全从回退栈中弹出，用户再次点击后退键，则会退出当前Activity。

FragmentTransaction.addToBackStack(String) 把当前事务的变化情况添加到回退栈。

**Fragment与Activity之间的通信**

Fragment依附于Activity存在，因此与Activity之间的通信可以归纳为以下几种：

* 如果Activity中包含自己管理的Fragment的引用，可以通过访问其public方法进行通信
* 如果Activiy中没有保存任何Fragment的引用，那么没关系，每个Fragment都有一个唯一的TAG或者ID，可以通过getFragmentManager.findFragmentByTag()或者findFragmentById()获得Fragment实例，然后进行操作
* Fragment中可以通过getActivity()得到当前绑定的Activity实例，然后进行操作
* setArguments(Bundle) / getArguments()

**通信优化**

接口实现

**如何处理运行时配置发生变化**

横竖屏切换时导致Fragment多次重建绘制：

不要在构造函数中传递参数，最好通过setArguments()传递。

在onCreate(Bundle savedInstanceState)中判断savedInstanceState为空时在重建，当发生重建时，原本的数据如何保持？类似Activity，Fragment也有onSaveInstanceState方法，在此方法中进行保存数据，然后在onCreate或者onCreateView或者onActivityCreated进行恢复都可以。

### 14. Json、XML

Json是一种轻量级的数据交换格式，具有良好的可读和便于编写的特性。XML即扩展标记语言，用来标记电子文件使其具有结构性的标记语言，可以用来标记数据。定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。

JSON在编码和解码上优于XML，并且数据体积更下，解析更快，与JavaScript交互更加方便，但是对数据的描述性较XML差。

### 15. Assets目录与res目录的区别

assets目录与res下的raw、drawable目录一样，也可用来存放资源文件，但它们三者区别如下：

|  |  |  |  |
| --- | --- | --- | --- |
|  | assets | res/raw | res/drawable |
| 获取资源方式 | 文件路径+文件名 | R.raw.xxx | R.drawable.xxx |
| 是否被压缩 | false | false | true(失真压缩) |
| 能否获取子目录下的资源 | true | false | false |

res/raw和assets的区别：

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即可，assets文件夹下的文件不会被映射到R文件中，访问的时候需要AssetManager类。  
  
res/raw不可以有目录结构，而assets则可以有目录结构，也就是assets目录下可以再建立文件夹。  
  
读取res/raw下的文件资源，通过以下方式获取输入流来进行写操作：

InputStream is = getResources().openRawResource(R.id.filename);

读取assets下的文件资源，通过以下方式获取输入流进行写操作，使用AssetManager。

注意：

1. AssertManager中不能处理单个超过1M的文件，而raw没有这个限制
2. assets文件夹是存放不进行编译加工的原生文件，即该文件夹里面的文件不会像xml、java文件被预编译，可以存放一些图片、html、js等等

### 16. View视图绘制过程原理

View视图绘制需要搞清楚两个问题，一个是从哪里开始绘制，一个是怎么绘制？

从哪里开始绘制？我们平常使用Activity的时候，都会调用setContentView来设置布局文件，没错，视图绘制就是从这个方法开始。

怎么绘制？

在我们的Activity中调用了setContentView之后，会转而执行PhoneWindow的setContentView，在这个方法里面会判断我们存放内容的ViewGroup（这个ViewGroup可以是DecorView也可以是DecorView的子View）是否存在。不存在的话，则会创建一个DecorView处理，并且会创建出相应的窗体风格，存在的话则会删除原先的ViewGroup上面已有的View，接着会调用LayoutInflater的inflate方法以pull解析的方式将当前布局文件中存在的View通过addView的方式添加到ViewGroup上面来，接着在addView方法里面就会执行我们常见的invalidate方法了，这个方法不只是在View视图绘制的过程中经常用到，其实动画的实现原理也是不断的调用这个方法来实现视图不断重绘的，执行这个方法的时候会调用父View的invalidateChild方法，这个方法是属于ViewParent的，ViewGroup以及ViewRootImpl中都会他进行了实现，invalidateChild里面主要做的是就是通过do while循环一层一层计算出当前View的四个点所对应的矩阵在ViewRoot中所对应的位置，那么有了这个矩阵的位置之后最终都会执行ViewRootImpl的invalidateChildInParent方法，执行这个方法的时候首先会检查当前线程是不是主线程，因为我们要开始准备更新UI了，不是主线程的话是不允许更新UI的，接着就会执行scheduleTraversals方法了，这个方法会通过handler来执行doTraversal方法，在这个方法里面就见到了我们平常所熟悉的View视图绘制的起点方法performTraversals了。

那么接下来就是真正的视图绘制流程了，大体上讲View的绘制流程经历了Measure测量、Layout布局以及Draw绘制的三个过程，具体来讲是从ViewRootImpl的performTraversals方法开始，首先执行的将是performMeasure方法，这个方法里面会传入两个MeasureSpec类型的参数，它在很大程度上决定了View的尺寸规格，对于DecorView来说宽高的MeasureSpec值的获取与窗口尺寸以及自身的LayoutParams有关，对于普通View来说其宽高的MeasureSpec值获取由父容器以及自身的LayoutParams属性共同决定，在performMeasure里面会执行measure方法，在measure方法里面会执行onMeasure方法，到这里Measure测量过程对View与ViewGroup来说是没有区别的，但是从onMeasure开始两者有差别了，因为View本身已经不存在子View了，所以他onMeasure方法将执行setMeasuredDimension方法，该方法会设置View的测量值，但是对于ViewGroup来说，因为它里面还存在着子View，那么我们就需要继续测量它里面的子View了，调用的方法是measureChild方法，该方法内部又会执行measure方法，而measure方法转而又会执行onMeasure方法，这样不断的递归进行下去，直到整个View树测量结束，这样performMeasure方法执行结束了。接着便是执行performLayout方法了，performMeasure只是测量出了View树中View的大小了，但是还不知道View的位置，所以也就出现了performLayout方法了，performLayout方法首先会执行layout方法，以确定View自身的位置，如果当前View是ViewGroup的话，则会执行onLayout方法。在onLayout方法里面又会递归的执行layout方法，直到当前遍历到的View不再是ViewGroup为止，这样整个layout布局过程就结束了。在View树中View的大小以及位置都确定之后，接下来就是真正的绘制View显示在界面的过程了，该过程首先从performDraw方法开始，performDraw首先会执行draw方法，在draw方法中首先绘制背景，接着调用onDraw方法绘制自己，如果当前View是ViewGroup的话，还要调用dispatchDraw方法绘制当前ViewGroup的子View，而dispatchDraw方法里面实际上是通过drawChild方法间接调用draw方法形成递归绘制整个View树，直到当前View不再是ViewGroup为止，这样整个View的绘制过程就结束了。

总结：

* ViewRootImpl会调用performTraversals()，其内部会调用performMeasure()、performLayout、performDraw
* performMeasure会调用最外层的ViewGroup的measure() --> onMeasure() ，ViewGroup的onMeasure()是抽象方法，但其提供了measureChildren()，这之中会遍历子View然后循环调用measureChild()，传入MeasureSpec参数，然后调用子View的measure()到View的onMeasure() -->setMeasureDimension(getDefaultSize(),getDefaultSize())，getDefaultSize()默然返回measureSpec的测量数值，所以继承View进行自定义的wrap\_content需要重写。
* performLayout()会调用最外层的ViewGroup的layout(l,t,r,b)，本View在其中使用setFrame()设置本View的四个顶点位置。在onLayout(抽象方法)中确定子View的位置，如LinearLayout会遍历子View，循环调用setChildFrame() --> 子View.layout()
* performDraw()会调用最外层的ViewGroup的draw()方法，其中会先后调用background.draw()绘制背景，onDraw(绘制自己)，dispatchDraw(绘制子View)、onDrawScrollBars(绘制装饰)
* MeasureSpec由两位SpecMode(UNSPECIFIED、EXACTLY(对于精确值和match\_parent)、AL\_MOST(对应warp\_content))和三十位SpecSize组成一个int，DecorView的MeasureSpec由窗口大小和其LayoutParams决定，其他View有父View的MeasureSpec和本View的LayoutParams决定。ViewGroup中有getChildMeasureSpec()来获取子View的MeasureSpec。

