阿里云Java实习岗面试记录整理

List 和 Set 的区别

List，Set 都是继承自 Collection 接口 List 特点：元素有放入顺序，元素可重复 ，

Set 特点：元素无放入顺序，元素不可重复，重复元素会覆盖掉，（元素虽然无放入顺序，但是元素在set中的位置是有该元素的 HashCode 决定的，其位置其实是固定的，加入Set 的 Object 必须定义 equals ()方法 ，另外list支持for循环，也就是通过下标来遍历，也可以用迭代器，但是set只能用迭代，因为他无序，无法用下标来取得想要的值。）Set和List对比 Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。
List：和数组类似，List可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变

HashSet 是如何保证不重复的

向 HashSet 中 add ()元素时，判断元素是否存在的依据，不仅要比较hash值，同时还要结合 equles 方法比较。HashSet 中的 add ()方法会使用 HashMap 的 add ()方法。以下是 HashSet 部分源码：

```
private static final Object PRESENT = new Object();
private transient HashMap<E,Object> map;
public HashSet() {
        map = new HashMap<>();
}
public boolean add(E e) {
        return map.put(e, PRESENT)==null;
}
```

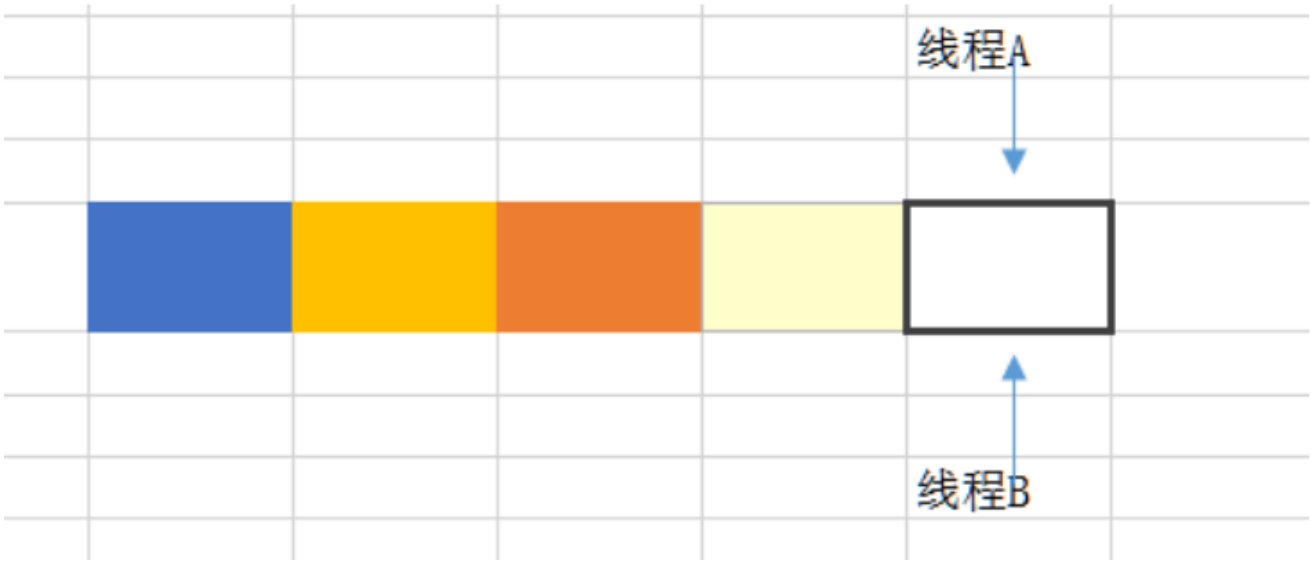
HashMap 的 key 是唯一的，由上面的代码可以看出 HashSet 添加进去的值就是作为 HashMap 的key。所以不会重复（ HashMap 比较key是否相等是先比较 hashcode 在比较 equals ）。

