小米面试精装汇编习题+解析

小米面试虽然分几次进行，今天我们主要讲解专业面试习题汇编。面试流程大体分三个步骤:

一、自我介绍

主要介绍本科研究生毕业学校，选修专业，选修课程，实习经历（校招）。曾经工作的公司，工作职能、项目经历（社招），项目经历一般要求讲解印象较深的项目。

二、专业知识

（java 基础、Android、设计模式、扩展（项目经历与算法）、综合提问

2.1 Java 基础

1. Java 的引用方式？

答：强引用：是指创建一个对象并把这个对象赋给一个引用变量。

软引用：实现内存敏感的高速缓存,比如网页缓存、图片缓存等 防止内存泄露，增强程序的健壮性。

弱引用：描述非必需对象的，当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。

虚引用：虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。

2. Java 线程生命周期？
答：NEW（初始化状态）
RUNNABLE（可运行 / 运行状态）
BLOCKED（阻塞状态）
WAITING（无时限等待）
TIMED_WAITING（有时限等待）
TERMINATED（终止状态）
3.Java 中的同步方法？

答： synchronized 关键字修饰方法

同步代码块

使用重入锁实现线程同步

使用局部变量实现线程同步

4.Java 内存模型？

答：在 Java 中应为不同的目的可以将 java 划分为两种内存模型：gc 内存模型。并发内存模型。

5. java 的内存区域？

答：指 Jvm 运行时将数据分区域存储，强调对内存空间的划分。

程序计数器：存储下一条指令的地址，每个线程都有一个程序计数器

虚拟机栈：也是线程私有的，每进入一个方法都会在栈中申请一个栈帧用于存储局部变量，参数等。

本地方法栈：为 native 方法服务

堆：所有的线程共享，是存储实例对象的空间，可通过-Xmx 和-Xms 来指定堆的大小

方法区：存储已经加载的类信息，运行时常量池是方法区的一部分，用于存储直接引用和 class 文件中的常量池中的常量

6. 对象都是在堆中吗？变量又是放在哪块？

答： 一般来说，如果你用 new 来生成的对象都是放在堆中的，而直接定义的局部变量都是放在栈中的，全局和静态的对象是放在数据段的静态存储区，例如：

Class People；People p;//栈上分配内存

People* pPeople；pPeople = new People;//堆上分配内存

7. 线程与进程，线程是什么单位，进程呢？

答：进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

线程（英语：thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。

8. 什么是线程池？

答：线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件），则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。

9. 什么是线程安全？

答：线程安全是多线程编程时的计算机程序代码中的一个概念。在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况。

10.线程池参数的意义？

答： 1. corePoolSize: 核心线程数

缺省值为 1

核心线程会一直存活，即使没有任务需要执行

当线程数小于核心线程数，即使有空闲线程，线程池也会优先创建新线程处理

设置 allowCoreThreadTimeout=true(默认 false)时，核心线程会超时关闭

2. queueCapacity: 任务队列容量（阻塞队列）

当核心线程数达到最大时，新任务会放在队列中排队等待执行

3. maxPoolSize: 最大线程数

当前线程数 >= corePoolSize, 且任务对列已满时，线程池会创建新线程来处理任务

当前线程数 = maxPoolSize, 且任务对列已满时，线程池会拒绝处理任务而抛出异常

4. keepAliveTime: 线程空闲时间

当空闲时间达到 keepAliveTime 时，线程会退出，直到线程数量等于 corePoolSize

如果 allowCoreThreadTimeout=true，则线程会退出，直到线程数量等于 0

5. allowCoreThreadTimeout：允许核心线程超时

缺省是 false

6. rejectedExecutionHandler: 任务拒绝处理器

当线程数达到 maxPoolSize，且队列已满，会拒绝新任务

当线程池调用 shutdown()和线程池真正 shutdown 之间提交的任务会被拒绝。

11.抽象类和接口有什么区别？

答：

参数	抽象类	接口
默认的方法实现	它可以有默认的方法实现	接口完全是抽象的。它根本不存在方法的实现
实现	子类使用 extends 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。	子类使用关键字 implements 来实现接口。它需要提供接口中所有声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
与正常 Java 类的区别	除了你不能实例化抽象类之外，它和普通Java类没有任何区别	接口是完全不同的类型
访问修饰符	抽象方法可以有 public、protected 和 default 这些修饰符	接口方法默认修饰符是 public。你不可以使用其它修饰符。
main 方法	抽象方法可以有 main 方法并且我们可以运行它	接口没有 main 方法，因此我们不能运行它。
多继承	抽象方法可以继承一个类和实现多个接口	接口只可以继承一个或多个其它接口
速度	它比接口速度要快	接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
添加新方法	如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。	如果你往接口中添加方法，那么你必须改变实现该接口的类。

11.由数组实现的集合有哪些，跟链表实现的有什么区别？

答：1.数组静态分配内存，链表动态分配内存；

2.数组在内存中连续，链表不连续；

3.数组元素在栈区，链表元素在堆区；

4.数组利用下标定位，时间复杂度为 O(1)，链表利用引用定位元素时间复杂度 O(n)；

5.数组插入或删除元素的时间复杂度 O(n)，链表的时间复杂度 O(1)。

12. hashmap，hashtable，concurrenthashmap 区别？

答：HashTable

底层数组+链表实现，无论 key 还是 value 都**不能为 null**，线程**安全**，实现线程安全的方式是在修改数据时锁住整个 HashTable，效率低，ConcurrentHashMap 做了相关优化

初始 size 为 **11**，扩容：newsize = olesize*2+1

计算 index 的方法：index = (hash & 0x7FFFFFFF) % tab.length

HashMap

底层数组+链表实现，可**以存储 null 键和 null 值**，线程**不安全**

初始 size 为 **16**，扩容：newsize = oldsize*2，size 一定为 2 的 n 次幂

扩容针对整个 Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入

插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生无效扩容）

当 Map 中元素总数超过 Entry 数组的 75%，触发扩容操作，为了减少链表长度，元素分配更均匀

计算 index 方法：index = hash & (tab.length – 1)

HashMap 的初始值还要考虑加载因子:

1.哈希冲突：若干 Key 的哈希值按数组大小取模后，如果落在同一个数组下标上，将组成一条 Entry 链，对 Key 的查找需要遍历 Entry 链上的每个元素执行 equals()比较。

2.加载因子：为了降低哈希冲突的概率，默认当 HashMap 中的键值对达到数组大小的 75%时，即会触发扩容。因此，如果预估容量是 100，即需要设定 100/0.75＝134 的数组大小。

3.空间换时间：如果希望加快 Key 查找的时间，还可以进一步降低加载因子，加大初始大小，以降低哈希冲突的概率。

HashMap 和 Hashtable 都是用 hash 算法来决定其元素的存储，因此 HashMap 和 Hashtable 的 hash 表包含如下属性：

容量（capacity）：hash 表中桶的数量

初始化容量（initial capacity）：创建 hash 表时桶的数量，HashMap 允许在构造器中指定初始化容量

尺寸（size）：当前 hash 表中记录的数量

负载因子（load factor）：负载因子等于“size/capacity”。负载因子为 0，表示空的 hash 表，0.5 表示半满的散列表，依此类推。轻负载的散列表具有冲突少、适宜插入与查询的特点（但是使用 Iterator 迭代元素时比较慢）

除此之外，hash 表里还有一个“负载极限”，“负载极限”是一个 0～1 的数值，“负载极限”决定了 hash 表的最大填满程度。当 hash 表中的负载因子达到指定的“负载极限”时，hash 表会自动成倍地增加容量（桶的数量），并将原有的对象重新分配，放入新的桶内，这称为 rehashing。

HashMap 和 Hashtable 的构造器允许指定一个负载极限，HashMap 和 Hashtable 默认的“负载极限”为 0.75，这表明当该 hash 表的 3/4 已经被填满时，hash 表会发生 rehashing。

“负载极限”的默认值（0.75）是时间和空间成本上的一种折中：

较高的“负载极限”可以降低 hash 表所占用的内存空间，但会增加查询数据的时间开销，而查询是最频繁的操作（HashMap 的 get()与 put()方法都要用到查询）

较低的“负载极限”会提高查询数据的性能，但会增加 hash 表所占用的内存开销

程序猿可以根据实际情况来调整“负载极限”值。

ConcurrentHashMap

底层采用分段的数组+链表实现，线程**安全**

通过把整个 Map 分为 N 个 Segment，可以提供相同的线程安全，但是效率提升 N 倍，默认提升 16 倍。(读操作不加锁，由于 HashEntry 的 value 变量是 volatile 的，也能保证读取到最新的值。)

Hashtable 的 synchronized 是针对整张 Hash 表的，即每次锁住整张表让线程独占，ConcurrentHashMap 允许多个修改操作并发进行，其关键在于使用了锁分离技术

有些方法需要跨段，比如 size()和 containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁

扩容：段内扩容（段内元素超过该段对应 Entry 数组长度的 75%触发扩容，不会对整个 Map 进行扩容），插入前检测需不需要扩容，有效避免无效扩容

Hashtable 和 HashMap 都实现了 Map 接口，但是 Hashtable 的实现是基于 Dictionary 抽象类的。Java5 提供了 ConcurrentHashMap，它是 HashTable 的替代，比 HashTable 的扩展性更好。

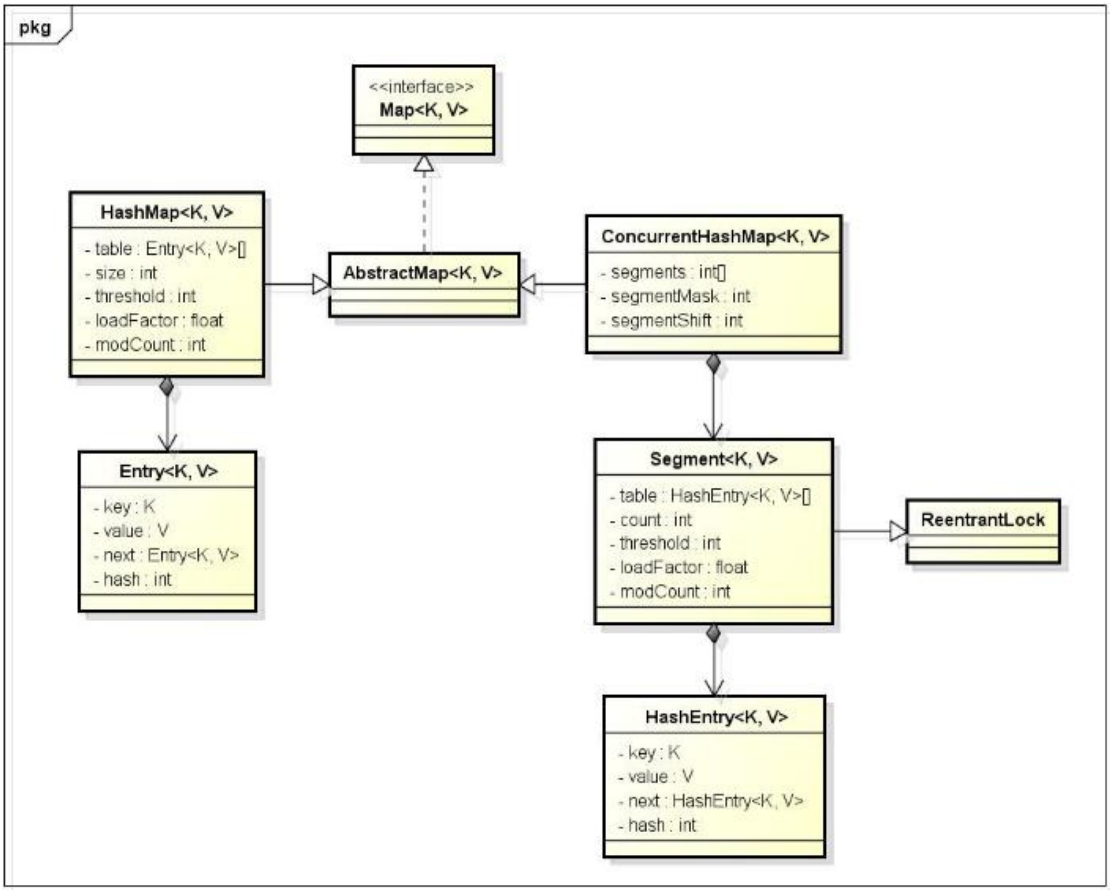
HashMap 基于哈希思想，实现对数据的读写。当我们将键值对传递给 put()方法时，它调用键对象的 hashCode()方法来计算 hashcode，然后找到 bucket 位置来存储值对象。当获取对象时，通过键对象的 equals()方法找到正确的键值对，然后返回值对象。HashMap 使用链表来解决碰撞问题，当发生碰撞时，对象将会储存在链表的下一个节点中。HashMap 在每个链表节点中储存键值对对象。当两个不同的键对象的 hashcode 相同时，它们会储存在同一个 bucket 位置的链表中，可通过键对象的 equals()方法来找到键值对。如果链表大小超过阈值（TREEIFY_THRESHOLD,8），链表就会被改造为树形结构。

在 HashMap 中，null 可以作为键，这样的键只有一个，但可以有一个或多个键所对应的值为 null。**当 get()方法返回 null 值时，即可以表示 HashMap 中没有该 key，也可以表示该 key 所对应的 value 为 null**。因此，在 HashMap 中不能由 get()方法来判断 HashMap 中是否存在某个 key，应该用 **containsKey()**方法来判断。而在 Hashtable 中，无论是 key 还是 value 都不能为 null。

Hashtable 是线程安全的，它的方法是同步的，可以直接用在多线程环境中。而 HashMap 则不是线程安全的，在多线程环境中，需要手动实现同步机制。

Hashtable 与 HashMap 另一个区别是 HashMap 的迭代器（Iterator）是 fail-fast 迭代器，而 Hashtable 的 enumerator 迭代器不是 fail-fast 的。所以当有其它线程改变了 HashMap 的结构（增加或者移除元素），将会抛出 ConcurrentModificationException，但迭代器本身的 remove()方法移除元素则不会抛出 ConcurrentModificationException 异常。但这并不是一个一定发生的行为，要看 JVM。

先看一下简单的类图：



从类图中可以看出来在存储结构中 ConcurrentHashMap 比 HashMap 多出了一个类 Segment，而 Segment 是一个可重入锁。

ConcurrentHashMap 是使用了锁分段技术来保证线程安全的。

锁分段技术：首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap 提供了与 Hashtable 和 SynchronizedMap 不同的锁机制。Hashtable 中采用的锁机制是一次锁住整个 hash 表，从而在同一时刻只能由一个线程对其进行操作；而 ConcurrentHashMap 中则是一次锁住一个桶。

ConcurrentHashMap 默认将 hash 表分为 16 个桶，诸如 get、put、remove 等常用操作只锁住当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。

13. concurrenthashmap 分段锁详解？

答：**继承关系：**

```
public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
        implements ConcurrentMap<K, V>, Serializable {
    private static final long serialVersionUID = 7249069246763182397L;
```

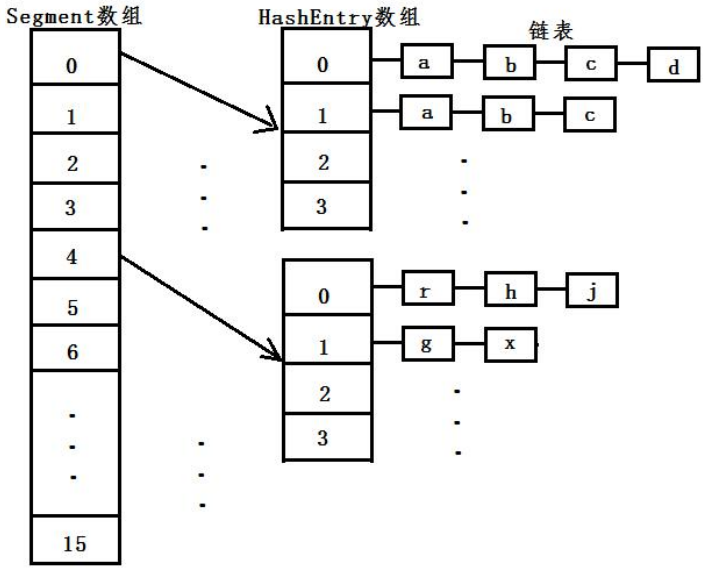
成员变量：

```
static final int DEFAULT_INITIAL_CAPACITY = 16;//初始容量为16
static final float DEFAULT_LOAD_FACTOR = 0.75f;//负载因子为0.75
static final int DEFAULT_CONCURRENCY_LEVEL = 16;//默认并发度为16（就是初始容量的大小，一旦指定了，就不能更改大小了），就是16个线
static final int MAXIMUM_CAPACITY = 1 << 30;//Segment的最大容量
static final int MAX_SEGMENTS = 1 << 16;
static final int RETRIES_BEFORE_LOCK = 2;
final Segment<K,V>[] segments;//存储数据的容器
```

