# Java基础

## [static关键字](https://www.cnblogs.com/xrq730/p/4820992.html)

被static修饰的变量、被static修饰的方法统一属于类的**静态资源，是类实例之间共享的，换言之，一处变、处处变**。

静态资源属于类，但是是独立于类存在的。从JVM的类加载机制的角度讲，**静态资源是类初始化的时候加载的，而非静态资源是类new（此处不准确）的时候加载的**。

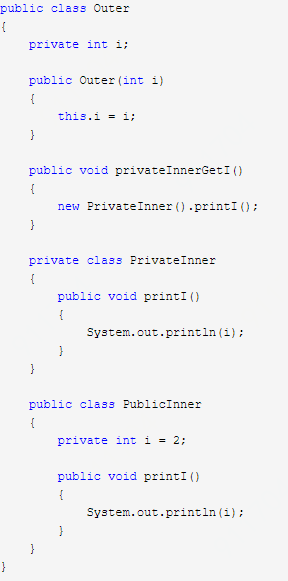
静态块也是static的重要应用之一。也是用于初始化一个类的时候做操作用的，和静态变量、静态方法一样，**静态块里面的代码只执行一次，且只在初始化类的时候执行**。

static一般情况下来说是不可以修饰类的，如果static要修饰一个类，说明这个类是一个静态内部类（注意static只能修饰一个内部类）。

## 内部类

内部类指的就是在一个类的内部再定义一个类。

### **成员内部类**



1、**成员内部类是依附其外部类而存在的，如果要产生一个成员内部类，必须有一个其外部类的实例**

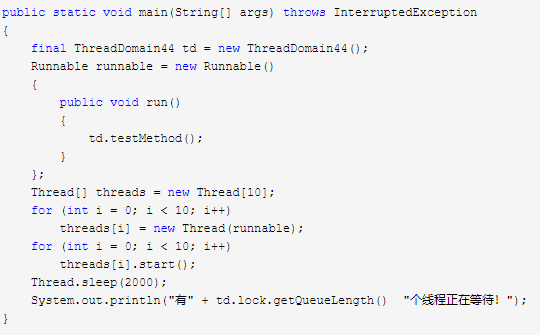
2、成员内部类中没有定义静态方法，不是例子不想写，而是**成员内部类中不可以定义静态方法**

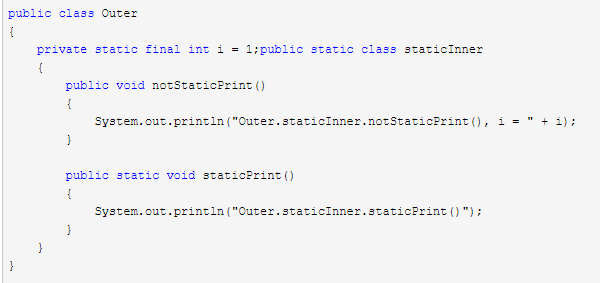
3、成员内部类可以声明为private的，声明为private的成员内部类对外不可见，外部不能调用私有成员内部类的public方法

4、成员内部类可以声明为public的，声明为public的成员内部类对外可见，外部也可以调用共有成员内部类的public方法

5、成员内部类可以访问其外部类的私有属性，如果成员内部类的属性和其外部类的属性重名，则以成员内部类的属性值为准

### **匿名内部类**

静态内部类



1、静态内部类中可以有静态方法，也可以有非静态方法

2、**静态内部类只能访问其外部类的静态成员与静态方法**

3、和普通的类一样，要访问静态内部类的静态方法，可以直接"."出来不需要一个类实例；要访问静态内部类的非静态方法，必须拿到一个静态内部类的实例对象

4、注意一下实例化成员内部类和实例化静态内部类这两种不同的内部类时写法上的差别

（1）成员内部类：外部类.内部类 XXX = 外部类实例.new 内部类();

（2）静态内部类：外部类.内部类 XXX = new 外部类.内部类();

## Final与finally

### Final

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**

**不可变的是变量的引用而非引用指向对象的内容**。

另外，被final修饰的常量，在编译阶段会存入调用类的常量池中（准备阶段）。

### Finally

**至少有两种情况下finally语句是不会被执行的：**

**（1）try语句没有被执行到，如在try语句之前就返回了，这样finally语句就不会执行，这也说明了finally语句被执行的必要而非充分条件是：相应的try语句一定被执行到。**

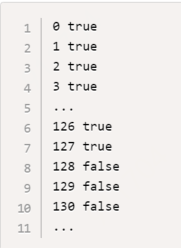
**（2）在try块中有System.exit(0);这样的语句，System.exit(0);是终止Java虚拟机JVM的，连JVM都停止了，所有都结束了，当然finally语句也不会被执行到。**