HashMap 是线程安全的吗，为什么不是线程安全的（最好画图说明多线程环境下不安全）？

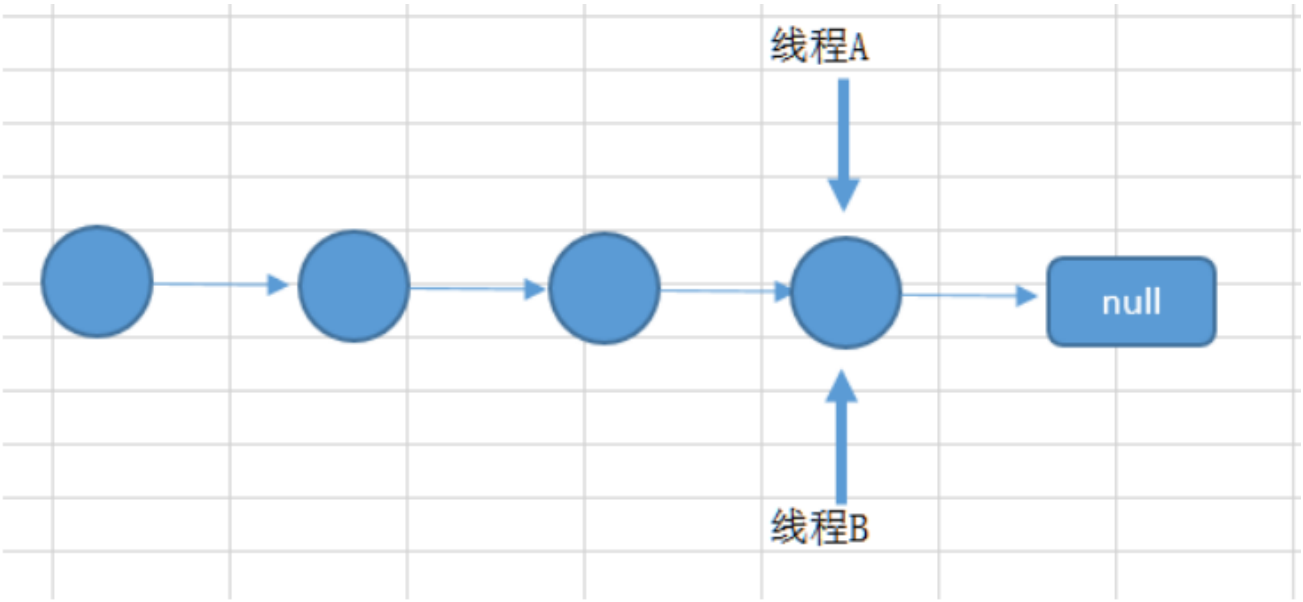
不是线程安全的；

如果有两个线程A和B，都进行插入数据，刚好这两条不同的数据经过哈希计算后得到的哈希码是一样的，且该位置还没有其他的数据。所以这两个线程都会进入我在上面标记为1的代码中。假设一种情况，线程A通过if判断，该位置没有哈希冲突，进入了if语句，还没有进行数据插入，这时候 CPU 就把资源让给了线程B，线程A停在了if语句里面，线程B判断该位置没有哈希冲突（线程A的数据还没插入），也进入了if语句，线程B执行完后，轮到线程A执行，现在线程A直接在该位置插入而不用再判断。这时候，你会发现线程A把线程B插入的数据给覆盖了。发生了线程不安全情况。本来在 HashMap 中，发生哈希冲突是可以用链表法或者红黑树来解决的，但是在多线程中，可能就直接给覆盖了。

上面所说的是一个图来解释可能更加直观。如下面所示，两个线程在同一个位置添加数据，后面添加的数据就覆盖住了前面添加的。



如果上述插入是插入到链表上，如两个线程都在遍历到最后一个节点，都要在最后添加一个数据，那么后面添加数据的线程就会把前面添加的数据给覆盖住。则



在扩容的时候也可能会导致数据不一致，因为扩容是从一个数组拷贝到另外一个数组。

`HashMap` 的扩容过程

当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值(知道这个阈字怎么念吗？不念`fa`值，念`yu`值四声)---即当前数组的长度乘以加载因子的值的时候，就要自动扩容啦。

扩容(`resize`)就是重新计算容量，向`HashMap`对象里不停的添加元素，而`HashMap`对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然`Java`里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

```
HashMap hashMap=new HashMap(cap);
```

