java基础

八种基本类型

boolean 数组中占据一个1byte而单个则占4byte 取决于虚拟机实现

byte(8bit)

short(16bit)

char(16bit)

int (32)

float(32)

long (64)

double(64)

拆箱装箱

```
Interger x=1;//Interger.valueof(1)
int y=x;//x.intValue()
```

```
int i =50;
int ii=50;
Integer i1 =50;
Integer i2 =50;
Integer i3 = new Integer(50);
Integer i4 = new Integer(50);
Integer i5 = 300;
Integer i6 = 300;
System.out.println(i == ii);// true：两基本类型比较值的大小
System.out.println(i == i1);// true：i1自动拆箱变成基本类型,两基本类型比较值
System.out.println(i == i3);// true： i3自动拆箱变成基本类型,两基本类型比较值
System.out.println(i1 == i2);// true; i1和i3都指向常量池中同一个地址
System.out.println(i1 == i3);// false; 两个不同的对象
System.out.println(i3 == i4);// false； 两个不同的对象
System.out.println(i5 == i6);// false; 自动装箱时，如果值不在-128到127，就会创建一个新的对象
```

==是如何比较的

基本类型和基本类型之间，比较值的大小

基本类型和包装类之间，先拆箱再比较值的大小

包装类之间，比较引用是否为同一个

ps:对于><则

new Integer(1)和Integer.valueof(1)的区别

一个是创建新的对象，一个是先去看缓存有没有，没有就创建。默认缓存池中-128-127

对于其他类型基本都是8个字节的缓存池除了boolean

String类型

为什么用byte和final

final byte val[] 因此不可变 使用byte不用char的作用是节约空间，对于中文等使用标识符进行标记 这时读取时一次性访问两位byte

好处有1hash值可不变，2字符串常量池

String,StringBuilder,StringBuffer比较

1后两者可变，没有final修饰

2 Builder线程不安全

3Buffer 有锁性能差

intern()

在JDK1.6中，字符串常量池是否命中，不命中则在常量池new一个

在JDK1.7中，字符串常量池是否命中，不命中则把自己在堆上的引用加入常量池

字符串拼接操作

直接使用双引号声明出来的String对象会直接存储在常量池中。 除了“abc”+"def"这种会被编译期间优化

常量与常量的拼接结果在常量池中 是编译期优化的结果,原来的两个不会放到常量池中

只要拼接的有一个变量 则结果就在堆中 原理是new StringBuilder(),然后append() 最后toString()

注意！这里toString()虽然是调用了new String()方法但是不会在常量池中创建对象

```
/**
 * ① String s = new String("1")
 * 创建了两个对象
 *       堆空间中一个new对象
 *       字符串常量池中一个字符串常量"1"（注意：此时字符串常量池中已有"1"）
 * ② s.intern()由于字符串常量池中已存在"1"
 * 这里new String()和new Integer()不一样 共同点是都在堆上创建一个，不同是前者常量池中没有的话会还会创建一个
 * s  指向的是堆空间中的对象地址
 * s2 指向的是堆空间中常量池中"1"的地址
 * 所以不相等
 */
String s = new String("1");
s.intern();
String s2 = "1";
System.out.println(s==s2); // jdk1.6 false jdk7/8 false

/*
 * ① String s3 = new String("1") + new String("1")
 * 等价于new String（"11"），但是，常量池中并不生成字符串"11"；
 *
```

```
 * ② s3.intern()
 * 由于此时常量池中并无"11"，所以把s3中记录的对象的地址存入常量池
 * 所以s3 和 s4 指向的都是一个地址
*/
String s3 = new String("1") + new String("1");
s3.intern();
String s4 = "11";
System.out.println(s3==s4); //jdk1.6 false jdk7/8 true
```

4种修饰符

修饰符	当前类	同一包内	子孙类(同一包)	子孙类(不同包)	其他包
`public`	Y	Y	Y	Y	Y
`protected`	Y	Y	Y	Y/N（说明）	N
`default`	Y	Y	Y	N	N
`private`	Y	N	N	N	N

protected讲解https://blog.csdn.net/justloveyou_/article/details/61672133

说明

- **子类与基类不在同一包中**：那么在子类中，子类实例可以访问其从基类继承而来的 protected 方法，而不能访问基类实例的protected方法。

  简单说是不同包继承只能“家族产业”不能继承”父亲资产“

  判断protected方法能否编译需要先找到其到底是哪个基类的方法，判断调用基类所在包和子类包是否相同，然后看

clone()方法及深拷贝浅拷贝

由于protected修饰的原因，如果没有重写的话，那么子类调用的一定是基类Object的clone()方法，这样就会导致，子类与Object不同包，由说明可知，无法访问当前基类的protected方法