**finally语句是在try的return语句执行之后，return返回之前执行**。

**finally块中的return语句会覆盖try块中的return返回。**

## int 和 Integer

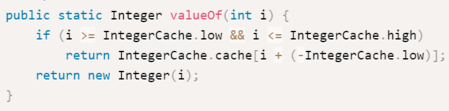




### 装箱

在为Integer类型的变量赋int类型值时，Java会自动将int类型转换为Integer类型，即

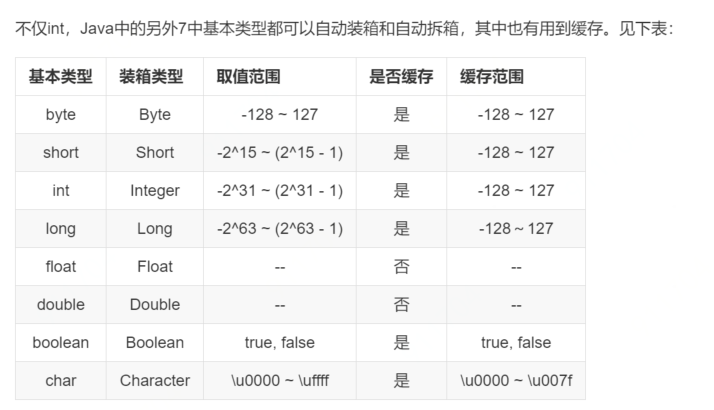
Integer a = Integer.valueOf(1);



-128到127之间的值都是直接从缓存中取出的。看看是怎么实现的：如果int型参数i在IntegerCache.low和IntegerCache.high范围内，则直接由IntegerCache返回；否则new一个新的对象返回。

### 拆箱

1 Integer i = 10; //装箱   
2  int t = i; //拆箱，实际上执行了 int t = i.intValue();



## [接口和抽象类](https://www.cnblogs.com/xrq730/p/4892019.html)

### **接口**

从定义上看，接口是个集合，并不是类。类描述了属性和方法，而接口只包含方法（未实现的方法）。接口和抽象类一样不能被实例化，因为不是类。但是接口可以被实现（使用 implements 关键字）。实现某个接口的类必须在类中实现该接口的全部方法。虽然接口内的方法都是抽象的（和抽象方法很像，没有实现）但是不需要abstract关键字。

1.接口中没有构造方式（因为接口不是类）

2.接口中的方法必须是抽象的（不能实现）

3.接口中除了static、final变量，不能有其他变量

4.接口支持多继承（一个类可以实现多个接口）

### 抽象类

一个抽象类不能实例化，因为“没有包含足够多的信息来描述一个具体的对象”。但终归属于类，所以仍然拥有普通类一样的定义。依然可以在类的实体（直白点就是能在｛｝里面)定义成员变量，成员方法，构造方法等。一个类中含有抽象方法（被abstract修饰），那么这个类必须被声明为抽象类（被abstract修饰）。

### 抽象类和接口的区别

1. 默认的方法实现

抽象类可以有默认的方法实现完全是抽象的。接口根本不存在方法的实现。

2. 方法实现

实现 抽象类使用extends关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。子类使用关键字implements来实现接口。它需要提供接口中所有声明的方法的实现。

3. 抽象类可以有构造器，而接口不能有构造器

4. 修饰符

抽象方法可以有public、protected和default这些修饰符 。接口方法默认修饰符是public。你不可以使用其它修饰符。

5. 抽象类在java语言中所表示的是一种继承关系，一个子类只能存在一个父类，但是可以存在多个接口。

## 序列化与反序列化

平时我们在Java内存中的对象，是无法进行IO操作或者网络通信的，因为在进行IO操作或者网络通信的时候，人家根本不知道内存中的对象是个什么东西，因此必须将对象以某种方式表示出来，即存储对象中的状态。一个Java对象的表示有各种各样的方式，Java本身也提供给了用户一种表示对象的方式，那就是序列化。换句话说，序列化只是表示对象的一种方式而已。