`cap`=3，`hashMap`的容量为4；

`cap`=4，`hashMap`的容量为4；

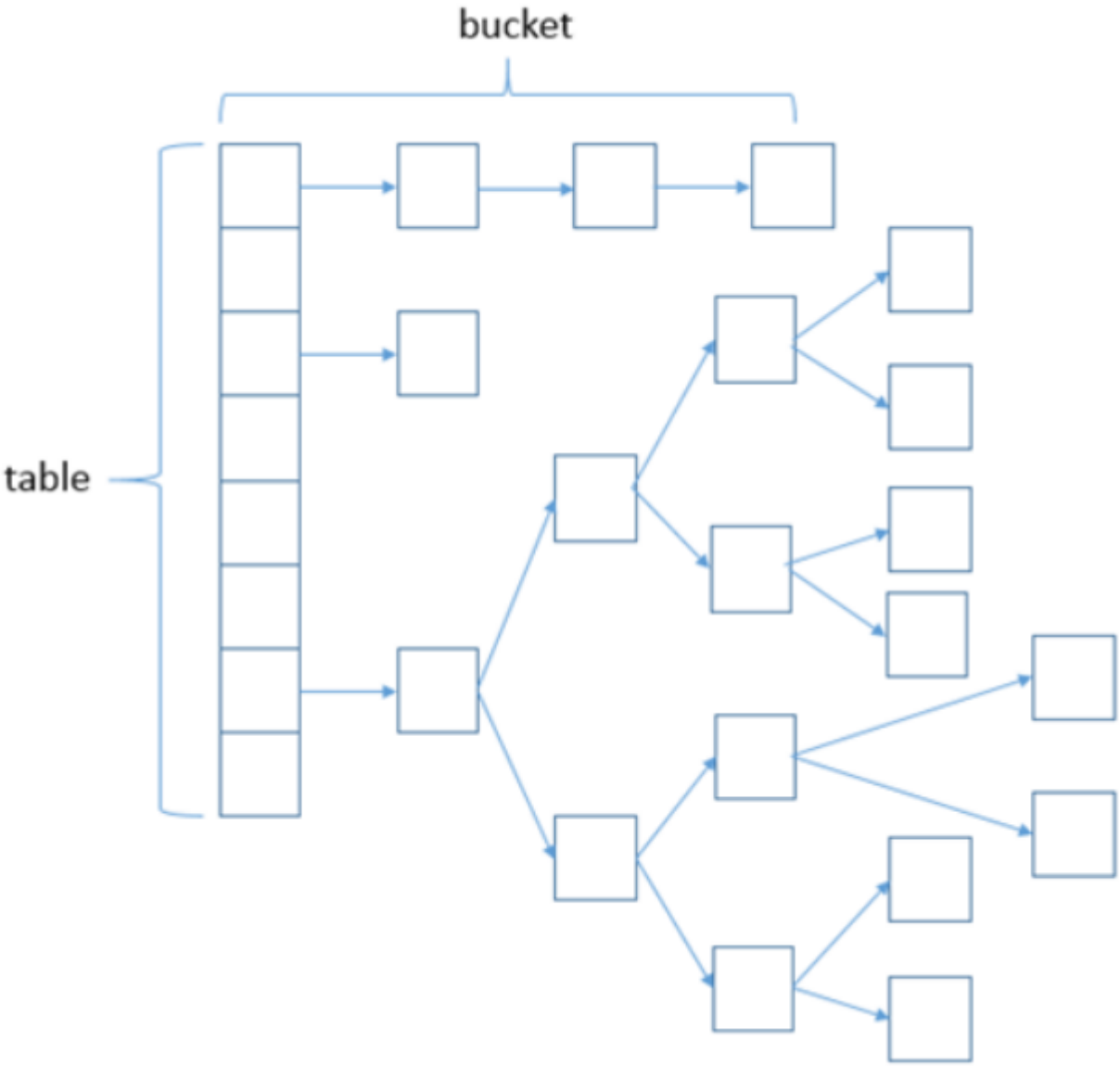
`cap`=5，`hashMap`的容量为8；

`cap`=9，`hashMap`的容量为16；

如果`cap`是2的n次方，则容量为`cap`，否则为大于`cap`的第一个2的n次方的数。

`HashMap` 1.7 与 1.8 的 区别，说明 1.8 做了哪些优化，如何优化的？

HashMap结构图



在`JDK1.7`及之前的版本中，`HashMap`又叫散列链表：基于一个数组以及多个链表的实现，hash值冲突的时候，就将对应节点以链表的形式存储。

`JDK1.8`中，当同一个hash值（`Table`上元素）的链表节点数不小于8时，将不再以单链表的形式存储了，会被调整成一颗红黑树。这就是`JDK7`与`JDK8`中`HashMap`实现的最大区别。

其下基于 JDK1.7.0_80 与 JDK1.8.0_66 做的分析

JDK1.7中

使用一个 `Entry` 数组来存储数据，用key的 `hashcode` 取模来决定key会被放到数组里的位置，如果 `hashcode` 相同，或者 `hashcode` 取模后的结果相同（ `hash collision` ），那么这些 `key` 会被定位到 `Entry` 数组的同一个格子里，这些 `key` 会形成一个链表。

在 `hashcode` 特别差的情况下，比方说所有key的 `hashcode` 都相同，这个链表可能会很长，那么 `put/get` 操作都可能需要遍历这个链表，也就是说时间复杂度在最差情况下会退化到 `O(n)`

JDK1.8中

使用一个 `Node` 数组来存储数据，但这个 `Node` 可能是链表结构，也可能是红黑树结构

- 如果插入的 `key` 的 `hashcode` 相同，那么这些key也会被定位到 `Node` 数组的同一个格子里。
- 如果同一个格子里的key不超过8个，使用链表结构存储。
- 如果超过了8个，那么会调用 `treeifyBin` 函数，将链表转换为红黑树。

那么即使 `hashcode` 完全相同，由于红黑树的特点，查找某个特定元素，也只需要O(log n)的开销

也就是说put/get的操作的时间复杂度最差只有 `O(log n)`

听起来挺不错，但是真正想要利用 `JDK1.8` 的好处，有一个限制：

key的对象，必须正确的实现了 `Compare` 接口

如果没有实现 `Compare` 接口，或者实现得不正确（比方说所有 `Compare` 方法都返回0）

那 `JDK1.8` 的 `HashMap` 其实还是慢于 `JDK1.7` 的

简单的测试数据如下：

向 `HashMap` 中 `put/get 1w` 条 `hashcode` 相同的对象

`JDK1.7:` `put 0.26s` ， `get 0.55s`

`JDK1.8`（未实现 `Compare` 接口）：`put 0.92s` ， `get 2.1s`

但是如果正确的实现了 `Compare` 接口，那么 `JDK1.8` 中的 `HashMap` 的性能有巨大提升，这次 `put/get` 100W条 `hashcode` 相同的对象

`JDK1.8`（正确实现 `Compare` 接口，）：`put/get` 大概开销都在320 `ms` 左右

`final finally finalize`

- final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。
- finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。
- finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用 `System.gc()` 方法的时候，由垃圾回收器调用finalize()，回收垃圾，一个对象是否可回收的最后判断。