重写之后还要实现一个接口Cloneable 这个接口的意义是flag，不是实现其方法

注意浅拷贝和深拷贝

A类中的变量B如果是自己实现的类，那么拷贝A时，浅拷贝将B的引用指向原来的那份，深拷贝的话，重写B实现的类的clone方法，在A的clone方法中主动进行set

static执行顺序

对于静态变量声明和静态代码块两者执行顺序为根据代码顺序执行

```
public class StaticBlockTest{

    static{
         a = 6;
    }

    static int a = 9;

    public static void main(String[] args)
    {
        System.out.println("a的值：" + StaticBlockTest.a);
    }
}
```

无论代码顺序，都是静态属性会按声明赋予初始值，int 为0。再根据代码顺序执行。

父类引用指向子类对象

动态绑定

如果子类新增了父类中不存在的方法，那么这个父类引用是不能调用这个仅在子类中存在的方法中，因为子类对象自动向上转型为了父类对象

如果子类与父类有同名的成员变量和静态变量，那么由于子类自动向上转型为父类对象，此时调用father.a，那么输出的必然是父类的成员变量和静态变量，这里不存在子类覆盖父类同名变量这一说，因为这里本身可以看做是一个父类对象

如果是Son son=new Son(),那么这就是实实在在的一个子类对象，那么son.a和son.b,这样就会覆盖父类的同名变量，输出的是子类的成员变量a和静态成员变量b，如果子类中没有同名变量，那么son.a和son.b调用的是父类的a和b。换句话说，也就是子类可以继承父类的成员变量和静态变量，同时可以覆盖父类的成员变量和静态变量

java是解释还是编译

.java->.class-字节码>01二进制机器码

.java编译为.class

jvm逐行解释执行

同时还有JIT编译器 运行时将热点代码编译直接执行

java泛型和c++模板

java泛型实现 语法糖!

类型擦除 执行时进行强制转换

```
Vector<String> vector = new Vector<String>();
String str = vector.get(0);
变为
Vector vector = new Vector();
String str = (String)vector.get(0);
```

c++ 是多次编译，生成了需要参数类型的函数原型

hashcode和equals

hashcode native 方法

将对象的内存地址转换为整数后返回

equals 直接比较引用是否相等

我们想要的逻辑是按照对象的属性进行比较而不是按照引用进行比较

对于hashmap中hashcode决定索引位置，equals决定元素是否相同

如果只重写hashcode 那么同一索引下会有两个相同元素

如果只重写equals 那么不同索引下会有两个相同元素

而我们想要的结果是一个索引下一个元素

接口和抽象类区别

语法层面上的区别：

- 抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract 方法；java8 default可以有默认实现
- 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的；
- 接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法；
- 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

设计层面上的区别：

- 抽象类是对一种事物的抽象
- 接口是一种行为规范。

多态如何实现?

方法重写和重载原理

重载

静态分派

依据静态类型，将方法的符号引用指向invokevirtual指令参数中

重写

动态分派

调用invokevirtual

过程

1. 找到操作数栈顶的第一个元素所执行的对象的实际类型，记作C。
2. 如果该类的虚方法表中有该方法，就调用否则
3. 从下到上找到同名方法调用

这样就把常量池中的类方法符号引用解析到了不同的直接引用上

JVM采用在类的方法区建立一个虚方法表 （virtual method table）来实现，虚方法表中存储着各个方法的实际入口地址。

如果某个方法在子类中没有被重写，那子类的虚方法表里面的地址入口和父类相同的方法地址入口是一致的，都是指向父类的实现入口。如果子类重写了这个方法，子类方法表中的地址将会替换为指向子类实现版本的入口地址