序列化：**将一个对象转换成一串二进制表示的字节数组，通过保存或转移这些字节数据来达到持久化的目的。**

反序列化：**将字节数组重新构造成对象。**

序列化只需要实现java.io.Serializable接口就可以了。序列化的时候有一个serialVersionUID参数，**Java序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的**。在进行反序列化，Java虚拟机会把传过来的字节流中的serialVersionUID和本地相应实体类的serialVersionUID进行比较，如果相同就认为是一致的实体类，可以进行反序列化，否则Java虚拟机会拒绝对这个实体类进行反序列化并抛出异常。

serialVersionUID有两种生成方式：

1、默认的1L

2、根据类名、接口名、成员方法以及属性等来生成一个64位的Hash字段

如果实现java.io.Serializable接口的实体类没有显式定义一个名为serialVersionUID、类型为long的变量时，Java序列化机制会根据编译的.class文件自动生成一个serialVersionUID，如果.class文件没有变化，那么就算编译再多次，serialVersionUID也不会变化。

**1、序列化之后保存的是对象的信息**

**2、被声明为transient的属性不会被序列化，这就是transient关键字的作用**

**3、被声明为static的属性不会被序列化，这个问题可以这么理解，序列化保存的是对象的状态，但是static修饰的变量是属于类的而不是属于对象的，因此序列化的时候不会序列化它**

**利用自定义的writeObject方法和readObject方法，用户可以自己控制序列化和反序列化的过程。**

**复杂序列化情况总结**

虽然Java的序列化能够保证对象状态的持久保存，但是遇到一些对象结构复杂的情况还是比较难处理的，最后对一些复杂的对象情况作一个总结：

1、当父类继承Serializable接口时，所有子类都可以被序列化

2、子类实现了Serializable接口，父类没有，父类中的属性不能序列化（不报错，数据丢失），但是在子类中属性仍能正确序列化

3、如果序列化的属性是对象，则这个对象也必须实现Serializable接口，否则会报错

4、反序列化时，如果对象的属性有修改或删减，则修改的部分属性会丢失，但不会报错

5、反序列化时，如果serialVersionUID被修改，则反序列化时会失败

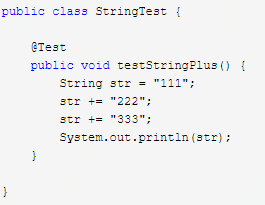
**Transient使用小结**

1）一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。

2）transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。

3）被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。

## StringBuilder和StringBuffer



**编译器每次碰到"+"的时候，会new一个StringBuilder出来，接着调用append方法，在调用toString方法，生成新字符串**。那么，这意味着，如果代码中有很多的"+"，就会每个"+"生成一次StringBuilder，这种方式对内存是一种浪费，效率很不好。

StringBuffer和StringBuilder原理一样，无非是在底层维护了一个char数组，每次append的时候就往char数组里面放字符而已，在最终sb.toString()的时候，用一个new String()方法把char数组里面的内容都转成String，这样，整个过程中只产生了一个StringBuilder对象与一个String对象，非常节省空间。StringBuilder唯一的性能损耗点**在于char数组不够的时候需要进行扩容，扩容需要进行数组拷贝，一定程度上降低了效率**。

StringBuffer和StringBuilder用法一模一样，唯一的区别只是**StringBuffer是线程安全的，它对所有方法都做了同步，StringBuilder是线程非安全的**，所以在不涉及线程安全的场景，比如方法内部，尽量使用StringBuilder，避免同步带来的消耗。另外，StringBuffer和StringBuilder还有一个优化点，上面说了，扩容的时候有性能上的损耗，那么**如果可以估计到要拼接的字符串的长度的话，尽量利用构造函数指定他们的长度。**

## HashCode

1、HashCode的存在主要是为了查找的快捷性，HashCode是用来在散列存储结构中确定对象的存储地址的

2、如果两个对象equals相等，那么这两个对象的HashCode一定也相同

3、如果对象的equals方法被重写，那么对象的HashCode方法也尽量重写

4、如果两个对象的HashCode相同，不代表两个对象就相同，只能说明这两个对象在散列存储结构中，存放于同一个位置

**举个例子**

1、假设内存中有0 1 2 3 4 5 6 7 8这8个位置，如果我有个字段叫做ID，那么我要把这个字段存放在以上8个位置之一，如果不用HashCode而任意存放，那么当查找时就需要到8个位置中去挨个查找

