Android整理

数据结构

HashMap

数据结构

哈希表结构（链表散列：数组+链表）实现，结合数组和链表的优点。当链表长度超过 8 时，链表转换为红黑树。
transient Node<K,V>[] table;

工作原理

HashMap 底层是 hash 数组和单向链表实现，数组中的每个元素都是链表，由 Node 内部类（实现 Map.Entry<K,V>接口）实现，HashMap 通过 put & get 方法存储和获取。

存储对象时，将 K/V 键值传给 put() 方法：
①、调用 hash(K) 方法计算 K 的 hash 值，然后结合数组长度，计算得数组下标；
②、调整数组大小（当容器中的元素个数大于 capacity * loadfactor 时，容器会进行扩容resize 为 2n）；
③、i.如果 K 的 hash 值在 HashMap 中不存在，则执行插入，若存在，则发生碰撞；
ii.如果 K 的 hash 值在 HashMap 中存在，且它们两者 equals 返回 true，则更新键值对；
iii. 如果 K 的 hash 值在 HashMap 中存在，且它们两者 equals 返回 false，则插入链表的尾部（尾插法）或者红黑树中（树的添加方式）。

（JDK 1.7 之前使用头插法、JDK 1.8 使用尾插法）
（注意：当碰撞导致链表大于 TREEIFY_THRESHOLD = 8 时，就把链表转换成红黑树）

获取对象时，将 K 传给 get() 方法：①、调用 hash(K) 方法（计算 K 的 hash 值）从而获取该键值所在链表的数组下标；②、顺序遍历链表，equals()方法查找相同 Node 链表中 K 值对应的 V 值。

hashCode 是定位的，存储位置；equals是定性的，比较两者是否相等。

当两个对象的 hashCode 相同会发生什么

因为 hashCode 相同，不一定就是相等的（equals方法比较），所以两个对象所在数组的下标相同，"碰撞"就此发生。又因为 HashMap 使用链表存储对象，这个 Node 会存储到链表中。

你知道 hash 的实现吗？为什么要这样实现？

JDK 1.8 中，是通过 hashCode() 的高 16 位异或低 16 位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度，功效和质量来考虑的，减少系统的开销，也不会造成因为高位没有参与下标的计算，从而引起的碰撞。

为什么要用异或运算符?

保证了对象的 hashCode 的 32 位值只要有一位发生改变，整个 hash() 返回值就会改变。尽可能的减少碰撞。

相同0不同1
真⊕假=真
假⊕真=真
假⊕假=假
真⊕真=假

HashMap 的 table 的容量如何确定？loadFactor 是什么？ 该容量如何变化？这种变化会带来什么问题?

①、table 数组大小是由 capacity 这个参数确定的，默认是16，也可以构造时传入，最大限制是1<<30；
②、loadFactor 是装载因子，主要目的是用来确认table 数组是否需要动态扩展，默认值是0.75，比如 table 数组大小为 16，装载因子为 0.75 时，threshold 就是12，当 table 的实际大小超过 12 时，table就需要动态扩容；
③、扩容时，调用 resize() 方法，将 table 长度变为原来的两倍（注意是 table 长度，而不是 threshold）
④、如果数据很大的情况下，扩展时将会带来性能的损失，在性能要求很高的地方，这种损失很可能很致命。

HashMap中put方法的过程?

“调用哈希函数获取Key对应的hash值，再计算其数组下标；
- 如果没有出现哈希冲突，则直接放入数组；如果出现哈希冲突，则以链表的方式放在链表后面；
- 如果链表长度超过阀值(TREEIFY THRESHOLD==8)，就把链表转成红黑树，链表长度低于6，就把红黑树转回链表；
- 如果结点的key已经存在，则替换其value即可；
- 如果集合中的键值对大于12，调用resize方法进行数组扩容。”

数组扩容的过程?

创建一个新的数组，其容量为旧数组的两倍，并重新计算旧数组中结点的存储位置。结点在新数组中的位置只有两种，原下标位置或原下标+旧数组的大小。

拉链法导致的链表过深问题为什么不用二叉查找树代替，而选择红黑树？为什么不一直使用红黑树?

之所以选择红黑树是为了解决二叉查找树的缺陷，二叉查找树在特殊情况下会变成一条线性结构（这就跟原来使用链表结构一样了，造成很深的问题），遍历查找会非常慢。而红黑树在插入新数据后可能需要通过左旋，右旋、变色这些操作来保持平衡，引入红黑树就是为了查找数据快，解决链表查询深度的问题，我们知道红黑树属于平衡二叉树，但是为了保持“平衡”是需要付出代价的，但是该代价所损耗的资源要比遍历线性链表要少，所以当长度大于8的时候，会使用红黑树，如果链表长度很短的话，根本不需要引入红黑树，引入反而会慢。

说说你对红黑树的见解?

- 1、每个节点非红即黑
- 2、根节点总是黑色的
- 3、如果节点是红色的，则它的子节点必须是黑色的（反之不一定）
- 4、每个叶子节点都是黑色的空节点（NIL节点）
- 5、从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）

jdk8中对HashMap做了哪些改变?

- 在java 1.8中，如果链表的长度超过了8，那么链表将转换为红黑树。（桶的数量必须大于64，小于64的时候只会扩容）
- 发生hash碰撞时，java 1.7 会在链表的头部插入，而java 1.8会在链表的尾部插入
- 在java 1.8中，Entry被Node替代(换了一个马甲)。

HashMap，LinkedHashMap，TreeMap 有什么区别?

HashMap 参考其他问题；
TreeMap 实现 SortMap 接口，能够把它保存的记录根据键排序（默认按键值升序排序，也可以指定排序的比较器）
LinkedHashMap 保存了记录的插入顺序，在用 Iterator 遍历时，先取到的记录肯定是先插入的；遍历比 HashMap 慢；

HashMap & TreeMap & LinkedHashMap 使用场景?

一般情况下，使用最多的是 HashMap。
HashMap：在 Map 中插入、删除和定位元素时；
TreeMap：在需要按自然顺序或自定义顺序遍历键的情况下；
LinkedHashMap：在需要输出的顺序和输入的顺序相同的情况下。

HashMap 和 HashTable 有什么区别?

①、HashMap 是线程不安全的，HashTable 是线程安全的；
②、由于线程安全，所以 HashTable 的效率比不上 HashMap；
③、HashMap最多只允许一条记录的键为null，允许多条记录的值为null，而 HashTable不允许；
④、HashMap 默认初始化数组的大小为16，HashTable 为 11，前者扩容时，扩大两倍，后者扩大两倍+1；
⑤、HashMap 需要重新计算 hash 值，而 HashTable 直接使用对象的 hashCode

Java 中的另一个线程安全的与 HashMap 极其类似的类是什么？同样是线程安全，它与 HashTable 在线程同步上有什么不同?

ConcurrentHashMap 类（是 Java并发包 java.util.concurrent 中提供的一个线程安全且高效的 HashMap 实现）。
HashTable 是使用 synchronize 关键字加锁的原理（就是对对象加锁）；
而针对 ConcurrentHashMap，在 JDK 1.7 中采用 分段锁的方式；JDK 1.8 中直接采用了CAS（无锁算法）+ synchronized。

HashMap & ConcurrentHashMap 的区别?

除了加锁，原理上无太大区别。另外，HashMap 的键值对允许有null，但是ConCurrentHashMap 都不允许。

为什么要设置这个加载因子

当加载因子增大，意味着到时候填充的元素就越多了(因为扩容的门槛上升)，空间利用率提升了，但是hash冲突的概率就会增大会降低查找效率，是一种时间换空间的做法。当加载因子减少，意味着填充的元素越少(扩容门槛的下降会导致插入没有多少元素就会进行一次扩容)，这样空间利用率就低了，但是hash冲突的概率会降低提升了查找效率，是一种以空间换时间的做法。这样可以根据项目本身的需要来设置不同的加载因子

加载因子=哈希表中的元素/哈希表的长度
所以加载因子越小哈希冲突的概率越小
装填因子的最大值一般选在0.65~0.9之间。比如HashMap中就将加载因子定为0.75。

默认加载因子为什么是0.75?

注释里面已经有给出，下面这段注释大致意思是说HashMap的Node分步频率服从λ为0.5的[泊松分布]，当长度为16的数组中存入了12个数据的时候，其中一个链表的长度为8的概率只有 0.00000006(链表长度大于8的时候会转化成红黑树)

如果是0.5 ， 那么每次达到容量的一半就进行扩容，默认容量是16， 达到8就扩容成32，达到16就扩容成64， 最终使用空间和未使用空间的差值会逐渐增加，空间利用率低下。

如果是1，那意味着每次空间使用完毕才扩容，在一定程度上会增加put时候的时间。

StackOverFlow： 一个bucket空和非空的概率为0.5，通过牛顿二项式等数学计算，得到这个loadfactor的值为log（2），约等于0.693. 同回答者所说，可能小于0.75 大于等于log（2）的factor都能提供更好的性能，0.75这个数说不定是 pulled out of a hat。

通常来讲，默认的加载因子(0.75)能够在时间和空间上提供一个好的平衡，更高的值会减少空间上的开支但是会增加查询花费的时间（体现在HashMap类中get、put方法上），当设置初始化容量时，应该考虑到map中会存放
entry的数量和加载因子，以便最少次数的进行rehash操作，如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作。

如何减少hash冲突?如何处理hash冲突?

除留取余法
链地址法：在碰到哈希冲突的时候，将冲突的元素以链表的形式进行存储。

链地址法的弊端与优化?

极端情况下哈希表可能退化成了一个链表。（每个对象存储都会碰撞）
优化：当哈希表中的链表过长时我们就可以把这个链表变成一棵红黑树

Java8中有什么改进?

Hash冲突后不再是用链表来保存相同index的节点，相应的采用红黑树（高性能的平衡树）来保存冲突节点。
节点查找优先级由 O（n）-> 提高到了O（logn）

为什么初始容量必须是2的n次幂值

```
tab[i = (n - 1) & hash])
```

这里会先判断(n-1)&hash值在数组里面是否存在，我们假设当前初始容量是16，那么n-1就是15换成二进制就是1111，这样与后面的hash做与运算就是等于hash值，所以只要hash值不一样，那么就不会出现hash冲突的情况。那如果初始容量不是2的n次幂值呢，假设为15，那么n-1就是14换成二进制就是1110，这样后面的hash值只要前三位一样，后面一位不管是0还是1，的出来的值就是一样的，这样会增大hash冲突的概率。所以设置成2的n次幂的值主要就是为了减少hash冲突。

JDK1.7 ~ JDK1.8在resize()的区别有哪些?

- 扩容时机：1.7是先判断>阀值扩容后再插入数据，1.8是先插入数据然后判断>阀值扩容。
- 存储位置计算：1.7 重新hashCode，扰动处理然后取模运算，并且每一个数据单独计算，1.8原位置/原位置+旧容量，在转移数据时统一计算。
- 扩容后转移数据：1.7使用头插法，原位置数据往后移动，会出现逆序/环形链表死循环问题，1.8使用尾插法，防止出现死循环，逆序，环形链表等问题。

threshold(不是加载因子LoadFactor)

threshold 这个阈值 在构造方法里，确实是指的容量。
在resize方法里，如果数组未初始化，会用这个值作为数组的容量，
然后在计算出新的threshold

为啥1.8之前用头插法，java8之后改成尾插了呢?

因为resize的赋值方式，也就是使用了**单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置**，在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

使用头插会改变链表的上的顺序，但是如果**使用尾插**，在扩容时会保持链表元素原本的顺序，就不会出现链表成环的问题了。

Java7在多线程操作HashMap时可能引起死循环，原因是扩容转移后前后链表顺序倒置，在转移过程中修改了原来链表中节点的引用关系。
Java8在同样的前提下并不会引起死循环，原因是扩容转移后前后链表顺序不变，保持之前节点的引用关系。

那是不是意味着Java8就可以把HashMap用在多线程中呢?

我认为即使不会出现死循环，但是通过源码看到put/get方法都没有加同步锁，多线程情况最容易出现的就是：无法保证上一秒put的值，下一秒get的时候还是原值，所以线程安全还是无法保证。

初始化大小为啥是16

因为在使用是2的幂的数字的时候，Length-1的值是所有二进制位全为1，这种情况下，index的结果等同于HashCode后几位的值。
只要输入的HashCode本身分布均匀，Hash算法的结果就是均匀的。
这是为了**实现均匀分布**。

为啥我们重写equals方法的时候需要重写hashCode方法呢?

因为在java中，所有的对象都是继承于Object类。Ojbect类中有两个方法equals、hashCode，这两个方法都是用来比较两个对象是否相等的。
在未重写equals方法我们是继承了object的equals方法，**那里的 equals是比较两个对象的内存地址**，显然我们new了2个对象内存地址肯定不一样
hashCode方法重写，以保证相同的对象返回相同的hash值，不同的对象返回不同的hash值。

准备用HashMap存1w条数据，构造时传10000还会触发扩容吗?

不会，当我们从外部传递进来 1w 时，实际上经过 `tableSizeFor()` 方法处理之后，就会变成 2 的 14 次幂 16384，再算上负载因子 0.75f，实际在不触发扩容的前提下，可存储的数据容量是 12288（16384 * 0.75f）。

1000呢？
虽然 HashMap 初始容量指定为 1000，会被 `tableSizeFor()` 调整为 1024，但是它只是表示 table 数组为 1024，扩容的重要依据扩容阈值会在 `resize()` 中调整为 768（1024 * 0.75）。
它是不足以承载 1000 条数据的，最终在存够 1k 条数据之前，还会触发一次动态扩容。

例如想要用 HashMap 存放 1k 条数据，应该设置 1000 / 0.75，实际传递进去的值是 1333，然后会被 `tableSizeFor()` 方法调整到 2048，足够存储数据而不会触发扩容。

当想用 HashMap 存放 1w 条数据时，依然设置 10000 / 0.75，实际传递进去的值是 13333，会被调整到 16384，和我们直接传递 10000 效果是一样的。

HashMap扩容原理，为什么要2的指数幂容量，如果输入17会是多少容量?

跟HashCode有关系，如果输入17，会向后达到2的指数幂，32。

HashMap的缺点，怎么解决

线程不安全

事先确定hashmap的大小，防止扩容
key通过hash计算存储在数组中某个槽位，如果一个类的hashcode与其成员变量name有关，name变更了，那么hashmap行为将不正常。

1.7的时候缺点
如果hash相同，链表就会很长，变为单链表
1.7使用**头插法**，原位置数据往后移动，会出现逆序/环形链表死循环问题，1.8使用**尾插法**，防止出现死循环，逆序，环形链表等问题。

ConcurrentHashMap

ConcurrentHashMap 简单介绍?

①、重要的常量：
```
private transient volatile int sizeCtl;
```
当为负数时，-1 表示正在初始化，-N 表示 N - 1 个线程正在进行扩容；
当为 0 时，表示 table 还没有初始化；
当为其他正数时，表示初始化或者下一次进行扩容的大小。

②、数据结构：
Node 是存储结构的基本单元，继承 HashMap 中的 Entry，用于存储数据；
TreeNode 继承 Node，但是数据结构换成了二叉树结构，是红黑树的存储结构，用于红黑树中存储数据；
TreeBin 是封装 TreeNode 的容器，提供转换红黑树的一些条件和锁的控制。

③、存储对象时（put() 方法）：
1.如果没有初始化，就调用 initTable() 方法来进行初始化；
2.如果没有 hash 冲突就直接 CAS 无锁插入；
3.如果需要扩容，就先进行扩容；
4.如果存在 hash 冲突，就加锁来保证线程安全，两种情况：一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入；
5.如果该链表的数量大于阀值 8，就要先转换成红黑树的结构，break 再一次进入循环
6.如果添加成功就调用 addCount() 方法统计 size，并且检查是否需要扩容。

④、扩容方法 transfer()：默认容量为 16，扩容时，容量变为原来的两倍。
helpTransfer()：调用多个工作线程一起帮助进行扩容，这样的效率就会更高。

⑤、获取对象时（get()方法）：
1.计算 hash 值，定位到该 table 索引位置，如果是首结点符合就返回；
2.如果遇到扩容时，会调用标记正在扩容结点 ForwardingNode.find()方法，查找该结点，匹配就返回；
3.以上都不符合的话，就往下遍历结点，匹配就返回，否则最后就返回 null。

ConcurrentHashMap 的并发度是什么?

程序运行时能够同时更新 ConccurentHashMap 且不产生锁竞争的最大线程数。默认为 16，且可以在构造函数中设置。当用户设置并发度时，ConcurrentHashMap 会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）

为什么 ConcurrentHashMap 比 HashTable 效率要高?

HashTable 使用一把锁（锁住整个链表结构）处理并发问题，多个线程竞争一把锁，容易阻塞；

ConcurrentHashMap
- JDK 1.7 中使用分段锁（ReentrantLock + Segment + HashEntry），相当于把一个 HashMap 分成多个段，每段分配一把锁，这样支持多线程访问。锁粒度：基于 Segment，包含多个 HashEntry。
- JDK 1.8 中使用 CAS + synchronized + Node + 红黑树。锁粒度：Node（首结点）（实现 Map.Entry<K,V>）。锁粒度降低了。

针对 ConcurrentHashMap 锁机制具体分析（JDK 1.7 VS JDK 1.8）？

JDK 1.7 中，采用分段锁的机制，实现并发的更新操作，底层采用数组+链表的存储结构，包括两个核心静态内部类 Segment 和 HashEntry。
①、Segment 继承 ReentrantLock（重入锁） 用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶；
②、HashEntry 用来封装映射表的键-值对；
③、每个桶是由若干个 HashEntry 对象链接起来的链表

JDK 1.8 中，采用Node + CAS + Synchronized来保证并发安全。取消类 Segment，直接用 table 数组存储键值对；当 HashEntry 对象组成的链表长度超过 TREEIFY_THRESHOLD 时，链表转换为红黑树，提升性能。底层变更为数组 + 链表 + 红黑树。

ConcurrentHashMap 在 JDK 1.8 中，为什么要使用内置锁 synchronized 来代替重入锁ReentrantLock?

①、粒度降低了；
②、JVM 开发团队没有放弃 synchronized，而且基于 JVM 的 synchronized 优化空间更大，更加自然。
③、在大量的数据操作下，对于 JVM 的内存压力，基于 API 的 ReentrantLock 会开销更多的内存。

CurrentHashMap 读写锁是如何实现的?

如果没有hash冲突的情况下，使用CAS进行插入，如果有hash冲突，使用synchronized加锁进行插入。当链表长度大于8且数据长度>=64时，会使用红黑树替代链表。
ReentrantReadWriteLock类中有readLock()和writeLock()方法可以分别获取WriteLock和ReadLock对象，调用他们的lock方法，就可以实现读写锁。实现原理就是AQS的acquire和acquireShared方法。

put如何保证线程安全
当对table中的某个节点进行写操作的时候，如果为null,直接进行插入，注意如果不为null，则用syncronized关键字锁住当前节点，然后进行写操作。为什么为空的时候插入不需要上锁，因为ConcurrentHashMap采用的是CAS的乐观锁机制，如果有线程在进行修改，就会在外面等待，然后过一会儿再过来查询一下，这样可以保证高效并发。

get如何保证线程安全
首先当多个线程同时读取的时候，ConcurrentHashMap给每个Node节点加了volatile关键字修饰来保证可见性。

concurrentHashMap get方法实现为什么不会导致脏数据

volatile Node[] table;

List

List加锁要如何加

SynchronizedList继承于SynchronizedCollection，使用装饰者模式，为原来的List加上锁，从而使List同步安全**
可以指定一个对象作为锁，如果不指定，默认就锁了集合了。
```
List<Long> sycList = Collections.synchronizedList(originList);
```

arraylist

arraylist:
是一个动态数组，其底层数据结构依然是数组，查询速度非常快，时间复杂度O1。

构造：
无参数：赋值空数组
自定义容量：创建数组
集合：toArray赋值给，如果出错没有返回Object[]时，调用Arrays.copyOf来复制集合

增：
直接添加：
判断扩容
数组末尾添加元素，并修改size

索引添加：
判断越界
判断扩容
将索引开始的数据 向后移动一位
索引位置赋值
修改size

添加集合：
判断扩容
复制数组
修改size

索引添加集合：
判断越界
判断扩容
将索引开始的数据 向后移动集合的长度
复制数组
修改size

扩容：
默认扩容一半。如果扩容一半不够，就用目标的size作为扩容后的容量。
扩容成功后，会修改modCount。

删
索引删除：
判断越界
修改modCount
获取删除元素
将索引位置，往前移动一位，进行覆盖
将数组尾部数据置空
修改size

元素删除：
循环遍历查找到相等的元素
查到了，修改modCount
将索引位置，往前移动一位，进行覆盖
将数组尾部数据置空
修改size

批量删除：
循环遍历，如果不包含则保留该元素（包含说明有相同的元素，要删除的，不能保留）
同时记录批量删除后，数组还剩多少元素
出现异常处后面的数据全部复制覆盖到数组里(当前索引和数组长度不相等)
循环置空多余的元素
修改modCount
修改size

改（不会修改modCount）
判断越界

取出元素
修改元素
返回元素

查（不会修改modCount）
越界判断
获取元素
返回元素

迭代器 Iterator:
cursor：默认是0
lastRet：上一次返回的元素 (删除的标志位) -1
expectedModCount = modCount： 用于判断集合是否修改过结构的标志

iterator:
new Itr()

hasNext:
返回当前游标不等于数组长度

next：
判断是否修改过了List的结构，如果有修改，抛出异常
判断是否越界
赋值，再次判断是否越界
游标+1
返回元素 ，并设置上一次返回的元素的下标

remove：（remove掉上一次next的元素）
先判断是否next过，没有next抛异常
判断是否修改过
删除元素（remove方法）
删除的游标
修改删除的标志位
更新判断集合是否修改的标志

lrucache

lrucache怎么实现？为什么是o(1)的时间复杂度？

LruCache(线程安全)内部数据结构是利用LinkedHashMap
将数据存放在链表中，如果有新数据，判断容量是否达到上限，
未超出：直接插入到尾部
超出：删除头部元素，在插入到对尾
新数据在链表中存在：将数据从前驱节点和后继节点中提出，重新插入到尾部

构造方法：创建LinkedHashMap的accessOrder为true代表排序
accessOrder设置为true则为访问顺序，为false，则为插入顺序。即最近访问的最后输出

put：方法将移动到链表的尾端，返回先前的value，有旧值移除 回调entryRemoved ，调用trimToSize 判断当前size是否超过maxSize
trimToSize：移除头结点，即近期最少访问的，直到缓存大小小于最大值。
get： 指定 key 对应的 value 值存在时返回，将访问的元素更新到队列头部的功能

LinkedHashMap

LinkedHashMap是 HashMap + 双向链表。
put：没有重写，调用的是hashMap的put，内部调用了addEntry
addEntry：让用户重写removeEldestEntry 来决定是否移除eldest节点
get：使用HashMap的getEntry方法，验证了我们所说的查询效率为O(1) ，e.recordAccess 在找到节点后调用节点recordAccess方法
recordAccess：将本身节点移除链表 ，将自己添加到节点头部 保证最近使用的节点位于链表前边

map 接口继承关系， set接口继承关系， list 接口继承关系

```
map:
AbstractMap implements Map
HashMap extends AbstractMap implements Map
LinkedHashMap extends HashMap implements Map
ConcurrentHashMap extends AbstractMap implements ConcurrentMap
ArrayMap extends SimpleArrayMap implements Map
Hashtable extends Dictionary implements Map

SortedMap extends Map
NavigableMap extends SortedMap
TreeMap extends AbstractMap implements NavigableMap



set:
AbstractSet extends AbstractCollection implements Set
HashSet extends AbstractSet implements Set
LinkedHashSet extends HashSet implements Set
ArraySet implements Collection, Set

SortedSet extends Set
NavigableSet extends SortedSet
TreeSet extends AbstractSet
```

```
list:
AbstractList extends AbstractCollection implements List
LinkedList extends AbstractSequentialList implements List, Deque
ArrayList extends AbstractList implements List

Vector extends AbstractList implements List
Stack extends Vector
```

B+树和二叉树的区别，性能对比

https://mp.weixin.qq.com/s/9zdhSU1XYHsXsKxtSn4mCw

https://mp.weixin.qq.com/s?__biz=MzIxNzU1Nzk3OQ==&mid=2247485932&idx=1&sn=d412fb907a70ae159a7c19f715797586&chksm=97f6b758a0813e4e459e98f338e649de44289b89ed234794250c4f514b67649aabb8888c4a81&scene=38#wechat_redirect

二叉树极端情况下会退化为单链表，所以才用到了平衡树。
平衡树降低了树的高度，加快了查询速度。
红黑树：因为平衡树要求每个节点的左子树和右子树的高度差至多等于1，导致每次进行插入/删除节点的时候，几乎都需要通过左旋和右旋来进行调整，所以有了红黑树。

B树是多路搜索树，可以拥有多于2个孩子节点，M路的B数最多能拥有M个孩子节点。
这样设计进一步降低了数的高度。
B树常用于文件系统和数据库索引，这2个都是存在磁盘上的，数据量大的话，不一定能一次性加载到内存中。
所以B树多路存储，每次加载B树的一个节点然后一步步往下找。
如果在内存中红黑树就比B树效率高了，但是在磁盘中B树就更优了。

B+树：升级版B树，数据都在叶子节点，同时叶子节点之间加了指针，形成链表。常用于数据库索引中。
数据库中查询语句是多条的话，B树需要做局部中序遍历，可能要跨层访问，而B+树所有数据都在叶子结点，不用跨层，又是链表结构，只需要找到首尾，通过链表就能把所有数据取出来了。

B+和Hash：查询一条数据HASH更快，多条数据B+更强。

ArrayMap的原理，为什么内存消耗低

ArrayMap是Android专门针对内存优化而设计的，用于取代Java API中的HashMap数据结构。
而SparseArray和ArrayMap性能略逊于HashMap，但更节省内存。

数据结构：2个数组，hashcode值组成的数组，key-value键值对所组成的数组，是hash数组大小的2倍

两个非常重要的静态成员变量：mBaseCache和mTwiceBaseCacheSize，用于ArrayMap所在进程的全局缓存功能
mBaseCache：用于缓存大小为4的ArrayMap
mTwiceBaseCacheSize：用于缓存大小为8的ArrayMap
两个缓存如果超过10个则不在缓存。
为了减少频繁地创建和回收Map对象，ArrayMap采用了两个大小为10的缓存队列来分别保存大小为4和8的Map对象。为了节省内存有更加保守的内存扩张以及内存收缩策略。

释放内存：
array[0]和[1]分别指向原来的缓存池，其他数据置空，赋值给缓存变量
分配内存：
缓存不为空，从缓存池中取出缓存的mArray和mHashes，将缓存池指向上一条缓存地址，再把mArray的第0和第1个位置的数据置为null。

