Android面试之百题经典

第一章 Java相关答案

一、线程中sleep和wait的区别

1，这两个方法来自不同的类分别是Thread和Object

2，最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法

3，wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

二、Thread中的start()和run()方法有什么区别

start()方法:启动一个线程，调用该Runnable对象的run()方法，不能多次启动一个线程

run()方法:在本线程内（常为main线程）调用该Runnable对象的run()方法，可以重复多次调用，也就是说，直接调用run()方法和调用普通方法一样（并不能启动一个新的线程，仍然按顺序执行，同步执行）

三、关键字final和static是怎么使用的

final类不能继承，没有子类，final类中的方法默认是final的

final方法不能被子类方法覆盖，但可以被继承

final成员变量表示常量，只能被赋值一次，赋值后值不再改变

final不能用于修饰构造方法

static表示“全局”或者“静态”的意思，用来修饰成员变量和成员方法，也可以形成静态static代码块，被static修饰的成员变量和成员方法独立于该类的任何对象

用public修饰的static成员变量和成员方法本质是全局变量和全局方法，当声明它类的对象时，不生成static变量的副本，而是类的所有实例共享同一个static变量

static final用来修饰成员变量和成员方法，可以理解为“全局变量”

四、String,StringBuffer,StringBuilder区别

String是Java中的类，在 Java 中字符串属于对象，Java 提供了 String 类来创建和操作字符串。String的值是不可变的，这就导致每次对String的操作都会生成新的String对象

当对字符串进行修改的时候，StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。StringBuffer提供append和add方法，可以将字符串添加到已有序列的末尾或指定位置，它的本质是一个线程安全的可修改的字符序列，把所有修改数据的方法都加上了synchronized。但是保证了线程安全是需要性能的代价的

StringBuilder 类在 Java 5 中被提出，它和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）

一般在多线程中操作字符串缓冲区下的大量数据使用StringBuffer；单线程操中操作字符串缓冲区下的大量数据使用StringBuilder

五、Java中重载和重写的区别

重载就是在同一类中允许同时存在一个以上的同名方法，但方法的参数类型不同

重写（重置、覆盖）是子类重新定义父类中已经定义的方法，即子类重写父类方法

六、equals跟==的区别

==是一个比较运算符，基本数据类型比较的是值，引用数据类型比较的是地址值（比较地址值即是指是否为同一个对象的引用）

equals()是Object基类一个方法，只能比较引用数据类型。重写前比较的是地址值，重写后比一般是比较对象的属性，如：String类中的equals()方法是被重写过的，比较的是对象属性

七、抽象类跟接口的区别

接口是对动作的抽象，抽象类是对根源的抽象。抽象类表示的是，这个对象是什么。接口表示的是，这个对象能做什么

所以，在高级语言上，一个类只能继承一个类（抽象类），但是可以实现多个接口(吃饭接口、走路接口)

接口是抽象类的变体，接口中所有的方法都是抽象的。而抽象类是声明方法的存在而不去实现它的类

接口定义方法，不能实现，而抽象类可以实现部分方法

接口中基本数据类型为static 而抽类象不是的

当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口

八、抽象类能否实例化，为什么

不能

抽象类是一个不完整的类实际上抽象类更多提供的是一个框架的功能，参数比接口更详细些。因为在设计时，考虑到某些类所具备的信息不足以实例化一个对象，才设计成抽象的

内存分配问题：抽象类只分配了在栈中的引用，没有分配堆中的内存。程序都有一个代码段,再内存中需要占据一定的内存,而抽象类没有具体的实现方法,无法具体的给它分配内存空间,所以为了安全,不JAVA不允许抽象类,接口直接实例化

九、Java中的锁

Java多线程加锁的方式：

1，synchronized关键字

2，Java.util.concurrent包中的lock接口和ReentrantLock实现类

采用synchronized不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象

Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现

Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；（I/O和Synchronized都能相应中断，即不需要处理interruptionException异常）

十、Http https区别，https的实现原理

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少

HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL

HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。具体说就是，一是建立SSL连接（信息加密连接），二是CA证书认证

十一、Http位于TCP/IP模型中的第几层？为什么说Http是可靠的数据传输协议？

从下往上第4层传输层(按5层分)，整个5层为：5，应用层(HTTP协议，支持电子邮件的SMTP协议，支持文件传送的FTP协议，DNS，POP3，SNMP，Telnet等等)，4，传输层(传输控制协议TCP、用户数据包协议UDP)，3，网络层(IP，ICMP，IGMP，ARP，RARP)，2，链路层(在两个相邻结点之间传送数据时，数据链路层将网络层交下来的IP数据报组装成帧(framing)，在两个相邻结点之间的链路上“透明”地传送帧中的数据。每一帧包括数据和必要的控制信息(如同步信息、地址信息、差错控制等))，1，物理层(在物理层上所传数据的单位是比特。物理层的任务就是透明地传送比特流)

由于Http是在传输层基于TCP协议的，而TCP又是面向连接的可靠协议，所以Http是可靠的传输协议

十二、HTTP链接的特点

1，支持客户/服务器模式

2，简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快

3，灵活：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

4，无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间

5，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。另一方面，在服务器不需要先前信息时它的应答就较快

十三、TCP和UDP的区别

1，连接方面区别：

TCP面向连接（如打电话要先拨号建立连接）

UDP是无连接的，即发送数据之前不需要建立连接

2，安全方面的区别：

TCP提供可靠的服务，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达

UDP尽最大努力交付，即不保证可靠交付

3，传输效率的区别

TCP传输效率相对较低

UDP传输效率高，适用于对高速传输和实时性有较高的通信或广播通信

4，连接对象数量的区别

TCP连接只能是点到点、一对一的

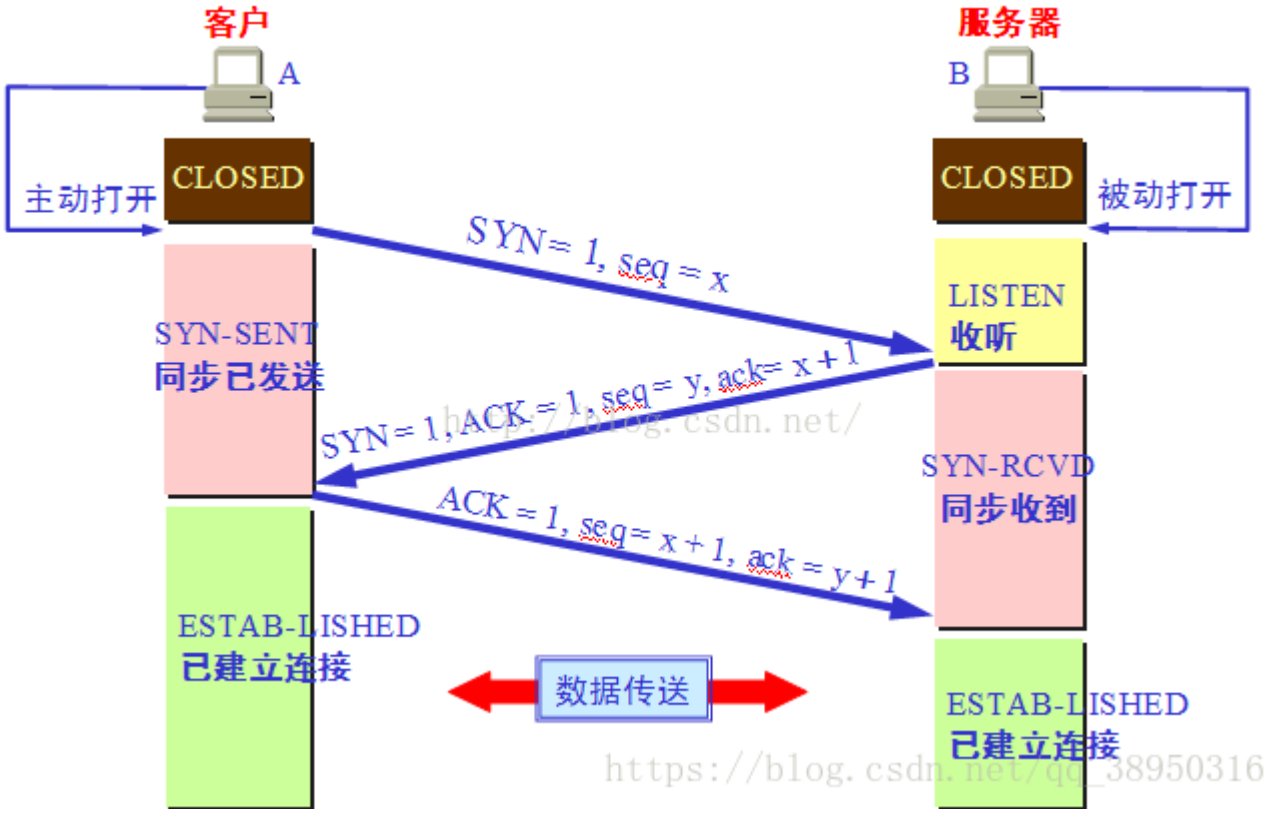
UDP支持一对一，一对多，多对一和多对多的交互通信

十四、Socket建立网络连接的步骤

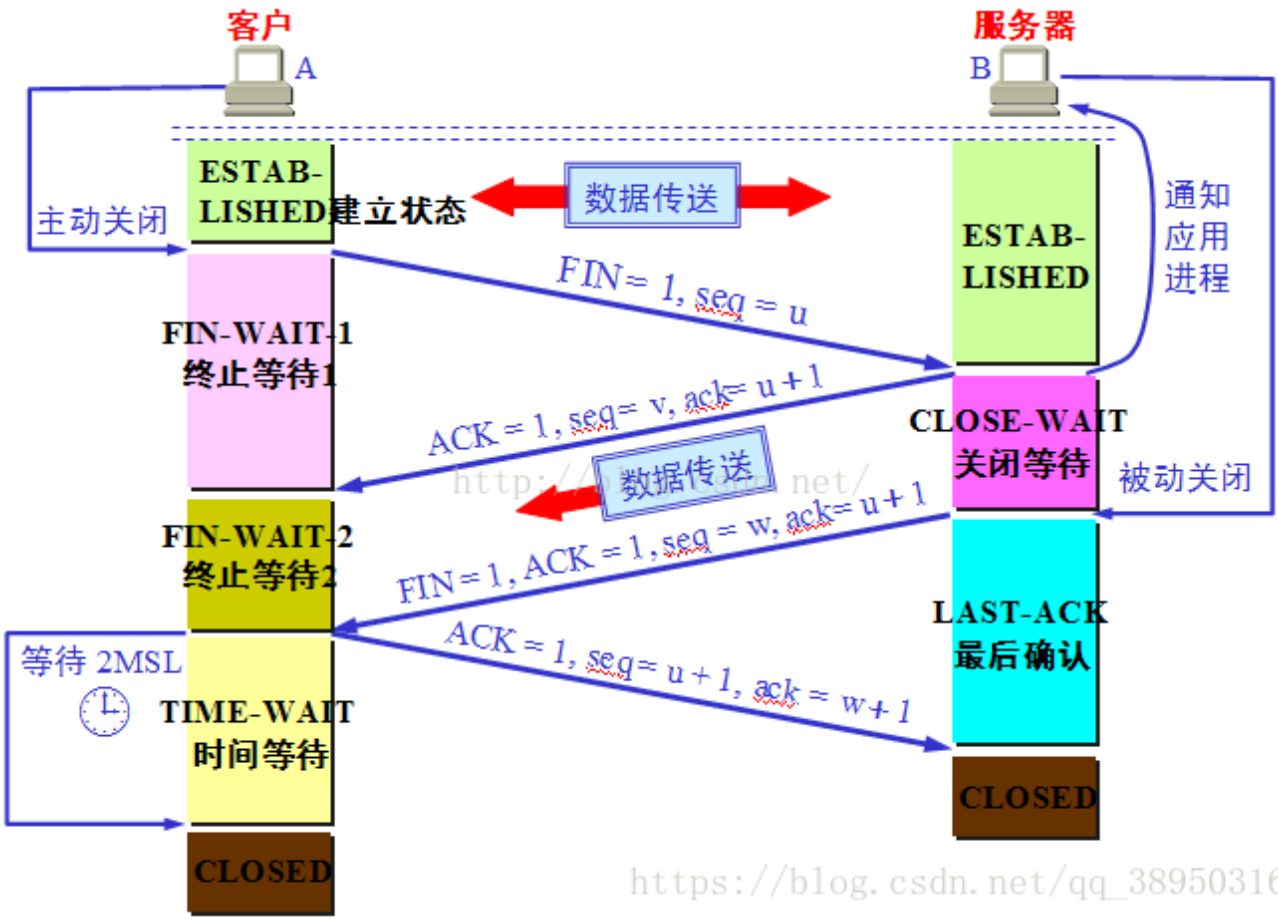
建立Socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket ，另一个运行于服务器端，称为ServerSocket 。套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认

十五、Tcp / IP三次握手，四次挥手

三次握手：



四次挥手：



十六、java注解，反射，泛型的理解与作用

注解：注解就类似一个标签，给字段，方法，类，或者给注解本身，做一个标记

javac编译工具、开发工具以及其他程序可以利用反射来了解你的类以及各种元素上有没有何种标记，就可以去做相应的事

标记可以加在包、类型（Type，包括类，枚举，接口）、字段、方法、方法的参数以及局部变量上面

反射：就是反过来获取类的信息。以前都是java文件最后生成class文件，现在反过来，在有Class对象的前提下，获取整个类的信息

泛型 ：Java采用泛型擦除机制来引入泛型。Java中的泛型仅仅是给编译器Javac使用的，确保数据的安全性和免去强制类型转换的麻烦。在不知道要操作什么数据类型的时候，用它代替。像装数据的容器数组，集合等，或者即将操作的对象，都不清楚时，用它代替

十七、java多个线程如何同时请求，返回的结果如何等待所有线程数据完成后合成一个数据线程种类

同时请求：

1， 创建n个线程，加一个闭锁CountDownLatch，开启所有线程，待所有线程都准备好后，按下开启按钮，就可以真正的发起并发请求了

2，使用CyclicBarrier

等待所有线程数据完成后合成一个数据线程种类：

1，放置一个公用的static变量，每个线程处理完上去累加下结果，然后后面用一个死循环，去数这个结果

2， 使用CountDownLatch

3，借助FutureTask，达到类似的效果，其get方法会阻塞线程，等到该异步处理完成。缺点就是，FutureTask调用的是Callable，必须要有返回值

4，使用Thread的Join方法

5，使用同步屏障CyclicBarrier

十八、JAVA GC原理

如何判断一个对象已经死去：可达性分析

GC算法：标记-清除(Mark-Sweep)算法 复制算法 标记-整理(Mark-Compat)算法 分代收集(Generational Collection)算法

内存分配

对象优先在Eden分配

大对象直接进老年代

长期存活的对象将进入老年代

十九、volatile的作用和原理

1，保持内存可见性

volatile如何保持内存可见性

Java通过几种原子操作完成工作内存和主内存的交互：

1， lock：作用于主内存，把变量标识为线程独占状态

2， unlock：作用于主内存，解除独占状态

3，read：作用主内存，把一个变量的值从主内存传输到线程的工作内存

4，load：作用于工作内存，把read操作传过来的变量值放入工作内存的变量副本中

5， use：作用工作内存，把工作内存当中的一个变量值传给执行引擎

6， assign：作用工作内存，把一个从执行引擎接收到的值赋值给工作内存的变量

7， store：作用于工作内存的变量，把工作内存的一个变量的值传送到主内存中

8，write：作用于主内存的变量，把store操作传来的变量的值放入主内存的变量中

volatile的特殊规则就是：

read、load、use动作必须连续出现

assign、store、write动作必须连续出现

所以，使用volatile变量能够保证:

每次读取前必须先从主内存刷新最新的值

每次写入后必须立即同步回主内存当中

也就是说，volatile关键字修饰的变量看到的随时是自己的最新值。线程1中对变量v的最新修改，对线程2是可见的

2，防止指令重排

下面是基于保守策略的JMM内存屏障插入策略：

在每个volatile写操作的前面插入一个StoreStore屏障

在每个volatile写操作的后面插入一个StoreLoad屏障

在每个volatile读操作的后面插入一个LoadLoad屏障

在每个volatile读操作的后面插入一个LoadStore屏障

二十、ReentrantLock原理

ReentrantLock主要利用CAS+CLH队列来实现。它支持公平锁和非公平锁，两者的实现类似

CAS：Compare and Swap，比较并交换。CAS有3个操作数：内存值V、预期值A、要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。该操作是一个原子操作，被广泛的应用在Java的底层实现中。在Java中，CAS主要是由sun.misc.Unsafe这个类通过JNI调用CPU底层指令实现

