HTTP 状态码

200 ok 成功返回状态，对应，GET,PUT,PATCH,DELETE.

201 created 成功创建。

304 not modified HTTP缓存有效。

400 bad request 请求格式错误。

401 unauthorized 未授权。

403 forbidden 鉴权成功，但是该用户没有权限。

404 not found 请求的资源不存在

405 method not allowed 该http方法不被允许。

410 gone 这个url对应的资源现在不可用。

415 unsupported media type 请求类型错误。

422 unprocessable entity 校验错误时用。

429 too many request 请求过多。

GET操作是安全的。所谓安全是指不管进行多少次操作，资源的状态都不会改变。比如我用GET浏览文章，不管浏览多少次，那篇文章还在那，没有变化。当然，你可能说每浏览一次文章，文章的浏览数就加一，这不也改变了资源的状态么？这并不矛盾，因为这个改变不是GET操作引起的，而是用户自己设定的服务端逻辑造成的。

PUT，DELETE操作是幂等的。所谓幂等是指不管进行多少次操作，结果都一样。比如我用PUT修改一篇文章，然后在做同样的操作，每次操作后的结果并没有不同，DELETE也是一样。顺便说一句，因为GET操作是安全的，所以它自然也是幂等的。

POST操作既不是安全的，也不是幂等的，比如常见的POST重复加载问题：当我们多次发出同样的POST请求后，其结果是创建出了若干的资源。

集合

遍历方式



Arraylist 与 LinkedList 异同

1. 是否保证线程安全： ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

2. 底层数据结构： Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向循环链表数据结构；

3. 插入和删除是否受元素位置的影响： ① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行add(E e) 方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element) ）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ② LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。

4. 是否支持快速随机访问： LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index) 方法)。

5. 内存空间占用： ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

**HashMap**:遍历方式：



HashMap的底层实现

JDK1.8 之前 HashMap 底层是 数组和链表 结合在一起使用也就是 链表散列。HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过 (n - 1) & hash 判断当前元素存放的位置（这里的 n 指的时数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

HashMap 和 Hashtable 的区别

线程是否安全： HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

效率： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；

对Null key 和Null value的支持： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。

初始容量大小和每次扩充容量大小的不同 ： ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的tableSizeFor()方法保证，下面给出了源代码）。也就是说 HashMap 总是使用2的幂作为哈希表的大小,后面会介绍到为什么是2的幂次方。

底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制

HashMap长度为什么是2的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648到2147483648，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。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ (n - 1) & hash ”。（n代表数组长度）。这也就解释了 HashMap 的长度为什么是2的幂次方。

**这个算法应该如何设计呢？**

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是2的 n 次方；）。” 并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

HashMap多线程导致死循环问题

在多线程下，进行 put 操作会导致 HashMap 死循环，原因在于 HashMap 的扩容 resize()方法。由于扩容是新建一个数组，复制原数据到数组。由于数组下标挂有链表，所以需要复制链表，但是多线程操作有可能导致环形链表。复制链表过程如下:

以下模拟2个线程同时扩容。假设，当前 HashMap 的空间为2（临界值为1），hashcode 分别为 0 和 1，在散列地址 0 处有元素 A 和 B，这时候要添加元素 C，C 经过 hash 运算，得到散列地址为 1，这时候由于超过了临界值，空间不够，需要调用 resize 方法进行扩容，那么在多线程条件下，会出现条件竞争，模拟过程如下：

线程一：读取到当前的 HashMap 情况，在准备扩容时，线程二介入

线程二：读取 HashMap，进行扩容

线程一：继续执行

这个过程为，先将 A 复制到新的 hash 表中，然后接着复制 B 到链头（A 的前边：B.next=A），本来 B.next=null，到此也就结束了（跟线程二一样的过程），但是，由于线程二扩容的原因，将 B.next=A，所以，这里继续复制A，让 A.next=B，由此，环形链表出现：B.next=A; A.next=B

ConcurrentHashMap和Hashtable的区别

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

底层数据结构： JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；

实现线程安全的方式（重要）： ① 在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。（默认分配16个Segment，比Hashtable效率提高16倍。） 到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6以后 对 synchronized锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② Hashtable(同一把锁) :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