容量扩张：当mSize大于或等于mHashes数组长度时则扩容
osize<4时,=4
4<=osize< 8,=8
osize>=8,*1.5

容量收紧:数组大小大于8，且存储数据的个数小于数组空间大小的1/3情况下，需要收紧数据，分配新数组，老的内存靠虚拟机自动回收。

构造方法：指定大小则会去分配相应大小的内存，如果没有指定默认为0，需要添加数据的时候再扩容。

put：采用二分查找，找到key，index返回正值，直接修改，没有找到返回负值，按位取反后所对应的值代表需要插入的位置。
插入时，判断扩容

append：轻量级插入，不会去扩容，明确知道肯定会插入队尾的情况下使用append()性能更好

remove:通过二分查找key的index，再根据index来选择移除动作；
移除的是最后一个元素，则释放该内存，否则只做移除操作。同时会根据容量收紧原则来决定是否要收紧，当需要收紧时会创建一个更小内存的容量。

clear：清理操作会执行freeArrays()方法来回收内存

ArrayMap中解决Hash冲突的方式是追加的方式，比如两个key的hash(int)值一样，那就把数据全部后移一位，通过追加的方式解决Hash冲突。

SparseArray中Key为int类型（避免了装箱和拆箱）
它使用了两个数组，一个保存key，一个保存value。Key数组Int值是按顺序排列的，查找是二分查找，Value数组位置和Key数组位置对应。
add的时候会进行位移，remove的时候不一定会进行位移，把某个值标记为delete，如果下次有符合的值直接放到该位置，就没有位移了。
从内存使用上来说，SparseArray不需要保存key所对应的哈希值，所以比ArrayMap还能再节省1/3的内存。

ArraySet用于替代HashSet。和ArrayMap差不多

总结一下：

数据结构
ArrayMap和SparseArray采用的都是两个数组，Android专门针对内存优化而设计的
HashMap采用的是数据+链表+红黑树
内存优化
ArrayMap比HashMap更节省内存，综合性能方面在数据量不大的情况下，推荐使用ArrayMap；
Hash需要创建一个额外对象来保存每一个放入map的entry，且容量的利用率比ArrayMap低，整体更消耗内存
SparseArray比ArrayMap节省1/3的内存，但SparseArray只能用于key为int类型的Map，所以int类型的Map数据推荐使用SparseArray；
性能方面：
ArrayMap查找时间复杂度O(logN)；ArrayMap增加、删除操作需要移动成员，速度相比较慢，对于个数小于1000的情况下，性能基本没有明显差异
HashMap查找、修改的时间复杂度为O(1)；
SparseArray适合频繁删除和插入来回执行的场景，性能比较好
缓存机制

ArrayMap针对容量为4和8的对象进行缓存，可避免频繁创建对象而分配内存与GC操作，这两个缓存池大小的上限为10个，防止缓存池无限增大；
HashMap没有缓存机制
SparseArray有延迟回收机制，提供删除效率，同时减少数组成员来回拷贝的次数
扩容机制
ArrayMap是在容量满的时机触发容量扩大至原来的1.5倍，在容量不足1/3时触发内存收缩至原来的0.5倍，更节省的内存扩容机制
HashMap是在容量的0.75倍时触发容量扩大至原来的2倍，且没有内存收缩机制。HashMap扩容过程有hash重建，相对耗时。所以能大致知道数据量，可指定创建指定容量的对象，能减少性能浪费。
并发问题
ArrayMap是非线程安全的类，大量方法中通过对mSize判断是否发生并发，来决定抛出异常。但没有覆盖到所有并发场景，比如大小没有改变而成员内容改变的情况就没有覆盖
HashMap是在每次增加、删除、清空操作的过程将modCount加1，在关键方法内进入时记录当前mCount，执行完核心逻辑后，再检测mCount是否被其他线程修改，来决定抛出异常。这一点的处理比ArrayMap更有全面。

算法

手写list的反转(注重细节)

```
public static void reverse(int[] arr) {
    for (int i = 0; i < arr.length / 2; i++) {
        int temp = arr[i];
        arr[i] = arr[arr.length - i - 1];
        arr[arr.length - i - 1] = temp;
    }
}
public static void reverseList(List arr) {
    for (int i = 0; i < arr.size() / 2; i++) {
        Object temp = arr.get(i);
        arr.set(i, arr.get(arr.size() - i - 1));
        arr.set(arr.size() - i - 1, temp);
    }
}
```

求二叉树中两个节点之间的最大距离。

```
public class FindMaxDistance {

    private int maxLen=0;

    public void findMaxDistance(Node root){
        if(root==null){
            return;
        }
        if(root.left==null){
            root.maxLeftDistance=0;
        }
        if(root.right==null){
            root.maxRightDistance=0;
```

```
        }

        //递归计算左右子树中每个节点最长距离
        if(root.left!=null){
            findMaxDistance(root.left);
        }
        if(root.right!=null){
            findMaxDistance(root.right);
        }

        if(root.left!=null){

 root.maxLeftDistance=max(root.left.maxLeftDistance,root.left.maxRightDistance)
                    +1;
        }

        if(root.right!=null){

 root.maxRightDistance=max(root.right.maxLeftDistance,root.right.maxRightDistanc
e)
                    +1;
        }

        if(maxLen<root.maxLeftDistance+root.maxRightDistance){
            maxLen=root.maxLeftDistance+root.maxRightDistance;
        }
    }

    private int max(int i, int j) {
        return i>j?i:j;
    }

    class Node{
        public int value;
        public Node left;
        public Node right;
        public int maxLeftDistance;
        public int maxRightDistance;

        public Node(int value){
            this.value=value;
            this.left=null;
            this.right=null;
        }
    }


    /**
     * @param args
     */
    public static void main(String[] args) {
        //
        FindMaxDistance fad=new FindMaxDistance();
        Node root1=fad.buildTestTree1();
        fad.findMaxDistance(root1);
        System.out.println(fad.maxLen);

        fad.maxLen=0;
```

```
        Node root2=fad.buildTestTree2();
        fad.findMaxDistance(root2);
        System.out.println(fad.maxLen);
    }

    private Node buildTestTree1(){
        Node root=new Node(1);
        root.left=new Node(2);
        root.right=new Node(3);

        root.left.left=new Node(4);
        root.left.right=new Node(5);

        root.left.left.left=new Node(6);

        root.right.left=new Node(7);
        root.right.right=new Node(8);

        root.right.left.right=new Node(9);

        return root;
    }

    private Node buildTestTree2(){
        Node root=new Node(1);
        root.left=new Node(2);

        root.left.left=new Node(4);
        root.left.right=new Node(5);

        root.left.left.left=new Node(6);

        root.left.right.right=new Node(7);

        return root;
    }
}
```

任意一颗二叉树，求最大节点距离

求深度就可以

```
fun maxDepth(root: TreeNode?): Int {
    if (root == null) {
        return 0
    } else {
        val leftHeight = maxDepth(root.left)
        val rightHeight = maxDepth(root.right)
        return Math.max(leftHeight, rightHeight) + 1
    }
}
```

三色球排序

```
public class Test3 {
    public static void main(String[] args) {
        int arr[] = {2,1,0,1,2,0,0,1,2};
        sort(arr);
        String s = Arrays.toString(arr);
        System.out.println(s);
    }

    public static void sort(int[] arr) {
        int begin = 0;
        int curr = 0;
        int end = arr.length - 1;
        while (curr <= end) {
            if (arr[curr] == 0) {
                swap(arr, begin, curr);
                begin++;
                curr++;
            } else if (arr[curr] == 1) {
                curr++;
            } else {
                swap(arr, curr, end);
                end--;
            }
        }
    }

    public static void swap(int[] arr, int val1, int val2) {
        int tmp = arr[val1];
        arr[val1] = arr[val2];
        arr[val2] = tmp;
    }
}
```

斐波拉契数列，递归的方式怎么优化?

https://blog.csdn.net/weixin_41463193/article/details/94707938

```
 public static int Fibonacci3(int n) {
        // 循环版本，空间复杂度为O(1)
        // 当n小于2时，就不用申请数组空间了，直接返回n
        if (n < 2)
            return n;
        // 标记上一个值，类似nums[n - 1]
        int pre = 1;
        // 标记上上一个值，类似nums[n - 2]
        int prePre = 1;
        // i从3开始，因为前两个元素已经确定，接下来要从第3个元素开始计算
        int i = 3;
        // 返回的结果
        int result = 1;
        while (i <= n) {
            // result等于前两个数相加
            result = pre + prePre;
            // 此时的prePre，即上上个数等于上个数
```

```
            prePre = pre;
            // 上个数就等于result
            pre = result;
            i++;
        }
        return result;
    }
```

快速排序？快速排序是稳定的么？

不稳定

https://blog.csdn.net/u014702653/article/details/100602842

https://mp.weixin.qq.com/s/GiNFE1dwmVtA99qxccK3aQ

```
    /**
     * 快速排序
     *
     * @param array 目标数组
     * @param left  数组最左下标值
     * @param right 数组最右下标值
     */
    public static void quickSort(int[] array, int left, int right) {

        //防止数组越界(这一步最后再回来理解)
        if (left >= right) return;

        int base = array[left];//基准数
        int i = left, j = right;//左探子i，右探子j
        int temp;//用于交换的临时变量

        while (i != j) {//两个探子相遇前

            //找出小于 基准数 的下标，当大于等于 基准数 的时候，探子j 向左移动即可（j--）
            while (array[j] >= base && i < j) {
                j--;
            }

            //找出大于 基准数 的下标，当小于等于 基准数 的时候，探子i 向右移动即可（i++）
            while (array[i] <= base && i < j) {
                i++;
            }

            //经过上面两个循环之后，探子j，探子i 都会停在符合条件的值的下标上面，然后就可以进行交换值啦
            if (i < j) {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        //经过上面的循环之后会触发 i=j 的条件（即两个探子相遇），此时将基准值的数与i或者j（随便哪个都一样）下标的值交换，完成一次交换探测
        array[left] = array[i];//基准值的位置
```

```
        array[i] = base;//基准值

        //完成交换之后再对左右两边的子数组进行递归调用走上面的方法，完成整个数组的排序
        quickSort(array, left, i - 1);//左边
        quickSort(array, i + 1, right); //右边
    }
```

minstack怎么设计?

https://mp.weixin.qq.com/s/SFbGNAEGnqZMib9VMC26Ew

```
import java.util.ArrayList;
import java.util.List;

/**
 * @author xiaoshi on 2018/9/1.
 */
public class MinStack {

    private List<Integer> data = new ArrayList<Integer>();
    private List<Integer> mins = new ArrayList<Integer>();

    public void push(int num) throws Exception {
        data.add(num);
        if(mins.size() == 0) {
            // 初始化mins
            mins.add(0);
        } else {
            // 辅助栈mins push最小值的索引
            int min = getMin();
            if (num < min) {
                mins.add(data.size() - 1);
            }
        }
    }

    public int pop() throws Exception {
        // 栈空，抛出异常
        if(data.size() == 0) {
            throw new Exception("栈为空");
        }
        // pop时先获取索引
        int popIndex = data.size() - 1;
        // 获取mins栈顶元素，它是最小值索引
        int minIndex = mins.get(mins.size() - 1);
        // 如果pop出去的索引就是最小值索引，mins才出栈
        if(popIndex == minIndex) {
            mins.remove(mins.size() - 1);
        }
        return data.remove(data.size() - 1);
    }

    public int getMin() throws Exception {
        // 栈空，抛出异常
        if(data.size() == 0) {
            throw new Exception("栈为空");
        }
```

```
        // 获取mins栈顶元素，它是最小值索引
        int minIndex = mins.get(mins.size() - 1);
        return data.get(minIndex);
    }

}
```

怎么用一个数组实现一个栈结构，说说思路。

```
public class IntStack implements IStack {
    //默认容量是10个
    private static final int CAPITILY = 10;
    //扩容因子,默认是扩充之前的两倍
    private static final int FACTOR = 2;
    //集合元素的个数
    private int mSize = 0;
    //核心数组，注意数组的长度>=集合长度(mArray.length>=mSize)
    private int[] mArray = new int[CAPITILY];

    @Override
    public int push(int element) {
        if(mSize>=mArray.length){
            //如果数据满了需要考虑扩容了,这里默认是扩容为之前的两倍
            //非常不推荐用Arrays.copy方法，其本质也是用了下面这个方法
            //System.arraycopy真的很重要，基本上数组扩容都是这个方法做的
            int[] newArray = new int[mArray.length*FACTOR];
            System.arraycopy(mArray,0,newArray,0,mArray.length);
            mArray = newArray;

        }
        mArray[mSize++] = element;
        return element;
    }

    @Override
    public int pop() {
        if(mSize>0){
            int result = mArray[mSize-1];
            mArray[mSize - 1] = 0;
            mSize--;
            return result;
        }else {
            throw new EmptyStackException();
        }

    }

    @Override
    public int peek() {
        if(mSize>0){
            return mArray[mSize - 1];
        }else {
            throw new EmptyStackException();
        }
    }

    @Override
```

```
    public boolean empty() {
        return mSize==0;
    }

    @Override
    public int size() {
        return mSize;
    }

}
```

写一个二叉树的层序遍历

层次遍历只有 BFS（广搜）

```
public static List<List<String>> levelOrder(TreeNode root) {
    List result = new ArrayList();
    if (root == null) {
        return result;
    }
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        ArrayList<String> level = new ArrayList<String>();
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode head = queue.poll();
            level.add(head.val);
            if (head.left != null) {
                queue.offer(head.left);
            }
            if (head.right != null) {
                queue.offer(head.right);
            }
        }
        result.add(level);
    }
    return result;
}
```

常用的对称加密算法，有什么同

```
DES：数据加密标准，是对称加密算法领域中的典型算法
特点：密钥偏短（56位）、生命周期短（避免被破解）

3DES：将密钥长度增至112位或168位，通过增加迭代次数提高安全性
缺点：处理速度较慢、密钥计算时间较长、加密效率不高

AES：高级数据加密标准，能够有效抵御已知的针对DES算法的所有攻击
特点：密钥建立时间短、灵敏性好、内存需求低、安全性高
```

快速排序

```
/**
 * 快速排序
 *
 * @param array 目标数组
 * @param left  数组最左下标值
 * @param right 数组最右下标值
 */
public static void quickSort(int[] array, int left, int right) {
    //防止数组越界(这一步最后再回来理解)
    if (left >= right) return;
    int base = array[left];//基准数
    int i = left, j = right;//左探子i，右探子j
    int temp;//用于交换的临时变量
    while (i != j) {//两个探子相遇前
        //找出小于 基准数 的下标，当大于等于 基准数 的时候，探子j 向左移动即可（j--）
        while (array[j] >= base && i < j) {
            j--;
        }
        //找出大于 基准数 的下标，当小于等于 基准数 的时候，探子i 向右移动即可（i++）
        while (array[i] <= base && i < j) {
            i++;
        }
        //经过上面两个循环之后，探子j，探子i 都会停在符合条件的值的下标上面，然后就可以进行交换值啦
        if (i < j) {
            temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
    //经过上面的循环之后会触发 i=j 的条件（即两个探子相遇），此时将基准值的数与i或者j（随便哪个都一样）下标的值交换，完成一次交换探测
    array[left] = array[i];//基准值的位置
    array[i] = base;//基准值
    //完成交换之后再对左右两边的子数组进行递归调用走上面的方法，完成整个数组的排序
    quickSort(array, left, i - 1);//左边
    quickSort(array, i + 1, right); //右边
}
```

二分查找

```
static int binarySearch(int[] arr, int dest) {
    int begin = 0;
    int end = arr.length - 1;
    while (begin <= end) {
        int mid = (begin + end) / 2;
        if (arr[mid] > dest) {
            end = mid - 1;
        } else if (arr[mid] < dest) {
            begin = mid + 1;
        } else {
            return mid;
```

```
        }
    }
    return -1;
}
```

rsa加密算法（公钥解密私钥密文）

反转链表 归并排序

两个队列实现栈 两个栈实现队列 判断链表是否成环，找到成环的交点。

手写一道算法题，关于DFS+回溯算法的

虚拟机

垃圾回收机制

```
Java 堆:
新生代:Eden空间、From Survivor、To Survivor
老年代:tentired 区
对象都会首先在 Eden 区域分配，如果放不下，执行MinorGC，还是无法存入Survivor区，提前转移到老年代。
老年代如果存的下就不会执行Full GC。

大对象直接进入老年代：为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。
长期存活的对象将进入老年代：对象出生在Eden，被Minor GC后仍存活，并且能被 Survivor 容纳，将被移动到 Survivor 空间中，年龄+1，到了15岁，升级到老年代。
动态对象年龄判定：如果 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需达到要求的年龄。

判断对象是否死亡：引用计数，可达性分析
不可达对象不是一定要被回收：finalize
如何判断一个常量是废弃常量：没有被引用
如何判断一个类是无用的类:
- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

垃圾收集算法：复制算法，标记清除，标记整理
新生代-复制
老年代-标记XX

垃圾收集器:
serial：串行，单线程，工作时必须暂停其他线程，直到完成，新生代-复制
serial-old：同上，老年代-标记整理
parNew：多线程和serial差不多，新生代-复制
Parallel Scavenge：收集关注点是吞吐量(CPU中用于运行用户代码的时间与CPU总消耗时间的比值)
多线程，新生代-复制
Parallel Old：Parallel Scavenge的老年版，多线程，老年代-标记整理

CMS：
```

获取最短回收停顿时间为目标的收集器。注重用户体验。
初始标记：暂停所有线程，记录与root相连的对象，速度很快 ；
并发标记：同时启动GC和用户线程，记录这些发生引用更新的地方。
重新标记：修正并发标记期间变动对象的标记记录
并发清除：开启用户和GC线程对为标记的区域做清扫。

并发，老年代-标记清除
缺点：
1.内存碎片
2.需要更多cpu资源
3.浮动垃圾问题，需要更大的堆空间
（并发清理阶段用户线程还在运行，这段时间就可能产生新的垃圾，新的垃圾在此次GC无法清除，只能等到下次清理）

G1：
面向服务端应用的垃圾回收器
采用分区回收的思维，基本不牺牲吞吐量的前提下完成低停顿的内存回收；可预测的停顿是其最大的优势；
跨新生代和老年代；标记整理 + 化整为零
并行与并发收集器

注：吞吐量=运行用户代码时间/(运行用户代码时间+ 垃圾收集时间)
垃圾收集时间= 垃圾回收频率 * 单次垃圾回收时间

Android是CMS

谈谈对android虚拟机的理解。

Android应用程序运行在Dalvik/ART虚拟机，并且每一个应用程序对应有一个单独的Dalvik虚拟机实例。
Dalvik虚拟机执行的dex文件，JVM执行的是Class文件
DVM也是实现了JVM规范的虚拟机，默认使用CMS垃圾回收，但是与JVM运行的Class字节码不同，DVM执行的是Dex(压缩格式)，
Dex是很多个class文件处理压缩后的产物，最终在Android环境执行
Dalvik：从Android 2.2版本开始，支持JIT即时编译（Just In Time）在程序运行的过程中进行选择热点代码（经常执行的代码）进行编译或者优化。
ART（Android Runtime）是在 Android 4.4 中引入的一个开发者选项，也是 Android 5.0 及更高版本的默认 Android 运行时。ART虚拟机执行的是本地机器码。
ART 引入了预先编译机制（Ahead Of Time），在安装时，ART 使用设备自带的 dex2oat 工具来编译应用，dex中的字节码将被编译成本地机器码。

7.0：混合编译。应用在安装时不做编译，而是运行时解释字节码，同时在JIT编译了一些代码后将这些代码信息记录至Profile文件，等到设备空闲的时候使用AOT(All-Of-the-Time compilation:全时段编译)编译生成称为app_image的base.art(类对象映像)文件，这个art文件会在apk启动时自动加载（相当于缓存）。
9.0：加入了JVMTI，黑科技,里面有几个重要的方法，
RetransformClasses 插桩class和RedefineClasses 重新定义class，插桩和重新定义，可以通过这两个方法动态修改运行时的文件

dexopt与dexaot
dexopt
在Dalvik中虚拟机在加载一个dex文件时，对 dex 文件 进行 验证 和 优化的操作，其对 dex 文件的优化结果
变成了 odex(Optimized dex) 文件，这个文件和 dex 文件很像，只是使用了一些优化操作码。
dex2oat
ART 预先编译机制，在安装时对 dex 文件执行AOT 提前编译操作，编译为OAT（实际上是ELF文件）可执行文件（机器码）。

基于栈的虚拟机，有JVM

基于寄存器的，有Dalvik
无论这两种方式实现机制如何，都要实现以下几点：
- 取指令，其中指令来源于内存
- 译码，决定指令类型（执行何种操作）。另外译码的过程要包括从内存中取操作数
- 执行。指令译码后，被虚拟机执行（其实最终都会借助于物理机资源）
- 存储计算结果
栈式虚拟机 VS 寄存器式虚拟机
（1）**指令条数：栈式虚拟机多**
（2）**代码尺寸：栈式虚拟机小**
（3）**移植性：栈式虚拟机移植性更好**
（4）**指令优化：寄存器式虚拟机更能优化**

栈式 VS 寄存器式	对比
指令条数	栈式 > 寄存器式
代码尺寸	栈式 < 寄存器式
移植性	栈式优于寄存器式
指令优化	栈式更不易优化
解释器执行速度	栈式解释器速度稍慢
代码生成难度	栈式简单
简易实现中的数据移动次数	栈式移动次数多

jvm 堆/栈/方法区/本地方法栈/程序计数器 分别有什么作用

程序计数器：程序计数器是一块很小的内存空间，主要用来记录各个线程执行的字节码的地址。

虚拟机栈：存储线程运行方法所需的数据，指令、返回地址。
Java 虚拟机栈是基于线程的。
栈里的每条数据，就是栈帧。在每个 Java 方法被调用的时候，都会创建一个栈帧，并入栈。一旦完成相应的调用，则出栈。所有的栈帧都出栈后，线程也就结束了。
每个栈帧，都包含四个区域：(局部变量表、操作数栈、动态连接、返回地址)

局部变量表：存放基础类型，Object对象存放的是引用地址
操作数栈：先进后出的栈结构，入栈和出栈的操作
动态连接：多态
返回地址：调用程序计数器中的地址作为返回

本地方法栈：本地方法栈是和虚拟机栈非常相似的一个区域，它服务的对象是 native 方法。

方法区：存放已被虚拟机加载的类相关信息，包括类信息、静态变量、常量、运行时常量池、字符串常量池。

堆：堆是 JVM 上最大的内存区域，我们申请的几乎所有的对象，都是在这里存储的。我们常说的垃圾回收，操作的对象就是堆。

jvm线程私有的有哪些

线程独占的：栈、本地方法栈、程序计数器
共享：堆、方法区

new一个对象的过程发生了什么

虚拟机接收到new的指令会执行以下过程：
1.检查加载:检查类是否已经被加载

2.分配内存:为对象分配内存
(1)指针碰撞:用过内存和空闲内存中间通过指针分界，将指针挪动一段与对象大小相等的距离，适用于规整内存
(2)空闲列表:虚拟机维护的列表记录哪些内存是可用的，分配时找到足够空间换分给对象，并更新记录，适合不规整内存
选择方式是根据垃圾收回机制
并发安全问题解决:
(1)CAS机制
(2)本地线程分配缓冲(TLAB):每个线程在Java堆中预先分配一小块私有内存

3.内存空间初始化:将内存初始化为零，保证对象不赋值就能直接使用(int=0，boolean=false)
4.设置:设置对象是哪个类的实例(对象的年龄、哈希等)
5.对象初始化:执行构造方法,创建真正可用对象

对象内存布局

1.对象头
(1)自身的运行数据(MarkWord,占用8字节):哈希码、GC分代年龄、锁状态、线程持有的锁、偏向线程id、偏向时间戳
(2)类型指针(8字节，有压缩是4字节)
(3)记录数组长度(如果有)

2.实例数据
3.对齐填充:占位作用,8字节的整数倍，因为是块读取

网络

http2的功能有哪些?

http2功能:
二进制分帧层
以二进制传输代替原本的明文传输，原本的报文消息被划分为更小的数据帧.
多路复用
在一个 TCP 连接上，我们可以向对方不断发送帧，每帧的 stream identifier 的标明这一帧属于哪个流，然后在对方接收时，根据 stream identifier 拼接每个流的所有帧组成一整块数据。把 HTTP/1.1 每个请求都当作一个流，那么多个请求变成多个流，请求响应数据分成多个帧，不同流中的帧交错地发送给对方，这就是 HTTP/2 中的多路复用。流的概念实现了单连接上多请求 - 响应并行，解决了线头阻塞的问题，减少了 TCP 连接数量和 TCP 连接慢启动造成的问题.http2 对于同一域名只需要创建一个连接，而不是像 http/1.1 那样创建 6~8 个连接。
服务端推送
浏览器发送一个请求，服务器主动向浏览器推送与这个请求相关的资源，这样浏览器就不用发起后续请求。
Header 压缩
使用 HPACK 算法来压缩首部内容

tcp方面拥塞控制?

tcp拥塞控制:(对资源的需求>可用资源)
常见的拥塞控制有：
（1）慢开始，拥塞避免（2）快重传，快恢复

慢开始算法:一开始向网络发送一个很小的数据包，用于探测网络的拥塞程度，而后再动态的加倍增加，直到达到网络所能接受的程度而不造成网络拥塞；
拥塞避免算法:每次在发送数据包的时候，只增加很小的窗口容量，比如cwnd加1，而不是加倍的增加；
如果慢开始算法加倍后，出现了网络拥塞，那么就把之前加倍的去掉，然后再采用拥塞避免算法，逐渐的递增，来试探网络的拥塞窗口大小；
拥塞避免算法并不是可以百分百的避免网络拥塞，而是会降低出现网络拥塞出现的概率；
快重传算法:要求接收方在接收到一个失序的数据后，就立刻发出重复确认报文
快恢复算法:当进行了快重传算法时，发送方接收到了重复的确认报文，而这时候就认为网络还没有处于拥塞的情况，那么就将当前传输速度减半后，不走慢开始算法；
==========
cwnd：拥塞窗口变量
ssthresh(slow start threshold)：慢开始门限
swnd：发送方拥塞窗口

慢开始，一开始发送一个很小的数据包(探测网络的拥塞程度)，然后每次*2，
cwnd<ssthresh ：慢开始
cwnd>ssthresh ：停止慢开始，改用拥塞避免
cwnd=ssthresh ：可以慢开始，可以拥塞避免
拥塞避免按线性规律增长，
如果丢失数据了，发送方判断出现拥塞，就更改ssthresh=cwnd/2，cwnd=1，并重新开始执行慢开始算法。
有时，个别报文段数据丢失，但实际上并不是发送拥塞，此时导致发送方超时重传，就会将cwnd设置为1，因而降低效率。所以引入了快重传和快恢复。
假设报文丢失，接收方收到的是无序的报文，此时接收方会发送3次重复确认，发送方收到3次确认后，就会执行快速恢复算法，将ssthresh和cwnd调整为当前窗口的一半，开始执行拥塞避免。