### 17. 解决滑动冲突的方式？

在自定义View的过程中经常会遇到滑动冲突问题，一般滑动冲突的类型有三种：（1）外部View滑动方向和内部View滑动方向不一致；（2）外部View滑动方向和内部View滑动方向一致；（3）上述两种情况的嵌套  
  
一般解决滑动冲突都是利用事件分发机制，有两种方式即外部拦截法和内部拦截法：

**外部拦截法：**

实现思路是事件首先是通过父容器的拦截处理，如果父容器不需要该事件，则不拦截，将事件传递到子View上面，如果父容器决定拦截的话，则在父容器的onTouchEvent里面直接处理该事件，这种方法符合事件分发机制；具体实现是修改父容器的onInterceptTouchEvent方法，在达到某一条件的时候，让该方法直接返回true，就可以把事件拦截下来进而调用自己的onTouchEvent方法来处理，但是有一点需要注意的是，如果想让子View能够收到事件，我们需要在onInterceptTouchEvent方法里面判断是DOWN事件的话，就返回false，这样后续的MOVE以及UP事件才有机会传递到子View上面，如果你直接在onInterceptTouchEvent方法里面DOWN情况下返回了true，那么后续的MOVE以及UP事件酱由当前View的onTouchEvent处理了，这样的拦截根本没有意义的，拦截只是在满足一定条件下才会拦截，并不是所有情况下都要拦截。

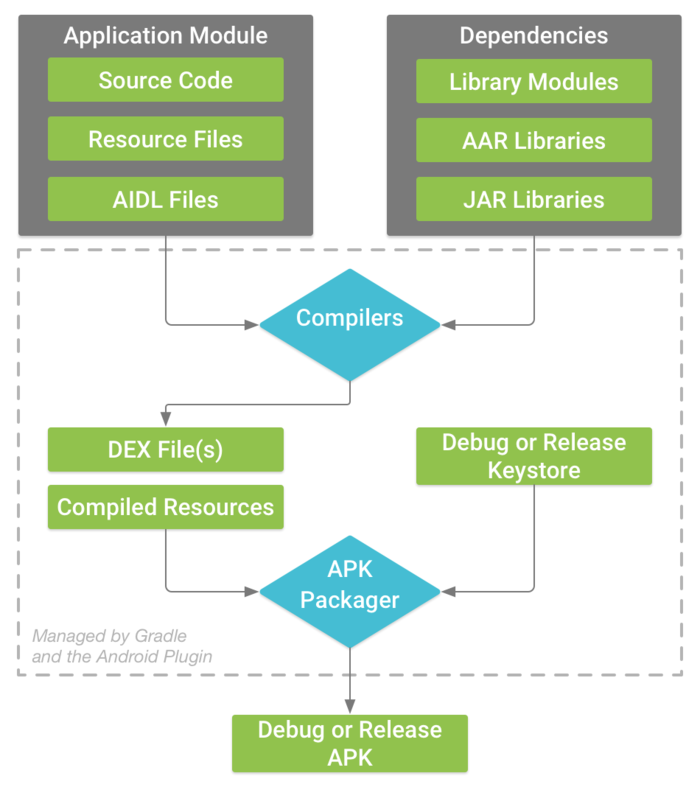
**内部拦截法：**

实现思路是

### 18. APP Build过程

Android Studio点击build按钮之后，AS就会编译整个项目，并将apk安装到手机上，这个过程就是Android工程编译打包过程。主要的流程是：

编译 --> DEX --> 打包 --> 签名

**APK构建概述** 

主要有两个过程：

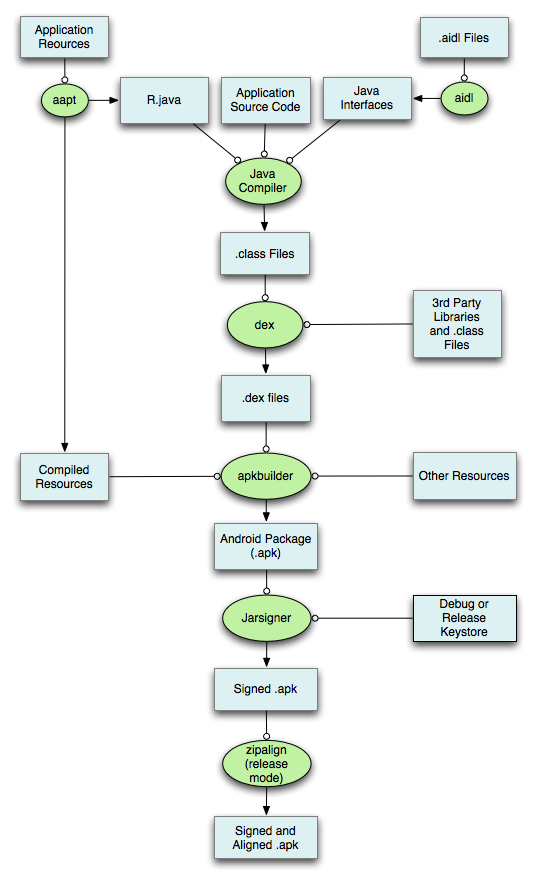
* 编译过程

输入是本工程的文件以及依赖的各种库文件

输出是dex文件和编译后的资源文件

* 打包过程

配合Keystore对上述的输出进行签名对齐，生成最终的apk文件

**APK构建步骤详解** 

主要分为六个步骤：

1. 通过aapt打包res资源文件，生成R.java文件、resource.arsc和res文件（二进制&非二进制如res/raw和pic保持原样）
2. 处理.aidl文件，生成对应的Java接口文件
3. 通过Java Compiler编译R.java、Java接口文件、Java源文件，生成.class文件
4. 通过dex命令，将.class文件和第三方库中的.class文件处理生成classes.dex
5. 通过apkbuilder工具，将aapt生成的resource.arsc和res文件、assets文件和classes.dex一起打包生成apk
6. 通过Jarsigner工具，对上面的apk进行debug或release签名
7. 通过zipalign工具，将签名后的apk进行对齐处理

参考自：

<https://www.jianshu.com/p/e86aadcb19e0>

<http://mouxuejie.com/blog/2016-08-04/build-and-package-flow-introduction/>

### 19. Android利用scheme协议进行跳转

scheme是一种页面跳转协议。

通过定义自己的scheme协议，可以非常方便的跳转App中的各个页面；

通过scheme协议，服务器可以定制化告诉App跳转到App内部页面。

**Scheme协议在Android中的使用场景**

* H5跳转到native页面
* 客户端获取push消息中后，点击消息跳转到App内部页面
* App根据URL跳转到另外一个App指定页面

**示例** 注册 SchemeActivity：

<activity android:name=".SchemeActivity">  
 <intent-filter>  
 //配置协议  
 <data android:scheme="scheme" android:host="mtime" android:path="/goodsDetail"/>  
 <category android:name="android.intent.category.DEFAULT"/>  
 <action android:name="android.intent.action.VIEW"/>  
 <category android:name="android.intent.category.BROWSABLE"/>  
 </intent-filter>  
 </activity>

在MainActivity里通过以下代码跳转到SchemeActivty：

String url = "scheme://mtime/goodsDetail?goodsId=2333";  
Intent intent = new Intent(Intent.ACTION\_VIEW, Uri.parse(url));  
startActivity(intent);

然后我们可以在SchemeActivity获取到请求的参数等等信息：

Uri uri = getIntent().getData();  
assert uri != null;  
String host = uri.getHost();  
String path = uri.getPath();  
String query = uri.getQuery();  
String param = uri.getQueryParameter("goodsId");  
Toast.makeText(this, host + " " + path + " " + query + " " + param, Toast.LENGTH\_SHORT).show();

### 20. MVC、MVP

**MVC模式**

**MVP模式**

MVP模式核心思想：

MVP把Activity中的UI逻辑抽象成View接口，把业务逻辑抽象成Presenter接口，Model类还是原来的Model。

这就是MVP模式，现在Activity的工作简单了，只是用来相应生命周期，其他工作都丢到Presenter中完成，从上图可以看出，Presenter是Model和View之间的桥梁，为了让结构变得更加简单，View并不能直接对Model进行操作，这也是MVP与MVC最大的不同之处。