修饰符权限大小

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| public | 类 | 包 | 子类 | 其他包 |
| protected | 类 | 包 | 子类 |  |
| default | 类 | 包 |  |  |
| private | 类 |  |  |  |

重载和重写的区别

重载(Overload)： 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。

重写(Override)： 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 则子类就不能重写该方法

String StringBuffer 和 StringBuilder 的区别是什么

String 类中使用 final 关键字字符数组保存字符串。而 StringBuilder 与StringBuffer 都继承自 AbstractStringBuilder 类但是没有用 final 关键字修饰，所以这两种对象都是可变的。StringBuilder 与 StringBuffer 的构造方法都是调用父类构造方法也就是AbstractStringBuilder 实现的

StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

性能每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用StringBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结：

1. 操作少量的数据 = String
2. 单线程操作字符串缓冲区下操作大量数据 = StringBuilde
3. 多线程操作字符串缓冲区下操作大量数据 = StringBuff

一个静态方法内调用一个非静态成员为什么是非法的

由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非

静态变量，也不可以访问非静态变量成员。

接口和抽象类的区别是什么

1. 接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始

接口方法可以有默认实现），抽象类可以有非抽象的方法

2. 接口中的实例变量默认是 final 类型的，而抽象类中则不一定

3. 一个类可以实现多个接口，但最多只能实现一个抽象类

4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定

5. 接口不能用 new 实例化，但可以声明，但是必须引用一个实现该接口

的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是

行为的抽象，是一种行为的规范。

hashCode()与 equals()关系

1. 如果两个对象相等，则 hashcode 一定也是相同的
2. 两个对象相等,对两个对象分别调用 equals 方法都返回 true
3. 两个对象有相同的 hashcode 值，它们也不一定是相等的
4. 因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖
5. hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数

Java 序列化中如果有些字段不想进行序列化怎么办

对于不想进行序列化的变量，使用 transient 关键字修饰。

transient 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化；

当对象被反序列化时，被 transient 修饰的变量值不会被持久化和恢复。

transient 只能修饰变量，不能修饰类和方法

获取Class方式



内部类与静态内部类

1.静态内部类可以有静态成员(方法，属性)，而非静态内部类则不能有静态成员(方法，属性)。

2.静态内部类只能够访问外部类的静态成员,而非静态内部类则可以访问外部类的所有成员(方法，属性)。

3.实例化一个非静态的内部类的方法：

a.先生成一个外部类对象实例

OutClass oc1 = new OutClass ();

b.通过外部类的对象实例生成内部类对象

OutClass.InnerClass no\_static\_inner = oc1.new InnerClass();

4.实例化一个静态内部类的方法：

a.不依赖于外部类的实例,直接实例化内部类对象

OutClassTest.InnerStaticClass inner = new OutClassTest.InnerStaticClass();

b.调用内部静态类的方法或静态变量,通过类名直接调用

OutClassTest.InnerStaticClass.static\_value

OutClassTest.InnerStaticClass.getMessage()

Linux

常用grep选项是：

-c 只输出匹配行的计数。

-i 不区分大小写（只适用于单字符）。

-h 查询多文件时不显示文件名。

-l 查询多文件时只输出包含匹配字符的文件名。

-n 显示匹配行及行号。

-s 不显示不存在或无匹配文本的错误信息。

-v 显示不包含匹配文本的所有行。

grep '^brad:' /etc /passwd 查找/etc/passwd文件中行首为brad的内容

grep 'bash$' /etc/passwd 查找/etc/passwd文件中行尾为bash的内容

常用netstat参数选项：

　　-t : 指明显示TCP端口

　　-u : 指明显示UDP端口

　　-l : 仅显示监听套接字(所谓套接字就是使应用程序能够读写与收发通讯协议

　　-p : 显示进程标识符和程序名称，每一个套接字/端口都属于一个程序。

　　-n : 不进行DNS轮询，显示IP(可以加速操作)