反射原理

类加载

Reflection和MethodHandle机制本质上都是在模拟方法调用，但是Reflection是在模拟Java代码层次的方 法调用，而MethodHandle是 在模拟字节码层次的方法调用。在MethodHandles.Lookup上的3个方法 findStatic()、findVirtual()、 findSpecial()正是为 了对应于invokestatic、invokevirtual (以及invokeinterface) 和invokespecial这几条字节码指令的执行权限校验行为，而这些底层细节在使用Reflection API时是不需要 关心的。 Reflection中的java.lang.reflect.Method对象远比MethodHandle机制中的java.lang.invoke.MethodHandle对 象所包含的信息来得多。前者是方法在Java端的全面映像，包含了方法的签名、描述符以及方法属性表 中各种属性的Java端表示方式，还包含执行权限等的运行期信息。而后者仅包含执行该方法的相关信息。 用开发人员通俗的话来讲，Reflection是 重量级，而MethodHandle是 轻量级。

io中的设计模式

装饰器和适配器

序列化与serialVersionUID

Serializable接口作用是flag，ObjectOutputStream的writeObject0()会进行判断，除了数组、字符串、枚举外，没有实现Serializable接口的话就会抛出NotSerializableException，而且如果传入的Object重写了ReadObject和WriteObject，那么会反射调用重写后的方法

