# 基础

## 三目运算符中的自动拆箱问题

对于条件表达式b?x:y，一个条件表达式从不会既计算x，又计算y。条件运算符是右结合的，也就是说，从右向左分组计算。例如，a?b:c?d:e将按a?b:（c?d:e）执行。

Map<String, Boolean> map = new HashMap<>();

Boolean b = map != null ? map.get("test") : false;

运行以上的代码会报空指针

这段反编译之后

Map map = new HashMap();

Boolean b = Boolean.valueOf(map == null ? false : ((Boolean)map.get("test")).booleanValue());

这是一种自动拆箱的操作。

这段代码为什么会自动拆箱呢？这其实是三目运算符的语法规范。

简单的来说就是：当第二，第三位操作数分别为基本类型和对象时，其中的对象就会拆箱为基本类型进行操作。

结果就是：由于使用了三目运算符，并且第二、第三位操作数分别是基本类型和对象。所以对对象进行拆箱操作，由于该对象为null，所以在拆箱过程中调用null.booleanValue()的时候就报了NPE（空指针）。

如果代码这么写，就不会报错：

Map<String,Boolean> map = new HashMap<String, Boolean>();

Boolean b = (map!=null ? map.get("test") : Boolean.FALSE);

就是保证了三目运算符的第二第三位操作数都为对象类型。这样就不会发生自动拆箱操作

## Java 布尔型的实现原理

总结一下，其实布尔型在 Java 虚拟机是用整型表示的，true 用 1 表示，false 用 0 表示。

## 为什么说使用枚举来实现单例是最好的选择

枚举可解决线程安全问题：枚举在经过javac的编译之后，会被转换成形如public final class T extends Enum的定义。

枚举可避免反序列化破坏单例：在序列化的时候Java仅仅是将枚举对象的name属性输出到结果中，反序列化的时候则是通过java.lang.Enum的valueOf方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的，因此禁用了writeObject、readObject等方法。普通的Java类的反序列化过程中，会通过反射调用类的默认构造函数来初始化对象。所以，即使单例中构造函数是私有的，也会被反射给破坏掉。由于反序列化后的对象是重新new出来的，所以这就破坏了单例。但是，枚举的反序列化并不是通过反射实现的。所以，也就不会发生由于反序列化导致的单例破坏问题。

原文：<https://mp.weixin.qq.com/s/FBd__cIYoTGTMnZGYri7-g>

## 增强for循环的坑

其实JAVA中的增强for循环底层是通过迭代器模式来实现的。

既然增强for循环通过迭代器实现，那么必然有迭代器的特性：Java中有fail-fast机制

Iterator是工作在一个独立的线程中，并且拥有一个 mutex 锁。 Iterator被创建之后会建立一个指向原来对象的单链索引表，当原来的对象数量发生变化时，这个索引表的内容不会同步改变，所以当索引指针往后移动的时候就找不到要迭代的对象，所以按照 fail-fast 原则 Iterator 会马上抛出java.util.ConcurrentModificationException异常。

所以 Iterator 在工作的时候是不允许被迭代的对象被改变的。但你可以使用 Iterator 本身的方法 remove() 来删除对象，Iterator.remove() 方法会在删除当前迭代对象的同时维护索引的一致性。

## stringbuffer与stringbuild底层区别

底层都是使用System.arraycopy实现拷贝，区别只是stringbuffer的方法使用了synchronized

## 单例模式饿汉模式的坑

即便加了synchronized与双层空判断也还存在隐患，这里涉及到了JVM编译器的指令重排。

比如java中简单的一句 instance = new Singleton，会被编译器编译成如下JVM指令：

memory =allocate(); //1：分配对象的内存空间

ctorInstance(memory); //2：初始化对象

instance =memory; //3：设置instance指向刚分配的内存地址

但是这些指令顺序并非一成不变，有可能会经过JVM和CPU的优化，指令重排成下面的顺序：

memory =allocate(); //1：分配对象的内存空间

instance =memory; //3：设置instance指向刚分配的内存地址

ctorInstance(memory); //2：初始化对象

当线程A执行完1,3,时，instance对象还未完成初始化，但已经不再指向null。此时如果线程B抢占到CPU资源，执行 if（instance == null）的结果会是false，从而返回一个没有初始化完成的instance对象。