底层结构：

数组+数组+链表

如下图：



默认的数组长度为16


```
final Segment<K,V>[] segments;
transient Set<K> keySet;
transient Set<Map.Entry<K,V>> entrySet;
transient Collection<V> values;
```

构造方法：

```java
public ConcurrentHashMap(int initialCapacity,float loadFactor, int concurrencyLevel) {//带参数的构造函数，初始容量，负载因子，
    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();

    if (concurrencyLevel > MAX_SEGMENTS)
        concurrencyLevel = MAX_SEGMENTS;

    // Find power-of-two sizes best matching arguments
    int sshift = 0;
    int ssize = 1;
    while (ssize < concurrencyLevel) {
        ++sshift;
        ssize <<= 1;
    }
    segmentShift = 32 - sshift;
    segmentMask = ssize - 1;
    this.segments = Segment.newArray(ssize);

    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    int c = initialCapacity / ssize;
    if (c * ssize < initialCapacity)
        ++c;
    int cap = 1;
    while (cap < c)
        cap <<= 1;

    for (int i = 0; i < this.segments.length; ++i)
        this.segments[i] = new Segment<K,V>(cap, loadFactor);
}

public ConcurrentHashMap(int initialCapacity, float loadFactor) {//带有参数的构造函数，初始容量，负载因子
    this(initialCapacity, loadFactor, DEFAULT_CONCURRENCY_LEVEL);
}

public ConcurrentHashMap(int initialCapacity) {//带有参数的构造函数，初始容量
    this(initialCapacity, DEFAULT_LOAD_FACTOR, DEFAULT_CONCURRENCY_LEVEL);
}

public ConcurrentHashMap() {//无参构造函数，使用默认值调用三参数的构造函数
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR, DEFAULT_CONCURRENCY_LEVEL);
}

```

基本操作：

读操作：

因为读操作不会改变数值的大小，所以它是不加锁的，而且 HashEntry 加了 volatile 关键字，也可以保证读取到的值是最新值

同时读操作可以多个线程进行操作，即至少可以有 16 个线程操作，至多无上限，只要在 CPU 的能力承受范围内即可。

```
*/
static final class HashEntry<K,V> {
    final K key;
    final int hash;
    volatile V value;
    final HashEntry<K,V> next;
```

```
1   V get(Object key, int hash) {
2       if (count != 0) { // read-volatile
3           HashEntry<K,V> e = getFirst(hash);
4           while (e != null) {
5               if (e.hash == hash && key.equals(e.key)) {
6                   V v = e.value;
7                   if (v != null)
8                       return v;
9                   return readValueUnderLock(e); // recheck
10              }
11              e = e.next;
12          }
13      }
14      return null;
15  }
```

写操作：

它是具有锁机制的，可以同时让 16（默认容量的）个线程进行操作，每个线程进入各自所需寻找的 Segment 数组中的 HashEentry 数组进行操作，大大的提高了操作效率。

```
        V put(K key, int hash, V value, boolean onlyIfAbsent) {
            lock();
            try {
                int c = count;
                if (c++ > threshold) // ensure capacity
                    rehash();
                HashEntry<K,V>[] tab = table;
                int index = hash & (tab.length - 1);
                HashEntry<K,V> first = tab[index];
                HashEntry<K,V> e = first;
                while (e != null && (e.hash != hash || !key.equals(e.key)))
                    e = e.next;

                V oldValue;
                if (e != null) {
                    oldValue = e.value;
                    if (!onlyIfAbsent)
                        e.value = value;
                }
                else {
                    oldValue = null;
                    ++modCount;
                    tab[index] = new HashEntry<K,V>(key, hash, first, value);
                    count = c; // write-volatile
                }
                return oldValue;
            } finally {
                unlock();
            }
        }
```

ConcurrentHashMap 的主要特点：

1.存储方面采用了数组加数组加链表的形式

2.在线程对每部分的 Segment 段读取数据时，可以多个线程访问，效率较高；在线程对每部分的 Segment 段写数据时，加锁机制启动，只许一个进入，线程安全

3.安全性方面采用了分段的锁机制机，即保证了多线程的安全性，又提高了访问的效率

14.HashMap、HashTable 和 ConcurrentHashMap 的区别？

答：1.底层结构

HashMap 是数组加链表

HashTable 是数组加链表

ConcurrentHashMap 是数组加数组加链表

2.扩容

HashMap 是初始容量右移一位加一

HashTable 是初始容量右移一位

ConcurrentHashMap 是初始的 Segment 数组保持不变，只将 HashEntry 的数组进行扩容

3.继承关系

HashMap 是继承于 extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

HashTable 是继承于 extends Dictionary<K,V>implements Map<K,V>, Cloneable, java.io.Serializable

ConcurrentHashMap 是继承于 extends AbstractMap<K, V>implements ConcurrentMap<K, V>, Serializable

4.在线程安全方面：

HashMap 在单线程下是安全的，多线程下不安全，

HashTable 在多线程下是安全的，因为它在基本操作前面都加上了 synchronized 锁，即就是就算在多个线程操作的时候，只要有一个线程进入了基本方法内，其他的线程就必须等待。只有当第一个线程操作结束，释放了锁，其他的线程才可以进行操作。虽然说在多线程操作的时候线程是安全的但是效率比较低，毕竟那么多的线程就在等一把锁。

```
public synchronized V put(K key, V value) {
// Make sure the value is not null
if (value == null) {
    throw new NullPointerException();
}
// Makes sure the key is not already in the hashtable.
public synchronized V remove(Object key) {
public synchronized void putAll(Map<? extends K, ? extends V> t) {
    for (Map.Entry<? extends K, ? extends V> e : t.entrySet())
        put(e.getKey(), e.getValue());
}
public synchronized void clear() {
```

不能存储null值

ConcurrentHashMap 是安全的，理由上方可寻。

15 多线程 wait 和 sleep 区别？

答：wait 和 sleep 区别

共同点：

1. 他们都是在多线程的环境下，都可以在程序的调用处阻塞指定的毫秒数，并返回。

2. wait()和 sleep()都可以通过 interrupt()方法 打断线程的暂停状态 ，从而使线程立刻抛出 InterruptedException。

如果线程 A 希望立即结束线程 B，则可以对线程 B 对应的 Thread 实例调用 interrupt 方法。如果此刻线程 B 正在 wait/sleep /join，则线程 B 会立刻抛出 InterruptedException，在 catch() {} 中直接 return 即可安全地结束线程。

需要注意的是，InterruptedException 是线程自己从内部抛出的，并不是 interrupt()方法抛出的。对某一线程调用 interrupt()时，如果该线程正在执行普通的代码，那么该线程根本就不会抛出 InterruptedException。但是，一旦该线程进入到 wait()/sleep()/join()后，就会立刻抛出 InterruptedException 。

不同点：

1. Thread 类的方法：sleep(),yield()等

Object 的方法：wait()和 notify()等

2. 每个对象都有一个锁来控制同步访问。Synchronized 关键字可以和对象的锁交互，来实现线程的同步。

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

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

4. sleep 必须捕获异常，而 wait，notify 和 notifyAll 不需要捕获异常

所以 sleep()和 wait()方法的最大区别是：

sleep()睡眠时，保持对象锁，仍然占有该锁；

而 wait()睡眠时，释放对象锁。

但是 wait()和 sleep()都可以通过 interrupt()方法打断线程的暂停状态，从而使线程立刻抛出 InterruptedException（但不建议使用该方法）。

sleep（）方法

sleep()使当前线程进入停滞状态（阻塞当前线程），让出 CUP 的使用、目的是不让当前线程独自霸占该进程所获的 CPU 资源，以留一定时间给其他线程执行的机会;

sleep()是 Thread 类的 Static(静态)的方法；因此他不能改变对象的机锁，所以当在一个 Synchronized 块中调用 Sleep()方法是，线程虽然休眠了，但是对象的机锁并木有被释放，其他线程无法访问这个对象（即使睡着也持有对象锁）。

在 sleep()休眠时间期满后，该线程不一定会立即执行，这是因为其它线程可能正在运行而且没有被调度为放弃执行，除非此线程具有更高的优先级。

wait（）方法

wait()方法是 Object 类里的方法；当一个线程执行到 wait()方法时，它就进入到一个和该对象相关的等待池中，同时失去（释放）了对象的机锁（暂时失去机锁，wait(long timeout)超时时间到后还需要返还对象锁）；其他线程可以访问；

wait()使用 notify 或者 notifyAlll 或者指定睡眠时间来唤醒当前等待池中的线程。

wiat()必须放在 synchronized block 中，否则会在 program runtime 时扔出“java.lang.IllegalMonitorStateException”异常。

16.两个线程分别打印 log ？

答：这个类是单例模式，因为要在多线程里使用，输出信息到同一个 log 文件中，因此单例模式是比较好的选择。

1.多线程是很难调试的。在可能引发错误的函数入口和出口分别打上 log 信息，这样就可以分析哪里出错了。先封装了一个用来输出 log 信息的类，经过调试，发现使用挺方便。

源代码：

```
#include "stdafx.h"

#include<iostream>

#include<process.h>

#include<windows.h>

#include<list>

#include<string>

using namespace std;

class CBaselock

{

public:

CBaselock()

{

InitializeCriticalSection(&m_Sec);//初始化临界区

}

~CBaselock()

{

DeleteCriticalSection(&m_Sec);

}

void Lock()

{
```

```
EnterCriticalSection(&m_Sec);
}
void UnLock()
{
LeaveCriticalSection(&m_Sec);
}
private:
CRITICAL_SECTION m_Sec;
};

class CLog
{
public:
static CLog* getInstance()//返回 CLog 的单例
{
static CLog m_log;
return &m_log;
}
void PushLogInfo(char* szInfo);
static unsigned __stdcall SaveLogThred(void * pThis)//保存 Log 线程
{
CLog * pthX = (CLog*)pThis;
pthX->SaveLogInfo();
return 1;
```

```
}

private:

CLog();

~CLog();

void SaveLogInfo();

FILE* m_pFile;

CBaselock m_lock;

char m_szLogPath[MAX_PATH];

list<string> m_InfoLst;

bool m_IsRun;



};

CLog::CLog()

{

GetModuleFileNameA(NULL, m_szLogPath, MAX_PATH); //获取当前程序 exe 的全路径

strrchr(m_szLogPath, '\\')[1] = '\0';//去掉 exe 文件名

strcat_s(m_szLogPath,"Log.txt");//重新命名

fopen_s(&m_pFile, m_szLogPath, "w+");//打开 Log.txt 文件，如果不存在，则创建该文件

m_IsRun = true;

_beginthreadex(NULL, 0, &SaveLogThred, this, 0, 0);

}

CLog::~CLog()

{

m_IsRun = false; //自然中止 SaveLogThred 线程
```

```
Sleep(1000);//确保 SaveLogInfo()中的 while 循环自然中止。
fclose(m_pFile);
}
void CLog::PushLogInfo(char* szInfo)
{
m_lock.Lock();
m_InfoLst.push_back(szInfo);
m_lock.UnLock();
}
void CLog::SaveLogInfo()
{
while (m_IsRun)
{
if (m_InfoLst.empty())
{
Sleep(10);
continue;
}
fputs(m_InfoLst.front().c_str(), m_pFile);
fflush(m_pFile);
m_lock.Lock();
m_InfoLst.pop_front();
m_lock.UnLock();
Sleep(0);
```

```
}

}
```

在需要输出 log 的地方这样调用：

```
CLog* pLog = CLog::getInstance();

pLog->PushLogInfo(“123456”);
```

2.解决多个线程同时写一个 log 文件的办法有几种，一是加锁，二是用一个消息队列，把要打印的信息放到一个链表里，然后开启一个专门的线程从这个链表里取数据，输出到 log 文件中。第二种方式效率很高，不用在调用的地方等待。

17.class 对象什么时候加载，如何控制类中某个块的加载？

答：当程序创建第一个对类的静态成员的引用时，就会加载这个类。这个证明构造器也是类的静态方法，即使构造器并没有使用 static 关键字，我们称构造器是隐式静态的。因此，使用 new 操作符创建类的新对象也会被当做对类的静态成员的引用。

1.Class 对象的概念：

对类的加载过程了解后，我们发现 Class 对象是在类的加载过程中产生的，而完成类的加载过程才可以创建类的普通对象.其实一旦某个类的 Class 对象被载入内存，它就被用来创建这个类的所有对象。

注意：Class 对象的创建和 Class 对象的初始化是分开来的。也就是说，创建了一个 Class 对象，不一定是对 Class 对象初始化了。

2.什么情况下会创建 class 对象

1.使用类字面常量（类名.class）

类字面常量(类名.class)和 Class.forName()方法 |

类字面常量的格式是

类名.class

这个语法会创建一个类的 Class 对象。注意 ,这仅仅是创建一个类的 Class 对象，并没有完成类的加载的全部的 3 个过程。如

Dog.class 表示创建一个 Dog 的 class 对象。

Class.forName()的格式是

Class.forName("类的全限定名")

如果上面的 Dog 类在 com.guo 包下，则

Class.forName("com.guo.Dog");

Class.forName()方法会执行类的加载的全部过程，即类加载过程的 3 个步骤都会执行。类加载重要的第 3 步会对 Class 对象初始化化。即创建 Dog 类的 Class 对象，对 Dog 类的静态域分配空间，初始化 Dog 类的 Class 对象。

18.问到双亲委派机制，但是是举例子问的。如果自己写了个跟 java 提供的一模一样的类。（包名类名都一样）为什么虚拟机可以正确加载系统提供的。

答： 双亲委派机制得工作过程：
1-类加载器收到类加载的请求；
2-把这个请求委托给父加载器去完成，一直向上委托，直到启动类加载器；
3-启动器加载器检查能不能加载（使用 findClass()方法），能就加载（结束）；否则，抛出异常，通知子加载器进行加载。
4-重复步骤 3；

18.什么样的对象会被 gc 回收？

答：Java 的虚拟机本身是蛮复杂的，我们不仔细讲细节。我们只针对我们平时最关心的**堆**中的哪些对象会被 GC 回收。我们这样思考，既然 GC 要回收这块内存，那总得有个方法让 GC 可以判断哪些对象时可以被回收而哪些是不能被回收的吧？这就引出了下面常见的两种判断方法。

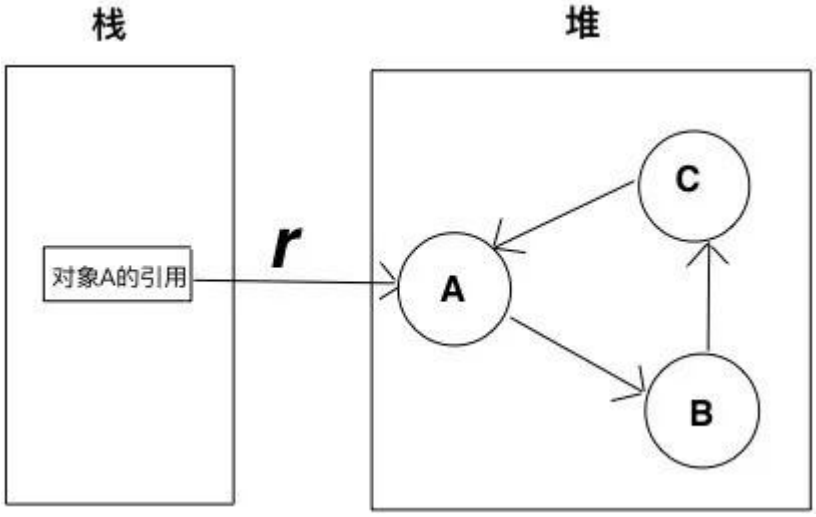
- 引用计数法
- 可达性分析算法

下面便具体讲解下两种方法：

. 引用计数法

这种方法是在对象的头处维护一个计数器 **Counter**，当有一个引用指向对象的时候 counter 就加一，当不在引用此对象时就让 counter 减一。所以，当 counter 等于零的时候虚拟机就认为此对象时可以被回收的。看起来好像有点道理，但是