serialVersionUID作用是在反序列化时先和本地进行对比

静态变量不会被序列化，但是serialVersionUID也是static类型，如何序列化的呢

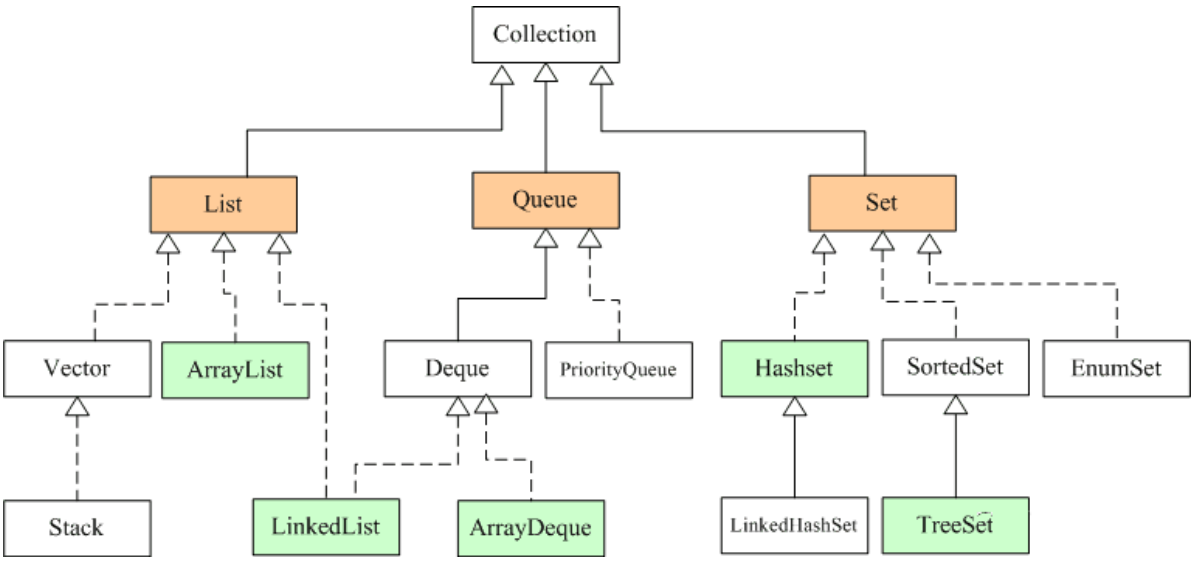
JVM在序列化对象时会自动生成一个serialVersionUID,然后赋值

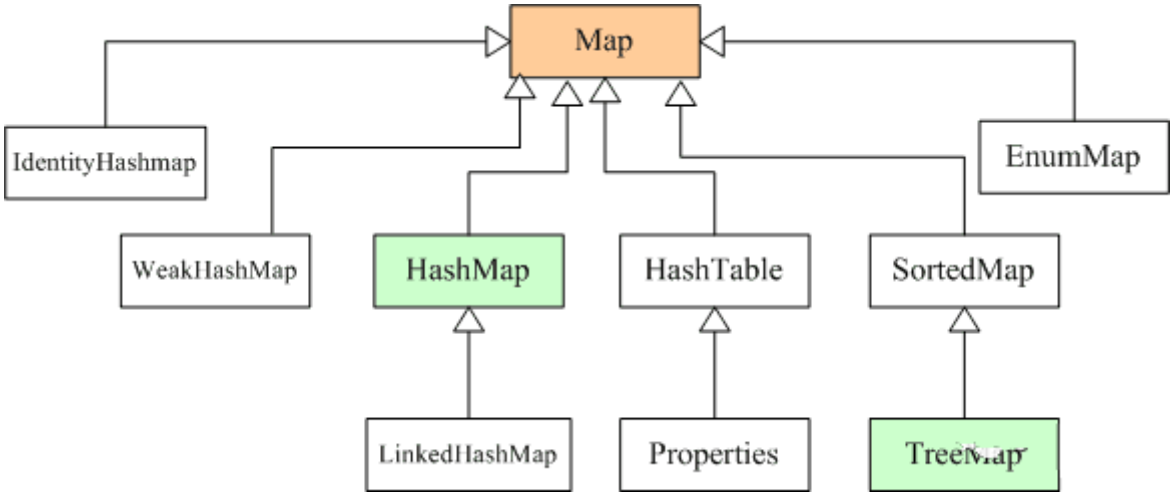
transient 防止变量序列化，反序列化后默认值为0，null

Throwable

- `Exception` :程序本身可以处理的异常，可以通过 `catch` 来进行捕获。`Exception` 又可以分为 Checked Exception (受检查异常，必须处理) 和 Unchecked Exception (不受检查异常，可以不处理)。
- `Error` ： `Error` 属于程序无法处理的错误 ，不建议通过`catch`捕获 。例如Java 虚拟机运行错误（`Virtual MachineError`）、虚拟机内存不够错误(`OutOfMemoryError`)、类定义错误（`NoClassDefFoundError`）等 。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

java容器





Iterator

迭代过程中不能进行原生集合的增加和删除

有一个modcount进行记录，如果不一致就会报错

foreach使用iterator实现同理

RandomAccess

ArrayList实现的一个空接口，标记作用，因为数组使用for循环直接访问较快，而双向链表使用迭代器较快，因为如果使用数组get(i)，相当于O(n2)

则根据是否实现此接口决定遍历的方法

List横向对比

- `Arraylist`： `Object[]` 数组
- `Vector`： `Object[]` 数组 线程安全 一次扩容1倍
- `LinkedList`： 双向链表
- 普通数组 可以基本类型和对象类型 固定大小

ArrayList

内置Object数组，

transient，作用是序列化时不将其显式写入，而自己实现了ReadObject和WriteObject，这样避免将扩容后的部分空数组全部序列化

```
    //默认初始容量
    private static final int DEFAULT_CAPACITY = 10;

    //对象数组，存储具体的元素
    transient Object[] elementData;

    //元素的个数
    private int size;

    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    private static final Object[] EMPTY_ELEMENTDATA = {};
;
```

扩容

无参构造初始大小为0 第一次为max(x,10) 之后为1.5倍，有参直接为x 之后为1.5倍

当使用addAll方法时，新数组扩容1.5倍后容量，仍然不能满足容量要求，会将数组大小直接一次扩容至原数组长度和addAll预添加数组长度之和。

缩容

trimToSize()，需要主动进行操作

CopyOnWriteArrayList区别

写时复制 修改时 加锁复制原来数组到新数组修改后将新数组指向原来引用

可以并发读

缺点是 内存占用以及数据不同步读到旧数据

遍历删除问题

由于删除元素，需要将元素后的数组进行向前拷贝，如果使用原生remove操作，需要将i--，否则会出现两个相邻元素漏删的问题

解决方法

i--

倒序遍历删除

使用iterator.remove 原理也是将下标前移一位

快速失败(fail-fast)和安全失败(fail-safe)了解吗?

有错误，立即停止，和有错误继续执行

在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改，则会抛出Concurrent Modification Exception。

util包下的集合类都是快速失败的 modcount 并发修改以及遍历修改都会导致不等于 expectedmodcount ， 快速失败

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

java.util.concurrent包下的容器都是安全失败

如何使得ArrayList线程安全

vector

Collections.synchronizedList

CopyOnWriteArrayList

Set横向对比

都线程不安全，唯一，实现set接口

- `HashSet`(无序): 基于 `HashMap` 实现的，底层采用 `HashMap` 来保存元素
- `LinkedHashSet`(插入顺序有序): `LinkedHashSet` 是 `HashSet` 的子类，并且其内部是通过 `LinkedHashMap` 来实现的。
- `TreeSet`(有序): 红黑树

