java-JobHunter

静态代码块，构造代码块，构造函数

静态代码块：用staitc声明，jvm加载类时执行，仅执行一次 **构造代码块：**类中直接用{}定义，每一次创建对象时执行。 **执行顺序优先级：**静态块 > main() > 实例变量初始化 > 构造代码块 > 构造函数

其实实际上 实例变量初始化和构造代码块 实际上是放在构造函数的超类构造函数之后，本类构造代码之前

> https://www.jianshu.com/p/8a3d0699a923

自动装箱和自动拆箱

```
Integer i = 10;  //装箱 基本数据类型 -> 包装器类型
int n = i;   //拆箱 包装器类型 -> 基本数据类型
```

自动装箱：调用包装器的valueOf（**对于有些类的实现就会走缓存**）

手动装箱：new Integer(value)就是直接赋值，这样是避免了自动装箱走缓存的影响，因为缓存可能会被反射修改

自动拆箱：调用包装器的 xxxValue（xxx代表对应的基本数据类型）

基本数据类型对应的包装器类型

int（4字节）	**Integer**
byte（1字节）	Byte
short（2字节）	Short
long（8字节）	Long
float（4字节）	Float
double（8字节）	Double
char（2字节）	Character
boolean（未定）	Boolean

Integer、Short、Byte、Character、Long这几个类的valueOf方法的实现是类似的（缓存）

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

Double、Float的valueOf方法的实现是类似的（new 新对象）

```
public static Double valueOf(double d) {
    return new Double(d);
}
```

为何? 因为在某个范围内的整型数值的个数是有限的，而浮点数却不是

Boolean的valueOf实现比较特殊

```
public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

TRUE和FALSE是随Boolean类加载初始化的两个类静态变量

```
public static final Boolean TRUE = new Boolean(true);
public static final Boolean FALSE = new Boolean(false);
```

> https://www.cnblogs.com/dolphin0520/p/3780005.html
>
> https://mp.weixin.qq.com/s/PWygd0Ro-B3yslm4de9KKA

hashCode、equals 和 ==

hashCode

1. 对象相等则hashCode一定相等 2. hashCode相等对象未必相等

Object类的hashCode方法，调用c实现的，默认是根据**对象地址**计算hashCode

```
public native int hashCode();
```

常用的类均已override hashCode方法

一般可以不需了解hashcode用法，但只要和哈希运算有关的地方HashMap，HashSet等**集合类时要注意下hashcode**

HashMap原理：是以hashCode作为key插入的，一般hashCode % 8得到所在的索引，如果所在索引处有元素了，则使用一个链表，把多的元素不断链接到该位置，所以hashCode的作用就是找到索引的位置，然后再用equals去比较元素是不是相等，**先用hashCode找到桶（bucket），然后再用equals在里面找东西**。

equals

是对象具体内容相等性比较，但Object的equals实现是 **==**

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

常用类均已override equals方法，实现了对象内容相等的比较

==

对于基本类型，比较存储的实际值

对于对象引用，比较对象引用的内存地址

总结：从语法角度，也就是从强制性的角度来说，hashCode和equals是两个独立的，互不隶属，互不依赖的方法，equals成立与hashCode相等这两个命题之间，谁也不是谁的充分条件或者必要条件。 但是，为了让程序正常运行，应该如Effective Java中所言，**重写equals的时候，一定要（正确）重写hashCode**

重写equals不重写hashCode

使用HashMap，HashSet时，逻辑上相等的两个元素被分到不同的bucket里，并且调用contains方法时即使已经存在逻辑上相等的元素也会返回false

重写equals需要满足5个条件

自反性：对于任何非空引用值 x，x.equals(x) 都应返回 true。

对称性：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

传递性：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true， 并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。

一致性：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false， 前提是对象上 equals 比较中所用的信息没有被修改。

非空性：对于任何非空引用值 x，x.equals(null) 都应返回 false。

即为了保证逻辑上的程序正确运行

> https://juejin.im/post/5a4379d4f265da432003874c

POJO类中布尔值变量成员变量（Boolean和boolean区别）

POJO不要使用 **isXXX** 这种命名方式，否则部分框架解析会引起序列化错误

boolean的自动生成的getter方法是 isXXX，而Boolean自动生成的getter和其他类型变量一致为getXXX

boolean 默认值 false Boolean默认值null（**对于除boolean以外的POJO成员变量，最好都用包装类型，即用NPE来避免了默认值导致的逻辑错误，但没有报错难以发现**）

```
public class IsSuccessTest {
    class Model1 {
        private boolean success;
        public boolean isSuccess() {
            return success;
        }
        public void setSuccess(boolean success) {
            this.success = success;
        }
    }
    class Model2 {
        private boolean isSuccess;
        public boolean isSuccess() {
            return isSuccess;
        }
        public void setSuccess(boolean success) {
            isSuccess = success;
        }
    }
    class Model3 {
        private Boolean succsee;
        public Boolean getSuccsee() {
            return succsee;
        }
        public void setSuccsee(Boolean succsee) {
            this.succsee = succsee;
        }
    }
    class Model4 {
        private Boolean isSuccess;
        public Boolean getSuccess() {
            return isSuccess;
        }
        public void setSuccess(Boolean success) {
            isSuccess = success;
        }
    }
}
```

fastJson、jackson：利用反射遍历POJO类中所有getter方法（包括boolean类型的 **isXXX()** 方法），会直接根据JavaBeans规则把方法 getXXX 和 isXXX 后的 XXX 当作属性名，把此方法的返回值当作属性值

此处还会有一个问题，就是如果你自己写了个 getXXX 形式的方法，这两个框架也会把 XXX 作为属性，其返回值作为属性值

Gson：利用反射遍历POJO类中的所有属性，并把其值序列化成json

> https://mp.weixin.qq.com/s/LTiN6800FbIPmG2UdWwqqg

静态导包

Java 5 加入的