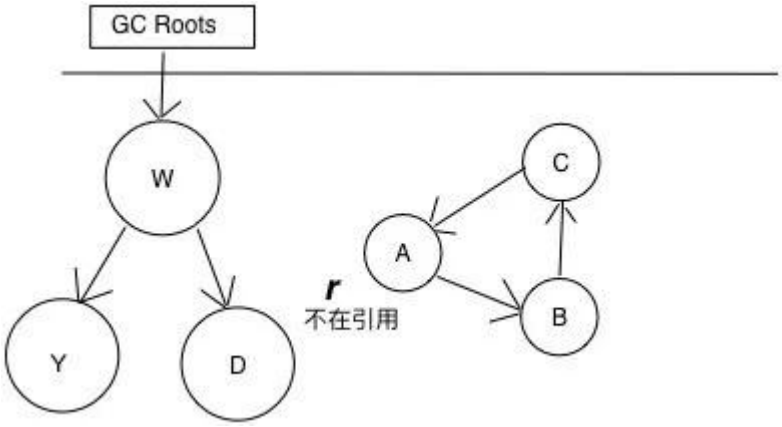
这种方法存在一个致命的问题：



如上图所示：**外部对对象 A 有一个引用，对象 A 持有对象 B，而对象 B 也持有一个对象 C，对象 C 又持有对象 A。如果对于对象 A 的引用 r 失效，按照引用计数方法，GC 永远无法回收上面的三个对象。**所以基于上面的存在内存泄漏的巨大缺陷，Java 虚拟机（应该是大多数虚拟机）不采用此方法进行回收内存。

．可达性分析算法

Java 就是使用此方法作为判断对象是否可被回收的。虚拟机会先将一些对象定义为 GC Roots，从 GC Roots 出发一直沿着引用链向下寻找，如果某个对象不能通过 GC Roots 寻找到，那么虚拟机就认为该对象可以被回收。我们举个例子，如下图：



当对象 D 不在引用对象 A 时，尽管 A、B、C 互相还持有引用，GC 依然会回收 ABC 所占用的内存。那么还有个疑问，什么样的对象可以被看做是 GC Roots 呢？

- 虚拟机栈(栈桢中的本地变量表)中的引用的对象
- 方法区中的类静态属性引用的对象
- 方法区中的常量引用的对象

- 本地方法栈中 JNI（Native 方法）的引用的对象

19.内部类，持有外部引用。为什么内部类访问外部变量时，外部变量要设置成 final ？

答：若定义为 final，则 Java 编译器则会在内部类生成一个外部变量的拷贝，而且既可以保证内部类可以引用外部属性，又能保证值的唯一性。

若不定义为 final，则无法通过编译（jdk 1.6 测试过）。因为编译器不会给非 final 变量进行拷贝，那么内部类引用的变量就是非法的。

匿名内部类用法：

```
public class TryUsingAnonymousClass {
    public void useMyInterface() {
        final Integer number = 123;
        System.out.println(number);

        MyInterface myInterface = new MyInterface() {
            @Override
            public void doSomething() {
                System.out.println(number);
            }
        };
        myInterface.doSomething();

        System.out.println(number);
    }
}
```

编译后的结果

```
class TryUsingAnonymousClass$1
        implements MyInterface {
    private final TryUsingAnonymousClass this$0;
    private final Integer paramInteger;

    TryUsingAnonymousClass$1(TryUsingAnonymousClass this$0, Integer paramInteger) {
        this.this$0 = this$0;
        this.paramInteger = paramInteger;
    }

    public void doSomething() {
        System.out.println(this.paramInteger);
    }
}

```

因为匿名内部类最终会编译成一个单独的类，而被该类使用的变量会以构造函数参数的形式传递给该类，例如：Integer paramInteger，如果变量不定义成 final 的，paramInteger 在匿名内部类被可以被修改，进而造成和外部的 paramInteger 不一致的问题（因为逻辑的意图就是为了引用外层那个特定的变量对应的值或对象），为了避免这种不一致的情况，因次 Java 规定匿名内部类只能访问 final 修饰的外部变量。需要注意的是，在 Kotlin 的 lambda 表达式中，引用外层方法的局部变量时，虽然该变量可以不标记为 final，但实际上是 final 的，在 lambda 表达式内变更会报错。

方法内不允许使用 static 修饰变量。

且对于外部类的成员变量，是可以随意使用的，不需要修饰其为 final，因为匿名内部类会持有外部类的引用，实际上是通过该外部类的引用去使用其成员变量。

20.然后问了语言基础，你说一 java 的集合框架吧？

答：早在 Java 2 中之前，Java 就提供了特设类。比如：Dictionary, Vector, Stack, 和 Properties 这些类用来存储和操作对象组。

虽然这些类都非常有用，但是它们缺少一个核心的，统一的主题。由于这个原因，使用 Vector 类的方式和使用 Properties 类的方式有着很大不同。

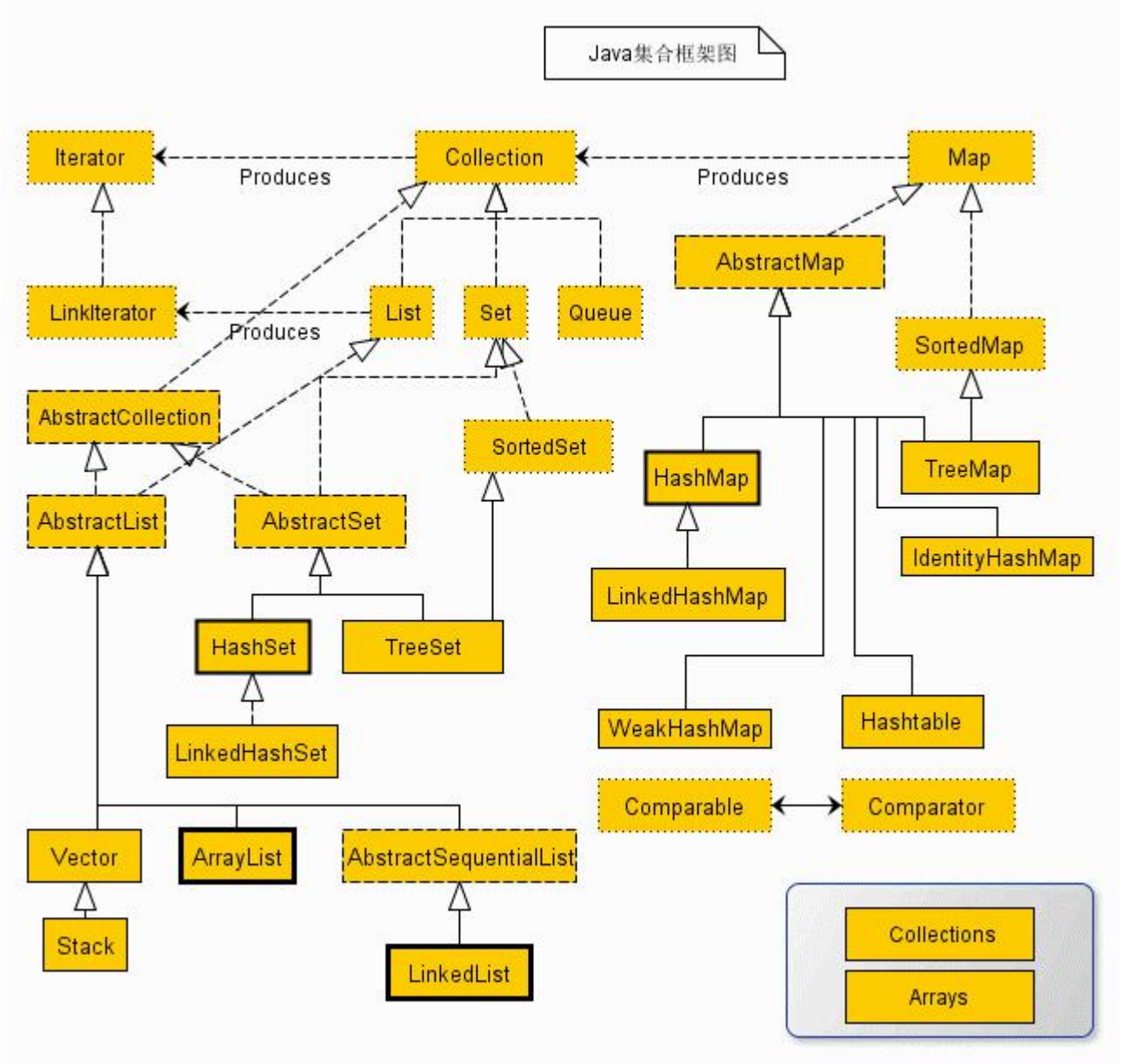
集合框架被设计成要满足以下几个目标。

该框架必须是高性能的。基本集合（动态数组，链表，树，哈希表）的实现也必须是高效的。

该框架允许不同类型的集合，以类似的方式工作，具有高度的互操作性。

对一个集合的扩展和适应必须是简单的。

为此，整个集合框架就围绕一组标准接口而设计。你可以直接使用这些接口的标准实现，诸如： **LinkedList**, **HashSet**, 和 **TreeSet** 等,除此之外你也可以通过这些接口实现自己的集合。



从上面的集合框架图可以看到，Java 集合框架主要包括两种类型的容器，一种是集合（Collection），存储一个元素集合，另一种是图（Map），存储键/值对映射。Collection 接口又有 3 种子类型，List、Set 和 Queue，再下面是一些抽象类，最后是具体实现类，常用的有 ArrayList、LinkedList、HashSet、LinkedHashSet、HashMap、LinkedHashMap 等等。

集合框架是一个用来代表和操纵集合的统一架构。所有的集合框架都包含如下内容：

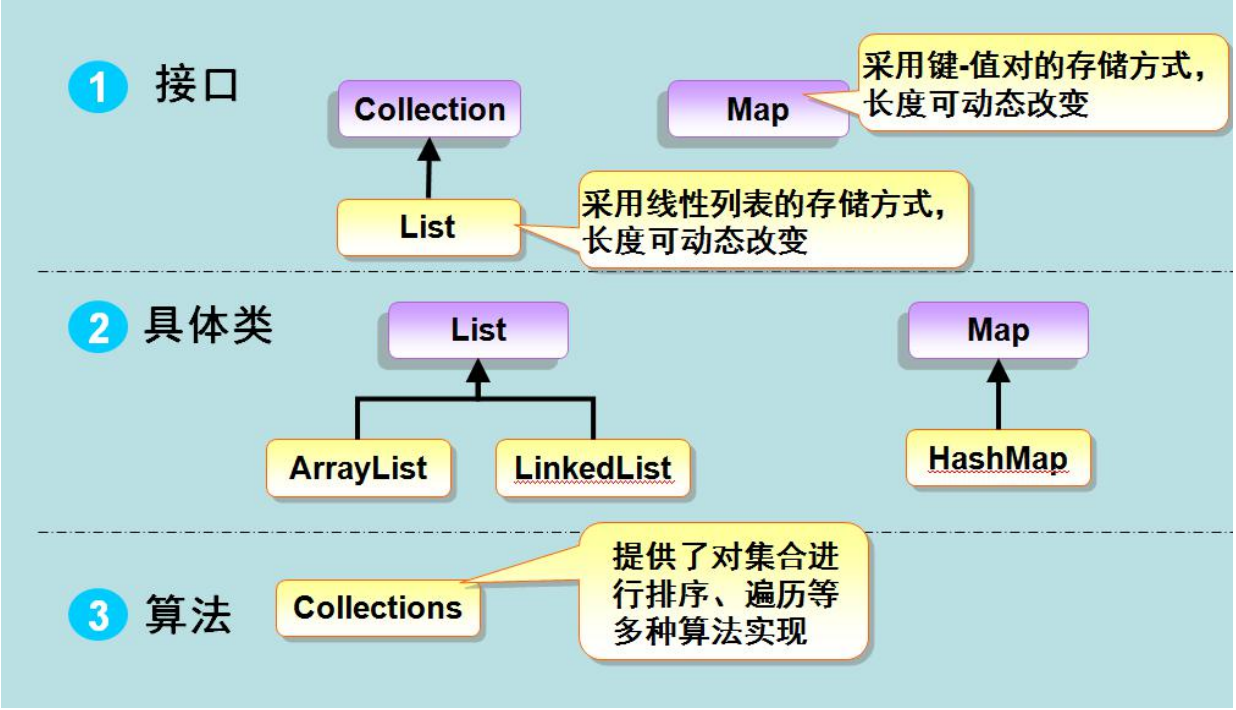
接口：是代表集合的抽象数据类型。例如 Collection、List、Set、Map 等。之所以定义多个接口，是为了以不同的方式操作集合对象

实现（类）：是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构，例如：ArrayList、LinkedList、HashSet、HashMap。

算法：是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序。这些算法被称为多态，那是因为相同的方法可以在相似的接口上有着不同的实现。

除了集合，该框架也定义了几个 Map 接口和类。Map 里存储的是键/值对。尽管 Map 不是集合，但是它们完全整合在集合中。

集合框架体系如图所示



Java 集合框架提供了一套性能优良，使用方便的接口和类，java 集合框架位于 java.util 包中， 所以当使用集合框架的时候需要进行导包。

21.写过一些脚本没有？
答：定义：为了缩短传统的编写-编译-链接-运行（edit-compile-link-run）过程而创建的计算机编程语言。

特点：程序代码即是最终的执行文件，只是这个过程需要解释器的参与，所以说脚本语言与解释型语言有很大的联系。脚本语言通常是被解释执行的，而且程序是文本文件。

典型的脚本语言有，JavaScript，Python，shell 等。

其他常用的脚本语句种类

PHP 是网页程序，也是脚本语言。是一款更专注于 web 页面开发（前端展示）的脚本语言，PHP 程序也可以处理系统日志，配置文件等，php 也可以调用系统命令。

Perl 脚本语言。比 shell 脚本强大很多，语法灵活、复杂，实现方式很多，不易读，团队协作困难，但仍不失为很好的脚本语言，存世大量的程序软件。MHA 高可用 Perl 写的

Python，不但可以做脚本程序开发，也可以实现 web 程序以及软件的开发。近两年越来越多的公司都会要求会 Python。

Shell 脚本与 php/perl/python 语言的区别和优势？

shell 脚本的优势在于处理操作系统底层的业务（linux 系统内部的应用都是 shell 脚本完成）因为有大量的 linux 系统命令为它做支撑。2000 多个命令都是 shell 脚本编程的有力支撑，特别是 grep、awk、sed 等。例如：一键软件安装、优化、监控报警脚本，常规的业务应用，shell 开发更简单快速，符合运维的简单、易用、高效原则.

PHP、Python 优势在于开发运维工具以及 web 界面的管理工具，web 业务的开发等。处理一键软件安装、优化，报警脚本。常规业务的应用等 php/python 也是能够做到的。但是开发效率和复杂比用 shell 就差很多了。

22.然后问了多线程，你能从硬件层面解释一下多线程么？

答： 线程安全性问题之硬件层面

一.CPU 高速缓存

线程是 CPU 调度的最小单元，线程出现的目的是为了更高效的利用 CPU 的计算处理能力，但是大部分的计算任务并不是仅仅依靠计算机的“计算”就能完成，处理器在读取运算数据，存储运算结果的过程中，还需要与内存进行交互，这个 IO 操作几乎是不能消除的。由于计算机的存储设备与处理器的计算速度差距非常大，所以现代的计算机都会增加一层读写速度尽可能接近处理器计算速度的高速缓存来作为内存与处理器之间的缓冲，将计算需要使用的数据复制到高速缓存中，让计算任务能够快速进行，计算完成后，在将结果从缓存同步到内存中。

从任务管理器中可以看到自己的计算机 CPU 和缓存的信息：

高速缓存越接近 CPU 速度越快，容量越小。现在的计算机大部分都有二级或三级缓存，如图，L1，L2，L3，缓存又可分为指令缓存（用 i 表示，缓存程序的代码）和数据缓存（用 d 表示，缓存程序的数据）。L3 缓存以前是服务器才会有的，现在家用电脑也有了，他出现的主要目的是进一步降低内存操作的延迟问题。

```
L1 Cache，一级缓存，本地 core 的缓存，分成 32K 的数据缓存 L1d 和 32k 指
令缓存 L1i，访问 L1 需要 3cycles，耗时大约 1ns；
L2 Cache，二级缓存，本地 core 的缓存，被设计为 L1 缓存与共享的 L3 缓存
之间的缓冲，大小为 256K，访问 L2 需要 12cycles，耗时大约 3ns；
L3 Cache，三级缓存，在同插槽的所有 core 共享 L3 缓存，分为多个 2M 的
段，访问 L3 需要 38cycles，耗时大约 12ns；
```

二.高速缓存带来的问题

高速缓存带来的问题即缓存一致性问题，在多核 CPU 机器中，当一个 CPU 读取主存的数据缓存到自己的高速缓存中时，另一个 CPU 也在做同一件事，并且把读取的值改变了，改变后把值同步到自己的高速缓存中，但是没有写入到主存，当第一个 CPU 去访问这个值的时候，由于第二个 CPU 没有把值同步到主存，所以，第一个 CPU 读取到的值还是改变之前的值，这时就会出现数据不一致的情况。

因为在多核 CPU 下会存在指令并行执行的情况，而各个 CPU 之间的数据又不共享就会导致缓存一致性问题，为了解决这个问题，各个 CPU 生产厂商提供了相应的解决方案。