CLH队列：带头结点的双向非循环链表

ReentrantLock的基本实现可以概括为：先通过CAS尝试获取锁。如果此时已经有线程占据了锁，那就加入CLH队列并且被挂起。当锁被释放之后，排在CLH队列队首的线程会被唤醒，然后CAS再次尝试获取锁。在这个时候，如果：

非公平锁：如果同时还有另一个线程进来尝试获取，那么有可能会让这个线程抢先获取

公平锁：如果同时还有另一个线程进来尝试获取，当它发现自己不是在队首的话，就会排到队尾，由队首的线程获取到锁

二十一、Java为什么要推出HashMap，它是如何解决hash冲突的

常规的线性存储，你若需要找到其中的某个元素，就需要遍历这个链表或者数组，而遍历的同时需要让链表中的每一个元素都和目标元素做比较，相等才返回，Java里面用equals或者==。这对性能是毁灭性的伤害

HashMap的优势：Hash算法就是根据某个算法将一系列目标对象转换成地址，当要获取某个元素的时候，只需要将目标对象做相应的运算获得地址，直接获取

如果存在相同的hashcode，那么他们确定的索引位置就相同，这时判断他们的key是否相同，如果不相同，这时就是产生了hash冲突。 Hash冲突后，那么HashMap的单个bucket里存储的不是一个 Entry，而是一个 Entry 链。系统只能必须按顺序遍历每个 Entry，直到找到想搜索的 Entry 为止——如果恰好要搜索的 Entry 位于该 Entry 链的最末端（该 Entry 是最早放入该 bucket 中），那系统必须循环到最后才能找到该元素。

二十二、ArrayMap跟SparseArray在HashMap上面的改进

HashMap中默认的存储大小就是一个容量为16的数组，所以当我们创建出一个HashMap对象时，即使里面没有任何元素，也要分别一块内存空间给它，而且，我们再不断的向HashMap里put数据时，当达到一定的容量限制时（这个容量满足这样的一个关系时候将会扩容：HashMap中的数据量>容量*加载因子，而HashMap中默认的加载因子是0.75），HashMap的空间将会扩大，而且扩大后新的空间一定是原来的2倍。假如我们有几十万、几百万条数据，那么HashMap要存储完这些数据将要不断的扩容，而且在此过程中也需要不断的做hash运算，这将对我们的内存空间造成很大消耗和浪费，而且HashMap获取数据是通过遍历Entry[]数组来得到对应的元素，在数据量很大时候会比较慢，所以在Android中，HashMap是比较费内存的，我们在一些情况下可以使用SparseArray和ArrayMap来代替HashMap

SparseArray比HashMap更省内存，在某些条件下性能更好，主要是因为它避免了对key的自动装箱（int转为Integer类型），它内部则是通过两个数组来进行数据存储的，一个存储key，另外一个存储value，为了优化性能，它内部对数据还采取了压缩的方式来表示稀疏数组的数据，从而节约内存空间。SparseArray只能存储key为int类型的数据，同时，SparseArray在存储和读取数据时候，使用的是二分查找法。SparseArray存储的元素都是按元素的key值从小到大排列好的，在获取数据的时候非常快，比HashMap快的多

虽说SparseArray性能比较好，但是由于其添加、查找、删除数据都需要先进行一次二分查找，所以在数据量大的情况下性能并不明显，将降低至少50%

满足下面两个条件我们可以使用SparseArray代替HashMap：

数据量不大，最好在千级以内

key必须为int类型，这中情况下的HashMap可以用SparseArray代替

ArrayMap是一个<key,value>映射的数据结构，它设计上更多的是考虑内存的优化，内部是使用两个数组进行数据存储，一个数组记录key的hash值，另外一个数组记录Value值，它和SparseArray一样，也会对key使用二分法进行从小到大排序，在添加、删除、查找数据的时候都是先使用二分查找法得到相应的index，然后通过index来进行添加、查找、删除等操作，所以，应用场景和SparseArray的一样，如果在数据量比较大的情况下，那么它的性能将退化至少50%

SparseArray和ArrayMap都差不多，使用哪个呢？

假设数据量都在千级以内的情况下：

1，如果key的类型已经确定为int类型，那么使用SparseArray，因为它避免了自动装箱的过程，如果key为long类型，它还提供了一个LongSparseArray来确保key为long类型时的使用

2，如果key类型为其它的类型，则使用ArrayMap

二十三、java虚拟机和Dalvik虚拟机的区别

1，java虚拟机（JVM）基于栈。 基于栈的机器必须使用指令来载入和操作栈上数据，由于手机上的硬件资源有限，无法支撑JVM频繁的从栈上读写的开销，dalvik虚拟机是基于寄存器的，其数据的访问通过寄存器间直接传递，效率远高于栈

2，java虚拟机运行的是java字节码。（java类会被编译成一个或多个字节码.class文件，打包到.jar文件中，java虚拟机从相应的.class文件和.jar文件中获取相应的字节码）

Dalvik运行的是自定义的.dex字节码格式。（java类被编译成.class文件后，会通过一个dx工具将所有的.class文件转换成一个.dex文件，然后dalvik虚拟机会从其中读取指令和数据）

Dalvik的Dex格式在未压缩的情况下都比压缩了的Jar文件还小

3，常量池已被修改为只使用32位的索引，以 简化解释器。dalvik的堆和栈的参数可以通过-Xms和-Xmx更改

4，一个应用，一个虚拟机实例，一个进程（所有android应用的线程都是对应一个linux线程，都运行在自己的沙盒中，不同的应用在不同的进程中运行。每个android dalvik应用程序都被赋予了一个独立的linux PID(app_*)）

整体上可以把Dalvik理解为为手机设备优化的jvm

二十四、设计模式、手写单例

java的设计模式大体上分为三大类：

创建型模式（5种）：工厂方法模式，抽象工厂模式，单例模式，建造者模式，原型模式

结构型模式（7种）：适配器模式，装饰器模式，代理模式，外观模式，桥接模式，组合模式，享元模式

行为型模式（11种）：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

设计模式遵循的原则有6个：

1，开闭原则（Open Close Principle）

对扩展开放，对修改关闭。

2，里氏代换原则（Liskov Substitution Principle）

只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为

3，依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，对接口编程，依赖于抽象而不依赖于具体

4，接口隔离原则（Interface Segregation Principle）

使用多个隔离的借口来降低耦合度

5，迪米特法则（最少知道原则）（Demeter Principle）

一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立

6，合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。继承实际上破坏了类的封装性，超类的方法可能会被子类修改

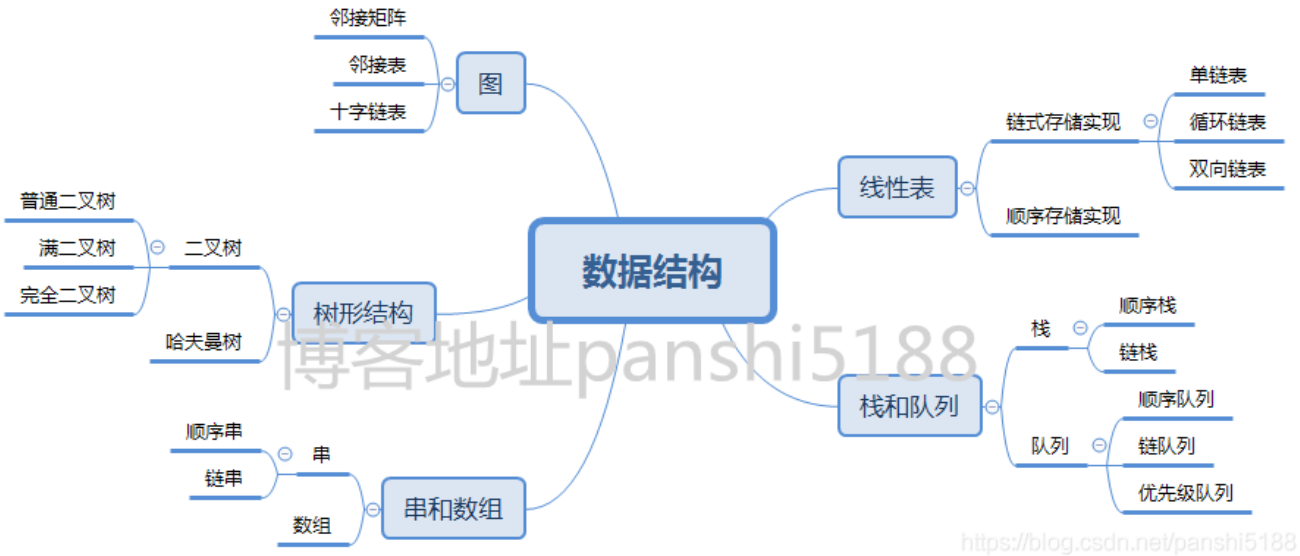
静态内部类单例：

```
public class SingletonDemo {

    private static class SingletonInstance{

        private static final SingletonDem instance=new SingletonDemo();
    }
    private SingletonDemo(){}

    public static SingletonDemo getInstance(){

        return SingletonInstance.instance;

    }
}
```

第二章 数据结构与算法相关

数据结构：

点击查看



数组

```
/**
 * 普通数组的Java代码
 * @author dream
 *
 */
public class GeneralArray {

    private int[] a;
    private int size;    //数组的大小
    private int nElem;    //数组中有多少项

    public GeneralArray(int max){
        this.a = new int[max];
        this.size = max;
        this.nElem = 0;
    }

    public boolean find(int searchNum){    //查找某个值
        int j;
        for(j = 0; j < nElem; j++){
            if(a[j] == searchNum){
                break;
            }
        }
        if(j == nElem){
            return false;
        }else {
            return true;
        }
    }


    public boolean insert(int value){    //插入某个值
        if(nElem == size){
```

```
            System.out.println("数组已满");
            return false;
        }
        a[nElem] = value;
        nElem++;
        return true;
    }

    public boolean delete(int value){   //删除某个值
        int j;
        for(j = 0; j < nElem; j++){
            if(a[j] == value){
                break;
            }
        }
        if(j == nElem){
            return false;
        }
        if(nElem == size){
            for(int k = j; k < nElem - 1; k++){
                a[k] = a[k+1];
            }
        }else {
            for(int k = j; k < nElem; k++){
                a[k] = a[k+1];
            }
        }
        nElem--;
        return true;
    }

    public void display(){   //打印整个数组
        for(int i = 0; i < nElem; i++){
            System.out.println(a[i] + " ");
        }
        System.out.println("");
    }
}
/**
 * 有序数组的Java代码
 * @author dream
 *
 */

/**
 * 对于数组这种数据结构，
 * 线性查找的话，时间复杂度为O(N)，
 * 二分查找的话时间为O(longN)，
 * 无序数组插入的时间复杂度为O(1)，
 * 有序数组插入的时间复杂度为O(N)，
 * 删除操作的时间复杂度均为O(N)。
 * @author dream
 *
```

```
 */
public class OrderedArray {

    private long[] a;
    private int size;   //数组的大小
    private int nElem;  //数组中有多少项

    public OrderedArray(int max){   //初始化数组
        this.a = new long[max];
        this.size = max;
        this.nElem = 0;
    }

    public int size(){   //返回数组实际有多少值
        return this.nElem;
    }

    /**
     * 二分查找
     * @param searchNum
     * @return
     */
    public int find(long searchNum){
        int lower = 0;
        int upper = nElem - 1;
        int curr;
        while (true) {
            curr = (lower + upper) / 2;
            if(a[curr] == searchNum){
                return curr;
            }else if(lower > upper){
                return -1;
            }else {
                if(a[curr] < searchNum){
                    lower = curr + 1;
                }else {
                    upper = curr - 1;
                }
            }
        }
    }


    public boolean insert(long value){   //插入某个值
        if(nElem == size){
            System.out.println("数组已满!");
            return false;
        }
        int j;
        for(j = 0; j < nElem; j++){
            if(a[j] > value){
                break;
            }
```

```
        }

        for(int k = nElem; k > j; k++){
            a[k] = a[k-1];
        }
        a[j] = value;
        nElem++;
        return true;
    }



    public boolean delete(long value){   //删除某个值
        int j = find(value);
        if(j == -1){
            System.out.println("没有该元素!");
            return false;
        }

        if(nElem == size){
            for(int k = j; k < nElem - 1; k++){
                a[k] = a[k+1];
            }
            a[nElem-1] = 0;
        }else {
            for(int k = j; k < nElem; k++){
                a[k] = a[k+1];
            }
        }
        nElem--;
        return true;
    }


    public void display(){   //打印整个数组
        for(int i = 0; i < nElem; i++){
            System.out.println(a[i] + " ");
        }
        System.out.println("");
    }
}
```

▶

栈和队列

##栈

栈只允许访问一个数据项：即最后插入的数据。溢出这个数据才能访问倒数第二个插入的数据项。它是一种"后进先出"的数据结构。

栈最基本的操作是出栈(Pop)、入栈(Push)，还有其他扩展操作，如查看栈顶元素，判断栈是否为空、是否已满，读取栈的大小等。

```
/**
 * 栈是先进后出
 * 只能访问栈顶的数据
 * @author dream
 *
 */

/**
 * 基于数组来实现栈的基本操作
 * 数据项入栈和出栈的时间复杂度均为O(1)
 * @author dream
 *
 */
public class ArrayStack {

    private long[] a;
    private int size;   //栈数组的大小
    private int top;   //栈顶

    public ArrayStack(int maxSize){
        this.size = maxSize;
        this.a = new long[size];
        this.top = -1;   //表示空栈
    }

    public void push(long value){   //入栈
        if(isFull()){
            System.out.println("栈已满!");
            return;
        }
        a[++top] = value;
    }

    public long peek(){   //返回栈顶内容，但不删除
        if(isEmpty()){
            System.out.println("栈中没有数据");
            return 0;
        }
        return a[top];
    }


    public long pop(){   //弹出栈顶内容
        if(isEmpty()){
            System.out.println("栈中没有数据!");
            return 0;
        }
        return a[top--];
    }
```

```
    public int size(){
        return top + 1;
    }

    /**
     * 判断是否满了
     * @return
     */
    public boolean isFull(){
        return (top == size - 1);
    }

    /**
     * 是否为空
     * @return
     */
    public boolean isEmpty(){
        return (top == -1);
    }


    public void display(){
        for (int i = top; i >= 0; i--) {
            System.out.println(a[i] + " ");
        }
        System.out.println("");
    }

}
```

##队列

依然使用数组作为底层容器来实现一个队列的封装

```
/**
 * 队列也可以用数组来实现，不过这里有个问题，当数组下标满了后就不能再添加了，
 * 但是数组前面由于已经删除队列头的数据了，导致空。所以队列我们可以用循环数组来实现，
 * @author dream
 *
 */
public class RoundQueue {

    private long[] a;
    private int size;     //数组大小
    private int nItems;    //实际存储数量
    private int front;     //头
    private int rear;       //尾

    public RoundQueue(int maxSize){
        this.size = maxSize;
        a = new long[size];
        front = 0;
```

```
        rear = -1;
        nItems = 0;
    }

    public void insert(long value){
        if(isFull()){
            System.out.println("队列已满");
            return;
        }
        rear = ++rear % size;
        a[rear] = value;  //尾指针满了就循环到0处，这句相当于下面注释内容
        nItems++;
    }

    public long remove(){
        if(isEmpty()){
            System.out.println("队列为空!");
            return 0;
        }
        nItems--;
        front = front % size;
        return a[front++];
    }

    public void display(){
        if(isEmpty()){
            System.out.println("队列为空!");
            return;
        }
        int item = front;
        for(int i = 0;i < nItems; i++){
            System.out.println(a[item++ % size] + " ");
        }
        System.out.println("");
    }

    public long peek(){
        if(isEmpty()){
            System.out.println("队列为空！");
            return 0;
        }
        return a[front];
    }

    public boolean isFull(){
        return (nItems == size);
    }

    public boolean isEmpty(){
        return (nItems == 0);
    }

    public int size(){
```

```
        return nItems;
    }


}
```

和栈一样，队列中插入数据项和删除数据项的时间复杂度均为O(1)

还有个优先级队列，优先级队列是比栈和队列更专用的数据结构。优先级队列与上面普通的队列相比，主要区别在于队列中的元素是有序的，关键字最小（或者最大）的数据项总在队头。数据项插入的时候会按照顺序插入到合适的位置以确保队列的顺序。优先级队列的内部实现可以用数组或者一种特别的树——堆来实现。这里用数组实现优先级队列。

```
public class PriorityQueue {

   private long[] a;
   private int size;
   private int nItems;  //元素个数

   public PriorityQueue(int maxSize){
    size = maxSize;
    nItems = 0;
    a = new long[size];
   }

   public void insert(long value){
    if(isFull()){
        System.out.println("队列已满！");
        return;
    }
    int j;
    if(nItems == 0){  //空队列直接添加
        a[nItems++] = value;
    }else {
        //将数组中的数字依照下标按照从大到小排列
        for(j=nItems-1; j>=0; j--){
            if(value > a[j]){
                    a[j+1] = a[j];
               }
               else {
                   break;
               }
        }
        a[j+1] = value;
        nItems++;
    }
   }

   public long remove(){
    if(isFull()){
        System.out.println("队列为空!");
        return 0;
    }
```