```
import static com.xxx.xx;
```

静态导入包后，可以直接用方法名调用其静态方法

但滥用可能会导致程序难以维护，可读性变差

Java计算时间为什么从1970 年 1 月 1 日开始?

Java, JavaScript, Python使用 Unix epoch (Midnight 1 January 1970)，时间戳的起始时间点即为距离此时间的毫秒数

为什么Unix epoc 是 Midnight 1 January 1970 ?

最初计算机操作系统是32 位，而时间也是用 32 位表示。 Integer在 JAVA 内用 32 位表 示，因此 32 位能表示的最大值是2147483647（ 范围是-2^31~2^31 - 1）。 另外1 年 365 天的总秒数是 3600 * 24 * 365 = 31536000， 2147483647/31536000 = 68.1。也就是说32 位能表示的最长时间是 68 年，而实际上到 2038年 01 月 19 日 03 时 14 分 07 秒，便会到达最大时间，过了这个时间点，所 有 32 位操作系统时间便会变 为 10000000 00000000 00000000 00000000也就是1901年 12月 13 日 20时 45 分 52 秒，这样便会出现时间回归的现象，很多软件便会运 行异常了。

至于时间回归的现象相信随着64 为操作系统 的产生逐渐得到解决，因为这个时间已经是千亿年以后了。

```
Date date = new Date(0);
System.out.println(date);
```

```
输出：Thu Jan 01 08:00:00 CST 1970
```

为什么是8点而非0点?

系统时间和本地时间问题，系统时间依然是0点，只不过电脑时区设置为东8区，故打印的结果是8点。

数组初始化尾随逗号会被忽略

```
int[] b = {1, 2, 3, 4, 5, 6,}; //A trailing comma causes no compiler error
```

> A trailing comma may appear after the last expression in an array initializer and is ignored.

Boolean的大小

《Java虚拟机规范》一书中的描述：虽然定义了boolean这种数据类型，但是只对它提供了非常有限的支持。在Java虚拟机中没有任何供boolean值专用的字节码指令，Java语言表达式所操作的boolean值，**在编译之后都使用Java虚拟机中的int数据类型来代替**，而boolean数组将会被编码成Java虚拟机的byte数组，每个元素boolean元素占8位”。这样我们可以得出boolean类型占了**单独使用是4个字节，在数组中又是1个字节**。

子类继承父类的时候，到底继承了什么?

- **1、子类继承父类所有的属性（除了private）**
- 2、子类继承父类（除private）所有的方法，（子类方法如果不调用 super.所复写方法名称，那么对应父类方法将不会执行）
  - **final方法不可以被继承**
  - **static方法不可以被继承**，。但是如果权限允许子类还是可以用。
  - **子类是不继承父类的static变量和方法的**。随着类的加载而加载，继承毛线。因为这是属于类本身的。但是子类是可以访问的。
  - 子类和父类中同名的static变量和方法都是相互独立的，并不存在任何的重写的关系。
- **3、子类可以通过super，表示父类的引用，调用父类的属性或者方法。**（构造函数`隐式static`和代码块是无法被继承）

String、StringBuffer和StringBuilder区别

1. 数据可变和不可变

1. `String`底层使用一个不可变的字符数组`private final char value[];`所以它内容不可变。
2. `StringBuffer`和`StringBuilder`都继承了`AbstractStringBuilder`底层使用的是可变字符数组：`char[] value;`

2. 线程安全

`StringBuilder`是线程不安全的，效率较高；而`StringBuffer`是线程安全的，效率较低。

通过他们的`append()`方法来看，`StringBuffer`是有同步锁，而`StringBuilder`没有：

```
@Override
public synchronized StringBuffer append(Object obj) {
    toStringCache = null;
    super.append(String.valueOf(obj));
    return this;
}
@Override
public StringBuilder append(String str) {
    super.append(str);
    return this;
}
```

3. 相同点

`StringBuilder`与`StringBuffer`有公共父类`AbstractStringBuilder`。

最后，**操作可变字符串速度**：`StringBuilder > StringBuffer > String`，这个答案就显得不足为奇了。

移位运算和异或运算（mod 2加法）

题目一：最有效率的计算2*8

```
2<<3
```

题目二：不借助第三方变量实现两个数的交换

```
//方法一：这确实是一种方法，但是不推荐使用，因为如果两个数的值过大，相加后可能超出int范围
a = a + b;
b = a - b;
a = a - b;
//方法二：其实原理和上面是一样的，但加法不进位，减法不移位，这两种独特的加减的运算是一样的，合为一个异或^
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

容器源码分析

ArrayList

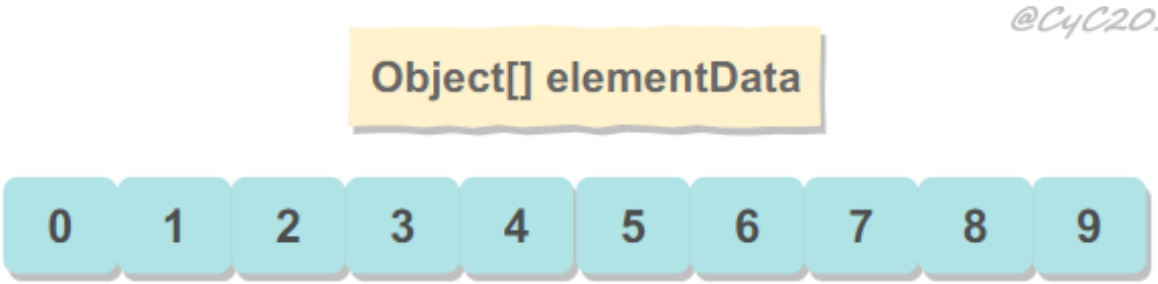
1. 概览

实现了 RandomAccess 接口，因此支持随机访问。这是理所当然的，因为 ArrayList 是基于数组实现的。

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

数组的默认大小为 10。，最大大小为Integer.MAX_VALUE - 8（因为有些VM会在数组里保存一些头部信息）

```
/**
 * 默认初始化容量
 */
private static final int DEFAULT_CAPACITY = 10;
/**
 * 如果自定义容量为0，则会默认用它来初始化ArrayList。或者用于空数组替换。
 */
private static final Object[] EMPTY_ELEMENTDATA = {};
/**
 * 如果没有自定义容量，则会使用它来初始化ArrayList。或者用于空数组比对。
 */
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
/**
 * 这就是ArrayList底层用到的数组
 * 非私有，以简化嵌套类访问
 * transient 在已经实现序列化的类中，不允许某变量序列化
 */
transient Object[] elementData;
/**
 * 实际ArrayList集合大小
 */
private int size;
/**
 * 可分配的最大容量
 */
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```



构造函数

注意这里，这里如果指定初始化容量的话，就**直接分配一个指定大小的对象数组，并不是懒加载！！！**

