1、HashMap、HashTable、ConcurrentHashMap 的区别

HashMap 和 HashTable 都实现了 Map 接口，里面存放的元素不保证有序，并且不存在相同元素；

区别（线程安全和保存值是否为 null 方面）：

（1） HashMap 和 HashTable 在功能上基本相同，但 HashMap 是线程不安全的，HashTable 是线程安全的；

HashMap 的 put 源码如下：

```
public V put(K key, V value) {

    if (table == EMPTY_TABLE) {

        inflateTable(threshold);

    }

    if (key == null)

                return putForNullKey(value);

    //说明key 和value 值都是可以为null

    int hash = hash(key);

    int i = indexFor(hash, table.length);

    for (Entry<K,V> e = table[i]; e != null; e = e.next) {

        Object k;

        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

            V oldValue = e.value;

            e.value = value;

            e.recordAccess(this);
```

```
            return oldValue;

        }

    }

    modCount++;

    addEntry(hash, key, value, i);

    return null;

}
```

（2）可以看到，HashMap 的 key 和 value 都是可以为 null 的，当 get()方法返回 null 值时，HashMap 中可能存在某个 key，只不过该 key 值对应的 value 为 null，也有可能是 HashM 中不存在该 key，所以不能使用 get()==null 来判断是否存在某个 key 值，对于 HashMap 和 HashTable，提供了 containsKey()方法来判断是否存在某个 key。

HashTable 的 put 源码如下：

```
public synchronized V put(K key, V value) {

    // Make sure the value is not null

    if (value == null) {

        //当value==null 的时候，会抛出异常

        throw new NullPointerException();

    }

    // Makes sure the key is not already in the hashtable.

    Entry tab[] = table;

    int hash = hash(key);

    int index = (hash & 0x7FFFFFFF) % tab.length;

    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
```

```
        if ((e.hash == hash) && e.key.equals(key)) {
            V old = e.value;
            e.value = value;
            return old;
        }
    }
    modCount++;
    if (count >= threshold) {
        // Rehash the table if the threshold is exceeded
        rehash();
        tab = table;
        hash = hash(key);
        index = (hash & 0x7FFFFFFF) % tab.length;
    }
    // Creates the new entry.
    Entry<K,V> e = tab[index];
    tab[index] = new Entry<>(hash, key, value, e);
    count++;
    return null;
}
```

（3）HashTable 是不允许 key 和 value 为 null 的。HashTable 中的方法大部分是同步的，因此 HashTable 是线程安全的。

拓展：
（1） 影响 HashMap（或 HashTable）性能的两个因素：初始容量和 load factor；

HashMap 中有如下描述： When the number of entries in the hash table exceeds the product of the load factor and the current capacity,

the hash table is rehashed (that is, internal data structures are rebuilt)

当我们 Hash 表中数据记录的大小超过当前容量，Hash 表会进行 rehash 操作，其实就是自动扩容，这种操作一般会比较耗时。所以当我们能够预估 Hash 表大小时，在初始化的时候就尽量指定初始容量，避免中途 Hash 表重新扩容操作，如：

```
HashMap<String, Integer> map = new HashMap<String, Integer>(20);
```

（类似可以指定容量的还有 ArrayList、Vector）

（2）使用选择上，当我们需要保证线程安全，HashTable 优先选择。当我们程序本身就是线程安全的，HashMap 是优先选择。

其实 HashTable 也只是保证在数据结构层面上的同步，对于整个程序还是需要进行多线程并发控制；在 JDK 后期版本中，对于 HashMap，可以通过 Collections 获得同步的 HashMap；如下：

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

这种方式获得了具有同步能力的 HashMap。

（3）在 JDK1.5 以后，出现了 ConcurrentHashMap，它可以很好地解决在并发程序中使用 HashMap 的问题，ConcurrentHashMap 和 HashTable 功能很像，不允许为 null 的 key 或 value，但它不是通过给方法加 synchronized 方法进行并发控制的。

