01、String、StringBuffer、StringBuilder的区别

三者共同之处：都是final类,不允许被继承，主要是从性能和安全性上考虑的，因为这几个类都是经常被使用着，且考虑到防止其中的参数被参数修改影响到其他的应用。

- StringBuffer是线程安全，可以不需要额外的同步用于多线程中;
- StringBuilder是非同步,运行于多线程中就需要使用着单独同步处理，但是速度就比StringBuffer快多了;
- StringBuffer与StringBuilder两者共同之处:可以通过append、indert进行字符串的操作。
- String实现了三个接口:Serializable、Comparable、CarSequence
- StringBuilder只实现了两个接口Serializable、CharSequence，相比之下String的实例可以通过compareTo方法进行比较，其他两个不可以。

这三个类之间的区别主要是在两个方面，即运行速度和线程安全这两方面。

01-01、关于Serializable的问题？

后续的开发中SpringCloud或者Dubbo的远程通讯的时候，实体需要去实现一个所谓的Serializable接口。

- RestTemplate
- RPC
- Redis
- MQ

```
String name = "飞哥是一个帅哥";
A服务 resttemplate.get("/xxxxx","xxxxx");--------GET----> B服务

User user = new User();// implments Serializable -- Socket
A服务 resttemplate.get("/xxxxx",user);--------GET----> B服务
```

02-02、String问题和弊端？

就因为String存在问题和弊端所以才有了：StringBuffer、StringBuilder。

```
String sql = "select * from table";
sql += "where a = 1 and b=1";
sql += " and c = 1 and d=1";
```

```
StringBuffer stringbuffer = new StringBuffer();
stringbuffer.append("select * from table");
stringbuffer.append("where a = 1 and b=1");
stringbuffer.append("and c = 1 and d=1");
```

02-03、StringBuffer是线程安全，StringBuilder是非线程安全的。

- 在我们实际应用开发中，我大部环境都是多线程的并发请求方式。也就是说98%都不用考虑线程安全的问题，98%是不需要考虑线程安全的问题，2%的业务可能要考虑到线程安全。
- 和硬件有关--CPU-多核时代---为了不浪费CPU，大部分语言都为迎合硬件都提供多线程。多线程的处理可以提升网站的并发执行能力，如果没有多线程，是单线程的话，如果我请求百度，你就必须等我执行完毕，才能访问。如果这个1000W

1、首先说运行速度，或者说是执行速度

在这方面运行速度快慢为：**StringBuilder > StringBuffer > String**

String最慢的原因：**String**为字符串常量，而**StringBuilder**和**StringBuffer**均为字符串变量，即**String**对象一旦创建之后该对象是不可更改的，但后两者的对象是变量，是可以更改的。以下面一段代码为例：

```
String str="abc";
System.out.println(str);
str=str+"de";
System.out.println(str);
```

运行这段代码会发现先输出“abc”，然后又输出“abcde”，好像是str这个对象被更改了，其实，这只是一种假象罢了，JVM对于这几行代码是这样处理的，首先创建一个String对象str，并把“abc”赋值给str，然后在第三行中，其实JVM又创建了一个新的对象也名为str，然后再把原来的str的值和“de”加起来再赋值给新的str，而原来的str就会被JVM的垃圾回收机制（GC）给回收掉了，所以，str实际上并没有被更改，也就是前面说的String对象一旦创建之后就不可更改了。所以，Java中对String对象进行的操作实际上是一个不断创建新的对象并且将旧的对象回收的一个过程，所以执行速度很慢。

而StringBuilder和StringBuffer的对象是变量，对变量进行操作就是直接对该对象进行更改，而不进行创建和回收的操作，所以速度要比String快很多。

另外，有时候我们会这样对字符串进行赋值

```
String str="abc"+"de";
StringBuilder stringBuilder=new
StringBuilder().append("abc").append("de");
System.out.println(str);
System.out.println(stringBuilder.toString());
```

这样输出结果也是“abcde”和“abcde”，但是String的速度却比StringBuilder的反应速度要快很多，这是因为第1行中的操作和String str="abcde";是完全一样的，所以会很快，而如果写成下面这种形式

```
public static void main(String[] args) {
        long a=new Date().getTime();
        String cc="";
        int n=10000;
        for (int i = 0; i < n; i++) {
            cc+="."+i;
        }
        System.out.println("String使用的时间"+
(System.currentTimeMillis()-a)/1000.0+"s");
        long s1=System.currentTimeMillis();
        StringBuilder sb=new StringBuilder();
        for (int i = 0; i < n; i++) {
            sb.append("."+i);
        }
```

```
        System.out.println("StringBuilder使用的时间"+
(System.currentTimeMillis()-s1)/1000.0+"s");
        long s2=System.currentTimeMillis();
        StringBuffer sbf=new StringBuffer();
        for (int i = 0; i < n; i++) {
            sbf.append("."+i);
        }
        System.out.println("StringBuffer使用的时间"+
(System.currentTimeMillis()-s2)/1000.0+"s");
    }
```

2、再来说线程安全

在线程安全上，**StringBuilder**是线程不安全的，而**StringBuffer**是线程安全的

如果一个StringBuffer对象在字符串缓冲区被多个线程使用时，StringBuffer中很多方法可以带有synchronized关键字，所以可以保证线程是安全的，但StringBuilder的方法则没有该关键字，所以不能保证线程安全，有可能会出现一些错误的操作。所以如果要进行的操作是多线程的，那么就要使用StringBuffer，但是在单线程的情况下，还是建议使用速度比较快的StringBuilder。

（一个线程访问一个对象中的synchronized(this)同步代码块时，其他试图访问该对象的线程将被阻塞）

3、总结一下

String：适用于少量的字符串操作的情况

StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况

StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况

02、java里equals和hashCode之间什么关系

推荐查看文章：https://www.cnblogs.com/tanshaoshenghao/p/10915055.html

02-01、两者的关系？

两者没有关系，

equals

- 是判断两个对象是否相同的方法。默认所有的类，如果没有覆盖都是调用Object的eqauls方法。也是一个 ==比较

```
package com.kuangstudy.equals;

public class User {

    private String name;
    private int age;
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

}
```

```
package com.kuangstudy.equals;

public class UserTest {

    public static void main(String[] args) {
        User user1 = new User("yykk",1);
        User user2 = new User("yykk",2);

        System.out.println(user1.equals(user2));
    }
}
```

核心代码分析

```
System.out.println(user1.equals(user2));
```

得到结果确实是：false

为什么是**false** 呢？

- 1：eqauls方法从哪里来的？从父类而来，如果一个类没有继承任何类，它的父类就是Object
- 2：因为Object里的equals方法是内存地址比较

```
  public boolean equals(Object obj) {
        return (this == obj);
    }
```

- 所以这个代码的话，就等价于

```
System.out.println(user1==user2);
```

- 所以答案是false.

如果我想让属性值相同的对象，代表是同一个对象的话？

答案：重写Object的eqauls，重写指定规则。

```
System.out.println(user1.equals(user2));
```

```
@Override
public boolean equals(Object user2) {
    if (user1 == user2) return true;
    if (user2 == null || getClass() != o.getClass()) return false;
    User user = (User) user2;
    return user1.age == user.age && Objects.equals(user1.name,
user.name);
}
```

飞哥什么样子的情况下要覆盖呢？

- 要过滤，比如一个集合中有很多对象，这个对象很多值相同，我想过滤掉，可以考虑覆盖eqauls方法
- Set功能---不允许重复无序，底层是：HashMap

02-01、概述

如果要比较实际内存中的内容，那就要用equals方法，但是！！！

如果是你自己定义的一个类，比较自定义类用equals和是一样的，都是比较句柄地址，因为自定义的类是继承于object，而object中的equals就是用来实现的，你可以看源码。

那为什么我们用的String等等类型equals是比较实际内容呢，是因为String等常用类已经重写了object中的equals方法，让equals来比较实际内容。

在一般的应用中你不需要了解hashcode的用法，但当你用到hashmap，hashset等集合类时要注意下hashcode。

02-01、 两个对象的 **hashCode()** 相同， 那么 **equals()** 也一定为 **true**吗？

不对，两个对象的 hashCode() 相同，equals() 不一定 true。

代码示例：

```
String str1 = "keep";
String str2 = "brother";
System. out. println(String. format("str1：%d | str2：%d",  str1.
hashCode(),str2. hashCode()));
System. out. println(str1. equals(str2));
```

执行的结果：

```
str1：1179395 | str2：1179395

false
```

代码解读：很显然“keep”和“brother”的 hashCode() 相同，然而 equals() 则为 false，因为在散列表中，hashCode() 相等即两个键值对的哈希值相等，然而哈希值相等，并不一定能得出键值对相等。

01、Java中的Collection框架

在工作中我们每天在使用集合框架，但是面试中每次都不厌其烦的背诵集合框架的底层源码。真的是面试造火箭，开发拧螺丝。Java中的集合框架融合了很多算法和数据结构的思想，让我们的开发者不用去思考这些问题。因为优秀的算法和数据结构被封装到了Java的集合框架之中。相比java语言其他的语言，比如C，可能需要程序员自己去实现和定义这些算法数据结构。