对象的四种引用

强引用 只要引用存在，垃圾回收器永远不会回收

```
Object obj = new Object();
User user=new User();
```

可直接通过obj取得对应的对象 如`obj.equels(new Object());`而这样`obj`对象对后面`new Object`的一个强引用，只有当`obj`这个引用被释放之后，对象才会被释放掉，这也是我们经常所用到的编码形式。

软引用 非必须引用，内存溢出之前进行回收，可以通过以下代码实现

```
Object obj = new Object();
SoftReference<Object> sf = new SoftReference<Object>(obj);
obj = null;
sf.get();//有时候会返回null
```

这时候sf是对obj的一个软引用，通过sf.get()方法可以取到这个对象，当然，当这个对象被标记为需要回收的对象时，则返回null； 软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

弱引用 第二次垃圾回收时回收，可以通过如下代码实现

```
Object obj = new Object();
WeakReference<Object> wf = new WeakReference<Object>(obj);
obj = null;
wf.get();//有时候会返回null
wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾
```

弱引用是在第二次垃圾回收时回收，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回null。弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的`isEnQueued`方法返回对象是否被垃圾回收器标记。

`ThreadLocal`中有使用到弱引用，

```
public class ThreadLocal<T> {
 static class ThreadLocalMap {
        static class Entry extends WeakReference<ThreadLocal<?>> {
            /** The value associated with this ThreadLocal. */
            Object value;

            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }
     //....
   }
    //.....
 }
```

虚引用 垃圾回收时回收，无法通过引用取到对象值，可以通过如下代码实现

```
Object obj = new Object();
PhantomReference<Object> pf = new PhantomReference<Object>(obj);
obj=null;
pf.get();//永远返回null
pf.isEnQueued();//返回是否从内存中已经删除
```

虚引用是每次垃圾回收的时候都会被回收，通过虚引用的get方法永远获取到的数据为null，因此也被成为幽灵引用。虚引用主要用于检测对象是否已经从内存中删除。

Java获取反射的三种方法

1.通过new对象实现反射机制 2.通过路径实现反射机制 3.通过类名实现反射机制

```
public class Student {
    private int id;
    String name;
    protected boolean sex;
    public float score;
}
```

```
public class Get {
    //获取反射机制三种方式
    public static void main(String[] args) throws ClassNotFoundException {
        //方式一(通过建立对象)
        Student stu = new Student();
        Class classobj1 = stu.getClass();
        System.out.println(classobj1.getName());
        //方式二（所在通过路径-相对路径）
        Class classobj2 = Class.forName("fanshe.Student");
        System.out.println(classobj2.getName());
        //方式三（通过类名）
        Class classobj3 = Student.class;
        System.out.println(classobj3.getName());
    }
}
```

Java反射机制

Java 反射机制是在运行状态中，对于任意一个类，都能够获得这个类的所有属性和方法，对于任意一个对象都能够调用它的任意一个属性和方法。这种在运行时动态的获取信息以及动态调用对象的方法的功能称为 Java 的反射机制。

Class 类与 java.lang.reflect 类库一起对反射的概念进行了支持，该类库包含了 Field,Method,Constructor 类 (每个类都实现了 Member 接口)。这些类型的对象时由 JVM 在运行时创建的，用以表示未知类里对应的成员。

这样你就可以使用 Constructor 创建新的对象，用 get() 和 set() 方法读取和修改与 Field 对象关联的字段，用 invoke() 方法调用与 Method 对象关联的方法。另外，还可以调用 `getFields()` `getMethods()` 和 `getConstructors()` 等很便利的方法，以返回表示字段，方法，以及构造器的对象的数组。这样匿名对象的信息就能在运行时被完全确定下来，而在编译时不需要知道任何事情。

```
import java.lang.reflect.Constructor;
public class ReflectTest {
    public static void main(String[] args) throws Exception {

        Class clazz = null;
        clazz = Class.forName("com.jas.reflect.Fruit");
        Constructor<Fruit> constructor1 = clazz.getConstructor();
        Constructor<Fruit> constructor2 = clazz.getConstructor(String.class);

        Fruit fruit1 = constructor1.newInstance();
        Fruit fruit2 = constructor2.newInstance("Apple");

    }
}
class Fruit{
    public Fruit(){
        System.out.println("无参构造器 Run..........");
    }
    public Fruit(String type){
        System.out.println("有参构造器 Run.........." + type);
    }

}
```

> 运行结果： 无参构造器 Run.......... 有参构造器 Run..........Apple

Arrays.sort 和 Collections.sort 实现原理 和区别

Collection和Collections区别

`java.util.Collection` 是一个集合接口。它提供了对集合对象进行基本操作的通用接口方法。
`java.util.Collections` 是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全等操作。 然后还有混排（Shuffling）、反转（Reverse）、替换所有的元素（fill）、拷贝（copy）、返回Collections中最小元素（min）、返回Collections中最大元素（max）、返回指定源列表中最后一次出现指定目标列表的起始位置（`lastIndexOfSubList`）、返回指定源列表中第一次出现指定目标列表的起始位置（`IndexOfSubList`）、根据指定的距离循环移动指定列表中的元素（Rotate）；