RTT:往返时间
RTO:重传超时时间
MSS:最大报段长度

TCP的拥塞控制机制是什么？请简单说说。
答：我们知道TCP通过一个定时器（timer）采样了RTT并计算RTO，但是，如果网络上的延时突然增加，那么，TCP对这个事做出的应对只有重传数据，然而重传会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这就导致了恶性循环，最终形成“网络风暴” —— TCP的拥塞控制机制就是用于应对这种情况。

首先需要了解一个概念，为了在发送端调节所要发送的数据量，定义了一个“拥塞窗口”（Congestion Window），在发送数据时，将拥塞窗口的大小与接收端ack的窗口大小做比较，取较小者作为发送数据量的上限。
拥塞控制主要是四个算法：
1.慢启动：意思是刚刚加入网络的连接，一点一点地提速，不要一上来就把路占满。
连接建好的开始先初始化cwnd = 1，表明可以传一个MSS大小的数据。
每当收到一个ACK，cwnd++; 呈线性上升
每当过了一个RTT，cwnd = cwnd*2; 呈指数让升
阈值ssthresh（slow start threshold），是一个上限，当cwnd >= ssthresh时，就会进入“拥塞避免算法”
2.拥塞避免：当拥塞窗口 cwnd 达到一个阈值时，窗口大小不再呈指数上升，而是以线性上升，避免增长过快导致网络拥塞。
每当收到一个ACK，cwnd = cwnd + 1/cwnd
每当过了一个RTT，cwnd = cwnd + 1
拥塞发生：当发生丢包进行数据包重传时，表示网络已经拥塞。分两种情况进行处理：
等到RTO超时，重传数据包
sshthresh = cwnd /2
cwnd 重置为 1
3.进入慢启动过程
在收到3个duplicate ACK时就开启重传，而不用等到RTO超时
sshthresh = cwnd = cwnd /2
进入快速恢复算法——Fast Recovery
4.快速恢复：至少收到了3个Duplicated Acks，说明网络也不那么糟糕，可以快速恢复。
cwnd = sshthresh + 3 * MSS （3的意思是确认有3个数据包被收到了）
重传Duplicated ACKs指定的数据包
如果再收到 duplicated Acks，那么cwnd = cwnd +1
如果收到了新的Ack，那么，cwnd = sshthresh ，然后就进入了拥塞避免的算法了。

TLS的握手和具体的非对称加密算法。

1.Client Hello
客户端向服务端发起请求，向服务端提供：
支持的协议版本（如：TLS 1.2）
随机数Random_C，第一个随机数，后续生成“会话密钥”会用到
支持的加密方法列表
支持的压缩方法，等等

2.Server Hello
服务端向客户端发起响应，响应信息包含：
确认使用的加密通信协议版本（如：TLS 1.2）
随机数 Random_S，第二个随机数，后续生成“会话密钥”会用到
确认使用的加密算法，（如 RSA 公钥加密）
确认使用的压缩方法

3.Certificate + Server Key Exchange + Server Hello Done
Certificate: 返回服务器证书，该证书中含有一个公钥，用于身份验证和密钥协商
Server Key Exchange: 当服务器证书中信息不足，不能让 Client 完成 premaster 的密钥交换时，会发送该消息
Server Hello Done: 通知客户端 server_hello 信息发送结束。

4.Certificate Request
如果需要双向验证时，服务端会向客户端请求证书

5.客户端验证证书

客户端收到服务器证书后，进行验证，如果证书不是可信机构颁发的，或者域名不一致，或者证书已经过期，那么客户端会进行警告；如果证书没问题，那么继续进行通信。

6.Client Key Exchange + Change Cipher Spec + Encrypted Handshake Message
Client Key Exchange：证书验证通过后，客户端会生成整个握手过程中的第三个随机数，并且从证书中取出公钥，利用公钥以及双方实现商定的加密算法进行加密，生成Pre-master key，然后发送给服务器。

服务器收到 Pre-master key后，利用私钥解密出第三个随机数，此时，客户端和服务端同时拥有了三个随机数：Random_C, Random_S,Pre-master key,两端同时利用这三个随机数以及事先商定好的加密算法进行对称加密，生成最终的“会话密钥”，后续的通信都用该密钥进行加密。这一个过程中，由于第三个随机数是通过非对称加密进行加密的，因此不容易泄漏，也就“会话密钥”是安全的，后续的通信也就是安全的。

Change Cipher Spec：客户端通知服务端，随后的信息都是用商定好的加密算法和“会话密钥”加密发送。

Encrypted Handshake Message：客户端握手结束通知，这一项同时也是前面发送的所有内容的hash值，用来供服务器校验。

7.Certificate
客户端发送证书给服务器

8.Change Cipher Spec + Encrypted Handshake Message

Change Cipher Spec：服务端通知客户端，随后的信息都是用商定好的加密算法和“会话密钥”加密发送。

Encrypted Handshake Message：服务器握手结束通知，这一项同时也是前面发送的所有内容的hash值，用来供客户端校验。

9.Application Data
至此，整个握手过程就完成了，客户端和服务端进入加密通信。

https单向认证只有服务端有证书验证
双向认证，客户端和服务端都需要进行证书验证

具体的非对称加密算法。非对称名称:
RSA， Diffie-Hellman

秘钥使用非对称加密
传输使用对称加密

网络状态码

200:成功
206:请求的数据区间成功

301:永久重定向
302:临时重定向
304:文件未变化(ETAG - If-Modified-Since,Last-modifield - If-None-Match)

404:请求资源未找到
405:请求方法不对
414:请求参数过长
416:请求的数据区间不在文件范围之内

500:服务器无法完成对请求的处理

tcp 三次握手/四次挥手， 为什么要三次握手，四次挥手?

初始状态:双端处于 CLOSE 状态，服务端为提供服务会主动监听某个端口进入 LISTEN 状态。

第一次握手:客户端将标志位SYN置为1，随机产生序列号seq=J，发送给服务端，客户端进入SYN_SENT状态，等待服务端应答。
第二次握手:服务器端收到数据包，数据包里SYN标志位=1，代表客户端需要建立连接，服务端就要做出应答，服务器端将标志位SYN和ACK都置为1，ack=J+1，(通过这个ack告诉客户端前面发送的SYN我收到了) 并且，随机产生一个序列号seq=K，进入SYN_RCVD状态。
第三次握手:客户端收到确认，检查ack是否为J+1，ACK是否为1，此时客户端就知道服务器前面收到我发送的SYN报文，客户端应答服务端，将标志位ACK置为1，ack=K+1，客户端进入ESTABLISHED状态，服务器收到应答后，也进入ESTABLISHED状态。 可以开始通讯

ACK:通讯报文的标志位
ack:具体数值(上一个报文的应答)

为什么要3次握手:
TCP是可靠的传输控制协议，而三次握手是保证数据可靠传输又能提高传输效率的最小次数。
三次握手是通信双方相互告知序列号起始值，并确认对方已经收到了序列号起始值的必经步骤。
如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认。
至于为什么不是四次，很明显，三次握手后，通信的双方都已经知道了对方序列号起始值，也确认了对方知道自己序列号起始值，第四次握手已经毫无必要了。

TCP的三次握手的漏洞-SYN洪泛攻击
在第二次握手，服务端向客户端应答请求，应答请求是需要客户端IP的，攻击者伪造这个IP，往服务端狂发送第一次握手的内容，导致服务器端被拖累死机。
解决:
无效连接监视释放：不停监视所有的连接，这种方法对于所有的连接一视同仁，不管是正常的还是攻击的，所以这种方式不推荐。
延缓TCB分配方法：做完第一次握手，服务器为该请求延迟分配控制资源，当正常连接后在分配，有效地减轻服务器资源的消耗。
使用防火墙：防火墙在确认了连接的有效性后，才向内部的服务器（Listener）发起SYN请求

四次挥手：
客户端和服务端都可能调用close

(1)假设客户端主动发起close，客户端发送一个FIN报文和seq=j，告诉服务端我的数据写完了，客户端进入FIN-WAIT-1（终止等待1）状态。
(2)服务端接收到这个FIN报文，服务端发送ACK=1,seq=j+1，服务端进入了CLOSE-WAIT（关闭等待）状态，客户端收到确认报文后进入FIN-WAIT-2（终止等待2）状态。
(3)服务端会在发送一个FIN报文，seq=k，告诉客户端，我的数据写完了,服务器进入CLOSE状态
(4)客户端收到FIN报文后，进入TIME_WAITING状态，并且应答服务器ACK=1,seq=k+1，服务器进入CLOSED状态

为什么TCP的挥手需要四次?
TCP是全双工模式,客户端发出FIN报文，只是表示客户端没数据发送了，但是客户端还是能接收服务端的数据，服务端返回ACK报文，表示服务端已经知道客户端没数据发送了，但是服务端还能发送数据，当服务端发送FIN报文时，这个时候服务端也就没数据发送了，客户端收到FIN之后彼此就会中断TCP连接。
所以全双工为了关闭，就需要4次。

第二第三次挥手也可能合并，当没数据时候就会合并。

为什么需要TIME-WAIT状态?

1、可靠的终止TCP连接。(因为会有丢包所以需要)
2、保证让迟来的TCP报文有足够的时间被识别并丢弃。(相同的端口被同时开启多次，如果处于TIME_WAIT状态就无法使用该链接的端口来建立一个新连接)

TIME-WAIT（时间等待）状态:必须经过2*MSL的时间(1-4分钟)
MSL:最长报文段寿命,RFC建议2分钟，其他操作系统可能是30秒

https解释下

HTTPS是基于HTTP上扩展出的一种安全的传输协议。通过加密协议对通信进行加密，该协议称之为传输层安全性（TLS）
该协议通过协商密钥，以及验证身份的方式，来保证通信的双方数据不被获取到，以此来保证数据传输的安全性；
涉及到很多算法，比如哈希算法，对称加密算法，非对称加密算法，数字证书等相关知识

SSL的设计
非对称加密算法比较耗时，而对称加密算法则不会很耗时，但是对称加密算法的密钥不安全，那么我们可以结合这两种加密算法来进行安全传输；
1，使用对称加密来加密数据；
2，使用非对称加密来加密对称加密的密钥；

HTTPS的实现，等于HTTP+SSL，而SSL协议做的主要工作是，帮助通信的双方，建立起通信的安全通道，通过四次握手，进行身份验证，密钥协商等操作，让服务器与客户端通过对称加密算法进行数据的加密，然后通过非对称加密算法来进行数据的签名，保证数据的完整性，以此来达到安全传输的目的；

常见的对称加密算法和非对称加密算法有哪些?

对称加密算法:AES,DES
计算有哪些:
（1） 移位和循环移位
移位就是数码按照一定的顺序进行移动，左移或右移
（2）置换
（3）扩展
（4）压缩
（5）异或
（6）迭代

非对称加密算法:
RSA， Diffie-Hellman

谈谈TCP与UDP的理解。

TCP：可靠
3次握手。。。

UDP：不可靠
面向无连接：发送数据不需要连接
尽量交付：不保证可靠性
面向报文
没有拥塞控制
灵活性高
首部开销小

场景：实时游戏，直播

DNS域名解析流程

互联网上的域名系统是一个分布式的系统，结构上是一个四层的树状层次结构

查询www.baidu.com为例:
1.首先检查 DNS 缓存，如果缓存未命中，客户端需要向 local DNS 发送查询请求报文
2.客户端向 local DNS 发送查询报文 query www.baidu.com，local DNS 首先检查自身缓存，如果存在记录则直接返回结果，查询结束；如果缓存未命中
3.local DNS 向根域名服务器发送查询报文 query www.baidu.com，返回 .com 顶级域名服务器的地址（如果查无记录）
4.local DNS 向 baidu.com 权威域名服务器发送查询报文 query www.baidu.com，得到 ip 地址，存入自身缓存并返回给客户端

在浏览器输入url发生了什么

浏览器的地址栏输入URL并按下回车。
浏览器查找当前URL是否存在缓存，并比较缓存是否过期。
DNS解析URL对应的IP。
根据IP建立TCP连接（三次握手）。
HTTP发起请求。
服务器处理请求，浏览器接收HTTP响应。
渲染页面，构建DOM树。
关闭TCP连接（四次挥手）。

http和tcp区别

TCP是底层协议，定义的是数据传输和连接方式的规范。
HTTP是应用层协议，定义的是传输数据的内容的规范。
HTTP协议中的数据是利用TCP协议传输的，所以支持HTTP就一定支持TCP。

Https可以伪造证书吗

伪造:
CA签发错误的证书、CA为了利益发恶意证书、攻击者伪造某个域名拥有者申请证书

解决:
在头部加上证书透明度，浏览器会检查站点使用的证书是否出现在公共CT日志中。

线程

多线程同步Sync和Lock实现原理

同步代码块:
代码开始地方会增加Monitor Enter指令
方法结束和异常会增加Monitor Exit指令
当代码执行到该执行时，会获取Monitor对象的所有权
如果是0，该线程进入monitor，然后设置为1

如果线程已经占有monitor，只是重新进入，则进入monitor的进入数加1.
如果其他线程已经占用monitor，就进入阻塞状态，直到monitor进入数为0，在重新获取monitor所有权

同步方法：
常量池中多了ACC_SYNCHRONIZED标示符。
执行线程会先获取monitor对象，获取成功后才能执行方法体，执行结束后在释放monitor对象，执行期间其他线程无法获取同一个monitor对象

Lock实现原理
大多数lock通过AQS实现
AQS内部通过int变量来实现锁的同步机制

锁的可重入：
1.线程再次获取锁
判断当前锁变量=0，如果是0，直接获取锁
如果！=0，但是是当前线程获取锁，则 变量+1
2.释放锁
获取了多少次就释放多少次变量，直到为0

公平和非公平
通过构造方法的boolean变量，来判断创建公平还是非公平
非公平，只要设置同步状态成功，就表示获取了当前锁
公平，加入判断同步队列中当前节点是否有前驱节点，如果有，则表示有线程比当前线程更早的去请求获取锁，因此需要等待前驱线程获取并释放锁之后，在去获取。

多线程加锁的几种方法

synchronized
Lock
ReentrantLock 可重入锁
1. synchronized 内置的Java关键字，lock是一个Java类
2. synchronized无法判断获取锁的状态，lock可以判断是否获取到了锁
3. synchronized会自动释放锁，lock必须要手动释放锁！如果不释放锁——死锁
4. synchronized 线程1（获得锁，阻塞）、线程2（等待，傻傻的等待），lock锁就不一定会等待下去
5. synchronized 可重入锁，不可以中断的，非公平；lock 可重入锁，可以判断锁，非公平（可以自己设置为公平锁，非公平是默认状态）
6. synchronized 适合锁少量的代码同步问题，Lock适合锁大量的同步代码

ReentrantReadWriteLock 读写锁
1.Java并发库中ReetrantReadWriteLock实现了ReadWriteLock接口并添加了可重入的特性
2.ReetrantReadWriteLock读写锁的效率明显高于synchronized关键字
3.ReetrantReadWriteLock读写锁的实现中，读锁使用共享模式；写锁使用独占模式，换句话说，读锁可以在没有写锁的时候被多个线程同时持有，写锁是独占的
4.ReetrantReadWriteLock读写锁的实现中，需要注意的，当有读锁时，写锁就不能获得；而当有写锁时，除了获得写锁的这个线程可以获得读锁外，其他线程不能获得读锁

volatile 只能保证可见性(能看到任意线程对这个变量的写入)，不能保证原子性(复合操作)
Lock会对CPU总线和高速缓存加锁
当前处理器缓存行的数据直接写回到系统内存中，且这个写回内存的操作会使在其他CPU里缓存了该地址的数据无效。

多线程线程安全问题是如何发生的

多个线程同时去修改某个文件

synchronized锁方法和锁静态方法有什么区别

锁普通方法，锁对象是this
锁静态方法，锁的对象是class

volatile关键字解释下？volatile 关键字不能实现什么？

volatile可以保证线程的可见性
volatile 还可以禁止指令重排序，从而保证有序性
volatile 不能保证原子性，所以不能保证线程安全。
如果修饰的变量操作都是原子性的，没有复合操作，那么也可以保证线程的安全。

可见性：
当一个线程更新了内存中的共享变量时，需要通知其他线程**重新从内存中读取值**。
原理:
有 volatile 变量修饰的共享变量进行写操作的时候会多出lock 前缀的指令，在多核处理器下会引发了两件事情。
- 将当前处理器缓存行的数据会写回到系统内存。
- 这个写回内存的操作会引起在其他 CPU 里缓存了该内存地址的数据无效。

有序性：
JVM会考虑性能效率问题然后对指令进行**重排序**
1.编译器优化的重排序
2.指令级并行的重排序
3.内存系统的重排序

原子性：
1.读取：把变量从主内存传输到线程的工作内存中
2.载入：读取到的变量放入工作内存的变量副本中
3.使用：工作内存变量传递到执行引擎
4.赋值：从执行引擎接收到的值赋值给工作内存的变量
5.存储：把工作内存中的一个变量的值传送到主内存中
6.写入：从工作内存中得到的变量的值放入到主内存的变量中。

常用于DCL失效问题：
1. 分配对象的内存空间
2. 初始化对象
3. 指向刚分配的内存空间地址

线程1执行new，分配内存空间，然后指向刚分配的内存空间地址，此时线程2调用getInstance，所以会直接返回一个**不为 null 但是未完成初始化的 instance 对象**。

如何做到禁止指令重排序：
通过内存屏障（就是CPU的指令）
- Load：将内存存储的数据拷贝到处理器的高速缓存中。
- Store：将处理器高速缓存中的数据刷新到内存中。

LoadLoad屏障：
抽象场景：Load1; LoadLoad; Load2

Load1 和 Load2 代表两条读取指令。在Load2要读取的数据被访问前，保证Load1要读取的数据被读取完毕。

StoreStore屏障：
抽象场景：Store1; StoreStore; Store2
Store1 和 Store2代表两条写入指令。在Store2写入执行前，保证Store1的写入操作对其它处理器可见

LoadStore屏障：
抽象场景：Load1; LoadStore; Store2
在Store2被写入前，保证Load1要读取的数据被读取完毕。

StoreLoad屏障：
抽象场景：Store1; StoreLoad; Load2
在Load2读取操作执行前，保证Store1的写入对所有处理器可见。StoreLoad屏障的开销是四种屏障中最大的。

在一个变量被volatile修饰后，JVM会为我们做两件事：
1.在每个volatile写操作前插入**StoreStore**屏障，在写操作后插入**StoreLoad**屏障。
2.在每个volatile读操作前插入**LoadLoad**屏障，在读操作后插入**LoadStore**屏障。

如何做到可见性：
当共享变量被 volatile 修饰后，在多线程环境下，当一个线程对它进行修改值后，会**立即写入到内存中**，然后让其他所有持有该共享变量的线程的**工作内存中的值过期**，这样其他线程就必须**去内存中重新获取最新的值**，从而做到共享变量及时可见性。

线程池使用，关键参数的取值依据和使用

核心线程数
最大线程数
阻塞队列
拒绝策略
空闲存活时间:线程数大于corePoolSize时才有用
空闲单位
线程工厂

执行任务-判断核心线程数，大于核心线程数，放入到阻塞队列等待执行，阻塞队列满了，判断最大线程数，最大线程数满了，执行决绝策略。

FixedThreadPool：固定线程数的线程池（适合有一定异步任务，周期较长的场景，能达到有效的并发状态）
```
new ThreadPoolExecutor(nThreads, nThreads,0L, TimeUnit.MILLISECONDS,new
LinkedBlockingQueue<Runnable>());
```

CachedThreadPool：缓存线程池，okhttp里使用的是这个（适合异步任务多，但周期短的场景）
```
new ThreadPoolExecutor(0, Integer.MAX_VALUE,60L, TimeUnit.SECONDS,new
SynchronousQueue<Runnable>());
```

ScheduledThreadPool：实现定时周期性任务的线程池（适合周期性执行任务的场景）
```
super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,new DelayedWorkQueue());
```

SingleThreadExecutor：单线程线程池（适合任务串行的场景）
```
new ThreadPoolExecutor(1, 1,0L, TimeUnit.MILLISECONDS,new
LinkedBlockingQueue<Runnable>()
```

不同的任务类别应采用不同规模的线程池，任务类别可划分为CPU密集型任务、IO密集型任务和混合型任务。(N代表CPU个数)

任务类别	说明
CPU密集型任务	线程池中线程个数应尽量少，如配置N+1个线程的线程池。
IO密集型任务	由于IO操作速度远低于CPU速度，那么在运行这类任务时，CPU绝大多数时间处于空闲状态，那么线程池可以配置尽量多些的线程，以提高CPU利用率，如2*N。
混合型任务	可以拆分为CPU密集型任务和IO密集型任务，当这两类任务执行时间相差无几时，通过拆分再执行的吞吐率高于串行执行的吞吐率，但若这两类任务执行时间有数据级的差距，那么没有拆分的意义。

线程同步的方案，常用的锁

悲观锁 synchronized，Lock:
悲观锁认为自己在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改

乐观锁 CAS（AtomicInteger。。。）:
乐观锁认为自己在使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据成功写入。如果数据已经被其他线程更新，则根据不同的实现方式执行不同的操作（例如报错或者自动重试）。

自旋锁 CAS(getAndAddInt)
当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销。这就是自旋锁。
如果自旋超过了限定次数没有成功获得锁，就应当挂起线程。

适应性自旋锁
自旋的时间（次数）不再固定，由前一次同一个锁上的自旋时间及锁的拥有者的状态来决定。
如果前一个锁，自旋等待刚刚成功获得过锁，并且正在运行，那么自旋等待持续相对更长的时间。
如果自旋很少成功获得过，可能以后获取这个锁直接省略自旋，直接阻塞线程，避免浪费处理器资源。

无锁 CAS
无锁没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。

偏向锁
偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁，降低获取锁的代价。
当线程访问同步代码块并获取锁时，会在Mark Word里存储锁偏向的线程ID。在线程进入和退出同步块时检测Mark Word里是否存储着指向当前线程的偏向锁。
引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令即可。

轻量级锁
是指当锁是偏向锁的时候，被另外的线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，从而提高性能。

重量级锁
若当前只有一个等待线程，则该线程通过自旋进行等待。但是当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁升级为重量级锁。此时等待锁的线程都会进入阻塞状态。

公平锁 ReentrantLock(true)
公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁。
限制条件：hasQueuedPredecessors()：主要是判断当前线程是否位于同步队列中的第一个。如果是则返回true，否则返回false。

非公平锁 ReentrantLock(false)
多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。

可重入锁 ReentrantLock和synchronized
同一个线程，若已持有这锁，就不需要重新申请锁。
避免死锁
通过AQS的status判断

不可重入锁 NonReentrantLock

共享锁（ReentrantReadWriteLock-ReadLock）
指该锁可被多个线程所持有。

排他锁(独享锁) ReentrantLock和synchronized ，写锁（ReentrantReadWriteLock-WriteLock）
指该锁一次只能被一个线程所持有

Java 中有哪些原子性操作

AtomicInteger、AtomicLong、AtomicBoolean、AtomicReference<T>

常用的多线程工具类。blockingqueue，concurrenthashmap，信号量，countdownlatch，cyclicbarrier，exchanger等，stringbuffer

blockingqueue 阻塞队列
concurrenthashmap 并发hashmap

countdownlatch：使一个线程等待其他线程各自执行完毕后再执行。
```
private static CountDownLatch latch = new CountDownLatch(3);
latch.countDown();
latch.await();
```

cyclicbarrier:类似一个闸门，指定数目的线程都必须到达这个闸门，闸门才会打开。
exchanger：在并发任务之间交换数据
stringbuffer：安全字符串拼接

lock和syncrognized适合什么场景。

多线程读写用lock

wait和sleep的区别

sleep是线程的方法，wait是Object的方法
sleep不会释放锁
wait会释放锁 --notify/notifyAll唤醒

它们都可以被interrupted方法中断。

三个线程wait，唤醒情况是什么样的

https://www.jianshu.com/p/3bba64487922

结果显示唤醒顺序与执行顺序是一样的（注意这里是强调唤醒顺序，而不是重新获得锁的顺序）。
wait后的线程确实是保存在一个FIFO的等待队列中。
为什么楼上的结果唤醒顺序与休眠顺序不一致？
因为他打印的不是唤醒顺序，而是唤醒后线程重新获得锁的顺序。
在循环中直接notify的话，这些等待的线程虽然是按顺序唤醒，但是间隔时间非常短，几乎会同时去竞争锁（这一点和在循环中start线程却不能保证线程执行的顺序是一个道理），因此在竞争下是无法保证获取锁的顺序的。
总结：wait后的线程会进入一个FIFO的队列中，notify是一个有序的出队列的过程。而短时间内多个线程竞争获得锁的顺序则是不确定的，这要靠cpu调度决定。

hotspot对notofy()的实现并不是我们以为的随机唤醒，而是“先进先出”的顺序唤醒！

唤醒的停顿时间写在sync里：
当我们执行notify之后,由于sleep在symchronized内部，因此没有释放锁！
由于synchronized实际上不是公平锁，其锁竞争的机制具有随机性,最终，我们看到的结果好像是随机的！

唤醒的停顿时间写在sync外：
lock锁立即被释放了,确保被唤醒的线程能够获得lock锁立刻执行,我们看到的结果才是正确的.

notify过程调用的是DequeueWaiter方法,实际上是将_WaitSet中的第一个元素进行出队操作,这也说明了notify是个顺序操作，具有公平性.