01、常见的数据结构

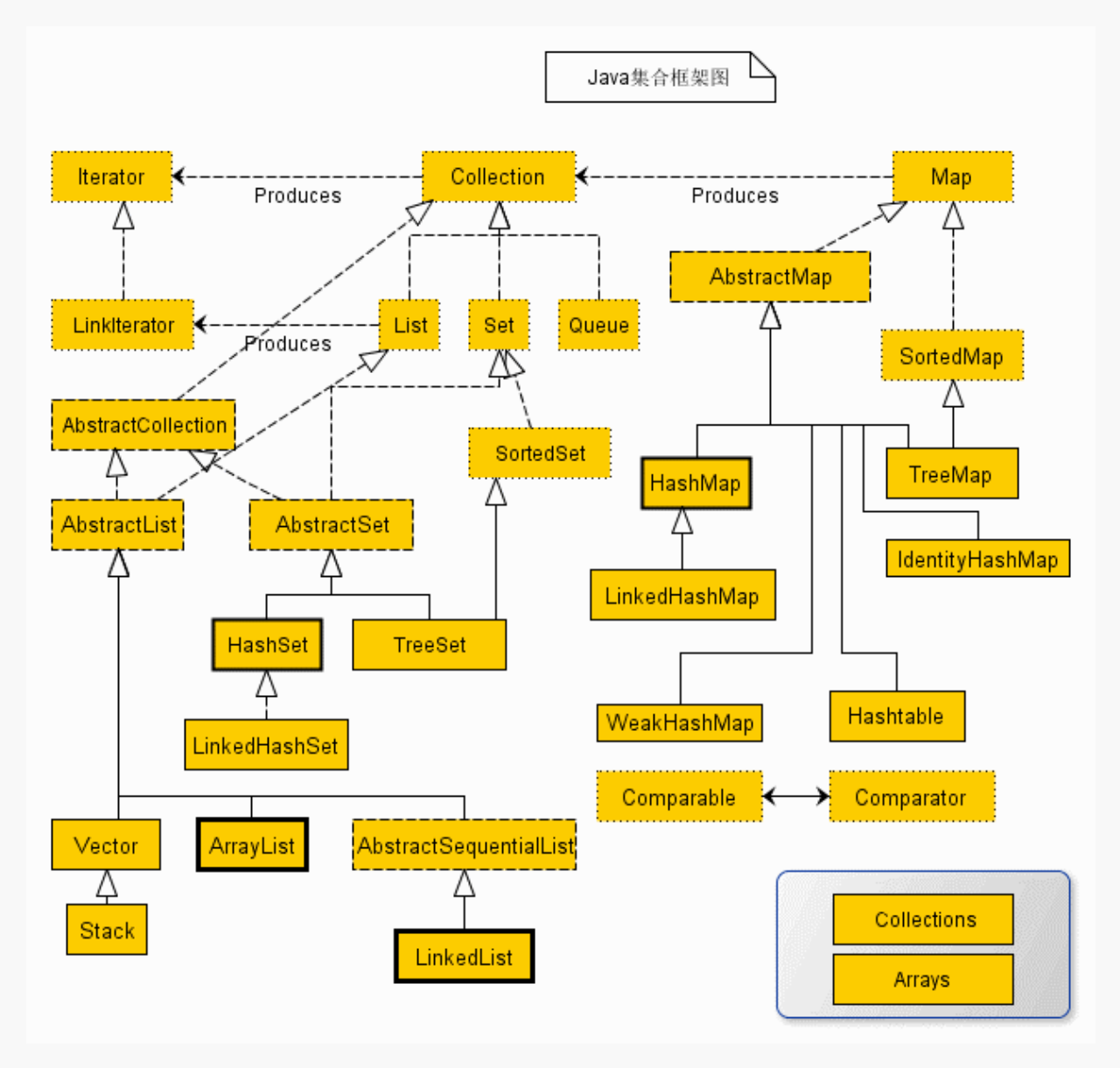
- 数组和链表的区别
- 链表的操作、如反转、链表环路检测、双向链表、循环链表相关操作
- 队列、栈的应用
- 二叉树的遍历方式及其递归和非递归的实现
- 红黑树的旋转
- B+Tree

数据结构：是一种内存分配的一种容器，是用来存储数据使用。它保证内存是一种连续的结构形式。

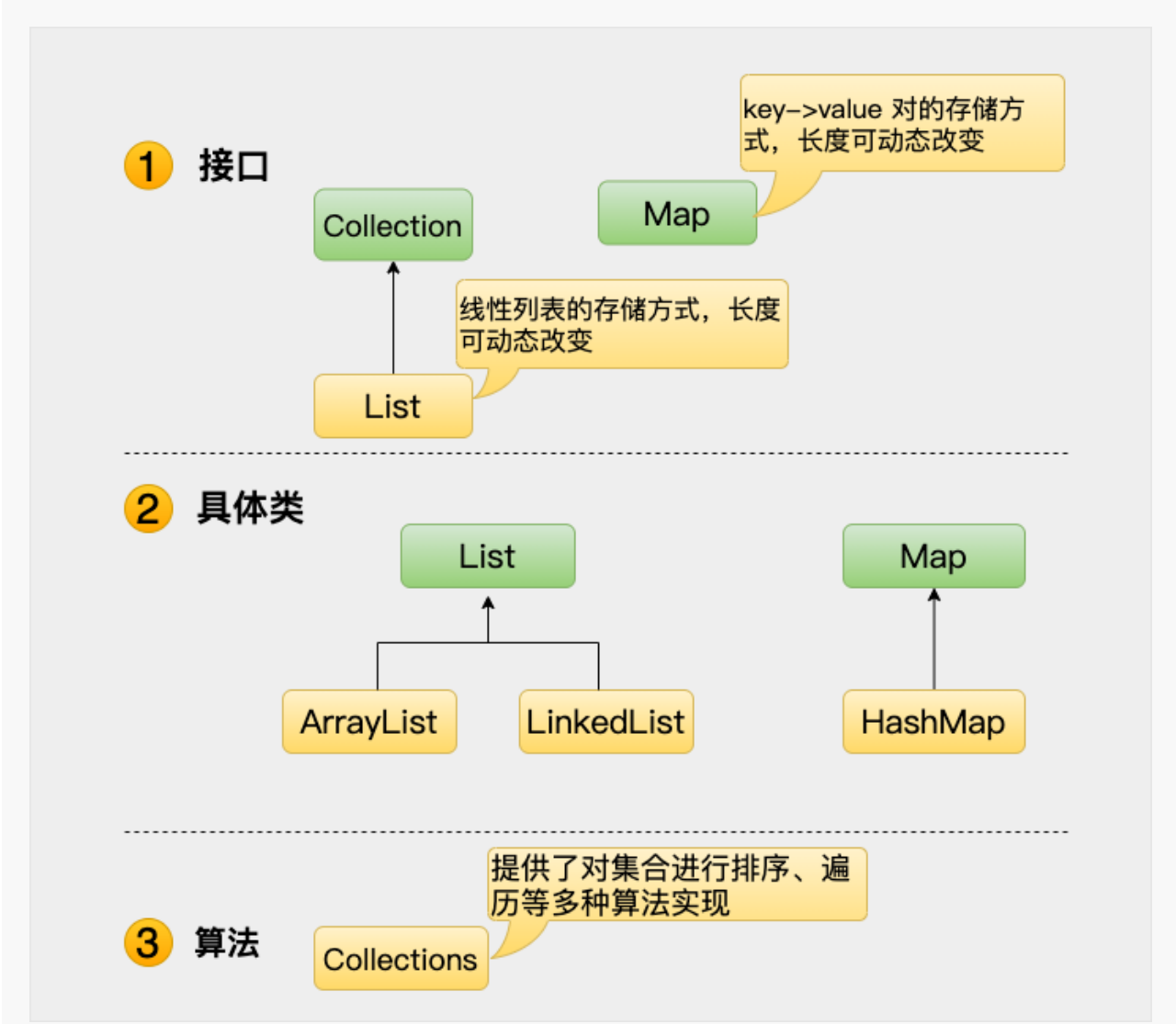
02、算法考点

- 内部排序；如递归排序、交换排序（冒泡，快速）、选择排序、插入排序。
- 外部排序；应该掌握如何利用有限的内存配合海量的外部存储来处理超大的数据集。
- 那些排序是不稳定的，稳定意味着什么？
- 不同数据集，各种排序的最好和最差的情况。
- 如何优化算法。

03、Java集合框架示意图



04、集合框架体系如图所示



05、Set和List的区别

- \1. Set 接口实例存储的是无序的，不重复的数据。List 接口实例存储的是有序的，可以重复的元素。
- \2. Set检索效率低下，删除和插入效率高，插入和删除不会引起元素位置改变 <实现类有**HashSet,TreeSet>**。
- \3. List和数组类似，可以动态增长，根据实际存储的数据的长度自动增长List的长度。查找元素效率高，插入删除效率低，因为会引起其他元素位置改变 <实现类有**ArrayList,LinkedList,Vector>** 。