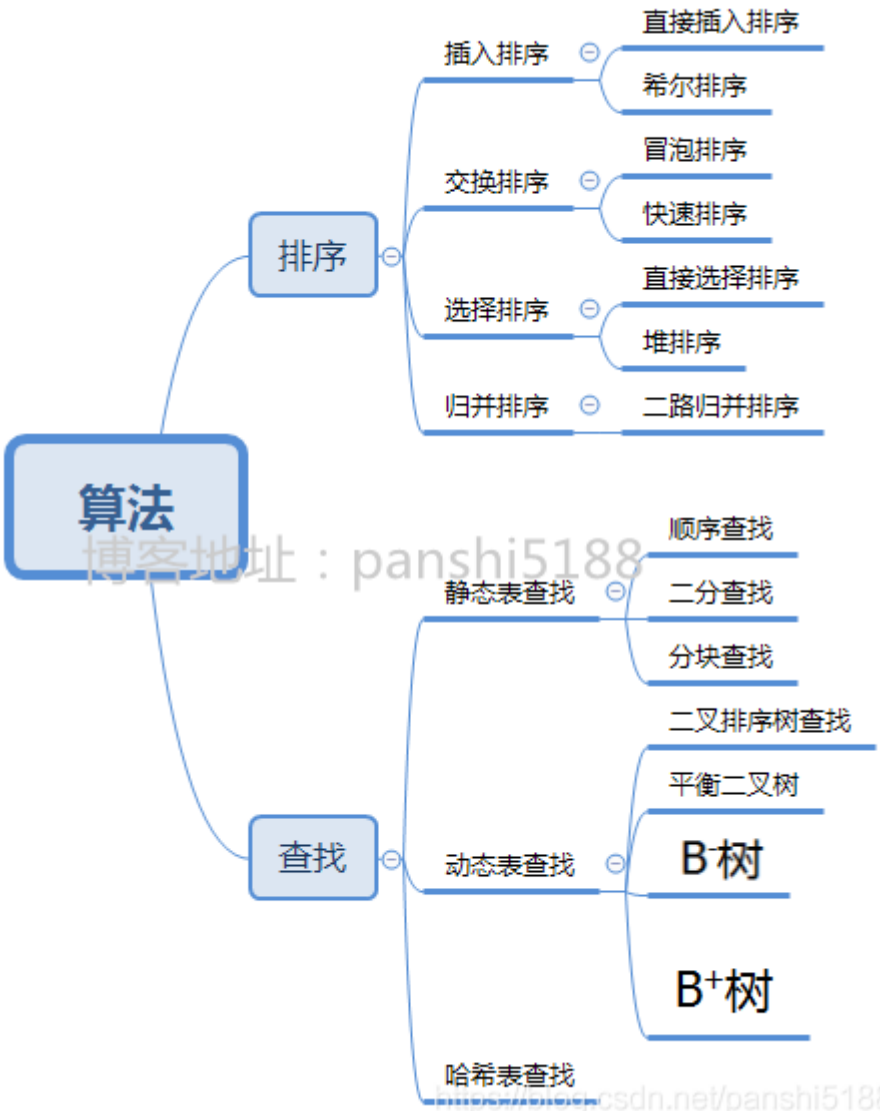
```
     return a[--nItems];
    }

    public long peekMin(){
     return a[nItems - 1];
    }

    public boolean isFull(){
     return (nItems == size);
    }

    public boolean isEmpty(){
     return (nItems == 0);
    }

    public int size(){
     return nItems;
    }

    public void display(){
     for(int i = nItems - 1;i >= 0; i--){
         System.out.println(a[i] + " ");
     }
     System.out.println(" ");
    }
}
```

优先级队列中，插入操作需要O(N)的时间，而删除操作则需要O(1)的时间。

算法：

算法部分要注意平时理解和积累，至少要会手写一种排序算法和一种查找算法，其它的算法原理最好理解

点击查看



冒泡排序：

- 背景介绍： 是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。----- 来自 wikipedia
- 算法规则： 由于算法每次都将一个最大的元素往上冒，我们可以将待排序集合(0...n)看成两部分，一部分为(k..n)的待排序unsorted集合，另一部分为(0...k)的已排序sorted集合，每一次都在unsorted集合从前往后遍历，选出一个数，如果这个数比其后面的数大，则进行交换。完成一轮之后，就肯定能将这一轮unsorted集合中最大的数移动到集合的最后，并且将这个数从unsorted中删除，移入sorted中。
- 代码实现（Java版本）

```
public void sort(int[] args)
        {
            //第一层循环从数组的最后往前遍历
            for (int i = args.length - 1; i > 0 ; --i) {
                //这里循环的上界是 i - 1，在这里体现出 “将每一趟排序选出来的最大的数从sorted中移除”
                for (int j = 0; j < i; j++) {
                    //保证在相邻的两个数中比较选出最大的并且进行交换(冒泡过程)
                    if (args[j] > args[j+1]) {
```

```
                        int temp = args[j];
                        args[j] = args[j+1];
                        args[j+1] = temp;
                    }
                }
            }
        }
```

递归和非递归方式实现二叉树先、中、后序遍历

先序遍历顺序为根、左、右，中序遍历顺序为左、根、右，后序遍历是左、右、根。

递归实现：

```
public class Node {

    public int value;
    public Node left;
    public Node right;

    public Node(int data){
        this.value = data;
    }
}
    /**
     * 前序遍历
     * @param head
     */
    public void preOrderRecur(Node head){
        if(head == null){
            return;
        }
        System.out.println(head.value + " ");
        preOrderRecur(head.left);
        preOrderRecur(head.right);
    }

    /**
     * 中序遍历
     * @param head
     */
    public void inOderRecur(Node head){
        if(head == null){
            return;
        }
        inOderRecur(head.left);
        System.out.println(head.value + " ");
        inOderRecur(head.right);
    }

    /**
     * 后序遍历
     * @param head
```

```
     */
    public void posOrderRecur(Node head){
        if(head == null){
            return;
        }
        posOrderRecur(head.left);
        posOrderRecur(head.right);
        System.out.println(head.value + "");
    }
```

归并排序：

- 背景介绍： 是创建在归并操作上的一种有效的排序算法，效率为O(n log n)。1945年由约翰·冯·诺伊曼首次提出。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用，且各层分治递归可以同时进行。 ----- 来自 wikipedia
- **算法规则： 像快速排序一样，由于归并排序也是分治算法，因此可使用分治思想：** 1.申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列 2.设定两个指针，最初位置分别为两个已经排序序列的起始位置 3.比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置 4.重复步骤3直到某一指针到达序列尾 5.将另一序列剩下的所有元素直接复制到合并序列尾
- 代码实现（Java版本）

```
  public void mergeSort(int[] ints, int[] merge, int start, int end)
  {
    if (start >= end) return;

    int mid = (end + start) >> 1;

    mergeSort(ints, merge, start, mid);
    mergeSort(ints, merge, mid + 1, end);

    merge(ints, merge, start, end, mid);

  }

  private void merge(int[] a, int[] merge, int start, int end,int mid)
  {
    int i = start;
    int j = mid+1;
    int pos = start;
    while( i <= mid || j <= end ){
        if( i > mid ){
            while( j <= end ) merge[pos++] = a[j++];
            break;
        }

        if( j > end ){
            while( i <= mid ) merge[pos++] = a[i++];
            break;
        }

        merge[pos++] = a[i] >= a[j] ? a[j++] : a[i++];
    }
```

```
    for (pos = start; pos <= end; pos++)
        a[pos] = merge[pos];

}
```

▶

快速排序：

- 背景介绍： 又称划分交换排序（partition-exchange sort），一种排序算法，最早由东尼·霍尔提出。在平均状况下，排序n个项目要O(n log n)次比较。在最坏状况下则需要O(n2)次比较，但这种状况并不常见。事实上，快速排序通常明显比其他O(n log n)算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来 ----- 来自 wikipedia **
- 算法规则： 本质来说，快速排序的过程就是不断地将无序元素集递归分割，一直到所有的分区只包含一个元素为止。 由于快速排序是一种分治算法，我们可以用分治思想将快排分为三个步骤： 1.分：设定一个分割值，并根据它将数据分为两部分 2.治：分别在两部分用递归的方式，继续使用快速排序法 3.合：对分割的部分排序直到完成
- 代码实现（Java版本）

```
    public int dividerAndChange(int[] args, int start, int end)
    {
        //标准值
        int pivot = args[start];
        while (start < end) {
            // 从右向左寻找，一直找到比参照值还小的数值，进行替换
            // 这里要注意，循环条件必须是 当后面的数 小于 参照值的时候
            // 我们才跳出这一层循环
            while (start < end && args[end] >= pivot)
                end--;

            if (start < end) {
                swap(args, start, end);
                start++;
            }

            // 从左向右寻找，一直找到比参照值还大的数组，进行替换
            while (start < end && args[start] < pivot)
                start++;

            if (start < end) {
                swap(args, end, start);
                end--;
            }
        }

        args[start] = pivot;
        return start;
    }

    public void sort(int[] args, int start, int end)
    {
        //当分治的元素大于1个的时候，才有意义
```

```
            if ( end - start > 1) {
                int mid = 0;
                mid = dividerAndChange(args, start, end);
                // 对左部分排序
                sort(args, start, mid);
                // 对右部分排序
                sort(args, mid + 1, end);
            }
        }

        private void swap(int[] args, int fromIndex, int toIndex)
        {
            args[fromIndex] = args[toIndex];
        }
```

▶

选择排序：

- 背景介绍： 选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。 ----- 来自 wikipedia
- 算法规则： 将待排序集合(0...n)看成两部分，在起始状态中，一部分为(k..n)的待排序unsorted集合，另一部分为(0...k)的已排序sorted集合,在待排序集合中挑选出最小元素并且记录下标i，若该下标不等于k，那么unsorted[i] 与 sorted[k]交换 ，一直重复这个过程，直到unsorted集合中元素为空为止。
- 代码实现（Java版本）

```
public void sort(int[] args)
{
        int len = args.length;
        for (int i = 0,k = 0; i < len; i++,k = i) {
            // 在这一层循环中找最小
            for (int j = i + 1; j < len; j++) {
                // 如果后面的元素比前面的小，那么就交换下标，每一趟都会选择出来一个最小值的下标
                if (args[k] > args[j]) k = j;
            }

            if (i != k) {
                int tmp = args[i];
                args[i] = args[k];
                args[k] = tmp;
            }
        }
    }
```

顺序查找

基本原理：依次遍历

```
public class Solution {

    public static int SequenceSearch(int[] sz, int key) {
        for (int i = 0; i < sz.length; i++) {
            if (sz[i] == key) {
                return i;
            }
        }
        return -1;
    }
}
```

折半查找

基本原理：每次查找都对半分，但要求数组是有序的

```
public class Solution {

    public static int BinarySearch(int[] sz,int key){
        int low = 0;
        int high = sz.length - 1;

        while (low <= high) {
            int middle = (low + high) / 2;
            if(sz[middle] == key){
                return middle;
            }else if(sz[middle] > key){
                high = middle - 1;
            }else {
                low = middle + 1;
            }
        }
        return -1;
    }
}
```

非递归遍历

1、怎么理解数据结构？

2、什么是斐波那契数列？

3、迭代和递归的特点，并比较优缺点

4、了解哪些查找算法，时间复杂度都是多少？

5、了解哪些排序算法，并比较一下，以及适用场景

6、快排的基本思路是什么？最差的时间复杂度是多少？如何优化？

7、AVL树插入或删除一个节点的过程是怎样的？

8、什么是红黑树？

9、100盏灯问题

10、老鼠和毒药问题，加个条件，必须要求第二天出结果

11、海量数据问题

12、（手写算法）二分查找

13、（手写算法）反转链表

14、（手写算法）用两个栈实现队列

15、（手写算法）多线程轮流打印问题

16、（手写算法）如何判断一个链有环/两条链交叉

17、（手写算法）快速从一组无序数中找到第k大的数/前k个大的数

18、（手写算法）最长（不）重复子串

19、七种方式实现Singleton模式

```
public class Test {

    /**
     * 单例模式，懒汉式，线程安全
     */
    public static class Singleton {
        private final static Singleton INSTANCE = new Singleton();

        private Singleton() {

        }

        public static Singleton getInstance() {
            return INSTANCE;
        }
    }

    /**
     * 单例模式，懒汉式，线程不安全
     */
    public static class Singleton2 {
        private static Singleton2 instance = null;

        private Singleton2() {

        }

        public static Singleton2 getInstance() {
            if (instance == null) {
                instance = new Singleton2();
            }
            return instance;
        }
    }

    /**
```

```
 * 单例模式，饿汉式，线程安全，多线程环境下效率不高
 */
public static class Singleton3 {
    private static Singleton3 instance = null;

    private Singleton3() {

    }

    public static synchronized Singleton3 getInstance() {
        if (instance == null) {
            instance = new Singleton3();
        }
        return instance;
    }
}

/**
 * 单例模式，懒汉式，变种，线程安全
 */
public static class Singleton4 {
    private static Singleton4 instance = null;

    static {
        instance = new Singleton4();
    }

    private Singleton4() {

    }

    public static Singleton4 getInstance() {
        return instance;
    }
}

/**
 * 单例模式，使用静态内部类，线程安全（推荐）
 */
public static class Singleton5 {
    private final static class SingletonHolder {
        private static final Singleton5 INSTANCE = new Singleton5();
    }

    private static Singleton5 getInstance() {
        return SingletonHolder.INSTANCE;
    }
}

/**
 * 静态内部类，使用枚举方式，线程安全（推荐）
 */
public enum Singleton6 {
```

```
        INSTANCE;
        public void whateverMethod() {

        }
    }

    /**
     * 静态内部类，使用双重校验锁，线程安全（推荐）
     */
    public static class Singleton7 {
        private volatile static Singleton7 instance = null;

        private Singleton7() {

        }

        public static Singleton7 getInstance() {
            if (instance == null) {
                synchronized (Singleton7.class) {
                    if (instance == null) {
                        instance = new Singleton7();
                    }
                }
            }
            return instance;
        }
    }

}
```

▶

20、重建二叉树

题目：

> 输入二叉树的前序遍历和中序遍历的结果，重建出该二叉树。假设前序遍历和中序遍历结果中都不包含重复的数字，例如输入的前序遍历序列 {1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}重建出如图所示的二叉 树。

解题思路：

> 前序遍历第一个结点是父结点，中序遍历如果遍历到父结点，那么父结点前面的结点是左子树的结点，后边的结点的右子树的结点，这样我们可以找到左、右子树的前序遍历和中序遍历，我们可以用同样的方法去构建左右子树，可以用递归完成。

代码：

```
public class BinaryTreeNode {

    public static int value;
    public BinaryTreeNode leftNode;
    public BinaryTreeNode rightNode;
}
```

```
public class Solution {

    public static BinaryTreeNode constructCore(int[] preorder, int[] inorder) throws
Exception {
        if (preorder == null || inorder == null) {
            return null;
        }
        if (preorder.length != inorder.length) {
            throw new Exception("长度不一样，非法的输入");
        }
        BinaryTreeNode root = new BinaryTreeNode();
        for (int i = 0; i < inorder.length; i++) {
            if (inorder[i] == preorder[0]) {
                root.value = inorder[i];
                System.out.println(root.value);
                root.leftNode = constructCore(Arrays.copyOfRange(preorder, 1, i + 1),
                        Arrays.copyOfRange(inorder, 0, i));
                root.rightNode = constructCore(Arrays.copyOfRange(preorder, i + 1,
preorder.length),
                        Arrays.copyOfRange(inorder, i + 1, inorder.length));
            }
        }
        return root;

    }
}
```

▶

21、数值的整数次方

题目：

> 实现函数double Power(double base,int exponent)，求base的exponent次方，不得使用库函数，同时不需要考虑大数问题。

看到了很多人会这样写：

```
public static double powerWithExponent(double base,int exponent){
        double result = 1.0;
        for(int i = 1; i <= exponent; i++){
            result = result * base;
        }
        return result;
    }
```