2、使用HashCode则效率会快很多，把ID的HashCode%8，然后把ID存放在取得余数的那个位置，然后每次查找该类的时候都可以通过ID的HashCode%8求余数直接找到存放的位置了

3、如果ID的HashCode%8算出来的位置上本身已经有数据了怎么办？这就取决于算法的实现了，比如ThreadLocal中的做法就是从算出来的位置向后查找第一个为空的位置，放置数据；HashMap的做法就是通过链式结构连起来。

4、如果ID的HashCode%8相等怎么办（这种对应的是第三点说的链式结构的场景）？这时候就需要定义equals了。先通过HashCode%8来判断类在哪一个位置，再通过equals来在这个位置上寻找需要的类。对比两个类的时候也差不多，先通过HashCode比较，假如HashCode相等再判断equals。**如果两个类的HashCode都不相同，那么这两个类必定是不同的**。

**再举个例子**

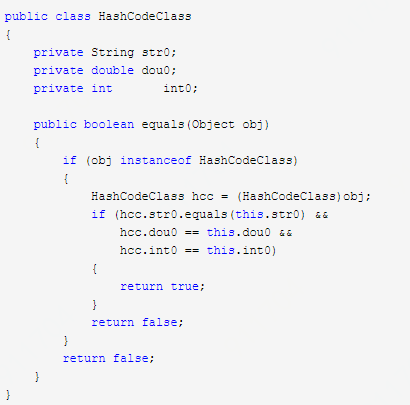
Set里面的元素是不可以重复的，那么如何做到？

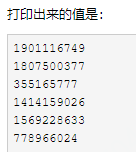
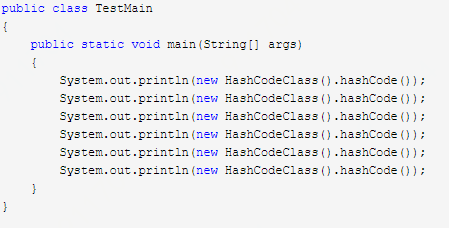
Set是根据equals()方法来判断两个元素是否相等的。比方说Set里面已经有1000个元素了，那么第1001个元素进来的时候，最多可能调用1000次equals方法，如果equals方法写得复杂，对比的东西特别多，那么效率会大大降低。

使用HashCode就不一样了，比方说HashSet，底层是基于HashMap实现的，先通过HashCode取一个模，这样一下子就固定到某个位置了，如果这个位置上没有元素，那么就可以肯定HashSet中必定没有和新添加的元素equals的元素，就可以直接存放了，都不需要比较；如果这个位置上有元素了，逐一比较，比较的时候先比较HashCode，HashCode都不同接下去都不用比了，肯定不一样，HashCode相等，再equals比较，没有相同的元素就存，有相同的元素就不存。

如果原来的Set里面有相同的元素，只要HashCode的生成方式定义得好（不重复），不管Set里面原来有多少元素，只需要执行一次的equals就可以了。这样一来，实际调用equals方法的次数大大降低，提高了效率。

### **为什么重写Object的equals(Object obj)方法尽量要重写Object的hashCode()方法**



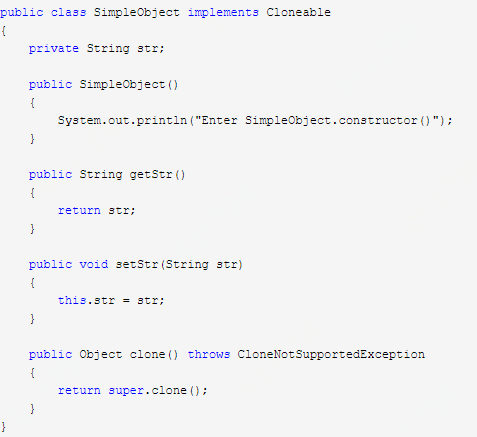


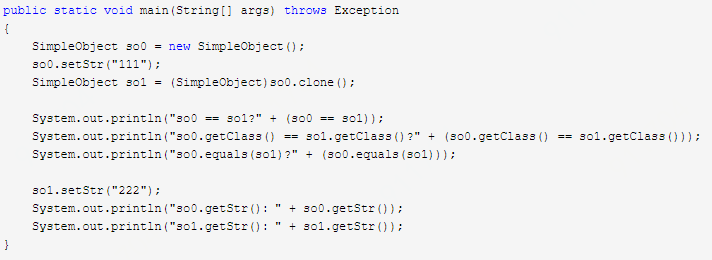
我们希望两个HashCodeClass类equals的前提是两个HashCodeClass的str0、dou0、int0分别相等。