02 Java中的数组

```
package com.example.kuangstudyjava.arraydemo;

public class ArrayDemo {

    public static void main(String[] args) {

        // 数组定义
        String[] names1 = {"yykk", "小卡", "小周"};
        // 数组定义
        String[] names2 = new String[3];
        names2[0] = "张三1";
        names2[1] = "张三2";
        names2[2] = "张三3";
        names2[3] = "xxx"; // ArrayIndexOutOfBoundsException  数组越界异常
    }
}
```

上面的代码是java中数组的定义方式：

1、java的数组是不可变，意思：长度一旦被固定就不能添加超过长度位的元素，否则就会报：数组越界异常

2、java中数组是不能够动态的进行元素追加。

```
/**
     * 查询小程序的轮播图
     *
     * @param pageNo   起始页
     * @param pageSize 每页显示多少条
     * @return
     */
    public List<Banner> findBanners(int pageNo, int pageSize) {
        // 1 :设置查询分页信息
        Page<Banner> page = new Page<>(pageNo, pageSize);
        // 2 :设置查询条件
        LambdaQueryWrapper<Banner> lambdaQueryWrapper = new LambdaQueryWrapper<>();
```

```
        // 查询发布的轮播图信息
        lambdaQueryWrapper.eq(Banner::getStatus, 1);
        // 查询进行appsort的升序返回
        lambdaQueryWrapper.orderByAsc(Banner::getAppsort);
        // 3 :查询返回
        Page<Banner> bannerPage = this.page(page, lambdaQueryWrapper);
        // 4:  分页返回轮播图信息
        List<Banner> records = bannerPage.getRecords();
        return CollectionUtils.isEmpty(records) ? new ArrayList<>() :
records;
    }


// 还原成数组

/**
     * 查询小程序的轮播图
     *
     * @param pageNo    起始页
     * @param pageSize 每页显示多少条
     * @return
     */
    public Banner[] findBanners(int pageNo) {

        //举例子：banner 今天：3条
        //举例子：banner 今天：8条
        //举例子：banner 今天：10条
        Banner[] bannerArr = new Banner[?];

        return bannerArr;
    }
```

06、Java集合List和Set分析

```
package com.example.kuangstudyjava;

import java.util.LinkedList;
import java.util.TreeSet;

public class ListDemo {

    public static void main(String[] args) {
        LinkedList linkedList = new LinkedList();
        linkedList.add("aaaa");
        linkedList.add("bbbb");
        linkedList.add("aaaa");
        linkedList.add("ccccc");
        linkedList.add("dddd");
        linkedList.add("eeee");
        System.out.println(linkedList);
        TreeSet treeSet = new TreeSet();
        treeSet.add("aaaa");
        treeSet.add("dddd");
        treeSet.add("ccccc");
        treeSet.add("bbbb");
        treeSet.add("aaaa");
        treeSet.add("eeee");
        System.out.println(treeSet);
    }
}
```

ArrayList 追求读 追求写 LinkedList

03、ArrayList分析

ArrayList 动态数组，解决java数组的动态扩容的问题。这里面绝对是用数组来实现。

1、数组的类型是什么？

```
object[]
```

2、ArrayList的数组的最大极限是多少？

```
    // 扩容的方法
    private static int hugeCapacity(int minCapacity) {
         if (minCapacity < 0) throw new OutOfMemoryError();
         // 扩容的最大边界值
         // 1：防止栈溢出和越界 2：提前让你达到最大，减少扩容一次机会
         // 这个时候你数量在集合中以及存储到极限值，任何一次数组迁移和复制都可能引发内存溢出。
         return (minCapacity > MAX_ARRAY_SIZE) ? Integer.MAX_VALUE : MAX_ARRAY_SIZE;
     }

     private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```

初始化长度又是多少呢？

```
private static final int DEFAULT_CAPACITY = 10;
```

03、ArrayList无参构造函数

无论new ArrayList() 还是new HashMap() 这种无参数的构造函数，都采用是一种：延迟加载的机制来完成初始化

```
如果是jdk1.8之前，确实  this.elementData = new Object[initialCapacity];
并初始化10长度的数组
```

答案：为什么要这样做呢？怕你定义new ArrayList()不用，这10长度的数组空间就白白浪费在哪里。

它什么时候初始化呢？

```
在add()方法的时候
```

当然你也可以使用，有长度的构造函数

```
ArrayList arrayList = new ArrayList(10);
```

3、如何进行动态扩容的呢？扩容时机是什么？

```
add()时候，扩容的长度是index=10的时候进行一次扩容15，index=16的时候  index = 24
```

1、底层是一个全局的数组结构进行元素的存储

2、在实现构造函数的，进行对象数组的初始化，初始化了一个并且指定定长的扩容。默认长度是：10

3、当调用add()方法添加元素的时候，如果超过固定长度10，就会调用grow方法进行扩容

```
ArrayList源码中的数组扩容的疑问
```

```
private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

这里的int newCapacity = oldCapacity + (oldCapacity >> 1);这句怎么理解?
------解决方案--------------------
int newCapacity = oldCapacity + (oldCapacity >> 1)
相当于int newCapacity = oldCapacity + (oldCapacity / 2)
也就是说newCapacity = oldCapacity * (1.5)
扩容50%的意思

4、关于数组的最大容量问题

如果`newCapacity - MAX_ARRAY_SIZE > 0`就会调用`hugeCapacity`， 判断传入参数`minCapacity`的大小，当 `minCapacity > MAX_ARRAY_SIZE`时， `ArrayList`的最大容量就会设置为`Integer.MAX_VALUE`。

```
/**
    * The maximum size of array to allocate.
    * Some VMs reserve some header words in an array.
    * Attempts to allocate larger arrays may result in
    * OutOfMemoryError: Requested array size exceeds VM limit
*/
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```

5、关于删除元素

```
源代码实现
public E remove(int index) {
        rangeCheck(index);//先判断数组是否越界

        modCount++;
        E oldValue = elementData(index);
```

```
        //处理数据
          int numMoved = size - index - 1;
        //remove方法是将原数组的指定下标位置后面的值复制好然后再覆盖原有的指定下标位置，再将最后的一个置为空方便gc
        调用的system.arraytcopy
        public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
        代码解释:
        Object src : 原数组
        int srcPos : 从元数据的起始位置开始
        Object dest : 目标数组
        int destPos : 目标数组的开始起始位置
        int length  : 要copy的数组的长度
如果一个arraylist集合有0,1，2,3,4,5的数据
然后 remove(3)从原数组的下标为4就是4开始，复制两个长度也就是4和5复制这两个，
接着 从目标数组开始（这里也是当前数组）的下标为3这里将其覆盖也就是变成0,1,2,4,5,5,
最后将最后一位置为null就变成0,1,2,4,5，null

        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // clear to let GC do its work

        return oldValue;
    }
```

02、LinkedeList和ArrayList的区别

1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。
2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。
3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

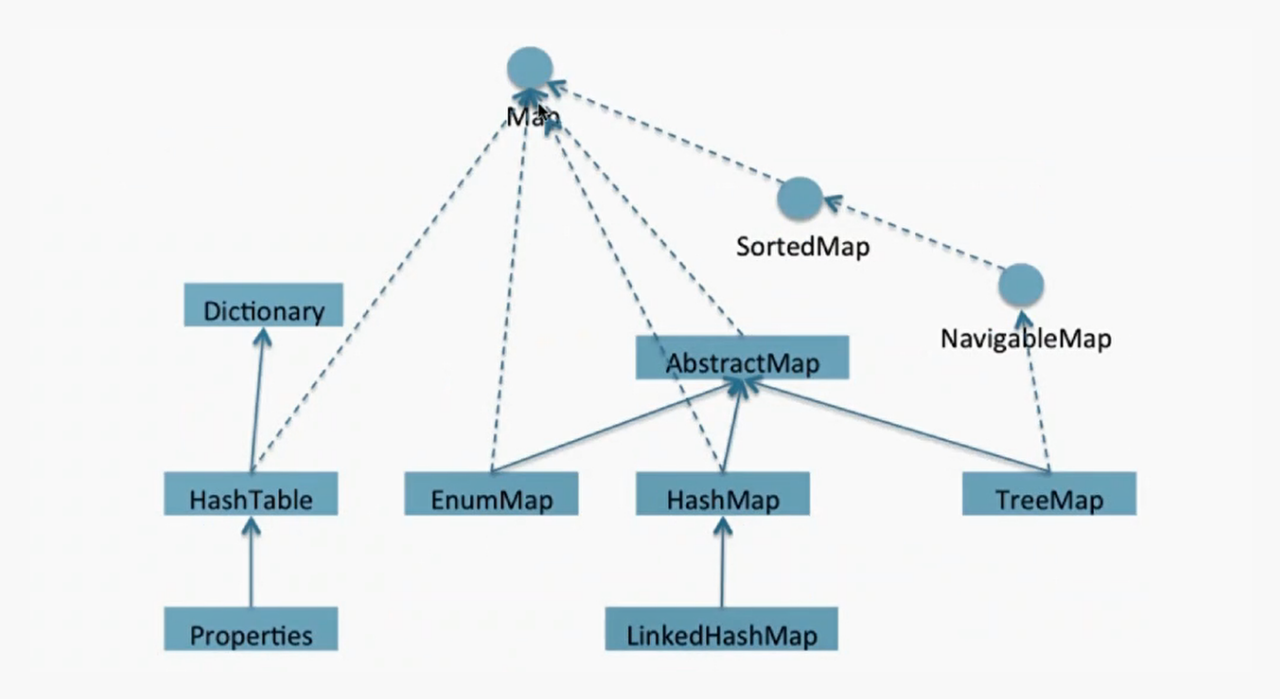
https://www.cnblogs.com/jiezzy/archive/2012/08/28/2660563.html

03、HashSet源码分析