**MVP模式的作用**

* 分离了视图逻辑和业务逻辑，降低了耦合
* Activity只处理生命周期的任务，代码变得更加简洁
* 视图逻辑和业务逻辑分别抽象到了View和Presenter的接口中去，提高了代码的可阅读性
* Presenter被抽象成接口，可以有多种具体的实现，所以方便进行单元测试
* 把业务逻辑抽象到Presenter中去，避免后台线程引用着Activity导致Activity的资源无法被系统回收从而引起内存泄露和OOM

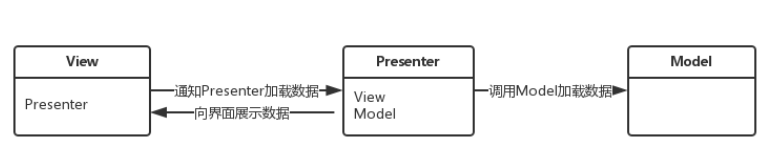
**MVP模式的使用**

从上述的UML图可以看出，使用MVP至少经历以下几个步骤：

1. 创建IPresenter接口，把所有业务逻辑的接口都放在这里，并创建它的实现类PresenterCompl（在这里可以方便的查看业务功能，由于接口可以有多种实现所以也方便写单元测试）
2. 创建IView接口，把所有视图逻辑的接口都放在这里，其实现类是当前的Activity/Fragment
3. 由UML图可以看出，Activity里包含了一个IPresenter，而PresenterCompl里又包含了一个IView并且依赖了Model。Activity里只保留了对IPresenter的调用，其他工作全部留到PresenterCompl中实现
4. Model并不是必须有的，但是一定会有View和Presenter

**总结** MVP模式的整个核心流程：

View与Model并不直接交互，而是使用Presenter作为View与Model之间的桥梁。其实Presenter中同时持有View层的interface的引用以及Model层的引用，而View层持有Presenter层引用。当View层某个界面需要展示某些数据的时候，首先会调用Presenter层的引用，然后Presenter层会调用Model层请求数据，当Model层数据加载成功之后会调用Presenter层的回调方法通知Presenter层数据加载情况，最后Presenter层在调用View层的接口将加载后的数据展示给用户。



### 21. SurfaceView

SurfaceView继承至View的，与View的主要区别在于：

* View主要适用于主动更新的情况下，而SurfaceView主要适用于被动更新，例如频繁的刷新
* View在主线程中对画面进行更新，而SurfaceView通常会通过一个子线程进行页面的刷新
* View在绘图时没有使用双缓存机制，而SurfaceView在底层机制中已经实现了双缓存机制

**总结一句话就是：如果自定义View需要频繁的刷新，或者刷新时数据处理量比较大，那么就可以考虑使用SurfaceView来取代View。**

/\*\*  
 \* 用Surface实现画板  
 \*/  
public class DrawBoardView extends SurfaceView implements SurfaceHolder.Callback, Runnable {  
  
 private SurfaceHolder mSurfaceHolder;  
 private Canvas mCanvas;  
 private volatile boolean mIsDrawing;  
 private Paint mPaint;  
 private Path mPath;  
  
 public DrawBoardView(Context context) {  
 super(context);  
 init();  
 }  
  
 public DrawBoardView(Context context, AttributeSet attrs) {  
 super(context, attrs);  
 init();  
 }  
  
 public DrawBoardView(Context context, AttributeSet attrs, int defStyleAttr) {  
 super(context, attrs, defStyleAttr);  
 init();  
 }  
  
 private void init() {  
 mSurfaceHolder = getHolder();  
 mSurfaceHolder.addCallback(this);  
 setFocusable(true);  
 setFocusableInTouchMode(true);  
 this.setKeepScreenOn(true);  
  
 mPaint = new Paint();  
 mPaint.setColor(Color.BLUE);  
 mPaint.setStyle(Paint.Style.STROKE);  
 mPaint.setStrokeWidth(10);  
 mPaint.setAntiAlias(true);  
 mPath = new Path();  
 }  
 @Override  
 public void surfaceCreated(SurfaceHolder holder) {  
 mIsDrawing = true;  
 new Thread(this).start();  
 }  
  
 @Override  
 public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {  
  
 }  
  
 @Override  
 public void surfaceDestroyed(SurfaceHolder holder) {  
 mIsDrawing = false;  
 }  
  
 @Override  
 public void run() {  
 long start = System.currentTimeMillis();  
 while (mIsDrawing) {  
 draw();  
 }  
 long end = System.currentTimeMillis();  
 // 50 - 100  
 if (end - start < 100) {//保证线程运行时间不少于100ms  
 try {  
 Thread.sleep(100 - (end - start));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 private synchronized void draw() {  
 try {  
 mCanvas = mSurfaceHolder.lockCanvas();  
 mCanvas.drawColor(Color.WHITE);  
 mCanvas.drawPath(mPath,mPaint);  
 } catch (Exception e) {  
 } finally {  
 if (mCanvas != null){  
 mSurfaceHolder.unlockCanvasAndPost(mCanvas);  
 }  
 }  
 }  
  
 @Override  
 public boolean onTouchEvent(MotionEvent event) {  
 switch (event.getAction()){  
 case MotionEvent.ACTION\_DOWN:  
 mPath.moveTo(event.getX(),event.getY());  
 break;  
 case MotionEvent.ACTION\_MOVE:  
 mPath.lineTo(event.getX(),event.getY());  
 break;  
 case MotionEvent.ACTION\_UP:  
  
 break;  
 }  
 return true;  
 }  
  
}

### 22. HandlerThread

HandlerThread继承了Thread，所以它本质上是一个Thread，与普通Thread的区别在于，它不仅建立了一个线程，并且创立了消息队列，有自己的looper，可以让我们在自己的线程中分发和处理消息，并对外提供自己这个Looper对象的get方法。

HandlerThread自带的Looper使它可以通过消息队列，来重复使用当前的线程，节省系统资源开销。这是他的优点也是缺点，每一个任务队列都是以队列的方式逐个被执行到，一旦队列中某个任务执行时间过长，那么就会导致后续的任务都会被延时执行。

**使用步骤：**

1. 创建HandlerThread实例并启动该线程

java HandlerThread handlerThread = new HandlerThread("myThread"); handlerThread.start();

1. 构造循环消息处理机制

Java private Handler.Callback mSubCallback = new Handler.Callback() { //该接口的实现就是处理异步耗时任务的，因此该方法执行在子线程中 @Override public boolean handleMessage(Message msg) { //doSomething 处理异步耗时任务的 mUIHandler.sendMessage(msg1); //向UI线程发送消息，用于更新UI等操作 } };