事实上Collections.sort方法底层就是调用的array.sort方法，

```
public static void sort(Object[] a) {
        if (LegacyMergeSort.userRequested)
            legacyMergeSort(a);
        else
            ComparableTimSort.sort(a, 0, a.length, null, 0, 0);
    }
//void java.util.ComparableTimSort.sort()
static void sort(Object[] a, int lo, int hi, Object[] work, int workBase, int workLen)
{
        assert a != null && lo >= 0 && lo <= hi && hi <= a.length;
        int nRemaining  = hi - lo;
        if (nRemaining < 2)
            return;  // Arrays of size 0 and 1 are always sorted
```

```
        // If array is small, do a "mini-TimSort" with no merges
        if (nRemaining < MIN_MERGE) {
            int initRunLen = countRunAndMakeAscending(a, lo, hi);
            binarySort(a, lo, hi, lo + initRunLen);
            return;
        }
    }
```

`legacyMergeSort`(a)：归并排序 `ComparableTimSort.sort()` ： `Timsort` 排序

`Timsort` 排序是结合了合并排序（merge sort）和插入排序（insertion sort）而得出的排序算法

Timsort的核心过程

> TimSort 算法为了减少对升序部分的回溯和对降序部分的性能倒退，将输入按其升序和降序特点进行了分区。排序的输入的单位不是一个个单独的数字，而是一个个的块-分区。其中每一个分区叫一个run。针对这些 run 序列，每次拿一个 run 出来按规则进行合并。每次合并会将两个 run合并成一个 run。合并的结果保存到栈中。合并直到消耗掉所有的 run，这时将栈上剩余的 run合并到只剩一个 run 为止。这时这个仅剩的 run 便是排好序的结果。

综上述过程，Timsort算法的过程包括

（0）如何数组长度小于某个值，直接用二分插入排序算法

（1）找到各个run，并入栈

（2）按规则合并run

`LinkedHashMap`的应用

基于 `LinkedHashMap` 的访问顺序的特点，可构造一个 `LRU（Least Recently Used）` 最近最少使用简单缓存。也有一些开源的缓存产品如 `ehcache` 的淘汰策略（ `LRU` ）就是在 `LinkedHashMap` 上扩展的。

`Cloneable`接口实现原理

Cloneable接口是Java开发中常用的一个接口， 它的作用是使一个类的实例能够将自身拷贝到另一个新的实例中，注意，这里所说的"拷贝"拷的是对象实例，而不是类的定义，进一步说，拷贝的是一个类的实例中各字段的值。

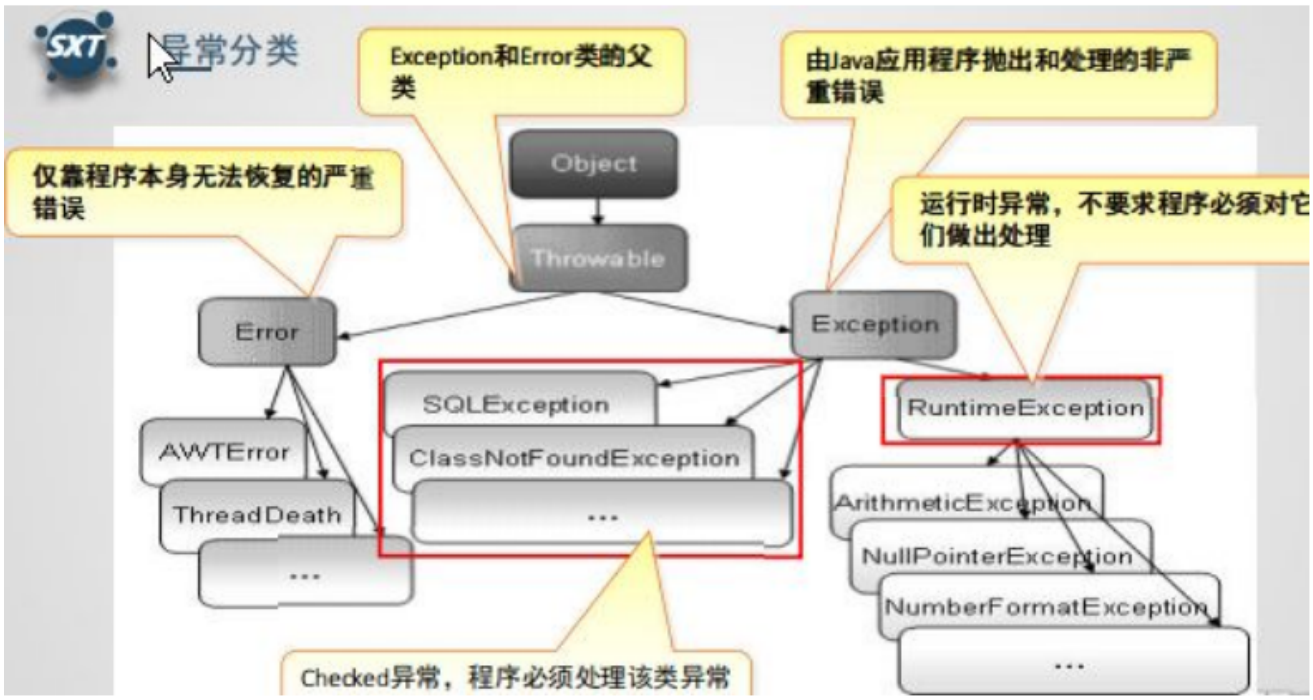
> 在开发过程中，拷贝实例是常见的一种操作，如果一个类中的字段较多，而我们又采用在客户端中逐字段复制的方法进行拷贝操作的话，将不可避免的造成客户端代码繁杂冗长，而且也无法对类中的私有成员进行复制，而如果让需要具备拷贝功能的类实现`Cloneable`接口，并重写`clone()`方法，就可以通过调用`clone()`方法的方式简洁地实现实例拷贝功能