多线程使用遇到的问题

https://juejin.cn/post/6844903880824881160#heading-1

避免内存泄漏，死锁，

Java基础

java的hashcode和equals的区别

equals没有重写比较的是hashcode
重写一般情况下比较的是对象的各个字段。

hashcode是对象的独特值？
在hashmap中用于找到数组的槽位

- 两个对象相等，则hashcode一定也是相同的
- 两个对象相等,对两个对象分别调用equals方法都返回true
- 两个对象有相同的hashcode值，它们也不一定是相等的
- 因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖
- hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

如何重写hashCode
1. - 把某个**非零**的常数值，比如17，保存在一个名为result的int类型的变量中。
   - 对于对象中每个关键域f(**指equals方法中涉及的每个域**)，完成以下步骤：
   - - 如果f是boolean，则计算 **f?1:0**
     - 如果是byte，char，short或int，则计算 **(int)f**
     - 如果是long，则**计算(int)(f^(f>>>32))**
     - 如果是float，则**Float.floatToIntBits(s)**
     - 如果是double，则计算**Double.doubleToLongBits(f)**,再按long类型计算一遍
     - 如果是f是个对象引用，并且该类的equals方法通过递归地调用equals的方式来比较这个域，则同样为这个域递归调用hashCode。如果需要更复杂的比较，则为这个域计算一个‘范式’，然后针对这个范式调用hashCode。如果这个域的值为null，则返回0(或者其他某个常数，但通常是0)。
     - 如果是个数组，则需要把每个元素当做单独的域来处理。也就是说，递归地应用上述规则，对每个重要的元素计算一个散列码，然后根据步骤b中的做法把这些散列值组合起来。 如果数组域中的每个元素都很重要，可以利用发行版本1.5中增加的其中一个`Arrays.hashCode`方法。
     - **步骤(b)** 按照下面公式，把(a)步骤中计算得到的散列码c合并到result中：`result = 31*result+c` (为什么是31呢？)

   - - **步骤(a)** 为该域计算`int`类型的散列码c：

   - 返回result

   - 测试，是否符合『相等的实例是否都具有相等的散列码』

```
@Override
public int hashCode() {
    int result = 17;//非0 任选
    result = 31*result + name.hashCode();
    result = 31*result + className.hashCode();
    return result;
}
```

为什么要选31？
因为它是个**奇素数**，另外它还有个很好的特性，即用移位和减法来代替乘法，可以得到更好的性能：
31*i == (i<<5)-i

反射

Class.forName加载类时将类进了初始化
ClassLoader的loadClass并没有对类进行初始化，只是把类加载到了虚拟机中。

Java 反射效率低主要原因是：
1. Method#invoke 方法会对参数做封装和解封操作
我们可以看到，invoke 方法的参数是 Object[] 类型，也就是说，如果方法参数是简单类型的话，需要在此转化成 Object 类型，例如 long ,在 javac compile 的时候 用了Long.valueOf() 转型，也就大量了生成了Long 的 Object, 同时 传入的参数是Object[]数值,那还需要额外封装object数组。
而在上面 MethodAccessorGenerator#emitInvoke 方法里我们看到，生成的字节码时，会把参数数组拆解开来，把参数恢复到没有被 Object[] 包装前的样子，同时还要对参数做校验，这里就涉及到了解封操作。
因此，在反射调用的时候，因为封装和解封，产生了额外的不必要的内存浪费，当调用次数达到一定量的时候，还会导致 GC。

2. 需要检查方法可见性
通过上面的源码分析，我们会发现，反射时每次调用都必须检查方法的可见性（在 Method.invoke 里）

3. 需要校验参数
反射时也必须检查每个实际参数与形式参数的类型匹配性（在NativeMethodAccessorImpl.invoke0 里或者生成的 Java 版 MethodAccessor.invoke 里）；

4. 反射方法难以内联
Method#invoke 就像是个独木桥一样，各处的反射调用都要挤过去，在调用点上收集到的类型信息就会很乱，影响内联程序的判断，使得 Method.invoke() 自身难以被内联到调用方。
参见
https://www.iteye.com/blog/rednaxelafx-548536

5. JIT 无法优化
因为反射涉及到动态加载的类型，所以无法进行优化。

Error和Exception的区别

Exception：表示程序可能需要捕获并且处理的异常。
Error：表示当触发 Error 时，它的执行状态已经无法恢复了，需要中止线程甚至是中止虚拟机。这是不应该被我们应用程序所捕获的异常。

RuntimeException:它是在程序运行时，动态出现的一些异常(NullPointerException、ArrayIndexOutOfBoundsException)

Error 和 RuntimeException 都属于非检查异常
普通 Exception属于检查异常

所有检查异常都需要在程序中，用代码显式捕获，或者在方法中用 throw 关键字显式标注。

重写和重载的区别，应用场景

关键字 Override

子类重写父类的方法，原有基础上对功能扩展

重载：
构造重载，方法重载，
调用请求方法，默认加载框是有的， 可以减少方法的参数调用

AOP与OOP的区别

oop
继承 多态 封装 ，6大原则

aop:通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。
利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。
通过ASM字节码来实现

OOP与AOP的区别：
1、面向目标不同：简单来说OOP是面向名词领域，AOP面向动词领域。
2、思想结构不同：OOP是纵向结构，AOP是横向结构。
3、注重方面不同：OOP注重业务逻辑单元的划分，AOP偏重业务处理过程的某个步骤或阶段。

java的四种引用分析

强->new
软->SoftReference 内存不足回收 图片缓存
弱->WeakReference 只要垃圾回收器回收，不一定要等到虚拟机内存不足时才强制回收 防止内存泄漏
虚->PhantomReference 随时可能被回收

Android基础

Activity 和 Fragment 生命周期有哪些?

```
onCreate
onStart
onResume
onRestart
onPause
onStop
onDestroy
==================
fragment
onAttach
onCreate
```

```
onCreateView
onActivityCreate
onStart
onResume
onPause
onStop
onDestroyView
onDestroy
onDetach
```

横竖屏切换时候 Activity 的生命周期?

```
页面旋转:
无配置:
横竖旋转
onPause-onStop-onDestroy-onCreate-onStart-onResume

配置:android:configChanges="orientation"
说明:orientation配置会执行onConfigurationChanged
旋转为横屏:
onConfigurationChanged-onPause-onStop-onDestroy-onCreate-onStart-onResume
旋转为竖屏:
onConfigurationChanged

配置:android:configChanges="orientation|screenSize"
说明:screenSize不会导致activity重启
旋转:onConfigurationChanged
```

onSaveInstanceState() 与 onRestoreIntanceState()?

```
onSaveInstanceState->保存数据
在异常终止情况下，会调用在onStop之前，和onpause没有时序的关系，可能前可能后


onRestoreIntanceState->恢复数据
activity被重新创建之后，会将onSaveInstanceState保存的bundle，传递给onCreate和
onRestoreIntanceState，调用时序在onStart之后


1.因为onSaveInstanceState 不一定会被调用，所以onCreate()里的Bundle参数可能为空，如果使用
onCreate()来恢复数据，一定要做非空判断。
2.而onRestoreInstanceState的Bundle参数一定不会是空值，因为它只有在上次activity被回收了才
会调用。
3.而且onRestoreInstanceState是在onStart()之后被调用的。有时候我们需要onCreate()中做的一
些初始化完成之后再恢复数据，用onRestoreInstanceState会比较方便。
```

android 中进程的优先级?

前台进程
可见进程
服务进程
后台进程
空进程

bundle 传递对象为什么需要序列化？Serialzable 和 Parcelable 的区别?

因为 bundle 传递数据时只支持基本数据类型，所以在传递对象时需要序列化转
换成可存储或可传输的本质状态（字节流）。
Serializable
Serializable 是序列化的意思，表示将一个对象转换成存储或可传输的状态。序列化后的对象可以在网络上进传输，也可以存储到本地。
Parcelable
除了 Serializable 之外，使用 Parcelable 也可以实现相同的效果，不过不同于将对象进行序列化，Parcelable 方式的实现原理是将一个完整的对象进行分解，而分解后的每一部分都是 Intent 所支持的数据类型，这也就实现传递对象的功能了。

Android 各版本新特性?

5.0
MaterialDesign风格
支持ART虚拟机

6.0
动态权限管理
Doze低电耗模式和StandBy待机模式

7.0
多窗口
V2签名

8.0
优化通知
画中画模式
WorkManager

9.0
安全增强
刘海屏
v3签名

10.0
夜间模式

android 中有哪几种解析 xml 的类,官方推荐哪种？以及它们的原理和区别?

DOM：文档驱动，解析xml前，需将整个xml存储到内存中
优：操作整个xml效率高，可随时多次访问解析过的文档
缺：耗内存
场景:xml小，频繁操作，多次访问

SAX,PULL：事件驱动，根据不同需求执行解析操作(不需要将整个xml存到内存)
SAX：
优：性能够，占存少
缺：解析方法复杂，拓展性差（无法修改xml结构），不能主动控制处理事件结束
场景：大型XML、解析性能要求高，不需要多次访问和修改

PULL：Android自带的，
优：性能够，占存少，使用比SAX简单，能主动控制事件结束
缺：拓展性差（无法修改xml结构）
场景：大型XML、解析性能要求高，不需要多次访问和修改

Jar 和 Aar 的区别

jar 内部只有class
arr内部除了有class，还有图片资源文件

Android 为每个应用程序分配的内存大小是多少

android 程序内存一般限制在 16M，也有的是 24M。近几年手机发展较快，一般都会分配两百兆左右，和具体机型有关

apk打包过程

1. 使用aapt工具处理所有的资源，生成一个R.java文件，一个resources.arsc文件以及其他资源。
2. aidl工具处理.aidl文件，生成对应的Java接口文件。
3. 将上述两步得到的R.java文件、Java接口文件，与Andorid源码一起，通过Java编译器，编译得到Java字节码文件.class文件。
4. 获取依赖的第三方库文件，将其与上一步得到的.class文件一起，通过使用dx工具，生成.dex文件。
5. 将资源索引文件resources.arsc、资源目录res、与上一步得到的.dex文件一起，通过apkbuilder工具，构建出初始的.apk文件。
6. 使用jarsigner工具，对.apk文件进行签名。
7. 使用zipalign工具，对.apk文件进行对齐。（让资源按四字节的边界进行对齐，加快资源的访问速度）

aar中是否含有R文件

是,资源命名最好统统加上你的项目名字前缀，防止跟宿主app下的资源重复

v1 v2 v3签名有什么区别

v1:
MANIFEST.MF
APK中的所有条目,SHA256消息摘要算法提取出该文件的摘要然后进行 BASE64 编码后，
作为「SHA1-Digest」属性的值写入到 MANIFEST.MF 文件中的一个块中。
该块有一个「Name」属性， 其值就是该文件在 APK 包中的路径。
CERT.SF
对 MANIFEST.MF 的各个条目做 SHA1（或者 SHA256）后再用 Base64 编码
CERT.RSA
之前生成的 CERT.SF 文件，用私钥计算出签名，然后将签名以及包含公钥信息的数字证书一同写入 CERT.RSA 中保存。

v1签名不校验META-INF，而v2签名将META-INF列入保护区，
这种方式打出来的渠道包是过不了安装时候的签名验证的。

v2:
v2签名，首先整个APK（ZIP文件格式）会被分为以下四个区块
1. Contents of ZIP entries（from offset 0 until the start of APK Signing Block）：ZIP 条目的内容
2. APK Signing Block ：APK 签名分块
3. ZIP Central Directory：ZIP 中央目录
4. ZIP End of Central Directory ：ZIP 中央目录结尾

应用的签名信息会被保存在区块2（APK Signing Block）中，其它3个区块是受保护的

区块2中APK Signing Block是由这几部分组成：

- 2个用来标示这个区块长度的8字节
- 这个区块的魔数（APK Sig Block 42）
- 这个区块所承载的数据（ID-value 组）

V2的签名信息是以ID-value格式保存在APK Signing Block这个区块的ID-value 组中，
所以美团开源的“瓦力”，原理就是将渠道信息以id-value的格式，写到区块2（APK Signing Block）的ID-value 组中
1. 对新的应用签名方案生成的APK包中的ID-value进行扩展，提供自定义ID—value（渠道信息），并保存在APK中
2. 而APK在安装过程中进行的签名校验，是忽略我们添加的这个ID-value的，这样就能正常安装了
3. 在App运行阶段，可以通过ZIP的`EOCD（End of central directory）`、`Central directory`等结构中的信息（会涉及ZIP格式的相关知识，这里不做展开描述）找到我们自己添加的ID-value，从而实现获取渠道信息的功能

v3:
新版v3签名在v2的基础上，仍然采用检查整个压缩包的校验方式。不同的是在签名部分增可以添加新的证书，即可以不用修改ApplicationID来完成证书的更新迭代。
签名机制主要有两种用途：
- 使用特殊的key签名可以获取到一些不同的权限
- 验证数据保证不被篡改，防止应用被恶意的第三方覆盖

v3版本签名验证证书步骤：（前三步同v2）
- 利用PublicKey解密Signature，得到SignerData的hash明文
- 计算SignerData的hash值
- 两个值进行比较，如果相同则认为APK没有被修改过，解析出SignerData中的证书。否则安装失败
- 逐个解析出level块证书并验证，并保存为这个应用的历史证书
- 如果是第一次安装，直接将证书与历史证书一并保存在应用信息中
- 如果是更新安装，验证之前的证书与历史证书，是否与本次解析的证书或者历史证书中存在相同的证书，其中任意一个证书符合即可安装

图片资源放在不同的文件夹中，加载出来的内存占用分别是多少，为什么会这样?

- ALPHA_8 -- (1B)
- RGB_565 -- (2B)
- ARGB_4444 -- (2B)
- ARGB_8888 -- (4B)
- RGBA_F16 -- (8B)

- ldpi-120
- mdpi-160
- hdpii-240
- xhdpi-320
- xxhdpi-480

Bitmap.decodeResource() 会根据图片存放的不同目录做一次分辨率的转换，而转换的规则是：
新图的高度 = 原图高度 * (设备的 dpi / 目录对应的 dpi)
新图的宽度 = 原图宽度 * (设备的 dpi / 目录对应的 dpi)

图片的分辨率是 1080*452，电脑 png 图片大小仅有 55.8KB
设备240，drawable 没带后缀对应 160 dpi：
转换后的分辨率：1080 * （240/160） * 452 * （240/160） = 1620 * 678
1620 * 678 * 4B = 4393440B ≈ 4.19MB

如果使用 fresco，那么不管图片来源是哪里，分辨率都是以原图的分辨率进行计算的

Glide 的处理与 fresco 又有很大的不同：
如果只获取 bitmap 对象，那么图片占据的内存大小就是按原图的分辨率进行计算。但如果有通过 into(imageView) 将图片加载到某个控件上，那么分辨率会按照控件的大小进行压缩。

类加载的双亲委派机制

首先会从缓存中找，如果没找到会从parent去找，都没有在去查找自己
双亲委派：1、安全 2、性能

Activity启动模式

Stander：后进先出
SingleTop：栈顶复用，栈顶有了就不会重新创建
SingleTask：栈内复用，并清除栈顶的其他activity
SingleInstance：全局复用，创建一个新的任务栈，容纳activity，后续请求不会重新创建，除非任务栈销毁

ClassNotFound的有可能的原因是什么

```
类没找到：
1.隐士跳转类没找到
2.APT生成的类写错
3.字节码插桩，插入的字节码不合法
```

webview中与js通信的手段有哪些

```
JsBridge
addJavaScriptInterface
loadUrl
```

Android中进程通信方式除了aidl，还有什么用的比较多（不包括Linux的）

```
使用Bundle
使用文件共享
Messenger
AIDL
ContentProvider
Socket
```

怎样获取当前线程是否是主线程

```
 Looper.myLooper() != Looper.getMainLooper()
 Thread.currentThread() == Looper.getMainLooper().getThread()
```

webview如何做资源缓存

https://www.jianshu.com/p/5e7075f4875f

```
浏览器 缓存机制
Cache-Control、Expires、Last-Modified & Etag四个字段
场景：静态资源文件的存储，如JS、CSS、字体、图片等。

Application Cache 缓存机制
manifest 文件
场景：存储静态文件（如JS、CSS、字体文件）
settings.setAppCachePath(cacheDirPath);

Dom Storage 缓存机制
Key - Value 对来提供
sessionStorage：具备临时性
localStorage：具备持久性
场景：存储临时、简单的数据
settings.setDomStorageEnabled(true);

Web SQL Database 缓存机制（不再维护）
settings.setDatabasePath(cacheDirPath);
```

```
settings.setDatabaseEnabled(true);

Indexed Database 缓存机制
场景:存储 复杂、数据量大的结构化数据
settings.setJavaScriptEnabled(true);

File System 缓存机制(暂时不支持)

缓存模式说明:
LOAD_CACHE_ONLY: 不使用网络，只读取本地缓存数据
LOAD_NO_CACHE: 不使用缓存，只从网络获取数据.
LOAD_DEFAULT: （默认）根据cache-control决定是否从网络上取数据。
LOAD_CACHE_ELSE_NETWORK，只要本地有，无论是否过期，或者no-cache，都使用缓存中的数据。
```

应用crash的本质原因是什么

```
非受检异常发生时，没有try…catch处理该异常对象，产生crash

NoClassDefFoundError错误的发生，是因为Java虚拟机在编译时能找到合适的类，而在运行时不能找到合适的类导致的错误。
例：JAR包冲突

OOM
```

js和java之间传递大对象会怎么样，怎么保证传输性能

```
压缩数据量，比如开启gzip，json进行格式化，减少空格等数据，或者可以考虑更换数据类型，json和xml都比较消耗内存
```

框架

AsyncTask

```
原理:
初始化有2个线程池变量
serialExecutor：将任务串行的线程池
ThreadPoolExecutor：执行任务的线程池
构造方法:
创建FutureTask和WorkRunnable，FutureTask线程执行完成后又返回值

执行
execute将任务放到串行线程池中的双端队列里,然后取出双端队列里的任务，交给执行线程池去执行，执行完成后接着取出任务，直到无任务。
执行publishProgress方法通过handler发送给主线程进度信息

处理
执行结束调用postResult通过handler将结果发送主线程调用onPostExecute
```

缺点
引发内存泄漏
当Activity被销毁时，还在运行的线程任务是不会被暂停的，这也就是造成了内存泄漏的第二个原因了。
调用cancel

由于是串行的，所以不适用于处理长时间的异步任务

RxJava

订阅过程

Observable.create(ObservableCreate).subscribe(Observer)
所有的被观察者的订阅入口都是subscribeActual方法，subscribeActual会调用观察者的抽象方法，最终调用到实现类。
通过Observable.create创建的被观察者是ObservableCreate，它是Observable的子类，
subscribeActual：内部会创建CreateEmitter发射器，将下游传给发射器，发射器实现了Disposable释放器。
将发射器传给下游的onSubscribe订阅回调，然后给上游添加订阅，并把下游传给上游。

发送数据

订阅完成，就可以通过发射器将数据发送给下游。
发射器持有这下游。调用下游的onNext方法，发送数据。

RXJava： 怎么实现线程切换的

```
subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())

subscribeOn:
创建发射器，回调onSubscribe，将上游的订阅封装到runnable，交给IoScheduler内的Work调用，
work内部创建了线程池，并执行runnable

observeOn:
onNext里，将任务放进队列，通过调用HandlerScheduler.schedule,Handler来切换线程，最终在取出队列里的任务去执行
```

Observer处理完onComplete后会还能onNext吗?

不能,onComplete会调用dispose方法。
CreateEmitter是AtomicReference<Disposable>原子类的实现，
dispose方法会将原子类添加了常量的DISPOSED，
onNext方法中会去判断isDisposed,如果==DISPOSED常量,返回true，不执行下去。

map 和 flatmap 操作符的区别

map:一对一
map操作符生成mapObservable的被观察者和observer订阅。map需要Function接口，第一泛型是入参，第二个是出参。通过apply返回。

flatMap:一对多
flatMap返回flatMapObservable来订阅observer，第二个返回参数是ObservableSource，Observable父类是ObservableSource，所以能返回Observable。

concatMap是有序的，flatMap是无序的

rxjava的背压了解吗

背压策略的具体实现：Flowable
决定观察者能够接收多少个事件:通过Subscription.request()
控制被观察者发送事件的速度:FlowableEmitter.requested()获取当前观察者需要接收的事件数量

异步情况会有缓存队列(默认大小128)
同步是没有缓存的，接收1个处理1个

观察者接收事件>=96,被观察内部调用request(96)

策略模式
ERROR-异常
MISSING-友好提示：缓存区满了
BUFFER-缓存区大小设置成无限大
DROP-超过缓存区大小（128）的事件丢弃
LATEST-只保存最新事件

OkHttp

原理

OkHttp是网络请求框架

OkHttpClient:可以配置自定义拦截器、超时时间等。
Request:具体请求
Call实现类是RealCall:调用execute和enqueue提交请求

Call是把Request交给OkHttpClient后返回的一个准备好的请求。

请求流程：
1.提交请求任务
2.通过拦截器做核心工作
3.执行完成调用finish

1.提交请求任务会通过分发器来调配任务的。
execute同步:加入到同步队列
enqueue异步:
正在执行任务未超过最大限制64，并且同一Host的请求不超过5个，添加到正在执行队列，同时提交给线程池，否则先加入等待队列

3.每次执行完都会调用finish：
移除执行完成的任务
异步任务执行结束会重新调配请求，将等待队列移动到执行队列执行。

分发器线程池：和Executors.newCachedThreadPool()创建的线程一样，望获得最大并发量。

2.拦截器：
通过责任链模式。
1. RetryAndFollowUpInterceptor
重试与重定向(重试次数为20次)
协议异常/不是超时异常/证书问题/->不能重试
DNS解析为多个IP会尝试另一个IP进行重试
3XX->重定向

2. BridgeInterceptor
补全请求头，Response解压

3. CacheInterceptor
请求前查询缓存，获得响应并判断是否需要缓存
4. ConnectInterceptor
与服务器完成TCP连接
5. CallServerInterceptor
与服务器通信；封装请求数据与解析响应数据(如：HTTP报文)

okhttp支持HTTP2?

ALPN是TLS的扩展，作用：告诉客户端，当前服务端支持的接口协议版本有哪些
连接拦截器，获取tcp连接后找到健康的链接，验证协议，开启tls验证，tls say hello验证结束后，获取tls额外参数，通过ALPN判断协议类型，查看是否支持http2.0。
ConnectInterceptor连接器，streamAllocation.newStream获取连接对象，调用connect方法。
connect方法开启构建socket连接，构建成功后调用establishProtocol。
获取协议，协议包含了HTTP_2，OKhttp就会开启Http2.0，否则则降级成1.1。
connectTls方法来获取协议，内部会通过Platform根据不同平台环境来反射调用实现类。
//Handshake则会把服务端支持的Tls版本，加密方式等都带回来，然后会把这个没有验证过的HandShake用X509Certificate去验证证书的有效性。
Platform去从SSLSocket去获取ALPN的协议支持信息，当后端支持http2.0，则把请求升级到Http2.0。

OKhttp：socket连接池怎么复用的

OKHTTP使用ConnectionPool实现连接池，默认支持5个并发，每个连接生命为5分钟
内部通过connections(双端队列)存储所有的连接。
在连接拦截器中，会通过连接池的get方法找到可用连接，
也就是遍历了双端队列，如果连接有效，就会调用acquire方法计数并返回这个连接。
没找到就创建连接，加入到双端队列，同时开启线程池清理任务。
清理任务中，会不断调用cleanup方法清理线程池，并返回下一次清理的时间间隔，然后进入wait等待。
cleanup方法中，如果空闲连接超过5个或者keepalive时间大于5分钟，则将该连接清理掉。
怎样属于空闲连接？acquire方法，会将连接加入到RealConnection中的虚引用列表，如果连接关闭会移除对象。

拦截器是怎么实现的，用到了什么设计模式， 对OKhttp 还做了哪些优化

责任链模式：拦截器
建造者模式：Request.Builder
工厂模式：CacheInterceptor中CacheStrategy对象
观察者模式：webSocket属于长连接，所以需要进行监听
单例模式

优化：
分发器线程池：并发量最大化

okhttp为什么会使用okio而不是用普通io

Okio并没有打算优化底层IO方式以及替代原生IO方式，Okio优化了缓冲策略以减轻内存压力和性能消耗，并且对于部分IO场景，提供了更友好的API

Okio 是一套基于 java.io 进行了一系列优化的十分优秀的 I/O 库，它通过引入了 Segment 机制大大降低了数据迁移的成本，减少了拷贝的次数，并且对 java.io 繁琐的体系进行了简化，使得整个库更易于使用。

Retrofit

原理

Retrofit是网络请求框架
Retrofit.create返回动态代理对象
invoke:调用loadServiceMethod，从map取出ServiceMethod缓存，如果有就返回。
无缓存就调用parseAnnotations方法。
parseAnnotations：
(1)解析注解存储到RequestFactory
(2)将RequestFactory传递到HttpServiceMethod并返回
接着调用HttpServiceMethod.invoke：创建OkHttpCall(自定义接口的Call),调用CallAdapted的adapt

OkhttpCall.enqueue方法发起请求
1.构建call，call就是OkHttpClient创建的call对象(RealCall)，
2.通过RequestFactory构建okhttp3.Request
3.最终调用RealCall.enqueue发起真正的请求

返回结果解析parseResponse：
T body = responseConverter.convert(catchingBody);
addConverterFactory(GsonConverterFactory.create())
转换器创建会存到集合里，通过responseType返回值类型来取到对应的Converter进行转换。

CallAdapter
addCallAdapterFactory(RxJava2CallAdapterFactory.create())
存储到集合中，之后编辑集合，根据returnType类型取出对应的适配器
adapt：将call包装成一个Observable对象返回

如何支持协程
suspend关键字修饰的方法转换成java会多出一个Continuation参数。
在解析的时候会去判断是否支持kotlin(是否存在suspend关键字)
parseAnnotations解析时返回SuspendForResponse是HttpServiceMethod的子类
adapt：
取出最后一个参数强转成Continuation类型，调用Call的扩展函数awaitResponse，
awaitResponse内部就调用Call的enqueue，成功回调continuation.resume

Glide

Glide：init 工作，比如缓存，线程池，复用池的构建等等。
RequestManagerRetrieve：获得RequestManager请求管理类，然后绑定Fragment 。
SupportRequestManagerFragment：空白Fragment，管理请求的生命周期
RequestManager：对请求的管理封装。

Glide.with().load().into()
生命周期 with
生命周期作用域：
Application：子线程使用with，没有空白fragment
非Application：主线程使用with，空白fragment控制生命周期

Fragment绑定：
通过commitAllowingStateLoss将事务post消息队列中，事务异步处理，
通过map临时记录映射关系，成功后post将移除记录映射

生命周期监听
实例化RequestManager中需要Lifecycle，Lifecycle会在Fragment中创建，Fragment生命周期变化会通过LifeCycle分发到RequestManager。LifeCycle内部存储这LifecycleListener的集合，
RequestManager实现了LifecycleListener。
RequestManager构造中添加Lifecycle监听，onDestroy移除监听

RequestManager分发到各个Target(ImageViewTarget)，也会监听到网络状态广播

线程池+缓存初始化。。。

构建RequestBuilder对象 load

缓存 into
构建ImageViewTarget

获取缓存：
活动缓存->内存缓存LRU->磁盘缓存->网络加载

为什么要活动缓存：
由于LRU的空间限制，可能正在显示的图片，从LRU中移除，导致图片显示出问题。所以用了活动缓存。
网络请求->放入到磁盘缓存->活动缓存
活动缓存释放->存入到LRU内存缓存

活动缓存和LRU是互斥的

内存溢出问题?