1. 构造子线程的Handler

```java //由于这里已经获取了workHandle.getLooper()，因此这个Handler是在HandlerThread线程也就是子线程中 childHandler = new Handler(handlerThread.getLooper(), mSubCallback);

button.setOnClickListener(new OnClickListener() { @Override public void onClick(View v) { //发送异步耗时任务到HandlerThread中,也就是上面的mSubCallback中 childHandler.sendMessage(msg); } }); ```

1. 构建UI线程Handler处理消息

java private Handler mUIHandler = new Handler(){ @Override public void handleMessage(Message msg) { //在UI线程中做的事情，比如设置图片什么的 } };

**实现原理**

我们知道，在子线程创建一个Handler，需要手动创建Looper，即先Looper.parpare()，完了在Looper.loop()。HandlerThread的run方法：

@Override  
 public void run() {  
 mTid = Process.myTid(); //获得当前线程的id  
 Looper.prepare();  
 synchronized (this) {  
 mLooper = Looper.myLooper();  
 //发出通知，当前线程已经创建mLooper对象成功，这里主要是通知getLooper方法中的wait  
 notifyAll();  
 }  
 //设置当前线程的优先级  
 Process.setThreadPriority(mPriority);  
 //该方法实现体是空的，子类可以实现该方法，作用就是在线程循环之前做一些准备工作，当然子类也可以不实现。  
 onLooperPrepared();  
 Looper.loop();  
 mTid = -1;  
 }

**特点：**

1. HandlerThread其实是Handler+Thread+Looper的组合，它本质上是一个Thread，因为它继承了Thread，相比普通的Thread，他不会阻塞，因为它内部通过Looper实现了消息循环机制，保证了多个任务的串行执行。缺点：效率低。
2. HandlerThread需要搭配Handler使用，单独使用的意义不大。HandlerThread线程中作具体的事情，必须要在Handler的callback接口中进行，他自己的run方法被写死了。
3. 子线程中的Handler与HandlerThread的联系是通过childHandler=new Handler(handlerThread.getLooper(),mSubCallback)执行的，也就是说，childHandler获得HandlerThread线程的Looper，这样他们两个就在同一个阵营了，这也就是创建Handler作为HandlerThread线程消息执行者，必须在调用start()方法之后的原因 —— HandlerThread.start()之后，run方法才能跑起来，Looper才得以创建，handlerThread.getLooper()才不会出错。
4. HandlerThread会将通过handleMessage传递进来的任务进行串行执行，由messageQueue的特性决定的，从而也说明了HandlerThread效率相对比较低。

### 23. IntentService

IntentService是继承并处理异步请求的一个类，其本质上是一个Service，因为它是继承至Service，所以开启IntentService和普通的Service一致。但是他和普通的Service不同之处在于它可以处理异步任务，在任务处理完之后会自动结束。另外，我们可以启动多次IntentService，而每一个耗时任务会以工作队列的方式在IntentService的onHandleIntent回调方法中执行，并且是串行执行。其实IntentService的内部是通过HandleThread和Handle来实现异步操作的。

### 24. 谈谈你对Application类的理解

首先，Application 在一个 Dalvik 虚拟机里面只会存在一个实例。为什么强调说是一个Dalvik虚拟机，而不是一个APP呢？那是因为，一个App有可能有多个Dalvik虚拟机，也就是传说中的多进程模式。在这种模式下，每一个Dalvik都会存在一个Application实例，它们之间没有关系，在A进程Application里面保存的数据不能在B进程的Application获取，因为它们根本不是一个对象。而且被隔离在两个进程里面，所以这里强调的是一个Dalvik虚拟机，而不是一个App。

Application有两个子类，一个是MultiDexApplication，如果你遇到了65535问题，可以选择继承至它，完成相关的工作。另外一个是在TDD（测试用例驱动）开发模式中使用Moke进行测试的时候用到的，可以来替代Application的Moke类MokeApplication。

在应用启动的时候，会首先调用Application.attach()，当然，这个方法是隐藏的，开发者能接触到的第一个被调用的方法其实是Application.onCreate()，我们通常会在这个方法里面完成各种初始化，比如图片加载库、网络请求库等初始化操作。但是最好别在这个方法里面进行太多的耗时操作，因为这会影响App的启动速度，所以对于不必要的操作可以使用异步操作、懒加载、延时加载等策略来减少对UI线程的影响。

除此之外，由于在Context中可以通过getApplicationContext()获取到Application对象，或者是通过Activity.getApplication()、Service.getApplication()获取到Application，所以可以在Application保存全局的数据，供所有的Activity或者Service使用。

使用上面的三种方法获取到的都是同一个Application对象，getApplicationContext()是在Context的实现类ContextImpl中具体实现的，而getApplication()则是在Activity和Service中单独实现的，所以他们的作用域不同，但是获取到的都是同一个Application对象，因为一个Dalvik虚拟机只有一个Application对象。

但是这里存在一个坑，那就是在低内存情况下，Application有可能被销毁，从而导致保存在Application里面的数据被销毁，最后程序错乱，甚至Crash。

所以当你想在Application保存数据的时候，请做好判null，或者选择其他方式保存你的数据，比如存储在硬盘上等等。

另外，在Application中存在几个有用的方法，比如onLowMemory()和onTrimMemory()（Activity里面也存在这两个方法），在这两个方法里面，我们可以实现自己的内存回收逻辑，比如关闭数据库链接、移除图片内存缓存等操作来降低内存消耗，从而降低被系统回收的风险。

最后，就是要注意Application的生命周期，他和Dalvik虚拟机生命周期一样长，所以在进行单例或者静态变量的初始化操作时，一定要用Application作为Context进行初始化，否则会造成内存泄漏的发生。使用Dialog的时候一般使用Activity的Context。但是也可以使用Application作为上下文，前提是你必须设置Window类型为TYPE\_SYSTME\_DAILOG，并且申请相关权限。这个时候弹出的Dialog是属于整个Application的，弹出这个Dialog的Activty销毁时也不会回收Dialog，只有在Application销毁时，这个Dialog才会自动消失。