三.解决方案

1.总线锁

当一个 CPU 对其缓存中的数据进行操作时，往总线中发送一个#Lock 信号，其他的处理器请求将会被阻塞，该处理器可以独占共享内存。

特点：总线锁相当于把整个 CPU 和内存之间的通信锁住了，这种方式会导致 CPU 性能下降。所以出现了缓存锁。

2.缓存锁

如果缓存在处理器缓存行中的内存区域在#Lock 期间被锁定（如果声明了 CPU 的锁机制，会生成一个#Lock 指令），当他执行锁操作回写内存时，处理器不会在总线上声明#Lock 信号，而是修改内部的缓存地址，通过缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时被两个以上处理器缓存的内存区域的数据去做修改操作，当另外一个处理器回写已经被锁定的缓存行的数据时，当前处理器缓存行的数据将会无效。

特点： 1.#Lock 前缀指令（汇编指令）会引起处理器缓存回写到内存，在 P6 系列以后的处理器中，Lock 指令一般不锁总线，只锁缓存。

2.一个处理器的缓存回写到内存会导致其他处理器的缓存无效。

四.缓存一致性机制

1.不同的操作系统或 CPU 架构，支持的协议不一样，每个处理器上会有一套完整的协议来保证缓存的一致性，如强一致性协议，MESI，比较经典的一种就是 MESI 协议了（它主要是在 CPU 缓存中保存一个标记位来实现 的），也是大部分处理器都在使用的。

M:modified，修改缓存，当前 CPU 的缓存数据已经被修改，和内存中的数据不一致了

E:exclusive，独占缓存，当前 CPU 的缓存和内存中的数据保持一致，而且其他处理器没有缓存该数据

S:shared，共享缓存，数据和内存中数据一致，并且该数据存在于多个 CPU 缓存中

I:invalid，失效缓存，CPU 的缓存已经不能使用 了

2.嗅探协议：每个处理器的缓存控制器不仅知道自己的读写操作，同时 也在监听着其他缓存的读写操作。

3. CPU 的读取会遵循几个原则：

a.如果缓存的状态是 I，就从内存中读取

b.如果缓存处于 M 或者 E 的 CPU 嗅探到其他 CPU 有读的操作，就把自己的缓存同步到内存，并把自己的状态设置为 S

c.只有缓存状态是 M 或者 E 的时候，CPU 才可以修改缓存

五.CPU 的优化执行

除了增加高速缓存以外，为了更充分利用处理器的运算单元，处理器可能会对输入的代码进行乱序执行优化，处理器会在计算之后将乱序执行的结果处理保证该结果与顺序执行之后的结果一致，但并不保证程序中各个语句先后的计算顺序和输入时的一致，这就是 CPU 的优化执行，与其类似的是编程语言的编译器的优化，比如指令重排序。Java 中在保证不影响我们最终执行的代码语义的情况下，允许编译器和指令器对我们的代码进行优化和指令的重排序去提升 CPU 的利用率。

然后问了一下网络知识，说一下网络的五层结构？
答：我用 http 请求，讲述了整个传输过程，应用层、传输层、网络层、数据链路层、物理层，各种加首部操作，大体讲了一下，空气暂停一秒，只听面试官说行，我们下一个问题...黑人，问号？心想怎么不深入问下去了

23. Java 虚拟机的调优经验？

答：1.jps:jvm process status tool-java 虚拟机进程状况工具

jps -l 是输出主类名 列出进程 id

jps -m 输出 JVM 启动时传递给 main()的参数

jps -v 显示虚拟机参数配置

-Xms 堆内存最小,-Xmx 堆内存最大,-XX:MaxPerSize=256m,永久代大小最大为多少，-Xmn 年轻代堆的大小，-Xss 栈、线程栈的大小

2.jstat:虚拟机运行时的信息监控

jstat -class 监视类装载，卸载数量，总空间以及类装载所耗费的时间。

jstat -gc 监视类堆状况，包括 Eden 区，两个 survivor 区，老年代，永久代的空间和已用的空间。

jstat -gcutil 监视已使用空间占总空间的百分比。

这两个命令里面还包括了新生代 GC 的次数和所花的时间，以及 Full GC 的次数和所花的时间。

3.jmap:生成虚拟机的内存转储快照，获取 dump 文件

可以查询 finalize 执行队列，java 堆和永久代的详细信息。空间的使用率，以及使用的哪种收集器。

jmap -histo pid:

展示 java 堆中对象的统计信息

instances（实例数）、bytes（大小）、classs name（类名）。它基本是按照使用使用大小逆序排列的。

jmap -dump:format=b,file=lijun pid 生成当前线程的 dump 文件

jmap -finalizerinfo

打印等待回收对象的信息

jmap -heap pid

查看堆的年轻代年老代的使用情况

jhat lijun 分析 dump 文件（现在一般不用这个了，因为分析 dump 文件非常消耗硬件资源，所以一般是复制到其他的机器来进行）现在一般都用 visualVM 来分析

4.jstack:用来显示虚拟机的线程快照

这里生成线程快照的原因主要是为了定位线程出现长时间停顿的原因，比如线程死锁，死循环，请求外部资源（数据库连接，网络资源，设备资源）导致的长时间等待

对于在 java 中最简单的死锁情况：

一个线程 T1 持有锁 L1 并且申请获得锁 L2，而另一个线程 T2 持有锁 L2 并且申请获得锁 L1，因为默认的锁申请操作都是阻塞的，所以线程 T1 和 T2 永远被阻塞了。导致了死锁。

另外一个原因是默认锁的申请操作是阻塞的

要尽量避免在一个对象的同步方法里面调用其他对象的同步方法或者延时方法。

减小锁的范围，只获对需要的资源加锁，我们锁定了完整的对象资源，但是如果我们只需要其中一个字段，那么我们应该只锁定那个特定的字段而不是完整的对象。

如果两个线程使用 thread join 无限期互相等待也会造成死锁，我们可以设定等待的最大时间来避免这种情况。

死锁的调优：

jstack 命令生成线程快照的原因主要是为了定位线程出现长时间停顿的原因，比如线程死锁，死循环，请求外部资源（数据库连接，网络资源，设备资源）导致的长时间等待

死锁的避免：

1.对多个资源、数据库表、对象同时加锁时，需要保持一致的加锁顺序，否则可能会 造成死锁。说明：线程一需要对表 A、B、C 依次全部加锁后才可以进行更新操作，那么线程二的加锁顺序也必须是 A、B、C，否则可能出现死锁。

5.jinfo 实时查看和调整虚拟机的各项参数

对于高性能硬件部署，假如我们-Xms=12g，仍然会出现每隔十几分钟就出现十几秒的停顿？

原因：就是我们在程序设计的时候，访问文档要将文档从磁盘提取到内存中，导致内存中出现很多由文档序列化产生的大对象，这些大对象都直接进入老年代，没有通过 minor gc 清除，即便有 12g 的堆，我们的内存也很快被消耗殆尽，最后进行一次 full gc，导致等待的时间很长

因此要想减少网站停顿的时间，我们就需要控制 full gc 的频率，因此就不能有成批的长时间存活的大对象产生，这样才能保证老年代空间的稳定。

堆外内存导致的溢出错误？

即使 Eden 区，Survivor 区，老年代，永久代都正常，但还是会不断抛出内存溢出异常。主要就是 direct memory 发生内存溢出异常，但是收集器一般只要在 fullgc 以后才会顺便处理 directmemory。只能在 catch 里面调用 System.gc()，这种情况一般要去代码中找 IO 操作。

外部命令导致的系统缓慢？

假如通过 shell 脚本获取系统的一些信息，java 虚拟机在执行这些命令时一般会克隆一个和当前虚拟机拥有相同环境变量的进程，再用这个新的进程去执行外部命令，频繁执行这个操作，系统的消耗会很大。

服务器 jvm 进程崩溃？

http 请求不能马上处理并返回，就会导致等待的线程和 socket 连接越来越多，最终会导致虚拟机进程崩溃。

不恰当的数据结构导致内存占用过大？

我们在新生代的垃圾收集器是采用的复制算法，如果在 survivor 区一直维持这些对象，就会导致 gc 暂停的时间非常长。可以通过设置参数去掉 survivor 区，但是仍然没有从根本上解决问题，根本还是在于 HashMap 存储数据文件效率太低。

对于 java 虚拟机的调优。

首先就是编译时间和类加载时间的优化 通过 -Xverify:none 来去掉验证码验证过程。

一般我们通过 jstat -gc 查看 java 虚拟机中堆的空间使用情况，通过 minor gc 和 full gc 的次数来判断我们的堆空间的大小设置是否合理，使用 jstat 查看 FGC 发生的频率及 FGC 所花费的时间，FGC 发生的频率越快、花费的时间越高，问题越严重；

频繁 gc 的原因一般有：

1.程序内调用了 System.gc()或 Runtime.gc().
2.一些中间件软件调用自己的 GC 方法,此时需要设置参数禁止这些 GC.
3.Java 的 Heap 太小,一般默认的 Heap 值都很小.
4.频繁实例化对象,Release 对象.此时尽量保存并重用对象,例如使用 StringBuffer()和 String()，也就是尽量少用字符串常量 String=“”。

在设置 heap 的大小的时候也不是越大越好，如果 heap 特别大，在进行 full gc 的时候所需要花费的时间也会非常的长。

也可以通过分析 gc 日志来分析 full gc 如何产生的。

也可以通过观察 cpu 资源使用情况来分析选择什么收集器。采用 CMS 收集器。

24. 问我 java 虚拟机虚拟化技术类似工具有哪些？

答：（1）平台虚拟化（PlatformVirtualization），它是针对计算机和操作系统的虚拟化，又分成服务器虚拟化和桌面虚拟化。服务器虚拟化是一种通过区分资源的优先次序，并将服务器资源分配给最需要它们的工作负载的虚拟化模式，它通过减少为单个工作负载峰值而储备的资源来简化管理和提高效率。桌面虚拟化是为提高人对计算机的操控力，降低计算机使用的复杂性，为用户提供更加方便

适用的使用环境的一种虚拟化模式。平台虚拟化主要通过 CPU 虚拟化、内存虚拟化和 I/O 接口虚拟化来实现。

（2）资源虚拟化（ResourceVirtualization），针对特定的计算资源进行的虚拟化，例如，存储虚拟化、网络资源虚拟化等。存储虚拟化是指把操作系统有机地分布于若干内外存储器，两者结合成为虚拟存储器。网络资源虚拟化最典型的是网格计算，网格计算通过使用虚拟化技术来管理网络上的数据，并在逻辑上将其作为一个系统呈现给消费者，它动态地提供了符合用户和应用程序需求的资源，同时还将提供对基础设施的共享和访问的简化。当前，有些研究人员提出利用软件代理技术来实现计算网络空间资源的虚拟化，如 Gaia，NetChaser[21]，SpatialAgent。

（3）应用程序虚拟化（ApplicationVirtualization），它包括仿真、模拟、解释技术等。Java 虚拟机是典型的在应用层进行虚拟化。基于应用层的虚拟化技术，通过保存用户的个性化计算环境的配置信息，可以实现在任意计算机上重现用户的个性化计算环境。服务虚拟化是近年研究的一个热点，服务虚拟化可以使业务用户能按需快速构建应用的需求，通过服务聚合，可屏蔽服务资源使用的复杂性，使用户更易于直接将业务需求映射到虚拟化的服务资源。现代软件体系结构及其配置的复杂性阻碍了软件开发生命周期，通过在应用层建立虚拟化的模型，可以提供最佳开发测试和运行环境。

（4）表示层虚拟化。在应用上与应用程序虚拟化类似，所不同的是表示层虚拟化中的应用程序运行在服务器上，客户机只显示应用程序的 UI 界面和用户操作。表示层虚拟化软件主要有微软的 Windows 远程桌面（包括终端服务）、CitrixMetaframePresentationServer 和 SymantecPcAnywhere 等。

1.2 虚拟化的方法通常所说的虚拟化主要是指平台虚拟化，它通过控制程序隐藏计算平台的实际物理特性，为用户提供抽象的、统一的、模拟的计算环境。通常虚拟化可以通过指令级虚拟化和系统级虚拟化来实现。

1.2.1 指令级虚拟化方法在指令集层次上实现虚拟化，即将某个硬件平台上的二进制代码转换为另一个平台上的二进制代码，实现不同指令集间的兼容，也被称作“二进制翻译”。二进制翻译是通过仿真来实现的，即在一个具有某种接口和功能的系统上实现另一种与之具有不同接口和功能的系统。二进制翻译的软件方式，它可以有 3 种方式实现：解释执行、静态翻译、动态翻译。近年来，最新的二进制翻译系统的研究主要在运行时编译、自适应优化方面，由于动态翻译和执行过程的时间开销主要包括四部分：即磁盘访问开销、存储访问开销、翻译和优化开销、目标代码的执行开销，所以要提高二进制翻译系统的效率主要应减少后 3 个方面的开销。目前典型的二进制翻译系统主要有 Daisy/BOA、Crusoe、Aeries、IA-32EL、Dynamo 动态优化系统和 JIT 编译技术等。

1.2.2 系统级虚拟化方法系统虚拟化是在一台物理机上虚拟出多个虚拟机。从系统架构看，虚拟机监控器（VMM）是整个虚拟机系统的核心，它承担了资源的调度、分配和管理，保证多个虚拟机能够相互隔离的同时运行多个客户操作系统。系统级虚拟化要通过 CPU 虚拟化、内存虚拟化和 I/O 虚拟化实现。

（1）CPU 虚拟化 CPU 虚拟化为每个虚拟机提供一个或多个虚拟 CPU，多个虚拟 CPU 分时复用物理 CPU，任意时刻一个物理 CPU 只能被一个虚拟 CPU 使用。VMM 必须为各虚拟 CPU 合理分配时间片并维护所有虚拟 CPU 的状态，当一个虚拟 CPU 的时间片用完需要切换时，要保存当前虚拟 CPU 的状态，将被调度的

虚拟 CPU 的状态载入物理 CPU。X86 的 CPU 虚拟化方法主要有：二进制代码动态翻译（dynamicbinarytranslation）、半虚拟化（para-virtualization）和预虚拟化技术。为了弥补处理器的虚拟化缺陷，现有的虚拟机系统都采用硬件辅助虚拟化技术。CPU 虚拟化需要解决的问题是：①虚拟 CPU 的正确运行，虚拟 CPU 正确运行的关键是保证虚拟机指令正确执行，各虚拟机之间不互相影响，即指令的执行结果不改变其他虚拟机的状态，目前主要是通过模拟执行和监控运行；②虚拟 CPU 的调度。虚拟 CPU 的调度是指由 VMM 决定当前哪一个虚拟 CPU 实际在物理 CPU 上运行，保证虚拟机之间的隔离性、虚拟 CPU 的性能、调度的公平。虚拟机环境的调度需求是要充分利用 CPU 资源、支持精确的 CPU 分配、性能隔离、考虑虚拟机之间的不对等、考虑虚拟机之间的依赖。常见的 CPU 调度算法有 BVT、SEDF、CB 等。

（2）内存虚拟化 VMM 通常采用分块共享的思想来虚拟计算机的物理内存。VMM 将机器的内存分配给各个虚拟机，并维护机器内存和虚拟机内存之间的映射关系，这些内存在虚拟机看来是一段从地址 0 开始的、连续的物理地址空间。在进行内存虚拟化后，内存地址将有机器地址、伪物理地址和虚拟地址三种地址。在 X86 的内存寻址机制中，VMM 能够以页面为单位建立虚拟地址到机器地址的映射关系，并利用页面权限设置实现不同虚拟机间内存的隔离和保护。为了提高地址转换的性能，X86 处理器中加入 TLB，缓存已经转换过的虚拟地址，在每次虚拟地址空间切换时，硬件自动完成切块 TLB。为了实现虚拟地址到物理地址的高效转换，通常采取复合映射的思想，通过 MMU 半虚拟化和影子页表来实现页表的虚拟化。虚拟机监控器的数据不能被虚拟机访问，因此需要一种隔离机

制，这种隔离机制主要通过修改客户操作系统或段保护来实现。内存虚拟化的优化机制，包括按需取页、虚拟存储、内存共享等。

（3）I/O 虚拟化由于 I/O 设备具有异构性强，内部状态不易控制等特点，VMM 系统针对 I/O 设备虚拟化有全虚拟化、半虚拟化、软件模拟和直接 I/O 访问等设计思路。近年来，的学者将 I/O 虚拟化的研究放在共享的网络设备虚拟化研究，提出将 IOVM 结构映射到多核心服务器平台。I/O 设备除了增加吞吐量和固有的并行数据流、联系串行特性以及基于分组的协议外，还应该考虑到传统的 PCI 兼容的 PCIExpress 的硬件，建立相应的总线适配器，以弥补象单一主机无专门的驱动程序时的需要。有些研究人员专注于外存储虚拟化的研究，提出让存储虚拟化系统上的 SCSI 目标模拟器运行在 SAN 上，存储动态的目标主机的物理信息，并使用映射表方法来修改 SCSI 命令地址，使用位图的技术来管理可用空间等思想。存储虚拟化系统应提供诸如逻辑卷大小、各种功能、数据镜像和快照，并兼容集群主机和多个操作系统。由于外存储虚拟化能全面提升存储区域网络的服务质量，而带外虚拟化与带内虚拟化相比具有性能高和扩展性好等优点，通过运用按序操作、Redo 日志以及日志完整性鉴别，设计基于关系模型的磁盘上虚拟化元数据组织方式，可以形成一致持久的带外虚拟化系统。