List和Set区别

有序 重复 底层是数组和链表

无序 不重复 底层是map val=null

comparable 和 Comparator

前者是在自己定义的类实现该接口，重写此类的所有比较逻辑

后者是在sort函数中写，一般匿名内部类，即插即用

Hashset

为什么val使用的new Object 不用null

因为在移除时Map的put,remove方法都会返回V,而Set方法的add和remove需要返回此次操作是否成功即bool，

通过判断put(k,Object)==null即可返回操作是否成功

Queue横向对比

- `PriorityQueue`：`Object[]` 数组来实现二叉堆
- `ArrayQueue`：`Object[]` 数组 + 双指针

Map横向对比

- `HashMap`： 数组、链表、红黑树
- `LinkedHashMap`： hashmap增加了一条双向链表
- `Hashtable`： 实现Dictionary，数组+链表组成的，数组是 `Hashtable` 的主体，链表则是主要为了解决哈希冲突而存在的
- `TreeMap`： 红黑树

HashMap

重要变量

```
initialCapacity 初始容量（默认16）一定是2的n次方
threshold 阈值 如果大于阈值需要扩容
loadFactor 加载因子（默认0.75）根据容量*负载因子算出阈值
static final int TREEIFY_THRESHOLD = 8; 树化阈值
static final int UNTREEIFY_THRESHOLD = 6;链化阈值
static final int MIN_TREEIFY_CAPACITY = 64; 最小树化容量
```

HashMap 提供了4种构造方法，分别是默认构造方法；可以指定初始容量的构造方法；可以指定初始容量和阈值的构造方法以及基于一个Map 的构造方法。

虽然是构造函数，但是真正的初始化都是在第一次添加操作里面实现的。

负载因子为什么是0.75

空间与时间的折中

越大的话，越容易冲突，减少扩容次数，增加查询次数，时间换空间

越小的话，越容易扩容，增加扩容次数，减少查询次数，空间换时间

为什么要2^n次

因为计算出的hash取模时用的不是%而是直接与&2^n-1

如何实现2的n次方

```
/**
 * 找到大于或等于 cap 的最小2的幂
 */
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    二分思想主逐渐将原数扩大为2^n-1
    先减1的原理是防止将原本是2^n的数扩大为2^(n+1)
    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;
}
```

hash均摊

让高16位参与运算，使得key更加分散

对于null的处理 0

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

null相等如何判断

先判断hash相等，然后判断引用相等，最后判断equals

put操作

判断hash表是否为空 是的话 resize()，

计算桶的位置 hash均摊

判断桶位置为空 插入

桶位置不空

如果是红黑树，查找找(k,v)中k相等的，有的话修改v，没有的话插入

如果是链表，遍历找(k,v)中k相等的，有的话修改v，没有的话插入，插入后判断是否树化

判断是否扩容，++modcount

同理remove 注意判断链化

扩容和树化

如果桶中的结点个数大于TREEIFY_THRESHOLD = 8 树化阈值 ，但是capacity小于MIN_TREEIFY_CAPACITY = 64 最小树化容量