### 25. Android为什么要设计出Bundle而不是直接使用HashMap来进行数据传递？

* Bundle内部是由ArrayMap实现的，ArrayMap的内部实现是两个数组，一个int数组是存储对象数据对应下标，一个对象数组保存key和value，内部使用二分发对key进行排序，所以在添加、删除、查找数据的时候，都会使用二分发查找，只适用于小数据量操作，如果在数据量比较大的情况下，那么它的性能将退化。而HashMap内部则是数组+链表结构，所以在数据量较小的时候，HashMap的Entry Array比ArrayMap占用更多的内存。因为使用Bundle的场景大多数为小数据量，所以相比之下，在这种情况下使用ArrayMap保存数据，在操作速度和内存占用上都具有优势。
* 另外一个原因，则是在Android中如果使用Intent来携带数据的话，需要数据是基本类型或者是可序列化类型，HashMap使用Serializable进行序列化，而Bundle则是使用Pracelable进行序列化。而在Android平台，更推荐使用Pracelable实现序列化，虽然写法复杂，但是开销更小，所以为了更加快速的进行数据的序列化和反序列化，系统封装了Bundle类，方便我们进行数据的传输。

### 26. SharedPreference在使用过程中有什么注意点？

**commit()和apply()的区别**

* 返回值

apply是没有返回值的，而commit返回boolean表明修改是否提交成功。

* 操作效率

apply是将修改数据提交到内存，而后异步真正提交到硬盘磁盘，而commit是同步的提交到硬盘磁盘上。

因此在多并发commit的时候，会等待正在处理的commit保存到磁盘后再操作，从而降低了效率。而apply只是原子的提交到内存，后面有调用apply的函数的将会直接覆盖前面的内存数据，从一定程度上提高了效

* 建议

如果对提交的结果不关心的话，建议使用apply，如果需要确保提交成功且有后续操作的话，还是使用commit。

**多进程表现**

在多进程中，如果要交换数据，不用使用SharedPreference，因为不同版本表现不稳定，推荐使用ContentProvider替代。

在有的文章中，有提到在多进程中使用SharePrefenerce添加标志位（MODE\_MULTI\_PROCESS）就可以了。这个标志位是2.3（API 9）之前默认支持的，但是在2.3之后，需要多进程的访问的情景，就需要显示的声明出来。

现在这个标志位被废弃了，因为在某些版本上表现不稳定。

### 27. SQLite有哪些可以优化的地方？

* 创建索引

索引有助于加快SELECT查询和WHERE子句，但是会减慢使用UPDATE和INSERT语句时的数据输入。索引可以创建或删除，但不会影响数据。

优点是加快了数据库检索的速度，包括对单表查询、分组查询、排序查询。

缺点是索引的创建和维护存在消耗，索引会占用物理内存，且随着数据量的增加而增加。在对数据库进行增删改时需要维护索引，所以会对增删改的性能存在影响。

使用场景：当某字段数据更新频率较低，查询频率较高，经常有范围查询（> < = >= <=）或order by、group by 发生时建议使用索引。并且选择度越大，建立索引越有优势，这里选择度指一个字段中唯一值的数量/总的数量。还有就是经常同时存取多列，且每列都含有重复值可考虑建立复合索引。

* 使用事务

原子性操作，要么全部成功，要么全部失败；在执行大量数据的插入操作时，避免频繁操作cursor，可以大幅减少insert操作时间，一般为1-2个数量级。

* 查询必要字段

查询时只取需要的字段和结果集，更多的结果集会消耗更多的时间以及内存，更多的字段会导致更多的内存消耗。

### 28. 嵌滑滑动机制

我们知道，Android的时间分发机制中，只要有一个控件消费了事件，其他控件就没办法在接受到这个事件了。因此，当有嵌套滑动场景时，我们需要自己动手解决事件冲突。

嵌套滑动机制的基本原理可以认为是事件共享，即当子控件接受到滑动事件，准备要滑动时，会先通知父控件（startNestedScroll）；然后在滑动之前，会先询问父控件是否要滑动（dispatchNestedPreScroll）；如果父控件响应该事件进行了滑动，那么就会通知子控件它具体消耗了多少滑动距离；然后交由子控件处理剩余的滑动距离；最后子控件滑动结束后，如果滑动距离还有剩余，就会在问一下父控件是否需要在继续滑动剩下的距离（dispatchNestedScroll）

### 29. RecyclerView 优化

1. 尽量减少Item布局嵌套，比如使用ConstraintLayout
2. 可以的话，写死ItemView的布局宽高，然后设置RecyclerView.setHasFixedSize(true)。当知道Adapter内Item的改变不会影响RecyclerView的宽高时，这样做可以避免重新计算大小。

但是，如果调用notifyDataSetChanged()，大小还是会重新计算（requestLayout）。当调用Adapter的增删改查方法，最后就会根据mHasFixedSize这个值来判断需不需要requestLayout()。

1. 根据需求修改RecyclerView默认的绘制缓存选项

recyclerView.setItemViewCacheSize(20)

recyclerView.setDrawingCacheEnabled(true)

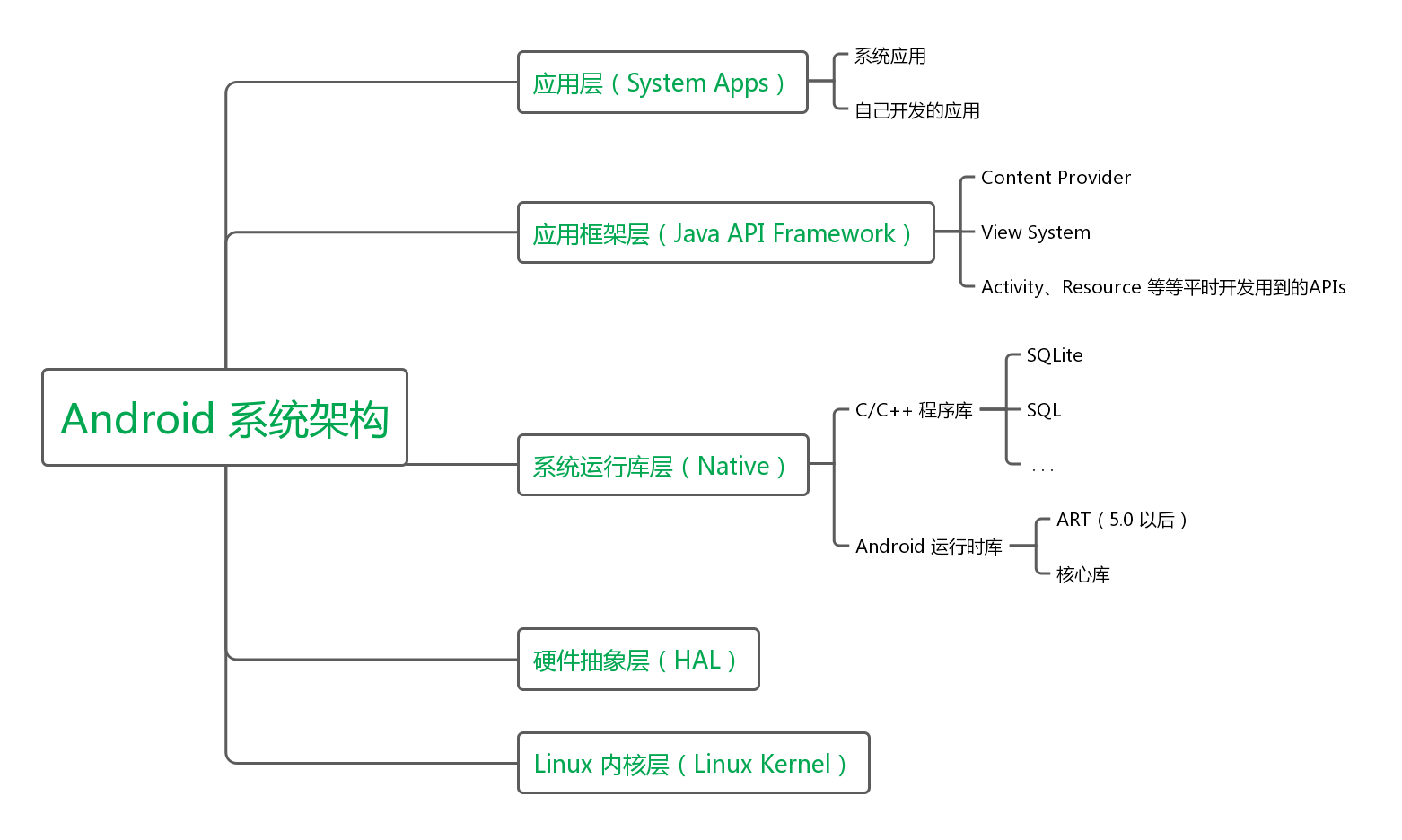
recyclerView.setDrawingCacheQuality(View.DRAWING\_CACHE\_QUALITY\_HIGH)

1. 在onBindViewHolder 或 getView 方法中，减少逻辑判断，减少临时对象创建。例如：复用事件监听，在其方法外部创建监听，可以避免生成大量的临时变量。
2. 避免整个列表的数据更新，只更新受影响的布局。例如，加载更多时，不使用notifyDataSetChanged，而是使用notifyItemRangeInserted(rangeStart,rangeEnd)

## Android 部分（二）高级知识点

### 1. Android 系统架构

* 应用层
* 应用框架层
* 系统运行库层
* 硬件抽象层
* Linux 内核层

​ 

### 2. Android 各个版本特性

1. Android 5

引入 Material Design、ART、优化通知栏、弃用 HTTP 类、增强 WebView、TLS / SSL 默认配置变更。

1. Android 6

运行时权限、低电耗模式、应用待机模式、指纹身份验证。

1. Android 7

SurfaceView、更多的表情支持、多窗口支持、通知增强功能。

1. Android 8

后台执行限制、后台位置限制、画中画模式、可下载字体、自动调整 TextView 的大小、自适应图标、新增权限。

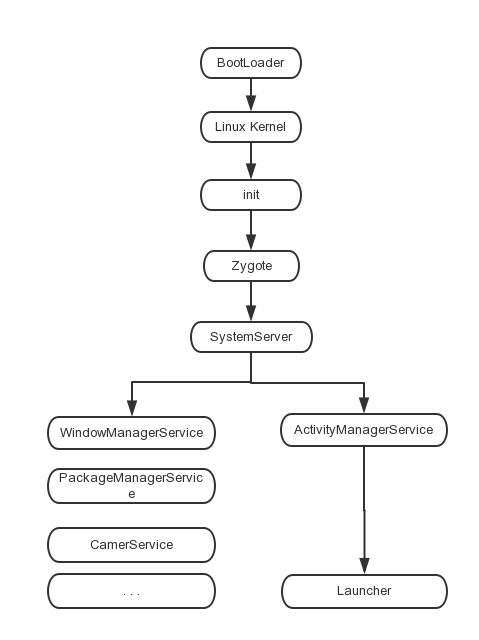
1. Android 9

对使用非 SDK 接口的限制、电池管理、强制执行 FLAG\_ACTIVITY\_NEW\_TASK 要求、利用 WI-FI 进行室内定位、全面屏支持、适用于可绘制图像和位图的 ImageDecoder、引入 AnimatedImageDrawable 类，用于绘制和显示 GIF 和 WebP 图像。