```
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}
```

2. 扩容

添加元素时使用 ensureCapacityInternal() 方法来保证容量足够，如果不够时，需要使用 grow() 方法进行扩容，新容量的大小为 `oldCapacity + (oldCapacity >> 1)`，**默认是旧容量的 1.5 倍**，如果1.5倍不够会使用真实需要的容量（前提：真实容量 < Integer.MAX_VALUE）

扩容操作需要调用 **`Arrays.copyOf()`** 把原数组整个复制到新数组中，这个操作代价很高，因此最好在创建 ArrayList 对象时就指定大概的容量大小，减少扩容操作的次数。

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}

private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    ensureExplicitCapacity(minCapacity);
}

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}

private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1); //算术右移相当于除以2
    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);
}
```

3. 删除元素

需要调用 `System.arraycopy()` 将 index+1 后面的元素都复制到 index 位置上，该操作的时间复杂度为 O(N)，可以看出 **ArrayList 删除元素的代价是非常高的**。

```
public E remove(int index) {
    rangeCheck(index);
    modCount++;
    E oldValue = elementData(index);
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index, numMoved);
    elementData[--size] = null; // clear to let GC do its work
    return oldValue;
}
```

4. Fail-Fast

modCount 用来记录 ArrayList 结构发生变化的次数。结构发生变化是指添加或者删除至少一个元素的所有操作，或者是调整内部数组的大小，**仅仅只是设置元素的值不算结构发生变化**。

在进行序列化或者迭代等操作时，需要比较操作前后 modCount 是否改变，如果改变了需要抛出 **ConcurrentModificationException**。禁止多进程同时修改同一容器内容。

```
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();

    // Write out size as capacity for behavioural compatibility with clone()
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
    }

    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}
```

5. 序列化

ArrayList 基于数组实现，并且具有**动态扩容**特性，因此保存元素的数组不一定都会被使用，那么就没必要全部进行序列化。

保存元素的数组 elementData 使用 transient 修饰，该关键字声明数组默认不会被序列化。

```
transient Object[] elementData; // non-private to simplify nested class access
```

ArrayList 实现了 writeObject() 和 readObject() 来控制只序列化数组中有元素填充那部分内容。

```
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    elementData = EMPTY_ELEMENTDATA;

    // Read in size, and any hidden stuff
    s.defaultReadObject();

    // Read in capacity
    s.readInt(); // ignored

    if (size > 0) {
        // be like clone(), allocate array based upon size not capacity
        ensureCapacityInternal(size);

        Object[] a = elementData;
        // Read in all elements in the proper order.
        for (int i=0; i<size; i++) {
            a[i] = s.readObject();
        }
    }
}
```

```
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();

    // Write out size as capacity for behavioural compatibility with clone()
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
    }

    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
```

```
}
```

序列化时需要使用 ObjectOutputStream 的 writeObject() 将对象转换为字节流并输出。而 writeObject() 方法在传入的对象存在 writeObject() 的时候会去反射调用该对象的 writeObject() 来实现序列化。反序列化使用的是 ObjectInputStream 的 readObject() 方法，原理类似，这里相当于使用 Serializable接口，实现了类似Externalizable 的功能。

这里要遍历序列化elemntData数组的原因是，只序列化其存储的元素（通常其会预留存储空间）

```
ArrayList list = new ArrayList();
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file));
oos.writeObject(list);
```

trimToSize()方法：

> 用来最小化实例存储，将容器大小调整为当前元素所占用的容量大小。

```
/**
 * 这个方法用来最小化实例存储。
 */
public void trimToSize() {
    modCount++;
    if (size < elementData.length) {
        elementData = (size == 0)
          ? EMPTY_ELEMENTDATA
          : Arrays.copyOf(elementData, size);
    }
}
```

clone()方法

> 用来克隆出一个新数组。

```
public Object clone() {
    try {
        ArrayList<?> v = (ArrayList<?>) super.clone();
        v.elementData = Arrays.copyOf(elementData, size);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
    }
}
```

通过调用`Object`的`clone()`方法来得到一个新的`ArrayList`对象，然后将`elementData`复制给该对象并返回。

add(E e)方法

> 在数组末尾添加元素

```
/**
 * 在数组末尾添加元素
 */
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

看到它首先调用了`ensureCapacityInternal()`方法.注意参数是**size+1**,这是个面试考点。

```
private void ensureCapacityInternal(int minCapacity) {
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}
```

这个方法里又嵌套调用了两个方法:**计算容量+确保容量**

计算容量：如果elementData是空，则返回默认容量10和size+1的最大值，否则返回size+1

```
private static int calculateCapacity(Object[] elementData, int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
}
```

计算完容量后，进行**确保容量可用**：(modCount不用理它，它用来计算修改次数)

```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

增加容量：默认**1.5倍**扩容。

```
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);
}
```

size +1的问题

size+1代表的含义是：

1. 如果集合添加元素成功后，集合中的实际元素个数。
2. 为了确保扩容不会出现错误。

假如不加一处理，如果默认size是0，则0+0>>1还是0。 如果size是1，则1+1>>1还是1。有人问:不是默认容量大小是10吗?事实上，jdk1.8版本以后，ArrayList的扩容放在add()方法中。之前放在构造方法中。我用的是1.8版本，所以默认`ArrayList arrayList = new ArrayList();`后，size应该是0.所以,size+1对扩容来讲很必要.

add(int index, E element)方法

```
public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

`System.arrayCopy`方法:

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos,
int length)
```

- 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

在`remove(int index)`方法中也使用了，`System.arrayCopy`，将index + 1开始的所有元素向前移动一格

Vector

1. 同步

它的实现与 ArrayList 类似，但是使用了 **synchronized** 进行同步。

```
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}

public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    return elementData(index);
}
```

2. 与ArrayList的比较

`Collections.synchronizedList(List<T> list)`方法ArrayList转换成线程安全的，但这种转换方式依然是通过synchronized修饰方法实现的，很显然这不是一种高效的方式

- Vector 是同步的，因此开销就比 ArrayList 要大，访问速度更慢。最好使用 ArrayList 而不是 Vector，因为同步操作完全可以由程序员自己来控制；
- Vector 每次扩容请求其大小的 **2 倍空间**，而 ArrayList 是 **1.5 倍**。

替代方案

可以使用 `Collections.synchronizedList();` 得到一个线程安全的 ArrayList

```
List<String> list = new ArrayList<>();
List<String> synList = Collections.synchronizedList(list);
```

也可以使用 concurrent 并发包下的 CopyOnWriteArrayList 类。

```
List<String> list = new CopyOnWriteArrayList<>();
```