大量使用的applicaiton作用域，applicaiton不会对页面管理生命周期，导致回收不及时。

EventBus

```
//key 事件类型
//value 订阅对象集合(订阅对象包含了订阅者和订阅方法)
private final Map<Class<?>, CopyOnWriteArrayList<Subscription>>
subscriptionsByEventType;

//key 订阅者
//value 事件类型集合
private final Map<Object, List<Class<?>>> typesBySubscriber;
```

注册
1.通过反射找出这个类的所有订阅方法，保存集合中
2.创建订阅对象将订阅的这个类对象和订阅方法绑定在一起，根据事件类型将订阅对象保存到map里来完成订阅。
然后在通过订阅的这个类对象作为key，事件类型集合作为value存到另一个map，用来注销遍历的。
3.如果是黏性事件，则取出黏性事件对应的事件类型，发送给订阅者

2.发送事件
找到所有事件对应的类型集合，根据事件类型取出订阅对象集合，进行处理，
如果当前是主线程，直接反射调用方法
如果不是，则将事件加入到主线程队列里通过handler，切换到主线程执行。

3.注销
通过订阅对象找到事件类型集合，遍历集合，通过事件类型获取到订阅对象集合，如果相等则移除。
最后在移除该订阅者对应的事件类型集合

Butterknife

Dagger2

```
module ->提供初始化对象
component->初始化入口桥梁连接activity和module，提供inject注入方法
使用:
DaggerActivityComponent.create().inject(this);

create:初始化module，module内提供通过工厂生成实例
inject:注入当前activity，通过module给activity成员变量赋值
```

Hilt

LeakCanary

```
内存泄漏分析框架。

通过此方法可以监听任何对象的内存泄漏
refWatcher  = LeakCanary.install(this);

通过registerLifeCallback监听activity的onDestroy，把当前activity放到弱引用中，查找引用队列是否存在。

调用packageManager.setComponentEnabledSetting（）方法，实现应用图标的隐藏和显示。
```

watch主要作用：
1.生成一个随机数key存放在retainedKeys集合中，用来判断对象是否被回收；
2.把当前Activity放到KeyedWeakReference（WeakReference的子类）中；
3.通过查找ReferenceQueue，看该Acitivity是否存在，存在则证明可以被正常回收，不存在则证明可能存在内存泄漏。

第三件事情详细描述：
从ReferenceQueue队列中查询是否存在该弱引用对象，如果不为空，则说明已经被系统回收了，则将对应的随机数key从retainedKeys集合中删除。
然后通过判断retainedKeys集合中是否存在对应的key判断该对象是否被回收。
如果没有被系统回收，则手动调用gcTrigger.runGc();后再调用removeWeaklyReachableReferences方法判断该对象是否被回收。
手动触发GC，紧接着线程睡100毫秒，给系统回收的时间，随后通过System.runFinalization()手动调用已经失去引用对象的finalize方法。
通过手动GC该对象还不能被回收的话，则存在内存泄漏，调用heapDumper.dumpHeap()生成.hprof文件目录，并通过heapdumpListener回调到analyze()方法，后面关于dump文件的分析

BlockCanary

UI卡顿监测框架

在Looper的loop方法中，消息处理前，和消息处理后都会通过printer打印日志。
Looper里也提供了setMessageLogging设置自定义printer的方法。
在println方法中通过计算时间差与我们自己设置的阀值来监控我们的程序是否发生卡顿。

ASM的原理

ASM 大致的工作流程是：
1. ClassReader 读取字节码到内存中，生成用于表示该字节码的内部表示的树，ClassReader 对应于访问者模式中的元素
2. 组装 ClassVisitor 责任链，这一系列 ClassVisitor 完成了对字节码一系列不同的字节码修改工作，对应于访问者模式中的访问者 Visitor
3. 然后调用 ClassReader#accept() 方法，传入 ClassVisitor 对象，此 ClassVisitor 是责任链的头结点，经过责任链中每一个 ClassVisitor 的对已加载进内存的字节码的树结构上的每个节点的访问和修改
4. 最后，在责任链的末端，调用 ClassWriter 这个 visitor 进行修改后的字节码的输出工作

JSbridge

JSbridge 是如何实现js和native联通的

JS调用Android基本有下面三种方式
- webView.addJavascriptInterface()
- WebViewClient.shouldOverrideUrlLoading()
- WebChromeClient.onJsAlert()/onJsConfirm()/onJsPrompt() 方法分别回调拦截JS对话框alert()、confirm()、prompt()消息

Android调用JS
- webView.loadUrl();
- webView.evaluateJavascript()

JSbridge
1. Android调用JS是通过loadUrl(url),url中可以拼接要传给JS的对象
2. JS调用Android是通过shouldOverrideUrlLoading

3. JsBridge将沟通数据封装成Message，然后放进Queue,再将Queue进行传输

native - js
调用callHandler函数
将函数名称，传参数据，回调方法，封装到message，
回调方法是通过生成callbackId 放入到map中(responseCallbacks)。
将message转成json，通过loadUrl方式传到js

js收到后，将json转成message，在预注册的messageHandlers map中匹配到方法后执行
如果需要回调，会创建回调对象，callbackId封装成message对象，存入sendMessageQueue
通过改变iframe的src从而通知native从h5取消息，iframe是不可见的的ifrmae，

native拦截到协议后，创建的回调对象,并执行js的_fetchQueue方法取消息，然后将_fetchQueue和回调对象存到了responseCallbacks map中。

js执行_fetchQueue，将messageQueue转换成json，放到iframe的src中，通知native取消息。

native拦截到协议后，取出回调对象并执行回调对象的方法，解析
将数据转成集合，遍历集合，取出message，如果message有callBackId，就从最初的map中取出回调消息，并且执行，执行完成后移除

==========
1、native通过loadurl方式执行js方法，并且生成回调ID，保存到map(id,CallBack)。
2、js匹配到方法后执行，创建回调对象，生成message存入到队列里。在通过iframe通知native取消息
3、native拦截到协议后，再次创建回调(用来解析队列)，保存到map(_fetchQueue,CallBack)，通过loadUrl执行js的_fetchQueue
4、js执行_fetchQueue，将messageQueue转换成json，放到iframe的src中，通知native取消息。
5、native拦截到协议后，取出map的回调（_fetchQueue），将messageQueue转成集合遍历，取出message的回调id，通过id取出map的CallBack，执行完成回调。
==========

js-native
js调用callHandler方法
js将方法，参数，回调的id封装到message，回调方法存到map，message放到队列里，
通过iframe的src通知native

native拦截到协议后，创建的回调对象,并执行js的_fetchQueue方法取消息，然后将_fetchQueue和回调对象(解析 队列)存到了responseCallbacks map中。

js执行_fetchQueue，将messageQueue转换成json，放到iframe的src中，通知native取消息。

native拦截到协议后，取出map里回调对象并执行回调对象的方法解析
将数据转成集合，遍历集合，取出message，如果message有callBackId，创建回调函数(java执行此回调，调用到js)
从messageHandlers(map)消息中取出方法名去匹配native提前的注册的方法池，并回调该方法(上面创建的回调)

回调方法里:
将要回传的数据封装成message,原先从js传过来的callbackId变成responseId
将message转成json，loadUrl调用JS的方法

js将json转成message，通过responseId取出方法，并执行

热修复

热修复框架： Andfix， Qzone， Tinker 修复原理， 有没有看过源码， 机型适配和版本适配怎么做的

Andfix：在native动态替换java层的方法，通过native层hook java层的代码。 即时生效、注解、NDK
Qzone：基于的是dex分包方案。把BUG方法修复以后，放到一个单独的dex补丁文件，让程序运行期间加载dex补丁，执行修复后的方法。
重启生效、反射、类加载
Tinker：Tinker通过计算对比指定的Base Apk中的dex与修改后的Apk中的dex的区别，补丁包中的内容即为两者差分的描述
重启生效、反射、类加载、DexDiff
Robust 预先定义了一个配置文件 robust.xml，在这个配置文件可以指定是否开启插桩、哪些包下需要插桩、哪些包下不需要插桩，

Android 7.0混合编译热修复解决方案
应用在安装时不做编译，在运行时解释字节码。jit编译了代码记录在profile文件，等空闲时，用aot编译生成映射文件。映射文件会在app启动加载。所以类被加载了就无法替换了。
所以需要运行时替换PathClassLoader方案。
Thread.currentThread().setContextClassLoader(classLoade r);
LoadApk classLoader
Resource classLoader
DrawableInflater classLoader

CLASS_ISPREVERIFIED
虚拟机在安装期间为类打上CLASS_ISPREVERIFIED标志是为了提高性能的

为什么报错？
方法中直接引用到的类和clazz都在同一个dex中的话，那么这个类就会被打上CLASS_ISPREVERIFIED，那么就会进行dex的校验。如果两个相关联的类在不同的dex中就会报错。
什么时候标记?
apk在安装的时候，apk中的classes.dex会被虚拟机优化成odex文件，然后才会拿去执行。
虚拟机在启动的时候，其中一项就是verify选项，当verify选项被打开的时候，那么就会执行校验，如果校验类成功，那么这个类会被打上CLASS_ISPREVERIFIED的标志
验证方法？
1. static方法
2. private方法
3. 构造函数

解决：
防止类被打上CLASS_ISPREVERIFIED标志。方案是往所有类的构造函数里面插入了一段代码。

打包成单独的dex，这样当安装apk的时候，classes.dex内的类都会引用一个在不相同dex中的AntilazyLoad类，这样就防止了类被打上CLASS_ISPREVERIFIED的标志了，只要没被打上这个标志的类都可以进行打补丁操作。

然后在应用启动的时候加载进来.AntilazyLoad类所在的dex包必须被先加载进来。

之所以选择构造函数是因为他不增加方法数，一个类即使没有显式的构造函数，也会有一个隐式的默认构造函数。

使用字节码插装

热修复资源id冲突怎么解决

构造一个package id为0x66的资源包，这个包里面只需要包含需要改变的资源项，然后直接在原有AssetManger中通过addAssetPath函数添加这个包就可以了。由于补丁包的package id为0x66，不与目前的已经加载的地址为0x7f的包冲突，因此直接加载到已有的AssetManager中就可以直接使用了。

插件化

启动activity的hook方式。taskAffity。

插件化：宿主activity启动插件activity
hook AMS方式：
启动activity会经过Instrumentation.execStartActivity，调用ActivityManager.getService().startActivity
ActivityManager.getService()会通过Singleton<IActivityManager>,来拿到Ibinder对象，Singleton又是静态的，所以只需要通过动态代理IActivityManager，替换Singleton里的mInstance，每次启动activity都会执行动态代理的方法，通过动态代理拿到启动的activity参数，转换成我们内部写好的代理activity，再将原有启动的activity放入到bundle里，由于插件activity不在宿主的androidmanifest文件中，所以需要欺骗系统。

hook handler:
AMS的startActivity最终会调用到ApplicationThread的scheduleLaunchActivity方法，通过发送消息到内部类H执行ActivityThread的performLaunchActivity启动。
Handler内部有个mCallback字段，发送消息时，会去判断该字段如果不为空就执行mCallback.handleMessage方法，否则就执行handleMessage。所以只需要创建Callback字段，反射设置到H类的mCallback。
接收到消息就能执行到handleMessage取出message。要启动的actiivity 替换代理的activity。

taskAffinity:activity的任务栈名字，默认是包名，也可以指定任务栈的名称。
singleTask配对：
它是具有该模式的Activity的目前任务栈的名字，待启动的Activity会运行在名字和TaskAffinity相同的任务栈中。
allowTaskReparenting配对：
当一个应用A启动了应用B的某个Activity后，如果这个Activity的allowTaskReparenting属性为true的话，那么当应用B被启动后，此Activity会直接从应用A的任务栈转移到应用B的任务栈中。

组件化

module和app之间的区别。moduler通信是如何实现的。

组件化：
把app通过业务分割成不同的module，每个module不能有依赖。module可以单独运行，提高编译速度。

module之间跳转：c
通过编译时注解框架(APT)，跳转功能通过编译生成的辅助类完成。
Activity通过注解填入group和path，注册到辅助类里，通过map保存。
需要跳转时，从map里取出，具体要跳转的类，实现跳转。

module之间通信：
不同模块的图片，方法，数据传递。
在公共基础库层，提供Call接口，主要用于跨模块业务回调，
然后在提供各个业务的接口继承此接口。提供方法。
在具体业务中实现该接口，通过注解填入path。

需要使用的使用通过自定义注解参数来获取。

IRouteRoot:
key为分组名，即路径的第一段，value为分组中所有的映射关系

IRouteGroup:
将module中使用@Route注解的activity或Fragment添加到集合中,key为路径，value为具体的RouteMeta
RouteMeta:具体的Activity.class，传参等

IInterceptorGroup:
将@Interceptor注解的类添加到参数集合中

IProviderGroup:
将继承自IProvider的类添加到参数集合中
IProvider接口是用来暴露服务

navigation:
通过build取出缓存集合中对应的跳转的页面进行跳转。
如果catch到跳转失败，交给全局降级策略处理。

拦截器:
init过程就是把所有注解的信息加载内存中，并且完成所有拦截器的初始化。
在拦截器内部会通过CountDownLatch，进行拦截器计数，
每放行一个拦截器就countDown，并交给后一个拦截器，如果有拦截，将计数器清零。

inject:
通过@Autowired注解的属性，通过调用ARouter.getInstance().inject(this);可以实现自动注入。

APT生成注射器。
通过navigation获取到参数解析，赋值给Activity/Fragment的注解参数。

ASM:
在初始化时，会找到app的dex，然后遍历出其中的属于`com.alibaba.android.arouter.routes`包下的所有类名(APT生成包下的所有类)，打包成集合返回。可以想象遍历整个dex查找指定类名的工作量有多大。
通过字节码插装去动态注册。
register("com.alibaba.android.arouter.routes.ARouter$$Root$$modulejava");

组件化-A组件要调用B组件的某个功能，怎么调用（回答了接口下沉，AutoService等等）

通用工具层：编写接口类(让接口下沉)
其他组件：引入工具层，编写实现接口类，并加上注解AutoService(接口的class)
通过Service.load方式获取到所有的实现类，调用相关的组件功能

组件化-如果不用接口下沉，没有任何依赖，怎么方便的调用（不让用反射，太麻烦）

通过APT，将所有组件加入到map管理。
事件分发？

组件总线
注册组件方法到map中，需要调用时，根据传入的参数进行匹配，使用反射调用。

接口+路由

ARouter CC

Jetpack

LifeCycle

Livecycle是一个表示android生命周期及状态的对象
LivecycleOwner用于连接有生命周期的对象，如activity,fragment
LivecycleObserver用于观察查LifecycleOwner
使用：getLifecycle().addObserver(new MyObserver());添加观察者

addObserver:将LivecycleObserver封装成ObserverWithState(将观察者进行封装)，然后放入自定义map里
Actiivty实现了LifecycleOwner，持有LifecycleRegistry对象(聚合多个LifecycleObserver,生命周期改变时进行通知)
在onCreate中注入了空白的fragment监听生命周期，fragment生命周期里会去分发相应的事件，
事件通过LifecycRegistry通知到各个LifecycleObserver。
事件触发后，获取到activity的下一个状态，进行分析是正推还是逆推。
通过map找到ObserverWithState，通知LifecycleObserver生命周期的变化。
观察者实现类ReflectiveGenericLifecycleObserver进行反射调用通知到注解方法

livedata

```
mLiveData = new MutableLiveData<>();
mLiveData.observe(this, new Observer<String>() {
    @Override
    public void onChanged(String s) {
    }
});
//主线程
mLiveData.setValue("onCreate");
//子线程
mLiveData.setValue("onCreate");
```

livedata:是一种具有生命周期感知能力的可观察数据持有类
observer入口:将观察者和被观察者包装在一起存到map中，通过addObserver将包装对象添加为观察者
setValue:遍历map，获取到包装对象，调用considerNotify通知方法
considerNotify:判断activity状态是否显示,调用onChanged。

管理所有的LiveData设计中有bug
事件黏性问题：未启动的页面，打开页面后添加监听，然后会接收到最新的消息。(此消息是之前发的，不是监听完成后发的)
问题起因:由于用map存储事件类型，每次订阅都会复用之前缓存的livedata，导致页面进入后mLastVersion<mVersion后直接调用onChanged方法。
解决:在observe中，将mLastVersion和mVersion改成相等，return，就不会触发黏性问题。

DataBinding

databinding:降低布局和逻辑的耦合性
xml分离:
layout->@{}生成tag
data->外层：id,tag,view类型;内层：对应的位置行号等

DataBindingUtil.setContentView:XML文件信息读取，并存入数组中
DatabindBinding.setData(Observable):
将ViewDataBinding保存到被观察者中的注册器列表里。
发送通知，通过注册器分发到ViewDataBinding，在调用到DataBinding的实现类，通过适配器设置相关的数据信息

ViewModel

管理数据的保存与恢复，比如屏幕转动

进入页面:
getLastNonConfigurationInstance
屏幕翻转:
onRetainCustomNonConfigurationInstance: 保存状态　转屏时自动调用
getLastNonConfigurationInstance: 恢复状态

在屏幕翻转，AMS会调用到Activity的retainNonConfigurationInstances

ViewModelProviders.of(this).get(ViewModel.class)
of:创建ViewModelStore相当于一个map容器，用来保存ViewModel的
get:通过工厂反射创建出ViewModel对象保存到map里。

onRetainCustomNonConfigurationInstance:进行数据保存
屏幕翻转后将ViewModelStore保存到NonConfigurationInstances
下次通过of去获取的时候直接从getLastNonConfigurationInstance获取到NonConfigurationInstances取出ViewModelStore

Navigition

NavHostFragment主导航持有控制器
NavHostController控制器
Navigator导航器

NavHostFragment.create:
初始化NavHostFragment将导航视图id和拦截系统返回标记存入到bundle里

onInflate：解析参数（调用时机：Fragment所关联的Activity在执行setContentView时）

onCreateNavController：
创建FragmentNavigator(实现导航功能和回退栈)存储到NavigatorProvider的HashMap容器里

onCreate：
初始化NavHostController控制器，设置导航视图，导航到第一个fragment

反射创建启动的fragment，通过repleace方式替换第一页

onCreateView:
创建FragmentContainerView

onViewCreate：
将控制器设置到Navigtaion

使用导航：
Navigation.findNavController(view).navigate(id);
通过控制器导航，获取目的地id，通过repleace方式替换第一页

基本流程
控制器持有导航器、解析器、导航目的地
通过解析器解析xml到目的地，获取到第一个目的地framgent，在通过导航器repleace第一个页面

Pagging

分页管理

WorkManager

处理非及时任务

WorkManager是怎么保证，当我把APP杀掉后呢？
记录用户的所有信息并全部保存到数据库，而并非保存在内存中，这样做的好处，就是持久性保存记录，所有APP被杀掉后 依然可以获取所有任务信息。

Android操作系统会在系统级别服务中，来判断用户的约束条件，当约束条件满足时就会执行任务，但是触发检测是采用广播的形式处理的，例如：网络连接成功就触发

View相关

view的创建过程

```
PhoneWindow在Activity.attach方法中创建
DecorView 的创建时机:
Activity.setContentView -> PhoneWindow.setContentView -> installDecor-
>generateDecor
setContentView 流程:
1. 创建 DecorView
2. 根据 layoutResId 创建 View 并添加到 DecorView 中
```

View的事件分发流程

```
IntercepuTouchEvent
dispatchTouchEvent
onTouchEvent

Activity和View没有拦截方法。
注意：ViewGroup先分发后拦截
```

View滑动冲突

```
内部拦截:调用requestDisallowInterceptTouchEvent修改父ViewGroup的mGroupFlags
外部拦截:intereceptTouchEvent方法内判断拦截
```

view分发反向制约的方法

```
父View如果拦截之后子View就 无法再消费到事件。

造成原因:
if (disallowIntercept) {
    mGroupFlags |= FLAG_DISALLOW_INTERCEPT;
} else {
    mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
}

解决：通过requestDisallowInterceptTouchEvent()方法能够设置mGroupFlag的值，从而调整if条
件是否满足。
```

View的三种测量模式理解，什么时候会发生Excatly

```
Excatly-精准测量  match_parent , xxxdp
AtMost-最大测量 wrap_content
UnSpecial - ScrollView

测量规格 32位int  = Mode 高2位 + size低30位
```

touch事件源码问题

其实android事件分发核心是在viewgroup的dispatchTouchEvent的action_down过程中找到mFirstTouchTarget是否为空，
通过反序遍历子view的dispatchTouchEvent的方法，
如果发现有一个子view的dispatchTouchEvent方法返回true，那么mFirstTouchTarget就不为空，否则为空。
如果mFirstTouchTarget不为空，那么action_move和action_up才会往下传递，
如果在action_move和action_up过程中有viewgroup拦截了事件，则此时先向子view的dispatchTouchEvent传递一个action_cancel，并且将mFirstTouchTarget至为null，所以此时action_move和action_up只会走viewgroup的dispatchTouchEvent和onTouchEvent；如果mFirstTouchTarget在action_down过程中就已经null的话，则从action_down一直向上层view传递，不会有后续的action_move和action_up了。

onTouchEvent/onTouchListenr.onTcouchEvent/onClickListenr

onTouch指setOnTouchListener的回调方法，它是优先于onTouchEvent事件的

onClick事件是在onTouchEvent消费事件中的action_up触发的，
onTouch是在dispatchTouchEvent中触发的，所以onTouch要先于onClick事件，我们也可以通过onTouch返回true来屏蔽掉onClick事件。

```
if (onFilterTouchEventForSecurity(event)) {
    ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnTouchListener != null
            && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnTouchListener.onTouch(this, event)) {
        result = true;
    }
    //li.mOnTouchListener.onTouch 返回true这边的onTouchEvent就不会执行了
    if (!result && onTouchEvent(event)) {
        result = true;
    }
}
```

谈谈对自定义View的理解。

构造函数:
1参：new出来
2参：xml里使用，自定义属性
3参和4参：与主题相关
int defStyleAttr：主题中优先级最高的属性
int defStyleRes： 优先级次之的内置于View的style

Xml定义 > xml的style定义 >defStyleAttr> defStyleRes> theme直接定义

获取view的宽高时机
Measure过程完成，就可通过getMeasuredWidth()、getMeasuredHeight()获取测量宽高。但某些极端情况
需要多次Measure才能确定最终宽高。所以在onLayout方法中获取测量宽高是真正ok的。

我们知道，activity的onCreate中无法获取到view的宽高。实际onCreate、onStart、onResume都不能保证view已完成测量，所以可能获取的都是0。因为view的measure和activity生命周期不是同步的。

onWindowFocusChanged：View已初始化完毕，宽高已准备ok。 但会多次调用，获取焦点、失去焦点都回调用。(这个回调是ViewRootIml中分发到DecorView，接着到Activity、到各级View。)

view.post可以把runnable放入消息队列，等待looper到此runnable是view已经初始化完成。

ViewTreeObserver有很多回调，其中有个OnGlobalLayoutListener，当View树的状态发生改变或者View树内部view的可见性发生改变时 方法 onGlobalLayout()都会被调用。所以是会回调多次。 此时也可以获取view的宽高：

view的事件分发

view的3个主要方法
onMeasure 测量规格

onLayout 确定本身位置，和子view的位置

onDraw
draw过程：
1、画背景
2、画自己-- onDraw，自己实现
3、画子view-- dispatchDraw
4、画装饰

measuredWidth在测量过程产生，getWidth()在layout过程产生

View.post:

1.View.post(Runnable) 内部会自动分两种情况处理，当 View 还没 attachedToWindow 时，会先将这些 Runnable 操作缓存下来；否则就直接通过 mAttachInfo.mHandler 将这些 Runnable 操作 post 到主线程的 MessageQueue 中等待执行。
2.如果 View.post(Runnable) 的 Runnable 操作被缓存下来了，那么这些操作将会在 dispatchAttachedToWindow() 被回调时，通过 mAttachInfo.mHandler.post() 发送到主线程的 MessageQueue 中等待执行。
3.mAttachInfo 是 ViewRootImpl 的成员变量，在构造函数中初始化，Activity View 树里所有的子 View 中的 mAttachInfo 都是 ViewRootImpl.mAttachInfo 的引用。
4.mAttachInfo.mHandler 也是 ViewRootImpl 中的成员变量，在声明时就初始化了，所以这个 mHandler 绑定的是主线程的 Looper，所以 View.post() 的操作都会发送到主线程中执行，那么也就支持 UI 操作了。
5.dispatchAttachedToWindow() 被调用的时机是在 ViewRootImpl 的 performTraversals() 中，该方法会进行 View 树的测量、布局、绘制三大流程的操作。
6.Handler 消息机制通常情况下是一个 Message 执行完后才去取下一个 Message 来执行（异步 Message 还没接触），所以 View.post(Runnable) 中的 Runnable 操作肯定会在 performMeaure() 之后才执行，所以此时可以获取到 View 的宽高。

view绘制流程

ActivityThread通过attach将ApplicationThread和AMS建立联系
经过层层调用执行到handleResumeActivity()方法，在方法中先调用Activity.onResume()方法，再执行WindowManager的addView()方法将Activity的根View(DecorView)添加上去，进而开始绘制流程。
WindowManager实现类为WindowManagerImpl，WindowManagerImpl中addView()方法又会调用WindowManagerGlobal的addView()方法。

addView()方法中先创建ViewRootImpl对象,随后执行setView()方法将其和DecorView绑定起来，绘制流程也将由ViewRootImpl()来执行。setView()方法中会执行requestLayout()方法。
requestLayout()方法走下去会异步执行performTraversals()方法

然后执行三大流程：
performMeasure
performLayout
performDraw

performMeasure：
measure流程开始执行之前，会先计算出DecorView的MeasureSpec。然后执行DecorView的measure()方法开始整个View树的测量。
所有View都会执行到View类的onMeasure()方法。
onMeasure
测量出子View的MeasureSpec后，再执行子View的measure流程
给自己mMeasureWidth&Height赋值。

performLayout
确定子View在父View中的位置
performLayout方法中会执行DecorView的layout()方法来开始整个View树的layout流程。
layout()方法中会先执行setFrame()方法确定View自己在父View中的位置，接着再执行onLayout()方法来遍历所有的子View，计算出子View在自己心中的位置(4个点)后，再执行子View的layout流程。

performDraw

```
//绘制自己的背景
drawBackground(canvas);
//空实现，绘制自己的内容，自定义时重写该方法
onDraw(canvas)
//绘制子View
dispatchDraw(canvas);
//绘制前景
onDrawForeground(canvas);
```

setContentView调用的是PhoneWindow的setContentView()
window在activity.attach中创建
setContentView内调用installDecor
installDecor:
generateDecor->初始化DecorView
generateLayout->初始化mContentParent
加载layoutResource添加到DecorView

Activity启动流程：
ActivityThread.handleLaunchActivity->ActivityThread.performLaunchActivity->Activity.attach->
Activity.onCreate->ActivityThread.handleResumeActivity->ActivityThread.performResumeActivity->
Activity.onResume->windowManager.addView