参考：[Android 各版本特性](https://www.jianshu.com/p/e8db7954a895)

### 3. Android 系统启动流程

1. init 进程启动过程
2. 创建和挂载启动所需的文件目录
3. 初始化和启动属性服务
4. 解析 init.rc 配置文件并启动 Zygote 进程
5. Zygote 进程启动过程
6. 创建 AppRuntime 并调用 start 方法，启动 Zygote 进程
7. 创建 Java 虚拟机并为 Java 虚拟机注册 JNI 方法
8. 通过 JNI 调用 ZygoteInit 的 main 方法进入 Zygote 的 Java 框架层
9. 通过 registerZygoteSocket 方法创建服务端 Socket，并通过 runSelectLoop 方法等待 AMS 的请求来创建新的应用程序进程
10. 启动 SystemServer 进程
11. SystemServer 处理过程
12. 启动 Binder 线程池，这样就可以与其他进程进行通信
13. 启动 SystemServiceManager，其用于对系统的服务进程创建、启动和生命周期管理
14. 启动各种启动服务
15. Launcher 启动过程



### 4. 应用程序进程启动过程

启动过程可以分为两步：

1. AMS 发送启动应用程序进程请求

AMS 如果想要启动应用程序进程，就需要向 Zygote 进程发送创建应用程序进程的请求，AMS 会通过调用 startProcessLocked 方法向 Zygote 进程发送请求。

1. Zygote 接收请求并创建应用程序进程

### 5. Activity 状态的保存与恢复

其实就是 onSaveInstanceState 和 onRestoreInstanceState 方法的使用。不过需要注意的是 onRestoreInstanceState 方法时，应当先调用 super 方法，这样由系统负责保存的部分才能够恢复，比如文本输入类型控件的输入文本以及光标位置。

整个保存 View 状态的流程如下：

1. 调用 Activity 的 onSaveInstanceState 方法
2. 该方法又调用 mWindow.saveHierarchyState，把返回的结果保存到 WINDOW\_HIERARCHY\_TAG 这个 key 对应的 value 中
3. mWindow 的实现类 PhoneWindow 当中，调用根布局的 saveHierarchyState 方法，这里面会从根布局按树形结构遍历，调用每个 ViewGroup / View 的 onSaveInstanceState。

于是，我们得出结论，保存的前提有两个：

1. View 的子类必须实现了 onSaveInstanceState
2. 它必须要有一个 ID，这个 ID 作为 Bundle 的 key，这也为我们实现自定义 View 时，需要保存状态提供了思路。

onSaveInstanceState 调用时机：

在 onPause() 方法之后，和 onStop() 方法没有既定的时序关系。

onRestoreInstanceState 调用时机：

在 onStart() 方法之后，onResume() 之前。

### 6. Android 动画框架实现原理

### 7. requestLayout、onLayout、onDraw、drawChild 区别与联系

### 8. requestLayout、invalidate、postInvalidate 的区别

1. requestLayout 会回掉 onMeasure、onLayout、onDraw（ViewGroup.setWillNotDraw(fasle)情况下）方法
2. invalidate 只会回掉 onDraw 方法
3. postInvalidate 只会回掉 onDraw 方法（可以在非 UI 线程中调用）

### 9. Activity、Window、View 的区别以及联系

### 10. Volley 的理解

### 11. 如何优化自定义 View

### 12. 低版本如何实现高版本 API

### 13. 描述一次网络请求的过程

### 14. HttpUrlConnection 与 OkHttp 的关系

### 15. Bitmap 的理解

### 16. Looper 架构

### 17. ActivityThread 的工作原理

### 18. AMS 的工作原理

### 19. WMS 的工作原理

### 20. 自定义 View 如何考虑机型适配

### 21. 自定义 View 的事件

### 22. LaunchMode 应用场景

### 23. SpareArray 原理

### 24. ContentProvider 是如何实现数据共享的

### 25. Service 与 Activity 的通信方式

### 26. IntentService 原理与作用

### 27. ApplicationContext 与 ActivityContext 的区别

### 28. SP 是进程同步的嘛？如何做到进程同步？

### 29. 谈谈多线程在 Android 中的应用

### 30. 进程和 Application 的生命周期

## 数据结构

* 线性表
* 栈和队列
* 树
* 图
* 散列查找
* 排序
* 海量数据处理

#### 线性表

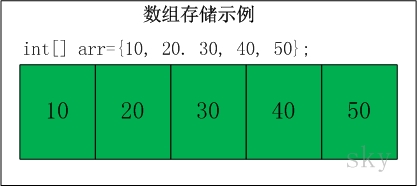
##### 概述

线性表是一种线性结构，它是具有相同类型的n(n>=0)个数据元素组成的有限序列。本章先介绍线性表的几个基本组成部分：数组、单向链表、双向链表。

**数组**

数组有上界和下界，数组的元素在上下界内是连续的。

存储10,20,30,40,50的数组的示意图如下：

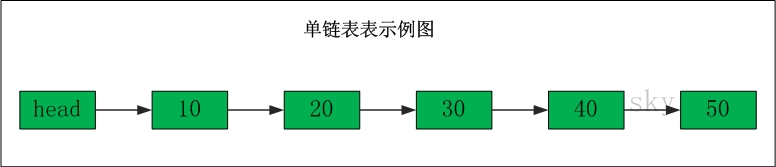


数组的特点是：

数据是连续的；随机访问速度快。数组中稍微复杂一点的是多维数组和动态数组。对于C语言而言，多维数组本质上也是通过一维数组实现的。至于动态数组，是指数组的容量能动态增长的数组；对于Java而言，Collection集合中提供了ArrayList和Vector。

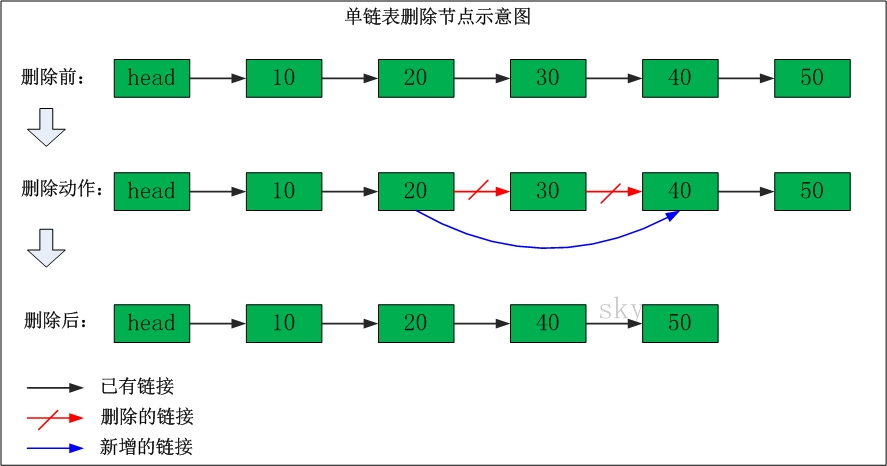
**单向链表**

单链表是链表的一种，它由节点组成，每个节点都包含下一个节点的指针。

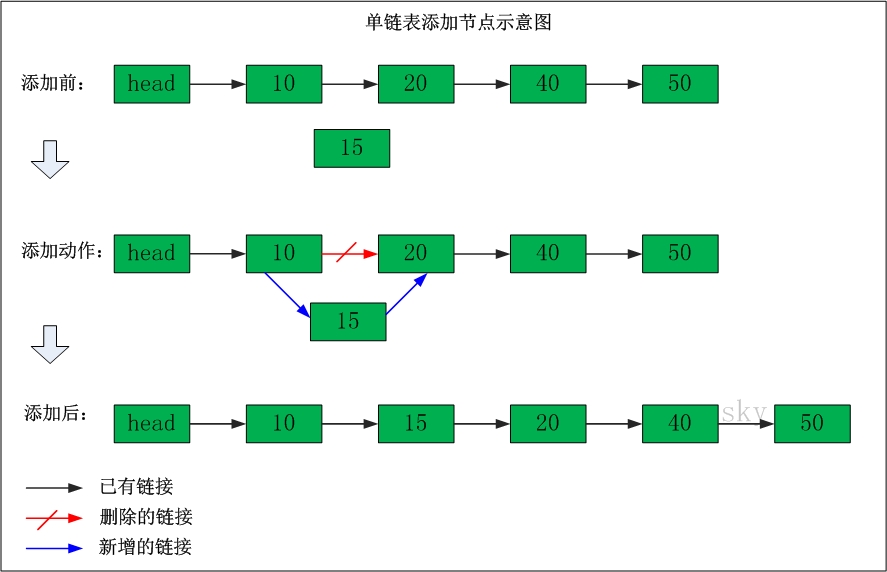


表头为空，表头的后继节点是“节点10”（数据是10的节点），“节点10”的后继节点是“节点20”（数据为20的节点）......

单链表删除节点：



单链表添加节点：

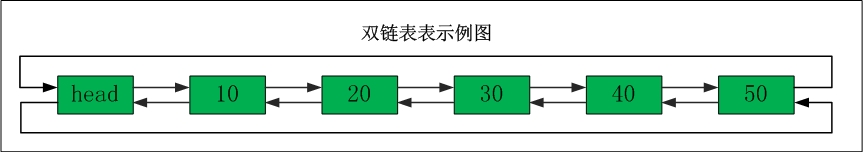


单链表的特点是：

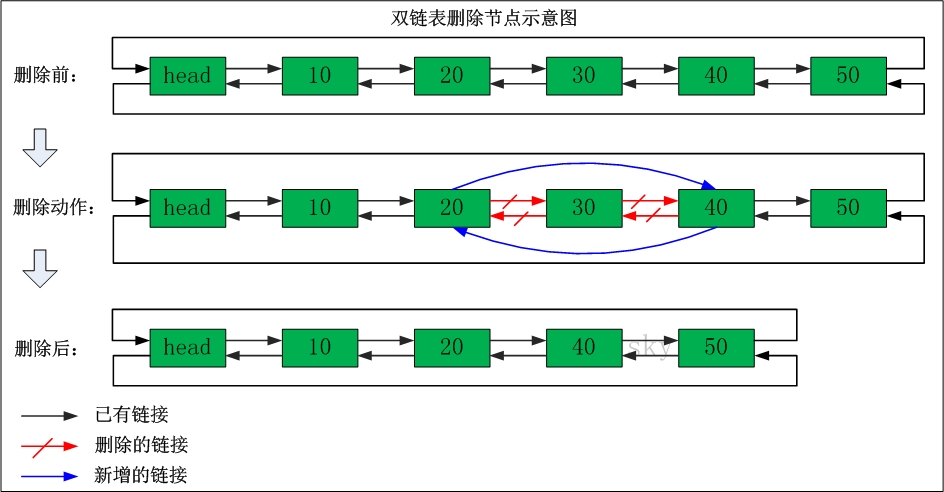
节点的链接方向是单向的；相对于数组来说，单链表的随机访问速度较慢，但是单链表删除 / 添加数据的效率很高。

**双向链表**

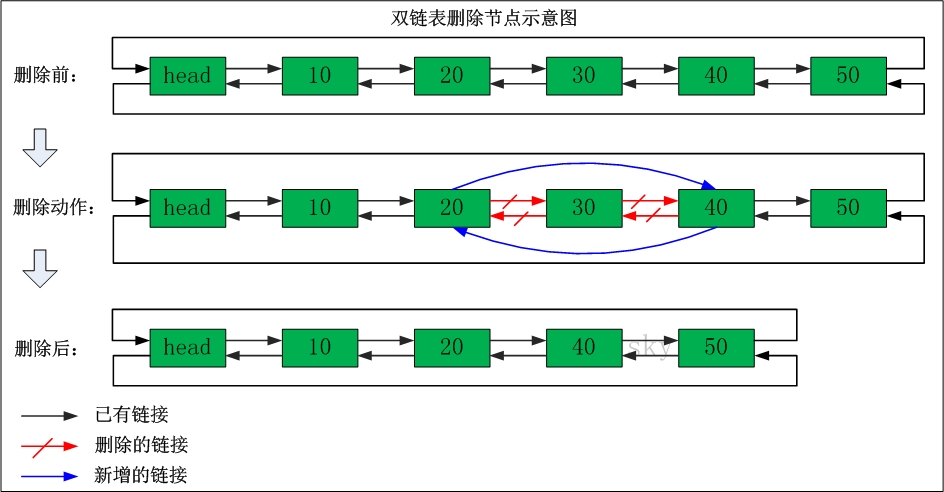
双向链表是链表的一种，和单链表一样，双链表也是由节点组成，它的每个数据节点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个节点开始，都可以很方便的访问它的前驱结点和后继结点，一般我们都构造双向循环链表。



双链表删除节点：



双链表添加节点：