1.3 虚拟化的管理虚拟化的管理主要指多虚拟机系统的管理，多虚拟机系统是指在对多计算系统资源抽象表示的基础上，按照自己的资源配置构建虚拟计算系统，其主要包括虚拟机的动态迁移技术和虚拟机的管理技术。

（1）虚拟机之间的迁移将虚拟化作为一种手段管理现有的资源和加强其在网络计算的利用率，通过构建分布式可重构的虚拟机，必要时在物理服务器运行时迁

移服务。通过移动代理技术、分布式虚拟机等提高资源利用率和服务可用性，通过寻找服务最优的策略在可重构和分布式虚拟机上迁移。为了将虚拟机运行的操作系统与应用程序从一个物理结点迁移到另外一个运行结点，同时保持客户操作系统和应用程序不受干扰，有些研究者提出以数据为中心的可迁移的虚拟运行环境，使得用户操作环境实现异地迁移、无缝重构；也有研究人员提出程序执行环境的动态按需配置机制。在跨物理服务器迁移虚拟机，进行自动化的虚拟服务器的管理，必须考虑高层次的服务质量要求和资源管理成本。有些研究人员提出了通过管理程序控制的方法，以支持移动 IP 的实时迁移虚拟机在网络上，使虚拟机实时迁移其分布计算资源，从而改善迁移性能，降低网络恢复延迟，提供高可靠性和容错。有些研究机构通过设计一个通用的硬件抽象层，实现多个虚拟机的移植，具有高效率执行环境中的移动设备。虚拟机的迁移步骤一般有启动迁移、内存迁移、冻结虚拟机、虚拟机恢复执行。

（2）虚拟机的管理对于多虚拟机来说，一个非常重要的方面是减少用户对动态的和复杂的物理设备的管理和维护，通过软件和工具来实现任务管理。当前典型的多虚拟机服务器管理软件是 VirtualInfrastructure，它通过 VirtualCenter 管理服务器的虚拟机池，通过 VMotion 完成虚拟机的迁移，通过 VMFS 管理多虚拟机文件系统。其次，Parallax 是针对 Xen 的多虚拟机管理器，它通过采用消除写共享，增强客户端的缓存等方式并利用模板映像来建立整个系统；同时使用快照（snapshot）以及写时复制（copy-on-write）机制来实现块级共享，并使用副本来保证可用性。虚拟机监控器直接控制 parallax 使用的物理盘，它们运行物理设备驱动器，并给虚拟磁盘镜像 VDI 的本地虚拟机提供一个普通的块接口。2 虚拟化在制造业信息化中的应用 2.1 虚拟化在制造业信息化中的应用框

架当今制造业正朝着精密化、自动化、柔性化、集成化、网络化、信息化和智能化的方向发展，在这种趋势下，诞生了许多先进制造技术和先进制造模式。这些先进制造技术和先进制造模式要求现有的 IT 基础设施能提供更高的计算服务水平，因此在制造业信息化中，需要建立以虚拟化为导向的资源分配体系结构，提供客户驱动的服务管理和计算风险管理，维持以服务水平协议（SLA）为导向的资源分配体系。虚拟化在制造业信息化中主要用于集中 IT 管理、应用整合、工业控制、虚拟制造等。处在最底层的是制造业企业的虚拟计算资源池（VirtualCluster），它由多台物理服务器（PhysicsMachine）形成，各物理服务器上运行着虚拟化软件（VMM），虚拟化软件上运行着完成各种任务需求的虚拟机，虚拟计算资源池的虚拟化管理软件（VMS）为 IT 环境提供集中化、操作自动化、资源优化的功能，可以快速部署向导和虚拟机模板。虚拟计算资源池中的虚拟机将不同类型的客户操作系统（GuestOS）和运行其上的数据层、服务层应用程序（App）封装在一起，形成一个企业协同设计制造的完整系统，为表示层的用户提供多种形态的数据处理和显示功能。在图 1 的框架中，虚拟计算资源池的动态资源调度（DRS）模块可以跨越物理机不间断地监控资源利用率，并根据反映业务需要和不断变化的优先级的预定规则，在多个虚拟机之间分配可用资源。在制造业信息化中，集中 IT 管理、应用整合、工业控制、虚拟制造等多种应用需求都将以各种服务的形式被封装到了虚拟机中，例如制造任务协同服务、资源管理服务、信息访问服务、WWW 服务、工业控制服务、应用系统集成服务、数据管理服务、高效能计算服务、工具集服务等；同时支撑所有应用需求的数据库也被封装到了虚拟机中，例如企业模型数据库、制造资源数据库、产

品模型数据库、专业知识数据库、用户信息数据库等。虚拟化特有的优点使它能确保所有虚拟机中的关键业务连续可靠地运行。

25.问我 mysql 的调优经验？
答： 1）数据库设计要合理（遵循第三范式 3f）

2）添加索引（普通索引、主键索引、唯一索引、全文索引）

a. 普通索引（由关键字 KEY 或 INDEX 定义的索引）的唯一任务是加快对数据的访问速度。因此，应该只为那些最经常出现在查询条件（WHEREcolumn=）或排序条件（ORDERBYcolumn）中的数据列创建索引。只要有可能，就应该选择一个数据最整齐、最紧凑的数据列（如一个整数类型的数据列）来创建索引 /

create index 索引名 on 表 (列 1,列名 2);

b.主键索引 (create table bbb (id int , name varchar(32) not null default '');

ALTER TABLE tablename ADD PRIMARY KEY (列的列表))、

alter table articles drop primary key;

c.唯一索引 (这种索引和前面的“普通索引”基本相同，但有一个区别：索引列的所有值都只能出现一次，即必须唯一；

CREATE UNIQUE INDEX <索引的名字> ON tablename (列的列表)

ALTER TABLE tablename ADD UNIQUE [索引的名字] (列的列表)

create table ddd(id int primary key auto_increment , name varchar(32) unique);

)

d.全文索引(

```
CREATE TABLE articles (
       id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
       title VARCHAR(200),
       body TEXT,
       FULLTEXT (title,body)
     )engine=myisam charset utf8;

INSERT INTO articles (title,body) VALUES
     ('MySQL Tutorial','DBMS stands for DataBase ...'),
     ('How To Use MySQL Well','After you went through a ...'),
     ('Optimizing MySQL','In this tutorial we will show ...'),
     ('1001 MySQL Tricks','1. Never run mysqld as root. 2. ...'),
     ('MySQL vs. YourSQL','In the following database comparison ...'),
     ('MySQL Security','When configured properly, MySQL ...');
```

```
错误用法:
select * from articles where body like '%mysql%'; 错误用法 索引不会生效
正确用法:
select * from articles where match(title,body) against ( 'database')
说明:
在mysql中fulltext 索引只针对 myisam生效
mysql自己提供的fulltext针对英文生效->sphinx (coreseek) 技术处理中文
使用方法是 match(字段名..) against('关键字')
全文索引：停止词， 因为在一个文本中，创建索引是一个无穷大的数，因此，对一些常用词和字符，就不会创建，这些词，称为停止词.比如（a,
mysql> select match(title,body) against ('database') from articles;（输出的是每行和database的匹配度）
```

26 大数据的相关框架？

答：一、Apache Hadoop

Apache Hadoop 是一种专用于批处理的处理框架。Hadoop 是首个在开源社区获得极大关注的大数据框架。吸收了谷歌有关海量数据处理所发表的多篇论文与经验的精华，Hadoop 重新实现了相关算法和组件堆栈，让大规模批处理技术变得更易用。

Apache Hadoop 及其 MapReduce 处理引擎提供了一套久经考验的批处理模型，最适合处理对时间要求不高的非常大规模数据集。与其他框架和引擎的兼容与集成能力使得 Hadoop 可以成为使用不同技术的多种工作负载处理平台的底层基础。

二、Apache Storm

Apache Storm 是一种侧重于极低延迟的流处理框架，也许是要求近实时处理的工作负载的最佳选择。这项技术可以处理非常大的量的数据，比其他解决方案能够在更低的延迟下提供结果。

Storm 可能最适合处理延迟需求很高的纯粹的流处理工作负载。因为 Storm 可以保证每条消息都被处理，还可配合多种编程语言使用。但是 Storm 无法进行批处理，如果需要这些能力可能还需要使用其他软件。

大数据开发学习有一定难度，零基础入门首先要学习 Java 语言打基础，一般而言，Java 学习 SE、EE，需要约 3 个月的时间；然后进入大数据技术体系的学习，主要学习 Hadoop、Spark、Storm 等，从零基础到精通学习大数据 qq 群：606859705，分享大数据学习资源，有大佬指导学习，学习路线清晰。

三、Apache Samza

Apache Samza 是一种与 Apache Kafka 消息系统紧密绑定的流处理框架。按照设计，相较于 Kafka，Samza 可以更好地发挥 Kafka 独特的架构优势和保障。Samza 能通过 Kafka 提供容错、缓冲，以及状态存储。

Samza 可使用 YARN 作为资源管理器。这意味着默认情况下需要具备 Hadoop 集群（至少具备 HDFS 和 YARN），但同时也意味着 Samza 可以直接使用 YARN 丰富的内建功能。

在已经有 Hadoop 和 Kafka 的环境，Apache Samza 是流处理工作负载一个很好的选择。

四、Apache Spark

Apache Spark 是一种包含流处理能力的新一代批处理框架。和 Hadoop 的 MapReduce 引擎基于各种相同原则开发而来的 Spark 主要侧重于通过完善的内存计算和处理优化机制加快批处理工作负载的运行速度。

Spark 可作为独立集群部署（需要相应存储层的配合），或可与 Hadoop 集成并取代 MapReduce 引擎。

多样化工作负载处理任务的最佳选择是 Spark。Spark 批处理能力以更高内存占用为代价，也提供了其他框架难以达到的速度优势。对于重视吞吐率而非延迟的工作负载， Spark Streaming 更适合作为流处理解决方案。

五、Apache Flink

Apache Flink 是能处理批处理任务的流处理框架。这项技术能将批处理数据视作具备有限边界的数据流，然后借此将批处理任务当做流处理的子集来处理。把所有处理任务优先采取流处理，会产生一系列有趣的副作用。

Flink 可提供低延迟流处理，与此同时还支持传统的批处理任务。对于有极高流处理需求，并有少量批处理任务的组织的任务，Flink 也许是最适合的。

27.TCP 和 UDP 有什么区别？

答：TCP 和 UDP 是 OSI 模型中的运输层中的协议。TCP 提供可靠的通信传输，而 UDP 则常被用于广播和细节控制交给应用的通信传输

UDP(User Datagram Protocol)

UDP 不提供复杂的控制机制，利用 IP 提供面向无连接的通信服务。并且它是将应用程序发来的数据在收到的那一刻，立刻按照原样发送到网络上的一种机制。 即使是出现网络拥堵的情况下，UDP 也无法进行流量控制等避免网络拥塞的行为。此外，传输途中如果出现了丢包，UDO 也不负责重发。甚至当出现包的到达顺序乱掉时也没有纠正的功能。如果需要这些细节控制，那么不得不交给由采用 UDO 的应用程序去处理。换句话说，UDP 将部分控制转移到应用程序去处理，自己却只提供作为传输层协议的最基本功能。UDP 有点类似于用户说什么听什么的机制，但是需要用户充分考虑好上层协议类型并制作相应的应用程序。

TCP(Transmission Control Protocol)

TCP 充分实现了数据传输时各种控制功能，可以进行丢包的重发控制，还可以对次序乱掉的分包进行顺序控制。而这些在 UDP 中都没有。此外，TCP 作为一种面向有连接的协议，只有在确认通信对端存在时才会发送数据，从而可以控制通信流量的浪费。TCP 通过检验、序列号、确认应答、重发控制、连接管理以及窗口控制等机制实现可靠性传输。

一、TCP 协议与 UDP 协议的区别

首先咱们弄清楚，TCP 协议和 UCP 协议与 TCP/IP 协议的联系，一直都是说 TCP/IP 协议与 UDP 协议的区别，是没有从本质上弄清楚网络通信

TCP/IP 协议是一个协议簇。里面包括很多协议的。UDP 只是其中的一个

TCP/IP 协议集包括应用层,传输层，网络层，网络访问层

应用层包括:

超文本传输协议(HTTP):万维网的基本协议

文件传输(TFTP 简单文件传输协议)

远程登录(Telnet),提供远程访问其它主机功能,它允许用户登录

internet 主机,并在这台主机上执行命令.

网络管理(SNMP 简单网络管理协议),该协议提供了监控网络设备的方法,以及配置管理,统计信息收集,性能管理及安全管理等.

域名系统(DNS),该系统用于在 internet 中将域名及其公共广播的网络节点转换成 IP 地址

网络层包括:

Internet 协议(IP)

Internet 控制信息协议(ICMP)

地址解析协议(ARP)

反向地址解析协议(RARP)

网络访问层:

网络访问层又称作主机到网络层(host-to-network).网络访问层的功能包括 IP 地址与物理地址硬件的映射,以及将 IP 封装成帧.基于不同硬件类型的网络接口,网络访问层定义了和物理介质的连接.

下面讲解一下 TCP 协议和 UDP 协议的区别

TCP（Transmission Control Protocol，传输控制协议）是面向连接的协议，也就是说，在收发数据前，必须和对方建立可靠的连接。一个 TCP 连接必须要经过三次"对话"才能建立，其中的过程非常复杂，过程：主机 A 向主机 B 发出连接请求数据包："我想给你发数据，可以吗？"，这是第一次对话；主机 B 向主机 A

发送同意连接和要求同步（同步就是两台主机一个在发送，一个在接收，协调工作）的数据包："可以，你什么时候发？"，这是第二次对话；主机 A 再发出一个数据包确认主机 B 的要求同步："我现在就发，你接着吧！"，这是第三次对话。三次"对话"的目的是使数据包的发送和接收同步，经过三次"对话"之后，主机 A 才向主机 B 正式发送数据。

TCP 建立连接要进行 3 次握手

1）主机 A 通过向主机 B 发送一个含有同步序列号的标志位的数据段给主机 B ,向主机 B 请求建立连接,通过这个数据段,主机 A 告诉主机 B 两件事:我想要和你通信;你可以用哪个序列号作为起始数据段来回应我

. 2）主机 B 收到主机 A 的请求后,用一个带有确认应答(ACK)和同步序列号(SYN)标志位的数据段响应主机 A,也告诉主机 A 两件事:我已经收到你的请求了,你可以传输数据了;你要用序列号作为起始数据段来回应我

3)主机 A 收到这个数据段后,再发送一个确认应答,确认已收到主机 B 的数据段:"我已收到回复,我现在要开始传输实际数据了

3 次握手就完成了,主机 A 和主机 B 就可以传输数据

3 次握手的特点:

没有应用层的数据

SYN 这个标志位只有在 TCP 建产连接时才会被置 1

握手完成后 SYN 标志位被置 0

TCP 断开连接要进行 4 次

1）当主机 A 完成数据传输后,将控制位 FIN 置 1,提出停止 TCP 连接的请求

2）主机 B 收到 FIN 后对其作出响应,确认这一方向上的 TCP 连接将关闭,将 ACK 置 1

3）由 B 端再提出反方向的关闭请求,将 FIN 置 1

4）主机 A 对主机 B 的请求进行确认,将 ACK 置 1,双方向的关闭结束.

由 TCP 的三次握手和四次断开可以看出,TCP 使用面向连接的通信方式,大大提高了数据通信的可靠性,使发送数据端

和接收端在数据正式传输前就有了交互,为数据正式传输打下了可靠的基础

名词解释

ACK TCP 报头的控制位之一,对数据进行确认.确认由目的端发出,用它来告诉发送端这个序列号之前的数据段都收到了.比如,确认号为 X,则表示前 X-1 个数据段都收到了,只有当 ACK=1 时,确认号才有效,当 ACK=0 时,确认号无效,这时会要求重传数据,保证数据的完整性.