通过调用windowManager.addView(decor, lp)，将DecorView和WindowManager建立联系。
WindowManagerImpl将DecorView作为根布局加入到PhoneWindow中去。
WindowManagerImpl并没有直接实现操作View的相关方法，而是全部交给WindowManagerGlobal。
WindowManagerGlobal是一个单例类，一个进程中最多仅有一个。

addView
将decorView添加到viewRootImpl中，通过viewRootImpl对其内部的view进行管理
为新窗口创建一个ViewRootImpl对象，负责与WMS进行通讯，管理Surface。

setView
viewRootImp.setView()->viewRootImp.requestLayout()-
>viewRootImp.scheduleTraversals()->
viewRootImp.doTraversal()->viewRootImp.performTraversals()->进入view的绘制流程

token:
每一个新窗口必须通过LayoutParams.token向WMS出示相应的令牌才可以。

view， window， activity关系

Activity->PhoneWindow->DecoreView->TitleView,ContentView
每个activity都会对应一个window

view 过度绘制解析，掉帧原因分析，怎么监测，怎么解决

Systrace分析掉帧原因
编舞者监测帧数丢失

重复定义背景
canvas.clipRect 调整画布大小

cpu和GPU 绘制UI 流程

https://www.jianshu.com/p/9f379c5ff66e

https://wanandroid.com/wenda/show/16190

https://wanandroid.com/wenda/show/15582

https://www.wanandroid.com/wenda/show/8644

https://tech.meituan.com/2017/01/19/hardware-accelerate.html

自定义view实现过程， 自定义view 怎么安全的刷新

onMeasure,onLayout,onDraw
在View#onDetachedFromWindow中停止动画或线程。
`invalidate`方法只能在UI线程里面调用，
`View`会通过`mPrivateFlags`来判断是否进行`measure`和`layout`操作。而在调用`invalidate`方法时，更新了`mPrivateFlags`，所以最终只会走`draw`流程。
而`postInvalidate`方法是无所谓线程。
requestLayout要在主线程调用

常用动画的种类，属性动画的使用， 插值器和估值器的使用

帧动画

补间动画

```
ImageView spaceshipImage = (ImageView) findViewById(R.id.spaceshipImage);
Animation hyperspaceJumpAnimation = AnimationUtils.loadAnimation(this,
R.anim.hyperspace_jump);
spaceshipImage.startAnimation(hyperspaceJumpAnimation);
```

属性动画

java 类名	xml 关键字	描述信息
ValueAnimator	`<animator>` 放置在 res/animator/ 目录下	在一个特定的时间里执行一个动画
TimeAnimator	不支持/点我查看原因	时序监听回调工具
ObjectAnimator	`<objectAnimator>` 放置在 res/animator/ 目录下	一个对象的一个属性动画
AnimatorSet	`<set>` 放置在 res/animator/ 目录下	动画集合

ValueAnimator封装了动画的 TimeInterpolator 时间插值器和一个 TypeEvaluator 类型估值

TimeInterpolator 使用了AccelerateDecelerateInterpolator

TypeEvaluator 使用了 IntEvaluator。

```
ValueAnimator animator = ValueAnimator.ofFloat(0, mContentHeight);  //定义动画
animator.setTarget(view);   //设置作用目标
animator.setDuration(5000).start();
animator.addUpdateListener(new AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator animation){
        float value = (float) animation.getAnimatedValue();
        view.setXXX(value);  //必须通过这里设置属性值才有效
        view.mXXX = value;  //不需要setXXX属性方法
    }
});
```

Evaluators 相关类解释： Evaluators 就是属性动画系统如何去计算一个属性值。它们通过 Animator 提供的动画的起始和结束值去计算一个动画的属性值。

- IntEvaluator：整数属性值。
- FloatEvaluator：浮点数属性值。
- ArgbEvaluator：十六进制color属性值。
- TypeEvaluator：用户自定义属性值接口，譬如对象属性值类型不是 int、float、color 类型， 你必须实现这个接口去定义自己的数据类型。

```
```

```
ValueAnimator valueAnimator = new ValueAnimator();
valueAnimator.setDuration(5000);
valueAnimator.setObjectValues(new float[2]); //设置属性值类型
valueAnimator.setInterpolator(new LinearInterpolator());
valueAnimator.setEvaluator(new TypeEvaluator<float[]>()
{
    @Override
    public float[] evaluate(float fraction, float[] startValue,
                            float[] endValue)
    {
        //实现自定义规则计算的float[]类型的属性值
        float[] temp = new float[2];
        temp[0] = fraction * 2;
        temp[1] = (float)Math.random() * 10 * fraction;
        return temp;
    }
});

valueAnimator.start();
valueAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener()
{
    @Override
    public void onAnimationUpdate(ValueAnimator animation)
    {
        float[] xyPos = (float[]) animation.getAnimatedValue();
        view.setHeight(xyPos[0]);   //通过属性值设置View属性动画
        view.setWidth(xyPos[1]);    //通过属性值设置View属性动画
    }
});
```

ObjectAnimator：继承自 ValueAnimator，需要目标对象的属性提供指定的处理方法（setXXX，getXXX），内部提供了类提供了 ofInt、ofFloat、ofObject 这个三个常用的方法。

动画原理是不停的调用 setXXX 方法更新属性值

```
ObjectAnimator mObjectAnimator= ObjectAnimator.ofInt(view,
"customerDefineAnyThingName", 0,  1).setDuration(2000);
mObjectAnimator.addUpdateListener(new AnimatorUpdateListener()
        {
            @Override
            public void onAnimationUpdate(ValueAnimator animation)
            {
                //int value = animation.getAnimatedValue();  可以获取当前属性值
                //view.postInvalidate();  可以主动刷新
                //view.setXXX(value);
                //view.setXXX(value);
                //......可以批量修改属性
            }
        });
```

```
ObjectAnimator.ofFloat(view, "rotationY", 0.0f,
360.0f).setDuration(1000).start();
```

插值器

Interpolators 相关类解释：

- AccelerateDecelerateInterpolator：先加速后减速。
- AccelerateInterpolator：加速。
- DecelerateInterpolator：减速。
- AnticipateInterpolator：先向相反方向改变一段再加速播放。
- AnticipateOvershootInterpolator：先向相反方向改变，再加速播放，会超出目标值然后缓慢移动至目标值，类似于弹簧回弹。
- BounceInterpolator：快到目标值时值会跳跃。
- CycleInterpolator：动画循环一定次数，值的改变为一正弦函数：`Math.sin(2 * mCycles * Math.PI * input)`。
- LinearInterpolator：线性均匀改变。
- OvershottInterpolator：最后超出目标值然后缓慢改变到目标值。
- TimeInterpolator：一个允许自定义 Interpolator 的接口，以上都实现了该接口。

```
//开始很慢然后不断加速的插值器。
public class AccelerateInterpolator implements Interpolator {
    private final float mFactor;
    private final double mDoubleFactor;

    public AccelerateInterpolator() {
        mFactor = 1.0f;
        mDoubleFactor = 2.0;
    }

    ......

    //input  0到1.0。表示动画当前点的值，0表示开头，1表示结尾。
    //return  插值。值可以大于1超出目标值，也可以小于0突破低值。
    @Override
    public float getInterpolation(float input) {
        //实现核心代码块
        if (mFactor == 1.0f) {
            return input * input;
        } else {
            return (float)Math.pow(input, mDoubleFactor);
        }
    }
}
```

硬件加速的原理

https://mp.weixin.qq.com/s?__biz=MjM5NjQ5MTI5OA==&mid=2651745986&idx=2&sn=c585a4e7a2265cef1012e5f6cc43a811&chksm=bd12b78f8a653e99ae1779538edf84c8d3da2a8e7ca904a9f078a1772a9bf8a63056ec7e87f1&scene=38#wechat_redirect

```
Android4.0默认开启硬件加速

CPU的ALU算术逻辑单元比较少，GPU的ALU比较多，计算能力强，为实现大量数学运算设计的。
硬件加速的主要原理，就是通过底层软件代码，将CPU不擅长的图形计算转换成GPU专用指令，由GPU完成。

Android绘制流程：
```

从ViewRootImpl.performTraversals到PhoneWindow.DecroView.drawChild是每次遍历View树的固定流程。
使用软件绘制，生成的Canvas对象
硬件加速，生成的DisplayListCanvas对象
通过isHardwareAccelerated来判断是否支持硬件加速

View中的draw(canvas,parent,drawingTime) - draw(canvas) - onDraw - dispachDraw - drawChild这条递归路径，
调用了Canvas.drawXxx()方法，在软件渲染时用于实际绘制；在硬件加速时，用于构建DisplayList。

View中的updateDisplayListIfDirty - dispatchGetDisplayList - recreateChildDisplayList这条递归路径仅在硬件加速时会经过，用于在遍历View树绘制的过程中更新DisplayList属性，并快速跳过不需要重建DisplayList的View。

硬件加速情况下，draw流程执行结束后DisplayList构建完成，然后通过ThreadedRenderer.nSyncAndDrawFrame()利用GPU绘制DisplayList到屏幕上。

view的绘制刷新机制是怎么样的？vsync信号发出后怎么触发绘制逻辑的？

单缓存:可能导致画面撕裂，在缓存中有些数据没有显示出来就被重写了，导致buffer里的数据可能是来自不同的帧的。

在Android4.1之前,屏幕刷新采用双缓存+VSync机制。
显示器使用 Frame Buffer，GPU将完成的一帧图像数据写入到 Back Buffer。当屏幕刷新时，进行交换。
缓存的交换是在Vsyn到来时进行，交换后屏幕会取Frame buffer内的新数据，正常显示后，开始后一帧的数据的处理。
缺陷：此时CPU/GPU的计算时间没有足够的16.6ms，所以掉帧(jank)的几率较大。

Android 4.1系统中对Android Display系统进行了重构,一旦收到VSync通知（16ms触发一次），CPU和GPU才立刻开始计算然后把数据写入buffer。VSync同步使得CPU/GPU充分利用了16.6ms时间，减少jank。
缺陷：CPU/GPU的处理时间超过了16.6ms，导致下一次应该执行的缓冲区交换又被推迟了——如此循环反复，便出现了越来越多的“Jank”。
两个buffer都被占用，CPU 则无法准备下一帧的数据。

三缓存：有效利用了等待vysnc的时间，减少了jank，但是带来了延迟。
(Buffer 正常还是两个，当出现 Jank 后三个足以。)

所有UI的变化都是走到ViewRootImpl的scheduleTraversals()方法。在VSync信号到来时才会执行绘制，即performTraversals()方法。

scheduleTraversals方法:
1.boolean mTraversalScheduled保证同时间多次更改只会刷新一次
2.添加同步屏障，屏蔽同步消息，保证VSync到来立即执行绘制
3.调用Choreographer.postCallback()发送下一帧执行的回调
下一个VSync到来时会执行TraversalRunnable-->doTraversal()--->performTraversals()-->绘制流程。
doTraversal里移除同步屏障，开始三大绘制流程

Choreographer是在ViewRootImpl构造中初始化的，通过ThreadLocal.get保证线程单例。
Choreographer构造中创建了FrameHandler、FrameDisplayEventReceiver VSync事件接收器、CallbackQueue任务链表数组
Choreographer.postCallback(任务类型，任务，NULL),内部调用到postCallbackDelayedInternal
任务类型:输入事件>动画>插入更新的动画>绘制>提交

postCallbackDelayedInternal:
1.将任务添加到数组中对应的事件链表中。
2.没有延迟会执行scheduleFrameLocked()方法，有延迟就会使用 mHandler发送MSG_DO_SCHEDULE_CALLBACK消息(消息设置异步)
FrameHandler处理：
MSG_DO_FRAME->doFrame 绘制过程
MSG_DO_SCHEDULE_VSYNC->doScheduleVsync申请VSYNC信号(最终调用scheduleVsyncLocked)
MSG_DO_SCHEDULE_CALLBACK->doScheduleCallback需要延迟的任务 (内部调用scheduleFrameLocked)

scheduleFrameLocked:
未开启VSYNC:发送MSG_DO_FRAME执行doFrame
4.1默认开启VSYNC:调用scheduleVsyncLocked，发出同步信号(接收器中的native方法)

接收回调onVsync：使用mHandler发送异步msg，最终执行的就是doFrame()方法了。

doFrame:按类型顺序执行任务
doCallbacks:查找到指定类型的队列，迭代执行任务CallbackRecord.run。
CallbackRecord.run:
token==FRAME_CALLBACK_TOKEN->doFrame (通过postFrameCallback 或postFrameCallbackDelayed，会执行这里)
token==null->最终就执行到ViewRootImpl发起的绘制任务mTraversalRunnable了

postFrameCallback->postFrameCallbackDelayed->postCallbackDelayedInternal(CALLBACK_ANIMATION类型)
postFrameCallback通常用来计算丢帧情况

页面静止的时候，onDraw会执行吗？vsync信号会发吗?

只有当界面有刷新的需要时，我们 app 才会在下一个屏幕刷新信号来时，遍历绘制 View 树来重新计算屏幕数据。
如果界面没有刷新的需要，一直保持不变时，我们 app 就不会去接收每隔 16.6ms 的屏幕刷新信号事件了，但底层仍然会以这个固定频率来切换每一帧的画面，只是后面这些帧的画面都是相同的而已。

vsync信号发出的时候，怎么控制需不需要onDraw?

如果发起了重绘的请求，需要先层层走到 ViewRootImpl 里去，而且还不是马上就执行重绘操作，而是需要等待下一个屏幕刷新信号来的时候，再从 DecorView 开始层层遍历到这些需要刷新的 View 里去重绘它们。

用canvas怎么绘制一个倒影

https://mp.weixin.qq.com/s?__biz=MzAxMTI4MTkwNQ==&mid=2650822527&idx=1&sn=d2f921339e996b7aa291aa785e85afb8&chksm=80b783e1b7c00af797aa56ab6914b7bbec85ed091b5ced42aab4f417cfac05bd93c5a6ee9643&scene=38#wechat_redirect

https://blog.csdn.net/briblue/article/details/53694042

```
创建原图
通过Matrix创建反向的bitmap
合成图片
通过画布将原图和倒影图画在合成图上
通过LinearGradient添加遮罩
通过Canvas.drawRect将遮罩画上去
```

对canvas做矩阵变换的matrix有哪些参数配置

```
public static final int MSCALE_X = 0;   //!< use with getValues/setValues
public static final int MSKEW_X  = 1;   //!< use with getValues/setValues
public static final int MTRANS_X = 2;   //!< use with getValues/setValues
public static final int MSKEW_Y  = 3;   //!< use with getValues/setValues
public static final int MSCALE_Y = 4;   //!< use with getValues/setValues
public static final int MTRANS_Y = 5;   //!< use with getValues/setValues
public static final int MPERSP_0 = 6;   //!< use with getValues/setValues
public static final int MPERSP_1 = 7;   //!< use with getValues/setValues
public static final int MPERSP_2 = 8;   //!< use with getValues/setValues

SkMatrix逻辑上是一个 3 x 3 矩阵，物理上是一个1 x 9 数组

set设置了矩阵相应属性，而其他属性被清除
preXXX左乘
postXXX右乘


Translate ：对应是 MTRANS_X 和 MTRANS_Y 值
Scale :对应的是 MSCALE_X 和 MSCALE_Y 值
Rotate:对应是 MSCALE_X、MSCALE_Y、MSKEW_X 和 MSKEW_Y 值
Skew:对应的是 MSKEW_X 和 MSKEW_Y

MPERSP_0:在3D变换中有着至关重要的作用
```

线性布局和相对布局那个性能更优

RelativeLayout会对子View做两次measure。
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 2个子View，横向上B依赖A，纵向上A依赖B。所以需要横向纵向分别进行一次排序测量。

如果不使用weight属性，LinearLayout会在当前方向上进行一次measure的过程，如果使用weight属性，LinearLayout会避开设置过weight属性的view做第一次measure，完了再对设置过weight属性的view做第二次measure。

RelativeLayout需要对其子View进行两次measure过程。而LinearLayout则只需一次measure过程，所以显然会快于RelativeLayout，但是如果LinearLayout中有weight属性，则也需要进行两次measure，但即便如此，应该仍然会比RelativeLayout的情况好一点。

1.RelativeLayout会让子View调用2次onMeasure，LinearLayout 在有weight时，也会调用子View2次onMeasure
2.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。
3.在不影响层级深度的情况下,使用LinearLayout和FrameLayout而不是RelativeLayout。

Viewstub实现原理，自己设计一个类似的

ViewStub 是一个看不见的，没有大小，不占布局位置的 View，可以用来懒加载布局。

ViewStub初始化会状态为GONE，设置为不会绘制。
通过setMeasuredDimension(0, 0)不占用大小。

setVisibility->inflate
获取布局,计算出ViewStub位置,删除ViewStub,添加需要加载的view

recyclerview和listview对比

listview 2级缓存:
屏幕内复用和移除屏幕复用
从屏幕内获取view获取不到从屏幕外缓存获取view调用adapter.getView，参数convertView就是复用屏幕外的view

RecyclerView 4级缓存:
屏幕内复用
屏幕外复用
自定义
缓存池

移除屏幕的ViewHolder存储到缓存view，如果缓存view超出数量，将最老的ViewHolder存进缓存池中，还原状态。

ListView复用的是getView方法里的参数convertView。(自己实现ViewHolder设置到convertView的tag中)
RecyclerView复用的是ViewHolder

ViewHolder:用来保持着 convertView 对里面每个子 View 的引用，避免每次都要 findViewById。

子线程更新UI

在onCreate中创建子线程更新ui是能更新成功的，如果子线程等待一会在去更新是失败的。
原因：
在ViewRootImpl会去判断UI更新是否在主线程，不在就抛异常
onCreate中ViewRootImpl还没启来，在ActivityThread.handleResumeActivity调用activity.makeVisible，
WindowManager.addView->WindowManagerImpl.addView->WindowManagerGlobal.addView->创建ViewRootImpl

性能优化

什么是 ANR 如何避免它

不要在主线程中进行耗时的操作。如要执行耗时操作，必须将耗时操作放到子线程中去执行，通过handlerd对主线程进行通信。
ANR类型有4种：
KeyDispatchTimeout:Input事件在5S内没有处理完成发生了ANR。

BroadcastTimeout:
前台Broadcast：onReceiver在10S内没有处理完成发生ANR。
后台Broadcast：onReceiver在60s内没有处理完成发生ANR。

ServiceTimeout
前台Service：onCreate，onStart，onBind等生命周期在20s内没有处理完成发生ANR。
后台Service：onCreate，onStart，onBind等生命周期在200s内没有处理完成发生ANR

ContentProviderTimeout
ContentProvider 在10S内没有处理完成发生ANR。

线上监控
watchDog方案，自定义线程，不断从主handler发送消息，发送前记录标志位，
是否完成，发送时间，wait 5S，查看下任务是否执行完成，是否超时。超时就上报数据

FileObserver已经不适用，在5.0之后会被selinux挡住。

怎么分析anr问题

1. 定位发生ANR时间点
2. 查看trace信息(保存在data/anr/traces.txt文件)
3. 分析是否有耗时的message,binder调用，锁的竞争，CPU资源的抢占
4. 结合具体的业务场景的上下文来分析

如果 CPU 使用量接近 100%，说明当前设备很忙，有可能是 CPU 饥饿导致了 ANR
如果 CPU 使用量很少，说明主线程被 BLOCK 了
如果 IOwait 很高，说明 ANR 有可能是主线程在进行 I/O 操作造成的

案例 1：关键词:ContentResolver in AsyncTask onPostExecute, high iowait
100%TOTAL: 6.9% user + 8.2% kernel +84%iowait
原因：IOWait 很高，说明当前系统在忙于 I/O，因此数据库操作被阻塞

案例 3：

关键词：Memoryleak/Thread leak
分析：
关键词：Memoryleak/Thread leak
解决：查看哪些内存没有释放
内存不足的时候不一定第一时间导致oom,因为当发现内存不足的时候，首先产生的现象是导致gc频率增大，从而可能导致anr

查看mobilelog文件夹下的events_log,从日志中搜索关键字：am_anr，找到出现ANR的时间点、进程PID、ANR类型。
在这个时间的前5秒时间段做了什么事情,再搜索一下pid的日志,查到收到了一个推送消息发生阻塞。
搜索关键字关键字：ANR IN查看占用CPU资源。查看traces文件日志分析,定位到具体位置。

如果是死锁导致的anr，在日志上怎么看

```
waiting on <0x1cd570> (a android.os.MessageQueue)
DALVIK THREADS:
"main" prio=5 tid=3 TIMED_WAIT
  | group="main" sCount=1 dsCount=0 s=0 obj=0x400143a8
  | sysTid=691 nice=0 sched=0/0 handle=-1091117924
  at java.lang.Object.wait(Native Method)
  - waiting on <0x1cd570> (a android.os.MessageQueue)
  at java.lang.Object.wait(Object.java:195)
  at android.os.MessageQueue.next(MessageQueue.java:144)
  at android.os.Looper.loop(Looper.java:110)
  at android.app.ActivityThread.main(ActivityThread.java:3742)
  at java.lang.reflect.Method.invokeNative(Native Method)
  at java.lang.reflect.Method.invoke(Method.java:515)
  at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:739)
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:497)
  at dalvik.system.NativeStart.main(Native Method)

"Binder Thread #3" prio=5 tid=15 NATIVE
  | group="main" sCount=1 dsCount=0 s=0 obj=0x434e7758
  | sysTid=734 nice=0 sched=0/0 handle=1733632
  at dalvik.system.NativeStart.run(Native Method)

"Binder Thread #2" prio=5 tid=13 NATIVE
  | group="main" sCount=1 dsCount=0 s=0 obj=0x1cd570
  | sysTid=696 nice=0 sched=0/0 handle=1369840
  at dalvik.system.NativeStart.run(Native Method)

"Binder Thread #1" prio=5 tid=11 NATIVE
  | group="main" sCount=1 dsCount=0 s=0 obj=0x433aca10
  | sysTid=695 nice=0 sched=0/0 handle=1367448
  at dalvik.system.NativeStart.run(Native Method)

----- end 691 -----
```

```
"main" prio=5 tid=1 Blocked
- waiting to lock <0x0520de84> (a java.lang.Object) held by thread 22
```
主线程 被Blocked，它在等待一个被22号线程持有的对象锁

```
"Thread-654" prio=5 tid=22 Blocked
- waiting to lock <0x00e3266d> held by thread 1
- locked <0x0520de84> (a java.lang.Object)
```
可以看出这个线程的确锁住了一个对象，该对象正是主线程正在等待上锁的对象，那这个线程为何没有释放锁呢，因为它在等一个被1号线程持有的对象锁，因此死锁问题导致了ANR现象。

内存泄露的分类。怎么查看内存泄露的问题

MAT,LeakCanary查看内存泄漏

1.集合用完记得clear，置空
2.工具类(单例)需要用context，用application的context (context.getApplicationContext())
3.非静态内部类/匿名类，改成静态内部类
4.匿名内部类 实现多线程，改用静态内部类
5.资源对象未关闭
6.ThreadLocal的value是从current thread连接过来的强引用，需要手动remove

webview 另开进程
adapter复用contentView

内存泄漏与内存溢出的区别

内存泄漏：界面关闭后，activity被其他生命周期更长的资源引用着，导致activity没有被释放
内存溢出：循环创建多个对象。来不及释放。或者 递归

最终是谁持有的activity，handler内存泄露

在Java中，非静态内部类会持有一个外部类的隐式引用，可能会造成外部类无法被GC；

在Handler消息队列 还有未处理的消息 / 正在处理消息时，消息队列中的Message持有Handler实例的引用

Handler持有activity
发送消息到消息队列
Message的target又持有handler

native的内存泄漏怎么监控

高德方案：
内存监控沿用LIBC的malloc_debug模块。不使用官方方式开启该功能，比较麻烦，不利于自动化测试，可以编译一份放到自己的项目中，hook所有内存函数，跳转到malloc_debug的监控函数leak_xxx执行，这样malloc_debug就监控了所有的内存申请/释放，并进行了相应统计。

用get_tls_backtrace实现malloc_debug模块中用到的__LIBC_HIDDEN__ int32_t get_backtrace_external(uintptr_t* frames, size_t max_depth)，刚好同上面说的栈回溯加速方式结合。

建立Socket通信，支持外部程序经由Socket进行数据交换，以便更方便获取内存数据。

搭建Web端，获取到内存数据上传后可以被解析显示，这里要将地址用addr2line进行反解。

编写测试Case，同自动化测试结合。测试开始时通过Socket收集内存信息并存储，测试结束将信息上传至平台解析，并发送评估邮件。碰到有问题的报警，研发同学就可以直接在Web端通过内存曲线及调用栈信息来排查问题了。

如何监控线上OOM，如何知道是哪里造成的OOM

2019 年的

https://blog.csdn.net/AndroidAlvin/article/details/103546677

Probe方案：

堆内存分配失败
Runtime.getRuntime.MaxMemory()获取每个进程被系统分配的内存上限
没有足够大小的连续地址空间

创建线程失败
（1）创建JNI失败
-通过Andorid的匿名共享内存分配 4KB（一个page）内核态内存。
建匿名共享内存时，需要打开/dev/ashmem文件，所以需要一个FD。如果创建的FD数已经达到上限，则会导致创建JNIEnv失败。

-再通过Linux的mmap调用映射到用户态虚拟内存地址空间。
如果进程虚拟内存地址空间耗尽，也会导致创建JNIEnv失败

（2）创建线程失败
-调用mmap分配栈内存。这里mmap flag中指定了MAP_ANONYMOUS，即匿名内存映射。这是在Linux中分配大块内存的常用方式。其分配的是虚拟内存，对应页的物理内存并不会立即分配，而是在用到的时候触发内核的缺页中断，然后中断处理函数再分配物理内存。
分配栈内存失败是由于进程的虚拟内存不足

-调用clone方法进行线程创建。
clone方法失败是因为线程数超出了限制

堆内存不足
开启后台线程每个1S获取当前进程的内存占用，打到阈值时，就去dump，获取内存快照。
（通过Runtime.getRuntime.totalMemory()-Runtime.getRuntime.freeMemory()计算得到内存占用）

分析进程自身OOM
分析进程占用内存与HPROF文件中的Instance数量是正相关的，Instance的数量太大，就会导致OOM。

HPROF文件映射到内存的过程：

```
// 1.构建内存映射的 HprofBuffer 针对大文件的一种快速的读取方式，其原理是将文件流的通道与 ByteBuffer 建立起关联，并只在真正发生读取时才从磁盘读取内容出来。
HprofBuffer buffer = new MemoryMappedFileBuffer(heapDumpFile);
// 2.构造 Hprof 解析器
HprofParser parser = new HprofParser(buffer);
// 3.获取快照
Snapshot snapshot = parser.parse();
// 4.去重 gcRoots
deduplicateGcRoots(snapshot);
```

为了解决这个问题，在HprofParser的解析逻辑中加入了计数压缩逻辑，目的是在文件映射过程去控制Instance的数量。
在解析过程中对于ClassInstance和ArrayInstance，以类型为key进行计数，当同一类型的Instance数量超过阈值时，则不再向Snapshot中添加该类型的Instance，只是记录Intsance被丢弃的数量和Instance大小。
这样就可以控制住每一种类型的Instance数量，减少了分析进程的内存占用。
丢弃了一部分Instance在计算RetainSize时我们会进行计数桶补偿，即把之前丢弃的相同类型的Instance数量和大小都补偿到这个对象上，累积去计算RetainSize。

链路分析时间过长
使用HAHA算法在手机上性能有局限性，如果对所有对象进行链路分析会导致分析耗时非常长。
在生成Reference之后，做了一步链路归并。即同一个对象的不同Instance，如果其底下的引用链路中的对象类型也相同，则进行归并，并记录Instance的个数和每个Instance的RetainSize。然后进行排序。