因此，要给private static Singleton install添加volatile修饰符，volatile关键字不但可以防止指令重排，也可以保证线程访问的变量值是主内存中的最新值。

但是这样其它代码依然可以通过反射获取。

## 代理

正向代理的用途：突破自身IP访问限制，提高访问速度，隐藏客户端真实IP。

反向代理的用途：隐藏服务器真实IP，负载均衡，提高访问速度，提供安全保障。

静态代理和动态代理的主要区别就是代理类创建的时间不同。

静态代理的用途

控制真实对象的访问权限：通过代理对象控制对真实对象的使用权限。

避免创建大对象：通过使用一个代理小对象来代表一个真实的大对象，可以减少系统资源的消耗，对系统进行优化并提高运行速度。

增强真实对象的功能：这个比较简单，通过代理可以在调用真实对象的方法的前后增加额外功能。

Cglib包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。

# 集合

## HashMap与ConcurrentHashMap

**为什么Map桶中个数超过8才转为红黑树**

为什么要转换:因为Map中桶的元素初始化是链表保存的，其查找性能是O(n)，而树结构能将查找性能提升到O(log(n))。当链表长度很小的时候，即使遍历，速度也非常快，但是当链表长度不断变长，肯定会对查询性能有一定的影响，所以才需要转成树。

链表长度达到8就转成红黑树，当长度降到6就转成普通bin。为什么长度为8的时候转成红黑树？说白了就是trade-off，空间和时间的权衡：当hashCode离散性很好的时候，树型bin用到的概率非常小，因为数据均匀分布在每个bin中，几乎不会有bin中链表长度会达到阈值。但是在随机hashCode下，离散性可能会变差，然而JDK又不能阻止用户实现这种不好的hash算法，因此就可能导致不均匀的数据分布。不过理想情况下随机hashCode算法下所有bin中节点的分布频率会遵循泊松分布，我们可以看到，一个bin中链表长度达到8个元素的概率为0.00000006，几乎是不可能事件。所以，之所以选择8，不是拍拍屁股决定的，而是根据概率统计决定的。

## HashMap中1.8与1.7的区别

1.8新增了TREEIFY\_THRESHOLD 用于判断是否需要将链表转换为红黑树的阈值。

HashEntry 修改为 Node。

ConcurrentHashMap 同样也分为 1.7 、1.8 版，两者在实现上略有不同

base1.7

1.7仍然是数组加链表，相比于HashMap，ConcurrentHashMap数组与entry很多核心数据都使用了volatile修饰.原理上来说：ConcurrentHashMap 采用了分段锁技术，其中 Segment 继承于 ReentrantLock。不会像 HashTable 那样不管是 put 还是 get 操作都需要做同步处理。因为volatile无法保证原子性， put 操作时仍然需要加锁处理。首先第一步的时候会尝试获取锁，如果获取失败肯定就有其他线程存在竞争，则利用 scanAndLockForPut() 自旋获取锁。如果重试的次数达到了 MAX\_SCAN\_RETRIES 则改为阻塞锁获取，保证能获取成功。

由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是最新值。ConcurrentHashMap 的 get 方法是非常高效的，因为整个过程都不需要加锁。

base1.8

1.8抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。也将 1.7 中存放数据的 HashEntry 改为 Node，但作用都是相同的。其中的 val next 都用了 volatile 修饰，保证了可见性。