在 ConcurrentHashMap 中使用分段锁技术 Segment，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，能够实现真正的并发访问。效率也比 HashTable 好的多。

2、TreeMap、HashMap、LinkedHashMap 的区别

关于 Map 集合，前面几篇都有讲过，可以去回顾一下。而 TreeMap、HashMap、LinkedHashMap 都是 Map 的一些具体实现类，其关系图如下：

其中，HashMap 和 HashTable 主要区别在线程安全方面和存储 null 值方面。HashMap 前面讨论的已经比较多了，下面说说 LinkedHashMap 和 TreeMap。

（1）LinkedHashMap 保存了数据的插入顺序，底层是通过一个双链表的数据结构来维持这个插入顺序的。key 和 value 都可以为 null；
（2）TreeMap 实现了 SortMap 接口，它保存的记录是根据键值 key 排序，默认是按 key 升序排列。也可以指定排序的 Comparator。

HashMap、LinkedHashMap 和 TreeMap 都是线程不安全的，HashTable 是线程安全的。提供两种遍历 Map 的方法如下：

（1）推荐方式：

```
Map<String, Integer> map = new HashMap<String, Integer>(20);

for (Map.Entry<String, Integer> entry : map.entrySet()){

    //直接遍历出 Entry

    System.out.println("key-->"+entry.getKey()+",value-->"+m.get(entry.getValue()))
;

}
```

这种方式相当于首先通过 Set<Map.Entry<String,Integer>> set = map.entrySet();方式拿到 Set 集合，而 Set 集合是可以通过 foreach 的方式遍历的。

（2） 普通方式：

```
Map<String, Integer> map = new HashMap<String, Integer>(20);

Iterator<String> keySet = map.keySet().iterator();
```

```
//遍历Hash 表中的key 值集合，通过key 获取value

while(keySet .hasNext()){

    Object key = keySet .next();

    System.out.println("key-->"+key+",value-->"+m.get(key));

}
```

3、Collection 包结构，与 Collections 的区别。

Collection 的包结构如下：

Statck 类为 Vector 的子类。由于 Collection 类继承 Iterable 类，所以，所有 Collection 的实现类都可以通过 foreach 的方式进行遍历。

Collections 是针对集合类的一个帮助类。提供了一系列静态方法实现对各种集合的搜索、排序、线程完全化等操作。

当于对 Array 进行类似操作的类——Arrays。

如，Collections.max(Collection coll); 取 coll 中最大的元素。

Collections.sort(List list); 对 list 中元素排序

4、OOM 你遇到过哪些情况，SOF 你遇到过哪些情况

OOM：OutOfMemoryError 异常，

即内存溢出，是指程序在申请内存时，没有足够的空间供其使用，出现了 Out Of Memory，也就是要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。

内存溢出分为上溢和下溢，比方说栈，栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。

有时候内存泄露会导致内存溢出，所谓内存泄露（memory leak），是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但

内存泄露堆积后果很严重，无论多少内存,迟早会被占光，举个例子，就是说系统的篮子（内存）是有限的，而你申请了一个篮子，拿到之后没有归还（忘记还了或是丢了），于是造成一次内存泄漏。在你需要用篮子的时候，又去申请，如此反复，最终系统的篮子无法满足你的需求，最终会由内存泄漏造成内存溢出。

遇到的 OOM：

（1）Java Heap 溢出

Java 堆用于存储对象实例，我们只要不断的创建对象，而又没有及时回收这些对象（即内存泄漏），就会在对象数量达到最大堆容量限制后产生内存溢出异常。

（2）方法区溢出

方法区用于存放 Class 的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。

异常信息：java.lang.OutOfMemoryError:PermGen space

方法区溢出也是一种常见的内存溢出异常，一个类如果要被垃圾收集器回收，判定条件是很苛刻的。在经常动态生成大量 Class 的应用中，要特别注意这点。

SOF：StackOverflow（堆栈溢出）

当应用程序递归太深而发生堆栈溢出时，抛出该错误。因为栈一般默认为 1-2m，一旦出现死循环或者是大量的递归调用，在不断的压栈过程中，造成栈容量超过 1m 而导致溢出。