CopyOnWriteArrayList

读写分离(ReentrantLock实现写锁)

写操作在一个**复制** `Arrays.copyOf` 的数组上进行，读操作还是在**原始数组**中进行，读写分离，互不影响。

写操作需要加锁，防止并发写入时导致写入数据丢失。

写操作结束之后需要把原始数组指向新的复制数组。

```
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}

final void setArray(Object[] a) {
    array = a;
}
```

```
@SuppressWarnings("unchecked")
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

适用场景

CopyOnWriteArrayList 在写操作的同时允许读操作，大大提高了**读操作**的性能，因此很适合**读多写少**的应用场景。

但是 CopyOnWriteArrayList 有其缺陷：

- 内存占用：在写操作时需要复制一个新的数组，使得内存占用为原来的两倍左右；
- 数据不一致：读操作不能读取实时性的数据，因为部分写操作的数据还未同步到读数组中。

所以 CopyOnWriteArrayList **不适合内存敏感以及对实时性要求很高的场景**。

LinkedList

概览

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable{
}
```

基于双向链表实现，使用 Node 存储链表节点信息。

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
}
```

每个链表存储了 first 和 last 指针：

```
transient Node<E> first;
transient Node<E> last;
```


```
/**
 * 将集合添加到链尾
 */
public boolean addAll(Collection<? extends E> c) {
    return addAll(size, c);
}
public boolean addAll(int index, Collection<? extends E> c) {
    checkPositionIndex(index);

    // 拿到目标集合数组
    Object[] a = c.toArray();
    //新增元素的数量
    int numNew = a.length;
    //如果新增元素数量为0，则不增加，并返回false
    if (numNew == 0)
        return false;

    //定义index节点的前置节点，后置节点
    Node<E> pred, succ;
    // 判断是否是链表尾部，如果是：在链表尾部追加数据
    //尾部的后置节点一定是null，前置节点是队尾
    if (index == size) {
        succ = null;
        pred = last;
    } else {
        // 如果不在链表末端(而在中间部位)
```

```
            // 取出index节点，并作为后继节点
            succ = node(index);
            // index节点的前节点 作为前驱节点
            pred = succ.prev;
        }
        // 链表批量增加，是靠for循环遍历原数组，依次执行插入节点操作
        for (Object o : a) {
            @SuppressWarnings("unchecked")
            // 类型转换
            E e = (E) o;
            // 前置节点为pred，后置节点为null，当前节点值为e的节点newNode
            Node<E> newNode = new Node<>(pred, e, null);
            // 如果前置节点为空， 则newNode为头节点，否则为pred的next节点
            if (pred == null)
                first = newNode;
            else
                pred.next = newNode;
            pred = newNode;
        }
        // 循环结束后，如果后置节点是null，说明此时是在队尾追加的
        if (succ == null) {
            // 设置尾节点
            last = pred;
        } else {
        //否则是在队中插入的节点 ，更新前置节点 后置节点
            pred.next = succ;
            succ.prev = pred;
        }
        // 修改数量size
        size += numNew;
        //修改modCount
        modCount++;
        return true;
    }
    /**
      * 取出index节点
      */
    Node<E> node(int index) {
        // assert isElementIndex(index);

        // 如果index 小于 size/2,则从头部开始找
        if (index < (size >> 1)) {
            // 把头节点赋值给x
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                // x=x的下一个节点
                x = x.next;
            return x;
        } else {
```

```
        // 如果index 大与等于 size/2，则从后面开始找
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}

// 检测index位置是否合法
private void checkPositionIndex(int index) {
    if (!isPositionIndex(index))
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
// 检测index位置是否合法
private boolean isPositionIndex(int index) {
    return index >= 0 && index <= size;
}
```

注意`node(index)`方法:寻找处于index的节点，有一个小优化，**结点在前半段则从头开始遍历，在后半段则从尾开始遍历**，这样就保证了只需要遍历最多一半结点就可以找到指定索引的结点。

addFirst(E e)方法

```
public void addFirst(E e) {
    linkFirst(e);
}
//将e链接成列表的第一个元素
private void linkFirst(E e) {

    final Node<E> f = first;
    // 前驱为空，值为e，后继为f
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    //若f为空，则表明列表中还没有元素，last也应该指向newNode
    if (f == null)
        last = newNode;
    else
    //否则，前first的前驱指向newNode
        f.prev = newNode;
    size++;
    modCount++;
}
```

add(int index, E element)方法

```
public void add(int index, E element) {
    checkPositionIndex(index);
    if (index == size)
```

```
        linkLast(element);
    else
        linkBefore(element, node(index));
}
/**
 * 在succ节点前增加元素e(succ不能为空)
 */
void linkBefore(E e, Node<E> succ) {
    // assert succ != null;
    // 拿到succ的前驱
    final Node<E> pred = succ.prev;
    // 新new节点：前驱为pred，值为e，后继为succ
    final Node<E> newNode = new Node<>(pred, e, succ);
    // 将succ的前驱指向当前节点
    succ.prev = newNode;
    // pred为空，说明此时succ为首节点
    if (pred == null)
        // 指向当前节点
        first = newNode;
    else
        // 否则，将succ之前的前驱的后继指向当前节点
        pred.next = newNode;
    size++;
    modCount++;
}
```

与ArrayList的比较

- 优点：
  1. 不需要扩容和预留空间,空间效率高
  2. 增删效率高
- 缺点：
  1. 随机访问时间效率低
  2. 改查效率低

HashMap

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable {
        /**
     * The default initial capacity - MUST be a power of two.
     */
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16，默认容量
    static final int MAXIMUM_CAPACITY = 1 << 30;        //最大容量 2^30
    static final float DEFAULT_LOAD_FACTOR = 0.75f;//负载因子=size/capacity
    static final int TREEIFY_THRESHOLD = 8;     //链表->红黑树阈值
    static final int UNTREEIFY_THRESHOLD = 6;   //红黑树->链表阈值
    static final int MIN_TREEIFY_CAPACITY = 64;

}
```

参数	含义
capacity	table 的容量大小，默认为 16。需要注意的是 capacity 必须保证为 2 的 n 次方。
size	键值对数量。
threshold	size 的临界值，当 size 大于等于 threshold 就必须进行扩容操作。
loadFactor	负载因子，table 能够使用的比例，threshold = capacity * loadFactor。