```
public HashSet(Collection<? extends E> c) {
        map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
        addAll(c);
    }
```

Hashset的底层原理是：HashMap

03、Map的整体结构



04、HashMap

1、数组的长度是多少呢?

2、如何扩容的呢？

3、链表里存储是什么东西？

4、jdk1.8增加红黑树的数据结构，它能解决什么问题？

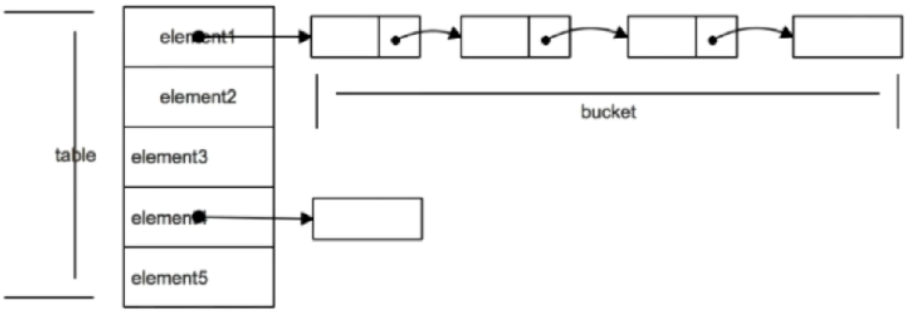
5、什么样子的情况下，会把链表变成红黑树呢？

6、什么样子的情况下，会红黑树变成链表呢？

7、如何解决hash算法的碰撞重复的问题呢？

8、为什么 ^ hash >>16 不能 &？

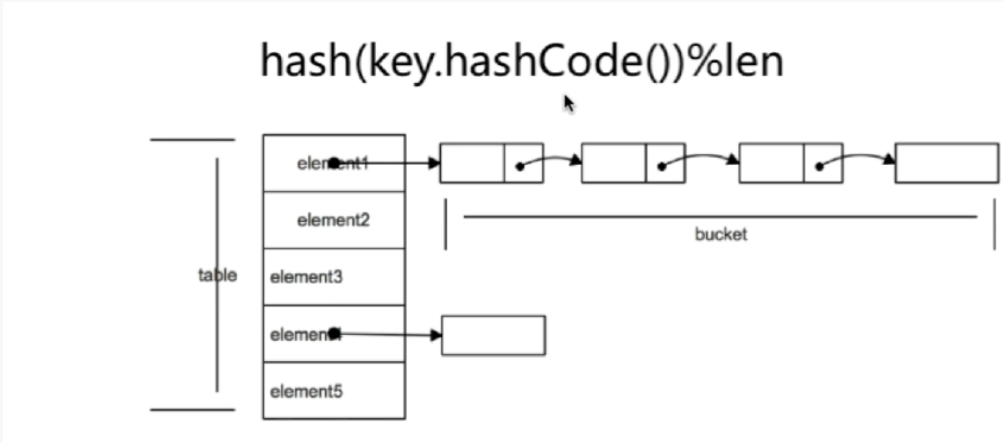
01、HashMap(jdk1.8前)：数组 （查询快）+ 链表（添加删除快）



1、默认情况：hashmap的数组的长度是：16 。

2、长度为16的数组的头部，存储的是链表的头部元素节点。

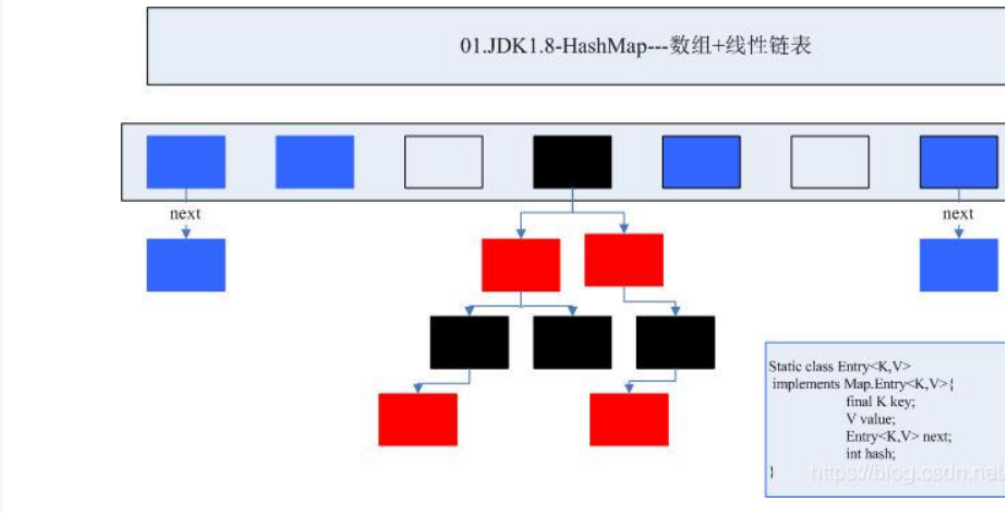
3、通过类似于hash(key.hashCode()) % len的hash算法，来确定元素存储在数组的那个位置。



4：通过上面的hash可能会存在一种极端的情况，就是大量的数据进行hash取模的时候，换算出来的元素数组元素位置都是一样的，元素都分配到一个bucket中，这样会使得某个数组的列表会变得非常的长。因为链表查询需要从头部逐个去遍历，因为性能恶化的情况是由：O1变成On。

5：map的key是使用Set来存储，values采用的Collection来存储，所以key不允许重复，values可以允许重复。

02、HashMap：JDK1.8以后：数组+链表+红黑树



1、性能从On提高到了O(logn)

JDK1.8-HashMap源码分析

1、jdk1.7 和jdk1.8 存储数据的方式都是采用数组：

jdk1.7

```
Entry<K,V>[] table;
```

jdk1.8

```
Node<K,V>[] table;
```

Node对象是有键值对够成：

```
/**
     * Basic hash bin node, used for most entries.  (See below for
     * TreeNode subclass, and in LinkedHashMap for its Entry
subclass.)
     */
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    // 键
    final K key;
    // 值
    V value;
    //链表指向得下一个节点
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    public final K getKey()        { return key; }
    public final V getValue()      { return value; }
    public final String toString() { return key + "=" + value; }

    public final int hashCode() {
```

```
        return Objects.hashCode(key) ^ Objects.hashCode(value);
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    public final boolean equals(Object o) {
        if (o == this)
            return true;
        if (o instanceof Map.Entry) {
            Map.Entry<?,?> e = (Map.Entry<?,?>)o;
            if (Objects.equals(key, e.getKey()) &&
                Objects.equals(value, e.getValue()))
                return true;
        }
        return false;
    }
}
```

2、在jdk1.8中对HashMap进行了优化，在发生hash碰撞，不再采用头插法方式，而是直接插入链表尾部，因此不会出现环形链表的情况，但是在多线程的情况下仍然不安全。

3、然后通过hash值，决定了键值对在数组得寻址，hash值相同得键值对以链表得形式进行存储。但是如果链表得大小超过TREEIFY_THRESHOLD>=8。就会进行Tree化。 当某个数组中的链表元素，删除以后的个数低于UNTREEIFY_THRESHOLD<=6的时候，会将红黑树转化成链表。

JDK1.8-HashMap构造器

这里的this.threshold本应该指的是HashMap的下次扩容的阈值，仔细看你会发现这里并没有对组成HashMap的数组按你写的大小进行初始化，而是把你的参数赋值给下次的扩容的阈值。

```
/**
   * Constructs an empty <tt>HashMap</tt> with the default initial
capacity
   * (16) and the default load factor (0.75).
*/
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields
defaulted
}
```

默认构造函数，默认情况下没有对数组进行初始化好，可以推断出HashMap是采用lazyLoad的原则，将　默认数组的初始化的过程放入到了使用的时候进行。也就是调用put方法。

```
public HashMap(int initialCapacity, float loadFactor) {
        //如果初始容量小0 则报错
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial
capacity: " +
                                               initialCapacity);
        //如果初始容量 大于冗余的最大容量 2的30次幂，
        //则改变初始容量为允许的最大容量
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        //如果传入的负载因子小于等于 0 或者 负载因子为空，则报错
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: "
+
                                               loadFactor);
        //赋值传入的负载因子
        this.loadFactor = loadFactor;
        //通过这个方法可以实现提前设置一个较为稳定的容量，从而避免频繁扩容导致的性能
下降。
        this.threshold = tableSizeFor(initialCapacity);
```