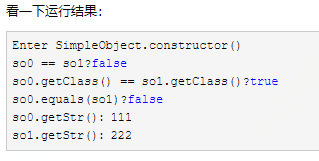
现在我的HashCodeClass都没有赋初值，那么这6个HashCodeClass应该是全部equals的。如果以HashSet为例，HashSet内部的HashMap的table本身的大小是16，那么6个HashCode对16取模分别为13、9、1、2、9、8。第一个放入table[13]的位置、第二个放入table[9]的位置、第三个放入table[1]的位置。。。但是明明是全部equals的6个HashCodeClass，怎么能这么做呢？HashSet本身要求的就是equals的对象不重复，现在6个equals的对象在集合中却有5份（因为有两个计算出来的模都是9）。

## Cloneable

**允许在堆中克隆出一块和原对象一样的对象，并将这个对象的地址赋予新的引用。**







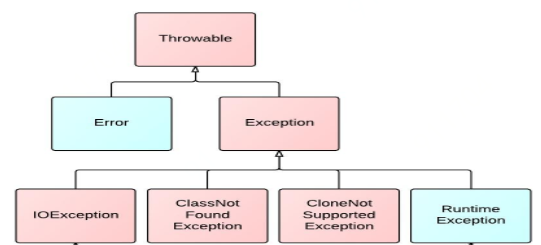
**浅克隆和深克隆**

**浅克隆（shallow clone）和深克隆（deep clone）反映的是，当对象中还有对象的时候，那么：**

**1、浅克隆，即很表层的克隆，如果我们要克隆对象，只克隆它自身以及它所包含的所有对象的引用地址**

**2、深克隆，克隆除自身对象以外的所有对象，包括自身所包含的所有对象实例**

## 异常



异常可分为两大类：java.lang.Exception与java.lang.Error，这两个类均继承自java.lang.Throwable。

1、Error专门用来处理严重影响程序运行的错误，可是通常程序设计者不会涉及程序代码去捕捉这种错误，其原因在于即使捕捉到它，也无法给予适当的处理，比如Java虚拟机出错就属于一种Error

2、Exception包含了一般性的异常，这些异常通常在捕捉到之后便可以做妥善的处理，以确保程序继续运行

RuntimeException即使不编写异常处理的程序代码，依然可以编译成功，而这种异常必须是在程序运行时才有可能发生，比如数组索引值超出了范围。

throw表示抛出一个类的**实例**，注意实例二字，实例意味着throw出去的是一个实例化的异常对象。**throw关键字可以写在任何地方，并不强制必须写在catch块中，运行到throw所在的行，打印异常并立即退出当前方法。**

throws用于方法声明，表示如果方法内的程序代码可能会发生异常，且方法内又没有使用代码的代码块来捕捉这些异常时，则必须在声明方法时一并指明所有可能发生的异常，以便让调用此方法的程序得以做好准备来捕捉异常。

# Java集合框架

### **ArrayList**

#### **优缺点**

ArrayList的优点如下：

1、ArrayList底层以数组实现，是一种**随机访问**模式，再加上它实现了RandomAccess接口，因此查找也就是get的时候非常快

2、ArrayList在顺序添加一个元素的时候非常方便，只是往数组里面添加了一个元素而已

ArrayList的缺点：

1、删除元素的时候，涉及到一次元素复制，如果要复制的元素很多，那么就会比较耗费性能

2、插入元素的时候，涉及到一次元素复制，如果要复制的元素很多，那么就会比较耗费性能

ArrayList扩容是将元素赋值到新数组里。

#### **ArrayList和Vector的区别**

1、Vector是线程安全的，ArrayList是线程非安全的

2、Vector可以指定增长因子，如果该增长因子指定了，那么扩容的时候会每次新的数组大小会在原数组的大小基础上加上增长因子；如果不指定增长因子，那么就给原数组大小\*2。