内部包含了一个`静态内部类Node[] 类型`的数组 table。**在1.7 中是一个Entry<K, V>[] table**

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

该数组长度始终**为2的n次幂**，通过以下函数实现

用位运算找到大于或等于 cap 的最小的（！！！）2的整数次幂的数。比如10，则返回16

```
static final int tableSizeFor(int cap) {
    int n = cap - 1;// 如果不做该操作， 则如传入的 cap 是 2 的整数幂， 则返回值是预想的 2
倍
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

因为int最大就`2^32`所以移动1、2、4、8、16位并取位或,会将**最高位的1后面的位全变为1。**

其原理是将传入参数 `(cap)` 的**低二进制全部变为1**，最后加1即可获得对应的大于 `cap` 的 2 的次幂作为数组长度。

为什么要使用2的次幂作为数组的容量呢?

`HashMap` 的 `hash` 函数及数组下标的计算， 键(key)所计算出来的哈希码有可能是大于数组的容量的，那怎么办?

可以通过简单的求余运算来获得，但此方法效率太低。**MOD运算是非常消耗CPU时间的**

HashMap中通过以下的方法**保证 `hash` 的值计算后都小于数组的容量**。

```
(n - 1) & hash  //容量是n=table.length() , 等价于对其取余
```

由于n是2的次幂，因此，**n-1类似于一个低位掩码**。通过**与**操作，高位的hash值全部归零，保证低位才有效 从而保证获得的值都小于n

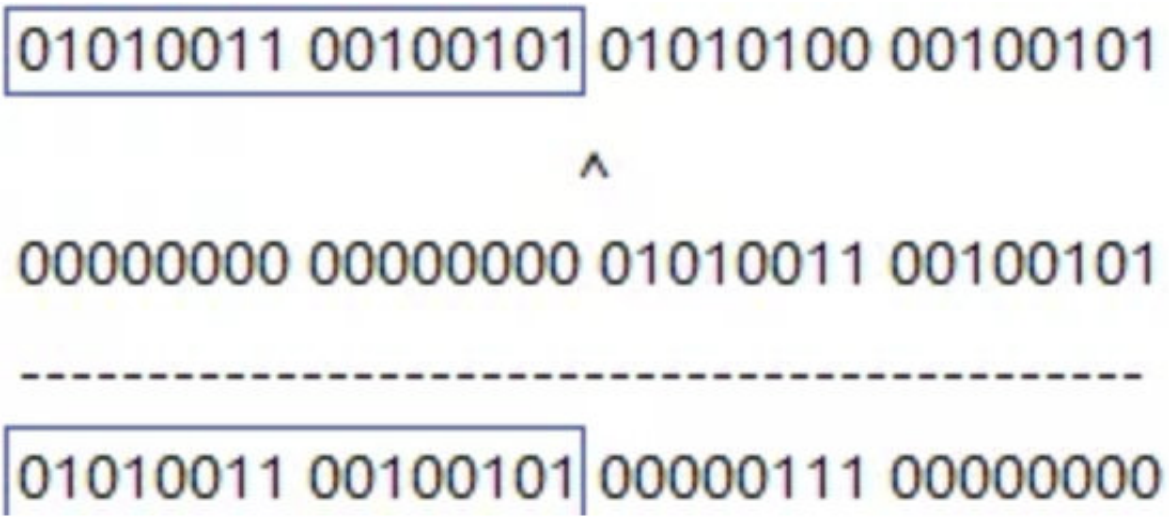
同时，在下一次 `resize()` 操作时， 重新计算每个 `Node` 的数组下标将会因此变得很简单，具体的后文讲解。以默认的初始值16为例

```
   01010011 00100101 01010100 00100101
&   00000000 00000000 00000000 00001111
----------------------------------------
    00000000 00000000 00000000 00000101    //高位全部归零，只保留末四位
    // 保证了计算出的值小于数组的长度 n
```

但是，使用了该功能之后，由于只取了低位，因此 hash 碰撞会也会相应的变得很严重。这时候就需要使用 **「扰动函数」**

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

该函数通过将**哈希码的高16位的右移后**与原哈希码进行**异或**而得到，以上面的例子为例



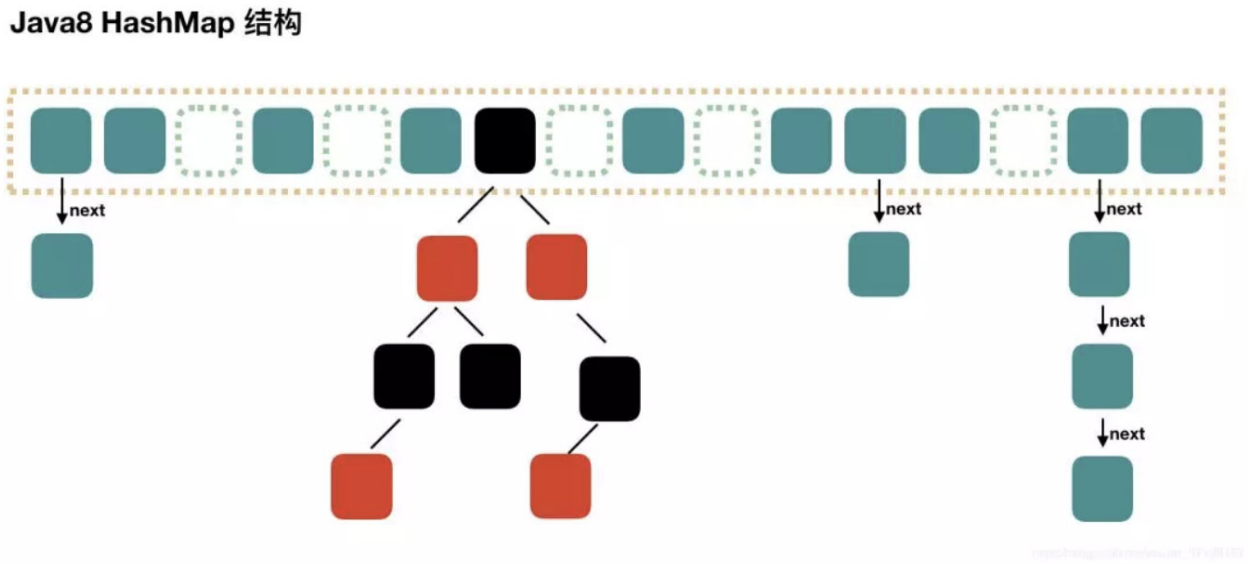
此方法保证了**高16位不变， 低16位根据异或后的结果改变**。计算后的数组下标将会**从原先的5变为0**。

流程是：先扰动函数得到**扰动后hash**，再(n - 1) & hash，得到最后的数组下标

存储结构

Node 存储着键值对。它包含了**四个字段**，从 next 字段我们可以看出 Node 是一个链表。即数组中的每个位置被当成一个桶，一个桶存放一个链表。HashMap 使用**拉链法**和**红黑树**来解决冲突，同一个链表中存放哈希值相同的 Node。

HashMap的链表插入是头插法