HAHA算法中基础类型检测不到
遍历堆中的Instance时，如果发现是ArrayInstance，且是byte类型时，将它自身舍弃掉，并将它的RetainSize加在它的父Instance上，然后用父Instance进行后面的排序。

Probe最后会自动分析出RetainSize大小Top N对象到GC Roots的链路，上报给服务器，进行报警。

裁剪回捞HPROF文件
HPROF文件太大，将无用的信息进行裁剪，只保留我们关心的数据。
裁减掉全部基本类型数组的值，原因是我们的使用场景一般是排查内存泄漏以及OOM，只关心对象间的引用关系以及对象大小即可，很多时候对于值并不是很在意，所以裁减掉这部分的内容不会对后续的分析造成影响。

使用GOT表（全局偏移表(Global Offset Table)，用于记录外部调用的入口地址）Hook技术。
在写入文件时，拿到字节流进行裁剪操作，然后把有用的信息写入文件。

方案融合
目前裁剪方案在部分机型上（主要是Android 7.X系统）不起作用。
即通过一次dump操作得到两份HPROF文件，一份原始文件用于下次启动时分析，一份裁剪后的文件用于上传服务器。

线程数超出限制
通过Thread.getAllStackTraces()可以得到进程中的所有线程以及对应的堆栈信息。
用线程池解决。

FD数超出限制
在后台启动一个线程，每隔1s读取一次当前进程创建的FD数量，当检测到FD数量达到阈值时（FD最大限制的95%），读取当前进程的所有FD信息归并后上报。

在/proc/pid/limits描述着Linux系统对对应进程的限制，其中Max open files就代表可创建FD的最大数目。
进程中创建的FD记录在/proc/pid/fd中，通过遍历/proc/pid/fd，可以得到FD的信息。

bitmap的像素数据在哪里？一直申请bitmap会oom吗?

```
   private void getPicturePixel(Bitmap bitmap) {

        int width = bitmap.getWidth();
        int height = bitmap.getHeight();

        // 保存所有的像素的数组，图片宽x高
        int[] pixels = new int[width * height];

        bitmap.getPixels(pixels, 0, width, 0, 0, width, height);

        for (int i = 0; i < pixels.length; i++) {
            int clr = pixels[i];
            int red = (clr & 0x00ff0000) >> 16; // 取高两位
            int green = (clr & 0x0000ff00) >> 8; // 取中两位
            int blue = clr & 0x000000ff; // 取低两位
            Log.d("tag", "r=" + red + ",g=" + green + ",b=" + blue);
        }

    }
```

一直申请OOM是会OOM的，通过缓存策略LruCache，DiskLruCache等解决。

APP性能优化，内存优化，cpu占用率 流畅性等

1.启动优化
- 通过AndoridProfile工具使用查看启动耗时
- 通过StrictMode严苛模式让，查看主线程中是否有io操作，查到让程序崩溃或打印日志
- 启动白屏解决方式，在启动页增加Theme,在onCreate生命周期里去还原主题(super之前)
- idleHandler空闲加载

2.卡顿分析
- Systrace查看帧数，查看每个帧数之间花费的时间
- Looper日志检测卡顿(BlockCanary)
  只要检测 msg.target.dispatchMessage(msg) 的执行时间，就能检测到部分UI线程是否有耗时的操作。
  Looper.getMainLooper().setMessageLogging(logMonitor);

- 线上监控通过演舞者Choreographer.FrameCallback
  通过doFrame回调，可以计算出，上一次的帧数时间-这次的帧数时间，就能计算出掉帧数
  如果>16.6，就说明掉帧了。
  不断发送Choreographer.getInstance().postFrameCallback(this);去执行回调

  Looper比较适合在发布前进行测试或者小范围灰度测试然后定位问题

  ChoreographerHelper适合监控线上环境 的 app 的掉帧情况来计算

3.布局优化
- Layout Inspector查看布局层次
  使用merge和ViewStub
- 开发者选项查看过度渲染
- 异步加载布局
- X2C
    APT技术，解析XML，将View创建出来添加

4.电量优化
Battery Historian工具分析
添加白名单

5.网络优化
利用阿里DNS优化
根据信号强度传递给服务器，让服务器返回对应的图片
protobuf
连接优化http2.0多路复用

6.图片优化
通过inJustDecodeBounds = true 只会加载图片的宽高信息，在通过宽高信息+view的宽高计算出需要压缩的大小，通过inSampleSize 设置。在通过inJustDecodeBounds =false 去加载图片

7.apk优化
图片->svg
Lint移除无用资源
国际化资源配置
so打包配置
压缩代码minifyEnabled true
压缩资源shrinkResources true
代码混淆
资源混淆

8.内存优化
MAT分析工具
LeakCanary分析
常见泄漏场景:
资源对象未关闭
注册对象未注销
类静态变量持有大数据对象
单例造成的内存泄漏->用application的context ，不要用activity的context
非静态内部类的静态实例
Handler临时性内存泄漏
容器中的对象没清理造成的内存泄漏
webView
使用ListView时造成的内存泄漏

dumpsys meminfo查看app内存使用情况

通过硬件加速，然后绘图从CPU转到GPU

如何加载一张30M的大图

BitmapRegionDecoder：区域解码器，可以用来解码一个矩形区域的图像，有了这个我们就可以自定义一块矩形的区域，然后根据手势来移动矩形区域的位置就能慢慢看到整张图片了。

FrameWork

app的启动流程，activity是在哪里创建的，application是在哪里创建的？与AMS是如何交互的

https://www.jianshu.com/p/501690f88f68

app启动：

Activity调用到startActivityForResult->Instrumentation.execStartActivity，
通过ActivityManager.getService获取IBinder，调用到ASM.startActivity
应用进程未启动->startProcessLocked->通过zygote启动
ZygoteInit的main方法内部会创建Server端的Socket，等待AMS请求创建新的应用程序进程
获取到应用进程的启动参数后，fork出应用进程,调用ActivityThread的main方法

应用进程启动->realStartActivityLocked->ApplicationThread通过handler发消息再到ActivityThread去处理启动，
performLaunchActivity 启动，通过Componment存储的类名，调用mInstrumentation.newActivity，用类加载器来创建该Activity的实例，在通过makeApplication创建application，
activity.attach调用Instrumentation来执行activity的onCreate方法
handleResumeActivity resume生命周期

activity是在哪里创建的：
performLaunchActivity 的 mInstrumentation.newActivity，用类加载器来创建该Activity的实例
application是在哪里创建的：
makeApplication是在LoadedApk内调用的，mInstrumentation.newApplication(appContext)

再一起捋一遍App的启动流程：
- Launcher被调用点击事件，转到Instrumentation类的startActivity方法。
- Instrumentation通过跨进程通信告诉AMS要启动应用的需求。
- AMS反馈Launcher，让Launcher进入Paused状态
- Launcher进入Paused状态，AMS转到ZygoteProcess类，并通过socket与Zygote通信，告知Zygote需要新建进程。
- Zygote fork进程，并调用ActivityThread的main方法，也就是app的入口。
- ActivityThread的main方法新建了ActivityThread实例，并新建了Looper实例，开始loop循环。
- 同时ActivityThread也告知AMS，进程创建完毕，开始创建Application，Provider，并调用Applicaiton的attach，onCreate方法。
- 最后就是创建上下文，通过类加载器加载Activity，调用Activity的onCreate方法。

Application和Activity在Context的继承树上有何区别？二者使用上有何不同？

Application和Service继承自ContextWrapper
Activity继承自ContextThemeWrapper，因为Activity有界面，需要主题

使用区别：
1.在单例上使用application的context，生命周期长，activity容易泄露
2.dialog使用activity的context，application操作UI会丢失主题
3.applicaiton启动activity需要指定task和singleTask，因为application没有任务栈

ContentProvider初始化的时机

makeApplication-installContentProviders-callApplicationOnCreate
创建application-创建ContentProvider-attachInfo-onCreate-调用Application的onCreate方法

Handler

handler的post(Runnable)如何实现的。

handler.post->将runnable存入到Message的callback
->sendMessageAtTime->enqueueMessage放入队列，调用nativeWake通知looper取消息

callback，runnable，msg.callback的执行优先级。

callback，runnable，msg的执行优先级：
在dispatchMessage分发消息中，首先判断msg的callback不为空就执行handleCallback，调用message的callback(Runnable)。
如果msg的callback为空，就判断Handler的mCallback，不为空就执行mCallback的handleMessage，为空就执行Handler的handleMessage

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}
```

阻塞是怎么实现的?

epoll机制在Handler中的应用，在主线程的 MessageQueue 没有消息时，便阻塞在 loop 的 queue.next() 中的 nativePollOnce() 方法里，最终调用到epoll_wait()进行阻塞等待。此时主线程会释放 CPU 资源进入休眠状态，直到下个消息到达或者有事务发生，通过往 pipe 管道写端写入数据来唤醒主线程工作。这里采用的 epoll 机制，是一种IO多路复用机制，可以同时监控多个描述符，当某个描述符就绪(读或写就绪)，则立刻通知相应程序进行读或写操作，本质同步I/O，即读写是阻塞的。 所以说，主线程大多数时候都是处于休眠状态，并不会消耗大量CPU资源。

为什么不会阻塞主线程?

Looper通过queue.next()获取消息队列消息，当队列为空，会堵塞，
此时主线程也堵塞在这里，好处是：main函数无法退出，APP不会一启动就结束！

handler 线程切换执行实现原理

当我们在主线程new Handler，会使用主线程的MainLooper，每个线程只有一个Looper，Looper实例化时，创建MessageQueue。
通过post和send发送消息最终会调用到sendMessageAtTime，最终会将Message加入到Looper的MessageQueue中被执行。

handler 延时执行实现原理

Handler post 消息以后，会传入距离开机时间的毫秒数，MessageQueue中会根据when的时长进行一个顺序排序。
获取消息时，会对比当前时间now和when，如果now<when，则通过epoll_wait的 timeout 进行等待
如果该消息需要等待，会进行 idel handlers 的执行，执行完以后会再去检查此消息是否可以执行
====
将我们传入的延迟时间转化成距离开机时间的毫秒数
MessageQueue 中根据上一步转化的时间进行顺序排序
在 MessageQueue.next 获取消息时，对比当前时间（now）和第一步转化的时间（when），如果 now < when，则通过 epoll_wait 的 timeout 进行等待
如果该消息需要等待，会进行 idel handlers 的执行，执行完以后会再去检查此消息是否可以执行

messagequeue 数据结构

messagequeue底层是单链表，
mQuitAllowed 是否允许退出
mIdleHandlers 空闲Handler

sendmessage,sendEmptyMessageDelayed(),postDelayed() 区别

sendEmptyMessageDelayed:发送一个空消息，消息体只有what
sendmessage和sendEmptyMessageDelayed发送出去的消息需要自己重写handlerMessage方法处理。
postDelayed:发送一个runnable消息，runnable会存到Message的callback里不需要重写handlerMessage方法

如果让自己实现一个handler ， 需要怎么实现，有哪些地方需要注意的

Handler
Looper
Message
MessageQueue

通过ThreadLocal确保每个线程只有一个Looper，一个MessageQueue
通过阻塞队列去发消息和取消息
多次初始化looper需要抛出异常

同步屏障原理

加了同步屏障，同步消息就会被屏蔽，让异步消息优先处理。

同步屏障就是发送一个空消息，该Message 的 target 为 null，
Message 的 target为null，通过循环找到下一个异步消息，
prevMsg.next = msg.next 断开指向异步消息的 msg；然后返回这条消息；（相当于优先取出异步消息）

如果同步屏障没有移除，调用 nativePollOnce(ptr, -1);一直阻塞，直到移除 同步屏障 时 调用 nativeWake 来唤醒。

1. 异步消息 要结合 同步屏障使用才有效果，不然就和同步消息是一样的
2. 使用同步屏障 后 能确保异步消息 能优先于 同步消息 执行
3. 使用同步屏障后要删除同步屏障，否则消息队列中的同步消息将得不到执行。

主线程有好多handler，分发消息的时候怎么知道给哪个handler

在enqueueMessage会赋值msg.target = this， this指的就是handler

主线程每5秒钟发一个需要执行10秒的消息到子线程，会发生什么

入队：根据时间排序，当队列满的时候，阻塞，直到用户通过next取出消息。当next方法被调用，通知MessagQueue可以进行消息的入队。

HandlerThread

Thread的子类，创建了Looper。方便使用，线程安全。

内部通过加锁+notifyAll方式解决Looper概率获取不到的问题

IntentServices

特殊services，执行任务在子线程执行，执行完成自动结束。

onCreate中创建HandlerThread，使用它的looper构造handler。
启动service->onStartCommand->发送消息处理->处理结束stopSelf

binder

zygote为什么不采用binder

主要是因为binder是多线程的，fork如果是多线程会有问题，所以采用的socket，而不是binder。

fork做如下事情
1. 父进程的内存数据会原封不动的拷贝到子进程中
2. 子进程在单线程状态下被生成

线程中执行doit，mutex加锁，fork子进程，子进程调用doit发现已经锁住，一直等待(实际没人用)，线程执行完解锁，此时子进程和线程的mutex是两份内存，所以导致死锁。

binder优势。

```
binder优势:
1. 开辟内存
2. 风险隔离-每一个进程，单独的一个app

性能方面，binder 小于 共享内存  优于其他IPC 进程间通信
binder需要拷贝1次，共享内存无需拷贝，socket拷贝2次
线程共享区域比较难操作，binder属于C/S架构，比较好操作
socket和内存共享身份识别为pid，依赖上层协议，不安全
binder身份识是系统分配的UID更安全

1次拷贝原理:
1.发送数据
2.通过系统调用copy_from_user将数据从用户空间拷贝到内核空间
3.通过mmap将内核空间和接收方的用户空间映射在同一块物理内存
4.接收数据

mmap函数原理:
mmap会从当前进程中获取用户态可用的虚拟地址空间,然后分配一块连续的虚拟地址空间,
通过这一步，就能把Binder在内核空间的数据直接通过指针地址映射到用户空间,
当数据从用户空间拷贝到内核空间的时候，是从当前进程的用户空间接拷贝到目标进程的内核空间

内核空间的数据映射到用户空间其实就是添加一个偏移地址
```

binder传输大小限制:

```
binder传输大小限制:
普通的由Zygote孵化而来的用户进程，所映射的Binder内存大小是不到1M的
#define BINDER_VM_SIZE ((1*1024*1024) - (4096 *2))

内核中是4M，不过由于APP中已经限制了不到1M，这里的限制似乎也没多大用途:
ServiceManager进程是128K，只是简单的提供一些addServcie，getService的功能，不涉及多大的数据传输，因此不需要申请多大的内存

超过1M:
FAILED BINDER TRANSACTION
java.lang.RuntimeException: android.os.TransactionTooLargeException:
```

多进程遇到哪些问题:

```
- 所有运行在不同的进程中的四大组件，只要它们之间需要通过内存来共享数据，都会共享失败
- 静态成员和单例模式完全失效（不同进程的内存区域都不一样了）
- 线程同步机制完全失效（不同的进程锁的都不是同一个对象）
- Application会多次创建
- SharedPreferce的可靠性下降（SharedPreferce底层是读写xml文件实现的，系统对它的读写有一定的缓存策略，在内存中会有一份SharedPreferce文件的缓存，所以多个进程并发写操作可能导致数据丢失）
```

aidl生成的java类细节。

asInterface:
绑定服务连接成功后回调方法里获取到服务端的IBinder对象，通过asInterface将服务端的Binder对象转换成客户端所需的AIDL接口类型的对象，如果客户端和服务端位于同一进程，那么此方法返回的就是服务端的Stub对象本身，否则返回的是系统封装后的Stub.proxy对象。

通过DESCRIPTOR(Binder的标识)判断同一进程(obj.queryLocalInterface(DESCRIPTOR))

asBinder
Stub的asBinder
    自己本身this
Proxy的asBinder
    通过asInterface传入的IBinder对象。
binder对象在Stub里就是本进程，在Proxy就是远程代理对象

onTransact(服务端接收)
onTransact方法运行在服务端中的 Binder 线程池中。
客户端发起跨进程请求时，远程请求会通过系统底层封装后交给此方法来处理。
如果此方法返回 false,那么客户端的请求就会失败。
code:确认调用的哪个方法
data:客户端传来的参数
reply:返回值
flag:0-客户端可以传输，服务端可以返回,1-客户端可以传输，服务端不会返回

transact(客户端调用)
transact方法运行在客户端，创建该方法所需要的输入型Parcel对象 _data、输出型Parcel对象 _reply;
接着调用绑定服务传来的 IBinder对象 的 transact方法 来发起远程请求，同时当前线程挂起;
然后服务端的 onTransact方法 会被调用，直到返回，当前线程继续执行，并从 _reply 中取出 RPC 过程的返回结果，也就是返回 _reply 中的数据。

总结:
AIDL 通过 Stub类 用来接收并处理数据，Proxy代理类 用来发送数据，而这两个类也只是通过对 Binder 的处理和调用。

Stub充当服务端角色，持有Binder实体（本地对象）。
获取客户端传过来的数据，根据方法 ID 执行相应操作。
将传过来的数据取出来，调用本地写好的对应方法。
将需要回传的数据写入 reply 流，传回客户端。

Proxy代理类充当客户端角色，持有Binder引用（句柄）。
生成 _data 和 _reply 数据流，并向 _data 中存入客户端的数据。
通过 transact() 方法将它们传递给服务端，并请求服务端调用指定方法。
接收 _reply 数据流，并从中取出服务端传回来的数据。

```
private ServiceConnection connection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        iLeoAidl = ILeoAidl.Stub.asInterface(service);
    }
    @Override
    public void onServiceDisconnected(ComponentName name) {
        iLeoAidl = null;
    }
};
```

```
public interface ILeoAidl extends IInterface {

    public static abstract class Stub extends Binder implements ILeoAidl {
        private static final String DESCRIPTOR = "ILeoAidl";

        public Stub() {
            this.attachInterface(this, DESCRIPTOR);
        }

        public static ILeoAidl asInterface(IBinder obj) {
            if ((obj == null)) {
                return null;
            }
            IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
            if (((iin != null) && (iin instanceof ILeoAidl))) {
                return ((ILeoAidl) iin);
            }
            return new Proxy(obj);
        }

        @Override
        public IBinder asBinder() {
            return this;
        }

        @Override
        public boolean onTransact(int code, Parcel data, Parcel reply, int
flags) throws RemoteException {
            switch (code) {
                case INTERFACE_TRANSACTION: {
                    reply.writeString(DESCRIPTOR);
                    return true;
                }
                case TRANSACTION_addPerson: {
                    data.enforceInterface(DESCRIPTOR);
                    Person _arg0;
                    if ((0 != data.readInt())) {
                        _arg0 = Person.CREATOR.createFromParcel(data);
                    } else {
                        _arg0 = null;
                    }
                    this.addPerson(_arg0);
                    reply.writeNoException();
                    return true;
                }
                case TRANSACTION_getPersonList: {
                    data.enforceInterface(DESCRIPTOR);
                    List<Person> _result = this.getPersonList();
                    reply.writeNoException();
                    reply.writeTypedList(_result);
                    return true;
                }
            }
            return super.onTransact(code, data, reply, flags);
        }

        private static class Proxy implements ILeoAidl {
```

```
            private IBinder mRemote;

            Proxy(IBinder remote) {
                mRemote = remote;
            }

            @Override
            public IBinder asBinder() {
                return mRemote;
            }

            public String getInterfaceDescriptor() {
                return DESCRIPTOR;
            }

            @Override
            public void addPerson(Person person) throws RemoteException {
                Parcel _data = Parcel.obtain();
                Parcel _reply = Parcel.obtain();
                try {
                    _data.writeInterfaceToken(DESCRIPTOR);
                    if ((person != null)) {
                        _data.writeInt(1);
                        person.writeToParcel(_data, 0);
                    } else {
                        _data.writeInt(0);
                    }
                    mRemote.transact(Stub.TRANSACTION_addPerson, _data, _reply,
0);
                    _reply.readException();
                } finally {
                    _reply.recycle();
                    _data.recycle();
                }
            }

            @Override
            public List<Person> getPersonList() throws RemoteException {
                Parcel _data = Parcel.obtain();
                Parcel _reply = Parcel.obtain();
                List<Person> _result;
                try {
                    _data.writeInterfaceToken(DESCRIPTOR);
                    mRemote.transact(Stub.TRANSACTION_getPersonList, _data,
_reply, 0);
                    _reply.readException();
                    _result = _reply.createTypedArrayList(Person.CREATOR);
                } finally {
                    _reply.recycle();
                    _data.recycle();
                }
                return _result;
            }
        }

        static final int TRANSACTION_addPerson = (IBinder.FIRST_CALL_TRANSACTION
+ 0);
```

```
        static final int TRANSACTION_getPersonList =
(IBinder.FIRST_CALL_TRANSACTION + 1);
    }

    public void addPerson(Person person) throws RemoteException;

    public List<Person> getPersonList() throws RemoteException;
}
```

实现了AIDL接口的服务的方法是运行在哪个线程？有多个方法，是不是运行在同一个线程？

绑定服务回调是在主线程。

服务端方法运行是在binder线程。测试下来运行在不同线程。
每一个事件就是一个消息，然后处理消息的时候会排队用线程去处理。

Binder通讯中，客户端调用服务端是阻塞的吗？怎么做到不阻塞？

oneway 主要有两个特性：异步调用和串行化处理。异步调用是指应用向 binder 驱动发送数据后不需要挂起线程等待 binder 驱动的回复，而是直接结束。像一些系统服务调用应用进程的时候就会使用 oneway，比如 AMS 调用应用进程启动 Activity，这样就算应用进程中做了耗时的任务，也不会阻塞系统服务的运行。

串行化处理是指对于一个服务端的 AIDL 接口而言，所有的 oneway 方法不会同时执行，binder 驱动会将他们串行化处理，排队一个一个调用。

涉及到的 binder 命令也有规律，由外部发送给 binder 驱动的都是 BC_ 开头，由 binder 驱动发往外部的都是 BR_开头。
客户端线程挂起相当于 Thread 的 sleep，是真正的"休眠"，底层调用的是
waiteventinterruptible() Linux 系统函数。
waiteventinterruptible函数也在Looper阻塞中用到，是调用了 nativePollOnce() 方法

非oneway:
client发送BC_TRANSACTION将数据写入out，
binder驱动回复BR_TRANSACTION_COMPLETE表示驱动已经收到客户端的请求，客户端休眠。
binder驱动发送BR_TRANSACTION给服务端，读取客户端参数，
服务端发送BC_REPLY向驱动写返回值，
驱动发送BR_TRANSACTION_COMPLETE给服务端表示已经收到，
驱动发送BR_REPLY表示客户端收到服务端的返回结果

oneway:
client发送BC_TRANSACTION将数据写入out
binder驱动回复BR_TRANSACTION_COMPLETE表示驱动已经收到客户端的请求
binder驱动发送BR_TRANSACTION给服务端处理

A进程调用B进程的b方法，B进程没有创建的情况下，B进程中是b方法先执行还是Application的onCreate方法先执行？如果b方法很耗时，A进程会被阻塞住吗

B进程没有创建。绑定服务不会成功的。
会阻塞

SharedPreferences

SharedPreferences线程安全吗

SharedPreferences线程安全
读写操作都会去加锁

getSharedPreferences：缓存未命中，才构造SharedPreferences对象，线程安全的
SharedPreferencesImpl：创建备份文件，用户写入失败时进行恢复工作，startLoadFromDisk加载数据
startLoadFromDisk：有备份文件直接回滚，第一次getSharedPreferences开启线程调用此方法进行异步读取的，
解析得到的键值对数据保存在mMap中，notifyAll通知唤醒其他等待线程，数据已经加载完毕
getString：线程安全的，直接操作内存mMap，可能会阻塞awaiteLoadedLocked直到loadFromDisk加载完成。
putXxx：线程安全，通过EditorImpl 对键值对数据的增删记录保存在mModified（map）中，会在commit/apply方法中同步内存

apply() 提交是异步的，在 IO线程完成的；
commit() 提交是同步的， 在当前线程完成的， 可能会出现阻塞主线程的情况；
将mModified同步到mMap，enqueueDiskWrite将数据写入到磁盘上，等待完成（阻塞等待），通知监听者返回结果。

QueuedWork ：apply异步提交数据后会被加入到 QueuedWork.enque(postrunnable) 进行处理
ActivityThread的handlePauseActivity会执行QueuedWork.waitToFinish();使得所有异步的操作都完成，完成后会执行onPause。
这时突然杀死 APP， QueuedWork 中未完全执行，发生ANR

SharedPreferences多进程调用会有问题吗？如果需要多进程调用，怎么实现

不支持多进程，MMKV
MODE_MULTI_PROCESS是在每次getSharedPreferences时检查磁盘上配置文件上次修改时间和文件大小，一旦所有修改则会重新从磁盘加载文件，所以并不能保证多进程数据的实时同步

MMKV中使用flock文件锁来完成多进程操作文件的同步,实现同一时间只有一个进程在操作持久化文件。

如果存在AB进程，在B进程修改完成之后，A进程如何知道B进程的修改？
在MMKV中会生成一份与数据文件同名的.crc文件，记录两个关键数据：数据内容的crc校验码 与 单调递增的序列号。

CRC校验码：循环冗余校验 ，类似文件MD5值。
递增的序列号：每次去重、扩容即执行全量更新，序列号+1并记录在crc文件中，不匹配则需要重新解析全部文件。
如果不同就重新读取

如何解决sp造成的界面卡顿、掉帧问题?

1.初始化sp放在application；
2.不要频繁的commit/apply,尽量使用一次事物提交；
3.优先选择用apply而不是commit，因为commit会卡UI；
4.sp是轻量级的存储工具，所以请你不要存放太大的数据，不要存json等；
5.单个sp文件不要太大，如果数据量很大，请把关联性比较大的，高频操作的放在单独的sp文件

Kotlin

kt的伴生对象是饿汉模式还是懒汉模式?

伴生对象是懒汉
object是饿汉

kotlin和java混用有哪些问题?

使用kotlin调用一个会返回空的java函数， 是不会提示空安全的，只有加上@nullable之后才会提示。

kotlin懒加载

lateinit
lateinit只能修饰变量var，不能修饰常量val
lateinit不能对可空类型使用
lateinit不能对java基本类型使用
编译成java->调用如果等于空会抛出异常

lazy
使用时，在类型后面加by lazy{}即可，{}中的最后一行代码，需要返回初始化的结果
lazy只能对常量val使用，不能修饰变量var
lazy的加载时机为第一次调用常量的时候，且只会加载一次（毕竟是个常量，只能赋值一次）

companion object:
编译成java会生成静态内部类，通过静态内部类来调用相关方法

@JvmStatic

在伴生对象中使用：可以在java中直接调用
未使用：需要在java中调用Companion类来调用

在object中使用：可以在java中直接调用
未使用：需要在java中调用INSTANCE类来调用

NDK

c++中，构造函数的调用顺序，析构函数是否需要virtual

https://blog.csdn.net/daheiantian/article/details/6438782
整体顺序**：虚基类 --> 直接父类 -->自己

D d;

虚基类的父类构造(每个虚基类都要执行父类的构造)
虚基类的构造
直接父类的构造
自己的构造

定义复制构造:
D d1(d);
虚基类的父类构造(每个虚基类都要执行父类的构造)
虚基类的构造
直接父类的构造
自己的复制构造

没定义复制构造:
虚基类的父类构造(每个虚基类都要执行父类的构造)
虚基类的复制
直接父类的复制
自己的复制

d = d1;
因为显式调用了赋值操作符，那么就只调用自己的代码，不会隐式调用其它的函数
析构函数的执行顺序与 构造函数 相反。

析构函数是否需要virtual：
析构建议为虚函数。
在父类指向子类的指针析构时， 会调用子类的析构.如不加virtual,则不会调用子类的析构.

构造函数不能是虚函数。

JNI的注册方法有哪些

静态注册
Java_PACKAGENAME_CLASSNAME_METHODNAME

动态注册
创建动态注册的数组:名称+方法签名+ C++函数
定义JNI_OnLoad函数
通过javaVM 获得JniEnv
通过JniEnv和类名找到需要动态注册的Class
通过JniEnv调用RegisterNatives将Class和数组传进去进行注册

```
static const JNINativeMethod mMethods[] ={
    {"dynamicNative","()V", (void *)dynamicNative1}
}
//需要动态注册native方法的类名
static const char* mClassName = "com/dongnao/jnitest/MainActivity";
jint JNI_OnLoad(JavaVM* vm, void* reserved){
    JNIEnv* env = NULL;
    //获得 JniEnv
    int r = vm->GetEnv((void**) &env, JNI_VERSION_1_4);
    if( r != JNI_OK){ return -1; }
    jclass mainActivityCls = env->FindClass( mClassName);
    // 注册 如果小于0则注册失败
    r = env->RegisterNatives(mainActivityCls,mMethods,2);
    if(r != JNI_OK ) { return -1; }
    return JNI_VERSION_1_4;
```

```
}
```

函数指针如何写?