深拷贝(深复制)和浅拷贝(浅复制)是两个比较通用的概念，尤其在C++语言中，若不弄懂，则会在delete的时候出问题，但是我们在这幸好用的是Java。虽然Java自动管理对象的回收，但对于深拷贝(深复制)和浅拷贝(浅复制)，我们还是要给予足够的重视，因为有时这两个概念往往会给我们带来不小的困惑。

浅拷贝是指拷贝对象时仅仅拷贝对象本身（包括对象中的基本变量），而不拷贝对象包含的引用指向的对象。深拷贝不仅拷贝对象本身，而且拷贝对象包含的引用指向的所有对象。举例来说更加清楚：对象 `A1` 中包含对 `B1` 的引用， `B1` 中包含对 `C1` 的引用。浅拷贝 `A1` 得到 `A2` ， `A2` 中依然包含对 `B1` 的引用， `B1` 中依然包含对 `C1` 的引用。深拷贝则是对浅拷贝的递归，深拷贝 `A1` 得到 `A2` ， `A2` 中包含对 `B2` （ `B1` 的 `copy` ）的引用， `B2` 中包含对 `C2` （ `C1` 的 `copy` ）的引用。

若不对clone()方法进行改写，则调用此方法得到的对象即为浅拷贝

异常分类以及处理机制

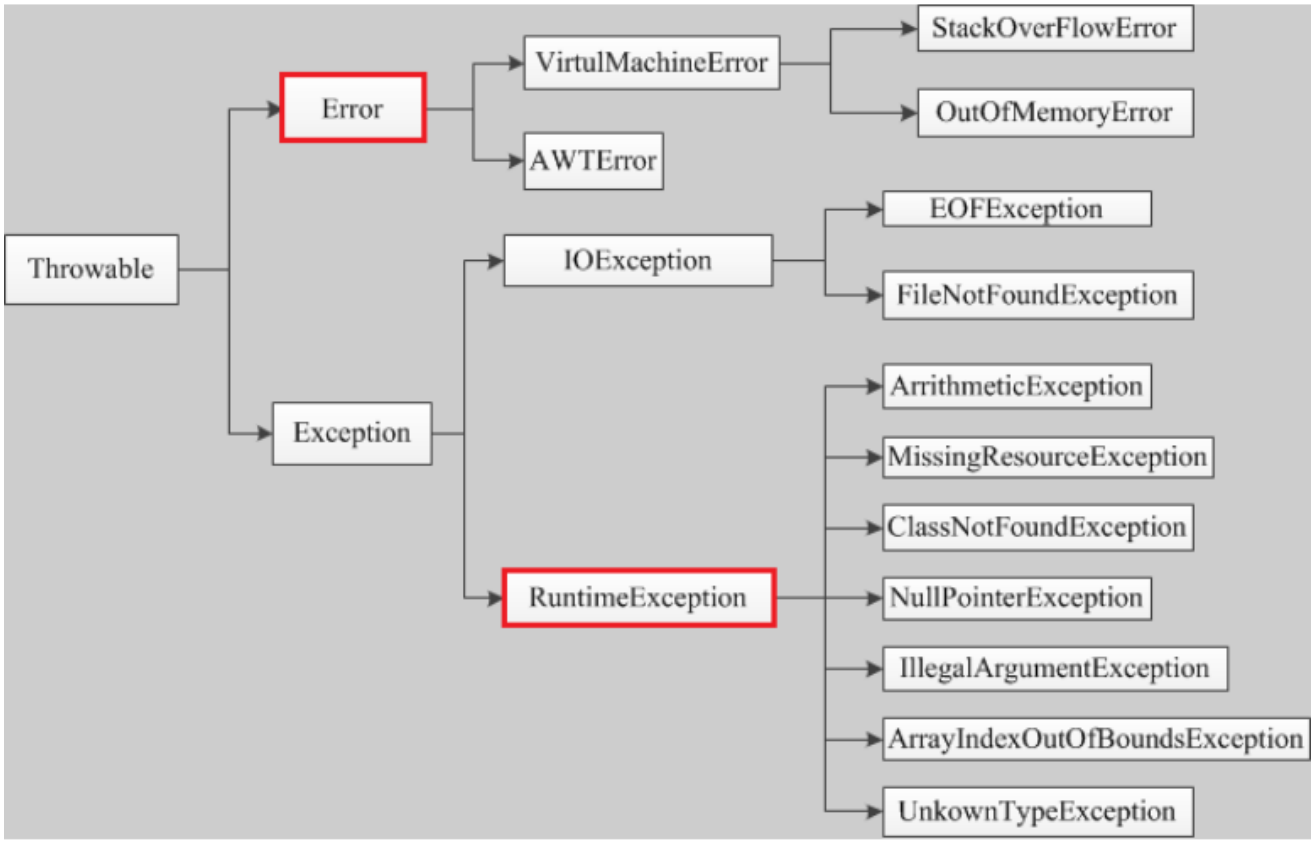


Java标准库内建了一些通用的异常，这些类以Throwable为顶层父类。

Throwable又派生出Error类和Exception类。

错误：Error类以及他的子类的实例，代表了JVM本身的错误。错误不能被程序员通过代码处理，Error很少出现。因此，程序员应该关注Exception为父类的分支下的各种异常类。

异常：Exception以及他的子类，代表程序运行时发送的各种不期望发生的事件。可以被Java异常处理机制使用，是异常处理的核心。



总体上我们根据`Javac`对异常的处理要求，将异常类分为二类。

非检查异常（`unckecked exception`）：`Error`和`RuntimeException`以及他们的子类。`javac`在编译时，不会提示和发现这样的异常，不要求在程序处理这些异常。所以如果愿意，我们可以编写代码处理（使用`try…catch…finally`）这样的异常，也可以不处理。对于这些异常，我们应该修正代码，而不是去通过异常处理器处理 。这样的异常发生的原因多半是代码写的有问题。如除0错误`ArithmeticException`，错误的强制类型转换错误`ClassCastException`，数组索引越界`ArrayIndexOutOfBoundsException`，使用了空对象`NullPointerException`等等。

检查异常（`checked exception`）：除了`Error`和`RuntimeException`的其它异常。`javac`强制要求程序员为这样的异常做预备处理工作（使用`try…catch…finally`或者`throws`）。在方法中要么用`try-catch`语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，于是程序员就应该为这样的异常时刻准备着。如`SQLException`，`IOException`，`ClassNotFoundException`等。

需要明确的是：检查和非检查是对于`javac`来说的，这样就很好理解和区分了。

`wait`和`sleep`的区别

源码如下

```
public class Thread implements Runnable {
public static native void sleep(long millis) throws InterruptedException;
     public static void sleep(long millis, int nanos) throws InterruptedException {
        if (millis < 0) {
            throw new IllegalArgumentException("timeout value is negative");
        }