栈溢出的原因：

- 递归调用
- 大量循环或死循环
- 全局变量是否过多
- 数组、List、Map 数据过大

OOM 在 Android 开发中出现比较多：

场景有： 加载的图片太多或图片过大时、分配特大的数组、内存相应资源过多没有来不及释放等。

解决方法：

（1）在内存引用上做处理，软引用是主要用于内存敏感的高速缓存。在 jvm 报

告内存不足之前会清除所有的软引用，这样以来 gc 就有可能收集软可及的对象，可能解决内存吃紧问题，避免内存溢出。什么时候会被收集取决于 gc 的算法和 gc 运行时可用内存的大小。
（2）对图片做边界压缩，配合软引用使用
（3）显示的调用 GC 来回收内存，如：

```
if(bitmapObject.isRecycled()==false) //如果没有回收

bitmapObject.recycle();
```

（4）优化 Dalvik 虚拟机的堆内存分配

增强程序堆内存的处理效率

```
//在程序 onCreate 时就可以调用 即可

privatefinalstaticfloat TARGET_HEAP_UTILIZATION = 0.75f;

VMRuntime.getRuntime().setTargetHeapUtilization(TARGET_HEAP_UTILIZATION);
```

设置堆内存的大小

```
privatefinalstaticintCWJ_HEAP_SIZE = 6* 1024* 1024;

//设置最小 heap 内存为 6MB 大小

VMRuntime.getRuntime().setMinimumHeapSize(CWJ_HEAP_SIZE);
```

（5）用 LruCache 和 AsyncTask<>解决

从 cache 中去取 Bitmap，如果取到 Bitmap，就直接把这个 Bitmap 设置到 ImageView 上面。

如果缓存中不存在，那么启动一个 task 去加载（可能从文件来，也可能从网络）。

5、Java 面向对象的三个特征与含义，多态的实现方式

Java 中两个非常重要的概念：类和对象。类可以看做是一个模板，描述了一类对象的属性和行为；而对象是类的一个具体实现。Java 面向对象的三大基本特征：

（1）封装

属性用来描述同一类事物的特征，行为用来描述同一类事物可做的一些操作。封装就是把属于同一类事物的共性（属性和行为）归到一个类中，只保留有限的接口和方法与外部进行交互，避免了外界对对象内部属性的破坏。Java 中使用访问控制符来保护对类、属性、方法的访问。

（2）继承

子类通过这种方式来接收父类所有的非 private 的属性和方法（构造方法除外）。这里的接收是直接拥有的意思，即可以直接使用父类字段和方法，因此，继承相当于“扩展”，子类在拥有了父类的属性和特征后，可以专心实现自己特有的功能。

（构造方法不能被继承，因为在创建子类时，会先去自动“调用”父类的构造方法，如果真的需要子类构造函数特殊的形式，子类直接修改或重载自己的构造函数就好了。）

（3）多态

多态是程序在运行的过程中，同一种类型在不同的条件下表现不同的结果。比如：

```
Animal a = new Dog();

// 子类对象当做父类对象来使用，运行时，根据对象的实际类型去找子类覆盖之后的方法
```

多态实现方式：

- 设计时多态，通过方法的重载实现多态；
- 运行时多态，通过重写父类或接口的方法实现运行时多态；

6、interface 与 abstract 类的区别

abstract class 只能被继承 extends，体现的是一种继承关系，而根据继承的特征，有继承关系的子类和父类应该是一种“is-a”的关系，也即两者在本质上应该是相同的（有共同的属性特征）。

interface 是用来实现的 implements，它并不要求实现者和 interface 之间在本质上相同，是一种“like-a”的关系，interface 只是定义了一系列的约定而已（实现者表示愿意遵守这些约定）。所以一个类可以去实现多个 interface（即该类遵守了多种约定）。