```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    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;
    }
}
```

HashMap初始化

```
 public HashMap();
 public HashMap(int initialCapacity);
 public HashMap(Map<? extends K, ? extends V> m);
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
//通过该函数进行了容量和负载因子的初始化，如果是调用的其他的构造函数， 则相应的负载因子和容量会使用默认值（默认负载因子=0.75， 默认容量=16）。在此时， 还没有进行存储容器 table 的初始化，该初始化要延迟到第一次使用时进行。
```

HashMap的所有构造函数，**最多只是设置了loadfactor和threshold的值，并未分配存储空间，懒加载！！！初始化扩容，如果指定了初始容量，会放在threshold中，并且在put时调用resize并且加载threshold值作为容量**

`new HashMap();`完成后，**如果没有`put`操作，是不会分配存储空间的。**

HashMap动态扩容（resize()方法）

作为数组， 其在初始化时就需要指定长度。在实际使用过程中， 我们存储的数量可能会大于该长度，因此 HashMap 中定义了一个阈值参数(threshold)， 在存储的容量达到指定的阈值时， 需要进行扩容。

> 我个人认为初始化也是动态扩容的一种， 只不过其扩容是容量从 0 扩展到构造函数中的数值（默认16）。 而且不需要进行元素的重hash.

扩容发生条件（再哈希）

size > threshold 时即动态扩容，也称为再哈希

```
threshold = loadFactor * capacity;
```

比如 HashMap 中默认的 loadFactor=0.75, capacity=16, 则

```
threshold = loadFactor * capacity = 0.75 * 16 = 12
```

总结起来，一共有三种**扩容方式（都是在第一次put时才进行扩容，分配存储空间！！！！！！！）**：

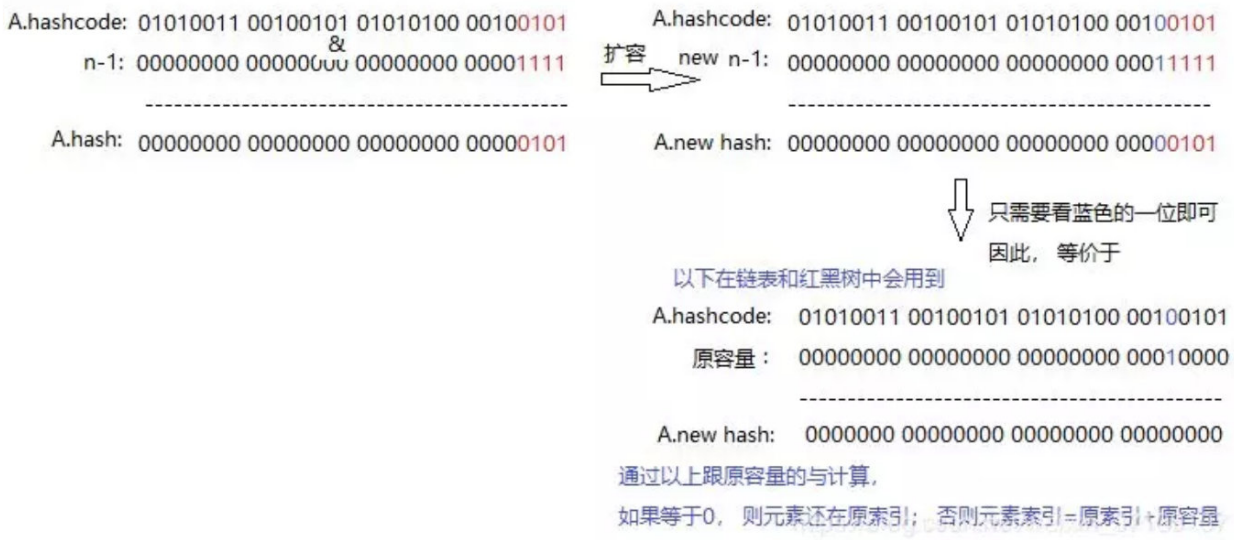
1. 使用**默认构造方法**初始化HashMap。从前文可以知道HashMap在一开始初始化的时候会返回一个空的table，并且thershold为0。因此第一次扩容的容量为默认值`DEFAULT_INITIAL_CAPACITY`也就是16。同时`threshold = DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR = 12`。
2. **指定初始容量**的构造方法初始化`HashMap`。那么从下面源码可以看到初始容量会等于`threshold`，接着`threshold = 当前的容量（threshold） * DEFAULT_LOAD_FACTOR`。
3. HashMap**不是第一次扩容**。如果`HashMap`已经扩容过的话，那么每次table的容量以及`threshold`量为**原有的两倍。**

再谈容量为2的整数次幂和数组索引计算

前面说过了数组的容量为 2 的整次幂， 同时， 数组的下标通过下面的代码进行计算