输入的指数(exponent)小于1即是零和负数时怎么办？

当指数为负数的时候，可以先对指数求绝对值，然后算出次方的结果之后再取倒数，当底数(base)是零且指数是负数的时候，如果不做特殊处理，就会出现对0求倒数从而导致程序运行出错。最后，由于0的0次方在数学上是没有意义的，因此无论是输出0还是1都是可以接受的。

```
public double power(double base, int exponent) throws Exception {
```

```
        double result = 0.0;
        if (equal(base, 0.0) && exponent < 0) {
            throw new Exception("0的负数次幂无意义");
        }
        if (equal(exponent, 0)) {
            return 1.0;
        }
        if (exponent < 0) {
            result = powerWithExponent(1.0 / base, -exponent);
        } else {
            result = powerWithExponent(base, exponent);
        }
        return result;
    }

    private double powerWithExponent(double base, int exponent) {
        double result = 1.0;
        for (int i = 1; i <= exponent; i++) {
            result = result * base;
        }
        return result;
    }

    // 判断两个double型数据，计算机有误差
    private boolean equal(double num1, double num2) {
        if ((num1 - num2 > -0.0000001) && (num1 - num2 < 0.0000001)) {
            return true;
        } else {
            return false;
        }
    }
```

一个细节，再判断底数base是不是等于0时，不能直接写base==0，这是因为在计算机内表示小数时(包括float和double型小数)都有误差。判断两个数是否相等，只能判断 它们之间的绝对值是不是在一个很小的范围内。如果两个数相差很小，就可以认为它们相等。

还有更快的方法。

如果我们的目标是求出一个数字的32次方，如果我们已经知道了它的16次方，那么只要在16次方的基础上再平方一次就好了，依此类推，我们求32次方只需要做5次乘法。

我们可以利用如下公式：

```
private double powerWithExponent2(double base,int exponent){
        if(exponent == 0){
            return 1;
        }
        if(exponent == 1){
            return base;
        }
        double result = powerWithExponent2(base, exponent >> 1);
        result *= result;
        if((exponent&0x1) == 1){
```

```
            result *= base;
        }
        return result;
    }
```

我们用右移运算代替除2，用位与运算符代替了求余运算符（%)来判断一个数是奇数还是偶数。位运算的效率比乘除法及求余运算的效率要高很多。

▶

22、扑克牌的顺子

题目:

> 从扑克牌中随机抽 5 张牌,判断是不是顺子,即这 5 张牌是不是连续的。 2-10 为数字本身,A 为 1,J 为 11,Q 为 12,K 为 13,而大小王可以看成任意的 数字。

> 解题思路：我们可以把5张牌看成是由5个数字组成的俄数组。大小王是特殊的数字，我们可以把它们都定义为0，这样就可以和其他的牌区分开来。

> 首先把数组排序，再统计数组中0的个数，最后统计排序之后的数组中相邻数字之间的空缺总数。如果空缺的总数小于或者等于0的个数，那么这个数组就是连续的，反之则不连续。如果数组中的非0数字重复出现，则该数组是不连续的。换成扑克牌的描述方式就是如果一幅牌里含有对子，则不可能是顺子。

详细代码：

```
import java.util.Arrays;

public class Solution {

    public boolean isContinuous(int[] number){
        if(number == null){
            return false;
        }
        Arrays.sort(number);
        int numberZero = 0;
        int numberGap = 0;
        //计算数组中0的个数
        for(int i = 0;i < number.length&&number[i] == 0; i++){
            numberZero++;
        }
        //统计数组中的间隔数目
        int small = numberZero;
        int big = small + 1;
        while(big<number.length){
            //两个数相等，有对子，不可能是顺子
            if(number[small] == number[big]){
                return false;
            }
            numberGap+= number[big] - number[small] - 1;
            small = big;
            big++;
        }
```

```
        return (numberGap>numberZero)?false:true;

    }
}
```

▶

24、圆圈中最后剩下的数字

题目

> 0,1,...,n-1这n个数字排成一个圆圈，从数字0开始每次从这个圆圈里删除第m个数字。求这个圆圈里剩下的最后一个数字。

解法:

> 可以创建一个总共有n个结点的环形链表，然后每次在这个链表中删除第m个结点。我们发现使用环形链表里重复遍历很多遍。重复遍历当然对时间效率有负面的影响。这种方法每删除一个数字需要m步运算，总共有n个数字，因此总的时间复杂度为O(mn)。同时这种思路还需要一个辅助的链表来模拟圆圈，其空间复杂度O(n）。接下来我们试着找到每次被删除的数字有哪些规律，希望能够找到更加高效的算法。

> 首先我们定义一个关于n和m的方程f(n,m)，表示每次在n个数字0，1，。。。。n-1中每次删除第m个数字最后剩下的数字。

> 在这n个数字中，第一个被删除的数字是（m-1)%n.为了简单起见，我们把（m-1)%n记为k，那么删除k之后剩下的n-1个数字为0，1，。。。。。k-1，k+1,.....n-1。并且下一次删除从数字k+1,.....n-1,0,1,....k-1。该序列最后剩下的数字也应该是关于n和m的函数。由于这个序列的规律和前面最初的序列不一样（最初的序列是从0开始的连续序列），因此该函数不同于前面的函数，即为f'(n-1,m)。最初序列最后剩下的数字f(n,m)一定是删除一个数字之后的序列最后剩下的数字，即f(n,m)=f'(n-1,m).

> 接下来我么把剩下的这n-1个数字的序列k+1,....n-1，0，1，,,,,,k-1做一个映射，映射的结果是形成一个从0到n-2的序列

```
k+1        ------>     0
k+2      --------->    1
。。。。
n-1      -----    > n-k-2
0         ------->    n-k-1
1       --------->    n-k
.....
k-1   --------->    n-k
```

> 我们把映射定义为p，则p(x) = (x-k-1)%n。它表示如果映射前的数字是x,那么映射后的数字是（x-k-1)%n.该映射的逆映射是p-1(x)= (x+k+1)%n.

> 由于映射之后的序列和最初的序列具有同样的形式，即都是从0开始的连续序列，因此仍然可以用函数f来表示，记为f(n-1,m).根据我们的映射规则，映射之前的序列中最后剩下的数字f'(n-1,m) = p-1[(n-1,m)] = [f(n-1,m)+k+1]%n ,把k= (m-1)%n代入f(n,m) = f'(n-1,m) =[f(n-1,m)+m]%n.

> 经过上面的复杂的分析，我们终于找到一个递归的公示。要得到n个数字的序列中最后剩下的数字，只需要得到n-1个数字的序列和最后剩下的数字，并以此类推。当n-1时，也就是序列中开始只有一个数字0，那么很显然最后剩下的数字就是0.我们把这种关系表示为：

$$f(n,m)=\begin{cases}0 & n=1\\ [f(n-1,m)+m]\%n & n>1\end{cases}$$

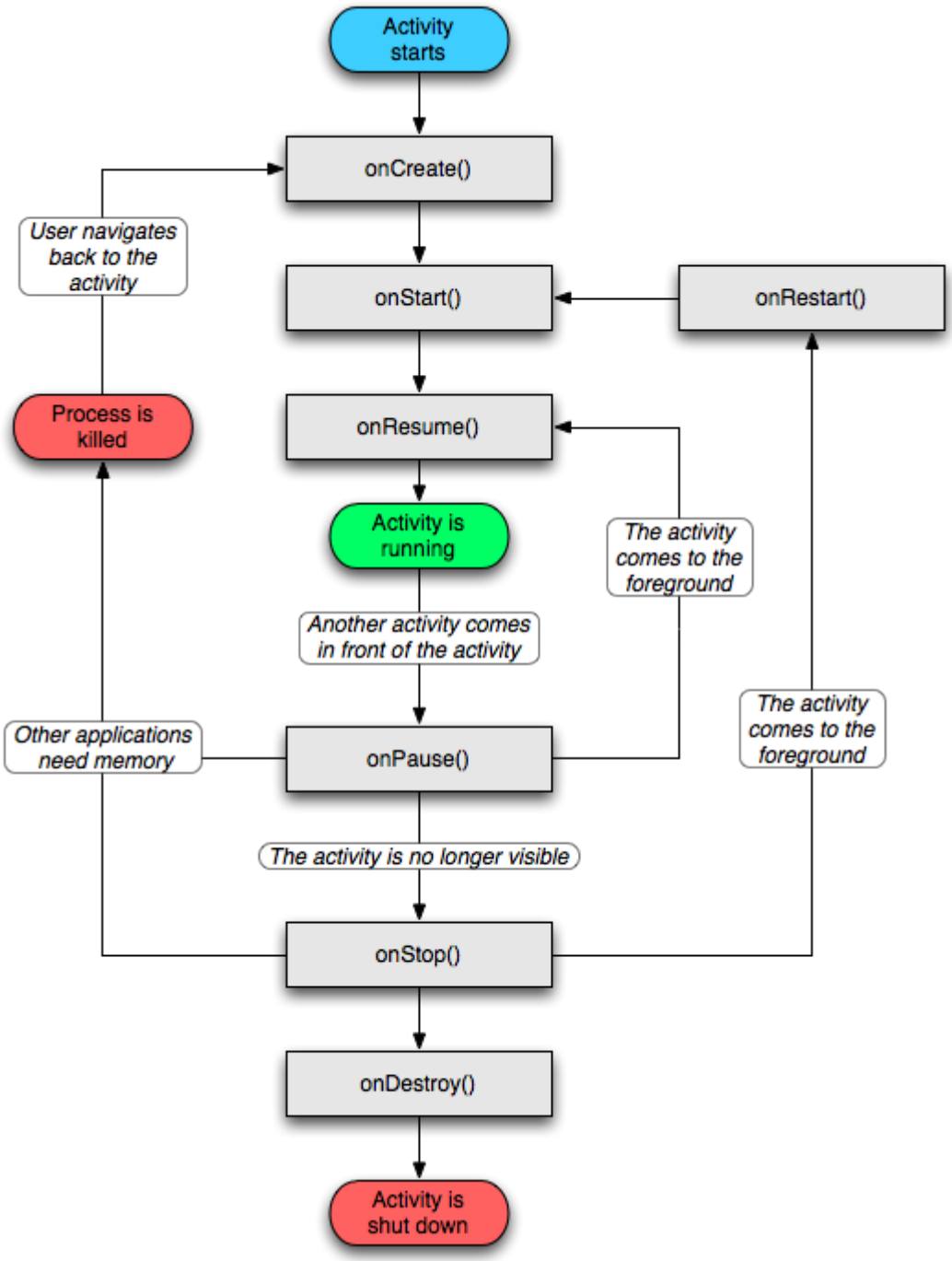
代码如下：

```
public static int lastRemaining(int n, int m){
        if(n < 1 || m < 1){
            return -1;
        }
        int last = 0;
        for(int i = 2; i <= n; i++){
            last = (last + m) % i;
        }
        return last;
    }
```

▶

第三章 Android答案

一、Activity生命周期



实际面试中可能会以实例形式出现，比如：启动A，再从A启动B，请描述各生命周期

二、Activity的启动模式

Activity的启动模式有4种，分别是Standard、SingleTop、SingleTask、SingleInstance

Standard模式：这种模式下，Activity可以有多个实例，每次启动Activity，无论任务栈中是否已经有这个Activity的实例，系统都会创建一个新的Activity实例

SingleTop模式：它和standard模式非常相似，主要区别就是当一个singleTop模式的Activity已经位于任务栈的栈顶，再去启动它时，不会再创建新的实例,如果不位于栈顶，就会创建新的实例

SingleTask模式：此模式下的Activity在同一个Task内只有一个实例，如果Activity已经位于栈顶，系统不会创建新的Activity实例，和singleTop模式一样。但Activity已经存在但不位于栈顶时，系统就会把该Activity移到栈顶，并把它上面的activity出栈

SingleInstance：此模式是全局单例的，启动一个singleInstanceActivity时，系统会创建一个新的任务栈，并且这个任务栈只有这一个Activity，与默认任务栈不在一个栈内，一般在系统应用中会用到

三、App及Activity的启动过程

应用的启动步骤：

1，Launcher通过Binder进程间通信机制通知ActivityManagerService，它要启动一个Activity

2，ActivityManagerService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ActivityManagerService，它已经准备就绪进入Paused状态，于是ActivityManagerService就创建一个新的进程，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ActivityManagerService，以便以后ActivityManagerService能够通过这个Binder对象和它进行通信

5，ActivityManagerService通过Binder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

四、Broadcast分类，使用、区别

广播大体分为两种不同的类型：普通广播和有序广播，其它如系统广播，本地广播也可以归为这两大类中

区别：

普通广播：所有跟广播的intent匹配的广播接收者都可以收到该广播，并且是没有先后顺序（同时收到）

有序广播：系统会根据接收者声明的优先级别按顺序逐个执行接收者，前面的接收者有权终止广播(BroadcastReceiver.abortBroadcast())，如果广播被前面的接收者终止，后面的接收者就再也无法获取到广播

使用：

先声明广播：

```
public class IncomingSMSReceiver extendsBroadcastReceiver {

    @Override


    public void onReceive(Contextcontext, Intentintent) {


    }


}
```

动态注册（记得解注册）：

```
IntentFilter filter = newIntentFilter("android.provider.Telephony.SMS_RECEIVED");


IncomingSMSReceiver receiver = newIncomingSMSReceiver();


registerReceiver(receiver, filter);
```

五、对服务的理解，如何杀死一个服务。服务的生命周期(start与bind)**

篇幅较长，可点击下方链接查看：

Android中对服务Service的理解，Service生命周期学习，如何启动Service及代码验证

六、View的绘制流程**

View 绘制中主要流程分为measure，layout， draw 三个阶段。

OnMeasure()：测量视图大小。从顶层父View到子View递归调用measure方法，measure方法又回调OnMeasure

OnLayout()：确定View位置，进行页面布局。从顶层父View向子View的递归调用view.layout方法的过程，即父View根据上一步measure子View所得到的布局大小和布局参数，将子View放在合适的位置上

OnDraw()：绘制视图。ViewRoot创建一个Canvas对象，然后调用OnDraw()。含六个步骤：绘制视图的背景；保存画布的图层(Layer)；绘制View的内容；绘制View子视图(如果没有就不用)；还原图层(Layer)；绘制滚动条

七、事件分发

事件分发的顺序

页面Activity->容器ViewGroup->控件View

事件分发的核心方法：

dispatchTouchEvent：进行实践分发处理，返回结果表示该事件是否需要分发。默认返回true表示分发给下级视图，不过最终是否分发成功还得根据根据onInterceptTouchEvent方法的拦截判断结果；返回false表示不分发

onInterceptTouchEvent：进行事件拦截处理，返回结果表示当前容器是否需要拦截该事件。返回true表示予以拦截，该手势不会分发给下级视图；默认返回false表示不拦截，该手势会分发给下级视图进行后续处理

onTouchEvent：近视事件触摸处理，返回结果表示该事件是否处理完毕。返回true表示处理完毕，无须处理上级视图的onTouchEvent方法，一路返回结束流程；返回false表示该手势尚未完成，返回继续处理上级视图的onTouchEvent方法，然后根据上级onTouchEvent方法的返回值判断直接结束或由上上级处理。

上述手势方法的执行者有3个：

页面类：可操作dispatchTouchEvent和onInterceptTouchEvent两种方法

容器类：可操作dispatchTouchEvent、onInterceptTouchEvent和onTouchEvent三种方法

控件类：可操作dispatchTouchEvent和onTouchEvent两种方法

八、Handler的原理**

1，在使用handler的时候，在handler所创建的线程需要维护一个唯一的Looper对象， 每个线程对应一个Looper，每个线程的Looper通过ThreadLocal来保证，Looper对象的内部又维护有唯一的一个MessageQueue，所以一个线程可以有多个handler， 但是只能有一个Looper和一个MessageQueue

2，Message在MessageQueue不是通过一个列表来存储的，而是将传入的Message存入到了上一个Message的next中（单链表结构），在取出的时候通过顶部的Message就能按放入的顺序依次取出Message

3，Looper对象通过loop()方法开启了一个死循环，不断地从looper内的MessageQueue中取出Message，然后通过handler将消息分发传回handler所在的线程

4，Handler的构造方法，会得到当前线程中保存的Looper实例，进而与Looper实例中的MessageQueue想关联。Handler的sendMessage方法，会给msg的target赋值为handler自身，然后加入MessageQueue中