**总结**

链表（LinkedList）是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存下一个节点的指针，简单来说链表并不像数组那样将数组存储在一个连续的内存地址空间里，它们可以不是连续的因为他们每个节点保存着下一个节点的引用地址，所以较数组来说是一个优势。

* 链表增删元素的时间复杂度为O(1)，查找一个元素的时间复杂度为O(n)
* 单链表不用像数组那样预先分配存储空间的大小，避免了空间浪费
* 单链表不能进行回溯操作

**单链表的基本操作**

获取单链表的长度：

由于单链表的存储地址是不连续的，链表并不具有直接获取链表长度的功能，对于一个链表的长度我们只能依次去遍历链表的节点，直到找到某个节点的下一个节点为空的时候得到链表的总长度。注意这里的出发点并不是一个空链表然后依次添加节点后，然后去读取已经记录的节点个数，而是已知一个链表的头节点然后去获取这个链表的长度。

public int getLength(Node head){  
   
 if(head == null){  
 return 0;  
 }  
   
 int len = 0;  
 while(head != null){  
 len++;  
 head = head.next;  
 }   
 return len;   
}

查询指定索引的节点值或指定值的节点值的索引：

由于链表是一种非连续性的存储结构，节点的内存地址不是连续的，也就是说链表不能像数组那样可以通过索引值获取索引元素的位置。所以链表的查询的时间复杂度要是O(n)级别的，这点和数组查询指定值的元素位置是相同的，因为你要查找的东西在内存中的存储地址都是不一定的。

/\*\* 获取指定角标的节点值 \*/  
 public int getValueOfIndex(Node head, int index) throws Exception {  
  
 if (index < 0 || index >= getLength(head)) {  
 throw new Exception("角标越界！");  
 }  
  
 if (head == null) {  
 throw new Exception("当前链表为空！");  
 }  
  
 Node dummyHead = head;  
  
 while (dummyHead.next != null && index > 0) {  
 dummyHead = dummyHead.next;  
 index--;  
 }  
  
 return dummyHead.value;  
 }  
  
   
 /\*\* 获取节点值等于 value 的第一个元素角标 \*/  
 public int getNodeIndex(Node head, int value) {  
   
 int index = -1;  
 Node dummyHead = head;  
   
 while (dummyHead != null) {  
 index++;  
 if (dummyHead.value == value) {  
 return index;  
 }  
 dummyHead = dummyHead.next;  
 }  
   
 return -1;  
 }

**链表添加一个元素**

链表的插入操作分为头插法、尾插法和随机节点插入法。当然数据结构讲的时候也是针对一个已经构造好的（保存了链表头部节点和尾部节点引用）

## 计算机网络

### 1. TCP 和 UDP 的区别

TCP：

传输控制协议，提供的是面向连接、可靠的字节流服务，传输数据前经过三次握手建立连接，保证数据传输的可靠性，但效率比较低，一般用于数据传输安全性较高的场合。

UPD：

用户数据报协议，是一个简单的面向数据报的运输层协议，面向无连接。UDP不提供可靠性，数据传输可能发生错序，丢包，但效率1较高，一般用于对于实时性要求较高的场合。

**总结：**

* UDP在传送数据之前不需要先建立连接。远程主机运输层在收到UDP报文后，不需要给出任何确认。因此UDP不提供可靠交付，但是效率高。TCP则提供面向连接的服务，在传送数据之前必须先建立连接，数据传送结束后要释放连接。TCP要提供可靠的、面向连接的运输服务，因此不可避免的增加了许多的开销，如确认、流量控制等等。
* TCP和UDP在发送报文时所采用的方式完全不同。TCP并不关心进程一次把多长的报文发送到TCP的缓存中，而是根据对方给出的窗口值和当前网络拥塞程度决定一个报文段包含多少字节，而UDP发送报文长度是应用进程给出的。如果应用进程传送到TCP缓存的数据块太长，TCP就划分短一些在传送。若过短也可以等待积累足够多的字节后再构成报文段发送出去。
* UDP程序结构比较简单，它的首部最少为8个字节而TCP最少为20字节。
* UDP不保证数据的顺序结构，而TCP必须保证数据的顺序结构。
* TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流；UDP是面向报文的，UDP没有阻塞控制，因此网络出现阻塞不会使源主机的发送速率降低。

### 2. Http1.1和Http1.0及2.0的区别

**HTTP1.1与1.0的区别**

* 长连接 HTTP1.0协议使用非持久连接，即在非持久连接下，一个TCP连接只传输一个Web对象。

HTTP1.1支持持久连接，也就是说长连接，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟。

一个包含有许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接。HTTP1.1还允许客户端不要等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样就显著的减少了整个下载过程所需要的时间。

HTTP1.0需要使用keep-alive参数来告知服务器端要建立一个长连接，而HTTP1.1默认支持长连接。HTTP是基于TCP/IP协议的，创建一个TCP连接是需要经过三次握手的，有一定的开销，如果每次通讯都要重新建立连接的话，对性能有影响。因此最好能维持一个长连接。

* 缓存
* 节约带宽

HTTP/1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象传送过来。例如，客户端只需要显示一个文档的部分内容，又比如下载大文件需要支持断点续传，而不是发送断连后不得不重新下载完整的包。

HTTP1.1中在请求消息中引入了range头域，他只允许请求资源的某个部分。在响应消息中Content-Range头域声明了返回的这部分对象的偏移值和长度。如果服务器响应的返回了对象所请求范围的内容，则响应码为206，它可以防止Cache将响应误以为是一个完整的对象。

节省带宽资源的一个非常有效的做法就是压缩要传送的数据。

* Host域
* 错误提示

**HTTP1.1与HTTP2.0的区别**

* 多路复用

HTTP2.0使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。

当然HTTP1.1也可以多建立几个TCP连接，来支持处理更多并发的请求，但是创建TCP连接本身也是有开销的。

TCP连接有一个预热和保护的过程，先检查数据是否传送成功，一旦成功过后，则会慢慢增加传输速度，因此对应瞬时并发的链接，服务器的响应就会变慢。所以最好能使用一个建立好的连接，并且这个连接支持瞬时并发的请求。

* 数据压缩

我们知道，HTTP请求和响应都是由状态行、请求/响应头部，消息主体三部分组成的。一般而言，消息主体都会经过gzip压缩，或者本身传输的就是压缩过后的二进制文件（如图片、音频等），但是状态行和头部多是没有经过任何压缩，而是以纯文本的方式进行传输的。