SYN 同步序列号,TCP 建立连接时将这个位置 1

FIN 发送端完成发送任务位,当 TCP 完成数据传输需要断开时,提出断开连接的一方将这位置 1

TCP 的包头结构:

源端口 16 位

目标端口 16 位

序列号 32 位

回应序号 32 位

TCP 头长度 4 位

reserved 6 位

控制代码 6 位

窗口大小 16 位

偏移量 16 位

校验和 16 位

选项 32 位(可选)

这样我们得出了 TCP 包头的最小长度，为 20 字节。

UDP（User Data Protocol，用户数据报协议）

（1） UDP 是一个非连接的协议，传输数据之前源端和终端不建立连接，当它想传送时就简单地去抓取来自应用程序的数据，并尽可能快地把它扔到网络上。在发送端，UDP 传送数据的速度仅仅是受应用程序生成数据的速度、计算机的能力和传输带宽的限制；在接收端，UDP 把每个消息段放在队列中，应用程序每次从队列中读一个消息段。

（2） 由于传输数据不建立连接，因此也就不需要维护连接状态，包括收发状态等，因此一台服务机可同时向多个客户机传输相同的消息。

（3） UDP 信息包的标题很短，只有 8 个字节，相对于 TCP 的 20 个字节信息包的额外开销很小。

（4） 吞吐量不受拥挤控制算法的调节，只受应用软件生成数据的速率、传输带宽、源端和终端主机性能的限制。

（5）UDP 使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的链接状态表（这里面有许多参数）。

（6）UDP 是面向报文的。发送方的 UDP 对应用程序交下来的报文，在添加首部后就向下交付给 IP 层。既不拆分，也不合并，而是保留这些报文的边界，因此，应用程序需要选择合适的报文大小。

我们经常使用“ping”命令来测试两台主机之间 TCP/IP 通信是否正常，其实“ping”命令的原理就是向对方主机发送 UDP 数据包，然后对方主机确认收到数据包，如果数据包是否到达的消息及时反馈回来，那么网络就是通的。

UDP 的包头结构：

源端口 16 位

目的端口 16 位

长度 16 位

校验和 16 位

小结 TCP 与 UDP 的区别：

1.基于连接与无连接；

2.对系统资源的要求（TCP 较多，UDP 少）；

3.UDP 程序结构较简单；

4.流模式与数据报模式 ；

5.TCP 保证数据正确性，UDP 可能丢包，TCP 保证数据顺序，UDP 不保证。

UDP 应用场景：

1.面向数据报方式

2.网络数据大多为短消息

3.拥有大量 Client

4.对数据安全性无特殊要求

5.网络负担非常重，但对响应速度要求高

TCP:

TCP 编程的服务器端一般步骤是：

1、创建一个 socket，用函数 socket()；

2、设置 socket 属性，用函数 setsockopt(); * 可选

3、绑定 IP 地址、端口等信息到 socket 上，用函数 bind();
4、开启监听，用函数 listen()；
5、接收客户端上来的连接，用函数 accept()；
6、收发数据，用函数 send()和 recv()，或者 read()和 write();
7、关闭网络连接；
8、关闭监听；

TCP 编程的客户端一般步骤是：
1、创建一个 socket，用函数 socket()；
2、设置 socket 属性，用函数 setsockopt();* 可选
3、绑定 IP 地址、端口等信息到 socket 上，用函数 bind();* 可选
4、设置要连接的对方的 IP 地址和端口等属性；
5、连接服务器，用函数 connect()；
6、收发数据，用函数 send()和 recv()，或者 read()和 write();
7、关闭网络连接；

UDP:

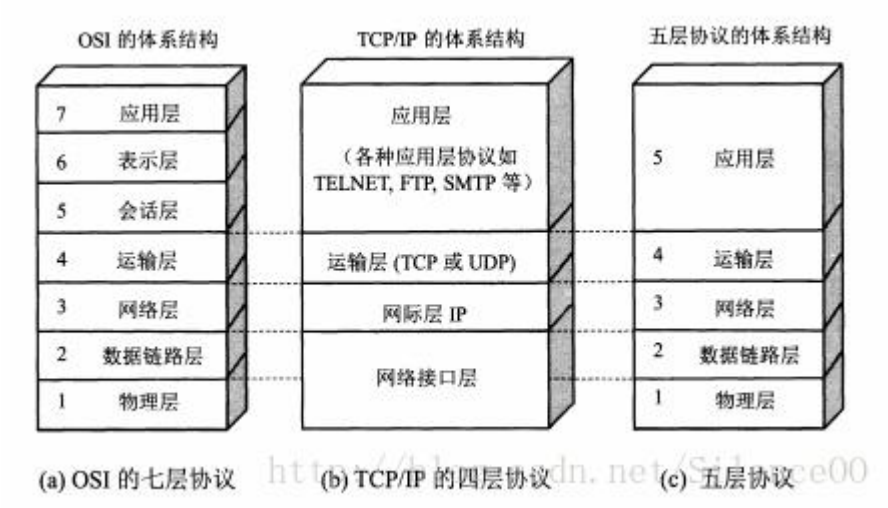
与之对应的 UDP 编程步骤要简单许多，分别如下：
UDP 编程的服务器端一般步骤是：
1、创建一个 socket，用函数 socket()；
2、设置 socket 属性，用函数 setsockopt();* 可选
3、绑定 IP 地址、端口等信息到 socket 上，用函数 bind();
4、循环接收数据，用函数 recvfrom();
5、关闭网络连接；

UDP 编程的客户端一般步骤是：
1、创建一个 socket，用函数 socket()；
2、设置 socket 属性，用函数 setsockopt();* 可选
3、绑定 IP 地址、端口等信息到 socket 上，用函数 bind();* 可选
4、设置对方的 IP 地址和端口等属性;
5、发送数据，用函数 sendto();
6、关闭网络连接；

28.简述 TCP/IP 五层协议

答：五层体系结构包括：应用层、运输层、网络层、数据链路层和物理层。
五层协议只是 OSI 和 TCP/IP 的综合，实际应用还是 TCP/IP 的四层结构。为了方便可以把下两层称为网络接口层。

三种模型结构：



OSI七层网络模型	TCP/IP四层概念模型	对应网络协议
应用层（Application）	应用层	HTTP、TFTP, FTP, NFS, WAIS、SMTP
表示层（Presentation）		Telnet, Rlogin, SNMP, Gopher
会话层（Session）		SMTP, DNS
传输层（Transport）	传输层	TCP, UDP
网络层（Network）	网络层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层（Data Link）	数据链路层	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层（Physical）		IEEE 802.1A, IEEE 802.2到IEEE 802.11

29.linux dump，还有一个文件什么忘了

答：Linux dump 命令用于备份文件系统。

dump 为备份工具程序，可将目录或整个文件系统备份至指定的设备，或备份成一个大文件。

语法

```
dump [-cnu][-0123456789][-b <区块大小>][-B <区块数目>][-d <密度>][-f <设备名称>][-h <层级>][-s <磁带长度>][-T <日期>][目录或文件系统] 或 dump [-wW]
```

参数：

- -0123456789　备份的层级。
- -b<区块大小>　指定区块的大小，单位为 KB。
- -B<区块数目>　指定备份卷册的区块数目。

- -c 修改备份磁带预设的密度与容量。
- -d<密度> 设置磁带的密度。单位为 BPI。
- -f<设备名称> 指定备份设备。
- -h<层级> 当备份层级等于或大于指定的层级时，将不备份用户标示为"nodump"的文件。
- -n 当备份工作需要管理员介入时，向所有"operator"群组中的使用者发出通知。
- -s<磁带长度> 备份磁带的长度，单位为英尺。
- -T<日期> 指定开始备份的时间与日期。
- -u 备份完毕后，在/etc/dumpdates 中记录备份的文件系统，层级，日期与时间等。
- -w 与-W 类似，但仅显示需要备份的文件。
- -W 显示需要备份的文件及其最后一次备份的层级，时间与日期。

实例

备份文件到磁带

```
# dump -0 -u /dev/tape /home/
```

其中"-0"参数指定的是备份等级"-u"要求备份完毕之后将相应的信息存储到文件 /etc/dumpdates 留作记录

2.2 Android

Activity 之类，生命周期等。举例子讲

答：它的生命周期也分为两种情况：

第一：正常情况下的生命周期

第二：非正常情况下的生命周期：比如屏幕翻转或者内存不足，被 kill 掉了。

一、 Activity 有哪些生命周期方法

先来过滤一下，一共有哪些生命周期的方法吧！

onCreate：表示窗口正在被创建，比如加载 layout 布局文件啊（setContentView）。 所以我们可以在这个方法中，做一些初始化的操作。

onStart：表示 Activity 正在被启动，即将开始，此时的窗口已经可见了，但是还没有出现在前台，所以无法和用户进行交互。也就是说此时的窗口正处在 不可见—>可见 的过程中。

onRestart：表示窗口正在重新启动。在什么场景下会调用这个呢？比如：从 A 页面进入 B 页面，然后点击 BACK 键（或者自己的返回上一页的按钮）回到 A 页面，那么就会调用 A 页面的 onRestart 方法了。。（当前也牵扯到 A 和 B 页面的其他生命周期方法的调用，这个我们后面再详细说明）

再比如：点击 HOME 键回到桌面，然后通过点击任务栏或者点击应用图标再次进入 A 页面，都可以触发调用这个方法

onResume：表示此时的窗口已经可见了，显示在前台并且进行活动了，我们也可以与窗口进行交互了。

onPause：表示窗口正在停止，这是我们可以做一些存储数据、或者停止动画等一些不太耗时的操作，因为会影响到下一个 Activity 的显示。onPause 执行完成之后，新的 Activity 的 onResume 才会执行。

onStop：表示窗口即将停止，此时，可以做一些稍微重量级的回收工作，但是也不能太耗时哈。

onDestroy：表示窗口即将被销毁。这是 Activity 生命周期中的最后一步了。这里，我们可以做一些回收工作和最终的资源释放工作。

下面，我们暴露一张图，来详细的描述一下窗口的生命周期的切换过程： 图片来源任玉刚的《Android 开发艺术探索》

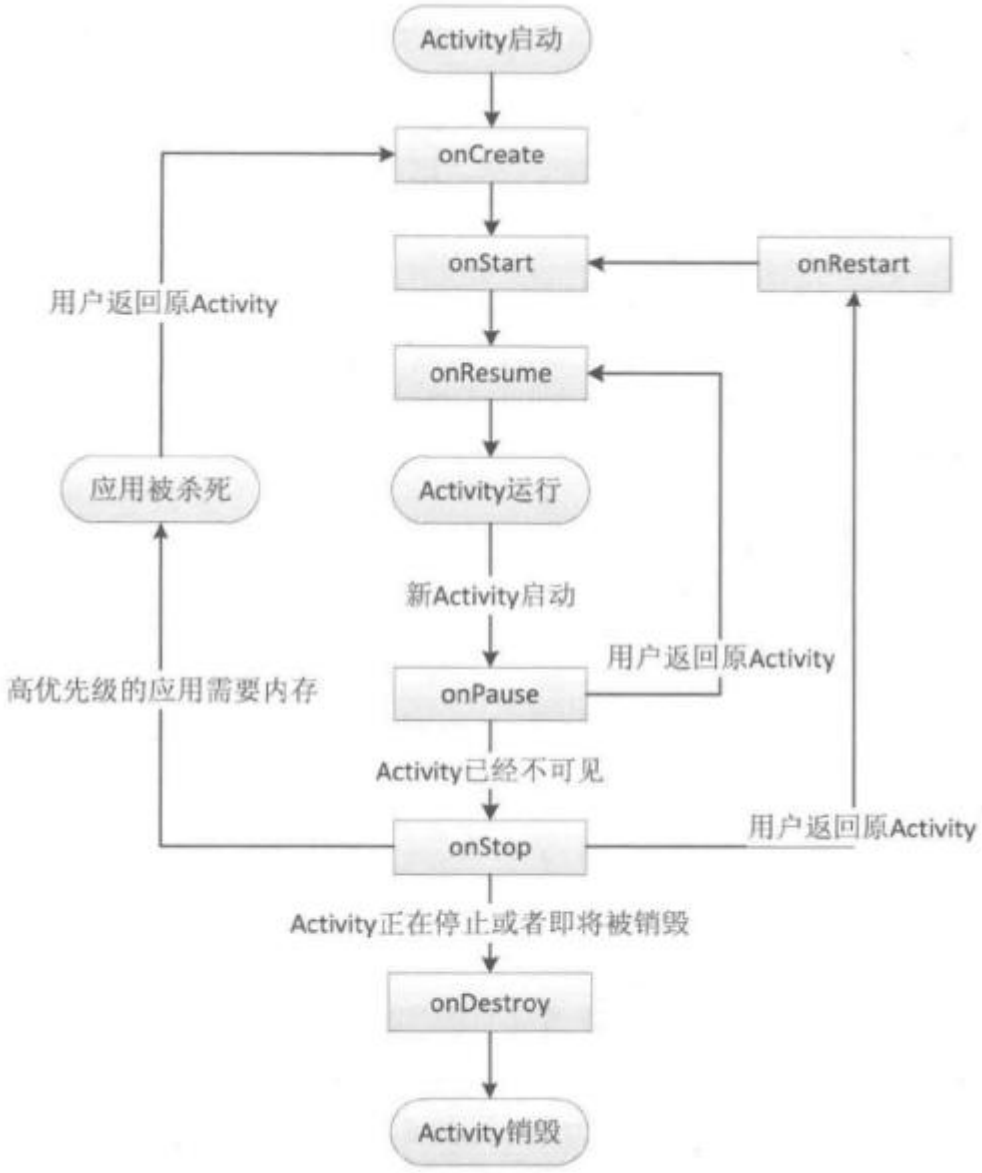


图 1-1 Activity 生命周期的切换过程

二、正常情况下，Activity 的生命周期调用过程

然后我们来梳理一下几种情况下，生命周期的方法调用顺序是怎么样的。

启动一个特定的 Activity 时，第一次启动，生命周期回调如下：

onCreate -> onStart -> onResume

当用户打开新的 Activity 或者切换到桌面的时候，回调如下：

onPause -> onStop

但是有一种特殊的情况，如果新的 Activity 采用了透明的主题，那么当前 Activity 不会回调 onStop

当用户再次回到原 Activity 时，回调如下：

onRestart -> onStart -> onResume

当用户点击 Back 键回退时，回调：

onPause -> onStop -> onDestory

当 Activity 被系统回收后再次打开时，回调 onCreate -> onStart -> onResume（和过程 1 一样，但是只是生命周期的回调过程一样，这里还是有一些其他过程的区别的，比如回收时 onSaveInstanceState 的方法的调用，以及打开时 onRestoreInstanceState 的调用）

在生命周期中，onCreate 和 onDestroy 是对应的，创建和销毁，并且在生命周期过程中，只被调用一次。从 Activity 是否可见来说，onStart 和 onStop 是配对的，这两个方法有可能会被多次调用。从 Activity 是否在前台来说，onResume 和 onPause 是配对的，也有可能会被多次调用。

其中有两个问题：

第一：onStart 和 onResume，onPause 和 onStop 差不对，但是又有什么本质的不同呢？

第二：当从 A 窗口打开 B 窗口时，B 窗口的 onResume 和 A 的 onPause 方法哪个先执行呢？

首先第一个问题：

从上面的描述中就知道了，虽然差不多，但是角度还是不通的。onStart 和 onStop 从 Activity 是否可见来说的，onResume 和 onPause 从 Activity 是否在前台来说的。其他的就没有什么区别了。

第二问题：

自己实践一下就知道了，先是 A 的 onPause 方法调用结束之后，才会执行 B 窗口的 onResume 方法。当然还是从源码了解的更彻底。。。请参考任玉刚的《Android 开发艺术探索》或者自己去 AndroidXRef 看吧。