5、在构造Handler实例时，我们会重写handleMessage方法，也就是msg.target.dispatchMessage(msg)最终调用的方法

Handler引起内存泄漏问题的解决：使用静态内部类+弱引用的方式；在外部类对象被销毁时，将MessageQueue中的消息清空

九、Android*动画种类、区别**

Android 中的动画有帧动画，补间动画，属性动画

帧动画：主要用于播放帧帧准备图片类似GIF图片优点使用简单便、缺点需要事先准备每帧图片

补间动画：补间动画可以对View进行位置，大小，旋转，透明度四种变化，但是补间动画并没有真正改变View的属性

属性动画：弥补了补间动画不能改变属性的缺点，可以直接更改属性、几乎适用于任何对象

十、Fragment*与Fragment、Activity通信的方式**

1，直接在一个Fragment中调用另外一个Fragment中的方法

2，使用接口回调

3，使用广播

4，Fragment直接调用Activity中的public方法

5，Bundle传递

十一、保存Activity*状态**

在当前Activity被销毁之前会调用**onSaveInstanceState()**方法

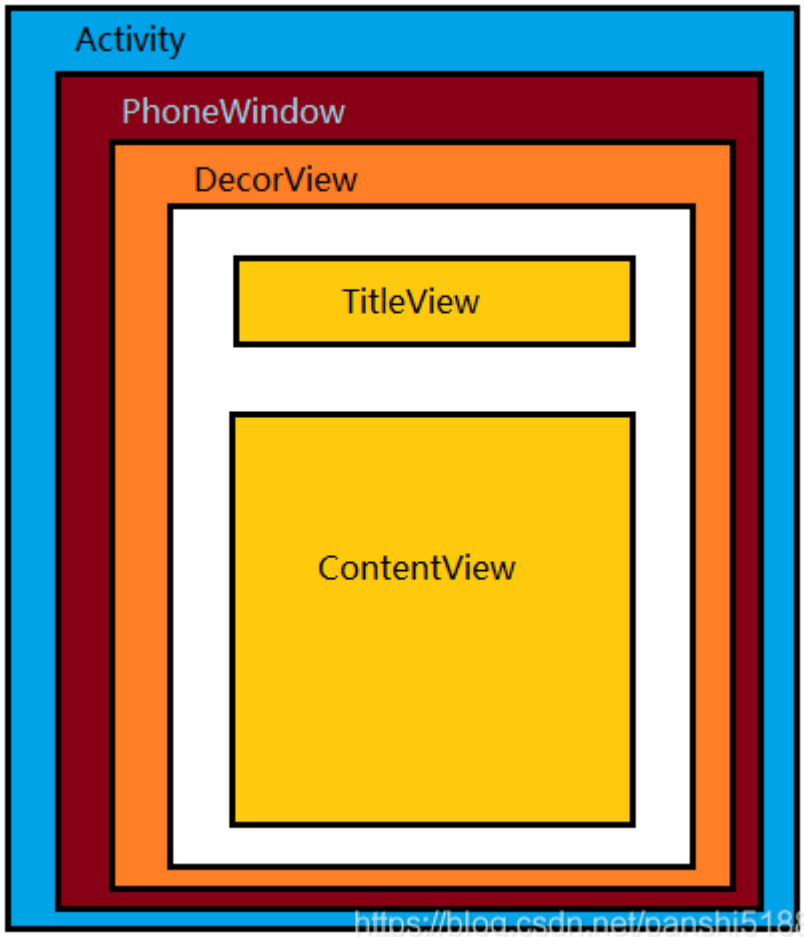
Activity重新创建之后会调用**onRestoreInstanceState()**方法(如果有数据需要恢复)

所以，重写这两个方法即可

调用时机：onSaveInstanceState方法是在onPaused之后onStop之前，onRestoreInstanceState方法是在onStart之后onResume之前

十二、Activity,Window*跟View之间的关系**

我们看图了解一下Activity，PhoneWindow，DecorView之间的联系



Activity：一个Activity就"相当于"一个界面（通过setContentView指定具体的View）。我们可以直接在Activity里处理事件，如onKeyEvent，onTouchEvent等。 并可以通过Activity维护应用程序的生命周期

Window：表示一个窗口，一般来说，Window大小取值为屏幕大小。但是这不是绝对的，如对话框、Toast等就不是整个屏幕大小。你可以指定Window的大小。Window是一个抽象基类，是 Activity 和整个 View 系统交互的接口，只有一个子类实现类PhoneWindow

View：主要是用于绘制我们想要的结果，是一个基本的UI组件

DecorView：DecorView是一个Window的根容器，它本质上是一个FrameLayout。DecorView有唯一一个子View，它是一个垂直LinearLayout，包含两个子元素，一个是TitleView（ActionBar的容器），另一个是ContentView（窗口内容的容器）。关于ContentView，它是一个FrameLayout（android.R.id.content)，我们平常用的setContentView就是设置它的子View

总体来说：一个Activity构造的时候只能初始化一个Window(PhoneWindow)，Activity会调用PhoneWindow的setContentView()将layout布局添加到DecorView上，而此时的DecorView就是那个最底层的View。然后通过LayoutInflater.infalte()方法加载布局生成View对象并通过addView()方法添加到Window上(PhoneWindow有一个View容器 mContentParent，这个View容器是一个ViewGroup，是最初始的根视图，然后通过addView方法将View一个个层叠到mContentParent上，这些层叠的View最终放在Window这个载体上面)。

十三、ViewHolder有什么用****

ViewHolder是一个持有者的类，它的作用就是一个临时的储存器。在列表中，每一个item 的图层都是一样的，那么每次getview 的时候就需要重复的去查找，使用ViewHolder可以把你getView方法中每次返回的View存起来，可以下次再用。这样做的好处就是不必每次都到布局文件中去拿到你的View，提高了效率。

十四、Intentservice有什么用****

IntentService是Service类的子类，用来处理异步请求，是一个封装了HandlerThread和Handler的异步框架，可用于执行后台耗时的任务（Service处于主线程不能直接进行耗时操作），任务执行后会自动停止，并且可以多次启动。

如何使用，创建一个类继承自IntentService，实现onHandleIntent()方法，用于实现任务逻辑，启动时要使用startService()方式

十五、有哪几种创建线程的方式，优缺点

1，继承Thread类

2，实现Runnable接口

3，实现Callable接口

继承Thread类的优缺点：编写简单，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this即可获得当前线程。缺点：线程类已经继承了Thread类，所以不能再继承其他父类

实现Runnable接口和Calllable接口的优缺点：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，较好地体现了面向对象的思想。缺点：编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法

十六、Android中跨进程通讯的几种方式****

1，通过Bundle：Bundle实现了Parcelable接口（一种特有的序列化方法），所以它可以很方便的在不同的进程之间进行传输。简单易用，但传输的数据类型受限

2，通过文件共享：将对象序列化之后保存到文件中，再通过反序列，将对象从文件中读取出来。此方式对文件的格式没有具体的要求，可以是文件、XML、JSON等。文件共享适合在对数据同步要求不高的进程间通信，并且要妥善处理并发读/写的问题。它不适合高并发场景，无法做到进程间的及时通信

3，使用Messenger：在Messenger中放入我们需要传递的数据，实现进程间数据传递。Messenger只能传递Message对象，Messenger是一种轻量级的IPC方案，它的底层实现是AIDL

4，使用AIDL（Android Interface Definition Language）：AIDL是一种IDL语言，用于生成可以在Android设备上两个进程之间进行进程间通信（IPC）的代码。如果在一个进程中（例如Activity）要调用另一个进程中（例如Service）对象的操作，就可以使用AIDL生成可序列化的参数。AIDL是IPC的一个轻量级实现。只有当你允许来自不同的客户端访问你的服务并且需要处理多线程问题时你才必须使用AIDL，其他情况下都可以选择其他方法。AIDL功能强大，支持实时通信，主要是处理多线程、多客户端并发访问的，而Messenger是单线程处理

5，Socket：主要还是应用在网络通信中

6，广播接收者（BroadcastReceiver）

7，内容提供者（ContentProvider）：ContentProvider是一种设备内部共享数据的机制，APP（包括系统应用）可以通过ContentProvider将自身应用的数据对外提供共享，使得其它应用可以对这些数据实现访问和操作

其实从大类分可以分为三类：socket，文件共享，Binder(AIDL、广播、ContentProvider、Messenger的底层实现都是Binder)

十七、进程和线程的区别，如何给四大组件指定多进程

1，进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位

2，同一个进程中可以包括多个线程，并且线程共享整个进程的资源（寄存器、堆栈、上下文），一个进行至少包括一个线程

3，线程是轻两级的进程，它的创建和销毁所需要的时间比进程小很多，所有操作系统中的执行功能都是创建线程去完成的

给四大组件指定多进程：在AndroidManifest.xml中的配置对应组件的android:process属性即可，参数即指定进程名称

十八、同步与异步

同步与异步是针对应用程序与内核的交互而言的。同步过程中进程触发IO操作并等待或者轮询的去查看IO操作是否完成。异步过程中进程触发IO操作以后，直接返回，做自己的事情，IO交给内核来处理，完成后内核通知进程IO完成

同步中各个任务按顺序执行，异步中多个任务可以同时执行

十九、Android UI适配****

1，选用主流分辨率来适配

2，使用限定符

3，利用weight和match

4，使用百分比布局和constraintLayou约束布局

5，第三方适配方案

二十、bitmap的三级缓存思想与如何优化bitmap****

三级缓存：内存缓存、本地缓存（磁盘缓存）、网络缓存

当 App 需要引用缓存时，首先到内存缓存中读取，读取不到再到本地缓存中读取，还获取不到就到网络异步读取，读取成功之后再保存到内存和本地缓存中

LruCache 只是内存缓存的一种比较优秀且常用的缓存算法，并不是说内存缓存就一定指的是 LruCache。很多优秀的图片加载库比如 Glide 有自己独特的内存、磁盘、网络缓存技术

bitmap优化：

1，及时回收Bitmap的内存（调用Bitmap的recycle()方法）

2， 缓存通用的Bitmap对象（多次用到同一张图片的情况）

3，压缩图片：

对图片质量进行压缩：BitmapConfig的配置、bitmap.compress()

对图片尺寸进行压缩：

使用decodeFile、decodeResource、decodeStream进行解析Bitmap时，配置inDensity和inTargetDensity

使用inJustDecodeBounds预判断Bitmap的大小及使用inSampleSize设置采样率进行压缩

bitmap.compress()

使用libjpeg.so库进行压缩：libjpeg是广泛使用的开源JPEG图像库，我们可以自己编译libjpeg进行图片的压缩

4，使用异步加载

二十一、SurfaceView 与View的区别****

View通过刷新来重绘视图，Android系统通过发出VSYNC信号来进行屏幕的重绘，刷新的时间间隔一般为16ms，在一些需要频繁刷新的界面，如果刷新执行很多逻辑绘制操作，就会导致刷新使用时间超过了16ms，就会导致丢帧或者卡顿，比如你更新画面的时间过长，那么你的主UI线程会被你的绘制函数阻塞，那么将无法响应按键，触屏等消息，会造成 ANR 问题。

SurfaceView虽然继承自View，但拥有独立的surface，即它不与其宿主窗口共享同一个surface，可以单独在一个线程进行绘制，并不会占用主线程的资源，这样，绘制就会比较高效，因为SurfaceView的窗口刷新的时候不需要重绘应用程序的窗口（android普通窗口的视图绘制机制是一层一层的，任何一个子元素或者是局部的刷新都会导致整个视图结构全部重绘一次，因此效率非常低下），所以可以实现复杂而高效的UI。游戏，视频播放，直播，都可以用SurfaceView来实现，SurfaceView有两个子类GLSurfaceView和VideoView。

二十二、Bundle传递数据为什么需要序列化****

序列化是一种处理对象流的机制——把内存中的Java对象转换成二进制流。 对象流化后，将对象内容保存在磁盘文件中或作为数据流进行网络传输。简单来说，序列化是将对象的状态信息转换为可以存储或传输的形式的过程。

Bundle传递数据为什么需要序列化：

1，永久性保存对象，保存对象的字节序列到本地文件中

2，对象可以在网络中传输

3，对象可以在IPC之间传递

二十三、Android OOM及ANR的产生原因及处理****

Android为不同类型的进程分配了不同的内存使用上限，如果程序在运行过程中出现了内存泄漏的而造成应用进程使用的内存超过了这个上限，则会产生内存泄漏，进而会导致崩溃

OOM产生原因：

1.数据库的cursor没有关闭

2.构造adapter没有使用缓存contentview

3.调用registerReceiver()后未调用unregisterReceiver()

4.未关闭InputStream/OutputStream

5.Bitmap使用后未调用recycle()

6.Context泄漏

7.static关键字

OOM处理：

1.数据库的cursor没有关闭：及时关闭

2.构造adapter没有使用缓存contentview：使用ViewHolder，实现convertView复用

3.调用registerReceiver()后未调用unregisterReceiver()：及时解注册

4.未关闭InputStream/OutputStream：及时关闭

5.Bitmap使用后未调用recycle()：及时回收，加载优化，参考第二十问

6.Context泄漏：除了static泄露context的方式外，内部类持有外部对象也会造成内存泄露，常见是内部线程造成的。解决：将线程的内部类改为静态内部类；在线程内部采用弱引用保存Context引用

7.static关键字：尽量避免static成员变量引用资源耗费过多的实例，比如Context；Context尽量使用Application Context；使用WeakReference代替强引用

ANR产生原因：当前的事件没有机会得到处理；当前的事件正在处理，但没有及时完成

ANR规则：

1，按键或触摸事件在5s内无响应

2，BroadcastReceiver在10s内无法处理完成

3，Service在20s内无法处理完成

ANR解决：

1，按键或触摸事件在5s内无响应：UI线程尽量只做跟UI相关的工作；耗时的工作（比如数据库操作，I/O，连接网络或者别的有可能阻碍UI线程的操作）把它放入单独的线程处理

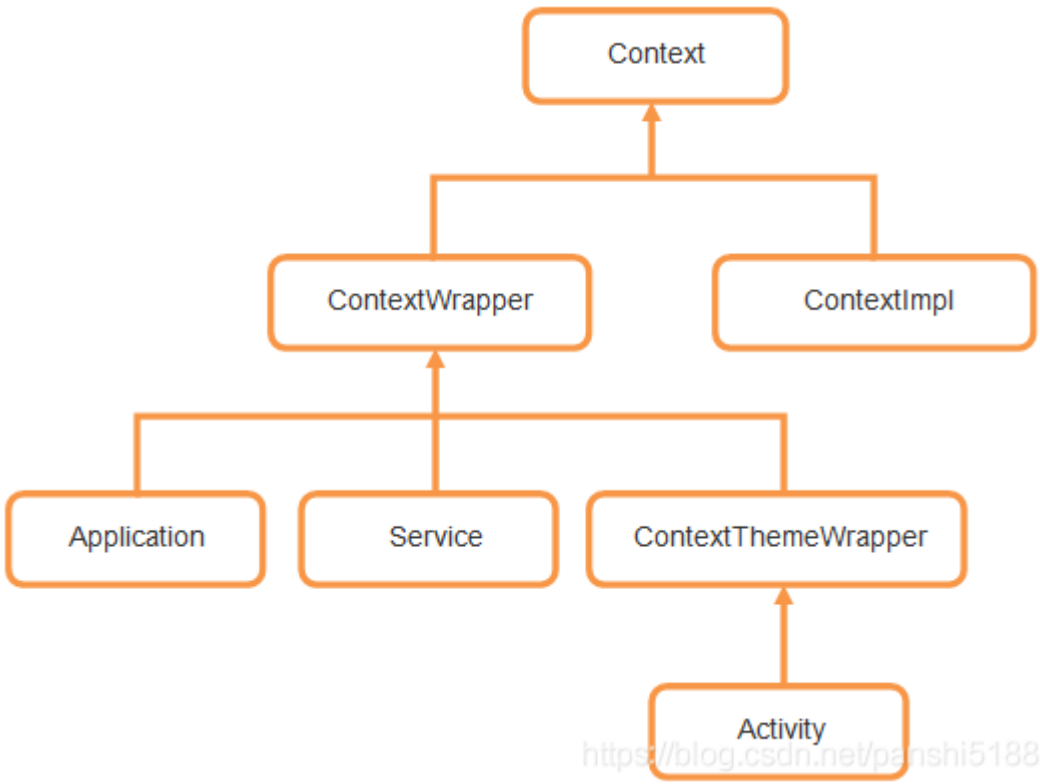
2，BroadcastReceiver在10s内无法处理完成：BroadCastReceiver 要进行复杂操作的的时候，可以在onReceive()方法中启动一个Service来处理(并不是直接在Service中执行耗时操作，参考下一条)

3，Service在20s内无法处理完成：Service是运行在主线程的，如果需要耗时操作，可在Service中开启线程或使用IntentService

二十四、app优化：性能优化、内存优化、启动优化、图片优化、布局优化、响应优化、电量优化、网络优化、安装包优化

Android App优化

二十五、讲解一下Context



Context是一个场景，代表与操作系统的交互的一种过程。从程序的角度上来理解：Context是个抽象类，Context一共有三种类型，分别是Application、Activity和Service。这三个类虽然分别各种承担着不同的作用，但它们都属于Context的一种，而它们具体Context的功能则是由ContextImpl类去实现的。