则选择扩容2倍，只有两个条件满足才会树化

扩容移动结点因为hash&newcap相当于往前看一位hash，如果为0，则还在原来的位置，如果为1就在原位置+oldcap

如果红黑树结点个数小于UNTREEIFY_THRESHOLD = 6 链化阈值退化为链表

HashMap和HashTable区别

线程安全

效率

扩容

table 是奇数 初始11或者构造传参，每次2n+1

map 是2的幂次 默认16

table不能插null，map key可以有一个null，val可以有多个null

HashMap和HashSet区别

一个实现map接口，一个实现set接口，但是后者中聚合一个hashmap

set的val是new Object

Hash冲突怎么解决

再散列法 将计算出的hash值再次hash计算直到不冲突

多重哈希

拉链法

公共溢出区 冲突时将冲突数据都放在一块

红黑树

弱平衡二叉树，旋转次数更少

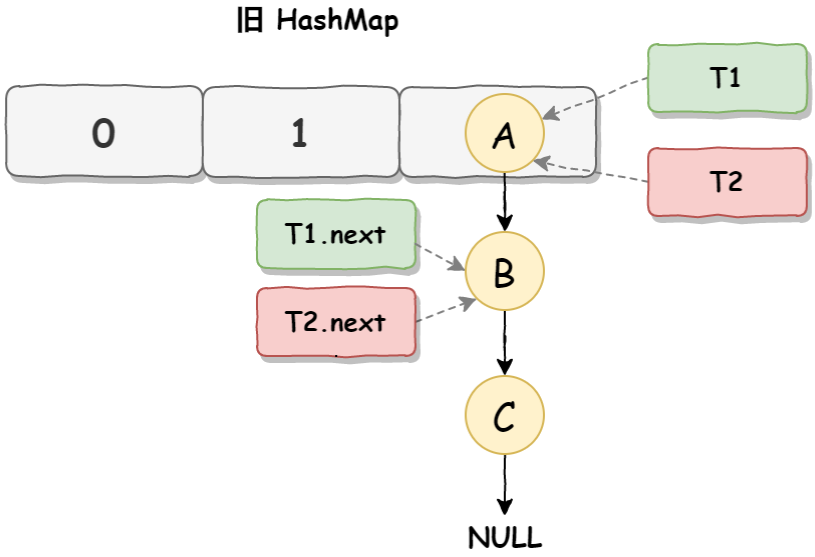
根和叶子节点黑

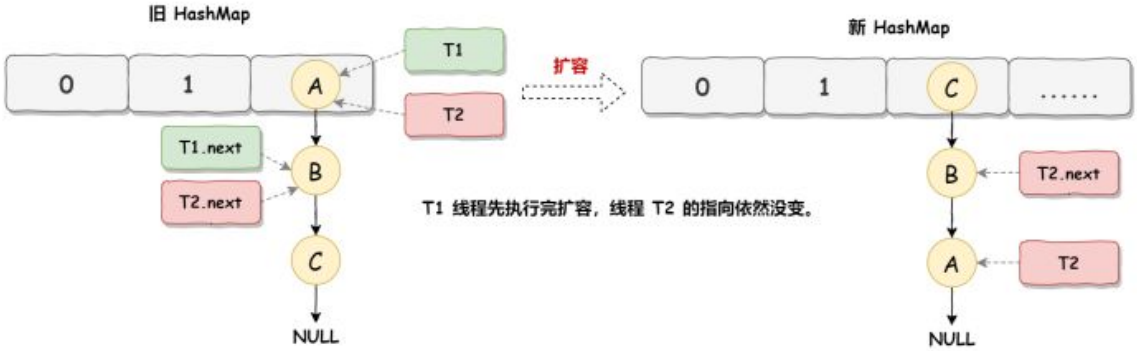
红色结点儿子都黑

任意结点到叶子节点的每条路径的都包含相同数量黑结点

为什么多线程不安全

jdk1.7扩容死链问题





put导致元素覆盖

ConcurrentHashMap

为什么kv不能放null

多线程情况下无法分辨get()获取的null是放进去的还是，本来就没有，

map可以通过containsKey()判断，但是多线程环境下 containsKey和put可能存在时间差

原理

1.7是由 `Segment` 数组结构和 `HashEntry` 数组结构组成，即ConcurrentHashMap 把哈希桶切分成小数组（Segment ），每个小数组有 n 个 HashEntry 组成。

其中，Segment 继承了 ReentrantLock

1.8采用`CAS + synchronized`实现更加低粒度的锁。

将锁的级别控制在了更细粒度的哈希桶元素级别，也就是说只需要锁住这个链表头结点（红黑树的根节点），就不会影响其他的哈希桶元素的读写，大大提高了并发度。

put过程

1. 根据 key 计算出 hash值。
2. 判断是否需要进行初始化。 CAS初始化 其他线程会yield
3. 定位到 Node，拿到首节点 f，判断首节点 f:
    - 如果为 null ， 则通过cas的方式尝试添加。
    - 如果为 `f.hash = MOVED = -1` ，说明其他线程在扩容，参与一起扩容。
    - 如果都不满足 ， synchronized 锁住 f 节点，判断是链表还是红黑树，遍历插入。
4. 当在链表长度达到8的时候，数组扩容或者将链表转换为红黑树。

get过程

不加锁

Node 的元素 val 和指针 next 是用 volatile 修饰的，保证可见性

一致性

如果变化发生在已遍历过的部分，迭代器就不会反映出来，而如果变化发生在未遍历过的部分，迭代器就会发现并反映出来，这就是弱一致性。