        if (nanos < 0 || nanos > 999999) {
```

```
                throw new IllegalArgumentException(
                                    "nanosecond timeout value out of range");
            }
            if (nanos >= 500000 || (nanos != 0 && millis == 0)) {
                millis++;
            }
            sleep(millis);
        }
        //...
    }
```

```
public class Object {
     public final native void wait(long timeout) throws InterruptedException;
     public final void wait(long timeout, int nanos) throws InterruptedException {
        if (timeout < 0) {
            throw new IllegalArgumentException("timeout value is negative");
        }
        if (nanos < 0 || nanos > 999999) {
            throw new IllegalArgumentException(
                                "nanosecond timeout value out of range");
        }
        if (nanos > 0) {
            timeout++;
        }
        wait(timeout);
    }
    //...
}
```

1、`sleep`来自`Thread`类，和`wait`来自`Object`类。 2、最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。
3、wait，notify和`notifyAll`只能在同步控制方法或者同步控制块里面使用，而`sleep`可以在任何地方使用(使用范围)

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

(1)`sleep`方法属于`Thread`类中方法，表示让一个线程进入睡眠状态，等待一定的时间之后，自动醒来进入到可运行状态，不会马上进入运行状态，因为线程调度机制恢复线程的运行也需要时间，一个线程对象调用了`sleep`方法之后，并不会释放他所持有的所有对象锁，所以也就不会影响其他进程对象的运行。但在`sleep`的过程中过程中有可能被其他对象调用它的`interrupt()`,产生`InterruptedException`异常，如果你的程序不捕获这个异常，线程就会异常终止，进入`TERMINATED`状态，如果你的程序捕获了这个异常，那么程序就会继续执行catch语句块(可能还有`finally`语句块)以及以后的代码。

注意`sleep()`方法是一个静态方法，也就是说他只对当前对象有效，通过`t.sleep()`让t对象进入`sleep`，这样的做法是错误的，它只会是使当前线程被`sleep`而不是`t`线程

(2)`wait`属于`Object`的成员方法，一旦一个对象调用了wait方法，必须要采用`notify()`和`notifyAll()`方法唤醒该进程;如果线程拥有某个或某些对象的同步锁，那么在调用了`wait()`后，这个线程就会释放它持有的所有同步资源，而不限于这个被调用了`wait()`方法的对象。`wait()`方法也同样会在`wait`的过程中有可能被其他对象调用`interrupt()`方法而产生。