Context数量 = Activity数量 + Service数量 + 1Application

二十六、Android与H5的交互

Android中是通过webView来加载html页面，而在网页中，JavaScript又是一个很举足轻重的脚本，所以与H5的交互一般就是指与js的交互

实现Java和js交互：

1，WebView开启JavaScript脚本执行

2，WebView设置供JavaScript调用的交互接口

3，客户端和网页端编写调用对方的代码

Android本地通过Java调用HTML页面中的JavaScript方法：

1，若调用的js方法没有返回值，则直接可以调用mWebView.loadUrl("javascript:do()")；其中do是js中的方法；

2，若有返回值时我们可以调用mWebView.evaluateJavascript()方法：

js调用Android本地Java方法：

1，在Android4.2以上可以直接使用@JavascriptInterface注解来声明

2，定义完这个方法后再调用mWebView.addJavascriptInterface()方法

3，在js中调用Java的方法

二十七、混合开发

目前App的开发主要包含三种方式：原生开发、HTML5开发和混合开发

原生开发：是在Android、IOS等移动平台上利用官方提供的开发语言、开发类库、开发工具进行App开发。原生应用在应用性能上和交互体验上应该是最好的，但是原生应用的可移植性比较差，特别是一款原生的App，Android和IOS都要各自开发，同样的逻辑、界面要写两套。

HTML5开发：，是利用Web技术进行的App开发，我们知道web技术本身需要浏览器的支持才能进行展示和用户交互。主要用到的技术是HTML5、JavaScript、CSS等。H5开发的好处是可以跨平台，编写的代码可以同时在Android、IOS、Windows上进行运行。由于Web技术本身的限制，H5移动应用不能直接访问设备硬件和离线存储，所以在体验和性能上有很大的局限性。

混合开发：结合了原生和H5开发的技术，是取长补短的一种开发模式，原生代码部分利用WebView插件或者其它的框架为H5提供了一个容器，程序主要的业务实现、界面展示是利用H5相关的Web技术进行实现的。比如现在的京东、淘宝、今日头条等都是利用的混合开发模式。

混合开发的优缺点：

优点：

1、开发效率高，节约时间，同一套代码Android和IOS基本都可用

2、更新和部署比较方便，不需要每次升级都要上传到App Store进行审核了，只需要在服务器端升级就可以

3、代码维护方便、版本更新快，降低产品成本

缺点是：

1、由于不能直接操控硬件有些方面性能不是很好

2、另外有技术比较新版本的兼容性比较差，还有就是即懂原生开发又懂H5开发的高端人才难找

混合App开发是未来的趋势，目前市面上的混合开发框架有：React Native，Weex，PhoneGap ，Ionic ，Hbuilder，appcan，ApiCloud等

二十八、RxJava

RxJava是一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库（一个支持异步的链式编程库）

RxJava 的优势是简洁，但它的简洁的与众不同之处在于，随着程序逻辑变得越来越复杂，它依然能够保持简洁

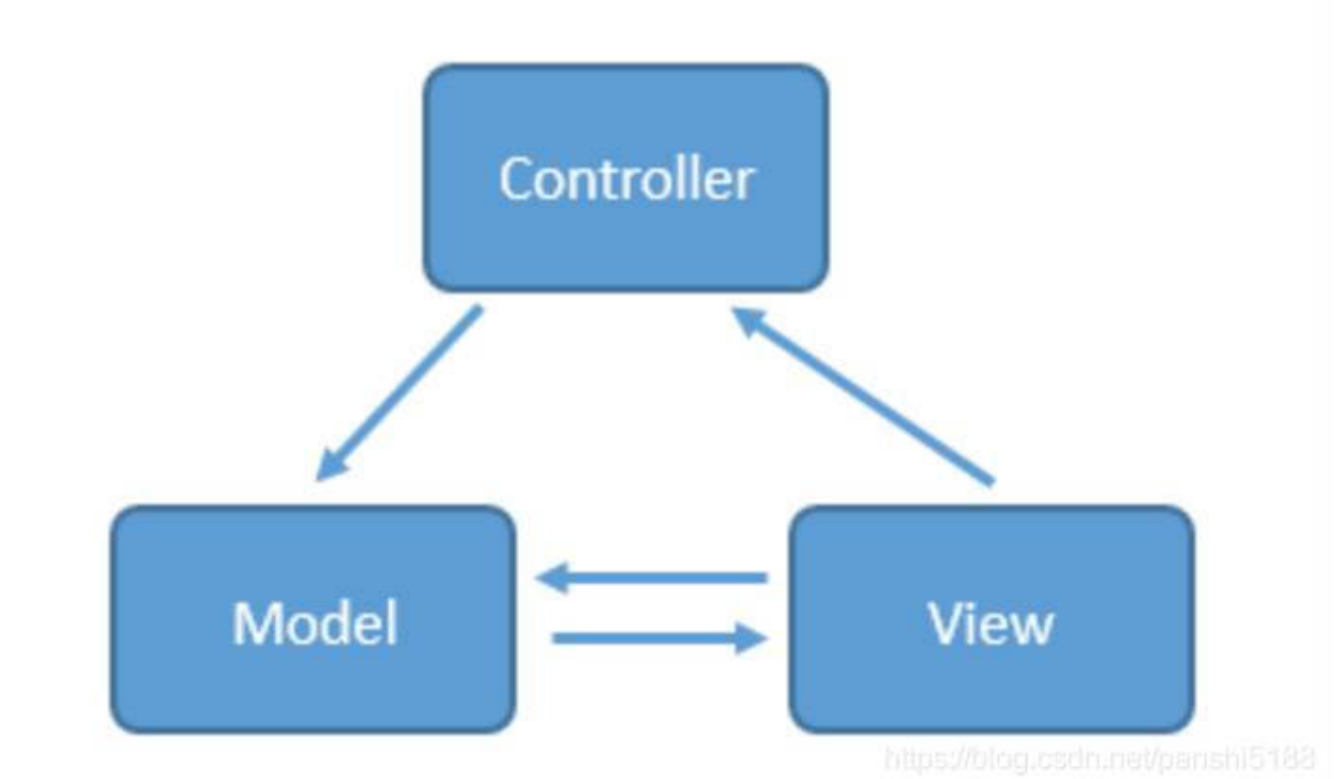
RxJava 的异步实现，是通过一种扩展的观察者模式来实现的

RxJava 的基本实现：创建 Observer、创建 Observable、Subscribe (订阅)

应用场景：与Retrofit联用、Rx类库的使用（如基于RxJava的开源类库Rxpermissions、RxBinding以及RxBus）、其它用到异步的地方

二十九、MVP，MVC，MVVM

MVC：



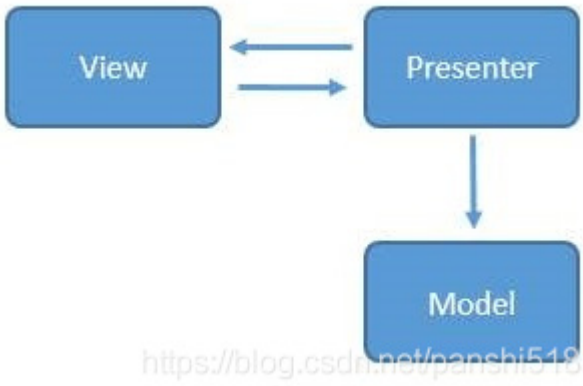
视图层(View) ：对应于xml布局文件和java代码动态view部分

控制层(Controller) ：MVC中Android的控制层是由Activity来承担的，Activity本来主要是作为初始化页面，展示数据的操作，但是因为XML视图功能太弱，所以Activity既要负责视图的显示又要加入控制逻辑，承担的功能过多

模型层(Model) ：针对业务模型，建立的数据结构和相关的类，它主要负责网络请求，数据库处理，I/O的操作

工作原理：当用户出发事件的时候，view层会发送指令到controller层，接着controller去通知model层更新数据，model层更新完数据以后直接显示在view层上，这就是MVC的工作原理

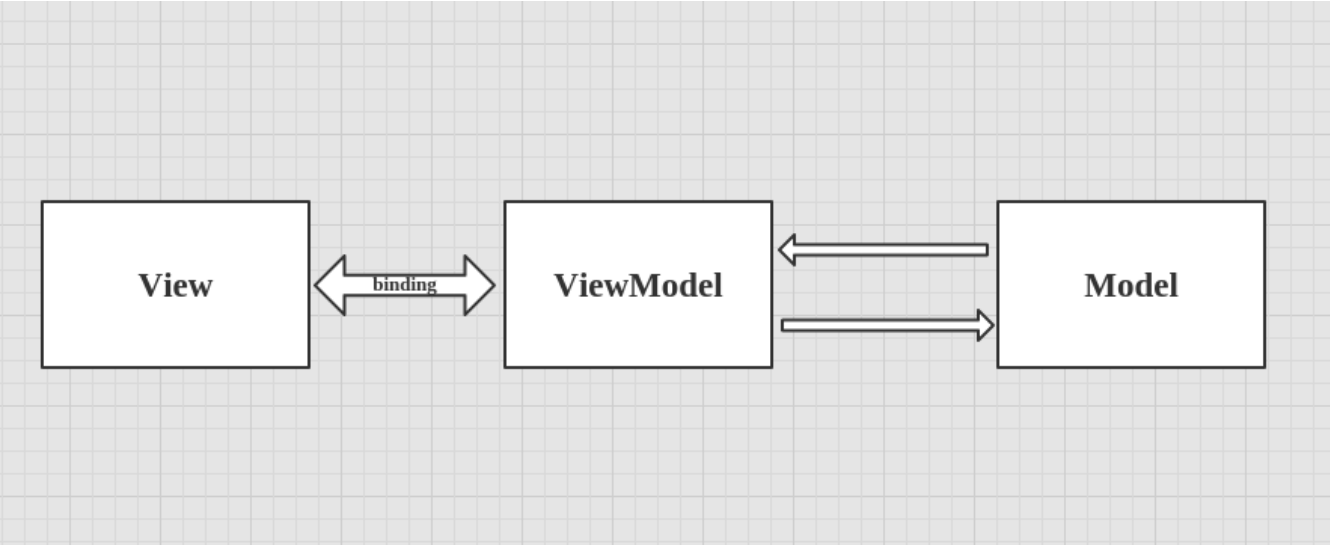
MVP：



MVP跟MVC很相像，唯一的差别是Model和View之间不进行通讯，都是通过Presenter完成

工作原理：view层发出的事件传递到presenter层中，presenter层去操作model层，并且将数据返回给view层

MVVM：



MVVM通过DataBinding双向绑定的机制，实现数据和UI内容，只要想改其中一方，另一方都能够及时更新的一种设计理念，这样就省去了很多在View层中写很多case的情况，只需要改变数据就行

工作原理：presenter层换成了viewmodel层，view层和viewmodel层是相互绑定的关系，当更新viewmodel层的数据的时候，view层会相应的更新ui

三十、session**与cookie的区别****

cookie是服务器在本地机器上存储的小段文本并随每一个请求发送至同一服务器，是在客户端保持状态的方案。

在网站中，http请求是无状态的。也就是说即使第一次和服务器连接后并且登录成功后，第二次请求服务器依然不能知道当前请求是哪个用户。cookie的出现就是为了解决这个问题。第一次登录后服务器返回一些数据（cookie）给浏览器，然后浏览器保存在本地，当该用户发送第二次请求的时候，就会自动的把上次请求存储的cookie数据自动的携带给服务器，服务器通过浏览器携带的数据就能判断当前用户是哪个了。cookie存储的数据量有限，不同的浏览器有不同的存储大小，但一般不超过4KB。因此使用cookie只能存储一些小量的数据。

session和cookie的作用有点类似，都是为了存储用户相关的信息。不同的是，cookie是存储在本地浏览器，而session存储在服务器。存储在服务器的数据会更加的安全，不容易被窃取。但存储在服务器也有一定的弊端，就是会占用服务器的资源，但现在服务器已经发展至今，一些session信息还是绰绰有余的。

区别：

1，存储数据量方面：session 能够存储任意的 java 对象，cookie 只能存储 String 类型的对象且存储量小

2，一个在客户端一个在服务端。因Cookie在客户端所以不是十分安全。

3，Session过多时会消耗服务器资源，大型网站会有专门Session服务器，Cookie存在客户端没问题

4，域的支持范围不一样，比方说a.com的Cookie在a.com下都能用，而www.a.com的Session在api.a.com下都不能用，解决这个问题的办法是JSONP或者跨域资源共享。

三十一、如何取消一个网络请求

1，使用 httpClient时调用httpClient.getConnectionManager().shutdown()

2，通过设置tag取消

3，使用Retrofit2时通过调用Call的cancel()方法

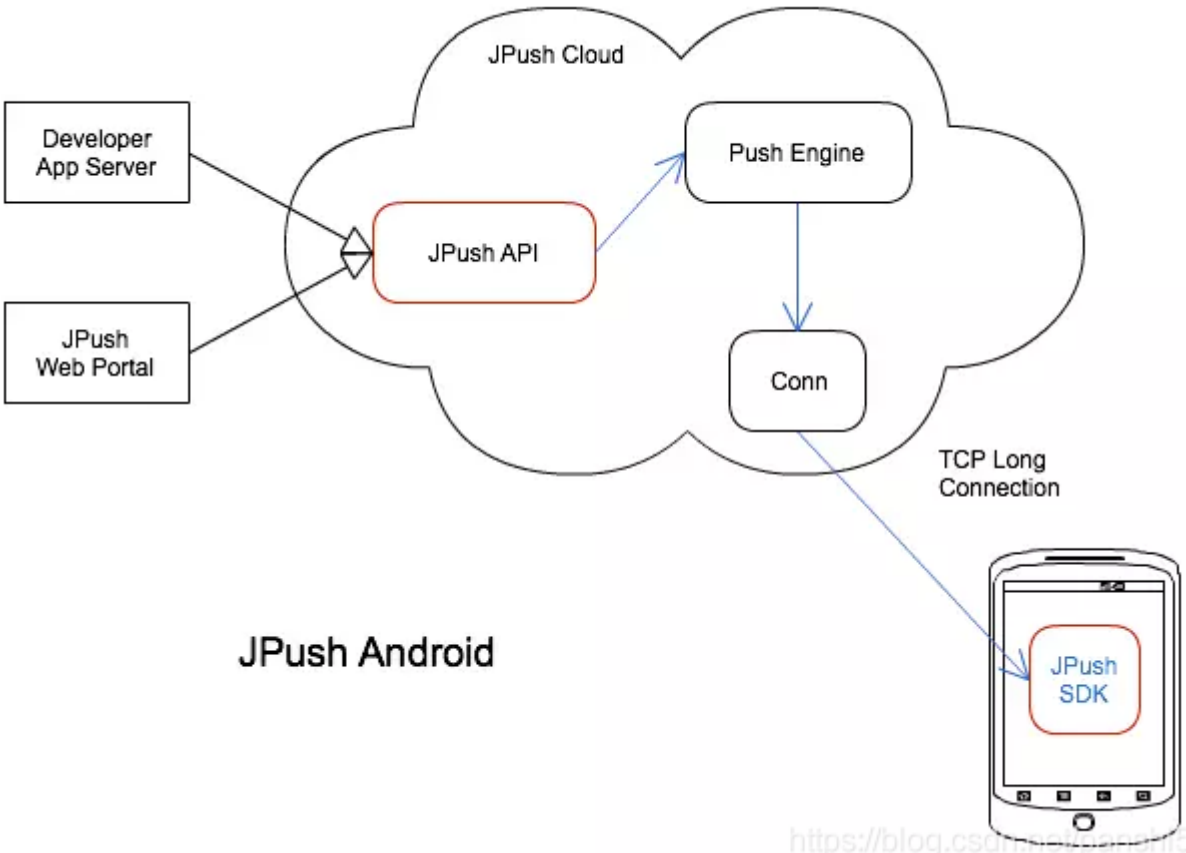
4，使用Retrofit2+Rxjava2通过调用 Disposable的dispose()方法

三十二、如何实现一个推送，极光推送原理

1，轮询法：这种方法最简单，Client每过一段时间向Server请求一次数据。优缺点很明显，优点是实现简单；缺点是间隔时间不好控制，并且消耗大（电量、流量）

2，长连接法：Client使用socket连接Server，并且保持socket连接，Server随时可以通过这个socket发送数据给Client。优点：最有效，客户端设备消耗比第一种小（设备应该从系统层对socket的长连接做优化，socket链接维护成本从客户端来讲应该是小于频繁的http请求的）；缺点：服务端压力大，每一个设备都需要一个socket连接