```
    }
```

HashMap中有这样一段代码，当初始化HashMap时，如果指定了初始容量initialCapacity，由于哈希桶的数目必须是2的n次幂，因此要把initialCapacity转化为比它大的最小的2的n次幂数，例如initialCapacity = 10 ，那就返回16， initialCapacity = 17，那么就返回32。

```
/**
这个函数是用来对你申请的容量进行处理让他变成最接近你申请的容量的2次幂的大小，
这里注意：假如你申请的容量为0，最后处理的结果会变成1，代表着你最小的容量为1
**/
static final int tableSizeFor(int cap) {
        int n = cap - 1;//n 为初始化容量 - 1
        n |= n >>> 1;
        n |= n >>> 2;
        n |= n >>> 4;
        n |= n >>> 8;
        n |= n >>> 16;
         /**
         如果（初始化容量-1）小于0，则初始化容量为1
         如果 （初始化容量-1）的值大于 允许的最大容量，则把容量设置为允许的最大容量
         否则 设置为 （（初始化容量-1） + 1）
        **/
        return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ?
MAXIMUM_CAPACITY : n + 1;
    }
```

JDK1.8-HashMap的put操作

1、如果HashMap没有初始化过，则初始化。
2、对Key求Hash值，然后在计算下标
3、如果没有碰撞、直接放入桶中
4、如果碰撞了，以链表的方式链接到后面
5、如果链表的长度超过阈值、就把链表转成红黑树
6、如果链表的长度低于6，就把红黑树转回链表
7、如果节点已经存在就替换旧值
8、如果桶满了（容量16 * 加载因子0.75），就需要resize扩容2倍后重排。

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
```

```
                boolean evict) {

    Node<K,V>[] tab; Node<K,V> p; int n, i;
    /**

     *  tab = table 值为当前哈希表的值
     *  n = tab.length 值为当前哈希表长度
     *  如果当前哈希表为空 或者 当前哈希表长度为0
     *  则tab = resize
     *  n = resize.length;
     */

    if ((tab = table) == null || (n = tab.length) == 0)
        //没有hash碰撞时，后续值直接覆盖
        n = (tab = resize()).length;

    /**
     * i = (n - 1) & hash 得到的值为当前hash应该插入的数组位置
     * p = tab[i];  把p 指向哈希表下标为i的位置
     * 如果该位置为空 ，代表该哈希位置还未插入过数据，
     * 则把当前要插入的数据生成新Node直接插入
     */
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {//如果哈希表下标为i的的位置有数据则执行以下操作
        Node<K,V> e; K k;
        /**
         * 判断一个两个node是否相同，有两个指标 1.两个node的hash值相同；2.两个
node的key相同
         * 注意：当前p指向哈希表中下标为i的位置的首位
         * 如果首位的哈希值与要新插入的哈希值相同  并且
         * k = p.key
         * (k == key || (key != null && key.equeals(k)；
         * 其实意思就是如果要新插入的node的与当前p指向的位置为同一个元素
         * 则 e = p;
         */
        if (p.hash == hash &&
                ((k = p.key) == key || (key != null &&
key.equals(k))))
            e = p;
        /**
         * 注意：当前p指向哈希表中下标为i的位置的首位
         * 如果当前p指向的位置的类型已经是红黑树
```

```
             * 则把新node数据直接插入红黑树中
             */
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
            /**
             * 注意：当前p指向哈希表中下标为i的位置的首位
             * 否则当前p指向的哈希表中下标为i的位置是一个线性链表
             */
            else {
                for (int binCount = 0; ; ++binCount) {
                    /**
                     * 注意：当前p指向哈希表中下标为i的位置的首位
                     * 循环执行 e = p.next ;  直到 e == null  其实就是循环访问线
性链表直到线性链表结尾
                     * 把要插入的值生成新Node插入线性链表结尾
                     */
                    if ((e = p.next) == null) {
                        //把要插入的值生成新Node插入线性链表结尾
                        p.next = newNode(hash, key, value, null);
                        //如果操作的长度大于等于（8 - 1） 则转红黑树
TREEIFY_THRESHOLD为转红黑树的门槛因子
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for
1st
                            treeifyBin(tab, hash);//把当前线性链表转为红黑树
                        break;//插入新数据后跳出for循环
                    }

                    /**
                     * 循环访问线性链表的过程中对每一个node元素与要插入的元素进行判断
                     * 判断一个两个node是否相同，有两个指标 1.两个node的hash值相
同；2.两个node的key相
                     * 如果 为同一个元素则跳出for循环
                     */
                    if (e.hash == hash &&
                            ((k = e.key) == key || (key != null &&
key.equals(k))))
                        break;
                    //如果未到达线性链表末尾且当前线性链表中不存在于要插入的元素相同的
node则继续for循环

                    p = e;
                }
```

```
        }

        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }

    ++modCount;//增加修改的次数
    if (++size > threshold)//判断当前哈希表长度是否超过负载门槛
        resize();//哈希表扩容
    afterNodeInsertion(evict);
    return null;
}
```

每个**put**操作都有可能会触发哈希表扩容

```
/**
 * JDK1.8---哈希表扩容
 * @return
 */
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    /**
     * 获取原哈希表容量   如果哈希表为空则容量为0 ， 否则为原哈希表长度
     */
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    /**
     * 获取原哈希表扩容门槛
     */
    int oldThr = threshold;
    /**
     * 初始化新容量和新扩容门槛为0
     */
    int newCap, newThr = 0;
    /**
     //如果原容量大于 0
     ---这个if语句中计算进行扩容后的容量及新的负载门槛
     */
```

```
    if (oldCap > 0) {
            //判断原容量是否大于等于HashMap允许的容量最大值 2的30次幂
            if (oldCap >= MAXIMUM_CAPACITY) {
                //如果原容量已经大于等于了允许的最大容量，
                // 则把当前HashMap的扩容门槛设置为Integer允许的最大值
                    threshold = Integer.MAX_VALUE;
                    return oldTab;//不再扩容直接返回
                }
            /**
             * newCap = oldCap << 1 ;  类似 newCap = oldCap * 2 移位操作
更加高效
             * 表示把原容量的二进制位向左移动一位，
             * 扩大为原来的2倍，同样还是2的n次幂
             * 如果新的数组容量小于HashMap允许的容量最大值 2的30次幂
             * 并且 原数组容量小于默认的初始化数组容量 2的4次幂 =16
              */
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                             oldCap >= DEFAULT_INITIAL_CAPACITY)
            /**
             * //新的扩容门槛为原来的扩容门槛的2倍，同样二进制左移操作
             //类似 newThr = oldThr * 2 移位操作更加高效
             */
                newThr = oldThr << 1; // double threshold
        }
    /**
     * 如果 原数组容量小于等于零
     * 并且 原负载门槛大于0 则
     * 新数组容量为原负载门槛大小
      */
    else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
    /**
     * 这个elese语句   初始化默认容量和默认负载门槛
     * 如果原数组容量小于等于0
     * 并且原负载门槛也小于等于0
     * 则
     * 新 数组容量为   默认HashMap设置的默认初始化容量   1《4 = 2的4次幂 = 16
     * 新 负载门槛为   默认负载因子（0.75f） * 16;
     */
    else {               // zero initial threshold signifies using
defaults
            newCap = DEFAULT_INITIAL_CAPACITY;
```

```
            newThr = (int)(DEFAULT_LOAD_FACTOR *
DEFAULT_INITIAL_CAPACITY);
        }
    /**
     * 如果新负载门槛为 0  则开始使用新的 数组容量进行计算
      */
    if (newThr == 0) {
            // 新的数组容量 * 负载因子
             float ft = (float)newCap * loadFactor;
        /**
         * 如果新数组容量 小于 HashMap允许的最大容量(2的30次幂)
         * 并且  新计算的负载门槛 小于 HashMap允许的最大容量(2的30次幂)
         * 则新的 负载门槛为 计算后的值 的最大整型 -直接截取
         * 否则  新的负载门槛为Integer.MAX_VALUE
         */
        newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
                                (int)ft : Integer.MAX_VALUE);
        }
    //设置全局负载门槛为计算后的新的负载门槛
    threshold = newThr;
    /**
     * 根据新的数组容量创建新的哈希桶 赋值给newTab
     */
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    /**
     * 把新创建的哈希桶赋值给全局table;
     */
    table = newTab;
    /**
     * 现在开始真正的扩容
     */
    if (oldTab != null) {//如果老的哈希表不为空则执行以下语句
            //for 循环，循环老的容量次
             for (int j = 0; j < oldCap; ++j) {
                    Node<K,V> e;
                /**
                 *   //从哈希数组的第一个下标（0）开始开始递增
                 *          注释：
                 *           e = oldTab[0]  ;
                 *           e = oldTab[1]  ;  循环访问每次哈希数组下标的内容
                 *   e = oldTab[j];
```

```
                     *    如果 e != null 则开始访问数组中的内容
                     */
                    if ((e = oldTab[j]) != null) {
                                把原数组中下标为j的位置置空
                                oldTab[j] = null;
                                //e.next == null 则代表线性链表只有一个元素e
                                if (e.next == null)
                                    /**
                                     * //根据e 的哈希值和 （新数组容量 -1）相与得到 e该存放到新数组中的下标
                                     * 然后把e放入对应新数组的下标中。
                                     */
                                    newTab[e.hash & (newCap - 1)] = e;
                                else if (e instanceof TreeNode)
                                    /**
                                     * //如果当前e的类型已经改变为红黑树
                                     * 则对红黑树进行分割  ？
                                     */
                                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                                else { // preserve order
                                    /**
                                     * 进入这个else循环代表当前数组下标中存放的元素还是线性链表
                                     */
                                        Node<K,V> loHead = null, loTail = null;//定义两个指针，分别指向低位头部和低位尾部
                                        Node<K,V> hiHead = null, hiTail = null;//定义两个指针，分别指向高位头部和高位尾部
                                        Node<K,V> next;
                                    /**
                                     * do-while循环中针对数组下标为j的 线性链表进行循环查询，直到线性链表结束
                                     * 并根据每个Node的hash值与原数组容量相与得到新的值。
                                     * 与原数组容量相与后的值只会为0 或 原数组容量。
                                     * 根据得到的这两个值 进行判断
                                     * 如果 值为  0                则把他们放到loHead和loTail指向的新的线性链表当中--尾部插入
                                     * 如果 值为  原数组容量       则把他们放到hiHead和hiTail指向的新的线性链表当中--尾部插入
                                     */
```

```
                                    do {
                                        next = e.next;
                                        if ((e.hash & oldCap) ==
0) {
                                            if (loTail ==
null)
                                                loHead =
e;
                                            else
                                                loTail.next =
e;
                                            loTail = e;
                                        }
                                        else {
                                            if (hiTail ==
null)
                                                hiHead =
e;
                                            else
                                                hiTail.next =
e;
                                            hiTail = e;
                                        }
                                    } while ((e = next) != null);
                                    /**
                                     * 原线性链表结束
                                     * 如果新的loTail指向的线性链表不为空，则把它
的最后结尾值的指针指向null值
                                     *       并把loHeah与loTail指向的新的链表放
到新数组下标为j的位置上。
                                     * 如果新的hiTail指向的线性链表不为空，则把它
的最后结尾值的指针指向null值
                                     *       并把hiHeah与hiTail指向的新的链表放
到新数组下标为 （j + oldCap） 的位置上。
                                     */
                                    if (loTail != null) {
                                        loTail.next = null;
                                        newTab[j] = loHead;
                                    }
                                    if (hiTail != null) {
                                        hiTail.next = null;
                                        newTab[j + oldCap] =
hiHead;
```

```
                                        }
                                    }
                                }
                            }
                        }
                    return newTab;
                }
```