三、非正常情况下，Activity 的生命周期调用过程

非正常情况下，比如屏幕的旋转，或者内存不够用的情况下，由系统的回收操作造成的非前台的 Activity 被意外杀死的情况。

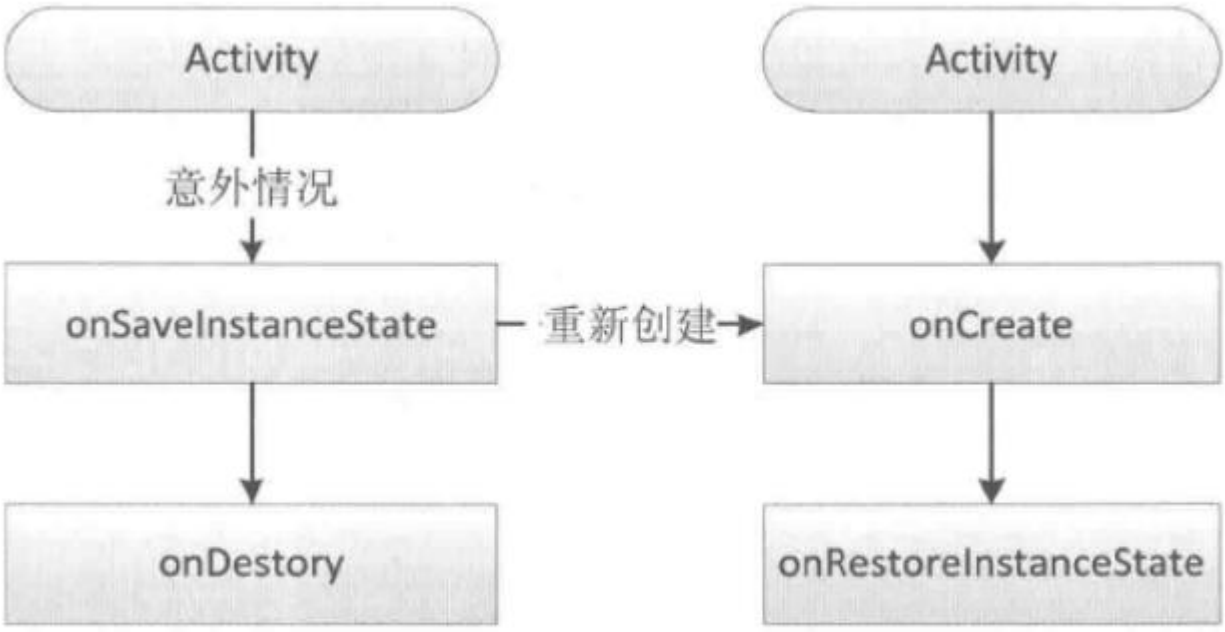


图 1-3　异常情况下 Activity 的重建过程

图片来源：任玉刚的《Android 开发艺术探索》

Activity 四种启动模式 ？

答： 1.standard：该启动模式为 Android 默认启动模式，每当启动一个 activity 就会在任务栈中创建一个 activity，这种模式默认的但是非常的浪费空间，但是可以有效的保存之前启动的 activity。用于保证之前页面不丢失的时候。

2.single Top：该启动模式是在查看任务栈顶和你将要启动的 activity 是否是一个 activity，是一个就直接复用，否则就新创一个实例，这个经常用于类似聊天界面，当有人给你发消息时更新 activity。

3.single Task：该启动模式是在任务栈中看是否有和你一样的 activity，有则直接把该 activity 之上的 activity 全部弹出使之置于栈顶。常用于一个程序的入口处。

4.single Instence：该启动模式是再创建一个任务栈把 activity 放进去。此模式用于不同应用调用一个 activity 时应用。用于程序和界面分开的时候。

生命周期，两个 activity 启动的回调。然后问如果把一个应用进程直接 kill 掉，生命周期会怎么走？

答：

singletask 方式启动 activity 时的生命周期？

答： 当 Activity 启动模式为 singleTask 时的生命周期

01-09 22:59:24.317 13063-13063/com.dongua.activitytest I/Activity1: onCreate:

01-09 22:59:24.319 13063-13063/com.dongua.activitytest I/Activity1: onStart:

01-09 22:59:24.321 13063-13063/com.dongua.activitytest I/Activity1: onResume:

01-09 22:59:29.501 13063-13063/com.dongua.activitytest I/Activity1: onStop:

01-09 22:59:37.846 13063-13063/com.dongua.activitytest I/Activity1: onRestart:

01-09 22:59:37.846 13063-13063/com.dongua.activitytest I/Activity1: onStart:

01-09 22:59:37.847 13063-13063/com.dongua.activitytest I/Activity1: onResume:

即再次启动会调用 onRestart-onStart-onResume

当 2 个都为 singleTask 的 Activity 互相跳转时，则会出现如下情况

activity2 跳转到 activity1 activity1 再跳转到 activity2 后边 activity1 执行了 onDestroy 方法被销毁了

01-09 23:24:37.547 7445-7445/com.dongua.activitytest I/Activity12: Activity2+onCreate:

01-09 23:24:37.548 7445-7445/com.dongua.activitytest I/Activity12: Activity2+onStart:

01-09 23:24:37.550 7445-7445/com.dongua.activitytest I/Activity12: Activity2+onResume:

01-09 23:24:40.636 7445-7445/com.dongua.activitytest I/Activity12: Activity1+onCreate:

01-09 23:24:40.638 7445-7445/com.dongua.activitytest I/Activity12: Activity1+onStart:

01-09 23:24:40.639 7445-7445/com.dongua.activitytest I/Activity12: Activity1+onResume:

01-09 23:24:40.971 7445-7445/com.dongua.activitytest I/Activity12: Activity2+onStop:

01-09 23:24:52.602 7445-7445/com.dongua.activitytest I/Activity12: Activity2+onRestart:

01-09 23:24:52.603 7445-7445/com.dongua.activitytest I/Activity12: Activity2+onStart:

01-09 23:24:52.604 7445-7445/com.dongua.activitytest I/Activity12: Activity2+onResume:

01-09 23:24:52.960 7445-7445/com.dongua.activitytest I/Activity12: Activity1+onStop:

01-09 23:24:52.960 7445-7445/com.dongua.activitytest I/Activity12: Activity1+onDestroy:

进程通信机制有哪些？。

答：IPC:每个进程都有自己独立的进程地址空间，任何在一个经常的全局变量在另一个进程中是看不见的，所以进程之间的数据交换是通过内核缓冲区实现的，一个进程先将数据从用户态拷贝到内核的缓冲区，然后另一个进程从内核缓冲区读取到用户态。内核提供的这种机制叫作进程间的通信（IPC：InterProcessCom

- System V IPC 与 POSIX IPC：

1. System V IPC 存在时间比较老，许多系统都支持，但是接口复杂，并且可能各平台上实现略有区别（如 ftok 的实现及限制），以为都是基于 systemV IPC 函数实现的。
2. POSIX 是新标准，现在多数 UNIX 也已实现，我觉得如果只是开发的话，那么还是 POSIX 好，因为语法简单，并且各平台上实现都一样。

SYSTEM V	POSIX
AT & T(1983)介绍了IPC设施的三种新形式，即消息队列，共享内存和信号量。	由IEEE指定的便携式操作系统接口标准来定义应用程序编程接口(API)。POSIX涵盖了所有三种形式的IPC
SYSTEM V IPC涵盖了所有的IPC机制，即管道，命名管道，消息队列，信号，信号量和共享内存。它还涵盖套接字和Unix域套接字。	几乎所有的基本概念都与系统V相同。它仅与接口有所不同。
共享内存接口调用 shmget() ， shmat() ， shmdt() ， shmctl()	共享内存接口调用 shm_open() ， mmap() ， shm_unlink()
消息队列接口调用 msgget() ， msgsnd() ， msgrcv() ， msgctl()	消息队列接口调用 mq_open() ， mq_send() ， mq_receive() ， mq_unlink()
信号量接口调用 semget() ， semop() ， semctl()	信号量接口调用命名信号量 sem_open() ， sem_close() ， sem_unlink() ， sem_post() ， sem_wait() ， sem_trywait() ， sem_timedwait() ， sem_getvalue() 未命名或基于内存的信号量 sem_init() ， sem_post() ， sem_wait () ， sem_getvalue() ， sem_destroy()
使用键和标识符来标识IPC对象。	使用名称和文件描述符来标识IPC对象
NA	可以使用 select() ， poll() 和epoll API来监视POSIX消息队列
提供 msgctl() 调用	提供函数(mq_getattr() 和 mq_setattr())来访问或设置属性
NA	多线程安全。包含线程同步函数，如互斥锁，条件变量，读写锁等
NA	为消息队列提供少量通知功能(如 mq_notify())
需要系统调用如 shmctl() ，命令(ipcs ， ipcrm)来执行状态/控制操作。	共享内存对象可以使用系统调用(如fstat()，fchmod()
System V共享内存段的大小在创建时是固定的(通过 shmget())	可以使用 ftruncate() 来调整底层对象的大小，然后使用 munmap() 和 mmap() (或Linux专用的 mremap())重新创建映射

AIDL 原理是什么？

答： Android 系统中的进程之间不能共享内存，因此，需要提供一些机制在不同进程之间进行数据通信。

为了使其他的应用程序也可以访问本应用程序提供的服务，Android 系统采用了远程过程调用（Remote Procedure Call，RPC）方式来实现。与很多其他的基于 RPC 的解决方案一样，Android 使用一种接口定义语言（Interface Definition Language，IDL）来公开服务的接口。我们知道 4 个 Android 应用程序组件中的 3 个（Activity、BroadcastReceiver 和 ContentProvider）都可以进行跨进程访问，另外一个 Android 应用程序组件 Service 同样可以。因此，可以将这种可以跨进程访问的服务称为 AIDL（Android Interface Definition Language）服务。

建立 AIDL 服务的步骤

建立 AIDL 服务要比建立普通的服务复杂一些，具体步骤如下：

（1）在 Eclipse Android 工程的 Java 包目录中建立一个扩展名为 aidl 的文件。该文件的语法类似于 Java 代码，但会稍有不同。

（2）如果 aidl 文件的内容是正确的，ADT 会自动生成一个 Java 接口文件（*.java）。

（3）建立一个服务类（Service 的子类）。

（4）实现由 aidl 文件生成的 Java 接口。

（5）在 AndroidManifest.xml 文件中配置 AIDL 服务，尤其要注意的是，<action>标签中 android:name 的属性值就是客户端要引用该服务的 ID，也就是 Intent 类的参数值。

实现 AIDL 接口的说明：

（1）AIDL 接口只支持方法，不能声明静态成员；

（2）不会有返回给调用方的异常。

线程通信，handler 机制，讲完流程后问当 messagequeue 空的时候，怎么进行阻塞的（面试官跟我讲涉及到 native 方法）。postdelay 详细讲一下，跟前面一个问题有关联。

答：

asynctask 内部机制讲一讲，onpost 回调执行的线程

答：

内存泄漏检测方案，常见内存泄漏有哪些。我自己举了举个例子，面试官又补充了好多。

答：

view 的 60ms 刷新机制了解，怎么检测 view 卡顿。

答：

anr 定位，监控。

答：

事件分发，讲完流程之后问的比较深入。如何让事件进行双向传递，比如子 view 已经消费了的事件怎么传回到 vg 去。

- Fragment 的生命周期
- Fragment 相比 Activity 有什么优劣
- Fragment 和 Activity 的交互
- Fragment 的使用场景
- 本地广播的用法
- 本地广播的原理
- 如何让广播只发送给特定的 App
- Intent 如何传递大文件
- Intent 传递超出范围的文件的后果
- 如何传递大文件
- 如何避免大图片 OOM
- 图片压缩的原理
- Glide 源码
- 图片的三级缓存
- Lru 算法
- EventBus 原理
- RemoteView 使用场景及其方法
- RemoteView 能使用的 View 的范围
- Notification 能否使用自定义 View 以及原因
- RemoteView 原理
- 简述事件分发机制
- 事件分发中的方法名、方法参数以及不同的返回值的意义
- ListView 中如何使不同的 Item 加载不同的样式
- ListView 中 convertview 的作用以及意义
- RecyclerView 与 ListView 的区别
- 如何解决 ListView 异步加载图片的问题
- ContentProvider 的用法,详细到方法
- 跨进程通信的方式
-
- Binder 原理,详细到方法
- 启动一个 Server 和绑定 Server 的区别
- 举例 Server 和绑定 Server 的不同使用场景

- IntentServer 与普通 Server 的区别
- IntentServer 的使用场景
- IntentServer 原理
- 如何保证 Server 不被杀死
- 实现多线程的几种方法
- 几种动画
- 如何自己去实现一个动画,具体到方法
- 属性动画的原理

手写快排

答：

最大子序列
答：

Fragment 的生命周期

答：

Fragment 相比 Activity 有什么优劣

答：

Fragment 和 Activity 的交互

答：

Fragment 的使用场景

答：

本地广播的用法

答：

本地广播的原理

答：

如何让广播只发送给特定的 App

答：

Intent 如何传递大文件

答：

Intent 传递超出范围的文件的后果

答：

如何传递大文件？

答：

如何避免大图片 OOM？

答：

图片压缩的原理？

答：

String、StringBuilder、StringBuffer 有什么区别？

答：

看过哪些开源库源码，

答：我说看过 eventbus 和 glide。

Glide 源码

答：

图片的三级缓存

答：

Lru 算法

答：

说说安卓分四大组件这种模块的原因。

答：

手写读者写者问题

答：

jpg、png、webp 区别和转换

答：有损压缩和无损压缩

有损(不可逆，损失部分图片的信息，按照一定的算法将临近的像素点进行合并)

索引色和直接色

索引：256 种，用一个字节的数字来索引一种颜色。

直接：四个数字表示，红绿蓝以及透明度。

点阵图和矢量图

点阵图：位图，像素图。缩放会失真

矢量图：也叫向量图。缩放不失真。并不记录画面上每一点的信息，而是记录了元素形状及颜色的算法，当打开的时候，软件对图像对应的函数进行计算，将运算结果显示出来。

BMP： 是无损的、及支持索引色也支持直接色的点阵图。较大。

jpg：有损的采用直接色的点阵图。

png-8：无损的，使用索引色的点阵图。具有更小的体积，支持透明度的调节。

png-24： 无损的，使用直接色的点阵图。体积较大，支持透明度的调节。

Webp： 支持有损和无损、使用直接色的点阵图。体积更小，支持透明度的调节。

在无损压缩的情况下，相同质量的 WebP 图片，文件大小要比 PNG 小 26%；

在有损压缩的情况下，具有相同图片精度的 WebP 图片，文件大小要比 JPEG 小 25%~34%；

WebP 图片格式支持图片透明度，一个无损压缩的 WebP 图片，如果要支持透明度只需要 22%的格外文件大小。

如何解决 ListView 异步加载图片的问题

答：

sharedpreferences 的 apply/commit 方法区别? ANR

答：1. apply()没有返回值而 commit()返回 boolean 表明修改是否提交成功。

2. apply()是将修改数据院子操作提交到内存，而后异步真正提交到硬盘，而多个并发的提交 commit()的时候，他们会等待正在处理的 commit()保存到磁盘后再操作，从而降低了效率。apply()效率高一些。
3. apply()方法不会提示任何失败的信息。
4. 由于在一个进程中，sharedpreferences 时单示例，一半不会出现并发冲突，如果对提交的结果不关系的话，可以使用 apply(),如果需要确保提交成功而且有后续操作的话，还是需要使用 commit()的。

git merge / git cherry-pick 提交到一个分支之后，提交到另一个分支

答：A ———> B (将 A 分支 merge 到 B 分支)

git checkout A

git pull

git checkout B

git merge A

(提交到一个分支之后，提交到另一个分支)

git checkout 另一个分支

git cherry-pick

打包流程、优化（减小包大小）、lint、混淆原因

答：

RemoteView 使用场景及其方法

答：

RemoteView 能使用的 View 的范围

答：

Notification 能否使用自定义 View 以及原因

答：

RemoteView 原理

答：

简述事件分发机制

答：

事件分发中的方法名、方法参数以及不同的返回值的意义

答：

ListView 中如何使不同的 Item 加载不同的样式

答：

ListView 中 convertview 的作用以及意义

答：

RecyclerView 与 ListView 的区别

答：

listview/recyclerview 显示不同样式的 item

答：ListView：重写 getItemViewType()、getViewTypeCount()。getView()的时候，根据类型加载。

RecyclerView：重写 onCreateViewHolder()、onBindViewHolder()、getItemViewType()。根据类型自定义 viewholder。

dex 怎么能让方法不被 65535 限制

答：

Activity、DecorView、Window？

答：

自定义 view 混淆后，构造方法会少吗？

答：自定义 view 默认不会被混淆

三个构造方法

答： public View (Context context)是在 java 代码创建视图的时候被调用，如 果是从 xml 填充的视图，就不会调用这个

public View (Context context, AttributeSet attrs)这个是在 xml 创建但是没有指定 style 的时候被调用

public View (Context context, AttributeSet attrs, int defStyle)给 View 提供一个基本的 style，如果我们没有对 View 设置某些属性，就使用这个 style 中的属性

EventBus 和 handler 有什么区别？

答：

EventBus 原理 ？

答：

EventBus 注册两次有什么后果

答：注册一般是在 onCreate 和 onStart 里注册，尽量不要在 onResume，可能出现多次注册的情况。

EventBusException: Subscriber class already registered to event class

可以先判断：

if (!EventBus.getDefault().isRegistered(this)) { EventBus.getDefault().register(this);

}

取消注册要写到 onDestroy,不要写到 onStop 里。

不要盲目相信官方文档，官方文档写的是在 onStart()和 onStop()里面注册和取消注册。但是实际使用中，Activity 的这两个生命周期可能会因为各种跳转而执行多次，所以不太适合做这样的事情。