然而，随着web功能越来越多，请求数量越来越多，随之而来的就是头部的流量越来越多，并且在建立初次连接之后的链接也要发送User-Agent等信息，实在是一种浪费，因此，HTTP2.0提出了对请求和响应的头部进行压缩，而不只是压缩主体部分，这种压缩方式就是HAPCK。

* 服务器推送

为了改善延迟，HTTP2.0引入了Server Push，它允许服务端推送资源给浏览器，在浏览器明确的请求之前，一个服务器经常知道一个页面需要很多附加资源，在它响应浏览器第一个请求的时候，就可以开始推送这些资源。这允许服务端充分的利用一个可能空闲的网络，改善页面加载时间。

### 3. DNS解析过程

<https://www.jianshu.com/p/7e268c559aff>

DNS域名解析服务器。解析过程：

1. 当用户在浏览器输入一个域名的时候，最先浏览器会从自己的缓存中寻找指定的结果。如果找到了域名对应的IP则域名解析完成。这个缓存大小是有限的，另外每一条结果都有过期时间，这个过期时间通过TTL属性来指定。
2. 如果在浏览器中的缓存没有命中，则会在系统的缓存中查找这个域名是否有对应的DNS解析结果，如果有则域名解析完成。这个缓存通常是以文件的方式来保存，比如windows下通过C:32，Linux则是etc/named/config文件，通过编辑这个文件我们能把域名映射到任意一个IP中。

如果前面两个流程都没有找到指定域名的解析结果，那么下面就要进行真正的域名解析了。为什么叫做真正的域名解析呢？因为前面的都是在本机中完成的，下面的流程就要依赖外部服务器来查找指定的域名的解析结果

1. 系统缓存未命中之后会把这个域名提交到指定的LDNS服务器中（本地DNS服务器），这个服务器就是你计算机设定的DNS服务器。如果你在学校的网络中，这个DNS服务器一定在你学校里，如果你在小区的网络，这个DNS服务器通常是运营商提供的。这个域名解析服务器缓存了大量的域名的DNS解析结果，通常80%的DNS解析需求在这一步就满足了，所以LDNS完成了大部分的DNS解析任务。
2. 如果指定的域名在LDNS服务器的缓存中仍然没有命中，LDNS会向ROOT Service发送请求。
3. ROOT Server会返回给LDNS一个指定域名对应的主域名服务器gTL的地址D，gTLD是顶级域名服务器，如com、cn、org等等。
4. LDNS接下来会向这个gTLD服务器发送域名解析请求。
5. 接收请求的gTLD会返回给LDNS一个该域名对应的Name Server服务器地址，这个Name Server通常就是你注册的域名服务器。例如你在某个域名服务提供商申请的域名，这个域名就由他们的服务器解析。
6. NAME Server会把指定域名的IP和一个TTL返回给LDNS。
7. LDNS会把这个结果缓存下来，缓存的过期时间由TTL来决定
8. 然后LDNS再把这个结果返回给用户，DNS解析结束。

从上面来看如果LDNS中没有查找到指定域名的对应IP，则需要很长的时间来获取解析结果。但是一旦解析结果被缓存了，下次在请求同样的域名就不会那么慢了。

### 4. HTTP请求报文和响应报文

[HTTP请求报文和响应报文](http://www.cnblogs.com/biyeymyhjob/archive/2012/07/28/2612910.html)

HTTP报文是面向文本的，报文中的每一个字段都是一些ASCII码串，各个字段的长度是不确定的。HTTP有两类报文：请求报文和响应报文。

**HTTP请求报文**

一个HTTP请求报文由请求行、请求头、空行和请求数据四个部分组成。



* 请求行

请求行由请求方法、URL字段、HTTP协议这三个字段组成，它们用空格分隔。例如：GET /index.html HTTP/1.1

* 请求头部

请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号分隔。请求头部通知服务器有关于客户端请求的信息，典型的请求头有：

User-Agent：产生请求的浏览器类型。

Accept：客户端可识别的内容类型列表。

Host：请求的主机名，允许多个域名同一个IP地址，即虚拟主机。

* 空行

最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

* 请求数据

请求数据不在GET方法中使用，而是在POST方法中使用。POST方法适用于需要客户端填写表单的场合。与请求数据相关的最常用的请求头是Content-Type和Content-Length。

**HTTP响应报文**

HTTP响应报文也是由三个部分组成，分别是：状态行、请求包头、响应正文。类似以下：

＜status-line＞  
  
＜headers＞  
  
＜blank line＞  
  
[＜response-body＞]

在相应中唯一真正的区别在于第一行中用状态信息代替了请求信息。状态行通过提供一个状态码来说明请求的资源情况。

状态行格式如下：

HTTP-Version Status-Code Reason-Phrase CRLF

其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。状态码由三位数字组成，第一个数字定义了响应的类别，且有五种可能的取值。

1. 1xx：指示信息，表示请求已接收，继续处理
2. 2xx：成功，表示请求已被成功接收
3. 3xx：重定向，要完成的请求必须进行更进一步的操作
4. 4xx，客户端错误，请求有语法错误或者请求无法实现
5. 5xx，服务器端错误，服务器未能实现合法的请求

常见的状态码：

100：客户端必须继续发出请求

101：客户端要求服务器根据请求转换HTTP协议版本

200：响应成功

206：客户端表明自己只需要目标URL上的部分资源的时候返回的。

301：永久重定向

302：临时重定向

503：服务不可用

504：网关超时

HTTP响应报文的例子：

HTTP/1.1 200 OK  
Date: Sat, 31 Dec 2005 23:59:59 GMT  
Content-Type: text/html;charset=ISO-8859-1  
Content-Length: 122  
  
＜html＞  
＜head＞  
＜title＞Wrox Homepage＜/title＞  
＜/head＞  
＜body＞  
＜!-- body goes here --＞  
＜/body＞  
＜/html＞

### 5. 简述HTTP

HTTP协议即超文本传输协议，是用于从万维网服务器传输超文本到本地浏览器的传送协议。HTTP是一个基于TCP/IP通信协议来传递数据。HTTP是一个属于应用层的面向对象的协议，由于其简单、快速的方式，适用于分布式超媒体信息系统。

**工作原理** ​ HTTP协议工作于客户端-服务端架构上。浏览器作为HTTP客户端通过URL向HTTP服务器即WEB服务器发送请求，WEB服务器接收请求后，向客户端发送响应信息。HTTP三点注意事项：

* 简单快速灵活：客户端向服务器端发送请求后，只需要传送请求方法和路径，HTTP允许传输任意类型的数据对象，正在传输的类型由Content-Type加以标记。
* HTTP是无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户端的请求，并收到客户端应答后，即断开连接。采用这种方式可以节省传输时间。
* HTTP是媒体独立的：这意味着，只要客户端和服务器知道如何处理的数据内容，任何类型的数据都可以通过HTTP发送。客户端以及服务器指定使用合适的MIME-type内容类型。
* HTTP是无状态：HTTP协议是无状态协议，无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另外一方面，在服务器不需要先前信息时它的应答就较快。