05、HashMap如何减少有效碰撞

1、扰动函数算法，促使元素位置分布均匀，减少碰撞几率；

原理是：两个不相等的对象，只要返回不相同的hashcode。这样就可以避免碰撞，减少碰撞几率；而HashMap是采用的是：拉链法来完成。

> 拉链法解决冲突的做法是：将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为*m*，则可将散列表定义为一个由*m*个头指针组成的指针数组*t[0..m-1]*。凡是散列地址为*i*的结点，均插入到以*t*为头指针的单链表中。*t*中各分量的初值均应为空指针。在拉链法中，装填因子*α*可以大于*1*，但一般均取*α≤1*。

何谓拉链法：HashCode是使用Key通过Hash函数计算出来的，由于不同的Key，通过此Hash函数可能会算的同样的HashCode，所以此时用了拉链法解决冲突，把HashCode相同的Value连成链表. 但是get的时候根据Key又去桶里找，如果是链表说明是冲突的，此时还需要检测Key是否相同

2、使用**final**对象，并采用合适的**equals**方法和**hashCode**方法；

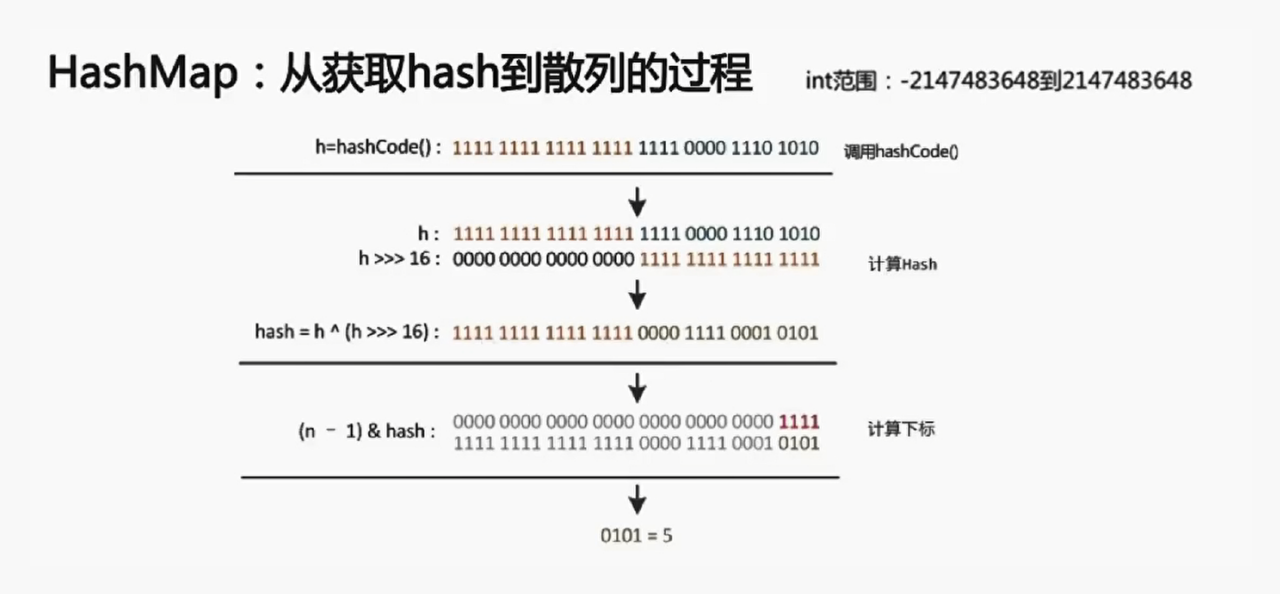
怎么让产生不同的对象返回的hashcode尽量的不一样呢？尽量去覆盖Object对象的equals和hashcode方法。而final修饰的对象使得能够缓存不同键的hashcode，这将提供获取对象的速度。而使用String,Integer这些类就特别适合做key。因为这些类都是final，也重写和覆盖了equals方法和hashcode方法。final是必要的，因为为了要计算hashcode。就要防止键值改变。如果键值在放入时和获取时返回的是不同的hashcode。就不能够在hashmap中找到对象的元素。

3、hash原理分析如下

```
/**
     * Computes key.hashCode() and spreads (XORs) higher bits of hash
     * to lower.  Because the table uses power-of-two masking, sets of
     * hashes that vary only in bits above the current mask will
     * always collide. (Among known examples are sets of Float keys
     * holding consecutive whole numbers in small tables.)  So we
     * apply a transform that spreads the impact of higher bits
     * downward. There is a tradeoff between speed, utility, and
     * quality of bit-spreading. Because many common sets of hashes
     * are already reasonably distributed (so don't benefit from
     * spreading), and because we use trees to handle large sets of
     * collisions in bins, we just XOR some shifted bits in the
     * cheapest possible way to reduce systematic lossage, as well as
     * to incorporate impact of the highest bits that would otherwise
     * never be used in index calculations because of table bounds.
     */
    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }
```

1、先将key的hashcode获取。`h = key.hashCode()`

2、然后将hashcode的高位数(`h >>> 16`是用来**取出h的高16**)移动低位。



05、深入理解JDK1.8中HashMap哈希冲突解决方案

hash函数是先拿到通过key 的hashcode，是32位的int值，然后让hashcode的高16位和低16位进行异或操作。

```
/**
     * Computes key.hashCode() and spreads (XORs) higher bits of hash
     * to lower.  Because the table uses power-of-two masking, sets of
     * hashes that vary only in bits above the current mask will
     * always collide. (Among known examples are sets of Float keys
     * holding consecutive whole numbers in small tables.)  So we
     * apply a transform that spreads the impact of higher bits
     * downward. There is a tradeoff between speed, utility, and
     * quality of bit-spreading. Because many common sets of hashes
     * are already reasonably distributed (so don't benefit from
     * spreading), and because we use trees to handle large sets of
     * collisions in bins, we just XOR some shifted bits in the
     * cheapest possible way to reduce systematic lossage, as well as
     * to incorporate impact of the highest bits that would otherwise
     * never be used in index calculations because of table bounds.
     */
```

```
    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }
```

hash函数是先拿到通过key 的hashcode，是32位的int值，然后让hashcode的高16位和低16位进行异或操作。

1. 一定要尽可能降低hash碰撞，越分散越好；
2. 算法一定要尽可能高效，因为这是高频操作, 因此采用位运算；

为什么采用**hashcode**的高**16**位和低**16**位异或能降低**hash**碰撞？**hash**函数能不能直接用**key**的**hashcode**？