### **LinkedList**

**LinkedList和ArrayList的对比：**

1、顺序插入速度ArrayList会比较快，因为ArrayList是基于数组实现的，数组是事先new好的，只要往指定位置塞一个数据就好了；LinkedList则不同，每次顺序插入的时候LinkedList将new一个对象出来，如果对象比较大，那么new的时间势必会长一点，再加上一些引用赋值的操作，所以顺序插入LinkedList必然慢于ArrayList

2、基于上一点，因为LinkedList里面不仅维护了待插入的元素，还维护了Entry的前置Entry和后继Entry，如果一个LinkedList中的Entry非常多，那么LinkedList将比ArrayList更耗费一些内存

3、有些说法认为LinkedList做插入和删除更快，这种说法其实是不准确的：

（1）LinkedList做插入、删除的时候，慢在寻址，快在只需要改变前后Entry的引用地址

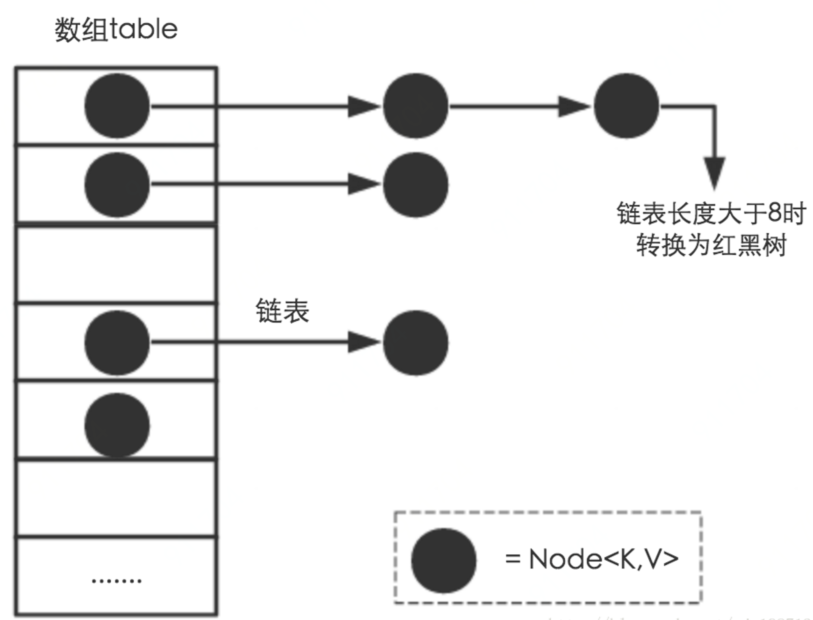
（2）ArrayList做插入、删除的时候，慢在数组元素的批量copy，快在寻址

**所以，如果待插入、删除的元素是在数据结构的前半段尤其是非常靠前的位置的时候，LinkedList的效率将大大快过ArrayList，因为ArrayList将批量copy大量的元素；越往后，对于LinkedList来说，因为它是双向链表，所以在第2个元素后面插入一个数据和在倒数第2个元素后面插入一个元素在效率上基本没有差别，但是ArrayList由于要批量copy的元素越来越少，操作速度必然追上乃至超过LinkedList**。

如果你十分确定你插入、删除的元素是在前半段，那么就使用LinkedList；如果你十分确定你删除、删除的元素在比较靠后的位置，那么可以考虑使用ArrayList。如果你不能确定你要做的插入、删除是在哪儿呢？那还是建议你使用LinkedList吧，因为一来LinkedList整体插入、删除的执行效率比较稳定，没有ArrayList这种越往后越快的情况；二来插入元素的时候，弄得不好ArrayList就要进行一次扩容，**ArrayList底层数组扩容是一个既消耗时间又消耗空间的操作**