```
index = (table.length - 1) & hash;
```

由于数组扩容之后， 容量是现在的 2 倍， 扩容之后 n-1 的有效位会比原来多一位， 而多的这一位与原容量二进制在同一个位置。 示例（**注意：这里的A.hashcode是没扰动过的，只是为了说明原理**）



扩容步骤

- 先判断是初始化还是扩容， 两者在计算newCap和newThr时会不一样
- 计算扩容后的容量，临界值。
- 将hashMap的临界值修改为扩容后的临界值
- 根据扩容后的容量新建数组，然后将hashMap的table的引用指向新数组。

- 将旧数组的元素复制到table中。在该过程中， 涉及到几种情况， 需要分开进行处理（只存有一个元素， 一般链表， 红黑树）

与 HashTable 的比较

- HashTable 使用 synchronized 来进行同步。
- HashMap 可以插入键为 **null** 的 Node（**但无法确认其hashCode，默认插入第0个桶**）。
- HashMap 的迭代器是 fail-fast 迭代器。
- HashMap **不能保证随着时间的推移 Map 中的元素次序是不变的**。

注意事项

虽然 `HashMap` 设计的非常优秀， 但是应该尽可能少的避免 `resize()`, 该过程会很耗费时间。

同时， 由于 `hashmap` **不能自动的缩小容量** 因此，如果你的 `hashmap` 容量很大，但执行了很多 `remove`操作时，**容量并不会减少**。如果你觉得需要减少容量，请重新创建一个 hashmap。

> 1. JDK1.7是基于数组+单链表实现（为什么不用双链表）

首先，用链表是为了解决hash冲突。

单链表能实现为什么要用双链表呢?(双链表需要更大的存储空间)

> 2. 为什么要用红黑树，而不用平衡二叉树?

插入效率比平衡二叉树高，查询效率比普通二叉树高。所以选择性能相对折中的红黑树。

> 3. 既然红黑树那么好，为啥hashmap不直接采用红黑树，而是当大于8个的时候才转换红黑树?

因为红黑树需要进行左旋，右旋操作， 而单链表不需要。

至于为什么选数字8，是大佬折中衡量的结果-.-，就像loadFactor默认值0.75一样。

ConcurrentHashMap（见Thinking in Java 笔记）

LinkedHashMap

内部维护了一个双向链表，用来维护插入顺序或者 LRU 顺序。

```
/**
 * The head (eldest) of the doubly linked list.
 */
transient LinkedHashMap.Entry<K,V> head;

/**
 * The tail (youngest) of the doubly linked list.
 */
transient LinkedHashMap.Entry<K,V> tail;
```

accessOrder 决定了顺序，**默认为 false，此时维护的是插入顺序。**

```
final boolean accessOrder;
```

LinkedHashMap 最重要的是以下用于维护顺序的函数，它们会**在 put、get 等方法中调用**。

```
void afterNodeAccess(Node<K,V> p) { }
void afterNodeInsertion(boolean evict) { }
```

afterNodeAccess()（get方法调用）

当一个节点被访问时，如果 accessOrder 为 true，则会将该节点移到链表尾部。也就是说指定为 LRU 顺序之后，**在每次访问一个节点时，会将这个节点移到链表尾部，保证链表尾部是最近访问的节点，那么链表首部就是最近最久未使用的节点。**

```
void afterNodeAccess(Node<K,V> e) { // move node to last
    LinkedHashMap.Entry<K,V> last;
    if (accessOrder && (last = tail) != e) {
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
        p.after = null;
        if (b == null)
            head = a;
        else
            b.after = a;
        if (a != null)
            a.before = b;
        else
            last = b;
        if (last == null)
            head = p;
        else {
            p.before = last;
            last.after = p;
        }
        tail = p;
        ++modCount;
    }
}
```

afterNodeInsertion()（put方法调用）

在 put 等操作之后执行，当 removeEldestEntry() 方法返回 true 时会移除最晚的节点，也就是链表首部节点 first。

evict 只有在构建 Map 的时候才为 false，在这里为 true。

```
void afterNodeInsertion(boolean evict) { // possibly remove eldest
    LinkedHashMap.Entry<K,V> first;
    if (evict && (first = head) != null && removeEldestEntry(first)) {
        K key = first.key;
        removeNode(hash(key), key, null, false, true);
    }
}
```

removeEldestEntry() 默认为 false，如果需要让它为 true，需要继承 **LinkedHashMap 并且覆盖这个方法的实现，这在实现 LRU 的缓存中特别有用**，通过移除最近最久未使用的节点，从而保证缓存空间足够，并且缓存的数据都是热点数据。

LRU缓存

以下是使用 LinkedHashMap 实现的一个 LRU 缓存：

- 设定最大缓存空间 MAX_ENTRIES 为 3；
- 使用 LinkedHashMap 的构造函数将 accessOrder 设置为 true，开启 LRU 顺序；
- 覆盖 **removeEldestEntry**() 方法实现，**在节点多于 MAX_ENTRIES 就会将最近最久未使用的数据移除**

```
class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private static final int MAX_ENTRIES = 3;

    protected boolean removeEldestEntry(Map.Entry eldest) {
        return size() > MAX_ENTRIES;
    }

    LRUCache() {
        super(MAX_ENTRIES, 0.75f, true);
    }
}
```

```
public static void main(String[] args) {
    LRUCache<Integer, String> cache = new LRUCache<>();
    cache.put(1, "a");
    cache.put(2, "b");
    cache.put(3, "c");
    cache.get(1);
    cache.put(4, "d");
    System.out.println(cache.keySet());
}
```

```
[3, 1, 4]
```

WeakHashMap

WeakHashMap 的 Entry 继承自 WeakReference，被 WeakReference 关联的对象在下一次垃圾回收时会被回收。

WeakHashMap 主要用来**实现缓存**，通过使用 WeakHashMap 来**引用缓存对象**，由 JVM 对这部分缓存进行回收。

```
private static class Entry<K,V> extends WeakReference<Object> implements
Map.Entry<K,V>
```

ConcurrentCache

Tomcat 中的 ConcurrentCache 使用了 WeakHashMap 来实现缓存功能。

ConcurrentCache 采取的是**分代缓存**：

- 经常使用的对象放入 eden 中，eden 使用 ConcurrentHashMap 实现，不用担心会被回收（伊甸园）；
- 不常用的对象放入 longterm，longterm 使用 WeakHashMap 实现，这些老对象会被垃圾收集器回收。
- 当调用 get() 方法时，会先从 eden 区获取，如果没有找到的话再到 longterm 获取，当从 longterm 获取到就把对象放入 eden 中，从而保证经常被访问的节点不容易被回收。
- 当调用 put() 方法时，如果 eden 的大小超过了 size，那么就将 eden 中的所有对象都放入 longterm 中，利用虚拟机回收掉一部分不经常使用的对象。