我们知道：HashMap是采用数组+链表+红黑树。进行hash的作用只不过是告诉每次put元素的时候，存放到那个数组中。因为key.hashCode()函数调用的是key键值类型自带的哈希函数，返回int型散列值。int值范围为**-2147483648~2147483647**，前后加起来大概40亿的映射空间。只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。你想，如果HashMap数组的初始大小才16，用之前需要对数组的长度取模运算，得到的余数才能用来访问数组下标。

源码中模运算就是把散列值和数组长度**-1**做一个"与"操作，位运算比**%**运算要快。

最后我们来看一下Peter Lawley的一篇专栏文章《An introduction to optimising a hashing strategy》里的的一个实验：他随机选取了352个字符串，在他们散列值完全没有冲突的前提下，对它们做低位掩码，取数组下标。

Mask	String.hashCode() masked	HashMap.hash(String.hashCode()) masked
32 bits	No collisions	No collisions
16 bits	1 collision	3 collisions
15 bits	2 collisions	4 collisions
14 bits	6 collisions	6 collisions
13 bits	11 collisions	9 collisions
12 bits	17 collisions	15 collisions
11 bits	29 collisions	25 collisions
10 bits	57 collisions	50 collisions
9 bits	103 collisions	92 collisions

结果显示，当**HashMap**数组长度为**512**的时候（**2**的**9**次方），也就是用掩码取低**9**位的时候，在没有扰动函数的情况下，发生了**103**次碰撞，接近**30%**。而在使用了扰动函数之后只有**92**次碰撞。碰撞减少了将近**10%**。看来扰动函数确实还是有功效的。

另外**Java1.8**相比**1.7**做了调整，**1.7**做了四次移位和四次异或，但明显**Java 8**觉得扰动做一次就够了，做**4**次的话，多了可能边际效用也不大，所谓为了效率考虑就改成一次了。

1.7的**hash**代码：

```
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

1.8的**hash**代码：

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

06、HashMap其他面试题

HashMap为什么使用数组？

数组在有下标的情况下时间复杂度是O(1)。

```
hashcode % array. length = Index。
```

通过对象的hashcode取余数组的长度得到对象在数组中下标的位置，使查找某个对象所在位置的时间复杂度降低为O(1)。

```
在HashMap中默认的数组长度为16，如果要指定，
那么传入的参数须为2的n次方的值，
如果传入的初始长度不为2的n次方的值，
也会通过位移改变为大于传入值且是最小的2的n次方的值。
/**

 * The default initial capacity - MUST be a power of two.
默认的初始容量为16---为2的4次方

 */

static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

变更传入值的大小，如果不为**2**的**n**次幂，则变更为**2**的**n**次幂。

如果传入的容量不为**2**的**n**次幂，则变更为大于传入数值且为**2**的**n**次幂的最小值。

例如：传入11，则大于11且大于2的n次幂的最小值为16.则把初始容量变更为16.

2的3次幂为8,2的4次幂为16. 2的5次幂为32.

```
private static int roundUpToPowerOf2(int number) {

    // assert number >= 0 : "number must be non-negative";

    return number >= MAXIMUM_CAPACITY

            ? MAXIMUM_CAPACITY

            : (number > 1) ? Integer.highestOneBit((number - 1) << 1)
: 1;

}
```

数组—》默认**16**长度的数组，最大长度**2**的**30**次幂，

数组——》当有下标读取时候，时间复杂度是O(1)。增删的时候由于要移动数组中数组的位置，因此时间复杂度为O(n)。

hash碰撞（冲突），如何解决？

```
Object1.hashcode % array.length = Object2.hashcode % array. length
```

当出现Hash冲突的时候，把数组中的位置变为线性链表。

线性链表的特点：

1. 查找慢，需要一个节点一个节点的访问，时间复杂度是O(n)。
2. 增删快，时间复杂度是O(1)。

```
在JDK1.7中的线性链表插入方式为头部插入法。
在JDK1.8中的线性链表插入方式为尾部插入法。
```

链表—》头部插入法（JDK1.7），尾部插入法（JDK1.8），equles

- 由于线性链表的查找速度为0(n),因此当HashMap中数据越来越多的时候，Hash冲突的概率也越来越大，因此线性链表的长度也越来越长，性能也越来越低。为了解决这个问题，在JDK1.8之后，当线性链表的长度超过8之后，把线性链表转为红黑树。
  红黑树的特性：
- 红黑树—》JDK1.8之后-链表长度超过8之后转为红黑树，左旋，右旋；
- 红黑树是接近于平衡的搜索二叉树。
- 红黑树确保最长长度不是最低长度的两倍。
- 性能均衡，查找，插入，删除等都是0(logn)的时间复杂度。

```
五个特性：
1.节点要么是红色、要么是黑色。
2.根节点必须是黑色。
3.每个叶子结点必须是黑色。
4.如果一个节点是红色，那么他的两个儿子必须是黑色。
5.对于任意节点而言，其到叶子节点树尾端指针的每条路径都包含相同数量的黑节点。
```

JDK1.8-之后长度超过8转变为红黑树，长度8的由来？

为什么这个长度是8.

因为统计学角度，一般很少hash的碰撞值会达到7.

泊松分布----> 根据泊松分布的概率统计学角度上。
在负载因子在0.75的时候，如果链表长度大于8之后的分布概率上来说，概率可以忽略不计了。
也就是说在大多数情况下是不会到达8转红黑树的。一亿分之六的概率。

```
/**

 * The load factor used when none specified in constructor.
默认的负载因素为不太精确的 0.75f构造

 */

static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

链表转红黑树是链表长度达到阈值，这个阈值是多少？

阈值是8，红黑树转链表阈值为6

又为什么红黑树转链表的阈值是**6**，不是**8**了呢？

因为经过计算，在hash函数设计合理的情况下，发生hash碰撞8次的几率为百万分之6，概率说话。。因为8够用了，至于为什么转回来是6，因为如果hash碰撞次数在8附近徘徊，会一直发生链表和红黑树的转化，为了预防这种情况的发生。

为什么**HashMap**允许的最大容量为**2**的**30**次方？