很多情况下 interface 和 abstract 都能满足我们要求，在我们选择用 abstract 火 interface 的时候，尽量符合上面的要求，即如果两者间本质是一样的，是一种“is-a”的关系，尽量用 abstract，当两者之间本质不同只是简单的约定行为的话，可以选择 interface。

特点：

- abstract 类其实和普通类一样，拥有有自己的数据成员和方法，只不过 abstract 类里面可以定义抽象 abstract 的方法（声明为 abstract 的类也可以不定义 abstract 的方法，直接当做普通类使用，但是这样就失去了抽象类的意义）。
- 一个类中声明了 abstract 的方法，该类必须声明为 abstract 类。
- interface 中只能定义常量和抽象方法。在接口中，我们定义的变量默认为 public static final 类型，所以不可以在显示类中修改 interface 中的变量；定义的方法默认为 public abstract，其中 abstract 可以不明确写出。

7、static class 与 non static class 的区别

static class--静态内部类，non static class--非静态内部类，即普通内部类

普通内部类：

内部类可以直接使用外部类的所有变量（包括 private、静态变量、普通变量），这也是内部类的主要优点（不用生成外部类对象而直接使用外部类变量）。如下例子：

```
public class OutClass {

    private String mName = "lly";

    static int mAge = 12;

    class InnerClass{

        String name;

        int age;

        private void getName(){
```

```
            name = mName;

            age = mAge;

            System.out.println("name="+name+",age="+age);

        }

    }

    public static void main(String[] args) {

        //第一种初始化内部类方法

        OutClass.InnerClass innerClass = new OutClass().new InnerClass();

        innerClass.getName();

        //第二种初始化内部类方法

        OutClass out = new OutClass();

        InnerClass in = out.new InnerClass();

        in.getName();

    }

}
```

输出：

```
name=lly,age=12
```

可以看到，内部类里面可以直接访问外部类的静态和非静态变量，包括 private 变量。在内部类中，我们也可以通过外部类.this.变量名的方式访问外部类变量，如：name = OutClass.this.mName;

内部类的初始化依赖于外部类，只有外部类初始化出来了，内部类才能够初始化。

私有内部类（包括私有静态内部类和私有非静态内部类）：

如果一个内部类只希望被外部类中的方法操作，那只要给该内部类加上 private 修饰，声明为 private 的内部类只能在外部类中使用，不能在别的类中 new 出来。如上 private class InnerClass{...}，此时只能在 OutClass 类中使用。

静态内部类：
静态内部类只能访问外部类中的静态变量。如下：

```
public class OutClass {

    private String mName = "lly";

    static int mAge = 12;

    static class StaticClass{

        String name = "lly2";

        int age;

        private void getName(){

            //            name = mName;    //不能引用外部类的非静态成员变量

            age = mAge;

            System.out.println("name="+name+",age="+age);

        }

    }

    public static void main(String[] args) {

        //第一种初始化静态内部类方法

        OutClass.StaticClass staticClass = new OutClass.StaticClass();

        staticClass.getName();

        //或者直接使用静态内部类初始化

        StaticClass staticClass2 = new StaticClass();

        staticClass2.getName();
```

```
    }

}
```

输出：

```
name=lly2,age=12
```

可以看到，静态内部类只能访问外部类中的静态变量，静态内部类的初始化不依赖于外部类，由于是 static，类似于方法使用，OutClass.StaticClass 是一个整体。

匿名内部类：

匿名内部类主要是针对抽象类和接口的具体实现。在 Android 的监听事件中用的很多。如：

```
textView.setOnClickListener(new View.OnClickListener(){

    //OnClickListener 为一个接口 interface

    public void onClick(View v){

        ...

    }

}

);
```

对于抽象类：

```
public abstract class Animal {

    public abstract void getColor();

}

public class Dog{
```

```
    public static void main(String[] args) {

        Animal dog = new Animal() {

            @Override

                        public void getColor() {

                System.out.println("黑色");

            }

        }

        ;

        dog.getColor();

    }

}
```

输出：黑色

对于接口类似，只需要把 abstract class 改为 interface 即可。