### HashMap

HashMap是数组+链表+红黑树实现的。



Java中HashMap采用了链地址法解决hash冲突。

#### 确定哈希桶数组索引位置

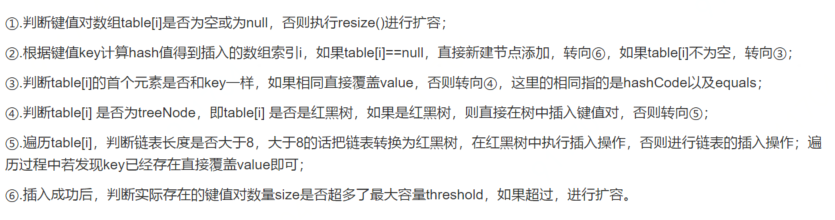




当length总是2的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。

在JDK1.8的实现中，优化了高位运算的算法，通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

#### put方法

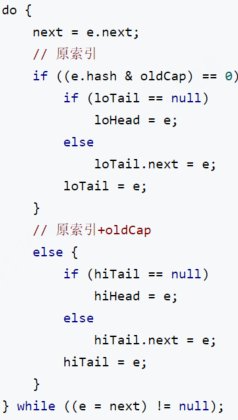


#### 扩容机制

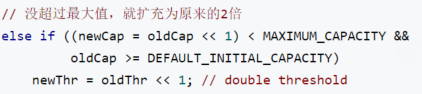
1.7头插法代码。



1.8尾插法代码



1.8定位优化

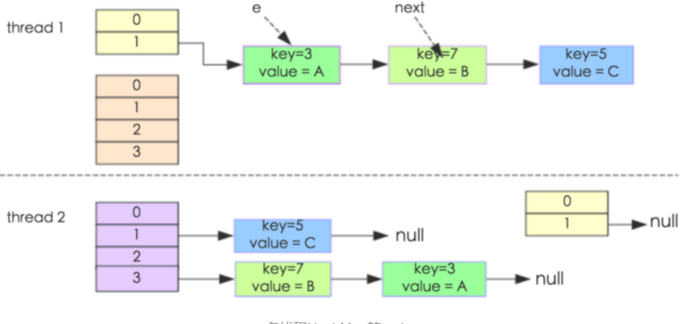


使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。

在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算key在数组中的位置，只需要看key的hash值与新数组长度新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”。

#### 线程安全性





我们假设有两个线程同时需要执行resize操作，我们原来的桶数量为2，记录数为3，需要resize桶到4，原来的记录分别为：[3,A],[7,B],[5,C]，在原来的map里面，我们发现这三个entry都落到了第二个桶里面。

假设线程thread1执行到了transfer方法的Entry next = e.next这一句，然后时间片用完了，此时的e = [3,A], next = [7,B]。线程thread2被调度执行并且顺利完成了resize操作，需要注意的是，此时的[7,B]的next为[3,A]。此时线程thread1重新被调度运行，此时的thread1持有的引用是已经被thread2 resize之后的结果。

线程thread1首先将[3,A]迁移到新的数组上，然后再处理[7,B]，而[7,B]被链接到了[3,A]的后面，处理完[7,B]之后，就需要处理[7,B]的next了啊，而通过thread2的resize之后，[7,B]的next变为了[3,A]，此时，[3,A]和[7,B]形成了环形链表。

#### **HashMap的table为什么是transient的**



看到table用了transient修饰，也就是说table里面的内容全都不会被序列化，因为HashMap是基于HashCode的，HashCode作为Object的方法，是native的：

这意味着的是：**HashCode和底层实现相关，不同的虚拟机可能有不同的HashCode算法**。再进一步说得明白些就是，可能同一个Key在虚拟机A上的HashCode=1，在虚拟机B上的HashCode=2，在虚拟机C上的HashCode=3。

这就有问题了，Java自诞生以来，就以跨平台性作为最大卖点，好了，如果table不被transient修饰，在虚拟机A上可以用的程序到虚拟机B上可以用的程序就不能用了，失去了跨平台性。

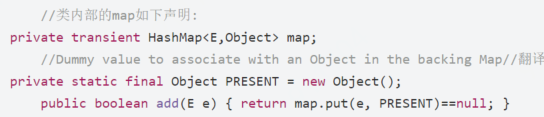
1、Key在虚拟机A上的HashCode=100，连在table[4]上

2、Key在虚拟机B上的HashCode=101，这样，就去table[5]上找Key，明显找不到

整个代码就出问题了。

因此，为了避免这一点，Java采取了重写自己序列化table的方法，在writeObject选择将key和value追加到序列化的文件最后面。而在readObject的时候重构HashMap数据结构

### HashSet



add方法，其实就是向这个map里面put(),HashSet的值是存储在一个HashMap的key里面的，而正好HashMap的key是不能重复的。

若key没有存值，则存进去，返回oldValue；此值初始值是null；  
若key已经有值了，那么add成功也返回oldValue，返回的是上次的值就不为null啦；也就解释了上面的add中 == null 为false。set的add失败。