极光推送原理：



服务端设计：针对 C10K 问题，极光采用了多消息循环、异步非阻塞的模型，在一台双核、24G内存的服务器上，实现峰值维持超过300万个长连接

客户端设计：

为了长时间保持外网IP不变，需要客户端定期发送心跳给运营商，以便刷新NAT列表

AlarmManager 是 Android 系统封装的用于管理 RTC 的模块，RTC (Real Time Clock) 是一个独立的硬件时钟，可以在 CPU 休眠时正常运行，在预设的时间到达时，通过中断唤醒 CPU。这意味着，如果我们用 AlarmManager 来定时执行任务，CPU 可以正常的休眠，只有在需要运行任务时醒来一段很短的时间。极光推送的 Android SDK 就是基于这种技术实现的。

三十三、如何进行单元测试

单元测试又称为模块测试，是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。 android中的单元测试基于JUnit，可分为本地测试和instrumented测试

在android测试框架中，常用的有以下几个框架和工具类：JUnit4、AndroidJUnitRunner、Mockito、Espresso

基本步骤：添加单元测试支持（默认提供），添加单元测试类，运行单元测试，查看运行结果

三十四、进程的优先级、进程保活（不死进程）

进程优先级（从高到低）：前台进程:Foreground process、可见进程:Visible process、服务进程:Service process、后台进程:Background process、空进程:Empty process

Android系统会在内存不足的时候去将进程杀死，俗称Low Memory Killer，它是基于Linux内核的 OOM Killer（Out-Of-Memory killer）机制，内存不足时，优先杀oom_adj值（不是进程优先级）高的进程

常见的保活拉起方式：

1，不同的app进程，用广播相互唤醒

2，利用系统Service机制拉活（前台服务）

3，Native进程拉起

4，双进程守护

5，JobScheduler

三十五、JNI/NDK基本使用，java如何调用C语言的方法

JNI：Java Native Interface（Java 本地编程接口），一套编程规范，它提供了若干的 API 实现了 Java 和其他语言的通信（主要是 C/C++）。Java与本地代码通过JNI可以相互调用。Java 通过 C/C++ 使用本地的代码的一个关键性原因在于 C/C++ 代码的高效性。

NDK全称是Native Development Kit，NDK提供了一系列的工具，帮助开发者快速开发C(或C++)的动态库，并能自动将so和java应用一起打包成apk。 NDK集成了交叉编译器(交叉编译器需要UNIX或LINUX系统环境)，并提供了相应的mk文件隔离CPU、平台、ABI等差异，开发人员只需要简单修改mk文件(指出“哪些文件需要编译”、“编译特性要求”等)，就可以创建出so。

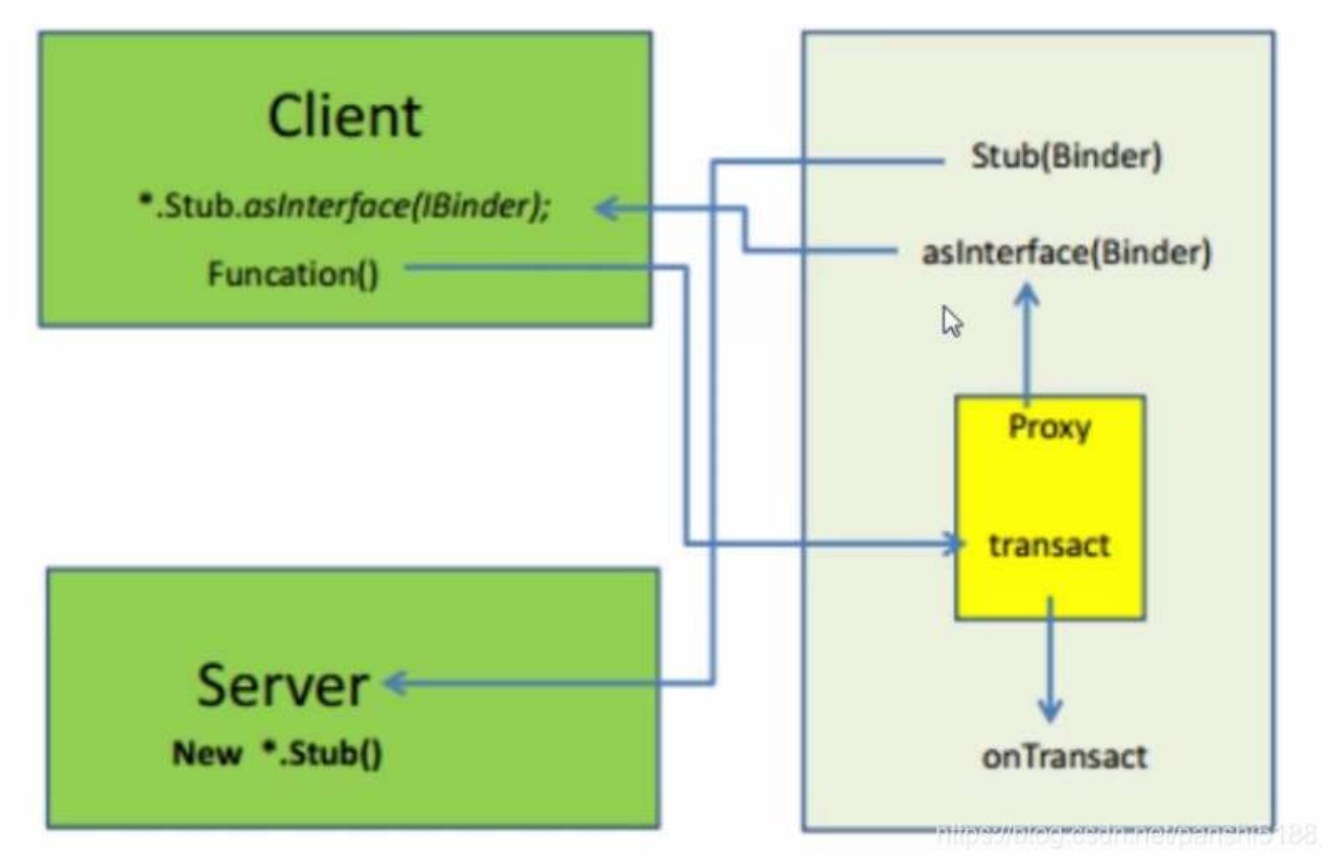
使用：添加NDK支持，按提示创建工程，根据需求使用NDK（待完善）

三十六、经典第三方框架原理

比较经典的有：OkHttp、Retrofit、Glide、ButterKnife、RxJava等，涉及的框架较多，需查找对应文章

三十七、AIDL理解****

具体概念请参考第十六问



service 根据AIDL 生成相应的 java interface声明文件和实现impl文件，并且需要定义新的service将这些实现暴露给用户，即提供onbind方法。

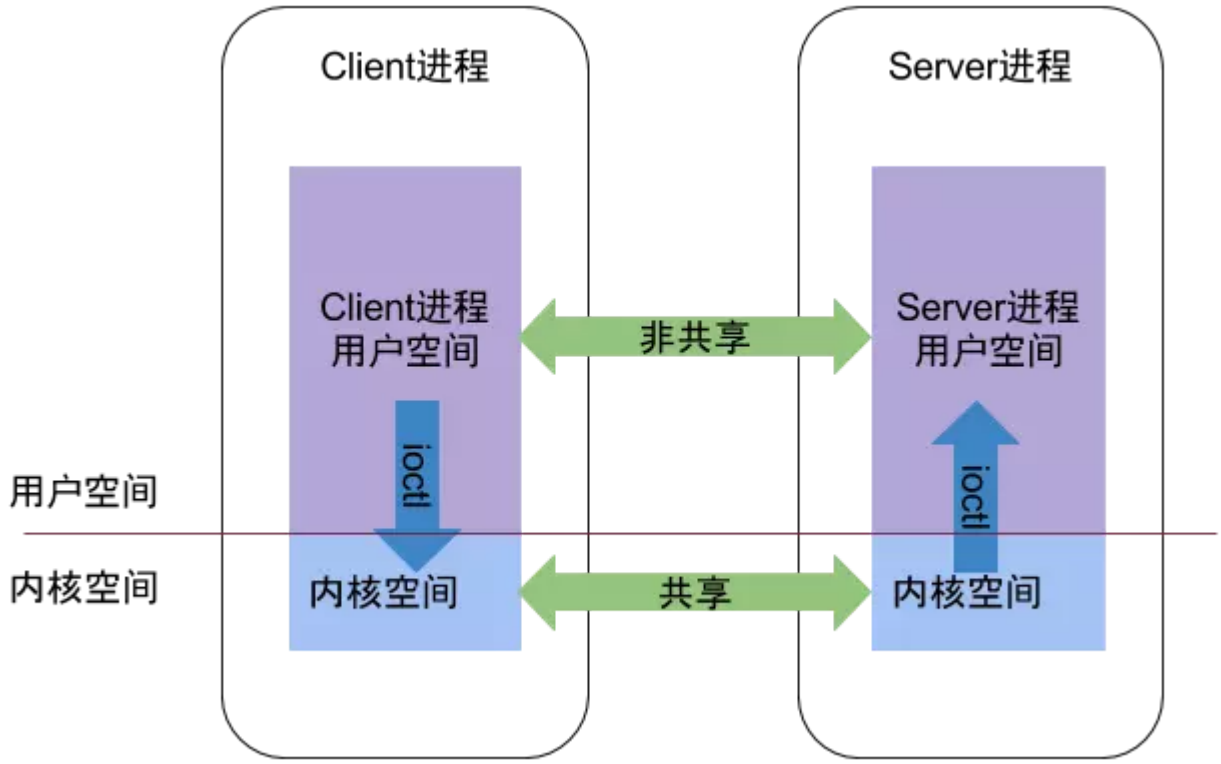
client，根据AIDL中的声明，通过相应的Intent，bind到对应的service。获得service的stub，请求相应的功能。

service 和client 运行在不同的进程中。client端通过stub向真正的service发起结果请求。 系统通过stub隐藏了内部的真正的通信机制，只返回相应的service的执行结果。

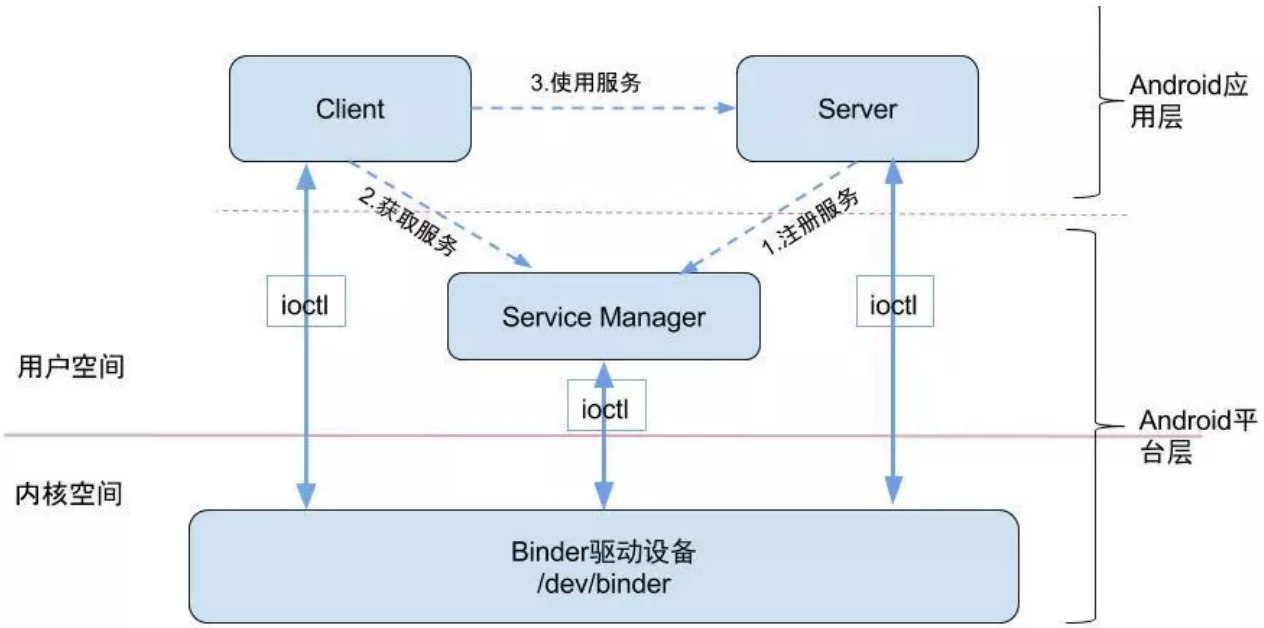
AIDL的核心是Binder，我们通过AIDL文件来描述接口，来得到一个封装好的IBinder代理，来实现接口的远程调用。

三十八、Binder机制原理****

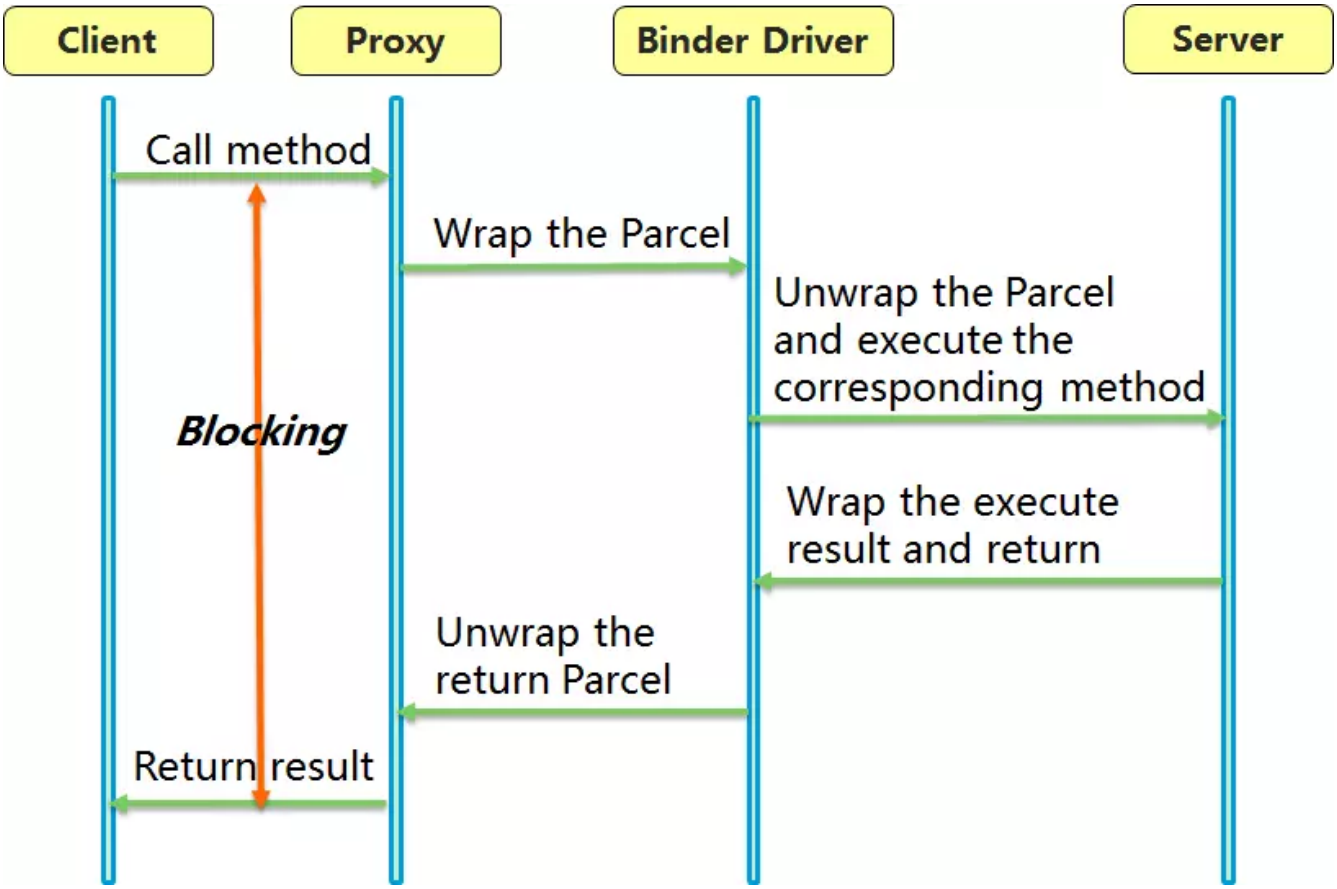
IPC机制：



Binder架构和运行机制：



Binder框架定义了四个角色：Server，Client，ServiceManager以及Binder驱动。其中Server，Client，SMgr运行于用户空间，驱动运行于内核空间。