如何中止一个线程

1. 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。

2. 使用stop方法强行终止线程

3. 使用interrupt方法中断线程。

Thread常用方法

void interrupt() 中断线程。

long getId()返回该线程的标识符。

String getName()返回该线程的名称。

int getPriority() 返回线程的优先级。

void join()等待该线程终止。

void join(long millis)等待该线程终止的时间最长为 millis 毫秒。

void setDaemon(boolean on)将该线程标记为守护线程或用户线程。

void setPriority(int newPriority)更改线程的优先级。

static void sleep(long millis)当前正在执行的线程休眠毫秒数

void start() 使该线程开始执行；Java 虚拟机调用该线程的 run 方法。

static void yield()暂停当前正在执行的线程对象，并执行其他线程。

boolean isAlive()测试线程是否处于活动状态。

static Thread currentThread() 返回对当前正在执行的线程对象的引用。

一个**进程的最大线程数：**

* 32位windows下，一个进程空间4G，内核占2G，留给用户只有2G，一个线程默认栈是1M，所以一个进程最大开2048个线程。当然内存不会完全拿来做线程的栈，所以最大线程数实际值要小于2048，大概2000个。
* 32位Linux下，一个进程空间4G，内核占1G，用户留3G，一个线程默认8M（ulimit –s），所以最多380个左右线程。（ps：ulimit -a 查看电脑的最大进程数，大概7000多个）

**进程最多创建线程数是根据分配给调用栈的大小和操作系统（32和64位）共同决定**

HashMap源码

**底层**：HashMap是Map接口基于哈希表的实现。

**是否允许null**：HashMap允许key和value为null。

**是否有序**：HashMap不保证映射的顺序，特别是它不保证该顺序恒久不变。

**何时rehash**：超出当前允许的最大容量。initial capacity\*load factor就是当前允许的最大元素数目，超过initial capacity\*load factor之后，HashMap就会进行rehashed操作来进行扩容，扩容后的的容量为之前的两倍。

**初始化容量对性能的影响**：不应设置地太小，设置地小虽然可以节省空间，但会频繁地进行rehash操作。rehash会影响性能。总结：小了会增大时间开销（频繁rehash）；大了会增大空间开销（占用了更多空间）和时间开销（影响遍历）。

**加载因子对性能的影响**：加载因子过高虽然减少了空间开销，但同时也增加了查询成本。0.75是个折中的选择。总结：小了会增大时间开销（频繁rehash）；大了会也增大时间开销（影响遍历）。

**是否同步**：HashMap不是同步的。

**迭代器**：迭代器是fast-fail的。

**get( Object key)**

从源码中可以看到，get(E e)可以分为三个步骤：

1. 通过hash(Object key)方法计算key的哈希值hash。
2. 通过getNode( int hash, Object key)方法获取node。
3. 如果node为null，返回null，否则返回node.value。

先来看看哈希值是如何计算的。

**hash( Object key)**

不管增加、删除、查找键值对，定位到哈希桶数组的位置都是很关键的第一步。计算位置的方法如下

（n-1） & hash

其中的n为数组的长度，hash为hash(key)计算得到的值。

从代码中可以看到，计算位置分为三步，第一步，取key的hashCode，第二步，key的hashCode高16位异或低16位，第三步，将第一步和第二部得到的结果进行取模运算。

看到这里有个疑问，**为什么要做异或运算？**

设想一下，如果n很小，假设为16的话，那么n-1即为15（0000 0000 0000 0000 0000 0000 0000 1111），这样的值如果跟hashCode()直接做与操作，实际上只使用了哈希值的后4位。如果当哈希值的高位变化很大，低位变化很小，这样很容易造成碰撞，所以把高低位都参与到计算中，从而解决了这个问题，而且也不会有太大的开销。

看完哈希值是如何计算之后，看看如何通过key和hash获取node。

getNode( int hash, Object key)