ContentProvider 的用法,详细到方法

答：

跨进程通信的方式

答：

service 和 AIDL 原理

答：

Binder 原理,详细到方法

答：

启动一个 Server 和绑定 Server 的区别

答：

举例 Server 和绑定 Server 的不同使用场景

答：

IntentServer 与普通 Server 的区别

答：

IntentServer 的使用场景

答：

IntentServer 原理

答：

如何保证 Server 不被杀死

答：

实现多线程的几种方法

答：

几种动画
答：

如何自己去实现一个动画,具体到方法
答：
属性动画的原理
答：
内存泄漏分析方式
答：MAT、android Monitor 等工具
ANR 处理
答：

ANR 和 crash 遇到过么？怎样解决

答：

滑动冲突（两种，父子）
答：
不同应用的 contentprovider 的 authority 可以一样吗
答： (不可以)
给一个界面，让写 xml
答：
哪些东西可以 gc root
答：
mark word
答：
monitor 机制
答：
gc 三种算法，标记清除缺点
答：
wait notify 内部机制，notify 和临界区。。
答：
volatile 和 synchronized 区别
答：
threadlocal 内部实现，维护的 map 里面 key 是啥
答：
arraylist 内部实现 扩容 复杂度 blabla
答：
项目为什么会 OOM
答：
handler 处理机制、底层源码？message 和 handler 怎么一一对应的？在 UI 线程中，有 looper，为什么不会发生死锁？
答：
类加载器的 6 个阶段？
答：
双亲委派机制以及应用场景？
答：

Android 多线程实现方式有哪些？

答：
hashmap 原理，hashmap 是不是线程安全的（hashtable）？
答：
hashmap 和 hashtable 区别。
答：

简述 HashMap 的存储数据结构

答：
线程安全的集合？

答：

简述你在项目中进行了哪些多线程开发？

答：
进程怎么通信？
答：
Android 进程怎么通信？
答：
进程与线程的区别？
答：
数组与链表的区别？
答：
onSaveInstance ？
答：
一个线程能获取其他线程的 looper 吗？
答：
Android 怎样启动新线程？
答：
双栈实现队列
答：
GC 算法
答：
项目中遇到的内存泄漏类型
答：
解决内存泄漏问题的方法
答：
MAT
答：
LeakCanary
答：
Handler 内存泄漏的原因
答：
软引用与弱引用
答：
虚拟机栈以及栈帧
答：
Binder 原理
答：

Sync 与 Lock 的区别
答：
Sync 的原理
答：
Lock 的内部实现
答：
AQS 的内部实现
答：
CAS
答：
startActivity 框架层实现
答：
ActivityMamagerService
答：
ApplicationThread
答：

2.3 设计模式

了解设计模式吗？
答：单例模式

聊聊你常用的单例模式

答：

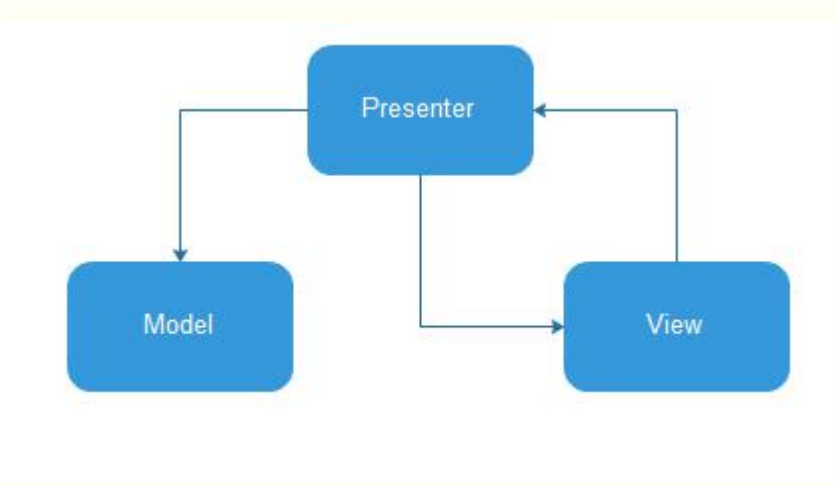
手写 DCL 单例模式
答：
synchronized 关键字以及具体实例，然后分析会不会互斥，已经同步的是什么？

答:让我讲讲设计模式，我说讲单例，他说可以。写了 dcl 之后问 dcl 失效的问题，什么情况会失效。然后说有没有更好的方式，我说静态内部类方式，但我自己没怎么写过。说到静态内部类方式的时候，发现跟第四个问题有关联。

30.MVP 设计模式？

答：按照 MVC 的分层，Activity 和 Fragment（后面只说 Activity）应该属于 View 层，用于展示 UI 界面，以及接收用户的输入，此外还要承担一些生命周期的工作。Activity

是在 Android 开发中充当非常重要的角色，特别是 TA 的生命周期的功能，所以开发的时候我们经常把一些业务逻辑直接写在 Activity 里面，这非常直观方便，代价就是 Activity 会越来越臃肿，超过 1000 行代码是常有的事，而且如果是一些可以通用的业务逻辑（比如用户登录），写在具体的 Activity 里就意味着这个逻辑不能复用了。如果有进行代码重构经验的人，看到 1000+行的类肯定会有所顾虑。因此，Activity 不仅承担了 View 的角色，还承担了一部分的 Controller 角色，这样一来 V 和 C 就耦合在一起了，虽然这样写方便，但是如果业务调整的话，要维护起来就难了，而且在一个臃肿的 Activity 类查找业务逻辑的代码也会非常蛋疼，所以看起来有必要在 Activity 中，把 View 和 Controller 抽离开来，而这就是 MVP 模式的工作了。



MVP 模式的核心思想

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

这就是 MVP 模式，现在这样的话，Activity 的工作的简单了，只用来响应生命周期，其他工作都丢到 Presenter 中去完成。从上图可以看出，Presenter 是 Model 和 View 之间的桥梁，为了让结构变得更加简单，View 并不能直接对 Model 进行操作，这也是 MVP 与 MVC 最大的不同之处。

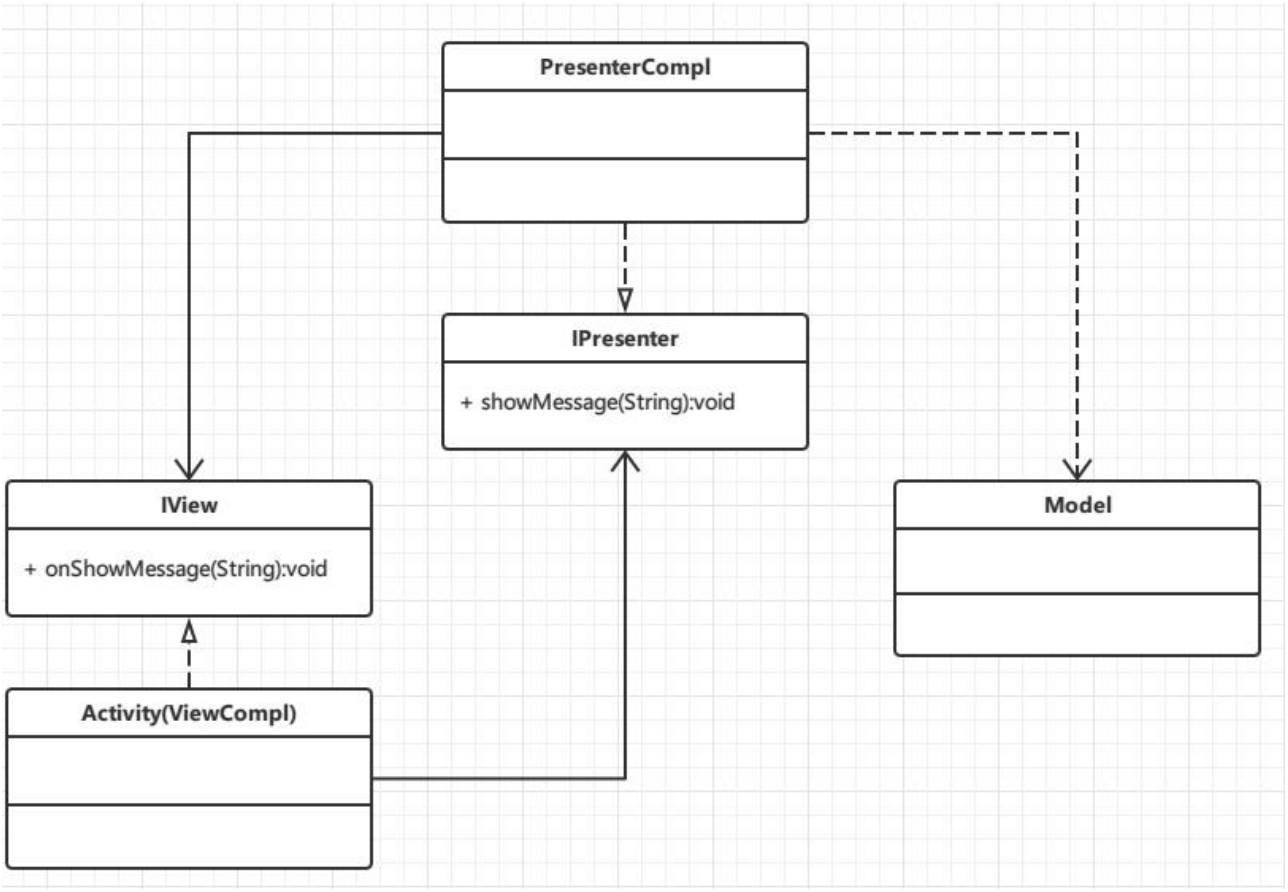
MVC 模式的作用：

- 分离了视图逻辑和业务逻辑，降低了耦合
- Activity 只处理生命周期的任务，代码变得更加简洁
- 视图逻辑和业务逻辑分别抽象到了 View 和 Presenter 的接口中去，提高代码的可阅读性

- 把业务逻辑抽到 Presenter 中去，避免后台线程引用着 Activity 导致 Activity 的资源无法被系统回收从而引起内存泄露和 OOM

- Presenter 被抽象成接口，可以有多种具体的实现，所以方便进行单元测试

MVC 模式的使用：



上面一张简单的 MVP 模式的 UML 图，从图中可以看出，使用 MVP，至少需要经历以下步骤：

1. 创建 IPresenter 接口，把所有业务逻辑的接口都放在这里，并创建它的实现 PresenterCompl（在这里可以方便地查看业务功能，由于接口可以有多种实现所以也方便写单元测试）

2. 创建 IView 接口，把所有视图逻辑的接口都放在这里，其实现类是当前的 Activity/Fragment

3. 由 UML 图可以看出，Activity 里包含了一个 IPresenter，而 PresenterCompl 里又包含了一个 IView 并且依赖了 Model。Activity 里只保留对 IPresenter 的调用，其它工作全部留到 PresenterCompl 中实现

4. Model 并不是必须有的，但是一定会有 View 和 Presenter

通过上面的介绍，MVP 的主要特点就是把 Activity 里的许多逻辑都抽离到 View 和 Presenter 接口中去，并由具体的实现类来完成。这种写法多了许多 IView 和 IPresenter 的接口，在某种程度上加大了开发的工作量，刚开始使用 MVP 的小伙伴可能会觉得这种写法比较别扭，而且难以记住。

但是在具体项目中多操作几次，就能熟悉 MVP 模式的写法，理解 TA 的意图，以及享受其带来的好处。

2.4 扩展（项目经历与算法）

2.4.1 校招：

写的项目，实习期间都做了什么，实习感受，有什么收获

需要中遇到的问题，如何解决 ？

学习路程和经历

用的框架 233

腾讯和百度实习感受，技术氛围

问我拿的奖项，参加的比赛

实习项目图片压缩

实习项目图片缓存

实习项目高斯模糊

实习项目高斯模糊动态效果

社招：

项目或者实验室怎么负责,多少人， 怎么分配职能？

项目难题中常见的难题如 crossword 算法，ocr 调优，自定义相机页面,项目最终有没有上线？

具体项目具体分析例如步态分析器这个项目的话，你感觉核心实现是什么？
glide 缓存了解吗(说了磁盘-内存缓存)
glide 缓存怎么实现？(不太清楚,只使用过)
拍照的图片反转问题
jetpack 了解吗，
flutter 了解吗(做过 demo)
hashmap
synchonized,wait ,notify 使用
volatile
本地服务

算法

手写快排？

答：快速排序是一种很不错的排序算法，算法复杂度为 n*logn。快排使用了分而治之的思想,每次排序是都找到一个基准(我们学习时经常使用第一个作为基准),然后把小于基准的元素放到基准元素的左边,大于基准的元素放到基准元素的右边，这样一次排序下来，基准元素左边都是小于（等于）基准的数，基准右边的元素都是大于（等于）基准的元素了。快速排序关键点就是找到这样一个基准并将其放到恰到的位置。

算法思路

定义一个快速排序函数，arr 是要排序的数组，l 指向要排序的数组最左边的元素，r 指向要排序的数组最右边的元素。

```
public static void quick_sort(int arr[],int l,int r){

        if(l >= r) return;

        //p 为快速排序返回的基准的位置

        int p = partition2(arr,l,r);

        //对基准左边的数进行快排

        quick_sort(arr,l,p-1);
```

```
        //对基准右边的数进行快排
        quick_sort(arr,p+1,r);
    }
```

那么现在关键就是这么实现这个 partition2 函数

假设现在有一个数组 arr[]：

1. 判断交叉链表并找出相交节点

题目：一个字符串，开始可能有 0-N 个空格，然后在这个串中每个单词之间有 1-N 个空格，最后的结束也有 N 个空格，现在要求最后的输出结果是开始不能有空格，每个单词之间只能有一个空格，最后全部是空格的格式

一、先实现，无要求

二、空间复杂度要求为 O(1)，再优化

两个链表寻找交叉

探讨 Json 解析器的实现(词法分析 语法分析 采用何种数据结构并且为什么等问题)

图片的三级缓存

MVC,MVP,MVVC 区别

Android 中使用 MVP 而不用 MVC 的原因

讲解 MVP(手写代码)

MVVM 与 DataBinding

实现监听手机拍照和截屏,悄悄地将一些数据隐藏进入图片并且可以通过图片恢复数据 **(写 Demo 发到指定 Leader 邮箱)**

算法：链表倒数第 k 个元素

（了解）说一下二叉树左右子树的交换（递归交换）

6. 写一个算法吧！二叉树-每个节点都有一个 int，求从根节点出发，求无回退路径最大值。发了一个网址过来，很快写好，打电话过去说写好了。讲一下思路（贪心-递归）

小米每次面试基本上必备的就是手撕算法，so，一定要做好点准备，难度都一般，不是难的那种。另外从 android 的技术上来说，从我的面试经历来说都是一点即过，面试官没有多少深入的挖掘知识点。面试官都很 nice，有些算法上卡着了，会一步一步的引导你。赞一个

2.5、综合类问题

为什么选择简历投递的工作地？

1. 有没有收到其他的 offer？

1. 都投了小米公司哪些岗位？ （安卓，移动端）

对公司有什么建议、对研发有什么建议、对研发方式有什么建议

说说测试给你提的 bug，以及解决方法

看过哪些安卓系统源码，讲下活动启动的过程，安卓系统启动的过程

聊人生，为什么学 android

android7.0 新特性，权限

除了 java 还用过哪些语言

1.个人有什么缺点和优点？

2.生活中遇到与你无法相处的人时，你如何处理？

答：短期相处的话适当谦让，长期相处的话通过团建、沟通等方式缓解矛盾。

3.有什么兴趣爱好？

4.大学生活中遇到了哪些挫折？

5.爱看什么书？

6.是否有个人博客？

7.今后有什么职业目标？

三、面试者提问

让我问了下他的几个问题？
答：我面试的部门技术选型、他说大数据那些，基础应用都有,我说有具体的语言限制么？他说 java、pyhton、GO 这些语言都有，根据具体的业务

致谢：

CSDN 博主「OneDeveloper」的原创文章，原文链接：https://blog.csdn.net/OneDeveloper/article/details/84571564

菜鸟教程 https://www.runoob.com/java/java-collections.html

本文为 CSDN 博主「木兰的胡萝北」原文链接：
https://blog.csdn.net/qq_44865616/article/details/90601115

本文为 CSDN 博主「lx_Frolf」原文链接：
https://blog.csdn.net/lx_Frolf/article/details/82560911

博客园 smallJunJun

https://www.cnblogs.com/smallJunJun/p/10652068.html

CSDN 博主「goto_Mazinger」原文链接：
https://blog.csdn.net/zy1471162851/article/details/91440119

简书原文链接：https://www.jianshu.com/p/3ca05281380f

博客园 steven520213 https://www.cnblogs.com/steven520213/p/8005258.html

博客园 TreeSirhttps://www.cnblogs.com/bigtreei/p/8109303.html

RUNOOB.COMhttps://www.runoob.com/linux/linux-comm-dump.html
博客园 jianhuazhao
https://www.cnblogs.com/zhaojianhua/p/9239656.html
CSDN 博主「刷了牙就睡 fdd」
原文链接：https://blog.csdn.net/fdd11119/article/details/80682390

CSDN 博主「pxM_Wxd」原文链接：https://blog.csdn.net/d_dmelon/article/details/54295764

简书 MagicDong https://www.jianshu.com/p/206a95ed784f