使用服务的具体执行过程：

1，client通过获得一个server的代理接口，对server进行调用

2，代理接口中定义的方法与server中定义的方法时一一对应的

3，client调用某个代理接口中的方法时，代理接口的方法会将client传递的参数打包成Parcel对象

4，代理接口将Parcel发送给内核中的binder driver

5，server会读取binder driver中的请求数据，如果是发送给自己的，解包Parcel对象，处理并将结果返回

6，整个的调用过程是一个同步过程，在server处理的时候，client会block住。因此client调用过程不应在主线程。

Binder数据拷贝只需要一次，而管道、消息队列、Socket都需要2次，共享内存方式一次内存拷贝都不需要，但实现方式又比较复杂。

Binder通信采用C/S架构，安全性好，简单高效。

三十九、ClassLoader

ClassLoader翻译过来就是类加载器，是用来加载 Class 的。它负责将 Class 的字节码形式转换成内存形式的 Class 对象。字节码可以来自于磁盘文件 *.class，也可以是 jar 包里的 *.class，也可以来自远程服务器提供的字节流。

JVM 启动的时候，并不会一次性加载所有的class文件，而是根据需要去动态加载。

JVM 中内置了三个重要的 ClassLoader，分别是 BootstrapClassLoader、ExtensionClassLoader 和 AppClassLoader。

BootstrapClassLoader 负责加载 JVM 运行时核心类，这些类位于 JAVA_HOME/lib/rt.jar 文件中，我们常用内置库 java.xxx.* 都在里面，比如 java.util.、*java.io.*、java.nio.、*java.lang.* 等等。这个 ClassLoader 比较特殊，它是由 C 代码实现的，我们将它称之为「根加载器」。

ExtensionClassLoader 负责加载 JVM 扩展类，比如 swing 系列、内置的 js 引擎、xml 解析器 等等，这些库名通常以 javax 开头，它们的 jar 包位于 JAVA_HOME/lib/ext/*.jar 中，有很多 jar 包。

AppClassLoader 才是直接面向我们用户的加载器，它会加载 Classpath 环境变量里定义的路径中的 jar 包和目录。我们自己编写的代码以及使用的第三方 jar 包通常都是由它来加载的。

ClassLoader采用双亲代理模型，加载一个类时，首先BootStrap进行寻找，找不到再由Extension ClassLoader寻找，最后才是App ClassLoader。

Android中的classLoader具有两个特点：

类加载共享：当一个class文件被任何一个ClassLoader加载过，就不会再被其他ClassLoader加载。

类加载隔离：不同ClassLoader加载的class文件肯定不是一个。

四十、静态代理跟动态代理

代理是一种常用的设计模式，其目的就是为其他对象提供一个代理以控制对某个对象的访问。代理类负责为委托类预处理消息，过滤消息并转发消息，以及进行消息被委托类执行后的后续处理。

静态代理类优缺点

优点：

代理使客户端不需要知道实现类是什么，怎么做的，而客户端只需知道代理即可（解耦合）

缺点：

1，代理类和委托类实现了相同的接口，代理类通过委托类实现了相同的方法。这样就出现了大量的代码重复。如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。

2，代理对象只服务于一种类型的对象，如果要服务多类型的对象。势必要为每一种对象都进行代理，静态代理在程序规模稍大时就无法胜任了。即静态代理类只能为特定的接口(Service)服务。如想要为多个接口服务则需要建立很多个代理类。

动态代理：在运行时，通过反射机制实现动态代理，并且能够代理各种类型的对象

动态代理优点：接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）。这样，在接口方法数量比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转。而且动态代理的应用使我们的类职责更加单一，复用性更强。

四十一、AMS,WMS,PMS

AMS（ActivityManagerService）是Android中最核心的服务，主要负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统中的进程管理和调度模块相类似

WMS（WindowManagerService）是系统服务的一部分，由SystemService启动，发生异常时自动重启，直到系统关机时才能退出。主要有两方面的功能：

全局的窗口管理（Output）：应用程序的显示请求在SurfaceFlinger和WMS的协助下有序地输出给物理屏幕或其他显示设备。

全局的事件管理派发（Input）：事件源包括键盘、触摸屏、鼠标、轨迹球（TraceBoll）等。

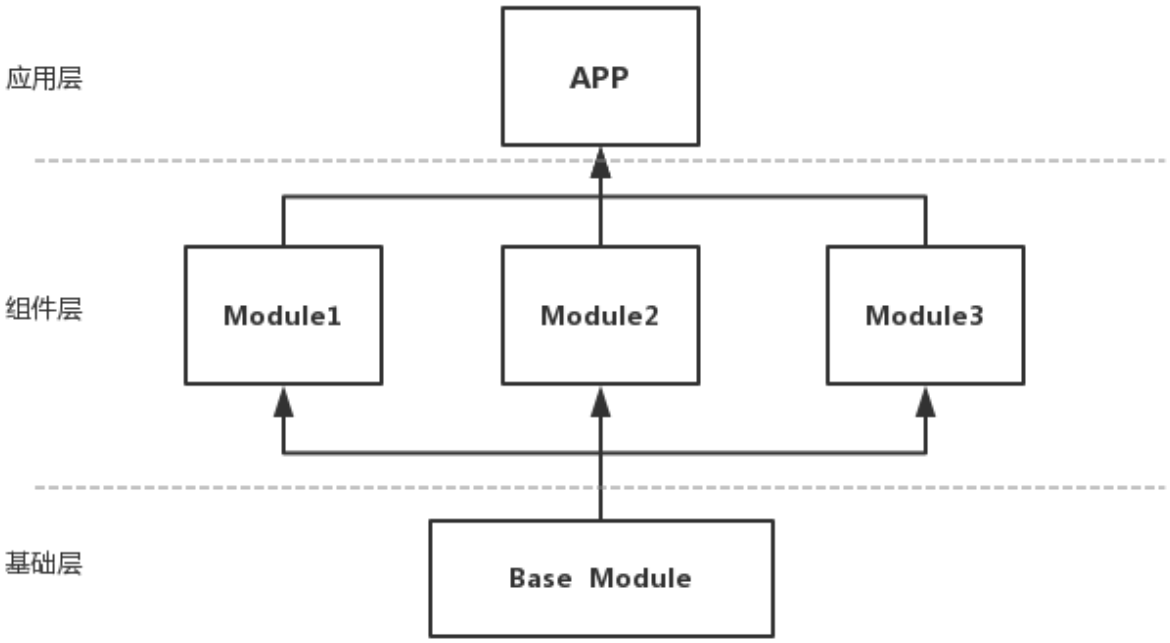
PMS（PackageManagerService）：Android Package管理服务，其主要管理Android系统中App的安装包等信息

（深入理解请查找相关资料）

四十二、组件化原理，模块化机制

组件化：将一个app的代码拆分成几份独立的组件，组件之间是低耦合的，可以独立编译打包；也可以将组件打包到一个apk中。

在开发中不断地增加新特性意味着更长的build时间和更长的增量build时间。在工程较大的项目中，build时间要占到10%～15%的工作时间。把Application分成多个modules可以解决这个问题，也就是组件化概念



组件层的模块都依赖于基础层，从而产生第三者联系，这种第三者联系最终会编译在APP Module中，那时将不会有这种隔阂，那么其中的Base Module就是跨越组件化层级的关键，也是模块间信息交流的基础。

模块化：

Android studio支持了多个module开发时，提出的模块化概念。

具体实践：把常用的功能、控件、基础类、第三方库、权限等公共部分抽离封装，把业务拆分成N个模块进行独立(module)的管理。而所有的业务组件都依赖于封装的基础库，业务组件之间不做依赖，这样的目的是为了让每个业务模块能单独运行。

模块化的特点是：模块之间解耦，可以独立管理。

四十三、热更新与插件化，它们的实现核心原理是什么，它们的区别，如何进行dex替换****

热更新：无须下载新apk即可完成对app线上bug的修复

插件化：App 的部分功能模块在打包时并不以传统方式打包进 apk 文件中，而是以另一种形式封装进 apk内部，或者放在网络上适时下载，在需要的时候动态对这些功能模块进行加载，称之为插件化。

热更新原理：基于ClassLoader 的 dex 文件替换。主要为以tinker为代表的multidex类加载法和以阿里andfix为代表的底层替换法，而阿里sophix为了提高热修复的成功率同时采用了上述两种方案，并在兼容性上进行了一定的优化。

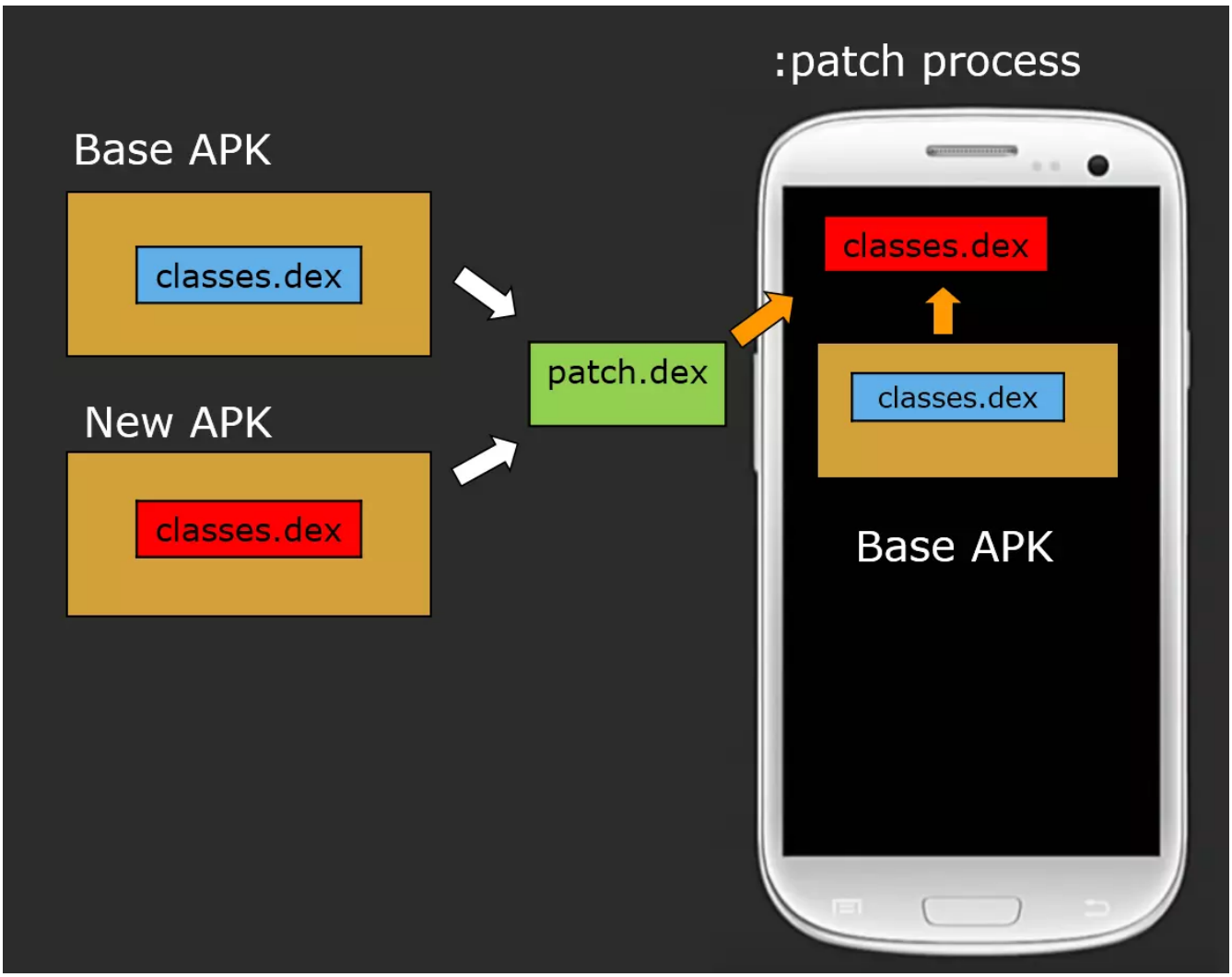
插件化原理：通过自定义 ClassLoader 来加载新的 dex 文件，从而让程序原本没有的类可以被使用，即动态加载

热更新和插件化的区别：

1，插件化的内容在原 App 中没有，而热更新是原 App 中的内容做了改动

2，插件化在代码中有固定的入口，而热更新则可能改变任意位置的代码

dex替换：以Tinker为例，Tinker 热修复的主要原理就是通过对比旧 APK 的 dex 文件与新 APK 的 dex 文件，生成补丁包，然后在 APP 中通过补丁包与旧 APK 的 dex 文件合成新的 dex 文件。流程如下图所示：



四十四、Hook**以及插桩技术****

Hook：翻译过来是钩子的意思，Hook技术，就是在事件传送到终点前截获并监控事件的传输，像个钩子钩上事件一样，并且能够在钩上事件时，处理一些自己特定的事件。

Hook 的选择点：静态变量和单例，因为一旦创建对象，它们不容易变化，非常容易定位。

Hook 过程：

寻找 Hook 点，原则是静态变量或者单例对象，尽量 Hook public 的对象和方法

选择合适的代理方式，如果是接口可以用动态代理

用代理对象替换原始对象

插桩技术：在保证被测程序原有逻辑完整性的基础上在程序中插入一些探针（又称为“探测仪”，本质上就是进行信息采集的代码段，可以是赋值语句或采集覆盖信息的函数调用），通过探针的执行并抛出程序运行的特征数据，通过对这些数据的分析，可以获得程序的控制流和数据流信息，进而得到逻辑覆盖等动态信息，从而实现测试目的的方法。

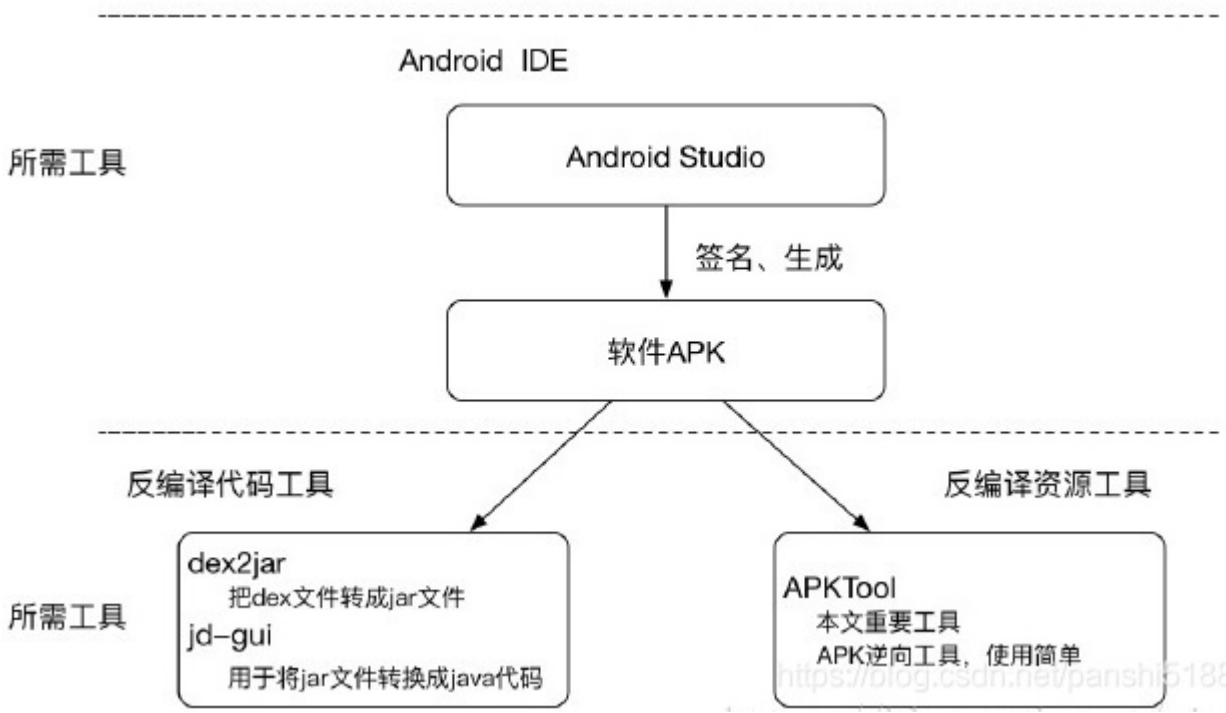
Hook与插桩的区别：

hook是通过代理等方式，可以随时取消这个hook

插桩是通过字节编码的方式，把修改后的方法重新编译成字节，然后比对的方式来修改源文件

四十五、逆向技术

逆向技术，狭义的讲就是apk反编译，主要用到了三个工具：dex2jar，jd-jui，APKTool，也有第三方工具，从下图可基本了解使用流程。



四十六、代码管理工具，为什么会产生代码冲突，该如何解决

常见代码管理工具：SVN是集中式管理，git是分布式管理

为什么产生代码冲突：两个用户修改了同一个文件的同一块区域，git不知道应该以哪份为准，就会报告内容冲突。