```
public final class ConcurrentCache<K, V> {

    private final int size;

    private final Map<K, V> eden;

    private final Map<K, V> longterm;

    public ConcurrentCache(int size) {
        this.size = size;
        this.eden = new ConcurrentHashMap<>(size);
        this.longterm = new WeakHashMap<>(size);
    }

    public V get(K k) {
        V v = this.eden.get(k);
        if (v == null) {
            v = this.longterm.get(k);
            if (v != null)
                this.eden.put(k, v);
        }
        return v;
    }

    public void put(K k, V v) {
```

```
            if (this.eden.size() >= size) {
                this.longterm.putAll(this.eden);
                this.eden.clear();
            }
            this.eden.put(k, v);
        }
    }
```

非线程安全和线程安全容器对比图

非线程安全	线程安全(java.util.concurrent，「基于CAS和volatile」)	线程安全(Java 1.1，已过时，基于「Synchronized」)
ArrayList	CopyOnWriteArrayList	Vector
LinkedList	ConcurrentLinkedQueue（非阻塞**Queue**），ConcurrentLinkedDeque（非阻塞**Deque or Stack**）	Stack
HashSet	CopyOnWriteArraySet	-
TreeSet	ConcurrentSkipListSet（**跳表**）	-
HashMap	ConcurrentHashMap	Hashtable
TreeMap	ConcurrentSkipListMap（**跳表**）	-

深拷贝和浅拷贝——Object中的clone()方法

引用拷贝

如下代码只是复制了一个Person对象的引用而已，**并没有开辟空间新建对象**

```
    Person p = new Person(23, "zhang");
    Person p1 = p;
    System.out.println(p);
    System.out.println(p1);
```

对象拷贝（深拷贝和浅拷贝基于对象拷贝）

Object.clone()方法是实实在在的复制了一个对象的，**开辟了新的空间**

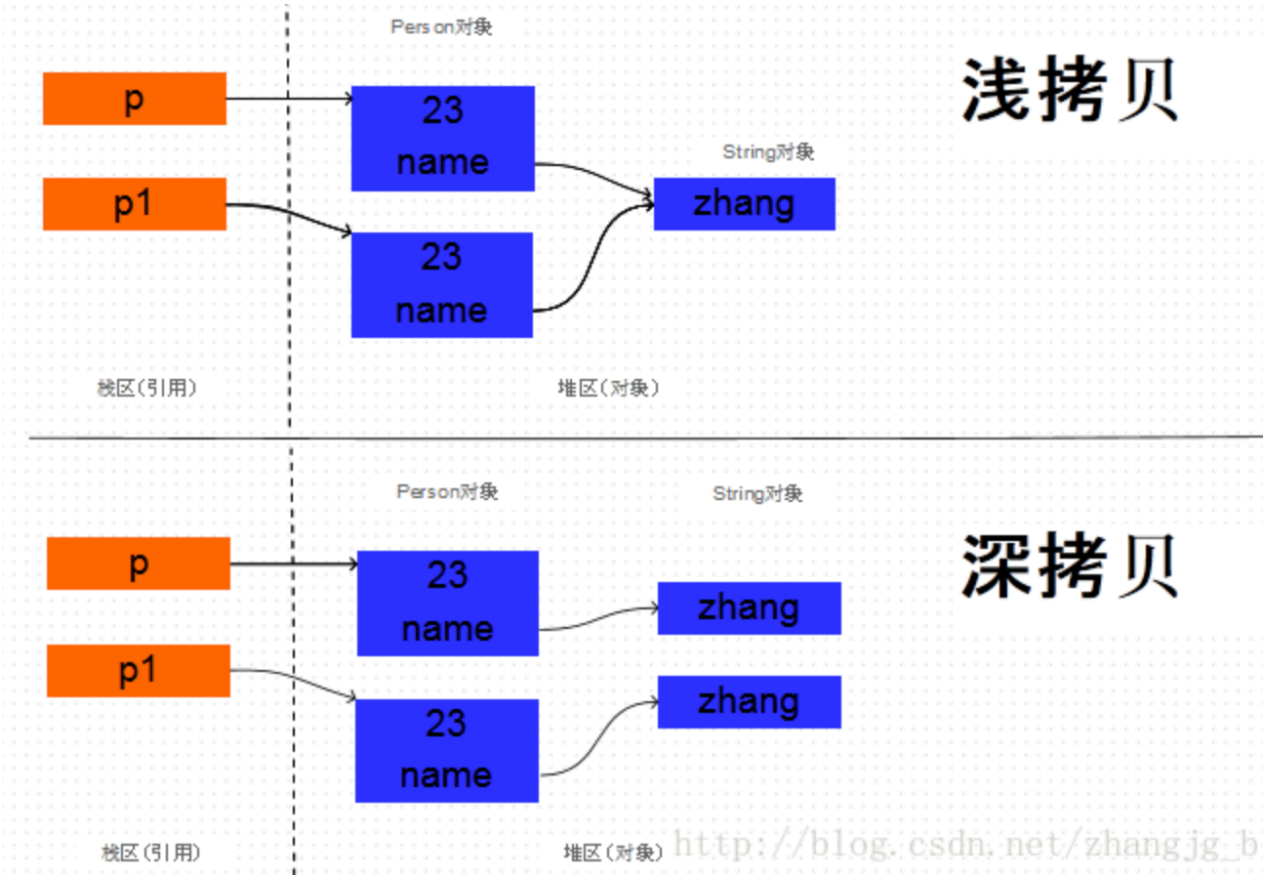
```
    Person p = new Person(23, "zhang");
    Person p1 = (Person) p.clone();
    System.out.println(p);
    System.out.println(p1);
```

浅拷贝——Object.clone()如果用于一个内部有其他对象的对象

```
public class Person implements Cloneable{
```

```
    private int age ;
    private String name;
    public Person(int age, String name) {
        this.age = age;
        this.name = name;
    }
    public Person() {}
    public int getAge() {
        return age;
    }
    public String getName() {
        return name;
    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return (Person)super.clone();
    }
}
```

由于age是基本数据类型， 那么对它的拷贝没有什么疑议，直接将一个4字节的整数值拷贝过来就行。但是name是String类型的， 它只是一个引用， 指向一个真正的String对象，那么**对它的拷贝有两种方式**： 直接将源对象中的name的引用值拷贝给新对象的name字段， 或者是根据原Person对象中的name指向的字符串对象创建一个新的相同的字符串对象，将这个新字符串对象的引用赋给新拷贝的Person对象的name字段。**这两种拷贝方式分别叫做浅拷贝和深拷贝**。



所以，clone方法执行的是浅拷贝（或者说是不彻底的深拷贝）， 在编写程序时要注意这个细节！！！！！

实现深拷贝（注意：很难有彻底的深拷贝）

如果想要深拷贝一个对象， **这个对象必须要实现Cloneable接口**，实现clone方法，**并且在clone方法内部，把该对象引用的其他对象也要clone一份** ， 这就要求这个**被引用的对象必须也要实现Cloneable接口并且实现clone方法。**