数组在内存中如何分配

对于Java数组的初始化，有以下两种方式，这也是面试中经常考到的经典题目：

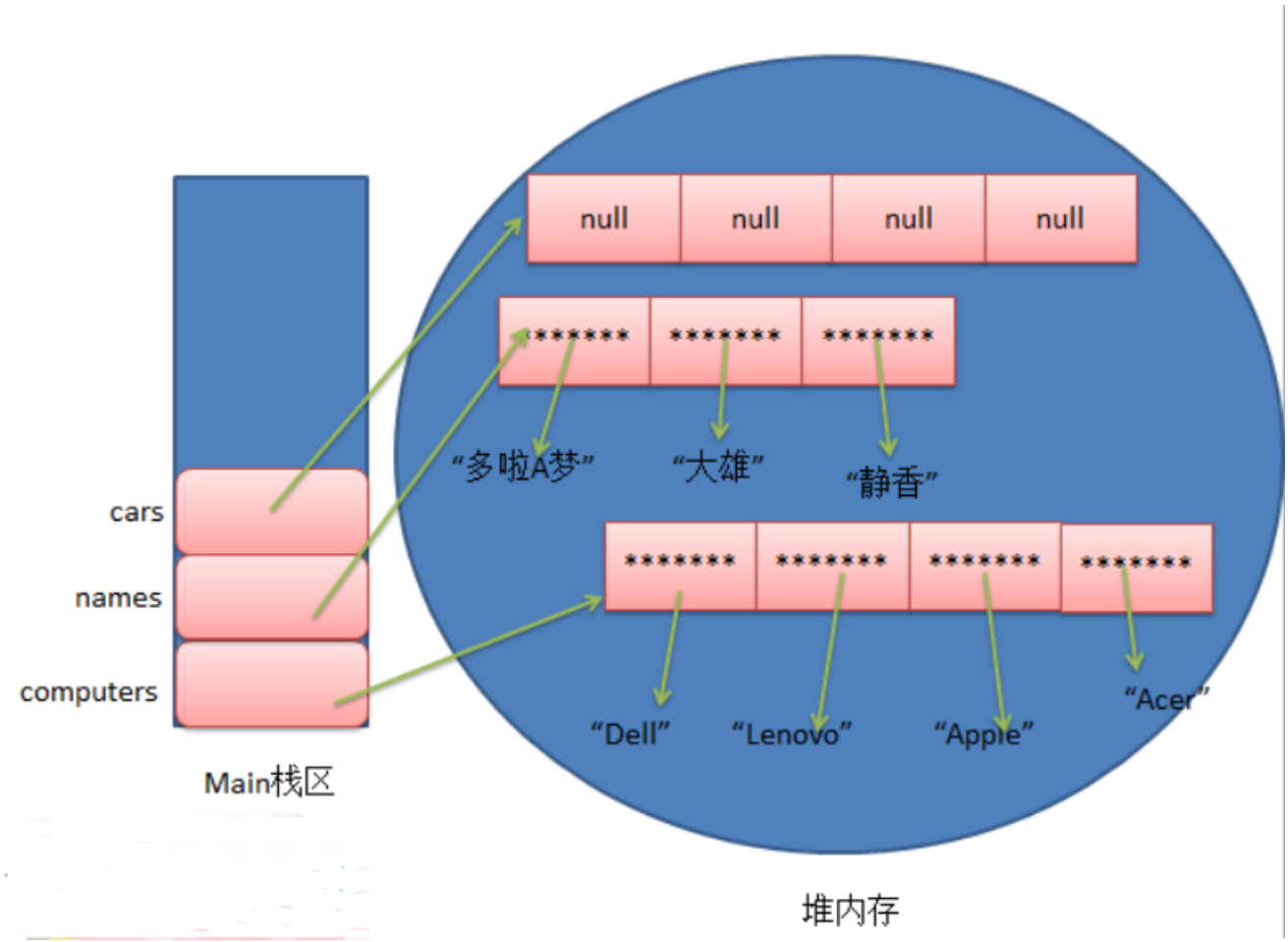
静态初始化：初始化时由程序员显式指定每个数组元素的初始值，由系统决定数组长度，如：

```
//只是指定初始值，并没有指定数组的长度，但是系统为自动决定该数组的长度为4
String[] computers = {"Dell", "Lenovo", "Apple", "Acer"};    //①
//只是指定初始值，并没有指定数组的长度，但是系统为自动决定该数组的长度为3
String[] names = new String[]{"多啦A梦", "大雄", "静香"};    //②
```

动态初始化：初始化时由程序员显示的指定数组的长度，由系统为数据每个元素分配初始值，如：

```
//只是指定了数组的长度，并没有显示的为数组指定初始值，但是系统会默认给数组数组元素分配初始值为null
 String[] cars = new String[4];    //③
```

因为Java数组变量是引用类型的变量，所以上述几行初始化语句执行后，三个数组在内存中的分配情况如下图所示：



由上图可知，静态初始化方式，程序员虽然没有指定数组长度，但是系统已经自动帮我们给分配了，而动态初始化方式，程序员虽然没有显示的指定初始化值，但是因为Java数组是引用类型的变量，所以系统也为每个元素分配了初始化值null，当然不同类型的初始化值也是不一样的，假设是基本类型int类型，那么为系统分配的初始化值也是对应的默认值0。