Hashmap中的node是单向链表

## 面试通常的套路是：

谈谈你理解的 HashMap，讲讲其中的 get put 过程。

1.8 做了什么优化？

是线程安全的嘛？

不安全会导致哪些问题？

如何解决？有没有线程安全的并发容器？

ConcurrentHashMap 是如何实现的？ 1.7、1.8 实现有何不同？为什么这么做？

# 加密算法

加密算法可以归结为三大类：哈希算法、对称加密算法、非对称加密算法。

哈希算法：如MD5算法，SHA系列算法

对称加密算法：DES算法，3DES算法，AES算法

非对称加密算法：RSA算法

# 时间处理

**为什么以下代码无法得到美国时间。（在东八区的计算机上）**

System.out.println(Calendar.getInstance(TimeZone.getTimeZone("America/Los\_Angeles")).getTime());

通过查看Date.toString的源码，发现在输出的过程中该方法只会去获取系统的默认时区，只有修改了默认时区才会显示该时区的时间。

但是，通过阅读Calendar的源码，我们可以发现，getInstance方法虽然有一个参数可以传入时区，但是并没有将默认时区设置成传入的时区。

而在Calendar.getInstance.getTime后得到的时间只是一个时间戳，其中未保留任何和时区有关的信息，所以，在输出时，还是显示的是当前系统默认时区的时间。

**什么是冬令时？什么是夏令时？**

夏令时、冬令时的出现，是为了充分利用夏天的日照，所以时钟要往前拨快一小时，冬天再把表往回拨一小时。其中夏令时从3月第二个周日持续到11月第一个周日。

冬令时：北京和洛杉矶时差16小时，北京和纽约时差13小时。 夏令时：北京和洛杉矶时差15小时，北京和纽约时差15小时。

**CET,UTC,GMT,CST几种常见时间的含义和关系？**

CET，欧洲中部时间（英語：Central European Time，CET）是比世界标准时间（UTC）早一个小时的时区名称之一。

UTC，协调世界时，又称世界标准时间或世界协调时间，简称UTC。

GMT，格林尼治标准时间，是指位于英国伦敦郊区的皇家格林尼治天文台的标准时间，因为本初子午线被定义在通过那里的经线。

CST，北京时间，China Standard Time，又名中国标准时间，是中国的标准时间。

CET=UTC/GMT + 1小时、CST=UTC/GMT +8 小时、CST=CET+9

# 数据结构

## 二叉堆

# 排序算法

那么，O(1), O(n), O(logn), O(nlogn)就可以看作既可表示算法复杂度，也可以表示空间复杂度。

大O加上（）的形式，里面其实包裹的是一个函数f(),O（f()）,指明某个算法的耗时/耗空间与数据增长量之间的关系。其中的n代表输入数据的量。

如果ax=N（a>0，且a≠1），那么数x叫做以a为底N的对数，记作x=logaN，读作以a为底N的对数，其中a叫做对数的底数，N叫做真数。





## 桶排序

第一步，就是创建这些桶，确定每一个桶的区间范围：

第二步，遍历原始数列，把元素对号入座放入各个桶中：

第三步，每个桶内部的元素分别排序

第四步，遍历所有的桶，输出所有元素

## 计数排序

计数排序不是基于元素比较，而是利用数组下标来确定元素的正确位置，计数排序适用一定范围的整数排序，在取值范围不是很大的情况下，它的性能甚至快过那些O(nlogn)的排序。

第一步：初始化一个下标数组

第二步：遍历需要排序的数组，在第一个数组中找对应位置，并在对应下标下加1

第三步：按下标数字输出第一个数组

## 镜像二叉树遍历方式

镜像二叉树有两种算法，一种是递归，一种是迭代。

public void mirrorFrom(Node<T> node) {

// 叶子结点

if (node.left == node.right) {

return;

}

// 递归镜像左子树

if (node.left != null)

mirrorFrom(node.left);

// 递归镜像右子树

if (node.right != null)

mirrorFrom(node.right);