```
函数返回值类型 (* 指针变量名) (函数参数列表);

int Func(int x);   /*声明一个函数*/
int (*p) (int x);  /*定义一个函数指针*/
p = Func;          /*将Func函数的首地址赋给指针变量p*/
```

设计模式

代理模式

动态代理传入的参数都有哪些？非接口的类能实现动态代理吗?

```
Proxy.newProxyInstance(ClassLoader loader, Class<?>[]
interfaces,InvocationHandler invocationHandler)

由于java的单继承，动态生成的代理类已经继承了Proxy类的，就不能再继承其他的类，所以只能靠实现被代理类的接口的形式，故JDK的动态代理必须有接口。
```

cglib动态代理

```
java动态代理只能对接口代理，如果要代理普通类就要用cglib

Enhancer既能够代理普通的class，也能够代理接口。
Enhancer不能够拦截final方法

enhancer.setCallback(new FixedValue())
FixedValue用来对所有拦截的方法返回相同的值

cglib底层采用ASM字节码生成框架，使用字节码技术生成代理类，也就是生成的.class文件，而我们在android中加载的是优化后的.dex文件，也就是说我们需要可以动态生成.dex文件代理类，cglib在android中是不能使用的。
```

```
public class Client {
    public static void main(String[] args) {
        // 代理类class文件存入本地磁盘方便我们反编译查看源码
        System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY,
"D:\\code");
        // 通过CGLIB动态代理获取代理对象的过程
        Enhancer enhancer = new Enhancer();
        // 设置enhancer对象的父类
        enhancer.setSuperclass(HelloService.class);
        // 设置enhancer的回调对象
        enhancer.setCallback(new MyMethodInterceptor());
```

```
        // 创建代理对象
        HelloService proxy= (HelloService)enhancer.create();
        // 通过代理对象调用目标方法
        proxy.sayHello();
    }
}
```

适配器和装饰模式各自特点和使用场景

适配器:
需要一个统一的输出接口，而输入端的接口不可预知。

ListView通过引入Adapter适配器类把那些多变的布局和数据交给用户处理，然后通过适配器中的接口获取需要的数据来完成自己的功能，从而达到了很好的灵活性。这里面最重要的接口莫过于getView()接口了，该接口返回一个View对象，而千变万化的UI视图都是View的子类，通过这样一种处理就将子View的变化隔离了，保证了AbsListView类族的高度可定制化。
当然这里的Adapter并不是经典的适配器模式，却是对象适配器模式的优秀示例.

总结
适配器模式的经典实现在于把原本不兼容的接口融合在了一起，使之能更好的合作。但在实际开发中也可以有一些灵活的实现，比如ListView。
当然过多的使用适配器会让系统显得过于凌乱。如果不是很有必要，可以不适用适配器而是直接对系统进行重构

装饰: 扩展一个对象的功能，可以使一个对象变得越来越强大。
File, ListView/RecyclerView的addHead ，addFoot

装饰模式和代理模式的区别

都是当前对象无法胜任主功能，转给第三方对象来完成主功能。
装饰目标是增强自身（第三方是我内部），代理目标是找经纪人/甩锅（外部）。

数据库

数据库有哪些，android sqlite存储数据的上限大小是多少?

https://www.sqlite.org/limits.html

百度
支持数据库大小至2TB

官网
SQLIte数据库最大支持281 tb
字符串 默认值是10亿

业务

大文件在传输过程中要考虑哪些问题，如何保证大文件的一致性

- 大文件传输，应该支持断点续传；
- 要有文件校验，校验不对的话能自动重传；(simhash？ md5?)
- 要考虑多线程分片上传，并发控制，带宽压力，限速上传；
- 多文件排队上传；
- 中转临时文件的删除机制；

如何做到单个信号源，多个页面响应

EventBus，观察者模式

怎么实现多线程下载，断点续传怎么实现。

a、对于网络上的一个资源，首先发送一个请求，从返回的Content-Length中回去需要下载文件的大小，然后根据文件大小创建一个文件。
b、根据线程数和文件大小，为每个线程分配下载的字节区间，然后每个线程向服务器发送请求，获取这段字节区间的文件内容。
c、利用RandomAccessFile的seek方法，多线程同时往一个文件中写入字节。

是否支持范围请求:Access-Ranges:bytes
使用范围请求:Content-Range
响应状态码为 206 Partial Content：成功，包含请求的数据区间

资源变化:在 If-Range 这个请求报文头中使用 ETag 或者 Last-Modified 两个参数任意一个
ETag：当前文件的一个验证令牌指纹，用于标识文件的唯一性。
Last-Modified：标记当前文件最后被修改的时间。
同一个资源文件-返回206，如果不是返回200，要从头下载

如果range填入范围错误，返回416请求的数据区间不在文件范围之内

设计一个日志系统/设计一个线上日志收集系统

卡顿，影响性能：
通过native来实现日志逻辑，来解决javaGC问题

日志丢失：
异常退出丢失日志，通过引入mmap机制
mmap使用逻辑内存对磁盘文件进行映射，操作内存相当于操作文件，
异常退出，系统自动写回文件

安全性
使用AES(对称)进行日志加密确保日志安全性。

日志分散
自定义日志协议，进行格式化处理，实现本地聚合

接收日志
主动上报可以通过客服引导用户上报，也可以进行预埋，在特定行为发生时进行上报（例如用户投诉）。

回捞上报是由后端向客户端发起回捞指令。

为了节约用户手机空间大小，日志文件只保留最近7天的日志

native奔溃的日志采集，怎么处理?

BreakPad 是 Google 开发的跨平台崩溃捕获方案，目前主要用于 Chromium。安卓 APP 也可以使用 BreakPad 来捕获异常。

Crash发生：
在 C 层，CPU 通过异常中断的方式，触发异常处理流程。不同的处理器，有不同的异常中断类型和中断处理方式，linux 把这些中断处理，统一为信号量，每一种异常都有一个对应的信号，可以注册回调函数进行处理需要关注的信号量。
发生crash函数：sigaction
对每个信号量订阅，调用sigaction

捕获Crash位置：
sigaction有一个sa_sigaction变量为函数指针，是crash位置，
给变量赋值，就可以回调我们自己的函数

设置紧急栈空间：
由于无限递归造成堆栈溢出，所以使用sigaltstack设置一个紧急处理的新栈。

捕获问题代码：
sa_sigaction赋值的回调函数中，第三个参数context是 cpu 相关的上下文
获取到崩溃时的pc，就能知道崩溃时执行的是那条指令(兼容其他平台)

pc值转内存地址：
pc值是程序加载到内存中的绝对地址，
通过dladdr()获得共享库加载到内存的起始地址,和pc值相减就可以获得相对偏移地址。
在使用addr2line 分析出是哪一行代码。

获取 Crash 发生时的函数调用栈：
第一种：直接使用系统的<unwind.h>库，可以获取到出错文件与函数名。只不过需要自己解析函数符号，同时经常会捕获到系统错误，需要手动过滤。
第二种：在4.1.1以上，5.0以下，使用系统自带的libcorkscrew.so，5.0开始，系统中没有了libcorkscrew.so，可以自己编译系统源码中的libunwind。libunwind是一个开源库，事实上高版本的安卓源码中就使用了他的优化版替换libcorkscrew。
第三种：使用开源库coffeecatch，但是这种方案也不能百分之百兼容所有机型。
第四种：使用 Google 的breakpad，这是所有 C/C++堆栈获取的权威方案，基本上业界都是基于这个库来做的。只不过这个库是全平台的 android、iOS、Windows、Linux、MacOS 全都有，所以非常大，在使用的时候得把无关的平台剥离掉减小体积。

unwind.h库：
库中_Unwind_Backtrace函数可以传入函数指针作为回调，
指针指向其中一个参数_Unwind_Context类型结构体。
使用_Unwind_GetIP()将当前函数调用栈中每个函数的绝对内存地址写入到_Unwind_Context结构体中，
最终返回的是当前调用栈的全部函数地址了，在转成相对地址。
通过Dl_info对象，我们就能获取到全部想要的信息了。(需要手动过滤掉各种系统错误)

数据回传到服务器：
写入文件，下次启动的时候直接由 Java 上报。
回调 Java 代码

App不崩溃方法

子线程异常处理：
Thread.setDefaultUncaughtExceptionHandler->uncaughtException回调
一个应用中所使用的线程都是在同一个线程组，线程组中的线程出现异常会获取线程组中的系统默认的异常处理器，来处理。

新进程启动会调用zygoteInit方法，设置异常处理器KillApplicationHandler。
平时看到的崩溃弹窗就是handleApplicationCrash方法中弹出的。

所以我们设置自定义处理器把原处理器覆盖掉就不会崩溃了。

主线程异常处理：
通过Handler往主线程发送了一个runnable任务，然后在这个runnable中加了一个死循环，死循环中执行了Looper.loop()进行消息循环读取。
这样就会导致后续所有的主线程消息都会走到我们这个loop方法中进行处理，也就是一旦发生了主线程崩溃，那么这里就可以进行异常捕获。同时因为我们写的是while死循环，那么捕获异常后，又会开始新的Looper.loop()方法执行。这样主线程的Looper就可以一直正常读取消息，主线程就可以一直正常运行了。

生命周期异常处理：
反射调用ActivityThread的H类,设置Callback，生命周期调用时会执行到callback，根据msg.what，try...catch,
catch住后杀进程或者activity

杀死activity:
通过msg.obj获取到ClientTransaction，获取到token、activityManager，执行finishActivity。
finishActivityMethod.invoke(activityManager, binder, Activity.RESULT_CANCELED, null, 0);

注解实现一个提示功能：如果int的值大于了3需要提示。

编译器无法获取到。field级别的apt处理器拿不到值的，注解是基于编译期的，参数则是在运行时才会有值，apt处理注解的时候，参数的值可能都还没初始化 获取这个参数的话，只能反射获取 需要用运行时的方式获取，比如反射获取field及对应的注解信息

图片缓存怎么做？你要设计一个图片缓存框架怎么搞？

强引用->软引用->硬盘缓存

当我们的APP中想要加载某张图片时，先去LruCache中寻找图片，如果LruCache中有，则直接取出来使用，如果LruCache中没有，则去SoftReference中寻找（软引用适合当cache，当内存吃紧的时候才会被回收。而weakReference在每次system.gc（）就会被回收）（当LruCache存储紧张时，会把最近最少使用的数据放到SoftReference中），如果SoftReference中有，则从SoftReference中取出图片使用，同时将图片重新放回到LruCache中，如果SoftReference中也没有图片，则去硬盘缓存中中寻找，如果有则取出来使用，同时将图片添加到LruCache中，如果没有，则连接网络从网上下载图片。图片下载完成后，将图片保存到硬盘缓存中，然后放到LruCache中。

Glide:
获取图片，优先从Lruche获取图片，获取到后从Lruche中清除，放到activeResources中,activeResources map是盛放正在使用的资源，以弱引用的形式存在。同时资源内部有被引用的记录。
如果资源没有引用记录了，那么再放回Lruche中，同时从activeResources中清除。
如果Lruche中没有，就从activeResources中找，找到后相应资源的引用加1。
如果Lruche和activeResources中没有，那么进行资源异步请求（网络/diskLrucache），请求成功后，资源放到diskLrucache和activeResources中。

让你设计一个打点系统怎么做?

1.你们的app主要类型是什么，你们觉得哪部分数据是需要埋点获取的，其次你们埋点上传的频率是多少
2.常见的埋点，一般分为这几类数据：
设备基础属性、APP基础属性、用户基础属性、页面元素、页面关联逻辑、点击事件等行为描述、Storage模块数据(缓存)
3.埋点的实现方式：
部分数据可以考虑无埋点，比如
1.设备、APP、用户等基础属性，直接通过api获取
2.Activity进入/离开等生命周期相关属性，直接通过LifeCycleCallback监听获取
3.Activity的唯一标记如pageId等属性，直接通过注解获取
4.UI元素点击/滑动等行为属性，需要通过hook代码才能实现
这部分数据通过代码完成自动搜集，第一次打开或者每次打开上传即可
需要使用埋点的数据可以考虑使用AOP + Hook +注解完成

怎么给所有的点击事件全局埋点，hook 系统的click事件

registerActivityLifecycleCallbacks监听resume生命周期
通过activity.findViewById找到contentView
反射获取view的mOnclickListener（View-getListenerInfo-ListenerInfo-mOnClickListener）
通过代理方式设置view的OnClickListener view.setOnClickListener(new WrapOnClickListener(listener))
递归子view

有做过推送相关吗？原理有去了解过吗?

http://zhangtielei.com/posts/blog-android-push.html

端内推送：当App在前台运行的时候，
端内推送一般是走App自己实现的一套推送系统：推送服务器是自己的，客户端维护一条长连接连到自己的推送服务器，不依赖任何第三方的推送系统。
端外推送：当App进程被杀了，长连接断开。这时的推送就称为端外推送了，只能走某个第三方推送平台了。

端内推送涉及内容：

- 采用什么协议？XMPP还是MQTT还是自定义二进制协议？是否像微信一样，需要推送二进制数据（比如短语音和缩略图数据）？
- 如何保证后台长连接不死？涉及到“保活”的问题。
- 如何做才能真正保证不丢数据？涉及到系统的方方面面，比如消息的确认，客户端和服务器的数据同步，客户端的数据存储的事务保证，后台消息队列如何设计保证不丢数据。如果是IM，离线数据如何处理？
- 长连接的Keep Alive和连接状态的检测与维护。比如XMPP相当于一个永远解析不完的XML流，使用一个空格作为Keep Alive消息。
- 长连接的安全性。验证以及加密。

推送平台：
- 小米推送（MiPush）
- 华为推送（华为Push）
- 友盟推送（U-Push）
- 个推
- 极光推送
- 阿里云移动推送（Alibaba Cloud Channel Service）
- 腾讯信鸽推送
- 百度云推送

选择推送标准：
- 端内使用自己的推送；
- 端外在小米手机上使用小米推送；
- 端外在华为手机上使用华为推送；
- 端外在其它手机上统一使用一种推送，也就是推送平台列表中的某一个。

第三方推送，手机内如果有多个APP都是用A推送，只要有1个app或者就能拉起其他进程的app。选择使用率较高的app推送很重要。

通知栏消息和透传消息：
通知栏消息，在被送达用户的设备后，直接以系统通知的形式展示给用户。它不会继续被传递到App。
而透传消息，在被送达用户的设备后，还会继续路由到App，通过回调App的某个BroadcastReceiver的形式将消息传递到App内部。然后由App决定如何处理和显示这个消息。

推送到达率？
“在线送达率”，各个推送平台优化到最后，可能都差不多。估计都能达到98%以上。
“通用送达率”才是真真正正把消息推送到你的App的最终的送达率，这个也才是用户最终能感受到的送达率。App开发者需要真正关注的也是这个。

推送原理：
目前大部分的第三方推送服务，都是基于长连接的推送方案，下面将对这个方式进行详细讲解。

NAT
首先，我们需要了解下一个网络基本知识——NAT，即网络地址转换（Network Address Translation，NAT），这是因为IP地址是有限的，手机无论是通过路由器还是数据网络，都有一个内网IP地址，同时，路由器上会维护一个外网IP地址，从而形成一个NAT路由表，即内网IP地址:端口，以及对应的外网IP地址:端口。这样通过一层层封装与解封装，就达到了内网与外网交换通信的方式。

NAT超时
由于NAT路由表的大小有效，所以一般路由都有NAT有效期，WIFI下，这个NAT有效期可能会比较长，而在数据流量下，运营商一般都会尽快更新NAT路由表，淘汰无效的设备，所以，在使用数据流量时，长连接经常容易断。
那么除了NAT路由表主动淘汰过期的设备之外，切换网络环境和DHCP服务器租期到期，这些情况都有可能导致NAT路由表改变，从而造成长连接中断。

心跳包
前面我们说了，现在的推送服务一般采用的是长连接的通信方式，而长连接会因为NAT路由表的更新而中断，所以，客户端会定时向服务端发送一个心跳包，来定期告知NAT路由表，我还活着，别杀我！这就是心跳包的作用——防止NAT路由表超时，同时检测连接是否被断开。

心跳包的心跳时间
　　既然心跳包的作用是防止NAT超时，那么就需要将心跳包的发送频率设置为小余NAT超时的检测频率，而WIFI和数据流量下，对于NAT路由表的超时时间又是不一样的，而且不同的网络运营商的超时时间，甚至都不一样，所以，一个比较好的方法就是根据设备当前网络环境，来动态的设置心跳时间。

注意，心跳包与轮询是不一样的，心跳包建立在长连接上，只要发送数据即可，而轮询每次都是一个完整的TCP连接。

心跳包谁来发
　　既然需要定时任务，那么就需要使用AlarmManager来作定时唤醒了，原因我之前的文章有讲过，是关于处理器唤醒的原因，这里就不赘述了，大家可以参考我之前的文章：

Android中的睡与不睡

进程保活
　　所谓进程保活，是指App希望尽可能的保证自己的App的推送进程能够存活在后台，以保证可以收到服务端的推送消息，因此，才出现了一大批关于进程保活的方式，例如NDK层的文件锁，fork子进程、前台服务、进程优先级等等方式，然而，这些东西，实际上，都不能完全保证手机的进程管理策略放过你，特别是Android 5.0以后的系统，Android对进程的管理更加严格，还有国内的这些ROM层的修改，ROM想要杀你这个进程，你怎么做也没有办法，哦，除了白名单。所以，不要再花心思去找什么进程保活的黑科技了，好好做好应用，提供用户的使用黏性，才是最佳的保活，而对于一些产品、运营所谓的『为什么微信、QQ都可以保活』这样的问题，我建议你回答它：『如果你能把产品做到微信、QQ那样的数量级，我也能让你活！』

app怎么保活

https://github.com/xxchenqi/DaemonApp

如果有一个任务需要线程池里所有任务执行完以后执行，应该怎么做

```
CountDownLatch

Callable

boolean loop = true;
do{
    loop=!executorService.awaitTermination(2, TimeUnit.SECONDS);
}while(loop);
```

有没有做过安全性能检测的项目

混淆
WebView明文存储密码带来的安全漏洞：
WebView.getSettings().setSavePassword(false)，显示调用API设置为false，让WebView不存储密码

动态注册Receiver风险：
registerReceiver()方法注册的BroadcastReceiver是全局的并且默认可导出的，如果没有限制访问权限，可以被任意外部APP访问，向其传递Intent来执行特定的功能。因此，动态注册的BroadcastReceive可能会导致拒绝服务攻击、APP数据泄漏或是越权调用等安全风险
解决
静态注册，exported="false"

必须动态注册 BroadcastReceiver时，使用registerReceiver(BroadcastReceiver,IntentFilter,broadcastPermission,android.os.Handle)函数注册
Android8.0新特性想要支持静态广播、需要添加intent.setComponent(new ComponentName（）)

公共组件配置风险：设置android:exported=false，自定义权限

数据越权备份风险：攻击者可通过adb backup和adb restore对APP的应用数据进行备份和恢复
allowBackup属性值设置为false来关闭应用程序的备份和恢复功能；或者加固

秘钥及敏感信息：使用JNI将敏感信息写到Native层

截屏攻击风险:getWindow().setFlags(LayoutParams.FLAG_SECURE,
LayoutParams.FLAG_SECURE);

WebView远程代码执行漏洞:应用不能在4.2以下系统上运行,调用的方法必须以@JavascriptInterface进行注解声明

本地数据安全加密用什么方式

https://www.jianshu.com/p/d4fee3a2de82

https://blog.csdn.net/zr940326/article/details/51549310

Base64

没用HTTPS的话，为了确保传输安全，还需对传输的数据进行加密，这里我推荐用AES+RSA进行加密

so

分段存放，C层（so文件）+String文件（string.xml）+gradle文件；也可以从服务获取

蓝牙流程:

获取蓝牙服务
BluetoothManager bluetoothManger = (BluetoothManager)
getActivity().getSystemService(Context.BLUETOOTH_SERVICE);
获取蓝牙适配器
BluetoothAdapter bluetoothAdapter = bluetoothManger.getAdapter();
扫描蓝牙设备
bluetoothAdapter.startLeScan(callback);
停止扫描
bluetoothAdapter.stopLeScan(callback);
回调callback：
回调内获取到蓝牙的名称和地址

连接蓝牙设备：
(1)获取蓝牙设备
BluetoothDevice device = bluetoothAdapter.getRemoteDevice(address);
(2)连接蓝牙设备
BluetoothGatt mBluetoothGatt = device.connectGatt(this, false, mGattCallback);
(3)mGattCallback
onConnectionStateChange:连接状态改变
onServicesDiscovered:发现服务，在蓝牙连接的时候会调用
    BluetoothGatt获取服务进行遍历
    找到和服务中相等的UUID

```
    获取服务的特征
    找到和特征相等的UUID
    setCharacteristicNotification启用给定的特征通知
    获取描述符：BluetoothGattDescriptor descriptor =
gattCharacteristic.getDescriptor(UUID_CCD);
    descriptor.setValue启用通知

    广播通知->发送handler；
    写入数据bluetoothService.WriteValue->触发onCharacteristicChanged

onCharacteristicRead:数据接收
onCharacteristicChanged:Characteristic 改变，数据接收会调用
    蓝牙数据返回处理
    获取密码->写数据解锁，返回处理

onCharacteristicWrite:发送数据后的回调
onDescriptorRead:descriptor读
onDescriptorWrite:descriptor写
onReadRemoteRssi:读Rssi
onReliableWriteCompleted



关闭蓝牙设备:
mBluetoothGatt.disconnect();
mBluetoothGatt.close();
```

git暂存代码

我在 feature_666 分支,非常聚精会神加持高专注地实现一个功能 666 模块，客户反馈出一个 bug ,然后这时， 非常严重，必须立马解决
使用 git stash ，将当前修改(未提交的代码)存入缓存区，切换分支修改 bug ,回来再通过 git stash pop 取出来。

广告

```
打开广告页面
调用广告SDK展示需要的广告
广告SDK获取手机设备的信息发送广告请求获取广告信息
解析广告返回结果，展示广告
展示成功，发送展示请求给服务器记录广告展示的追踪链接
点击广告，请求广告的追踪链接到服务器记录点击日志
处理页面跳转，执行文件下载，请求下载成功追踪链接，记录下载成功日志
```

未知:

如何设计一个类似于微信朋友圈的首页的功能，UI 数据等方面

如何设计一个无限数据的气泡显示聊天内容

5个G数据，如何在500M内存的情况下实现排序

项目中的亮点有哪些

kotlin的协程，怎么做到和rxjava的zip操作一样，等待所有结果后再处理

介绍下flutter的启动流程

介绍下flutter与weex的区别

介绍下flutter_boost的原理

视频编解码是怎么做的

手淘这种大型app是怎么迭代起来的?

你对小程序的原理了解么?

v8binding怎么做?

你怎么设计埋点系统一个线程模型?

如何实现一个快速排序的稳定性?

设计一个数据结构，微博里面有人发了文章怎么实现？关注怎么实现?

微博里面有人发了文章，关注的人如何获取最新的10条？如何获取关注的人的文章的最新1000条？设计一个高效的算法

Glide：缓存怎么做的， 怎么和页面生命周期绑定（低版本是fragment，新版本 lifecycle？）

一个巨大无序数组，查第一个不连续自然数的节点， 例如 1、2、3、5、6、7.... 第一个不连续自然数的节点是 4。

给定一个二进制数据位数， 输出所有2进制数所对应的所有自然数，要求时间复杂度最优： 例如 输入1， 输出 0， 1， 输入2， 输出 0，1，2，3，4，5，6，7

做过哪些性能优化，优化成果怎么样，是否有数据支撑，数据来源怎么取。

一个送礼的场景， 礼物有权重属性，怎么根据权重进行对礼物进行处理，然后再排队纷发，每次取一个礼物，怎么设计数据结构： 用有序队列，权重最大的排在队列最前，每次取礼物只拿第一个就行。细节还有数据重排，队列维护，数据同步等

音视频采集编码播放流程

Mediacodec编码内部原理与工作流程（什么状态机之类的），使用Mediacodec时遇到过什么问题，怎么解决的

ffmpeg怎样编码和解码的，怎么做视频编辑，添加特效

怎么取一帧视频画面编辑成图片并将二维码合成到图片中，然后进行压缩处理，压缩要保证图片的清晰度不变

生产者消费者（非阻塞式）

怎么监控应用中的线程，都是在哪创建的（AOP）

数据库的索引原理

flutter与其他跨平台方案对比，flutter为什么好（除了跨平台），跟原生的性能比怎么样 JavaScript中bind，call，apply的区别

介绍在上一家公司做的app，app的crash率是多少 hybrid开发中遇到了什么问题

17.Binder原理,缺点

11.跨进程文件写入如何保证安全

14.epoll机制是怎么样子的

MVVM kotlin协程原理，怎么实现的，扩展函数

连接数太多服务端关闭socket接口怎么解决

1.gradle task耗时分析 2.method耗时分析

13.ANR系统检查机制

10.static引用外部类变量