```
/**
 * The maximum capacity, used if a higher value is implicitly
specified
 * by either of the constructors with arguments.
 * MUST be a power of two <= 1<<30.
HashMap允许的最大容量。---必须为2的幂且小于2的30次方，传入过大的值，将被替换
 */
static final int MAXIMUM_CAPACITY = 1 << 30;
```

首先：JAVA中规定了该 static final 类型的静态变量为int 类型，
至于为什么不是byte、long等类型，
原因是由于考虑到HashMap的性能问题而做的这种处理。

由于int类型的长度为4字节，也就是32个二进制位。按理说可以向左移动31位，
即2的31次幂。但是由于二进制数字中最高的一位，也就是最左边的一位是符号位，
用来表示正负之分（0为正，1为负），所以只能向左移动30位，而不能移动到最高位。

HashMap扩容机制-为什么负载因子默认为0.75f?

负载因子0.75 如果容量大大0.75则扩容为原来的两倍。扩容因此 0.75
空间利用率和时间效率在0.75的时候达到了平衡。
在统计学上0.693是最佳的选择。然后可能更想着有空间利用率，而且在。Net语言中hashmap的负载因子是0.7。

为什么Hashmap长度保证2的n次幂

Hashmap默认初始长度16，后续每次加入的值都是2的指数次幂的值。

如果传入的值不是2的指数次幂，则变成大于这个值的最接近的2的指数次幂的值。

变更传入值的大小，如果不为2的n次幂，则变更为2的n次幂。
如果传入的容量不为2的n次幂，则变更为大于传入数值且为2的n次幂的最小值。

例如：传入11，则大于11且大于2的n次幂的最小值为16.则把初始容量变更为16.

06、Final

---

final关键字的作用

相信对于final的用法，大多数人都可以随口说出三句话：

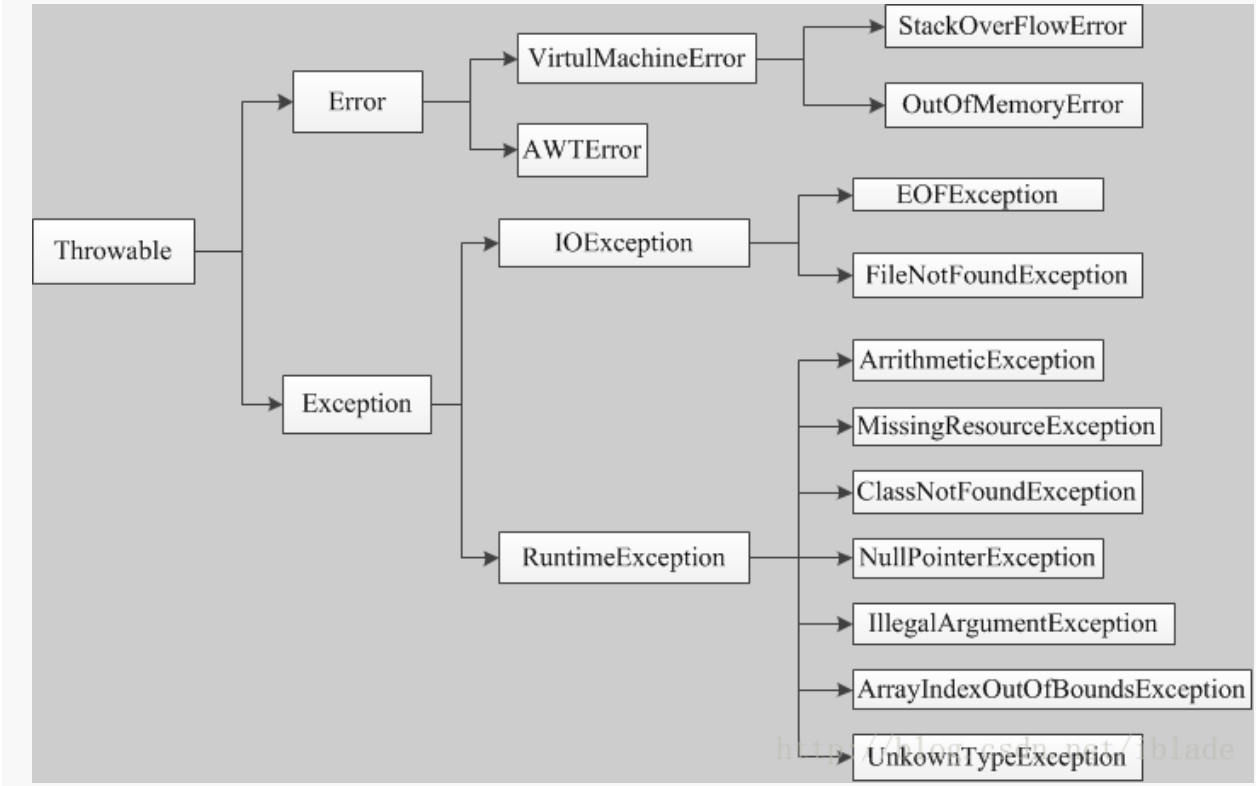
1、被final修饰的类不可以被继承

2、被final修饰的方法不可以被重写

3、被**final**修饰的变量不可以被改变

重点就是第三句。被final修饰的变量不可以被改变，什么不可以被改变呢，是变量的引用？还是变量里面的内容？还是两者都不可以被改变？写个例子看一下就知道了：

02、Java异常-Error和Exception的区别



Error：

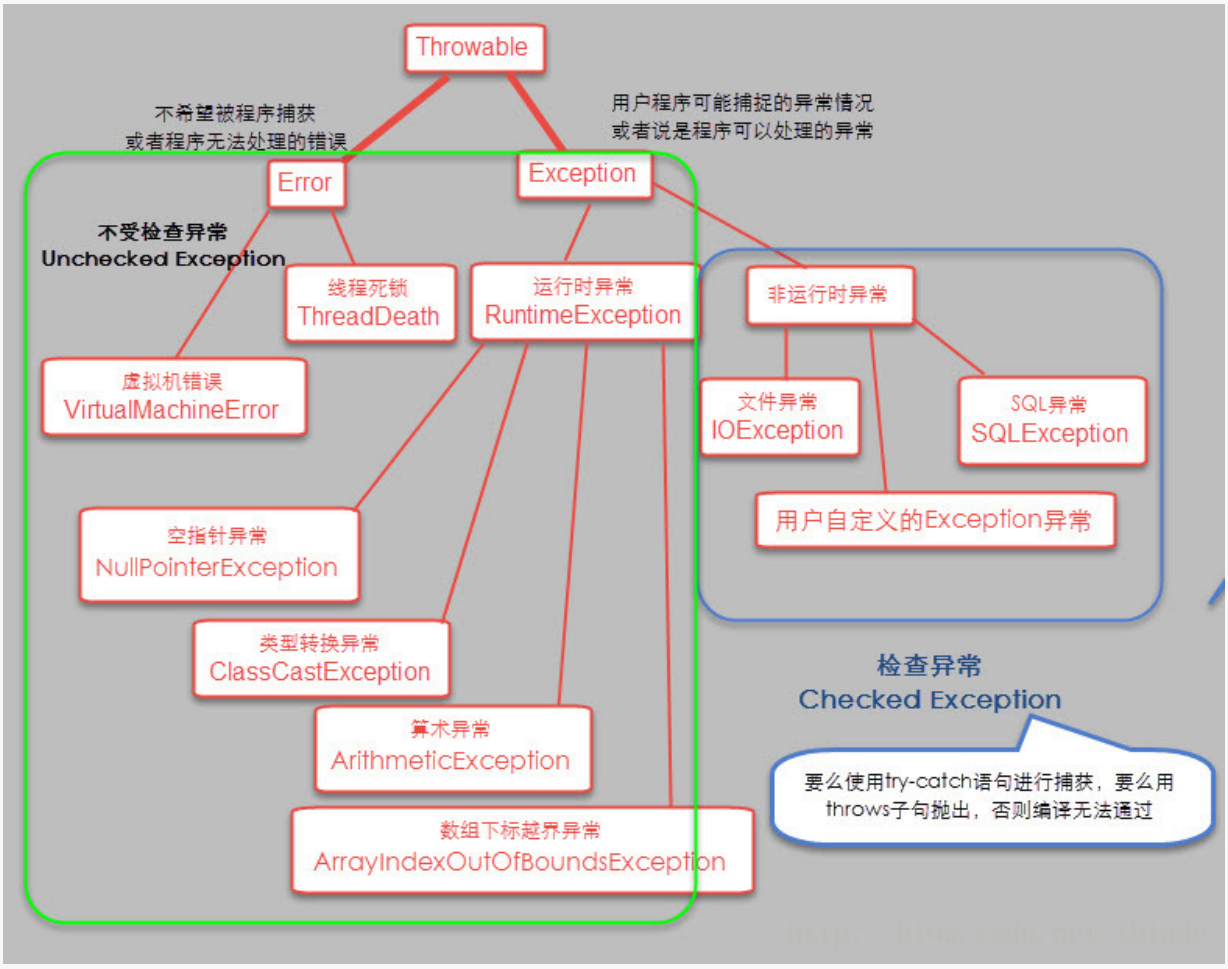
Error类对象由 Java虚拟机生成并抛出，大多数错误与代码编写者所执行的操作无关。

例如，

Java虚拟机运行错误（VirtualMachineError），当JVM不再有继续执行操作所需的内存资源时，将出现OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止；还有发生在虚拟机试图执行应用时，如类定义错误（NoClassDefFoundError）、链接错误（LinkageError）。这些错误是不可查的，因为它们在应用程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在Java中，错误通常是使用Error的子类描述。

Exception

在Exception分支中有一个重要的子类RuntimeException（运行时异常），该类型的异常自动为你所编写的程序定义ArrayIndexOutOfBoundsException（数组下标越界）、NullPointerException（空指针异常）、ArithmeticException（算术异常）、MissingResourceException（丢失资源）、ClassNotFoundException（找不到类）等异常，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生；而RuntimeException之外的异常我们统称为非运行时异常，类型上属于Exception类及其子类，从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。



- Error：程序无法处理系统异常，编译器不做检查。
- Exception：程序可以处理的异常，捕获后可能恢复。
- 总结：前者是程序无法处理的错误，后者是可以通过程序处理的异常。

```
package com.kuangstudy.kuangstudyjavajob;

import java.io.IOException;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/8/8 17:19
 */
public class ErrorAndException {

    public void throwError(){
        throw  new StackOverflowError();
    }

    // 运行时异常
    public void throwRumtimeException(){
        throw  new RuntimeException();
    }

    // 检查时异常
    public void throwCheckException(){
        try {
            throw  new IOException();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

03、常见Error以及Exception

RuntimeException

- NullPointerException：空指针异常
- ArithmeticException算术异常类
- ArrayIndexOutOfBoundsException数组下标越界异常
- NegativeArraySizeException数组负下标异常
- ClassCastException类型强制转换异常
- StringIndexOutOfBoundsException

```
# 指示索引或者为负，或者超出字符串的大小，抛出异常;
"hello".indexOf(-1);
```

- IllegalArgumentException参数异常
  抛出的异常表明向方法传递了一个不合法或不正确的参数
- NumberFormatException数字格式异常
- ClassNotFoundException找不到类异常
- dNoSuchMethodException 方法未找到异常
- FileNotFoundException 文件未找到异常

Error

- NoClassDefFoundError 找不到class定义异常类
- StackOverflowError 深度递归导致栈被耗尽而抛出的异常
- OutOfMemoryError 内存异常异常

04、Java异常处理的机制

1、抛出异常：创建异常对象，交由运行时系统处理

2、捕获异常：寻找合适的异常处理器处理异常，否则终止运行

05、Java异常处理的原则

1、具体明确：抛出的异常应能通过异常类名和Message准备说明异常的类型和产生异常的原因

2、提早抛出：应尽可能早的发现并抛出异常，便于精准定位问题

3、延迟捕获：异常的捕获和处理应尽可能延迟，让掌握更多信息的作用域来处理异常。

06、Java异常中return和finally的关系