// 交换当前节点的左右子树

Node<T> tmp = node.left;

node.left = node.right;

node.right = tmp;

}

迭代如何设计循环？

一个很明显的方法是分层循环，先循环第一层，即根节点，在循环第二层依此类推，循环的终止条件就是后代节点没有了。

public MirrorBinaryTree<T> mirrorByLoop() {

// 空树不必处理

if (root == null) {

return this;

}

// 当前循环需要处理的节点

LinkedList<Node<T>> expandings = new LinkedList<>();

expandings.add(root);

// 没有后台节点就可以终止循环

while (!expandings.isEmpty()) {

// 下一次循环需要处理的节点

// 也就是当前节点的所有儿子节点

LinkedList<Node<T>> nextExpandings = new LinkedList<>();

// 遍历处理当前层的所有节点

for (Node<T> node : expandings) {

// 将后代节点收集起来，留着下一次循环

if (node.left != null) {

nextExpandings.add(node.left);

}

if (node.right != null) {

nextExpandings.add(node.right);

}

// 交换当前节点的左右指针

Node<T> tmp = node.left;

node.left = node.right;

node.right = tmp;

}

// 将后代节点设置为下一轮循环的目标节点

expandings = nextExpandings;

}

return this;

}

原文：<https://mp.weixin.qq.com/s/TVP_-HVhSIUH3_wrz_SfOw>

# 其它算法

## TCP Tahoe算法

## LRU算法的基本思路

使用链表数据结构，将最新查询的数据放到链表的最右端，假设缓存容量已经达到上限，则把最左端的数据删除。java的linkedHashMap已经对哈希链表做了很好的实现。

## Hash算法

Hash就是把任意长度的输入，通过散列算法，变换成固定长度的输出，该输出就是散列值。

两个不同的输入值，根据同一散列函数计算出的散列值相同的现象叫做碰撞。

常见的Hash函数有以下几个：

直接定址法：直接以关键字k或者k加上某个常数（k+c）作为哈希地址。

数字分析法：提取关键字中取值比较均匀的数字作为哈希地址。

除留余数法：用关键字k除以某个不大于哈希表长度m的数p，将所得余数作为哈希表地址。

分段叠加法：按照哈希表地址位数将关键字分成位数相等的几部分，其中最后一部分可以比较短。然后将这几部分相加，舍弃最高进位后的结果就是该关键字的哈希地址。

平方取中法：如果关键字各个部分分布都不均匀的话，可以先求出它的平方值，然后按照需求取中间的几位作为哈希地址。

伪随机数法：采用一个伪随机数当作哈希函数。

衡量一个哈希函数的好坏的重要指标就是发生碰撞的概率以及发生碰撞的解决方案。任何哈希函数基本都无法彻底避免碰撞，常见的解决碰撞的方法有以下几种：

开放定址法：开放定址法就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。

链地址法：将哈希表的每个单元作为链表的头结点，所有哈希地址为i的元素构成一个同义词链表。即发生冲突时就把该关键字链在以该单元为头结点的链表的尾部。

再哈希法：当哈希地址发生冲突用其他的函数计算另一个哈希函数地址，直到冲突不再产生为止。

建立公共溢出区：将哈希表分为基本表和溢出表两部分，发生冲突的元素都放入溢出表中。

在Java中，保存数据有两种比较简单的数据结构：数组和链表。

数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。

在同一个版本的Jdk中，HashMap、HashTable以及ConcurrentHashMap里面的hash方法的实现是不同的。再不同的版本的JDK中（Java7 和 Java8）中也是有区别的。

X % 2^n = X & (2^n - 1)：Java之所有使用位运算(&)来代替取模运算(%)，最主要的考虑就是效率

# 常见问题解决思路

## 大整数数据相加

可以使用数组

## 大整数，删去k个数字后得到最小值

从首位开始比较，如果第一位比第二位数字大，则删除第一位，依次类推。