**get方法总结**

从源码中可以看到，get(E e)可以分为三个步骤：

1. 通过hash(Object key)方法计算key的哈希值hash。
2. 通过getNode( int hash, Object key)方法获取node。
3. 如果node为null，返回null，否则返回node.value。

hash方法又可分为三步：

1. 取key的hashCode第二步
2. key的hashCode高16位异或低16位
3. 将第一步和第二部得到的结果进行取模运算。

getNode方法又可分为以下几个步骤：

1. 如果哈希表为空，或key对应的桶为空，返回null
2. 如果桶中的第一个节点就和指定参数hash和key匹配上了，返回这个节点。
3. 如果桶中的第一个节点没有匹配上，而且有后续节点
   1. 如果当前的桶采用红黑树，则调用红黑树的get方法去获取节点
   2. 如果当前的桶不采用红黑树，即桶中节点结构为链式结构，遍历链表，直到key匹配
4. 找到节点返回null，否则返回null。

**putVal( int hash, K key, V value, boolean onlyIfAbsent,boolean evict)**



putVal方法可以分为下面的几个步骤：

1. 如果哈希表为空，调用resize()创建一个哈希表。
2. 如果指定参数hash在表中没有对应的桶，即为没有碰撞，直接将键值对插入到哈希表中即可。
3. 如果有碰撞，遍历桶，找到key映射的节点
   1. 桶中的第一个节点就匹配了，将桶中的第一个节点记录起来。
   2. 如果桶中的第一个节点没有匹配，且桶中结构为红黑树，则调用红黑树对应的方法插入键值对。
   3. 如果不是红黑树，那么就肯定是链表。遍历链表，如果找到了key映射的节点，就记录这个节点，退出循环。如果没有找到，在链表尾部插入节点。插入后，如果链的长度大于TREEIFY\_THRESHOLD这个临界值，则使用treeifyBin方法把链表转为红黑树。
4. 如果找到了key映射的节点，且节点不为null
   1. 记录节点的vlaue。
   2. 如果参数onlyIfAbsent为false，或者oldValue为null，替换value，否则不替换。
   3. 返回记录下来的节点的value。
5. 如果没有找到key映射的节点（2、3步中讲了，这种情况会插入到hashMap中），插入节点后size会加1，这时要检查size是否大于临界值threshold，如果大于会使用resize方法进行扩容。

# HashMap中插入null key的过程分析

1.先在table[0]的链表中寻找null key，如果有null key就直接覆盖原来的value，返回原来的value；

2.如果在table[0]中没有找到，就进行头插，但是要先判断是否要扩容，需要就扩容，然后进行头插，此时table[0]就是新插入的null key Entry了。

排序算法

直接插入排序

1.基本思想：直接插入排序的基本思想是：将数组中的所有元素依次跟前面已经排好的元素相比较，如果选择的元素比已排序的元素小，则交换，直到全部元素都比较过为止。

*/\*\*  
 \* 直接插入排序  
 \* <p>  
 \* 1. 从第一个元素开始，该元素可以认为已经被排序  
 \* 2. 取出下一个元素，在已经排序的元素序列中从后向前扫描  
 \* 3. 如果该元素（已排序）大于新元素，将该元素移到下一位置  
 \* 4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置  
 \* 5. 将新元素插入到该位置后  
 \* 6. 重复步骤2~5  
 \*  
 \** ***@param*** *arr 待排序数组  
 \*/*public static void insertionSort(int[] arr) {  
 for (int i = 1; i < arr.length; i++) {  
 int temp = arr[i]; // 取出下一个元素，在已经排序的元素序列中从后向前扫描  
 for (int j = i; j >= 0; j--) {  
 if (j > 0 && arr[j - 1] > temp) {  
 arr[j] = arr[j - 1]; // 如果该元素（已排序）大于取出的元素temp，将该元素移到下一位置  
 System.*out*.println("Temping: " + Arrays.*toString*(arr));  
 } else {  
 // 将新元素插入到该位置后  
 arr[j] = temp;  
 System.*out*.println("Sorting: " + Arrays.*toString*(arr));  
 break;  
 }  
 }  
 }  
}

平均时间复杂度O(n^2),由于直接插入排序每次只移动一个元素的位， 并不会改变值相同的元素之间的排序， 因此它是一种稳定排序。